

## II e III parte: Gestione dei Processi

- processi (cap. 3)
- Thread (cap. 4)
- Scheduling della CPU (cap. 5)
- Sincronizzazione fra processi (cap. 6 e 7)
- Deadlock (stallo dei processi) (cap. 8)

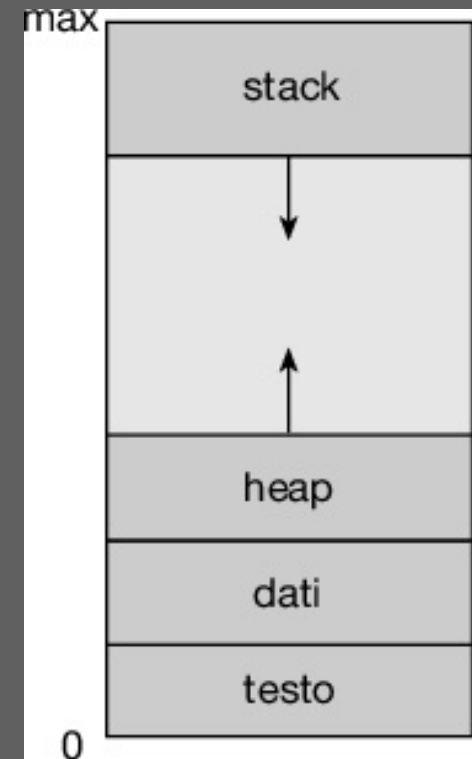
# 3. Processi

- Il **processo** è l'unità di lavoro del sistema operativo, perché ciò che fa un qualsiasi SO è innanzi tutto amministrare la vita dei processi che girano sul computer gestito da quel SO.
- Il sistema operativo è responsabile della **creazione** e **cancellazione** dei processi degli utenti, gestisce lo **scheduling dei processi**, fornisce dei **meccanismi di sincronizzazione e comunicazione** fra i processi

## 3.1.1 Concetto di Processo

3

- **Processo**  $\equiv$  programma in esecuzione, tuttavia:
- Un processo è più di un semplice programma, infatti ha una **struttura in memoria primaria** (in un'area assegnatagli dal SO) suddivisa in più parti (fig. 3.1):
  - **Codice da eseguire** (il “testo”) +
  - **dati** +
  - **stack** (per le chiamate alle procedure / metodi e il passaggio dei parametri) +
  - **heap** (la memoria dinamica)

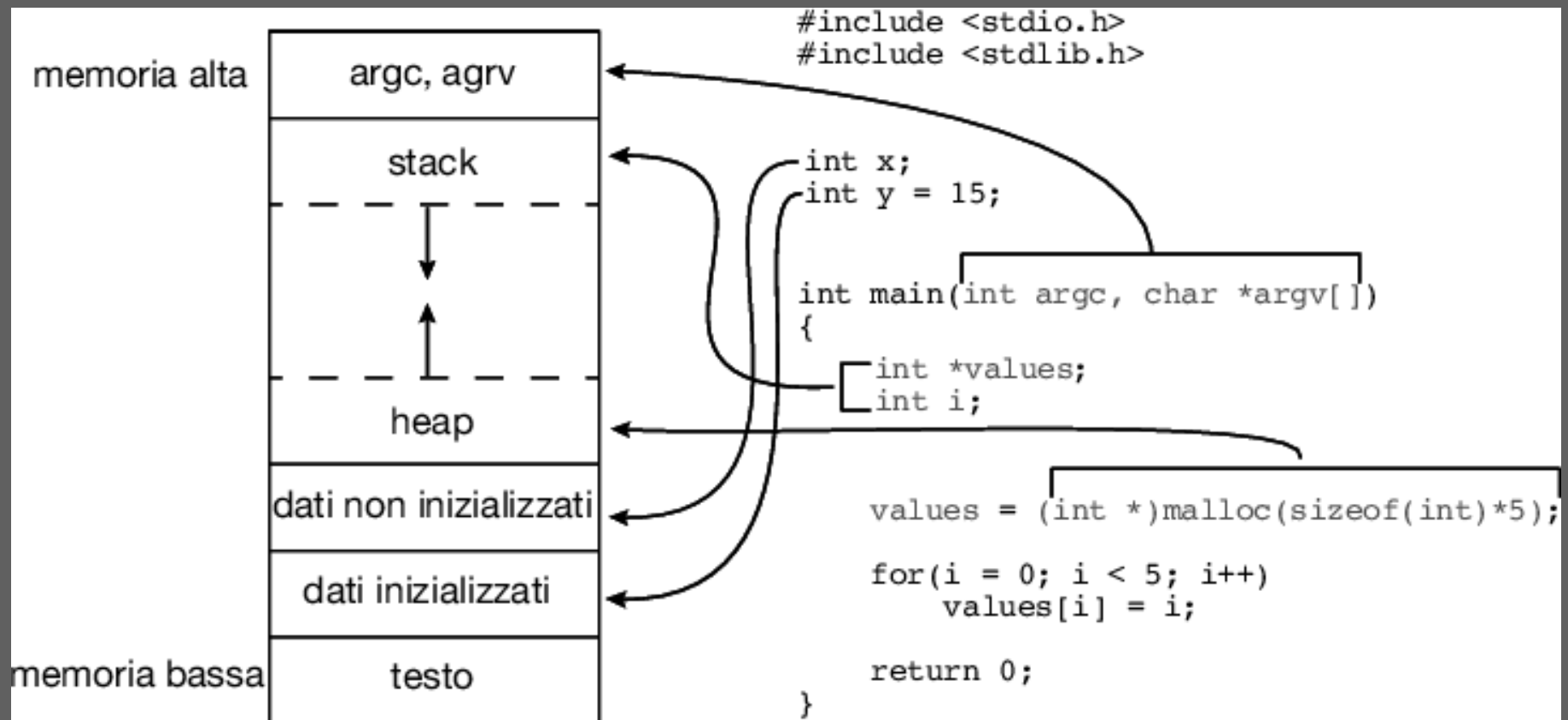


- codice + dati + stack + heap = **immagine del processo**

## 3.1.1 Concetto di Processo

4

- Ecco ad esempio più in dettaglio la struttura in memoria primaria di un processo che esegue del codice scritto in C. (figura 3.1a)



## 3.1.1 Concetto di Processo

- E' anche corretto osservare che attraverso un programma si possono definire più processi, infatti:
  - Lo stesso programma **può contenere codice per generare più processi**
  - Più processi possono **condividere lo stesso codice**
- Tuttavia, la distinzione fondamentale tra processo e programma è che **un processo è un'entità attiva, un programma è un'entità statica.**
- Lo stesso programma lanciato **due volte, può dare origine a due processi diversi** (perché?)
- Attenzione: **processo, task, job** sono sinonimi

