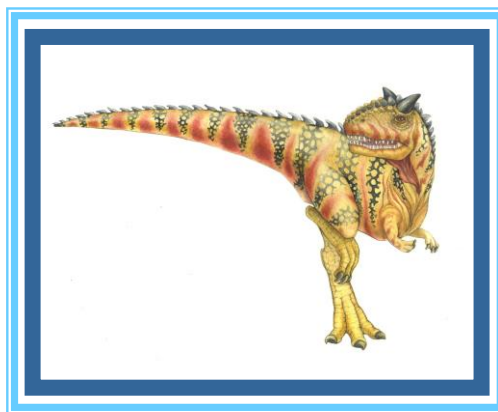


I processi





Obiettivi

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





Sommario

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





Definizione di processo – 1

lista sistemi batch, possono venire uno alla volta

❖ Un SO esegue programmi di varia natura

senza intervento umano

▪ Sistemi batch: **job** *possono di diventare sub. averanno un job scheduling p.esempio*

▪ Sistemi time-sharing: **processi utente** o **task** *dopo che i sys si sono esauriti, con multithreading*

▪ Sistemi che supportano il multithreading: **thread**

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

❖ **Processo**: programma in esecuzione

SO esegue processi

Def. processo:

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

caching per migliorare prestazioni

esecuzione programmi è locale in tempo e spazio. Istanze vicine in tempi vicini

• Stato: stato rep. della CPU

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





Definizione di processo – 2

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





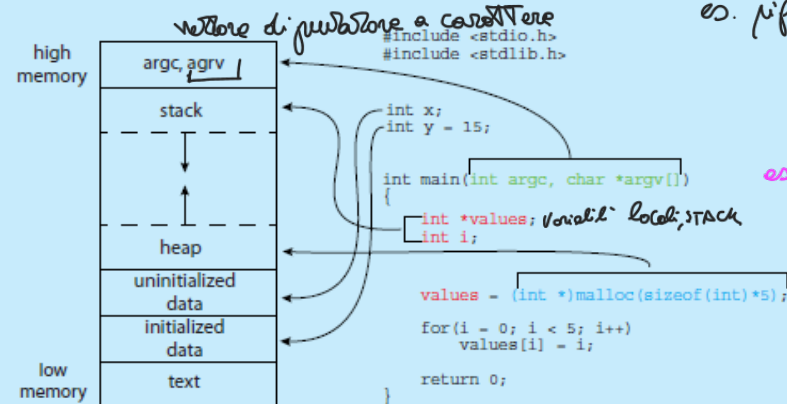
Processo allocato in memoria

Spazio indirizzi processo layout di un programma C in memoria

MEMORY LAYOUT OF A C PROGRAM

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

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



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

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

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





Definizione di processo – 3

posso avere più istanze
in ESECUZIONE dello stesso
Programma

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

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





Stato del processo – 1

Processo Carlo's i 1
Seo stato durante l'esecuzione

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

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

IL NUOVO STATO DIPENDE DA EVENTI SINCRONI O ASINCRONI

- **New** (nuovo): il processo viene creato in RAM. Da accesso alla contesa attiva della CPU

- **Running** (in esecuzione): se ne eseguono le istruzioni ^{ottiene l'utilizzo della CPU}

- **Waiting** (in attesa): il processo è in attesa di un evento ^{es. data I/O o deve terminare un poco prima di gli ritorni dati}

- **Ready** (pronto): il processo è in attesa di essere assegnato ad un processore ^{è nella coda dei processi pronti, non tocca ancora a lui}

- **Terminated** (terminato): il processo ha terminato la propria esecuzione ^{o in maniera corretta (exit / return) restituendo le risorse che deteneva; oppure con un abort. In ogni caso restituisce le risorse}

o in maniera corretta (exit / return) restituendo le risorse che deteneva; oppure con un abort. In ogni caso restituisce le risorse



che come in un processore permette l'esecuzione di un solo processo (parallelismo) in un certo istante di tempo
tempo atteso
valutato nella Ready Queue

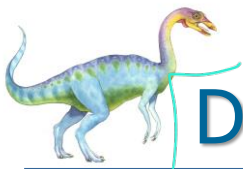
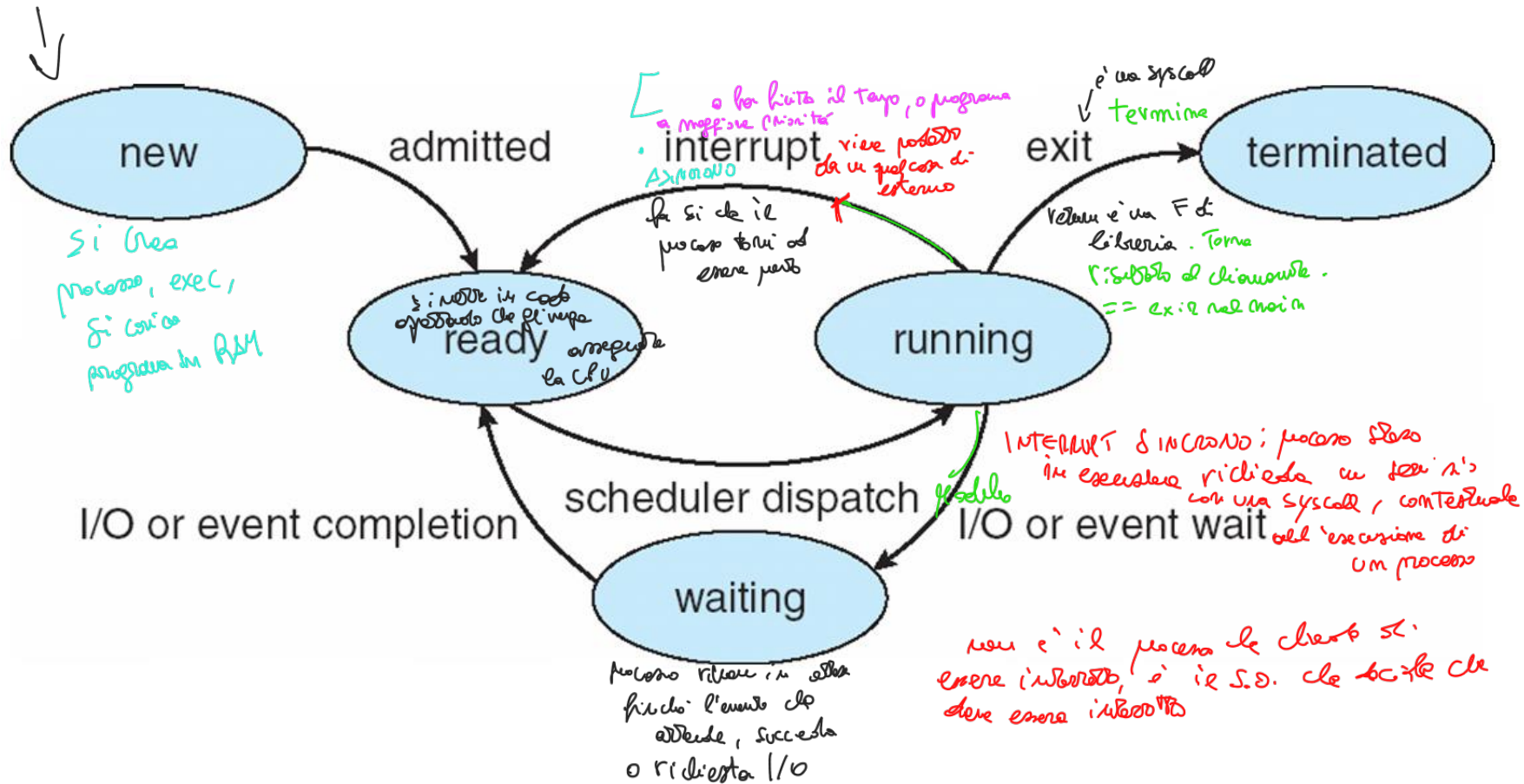


Diagramma di transizione di stato di un processo

Processo creato, gli ho assegnato un scheduler ... Lancio, stoppo click di un process, portavo alla console prima succedeva tra vari programmi
Questa, non viene eseguito subito.





Stato del processo – 2

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

DISPATCHER SCEGLIE PROCESSO DA ESEGUIRE





Stato del processo – 3

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





Stato del processo – 4

❖ Lo stato *terminated* si raggiunge per:

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





Process Control Block – 1

Struttura dati
per gestire processi

- ❖ Ad ogni processo è associata una struttura dati del kernel, il descrittore di processo – **Process Control Block (PCB)** *Per gestire processi*
info che il S.O. deve usare per gestire processo

- ❖ I dati utilizzati per descrivere i processi dipendono dal SO e dalla struttura della macchina fisica *è una struct, un record, dati aggregati con tipi di varia natura*
il PCB varia tra i vari PCB

- ❖ Le informazioni contenute nei PCB sono vitali per il SO; devono essere memorizzate in un'area di memoria accessibile solo dal nucleo del sistema operativo
PCB NEL KERNEL
Solo del Kernel





processi identificati con un numero, PID

Process Control Block – 2

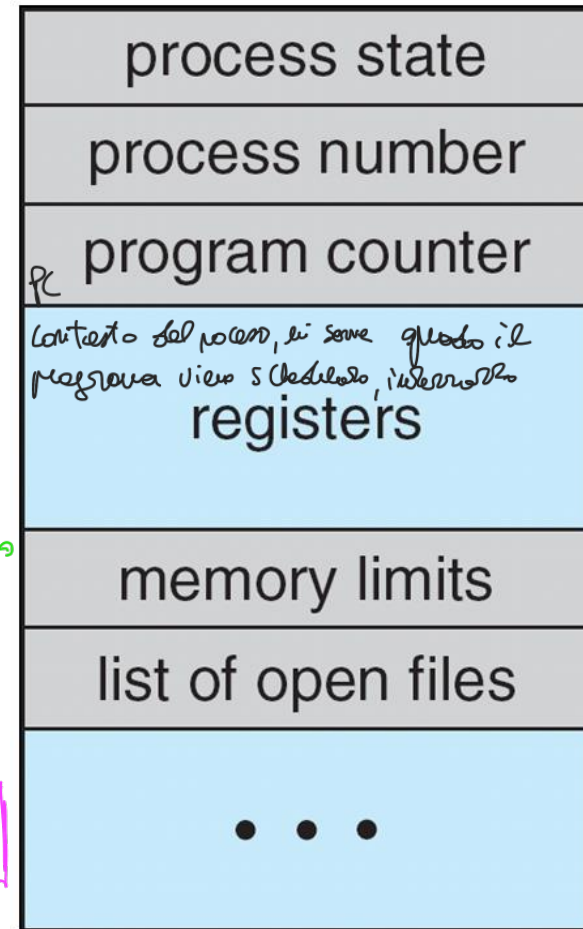
• STATO • PID • CONTESTO (PC e REG)

• LISTA FILE APERTI

• INFO PRIORITA'
IN EG BASE E LIMITE

❖ Informazione associata a ciascun processo:

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



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





Thread – 1

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

es. browser, sottoprocessi

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

❖ Alla base del concetto di **thread** sta la constatazione che la definizione di processo è basata su due aspetti:

- Possesso delle risorse
- Esecuzione *del codice*

Thread viene eseguito
Processo definisce le risorse

❖ I due aspetti sono indipendenti e come tali possono essere gestiti dal SO

- L'elemento che viene eseguito è il thread
- L'elemento che possiede le risorse è il processo

❖ Il termine **multithreading** è utilizzato per descrivere la situazione in cui ad un processo sono associati più thread

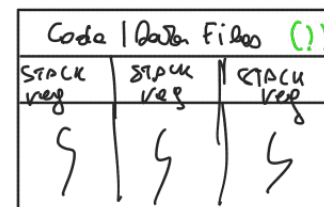
la contesa globale CPU è tra Thread, non process.

