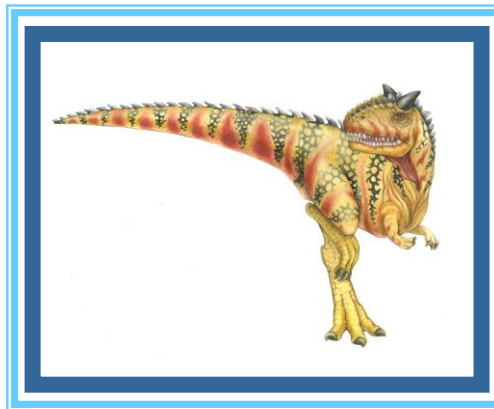


Introduzione





Obiettivi

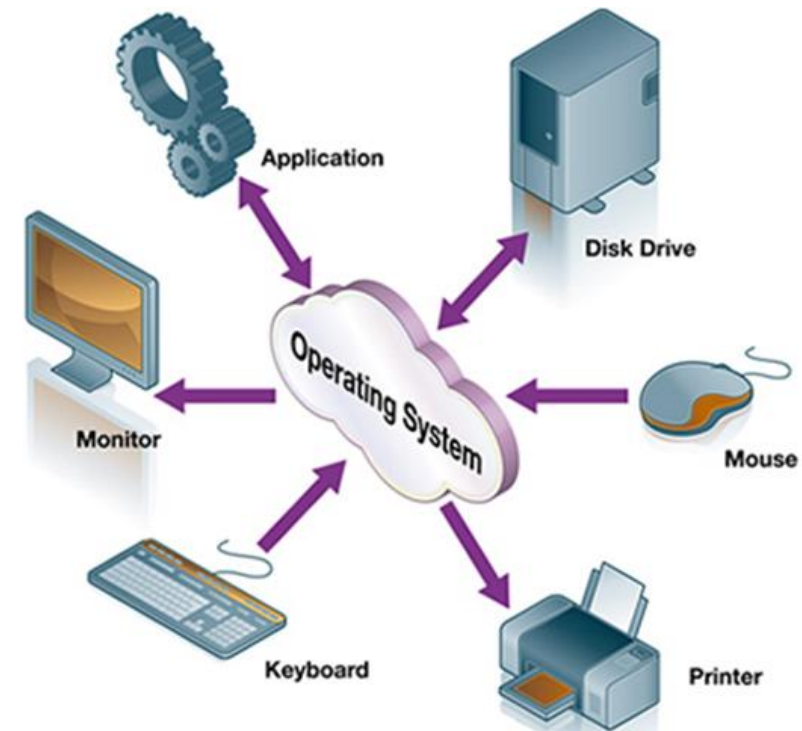
- ❖ Descrivere l'organizzazione generale di un sistema di calcolo con particolare riferimento alle architetture multiprocessore *ogni PC moderno ha più processori alla volta*
- ❖ Analizzare i più importanti componenti di un sistema operativo e il ruolo degli interrupt *S.O. interrupt driver es. printf c'è una syscall*
- ❖ Illustrare la transizione dalla modalità utente alla modalità kernel *l'utente opera le richieste S.O. sono svolto in kernel mode l'utente non può accedere dirett. alle periferiche*
- ❖ Discutere l'utilizzo dei sistemi operativi in ambienti di calcolo ampiamente differenziati
- ❖ Fornire esempi di sistemi operativi free e open-source }





Sommario

- ❖ Che cos'è un sistema operativo
- ❖ Organizzazione del sistema di calcolo
- ❖ Architettura degli elaboratori
- ❖ Struttura del sistema operativo
- ❖ Attività del sistema operativo
 - Gestione dei processi
 - Gestione della memoria
 - Gestione dei file
 - Gestione dei dispositivi di I/O
- ❖ Protezione e sicurezza
- ❖ Virtualizzazione
- ❖ Strutture dati del kernel
- ❖ Ambienti di calcolo
- ❖ Sistemi operativi open-source





Che cos'è un sistema operativo? – 1

è un SW che fa da interf. tra utente e HW macchina

- ❖ Il **Sistema Operativo** perfetto è invisibile all'utente
- ❖ Ogni utente pensa in modo diverso all'utilizzo del computer: *si vogliono delle HW* ha una propria *visione astratta* del sistema di elaborazione, che tiene conto solo di quelle caratteristiche che l'utente ritiene importanti *SO deve mettere a disp. tutte le opzioni per realizzare le visioni di quei users*
- ❖ Il sistema operativo (SO) è un software che agisce da *INTERFACCIA* intermediario tra l'utente e l'hardware del computer
 - Deve fornire servizi e caratteristiche presenti nelle visioni astratte di *tutti* i suoi utenti — servizi che mantiene aggiornati, evolvendosi nel tempo per adeguarsi alle loro esigenze





Che cos'è un sistema operativo? – 2



SO, presenta una
macchina utilizzabile,
in ambiente facile da utilizzare





Che cos'è un sistema operativo? – 3

❖ Scopi del sistema operativo

- Eseguire i programmi, supportando l'utente nel "risolvere problemi"
- ^{int. user-friendly} Rendere agevole l'interfaccia fra l'uomo e la macchina — il sistema di calcolo è "conveniente" da usare
- Gestire in modo efficiente le risorse (hardware/software) del sistema di calcolo *SO è principal. un sistema di controllo. Gestisce risorse e aff. alle risorse. I vari programmi non devono interf. tra loro*
- Prevenire le interferenze fra/con le attività degli utenti
- Dare agli utenti e alle applicazioni una visione "astratta" del sistema di calcolo

utente non sa usare il molo HW, ha bisogno di una interfaccia





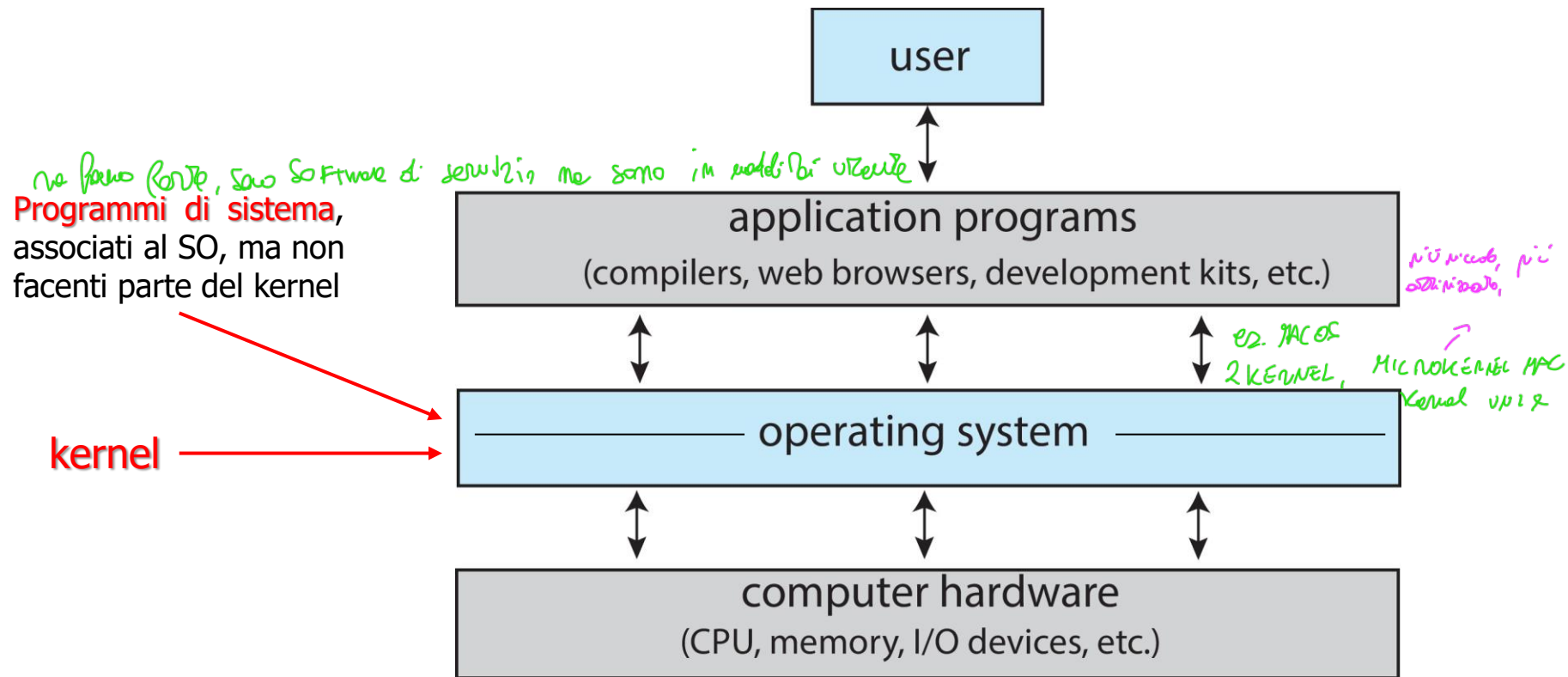
Organizzazione del sistema di calcolo

- ❖ Il sistema di calcolo si suddivide in quattro componenti principali:
 - **Hardware** — fornisce le risorse fondamentali di calcolo
 - ▶ CPU, GPU, memorie, device di I/O
L'hardware come controllori di video
 - **Sistema operativo**
 - ▶ Controlla e coordina l'utilizzo delle risorse hardware da parte dei diversi utenti e delle varie applicazioni
 - **Programmi applicativi** — definiscono le modalità di utilizzo delle risorse del sistema, per risolvere i problemi di calcolo degli utenti
 - ▶ Word processor, compilatori, Web browser, sistemi per basi di dati, fogli di calcolo, giochi
 - **Utenti**
 - ▶ Persone, altri macchinari, altri elaboratori
es. macchina





Le componenti del sistema di calcolo – 1



la gestione delle risorse → privilegia interfaccia

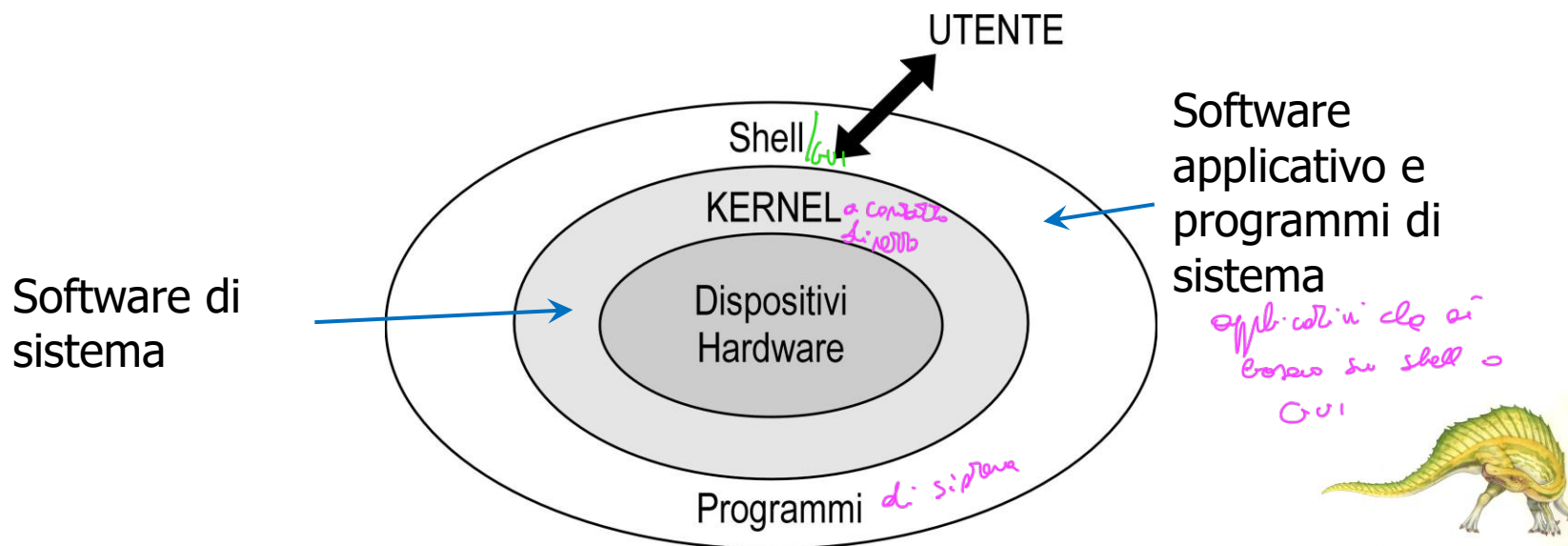




Le componenti del sistema di calcolo – 2

Schea a cipolla

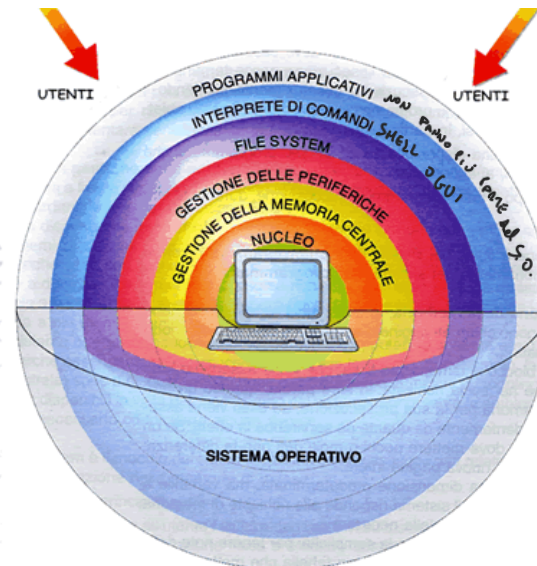
- ❖ La struttura del sistema di calcolo può essere schematizzata anche attraverso il “modello a cipolla”, composto da una serie di *gusci concentrici*, che racchiudono l'hardware, posto al centro, e i diversi strati di software che lo gestiscono/utilizzano
- ❖ I gusci rappresentano programmi, che operano a livelli diversi di interazione uomo–macchina





Le componenti del sistema di calcolo – 3

- ❖ Anche il SO può essere descritto attraverso un modello a gusci concentrici che circondano l'hardware, a contatto diretto con il quale è il **nucleo** o **kernel**, che ingloba le funzioni più elementari eseguibili direttamente dall'hardware stesso
- ❖ All'esterno c'è invece la cosiddetta **shell**, che consente all'utente di accedere alle funzioni più evolute del sistema (gestione dei file, esecuzione dei programmi applicativi, operazioni complesse sulle periferiche)





Cosa ci aspettiamo dal SO? – 1

facile da usare e buone prestazioni

- ❖ Dipende dai punti di vista
- ❖ Gli utenti, dal proprio computer personale, si aspettano un sistema conveniente, **facile da usare** e dotato di **buone prestazioni**
 - Non si preoccupano della **gestione delle risorse**
- ❖ Ma gli *shared computer*, come i **mainframe** o i **server** devono "accontentare" tutti i loro utenti *se ci sono più utenti, gestione delle risorse come di insieme*
- ❖ Gli utenti dei sistemi cosiddetti "dedicati" utilizzano prevalentemente risorse locali, ma anche risorse condivise, accedendo a **server**
 - In questi casi una gestione delle risorse equa e regolamentata è fondamentale per il buon funzionamento del sistema di calcolo

*Shared
different*





Cosa ci aspettiamo dal SO? – 2

Quella più importante, es. efficienza e diversità degli accessi.

- ❖ I device mobili, quali **tablet** e **smartphone**, non dispongono di grandi risorse, che vengono ottimizzate dal SO per fornire usabilità e basso consumo energetico

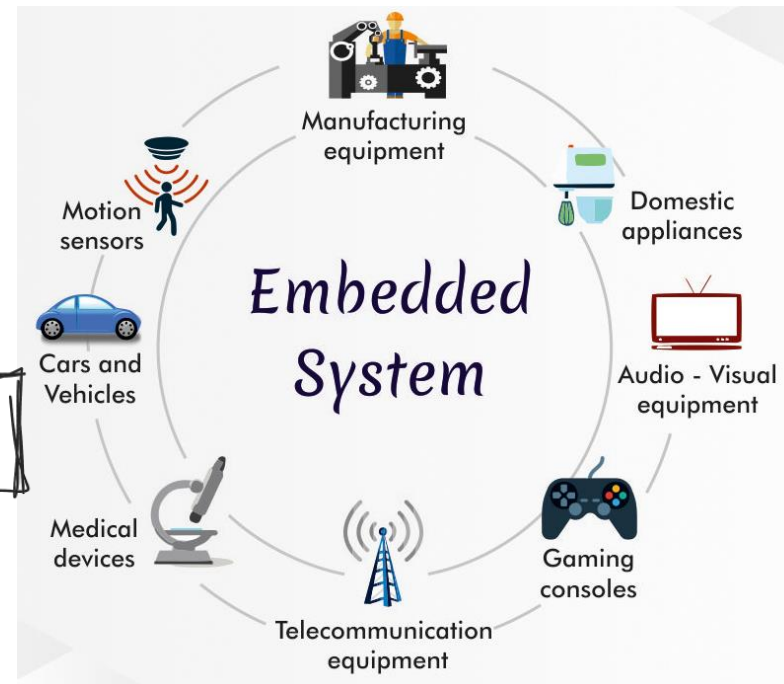
accon diverso e utilizzo diverso; sensori es GPS, giroscopi che non ho sui PC 110 volte diversi

- Interfacce utente tattili e riconoscimento vocale (**Siri**, **Cortana**, **Google Assistant**, **Alexa**)

- ❖ Certi computer, come i **sistemi embedded**, presenti negli elettrodomestici o nelle automobili, hanno SO non dotati di interfaccia utente

- Elaborazione senza l'intervento dell'utente

SO non accessibile all'utente, no interfaccia





Definizione di sistema operativo

❖ Il SO funge da **allocatore di risorse**

- Gestisce tutte le risorse hardware/software (programmi e dati) del sistema di calcolo *per garantire che siano a suff. per tutti gli utenti.*
- Arbitra i conflitti per l'allocazione di risorse in base ad una politica equa ed efficiente

❖ Il SO è un **programma di controllo**

- Controlla l'esecuzione dei programmi utente per prevenire errori e/o uso improprio delle risorse del sistema

proprio: utenti non devono danneggiare S.O. e altri utenti.

es. DOS non aveva la 2 possibilità di esecuzione, es. posso cancellare il kernel S.O. Rip SYSTEM32





Esempio: il SO come gestore risorse – 1

- ❖ Si consideri un ristorante con un capo-cuoco (che dirige la cucina) e i suoi aiutanti, camerieri e clienti:
 - I clienti scelgono un piatto dal menù
 - Un cameriere prende l'ordine e lo consegna al capo-cuoco
 - Il capo-cuoco riceve l'ordine e assegna uno o più aiutanti alla preparazione del piatto *eseguire piatto: effettuare più operazioni*
 - Ogni aiutante si dedicherà alla preparazione di un piatto, il che potrà richiedere più attività diverse
 - Il capo-cuoco supervisiona la preparazione dei piatti e gestisce le risorse (limitate) disponibili





Esempio: il SO come gestore risorse – 2

❖ Il capo-cuoco è il sistema operativo!

- I clienti sono gli utenti
- Le ricette associate ai piatti sono i programmi; gli ingredienti sono i dati
- Il menù ed il cameriere costituiscono l'interfaccia verso il SO (grafica e ^{shell} non)
- Gli aiutanti sono i processi (eventualmente esercitanti più attività – thread) *più processi esecutivi nello stesso processo*
- La cucina è il computer; pentole, fornelli, etc., sono le componenti hardware

dati: ingredienti per le ricette





Esempio: il SO come gestore risorse – 3

❖ Problemi del capo-cuoco:

- Esecuzione fedele delle ricette *program*
- Allocazione/utilizzo efficiente delle risorse esistenti (aiutanti, fornelli, ingredienti, etc.)
- Coordinamento efficiente degli aiutanti *deve coordinare i processi (scheduling, alloc. memoria)*
- Licenziamento degli aiutanti che non si comportano secondo le regole *kill process se tenta di es. in modo perz. memoria*

❖ Problemi del sistema operativo:

- Esecuzione dei programmi utente
- Efficienza nell'uso delle risorse (processori, memoria, dischi, dati, etc.)
- Coordinamento dei processi
- Protezione nell'uso delle risorse (e terminazione dei processi che ne abusano)





Definizione di sistema operativo (cont.)

- ❖ Il termine Sistema Operativo descrive “ruoli diversi” a causa della miriade di hardware design e usi dei sistemi di calcolo
- SO presenti negli elettrodomestici (Internet of Things), nelle auto, nelle navi, nei veicoli spaziali, nelle piattaforme per videogiochi, nei decoder, nei sistemi di controllo industriale...
- Nati quando i computer, originariamente utilizzati solo per scopi militari e per usi governativi (decifrazione di codici, balistica, censimenti), sono diventati *general purpose* e hanno richiesto la gestione delle risorse hardware e il controllo dei programmi





Definizione di sistema operativo (cont.)

- ❖ Non esiste una definizione universalmente accettata di sistema operativo *def. differente dell'hw presente*
- ❖ Tuttavia, "tutto ciò che ti fornisce il rivenditore quando gli chiedi un sistema operativo" è una buona approssimazione
 - Altamente variabile però...
- ❖ "Il sistema operativo — o meglio, il kernel del SO — è l'unico programma sempre in esecuzione quando il computer è acceso" *Sepe in stato di running*
Ci sono anche i demoni
 - Gli altri programmi sono software di sistema o applicativi
 - ▶ **Software di sistema:** software ausiliario e di servizio, abitualmente presente ed utilizzato insieme al SO
 - ▶ **Applicativi:** strumenti di office automation, applicazioni aziendali, ambienti di sviluppo, giochi, etc.





Sistemi operativi per dispositivi mobili

- ❖ I SO per dispositivi mobili non sono costituiti esclusivamente da kernel e programmi di sistema, ma anche dal middleware *Fw, SW cablato nell'HW, difficile da risolvere*
- ❖ Il **middleware** può considerarsi una collezione di ambienti software per gli sviluppatori di applicazioni *AMBIENTI DI SVILUPPO per sviluppare APP, che non è così semplice come fare programmi PC*
- ❖ Sia **iOS** di Apple che **Android** dispongono di un middleware che supporta lo sviluppo di **app** multimediali, grafiche e database-oriented

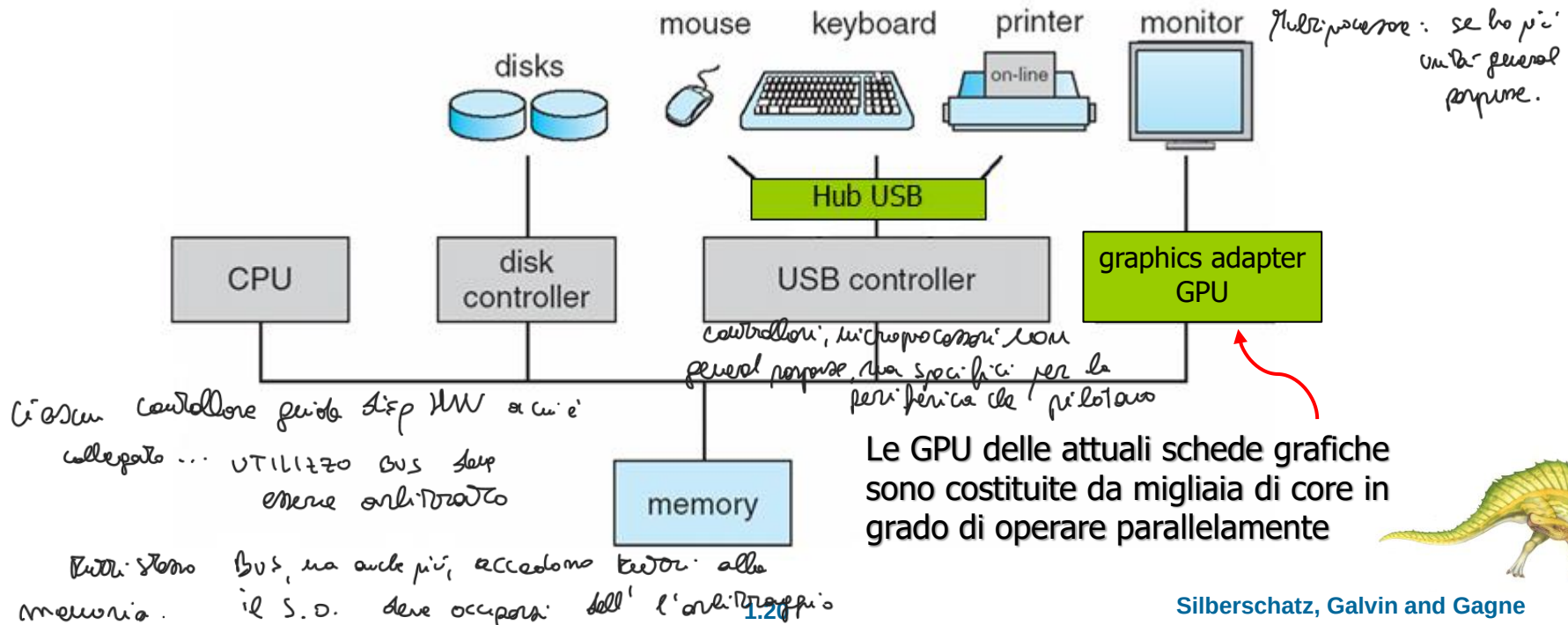




Architettura del sistema di calcolo – 1

- ❖ Una o più CPU e GPU e diversi controllori di device connessi ad un bus comune, che permette l'accesso ad una memoria condivisa *Memorie MASSA e I/O*
- ❖ Ciascun controllore si occupa di un particolare dispositivo fisico e può gestire una o più unità ad esso connesse

e' interesse della parte del sistema di calcolo ambiente usando microprocessore





Architettura del sistema di calcolo – 2

BUFFER, MEMORIA TAMPONE, e registri.

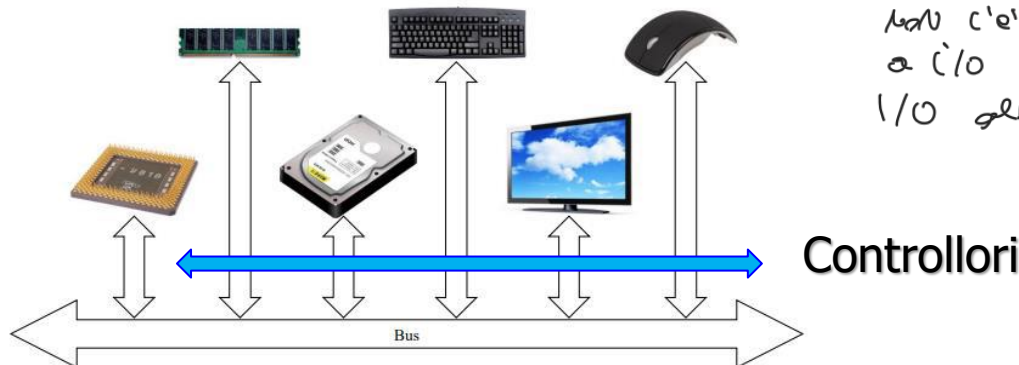
- ❖ Un controllore dispone della propria memoria interna, detta *buffer*, e di un insieme di registri specializzati
- ❖ Per ogni controllore, il SO mette a disposizione un driver del dispositivo, che gestisce la specificità del controllore (e del dispositivo) e funge da interfaccia uniforme con il resto del sistema
- ❖ Esecuzione concorrente nelle CPU e nei controllori di device, che competono per garantirsi cicli di memoria
- ❖ Il controllore della memoria si occupa della sincronizzazione degli accessi

Poco del S.O.
dedicato a quel
dispositivo per
interfacciare
col S.O.

Driver si integrano nel S.O. ma sono un
SISTEMA operativo dedicato a quel disp

della RAM

Caratteri letti dalla memoria, vero
nei buffer
non c'è scambio diretto da memoria
a I/O, ma serve la BUFFER
I/O alla memoria centrale



ogni I/O ha sua memoria (BUFFER), due REGISTRI e suo S/O (driver)



Bios e' sulla ROM (EEPROM)

Firmware e avviamento

Bios carica
in memoria
il programma
di boot

Bios, FW standard

ROM non e' volatile e non
e' cancellabile

- ❖ Il **programma di bootstrap** viene caricato all'accensione del computer, o in fase di reebot, dall'UEFI (per *Unified Extensible Firmware Interface*) o, nei computer più datati, da BIOS

Bios inizializza il computer

Interfaccia grafica
al BIOS e sistema.

- ❖ Sostanzialmente, UEFI ha inglobato, come specifico componente, il BIOS, non lo ha sostituito
- ❖ Il BIOS continua a essere il firmware incorporato nella scheda madre ma l'UEFI, via interfaccia grafica, ne sovrintende l'esecuzione
 - Il BIOS è tipicamente memorizzato nella ROM o nella EEPROM e viene identificato con il termine di **firmware**
 - Lo UEFI può essere considerato come il successore del BIOS e costituisce comunque un'interfaccia posta tra l'hardware ed il SO

Bios aveva un'interfaccia grafica, era una shell, non aveva interfaccia grafica





UEFI

*Posso avere programmi BIOS/UEFI più completi, posso avere più sistemi operativi. Ma
dovranno usare un altro programma che li carica*

- ❖ Infatti, rispetto al BIOS, UEFI abbandona la scarsa interfaccia testuale per abbracciare una più accattivante interfaccia grafica che supporta direttamente l'utilizzo del mouse
- ❖ Mentre il BIOS permetteva la gestione di comandi a 16 bit con la possibilità di indirizzare solo 1 MB di memoria, UEFI può operare in modalità a 32 o 64 bit con la possibilità di indirizzare un maggior quantitativo di memoria RAM in modo tale da assolvere a compiti nettamente più complessi
- ❖ Fornisce, infatti, strumenti per la diagnostica e il ripristino dei dati, servizi di crittografia e funzionalità per la gestione dei consumi
- ❖ UEFI può essere anche del tutto indipendente dall'architettura hardware e può integrare driver per componenti anch'essi indipendenti dalla tipologia di processore in uso





La fase di avviamento – 1

- ❖ Il programma di bootstrap ha lo scopo principale di caricare il bootloader, che è memorizzato su memoria di massa

Carico
BOOTLOADER
in RAM diverso
rispetto ad
altri
programmi

- Il bootloader controlla tutte le componenti del sistema — questa procedura viene chiamata POST (Power-On Self-Test) — e le inizializza
- Procedo al caricamento del kernel del SO e ne lancia l'esecuzione

CARICO IN RAM il BOOTLOADER: Copia del disco in RAM, non ho un filesystem, non ho ancora in esecuzione il S.O.
Conosce esattamente la memoria esatta del disco e delle RAM, senza per rif. ai FS

Controllo di tutte le periferiche HW e loro inizializzate (anche la RAM)

- ❖ Quando si accende un elaboratore, occorre attendere alcuni istanti per poter iniziare a lavorare: durante questa pausa il computer carica il kernel e i programmi di sistema, che garantiscono i servizi fondamentali per l'esecuzione dei programmi utente





La fase di avviamento – 2

- ❖ I programmi di sistema, detti anche *daemon*, vengono eseguiti in background e restano in esecuzione per tutto il tempo in cui è in esecuzione il kernel
- ❖ Rimangono in ascolto di eventi specifici e forniscono, in risposta, dei servizi agli utenti
- ❖ In Linux, il primo processo di sistema è systemd, che funge da gestore di tutti gli altri demoni

programmi del sistema di supporto

sono in ascolto

padre di tutti gli altri processi

avviato una gerarchia di processi

*es. SSH, porta 22, ci serve un daemon in ascolto, pronto a ricevere i comandi
Programmi pronti, in ascolto, aspettano*

• init è padre dei processi figli

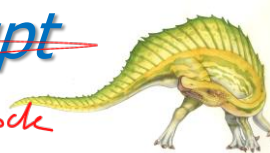




Operatività del sistema di calcolo

- ❖ La CPU e i device di I/O possono eseguire operazioni concorrenti *memoria e dispositivi I/O*
- ❖ Ciascun controllore gestisce un particolare tipo di periferica *es. un HUB con dispositivi tutti dello stesso tipo*
- ❖ Ciascun controllore gestisce un buffer locale ed è responsabile del trasferimento dei dati dalle periferiche ad esso connesse al buffer
- ❖ Ogni controllore ha un driver di dispositivo — *sup. so. applicativo* — facente parte del sistema operativo — *permette di interfacciare col S.O.* ad esso dedicato
- ❖ La CPU sposta i dati dalla memoria principale verso i buffer locali e viceversa
- ❖ L'I/O avviene effettivamente utilizzando i buffer dei controllori come memorie di transito *e non le periferiche vere e proprie*
- ❖ Il controllore informa la CPU quando ha portato a termine l'operazione di I/O, causando un interrupt

- 1) Solo contesto, i suoi registri
li salvo nel process control block
- 2) servo l'interrupt





3) recupero: i registri, lo stato, e il punto

Gli interrupt

• CONTEXT SWITCH
• TODO K=NOT
• EXECUT. ROUTINE SERVIZIO • CONTEXT SWITCH • TODO UTICARE

- ❖ Al verificarsi di un interrupt, il controllo viene trasferito all'opportuna routine di gestione utilizzando il vettore degli interrupt (residente nella memoria bassa), che contiene gli indirizzi di tutte le routine di servizio
 - Si salva inoltre l'indirizzo dell'istruzione "interrotta" – e lo "stato" della CPU – dalla quale dovrà riprendere l'esecuzione al termine della gestione dell'interruzione
- ❖ Durante la gestione di un interrupt, altri eventuali interrupt giunti al sistema possono essere momentaneamente "disabilitati"

routine servizio in memoria stessa
NON MPSCENBIL: errore, paralleli degenerare il tutto
- ❖ Una **trap** (o **eccezione**) è un interrupt generato via software, causato da un errore o da una richiesta utente (effettuata tramite una **system call**)

↳ meno importante
↳ importante ↳ Wrappo una sys call
- ❖ Il SO è **interrupt driven**

contiene la causa.
contiene indirizzi
la causa indica il vettore,
ovvero la routine di servizio, la prima istruzione

interrupt SW

es. printf, sleep, write, syscall, esegui routine





Interrupt timeline

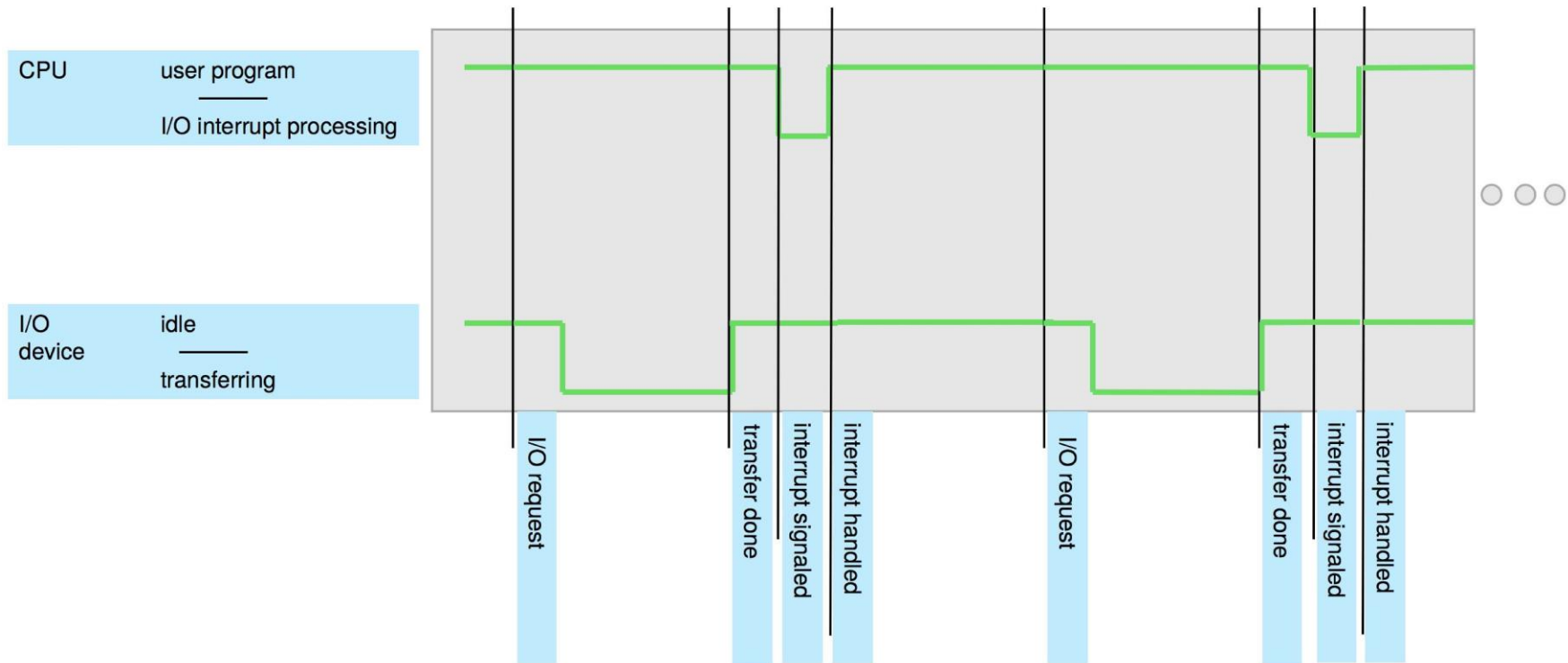


Diagramma temporale delle interruzioni per un programma che invia dati in output





Gli interrupt: implementazione

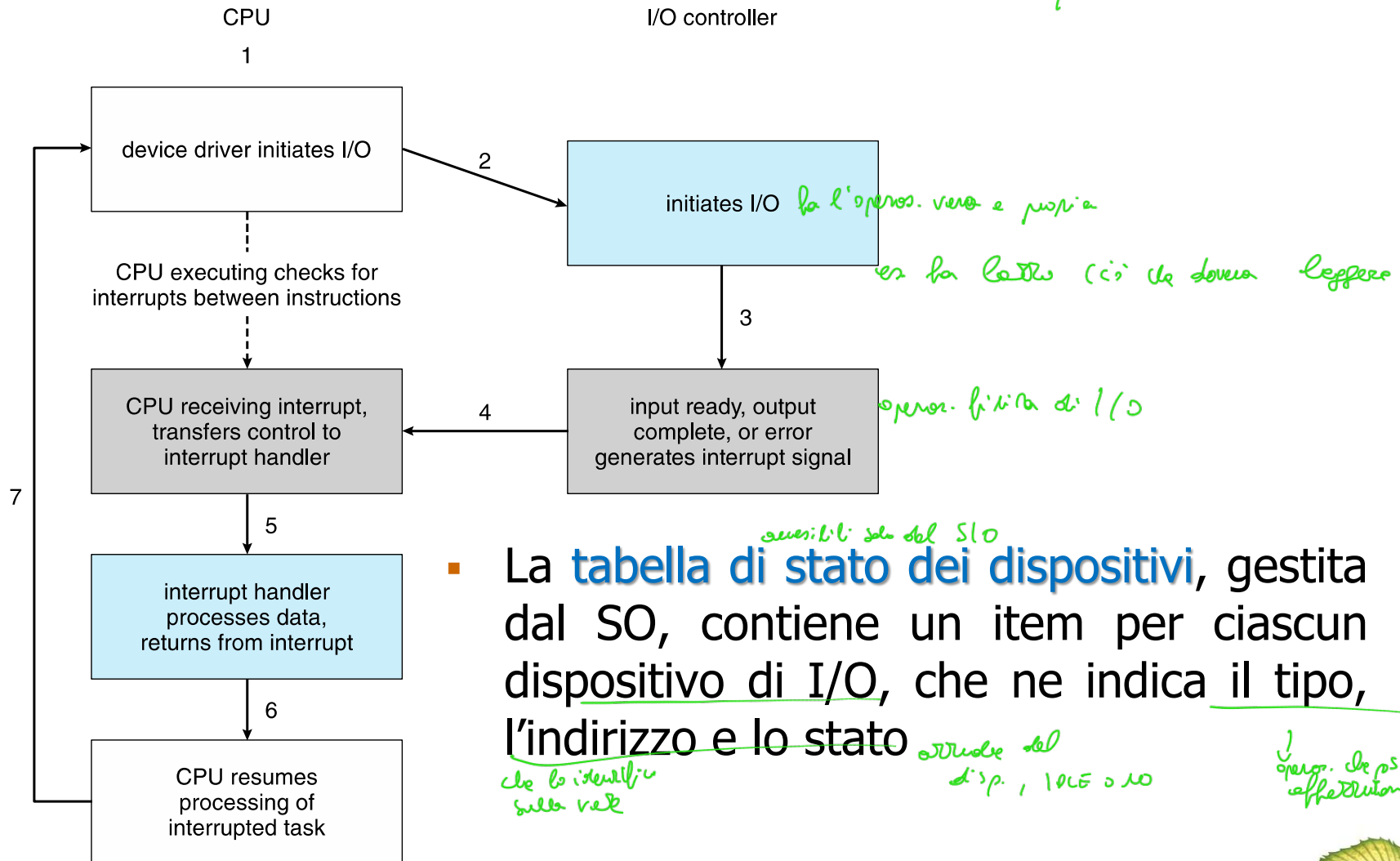
- ❖ L'hardware della CPU dispone di una o più linee (dedicate) di richiesta di interruzione *però alcuni sono mascherabili, altri no*
- ❖ Il gestore delle interruzioni ne rileva la natura, salva le informazioni di stato, invoca la routine di servizio opportuna e, infine, esegue un'istruzione di ritorno e ripristina lo stato della CPU *vedere interrupt -> routine service*
- ❖ Nei sistemi di ultima generazione, hardware dedicato al controllo delle interruzioni *IC?*
 - Efficienza nel recuperare la routine di servizio
 - Possibilità di posticipo della gestione degli interrupt durante un'elaborazione critica *posso posticipare interrupt nei vircoli, cioè mascherarlo*
 - Interruzioni mascherabili, cioè temporaneamente disattivabili (inviate da controllori), e non mascherabili (condizioni di errore) → due *interrupt request line*
 - Interruzioni multilivello, con diverse priorità, per reagire con l'urgenza appropriata *CONTEXT SWITCHING, Contexte salvato nel PC central Board*
↳ devo decidere quale servizio per primo





Ciclo di I/O interrupt driven

processo utente ha richiesto operaz. I/O
CPU I/O controller



- La **tabella di stato dei dispositivi**, gestita dal SO, contiene un item per ciascun dispositivo di I/O, che ne indica il tipo, l'indirizzo e lo stato





Interrupt nei processori Intel

numero di vettore	descrizione
0	errore di divisione
1	eccezione di debug
2	interruzione null
3	breakpoint
4	eccezione di overflow
5	eccezione di range exceeded
6	codice operativo non valido
7	dispositivo non disponibile
8	doppio errore
9	overrun del segmento coprocessore (riservato)
10	task state segment (tss) non valido
11	segmento non presente
12	errore di stack
13	protezione generale
14	errore di pagina
15	(riservato Intel, non utilizzare)
16	errore in virgola mobile
17	controllo dell'allineamento
18	controllo della macchina
19-31	(riservato Intel, non utilizzare)
32-255	interruzioni mascherabili

non mascherabili

Gli eventi da 0 a 31 non sono mascherabili e servono per segnalare condizioni di errore; gli eventi da 32 a 255 vengono utilizzati per le interruzioni provenienti da dispositivi *IO*





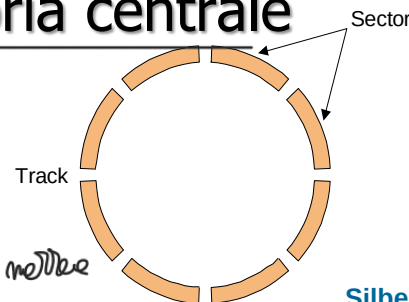
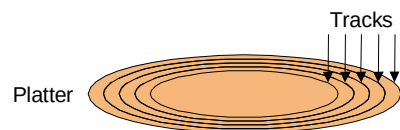
Struttura della memoria

Gerarchia della MEMORIA

cache non gestite
dal S.O.

CPU può accedere direttamente alla RAM

- ❖ **Memoria principale:** l'unico dispositivo di memoria cui la CPU può accedere direttamente (volatile, accesso random)
- ❖ **Memorie di massa:** "estensioni" della memoria principale; forniscono notevole capacità di memorizzazione non volatile
 - ⇒ **"Dischi" a stato solido** – *Non volatile memory* (NVM), memorie elettroniche non volatili, prive di organi meccanici
 - Più veloci dei vecchi hard disk
 - Estremamente diffuse, diverse tecnologie
 - ⇒ **Dischi magnetici** – "piatti" rigidi di metallo o in fibra di vetro, ricoperti di materiale magnetico
 - La superficie del disco è logicamente suddivisa in **tracce**, a loro volta suddivise in **settori**
- ❖ In entrambi i casi: il **controllore del disco** determina l'interazione logica fra device e memoria centrale



Memoria terziaria, cache solo driver e un controller, posso mettere
media di backup





Gerarchia dei dispositivi di memoria – 1

- ❖ I dispositivi di memoria sono organizzati gerarchicamente in base a:
 - Velocità di accesso
 - Costo (al byte)
 - Volatilità
- ❖ *Caching* – produzione di una copia dell'informazione in un dispositivo di memoria più veloce (situato più in alto nella gerarchia delle memorie); la memoria principale costituisce una *cache* per le memorie secondarie

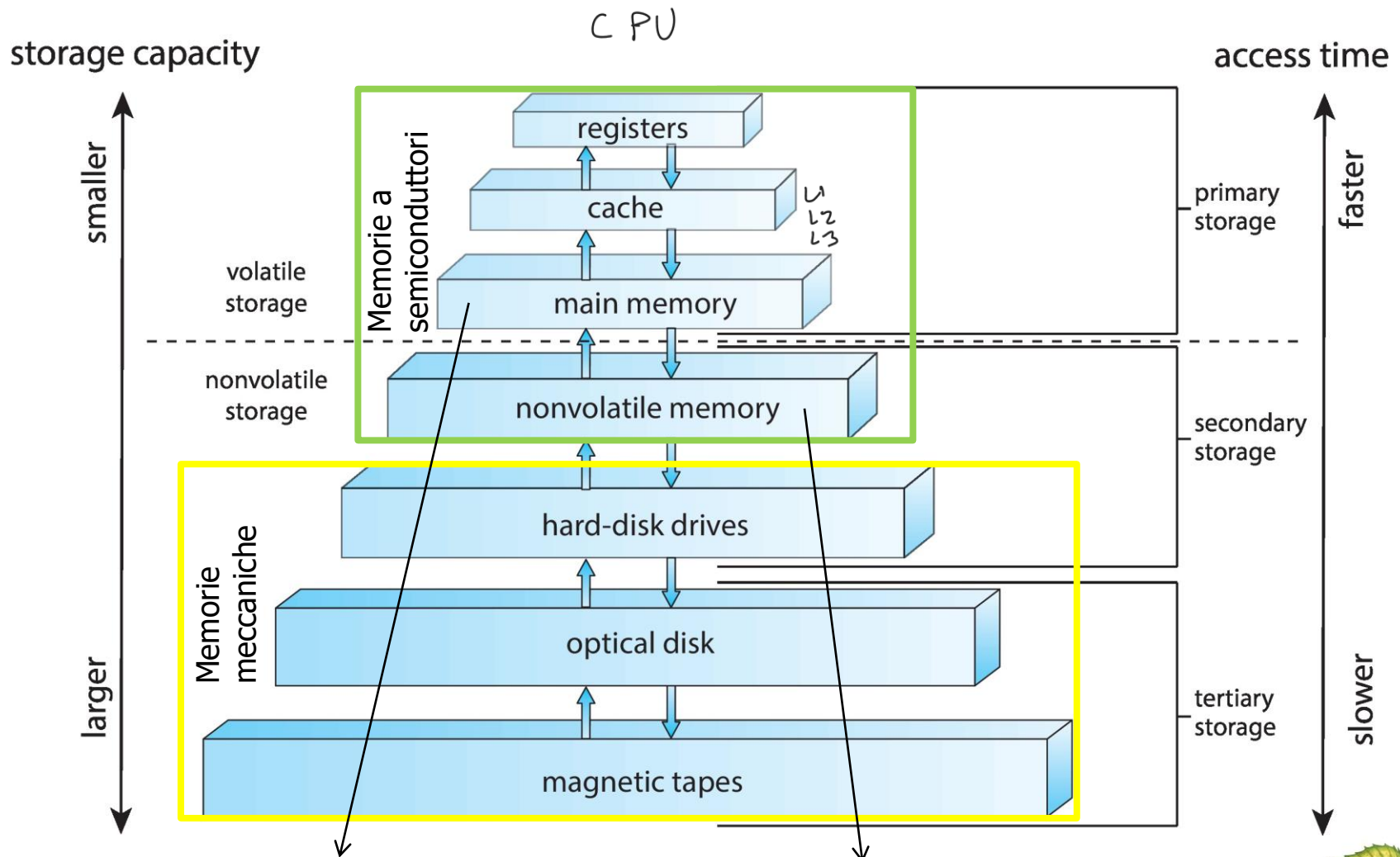
A tutti i livelli di gerarchia

es. cache principale, va da 1MB a 1GB



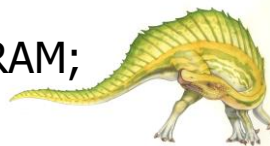


Gerarchia dei dispositivi di memoria – 2



Memoria DRAM (Dynamic Random Access Memory)

Memoria flash (NAND): Più lenta della DRAM; non necessita di alimentazione esterna





Caching – 1

- ❖ Principio informatore fondamentale dei moderni sistemi di calcolo, attuato a vari livelli (hardware, SO, software)
*forse MT, per questo. molto molto più veloce
migliaia di memorie di diversi ordini di grandezza*
- ❖ L'informazione in uso viene copiata temporaneamente da un dispositivo di memoria più lento ad uno più veloce
- ❖ Il dispositivo più veloce, che funge da cache, viene controllato per verificare se l'informazione d'interesse è già presente, nel qual caso...
 - ...l'informazione viene reperita direttamente dalla cache
 - Altrimenti, i dati vengono preventivamente copiati nella cache e quindi acceduti





Caching – 2

- ❖ La cache ha dimensioni inferiori rispetto al supporto di memoria dal quale si effettua il caching
- La gestione della cache, data la capacità limitata dei dispositivi di caching, rappresenta un problema di progettazione fondamentale
- Un'appropriate selezione delle dimensioni e dei criteri di aggiornamento della cache può garantire che una percentuale variabile dall'80% al 99% degli accessi si limiti alla cache, con un notevole incremento delle prestazioni del sistema





Struttura dell'I/O – 1

- ❖ Una percentuale cospicua del codice del SO è dedicata alla gestione dell'I/O, perché:

*Sono molto variegati, es. tutto lo spazio
delle I/O, e soprattutto, sono REGOLAMENTATI*

*DRIVER DISPOSITIVI
e interfaccia come
dei dispositivi
verso il centro
del SO*

■ l'interazione con i dispositivi di I/O è fondamentale per il progetto di un sistema affidabile ed efficiente

- vi è una grande variabilità nei dispositivi di ingresso/uscita

- ❖ Ciascun controllore si occupa di un particolare tipo di dispositivo (può gestire una o più unità ad esso connesse)
- ❖ Il driver del dispositivo è il “sistema operativo” del controllore





Struttura dell'I/O – 2

I/O Sincrono o Asincrono

- ❖ In seguito ad una richiesta di I/O, da parte di un processo utente, si verifica un'interruzione
- ❖ Dopo l'inizio dell'operazione di I/O, il flusso di esecuzione può seguire due percorsi distinti
 - Nel caso più semplice, si restituisce il controllo al processo utente solo dopo il completamento dell'operazione di I/O (*I/O sincrono*) *più semplice*
 - ▶ La CPU è mantenuta in stato idle da una *wait*
 - ▶ Ad ogni istante, si può avere una sola richiesta di I/O in esecuzione
 - Altrimenti, si può prevedere la restituzione immediata del controllo al (ad altro) processo utente: in questo modo l'I/O può proseguire mentre il sistema esegue altre operazioni (*I/O asincrono*) *continua ad essere eseguita*

se la gestisce il controller





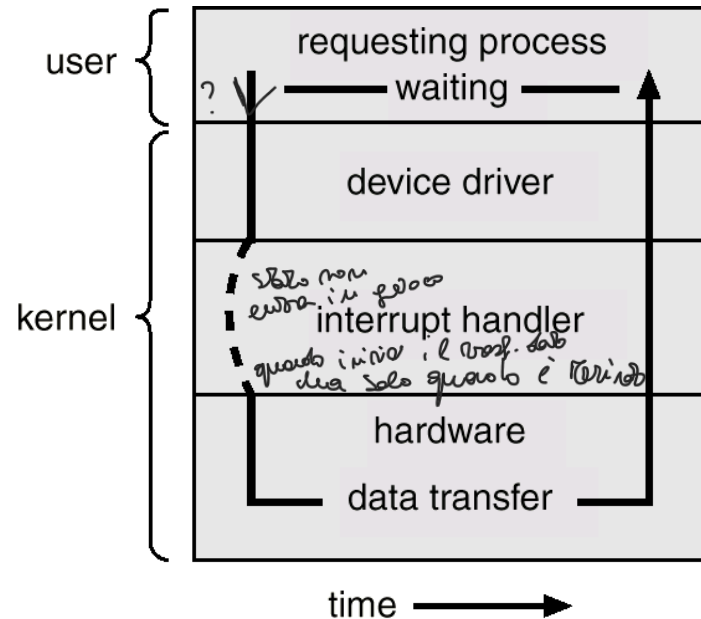
Sincronia, diversi dai veri effetti.

I/O sincrono e asincrono

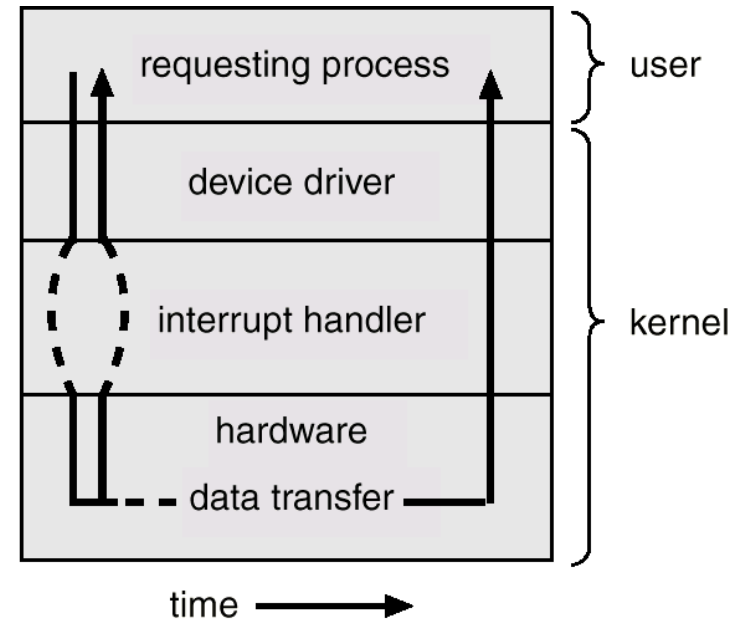
Il processo aspetta o no l'operazione di I/O

*interrupt sincroni: chi fa questo l'interrupt e il processo non
struttura: interruzione imposta dall'esterno*

Sincrono



Asincrono





Dispositivi di I/O, centri tra loro.

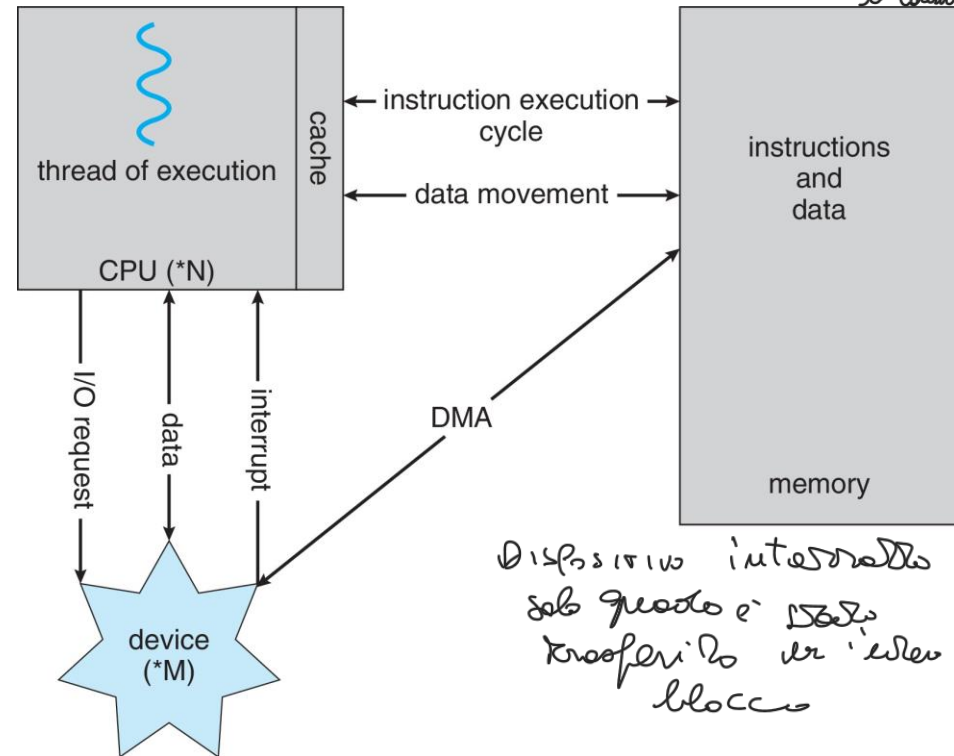
Direct Memory Access (DMA)

Tipo di trasferimento per

CPU non interviene di continuo

Dispositivi di I/O veloci quasi come la memoria

- ❖ Utilizzato per dispositivi di I/O ad alte prestazioni, in grado di trasmettere informazioni a velocità prossime all'accesso alla memoria
- ❖ Il controller del dispositivo trasferisce blocchi di dati dal buffer locale direttamente nella memoria principale, senza l'intervento della CPU
- ❖ Viene generato un solo interrupt per blocco, anziché un interrupt per byte



Dispositivo interviene solo quando è necessario trasferire un intero blocco

CPU fa quello che deve fare, controller I/O accede direttamente alla memoria, DMA è interposto

quando lo trasferisce al blocco





Architettura degli elaboratori – 1

- ❖ Fino a due decenni fa, la maggior parte dei sistemi di calcolo utilizzava una singola CPU general-purpose
 - ...pur inglobando molti microprocessori special-purpose *controllori dei vari sottosistemi*
- ❖ A partire dal 2005, i **sistemi multiprocessore**, detti anche **sistemi paralleli** *cluster di elaboratori* o **multicore**, hanno iniziato a dominare il mondo della computazione
 - **Maggiore throughput** (maggiore numero di processi terminati nell'unità di tempo) *Se ogni processore può girare un po' come perso ci saranno problemi, rimpiazzare alcuni di loro*
 - **Migliore economia di scala** (prezzi più bassi rispetto a più sistemi monoprocessore — condivisione di periferiche, memorie di massa, sistemi di alimentazione)
 - **Incremento dell'affidabilità** (i guasti non bloccano il sistema, *graceful degradation* o *fault tolerance*)
Sistemi in cui permettono degradazione con... *Resilienza ai guasti, grazie alla ridondanza*





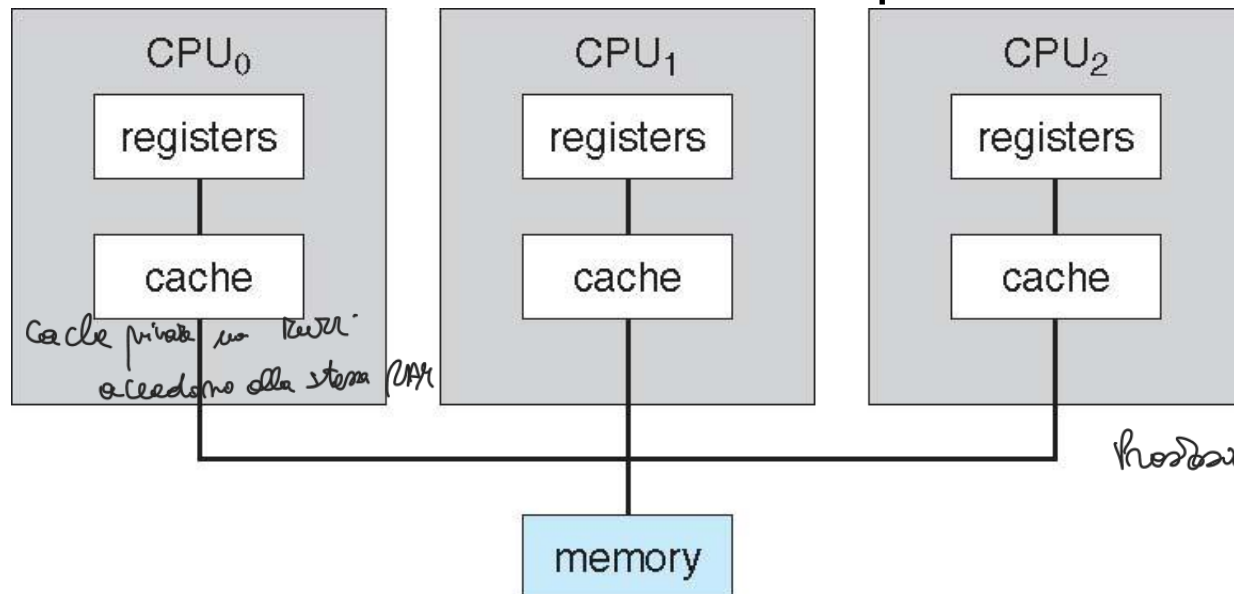
Architettura degli elaboratori – 2

❖ I sistemi multiprocessore attualmente in uso permettono...

- la **multielaborazione asimmetrica** — ad ogni processore viene assegnato un compito specifico; un processore principale sovrintende all'intero sistema
- la **multielaborazione simmetrica** — ogni processore è abilitato all'esecuzione di tutte le operazioni del sistema

un processore gestiva gli altri. Nel processore gestore il SO e negli altri processori

sim. User e Kernel



Architettura per il multiprocessing simmetrico (SMP)



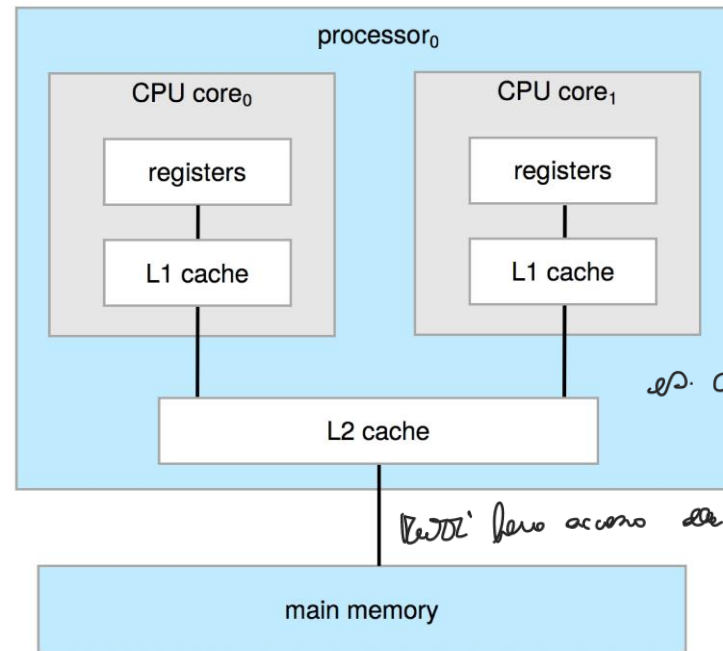


Sistemi multicore

*acquisti in serie
stessa, stesso clock*

- ❖ Nei **sistemi multicore** si raggruppano diverse unità di calcolo — **core** — in un singolo chip
 - Più efficienti, perché le comunicazioni, sul singolo chip, sono più veloci
 - Un chip multicore usa molta meno potenza di diversi chip single core

*in un CHIP non sono
più core*



es. come versioni rivoltate

questi hanno accesso alla RAM

*Migliore di più
processore o singolo
core
Scalari e altri cache
più veloce, meno
accesso alla
memoria
e sono più efficienti*

Architettura dual-core



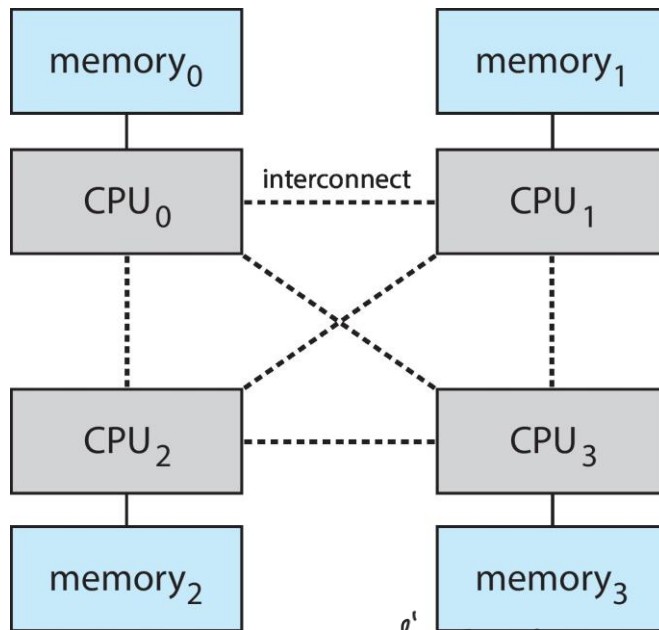


Non-Uniform Memory Access System (NUMA)

Possibile collocazione: tutti i core/CPU hanno rif. alla mem. come sullo stesso bus... idea: RAM dedicata a ciascun chip e una interconnessione che consente le varie CPU di accedere a quella non propria
tutto più scalabile

❖ Architettura **NUMA**: ogni CPU è dotata di una propria memoria locale; le CPU sono collegate da un'interconnessione di sistema condivisa

- Elimina il collo di bottiglia dovuto al bus di sistema comune
- Migliore economia di scala
- Accesso non uniforme alle memorie



se non trova l'info nella
RAM locale, deve trovarla
altrove

- Il SO può ottimizzare le prestazioni con un attento scheduling della CPU ed un'accurata gestione della memoria

Politica di bilanciamento
Serie di processi da eseguire sulla stessa architettura. Bilanciare i processi tra le varie CPU. Utile carico bilanciato di processi, politica FIFO, quando arriva un certo processo in RAM, la mette nella RAM dedicata a quella CPU, altrimenti deve passare sulla rete di interconnessione.





Cluster di elaboratori – 1

NOI LOOSELY COUPLED

- ❖ Simili ai sistemi multiprocessore, ma composti da due o più calcolatori completi, detti *nodi*, collegati tra loro
 - I cluster sono sistemi debolmente accoppiati
 - Normalmente, memoria secondaria condivisa tramite **SAN (storage–area network)**
- ❖ Notevole fault tolerance
 - **Clustering asimmetrico:** un calcolatore rimane nello stato di attesa attiva (in *hot–standby mode*), mentre gli altri eseguono le applicazioni
 - ➔ Il calcolatore in stand–by controlla gli altri nodi e si attiva nel caso di malfunzionamenti
 - **Clustering simmetrico:** tutti i nodi eseguono le applicazioni e si controllano reciprocamente

possono essere SIMM E ASIMM, ed. nodo gestione e altri calcoli

ma però si usa

Suoi processori, sua RAM e suo clock
nodi con la propria velocità

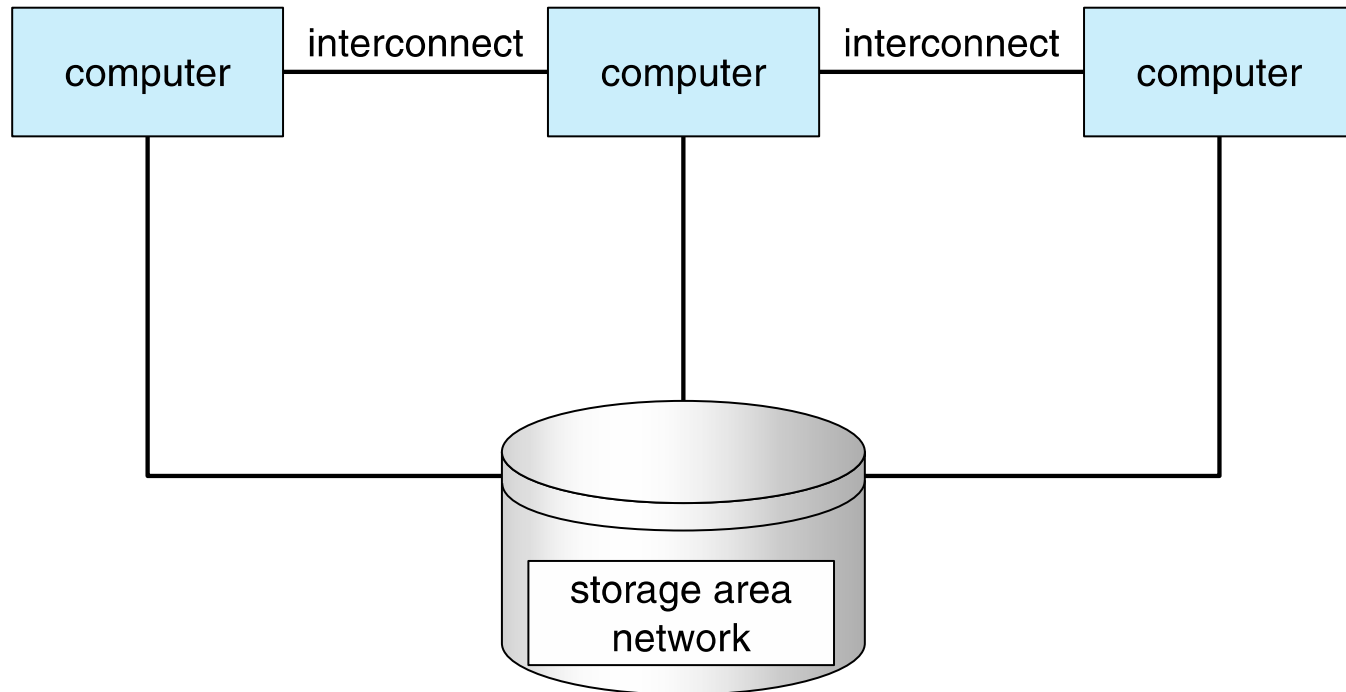
Di memoria volatile non si condivide nulla
ma solo memoria di MASSA





Cluster di elaboratori – 2

- ❖ I cluster vengono impiegati nell'**high-performance computing (HPC)**
 - Le applicazioni devono essere **parallelizzate**, cioè scritte per trarre vantaggio dall'architettura *devo essere interconnessi fra loro*





Struttura del sistema operativo – 1

*No più processi, più processi di processi si
memoria in RAM.*

❖ La **multiprogrammazione** è fondamentale per l'efficienza del sistema *avere contemporaneamente più processi in memoria.
Voglio la CPU sempre attiva, mai inattiva.*

- Un singolo processo non può tenere occupati contemporaneamente la CPU ed i device di I/O
- Con la multiprogrammazione, più processi (codice e dati) sono memorizzati contemporaneamente nella memoria principale, così da mantenere la CPU sempre occupata
- Il processo che verrà eseguito, ed otterrà quindi l'utilizzo esclusivo della CPU, viene selezionato mediante un algoritmo di **CPU scheduling** *quale processo eseguo per primo tra i tanti?
POLITICA per priorità, natura del processo o a priori, o equità FIFO*
- Quando un processo è in attesa di qualche evento (per esempio, un servizio di I/O), il SO seleziona un nuovo processo da eseguire *non deve essere selezionato, qui perché il processo deve stare nel suo blocco, non deve finire*

*Se un processo si ferma per fare ques. di I/O, lo metto in pausa e ne eseguo un altro,
non deve essere selezionato, qui perché il processo deve stare nel suo blocco, non deve finire*





Struttura del sistema operativo – 2

opzioni della CPU per un cambio di contesto, o se non c'è tempo, si riprendono dove ero prima
il SO deve GESTIRE l'AMMONTICCIAMENTO delle CPU, POLITICA equa e eff. per gestire Processi

❖ Il timesharing (o multitasking) è l'estensione logica della multiprogrammazione, nella quale la CPU passa così rapidamente dall'esecuzione di un processo a quella del successivo che gli utenti possono interagire con i loro processi durante la fase di running — interactive computing

- Il tempo di risposta deve essere $\ll 1$ secondo
- Ciascun utente ha almeno un programma allocato in memoria centrale "in esecuzione" $\Rightarrow$ un processo è un programma in esecuzione
- Se più processi sono pronti per essere eseguiti, deve essere implementato un algoritmo per lo scheduling della CPU
- Se la memoria non è sufficiente a contenere tutti i processi, mediante swapping, essi vengono spostati fuori/dentro la memoria principale in base al loro stato di esecuzione effettivo
Processo eseguito lo scarico sulla memoria di MASSA, dove sono forse un grigio più che ribocchi a lui e la recupero. Ma LEVATE memoria di MASSA
- La metafora della memoria virtuale garantisce l'esecuzione di processi non contenuti completamente nella memoria principale e più grandi della memoria fisica presente nel sistema

Ma in
RAM molti
più processi

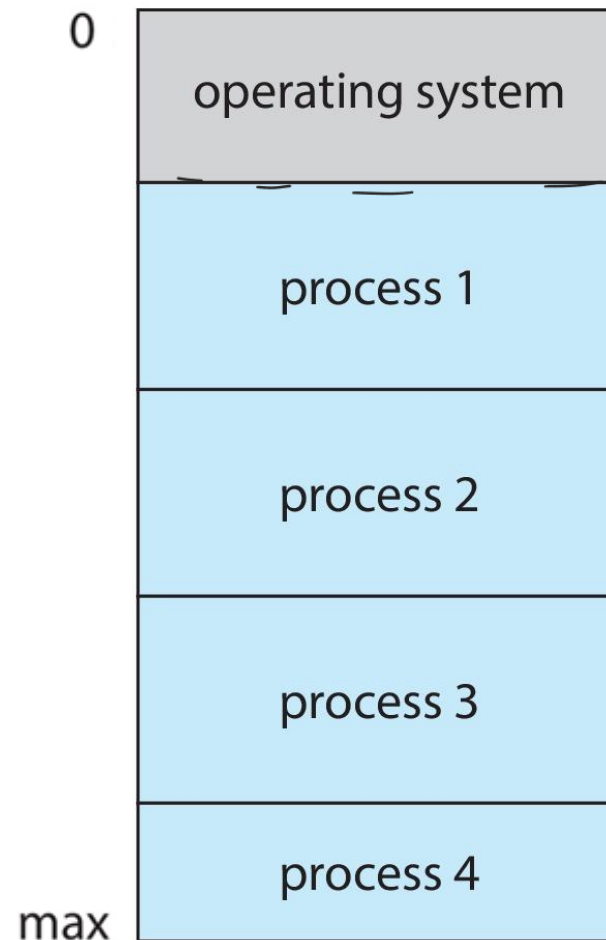
Contra solo un po' della memoria che sia in RAM, non
riferendo più interi processi, ma pagine
1.48 POSSO avere un processo più GRANDE
nella MEM. FISICA
PAGING
Silberschatz, Galvin and Gagne





Multiprogrammazione

Configurazione della memoria per un sistema multiprogrammato



- So braccia il
quanto è grande il
limite della memoria
i quanto RAM per
occupare

per mezzo
occupo un
blocco





Attività del sistema operativo – 1

Sullo CPU trasferiscono
il codice utente, for ANOVA
interrupt. -

- ❖ Il **Sistema Operativo** è **interrupt-driven**
 - Interrupt hardware provenienti dai device
 - Errori nel software o richieste di servizi creano situazioni di **eccezione** o **trap**
 - ▶ Divisioni per zero, processi in loop (riempimento dello stack), processi che tentano di modificare lo spazio di memoria dedicato ad altri processi o al sistema operativo
 - ▶ Richieste di servizio al SO
- ❖ La modalità operativa **dual-mode** permette al SO di proteggersi e di proteggere le varie componenti del sistema di calcolo





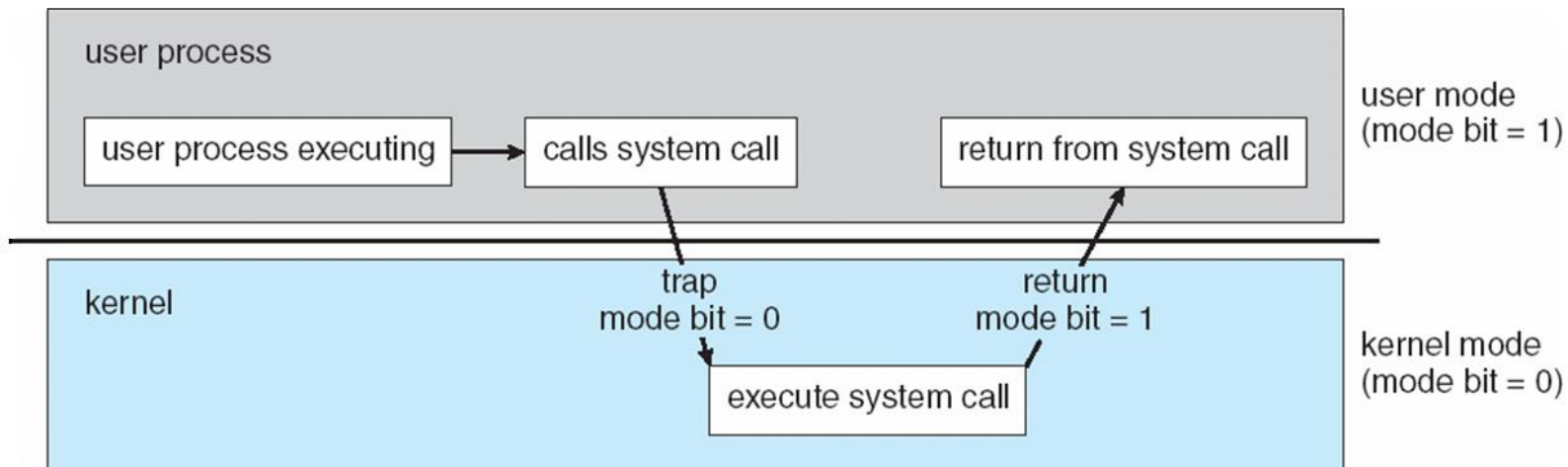
Attività del sistema operativo – 2

interrupt, Mr
trap, e SW

bit a 0 in Modalità kernel, 1 user

❖ *User mode* e *kernel mode*

- *Mode bit* cablato in hardware
 - ▶ Il valore assunto dal mode bit distingue le due situazioni in cui il sistema esegue codice utente o codice kernel; le istruzioni privilegiate possono essere eseguite soltanto in modalità kernel
 - ▶ Ogni *system call* effettua il passaggio in modalità kernel; il ritorno dalla chiamata riporta il sistema in modalità utente





Attività del sistema operativo – 3

- ❖ Il concetto di modalità può essere esteso oltre il dual mode (utilizzo di più bit di modo)
 - Le CPU hanno spesso una modalità distinta per gestire la virtualizzazione, ovvero per indicare che il gestore della macchina virtuale ha il controllo del sistema (più privilegi dei processi utente, ma meno del kernel del SO ospitante)
 - Possibilità di modalità distinte per le diverse componenti del kernel
 - **Esempio:** le CPU Intel 64 supportano quattro livelli di privilegio

es. Virtual Machine / Simula HW e SW, può essere
in uno SO. completo. isolato dal S.O.
host





Passaggio da modo utente a modo kernel

SO deve avere una limitazione, non possono più ottenere
nessuna risorsa senza poi restituirlo e riutilizzarlo.

❖ Per prevenire processi che eseguono cicli infiniti senza
più restituire il controllo al SO $\Rightarrow$ **Timer**

Program. e controllo del
clock del sistema
il SO lo imposta

- Si realizza mediante un clock e un contatore
- Emissione di un interrupt dopo un dato periodo di tempo

❖ Il SO inizializza il contatore al tempo massimo (stima-
to) di esecuzione del processo

Timer a 0, interrupt, SO riprende
controllo CPU.

- Il contatore viene decrementato ad ogni impulso del clock
- Quando il contatore assume valore zero, si genera un interrupt, che restituisce il controllo al SO
- Il timer viene impostato ogni volta che il processo ottiene l'uso della CPU, in base al tempo di esecuzione assegnatogli, e ne garantirà l'interruzione, con restituzione del controllo al SO, in caso di superamento

SO in modo di
controllo.
es. ROUND ROBIN
timer a 0, SO
Linux ha
non timer
ma tempo diverso
in base alla
priorità





Gestione dei processi – 1

❖ Un processo è un programma in esecuzione

- Il programma è un'entità passiva, il processo un'entità attiva *la contiene tutto, i suoi registri, la stessa risorsa, variare* *è un file, oggetto solo memorizzato nella memoria fisica*

❖ Il processo è l'unità di lavoro del sistema di calcolo

❖ Un processo necessita di alcune risorse per assolvere al proprio compito: tempo di CPU, memoria, file, dispositivi di I/O e dati di inizializzazione

❖ La terminazione di un processo prevede il recupero di tutte le risorse riutilizzabili ad esso precedentemente allocate

Unità di lavoro, thread

Contenitore di thread ← processo - window

+ *controlla controllo*
 + *stampa video*
 + *controlla interfaccia*

} *thread*

Processo, istanza programma in esecuzione

Un programma può essere visto a più processi

fa eseguire contenuto registri CPU





Gestione dei processi – 2

*eseguiti in modo
sequenziale*

- ❖ I processi *single-thread*^{FILLO, + NAMA} hanno un unico *program counter*, che indica la locazione in memoria della prossima istruzione da eseguire
 - Le istruzioni di ogni processo vengono eseguite sequenzialmente, una dopo l'altra, fino al termine del processo
- ❖ I processi *multi-thread* hanno un program counter per ogni thread
 - un filo che il quale esegue varie istanze*
importante exec. sequenziale per catturare localita' spaziale
- ❖ Normalmente, in un sistema, vi sono molti processi di uno o più utenti, e alcuni processi del SO, che vengono eseguiti in concorrenza su una o più CPU
 - La concorrenza è ottenuta effettuando il *multiplexing* delle CPU fra i vari processi/thread

*lotta per un process.
quidiamo al successo*





Gestione dei processi – 3

❖ Il SO è responsabile delle seguenti attività relative alla gestione dei processi:

■ Creazione e cancellazione di processi (utente e di sistema) *→ nel senso NT, o kernel, o proprio tempo*
UNIX, sys call Fork crea processo

■ Scheduling dei processi/thread sulla/e CPU *o a se thread*

■ Sospensione e riattivazione di processi *anche RL un processo in WAIT ma ripreso*

■ Fornire meccanismi per

▶ la comunicazione fra processi *sulle stesse macchine comunicano tra loro,*

▶ la sincronizzazione dei processi

*dopo condivisione sbli, ma
devo mantenere coerenza
con sincronia*

▶ la gestione del **deadlock**

*stallo, circolo vizioso
di attesa per accesso delle risorse*

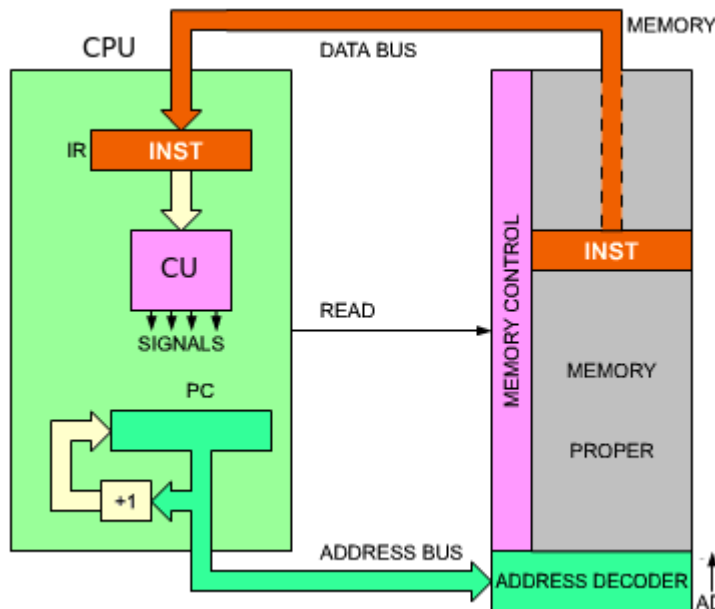
• o attesa indefinita





Gestione della memoria centrale – 1

- ❖ La **memoria centrale** è un vettore di miliardi di parole, ciascuna con un proprio indirizzo ^{Word individuale al byte}
- ❖ È un deposito di dati rapidamente accessibili, condiviso dalla CPU e dai dispositivi di I/O
 - ...ma è lenta rispetto alla CPU
- ❖ La CPU legge le istruzioni dalla memoria durante la fase di *fetch*, oltre a leggere e scrivere i dati



- ❖ Il modulo per la gestione della memoria determina cosa è in essa contenuto ad un certo istante, per ottimizzare l'utilizzo della CPU ed il tempo di risposta del sistema agli utenti

Fetch instr., controllo PC, memoria istru., viaggio sul BUS DATI, arrivo dell'INSTRUC. REG, arrivo deposito ALU, arrivo memoria - LENTO.





Gestione della memoria centrale – 2

Processi in RAM ha un costo involontario: deve stare nel suo SPAZIO INDIVIDUALE

- ❖ Per eseguire un programma, tutte le sue istruzioni (o, almeno, una parte di esse) devono risiedere in memoria centrale; ugualmente dicasi per i dati
- ❖ Il SO è responsabile delle seguenti attività connesse alla gestione della memoria centrale:

- Tenere traccia di quali parti della memoria sono attualmente usate e da chi
- Decidere quali (parti di) processi caricare in memoria quando vi è spazio disponibile
- Allocare e deallocare lo spazio di memoria secondo necessità

quantità grande MULTIPROGRAMMING può far decodificare le istruzioni; se ne ha bisogno di programmi, passo verso TRAPPING, LIMITI della MULTIPROGRAMMING.





Gestione della memoria secondaria

↑ accesso tramite "metafora del FILE"

Prima di tutto devo aprire il FILE

❖ Il SO garantisce una visione logica uniforme del processo di memorizzazione secondaria

- Astrae dalle caratteristiche fisiche dei dispositivi per definire un'unità di memorizzazione logica — il **file**

File, concetto logico di alto livello
virtuale: MM,
file e directory

► Il concetto di file è indipendente dal mezzo sul quale viene memorizzato (che ha caratteristiche proprie e propria organizzazione fisica) *non dipende dalle specifiche memoria secondaria, dipende dalla tecnologia*

► I file possono contenere programmi e dati

► I file di dati possono essere numerici, alfabetici, alfanumerici o binari e possono avere una struttura libera (es. file di testo) o rigidamente formattata (es. file mp3) *Struttura formattata, illeggibile*

- Ciascuna periferica viene controllata dal relativo **device driver**, che nasconde all'utente (e al resto del sistema) le caratteristiche fisiche variabili dell'hardware

► Modalità e velocità di accesso, capacità, velocità di trasferimento, etc.

Gestore memoria di massa





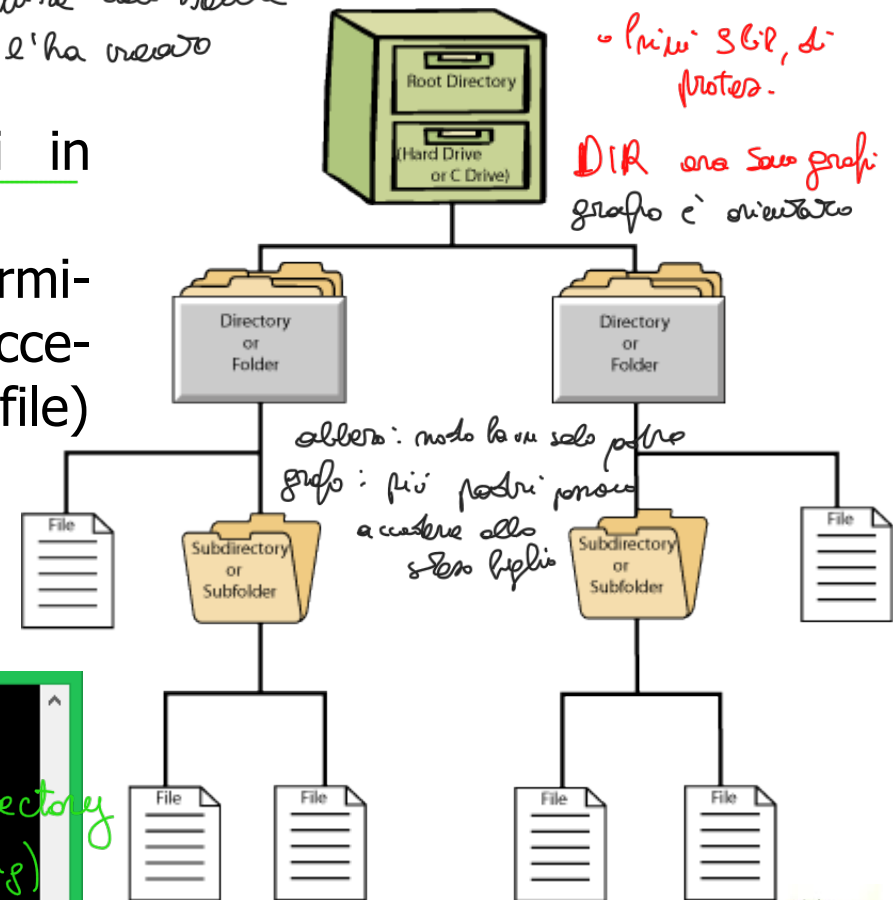
Gestione del file system – 1

es. insieme informazioni, forse appetto, file di dati: numerici, alfanumerici, multimediali.

❖ Un file è una collezione di informazioni correlate, definita dal suo creatore *definita dall'utente che l'ha creato*

❖ Gestione del file system

- I file vengono organizzati in directory
- Controlli di accesso per determinare quali utenti possono accedere e come a quali risorse (file)



path: percorso dalla radice al percorso

UNIX Windows

```
root@tecadmin:~/test#  
root@tecadmin:~/test# ls  
logs myfile.txt test.log  
root@tecadmin:~/test#  
root@tecadmin:~/test# ls -l  
total 232  
drwxr-xr-x (2) root root 4096 Sep 5 17:05 logs  
-rw-r--r-- (4) root root 12 Sep 5 17:04 myfile.txt  
-rw-r--r-- (4) root root 227829 Sep 5 17:04 test.log  
root@tecadmin:~/test#
```

directory
File

Bit di protezione che sono
presenti sul file?

directory: collezione di inform. di file, è un file che contiene le info dei file e i puntatori ai file nella memoria





r: read, l.s. | vedo contenuto file
w: write, inserire o cancellare file
x: accendo
alla oliv, cioè posso fare col, (no) lavoro:

n w x per le cartelle

Gestione del file system – 2

- ❖ Il SO è responsabile delle seguenti attività per la gestione dei file:
 - Creazione e cancellazione di file e directory
 - Supporto alle funzioni elementari per la manipolazione di file e directory
 - Associazione dei file ai dispositivi di memoria secondaria *parte mancante a noi*
 - Backup di file su dispositivi stabili di memorizzazione (memoria terziaria) *memorizzo. permanente*





Gestione dei supporti di memoria secondaria

Memoria di Memoria: Parte visibile utente (per il File system) e NON (per il sistema operativo)
 • Parte non visibile
 più delle Memorie

- ❖ La memoria centrale è volatile e troppo piccola per contenere permanentemente tutti i dati
 - Il sistema di elaborazione deve consentire l'archiviazione secondaria, per salvare i contenuti della memoria centrale
 - Gli SSD (e i dischi magnetici) costituiscono il principale mezzo di memorizzazione permanente
- ❖ La velocità di accesso alla **memoria secondaria** è fondamentale per le prestazioni del sistema, che risentono fortemente della bontà degli algoritmi utilizzati per il reperimento dei dati in essa contenuti
- ❖ Il SO è responsabile delle seguenti attività:

- **Creazione delle partizioni**
- **Operazioni di montaggio e smontaggio**
- **Gestione dello spazio libero**
- **Allocazione dello spazio**
- **Accesso/protezione al/del dispositivo di memoria**

Attacco nuovo
 di p. di memoria
 e il suo Filesystem
 sono appaiono
 a quello del
 PC

Controllo dei blocchi / pagine libere

di per fare cosa nel particolare File

il SO si accorge che c'è un file system, e ne ha ricercato la MONTA, lo appaia al proprio della Directory o. v. m. la monta a livello di PC





Gestione della memoria terziaria

rimovibili

- in corrispondenza di un unico drive o driver, più disponibili*
- ❖ I dispositivi di **memoria terziaria** (utilizzati tipicamente off-line) non devono essere particolarmente veloci
 - Le memorie terziarie sono normalmente realizzate su supporti rimovibili (ad esempio: dispositivi USB basati su memorie flash) → un unico drive utilizzato per diversi media *non impattano sull'esecuzione di processi*
 - Devono comunque essere gestite in modo efficiente (dal SO o da programmi di sistema)
 - Supporto del SO: installazione/rimozione dei media (montaggio/smontaggio dei relativi file system), allocazione dei dispositivi ai processi che ne richiedono l'uso (esclusivo)





Performance dei diversi livelli di memoria

- ❖ Lo "spostamento" di dati nella gerarchia di memorie può essere **esplicito** (mediato dal SO) o **implicito** (gestito dall'hardware)

- da disco a RAM
- da cache a registri

*p.es. da disco a RAM
e eseguire un programma,*

*la cache è
rapida, e ciò è un
po' che "almeno" è più veloce
nella CPU*

livello più alto è la cache e quello sottostante

Level	1	2	3	4	5
Name	registers	cache	main memory	solid state disk	magnetic disk
Typical size	< 1 KB	< 16MB	< 64GB	< 1 TB	< 10 TB
Implementation technology	custom memory with multiple ports CMOS	on-chip or off-chip CMOS SRAM	CMOS DRAM	flash memory	magnetic disk
Access time (ns)	0.25 - 0.5	0.5 - 25	80 - 250	25,000 - 50,000	5,000,000
Bandwidth (MB/sec)	20,000 - 100,000	5,000 - 10,000	1,000 - 5,000	500	20 - 150
Managed by	compiler	hardware	operating system	operating system	operating system
Backed by	cache	main memory	disk	disk	disk or tape

Dati 2019



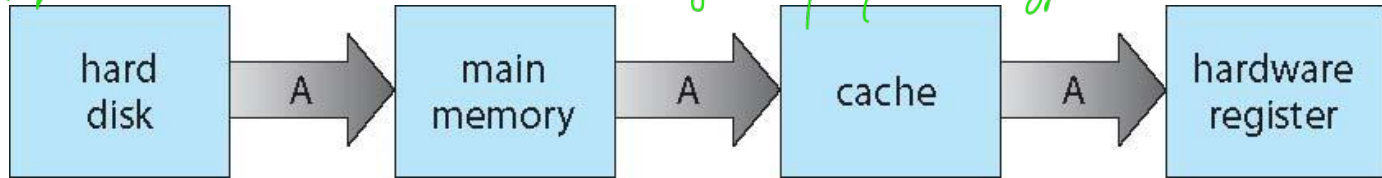


es. voglio aggiornare variabile file A++ contenuto nel file B

Migrazione di un dato "A" da disco a registro

- ❖ Si voglia modificare il dato "A", residente nel file "B"
- ❖ In ambiente multitasking, è necessario assicurare che qualsiasi processo che desideri accedere ad "A", ottenga dal sistema il valore aggiornato *Coerenza*

1) spostato in memoria => in cache => nel registratore per poterlo aggiornare



Se ho più processi che vogliono modificare A? la vanno a prendere dal file, dato non coerente

Si sistema multiprocessore
coerenza più
complicata, devo
aggiornare tutte le
singole cache dei processori

In ambiente multiprocessore, la **coerenza della cache** si ottiene garantendo che l'aggiornamento di "A" in una qualsiasi cache si rifletta immediatamente in tutte le cache in cui "A" risiede (il problema si risolve a livello hardware, senza intervento del SO)

meglio ancora
MM diverse, dati devono
essere disponibili alla
rete

es. cluster
In ambiente distribuito, più copie del dato possono coesistere in memorie non volatili distinte con cache distinte, ma la coerenza deve essere comunque garantita

Appena modifico il file lo rendo subito disponibile ai tutti i processori
per evitare coerenza,





Parte grossa del sistema perché l'IO fa tutto e varia ogni

Il sottosistema di I/O – 1

Filo solo esistente per la memoria di massa una volta per l'accesso ad alcuni dispositivi di I/O

- ❖ Uno degli scopi del SO consiste nel nascondere all'utente le caratteristiche dei dispositivi hardware
- ❖ Il sottosistema di I/O è formato da: con in questo fa cioè I/O
 - Una componente di gestione della memoria che include
 - il buffering (la memorizzazione temporanea di dati in memorie locali alle periferiche, durante il trasferimento);
 - il caching (la memorizzazione parziale dei dati in memorie ad accesso rapido – mappatura dei buffer in memoria centrale);
 - lo spooling (*simultaneous peripheral operations on-line*, per la gestione dell'I/O asincrono)
 - Un'interfaccia generale per i driver dei dispositivi
 - I driver per i dispositivi specifici presenti nel sistema di calcolo

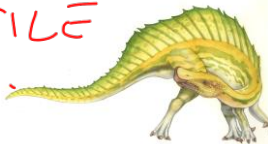
l'info si sposta nel buffer del disco e poi in RAM

Mappa in RAM per i processi es. MAPPO buffer in RAM

per oper. di SMP

UNIX, file e I/O fanno la stessa cosa

mi permette di fare su file come se fossero strutture virtuali, su FILE il processo può andare avanti

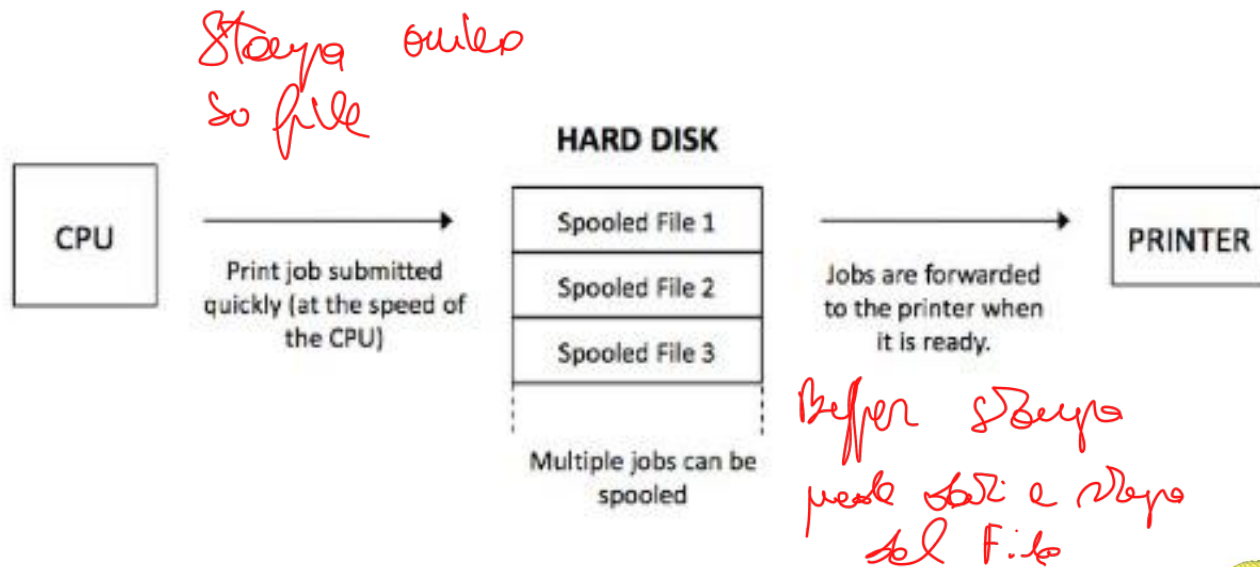




Il sottosistema di I/O – 2

Spooling

Spooling is the process of sending output (intended for a printer or other peripheral) to a disk file, temporarily, so that it can be forwarded to the printer (or other peripheral) when it is free.





Protezione e sicurezza – 1

*↳ processo che definiscono cosa
gli utenti possono fare e non*

- ❖ Se diversi utenti usufruiscono dello stesso elaboratore che consente l'esecuzione simultanea e concorrente dei processi, l'accesso alle risorse del sistema di calcolo dovrà essere disciplinato da regole imposte dal SO
- ⇒ File, segmenti di memoria, CPU, altre risorse possono essere utilizzate solo dai processi che hanno ottenuto apposita autorizzazione dal SO
- ❖ **Protezione** — è l'insieme di tecniche usate per controllare l'accesso da parte di processi e/o utenti a risorse del sistema di calcolo
 - Le strategie di protezione devono fornire le specifiche dei controlli da attuare (politiche) e gli strumenti per la loro applicazione (meccanismi)

Politica: scelta di un algoritmo per risolvere i problemi

Meccanismo: meccanismi per implementare : se generale, posso implementare





es. TIMEL è un meccanismo. Come lo uso è una politica

Protezione e sicurezza – 2

es. fare lo stesso
ma su un
processo

→ generale, non implementare per politiche

❖ **Sicurezza** — è il meccanismo di difesa implementato dal sistema per proteggersi da attacchi interni ed esterni

attacchi classici

❖ **Tipi di violazione**

down to server

uso delle risorse

- **Denial-of-Service:** malfunzionamento dovuto ad un attacco informatico in cui si esauriscono deliberatamente le risorse di un sistema di calcolo che fornisce un servizio, fino a renderlo non più in grado di erogarlo
- **Trojan:** programmi che hanno una funzione conosciuta legittima e una funzione dannosa nascosta
- **Worm:** malware in grado di autoreplicarsi
- **Virus:** porzioni di codice dannoso che si legano ad altri programmi del sistema per diffondersi





Protezione e sicurezza – 3

- ❖ Chi mette in atto gli attacchi alla sicurezza?
 - **Hacker:** *pirata informatico* che si impegna nell'affrontare sfide intellettuali per aggirare o superare creativamente le limitazioni di accesso ai sistemi
 - **Cracker:** chi si ingegna per eludere blocchi imposti da qualsiasi software al fine di trarne guadagno
 - Alcune tecniche di attacco sono simili, ma l'hacker è colui che sfrutta le proprie capacità informatiche per esplorare, divertirsi, apprendere, senza creare danni; al contrario il cracker è colui che sfrutta le proprie capacità (o, in certi casi, quelle degli altri) al fine di distruggere, ingannare e arricchirsi





Protezione e sicurezza – 4

- ❖ Più in dettaglio... *Come influenzerà la sicurezza?*
 - La funzione della sicurezza riguarda la prevenzione dall'uso illegale o dalle interferenze operate da persone o programmi fuori dal controllo del sistema operativo
 - La funzione di protezione riguarda l'accesso improprio a risorse del sistema causato da utenti accreditati
- ❖ *Autenticazione. Nome e password* In prima istanza, infatti, il sistema distingue i propri utenti, per determinare chi può fare cosa
 - L'identità utente (*Numero* **user ID**) include nome dell'utente e numero associato — uno per ciascun utente
 - L'user ID garantisce l'associazione corretta di file e processi all'utente e ne regola la manipolazione
 - L'identificativo di gruppo permette inoltre ad un insieme di utenti di accedere correttamente ad un pool di risorse comuni (file e processi)

es. user name e password

numero psw, confronto con un dato crittato, se è uguale, utente entra

```
- es. ls -l pippo.c  
-rw-r--r-- 1 susanna users 1064 Feb 6 2002 pippo.c
```

Protezione
r - permesso di lettura (directory, listing)
w - permesso di scrittura (directory, aggiungere file)
x - permesso di esecuzione (directory, accesso)

File creato da utente, egli ne è il proprietario

Stell'utente ogni file, ogni processo avrà associato l'user ID e non utente





La virtualizzazione – 1

- Modelli per cui più sistemi operativi possono essere eseguiti come processi in un LO HOST
- ❖ La virtualizzazione è una tecnica che permette di eseguire un sistema operativo come applicazione all'interno di un altro SO *architettura HW e SW, e da qui quali ci sono macchine virtuali, gestite da SO diversi.*
 - ❖ I programmi di virtualizzazione sono software di **emulazione**: *replano con linguaggio macchina diverso, i SW. bisogno macchine di vecchia architettura deve essere tradotti in un'altra architettura.*
 - ❖ L'emulazione viene utilizzata, per esempio, quando la CPU su cui è stata compilata un'applicazione è diversa da quella su cui verrà eseguita
 - Quando Apple è passata dalle CPU IBM Power alle Intel ha fornito l'emulatore Rosetta per permettere la portabilità delle vecchie applicazioni
 - Analogamente, Rosetta 2 emula le istruzioni Intel sui nuovi processori M1/M2 di Apple

VM meno performanti, e che consumano le risorse HW, CPU, DISCO





La virtualizzazione – 2

- ❖ Il concetto di emulazione può essere esteso per permettere ad un SO, scritto e compilato per una data piattaforma, di essere eseguito su piattaforme diverse
- ❖ Una macchina virtuale realizza un'interfaccia indistinguibile dalla macchina fisica sottostante: ogni **processo ospite** usufruisce di una copia virtuale del calcolatore
- ❖ Le risorse del computer fisico vengono condivise dall'**hypervisor** o **VMM** (*Virtual Machine Manager*) in modo che ciascuna macchina virtuale sembri possedere il proprio processore, la propria memoria e i propri dispositivi
- ❖ Sia l'hardware virtuale che il sistema operativo (ospite) vengono eseguiti in un ambiente (strato) isolato
- ❖ Le risorse del computer fisico vengono condivise in modo da creare le macchine virtuali

gerente di macchine virtuali
Hypervisor
livello di interfaccia
intermedio tra
supervisor e
user
compone le
risorse
Kernel Host
pero' mappa
accesso alle
risorse





La virtualizzazione – 3

- ❖ L'hypervisor può essere parte di un SO host (per esempio un modulo) o un micro-kernel che genera le macchine virtuali (VM)
- ❖ L'hypervisor può sfruttare caratteristiche specifiche del processore (hardware virtualization)
 - Lo scheduling della CPU può creare l'illusione che ogni utente abbia un proprio processore
 - Lo spooling e il file system possono fornire dispositivi di I/O virtuali (per esempio, stampanti) *STAMP VIRTUALE, STAMPA contemporanea*
 - Lo spazio disco può essere "suddiviso" per creare dischi virtuali
 - Con due bit di modo, in certe architetture, si può avere una modalità esecutiva tipica dell'hypervisor (intermedia rispetto a modo utente e modo kernel)



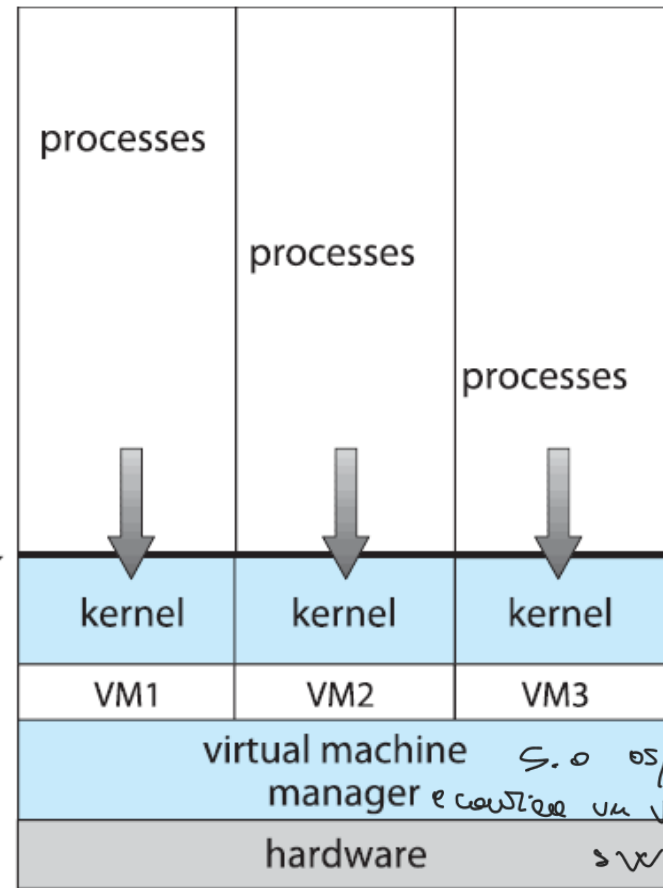


La virtualizzazione – 4



sistema normale
VM e SO che lo gestisce, e
sopra ci sono i processi

programming
interface



virtualinos.

S.O. ospita
e controlla VM
SW, su cui
facio girare
so differenti con le
applicazioni





La virtualizzazione – 5

*VM separate l'ine dell'OS. Sono macchine a se stanti
condividono FISICAMENTE ma non logicamente*

- ❖ Il concetto di macchina virtuale fornisce una protezione completa delle risorse di sistema (hardware e SO ospitante), dato che ciascuna macchina virtuale è isolata da tutte le altre
 - Questo isolamento, tuttavia, non permette, in generale, una condivisione diretta delle risorse
 - In Linux, condivisione possibile, per esempio, con **VirtIO**
- ❖ Per la condivisione di risorse... *va pure so rete virtuale, virt I/O, meglio in rete macchine virtuali*
 - Condivisione di un volume del file system
 - Rete di macchine virtuali: ogni macchina virtuale può inviare/ricevere informazioni sulla rete privata virtuale, modellata come una rete fisica, ma realizzata via software
 - La memoria inutilizzata viene ceduta ad altre VM





La virtualizzazione – 6

- ❖ Un sistema con macchine virtuali è un mezzo perfetto per la ricerca e lo sviluppo di sistemi operativi
 - Lo sviluppo del SO è effettuato sulla macchina virtuale, invece che sulla macchina fisica, così da non interferire con il normale funzionamento del sistema
 - Difficoltà nell'ottimizzare le prestazioni a causa dell'emulazione *base per debugging, ma per ottimizzare serve HW fisico*
- ❖ Ulteriori casi d'uso ed esempi
 - Computer Apple con Mac OS come host e Windows come guest
 - Sviluppo e messa a punto (quality assurance) di applicazioni per diversi SO sullo stesso sistema HW/SW
 - Gestione dei dati di varia provenienza nei data center
 - Possibilità di incrementare la vita dei programmi e mezzo per studiare vecchie architetture di sistema





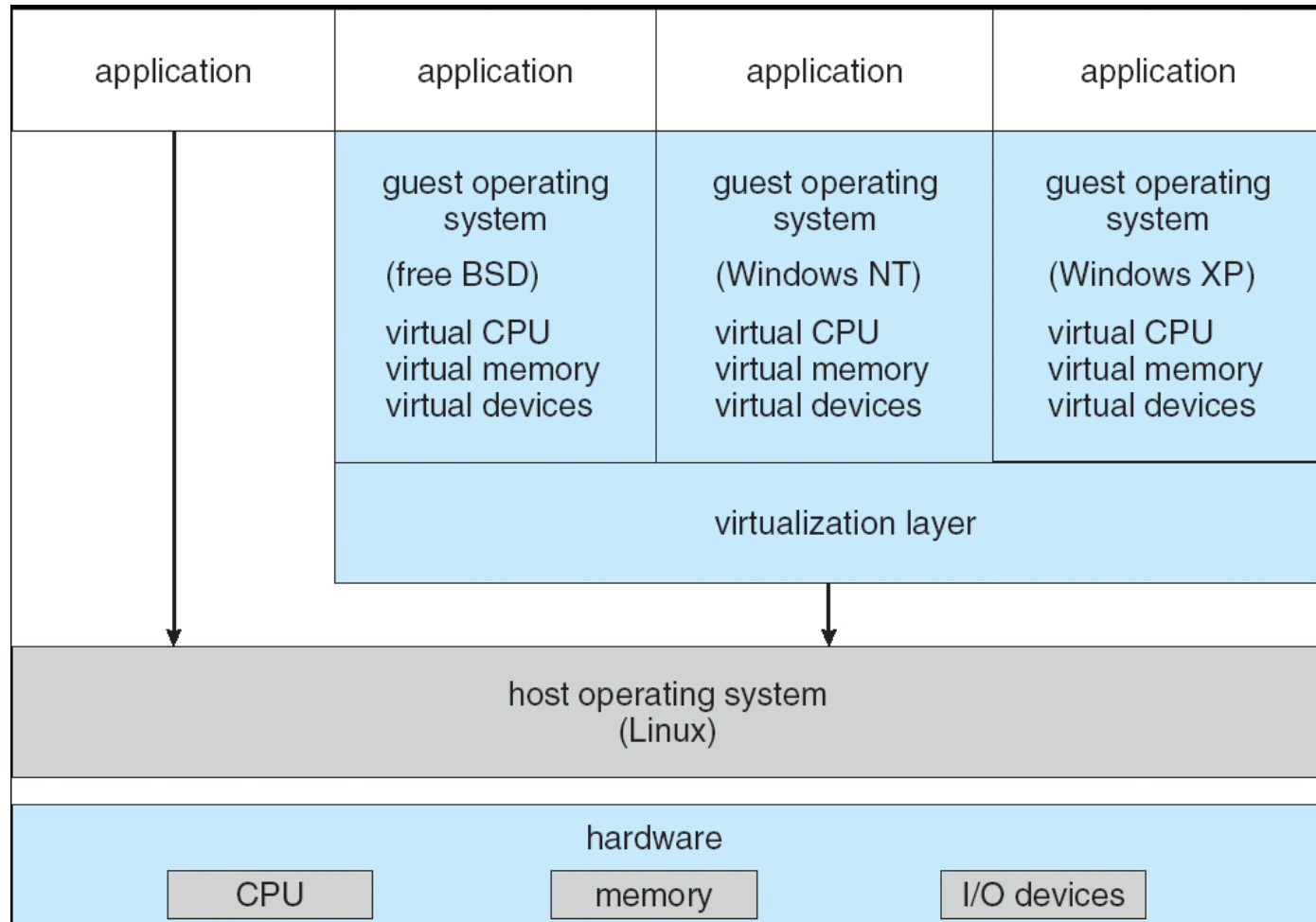
La virtualizzazione – 7

- ❖ Tra i vantaggi dell'utilizzo di macchine virtuali vi è inoltre il fatto di poter offrire contemporaneamente a utenti diversi ambienti operativi separati
- ❖ Il concetto di macchina virtuale è difficile da implementare per il notevole sforzo richiesto per fornire un duplicato esatto (sebbene meno potente) della macchina fisica
- ❖ Le più diffuse sono **Kvm**, **VMware**, **Virtualbox**, **Parallels**, **ESXi**, **Citrix Hypervisor**
- ❖ In alcuni casi i VMM possono fungere essi stessi da sistemi nativi, senza la presenza di un SO host





La virtualizzazione – 8



Architettura VMware





I vantaggi della virtualizzazione

Partizionamento

- Esecuzione di più sistemi operativi su una sola macchina fisica.
- Suddivisione delle risorse di sistema tra le macchine virtuali.

Isolamento

- Isolamento di guasti e problemi di sicurezza a livello di hardware.
- Protezione delle prestazioni grazie a controlli avanzati delle risorse.

Incapsulamento

- Salvataggio su file dell'intero stato di una macchina virtuale.
- Spostamento e copia delle macchine virtuali con estrema facilità, in modo analogo ai file.

Indipendenza dall'hardware

- Provisioning o migrazione delle macchine virtuali a qualsiasi server fisico.





Strutture dati del kernel – 1



Vincolo di dimensione FISSA

❖ Un array è una semplice struttura dati in cui ogni elemento è direttamente accessibile tramite un indice

- La memoria principale è costruita come un array, così come il vettore degli interrupt

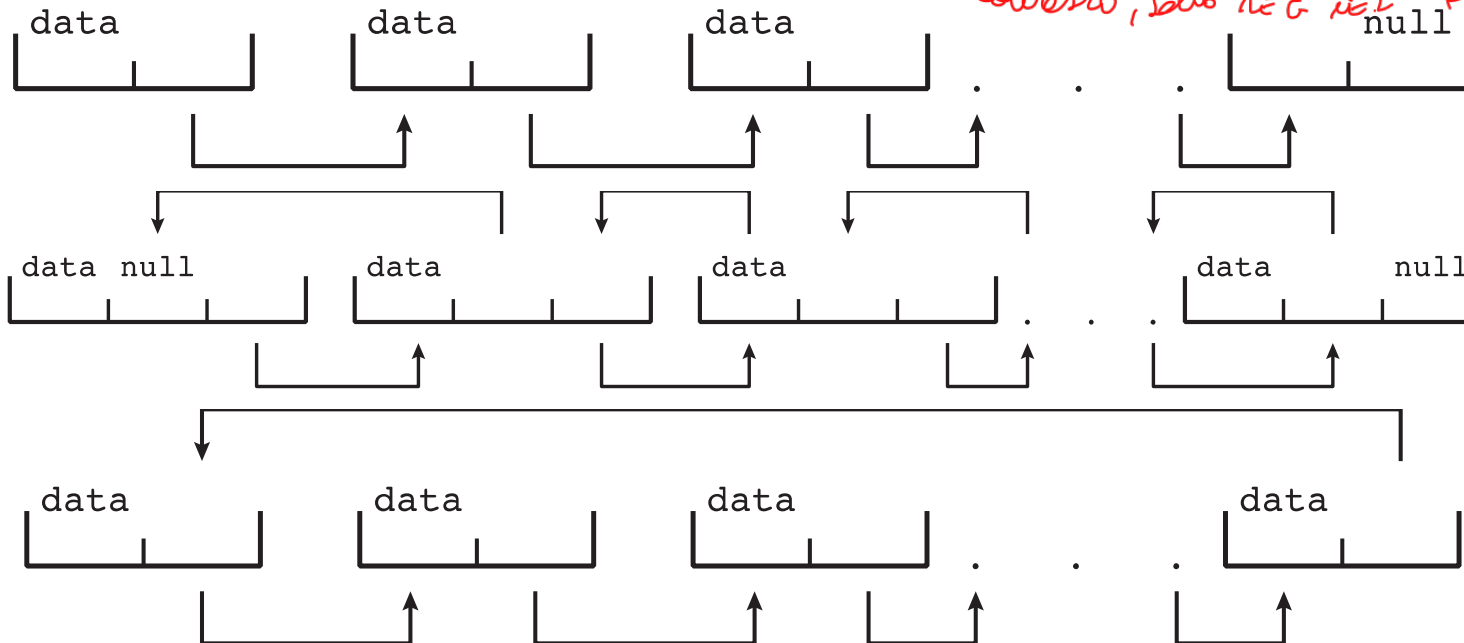
tutti i tipi di memoria sono array, indicata a byte. anche il disco è un array

❖ Come procedere però in caso di dati di dimensione variabile? Come si può rimuovere un elemento, mantenendo l'ordine relativo degli altri?

es. liste il PCB

- Uso di liste concatenate

se ho oggetti non uniformi? RECORD, dato che gestisce il SO, vuole di contesto, solo nel PCB



Lista *non al* concatenata semplice

Lista doppiamente concatenata

non al succ e il pre corrente

Lista circolare

es PCB



PCB è un record, ha dati del reg, nome utente, dove è in mem. - info process

esempi in cui
liste usate
dal S.O.



