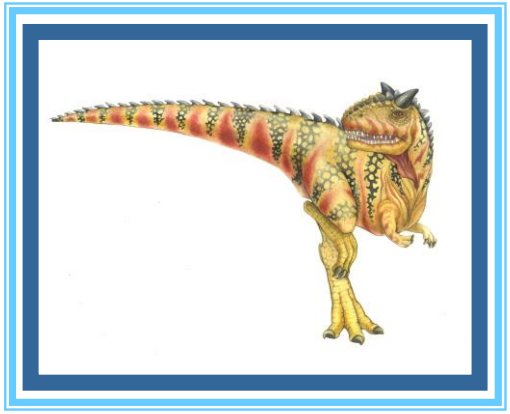
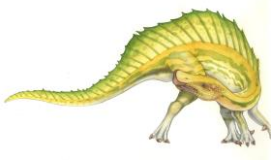
I processi



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

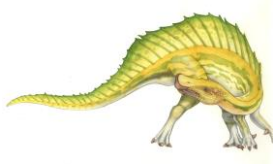
- Definizione di processo
- Scheduling dei processi
- Operazioni sui processi
- Processi cooperanti
- Comunicazione fra processi
- POSIX come esempio di sistema IPC
- Comunicazioni in sistemi client–server

Definizione di processo – 1

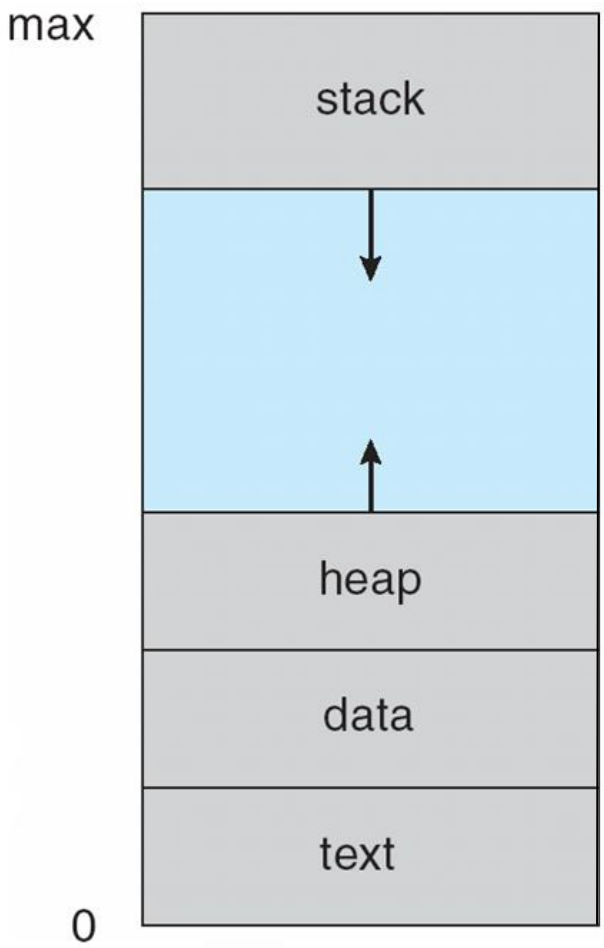
- Un SO esegue programmi di varia natura
  - Sistemi batch: *job*
  - Sistemi time–sharing: *processi utente* o *task*
  - Sistemi che supportano il multithreading: *thread*
- I libri di testo impiegano indifferentemente i termini *job* o *processo* (o *task*)
- Processo
  - Un programma in esecuzione – 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

Definizione di processo – 2

- Un processo include:
  - 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)
  - uno heap (letteralmente "mucchio", grande quantità – memoria dinamicamente allocata durante l'esecuzione del task)
  - il program counter ed il contenuto dei registri della CPU
- Il programma è un'entita "passiva" memorizzata su disco (un file eseguibile), un processo è "attivo"
  - I programmi divengono processi quando vengono caricati nella memoria principale
- 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



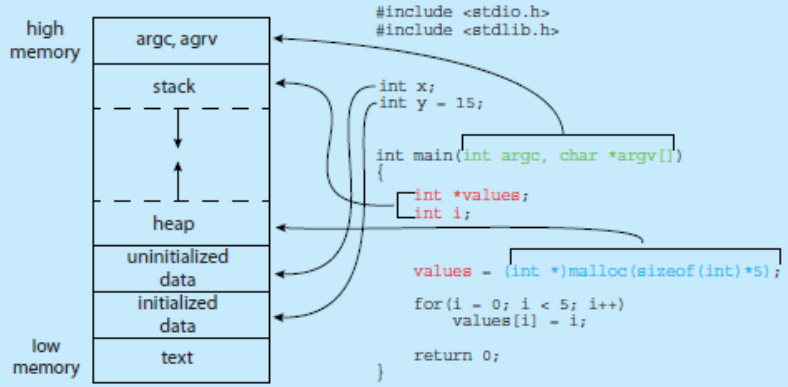
Processo allocato in memoria



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:

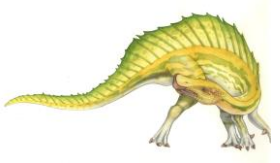
- The global data section is divided into different sections for (a) initialized data and (b) uninitialized data.
- A separate section is provided for the `argc` and `argv` parameters passed to the `main()` function.



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`:

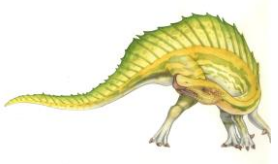
```
text    data    bss     dec     hex     filename
1158    284     8       1450    5aa     memory
```

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

- 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
- Un processo può rappresentare un ambiente di esecuzione di altri processi
  - Java Virtual Machine e programmi scritti in Java



Stato del processo – 1

- Mentre viene eseguito, un processo è soggetto a transizioni di stato, definite in parte dall'attività corrente del processo ed in parte da eventi esterni, asincroni rispetto alla sua esecuzione:
  - *New* (nuovo): il processo viene creato
  - *Running* (in esecuzione): se ne eseguono le istruzioni
  - *Waiting* (in attesa): il processo è in attesa di un evento
  - *Ready* (pronto): il processo è in attesa di essere assegnato ad un processore
  - *Terminated* (terminato): il processo ha terminato la propria esecuzione

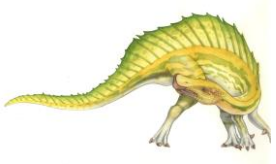
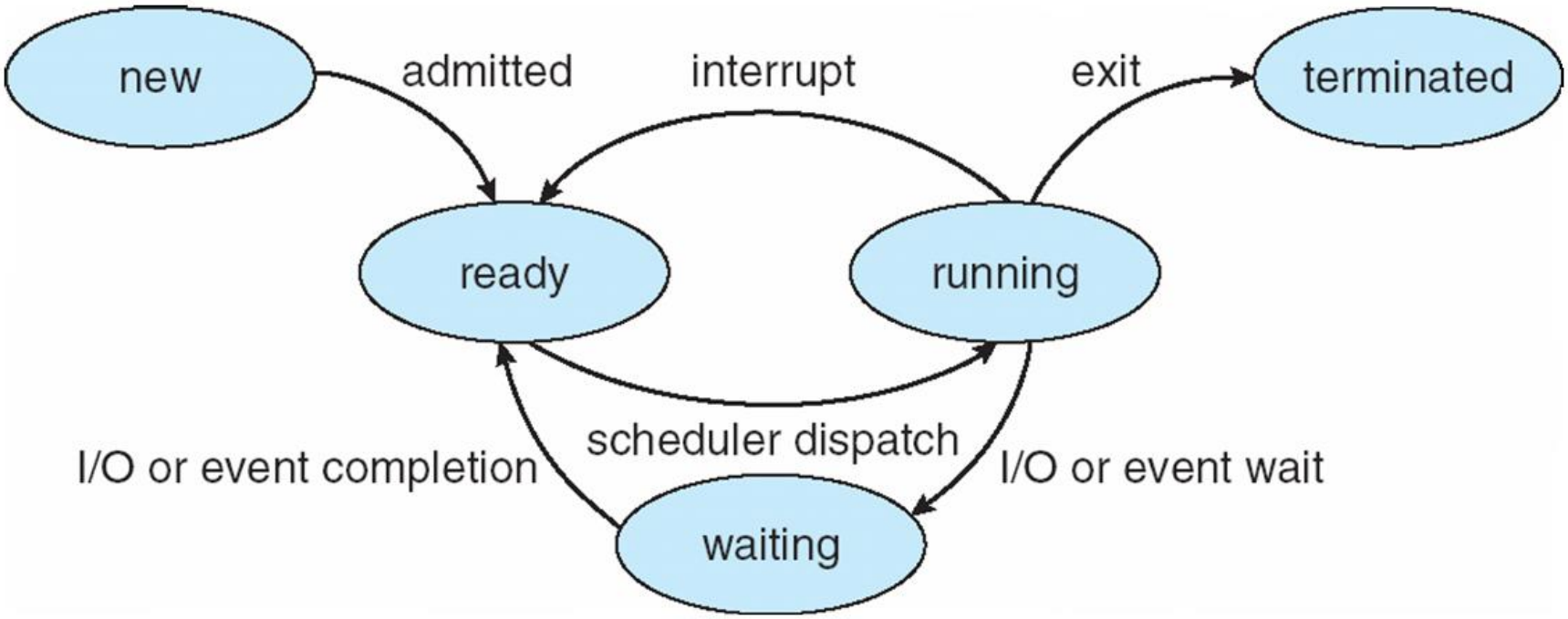
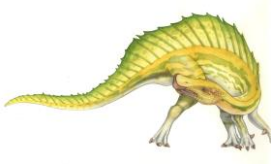


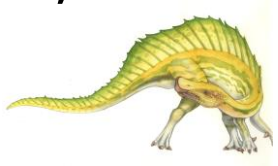
Diagramma di transizione di stato di un processo