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

Tempo context switch,
meno pesante



thread



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





Thread – 2

OGNI THREAD CONDIVIDE
RISORSE PROCESSO MA

MA IL PROPRIO CONTESTO,
PC E STACK

CONTEXT SWITCH PIÙ VELOCE

❖ Tutti i thread condividono le risorse del processo, risiedono nello stesso spazio di indirizzamento ed hanno accesso agli stessi dati

❖ Ad ogni thread è però associato un program counter, uno stato di esecuzione, uno spazio di memoria per le variabili locali, uno stack, un contesto *e i registri (!)*

⇒ **un descrittore**

{ ❖ Le informazioni associate al thread sono poche: le operazioni di cambio di contesto, di creazione e terminazione sono più semplici *e meno pesanti, dunque più veloci*





Processi in LINUX – 1

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

/usr/include/sched.h

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

risorse allo spazio degli indirizzi del processo

Indirizzo degli indirizzi

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

Indirizzo padre e processi figli

Indirizzo e lista di file aperti

Alcuni campi della struttura **task_struct**





Processi in LINUX – 2

READY QUEUE LISTA
DOPPIAMENTE CONCATENATA
DI TCB CON

Coda Processi: Linux è un doppio coda concatenata, mantenendo un puntatore al

PCB del processo attualmente in esecuzione

AL PROCESSO IN ESECUZIONE

- ❖ La ready queue è realizzata mediante una lista doppiamente concatenata di task_struct ed il kernel mantiene un puntatore di nome current al processo attualmente in esecuzione

puntatore al PCB del processo corrente

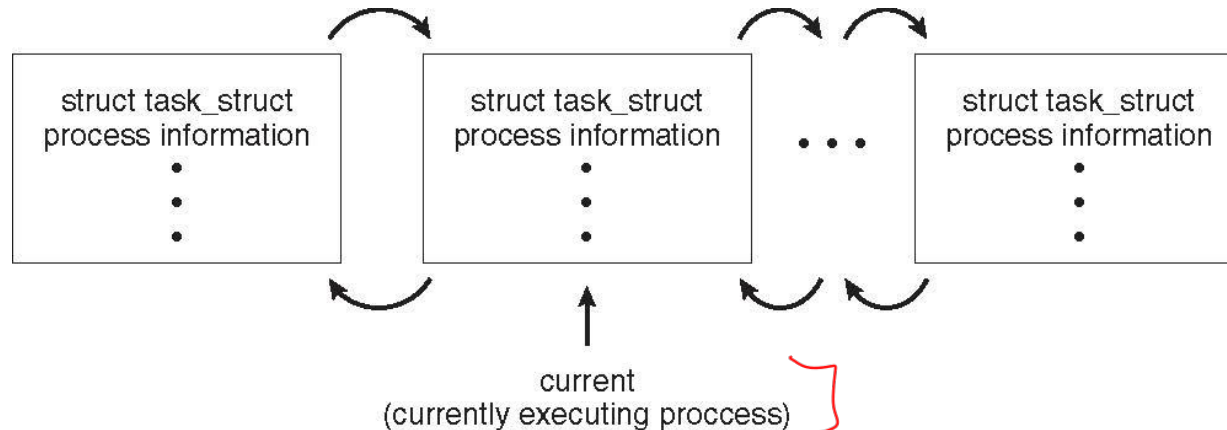
- ❖ Per esempio... se il SO deve cambiare lo stato del processo in esecuzione in nuovo_stato, l'istruzione da eseguire è:

accesso alla
struttura

`current->state = nuovo_stato;`

es. da ready a ready
o in waiting
o Terminated

che solo lo kernel il
quello di tempo





Scheduling dei processi – 1

Molto più come in esecuzione della CPU.

SCHEDULER SI OCCUPA
DELLA SUFFICIENZA PROCESSI
QUALE PUÒ ESSERE IN ESERCIZIO
E
IL SCHEDULER

❖ Meccanismo messo in atto per massimizzare l'utilizzo della CPU, che consiste nel passare rapidamente dall'esecuzione di un processo al successivo, garantendo il time sharing *CONDIVISIONE RISORSE*

❖ È il modulo del SO noto come **CPU scheduler** che esegue questo compito *include Kernel di scheduler, affetto Context switch, salva reg. in processore nel PCB e carica reg. per processore entrante*

❖ Code per lo scheduling dei processi

▪ **Ready queue** (Coda dei processi pronti) — Insieme dei processi pronti ed in attesa di essere eseguiti, che risiedono in memoria centrale

▪ **Code di attesa** — Insieme dei processi in attesa per un dispositivo di I/O o per il verificarsi di un evento

nel corso della loro vita i processi sono anche in altre code.





Scheduling dei processi – 2

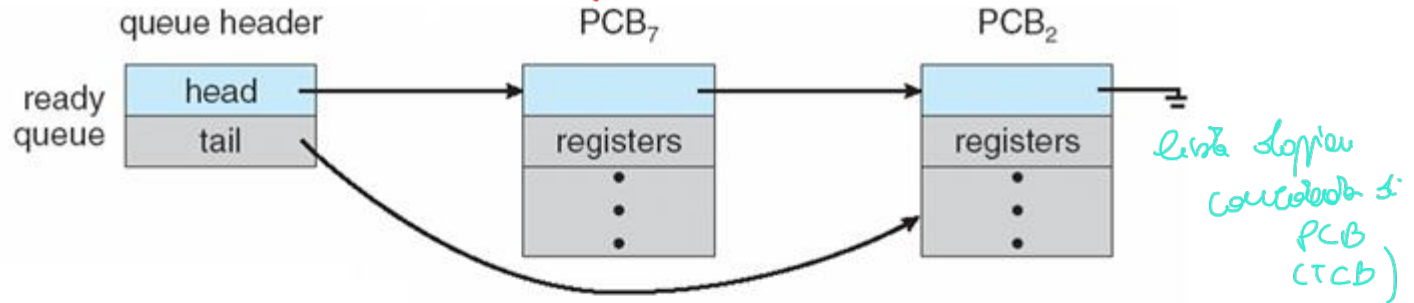
- ❖ I processi migrano fra le varie code
- ❖ Un nuovo processo si colloca inizialmente nella coda dei processi pronti, ^{Ready queue} dove attende fino a quando il dispatcher ^{incluso nello scheduler} non lo seleziona per l'esecuzione
- ❖ Un processo rilascia la CPU (esce dalla propria fase di running)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





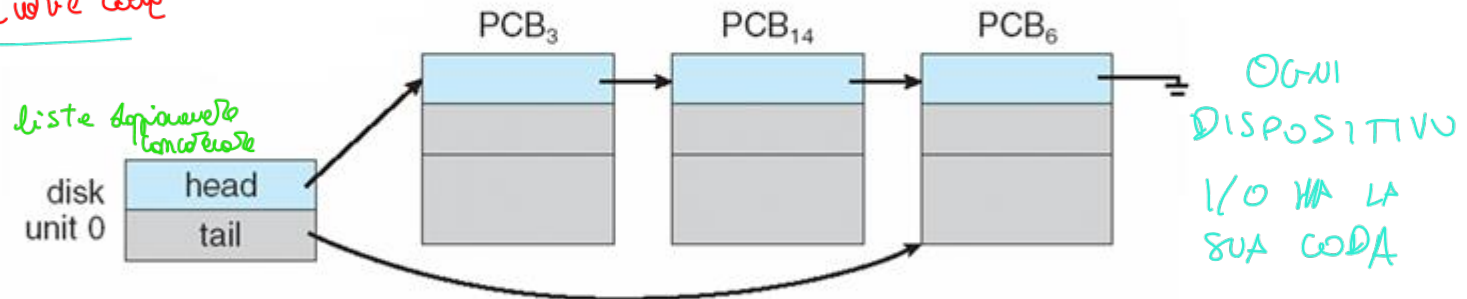
Ready queue e code ai dispositivi di I/O

Processi pronti per essere eseguiti

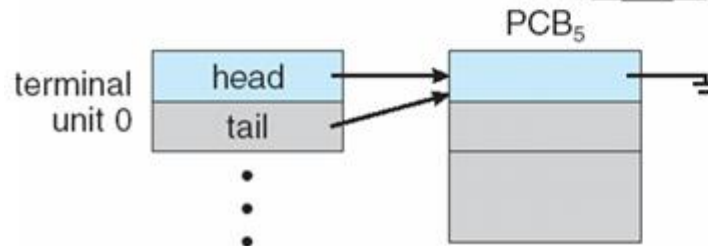


sempre vita i processi migrano tra le varie code

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



wait queues

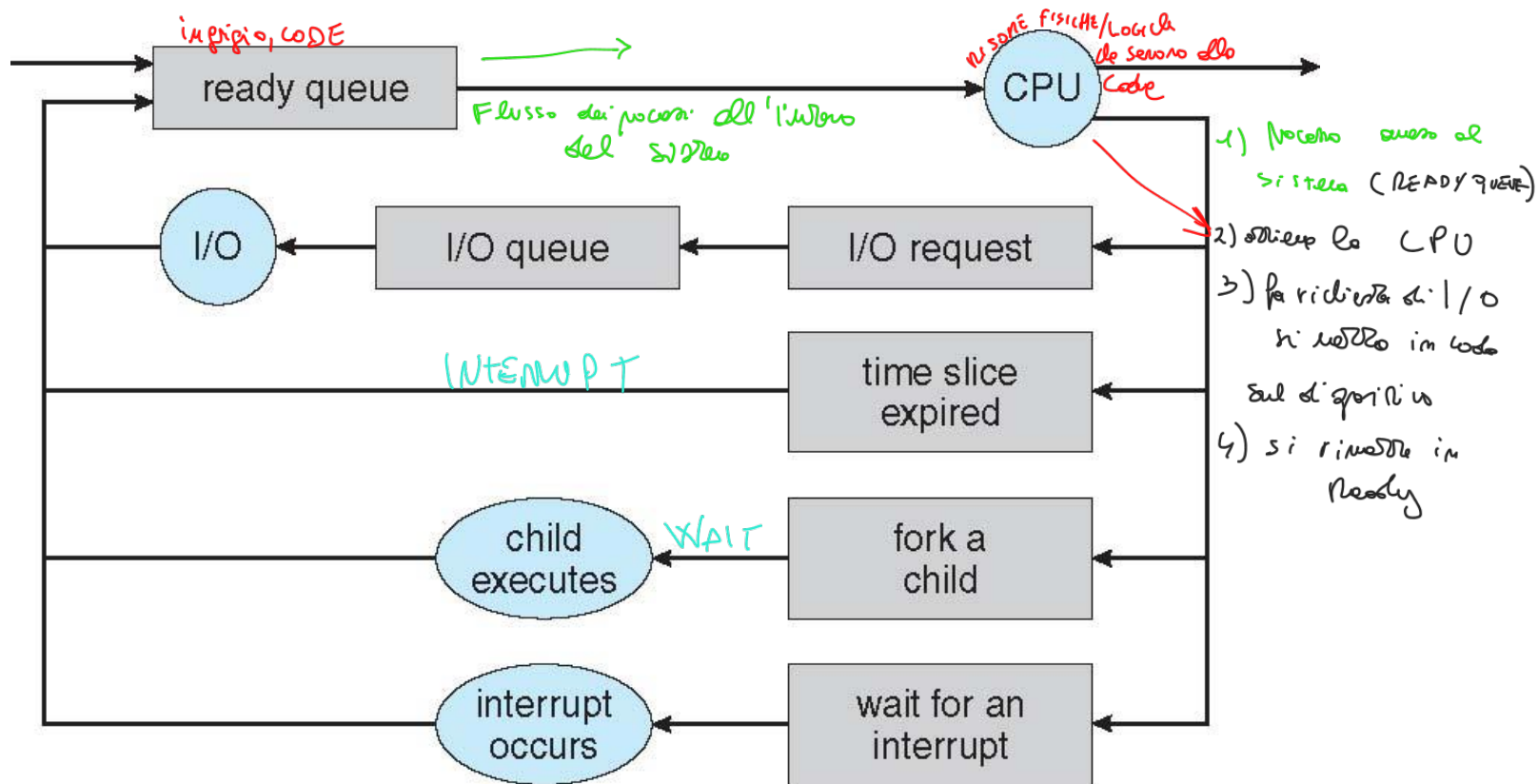


Sistema alla single CPU, un solo processo running, non posso essere più in waiting e/o ready
decisione dello scheduler per farli accedere alla CPU





