

# Sistemi Operativi

## 1. INTRODUZIONE

### Cos'è un Sistema Operativo

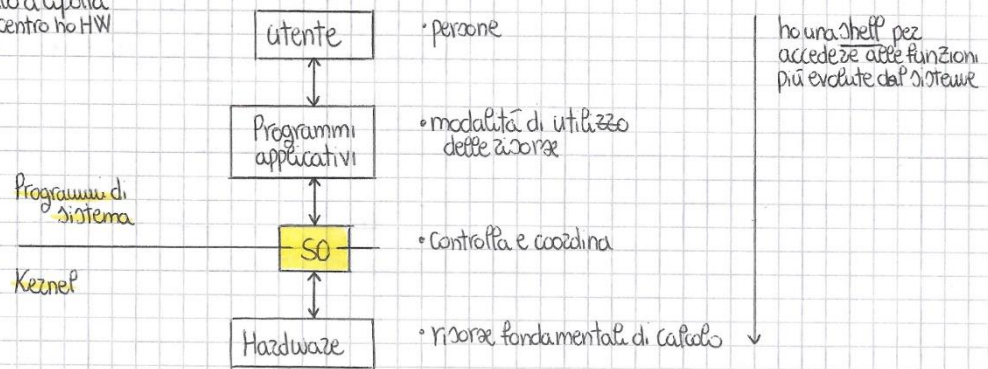
Il sistema operativo è un software che agisce da intermediazione tra l'utente e l'hardware

- un SO perfetto è invisibile all'utente, e fornisce tutti i servizi necessari per le diverse categorie di utenti

- Scopi:
- eseguire i programmi
  - rendere agevole l'interfaccia uomo-macchina
  - gestire efficientemente le risorse
  - prevenire le interferenze tra utenti
  - dare una visione "virtuale" del sistema di calcolo

### Organizzazione del sistema di calcolo

Modello a cipolla  
al centro ho HW



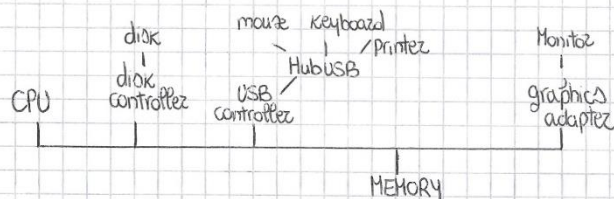
- SO funge da allocatore di risorse
- SO è un programma di controllo
- gestisce le risorse HW e SW
- arbitra i conflitti
- controlla l'esecuzione dei programmi e previene l'uso improprio

Il SO, o meglio, il kernel del SO, è l'unico programma sempre in esecuzione quando il computer è acceso

Quando: SO è nato quando i computer sono diventati general purpose

### Architettura del sistema di calcolo

- Per ogni controllore, il SO mette a disposizione un driver del dispositivo (per la gestione della specifica)



- es. il controllore della memoria si occupa della sincronizzazione degli accessi

I controllori gestiscono le periferiche diverse per tipi e gestiscono un buffer locale

## fase di avvio

Il BIOS, all'accensione, carica il programma di bootstrap

↓ memorizzato nella ROM  
FIRMWARE

↓ lo scopo principale è caricare il bootloader (sul disco fisso)

- Inizializza tutte le componenti del sistema POST
- carica il kernel del SO e ne inizia l'esecuzione

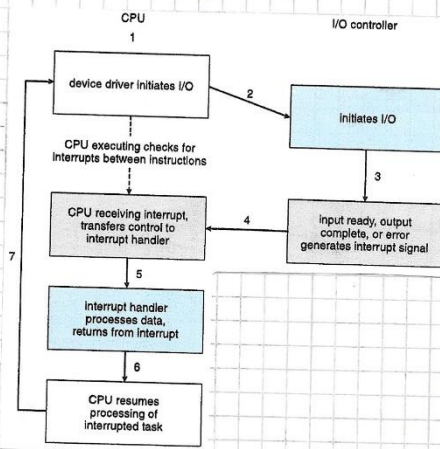
## Gli interrupt (Architettura dei calcolatori)

Interrupt => il controllo va a una ROUTINE DI GESTIONE  
è causato o da un errore o da una system call

vettore degli interrupt

NB: Salvo istruzione colpendo  
e stato CPU  
• disabilita gli altri interrupt

## Il SO è interrupt driven



HW CPU ha una o più linee di richiesta di interruzione:

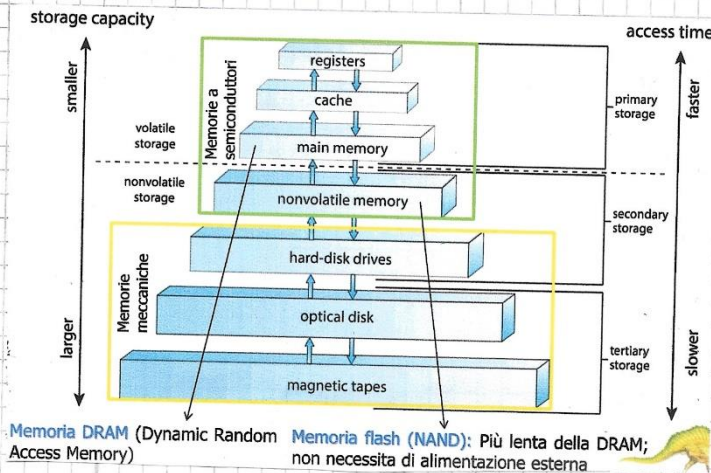
GESTORE DELLE INTERRUZIONI (HW dedicato)

- \* natura
- \* info di stato
- \* routine di servizio
- \* ritorno

\* 0-31 errori  
31-255 dispositivi

Tabella di stato dei dispositivi: gestita da SO  
↳ TIPO, INDIRIZZO, STATO

## Struttura della memoria



## Struttura dell'I/O: sono fondamentali e variabili

driver del dispositivo = SO del controllore

richiesta I/O  
↓  
interrupt

- SINCRONO: CPU in wait, una richiesta alla volta
- ASINCRONO: il contrario



## Architettura degli elaboratori

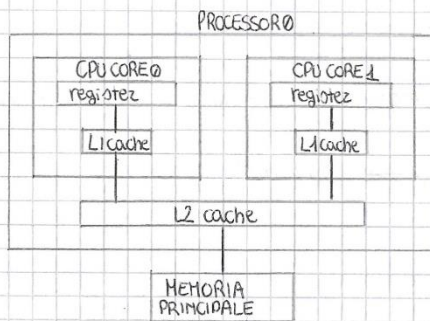
PRIMA: sistemi a singolo processore general purpose

DOPO (2005): sistemi multiprocessore o multicore

✓ Maggiore throughput: maggior numero di processi terminati in unità di tempo

✓ Migliore economia di scala: prezzi più bassi

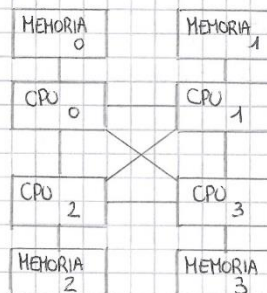
✓ Incremento dell'affidabilità



Multielaborazione

- asimmetrica: un processore che sovrintende altri processori lavorano
- simmetrica: tutti i processori eseguono

ESEMPIO: architettura NUHA



• Ogni CPU ha una propria memoria locale e sono interconnesse

✓ elimina il collo di bottiglia

✗ Accesso non uniforme alle memorie

=> SO deve fare un attento scheduling della CPU

Cluster di elaboratori: sono composti da due o più calcolatori completi (NOBI)  
si usano nell'high-performance computing

## Struttura del sistema operativo

La multiprogrammazione è fondamentale per l'efficienza del sistema

• più processi sono memorizzati contemporaneamente nella memoria principale  
=> CPU sempre occupata

ALGORITMO di CPU scheduling • se un processo è in attesa di un evento => SO seleziona un nuovo processo

Il time sharing (multitasking) è l'estensione logica • poter interagire col processo in running

• se la memoria è insufficiente, con lo swapping sposta dentro/fuori dalla memoria principale



## Attività del sistema operativo

Il SO è intereupt driven

- HW intereupt: da device
- errore del SW
- system call (kernel mode)

=>

DUAL  
MODE

per proteggere e proteggersi

(codice utente) (codice kernel) PRIVILEGIATA  
User mode e kernel mode

Per capire la modalità uso un bit di modo (che può essere esteso anche ad esempio per la virtualizzazione)

Per prevenire cicli infiniti => TIMER: emette un intereupt dopo <sup>tot</sup> tempo ed il controllo  
(che non restituiscono) il controllo ritorna al SO

## Gestione dei processi

Un processo è un programma in esecuzione.  
↓ fattivo ↓ passivo

- è l'unità di lavoro del sistema di calcolo
- necessita: tempo di CPU, memoria, file di dispositivi I/O, dati di inizializzazione
- terminare un processo significa recuperare le risorse

→ single thread: PC unico (locazione di memoria della prossima istruzione da eseguire)

Processi → multi thread: PC per ogni thread

SO è responsabile:

- Cancellare e creare processi
- Sospensione e attivazione processi
- Scheduling
- Comunicazione e sincronizzazione di processi

## Gestione della memoria centrale e della memoria secondaria/terziaria

+ La memoria centrale è un retore di parole, ciascuna con un proprio indirizzo  
↓  
è un deposito di dati, ma lento rispetto la CPU

Per eseguire un programma, tutte le istruzioni devono risiedere in memoria centrale

SO è responsabile:

- Tenere traccia della memoria usate
- Allocare/deallocare lo spazio per necessità
- Decidere quali processi caricare in memoria

+ So dà una visione logica della memoria secondaria

- ↓
- unità di memorizzazione logica: il file
  - Ogni periferica ha il proprio device driver

NB Memoria centrale e memoria secondaria devono collaborare  
(volatile e piccola) (permanente e veloce)  
RAM SSD

SO è responsabile:

- Partizionare
- Montaggio e smontaggio
- Gestione spazio libero
- Scheduling disco
- Protezione

+ Per la memoria terziaria vale lo stesso  
tranne che non è necessario che siano  
veloci



## Gestione file system

Il file è una collezione di informazioni, organizzati in directory

- SO è responsabile:
- creazione/cancellazione directory e file
  - associazione alla memoria
  - funzioni per la manipolazione
  - backup

NB

Spostamento dei dati

- ESPLICITO (mediato da SO) es da disco a RAM
- IMPLICITO (mediante HW) es da cache a registro

PROBLEMA: coerenza della cache

## Gestione dispositivi I/O

SO deve nascondere le caratteristiche dei dispositivi HW

- buffering: memorizzazione temporanea
- caching: memorizzazione parziale
- spooling: per I/O asincrono

## Protezione e sicurezza

**Protezione:** è l'insieme di tecniche usate per controllare l'accesso alle risorse di calcolo del sistema

↓  
più utenti non possono accedere alle stesse risorse

**Sicurezza:** è la strategia di difesa implementata dal sistema per proteggerla da attacchi esterni/interni  
prevenzione dall'uso illegale  
es. Denial of Service, Trojan, Worm, Virus ...

CHI: Hacker: pirata informatico

Cracker: che cerca di trarne guadagno

SOLUZIONE: • distinguere gli utenti con user e password

- proprietà del file -CS PS -P pippo.c
- RWX r-- r--

Permettono di  
• r: read  
• w: write  
• x: execute

## Virtualizzazione

**Virtualizzazione:** tecnica per eseguire un SO come applicazione all'interno di un altro SO

- Ogni processo ospite usufruisce di una copia virtuale del calcolatore
- ↓ software di emulazione (su piattaforme diverse)

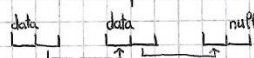
- Le risorse sono condivise dall'hypervisor (modulo o micro-kernel)
  - HW e SO sono eseguiti in un ambiente isolato
- garantisce la protezione

- + scheduling CPU
- + spooling
- + divisione spazio disco
- + bit di modo

## Strutture dati del kernel

■ **ARRAY**: ogni elemento è direttamente accessibile tramite un indice

↓ **liste concatenate** per dati di dimensione variabile ed eliminazione elementi

 Costo ricerca  $O(n)$

⇒ **pile**  
**code**

- allocazione di file
- algoritmo clock per la sostituzione di pagine

• **Stack**: le operazioni di inserimento e svuotamento, con politica LIFO (Last in First out)

↓ push      ↓ pop

Si può usare per la gestione di chiamate di funzione, ed i cambi di contesto

• **Code**: con politica FIFO (First in First out)

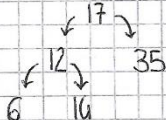
Si può usare per i documenti dati alla stampante, o processi in attesa

■ **ALBERI**: per organizzare i dati in maniera gerarchica (relazione padre-figlio)

- attività: da ciascun nodo possono nascere più altri

+ **albero binario di ricerca**: relazione d'ordine figlio left < padre < figlio right

Costo ricerca  $O(\log_2 n)$



■ **Funzioni hash**: si cerca un modo per ottimizzare la ricerca ad  $O(1)$

(minimizzare)  
 $h: K \rightarrow \{0, 1, \dots, n-1\}$

↓ insieme dei valori distinti che assumono le chiavi dei record

• n: dimensione del vettore  
in cui voglio memorizzare  
la tabella

$h(K) = K \text{ MOD } n$

• se K non è un insieme di numeri allora devo fare una associazione

ma possono avere **collisione**: a chiavi distinte dà la stessa posizione

■ **bitmap**: stringa di n caratteri binari per rappresentare l'esistenza di n elementi

es. spazio di gestione dei dischi



## Ambienti di calcolo

### Elaborazione tradizionale

Ad oggi, non c'è molta distinzione tra i vari ambienti.

+ Per ufficio: • PC con rete e server per accedere ai file  
↓  
• Portali con accesso ad Internet

+ Nelle case: • PC con modem  
↓  
• Reti locali con Internet

Sistemi distribuiti: insieme di calcolatori che non condividono memoria, né clock ma hanno una rete di comunicazione

WAN, MAN, LAN, PAN con protocollo TCP/IP

Modello client-server: + CLIENT: sono i PC e richiedono servizi ai server

+ SERVER: permettono gli accessi alle risorse

Modello peer-to-peer: tutti i nodi sono equivalenti (sia client che server), con un registro centralizzato per salvare l'identità dei nodi

### Cloud computing

Sistemi embedded: molto semplici

Permettono SO real-time • Vincoli sui tempi di esecuzione (hard)

## Sistemi operativi open-source

Vantaggi: ✓ Non è necessario il reverse-engineering per capirli  
✓ Comunità di programmatori ed aziende per lo sviluppo  
✓ Codice sicuro e bug fixati velocemente  
✓ Libertà

Svantaggi: X Copyright

Storia: 1983 Richard Stallman => GNU (is NOT a UNIX)  
SO gratuito, open-source

=> Copyleft: è il permesso di autore, ovvero se si vuole distribuire una opera modificata, ciò va fatto sotto lo stesso regime giuridico

• 1985: Manifesto GNU  
→ tutti i software dovrebbero essere gratuiti ed open-source

• 1991: Linus Torvalds rilascia UNIX => LINUX

- LINUX: SO open-source
- Windows: proprietario e closed-source
- Mac OS: ibrido
- Solaris:

## 2. STRUTTURA DEI SISTEMI OPERATIVI

I SO forniscono un ambiente dove a eleggono i programmi

La sua progettazione è completa, deve essere definito/valutato in base a:

- servizi da fornire
- interfaccia per programmatori ed utenti

### Servizi del sistema operativo

#### Servizi user-oriented

Interfaccia utente:

- linea di comandi (CLI)
- grafica (GUI)
- touch screen

Esecuzione di programmi: capacità di caricare un programma in memoria ed eseguirlo

Operazioni di I/O: mezzi per effettuare l'I/O su file o periferica

Gestione del file system: creare, leggere, scrivere e cancellare file  
navigare nella struttura delle directory

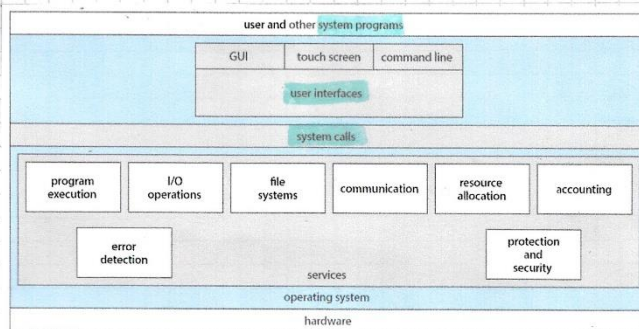
Comunicazioni: scambio di informazioni fra processi

- memoria condivisa
- scambio di messaggi

Riferimento di errori

In più per migliorare l'efficienza:

- + Allocations di risorse
- + Accounting e contabilizzazione dell'uso delle risorse
- + Protezione e sicurezza



schema dei servizi



## Interfaccia utente

**Interfaccia utente CLI**: permette di impartire direttamente comandi al SO

- implementata
  - nel kernel
  - con programmi di sistema
- personalizzabile, più shell per ambienti diversi

interpreta ed esegue le istruzioni di comando in due modalità:

- Se il codice relativo al comando è parte del codice dell'interprete  $\Rightarrow$  salto alla sezione
  - ogni comando richiede il proprio segmento di codice
  - il numero dei programmi implementati  $\Rightarrow$  dimensione dell'interprete
- Implemento nuovi comandi con programmi di sistema (`/usr/bin`)
  - interprete non viene modificato

- ✓ Semplifica l'esecuzione di comandi ripetuti
- ✓ Un task frequente può diventare script

ESEMPIO (Linux)

`$ rm file.txt`  $\rightarrow$  `-i` lo cancella solo dopo conferma  
 $\downarrow$  cerca l'eseguibile chiamato `rm`, lo cerca in memoria, lo esegue con parametro `file.txt`  
esegue l'eliminazione del file `file.txt`

ESEMPIO

script `primo.sh`  $\rightarrow$  `cd` `mkdir d1` `chmod 666 d1` `cd d1`  
Cambia directory corrente  
crea la directory `d1`  
imposta i permessi `r-- r-- r--`  
fallire  $\rightarrow$  permesso negato

**Interfaccia utente GUI**: è user-friendly, nati negli anni '70 da Xerox PARC

NB Linux è modulare  
sia CLI che GUI

- Interazione semplice
- Utilizzo del mouse
- Uso di icone

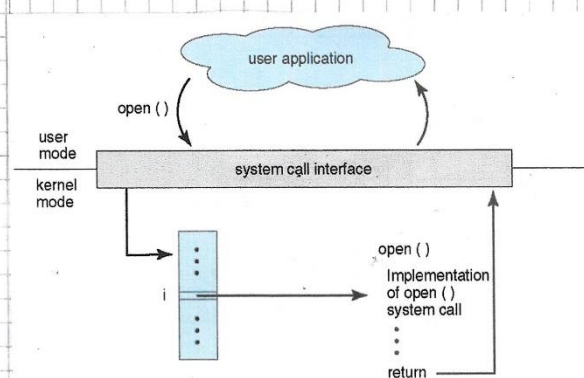
## Chiamate di sistema

**Chiamate di sistema**: forniscono l'interfaccia fra i processi e i servizi offerti dal SO

- realizzate in C++
- richiamate dagli applicativi attraverso API (Application Programming Interface)
  - Win64 API
  - POSIX API
  - Java API

a ciascuna system call è associato un numero (ovvero una tabella)

l'utente non deve conoscere l'implementazione, ma solo le modalità di utilizzo dell'API  $\rightarrow$  funzioni di libreria standard





3 metodi per passare parametri al SO:

1. Passaggio di parametri nei registri (a volte può non bastare)
2. Memorizzazione dei parametri in un blocco di memoria e passaggio dell'indirizzo del blocco
3. Push dei parametri nello stack ed SO recupera con un pop

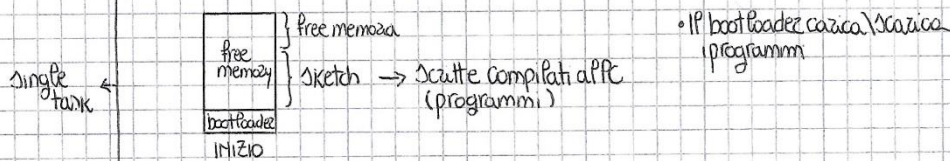
<5 parametri  
>5 parametri

## Tipi di chiamate di sistema

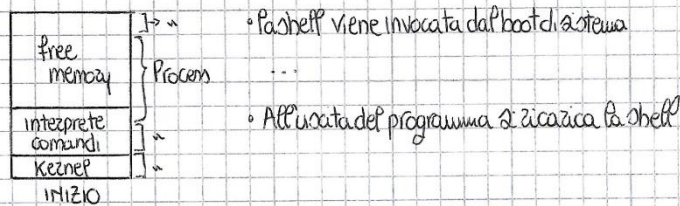
### CONTROLLO DEI PROCESSI

- Creazione/terminazione di processi (fork, exit)
- Caricamento/esecuzione (exec, execve)
- Lettura/modifica degli attributi (get-set process attributes)
- Attesa (wait)
- Alloggiamento/riutilizzo memoria (alloc, free)
- Invio di segnali (signal, kill)
- Stampa della mappa di memoria
- Debugger
- Lock e memoria condivisa

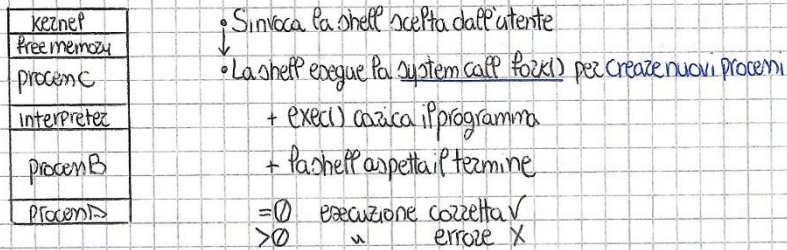
esempio: Arduino (hardware + microcontrollore e sensori) (NO SO)



esempio: MS-DOS



Multi task ← esempio: FreeBSD



### GESTIONE DEI FILE

- Creazione/cancellazione di file (create, delete)
- Apertura/chiusura di file (open, close)
- Lettura, scrittura e posizionamento (read, write, seek)
- Lettura/modifica attributi

### GESTIONE DEI DISPOSITIVI

- Richiesta/riutilizzo dispositivo (request, release)
- Connessione/disconnessione logica (ioctl)

→ Sono molto simili



## GESTIONE DELLE INFORMAZIONI

- Lettura/Modifica di ora/data
- Informazioni sul sistema
- Lettura/Modifica attributi (ps, getpid)

## COMUNICAZIONE

- Apertura/chiusura connessione
- Invio/ricezione messaggi
- Stato dei trasferimenti
- Condivisione della memoria

## PROTEZIONE

- Controllo di accesso alle risorse
- Lettura/Modifica dei permessi/chiamate
- Accredimento degli utenti

## Programmi di sistema

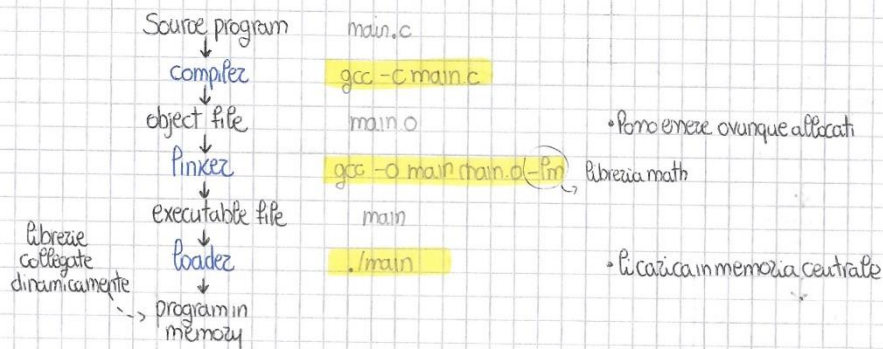
Programmi di sistema forniscono un ambiente conveniente allo sviluppo e l'esecuzione di programmi utente

- Gestione file (cp)
- Editing file (vi) ...
- Informazioni di stato (who)
- Supporto a linguaggi di programmazione

per l'utente il PS è definito dai programmi di sistema

- Servizi in background: daemon

## Linker e Loader



- Applicazioni compilate su un sistema non sono eseguibili su altri SO

lo sono se:

- Scritte in linguaggio interpretato (Python)
- Include un VM (in Java)
- Standard (C) ma compilate a parte

## Progettazione e realizzazione di SO

La struttura interna di diversi SO può variare:

- Hardware
- Scopo del sistema

- Richieste utente: semplice, affidabile, sicuro e veloce
- Obiettivi del sistema: semplice da progettare e realizzare, flessibile, error-free

Concetti: Politiche: quali sono i compiti del SO? Cosa è necessario fare?