Stato del processo – 2

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



Stato del processo – 3

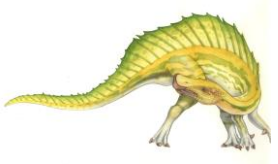
- La transizione da *running* a *ready*, chiamata revoca o prerilascio avviene…
  - …nello scheduling a priorità, quando arriva al sistema un processo con priorità maggiore
  - …nei sistemi a partizione di tempo, per esaurimento del quanto
  - …al verificarsi di un interrupt esterno (asincrono)
- La 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
  - terminazione anomala
    - Uso scorretto delle risorse (superamento dei limiti di memoria, superamento del tempo massimo di utilizzo della CPU, etc.)
    - Esecuzione di istruzioni non consentite o non valide (trap)
  - In entrambi i casi, le risorse riusabili, previamente allocate al processo terminato, vengono riprese in carico dal SO per usi successivi

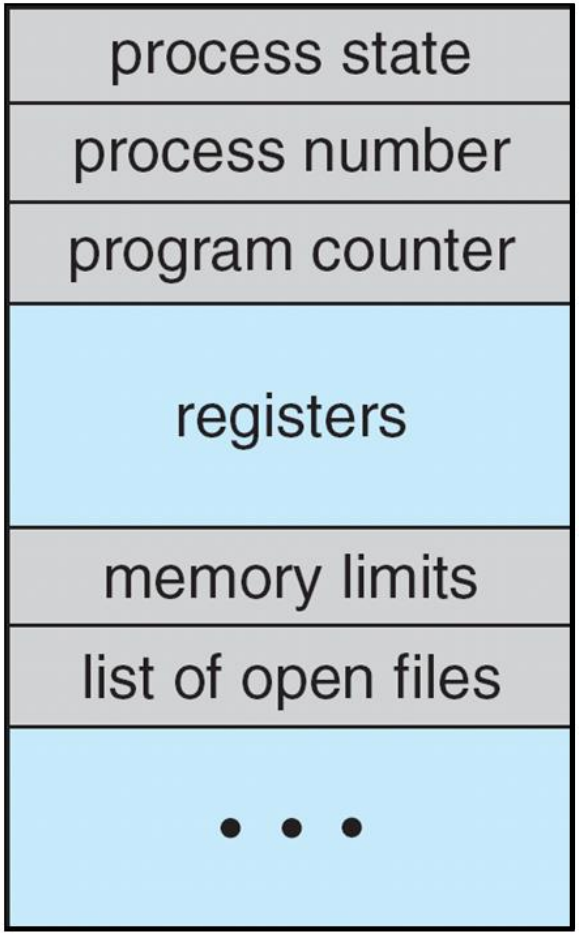
Process Control Block – 1

- Ad ogni processo è associata una struttura dati del kernel, il descrittore di processo – *Process Control Block* (PCB)
- I dati utilizzati per descrivere i processi dipendono dal SO e dalla struttura della macchina fisica
- 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



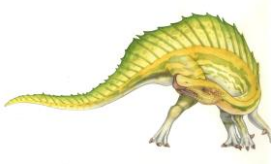
Process Control Block – 2

- Informazione associata a ciascun processo:
  - Stato del processo
  - Nome (numero) del processo
  - Contesto del processo: program counter, registri della CPU (accumulatori, registri indice, stack pointer, registri di controllo)
  - Informazioni sullo scheduling della CPU (priorità, puntatori alle code di scheduling)
  - Informazioni sulla gestione della memoria allocata al processo (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.
  - Informazioni sull'I/O: elenco dispositivi assegnati al processo, file aperti, etc.



Thread – 1

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



Thread – 2

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

Processi in LINUX – 1

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

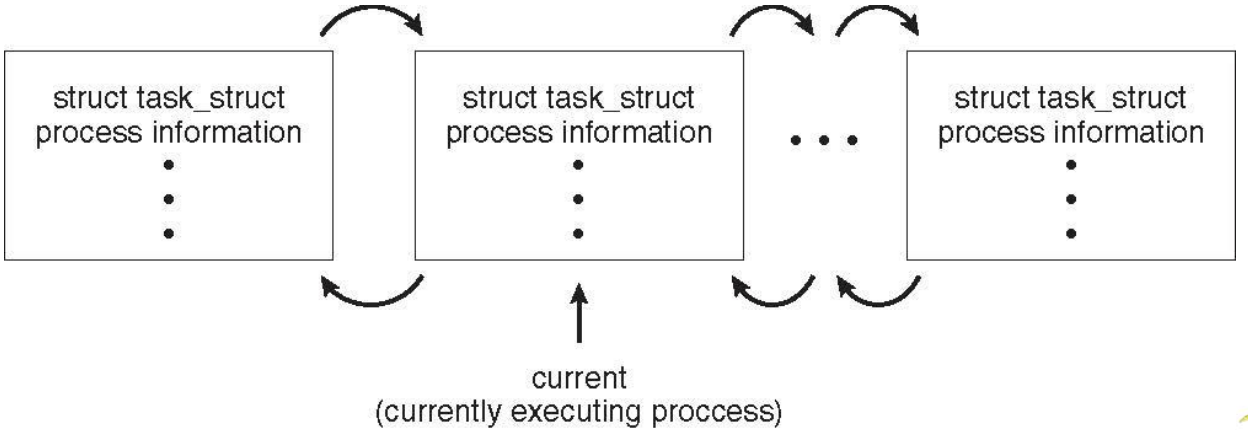
```
pid_t pid; /* process identifier */
long state; /* state of the process */
struct sched_entity se;     /* scheduling information */
struct task_struct *parent; /* this process's parent */
struct list_head *children; /* this process's children */
struct files_struct *files; /* list of open files */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

Alcuni campi della struttura **`task_struct`**

Processi in LINUX – 2

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