Devo avere una LISTA dei record

lista di PC, nomi col'm, lista circolare

Strutture dati del kernel – 2

lista doppiamente concatenata o lista in disco per avere ridondanza

ogni el. della lista è un RECORD

RECORD = STRUTTURA dati variabili

- ❖ Le liste concatenate possono contenere dati di diversa natura e dimensione (collezionati all'interno di record)
- ❖ Potenziale svantaggio: il costo della ricerca è $\mathcal{O}(n)$
per cercare nella lista devo scorrerla
- ❖ Le liste sono talvolta utilizzate direttamente dagli algoritmi del kernel, ad esempio in per certe modalità di scheduling della CPU o nell'algoritmo Clock per la sostituzione delle pagine
- ❖ Più frequentemente le si utilizza per implementare **pile** e **code**

chiamata funzione

Pila, li mette
variab. locali della
funzione, parametri attuali
e indirizzo di ritorno

STACK FRAME



Fino pile, faccio pop. PILA

STACK implementata con una LISTA





Strutture dati del kernel – 3

- ❖ Lo **stack** viene acceduto con politica LIFO (Last In First Out) e le operazioni di inserimento ed estrazione (cancellazione) vengono rispettivamente dette *push* e *pop*
 - Il SO utilizza lo stack per gestire le chiamate di funzione
 - ▶ Al momento della chiamata, si inseriscono nello stack l'indirizzo di ritorno, le variabili locali e i parametri attuali
 - ▶ ...che vengono recuperati dalla funzione chiamata
 - Anche il cambio di contesto viene gestito tramite stack
- ❖ Le **code** si accedono in modalità FIFO (First In First Out)
 - Il SO gestisce come una coda, per esempio, i documenti inviati ad una stampante o i processi in attesa di ottenere l'accesso alla CPU

STACK, LIFO,
uso per chiamate di funzione
e cambio contesto

CODE:

• processo invocato prima deve essere il primo
ad essere servito

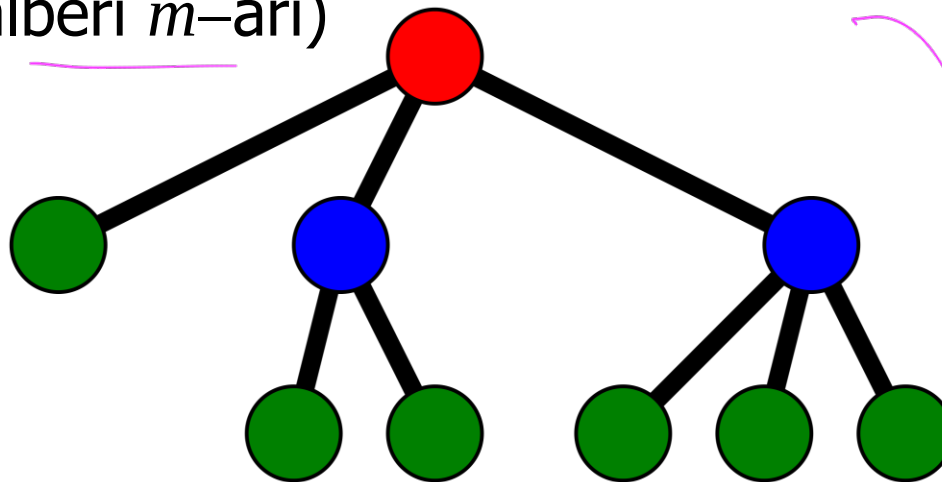




Strutture dati del kernel – 4

altra struttura dati, c'è una gerarchia

- ❖ Gli **alberi** sono utilizzati per organizzare i dati in maniera gerarchica e sono basati sulla relazione causale padre–figlio
- ❖ Gli alberi possono avere diversa *arietà*, cioè, da ciascun nodo, possono dipartirsi al più un numero m di archi (alberi m –ari)



numero medio figli di ciascun nodo

albero ternario

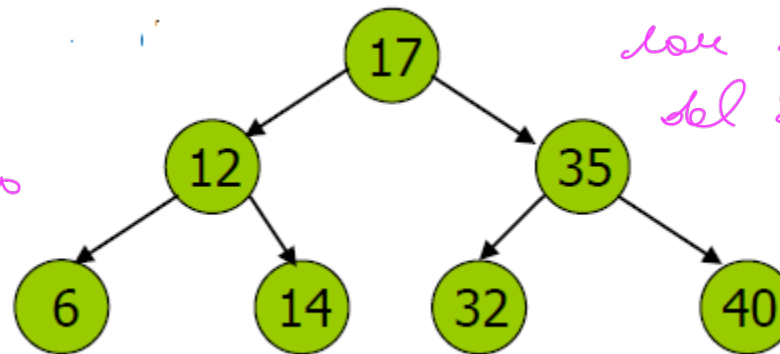
- ❖ Se ogni padre ha, al più, due figli l'albero si dice **binario**





Strutture dati del kernel – 5

- UNIX usa Algoritmi binari di ricerca per scegliere proprio poco*
- ❖ Sono **alberi binari di ricerca** quelli per cui vale la relazione d'ordine figlio_{sinistro} < padre < figlio_{destro}
 - Nel caso peggiore la ricerca costa $\mathcal{O}(n)$, ma, a patto di mantenere l'albero "bilanciato" il costo medio di questa operazione scende a $\mathcal{O}(\log_2 n)$
- {17,35,12,14,32,40,6}*



Per avere $\mathcal{O}(1)$ devo avere un puntatore alla foglia a S_x , posso processare ed essere esibito ... verrà spiegato poi

Sceglia minimo processo che dipende dal carico del sistema, ma da quanti processi sono nel sistema

- Linux utilizza un albero binario di ricerca bilanciato nel suo algoritmo di scheduling della CPU





Strutture dati del kernel – 6

Struttura dati più veloce?

- ❖ **Problema** — Memorizzare in maniera opportuna un insieme di dati, tipicamente sotto forma di record, in modo da poter reperire un qualsiasi elemento dell'insieme con un numero "piccolo" di tentativi

- Cosa significa "piccolo"? *non dipende da N*
 - ▶ Indipendente (o quasi) dalla dimensione della tabella su cui si effettua la ricerca, quindi con una complessità in tempo pari a $O(1)$ *in un vettore, se lo indicavi, $O(1)$, ricerca (N) in un albero $O(\log_2 N)$*

- ❖ **Funzioni hash** *ricerca costa $O(1)$*

- Tabelle hash più performanti*
Richiede indice della Tabella hash
 - $h: K \rightarrow \{0, 1, 2, \dots, n-1\}$ (*to hash*, in inglese, significa sminuzzare) *Calcolatore hash sulla chiave*
 - ▶ K : insieme dei valori distinti che possono essere assunti dalle chiavi dei record *o che viene nella tabella con le chiavi*
 - ▶ n : dimensione del vettore in cui si intende memorizzare la tabella
- Problema: collisione*

Problema del complesso, finire casuale



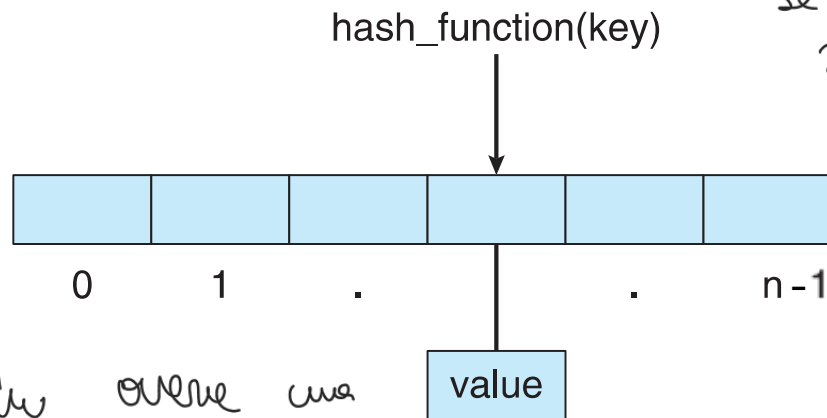


Strutture dati del kernel – 7

❖ Ipotesi: K sottoinsieme dei numeri naturali F. Ho più come

⇒ Possibile funzione di accesso: più facile, modulo della
chiave modulo n
$$h(k) = k \text{ MOD } n, k \in K$$
 diversità della Tabella

- Valore della funzione sempre compreso fra 0 e $n-1$



*Se ho 23 elementi in una
tabella di 365, ho il
50% di avere collisione,
hash map
so el. 57%.*

*e' difficilissimo avere una
funzione di hash*





Strutture dati del kernel – 8

- ❖ ^{dominio} Se K non è un sottoinsieme dei numeri naturali
 - **Esempio:** insieme di stringhe alfanumeriche
 - La funzione hash si applica a numeri
- ⇒ Per utilizzarla in corrispondenza di una chiave non numerica occorre associare alla chiave un valore numerico
- ❖ Necessità di definire funzioni hash generali
 - Associazione di un valore numerico ad una chiave di qualunque tipo
 - Applicazione della funzione hash a tale valore
 - **Esempio:** si utilizza la somma dei codici ASCII dei caratteri che costituiscono la stringa





Strutture dati del kernel – 9

- ❖ **Collisione:** associazione, da parte di una trasformazione hash, della stessa posizione a chiavi distinte
- ❖ *Le chiavi sono dette **sinonime***
- ❖ **Esempio:**
[10,12,20,23,27,30,31,39,42,44,45,49,53,57,60]
 - $h(\text{chiave}) = \text{chiave} \text{ MOD } 15$
 - ▶ Posizione 0 $\leftarrow$ 30, 45, 60
 - ▶ Posizione 8 $\leftarrow$ 23, 53
 - ▶ Posizione 12 $\leftarrow$ 12, 27, 42, 57
- ❖ Ciascuna posizione dell'array può contenere al più un elemento; occorre:
 - Ridurre il più possibile le collisioni
 - Gestire quando si verificano





Strutture dati del kernel – 10

- ❖ Funzioni di *hashing perfetto* (che evitano i duplicati) sono difficili da trovare anche per tabelle grandi — cioè accettando un notevole spreco di memoria
- ❖ **Esempio:** *paradosso del compleanno*
Dato un gruppo di 23 persone, ci sono più del 50% di probabilità che due di esse siano nate nello stesso giorno dell'anno
 - ⇒ In altre parole, se scegliamo una funzione aleatoria che trasforma 23 chiavi in un indirizzo di una tabella di 365 elementi, la probabilità che due chiavi NON collidano è solo 0.4927 (meno della metà)
- ❖ Individuare una funzione di accesso che porti ad un numero ridotto di collisioni è un problema complesso





Strutture dati del kernel – 11

- ❖ Tuttavia... numero di collisioni ridotto drasticamente se accettiamo uno spreco del 20% di memoria extra
- ❖ **Esempio:** array di 19 elementi (indicizzati da 0 a 18)

- ▶ Posizione 0 $\leftarrow$ 57
- ▶ Posizione 8 $\leftarrow$ 27
- ▶ Posizione 1 $\leftarrow$ 20, 39
- ▶ Posizione 10 $\leftarrow$ 10
- ▶ Posizione 3 $\leftarrow$ 60
- ▶ Posizione 11 $\leftarrow$ 30, 49
- ▶ Posizione 4 $\leftarrow$ 23, 42
- ▶ Posizione 12 $\leftarrow$ 12, 31
- ▶ Posizione 6 $\leftarrow$ 44
- ▶ Posizione 15 $\leftarrow$ 53
- ▶ Posizione 7 $\leftarrow$ 45

0	57
1	20, 39
2	
3	60
4	23, 42
5	
6	44
7	45
8	27
9	
10	10
11	30, 49
12	12, 31
13	
14	
15	53
16	
17	
18	

- ❖ Collisioni non eliminate del tutto





Strutture dati del kernel – 12

*dovno' usare chiave con merico ed un vero
Se la chiave d'inverso porta il nome? dei Barbari
in un vero intero. Tecnica per risolvere*

- ❖ Uso di liste concatenate destinate alla memorizzazione degli elementi che, in inserimento, hanno portato ad una collisione

*o) hash, HASHING LINEARE,
se provo collisione vedo
il successo finché
non provo l'altro
ricerca $O(n)$ con peffione*

- ❖ Ricerca di un elemento di chiave k

LISTE di trabocco

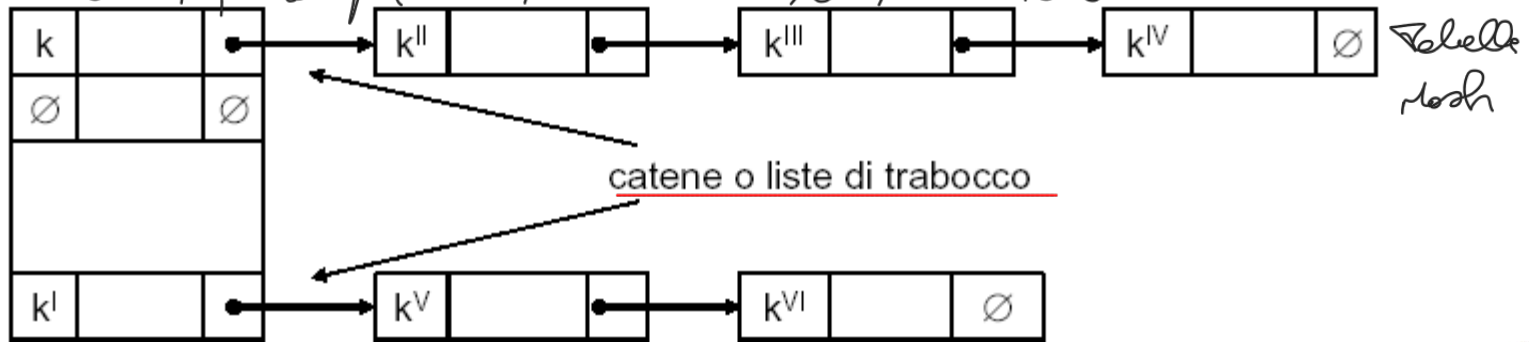
attacco a ■ Si calcola $h(k)$

posto una lista

► Se si verifica una collisione allora si accede alla lista associata alla posizione $h(k)$ e la si scandisce

*Ricerca con
minore di $O(n)$*

casuale indipendente da N , del vero si cerca nella





Strutture dati del kernel – 13

*Hash per influenzare più grande pena di sicurezza.
per ID, numero, per fare l'accuso LA CHIAVE è l'USER ID della tabella HASH*

❖ Il costo dell'operazione di ricerca — realizzata in modo lineare relativamente alle liste di elementi in collisione — si mantiene pressoché indipendente da n (numero degli elementi contenuti nella tabella) *nella Tabella la Piv è contraddistorta*

❖ Nei sistemi operativi, per l'autenticazione si procede come segue:

- L'utente inserisce username e password
- Viene applicata la funzione hash allo username (cui corrisponde un ID utente numerico) in modo da recuperare la password memorizzata in tabella
- La password viene confrontata con quella inserita dall'utente per l'autenticazione





Strutture dati del kernel – 14

Struttura dati, soprattutto su dischi piccoli

- ❖ Una **bitmap** è una stringa di n caratteri binari utilizzabile per rappresentare lo stato di n elementi

non credo esista per PC, serve di
 LUT... quel particolare elemento 001011101... lì si dice se spazio serve
 o USABILE o meno? o no

- ❖ Efficienza in termini di spazio utilizzato
- ❖ Nei sistemi operativi, la gestione dello spazio sui primi dischi magnetici offriva un ottimo esempio di utilizzo di bitmap

disco diviso in blocchi (pagine)

- Un partizione di disco di medie dimensioni può essere divisa in diverse migliaia di unità, dette **blocchi** (dim. media: 4KB)
- Si utilizza una bitmap per indicare la disponibilità o meno di ognuno di essi

quanto sarebbe grande la map con blocchi di 4k?
 ha senso tenerla in RAM? no.





Ambienti di calcolo – 1

$$\begin{aligned} \text{es. di } 2^{26} \text{ GB} &= 2^{30} / 2^{12} \\ &= 2^{26} = 64 \text{ MByte} \\ &\text{no.} \end{aligned}$$

❖ Elaborazione tradizionale

- Con l'evoluzione tecnologica, i confini tra i diversi ambienti di elaborazione tradizionali diventano sempre più sfumati
- Negli uffici...
 - ▶ Fino a non molti anni fa: diversi PC connessi in rete, con server per servizi di accesso ai file e per stampa
 - ▶ Ora: *portali* che permettono l'accesso alla rete Internet e a risorse allocate su server interni e sistemi remoti tramite dispositivi mobili (e *thin client*)
 - ▶ Facilità di sincronizzazione con i dispositivi mobili per mezzo di reti wireless
- Nelle case...
 - ▶ Fino a non molti anni fa: un PC connesso alla rete via modem
 - ▶ Ora: piccole reti locali, connessioni veloci a Internet, barriere anti-intrusione (*firewall*)





Ambienti di calcolo – 2

❖ Mobile computing

Strumenti. sarebbe difficile usare lo desktop e ci

- Ad oggi, le funzionalità disponibili sui dispositivi mobili si sono talmente arricchite da rendere indistinguibili computer portatili e tablet *come GS*
- Talvolta le funzioni di un moderno dispositivo mobile non sono disponibili, o non sono facilmente realizzabili, su computer desktop o portatili
 - ▶ Posta elettronica ed accesso al Web
 - ▶ Accesso/creazione a/di dati multimediali (musica, foto e video in alta definizione, ebook)
 - ▶ Presenza di GPS, accelerometro e giroscopio: rotazione dello schermo, navigazione, giochi, augmented reality *o cellulare come il livello*
- Accesso ai servizi online per mezzo dello standard wireless IEEE 802.11 o tramite reti cellulari

standard





Ambienti di calcolo – 3

❖ Mobile computing

- Anche la capacità di memoria e la velocità dei processori si stanno sempre più avvicinando a quelle dei portatili, anche se si mantiene l'attenzione al risparmio energetico

▶ **Esempio:** Nell'iPhone 14 Pro, processore A16 Bionic con CPU con 6 core — 2 power core Avalanche a 3.46 GHz e 4 efficiency core Blizzard a 2.02 GHz —, GPU con 5 core, Neural Engine con 16 core, 6 GB di RAM, 1 TB di memoria

- Mercato dei SO dominato da **Apple iOS** e **Google Android**

ottimizzato per HW specifico

più portatile





Ambienti di calcolo – 4

❖ Sistemi distribuiti *più calcoli Loose coupling*

*CONDIVIDONO
solo MEMORIA
MASSIVA
SAN*

- Un sistema distribuito è un insieme (eterogeneo) di calcolatori che non condividono né la memoria né il clock; ciascun processore ha la sua propria memoria locale
- I computer nel sistema sono connessi attraverso una rete di comunicazione
 - ▶ Wide Area Network (WAN) *wide Area network*
 - ▶ Metropolitan Area Network (MAN)
 - ▶ Local Area Network (LAN)
 - ▶ Personal Area Network (PAN)
- La comunicazione avviene secondo un dato protocollo (TCP/IP è il più diffuso)
- Un sistema distribuito fornisce agli utenti l'accesso a varie risorse di sistema
- L'accesso a risorse condivise consente di:
 - ▶ Accelerare l'elaborazione *es. codice finito in secondi*
 - ▶ Aumentare la disponibilità di dati *→ più STORAGE*
 - ▶ Migliorare l'affidabilità

GRACEFUL DEGRADATION



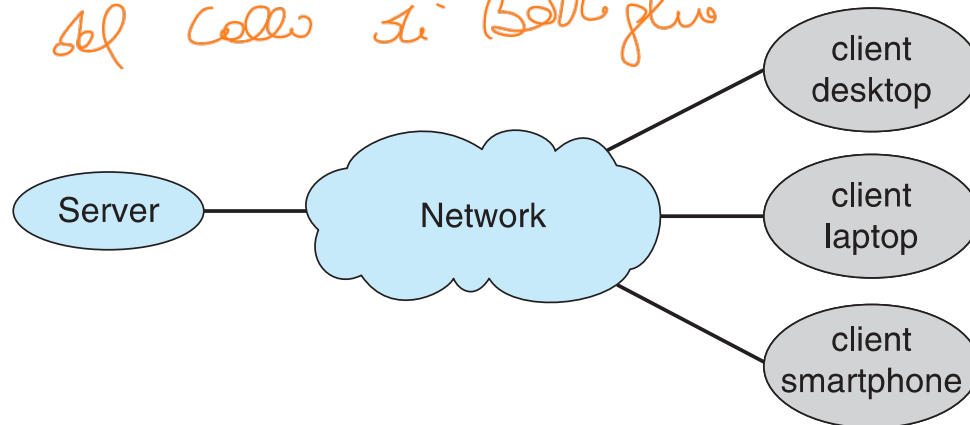


Ambienti di calcolo – 5

❖ Modello client-server *Più client che fanno rif. a uno server*

- Terminali e mainframe soppiantati da PC e server
- I PC fungono da **client** e richiedono servizi a server
- I **server** permettono l'accesso a servizi e risorse (i.e., stampa, memorizzazione e reperimento di file, database, accesso alla rete Internet) *Server*
 - ▶ Server elaborativi — per esempio, un server che ospita una base di dati e risponde a interrogazioni su di essa
 - ▶ File server — per esempio, un server Web che fornisce pagine ai browser dei client *e i client ne fanno DOWNLOAD*

↳ PROBLEMA del collo di bottiglia



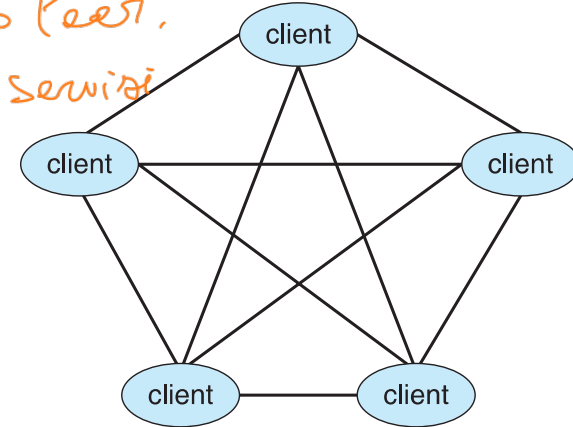


Ambienti di calcolo – 6

❖ Modello peer-to-peer

Tutti i computer sono Peer, possono offrire e richiedere servizi.

- Un modello diverso di sistema distribuito
- P2P non distingue client e server, ma è costituita da *nodi equivalenti* (**peer**, appunto) che fungono sia da client che da server verso altri nodi della rete
- Ogni nodo deve connettersi ad una rete P2P...
 - ▶ ...registrando i propri servizi in un registro centralizzato di consultazione della rete
 - ▶ o inoltrando le proprie richieste di servizio a tutti gli altri nodi in broadcast (utilizzando un protocollo di scoperta)
- Esempi storici sono **Napster** (registro centralizzato) e **Gnutella** (protocollo di scoperta) e, più recentemente, i servizi **Voice over IP (VoIP)** come **Skype**



es. NAPSTER, registro centralizzato, si fa riferimento ad un PC centralizzato per smistare dati e servizi

registro centralizzato e protocollo di scoperta





Cloud computing – 1

Cloud: serie di servizi disponibili dal / HW, applicos. rende facendo riferimento al HW remoto

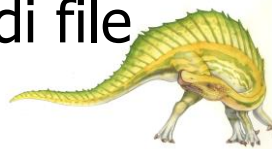
- ❖ La diffusione della connettività Internet, in mobilità e wi-fi, e il numero sempre maggiore di smartphone e tablet hanno contribuito al recente successo dei **servizi di cloud computing**
- ❖ L'espressione, che in italiano potrebbe tradursi come "nuvola informatica", si riferisce alle piattaforme e alle tecnologie che permettono di archiviare file (documenti, immagini, video, musica) o di sviluppare/utilizzare programmi e applicazioni direttamente sui server di chi fornisce il servizio, anziché sul proprio dispositivo





Cloud computing – 2

- ❖ Pertanto, il **cloud computing** è una tecnica che permette la fruizione di risorse computazionali, di storage e di applicazioni come servizi di rete
 - Non si conservano più i file direttamente su memorie locali, ma su server remoti
 - Ugualmente dicasi per i programmi: anziché installarli, possono essere usati direttamente online
- ❖ È un'estensione logica della virtualizzazione perché utilizza la virtualizzazione come strumento base per offrire le proprie funzionalità
 - **EC2** (*Elastic Compute Cloud*) di Amazon dispone di migliaia di server, milioni di macchine virtuali e petabyte di spazio dati accessibili via Internet
 - **Google Drive** è la suite Office dell'azienda di Mountain View, permette l'editing e l'archiviazione di vari tipi di file
 - **Dropbox, OneDrive, iCloud...**





Cloud computing – 3

❖ Tipologie di cloud computing

- **Cloud pubblico:** disponibile attraverso Internet a chiunque si abboni al servizio
- **Cloud privato:** gestito da un'azienda ad uso interno
- **Cloud ibrido:** comprende componenti sia pubbliche che private

per insieme ristretto di persone

con più potenza





Cloud computing – 4

❖ Tipologie di servizi

- **SaaS** (*Software as a Service*): ^{es word} una o più applicazioni fruibili via Internet (es.: word processor, fogli di calcolo)
- **Daas** (*Data as a Service*): disponibilità di dati, ai quali gli utenti possono accedere tramite qualsiasi applicazione, come se fossero residenti su un disco locale
- **Haas** (*Hardware as a Service*): computer messi a disposizione a utenti che inviano dati e ricevono risultati
- **PaaS** (*Platform as a Service*): ambiente software predisposto per usi applicativi via Internet (es.: database server) ^{piattaforma di sviluppo per software}
- **IaaS** (*Infrastructure as a Service*): server o storage disponibili attraverso Internet (es.: dispositivi per backup)

Accesso e
utilizzo
in remoto

→ STORAGE, cloud come memoria Terabyte
memoria permanente info da storage





Cloud computing – 5

- ❖ Gli ambienti di cloud computing inglobano SO tradizionali, VMM per le macchine virtuali, oltre agli strumenti specifici per la gestione del cloud

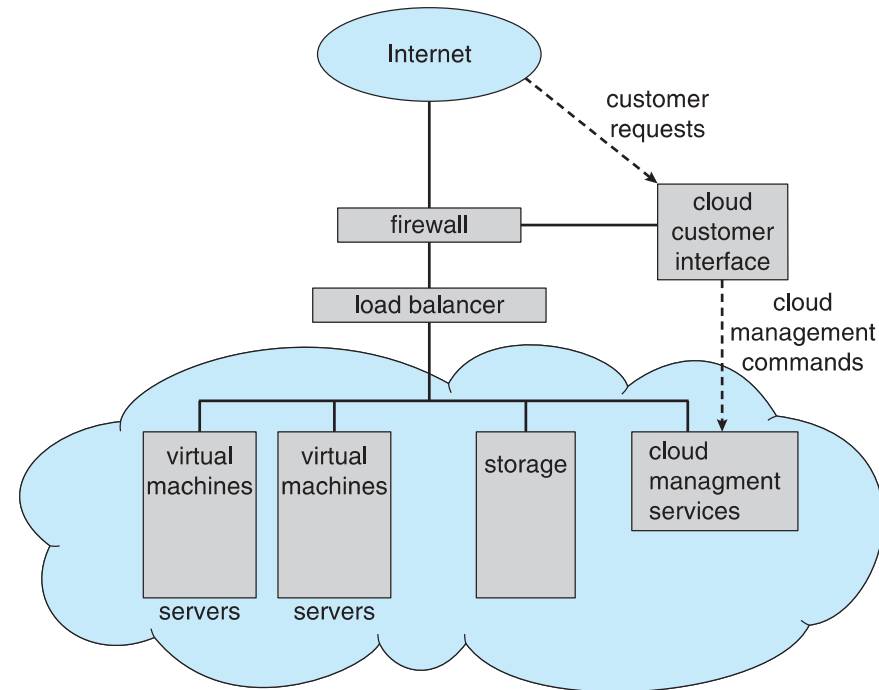
- VMware vCloud Director
- Eucalyptus (open-source)

- ❖ Inoltre...

- La connettività Internet impone l'uso intensivo di dispositivi di sicurezza (firewall) e di bilanciatori di carico per lo smistamento efficiente delle applicazioni

l'accesso
del cloud
deve essere
firewall per
garantire sicurezza,
però attento.

VMM e SO sono virtualizzatori delle macchine
che controllano il SO.





Sistemi embedded – 1

- ❖ Rappresentano la tipologia dominante di elaboratore: sono ubiquitari (elettrodomestici, motori delle auto, robot industriali,...) *varante hanno necessita' di comunicare con l'utente*
- ❖ Normalmente installati su sistemi rudimentali, hanno compiti molto precisi e offrono funzionalità limitate e interfaccia utente scarsamente sviluppata (se presente)
- ❖ I sistemi embedded sono caratterizzati da una grande variabilità
 - Elaboratori general-purpose con sistemi operativi standard che sfruttano applicazioni create appositamente per implementare una particolare funzionalità *es. solo un programma in assembly*
 - Microprocessori dedicati con SO special-purpose
 - Application Specific Integrated Circuit (ASIC) privi di SO *hanno colto nell'HW le operazioni da eseguire*

• Embedded, no interfaccia utente





Sistemi embedded – 2

versione HW, grande variabilità di S.O.

per fare un S.O. devo scegliere architettura Target e scopi del S.O.

❖ La tipologia più diffusa per gli embedded sono i sistemi operativi real-time

- Vincoli definiti a priori sui tempi di esecuzione
- L'elaborazione si considera portata a termine correttamente solo se il compito viene eseguito entro i limiti prestabiliti (hard real-time):
- In effetti, i vincoli sui tempi di risposta sono così stringenti che, se non vengono rispettati, il sistema di controllo è inutile se non dannoso

cresce task completion

▪ Esempi:

entro un DEADLINE è stato eseguito

▶ Programmi di controllo delle superfici di volo di un aereo

▶ Robot per la costruzione di autovetture

*grazie alle
priorità perfette
in orario reale
macchina*

*soft-real time: considerare dei pesanti indicatori con maggior priorità
(Slot Temp. CPU più lunghi)*

• *Hard Real Time: tempo scheduling diverse TASK PORTATO a termine se
fatto in un determinato LASSO 1.107 di Tempo*





Sistemi operativi open-source – 1

- ❖ Disponibili in formato sorgente anziché come codice binario compilato *su quello, posso intervenire e poter capire cosa fa quel SW*
- ❖ **Vantaggi**
 - Non è necessario effettuare il reverse engineering per comprendere il funzionamento del sistema *tentare di risolvere al linguaggio di alto livello*
 - ▶ Operazione che non consente, ad esempio, di ricostruire i commenti integrati al codice
 - Costituzione di una comunità di programmatori e aziende che contribuiscono allo sviluppo, al debugging, all'assistenza e al supporto (spesso gratuito) agli utenti
 - Codice più sicuro e bug scoperti e risolti velocemente
 - ▶ Molti più «occhi puntati sul codice»
 - Libertà *maggiore garanzia di sicurezza*





Sistemi operativi open-source – 2

- ❖ Qualche problema... *difficile mettere barriere*
 - ...con la gestione dei diritti digitali (*DRM – Digital Rights Management*): limiti imposti facilmente violabili, modificando il codice aperto
 - Le leggi di molti paesi, incluso il *Digital Millennium Copyright Act* (DMCA) in USA, rendono illegale recuperare un codice DRM per provare a eludere la protezione di materiale digitale
 - I SO chiusi (e altri software) possono invece implementare blocchi di accesso a contenuti protetti da copyright





Sistemi operativi open-source – 3



❖ Come nascono...

- 1983: Richard Stallman dà vita al progetto **GNU** (è un acronimo ricorsivo che significa GNU is Not Unix), con la finalità di creare un SO gratuito, open-source e compatibile con UNIX

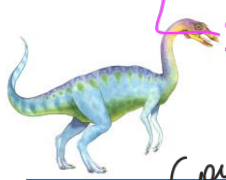
- **Copyleft** (1984)



- ▶ Il concetto di **Copyleft** nacque quando Richard Stallman stava lavorando ad un interprete Lisp
- ▶ L'azienda Symbolics chiese di poter utilizzare l'interprete e Stallman accettò di fornire una versione di pubblico dominio
- ▶ Symbolics estese e migliorò l'interprete Lisp, ma quando Stallman volle accedere ai miglioramenti, Symbolics rifiutò

*Codice copyleft, ossia un eseguibile.
codice con i commenti: interprete legge i commenti e la produce, viene distribuito
il sorgente*





Sistemi operativi open-source – 4

- Capilobes, 3 nomi: analisi fonetica (TO KEN I Z A N E), sintattica (e frase) e morfologica, oppure sintattica, analisi semantica, parola singola.*
- L'espressione inglese copyleft è un gioco di parole sul termine *copyright* nel quale la parola "right" significa "diritto" (in senso legale), ma giocando sul suo secondo significato (ovvero "destra") viene scambiata con "left" ("sinistra", ma anche "lasciato")
 - Il copyleft è il "permesso d'autore": individua un modello di gestione dei diritti d'autore basato su un sistema di licenze attraverso le quali l'autore (in quanto detentore originario dei diritti) indica ai fruitori dell'opera che essa può essere utilizzata, diffusa e modificata liberamente, nel rispetto di alcune condizioni essenziali
 - Nella versione originaria del copyleft, la condizione principale obbliga i fruitori dell'opera, nel caso vogliano distribuire l'opera modificata, a farlo sotto lo stesso regime giuridico (e generalmente sotto la stessa licenza)
 - In questo modo, il regime di copyleft e tutto l'insieme di libertà da esso derivanti sono sempre garantiti

licenza d'autore GPL: chi usa SW libero, ne usa solo allo stesso modo





Sistemi operativi open-source – 5

- ❖ 1985: Viene pubblicato il Manifesto GNU che sostiene che tutti i software dovrebbero essere gratuiti ed open-source; si costituisce la FSF, **Free Software Foundation**, con lo scopo di incoraggiare il libero scambio di sorgenti ed il libero utilizzo del software
 - La FSF distribuisce il software con licenza **GPL** (General Public License)
 - GPL presuppone la distribuzione congiunta di sorgente e binario ed impone che qualsiasi cambiamento apportato al sorgente sia reso disponibile con licenza GPL
 - Anche la licenza **Creative Commons** “Attribution Sharelike” è una licenza copyleft





Sistemi operativi open-source – 6



UNIX per macellaio
personale

Usando strumenti di standard, Kernel GNU con 16

- 1991: Linus Torvalds rilascia un kernel rudimentale, simile a UNIX, utilizzando compilatori, librerie, strumenti di GNU – nasce **LINUX** (versione 1.0, 13 marzo 1994)
 - ▶ Moltissime distribuzioni: **Red Hat**, **SUSE**, **Fedora**, **Debian**, **Slackware**, **Ubuntu**, **Sabayon**, etc.
 - ▶ L'ultima versione del kernel Linux è la 6.2.5 ed è stata rilasciata l'11 marzo 2023 (5.0 rilasciata il 4 marzo 2019, 6.0 rilasciata il 4 ottobre 2022)

“Il cambio della numerazione da 4.x a 5 non implica che è in arrivo qualcosa di speciale. Se volete avere un motivo ufficiale diciamo che ho finito le dita delle mani e dei piedi per contare, così il kernel 4.21 è diventato 5.0”





Sistemi operativi open-source – 7

- Attualmente: il progetto GNU ha prodotto molti strumenti compatibili con UNIX, quali compilatori, editor e varie utility, ma non ha mai prodotto (e rilasciato) un kernel «definitivo»
 - ▶ **Hurd**, il kernel GNU, è in sviluppo, ma è ancora lontano dall'essere pronto all'uso quotidiano
 - ▶ Utility e programmi di sistema GNU utilizzati con kernel LINUX: **GNU/LINUX**
 - ▶ Debian GNU/Hurd 2021 Bullseye, 14.08.2021

“The latest releases are GNU Hurd 0.9, GNU Mach 1.8, GNU MIG 1.8, 18.12.2016. The Hurd is developed by a few volunteers in their spare time. The project welcomes any assistance you can provide. Porting and development expertise is still badly needed in many key areas.”





Sistemi operativi open-source – 8

- ❖ Il software libero e il software open-source sono due idee “filosoficamente” distinte, sostenute da gruppi di persone diverse

<https://www.gnu.org/philosophy/philosophy.en.html>

- Il software libero non solo rende il codice sorgente disponibile, ma è anche dotato di una licenza che ne consente l'uso, la redistribuzione e la modifica senza costi
- Il software open-source non offre necessariamente tale licenza
- ❖ Linux è il SO open-source più famoso, con alcune distribuzioni libere e altre solo open-source
- ❖ Windows è proprietario e closed-source
- ❖ Mac OS è ibrido: contiene il kernel Darwin open-source, ma include anche componenti chiuse e proprietarie





Sistemi operativi open-source – 9

❖ UNIX BSD

- Derivato da UNIX di AT&T, risale al 1978, e le sue prime versioni furono rilasciate dalla UCB, in codice sorgente e in binario, ma non open-source (necessaria la licenza AT&T)
- Sviluppo rallentato dalla querela di AT&T: versione completa open-source del sistema (4.4BSD-lite) nel 1994
- Varie distribuzioni:
 - ▶ FreeBSD, NetBSD, OpenBSD, DragonflyBSD
- **Darwin**, il componente kernel fondamentale di Mac OS e di iOS, è basato su UNIX BSD ed è open-source (consultare <https://opensource.apple.com/>)





Sistemi operativi open-source – 10

❖ Solaris *since genl. workstation Sun*

- Sviluppato, a partire dal 1991, da Sun Microsystems come derivato di UNIX System V di AT&T
- Nel 2005 la Sun, nell'ambito del progetto OpenSolaris, rese disponibile parte del SO, continuando costantemente ad evolvere il codice open-source
- Ad oggi, tuttavia, parte del codice è ancora di proprietà AT&T e, dal 2009, con l'acquisto di Sun da parte di Oracle, non vi sono stati ulteriori sviluppi "ufficiali"
- Solaris può comunque essere compilato da sorgente e collegato (tramite linker) al codice binario delle componenti a codice chiuso
- **Illumos**, portato avanti da gruppi di sviluppatori liberi sulla base di Solaris, ne ha comunque esteso le funzionalità





Impatto didattico dell'open-source – 1

- ❖ Il movimento a sostegno dei software gratuiti spinge legioni di programmatori a creare migliaia di progetti open-source, correlati anche allo sviluppo di SO
- ❖ Ultimo aspetto etico legato al software libero è quello che i progetti open-source permettono di...
 - ...utilizzare il codice sorgente come strumento di apprendimento, con l'opportunità di modificarlo, testarlo, contribuire all'individuazione/eliminazione di errori, customizzarlo
 - ...esplorare SO già maturi e completi, con i relativi compilatori, programmi di sistema, interfacce utente, applicazioni





Impatto didattico dell'open-source – 2

There has never been a more interesting time to study operating systems, and it has never been easier. The open-source movement has overtaken operating systems, causing many of them to be made available in both source and binary (executable) format. The list of operating systems available in both formats includes Linux, BSD UNIX, Solaris, and part of Mac OS. The availability of source code allows us to study operating systems from the inside out. Questions that we could once answer only by looking at documentation or the behavior of an operating system we can now answer by examining the code itself.

Operating systems that are no longer commercially viable have been open-sourced as well, enabling us to study how systems operated in a time of fewer CPU, memory, and storage resources. An extensive but incomplete list of open-source operating system projects is available from https://curlie.org/Computers/Software/Operating_Systems/Open_Source/

In addition, the rise of virtualization as a mainstream (and frequently free) computer function makes it possible to run many operating systems on top of one core system. For example, VMware (<http://www.vmware.com>) provides a free “player” for Windows on which hundreds of free “virtual appliances” can run. Virtualbox (<http://www.virtualbox.com>) provides a free, open-source virtual machine manager on many operating systems. Using such tools, students can try out hundreds of operating systems without dedicated hardware.

The advent of open-source operating systems has also made it easier to make the move from student to operating-system developer. With some knowledge, some effort, and an Internet connection, a student can even create a new operating system distribution. Just a few years ago, it was difficult or impossible to get access to source code. Now, such access is limited only by how much interest, time, and disk space a student has.



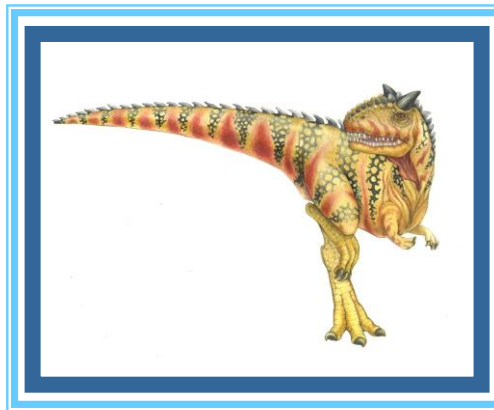


Suggerimenti

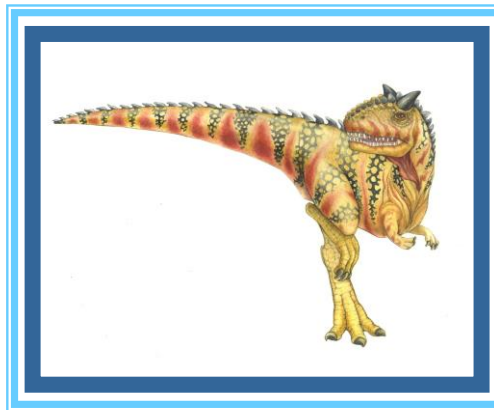
- ❖ Per eseguire Linux su un altro sistema operativo:
 - Scaricare il software di virtualizzazione gratuito Virtualbox VMM tool all'indirizzo www.virtualbox.org e installarlo sul proprio sistema
 - Scegliere da virtualboxes.org/images/ un'immagine del sistema operativo preconfigurata
 - Avviare la macchina virtuale all'interno di Virtualbox



Fine del Capitolo 1



Struttura dei Sistemi Operativi





Obiettivi

Compiti del SO per la gestione Processi

- ❖ Identificare i servizi forniti da un sistema operativo
- ❖ Illustrare come si utilizzano le chiamate di sistema per ottenere servizi dal SO
- ❖ Confrontare strategie monolitiche, stratificate, microkernel, modulari e ibride per la progettazione di sistemi operativi
- ❖ Illustrare il processo di generazione di un sistema operativo
- ❖ Utilizzare strumenti di monitoraggio delle prestazioni





Sommario

- ❖ Servizi del sistema operativo *orientati all'utente e loro gestione risorse (sistemi)*
- ❖ Interfaccia utente
- ❖ Chiamate di sistema *→ fork, per creare processi*
- ❖ Programmi di sistema
- ❖ Linker e loader
- ❖ Progettazione e realizzazione
- ❖ Struttura del sistema operativo
- ❖ Generazione e avvio del sistema operativo
- ❖ Debugging del sistema operativo





Introduzione

ambiente SW che rende facile sviluppare e eseguire programmi

- ❖ I sistemi operativi forniscono l'ambiente in cui si eseguono i programmi *e anche SDC*
 - Essendo organizzati secondo criteri che possono essere assai diversi (e dovendo "girare" su architetture molto differenziate), tale può essere anche la loro struttura interna *HW variabili, SO difficili da implementare*
- ❖ La progettazione di un nuovo SO è un compito difficile
 - ⇒ il tipo di sistema desiderato definisce i criteri di scelta dei metodi e gli algoritmi implementati
- ❖ In fase di progettazione, il sistema operativo può essere definito/valutato in base a...
 - ...i servizi che esso dovrà fornire *facile accesso all'HW sottostante*
 - ...l'interfaccia messa a disposizione di programmatori e utenti
 - ...la complessità di realizzazione





Servizi del sistema operativo – 1

❖ Servizi user-oriented *utente facile e accogliente*

1° servizio offerto:

■ **Interfaccia utente** — tutti gli attuali SO general-purpose sono dotati di un'interfaccia utente, a linea di comando (*Command Line Interface, CLI*) e grafica (*Graphic User Interface, GUI*), o touch-screen *interfacci con cui ci si rivolge al S.O.*

shell, interprete

*lo maggior intras.
o in relazione*

*o il FILESYSTEM
nella memoria
secondaria*

■ **Esecuzione di programmi** — capacità di caricare un programma in memoria ed eseguirlo, eventualmente rilevando, ed opportunamente gestendo, situazioni di errore *Programmi e processi sono sempre stored in memoria, intercon.*

■ **Operazioni di I/O** — il SO fornisce ai programmi utente i mezzi per effettuare l'I/O su file o periferica *con un FILE*

■ **Gestione del file system** — capacità dei programmi di creare, leggere, scrivere e cancellare file e muoversi nella struttura delle directory





Servizi del sistema operativo – 2

❖ Servizi user-oriented (cont.)

servizi meno percepiti

- **Comunicazioni** — scambio di informazioni fra processi in esecuzione sullo stesso elaboratore o su sistemi diversi connessi via rete

ogni processo ha il suo spazio di indirizzi ma se comunico ci servono aree di memoria a cui accedono più processi

Comunicazione tra processi, in 2 modi, per mezzo di memoria condivisa oppure tramite scambio di messaggi

- ▶ Le comunicazioni possono avvenire utilizzando **memoria condivisa** o **scambio di messaggi**

che es. si trovano nella rete o nella stessa macchina tramite condivisione memoria

Rilevamento di errori — il SO deve tenere il sistema di calcolo sotto controllo costante, per rilevare errori, che possono verificarsi nella CPU e nella memoria, nei dispositivi di I/O o durante l'esecuzione di programmi utente

- ▶ Per ciascun tipo di errore, il SO deve prendere le opportune precauzioni per mantenere una modalità operativa corretta e consistente
- ▶ I servizi di debugging possono facilitare notevolmente la programmazione e, in generale, l'interazione con il sistema di calcolo

deve esserci un messaggio di errore

errori HW, SW





Servizi del sistema operativo – 3

S.O. come gestore risorse

- ❖ Esistono funzioni aggiuntive atte ad assicurare l'efficienza del sistema (non esplicitamente orientate all'utente)

- **Allocazione di risorse** — quando più utenti o più processi vengono serviti in concorrenza, le risorse disponibili devono essere allocate equamente ad ognuno di essi

- **Accounting e contabilizzazione dell'uso delle risorse** — tener traccia di quali utenti usano quali e quante risorse del sistema (utile per ottimizzare le prestazioni del sistema di calcolo)

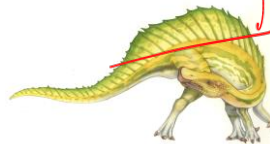
- **Protezione e sicurezza** — i possessori di informazioni memorizzate in un sistema multiutente o distribuito devono essere garantiti da accessi indesiderati ai propri dati; processi concorrenti non devono interferire fra loro

- ▶ **Protezione:** assicurare che tutti gli accessi alle risorse di sistema siano regolamentati

- ▶ **Sicurezza:** si basa sull'obbligo di identificazione tramite *password* e si estende alla difesa dei dispositivi di I/O esterni (es.: scheda di rete) da accessi illegali

*più utenti
alla stessa macchina*

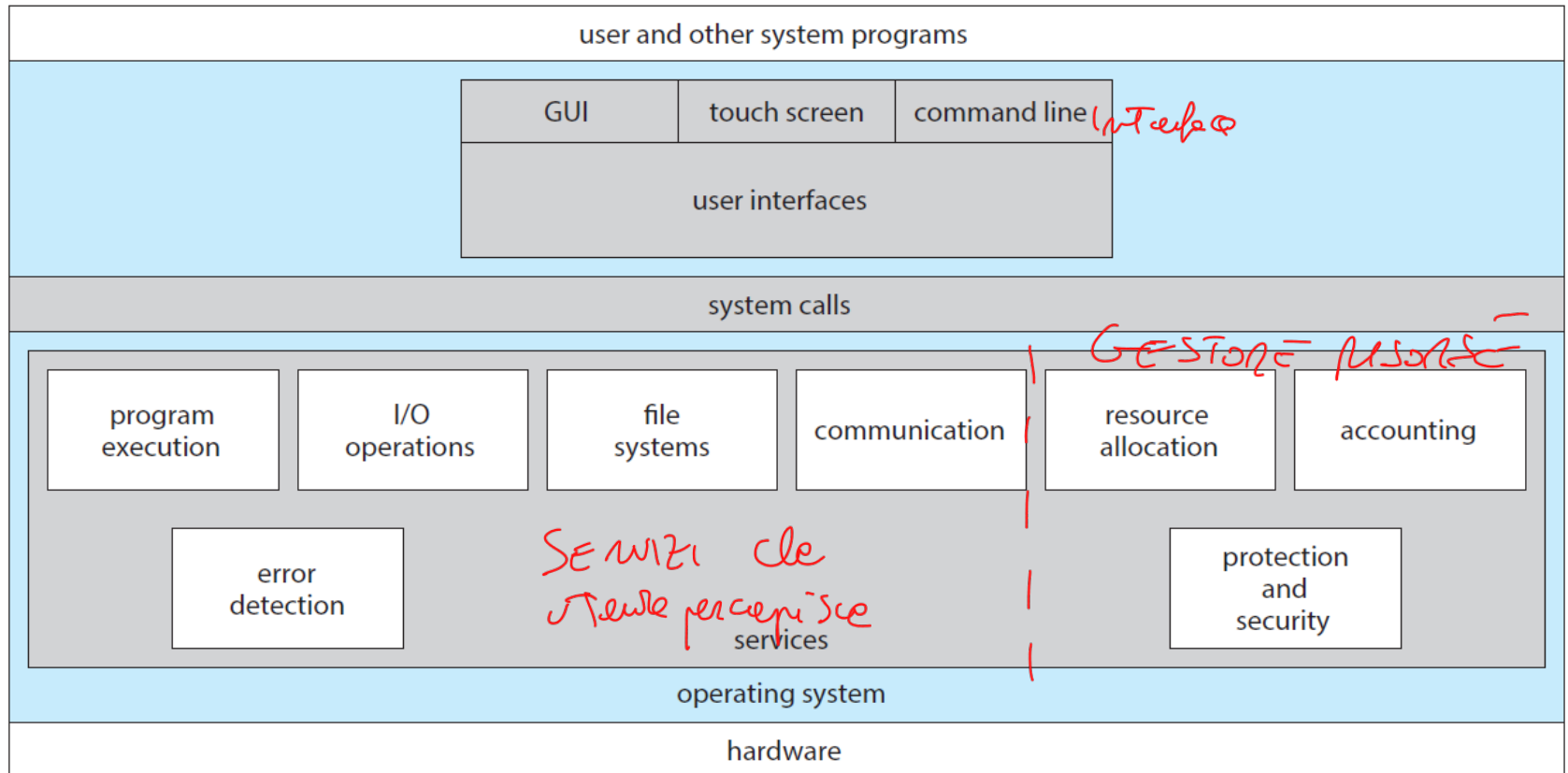
*motore
all'interno
del sistema
e sicurezza,
inaccessibilità
all'esterno*





Panoramica dei servizi del sistema operativo

organism. dei servizi offerti al S.O.





Interfaccia utente CLI

→ *programmazione*

linguaggio CLI

Strutture

= linguaggio S.O.

❖ L'interfaccia utente a linea di comando permette di impartire direttamente comandi al SO (istruzioni di controllo) *Modello classico con cui ci si interfaccia col S.O.*

▪ Talvolta viene implementata nel kernel, altrimenti attraverso programmi di sistema (UNIX/Linux) *CLI = Prog. S.O. in Mod. utente*

▪ Può essere parzialmente personalizzabile, ovvero il SO può offrire più *shell*, più ambienti diversi, da cui l'utente può impartire le proprie istruzioni al sistema *es. più shell*

► Per UNIX/Linux, consultare: *es. Bash, altro con permesso*

<http://www.faqs.org/faqs/unix-faq/shell/shell-differences/>

▪ La sua funzione è quella di interpretare ed eseguire le istruzioni impartite dall'utente (siano esse istruzioni built-in del SO o nomi di eseguibili utente) – *interprete*

dei comandi

interprete

COMANDI





L'interprete dei comandi – 1

❖ I comandi ricevuti dall'interprete possono essere eseguiti secondo due modalità:

- Se il codice relativo al comando è parte del codice dell'interprete, si effettua un salto all'opportuna sezione di codice

*ho lista di tutti i comandi del S.I.
interprete ha chiamato il suo insieme i comandi che il S.I. ha*

⇒ Poiché ogni comando richiede il proprio segmento di codice (passaggio dei parametri e invocazione delle opportune chiamate di sistema), il numero dei comandi implementati determina le dimensioni dell'interprete

- I comandi vengono implementati per mezzo di programmi di sistema *Cerco un eseguibile in una directory in /usr/bin e eseguirlo*

programmi possono scrivere script facili...

⇒ I programmatori possono aggiungere nuovi comandi al sistema creando nuovi file con il nome appropriato (e memorizzandoli nelle directory opportune, es. **/usr/bin**)

Per l'interprete è un programma da compilare e eseguire

⇒ L'interprete dei comandi non viene modificato e può avere dimensioni ridotte *codice di tutti i comandi non è nell'interprete, deve solo riconoscere comandi, trovare programmi e eseguirli con alcuni comandi*





L'interprete dei comandi – 2

- ❖ In molti sistemi, solo un sottoinsieme delle funzionalità è disponibile via GUI e le funzioni meno comuni sono accessibili solo tramite linea di comando
- ❖ Le interfacce CLI: *programmazione, scripting*
 - Semplificano l'esecuzione di comandi ripetuti, perché sono programmabili
 - Un task eseguito di frequente e composto da più comandi può andare a costituire uno script
 - Gli shell script, molto comuni nei sistemi UNIX-like, non vengono compilati, ma interpretati dall'interfaccia a riga di comando





L'interprete dei comandi DOS

```
C:\>dir ectory

Volume in drive C is MS-DOS 5_0
Volume Serial Number is 446B-2781
Directory of C:\
COMMAND.COM, e' il kernel
COMMAND  COM      47845 11-11-91   5:00a
          1 file(s)      47845 bytes
                   10280960 bytes free

C:\>ver versione del DOS

MS-DOS Version 5.00

C:\>
```





L'interprete dei comandi di Solaris

```
Default
New Info Close Execute Bookmarks

PBGMac-Pro:~ pbg$ w
15:24 up 56 mins, 2 users, load averages: 1.51 1.53 1.65
USER      TTY      FROM          LOGIN@   IDLE   WHAT
pbg       console  -             14:34    50     -
pbg       s000    -             15:05    -      w
PBGMac-Pro:~ pbg$ iostat 5
          disk0      disk1      disk10     cpu      load average
      KB/t tps MB/s  KB/t tps MB/s  KB/t tps MB/s  us sy id  1m  5m  15m
      33.75 343 11.30   64.31 14  0.88   39.67  0  0.02  11  5 84  1.51 1.53 1.65
      5.27 320  1.65    0.00  0  0.00    0.00  0  0.00   4  2 94  1.39 1.51 1.65
      4.28 329  1.37    0.00  0  0.00    0.00  0  0.00   5  3 92  1.44 1.51 1.65
^C
PBGMac-Pro:~ pbg$ ls
Applications          Music                  WebEx
Applications (Parallels) Pando Packages        config.log
Desktop               Pictures               getsmartdata.txt
Documents             Public                 imp
Downloads             Sites                  log
Dropbox              Thumbs.db              panda-dist
Library              Virtual Machines       prob.txt
Movies               Volumes                scripts
PBGMac-Pro:~ pbg$ pwd
/Users/pbg
PBGMac-Pro:~ pbg$ ping 192.168.1.1
PING 192.168.1.1 (192.168.1.1): 56 data bytes
64 bytes from 192.168.1.1: icmp_seq=0 ttl=64 time=2.257 ms
64 bytes from 192.168.1.1: icmp_seq=1 ttl=64 time=1.262 ms
^C
--- 192.168.1.1 ping statistics ---
2 packets transmitted, 2 packets received, 0.0% packet loss
round-trip min/avg/max/stddev = 1.262/1.760/2.257/0.498 ms
PBGMac-Pro:~ pbg$
```

Mostra le statistiche per i dischi e la CPU (ogni 5 secondi)

*print working directory - Mi piace dove mi trovo
non'altro che l'it*
*vedo se una cello
in vela e' raggiungibile*

Bourne shell, l'interprete dei comandi utilizzato da Solaris 10





L'interprete dei comandi di Mac OS

```
1. root@r6181-d5-us01:~ (ssh)
root@r6181-d5-us01:~ (ssh)
Last login: Thu Jul 14 08:47:01 on ttys002
iMacPro:~ pbg$ ssh root@r6181-d5-us01
root@r6181-d5-us01's password:
Last login: Thu Jul 14 06:01:11 2016 from 172.16.16.162
[root@r6181-d5-us01 ~]# uptime
06:57:48 up 16 days, 10:52, 3 users, load
[root@r6181-d5-us01 ~]# df -kh
Filesystem      Size  Used Avail Use%
/dev/mapper/vg_ks-lv_root
tmpfs           50G   19G   28G   41%
/dev/sda1       127G  520K  127G    1%
/dev/dssd0000   477M   71M  381M   16%
tcp://192.168.150.1:3334/orangefs
/dev/gpfs-test  1.0T  480G  545G   47%
12T  5.7T  6.4T  47% /mnt/orangefs
/dev/gpfs-test  23T  1.1T  22T    5% /mnt/gpfs
[root@r6181-d5-us01 ~]# ps aux | sort -nrk 3,3 | head -n 5
root      97653  11.2  6.6 42665344 17520636 ?    S<Ll  Jul13 166:23 /usr/lpp/mmfs/bin/mmfscd
root      69849   6.6  0.0      0      0 ?        S    Jul12 181:54 [vpthread-1-1]
root      69850   6.4  0.0      0      0 ?        S    Jul12 177:42 [vpthread-1-2]
root       3829   3.0  0.0      0      0 ?        S    Jun27 730:04 [rp_thread 7:0]
root       3826   3.0  0.0      0      0 ?        S    Jun27 728:08 [rp_thread 6:0]
[root@r6181-d5-us01 ~]# ls -l /usr/lpp/mmfs/bin/mmfscd
-r-x----- 1 root root 20667161 Jun  3  2015 /usr/lpp/mmfs/bin/mmfscd
[root@r6181-d5-us01 ~]#
```

Sono visualizzati nell'ordine:

1. l'ora corrente
2. da quanto tempo il sistema è in funzione, espresso in giorni, ore e minuti
3. quanti sono gli utenti collegati al sistema
4. il carico medio del sistema nell'ultimo minuto
5. il carico medio negli ultimi cinque minuti
6. il carico medio negli ultimi quindici minuti

Bourne Again Shell (bash)





L'interprete dei comandi LINUX – 1

❖ Esempio

- A fronte del comando

```
$ rm file.txt
```

l'interprete cerca un file eseguibile chiamato **rm**, generalmente seguendo un percorso standard nel file system (**usr/bin**), lo carica in memoria e lo esegue con il parametro **file.txt**

- Esegue la cancellazione – *remove* – del file **file.txt**
- In alternativa...

```
$ rm -i file.txt
```

esegue la cancellazione solo dopo avere chiesto conferma all'utente





L'interprete dei comandi LINUX – 2

❖ **Esempio:** Un semplice script, salvato nel file **primo.sh**

```
cd change directory, mi riporta Home directory  
mkdir d1 make directory  
change mod chmod 444 d1 100 100 100 in HG x  
r r r  
cd d1 leggibile
```

- Da linea di comando, si lancia

```
$ primo > lire 100 100 100
```

ottenendo...

- ✗ **cd** cambia la directory corrente nella home dell'utente
- ✗ **mkdir d1** crea la directory **d1** nella directory corrente
- ✗ **chmod 444** imposta i permessi di **d1** a **r--r--r--**
- ✗ **cd d1** fallisce, provocando la visualizzazione del messaggio di errore

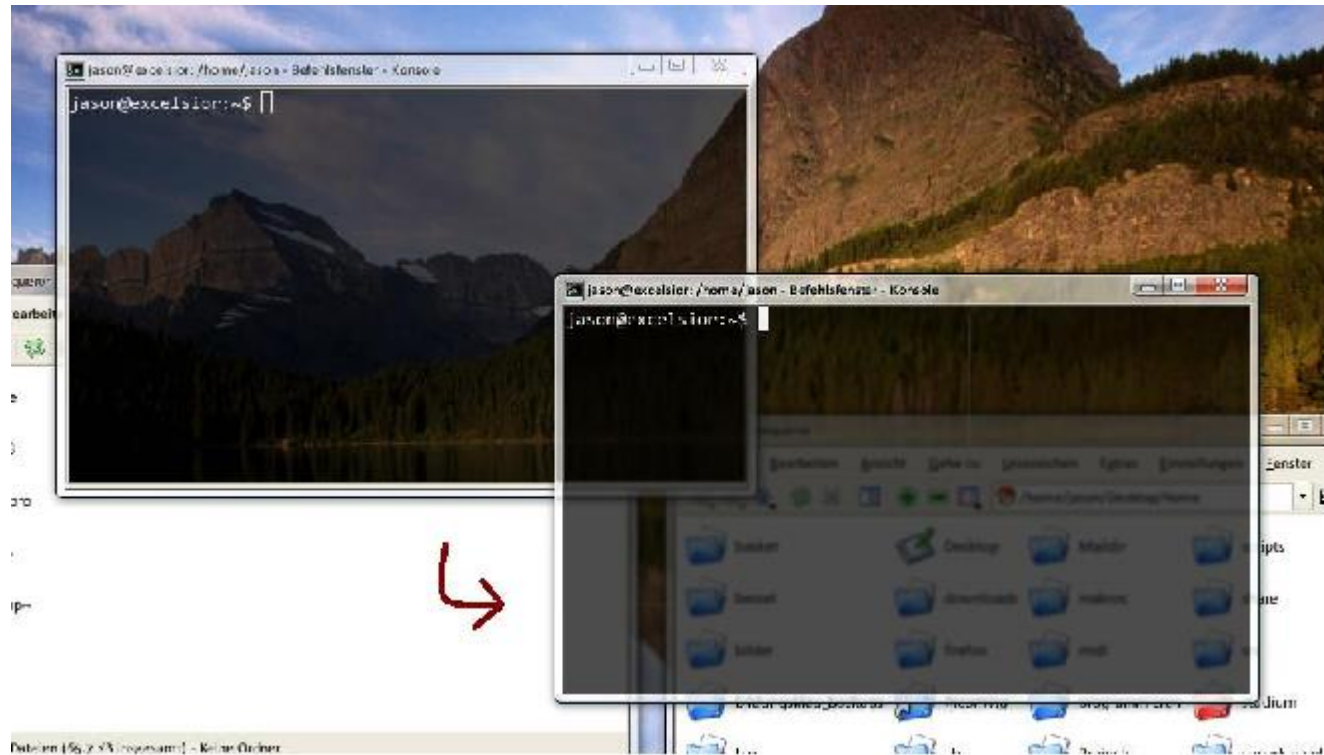
d1: Permission denied

*5 4 4
per errore*





L'interprete dei comandi LINUX – 3



Bash shell è una shell testuale del progetto GNU, ma disponibile anche per alcuni sistemi Microsoft Windows





Interfaccia utente GUI – 1

- ❖ Interfaccia user–friendly che realizza la metafora della scrivania (**desktop**)
 - Interazione semplice tramite mouse, tastiera, monitor
 - Le **icone** rappresentano file, directory, programmi, etc.
 - I diversi tasti del mouse, posizionato su oggetti differenti, provocano diversi tipi di azione (forniscono informazioni sull'oggetto in questione, eseguono funzioni tipiche dell'oggetto, aprono directory – *folder*, o *cartelle*, nel gergo GUI)
 - Realizzate per la prima volta, all'inizio degli anni '70, dai laboratori di ricerca Xerox PARC di Palo Alto (computer Xerox Alto, 1973)





Interfaccia utente GUI – 2

Apple, Xerox e l'interfaccia grafica

Molti testi sulla nascita dell'interfaccia grafica riportano la storia di una Apple che approfittò del lavoro del centro di ricerca Xerox Palo Alto Research Center (**PARC**) e lo commercializzò con il Lisa e con il Macintosh, derubando Xerox ed i suoi ricercatori di gloria e profitti.

Si tratta di un falso mito: ad indagare meglio, ad esempio leggendo testi come "Dealers of Lightning", si scopre che Apple pagò per la possibilità di visitare i laboratori del PARC. In cambio della visione dei prototipi di GUI di Xerox e dei colloqui con ingegneri e sviluppatori, la Xerox ricevette in cambio l'opzione su un pacchetto di azioni di Apple, che all'epoca era in procinto quotarsi in borsa.

Addirittura i ricercatori di Xerox sulle prime si opposero a mostrare a Steve Jobs e ai suoi le idee sviluppate nel centro (nell'ordine: interfaccia grafica, stampante laser e ethernet) e lo fecero solo in seguito a pressione dei dirigenti.



Un dettaglio importante è che **quella al PARC fu solo una dimostrazione e che non ci fu alcun passaggio di materiale o codice**: il risultato fu indubbiamente di ispirare gli sviluppatori del Lisa e del Mac ma gli uomini di Apple reimplementarono, ripensarono e completarono le idee ancora acerbe di Xerox, in modo diverso ed originale.

Un caso esemplare fu quello di Bill Atkinson che, all'oscuro di come funzionasse il sistema di gestione della grafica su schermo del PARC e cercando di eguagliarla, ne realizzò una versione superiore chiamata prima LisaGraf e poi QuickDraw.

La perspicacia di Apple nel mettere in pratica le idee in nuce del PARC fu infine capita anche dagli stessi ricercatori del PARC: alcuni di loro, come Larry Tesler e Alan Kay, passarono a Cupertino perché frustrati dalla miopia di Xerox e lavorarono al Lisa ed a vari progetti di Apple nel corso degli anni '80.

Triste epilogo dei rapporti tra le due aziende è il ripensamento che ebbe Xerox nel dicembre 1989 quando provò a fare causa ad Apple sul copyright dell'interfaccia del Lisa e del Macintosh: l'anno dopo una corte statunitense pose fine ai procedimenti scagionando Apple.



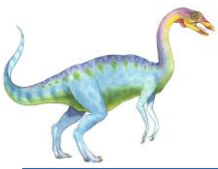


Interfaccia utente GUI – 3

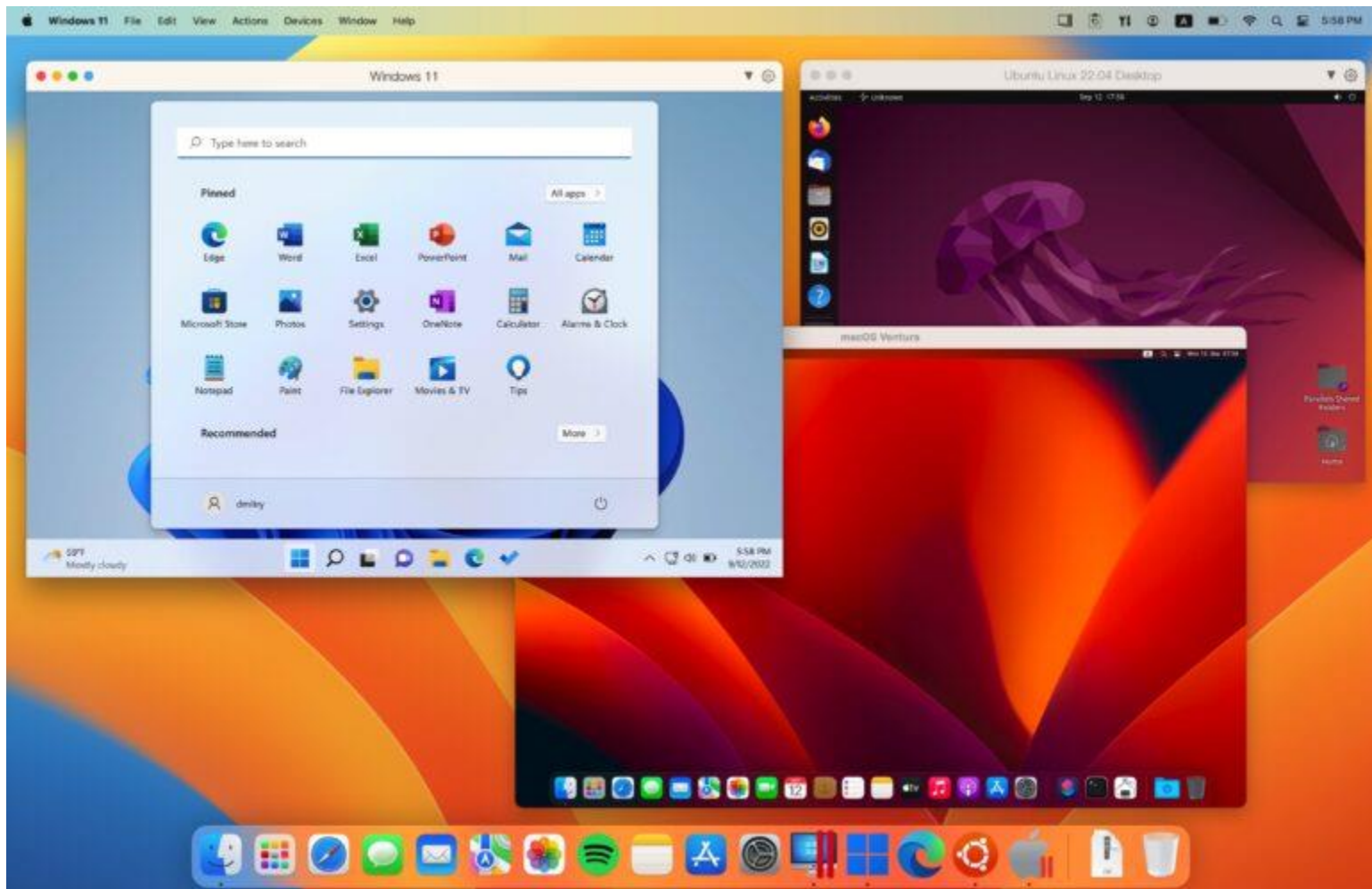
The collage displays four different Linux desktop environments:

- Top Left (Typhoon):** A desktop with a blue background featuring a cat. A weather widget shows 12°C and 37% humidity. The system clock shows Friday, January 23, 2015, at 23:36. The taskbar on the left contains various application icons.
- Top Right (Dolphin):** A file manager window titled 'Documents - Dolphin' showing a directory structure with folders like 'Atra Dot', 'mockups', 'plasma-4.3', 'Wade', 'Wallpapers', 'akademy-2008-group-photo.jpg', and 'Dawn_of_a_New_Desktop.pdf'. It includes a sidebar with 'Places' and 'Information' panels.
- Bottom Left (KDE Plasma 4):** A desktop with a dark background and a cat. A system monitor shows CPU temperature at 53°C. The system clock shows Friday, January 23, 2015, at 23:36. The taskbar on the left contains various application icons.
- Bottom Right (KDE Plasma 5):** A desktop with a dark background and a cat. A terminal window shows system statistics. The system clock shows Friday, August 21, 2015, at 11:00. The taskbar on the left contains various application icons.

Il desktop di GNU/Linux



Interfaccia utente GUI – 4



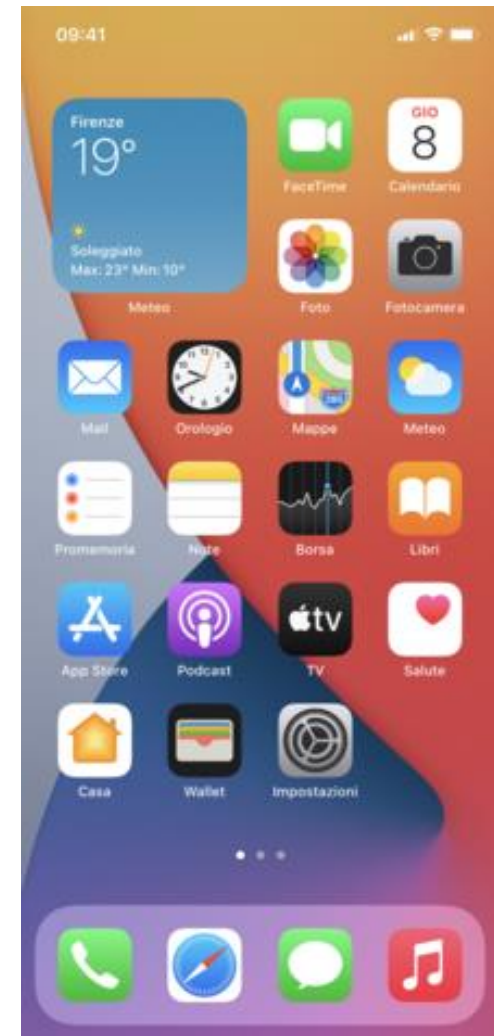
GUI di Mac OS (Ventura)





Interfaccia utente GUI – 5

- ❖ I device mobili con touch-screen richiedono nuovi tipi di interfaccia
 - Accesso senza il supporto del mouse (impossibile da usare o poco pratico)
 - Azioni ed operazioni di selezione realizzate tramite “gesti” (pressioni e strisciamenti delle dita)
 - Tastiera virtuale per l’immissione di testo
 - Comandi vocali



Springboard per iOS





Interfacce utente

- ❖ Molti sistemi operativi attuali includono interfacce sia CLI che GUI
 - **Microsoft Windows** principalmente basato su interfaccia grafica, ma dotato anche di una shell di comandi DOS-based (in Windows, tasto destro sul pulsante home → Windows PowerShell) *shell DOS, windows powershell*
 - **Apple Mac OS** interagisce per mezzo della GUI "Aqua" (non una specifica interfaccia, ma una serie di regole legate alla sua realizzazione); inoltre, è dotato di un kernel UNIX e mette a disposizione diversi tipi di shell
 - **Solaris** è tipicamente CLI, con interfaccia GUI opzionale (Java Desktop, KDE)
 - **Linux** è modulare; si può scegliere tra GUI molto avanzate (KDE, GNOME, CDE, Unity,...) e la CLI





Coza c'è sotto. syscall, chiamate di sistema.

Chiamate di sistema

è una oper. continua, riguarda dove essere nello spazio

- ❖ Le chiamate al sistema forniscono l'interfaccia fra i processi e i servizi offerti dal SO scrivono in C, C++ le chiamate di sistema
- ❖ Sono realizzate utilizzando linguaggi di alto livello (C o C++) e solo raramente l'assembly per piccole procedure che richiedono poca velocità
- ❖ Normalmente, vengono richiamate dagli applicativi attraverso **API** (Application Programming Interface), piuttosto che per invocazione diretta risultato di libreria per es. printf
- ❖ Alcune API molto diffuse sono la Win64 API per Windows, la POSIX API per i sistemi POSIX-based (tutte le versioni di UNIX, Linux, e Mac OS), e la Java API per la Java Virtual Machine (JVM) normalmente viene una linea di sistema, facile nome ma vuole una F. di LIBRERIA che impegna la chiamata al SYS os. printf
- ❖ **POSIX: P**ortable **O**perating **S**ystem **I**nterface per UNIX

Non so
come ma
il servizio
forza dello
syscall



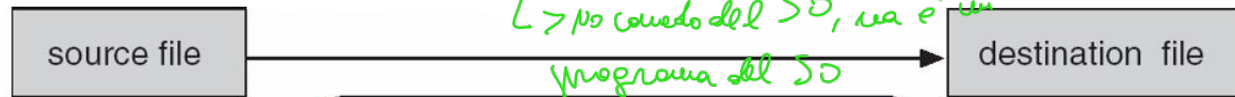


Esempio di chiamate di sistema

Pseudocodice, voglio copiare un file in un altro

- ❖ Sequenza di chiamate di sistema per realizzare la copia di un file in un altro

*CP file1.txt file2.txt
-> no comando del SO, ma è un programma del SO*



Possibili errori: il file non esiste o non può essere letto

Da gestire con ulteriori chiamate di sistema

error.

Possibili errori: il file esiste già

Example System Call Sequence

Acquire input file name *es. Messaggio a VIDEO: inserire nome file*
Write prompt to screen *syscall*
Accept input *syscall, da Tastiera*
Acquire output file name
Write prompt to screen
Accept input
Open the input file
if file doesn't exist, abort *errore, il file non esiste o usare*
Create output file
if file exists, abort *-> chiesto se voglio sovrascrivere o appendere*
Loop
Read from input file
Write to output file
Until read fails
Close the files *-> chiesto entrambi i files*
Write completion message to screen
Terminate normally

interfaccia segue con I/O

⇒ Migliaia di chiamate di sistema al secondo!

anche in semplice programma





Libreria standard del C come API

EXAMPLE OF STANDARD API

As an example of a standard API, consider the `read()` function that is available in UNIX and Linux systems. The API for this function is obtained from the `man` page by invoking the command

`man read`

man = manual

on the command line. A description of this API appears below:

```
#include <unistd.h>
ssize_t read(int fd, void *buf, size_t count)
```

return value *function name* *parameters*

man page *man page* *man page*

man page *man page* *man page*

→ *not tip file C, tipo
argomenti, numero, e tipo dato
restituire*

A program that uses the `read()` function must include the `unistd.h` header file, as this file defines the `ssize_t` and `size_t` data types (among other things). The parameters passed to `read()` are as follows:

- `int fd`—the file descriptor to be read
- `void *buf`—a buffer where the data will be read into
- `size_t count`—the maximum number of bytes to be read into the buffer

The `<unistd.h>` file defines miscellaneous symbolic constants and types, and declares miscellaneous functions

*SSIZE_t solo abito
per int*

On a successful read, the number of bytes read is returned. A return value of 0 indicates end of file. If an error occurs, `read()` returns -1.

int == count





Chiamate di sistema (cont.)

- ❖ Normalmente, a ciascuna **system call** è associato un numero ^{che codifica il tipo di servizio} ^{è un INTERRUPT SOFTWARE, versione di interrupt.}
 - L'**interfaccia alle chiamate di sistema**, ovvero l'ambiente di esecuzione a run-time (RTE), mantiene una tabella indicizzata dal numero di system call, effettua la chiamata e ritorna al chiamante lo stato del sistema dopo l'esecuzione (ed eventuali valori restituiti)
- ❖ L'utente non deve conoscere i dettagli implementativi delle system call: deve conoscere la modalità di utilizzo dell'**API** (ed eventualmente il compito svolto dalle chiamate di sistema)
 - L'intermediazione della API garantisce la portabilità delle applicazioni
 - Molto spesso una system call viene chiamata tramite una funzione di libreria standard (ad esempio contenuta in **stdlibc**)

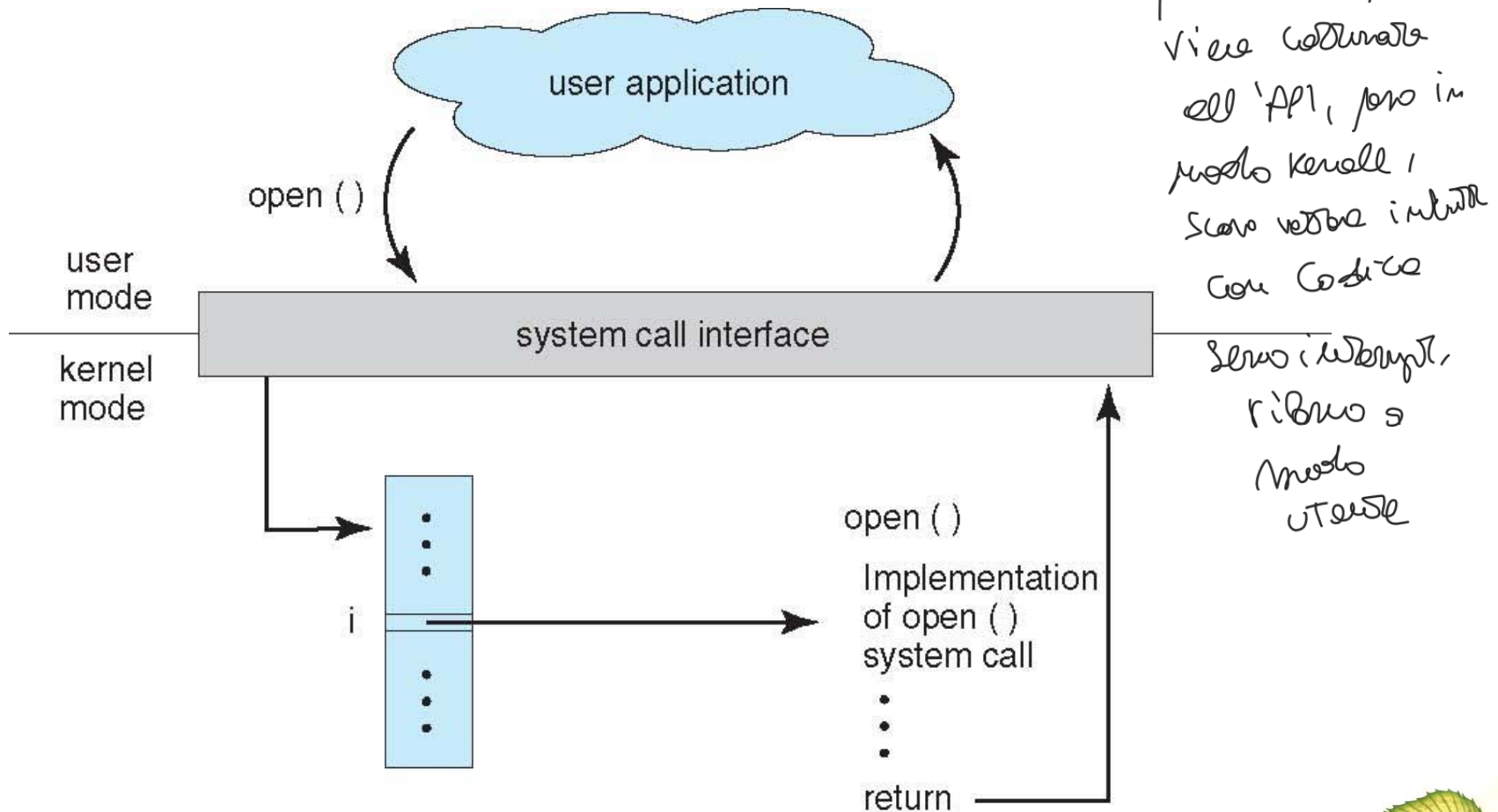
nelle
libreria
standard
di C





Relazioni API – System call – SO

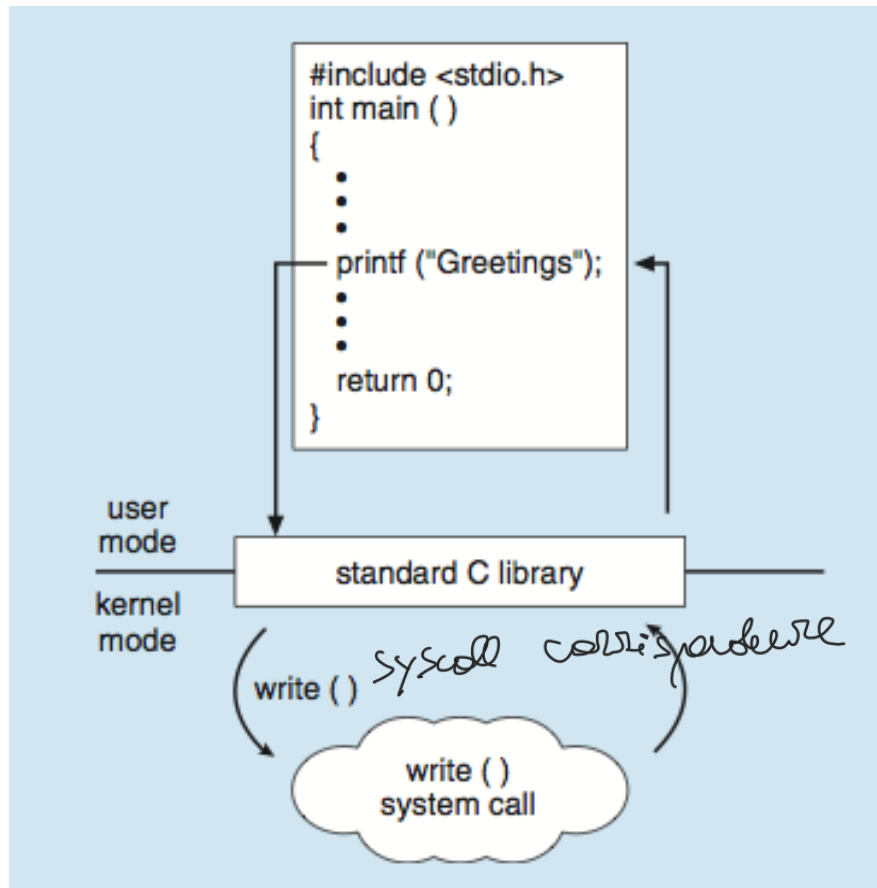
Gestione della chiamata di sistema *funzione del C per aprire file* **open ()** invocata da un'applicazione utente





Esempi di syscall con `stdlibc` – 1

- ❖ Per Linux, la libreria standard del linguaggio C (il *run-time support system*) fornisce una parte della API



- Programma C che invoca la funzione di libreria per la stampa `printf()`
- La libreria C intercetta la funzione e invoca la system call `write()`
- La libreria riceve il valore restituito dalla chiamata al sistema e lo passa al programma utente





Esempi di syscall con `stdlibc` – 2

❖ Funzione C che copia il contenuto di un file in un altro

```
#include <stdio.h>
#include <stddef.h>
#define FAIL 0 errore di programmazione, def. la costante
#define SUCCESS 1

int copy_file(infile, outfile) funzione di libreria
char *infile, *outfile; si definiscono 2 puntatori a file
{
    FILE *fp1, *fp2; struttura che contiene info del file, è un puntatore interno che punta alla memoria
    if ((fp1 = fopen(infile, "rb")) == NULL) read binary > se ritorno NULL allora c'è un problema con il file
        return FAIL;
    if ((fp2 = fopen(outfile, "wb")) == NULL)
    {
        fclose(fp1);
        return FAIL;
    }
    while (!feof(fp1)) finché file intero non è finito
    {
        macchina macchina non necessita molto syscall
        putc(getc(fp1), fp2);
        fclose(fp1);
        fclose(fp2); chiudo file
        return SUCCESS;
    }
} put e get, hanno anche l'incremento del puntatore del file
```

- Per eseguire l'I/O, è necessario associare un flusso ad un file o a una periferica
 - ⇒ occorre dichiarare un puntatore alla struttura **FILE**
- La struttura **FILE**, definita in **stdio.h**, è costituita da campi che contengono informazioni quali il nome del file, la modalità di accesso, il puntatore al prossimo carattere nel flusso
- Entrambi i file vengono acceduti in modalità binaria
- La macro **getc()** legge il prossimo carattere dal flusso specificato e sposta l'indicatore di posizione del file avanti di un elemento ad ogni chiamata



serve un minimo di cui può fare i vari dei file



Passaggio di parametri alle system call

- ❖ Spesso l'informazione necessaria alla chiamata di sistema non si limita al solo nome (o numero di identificazione)
 - Il tipo e la quantità di informazione varia per chiamate diverse e diversi sistemi operativi *So. come fa passaggio di parametri*
- ❖ Esistono tre metodi generali per passare parametri al SO
 - Il più semplice: passaggio di parametri nei registri
 - ▶ Talvolta, possono essere necessari più parametri dei registri presenti *non c'è un limite al # parametri da passare*
 - Memorizzazione dei parametri in un blocco in memoria e passaggio dell'indirizzo del blocco come parametro in un registro *mantenere nei registri indirizzo di memoria con i parametri*
approccio UNIX
 - ▶ Approccio seguito da Linux (per > 5 parametri) e Solaris
 - *Poco* *parametri usati* *lo stack* **Push** dei parametri nello stack da parte del programma; il SO recupera i parametri con un **pop**
 - Gli ultimi due metodi non pongono limiti al numero ed alla lunghezza dei parametri passati

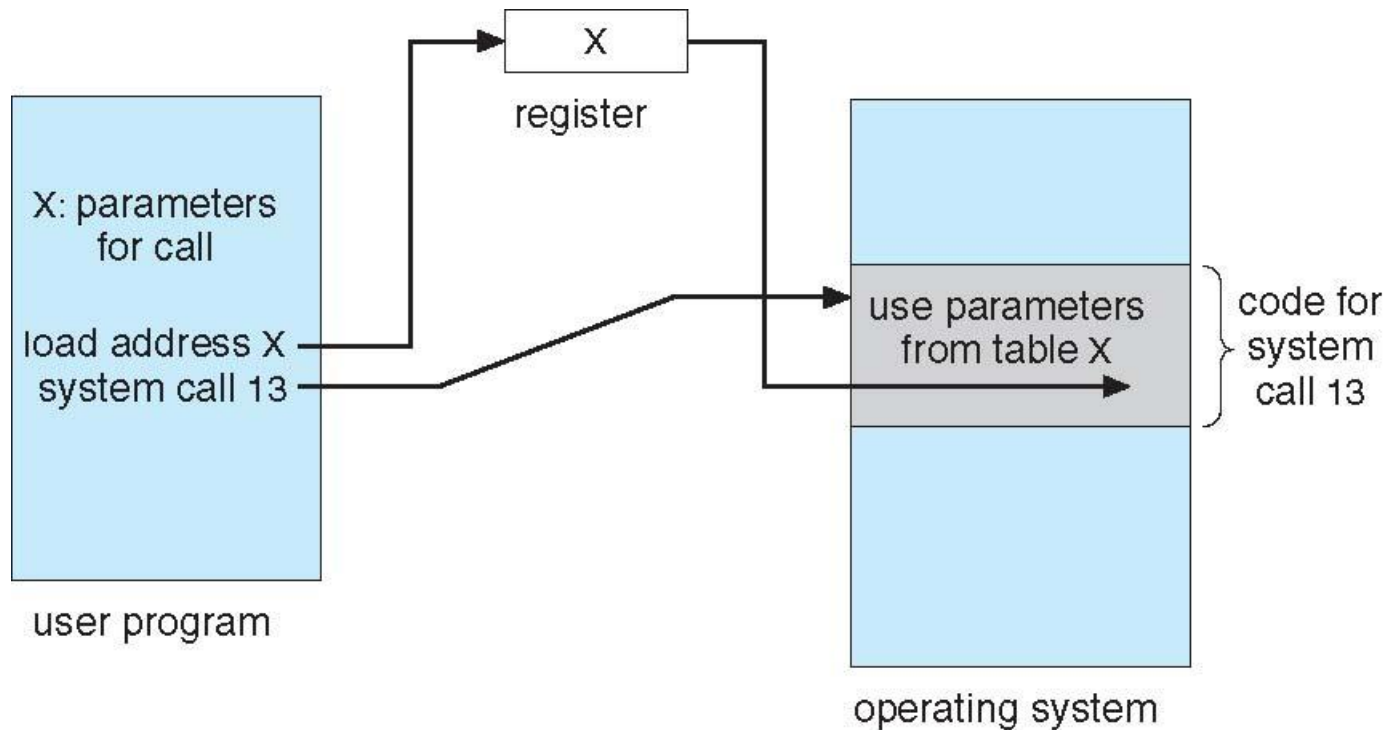
UNIX #parametri < 5, registri, > 5 poco indic. mem

UNIX No stack per progetto
param alle sys call
Silberschatz, Galvin and Gagne





Passaggio di parametri per indirizzo





Tipi di chiamate di sistema – 1

- ❖ Controllo dei processi
- ❖ Gestione dei file
- ❖ Gestione dei dispositivi di I/O
- ❖ Gestione delle informazioni
- ❖ Comunicazione *tra processi*
- ❖ Protezione *delle risorse*





Tipi di chiamate di sistema – 2

Per un process, no threads

Syscall per la gestione dei processi

❖ Controllo dei processi

- Creazione e terminazione di un processo (**fork**, **exit**)
- Caricamento ed esecuzione (**exec/execve**)
- Lettura/modifica degli attributi di un processo (priorità, tempo massimo di esecuzione – **get/set process attributes**)
- Attesa per il tempo indicato o fino alla segnalazione di un evento (**wait/waitpid**)
- Assegnazione e rilascio di memoria (**alloc, free**)
- Invio di segnali (**signal, kill**)
- Dump della mappa di memoria in caso di errore
- Debugger ed esecuzione a passo singolo
- Gestione di lock per l'accesso a memoria condivisa

Corico processo in memoria

es. scelta di priorità

Processo Termina in memoria, posso avere la mappa di memoria, lo di un debugger

Termino in modo normale

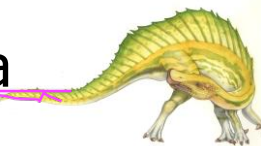
EXIT con codice ≠ 0 Termino - anomalo

come si usa o di un evento, aspetto la terminazione del processo figlio

es. kill -9 PID lo uccido

PS - a un mi ha veduto. Per i processi e i d

accesso a risorse comuni in modo regolamentato





Controllo dei processi in Arduino

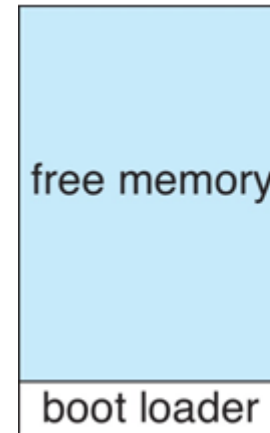
-> embedded,
no SO,

es. come si caricano programmi
in base alle diverse architetture

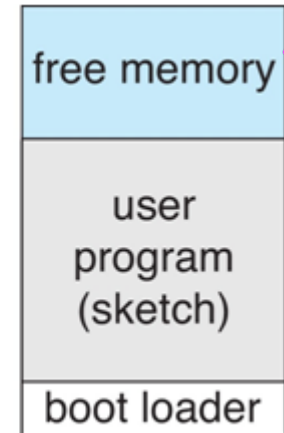
- ❖ Piattaforma hardware costituita da un microcontrollore e da sensori
- ❖ I programmi (*sketch*) devono essere preventivamente scritti e compilati su PC
- ❖ Gli sketch vengono caricati da una flash attraverso la porta USB
- ❖ È il *boot loader* che si occupa di caricare/scaricare i programmi
- ❖ Single-task
- ❖ Nessun SO (nessuna interfaccia utente)

Boot loader accetta ciò che viene caricato
via USB e lo esegue
UN solo processo alla volta

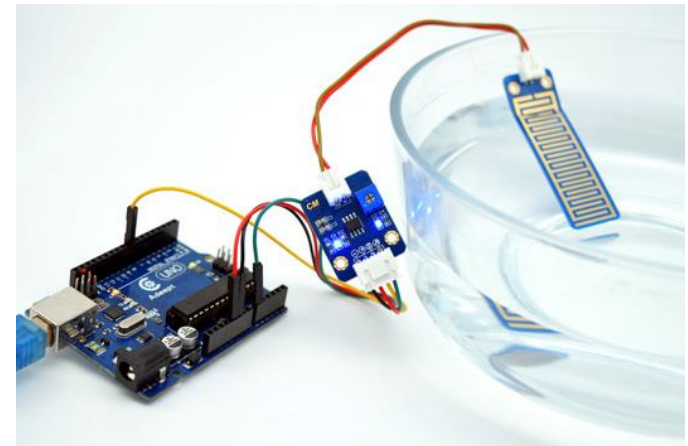
Allo startup del
sistema



Durante
l'esecuzione



diviso
lo spazio
memoria





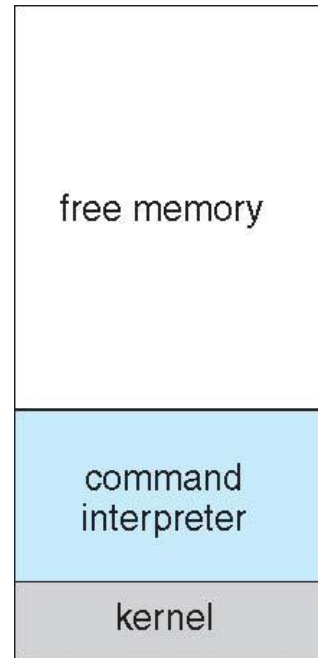
Controllo dei processi in MS-DOS

S.O. - MONOTASKING

Memoria So non c'è vero
un posto di tutti dell'
interprete

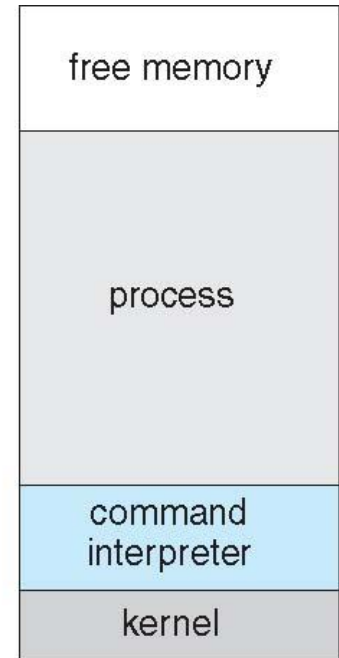
- ❖ Single-task
- ❖ La shell viene invocata al boot del sistema
- ❖ Metodo semplice di esecuzione dei processi
- ❖ Nessuna partizione della memoria
- ❖ Caricamento del processo in memoria a sovrascrivere lo stesso interprete (ma non il kernel)
- ❖ All'uscita dal programma → si ricarica la shell

SO non è interessato di gestire CPU e Memoria



(a)

Allo startup del sistema



(b)

Durante l'esecuzione di un programma utente

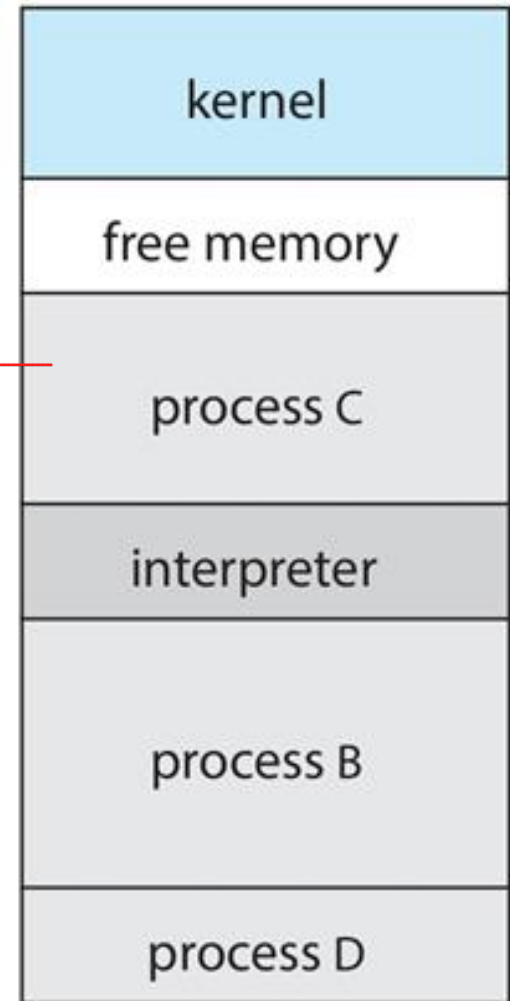




Controllo dei processi in FreeBSD

- ❖ È un sistema Unix open-source
- ❖ Multitasking *ho più processi in memoria
so fare in modo di gestire i processi*
- ❖ User login → si invoca la shell scelta dall'utente
- ❖ La shell esegue la system call **fork()** per creare nuovi processi
 - Si esegue la **exec()** per caricare il programma nel processo
 - La shell attende la terminazione del processo o continua ad eseguire i comandi utente
- ❖ I processi terminano con:
 - codice = 0 – esecuzione corretta
 - codice > 0 – tipo di errore

gestire risorse



Fork viene invocato in un processo che esegue lo stesso codice del padre. voglio che esegua codice differente,

EXEC Sostituisci il M.E.N. process con codice e idoli del mio programma





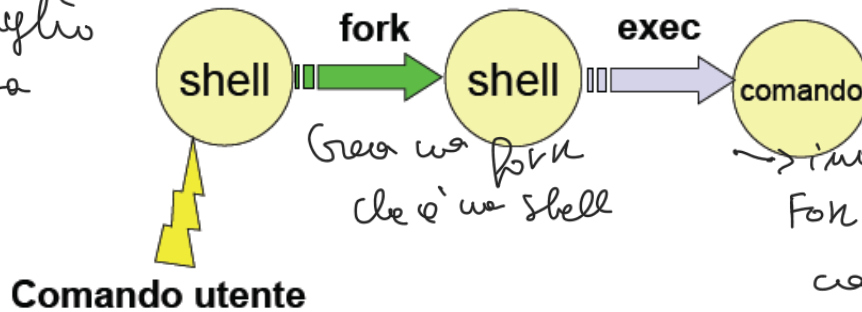
\$ primo

parte esecuzione

Esempio 1

- ❖ Per ogni comando, la shell genera un processo figlio, una nuova shell, dedicato all'esecuzione del comando:

come nasce il figlio
ma non viceversa
come può terminare
processo figlio



Gravida fork
che è una shell

→ primo
Fork. so scriviamo per shell
con codice. RETURN

- ❖ Possibilità di due diversi comportamenti

- Il padre si pone in attesa della terminazione del figlio (esecuzione in foreground)

Forka via al padre
non il figlio.

* Wilco N conobbe

? Wilco N conobbe
in solo
conobbe
→ \$ ls -l prog* > proginfo

Scrivo OUT in proginfo

- Il padre procede nell'esecuzione concorrentemente al figlio (esecuzione in background)

Shell in fork
Fino alla fine

→ \$ ls -l prog* > proginfo

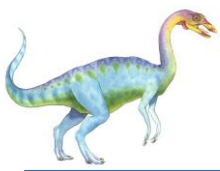
redies. output

&

→ lo manda
in background
lo subito Prompt shell

es. prog? *





Questo in background input e output deve essere fatto in **Esempio 2** background

❖ Cosa producono in stampa questi codici?

```
main ()
{
    val = 5;
    if(fork())
        wait(&val);
    val++;
    printf("%d\n", val);
    return val;
}
```

lo so process main figlio
Restituisce pid al padre e al figlio
Fork vero per il padre
Restituisce 0 al processo figlio, il pid del figlio al processo padre
per il figlio la fork non e' vera
val++, val = 6

• padre = val++, 7. ha stampato 6,7

Il processo figlio incrementa il valore di val e lo stampa, quindi lo restituisce al padre (che era in attesa)

Il padre incrementa ancora val e lo stampa

⇒ 6, 7

```
main ()
{
    val = 5;
    if(fork())
        wait(&val);
    else
        return val;
    val++;
    printf("%d\n", val);
    return val;
}
```

figlio ritorna val

Il processo figlio termina immediatamente restituendo il controllo al padre

Il padre incrementa val e lo stampa

⇒ 6



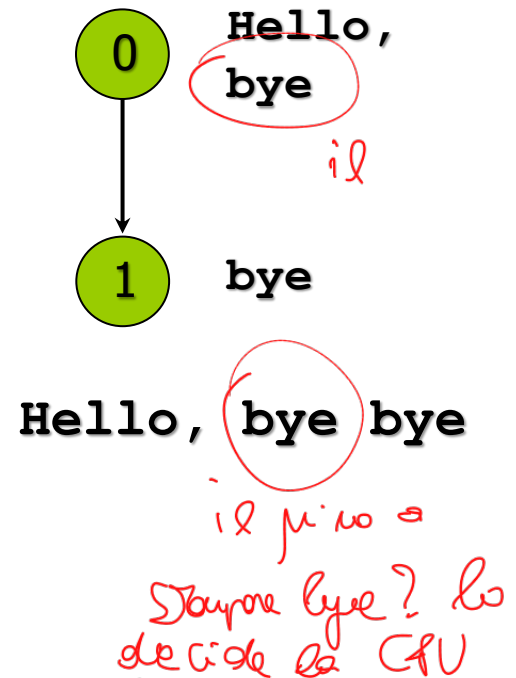


Esercizio 1

❖ Si consideri il codice seguente:

```
#include <stdio.h>
#include <sys/types.h>
#include <unistd.h>
int main(void)
{
    printf("Hello, ");
    fork();
    printf("bye ");
    return 0;
}
```

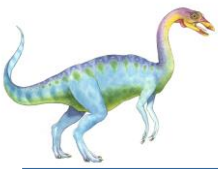
figlio →



Quale stampa produce la sua esecuzione? Fornire una chiara spiegazione alla risposta data.

*che non mi si
concorra*





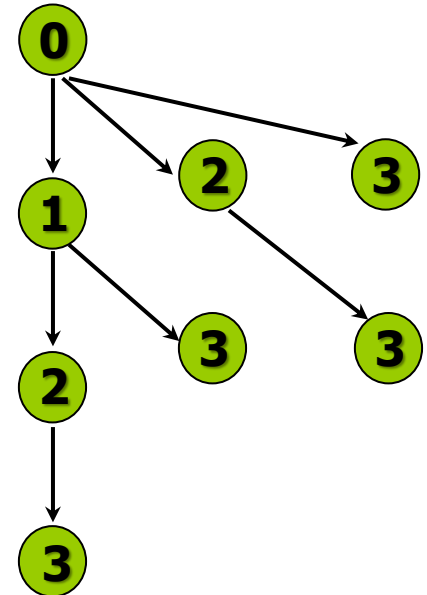
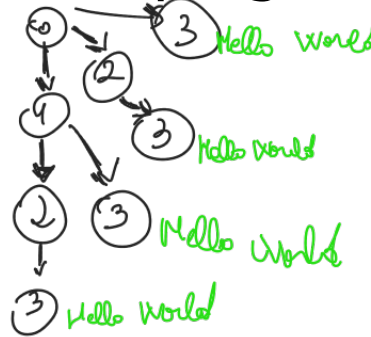
Esercizio 2

❖ Si consideri il seguente programma:

```
int main()
{
    pid_t pid;
    fork();
    fork();
    pid = fork();
    if (!pid)
        printf("Hello World\n");
}
```

0 ↓ 1 ↓ 2

tutti figli del vero padre, step



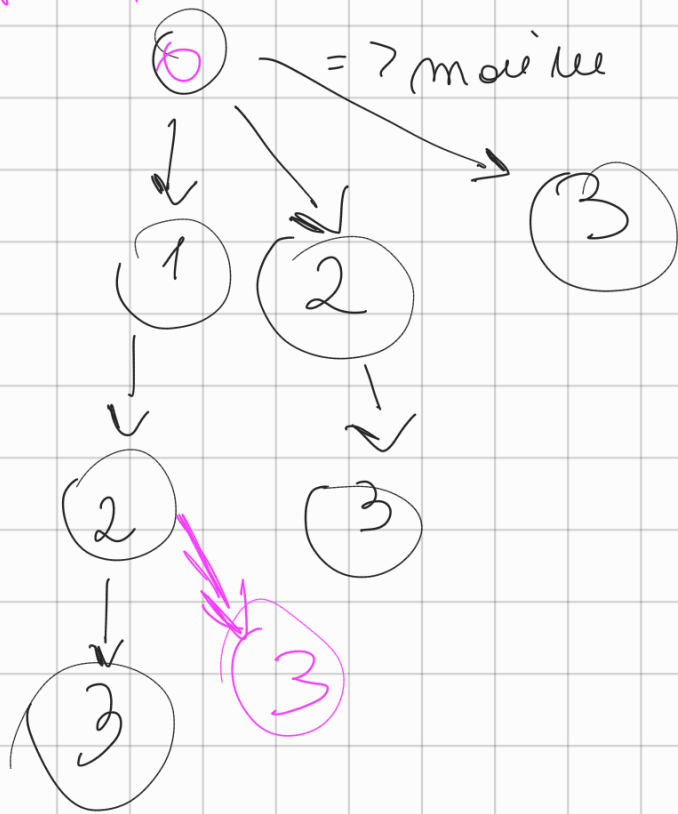
Hello World
Stampato da tutti i nodi
etichettati con 3

Si descriva l'albero dei processi creati e si dica quante volte viene stampata la stringa "Hello World".



Proceso 1, 2, 3

3rd fork



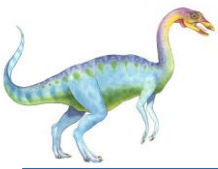
Fork residue list process

al helye B

Print f neue Stempel
solo 3 wolle
4

3. Für welche α ist die Reihe folge

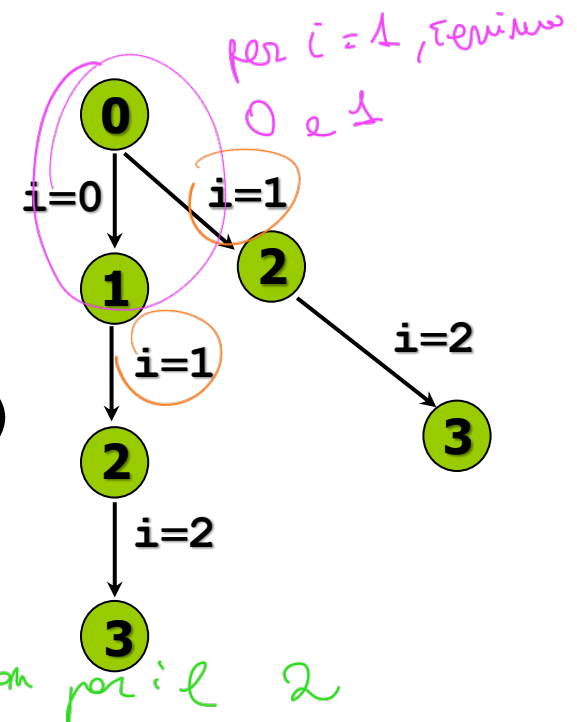
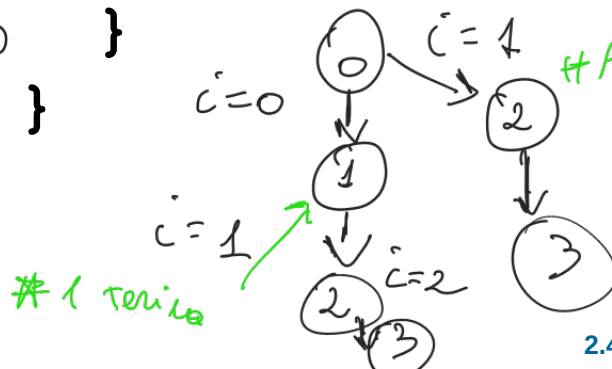




Esercizio 3

- ❖ Si stabilisca il numero di processi creati dal seguente codice, giustificando la risposta data con una descrizione dettagliata del codice stesso.

```
#include <unistd.h>
int main(void) {
    int i;
    for (i = 0; i < 3; i++) {
        if (fork() && (i == 1))
        {
            break;
        }
    }
}
```



le variabili i sono locali
e non si aggiornano

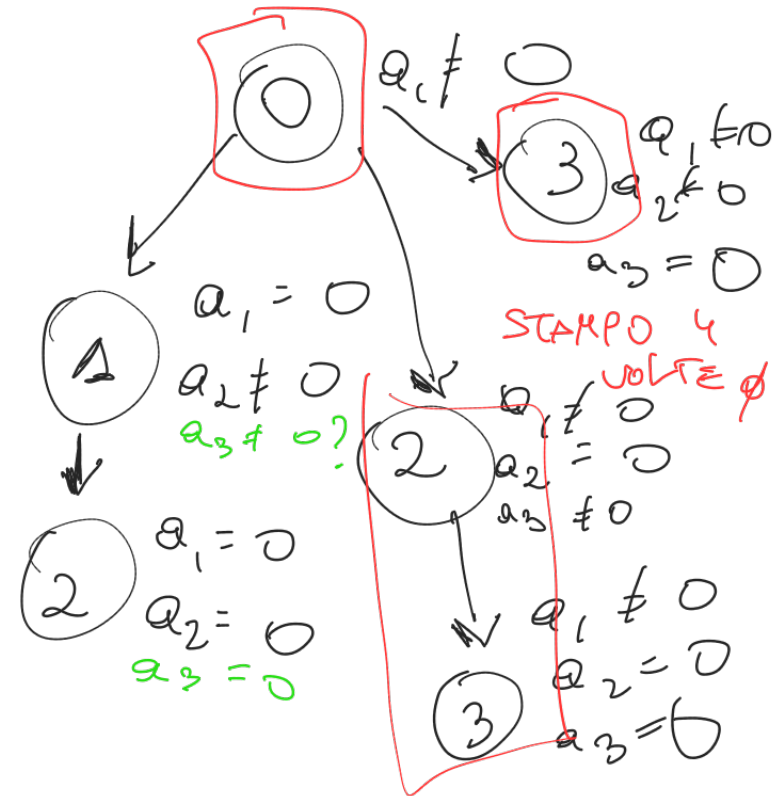




Esercizio 4

❖ Si consideri il seguente codice concorrente:

```
main() {
• int a1=0, a2=0, a3=0;
  a1=fork();
  a2=fork();
  if (a1 != 0) {
    a3=fork();
    printf("0");
  }
  return 0;
}
```



Si disegni il relativo albero dei processi motivandone la costruzione. Si descriva inoltre l'output prodotto dal programma.