≠

Meccanismi: come realizzarli?

Siccome un mix di linguaggi:

- basso livello  $\Rightarrow$  ASSEMBLY
- kernel  $\Rightarrow$  C
- programmi sistema  $\Rightarrow$  C++, C + ...

• Windows MacOS: firmati a priori

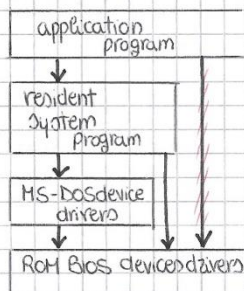
• Linux: separati

## Struttura del sistema operativo

### • MONOLITICA (Sistemi storici)

- + SO è identificato col kernel che realizza le funzioni di gestione delle risorse
  - + Tutti i moduli operano sullo stesso spazio di indirizzi (es. molto tutto nel main)
- X un bug rompe tutto  
✓ estremamente efficiente

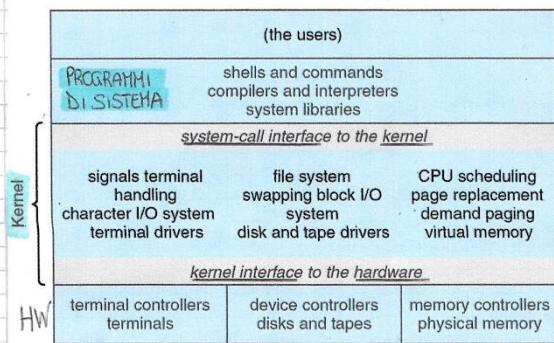
- MS-DOS: maggior numero di funzionalità usando minor spazio
- Non ha moduli



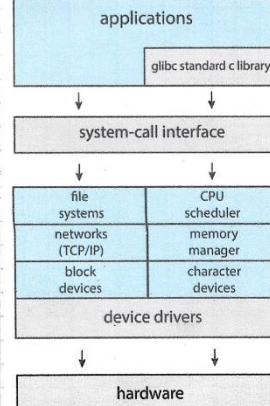
X l'utente interagisce con tutto

- UNIX (accostamente stratificato)

Beyond simple but not fully layered



Monolithic plus modular design

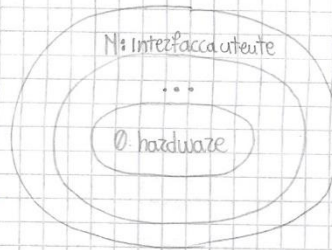


nuovi moduli possono essere caricati a run-time



## • APPROCCIO STRATIFICATO

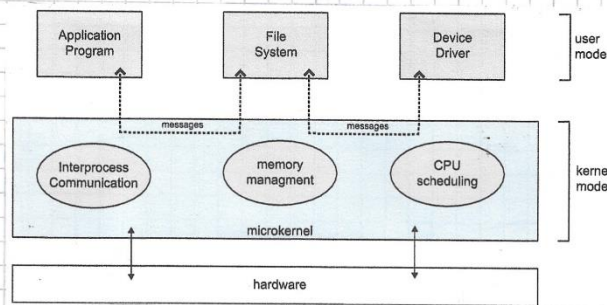
- ✓ Si garantisce il controllo sulle applicazioni
- Ciascuno strato impiega esclusivamente funzioni e servizi di strati di livello inferiore
- ✓ Semplice realizzazione
- ✗ Difficile definire i vari strati
- ✗ Scarsa efficienza
- TCP/IP
- Linux



## • STRUTTURA DEI SISTEMI MICROKERNEL

- Le funzionalità del kernel sono spostate nello spazio utente
- Offre i servizi minimi di gestione dei processi, della memoria e di comunicazione

- ✓ Funzionalità più semplici da estendere (non comportano modifiche al kernel)
- ✓ Sicuro e affidabile
- ✓ Facile da portare su nuove architetture
- ✗ Decadimento delle prestazioni per l'overhead (troppe richieste)



## • KERNEL MODULARI (a strati malleabili)

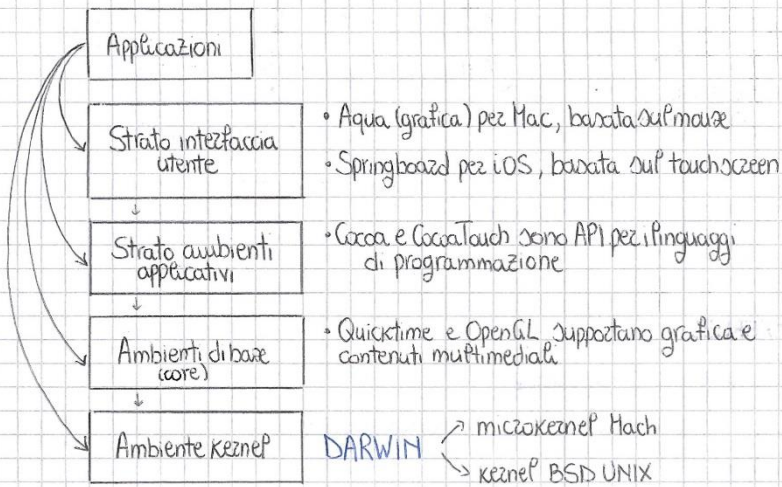
- + Programmazione ad oggetti
- + Ciascun modulo implementa una componente base del kernel colloquia con gli altri caricato o no in memoria

es. Linux, Solaris

## • SISTEMI IBRIDI: gli SO moderni combinano diverse approcci

- + Kernel Linux e Solaris:
  - monolitici: SO è in un unico spazio di indirizzamento → migliori prestazioni
  - modulari: nuove funzionalità possono essere aggiunte dinamicamente
- + Windows:
  - monolitico
  - microkernel: supporta sottosistemi separati (personalità)
- + MacOS e iOS
- + Android

## +Mac OS e iOS

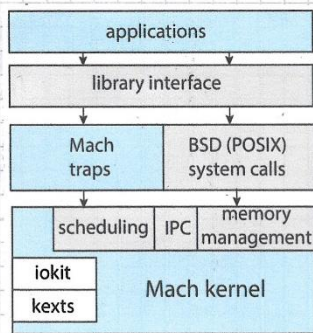


MAC: destinato a Laptop e desktop, architettura Intel/ARM

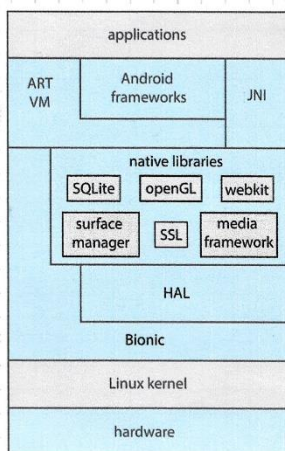
iOS: destinato a Smartphone, architettura ARM

- ↳ risparmio energetico
- ↳ ottimizzazione memoria
- ↳ sicurezza severa

→ Testativo



## +Android (open-source)



Sviluppato da Open Handset Alliance, basato su kernel Linux



## Generazione e avvio del SO

SO sono progettati per funzionare su una classe di architetture, con una varietà di periferiche.

- SO è già installato, ma ne possono installare anche altri

from scratch:

(scrivere il codice sorgente)

1. Configurare SO per lo specifico sistema di calcolo
2. Compilare SO
3. Installare SO
4. Avviare il computer

GENERARE SO - leggere da file le informazioni per la configurazione  
- determinare i componenti del sistema di calcolo

- Tipo di CPU
- Tipo di formattazione del disco
- Quantità di memoria
- Dispositivi disponibili
- Politiche varie

+ l'esecuzione inizia da una posizione di memoria predefinita

+ l'hardware deve avviarlo

bootstrap loader: (BIOS) memorizzato in ROM, individua il kernel, lo carica in memoria e lo avvia.

NB In Linux: l'immagine del kernel è un file compresso, che viene estratto in RAM. Il boot loader crea un filesystem temporaneo in RAM **initramfs** ...  
→ crea il processo iniziale **systemd**  
prompt di login

## Debugging del sistema operativo

Debugging: individuazione e risoluzione di errori nel sistema (bug)  
⇒ **performance tuning** (eliminazione colli di bottiglia)

SO può: - generare file di log, che danno informazioni sugli errori verificati  
- acquisire e memorizzare immagine del contenuto della memoria ⇒ **core-dump**

istantanea post-mortem  
dello stato del processo al momento della terminazione anomala

+ Un guasto nel kernel è chiamato **crash** → core dump

- il crash dump è rischioso se il kernel è instabile/inconsistente

→ crash dump: viene salvato lo stato della memoria

Il dump è salvato in un'area di memoria dedicata e da lì recuperato per non rischiare di compromettere il file system

**Legge di Kernigham** "Il debugging è due volte più difficile rispetto alla stesura del codice."

**Performance tuning** è l'insieme delle tecniche per ottimizzare le prestazioni del sistema.

- Analisi dei dati
- Monitorazione del sistema
- Profiling: campionamento periodico IP, quali chiamate di sistema sono utilizzate maggiormente

NB su Linux: BCC (BPF Compiler Collection)



### 3. I PROCESSI

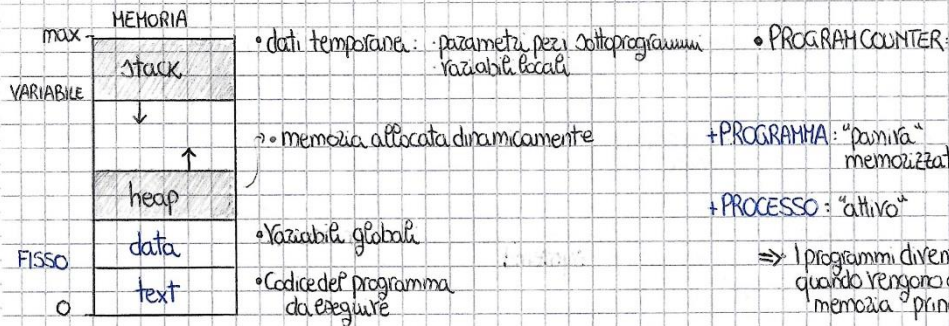
#### Definizione di processo

SO esegue programmi di varia natura: SISTEMI batch (job)

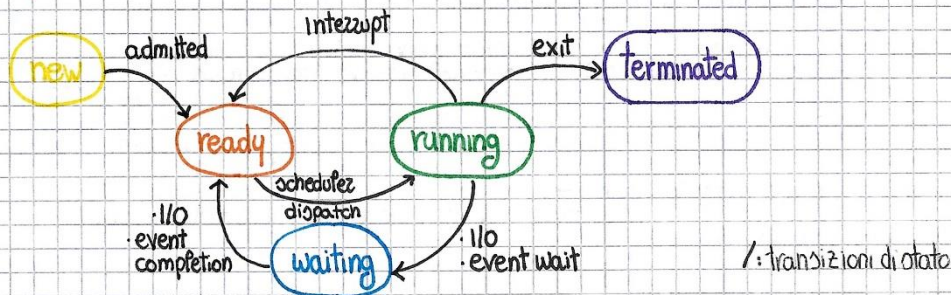
- time-sharing (processi utente o task)
- multithreading (thread)

**Processo:** - un programma in esecuzione

- sequenza di eventi che avvengono in un elaboratore sotto il controllo di un particolare programma



#### Stato del processo



• **New**: il processo viene creato (caricamento codice da disco a RAM)

• **Ready**: il processo è in attesa di essere assegnato ad un processore

• **Running**: viene eseguita l'istruzione

• **Waiting**: il processo è in attesa di un evento

• **Terminated**: il processo ha terminato la propria esecuzione

DISPATCHER: il processo viene scelto

ADMITTED: il processo è ammesso alla contesa attiva per la CPU

INTERRUPT: scheduling a porzione di tempo esterno (avvicinamento)

EXIT: • terminazione normale  
• terminazione anomala



## Process Control Block

Ad ogni processo è associata una struttura dati del kernel (PCB)

Le informazioni contenute nel PCB sono vitali per il SO:

devono essere memorizzate in un'area di memoria accessibile solo dal kernel del SO

|                     |
|---------------------|
| stato del processo  |
| numero del processo |
| program counter     |
| registers           |
| memory limits       |
| list of open files  |
| ...                 |

Contesto del processo

Informazioni

- scheduling della CPU
- gestione della memoria allocata
- contabilizzazione delle risorse
- su I/O

## Thread

Il processo è basato su due aspetti: - Possono della risorse e sono indipendenti

- Esecuzione

Per il SO: + l'elemento che viene eseguito è il thread

+ l'elemento che possiede le risorse è il processo

NB Multithreading: ad un processo sono assegnati più thread

• Tutti i thread condividono le risorse del processo, e sono nello stesso spazio di indirizzamento

• Ad ogni thread è associato un PC, uno stato di esecuzione, uno spazio di memoria, uno stack ...

PCB

è un descrittore

## Processi in Linux

I PCB sono chiamati TCB (Task), e rappresentato dalla struttura C `task_struct` nel file `linux/sched.h`

contiene:

- Identificatore del processo PID

`pid_t pid;`

- Stato del processo

`long state;`

- Info sullo scheduling

`unsigned int time_slice;`

- Lista file aperti

`struct files_struct *files;`

- Puntatori a processo padre e figli

`struct task_struct *parent;`

`struct list_head *children;`

`struct mm_struct *mm;`

La ready queue è realizzata con lista doppiamente concatenata `task_struct` (così non perdodati)  
il kernel mantiene un puntatore `current` al processo attualmente in esecuzione

Per cambiare stato `current->state = nuovo_stato;`

## Scheduling dei processi

Lo scheduling è messo in atto per massimizzare l'utilizzo della CPU

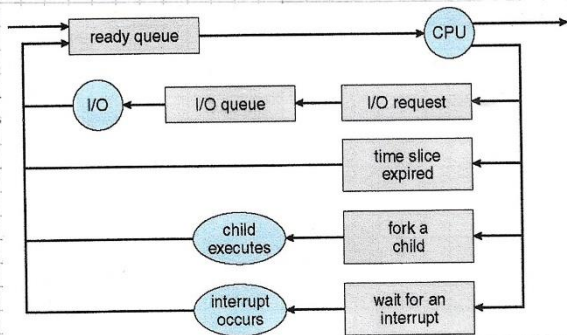
• Passare rapidamente da un processo all'altro con time sharing

↓ Modulo SO: CPU scheduler

Code per lo scheduling: + **READY QUEUE**: insieme dei processi pronti ed in attesa di essere eseguiti che risiedono in memoria centrale

+ **CODE IN ATTESA**: insieme dei processi in attesa di I/O o di un evento

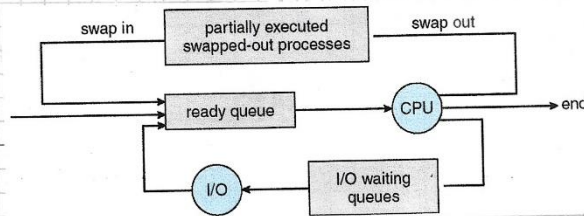
- I processi migrano fra varie code
- Un nuovo processo va nella ready queue
- Un processo rilascia la CPU:
  - emette I/O e viene inserito nella coda di attesa
  - Crea un processo figlio e attende la terminazione
  - Rimosso forzato



## Tipi di scheduling

+ Scheduling a **breve termine**: seleziona quale processo deve essere eseguito in successione • veloce

+ Scheduling a **medio termine**: rimuove temporaneamente i processi dalla memoria e regola il grado di multiprogrammazione



Processi → I/O bound: più tempo effettuando I/O

→ CPU bound: più tempo effettuando elaborazioni

• molti burst CPU di breve durata

• pochi burst CPU di lunga durata



## Context switch

Quando la CPU passa da un processo all'altro, il sistema deve salvare il contesto e poi caricare il contesto del nuovo processo in esecuzione.

Il contesto è all'interno del PCB.

Il tempo di context-switch è un sottacostoso, dipende dal supporto hardware.

## Multi-tasking per sistemi mobili

- Alcuni SO (vecchi) prevedevano un processo solo in esecuzione.
- iOS : + un processo in foreground => ora ce ne possono eseguire di più (split-view)  
+ più processi in background (che non possono usare il display)
- Android: un'applicazione in background può continuare ad elaborare servizi (es. Spotify)

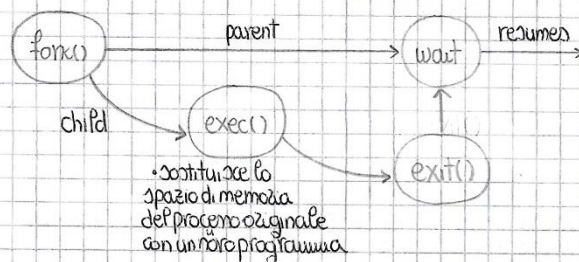
## Operazione sui processi

SO deve fornire meccanismi su: creazione e terminazione di processi che sono eseguiti in concorrenza.