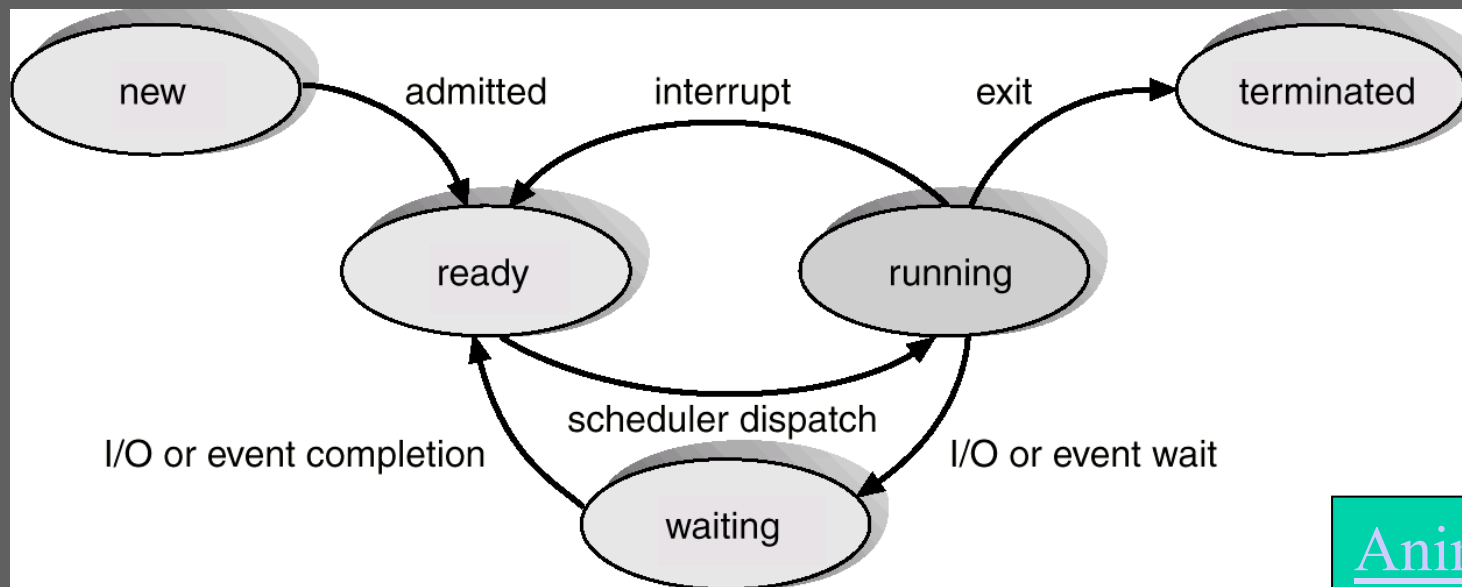
## 3.1.1 Concetto di Processo

- Un programma si trasforma in un processo quando viene lanciato, con il **doppio click o da riga di comando**.
- Un processo può anche nascere a partire da un altro processo, quando quest'ultimo esegue una opportuna system call
- In realtà, non sono due meccanismi distinti: **un processo nasce sempre a partire da un altro processo, e sempre sotto il controllo e con l'intervento del SO** (con un'unica eccezione, all'accensione del sistema).

## 3.1.2 Stato del processo

7

- Da quanto nasce a quando termina, un processo *passa la sua esistenza* muovendosi tra un insieme di stati, e in ogni istante ogni processo si trova in un ben determinato **stato**.
- Lo stato di un processo evolve a causa del codice eseguito e dell'azione del SO sui processi presenti nel sistema in un dato istante, secondo quanto illustrato dal **diagramma di transizione degli stati di un processo** (fig. 3.2).



Animazione

## 3.1.2 Stato del processo

- Gli stati in cui può trovarsi un processo sono:
- **New**: il processo è appena nato nel sistema, e il SO sta allestendo le strutture dati necessari per amministrarlo
- **Ready (to Run)**: il processo è pronto per entrare in esecuzione, quando sarà il suo turno
- **Running**: la CPU sta eseguendo codice del processo
- **Waiting**: il processo ha lasciato la CPU e attende il completamento di un evento
- **Terminated**: il processo è terminato, il SO sta recuperando le strutture dati e le aree di memoria liberate.



## 3.1.2 Stato del processo

9

- Il diagramma di transizione degli stati di un processo sintetizza una serie di possibili varianti del modo in cui un SO può amministrare la vita dei processi di un computer.
- Infatti, nel caso reale lo sviluppatore del SO dovrà decidere quali scelte implementative fare quando (ad esempio):
  - Mentre Px è running, **un processo** entra nello stato Ready to Run
  - Mentre Px è running, **un processo più importante** di Px entra nello stato Ready to Run.
  - Mentre Px è in stato Ready to Run, un processo più importante di Px entra nello stato Ready to Run
- Domanda: che significato ha eliminare l'arco "interrupt"?

1,1V  
1mP

## 3.1.3 Process Control Block (PCB)

10

- Per ogni processo il SO mantiene una struttura dati, il **Process Control Block**, che contiene le informazioni necessarie ad amministrare la vita di quel processo, tra cui: (fig. 3.3):
  - il numero del processo (o “Process ID”)
  - lo stato del processo (ready, waiting,...)
  - il contenuto dei registri della CPU salvati nel momento in cui il processo è stato sospeso (valori significativi solo quando il processo non è running)
  - gli indirizzi in RAM delle aree dati e codice del processo
  - i file e gli altri dispositivi di I/O correntemente in uso dal processo
  - Le informazioni per lo scheduling della CPU (ad esempio, quanta CPU ha usato fino a quel momento il processo).