STAMPA 4 volte
all'interno della printf





Esercizio 5

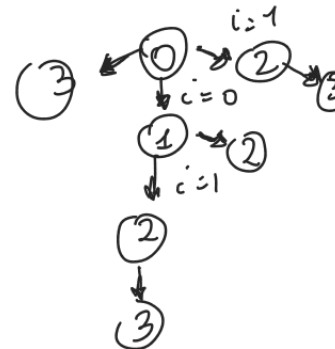
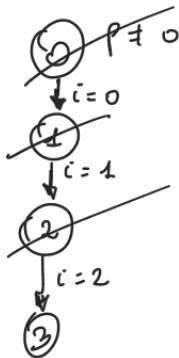
❖ Quanti processi creano i due codici seguenti?

```
#define N 3
int main()
{
    int i, p;
    for(i=0; i<N; i++) {
        if(p=fork()) break;
    }
}
```

```
#define N 3
int main()
{
    int i, p;
    for(i=0; i<N; i++) {
        p=fork();
    }
}
```

*3 Forks in combination
2³ processi*

Disegnare i relativi alberi dei processi.



define N 3

int main()

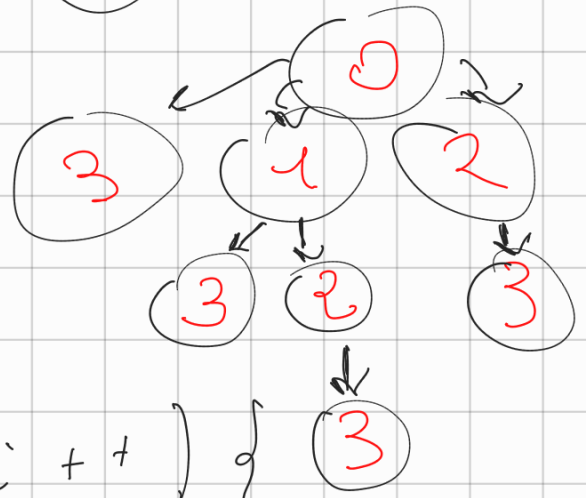
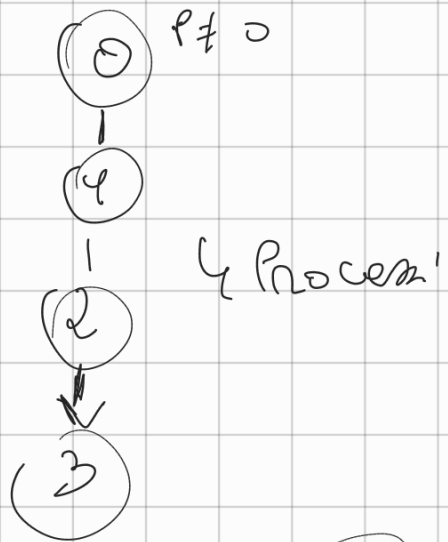
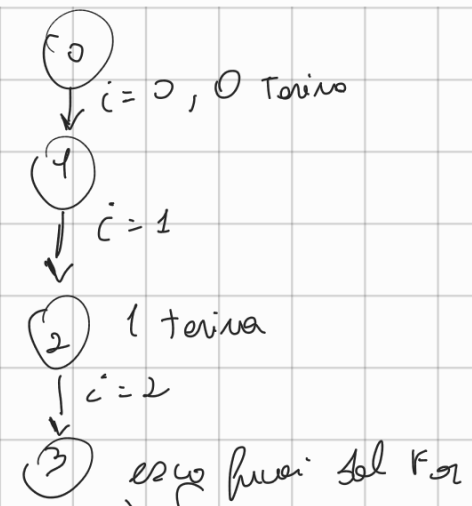
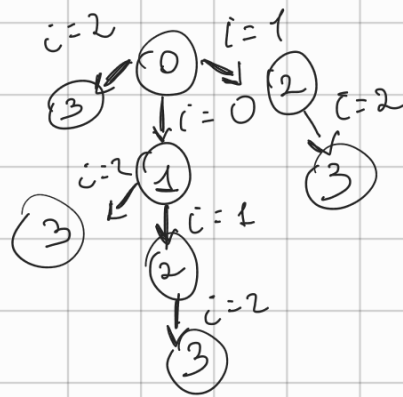
{ int i, p;

for (i=0; i < N; i++) {

if (p == fork()) break;

}

}



* Define N 3

int main ()

{

int i, p;

for (i=0; i < N; i++) {

p = fork();

}

}

For k in range(1, N+1): # process
p = 2^k, then zero more



Si vuole anche ai C/O

- Oves file
 se non esiste, = opendir()
 dipende da parametri che si

dispositivi di I/O → Charge File Mode 3 LTS

- Lettura, scrittura e posizionamento (read ^{i/b come file} ~~write~~ ^{modifiche degli attributi del device memorie})

mel. e
22-23





File e dispositivi

- ❖ Tutte le risorse controllate dal sistema operativo si possono concepire come dispositivi, alcuni dei quali sono fisici (come memorie non volatili e stampanti), mentre altri sono astratti o virtuali (i file, per esempio)
- ❖ La somiglianza fra file e dispositivi è infatti tale che molti SO, tra cui UNIX e Linux, ne forniscono una visione unificata
- ❖ In tal caso, l'insieme di system call è comune (eventualmente con differenze sui parametri)
- ❖ Linux rappresenta i dispositivi collegati al sistema come file nel file system e li colleziona nella directory /dev

per voir i dispositivi con help





Tipi di chiamate di sistema – 4

❖ Gestione delle informazioni

- Lettura/modifica dell'ora e della data (**time, date**) *comandi standard nel sistema operativo (molto sys call) fanno sys call per ottenere l'ora e la data del sistema*
- Informazioni sul sistema (**uname -r**) *uname: versione kernel*
- Lettura/modifica degli attributi di processi, file e dispositivi (**ps, getpid**) *ps: Processi in esecuzione, in memoria e in servizio. Ls: lista dei processi in esecuzione. Se da invoco dentro processo, mi ritorna il suo id*

❖ Comunicazione

- Apertura e chiusura di una connessione (open connection, close connection, **pipe**) *2 modelli: \$ ls | wc -l output di ls va in input a wc. non si apre a video, output nella pipe. PIPE: non si apre a video, output nella pipe. comandi: ls | wc -l*
- Invio e ricezione di messaggi (send, receive) *comandi: send, receive*
- Informazioni sullo stato dei trasferimenti (**traceroute**)
- Inserimento ed esclusione di dispositivi remoti
- Condivisione della memoria (**shm_open, shmget, mmap**) *comandi: shm_open, shmget, mmap*

❖ Protezione

- Controllo di accesso alle risorse
- Lettura/modifica dei permessi di accesso (**chown, chmod**)
- Accreditamento degli utenti *Regolazione accesso al file*





Esempi di chiamate di sistema in Windows e UNIX

Programmi non portabili, hanno nomi differenti

	Windows	Unix
Process Control	CreateProcess() ExitProcess() WaitForSingleObject()	fork() exit() wait()
File Manipulation	CreateFile() ReadFile() WriteFile() CloseHandle()	open() read() write() close()
Device Manipulation	SetConsoleMode() ReadConsole() WriteConsole()	ioctl() read() write()
Information Maintenance	GetCurrentProcessID() SetTimer() Sleep()	getpid() alarm() sleep()
Communication	CreatePipe() CreateFileMapping() MapViewOfFile()	pipe() shmget() mmap()
Protection	SetFileSecurity() InitializeSecurityDescriptor() SetSecurityDescriptorGroup()	chmod() umask() chown()





Programmi di sistema – 1

- ❖ I programmi di sistema forniscono un ambiente conveniente per lo sviluppo e l'esecuzione di programmi utente (semplici interfacce alle system call o programmi complessi)

Programmi che fanno parte del sistema operativo, in modo che l'utente

- ❖ Esistono programmi di sistema per...

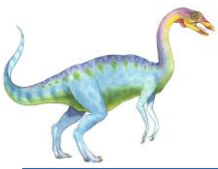
- Gestione di file (es.: **cp**) *copy*
- Informazioni di stato (es.: **du, who**) *→ chi è collegato ora al sistema*
- Editing di file (es.: **vi**) *disk usage, occup. dei dischi, disk free, spazio libero.*
- Supporto a linguaggi di programmazione (es.: **gcc**) *compilatore C*
- Caricamento e esecuzione di programmi (linker e loader)
- Comunicazioni (es.: **telnet, rlogin, ftp**) *collegarsi alle macchine virtuali.*
- Supporto alla realizzazione di applicativi

Programma in memoria, per cui può essere attivato dalla CPU. Ci sono uno scheduler

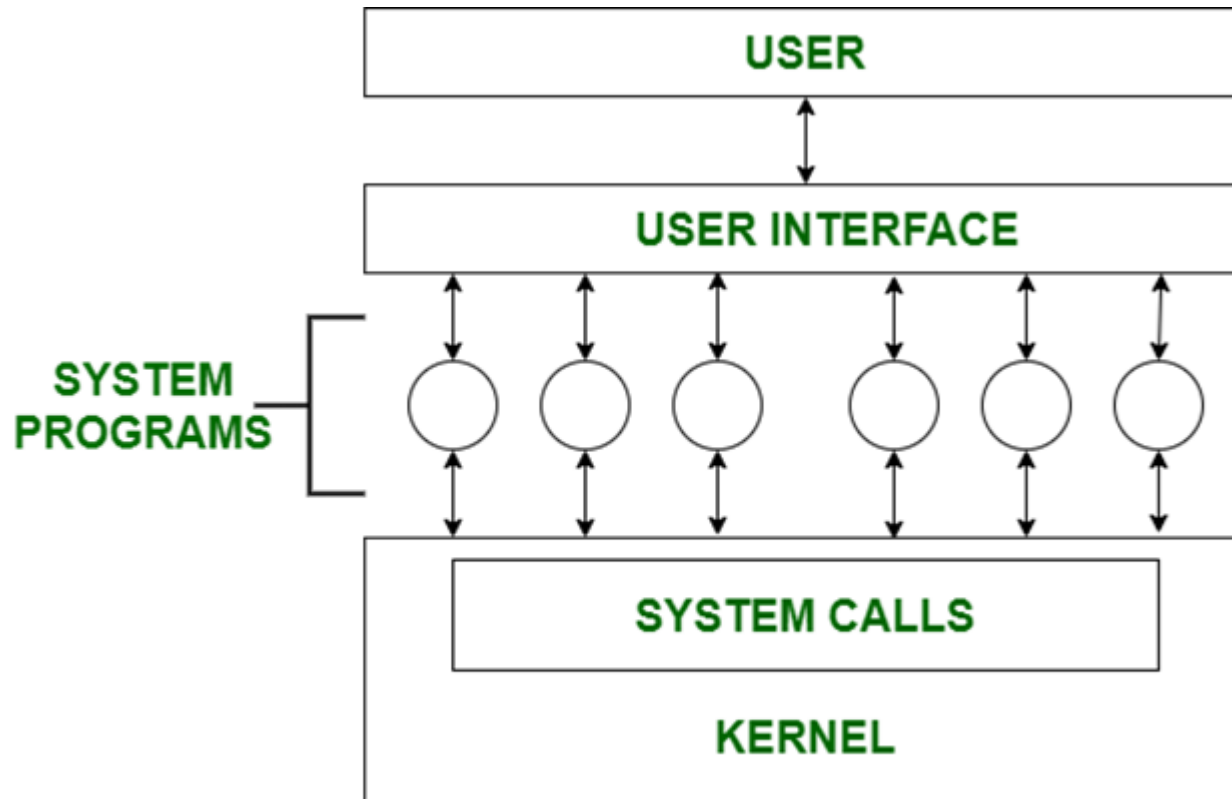
Un file è un eseguibile più grande. Un file è un programma in memoria. Lo fa diventare un programma.

- ❖ L'aspetto del SO per la maggioranza degli utenti è definito dai programmi di sistema, non dalle chiamate di sistema vere e proprie





Programmi di sistema – 2



interfaccia
tra le syscall





Programmi di sistema – 3

- ❖ **Gestione di file** – per creare, cancellare, copiare, rinominare, listare, stampare e, genericamente, gestire le operazioni su file e directory *sono programmi di sistema*
- ❖ **Informazioni di stato**
 - Per ottenere dal sistema informazioni tipo data, spazio di memoria disponibile, spazio disco, numero di utenti abilitati *info dettagliate sulla sua operatività*
 - Per ottenere informazioni sulle statistiche di utilizzo del sistema di calcolo (prestazioni, logging, etc.) e sul debugging
 - Per effettuare operazioni di formattazione e stampa dei dati
 - Per ottenere informazioni sulla configurazione del sistema (registry) *es. tipo SO, versione kernel, modelli applicativi e kernel ecc*





Programmi di sistema – 4

- ❖ **Modifica di file** *es. cercare un file e modificare di più il contenuto* *manipolazione di programmi di sistema, perché sono operazioni complesse*
 - Editori di testo, per creare e modificare file
 - Comandi speciali per cercare informazioni all'interno di file o effettuare trasformazioni sul testo
- ❖ **Supporto a linguaggi di programmazione** – assembler, compilatori e interpreti
- ❖ **Caricamento ed esecuzione di programmi** – linker, loader, debugger per linguaggio macchina e linguaggi di alto livello
- ❖ **Comunicazioni** – per creare connessioni virtuali tra processi, utenti e sistemi di elaborazione
 - Permettono agli utenti lo scambio di messaggi video e via e-mail, la navigazione in Internet, il login remoto ed il trasferimento di file





Programmi di sistema – 5

❖ Servizi background

servizi in background di cui utente ha poca percezione

- Lanciati durante la fase di boot
 - ▶ Taluni utili solo nella fase di startup del sistema
 - ▶ Altri in esecuzione dal boot allo shutdown *es. daemon in ascolto sullo porta 80*
- Supportano servizi quali controllo del disco, scheduling dei processi, logging degli errori, stampa, connessioni di rete
- Vengono eseguiti in modalità utente
- Noti anche come **sottosistemi**, **daemon**

❖ Programmi applicativi

*Programmi utente, codice scritto al di fuori del S.O., e
Eseguiti come tali*

- Non fanno parte del sistema operativo
- Eseguiti dagli utenti
- Lanciati da linea di comando, dal click del mouse o da pressione sul touchscreen





Linker e loader – 1

↳ *Great File oggetto rilocabile, non ci sono indirizzi veri, assoluti, ma all'inizio di quel programma* • *istruzioni relativi all'inizio del programma*

- ❖ Codice sorgente compilato in file oggetto, progettati per essere caricati in qualsiasi posizione di memoria fisica – **file oggetto rilocabili**
in essi ci sono solo indirizzi relativi *indirizzi fisici solo dopo avere caricato*
- ❖ Combinati dal **linker** in un singolo **file eseguibile binario** in cui vengono incluse anche le funzioni di libreria
- ❖ I programmi risiedono quindi nella memoria secondaria come eseguibili binari
- ❖ Devono essere caricati in ^{RAM} memoria centrale dal **loader** per essere eseguiti *lo controllo della consegna della CPU*
 - Durante la fase di rilocalizzazione si assegnano gli indirizzi assoluti alle parti del programma e ai dati *mentre prima c'erano solo indirizzi relativi*





Linker e loader – 2

I Sg di UNIX non hanno interesse al tipo di file. SO non di cura se è txt, ma l'utente lo sa che è un file leggibile.
MacOS, Tipo file stream leggibile a program.

- ❖ I moderni sistemi general purpose non collegano le librerie ai file eseguibili *se LINKER automaticamente le librerie, sono un oggetto e ci sarebbe linkato tutti gli eseguibili alla libreria*

unico eseguibile in UNIX per le librerie, tutti i programmi fanno riferimento allo stesso eseguibile della memoria non il codice modificabile il codice di libreria, non ho problemi di coerenza

■ Piuttosto, le **librerie collegate dinamicamente** (e.g., le **DLL** di Windows) vengono caricate secondo necessità e condivise da tutti i programmi che utilizzano la stessa versione della stessa libreria (presente in memoria in un'unica copia) *apparso a eseguibili di libreria solo in esecuzione. LIBRERIE CONDIVISE tutti i programmi che lo usano fanno ref. allo stesso area memoria*

- ❖ I file oggetto e eseguibili hanno formati standard, perché il sistema operativo sappia come caricarli e avviarli
- ❖ Nei sistemi UNIX-like, il formato standard è l'**ELF**, per **Executable and Linkable Format** *Formato di exec. è tipico del SO.*

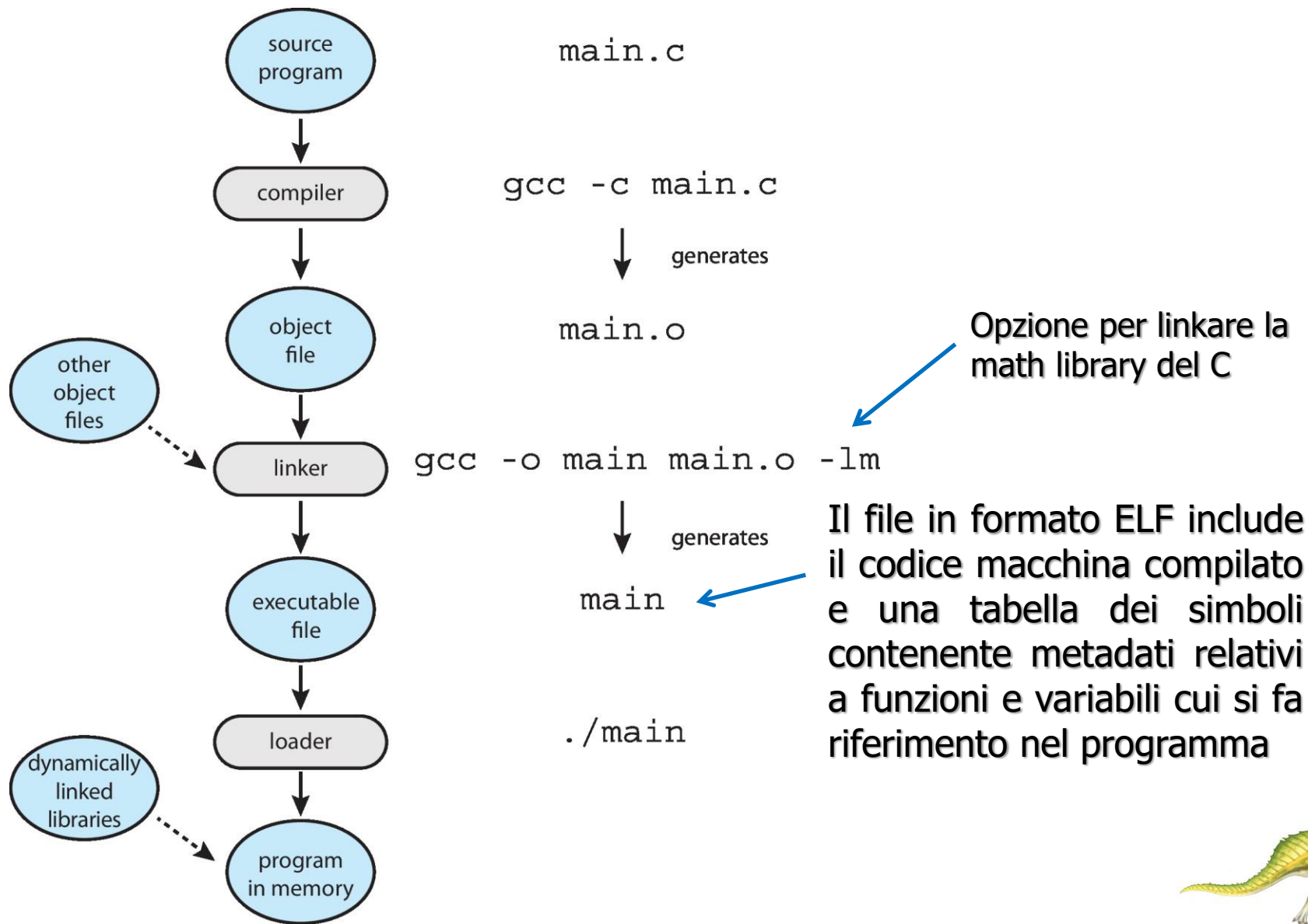
- Contiene il punto di inizio del programma, cioè l'indirizzo della prima istruzione da eseguire

- ❖ I sistemi Windows utilizzano il formato PE (Portable Executable), mentre MacOS usa file Mach-O





Linker e loader – 3





Perché le applicazioni sono SO-specific?

- ❖ Le applicazioni compilate su un sistema di solito non sono eseguibili su altri sistemi operativi *Perché i loro SO hanno il loro formato dell'eseguibile. Devono ricompilare e l'eseguibile*
- ❖ Ogni sistema operativo offre le proprie chiamate di sistema uniche, i propri formati di file eseguibili, etc.
- ❖ Le applicazioni possono essere multi-SO se:
 - scritte in un linguaggio interpretato come Python, con interprete disponibile su più sistemi operativi
 - scritte in un linguaggio (come Java) che include una VM contenente l'app in esecuzione
 - scritte in un linguaggio standard (come C), ma compilate separatamente su ciascun SO
- ❖ **Application Binary Interface (ABI)** è l'equivalente di API, ma definisce in che modo i diversi componenti del codice binario possono interfacciarsi per un dato SO su una data architettura, CPU, etc.





Riassumendo...

- ❖ I tipi di richieste di servizio al SO variano secondo il tipo di sistema ed il livello delle richieste stesse
- ❖ Il livello cui appartengono le chiamate di sistema deve offrire le funzioni di base (controllo di processi e memoria e gestione di file e dispositivi)
- ❖ Le richieste di livello superiore, soddisfatte dall'interprete dei comandi o dai programmi di sistema, vengono tradotte in una sequenza di chiamate al SO
- ❖ Oltre le categorie di richieste di servizio standard, gli errori nei programmi possono considerarsi richieste di servizio implicite





Progettazione e realizzazione di SO – 1

- ❖ Progettare il SO “perfetto” è un compito che non ammette soluzione, ma alcuni approcci implementativi si sono dimostrati comunque validi *software nello stesso e ambiente*
- ❖ La struttura interna dei diversi SO può variare notevolmente...
 - ...in dipendenza dall'hardware *sottostante*
 - ...e dalle scelte progettuali che, a loro volta, dipendono dallo scopo del sistema operativo e ne influenzano i servizi offerti *es. general purpose o embedded*
- ❖ Richieste utente ed obiettivi del SO
 - **Richieste utente** – il SO deve essere di semplice utilizzo, facile da imparare, affidabile, sicuro e veloce
 - **Obiettivi del sistema** – il SO deve essere semplice da progettare, facile da realizzare e mantenere, flessibile, affidabile, error-free ed efficiente

da li lo sviluppa ->





Progettazione e realizzazione di SO – 2

- ❖ Per la progettazione e la realizzazione di un sistema operativo è fondamentale mantenere separati i due concetti di...

di vido le politiche dei meccanismi

*Po' conline
nel tempo e lo
essere di norma*

❖ **Politiche:** Quali sono i compiti e i servizi che il SO dovrà svolgere/fornire? (Es.: scelta di un particolare algoritmo per lo scheduling della CPU)

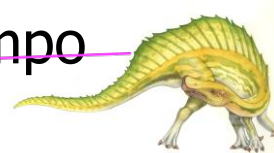
*UNIX, in runtime viene valutata la priorita
della c'è scheduler*

- **Meccanismi:** Come realizzarli? (Es.: timer)

- ❖ I meccanismi determinano “come fare qualcosa”, le politiche definiscono “cosa è necessario fare”

- ❖ **Esempio:** i problemi di assegnazione delle risorse impongono decisioni politiche, i meccanismi servono ad implementarne l'accesso controllato

- La separazione fra politiche e meccanismi è fondamentale; si garantisce la massima flessibilità se le decisioni politiche subiscono cambiamenti nel corso del tempo





Progettazione e realizzazione di SO – 3

❖ Esempi

- **Windows e MacOS:** sia le politiche sia i meccanismi sono fissati a priori e cablati nel sistema; tutte le applicazioni hanno interfacce simili perché l'interfaccia stessa fa parte del kernel e delle librerie di sistema
- **UNIX/Linux:** Netta separazione fra meccanismi e politiche

❖ Progettare e realizzare un SO è un compito di **ingegneria del software** eminentemente creativo





Sviluppo software

- ❖ Tradizionalmente i SO venivano scritti in linguaggio assembly; successivamente, furono utilizzati linguaggi specifici per la programmazione di sistema, quali Algol e PL/1; attualmente vengono invece sviluppati in linguaggi di alto livello, particolarmente orientati al sistema: C o C++
- ❖ In realtà, normalmente, si utilizza un mix di linguaggi:
 - Componenti di basso livello sviluppate in assembly (es.: driver dei dispositivi)
 - Kernel in C *per forza... con Pipeline, cache, bus, etc. e compilatore*
 - Programmi di sistema realizzati tramite, C, C++, e linguaggi di scripting, quali PERL, Python, shell script
- ❖ **Caratteristiche**
 - Veloci da codificare; codice compatto, di facile comprensione, messa a punto e manutenzione
 - Portabilità
 - Potenziale minor efficienza del codice C rispetto all'assembly





Struttura del sistema operativo

OS opera nel kernel
con syscalls

in un'area virtuale

Si ritiene che in alcune
architetture si possa avere

- non differenziato
- ❖ Sistemi storici (e non solo): **monolitici**
- Facilmente tutto il SO nel kernel
efficiente ma difficile da debuggare
- Le funzioni di gestione delle risorse sono tutte realizzate nel nucleo e l'intero sistema operativo tende a identificarsi col nucleo
 - Anche se ogni funzione è separata dal resto, l'integrazione del codice è molto stretta e, poiché tutti i moduli operano nello stesso spazio di indirizzi, un bug in uno di essi può bloccare l'intero sistema
 - Tuttavia, quando l'implementazione è sicura, la stretta integrazione interna dei componenti rende un buon kernel monolitico estremamente efficiente
 - "Filosofia" simile a quando non si strutturano i programmi in sottoprogrammi e si inserisce tutto il codice nella funzione **main()**





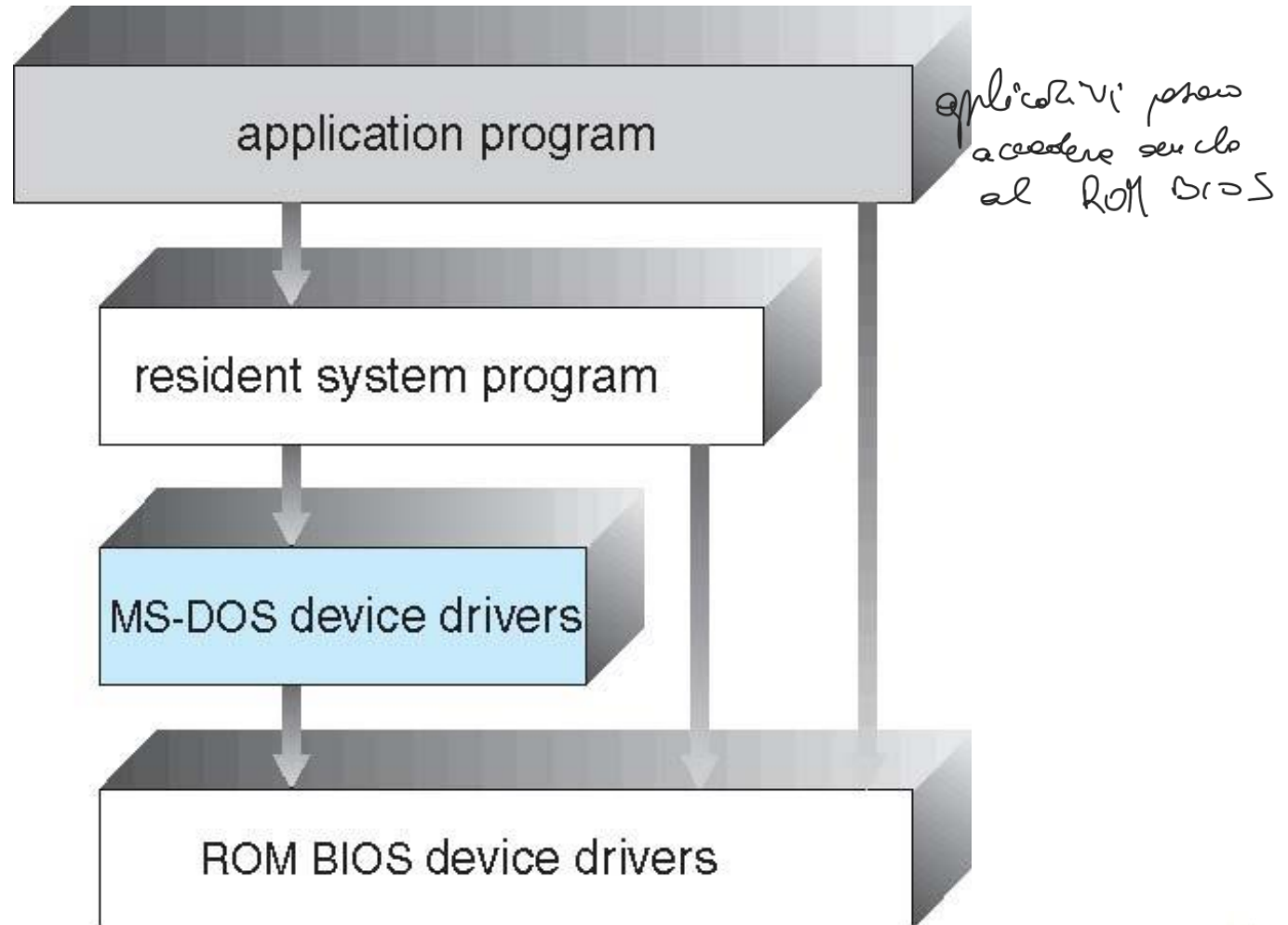
SO con struttura semplice

- ❖ **MS-DOS** — scritto per fornire il maggior numero di funzionalità utilizzando la minor quantità di spazio possibile *tutto concentrato nel kernel, non si hanno moduli distinti caricabili in runtime*
 - Non è suddiviso in moduli
 - Sebbene MS-DOS abbia una qualche struttura (a strati), le sue interfacce e livelli di funzionalità non sono ben separati
 - ▶ Le applicazioni accedono direttamente alle routine di sistema per l'I/O (ROM BIOS)
 - ▶ Vulnerabilità agli errori ed agli "attacchi" dei programmi utente
 - Intel 8088, per cui MS-DOS fu progettato, non offre duplice modo di funzionamento e protezione hardware
 - ⇒ impossibile proteggere hardware/SO dai programmi utente





Struttura a strati di MS-DOS





UNIX

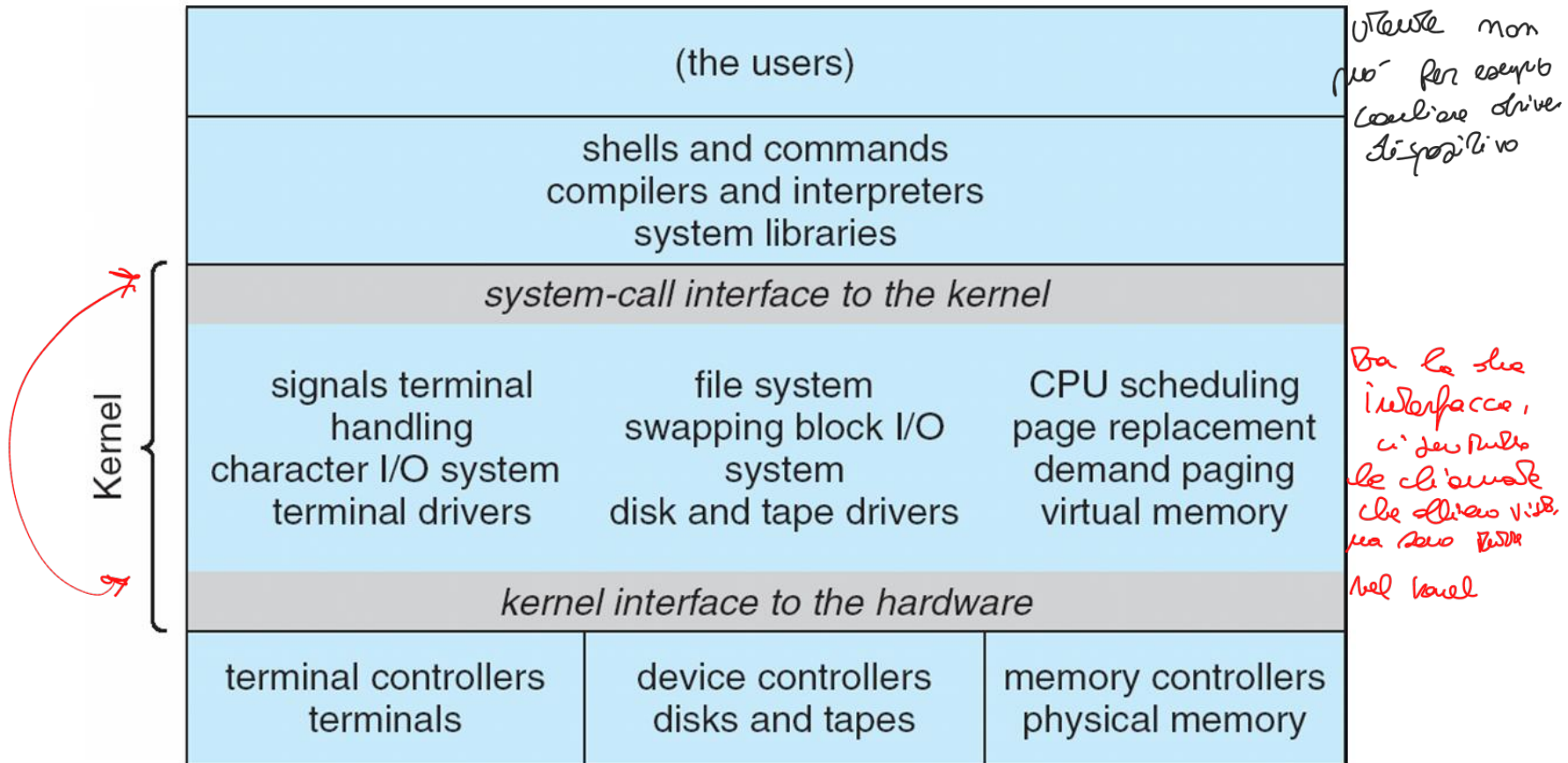
- ❖ **UNIX** — a causa delle limitate funzionalità hardware disponibili all'epoca della sua realizzazione, il sistema originale aveva una struttura scarsamente stratificata
- ❖ UNIX è costituito di due parti separate: *risposta per Mainframe*
 - I programmi di sistema
 - Il kernel
 - ▶ È formato da tutto ciò che si trova sotto l'interfaccia delle chiamate di sistema e sopra l'hardware
 - ▶ Fornisce il file system, lo scheduling della CPU, la gestione della memoria ⇒ un gran numero di funzioni per un solo livello!

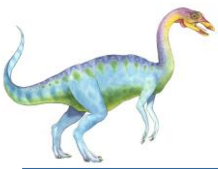




Struttura a strati di UNIX

Beyond simple but not fully layered





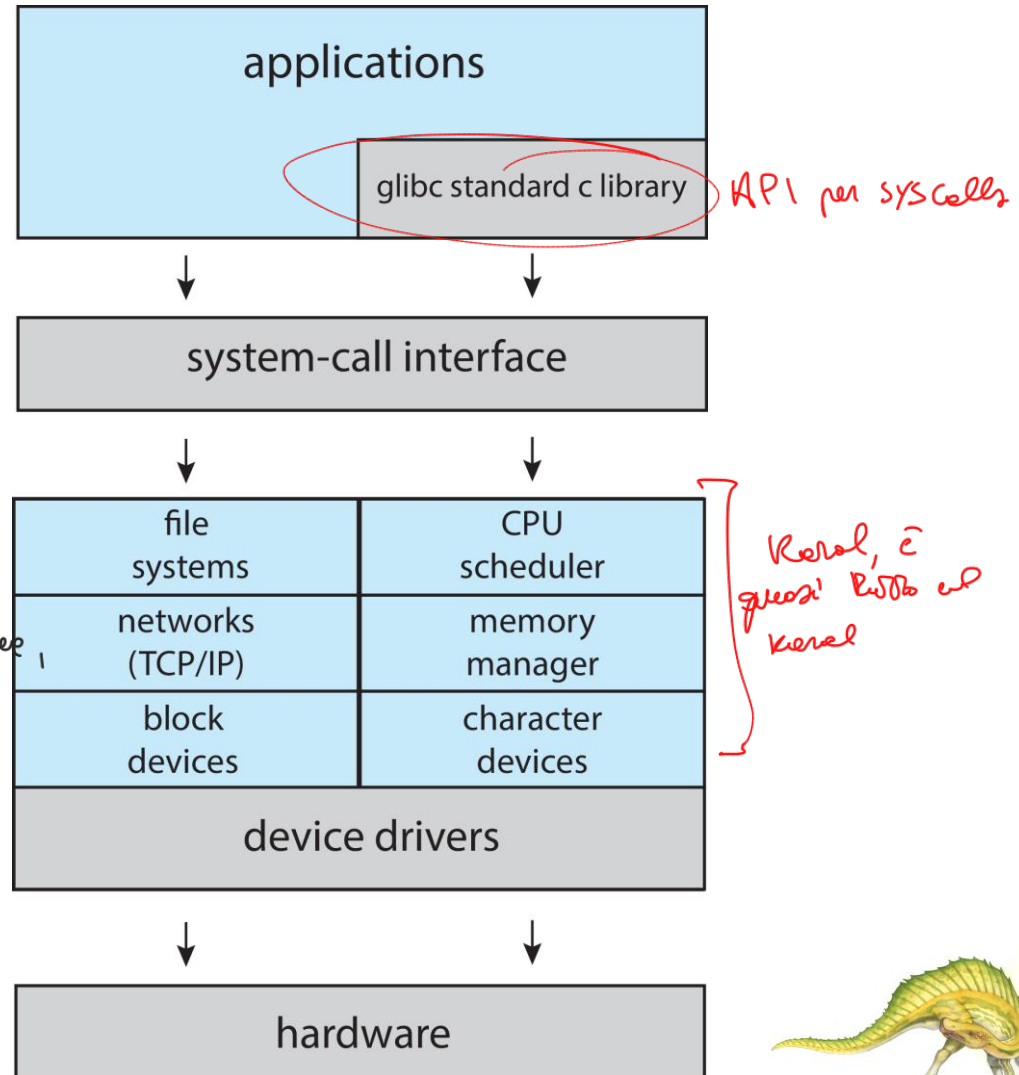
Struttura a strati di LINUX

glibc costituisce la API

Il kernel di Linux è monolitico, in quanto tutte le funzioni del SO vengono eseguite in modalità kernel in un unico spazio di indirizzi, ma è dotato anche di struttura modulare, dato che nuovi moduli possono essere caricati a runtime

*Pero' Moduli modulari
si possono aggiungere al SO
come moduli*

Monolithic plus modular design

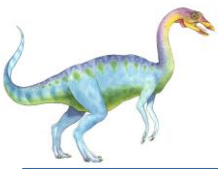




- ❖ In presenza di hardware appropriato, i SO possono assumere architettura stratificata, per meglio garantire il controllo sulle applicazioni

-
- layer N
user interface
- ⋮
- layer 1
- layer 0
hardware
- le interfacce di rete sono a chi sta sopra?
serve per la comunicazione
che serve per la comunicazione
- c'è proprio tutto ciò che serve per la comunicazione
- c'è un metodo per farlo
metodo per farlo

-



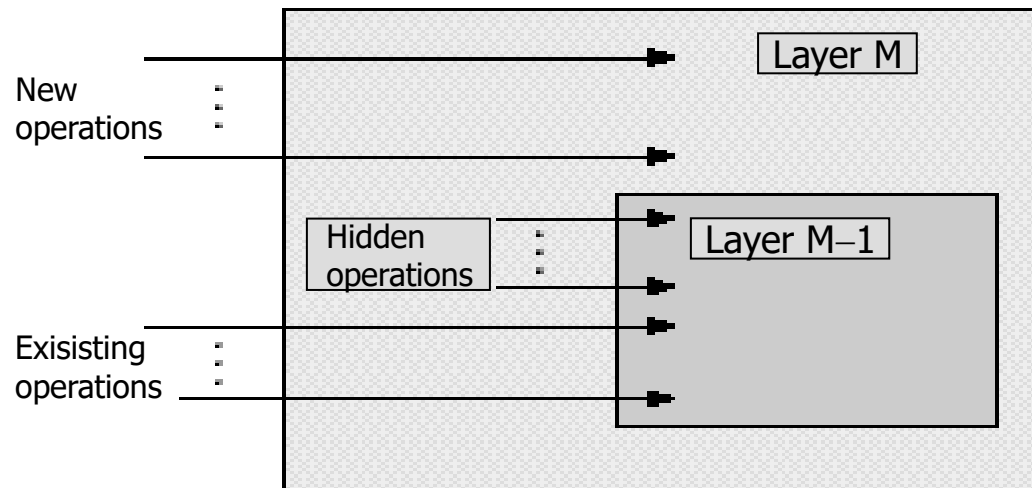
Approccio stratificato – 2

❖ Vantaggi

- Semplicità di realizzazione e messa a punto (che viene attuata strato per strato)

❖ Svantaggi

- Difficoltà nella definizione appropriata dei diversi strati, poiché ogni strato può sfruttare esclusivamente le funzionalità degli strati su cui poggia
- Tempi lunghi di attraversamento degli strati (passaggio di dati) per portare a termine l'esecuzione di una syscall





Approccio stratificato – 3

❖ Esempio 1 – Difficoltà di definizione degli strati

mettere il driver in ordine

- Il driver della ^{ho messo solo parte dei processi}memoria virtuale (backing store) dovrebbe trovarsi sopra lo scheduler della CPU, perché può accadere che il driver debba attendere un'istruzione di I/O e, in questo periodo, la CPU viene sottoposta a scheduling ^{scheduler parallel over l'io di usare memoria virtuale.}
- Lo scheduler della CPU deve mantenere più informazioni sui processi attivi di quante ne possono essere contenute in memoria: deve fare uso del driver della memoria ausiliaria

❖ Esempio 2 – Scarsa efficienza del SO

- Per eseguire un'operazione di I/O, un programma utente invoca una system call che è intercettata dallo strato di I/O...
- ...che esegue una chiamata allo strato di gestione della memoria...
- ...che richiama lo strato di scheduling della CPU...
- ...che la passa all'opportuno dispositivo di I/O

idea stratificazione, non più così presente.





Approccio stratificato – 4

- ❖ I sistemi stratificati sono stati utilizzati con successo nelle reti (TCP/IP) e nelle applicazioni Web
- ❖ Pochi SO utilizzano un approccio a strati puro
- ❖ Tuttavia, una parziale stratificazione della struttura è comune nei SO contemporanei (pochi strati multifunzionali, internamente modulari)
- ❖ Linux è “stratificato” poiché si effettuano diversi livelli (6–10) di chiamate di sistema prima di svolgere concretamente un compito





Struttura dei sistemi microkernel – 1

- ❖ Quasi tutte le funzionalità del kernel sono spostate nello spazio utente *Kernel minimale con servizi minimi*
- ❖ Un **microkernel** offre i servizi minimi di gestione dei processi, gestione della memoria e di comunicazione *fra processi*
 - Scopo principale: fornire funzioni di comunicazione fra programmi client e servizi (implementati esternamente)
 - Le comunicazioni hanno luogo tra moduli utente mediante scambio di messaggi (mediati dal kernel)
 - **Esempi:** prime versioni di Windows NT, Mach, GNU Hurd; Mac OS, con kernel Darwin, parzialmente basato su Mach *FS era implementata a livello user, cioè da processo user implementato dal S.O.*

Accediuto all'interno del kernel i servizi più ipotetici e correlati con l'HW del sistema e delegano tutte le cose non specifiche e sporadiche a livello utente





Struttura dei sistemi microkernel – 2

*kernel piccolo: componenti fondamentali,
estendendo il sistema non a liv. kernel, non lo
tocco, ma AGGIUNGO servizi, più semplice*

❖ Vantaggi

- Funzionalità del sistema più semplici da estendere: i nuovi servizi sono programmi di sistema che si eseguono nello spazio utente e non comportano modifiche al kernel
- Facilità di modifica del kernel
- Sistema più facile da portare su nuove architetture
- Più sicuro e affidabile (meno codice viene eseguito in modo kernel) *viene tutto svolto in modo utente*

❖ Svantaggi

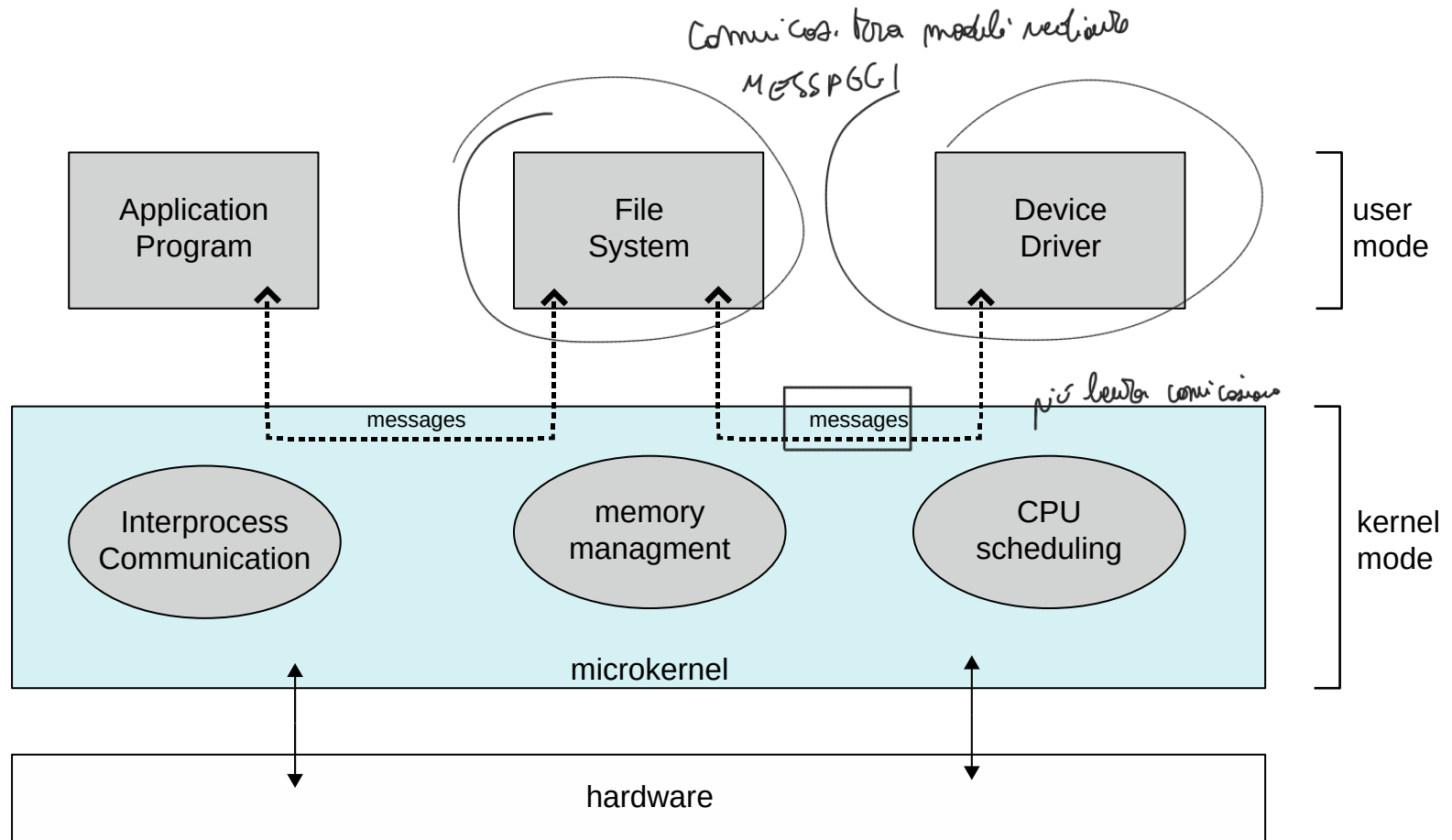
- Possibile decadimento delle prestazioni a causa dell'overhead di comunicazione fra spazio utente e spazio kernel *(meno efficienti, non come kernel monolitico)*

*tra gli i servizi viene IPC lo comunica molto rapidamente, tutto nello stesso spazio si muovono tutti
gestisce il kernel ... scambio di messaggi
che fa decadere prestazioni*





Struttura dei sistemi microkernel – 3





Kernel modulari

*kernel modulari, no
servizi opzionali a runtime o
quando il sys va ON*

❖ Molti degli attuali SO implementano **kernel modulari** (Loadable Kernel Modules, LKMs)

- Si utilizza un approccio alla programmazione object-oriented
- Ciascun modulo implementa una componente base del kernel, con interfacce e funzioni definite con precisione
- Ciascun modulo colloquia con gli altri mediante l'interfaccia comune
- Ciascun modulo può essere o meno caricato in memoria come parte del kernel, secondo le esigenze (caricamento dinamico dei moduli, all'avvio o a run-time)

❖ L'architettura a moduli è simile all'architettura a strati, ma garantisce SO più flessibili (ogni modulo può invocare funzionalità da qualsiasi altro modulo): più facili da mantenere ed evolvere

- Linux (periferiche, file system), Solaris





Approccio modulare di Solaris

Caricati e runtime
non interagiscono
in maniera
più diretta

Moduli esterni all'accesso
o runtime

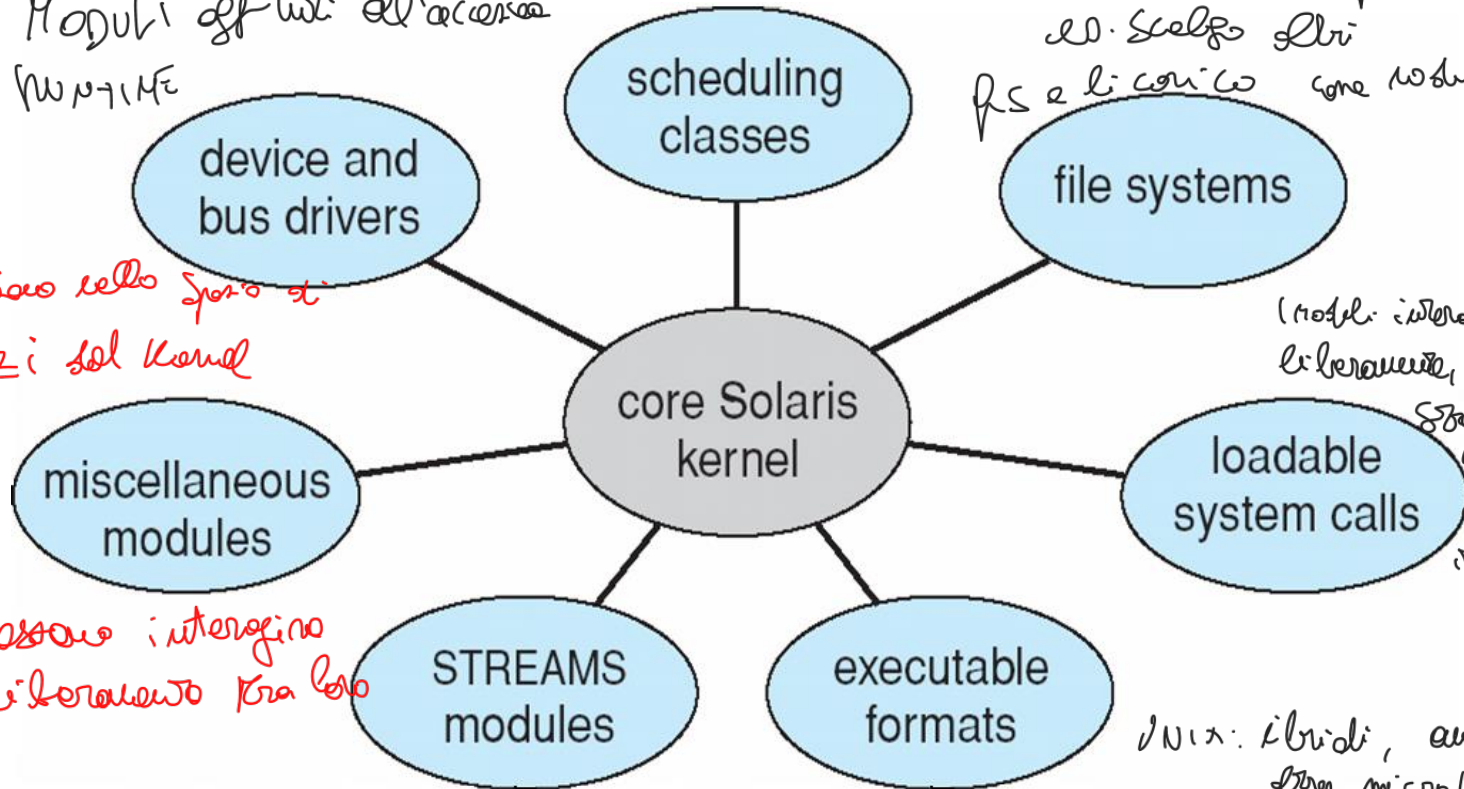
es. Selezio di
fs e li carico come moduli

Moduli sono nello spazio di
indirizzi del kernel

(moduli interagiscono
liberamente, non come
spazio di
interazione
solo con
il kernel)

Moduli possono interagire
liberamente tra loro

Unix: ibridi, anche moduli
di un microkernel



- ❖ L'organizzazione modulare lascia la possibilità al kernel di fornire i servizi essenziali, ma permette anche di implementare dinamicamente servizi aggiuntivi, specifici per il particolare sistema di calcolo

kernel o approccio stratificato o Modulare





Sistemi ibridi: S.O. studi

❖ La maggior parte dei SO attuali non adotta un modello "puro" per migliorare prestazioni, sicurezza e affidabilità

- I modelli ibridi combinano diversi approcci implementativi allo scopo di migliorare le performance, la sicurezza e l'usabilità

- I kernel di Linux e Solaris ^{monolitico modificato e ibrido} sono fondamentalmente monolitici, perché mantenere il SO in un unico spazio di indirizzamento garantisce prestazioni migliori; sono però anche modulari, per cui le nuove funzionalità possono essere aggiunte dinamicamente al kernel

presto migliori
resto su uno
spazio di indirizz.

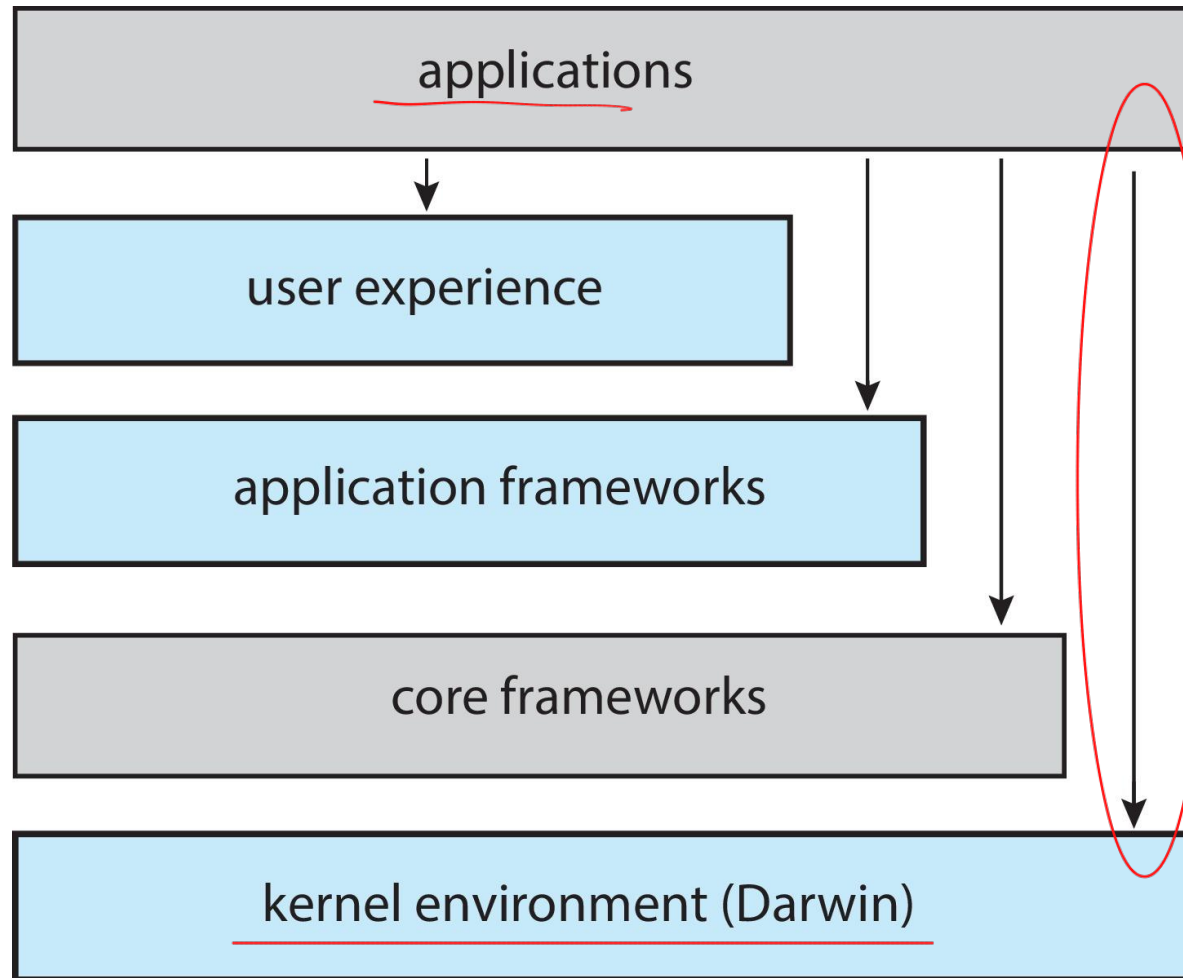
- ^{macOS, ibrido} Windows è perlopiù monolitico, ma conserva alcuni comportamenti tipici dei sistemi microkernel, tra cui il supporto per sottosistemi separati (detti personalità) che vengono eseguiti come processi in modalità utente





Strati in Mac OS e iOS

*caulio con c'è nei ringel-
stbi.*



*le gn possono
usare tutti
pti strati SW
ma possono anche
bypassarli.
Struttura
molto
monolitica*





Mac OS e iOS – 1

❖ Anche se progettati per hardware diversi, Mac OS e iOS hanno architettura simile

- Strato dell'interfaccia utente (*user experience*): **Aqua** per Mac OS, progettata per interazione con mouse e trackpad, **Springboard** per iOS, per touch screen *versione specifica per sviluppo e interfaccia*
- Strato degli ambienti applicativi: **Cocoa**, per Mac OS, **Cocoa Touch**, per iOS, forniscono un'API per i linguaggi di programmazione Objective-C e Swift *Framework, API per sviluppo programmi*
- Ambienti di base (core): supportano grafica e contenuti multimediali, inclusi Quicktime e OpenGL *API per contenuti video*
- Ambiente kernel: Darwin, include il microkernel Mach e il kernel BSD UNIX *MONOLITICO, ci si possono riferire con 2 sys call diversi, per richieste di servizi differenti*

❖ Le applicazioni possono essere progettate per sfruttare le funzionalità di *user experience* o dell'*application framework* o per aggirarle completamente *app possono rivolgersi dirett. al kernel*





Mac OS e iOS – 2

❖ Differenze significative:

- Poiché Mac OS è destinato a laptop e desktop, è compilato per architetture M1/M2 (o Intel); iOS invece è compilato su ARM
 - ▶ Il kernel di iOS garantisce risparmio energetico e gestione ottimizzata della memoria
 - ▶ Impostazioni di sicurezza più severe *il dispositivo mobile sempre governato da reti*
- Il sistema iOS è molto più restrittivo nei confronti degli sviluppatori; limita l'accesso alle API POSIX e BSD, che sono liberamente disponibili agli sviluppatori di Mac OS *Sviluppatori non hanno accesso libero al kernel*





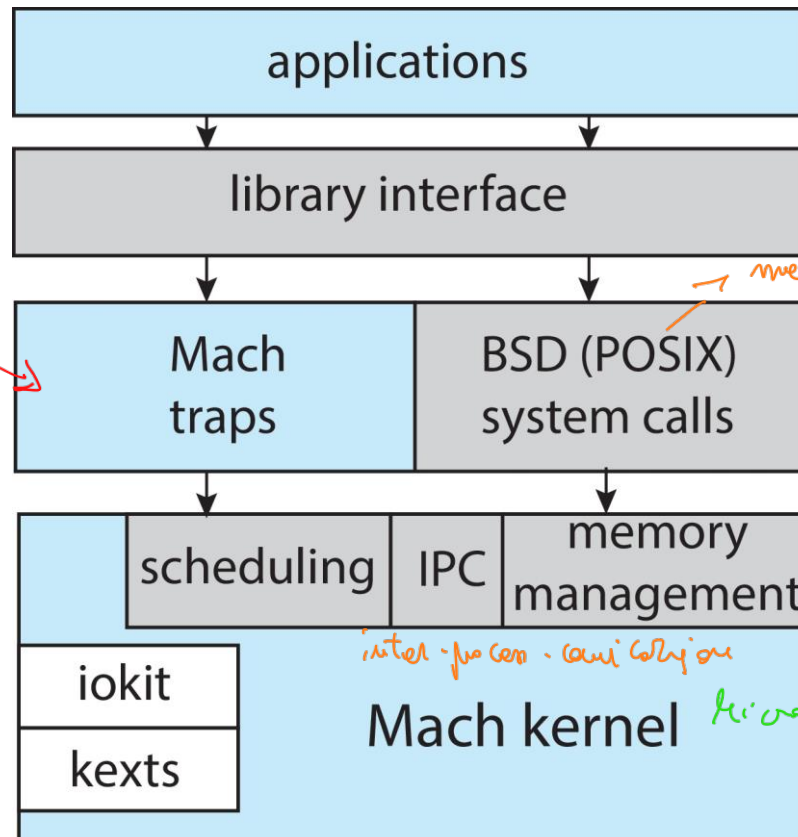
Il kernel Darwin

- ❖ Il microkernel Mach gestisce la memoria, la comunicazione fra processi (IPC), le chiamate di procedura remota (RPC) e lo scheduling dei thread *inter-process communication*
- ❖ Il kernel BSD mette a disposizione una CLI, i servizi legati al file system e alla rete e la API POSIX *CLI ci avvicina al S.O.*

BSD: FS, CLI, NETC
API POSIX

MACH: Scheduling, gestione memoria
IPC, RPC

*Syscall per Mach
si risolvono al microkernel*



*mette a disposizione CLI
al kernel ci si può rivolgere
con 2 syscall differenti*

*UNIX/BSD gestisce
Filesystem, device
drivers e
rete*

*inter-process communication
Microkernel*



Struttura di Mac OS

- ❖ Il SO Apple Mac OS adotta una struttura ibrida (stratificata) *iokit per sviluppare driver dispon.*
Next per aggiungere moduli al kernel
 - Gli strati superiori comprendono l'interfaccia utente **Aqua** e una collezione di ambienti di sviluppo e servizi per le applicazioni; l'ambiente **Cocoa** viene utilizzato per la scrittura di applicazioni native in Objective-C
 - Il kernel si trova in uno strato sottostante e, oltre a al microkernel Mach e al kernel BSD, include un kit di strumenti per lo sviluppo di driver di I/O e moduli caricabili dinamicamente, detti estensioni del kernel (kext)





Struttura di iOS

❖ SO progettato da Apple per i dispositivi mobili, *iPhone* e *iPad*

- Strutturato sul Mac OS, con l'aggiunta di funzionalità specifiche per il mobile
- Non esegue direttamente le applicazioni native di Mac OS perché "gira" su processori diversi (ARM vs. M1/M2/Intel)
- **Cocoa Touch** è un Objective-C API per lo sviluppo di app
- I **media service** costituiscono un layer per le applicazioni multimediali (grafica, audio, video)
- I **core service** forniscono supporto al cloud computing e ai database
- Il **core** è basato sul kernel di Mac OS (Darwin)

Cocoa Touch

Media Services

Core Services

Core OS

non è pensata da sviluppatori
so più protetta

non ha accesso al
API POSIX





Struttura di Android – 1

- ❖ Sviluppato dalla Open Handset Alliance – guidata da Google e costituita da oltre 30 compagnie, tra cui ASUS, HTC, Intel, Motorola, Qualcomm, T-Mobile, Samsung e NVIDIA – gestisce una grande varietà di piattaforme mobile ed è open-source *portabile*
- ❖ Costituito da una “pila” di strati software (come iOS)
- ❖ Basato su un kernel Linux modificato (al di fuori delle distribuzioni standard) *adatto ai dispositivi mobili, non lo sviluppo della GPL*
 - Gestione dei processi, della memoria, delle periferiche
 - Ampliato per includere l'ottimizzazione dei consumi energetici





Struttura di Android – 2

- ❖ L'ambiente di run-time include un insieme di librerie di base e la macchina virtuale **ART** (**Android Run Time**)
 - App sviluppate in Java con il supporto dell'Android API
 - ▶ Class file di Java compilati in bytecode e quindi tradotti in eseguibili per la ART virtual machine
 - Compilazione anticipata, per migliorare l'efficienza delle applicazioni
- ❖ Disponibile anche l'interfaccia Java nativa, **JNI**, per bypassare ART, e scrivere programmi Java con accesso più diretto all'hardware





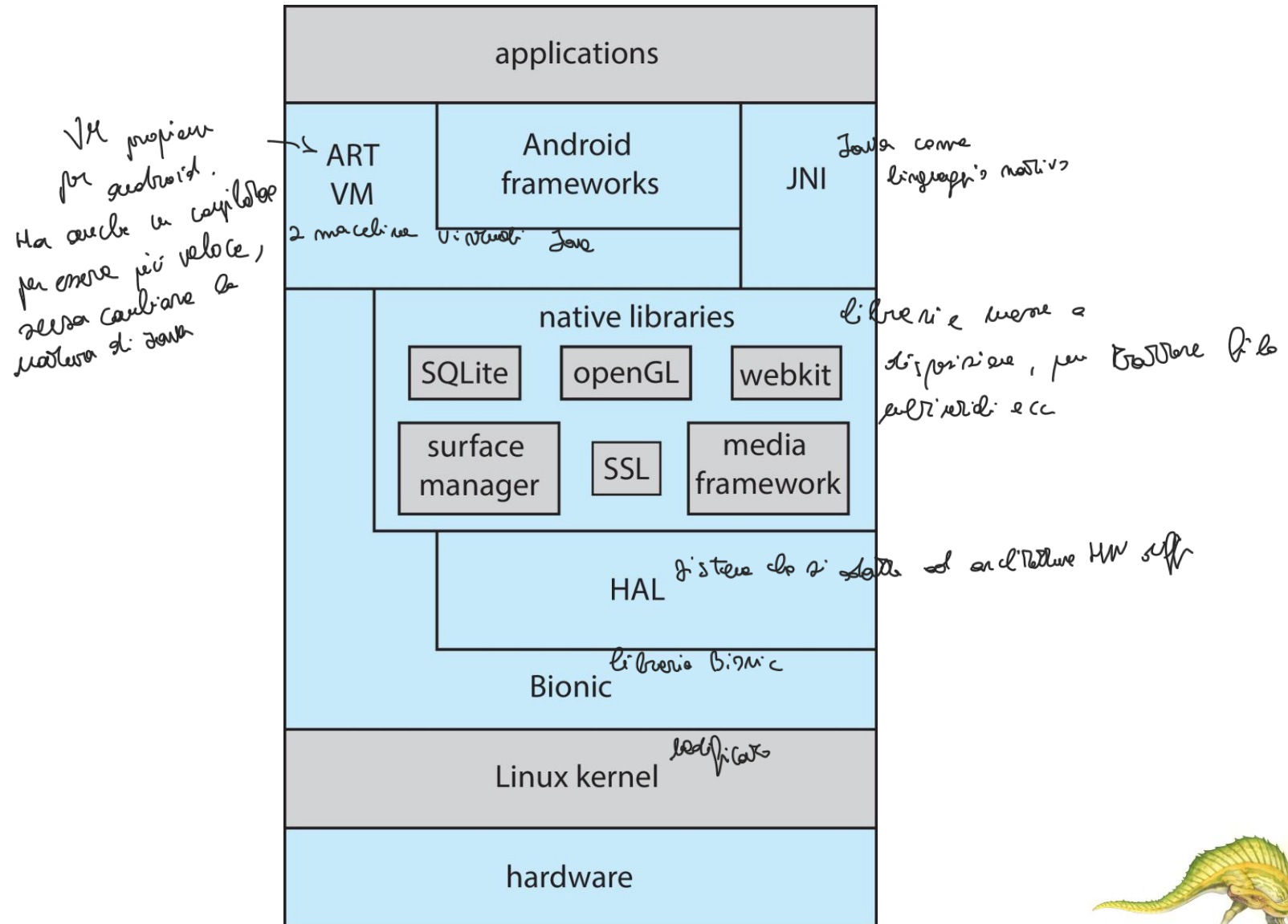
Struttura di Android – 3

- ❖ Le librerie includono ambienti per lo sviluppo di browser ([webkit](#)), di supporto ai database ([SQLite](#)) e all'accesso alla rete ([SSL](#)), ambienti multimediali e una libreria C standard minimale ([Bionic](#), libera da GPL)
- ❖ Poiché Android può essere eseguito su un numero quasi illimitato di dispositivi, include uno strato di astrazione dell'hardware ([HAL](#) – Hardware Abstraction Layer) *so si deve scrivere per tutte architetture diverse*
- ❖ Kernel Linux modificato: gestione e allocazione di memoria, IPC (fornita da [Binder](#))
↳ inter-process communication





Struttura di Android – 4





Generazione e avvio del SO – 1

- ❖ I sistemi operativi sono generalmente progettati per funzionare su una classe di architetture, che possono disporre di una grande varietà di periferiche
- ❖ Comunemente, il sistema operativo è già installato quando si acquista un computer
 - Nulla vieta tuttavia che se ne possano (*realizzare e*) installare altri
 - Se si genera un sistema operativo *from scratch*, occorre (*scrivere il codice sorgente del sistema operativo*)
 - ▶ Configurare il sistema operativo per il sistema di calcolo su cui verrà eseguito
 - ▶ Compilare il sistema operativo; *è un SW*
 - ▶ Installare il sistema operativo
 - ▶ Avviare il computer con il nuovo SO

*serve file di config.
che specifica sul sistema
a quale driver gli servono*

Debugging e performance tuning, libero da errori ed efficiente





Generazione e avvio del SO – 2

devo dirlo
al SO se
posso implementare
alcune funzioni, es.
multi-thread

es. pium del CPU 386, o 486 no coprocess
moderata, no oper. con float
ma simili. da 386
integrati.

In particolare, per generare un SO è necessario...

- leggere da un file le informazioni riguardanti la configurazione specifica del sistema o, alternativamente... *sub: reading HW oggi come è diviso in 2 CPU virtuali, durante lo stesso*
- esplorare il sistema di calcolo per determinarne i componenti *non è un*

❖ Informazioni necessarie:

- Tipo di CPU impiegate e opzioni installate *SO vede es. 2 CPU, gestite dallo scheduler*
- Tipo di formattazione del dispositivo di memoria di avvio (es. numero di partizioni) *es. dove è il blocco d'avvio*
- Quantità di memoria disponibile
- Dispositivi disponibili *collegati*
- Scelta delle politiche (algoritmi di scheduling, numero massimo di processi sostenibili, etc.) *scelte SW*

o'col-time

❖ Generare il sistema serve per:

File config: info
su HW e politiche SW

- ottenere un kernel system-specific
- garantire un sistema più efficiente perché customizzato

la nostra macchina





Generazione e avvio di Linux

- ❖ Accedere a <http://www.kernel.org> per scaricare il codice sorgente di Linux
- ❖ Configurare il kernel tramite **make menuconfig**; si genera il file di configurazione **.config** *nascono i file di sistema*
- ❖ Compilare il kernel usando **make** (che utilizza i parametri di configurazione presenti in **.config**)
 - Si produce **vmlinuz**, l'immagine del kernel
- ❖ Compilare i moduli del kernel tramite **make modules** (che utilizza i parametri di configurazione presenti in **.config**) *affaccio moduli con driver specifici*
- ❖ Installare i moduli del kernel in **vmlinuz** tramite **make modules_install** *linko i moduli al kernel generato*
- ❖ Installare il nuovo kernel sul sistema tramite **make install**





Avvio del sistema operativo – 1

eccetto il SO

- ❖ Quando si accende un computer, l'esecuzione inizia da una posizione di memoria prefissata
- ❖ Il sistema operativo deve essere reso disponibile all'hardware affinché l'hardware possa avviarlo

- Piccolo pezzo di codice – **bootstrap loader** memorizzato in ROM o nella EEPROM, individua il kernel, lo carica in memoria e lo avvia

caricato dal BIOS

programma minimale, ha info a sufficienza per andare al blocco di avvio, dove c'è il vero e proprio bootstrap loader e viene caricato in memoria

Caricare l'indirizzo nella memoria di lavoro

- A volte il bootstrap avviene in due passi: si carica il **bootloader**, contenuto nel **blocco di avvio**, che provvede a caricare il kernel

POST e caricamento kernel in RAM

- ❖ I sistemi moderni sostituiscono il BIOS con l'interfaccia **UEFI (Unified Extensible Firmware Interface)**

Comandi a 64 bit, 601 e fa tutto insieme

- ❖ I comuni bootstrap, come **GRUB** di Linux, permettono la selezione del kernel in varie versioni, con differenti opzioni e da memorie diverse

Selezione di diversi kernel
GRUB





Avvio del sistema operativo – 2

❖ In Linux:

- Per risparmiare spazio e ridurre il tempo di avvio, l'immagine del kernel è un file compresso, che viene estratto in RAM *KERNEL viene ZIPATO, più facile da caricare in RAM, PC operativo prima*
- Il bootloader crea un file system temporaneo in RAM, chiamato **initramfs**: contiene driver e moduli del kernel *crea un oggetto che è un file system che viene in RAM FS non gestiscono solo kernel MASSA*
FS che stanno in RAM su LINUX, non gestiscono solo memoria di massa
- Dopo l'avvio del kernel e l'installazione dei driver necessari, viene montato il file system principale su memoria di massa (nella posizione appropriata)
- Si crea il processo iniziale di sistema **systemd** *padre di tutti i processi, PID 1 la radice dell'albero process*
- Si presenta il prompt di login

INITD per UNIX





Debugging del sistema operativo – 1

- ❖ Il **debugging** è l'attività di individuazione e risoluzione di errori nel sistema, i cosiddetti **bachi** (*bug*); comprende anche il **performance tuning** *influenza molto la velocità e l'efficienza della macchina*
- ❖ Il SO può generare **file di log** che danno informazioni sugli errori rilevati durante l'esecuzione di un processo *magari interpretato dal debugger*
- ❖ Il SO può anche acquisire e memorizzare in un file un'immagine del contenuto della memoria utilizzata dal processo, chiamata **core dump** *memoria del processo in un file* *si chiama così perché è un dump*
- ❖ Il core dump, un'istantanea post-mortem dello stato del processo al momento della terminazione anomala, può essere passato a un programma di supporto, un debugger, in grado di rilevare l'errore *Dump*

Salvo che file nella memoria di RAM





Debugging del sistema operativo – 2

❖ Un guasto nel kernel viene chiamato **crash**

- Come avviene per i processi utente, l'informazione riguardante l'errore viene salvata in un file di log, mentre lo stato della memoria viene salvato in un'immagine su memoria di massa (**crash dump**)

non fare uso tecniche FS standard

non sapere quale servizio del SO ha generato l'errore

- Tecniche più complesse per la natura delle attività svolte dal kernel

es. nel caso sia stato il FS a fare crash non può essere per salvare crash dump, forse danni

- ▶ Il salvataggio del crash dump su file potrebbe essere rischioso se il kernel è in stato inconsistente (es.: malfunzionamento del codice kernel relativo allo stesso file system)
- ▶ Il dump viene salvato in un'area di memoria dedicata e da lì recuperato per non rischiare di compromettere il file system





Debugging del sistema operativo – 3

❖ Legge di Kernigham

“Il debugging è due volte più difficile rispetto alla stesura del codice. Di conseguenza, chi scrive il codice nella maniera più intelligente possibile non è, per definizione, abbastanza intelligente per eseguire il debugging.”

- ❖ Anche i problemi che condizionano le prestazioni sono considerati bachi, quindi il debugging include anche il *performance tuning*, che ha lo scopo di eliminare i colli di bottiglia del sistema di calcolo





Performance tuning – 1

❖ Il **performance tuning** è l'insieme delle tecniche atte ad ottimizzare le prestazioni del sistema, eliminando i colli di bottiglia *System da monitorare meglio.*

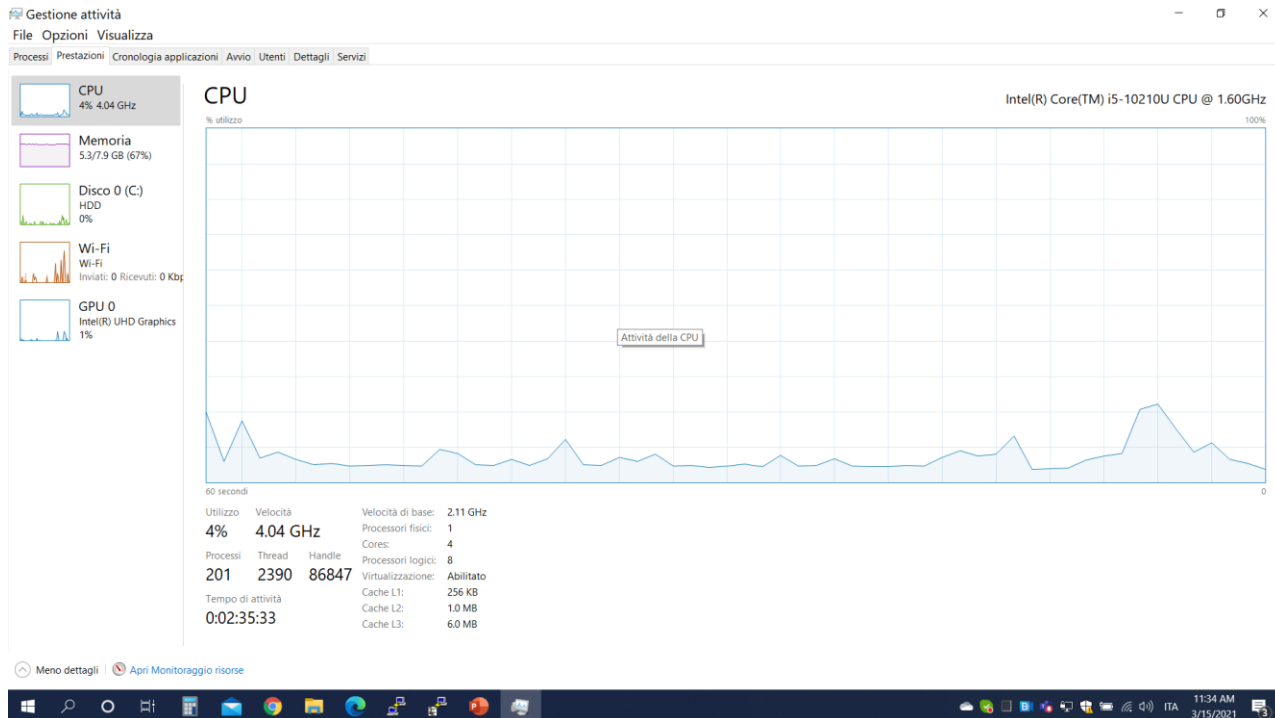
- Esecuzione di codice che effettui misurazioni sul comportamento del sistema e salvi i dati su un file di log
 - ▶ Analisi dei dati salvati nel file di log, che descrive tutti gli eventi di rilievo, per identificare ostacoli ed inefficienze
- Introduzione, all'interno del SO, di strumenti interattivi che permettano ad amministratore ed utenti di monitorare il sistema (es., istruzione **top** di UNIX: mostra le risorse di sistema impiegate ed un elenco ordinato dei principali processi che le utilizzano) *monitorare meglio sistema*
- **Profiling**, campionamento periodico dell'IP (Instruction Pointer) per valutare, per esempio, quali chiamate di sistema siano utilizzate maggiormente

es. instruction pointer, ci che è contenuto negli indirizzi memoria viene chiamato IP
Se una syscall viene chiamata molto, però 2.96 ottimizzare il codice





Performance tuning – 2



- ❖ In Windows 10, selezionando la voce **Gestione attività** (dal menù ottenuto posizionandosi sulla barra delle applicazioni e premendo il tasto destro) si può visualizzare, in tempo reale, l'utilizzo delle risorse principali





Linux BCC – 1

- ❖ Il debugging a livello di interazioni tra codice utente e codice kernel è quasi impossibile senza un set di strumenti integrati al SO
- ❖ **BCC** (BPF Compiler Collection) è un ricco toolkit che fornisce funzionalità di tracciamento per Linux
- ❖ Ad esempio, **disksnoop.py** traccia l'attività di I/O su disco

TIME(s)	<i>che tipo di oper.</i> T	<i>byte operati dell'oper.</i> BYTES	<i>quanto, in base a quale solo la code</i> LAT(ms)
1946.29186700	R	8	0.27
1946.33965000	R	8	0.26
1948.34585000	W	8192	0.96
1950.43251000	R	4096	0.56
1951.74121000	R	4096	0.35

- ❖ BCC include strumenti di monitoraggio di tutti i moduli del sistema



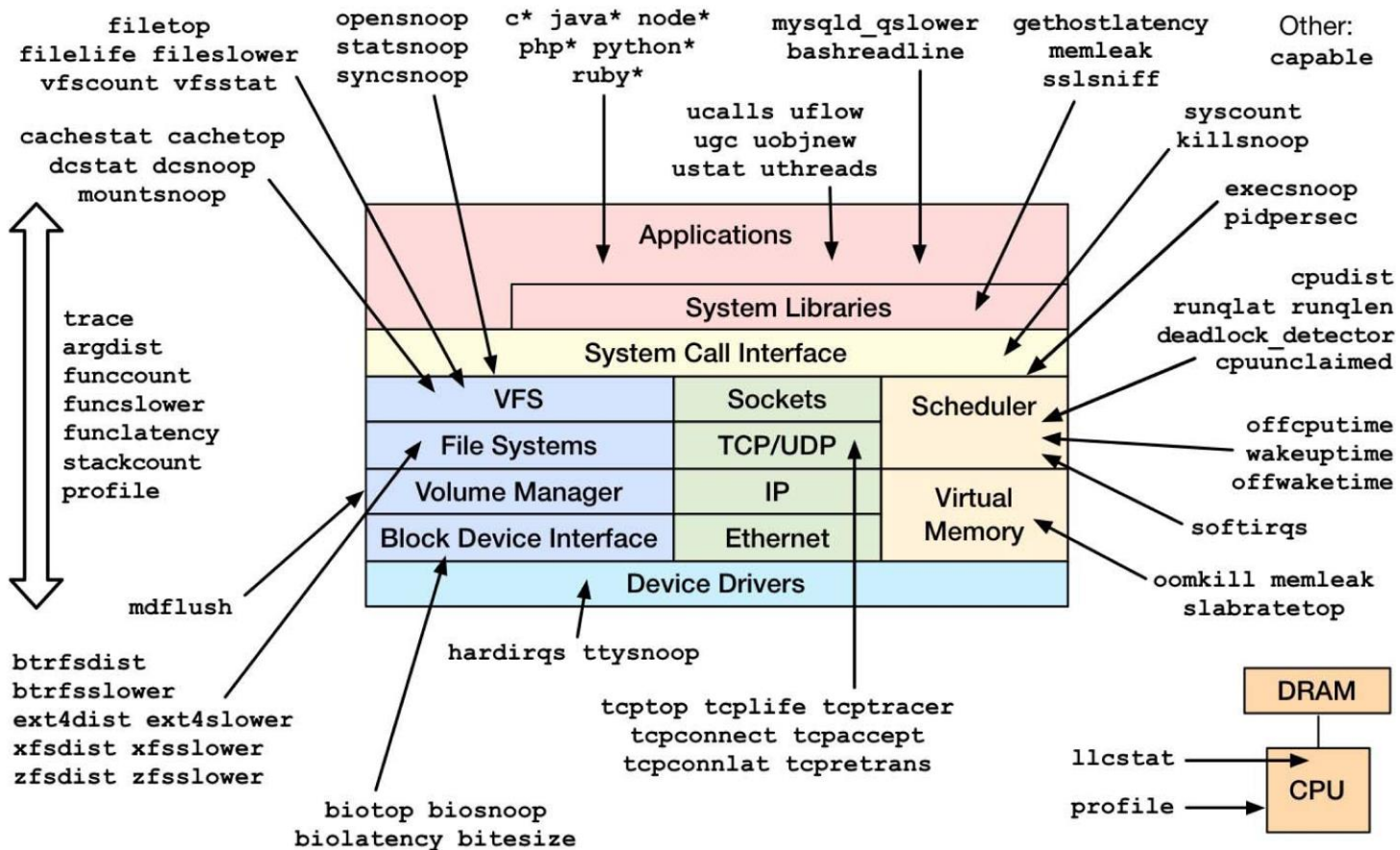


Linux BCC – 2

Strumenti per monitorare

*questo le parti del
nostro sistema*

Linux bcc/BPF Tracing Tools



<https://github.com/iovisor/bcc#tools> 2017





Il sottosistema Windows per Linux – 1

- ❖ Windows possiede un'architettura ibrida che fornisce la possibilità di includere sottosistemi, in particolare capaci di emulare diversi SO
 - I sottosistemi vengono eseguiti in modalità utente e comunicano con il kernel di Windows per fornire i servizi
 - Il sottosistema di Windows per Linux si chiama **WSL** e consente alle applicazioni Linux native (specificate come binari ELF) di essere eseguite su Windows 10
 - Occorre avviare l'applicazione Windows **bash.exe**, che presenta all'utente una shell **bash** che esegue Linux
 - Internamente, WSL crea un'istanza Linux costituita dal processo **init**, che a sua volta crea la shell **bash** che esegue le applicazioni Linux native

→ padre di tutti i processi





Il sottosistema Windows per Linux – 2

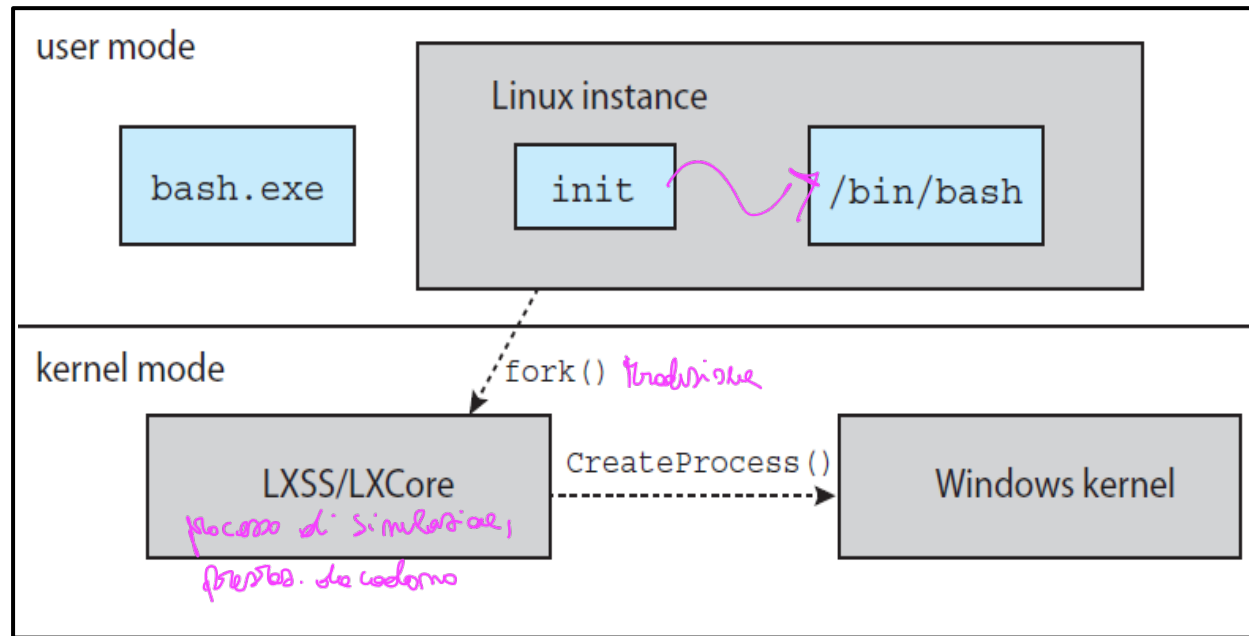
- ❖ Ognuno di questi processi viene eseguito in un processo **Pico** di Windows, che carica il file binario di Linux nel proprio spazio degli indirizzi
 - I processi Pico comunicano con i servizi del kernel LXCORE e LXSS, per tradurre le chiamate di sistema di Linux nelle chiamate di sistema di Windows, quando possibile *syscall diverse per nome e/o per implementazione*
 - In mancanza di corrispondenza diretta, LXSS deve fornire una funzionalità equivalente, eventualmente gestendola in proprio
 - **Esempio:** La chiamata di sistema `fork()` di Linux non corrisponde esattamente alla `CreateProcess()` di Windows; LXSS esegue parte del lavoro di `fork()` e, solo successivamente, invoca `CreateProcess()`

adulterio simulato, WLS deve rendere equivalenti le syscall, è un adulterio SIMULATO





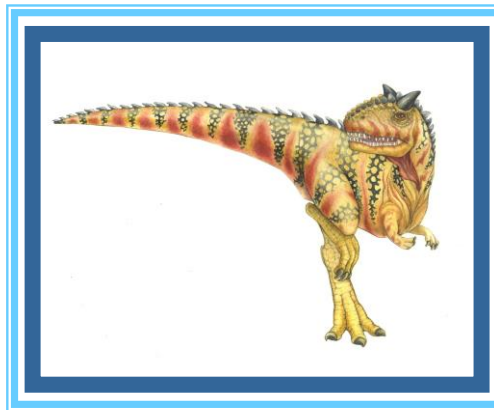
Il sottosistema Windows per Linux – 3



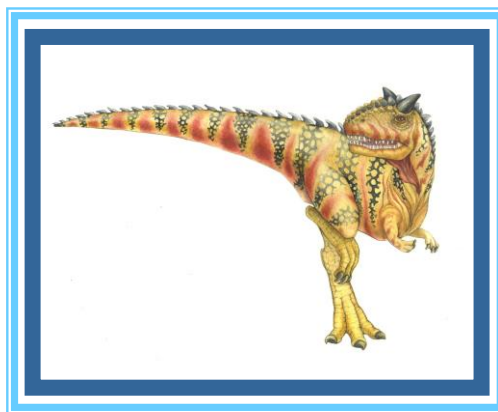
Servizio Kernel



Fine del Capitolo 2



I processi





Obiettivi

- ❖ Introdurre la nozione di processo – un programma in esecuzione – e identificarne i diversi componenti
- ❖ Descrivere come i processi vengano generati e terminati e come si sviluppino programmi utilizzando chiamate di sistema
- ❖ Descrivere la comunicazione tra processi mediante memoria condivisa e scambio di messaggi
- ❖ Descrivere la comunicazione nei sistemi client–server





Sommario

- ❖ Definizione di processo *durante la loro vita cambiano il loro stato*
- ❖ Scheduling dei processi *diversi tipi di scheduler, più uno, quello attuale
Terminare*
- ❖ Operazioni sui processi *es. Fork*
- ❖ Processi cooperanti
- ❖ Comunicazione fra processi
- ❖ POSIX come esempio di sistema IPC
- ❖ Comunicazioni in sistemi client-server





Definizione di processo – 1

*Un sistema batch, può essere visto
come una volta*

- ❖ Un SO esegue programmi di varia natura
 - Sistemi batch: **job** *possono essere schedati p. esempio*
 - Sistemi time-sharing: **processi utente** o **task** *dopo che i sys si sono esauriti, con multitasking*
 - Sistemi che supportano il multithreading: **thread**

- ❖ I libri di testo impiegano indifferentemente i termini **job** o **processo** (o **task**) *noi usiamo processo, e thread quando un processo può essere multithread*

- ❖ **Processo**: programma in esecuzione

Def. processo:

- Un programma in esecuzione – l'esecuzione di un processo deve avvenire in modo sequenziale
- ...o, in alternativa, la sequenza di eventi che avvengono in un elaboratore quando opera sotto il controllo di un particolare programma

caching per migliorare prestazioni

esecuzione programmi è locale in tempo e spazio. I dati vicini in tempi vicini

• *Stato: stato corrente della CPU*

eseguire program, e verificare eventi che modificano lo stato dell'elaboratore





Definizione di processo – 2

- ❖ Un processo include: *codice eseguibile che descrive il processo*
 - una **sezione di testo** (il codice del programma da eseguire)
 - una **sezione dati** (variabili globali)
 - lo **stack** (dati temporanei – parametri per i sottoprogrammi, variabili locali e indirizzi di ritorno, un record di attivazione) *usa tutta la parte della memoria del processo*
 - uno **heap** (letteralmente "mucchio", grande quantità – memoria dinamicamente allocata durante l'esecuzione del task) *collegiate informazioni*
 - il **program counter** ed il contenuto dei registri della CPU *memoria all'allocazione di manipolazione*
- ❖ Il programma è un'entità "passiva" memorizzata su disco (un **file eseguibile**), un processo è "attivo" *differenza processo / programma*
 - I programmi divengono processi quando vengono caricati nella memoria principale *usa tutta la dinamicità (processo) / staticità (programma)*
- ❖ L'esecuzione di un programma viene attivata dal doppio click del mouse in una GUI, o immettendo il nome del file eseguibile da linea di comando *processo può cambiare dato registro*

Ci sono dati che servono per invocare funzioni. Record di attivazione: dati per fare una chiamata o funzione

Registro che contiene processo istruzione istruzione

Contenuto processo: deve essere salvato prima di eseguire uno nuovo





Processo allocato in memoria

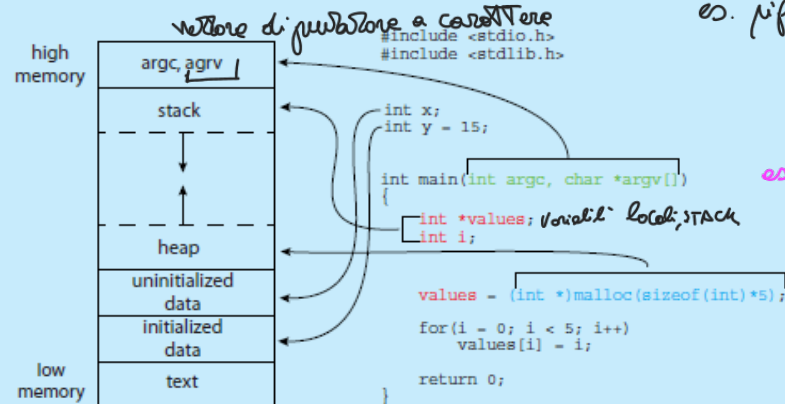
Spazio indirizzi processo

layout di un programma C in memoria

MEMORY LAYOUT OF A C PROGRAM

The figure shown below illustrates the layout of a C program in memory, highlighting how the different sections of a process relate to an actual C program. This figure is similar to the general concept of a process in memory as shown in Figure 3.1, with a few differences:

- The global data section is divided into different sections for (a) initialized data and (b) uninitialized data.
- A separate section is provided for the argc and argv parameters passed to the main() function.



es. printf(5) è una stringa, deve convertirlo ad intero con atoi

malloc, Heap

SIZEC()

The GNU size command can be used to determine the size (in bytes) of some of these sections. Assuming the name of the executable file of the above C program is memory, the following is the output generated by entering the command size memory:

occupazione di memoria delle sezioni

text	data	bss	dec	hex	filename
1158	284	8	1450	5aa	memory

non inizializzato

inizializzato

The data field refers to uninitialized data, and bss refers to initialized data. (bss is a historical term referring to *block started by symbol*.) The dec and hex values are the sum of the three sections represented in decimal and hexadecimal, respectively.





Definizione di processo – 3

ed un programma posso corrispondere più istanze, più processi diversi, che si differenziano perché operano su dati diversi

- ❖ Un programma può corrispondere a diversi processi
 - Si pensi a un insieme di utenti che utilizzano uno stesso editor di testo in relazione a file diversi
- ❖ Quindi, un programma descrive non un processo, ma un insieme di processi – *istanze del programma* – ognuno dei quali è relativo all'esecuzione del programma da parte dell'elaboratore per un particolare insieme di dati in ingresso *le istanze si differenziano per*
- ❖ Un processo può rappresentare un ambiente di esecuzione di altri processi *processo che esegue altri processi*
 - Java Virtual Machine e programmi scritti in Java *che eseguono altri processi*





Stato del processo – 1

Processo con il suo stato durante l'esecuzione

PRODOTTORE A STATI FINITI: grafico con nodi ed archi, e il proprio simbolo del contenuto dello stato e dell'arco

- ❖ Mentre viene eseguito, un processo è soggetto a transizioni di stato, definite in parte dall'attività corrente del processo ed in parte da eventi esterni, asincroni rispetto alla sua esecuzione:

- **New** (nuovo): il processo viene creato in RAM. Da accesso alla contesa attiva dello /della CPU
- **Running** (in esecuzione): se ne eseguono le istruzioni *ottiene l'utilizzo della CPU*
- **Waiting** (in attesa): il processo è in attesa di un evento *es. da I/O o da terminazione un poco prima di gli ritorni dati*
- **Ready** (pronto): il processo è in attesa di essere assegnato ad un processore *è nella coda dei processi pronti, non tocca ancora a lui*
- **Terminated** (terminato): il processo ha terminato la propria esecuzione *o in maniera corretta (exit / return) restituendo le risorse che deteneva; oppure con un abort. In ogni caso restituisce le risorse*

che come in un processore permette l'esecuzione di un solo processo (parallelismo) in un certo istante di tempo

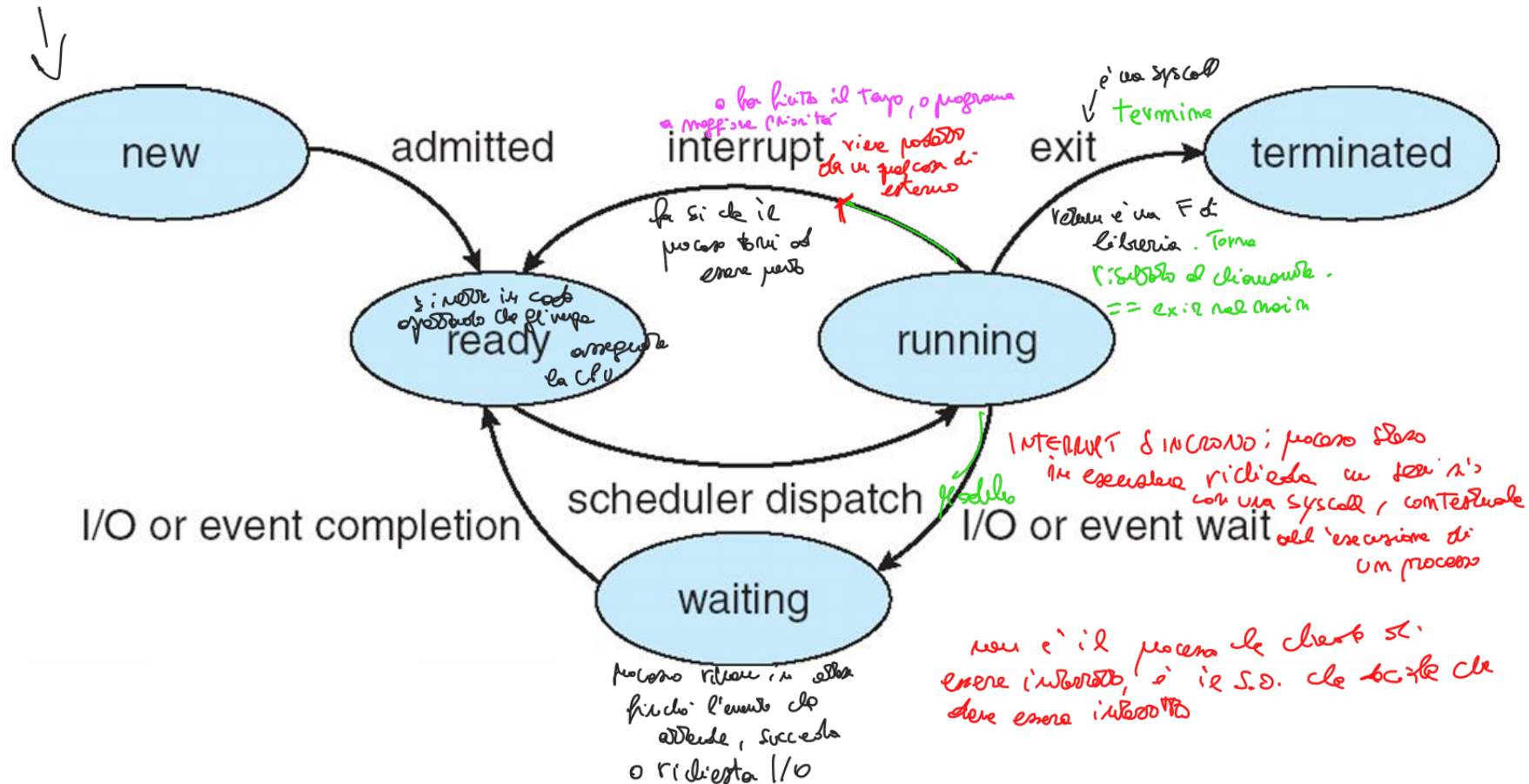
Tempo atteso
valutato nella Ready Queue





Diagramma di transizione di stato di un processo

Processo creato, che lo aspetta era uno scheduler ... Lancio, stoppo click di un process, portava alla console prima succedeva tra vari programmi
Questa, non viene eseguito subito.





Stato del processo – 2

- ❖ Normalmente, nel sistema, sono presenti più processi nei diversi stati
 - Necessità di scegliere opportunamente processo/i e transizione/i da eseguire
- ❖ Lo stato *new* corrisponde alla creazione di un nuovo processo (~~al caricamento del codice da disco in RAM~~)
 - La transizione dallo stato *new* a *ready* avviene quando il SO (scheduler a lungo/medio termine) ammette il nuovo processo alla contesa attiva per la CPU (inclusione nella ready queue) ←
- ❖ La transizione dallo stato *ready* a *running* avviene – ad opera del *Blocco process in exec*
dispatcher – quando, in seguito al blocco del processo in esecuzione, il processo viene scelto, fra tutti i processi pronti, per essere eseguito





Stato del processo – 3

- ❖ La transizione da *running* a *ready*, chiamata **revoca** o **prerilascio** avviene...
 - ..(nello scheduling a priorità), quando arriva al sistema un processo con priorità maggiore
 - ...nei sistemi a partizione di tempo, per esaurimento del quanto *di Tempo*
 - ...al verificarsi di un interrupt esterno (asincrono) *rispetto al tempo*
- ❖ La transizione da *running* a *waiting* avviene per la richiesta di un servizio di I/O al SO (o per l'attesa di un qualche evento), mentre il passaggio da *waiting* a *ready* avviene al termine del servizio (al verificarsi dell'evento)





Stato del processo – 4

❖ Lo stato *terminated* si raggiunge per:

- terminazione normale, con chiamata al SO per indicare il completamento delle attività *es. exit / return*
- terminazione anomala *errore*
 - ▶ Uso scorretto delle risorse (superamento dei limiti di memoria, superamento del tempo massimo di utilizzo della CPU, etc.) *es. accesso ad indirizzo memoria che non esiste*
 - ▶ Esecuzione di istruzioni non consentite o non valide (trap) *es. divisione per 0, loop infinito, riempimento dello stack*
- In entrambi i casi, le risorse riusabili, previamente allocate al processo terminato, vengono riprese in carico dal SO per usi successivi





Process Control Block – 1

Struttura dati
per gestire processi

- ❖ Ad ogni processo è associata una struttura dati del kernel, il descrittore di processo – **Process Control Block (PCB)** info che il S.O. deve usare per gestire processo
è una struct, un record, dati aggregati con tipi di varia natura
- ❖ I dati utilizzati per descrivere i processi dipendono dal SO e dalla struttura della macchina fisica il PCB varia tra i vari PCB
- ❖ Le informazioni contenute nei PCB sono vitali per il SO; devono essere memorizzate in un'area di memoria accessibile solo dal nucleo del sistema operativo
Solo del kernel



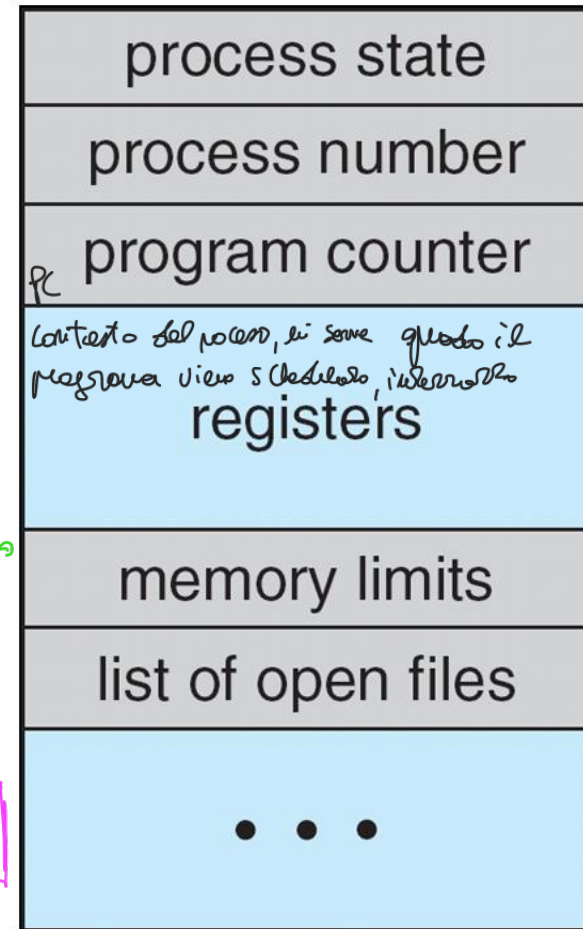


processi identificati con un numero, PID

Process Control Block – 2

❖ Informazione associata a ciascun processo:

- Stato del processo
- Nome (numero) del processo
- **Contesto del processo:** ^{PC del processo} program counter, registri della CPU (accumulatori, registri indice, stack pointer, registri di controllo)
- Informazioni sullo scheduling della CPU (^{info sulla priorità processo, dove ho memorizzato in RAM i processi.} priorità, puntatori alle code di scheduling)
- Informazioni sulla gestione della memoria allocata al processo (registri base e limite, ^{della memoria} tabella delle pagine o dei segmenti)
- Informazioni di contabilizzazione delle risorse: utente proprietario, tempo di utilizzo della CPU, tempo trascorso dall'inizio dell'esecuzione, etc. ^{info di contabilizzazione}
- Informazioni sull'I/O: elenco dispositivi assegnati al processo, file aperti, etc.



ora processi sono immaginati, varie pagine, un sacco di info relative a ciascun processo





Thread – 1

possiamo individuare 2 livelli di processo.
L'1° definisce le risorse, ma il suo intento
possono essere più percorsi

es. browser, sottoprocessi

FTP per prendere pagine
visuale e gestione delle pagine
← thread che fa download pagina

- ❖ Alla base del concetto di **thread** sta la constatazione che la definizione di processo è basata su due aspetti:
 - Possesso delle risorse
 - Esecuzione *del codice*
- ❖ I due aspetti sono indipendenti e come tali possono essere gestiti dal SO
 - L'elemento che viene eseguito è il thread
 - L'elemento che possiede le risorse è il processo
- ❖ Il termine **multithreading** è utilizzato per descrivere la situazione in cui ad un processo sono associati più thread

Thread viene eseguito
Processo definisce le risorse

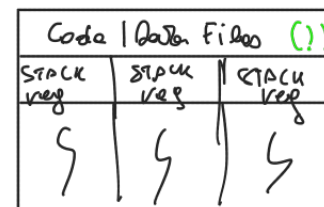
la contesa globale CPU è tra Thread, non processi

Ogni thread ha il suo counter,
la sua memoria dinamica e ... il suo contesto

Tempo context switch,
meno pesante



thread



Context switch
è l'operazione in cui la
CPU non fa niente





Thread – 2

- ❖ Tutti i thread condividono le risorse del processo, risiedono nello stesso spazio di indirizzamento ed hanno accesso agli stessi dati
- ❖ Ad ogni thread è però associato un program counter, uno stato di esecuzione, uno spazio di memoria per le variabili locali, uno stack, un contesto *e i registri (!)*
⇒ **un descrittore**
- { ❖ Le informazioni associate al thread sono poche: le operazioni di cambio di contesto, di creazione e terminazione sono più semplici *e meno pesanti, dunque più veloci*





Processi in LINUX – 1

- ❖ Il PCB, o meglio il TCB (nel gergo di Linux si parla di ^{non proven}task), è rappresentato dalla struttura C `task_struct` che si trova nel file `linux/sched.h` e che contiene fra l'altro...

/usr/include/sched.h

- Identificatore del processo (PID)
- Stato del processo
- Informazioni per scheduling e gestione della memoria
- Lista dei file aperti
- Puntatori a processo padre ed eventuali processi figli

risorse allo spazio degli indirizzi del processo

Indirizzo degli indirizzi

```
pid_t pid; /* process identifier */
long state; /* state of the process */
struct sched_entity se; /* scheduling information */
struct task_struct *parent; /* this process's parent */
struct list_head *children; /* this process's children */
struct files_struct *files; /* list of open files */
struct mm_struct *mm; /* address space of this process */
```

Indirizzo padre e processi figli

Indirizzo a lista di file aperti

Alcuni campi della struttura `task_struct`





Processi in LINUX – 2

Coda Processi: Linux è un doppio coda concorrente, mantenendo un puntatore al PCB del processo attualmente in esecuzione

- ❖ La ready queue è realizzata mediante una lista doppiamente concatenata di `task_struct` ed il kernel mantiene un puntatore di nome `current` al processo attualmente in esecuzione
- ❖ Per esempio... se il SO deve cambiare lo stato del processo in esecuzione in nuovo_stato, l'istruzione da eseguire è:

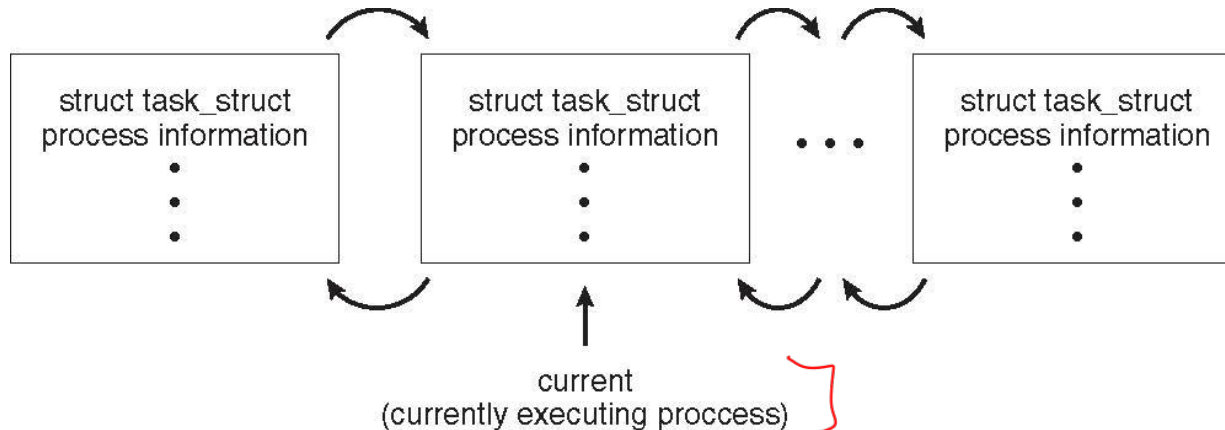
puntatore al PCB del processo corrente

accesso alla
struttura

es. da running a ready
o in waiting
o Terminated

che è il primo il
quello di tempo

`current->state = nuovo_stato;`





Scheduling dei processi – 1

Molto più come in esecuzione dello CPU.

- ❖ Meccanismo messo in atto per massimizzare l'utilizzo della CPU, che consiste nel passare rapidamente dall'esecuzione di un processo al successivo, garantendo il time sharing
- ❖ È il modulo del SO noto come **CPU scheduler** che esegue questo compito
include kernel di scheduler, affetto context switch, salva reg. in, processo uscire nel PCB e carica reg per processo entrante
- ❖ Code per lo scheduling dei processi
 - **Ready queue** (Coda dei processi pronti) — Insieme dei processi pronti ed in attesa di essere eseguiti, che risiedono in memoria centrale
 - **Code di attesa** — Insieme dei processi in attesa per un dispositivo di I/O o per il verificarsi di un evento
nel corso della loro vita i processi sono anche in altre code.