### Creazione dei processi

- Un processo può creare altri processi (albero dei processi)  
padre → figlio NB: il padre di tutti i processi è **systemd**
- I processi sono identificati con un pid
- + Condivisione di risorse
  - oppure: padre e figlio condividono tutte le risorse
  - oppure: il figlio condivide un sottoinsieme delle risorse
  - oppure: padre e figlio non condividono risorse
- + Esecuzione
  - oppure: p e f sono eseguiti in concorrenza
  - oppure: padre attende il figlio
- + Spazio degli indirizzi
  - oppure: il figlio è duplicato del padre
  - oppure: figlio ha caricato un altro programma

UNIX/Linux:



PID → figlio è zero  
→ padre è un valore > 0

- **execfp()**: sostituisce il processo corrente con un nuovo processo  
↓  
primo argomento: nome del programma che verrà eseguito  
ultimo: puntatore a NULL  
termina correttamente non effettuando ritorno al chiamante  
ultimi -1



## Terminazione di processi

Un processo termina quando esegue l'ultima istruzione e chiede al SO di essere cancellato (`exit()`)

- può restituire dati al processo padre
- le risorse vengono deallocate dal SO

↓  
`abort()` termine  
volontario anomalo

• Il padre può terminare l'esecuzione dei processi figli `Kill(pid, signal)`;

- es.:
- il figlio ha ecceduto nell'uso di risorse
  - il compito assegnatogli non azzere più
  - il padre termina

NB Se il padre termina, il figlio:

- diventa orfano e viene adottato da `init()` (cioè `systemd()`)

• Il padre aspetta la terminazione del figlio invocando `wait()`;

`pid = wait(&status)`

↓ valore restituito dal figlio

- non può continuare poi avviene una terminazione a cascata

se non viene invocata il processo diventa Zombie, e se termina prima orfano

della wait

## Gerarchia dei processi Android:

A causa delle limitate risorse di calcolo, SO è costretto a terminare dei processi

|           |                     |                                        |
|-----------|---------------------|----------------------------------------|
|           | Processo foreground | : l'utente interagisce                 |
| VISIBLE   | ↳ visibile          | : attività di riferimento per l'utente |
|           | ↳ di servizio       | : in background ma visibile all'utente |
| INVISIBLE | ↳ in background     | : non visibile all'utente              |
|           | ↳ vuoto             | : componenti non attive                |

Prima di terminare salva sempre lo stato, per poterlo alla riapertura ripristinare

↓ inutile

## Comunicazione tra processi

Processo indipendente: fa la sua esecuzione non influenza su altri processi

Processi cooperanti: influenzano / sono influenzati da altri processi

- ↓
- ✓ Condivisione di informazioni
  - ✓ Accelerazione del calcolo
  - ✓ Modularità
  - ✓ Convenienza

## Meccanismi di comunicazione

• Scambio di messaggi: utile per piccole quantità di dati, nessun conflitto => migliore per i multiprocessori  
si usano system call (uso del kernel) e creano un canale (`send()`, `receive()`)

• Memoria condivisa: la più efficiente, il kernel interviene per l'allocazione della memoria  
gli accessi sono gestiti dai processi (non dal SO)  
\* coerenza della cache

- I processi devono annettere la zona di memoria al loro spazio di indirizzi e non devono sovrapporre sullo stesso spazio contemporaneamente

=> devono sincronizzarsi (Produttore-Consumatore)



Produttore-Consumatore

+ buffer illimitato: il consumatore può attendere  
il produttore può produrre

+ buffer limitato: il consumatore attende se il buffer è vuoto  
il produttore " " " " è pieno

Comunicazione diretta: i processi "nominano" esplicitamente i loro interlocutori

SIMMETRICO

send(P, messaggio): invia messaggio al processo P

receive(Q, messaggio): riceve un messaggio dal processo Q

ASIMMETRICO

send(P, messaggio)

receive(id, messaggio): in id c'è il nome del processo con cui è avvenuta la comunicazione

• I canali sono stabiliti automaticamente tra coppie ed è unico

- unidirezionale
- bidirezionale

Comunicazione indiretta: i messaggi vengono inviati/ricevuti alla mailbox o porte

• Un canale è stabilito solo se i processi hanno una mailbox comune

id unico

appartiene a

• PROCESSO: solo il proprietario può ricevere

al termine la mailbox scompare

• SO: ha vita autonoma

• Un canale può essere associato a più processi e a più coppie

- unidirezionale
- bidirezionale

### Sincronizzazione

Lo scambio di messaggi:

• BLOCCANTE: comunicazione SINCRONA

mittente e destinatario aspettano l'uno all'altro

• NON BLOCCANTE: comunicazione ASINCRONA

non c'è attesa, proseguono l'esecuzione

### Buffering (coda dei messaggi)

• Capacità 0: il canale non può avere messaggi in attesa (c'è bisogno di attesa) } buffering manuale

• Capacità limitata: n messaggi

• Capacità illimitata: infiniti messaggi

} automatiche

### Sistemi per IPC

• UNIX: IPC sia memoria condivisa, sia scambio di messaggi

• Mach: scambio di messaggi

• Windows: memoria condivisa

\* POSIX: memoria condivisa usa file mappati in memoria

↓ associano la regione condivisa ad un file

crea il

1. segmento di m.c. shm\_fd = shm\_open(name, O\_CREAT | O\_RDWR, 0666);

nome segmento

↓ creato

↓ apro in

↓ protezione

2. dimensione segmento ftruncate(shm\_fd, 4096);

3. scrivo al suo interno sprintf(pos, "writing");

↓ puntatore

↓ share memory

• mmap(): crea un file mappato in memoria

• shm\_unlink(): rimuove il segmento di memoria condivisa



**Pipe:** canale di comunicazione fra processi

Pipe convenzionali  
(anonime)

- possono essere create solo dal processo che le ha create
- unidirezionale
- esistenza limitata all'utilizzo
- usano produttore-consumatore

UNIX: ne creo una con **pipe(int fd[2])**

file speciale

vi accedo tramite descriptori fd  
(read(), write())

usata per padre-figlio

**Named Pipe**

- possono essere usate da tutti
- bidirezionale (o uni.)
- esistenza variabile
- possono usarla più processi

UNIX: sono normali file all'interno del file system  
(FIFO)

- half duplex
- devono esistere sulla stessa macchina

WINDOWS: hanno più funzionalità

... in pratica

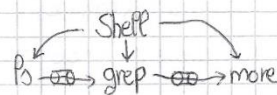
**\$ ls -lR | grep Jun | more**

↳ paginata

↳ stringa da contenere

↳ base dei contenuti

↳ pongi l'obiettivo



## Comunicazioni in sistemi client-server



## G. THREAD E CONCORRENZA

### Motivazioni e definizione di thread

La gestione dei processi può diventare molto onerosa

Con un'applicazione che genera molti processi, il SO passa la maggior parte del tempo a fare gestione, piuttosto che esecuzione

• **Creazione**: allocazione e popolazione dello spazio di indirizzi

• **Context-Switch**: saltare e ripristinare

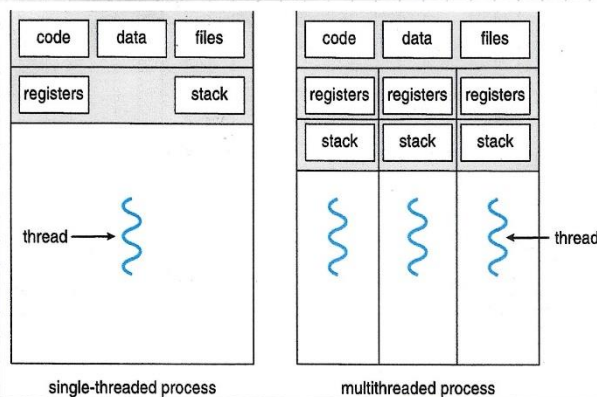
**Thread**: è l'unità base d'uso della CPU e comprende un identificatore di thread (TID)

un contatore di programma  
un insieme di registri  
uno stack

Sono detti "processi leggeri" perché hanno un contesto ridotto

Condividono lo spazio degli indirizzi

Un processo multithread esegue più compiti in modo concorrente



**kernel multithread**: ci sono thread dedicati a servizi specifici

• **Solazio**: gestione delle interruzioni

• **Linux**:  
- gestione di dispositivi  
- " di memoria  
- " delle interruzioni

ps -ef

### Vantaggi

• **Tempo di risposta**: ad una applicazione multithread è permesso continuare l'esecuzione anche se una parte è bloccata

• **Condivisione delle risorse**: i thread condividono la memoria e le risorse del processo. In un'applicazione una stessa applicazione ha thread relativi ad attività diverse, tutti nello stesso spazio di indirizzi

• **Economia**: Condividendo risorse, si risparmia rispetto ai processi ai quali va assegnata memoria

• **Scalabilità**: nelle architetture multi processore, i thread possono essere eseguiti in parallelo

# Programmazione multicore

Architettura multicore: diverse unità di calcolo sullo stesso chip

per il SO è un processore separato  $\Rightarrow$  i thread possono essere eseguiti in parallelo

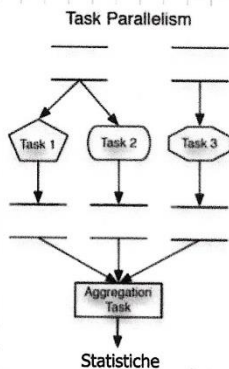
- NB - Un sistema è PARALLELO se può eseguire simultaneamente più task
- Un sistema CONCORRENTE supporta più task facendo multiplexing della CPU

## Tipi di parallelismo

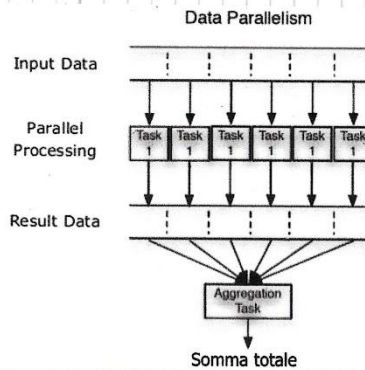
DATI: data parallelism: la stessa operazione è eseguita contemporaneamente sull'insieme dei dati  $\Rightarrow$  l'insieme è diviso in porzioni in modo che più thread possano agire simultaneamente

APPLICATIONI: task parallelism: codice parallelo tra più processori in ambienti di calcolo parallelo  $\Rightarrow$  distribuzione di thread diversa in esecuzione nei diversi nodi di calcolo

Calcolo di MAX e MIN



Calcolo della somma degli elementi di un vettore



- Obiettivi:
- Separazione dei task: trovare task della stessa applicazione che possano essere eseguiti in parallelo
  - Bilanciamento: task che abbiano compiti di mole confrontabile
  - Suddivisione dei dati: i dati devono essere suddivisi per essere utilizzati da unità di calcolo distinte
  - Dipendenze nei dati: verificare le dipendenze tra task per sincronizzare l'esecuzione
  - Test e debugging: garantire la correttezza nonostante i possibili problemi di concorrenza nell'accesso ai dati

hyperthreading: progetti hardware per implementare unità di calcolo multithread in cui due o più thread sono assegnati ad una singola CPU

In modo da ottimizzare lo stato della memoria

## Legge di Amdahl

$$Speedup \leq \frac{1}{S + \frac{1-S}{N}}$$

- S: porzione seriale
- N: core

$\downarrow$  determina quanto tutto

Permette di determinare i potenziali guadagni in termini di prestazioni ottenuti dall'aggiunta di core, sia componenti seriali sia parallele  
 $\downarrow$  non possono essere in parallelo

$$N \rightarrow \infty \quad Sp \rightarrow 1/S$$

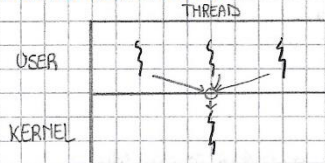


## Supporto del SO ai thread

- Thread a livello utente: tramite librerie di funzioni API ; Posix Pthreads (Unix)  
• Strato separato sopra il kernel ; Windows Thread
- Thread a livello kernel: tramite "un'astrazione di traccia di esecuzione"  
• Gestiti direttamente dal SO • Creazione • Sincronizzazione  
• Scheduling • Cancellazione

## Modelli di programmazione multithread

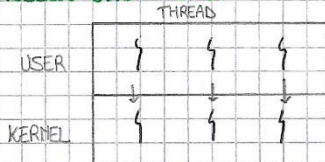
### MODELLO MULTI-A-UNO



- bisogna decidere quale è il thread più importante da eseguire
- i thread sono implementati a livello di applicazione (il loro scheduler non sta nel SO)
- ✓ Efficiente nella gestione su spazio utente
- ✓ Non richiede un kernel multithread

- ✗ l'intero processo rimane bloccato se un thread invoca una chiamata bloccante
- ✗ i thread sono legati allo stesso processo a livello kernel  $\Rightarrow$  non possono essere eseguiti su processori fisici distinti

### MODELLO UNO-A-UNO

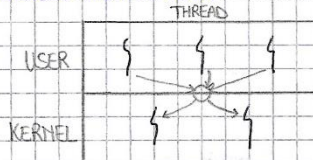


- Ad ogni thread-utente è associato un thread-kernel
- I thread sono gestiti dallo scheduler del kernel (nativi)

- ✓ Scheduling è efficiente
- ✓ Se un thread fa una chiamata bloccante, gli altri possono continuare
- ✓ i thread possono essere eseguiti su processori fisici distinti

- ✗ inefficienza per il carico di lavoro sul kernel
- ✗ il kernel deve essere multithread

### MODELLO MULTI-A-MOLTI con $m > n$



- Più thread (utente) sono associati ad un numero minore / uguale ai thread kernel

- m ✓ Creare tanti thread a livello kernel quanti sono necessari per la particolare applicazione, eseguibili in parallelo
- n ✓ Se un thread invoca una chiamata bloccante, il kernel fa eseguire un altro thread

- ✗ Difficoltà nel scegliere la dimensione di n e la modalità di cooperazione tra scheduling

### MODELLO A DUE LIVELLI (ibrido)

M: M, ma con possibilità U:U



**Librerie dei thread**: al programmatore è fornita una API per creazione / gestione dei thread

Libreria a livello utente:

- Codice e strutture dati nello spazio UTENTE
- Invoca una funzione a traduce in una CHIAMATA LOCALE a funzione

Libreria a livello kernel:

- Codice e strutture dati nello spazio KERNEL
- Invoca una funzione a traduce in una CHIAMATA DI SISTEMA (System call)

• Posix Pthreads: a livello utente che kernel

• Java: API gestibile, usa la Libreria dei thread del sistema che ospita la JVM

• Win32: a livello kernel per Windows

→ Tutte le variabili globali sono visibili a tutti i thread appartenenti allo stesso processo

qui si crea concorrenza

NB Le variabili locali sono poste nello stack (poi vanno perse)  
ogni thread ha il proprio stack

### Strategie di threading

Threading asincrono:

- il genitore crea un figlio <sup>(opio)</sup> => riprende l'esecuzione in concorrenza
- Ogni thread è eseguito in modo indipendente

→ Scarica condivisione dei dati es. server web

Threading sincrono:

- il genitore crea un figlio => attende che tutti terminino
- i thread figli si eseguono in concorrenza, una volta terminato, torna al padre

→ Significativa condivisione dei dati

**Pthreads**: è lo standard Posix che definisce la API per la creazione e la sincronizzazione dei thread

### Threading implicito

Le applicazioni multithread sono più difficili da programmare, ed aumentano in complessità all'aumentare del numero dei thread

Quindi la creazione / gestione del threading passa dagli sviluppatori ai compilatori e librerie di runtime

• **Gruppi di thread** (API per thread Windows)

Un numero illimitato di thread può eseguire le azioni del sistema

- create un numero di thread alla creazione del processo <sup>(worker)</sup>
- organizzati in gruppo in cui attendono il lavoro che verrà richiesto

✓ Se il thread già esiste => la risposta è più rapida (l'attesa viene eliminata)

✓ Limitiamo il numero di thread per processo

✓ Separa il task da eseguire dal meccanismo => due strategie diverse di creazione

• **OpenMP** è un insieme di direttive del compilatore ed una API per programmi scritti in C, C++, Fortran

• Supporto alla programmazione parallela

• Permette di impostare il numero di thread



# Threading

## Semantica di fork() ed exec()

In un programma multithread la semantica di fork() cambia:

- Se un thread invoca la fork(), il nuovo processo contiene:
  - Un duplicato di tutti i thread
  - Solo del thread invocante
- + Se exec() è chiamata immediatamente dopo la chiamata fork()  $\Rightarrow$  si duplica solo il thread chiamante

## Gestione dei segnali

In UNIX si usano segnali per comunicare ai processi il verificarsi di determinati eventi

- sono elaborati dal gestore dei segnali:
  - Predefinito (dal SO)
  - Definito dall'utente

- SEGNALI
- **SINCRONI**: se sono inviati dallo stesso processo che ha eseguito l'operazione causa del segnale
  - **ASINCRONI**: da eventi esterni al processo

Funzione per l'invio di un segnale: `kill(pid_t pid, int signal)`

- Nei sistemi multithread posso:
- Inviare il segnale
    - al thread cui il segnale è riferito
    - ad ogni thread del processo
    - a specifica thread del processo
  - definire un thread specifico per ricevere tutti i segnali diretti al processo

## Cancellazione di thread

- Il thread da cancellare è definito thread bezaglio

Modalità asincrona: terminazione immediata ~~X predefinita~~

Modalità differita: il thread bezaglio controlla periodicamente se deve terminare

Codice: `pthread_t tid;`

`pthread_create(&tid, 0, worker, NULL);` crea

`pthread_cancel(tid);` cancella, RICHIESTA

avviene solo se è attiva  
al punto di cancellazione

↓ clean up handler, rilascia le risorse

l'eliminazione dipende dallo  
stato del thread

su Linux si usano segnali

## Dati specifici del thread

Thread Local Storage (TLS) permette a ciascun thread di possedere una propria copia dei dati

↓  
sono visibili attraverso tutte le chiamate

## Attirazione dello scheduler

Il modello M:M richiede una comunicazione costante per garantire un numero sufficiente di thread del kernel allocati ad una applicazione

• Usa una struttura dati intermedia  
Light Weight Process

=>

per la libreria dei thread a livello utente, LWP è un processore virtuale

## Thread in LINUX

↓ TASK

• Creazione: **clone()** + insieme di flag per definire

↳ figli, per cui stabilisco risorse...  
attraverso puntatori

NB senza flag è una fork()

↓  
duplica

≠



## 5. SCHEDULING DELLA CPU

### Concetti fondamentali

Lo scheduling è una funzione fondamentale => **SCHEDULING DELLA CPU** è alla base dei sistemi operativi multi-programmati

Massimo impiego della CPU è la multi-programmazione

• So rende più produttivo il sistema di calcolo commutando il controllo della CPU tra processi

### Elaborazione di un processo

Si alternano cicli di esecuzione della CPU ed attese all'utilizzo di dispositivi I/O

↓ CPU-burst

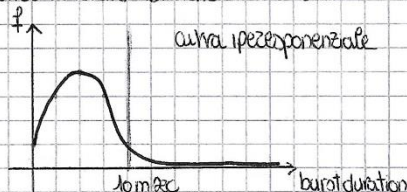
se prevalgono, un programma è definito CPU-bound (sequenze molto lunghe)

↓ I/O burst

se prevalgono, un programma è definito I/O bound (sequenze brevi)

↓ avere entrambi garantisce un uso intensivo e bilanciato di tutte le risorse

Grafico della distribuzione dei CPU-burst



• + scheduling dei processi → CPU

• + scheduling dei thread → multithreading

eseguito su una CPU = eseguito su un core

**Scheduler della CPU** gestisce la coda dei processi pronti, selezionando il prossimo processo cui verrà alllocata la CPU

READY QUEUE: gli elementi sono i PCB dei processi pronti

- FIFO
- coda con priorità
- lista concatenata
- alberi

Va presa una decisione quando un processo passa:

- 1 • Running → wait
- 2 • Running → ready
- 3 • wait → ready
- 4 • termina

+ 1 e 4 lo scheduling è nonpreemptive/cooperativo (il processo sceglie di lasciare)

+ 2 e 3 lo scheduling è preemptive (migrazione)

- accesso a dati condivisi
- prelazione di processi del kernel
- interruzione di operazioni cruciali

**Dispatcher** passa il controllo della CPU al processo selezionato dallo scheduler

- Context switch
- Passaggio a modo utente
- Salta alla posizione corretta

↓ frequenza del context switch

vmstat

P0 executing

[ save state in PCB  
restore state from PCB ]

P1 executing

dispatch latency

tempo per sospendere un processo e avviare un altro



## Criteri di scheduling

- Utilizzo della CPU: la CPU deve essere il più attiva possibile per ottimizzare  
MASSIMO
- Throughput: # processi completati per unità di tempo  
MASSIMO
- Tempo di turnaround: tempo impiegato per l'esecuzione di un processo  $\Rightarrow$  MINIMO
- Tempi di attesa: tempo passato dal processo in ready queue  
MINIMO
- Tempi di risposta: tempo tra la sottomissione di un processo e la prima risposta prodotta  
MINIMO

- Si ottimizzano:
  - i valori medi
  - i valori MAX/min
  - Varianza (per sistemi time sharing)

## Algoritmi di scheduling

### FIRST COME FIRST SERVED (FCFS)

- La CPU è assegnata al processo che la richiede per primo + ready queue è una coda FIFO

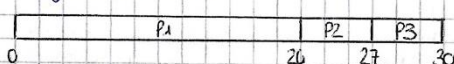
Quando un processo entra nella ready queue, si collega il suo PCB all'ultimo elemento della coda.

- ✗ Non bilanciato
- ✗ Lento

Esempio

Tempo burst  
 $P_1: 24$  (msec)  
 $P_2: 3$   
 $P_3: 3$

Diagramma di Gantt



Tempo di attesa  $P_1 \rightarrow 0, P_2 \rightarrow 24, P_3 \rightarrow 27$

Tempo medio di attesa  $T_a = \frac{0 + 24 + 27}{3} = 17 \text{ msec}$

effetto coda: i processi di breve durata potrebbero aspettare molto  
 $\rightarrow$  dipende dall'ordine di arrivo

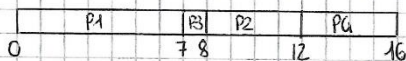
### SHORTEST JOB FIRST (SJF)

- La CPU è assegnata dal processo con burst minore fino al maggiore
- + non preemitive
- + preemitive: se arriva un processo con burst più corto, si cambia processo in esecuzione

✓ ottimo

Esempio (non-preemitive)

| Processo | Tempo arrivo | Tempo burst |
|----------|--------------|-------------|
| $P_1$    | 0            | 7           |
| $P_2$    | 2            | 6           |
| $P_3$    | 4            | 1           |
| $P_4$    | 5            | 6           |



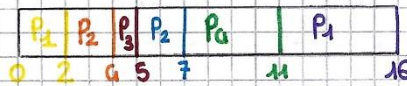
$P_1 \rightarrow 0, P_3 \rightarrow 7-6=1, P_2 \rightarrow 8-2=6, P_4 \rightarrow 12-5=7$

$T_a = \frac{0 + 1 + 6 + 7}{4} = 6 \text{ msec}$



esempio: **SRTF** (Shortest Remaining Time First)

| Processo       | T arrivo | T burst |
|----------------|----------|---------|
| P <sub>1</sub> | 0        | 7       |
| P <sub>2</sub> | 2        | 4       |
| P <sub>3</sub> | 4        | 1       |
| P <sub>4</sub> | 5        | 4       |



$$P_1 \rightarrow 11 - 2 = 9 \quad P_2 \rightarrow 5 - 2 - 2 = 1 \quad P_3 \rightarrow 4 - 4 = 0 \quad P_4 \rightarrow 7 - 5 = 2$$

$$T_a = \frac{9 + 1 + 0 + 2}{4} = 3 \text{ msec}$$

**Lunghezza del successivo CPU-burst** è una media esponenziale delle lunghezze dei burst di CPU precedenti (ciascun termine ha minor peso dell'uso precedente)

$$T_{n+1} = d T_n + (1-d) T_n$$

↓  
valore stimato  
del prossimo burst

↓  
lunghezza  
n-esimo CPU-burst

•  $d: 0 \leq d \leq 1$  (combinazione convessa)

$d = 1/2$  di solito

•  $d = 0 \Rightarrow T_{n+1} = T_n$  la storia recente non viene presa in considerazione

•  $d = 1 \Rightarrow T_{n+1} = T_n$  considero solo l'ultimo CPU-burst

### SCHEDULING A PRIORITÀ:

• Ad ogni processo è associato un valore di priorità

• La CPU è assegnata al processo con priorità più alta

- non-preemptive
- preemptive

✗ Stazionario: i processi a bassa priorità potrebbero non venirci mai eseguiti  $\Rightarrow$  ✓ Aging: aumento gradualmente la priorità dei processi in attesa

### ROUND ROBIN (RR)

• Ad ogni processo è allottato un tempo di CPU (time slice), dopo quello il processo deve lasciare la CPU

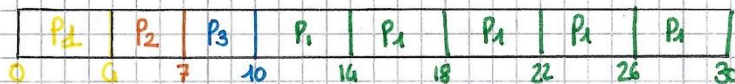
• Il timer interrompe la CPU

•  $q$  grande  $\Rightarrow$  FCFS

•  $q$  piccolo  $\Rightarrow$  troppi context switch

esempio

| Processo       | T burst | $q = 4$ |
|----------------|---------|---------|
| P <sub>1</sub> | 26      | 20      |
| P <sub>2</sub> | 3       | ✓       |
| P <sub>3</sub> | 3       | ✓       |



$$P_1 \rightarrow 10 - 4 = 6 \quad P_2 \rightarrow 4 \quad P_3 \rightarrow 7$$

$$T_a = \frac{6 + 4 + 7}{3} = 5.6 \text{ msec}$$



## Scheduling con code multiple

• Finora i processi erano collocati in singola coda  $\Rightarrow$  la ricerca costa  $O(n)$

conviene usare code distinte: lo scheduler assegna la CPU al processo "intesta" alla coda a priorità più alta

La ready queue è divisa in:

- foreground (interattiva) ①
- background (batch) ②

Ciascuno con un algoritmo di scheduling

- ①  $\Rightarrow$  RR
- ②  $\Rightarrow$  FCFS

effettua lo scheduling tra le code:

- + Scheduling a priorità fissa: prima foreground poi background  $\Rightarrow$  starvation

- + Time slice: do da temp. 80% ai foreground 20% ai background

I processi si assegnano in modo permanente ad una coda  $\Rightarrow$  STATICHE

Code multiple con feedback: un processo può spostarsi fra varie code

Scheduler:

- # di code
- Algoritmo scheduling per coda
- Metodo per spostare un processo da priorità maggiore a minore
- Metodo per innalzare un processo alla coda minore a maggiore

separa i processi in base alle loro caratteristiche d'uso della CPU

$\hookrightarrow$  Minima priorità con CPU-burst brevi

## Scheduling dei thread

Scheduling dei thread a

- $\nearrow$  livello utente
- $\searrow$  livello kernel

ma se supporti thread allora avviene a livello thread

Scheduling locale: modelli M:1 e M:H, è l'algoritmo con cui la libreria dei thread decide quale thread allocare a un LWP (competizione ristretta al processo PCS)

Algoritmo a priorità con prelazione

NB. Il thread non va in esecuzione finché non pianifica la sua esecuzione a livello kernel

Scheduling globale: è l'algoritmo con cui il kernel decide quale thread allocare a livello kernel per esecuzione (competizione allargata all'intero SCS)

— LINUX: API Pthreads di POSIX usa PCS o SCS (M:H)

PTHREAD\_SCOPE\_PROCESS  
pianifica thread a livello utente su LWP

PTHREAD\_SCOPE\_SYSTEM  
crea sup thread utente un LWP vincolato  $\Rightarrow$  M:1



## Scheduling multiprocessore

$P_1, P_2, \dots$

Con più CPU, il problema diventa più complesso.