Diagramma per lo scheduling dei processi



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





Tipi di scheduler – 1

un tempo c'era lo scheduler a tempo finito, lo decideva il massimo process
processo 30 e nei sistemi batch

processi, spin logico assegnato > di quello fisico non erano

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

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

in azione
solo quando
memoria è piena

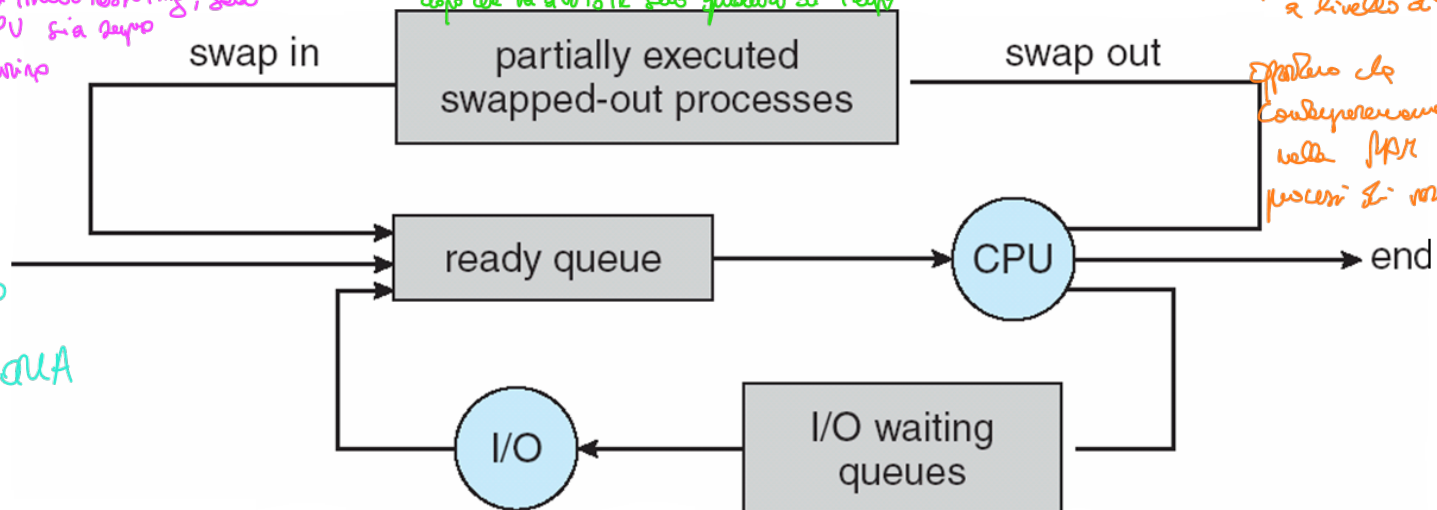
Se aumento grado di multitasking, devo
sicuro che la CPU sia sempre
attiva, ma poco usando
risorse

SWAPPER IN
AZIONE SOLO
QUANDO MEMORIA
E' PIENA

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

questo ora avviene
a livello di pagine.

spesso che
contemporaneamente
nella RAM ci siano
processi di vario genere





Tipi di scheduler – 2

- ❖ Lo scheduler a breve termine viene chiamato molto spesso (intervalli di tempo misurabili in millisecondi)

⇒ [deve essere veloce]

- ❖ Lo scheduler a medio termine controlla il grado di multiprogrammazione e la "miscela" di processi presenti in RAM ad un certo istante

QUANTI PROCESSI PARTECIPANO ALLA CORSA ATTIVA

- ❖ I processi possono essere classificati in:

processi fortemente interattivi

- Processi I/O-bound: impiegano più tempo effettuando I/O rispetto al tempo usato per l'elaborazione (in generale, si hanno molti burst di CPU di breve durata)

- Processi CPU-bound: impiegano più tempo effettuando elaborazioni (in generale, si hanno pochi burst di CPU di lunga durata)

usano molto la CPU. Fanno tanti CPU burst per poco tempo, spesso con sole operazioni senza operazioni di I/O

- ❖ Lo scheduler a medio termine deve mantenere in memoria un mix di processi "equilibrato"

ovvero in RAM un buon mix

devo avere un mix di pezzi due process

Burst: istantanee di sole operazioni senza I/O

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

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





Schedulatore: mediazione che non solo decide quali processi sono in RPH e in che ordine li serve ma anche per usare tutte le risorse

Context switch – 1

↳ tempo sprecato idea: avere più pezzi di reperto

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

CONTESTO NEL

TEMPO CW e' TEMPO PERSO

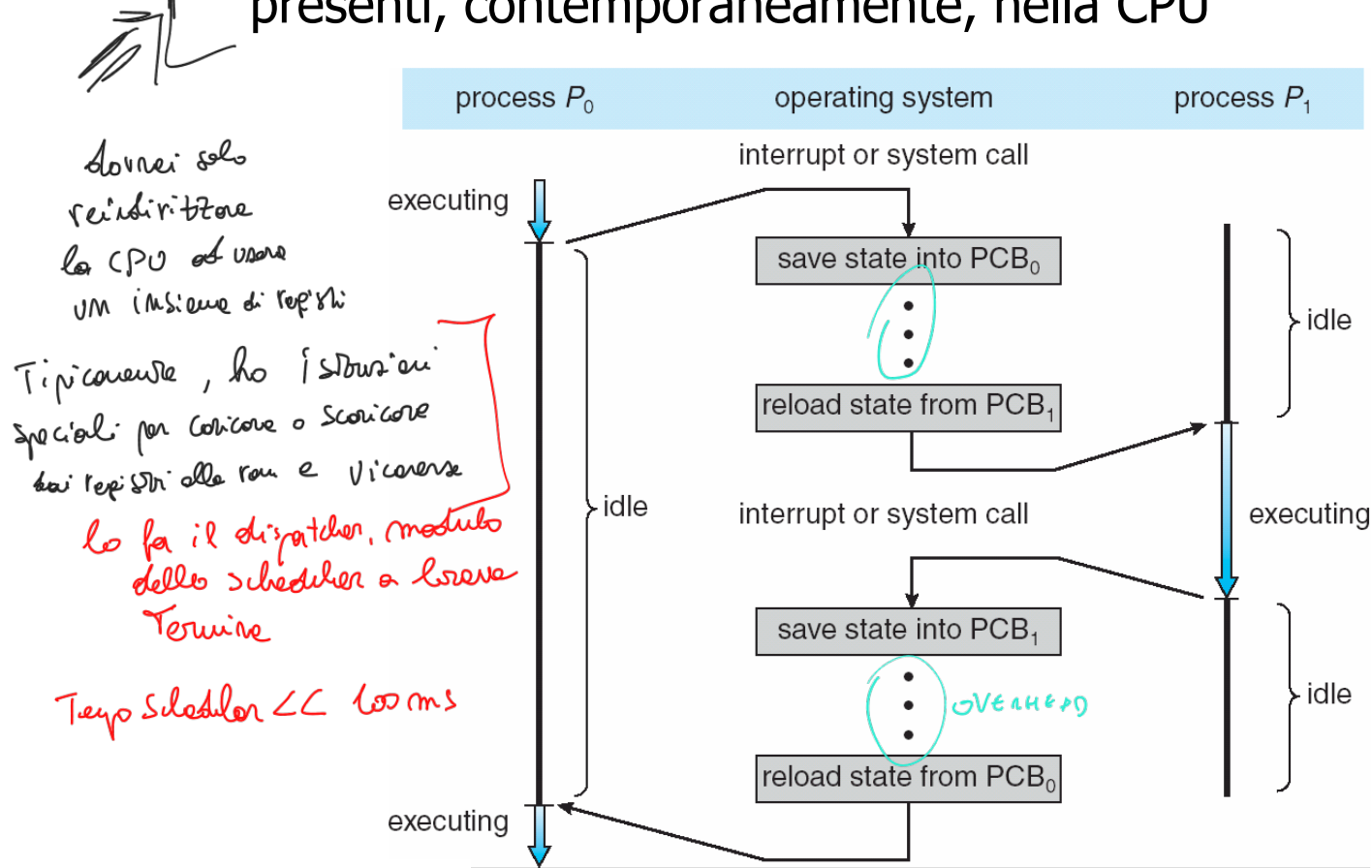
CPU ferma





Context switch – 2

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



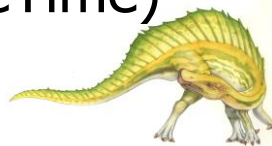
CPU Tabletop
ferma mentre si caricano
e scaricano i registri

