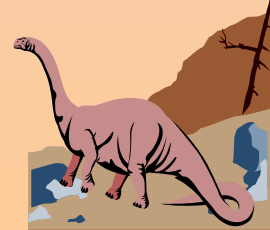




# Capitolo 3: Processi

- Concetto di processo.
- Scheduling dei processi.
- Operazioni sui processi.
- Processi cooperanti.
- Comunicazione tra processi.
- Esempi di sistemi per la IPC.
- Comunicazione nei sistemi client/server.





# Concetto di processo

- Un S.O. esegue una varietà di “programmi”:
  - ☞ i sistemi a lotti (*batch*) eseguivano *lavori (job)*
  - ☞ un sistema a partizione di tempo esegue *programmi utente (task)*
- Spesso i termini *lavoro* e *processo* sono utilizzati in modo intercambiabile.
- Informalmente un processo può essere considerato come un programma in esecuzione.
- Un programma di per sé non è un processo.
- Il programma, anche detto *sezione testo*, è un’entità *passiva*, mentre il processo è un’entità *attiva*.
- Un processo include:
  - ☞ Contatore di programma (program counter)
  - ☞ Pila (stack)
  - ☞ Sezione di dati (data section)





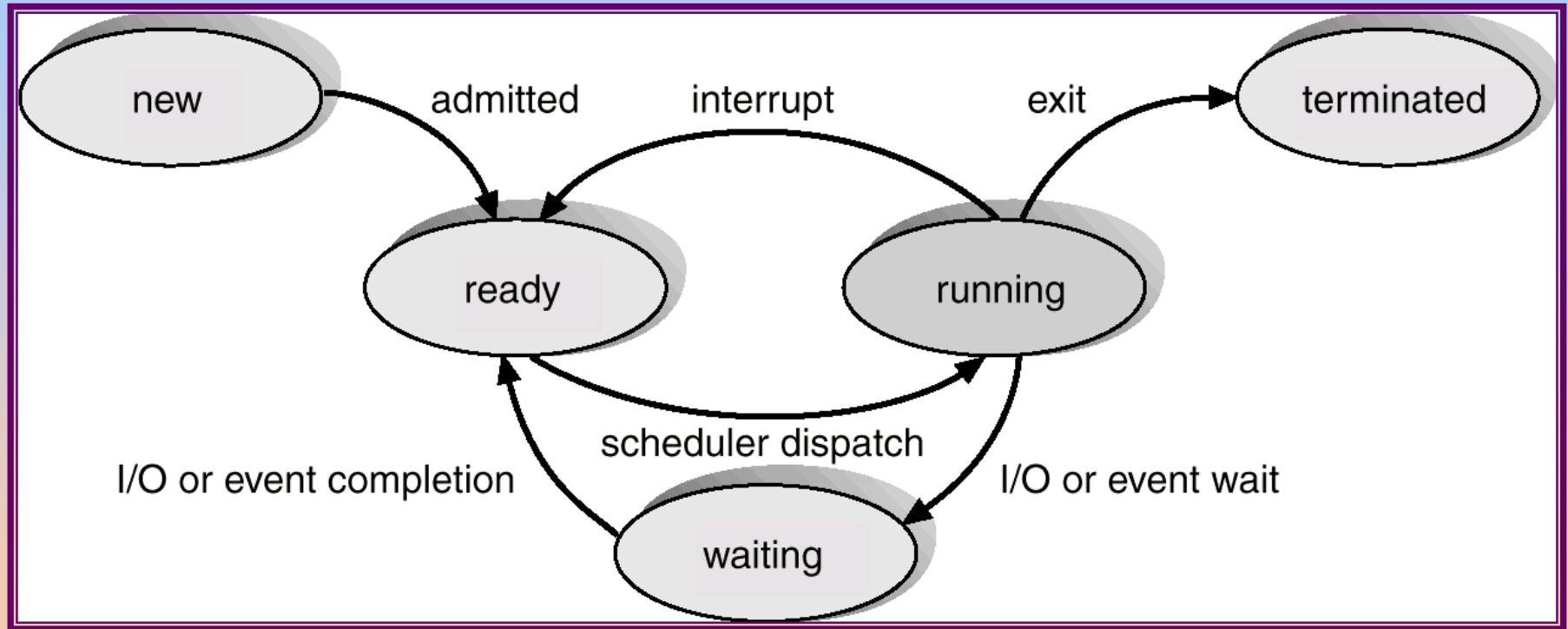
# Stato del processo

- L'esecuzione di un processo deve progredire in maniera sequenziale.
- Mentre un processo è in esecuzione è soggetto a cambiamenti di *stato*.
- Ogni processo può trovarsi in uno tra i seguenti stati:
  - ☞ **Nuovo (new)**: Il processo viene creato.
  - ☞ **In esecuzione (running)**: quando è in memoria ed ha il controllo della CPU;
  - ☞ **In attesa (waiting)**: quando è temporaneamente sospeso in attesa di un evento, quale la terminazione di I/O, lo scadere di un timer, la ricezione di un messaggio etc.;
  - ☞ **Pronto (ready)**: quando è in memoria e pronto per l'esecuzione, ma non ha il controllo della CPU e attende di essere assegnato ad un'unità di elaborazione.
  - ☞ **Terminato (terminated)**: quando termina l'esecuzione e abbandona il sistema.





# Diagramma di transizione degli stati di un processo





# Blocco di controllo dei processi

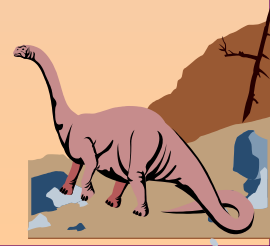
- In un sistema operativo ogni processo è rappresentato da un descrittore di processo:
  - ☞ *process descriptor - PD* - detto anche blocco di controllo di un processo (*process control block - PCB*).
- Nel PCB sono contenute informazioni connesse ad uno specifico processo:
  - ☞ **Stato del processo:** tra new, ready, running, waiting, terminated.
  - ☞ **Contatore di programma:** indica l'indirizzo della successiva istruzione da eseguire.
  - ☞ **Registri di CPU:** accumulatori, registri indice, puntatori alla cima delle strutture a pila (stack pointer), registri d'uso generale e registri contenenti informazioni relative ai codici di condizione.
  - ☞ **Informazioni sullo scheduling di CPU:** priorità del processo, puntatori alle code di scheduling e altri parametri di schedulazione.





# Descrittore di processo (II)

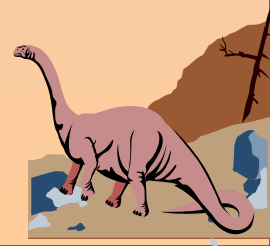
- ✎ **Informazioni sulla gestione della memoria:** registri di base e di limite, tabelle delle pagine in memoria o dei segmenti (a seconda della tecnica usata dal S.O.).
- ✎ **Informazioni di contabilizzazione delle risorse:** tempo di CPU, tempo reale di CPU, numero del processo, etc..
- ✎ **Informazioni di I/O:** lista dei dispositivi di I/O assegnati al processo, elenco file aperti, etc..



