



procedi

capitolo 3 e 5 del libro (VII ed.)

Processi

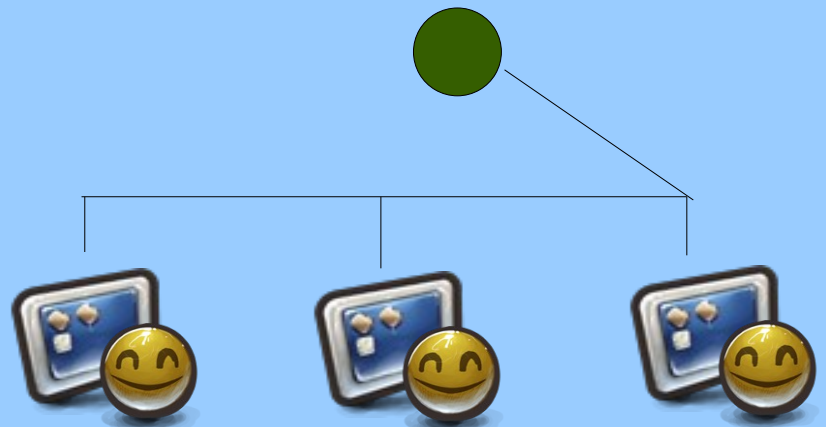


sistema batch



la CPU esegue i diversi job
uno di seguito all'altro

il SO deve mantenere informazioni riguardo i diversi task: ogni task esegue un programma, elabora dei dati, ha un utente "proprietario", può avere una priorità. Un task viene **interrotto** e **ripreso**: occorre mantenere tutte le informazioni necessarie



sistema time-sharing:

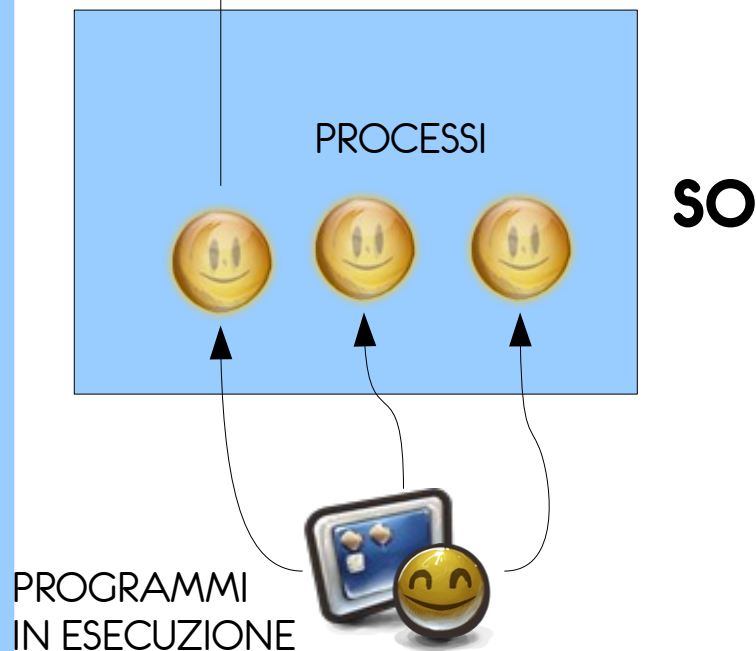
il tempo di CPU è diviso fra i task di più utenti collegati tramite terminali diversi

Processi

- programma (o **sezione testo**)
- program counter (istruzione da eseguire)
- stack di esecuzione (con vrb, parametri, ind. di ritorno)
- heap (memoria allocata dinamicamente)

**SEZIONE
DATI**

NB: ogni processo ha la propria sezione dati e non gli è consentito leggere o modificare le sezioni dati di altri processi!!!



“processo” è un’astrazione, una rappresentazione interna al SO, che consente di pensare e realizzare meccanismi quali multi-tasking, scheduling della CPU, protezione

Esempio

Quanti processi erano in esecuzione sul mio portatile quando scrivevo queste slide?

slidesSisOp0607.odp - OpenOffice.org Impress

File Edit View Insert Format Tools Slide Show Window Help

Normal Outline Notes Handout Slide Sorter

1 2 3 4 5 6 7 8 9 10 11 12

processi

- programma (o sezione testo)
- program counter (istruzione da eseguire)
- stack di esecuzione (con vrb, parametri, ind. di ritorno)
- heap (memoria allocata dinamicamente)

SEZIONE DATI

NB: ogni processo ha la propria sezione dati e non gli è consentito leggere o modificare le sezioni dati di altri processi!!!

PROCESSI

SO

PROGRAMMI IN ESECUZIONE

"processo" è un'astrazione, una rappresentazione interna al SO, che consente di pensare e realizzare meccanismi quali multi-tasking, scheduling della CPU, protezione

Cristina Baroglio
S.S. 2006/2007

Tas View x

Master Pages

Layouts

Custom Animation

Slide Transition

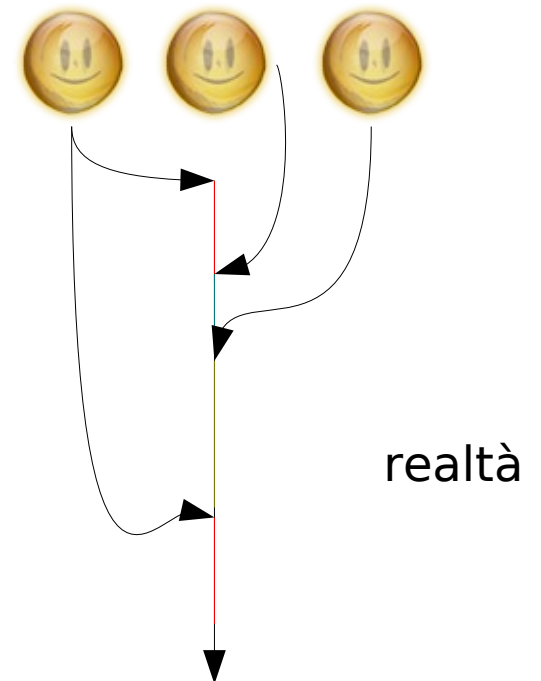
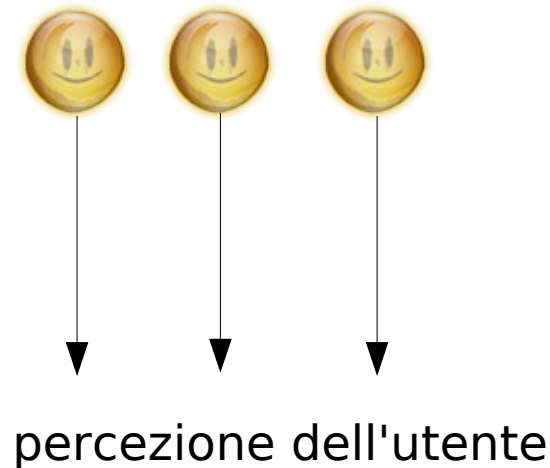
8.00 / 4.64 0.00 x 0.00 59% * Slide 72 / 73 Default

103

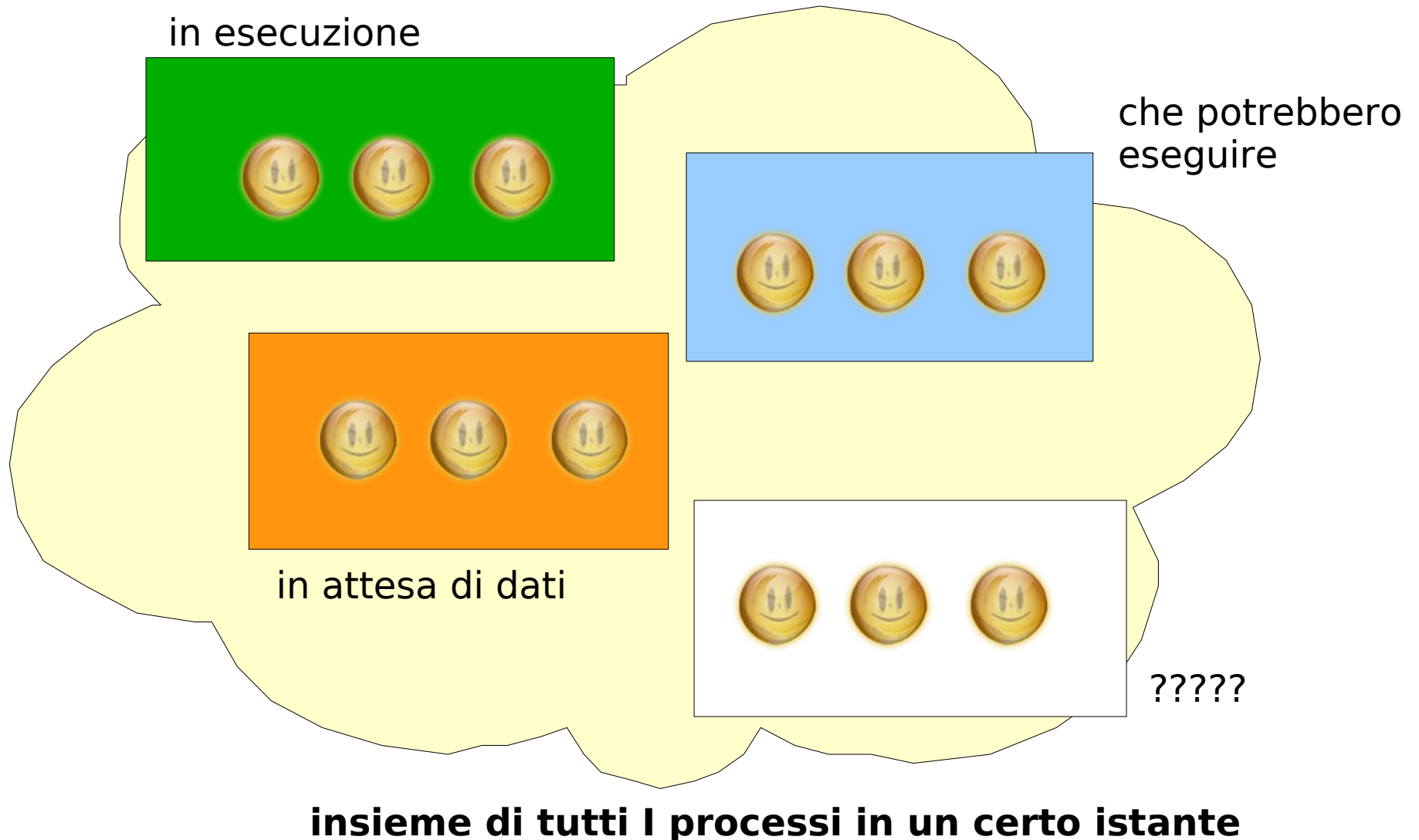
[illegible]

Parallelismo virtuale

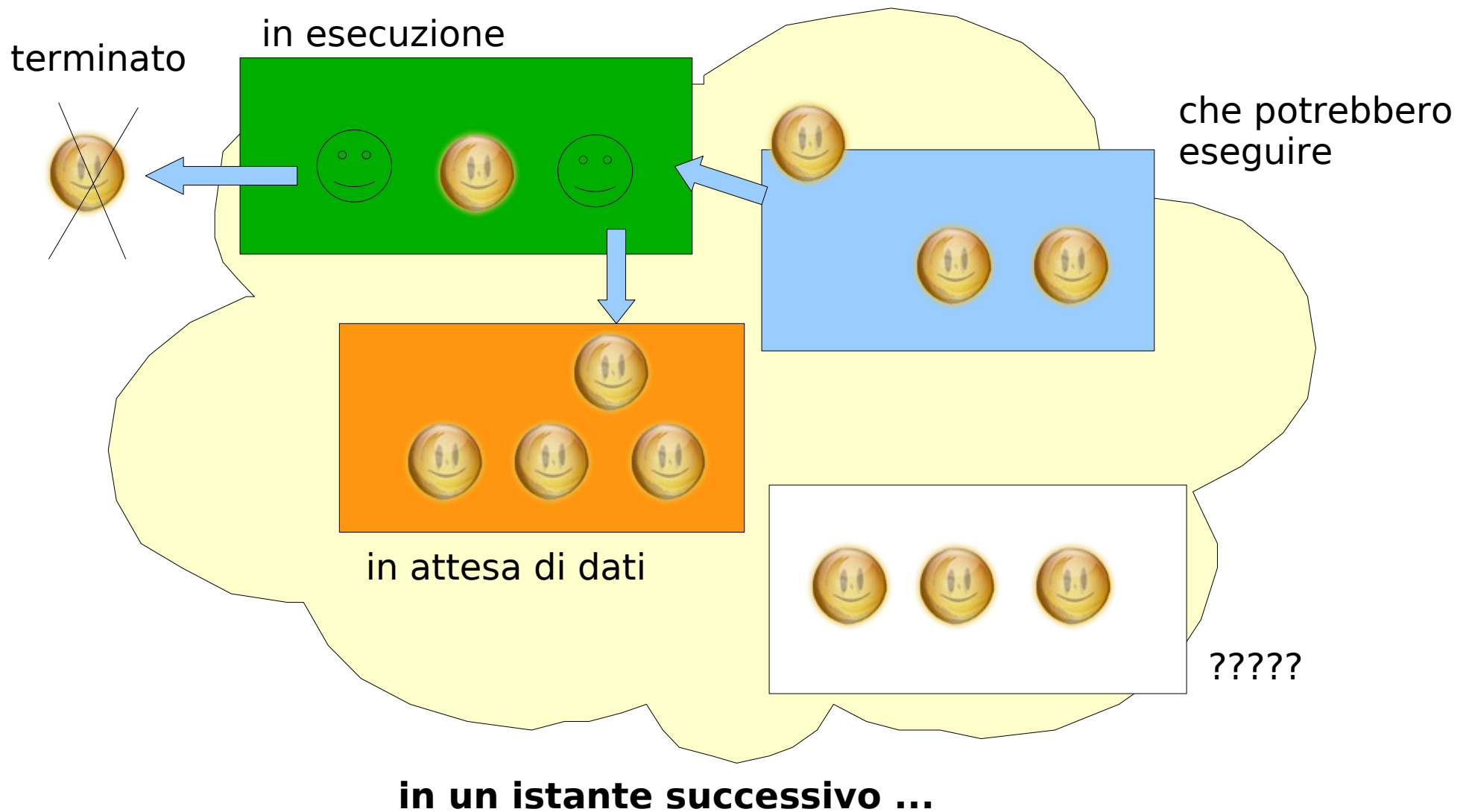
- Attraverso al SO i processi si suddividono l'uso delle risorse in modo tale portare avanti tutti quanti insieme la propria computazione
- Al più un processo per CPU può, in realtà, essere attivo in ogni istante ma gli utenti non se ne accorgono, percepiscono le diverse esecuzioni come parallele



Categorie di processi



I processi cambiano stato



Parallelismo virtuale

- In un contesto in cui la CPU è una sola e i processi sono tanti (anche centinaia o migliaia) nasce l'esigenza di associare un'**informazione di stato** a ogni processo:
 - 1)**nuovo**, è lo stato di un processo appena creato
 - 2)**running**, in esecuzione
 - 3)**waiting**, in attesa di un evento (es. completamento di un'operazione di I/O)
 - 4)**ready**, il processo è pronto per essere eseguito ma al momento non ha assegnata la CPU
 - 5)**terminato**, ha cessato l'esecuzione