Scheduling dei processi – 2

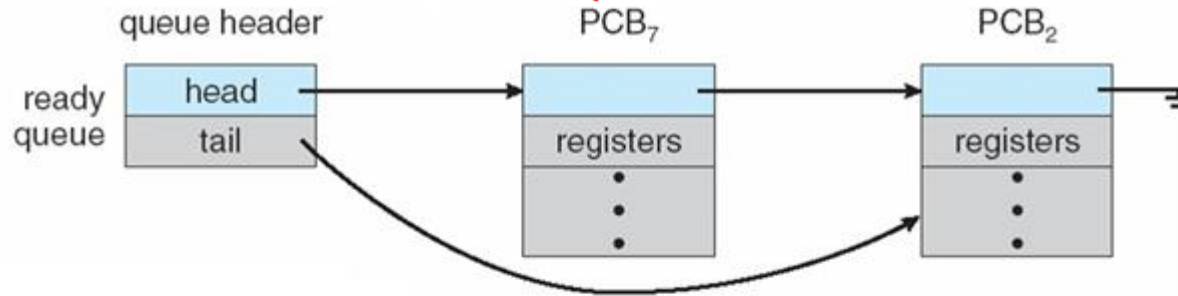
- ❖ I processi migrano fra le varie code
- ❖ Un nuovo processo si colloca inizialmente nella coda dei processi pronti, *Ready queue* dove attende fino a quando il dispatcher non lo seleziona per l'esecuzione
- ❖ Un processo rilascia la CPU (esce dalla propria fase di running) a causa di uno dei seguenti eventi:
 - emette una richiesta di I/O e viene inserito nella relativa coda *Waiting queue*
 - crea un nuovo processo e ne attende la terminazione *attende return o terminazione del figlio*
 - viene rimosso forzatamente a causa di un'interruzione e reinserito nella ready queue *es. esaurito il tempo o processo con priorità più alta*





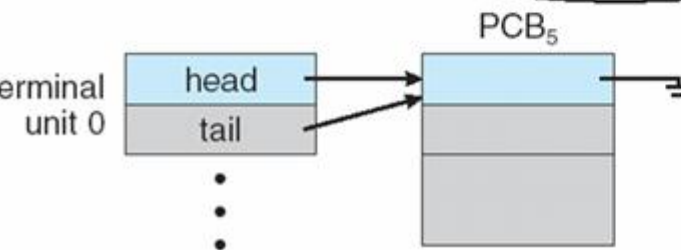
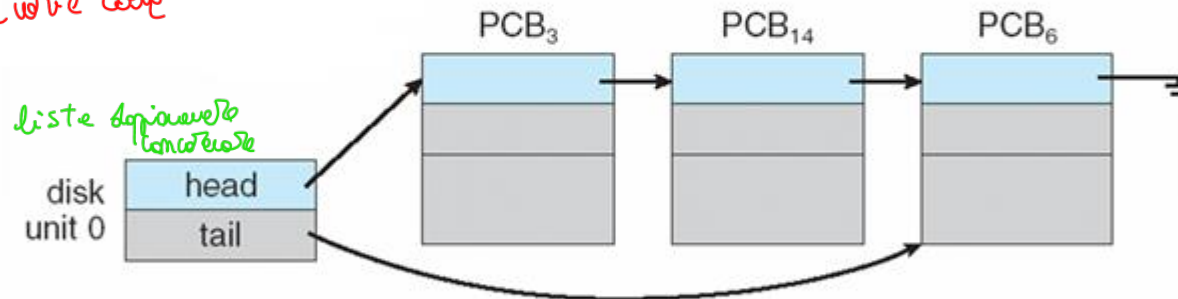
Ready queue e code ai dispositivi di I/O

Processi pronti per essere eseguiti



Quando viene il processo migrato tra le varie code

abbiamo una coda per ciascuna periferica collegata al nostro sistema di calcolo



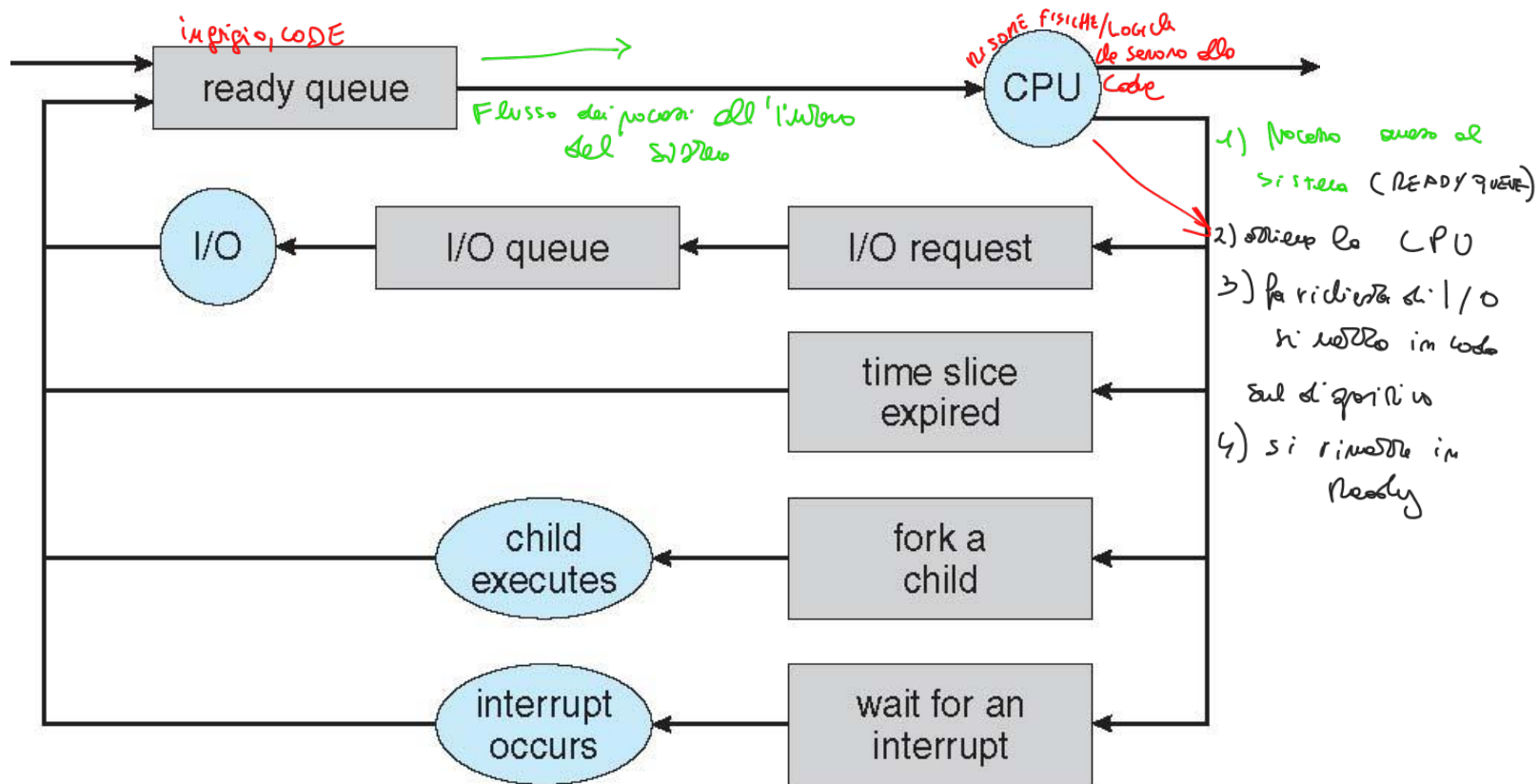
wait queues

*Sistema alla single CPU,
un solo processo running,
non posso essere più in waiting
e/o ready
decisione dello scheduler per
farli accedere alla
CPU*





Diagramma per lo scheduling dei processi



- I riquadri rappresentano le code
- Le ellissi rappresentano le "risorse" che servono le code
- Le frecce indicano il flusso dei processi nel sistema





Tipi di scheduler – 1

un tempo c'era lo scheduler a tempo fisso, lo decideva il massimo process
primario 300 nei sistemi batch

processi, spin logico assegnato > di quello fisico non aveva

❖ **Scheduler a breve termine** (o scheduler della CPU):
seleziona quale processo debba essere eseguito successivamente ed alloca la CPU (frequenza < 100msec)

❖ Lo **scheduler a medio termine** (o **swapper**) rimuove temporaneamente i processi dalla memoria (e dalla contesa attiva per la CPU) e regola il grado di multi-programmazione

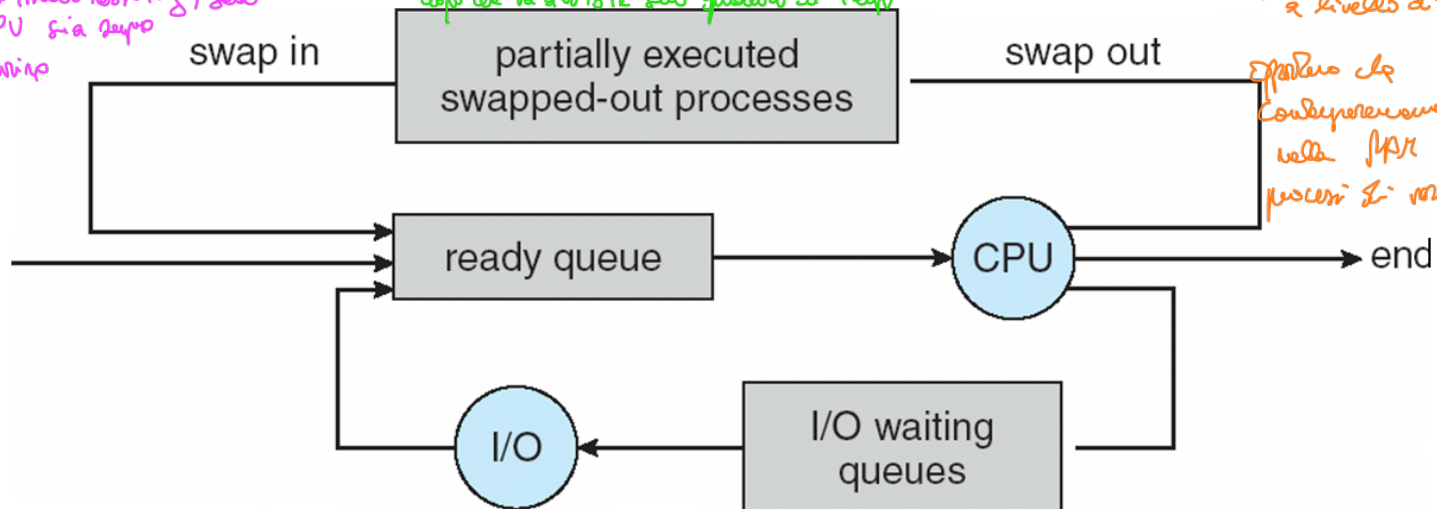
in azione
solo quando
memoria è piena

Se aumento grado di multitasking, devo
sicuro che la CPU sia sempre
occupata, ma posso esaurire
risorse

memoria piena, un processo può essere messo nella memoria di
scambio e viene ripreso quando c'è il suo turno
dopo che ha avuto il suo quanto di tempo

questo ora avviene
a livello di pagine.

spesso che
contemporaneamente
nella RAM ci siano
processi di vario genere





Tipi di scheduler – 2

- ❖ Lo scheduler a breve termine viene chiamato molto spesso (intervalli di tempo misurabili in millisecondi)
⇒ [deve essere veloce]
- ❖ Lo scheduler a medio termine controlla il **grado di multiprogrammazione** e la "miscela" di processi presenti in RAM ad un certo istante
- ❖ I processi possono essere classificati in:
 - **Processi I/O-bound**: impiegano più tempo effettuando I/O rispetto al tempo usato per l'elaborazione (in generale, si hanno molti **burst** di CPU di breve durata)
 - **Processi CPU-bound**: impiegano più tempo effettuando elaborazioni (in generale, si hanno pochi burst di CPU di lunga durata)
- ❖ Lo scheduler a medio termine deve mantenere in memoria un mix di processi "equilibrato"

processi fortemente interattivi

uso molto la CPU. Faio tanti CPU burst per poco tempo, l'80% di sole operazioni, senza operazioni di I/O

essere in RAM un buon mix

devo avere un mix di questi due processi

Burst: istanze di sole operazioni senza I/O

Se avessi in RAM solo processi CPU bound: lunghissima ready queue e code ai dispositivi brevissime

• Solo I/O bound, ho CPU scontenta e lunghe code ai dispositivi I/O





Schedulatore medio termine deve non solo decidere quali processi sono in MP e in cui
luogo di K della sua per usare tutte le risorse

Context switch – 1

↳ tempo sprecato idea: avere più processi in esecuzione

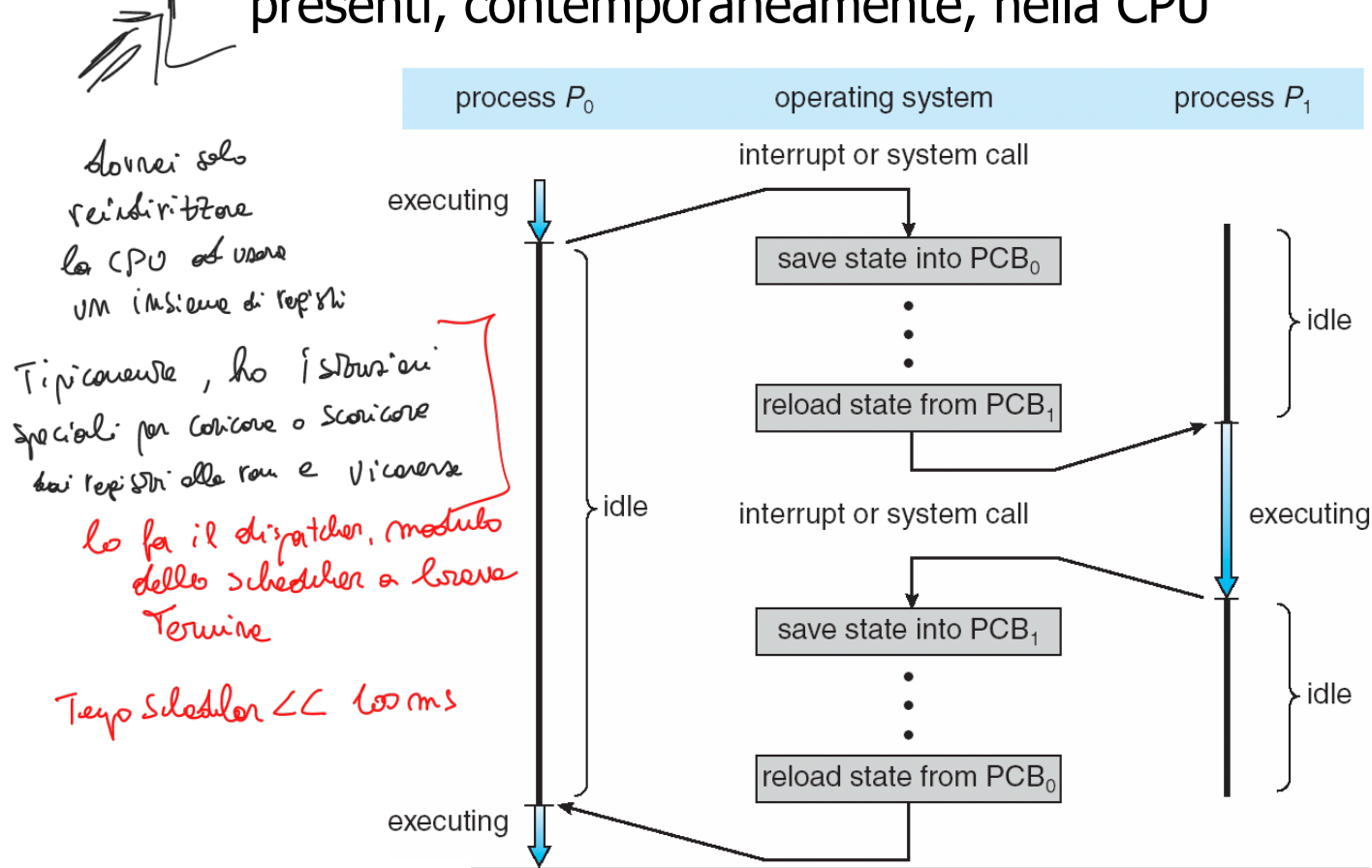
- ❖ Quando la CPU passa da un processo all'altro, il sistema deve salvare il **contesto** del vecchio processo e caricare il contesto, precedentemente salvato, per il nuovo processo in esecuzione
- ❖ Il contesto è rappresentato all'interno del PCB del processo e comprende i valori dei registri della CPU, lo stato del processo e le informazioni relative all'occupazione di memoria
- ❖ Il tempo di **context-switch** è un sovraccarico (overhead); il sistema non lavora utilmente mentre cambia contesto *CPU ferma*
 - Più sono complicati il SO e, conseguentemente, il PCB, più è lungo il tempo di context-switch





Context switch – 2

- ❖ Il tempo di context-switch (msec) dipende dal supporto hardware (velocità di accesso alla memoria, numero di registri da copiare, istruzioni speciali, gruppi di registri multipli)
 - Nel caso di registri multipli, più contesti possono essere presenti, contemporaneamente, nella CPU

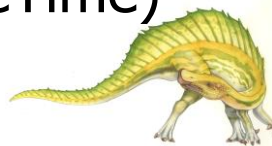


CPU Tabletop
ferma mentre si caricano e scaricano i registri



Multitasking per sistemi mobili – 1

- ❖ Alcuni SO per mobile (per es., le prime versioni di iOS) prevedevano un solo processo utente in esecuzione (sospendendo tutti gli altri)
- ❖ A partire da iOS 4 (la versione attuale è la 16.3.1):
 - Un unico processo in foreground, controllabile mediante GUI
 - Più processi in background, in memoria centrale, in esecuzione, ma impossibilitati ad utilizzare il display (quindi con “capacità” limitate)
 - Applicazioni eseguibili in background se...
 - ▶ ...realizzano un unico task di lunghezza finita (completamento di un download dalla rete)
 - ▶ ...ricevono notifiche sul verificarsi di un evento (ricezione di email)
 - ▶ ...impongono attività di lunga durata (lettore audio)
- ❖ Versioni più recenti di iOS per iPad consentono di eseguire più app in foreground contemporaneamente con la tecnica split-view; per iPhone, la modalità PiP permette la condivisione dello schermo solo per particolari app (es. FaceTime)





Multitasking per sistemi mobili – 2

- ❖ Android non pone particolari limiti alle applicazioni eseguite in background
 - ❖ Un'applicazione in elaborazione in background deve utilizzare un servizio, un ~~componente applicativo separato, che viene eseguito per conto della app~~
 - ❖ **Esempio:** app per streaming audio – se l'app va in background, il servizio continua comunque ad inviare il file audio al driver
 - ❖ Il servizio continua a funzionare anche se l'app in background viene sospesa
 - ❖ I servizi non hanno un'interfaccia utente e hanno un ingombro di memoria moderato
- ⇒ Tecnica efficace per mobile multitasking





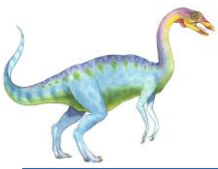
Operazioni sui processi

- ❖ Nella maggior parte dei SO, i processi si possono eseguire in concorrenza, e si devono creare e cancellare dinamicamente

Parallelismo: seguir come in esecuzione un processo

- ❖ Il SO deve fornire meccanismi per la:
 - creazione di processi
 - terminazione di processi





Creazione di processi – 1

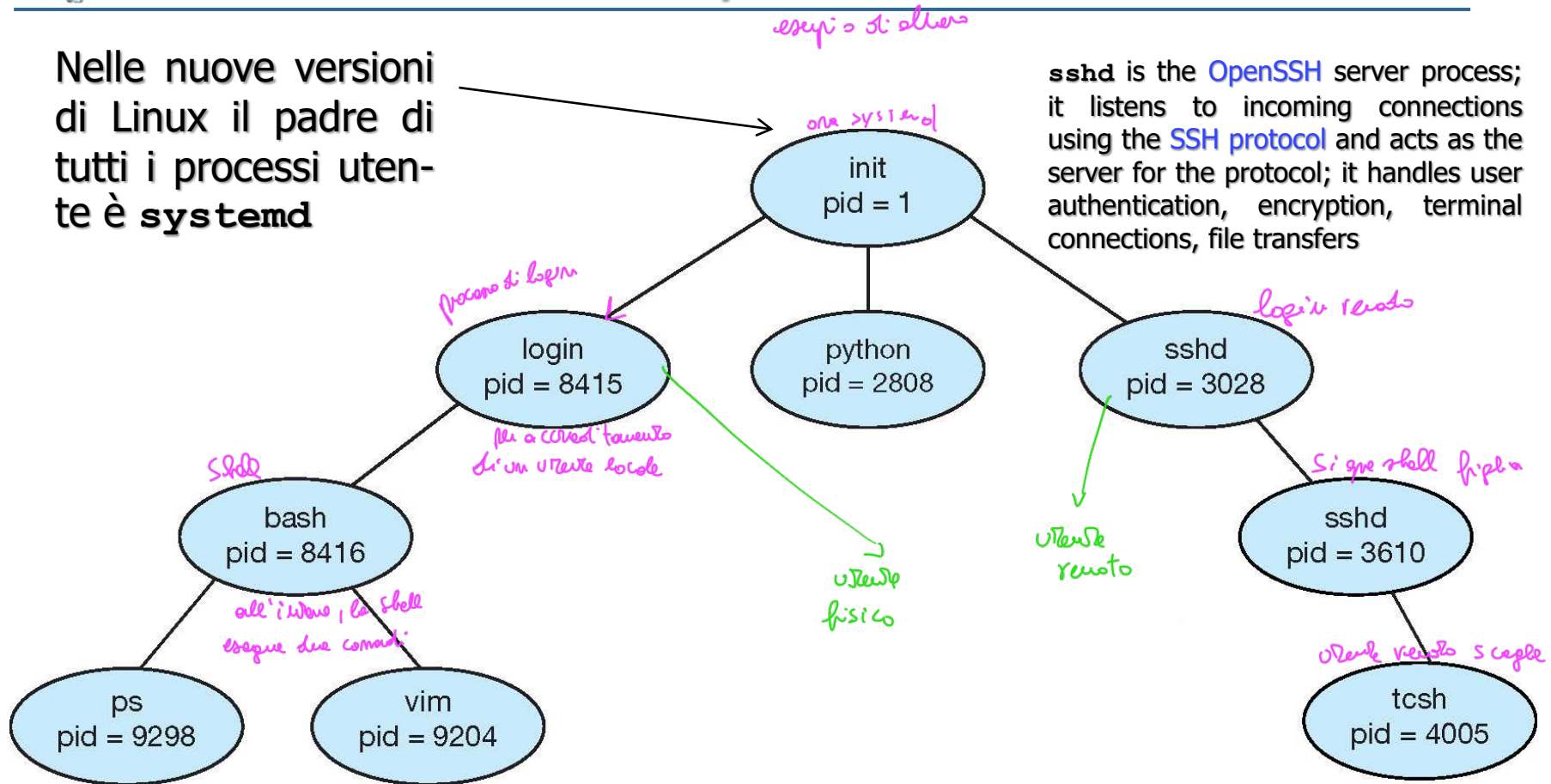
- ❖ Un processo ^(padre) può creare altri processi fino a creare alberi di processi
- ❖ Il **processo padre** crea **processi figli** che, a loro volta, creano altri processi, formando un **albero di processi**
- ❖ I processi vengono identificati all'interno del sistema tramite un **pid** (per process identifier) es. SYSTEM, Padre di tutti i processi
- ❖ **Condivisione di risorse** ↳ Quando vengono creati i processi, il padre può decidere il livello di condivisione delle risorse
 - Il padre e i figli condividono tutte le risorse
 - I figli condividono un sottoinsieme delle risorse del padre
 - Il padre e i figli non condividono risorse oppure all'inizio condividono tutte le risorse poi sono create spazi indirizzi con una exec, e così si condividono più niente
- ❖ **Esecuzione**
 - Il padre e i figli vengono eseguiti in concorrenza
 - Il padre attende la terminazione dei processi figli (con una wait())
- ❖ **Spazio degli indirizzi**
 - Il processo figlio è un duplicato del processo padre] ha spazio indirizzi identico a quello del padre
 - Nel processo figlio viene caricato un diverso programma exec





Albero dei processi in UNIX

Nelle nuove versioni di Linux il padre di tutti i processi utente è **systemd**



In Linux, con il comando **pstree** può essere ricostruito l'albero di tutti i processi del sistema

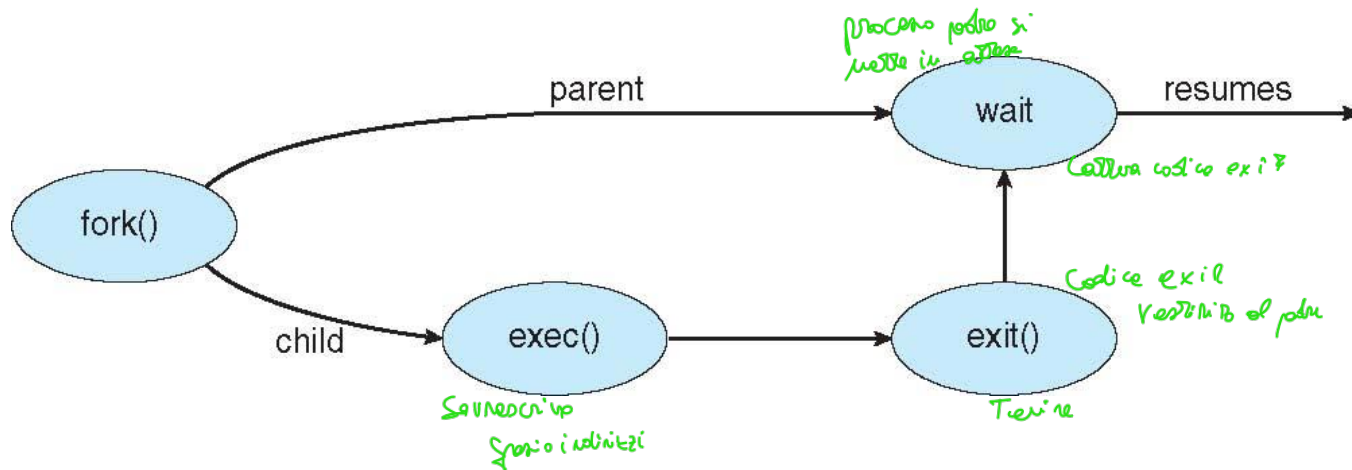




Creazione di processi – 2

❖ In UNIX/Linux...

- ...la system call **fork()** crea un nuovo processo
- quindi, la **exec()** viene impiegata dopo la **fork()** per sostituire lo spazio di memoria del processo con un nuovo programma





Creazione di processi – 3

```
#include <sys/types.h>
#include <stdio.h>
#include <unistd.h>

int main()
{
    pid_t pid;

    /* fork a child process */
    pid = fork();

    if (pid < 0) { /* error occurred */
        fprintf(stderr, "Fork Failed");
        return 1;
    }
    else if (pid == 0) { /* child process */
        execvp("/bin/ls", "ls", NULL);
        funzione di libreria, nasconde syscall exec ↳ testo del processo
    }
    else { /* parent process */
        /* parent will wait for the child to complete */
        wait(NULL);
        printf("Child Complete");
    }

    return 0;
}
```





Creazione di processi – 4

- ❖ Dopo l'esecuzione della `fork()` vi sono due processi diversi che eseguono copie dello stesso programma
- ❖ Il valore di `pid` per il processo figlio è zero, mentre per il genitore è un intero maggiore di zero (il pid del figlio)
- ❖ Il processo figlio eredita i privilegi e gli attributi di scheduling dal genitore, così come certe risorse, quali i file aperti
- ❖ Il processo figlio sovrappone quindi al proprio spazio di indirizzi il comando UNIX "`ls`" utilizzando `execvp()`
- ❖ Il genitore attende il completamento del processo figlio con la chiamata di sistema `wait()`
- ❖ Al termine del processo figlio, il processo padre riprende dalla chiamata `wait()` e quindi termina invocando `exit()`





Creazione di processi – 5

- ❖ Tutte le funzioni `exec1()`, `exec1p()`, `execle()`, `execv1()`, `execvp()` sostituiscono il processo corrente con un nuovo processo: rappresentano un'interfaccia per la system call `execve()`
- ❖ La funzione `exec1p()` accetta una lista di argomenti
 - Il primo argomento è convenzionalmente il nome del programma che verrà eseguito (e ne specifica il path-name); all'ultimo argomento valido deve seguire un puntatore a `NULL`
 - Se `exec1p()` termina correttamente non effettua ritorno al chiamante; viceversa restituisce `-1`
 - L'invocazione `wait(NULL)` fa sì che il processo padre attenda la prima terminazione di un figlio





Esercizio 1

❖ Si consideri il seguente codice:

```
#include <stdlib.h>
#include <stdio.h>
#include <unistd.h>
int main ()
{
    pid_t res; /* pid_t, in UNIX, è il tipo del PID */
    int x = 1;
    res = fork();
    if (res < 0) {
        printf("La fork è fallita.\n");
        exit (-1); }
    x++; /* istruzione 1 */ eseguita da padre e figlio
    x = ((res)?(x-1):(x+8)); padre x=1 figlio = 10
    printf("x = %d.\n", x);
    return;
}
```

Chi, fra padre e figlio, esegue **istruzione 1**? Nell'ipotesi che, all'atto dell'esecuzione della `fork()`, lo scheduler scelga di eseguire prima il figlio e subito dopo il padre, qual è l'output del programma?



Soluzione 1

```
int x = 1;
res = fork();
if (res < 0) {
    printf("La fork è fallita.\n");
    exit (-1); }
x++; /* istruzione 1 */
x = ((res)?(x-1):(x+8));
printf("x = %d.\n", x);
return;
```

- ❖ Sia il padre che il figlio eseguono **istruzione 1**
- ❖ Se dopo l'esecuzione della **fork()**, lo scheduler esegue prima "tutto" il figlio e poi il padre, l'output del programma sarà
 x = 10.
 x = 1.



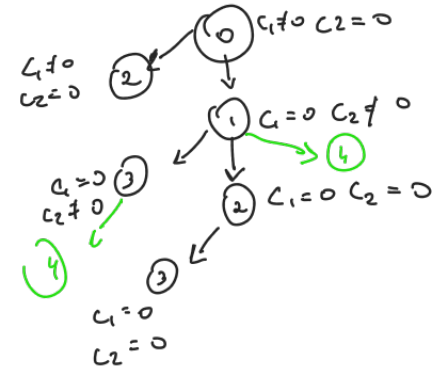


Esercizio 2

- ❖ Si consideri il frammento di codice seguente e si determini quanti processi vengono creati:

```

c2 = 0;
c1 = fork() ;           /* fork 1 */
if (c1 == 0)
    c2 = fork() ;       /* fork 2 */
fork() ;                 /* fork 3 */
if (c2 > 0)
    fork() ;             /* fork 4 */
    
```



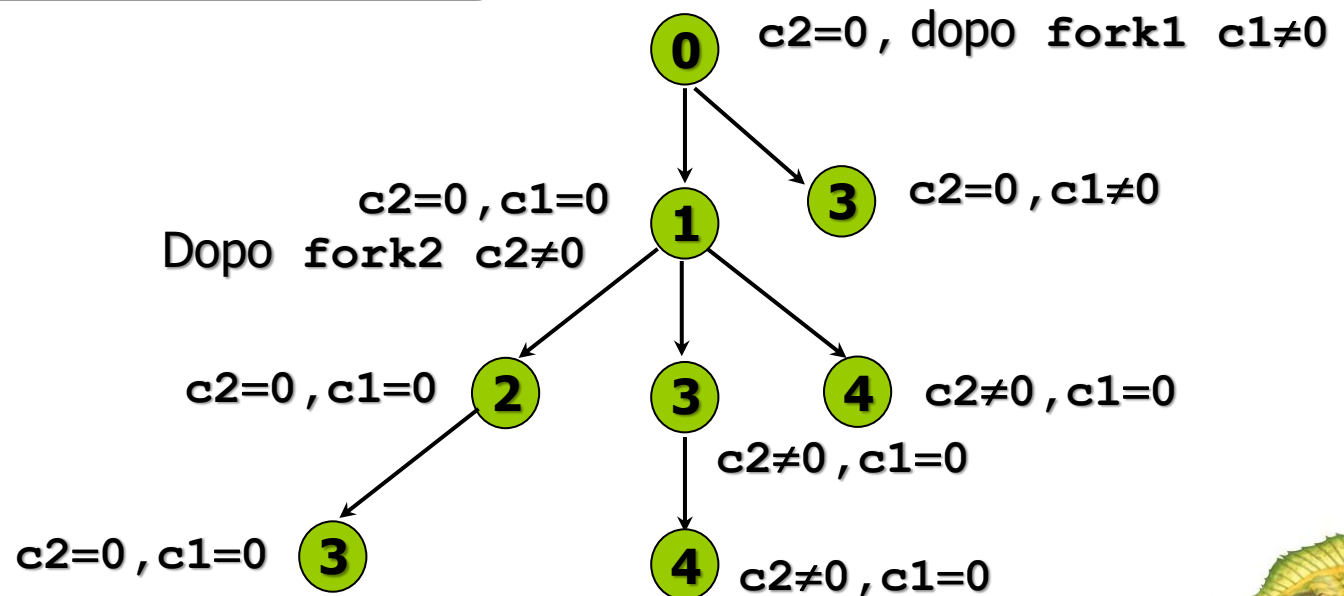
Si supponga che non si verifichino errori. Disegnare un albero che mostra come sono correlati i processi. Nell'albero, ogni processo dovrà essere descritto da un cerchio contenente un numero che rappresenta la fork che ha creato il processo. Il nodo relativo al processo originale sarà etichettato con 0, mentre il nodo del processo creato dalla prima fork conterrà 1. Gli archi indicheranno la relazione di parentela genitore→figlio.

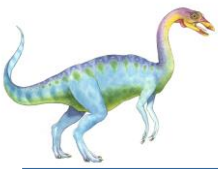




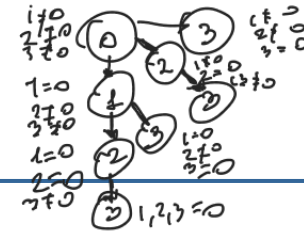
Soluzione 2

```
c2 = 0;  
c1 = fork();      /* fork 1 */  
if (c1 == 0)  
    c2 = fork();  /* fork 2 */  
fork();           /* fork 3 */  
if (c2 > 0)  
    fork();       /* fork 4 */
```





Esercizio 3

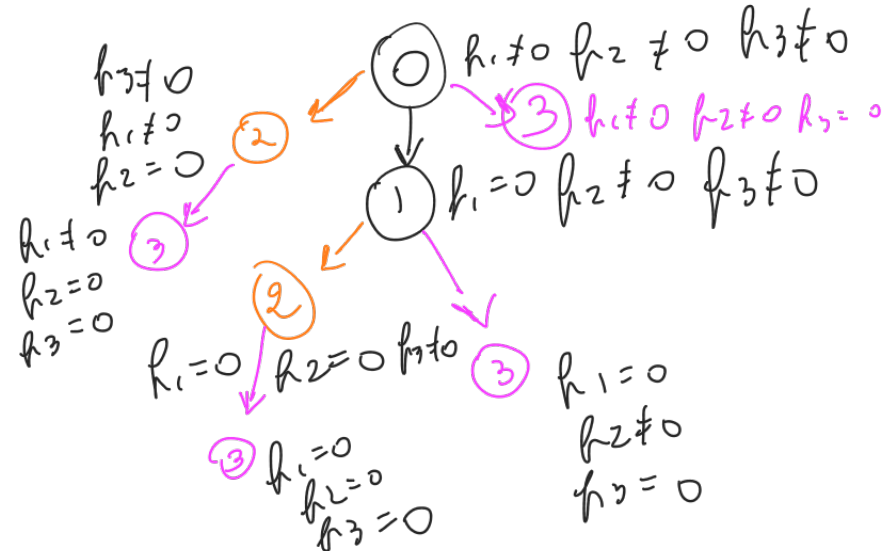


- ❖ Si consideri il seguente programma C. Quante e quali stampe si ottengono dalla sua esecuzione?

```
#include <stdio.h>
main ( )
```

```
{
    pid_t f1, f2, f3;
    f1 = fork();
    f2 = fork();
    f3 = fork();
    printf ("%i %i %i ", (f1>0), (f2>0), (f3>0));
}
```

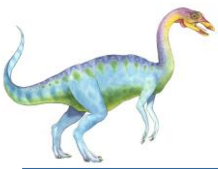
2³ processi



1 0 1
0 0 0
0 1 0
0 0 1
1 0 0
1 1 0
1 1 1
0 1 1

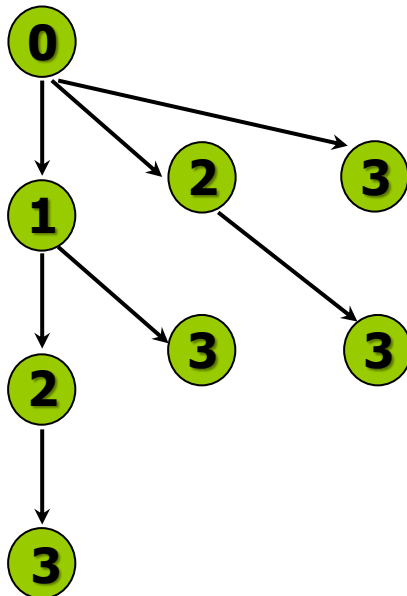
- ❖ Cosa identificano i numeri stampati?





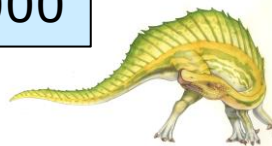
Soluzione 3

```
#include <stdio.h>
main ( )
{
    pid_t f1, f2, f3;
    f1 = fork();
    f2 = fork();
    f3 = fork();
    printf ("%i %i %i ", (f1>0), (f2>0), (f3>0));
}
```



0 →	f1≠0 f2≠0 f3≠0	111
1 →	f1=0 f2≠0 f3≠0	011
2(0) →	f1≠0 f2=0 f3≠0	101
2(1) →	f1=0 f2=0 f3≠0	001
3(0) →	f1≠0 f2≠0 f3=0	110
3(1) →	f1=0 f2≠0 f3=0	010
3(20) →	f1≠0 f2=0 f3=0	100
3(21) →	f1=0 f2=0 f3=0	000

muoi da 0 a 7





Esercizio 4

- ❖ Si consideri il file `test.c`:

```
#include <stdio.h>
int num = 0;
int main(int argc, char **argv) {
    int n, pid;
    n = atoi(*++argv);
    pid = fork();
    printf("%d", num);
    if(pid == 0 && n) {
        num = 3;
    } else if(pid > 0) {
        num = 8;
    }
    printf("%d", num);
}
```

Padre
0
num = 8
8

Figlio
0
num = 3
3

$n = 5$

0083
0803
0308
0038

- Entrambi i processi stampano 0 (il valore di num). Quindi il figlio stampa 3 ed il padre stampa 8 (in ordine non predicibile):
Testi possibili: 0038 0083 0308 0803
⇒ 0038 0083 0308 0803
- Entrambi i processi stampano 0 (il valore di num). Quindi il figlio stampa 0 ed il padre stampa 8 (in ordine non predicibile):
⇒ 0008 0080 0800

- ❖ Descriverne tutti i possibili output relativi alle due invocazioni:

\$ test 5

\$ test 0

motivando la risposta data.

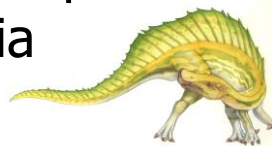
$n = 0$, non entra nell'if figlio, num = 0
Padre
0
8
Figlio
0
0
0080
0800





Architettura multiprocesso – Il browser CHROME

- ❖ Molti browser attuali (es.: Mozilla Firefox) supportano la navigazione a schede, che permette di aprire contemporaneamente, in una singola istanza del browser, più siti web
 - ❖ Se si blocca un'applicazione web in una qualsiasi scheda, si blocca l'intero processo
- ❖ **Google Chrome** ha un'architettura multiprocesso, con tre diversi tipi di processi
 - Il **browser**, che gestisce l'interfaccia utente, l'I/O da disco e da rete
 - I **renderer**, che contengono la logica per il rendering di pagine web, per la gestione di HTML, Javascript, immagini, etc.
 - ▶ Si crea un renderer per ogni sito aperto
 - ▶ Vengono eseguiti in una **sandbox**, con accesso a disco e rete limitati per minimizzare gli effetti negativi di exploit di sicurezza
 - ▶ Solo il renderer "crasha" se si verifica un problema in una scheda
 - I **plug-in**, che agevolano il browser nell'elaborazione di particolari contenuti web, come i file Flash o Windows Media





Terminazione di processi – 1

- ❖ Un processo termina quando esegue l'ultima istruzione e chiede al sistema operativo di essere cancellato per mezzo di una specifica chiamata di sistema (**exit()** in UNIX) che compie le seguenti operazioni:
 - può restituire dati (output) al processo padre (attraverso la system call **wait()**)
 - le risorse del processo vengono deallocate dal sistema operativo
- ❖ Con **abort()** il processo può terminare volontariamente segnalando una terminazione anomala al padre

• PCB, uno per ogni processo } restauro tab' anche a fronte di un exit o abort
• Tabella processi





Terminazione di processi – 2

padre può terminare i processi figli perché ne conosce l'identità

❖ Il padre può terminare l'esecuzione dei processi figli (**kill(pid, signal)**)... *kill -9*

- ...se il figlio ha ecceduto nell'uso delle risorse ad esso allocate
- ...se il compito assegnato al figlio non è più richiesto
- ...se il padre termina: *non possono restare in esecuzione i figli, ci sarebbe un effetto cascata*
 - ▶ Alcuni sistemi operativi non consentono ad un processo figlio di continuare l'esecuzione ⇒ Questo fenomeno è detto **terminazione a cascata** e viene avviato dal SO
 - ▶ Nel caso di UNIX, i processi "orfani" non vengono terminati, ma diventano figli di **init()** (**systemd()**), che è il progenitore di tutti i processi utente *per evitare l'effetto cascata*





Padre può invocare la wait

Terminazione di processi – 3

- ❖ Il processo padre può attendere la terminazione di un figlio invocando la system call **wait()**
- ❖ La chiamata ritorna informazioni di stato ed il pid del figlio terminato

pid = wait(&status); *processo terminato, nel sistema ci sono ancora tracce del processo, processo zombie.*

- ❖ Nel sistema esiste una tabella dei processi
 - Un processo terminato, il cui padre non ha ancora invocato la **wait()**, è uno **zombie** *terminato ma non sparito (c'è nel PCB e entry nella Tabella Processi)*
 - Dopo la chiamata alla **wait()** il pid del processo zombie e la relativa voce nella tabella dei processi vengono rilasciati
 - Se il padre termina senza invocare la **wait()**, il processo è un **orfano** *→ Per toglierlo definitivamente, un processo deve invocare una wait*
 - In Linux, **systemd()** diventa il padre e invoca periodicamente la **wait()** in modo da raccogliere lo stato di uscita del processo, rilasciando il suo pid e la sua entry nella tabella dei processi

Tabella processi e PCB possono appartenere a sistema, da eliminare processi orfani





Terminazione di processi – 4

- ❖ Più in dettaglio... anche se il padre e il figlio sono processi concorrenti, quando un figlio termina, il padre riceve un segnale SIGCHLD che gli notifica la necessità di accettare il codice di terminazione del figlio
- ❖ Infatti, un processo che termina non scompare dal sistema fino a che il padre non ne accetta il codice di terminazione
- ❖ Uno zombie non ha aree codice, dati o pila allocate, quindi non usa molte risorse di sistema, ma continua ad avere un PCB





Gerarchia dei processi Android

penale del Android,
quale processo uccidere
prima

- ❖ A causa delle limitate risorse di calcolo, i SO per mobile possono dover terminare dei processi
- ❖ In Android è definita una gerarchia:
 - **Processo foreground:** con cui l'utente sta interagendo direttamente *Unica app*
 - **Processo visibile:** che esegue un'attività a cui il processo foreground fa riferimento *processo relativo al processo in foreground*
 - **Processo di servizio:** che esegue in background ma svolgendo un'attività evidente per l'utente *p.es. lettore audio, processo in background*
 - **Processo in background:** svolge un'attività non evidente per l'utente
 - **Processo vuoto:** *sembra non avere...* non contiene componenti attive associate ad una app *possono essere i primi ad essere tagliati fuori*
- ❖ Android interrompe i processi in base alla gerarchia, ma salva lo stato di un processo prima della sua terminazione forzata, per poterlo ripristinare quando l'utente ritorna all'applicazione

KILL ↓





IPC

Comunicazione tra processi – 1

- ❖ Un processo è **indipendente** *se non può essere influenzato* se la sua esecuzione non può influire sull'esecuzione di altri processi nel sistema o subirne gli effetti
 - ❖ I processi **cooperanti** *se influenza o può essere influenzato* possono, invece, influire sull'esecuzione di altri processi e/o esserne influenzati
 - ❖ La presenza o meno di dati condivisi determina univocamente la natura del processo *e il risultato cooperanti, altrimenti sono INDIPEND. Due o più processi possono cooperare. SSE condividono dati*
 - ❖ Vantaggi della cooperazione fra processi
 - **Condivisione di informazioni:** ambienti con accesso concorrente a risorse condivise *Posso spedire compito da svolgere su dati in più processi cooperanti*
 - **Accelerazione del calcolo:** possibilità di elaborazione parallela (in presenza di più CPU)
 - **Modularità:** funzioni distinte che accedono a dati condivisi (es. kernel modulari) *S. D. modulari per esempio classico di processi cooperanti per svolgere compito come*
 - **Convenienza:** anche per il singolo utente, possibilità di compiere più attività in parallelo *più performance*
- "parallelismo" c'è se ci sono più core*





Comunicazione tra processi – 2

❖ Due meccanismi di comunicazione fra processi (*IPC*, *Inter-Process Communication*): *Del punto di vista logico, interazione senza overhead nei dettagli fisici.*

- **Scambio di messaggi:** utile per trasmettere piccole quantità di dati, nessuna conflittualità, utilizzo di system call per la messaggeria
- **Memoria condivisa:** massima efficienza nella comunicazione *↳ il SO mette a dispos. della fysicall ma il SO non si cura più del controllo della coerenza*
 - ▶ Richiede l'intervento del kernel solo per l'allocazione della memoria *Se ci sono errori sysop non interviene con memoria condivisa*
 - ▶ Gli accessi successivi sono gestiti (/arbitrati) dai processi
- Nei multicore lo scambio di messaggi fornisce prestazioni migliori
 - ▶ La memoria condivisa soffre dei problemi di coerenza dovuti alla migrazione dei dati condivisi tra le varie cache
 - ⇒ Al crescere del numero di core, lo scambio di messaggi può garantire migliori performance

nei sistemi multicore, uso scambi di messaggi, dovrai mantenere coerenza cache, difficile, incoerenza amplificata



Es di dati inconsistenti

P_1 1) istruzione di alto livello, in assembly

$$X = 3$$

P_1

$$1) X = X + 1$$

P_2

$$X = X - 1$$

I_1 • load r_1, X

I_2 • add $r_1, r_1, 1$

I_3 • Store r_1, X

P_2 posso usare lo stesso registro, se prima non viene l'altro

• Si esegue prima uno e poi l'altro,
non posso garantire che lo scheduler
non interrompa un processo per
farne partire l'altro

I_4 • load R_1, X ^{la R_1, X} $\leftarrow$ load $R_1, 0(R_1)$

I_5 • sub $r_1, r_1, 1$

I_6 • Store r_1, X

• I_1 $R_1 = 3$

• I_2 $R_1 = 4$

• I_4 $R_1 = 3$

• I_3 $R_1 = 3$

• I_5 $R_1 = 2$

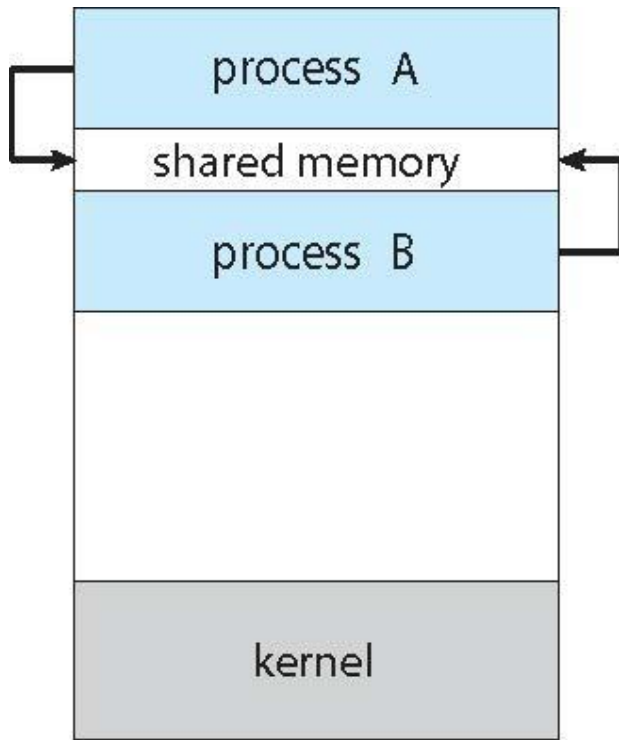
• I_6 $X = 2$

almeno che non lo risolga in ordine, il risultato
è $0 \neq 1 \neq 2$, non corretto

almeno che non garantisca che le istruzioni si
hanno livello nessuno solve

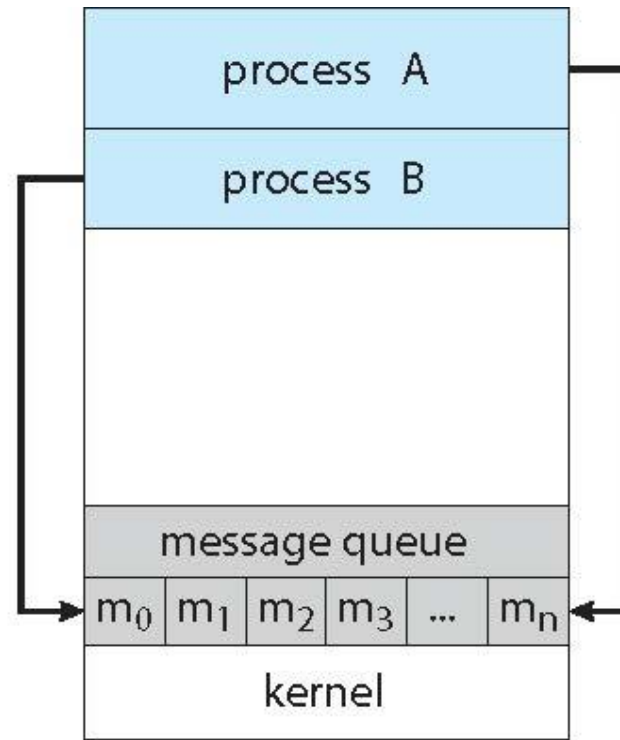


Modelli di comunicazione – 1



(a)

(a) Memoria condivisa



(b)

Kernel mantiene azioni dei processi a' loro
(b) Scambio di messaggi

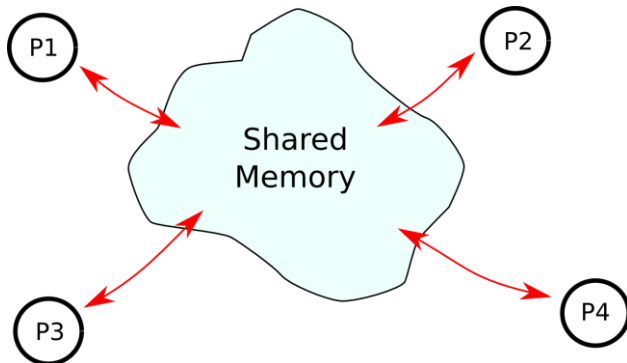




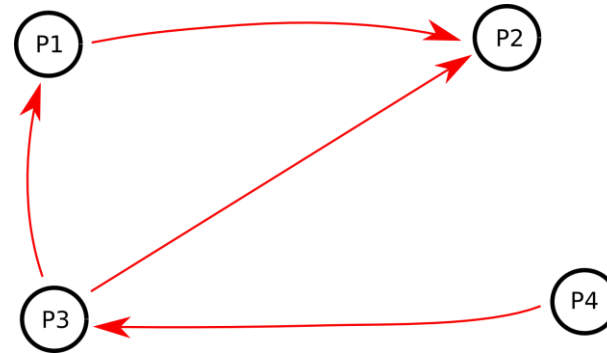
Modelli di comunicazione – 2

come Suenli's

Processi come nodi rete, com. us. mediata



Shared memory: i processi o i thread condividono dati in memoria e accedono in lettura e scrittura a tali dati condivisi



Message passing: i processi o i thread si scambiano informazioni tramite messaggi (simile a quanto avviene sulla rete)

- ❖ Lo scambio di messaggi costituisce il modello di riferimento per la comunicazione tra processi *STANDARD*
- ❖ La memoria condivisa è, invece, il modello di riferimento per la comunicazione tra thread; i thread, infatti, condividono tutte le risorse del processo cui appartengono, tra cui anche la memoria

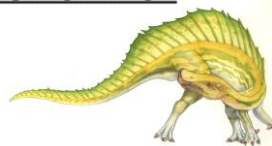
Thread fanno parte dello stesso processo, ne condividono le risorse





Sistemi a memoria condivisa

- ❖ **Memoria condivisa:** inizialmente, parte dello spazio di indirizzi del processo che la alloca *io SO in quanto ho le mani a disposizione direttive POSIX, ma se ne lava le mani*
- ❖ I processi cooperanti – che la usano per comunicare – dovranno anettere la zona di memoria al loro spazio di indirizzi *servirà ma serve di locazione mentre modifica i dati che mette un processo e lo rilascia dopo la modifica dei dati*
- ❖ La gestione della memoria condivisa, una volta allocata, non dipende dal SO
 - Il tipo e la collocazione dei dati sono determinati dai processi *MONITOR: offerta, stato e metodi che operano su questi dati,*
 - ...che hanno anche la *per regolare la consistenza dei processi comunicanti all'interno del* responsabilità di non scrivere simultaneamente nella stessa locazione
- ⇒ Fornire opportuni meccanismi per la sincronizzazione dei processi nell'accesso a dati condivisi
- ⇒ I processi cooperanti devono sincronizzare le loro azioni per garantire la coerenza dei dati





Problema del produttore–consumatore

- ❖ È un paradigma classico per processi cooperanti: il processo produttore produce informazioni che vengono consumate da un processo consumatore
- ❖ L'informazione viene passata dal produttore al consumatore attraverso un buffer
- ❖ Utile metafora del meccanismo client–server
 - **Buffer illimitato:** non vengono posti limiti pratici alla dimensione del buffer *Memoria "illimitata" produttore può sempre produrre*
 - ▶ Il consumatore può trovarsi ad attendere nuovi oggetti, ma il produttore può sempre produrne
 - **Buffer limitato:** si assume che la dimensione del buffer sia fissata
 - ▶ Il consumatore attende se il buffer è vuoto, il produttore se è pieno *Consumatore attende se buffer vuoto
Produttore attende se buffer pieno*





Buffer limitato e memoria condivisa – 1

❖ Dati condivisi

```
#define BUFFER_SIZE 10
```

```
typedef struct {
```

```
... ..
```

```
} elemento;
```

```
elemento buffer[BUFFER_SIZE];
```

```
int in = 0;
```

```
int out = 0;
```

*ogni entry è di tipo elemento
un array di*

in indica la successiva posizione libera nel buffer

out indica la prima posizione piena nel buffer

serve al produttore, se deve mettere il prossimo elemento

serve al consumatore, per l'elemento da consumare

❖ La soluzione ottenuta è corretta, ma consente l'utilizzo di soli **BUFFER_SIZE-1** elementi

*Si spreca sempre una
posizione di memoria*





Buffer limitato e memoria condivisa – 2

```

elemento next_produced;
while (true) {
    /* produce un elemento in next_produced */
    while (((in + 1) % BUFFER_SIZE) == out)
        ; /* non fa niente */
    buffer[in] = next_produced;
    in = (in + 1) % BUFFER_SIZE;
}
    
```

```

elemento next_consumed;
while (true) {
    while (in == out)
        ; /* non fa niente */
    next_consumed = buffer[out];
    out = (out + 1) % BUFFER_SIZE;
    /* consuma l'elemento in next_consumed */
}
    
```

Le due istruzioni su variabili
condivise devono essere protette (es. in e out)

forniscono l'accesso fisico
che non ci sia concorrenza

Controllo su $in+1$

**Processo
produttore**

all'inizio, $in = out$, per dire
che buffer è vuoto, ma
spreca una posizione

ARITMETICA MODULARE: resto
divisione $\% N$, tutto buffer
in maniera circolare
sia per produzione che per
consumazione

**Processo
consumatore**

serve un contatore
se non vogliamo spreco
memoria





Scambio di messaggi – 1

- ❖ **Scambio di messaggi:** meccanismo per la comunicazione e la sincronizzazione fra processi, particolarmente utile in ambiente distribuito
- ❖ Sistema di messaggi — i processi comunicano fra loro senza far uso di variabili condivise
- ❖ La funzionalità IPC consente due operazioni:
 - **send(messaggio)** — la dimensione del messaggio può essere fissa o variabile
 - **receive(messaggio)**
- ❖ Se i processi P e Q vogliono comunicare, devono:
 - stabilire fra loro un canale di comunicazione *non c'è interazione fisica*
 - scambiare messaggi per mezzo di send/receive *usando questo canale*





Scambio di messaggi – 2

❖ Problemi di implementazione

- Come vengono stabiliti i canali (connessioni)?
- È possibile associare un canale a più di due processi?
- Quanti canali possono essere stabiliti fra ciascuna coppia di processi comunicanti?
- Qual è la capacità di un canale? *è un posto di parcheggio o contiene un buffer in cui vengono memorizzati i messaggi?*
- Il formato del messaggio che un canale può gestire è fisso o variabile? *non variabile? buffer fissa: più o meno applico la favorisco se è variabile e viceversa se variabile*
- Sono preferibili canali monodirezionali o bidirezionali? *half o full duplex?*

❖ Implementazione del canale di comunicazione:

- fisica – memoria condivisa, bus hardware, rete
- logica – comunicazione diretta o indiretta, sincrona o asincrona, con bufferizzazione assente o automatica

tra coppie di processi che si conoscono o via mailbox

i processi devono attendere o possono mandare dati e continuare con i processi? Buffer presente o no?





Comunicazione diretta – 1

❖ I processi devono “nominare” esplicitamente i loro interlocutori (modalità simmetrica):

Solo Teorico

- **send(P, messaggio)** – invia un messaggio al processo P
- **receive(Q, messaggio)** – riceve un messaggio dal processo Q

destinatario deve conoscere l'identità mittente

DIRETTA, e' difficile di implementare, PID non e' sempre lo stesso

❖ **Modalità asimmetrica** *chi manda messaggio conosce destinatario, ma il ricevente sa il mittente solo alla ricezione*

COMUNICAZIONE

Paдре e Figlio, padre acquisisce PID

- **send(P, messaggio)** – invia un messaggio al processo P
- **receive(id, messaggio)** – in id si riporta il nome del processo con cui è avvenuta la comunicazione

NOME PROCESSO viene consegnato sulla base di memoria intera, può cambiare

❖ **Proprietà del canale di comunicazione:**

- I canali vengono stabiliti automaticamente
- Ciascun canale è associato esattamente ad una coppia di processi
- Tra ogni coppia di processi comunicanti esiste esattamente un canale
- Il canale può essere unidirezionale (ogni processo collegato al canale può soltanto trasmettere/ricevere), ma è normalmente bidirezionale

es. da padre → Figlio

dopo primo messaggio figlio conosce identità padre





Comunicazione diretta – 2

- ❖ **Vantaggi:** la nominazione (simmetrica/asimmetrica) è molto semplice e permette la comunicazione diretta tra coppie di processi *Nono istituto e seguito per aprire un canale*
- ❖ **Svantaggi:** È necessario un "accordo" sui nomi dei processi *difficile, al qui esecuzione PID cambia per un po' poco*
 - Il PID viene assegnato dinamicamente dal sistema e non possiamo prevederne il valore a priori
 - Nella pratica è difficile da implementare a meno che i processi non siano in relazione di parentela padre-figlio
 - In tal caso, infatti, il padre conosce il PID dei vari figli e può comunicare direttamente con loro utilizzando tale identificativo





Comunicazione indiretta – 1

- ❖ Per ovviare ai difetti della nominazione diretta si usa la nominazione indiretta basata su **porte** o **mailbox**
- ❖ I messaggi vengono inviati/ricevuti a/dal mailbox, o porte
- ❖ Proprietà dei canali di comunicazione:
 - Un canale viene stabilito solo se i processi hanno una mailbox in comune
 - Un canale può essere associato a più processi
 - Ogni coppia di processi può condividere più canali di comunicazione
 - I canali possono essere unidirezionali o bidirezionali

FILE MAPPA
IN MEMORIA
Cui posso leggere o scrivere
(Processi)

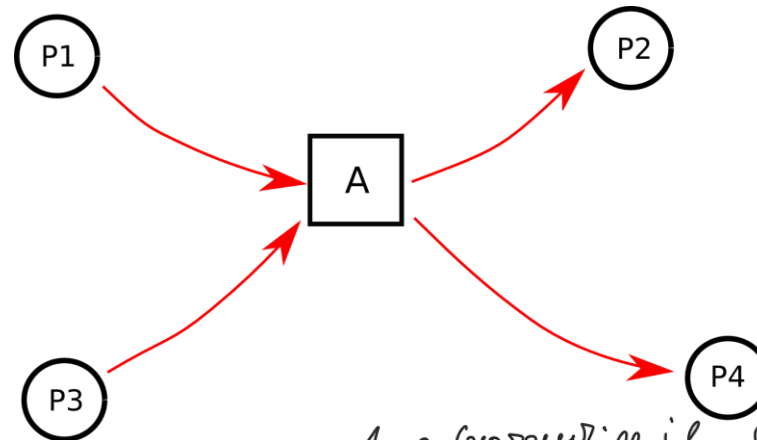
Mailbox: file in cui alcuni processi scrivono e altri scrivono. Avere un nome
• solo sapere dove andare a leggere / scrivere





Comunicazione indiretta – 2

Servizi del SO per



Cosa deve conoscere il S.O.?

❖ Operazioni:

- Creare una nuova mailbox
- Inviare/ricevere messaggi attraverso mailbox `send()` e `receive()`
- Rimuovere una mailbox se non serve più

❖ Primitive di comunicazione:

- `send(A, messaggio)` – invia un messaggio alla mailbox A
- `receive(A, messaggio)` – riceve un messaggio dalla mailbox A

Processi non si devono conoscere, devono solo condividere una mailbox





Comunicazione indiretta – 3

❖ Condivisione di mailbox

- P_1 , P_2 e P_3 condividono la mailbox A
- P_1 invia; P_2 e P_3 ricevono
- Chi si assicura il messaggio? *non lo devono ricevere subito*

❖ Soluzioni:

3 Sol.

- Permettere ad un canale di essere associato ad al più due processi *mailbox solo ha 2 processi - def mittente, lo chi riceve*
- Permettere ad un solo processo alla volta di eseguire un'operazione di ricezione *es. ricevente non riceve attraverso R.R. mezzo stabilizzato*
- Permettere al SO di selezionare arbitrariamente il ricevente o in base ad uno scheduling (es.: round robin); il sistema ne comunica l'identità al mittente

Mailbox proprietà di un processo, o del SO





Comunicazione indiretta – 4

- ❖ Una mailbox può appartenere ad un processo utente o al SO
 - Se appartiene ad un processo, cioè fa parte del suo spazio di indirizzi, solo il proprietario può ricevere messaggi, mentre gli altri processi possono solo inviarli *questo loro possono scrivere*
 - Quando il processo termina, la mailbox scompare
 - Tuttavia... il diritto di proprietà (e di ricezione) *di messaggi* si può passare ad altri processi mediante opportune chiamate al sistema *passato al SO mediante o dip. syscall, quello poi non possono tornare*
 - Una mailbox *proprietà di un altro processo / utente* creata dal SO ha vita autonoma (ma presenta i problemi di smistamento descritti) *esiste ma so pure il problema dei ricevimenti multiple*





Sincronizzazione – 1

- ❖ Lo scambio di messaggi può essere sia bloccante che non-bloccante
- ❖ In caso di scambio di messaggi bloccante, la comunicazione è **sincrona**

- Il processo mittente si blocca nell'attesa che il processo ricevente, o la porta, riceva il messaggio
- Il ricevente si blocca nell'attesa dell'arrivo di un messaggio

▶ *Rendezvous* tra mittente e ricevente

Mittente invia messaggio solo
quando viene ricevuto un ACK
dal DESTINATARIO
"OK puoi
inviare"

- ❖ In caso di scambio di messaggi non-bloccante la comunicazione è **asincrona**

- Il processo mittente invia il messaggio e riprende la propria esecuzione
- Il ricevente riceve un messaggio valido o un valore nullo





Sincronizzazione – 2

- ❖ **Send sincrona:** il messaggio viene inviato solo quando la corrispondente **receive** viene eseguita
 - La send blocca il processo mittente finché il messaggio non viene ricevuto
 - **Esempio:** durante una telefonata si inizia a parlare solo ~~quando all'altro capo c'è un interlocutore~~
- ❖ **Send asincrona:** il messaggio viene inviato indipendentemente dalla **receive**
 - Per realizzare una send asincrona è necessaria un buffer in cui depositare temporaneamente il messaggio
 - **Esempio:** la posta, che viene depositata nella cassetta o, analogamente, le email che vengono inviate indipendentemente dalla presenza online del ricevente





Sincronizzazione – 3

- ❖ **Receive sincrona:** il messaggio viene ricevuto solo se presente
 - La receive blocca il ricevente finché non giunge un messaggio da leggere *sempre in ascolto di richieste*
 - **Esempio:** un server web in attesa di connessioni
- ❖ **Receive asincrona:** la receive ritorna un messaggio se presente o NULL se non ci sono messaggi da ricevere
 - **Esempio:** Un client di email che, se non ci sono nuovi messaggi prosegue con quelli presenti, altrimenti li aggiorna *C'è un daemon in ascolto, può ricevere o no una Mail*





Sincronizzazione – 4

- ❖ La soluzione (banale) al problema del produttore–consumatore, con primitive bloccanti, diventa:

*risolve concorrenza
la comunic. flisc
dell'uno o dell'altro
e si sincronizza*

```
message next_produced;  
while (true) {  
    /* produce un elemento in next_produced */  
    send(next_produced);  
}
```

```
message next_consumed;  
while (true) {  
    receive(next_consumed);  
  
    /* consuma l'elemento in next_consumed */  
}
```





Buffering

❖ La coda dei messaggi legata ad un canale può essere implementata in tre modi

Bufferizzazione
assente



1. **Capacità zero** – Il canale non può avere messaggi in attesa al suo interno; il trasmittente deve attendere che il ricevente abbia ricevuto il messaggio (rendezvous)

Bufferizzazione
automatica



2. **Capacità limitata** – Il canale può avere al più n messaggi in attesa; se il canale è pieno, il trasmittente deve attendere

nessa sgi fornito via code.

*potrebbe destare. se buffer pieno
attendendo n' di se buffer vuoto*

3. **Capacità illimitata** – Il canale può "contenere" infiniti messaggi; il trasmittente non attende mai

infinite non attende mai





Sistemi per IPC

- libreria per permettere comunic. tra processi*
- ❖ UNIX utilizza lo standard POSIX, che prevede IPC sia tramite memoria condivisa che scambio di messaggi *usando direttamente POSIX*
 - ❖ Mach implementa la comunicazione tramite scambio di messaggi *perché è un MICROKERNEL, si parlano dell'avere perché funziona comunic. e gli altri processi comunicano (interabilmente) v.a. messaggi.*
 - I messaggi si inviano/ricevono tramite porte
 - Anche le chiamate di sistema si realizzano tramite messaggi
 - Quando si crea un nuovo task si creano anche due porte speciali: **Task Self** e **Notify**
 - Il kernel possiede i permessi di ricezione per la porta Task Self per ricevere richieste dal task e usa la porta Notify per segnalare l'occorrenza di eventi
 - ❖ Windows usa la memoria condivisa come meccanismo per supportare alcune forme di message passing

*ad ogni task
si associa 2
porte*





Esempio di sistema IPC: POSIX

- ❖ La memoria condivisa è organizzata utilizzando i file mappati in memoria, che associano la regione di memoria condivisa ad un file

- Il processo prima crea il segmento di memoria condivisa...

IDENTIFICATIVO FILE
apre un file mappato in memoria
divisione
FILE rwxrwxrwx per tutti.
le "permessi" se tu interno, SECON. di MEMORIA
lo crea se non c'è
Ci accede in lettura e in scrittura

```
shm_fd = shm_open(name, O_CREAT | O_RDWR, 0666);
```

*Come file
mappato in MEM*

- Il primo parametro specifica il nome del segmento; i parametri successivi specificano che il segmento deve essere creato, se non esiste ancora e, quindi, aperto in lettura e scrittura; 0666 bit di protezione)

- Il processo definisce quindi la dimensione del segmento

```
ftruncate(shm_fd, 4096);
```

BYTE

- Infine, può scrivere al suo interno (psm puntatore alla memoria condivisa)

```
sprintf(psm, "Writing to shared memory");
```





IPC POSIX: Produttore

```
int main()
{
    const int SIZE = 4096; /* dimensione, in byte, del segmento di memoria condivisa */
    const char STAMPA COSTANTE *name = "OS"; /* nome del segmento di memoria condivisa */ identico MEM
    /* stringa scritta nella memoria condivisa */
    const char *message0= "Studiare ";
    const char *message1= "Sistemi Operativi ";
    const char *message2= "è divertente!";
    int shm_fd; descrittore del file, INT /* descrittore del file di memoria condivisa */
    void *ptr; puntatore ad un tipo /* puntatore al segmento di memoria condivisa */
    /* crea il segmento di memoria condivisa */
    shm_fd = shm_open(name, O_CREAT | O_RDWR, 0666); crea segm.
    /* configura la dimensione del segmento di memoria condivisa */
    ftruncate(shm_fd, SIZE); definisce dimensione
    /* mappa in memoria il file che contiene il segmento di memoria condivisa */
    ptr = (char *)mmap(0, SIZE, PROT_READ | PROT_WRITE, MAP_SHARED, shm_fd, 0);
    /* Scrive sul segmento di memoria condivisa */
    sprintf(ptr, "%s", message0);
    ptr += strlen(message0); si incrementa il puntatore della
    sprintf(ptr, "%s", message1); lunghezza della stringa
    ptr += strlen(message1);
    sprintf(ptr, "%s", message2);
    return 0;
}
```

mmap, mappa contenuto file in memoria, restituisce puntatore a quel segm. di memoria

↑ tutte le volte che si aggiornano la loc. di memoria tutti i processi lo vedono

Produttore: crea spazio memoria condivisa e ci scrivo sopra





IPC POSIX: Consumatore

```
int main()
{
    const int SIZE = 4096;    /* dimensione, in byte, del segmento di memoria condivisa */
    const char *name = "OS"; /* nome del segmento di memoria condivisa */
    int shm_fd;               /* descrittore del file di memoria condivisa */
    void *ptr;                /* puntatore al segmento di memoria condivisa */

    /* apre il segmento di memoria condivisa */
    shm_fd = shm_open(name, O_RDONLY, 0666); è qui in lettura. Poter anche scrivere per i permessi.

    /* mappa in memoria il file che contiene il segmento di memoria condivisa */
    ptr = (char *)mmap(0, SIZE, PROT_READ, MAP_SHARED, shm_fd, 0); è qui

    /* legge dal segmento di memoria condivisa e stampa il contenuto */ quello che legge lo
    printf("%s", (char *)ptr);

    /* rimuove il segmento di memoria condivisa */
    shm_unlink(name); Si disconnette dalla memoria condivisa

    return 0;
}
```





Esempio di sistema IPC: POSIX (cont.)

- ❖ La funzione **mmap()** crea un file mappato in memoria, che contiene il segmento di memoria condivisa, e restituisce un puntatore al file mappato in memoria che viene utilizzato per accedervi
 - Il flag **MAP_SHARED** specifica che le modifiche apportate alla memoria condivisa saranno visibili a tutti i processi che la condividono
- ❖ La funzione **shm_unlink()** rimuove il segmento di memoria condivisa





Pipe – 1

Pipe, per farli
comunicare. Tra processi

- ❖ Una pipe agisce come un canale di comunicazione fra processi
- ❖ Come implementarla?
 - La comunicazione permessa dalla pipe è unidirezionale o bidirezionale?
 - Se è ammessa la comunicazione a doppio senso, essa è di tipo half-duplex o full-duplex?
 - Deve esistere una relazione (tipo padre-figlio) tra i processi in comunicazione? *Sì, almeno per Unix*
 - Le pipe possono comunicare in rete o i processi comunicanti devono risiedere sulla stessa macchina? *Varie*

Pipe sono Filo di 2 categorie

Pipe NAMED e UN NAMED

è un filo

usato in modo molto più
estens. Windows usa anche per la comunicazione di processi
3.75 attraverso la rete

No per Unix
Sì per Windows

Unix: Solo per processi
presenti nello stesso
calcolatore, che condividono
memoria

Silberschatz, Galvin and Gagne





Pipe – 2

❖ Pipe convenzionali

- Non possono essere accedute se non dal processo che le ha create *come mailbox proprietario* *PIPE muore con il processo*

- Esistono solo per la durata del tempo di comunicazione

- Pipe realizzato
in produttore-consumatore* ■ Permettono ai processi di comunicare seguendo il paradigma del produttore-consumatore

- Sono unidirezionali: se viene richiesta la comunicazione a doppio senso devono essere utilizzate due pipe *Se com'uso.
bidirezionale,
servono 2 pipe*
- Nel gergo di Windows, sono dette **pipe anonime**





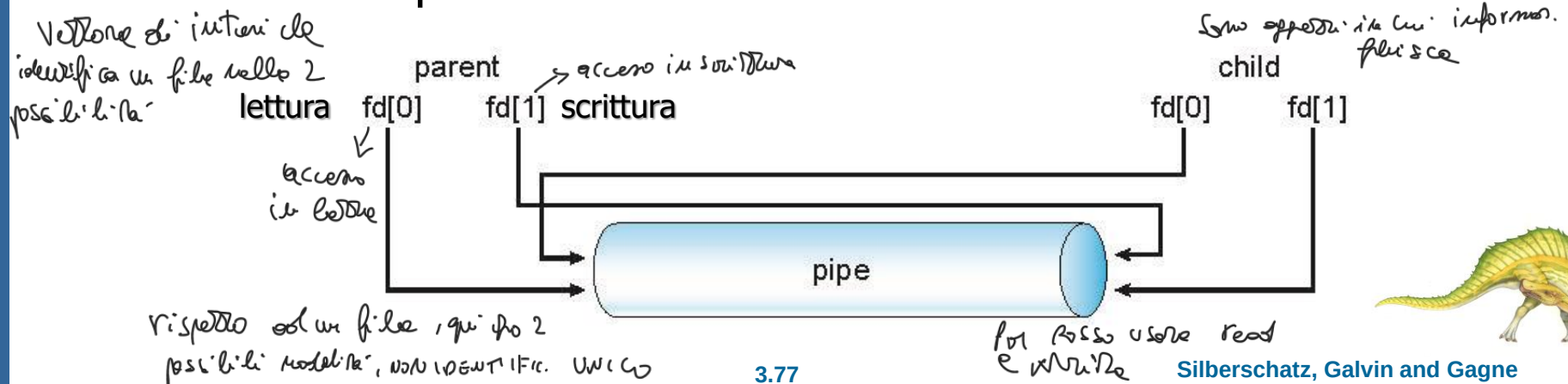
Pipe – 3

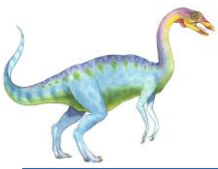
❖ Pipe convenzionali *non è di nuovo di sist diverse per SO*

- In UNIX:
 - ▶ Le pipe convenzionali sono realizzate utilizzando la funzione

pipe(int fd[])

- ▶ La pipe è un file di tipo speciale, cui si può quindi accedere tramite i descrittori **fd[]** per mezzo delle usuali chiamate di sistema **read()** e **write()**
- ▶ Solitamente è utilizzata nelle comunicazioni da padre a figlio (poiché i figli ereditano i file aperti dal padre)
⇒ i processi devono risiedere sulla stessa macchina





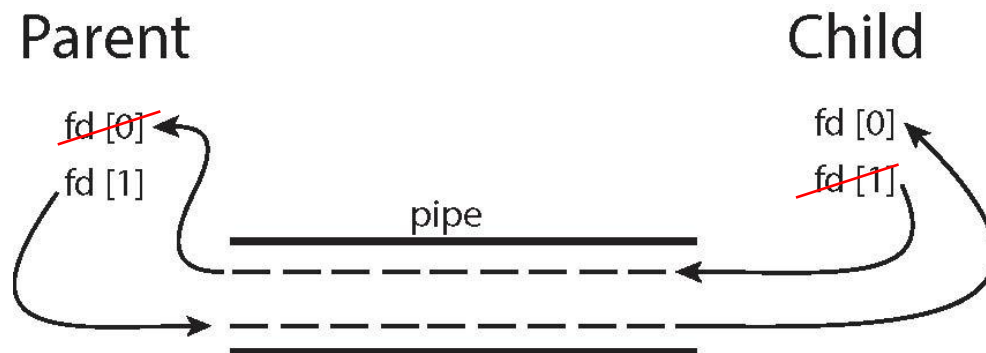
Pipe – 4

❖ Esempio: Pipe convenzionali in UNIX

- Il processo padre crea una pipe e, in seguito, invoca `fork()`
- Il padre scrive sulla pipe ed il figlio legge da essa
- Entrambi chiudono subito le estremità inutilizzate del canale di comunicazione

Padre chiude PLUSO LETTURA
FIGLIO chiude PLUSO SCRITTURA

COMUNICAZ. Padre → FIGLIO



se pipe

Fork: figlio condivide risorse padre





Pipe – 5

```
#define BUFFER_SIZE 25
#define READ_END 0 0 = vuole leggere
#define WRITE_END 1 1 = vuole scrivere

int main(void)
{
    char write_msg[BUFFER_SIZE] = "Salve";
    char read_msg[BUFFER_SIZE];
    pid_t pid; P.I.D. FIGLIO
    int fd[2];

    /** crea la pipe */
    if (pipe(fd) == -1) { = 0 se pipe creata correttamente.
        fprintf(stderr, "Errore nella creazione della pipe");
        return 1;
    }

    /** crea un nuovo processo */
    pid = fork();

    if (pid < 0) {
        fprintf(stderr, "Errore nella fork");
        return 1;
    }

    if (pid > 0) { /* processo padre */ MODALITA' read. FIGLIO QUI TIRIAMO
        /* chiude l'estremità non utilizzata della pipe */
        close(fd[READ_END]);

        /* scrive nella pipe */
        write(fd[WRITE_END], write_msg, strlen(write_msg)+1); ← carattere terminale

        /* chiude l'estremità di scrittura della pipe */
        close(fd[WRITE_END]);
    }
    else { /* processo figlio */
        /* chiude l'estremità non utilizzata della pipe */
        close(fd[WRITE_END]);

        /* legge dalla pipe */
        read(fd[READ_END], read_msg, BUFFER_SIZE);
        printf("Lettura del figlio %s\n", read_msg); STAGNA CIO' CHE HO LETTO

        /* chiude l'estremità di lettura della pipe */
        close(fd[READ_END]);
    }

    return 0;
}
```





Pipe – 6

❖ **Named pipe** *si identificano con FIFO, COMUNICAZ HALF DUPLEX*

- Possono essere utilizzate da processi qualunque (non necessariamente in relazione padre-figlio) *Processi sulla stessa macchina*
- La comunicazione può essere bidirezionale e la pipe continua ad esistere anche quando i processi comunicanti sono terminati
- Più processi possono usarla per comunicare
- Disponibili sia in Windows che in UNIX
- In UNIX: *solo formata da message oriented*

*non formata da
sequenza
di byte*

Le named pipe sono dette **FIFO** e, una volta create (con **mkfifo()**) appaiono come normali file all'interno del file system

- ▶ La comunicazione è half-duplex
- ▶ Per utilizzare le FIFO, i processi comunicanti devono risiedere sulla stessa macchina

*NON NECESS. che processi siano
padre - figlio*

*VITA NON DIPENDE dalla durata del
processo*





Pipe – 7

- ❖ Viceversa, le named pipe in Windows:
 - Offrono un meccanismo di comunicazione più ricco
 - È permessa la comunicazione full-duplex e i processi comunicanti possono risiedere in remoto
 - Diversamente da UNIX, che permette solo trasmissioni byte-oriented, Windows permette anche la trasmissione di dati formattati (message-oriented)





Le pipe in pratica – 1

- ❖ Le pipe sono usate spesso dalla riga di comando di Linux in situazioni nelle quali l'output di un comando serve da input per un secondo comando
- ❖ Dalla riga di comando, si può costruire una pipe utilizzando il carattere "|" *es. pipe in linea di comando*

- ❖ **Esempio:**

→ ricerca nelle sottodirectory
recursive
long. **\$ ls -lR | grep Jun | more** *→ posto la linea di file che contiene Jun nel record*
→ se non trova nella lista
Sceglie tutte le cose di contenere Jun

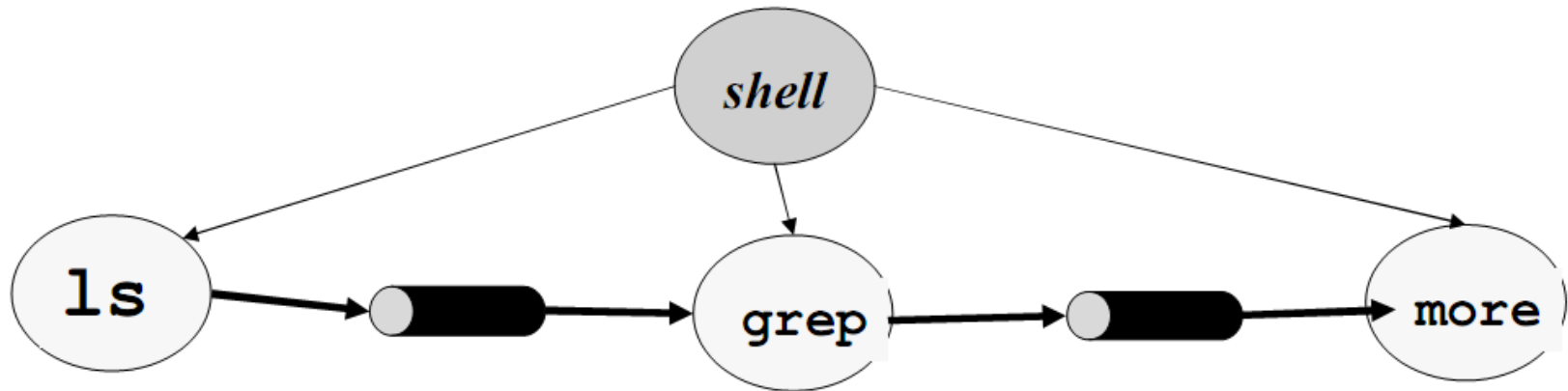
- Si ottiene la lista dei file estesa e con lettura ricorsiva del contenuto delle eventuali sottodirectory
- All'interno di questa lista, si selezionano e si stampano a video le linee contenenti la stringa **Jun**, paginate

more impiego





Le pipe in pratica – 2



- Il comando **ls** funge da produttore ed il suo output è consumato dal comando **grep**
- Il comando **grep** funge a sua volta da produttore ed il suo output è consumato dal comando **more**



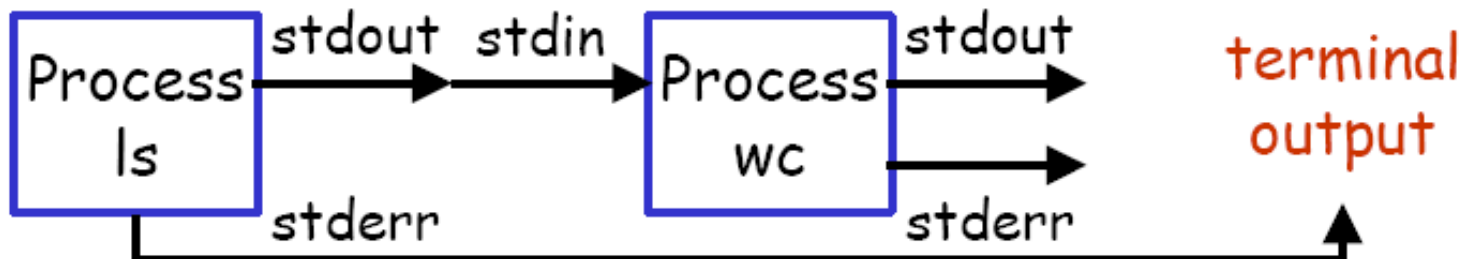


Le pipe in pratica – 3

❖ Esempio:

```
$ ls | wc -w
```

- Si ottiene la lista dei file della directory corrente
- Si “contano” le parole contenute nella lista $\Rightarrow$ numero di file contenuti nella directory





Le pipe in pratica – 4

❖ Esempi:

```
$ ls | grep rwxrwxrwx
```

*tutti i file che hanno
questi permessi o i file*

- Stampa l'elenco di tutti i file nella directory locale che hanno permesso **rwxrwxrwx** (o che hanno **rwxrwxrwx** nel loro nome)

```
$ ps -ef | grep http | wc -l
```

Processi simili

conta i

OUT count e ST, INPUT a DX word count sulle linee

- Stampa l'elenco di tutti i processi in esecuzione, quindi si filtrano quelli che contengono la stringa **http**, che vengono «contati»

→ trova questi

*-C
-w
-l line*





Comunicazioni in sistemi client-server

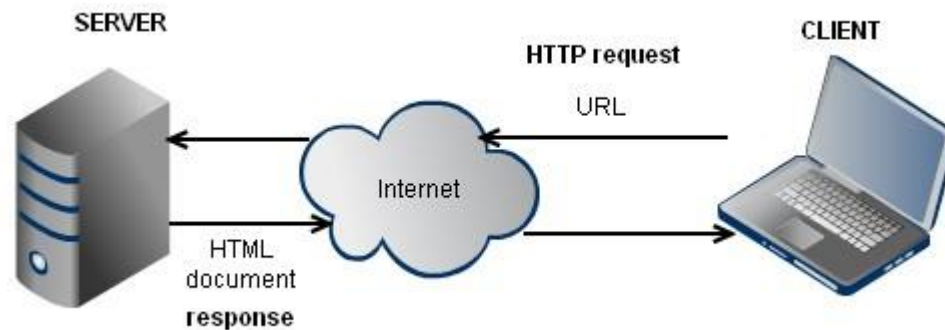
- ❖ Socket
- ❖ Chiamate a procedura remota (*Remote Procedure Call* – RPC)
 - RPC in Android
 - Invocazione di un metodo remoto (*Remote Method Invocation* – RMI, Java)





Socket – 1

- ❖ Nel modello client–server un programma chiamato client richiede servizi ad un diverso programma chiamato server
 - Quest'ultimo, che è in ascolto, ovvero in attesa di richieste da parte dei client, le esegue con le risorse che ha a disposizione, e rispedisce, se necessario, i risultati al client
 - Client e server possono risiedere (e spesso risiedono) su computer diversi, ma questa situazione è trasparente ad entrambi





Socket – 2

❖ Esistono tuttora molte applicazioni, soprattutto in Internet, che fanno uso di questo paradigma

- **telnet/ssh:** se si dispone del programma telnet/ssh (programma client), è possibile operare su un computer remoto come si opera su un computer locale (essendone utenti accreditati); sulla macchina remota deve essere presente un programma server in grado di esaudire le richieste del client telnet/ssh

posso fare
qualsiasi tipo
di operazione
da CLI

- **ftp/sftp:** tramite un client ftp (*file transfer protocol*) si possono prelevare ed inserire file su un computer remoto, purché qui sia presente un server ftp

Solo operazioni
file, o put
o get della
Macchina Remota

- **browser:** è un client web: richiede pagine ai vari computer su cui è installato un web server, che esaudirà le richieste spedendo la pagina desiderata

posso farlo anche nel
modo un utente accreditato

quando vedo una pagina, faccio un download della macchina
o un wrapper di
getter su ip

→ può essere

nomine o
anonimo

es. put nominale, get anonimo, FS disponibile a chiunque

FTP anonimo, mette a disposizione un database di file e servizi





Per comunicazione attraverso la rete usiamo **Socket – 3** il paradigma del socket

- ❖ Il meccanismo di comunicazione attraverso la rete è rappresentato dal paradigma **socket**, presentato per la prima volta nella UNIX Berkeley Software Distribution (BSD) della University of California at Berkeley
- ❖ Una socket è "una porta di comunicazione": tutto ciò che è in grado di comunicare tramite il protocollo standard TCP/IP può collegarsi ad una socket e comunicare tramite essa
canale di comunicazione, su 2, un client e server
socket : < ip, porta >
- ❖ Le socket forniscono quindi un'astrazione, che permette di far comunicare computer diversi che utilizzano lo stesso protocollo
supponendo che i pc vogliono interagire con questo protocollo

Socket, canale per l'interazione tra client
e server





Socket – 4

- ❖ Una socket è definita come l'estremità di un canale di comunicazione
- ❖ Ogni socket è identificata da un indirizzo IP concatenato ad un numero di porta
- ❖ **Esempio:** la socket 161.25.19.8:1625 si riferisce alla porta 1625 sull'host 161.25.19.8
- ❖ La comunicazione si stabilisce fra coppie di socket (una per ogni processo $\Rightarrow$ tutte le connessioni devono essere uniche)

$\text{If Address} + \text{Port Number} = \text{Socket}$



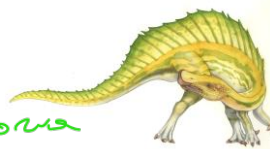


Socket – 5

- ❖ In un'architettura client-server...
 - ...il server attende la richiesta del client, stando in ascolto ad una porta specificata
 - Quando il server riceve una richiesta, se accetta la connessione proveniente dalla socket del client, si stabilisce la comunicazione
- ❖ I server che svolgono servizi specifici stanno in ascolto su porte note (per esempio, 23: telnet, 22: ssh, 21: ftp, 80: http)
*a queste porte c'è sempre un server in ascolto
cioè un daemon, un servizio*
- ❖ Tutte le porte al di sotto del valore 1024 sono considerate note e si usano per realizzare servizi standard
porta note
- ❖ L'indirizzo IP 127.0.0.1, noto come **loopback**, è peculiare ed è usato da un elaboratore per riferirsi a sé stesso
faccio riferimento la macchina a se stessa
 - ⇒ Un client e un server residenti sulla stessa macchina sono in grado di comunicare via TCP/IP

→ posso usare TCP/IP anche per client-server sulla stessa macchina

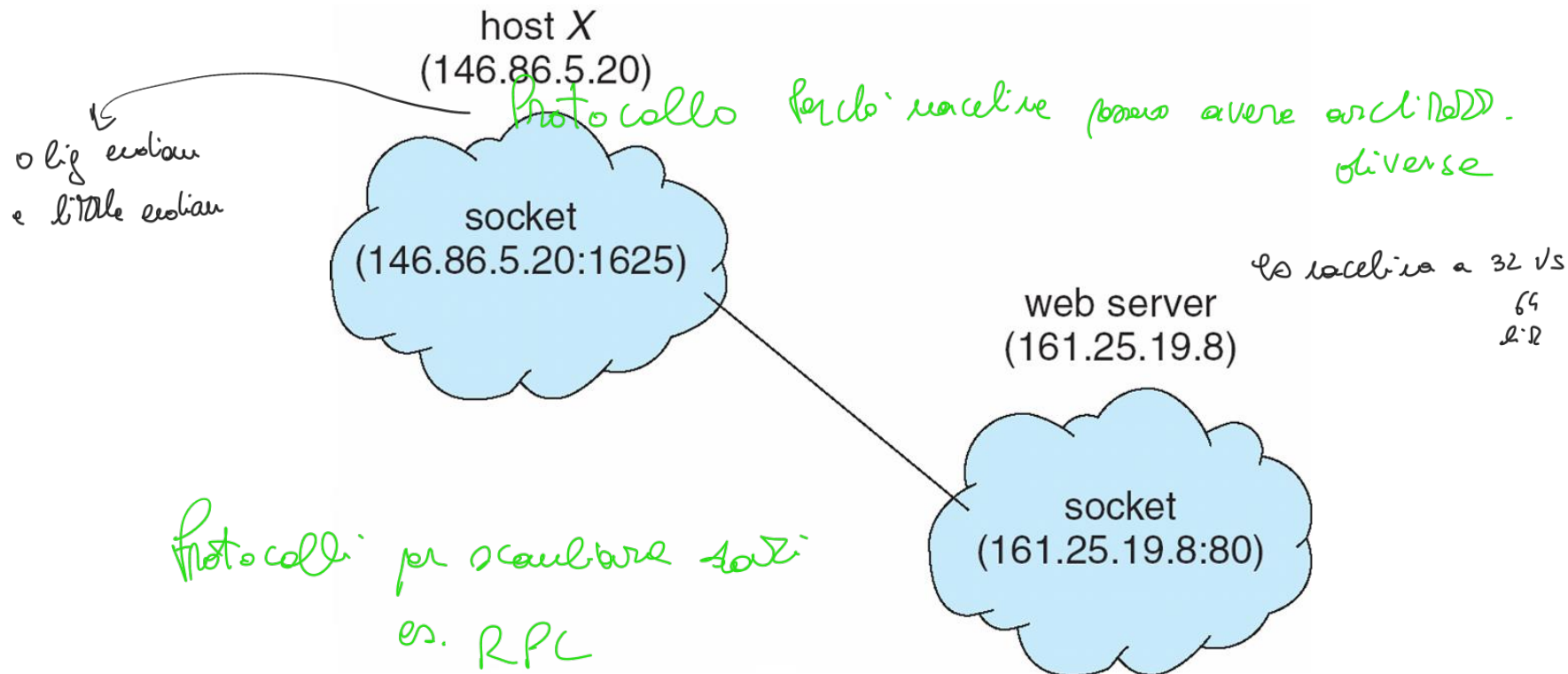
oppure: comunicano via TCP/IP invece di usare messaggio nuovo





Socket – 6

- ❖ La comunicazione tramite socket è una forma di comunicazione fra sistemi distribuiti a basso livello *cioè flusso di byte*
- ❖ Le socket permettono la trasmissione di un flusso non strutturato di byte *in questo modello, posso anche percolare i dati e non vederlo come flusso di byte*
 - È responsabilità del client e del server organizzare ed interpretare i dati





Chiamate a procedura remota – 1

Posso invocare una procedura, per sé che un processo invoca una funzione che si trova in remoto, un'azione
interprocesso comunicazione

❖ Il concetto di chiamata a procedura remota estende il paradigma della chiamata di procedura a processi residenti su sistemi remoti collegati in rete

- Gestite, tramite un meccanismo simile a IPC (ma con messaggi ben strutturati), utilizzando porte che offrono servizi diversi

STUB: Rappresentazione di codice

- ▶ La porta è un numero inserito all'inizio del messaggio

Stub sia lato client che lato server che consentono la comunicazione e strutturano i parametri (*marshalling*)

- ▶ **Stub lato client:** individua la porta del server e "impacchetta" i parametri; riceve i risultati e li "spacchetta" ad uso del processo che ha invocato la RPC

- ▶ **Stub lato server:** riceve il messaggio, spacchetta i parametri ed esegue la procedura sul server; "impacchetta" i risultati e li invia

❖ La semantica delle RPC permette ad un client di invocare una procedura presente su un sistema remoto nello stesso modo in cui esso invocherebbe una procedura locale

Remote Procedure Call simile a IPC
una macchina remota rappresentazione dati in modo diverso



invoco la remota
usando nel nome
dell'informazione e
dono fornire l'indirizzo
e porta del
servizio

impacchetta in
modo che siano
comprensibili al server

e impacchetta i parametri
alla risposta in

al client



Chiamate a procedura remota – 2

Come avviene?

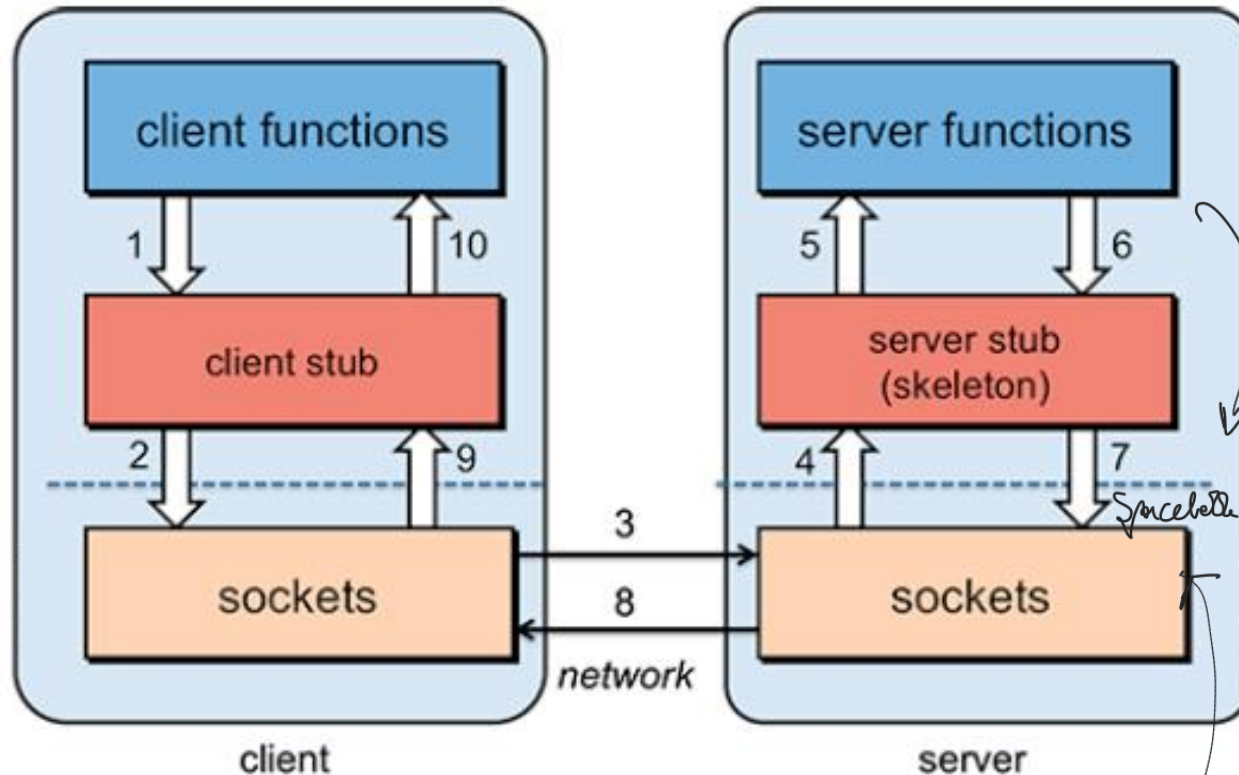
a loro livello il
meccanismo fisico via

SOCKET

per identificare
macchina e
servizio

e rinvia

↑ ↓ incapsula

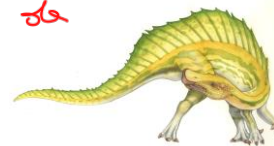


Uno **stub** è una interfaccia di comunicazione che implementa il protocollo RPC e specifica come i messaggi vengono costruiti ed interscambiati

incapsula le info relative alla macchina da

raggiungere sia provider in formato opportuno

→ fa circolare questo vis come flusso di byte





Chiamate a procedura remota – 3

❖ Il sistema delle RPC nasconde i dettagli necessari alla comunicazione (socket)

❖ Problemi

- Eventuale diversità di rappresentazione dei dati fra sistemi remoti (es.: CPU 32/64 bit, *big-endian*, *little-endian*)

⇒ Utilizzo di un formato standard di trasmissione (es.: *XDR – External Data Representation*)

- Le chiamate di procedure remote possono fallire a causa di errori dovuti alla trasmissione via rete

sono
irrelevanti
Semantica

- Semantica delle chiamate per assicurarsi che ogni RPC venga eseguita una volta soltanto

⇒ Messaggi di richiesta RPC marcati con l'ora di emissione qui non proprio ha un timestamp del servizio

⇒ Ricevuta al client

impone una chiamata e procedura viene eseguita una volta e una volta solo
• Server accetta richiesta, manda un ACK per confermare che messa in corso del servizio
3.95 Silberschatz, Galvin and Gagne





Chiamate a procedura remota – 4

server che prende in considerazione le richieste con timestamp più basso
non si ripete la posizione del servizio

- ❖ Come avviene l'associazione fra client e server?
- ❖ Come può il client conoscere il numero di porta associato ad una particolare RPC?
 - Associazione fissa e nota a priori RPC $\leftrightarrow$ porta
 - Associazione dinamica mediata da un servizio di rendez-vous
 - ▶ Il server ha un demone in ascolto che riceve una richiesta dal client, alla quale risponde con il numero di porta relativo alla particolare chiamata

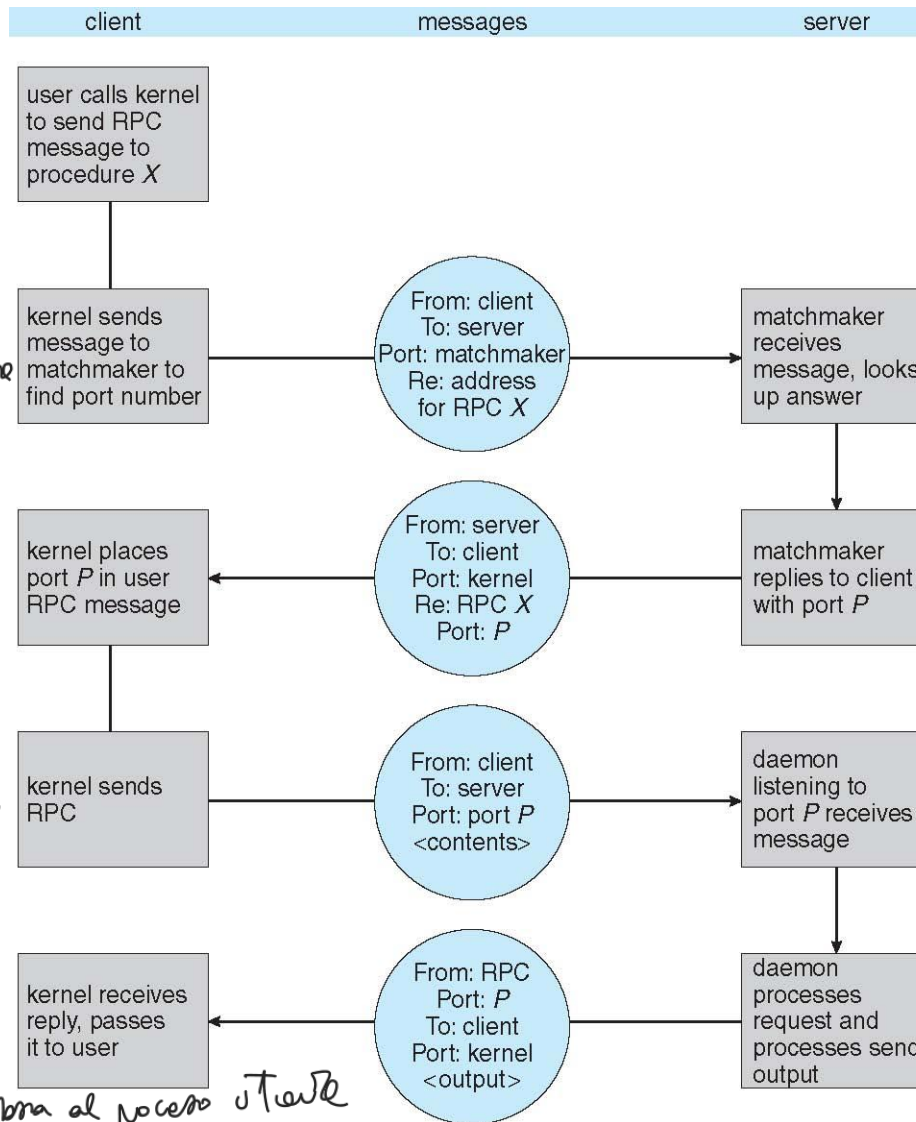
NON
CLIENT può conoscere a priori la porta del servizio del server
per servizi non noti

(
→ stesso client usa una porta nota per mandare una richiesta al server





Chiamate a procedura remota – 5



1) utente chiede al kernel di mandare RPC

messaggio al server
il server dovrebbe ritornare
porta effettiva per
effettuare servizio

kernel ha
la porta, può richiedere
servizio

prima al processo utente
dopo che ha fatto la
richiesta

procedura esegui'le





Chiamate a procedura remota – 6

→ uso da android da RPC

- ❖ Sebbene le RPC siano di solito associate all'elaborazione client-server, possono anche essere usate per la comunicazione fra processi in esecuzione sullo stesso sistema
 - Android ha un ricco insieme di meccanismi IPC (nel Binder), tra cui le RPC utilizzabili in locale

mette a disposizione servizi

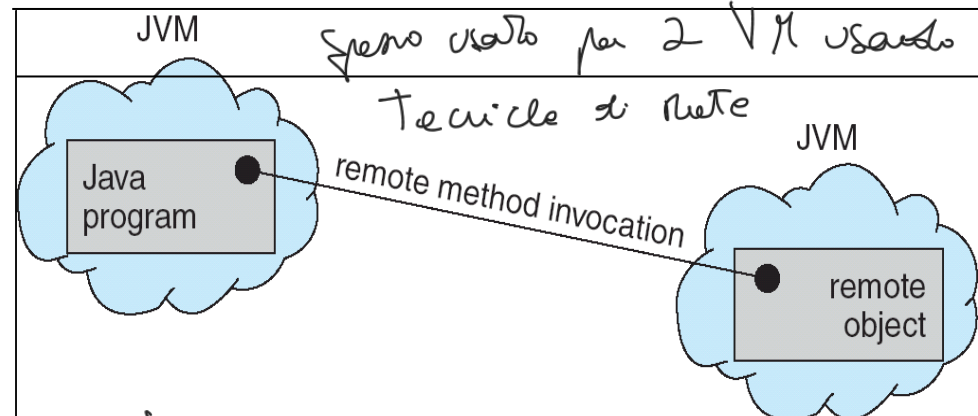
RPC Android per com. tra processi

- ❖ L'invocazione di metodi remoti è una funzione del linguaggio Java simile alla RPC

Remote Method Invocation, tecnica virtuale che invoca un metodo su un'istanza remota

- ❖ L'RMI permette ad un processo Java residente su una data JVM l'invocazione di un metodo su un oggetto remoto, dove per remoto si intende un oggetto residente su una diversa macchina virtuale

due macchine virtuali possono comunicare con una rete virtuale usando RPC



Java: non invoca procedure ma Metodi



Esercizio 1 *Per Fork*

❖ Descrivere un possibile output del seguente codice:

```
void main()
{
    int j, ret;
    j = 10;
    ret = fork();
    printf("Child created \n");
    j = j * 2;
    if(ret == 0) {
        j = j * 2;
        printf("The value of j is %d \n", j);
    }
    else {
        ret = fork();
        j = j * 3;
        printf("The value of j is %d \n", j);
    }
    printf("Done \n");
}
```





Esercizio 2

- ❖ La successione di Fibonacci (0,1,1,2,3,5,8,...) è definita ricorsivamente da

$$f_0 = 0$$

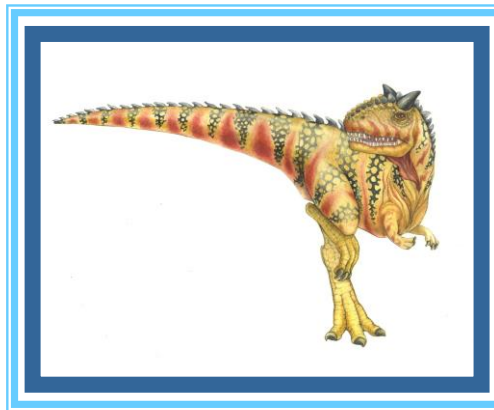
$$f_1 = 1$$

$$f_n = f_{n-1} + f_{n-2}$$

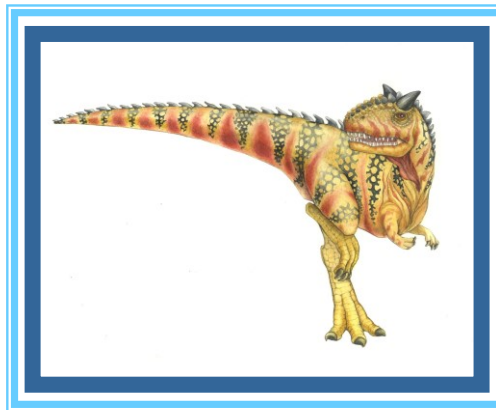
Scrivere un programma C, che usi la chiamata di sistema **fork()**, per generare la successione di Fibonacci all'interno del processo figlio. Il processo figlio produrrà anche le relative stampe. Il genitore dovrà rimanere in attesa tramite **wait()** fino alla terminazione del figlio. Il numero di termini da generare sarà specificato a riga di comando. Implementare i necessari controlli per garantire che il valore in ingresso sia un numero intero non negativo.



Fine del Capitolo 3



Thread e concorrenza





Paralleli-zei nello stesso processore

Obiettivi

- ❖ Introdurre la nozione di thread – l'unità fondamentale di utilizzo della CPU nei computer che supportano il multithreading *di base*
- ❖ Descrivere i principali vantaggi e le problematiche più significative della progettazione di processi multithread *di usare Thread e Processi*
difficoltà nell'usare Thread
- ❖ Discutere le API per le librerie di thread, con particolare riferimento a Pthreads *chiedo al compilatore di vedere un programma multithread*
- ❖ Presentare approcci al threading implicito
- ❖ Definire il supporto di Linux ai thread *usando uno syscall clone, fork con portabili parametri*





Sommario

- ❖ Thread: motivazioni e definizione
- ❖ Programmazione multicore
- ❖ Modelli di programmazione multithread
- ❖ Librerie per i thread
- ❖ Threading implicito
- ❖ Problemi nella programmazione multithread
- ❖ Thread di Linux





Motivazioni – 1

❖ La maggior parte delle applicazioni attuali sono **multi-thread**

❖ I **thread** vengono eseguiti “all’interno” delle applicazioni

❖ La gestione dei processi può diventare molto onerosa, sia dal punto di vista computazionale che per l'utilizzo di risorse

- Creazione: *di un thread, non devo allocare nuovo spazio degli indirizzi e non vanno allocati DATI e CODICE Thread* allocazione dello spazio degli indirizzi e successiva popolazione
 - Context-switch: *Tempo di overhead puro per CPU* salvataggio e ripristino degli spazi di indirizzamento (codice, dati, stack) di due processi
- per i thread molto più veloci, meno tempo sprecato*





Creazione Thread, ottimo se è veloce

Motivazioni – 2

ogni volta che arriva una richiesta per farla girare su un Thread per gestire la richiesta

- ❖ Con un'applicazione che genera molti processi (es.: server), il SO rischia di passare la maggior parte del tempo a svolgere operazioni interne di gestione, piuttosto che ad eseguire codice applicativo
- ❖ Inoltre, a volte, l'attività richiesta ha vita relativamente breve rispetto ai tempi di gestione
 - **Esempio:** invio di una pagina html da parte di un server Web $\Rightarrow$ "operazione leggera", per motivare un nuovo processo
- ❖ Infine, esistono applicazioni che, presentando un intrinseco grado di parallelismo, possono essere decomposte in attività da eseguire concorrentemente, sulla base di un insieme di dati e risorse comuni

app che si eseguono in parallelo, molti copy di se stesse

in parallelo o concorrente

non vogliono che tutti i processi di esecuzione del Thread

tutti i Thread fanno riferimento a dati comuni





1 processo, tanti thread, si usa per codice, dati o file o agenti

Motivazioni – 3

❖ **Esempio:** Applicazioni in tempo reale per il controllo di impianti fisici, in cui si individuano attività quali...

- Controllo dei singoli dispositivi di I/O dedicati a raccogliere i dati del processo fisico (sensori)
- Invio di comandi verso l'impianto

Attivazione a voce multithread

⇒ Le diverse attività devono frequentemente accedere a strutture dati comuni, che rappresentano lo stato complessivo del sistema da controllare

❖ Infine...

- La programmazione multithread può produrre codice più semplice e più efficiente non sempre
- I kernel attuali sono normalmente multithread
- Il multithreading si adatta perfettamente alla programmazione nei sistemi multicore

Supporto la possibilità di avere thread

perché ora su un core posso avere 2 thread dello stesso processo in esecuzione

in Solaris, un core thread costa 1/30 rispetto alla versione mono-processo
1/5 per il 4.6 context switch





Definizione – 1

- ❖ Un *thread* (trama, filo) è l'unità base d'uso della CPU e comprende un identificatore di thread (TID), un contatore di programma, un insieme di registri ed uno stack
 - Condivide con gli altri thread che appartengono allo stesso processo la sezione del codice, la sezione dei dati (globali) e le altre risorse allocate al processo originale (file aperti e segnali)
- ❖ Un processo tradizionale – *heavyweight process* – è composto da un solo thread *PROCESSO PESANTE*
 - Un processo è "pesante", in riferimento al contesto (spazio di indirizzamento, stato) che lo definisce

Thread
percorsi esecutivi
voglio di visuale
Dati globali

anche processi pesanti hanno sempre 2 parti
dati e risorse
Filo di esecuzione
voglio distinguere le due cose
così da avere più percorsi





Definizione – 2

- ❖ I thread sono detti anche “processi leggeri”, perché hanno un contesto ridotto (condividono lo spazio degli indirizzi)
- ❖ Poiché i thread appartenenti ad uno stesso processo ne condividono codice, dati e risorse...
 - Se un thread altera una variabile non locale al thread, la modifica è visibile a tutti gli altri thread
 - Se un thread apre un file, allora anche gli altri thread possono operare sul file
- ❖ Un processo multithread è in grado di eseguire più compiti in modo concorrente

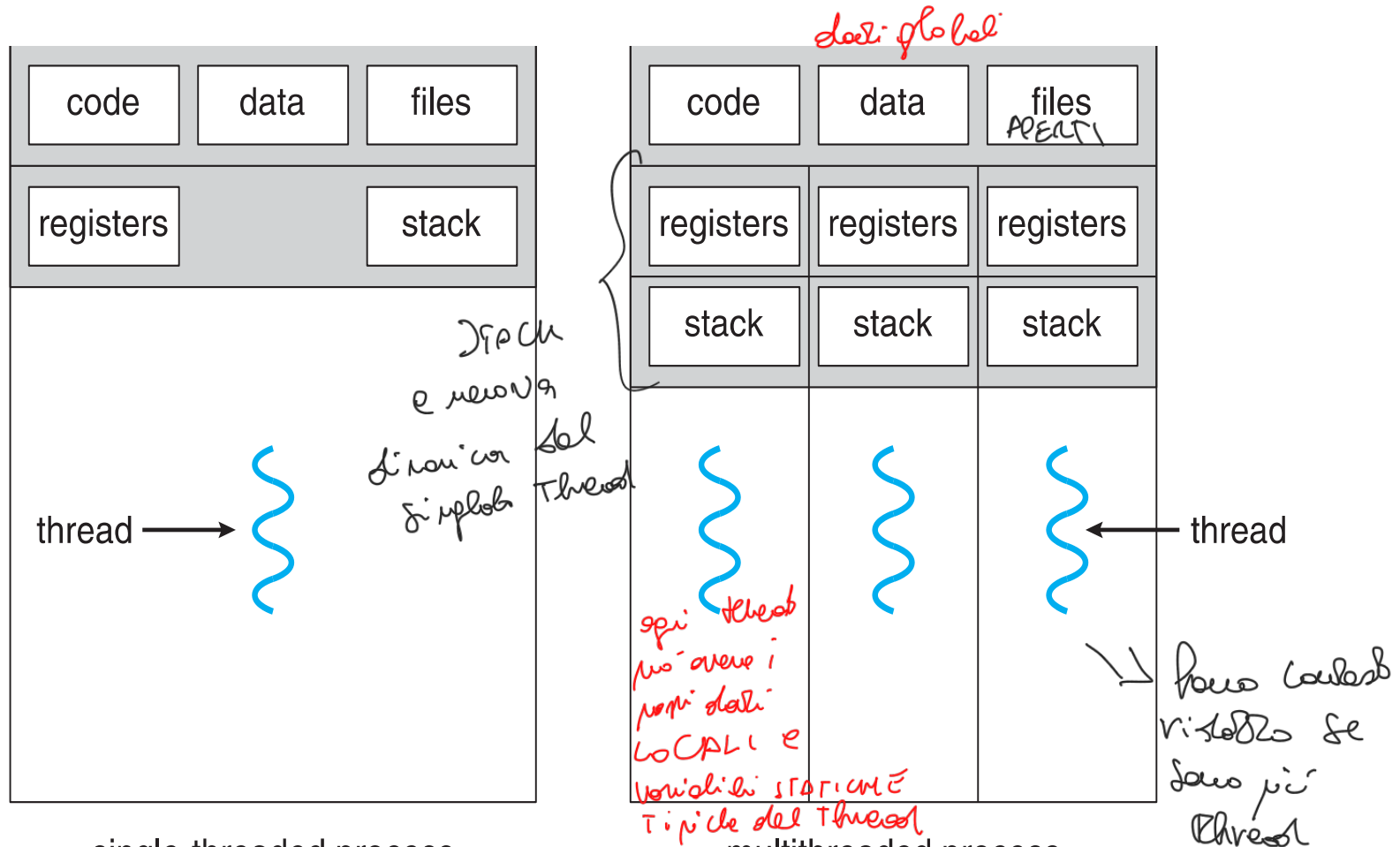
• essendo percorsi esecutivi:
cambiano i contenuti dei
registri della CPU
i THREAD hanno un loro
CONTESTO

hanno un loro TID (non PID), contesto
e PC





Processi a singolo thread e multithread



single-threaded process
CONTEXT SWITCH, qui processo ha il suo spazio degli indirizzi.

Processi ora vengono implementati. Rappresentano i processi sono sparsi nella RAM, e per mapparli, c'è una struttura dati (TA BELLA delle PAGINE)





PAGINE
1
2

FRANC
1 2 3
4 20

Esempi – 1

Ci sono dei registri che devono contenere l'indirizzo della Tabella o della pagina

Thread condividevano PCB, anche info sui file aperti ecc, non o da condividere nel context switch

non se lo più thread non cambia la Tabella
popolo. Popolo da copiare nel context
e' un sottoinsieme dei registri del processore

❖ Browser web

- Un thread per la rappresentazione sullo schermo di immagini e testo *Client web multithread*
- Un thread per il reperimento dell'informazione in rete *es un thread per gestire tutte queste cose*

❖ Text editor

- Possibile avere un thread per ciascun documento aperto (i thread sono tutti uguali ma lavorano su dati "locali" diversi)
- Relativamente ad ognuno è anche possibile avere...
 - ▶ Un thread per la visualizzazione dei dati su schermo
 - ▶ Un thread per la lettura dei dati immessi da tastiera
 - ▶ Un thread per la correzione ortografica e grammaticale
 - ▶ Un thread per il salvataggio periodico su memoria di massa

Se thread condividono file o questo file, e' verosimile condividerli con gli altri thread

Queste operaz. usate nella stessa cornice





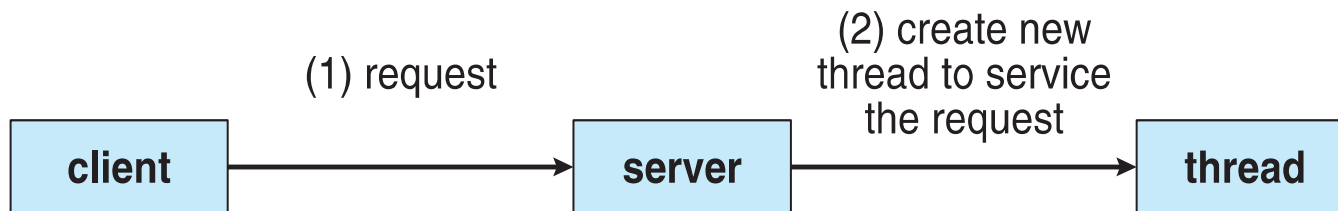
Esempi – 2

❖ Server per RPC

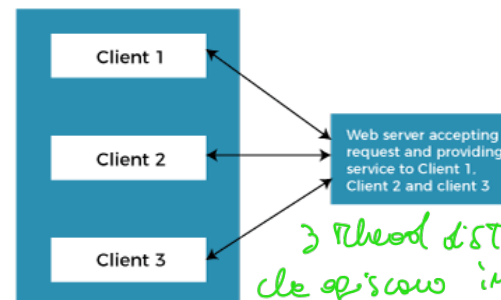
- Ogni invocazione di procedura remota viene delegata ad un thread separato (diverse richieste gestibili in concorrenza)

❖ Server web *1) thread server over in ascolto*

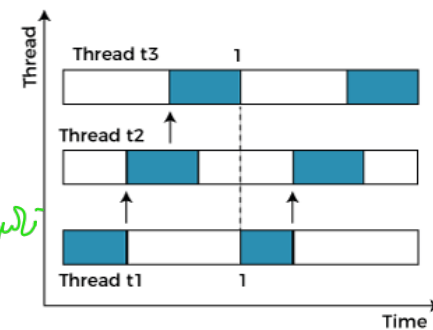
- Un thread per il servizio di ciascuna richiesta da parte dei client



thread server
(3) resume listening for additional client requests



*3 thread distinti
che operano in
CONCORRENZA*





Kernel multithread *funzionale per*

- ❖ I kernel dei sistemi operativi attuali *poterlo estendere sopra alle applicazioni* sono perlopiù multithread
 - Thread dedicati a servizi specifici
- ❖ **Solaris** *l'utente richiama interrupt e lo gestisce*
 - Thread specializzati nella gestione delle interruzioni
- ❖ **Linux**
 - Impiega thread a livello kernel per la gestione dei dispositivi, della memoria e delle interruzioni
 - Si può utilizzare il comando "**ps -ef**" per visualizzare i thread a livello kernel
 - Il thread **kthreadd** (con **pid=2**) funge da genitore di tutti i thread a livello kernel





Vantaggi – 1

Vantaggi Multithreading

❖ Tempo di risposta

*Metрика es-tempo attesa: tempo in cui processore
nella ready queue*

*quando ci mette
dal primo al
ultimo output*

- Rendere multithread un'applicazione interattiva permette ad un programma di continuare la propria esecuzione, anche se una parte di esso è bloccata o sta eseguendo un'operazione particolarmente lunga, riducendo il tempo medio di risposta
- Utile per le interfacce utente

*con multithread ripieno tempo di risposta
più cose, anche se un thread è lento
ci saranno altri più veloci della lista
problem da risolvere out*

❖ Condivisione delle risorse

- I thread condividono la memoria e le risorse del processo cui appartengono; il vantaggio della condivisione del codice e dei dati risiede nel fatto che un'applicazione può consistere di molti thread relativi ad attività diverse, tutti nello stesso spazio di indirizzi (comunicazioni molto più rapide che con memoria condivisa o message passing)

*evita scambio message (lento) e condivisione
memoria tra processi (no controllo I/O)*





Vantaggi – 2

Context switch più rapido

❖ Economia

- Assegnare memoria e risorse per la creazione di nuovi processi è oneroso; poiché i thread condividono le risorse del processo cui appartengono, è molto più conveniente creare thread e gestirne i cambiamenti di contesto (il TCB dei thread non contiene informazioni relative a codice e dati, quindi alla gestione della memoria) *no nuovo spazio di memoria*
 - ▶ **Esempio:** in Solaris la creazione di un processo richiede un tempo trenta volte maggiore della creazione di un thread; il cambio di contesto è cinque volte più lungo per i processi rispetto ai thread





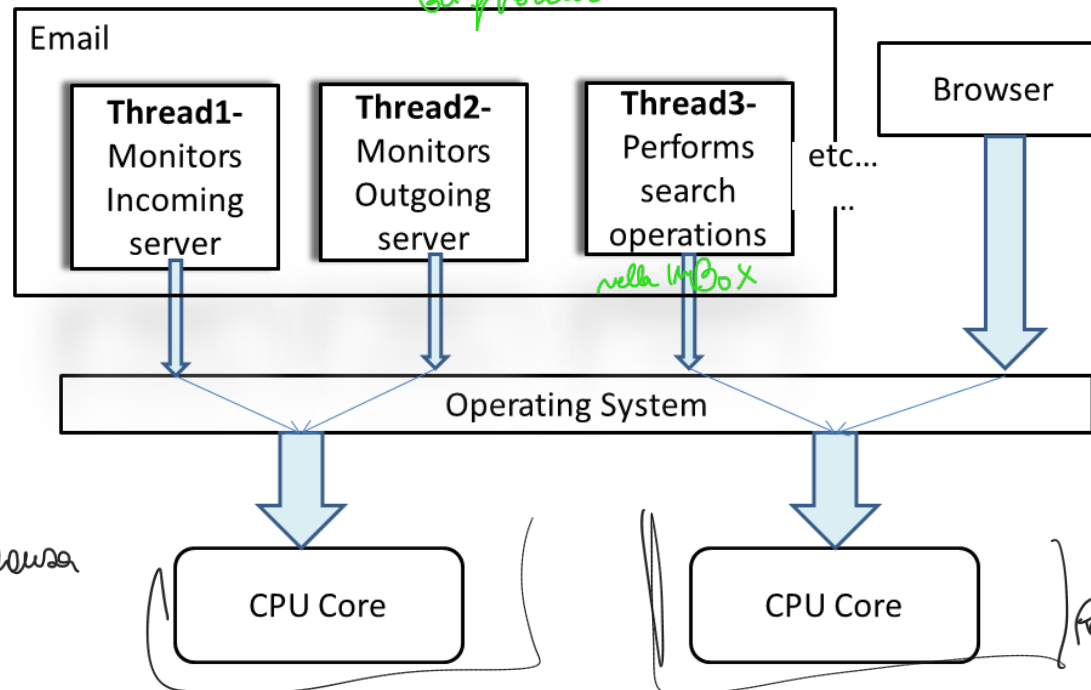
Vantaggi – 3

❖ Scalabilità

multithread più vantaggioso più thread abbiamo a disposizione

- I vantaggi della programmazione multithread aumentano notevolmente nelle architetture multiprocessore, dove i thread possono essere eseguiti in parallelo; l'impiego della programmazione multithread in un sistema con più unità di elaborazione fa aumentare il grado di parallelismo

Servizi in corso differenti



Server per la gestione della mail

nella mailbox

ho concorrenza e parallelismo

Concorrenza

Parallelismo



Programmazione multicore – 1

❖ Architetture multicore: diverse unità di calcolo sullo stesso chip *RAM e 1 o 2 livelli di cache condivisa*

- Ogni unità appare al SO come un processore separato

❖ Sui sistemi multicore i thread possono essere eseguiti in parallelo, poiché il sistema può assegnare thread diversi a ciascuna unità di calcolo

❖ Distinzione fra **parallelismo** e **concorrenza**

- Un sistema è parallelo se può eseguire simultaneamente più task *Solo se ci sono più unità di calcolo. V CPU in solo processore in esecuzione*

- Un sistema concorrente, invece, supporta più task, consentendo a tutti di progredire nell'esecuzione grazie al multiplexing della CPU (unica) *scambio di processi in elaborazione*

CPU switching tra più processori

*ogni core è un processore a se stante dal pto di vista del SO
e dunque uno scheduler V core.*

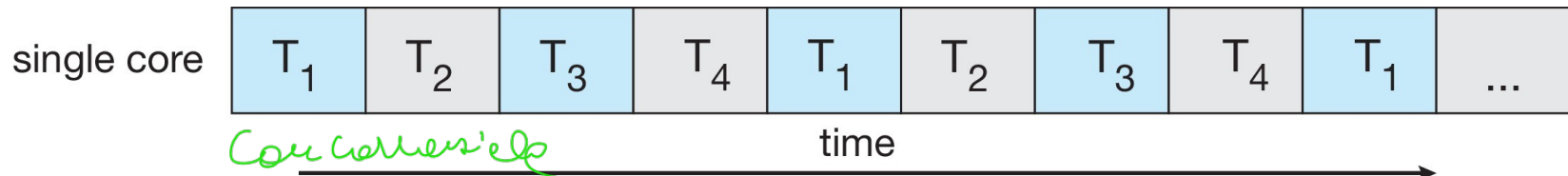
ancora più difficile schedare



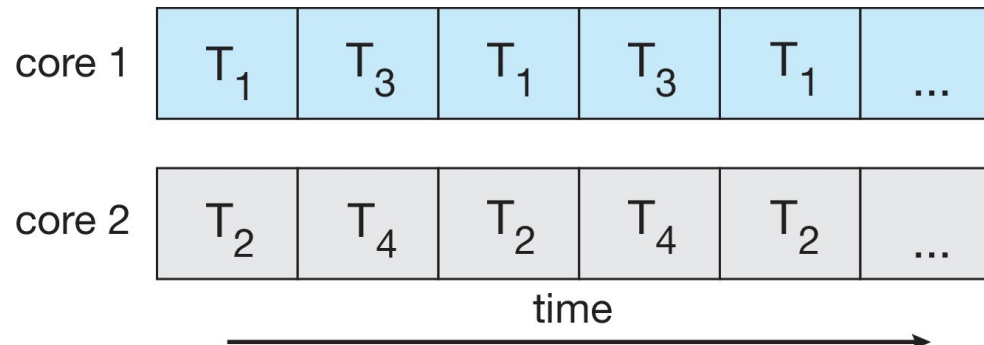


Concorrenza vs Parallelismo

❖ Esecuzione concorrente su un sistema single core



❖ Parallelismo nei sistemi multicore *ho due core ho parallelismo*





Programmazione multicore – 2

❖ Tipi di parallelismo

Una stessa operazione può essere eseguita in parallelo su più core su sottoinsiemi di dati.