# Descrittore di processo (PCB)

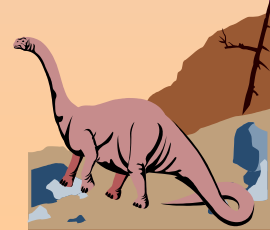
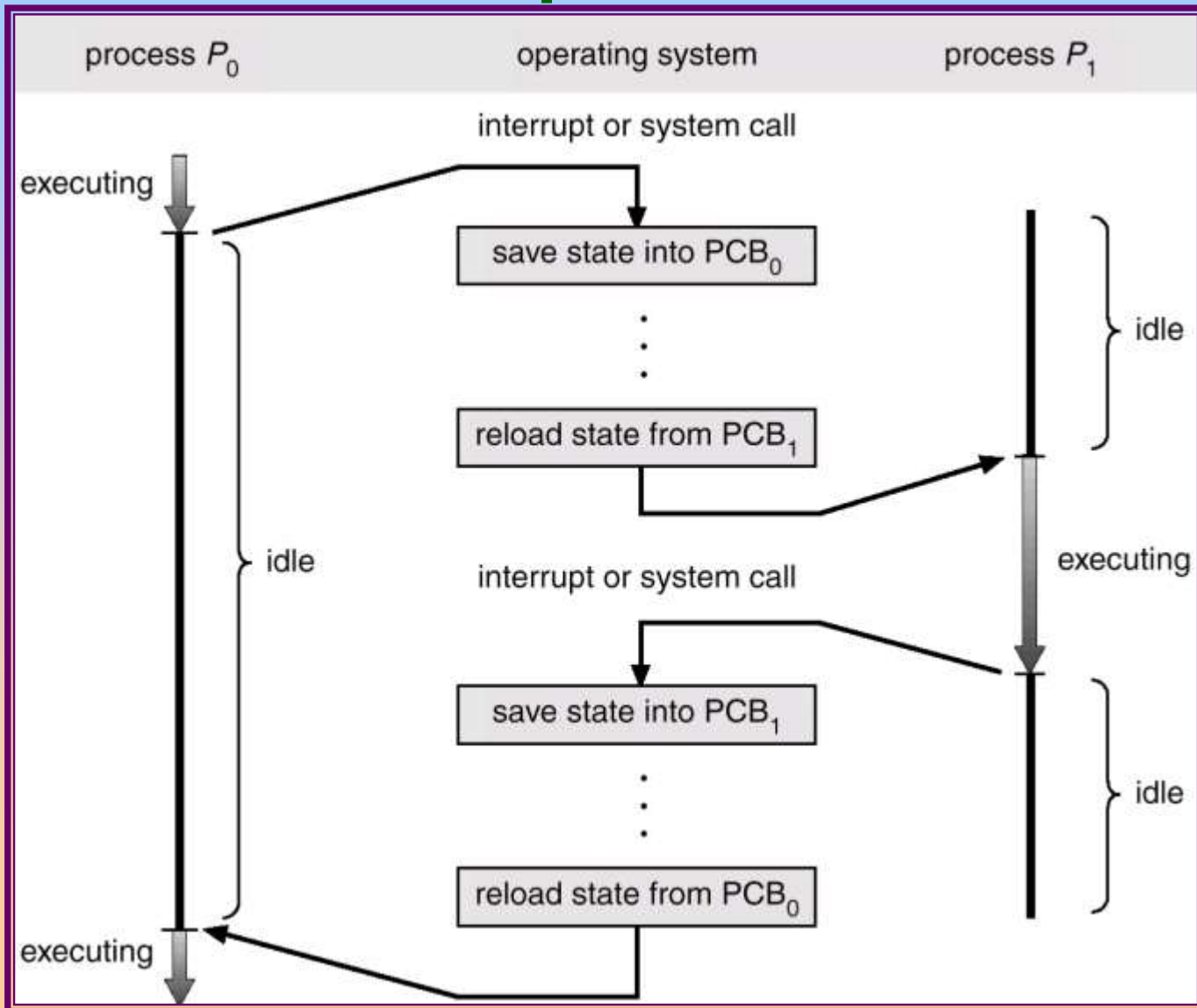


pointer	process state
process number	
program counter	
registers	
memory limits	
list of open files	
• • •	





# La CPU può essere commutata tra i processi







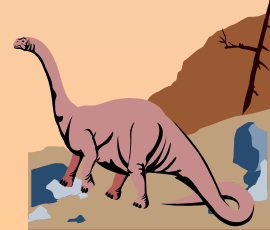
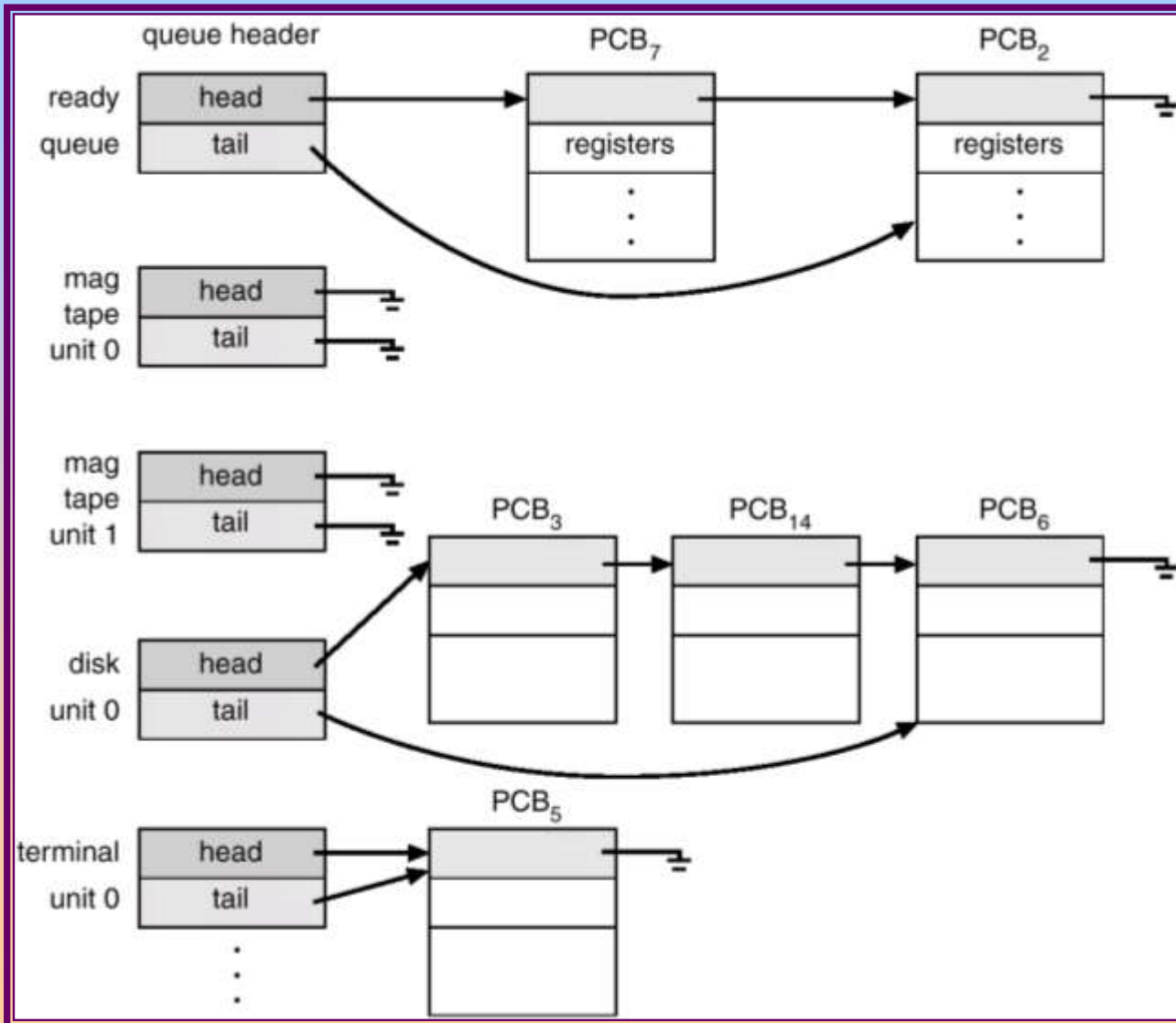
# Scheduling e code di scheduling