Diagramma di transizione

- Diagramma di transizione degli stati di un processo

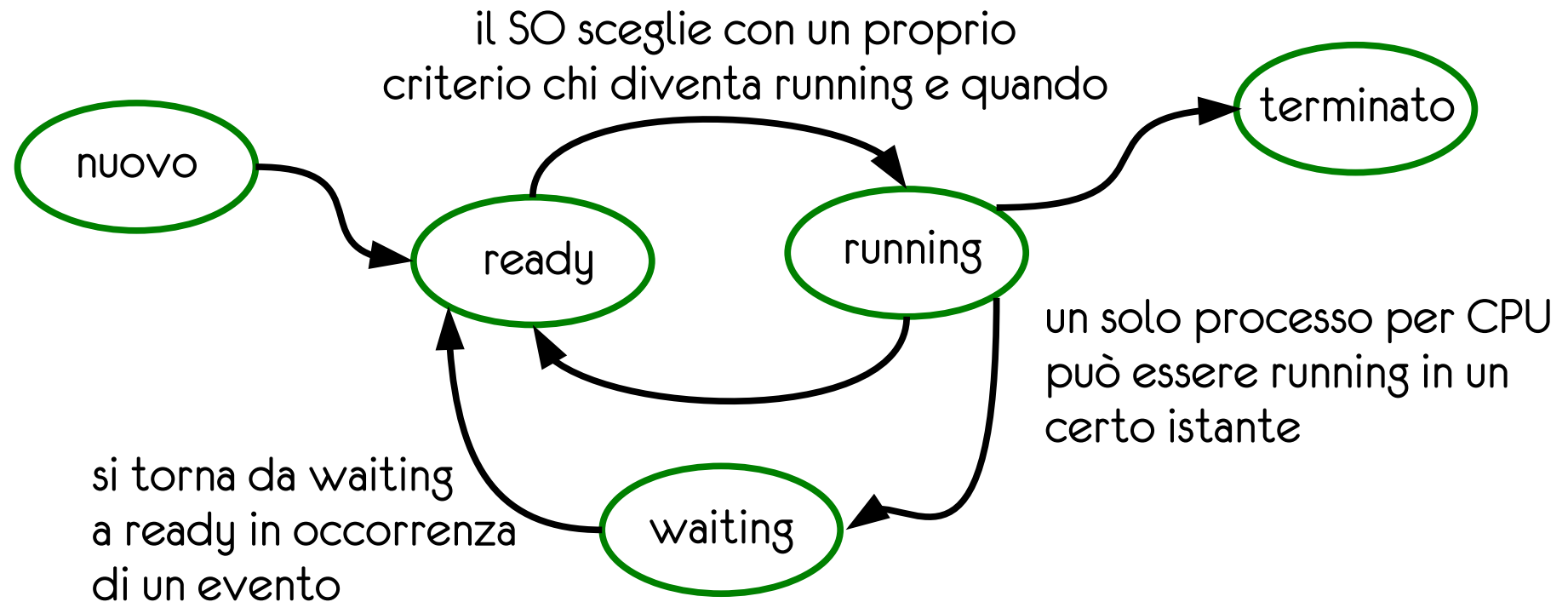
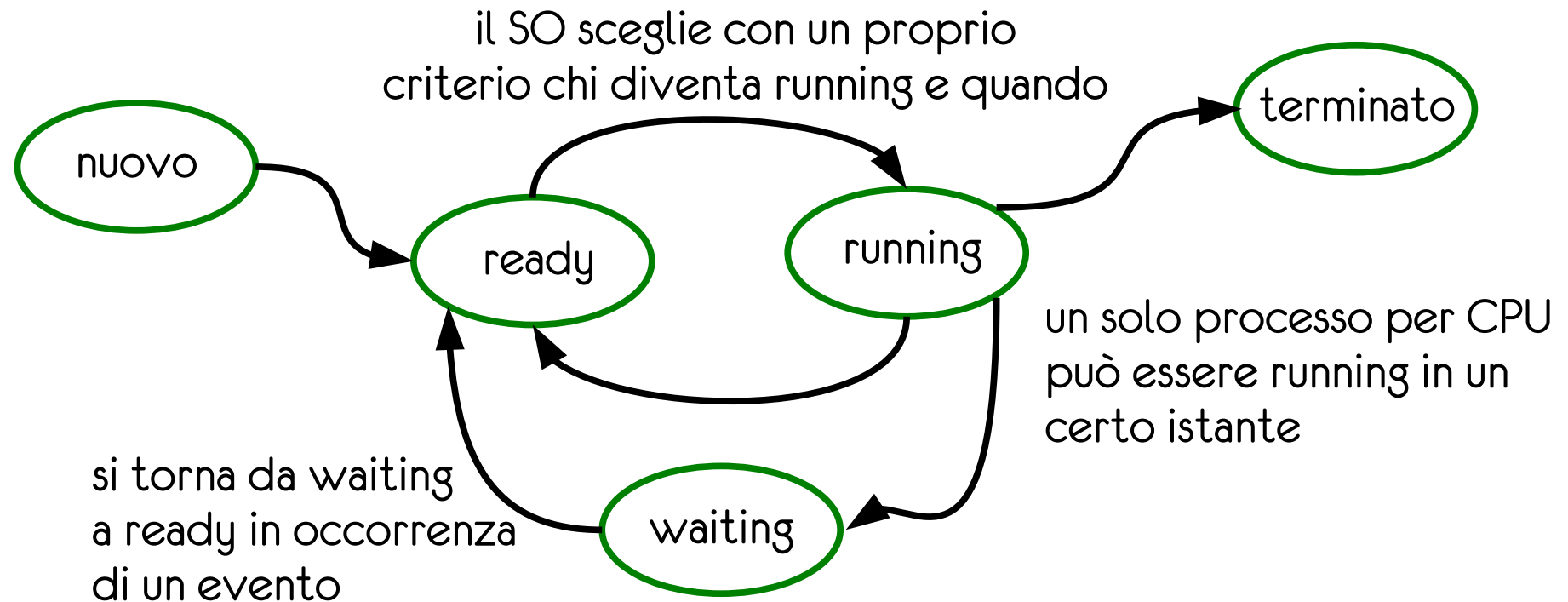


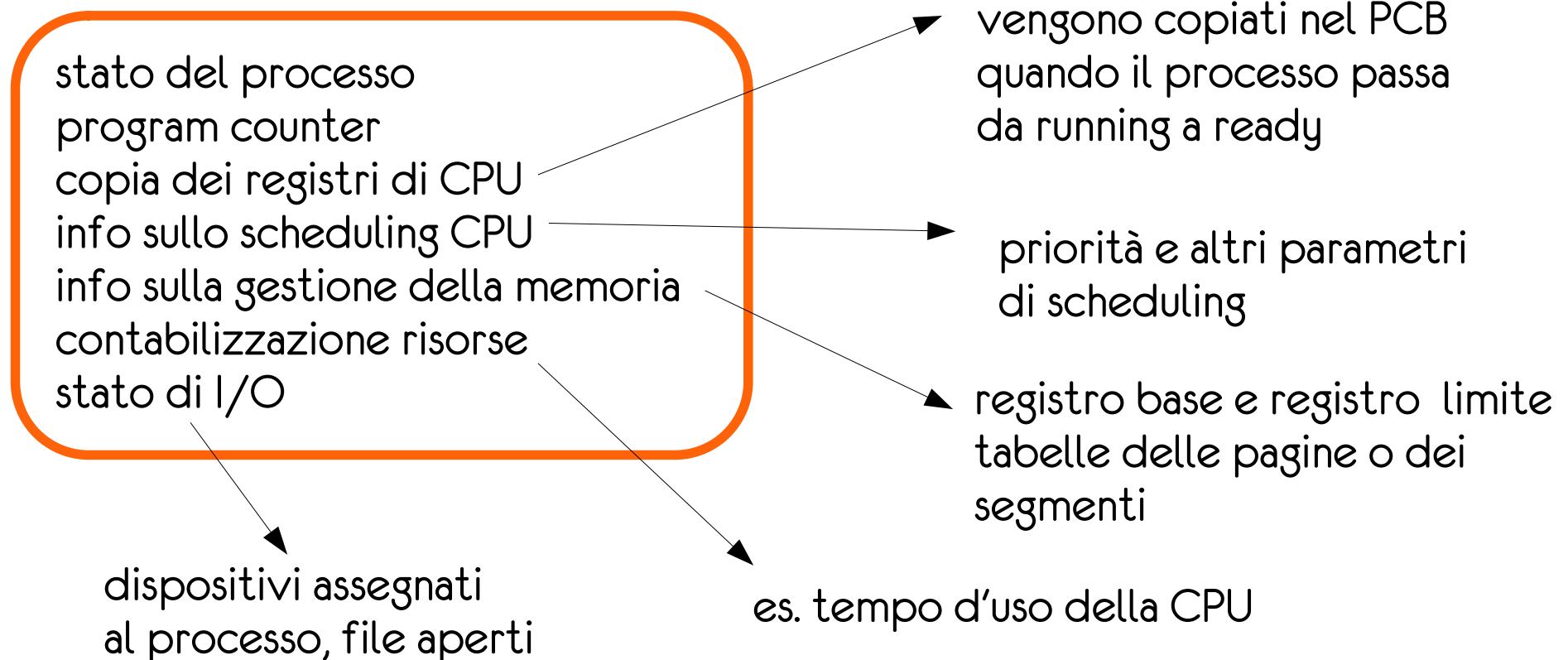
Diagramma di transizione

- Diagramma di transizione degli stati di un processo



Processi e PCB

- In un SO un processo è rappresentato da un PCB, **Process Control Block**, contenente queste informazioni:



Es. commutazione della CPU

- vediamo cosa succede quando il SO toglie la CPU a un processo running (P1) per passarla a un processo ready (P2)



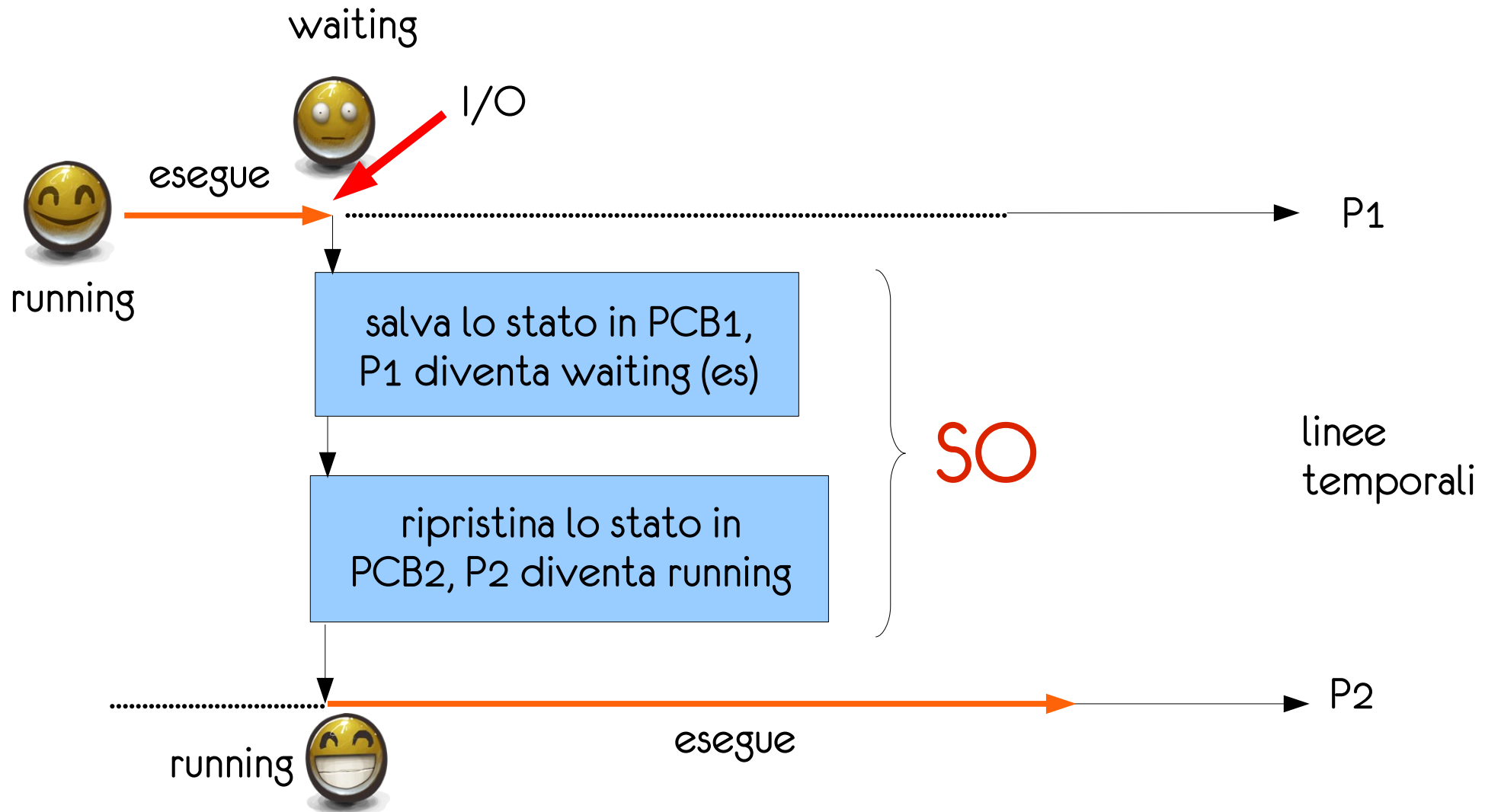
Es. commutazione della CPU

- vediamo cosa succede quando il SO toglie la CPU a un processo running (P1) per passarla a un processo ready (P2)



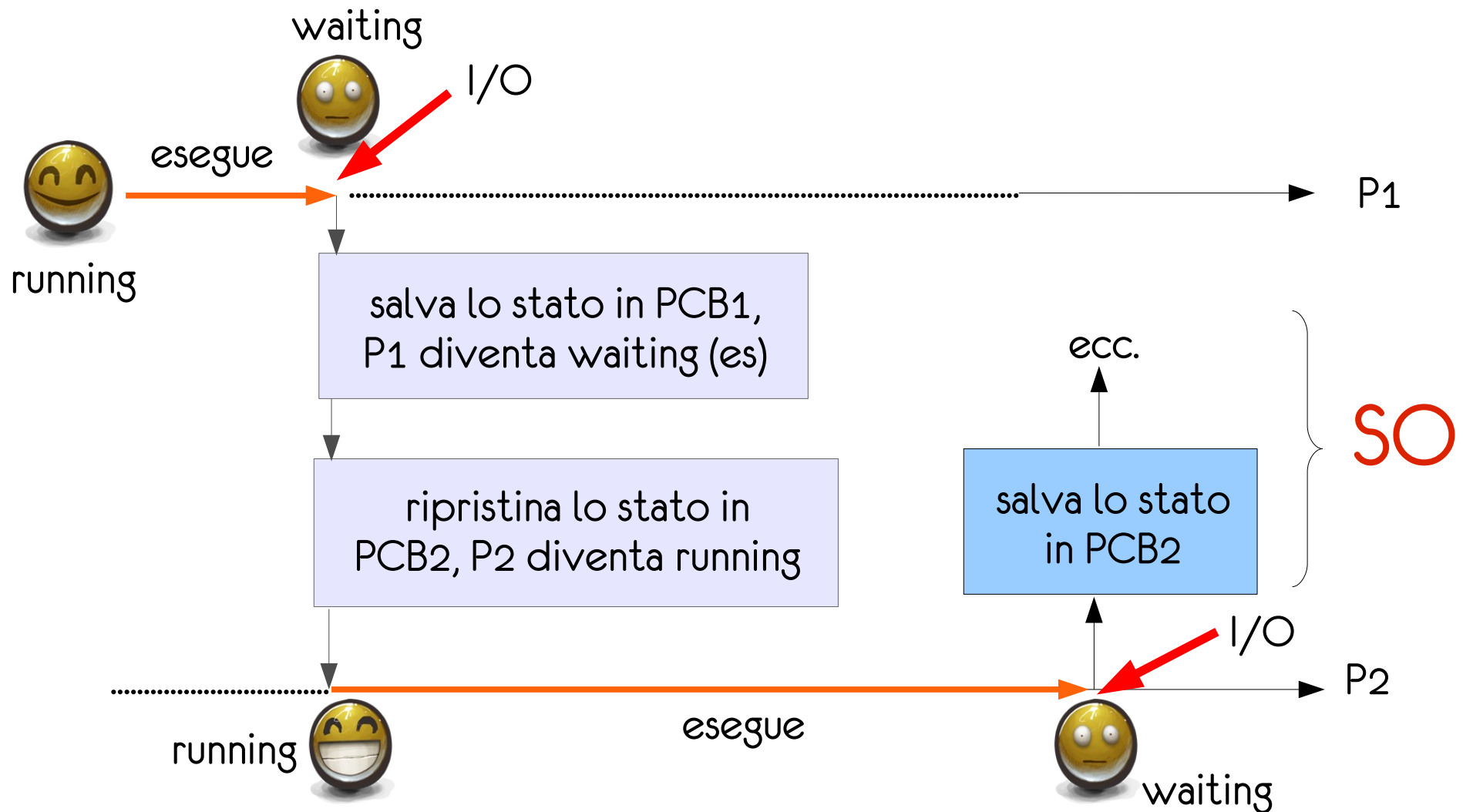
Es. commutazione della CPU

- vediamo cosa succede quando il SO toglie la CPU a un processo running (P1) per passarla a un processo ready (P2)