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

## 3.2 Scheduling dei Processi

- Conosciamo già i seguenti due concetti:
  - **Multiprogrammazione**: avere sempre un processo running  $\Rightarrow$  massima utilizzazione della CPU
  - **Time Sharing**: distribuire l'uso della CPU fra i processi a intervalli prefissati. Così più utenti possono usare “allo stesso tempo” la macchina, e i loro processi procedono in “parallelo” (notate sempre le virgolette)
- per implementare questi due concetti, **il SO deve decidere periodicamente** quale sarà il prossimo processo a cui assegnare la CPU. Questa operazione è detta **Scheduling**

## 3.2 Scheduling dei Processi

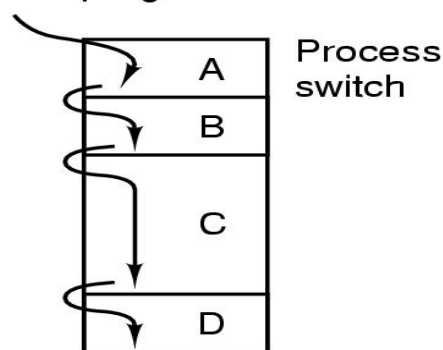
- In un sistema time sharing single-core, attraverso lo scheduling, ogni processo “crede” di avere a disposizione una macchina “tutta per se”...
- Ci pensa il SO a farglielo credere, commutando la CPU fra i processi (ma succede la stessa cosa in un sistema ad n-core se ci sono più di n processi attivi contemporaneamente)

ciò che succede  
in realtà

ciò che vede ogni  
singolo processo

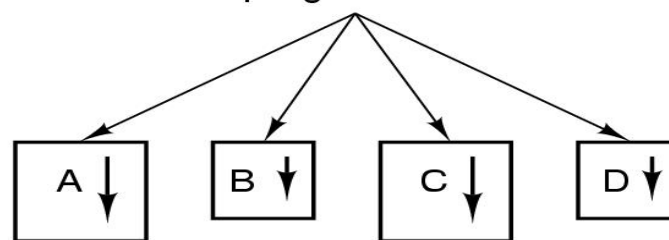
il risultato finale

One program counter

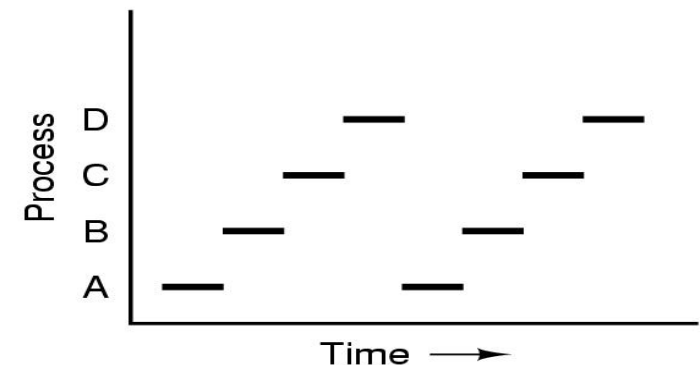


(a)

Four program counters



(b)



(c)

## 3.2.3 il cambio di contesto (context switch)

- Per commutare la CPU tra due processi il SO deve:
  1. **Riprendere il controllo della CPU** (ad esempio attraverso il meccanismo del **Timer** visto nel capitolo 1)
  2. Con l'aiuto dell'hardware della CPU, **salvare** lo stato corrente della computazione del processo che lascia la CPU, ossia, copiare il valore del PC e **degli altri registri nel suo PCB**
  3. **Scrivere** nel PC e nei registri della CPU i valori relativi contenuti **nel PCB del processo utente scelto** per entrare in esecuzione
- Questa operazione prende il nome di:  
“**cambio di contesto**”, o “**context switch**”
- notate che, tecnicamente, anche il punto 1 è già di per se un context switch

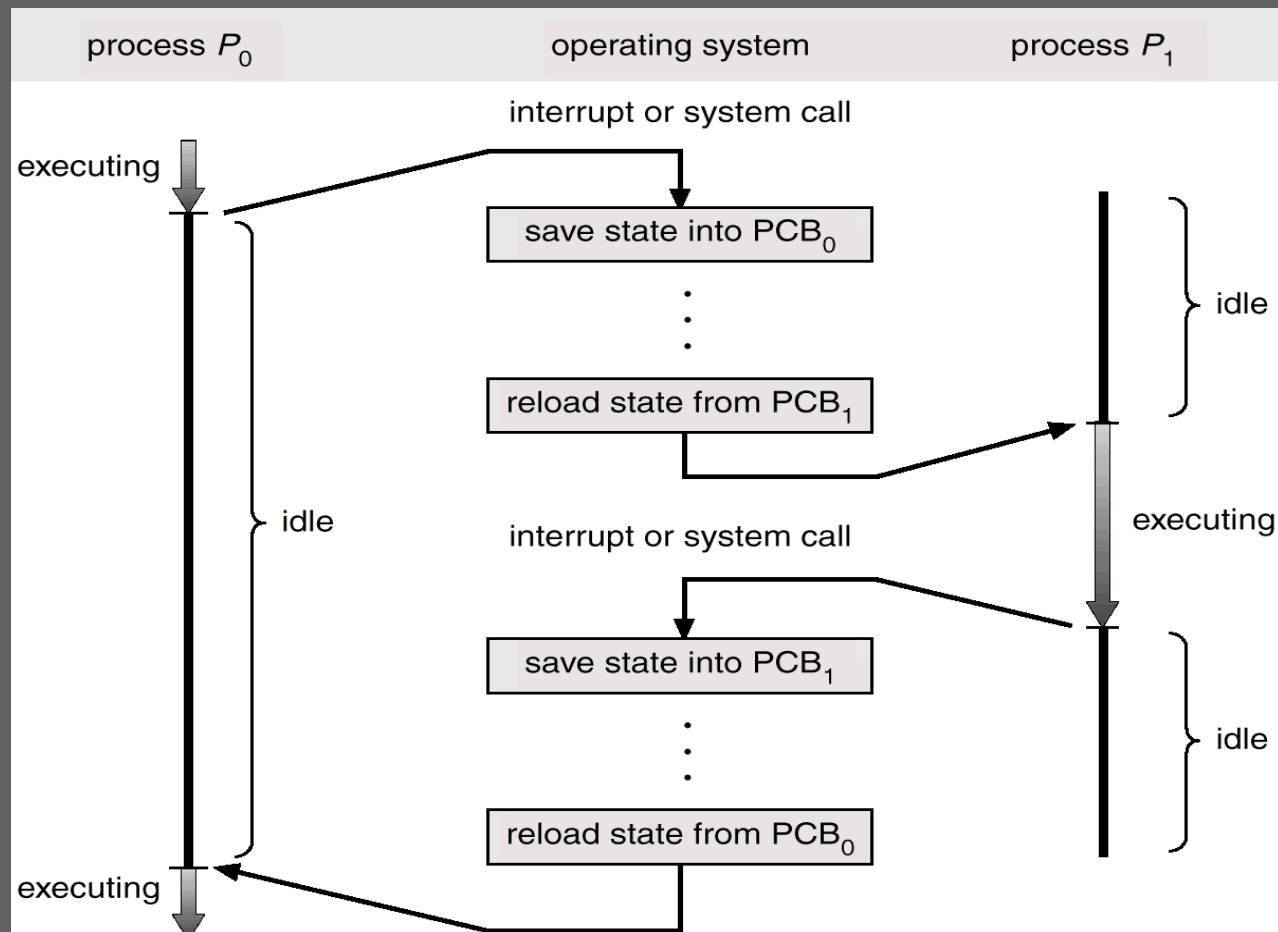
## 3.2.3 il cambio di contesto (context switch)