- Quando i processi entrano nel sistema vengono inseriti in una *coda dei processi* composta da tutti i processi del sistema.
- I processi che risiedono in memoria centrale e che sono pronti e in attesa di essere eseguiti si trovano in una lista detta *coda dei processi pronti (ready queue)*.
- Generalmente la *coda dei processi pronti* è memorizzata come lista concatenata.
- Quando un processo fa una richiesta di I/O il suo identificativo viene inserito nella *coda del dispositivo*:
  - ☞ non è detto che il dispositivo richiesto sia libero.
- I processi quindi si spostano tra varie code.



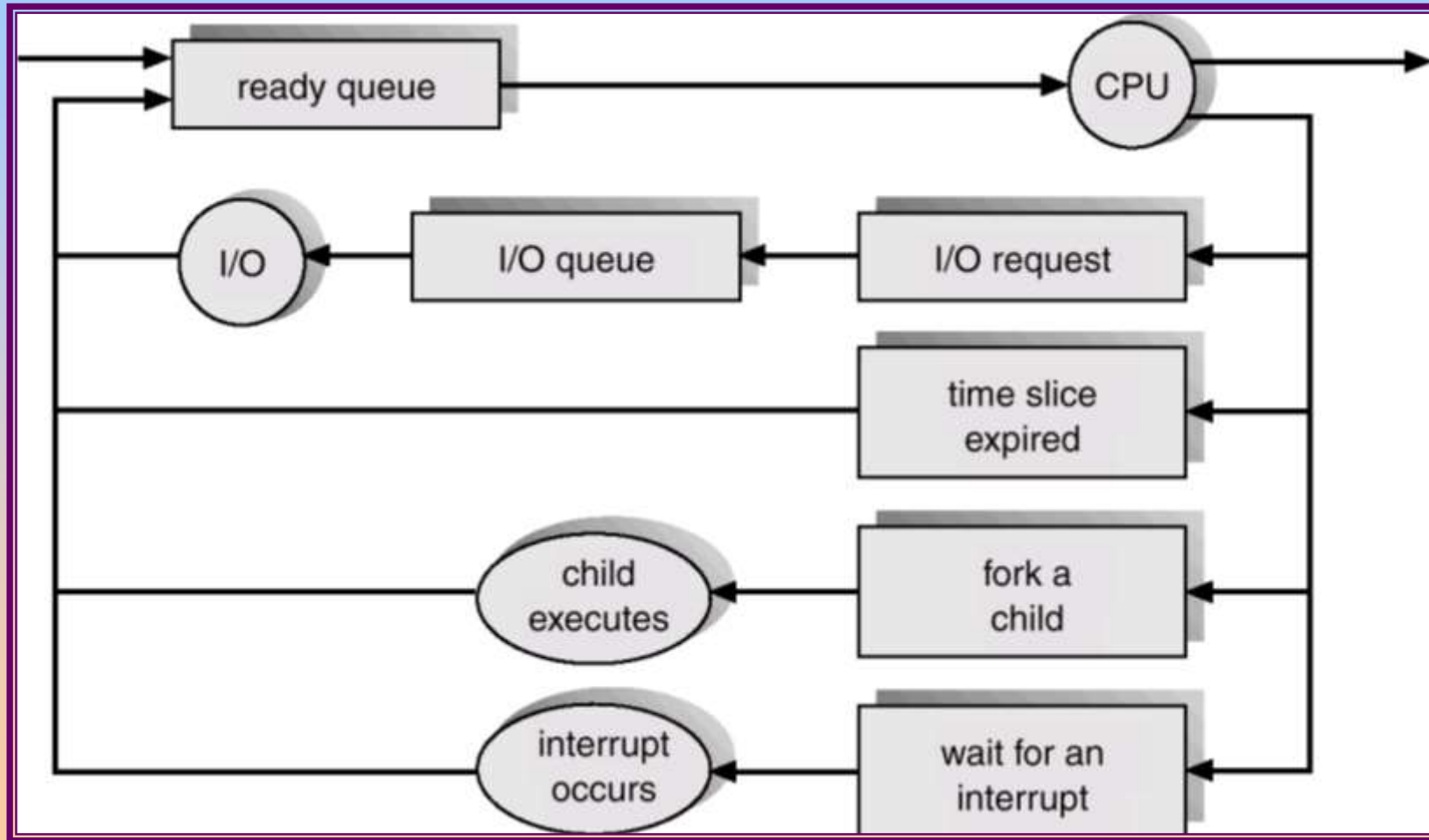


# Coda dei processi pronti e diverse code di dispositivi di I/O





# Diagramma di accodamento per lo scheduling dei processi





# Scheduler di CPU

- Lo scheduler seleziona i processi e li carica in memoria assegnando la CPU in modo da ottimizzarne l'utilizzo.
- **Scheduler a lungo termine** o di job (*long-term scheduler*):
  - ☞ seleziona i processi da inserire nella coda dei processi pronti.
- **Scheduler a breve termine** o di CPU (*Short-term scheduler* o *CPU scheduler*):
  - ☞ seleziona un processo dalla coda dei processi pronti e gli assegna la CPU.
- Lo scheduler di job è uno scheduler che viene eseguito con una frequenza molto bassa,
  - ☞ possono infatti passare dei minuti affinché vengano creati dei nuovi processi.
- Lo scheduler di job controlla il grado di multi-programmazione,
  - ☞ cioè il numero di processi in memoria.
- In un sistema stabile, la velocità media della creazione dei processi deve essere uguale alla velocità con cui i processi lasciano il sistema.