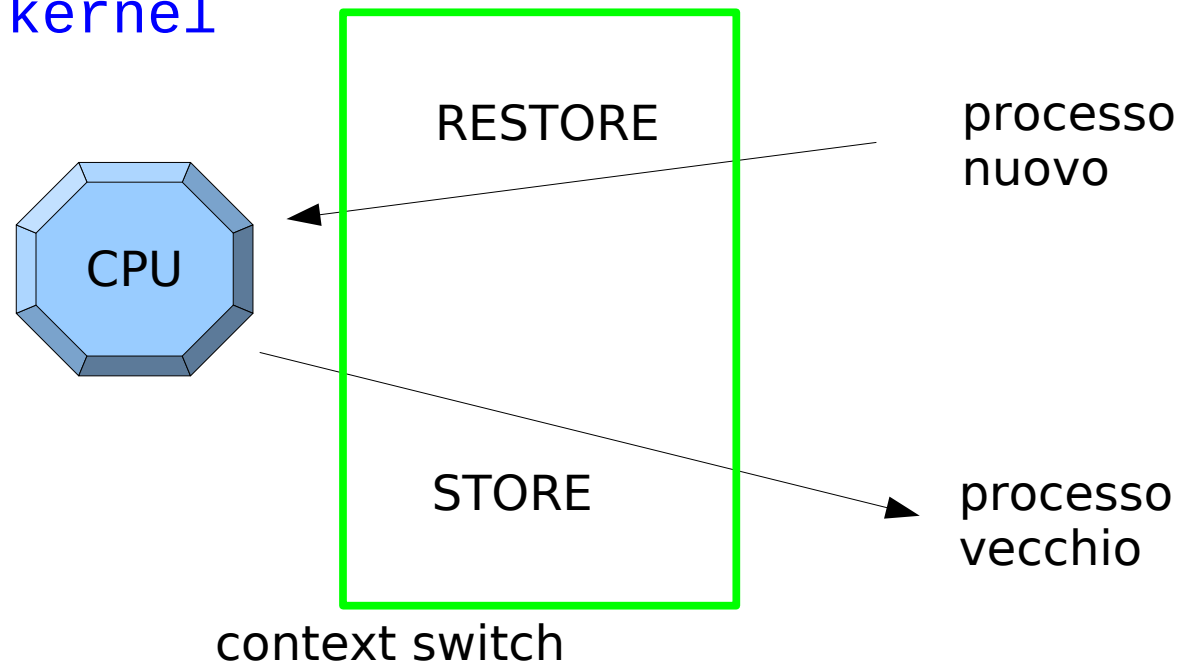
Es. commutazione della CPU

- vediamo cosa succede quando il SO toglie la CPU a un processo running (P1) per passarla a un processo ready (P2)



Context switch

- Il passaggio di un processo da **running** a **waiting** e il concomitante passaggio di un altro processo da **ready** a **running** richiede un **context switch** (cambio del contesto di esecuzione)
- contesto = contenuto dei registri della CPU e del program counter
- il context switch avviene esclusivamente in modalità kernel



Context switch

- ogni volta che un'interruzione causa la riassegnazione della CPU viene effettuato un **context switch** (cambiamento di contesto):
 - lo stato corrente della CPU viene salvato nel PCB del processo sospeso (**se viene sospeso e non terminato**)
 - lo stato relativo al processo scelto dallo scheduler a breve termine viene caricato
 - lo stato comprende i valori dei registri
- il **tempo per un context switch** dipende dall'architettura:
alcune architetture prevedono diversi set di registri, il context switch indica semplicemente quale set va usato

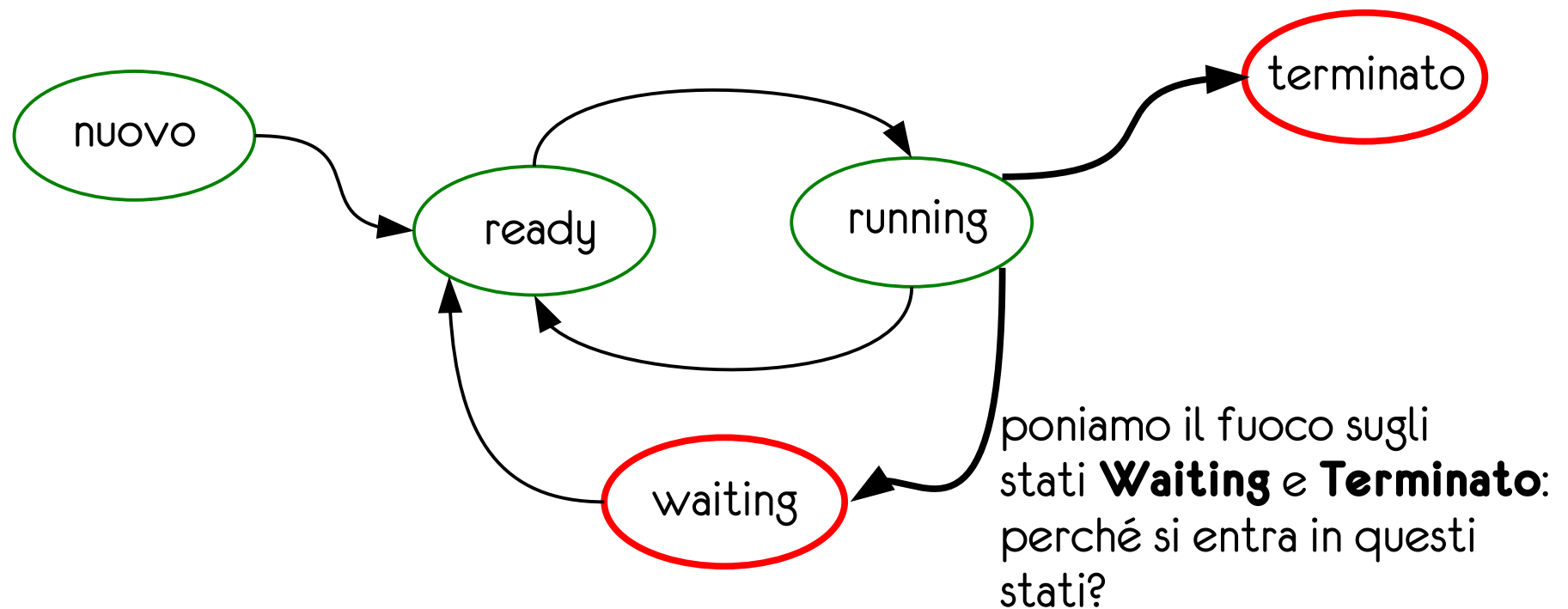
Quanto tempo dura un C.S.?

```
baroglio@esmeralda:~/BACKUP/LAB_S0/Slide$ vmstat
procs -----memory----- --swap--  -----io----- -system--  -----cpu-----
r  b   swpd   free   buff  cache   si   so    bi    bo    in    cs   us  sy  id  wa
1  0       0 994084 157708 572248    0    0   299   205   295   815  12   2  78   7
baroglio@esmeralda:~/BACKUP/LAB_S0/Slide$
```

- $815 \text{ cs} / \text{sec} \approx 1 \text{ c.s. ogni } 0.001 \approx 1 \text{ ogni millisecondo } (10^{-3} \text{ sec})$
- un c.s. dura alcuni nanosecondi (10^{-9} sec)
- molti più c.s. (815) che interrupt (295) \Rightarrow c.s. non sono causati esclusivamente da operazioni di I/O

Diagramma di transizione

- Diagramma di transizione degli stati di un processo



Passaggio a waiting/terminato

- **waiting:**
 - esecuzione di un comando di I/O
 - sospensione volontaria per un lasso di tempo
 - sospensione volontaria in attesa di un evento (es. morte di un altro processo)
 - sincronizzazione (es. attesa di un messaggio, attesa legata ad altri strumenti di sincronizzazione come i semafori)

Passaggio a waiting/terminato

- **waiting:**

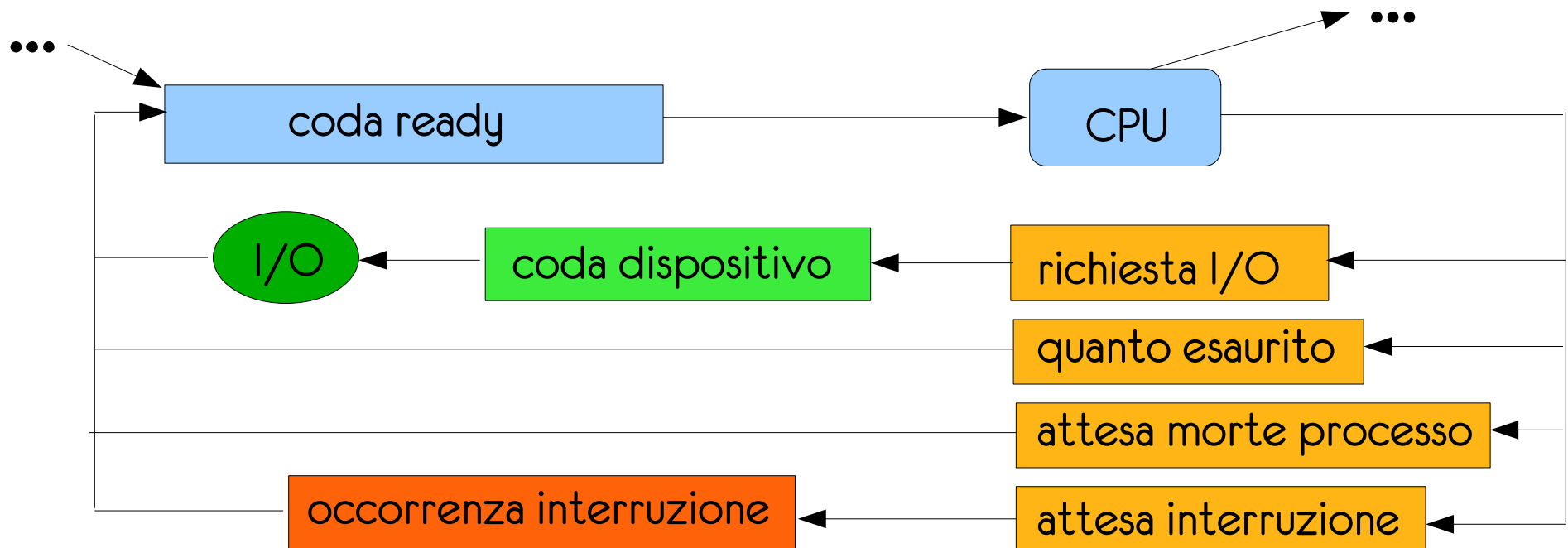
- esecuzione di un comando di I/O
- sospensione volontaria per un lasso di tempo
- sospensione volontaria in attesa di un evento (es. morte di un altro processo)
- sincronizzazione (es. attesa di un messaggio, attesa legata ad altri strumenti di sincronizzazione come i semafori)

- **terminato**

- exit
- abort
- kill da parte di altri processi
- interruzione da parte dell'utente

Code di processi

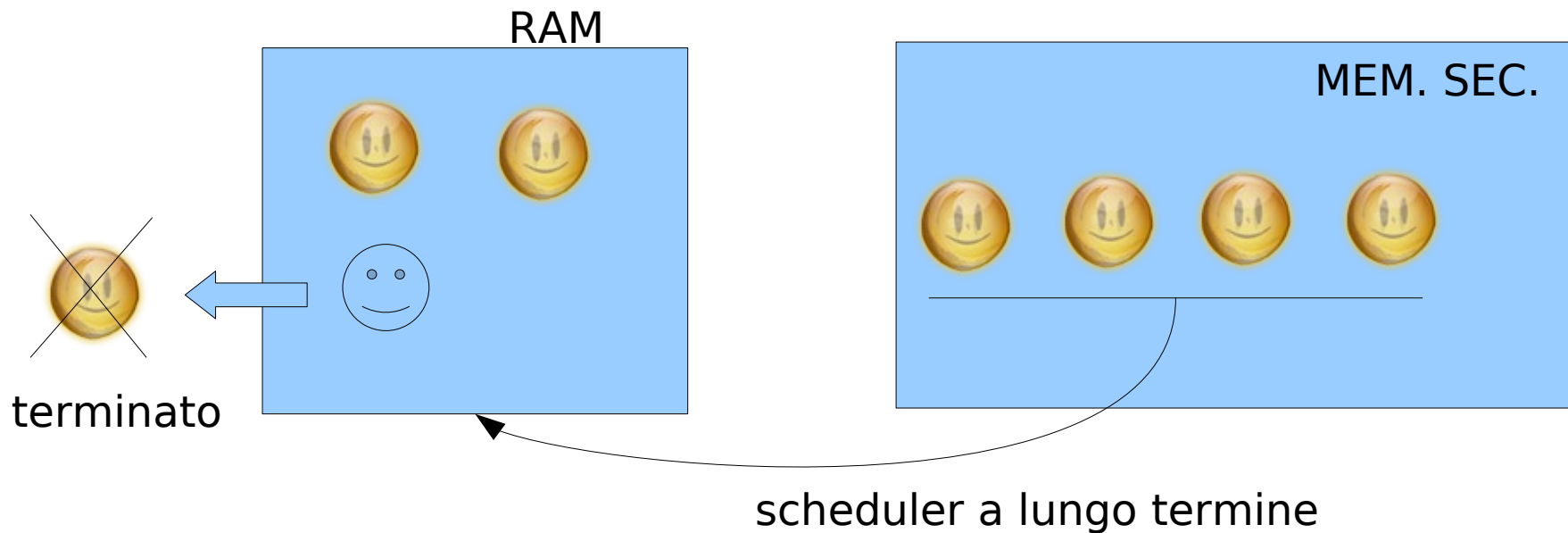
- PCB mantenuti in strutture dati (code):
 - **coda ready**: contiene tutti i PCB dei processi caricati in memoria e pronti all'esecuzione
 - **coda di dispositivo** (ne esistono diverse): contiene tutti i PCB dei processi in attesa che si completi un'operazione da loro richiesta su quel dispositivo



Scheduling

- **Scheduling a lungo termine:**
presente in **sistemi batch**, in cui la coda dei processi da eseguire era conservata in memoria secondaria. Si attiva quando un processo caricato in memoria principale **termina**, scegliendone uno in memoria secondaria e caricandolo in memoria principale
- **Scheduling a medio termine:**
quando il **grado di multiprogrammazione** è troppo alto, non tutti i processi possono essere contenuti in RAM, a turno alcuni vengono spostati in memoria secondaria
- **Scheduling a breve termine:**
politica di **avvicendamento alla CPU** dei processi caricati in RAM

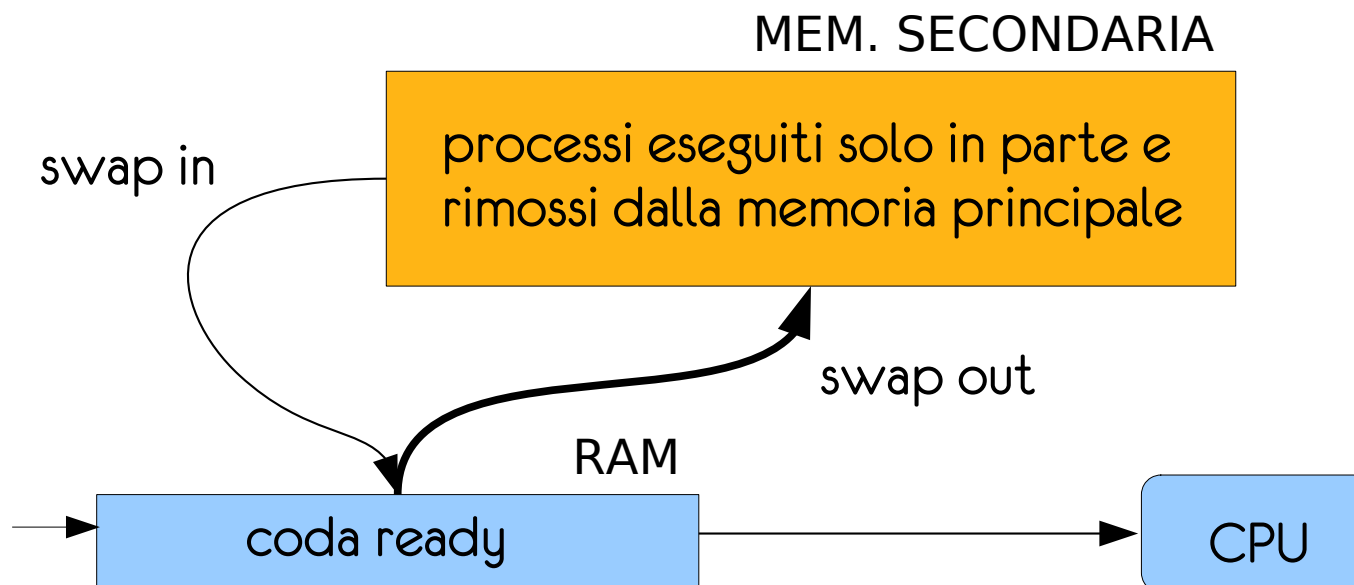
Scheduling a lungo termine



- si limita a sostituire processi terminati, usando uno **scheduler a lungo termine**, si attiva al termine di un processo e carica dalla memoria secondaria un altro processo
- **criterio:**
mantenere un buon equilibrio fra processi CPU-bound e (che in prevalenza fanno computazione) processi IO-bound (che in prevalenza fanno I/O)