- Il context switch richiede tempo, perché il **contesto di un processo** è fatto di molte informazioni (alcune le vedremo quando parleremo della gestione della memoria): in questa frazione di tempo la macchina non è usata da nessun processo utente
- In generale **il context switch costa da qualche centinaio di nanosecondi a qualche microsecondo**
- questo tempo “sprecato” è un **overhead** (sovraccarico) per il sistema, e ne influenza le prestazioni.
- (nei prossimi capitoli scopriremo molte altre forme di overhead introdotte dal SO)

## 3.2.3 il cambio di contesto (context switch)

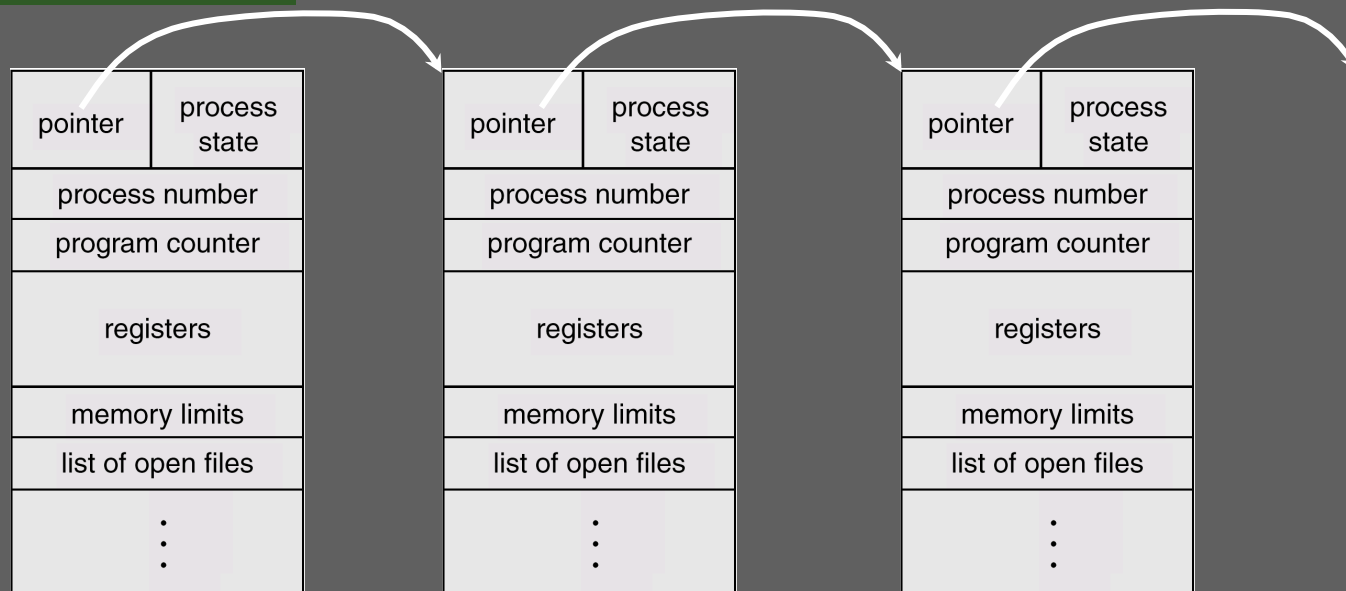
15

- Fasi dello scheduling tra un processo e un altro (fig. 3.6):



## 3.2.1 Code di scheduling

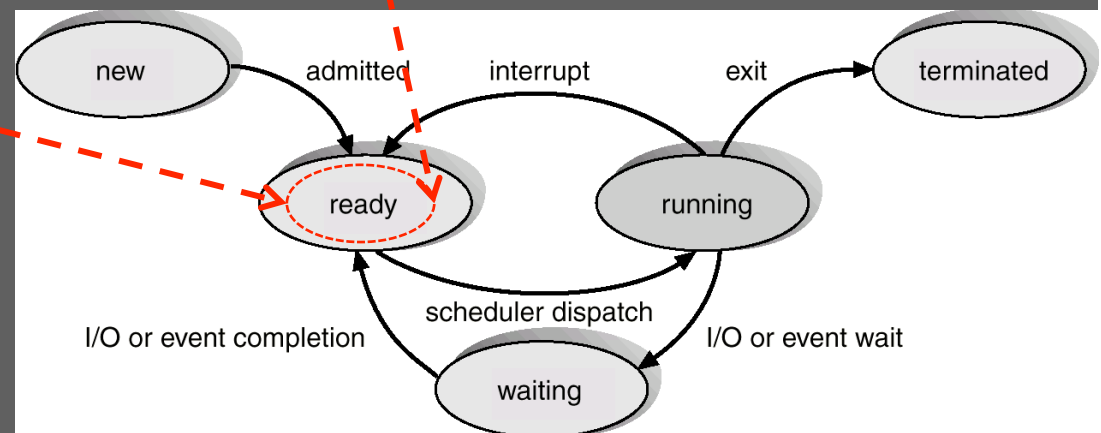
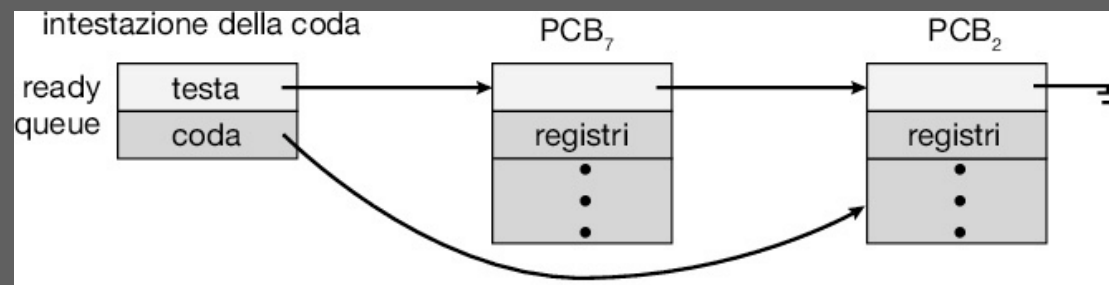
- Per amministrare la vita di ciascun processo, il SO gestisce varie **code di processi**. Ogni processo “si trova” in una di queste code, a seconda di cosa sta facendo.
- Una coda di processi **non è altro che una lista di PCB**, mantenuta in una **delle aree di memoria primaria che il SO riserva a se stesso**