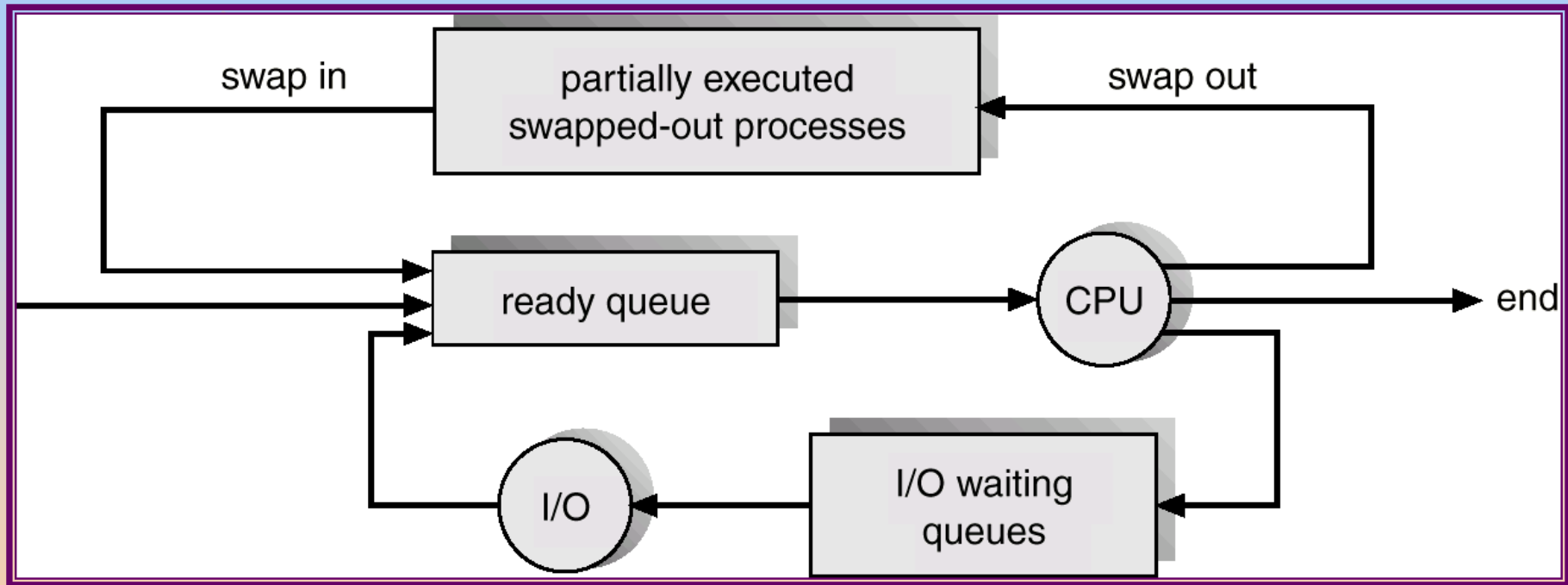


# Scheduler di CPU (II)

- Lo scheduler di CPU viene eseguito molto frequentemente:
  - ☞ il processo potrebbe rimanere in esecuzione pochi millisecondi prima di fare una richiesta di I/O.
- I processi possono essere descritti come:
  - ☞ *Con prevalenza di I/O (I/O-bound):*
    - 📄 impiega la maggior parte del tempo in operazioni di I/O, inframezzando piccoli tempi di CPU.
  - ☞ *Con prevalenza d'elaborazione (CPU-bound):*
    - 📄 poche operazioni di I/O, prevalenza di elaborazione
- Uno scheduler per rendere efficiente il suo lavoro è tenuto a selezionare una buona combinazione di processi I/O bound - CPU bound, altrimenti il sistema risulterebbe sbilanciato.
- **Scheduler a medio termine:**
  - ☞ vantaggio derivato dalla rimozione dei processi dalla memoria; il processo può essere reintrodotta in memoria successivamente, riprendendo l'esecuzione da dove era stata abbandonata
- **Avvicendamento (swapping).**



# Aggiunta di scheduling a medio termine





# Cambio di contesto

## ■ Cambio di contesto (context switch):

- ☞ il passaggio della CPU ad un nuovo processo implica
  - 📄 la registrazione dello stato del processo vecchio
  - 📄 il caricamento dello stato precedentemente registrato del nuovo processo.

## ■ Il tempo necessario al cambio di contesto e' overhead:

- ☞ il sistema non compie nessun lavoro direttamente utile alla computazione.
- ☞ Varia in genere tra 1 e 1000 microsecondi

## ■ La durata del cambio di contesto dipende molto dall'architettura.





# Operazioni: creazione di un processo

- Durante la sua esecuzione un processo può creare altri processi.
- Il processo creante è chiamato *padre*, mentre il processo creato è chiamato *figlio*.
- La gerarchia che esiste in questa struttura è ricorsiva, per cui un processo figlio può essere, a sua volta, padre di altri processi.
- Così facendo si viene a creare un *albero dei processi*.
- Risorse
  - ☞ Padre e figlio condividono tutte le risorse.
  - ☞ Il figlio condivide un sottinsieme delle risorse del padre.
  - ☞ Padre e figlio non condividono risorse.
- Esecuzione
  - ☞ Il padre continua l'esecuzione, concorrentemente al processo figlio.
  - ☞ Il padre attende che i processi figli terminano la loro esecuzione.







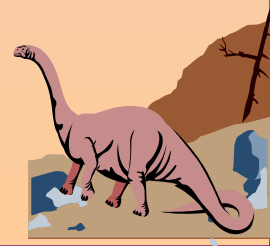
# Creazione di un processo (II)

## ■ Spazio di indirizzi

- Il processo figlio è un duplicato del padre
- Nel processo figlio si carica un programma

## ■ UNIX

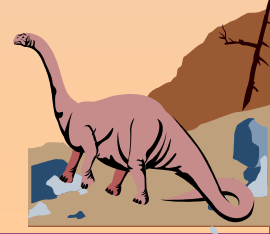
- fork
- exec dopo fork carica un programma nel processo figlio.