PCB in memoria secondaria

- Un sistema può avere in esecuzione più processi di quanti ne possano coesistere nella memoria principale ...
- **soluzione**: una parte dei processi va trasferita in memoria secondaria fino a quando non sarà possibile eseguirli (**swapping** o avvicendamento dei processi in memoria o **scheduling a medio termine**)



NB. lo scheduling a lungo termine non prevede l'analogo dello swap out

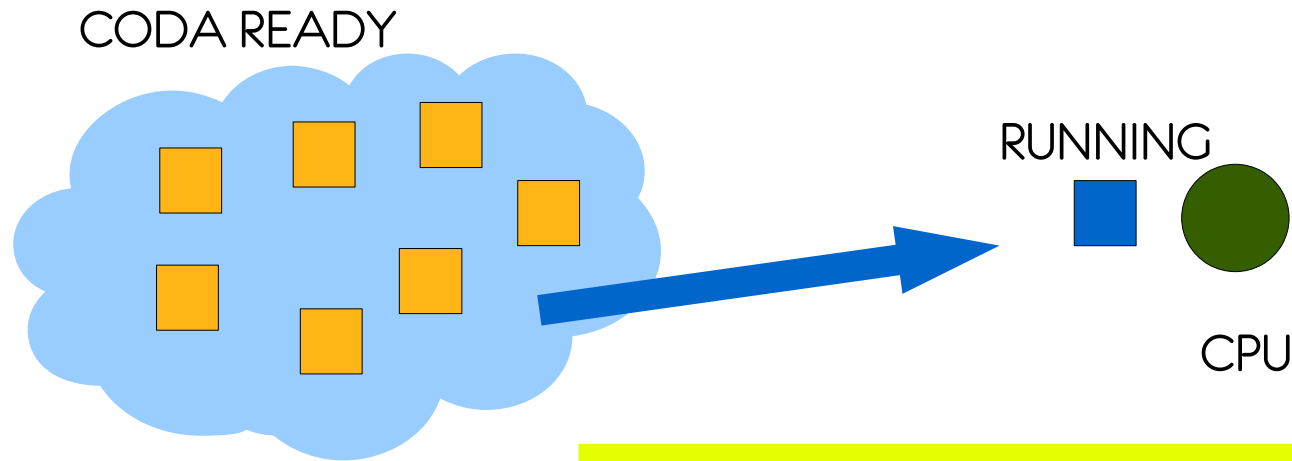
scheduling della cpu

capitolo 5 del libro (VII ed.), fino a 5.4

Scheduler della CPU

- ... o **scheduler a breve termine**: seleziona dalla coda ready il processo a cui assegnare la CPU
- NB: la coda ready contiene sempre dei PCB ma può essere implementata in modi diversi, a seconda dell'algoritmo di scheduling usato; non necessariamente è una coda di tipo FIFO
- casi in cui occorre scegliere:
 - 1) il processo running diventa waiting
 - 2) il processo running viene riportato a ready (interrupt)
 - 3) un processo waiting diventa ready
 - 4) il processo running termina

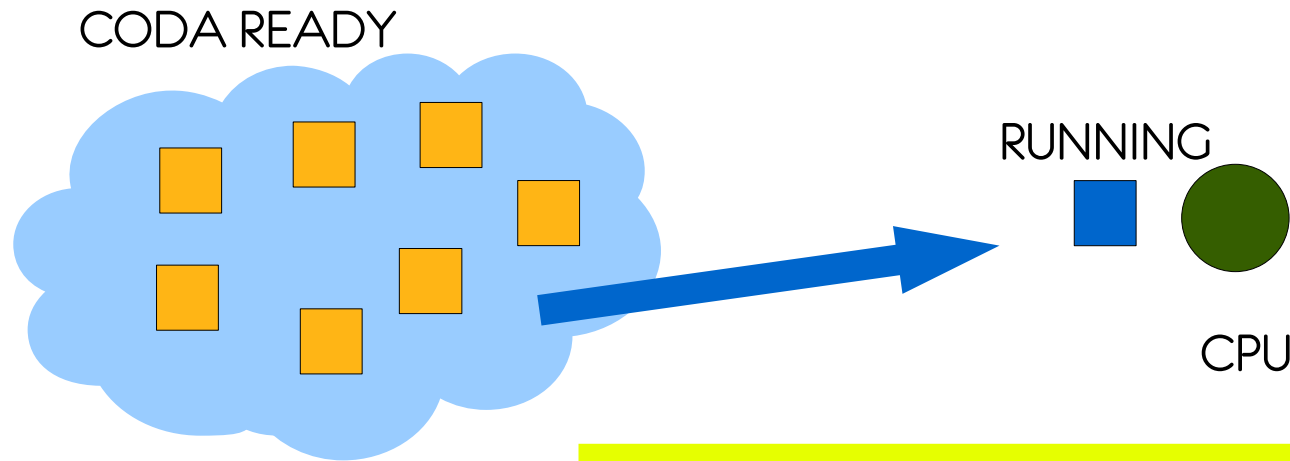
Scheduler della CPU



Lo **scheduler** sceglie un processo ready

Il **DISPATCHER** effettua il cambio di contesto, effettua il posizionamento alla giusta istruzione, passa nella modalità di esecuzione giusta

Scheduler della CPU



Lo **scheduler** sceglie un processo ready

Il **DISPATCHER** effettua il cambio di contesto, effettua il posizionamento alla giusta istruzione, passa nella modalità di esecuzione giusta

condizione che fa scattare lo scheduler
 \oplus criterio di selezione \Rightarrow politica

Algoritmi di scheduling

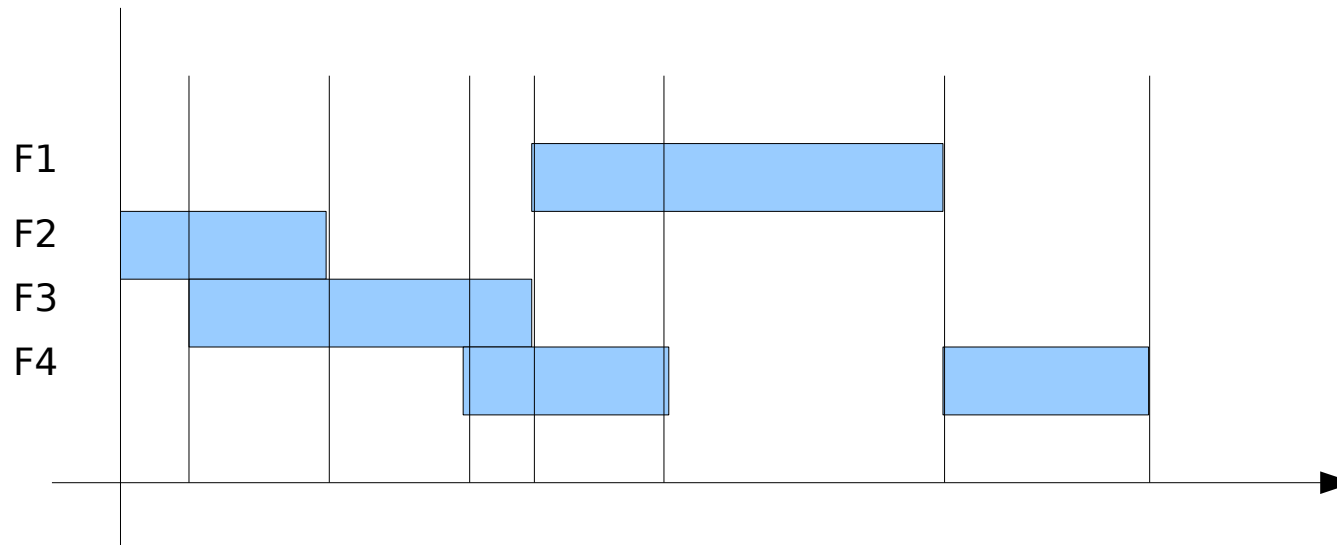
- first-come-first-served (FCFS)
- shortest jobs first (SJF)
- a priorità
- round-robin (RR)
- a code multilivello (multilevel queue scheduling)
- a code multilivello con feedback (multilevel feedback queue scheduling)

Criteri di scheduling

- mantenere la CPU il più attiva possibile
- **throughput** (produttività): numero di processi completati nell'unità di tempo
- **turnaround time**: tempo di completamento di un processo, è la somma non solo del tempo di esecuzione e dei vari I/O ma anche di tutti i tempi di attesa legati allo scheduling (attesa di ingresso in main memory, attesa nella coda ready)
- **tempo di attesa**: visto che l'algo. di scheduling della CPU non influisce sui tempi legati all'esecuzione del processo si può considerare anche il solo tempo di attesa trascorso in ready queue
- **tempo di risposta**: nei sistemi interattivi è importante ridurre il tempo che intercorre fra la sottomissione di una richiesta (da parte dell'utente) e la risposta

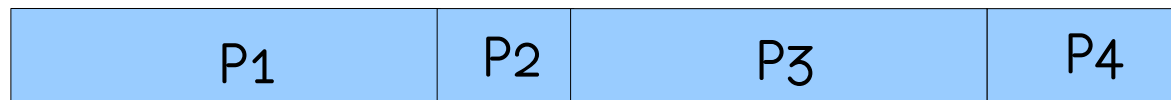
Diagrammi di Gantt

- Ideati nel 1917 come supporto per la gestione di progetti
- Asse orizzontale: tempo
- Asse verticale: progetti oppure fasi di un progetto
- Delle barre orizzontali indicano quando (sia in senso assoluto che in relazioni agli altri progetti/fasi) e per quanto tempo durano le diverse fasi



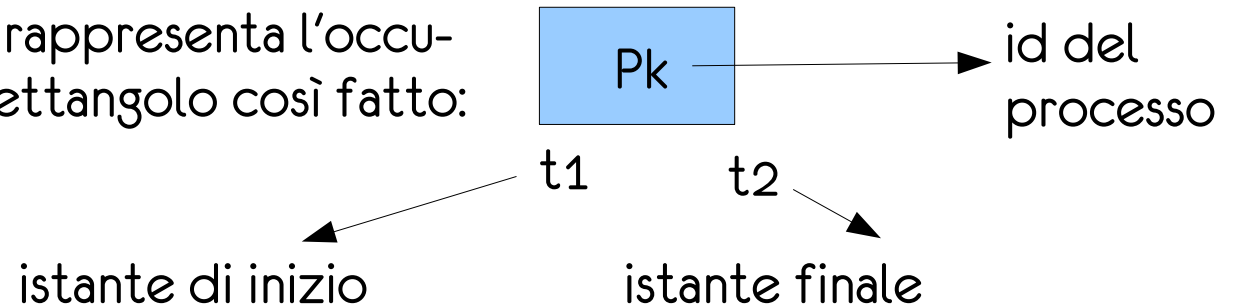
Diagrammi di Gantt

- Rappresenteremo il tempo di occupazione della CPU da parte dei diversi processi
- Una CPU può essere occupata da un solo processo per volta, quindi il Gantt non può contenere sovrapposizioni di fasi
- Si può comprimere a una sola barra contenente blocchetti, ognuno dei quali corrisponde all'esecuzione di un processo o di una sua parte



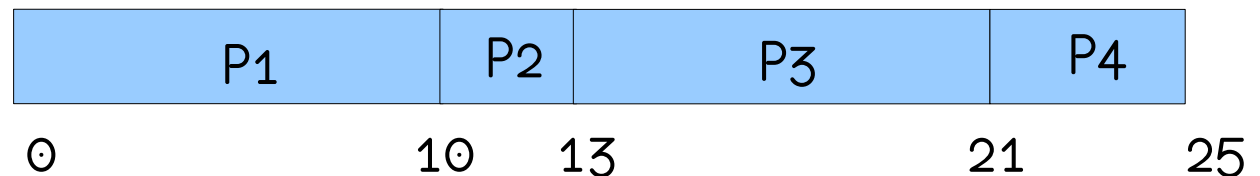
Premessa: diagrammi di Gantt

In un diagramma di Gantt si rappresenta l'occupazione della CPU con un rettangolo così fatto:



Es. dati i processi e relativi CPU burst in tabella, supponendo che la CPU venga allocata nell'ordine con cui sono dati i processi avremo il seguente diagramma

| P | D |
|----|----|
| P1 | 10 |
| P2 | 3 |
| P3 | 8 |
| P4 | 4 |



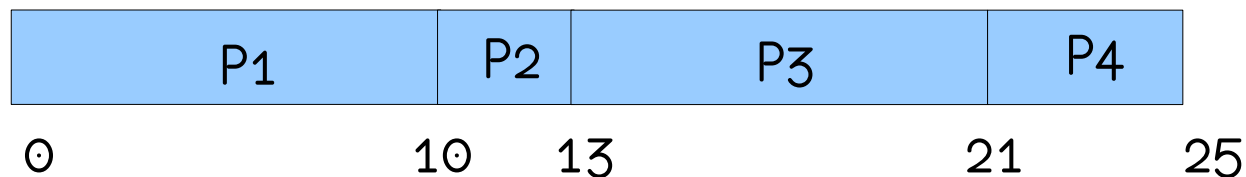
esempio di
diagramma
di Gantt

P: processo

D: durata del prossimo CPU burst, sequenza contigua di operazioni di CPU fra un I/O e l'altro

First-come-first-served

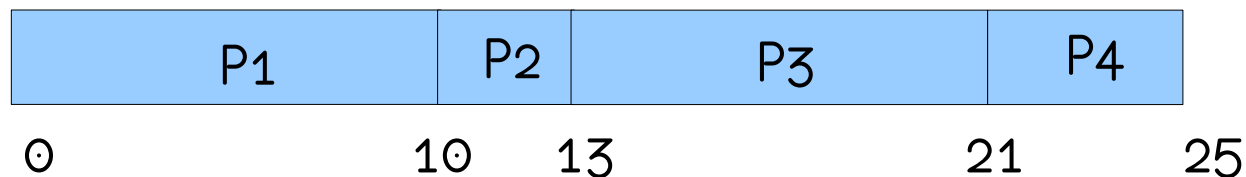
- **FCFS**: PCB organizzati in una coda FIFO, non è preemptive, il primo processo arrivato è il primo ad avere la CPU, la mantiene per un intero CPU burst
- il tempo medio di attesa è abbastanza lungo, es:



- $t.m.a. = ???$

First-come-first-served

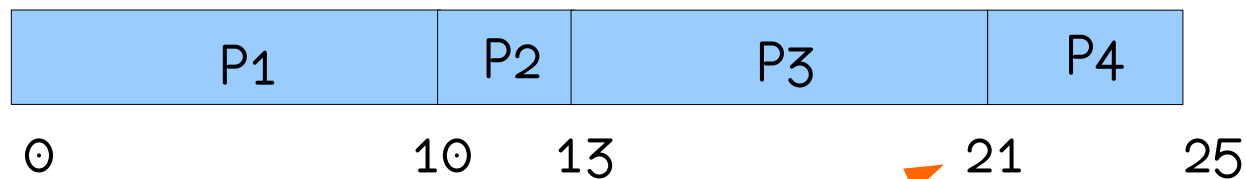
- **FCFS**: PCB organizzati in una coda FIFO, non è preemptive, il primo processo arrivato è il primo ad avere la CPU, la mantiene per un intero CPU burst
- il tempo medio di attesa è abbastanza lungo, es:



- $tma = (ta_{P1} + ta_{P2} + \dots + ta_{Pn}) / num_proc$

First-come-first-served

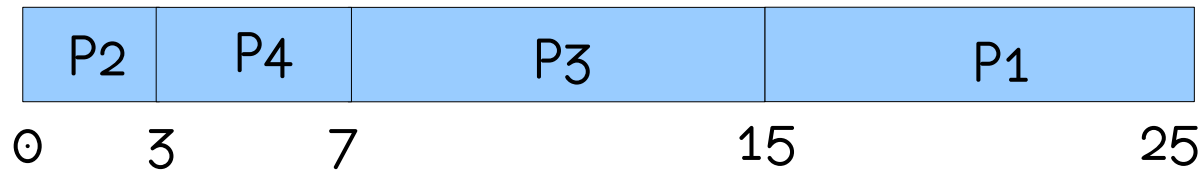
- **FCFS**: PCB organizzati in una coda FIFO, non è preemptive, il primo processo arrivato è il primo ad avere la CPU, la mantiene per un intero CPU burst
- il tempo medio di attesa è abbastanza lungo, es:



- $$\begin{aligned} \text{tma} &= (0 + 10 + 13 + 21) / 4 = \\ &= 44/4 = \\ &= 11 \text{ millisec} \end{aligned}$$

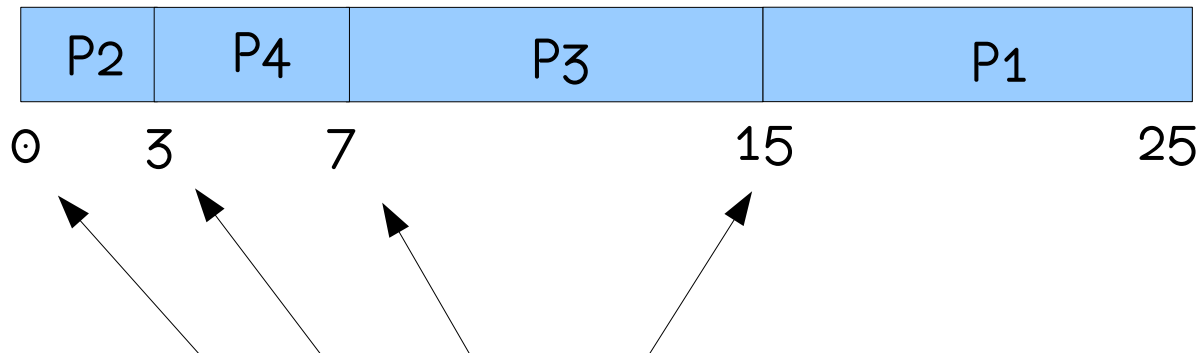
First-come-first-served

- Se i PCB fossero giunti in un altro ordine il tma sarebbe cambiato? Es.