```
current->state = nuovo_stato;
```

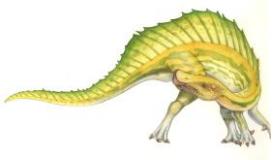


Scheduling dei processi – 1

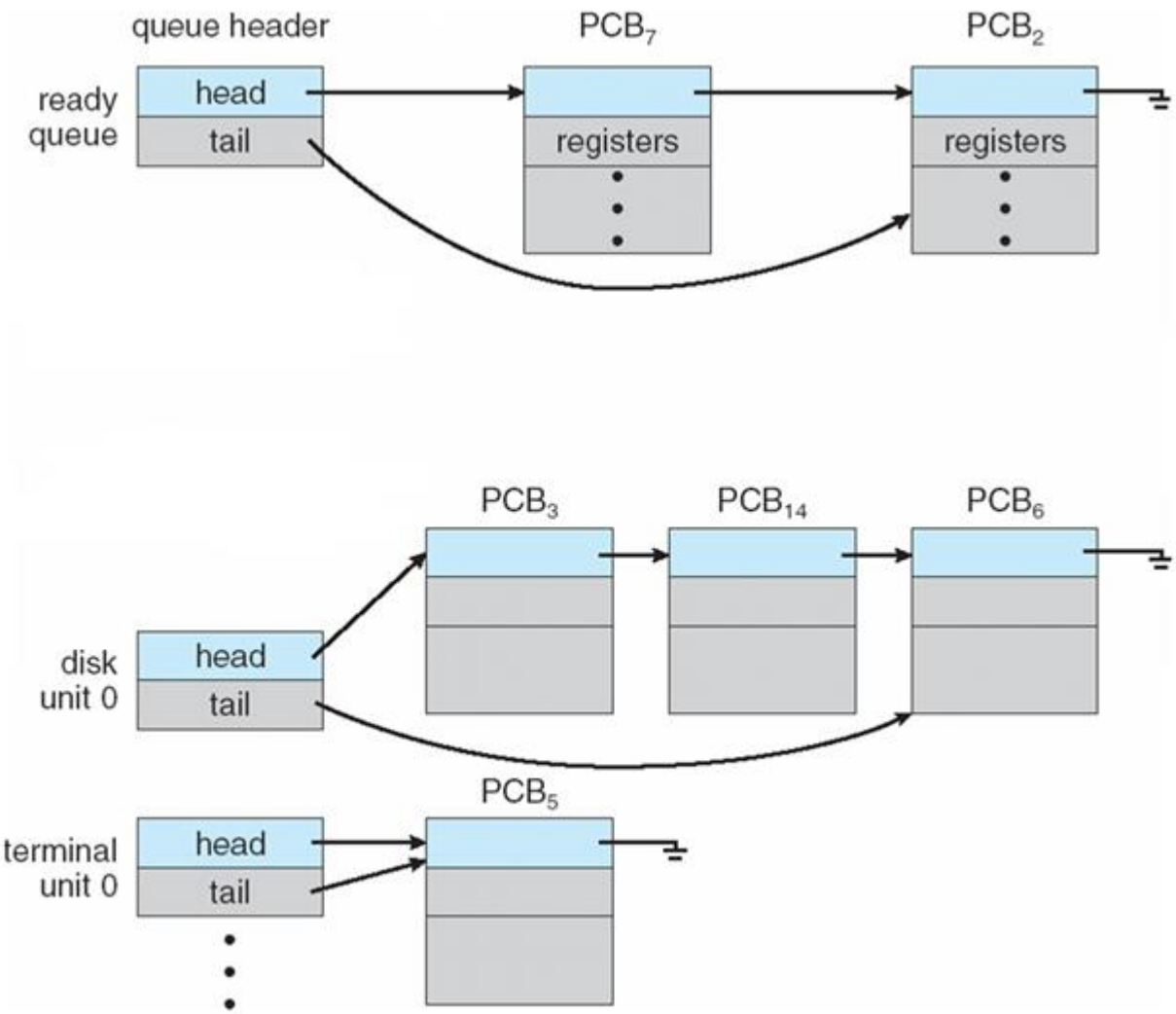
- Meccanismo messo in atto per massimizzare l'utilizzo della CPU, che consiste nel passare rapidamente dall'esecuzione di un processo al successivo, garantendo il time sharing
- È il modulo del SO noto come CPU scheduler che esegue questo compito
- Code per lo scheduling dei processi
  - Ready queue (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

Scheduling dei processi – 2

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

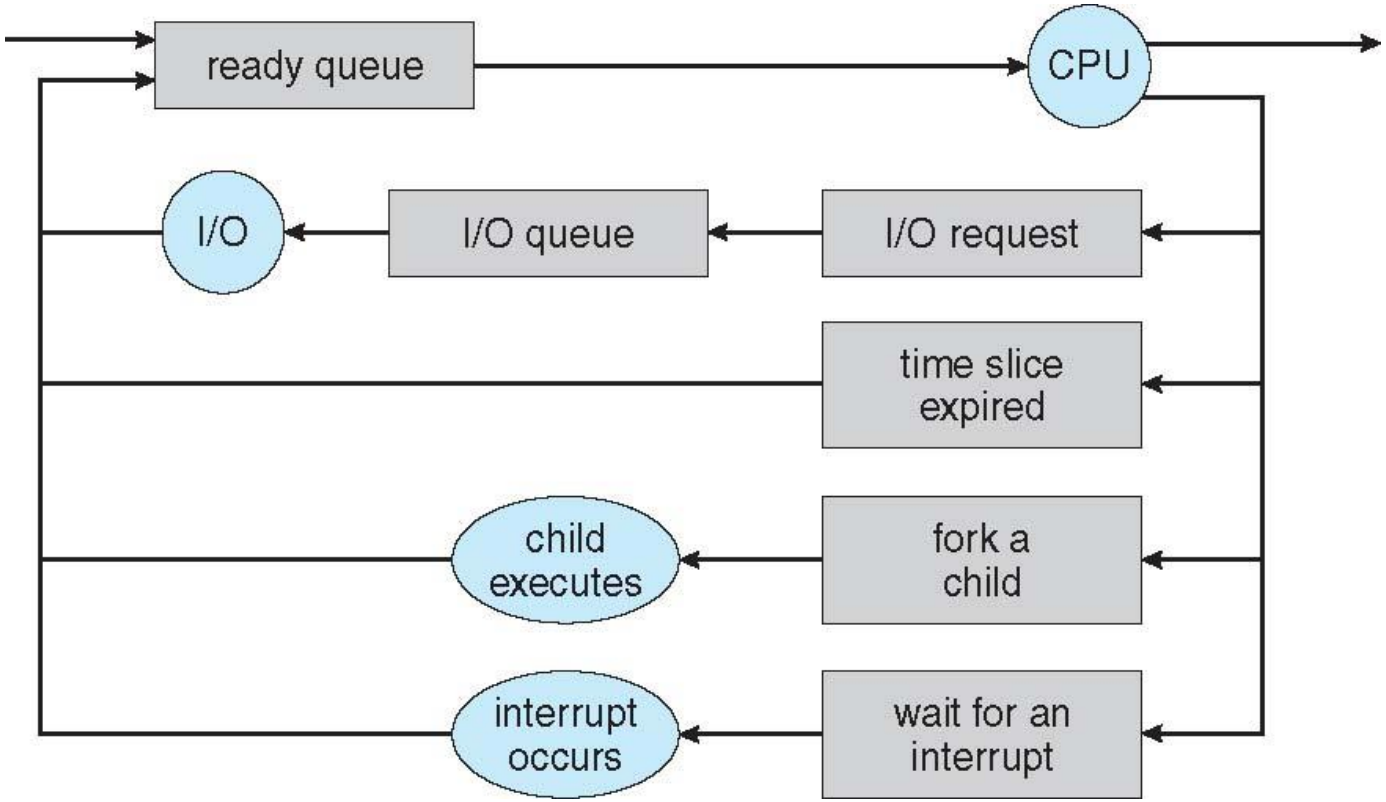


Ready queue e code ai dispositivi di I/O



wait queues

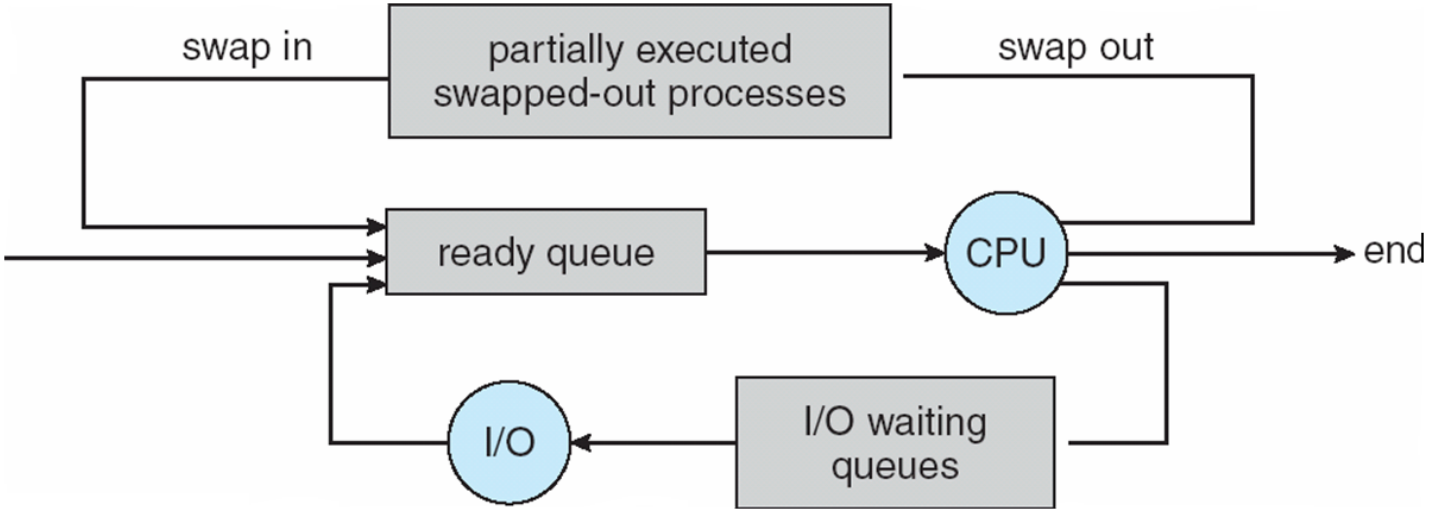
Diagramma per lo scheduling dei processi



- 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

- 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 multiprogrammazione

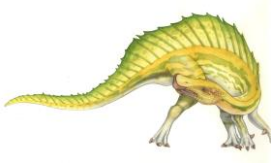


Tipi di scheduler – 2

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

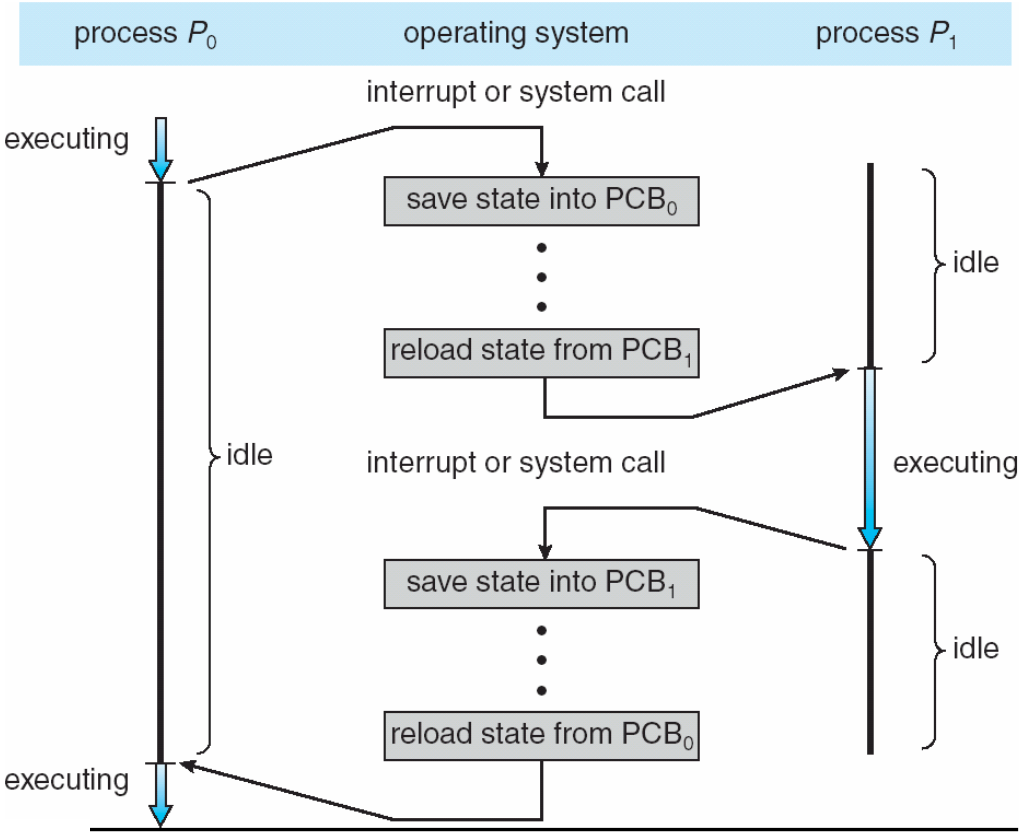
Context switch – 1

- 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



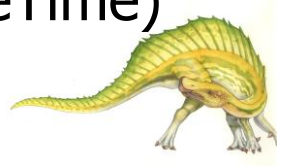
Context switch – 2

- 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



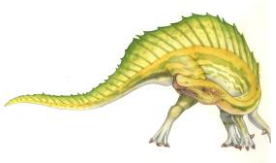
Multitasking per sistemi mobili – 1

- 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

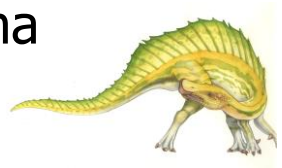


Operazioni sui processi

- Nella maggior parte dei SO, i processi si possono eseguire in concorrenza, e si devono creare e cancellare dinamicamente
- Il SO deve fornire meccanismi per la:
  - creazione di processi
  - terminazione di processi

Creazione di processi – 1

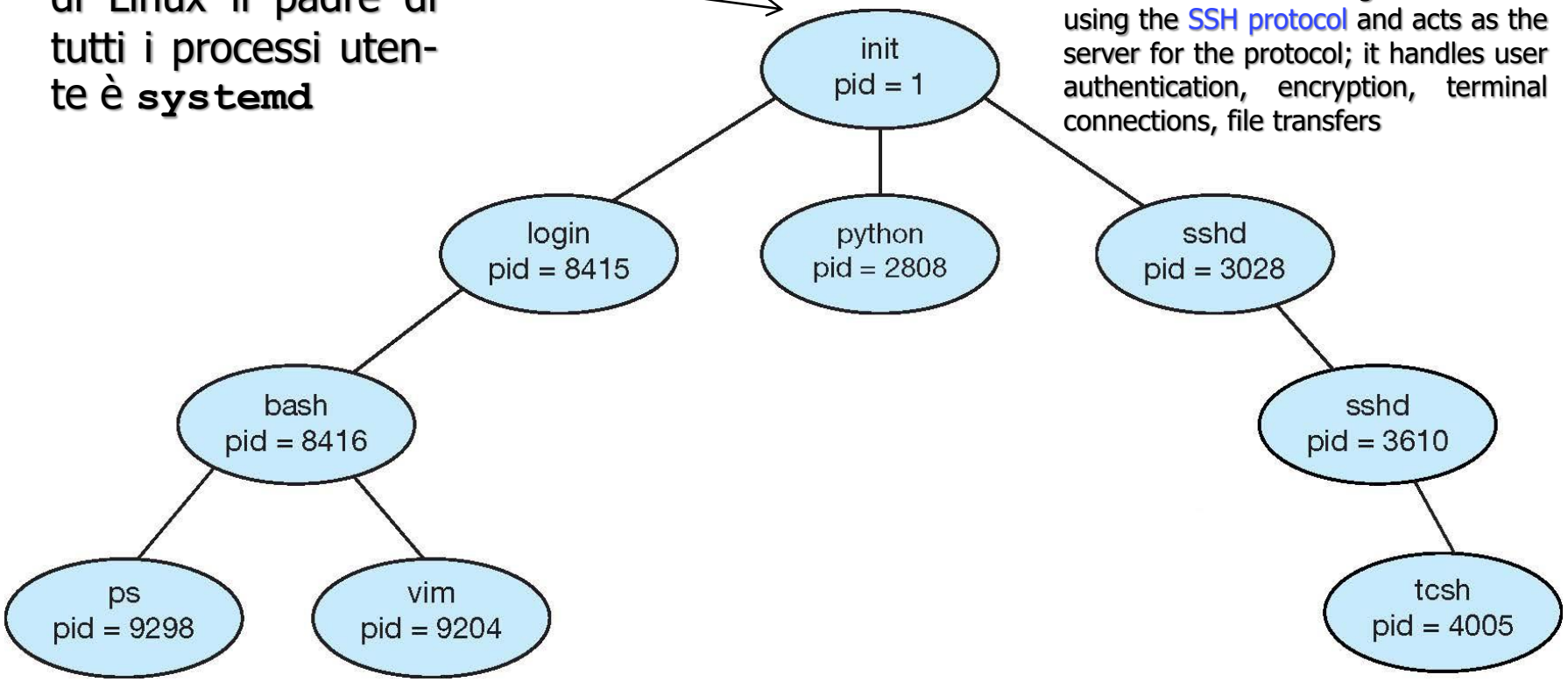
- Un processo può creare altri processi
- Il processo padre crea processi figli che, a loro volta, creano altri processi, formando un albero di processi
- I processi vengono identificati all'interno del sistema tramite un pid (per process identifier)
- Condivisione di risorse
  - Il padre e i figli condividono tutte le risorse
  - I figli condividono un sottoinsieme delle risorse del padre
  - Il padre e i figli non condividono risorse
- Esecuzione
  - Il padre e i figli vengono eseguiti in concorrenza
  - Il padre attende la terminazione dei processi figli
- Spazio degli indirizzi
  - Il processo figlio è un duplicato del processo padre
  - Nel processo figlio viene caricato un diverso programma



Albero dei processi in UNIX

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

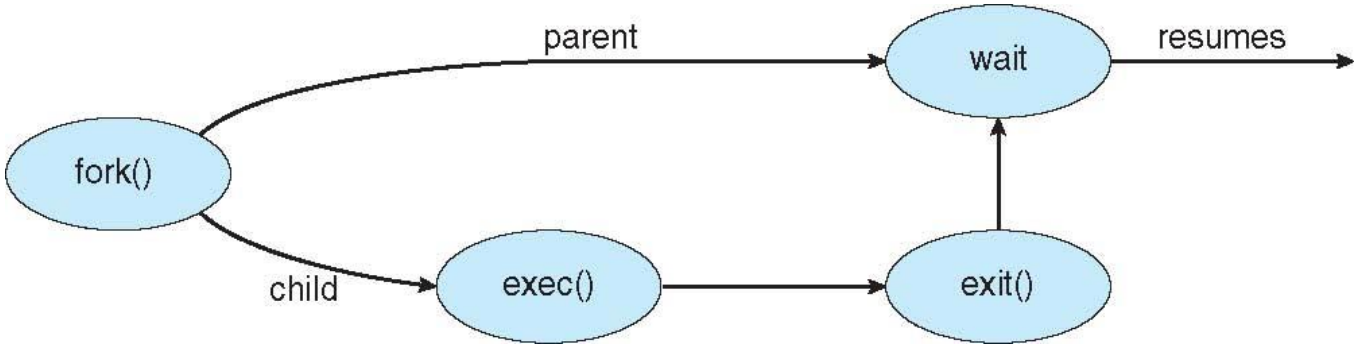
`sshd` is the OpenSSH server process; it listens to incoming connections using the SSH protocol and acts as the server for the protocol; it handles user authentication, encryption, terminal connections, file transfers



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

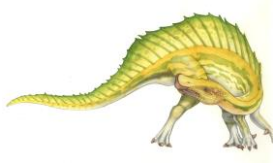
```
#include <sys/types.h>
#include <stdio.h>
#include <unistd.h>