- Con il termine **data parallelism** ci si riferisce a scenari in cui la stessa operazione viene eseguita contemporaneamente (ovvero in parallelo) su sottoinsiemi dei dati; nelle operazioni in parallelo sui dati, infatti, l'insieme originale viene suddiviso in partizioni in modo che più thread (che compiono la stessa operazione) possano agire simultaneamente su segmenti diversi
- Con il termine **task parallelism** si indica una forma di parallelizzazione del codice tra più processori in ambienti di calcolo parallelo; il task parallelism si concentra cioè sulla distribuzione di thread diversi in esecuzione nei diversi nodi di calcolo (che possono agire su dati comuni)

ogni core opera su un insieme di dati differenti, in parallelo

→
Si core diff. hanno thread che eseguono op. diversi dello stesso programma su dati uguali o differenti

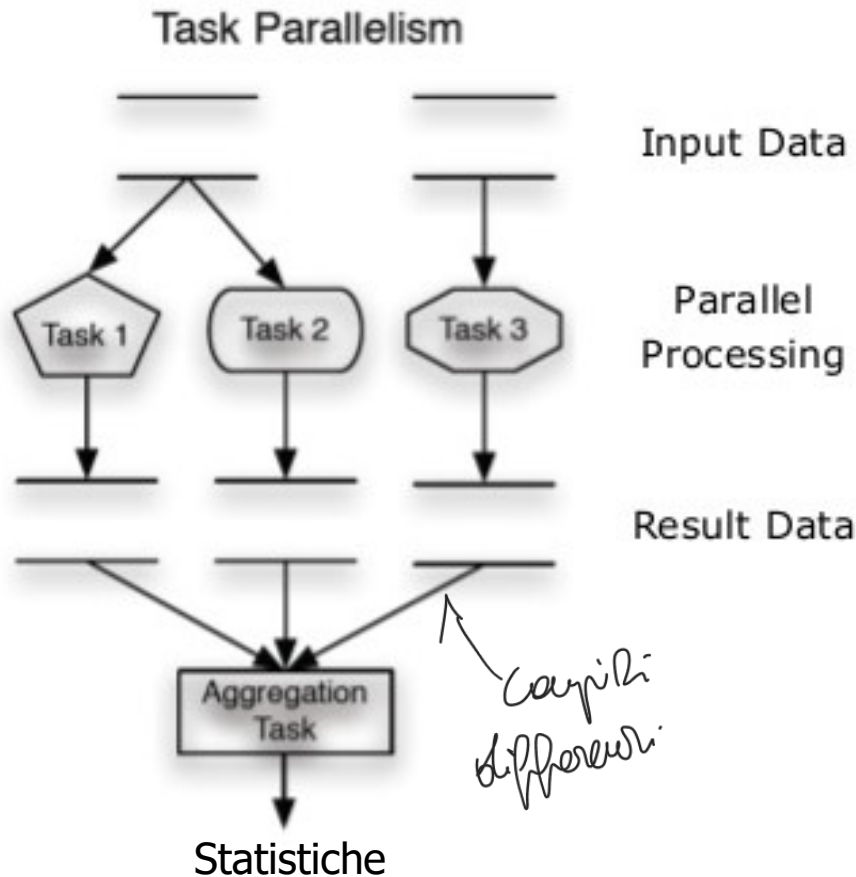
❖ Nella maggior parte dei casi, le applicazioni utilizzano un ibrido delle due strategie



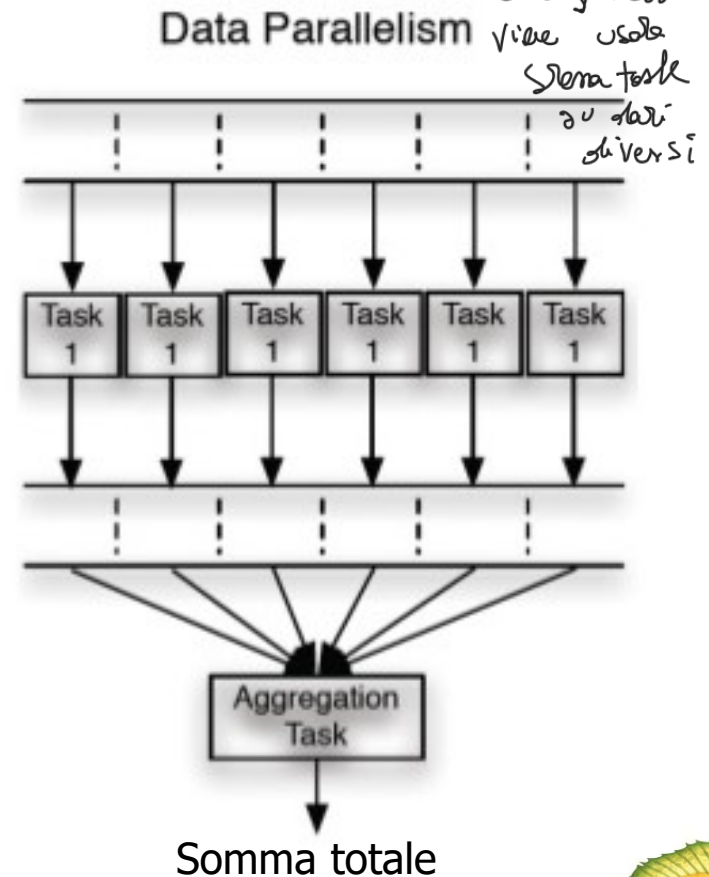


Programmazione multicore – 3

Calcolo di max e min e della media degli elementi di un vettore



Calcolo della somma degli elementi di un vettore



nella realtà, natura ibrida, uso esemplare





Programmazione multicore – 4

❖ Obiettivi della programmazione nei sistemi multicore

es. dati comuni
e compiti diversi
e ogni compito è
in threads

Separazione dei task: esaminare le applicazioni per individuare aree separabili in task distinti che possano essere eseguiti in parallelo

Per computazione dei thread deve essere bilanciato

- **Bilanciamento:** produrre task che eseguano compiti di mole confrontabile (per carico computazionale)

dati suddivisi
contribuzione, esempio
architettura NUMA,
RAM UNICA

Suddivisione dei dati: i dati a cui i task accedono, e che manipolano, devono essere suddivisi per essere utilizzati da unità di calcolo distinte

porzione cache & CPU

- **Dipendenze nei dati:** verificare le dipendenze tra due o più task per sincronizzarne l'esecuzione

Test e debugging: garantire la correttezza malgrado la possibilità di flussi di esecuzione multipli e di eventuali problemi di concorrenza nell'accesso ai dati

è possibile che
non tutti i thread
possono accedere
in parallelo e
devono essere
sequenziali

↳ uso programmi, debugger

↳ difficile verificare
correttezza dati





Programmazione multicore – 5

ogni core può essere visto come più core logici distinti in cui fare scheduling

multithreading Hardware

- ❖ In corrispondenza con la proliferazione di programmi multithread, le nuove architetture hardware forniscono sempre maggior supporto al multithreading
- ❖ Progetti hardware recenti (*hyperthreading*) implementano unità di calcolo multithread in cui due o più thread hardware sono assegnati ad un singolo core
 - Quando un processore accede alla memoria, infatti, una quantità significativa di tempo (fino al 50%) trascorre in attesa della disponibilità dei dati: **stallo della memoria**
- ❖ Dal punto di vista del SO, ogni thread hardware appare come un processore logico in grado di eseguire un thread software

Un processore si trova ad eseguire una istruzione in un solo core in memoria e non nei registri (cache) (sono in RAM); entro lo stallone della memoria

Mentre un core è in stallone, S.O. vede questo core come 2 CPU, e scheduler separa i thread in stallone, però al thread è e' col HW





Programmazione multicore – 6

- ❖ **Esempio:** il microprocessore Oracle SPARC T4 (2011) ha otto core, ciascuno dotato di otto thread hardware
⇒ 64 unità di calcolo distinte che il SO deve gestire

*few silberschatz 64 processi logici
così sempre CPU usata*

C

M

Ciclo di computazione

Accesso alla memoria



Stallo della memoria

Thread₁



Thread₀



Sistema con multithreading hardware (relativo ad un core)





La legge di Amdahl – 1

quantità nuovi core, come avere
trasmissioni?
non tutto il programma può
essere parallelizzato

- ❖ Permette di determinare i potenziali guadagni in termini di prestazioni ottenuti dall'aggiunta di core, nel caso di applicazioni che contengano sia componenti seriali (non parallelizzabili) sia componenti parallele

- ❖ S porzione seriale

- ❖ N core *quanto moltiplica
il tempo di esecuzione?*

$$\text{speedup} \leq \frac{1}{S + \frac{(1-S)}{N}}$$

aumento prest.
non è lineare

- ❖ **Esempio:** Se un'applicazione è al 75% parallela ed al 25% seriale, passando da 1 a 2 core, si ottiene uno speedup pari a 1.6; se i core sono 4, lo speedup è invece di 2.28

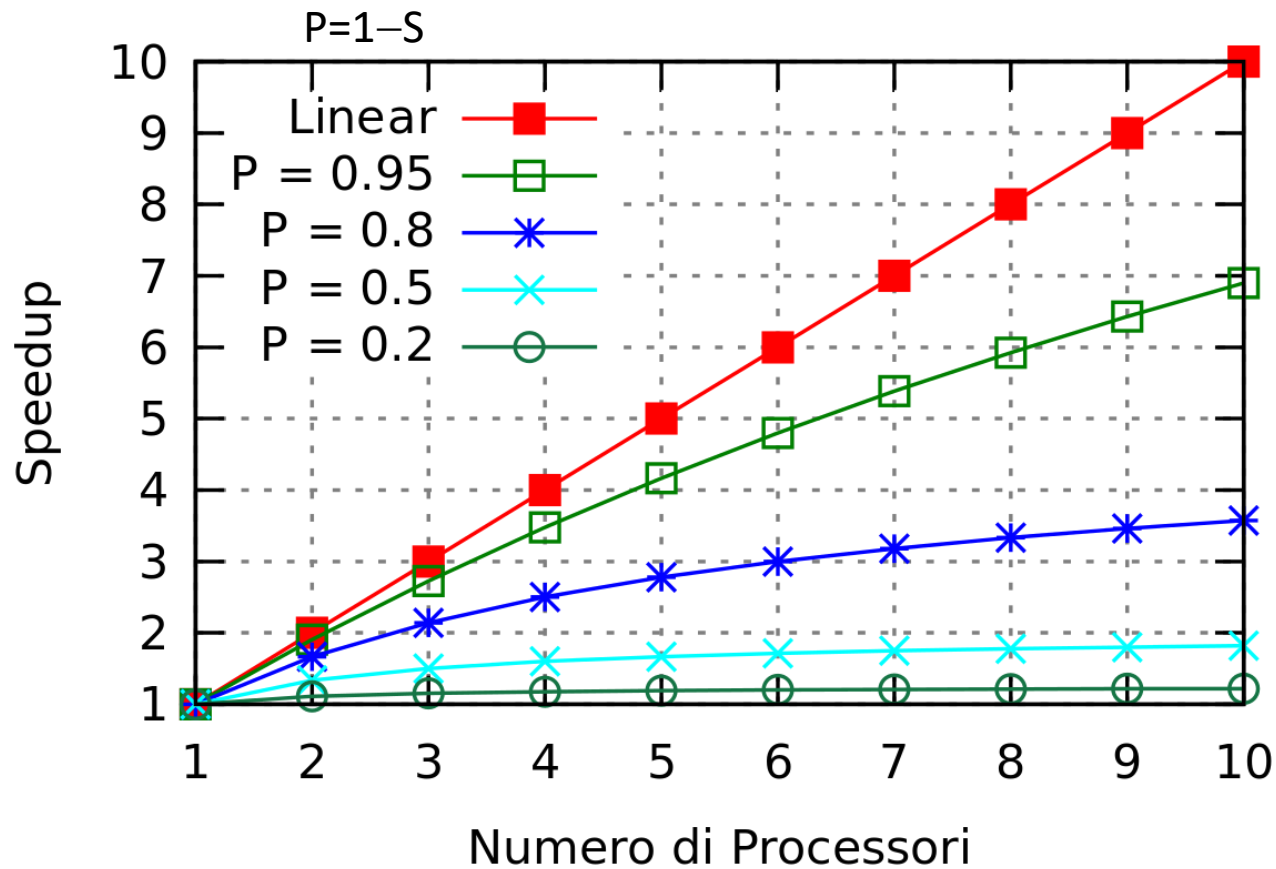
- ❖ In generale... per $N \rightarrow \infty$ lo speedup tende a $1/S$ es. qui L_1
⇒ Se il 40% di un'applicazione è seriale il massimo aumento di velocità è di 2.5 volte *speed up limitato dalla parte seriale, non riproducibile*

- ❖ La porzione seriale di un'applicazione ha un effetto dominante sulle prestazioni ottenibili con l'aggiunta di nuovi core





La legge di Amdahl – 2





Supporto del SO ai thread – 1

Supporto a livello kernel.

- 2 silhouette*
- ❖ Supporto user-level, tramite librerie di funzioni (API) per gestire n thread in esecuzione
 - ❖ Supporto kernel-level, tramite "un'astrazione di traccia di esecuzione" (su un processore virtuale)
- a livello UTENTE, NO SYSCALL*
kernel si occupa del suo scheduling
- kernel interviene pesantemente*
- ## Thread a livello utente

- Sono gestiti come uno strato separato sopra il nucleo del sistema operativo *no intervento kernel*
- Sono realizzati tramite librerie di funzioni *che formano API* per la creazione, lo scheduling e la gestione dei thread, senza alcun intervento diretto del nucleo
- POSIX Pthreads è la libreria per la realizzazione di thread utente in sistemi UNIX-like
- Windows threads per sistemi Windows e Java threads per la JVM





Supporto del SO ai thread – 2

❖ Thread a livello kernel

- Sono gestiti direttamente dal SO: il nucleo si occupa di ^{the thread} ~~creazione, scheduling, sincronizzazione e cancellazione~~ dei thread nel suo spazio di indirizzi
- Supportati da tutti i sistemi operativi attuali
 - ▶ Windows
 - ▶ ~~Solaris~~ ^{obsolete}
 - ▶ Linux
 - ▶ Mac OS
 - ▶ iOS
 - ▶ Android





Modelli di programmazione multithread

una possibilità
→ ❖

Modello multi-a-uno (M:1)

Non c'è vero parallelismo tra thread della stessa app

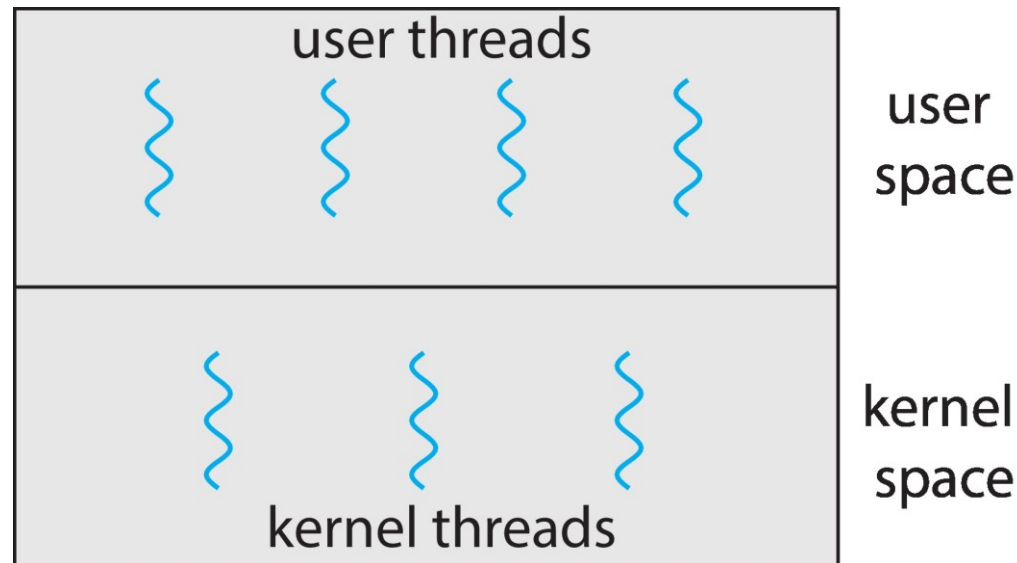
Come si agitano i thread a livello utente su thread a livello utente

❖ **Modello uno-a-uno (1:1)**

COMUNE. Corrisponde 1:1 e parallelismo

❖ **Modello multi-a-molti (M:M)**

IBM OS, non implementato



Kernel gestisce thread, non processi

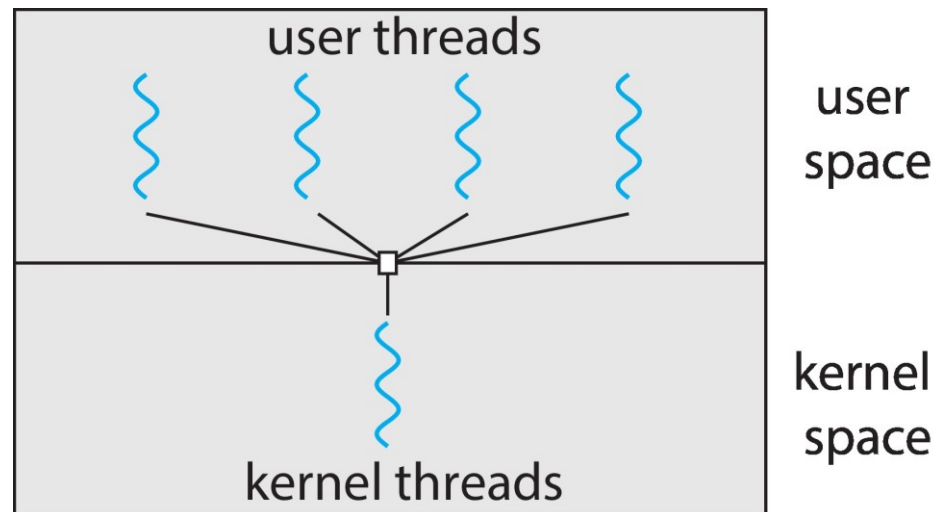




Modello multi-a-uno – 1

*libreria utente schedula i thread
ma kernel schedula in solo thread*

- ❖ Molti thread a livello utente vanno a corrispondere ad un unico thread a livello kernel
a livello utente deciso un unico thread che entra nella READY QUEUE
 - Il kernel “vede” una sola traccia di esecuzione
- ❖ In altre parole: i thread sono implementati a livello di applicazione, il loro scheduler non fa parte del SO, che continua ad avere solo la visibilità del processo





Modello multi-a-uno – 2

❖ Vantaggi

- Gestione efficiente dei thread nello spazio utente (scheduling poco oneroso) *efficiente e poco pericoloso*
- Non richiede un kernel multithread per poter essere implementato *kernel vede una processa unica, poco usare kernel unico*

❖ Svantaggi *Tempo di risposta lento, no parallelismo*

- L'intero processo rimane bloccato se un thread invoca una chiamata di sistema di tipo bloccante
- I thread sono legati allo stesso processo a livello kernel e non possono essere eseguiti su processori fisici distinti

❖ Attualmente, pochi SO implementano questo modello; sono esempi storici:

- GNU Portable Threads (2006)
- Solaris Green Threads (2010)

è qm ha un unico thread in esecuzione

No parallelismo anche con più core, solo su processori differenti.

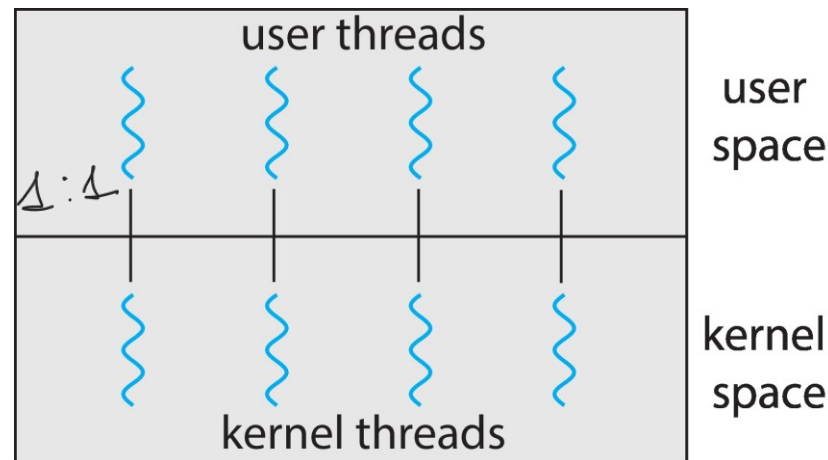




Modello uno-a-uno – 1

*tutti i thread di tutte le app partecipano alla contesa
azione della CPU*

- ❖ Ciascun thread a livello utente corrisponde ad un thread a livello kernel
 - Il kernel “vede” una traccia di esecuzione distinta per ogni thread
 - I thread vengono gestiti dallo scheduler del kernel (come se fossero processi; si dicono thread nativi)



*scheduling fatto
dal S.O.*





Modello uno-a-uno – 2

❖ Vantaggi

- Scheduling molto efficiente
- Se un thread effettua una chiamata bloccante, gli altri thread possono proseguire nella loro esecuzione *es. si muove in attesa di I/O*
- I thread possono essere eseguiti su processori fisici distinti *PARALLELISMO della stessa app*

❖ Svantaggi

- Possibile inefficienza per il carico di lavoro dovuto alla creazione di molti thread a livello kernel (limiti imposti a priori) *molto carico nel sistema, scheduling difficile*
- Richiede un kernel multithread per poter essere implementato *posso avere limiti posti a priori*
↳ ho thread nativi

❖ Esempi

- Windows 95/98/NT/2000/XP/Vista e versioni attuali
- Linux, Solaris (versione 9 e successive)



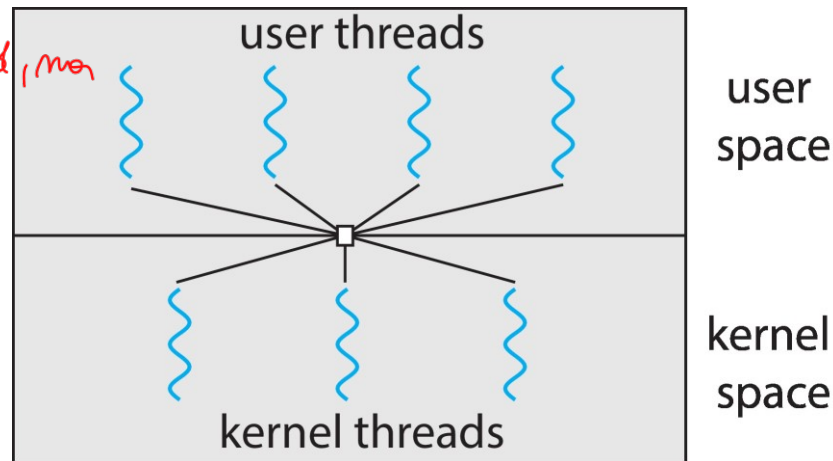


Modello multi-a-molti – 1

- ❖ Si mettono in corrispondenza più thread a livello utente con un numero minore o uguale di thread a livello kernel *molti user threads > molti kernel*
 - In altre parole, il sistema dispone di un pool di thread – *worker* – ognuno dei quali viene assegnato di volta in volta ad un thread utente

*C'è Scheduler a livello utente
posso schedulare più thread, ma
un solo sistema*

*Posso regolare il carico
sul sistema*





Modello multi-a-molti – 2

❖ Vantaggi

- Possibilità di creare tanti thread a livello kernel quanti sono necessari per la particolare applicazione (e sulla particolare architettura), eseguibili in parallelo su architetture multiprocessore
- Se un thread invoca una chiamata di sistema bloccante, il kernel può fare eseguire un altro thread



❖ Svantaggi

- Difficoltà nel definire la dimensione del pool di worker e le modalità di cooperazione tra i due scheduler

❖ Esempi:

- Windows con il ThreadFiber package
- Solaris (versioni precedenti alla 9)
- Tru64 UNIX

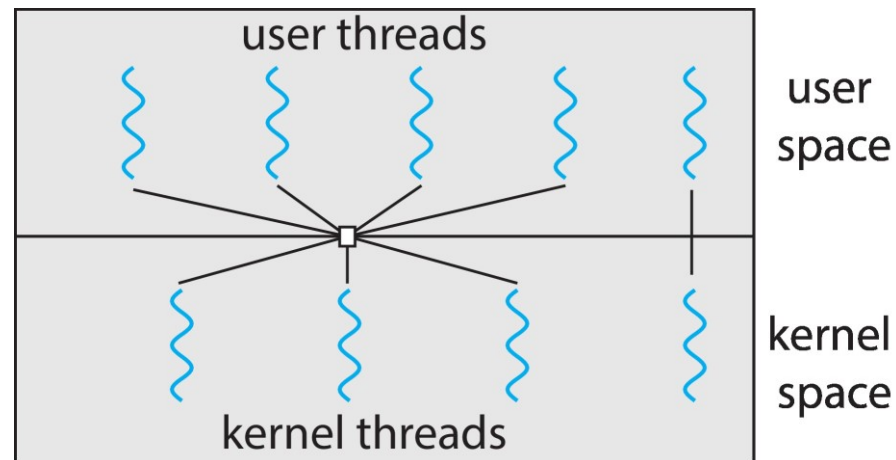
poco usato





Modello a due livelli

- ❖ Simile al modello M:M, offre però la possibilità di vincolare un thread utente ad un thread del kernel
- ❖ Esempi:
 - IRIX
 - HP-UX
 - Solaris (versioni precedenti alla 8)
 - Tru64 UNIX





Librerie dei thread – 1

per Neelzone thread

- ❖ Forniscono al programmatore una API per la creazione e la gestione dei thread *Applic. Programming Interface*

- ❖ Librerie a livello utente

- Codice e strutture dati nello spazio utente
- Invocare una funzione di libreria si traduce in una chiamata locale a funzione, non in una system call

- ❖ Librerie a livello kernel

- Codice e strutture dati nello spazio del kernel
- Invocare una funzione della API provoca, generalmente, una chiamata di sistema

↳ chiama f. libreria => syscall





Librerie dei thread – 2

Standard IEEE

- ❖ **POSIX Pthreads** = Specifico per implementare Specifiche
 - Può presentarsi sia come libreria a livello utente sia a livello kernel
 - ❖ **Win64**
 - Libreria di thread a livello kernel per sistemi Windows
 - ❖ **Java**
 - API gestibile direttamente dai programmi Java
 - La API di Java per i thread è solitamente implementata per mezzo della libreria dei thread del sistema che ospita la JVM
- principale a livello kernel*
- Java VM fa riferimento all'HW sottostante
es. se c'è sotto WINDOWS, saranno gestiti da LW*





Librerie dei thread – 3

- ❖ Nel threading di POSIX e di Windows tutti i dati dichiarati a livello globale, ovvero al di fuori da ogni funzione, sono condivisi fra tutti i thread appartenenti allo stesso processo

Dati globali: dati che un processo condivide con i thread

Dato globale e' tutto cio' che e' dichiarato fuori da MAIN e altre FUNZIONI

- ❖ I dati locali di una funzione sono, normalmente, memorizzati nello stack

Memoria Globale non sono in stack, occupano spazio

- ❖ Ogni thread ha un proprio stack

⇒ ogni thread ha la propria copia di dati locali

Thread Local STORAGE; variabili che nel C sono variabili STATIC

→ durata e' globale, ma visibilita' e' LOCALE, visibile solo dalla funzione che la dichiara





Strategie di threading

❖ Threading asincrono

Un processo crea un thread e poi continua la sua esecuzione in concorrenza

- Il genitore crea un figlio e quindi riprende la propria esecuzione in concorrenza

es. nel server: continua a fare il suo e delega il thread a fare un'altra operazione

Ogni thread viene eseguito in modo indipendente rispetto agli altri ed il thread genitore non ha bisogno di sapere quando terminano i figli

Similare e asincrono, segue diverso

Scarsa condivisione dei dati fra i thread

ogni thread ha solo bisogno di un proprio set di dati. interrupt hanno guasto del processore, segue te e' esterno del comm.

- Esempio: server web, interfacce utente responsive

❖ Threading sincrono

- Il genitore crea uno o più figli e attende che tutti terminino prima di riprendere l'esecuzione ⇒ fork-join

No CONCURRENZA padre-figlio

- I thread figli si eseguono in concorrenza

non con il padre

- Quando un thread ha terminato il proprio compito, termina e si unisce al genitore

lo rende noto al genitore che la terminato

- Significativa condivisione dei dati fra i thread

- Il genitore può combinare i risultati calcolati dai figli

genitore deve combinare i dati prodotti dai figli





Pthreads

- ❖ È lo standard POSIX (IEEE 1003.1c) che definisce la API per la creazione e la sincronizzazione dei thread (definizione – no realizzazione) → dipende dai S.O.
- ❖ Viceversa, è la API che specifica il comportamento della libreria dei thread, la cui implementazione dipende dal particolare sistema operativo (può essere a livello utente o a livello kernel)
- ❖ Comune nei SO UNIX-like (Solaris, Linux, Mac OS)
- ❖ **Esempio:** threading sincrono per il calcolo della somma dei primi N interi

Programma principale delega i thread.
Sottoprogramma è un programma a parte





Esempio con Pthreads – 1

```
/* Programma per il calcolo della somma dei primi N interi */
#include <pthread.h> /* include per la libreria Pthreads */
#include <stdio.h>
#include <stdlib.h>

int sum; /* variabile globale (condivisa) */ dato condiviso dal process
void *somma(char *param); funzione che darà vita ai thread /* funzione del nuovo thread */
int main(int argc, char *argv[])
{
    pthread_t tid; /* identificatore UNICO del thread */
    pthread_attr_t attr; /* attributi del thread */ Struttura dati che ha più attributi

    if (argc != 2) {
        fprintf(stderr, "Occorreva inserire N!");
        exit(1); devo inserire un numero come argomento da linea di comando
    }
    if (atoi(argv[1]) < 0) {
        fprintf(stderr, "%d deve essere >=0\n", atoi(argv[1]));
        exit(1);
    }

    pthread_attr_init(&attr); /* reperisce attributi predefiniti */
    pthread_create(&tid, &attr, somma, argv[1]); /* crea il thread */ NOME ARGOMENTO
    /* il padre attende la terminazione del nuovo thread */
    pthread_join(tid, NULL); ricomprico figlio attraverso tid
    printf("somma = %d\n", sum);
    exit(0); Se c'è successo buon fine
}
```

es. dati dello stack
Se riesco lo stack, finisco spazio degli indirizzi
controlli
inizializzo gli attrib. del thread con quelli predefiniti
Dimensione della pila, informazioni per lo scheduling, etc.
è una stringa costante
recupero del processo padre, uso la shell



↳ Sono nome sottoprogramma che viene eseguito come corpo del Thread

Esempio con Pthreads – 2

Padre ha un solo figlio

```
/* Il nuovo thread assume il controllo da questa funzione */  
void *somma(char *param);  
{  
    int i, upper=atoi(param);  
    sum = 0;  
  
    for (i=1, i<=upper; i++)  
        sum += i;  
  
    pthread_exit(0);  
}
```

Se ho più figli, devo fare il join di più figli.





Join in Pthreads

```
#define NUM_THREADS 10
```

*creo 3 threads
LAVORATORI*

```
/* an array of threads to be joined upon */  
pthread_t workers[NUM_THREADS];
```

```
for (int i = 0; i < NUM_THREADS; i++)  
    pthread_join(workers[i], NULL);
```

*Si fa vestizione Petri e lo i tibi prima di
lavorare l'esecuzione*





Threading implicito

Interazione thread difficile

- ❖ Le applicazioni multithread sono più difficili da programmare ed aumentano in complessità al crescere del numero dei thread *Difficile da programmare, Difficile prendere errori di sincronizzazione*
- ❖ Trasferimento della creazione e della gestione del threading dagli sviluppatori di applicazioni ai compilatori ed alle librerie di runtime

⇒ Threading implicito

- Gruppi di thread (Thread pool – Windows, Java)
- OpenMP C, C++... *insieme di direttive al compilatore che include il codice*
- Grand Central Dispatch (Mac OS e iOS), Java Fork-Join, Intel Threading Building Blocks *sia parallelo che*





Gruppi di thread – 1

*non come thread di un web server
che si creano in base alle richieste client*

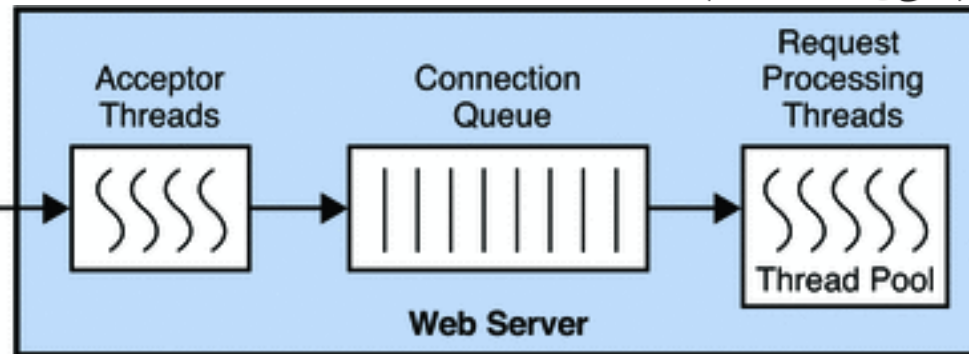
❖ Un numero illimitato di thread presenti nel sistema potrebbe esaurirne le risorse

→ già creati, esistono già

⇒ Creare un certo numero di thread (worker) alla creazione del processo ed organizzarli in un gruppo in cui attendano il lavoro che verrà loro successivamente richiesto

Per thread definiti a priori, non caricano il server stesso, max 10 thread, per non max 10 richieste.

• Se ho lo più pronto, non posso tempo per crearlo, devo solo programmare subito per eseguire il lavoro, meno tempo res.





Gruppi di thread – 2

- ❖ Vantaggi *Separo creazione da esecuzione -
=> Thread più in Memoria*
 - Di solito il servizio di una richiesta tramite un thread esistente è più rapido, poiché elimina l'attesa della creazione di un nuovo thread
 - Si limita il numero di thread relativi a ciascun processo alla dimensione prestabilita (rilevante per sistemi che non possono sostenere un numero elevato di thread concorrenti) *Coppio di creazione diverso dal coppia di esecuzione*
 - Separare il task da eseguire dal meccanismo di creazione dello stesso permette strategie diverse di elaborazione *poss o periodici*
 - ▶ **Esempio:** i task possono essere schedulati per essere eseguiti periodicamente o dopo un dato intervallo di tempo *per un tempo ben preciso*
- ❖ La API per i thread di Windows supporta i thread pool





OpenMP – 1

- ❖ **OpenMP** è un insieme di direttive del compilatore ed una API per programmi scritti in C, C++ e FORTRAN
- ❖ Fornisce supporto per la programmazione parallela in ambienti a memoria condivisa
- ❖ Identifica le **regioni parallele**, cioè i blocchi di codice eseguibili in parallelo

- ❖ **Esempio**

#pragma omp parallel

crea tanti thread quanti sono i core^{disponibili}, mentre
L'è processori virtuali

#pragma omp parallel for

```
for(i=0; i<N; i++) {  
    c[i]=a[i]+b[i];  
}
```

avvno' paralleli sono dei dati, non solo task

divide il compito di esecuzione del ciclo **for** fra i diversi thread (tanti quanti sono i core)

Se non viene specificato #thread = #CPU virtuali





OpenMP – 2

- ❖ **Esempio:** si producono tante stampe quanti sono i core

```
#include <omp.h>
#include <stdio.h>

int main(int argc, char *argv[])
{
    /* sequential code */

    #pragma omp parallel una oper. di 5 steps
    {
        printf("I am a parallel region.");
    } come dire qual era

    /* sequential code */

    return 0;
}
```

- ❖ OpenMP permette inoltre di impostare manualmente il numero di thread ed il livello di condivisione dei dati fra i thread

Dov'è sapere quale è il codice parallelizzabile





Threading

Come si regolano i processi di Pauli (che si relazionano a:

❖ Semantica delle chiamate di sistema **fork()** ed **exec()** *cosa succede se thread riceve segnale o fa fork?*

❖ Gestione dei segnali (sincroni vs asincroni)

❖ Cancellazione dei thread (asincrona vs differita)

per terminare processo è facile, PERIL PID FUGA

❖ Dati specifici dei thread

*ma ora con i thread condiviso dati
in nome ad altri thread*

❖ Attivazione dello scheduler

es. se sono a MOLT-MOLT

* Thread in LINUX





Semantica di `fork()` ed `exec()`

figlio exec,
sostituisce
il padre

di solito si copia sono involontario no come padre, figlio eredita.

❖ In un programma multithread, la semantica della system call `fork()` cambia:

Se thread
dopo fork fa una
exec, non duplica
risorse,

- Se un thread in un programma invoca la `fork()`, il nuovo processo potrebbe, in generale, contenere un duplicato di tutti i thread oppure del solo thread invocante
- La scelta fra le due opzioni dipende dalla immediatezza o meno della successiva chiamata ad `exec()` – la cui modalità di funzionamento non cambia
 - ▶ Se la chiamata di `exec()` avviene immediatamente dopo la chiamata alla `fork()`, ~~la duplicazione dei thread non è necessaria~~, poiché il programma specificato nei parametri della `exec()` sostituirà in toto il processo ⇒ si duplica il solo thread chiamante 
 - ▶ Altrimenti, tutti i thread devono essere duplicati 
- Alcune release di UNIX rendono disponibili entrambe le implementazioni della `fork()`

Se si duplicano
tutti i thread
del processo

Solo se si duplicano tutti i processi o no.

So. deve sapere dopo cosa verrà eseguito, se c'è un exec





Gestione dei segnali – 1 *es. interazioni*

- ❖ Nei sistemi UNIX si usano segnali per comunicare ai processi il verificarsi di determinati eventi
- ❖ I segnali vengono elaborati da un gestore dei segnali *servizio del S.O.*
 - All'occorrenza di un particolare evento, si genera un segnale
 - S'invia il segnale ad un processo
 - Una volta ricevuto, il segnale deve essere gestito da un gestore, che può essere:
 - ▶ predefinito (gestito dal SO)
 - ▶ definito dall'utente *difficile da verificare implementati*
- ❖ I segnali possono essere...
 - sincroni, se vengono inviati allo stesso processo che ha eseguito l'operazione causa del segnale (es.: divisione per 0) *genera e riceve il segnale*
 - asincroni, se causati da eventi esterni al processo in esecuzione (es.: scadenza di un timer o $\uparrow C$)

Control - Z invece sospende, con fg, programma lo riprende, serve

Control - C, uccide processo

4.50 Silberschatz, Galvin and Gagne





intero

Gestione dei segnali – 2

- invio un segnale* *kill - 9 = Ctrl - C*
- ❖ Nei sistemi UNIX-like, la funzione standard per l'invio di un segnale è kill (`pid_t pid, int signal`)
 - ❖ ~~Per ogni segnale esiste un gestore predefinito del segnale~~, che il kernel esegue in corrispondenza del particolare evento *opportune occas. richieste del segnale*
 - Il gestore definito dall'utente, se presente, si sostituisce al gestore standard
 - Nei processi a singolo thread, i segnali vengono inviati al processo
 - ❖ Cosa accade nel caso di processi multithread?





Gestione dei segnali – 3

❖ Nei sistemi multithread è possibile...

- inviare il segnale al thread cui il segnale si riferisce *Se un thread crea un altro thread, problema, uccido un solo thread*
- inviare il segnale ad ogni thread del processo *es. Ctrl-C, uccido anche il thread*
- inviare il segnale a specifici thread del processo
- definire un thread specifico per ricevere tutti i segnali diretti al processo *Thread che cattura segnali e gestisce*

*all'ho alle
vices. di
altri segm solo
altri thread*

*gestione deve
intervenire una
sola volta*

- ▶ I segnali sincroni devono essere recapitati al thread che ha generato l'evento causa del segnale
- ▶ Alcuni segnali asincroni (per esempio il segnale di terminazione di un processo, ^C) devono essere inviati a tutti i thread
- ⇒ Possibilità di specificare, per ciascun thread, quali segnali accettare e quali bloccare (`pthread_kill()`)
- ⇒ Tuttavia, poiché i segnali vanno gestiti una sola volta, il segnale è recapitato al primo thread che non lo blocca





Cancellazione di thread

devo stare attento
a non vedere altri
LUCONGWEITI

- ❖ È l'operazione che permette di terminare un thread prima che completi il suo compito
 - **Esempio:** pressione del pulsante di terminazione di un programma di consultazione del Web per interrompere il caricamento di una pagina
- ❖ Il thread da cancellare viene definito thread bersaglio
- ❖ La cancellazione di un thread bersaglio può avvenire...
 - ...in modalità asincrona: terminazione immediata del thread bersaglio *✓ si rius. in cui si trova, cancella*
 - ▶ Problemi se si cancella un thread mentre sta aggiornando dati che condivide con altri thread
 - *STANDARD* → ...in modalità differita: ~~il thread bersaglio può periodicamente controllare se deve terminare, in modo da riuscire al momento opportuno (cancellation point, in Pthreads)~~ *controllo in tempi ben definiti. fatti in cui se thread viene cancellato, i dati sono corrotti.*

devo fare una read, lo cancella prima della lettura





Cancellazione di thread in Pthread – 1

- ❖ Codice Pthread per creare e cancellare un thread

```
pthread_t tid;
```

```
/* create the thread */  
pthread_create(&tid, 0, worker, NULL);
```

```
. . .
```

```
/* cancel the thread */  
pthread_cancel(tid);
```

*← non è solo che venga fatto sul thread
dipende dallo stato*

- ❖ La chiamata della **pthread_cancel()** comporta solo una richiesta di cancellazione del thread bersaglio: l'effettiva cancellazione dipende dallo stato del thread

Mode	State	Type
Off	Disabled	-
Deferred	Enabled	Deferred
Asynchronous	Enabled	Asynchronous

*Un certo thread viene cancellato solo se non ha i flag
su stati cancellabili*





Cancellazione di thread in Pthread – 2

- ❖ In caso di cancellazione disabilitata, l'operazione rimane pendente finché il thread non procede all'abilitazione
- ❖ Per ogni thread, lo stato di default è "differita"
 - La cancellazione avviene solo al momento in cui il thread raggiunge un punto di cancellazione
 - ▶ Il punto di cancellazione viene creato con una invocazione alla funzione ~~pthread_testcancel()~~
 - ▶ Si richiama quindi il cleanup handler, che permette di rilasciare tutte le risorse del thread
- ❖ Nei sistemi Linux, la cancellazione dei thread è gestita tramite segnali





Dati specifici dei thread

Dati LOCALI al thread, analogo a Dati statici

- ❖ Il **Thread Local Storage** (TLS) permette a ciascun thread di possedere una propria copia dei dati (non necessariamente tutti i dati del processo di origine)

*Utile
Solo del
thread
che li
porta in*

- **Esempio:** In un sistema per transazioni, si può svolgere ciascuna transazione tramite un thread distinto ed un identificatore unico per ogni transazione *es. Banca,*

*vi è solo
va dove quello
del singolo
thread*

- Lo stesso TID può essere considerato un dato TLS
- ❖ Utili quando non si ha controllo sul processo di creazione dei thread (per esempio, nel caso dei gruppi di thread)
- ❖ Diversamente dalle variabili locali (che sono visibili solo durante una singola chiamata di funzione), i dati TLS sono visibili attraverso tutte le chiamate
- ❖ Simili ai dati dichiarati **static** in C
 - I dati TLS sono però unici per ogni thread





Attivazione dello scheduler – 1

thread utente > # thread kernel

❖ Sia il modello M:M che il modello a due livelli richiedono che sia mantenuta una comunicazione costante per garantire un numero sufficiente di thread del kernel allocati ad una particolare applicazione *il SO usa una metafora del*

- Si alloca una struttura dati intermedia nota come processo leggero o **LWP** (*LightWeight Process*) *processo virtuale*
- Per la libreria dei thread a livello utente, LWP è un processore virtuale a cui l'applicazione può richiedere lo scheduling di un thread a livello utente
- Corrispondenza fra LWP e thread a livello kernel: quanti LWP dovranno essere creati per una data applicazione?
- I kernel thread vanno in esecuzione sui processori fisici
 - ▶ Se un thread del kernel si blocca, l'effetto si propaga attraverso l'LWP fino al thread al livello utente ad esso associato

Kernel SO mette a disposizione di ogni App un # di processi virtuali = # thread a livello kernel che possono eseguire a quella App





* thread che aspetta come SCHEDULABILI (non si perdono slot) anche l'utente e del proprio

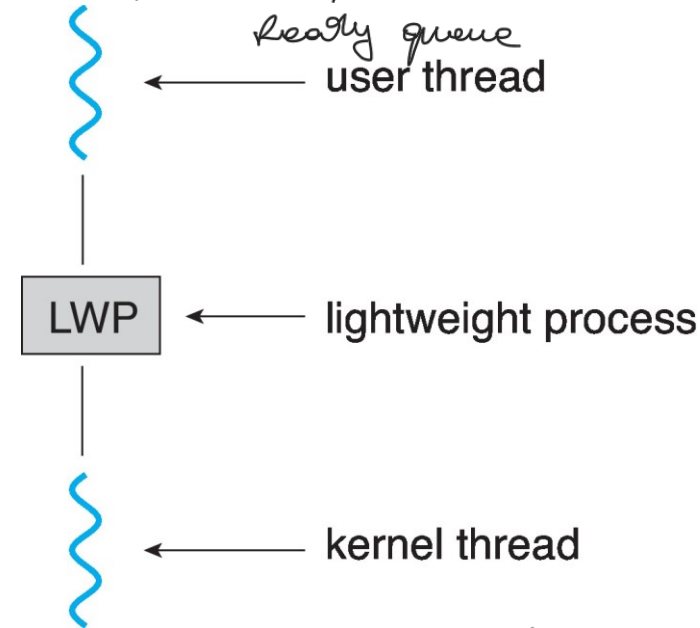
Attivazione dello scheduler – 2

- ❖ L'attivazione dello scheduler mette a disposizione la procedura di upcall – un meccanismo di comunicazione dal kernel alla libreria di gestione dei thread

- Si informa l'applicazione che un suo thread sta per bloccarsi
- Si assegna all'applicazione un nuovo processore virtuale su cui gestire il segnale (salvataggio del contesto del thread bloccante) e scheduling di un nuovo thread

- ❖ Tale comunicazione garantisce all'applicazione la capacità di mantenere attivi un numero opportuno di thread a livello kernel

LWP, processori virtuali,
1 Thread utente diventa un
Thread kernel, viene messo nella
ready queue
user thread



Ci si occupa di
un thread che per
bloccarsi

questo kernel si vuole conto che un thread è importante e richiede operos./160,
per avere che si sta per bloccare e
che si scopre un altro
4.58





L'utente si occupa di creare un altro thread e non il kernel

Thread in Linux – 1

Implementazione LINUX

non possono che thread

- ❖ Linux li definisce task (come i processi) usa una Task Struct con struttura locale
- ❖ La creazione dei task avviene con la chiamata della system call **clone()** rispetto alla fork si prende delle flag
- ❖ All'invocazione, **clone()** riceve come parametro un insieme di indicatori (flag), che definiscono quante e quali risorse del task genitore saranno condivise dal task figlio nella Task Struct, dati non sono condivisi, ma le porzioni di dati
 - **clone()** può permettere al task figlio la condivisione totale dello spazio degli indirizzi e delle risorse del task padre: si crea un thread può decidere cosa condividere

flag	meaning
CLONE_FS = 1	File-system information is shared.
CLONE_VM	The same memory space is shared.
CLONE_SIGHAND	Signal handlers are shared.
CLONE_FILES	The set of open files is shared.

Se flag arriva
Se Flag è arrivato, ho creato un thread, che condivide tutto.
Se non condivide nulla è un processo non condiviso nulla

non è possibile creare FS con padre
non Task condivide spazio memoria con chi l'ha creato



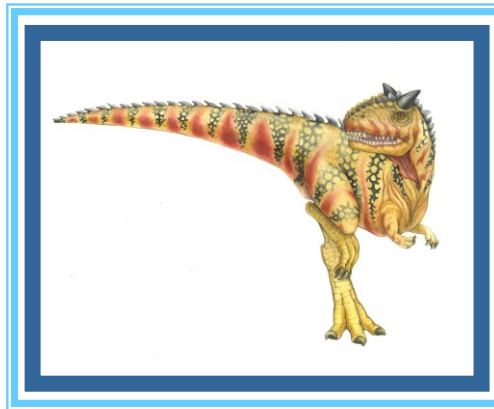


Thread in Linux – 2

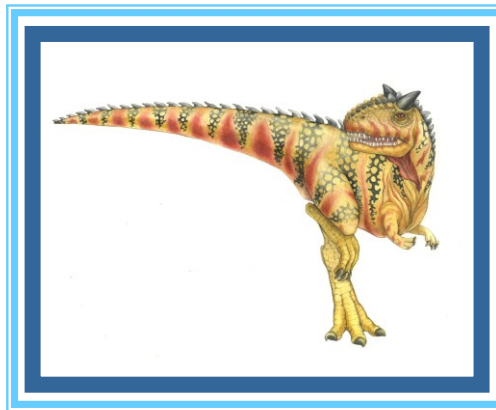
- ❖ Se nessun flag è impostato al momento dell'invocazione di `clone()`, non si ha alcuna condivisione: si ottiene una situazione analoga all'invocazione di una `fork()`
- ❖ Condivisione ad intensità variabile possibile perché:
 - Ogni task è rappresentato da una struttura dati nel kernel (TCB, `struct task_struct`, che punta alle strutture dati del processo, che siano esse uniche o condivise)
 - La struttura non contiene dati, ma puntatori alle strutture contenenti dati
 - La chiamata a `fork()` duplica tutte le strutture dati del task genitore
 - Con `clone()` il nuovo task non riceve una copia di tutte le strutture dati, ma solo alcuni puntatori a tali strutture (quelle del padre), in base ai parametri di chiamata di `clone()`



Fine del Capitolo 4



Scheduling della CPU





Obiettivi

- ❖ Descrivere vari algoritmi di scheduling della CPU *vale anche per gli I/O per il buffer cache del pc, non solo CPU*
- ❖ Valutare gli algoritmi in base ai criteri di scheduling
- ❖ Spiegare le problematiche relative allo scheduling multiprocessore e multicore
- ❖ Descrivere vari algoritmi di scheduling real-time
- ❖ Utilizzare modellazione e simulazioni per valutare gli algoritmi di scheduling della CPU in contesti reali





Sommario

- ❖ Scheduling della CPU: concetti fondamentali *parte fondamentale del sistema di calcolo, più peso*
- ❖ Criteri di scheduling *si devono scegliere a chi tocca* *più processi vivono nella RAM per non sprecares CPU (MULTITASKING)*
- ❖ Algoritmi di scheduling
- ❖ Scheduling dei thread
- ❖ Scheduling multiprocessore
- ❖ Scheduling real-time
- ❖ Scheduling in Linux
- ❖ Valutazione degli algoritmi





Concetti fondamentali – 1

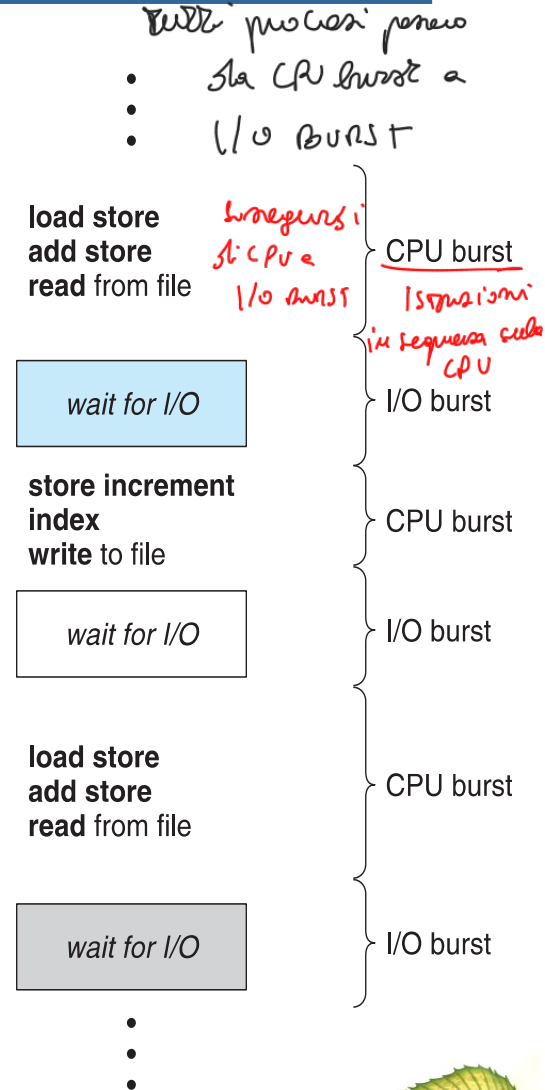
- ❖ Lo **scheduling** è una funzione fondamentale dei sistemi operativi
 - Si sottopongono a scheduling quasi tutte le risorse di un calcolatore
- ❖ Lo scheduling della CPU è alla base dei sistemi operativi multiprogrammati
 - Attraverso la commutazione del controllo della CPU tra i vari processi, il SO rende più “produttivo” il sistema di calcolo
 - Ottimizza l’utilizzo del/i processore/i





Concetti fondamentali – 2

- ❖ Il massimo impiego della CPU è pertanto ottenuto con la multiprogrammazione
- ❖ **Ciclo di CPU–I/O burst** – L'elaborazione di un processo consiste di *cicli* di esecuzione nella CPU ed attese/utilizzo ai/dei dispositivi di I/O
- ❖ I **CPU–burst** si susseguono agli **I/O–burst** durante tutta la vita del processo

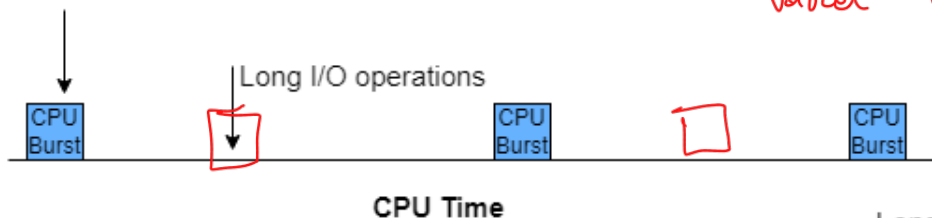




Concetti fondamentali – 3

- ❖ Un programma con prevalenza di I/O si definisce I/O-bound e produce generalmente molte sequenze di operazioni della CPU di breve durata *in base se sono lunghi*
- ❖ Un programma con prevalenza di elaborazione si definisce CPU-bound e produce (poche) sequenze di esecuzione molto lunghe
- ❖ Avere contemporaneamente presenti in memoria processi di entrambi i tipi garantisce l'uso intensivo e "bilanciato" di tutte le risorse del sistema

Short CPU Bursts



Processo CPU bound



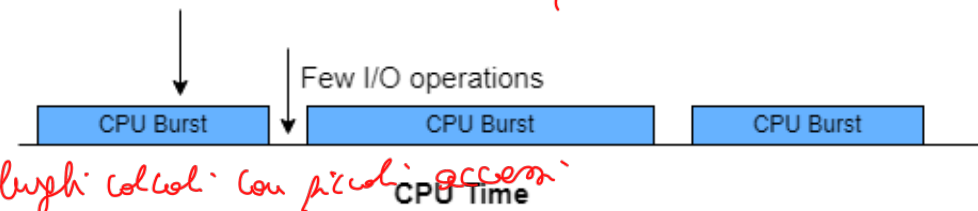
varrei un mix dei due => se CPU Bound, ovvero

lungissima ready queue

Processo I/O bound

se I/O bound, lunga coda ad accesso ai dispositivi

Long CPU Bursts



lunghi calcoli con piccoli accessi



Concetti fondamentali – 4

Scheduler: Completely Fair Scheduler

CPU scheduling

computers run 100s of programs at a time

processi in exec.

```
$ ps aux | wc -l
```

324

word count

I have 324 processes "running"

Case Concorde

but each CPU can only run 1 at a time

CPU 1: I'm running firefox!

CPU 2: I'm running gcc!

we're not on any CPU!

other programs

every program gets its turn on the CPU

Linux: CPU 1, run firefox!

... 1 ms later ...

Linux: ok, now run gcc!

Case for Linux the scheduler

what's running can change 1000x/second

cron: I just run for 1ms every now and then, don't mind me!

this system is called the "scheduler"

Linux's scheduling algorithm is called the "Completely Fair Scheduler"

NO STARVATION

MONITORING per FAIR

the scheduler wakes you up after a sleep

program: sleep (10)

... 10 seconds later ...

time to run!

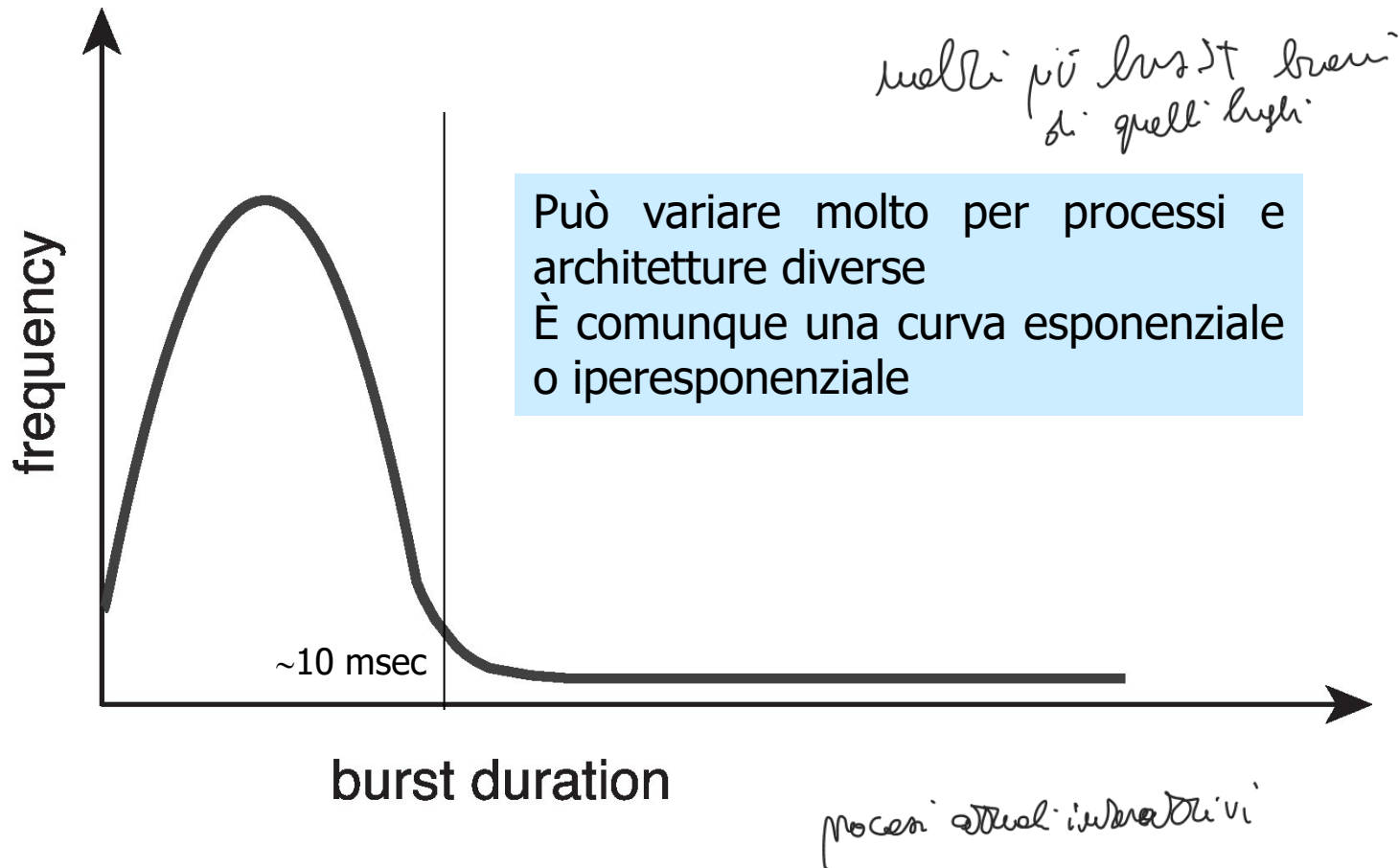
Linux

Tipica per definizione scheduling a priori





Grafico della distribuzione dei CPU-burst





Scheduling dei processi o dei thread?

- ❖ Nei SO che li supportano, l'oggetto dell'attività di scheduling sono i thread
- ❖ Nella presente trattazione useremo:
 - il termine scheduling dei processi ^{non CAMBIA NULLA, gli algoritmi sono gli stessi} per analizzare i concetti generali del CPU scheduling
 - la locuzione scheduling dei thread ^{thread a livello utente} nel trattare concetti ^{modello 1:1 si applica su thread a livello kernel} strettamente inerenti al multithreading ^{thread NATIVI}
- ❖ Allo stesso modo "eseguito su una CPU" coinciderà con "eseguito su un core" nel caso di architetture multicore
** Solo se il thread è legato al multicore specifico, altrimenti è totalmente uguale*





Lo scheduler della CPU – 1

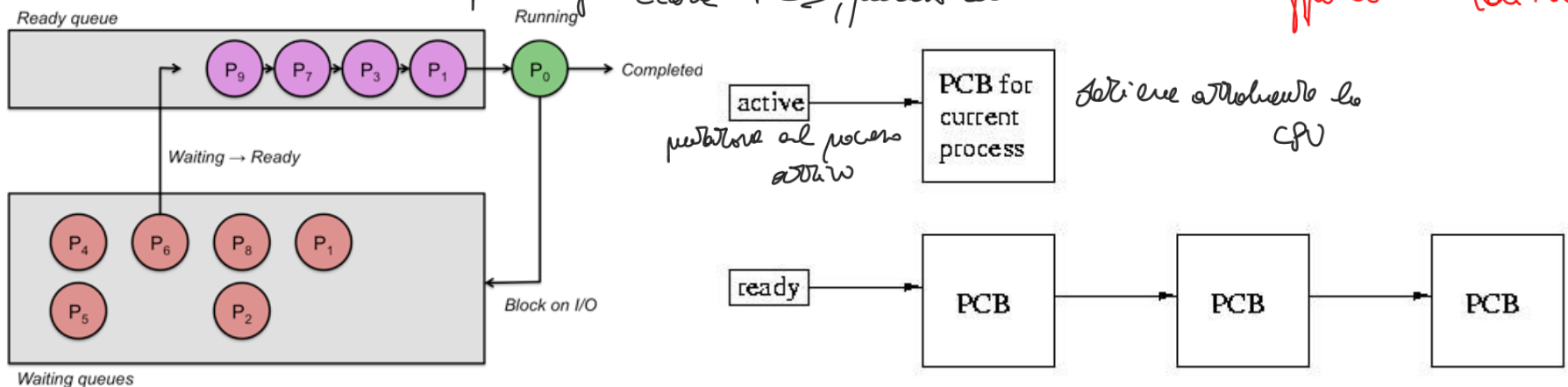
- Si occupa dei processi in CODA, di un evento I/O o*
- ❖ Lo scheduler della CPU gestisce la coda dei processi pronti, selezionando il prossimo processo cui verrà allocata la CPU
 - Gli elementi nella ready queue sono i PCB dei processi pronti
 - La ready queue può essere realizzata come una coda FIFO (first-in-first-out), una coda con priorità, una lista concatenata o un albero
 - La realizzazione (meccanismo) è spesso indipendente dalla politica implementata dallo scheduler

ORGANIZZAZ. della Ready Queue è un MECCANISMO

è un tipo di rappresentazione

In Meccanismi applico una Policy

per organizzazione PCB, process control block





Lo scheduler della CPU – 2

❖ Lo scheduler della CPU deve prendere una decisione quando un processo: *Quando dico Ready avere?*

1. passa da stato *running* a stato *wait* (es.: richiesta di I/O o attesa terminazione di un processo figlio)

2. passa da stato *running* a stato *ready* (interrupt) *da evento esterno* *es. possono la finire quando il tempo*

3. passa da stato *wait* a stato *ready* (es.: completamento di un I/O)

4. termina

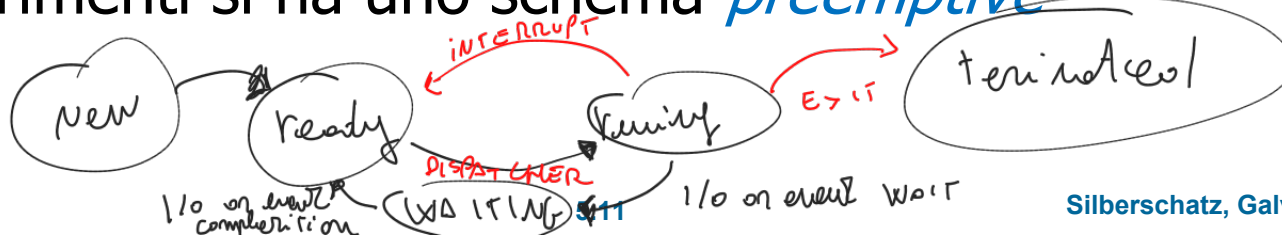
❖ Se lo scheduling viene effettuato solo nei casi 1 e 4, si dice che lo schema di scheduling è *nonpreemptive* (senza diritto di prelazione) o *cooperativo*

❖ Altrimenti si ha uno schema *preemptive*

*è una
Schella
dello
Scheduler*

*Cioè quando un processo entra in esecuzione, rilascia CPU solo quando lo decide lui
→ quando ha bisogno di una risorsa o finisce*

*DISPATCHER: modulo Scheduler
CON DIRITTO DI PREL. Scheduler Topico CPU per cambio contesto*





Lo scheduler della CPU – 3

❖ In caso di scheduler preemptive, sono aspetti critici da tenere in considerazione... *Problemi? Se interrupto forzatamente processo...*

- L'accesso a dati condivisi ^{con altri processi} (prelazione durante la modifica dei dati con risultati imprevedibili) *non lavorare dati in stato inconsistente interrupto va regolamentato*
- La possibilità di prelazione di processi del kernel (che normalmente modificano dati vitali per il sistema) *DISABLE O le INTERRUZIONI se*
- L'interruzione di operazioni cruciali effettuate dal sistema operativo (system call, inibizione delle interruzioni) *processo kernel*

*negli OS moderni, si disabilita per processi vitali... Per poco tempo però
potrei perdere proprio dati, catturare di interrupt*





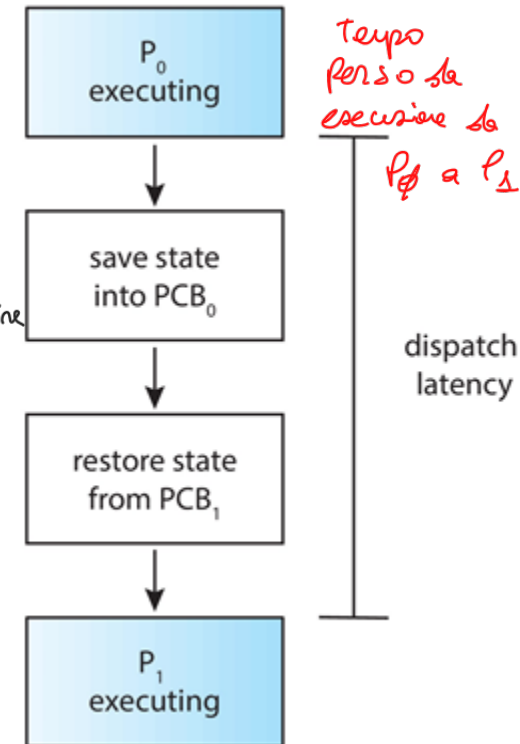
Il dispatcher

Modulo da Ready a Running

- ❖ Il modulo **dispatcher** passa il controllo della CPU al processo selezionato dallo scheduler; il dispatcher effettua:

- Context switch *① copia stato e registri nel PCB del processo interrotto*
- Passaggio a modo utente *② caricamento contenuto registri nel PCB del processo da eseguire*
- Salto alla posizione corretta del programma utente per riavvianne l'esecuzione *avvenire al PC salto alla prossima istruzione da eseguire*

- ❖ **Latenza di dispatch** — è il tempo impiegato dal dispatcher per sospendere un processo e avviare una nuova esecuzione *Latenza minima*





Context-switch

- ❖ Con quale frequenza avvengono i context-switch?
- ❖ In Linux, il comando `vmstat` fornisce il numero di cambi di contesto

```
$ vmstat 1 3
```

---cpu---

3 righe di output con numero medio di context-switch dall'avvio del sistema e nei due precedenti intervalli di tempo di 1 sec

```
243
225 }
339 }
```

frequenza media (al secondo)

frequenze durante i due secondi precedenti

rate ogni 1 sec

- ❖ È anche possibile usare il file system temporaneo `/proc`

Esempio

```
$ cat /proc/2166/status
voluntary_ctxt_switches
nonvoluntary_ctxt_switches
... ..
```

processo

interfaccia FS anche per info in RPM

/proc/2166 è in RPM

150

processo che controlla

senza pulsazione, NO. SD deve per forza intervenire

non può non intervenire, SO deve avere controllo





Criteri di scheduling

- ❖ ^{MASSIMIZZARE CPU} **Utilizzo della CPU** — la CPU deve essere più attiva possibile (comando **top** in Linux, Mac OS e UNIX)
- ❖ **Throughput** (produttività) — ^{# Processi Terminati nell'unità di tempo} numero dei processi che completano la loro esecuzione nell'unità di tempo ^{dipende anche dal tipo di Processi.}
- ❖ ^{Aa quanto o quanto ed è Terminato} **Tempo di turnaround** (tempo di completamento) — tempo impiegato per l'esecuzione di un determinato processo ^{Calcolo di tutti i Tempi}
 - Tempo di attesa nella ready queue e nelle code dei dispositivi, tempo effettivo di utilizzo della CPU e dei dispositivi di I/O
- ❖ **Tempo di attesa** — tempo passato dal processo in attesa nella ready queue ^{Tempo di attesa, solo su questo lo Scheduler può ottimizzare}
- ❖ **Tempo di risposta** — tempo che intercorre tra la ^{da esecuzione a risposta} sottomissione di un processo e la prima risposta prodotta ^{e il primo output prodotto}
 - Nei sistemi time-sharing, il tempo di turnaround può essere influenzato dalla velocità del dispositivo di output





Criteri di ottimizzazione – 1

- ❖ Massimo utilizzo della CPU
- ❖ Massimo throughput
- ❖ Minimo tempo di turnaround
- ❖ Minimo tempo di attesa
- ❖ Minimo tempo di risposta

} RIGUARDATA SOLO attesa nella
Ready Queue, non l'attesa dei
dispositivi

importante per sistemi interattivi





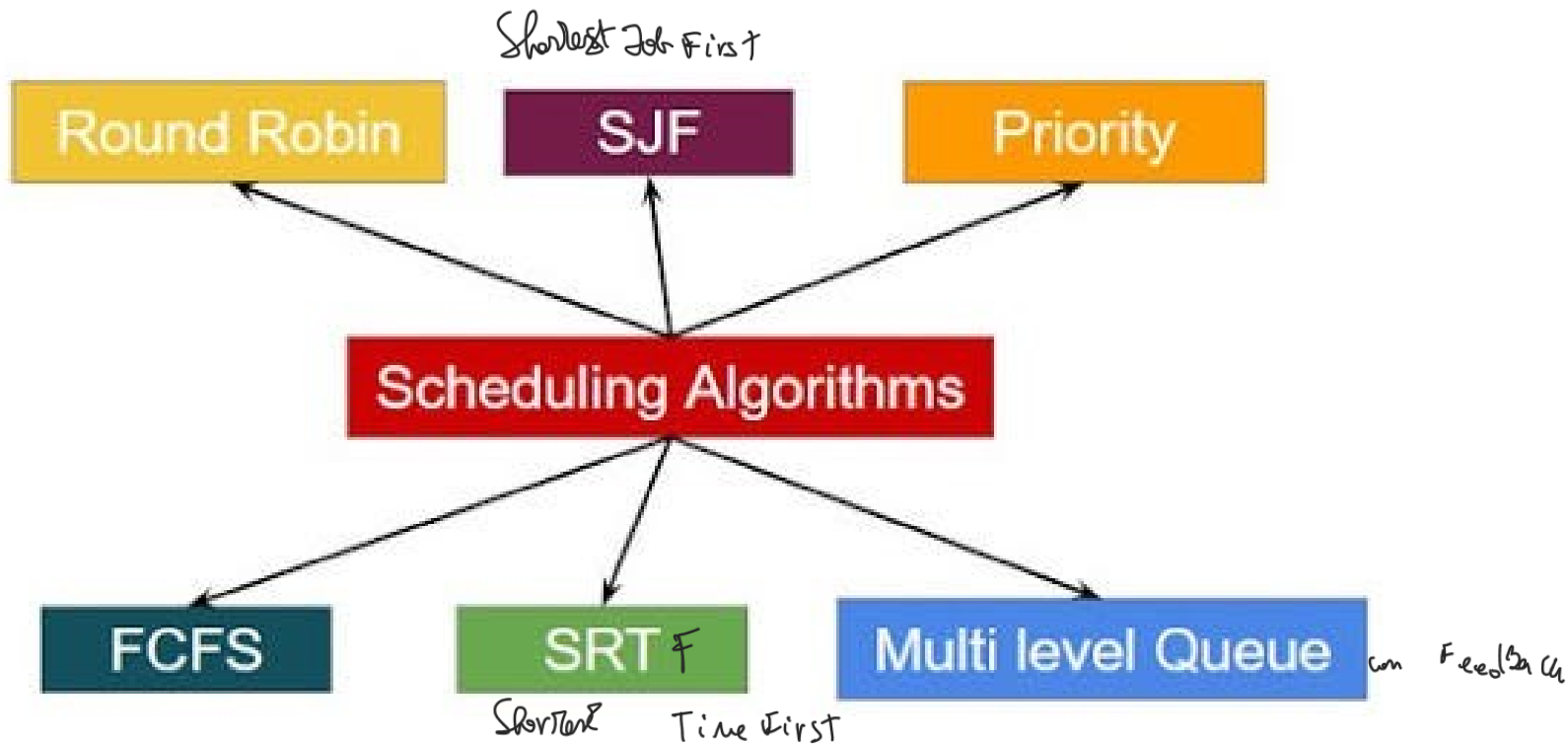
Criteri di ottimizzazione – 2

- ❖ Normalmente si ottimizzano i valori medi, ma talvolta è più opportuno ottimizzare i valori minimi/massimi
 - Per esempio, in sistemi interattivi (desktop), per garantire che tutti gli utenti ottengano un buon servizio
⇒ ridurre il tempo massimo di risposta
- ❖ Per i sistemi time-sharing è invece più significativo ridurre la **varianza** rispetto al tempo medio di risposta: un sistema con tempo di risposta prevedibile è migliore di un sistema mediamente più rapido, ma molto variabile





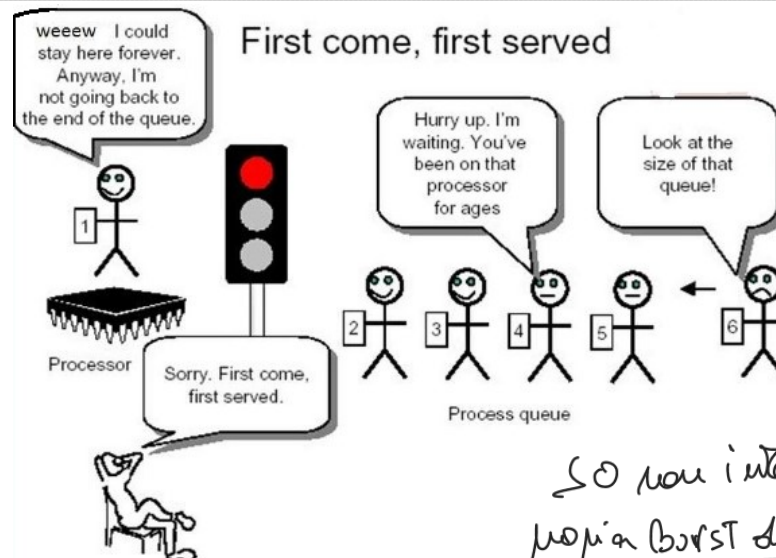
Algoritmi di scheduling





Scheduling First-Come-First-Served (FCFS)

- ❖ La CPU viene assegnata al processo che la richiede per primo: la realizzazione del criterio **FCFS** si basa sull'implementazione della ready queue per mezzo di una coda FIFO
- ❖ Quando un processo entra nella ready queue, si collega il suo PCB all'ultimo elemento della coda; quando la CPU è libera, viene assegnata al processo che si trova alla testa della ready queue, rimuovendolo da essa



no priorità, chi
è arrivato per
primo
non preemptive

Autonomia solo dopo che
ha finito burst

Lo non interviene, deve finire la
proprio burst di CPU

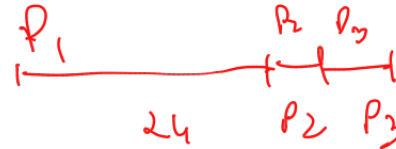
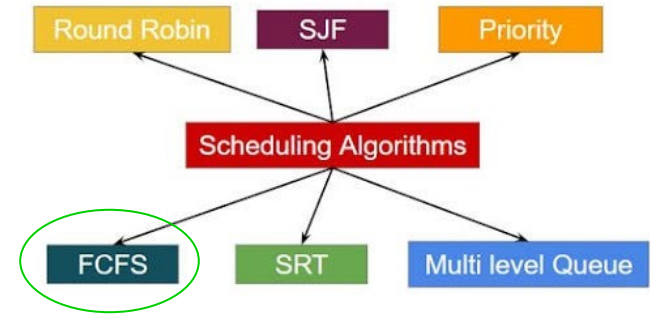




Scheduling FCFS (cont.)

❖ Esempio 1

Processo	Tempo di burst (millisecondi)
P_1	24
P_2	3
P_3	3



- I processi arrivano al sistema nell'ordine: P_1 , P_2 , P_3
- Per descrivere come si realizza lo scheduling si usa un **diagramma di Gantt**
 - ▶ Un istogramma che illustra una data pianificazione, inclu-
dendo i tempi di inizio e di fine di ogni processo





Scheduling FCFS (cont.)

Processo	Tempo di burst (millisecondi)
P ₁	24
P ₂	3
P ₃	3

- Il diagramma di Gantt per lo scheduling **FCFS** è:



P₁ non attende nulla

- Tempi di attesa in msec: P₁ → 0, P₂ → 24, P₃ → 27
- Tempo medio di attesa $T_a = (0 + 24 + 27) / 3 = \underline{17 \text{ msec}}$





Scheduling FCFS (cont.)

- Supponiamo che l'ordine di arrivo dei processi sia

P_2, P_3, P_1

- Il diagramma di Gantt diventa:

Processo	Tempo di burst (millisecondi)
P_1	24
P_2	3
P_3	3



- Tempi di attesa in msec: $P_1 \rightarrow 6, P_2 \rightarrow 0, P_3 \rightarrow 3$
- Tempo medio di attesa $T_a = (6+0+3)/3 = 3$ msec
- Molto inferiore al precedente: in questo caso, non si verifica l'effetto convoglio, per cui processi di breve durata devono attendere che un processo lungo liberi la CPU

Se ci crea,
non ho scelto
l'algoritmo
ottimale

Se processi corti vengono
eseguiti per primi si produce molto
di più di quello che si perde
avvicinando l'ordine dei processi
lungi

Faccio eseguire prima i processi corti

- Situazione in cui vi è un solo processo CPU-bound e molti processi I/O-bound

poche usate CPU
molti burst di piccola
durata

esce per lungo periodo la CPU





Scheduling Shortest–Job–First (SJF)

- ❖ Si associa a ciascun processo la lunghezza del suo burst di CPU successivo; si opera lo scheduling in base alla brevità dei CPU–burst
- ❖ Due schemi: 2 forme
↳ si manda in esecuzione e basta... si lascia finire
 - **non–preemptive** — dopo che la CPU è stata allocata al processo, non gli può essere prelazionata fino al termine del CPU burst corrente
 - **preemptive** — se arriva un nuovo processo con burst di CPU minore del tempo rimasto per il processo corrente, il nuovo processo prelazona la CPU: SRTF, Shortest–Remaining–Time–First
- ❖ **SJF** è *ottimo* — minimizza il tempo medio di attesa
 - Difficoltà nel reperire l'informazione richiesta

anche conoscere il tempo medio... difficile

Viene usato per lo scheduling a lungo termine, con solo 2013 alla volta, in cui il
prezioso nuovo spec. 5.2.0 a priori il tempo

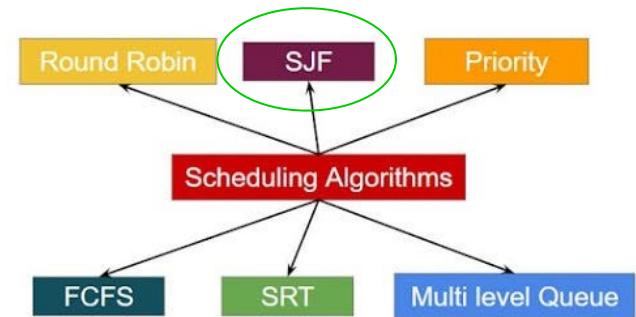




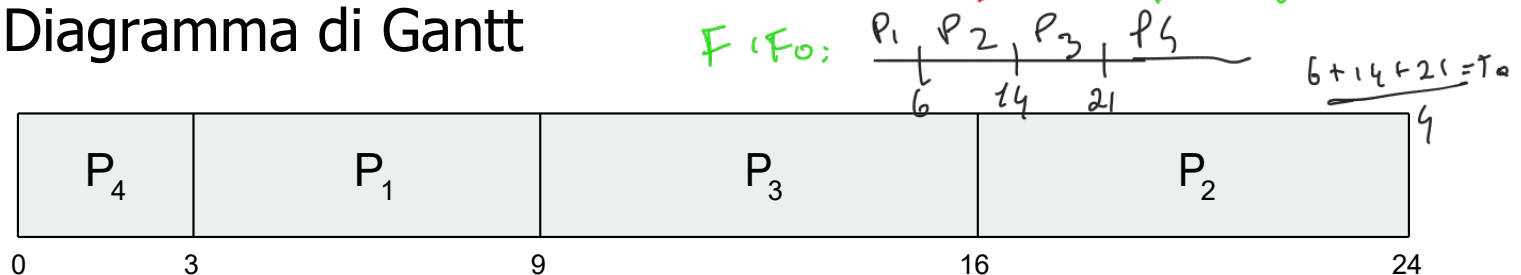
Scheduling SJF non-preemptive

❖ Esempio 2

Processo	Tempo di burst
P ₁	6
P ₂	8
P ₃	7
P ₄	3



■ Diagramma di Gantt



- Tempo medio di attesa $T_a = (3 + 16 + 9 + 0) / 4 = 7 \text{ msec}$
- Usando lo scheduling FCFS, $T_a = 10.25 \text{ msec}$





Scheduling SJF non-preemptive (cont.)

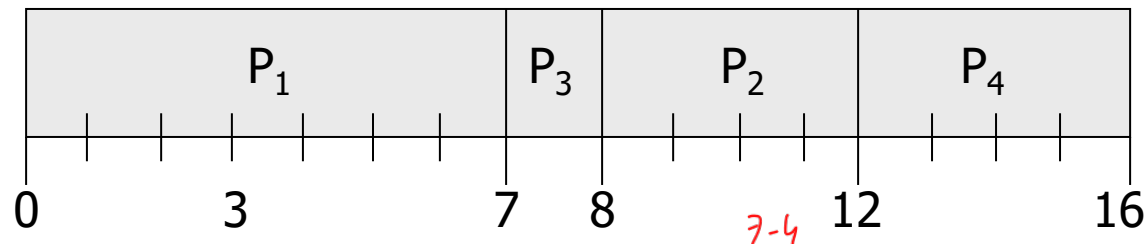
E' UN ALGORITMO a PRIORITA', possibile starvation per processi lunghi

❖ Esempio 3

Processo	Tempo di arrivo	Tempo di burst
P ₁	t 0	7
P ₂	<i>arriva di 2 che è più breve</i>	<i>non serve 4 volte</i>
P ₃	4	1
P ₄	5	4

*a t = 7, se si arriva
P₂*

■ Diagramma di Gantt



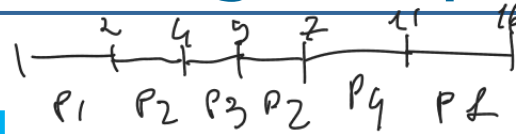
- Tempo medio di attesa $T_a = (0+6+3+7)/4 = 4\text{msec}$

8-2 *12-5* *tempo attesa - tempo arrivo*





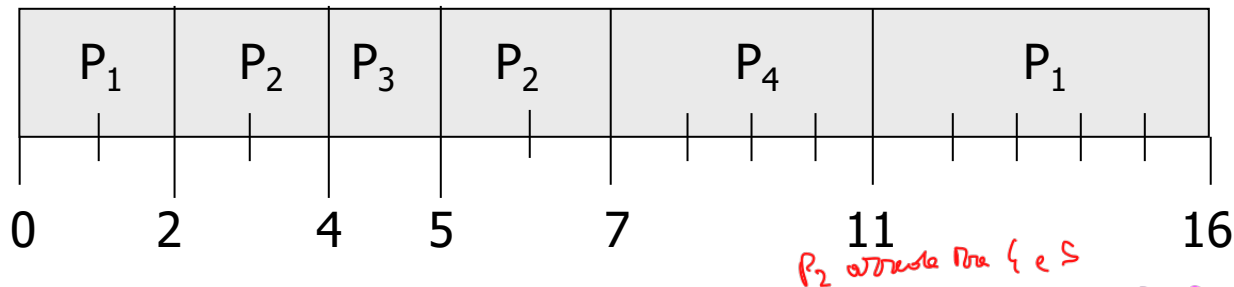
Scheduling SJF preemptive (SRTF)



❖ Esempio 4

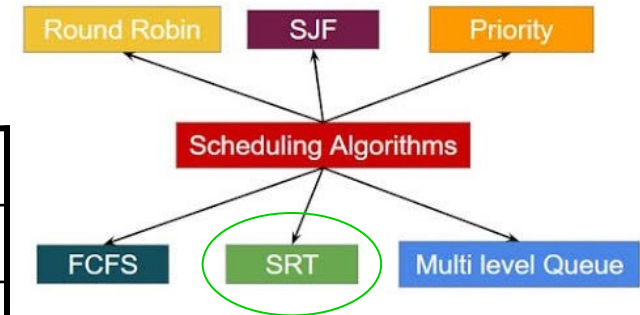
Processo	Tempo di arrivo	Tempo di burst
P ₁	0	7 5
P ₂	2 ₀₊₁	4 2
P ₃	4	1
P ₄	5	4

■ Diagramma di Gantt



■ Tempo medio di attesa $T_a = (9+1+0+2)/4 = 3\text{msec}$

*P₁ aspetta 9 sec
P₂ aspetta 1 sec
P₃ aspetta 0 sec
P₄ aspetta 2 sec*



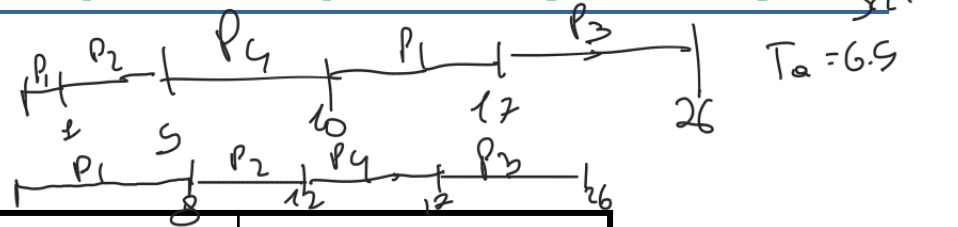
*t=0, P₁ c'è solo lui
t=2, P₁ (-5), P₂ arriva 4, entra lui
t=4, P₂ (-2), P₃ vuole 1, entra lui
t=5, li ha tutti, P₁ resta 5, ← 3
P₂ resta 2 ← lui
P₄ arriva 4 ← 2*





Scheduling SJF preemptive (SRTF)

❖ Esempio 5



Processo	Tempo di arrivo	Tempo di burst
P ₁	0	8 / 7
P ₂	1	4
P ₃	2	9
P ₄	3	5

$$0 + 7 + 9 + 15$$

$$T_a = \frac{31}{4} = 7.75$$

■ Diagramma di Gantt



■ Tempo medio di attesa

$$T_a = [(10-1) + (1-1) + (17-2) + (5-3)] / 4 = 26 / 4 = 6.5 \text{ msec}$$

■ Usando SJF non preemptive, $T_a = 7.75 \text{ msec}$





Lunghezza del successivo CPU-burst

- ❖ Può essere stimata come una media esponenziale delle lunghezze dei burst di CPU precedenti (impiegando una combinazione convessa)

di base di parametro α

Media in base ai pesi CPU burst dando meno peso ai burst vecchi

1. t_n = lunghezza dell' n -esimo CPU burst

2. τ_{n+1} = ^{← CPU burst al tempo $n+1$} valore stimato del prossimo CPU burst

Il maggiore importanza al campione recente

3. $0 \leq \alpha \leq 1$

Stima burst precedente

⇒ Si definisce

$$\tau_{n+1} = \alpha t_n + (1-\alpha)\tau_n$$

*$\alpha = 1$, stima peso = stima recente
 $\alpha = 0$, da peso allo storico passato*

τ_n *stima precedente* τ_1 = valore costante o stimato come media dei burst

t_n *burst recente* del sistema

⇒ Solitamente, si usa $\alpha = 1/2$ *da peso alla stima precedente e burst precedente*

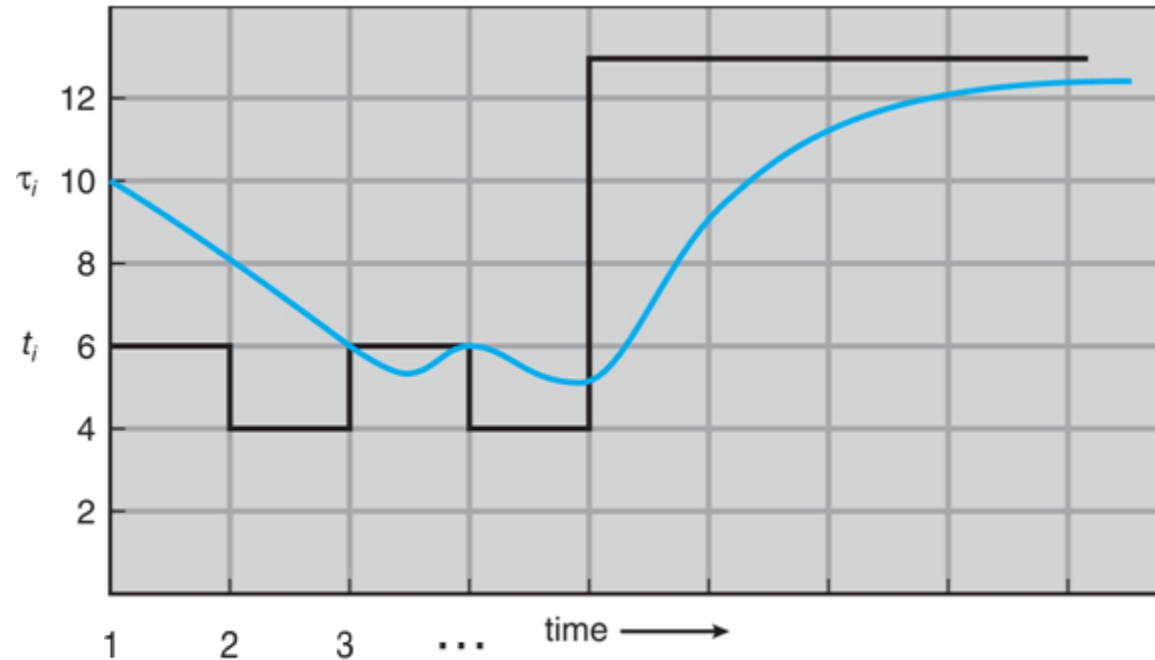
$$\tau_{n+1} = \alpha t_n + (1-\alpha)\tau_n$$

τ_n non ha ho... per fornire una media dei burst





Predizione della lunghezza del successivo CPU-burst



CPU burst (t_i) 6 4 6 4 13 13 13 ...
"guess" (τ_i) 10 → 8 → 6 → 6 ... 5 9 11 12 ...
 τ_1

$$\tau_2 = \alpha t_1 + (1-\alpha) \tau_1 \dots \text{in modo ricorsivo}$$
$$\tau_3 = \dots$$

α e $1-\alpha$, minori di uno
divergente sempre più piccolo





Esempi di calcolo – 1

$$\tau_{n+1} = \alpha t_n + (1-\alpha)\tau_n$$

❖ $\alpha = 0$

- $\tau_{n+1} = \tau_n$
- La storia recente non viene presa in considerazione: le condizioni attuali sono transitorie

❖ $\alpha = 1$

- $\tau_{n+1} = t_n$
- Si considera solo l'ultimo CPU burst: la storia passata è irrilevante





Esempi di calcolo – 2

$$\tau_{n+1} = \alpha t_n + (1-\alpha)\tau_n$$

❖ Espandendo la formula si ottiene:

$$\tau_2 = \alpha t_1 + (1-\alpha)\tau_1$$

$$\begin{aligned}\tau_3 &= \alpha t_2 + (1-\alpha)\tau_2 = \alpha t_2 + (1-\alpha)[\alpha t_1 + (1-\alpha)\tau_1] \\ &= \alpha t_2 + (1-\alpha)\alpha t_1 + (1-\alpha)^2 \tau_1\end{aligned}$$

...

$$\tau_{n+1} = \alpha t_n + (1-\alpha)\alpha t_{n-1} + \dots + (1-\alpha)^j \alpha t_{n-j} + \dots + (1-\alpha)^n \tau_1$$

serve più
piccolo
per me
che uno
avanti

❖ Poiché α e $(1-\alpha)$ sono entrambi minori o uguali ad 1, ciascun termine ha minor peso del suo predecessore





Scheduling a priorità

- ❖ Un valore di priorità (intero) viene associato a ciascun processo
- ❖ La CPU viene allocata al processo con la priorità più alta (spesso, intero più basso $\Rightarrow$ priorità più alta) negli esercizi verrà specificato
 - Preemptive
 - Non-preemptive
- ❖ SJF è uno scheduling a priorità in cui la priorità è calcolata in base al successivo tempo di burst
- ❖ Problema: *Starvation* ("inedia", blocco indefinito) – i processi a bassa priorità potrebbero non venir mai eseguiti
- ❖ Soluzione: *Aging* (invecchiamento) – aumento graduale della priorità dei processi che si trovano in attesa nel sistema da lungo tempo





Scheduling a priorità (cont.)



❖ Esempio 6

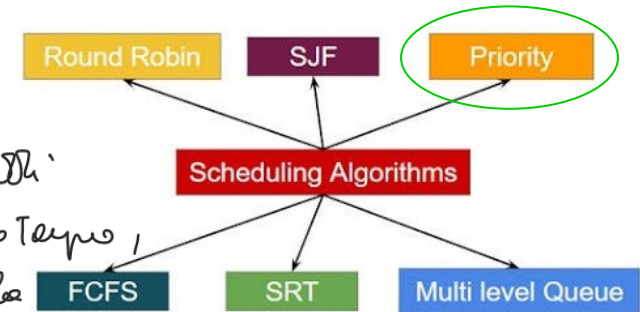
Processo	Tempo di burst	Priorità
P ₁	10	3
P₂	1	1
P ₃	2	4
P ₄	1	5
P₅	5	2

arrivano tutti
allo stesso tempo,
secondo la
priorità

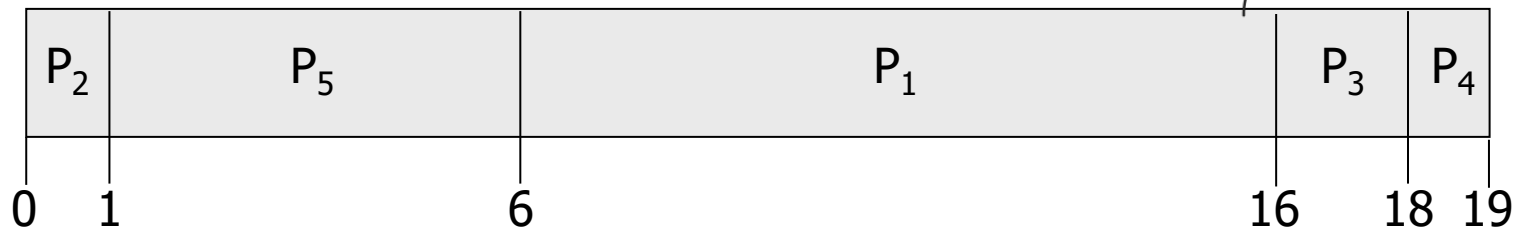
Priorità
massima

Se due processi hanno stessa priorità

uso una politica, parente o no



- Il diagramma di Gantt è:



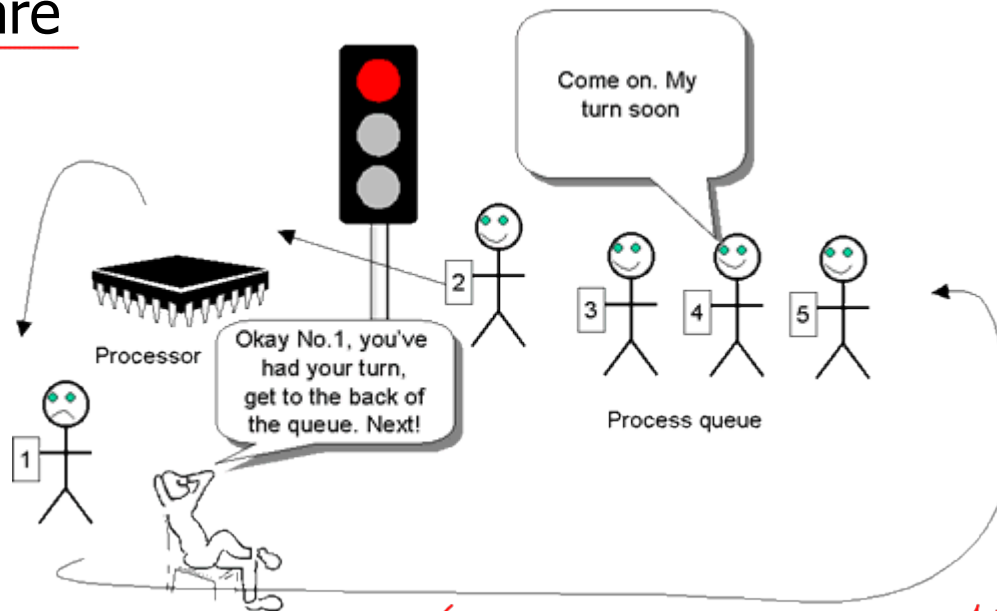
- Tempi di attesa: P₁→6, P₂→0, P₃→16, P₄→18, P₅→1
- Tempo medio di attesa $T_a = (6+0+16+18+1)/5 = 8.2\text{msec}$



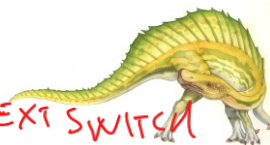


Scheduling Round Robin (RR) *non e' ottimo ma e' equo*

- ❖ A ciascun processo viene allocata una piccola quantità di tempo di CPU, un quanto di tempo (*time slice*), generalmente 10–100 millisecondi
 - Dopo un quanto di tempo, il processo è forzato a rilasciare la CPU e viene accodato alla ready queue
 - La ready queue è trattata come una coda (FIFO) circolare



più SIGNIFICATIVO il tempo di CONTEXT SWITCH





Scheduling RR (cont.)

tempo per

- ❖ Se vi sono n processi nella ready queue ed il quanto di tempo è q , ciascun processo occupa $1/n$ del tempo di CPU in frazioni di, al più, q unità di tempo; nessun processo attende per più di $(n-1) \times q$ unità di tempo
- ❖ Il timer interrompe la CPU allo scadere del quanto per programmare il prossimo processo
- ❖ Prestazioni: *5 processi, 2ms, al più 8ms prima di avere servizio*
 - q grande $\Rightarrow$ FCFS
 - q piccolo $\Rightarrow q$ deve essere grande rispetto al tempo di context switch (che, tuttavia, è normalmente $< 10\mu\text{sec}$), altrimenti l'overhead è troppo alto

*Come scelgo
q*

*Se tutti i processi hanno q,
sarebbe una FCFS*

REGOLA DEL POLICE

*Se 1/ dei processi deve finire nel quanto
20% b.*

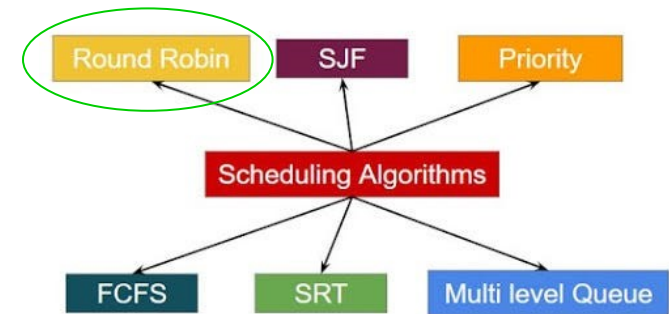




Scheduling RR (cont.)

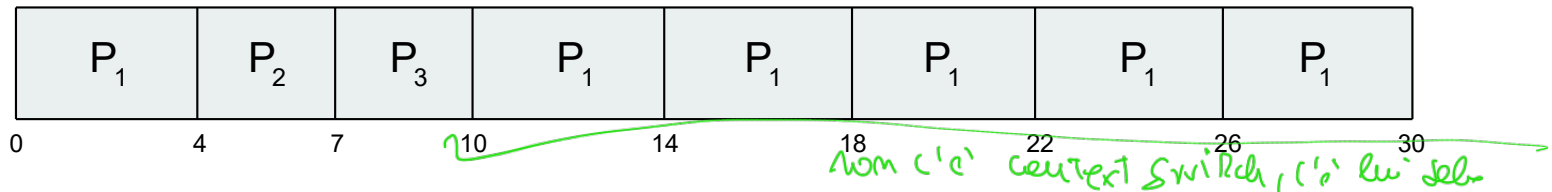
❖ Esempio 7

Processo	Tempo di burst
P ₁	24
P ₂	3
P ₃	3



P₁ non sta nel queue

- Il diagramma di Gantt con $q = 4$ è:



- In genere si ha un tempo medio di attesa e di turnaround maggiore rispetto a SJF, tuttavia si ottiene un miglior tempo medio di risposta
- Tempi di attesa: *da 4 a 10* P₁ → 6, P₂ → 4, P₃ → 7
- Tempo medio di attesa $T_a = (6+4+7)/3 = 5.6\text{msec}$

peggio di SJF

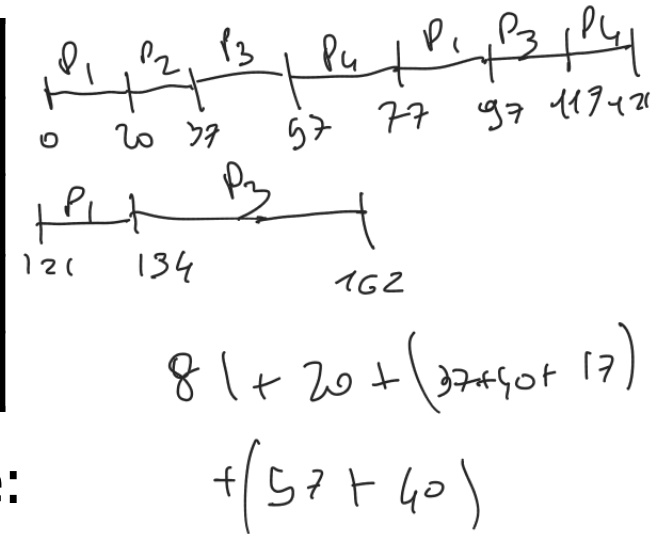




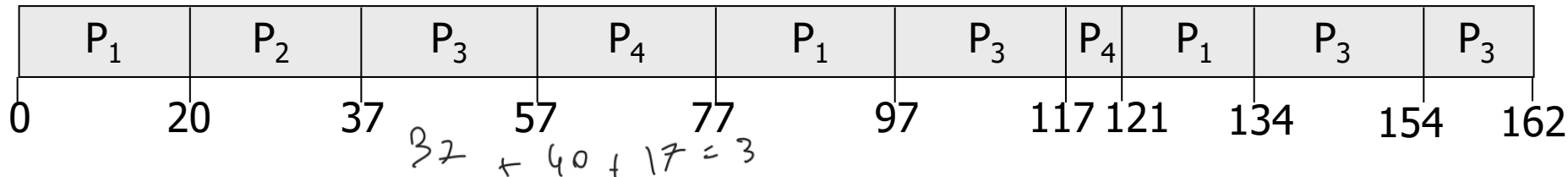
Scheduling RR (cont.)