First-come-first-served

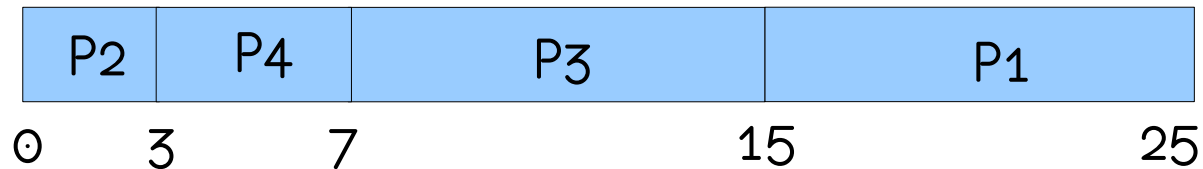
- **si:**



- $t_{ma} = (0 + 3 + 7 + 15) / 4 = 25/4 = 6,25 \text{ millisec}$

First-come-first-served

- **sì:**



- $t_{ma} = (0 + 3 + 7 + 15) / 4 = 25/4 = 6,25 \text{ millisec}$
- **NB:** FCFS non adeguato a sistemi in time-sharing in cui tutti gli utenti devono disporre della CPU a intervalli regolari

First-come-first-served

- **effetto convoglio**: supponiamo di avere n processi I/O-bound e un processo CPU-bound in coda ready:



- il processo CPU-bound occupa la CPU per un tempo lungo
- quando fa, per es., I/O la cede
- i processi successivi hanno CPU-burst molto brevi, si alternano in fretta e tornano ad accodarsi

Prelazione di una risorsa

- Meccanismo generale per il quale il SO può togliere una risorsa riservata per un processo anche se:
 - il processo la sta utilizzando
 - il processo utilizzerà la risorsa in futuro
- Nello scheduling a breve termine la risorsa contesa è la CPU

Prelazione

- 1) il processo running diventa waiting
- 2) il processo running viene riportato a ready (interrupt)
- 3) un processo waiting diventa ready
- 4) il processo running termina

| | 1 | 2 | 3 | 4 |
|----------------|---|----|----|---|
| preemptive | | + | + | |
| non-preemptive | + | no | no | + |

Lo scheduling è **PREEMPTIVE** se interviene in almeno uno dei casi 2 e 3, può occorrere anche in 1 o 4

Lo scheduling è **NON-PREEMPTIVE** se interviene solo nei casi 1 o 4



La CPU non viene mai tolta al processo running

Prelazione, supporto HW . . .

- La prelazione non è sempre possibile, **occorre un'architettura che la consenta.**
- In particolare occorrono dei **timer**, se non ci sono l'unico tipo di scheduling che si può avere è non-preemptive (anche detto cooperativo)
- **Es. Apple ha introdotto la prelazione nei suoi sistemi operativi con Mac OS X; Windows l'ha adottata con Windows 95**

Prelazione, supporto HW ... e rischi

- La prelazione richiede l'introduzione di meccanismi di **sincronizzazione** per evitare inconsistenze.
- Se due processi **condividono dati** e uno dei due li sta aggiornando quando gli viene tolta la CPU (a favore dell'altro), quest'ultimo troverà dati inconsistenti.
- Affronteremo il problema più avanti

Shortest job first

- SJF è **ottimale** nel minimizzare il tempo medio di attesa, **seleziona sempre il processo avente CPU burst successivo di durata minima** (shortest next CPU burst)
- **Problema**: la durata dei CPU burst non è nota a priori!
- **Soluzione**: prevedere la durata sulla base dei burst precedenti, combinati con una media esponenziale:

$$\tau_{n+1} = \alpha t_n + (1 - \alpha) \tau_n$$

- espansa la formula diventa:

$$\tau_{n+1} = \alpha t_n + \alpha(1 - \alpha)t_{n-1} + \dots + \alpha(1 - \alpha)^j t_{n-j} + \dots + (1 - \alpha)^{n+1} \tau_0$$

- α è un numero compreso fra 0 e 1, quindi gli addendi più vecchi hanno un influsso via via inferiore.
- Maggiore è α maggiore è il peso dato alla storia recente

Shortest job first

- **SJF può essere preemptive o meno**
- **comportamento preemptive** (anche detto “shortest remaining time left”): quando un nuovo processo diventa ready, lo scheduler controlla se il suo burst è inferiore a quanto rimane del burst del processo running:
 - SÌ: il nuovo processo diventa running, si ha commutazione di contesto
 - NO: il nuovo processo viene messo in coda ready
- **comportamento non-preemptive**: il processo viene messo in coda ready, eventualmente in prima posizione

Priorità e Starvation

- **Problema:** tutti gli algoritmi a priorità sono soggetti a **starvation**:

un processo potrebbe non ottenere mai la CPU perché continuano a passargli davanti nuovi processi a priorità maggiore

- **Soluzione:** **aging**. La priorità viene alzata con il trascorrere del tempo

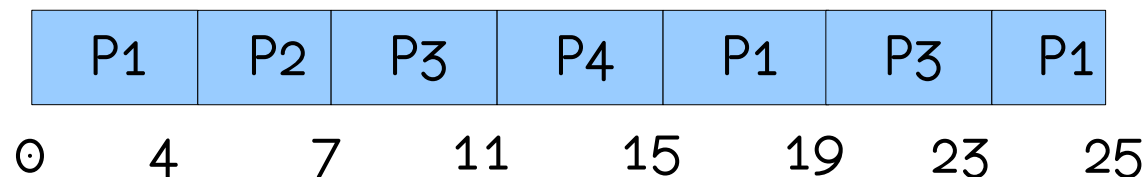
Scheduling a Priorità

- Ogni processo ha associata una priorità che decide l'ordine di assegnazione della CPU. Può essere con o senza prelazione
- La priorità può essere definita:
 - **Internamente:** sulla base di caratteristiche del processo. Es. SJF è un algoritmo di scheduling basato su priorità definita internamente, priorità = inverso della durata stimata del CPU burst
 - **esternamente:** non calcolabile, imposta in base a criteri esterni, dagli utenti per esempio

Round-robin

- Ideato espressamente per **sistemi time-sharing**, è preemptive. La coda ready è circolare, gestita FIFO. A ogni processo viene assegnata la CPU per un quanto di tempo, se non è sufficiente a concludere, il processo viene reinserito in coda ready e la CPU riassegnata al successivo
- **Esempio**, supponiamo di avere i soliti quattro processi e relativi CPU burst e che il quanto sia lungo 4:

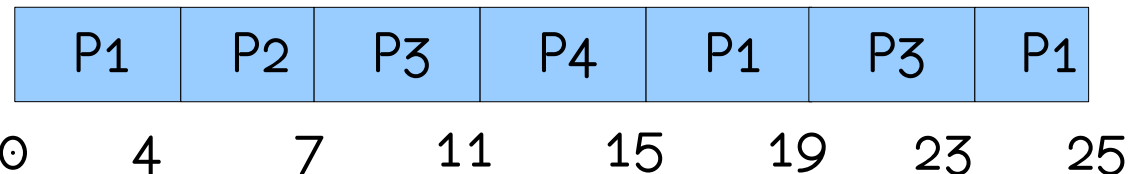
| P | D |
|----|----|
| P1 | 10 |
| P2 | 3 |
| P3 | 8 |
| P4 | 4 |



Round-robin

- Ideato espressamente per **sistemi time-sharing**, è preemptive. La coda ready è circolare, gestita FIFO. A ogni processo viene assegnata la CPU per un quanto di tempo, se non è sufficiente a concludere, il processo viene reinserito in coda ready e la CPU riassegnata al successivo
- **Esempio**, supponiamo di avere i soliti quattro processi e relativi CPU burst e che il quanto sia lungo 4:

| P | D |
|----|----|
| P1 | 10 |
| P2 | 3 |
| P3 | 8 |
| P4 | 4 |



$$t_{ma} = (0 + 4 + 7 + 11 + 11 + 8 + 4) / 4 = 45 / 4 = 11,25$$

Round-robin

- il tma è abbastanza alto ma **non si ha starvation**:
ogni processo viene servito ogni **$(n_proc - 1) * \text{quanto}$** millisecondi
- All'utente sembra di avere a disposizione una CPU solo un po' più lenta (**$1/n$, $n = \text{num. processi}$**)
- La scelta del quanto è critica:
 - Se è molto lungo, allora RR tende a diventare un FCFS
 - Se è troppo corto, allora i tempi necessari ai cambi di contesto rischiano di appesantire e rallentare molto il processamento

Code a multilivello / con feedback

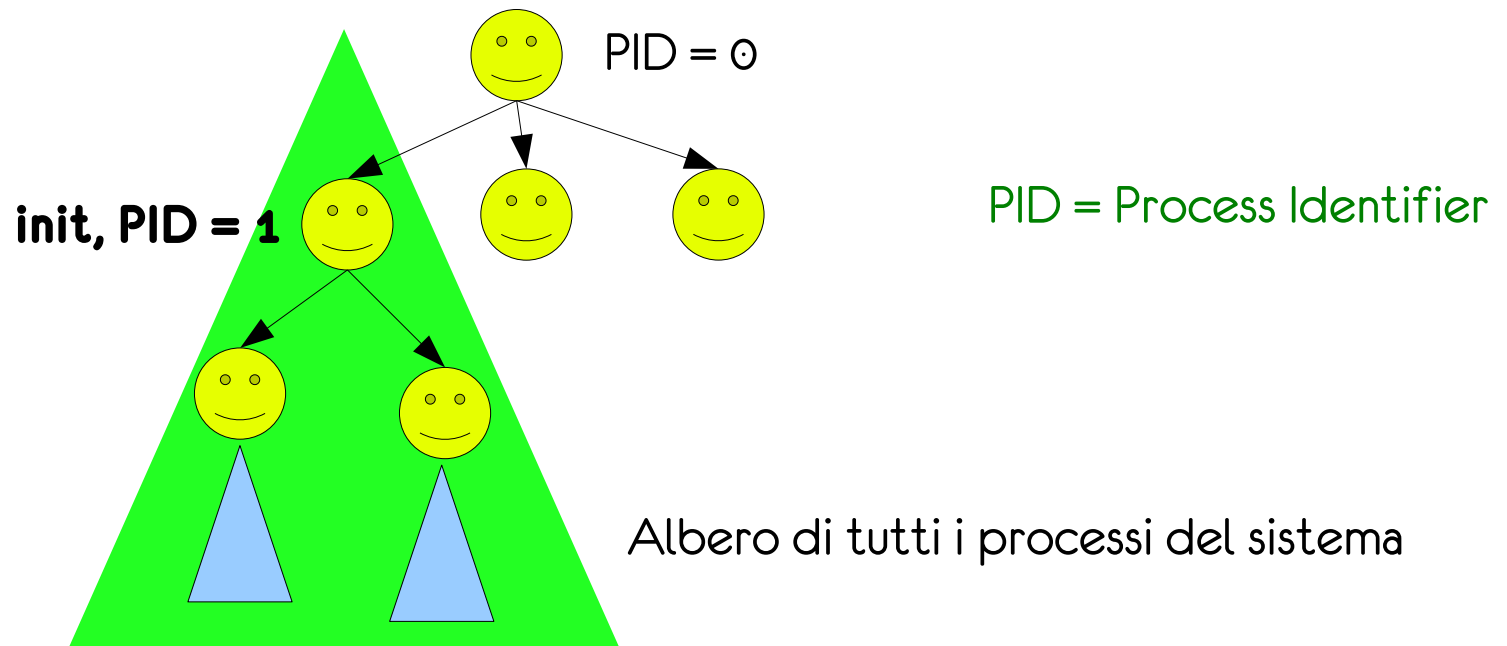
- Algoritmi utili quando è possibile distinguere i processi in categorie legate alla loro natura
- La ready queue è suddivisa in tante code quante sono le categorie
- Ogni coda può avere un algoritmo di scheduling diverso
- Esiste una priorità fra le code (eventualmente associata a meccanismi di invecchiamento per evitare starvation)
- Se ai processi è consentito cambiare coda allora si parla di multilivello con feedback

creazione e terminazione

sezione 3.3 del libro (VII ed.)

Creazione

- come nasce un processo?
- un processo viene generato dall'unica entità attiva gestita dal S0: **un altro processo**
- con l'avvio del S0 si genera un **albero di processi** che cresce e decresce dinamicamente con l'evoluzione dell'elaborazione



Creazione

- Ogni processo ha un identificatore univoco (numero intero), il **process identifier** o **PID**
- Ogni processo generatore è detto “**padre**”, ogni processo generato è detto “**figlio**”
- Un processo padre in generale **condivide delle risorse** (es. file aperti) con i propri processi figli, certe volte un figlio può usare solo un sottoinsieme delle risorse del padre
- Il processo padre può avere necessità di **passare dei dati** al processo figlio, che viene generato per eseguire una particolare elaborazione. Per esempio in Unix il figlio riceve **una copia di tutte le variabili** del padre

Creazione

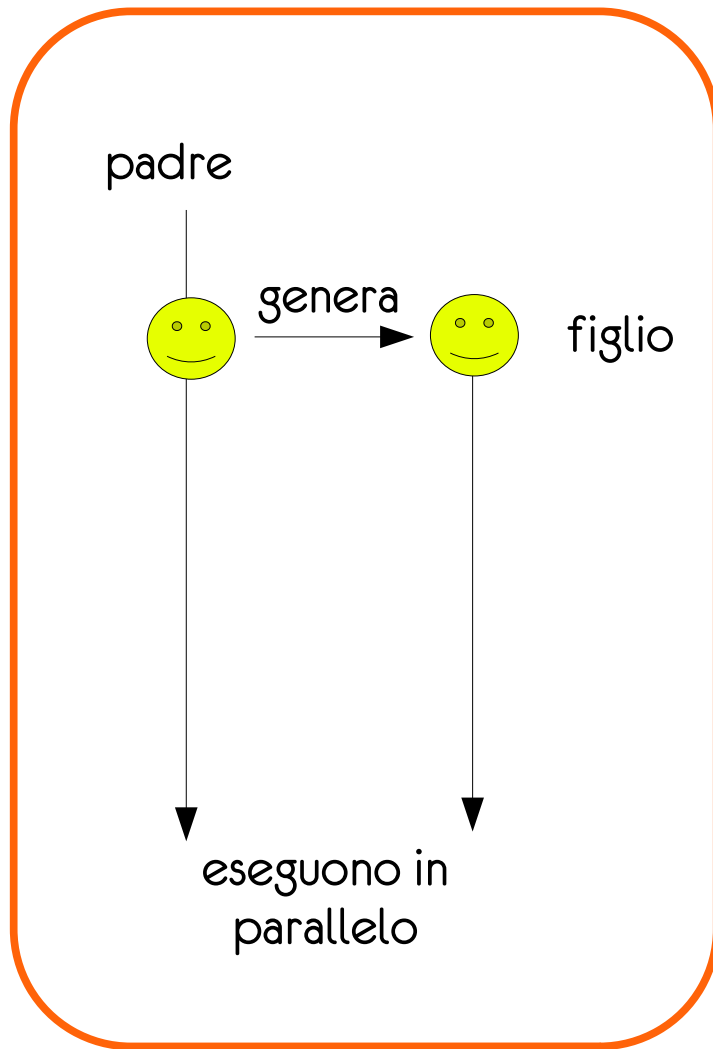
- **Esecuzione:**

- A) il padre continua la propria esecuzione in modo concorrente a quella dei figli
- B) il padre si sospende in attesa della terminazione di tutti i figli