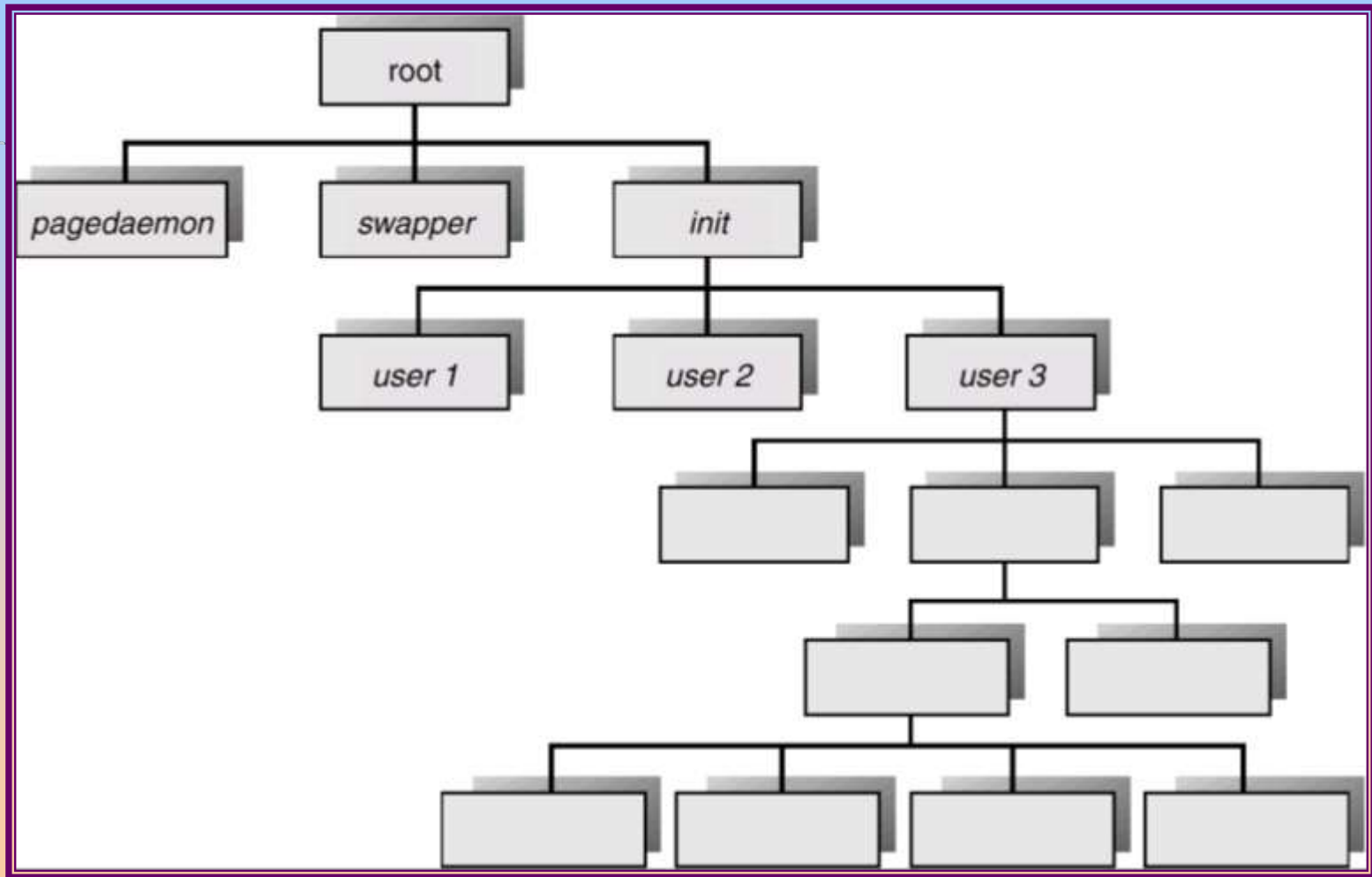
# Creazione di un processo in Unix

```
# include <stdio.h>
void main (int argc, char *argv[ ])
{ int pid;
  /* genera un nuovo processo */
  pid = fork ();
  if (pid < 0)    { /* errore */
                  fprintf(stderr, "errore");
                  exit(-1);
                }
  else if (pid == 0)    { /* processo figlio */
                        execlp ("/bin/ls", "ls", NULL);
                        }
                    else { /* processo genitore */
                        wait (NULL);
                        printf("Il figlio ha terminato");
                        exit(0);
                      }
}
```





# Albero di processi in un sistema UNIX





# Operazioni: terminazione di un processo

- Un processo termina quando finisce l'esecuzione della sua ultima istruzione.
- Con un'apposita system call (*exit*) il processo chiede al sistema di essere cancellato:
  - ☞ Il processo figlio può restituire output al processo padre.
- Si può avere anche una terminazione spuria del processo attraverso una system call (*abort*) che è gestita dal padre del processo.
- Si arriva a questa soluzione ad es. nei seguenti casi:
  - ☞ Il processo figlio ha ecceduto nell'uso delle risorse che gli erano state assegnate.
  - ☞ Il padre non ha più bisogno del compito che il figlio svolgeva.
  - ☞ Il padre termina e il sistema operativo non consente al figlio di continuare l'esecuzione.
  - ☞ La maggior parte dei sistemi non consentono a processi figli di esistere dopo la terminazione del padre,
    - 📄 quest'azione porta all'effetto di una *terminazione a cascata*.





# Processi cooperanti

- I processi concorrenti nel sistema operativo possono essere *indipendenti* o *cooperanti*.
- Un processo è *indipendente* se non può influire su altri processi, né può subirne l'influsso, durante la sua esecuzione.
- Un processo *cooperante* può influire su altri processi, e può subirne l'influsso, durante la sua esecuzione.





# Processi cooperanti (II)

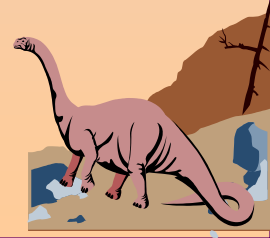
- Vantaggi della cooperazione tra processi:
  - ☞ Condivisione di informazioni:
    - 📄 più utenti possono essere interessati alle stesse informazioni.
  - ☞ Accelerazione del calcolo:
    - 📄 se vi sono più CPU o canali di I/O.
  - ☞ Modularità:
    - 📄 può essere più pratica ed efficiente la costruzione di un sistema modulare.
  - ☞ Convenienza:
    - 📄 anche un solo utente può avere la necessità di compiere più operazioni contemporaneamente.
- I modelli fondamentali della comunicazione tra processi sono due:
  - ☞ a memoria condivisa;
  - ☞ a scambio di messaggi.