❖ Esempio 8

Processo	Tempo di burst
P ₁	53 38 13
P ₂	17
P ₃	68 48 20
P ₄	24 4



- Il diagramma di Gantt con $q = 20$ è:

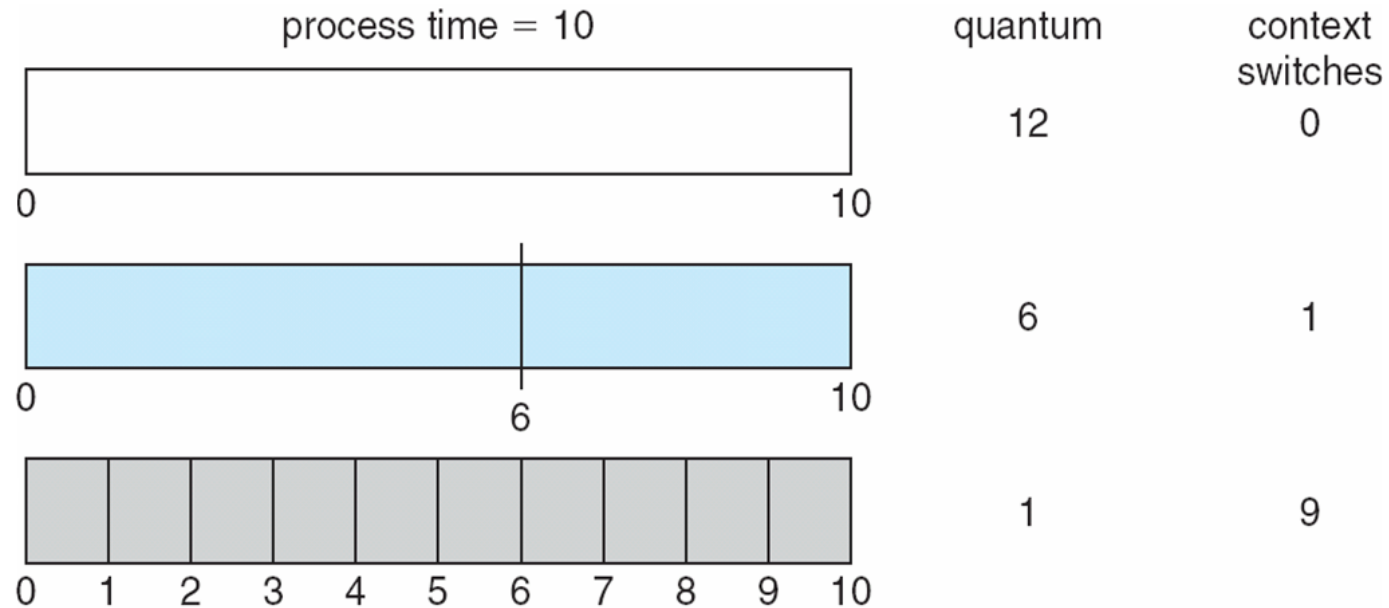


- Tempi di attesa: P₁ → 81, P₂ → 20, P₃ → 94, P₄ → 97
- Tempo medio di attesa $T_a = (81 + 20 + 94 + 97) / 4 = 73 \text{ msec}$





Quanto di tempo e tempo di context-switch

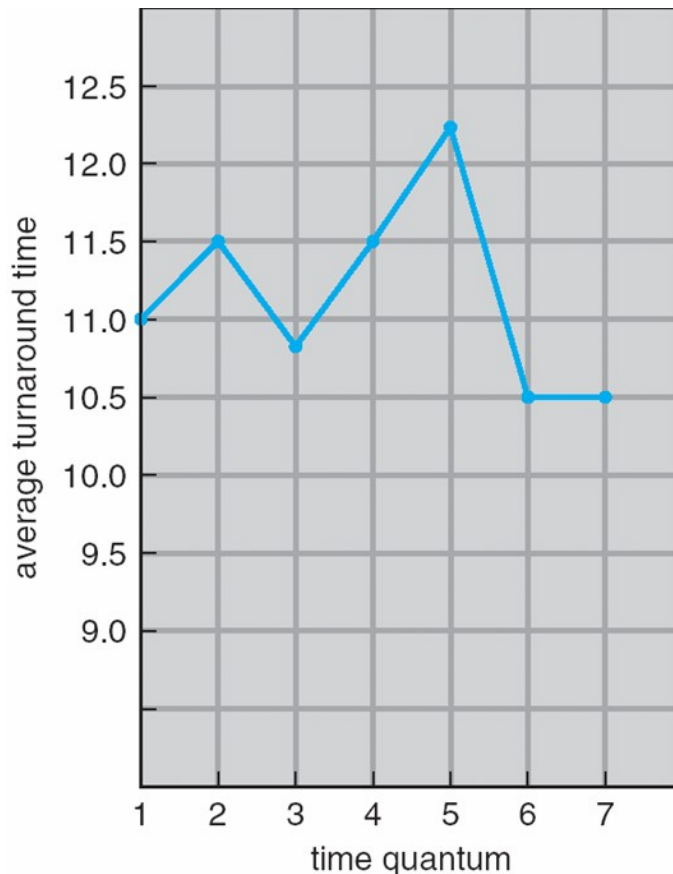


Un quanto di tempo minore incrementa il numero di context switch





Quanto di tempo e tempo di turnaround



process	time
P_1	6
P_2	3
P_3	1
P_4	7

Empiricamente: il quanto di tempo deve essere più lungo dell'80% dei CPU burst

Variazione del tempo medio di turnaround in funzione della lunghezza del quanto di tempo

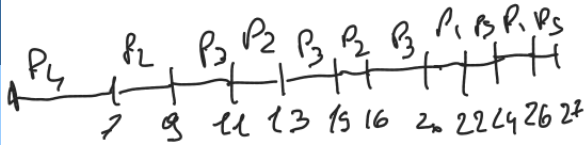




Scheduling a priorità con RR

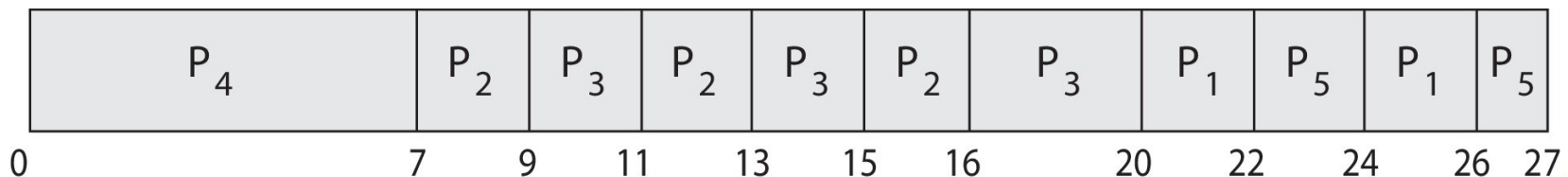
❖ Esempio 9

Scheduling a priorità
si può fare
RR



Processo	Tempo di burst	Priorità
P ₁	4	3
P ₂	5	2
P ₃	8	2
P ₄	7	1
P ₅	3	3

- Viene eseguito il processo a priorità maggiore; a parità di priorità, si usa RR
- Il diagramma di Gantt con $q = 2$ è:





Scheduling con code multiple – 1

- ❖ Nello scheduling a priorità tutti i processi possono essere collocati in una singola coda, da cui lo scheduler seleziona il prossimo processo da eseguire

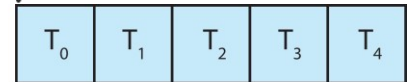
- La ricerca può costare $\mathcal{O}(n)$ *quale prossimo processo?*

- ❖ Nella pratica, risulta più conveniente organizzare i processi in code distinte

- Lo scheduler assegna la CPU al processo "in testa" alla coda a priorità più alta

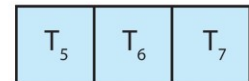
più code con diversa priorità

facile lavoro scheduler



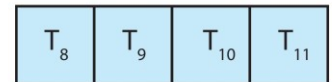
- ❖ L'approccio funziona bene anche per lo scheduling combinato con RR

priority = 1



- Se ci sono più processi in una coda con una data priorità, questi verranno eseguiti con scheduling circolare

priority = 2

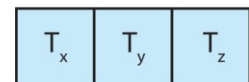


•
•
•

internamente uso priority

Se si suona coda $\emptyset$, passo a coda 1 ma Se arriva prima coda $\emptyset$, fermo coda 1, coda con prio.

priority = n





Scheduling con code multiple – 2

❖ Esempio 10

- La ready queue è suddivisa nelle due code
 - ▶ foreground (interattiva) *non interattiva*
 - ▶ background (batch) *in background, possono occupare tutto CPU se no interagiscono*
- Ogni coda ha il suo proprio algoritmo di scheduling
 - ▶ foreground: RR *← sono I/O BOUND*
 - ▶ background: FCFS *← CPU BOUND uso CPU per lunghi periodi*
- È necessario effettuare lo scheduling tra le code
 - ▶ *Scheduling a priorità fissa*: si servono tutti i processi foreground poi quelli background
 - ⇒ Rischio di starvation *Code STATICHE non c'è priorità dinamica*
 - ▶ *Time slice*: ciascuna coda occupa un certo tempo di CPU che suddivide fra i propri processi; ad esempio...
 - 80% per foreground con RR
 - 20% per background con FCFS *evita la starvation*

80% CPU per priorità più alta

20% per priorità più bassa

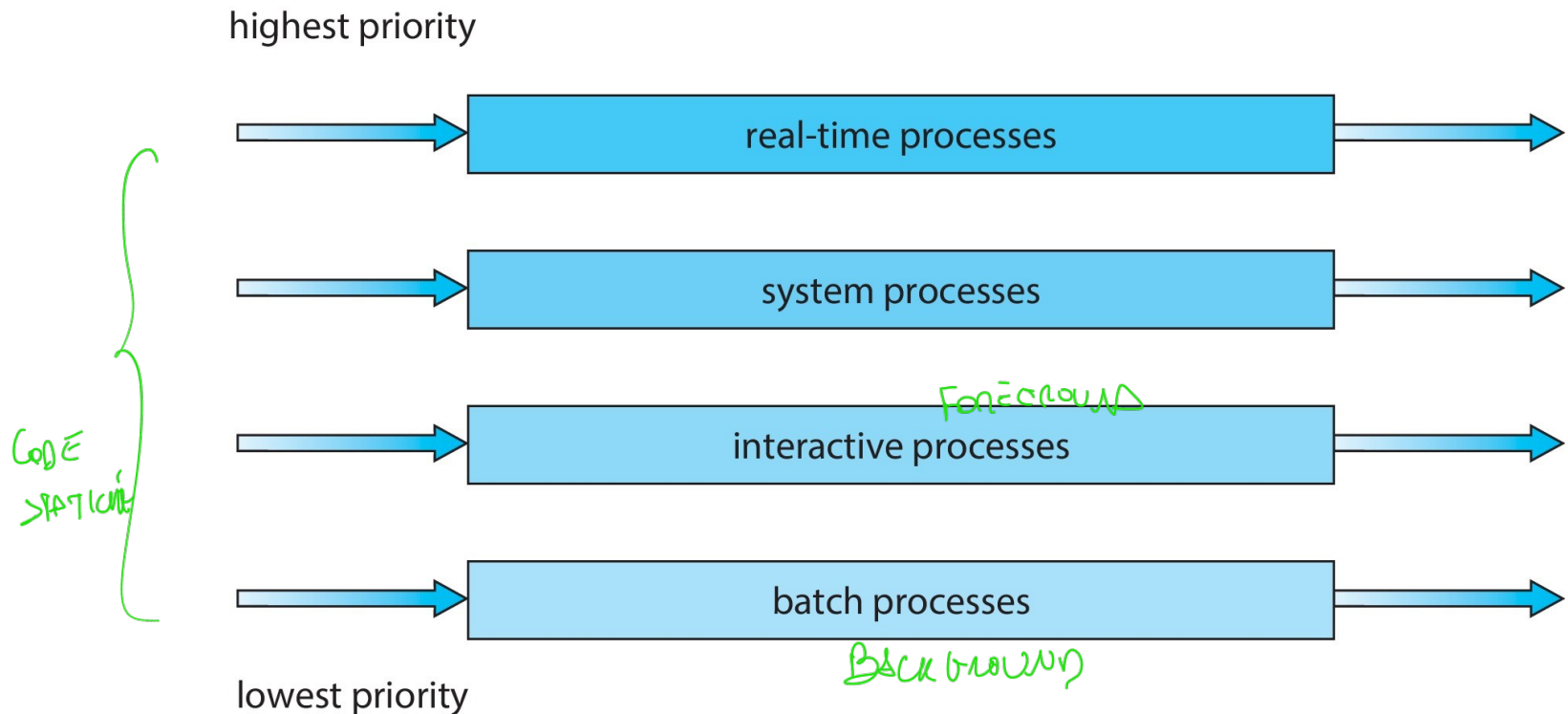




Scheduling con code multiple – 3

- ❖ I processi si assegnano in modo permanente ad una coda, generalmente secondo qualche caratteristica (invariante) del processo

*in realtà' sono ovvero guardando
processo nella coda, RUVAWTAZ. MOUNT A
MAGGIO*



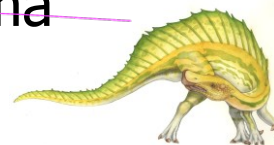


Code multiple con feedback – 1

NON PIÙ SITUAZ. STATICA

opp'osto al non-come

- ❖ Se la priorità viene ricalcolata dinamicamente in base al comportamento dei processi... *durante l'esecuzione*
- ❖ I processi possono spostarsi fra le varie code; *sia scelto lo schedato* si può implementare l'aging *→ possono a loro volta fare nella gerarchia delle cose*
- ❖ Lo scheduler a code multiple con feedback è definito dai seguenti parametri:
 - Numero di code
 - Algoritmo di scheduling per ciascuna coda
 - Metodo impiegato per determinare quando spostare un processo in una coda a priorità maggiore *Come faccio a scegliere un processo nella coda?*
 - Metodo impiegato per determinare quando spostare un processo in una coda a priorità minore
 - Metodo impiegato per determinare in quale coda deve essere posto un processo quando entra nel sistema *in quale coda metter processo?*





Code multiple con feedback – 2

❖ Esempio 11

■ Tre code:

- ▶ Q_0 – RR con quanto di tempo di 8 millisecondi
- ▶ Q_1 – RR con quanto di tempo di 16 millisecondi
- ▶ Q_2 – FCFS *a più bassa priorità*

es. per politica, tutti i processi nuovi a priorità 0

❖ Scheduling

- Un nuovo processo viene immesso nella coda Q_0 , che è servita con RR; quando prende possesso della CPU il processo riceve 8 millisecondi; se non termina, viene spostato nella coda Q_1
- Nella coda Q_1 il processo è ancora servito RR e riceve ulteriori 16 millisecondi; se ancora non ha terminato, viene spostato nella coda Q_2 , dove verrà servito con criterio FCFS all'interno dei cicli di CPU lasciati liberi dai processi delle code Q_0 e Q_1

Se BURST > 8 ms, code

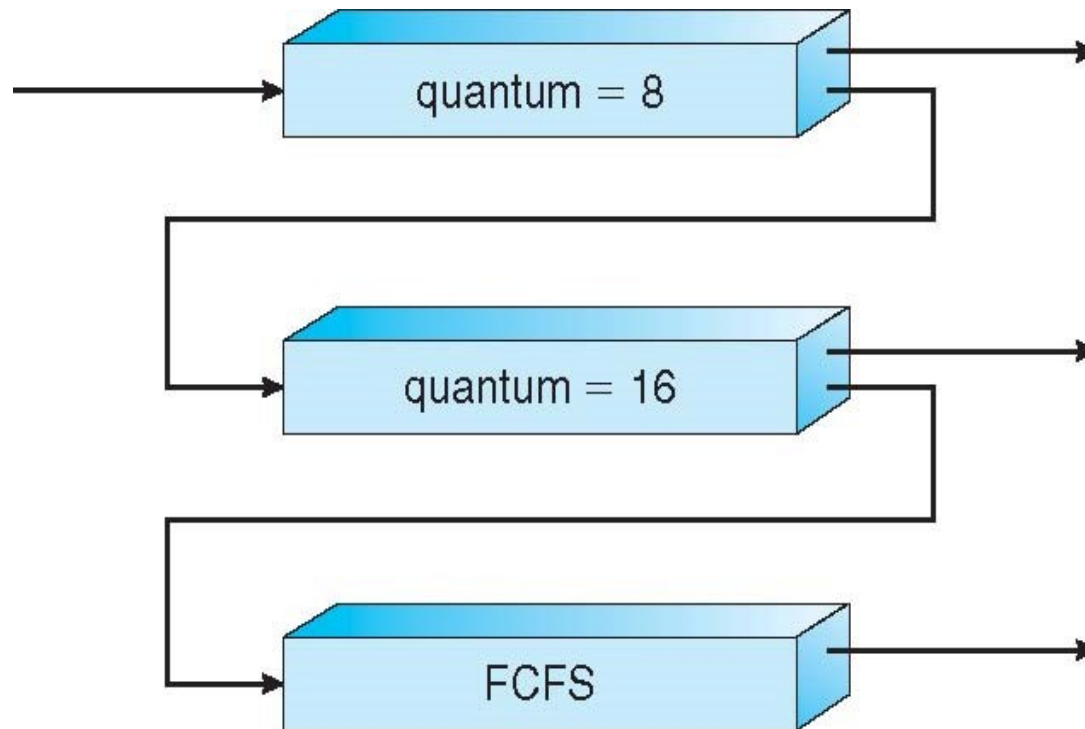
*Scende solo per CPU burst.
SO*

*Si scende e basta!
nella coda!*





Code multiple con feedback – 3



- ❖ L'idea sottesa all'algoritmo di scheduling è quella di separare i processi in base alle loro caratteristiche d'uso della CPU

⇒ Massima priorità con CPU-burst brevi

IMPLEMENTO AGING per garantire anche che CPU-burst venga servito

• livello 2 e 3

• minima "pulizia" interattiva





Esempio 1

compos. delle Tecniche precedenti

❖ Esercizio

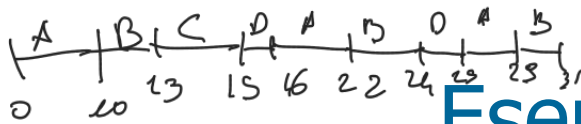
Un processore può avere più burst.

Un insieme di task indipendenti, A, B, C e D, deve essere eseguito su di un unico processore. I task possono venire elaborati più di una volta. Appena un task ha terminato la propria esecuzione, è pronto per la successiva (che non necessariamente durerà lo stesso tempo). I tempi di arrivo e di esecuzione sono descritti nella seguente tabella:

Task	Tempo di arrivo	1° esecuzione	2° esecuzione	3° esecuzione
A	0	10 Tempo Burst	6	4
B	2	3	2	2
C	3	2	—	—
D	5	1	1	—

Ci sarà effetto convoglio. A è il più lungo di tutti, gli altri più brevi e poi
Si tracci il diagramma di Gantt per lo scheduling FCFS e si calcoli il tempo medio di attesa. Lo scheduling generato soffre dell'effetto convoglio? Motivare la risposta data.



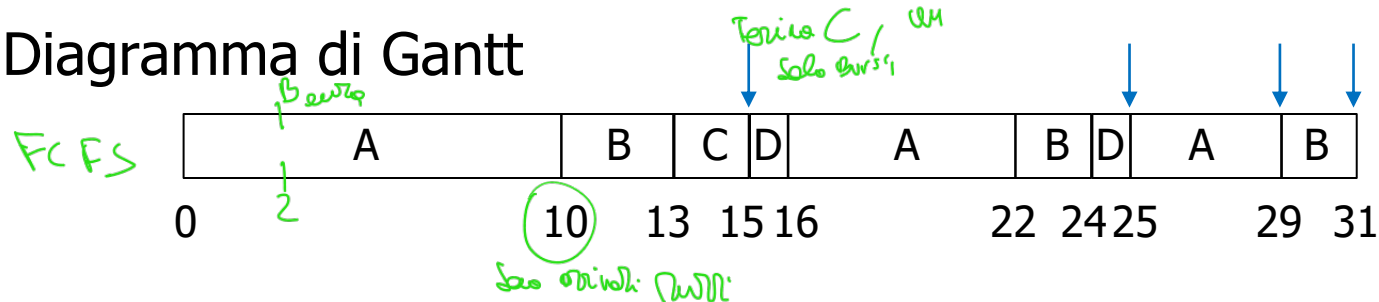


Esempio 1

❖ Soluzione

Task	Tempo di arrivo	1° esecuzione	2° esecuzione	3° esecuzione
A	0	10	6	4
B	2	3	2	2
C	3	2	–	–
D	5	1	1	–

Diagramma di Gantt



$$T_a(A)=9, T_a(B)=22, T_a(C)=10, T_a(D)=18$$

$$T_a = (9+22+10+18)/4 = 14.75 \text{ msec}$$

Si verifica l'effetto convoglio perché il processo A, che viene sempre eseguito per primo, ha burst significativamente più lunghi di tutti gli altri processi.





Esempio 2

De Lorenzo J

❖ Esercizio

Si considerino i processi P_1 , P_2 e P_3 . Ciascun processo esegue un CPU-burst ed un I/O-burst, quindi nuovamente un CPU-burst ed un I/O-burst ed infine un ultimo CPU-burst. La lunghezza dei burst ed il tempo di arrivo dei processi (in millisecondi) è riportato in tabella:

Processo	Burst1	I/O_1	Burst2	I/O_2	Burst3	Arrivo
P_1	2	4	2	2	2	0
P_2	2	2	3	3	1	1
P_3	1	2	1	1	1	1

Si disegnino i diagrammi di Gantt che illustrano l'esecuzione dei tre processi utilizzando FCFS ed RR con quanto di tempo pari a 2. Se il termine di un servizio di I/O ed un timeout della CPU si verificano nello stesso istante, si assegni la precedenza al processo che ha appena terminato il proprio I/O. Si calcoli il tempo medio di attesa ed il tempo medio di turnaround nei due casi.

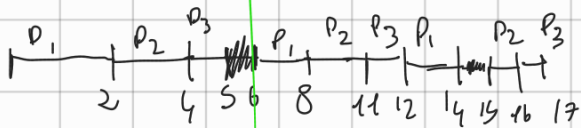
Attenzione: bisogna l/0 e' uno solo, si mette in coda FCFS



FCFS

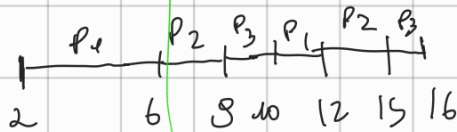
NON ATTEDE P₁

Ready Queue



Tempo attesa, solo quando c'è processo

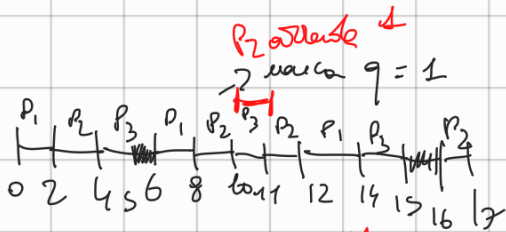
I/O Queue



$$T_g = \frac{0 + 1 + (3 + 1)}{3} = 1,6 \mu s$$

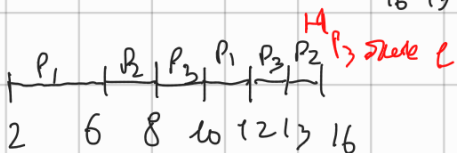
$$T_r = \frac{14 + 15 + 16}{3} = 15 \mu s$$

RR



$$T_a = \frac{0 + (1 + 1) + (3 + 1)}{3} = 2 \mu s$$

I/O Queue



$$\frac{14 + 16 + 14}{3} = T_r = 14,6 \mu s$$



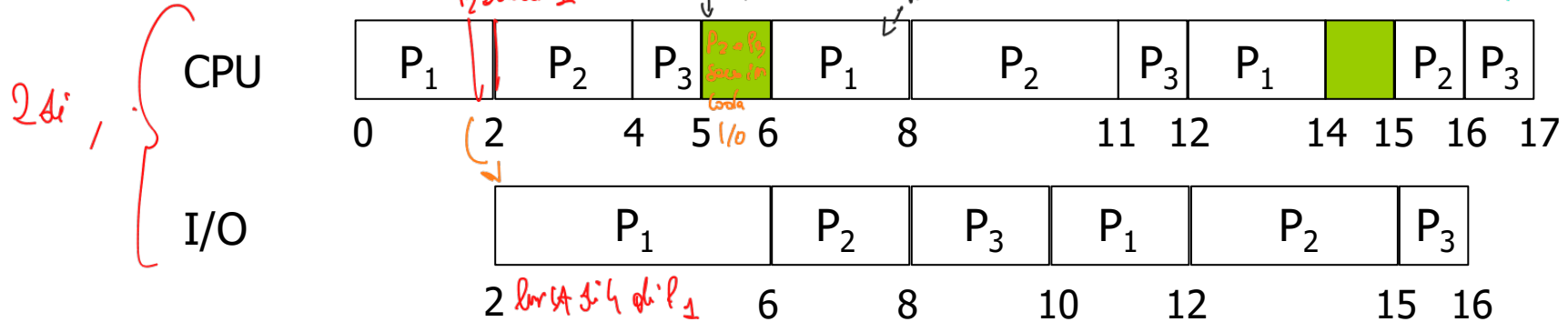
Esempio 2

❖ Soluzione FCFS

Diagrammi di Gantt

Processo	Burst1	I/O_1	Burst2	I/O_2	Burst3	Arrivo
P ₁	2	4	2	2	2	0
P ₂	2	2	3	3	1	1
P ₃	1	2	1	1	1	1

P₁ non ASPETTA 11P1!



CPU inutilizzata

$$\%T_{CPU} = \frac{17 - 15}{17}$$

$$T_a P_1 = 14 \quad (\text{arrivo a 0, Fine a 14})$$

$$P_2 = 15 \quad (\text{Fine (6)})$$

$$16 \quad (\text{Fine (7) arrivo a 1})$$

$$T_a = (0 + 1 + 4) / 3 = 1.6 \text{ msec}$$

$$T_t = (14 + 15 + 16) / 3 = 15 \text{ msec}$$

❖ Nota: Nel tempo di attesa *Tempo attesa I/O non conta come tempo attesa* deve essere considerato esclusivamente il tempo trascorso dai processi nella ready queue (senza considerare il tempo di attesa/servizio di I/O)

Tempo in cui i processi attendono avendo pronti

Se molti processi "TANTI I/O quanto sono", I/O vanno in parallelo





Esempio 2

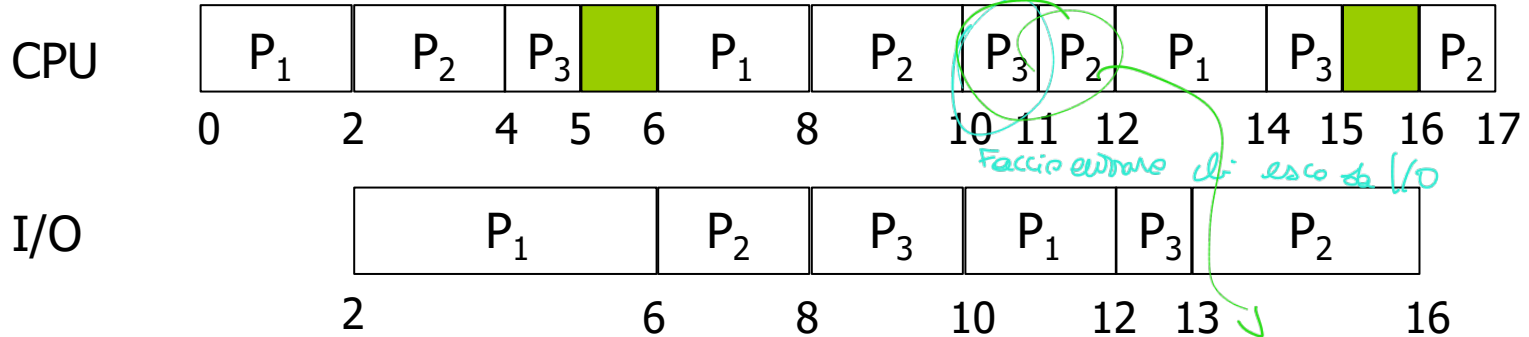
RR solo per CPU
Per I/O segue FCFS

❖ Soluzione RR, $q=2$

Diagrammi di Gantt

Processo	Burst1	I/O_1	Burst2	I/O_2	Burst3	Arrivo
P ₁	2	4	2	2	2	0
P ₂	2	2	3	3	1	1
P ₃	1	2	1	1	1	1

BURST < del quantum di tempo, 1/2 = FCFS nel primo burst



CPU inutilizzata

$$T_a = (0+2+4)/3 = 2\text{msec}$$

$$T_t = (14+16+14)/3 = 14.\bar{6}\text{msec}$$

Tononard

no cen
sta di più tempo
nella ready queue
no finisco primo





Esempio 3

❖ Esercizio

Si considerino 5 processi (A,B,C,D,E) tali che:

Job	Tempo di arrivo	CPU burst	Priorità
A	0	8 1	2 1
B	1	6	3
C	2	3	2
D	3	4	2
E	4	3	2

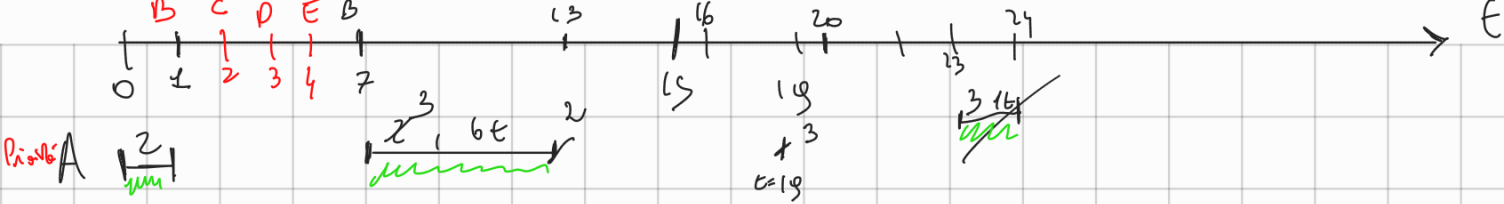
Priorità
massima

classi priority

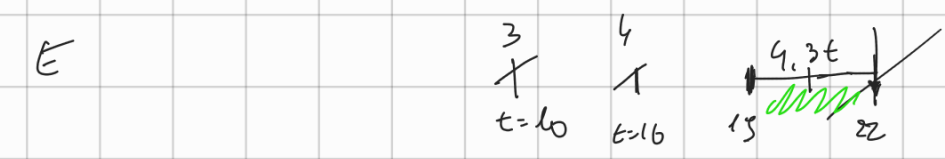
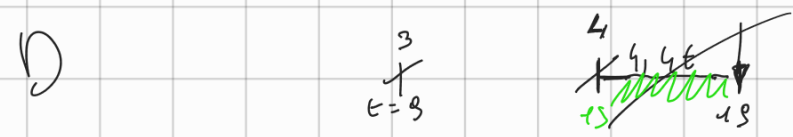
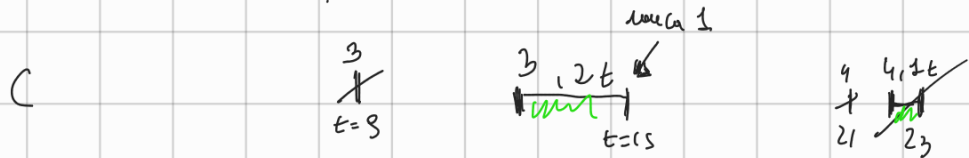
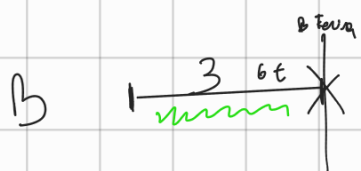
Descrivere la sequenza di esecuzione dei job (tramite Gantt chart) e calcolare il tempo totale di completamento ottenuto con uno scheduling a code multiple a priorità e feedback, dove ogni coda è gestita con la strategia FCFS. Il feedback è definito come segue: la priorità diminuisce di 1 (fino al livello base 1) se si passano più di 6 unità di tempo consecutive in esecuzione nella CPU ed aumenta di 1 per ogni 6 unità di tempo passate in attesa in una qualsiasi coda.

si siede se si va nella CPU





Problem A



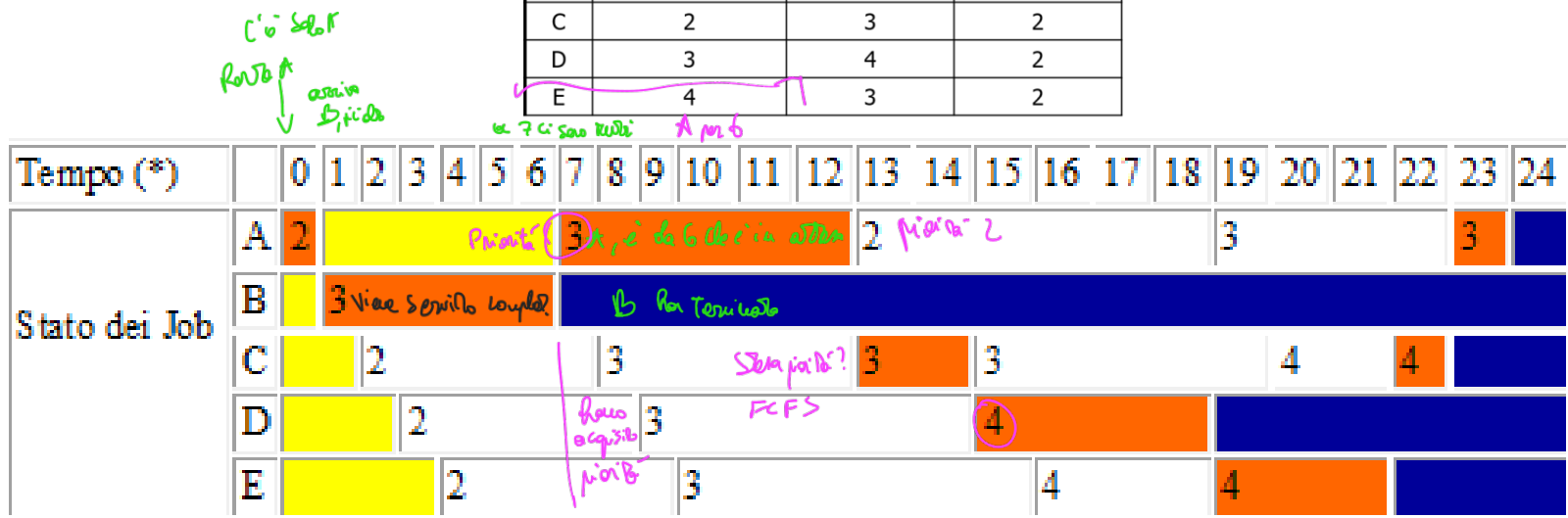
Force over time...



Esempio 3

❖ Soluzione

Job	Tempo di arrivo	CPU burst	Priorità
A	0	8	2
B	1	6	3
C	2	3	2
D	3	4	2
E	4	3	2



❖ Istante di completamento:

A→24, B→7, C→23, D→19, E→22

- ❖ Nota: La preemption avviene al tempo 13 – C sostituisce A – e al 15 – D sostituisce C. Al tempo 19, la coda di priorità 4 contiene solo E, mentre C entra in coda al tempo 20





Scheduling dei thread – 1

❖ Esiste una distinzione fondamentale fra lo scheduling dei thread a livello utente e a livello kernel ma, certamente, se i thread sono supportati dal sistema, lo scheduling avviene a livello di thread

❖ **Scheduling locale:** nei modelli multi-a-uno e multi-a-molti, è l'algoritmo con cui la libreria dei thread decide quale thread (a livello utente) allocare a un LWP disponibile (*ambito della competizione ristretto al processo, PCS*) – normalmente un algoritmo a priorità con prelazione definito dal programmatore

■ La libreria dei thread pianifica l'esecuzione su LWP

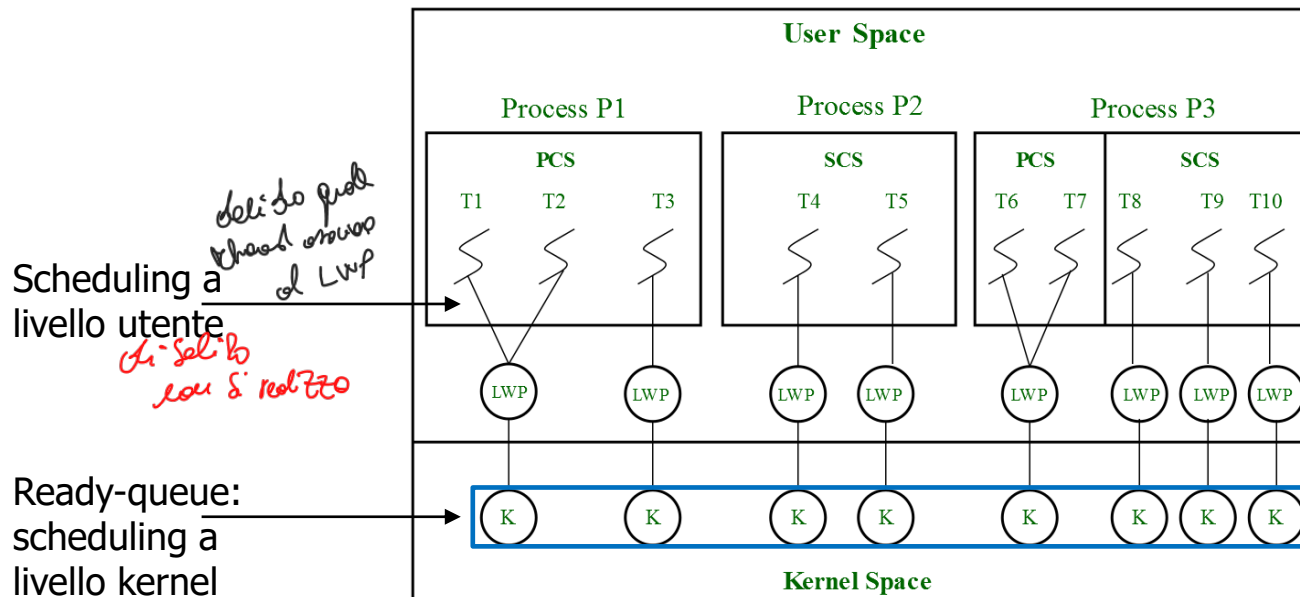
■ Tuttavia, il thread prescelto non è effettivamente in esecuzione su una CPU fisica finché il SO non pianifica l'esecuzione del thread a livello kernel corrispondente





Scheduling dei thread – 2

- ❖ **Scheduling globale:** *Modello 1:1 a livello kernel*
l'algoritmo con cui il kernel decide quale thread a livello kernel dovrà venire eseguito successivamente (*ambito di competizione allargato al sistema, SCS*)
più thread che competono, poichè da processi diversi
- ❖ I sistemi operativi caratterizzati dall'impiego del modello uno-a-uno (quali Windows, Linux e Mac OS) pianificano i thread unicamente con scheduling globale





API Pthreads per lo scheduling dei thread – 1

LIBRERIA

- ❖ La API Pthreads di POSIX consente di specificare **PCS** (Process–Contention–Scope) o **SCS** (System–Contention–Scope) nella fase di generazione dei thread
- ❖ Per specificare l'ambito della contesa, Pthreads usa i valori: *due costanti, definite a livello viene effettuato lo scheduling*
 - PTHREAD_SCOPE_PROCESS**: scheduling PCS
 - PTHREAD_SCOPE_SYSTEM**: scheduling SCS *LINUX*
- ❖ Nei sistemi con modello da molti–a–molti:
 - La politica **PTHREAD_SCOPE_PROCESS** pianifica i thread a livello utente sugli LWP disponibili
 - La politica **PTHREAD_SCOPE_SYSTEM** crea, in corrispondenza del thread a livello utente, un LWP ad esso vincolato, realizzando in effetti una corrispondenza secondo il modello da uno–a–uno





API Pthreads per lo scheduling dei thread – 2

Programma che determina l'ambito della contesa ^{attesa} in vigore e lo imposta a `PTHREAD_SCOPE_SYSTEM`; si creano quindi 5 thread distinti in esecuzione con modello SCS

```
#include <pthread.h>
#include <stdio.h>
#define NUM_THREADS 5 costante
```

```
int main(int argc, char *argv[])
```

```
{
```

```
    int i, scope; 7 vettore che contiene l'ioi dei thread
```

```
    pthread_t tid[NUM_THREADS];
```

```
    pthread_attr_t attr;
```

```
    /* Ottiene gli attributi di default dei thread */
```

```
    pthread_attr_init(&attr);
```

```
    /* Per prima cosa, appura l'ambito della contesa*/
```

```
    if (pthread_attr_getscope(&attr, &scope) != 0) quel e' l'ambito della contesa
```

```
        fprintf(stderr, "Impossibile stabilire l'ambito della contesa \n");
```

```
    else {
```

```
        if (scope == PTHREAD_SCOPE_PROCESS)
```

```
            printf("PTHREAD_SCOPE_PROCESS");
```

```
        else if (scope == PTHREAD_SCOPE_SYSTEM)
```

```
            printf("PTHREAD_SCOPE_SYSTEM");
```

```
        else
```

```
            printf(stderr, "Valore non ammesso.\n");
```

```
}
```

qualora non sia quello corretto, lo pone a SCS, lo pone a PTHREAD_SCOPE_SYSTEM, poi si creano 5 thread che competono per l'utilizzo della CPU

Il valore si mette in attesa che i figli finiscano

Se restituisce un valore diverso da 0 la funzione non è stata eseguita correttamente

Se scope impone valore desiderato





API Pthread per lo scheduling dei thread – 3

```
/* Imposta l'algoritmo di scheduling a SCS */
pthread_attr_setscope(&attr, PTHREAD_SCOPE_SYSTEM);
/* Genera i thread */
for (i = 0; i < NUM_THREADS; i++)
    pthread_create(&tid[i], &attr, runner, NULL);
/* Il processo padre aspetta la terminazione di tutti i thread */
for (i = 0; i < NUM_THREADS; i++)
    pthread_join(tid[i], NULL);
}

/* Ciascun thread inizia l'esecuzione da questa funzione */
void *runner(void *param)
{
    /* fai qualcosa... */
    pthread_exit(0);
}
```

*il tipo scope a sistema
Modello 1: 1*

crea i thread

*PTHREAD è una libreria
usabile anche da SO con
Modello 1: 1-4-1*





Scheduling multiprocessore – 1

- ❖ Lo scheduling diviene più complesso quando nel sistema di calcolo sono presenti più CPU
- ❖ Le architetture multiprocessore possono essere:
 - CPU multicore
 - Multicore multithread *più thread HW sullo stesso core*
 - Sistemi NUMA *NON UNIFORM MEMORY ACCESS*
ogni CHIP può avere solo memoria locale
 - Sistemi eterogenei *alcuni processori più omogenei
più core, ma con velocità e efficienza diverse*
- ❖ Ipotesi:
 - Nei primi tre casi, le unità di elaborazione sono, in relazione alle loro funzioni, identiche – sistemi omogenei
⇒ Ripartizione del carico di lavoro
 - Sistemi eterogenei composti da processori con diverse caratteristiche hardware



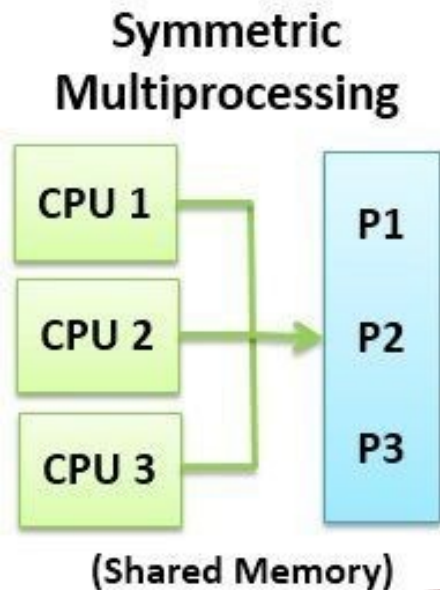


Scheduling multiprocessore – 2

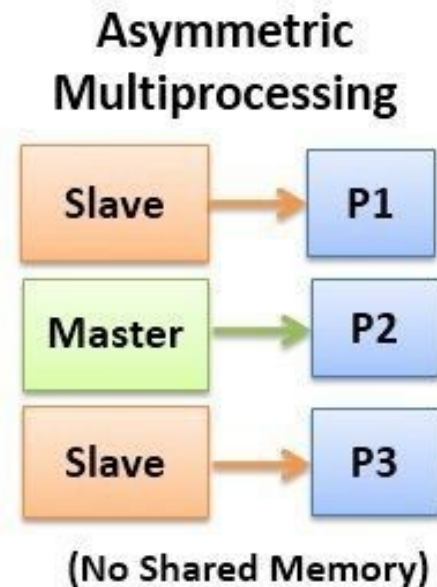
- CPU master e lo slave*
- ❖ **Multielaborazione asimmetrica** – lo scheduling, l'elaborazione dell'I/O e, in genere, le attività di sistema sono affidate ad un solo processore, detto *master server*

- Solo unico CPU si occupa dei thread kernel e dei kernel*
- Si riduce la necessità di condividere dati, grazie all'accesso di un solo processore alle strutture dati del kernel
 - Il master server costituisce un potenziale collo di bottiglia, capace di ridurre le prestazioni dell'intero sistema
- No concorrenza al livello kernel*

*Perché con
esegono
codice kernel
e utente,
codici della
stessa memoria*



Vs



serve





Scheduling multiprocessore – 3

- ❖ **Multielaborazione simmetrica** (SMP, *Symmetric Multi-Processing*) – i thread pronti vanno a formare una coda comune oppure vi è un'apposita coda per ogni processore; ciascun processore ha un proprio scheduler che esamina la coda opportuna per prelevare il prossimo thread da eseguire

tutti i processori possono prelevare da una coda opportuna i processi (un core | core virtuale o Hyperthreading)

Il SO decide quale sarà prossimo processore su un determinato core

- L'accesso concorrente di più processori ad una struttura dati comune rende delicata la programmazione degli scheduler che...
 - ▶ ...devono evitare di scegliere contemporaneamente lo stesso thread da eseguire
 - ▶ ...e devono evitare che qualche thread pronto "vada perso"
- ❖ Adottata da molti SO attuali, per esempio Windows, Mac OS, Linux, iOS e Android





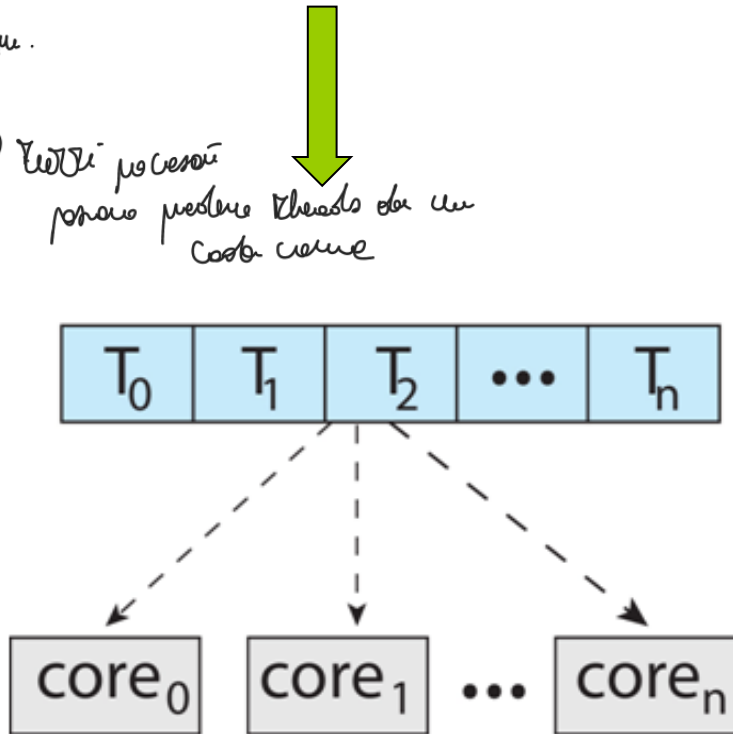
Scheduling multiprocessore – 4

Possibili race condition; l'accesso alla coda deve essere gestito tramite lock

l'accesso alla stessa coda

SMP Sym.

① tutti i processori possono prendere threads da una coda comune

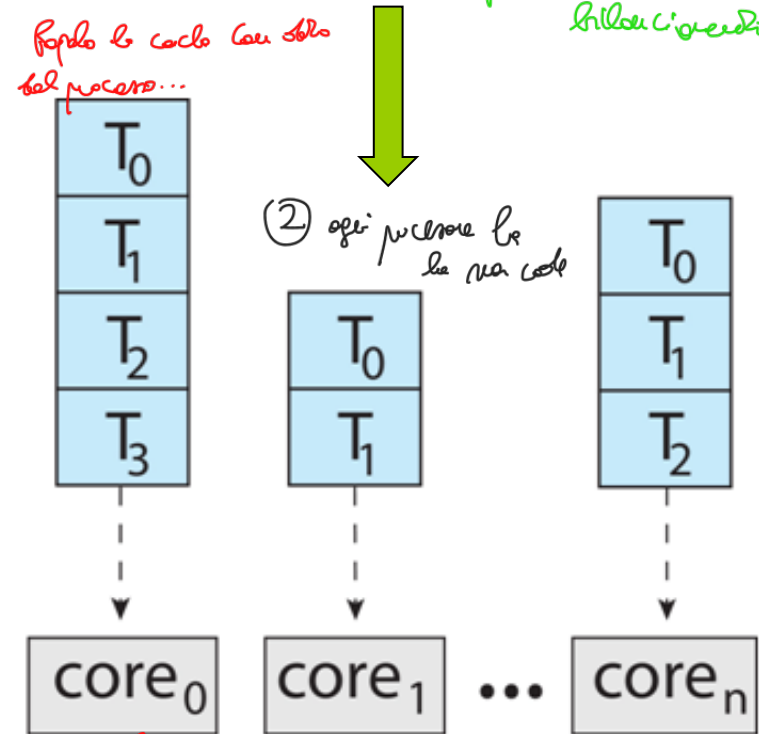


common ready queue

complicato, si rischia che stesso processore scelto da più scheduler, dovremmo mettere un lock sul thread

ogni processore ha una coda
Possibile sbilanciamento del carico di lavoro; uso più efficiente della cache

problema del bilanciamento



per-core run queues

Favorisce HIT

non ho uso della cache





Processori multicore – 1

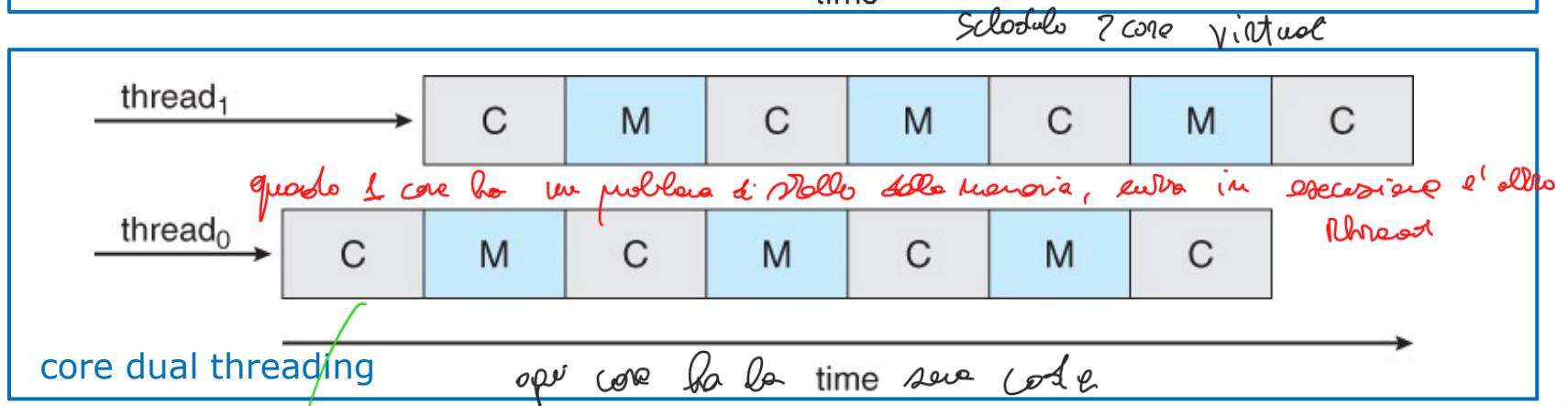
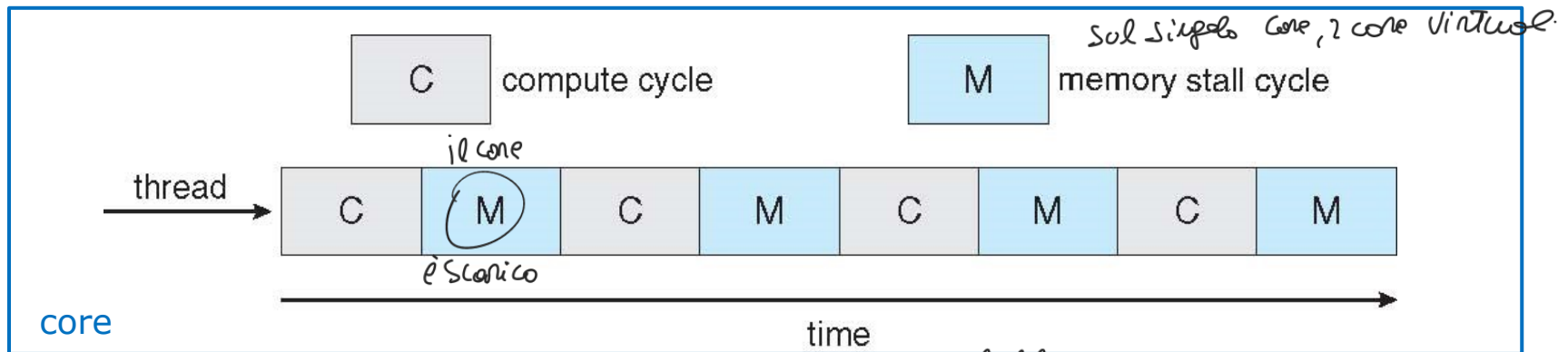
- ❖ Tradizionalmente i sistemi SMP hanno reso possibile la concorrenza fra thread con l'utilizzo di diversi processori fisici
- ❖ I sistemi SMP su processori multicore sono più veloci e consumano meno energia *costo 1/2 ridotto di cache*
- ❖ Quando un processore accede alla memoria (cache miss), una quantità significativa di tempo (fino al 50%) trascorre in attesa della disponibilità dei dati:
stallo della memoria
 - Nel *multithreading simultaneo* (SMT o *hyperthreading*, nella terminologia Intel) si implementano unità di calcolo multithread, in cui due o più thread hardware sono assegnati ad un singolo core
- ❖ Dal punto di vista del SO, ogni thread hardware appare come un processore logico in grado di eseguire un thread software





Processori multicore – 2

- ❖ **Esempio:** sistema dual core, dual threading, il SO vede quattro processori logici



se un istruzione viene interrotta, va a

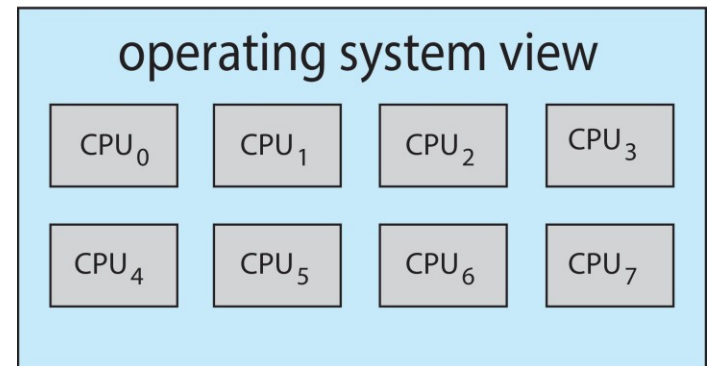
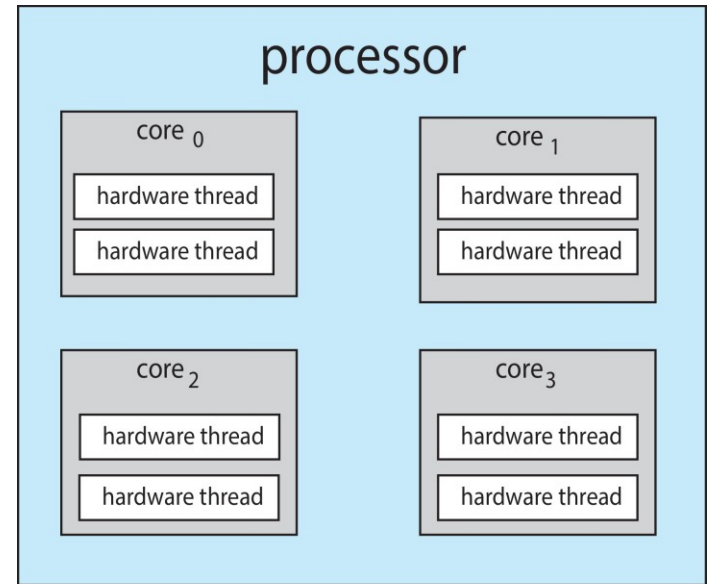
prima la pipe





Processori multicore – 3

- ❖ Nei **chip multithreading** (CMT), a ciascun core vengono assegnati un certo numero di thread hardware
- ❖ In un quad-core, con 2 thread hardware per core, si hanno 8 processori logici *per il SO*
- ❖ **Esempio:** Core i9–10900K di Intel, con 10 core, ha 20 thread quando l'hyperthreading è abilitato





Processori multicore – 4

comp resto a multi-threading HW

❖ Multithreading coarse-grained

- Il thread resta in esecuzione su un processore fino al verificarsi di un evento a lunga latenza (cache miss)
- Occorre ripulire la pipeline (tempo di latenza) *si perde tempo*

❖ Multithreading fine-grained *NO PIPELINE*

- Passa da un thread ad un altro al termine di un ciclo di istruzione o, in altre parole, il processore esegue una singola istruzione per ogni thread
- Necessaria una logica dedicata che garantisca una estrema rapidità nel context-switch





Processori multicore – 5

- ❖ Un processore multicore e multithread richiede due livelli di scheduling

- Liv. Logico* {
 - Scheduling effettuato dal SO per stabilire quale thread software mandare in esecuzione su ciascun thread hardware → algoritmi standard *lista di thread da schedare sui 2N core (virtual-)*
 - Scheduling relativo a ciascun core per decidere quale thread hardware mandare in esecuzione *nel singolo core*
 - ▶ Round Robin (UltraSPARC T3, 16 core a 8 thread)
 - ▶ *Liv. Fisico e' 1* Priorità o *urgency* (Intel Itanium, 2 core a 2 thread) *addeco a ciscu thread una prio*

- ❖ Se il SO è consapevole del multithreading hardware può evitare di schedare, qualora il sistema sia sufficientemente scarico, due thread software su due thread hardware relativi allo stesso core

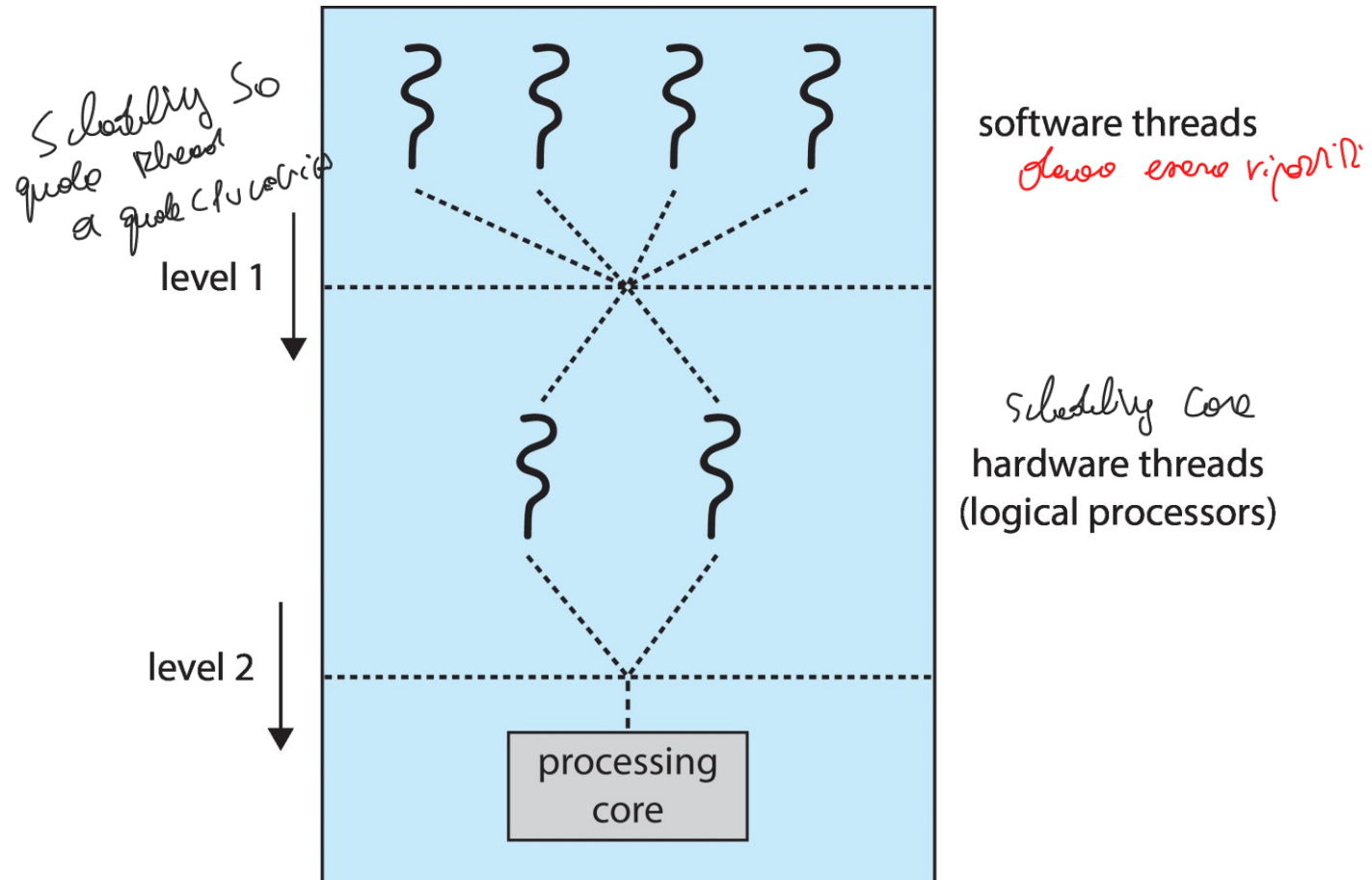
Se ho thread a sufficienza, Per evitare concorrenza, schedo thread ai singoli core





Processori multicore – 6

Bilanciamento del carico





Multielaborazione simmetrica – 1

❖ Bilanciamento del carico – Ripartire uniformemente il carico di lavoro sui diversi processori

Scheduler
Visto, posto
in attesa il
processo
pronto
da eseguire

Se ho più
code

- Nei sistemi con ready queue comune è automatico *il bilanciamento*
- Migrazione guidata (*push migration*): un thread dedicato controlla periodicamente il carico dei processori per effettuare eventuali riequilibri *controlla il bilanciamento*
- Migrazione spontanea (*pull migration*): un processore inattivo sottrae ad uno sovraccarico un thread in attesa
- Linux le implementa entrambe: esegue il proprio algoritmo di bilanciamento ad intervalli regolari e ogniqualvolta si svuota la coda di attesa di un processore

❖ Carico bilanciato può significare

- Numero simile di thread in ogni coda
- Equa distribuzione delle priorità dei thread
- ...





OPPOSITE TENDENZE

Multielaborazione simmetrica – 2

Se bilanciamento,
spinto thread,
nessuno come più probabile

Se tendono su processore sullo stesso core, ha più probabilità di HIT nella cache

- ❖ **Predilezione per il processore** – Mantenere un thread in esecuzione sempre sullo stesso processore, per riutilizzare il contenuto della cache per burst successivi
 - **Predilezione debole** – soft affinity: il SO tenta di mantenere il thread su un singolo processore, ma la migrazione non è interdetta *N*
 - **Predilezione forte** – hard affinity: il thread è vincolato all'esecuzione su uno o più processori *legato processore ed un solo processore*
 - Linux implementa sia la predilezione debole che forte; per la predilezione forte dispone di chiamate di sistema (es., `sched_setaffinity()`) con cui specificare che un processo non può abbandonare un dato processore *Se deve essere più rapido*
 - Solaris supporta la predilezione debole: non garantisce che, per particolari condizioni di carico, i processi non subiscano spostamenti
- ❖ Il bilanciamento del carico è antitetico rispetto alla filosofia di predilezione del processore





NUMA e CPU scheduling

- ❖ L'architettura della memoria influenza la predilezione
- In caso di **NUMA** (*Non Uniform Memory Access*), situazione standard in sistemi costituiti da diverse schede ognuna con una o più CPU e memoria, le CPU di una scheda accedono più velocemente alla memoria locale, rispetto alle memorie residenti sulle altre schede
- ⇒ Ogni thread dovrebbe risiedere nella memoria locale al processore su cui viene eseguito

Ciò che CPU
fa con memoria
locale

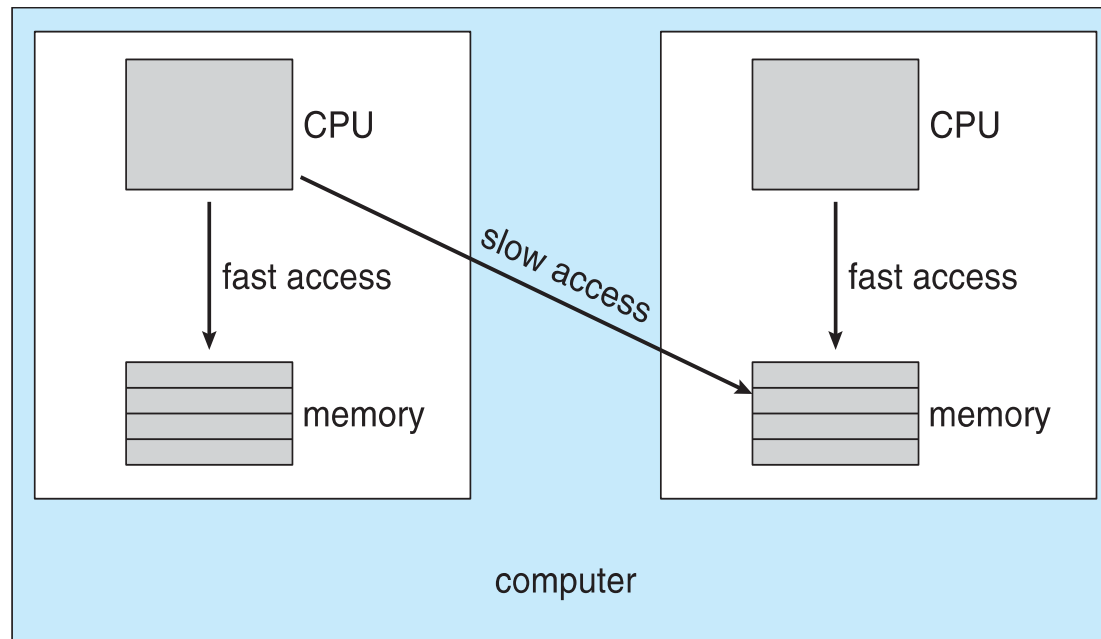
l'accesso all'altra
memoria REMOTA

È MOLTO
LENTA



Tocca forzare l'affinità alla
singola CPU

WAX solo serve
eserc' quello
che serve





Multiprocessing eterogeneo – 1

architettura eterogenea

- ❖ Sistemi (mobili) a multielaborazione eterogenea (HMP – *Heterogeneous MultiProcessing*) *Diversi core con performance diversi*
 - Non seguono il paradigma del multiprocessing asimmetrico: tutte le attività possono essere eseguite su qualsiasi core
 - Sebbene i sistemi mobili includano architetture multi-core, sono spesso progettati utilizzando core che eseguono lo stesso set di istruzioni, ma differiscono sia per la velocità del clock che per la gestione energetica
 - Possibilità di regolare il consumo energetico di un core fino a renderlo inattivo
 - Per risparmiare energia, si assegna un task ad un determinato core in base alle sue specifiche esigenze





Multiprocessing eterogeneo – 2

- ❖ Nei processori ARM che la supportano, l'architettura HMP è detta **big.LITTLE** ed è costituita da grandi core con prestazioni elevate (big) combinati con piccoli core ad alta efficienza energetica (LITTLE)
 - Core big per applicazioni in foreground *de interesse con l'utente*
 - Core LITTLE per applicazioni (lunghe) in background
 - Se il dispositivo mobile si trova in modalità di risparmio energetico, è possibile disabilitare i core big e forzare il sistema ad utilizzare solo i LITTLE
- ❖ Nell'iPhone 15 Pro, processore A17 Bionic con CPU con 6 core, con 2 core **performance** (3.78 GHz) e 4 **efficiency** (2.11 GHz) a ridotto consumo energetico





Scheduling real-time – 1

- ❖ **Sistemi hard real-time:** *Task deve essere eseguito in tempi prestabiliti* richiedono il completamento dei processi critici entro un intervallo di tempo predefinito e garantito
- ❖ **Sistemi soft real-time:** richiedono solo che ai processi critici sia assegnata una maggiore priorità rispetto ai processi di routine (nessuna garanzia sui tempi di completamento)
- ❖ I sistemi real-time, specialmente hard real-time, sono per loro natura guidati dagli eventi. *-> bisogno nuovo di dati da elaborare e esempio delle operazioni di gestione*
 - Si attende che si verifichi un evento cui si deve reagire in tempo reale
 - **Latenza dell'evento:** tempo che intercorre tra l'occorrenza dell'evento e il momento in cui il sistema ne effettua la gestione





Scheduling real-time – 2

- ❖ Più in dettaglio: le applicazioni hard-real time richiedono un tempo di risposta agli eventi esterni che sia prevedibile
- ❖ Una tipica applicazione in tempo reale presume:
 - Un dispositivo che genera interruzioni
 - Una routine di servizio dell'interrupt che raccoglie dati dal dispositivo
 - Un codice a livello utente^{o a livello kernel} che elabora i dati raccolti
- ❖ La **reattività** (*responsiveness*) è una caratterizzazione della velocità con cui un SO ed il codice di gestione dell'evento sono in grado di reagire minore è la latenza, più è la reattività
- ❖ La latenza è quindi una misura della responsiveness



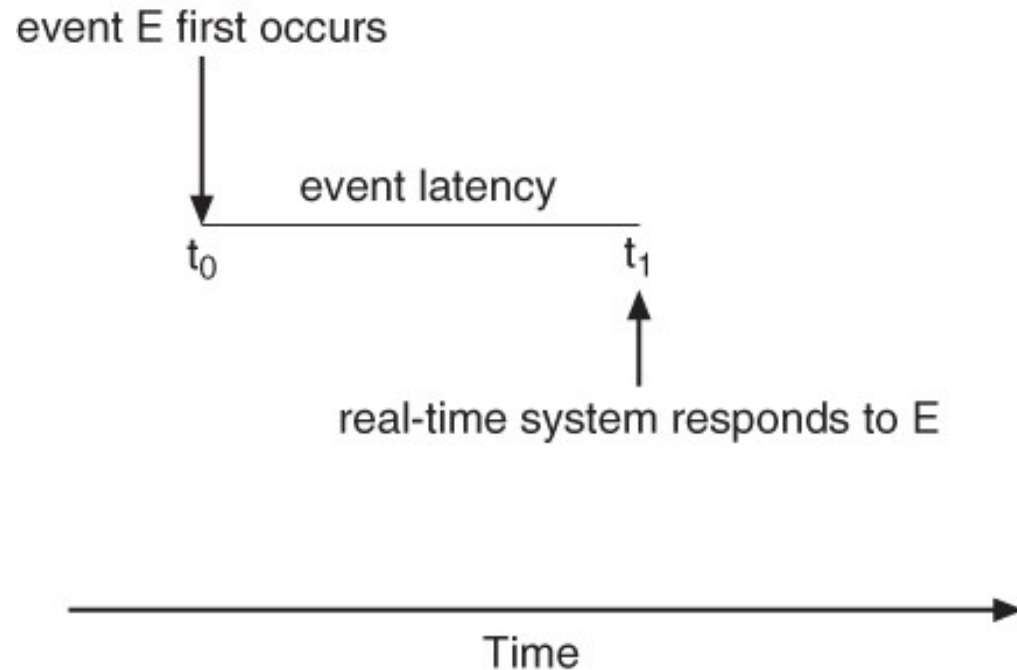


Scheduling real-time – 3

❖ Eventi diversi $\Rightarrow$ diversi requisiti di latenza

- ABS: 3–5 msec
- Meccanismo di controllo radar su aerei di linea: <10 sec

Real-time, Tempo relativo all'applicazione



Latenza relativa all'evento





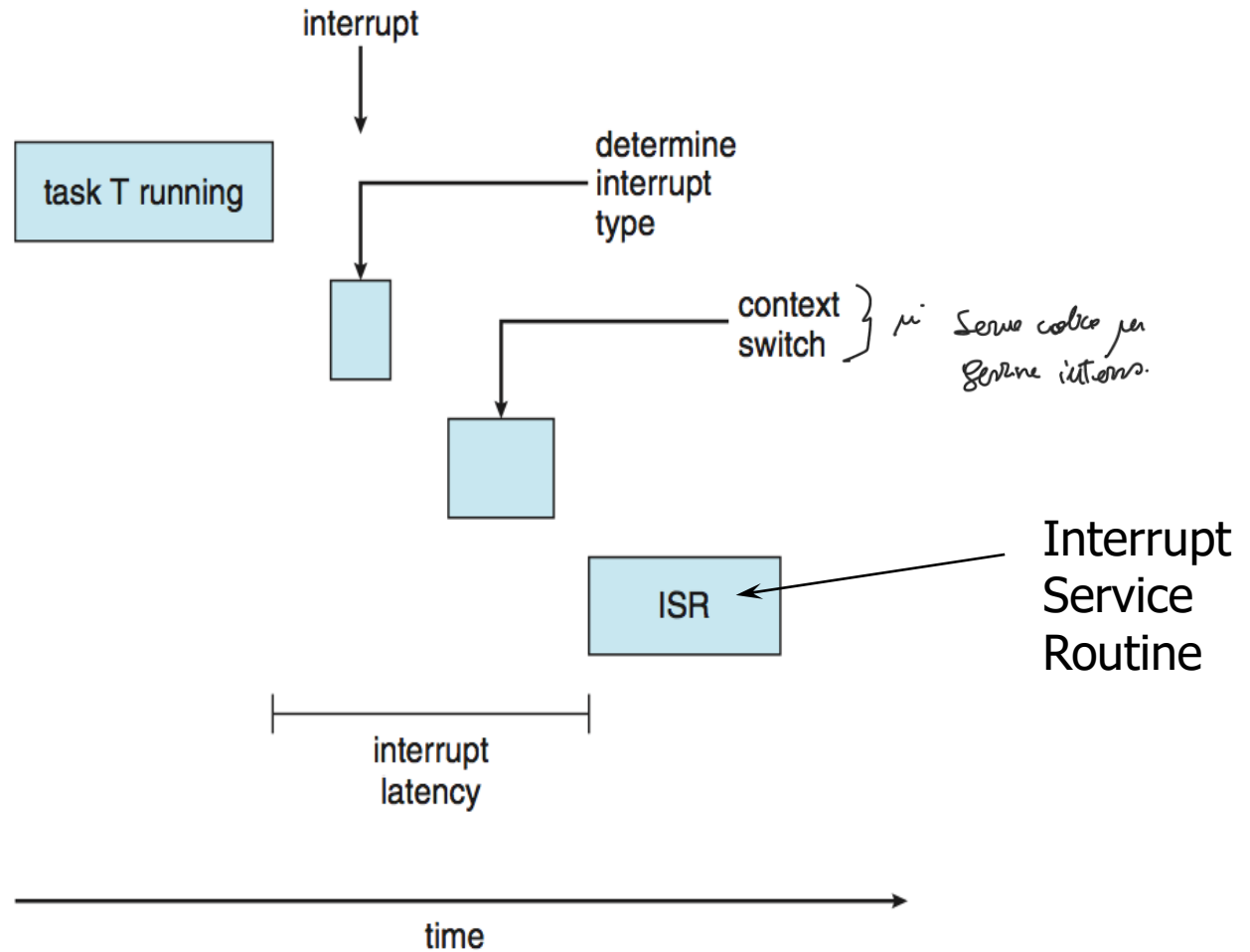
Scheduling real-time – 4

- ❖ Categorie di latenza che influiscono sul funzionamento dei sistemi real-time
 - Latenza relativa alle interruzioni: tempo compreso tra la notifica di un'interruzione alla CPU e l'avvio della routine che la gestisce
ma le interruz. possono essere ritardate durante alcune ops. di kernel
 - ▶ Fondamentale minimizzare l'intervallo di tempo in cui le interruzioni sono disattivate per l'aggiornamento dei dati del kernel
ci aggiust. anche il tempo per cui il SO disattiva gli interrupt
 - Latenza di dispatch: tempo necessario al dispatcher per interrompere un processo ed avviare il codice di gestione dell'evento
*prendere controllo della CPU dal vecchio processo e nuovo processo real-time
però anche prendere risorse al vecchio processo*





Scheduling real-time – 5



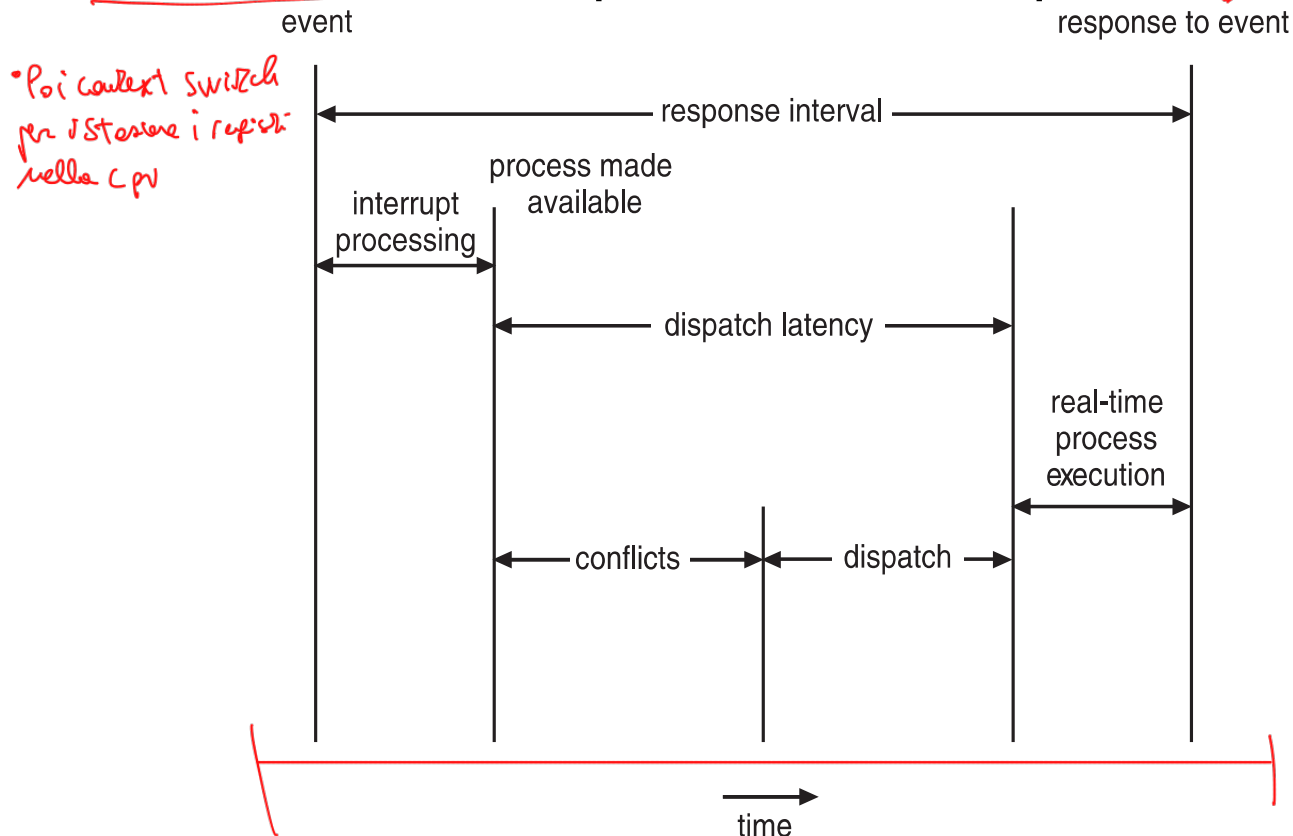
Latenza relativa alle interruzioni





Scheduling real-time – 6

- ❖ La cosiddetta **fase di conflitto** nella latenza di dispatch consiste di due componenti
 - Prelazione di ogni processo in esecuzione nel kernel
 - Cessione, da parte dei processi a bassa priorità, delle risorse richieste dal processo ad alta priorità





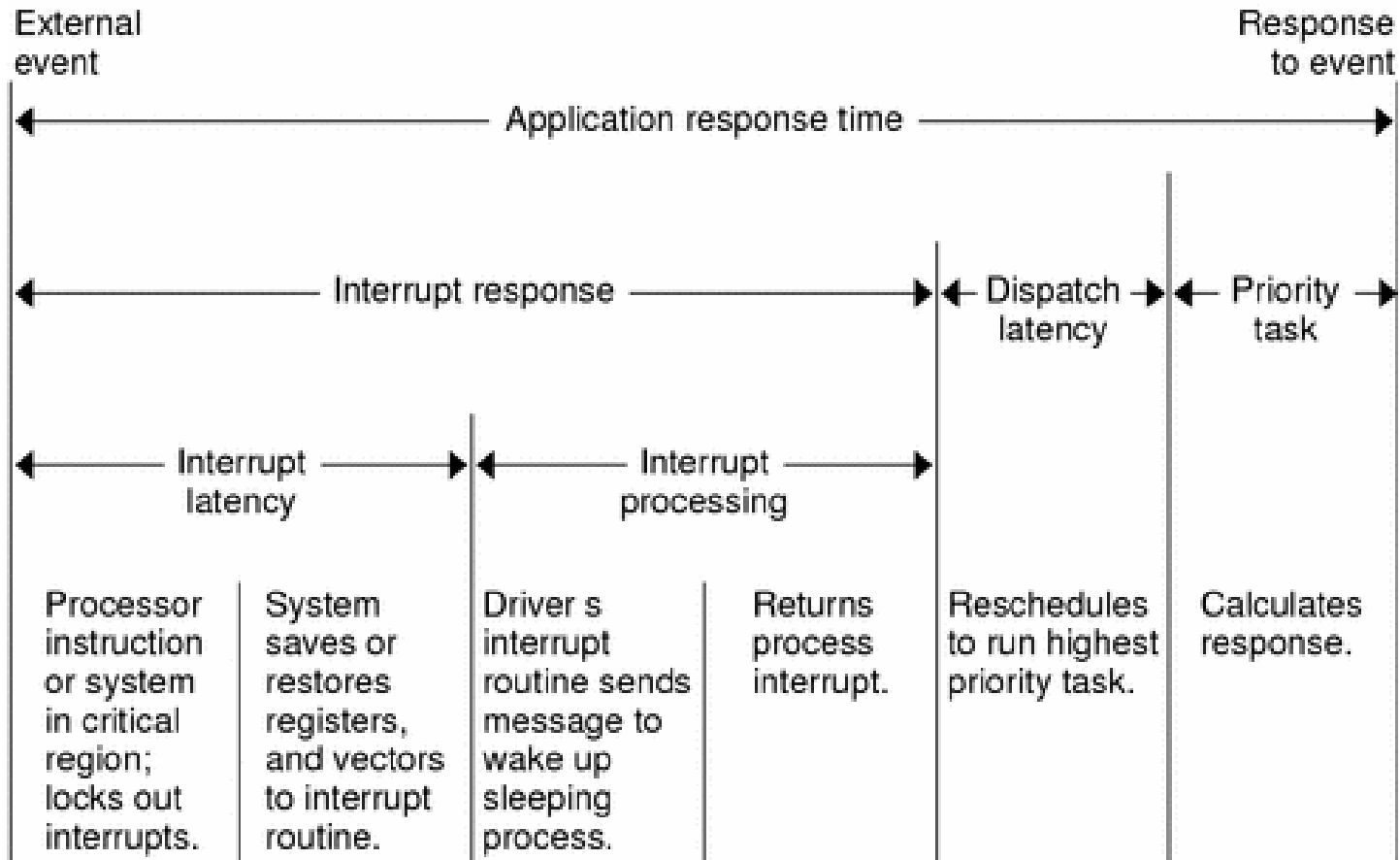
Scheduling real-time – 7

- ❖ La *event latency* complessiva è costituita quindi dal tempo di risposta all'interrupt, dalla latenza di dispatch e dall'esecuzione del task ad alta priorità
 - La latenza di interrupt è determinata dall'intervallo più lungo di tempo durante il quale il sistema esegue codice che impone che le interruzioni siano disabilitate
 - Durante l'elaborazione dell'interrupt, la routine di servizio cattura l'I/O e riattiva il processo ad alta priorità; il sistema rileva che è pronto un processo con priorità più elevata rispetto al processo interrotto e lo elabora
 - Il tempo per passare da un processo a priorità inferiore a un processo a priorità più alta è incluso nel tempo di latenza di dispatch





Scheduling real-time – 8





Scheduling real-time – 9

Per prima cosa

Devo sicuramente usare algoritmi a priori

❖ Per realizzare lo scheduling real-time, lo scheduler deve implementare un algoritmo con prelazione basato su priorità *calcolo in tempo statico o dinamico*

- Si garantisce solo la funzionalità soft real-time

❖ Nel caso dell'hard real-time si deve anche garantire che le attività in tempo reale vengano servite rispettando le scadenze

- Si considerino, per il momento, solo processi periodici, che richiedono la CPU a intervalli costanti di tempo (periodi); in particolare, ciascun processo... *potrei dire come per il 1° algoritmo*

- ▶ ...ha un tempo di elaborazione fisso t , una scadenza d , e un periodo p *all'interno del quale il processo viene ripulito*

→ ▶ $0 \leq t \leq d \leq p$

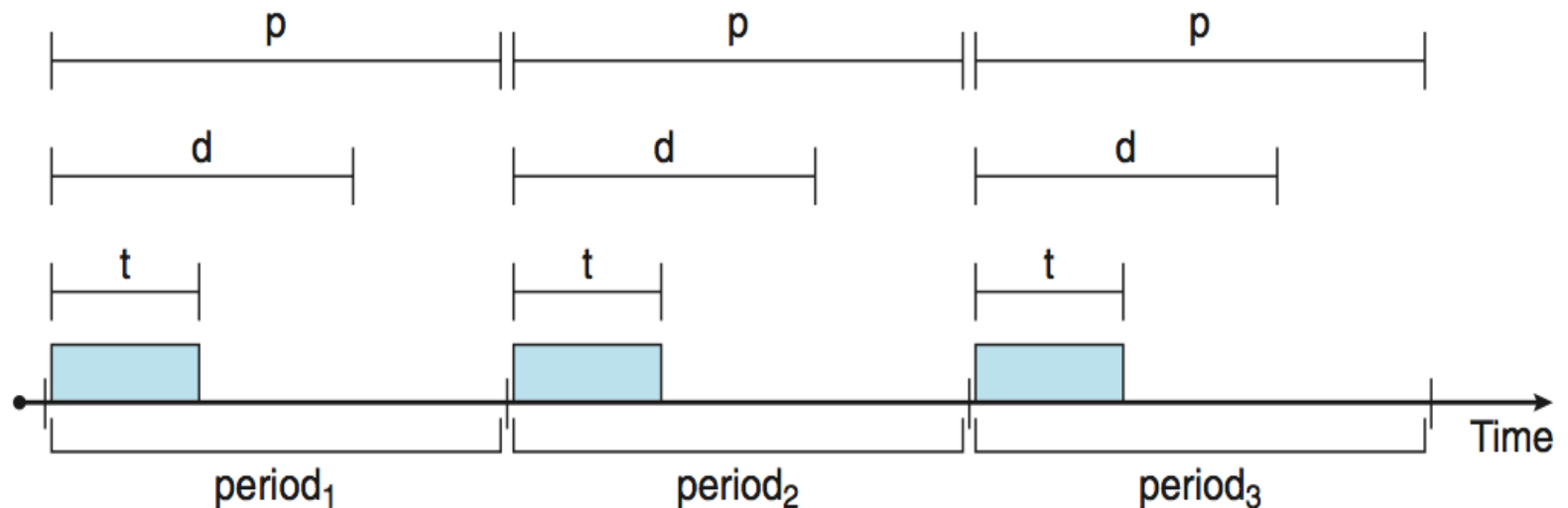
- ▶ La frequenza di un processo periodico è data da $1/p$





Scheduling real-time – 10

es se fo um unico processo



Processo periodico





Scheduling hard real-time – 1

❖ Scheduling con priorità proporzionale alla frequenza (o Rate Monotonic)

Priorità statica calcolata con la freq

- Si applica un modello statico di attribuzione delle priorità con prelazione *Priorità statica in base a freq*

- La priorità è inversamente proporzionale al periodo

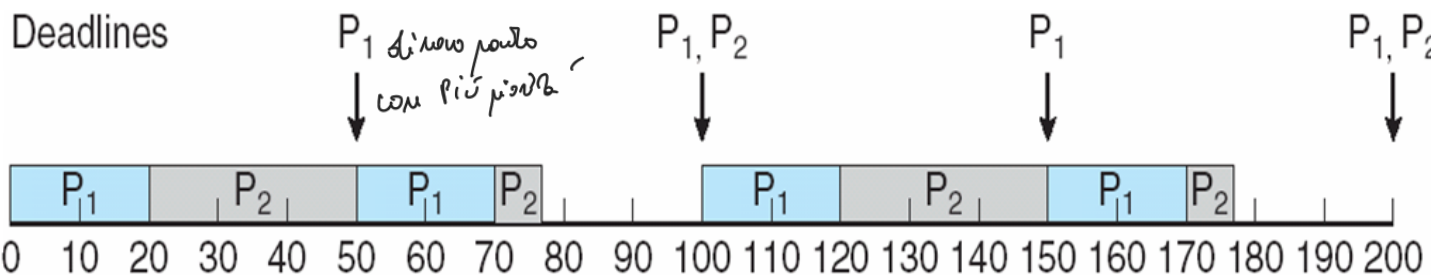
▶ Si assegnano priorità più elevate ai processi che fanno uso più frequente della CPU

- **Esempio:** siano P_1 e P_2 due processi con periodi rispettivamente pari a 50 e 100; supponiamo che i tempi di elaborazione siano invece $t_1=20$ e $t_2=35$ e sia $d=p$

P₁ maggiore priorità

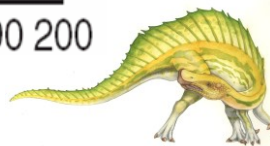
- P_1 ha priorità maggiore di P_2

in un periodo max un'exec. a processo



Algoritmo con prelazione

75% di utilizzo della CPU = $\frac{20}{50} + \frac{35}{100}$





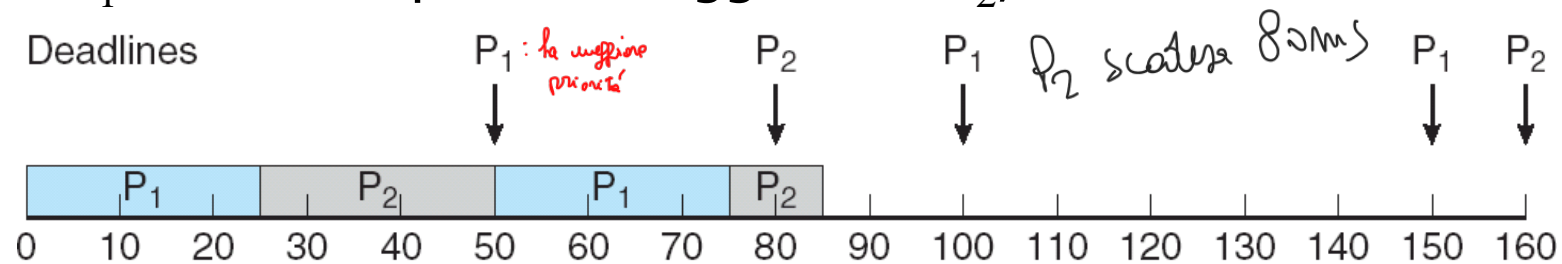
Scheduling hard real-time – 2

non c'è algoritmo migliore di priorit  statica

- ❖ Lo scheduling con priorit  proporzionale alla frequenza   un algoritmo ottimale, nel senso che se non   in grado di pianificare l'esecuzione di una serie di processi rispettando i vincoli temporali, nessun altro algoritmo che assegna priorit  statiche vi riuscir 

dove fallisce?

- **Esempio:** siano P_1 e P_2 due processi con periodi rispettivamente pari a 50 e 80; Supponiamo che i tempi di elaborazione siano invece $t_1=25$ e $t_2=35$
- P_1 ha ancora priorit  maggiore di P_2 , ma...



- ...in questo caso la scadenza relativa a P_2 non pu  essere rispettata (anche se si ha il 94% di utilizzo della CPU)

non riuscivo a stabilire le task per garantire la real-time





Neudruck: Schedulable

Scheduling hard real-time – 3

Perché una Funzione in quel caso?

- ❖ Nel caso di task periodici, se esistono m task e se il task i arriva con periodo P_i e richiede C_i secondi di tempo di CPU, il carico può essere gestito solo se:

$$\sum_{i=1}^m \frac{C_i}{P_i} \leq 1$$

$\nearrow$ Tempo di exec.
 $\searrow$ Periodo

} Per essere Schedulabile

- ❖ Un sistema real-time che rispetta questo vincolo è detto schedulabile *Processi Schedulabili*

$\frac{\text{tempo}}{\text{periodo}}$





Scheduling hard real-time – 4

- ❖ Tuttavia, è possibile dimostrare (Liu & Layland, 1973) che, per ogni sistema con processi periodici, il funzionamento dello scheduling con priorità proporzionale alla frequenza è garantito se:

Per processi periodici

*Scheduling
Hard Real-time
guaranteed*

$$\sum_{i=1}^m \frac{C_i}{P_i} \leq m \cdot (2^{1/m} - 1)$$

0,83
m=2
1
m

vale solo se m=2

x=1/m

$$\lim_{x \rightarrow 0} \frac{a^x - 1}{x} = \ln(a)$$

- ❖ Per $m=2$, il carico della CPU non può superare l'83%
- ❖ Per $m \rightarrow \infty$, il termine di maggiorazione tende a $\ln(2)$, quindi l'utilizzo della CPU si assesta a ~69.3% *non può essere > 90%*
- ❖ Si noti che, essendo il test di schedulabilità solo sufficiente, l'insieme di task può essere schedulabile con RM anche se la condizione non è soddisfatta

risco a trovare delle combinazioni per cui funziona





Scheduling hard real-time – 5

- ❖ **Scheduling EDF (Earliest-Deadline-First)** *Taski, anche con Task non periodici con tempi di esecuzione variabili.*
- Schedula i processi assegnando le priorità dinamicamente a seconda delle scadenze *Favorisco task con Deadline vicina*
- Quando un processo diventa eseguibile:
 - ▶ Deve annunciare la sua prossima scadenza allo scheduler
 - ▶ La priorità viene calcolata dinamicamente in base alla scadenza *priorità dinamica tra tutti i processi*
 - ▶ Più vicina è la scadenza più alta è la priorità
 - ▶ La priorità di altri processi già nel sistema viene modificata per riflettere la scadenza del nuovo processo
- Si applica anche a processi non periodici e con tempo di elaborazione variabile
- Ottimale e (idealmente) porta ad un utilizzo della CPU pari al 100%

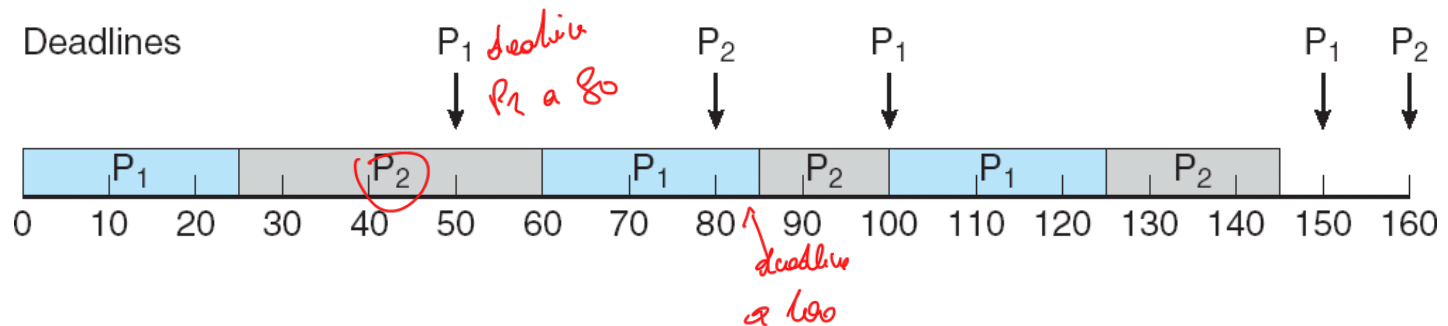




Scheduling hard real-time – 6

❖ Scheduling EDF *Early Deadline - First C?*

- **Esempio:** siano P_1 e P_2 due processi con periodi rispettivamente pari a 50 e 80; Supponiamo che i tempi di elaborazione siano invece $t_1=25$ e $t_2=35$





Scheduling hard real-time – 7

❖ Scheduling a quote proporzionali *es. 100%. 100 CPU*

- Opera distribuendo un certo numero di quote, T , fra tutte le applicazioni nel sistema *divido in quote la quota complessiva che assegno ai singoli Task*
- Un'applicazione può ricevere N quote di tempo, assicurandosi così l'uso di una frazione N/T del tempo totale di CPU
- **Esempio:** $T=100$ quote a disposizione da ripartire fra i processi A , B e C ; se A dispone di 50 quote, B di 15 e C di 20, la CPU è occupata all'85% *es. se vuole entrare D non necessita di 4 quote... non viene scusato*
- Lo scheduler deve lavorare in sinergia con un meccanismo di controllo dell'ammissione, per garantire che ogni applicazione possa effettivamente ricevere le quote di tempo che le sono state destinate (altrimenti non viene momentaneamente ammessa al sistema)





Esempio 4

$$T_{\text{ceda1}} = 2 \text{ ms}$$
$$T_{\text{ceda2}} = 4 \text{ ms}$$

$$\tau_1 = 5 \text{ ms}$$
$$\tau_2 = 7 \text{ ms}$$

Schedulabile

❖ Esercizio

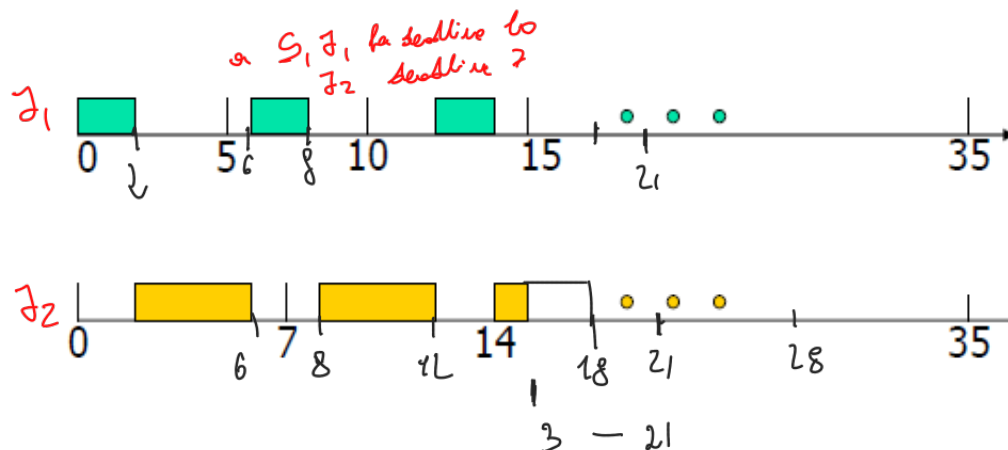
Si considerino i due task periodici real-time J_1 , con CPU-burst pari a 2msec e periodo di 5msec, e J_2 con tempo di esecuzione di 4msec e periodo di 7msec. Si valuti se i due task sono effettivamente schedulabili tramite EDF e, in caso affermativo, si mostri il relativo diagramma di Gantt.

❖ Soluzione

■ Task set: $\{(2,5),(4,7)\}$

Si vede per primo periodo deadline 6.5

■ $U = 2/5 + 4/7 = 34/35 \sim 0.97$ (schedulable!) *tecnicamente*





Esempio 5

❖ Esercizio

Si consideri il seguente insieme di processi con relativi tempi di arrivo, tempi di esecuzione e di scadenza:

Processo	Tempo di arrivo	Esecuzione	Scadenza
J_1	0	3	16
J_2	2	1	7
J_3	0	6	8
J_4	8	2	11
J_5	13	3	18

Processi
non
periodici

Caro real-time

Si tracci il diagramma di Gantt in base alla politica di scheduling EDF e si dica se tutte le scadenze risultano rispettate (lo scheduling è feasible, cioè ammissibile).





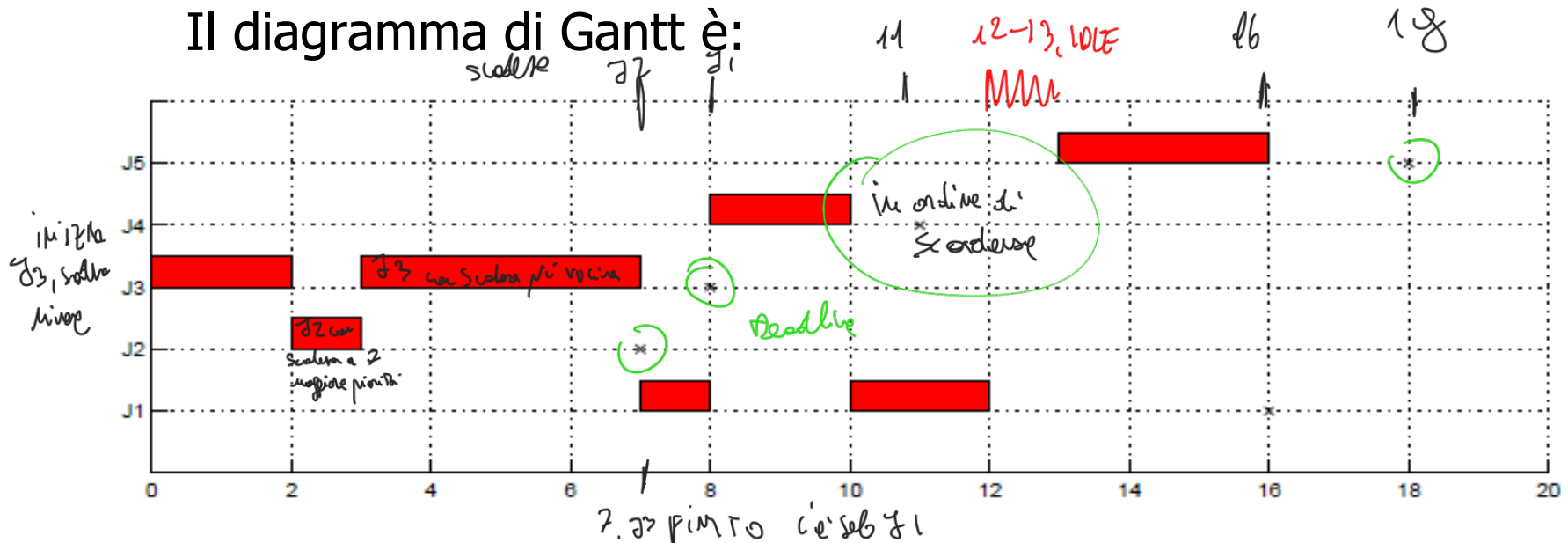
Esempio 5

❖ Soluzione

Processo	Tempo di arrivo	Esecuzione	Scadenza
J_1	0	3	16
J_2	2	1	7
J_3	0	6	8
J_4	8	2	11
J_5	13	3	18

Scadenza in ritardo

Il diagramma di Gantt è:



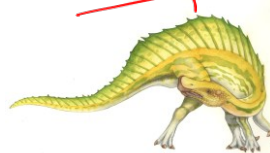
Lo scheduling è dunque ammissibile. deadline rispettata





Virtualizzazione e scheduling

- ❖ Il SO ospitante crea e gestisce le macchine virtuali, ciascuna dotata di un proprio SO ospite e proprie applicazioni; ogni SO ospite può essere messo a punto per usi specifici
 - ~~Ogni algoritmo di scheduling che fa assunzioni sulla quantità di lavoro effettuabile in tempo prefissato verrà influenzato negativamente dalla virtualizzazione, perché i singoli SO virtualizzati sfruttano a loro insaputa solo una parte dei cicli di CPU disponibili~~
 - ▶ Tempi di risposta dilatati
 - ▶ L'orologio di sistema delle macchine virtuali può essere rallentato
 - La virtualizzazione può vanificare i benefici di un buon algoritmo di scheduling implementato dal SO ospite sulla macchina virtuale





Scheduling in LINUX – 1

- ❖ **Prima della versione 2.5 del kernel**, era una variante dell'algoritmo tradizionale di UNIX (code multiple con feedback gestite perlopiù con RR) *Scegliere e salvare di priorità non buono per un cluster multiprocessore*
 - Prelazione sulle code con priorità attribuite internamente, dipendenti dall'utilizzo della CPU da parte dei processi (più è grande, minore è la priorità)
 - Periodicamente il kernel calcola quanta CPU un processo ha consumato dall'ultimo controllo: questa quantità è "inversamente proporzionale" alla priorità del processo fino al prossimo controllo (l'aging si realizza "naturalmente") *più veloce, più priorità*
 - Poco scalabile e non adeguato ai sistemi SMP





Scheduling in LINUX – 2

- ❖ Con la **versione 2.5** del kernel, lo scheduler è stato rivisitato per ottenere un algoritmo noto come $\mathcal{O}(1)$, indipendente dal numero di task presenti nel sistema

- Algoritmo a priorità preemptive
- Due intervalli di priorità, relativi a task time-sharing, detti nice, e (soft) real-time
- Priorità dei task real-time con valori compresi fra 0 e 99, fra 100 e 139 per i task nice (valori numericamente inferiori indicano priorità maggiori) *↳ si aumenta*
- Task a priorità maggiore ottengono time-slice più lunghi
- I task vengono eseguiti (rimangono attivi) fino ad esaurimento del time-slice *algoritmo RR, con quanto tempo massimo in base alla priorità*
- Esaurito il quanto di tempo, il task viene considerato "scaduto" e non verrà più eseguito finché tutti gli altri task del sistema siano scaduti

- ❖ Adatto ai sistemi SMP (supporta bilanciamento del carico e predilezione) *del processore*

- ❖ Tempi di risposta elevati per task time-sharing *lento per*

architetture monoprocesore

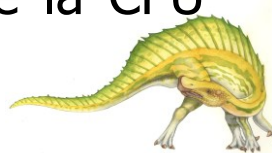




Scheduling in LINUX – 3

❖ Dalla versione 2.6.23: *Completely Fair Scheduler* (CFS)

- **Classi di scheduling** *Divisioni in 2 classi*
 - ▶ Ciascuna dotata di una specifico range di priorità
 - ▶ Lo scheduler seleziona il processo a più alta priorità, appartenente alla classe di scheduling a priorità più elevata
 - ▶ Invece di assegnare un quanto di tempo, lo scheduler CFS assegna ad ogni task una percentuale del tempo di CPU
 - ▶ Il kernel Linux standard implementa due classi di scheduling, dette default e real-time (possono però essere aggiunte altre classi) *task nice* *processi real-time / maggiori per essere* *Priorità diversa (cpu time) a seconda della*
- Per la classe default, la percentuale del tempo di CPU viene calcolata sulla base dei valori nice associati a ciascun task (da -20 a +19, valori minori per priorità maggiori)
 - ▶ CFS non utilizza valori discreti per i quanti di tempo, ma definisce una **latenza obiettivo**, cioè un intervallo di tempo entro il quale ogni task eseguibile dovrebbe ottenere la CPU almeno una volta *(un CPU burst)* *NON PUÒ ESSERE STIPULATO*
 - ▶ La latenza obiettivo cresce con il numero di task





Scheduling in LINUX – 4

calcolo priorità

- ❖ Lo scheduler CFS non assegna direttamente le priorità, ma registra per quanto tempo è stato eseguito ogni task, mantenendo il **tempo di esecuzione virtuale** di ogni task nella variabile **vruntime**

il sistema tiene un valore vruntime

- Il tempo di esecuzione virtuale è associato ad un **fattore di decadimento** che dipende dalla priorità del task (task a bassa priorità hanno fattori di decadimento più alti e **vruntime** maggiore rispetto al tempo effettivo di esecuzione)

Calcolo ultimo CPU-BURST, e viene moltiplicato per un nuovo moltiplicato per il nuovo limite

*nice < 0
vruntime < runtime vero*

*es. vruntime = 20
nice = -20
si considera che abbia finito per 1ms*

*nice > 0
vruntime > runtime*

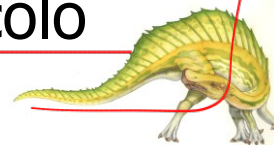
vruntime piccolo, vera esecuzione più

*nice = 0
vruntime = runtime reale*

- Per i task con priorità normale (nice=0), il tempo di esecuzione virtuale coincide con il tempo effettivo di esecuzione

nice = priorità

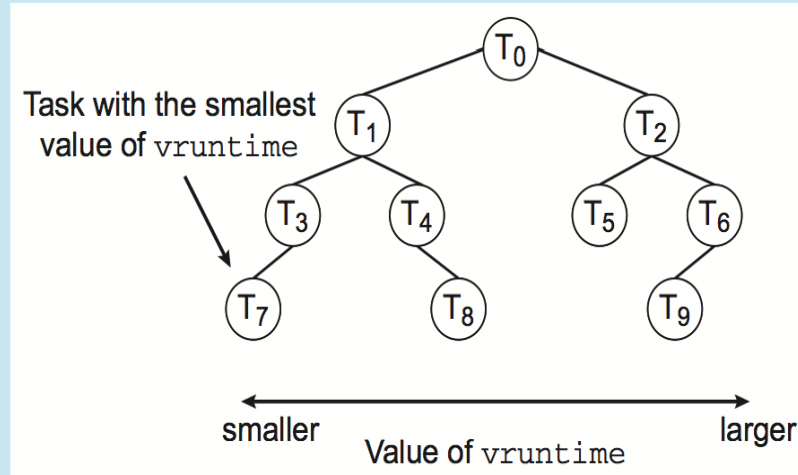
- ❖ Per decidere il prossimo task da eseguire, lo scheduler seleziona il task con il valore **vruntime** più piccolo





Scheduling in LINUX – 5

The Linux CFS scheduler provides an efficient algorithm for selecting which task to run next. Each runnable task is placed in a red-black tree—a balanced binary search tree whose key is based on the value of `vruntime`. This tree is shown below:



*implemento ci: con
un albero bilanciato
per trovare più in fretta
il task con vruntime
minore*

When a task becomes runnable, it is added to the tree. If a task on the tree is not runnable (for example, if it is blocked while waiting for I/O), it is removed. Generally speaking, tasks that have been given less processing time (smaller values of `vruntime`) are toward the left side of the tree, and tasks that have been given more processing time are on the right side. According to the properties of a binary search tree, the leftmost node has the smallest key value, which for the sake of the CFS scheduler means that it is the task with the highest priority. Because the red-black tree is balanced, navigating it to discover the leftmost node will require $O(\lg N)$ operations (where N is the number of nodes in the tree). However, for efficiency reasons, the Linux scheduler caches this value in the variable `rb_leftmost`, and thus determining which task to run next requires only retrieving the cached value.

Prestazioni di CFS

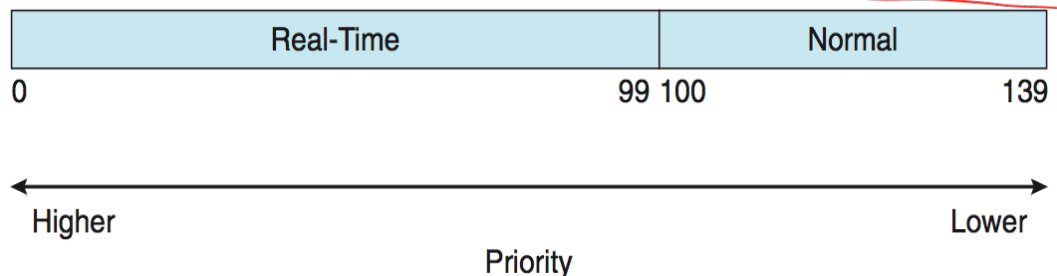




Scheduling in LINUX – 6

- ❖ **Real-time** (priorità 0–99)
 - Soft real-time: implementa lo standard POSIX.1b
 - ▶ Priorità (esterne) statiche
- ❖ **Task default**
 - Nice=-20 si mappa sul valore di priorità globale 100, mentre +19 viene mappato su 139
- ❖ Le priorità relative dei processi in tempo reale sono assicurate, ma il nucleo non fornisce garanzie sui tempi di attesa nella coda dei processi pronti
- ❖ Se un segnale d'interruzione dovesse rendere eseguibile un processo real-time mentre il nucleo è impegnato nell'esecuzione di una chiamata di sistema, il processo attende

“Real-time plus normal processes map into a global priority scheme”





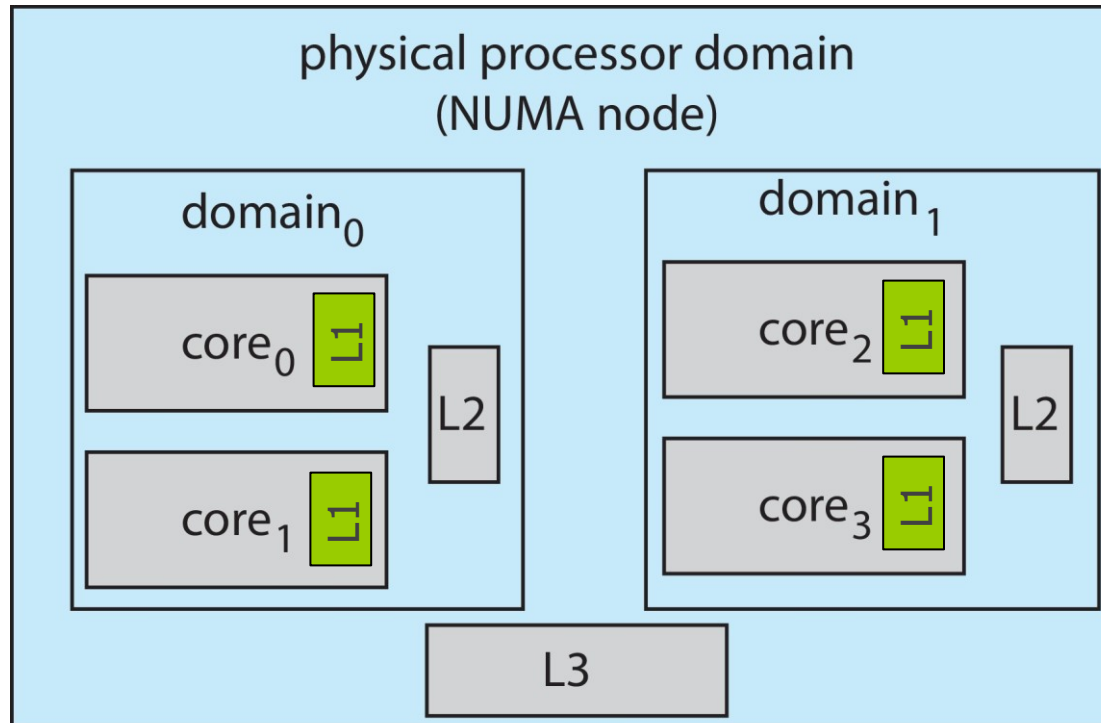
Scheduling in LINUX – 7

- ❖ Lo scheduler CFS supporta il bilanciamento del carico, ma è anche compatibile con NUMA (*NUMA-aware*) e riduce al minimo la migrazione dei thread
 - Il carico di ogni thread è dato dalla combinazione della sua priorità e del suo tasso medio di utilizzo della CPU
 - ▶ **Esempio:** Carico basso per task ad alta priorità, ma I/O-bound, o con bassa priorità e CPU-bound
 - ▶ Il carico delle code, che è la somma dei carichi di tutti i task che le compongono, deve essere approssimativamente uguale *carico code = somma dei 2 contributi*
 - Per ridurre la migrazione:
 - ▶ Linux identifica un sistema gerarchico di domini di scheduling, cioè un insieme di core che possono essere bilanciati con il minor impatto possibile sull'accesso alla memoria
 - ▶ I domini sono organizzati in base a ciò che condividono (ad esempio, memorie cache di diversi livelli)
 - ▶ Limitare la migrazione tra domini





Scheduling in LINUX – 8



spostare task da un processore all'altro, minimizzare

di spostare tra i core della singola CPU

di localizzare

Bilanciamento del carico NUMA-aware nello scheduler CFS di Linux





Valutazione degli algoritmi – 1

❖ Come selezionare un algoritmo di scheduling della CPU per un particolare sistema operativo?

⇒ Definizione dei criteri da usare per la scelta dell'algoritmo

Coesistenza possibile

- **Esempio 1:** massimizzare l'utilizzo della CPU, con tempo massimo di risposta inferiore a 300 millisecondi *system thresholding interval*
- **Esempio 2:** massimizzare la produttività, con tempo di completamento proporzionale (in media) al tempo di esecuzione effettivo *throughput*





Valutazione degli algoritmi – 2

- ❖ **Valutazione analitica** – in base all'algoritmo ed al carico di lavoro del sistema, fornisce una formula per valutarne le prestazioni
- ❖ **Modelli deterministici** – *Snapshot dello stato del sistema* valutazione analitica che considera un carico di lavoro predeterminato e definisce le prestazioni di ciascun algoritmo per quel carico di lavoro
 - **Esempio:** si considerino cinque processi che arrivano tutti al tempo 0

<u>Process</u>	<u>Burst Time</u>
P_1	10
P_2	29
P_3	3
P_4	7
P_5	12

non significa burst time, altrimenti userei SRTF
SRTF

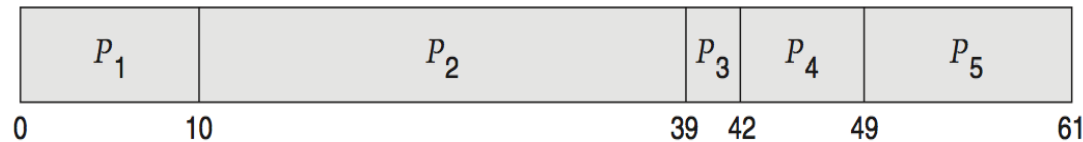




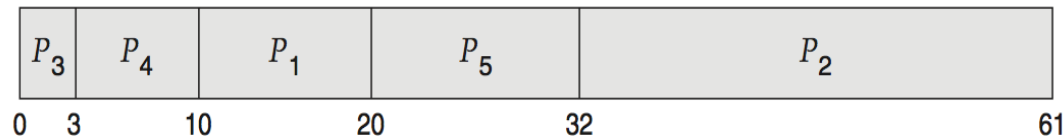
Valutazione degli algoritmi – 3

- ❖ Per ciascun algoritmo si calcola il tempo medio di attesa
- ❖ Metodo semplice e veloce che richiede però una conoscenza delle dinamiche del sistema normalmente non disponibile

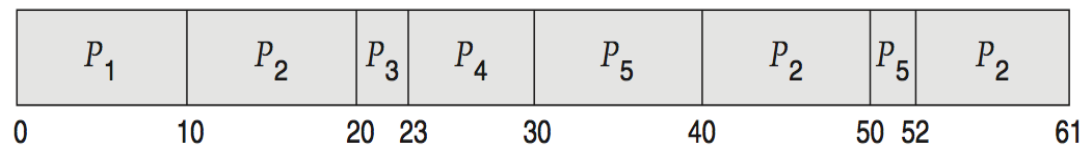
- **FCFS** — $T_a = (0 + 10 + 39 + 42 + 49) / 5 = 28 \text{ msec}$



- **SJF (non preemptive)** — $T_a = (10 + 32 + 0 + 3 + 20) / 5 = 13 \text{ msec}$



- **RR** con time-slice=10 — $T_a = (0 + 32 + 20 + 23 + 40) / 5 = 23 \text{ msec}$





Valutazione degli algoritmi – 4

Si suppone che quest'insieme HW sia modellato come...

- ❖ **Reti di code** – il sistema di calcolo viene descritto come una rete di server, ciascuno con una coda d'attesa
 - La CPU è un server con la propria coda dei processi pronti, ed il sistema di I/O ha le sue code dei dispositivi
 - Se sono noti l'andamento degli arrivi e dei servizi (sotto forma di distribuzioni di probabilità), si può calcolare utilizzo di CPU e dispositivi, lunghezza media delle code, tempo medio d'attesa, throughput medio, etc.
 - ▶ Normalmente, si rilevano distribuzioni esponenziali, descritte attraverso valori medi
 - Problema: difficoltà nell'utilizzo di distribuzioni complicate/metodi matematici appositi $\Rightarrow$ modelli non realistici

*nella base di un modello statico
modello distribuito di eventi con distribuzioni standard ma
sono
irrealistiche*





Valutazione degli algoritmi – 5

per avere qualcosa di rappresentativo del sistema reale

- ❖ Sia n la lunghezza media di una coda e siano W il tempo medio di attesa nella coda e λ l'andamento medio di arrivo dei nuovi processi *nell'unità di medio*
- ❖ Se il sistema è stabile, il numero di processi che lasciano la coda deve essere uguale al numero di processi che vi arrivano *Se # processi che entrano = # processi che escono*

$$\begin{matrix} \text{lunghezza} \\ \text{coda} \end{matrix} n = \begin{matrix} \text{tempo arrivo processi} \\ \text{tempo medio} \end{matrix} \lambda \times W$$

è la **formula di Little**, valida per qualsiasi algoritmo di scheduling e distribuzione degli arrivi

- ❖ È utilizzabile per il calcolo di una delle tre variabili, quando sono note le altre due: per esempio, se ogni secondo arrivano 7 processi e la coda ne contiene mediamente 14, il tempo medio d'attesa per ogni processo è pari a 2 secondi *non troppo realistico*





Valutazione degli algoritmi – 6

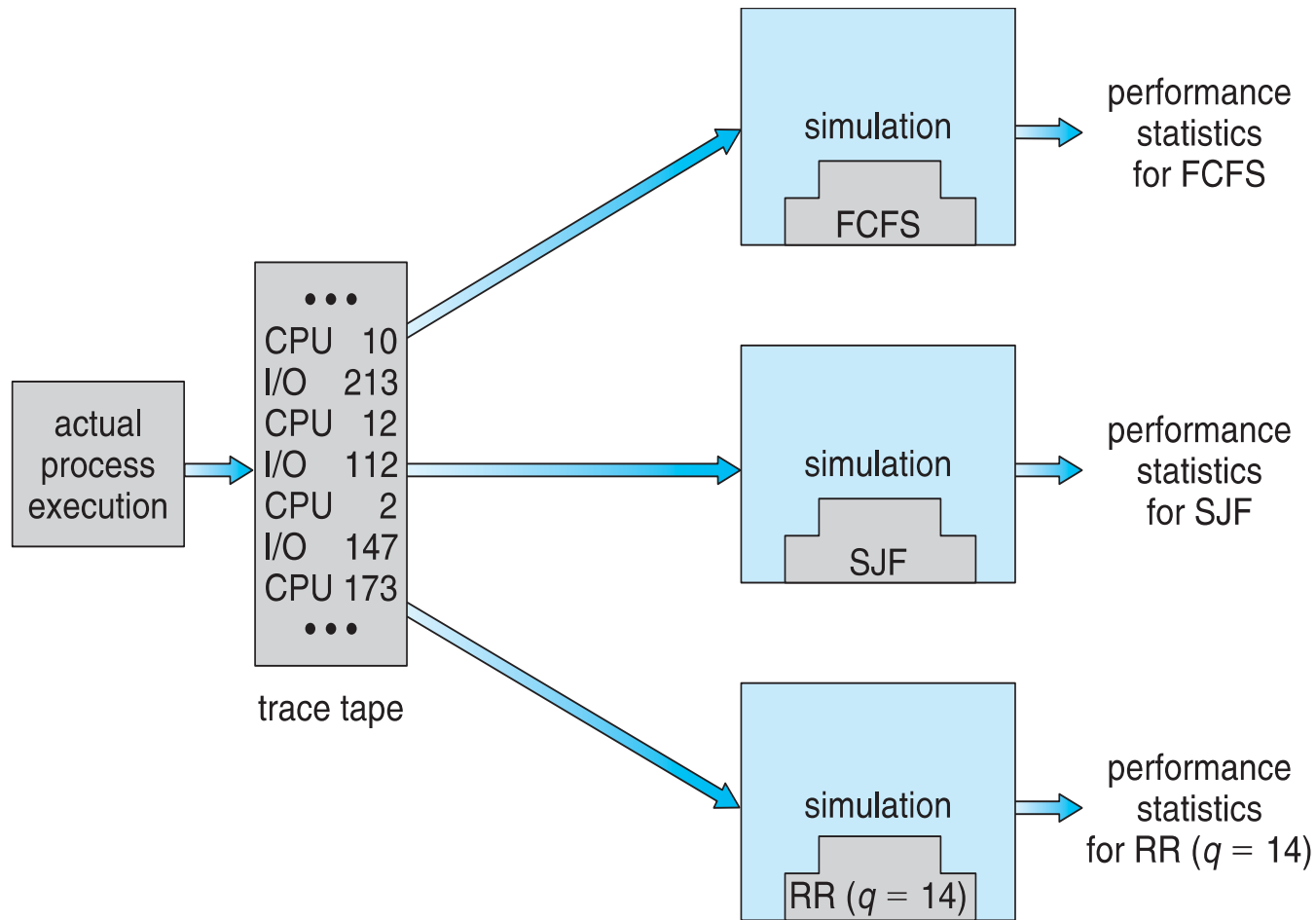
alternativa per vedere se algoritmo e' buono... simulato l'interossier

- ❖ **Simulazione** – implica la programmazione di un modello del sistema di calcolo; le strutture dati rappresentano gli elementi principali del sistema
 - Il simulatore dispone di una variabile che rappresenta il clock e modifica lo stato del sistema in modo da descrivere le attività dei dispositivi, dei processi e dello scheduler
 - Durante l'esecuzione della simulazione, si raccolgono e si stampano statistiche che descrivono le prestazioni degli algoritmi *uso generano cose di cui*
 - I dati per la simulazione possono essere generati artificialmente, modellati da distribuzioni matematiche o empiriche o raccolti da un sistema reale mediante un *trace tape* *file di log, dati li sono ad un simulazione*





Valutazione degli algoritmi – 7



Valutazione di algoritmi di CPU scheduling tramite simulazione





Valutazione degli algoritmi – 8

- ❖ **Implementazione** – implica la codifica effettiva degli algoritmi di scheduling da valutare, ed il loro inserimento nel sistema operativo, per osservarne il comportamento nelle condizioni reali di funzionamento del sistema
 - Difficoltà nel fare accettare agli utenti un sistema in continua evoluzione
 - Alti costi, alti rischi
 - ▶ Adeguamento del comportamento degli utenti alle caratteristiche dello scheduler
 - Gli scheduler più flessibili possono essere modificati *per-site* o *per-system*





Considerazioni finali

- ❖ Gli algoritmi di scheduling più adattabili sono quelli che possono essere tarati dagli amministratori di sistema, che li adattano al particolare ambiente di calcolo, con la propria specifica gamma di applicazioni
- ❖ Molti degli attuali sistemi operativi UNIX-like forniscono all'amministratore la possibilità di calibrare i parametri di scheduling in vista di particolari configurazioni del sistema (es. Solaris)





Esercizi

❖ Esercizio 1

Cinque lavori, indicati con le lettere da A a E, arrivano al calcolatore approssimativamente allo stesso istante. I processi hanno un tempo di esecuzione stimato di 8, 10, 2, 4 e 8 secondi, rispettivamente, mentre le loro priorità (determinate esternamente) sono 2, 4, 5, 1, 3 (con 5 priorità max). Per ognuno dei seguenti algoritmi di scheduling

- Round robin (2 sec)
- Scheduling a priorità (non preemptive)
- FCFS
- SJF (non preemptive)

si calcoli il tempo medio di turnaround. Si ignori l'overhead dovuto al cambio di contesto.