OVERHEAD



Multitasking per sistemi mobili – 1

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





Multitasking per sistemi mobili – 2

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





Operazioni sui processi

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

Parallelismo: seguir come in esecuzione un processo

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





Creazione di processi – 1

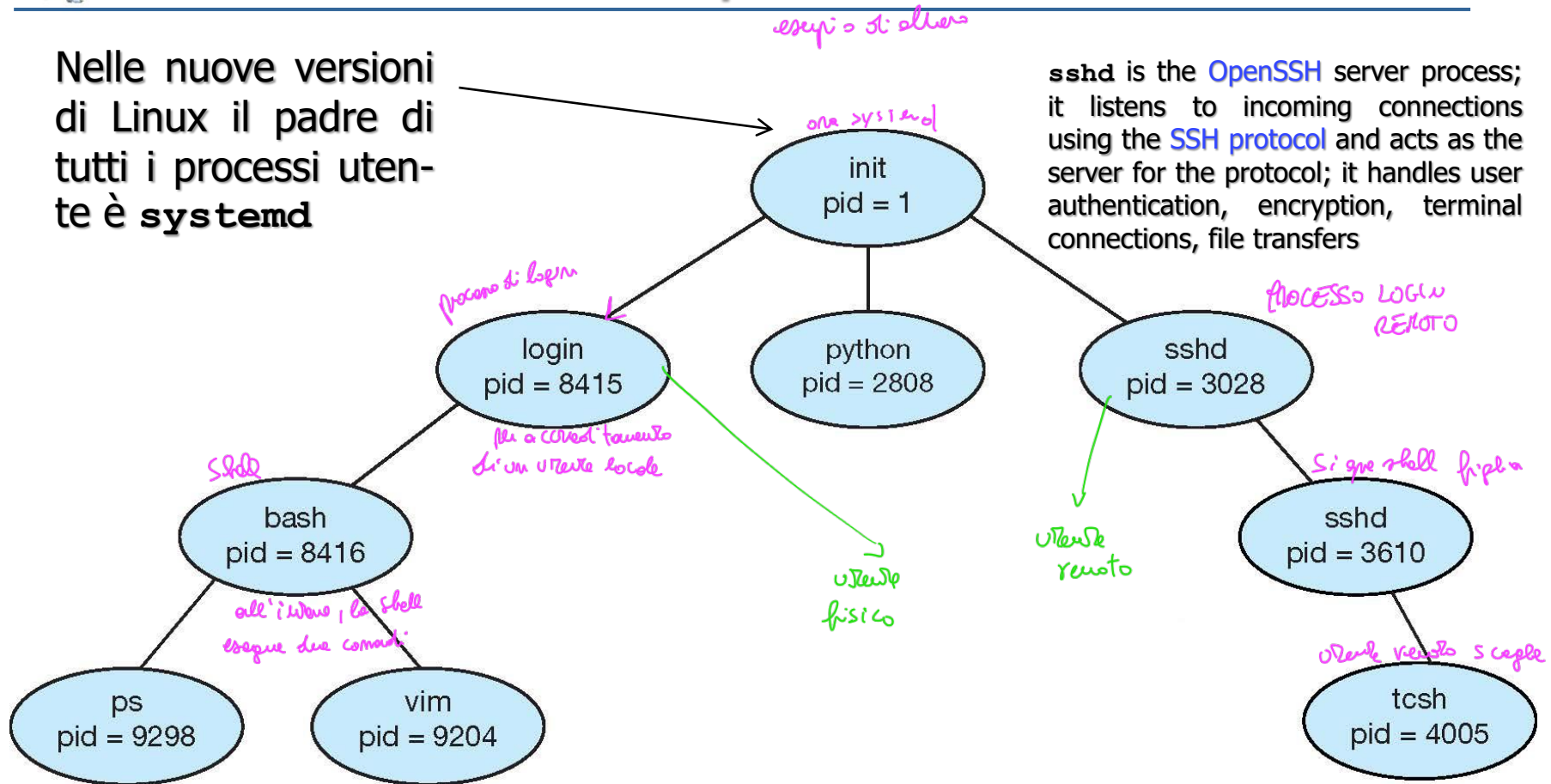
- ❖ Un processo può creare altri processi *(padre)* fino a creare alberi di processi
- ❖ Il **processo padre** crea **processi figli** che, a loro volta, creano altri processi, formando un **albero di processi**
- ❖ I processi vengono identificati all'interno del sistema tramite un **pid** (per process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





Albero dei processi in UNIX

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



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

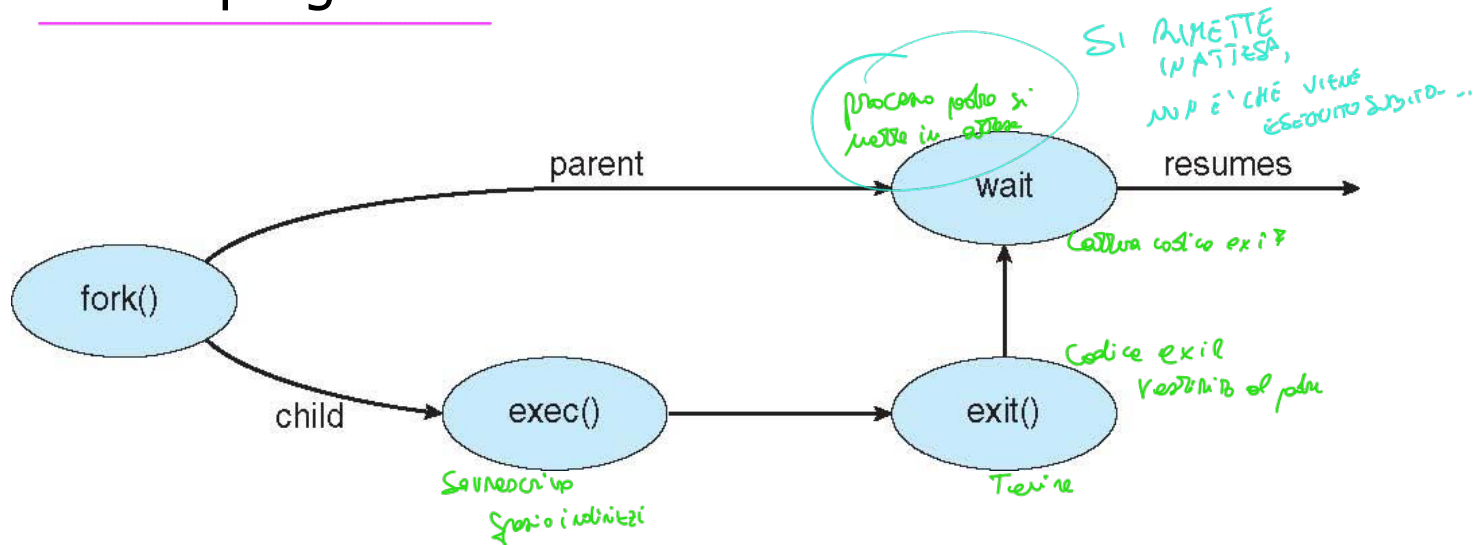




Creazione di processi – 2

❖ In UNIX/Linux...

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





Creazione di processi – 3

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

int main()
{
    pid_t pid;

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

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

    return 0;
}
```





Creazione di processi – 4

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

FIGLIO EREDITA PRIVILEGI,
RISORSE (FILE APERTI)
& SCHEDULING
GENITORE





Creazione di processi – 5

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





Esercizio 1

❖ Si consideri il seguente codice:

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

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



Soluzione 1

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

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



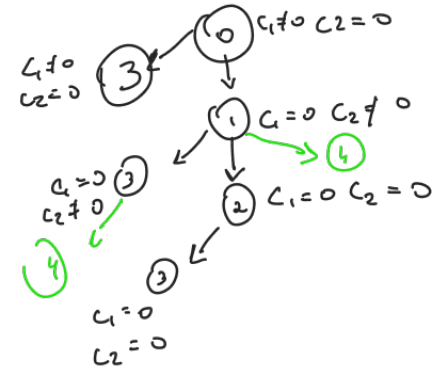


Esercizio 2

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

```

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



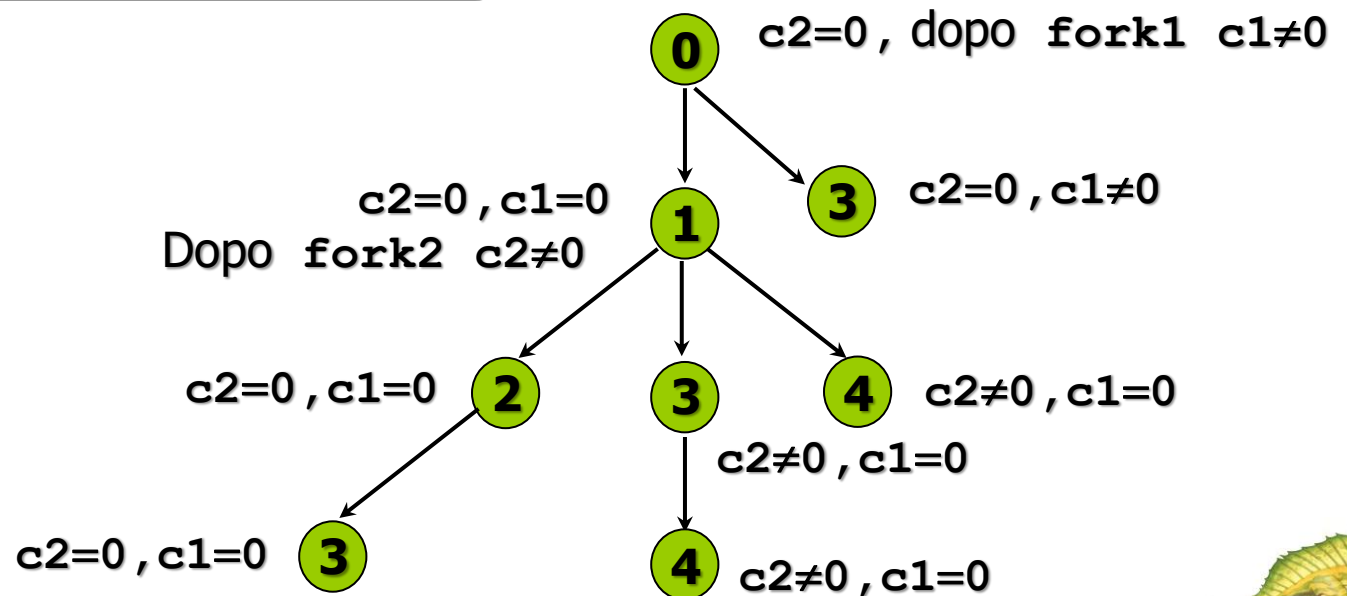
Si supponga che non si verifichino errori. Disegnare un albero che mostra come sono correlati i processi. Nell'albero, ogni processo dovrà essere descritto da un cerchio contenente un numero che rappresenta la fork che ha creato il processo. Il nodo relativo al processo originale sarà etichettato con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

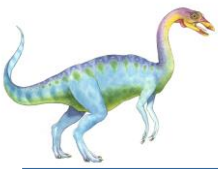




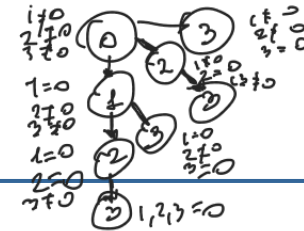
Soluzione 2

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





Esercizio 3



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

```
#include <stdio.h>
```

```
main ( )
```

```
{
```

```
    pid_t f1, f2, f3;
```

```
    f1 = fork();
```

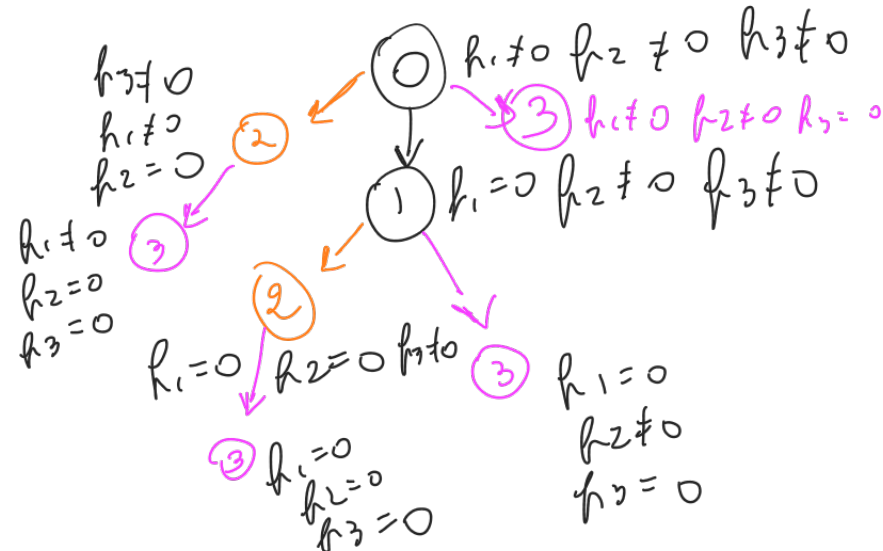
```
    f2 = fork();
```

```
    f3 = fork();
```

```
    printf ("%i %i %i ", (f1>0), (f2>0), (f3>0));
```

```
}
```

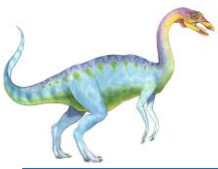
2^3 processi



- ❖ Cosa identificano i numeri stampati?