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);
    }
    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 **`execlp()`**
- Il genitore attende il completamento del processo figlio con la chiamata di sistema **`wait()`**
- Al termine del processo figlio, il processo padre riprende dalla chiamata **`wait()`** e quindi termina invocando **`exit()`**

Creazione di processi – 5

- Tutte le funzioni **`execl()`**, **`execlp()`**, **`execle()`**, **`execvl()`**, **`execvp()`** sostituiscono il processo corrente con un nuovo processo: rappresentano un'interfaccia per la system call **`execve()`**
- La funzione **`execlp()`** accetta una lista di argomenti
  - Il primo argomento è convenzionalmente il nome del programma che verrà eseguito (e ne specifica il pathname); all'ultimo argomento valido deve seguire un puntatore a NULL
  - Se **`execlp()`** 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 */
   x = ((res)?(x-1):(x+8));
   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?

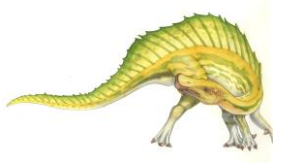
Soluzione 1

```
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

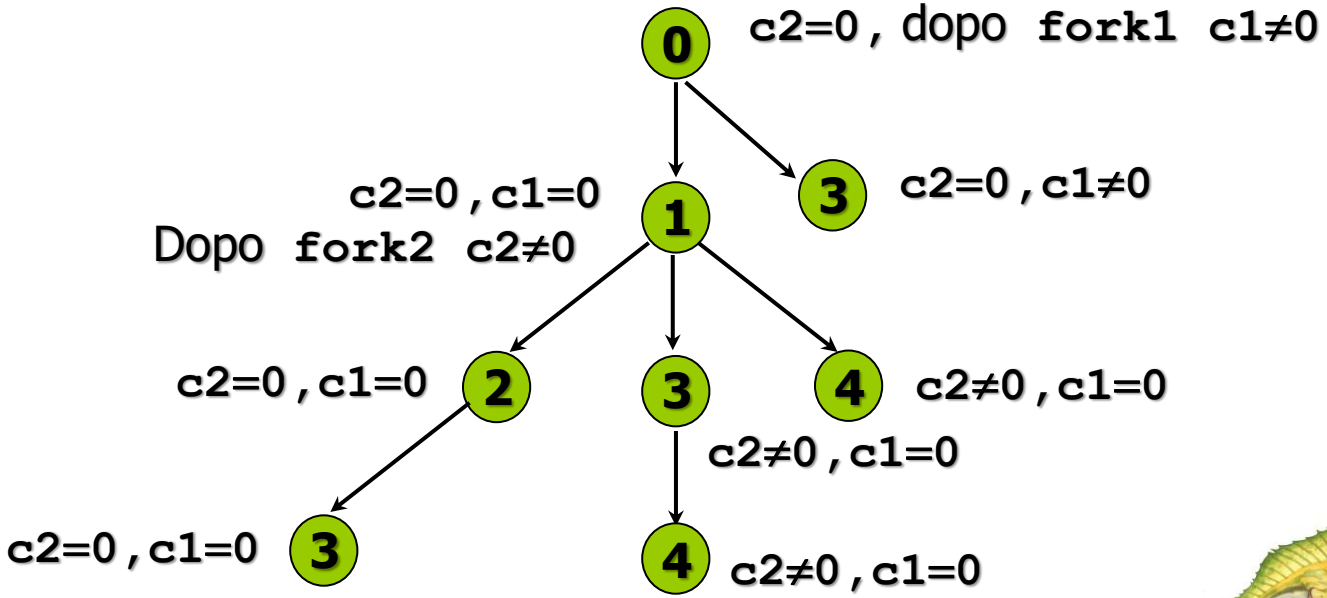
- Si consideri il frammento di codice seguente e si determini quanti processi vengono creati:

```
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 */
```

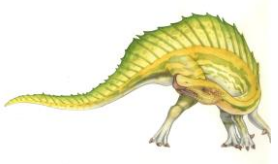


Esercizio 3

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

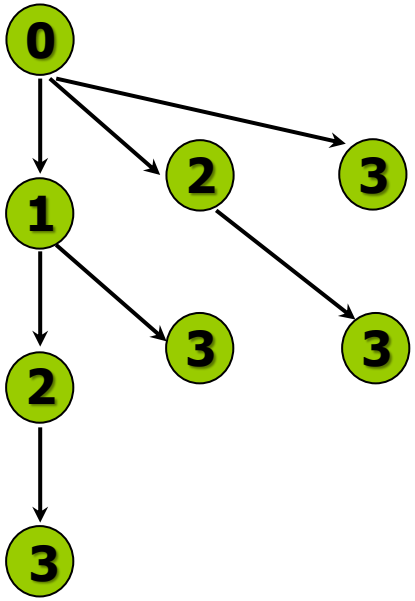
```
#include <stdio.h>
main ( )
{
    pid_t f1, f2, f3;
    f1 = fork();
    f2 = fork();
    f3 = fork();
    printf ("%i %i %i ",(f1>0),(f2>0),(f3>0));
}
```

- Cosa identificano i numeri stampati?

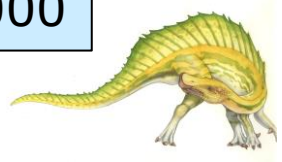


Soluzione 3

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



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



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);
}
```

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

- Descriverne tutti i possibili output relativi alle due invocazioni:

```
$ test 5
$ test 0
```

motivando la risposta data.

Architettura multiprocesso – Il browser CHROME

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

Terminazione di processi – 1

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

Terminazione di processi – 2

- Il padre può terminare l'esecuzione dei processi figli (`kill(pid,signal)`)…
  - …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:
    - Alcuni sistemi operativi non consentono ad un processo figlio di continuare l'esecuzione ⇨ Questo fenomeno è detto *terminazione a cascata* e viene avviato dal SO
    - Nel caso di UNIX, i processi "orfani" non vengono terminati, ma diventano figli di `init()` (**`systemd()`**), che è il progenitore di tutti i processi utente

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

Terminazione di processi – 4

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

Gerarchia dei processi Android

- 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
  - Processo visibile: che esegue un'attività a cui il processo foreground fa riferimento
  - Processo di servizio: che esegue in background ma svolgendo un'attività evidente per l'utente
  - Processo in background: svolge un'attività non evidente per l'utente
  - Processo vuoto: non contiene componenti attive associate ad una app
- 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

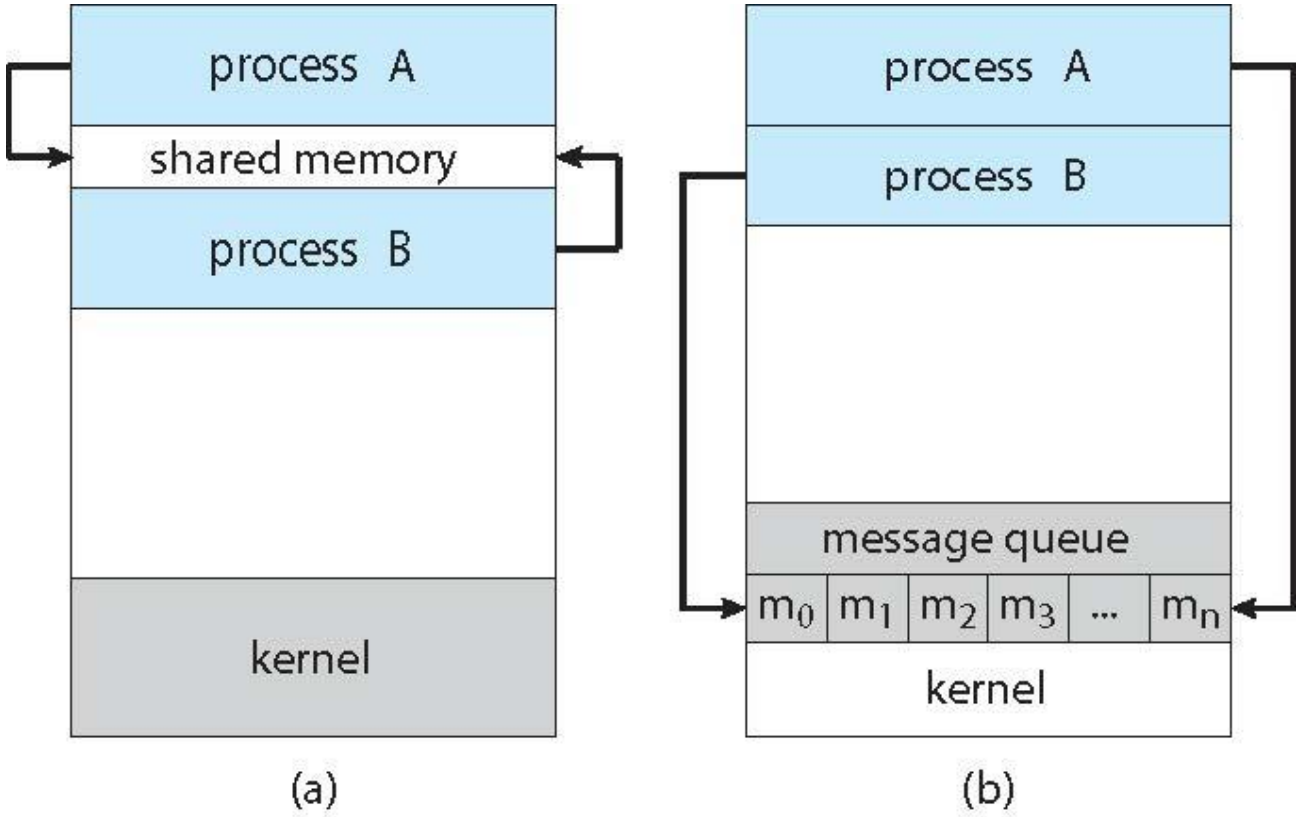
Comunicazione tra processi – 1

- Un processo è **indipendente** se la sua esecuzione non può influire sull'esecuzione di altri processi nel sistema o subirne gli effetti
- I processi **cooperanti** 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
- Vantaggi della cooperazione fra processi
  - **Condivisione di informazioni:** ambienti con accesso concorrente a risorse condivise
  - **Accelerazione del calcolo:** possibilità di elaborazione parallela (in presenza di più CPU)
  - **Modularità:** funzioni distinte che accedono a dati condivisi (es. kernel modulari)
  - **Convenienza:** anche per il singolo utente, possibilità di compiere più attività in parallelo

Comunicazione tra processi – 2

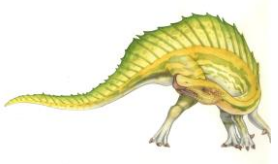
- Due meccanismi di comunicazione fra processi (*IPC*, *Inter–Process Communication*):
  - Scambio di messaggi: utile per trasmettere piccole quantità di dati, nessuna conflittualità, utilizzo di system call per la messaggeria
  - Memoria condivisa: massima efficienza nella comunicazione
    - Richiede l'intervento del kernel solo per l'allocazione della memoria
    - 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

Modelli di comunicazione – 1

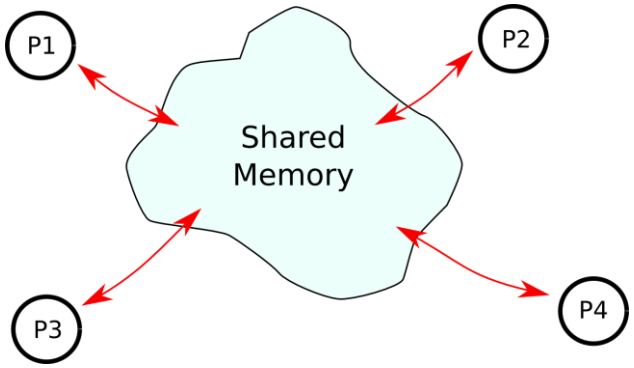


(a) Memoria condivisa

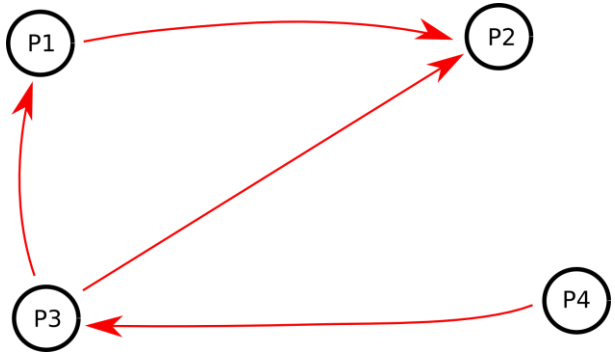
(b) Scambio di messaggi



Modelli di comunicazione – 2



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



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

- Lo scambio di messaggi costituisce il modello di riferimento per la comunicazione tra processi
- 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

Sistemi a memoria condivisa

- Memoria condivisa: inizialmente, parte dello spazio di indirizzi del processo che la alloca
- I processi cooperanti – che la usano per comunicare – dovranno annettere la zona di memoria al loro spazio di indirizzi
- La gestione della memoria condivisa, una volta allocata, non dipende dal SO
  - Il tipo e la collocazione dei dati sono determinati dai processi
  - …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
    - 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

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;
```

`in` indica la successiva posizione libera nel buffer

`out` indica la prima posizione piena nel buffer

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

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;
}
```

Processo produttore

```
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 */
}
```

Processo consumatore

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
  - scambiare messaggi per mezzo di send/receive