Architetture multi processore sono:

- CPU multicore
- Multicore multithread
- Sistemi NUMA
- Sistemi eterogenei

Le unità di elaborazione sono IDENTICHE  
↓ (omogenea)

Ripartizione del carico di lavoro

Processori con diverse caratteristiche HW

Multielaborazione asimmetrica: tutto è affidato ad un solo processore (master, server)

- scheduling
- elaborazione I/O

✓ Riduce la necessità di condividere dati

X È un possibile collo di bottiglia

Multielaborazione simmetrica (SMP): i thread pronti formano una coda oppure ogni processore ha una propria coda e ciascun processore ha uno scheduler

X evita di scegliere contemporaneamente lo stesso thread

X race condition ⇒ l'accesso alla coda deve essere gestito tramite lock

X qualche thread pronto varia poco

X Sbilanciamento del carico di lavoro ⇒ uso cache

## Processore multicore

$P_1, P_2, \dots$

- fanno uso di SMP: ✓ evita ✓ consumano meno energia

PROBLEMA: Stallo della memoria

Quando un processore accede alla memoria, il tempo trascorre in attesa della disponibilità dei dati

↓

viene risolto con progetti HW (hyperthreading): per il SO, ogni thread hardware appare come un processore logico in grado di eseguire thread software

- Chip multithreading ⇒ a ciascuno sono assegnati thread hardware

multithreading

- coarse-grained: thread resta in esecuzione su un processore fino a quando non ha evento a lunga latenza
- fine-grained: passa da un thread ad un altro al termine di un ciclo di istruzione
  - una istruzione per ogni thread
  - rapidità di context-switch

• ripulisce la pipeline

- Un processore multicore e multithread richiede due livelli di scheduling:

- + SO stabilisce quale thread software mandare in esecuzione su ciascun thread hardware
- + ciascun core sceglie quale thread hardware mandare in esecuzione



Sulla SMP...

+ Bilanciamento del carico: ripartire uniformemente il carico di lavoro su diverse processori

• Se la ready queue è comune  $\Rightarrow$  AUTOMATICO

• Migrazione guidata: **thread** dedicato a controllare i carichi e riequilibrarli

• Migrazione Spontanea: **processore** inattivo sottrae carico a quelli in attesa

+ Predizione per il processore: è utile mantenere un thread sullo stesso processore, per utilizzare la cache

+ debole: può avvenire la migrazione

+ forte: è bloccato

Queste due filosofie sono antitetiche.

• NUMA: l'architettura della memoria influenza la predizione

$\downarrow$  dove si accede alla memoria di un'altra CPU costa tempo

Multi processing eterogenei: riguarda i dispositivi mobili

• Tutte le attività possono essere eseguite su qualsiasi core

• Ciascuno differisce in velocità del clock ed in gestione energetica

Architettura big LITTLE

$\swarrow$  grandi core a prestazioni elevate

$\searrow$  piccoli core ad efficienza energetica

## Scheduling real-time

Studieremo i sistemi **HARD real-time**: richiedono il completamento dei **processi critici** entro un **intervallo di tempo** predeterminato e garantito

+ Sono guidati dagli eventi

**Latenza dell'evento**: tempo che intercorre tra l'occorrenza dell'evento e il momento in cui il sistema effettua la gestione

+ Richiedono un tempo di risposta prevedibile agli eventi esterni

1. Un dispositivo genera un interrupt
2. La routine di servizio  $\rightarrow$  raccoglie i dati
3. Codice a livello utente elabora i dati raccolti

**Reattività**: velocità con cui SO e codice elaborano l'evento

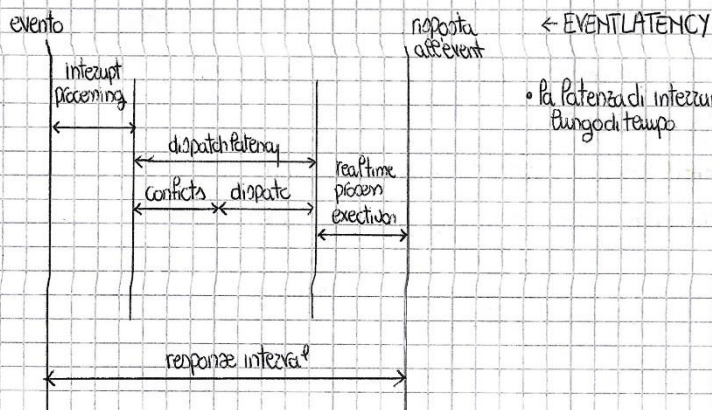
**Latenza**  $\rightarrow$  **relativa alle interruzioni**: tempo tra la notifica di interruzione alla CPU e avvio della routine  
• minimizzare il tempo in cui le interruzioni sono disattivate

$\rightarrow$  **di dispatch**: tempo del dispatcher per interrompere un processo e avviare il codice di gestione

$\downarrow$  **fase di conflitto**: 1. Prefazione di ogni processo in esecuzione sul kernel

2. Confronto dei processi a minore priorità, delle loro richieste dai processi ad alta priorità





- La latenza di interrupt è determinata dall'intervallo più lungo di tempo

Lo scheduler deve implementare un algoritmo con priorità basata su priorità, e il hard real time deve anche rispettare le scadenze

Consideriamo processi periodici (richiedono la CPU a intervalli costanti di tempo)

- tempo di elaborazione  $t$
- scadenza  $d$
- periodo  $p$
- $0 \leq t \leq d \leq p$
- la frequenza è  $1/p$

### Scheduling con priorità proporzionale (Rate Monotonic) alla frequenza

- La priorità è inversamente proporzionale al periodo, si assegnano priorità più elevate ai processi che fanno uso più frequente della CPU
- È un algoritmo ottimale, se non è in grado di pianificare l'esecuzione nessun altro algoritmo che assegna priorità statiche alla task.

In caso di task periodici, se ho  $m$  task, il caso può essere gestito solo se: (necessaria, ma non sufficiente)

$$\sum_{i=1}^m \frac{C_i}{P_i} \leq 1$$

- $P_i$ : periodo
- $C_i$ : secondi di tempo di CPU

SISTEMA  
→ SCHEDULABILE

⇒ In realtà lo scheduling è garantito per  $\sum_{i=1}^m \frac{C_i}{P_i} \leq m(2^{1/m} - 1)$

- $m=2$  il caso non deve superare l'83%
- $m \rightarrow \infty$  " " 69,3%

### Scheduling EDF (Earliest-Deadline-First) ⇒ ottimale (permette di usare CPU al 100%)

- Assegna dinamicamente la seconda delle scadenze

Quando un processo diventa eseguibile:

1. Annuncia la sua scadenza allo scheduler
2. Si calcola la priorità dinamicamente
3. Più è vicina la scadenza più alta la priorità

+ Anche per processi non periodici e con tempo di elaborazione variabile



## Scheduling a quote proporzionali

- Distribuisce un numero di quote  $T$  fra tutte le applicazioni del sistema
- Lo scheduler lavora in sinergia con un meccanismo di controllo dell'ammissione, così ogni applicazione avrà la giusta quota
- ↳ ciascuna fino a  $M$  quote, in modo da usare  $M \cdot T$  del tempo totale di CPU

## Virtualizzazione e scheduling

SO ospitante crea e gestisce le VM, ciascuna con proprio SO ospite

- Per ciascuna VM, algoritmi di scheduling ci saranno da problemi
- Tempi di risposta dilatati
- Orologio di sistema delle VM può essere rallentato

X La virtualizzazione può vanificare i benefici di un buon algoritmo di scheduling

## Scheduling in LINUX

+ Prima della versione 2.5 => Code multiple con feedback gestite con RR

X Poco scalabile

- Prefazione sulle code con priorità attribuite internamente (più grande, minore e la priorità)

X Non adeguato ai sistemi SMP

- Il kernel controlla quanta CPU un processo consuma periodicamente e inversamente proporzionale alla priorità del processo fino al prossimo controllo

+ Dopo algoritmo  $O(1)$  indipendente dal numero dei task

- a priorità preemitive

- Per i task time-sharing, ha due livelli di priorità

↳ nice 100-139

↳ soft (realtime) 0-99

- Task a priorità maggiore ottengono time-slice più lunghi

- Esaurito il tempo, il task è scaduto e non verrà rieseguito fino a che tutti gli altri task non siano esauriti

✓ Sistemi SMP (bilanciamento e predizione)

X tempi di risposta elevati per task a time-sharing

+ Versione 2.6.23: Completely Fair Scheduler (CFS)

Classi di scheduling: default e realtime

La % di tempo di CPU è calcolata da nice

- Lo scheduler seleziona il processo a priorità più alta e lo mette nella classe a priorità maggiore

Latency obiettivo: intervallo di tempo entro il quale ogni task eseguita dovrebbe ottenere la CPU almeno una volta

Lo scheduler non assegna direttamente le priorità ma registra per quanto tempo è stato eseguito ogni task, nella variabile vruntime segna il tempo di esecuzione virtuale

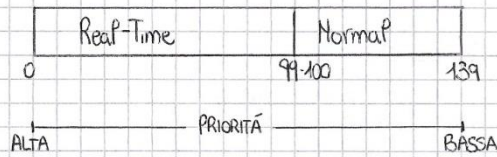
↳ è associato ad un fattore di decadimento (dipende dalla priorità del task)

↳ il prossimo task da eseguire ha il più piccolo valore

NB Per i task con priorità normale (nice=0) il t. di exec. virtuale è uguale al tempo effettivo di esecuzione.

Ho un albero binario di ricerca basato sui valori di vruntime





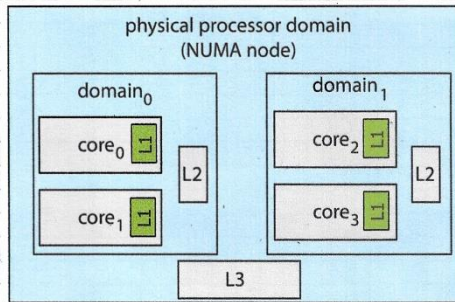
• Le priorità sono ammantate, ma non i tempi di attesa nella coda dei processamenti

- Supporta il bilanciamento del carico
- Compatibile con NUMA e riduce al minimo la migrazione dei thread

• Carico di un thread:

- Priorità
- Tempo medio di utilizzo della CPU

• Sistema gerarchico di domini scheduling, divisi su ciò che condividono



**Valutazione degli algoritmi**: come scegliere il miglior algoritmo di scheduling?

+ Valutazione analitica: in base ad algoritmo e carichi di lavoro, si fornisce una formula per valutare le prestazioni

+ Modelli deterministici: ↓ ma con carico predeterminato

+ Reti di code: il sistema di calcolo è una "rete a zvez", ciascuno con una coda d'attesa

• CPU è un server con la propria coda, così come il sistema di I/O

• È noto l'andamento degli arrivi e dei servizi

⇒ Si riferano sui dati di distribuzione

MODELLI MATEMATICI NON REALISTICI

Formula di Little

$$n = \lambda \times W$$

lunghezza media di una coda

andamento medio di arrivo

tempo medio di attesa

se vale il sistema è STABILE

+ Simulazione: si programma un modello del sistema di calcolo

• Si raccolgono dati durante l'attività

X Completo

X Costoso

+ Implementazione: si codifica l'algoritmo e lo si testa

X Fa accettare agli utenti un sistema in continua evoluzione

X Alti costi e rischi

⇒ I migliori sono quelli adattabili, tarabili dagli amministratori di sistema



## 6. STRUMENTI DI SINCRONIZZAZIONE

### Introduzione

Ora i task vengono eseguiti concorrentemente su core...

↓ può influenzare gli altri processi in esecuzione • Condivisione di codice e/o dati

L'accesso concorrente ai dati condivisi può causare inconsistenza nei dati

⇒ bisogna garantire la coerenza dei dati

↓ ogni processo va eseguito in modo ordinato, ricordando che lo scheduler può interromperli in qualsiasi momento

### Produttore-Consumatore

• Per risolvere il problema aggiungo una variabile **contatore**:  
(aumenta una posizione) - Viene incrementato ogni volta che un nuovo elemento è inserito nel buffer e viceversa

```
#define BUFFER_SIZE 10
typedef struct {
    ...
    elemento;
    elemento buffer[BUFFER_SIZE];
    int in = 0;
    int out = 0;
    int contatore = 0;
}
```

#### PRODUTTORE

```
elemento appena_prodotto;
while(1) {
    while (contatore == BUFFER_SIZE)
        ; /* do nothing */
    buffer[in] = appena_prodotto;
    in = (in + 1) % BUFFER_SIZE;
    contatore++;
}
```

#### CONSUMATORE

```
elemento da_consumare;
while(1) {
    while (contatore == 0)
        ; /* do nothing */
    da_consumare = buffer[out];
    out = (out + 1) % BUFFER_SIZE;
    contatore--;
}
```

out: indice della 1° posizione piena

in: indice della successiva posizione libera

Se i due moduli sono eseguiti in concorrenza  
possono avere problemi: contatore++  
contatore--

devono essere eseguite atomicamente  
↓ eseguita senza subire interruzioni

R: registro

$R_1 = \text{contatore}$   
 $R_1 = R_1 + 1$   
↓  
 $\text{contatore} = R_1$

$R_2 = \text{contatore}$   
 $R_2 = R_2 - 1$   
↓  
 $\text{contatore} = R_2$

• i registri possono coincidere

• se non sono eseguiti in ordine ho una interferenza (interferenza)

**Race condition**: più processi accedono in concorrenza e modificano dati condivisi  
↓ soluzione

### Sincronizzazione

• esempio: i processi P0 e P1 creano processi figli (fork()) ⇒ ho race condition sul PID

! Senza mutua esclusione, due processi figli possono avere lo stesso PID!



## Il problema della sezione critica

Siano  $n$  processi  $P_0, \dots, P_{n-1}$  competono per usare dati condivisi

↓  
Ciascun processo è costituito da un segmento di codice, sezione critica, in cui accede ai dati condivisi.

PROBLEMA: quando un processo accede alla propria sezione critica, a nessun altro processo è concesso di fare l'esecuzione di un'azione analoga

⇓ MUTUAMENTE ESCLUSIVA

SOLUZIONE progettare un protocollo di cooperazione fra processi

```
do {  
  ENTRY SECTION  
  sezione critica  
  EXIT SECTION  
  sezione non critica  
} while(1);
```

• Ogni processo chiede il permesso di accesso alla sezione critica.

+ **Mutua esclusione**: se  $P_i$  è in esecuzione nella sua regione critica, nessun altro processo può eseguire la sua sezione critica.

+ **Progresso**: se nessun processo esegue la sezione critica, e ce ne sono alcuni in attesa, questa attesa non può essere rimandata indefinitamente

+ **Attesa limitata**: se un processo ha effettuato la richiesta, bisogna porre un limite al numero di richieste dallo stesso  $\Rightarrow$  per evitare starvation

Ciascun processo a velocità  $\neq 0$

Il codice kernel deve regolare gli accessi ai dati condivisi

kernel

- con di più di **prelazione** (usabilità): consente la prelazione di processi in esecuzione in modalità kernel  
 $\checkmark$  real-time e time-sharing
- senza di più di **prelazione**: i processi sono eseguiti fino a quando non escono dalla modalità kernel

\* critica per SMP, due P kernel possono essere eseguiti su processori distinti

## La soluzione di Peterson (1981)

Per due processi  $P_i$  e  $P_j$ , condividono: int turno;  
boolean flag[2];  $\sim$  Processi turno per accedere alla sezione critica  
 $\sim$  indica se un processo è pronto ad entrare nella propria sezione critica

```
Pi:  
do {  
  flag[i] = true;  
  turn = i;  
  while (flag[j] && turn == j) ;  
  SEZIONE CRITICA  
  flag[i] = false;  
  SEZIONE NON CRITICA  
} while  
Pj è speculare
```

$\checkmark$  **Mutua esclusione**, grazie a turn

$\checkmark$   $P_i$  entra dopo al massimo un ingresso di  $P_j$   $\Rightarrow$  **attesa limitata**

è basata sul software, non richiede supporto speciale da SO

$\times$  sulle architetture attuali non funziona

\* fin quando la condizione è vera,  $P_i$  non può accedere alla sezione critica

Per migliorare le prestazioni, i processi riordinano le operazioni indipendenti  $\Rightarrow$  con Peterson non posso farlo

↓  
nei multithread può causare risultati inconsistenti

↓  
consente l'accesso concorrente



## Supporto hardware alla sincronizzazione

Viene fornito un supporto hardware per risolvere il problema della sezione critica.

- Per i monoprocessori basta interdire le interruzioni mentre si modifica le variabili condivise. • no prelazione

### • Bazziera di memoria

### • Istruzione hardware

### • Variabili atomiche

In generale, ogni istruzione richiede l'utilizzo del lock (HW o SW). Per accedere alla propria sezione critica, il processo acquisisce un lock e poi lo restituisce all'uscita.

```
do {  
    acquisisce il lock  
    sezione critica  
    restituisce il lock  
    sezione non critica  
} while (1);
```

### • Bazziera di memoria

Il modello di memoria può essere:

- Fortemente ordinato: la modifica alla memoria è immediatamente visibile a tutti gli altri

- Debolmente ordinato:

non

Una bazziera di memoria è un'istruzione che forza la propagazione a tutti i processori di ogni cambiamento alla memoria.

meccanismi di basso livello per garantire la mutua esclusione

```
while (!flag)  
    memory_bazzer();  
print x;
```

```
x = 100;  
memory_bazzer();  
flag = TRUE;
```

Vengono eseguiti in modo ordinato

### • Istruzione hardware

Si forniscono speciali istruzioni atomiche, eseguibili come unità non prelazionabili e sequenziali (su multiproc.).

- Test And Set: controllano e modificano il contenuto di una parola di memoria.

- Compare And Swap: scambiano il contenuto di due parole di memoria.

```
boolean lock = FALSE;
```

```
boolean TestAndSet (boolean *object) {  
    boolean value = *object;  
    *object = TRUE;  
    return value;  
}
```

```
do {  
    while (TestAndSet (&lock));  
    sezione critica  
    lock = FALSE;  
    sezione non critica  
} while (1);
```

P<sub>i</sub>:

• Mutua esclusione ✓

- eseguita atomicamente ✓
- ritorna il valore originale del parametro passato
- pone il nuovo valore del parametro TRUE

✗ non garantisce l'alternanza → starvation

### • Variabili atomiche

Le istruzioni hardware per garantire la mutua esclusione sono utilizzate come blocchi costruttivi per altri strumenti di sincronizzazione => variabili atomiche (interi o booleani).