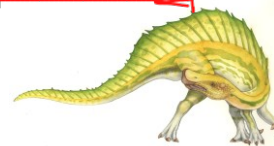
Esercizi (cont.)

❖ Esercizio 2

Si consideri il seguente insieme di processi:

Processo	Tempo di arrivo	Tempo di burst (msec)
P_0	0	7
P_1	2	4
P_2	3	4
P_3	5	2
P_4	7	3
P_5	10	2

I processi arrivano nell'ordine indicato. Si calcoli il tempo medio di turnaround con scheduling SJF preem-
ptive.





Esercizi (cont.)

❖ Esercizio 3

Si consideri il seguente insieme di processi:

Processo	Tempo di arrivo	Tempo di burst (msec)
A	0	3
B	2	6
C	4	4
D	6	5
E	8	2

Si calcoli il tempo medio di attesa ed il tempo medio di turnaround, nel caso di scheduling FCFS, RR con quanto di tempo 1 e 4 e SJF non preemptive.





Esercizi (cont.)

❖ Esercizio 4

Si consideri un sistema con scheduling a priorità con tre code, A, B, C, di priorità decrescente e con prelazione tra code. Le code A e B sono servite con modalità RR con quanto di 15 e 20 ms, rispettivamente; la coda C è FCFS. Se un processo nella coda A o B consuma il suo quanto di tempo, viene spostato in fondo alla coda B o C, rispettivamente. Si supponga che i processi P_1 , P_2 , P_3 , P_4 arrivino rispettivamente nelle code A, C, B, A, con CPU burst e tempi indicati nella tabella seguente:

Processo	Tempo di arrivo	Tempo di burst (msec)
P_1	0	20
P_2	10	25
P_3	16	20
P_4	25	20

Si tracci il diagramma di Gantt relativo all'esecuzione dei quattro processi e si calcoli il tempo medio di turnaround.





Esercizi (cont.)

❖ Esercizio 5

Si considerino i seguenti processi, attivi in un sistema multiprogrammato:

Processo	Tempo di arrivo	Tempo di burst (msec)	Priorità
P_1	0	8	2
P_2	1	6	3
P_3	2	3	2
P_4	3	4	2
P_5	4	3	2

Descrivere la sequenza di esecuzione dei job, tramite diagramma di Gantt e calcolare l'istante in cui ogni job completa il suo task, ed il relativo tempo di turnaround, ottenuto mediante scheduling a code multiple a priorità e feedback, dove ogni coda è gestita con la strategia FCFS. Il feedback è definito come segue: la priorità diminuisce di 1 (fino al livello base 1) se si passano più di 6 unità di tempo (consecutive) in esecuzione ed aumenta di 1 per ogni 6 unità di tempo passate in attesa in una qualsiasi coda.





Esercizi (cont.)

❖ Esercizio 6

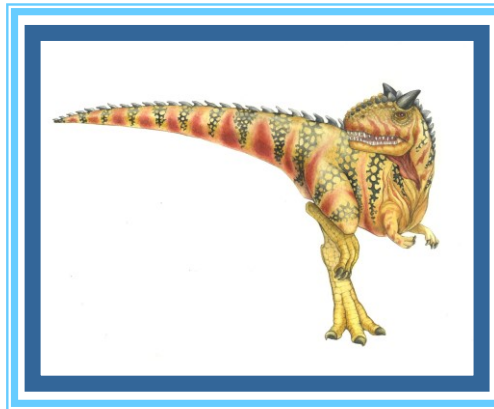
Si considerino N ($N \geq 2$) processi in contesa su una singola CPU gestita con scheduling Round Robin. Supponiamo che ciascun context switch abbia una durata di S msec e che il quanto di tempo sia pari a Q . Per semplicità, si assuma inoltre che i processi non siano mai bloccati a causa di un qualche evento e semplicemente passino dall'essere in esecuzione all'essere in attesa nella ready queue.

Si calcoli il massimo quanto di tempo Q tale che, per ciascun processo, non trascorrono più di T msec tra l'inizio di due burst successivi.

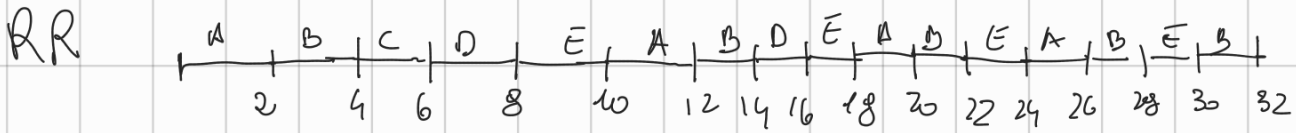
Si calcoli il massimo quanto di tempo Q tale che ciascun processo non debba attendere più di T msec nella ready queue. I valori di Q dovranno essere calcolati in funzione di N , S e T .



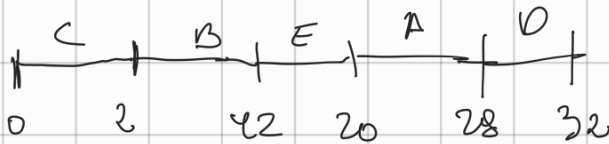
Fine del Capitolo 5



	BURST	PRIORITY	no t. context switch	
A	8	2	$T_T?$	1
B	10	4		
C	2	5 MAX	$t_{HT} = \frac{26 + 32 + 6 + 16}{4} = 20s$	
D	6	1		
E	8	3	$t_{TA} = 26s \quad t_{TB} = 32s$ $t_{TC} = 6s \quad t_{TD} = 16s$	

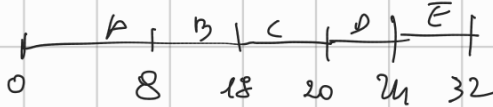


PRIORITY



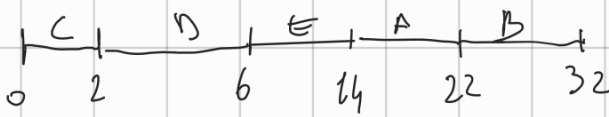
$$t_{HT} = \frac{28 + 12 + 2 + 32 + 20}{4} = 23,5s$$

FCFS



$$\frac{8 + 18 + 20 + 24 + 32}{4} = 29,5$$

SJF

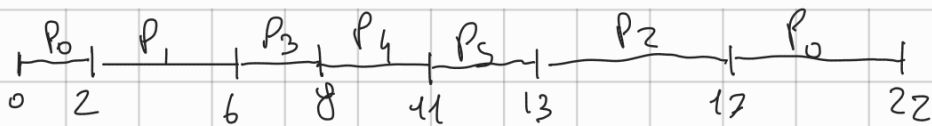


$$\frac{22 + 32 + 2 + 6 + 14}{4} = 19s$$

2

P	Arrivo	Burst msec
P ₀	0	7 5
P ₁	2	4
P ₂	3	4
P ₃	5	2
P ₄	7	3
P ₅	10	2

SJF Preemptive
con release

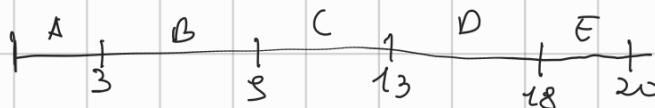


$$t_{m_T} = \frac{22 + 4 + 14 + 3 + 4 + 3}{6} = 8,33$$

P	Anno	Burst
A	0	3
B	2	6
C	4	4
D	6	5
E	8	2

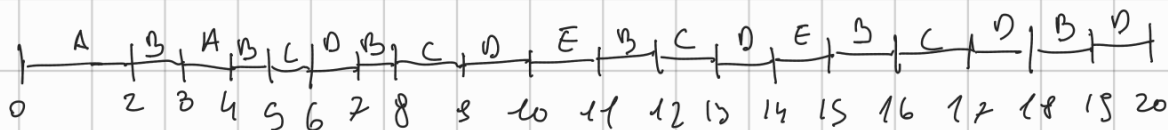
FCFS

$$t_{m_T} = 3 + (6-2) + (13-4) + (18-6) + (20-8) = 26$$



$$t_{m_T} = \frac{0 + 1 + 9 + 7 + 10}{5} = 4,6ms$$

RR $q=1$

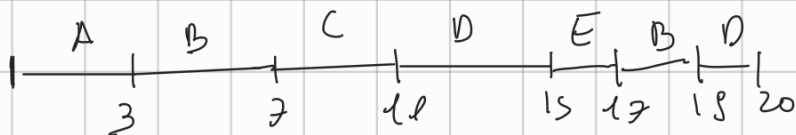


$$T_{m_T} = \frac{4 + 17 + 13 + 14 + 7}{5} = 11ms$$

$$t_a = \frac{1 + (1 + 2 + 3 + 3 + 2) + (1 + 2 + 3 + 3) + (2 + 3 + 3 + 1) + (2 + 3)}{5} = 7ms$$

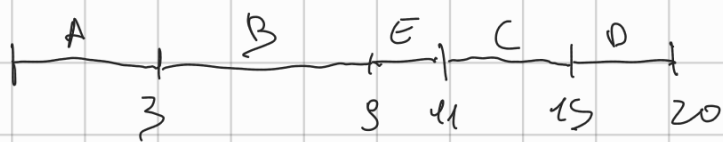
RR $q=4$

$$T_{m_T} = \frac{3 + 17 + 7 + 14 + 9}{5} = 10ms$$



$$t_a = \frac{0 + (1+10) + 3 + (5+4) + 7}{5} = 6ms$$

SJF non Preemptive



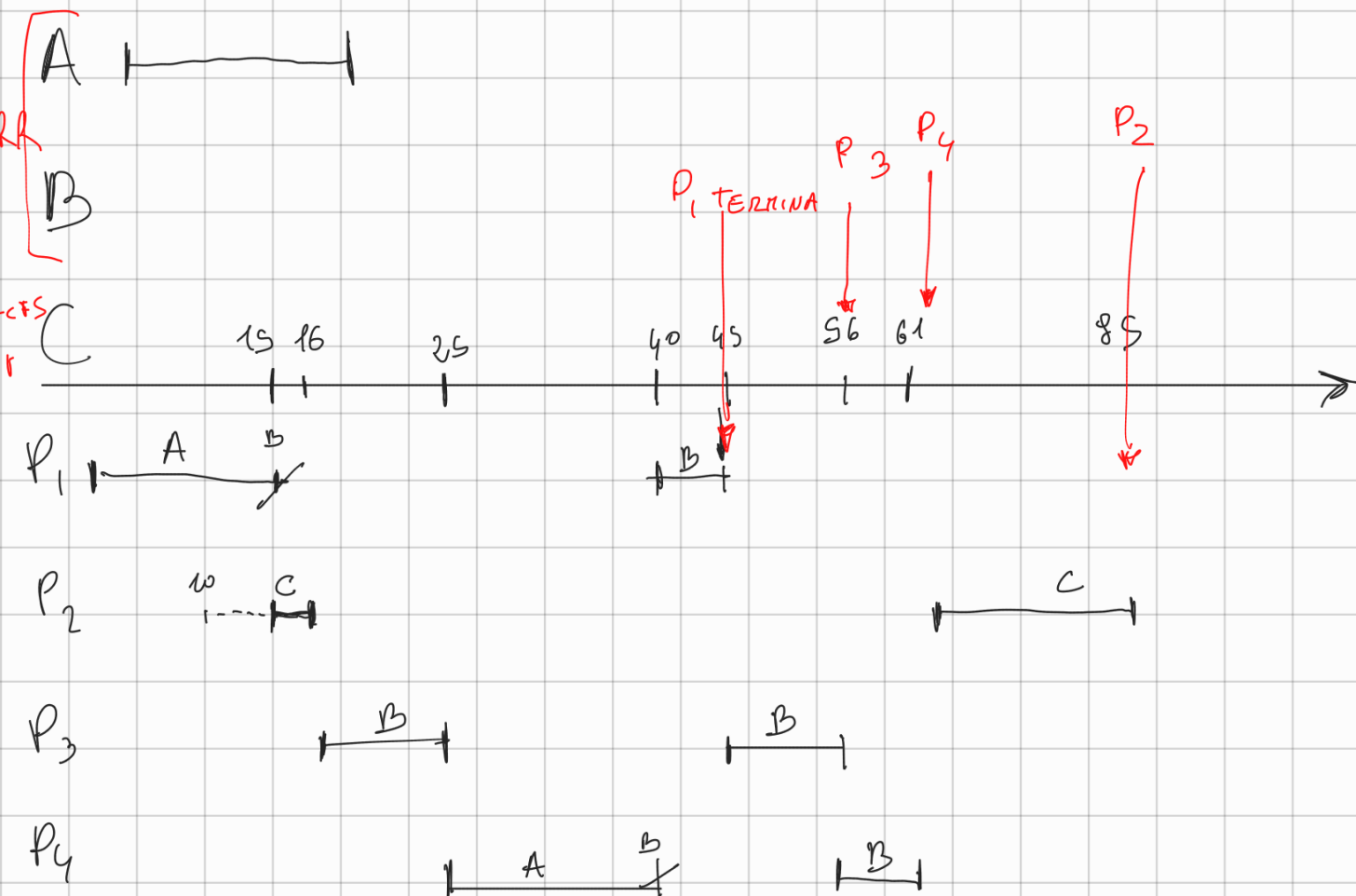
$$T_{avg} = \frac{3 + 7 + 11 + 15 + 20}{5} = 11,2$$

$$T_{avg} = \frac{0 + 1 + 7 + 9 + 7}{5} = 4,8$$

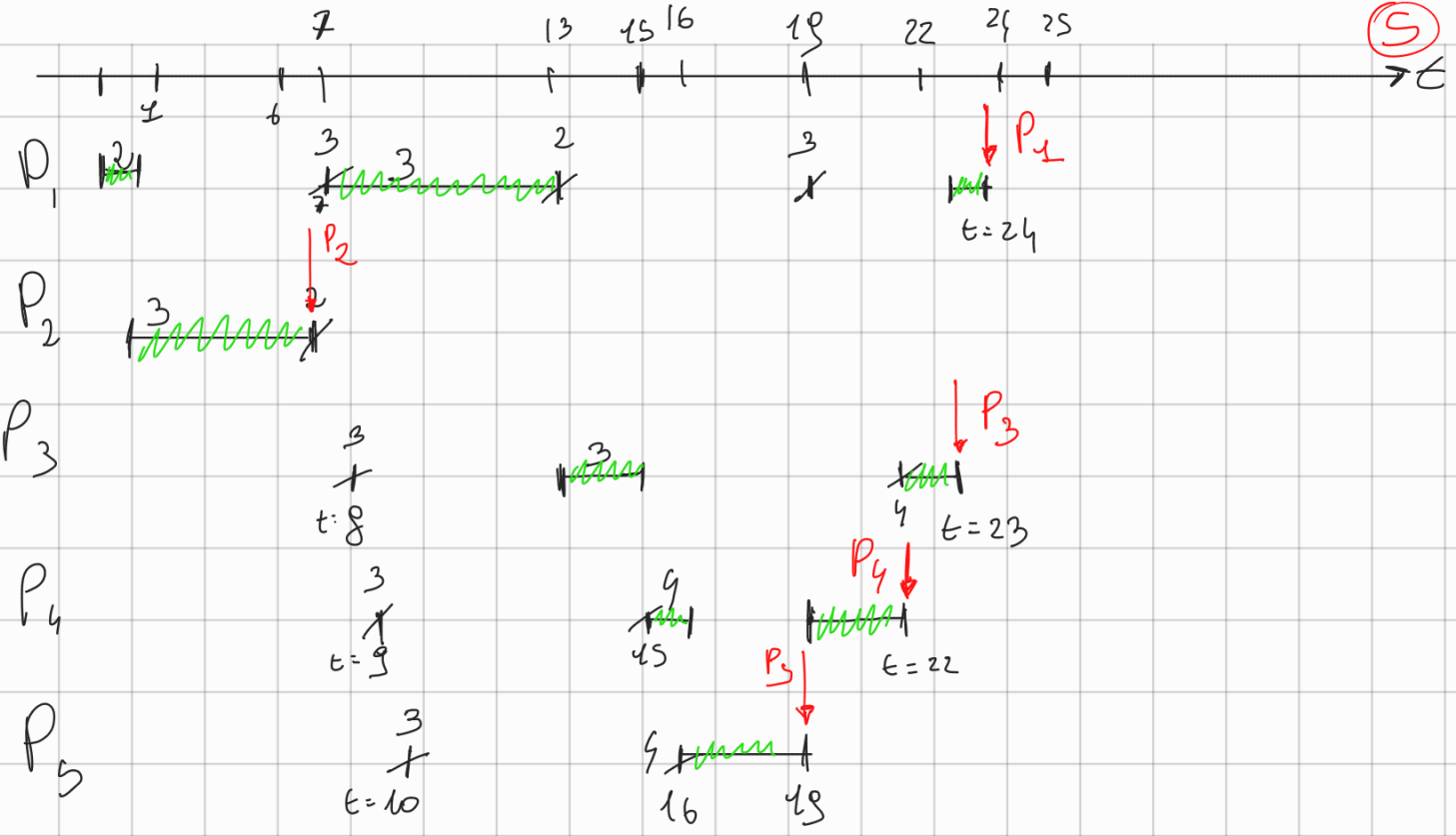
Priority: the Code Priority: process e PRELAPZ

Process	arrival	Burst
P ₁ A	0	20 5 B
P ₂ C	10	25 24 C
P ₃ B	16	20 11 B
P ₄ A	25	20 5 B

Code



5



Process	Arrivo	Esito	Quantità
P ₁	0	8	2
P ₂	1	6	3
P ₃	2	3	2
P ₄	3	4	2
P ₅	4	3	2

* Dubbio: Indagine o no ?

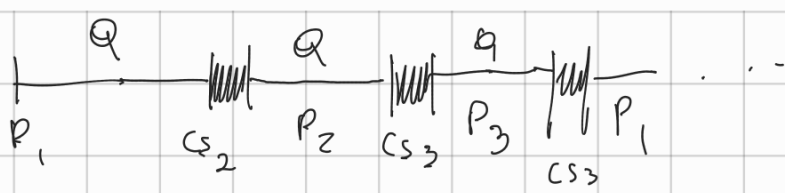
$$\begin{aligned}
 P_{1T} &= 24 \\
 P_{2T} &= 6 \\
 P_{3T} &= 21 \\
 P_{4T} &= 18 \\
 P_{5T} &= 19
 \end{aligned}$$

Es. 6 N processi RR C → W = S msec

Q = quantum

Q ∈ c.

6



$$T = 2Q + 3CS$$

$$T = (N-1)Q + NCS$$

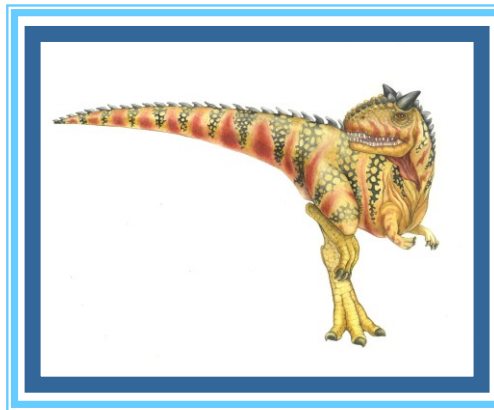
$$T \leq (N)Q + N \cdot S$$

$$Q_1 = \frac{T}{N} - N \cdot S \quad \text{max number of } T_{\text{type}} \text{ a case}$$

② T_{type} also body given $(N-1)Q + N \cdot S$

$$Q_2 = \frac{T}{N-1} - N \cdot S$$

Strumenti di sincronizzazione





Obiettivi

- ❖ Descrivere il problema della sezione critica ed illustrare come avvengono le race condition
- ❖ Mostrare soluzioni hardware alla mutua esclusione
- ❖ Dimostrare come lock mutex, semafori, monitor e variabili condizionali possono essere utilizzati per gestire la mutua esclusione
- ❖ Valutare gli strumenti che risolvono il problema della sezione critica in scenari a bassa, moderata e alta contesa

interpretos. di codice concorrente





Sommario

- ❖ Introduzione
- ❖ Il problema della sezione critica
- ❖ La soluzione di Peterson
- ❖ Supporto hardware alla sincronizzazione
- ❖ Lock mutex
- ❖ Semafori
- ❖ Monitor
- ❖ Liveness, stallo e inversione di priorità
- ❖ Valutazione delle soluzioni





Introduzione – 1

- ❖ Nei moderni SO, i task vengono eseguiti concorrentemente, su un core, e in parallelo, su core distinti *Thread concorrenti su core distinti*
- ❖ Un processo cooperante può influenzare gli altri processi in esecuzione nel sistema o subirne l'influenza
- ❖ I processi cooperanti possono... *processi cooperanti, memoria condivisa o messaggi*
 - ...condividere uno spazio logico di indirizzi, cioè codice e dati (thread, memoria condivisa) *memoria condivisa, si verifica...*
 - ...oppure solo dati, attraverso file o messaggi *Stessa cosa naturalmente di thread*
- ❖ L'accesso concorrente a dati condivisi può causare inconsistenza nei dati
- ❖ Per garantire la coerenza (l'integrità) dei dati occorrono meccanismi che assicurano l'esecuzione ordinata dei processi cooperanti

INTEGRITÀ DEI DATI





Introduzione – 2

- ❖ I processi possono essere interrotti dallo scheduler in qualsiasi punto del loro flusso di esecuzione, per assegnare la CPU (il core) ad un altro processo
- ❖ Affrontiamo di nuovo il problema del produttore–consumatore con buffer limitato
- ❖ Una soluzione in cui vengano impiegati tutti gli elementi del buffer presuppone...
 - ...la modifica del codice del produttore–consumatore con l'aggiunta di una variabile contatore, inizializzata a zero ** el del buffer che se ne pie **
 - **contatore** viene incrementato ogni volta che un nuovo elemento viene inserito nel buffer, decrementato dopo ogni estrazione





Dati condivisi



```
#define BUFFER_SIZE 10
typedef struct {
    . . .
} elemento;
elemento buffer[BUFFER_SIZE];
int in = 0;
int out = 0;
int contatore = 0;
```

in indica la successiva posizione libera nel buffer →

out indica la prima posizione piena nel buffer →

contatore conta il numero di elementi presenti nel buffer →

Buffer di struct elemento

*3 int condivisi
Da i due
processi*





Produttore-Consumatore

Processo produttore

*Finisce nel while se
buffer è pieno*

```
elemento appena_prodotto;  
while (1) {  
    while (contatore == BUFFER_SIZE)  
        ; /* do nothing */  
    buffer[in] = appena_prodotto;  
    in = (in + 1) % BUFFER_SIZE;  
    contatore++;  
}
```

*riempie
il buffer*

*Se buffer è vuoto non c'è nulla da
Processo' consumatore*

```
elemento da_consumare;  
while (1) {  
    while (contatore == 0) ; /* do nothing */  
    da_consumare = buffer[out];  
    out = (out + 1) % BUFFER_SIZE;  
    contatore--;  
}
```

*Se eseguiti in concorrenza,
nessun produttore possa criticare*





Race condition – 1

- ❖ Sebbene siano corrette se si considerano separatamente, le procedure del produttore e del consumatore possono non “funzionare” se eseguite in concorrenza
- ❖ In particolare, le istruzioni:

```
contatore++;  
contatore--;
```

Se c'è interferimento a livello di codice macchina...

devono essere eseguite *atomicamente*

*a livello di codice macchina
sono 3 istruzioni*

*istruzioni eseguite
ATOMICAMENTE: Senza interruzione da parte dello scheduler*





Race condition – 2

*tipo atomico int, le oper. intere
non vengono interrotte*

- ❖ Un'operazione è **atomica** quando viene completata senza subire interruzioni
- ❖ Le istruzioni di aggiornamento del contatore vengono realizzate in linguaggio macchina come...

```
registro1 = contatore  
registro1 = registro1 + 1  
contatore = registro1
```

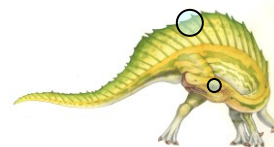
```
LOAD R1, CONT  
ADD 1, R1  
STORE R1, CONT
```

```
registro2 = contatore  
registro2 = registro2 - 1  
contatore = registro2
```

```
LOAD R2, CONT  
SUB 1, R2  
STORE R2, CONT
```

I due registri
possono
coincidere...

*per il context
switch...*





Race condition – 3

- ❖ Se il produttore ed il consumatore tentano di accedere al buffer contemporaneamente, poiché le istruzioni in linguaggio macchina possono risultare *interfogliate*, si ha una race condition

- ❖ La sequenza effettiva di esecuzione dipende da come i processi produttore e consumatore vengono schedati

- ❖ **Esempio:** inizialmente contatore=5 *l'on dit ne doit être v. il y a comme 2 interfogliure dépend de scheduler*
 T_0 produttore: $\text{registro}_1 = \text{contatore}$ ($\text{registro}_1 = 5$)

T_1 produttore: $\text{registro}_1 = \text{registro}_1 + 1$ ($\text{registro}_1 = 6$)

T_2 consumatore: $\text{registro}_2 = \text{contatore}$ ($\text{registro}_2 = 5$)

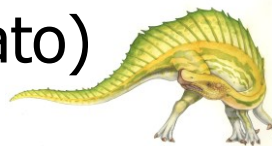
T_3 consumatore: $\text{registro}_2 = \text{registro}_2 - 1$ ($\text{registro}_2 = 4$)

T_4 produttore: $\text{contatore} = \text{registro}_1$ ($\text{contatore} = 6$)

T_5 consumatore: $\text{contatore} = \text{registro}_2$ ($\text{contatore} = 4$)

- ❖ **contatore** varrà 4, 5 o 6, mentre dovrebbe rimanere uguale a 5 (un elemento prodotto ed uno consumato)

5 solo se eseguite nel giusto ordine





Race condition – 4 *corse critiche*

- accesso incontrollato ai dati*
- ❖ **Race condition** — Più processi accedono in concorrenza e modificano dati condivisi; l'esito dell'esecuzione (il valore finale dei dati condivisi) dipende dall'ordine nel quale sono avvenuti gli accessi
 - ❖ Per evitare le corse critiche occorre che i processi concorrenti vengano sincronizzati
 - Tali situazioni si verificano spesso nei SO, nei quali diversi task compiono operazioni su risorse condivise
 - Problema particolarmente significativo nei sistemi multi-core, in cui i thread vengono eseguiti in parallelo su unità di calcolo distinte

a livello di kernel ancora più importante!

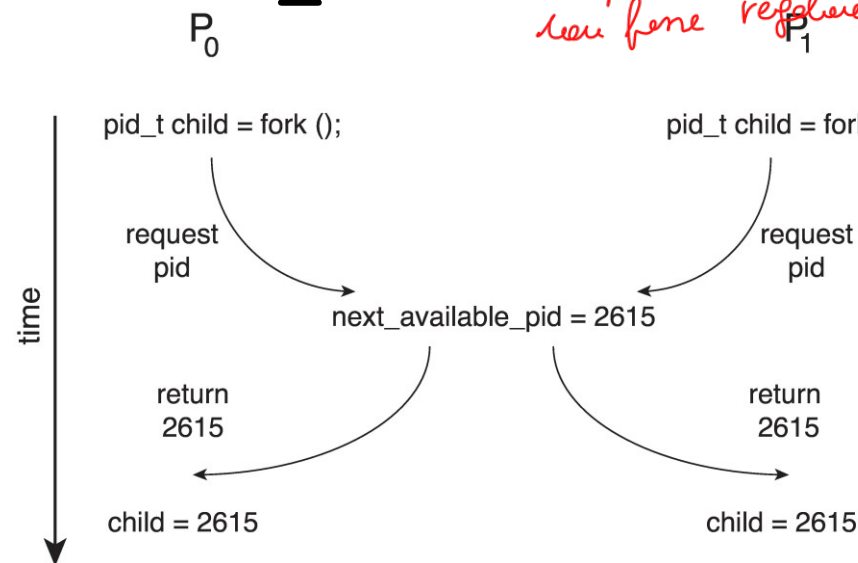
nei multicore ancora peggio, i thread condividono per loro natura la memoria





Race condition – 5

- ❖ I processi P_0 e P_1 (eseguiti in parallelo) creano processi figlio utilizzando la chiamata di sistema **fork()** *non possono essere P_1 e P_2*
- ❖ Race condition sulla variabile kernel che rappresenta il prossimo identificatore di processo disponibile (pid), next_available_pid *Se questa variabile non fosse regolamentata, assegna al padre*



2 processi distinti con lo stesso PID, nel PCB, ??

Sembra una politica di mutua esclusione

- ❖ Senza mutua esclusione, lo stesso pid potrebbe essere assegnato a due processi diversi





Il problema della sezione critica – 1

- ❖ n processi $\{P_0, P_1, \dots, P_{n-1}\}$ competono per utilizzare dati condivisi *conta anche leggerli: i dati, non solo scrivere*
- ❖ Ciascun processo è costituito da un segmento di codice, chiamato sezione critica (o *regione critica*), in cui accede a dati condivisi
- ❖ **Problema** — assicurarsi che, quando un processo accede alla propria sezione critica, a nessun altro processo sia concessa l'esecuzione di un'azione analoga
- ❖ ~~L'esecuzione di sezioni critiche da parte di processi cooperanti è~~ mutuamente esclusiva nel tempo

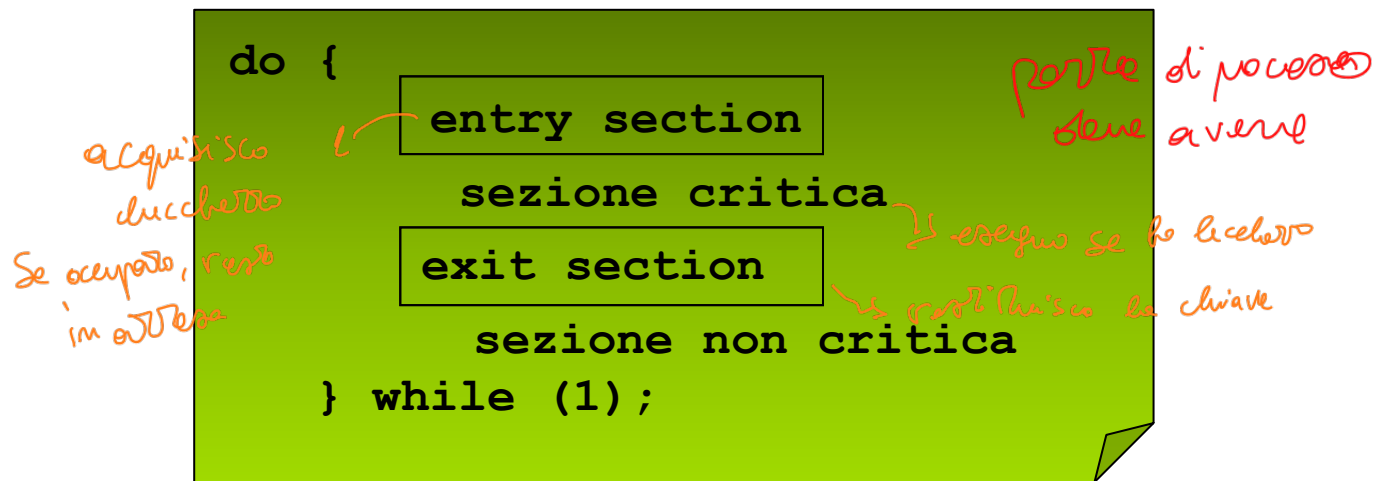
Serve un lucchetto





Il problema della sezione critica – 2

- ❖ **Soluzione** — progettare un protocollo di cooperazione fra processi:
- Ogni processo deve chiedere il permesso di accesso alla sezione critica, tramite una entry section
 - La sezione critica è seguita da una exit section
 - Il rimanente codice è non critico



Struttura generale del processo P_i





Soluzione al problema della sezione critica

Cosa devo garantire

1. **Mutua esclusione** — Se il processo P_i è in esecuzione nella sua sezione critica, nessun altro processo può eseguire la propria sezione critica
2. **Progresso** — Se nessun processo è in esecuzione nella propria sezione critica ed esiste qualche processo che desidera accedervi, allora la selezione del processo che entrerà prossimamente nella propria sezione critica non può essere rimandata indefinitamente
3. **Attesa limitata** — Se un processo ha effettuato la richiesta di ingresso nella sezione critica, è necessario porre un limite al numero di volte che si consente ad altri processi di entrare nelle proprie sezioni critiche, prima che la richiesta del primo processo sia stata accordata (politica *fair* per evitare la *starvation*)
 - Si assume che ciascun processo sia eseguito ad una velocità diversa da zero
 - Non si fanno assunzioni sulla velocità relativa degli n processi

Per evitare starvation

tutto ciò non vale per i sistemi real time





Sezione critica e kernel – 1

❖ In un dato istante, più processi in modalità utente possono richiedere servizi al SO

⇒ Il codice del kernel deve regolare gli accessi a dati condivisi *Non importante mantenere integrità di questi sez.*

❖ Esempio

- Una struttura dati del kernel mantiene una lista di tutti i file aperti nel sistema *record dei file aperti*
 - ▶ Deve essere modificata ogniqualvolta si aprono/chiudono file
 - ▶ Due processi che tentano di aprire file in contemporanea potrebbero ingenerare nel sistema una corsa critica





Sezione critica e kernel – 2

❖ **Kernel senza diritto di prelazione** — ^{Se l'utente kernel} i processi vengono eseguiti fino a quando non escono dalla modalità kernel, si bloccano o restituiscono volontariamente la CPU ^{i dati verranno modificati}

- Immuni dai problemi legati all'ordine degli accessi alle strutture dati del kernel: non si consente l'interruzione forzata di processi attivi in modalità di sistema
- Windows XP/2000 ^{molto lento, se}

❖ **Kernel con diritto di prelazione** — si consente la prelazione di processi in esecuzione in modalità kernel

- Critici per sistemi SMP, in cui due processi in modalità kernel possono essere eseguiti su processori distinti ^{su core diversi più veloci kernel in parallelo}
- Necessari in ambito real-time: permettono ai processi in tempo reale di far valere il loro diritto di precedenza
- Utili nei sistemi time-sharing: diminuiscono il tempo medio di risposta
- Linux, dal kernel versione 2.6





Soluzione di Peterson (1981)

*no nelle architetture
attuali*

i compilatori

*ora si permettono di inventare al problema della mutua esclusione
istruzioni che non ritengono correlati*

- ❖ Soluzione per due processi P_0 e P_1 ($i=1-j$)
- ❖ I due processi condividono due variabili:
 - **int turno**

Pseudocodice **boolean flag[2]**

barriera che un processo deve prima d'entrare

con entrambi gli elementi di **flag** inizializzati a **FALSE**

- ❖ La variabile **turno** indica il processo che "è di turno" per accedere alla propria sezione critica
- ❖ L'array **flag** si usa per indicare se un processo è pronto ad entrare nella propria sezione critica
 - **flag[i]=TRUE** implica che il processo P_i è pronto per accedere alla propria sezione critica





Algoritmo per il processo P_i

```
while(1) {
```

```
    flag[i] = true; P_i è Fair
```

```
    turn = j; Se il turno è di P_j, allora
```

```
    while (flag[j] && turn == j);
```

sezione critica

```
    flag[i] = false;
```

sezione non critica

```
}
```

P_0 : flag[0]=true

P_1 : flag[1]=true

P_1 : turn=0

P_1 : while(flag[0] && turn=0);

P_0 : turn=1 *← lo blocca P_1, entra*

P_0 : while(flag[1] && turn=1);

P_1 entra in sezione critica

P_1 : flag[1]=false

P_0 entra in sezione critica

...

se il turno è di P_j e P_i ha la flag → ATTENDO...

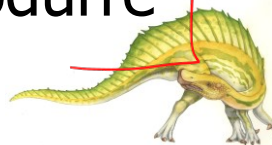
- ❖ **Soluzione** — Mutua esclusione garantita dal valore di **turn**; P_i entra nella sezione critica (progresso) al massimo dopo un ingresso da parte di P_j (attesa limitata): **alternanza stretta** *turno ordinato di P*





Soluzione di Peterson (cont.)

- ❖ La soluzione di Peterson è basata sul software, perché l'algoritmo garantisce la mutua esclusione senza richiedere alcun supporto speciale dal SO, né istruzioni hardware specifiche
- ❖ Sebbene rappresenti un utile approccio algoritmico, il funzionamento della soluzione di Peterson non è garantito sulle architetture attuali
- ❖ Capire perché l'approccio potrebbe non funzionare aiuta anche a comprendere le race condition
- ❖ Per migliorare le prestazioni, i processori e i compilatori possono riordinare le operazioni che non hanno interdipendenze
 - Nel caso di processi single-thread il risultato non cambia
- ❖ Tuttavia, nel multithreading, il riordino può produrre risultati inconsistenti o inaspettati





Esempio

- ❖ Due thread condividono i dati:

```
boolean flag = false;  
int x = 0;
```

- ❖ Il thread₁ esegue:

```
while (!flag);  
print x;
```

T₁ bloccato

devo aspettare che flag = true

L'output dovrebbe essere 100, ma se le operazioni di thread₂, che sono indipendenti, vengono riordinate, si può stampare 0

- ❖ Mentre il thread₂ assegna:

```
x = 100;  
flag = true;
```

- ❖ Quale sarà l'output prodotto?





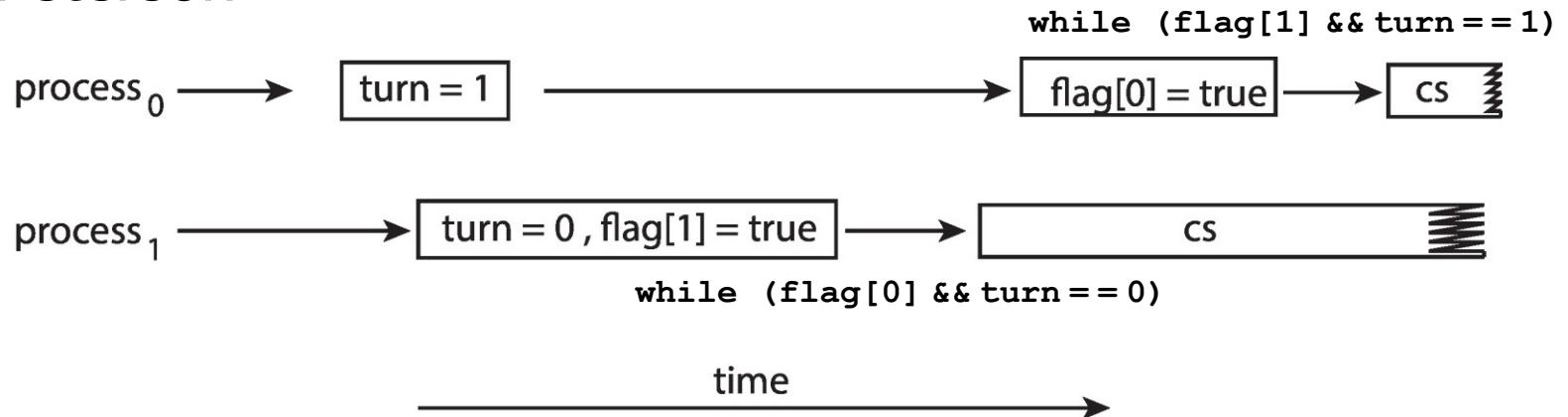
Soluzione di Peterson (cont.)

```
while(1) {  
    turn = j;  
    flag[i] = true;  
    while (flag[j] && turn == j);  
    sezione critica  
    flag[i] = false;  
    sezione non critica  
}
```

Processo P_i
entry section
exit section

P_0 : turn=1
 P_1 : turn=0
 P_1 : flag[1]=true
 P_1 : while(flag[0] && turn==0); *P_0 non fa più nulla, si ferma*
flag[0] è false → P_1 entra in sezione critica *flag[0]*
 P_0 : flag[0]=true *TURN = 0, P_0 passa sia il suo turno*
 P_0 : while(flag[1] && turn==1); *turno*
turn=0 → P_0 entra in sezione critica

❖ Effetto dello scambio di istruzioni nella soluzione di Peterson



❖ Si consente ai processi l'accesso concorrente alla sezione critica





Hardware di sincronizzazione – 1

Tecniche MW, inglobati anche più ad alto livello

- ❖ Molti sistemi forniscono supporto hardware per risolvere efficacemente il problema della sezione critica
- ❖ In un sistema monoprocesso ~~è sufficiente interdire le~~ interruzioni mentre si modificano le variabili condivise

- Il codice attualmente in esecuzione viene eseguito senza possibilità di prelazione

- Soluzione inefficiente nei sistemi multiprocessore

- ▶ SO non scalabili

ci sono i sottos. MW

- ❖ Tre forme di supporto hardware

- Barriere di memoria

- Istruzioni hardware

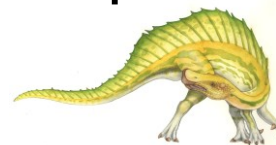
- Variabili atomiche *interni atomici*

1 Semaphore sono SW

Non sono effetti, contatori

- ❖ Queste primitive possono essere utilizzate direttamente come strumenti di sincronizzazione o costituire la base per costruire meccanismi di sincronizzazione più astratti

Vengono inglobati a più alto livello





Hardware di sincronizzazione – 2

- ❖ In generale, qualsiasi soluzione al problema della sezione critica richiede l'uso di un semplice strumento, detto *lock* (lucchetto)
 - Per accedere alla propria sezione critica, ~~il processo deve acquisire il possesso di un lock, che restituirà al momento della sua uscita~~

```
do {  
    acquisisce il lock  
    sezione critica  
    restituisce il lock  
    sezione non critica  
} while (1);
```

- L'implementazione del lock può essere a livello hardware o software





Barriere di memoria – 1

❖ Il modello di memoria è il modo in cui l'architettura di un computer determina quali garanzie relative alla memoria vengono fornite ai programmi applicativi

❖ Il modello di memoria può essere: *servisce il bisogno memoria da un punto all'altro*

Contenuto memoria deve venire sul campo

■ **Fortemente ordinato** — se una modifica alla memoria di un processore è immediatamente visibile a tutti gli altri *inaff'c.*

ho processori che dati coerenti non EFFICIENTE

■ **Debolemente ordinato** — se una modifica alla memoria di un processore può non essere immediatamente visibile a tutti gli altri

❖ Una **barriera di memoria** (o recinzione di memoria) è un'istruzione che forza la propagazione a tutti i processori di ogni cambiamento alla memoria *Forza la coerenza*

❖ Sono meccanismi di basso livello utilizzati dagli sviluppatori di SO per garantire la mutua esclusione sui dati del kernel





Barriere di memoria – 2

- ❖ Relativamente all'esempio precedente, potremmo aggiungere una barriera di memoria sia in thread_1 che in thread_2 per garantire che thread_1 stampi il valore 100
- ❖ thread_1 :

```
while (!flag)
    memory_barrier();
print x;
```

*ho paura che un processo
capibolare intervenga.
devo essere sicuro che x=100 sia
ovunque così.*

- ❖ thread_2 :

```
x = 100;
memory_barrier();
flag = true;
```

→ Nella soluzione di Peterson, inserire una barriera di memoria fra le due istruzioni che costituiscono la entry section garantisce l'esecuzione ordinata dei processi





Istruzioni hardware – 1

- ❖ Molte architetture attuali forniscono speciali istruzioni atomiche – cioè eseguibili come unità non prelazionabili e, sui multiprocessori, eseguite sequenzialmente – implementate in hardware

- permettono di controllare e modificare il contenuto di una parola di memoria (TestAndSet) *→ nome generico di una operazione, in HW*
- oppure, di scambiare il contenuto di due parole di memoria (CompareAndSwap) *usato per semafori ecc*

questioni atomiche

se eseguite dall'inizio alla fine





Istruzioni hardware – 2

```
boolean lock = FALSE;
```

```
boolean TestAndSet (boolean *object)
{
    boolean value = *object;
    *object = TRUE;
    return value;
}
```

*↳ forse l'indiretto
cambia valore del parametro*

Definizione di TestAndSet

- eseguita atomicamente
- ritorna il valore originale del parametro passato
- pone il nuovo valore del parametro a **TRUE**

```
do {
    while TestAndSet (&lock);
        sezione critica
    lock = FALSE;
        sezione non critica
} while(1);
```

↳ porta prima delle entry section

forse, se lock = true i processi succ. rimangono nel while
Realizzazione della mutua esclusione

↳ gli altri avranno TRUE, bloccati

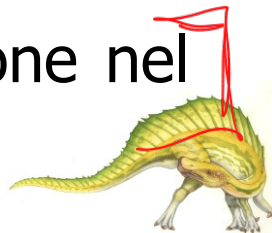
• solo il primo processo entra nella sezione critica, gli altri sono bloccati finché lock = FALSE





Variabili atomiche – 1

- ❖ In genere, le istruzioni hardware per garantire la mutua esclusione vengono utilizzate come blocchi costruttivi per altri strumenti di sincronizzazione
- ❖ Uno di questi sono le variabili atomiche, ^{Solo INTERI} normalmente definite su tipi di dati quali interi e booleani, che vengono aggiornate senza soluzione di continuità
- ❖ I SO che supportano le variabili atomiche forniscono anche tipi speciali di dati atomici e funzioni per l'accesso e la manipolazione di variabili di tali tipi
- ❖ Le variabili atomiche sono comunemente utilizzate nei SO e nelle applicazioni concorrenti, ma il loro uso è spesso limitato ad aggiornamenti di singoli dati condivisi, come contatori e generatori random
- ❖ Linux utilizza interi atomici per la sincronizzazione nel kernel
se ho problemi - conv con variabili atomiche, non risolvo, perche' risolvo il problema dell'incremento ma non quello del controllo





Variabili atomiche – 2

- ❖ **Esempio:** nel problema del produttore–consumatore con buffer limitato
 - Se **contatore** è una variabile atomica non vi sarà race condition all'atto dell'incremento/decremento
 - Una race condition invece si può avere nel controllo del ciclo **while**
 - Se due consumatori sono in attesa (nel loro ciclo **while**) e un produttore produce un elemento, entrambi potrebbero venire sbloccati e procedere al consumo di quell'unico elemento!





Lock mutex – 1

- ❖ Le soluzioni hardware al problema della sezione critica sono complicate e generalmente inaccessibili ai programmatori di applicazioni
- ❖ I progettisti di SO hanno implementato strumenti software per risolvere il problema: il più semplice è il lock mutex *m*utual *e*xclusion *m*utex: blocco / 2 sysCall
- ❖ Si proteggono le regioni critiche prima acquisendo (**acquire()**) e quindi rilasciando (**release()**) il lock
 - Si definisce una variabile booleana **available** che indica se il lock è disponibile o meno *lock variabile booleana*

```
acquire() {  
    while (!available) agosto se non  
        ; /* busy wait */  
    available = false;  
}
```

nella entry section

```
release() {  
    available = true;  
}
```

lock = true se è disp





Lock mutex – 2

processi che sono
a idle di CPU nel
non fare niente

DEVO GARANTIRE ATTESA
UTILE

Processo P →

```
do {  
    acquire()  
    sezione critica  
    release()  
    sezione non critica  
} while (true);
```

① spinlock
variabile booleana
h. acquire(), release()

se ha l'altro... entra
ottenere blocco

se non lo faccio, spinto la strada
a quell'altro

❖ Le chiamate ad **acquire()** e **release()** devono essere atomiche

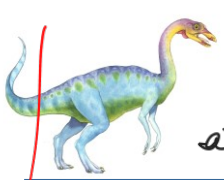
- Normalmente vengono implementate tramite istruzioni (atomiche) hardware

❖ La soluzione tramite lock impone l'**attesa attiva**: mentre un processo si trova nella propria sezione critica, ogni altro processo che ne tenti l'accesso deve ciclare effettuando la chiamata ad **acquire()**

- Per questo motivo si parla anche di **spinlock**

rimane nel ciclo
in attesa





lo fa c'è solo
 se il tempo di attesa
 arriva dura più di 2 context
 switch

Sono 2 context switch

Semafori – 1

Se durano più, lo kernel blocca

non voglio aver
 attesa in attesa
 voglio che i thread
 vengono rischedolati

variabile
 intera

- ❖ I **semafori** rappresentano un approccio più sofisticato alla sincronizzazione fra processi
- ❖ Il semaforo **S** è una ~~variabile intera~~ ^S
- ❖ Si può accedere al semaforo **S** (dopo l'inizializzazione) solo attraverso due operazioni indivisibili (operazioni atomiche, primitive) **wait()** e **signal()** (originariamente in olandese: **P** per *proberen*/verificare e **V** per *verhogen*/incrementare)

op.
 atomiche

```
wait(S) {
    while S <= 0
        ; // busy waiting
    S--;
}
```

Spinlock

ripete > solo un valore positivo

```
signal(S) {
    S++;
}
```

② Semafori, var. intera
 wait(), signal





Semafori – 2

- ❖ Tutte le modifiche al valore del semaforo contenute nelle operazioni **wait()** e **signal()** devono essere eseguite in modo indivisibile *atomico*
 - Mentre un processo cambia il valore del semaforo, nessun altro processo può contemporaneamente modificare quello stesso valore
 - Nel caso di **wait()** devono essere effettuate atomicamente sia la verifica del valore del semaforo che il suo decremento





Sezione critica con n processi

❖ Variabili condivise:

```
semaphore mutex = 1; // inizialmente mutex = 1
```

❖ Processo P_i :

```
do {  
    entry Section  
    wait(mutex);  
    sezione critica  
    signal(mutex); exit  
    sezione non critica  
} while (1);
```

quanti sono semafori critici

Gli n processi condividono un semaforo, **mutex** per mutual exclusion





Definizione e uso dei semafori – 1

- ❖ **Semaforo contatore:** intero che può assumere valori in un dominio non limitato *← # delle istanze disponibili di quella risorsa*
- ❖ **Semaforo binario:** assume soltanto i valori 0 e 1
- ❖ Permettono l'approccio a diversi problemi di sincronizzazione
- ❖ Siano P_1 e P_2 due processi che contengono, rispettivamente due frammenti di codice C_1 e C_2 che devono essere eseguiti in sequenza
⇒ Si crea il semaforo **synch** inizializzato a 0 *se ho n risorse*

```
P1:      sync = 0  
        C1;  
        signal(synch);  
P2: ← P2 aspetta, sync = 0  
        wait(synch);  
        C2;
```

*tutti i processi che vogliono
una risorsa fanno un wait,
decrementano il semaforo*

*processi che trovano semaforo a 0,
attestano*

*nesso su ciò che
sincronizza i processi
vedo che blocco P₁ venga
eseguito prima di P₂*

sync





Definizione e uso dei semafori – 2

❖ Più in dettaglio...

■ Semaforo contatore

- ▶ Trova applicazione nel controllo dell'accesso a risorse presenti in un numero finito di esemplari
- ▶ Il semaforo è inizialmente impostato al numero di risorse disponibili
- ▶ I processi che desiderano utilizzare un'istanza della risorsa, invocano una `wait()` sul semaforo, decrementandone così il valore
- ▶ I processi che rilasciano una risorsa, invocano `signal()`, incrementando il valore del semaforo
- ▶ Quando il semaforo vale 0, tutte le istanze della risorsa sono allocate e i processi che le richiedono devono sospendersi sul semaforo

■ Semaforo binario — simile al lock mutex

- ▶ Utilizzabili in maniera pressoché interscambiabile

*Spinlock, però a vuoto mg con
context switch*





Esempio

- ❖ Si consideri il seguente programma concorrente:

Variabili condivise

```
semaphore s1=0, s2=0;
```

processo P1

```
{  
    signal(s1);  
    wait(s2);  
    print "P"  
    wait(s2); S2=2  
    print "R"  
    signal(s1);  
}
```

processo P2

```
{  
    semforo e non  
    signal(s2); eseguito  
    wait(s1);  
    print "A";  
    signal(s2);  
    wait(s1); parte P1, S1=0  
    print "I";  
}
```

- ❖ Descrivere il comportamento ed i possibili output dell'esecuzione parallela di P1 e P2

A

*PM
PARI*

si può essere





Esempio (cont.)

- ❖ Inizialmente uno qualsiasi tra P1 e P2 può essere selezionato per l'esecuzione della prima istruzione
- ❖ Il processo P1 deve attendere l'esecuzione di `signal(s2)` da parte di P2 e viceversa P2 attende l'esecuzione di `signal(s1)` da parte di P1
- ❖ L'esecuzione delle prime due stampe "P" in P1 e "A" in P2 può avvenire in qualsiasi ordine; dopo aver stampato "P", P1 attende che P2 esegua `signal(s2)` e quindi stampa "R" e "segnala" sul semaforo `s1` sul quale attende P2
- ❖ Solo dopo la `signal(s1)` eseguita da P1, P2 può stampare "I"
- ❖ I possibili output sono quindi due: la stringa "PARI" e la stringa "APRI"





Implementazione dei semafori

- ❖ Dato che occorre garantire che due processi non eseguano mai `wait()` e `signal()` sullo stesso semaforo allo stesso istante (ovvero occorre garantire l'atomicità delle due operazioni)
 - ⇒ L'implementazione dei semafori impone che `wait()` e `signal()` vengano eseguite esse stesse all'interno di una sezione critica ⇒ possibile il busy waiting
 - Codice di implementazione molto ridotto (<10 istruzioni)
 - Brevi attese se la sezione critica è raramente occupata
- ❖ Per rendere le primitive semaforiche atomiche occorre:
 - Sui monoprocessori, disabilitare le interruzioni
 - Impossibile sui multiprocessori → alternative all'uso dei semafori, costruite sulla base dello specifico hardware per la sincronizzazione





Busy waiting

- ❖ Viceversa, con i semafori spinlock, si crea una situazione di *busy waiting* (attesa attiva) dei processi in attesa ad un semaforo
 - ⇒ Poiché i processi possono passare anche molto tempo all'interno delle loro sezioni critiche, il semaforo spinlock può provocare notevole spreco di CPU da parte dei processi in attesa sullo stesso semaforo
- ❖ In altre parole... mentre un processo si trova nella propria sezione critica, qualsiasi altro processo che tenti di entrare nella sezione critica si trova nel ciclo del codice della entry section
- ❖ Il *busy waiting* o *attesa attiva* costituisce un problema per i sistemi multiprogrammati, perché la condizione di attesa spreca cicli di CPU che altri processi potrebbero sfruttare produttivamente
- ❖ L'attesa attiva è vantaggiosa solo quando è molto breve perché, in questo caso, può evitare i context switch





Da notare...

❖ Un lock è **conteso** se un thread si blocca mentre tenta di acquisirlo; i lock contesi possono essere soggetti ad una contesa elevata (un numero grande di thread che tentano di acquisire il lock contemporaneamente) o moderata (un numero piccolo di thread si contendono il lock) $\Rightarrow$ i lock molto contesi tendono a ridurre le prestazioni complessive delle applicazioni concorrenti

❖ Gli spinlock sono il meccanismo di locking classico sui sistemi multiprocessore, quando il lock deve essere mantenuto per un breve periodo di tempo

- Quanto breve?
- Poiché l'attesa su un lock richiede due cambi di contesto (uno per spostare il thread in esecuzione nello stato di attesa ed uno per ripristinare il thread in attesa quando il lock diventa disponibile) $\Rightarrow$ usare uno spinlock quando il lock deve essere trattenuto per una durata inferiore a due cambi di contesto

*Se la contesa è alta,
usa altri metodi per non
spremere cicli di CPU*

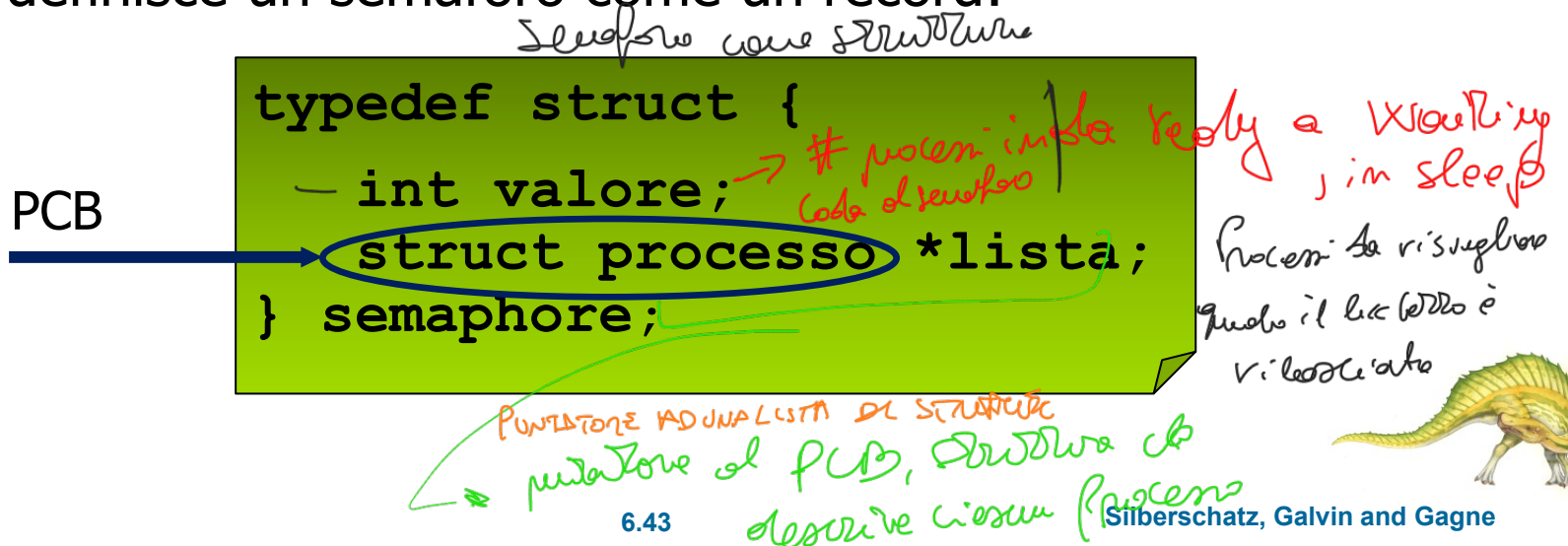




Come evitare il busy waiting – 1

- ❖ Per evitare di lasciare un processo in attesa nel ciclo while si può ricorrere ad una definizione alternativa di semaforo
 - Associata ad ogni semaforo, vi è una coda di processi in attesa
 - La struttura semaforo contiene:
 - ▶ un valore intero (numero di processi in attesa)
 - ▶ un puntatore alla testa della lista dei processi

⇒ Si definisce un semaforo come un record:





Come evitare il busy waiting – 2

❖ Si assume che siano disponibili due operazioni (fornite dal SO come system call):

- **block** – posiziona il processo che richiede di essere bloccato nell'opportuna coda di attesa, ovvero sospende il processo che la invoca (stato waiting) *da ready a waiting.*

- **wakeup (P)** – rimuove un processo P dalla coda di attesa e lo sposta nella ready queue (stato ready) *FIFO il primo sorge svegliato*

→ riguarda un singolo processo che verrà risvegliato





Come evitare il busy waiting – 3

- ❖ Le operazioni sui semafori possono essere definite come...

$S = 1$

è una struttura

nel caso di nuovo def. di

decremento

blocco

```
wait(semaphore *S) {  
    S->valore--;  
    if (S->valore < 0) {  
        aggiungi il processo P a S->lista;  
        block();  
    }  
}
```

È NEGATIVO!, ci sono processi in coda

valore del semaforo negativo

in valore negativo, # seu = # processi in coda

Si addiziona il processo

```
signal(semaphore *S) {  
    S->valore++;  
    if (S->valore <= 0) {  
        rimuovi il processo P da S->lista;  
        wakeup(P);  
    }  
}
```

in testa alla lista

si sveglia il processo in testa alla lista

S deve essere inizializzata a 1, il primo deve poterla





Semaforo vale 1 se non è stato usato

Come evitare il busy waiting – 4

- ❖ Mentre la definizione classica di semaforo ad attesa attiva è tale che il valore del semaforo non è mai negativo, il semaforo appena descritto (ed inizializzato a 1) può assumere valori negativi (nel campo **valore**)
- ❖ Se valore è negativo, |valore| è il numero dei processi che attendono al semaforo
 - Ciò avviene a causa dell'inversione dell'ordine delle operazioni di decremento e verifica del campo **valore** (del semaforo) nella **wait()**





Errori comuni

❖ Errori comuni nell'uso dei semafori:

- `signal(mutex) ... wait(mutex)` *Inverti, ... tutti i processi lo fanno*

inversione delle chiamate di sistema: più processi possono eseguire le proprie sezioni critiche in contemporanea

- `wait(mutex) ... wait(mutex)` *ciclo de morte*

doppia chiamata di una stessa system call: si genera un deadlock

- Omissione di `wait(mutex)`, `signal(mutex)` o di entrambe → violazione della mutua esclusione o stallo

auto wait, no MUTUA esclusione

no signal... processi bloccati





Esempio 1

- ❖ Si considerino due processi, un produttore P ed un consumatore C, ed un buffer con capacità pari ad 1

Buffer e' il solo condiviso

- ❖ Sia

```
semaphore empty=1;
```

l'unica variabile condivisa (oltre al buffer)

- ❖ Si considerino le seguenti porzioni di codice includenti la sezione critica per l'accesso al buffer da parte di P e C

Processo P

```
while(true) {
```

empty=1
p1

```
wait(empty);
```

p2

```
enter_item(item);
```

```
}
```

a questo punto la sezione critica e' finita

Processo C

```
while(true) {
```

c1

```
remove_item(item);
```

c2

```
signal(empty);
```

```
}
```

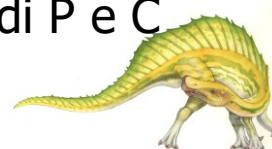
non e' la sezione critica

non e' la sezione critica

non e' la sezione critica

- ❖ Supponendo che le istruzioni p1 e p2 vengano eseguite per prime, il sistema si comporta correttamente come un produttore-consumatore? Descrivere, se esiste, una sequenza di esecuzione delle istruzioni di P e C non corretta

*Se viene eseguito prima C
causerebbe un deadlock che non c'è*





Esempio 1 (cont.)

- ❖ Il comportamento non è corretto dato che il protocollo per l'accesso ai dati condivisi (entry–exit section) non viene rispettato
- ❖ Sequenza scorretta
 - $p1, p2, c1, c2, c1$: attenzione, il consumatore consuma da un buffer vuoto! *se viene eseguito più C_1*





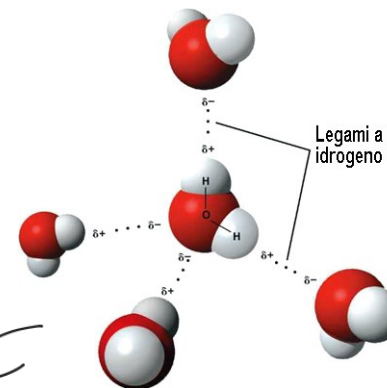
Esempio 2

- ❖ Il problema delle molecole di H_2O – Un sistema è costituito da due processi concorrenti: un processo costruttore di atomi di idrogeno e un processo costruttore di atomi di ossigeno; ogni processo ciclicamente produce un atomo e poi invoca una procedura comune (nell'esempio è una procedura di stampa)
- ❖ Si vuole che l'attività dei due processi sia sincronizzata in modo tale che le chiamate di questa procedura siano eseguite con gli input nell'ordine HHOHHOHHO...
- ❖ Siano

semaphore Hy=0, Oxy=2;

le variabili condivise *Problema di sincronizzazione*

6.50





Esempio 2 (cont.)

processi concorrenti: servono
elencati

- ❖ I processi Idrogeno ed Ossigeno possono essere descritti come segue:

Processo Idrogeno

```
{
while(1)
{
    <make a hydrogen atom>
    wait(Oxy); Oxy = 1
    write(H); H
    signal(Hy); Hy = 1
}
}
```

Processo Ossigeno

```
{
while(1)
{
    <make an oxygen atom>
    wait(Hy);
    wait(Hy);
    write(O);
    signal(Oxy);
    signal(Oxy);
}
}
```





Semafori e monitor

*non solo processo alla volta
processo eccedere il monitor*

- ❖ Benché i semafori costituiscano un meccanismo pratico ed efficace per la sincronizzazione dei processi, il loro uso scorretto può generare errori difficili da individuare, in quanto si manifestano solo in presenza di particolari sequenze di esecuzione
- ❖ I monitor semplificano molto la gestione della mutua esclusione (minori possibilità di errore)
- ❖ Sono veri costrutti, non funzioni di libreria $\Rightarrow$ modifica dei compilatori
- ❖ Problema comune sia con i semafori che con i monitor: è necessario avere memoria condivisa (costrutti non applicabili in sistemi distribuiti)





I monitor – 1

- ❖ Il monitor è un costrutto di alto livello (un ADT, per *Abstract Data Type*) che fornisce un meccanismo efficiente per la sincronizzazione dei processi *dati da istanze e incapsola i metodi che operano*
- ❖ Incapsula dati mettendo a disposizione metodi per operare su di essi
- ❖ La rappresentazione di un tipo monitor è costituita da:
 - dichiarazioni di variabili i cui valori definiscono lo stato di un'istanza del tipo
 - procedure o funzioni che realizzano le operazioni sulle variabili definite nel monitor
- ❖ Supportati (implementati in certa misura) da C#, Java, Concurrent Pascal





I monitor – 2

```
monitor monitor-name
{
    // dichiarazione delle variabili condivise
    // da monitor

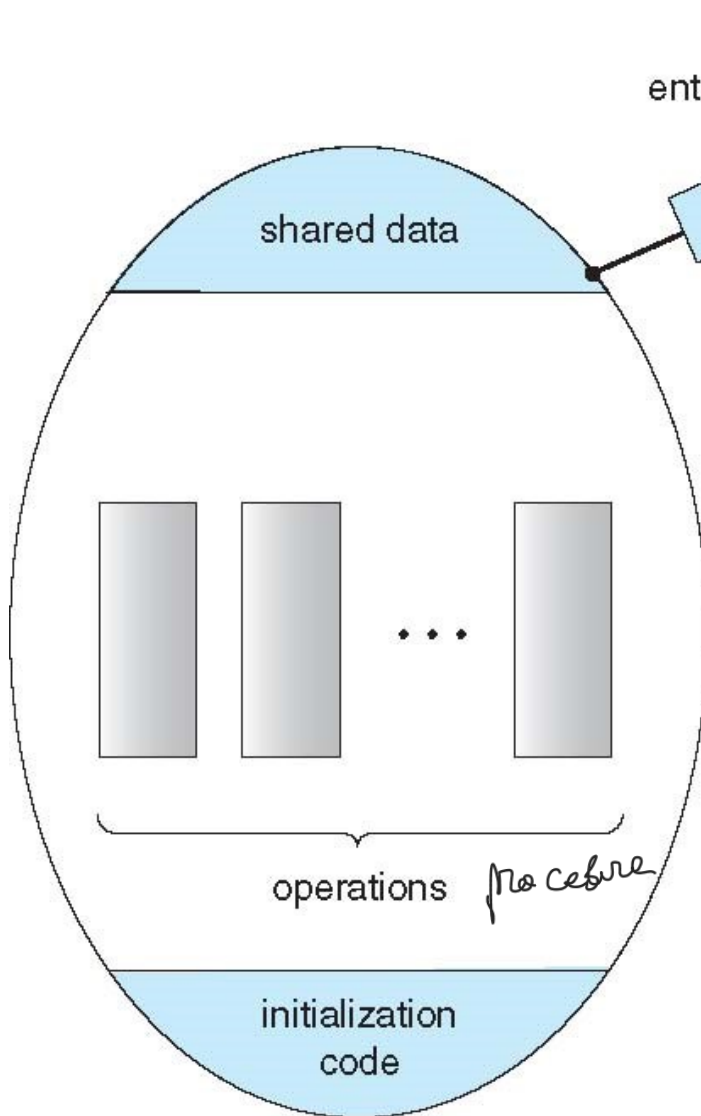
    function  $P_1$  (...) {
        . . .
    }
    function  $P_2$  (...) {
        . . .
    }
    function  $P_n$  (...) {
        . . .
    }
    {
        // codice di inizializzazione
    }
}
```

processi





I monitor – 3



*Cosa si può fare: da
vegliare accettare
ai dati, no
accettare
no da usare*

Il costrutto monitor assicura che al suo interno possa essere attivo un solo processo
➡ non si deve codificare esplicitamente il vincolo di mutua esclusione





I monitor – 4

- ❖ Per effettuare l'allocazione di risorse tramite monitor, un processo deve richiamare una routine interna al monitor
 - ❖ La mutua esclusione è rigidamente forzata ai confini del monitor stesso, dove può accedere un solo processo alla volta
 - ❖ Tuttavia, tale definizione di monitor non è sufficientemente potente per modellare alcuni schemi di sincronizzazione
- ⇒ Per permettere ad un processo di attendere dopo l'ingresso al monitor, causa occupazione della risorsa richiesta, si dichiarano apposite variabili *condition*

```
condition x,y;
```





I monitor – 5

- ❖ Le variabili condition possono essere usate solo in relazione a specifiche operazioni **wait()** e **signal()**

- L'operazione

x.wait(); *esce dal monitor*

fa sì che il processo chiamante rimanga sospeso fino a che un diverso processo non effettui la chiamata

x.signal();

- L'operazione **x.signal()** attiva esattamente un processo sospeso; se non esistono processi sospesi l'operazione **signal()** non ha alcun effetto *se no coda su una variabile condition*

*un processo in exec. di un monitor può
interrompersi in attesa di un evento*





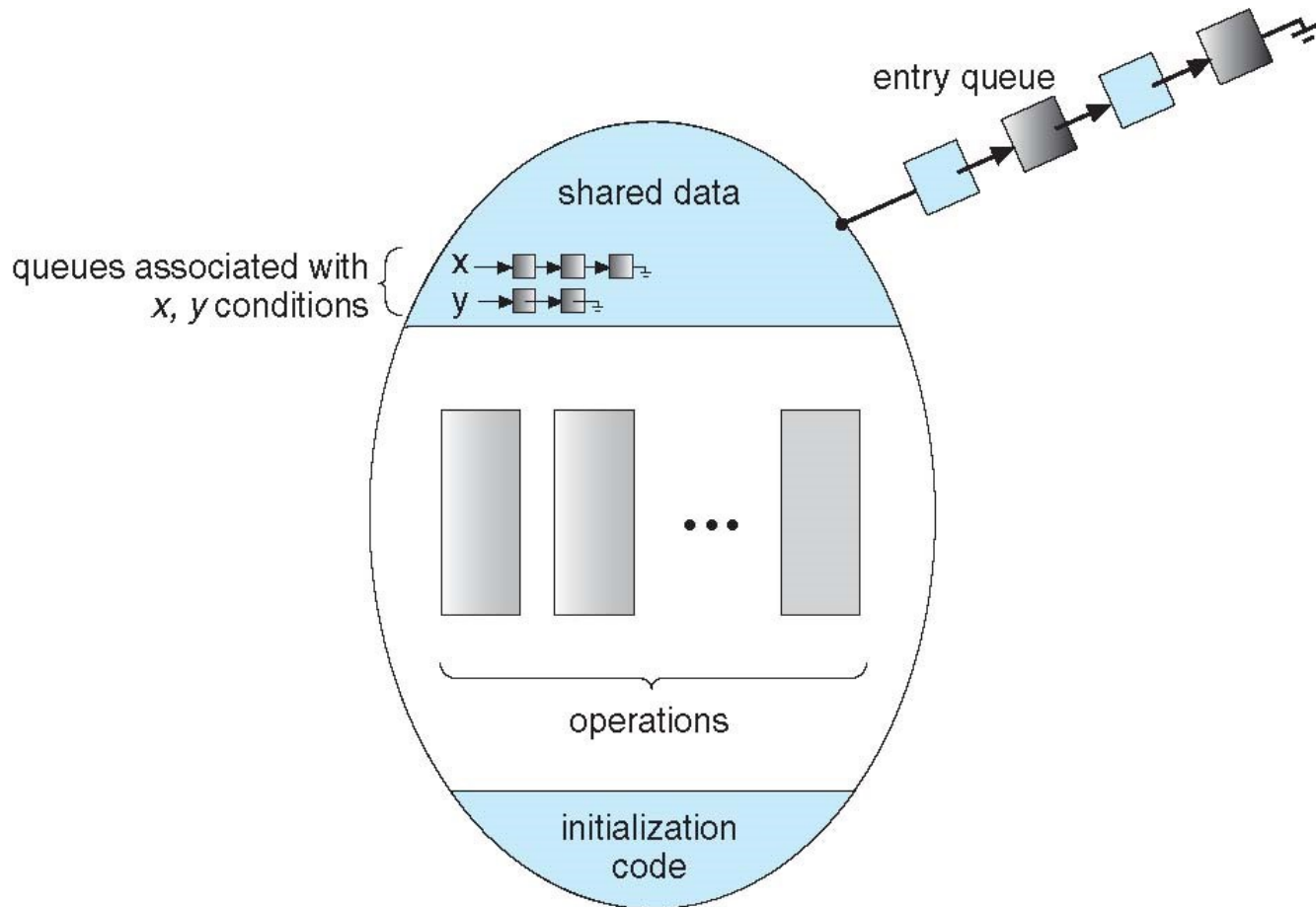
I monitor – 6

- ❖ Un processo utilizza una variabile condition (interna al monitor) per attendere il verificarsi di una certa condizione, uscendo dal monitor nell'attesa
- ❖ Un monitor associa una diversa variabile condition ad ogni situazione che potrebbe determinare l'attesa di un processo
- ❖ Un processo può uscire dal monitor mettendosi in attesa su una variabile condition o completando l'esecuzione del codice protetto dal monitor





Monitor con variabili condition





I monitor – 7

- ❖ Le variabili condition si differenziano dalle variabili convenzionali poiché ognuna di esse ha una coda associata
 - Un processo che richiama una `wait()` su una variabile condition particolare è posizionato nella sua coda e, finché vi rimane, è al di fuori del monitor, in modo che un altro processo possa entrarvi e richiamare una `signal()`
 - Un processo che richiama una `signal()` su una variabile condition particolare fa sì che un processo in attesa su quella variabile sia rimosso dalla coda associata e possa accedere al monitor
- ❖ Gestione FIFO o a priorità delle code dei processi
 - `conditional-wait` si ottiene con il costrutto `x.wait(c)`, dove `c` è un numero che rappresenta la priorità
 - Processi a più alta priorità vengono schedulati per primi

*Processo monitor
parametri*

priorità

priorità del processo che ha invocato la wait





Semantica delle operazioni su variabili condition

- ❖ Se il processo P invoca `signal()` su una variabile condition ed esiste un processo Q in coda su quella variabile
 - **Segnalare ed attendere:** nel monitor entra Q e P attende (fuori dal monitor) che Q abbia terminato o si metta in attesa su una condition
 - **Segnalare e proseguire:** Q attende che P lasci il monitor o si metta in attesa su una condition
- ❖ Pros & cons
 - Dato che P era già in esecuzione nel monitor **segnalare e proseguire** sembra più ragionevole
 - Però... quando P termina, la condizione attesa da Q potrebbe non essere più valida
- ❖ Soluzione adottata da Concurrent Pascal: P deve eseguire la `signal()` come ultima istruzione prima di uscire dal monitor
- ❖ Variabili condizionali implementate in C# e Java





Esempio 1

Singola risorsa

- ❖ Allocazione di una risorsa a singola istanza tra N processi in competizione, che ne specificano il tempo massimo di utilizzo

wait(time)

```
monitor ResourceAllocator
```

```
{  
    boolean busy; variabile condivisa del monitor  
    condition x;
```

```
    void acquire(int time) {  
        if (busy) se risorsa è occupata, il processo si mette in attesa  
            x.wait(time); specificando il tempo che vuole o meno
```

```
        busy = TRUE; se non è occupata, la occupa
```

```
    }  
    void release() {  
        busy = FALSE; si rilascia
```

```
        x.signal(); libero il processo che aspetta per nuovo tempo
```

```
    initialization code() {
```

```
        busy = FALSE;
```

risorsa disponibile

```
}
```





Esempio 1 (cont.)

- ❖ Se R è un'istanza del tipo **ResourceAllocator**, il protocollo di accesso al monitor per l'uso della risorsa per un tempo t è:

invoco i metodi

```
R.acquire(t);  
    ...  
    access the resource;  
    ...  
R.release();
```

*LA Risorsa è
ESTERNA AL
MONITOR*

*Se un processo
esterno accede
senza monitor...
contesa*

*Dovrei inserire la risorsa del MONITOR
ma non più schedata da SO*





Esempio 1 (cont.)

- ❖ Sfortunatamente, il concetto di monitor non può garantire che la sequenza di acquisizione della risorsa sia rispettata
 - ⇒ Possibile accesso senza permesso o non rilascio della risorsa contesa
- ❖ Soluzione: inclusione della “risorsa” all’interno del monitor
 - Ma, in questo caso... lo scheduling sulla coda dei processi che si contendono la risorsa non è più gestito dal SO, ma coincide con lo scheduling (la politica di gestione) di accesso al monitor





Liveness

low evitoto starvation

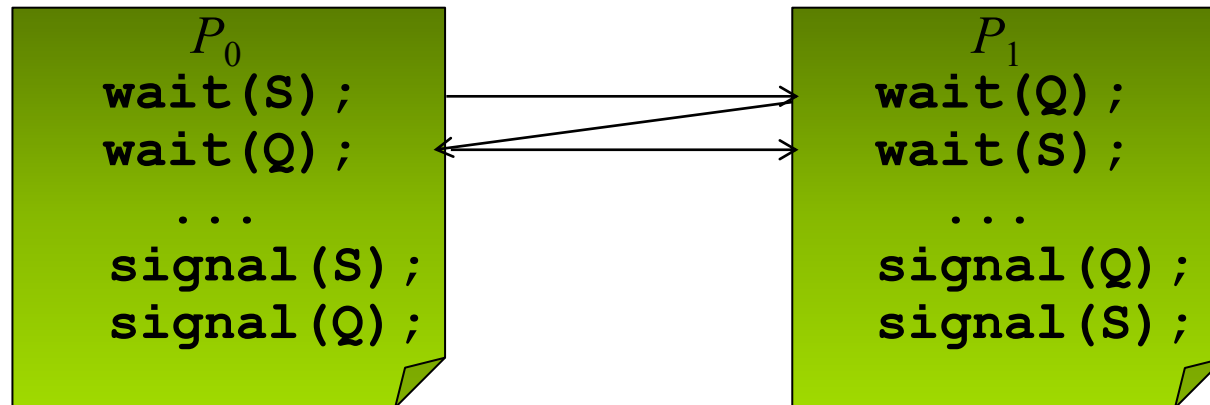
- ❖ I processi potrebbero dover attendere indefinitamente durante il tentativo di acquisire uno strumento di sincronizzazione come un lock mutex o un semaforo
- ❖ L'attesa indefinita viola i requisiti di progresso e attesa limitata
- ❖ Il concetto di **liveness** fa riferimento a un insieme di proprietà che un sistema deve soddisfare per garantire che i processi progrediscano
- ❖ L'attesa indefinita è un esempio di fallimento della liveness





Deadlock

- ❖ La realizzazione di un semaforo con coda di attesa può condurre a situazioni in cui ciascun processo attende l'esecuzione di un'operazione `signal()`, che solo uno degli altri processi in coda può causare
- ❖ Più in generale, si verifica una situazione di *deadlock*, o *stallo*, quando due o più processi attendono indefinitamente un evento che può essere causato soltanto da uno dei processi in attesa
- ❖ Siano `S` e `Q` due semafori inizializzati entrambi ad 1





Starvation

- ❖ *Starvation* – Un processo può attendere per un tempo indefinito nella coda di un semaforo senza venir mai rimosso
- ❖ L'attesa indefinita (senza blocco) si può verificare qualora i processi vengano rimossi dalla lista associata al semaforo in modalità LIFO (*Last-In-First-Out*) i più non vengono mai serviti
- ❖ Viceversa, per aggiungere e togliere processi dalla "coda al semaforo", assicurando un'attesa limitata, la lista può essere gestita in modalità FIFO





Inversione di priorità – 1

- ❖ Quando: un processo a priorità più alta ha bisogno di leggere o modificare dati a livello kernel utilizzati da un processo, o da una catena di processi, a priorità più bassa

*es. processo a bassa priorità ha preso un lock
e non dato => il processo ad alta priorità deve attendere*

- ❖ Perché: poiché i dati a livello kernel sono tipicamente protetti da un lock, il processo a priorità maggiore dovrà attendere finché il processo a priorità minore non avrà finito di utilizzare le risorse





Inversione di priorità – 2

❖ Esempio

- Siano dati 3 task, T_1, T_2, T_3 , con rispettive priorità $p_1 > p_2 > p_3$
- Supponiamo che T_1 e T_3 debbano accedere alla risorsa a cui è associato il mutex S
Si considerano un semaforo, T_3 va prima
- Se T_3 inizia la sua fase di elaborazione ed esegue una **wait(S)** prima che T_1 inizi la propria, quando T_1 tenterà di eseguire **wait(S)** verrà bloccato per un tempo non definito, cioè fino a quando T_3 non eseguirà **signal(S)**
 - ▶ T_1 viene penalizzato a favore di T_3 a dispetto delle priorità $p_1 > p_3$ (**blocco diretto**)
- Se, prima che T_3 esegua **signal(S)**, inizia l'esecuzione di T_2 , T_3 verrà sospeso per permettere l'elaborazione di $T_2 \Rightarrow T_1$ dovrà attendere che anche T_2 finisca (**blocco indiretto**)
 T_2 ha priorità superiore T_3 che non accede al semaforo, variabili diverse, T_1 dovrà attendere

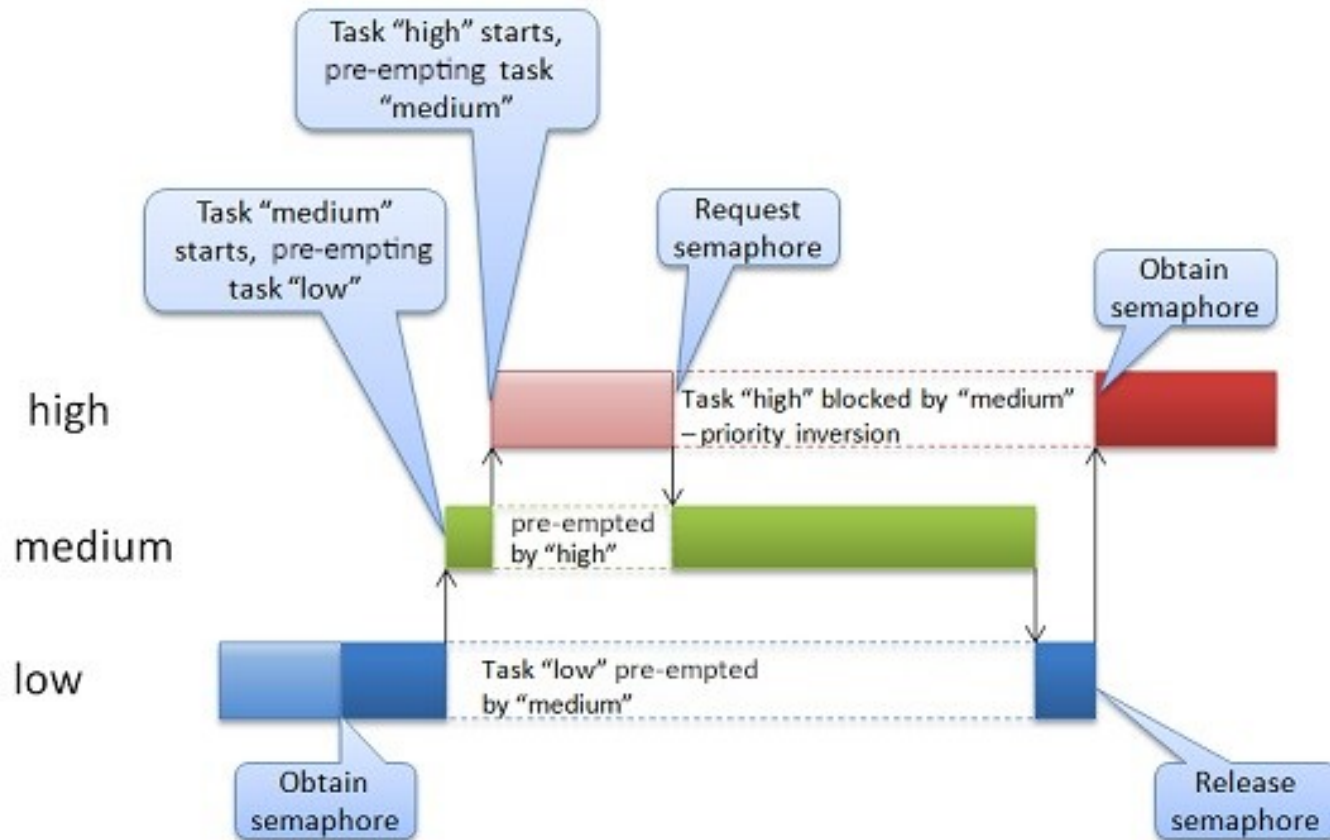




Solus. T_3 acquisisce priorità massima, la cessione di T_1

Inversione di priorità – 3

❖ Esempio (cont.)





Inversione di priorità – 4

❖ Soluzioni

- Limitare a due il numero di possibili valori di priorità (poco realistico)
- Implementare un **protocollo di ereditarietà delle priorità**
 - ▶ Tutti i processi che stanno accedendo a risorse di cui hanno bisogno processi con priorità maggiore ereditano la priorità più alta finché non rilasciano l'uso delle risorse contese
 - ▶ Quando hanno terminato, la loro priorità ritorna al valore originale

❖ Esempio famoso: Mars Pathfinder/Sojourner (1997)





Valutazione delle soluzioni – 1

- ❖ La scelta di un meccanismo per affrontare le race condition può influire sulle prestazioni del sistema
 - Le soluzioni basate sull'hardware (CAS – ottimistiche e difficili da sviluppare e da testare) permettono di costruire algoritmi privi di lock che forniscono protezione dalle race condition *Spec' lock, speco di CPU*
 - Gli interi atomici sono più leggeri e più appropriati dei lock mutex o dei semafori per aggiornamenti di singole variabili condivise (come i contatori) *Applic. limitato*
 - I ~~lock mutex~~ sono più semplici e comportano meno overhead dei semafori; ~~sono preferibili ai semafori binari~~ per proteggere l'accesso alla sezione critica
 - Tuttavia, nel controllo dell'accesso ad un numero limitato di risorse, un semaforo contatore è generalmente più adatto di un lock mutex

*Se ci
sono tanti
processi,
no*

*Soluzioni HW sono ottimistiche, ma sempre con rischio, no per sezioni
critiche, se usate allora o buona contro*



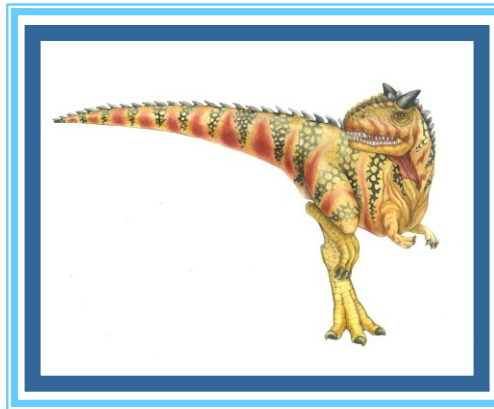


Valutazione delle soluzioni – 2

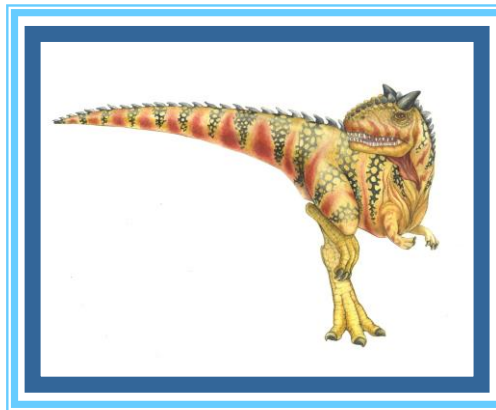
- ❖ L'attrattiva di strumenti di alto livello (semafori, monitor, variabili condizionali) si basa sulla loro semplicità d'uso
 - Possibile overhead significativo, in dipendenza dall'implementazione
 - Poco scalabili a situazioni a contesa elevata
- ❖ In futuro:
 - Progettazione di compilatori che generano codice più efficiente
 - Sviluppo di nuovi linguaggi che forniscano supporto alla programmazione concorrente
 - Miglioramento delle prestazioni delle librerie e delle API esistenti



Fine del Capitolo 6



Esempi di sincronizzazione





Obiettivi

- ❖ Illustrare i problemi classici di sincronizzazione — produttore–consumatore con buffer limitato, lettori–scrittori, e filosofi a cena
- ❖ Descrivere gli strumenti implementati da Linux per la sincronizzazione
- ❖ Illustrare come utilizzare *estensione libera posix* POSIX per risolvere i problemi di sincronizzazione fra processi





Sommario

- ❖ Problemi classici di sincronizzazione
- ❖ Sincronizzazione in Linux
- ❖ Sincronizzazione POSIX
- ❖ Approcci alternativi





Problemi classici di sincronizzazione

- ❖ Problema del produttore–consumatore con buffer limitato
- ❖ Problema dei lettori–scrittori
- ❖ Problema dei cinque filosofi
 - ⇒ Esempi di una vasta classe di problemi di controllo della concorrenza





Problema del buffer limitato – 1

- ❖ Variabili condivise: *e il buffer, accesso da entrambi*

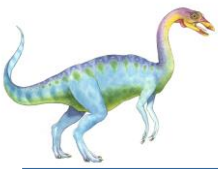
```
int n; dim - buffer
```

```
semaphore semafori contatori full, empty, semaforo binario mutex; regola accesso alla memoria
```

```
// inizialmente full = 0, empty = n, mutex = 1
```

- ❖ Il buffer ha n posizioni, ciascuna in grado di contenere un elemento
- ❖ Il semaforo **mutex** garantisce la mutua esclusione sugli accessi al buffer
- ❖ I semafori **empty** e **full** contano, rispettivamente, il numero di posizioni vuote ed il numero di posizioni piene nel buffer





Problema del buffer limitato – 2

❖ Struttura del processo produttore

```
while(true) {
```

```
    ...
```

```
    /* produce un elemento in next_produced */
```

```
    Se non ci sono pos. libere, lui non produce (empty < 0)  
    wait(empty);
```

Sezione critica

```
    ← wait(mutex);
```

```
    ...
```

```
    /* inserisce next_produced nel buffer */
```

```
    ...
```

```
    signal(mutex);
```

```
    signal(full);
```

```
}
```

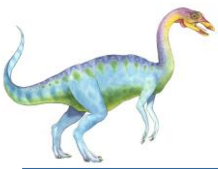
mutua esclusione

M posizioni sono libere (empty)

n --, sto per riepilogare un

full ++, = 1 il consumatore può consumare





Problema del buffer limitato – 3

❖ Struttura del processo consumatore : produce posizioni vuote nel buffer

```
while(true) {  
    WAIT FULL    wait(full);  
    SIGNAL EMPTY wait(mutex);  
    ...  
    /* sposta un elemento dal buffer in next_consumed */  
    ...  
    signal(mutex);  
    signal(empty);  
    ...  
    /* consuma un elemento in next_consumed */  
    ...  
}
```

Handwritten notes:

- se full < 0 lui produce* (next to `wait(full)`)
- se full = 0, non aveva nulla di consumare* (next to `wait(full)`)
- zona critica* (bracketed around the `wait(mutex)` and `signal(mutex)` lines)
- aumenta n posizioni vuote, così il produttore può produrre* (next to `signal(empty)`)

Da notare la perfetta simmetria fra i due processi: il produttore – che produce posizioni piene per il consumatore – e il consumatore – che produce posizioni vuote per il produttore

con scambi wait e full





Problema dei lettori–scrittori – 1

- ❖ Un insieme di dati (ad es., un file, una base di dati, etc.) deve essere condiviso tra più processi concorrenti che possono: *devo sapere a priori l'implementazione*
 - richiedere la sola lettura del contenuto... *processi possono anche accedere in concorrenza*
 - o l'accesso ai dati sia in lettura che in scrittura *devo aprire il mutex esclusivo*
- ❖ **Problema:** permettere a più lettori di accedere ai dati contemporaneamente; solo uno scrittore, viceversa, può accedere ai dati condivisi (accesso esclusivo)
- ❖ Variabili condivise (oltre ai dati):

```
semaphore mutex, rw_mutex; semaphore rw  
int read_count; # lettori che stanno accedendo  
// inizialmente mutex=1, rw_mutex=1, read_count=0
```

*per offuscamento
di read_count, contatore condiviso*





Problema dei lettori-scrittori – 2

Prima soluzione, per evitare starvation scrittori, no priorità a lettori ma a scrittori

```
while(true) {  
    wait(rw_mutex);  
  
    ...  
    /* scrittura */  
  
    ...  
  
    signal(rw_mutex);  
  
}
```

Scrittore

```
while(true) {  
    wait(mutex);  
    read_count++;  
    if (read_count == 1)  
        wait(rw_mutex);  
    ...  
    /* lettura */  
    ...  
    wait(mutex);  
    read_count--;  
    if (read_count == 0)  
        signal(rw_mutex);  
    signal(mutex);  
  
}
```

Handwritten notes:

- l'è a 1* (next to `wait(mutex);`)
- mutua esclusione per readcount* (next to `wait(mutex);`)
- Controlla solo per il 1° lettore* (next to `if (read_count == 1)`)
- il 2° scrittore non si mette* (next to `wait(rw_mutex);`)
- Se non ci sono scrittori, lui per* (next to `wait(rw_mutex);`)
- Se c'è uno scrittore, resta bloccato* (next to `wait(rw_mutex);`)
- mutua esclusione per scrittore e 1° lettore* (next to `wait(rw_mutex);`)
- quanto ci vuole* (next to `wait(mutex);` in the second block)
- non è l'ultimo lettore* (next to `if (read_count == 0)`)

Lettore





Problema dei lettori–scrittori – 3

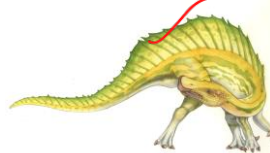
- ❖ Il semaforo **rw_mutex** è comune ad entrambi i tipi di processi
- ❖ Il semaforo **mutex** serve per assicurare la mutua esclusione all'atto dell'aggiornamento (o del controllo del valore) di **read_count**
- ❖ La variabile **read_count** contiene il numero dei processi che stanno attualmente leggendo l'insieme di dati
- ❖ Il semaforo **rw_mutex** realizza la mutua esclusione per gli scrittori e serve anche al primo e all'ultimo lettore che entra/esce dalla sezione critica
 - Non serve ai lettori che accedono alla lettura in presenza di altri lettori





Problema dei lettori–scrittori – 4

- ❖ Diverse varianti: la soluzione proposta riguarda il primo problema dei lettori–scrittori, in cui si richiede che nessun lettore attenda, almeno che uno scrittore abbia già ottenuto l'accesso ai dati condivisi
 - Priorità ai lettori $\Rightarrow$ gli scrittori possono essere soggetti a starvation
- ❖ Il secondo problema dei lettori–scrittori richiede invece che uno scrittore, se pronto, esegua il proprio compito di scrittura al più presto; in altre parole, se uno scrittore attende l'accesso all'insieme di dati, nessun nuovo lettore deve iniziare la lettura
 - Priorità agli scrittori $\Rightarrow$ i lettori possono essere soggetti a starvation





Lock di lettura–scrittura

- ❖ Alcuni sistemi dispongono di **lock di lettura–scrittura**

- In Linux, sia i lock mutex che i semafori hanno una variante per lettura/scrittura: **rwlock_t** e **struct rw_semaphore**

- ❖ È permesso a più processi di ottenere il lock in modalità lettura, ma un solo processo alla volta può acquisire il lock in modalità scrittura

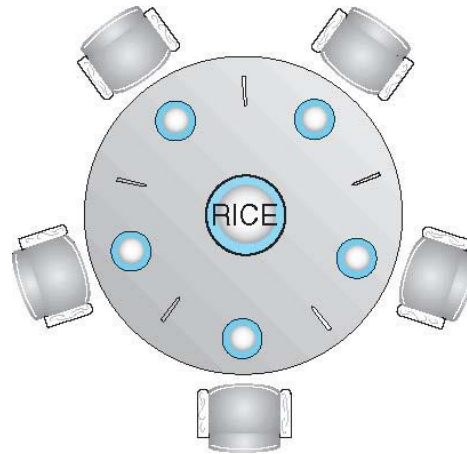
- ❖ Lock di lettura–scrittura utili:

- nelle applicazioni in cui è facile identificare processi che si limitano alla lettura dei dati condivisi e processi che si limitano alla scrittura degli stessi
- nelle applicazioni che prevedono più lettori che scrittori; i lock di lettura–scrittura sono più “pesanti” di lock mutex e semafori, ma forniscono la possibilità di eseguire molti lettori in concorrenza





Problema dei cinque filosofi – 1



- ❖ Cinque filosofi passano la vita pensando e mangiando, attorno ad una tavola rotonda
- ❖ Non interagiscono con i loro vicini; occasionalmente tentano di procurarsi due bacchette per mangiare
- ❖ Al centro della tavola vi è una zuppiera di riso e la tavola è apparecchiata con cinque bacchette

Problema di concorrenza





Problema dei cinque filosofi – 2

- ❖ Quindi... quando un filosofo pensa, non interagisce con i colleghi, quando gli viene fame, tenta di impossessarsi delle bacchette che stanno alla sua destra ed alla sua sinistra
- ❖ Il filosofo può appropriarsi di una sola bacchetta alla volta e non può strapparla dalle mani di un vicino
- ❖ Quando un filosofo affamato possiede due bacchette contemporaneamente, mangia; terminato il pasto, appoggia le bacchette e ricomincia a pensare
- ❖ Variabili condivise:
 - La zuppiera di riso (l'insieme dei dati)
 - Cinque semafori, ^{vettori} **bacchetta[5]**, inizializzati ad 1





Problema dei cinque filosofi – 3

❖ Codice per l' i -esimo filosofo

```
while(true) {  
    wait (bacchetta[i]);  
    wait (bacchetta[(i + 1) % 5]);  
  
    // mangia  
  
    signal (bacchetta[i]);  
    signal (bacchetta[(i + 1) % 5]);  
  
    // pensa  
  
}
```

ho bisogno di 2 bacchette { *se disponibile, la preleva*
luffer circolare, tonda
rilascio
Se tutti raccolgono la bacchetta è... DEADLOCK, STALLO

❖ Soluzione che garantisce che due vicini non mangino contemporaneamente, ma non riesce ad evitare lo stallo





Problema dei cinque filosofi – 4

❖ **Stallo:** i filosofi hanno fame simultaneamente ed ognuno si impossessa della bacchetta alla propria sinistra

❖ **Soluzioni**

- Solo quattro filosofi possono stare a tavola contemporaneamente
- Un filosofo può prendere le bacchette solo se sono entrambe disponibili (operazione da eseguire all'interno di una sezione critica)
- I filosofi di posto "pari" raccolgono prima la bacchetta alla loro sinistra, quelli di posto "dispari" la bacchetta alla loro destra

Stabilisco un protocollo ... almeno uno lo prenderà entrambi





alternativa con i monitor... no stallo su starvation

Problema dei cinque filosofi – 5

monitor filosofo

```
{
  Tipo dato enumerativo
  opzioni del monitor: pensare, affamato, mangiare
  enum{PENSA, AFFAMATO, MANGIA} stato[5]; stato, solo uno di questi 3 costanti.
  condition self[5]; variabili relative alla bacchetta

  void prende (int i) {
    stato[i] = AFFAMATO; fil. dichiara di avere fame
    test(i); controlla
    if (stato[i] != MANGIA) self[i].wait; si mette in attesa
  }
  Se da Test, non viene messo stato[i] = mangia, qualcuno aveva la bacchetta altrimenti ha acquisito la bacchetta

  void posa (int i) {
    stato[i] = PENSA; Stato di interfaccia con phi. alla
    // controlla i vicini a sinistra e a destra
    test((i+4) % 5);
    test((i+1) % 5); Se trovato, se loro hanno fame, si riprendono la bacchetta, con signal
  }
}
```

Se non includiamo il riso nelle variabili del monitor, mi serve un protocollo esterno di accesso al riso



Problema dei cinque filosofi – 6

```
void test (int i) {  
    if ( (stato[(i+4) % 5] != MANGIA) && controlla che il vicino di DX non stia mangiando  
        (stato[i] == AFFAMATO) &&  
        (stato[(i+1) % 5] != MANGIA) ) { vicino di SX non stia mangiando  
        stato[i] = MANGIA;  
        self[i].signal();  
    }  
}  
  
codice_iniz() { inizializza  
    for (int i = 0; i < 5; i++)  
        stato[i] = PENSA; all'inizio tutti e 5 pensano  
}  
}
```

da modo agli altri di prendere la forchetta

*ridare i posceti
Severi o non fa nulla*



Problema dei cinque filosofi – 7

- ❖ Il filosofo i -esimo invocherà le operazioni **prende()** e **posa()** nell'ordine:

```
filosofo.prende(i)
... ..
mangia
... ..
filosofo.posa(i)
```

*Ci si siede nel proprio posto,
che dall'esterno quello accanto
P. co. alla circolo d'opposto*

- ❖ La soluzione non provoca deadlock, ma la starvation è ancora possibile!

*Se altri processi non avevano
fatto, perdono il
loro turno*

*6 possibili
filari
esec.*

$A_1 A_2 B_1 B_2$ (13)
 $A_1 B_1 A_2 B_2$ (12)
 $A_1 B_1 B_2 A_2$ (11)
 $B_1 A_1 B_2 A_2$ (11)

 $B_1 B_1 A_2 B_2$ (12)
 $B_1 B_2 A_1 A_2$ (13)





Esempio 1

A_1, A_2, B_1, B_2 ✓3
 A_1, B_1, A_2, B_2 12
 B_1, B_2, A_1, A_2 13
 B_1, A_1, B_2, A_2 11
• B_1, A_1, B_2, B_1 ✓7
 A_1, B_1, B_2, A_2 11

- ❖ La seguente coppia di processi condivide l'insieme di variabili **counter**, **tempA** e **tempB**:

counter = 10

Processo A

```
...  
A1: tempA = counter + 1;  
A2: counter = tempA;  
...
```

Processo B

```
...  
B1: tempB = counter + 2;  
B2: counter = tempB;  
...
```

con **counter** inizializzato al valore 10. Quanti sono i possibili valori assunti da **counter** al termine dell'esecuzione di entrambi i processi? Modificare i processi **A** e **B**, inserendo opportune istruzioni **wait()/signal()** sul semaforo **synch** per ottenere il solo valore **counter=13**. Quale deve essere il valore iniziale di **synch**? *deve essere 1*

Se wait è molto usato





Esempio 1 (cont.)

- ❖ Si ottengono tre diversi valori per **counter**, in base all'ordine di esecuzione delle istruzioni, in particolare:

Processo A

```
...  
A1: tempA = counter + 1;  
A2: counter = tempA;  
...
```

Processo B

```
...  
B1: tempB = counter + 2;  
B2: counter = tempB;  
...
```

A1	A2	B1	B2:	counter = 13
A1	B1	A2	B2:	counter = 12
A1	B1	B2	A2:	counter = 11
B1	A1	B2	A2:	counter = 11
B1	A1	A2	B2:	counter = 12
B1	B2	A1	A2:	counter = 13





Esempio 1 (cont.)

❖ Considerando:

```
semaphore synch = 1
```

Processo A

...

— wait(synch);

A1: tempA = counter + 1;

A2: counter = tempA;

— signal(synch);

...

Processo B

...

— wait(synch);

B1: tempB = counter + 2;

B2: counter = tempB;

— signal(synch);

...

si ottiene che i processi **A** e **B** vengano eseguiti in sequenza (in ordine qualsiasi), producendo in stampa il solo valore 13





Esempio 2

Per compiti interpretare
codice concorsuale

- ❖ **Selvaggi a cena** – Una tribù di N selvaggi mangia in comune da una pentola che può contenere fino ad M (con $M < N$) porzioni di stufato di missionario; quando un selvaggio ha fame controlla la pentola: se non ci sono porzioni, sveglia il cuoco ed attende che questo abbia riempito di nuovo la pentola; se la pentola contiene almeno un pasto, se ne appropria; il cuoco controlla che ci siano delle porzioni e, se ci sono, si addormenta, altrimenti cuoce M porzioni e le serve ai cannibali in attesa. *risorse equivalenti*
tutte le M porzioni sono equivalenti

- ❖ Si descriva una soluzione al problema che impieghi i semafori *ci sono altre soluzioni con semafori contatori*

- ❖ Siano:

per esempio devo interpretare
semaphore mutex=1, pieno=0, vuoto=0;
due contatori
int porzioni=M;

le risorse condivise





Esempio 2 (cont.)



❖ I processi Cuoco e Cannibale sono:

Processo Cuoco

```
{  
    non c'è necessità all'inizio,  
    C'è solo la M Porzioni  
...  
while(1)  
{  
    wait(vuoto);  
    <riempi la pentola>  
    porzioni = M;  
    signal(pieno);  
}  
}
```

vuoto o pieno

*la porzione
il cannibale*

Processo Cannibale

```
{  
...  
wait(mutex);  
if (porzioni==0) {  
    signal(vuoto);  
    wait(pieno);  
}  
porzioni--;  
signal(mutex);  
<mangia porzione>  
...  
}
```

mutex

semplice

*hai solo
semfori contatori
ma 0/1*

*regola accen-
to alle porzioni
in maniera esclusiva*



Sincronizzazione in Linux – 1

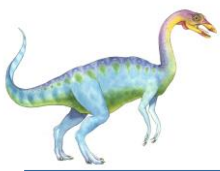
Strumenti per gestione accesso al core del kernel

- ❖ Prima della versione 2.6, Linux adoperava un kernel senza diritto di prelazione *se c'è un processo in exec, non veniva interrotto non l'uso per sleep, non scalabile*
 - Gli interrupt venivano disabilitati per ottenere sezioni critiche di breve durata
- ❖ Attualmente, Linux fornisce diversi meccanismi per la sincronizzazione nel kernel
 - **Lock mutex:** una variabile booleana che indica se il lock è disponibile, modificata tramite le chiamate di sistema `mutex_lock()` e `mutex_unlock()` *acquire il lock e rilasciare*
 - **Spinlock e semafori** (con variante lettore-scrittore)
 - **Interi atomici**, rappresentati mediante il tipo opaco **`atomic_t`** *opaco, kernel non vede come e' codificato il dato sono solo per operazioni specifiche*
 - ▶ Un tipo di dato la cui reale struttura non è esposta nell'interfaccia (non vengono forniti dettagli sui campi e, di conseguenza, l'accesso alle informazioni può avvenire solo tramite l'utilizzo di particolari funzioni)

linari
o variabili

posso effettuare
solo specifici
operazioni





semforo centrale, possono essere usati solo con la wait

Sincronizzazione in Linux – 2

*2003
inter
critical*

```
atomic_t counter;  
int value; questo da usare fare una interruzione  
atomic_set(&counter, 5); /* counter = 5 */ non possono  
atomic_add(10, &counter); /* counter = 15 */ COUNTER + 1  
atomic_sub(4, &counter); /* counter = 11 */  
atomic_inc(&counter); /* counter = 12 */  
value = atomic_read(&counter); /* value = 12 */
```

- ❖ Nel caso di interi atomici, tutte le operazioni matematiche vengono eseguite senza interruzioni
- ❖ Gli interi atomici sono particolarmente utili per gli aggiornamenti dei contatori *atomic counter menu*
 - Non risentono dell'overhead dei meccanismi di lock
- ❖ Insufficienti se la race condition può coinvolgere diverse variabili ⇒ strumenti di lock più sofisticati





Sincronizzazione in Linux – 3

questo e' servito a avere un lock ?

- ❖ Se il lock deve essere mantenuto per breve tempo:
 - sulle architetture SMP, il meccanismo fondamentale per la gestione della sezione critica è lo spinlock
 - Sui monoprocessori, come nel caso dei core embedded, si ricorre all'abilitazione/inibizione del diritto di prelazione (abilitazione/inibizione interruzioni)
 - Non è tuttavia possibile sottoporre a prelazione un task, attivo nel kernel, che sia possessore di un lock
- ❖ Quando sia necessario mantenere il lock più a lungo, si utilizzano lock mutex o semafori implementati senza busy waiting





Sincronizzazione in POSIX – 1

- ❖ La API Pthreads (lo standard POSIX) fornisce diversi tipi di lock e variabili condizionali per la sincronizzazione dei thread a livello utente
- ❖ API utilizzata su sistemi UNIX, Linux e MacOS *il kernel BSD e su kernel MAX*
- ❖ I lock mutex rappresentano la tecnica di sincronizzazione fondamentale
 - Un thread, *in questo caso condizione volatile* che sia in procinto di accedere alla propria sezione critica, si appropria del lock, salvo rimuoverlo all'uscita
- ❖ POSIX *sempre non prevede semafori* prevede due versioni di semafori (definiti nell'estensione POSIX SEM), con e senza nome *(come PIPE, con e senza nome)*
 - I semafori con nome possono essere utilizzati da processi non correlati, diversamente da quelli senza nome





Sincronizzazione in POSIX – 2

❖ Creazione e inizializzazione del lock

```
#include <pthread.h>

pthread_mutex_t mutex;

/* create and initialize the mutex lock */
pthread_mutex_init(&mutex, NULL);
```

Tipo predefinito per la dichiarazione di lock

Il primo parametro è un puntatore al mutex, con il secondo parametro posto a NULL si inizializza il lock con gli attributi di default

❖ Acquisizione e rilascio del lock

```
/* acquire the mutex lock */
pthread_mutex_lock(&mutex);

/* critical section */

/* release the mutex lock */
pthread_mutex_unlock(&mutex);
```

acquisizione
e rilascio
mutex

Tutte le funzioni di gestione dei lock mutex restituiscono 0 in caso di corretto funzionamento; viceversa, restituiscono un codice di errore diverso da 0

Restituiscono 0 se sono state eseguite correttamente





Semafori con nome in POSIX

❖ Creazione e inizializzazione del semaforo

pipe con nome
e semafori
con nome...
SONO FILE
di tipo con
nome

```
#include <semaphore.h>
```

```
sem_t *sem;
```

Tipo predefinito per la
dichiarazione di semafori

```
/* Create the semaphore and initialize it to 1.*/
```

```
sem = sem_open("SEM", O_CREAT, 0666, (1));
```

Se non c'è, crealo *0666* *1* *valore iniziale* *0666* *0666*

- ❖ Un diverso processo può riferirsi al semaforo usandone il nome, **SEM** (le successive chiamate a **sem_open** restituiscono un descrittore al semaforo esistente)
- ❖ Acquisizione e rilascio del semaforo

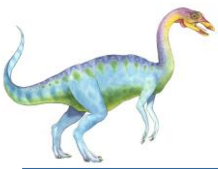
```
/* acquire the semaphore */  
sem_wait(sem);
```

```
/* critical section */
```

```
/* release the semaphore */  
sem_post(sem);
```

Tutte le funzioni di gestione dei semafori restituiscono 0 in caso di corretto funzionamento; viceversa restituiscono un codice di errore diverso da 0





Semafori senza nome in POSIX

```
#include<semaphore.h> Tipo e libreria  
sem_t sem;  
...  
/* crea il semaforo e lo inizializza a 1 */  
sem_init(&sem, 0, 1); oggetto che e' usato solo dal processo e ai suoi thread  
...
```

```
/* acquisisci il semaforo */  
sem_wait(&sem);  
  
/* sezione critica */  
  
/* rilascia il semaforo */  
sem_post(&sem);
```

Indica il livello di condivisione:
semaforo utilizzabile solo dai
thread appartenenti al processo
che ha creato il semaforo





Variabili condition in POSIX – 1

- ❖ Poiché POSIX è tipicamente utilizzato in C/C++, che non forniscono il tipo monitor, le variabili condition POSIX sono associate ai lock mutex per garantire la mutua esclusione *Non ha monitor, che serve in linguaggi che lo definiscono dato*
- ❖ Creazione e inizializzazione della variabile condition

que API Posix

*Variabili condition all'interno dei
blocchi lock mutex*

```
pthread_mutex_t mutex;  
pthread_cond_t cond_var;  
  
pthread_mutex_init(&mutex, NULL);  
pthread_cond_init(&cond_var, NULL);
```

Sono strutture

*Seleziona strutture
in file <pthread.h>
Perché i campi
con valori standard*





Variabili condition in POSIX – 2

- ❖ Thread in attesa che valga **a==b**

mi mette in attesa

```
pthread_mutex_lock(&mutex); lock. entry section
while (a != b)
    pthread_cond_wait(&cond_var, &mutex);
pthread_mutex_unlock(&mutex);
```

mi mette in attesa sulla variabile cond. var

2 case: attesa e mutex = 1 -> c'è anche il mutex

MUTEX TONDA PERCHÉ

Se un altro thread che sblocca la sezione come monitor

Rilascia il lock mutex, consentendo ad un altro thread di accedere al dato condiviso

Ci vuole un altro thread che cavi la situazione

- ❖ Thread che “sveglia” un altro thread in attesa sulla variabile condition

```
pthread_mutex_lock(&mutex);
a = b;
pthread_cond_signal(&cond_var);
pthread_mutex_unlock(&mutex);
```





Approcci alternativi – 1

❖ OpenMP *parallelizza codice*

- Include la direttiva

#pragma omp critical *segna una sezione critica*

che specifica che la successiva regione di codice è una sezione critica in cui solo un thread alla volta può essere attivo

- La direttiva per la sezione critica si comporta come un semaforo binario o un lock mutex (**counter** variabile condivisa)

```
void update (int value)
{
    #pragma omp critical
    {
        counter += value;
    }
}
```

sez critica



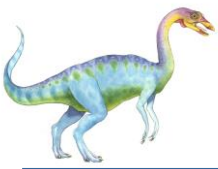


Approcci alternativi – 2

❖ Linguaggi di programmazione funzionali

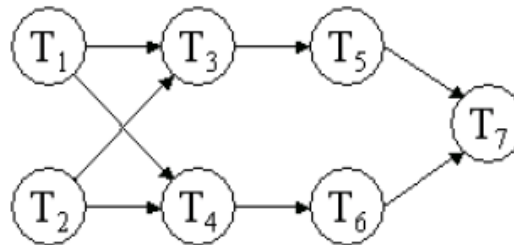
- Il flusso di esecuzione del programma assume la forma di una serie di valutazioni di funzioni matematiche
- La differenza fondamentale fra linguaggi procedurali *esecuzione sequenziale* e funzionali *variabili* è che i linguaggi funzionali non mantengono uno stato *quella le variabili sono immutabili*
- **Erlang, Scala, Haskell**
 - ▶ Erlang: notevole supporto alla concorrenza, particolarmente adatto in ambiente parallelo
 - ▶ Scala: orientato agli oggetti (in gran parte simile a Java e C#)
 - ▶ Haskell: supporta una semantica di tipo *lazy* in cui gli argomenti delle funzioni vengono valutati solo se e quando richiesto





Esercizio 1

- ❖ Si consideri il seguente grafo di precedenza fra processi:



- ❖ Ciascun processo viene eseguito in base allo schema:

```
processo Ti (i=1,...,7) {  
  attende l'esecuzione dei predecessori, se ce ne sono;  
  esegue il proprio compito;  
  segnala la terminazione ai successori, se ce ne sono;  
}
```

- ❖ Descrivere il codice relativo ai sette processi, inserendo opportunamente **wait()** e **signal()** ed utilizzando il minimo numero di semafori (definirne il valore di inizializzazione).