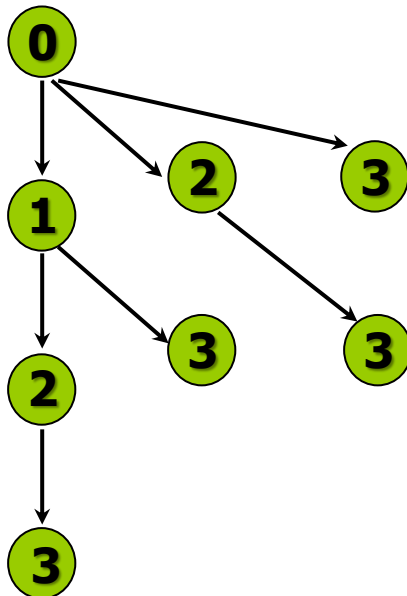
1 0 1
0 0 0
0 1 0
0 0 1
1 0 0
1 1 0
1 1 1
0 1 1





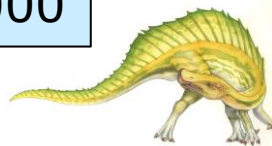
Soluzione 3

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



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

muoi da 0 a 7





Esercizio 4

- ❖ Si consideri il file `test.c`:

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

Padre
0
num = 8
8

Figlio
0
num = 3
3

$n = 5$

0	0	8	3
0	8	0	3
0	3	0	8
0	0	3	8

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

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

\$ test 5

\$ test 0

motivando la risposta data.

$n = 0$, non entra nell'if figlio, num = 0

Padre
0
8

Figlio
0
0

0	0	8	0
0	0	0	8
0	8	0	0

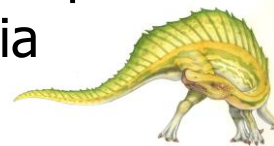




Architettura multiprocesso – Il browser CHROME

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

anche FIREFOX..





Terminazione di processi – 1

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

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





Terminazione di processi – 2

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

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

- ...se il figlio ha ecceduto nell'uso delle risorse ad esso allocate

- ...se il compito assegnato al figlio non è più richiesto

- ...se il padre termina: *non possono restare in esecuzione i figli, ci sarebbe un effetto cascata*

- ▶ Alcuni sistemi operativi non consentono ad un processo figlio di continuare l'esecuzione ⇒ Questo fenomeno è detto terminazione a cascata e viene avviato dal SO

red arrow ▶ Nel caso di UNIX, i processi "orfani" non vengono terminati, ma diventano figli di **init()** (**systemd()**), che è il progenitore di tutti i processi utente *per evitare l'effetto cascata*





Padre può invocare la wait

Terminazione di processi – 3

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

pid = wait(&status);

- ❖ Nel sistema esiste una tabella dei processi

Un processo terminato, il cui padre non ha ancora invocato la **wait()**, è uno **zombie**

Dopo la chiamata alla **wait()** il pid del processo zombie e la relativa voce nella tabella dei processi vengono rilasciati

Se il padre termina senza invocare la **wait()**, il processo è un **orfano**

In Linux, **systemd()** diventa il padre e invoca periodicamente la **wait()** in modo da raccogliere lo stato di uscita del processo, rilasciando il suo pid e la sua entry nella tabella dei processi

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





Terminazione di processi – 4

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





Gerarchia dei processi Android

penale del Android,
quale processo uccidere
prima

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

KILL ↓





IPC

Comunicazione tra processi – 1

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





Comunicazione tra processi – 2

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

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

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



Es di dati inconsistenti

P_1 1) istruzione di alto livello, in assembly

$$X = 3$$

P_1

$$1) X = X + 1$$

P_2

$$X = X - 1$$

I_1 • load r_1, X

I_2 • add $r_1, r_1, 1$

I_3 • Store r_1, X

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

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

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

I_5 • sub $r_1, r_1, 1$

I_6 • Store r_1, X

• I_1 $R_1 = 3$

• I_2 $R_1 = 4$

• I_4 $R_1 = 3$

• I_3 $R_1 = 3$

• I_5 $R_1 = 2$

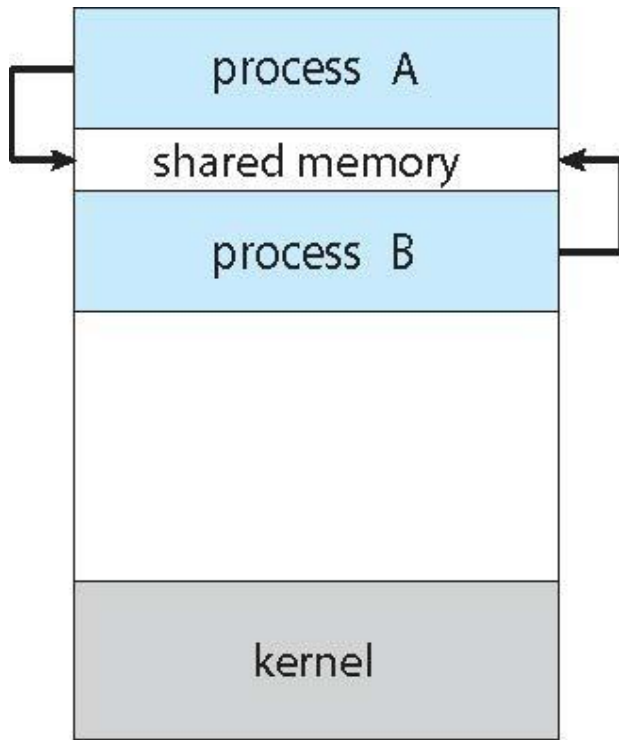
• I_6 $X = 2$

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

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

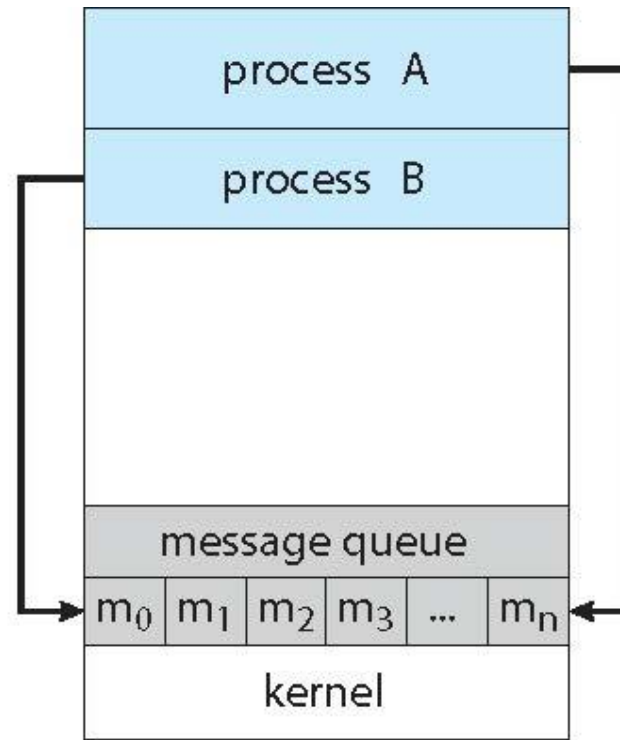


Modelli di comunicazione – 1



(a)

(a) Memoria condivisa



(b)

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

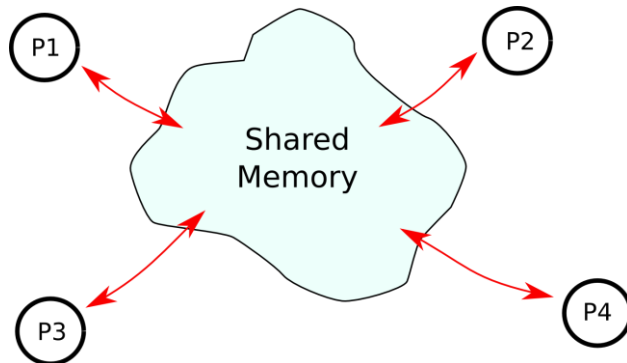




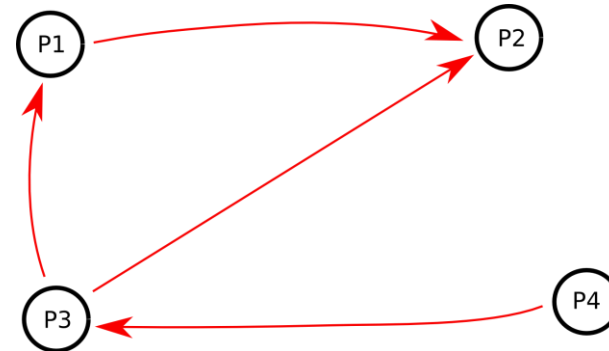
Modelli di comunicazione – 2

come Suenli's

Processi come nodi rete, com. us. mediata



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



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

tra processi STANDARD
MESSAGGI

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

thread condiziona
METODO PROCESSO

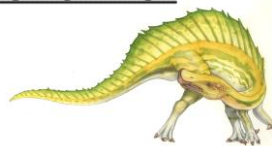
Thread fanno parte dello stesso processo,
e condividono le risorse





Sistemi a memoria condivisa

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





Problema del produttore–consumatore

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





Buffer limitato e memoria condivisa – 1

❖ Dati condivisi

```
#define BUFFER_SIZE 10
```

```
typedef struct {
```

```
... ..
```

```
} elemento;
```

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

```
int in = 0;
```

```
int out = 0;
```

*ogni entry è di tipo elemento
un array di*

in indica la successiva posizione libera nel buffer

out indica la prima posizione piena nel buffer

serve al produttore, se dove mettere il prossimo elemento

serve al consumatore, per l'elemento da consumare

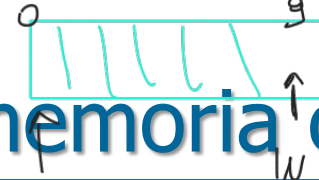
❖ La soluzione ottenuta è corretta, ma consente l'utilizzo di soli **BUFFER_SIZE-1** elementi

*Si spreca sempre una
posizione di memoria*





BUFF



Buffer limitato e memoria condivisa – 2

```

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

```

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

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

fornisco l'accesso fisico
che non ci sia concorrenza
SE DICO CHE BUFF VOTO
PER $in = out$, SPACCO
controllo su $in + 1$ una pos

Processo produttore

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

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

Processo consumatore

serve un contatore
se non voglio sprecare
memoria





Scambio di messaggi – 1

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





Scambio di messaggi – 2

❖ Problemi di implementazione

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

❖ Implementazione del canale di comunicazione:

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

tra coppie di processi che si conoscono o via mailbox

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





IMPLEMENTAZ. CANALE DI COMUNICAZ. TRA PROCESSI

Comunicazione diretta – 1

Stabilisco canale e uso SEND. e RECEIVE

❖ I processi devono "nominare" esplicitamente i loro in-

Solo Teorico
 interlocutori (modalità simmetrica): Comunicos. Diretta SIMMETRICA
 ■ **send(P, messaggio)** – invia un messaggio al processo **P**
 ■ **receive(Q, messaggio)** – riceve un messaggio dal processo **Q**
destinatario deve conoscere l'identità mittente
DIRETTA, e' difficile di implementare, PID non e' sempre lo stesso

PRATICABILI
 ❖ Modalità asimmetrica *chi manda messaggio conosce destinatario, ma il ricevente sa il mittente solo alla ricezione*
 ■ **send(P, messaggio)** – invia un messaggio al processo **P**
 ■ **receive(id, messaggio)** – in id si riporta il nome del processo con cui è avvenuta la comunicazione
COMUNICAZ. Tra padre e figlio, padre ne acquisisce PID
NOTE: PROCESSO viene consegnato sulla base di "memoria" intera, può cambiare

❖ Proprietà del canale di comunicazione:

- I canali vengono stabiliti automaticamente
- Ciascun canale è associato esattamente ad una coppia di processi
Comunicos. diretta tra 2 processi, si deve conoscere IDENTITA'. CANALE tra coppia di processi
- Tra ogni coppia di processi comunicanti esiste esattamente un canale
- Il canale può essere unidirezionale (ogni processo collegato al canale può soltanto trasmettere/ricevere), ma è normalmente bidirezionale
es. da padre -> figlio
 dopo primo messaggio figlio conosce identità padre





Comunicazione diretta – 2

❖ **Vantaggi:** la nominazione (simmetrica/asimmetrica) è molto semplice e permette la comunicazione diretta tra coppie di processi *Mosto intuitivo e semplice per aprire un canale*

❖ **Svantaggi:** È necessario un "accordo" sui nomi dei processi *difficile, al gen. esclusione PID collision per un po' sono*

- Il PID viene assegnato dinamicamente dal sistema e non possiamo prevederne il valore a priori
- Nella pratica è difficile da implementare a meno che i processi non siano in relazione di parentela padre-figlio
- In tal caso, infatti, il padre conosce il PID dei vari figli e può comunicare direttamente con loro utilizzando tale identificativo

*Canale stabilito automaticamente
tra coppie di processi*





Comunicazione indiretta – 1

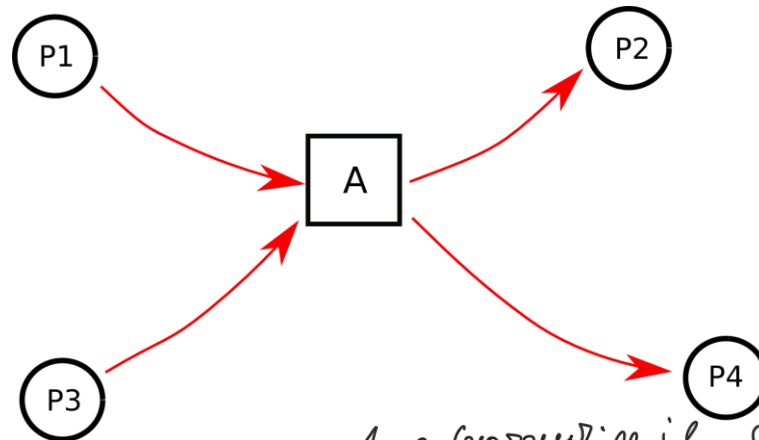
- ❖ Per ovviare ai difetti della nominazione diretta si usa la nominazione indiretta basata su **porte** o **mailbox** FILE MAPPA O IN MEMORIA IN CUI POSSONO LEGGERE O SCRIVERE (PROCESSI)
- ❖ I messaggi vengono inviati/ricevuti a/da mailbox, o porte Mailbox: file in cui alcuni processi scrivono e altri scrivono. A ogni nome suo solo sapere dove andare a leggere / scrivere
 - Ciascuna mailbox è identificata con un id unico
 - I processi possono comunicare solo se condividono una mailbox
- ❖ Proprietà dei canali di comunicazione:
 - Un canale viene stabilito solo se i processi hanno una mailbox in comune
 - Un canale può essere associato a più processi
 - Ogni coppia di processi può condividere più canali di comunicazione più mailbox a comune per ogni processo
 - I canali possono essere unidirezionali o bidirezionali





Comunicazione indiretta – 2

Servizi del SO per



Cosa deve conoscere il S.O.?

❖ Operazioni:

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

❖ Primitive di comunicazione:

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

Processi non si devono conoscere, devono solo condividere una mailbox





Comunicazione indiretta – 3

❖ Condivisione di mailbox

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

❖ Soluzioni:

3 Solrs.

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

Mailbox proprietà di un processo, o del SO





Comunicazione indiretta – 4

- ❖ Una mailbox può appartenere ad un processo utente o al SO
 - Se appartiene ad un processo, cioè fa parte del suo spazio di indirizzi, solo il proprietario può ricevere messaggi, mentre gli altri processi possono solo inviarli *questo dir. possono riceverli*
 - Quando il processo termina, la mailbox scompare
 - Tuttavia... il diritto di proprietà (e di ricezione) *di messaggi* si può passare ad altri processi mediante opportune chiamate al sistema *passato al SO o mezzo di lib. syscall. anche processi possono tornare proprietà ad un altro processo / utente*
 - Una mailbox creata dal SO ha vita autonoma (ma presenta i problemi di smistamento descritti) *esiste no*
so per il problema del ricevimento multipl. chi si assicura lettura?





Sincronizzazione – 1

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

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

SCAMBIO BLOCCANTE:
APPUNTAMENTO TRA LA E TX

APPUNTAMENTO

Rendezvous tra mittente e ricevente

Mittente invia messaggio solo
quando viene ricevuto un ACK
dal DESTINATARIO

OK PUOI
INVIARE

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

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





Sincronizzazione – 2

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





Sincronizzazione – 3

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





Sincronizzazione – 4

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

*risolve concorrenza
la comunic. fluisce
dall'uno o dall'altro
e si sincronizza*

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

*Se bloccato, produttore
non manda
finché consumatore
non riceve
e viceversa*

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





Buffering

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

Bufferizzazione
assente



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

Bufferizzazione
automatica



2. **Capacità limitata** – Il canale può avere al più n messaggi in attesa; se il canale è pieno, il trasmittente deve attendere *nessa sgi fornito via code. potendo destare. se buffer pieno attendo n' di se buffer vuoto*

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

infinite non attende mai





Sistemi per IPC

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

*ad ogni task
si associa 2
porte*





Esempio di sistema IPC: POSIX

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

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

IDENTIFICATIVO FILE *apre un file mappato in memoria* *divisione* *FILE rwrwrw per tutti.*
`shm_fd = shm_open(name, O_CREAT | O_RDWR, 0666);`
le "parent" se tu interno, SEGN. di MEMORIA *lo crea se non c'è* *Ci accede in lettura e in scrittura*

*Come file
mappato in MEM*

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

- Il processo definisce quindi la dimensione del segmento

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

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

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





IPC POSIX: Produttore

```
int main()
{
    const int SIZE = 4096; /* dimensione, in byte, del segmento di memoria condivisa */
    const char STAMPA COSTANTE *name = "OS"; /* nome del segmento di memoria condivisa */ identico MEM
    /* stringa scritta nella memoria condivisa */
    const char *message0= "Studiare ";
    const char *message1= "Sistemi Operativi ";
    const char *message2= "è divertente!";
    int shm_fd; descrittore file, INT /* descrittore del file di memoria condivisa */
    void *ptr; puntatore ad un tipo anonimo non def /* puntatore al segmento di memoria condivisa */

    /* crea il segmento di memoria condivisa */
    shm_fd = shm_open(name, O_CREAT | O_RDWR, 0666); crea segm.

    /* configura la dimensione del segmento di memoria condivisa */
    ftruncate(shm_fd, SIZE); definisce dimensione

    /* mappa in memoria il file che contiene il segmento di memoria condivisa */
    ptr = (char *)mmap(0, SIZE, PROT_READ | PROT_WRITE, MAP_SHARED, shm_fd, 0);
    mmap, mappa contenuto file in memoria, restituisce puntatore a quel segm. di memoria

    /* Scrive sul segmento di memoria condivisa */
    sprintf(ptr, "%s", message0);
    ptr += strlen(message0); si incrementa il puntatore della lunghezza della stringa
    sprintf(ptr, "%s", message1);
    ptr += strlen(message1);
    sprintf(ptr, "%s", message2);

    return 0;
}
```

^{↓ tutte le volte che si aggiornano la loc. di memoria tutti i processi lo vedono}

^{Produttore: crea spazio memoria condivisa e ci scrive sopra}





IPC POSIX: Consumatore

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

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

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

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

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

    return 0;
}
```





Esempio di sistema IPC: POSIX (cont.)

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





Pipe – 1

Pipe, per faro
communicar. Due process

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

Pipe sono Filo di 2 categorie

Si, almeno per due UNNAMED

Varie

No per UNIX
Si per Windows

Pipe NAMED e UNNAMED

è un filo

usato in modo molto più

esempio. Windows usa anche per la comunicazione di processi
3.75 attraverso la RETE

UNIX: Solo per comunicazione
processi nello stesso
calcolatore, che condividono
memoria

Silberschatz, Galvin and Gagne





• Comunic. diretta tra 2 processi
 • UNIDIRETTA con MAILBOX tra
 più processi

Pipe – 2 altro modo di IPC

❖ Pipe convenzionali

- Non possono essere accedute se non dal processo che le ha create come MAILBOX Proprietarie poteva ricevere solo proprietario

- Esistono solo per la durata del tempo di comunicazione

Pipe realizzato
 in modo produttore-consumatore

- Permettono ai processi di comunicare seguendo il paradigma del produttore-consumatore

- Sono unidirezionali: se viene richiesta la comunicazione a doppio senso devono essere utilizzate due pipe Se comunic. bidirezionale, servono 2 pipe

- Nel gergo di Windows, sono dette **pipe anonime**

Pipe

ANONIME, canale di comunic.

UNIDIREZIONALE tra 2 processi





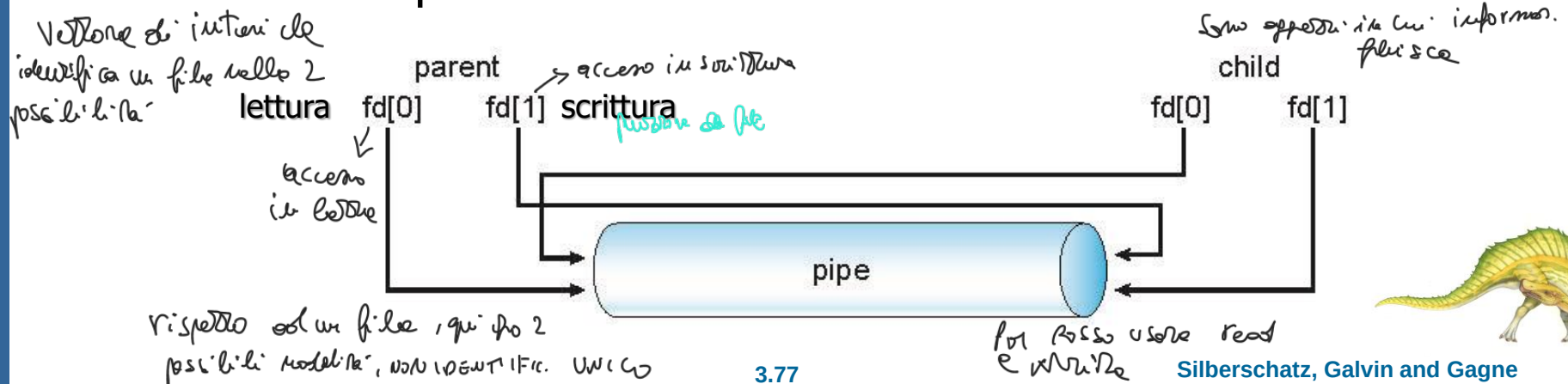
Pipe – 3

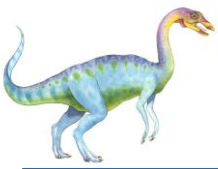
❖ Pipe convenzionali *sono a carattere di sist. diverse per SO*

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

pipe(int fd[])

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





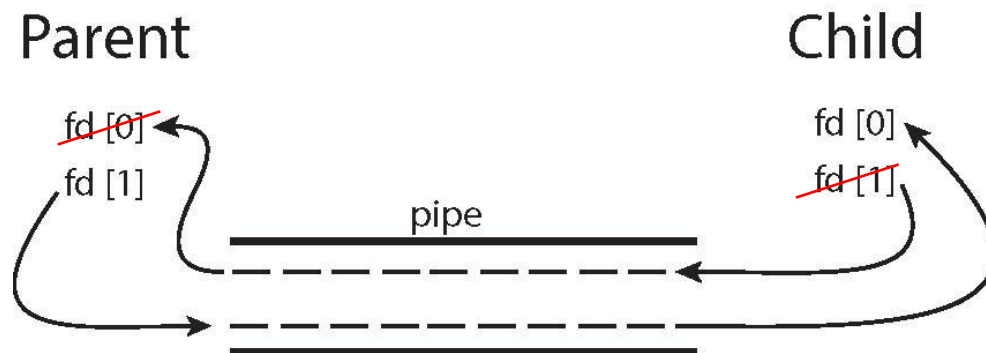
Pipe – 4

❖ Esempio: Pipe convenzionali in UNIX

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

Padre chiude PLUSO LETTURA
FIGLIO chiude PLUSO SCRITTURA

COMUNICAZ. Padre → FIGLIO



se pipe

Fork: figlio condivide risorse padre





Pipe – 5

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

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

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

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

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

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

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

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

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

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

    return 0;
}
```