- aggiornate senza soluzione di continuità

- Sono usate come contatori e generatori random



## Lock mutex

↓  
mutual exclusion

Le soluzioni hardware al problema della sezione critica sono complicate ⇒ LOCKMUTEX  
e generalmente inaccessibili ai programmatori di applicazioni

- Si proteggono le sezioni critiche prima acquistando e quindi rilasciando il lock (acquire()) (release()) devono essere atomiche
- Definisco una variabile booleana available che indica se il lock è disponibile o meno

```
acquire() {  
    while (!available)  
        ; /* busy wait */  
    available = false;  
}  
  
release() {  
    available = true;  
}  
  
do {  
    acquire();  
    azione critica  
    release();  
    azione non critica  
} while (true);
```

Il 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().  
⇒ spinlock

Semafiori: sono i più sofisticati usati nei multi processori

Il semaforo S è una variabile intera, e posso accedervi solo tramite due operazioni indivisibili

```
wait(S) {  
    while (S <= 0)  
        ; // busy waiting  
    S--;  
}  
  
signal(S) {  
    S++;  
}
```

modificano il valore di S  
è importante che siano  
eseguite in ordine

• wait() (P)  
• signal() (V)

• Mentre un processo cambia il valore del semaforo, nessun altro può farlo

• Variabile condivisa: semaphore mutex = 1;  
// inizialmente mutex = 1

• Processo: 

```
do {  
    wait(mutex);  
    azione critica  
    signal(mutex);  
    azione non critica  
} while (1);
```

• Semaforo contatore: intero che può assumere valori in un dominio non limitato

• Semaforo binario: assume solo 0 o 1

esempio. P1 e P2 sono due processi con i frammenti di codice C1 e C2  
e devono essere eseguiti in sequenza  
uso un semaforo "synch"

```
P1:  
C1;  
signal(synch);  
P2:  
wait(synch);  
C2;
```

ESEMPIO: descrivi il comportamento

Semaphore S1=0, S2=0;

```
P1 { signal(S1);  
    wait(S2);  
    print "P"  
    wait(S2);  
    print "R"  
    signal(S1);  
}  
  
P2 { signal(S2);  
    wait(S1);  
    print "A"  
    signal(S2);  
    wait(S1);  
    print "I";  
}
```

→ "P" "A" oppure "A" "P"

→ "R" "I"



## Implementazione dei semafori

!!! Bisogna garantire l'atomicità di wait() e signal(): devono essere eseguite all'interno di una sezione critica

- Codice di implementazione ridotto (4 istruzioni) ↓ busy waiting possibile
- Brevi attese se la sezione critica è occupata

• Se uso semaforo spinlock si crea busy waiting (attesa attiva) dei processi in attesa ad un semaforo

↓  
X spreco di CPU da parte dei processi in attesa

Busy waiting: È vantaggioso solo se molto breve, perché evita i context-switch

- Un lock è conteso se un thread si blocca mentre tenta di acquistarlo ⇒ riduzione delle prestazioni  
↳ elevata o moderata
- Usare uno spinlock quando il lock deve essere trattenuto per una durata inferiore a due cambi di contesto

### Come evitare il busy waiting?

- Ad ogni semaforo associare una coda di processi in attesa

↳ contiene:  
- Valore intero (# processi in attesa)  
- puntatore all'indice della lista dei processi

- Due operazioni: block: posiziona il processo che richiede di essere bloccato nell'appropriata coda di attesa  
(System call)

wakeup(P): rimuovere il processo P dalla coda di attesa e lo mette in ready queue

```
wait (semaphore *S) {  
    S->valore--;  
    if (S->valore <= 0) {  
        aggiungi il processo P a S->lista;  
        block();  
    }  
}
```

↳ ha sbarrato a scivolare

```
signal (semaphore *S) {  
    S->valore++;  
    if (S->valore <= 0) {  
        rimuovi il processo P da S->lista;  
        wakeup(P);  
    }  
}
```

Il seguente semaforo può assumere valori negativi

- Se valore è negativo, |valore| è il numero di processi che attendono al semaforo

### Errori comuni

- signal(mutex) ... wait(mutex): inversione delle chiamate ⇒ più processi possono entrare nella sezione critica contemporaneamente
- wait(mutex) ... wait(mutex): doppia chiamata alla stessa System call ⇒ si genera deadlock
- Omissione di wait(mutex) o signal(mutex) o entrambe ⇒ violazione mutua esclusione o stallo

## Il monitor (C#, Java, Pascal)

- Semplificano la gestione della mutua esclusione (↑ meno errori), ho sempre memoria condivisa
- Sono costruiti, non librerie ⇒ ADT (Abstract Data Type) di alto livello

Incapsula dati mettendola a disposizione metodi per operare su di essi

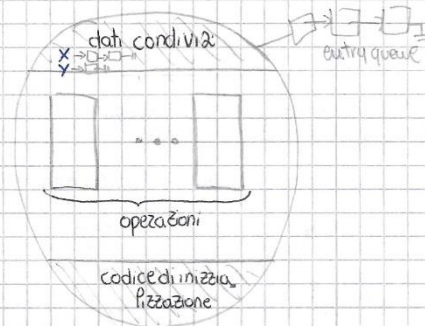
- dichiarazioni di variabili i cui valori definiscono lo stato di un'istanza del tipo
- procedure o funzioni che realizzano operazioni su queste variabili



```

monitor monitor-name {
  datatype // dichiarazione variabili condivise
  function P1(...) {
    ...
  }
  // codice di inizializzazione
}

```



• All'interno di monitor può essere attivo un solo processo  
 => non va codificato il vincolo di mutual exclusion

• Per allocare le risorse tramite monitor, P deve richiamare una routine interna

• In realtà, per permettere ad un processo di attendere dopo l'ingresso al monitor (risorsa richiesta occupata) si dichiarano variabili condition

condition x, y;

si utilizzano in relazione a wait e signal



• X.wait(); P chiama e rimane sospeso finché quando un altro P chiama signal()

• X.signal(); attiva un P sospeso  
 \* Se non ce ne sono, non fa nulla

• Usa una variabile condition per attendere il verificarsi di una condizione, uscendo dal monitor nell'attesa

• Un monitor associa a ogni variabile condition ad una situazione che può determinare l'attesa di un processo

• Ogni variabile condition ha una coda associata

+ Se P chiama wait() su v. condition: è posizionato nella sua coda (fuori dal monitor) così che un altro processo possa entrare e fare signal

+ Se P chiama signal() su v. condition: un P in attesa su quella v. è rimosso dalla coda associata e può accedere al monitor



+ Segnalare ed attendere entra Q e P attende

+ Segnalare e proseguire attende Q => ma al termine di P, la condizione attesa da Q può non essere più valida

• FIFO oppure della coda dei processi  
 • priorità

=> P esegue signal come sua ultima istruzione

• Non garantisce la sequenzialità dell'acquisizione della risorsa, potremmo includere la risorsa nel monitor ma così lo scheduling è quello di accesso al monitor e non di SO



**Liveness:** i processi potrebbero dover attendere indefinitamente, nel tentativo di acquisire uno strumento di sincronizzazione  $\Rightarrow$  viola i requisiti di progresso ed attesa limitata  
 $\downarrow$  tutelati dalla liveness

**Deadlock:** due o più processi attendono indefinitamente un evento che può essere causato solo da uno dei processi in attesa

**Starvation:** un processo può attendere per un tempo infinito nella coda di un semaforo senza mai venir chiamato

### Inversione di priorità

QUANDO: un processo a priorità più alta ha bisogno di leggere/modificare dati a livello kernel usati da processi a priorità più bassa.

PERCHÉ: poiché i dati a livello kernel sono protetti da lock, la priorità maggiore dovrà attendere

Siano 3 task  $T_1, T_2, T_3$  con priorità  $p_1 > p_2 > p_3$ ;  $T_1$  e  $T_3$  devono accedere alla risorsa con mutex  $S$

$\downarrow$   $T_3$  inizia ed esegue `wait(S)`  
 $T_1$  esegue `wait(S)` ma viene bloccato per tempo infinito (fino a  $T_3$ : `signal(S)`)  
ed anche  $T_2$  attende

SOLUZIONE: implementare un protocollo di ereditarietà delle priorità

I processi che accedono ad una risorsa di cui hanno bisogno processa priorità maggiore ereditano la priorità più alta finché non ritolcano l'uso della risorsa

Quando hanno terminato, la loro priorità ritorna al valore originale

### Valutazione delle soluzioni

• **Interi atomici:** leggeri ed adatti ad aggiornamenti di singole variabili condivise (contatori)

• **Lock/mutex:** semplici e meno overhead  
proteggono dall'accesso alla sezione critica  $\rightarrow$  meglio il semaforo che risorse limitate



## 7. ESEMPI DI SINCRONIZZAZIONE

### Problemi classici di sincronizzazione

#### Problema del produttore-consumatore con buffer limitato

Variabili condivise: `int n;`  
`semaphore full, empty, mutex;` // inizialmente `full=0`  
`empty=n`  
`mutex=1`

- buffer ha `n` posizioni, ciascuna contiene un elemento
- `mutex` garantisce la mutua esclusione sugli accessi al buffer
- `empty` e `full` contano # posizioni vuote e piene nel buffer

```
PRODUTTORE
while (true) {
    /* produce un elemento in next_produced */
    ...
    wait(empty);
    wait(mutex);
    ...
    /* inserisce next_produced nel buffer */
    ...
    signal(mutex);
    signal(full);
}
```

```
CONSUMATORE
while (true) {
    wait(full);
    wait(mutex);
    ...
    /* sposta un elemento dal buffer in next_consumed */
    ...
    signal(mutex);
    signal(empty);
    ...
    /* consuma un elemento in next_consumed */
    ...
}
```

NB I processi sono simmetrici

#### Problema del lettore-scrittore

Un insieme di dati deve essere condiviso  
 tra più processi concorrenti:

- Lettore
- Lettore/Scrittore

⇒

PROBLEMA • Permettere a più lettori di accedere ai dati contemporaneamente  
 • Solo uno scrittore può accedere ai dati condivisi

Variabili condivise: `Semaphore mutex, rw_mutex;`  
`int read-count;` // inizialmente `mutex=1`  
`rw_mutex=1`  
`read-count=0`

```
SCRITTORE
while (true) {
    wait(rw_mutex);
    ...
    /* scrittura */
    ...
    signal(rw_mutex);
}
```

```
LETORE
while (true) {
    * wait(mutex);
    * read_count++;
    if (read_count == 1)
        wait(rw_mutex);
    * signal(mutex);
    ...
    /* lettura */
    ...
    * wait(mutex);
    * read_count--;
    if (read_count == 0)
        signal(rw_mutex);
    * signal(mutex);
}
```

- `rw_mutex` è in comune, mutua esclusione per gli scrittori, ed al primo ed ultimo lettore serve per entrare/usare dalla sezione critica
- `mutex` mutua esclusione all'atto dell'aggiornamento di `read-count`
- `read-count` # processi che stanno leggendo l'insieme dei dati



Alcune varianti: • Il problema dei lettori-scrittori: nessun lettore attende, a meno che uno scrittore non abbia già ottenuto accesso ai dati

• Scrittori → starvation

• Il problema dei lettori-scrittori: se uno scrittore è pronto deve eseguire il suo compito al più presto, quindi nessun lettore può iniziare la lettura

• Lettori → starvation

Lock di lettura-scrittura (più costoso)

In LINUX: `rwlock_t` e `struct rw_semaphore`

+ A più processi è permesso ottenere il lock in modalità lettura a un solo processo " " " " " scrittura

Utili quando: • È facile identificare processi che si limitano alla lettura/scrittura  
• Ci sono più lettori che scrittori

### Problema dei cinque filosofi

5 filosofi passano la vita pensando e mangiando, non interagiscono, tentano di procurarsi due bacchette per mangiare il loro posto al centro della tavola rotonda

atavola ce ne sono  
cinque

• Il filosofo può appropriarsi di una bacchetta alla volta e non può strapparla dalle mani del vicino

• Un filosofo affamato, che possiede due bacchette, mangia; terminato, posa le bacchette e torna a pensare

Variazioni condivise: Zupprezza di uso (dati)

bacchetta[5] (5 semafori inizializzati a 1)

i - FILOSOFO

```
while (true) {  
    wait (bacchetta[i]);  
    wait (bacchetta[(i+1)%5]);  
    // mangia  
    signal (bacchetta[i]);  
    signal (bacchetta[(i+1)%5]);  
    // pensa  
}
```

• Garantisce che due non mangino contemporaneamente

• Possibile stallo => deadlock

↓  
Ciascuno a impo mena di una bacchetta

Soluzioni: • Ammetto solo quattro filosofi

\* Un filosofo può prendere le bacchette solo se entrambe disponibili (solo nella sezione critica)

• Posto pari → bacchetta alla sinistra ⇒ Protocollo di accesso  
Posto dispari → " " destra alle risorse



monitor filosofo {

enum { PENSA, AFFAMATO, MANGIA; stato[5];  
condition self[5];

5 filosofi, ciascuno con i propri stati  
variabile, numero finito  
di valori costanti  
• se le bacchette non ci sono va in condition

void prende (int i) {  
    stato[i] = AFFAMATO;  
    test(i);  
    if (stato[i] != MANGIA)  
        self[i].wait;  
}

void posa (int i) {  
    stato[i] = PENSA;  
    // controllo a vicinia sinistra e a destra  
    test((i+4)%5);  
    test((i+1)%5);  
}

void test (int i) {  
    \* if ( (stato[(i+4)%5] != MANGIA) && (stato[i] == AFFAMATO) && (stato[(i+1)%5] != MANGIA) )  
        stato[i] = MANGIA;  
        self[i].signal(); // mi regolo  
}

Codice\_iniz() {  
    for (int i=0; i<5; i++)  
        stato[i] = PENSA; // tutti i filosofi pensano  
}

• Ora ho solo possibile starvation

## Sincronizzazione in Linux

- Prima della versione 2.6, kernel senza diritto di prelazione => disabilitavo gli interrupt per ottenere sezioni critiche di breve durata
- Ora ho diversi meccanismi di sincronizzazione:
  - Lock mutex: variabile booleana, indica se il lock è disponibile a modificare con le system call mutex\_lock() mutex\_unlock()
  - Spinlock e semafori
  - Interatomici: rappresentati con il tipo opaco atomic\_t (senza interruzioni) la struttura è su file  
+ aggiornamenti del contatore  
x insufficienti se la race condition coinvolge diverse variabili

Se il lock deve essere mantenuto per breve tempo:

- su SMP lo spinlock gestisce la sezione critica
- su monoprocesso grazie all'abilitazione/inibizione del diritto di prelazione

Se il lock va mantenuto più a lungo, uso lock mutex o semafori senza busy waiting



## Sincronizzazione in POSIX

La API Pthreads fornisce diversi tipi di lock e variabili condizionali per la sincronizzazione dei thread a livello utente

- Lock mutex è fondamentale: thread, che sta per accedere alla propria sezione critica, si appropria del lock e lo rimuove all'uscita
- Due versioni per i semafori, con e senza nome
  - usati per i processi non correlati

LOCK:

```
#include <pthread.h>
pthread_mutex_t mutex;
/* creazione ed inizializzazione del lock mutex */
pthread_mutex_init(&mutex, NULL);
/* acquisizione */
pthread_mutex_lock(&mutex);
/* sezione critica */
/* rilascio */
pthread_mutex_unlock(&mutex);
```

↳ puntatore → Inizializzo il lock con gli attributi predefiniti.

0: per corretto funzionamento  
≠ 0: errore

SEMAFORO CON NOME

```
#include <semaphore.h>
sem_t *sem;
/* creazione e inizializzazione ad 1 */
sem = sem_open("SEM", O_CREAT, 0666, 1);
/* acquisizione */
sem_wait(sem);
/* sezione critica */
/* rilascio */
sem_post(sem);
```

SENZA NOME

```
sem_t sem;
sem_init(&sem, 0, 1);
sem_wait(&sem);
sem_post(&sem);
```

↳ livello di condivisione

usabile da tutti i processi che ne conoscano il nome

### Nono monitor (C, C++)

↓ VARIABILI CONDIZION

```
pthread_mutex_t mutex;
pthread_cond_t cond_var;

pthread_mutex_init(&mutex, NULL);
pthread_cond_init(&cond_var, NULL);

in attesa pthread_mutex_lock(&mutex);
while (a != b)
    pthread_cond_wait(&cond_var, &mutex);

pthread_mutex_unlock(&mutex);

reglia pthread_mutex_lock(&mutex);
a = b;
pthread_cond_signal(&cond_var);
pthread_mutex_unlock(&mutex);
```



## Approcci alternativi

- OpenMP **#pragma omp critical** specifica che la successiva regione di codice è una regione critica in cui un solo thread può essere attivo  
↳ semaforo o lock/mutex
- Linguaggi di programmazione funzionali + IP Pluriodi esecuzione è una serie di valutazioni di funzioni matematiche  
↳ non mantengono uno stato + una volta definito il valore di una variabile, questo è immutabile  
Erlang, Scala, Haskell



## 8. DEADLOCK

### Il problema del deadlock

In un ambiente multiprogrammato, più thread possono entrare in competizione per ottenere un numero finito di risorse

**Deadlock** è l'insieme di thread bloccati, ciascuno dei quali possiede almeno una risorsa ed attende di acquisire una o più risorse allocate ad altri thread

- È difficile individuarlo, in quanto avviene solo in particolari condizioni di scheduling
- Maggio: SO non offrono soluzione, in quanto è compito del programmatore di evitare deadlock

... Ciascun processo in un insieme di processi attende un evento che può essere causato solo da un altro processo dell'insieme

ESEMPIO 1

| $T_1$   | $T_2$   |
|---------|---------|
| wait(A) | wait(B) |
| wait(B) | wait(A) |

ESEMPIO 2: attraversamento di un ponte:

↓  
(risorsa)

se due macchine, provenienti da sensi opposti, attraversano il ponte si genera un deadlock, che è possibile risolvere solo se una delle due torna indietro

• Possibile starvation

### Modello del sistema

Tipi di risorse:  $R_1, R_2, \dots, R_m$ , ciascun tipo di risorsa  $R_i$  ha  $W_i$  istanze

Il protocollo di accesso alle risorse da parte dei thread prevede:

+ Richiesta: se la richiesta non può essere soddisfatta immediatamente, il thread ne deve attendere il rilascio

+ Utilizzo

+ Rilascio

System call  
• open/close file  
• wait/signal semaphore  
...

• SO controlla (e se si usa una risorsa gestita da kernel) se il thread utente ne abbia fatto richiesta e se gli sia stata accordata

• Una tabella di sistema registra lo stato di ogni risorsa se allocata, indica il thread relativo

→ può avere una lista di attesa (coda)

### Deadlock nelle applicazioni multithread

• Livelock: i thread non sono bloccati ma di fatto non progrediscono

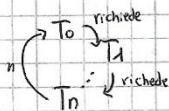
es. Collisioni su rete Ethernet



## Caratterizzazione dei deadlock

Una situazione di deadlock può verificarsi se valgono simultaneamente:

- **MUTUA ESCLUSIONE:** esiste almeno una risorsa non condivisibile (usabile da un thread alla volta)
- **POSSESSO ED ATTESA:** un thread, che possiede almeno una risorsa, attende di acquisire altre risorse possedute da altri thread
- **IMPOSSIBILITÀ DI PRELAZIONE:** una risorsa può essere rilasciata solo volontariamente
- **ATTESA CIRCOLARE:** insieme  $\{T_0, T_1, \dots, T_n\}$  di thread in attesa



## Grafo di assegnazione delle risorse

Grafo orientato costituito da nodi  $V$  connessi con archi  $E$

arco di richiesta  $T_i \rightarrow R_j$

arco di assegnazione  $R_j \rightarrow T_i$

• l'insieme  $V$  è partizionato in:

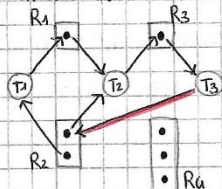
+  $T = \{T_1, T_2, \dots, T_n\}$  thread attivi

+  $R = \{R_1, R_2, \dots, R_m\}$  tipi di risorse

Thread

Tipo di risorsa con 2 istanze

esempio con deadlock



ho due cicli

Non ho cicli  $\Rightarrow$  non c'è deadlock

Ho cicli  $\Rightarrow$  può essere deadlock



## Metodi per la gestione dei deadlock

Aziunare che il sistema non entri mai in deadlock:

+ **Prevenire i deadlock**: evitare che si verifichino contemporaneamente le 4 situazioni  
 $\Rightarrow$  basso utilizzo delle risorse e throughput ridotto

+ **Evitare i deadlock**: evitare quegli stati che possono evolvere verso deadlock  
 $\Rightarrow$  informazioni aggiuntive sulle richieste dei thread

• Se ho deadlock, posso sempre ripristinare il sistema

- **PREVENIRE I DEADLOCK**: limitare le modalità di richiesta/accanto delle/alle risorse per invalidare una delle quattro condizioni necessarie

X La **Mutua esclusione** non può essere prevenuta

✓ **Posse e attesa**: bisogna garantire, quando un thread richiede una risorsa, non ne possiede altre  
 o di esigenze del thread di stabilire ed allocare tutte le risorse necessarie prima che inizi l'esecuzione  
 • Possibile **attesa indefinita**

✓ **Impossibilità di prelazione**: Se un thread richiede altre risorse che non possono essere allocate immediatamente allora rilascia tutte le risorse possedute  
 $\downarrow$   
 aggiunte alla lista di attesa

✓ **Attesa circolare**: Impongo un ordinamento totale su tutti i tipi di risorse e si pretende che ciascun thread richieda risorse in ordine crescente

In alternativa, ogni thread, prima di richiedere una istanza di  $R_j$  deve rilasciare tutte le istanze di  $R_i$  tali che  $\#(R_i) > \#(R_j)$

- **EVITARE IL DEADLOCK**: mentre prevenire può causare scarso utilizzo e ridotta produttività del sistema, evitare il deadlock bisogna conoscere informazioni aggiuntive

✓ Ciascun thread dichiara il numero massimo di risorse di ciascun tipo di cui potrà usufruire

✓ **algoritmo deadlock-avoidance**: esamina dinamicamente lo stato di allocazione delle risorse (per evitare attesa circolare)  
 $\downarrow$  garantisce che un sistema rimanga nello stato sicuro  
 $\downarrow$  # di risorse disponibili e allocate  
 # massimo di richieste dei thread

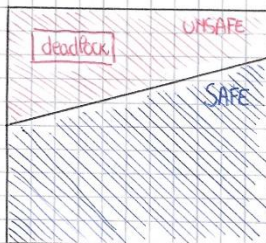
**Stato sicuro**

Quando un thread richiede una risorsa disponibile, il sistema deve decidere se l'allocazione immediata lascia il sistema in stato sicuro

• Se le richieste di  $T_i$  non possono essere soddisfatte, allora  $T_i$  attende che tutti i  $T_j$  abbiano terminato

$\Downarrow$   
 esiste una **sequenza sicura** di esecuzione di tutti i thread

$\Downarrow$   
 la sequenza  $\langle T_1 \dots T_n \rangle$  è sicura, se  $\forall T_i$ , le risorse che  $T_i$  può ancora richiedere possono essere allocate sfruttando tutte le risorse disponibili



ESEMPIO: 12 istanze disponibili

| Thread | Richiesta MAX | Unità possedute |
|--------|---------------|-----------------|
| $T_0$  | 10            | 5               |
| $T_1$  | 6             | 2               |
| $T_2$  | 9             | 2               |

Se sicura  $\langle T_1, T_0, T_2 \rangle \rightarrow$  se  $T_2$  richiedesse un'altra unità lo stato è non sicuro



## Algoritmi per evitare il deadlock

### Algoritmo con grafodi assegnazione risorse $O(n^2)$

Arco di reclamo  $T_i \rightarrow R_j$  indica che il thread  $T_i$  può richiedere la risorsa  $R_j$  in futuro

Quando una risorsa viene rilasciata da un thread  
l'arco di assegnazione diventa arco di reclamo

- Le risorse sono reclamate a priori nel sistema: prima che il thread  $T_i$  inizi l'esecuzione, tutti i suoi archi di reclamo devono essere già stati inseriti
- Supponiamo che  $T_i$  richieda  $R_j$ , la richiesta può essere accordata solo se la conferenza dell'arco di reclamo in arco di assegnazione non provoca la formazione di un ciclo nel grafodi assegnazione delle risorse

### Algoritmo del banchiere $O(m \times n^2)$

Permette di gestire istanze multipli di una risorsa

- Ciascun thread deve dichiarare a priori il massimo impiego di risorse
- Quando un thread richiede una risorsa ~~non~~<sup>non</sup> può essere servito immediatamente, per mantenere lo stato sicuro
- Quando ad un thread sono allocate tutte le risorse, deve restituirle in tempo finito

Sia  $n$  # di thread,  $m$  # tipi di risorse

• Available: vettore di lunghezza  $m$   
 $Available[j] = k$   $k$  istanze sono disponibili per  $R_j$

• Max: matrice  $n \times m$   
 $Max[i, j] = k$  il thread  $T_i$  può richiedere  $k$  istanze a  $R_j$

$Need[i, j] = Max[i, j] - Allocation[i, j]$

• Allocation: matrice  $n \times m$   
 $Allocation[i, j] = k$  a  $T_i$  sono allocate  $k$  istanze di  $R_j$

NB Nel tempo le strutture variano di dimensione e di valore

• Need: matrice  $n \times m$   
 $Need[i, j] = k$   $T_i$  può richiedere  $k$  istanze a  $R_j$  per completare il task

### Algoritmo di verifica

- Siano Work e Finish vettori di lunghezza  $m$  ed  $n$   
 $Work \leftarrow Available$   
 $Finish[i] \leftarrow false$   $i = 1 \dots n$
- Si cerchi  $i$  tale che valgano contemporaneamente  
 $Finish[i] = false$  e  $i$  non esiste  
 $Need_i \leq Work$  vai a 6.
- Si assegna:  $Work \leftarrow Work + Allocation_i$  e si torna a 2  
 $Finish[i] = true$
- Se  $Finish[i] = true \forall i$ , il sistema è in stato sicuro altrimenti non lo è.