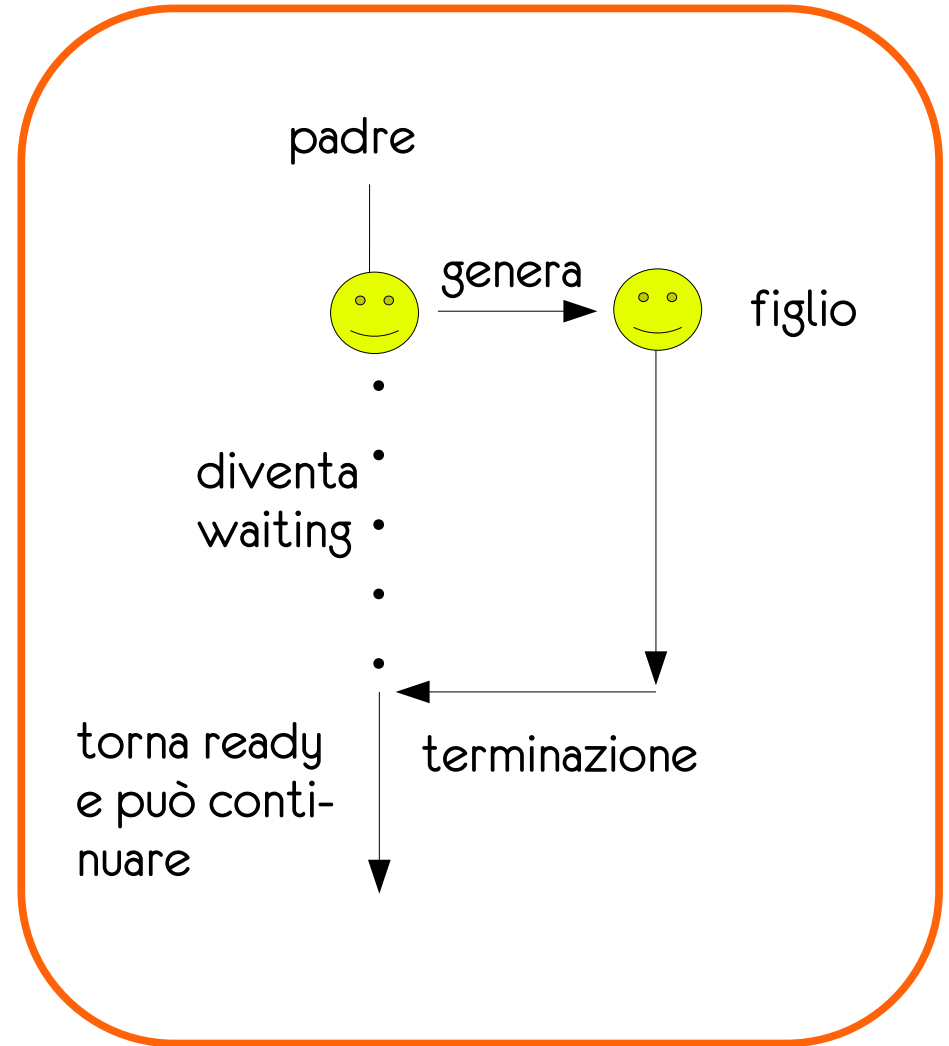
- **Programma eseguito:**

- A) il figlio è una copia del processo padre
 - B) il figlio esegue un programma diverso
- in Unix un processo figlio inizialmente è una copia del processo padre e i due eseguono in parallelo. Un processo può cambiare il programma che esegue tramite una system call (**exec**)

Creazione: fork



CASO A



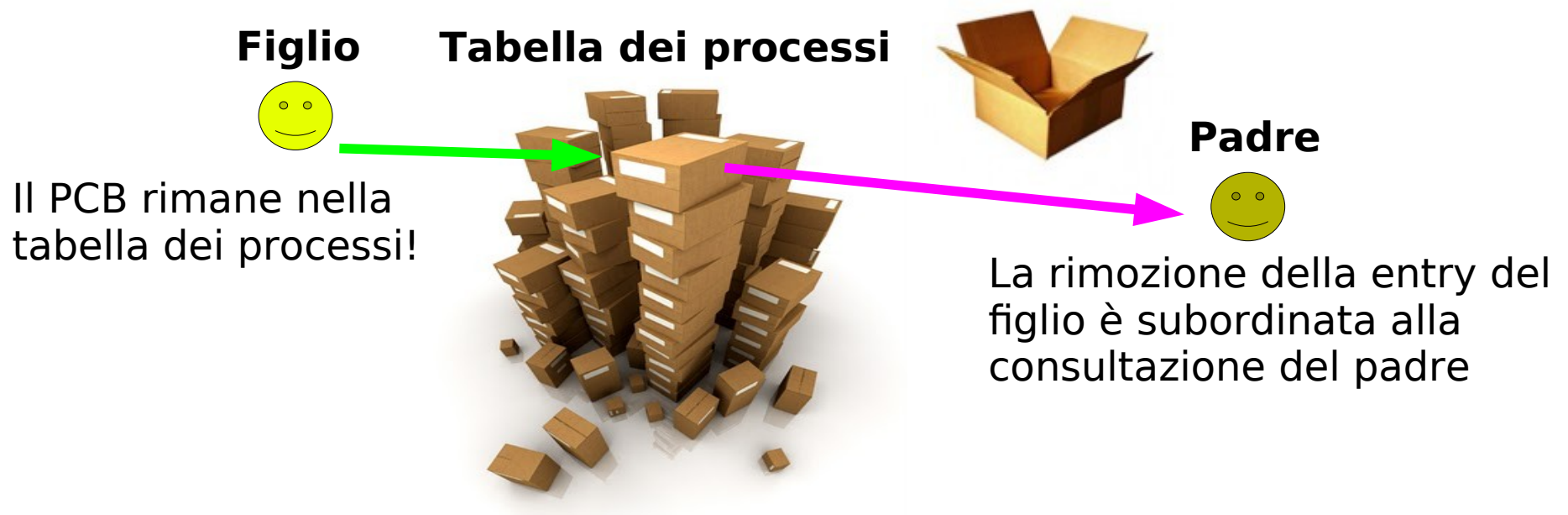
CASO B

Terminazione

- Un processo giunto alla sua ultima istruzione notifica al SO che è pronto per terminare tramite la system call `exit`
- Il SO libera le risorse allocate per il processo. Se il padre del processo terminato attendeva la sua terminazione, **questa gli viene notificata**, insieme ad alcuni dati inerenti la terminazione del figlio

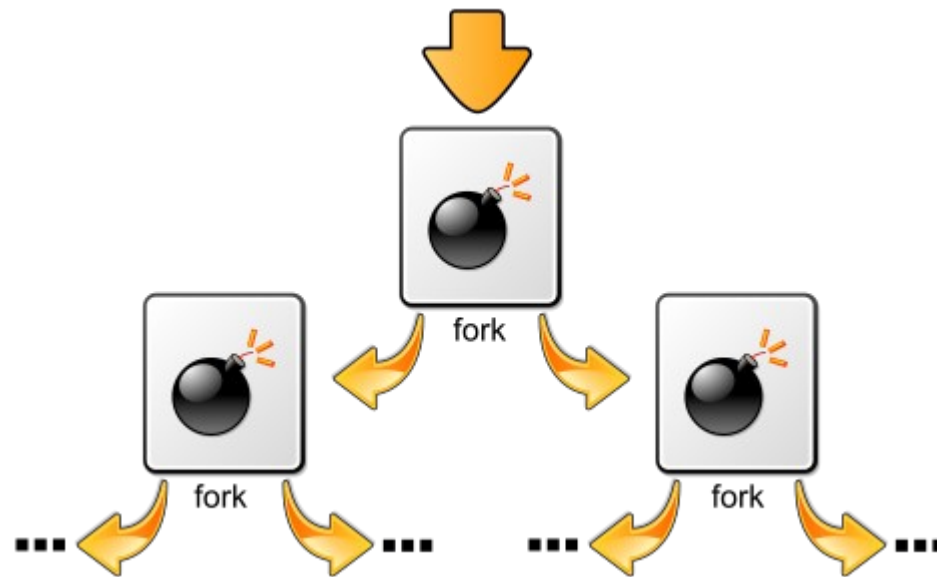
Problemi

- ... `system call exit`
- Il SO libera le risorse allocate per il processo. Se il padre del processo terminato attendeva la sua terminazione, **questa gli viene notificata**, insieme ad alcuni dati inerenti la terminazione del figlio
- **Attenzione:** in certi SO i dati sulla terminazione del figlio sono conservati fino a quando non vengono ispezionati dal padre!



Fork bomb / wabbit

- Dicesi **wabbit** (o **fork bomb**) un programma che crea un numero smisurato di figli, che a loro volta creano un numero smisurato di figlio che a loro volta ... ecc. ecc.
- Effetto: il numero di processi coesistenti, gestibili da un SO è limitato. Uno wabbit lo riempie rapidamente bloccando, di fatto, il sistema.



Terminazione

- Un processo può causare la terminazione di un altro in modo esplicito, tramite system call, a patto di conoscerne il PID
- Alcuni possibili motivi:
 - il processo sta usando troppe risorse
 - la sua elaborazione non occorre più
 - il padre è terminato e il SO forza i suoi figli a fare altrettanto (es. SO VMS)



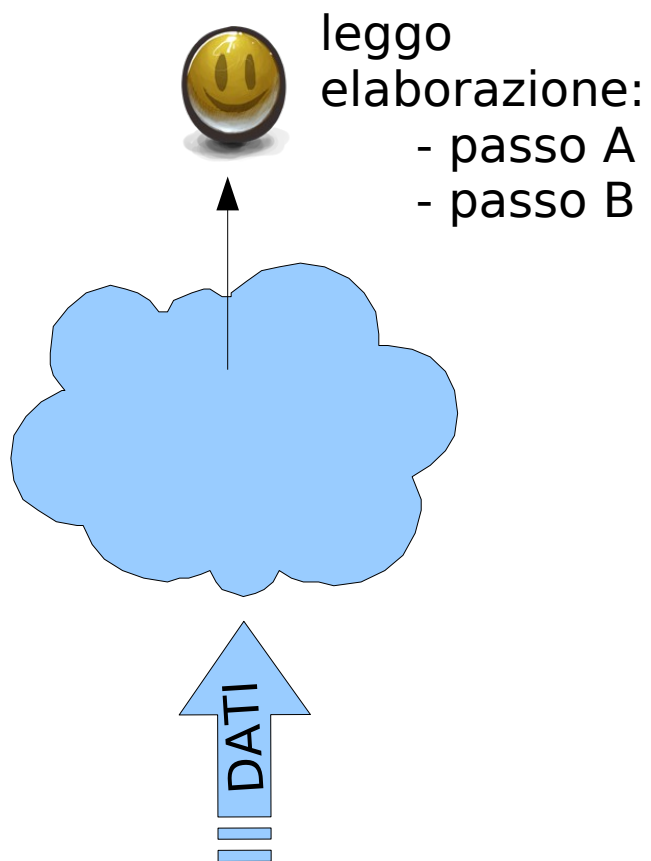
modelli di comunicazione

sezioni 3.4, 3.6.1 e 3.6.2 del libro (VII
ed.)

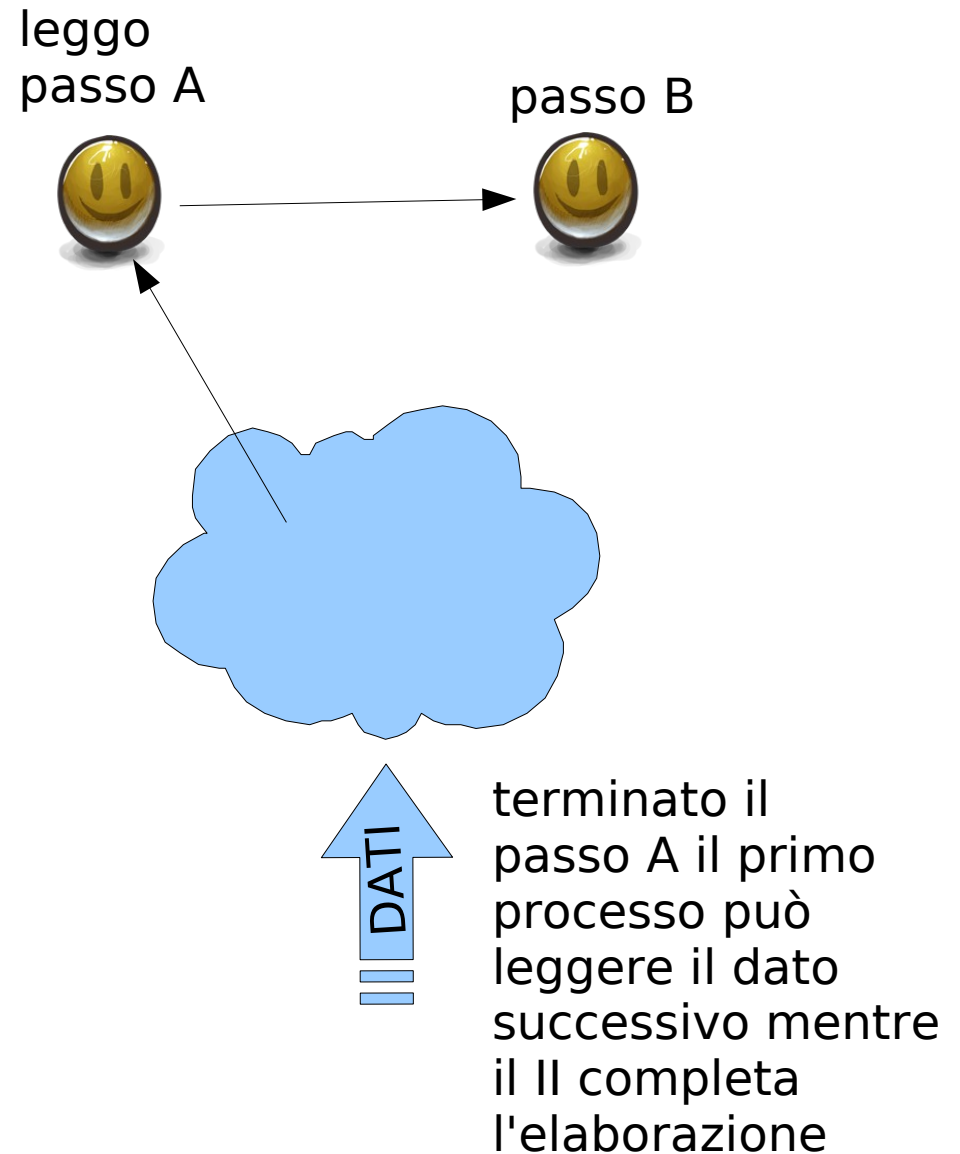
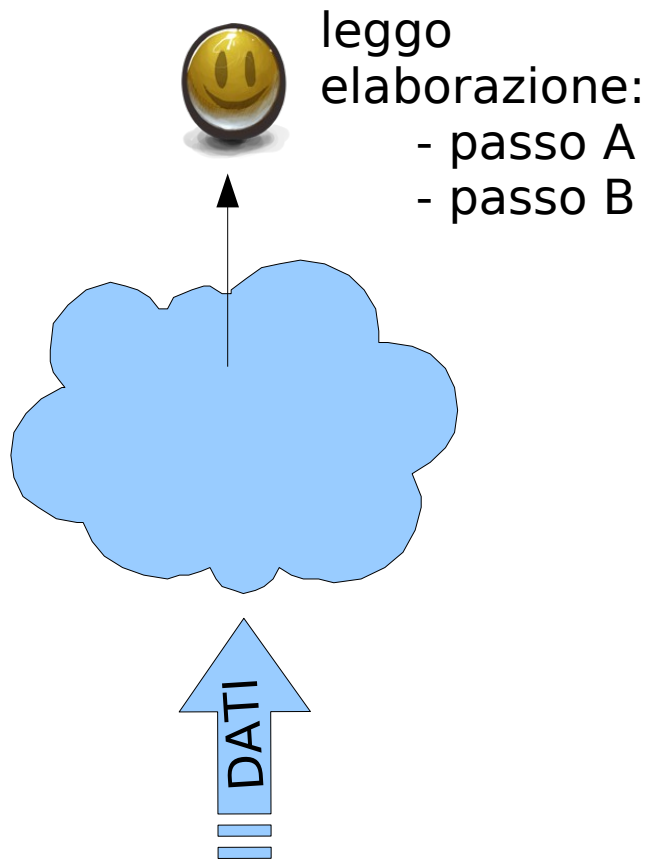
Processi cooperanti