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?
  - Il formato del messaggio che un canale può gestire è fisso o variabile?
  - Sono preferibili canali monodirezionali o bidirezionali?
- Implementazione del canale di comunicazione:
  - fisica – memoria condivisa, bus hardware, rete
  - logica – comunicazione diretta o indiretta, sincrona o asincrona, con bufferizzazione assente o automatica

Comunicazione diretta – 1

- I processi devono "nominare" esplicitamente i loro interlocutori (modalità simmetrica):
  - send(P, messaggio) – invia un messaggio al processo P
  - receive(Q, messaggio) – riceve un messaggio dal processo Q
- Modalità asimmetrica
  - 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
- Proprietà del canale di comunicazione:
  - I canali vengono stabiliti automaticamente
  - Ciascun canale è associato esattamente ad una coppia di processi
  - Tra ogni coppia di processi comunicanti esiste esattamente un canale
  - Il canale può essere unidirezionale (ogni processo collegato al canale può soltanto trasmettere/ricevere), ma è normalmente bidirezionale

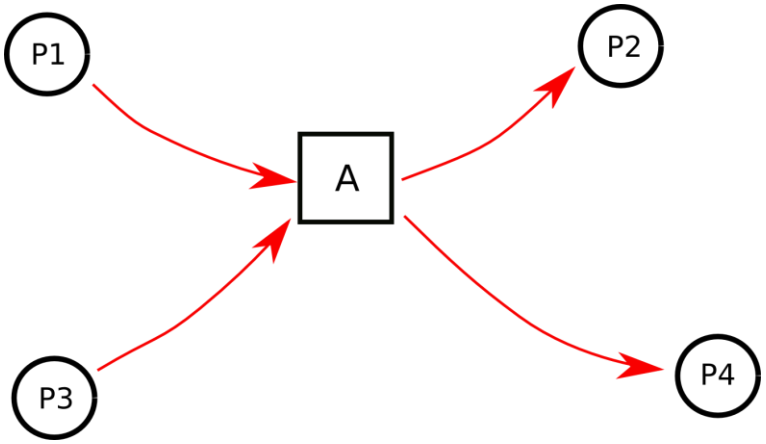
Comunicazione diretta – 2

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

Comunicazione indiretta – 1

- Per ovviare ai difetti della nominazione diretta si usa la nominazione indiretta basata su porte o mailbox
- I messaggi vengono inviati/ricevuti a/da mailbox, o porte
  - 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
  - I canali possono essere unidirezionali o bidirezionali

Comunicazione indiretta – 2



- Operazioni:
  - Creare una nuova mailbox
  - Inviare/ricevere messaggi attraverso mailbox
  - Rimuovere una mailbox
- Primitive di comunicazione:
  - send(A, messaggio) – invia un messaggio alla mailbox A
  - receive(A, messaggio) – riceve un messaggio dalla mail-box A

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?
- Soluzioni:
  - Permettere ad un canale di essere associato ad al più due processi
  - Permettere ad un solo processo alla volta di eseguire un'operazione di ricezione
  - 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

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
  - Quando il processo termina, la mailbox scompare
  - Tuttavia… il diritto di proprietà (e di ricezione) si può passare ad altri processi mediante opportune chiamate al sistema
  - Una mailbox creata dal SO ha vita autonoma (ma presenta i problemi di smistamento descritti)

Sincronizzazione – 1

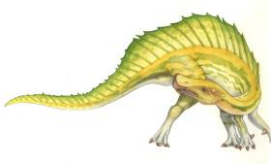
- Lo scambio di messaggi può essere sia **bloccante** che **non–bloccante**
- In caso di scambio di messaggi bloccante, la comunicazione è **sincrona**
  - Il processo mittente si blocca nell'attesa che il processo ricevente, o la porta, riceva il messaggio
  - Il ricevente si blocca nell'attesa dell'arrivo di un messaggio
    - *Rendezvous* tra mittente e ricevente
- 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
  - **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



Sincronizzazione – 4

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

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

```
message next_consumed;
while (true) {
   receive(next_consumed);

   /* consuma l'elemento in next_consumed */
}
```

Buffering

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

Bufferizzazione assente

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

Bufferizzazione automatica

2. **Capacità limitata** – Il canale può avere al più n messaggi in attesa; se il canale è pieno, il trasmittente deve attendere
3. **Capacità illimitata** – Il canale può "contenere" infiniti messaggi; il trasmittente non attende mai

Sistemi per IPC

- UNIX utilizza lo standard POSIX, che prevede IPC sia tramite memoria condivisa che scambio di messaggi
- Mach implementa la comunicazione tramite scambio di messaggi
  - I messaggi si inviano/ricevono tramite porte
  - Anche le chiamate di sistema si realizzano tramite messaggi
  - Quando si crea un nuovo task si creano anche due porte speciali: Task Self e Notify
  - Il kernel possiede i permessi di ricezione per la porta Task Self per ricevere richieste dal task e usa la porta Notify per segnalare l'occorrenza di eventi
- Windows usa la memoria condivisa come meccanismo per supportare alcune forme di message passing

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…

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

  - 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);
```

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

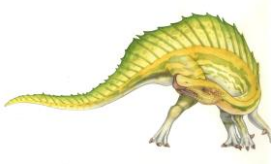
```
sprintf(psm,"Writing to shared memory");
```

IPC POSIX: Produttore

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

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

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

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

    /* Scrive sul segmento di memoria condivisa */
    sprintf(ptr,"%s",message0);
    ptr += strlen(message0);
    sprintf(ptr,"%s",message1);
    ptr += strlen(message1);
    sprintf(ptr,"%s",message2);

    return 0;
}
```

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);

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

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

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

    return 0;
}
```



Esempio di sistema IPC: POSIX (cont.)

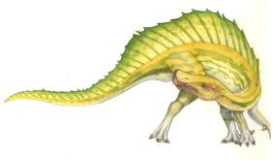
- La funzione **`mmap()`** crea un file mappato in memoria, che contiene il segmento di memoria condivisa, e restituisce un puntatore al file mappato in memoria che viene utilizzato per accedervi
  - Il flag **`MAP_SHARED`** specifica che le modifiche apportate alla memoria condivisa saranno visibili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

Pipe – 1

- 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 – 2

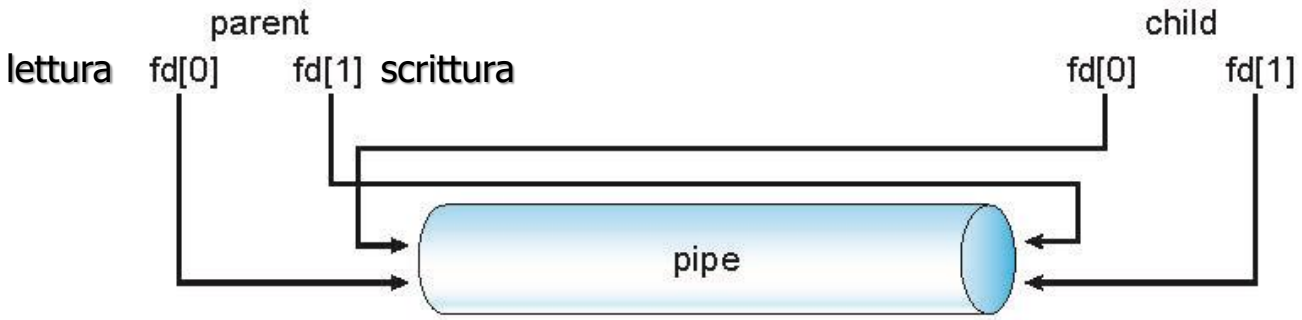
- Pipe convenzionali
  - Non possono essere accedute se non dal processo che le ha create
  - Esistono solo per la durata del tempo di comunicazione
  - Permettono ai processi di comunicare seguendo il paradigma del produttore–consumatore
  - Sono unidirezionali: se viene richiesta la comunicazione a doppio senso devono essere utilizzate due pipe
  - Nel gergo di Windows, sono dette pipe anonime



Pipe – 3

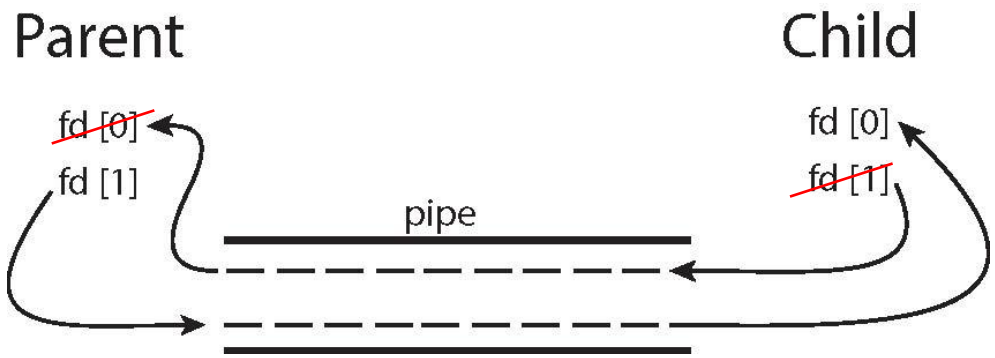
- Pipe convenzionali
  - In UNIX:
    - Le pipe convenzionali sono realizzate utilizzando la funzione

      `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



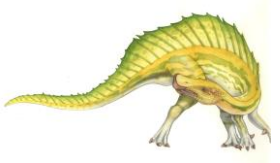
Pipe – 4

- **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



Pipe – 5

```
#define BUFFER_SIZE 25
#define READ_END 0
#define WRITE_END 1

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

      /** crea la pipe */
      if (pipe(fd) == -1) {
            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 */
            /* chiude l'estremità non utilizzata della pipe */
            close(fd[READ_END]);

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

            /* 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);