## - RILEVAMENTO DEL DEADLOCK

SO non ha tecniche per prevenire il DEADLOCK  $\Rightarrow$  gli permette così di |  
lucharvi

per ripristinare le funzionalità  
devono:

- algoritmi di alteramento del deadlock
- algoritmi di ripristino dal deadlock

+ Costi aggiuntivi • memorizzazione informazioni  
per il ripristino

- perdita di calcoli già svolti (dovrà rieseguirli)

### Grafo di attesa

- nodi  $\rightarrow$  thread ( $n$ )
- L'arco  $T_i \rightarrow T_j$  esiste se  $T_i$  attende  $T_j$  che  
satisfaccia una risorsa che  
gli occorre

! Si verifica l'eventuale presenza!  
di cicli nel grafo  $\downarrow$   
 $O(n^2)$  DEADLOCK

! Toolkit BCC: deadlock\_detector  $\leadsto$  pthread\_mutex\_lock();  
costruisce il grafo di attesa  $\rightarrow$  unlock();

### Algoritmo di alteramento

- Available: vettore  $[m]$ , numero di risorse disponibili
- Allocation: matrice  $[n \times m]$ , numero di risorse di ciascun tipo attualmente allocate a ciascun thread
- Request: matrice  $[n \times m]$ , richiesta corrente di ciascun thread  
Request  $[i, j] = K$   
 $T_i \downarrow R_j$   $\rightarrow$  istanze supplementari

+ indaga su ogni possibile sequenza di assegnazione  
per i thread da completare

1. Work e Finish  
 $[m]$   $[n]$   $Work \leftarrow Available$   
 $Finish[i] \leftarrow False$  se Allocation  $\neq 0$   
 $\leftarrow True$  altrimenti
2. Se trovi un indice  $i$ :  $Finish[i] = False$  se  $i$  non esiste  
Request  $i \leq work$  vai a 4.
3. Si assegna  $work \leftarrow work + allocation_i$  e si torna a 2  
 $Finish[i] \leftarrow true$
4. Se  $Finish[i] = False$  per qualche  $1 \leq i \leq n$  il sistema è in uno stato di deadlock  
Inoltre  $T_i$  è in deadlock

$\downarrow O(m \times n^2)$

- Si ipotizza ottimisticamente che  $T_i$  non intenda richiedere altre risorse  
e restituisca in tempo finito quelle che già possiede

quando e quanto spesso  
chiamare l'algoritmo

- frequenza dei deadlock
- # thread influenzati  $\rightarrow$

$\Rightarrow$  • ogni volta che la richiesta  
non può essere soddisfatta  
oppure  
• l'utilizzo della CPU è sotto al 60%



## -RIPRISTINO DAL DEADLOCK

- Terminazione in **abort**:
  - tutti i thread del deadlock
  - un thread alla volta

L'ordine di eliminazione è dato da:

- priorità dei thread
- tempi esecuzione
- risorse impiegate e necessità

- prelazione di risorse
  - + selezione di una vittima
  - + roll back: thread deve tornare allo stato sicuro
  - + starvation



## 9. MEMORIA CENTRALE Background

Lo scheduling della CPU:

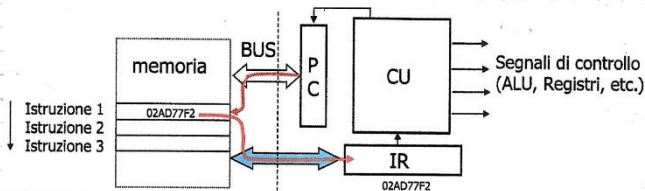
- ✓ migliora l'utilizzo della risorsa CPU
- ✓ migliora la reattività del sistema

- minor tempo di risposta
- maggiore throughput



NB In memoria saranno presenti più processi

Il metodo di gestione della memoria dipende dall'architettura HW



fase di fetch: la CPU preleva le istruzioni dalla memoria, in base al PC (contatore di programma)

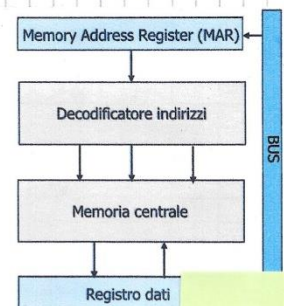
La memoria vede solo un flusso di indirizzi, non sa né la modalità di creazione né a cosa si riferiscono

La CPU accede direttamente solo alla memoria principale e ai registri

|                  |                                                       |
|------------------|-------------------------------------------------------|
|                  | ACCESSO                                               |
| memoria centrale | diversi cicli di clock ⇒ <i>stallo del processore</i> |
| registri         | 1 ciclo di clock                                      |

indirizzo + richiesta di lettura

indirizzo + dati + richiesta di scrittura



SOLUZIONE: + uso memoria cache  
+ multithreading hardware

NB fondamentale la protezione della memoria

### Registri base e limite

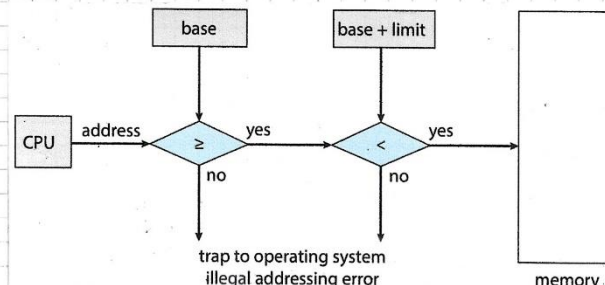
Ogni processo deve avere uno spazio di memoria separato

+ Possibile soluzione: **registro base**: contiene il più piccolo indirizzo della memoria fisica allocata al processo

**registro limite**: determina la dimensione dello spazio di memoria

Per la protezione, la CPU confronta che  $\text{Indirizzo} \geq \text{base}$   
 $\text{Indirizzo} < \text{base} + \text{limite}$

SO si carica con istruzioni privilegiate  
↓  
l'utente non può accedere o modificare





## Associazione degli indirizzi

I programmi risiedono su disco sotto forma di file binari eseguibili

Per essere eseguito va caricato in memoria, inserito nel contesto di un processo  $\Rightarrow$  ESECUZIONE

Terminato, il processo rilascia lo spazio ad esso allocato

- Ci sono diversi stadi:
1. Gli indirizzi nel programma sorgente sono simbolici nome variabile
  2. Il compilatore associa INDIRIZZI SIMBOLICI a INDIRIZZI RILOCABILI tabella dei simboli
  3. Il linker/loader associa INDIRIZZI RILOCABILI a INDIRIZZI ASSOLUTI 76010

Si crea una mappa.

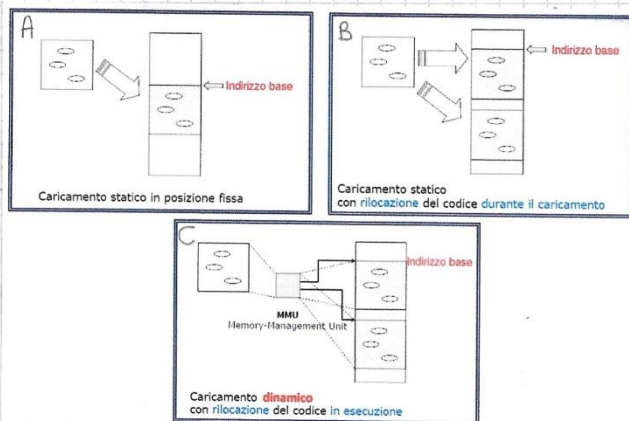
L'associazione (binding) di istruzioni - dati a indirizzi di memoria avvengono:

A Compilazione: se la posizione in memoria del processo  $\rightarrow$  genero codice ASSOLUTO  
1  $\rightarrow$  3  
 se cambia devo ricompilare il codice

B Caricamento: se la posizione in compilazione non è nota  $\rightarrow$  devo generare codice RILOCABILE  
2  $\rightarrow$  3  
 se cambia, il codice deve essere ricaricato in memoria (il linker/loader associa poi indirizzi ASSOLUTI)

C Esecuzione: se il processo può essere spostato a run-time da un segmento di memoria all'altro  
3  $\rightarrow$  3  
 (CODICE DINAMICAMENTE RILOCABILE)  
 il binding viene rimandato fino all'esecuzione

- Contiene solo riferimenti a se stesso
- Supporto HW per il mapping degli indirizzi (Tabella delle pagine)



### • Binding statico (A o B)

Indirizzi logici = indirizzi fisici

### • Binding dinamico (C)

Indirizzi logici  $\neq$  indirizzi fisici (virtuali)  
 + Posso spostare processi in memoria in esecuzione

$\Rightarrow$  MEMORIA VIRTUALE  
 e  
 SWAPPING

La CPU genera indirizzi logici, ma l'unità di memoria vede indirizzi fisici

INDIRIZZI LOGICI  $\Rightarrow$  binding  $\Rightarrow$  INDIRIZZI FISICI



## Spazi di indirizzi logici e fisici

### INDIRIZZI LOGICI

generati dalla CPU (indirizzi virtuali)  
(piccoli)

(contiguo)  
→ lo spazio è l'insieme di tutti gli indirizzi logici generati da un programma

### INDIRIZZI FISICI

contenuti nel memory address register (MAR)  
utilizzati dall'unità di memoria  
(grandi)

→ lo spazio è l'insieme di tutti gli indirizzi fisici visti dalla memoria per quel progr.  
(frammentato)

## Memory Management Unit (MMU)

Dispositivo HW che mappa indirizzi virtuali su indirizzi fisici a RUN TIME

- Il valore contenuto nel registro di allocazione viene sommato ad ogni indirizzo generato dal processo utente
- Il programma utente ragiona per indirizzi virtuali, non conosce il mapping fisico



## Caricamento dinamico

L'intero programma non deve risiedere in memoria per essere eseguito

Le routine vengono caricate, solo quando utilizzate

Il caricamento dinamico è implementato mediante software codificato (modulare)

↓ SO fornisce solo librerie di procedure

⇒ routine come quelle di errore non devono necessariamente ma solo all'occorrenza

## Link dinamico delle librerie

Link statico: librerie di sistema e codice sono combinati dal caricatore nell'immagine binaria

Link dinamico: link rinviato fino all'esecuzione

→ • evita che ogni programma debba avere copie di librerie

"Stub": piccole librerie per localizzare le routine opportune. Lo stub rimpiazza se stesso ed esegue.

SO verifica se le routine sono nello spazio di indirizzi del processo oppure no e le rende assemblabili

SO supera le barriere di protezione fra processi per accedere a codice condiviso



## Allocazione contigua della memoria

La memoria principale è una risorsa limitata, e deve essere allocata in modo efficiente

PARTE ALTA: processi di sistema  
+ vettore degli interrupt

PARTE BASSA: processi utente

Allocazione  
contigua

- a partizione singola: un solo processo alla volta può essere allocato in memoria (memoria multiprogrammata)

- a partizioni multiple:

- + PARTIZIONI FISSE: la dimensione di ogni partizione è fissata a priori

SO cerca, per un processo schedato, una partizione libera di dimensioni sufficienti

- X Frammentazione interna: sottoutilizzo

- X Grado di multiprogrammazione limitato (dal numero di partizioni)

- X Dimensione max di un processo data dalla partizione più estesa

- + PARTIZIONI VARIABILI: ogni partizione è allocata dinamicamente in base alla dimensione del processo da allocare

- ✓ Elimina frammentazione interna

- ✓ Grado di multiprogrammazione variabile

- ✓ Dim. max di un processo è data dalla disponibilità di spazio libero

- X Scelta area da allocare

- X Frammentazione esterna

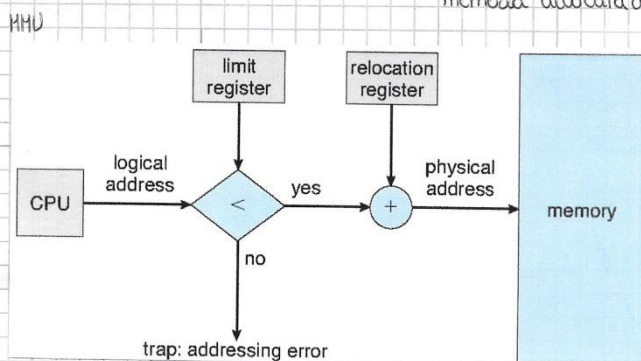
SO conserva informazioni su partizioni:

- allocate
- libere

- + registro limite: definisce l'intervallo di variabilità degli indirizzi logici

$\downarrow$  valore contenuto nel registro

- + registro di allocazione: valore del più piccolo indirizzo fisico dell'area di memoria allocata al processo



Si usano per proteggere fra loro i processi utente

Codice kernel può variare dimensione



## Come allocare memoria dinamicamente (in blocchi)

Best-fit: è allocato il buco più piccolo capace di contenere il processo } migliore

First-fit: " " primo grande abbastanza

Worst-fit: " " più grande

0.5n blocchi vanno persi per fram.

1/3 memoria utilizzabile!

## Frammentazione

**INTERNA:** la memoria allocata può essere leggermente maggiore della memoria effettivamente richiesta e la memoria non utilizzata tra stack e heap

**ESTERNA:** è disponibile spazio, ma non è contiguo

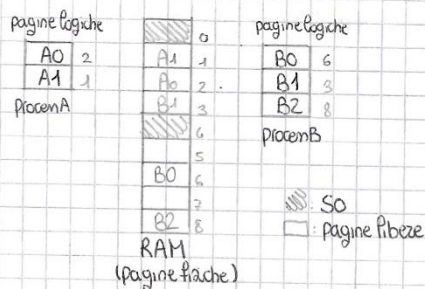
Possibilità di ridurre con la **compattazione**:

- spostando processi per ottenere un blocco contiguo
- possibile solo con rilocazione dinamica
- processi con I/O pendenti non si spostano almeno che non lo faccia il kernel

## Paginazione

Con la paginazione lo spazio degli indirizzi fisici di un processo può essere non contiguo

- ✓ Evita frammentazione esterna → alloco solo blocchi di dimensione prefissata
- ✓ Evita blocchi di memoria di dimensione variabile



frame / pagine fisiche: blocchi di memoria fisica  
dimensione: potenza di 2 (4KB-1GB)

pagine logiche: blocchi di memoria logica

Per eseguire un programma di dimensione  $n$  pagine, è necessario trovare  $n$  frame liberi prima di caricare il programma

Si impiega una **tabella delle pagine** per tradurre gli indirizzi logici negli indirizzi fisici corrispondenti

X Ho solo frammentazione interna

## Schema di traduzione degli indirizzi

Indirizzo logico generato da CPU

Numero di pagina  
indice della tabella  
delle pagine

|       |     |
|-------|-----|
| $p$   | $d$ |
| $m-n$ | $n$ |

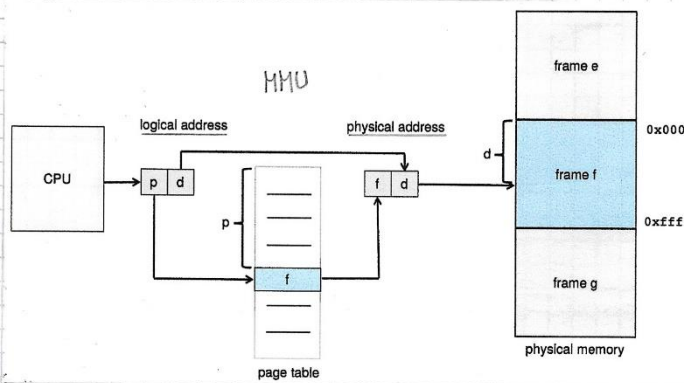
Offset di pagina

Combinato con l'indirizzo base per definire l'indirizzo fisico che viene inviato all'unità di memoria

- contiene l'indirizzo base di ciascuna frame nella memoria fisica (# frame)

spazio logico di  $2^m$  indirizzi con  $2^{m-n}$  pagine di dimensione  $2^n$





esempio

MEMORIA LOGICA

|    |        |
|----|--------|
| 0  | page 0 |
| 4  | page 1 |
| 8  | page 2 |
| 12 | page 3 |
| 16 |        |

TABELLA DELLE PAGINE

|   |   |
|---|---|
| 0 | 5 |
| 1 | 6 |
| 2 | 1 |
| 3 | 2 |

MEMORIA FISICA

| # Frame |        |   |
|---------|--------|---|
| 0       |        | 0 |
| 4       | Page 2 | 1 |
| 8       | Page 3 | 2 |
| 12      |        | 3 |
| 16      |        | 4 |
| 20      | Page 0 | 5 |
| 24      | Page 1 | 6 |
| 28      |        | 7 |
| 32      |        |   |

Quindi  $2^{m-n}$  pagine di dimensione  $2^n$

$$2^{m-2} = 4 \text{ pagine} \\ \leadsto m=6$$

$$2^n = 4 \leadsto n=2 \\ \text{pagine da 4 byte}$$

32 byte con pagine da 4 byte

Supponiamo di prendere 6, in binario  $0110$  offset rappresenta 1

ad 1 nella tabella delle pagine ho 6

$$\Rightarrow 6 \times 4 + 2 = 26$$

esempio dim pagina = 2048 byte  
dim processo = 72768 byte  $\Rightarrow$  avrò 35 pagine + 1088 byte (rimanenti) = 36 pagine

frammentazione 2048 - 1088 = 960 byte  
intera

nel caso peggiore avrei 1 frame - 1 byte

NB Piccole frame non sono sempre preferibili, l'HO lavora meglio per pagine grandi



## Tabella dei frame

SO gestisce la memoria fisica

deve sapere

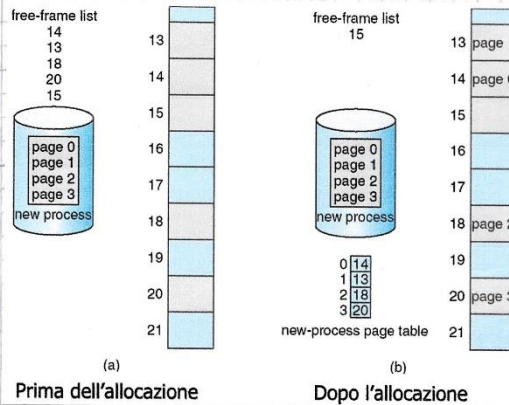
- quali frame sono assegnati e a chi
- quanti e quali frame sono liberi

es: memoria 1G =  $2^{30}$   
 pagina 1K =  $2^{12}$   
 $\# \text{ tab. frame} = \frac{2^{30}}{2^{12}} = 2^{18}$

scrittura nella tabella dei frame

entry: libera/assegnata

→ a chi



**NB** La paginazione impone al SO la gestione di strutture dati aggiuntive che occupano memoria e aggiungono overhead

↓ Context-switch

→ Privilegio globale (tabella frame)  
 → Privilegio singolo (tabella delle pagine)

MA rende migliore la memorizzazione: spazio logico  $2^{32}$  Byte = 4GB  
 spazio fisico  $2^{32}$  frame da 4Kb = 16TB  
 $\times 2^{12} \text{ byte}$

## Implementazione della tabella delle pagine

PRIMA insieme di registri dedicati

- ✓ Traduzione dell'indirizzo efficiente
- ✗ lungo context-switch
- ✗ solo 256 elementi

ORA

- registro Page-Table Base Register (PTBR) punta all'inizio della tabella
- registro Page-Table Length Register (PTLR) indica la dimensione della tabella

risiede in memoria centrale (ho una tabella per ogni processo)

✗ due accessi alla memoria uno per la tabella e uno per il dato/istruzione

⇒ uso registri associativi (translation look-aside buffer TLB)

faccio una ricerca parallela veloce su una piccola tabella

+ ASID (address space identifier) per ogni elemento per la protezione, prima di controllare l'indirizzo fisico lo controllo con quello della pagina virtuale, altrimenti ho TLB miss