- L'esigenza di far comunicare dei processi nasce quando tali processi cooperano allo svolgimento di un compito
- Un approccio a **processi cooperanti** è vantaggioso rispetto a un sistema a **processi monolitici** perché:
 - **maggiore efficienza**: in molte circostanze è più veloce un'esecuzione a processi paralleli, si sfruttano meglio i tempi morti
 - **accesso concorrente a dati condivisi**: più utenti/applicativi possono dover usare gli stessi dati, un accesso concorrente migliora i tempi di risposta
- **NECESSITÀ**: far comunicare / sincronizzare i processi che interagiscono tramite meccanismi di **"inter-process communication"**
- **due modelli: a memoria condivisa e a scambio di messaggi**

Processi cooperanti



Processi cooperanti



Produttore / consumatore

- Caso molto frequente: un processo produce dei dati che vengono consumati da un altro processo
- Es. un web server produce pagine HTML consumate da un web browser, un compilatore produce codice assembly, consumato da un assembler



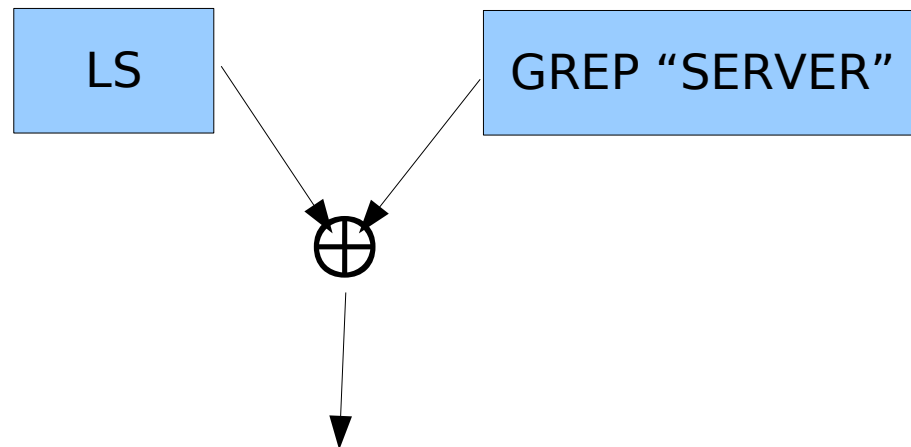
pipeline di processi

come fanno i vari processi a passarsi i dati semi-lavorati?

Esempio

ls: comando Unix che lista il contenuto di una directory

grep: comando Unix che identifica linee di un file che fanno match con un pattern dato (es. che contengono una stringa)



Se potessi far collaborare i due processi potrei fornire all'utente l'elenco dei contenuti di una directory i cui dati rispettano un certo schema!

Memoria condivisa

- tramite system call un processo richiede l'**allocazione di una porzione di memoria accessibile anche ad altri processi**, che dovranno successivamente agganciarla al proprio spazio degli indirizzi
- l'area di memoria può essere "**plasmata**" secondo qualsiasi tipo di dati utile ai programmi comunicanti, per esempio un buffer (di dimensione limitata)

```
#define D 10
```

```
typedef ... elemento;
```

```
typedef struct _b {  
    elemento dato[D];
```

```
    int inserisci, preleva;  
} buffer_cond;
```

spazio per i dati
condivisi

entrambi inizializzati a 0

Memoria condivisa

(1)



(2)



I processi hanno bisogno di dati in un formato specifico, devono sapere come interpretare i byte condivisi

```
typedef struct _b {  
    elemento dato[D];  
    int inserisci, preleva;  
} buffer_cond;
```

I processi condividono un tipo di dato: il tipo di dato secondo il quale vanno interpretati i byte della memoria condivisa

(3)

Memoria condivisa

PRODUTTORE

...

alloca b di tipo buffer_cond come memoria condivisa

```
while (1) {  
    if (! pieno(b) ) {  
        nuovo = ... produci ...;  
        b.dato[b.inserisci] = nuovo;  
        b.inserisci = (b.inserisci+1) % D;  
    }  
}
```

CONSUMATORE

...

aggancia b al proprio spazio indirizzi

```
while (1) {  
    if (! vuoto(b) ) {  
        nuovo = b.dato[b.preleva];  
        b.preleva = (b.preleva+1) % D;  
    }  
}
```


Memoria condivisa

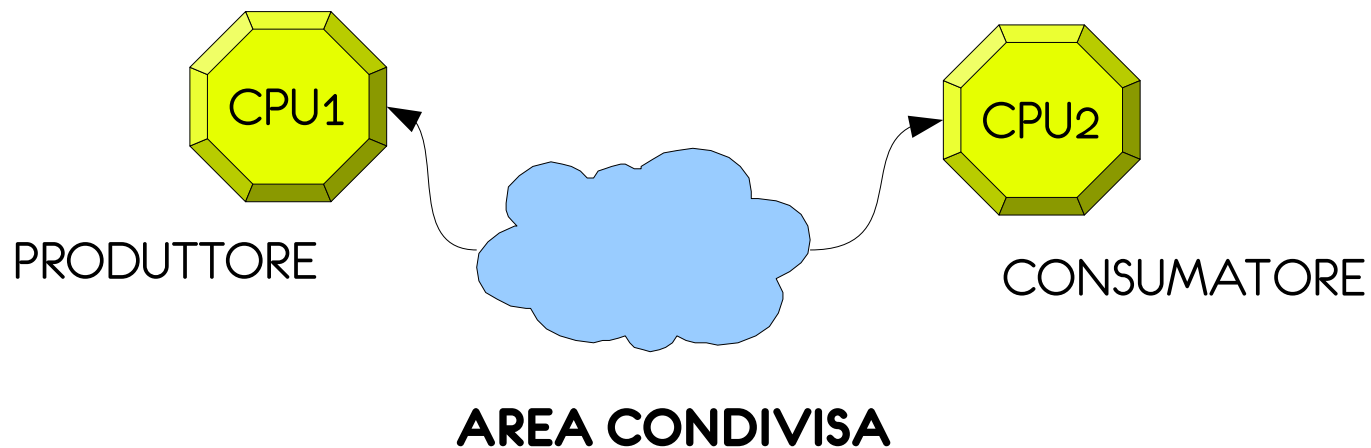
- Gli accessi vanno controllati per evitare inconsistenze!!!

```
pieno(b): (b.inserisci+1)%D == b.preleva  
vuoto(b): (b.inserisci == b.preleva)
```

- Se produttore ha eseguito `b.dato[b.inserisci] = nuovo` ma non ancora `b.inserisci = (b.inserisci+1) % D` consumatore può ritenere il buffer vuoto, erroneamente
- Peggio ancora: e se ci fossero tanti consumatori? Se il buffer contiene un solo elemento e un consumatore ha già eseguito `nuovo = b.dato[b.preleva]` ma non ancora `b.preleva = (b.preleva+1) % D` un altro consumatore potrebbe ritenere il buffer erroneamente pieno!!!

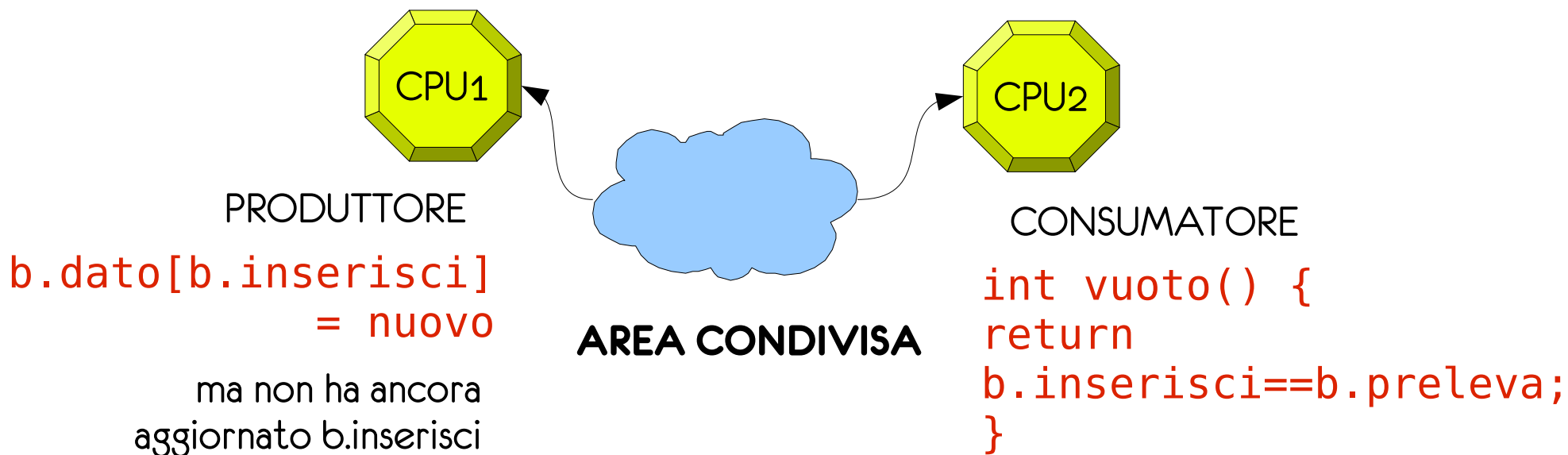
Inconsistenza dei dati

- Abbiamo visto che le inconsistenze dei dati del tipo descritto possono essere causate dallo scheduling della CPU
- Se avessimo due processori, uno per il produttore e uno per il consumatore, il problema potrebbe presentarsi comunque?



Inconsistenza dei dati

- Sì
- Supponiamo che all'inizio
`b.inserisci == b.preleva == 0`
- è un problema intrinseco all'**interleaving** delle istruzioni del produttore e del consumatore



Scambio di messaggi

- Consente a due processi di comunicare senza condividere una stessa area di memoria. Questo meccanismo può essere caratterizzato in modi diversi, a livello logico:
 - **diretto** o **indiretto**: è diretto se un processo deve fornire il PID del processo con cui desidera comunicare
 - **sincrono** (bloccante) o **asincrono** (non bloccante):
 - **invio sincrono**: il mittente si blocca in attesa che il ricevente riceva il messaggio
 - **recezione sincrona**: il ricevente rimane in attesa di un messaggio fintantoché non ne viene effettivamente ricevuto uno
 - **rendez vous**: invio sincrono + recezione sincrona
 - a gestione automatica o esplicita del buffer

Send e receive dirette

- A livello logico due processi che intendono comunicare devono essere connessi da un **canale**. Per scambiarsi messaggi usano **send** e **receive**
-



- **comunicazione diretta:**
 - **send(P, msg)**: invia msg al processo P
 - **receive(P, msg) / receive(id, msg)**: attendi un messaggio dal processo P, tale messaggio verrà memorizzato in msg oppure ricevi un messaggio e salva in id il PID del mittente e in msg il messaggio
 - **la reciproca conoscenza del PID definisce un canale logico**

Send e receive indirette

- in questo caso l'invio/recezione sono effettuati non a processi ma a porte o mailbox, distinte dall'identità del ricevente
-



-
- **comunicazione indiretta:**
 - `send(M, msg)`: invia msg alla mailbox M
 - `receive(M, msg)`: attendi un messaggio alla mailbox M
 - il canale logico è definito dalla mailbox
 - NB: più mittenti/riceventi possono usare la stessa mailbox, una stessa coppia di processi può usare diverse mailbox per comunicare

Buffering dei messaggi

- la mailbox una una capacità



- **tipi di buffering:**
 - **capacità 0:** il canale non ha memoria (meccanismo no buffering); il mittente rimane sospeso se il ricevente non ha ancora consumato il messaggio ricevente (gestione esplicita del buffer)
 - **capacità $N > 0$:** il mittente rimane in attesa solo se il buffer è pieno (meccanismo automatic buffering)
 - **capacità illimitata:** il mittente non attende mai (meccanismo automatic buffering)

Un paio di esempi

- **socket**: definizione di un canale di comunicazione fra processi in esecuzione su macchine diverse
- **remote procedure call**: invocazione di una procedura definita ed eseguita da un altro sistema

Socket

- usato in sistemi client-server
- **socket**: è il nome dato a un estremo (endpoint) di un canale di comunicazione fra due processi
- due processi interagenti in rete usano una coppia di socket (uno per ciascuno), ciascuno dei quali ha per identificatore l'IP della macchina concatenato a un identificatore di porta:



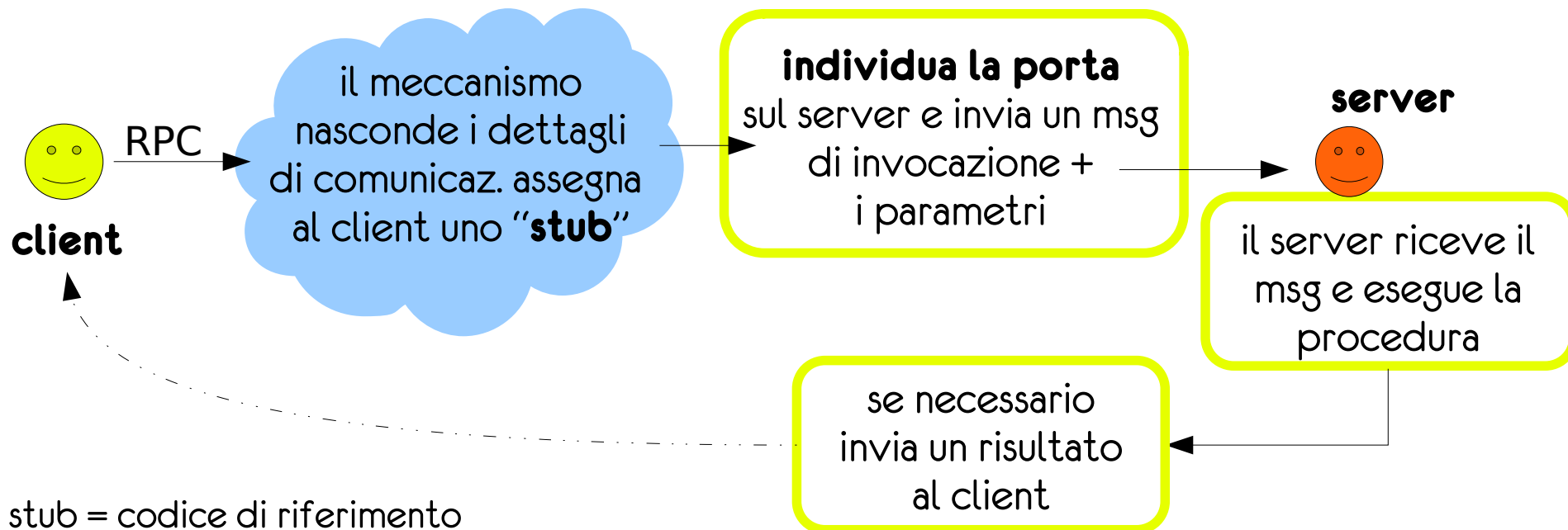
- Esempio di id: **140.228.112:80** la parte in blu (140.228.112) è l'IP di una macchina, quella in rosso (80) un numero di porta, sono concatenati da un due punti.
- I messaggi sono semplici pacchetti dati non strutturati

Socket

- tutte le porte < 1024 sono riservate
- un processo utente che richieda una porta riceverà un numero maggiore di 1024 e lo stesso numero non potrà essere assegnato a due processi diversi. La coppia **IP:PORTA** è un identificatore univoco.
- esempi di porte predefinite:
 - telnet 23
 - ftp 21
 - web 80
- Caso particolare: tramite l'**indirizzo di loopback 127.0.0.1** un computer può fare riferimento a se stesso, quindi il meccanismo visto è usabile anche per far comunicare processi diversi su di uno stesso computer.