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

      return 0;
}
```

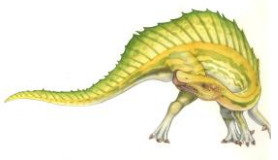


Pipe – 6

- Named pipe
  - Possono essere utilizzate da processi qualunque (non necessariamente in relazione padre–figlio)
  - La comunicazione può essere bidirezionale e la pipe continua ad esistere anche quando i processi comunicanti sono terminati
  - Più processi possono usarla per comunicare
  - Disponibili sia in Windows che in UNIX
  - In UNIX:
    - 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

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)



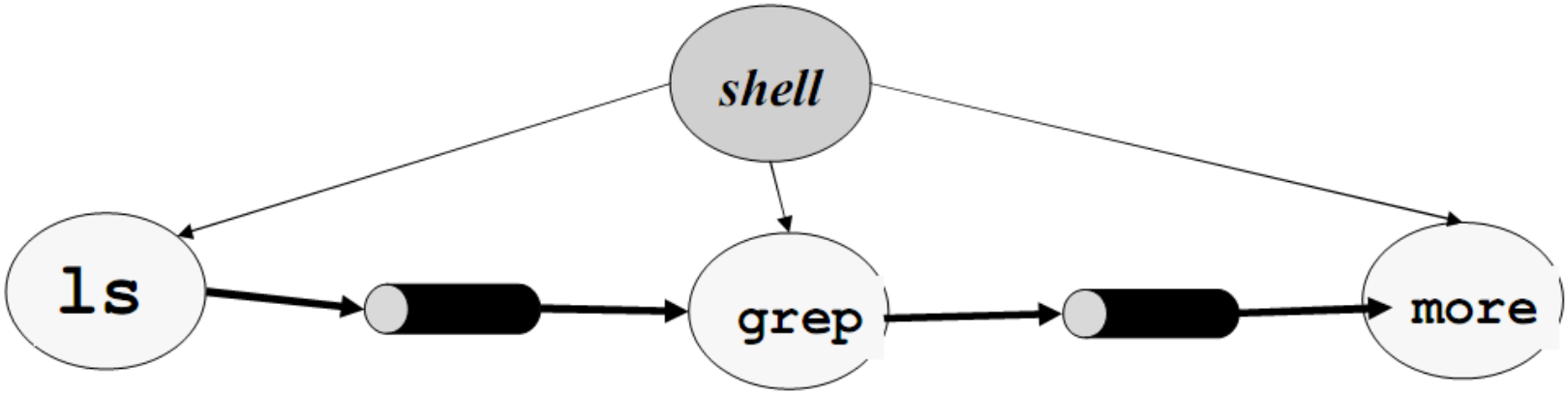
Le pipe in pratica – 1

- 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 "|"
- Esempio:

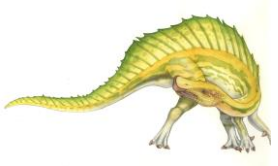
```
$ ls –lR|grep Jun|more
```

  - 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

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`**

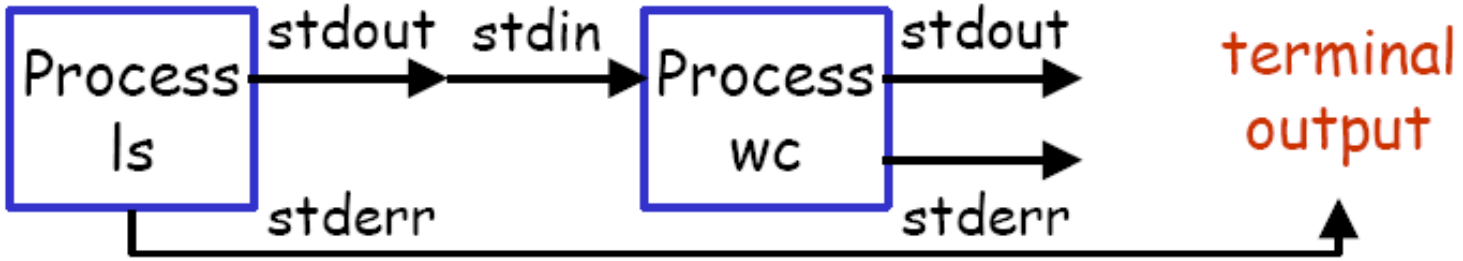


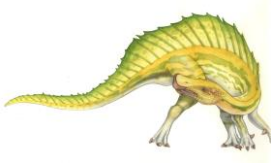
Le pipe in pratica – 3

- Esempio:

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

  - Si ottiene la lista dei file della directory corrente
  - Si "contano" le parole contenute nella lista ⇨ numero di file contenuti nella directory





Le pipe in pratica – 4

- Esempi:

```
$ ls | grep rwxrwxrwx
```

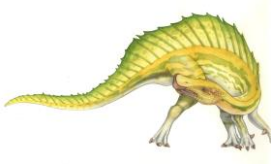
  - Stampa l'elenco di tutti i file nella directory locale che hanno permesso **`rwxrwxrwx`** (o che hanno **`rwxrwxrwx`** nel loro nome)

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

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

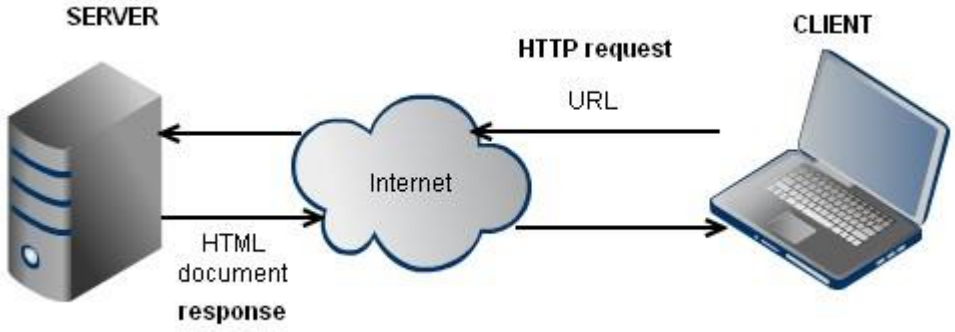
Comunicazioni in sistemi client–server

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



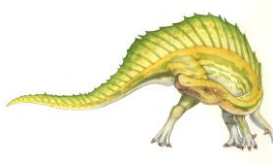
Socket – 1

- 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
  - 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
  - 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



Socket – 3

- 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
- Le socket forniscono quindi un'astrazione, che permette di far comunicare computer diversi che utilizzano lo stesso protocollo

Socket – 4

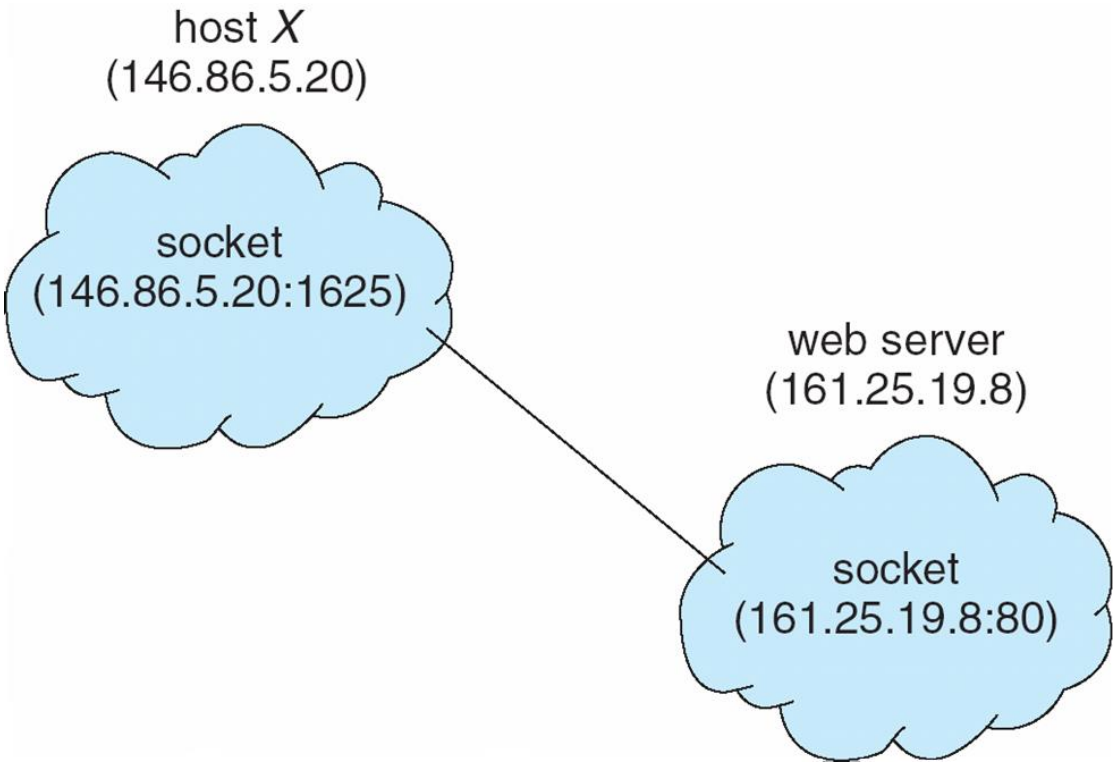
- Una socket è definita come l'estremità di un canale di comunicazione
- Ogni socket è identificata da un indirizzo IP concatenato ad un numero di porta
- Esempio: la socket 161.25.19.8:1625 si riferisce alla porta 1625 sull'host 161.25.19.8
- La comunicazione si stabilisce fra coppie di socket (una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

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)
- Tutte le porte al di sotto del valore 1024 sono considerate note e si usano per realizzare servizi standard
- L'indirizzo IP 127.0.0.1, noto come *loopback*, è peculiare ed è usato da un elaboratore per riferirsi a sé stesso
  - ⇨ Un client e un server residenti sulla stessa macchina sono in grado di comunicare via TCP/IP

Socket – 6

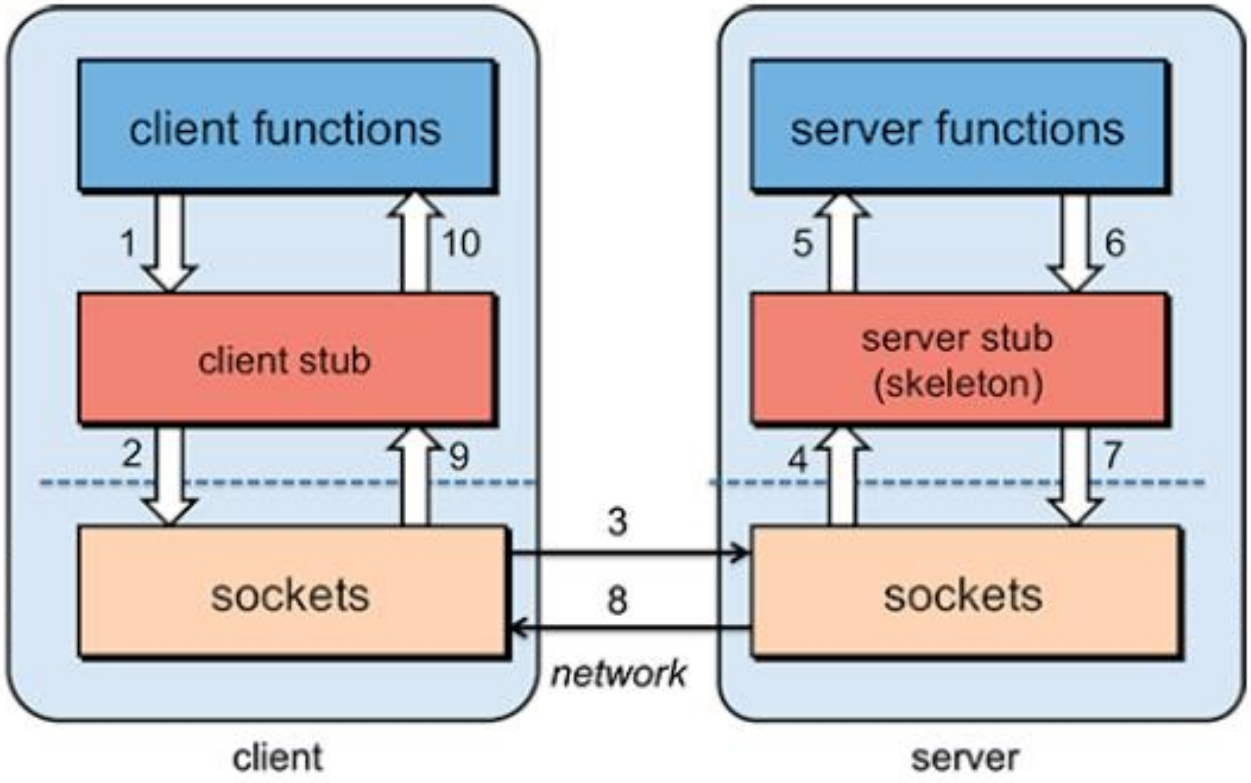
- La comunicazione tramite socket è una forma di comunicazione fra sistemi distribuiti a basso livello
- Le socket permettono la trasmissione di un flusso non strutturato di byte
  - È responsabilità del client e del server organizzare ed interpretare i dati



Chiamate a procedura remota – 1

- 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
    - 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

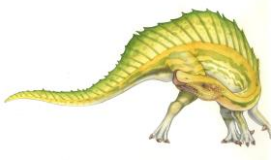
Chiamate a procedura remota – 2



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

Chiamate a procedura remota – 3

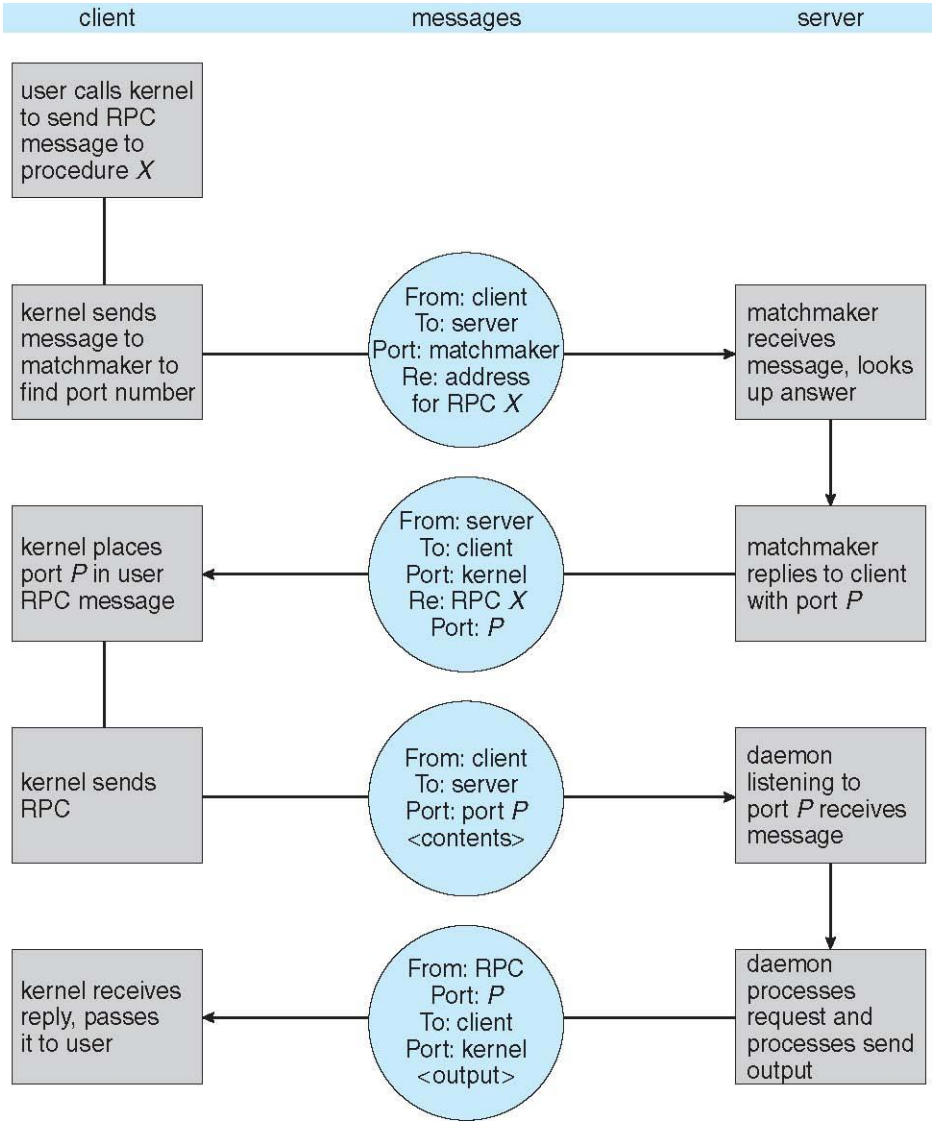
- Il sistema delle RPC nasconde i dettagli necessari alla comunicazione (socket)
- Problemi
  - Eventuale diversità di rappresentazione dei dati fra sistemi remoti (es.: CPU 32/64 bit, *big–endian*, *little–endian*)
    - ⇨Utilizzo di un formato standard di trasmissione (es.: *XDR – External Data Representation*)
  - Le chiamate di procedure remote possono fallire a causa di errori dovuti alla trasmissione via rete
  - Semantica delle chiamate per assicurarsi che ogni RPC venga eseguita una volta soltanto
    - ⇨Messaggi di richiesta RPC marcati con l'ora di emissione
    - ⇨Ricevuta al client

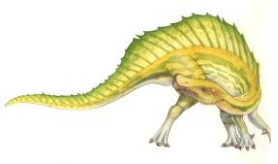


Chiamate a procedura remota – 4

- 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 ↔ porta
  - Associazione dinamica mediata da un servizio di rendez-vous
    - Il server ha un demone in ascolto che riceve una richiesta dal client, alla quale risponde con il numero di porta relativo alla particolare chiamata

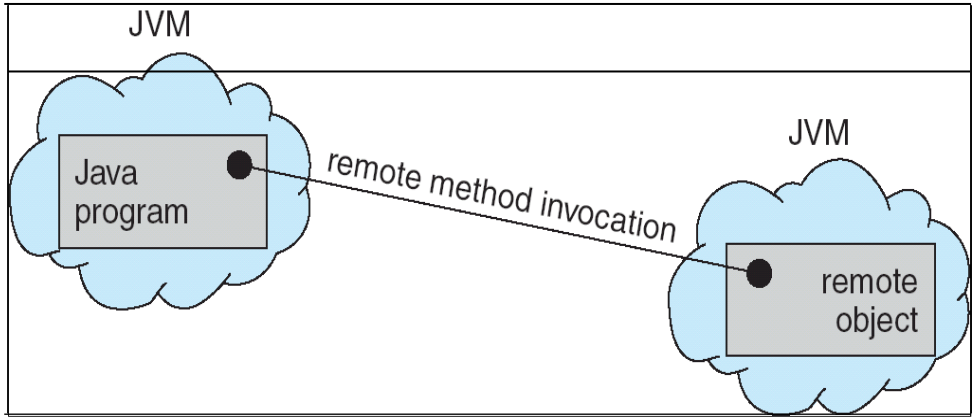
Chiamate a procedura remota – 5





Chiamate a procedura remota – 6

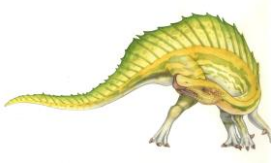
- Sebbene le RPC siano di solito associate all'elaborazione client–server, possono anche essere usate per la comunicazione fra processi in esecuzione sullo stesso sistema
  - Android ha un ricco insieme di meccanismi IPC (nel Binder), tra cui le RPC utilizzabili in locale
- L'invocazione di metodi remoti è una funzione del linguaggio Java simile alla RPC
- 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



Esercizio 1

- 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