• consente di avere in TLB elementi di diversi processi

• aggiornamenti per ogni contesto, TLB va svuotata per evitare informazioni errate



## Memoria associativa

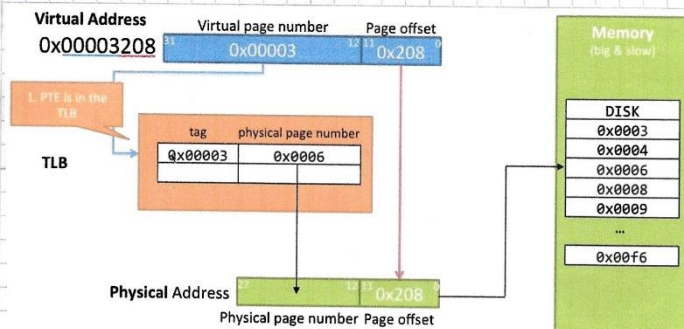
| # Pagina | # Frame | ASID |
|----------|---------|------|
|          |         |      |
|          |         |      |
|          |         |      |

processo a cui la pagina logica appartiene

Traduzione dell'indirizzo (p.d)

A. p è nel registro associativo, recupero il numero del frame

B. altrimenti, recupero il numero di frame dalla tabella delle pagine in memoria



TLBmin: l'informazione deve essere spostata in TLB per essere acceduta rapidamente

+ POLITICHE DI SOSTITUZIONE

NB alcuni elementi possono essere vincolati permanentemente (elementi codice kernel)

## Tempo di accesso effettivo

- Lookup associativo =  $\epsilon$  unità tempo
- Hit ratio  $\alpha$  = # volte che un numero di pagina viene reperito nel registro ass.

$$EAT = (\epsilon + \epsilon)\alpha + (2\epsilon + \epsilon)(1 - \alpha) = 2\epsilon - \alpha\epsilon + \epsilon$$

•  $\epsilon$  durata = 10 n sec  
 $\epsilon \approx 1$  n sec

$$\alpha = 99\% \Rightarrow EAT = 11,1 \text{ n sec}$$

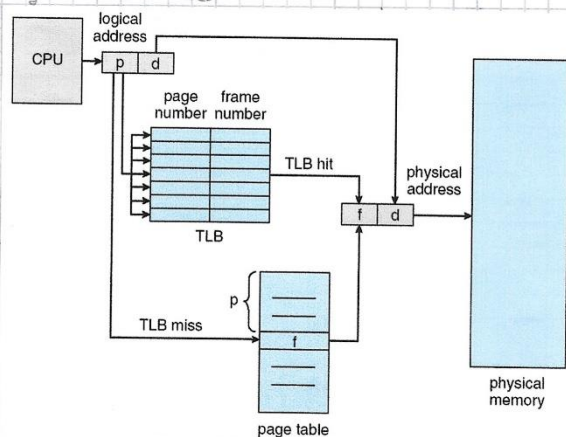
ad oggi, l'efficienza della TLB è parte della PIPELINE  
✓ nessuna penalizzazione nelle prestazioni  $\epsilon = 0$

HA devo • TBL piccola (64-1024 voci)

• TLB separate per istruzioni e dati (aumentano # elementi disponibili)

Con più livelli di TLB, quindi EAT è difficile da calcolare

## Paginazione HW con TLB





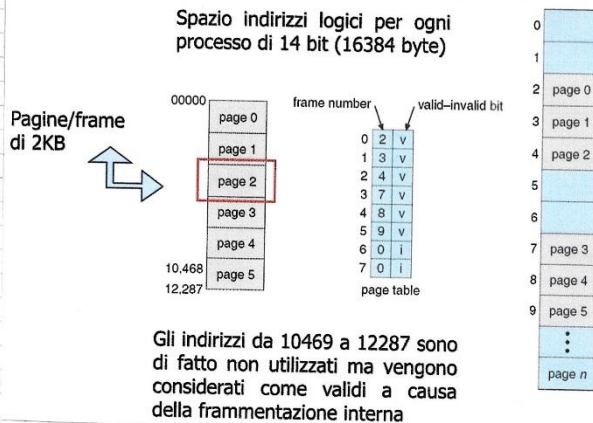
## Protezione della memoria

Possiamo associare uno o più bit di protezione a ciascun frame della tabella delle pagine.

↓  
solo lettura  
lettura/scrittura e/o esecuzione

Tutti i riferimenti in memoria passano attraverso la tabella delle pagine, e quando codificato viene violato causa **trap al SO**

+ Nelle architetture più vecchie, al posto del PTLR, si usava un bit di validità  
VALIDO: la pagina era nello spazio degli indirizzi logici del processo  
NON VALIDO: non  
consentiva di riconoscere gli indirizzi illegali  
Impostato dal SO



## Pagine condivise

- **Codice condiviso**: una copia di codice di sola lettura viene condivisa fra processi.  
↓ deve apparire nella stessa locazione nello spazio degli indirizzi logici di tutti i processi.
- **Codice e dati privati**: ciascun processo mantiene una copia separata dei dati e del codice.  
può apparire ovunque

## Struttura della tabella delle pagine

Queste strutture dati **occupano troppa memoria** => ottimizziamo l'occupazione della memoria e l'accesso all'informazione contenuta

## PAGINAZIONE GERARCHICA

Dividere lo spazio degli indirizzi logici in più tabelle delle pagine (a due livelli).

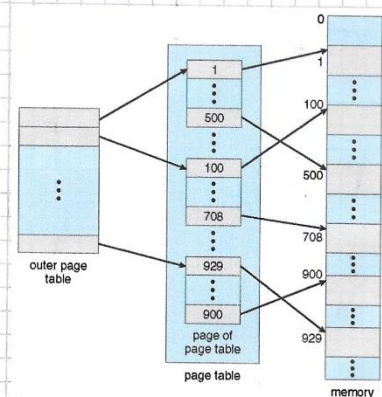
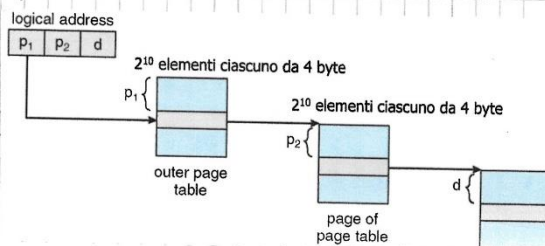
X Troppi accessi, non adatto a 64

32 bit con pagine da 4KB

| # pagina | offset |
|----------|--------|
| P1       | d      |
| 10       | 12 = n |

20 = m - n

P1: indice della tabella delle p. esterne  
P2: indice della tabella delle p. interne (spostamento)





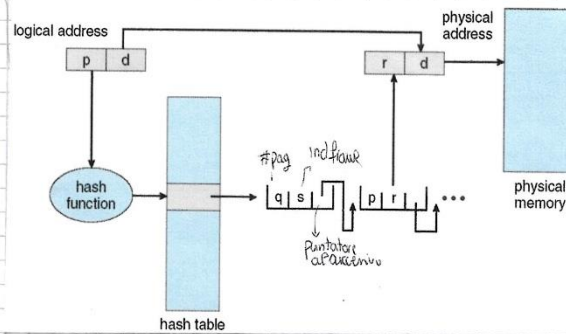
## TABELLA DELLE PAGINE HASH

L'argomento della funzione hash è il numero di pagina virtuale

Gestione collisioni: ogni elemento della tabella hash, ha una lista concatenata (di trabocco)

- 3 campi:
- # pagina virtuale
  - indirizzo del frame
  - puntatore al successivo elemento della lista

✓ Indirizza logica a spazi



Per 64 bit si usa tabella delle pagine a gruppi

- ✓ riduco memoria utilizzata
- ✓ aumento prestazioni

ma ciascun elemento della tab delle pag contiene riferimenti alle pagine fisiche corrispondenti ad un gruppo di pagine virtuali contigue

## TABELLA DELLE PAGINE INVERTITA

Si associa una tabella delle pagine ad ogni processo, che contiene un elemento per ogni pagina virtuale che il processo sta utilizzando

MA ogni tabella può contenere milioni di elementi ed occupare molta memoria

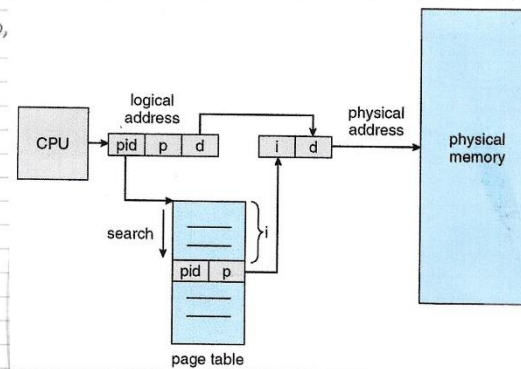
allora uso la tabella delle pagine inversa

- un elemento della tabella per ogni frame
- + # pagina virtuale memorizzata nel frame
- + PID del processo che possiede tale pagina

✓ Riduzione memoria richiesta

X Incremento del tempo di ricerca (cerca su tutta la tabella)

X è difficile realizzare memoria condivisa → ho solo un elemento che indica la pagina virtuale corrispondente ad ogni pagina fisica



Possiamo usare tabelle hash, dato che la ricerca è  $O(1)$ , anche se ogni accesso aggiunge un riferimento alla memoria  
si aggiunge TLB ↓



# Swapping

Swapped out: un processo può essere temporaneamente trasferito dalla memoria centrale alla backing store

partizione del disco ad accesso rapido  
capiente per accogliere tutti i processi utente  
(doppia alla RAM)

X costoso => fa accesso alla memoria di massa

✓ aumento il grado di multiprogrammazione

tempo di swap: dovuto al trasferimento di dati, e direttamente proporzionale alla quantità di memoria riservata

SO mantiene una ready queue dei processi

- presenti in memoria
- con l'immagine riversata su disco

! Swapping con processi con I/O pendenti !

durante l'esecuzione può essere ricaricato in uno spazio di memoria diverso

Normalmente lo swapping è disattivato, si attiva quando la memoria allocata supera una soglia.

Se il primo processo da eseguire non è in memoria, e la memoria è piena => tempo di context-switch elevato

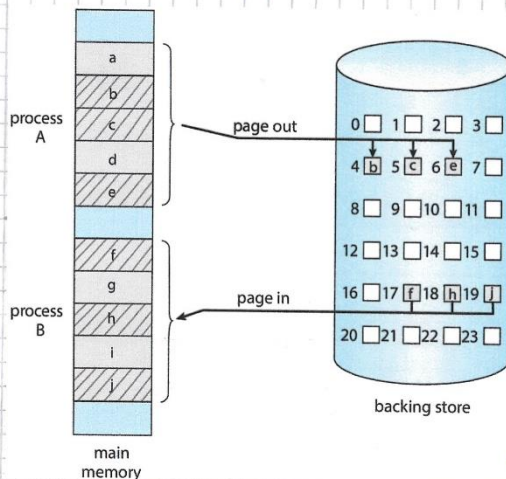
Processi con I/O pendenti non possono essere spostati su disco, salvo effettuare il trasferimento di dati dovuto all'I/O nello spazio kernel => aggiunge overhead

Lo swapping più utilizzato prevede:

- Si spostano solo alcune pagine di un processo
- Serve memoria virtuale

PAGING

page in/out: spostare pagine dalla memoria centrale



NB Non è supportato sugli smartphone



## 10. LA MEMORIA VIRTUALE

### Background

Il codice, per poter essere eseguito, deve risiedere nella memoria centrale.

- Non serve caricarlo tutto, è raro che serva tutto il codice nello stesso momento
- Codice di errore
- Array e tabelle sovradimensionati
- Raramente eseguite

Uteremo d'ora in poi porzioni di codice

- ✓ Non c'è il vincolo sulla quantità di memoria fisica disponibile
- ✓ aumentare il grado di multi-programmazione
- ✓ Per caricare/scaricare le porzioni è necessario un minor numero di operazioni di ingresso/uscita

⇒ aumentano le prestazioni del sistema

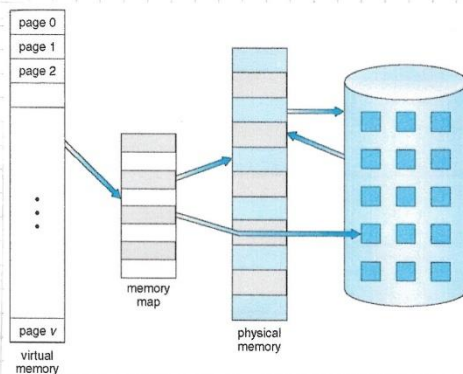
### Memoria Virtuale

separazione della memoria logica dell'utente dalla memoria fisica

- Solo parte del programma si trova in memoria per l'esecuzione
- spazio indirizzi LOGICI >> spazio indirizzi FISICI
- Porzioni di spazio fisico sono condivise da più processi

implementata con paginazione su richiesta

- spazio degli indirizzi virtuali → è la collocazione dei processi in memoria dal punto di vista logico

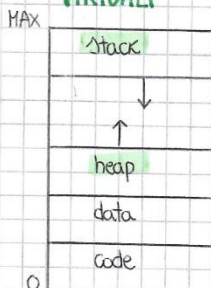


- La memoria fisica allocata al processo è composta da pagine sparse (RAM e backing store)

- Il processo inizia dall'indirizzo logico 0 e si estende in contigua fino a MAX \*

La MMU effettua il MAPPING (logico ↔ fisico)

### \* Spazio degli indirizzi VIRTUALI



STACK: conserva i parametri/variabili locali ai vari sottoprogr. (cerce ad ogni chiamata di funzione)

HEAP: memoria allocata dinamicamente

→ è parte dello spazio degli indirizzi virtuali del processo

- alloco pagine fisiche solo se viene riempito
- se ho "buchi" è definito spazio: utile per il collegamento dinamico di librerie
- facilita la condivisione di file e memoria, mediante condivisione delle pagine fisiche

+ Le librerie di sistema sono condivisibili mediante mappatura delle pagine logiche di più processi sugli stessi frame

## Paginazione su richiesta

Si può caricare una particolare pagina in memoria quando è necessario (quando è richiesta)

- ✓ Minor # operazioni I/O
- ✓ Minor utilizzo della memoria
- ✓ Minor tempo di risposta
- ✓ Più programmi presenti in memoria

=> **Swapping** **PIRO**: solo quando necessario

↓ riferimento alla pagina (richiesta)

CHECK → riferimento non valido → **abort**

MODULO → **page**  
SO

se pagina non è in memoria → **trasferimento**

- trasferisce in memoria solo le pagine che ritiene necessarie (oppure blocca)

=> MMU deve avere altre funzionalità

Quando si fa riferimento ad una pagina:

- Se è presente in memoria: l'esecuzione procede
- Altrimenti:
  - localizzo la pagina su memoria di massa
  - la trasferisco in memoria centrale

Per capirci aggiungo un **bit di validità** nella tabella delle pagine.

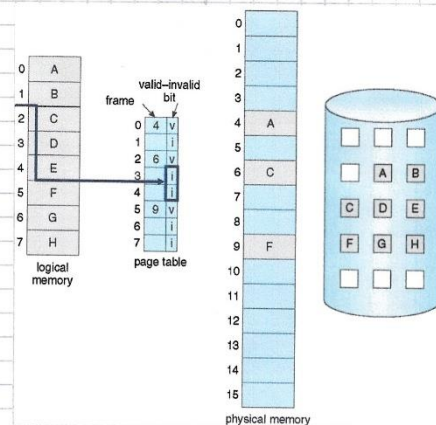
### Bit di validità

V => pagina in memoria

i => pagina non valida / non presente

Valore di default:

=> **page fault** dovrò accedere alla memoria

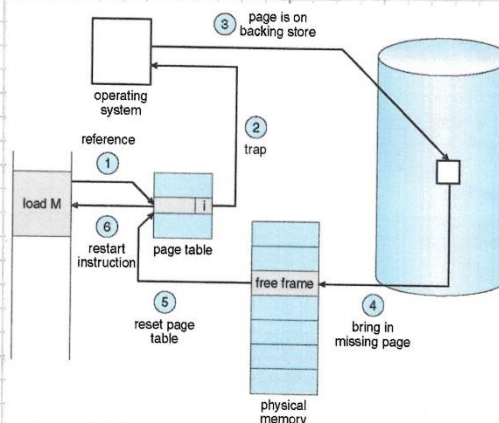


### Page fault: trap al sistema operativo

SO consulta il **PTLR** per capire se è:

- r.f. non valido → **abort**
- \* • pagina non in memoria

1. Seleziona un frame vuoto
2. Sposta la pagina nel frame
3. Aggiorna la tabella delle pagine (i → v) e del frame
4. Riavvia l'istruzione interrotta





**Paginazione Pura** (teorica): è possibile avviare l'esecuzione di un processo senza pagine in memoria

SO carica nel program counter l'indirizzo della prima istruzione del processo  $\Rightarrow$  ho subito page fault

così per tutte le altre

Alcuni programmi, eseguendo una singola istruzione, possono accedere a diverse pagine di memoria provocando più page fault

$C = a + b$  4 page fault (p.e. anche se improbabile, dato che si tende a preservare la località dei riferimenti)

Se il page fault avviene all'atto del salvataggio in memoria del risultato in C occorre...

- Prelevare la pagina desiderata e caricarla in memoria
- Correggere la tabella delle pagine e dei frame
- Riavviare l'istruzione...

NB Un'istruzione interrotta a causa di un page fault deve essere riavviata

Hardware di supporto:

- Tabella delle pagine con bit di validità
- Spazio di swap in memoria secondaria
- Ri-esecuzione di una istruzione che ha causato un page fault

### Lista dei frame liberi

Quando si verifica un page fault, SO deve portare la pagina dalla memoria secondaria nella memoria principale

$\Rightarrow$  si mantiene un elenco di frame liberi

devono essere allocati anche quando stack/heap si espandono

SO assegna frame liberi con la tecnica **Zero-fill-on-demand**: il contenuto dei frame è azzerato prima dell'allocazione (per una questione di sicurezza)

- All'avvio del sistema, tutta la memoria disponibile è inserita nell'elenco dei frame liberi, dalla quale attinge la paginazione su richiesta

hanno effetto  
riferimento

Tempo di servizio:

- Servizio dell'segnale di eccezione di page fault
- Lettura della pagina da disco
- Riavvio del processo

### Prestazioni:

Page Fault Rate  $0 \leq p \leq 1$   $\rightarrow$  nessun riferimento a page fault

- 1) Trap al SO
- 2) Salvataggio dei registri e dello stato del processo
- 3) Verifica che l'interruzione sia dovuta o meno ad un page fault
- 4) Controllo della correttezza del riferimento alla pagina e localizzazione della pagina sulla memoria di massa
- 5) Lettura dalla memoria di massa e trasferimento
  - a) Attesa nella coda al dispositivo di I/O
  - b) (Attesa dovuta a posizionamento e latenza)
  - c) Trasferimento della pagina in un frame libero
- 6) Durante l'attesa: allocazione della CPU ad altro processo utente
- 7) Ricezione dell'interrupt dal dispositivo di I/O (I/O completato)
- 8) Context switch (se è stato eseguito il passo 6)
- 9) Verifica della provenienza dell'interruzione
- 10) Aggiornamento della tabella delle pagine (e dei frame) per segnalare la presenza in memoria della pagina richiesta
- 11) Attesa nella ready queue
- 12) Context switch di accesso alla CPU

$$EAT = (1-p) \cdot t[\text{accesso alla memoria}] + p \cdot (t[\text{servizio PF}] + t[\text{swap out pagina}] + t[\text{swap in pagina}] + t[\text{riavvio processo}])$$

Tempo medio di accesso

ESEMPIO

- t[accesso memoria] : 200 nsec
- t[servizio PF] : 8 m sec = 8000000 nsec

$$EAT = (1-p) \cdot 200 + p(8000000) = 200 + 7999800p \text{ (nsec)}$$

se  $p = 1/1000$  (ogni 1000 accessi a page fault)  $EAT = 8,2 \mu \text{sec}$  10% di rallentamento

NB Se voglio un rallentamento < 10%  
 $\Rightarrow p < 0,00000025$   
 devo avere page fault ogni  
 399.999 accessi alla memoria



## Ottimizzare la paginazione su richiesta

- I/O dallo spazio di swap più rapido che da file system
  - memoria allocata in blocchi più grandi
- Interi processi caricati da subito nell'area di swap
- Processi caricati dal file system, ma con paginazione dalla sola area di swap
  - Swap: pagine non associate a file (Memoria anonima - heap e stack)  
pagine modificate in RAM e non scritte nel file system

## Copy-on-Write

Copy-on-write: processo di copiatura su scrittura, permette ai processi padre-figlio di condividere le stesse pagine in memoria

- Se uno modifica una pagina condivisa, solo in quel caso si crea una copia della pagina  
→ rende l'allocazione più efficiente
- va allocata con una tecnica nota: zeroamento su richiesta (prima di allocare, è cancellato il contenuto)

Vfork(): il padre si sospende in attesa del figlio che usi lo spazio indirizzato del genitore non si utilizza copy-on-write ⇒ molto pericolosa

- Crea shell in UNIX
- Efficiente
- Adatta se il figlio esegue subito una exec()

## Sostituzione delle pagine

Cosa accade quando non ci sono frame liberi?