fd[1] , posizione di
write end della PIPE,
al FILE





Pipe – 6

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

- Possono essere utilizzate da processi qualunque (non necessariamente in relazione padre-figlio) *Processi sulla stessa macchina basta sapere padre e figlio*
- La comunicazione può essere bidirezionale e la pipe continua ad esistere anche quando i processi comunicanti sono terminati *entrambi possono leggere o scrivere*
- Più processi possono usarla per comunicare
- Disponibili sia in Windows che in UNIX
- In UNIX: *senza formattare messaggi o niente!*

*non formattati
sequenza
di byte*

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

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

*NON NECESS. che processo siano
padre - figlio*

*VITA NON DIPENDE dalla durata del
processo*





Pipe – 7

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





Le pipe in pratica – 1

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

- ❖ **Esempio:**

→ ricerca nelle sottodirectory
recursive
\$ ls -lR | grep Jun | more
long. *Sceglie quelle cose di contenere Jun*
→ posto da dove si fa la ricerca nel file che contiene Jun
→ se non trova nella stessa directory

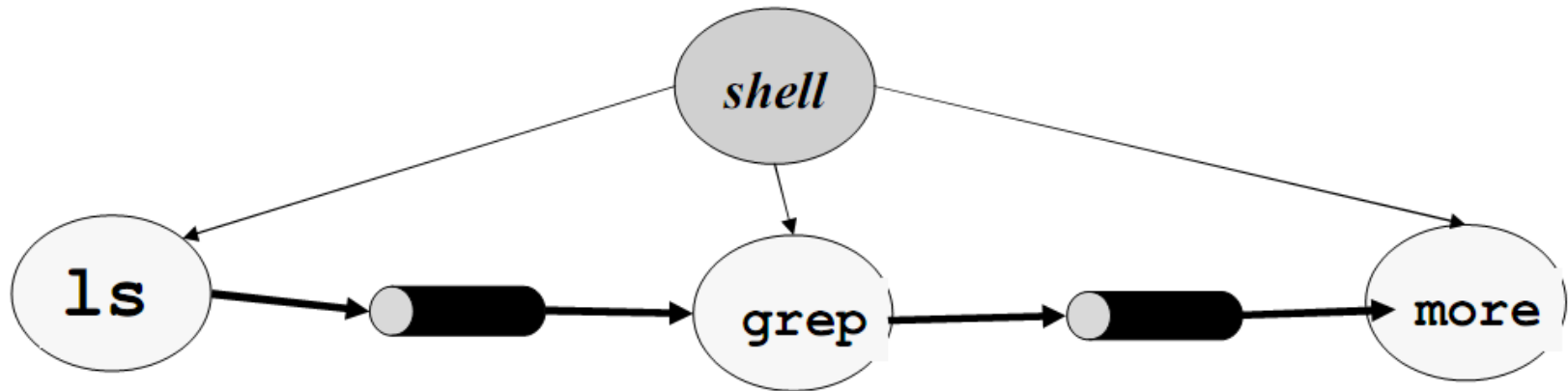
- Si ottiene la lista dei file estesa e con lettura ricorsiva del contenuto delle eventuali sottodirectory
- All'interno di questa lista, si selezionano e si stampano a video le linee contenenti la stringa **Jun**, paginate

more impaginato





Le pipe in pratica – 2



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



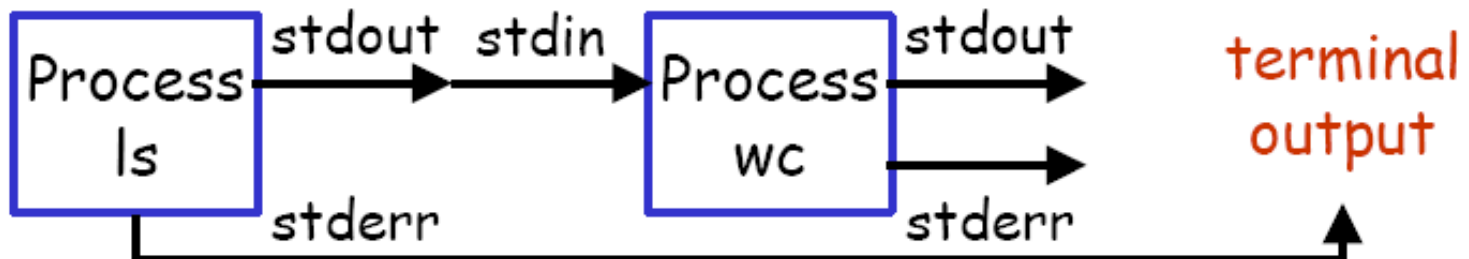


Le pipe in pratica – 3

❖ Esempio:

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

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





Le pipe in pratica – 4

❖ Esempi:

```
$ ls | grep rwxrwxrwx
```

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

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

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

Processi simili

conta i

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

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

→ trova tutti

*-C
-w
-l line*





Comunicazioni in sistemi client-server

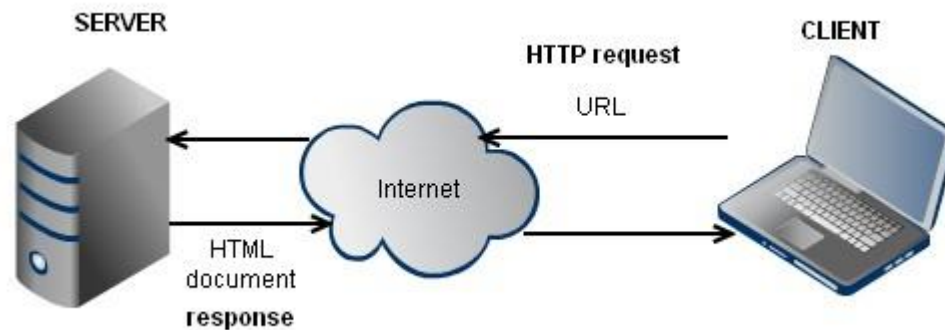
- ❖ Socket
- ❖ Chiamate a procedura remota (*Remote Procedure Call* – RPC)
 - RPC in Android
 - Invocazione di un metodo remoto (*Remote Method Invocation* – RMI, Java)





Socket – 1

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





Socket – 2

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

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

posso fare
qualsiasi tipo
di operazione
da CLI

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

Solo operazioni
file, o put
o get della
Macchina Remota

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

posso farlo anche nel
modo un utente accreditato

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

→ può essere

nomine o
anonimo

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

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





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

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

Socket, canale per l'interazione tra client
e server





Socket – 4

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

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



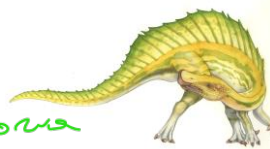


Socket – 5

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

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

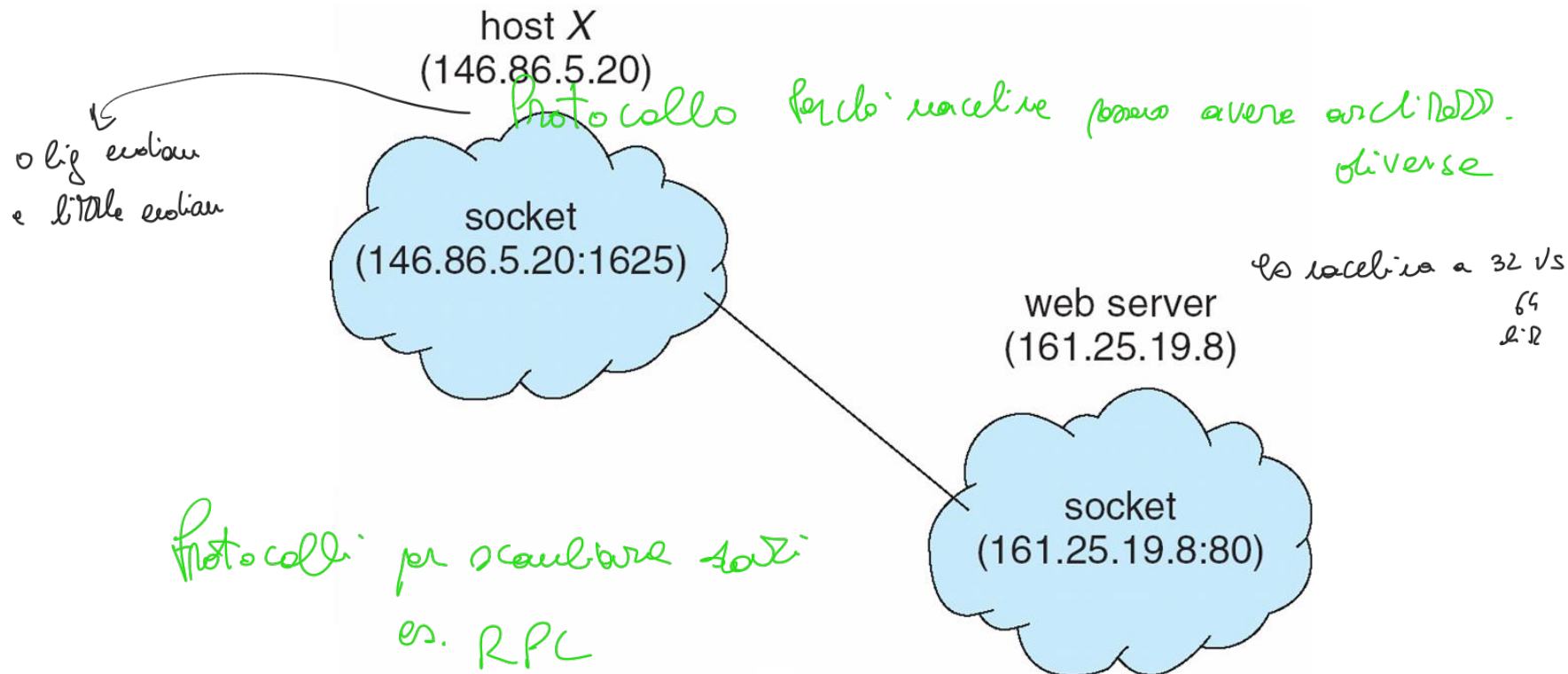
oppure: comunicano via TCP/IP invece di usare messaggio nuovo
3.91
comunicano