# Il problema del produttore e del consumatore

- Paradigma per processi cooperanti
- Un processo produttore “produce” informazioni, che sono “consumate” da un processo consumatore.
- Necessità di un *buffer* su cui:
  - ☞ chi produce può immagazzinare informazioni,
  - ☞ chi consuma può prelevarle:
- Memoria illimitata (*unbounded-buffer*) non ci sono limiti alla dimensione del buffer
- Memoria limitata (*bounded-buffer*) dimensione fissata del buffer

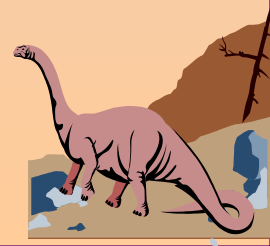




# Memoria limitata : soluzione con memoria condivisa

- Soluzione che permette di memorizzare (in *buffer*) fino a *BUFFER\_SIZE-1 elementi*.
- Il produttore e il consumatore condividono le seguenti variabili:

```
#define BUFFER_SIZE 10  
Typedef struct {  
    . . .  
} item;  
item buffer[BUFFER_SIZE];  
int in = 0;  
int out = 0;
```







# Memoria limitata : soluzione con memoria condivisa (II)

```
item Processo_prodotto;re;
```

```
while (1) {
```

```
    while (((in + 1) % BUFFER_SIZE) == out); /* non fa niente */
```

```
    buffer[in] = nextProduced;
```

```
    in = (in + 1) % BUFFER_SIZE;
```

```
}
```

```
item Processo_consumatore;
```

```
while (1) {
```

```
    while (in == out); /* non fa niente */
```

```
    nextConsumed = buffer[out];
```

```
    out = (out + 1) % BUFFER_SIZE;
```

```
}
```





# Sistemi a scambio di messaggi

- Attraverso le funzionalità della comunicazione tra processi i processi sono in grado di comunicare tra loro e di sincronizzare le proprie azioni:
  - ➡ Inter Process Communication - IPC.
- Un IPC fornisce almeno due operazioni, *send* e *receive* :
  - ➡ *send* (message) ,
  - ➡ *receive* (message) .
- I messaggi possono essere di dimensione fissa o variabile.
- Se  $P$  e  $Q$  vogliono comunicare tra loro, devono:
  - ➡ Stabilire il canale di comunicazione.
  - ➡ Scambiare messaggi via *send/receive*.
- Realizzazione del canale di comunicazione due aspetti:
  - ➡ realizzazione fisica (ad es. memoria condivisa, bus, etc.),
  - ➡ realizzazione logica.





# Implementazione

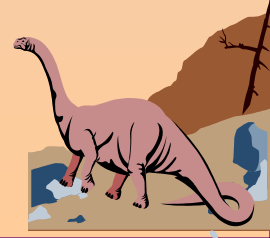
- Per implementare la comunicazione tra processi è importante sapere quanto segue:
  - ☞ Come vengono stabiliti i canali.
  - ☞ Se un canale può essere associato a due o più processi.
  - ☞ Quanti canali possono esistere tra ogni coppia di processi.
  - ☞ Conoscere la capacità di un canale.
  - ☞ Sapere se un canale può supportare messaggi a dimensione fissa o variabile.
  - ☞ Se la comunicazione sullo specifico canale è unidirezionale o bidirezionale.
- Esistono inoltre diversi modi per effettuare l'implementazione logica di un canale:
  - ☞ Comunicazione diretta o indiretta.
  - ☞ Comunicazione simmetrica o asimmetrica.
  - ☞ Capacità zero, limitata o illimitata della coda di messaggi.
  - ☞ Invio per copiatura o per riferimento.
  - ☞ Messaggi di dimensione fissa o di dimensione variabile.





# Comunicazione diretta

- I processi si devono *nominare* esplicitamente:
  - ☞ `send (P, messaggio)` – invia *messaggio* al processo P
  - ☞ `receive (Q, messaggio)` – ricevi, in *messaggio*, un messaggio dal processo Q
- Un canale di comunicazione ha le seguenti caratteristiche:
  - ☞ I canali sono stabiliti automaticamente.
  - ☞ Un canale è associato esattamente a due processi.
  - ☞ Esiste esattamente un canale tra ciascuna coppia di processi.
  - ☞ Il canale può essere unidirezionale, ma generalmente è bidirezionale.





# Comunicazione indiretta

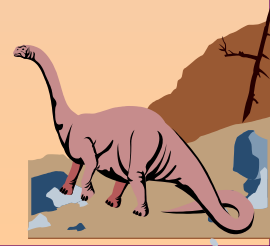
- I messaggi sono inviati e ricevuti tramite delle porte (mailboxes):
  - ☞ Ciascuna porta ha un identificativo unico.
  - ☞ I processi possono comunicare solo se hanno una porta in comune
- Proprietà dei canali di comunicazione:
  - ☞ Tra una coppia di processi si stabilisce un canale solo se entrambi i processi condividono una stessa porta.
  - ☞ Un canale può essere associato a più di due processi.
  - ☞ Tra ogni coppia di processi comunicanti può esserci un certo numero di canali diversi.
  - ☞ Un canale può essere unidirezionale o bidirezionale.
- Il S.O. offre un meccanismo che permette al processo di
  - ☞ Creare una nuova porta.
  - ☞ Inviare e ricevere messaggi tramite la porta.
  - ☞ Rimuovere la porta.





# Comunicazione indiretta (II)

- Le primitive di comunicazione sono:
  - send ( $A$ , *messaggio*) – manda un messaggio alla porta  $A$ .
  - receive ( $A$ , *messaggio*) – ricevi un messaggio dalla porta  $A$ .
- Condivisione delle porte:
  - ☞  $P_1$ ,  $P_2$ , e  $P_3$  condividono la porta  $A$ .
  - ☞  $P_1$  invia;  $P_2$  e  $P_3$  ricevono.
  - ☞ Quale processo riceverà il messaggio?
- Soluzioni:
  - ☞ Fare in modo che un canale sia associato al più a due processi.
  - ☞ Consentire a un solo processo alla volta di eseguire l'operazione di receive.
  - ☞ Consentire al sistema di scegliere arbitrariamente quale processo riceve il messaggio.
  - ☞ Il sistema comunicherà l'identità del ricevente al trasmittente.





# Sincronizzazione

- Lo scambio di messaggi può essere sincrono (o bloccante) oppure asincrono (o non bloccante).
- Le primitive send e receive possono essere sincrone o asincrone.





# Code di messaggi

- Un canale ha una capacità che determina il numero dei messaggi che possono risiedere al suo interno.
- Questa caratteristica può essere immaginata come una coda di messaggi legata al canale.
- Esistono tre tipi di code:
  1. **Capacità zero** - la coda ha capacità zero, per cui il canale non può avere messaggi in attesa al suo interno. Con questa specifica il trasmittente deve attendere che il ricevente abbia ricevuto il messaggio per inviarne un altro (rendezvous).
  2. **Capacità limitata** - la coda ha lunghezza finita  $n$ , quindi al suo interno non può contenere più di  $n$  messaggi. Finché la coda non è piena il trasmittente può inviare messaggi.
  3. **Capacità illimitata** - la coda ha lunghezza infinita per cui il trasmittente può sempre inviare messaggi e non resta mai in attesa.