## 3.2.1 Code di scheduling: coda di ready

- La coda dei processi più importante è la **coda di ready**, o **ready queue (RQ)**: l'insieme dei processi **ready to run**.
- Dunque, la RQ coincide con lo stato **ready** nel diagramma di transizione degli stati di un processo (fig. 3.4a)

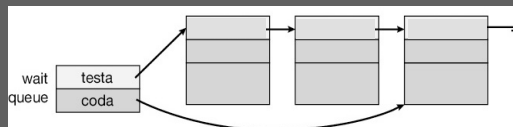
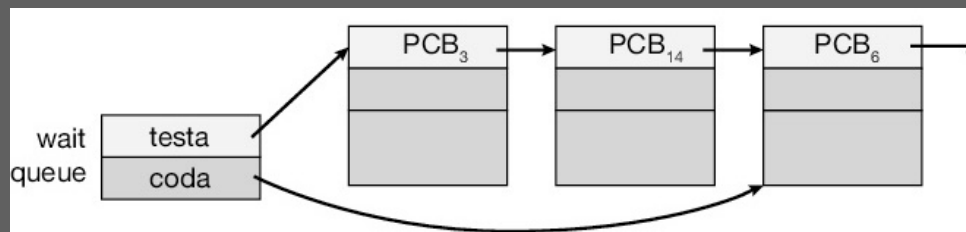


## 3.2.1 Code di scheduling

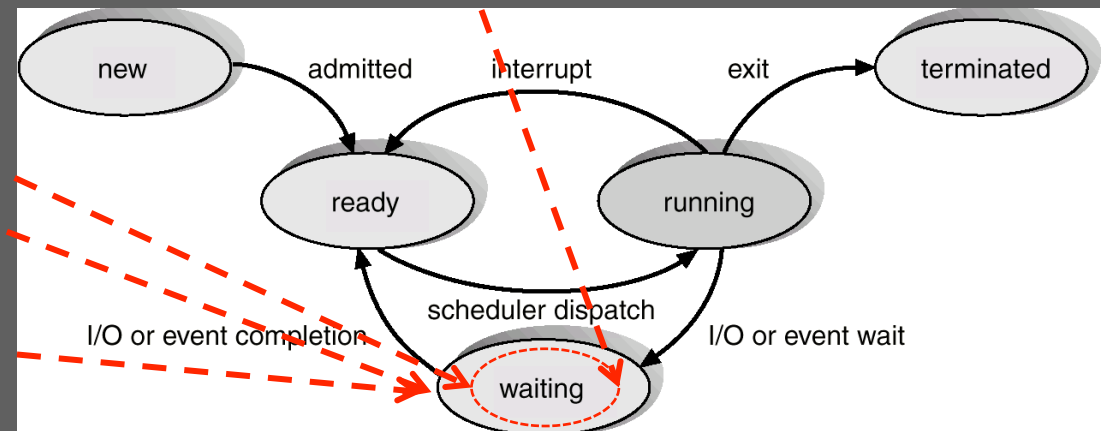
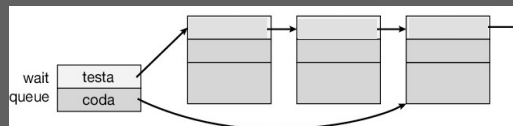
- Quando un processo rilascia la CPU, ma non termina e non torna in ready queue, vuol dire che si è messo in attesa di “qualcosa”, e il SO lo “parcheggia” in una tra  $n$  possibili code, che possiamo dividere in due grandi categorie:
- **device queues:** Code dei processi in attesa per l'uso di un dispositivo di I/O. Una coda per ciascun dispositivo.
- **Code di waiting:** code di processi in attesa che si verifichi un certo evento. Una coda per ciascun evento (ci torneremo nella sezione 6.6)
- Dunque, durante la loro vita, i processi si spostano (meglio: il SO sposta i corrispondenti PCB) tra le varie code

## 3.2.1 Code di scheduling

- Quindi, lo stato **waiting** nel diagramma di transizione degli stati di un processo **corrisponde a più di code di attesa**:
  - del completamento di una operazione di I/O (device queues)
  - del verificarsi di un evento atteso (waiting queues) (fig. 3.4b)



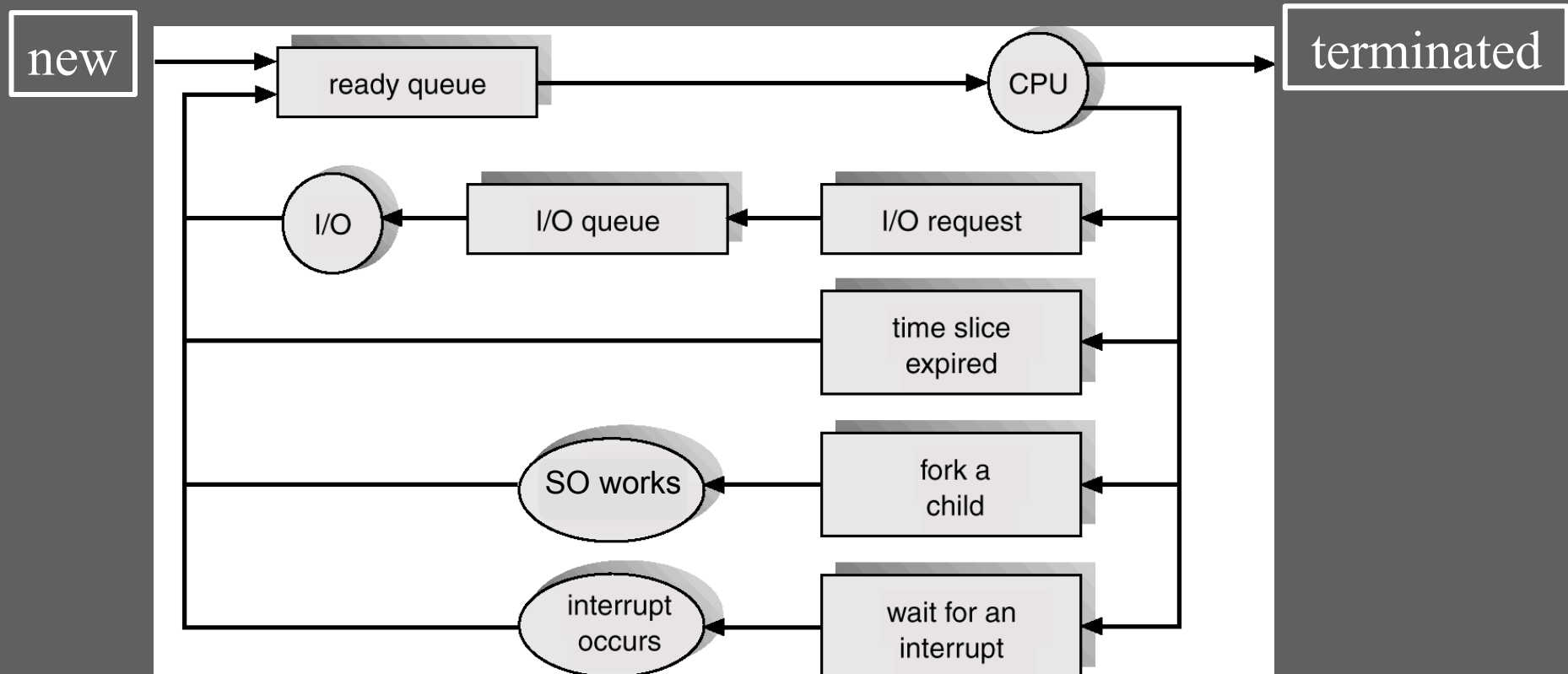
...