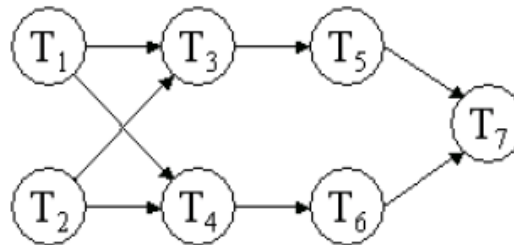


Semaphore $S[6]$;
 Process $T_i (i=1, \dots, 7)$ {
 ... T_1 ; $\text{signal}(S[3])$; $\text{signal}(S[4])$;
 ... T_2 ; $\text{signal}(S[3])$; $\text{signal}(S[4])$;
 $\text{wait}(S[3])$; $\text{wait}(S[3])$; ... T_3 ... ; $\text{signal}(S[5])$;
 $\text{wait}(S[4])$; $\text{wait}(S[4])$; ... T_4 ... ; $\text{signal}(S[6])$;
 $\text{wait}(S[5])$; T_5 ... ; $\text{signal}(S[7])$;
 $\text{wait}(S[6])$; T_6 ... ; $\text{signal}(S[7])$;
 $\text{wait}(S[7])$; $\text{wait}(S[7])$; T_7 ...
 }



Esercizio 1 – Soluzione

❖ Utilizzando 5 semafori si ottiene: *non ottima*



with initialization of

```
semaphore S[3:7] = ([5] 0); // init
T1: expand, init ...signal(S[3]); signal(S[4]);
T2: ...signal(S[3]); signal(S[4]);
denovo prima di T1, T2 {
T3: wait wait(S[3]); wait(S[3])... signal(S[5]);
T4: wait(S[4]); wait(S[4])... signal(S[6]);
T5: wait(S[5]); ... signal(S[7]);
T6: wait(S[6]); ... signal(S[7]);
T7: wait(S[7]); wait(S[7]);...
```

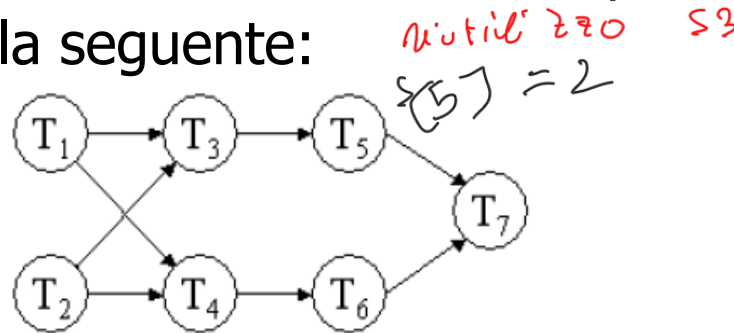
processi eseguiti nell'ordine sequenziale





Esercizio 1 – Soluzione (cont.)

- ❖ Poiché T_7 viene eseguito dopo T_3 e T_4 , un semaforo, per esempio $S[3]$, può essere riutilizzato. Una possibile soluzione con 4 semafori è la seguente:



```
semaphore S[3:6] = ([4] 0); // init
```

```
T1: ...signal(S[3]); signal(S[4]);
```

```
T2: ...signal(S[3]); signal(S[4]);
```

```
T3: wait(S[3]); wait(S[3])... signal(S[5]); signal(s[5]);
```

```
T4: wait(S[4]); wait(S[4])... signal(S[6]) ← come prima
```

```
T5: wait(S[5]); ... signal(S[3]); a + 5 code una sol
```

```
T6: wait(S[6]); ... signal(S[3]);
```

```
T7: wait(S[5]); wait(S[3]); wait(S[3]);...
```

Handwritten notes: $\hookrightarrow$ secondo la nostra regola, diventa parallelo per il livello T2. $\hookrightarrow$ da allora signal T5, abbiamo 7.38 $\hookrightarrow$ riutilizzo



Esercizio 2

❖ Monitor per la gestione di mailbox

- Utilizziamo il costrutto monitor per risolvere il problema della comunicazione tra processi:
 - ▶ il monitor incorpora il buffer dei messaggi (gestito in modo circolare)
 - ▶ i processi produttori (o consumatori) inseriranno (o preleveranno) i messaggi mediante le funzioni **send()** (o **receive()**) definite nel monitor
- La struttura dati che rappresenta il buffer fa parte delle variabili locali al monitor e quindi le operazioni **send()** e **receive()** possono accedere solo in modo mutualmente esclusivo a tale struttura





Esercizio 2 – Soluzione

monitor buffer_circolare

condizioni {
 messaggio buffer[N];
 int contatore=0; int testa=0; int coda=0;
 condition pieno;
 condition vuoto; } *vedi le condizioni*

*accesso in lettura esclusiva al buffer
garantisce del monitor*

void send(messaggio m)

{
 if (contatore==N) pieno.wait;

 buffer[coda] = m; *messaggio messo in coda, l'elemento usato*

 coda = (coda + 1) % N; *incremento*

 ++contatore;

 vuoto.signal; *non è più vuoto, se era vuoto*

}

*so after pieno,
si mette in attesa
che il processo
che riceve segnali*





Esercizio 2 – Soluzione (cont.)

```
messaggio receive()  
{  
    messaggio m;  
    if (contatore == 0) vuoto.wait;  
    m = buffer[testa]; lepo delle porta del buffer  
    testa = (testa + 1) % N;  
    --contatore;  
    pieno.signal; se era pieno, ora no e' più  
    return m; sblocco risorse in attesa  
}  
} /* fine monitor */
```





Esercizio 3

❖ Gestione parcheggio

- Il gestore di un parcheggio vuole automatizzare il sistema di ingresso/uscita in modo da impedire l'accesso quando il parcheggio è pieno (cioè la sbarra d'ingresso deve sollevarsi solo se ci sono posti disponibili) e da consentire l'uscita solo in caso di parcheggio non vuoto (cioè la sbarra d'uscita deve rimanere abbassata in caso di parcheggio vuoto)
- Scrivere un monitor che permetta di gestire l'accesso al parcheggio





Esercizio 3 – Soluzione

```
monitor CarPark
  condition full, empty;
  integer spaces, capacity;

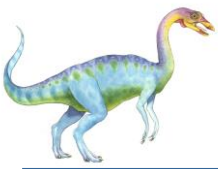
  procedure enter();
  begin
    while(spaces=0) do full.wait; impedisce accessi se pieno
    spaces := spaces-1;
    empty.signal;
  end;

  procedure exit();
  begin
    while(spaces=capacity) do empty.wait; e' vuoto, non si può uscire.
    spaces := spaces+1;
    full.signal; si è liberato un posto
  end;

  spaces := N;
  capacity := N;
end monitor;
```

desagregato da altri qualcos





Esercizio 4

❖ Variabili condivise:

```
semaphore S=1, M=1;  
int x=10;
```

```
{  
  R1  wait(M);  
      x=x+1;  
      signal(M);  
  
  R2  wait(M);  
      write(x);  
      signal(M);  
  
      signal(S);  
}
```

Processo P_1

```
{  
      wait(S);  
  
  R3  wait(M);  
      x=x-1;  
      signal(M);  
  
  R4  wait(M);  
      write(x);  
      signal(M);  
}
```

Processo P_2

*x no race condition,
4 sezioni ~~for~~ critici,
X ~~accade~~ in MUTUA ESCLUSIONE*





Esercizio 4 (cont.)

- ❖ Individuare le regioni critiche nel codice di P_1 e P_2 ; si possono verificare race condition per la variabile condivisa x ?

Le regioni critiche (in R1, R2, R3 e R4) sono protette da semafori per cui non vi sono race condition su x

- ❖ Determinare tutti i possibili output del programma concorrente (P_1 in parallelo con P_2).

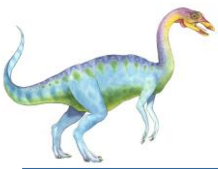
R_1, R_2, R_3, R_4 ^{11, 10} Il semaforo s non influisce sul flusso di esecuzione e quindi l'output dipende dall'ordine di esecuzione di R1–R2 e R3–R4:

- R_1, R_3, R_4, R_2 (in qualsiasi ordine) → stampa: 10 10
- R_1, R_2, R_3, R_4 → stampa: 11 10
- R_3, R_4, R_1, R_2 → stampa: 9 10

- ❖ Supponiamo di inizializzare il semaforo s a 0 invece che a 1; determinare tutti i possibili output del programma (P_1 in parallelo con P_2) così modificato.

Il semaforo s forza l'ordine di esecuzione $P_1 \rightarrow P_2$ e quindi ci sarà un solo output: 11 10





Esercizio 5

❖ Variabili condivise:

```
semaphore S=2, T=0;  
int x=0;
```

```
{  
    wait(T);  
    x=5;  
    signal(S);  
    x=6;  
}
```

Processo P_1

inizializza lui

```
{  
    wait(S);  
    x=1;  
    signal(T);  
    wait(S);  
    x=10;  
    write(x);  
}
```

Processo P_2

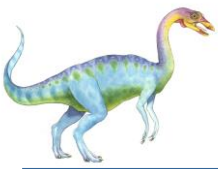




Esercizio 5 (cont.)

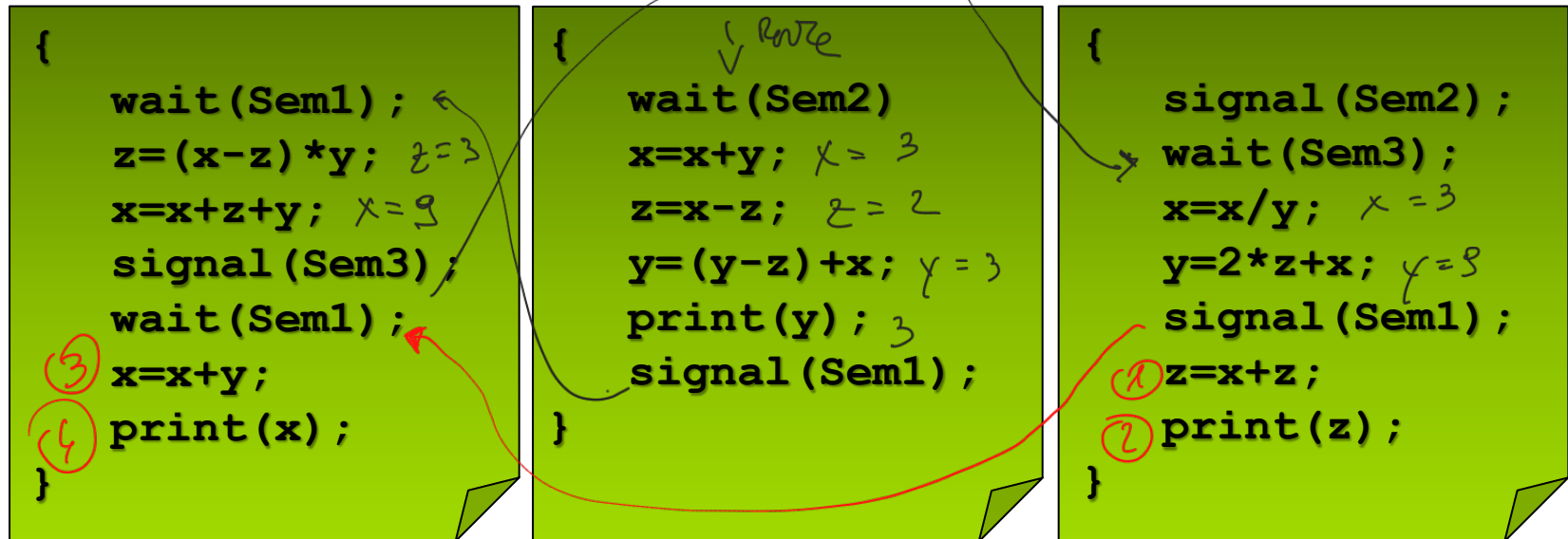
- ❖ Si supponga che i processi vengano eseguiti concorrentemente sulla stessa CPU:
 - Determinarne tutti i possibili output.
Con $S=2$ e $T=0$, l'insieme di possibili output è $\{5,6,10\}$
 - Cosa succede se inizialmente i semafori hanno valore $S=1$ e $T=0$?
Con $S=1$ e $T=0$, l'insieme di possibili output è $\{6,10\}$
 - Nel caso $S=1$ e $T=0$, inserire una sola nuova chiamata `wait()` su S o T in uno dei processi P_1 e P_2 in modo da ottenere come unico possibile output il valore 10.
Ponendo `wait(T)` prima di $x=6$ in P_1 si forza l'output a 10





Esercizio

- ❖ Si considerino i processi A, B e C che si sincronizzano attraverso i semafori **Sem1**, **Sem2** e **Sem3** (inizializzati a 0) e che operano sulle variabili condivise **x**, **y** e **z**, inizializzate rispettivamente a 1, 2 e 1.



- ❖ Con quale ordine i processi stampano i valori delle tre variabili? Quali valori vengono stampati?
- ❖ Motivare la risposta data.



$$\text{Sen } 1, 2, 3 = 0$$

$$x = 1 \quad y = 2 \quad z = 1$$

• Parte 2

$$\bullet x = 3, z = 2, y = 3$$

output

$$y = 3$$

$$\bullet x = 3, y = 9, z = 3$$

• Possibili valori 'stop' 2

$$1, 2, 3, 4, \text{ step } 3, 6, 12$$

$$1, 3, 2, 4, (z = 6), (x = 12), 3, 6, 12$$

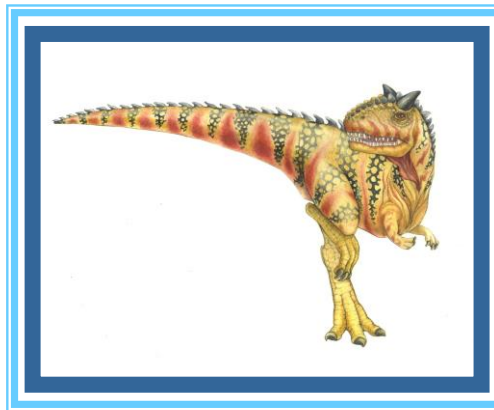
$$1, 3, 4, 2, 3, 12, 6$$

$$3, 4, 1, 2 \Rightarrow 3, 12, 15$$

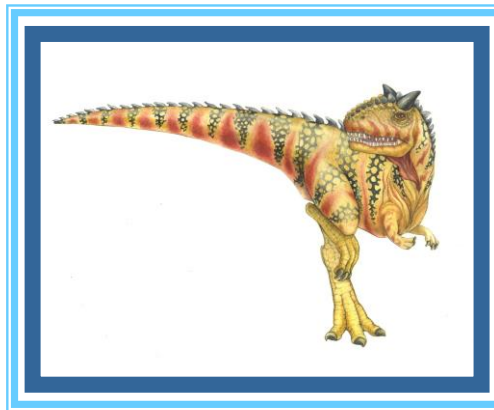
$$3, 1, 4, 2 \quad \text{u u}$$

$$3, 1, 2, 4 \quad 3, 15, 12$$

Fine del Capitolo 7



Deadlock





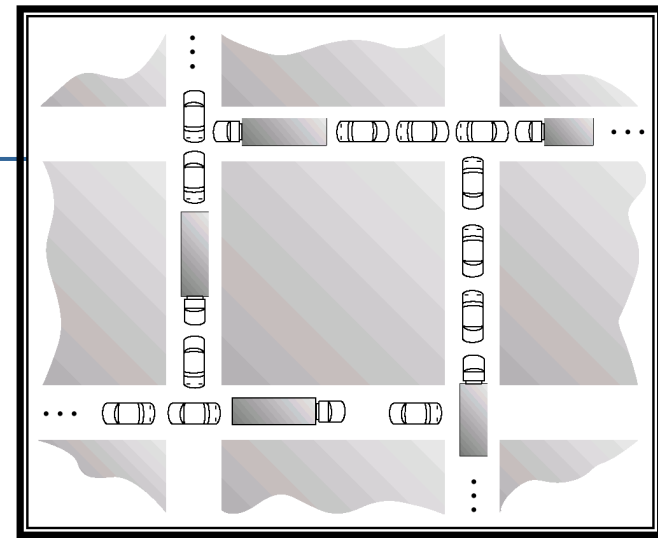
Obiettivi

- ❖ Illustrare come e quando può verificarsi il deadlock
- ❖ Definire le quattro condizioni necessarie che lo caratterizzano
- ❖ Identificare il deadlock in un grafo di allocazione delle risorse
- ❖ Valutare diversi approcci per prevenire il deadlock
- ❖ Applicare l'algoritmo del Banchiere per evitare il deadlock 
- ❖ Applicare l'algoritmo di rilevamento dei deadlock
- ❖ Valutare gli approcci per il recupero da situazioni di deadlock *multicone, alta possibilità di deadlock*

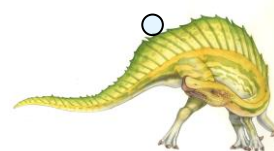




Sommario



- ❖ Il problema del deadlock *in ogni processo c'è traffico*
- ❖ Modello del sistema *sblocco facendo marciare i processi...*
- ❖ Deadlock nelle applicazioni multithread *prelazioniamo una risorsa*
- ❖ Caratterizzazione dei deadlock
- ❖ Metodi per la gestione dei deadlock
 - Prevenire i deadlock
 - Evitare i deadlock
 - Rilevare i deadlock
 - Ripristino da situazioni di deadlock





Il problema del deadlock – 1

insieme di processi che detengono risorse ma devono aspettare altre risorse per finire, devono aspettare altre...

- ❖ In un ambiente multiprogrammato più thread possono entrare in competizione per ottenere un numero finito di risorse
- ❖ **Deadlock** o **stallo** — insieme di thread bloccati, ciascuno dei quali “possiede” (almeno) una risorsa ed attende di acquisire una o più risorse allocate ad altri thread dell'insieme (che detengono risorse, ma ne attendono altre)
- ❖ Nella programmazione multithread, gli stalli si verificano più frequentemente, poiché più thread possono competere per un insieme (limitato) di risorse condivise

*Costa tanto risolvere il deadlock ... $O(n^4)$
Solo un check costante tantissimo
 $n = \# \text{ thread in esecuzione}$*





Il problema del deadlock – 2

- ❖ Difficoltà intrinseca nell'identificare deadlock che, generalmente, si verificano solo in particolari condizioni di scheduling
- ❖ Anche se esistono applicazioni che possono rilevare i programmi suscettibili di stallo, la maggior parte dei SO attuali non offre strumenti di prevenzione
- ❖ Progettare programmi che non provochino deadlock è responsabilità dei programmatori *meglio prevenire*
 - Compito sempre più difficile nei sistemi multicores in cui, più in generale, tutti i problemi di liveness, si presentano con maggior frequenza e complessità





Esempi di deadlock

- ❖ *Ciascun processo in un insieme di processi attende un evento che può essere causato solo da un altro processo dell'insieme*

evento: rilascio di una risorsa

- ❖ **Esempio 1**

- Nel sistema sono presenti due stampanti *Deadlock*
- T_1 e T_2 si sono assicurati una stampante ciascuno ed entrambi richiedono anche l'altra *Se T_1, T_2 accedono*

- ❖ **Esempio 2**

due 2 processi, deadlock

- Semafori A e B, ^{$A, B = 1$} inizializzati a 1:

T_1

T_2

devo acquisire il semaforo, risorse

1) **wait (A) ;**

2) **wait (B) ;**

3) **wait (B) ;**

4) **wait (A) ;**

Se lo scheduler interlopla...

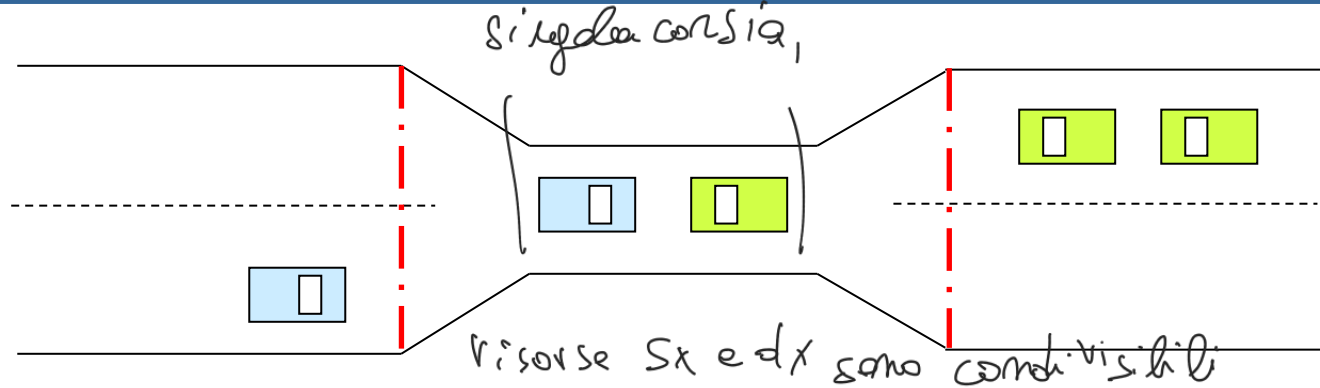
nessuno dei due

è in grado di andare avanti e fare una signal





Esempio: attraversamento di un ponte



- ❖ In ogni istante, sul ponte possono transitare autoveicoli solo in una direzione
- ❖ Ciascuna sezione del ponte può essere immaginata come una risorsa
- ❖ Se si verifica un deadlock (come in figura), il *recovery* può essere effettuato se un'auto torna indietro (rilascia la risorsa ed esegue un roll back)
- ❖ In generale, in caso di deadlock, può essere necessario che più auto debbano tornare indietro
- ❖ È possibile si verifichi ~~starvation (attesa indefinita)~~

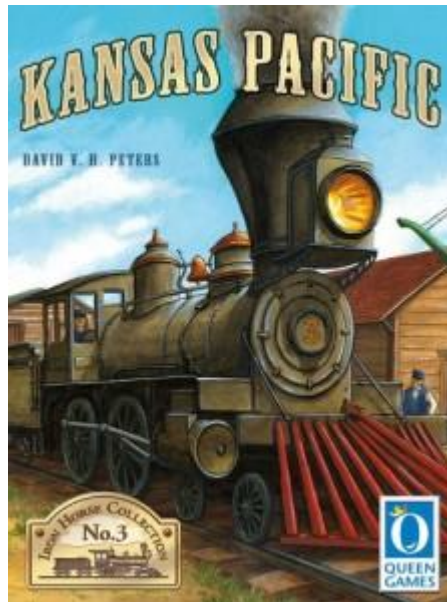




Esempio: leggi ferroviarie in Kansas

- ❖ Testo parziale della proposta di legge per la regolamentazione delle ferrovie del Kansas (Aprile 1913):

"Quando due treni convergono ad un incrocio, ambedue devono arrestarsi, e nessuno dei due può ripartire prima che l'altro si sia allontanato"



When two trains approach each other at a crossing, both shall come to a full stop and neither shall start up again until the other is gone





Modello di sistema – 1

- ❖ Tipi di risorse $R_1, R_2, \dots, R_m$
Cicli di CPU, spazio di memoria, file, dispositivi di I/O, lock, semafori
nel caso semaforo o lock, è una classe a parte
- ❖ Le risorse di una classe vengono dette istanze della classe
- ❖ Ciascun tipo di risorsa R_i ha W_i istanze
- ❖ Il numero di risorse in una classe viene detto molteplicità del tipo di risorsa *se risorse ed istanza singola, facile vedere deadlock*
- ❖ Un processo non può richiedere una specifica risorsa, ma solo una risorsa di una specifica classe *Più è una risorsa...*
- ❖ Una richiesta per una classe di risorse può essere soddisfatta da qualsiasi istanza di quel tipo
- ❖ Risorse ad *assegnazione statica* *risorse eseguite non vanno in stile*
 - Avviene al momento della creazione del processo e rimane valida fino alla terminazione
 - **Esempio:** descrittore del processo *se assegnazione dinamica potrebbe essere*





Modello di sistema – 2

- ❖ Il protocollo di accesso dei thread alle risorse ad *assegnazione dinamica* prevede...
 - **Richiesta** – se la richiesta non può essere soddisfatta immediatamente, causa diverso utilizzo della risorsa, il thread richiedente deve attendere fino al rilascio della risorsa da parte del thread che la detiene
 - **Utilizzo**
 - **Rilascio**
- ❖ Richiesta/rilascio realizzati con apposite system call, quali **request/release** device, **open/close** file, **allocate/free** memory, **wait/signal** semaphore, **acquire/release** lock mutex





Modello di sistema – 3

- ❖ Ogni volta che si usa una risorsa gestita dal kernel, il SO controlla che il thread utente ne abbia fatto richiesta e che questa gli sia stata accordata
- ❖ Una tabella di sistema registra lo stato di ogni risorsa e, se è allocata, indica il thread relativo
- ❖ Ogni risorsa può avere una coda d'attesa





Deadlock nelle applicazioni multithread

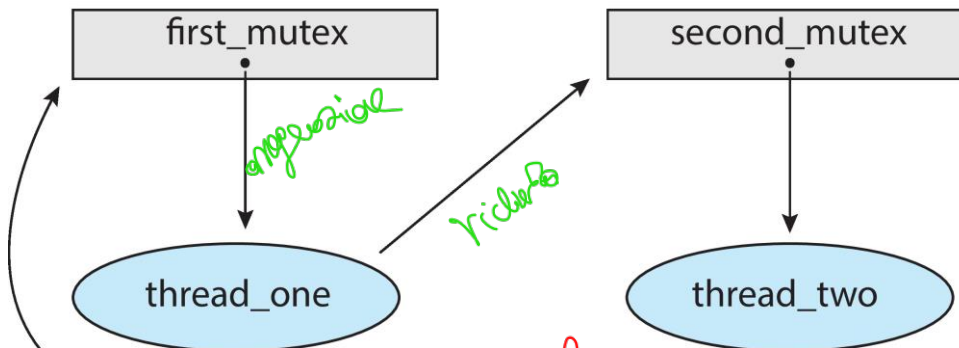
❖ Si creano e si inizializzano due lock mutex

```
pthread_mutex_t first_mutex;  
pthread_mutex_t second_mutex;
```

```
pthread_mutex_init(&first_mutex, NULL);  
pthread_mutex_init(&second_mutex, NULL);
```

Stallo se **thread_one** acquisisce **first_mutex** e **thread_two** acquisisce **second_mutex**

grafo allocazione delle risorse



*e' un grafico
ciclico, per risorse per e' a singola
istanza*

*se 1° lock acquisito
1° mutex e 2° lock
acquisisce 2°...
deadlock*

```
/* thread_one runs in this function */  
void *do_work_one(void *param)
```

```
pthread_mutex_lock(&first_mutex);  
pthread_mutex_lock(&second_mutex);  
/**  
 * Do some work  
 */  
pthread_mutex_unlock(&second_mutex);  
pthread_mutex_unlock(&first_mutex);  
  
pthread_exit(0);
```

```
/* thread_two runs in this function */  
void *do_work_two(void *param)  
{
```

```
pthread_mutex_lock(&second_mutex);  
pthread_mutex_lock(&first_mutex);  
/**  
 * Do some work  
 */  
pthread_mutex_unlock(&first_mutex);  
pthread_mutex_unlock(&second_mutex);
```

```
pthread_exit(0);
```

8.12 SIANO SUCUNI che e' DEADLOCK



Stallo attivo – 1

tipo contenzione STUOLLO, a dx o a sx entell. (due persone)

- ❖ Il **livelock** o **stallo attivo** è una situazione simile al deadlock in cui, tuttavia, i thread non sono effettivamente bloccati, ma di fatto non progrediscono
- ❖ Gli stati dei thread coinvolti nel livelock cambiano costantemente l'uno rispetto all'altro
- ❖ Pertanto, il livelock può considerarsi una forma speciale di starvation
- ❖ Come il deadlock, il livelock può verificarsi solo in determinate condizioni di scheduling

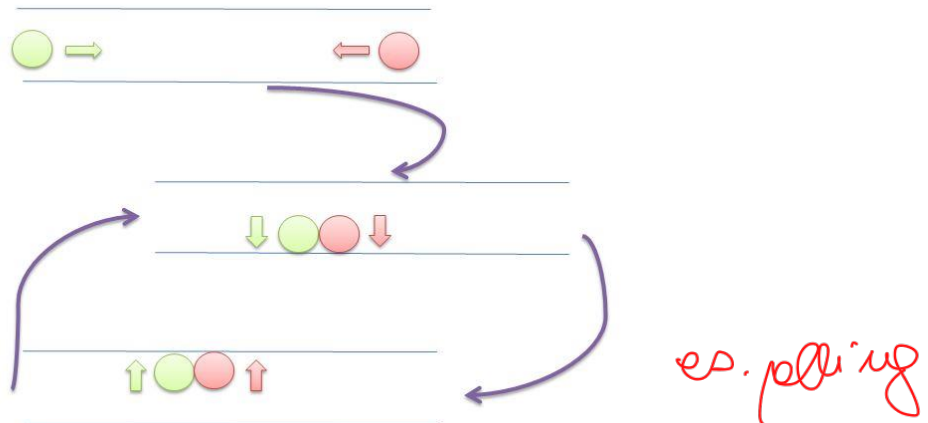




Stallo attivo – 2

❖ Esempi

- Due persone si incontrano in un corridoio e, cercando di passare, si spostano ripetutamente da un lato all'altro del corridoio stesso



- Due thread effettuano polling (busy waiting) per verificare l'uno lo stato dell'altro e non fanno progressi (livelock mutuo), ma non sono in deadlock dato che comunque effettuano il polling
- Collisioni su rete Ethernet *es. canale half duplex, collisione*
 - ▶ Soluzione: ritrasmettere un pacchetto dopo un intervallo di tempo casuale, *spesso Tx dei trasmettitori in un mezzo*



Stallo attivo – 3

```
/* thread_one runs in this function */
void *do_work_one(void *param)
{
    int done = 0;

    while (!done) {
        pthread_mutex_lock(&first_mutex);
        if (pthread_mutex_trylock(&second_mutex)) {
            /**
             * Do some work
             */
            pthread_mutex_unlock(&second_mutex);
            pthread_mutex_unlock(&first_mutex);
            done = 1;
        }
        else
            pthread_mutex_unlock(&first_mutex);
    }

    pthread_exit(0);
}
```

```
/* thread_two runs in this function */
void *do_work_two(void *param)
{
    int done = 0;

    while (!done) {
        pthread_mutex_lock(&second_mutex);
        if (pthread_mutex_trylock(&first_mutex)) {
            /**
             * Do some work
             */
            pthread_mutex_unlock(&first_mutex);
            pthread_mutex_unlock(&second_mutex);
            done = 1;
        }
        else
            pthread_mutex_unlock(&second_mutex);
    }

    pthread_exit(0);
}
```

pthread_mutex_trylock() tenta di acquisire un mutex senza bloccare il thread



Caratterizzazione del deadlock

- ❖ Una situazione di deadlock può verificarsi solo se valgono simultaneamente le seguenti condizioni
- **Mutua esclusione:** nel sistema, esiste almeno una risorsa non condivisibile, cioè utilizzabile da un solo thread alla volta
 - **Possesso ed attesa:** un thread, che possiede almeno una risorsa, attende di acquisire ulteriori risorse possedute da altri thread
 - **Impossibilità di prelazione:** una risorsa può essere rilasciata dal thread che la detiene solo volontariamente, al termine del suo utilizzo
 - **Attesa circolare:** esiste un insieme $\{T_0, T_1, \dots, T_n\}$ di thread in attesa, tali che T_0 è in attesa di una risorsa che è posseduta da T_1 , T_1 è in attesa di una risorsa posseduta da $T_2, \dots, T_{n-1}$ è in attesa di una risorsa posseduta da T_n e T_n è in attesa di una risorsa posseduta da T_0

es. La CPU, gli accessi al rotta esclusione *questo valore è 1 e 4, non può essere*

*con
ness
di poterli*

*Però a
almeno uno
e si possono
di attesa
circular*





Grafo di assegnazione delle risorse – 1

Struttura dati complesso, coppia (V, E) vertici e archi

- ❖ Un grafo è costituito da un insieme di vertici (o nodi) V variamente connessi per mezzo di un insieme di archi E

Grafo completo, ogni nodo è connesso a tutti gli altri, non è questo il caso

- ❖ Nel **grafo di assegnazione delle risorse...**

- L'insieme V è partizionato in due sottoinsiemi:

- ▶ $T = \{T_1, T_2, \dots, T_n\}$ è l'insieme costituito da tutti i thread attivi nel sistema

- ▶ $R = \{R_1, R_2, \dots, R_m\}$ è l'insieme di tutti i tipi di risorse presenti nel sistema

- ▶ **Arco di richiesta:** $T_i \rightarrow R_j$

- ▶ **Arco di assegnazione:** $R_j \rightarrow T_i$

Nodi diversi e archi che si scambiano risorse diverse (non omogenee)

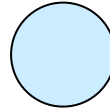
D. Semplicità, da un lato ho $T_1 - T_n$ thread, e dall'altro le classi delle risorse





Grafo di assegnazione delle risorse – 2

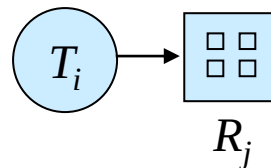
❖ Thread



❖ Tipo di risorsa con 4 istanze

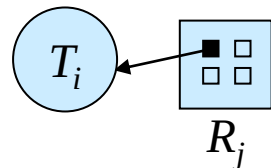


❖ T_i richiede un'istanza di R_j *del tipo R_j un arco di richiesta*



Arco di richiesta

❖ T_i possiede un'istanza di R_j



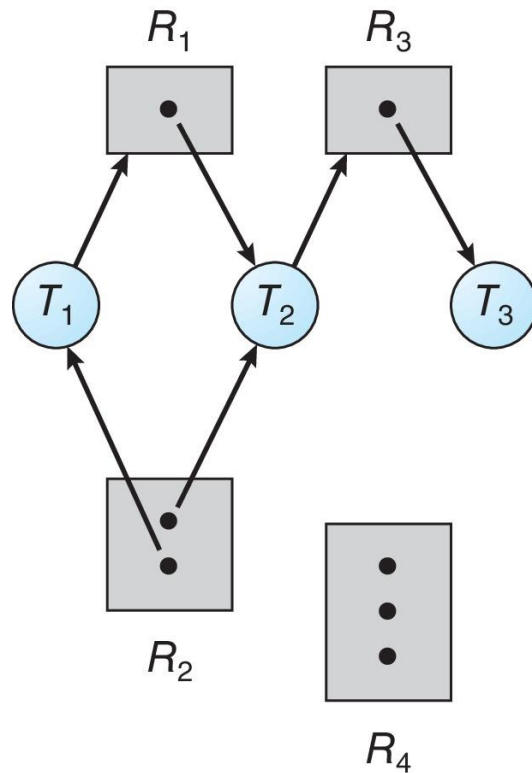
Arco di assegnazione



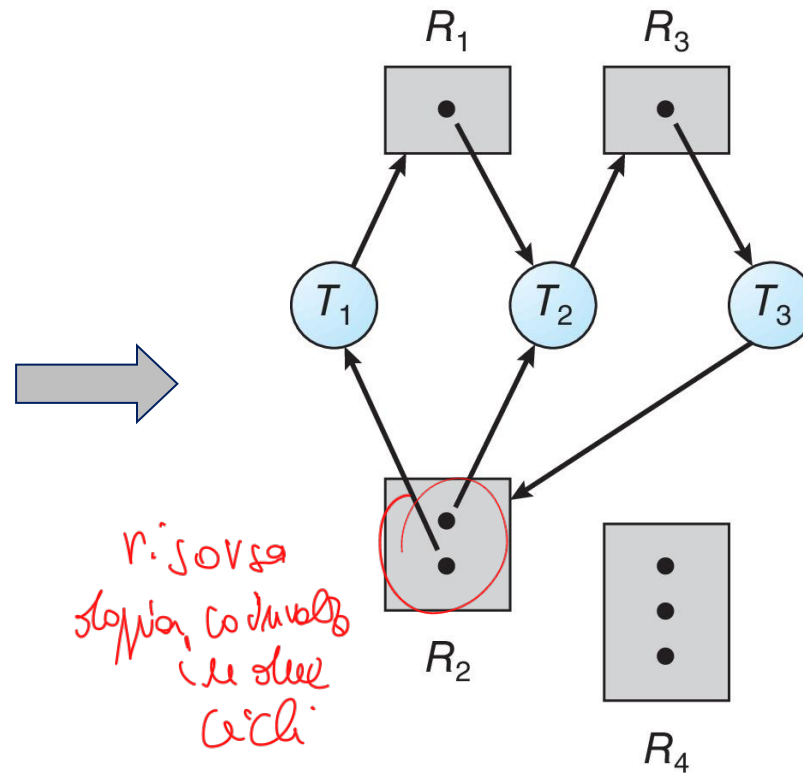


Esempi di grafo di assegnazione risorse – 1

Grafo di assegnazione risorse



Grafo di assegnazione risorse con deadlock



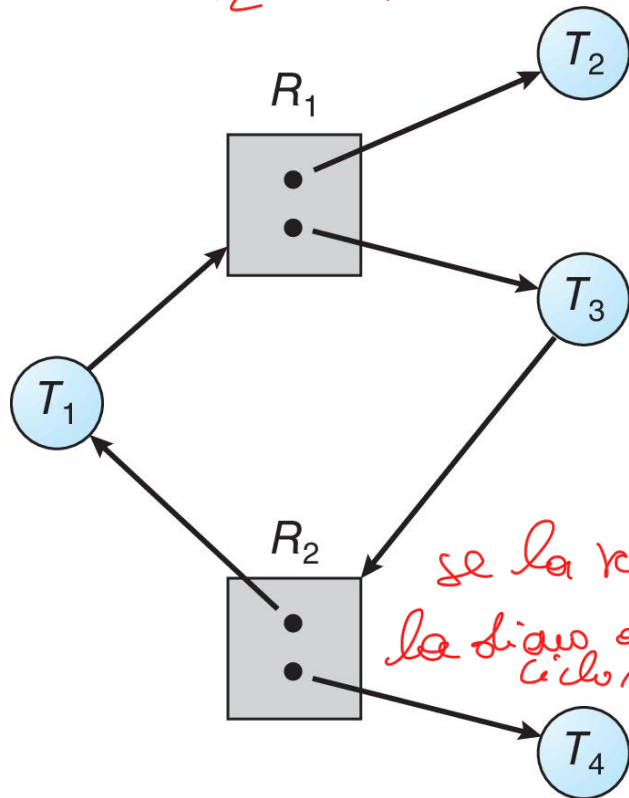
Deadlock articolato con grafo a incalce a risorse singole,
ma se ci sono più istanze e' condizione necessaria ma non suff.
devo vedere caso per caso. Se non e' a incalce, no deadlock





Esempi di grafo di assegnazione risorse – 2

T₂ ha richieste due, T₂ Termina



❖ Se il grafo non contiene cicli

⇒ Non ci sono deadlock

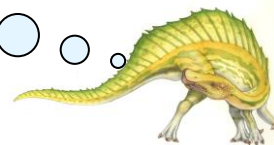
❖ Se il grafo contiene un ciclo

⇒ Se vi è una sola istanza per ogni tipo di risorsa, allora si ha un deadlock

se la richiedo ⇒ Se si hanno più istanze per tipo di risorsa, allora il deadlock è possibile (ma non certo)
la danno a T3, il ciclo non c'è

Grafo di assegnazione risorse con un ciclo, ma privo di deadlock

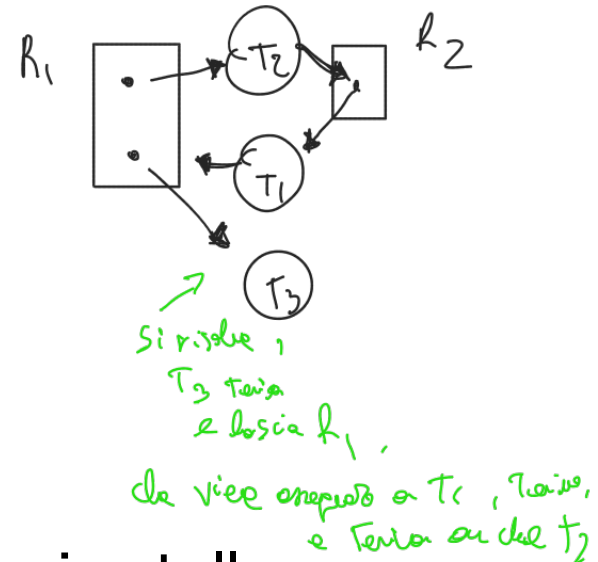
Il ciclo c'è, ma il deadlock no...





Se un processo richiede una risorsa, se la risorsa disponibile
Esempio 1 arco di richiesta di Brachiosaurus
in arco di assegnazione

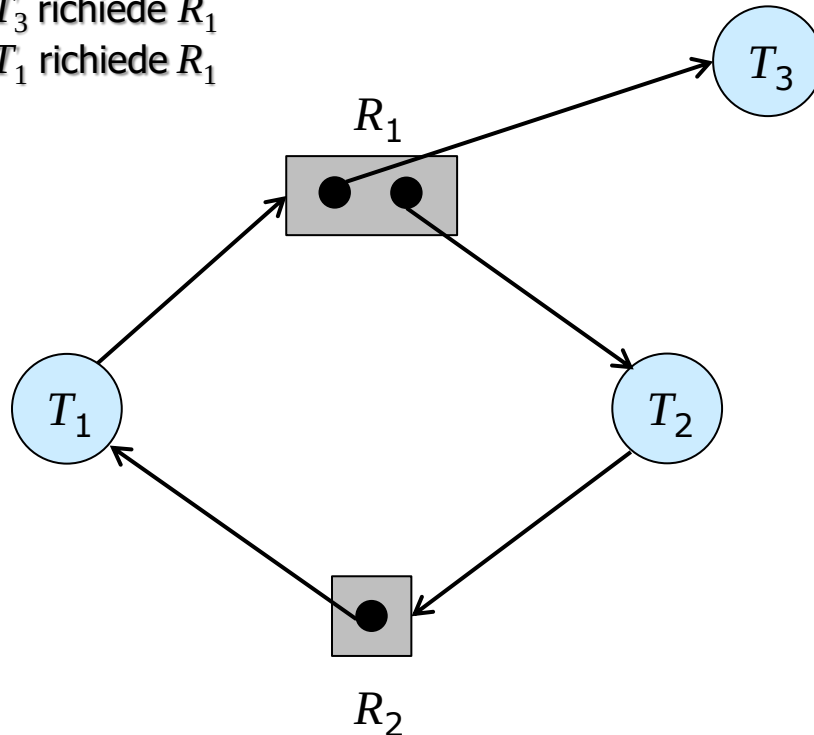
- ❖ Si consideri un sistema con tre thread (T_1, T_2, T_3), e due tipologie di risorse (R_1, R_2), con la disponibilità di due risorse di tipo R_1 e una risorsa di tipo R_2
- ❖ Si consideri la seguente cronologia di richieste:
 - T_2 richiede R_1
 - T_1 richiede R_2
 - T_2 richiede R_2
 - T_3 richiede R_1
 - T_1 richiede R_1
- ❖ Verificare se il sistema si trova in stallo





Esempio 1 (cont.)

T_2 richiede R_1
 T_1 richiede R_2
 T_2 richiede R_2
 T_3 richiede R_1
 T_1 richiede R_1



- ❖ Si forma un ciclo ma il sistema NON è in stallo
- ❖ Infatti un elemento del ciclo (R_1) ha molteplicità due e una delle due istanze è "fuori dal ciclo": T_3 può terminare e rilasciare la sua istanza di R_1 , che viene così assegnata a T_1

*c'è ciclo, ma non c'è deadlock,
 T_3 rilascia*





Esempio 2

- ❖ Si consideri un sistema con 4 thread (T_1, T_2, T_3, T_4) e 4 tipologie di risorse (R_1, R_2, R_3, R_4) con disponibilità:
 - 1 risorsa di tipo R_1
 - 1 risorsa di tipo R_2
 - 1 risorsa di tipo R_3
 - 2 risorse di tipo R_4
- ❖ Si assuma che:
 - ogni volta che un thread richiede una risorsa libera, questa venga assegnata al thread richiedente (l'arco di richiesta si trasforma *istantaneamente* in arco di assegnazione)
 - ogni volta che un thread richiede una risorsa già occupata, il richiedente deve attendere che la risorsa si liberi prima di impossessarsene (coda FIFO di attesa)



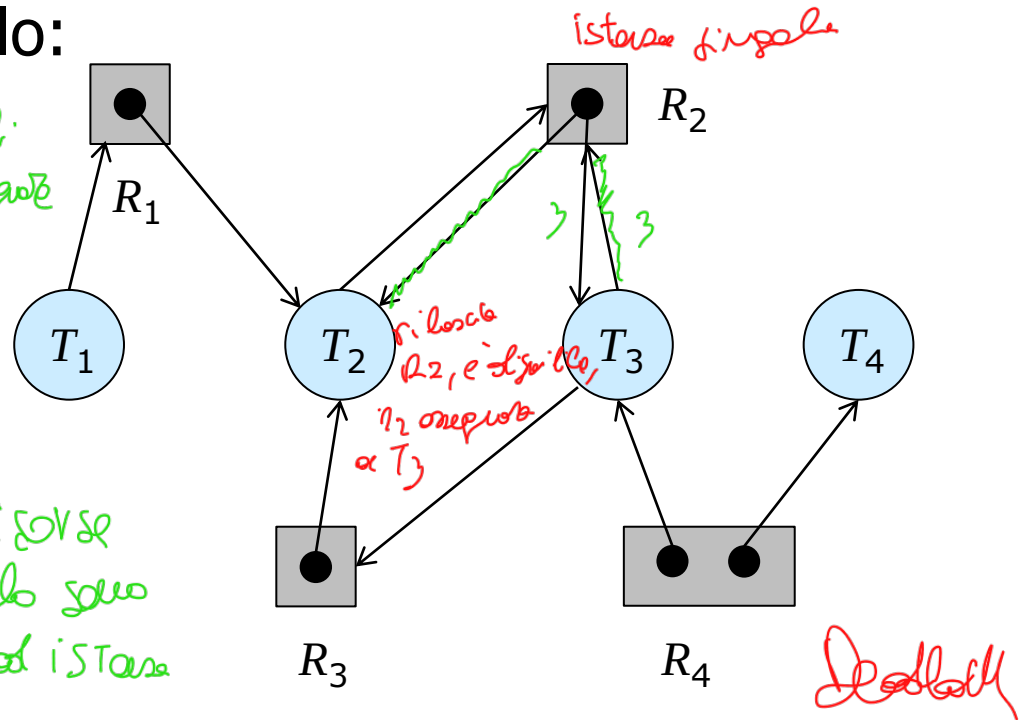


Esempio 2 (cont.)

❖ Si consideri la seguente successione cronologica di richieste e rilasci di risorse e si verifichi se, alla fine, il sistema si trova in stallo:

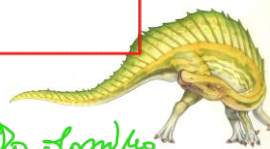
- T_2 richiede R_1, R_2, R_3 *gli
hanno bisogno*
- T_3 richiede R_2, R_4
- T_2 rilascia R_2
- T_4 richiede R_4
- T_1 richiede R_1
- T_2 richiede R_2
- T_3 richiede R_3

*Le risorse
nel ciclo sono
tutte al istanza
singola*



Ciclo di richieste/assegnazioni che coinvolge
 $T_2, R_2, T_3, R_3 \Rightarrow$ **il sistema è in stallo**

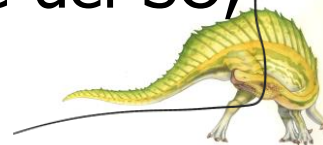
Se allora processi richiedano R2, R3, in quel caso anche loro, offerta singola





Metodi di gestione del deadlock

- ❖ Assicurare che il sistema non entri mai in uno stato di deadlock *far sì che una condizione non si verifichi, per esempio a mutua esclus.*
 - **Prevenire i deadlock:** evitare che si verifichino contemporaneamente mutua esclusione, possesso e attesa, impossibilità di prelazione e attesa circolare *scorso uso delle risorse*
 - ⇒ Basso utilizzo delle risorse e throughput ridotto
 - **Evitare i deadlock:** evitare gli stati del sistema a partire dai quali si può evolvere verso il deadlock *mantenere accesso sicuro...*
 - ⇒ Informazioni aggiuntive sulle richieste dei thread
- ❖ Permettere al sistema di entrare in uno stato di deadlock, quindi **ripristinare** il sistema *più comodo di uccidere il processo*
 - Determinare la presenza di un deadlock
 - Ripristinare la corretta operatività del sistema
- ❖ Ignorare il problema e fingere che i deadlock non si verifichino mai; impiegato dalla maggior parte dei SO, inclusi Windows e UNIX/Linux





Prevenire i deadlock – 1

❖ Limitare le modalità di richiesta/accesso delle/alle risorse per invalidare una delle quattro condizioni necessarie

- **Mutua esclusione** — non è richiesta per risorse condivisibili $\Rightarrow$ non possono essere coinvolte in uno stallo; deve valere invece per risorse che non possono essere condivise, quindi non può essere prevenuta

non è indipendente

- **Possesso e attesa** — occorre garantire che, quando un thread richiede una risorsa, non ne possieda altre *PRESENT*

Thread entra

nel sistema, trova

richiedere TUTTE le risorse disponibili.

Tutte risorse occupate anche

se non le uso...

► Esigere dal thread di stabilire ed allocare tutte le risorse necessarie prima che inizi l'esecuzione, o consentire la richiesta di risorse solo quando il thread non ne possiede alcuna

► Basso impiego delle risorse; possibile l'attesa indefinita per thread con forti richieste di risorse

*• Rilascio tutte le risorse attesi, poi ripreso tutte le vecchie e le nuove
meglio quelle rilasciate, vengono occupate da altri processi. quando voglio riprenderle*





Prevenire i deadlock – 2

Impossibilità di prelazione

rilascio risorse se richiedo
da lui, ma potrei andare in
starvation

Se un thread le
risorse possono
essere prelevate...
non posso lasciare
stare a metà
per far usare
stare a metà
altro process

- ▶ Se un thread, che possiede alcune risorse, richiede un'altra risorsa, che non gli può essere allocata immediatamente, allora rilascia tutte le risorse possedute
- ▶ Le risorse rilasciate (prelazionate al thread) vengono aggiunte alla lista delle risorse che il thread sta attendendo
- ▶ Il thread viene avviato nuovamente solo quando può ottenere sia le risorse precedentemente possedute sia quelle attualmente richieste
- ▶ Alternativamente, le risorse possono essere assegnate al thread che le richiede, qualora siano attualmente possedute da un altro thread in attesa
- ▶ Protocollo adatto per risorse il cui stato si può salvare e recuperare facilmente (registri, memoria), non per dispositivi di I/O o primitive di sincronizzazione





Prevenire i deadlock – 3

per le risorse ... definito l'ordinamento a cosa posso accedere

- **Attesa circolare** — si impone un ordinamento totale (possibilmente "logico") su tutti i tipi di risorsa, $f: R \rightarrow N$, e si pretende che ciascun thread richieda le risorse in ordine crescente *ovvero le risorse nel sistema, ordinamento non serve un protocollo*
- In alternativa, ogni thread, prima di richiedere una istanza di R_j , deve rilasciare tutte le istanze di R_i , tali che $f(R_i) \geq f(R_j)$
 - ▶ Supponiamo che, avendo imposto l'ordinamento suddetto, si verifichi comunque l'attesa circolare
 - ▶ Siano $\{T_0, T_1, \dots, T_n\}$ i thread coinvolti nell'attesa circolare, dove T_i attende la risorsa R_{i+1} posseduta dal thread T_{i+1} (in aritmetica modulare $\Rightarrow T_n$ attende R_0)
 - ▶ Poiché il thread T_i possiede la risorsa R_i mentre richiede la risorsa R_{i+1} , $\forall i$, vale la condizione *se acquisisco le risorse in un certo ordine, ho finito che non c'è lo STALCO*

$$f(R_0) < f(R_1) < \dots < f(R_n) < f(R_0)$$

assurdo



*La F delle
che indica
quella classe di
risorse*



Esempio di attesa circolare

```
/* thread_one esegue in questa funzione */
void *do_work_one(void *param)
{
    pthread_mutex_lock(&first_mutex);
    pthread_mutex_lock(&second_mutex);
    /* Fa qualcosa... */
    pthread_mutex_unlock(&second_mutex);
    pthread_mutex_unlock(&first_mutex);
    pthread_exit(0);
}

/* thread_two esegue in questa funzione */
void *do_work_two(void *param)
{
    pthread_mutex_lock(&second_mutex);
    pthread_mutex_lock(&first_mutex);
    /* Fa qualcosa... */
    pthread_mutex_unlock(&first_mutex);
    pthread_mutex_unlock(&second_mutex);
    pthread_exit(0);
}
```

Si risolve imponendo, per es.,
 $f(\text{first_mutex})=1$
 $f(\text{second_mutex})=3$
e forzando l'invocazione dei
lock mutex in ordine crescente

lockdep di Linux è uno strumento per rilevare possibili stalli nell'acquisizione di lock nel kernel; verifica che i lock vengano effettivamente richiesti secondo l'ordine stabilito dal loro identificativo





Evitare i deadlock

- ❖ Prevenire i deadlock può causare scarso utilizzo dei dispositivi e ridotta produttività del sistema
- ❖ Viceversa, evitare i deadlock presuppone che il sistema conosca a priori informazioni aggiuntive sulle richieste future dei thread
 - Il modello più semplice e utile richiede che ciascun thread dichiari il numero massimo di risorse di ciascun tipo di cui potrà usufruire nel corso della propria esecuzione
 - L'algoritmo di **deadlock-avoidance** esamina dinamicamente lo stato di allocazione delle risorse per assicurarsi che non si possa verificare una condizione di attesa circolare
 - Lo stato di allocazione delle risorse è definito dal numero di risorse disponibili ed allocate, e dal massimo numero di richieste dei thread

mantenere
il sistema
in stato
sicuro,
l'allocazione
di risorse
sicura,
che si possono
terminare il più presto possibile
l'allocazione delle risorse al 2° ecc

• Thread richiede risorsa, può asseverare
senza che... se no, non la assegna -
che scarso risorse





Stato sicuro – 1

- ❖ Quando un thread richiede una risorsa disponibile, il sistema deve decidere se l'allocazione immediata lasci il sistema in **stato sicuro**
- ❖ Il sistema si trova in uno stato sicuro se e solo se esiste una sequenza sicura di esecuzione di tutti i thread
- ❖ La sequenza $\langle T_1, T_2, \dots, T_n \rangle$ è sicura se, per ogni T_i , le risorse che T_i può ancora richiedere possono essergli allocate sfruttando le risorse disponibili, più le risorse possedute da tutti i T_j , con $j < i$





Stato sicuro – 2

❖ Cioè...

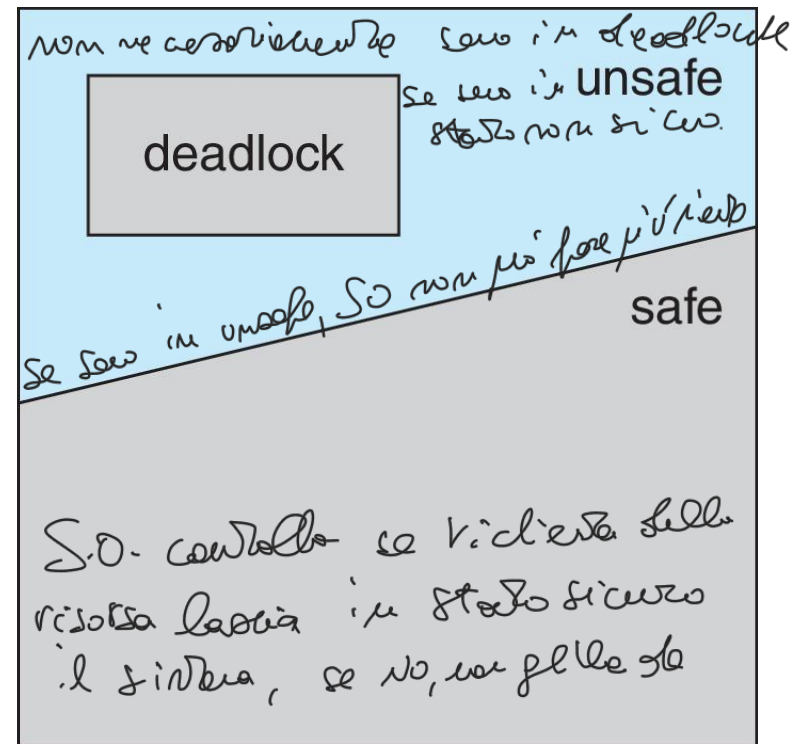
- Se le richieste di T_i non possono essere soddisfatte immediatamente, allora T_i può attendere finché tutti i T_j ($j < i$) abbiano terminato
- Quando i T_j hanno terminato, T_i può ottenere le risorse richieste, eseguire i suoi compiti, restituire le risorse allocate e terminare
- Quando T_i termina, T_{i+1} può ottenere le risorse richieste, etc.





Stato sicuro – 3

- ❖ Se un sistema è in stato sicuro
 - ⇒ non si evolve verso il deadlock
- ❖ Se un sistema è in stato non sicuro
 - ⇒ si può evolvere in deadlock
- ❖ In stato non sicuro, il SO non può impedire ai thread di richiedere risorse in modo da causare un deadlock
- ❖ **Deadlock-avoidance**
 - garantisce che un sistema non evolva mai verso uno stato non sicuro
 - se un thread richiede una risorsa disponibile, può dovere attendere
 - ⇒ scarso utilizzo delle risorse del sistema





Stato sicuro – 4

❖ Esempio

- 12 istanze di una risorsa complessivamente disponibili
- Al tempo t_0

Thread	Richiesta max	Unità possedute
T_0	10	5
T_1	4	2
T_2	9 t_0	2 3

ne ho 2 a t_1 , che
Termino e ne rilascio
altre 2. ne ho 5, che
do a T_0 che Termino
e rilascio risorse a
 T_2 che Termino
→ 3 unità
2 disponibili

- Stato sicuro: $\langle T_1, T_0, T_2 \rangle$ sequenza sicura
- Se all'istante t_1 il thread T_2 richiede un'ulteriore unità, e questa gli viene assegnata, lo stato diventa non sicuro (solo T_1 può terminare)

T_0 NO
Termino t_1 , ne ho 4, ne bastano per T_0





Algoritmi per evitare il deadlock

- ❖ Risorse con istanza singola: **utilizzo del grafo di assegnazione delle risorse** (versione modificata)
- ❖ Risorse con istanza multipla: **algoritmo del banchiere**





Singola risorsa, modifica grafo

Algoritmo con grafo di assegnazione risorse – 1

qual è la modifica? all'inizio il thread deve dire le risorse che usa

- ❖ Un **arco di reclamo** $T_i \rightarrow R_j$ (rappresentato da una linea tratteggiata) indica che il thread T_i può richiedere la risorsa R_j in futuro *alterazione arco, non nuovo 3*
- ❖ Un arco di reclamo viene convertito in un **arco di richiesta** quando il thread T_i richiede effettivamente la risorsa R_j
- ❖ Un arco di richiesta viene convertito in un **arco di assegnazione** quando la risorsa viene allocata al thread
- ❖ Quando una risorsa viene rilasciata da un thread, l'arco di assegnazione viene riconvertito in un arco di reclamo





Algoritmo con grafo di assegnazione risorse – 2

- ❖ Le risorse devono venir reclamatione *a priori* nel sistema
 - Prima che il thread T_i inizi l'esecuzione, tutti i suoi archi di reclamo devono essere già inseriti nel grafo di assegnazione delle risorse
 - Alternativamente, un arco di reclamo può essere aggiunto successivamente al thread T_i se esso è connesso al grafo solo mediante archi di reclamo
- ❖ Supponiamo che il thread T_i richieda una risorsa R_j
- ❖ La richiesta può essere accordata solo se la conversione dell'arco di reclamo in arco di assegnazione non provoca il formarsi di un ciclo nel grafo di assegnazione delle risorse *non di graf non anche classi risorse*

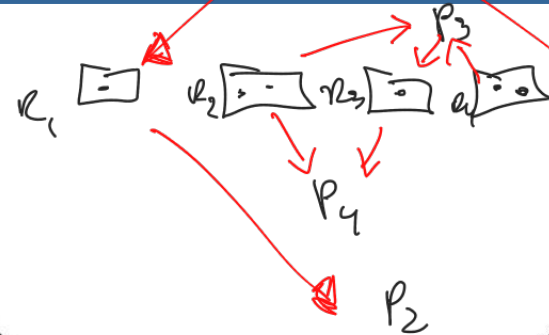
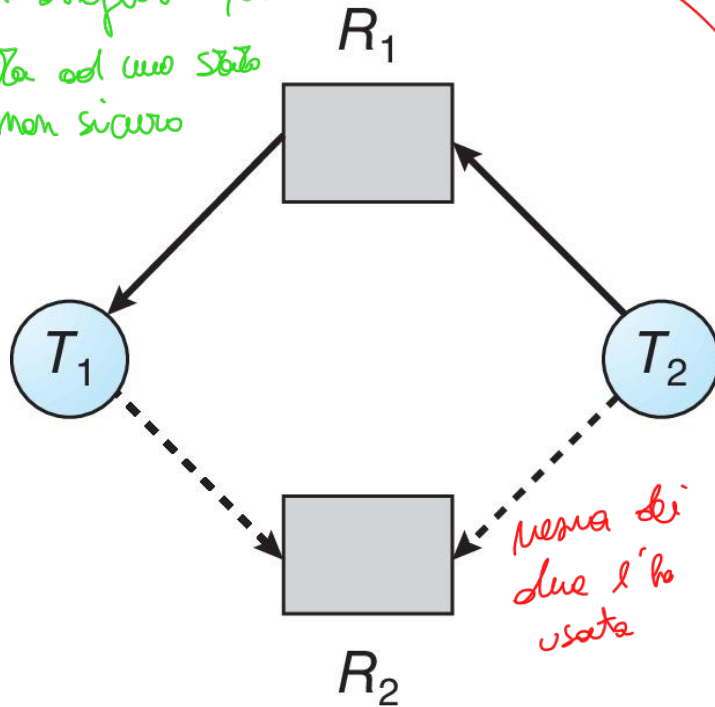
Costo della ricerca di cicli in un grafo pari a $O(n^2)$





Algoritmo con grafo di assegnazione risorse – 3

o.c. vedere per decidere
se l'operazione può
portare ad uno stato
non sicuro



• Terza P_4
 R_3 data a P_3 ,
 R_4 terza,
finisce P_2

→
complicato
analizzare
STATO NON SICURO, ma non deadlock

Grafo di assegnazione risorse

Se T_2 richiede R_2 e gli viene
assegnata...



Stato non sicuro

⇒ Stallo se T_1 richiede R_2

Esempio 2 (cont.) nuove slides.





Esempio 1

- ❖ Sia dato il sistema descritto da P insieme dei processi, R insieme delle risorse R_{ij} di indice i e molteplicità j , $E(P)$ insieme delle richieste di accesso a risorse in R emesse da processi in P e attualmente pendenti, e $E(R)$ insieme degli accessi attualmente soddisfatti di processi in P a risorse in R :

$$P = \{P_1, P_2, P_3, P_4\}$$

$$R = \{R_{11}, R_{22}, R_{31}, R_{42}\}$$

$$E(P) = \{P_1 \rightarrow R_1, P_2 \rightarrow R_4, P_3 \rightarrow R_3\}$$

$$E(R) = \{R_1 \rightarrow P_2, R_2 \rightarrow (P_3, P_4), R_3 \rightarrow P_4, R_4 \rightarrow (P_1, P_3)\}$$

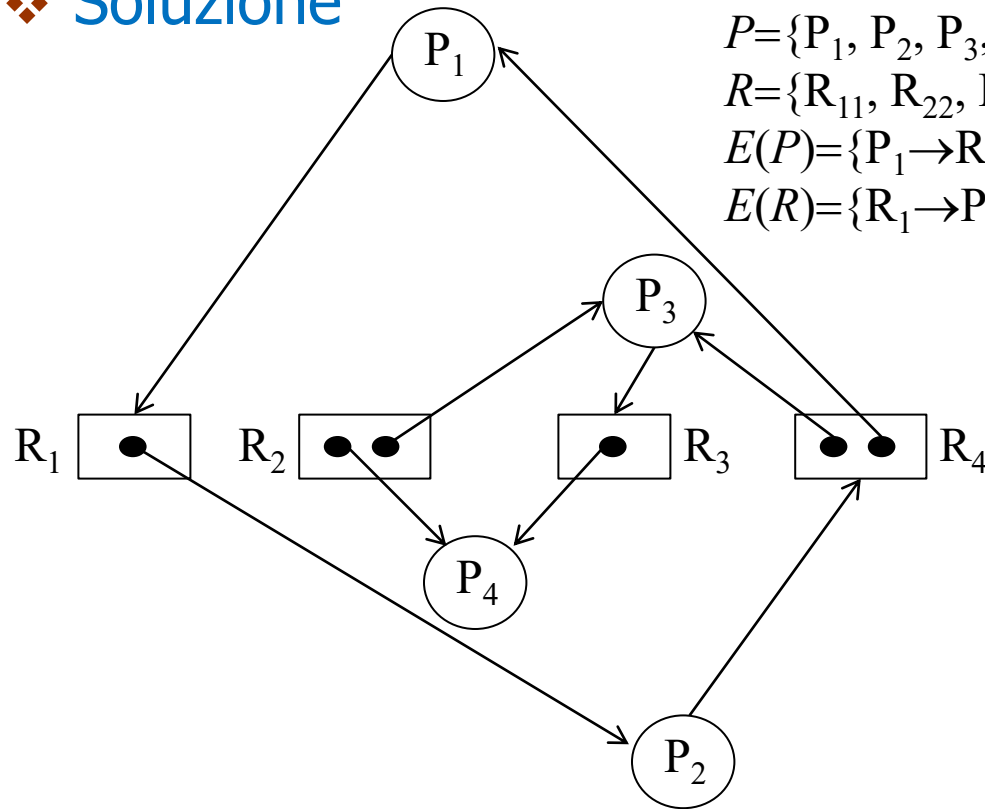
- ❖ Si analizzi il grafo di allocazione delle risorse, determinando se il sistema si trova attualmente in stallo.





Esempio 1 (cont.)

❖ Soluzione



$$P = \{P_1, P_2, P_3, P_4\}$$

$$R = \{R_{11}, R_{22}, R_{31}, R_{42}\}$$

$$E(P) = \{P_1 \rightarrow R_1, P_2 \rightarrow R_4, P_3 \rightarrow R_3\}$$

$$E(R) = \{R_1 \rightarrow P_2, R_2 \rightarrow (P_3, P_4), R_3 \rightarrow P_4, R_4 \rightarrow (P_1, P_3)\}$$

- ❖ Se P_4 termina, poiché ha tutte le risorse a disposizione, R_3 può essere concessa a P_3 , che pure terminerà rilasciando un'istanza di R_4 utile a P_2 e che interrompe il ciclo.

No DEADLOCK





Algoritmo del banchiere

- ❖ Deve il suo nome al fatto che le banche non assegnano mai tutto il denaro di cui dispongono, per non restare sprovviste di fronte a nuove richieste dei clienti
- ❖ Permette di gestire istanze multiple di una risorsa (a differenza dell'algoritmo con grafo di assegnazione risorse)
- ❖ Ciascun thread deve dichiarare a priori il massimo impiego di risorse
- ❖ Quando un thread richiede una risorsa può non venir servito istantaneamente, per mantenere lo stato sicuro
- ❖ Quando ad un thread vengono allocate tutte le risorse deve restituirle in tempo finito

la liquidità deve rimanere

senza più info sui thread

*con altri thread
possono usare*

*Sistema più
semplice ma con deadlock
non avere risorse ma non è
solo da la capo*





Strutture dati dell'algoritmo del banchiere

- ❖ Sia n il numero dei thread presenti nel sistema ed m il numero dei tipi di risorse

difficile con le strutture dati

- **Available:** vettore di lunghezza m ; se $\text{Available}[j]=k$, vi sono k istanze disponibili del tipo di risorsa R_j

m cloni risorse con k istanze, n thread

in relaz. alle cose che sono in istanze disponibili, allo stato

ripete thread, colonne lo cloni di risorse

- **Max:** matrice $n \times m$; se $\text{Max}[i,j]=k$, il thread T_i può richiedere al più k istanze del tipo di risorsa R_j

- **Allocation:** matrice $n \times m$; se $\text{Allocation}[i,j]=k$, a T_i sono attualmente allocate k istanze di R_j

si trova attuale dell'allocation delle risorse

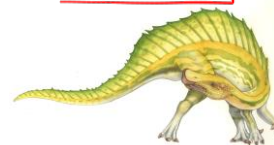
- **Need:** matrice $n \times m$; se $\text{Need}[i,j]=k$, T_i può richiedere k ulteriori istanze di R_j per completare il proprio task

il processo ci serve

parla di risorse risorse alla risorsa giusta

$$\text{Need}[i,j] = \text{Max}[i,j] - \text{Allocation}[i,j]$$

- ❖ Al trascorrere del tempo, le strutture variano sia nelle dimensioni che nei valori





Algoritmo di verifica della sicurezza

1. Siano Work e Finish vettori di lunghezza m ed n , rispettivamente; si inizializzi

a) $Work \leftarrow Available$ *con le risorse a disposizione*

b) $Finish[i] \leftarrow false$, per $i=1,2,\dots,n$ *Supponiamo, tutti i thread sono NON FINITI*

2. Si cerchi i tale che valgano contemporaneamente:

a) $Finish[i]=false$

b) $Need_i \leq Work$

Cerco un thread (i) per cui valga che l'intero processo non s'è finito e il vettore delle richieste del processo è minore o uguale a quello necessario (work)

Se tale i non esiste, si esegua il passo 4.

3. Si assegni: $Work \leftarrow Work + Allocation_i$

$Finish[i] \leftarrow true$

gli esegui risorse, lui ne ha richiesto tutte quelle il processo termina

si torni al passo 2.

4. Se $Finish[i]=true$ per ogni i , il sistema è in stato sicuro; altrimenti lo stato è non sicuro

Costo dell'algoritmo pari a $O(m \times n^2)$

Ciclo eseguito $m \times n$ volte.

cerco prima su tutti gli i poi $m-1, m-2, \dots$



o che $Finish[i]=true$ o non c'è una richiesta soddisfacibile per lo stato di esecuzione non sicuro



Algoritmo di richiesta delle risorse per il thread T_i

non vedo ne, concedendo le risorse, rimpio lo stato sicuro

- ❖ Sia **Request_i** il vettore delle richieste per il thread T_i
- ❖ Se $\text{Request}_i[j]=k$, il thread T_i richiede ulteriori k istanze del tipo di risorsa R_j *non sopra il massimo previsto*

*richieste
accettabili*

1. Se $\text{Request}_i \leq \text{Need}_i$, si vada al passo 2.; altrimenti, si riporti una condizione di errore, poiché il thread ha ecceduto il massimo numero di richieste

*risorse
disponibili*

2. Se $\text{Request}_i \leq \text{Available}$, *vedo se è soddisfacibile la richiesta in base a quella che ho*, si vada al passo 3.; altrimenti T_i deve attendere, poiché le risorse non sono disponibili

3. Il sistema simula l'allocazione a T_i delle risorse richieste, modificando lo stato di allocazione come segue:

*lo uno nuovo stato
aggiornato*

$$\text{Available} = \text{Available} - \text{Request}_i$$

$$\text{Allocation}_i = \text{Allocation}_i + \text{Request}_i$$

$$\text{Need}_i = \text{Need}_i - \text{Request}_i$$

perché non aveva richieste risolte

- ➡ Se lo stato è sicuro, le risorse vengono definitivamente allocate a T_i
- ➡ Se lo stato è non sicuro, T_i deve attendere, e viene ripristinato il vecchio stato di allocazione delle risorse





Esempio 1

❖ 5 thread, da T_0 a T_4

❖ 3 tipi di risorse:

A (10 istanze), B (5 istanze) e C (7 istanze)

❖ Istantanea al tempo t_0 :

Stato di alloc.

	<u>Allocation</u>	<u>Max</u>	<u>Available</u>
	A B C	A B C	A B C
T_0	0 <u>1</u> 0	7 5 3	3 3 2
T_1	2 0 0	3 2 2	
T_2	3 0 2	9 0 2	
T_3	2 <u>1</u> 1	2 2 2	
T_4	0 0 2	4 3 3	

7 *abete* *reside*
3 *reside* 3

$C = 2 + \text{abete}$

com $\{5, 3, 2\}$ posso
servire T_3

① calcolo Matrice NEE $D = \text{MAX-ALLOC.} - \text{SOST. el. per essere}$

Matrice NEE

T_0	7	4	3
T_1	1	2	2
T_2	6	0	0
T_3	0	1	1
T_4	4	3	1

$\{3, 3, 2\}$ *può essere*
posso servire solo

Servo T_1 $A_v = (5, 3, 2)$

lo servo anche perché con
allocazione risorse da T_1





$$Av_2 = [7, 4, 3]$$

Esempio 1 (cont.)

da qui in poi posso servire chi voglio T_4, T_2, T_0
 Great Brownie VMA
 dog. sicuro

❖ La matrice Need è definita come Need=Max–Allocation

<u>Allocation</u>			<u>Max</u>			<u>Available</u>			<u>Need</u>				
	A	B	C	A	B	C	A	B	C	A	B	C	
T_0	0	1	0	7	5	3	3	3	2	T_0	7	4	3
T_1	2	0	0	3	2	2				T_1	1	2	2
T_2	3	0	2	9	0	2				T_2	6	0	0
T_3	2	1	1	2	2	2				T_3	0	1	1
T_4	0	0	2	4	3	3				T_4	4	3	1

❖ Il sistema è in stato sicuro, dato che la sequenza $\langle T_1, T_3, T_4, T_2, T_0 \rangle$ soddisfa i criteri di sicurezza

$\swarrow$ $\searrow$
 (5,3,2) (7,4,3)... si può eseguire un T_i qualsiasi





Esempio 1 (cont.)

- ❖ T_1 richiede $(1,0,2) \Rightarrow \text{Request}_1 \leq \text{Need}_1$ $(1,0,2) \leq (1,2,2)$
- ❖ Verificare che $\text{Request}_1 \leq \text{Available}$ (cioè, $(1,0,2) \leq (3,3,2)$)

	<u>Allocation</u>	<u>Need</u>	<u>Available</u>
	A B C	A B C	A B C
T_0	0 1 0	7 4 3	2 3 0
T_1	3 0 2	0 2 0	
T_2	3 0 2	6 0 0	
T_3	2 1 1	0 1 1	
T_4	0 0 2	4 3 1	

Esempio 2 da sudice move

- ❖ L'esecuzione dell'algoritmo di sicurezza mostra che la sequenza $\langle T_1, T_3, T_4, T_0, T_2 \rangle$ soddisfa i requisiti di sicurezza
 - ⇒ La richiesta di T_1 viene soddisfatta immediatamente
 - ⇒ Ma dopo...
 - Può essere soddisfatta la richiesta di T_4 per $(3,3,0)$?
 - Può essere soddisfatta la richiesta di T_0 per $(0,2,0)$?



ALLOCATION

0	1	0
3	0	2
3	0	2
2	1	1
0	0	2

NEED

7	4	3
0	2	0
6	0	0
0	1	1
4	3	1

AVAILABLE

2, 3, 0

AVAILABLE [2, 3, 0]

Se T_1 richiede

Request, $\leq$ NEED (1, 0, 2) & (1, 1, 2)

• Posso servire solo T_1

Da SLIDES MUOVE

$$\rightarrow AV_1 = [2, 3, 0] + ALLOC T_1 = (5, 3, 2)$$

Servo T_3 ... posso anche T_4

$$AV_2 = (5, 3, 2) + (2, 1, 1) = (7, 4, 3) \dots \text{ma non posso servire l'altro.}$$

Scelta non univoca

MAX si serve da all'inizio per calcolare NEED

Se T_4 richiede 3, 3, 0, non posso soddisfarlo, richiesta non accettabile
Request: $\rightarrow$ Avrebbe

T_0 richiede (0, 2, 0) richiesta lecita, devo notare se sono in Safe

$$\bullet \text{NEW } AV = (2, 1, 0)$$

$$All \begin{bmatrix} 0 & 3 & 0 \\ 3 & 0 & 2 \\ 3 & 0 & 2 \\ 2 & 1 & 1 \\ 0 & 0 & 2 \end{bmatrix}$$

$$Max = \begin{bmatrix} 7 & 2 & 3 \\ 0 & 2 & 0 \\ 6 & 0 & 0 \\ 0 & 1 & 1 \\ 4 & 3 & 1 \end{bmatrix}$$

$$AV = (2, 1, 0)$$

non posso
servire
nessuno

per posso assegnare
non ci sarebbe
una risposta.
sicuro

Esempio 3

$$Allocazione = \begin{pmatrix} 1 & 0 & 2 & 1 & 1 \\ 2 & 0 & 1 & 1 & 1 \\ 1 & 1 & 0 & 1 & 0 \\ 1 & 1 & 1 & 1 & 0 \end{pmatrix}$$

$$MAX \begin{pmatrix} 1 & 1 & 2 & 1 & 3 \\ 2 & 2 & 2 & 1 & 1 \\ 2 & 1 & 3 & 1 & 0 \\ 1 & 1 & 2 & 2 & 1 \end{pmatrix}$$

$$NEED = \begin{matrix} T_A \\ T_B \\ T_C \\ T_D \end{matrix} \begin{pmatrix} 0 & 1 & 0 & 0 & 2 \\ 0 & 2 & 1 & 0 & 0 \\ 1 & 0 & 3 & 0 & 0 \\ 0 & 0 & 1 & 1 & 1 \end{pmatrix}$$

MAX-ALLOCATION

$$Se Available = (0, 0, X, 1, 1)$$

non posso fare T_A, T_B, T_C
ma T_D si

$$X=1, \text{ risolvo } T_D, AV_1 = (0, 0, X, 1, 1) + (1, 1, 1, 1, 0) = (1, 1, X+1, 2, 1)$$

quale che viene
 T_1

Da allocazione

→ posso fare T_3 solo se $X=2$

$$AV_2 = (1, 1, X+1, 2, 1) + (1, 1, 0, 1, 0) = (2, 2, X+1, 3, 1) \sim \text{Ri}$$

$\langle T_C, T_D, T_A, T_B, T_A \rangle$

STATO SICURO

SE $X=2$ almeno

posso fare T_B ... poi T_A



Esempio 3

- ❖ Si consideri la situazione in cui vi siano in esecuzione quattro thread, T_A , T_B , T_C e T_D , con le matrici relative alle risorse allocate ed al numero massimo di risorse richieste dai thread definite come segue:

$$\text{Allocation} = \begin{pmatrix} 1 & 0 & 2 & 1 & 1 \\ 2 & 0 & 1 & 1 & 1 \\ 1 & 1 & 0 & 1 & 0 \\ 1 & 1 & 1 & 1 & 0 \end{pmatrix} \quad \text{Max} = \begin{pmatrix} 1 & 1 & 2 & 1 & 3 \\ 2 & 2 & 2 & 1 & 1 \\ 2 & 1 & 3 & 1 & 0 \\ 1 & 1 & 2 & 2 & 1 \end{pmatrix}$$

- ❖ Se il vettore delle risorse disponibili è

$$\text{Available} = (0, 0, x, 1, 1),$$

qual è il minimo valore di x che rende lo stato attuale sicuro?





Esempio 3 (cont.)

❖ Soluzione

- Calcolo dell matrice Need

$$\text{Need} = \begin{pmatrix} 0 & 1 & 0 & 0 & 2 \\ 0 & 2 & 1 & 0 & 0 \\ 1 & 0 & 3 & 0 & 0 \\ 0 & 0 & 1 & 1 & 1 \end{pmatrix} \quad \text{Available} = (0,0,x,1,1)$$

- Nello stato corrente, può essere eseguito solo il thread T_D , ponendo $x=1$. T_D restituisce quindi le risorse che aveva allocate e **Available** diviene $(1,1,x+1,2,1)$.
- Ora, solo il thread T_C può ricevere tutte le risorse necessarie, nell'ipotesi però che $x=2$. Quindi **Available** si aggiorna a $(2,2,x+1,3,1)$.
- Si può servire T_B , senza vincoli ulteriori sul valore di x . Il vettore **Available** diviene $(4,2,x+2,4,2)$, sufficiente per T_A .
- Quindi $\langle T_D, T_C, T_B, T_A \rangle$, con $x=2$.





Esempio 4

- ❖ In un giardino sono disponibili quattro vanghe e tre carriole e ci sono quattro giardinieri al lavoro. Al giardiniere₁ occorrono complessivamente due vanghe e due carriole; al giardiniere₂ occorrono due vanghe ed una carriola; al giardiniere₃ occorrono una vanga e due carriole; al giardiniere₄ occorrono tre vanghe e tre carriole. Ad un certo istante t , giardiniere₁ sta utilizzando una carriola, giardiniere₂ una vanga, giardiniere₃ una carriola e giardiniere₄ due vanghe.
- ❖ Il sistema è sicuro? Verificarlo tramite l'algoritmo del Banchiere.
- ❖ Se giardiniere₁ avesse necessità di un'ulteriore carriola, gli potrà effettivamente essere concessa? Perché?

$$R_{tot} = (4, 3) \quad MAX = \begin{bmatrix} 2 & 2 \\ 2 & 1 \\ 1 & 2 \\ 3 & 3 \end{bmatrix} \quad Alloc = \begin{bmatrix} 0 & 1 \\ 1 & 0 \\ 0 & 1 \\ 2 & 0 \end{bmatrix}$$





AV = current allocation of Alloc

AV = $R_{TOT} - \text{escluso alloc}$

$$AV = (1, 1)$$

Esempio 4 (cont.)

NEED =

$$\begin{bmatrix} 2 & 1 \\ 1 & 1 \\ 1 & 1 \\ 1 & 3 \end{bmatrix}$$

❖ Soluzione

- Il vettore delle risorse globali è (4,3), mentre

Max = $\begin{pmatrix} 2 & 2 \\ 2 & 1 \\ 1 & 2 \\ 3 & 3 \end{pmatrix}$ Al tempo t $\longrightarrow$ Allocation = $\begin{pmatrix} 0 & 1 \\ 1 & 0 \\ 0 & 1 \\ 2 & 0 \end{pmatrix}$ Trovato AVAILABLE

Available = (1,1)

$R_{TOT} - \text{Alloc}$

- La matrice Need risulta

Need = $\begin{pmatrix} 2 & 1 \\ 1 & 1 \\ 1 & 1 \\ 1 & 3 \end{pmatrix}$ $\longrightarrow$ $\begin{matrix} \text{AV} \\ (2,1) & (2,2) & (2,3) \\ \nearrow & \nearrow & \nearrow \end{matrix}$ $\langle G_2, G_1, G_3, G_4 \rangle$ successione sicura

$$AV_1 = (1,1) \rightarrow (2,1)$$

$$\langle G_2, G_3, G_1, G_4 \rangle$$

$$AV_2 = (2,1) \rightarrow (2,2) \Rightarrow AV_3 \rightarrow (2,3), AV_4 \rightarrow (2,3)$$

$$NEED = 2, 1$$





Esempio 4 (cont.)

❖ Soluzione

- Se G_1 richiede un'ulteriore carriola si avrebbe

$$\text{Allocation} = \begin{pmatrix} 0 & 2 \\ 1 & 0 \\ 0 & 1 \\ 2 & 0 \end{pmatrix}$$

$$\text{Available} = (1, 0)$$

richiede o
corretto, ma
(1,1) available
ma solo
sicuro

di nuovo, da (1,1) non
sicuro

$$\text{Need} = \begin{pmatrix} 2 & 0 \\ 1 & 1 \\ 1 & 1 \\ 1 & 3 \end{pmatrix}$$



Nessun giardiniere può
ottenere tutte le risorse
richieste





Rilevamento del deadlock

- ❖ Se il SO non si avvale di tecniche per prevenire o evitare lo stallo, indirettamente si permette al sistema di entrare in uno stato di deadlock
- ❖ Per ripristinarne le funzionalità, sono necessari...
 - ...un algoritmo di rilevamento del deadlock *devo l'ordinare come*
 - e, successivamente, un algoritmo di ripristino dal deadlock *o uccidere thread o pulsazione v.sense*
- ❖ Costi aggiuntivi per la memorizzazione delle informazioni necessarie al ripristino, per l'esecuzione dell'algoritmo di rilevamento, e dovuti alla possibile perdita di informazioni (e calcoli già svolti) connessa al ripristino stesso *es. salvare prof. di allocazione
es. se c'è ciclo in grafico
devo metterlo costoso*





Istanza singola di ciascun tipo di risorsa

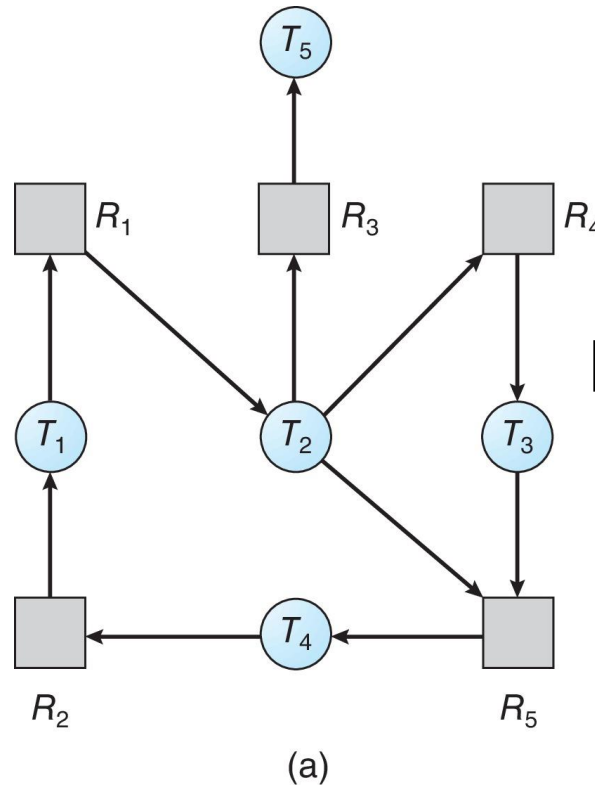
- ❖ Si impiega un **grafo di attesa** *non ci sono più due tipi di nodi*
 - I nodi rappresentano i thread *è un grafo omogeneo (tutti i nodi sono processi)*
 - L'arco $T_i \rightarrow T_j$ esiste se T_i è in attesa che T_j rilasci una risorsa che gli occorre *Arco: processo attende una risorsa che è alla fine della posse*
 - Periodicamente viene richiamato un algoritmo che verifica la presenza di cicli nel grafo (che deve essere conservato e mantenuto aggiornato) *Se si mantiene aggiornato il grafo e poi si trova un algoritmo*
 - ▶ La presenza di un ciclo attesta una situazione di deadlock
- ❖ Un algoritmo per rilevare cicli in un grafo richiede un numero di operazioni dell'ordine di n^2 , dove n è il numero di vertici del grafo *$n = \# \text{ thread}$*



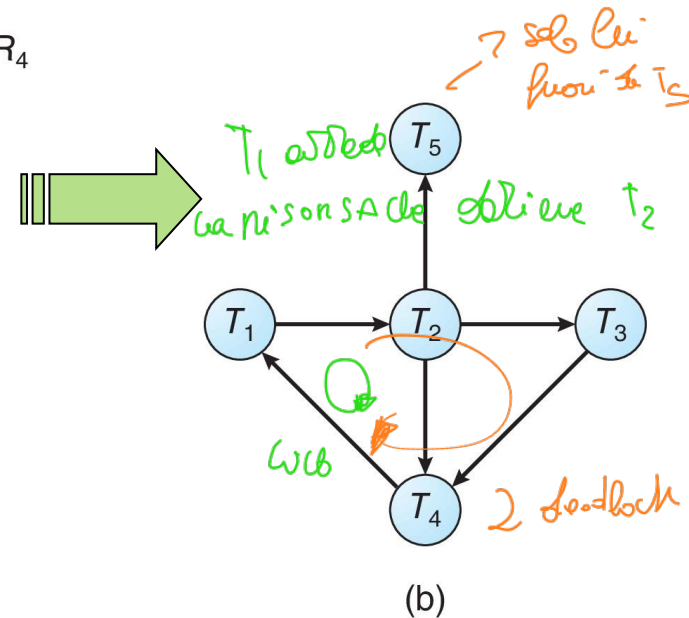


Grafo di assegnazione risorse e grafo di attesa

Un arco $T_i \rightarrow T_j$ esiste nel grafo di attesa se e solo se il corrispondente grafo di assegnazione risorse contiene i due archi $T_i \rightarrow R_q$, $R_q \rightarrow T_j$ per una qualche risorsa R_q



Grafo di assegnazione risorse



Grafo di attesa corrispondente





Toolkit BCC di Linux

L'> strumento di diagnostica

- ❖ Fornisce lo strumento **deadlock_detector** per il tracciamento delle chiamate alle funzioni Pthreads **pthread_mutex_lock()** e **pthread_mutex_unlock()** nei processi utente
- ❖ Quando il processo specificato effettua una chiamata ad una delle due funzioni, **deadlock_detector** costruisce un grafo d'attesa dei lock mutex del processo e segnala il possibile verificarsi di stalli, se rileva un ciclo nel grafo

*Chiamata quando il processo usa MUTEX con e UNLOCK
riferimento solo al processo specifico, vuole se il processo è
in una sequenza di richieste delle quali non si sblocca*





Più istanze per ciascun tipo di risorsa

Sopra: ipotesi ancora più ottimistica rispetto all'algoritmo Bankier
NON HO MAX

- ❖ Occorrono strutture dati variabili, simili a quelle utilizzate per l'algoritmo del Banchiere
- ❖ **Available:** vettore di lunghezza m , indica il numero di risorse disponibili di ciascun tipo
- ❖ **Allocation:** matrice $n \times m$, definisce il numero di risorse di ciascun tipo attualmente allocate a ciascun thread *Stato attuale dell'allocaz. di risorse a process.*
- ❖ **Request:** matrice $n \times m$, *richiede attuale* indica la richiesta corrente di ciascun thread; se $\text{Request}[i,j]=k$, il thread T_i richiede k istanze supplementari della risorsa R_j
- ❖ L'algoritmo di rilevamento indaga su ogni possibile sequenza di assegnazione per i thread da completare





Algoritmo di rilevamento – 1

1. Siano **Work** e **Finish** due vettori di lunghezza m e n , rispettivamente; inizialmente sia:
 - a) $\text{Work} \leftarrow \text{Available}$
 - b) Per $i=1,2,\dots,n$, se $\text{Allocation}_i \neq 0$, $\text{Finish}[i] \leftarrow \text{false}$; altrimenti, $\text{Finish}[i] \leftarrow \text{true}$
2. Si trovi un indice i tale che valgano entrambe le condizioni:
 - a) $\text{Finish}[i] = \text{false}$
 - b) $\text{Request}_i \leq \text{Work}$ *più potrebbe essere assegnato*Se tale i non esiste, si vada al passo 4.
3. Si assegna: $\text{Work} \leftarrow \text{Work} + \text{Allocation}_i$ *non ho max*
 $\text{Finish}[i] \leftarrow \text{true}$
Si vada al passo 2. *numero accettabile, ottimistico
se più non ha risorse, lo termina
e mi dà quello più basso
e quello da ora all'infinito*
4. Se $\text{Finish}[i] = \text{false}$, per qualche i , $1 \leq i \leq n$, il sistema è in uno stato di deadlock; inoltre, se $\text{Finish}[i] = \text{false}$, T_i è in deadlock





Algoritmo di rilevamento – 2

❖ Note

- L'algoritmo richiede un numero di operazioni dell'ordine di $O(m \times n^2)$ per determinare se il sistema è in uno stato di deadlock
- Si ipotizza (ottimisticamente) che T_i non intenda richiedere altre risorse per completare il proprio compito, ma piuttosto che restituisca in tempo finito quelle che già possiede
 - ▶ Se ciò non accade, può verificarsi uno stallo, che verrà rilevato alla prossima verifica sul sistema





Esempio di applicazione dell'algoritmo di rilevamento

SKIP ;C

- ❖ 5 thread, da T_0 a T_4
- ❖ 3 tipi di risorse:
A (7 istanze), B (2 istanze) e C (6 istanze)
- ❖ Istantanea al tempo t_0 :

	<u>Allocation</u>			<u>Request</u>			<u>Available</u>		
	A	B	C	A	B	C	A	B	C
T_0	0	1	0	0	0	0	0	0	0
T_1	2	0	0	2	0	2			
T_2	3	0	3	0	0	0			
T_3	2	1	1	1	0	0			
T_4	0	0	2	0	0	2			

- ❖ La sequenza $\langle T_0, T_2, T_1, T_3, T_4 \rangle$ conduce a $\text{Finish}[i] = \text{true}$ per ogni i





Esempio (cont.)

- ❖ T_2 richiede un'istanza supplementare della risorsa C

Request

	A	B	C
T_0	0	0	0
T_1	2	0	2
T_2	0	0	1
T_3	1	0	0
T_4	0	0	2

- ❖ Qual è lo stato del sistema?
 - Può reclamare le risorse possedute dal thread T_0 , ma il numero di risorse disponibili non è sufficiente a soddisfare gli altri thread
 - ⇒ deadlock: coinvolge i thread T_1, T_2, T_3 e T_4





quanto ha senso?

Impiego dell'algoritmo di rilevamento

❖ Quando e quanto spesso richiamare l'algoritmo di rilevamento dipende da:

- Frequenza (presunta) con la quale si verificano i deadlock *ogni quanto si verifica? monitoro stato risorse del sistema*
- Numero di thread che vengono eventualmente influenzati dal deadlock (e sui quali occorre effettuare un roll back)

▶ Almeno un thread per ciascun ciclo

*over for pinning
algoritmo qui vuole che
un processo richieda risorse e non venga
Roll back.*

❖ Se l'algoritmo viene richiamato con frequenza casuale, possono essere presenti molti cicli nel grafo delle risorse $\Rightarrow$ non si può stabilire quale dei thread coinvolti nel ciclo abbia "causato" il deadlock

→ Probabile chiudere il ciclo.

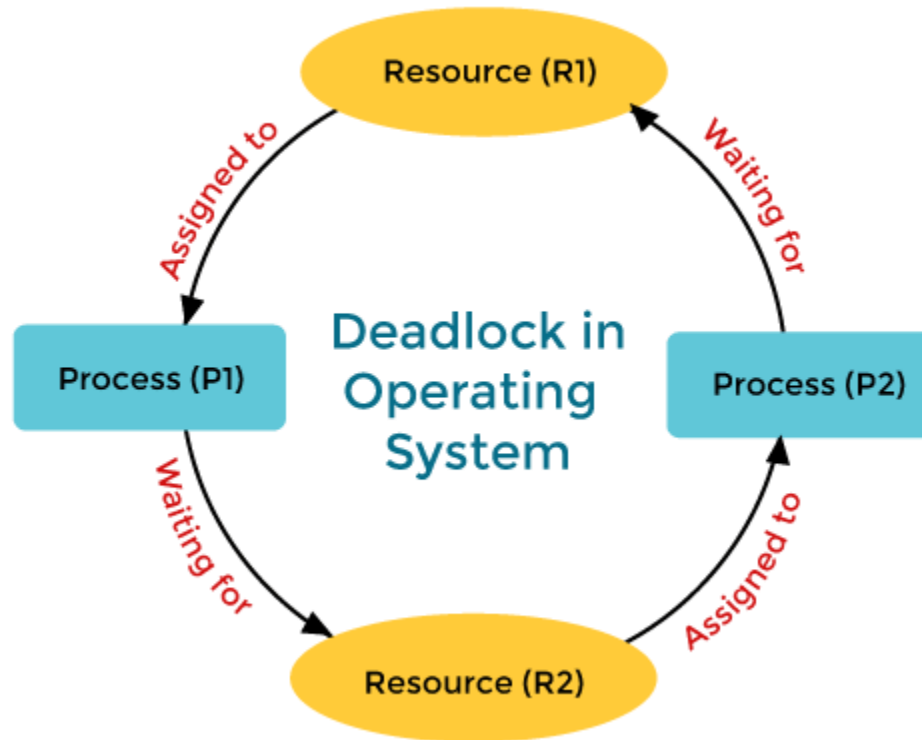
❖ Alternativamente, l'algoritmo può essere richiamato ogni volta che una richiesta non può essere soddisfatta immediatamente, a intervalli regolari, o quando l'utilizzo della CPU scende al di sotto del 40%





Ripristino dal deadlock

- ❖ Occorre interrompere l'attesa circolare
 - Terminare uno o più processi
 - Prelazionare una o più risorse





Ripristino dal deadlock: terminazione dei thread

- ❖ Terminazione in abort di tutti i thread in deadlock *es. abort*
- ❖ Terminazione in abort di un thread alla volta fino all'eliminazione del ciclo di deadlock *se faccio girare l'algoritmo*
- ❖ In quale ordine si decide di terminare i thread?
- ❖ I fattori significativi sono:
 - La priorità del thread *es. a bassa priorità*
 - Il tempo di computazione trascorso e il tempo ancora necessario al completamento del thread *kill processo che è in esecuzione da poco*
 - Quantità e tipo di risorse impiegate (non/facilmente prelazionabili) *es. non ho tempo senza formare un processo di attesa*
 - Risorse ulteriori necessarie al thread per terminare il proprio compito
 - [Numero di thread che devono essere terminati] *preferisco minimizzare*





Ripristino dal deadlock: prelazione di risorse

- ❖ **Selezione di una vittima** — È necessario stabilire l'ordine di prelazione per minimizzare i costi (numero delle risorse possedute e quantità di tempo di esecuzione già utilizzato) *Prelazione risorse, loro ripristino es. furto dati: solo risorse che aveva un lock non erano modificabili*
- ❖ **Roll back** — Un thread a cui sia stata prelazionata una risorsa deve ritornare ad uno stato sicuro, da cui ripartire
 - Per non mantenere troppe informazioni aggiuntive, la soluzione più semplice consiste nel terminare il thread
- ❖ **Starvation** — Alcuni thread possono essere sempre selezionati come vittime della prelazione: includere il numero di roll back nel fattore di costo

*considero # volte
in cui è stato ucciso*





Esercizio 1

- ❖ Si consideri la situazione in cui vi siano in esecuzione cinque thread, T_0 , T_1 , T_2 , T_3 e T_4 , con le matrici relative alle risorse allocate ed al numero massimo di risorse richieste dai thread definite come segue:

$$\text{Allocation} = \begin{pmatrix} 1 & 0 & 1 & 0 & 0 \\ 1 & 1 & 0 & 1 & 0 \\ 0 & 0 & 0 & 0 & 1 \\ 0 & 1 & 1 & 1 & 1 \\ 0 & 0 & 1 & 0 & 0 \end{pmatrix}$$

$R_{T_0 \dots}$ Available + Escluso Alloc.
 (The row sum)

$$\text{Max} = \begin{pmatrix} 2 & 0 & 2 & 1 & 3 \\ 1 & 2 & 1 & 2 & 2 \\ 2 & 1 & 3 & 1 & 2 \\ 0 & 1 & 2 & 1 & 2 \\ 1 & 1 & 2 & 2 & 2 \end{pmatrix}$$

Se il vettore delle risorse disponibili è

$$\text{Available} = (1, 0, 1, 0, 1),$$

lo stato appena descritto è sicuro? Cosa accadrebbe se il thread T_0 richiedesse e ottenesse un'ulteriore istanza della risorsa del primo tipo?



& Resource:

$$NEED = MAX - ALLOC.$$

$$\langle T_3, T_1, T_4, T_2, T_0 \rangle$$

Alloc.

$$= \begin{pmatrix} 1 & 0 & 1 & 1 & 3 \\ 0 & 1 & 1 & 1 & 2 \\ 2 & 1 & 3 & 1 & 1 \\ 0 & 0 & 1 & 0 & 1 \\ 1 & 1 & 1 & 2 & 2 \end{pmatrix} \begin{matrix} T_0 \\ T_1 \\ T_2 \\ T_3 \\ T_4 \end{matrix}$$

$$AV_1 = (1, 0, 1, 0, 1) + (0, 1, 1, 1, 1) = (1, 1, 2, 1, 2)$$

$$AV_2 = (1, 1, 2, 1, 2) + (1, 1, 0, 1, 0) = (2, 2, 2, 2, 2)$$

$$AV_3 = AV_2 + (0, 0, 1, 0, 0) = (2, 2, 3, 2, 2)$$

$$AV_4 = 2, 2, 3, 2, \dots$$

=> OK, to resource

$$\Delta_V = (0, 0, 1, 0, 1)$$

$$AV = (0, 0, 1, 0, 0) + (0, 1, 1, 1, 1) = (0, 1, 2, 1, 2)$$

$$AV_2 = (1, 2, 2, 2, 2) + (0, 0, 1, 0, 0)$$

$$AV_3 = (1, 2, 3, 2, 2)$$

STATE NON SECURED

$$\langle T_3, T_1, T_4 \rangle$$



Esercizio 2

- ❖ Si supponga che cinque processi, P_0 , P_1 , P_2 , P_3 e P_4 , condividano un insieme di risorse di quattro tipi diversi, A, B, C, D. Si supponga di trovarsi nella configurazione seguente:

	<i>Allocation</i>	<i>Max</i>	<i>Available</i>
P_0	2 4 0 6	3 X 2 7	5 12 6 8
P_1	3 0 3 5	5 7 10 6	
P_2	2 10 2 1	6 12 10 12	
P_3	6 3 3 0	X Y 5 4	
P_4	1 3 2 8	2 6 10 Z	

X $\geq$ 4

X $\geq$ 6 Y $\geq$ 3

Z $\geq$ 8

- ❖ Determinare quali sono le risorse massime che ciascun processo può richiedere tali che sia comunque garantita l'assenza di stallo a fronte dell'ulteriore richiesta di P_2 per (3, 2, 2, 7).



$$NEED = \begin{bmatrix} 1 & x-4 & 2 & 1 \\ 2 & 7 & 7 & 1 \\ 1 & 2 & 6 & 8 \\ x-6 & y-3 & 2 & 4 \\ 1 & 3 & 8 & 2-8 \end{bmatrix} \begin{matrix} P_0 \\ P_1 \\ P_2 \\ P_3 \\ P_4 \end{matrix}$$

se P_2 richiede
albero

• prima di tutto sono dopo l'allocazione

• controllo se lo stato è sicuro

$$\langle P_0, P_3, P_1 \rangle$$

• P_0 può servirlo Se $x-4 < 12 \Rightarrow x \leq 16$

$$A_v = (5, 12, 6, 8) + (2, 4, 0, 6) = (7, 16, 6, 14)$$

• $x-6 \leq 7 \quad y-3 \leq 16$ (seno P_3 se)

$$\Rightarrow x \leq 13 \quad y \leq 19$$

$$A_v = (7, 16, 6, 14) + (6, 3, 3, 0) = (13, 19, 9, 14)$$

$$A_v = A_{v2} + (3, 0, 3, 5) = (16, 19, 12, 19)$$

$$A_{v4} = A_{v3} + (2, 10, 2, 1) = (18, 29, 14, 20) \Rightarrow P_4 \quad z-8 \leq 20$$

non e' stato

variante 12

valore di ALLOCATION di PL

$$z \leq 28$$

Stato sicuro se EQRORÉ

in questo Range ho stato sicuro

$$6 \leq x \leq 13 \quad 3 \leq y \leq 19 \quad 8 \leq z \leq 28$$

eseguendo in questo ordine

• domanda es. P_2 richiede $(3, 2, ?, ?)$ All(?) = $(5, 12, 4, 8)$

$$x \geq 6 \quad y \geq 3 \quad z \geq 8$$

$< P_0, P_3, P_2,$

$$x - 4 \leq 10 \Rightarrow x \leq 14$$

$$A_v = (2, 10, 4, 1) + (2, 4, 0, 6) = (4, 14, 4, 7) \quad \text{servo } P_3$$

$$\Rightarrow x - 6 \leq 4 \Rightarrow x \leq 10 \quad y - 3 \leq 14 \quad y \leq 17$$

$$A_v = (10, 17, 7, 7) + \overset{\text{da } P_2}{(5, 12, 4, 8)} = (15, 29, 11, 15)$$



Esercizio 3

- ❖ Sia dato il sistema descritto da T insieme dei thread, R insieme delle risorse R_{ij} di indice i e molteplicità j , $E(T)$ insieme delle richieste di accesso a risorse in R emesse da thread in T e attualmente pendenti, e $E(R)$ insieme degli accessi attualmente soddisfatti di thread in T a risorse in R :

$$T = \{T_1, T_2, T_3, T_4, T_5, T_6\}$$

$$R = \{R_{12}, R_{21}, R_{32}, R_{41}\}$$

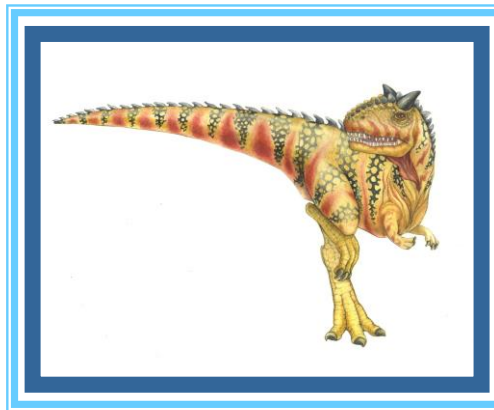
$$E(T) = \{T_2 \rightarrow R_2, T_3 \rightarrow R_4, T_4 \rightarrow R_1, T_5 \rightarrow R_4, T_6 \rightarrow R_3\}$$

$$E(R) = \{R_1 \rightarrow (T_1, T_2), R_2 \rightarrow T_3, R_3 \rightarrow (T_4, T_5), R_4 \rightarrow T_6\}$$

- ❖ Si analizzi il grafo di allocazione delle risorse, determinando se il sistema si trovi attualmente in situazione di stallo o meno
- ❖ Si studi l'evoluzione successiva dello stato del sistema se il thread T_1 richiedesse la risorsa R_2



Fine del Capitolo 8



Esercizio (1) La Casa:

T_0, T_1, T_2, T_3, T_4

Available = (1, 0, 1, 0, 1)

Allocation

T_0	2	0	1	0	0
T_1	1	1	0	1	0
T_2	0	0	0	0	1
T_3	0	1	1	1	1
T_4	0	0	1	0	0

MAX

T_0	2	0	2	1	3
T_1	1	2	1	2	2
T_2	2	1	3	1	2
T_3	0	1	2	1	2
T_4	1	1	2	2	2

NEED

T_0	1	0	1	1	3
T_1	0	1	1	1	2
T_2	2	1	3	1	1
T_3	0	0	1	0	1
T_4	0	0	1	2	2

$\langle T_3, T_1, T_4, T_2, T_0 \rangle$
Seq. sicura

$$A_{v1} = \langle 1, 1, 2, 1, 2 \rangle$$

$$A_{v3} = \langle 2, 2, 3, 2, 2 \rangle$$

$$A_{v2} = \langle 2, 2, 2, 2, 2 \rangle$$

$$A_{v4} = \langle 1, 2, 3, 2, 3 \rangle$$

$$A_{v5} = \langle 3, 2, 4, 2, 3 \rangle$$

ho risorse disponibili
e posso distribuirle

Se T_0 richiede 1 risorsa, Richiesta legittima, $Req_i \leq MAX$
 $Req_i \leq AV$

$\langle T_3, T_1, T_4 \rangle$

$$A_{v1} = \langle 0, 1, 2, 1, 2 \rangle, A_{v2} = \langle 1, 2, 2, 2, 2 \rangle$$

$$A_{v3} = \langle 1, 2, 3, 2, 2 \rangle$$

STOP! non posso servire T_2 o T_0
STATO NON SICURO

Exercise 2

	Alloc.	Max	Available
P ₀	2 4 0 6	3 X 2 7	(5, 12, 6, 8)
P ₁	3 0 3 5	5 7 10 6	
P ₂	2 10 2 1	6 12 10 12	
P ₃	6 3 3 0	X Y 5 4	
P ₄	1 3 2 8	2 6 10 Z	

P₂ richiede (3, 2, 2, 2)

Need	P ₀	1	X-4	2	1
	P ₁	2	7	7	1
	P ₂	1	0	6	4
	P ₃	X-6	Y-3	2	4
	P ₄	1	3	8	Z-8

$$\begin{array}{l} \cancel{X-4} \\ X \geq 6 \\ Y \geq 3 \\ Z \geq 8 \end{array}$$

Av (5, 12, 6, 8)

$$X \leq 16$$

$$X-4 \leq 12$$

<P₀, P₃, P₁, P₂, P₄>

$$Av_1 = (7, 16, 6, 14)$$

$$Av_2 = (13, 18, 9, 14)$$

$$Av_3 = (16, 18, 12, 18)$$

$$Av_4 = (21, 31, 16, 27)$$

$$Av_5 =$$

$$\begin{array}{l} X-6 \leq 7 \Rightarrow X \leq 13 \\ Y-3 \leq 16 \Rightarrow Y \leq 19 \end{array}$$

$$\text{Da } P_4 \quad Z-8 \leq 27$$

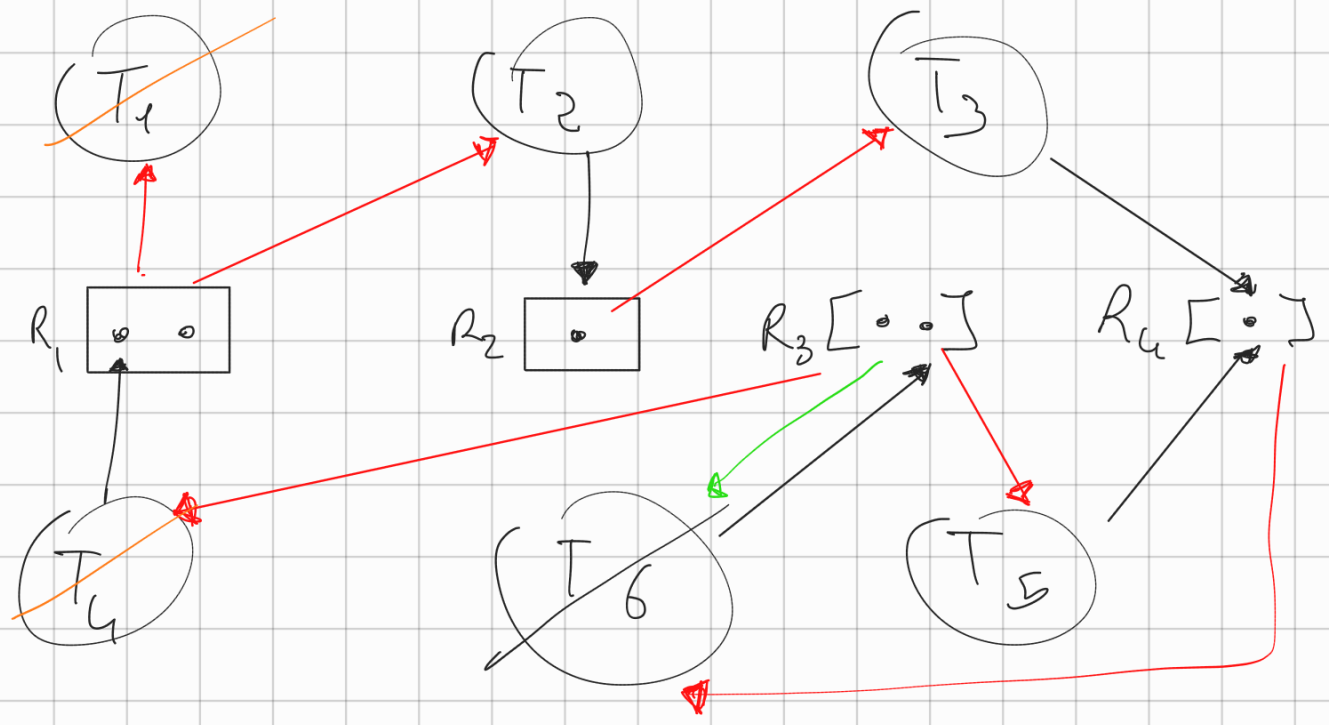
$$\Rightarrow Z \leq 35$$

$$6 \leq X \leq 16 \quad 3 \leq Y \leq 19 \quad 8 \leq Z \leq 35$$

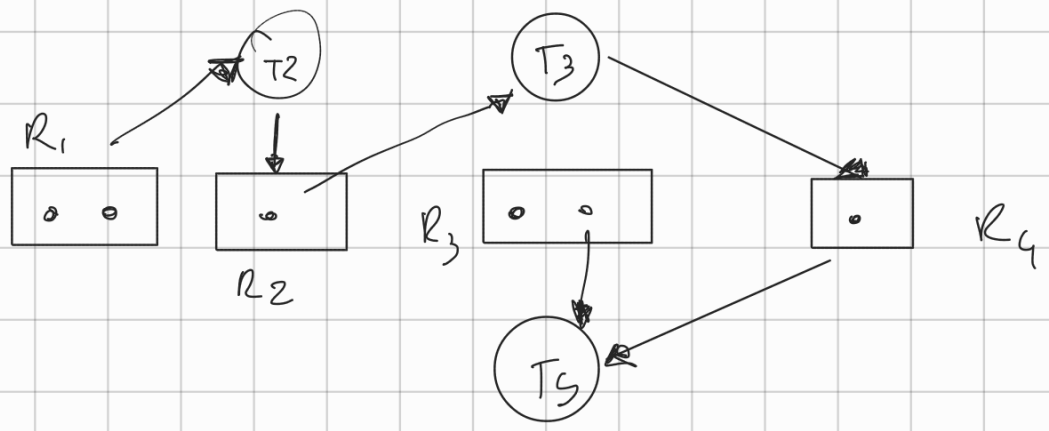
3

$$T = \{T_1, T_2, T_3, T_4, T_5, T_6\}$$

$$R_1: [\cdot \cdot] \quad R_2: [\cdot] \quad R_3: [\cdot \cdot] \quad R_4: [\cdot]$$



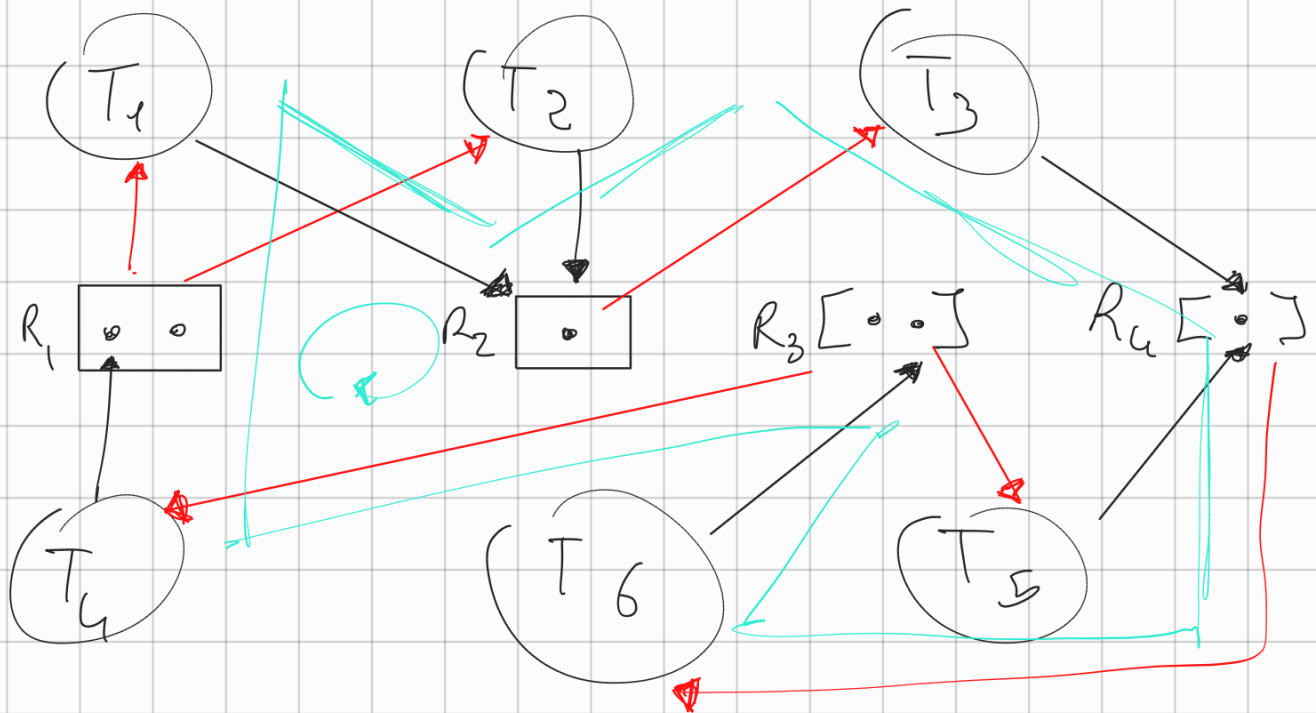
• T_1 termina, R_1 depende de T_4 , T_4 termina,
 R_3 depende de T_6 , T_6 termina



• T_6 ha sido usado R_4 , Se R_4 viene depende de T_5
 o de T_3 , termina

R_4 aspetta a T_5 , termina, T_3 termina, T_2 termina.

T_1 richiede R_1



• Stallo tra T_1, T_2, T_3, T_6, T_4