Socket – 6

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





Chiamate a procedura remota – 1

Posso invocare una procedura, per sé che è un processo in un altro sistema, ma
 fare che si trova in remoto, un'azione
 inter-processo comunicazione

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

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

STUB: Rappresentazione di codice

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

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

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

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

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

Remote Procedure Call, simile a IPC
 ma macchine possono rappresentare dati in modo diverso



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

impacchetta in
 modo che siano
 comprensibili al server

e impacchetta dal processore
 alla macchina in

Procedura: da processore alla procedura da eseguire sul server

al client



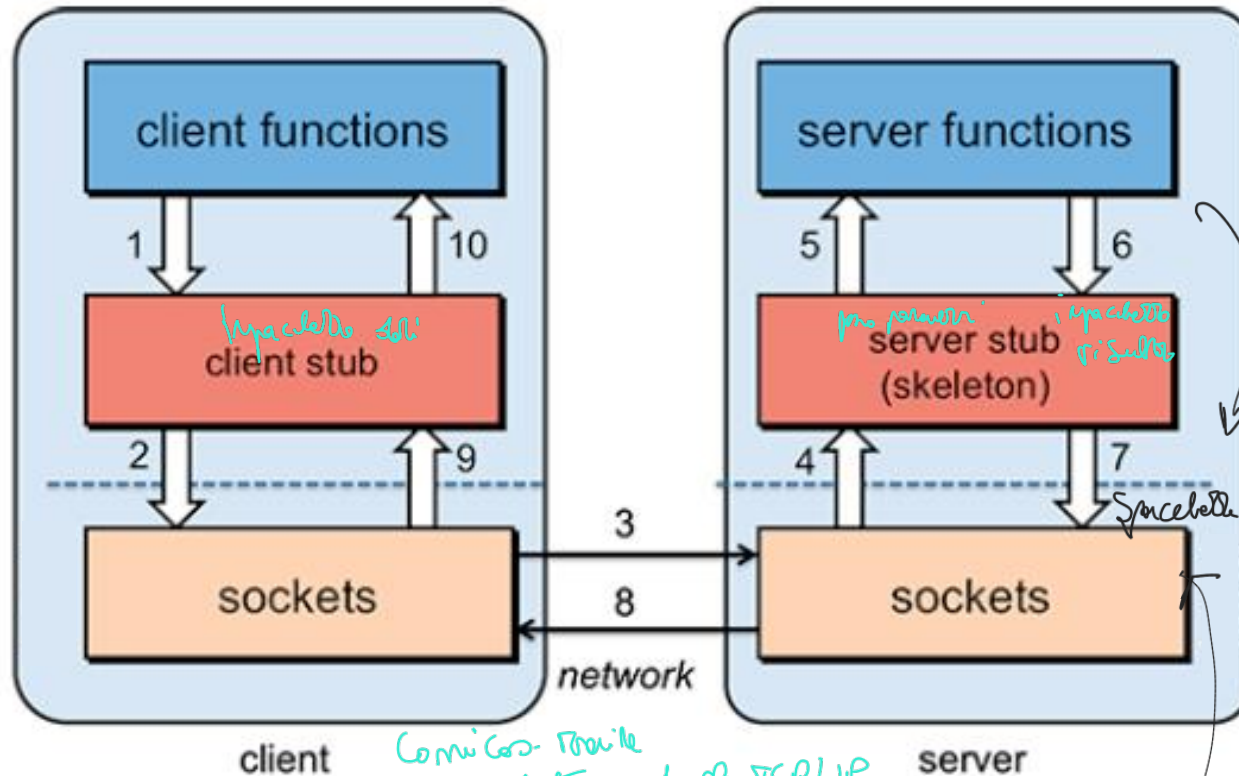
Chiamate a procedura remota – 2

Come avviene?

a loro livello il
meccanismo fisico via

SOCKET

per identificare
macchina e
servizio

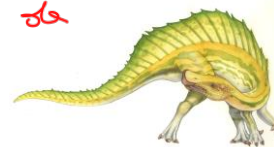


Uno **stub** è una interfaccia di comunicazione che implementa il protocollo RPC e specifica come i messaggi vengono costruiti ed intercambiati

incapsula le info relative alla macchina da

raggiungere sia parametri in formato opportuno

→ fa circolare questo tutto come flusso di byte





Chiamate a procedura remota – 3

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

❖ Problemi

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

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

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

il server remoto è spento...

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

non voglio chiamare 2 volte la stessa procedura
qui non proprio ha un timestamp
del servizio

⇒ Messaggi di richiesta RPC marcati con l'ora di emissione

⇒ Ricevuta al client

sono implementato
Semantica

impone che chiamata e procedura viene eseguita una volta e una volta sola
Server accetta richiesta, manda un ACK per confermare servizio
3.95





Chiamate a procedura remota – 4

server che prende in considerazione la richiesta con timestamp più basso
non si ripete la procedura con il servizio

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

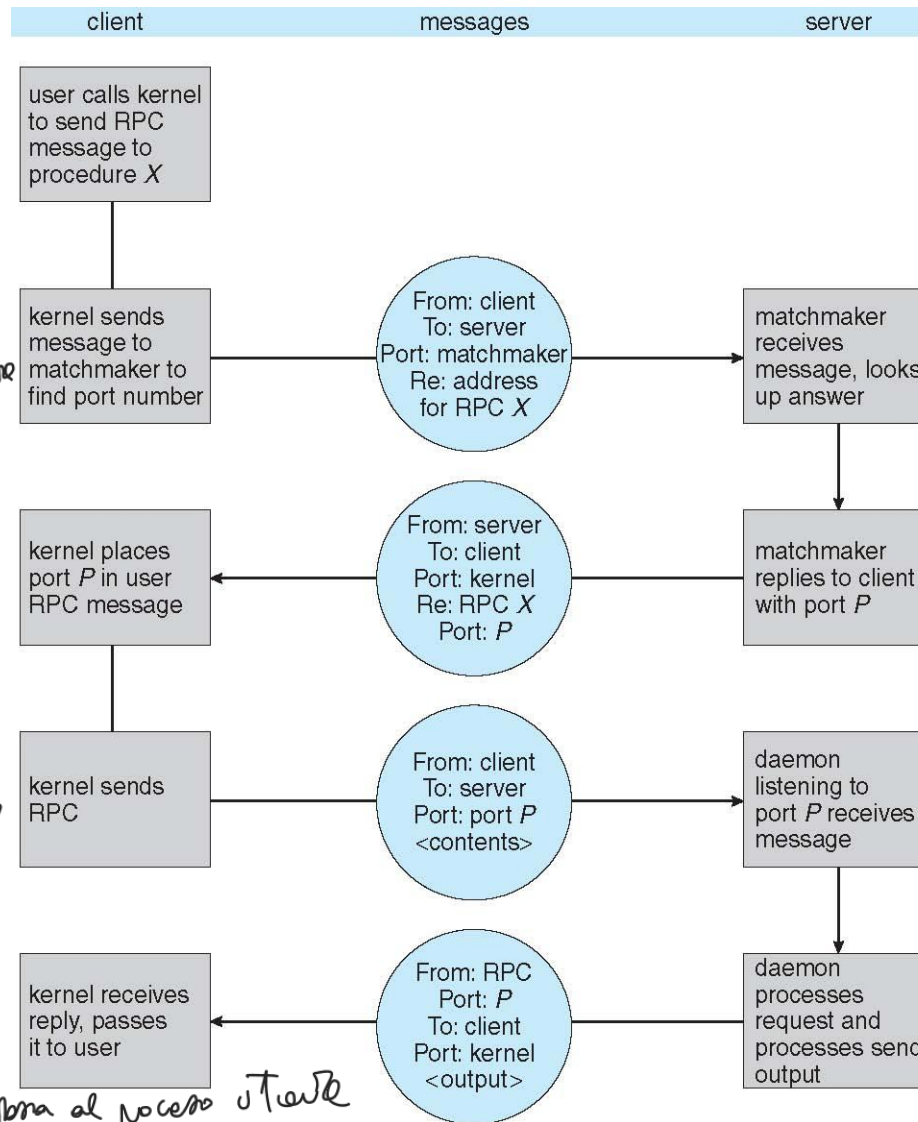
NON
CLIENT può conoscere a priori la porta del servizio del server
per servizi non noti

(
→ stesso client usa una porta nota per mandare una richiesta al server e server risponde con la porta con cui effettuare comunicazione





Chiamate a procedura remota – 5



1) utente chiede al kernel di mandare RPC

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

kernel ha
la porta, può richiedere
servizio

passa al processo utente
della che ha fatto la
richiesta

procedura esegui'le





Chiamate a procedura remota – 6

→ uso da android da RPC

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

mette a disposizione servizi

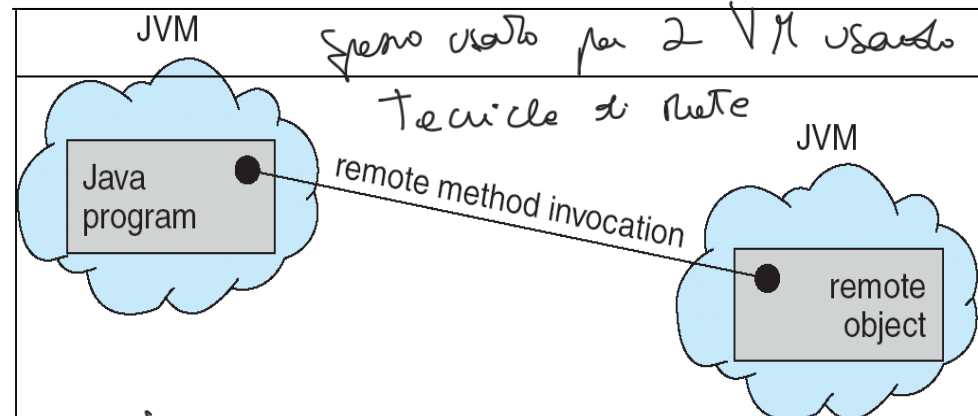
RPC Android per com. tra processi

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

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

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

due macchine virtuali possono comunicare con una rete virtuale usando RPC



Java: non invoca procedure ma Metodi



Esercizio 1 *Per Fork*

❖ Descrivere un possibile output del seguente codice:

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





Esercizio 2

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

$$f_0 = 0$$

$$f_1 = 1$$

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

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



Fine del Capitolo 3