## 3.2.1 Code di scheduling

20

- Possiamo riformulare il diagramma di transizione degli stati di un processo come un **diagramma di accodamento** in cui i processi si muovono fra le varie code (fig. 3.4 modificata: il caso *wait for an interrupt* lo vediamo nel capitolo 6)



## 3.2.2 CPU Scheduler

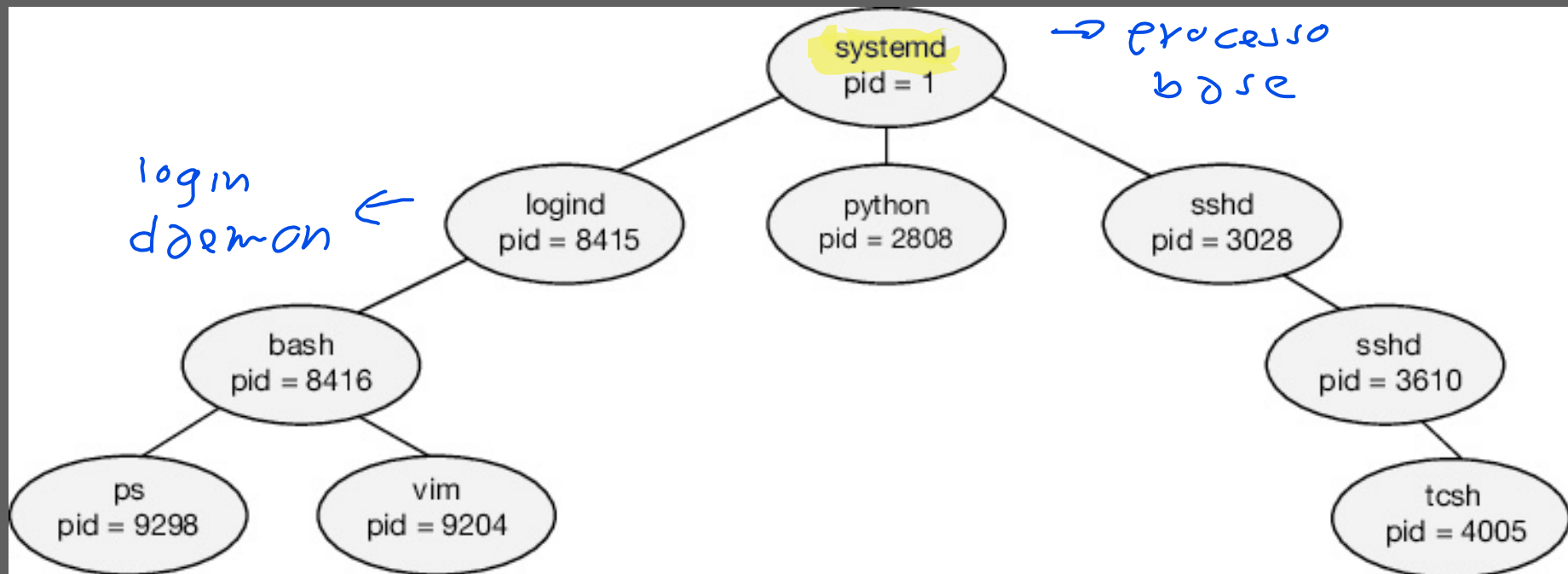
- Un componente del Sistema Operativo detto **(CPU) Scheduler** sceglie uno dei processi in coda di ready e lo manda in esecuzione
- Il CPU scheduler si attiva ogni **50/100 millisec**, ed è quello che produce l'effetto *time sharing*
- Per limitare l'overhead, deve essere molto veloce.
- Il CPU scheduler è anche chiamato **Short Term Scheduler**.
- Quali criteri di scelta usa, e quando interviene? Lo vedremo nel capitolo 5.

## 3.3 Operazioni sui processi

- la creazione di un processo è di gran lunga l'operazione più importante all'interno di qualsiasi sistema operativo
- Ogni SO possiede almeno una System Call di creazione processi, e ogni processo è creato a partire da un altro processo usando la system call relativa (eccetto il processo che nasce all'accensione del sistema)
- Il processo “creatore” è detto **processo padre (o parent)**
- Il processo creato è detto **processo figlio (o child)**
- Poichè ogni processo può a sua volta creare altri processi, nel sistema **va formandosi un “albero di processi”**

## 3.3.1 Creazione di un processo

- Un esempio di albero di processi in Linux (fig. 3.7):



## 3.3.1 Creazione di un processo

- Quando nasce un nuovo processo, il SO:
  - gli assegna un **identificatore del processo** unico, **un numero intero detto pid (process-id)**. E' il modo con cui il SO conosce e si riferisce a quel processo.
  - **recupera dall'hard disk** il codice da eseguire e lo porta in RAM (a meno che il codice non sia già in RAM)
  - **alloca un nuovo PCB** e lo inizializza con le informazioni relative al nuovo processo
  - **inserisce il PCB in coda di ready**.



## 3.3.1 Creazione di un processo

- Nello scrivere il codice della system call che genera un nuovo processo, il progettista del SO che userà quella system call deve fare alcune scelte implementative:



### 1. **Che cosa fa il processo padre quando ha generato un processo figlio?**

- prosegue la sua esecuzione in modo concorrente all'esecuzione del processo figlio, oppure:
- si ferma, in attesa del completamento dell'esecuzione del processo figlio.