# Comunicazione nei sistemi client-server

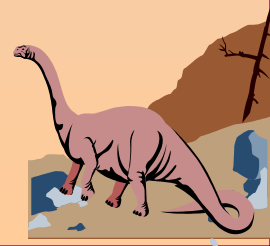
- Socket.
- Chiamate di procedure remote (Remote Procedure Calls - RPC).
- Invocazione di metodi remoti (Remote Method Invocation - RMI) in Java.





# Socket

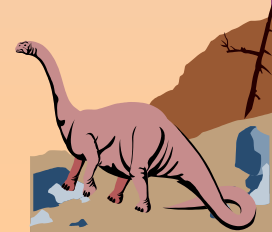
- Si consideri un utente che deve accedere a dati presenti in un altro server.
  - ✎ Ad es. vuole calcolare il numero di righe, parole e caratteri di un file presente in un server A.
  - ✎ Questa richiesta di esecuzione è gestita da un server remoto nel sito A, che accede al file, calcola i dati richiesti ed invia i dati all'utente.
- Un socket è definito come l'estremità di un canale di comunicazione.
- Una coppia di processi che comunica attraverso una rete usa una coppia di socket, una per ogni processo.
- Ogni socket è la concatenazione di un indirizzo IP e di una porta.
- Il socket **161.25.19.8:1625** fa riferimento alla porta **1625** sull'host **161.25.19.8** .



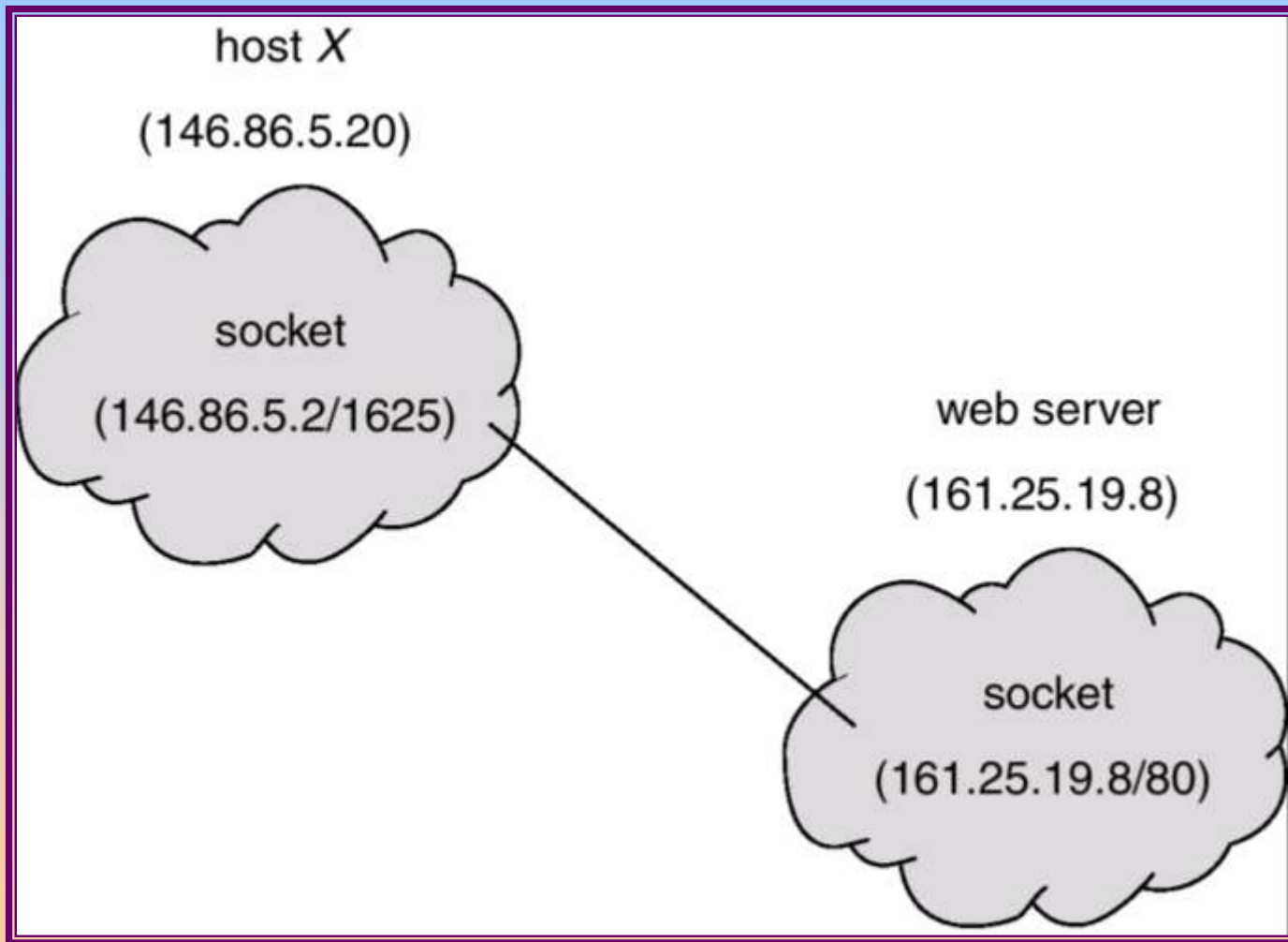


# Socket (II)

- In generale i socket impiegano una architettura client-server.
- Il server attende le richieste del client stando in ascolto ad una porta specificata.
- Quando riceve una richiesta, se accetta la connessione, si stabilisce la comunicazione.
- I server che eseguono servizi specifici (telnet, ftp, http, etc.) stanno in ascolto a porte note.
  - ☞ I server telnet alla porta 23.
  - ☞ I server ftp alla porta 21
  - ☞ I web server (http) alla porta 80.
- Tutte le porte sotto il valore 1024 sono considerate note e si utilizzano per realizzare servizi standard.