SOLUZIONE

Sovrallocazione: ho un page fault, devo caricare una nuova pagina ma non ci sono frame liberi

- 1. Sostituzione di pagina: usando un algoritmo che produca il minimo numero di page fault  
Usa un bit di modifica per ridurre il sovraccarico dei trasferimenti

1. Individuo la pagina richiesta su disco

2. Individuo un frame libero:  
a) Se esiste l'uso FINE  
b) Se non esiste, uso un algoritmo per scegliere il frame vittima.  
La vittima è scelta sulla base della tabella delle pagine e dei frame sono modificate

3. Lettura della nuova pagina

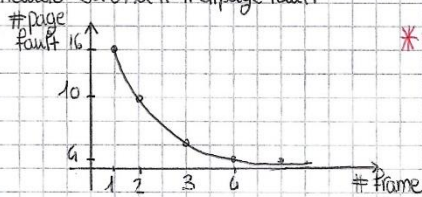
4. Riavvio del processo utente FINE

ALGORITHM

- allocazione dei frame: quanti frame devono essere assegnati a ciascun processo?
- sostituzione delle pagine: quali frame sostituire quando necessario? (minimizzare la freq. di page fault)



Si utilizza, per la valutazione, una stringa di riferimenti a memoria - reference string contando su essa il # di page fault

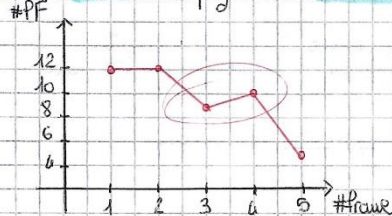


\* Non funziona così per tutti gli algoritmi

- solo numeri di pagina
- accessi multipli alla stessa pagina non provocano page fault

**FIRST IN FIRST OUT** : si sostituisce la pagina che si trova in testa alla coda (FIFO)

\* Anomalia di Belady



**OPTIMO** : sostituisce la pagina che non verrà usata per il periodo di tempo più lungo

↳ non posso conoscere il futuro, quindi vale solo in teoria

**LEAST RECENTLY USED (LRU)** : sostituisce la pagina che non è stata utilizzata per più tempo

↳ Come implementarlo?

- + con contatore : ciascuna pagina ha un contatore a ogni riferimento, copia l'orologio nel contatore + analizza il contatore per scegliere la pagina da rimuovere
- + con stack : hanno stack di numeri di pagina in forma di lista doppiamente concatenata
  - pagina referenziata in cima allo stack

↳ l'insieme delle pagine in memoria per  $n$  frame, è un sottoinsieme dell'insieme delle pagine contenute in memoria per  $m-1$  frame

**Bit di riferimento** : quando riferisco una pagina lo pongo a 1  
rimpiazzo le pagine con bit a 0

oppure registro i bit di rif. a intervalli regolari in un registro di scorrimento

**ALGORITMO A SECONDA CHANCE** (FIFO) alla seconda chance, azzecco il bit di riferimento

accendo 0 → rimpiazzo

1 → bit rif. = 0, lascio la pagina e rimpiazzo la successiva (0)

+ migliorato : (bit riferimento, bit-modifica)

- (0,0) : sostituisco
- (0,1) : salvo su memoria di massa
- (1,0) : verrà acceduta a breve
- (1,1) : "

MB la coda va scandita più volte



**ALGORITMI CON CONTAGGIO**: ho un contatore del # di riferimenti per ciascuna pagina

• **Least Frequently Used (LFU)**: rimpiazza la pagina col valore più basso del contatore

X costoso e scadenti

• **Most Frequently Used (MFU)**

generale • tenere sempre un pool di frame liberi

## locazione dei frame

Ciascun processo richiede un numero minimo di pagine fisiche (frame)

• definito dall'architettura  
• disponibilità della memoria

+ **Allocazione uniforme**: i frame si distribuiscono equamente fra i processi

+ **Allocazione proporzionale**: alloca frame sulla base delle dimensioni del processo

$S_i$ : DIM PROCESSO  $P_i$  (pagine)  $S = \sum S_i$   $m$  = # complessivo frame

$$a_i = \left( \frac{S_i}{S} \right) \cdot m$$

si usa questo basato sulla priorità invece che sulla dimensione

+ **Sostituzione globale**

SO sceglie il frame dall'insieme di tutti i frame

• un processo non controlla la frequenza di PF

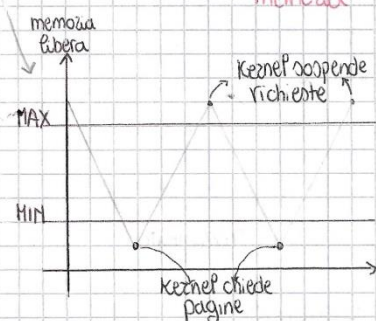
NB Un processo può selezionare un frame di un altro processo

X **Sostituzione locale**

SO sceglie il frame dal relativo insieme di frame allocati

X non è efficiente per la memoria

NO



Così si garantisce sufficiente memoria libera per soddisfare richieste

+ **ROUTINE del REAPER** kernel

• usa una qualsiasi strategia per scegliere vittime

• LRU  
• FIFO

+ in LINUX, memoria libera è bassa, routine **OOM** seleziona un processo da eliminare recuperando la sua memoria

**Page fault principale**: riferimento ad una pagina che non è in memoria

• caricare la pagina e aggiornare la tabella delle pagine

**Page fault secondaria**: un processo non ha una mappatura logica per una pagina (anche se la pagina è in memoria)

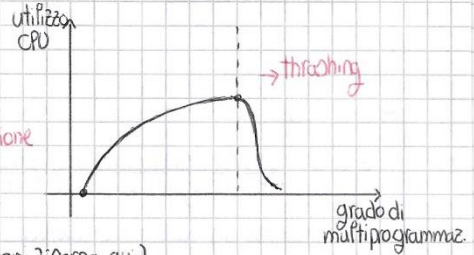
• aggiornare la tabella delle pagine



# Thrashing

**Thrashing:** un processo è costantemente occupato a spostare pagine dal disco alla memoria, e non esegue calcoli all'interno della CPU

1. Memoria fisica libera insufficiente a contenere i frame necessari  $\rightarrow$  generano PF (i processi) rallentandosi
2. SO deduce di aumentare il grado di multiprogrammazione (data che la CPU rimane inattiva)
3. Vengono avviati nuovi processi  $\rightarrow$  altri PF...



**PRESTAZIONI COLASSATE**

- paging eccessivo (spende ogni risorsa qui)
- CPU inutilizzata

Come evitarlo?

X uso sostituzione locale: il processo non può sostituire frame ad altri processi  $\Rightarrow$  **utilizzo peggioramento**

X i processi thrashing rallentano l'attività

$\Rightarrow$  fornire al processo tutte le pagine che fanno parte della sua località attuale

↳ spazio-temporale: quando invoca un sottoprogramma, definisco una nuova località. Termina, il processo lascia la località corrispondente.

↳ riferimenti: istruzioni, variabili locali, ...

## + Modello working-set

- $\Delta$  "finestra scorrevole" # frame di riferimenti alla memoria
- $D = \sum WSS_i$  # totale pagine richieste
- WSS<sub>i</sub>: # pagine referenziate nel più recente  $\Delta$
- $\Delta$  piccolo: non comprende la località
- $\Delta$  grande: più località
- $\Delta \rightarrow \infty$  l'intero programma

**NB**  $D > m \Rightarrow$  Thrashing  $\Rightarrow$  occorre sospendere un processo

emendo una finestra mobile (i riferimenti entrano ed escono)  $\sim$  stabilisco il WS approssimando con un interrupt del timer e un bit di riferimento

## + Frequenza di page-fault stabilisco una $\#$ PF accettabile ed una politica di sostituzione locale

- se  $f \ll$   $\Rightarrow$  processo rilascia frame
- se  $f \gg$   $\Rightarrow$  processo acquiesce frame

## Compressione della memoria

compriamo diversi frame in un singolo frame (non ricorro al paging)

- fondamentale nei mobile (che non hanno swapping)



## Allocazione di memoria al kernel

Il kernel, per allocare la propria memoria, attinge ad un fratello di memoria diverso.

- richiede memoria per strutture dati di dimensione variabile più piccole di un frame devono essere contigue

NB codice e dati del kernel non sono sottoposti a paginazione

### Sistema buddy

usa un segmento di dimensione fissa per allocare memoria  
↳ pagine fisicamente contigue

attraverso un allocatore potenza di 2

- blocchi sono DIM di potenza del 2
- la quantità richiesta è approssimata

fram. interna

- quando chiedo meno memoria del segmento => questo viene diviso in due identici  
↓  
continua fino ad ottenere il segmento minimale
- quando rilascio la memoria, congiungo rapidamente segmenti adiacenti (coalescing)

### Allocazione a lastra

Una lastra è composta da frame fisicamente contigue

↳ una cache è una o più lastre contigue => ha una cache per ciascuna categoria di strutture dati del kernel

Stati di una lastra:

- Piena → oggetti usati
- Vuota → "liberi"
- Parzialmente occupata

- PCB
- semafori...

tutti gli oggetti contenuti sono "liberi"  
↓  
successo mancato  
inuso

- se è piena la lastra → aggiungo lastra

- Riduce la frammentazione interna  
Lo spazio rimanente è usato per strutture più piccole

↳ scelto da LINUX



## Altre considerazioni

• **Prepaging:** si portano in memoria tutte le pagine necessarie al processo prima che vengano referenziate

MB non vengono usate  
ho uno spreco

S: pagine prepaginate  
 $\alpha$ : % di utilizzo  
se  $\alpha \rightarrow 0$  non è conveniente

• **Dimensione delle pagine:**

Criteri:

- Fram. interna
- Località
- DIM tabella delle pagine
- # PF

è sempre una potenza di 2

• **TLB reach** quantità di memoria accessibile via TLB  
(DIM TAB)  $\times$  (DIM pagina)

- il working set di ogni processo dovrebbe essere contenuto nella TLB
- aumentare la dimensione delle pagine  $\Rightarrow$  incremento frammentazione  
↳ pagine di diverse dimensioni

• **Struttura dei programmi:**

a seconda della struttura ho più o meno PF

- separazione di codice e dati garantisce la presenza di pagine mai modificate  $\rightarrow$  facile sostituzione
- il compilatore può impackare procedure che si invocano a vicenda nella stessa pagina

• **Interlock di I/O** blocco delle pagine in memoria



## 11. LA MEMORIA SECONDARIA

### Struttura dei dispositivi di memorizzazione

- I dispositivi collegati possono avere caratteristiche variabili
- SO offre funzioni di accesso con interfacce semplici e uniformi
- ottimizzate PIO (che è il collo di bottiglia)

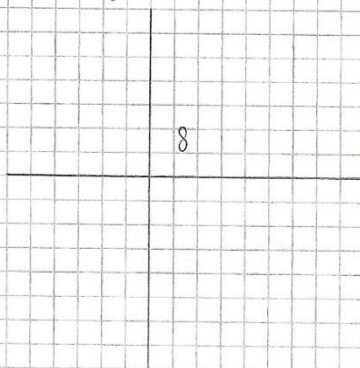
- Penti
- Voluminosi

Memorie di massa

+ Memoria non volatile (NVM)

Dischi  $\left\{ \begin{array}{l} \text{magnetica (HDD)} \\ \text{a stato solido (SSD)} \end{array} \right.$

Dischi magnetici



• mezzodiffuso

- Si utilizza il retro  $\checkmark$  superficie uniforme
- $\checkmark$  rigido
- $\checkmark$  resistente

PIATTO: disco rigido, identificato da un numero

TRACCIA: su ogni piatto, sono gli anelli concentrici

CIUNITO: insieme delle tracce poste alla stessa distanza dal centro

SETTORE: ogni traccia è suddivisa in settori circolari  
di unità minima di trasferimento ciascuno con 1B (4KB)

BLOCCO: insieme dei settori posti nella stessa posizione in tutti i piatti

TESTINA: lettura/scrittura, sono tutte solidali

• Accesso: i dischi ruotano ad una velocità 60/250 giri/s

• velocità di trasferimento: dati vanno dall'unità disco a RAM

• tempo di posizionamento: necessità a spostare il braccio del disco nel cilindro desiderato (SEEK TIME)

+  
necessario affinché il settore desiderato si trovi sotto la testina (LATENZA DI ROTAZIONE)

CRASH DELLA TESTINA  
La testina impatta con la superficie  
→ sostituire l'unità del disco

• Lettura/scrittura: memorizzazione e recupero dell'informazione tramite testina (bobina)  
→ tramite campo magnetico 60-250 HB/cc  
e orientano gli aghi



## Dichiarato solido

costituiti da un controllore e diversi chip di memoria NAND Flash

↓ a preoccupa della  
realizzazione delle operazioni

↓ lo rende più affidabile

+ SO legge e scrive blocchi logici

✓ • impossibilità di cancellazione  
per sovrascrittura

✓ • Contenuto prima cancellato, poi può sovrascrivere  
(per blocchi) (costoso)

ALGORITMI DI  
GESTIONE  
OTTIMIZZATA

⇐ X • la cella si deteriora ad ogni cancellazione  
(Data Writes per Day)

• **Memorie Flash (EEPROM)**: le informazioni sono registrate in un array Floating Gate MOSFET  
transistor → cella di memoria (mantiene carica elettrica a lungo)  
(valore di un bit)

**NAND FLASH**: i blocchi sono un mix di pagine valide/non valide

↳ usa la tabella **FTL** (Flash Translation Layer)  
per tenere traccia

e una **garbage collection** per liberare spazio

- Per cancellare dati non validi:
  1. controller copia tutti i dati validi nelle pagine vuote di un altro blocco
  2. cancellare tutte le celle del blocco da liberare
  3. scrivere nuovi dati

**NB** si tenta di mantenere il livello di usura  
uniforme

## + Memoria volatile

• **DRAM**: può contenere file system temporaneamente

## Connessione

l'unità è connessa tramite **BUS di I/O**

• in generale troppo lenti

ed è eseguito da speciali unità di elaborazione: **Controller** ⇒ gestore del disco



## Mappatura degli indizzi

Le unità a disco sono indirizzate con vettori monodimensionali di **blocchi logici**

L'array di blocchi logici è mappato sequenzialmente nei settori del disco o sulle pagine di un blocco di una NVM

settori/pagine  
blocchi logici



indizzi programmati  
lineari

LBA

(Logical Block Address)

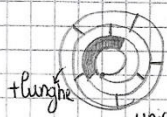
La traduzione diretta è difficile

unità minima di trasferimento

• settori settori danneggiati

## Struttura logica del disco magnetico

+ CLV (Constant Linear Velocity)



densità dei bit per traccia  
uniforme

• CD, DVD

• unicità traccia  
• spirale

+ CAV (Constant Angular Velocity)



velocità di rotazione costante

• densità

• Dischi magnetici

• tracce  
• concentriche

## Scheduling del disco

SO deve rendere efficiente l'uso dell'hardware  $\Rightarrow$  posso intervenire sul tempo di ricerca minimizzandolo

• Ampiezza di banda:  $\frac{\text{\# totale di byte trasferiti}}{\text{tempo (prima richiesta e ultima richiesta)}}$

• Processo che chiede un'operazione I/O  $\Rightarrow$  chiamata a SO

• tipo di operazione  
• indirizzo sul disco / memoria  
• \# byte da trasferire

La richiesta può:

• accettata subito (disco e controller disponibili)

• aggiunta alla coda

prima era responsabile SO  
ora è integrato nel controller

Gli algoritmi di scheduling del disco sono testati sulla coda di richieste per i calcoli (1999)

• **FCFS (First Come First Served)**: X produce troppi spostamenti

• **SCAN** (Algoritmo dell'ascensore)

• **C-SCAN**  $\downarrow$  simile, garantisce un tempo di attesa più uniforme

$\Rightarrow$  **LOOK**  
(arrivano fin dove servono)  
**C-LOOK**

**Scheduling NVM**: non esistono più  $\rightarrow$  FCFS  
mobile

• Sono meno sensibili in caso di accesso sequenziale



## Rilevamento e correzione di errori

- Al rilevamento di un errore il sistema può interrompere l'operazione prima che l'errore venga propagato. La ripetizione si effettua con un \* bit di parità.

↓ forma di checksum : usa l'aritmetica modulare per calcolare, archiviare valori su parole a lunghezza fissa

+ controllo di ridondanza : usa una funzione hash ciclica

=> Correggo con ECC (codice di correzione degli errori) → hard NO  
→ soft SI

## Gestione dell'unità a disco

- + Formattazione di basso livello (o fisica) • si suddivide il disco in settori (letti/scritti dal controllore)



## 12. INTERFACCIA DEL FILE SYSTEM

### Introduzione

- processi devono:
- memorizzare informazioni
  - accedere in modo concorrente e coerente
  - garantire la protezione

=> FILE SYSTEM  
↳ file

è la parte più visibile

- virtualizza i dispositivi di memorizzazione permanente, che sono molto variabili
- ↳ astrarre la complessità di utilizzo con una visione logica e metodi di accesso uniformi

### File system

La gestione è fatta in modo gerarchico

NB l'organizzazione gerarchica non ha alcun legame con la posizione fisica effettiva in memoria di massa

è responsabile della gestione dei file contenuti nella memoria di massa  
(indipendentemente dalle caratteristiche fisiche)

- strutturare dati in file  
↳ sono organizzati in directory
- fornisce all'utente funzioni di alto livello per operare su di essi

**File**: spazi di indirizzi logici continui, tipo di dato astratto

**UTENTE**: è la più piccola porzione di memoria secondaria indirizzabile logicamente

**SO**: i file sono mappati su dispositivi fisici di memorizzazione non volatile

FILE  
↳ DATI  
↳ PROGRAMMI

### Attributi

- nome (non sufficiente per l'identificazione)
- identificativo
- tipo
- locazione
- dimensione
- protezione
- ora, data, identificatore dell'utente

↳ metadati

Le informazioni sono conservate nella struttura di directory

indica struttura logica interna per interpretare correttamente il contenuto

- estensioni
- attributo
- magic number

### Operazioni sui file

+ Creazione: 1. reperisco spazio nel file system  
2. creo il nuovo elemento, registro i metadati

+ Scrittura: 1. chiamata al sistema  
2. reperimento del file nel file system  
3. Occhio, puntatore di scrittura e aggiornamento

+ Lettura: 1. ↳  
2. ↳  
3. ↳ lettura

+ Posizionamento nel file

+ Cancellazione

+ Troncamento: elimino i dati ma mantengo i metadati

+ Lettura/impostazione degli attributi

=> altre operazioni richiedono una combinazione

NB SO mantiene in memoria centrale la tabella dei file aperti

=> System call

open(Fi) | close(Fi) |  
payano su file chiuso  
create() delete()



## File aperti

Tabella di sistema: riferimenti a tutti i file aperti nel sistema

Tabella associata: riferimenti a tutti i file aperti dal processo al processo

- + • Contatore di aperture: # processi che hanno aperto il file
- Locazione del file su disco
- Puntatore coerente alla posizione del file
- Diritti di accesso: SO permettono le operazioni di I/O richieste

## Lock sui file aperti

- shared
- exclusive:
  - obbligatorio: accesso negato se il lock è stato già acquistato (Windows) ✓ SICURO
  - consigliato: decidono i processi ↓ (UNIX) ✗ NON SICURO

## Struttura del file

- Minima: sequenza di byte
  - a record semplice:
  - Completa: file eseguibile e modificabile
- a decidere è:
  - SO oppure
  - l'applicativo che crea il fileultimamente il significato è dato dall'utente

STRUTTURA ↔ TIPO

- Struttura interna: blocchi del disco → record fisico
- record logico
- packing (corrispondenza)

su UNIX: file sono stringhe di byte  
• l'unico tipo riconosciuto da SO è .exe

## Modalità di accesso a file

- Accesso sequenziale: "accesso al nastro" - impossibilità di lettura oltre l'ultima posizione scritta
- Accesso diretto: "accesso al disco"

SEQ  
record a lunghezza variabile

DIR  
record a lunghezza fissa



# Struttura delle directory

risiede su memoria di massa.

È una "tabella dei simboli", traduce i nomi dei file nelle modalità di accesso alle informazioni contenute

- ricerca file
- crea (aggiungo record)
- cancella (rimuovo)

• Elenco dei contenuti della directory `$ ls -l pr*.*`

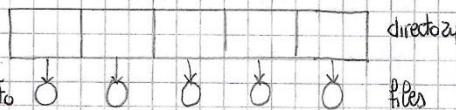
## Come organizzare una directory?

Principi: ✓ efficienza  
✓ nominazione  
✓ Grouping  
i file possono essere molti

struttura logica ≠ struttura fisica  
↓  
organizzazione delle informazioni presentata all'utente  
↓  
come le informazioni sono scritte fisicamente su memoria di massa

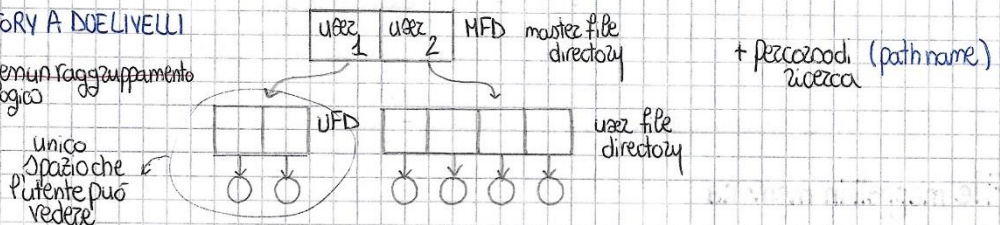
### DIRECTORY MONOLIVELLO

- ✗ non sono nomi uguali
- ✗ nessun raggruppamento logico



### DIRECTORY A DUE LIVELLI

- ✗ nessun raggruppamento logico



### DIRECTORY CON STRUTTURA AD ALBERO

- ✓ ricerca efficiente
- ✓ raggruppamento logico
- ✓ directory coerente

- crea directory
- elimina

### DIRECTORY A GRAFO ACICLICO

- file/directory condivisi
- + Abusing per i nomi uguali
- SOFT LINK
- HARD LINK: duplicazione dell'elemento di directory



## DIRECTORY A GRAFO GENERALE

- aciclico: garantisce semplicità degli algoritmi necessari per attraversarlo
- + ammette link a file (non a sottodirectory) e controllo che non ci siano cicli

GL

## Protezione

Affidabilità: salvaguarda dai danni fisici  $\Rightarrow$  RIDONDANZA

Protezione: da accessi impropri  $\Rightarrow$  CONTROLLO DEGLI ACCESSI

- Lettura
- Scrittura
- Esecuzione
- Append
- Cancellazione
- elenco attributi

L'accesso dipende dall'identità dell'utente

ACL (Access Control List): proprietario - gruppo - universo

## File mappati in memoria

Mappatura file in memoria = accesso alla memoria  $\Rightarrow$  alcuni SO prevedono **mmap()** il file è mappato nello spazio di indirizzi del processo

- associa un blocco del disco a una o più pagine residenti in memoria
- + più processi possono mappare contemporaneamente lo stesso file (condivisione di pagine id in memoria)
- + bisogna aggiornare il contenuto del file su disco periodicamente (o a discesa)

## Struttura del disco

I dispositivi di memorizzazione sono suddivisi in partizioni  
+ utile per limitare la dimensione del file system

↳ volumi  
↳ file system

- general purpose
- speciali