## 3.3.1 Creazione di un processo

### 2. **Quale codice esegue il processo figlio?**

- al processo figlio viene data una copia del codice e dei dati in uso al processo padre, oppure:
  - al processo figlio viene dato un nuovo programma, con eventualmente nuovi dati.
- 
- In realtà, di solito, le system call dei moderni SO permettono di implementare una qualsiasi combinazione delle quattro modalità elencate.
  - (prossimo lucido: fig. 3.8 modificata)

## 3.3.1 creazione di un processo in Unix

*Passo usare*  
*int*

- `int main(){`
- *str. dati* `pid_t pid, childpid;`
- `pid = fork();` /\* genera un nuovo processo \*/ *da esso*

*padre*

```
printf("questa la stampano padre e figlio");  
if (pid == 0)
```

*figlio*

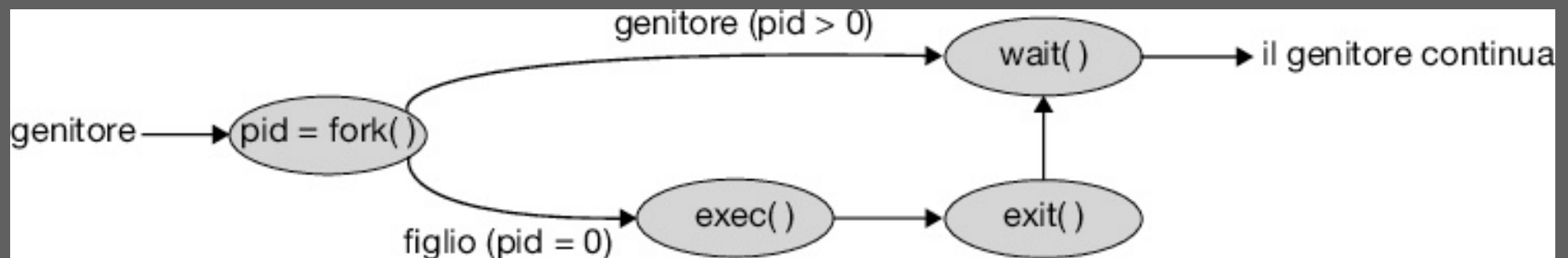
```
{ /* processo figlio */  
printf("processo figlio");  
execlp("/bin/ls", "ls", NULL);}
```

*padre*

```
else { /* processo padre */  
printf("sono il padre, aspetto il figlio");  
childpid = wait(NULL);  
printf("il processo figlio è terminato");  
exit(0);}}
```

## 3.3.1 Creazione di un processo

- Uno schema del funzionamento del codice del lucido precedente (fig. 3.9).
- Come cambia lo schema se il processo padre non esegue la wait?



## 3.3.1 creazione di un processo in Unix

- `int main(){ /* un altro esempio */`
- `int a, b, c = 57;`
- `a = fork(); // genera un nuovo processo`
- `printf("questa la stampano padre e figlio");`
- `if (a == 0)`
- `{ /* processo figlio */`
- `c = 64; // ***`
- `printf("c = %d",c);}`
- `else { /* processo padre */`
- `printf("c = %d",c);`
- `b = wait(NULL);`
- `printf("b = %d",b);}`
- `}`

## 3.3.1 creazione di un processo in Unix



## 3.3.1 Osservazioni

- In realtà solo lo spazio dei dati del processo padre viene duplicato per il figlio. Il codice viene condiviso (vedremo come il SO faccia funzionare tutto ciò nel capitolo 9)
- La ragione fondamentale è che è inutile duplicare la stessa informazione due (o più) volte, si spreca spazio in RAM.
- Il fatto che lo spazio dei dati del padre sia invece davvero duplicato per il figlio implica che:
  1. La modifica di una variabile da parte di uno dei due processi non è vista dall'altro processo
  2. Se uno dei due processi dichiara una nuova variabile dopo la fork l'altro processo non la vede nemmeno

## 3.3.1 Osservazioni

- **Un padre può chiamare la fork più volte:** il valore restituito da ogni chiamata di fork al padre (il PID del figlio appena creato) può essere usato per tenere traccia di quali sono i suoi processi figli.
- La fork restituisce invece il **valore 0 ad un figlio appena creato proprio** per poterlo distinguere dal padre.
- Se la fork restituisse al processo appena creato un valore maggiore di 0 (ad esempio il suo PID) come potremmo implementare il meccanismo che abbiamo visto nella figura 3.8 per far fare qualcosa di diverso a padre e figlio?



## 3.3.2 Terminazione di un Processo

- Un processo termina **dopo l'esecuzione dell'ultima** istruzione del suo codice
- Spesso è anche disponibile una opportuna system call, di **solito chiamata `exit()`**
- Dati di output del processo terminato (in particolare il suo pid) possono essere **inviati al padre, se questo è in attesa per la terminazione del figlio**
- Il SO provvede a rimuovere le risorse che erano state allocate al processo terminato, in particolare recupera la porzione di RAM usata dal processo e chiude eventuali file aperti

## 3.3.2 Terminazione di un Processo

- Di solito, un processo può anche **uccidere** esplicitamente un altro processo appartenente allo stesso utente con una opportuna system call **kill** (in unix) o **TerminateProcess** (in Win32)
- In alcuni casi, il sistema operativo stesso può decidere di uccidere un processo utente se:
  - **il processo sta usando troppe risorse**
  - **il suo processo padre è morto. In questo caso si verifica una terminazione a cascata (*attenzione, questo non è il caso dello Unix o di Windows*).**

## 3.4 Comunicazione tra processi

- I processi attivi in un sistema concorrono all'uso delle sue risorse. Due o più processi sono poi fra loro:
- **indipendenti**: se non si influenzano esplicitamente l'un l'altro durante la loro esecuzione.
- **cooperanti**: se si influenzano l'un l'altro, allo scopo di:
  - scambiarsi informazioni
  - portare avanti una elaborazione che è stata suddivisa tra i vari processi per ragioni di efficienza e/o di modularità nella progettazione dell'applicazione.
- La presenza di processi cooperanti richiede che il SO metta a disposizione meccanismi di **comunicazione e sincronizzazione** (ne ripareremo nel cap. 6)

## 3.5. Esempio: il problema del Produttore - Consumatore

36

Confo  
vedo  
dopo

- Un classico problema di processi cooperanti:
- un processo **produttore** produce informazioni che sono consumate da un processo **consumatore**; le informazioni sono **poste in un buffer** di dimensione limitata.
- Un esempio reale di questo tipo di situazione è quella in cui **un processo compilatore (il produttore) compila dei moduli producendo del codice assembler.**
- I moduli in assembler devono essere tradotti in linguaggio **macchina dall'assemblatore (il consumatore)**
- L'assemblatore potrebbe poi fare da *produttore* per un eventuale modulo che carica in RAM il codice.