# Comunicazione tramite socket



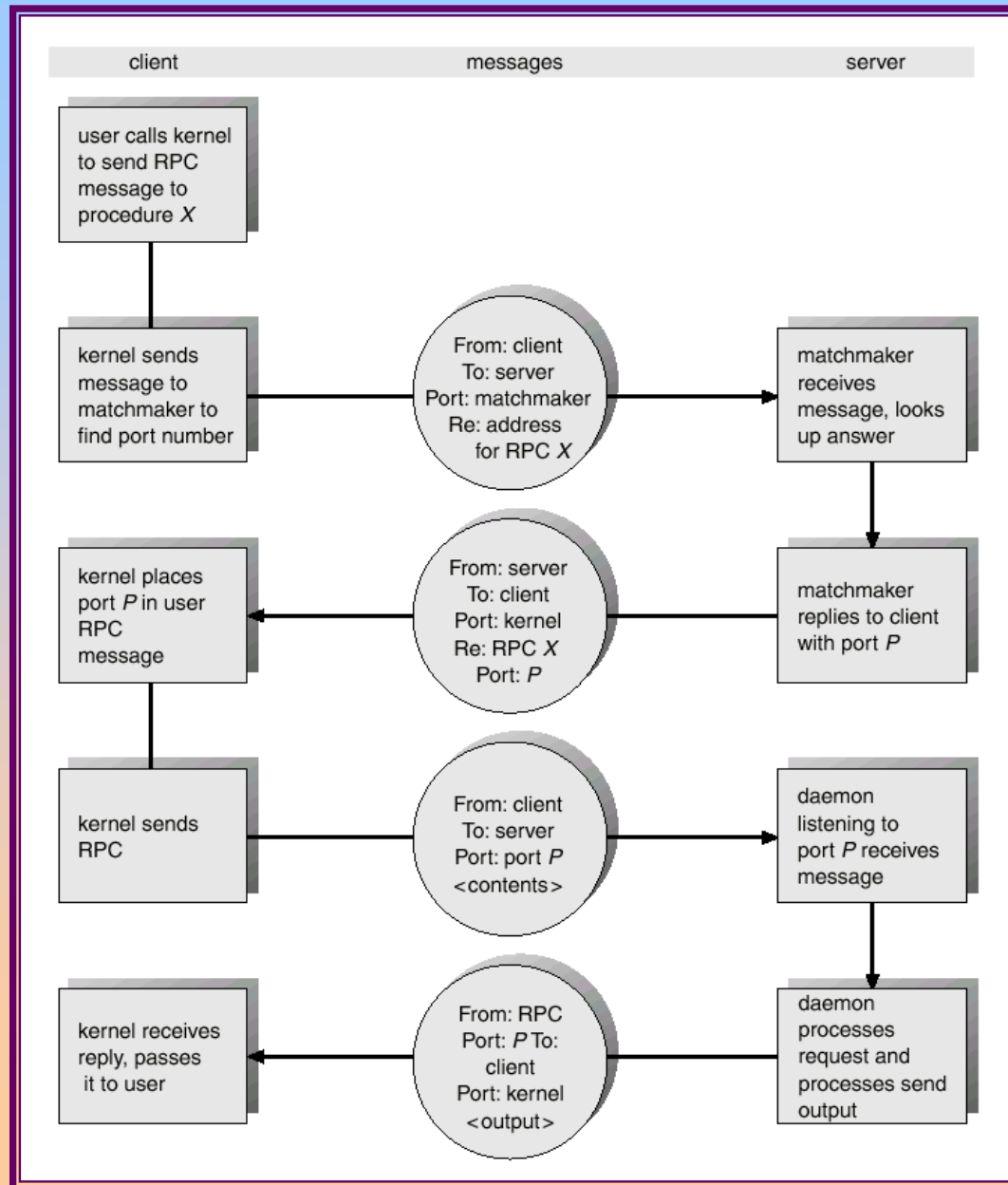


# Chiamate di procedure remote

- Le Remote procedure call (RPC) permettono di astrarre il meccanismo della chiamata di procedura affinché si possa usare tra sistemi collegati tramite una rete.
- **Segmento di codice di riferimento (stub)**, assegnato alla parte client:
  - ☞ quando il client invoca una procedura remota, RPC chiama l'appropriato stub passando i parametri della procedura remota,
  - ☞ lo stub individua la porta del server e struttura i parametri,
  - ☞ (*marshalling*: assemblaggio dei parametri in una forma che si possa trasmettere tramite una rete).
  - ☞ Lo stub quindi trasmette un messaggio al server usando lo scambio di messaggi.
- Un analogo segmento di codice di riferimento nel server riceve questo messaggio e invoca la procedura nel server e, ove necessario, riporta i risultati al client usando la stessa tecnica.

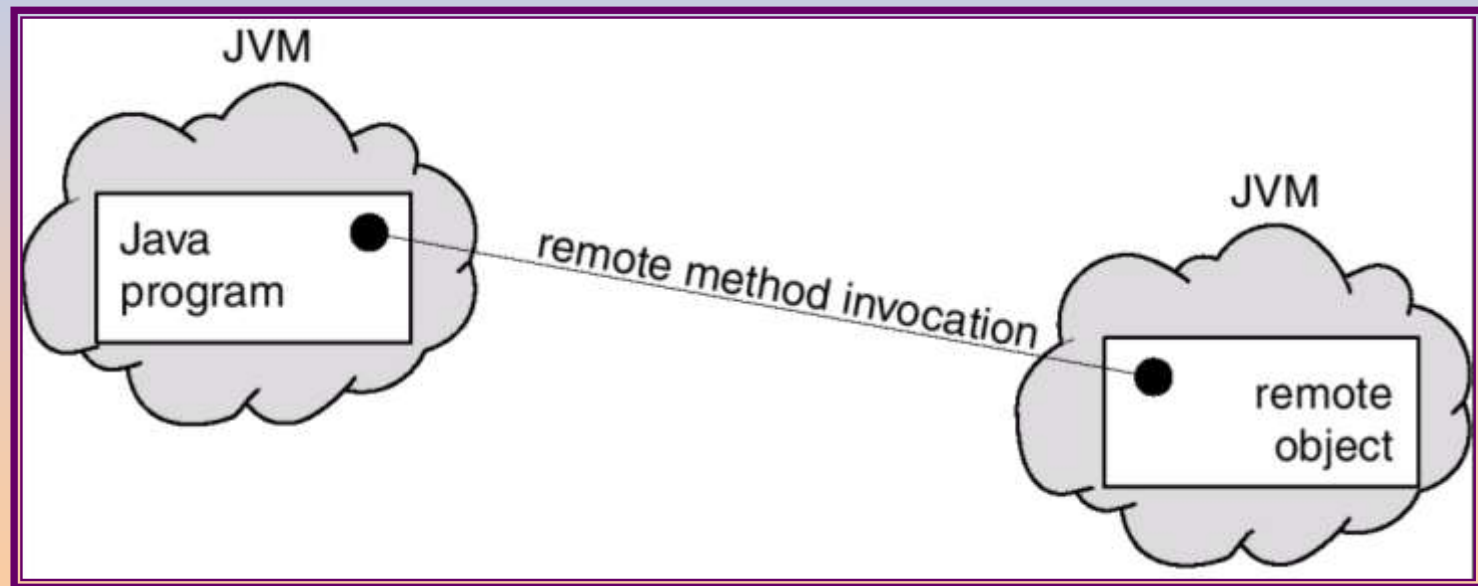


# Esecuzione di una RPC



# Invocazione di metodi remoti

- L'invocazione di metodi remoti (Remote Method Invocation - RMI) è un meccanismo Java simile alle RPC.
- RMI permette ad un programma Java su una macchina di invocare un metodo su un oggetto remoto.



# Strutturazione dei parametri