## 3.5. Produttore - Consumatore

- `#define SIZE 10`
- `typedef struct {...} item;`
- `item buffer [SIZE];` (shared array)
- `int in = 0, out = 0;` (shared variables `[0..SIZE-1]`)
- Buffer circolare di SIZE **item** con due puntatori **in** e **out**
  - **valore corrente di in**: prossimo item libero;
  - **valore corrente di out**: primo item pieno;
  - **condizione di buffer vuoto**: `in=out`;
  - **condizione di buffer pieno**: `in+1 mod SIZE = out`
- Notate: la soluzione usa solo SIZE-1 elementi...

## 3.5. Produttore - Consumatore

### PRODUTTORE:

item nextp;

repeat

*<produci un nuovo item in nextp>*

while (in+1 mod SIZE == out) do no\_op; */\* buffer full \*/*

buffer[in] = nextp;

in = in+1 mod SIZE;

until false;

## 3.5. Produttore - Consumatore

### CONSUMATORE:

item nextc;

repeat

    while (in == out) do no\_op; */\*buffer empty\*/*

    nextc = buffer[out];

    out = out+1 mod SIZE;

    <consuma l'item in nextc>

until false;

## 3.5 Inter-Process Communication (IPC)

- Ma come fanno due processi a scambiarsi le informazioni necessarie alla cooperazione?
- Il SO mette a disposizione dei meccanismi di **Inter-Process Communication (IPC)**
- opportune system call che permettono a due (o più) processi di:
  - scambiarsi dei **messaggi** o di
  - usare la stessa area di memoria primaria in cui scrivono e leggono: la **memoria condivisa**

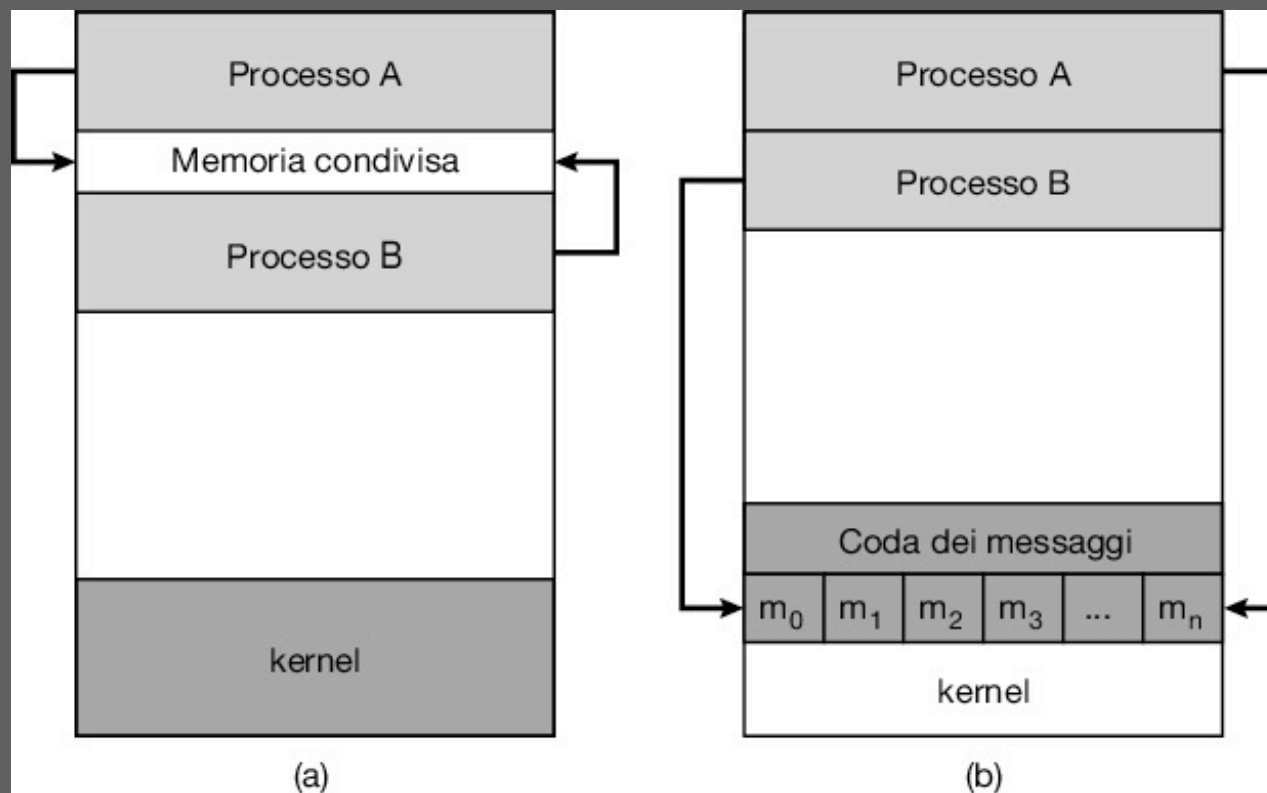


## 3.5 Inter-Process Communication (IPC)

- I due meccanismi fondamentali dell'IPC sono quindi la memoria condivisa e lo scambio di messaggi (fig. 3.11)

memoria condivisa

scambio di messaggi



## 3.5 Inter-Process Communication (IPC)

- In entrambi i casi il SO mette a disposizione delle opportune system call. Ad esempio, per lo scambio di messaggi, saranno disponibili delle system call del tipo:
  - *line = msgget();*
  - *send(message,line,process-id)*
  - *receive(message,line,process-id)*
- (ATT: i parametri sono solo indicativi, ogni specifica implementazione avrà il proprio insieme di argomenti)

## 3.5 Inter-Process Communication (IPC)

- E saranno necessarie alcune scelte implementative. Nel caso dei **messaggi** (si parla di solito di *code di messaggi*):
  - Una coda può essere usata da più di due processi?
  - Quanti messaggi può ospitare al massimo una coda?
  - Cosa deve fare un processo ricevente se non ci sono messaggi, o un processo trasmittente se la linea è piena?
  - Si possono trasmettere messaggi di lunghezza variabile?
- Nel caso della **memoria condivisa**:
  - può avere dimensione variabile?
  - quali processi hanno diritto di usarla?
  - che succede se la memoria condivisa viene rimossa?

# Per chi vuole approfondire

- Sezione 3.7.4: Pipe
- Sezione 3.8: Comunicazione nei sistemi client-server: socket ed RPC (Remote Procedure Call)