Remote Procedure Call

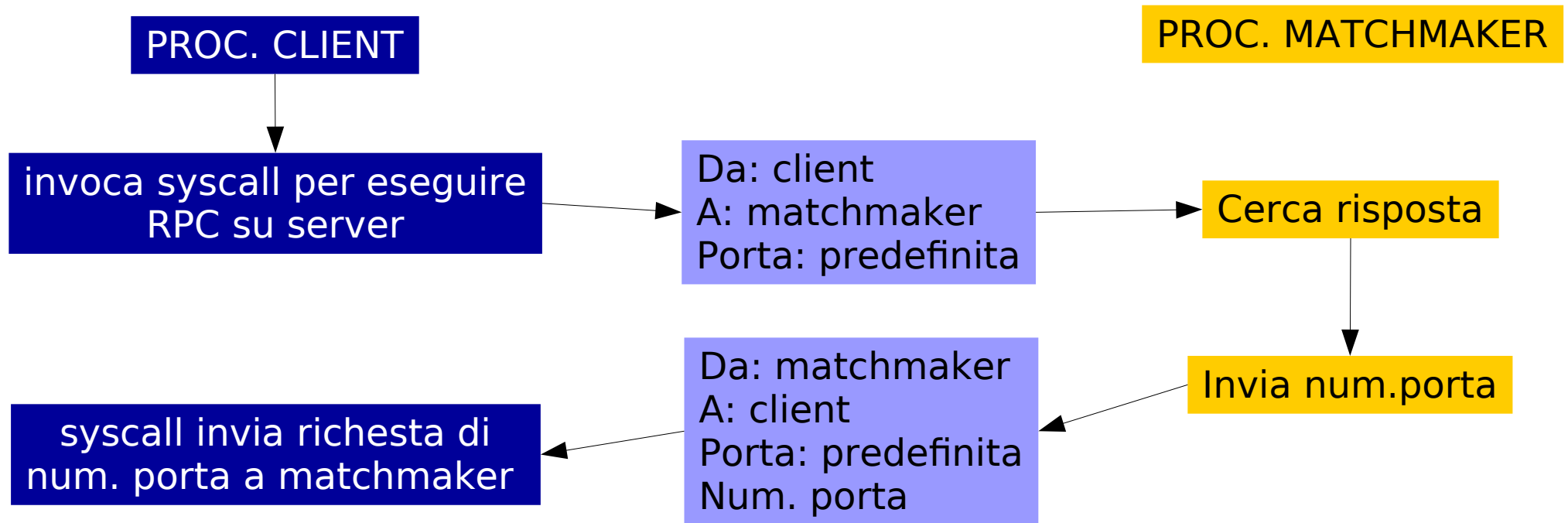
- meccanismo usato in sistemi client-server
- in certi casi è utile consentire a un processo di invocare l'esecuzione di una procedura che risiede su di un'altra macchina connessa in rete: meccanismo noto come **remote procedure call** (RPC)
- i messaggi sono ben strutturati, sono costituiti dall'identificatore della procedura da eseguire e dai parametri su cui viene invocata



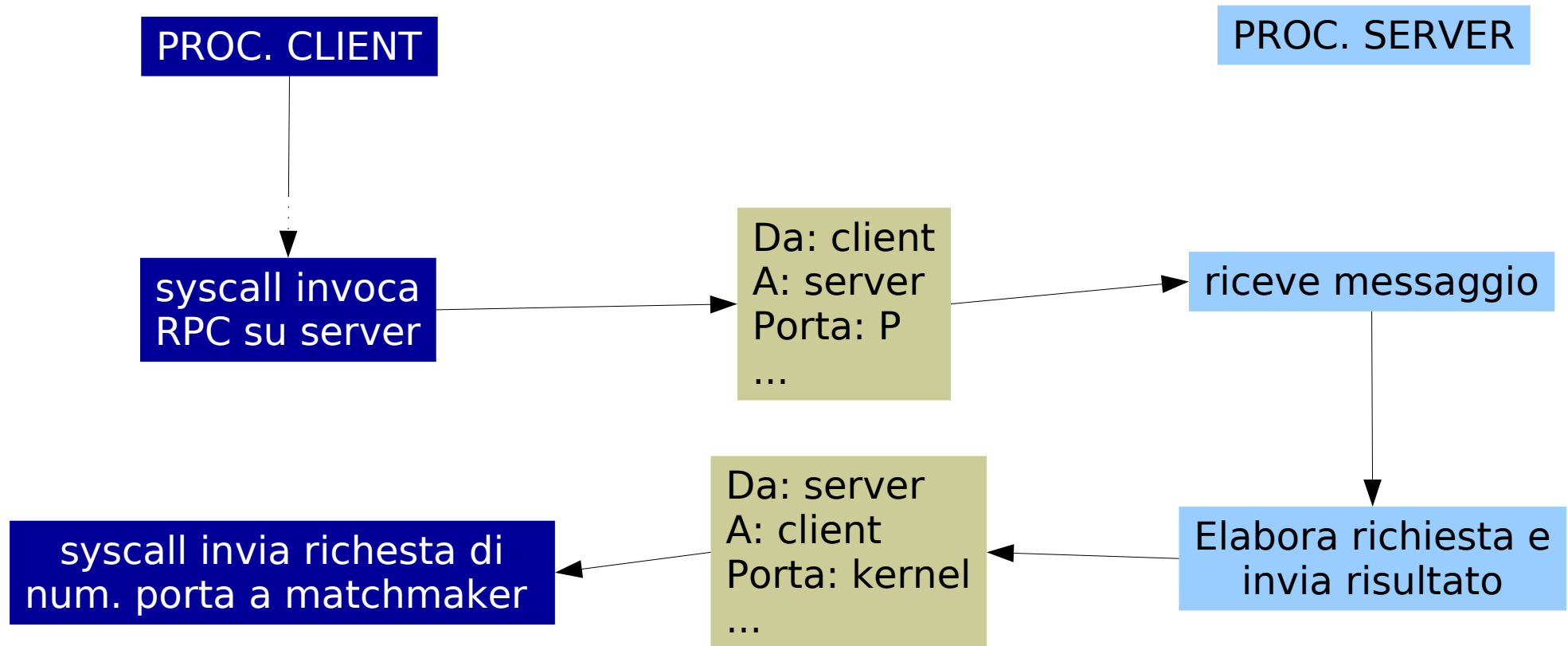
Associazione porte a RPC

- Il client deve conoscere l'associazione fra le RPC di un server e le relative porte.
Come/quando avviene tale associazione?
- Client e server non condividono memoria!
- Soluzioni
 - **Associazione Predefinita**: fissata in fase di compilazione delle RPC
 - **Associazione Dinamica**:
 - si introduce un servizio intermedio di rendez-vous, detto "**matchmaker**" invocabile su di una porta fissa
 - **Interazione con due demoni**: il matchmaker e il demone in ascolto sulla porta identificata

Interazione col matchmaker



Interazione col server



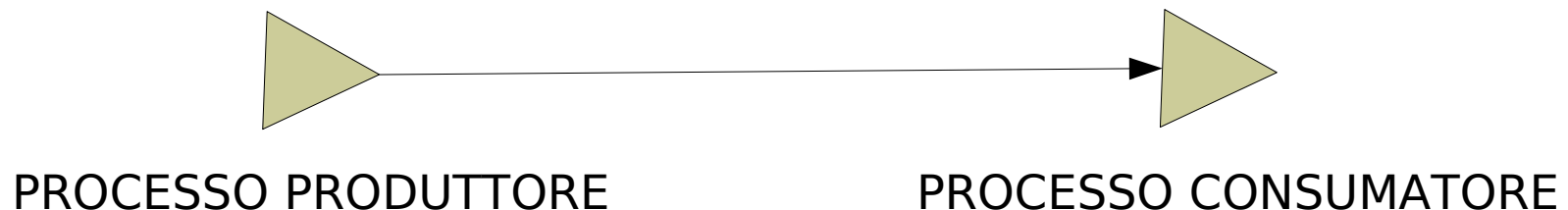
Remote Procedure Call

- **Problemi**

- 1) macchine diverse possono adottare rappresentazioni diverse dei dati!! **Soluzione:** usare un formato **intermedio**, es. **XDR** (external data representation). Il client converte i dati in XDR, il server converte da XDR al proprio formato
- 2) una RPC va eseguita una e una sola volta però come essere sicuri che l'esecuzione è avvenuta? E se problemi di connessione facessero perdere la richiesta? **Soluzione:**
 - a. il server archivia tutte le richieste pervenute associando loro un timestamp (sicurezza che una richiesta venga eseguita al + 1 volta)
 - b. meccanismo di riveuta: il client continua a reinviare la richiesta fino a quando non riceve una ricevuta dal server

Pipe

- Canale di comunicazione fra processi
- **Pipe anonima:**
 - Canale simplex, FIFO
 - interazione basata sul meccanismo produttore consumatore
 - Estremità di scrittura, estremità di lettura (unidirezionalità)
 - Consentono la comunicazione fra una singola coppia di processi, tipicamente un padre crea una pipe anonima e la usa per interagire con un figlio
 - Non sopravvive al processo creatore



Named pipe

- **Named Pipe (o FIFO):**
 - interazione basata sul meccanismo produttore consumatore
 - FIFO
 - In Unix è unidirezionale, in Windows è bidirezionale
 - Consente la comunicazione di più di due processi
 - Sopravvive alla terminazione del processo creatore
 - Va disallocata esplicitamente
 - Spesso realizzata come file
 - VMware virtualizza le porte tramite named pipe

Process tree

- Per visualizzare l'albero dei processi si possono utilizzare, in alternativa, i comandi:
 - **ps axjf**
 - **ps -ejH**

ESEMPIO DI OUTPUT

```
1913 tty2      Sl+  /usr/lib/gnome-terminal/gnome-terminal-server
1945 pts/0     Ss   \_ bash
2142 pts/0     S+   |  \_ alpine
20606 pts/1    Ss   \_ bash
20798 pts/1    R+   |  \_ ps f
20742 pts/2    Ss   \_ bash
20778 pts/2    S+       \_ man vmstat
20790 pts/2    S+       \_ pager
1882 tty2      Sl+   /usr/lib/tracker/trac
```

“vedere” code e mem. Cond.

baroglio@rhialto:~\$ **ipcs**

----- Message Queues -----

| key | msqid | owner | perms | used-bytes | messages |
|-----|-------|-------|-------|------------|----------|
|-----|-------|-------|-------|------------|----------|

----- **Shared Memory Segments** -----

| key | shmid | owner | perms | bytes | nattch | status |
|------------|--------------|--------------|-------|--------------|---------------|--------|
| 0x00000000 | 131072 | baroglio | 600 | 524288 | 2 | dest |
| 0x00000000 | 229377 | baroglio | 600 | 4194304 | 2 | dest |
| 0x00000000 | 393218 | baroglio | 600 | 524288 | 2 | dest |
| 0x00000000 | 294915 | baroglio | 600 | 67108864 | 2 | dest |
| 0x00000000 | 1605636 | baroglio | 600 | 524288 | 2 | dest |
| 0x00000000 | 13172741 | baroglio | 600 | 2304 | 2 | dest |
| 0x00000000 | 6717446 | baroglio | 600 | 36864 | 2 | dest |
| 0x00000000 | 2392071 | baroglio | 600 | 4915200 | 2 | dest |

...

----- Semaphore Arrays -----

| key | semid | owner | perms | nsems |
|-----|-------|-------|-------|-------|
|-----|-------|-------|-------|-------|

“vedere” code e mem. Cond.

```
baroglio@rhialto:~$ ipcs -cu
```

```
----- Messages Status -----
```

```
allocated queues = 0
```

```
used headers = 0
```

```
used space = 0 bytes
```

```
----- Shared Memory Status -----
```

```
segments allocated 30
```

```
pages allocated 30971
```

```
pages resident 6906
```

```
pages swapped 0
```

```
Swap performance: 0 attempts 0
```

```
successes
```

```
----- Semaphore Status -----
```

```
used arrays = 0
```

```
allocated semaphores = 0
```

thread

capitoli 4 e 5.5 del libro (VII ed.)

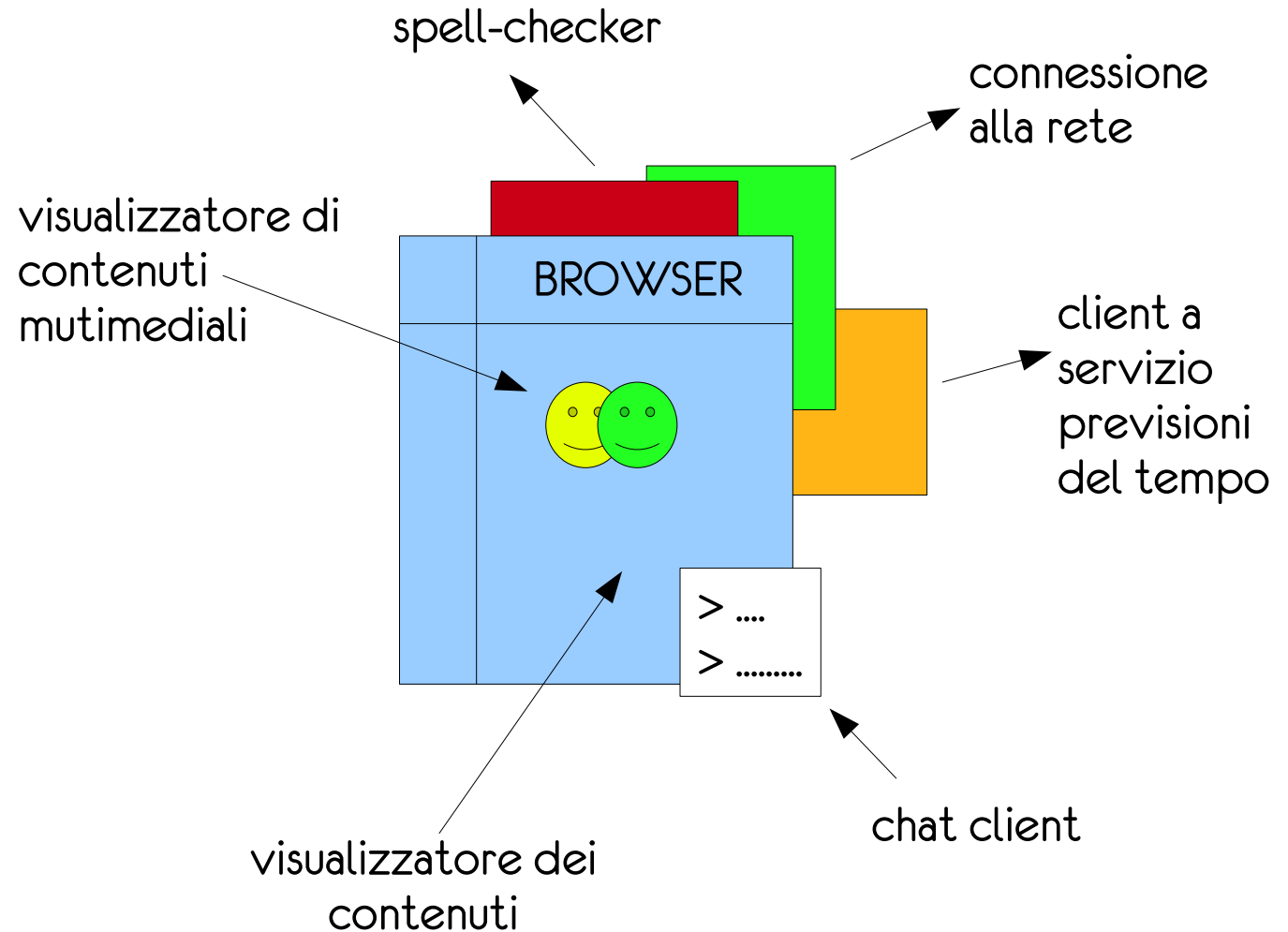
+

cap. 4 di Sistemi Operativi, di H. Deitel,
P. Deitel e D. Choffnes

Introduzione

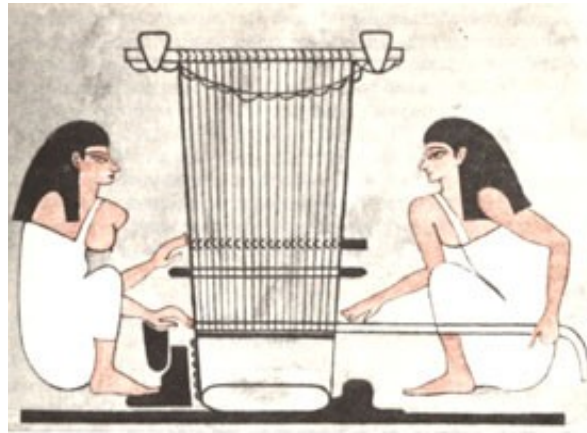
io avvio il programma
con un click oppure
digitando il nome del
browser in una shell ...

**ma tutto questo è un
processo solo?**



Thread

Thread: esecuzione sequenziale di codice



Ткачество на фресках Бени-Хасана

Thread

```
Main() {  
    int ris = 0;  
    ...  
    ris = f1(x1, x2, ...) + f2(y1, y2, ...);  
}
```

Se f1 e f2 non hanno dipendenze, eseguire l'una prima dell'altra non fa differenza. Allora è anche possibile pensare a una loro esecuzione parallela ...

f1 e f2 condividono il loro contesto di esecuzione, ha senso pensare a combinare due processi per consentirne l'esecuzione parallela?

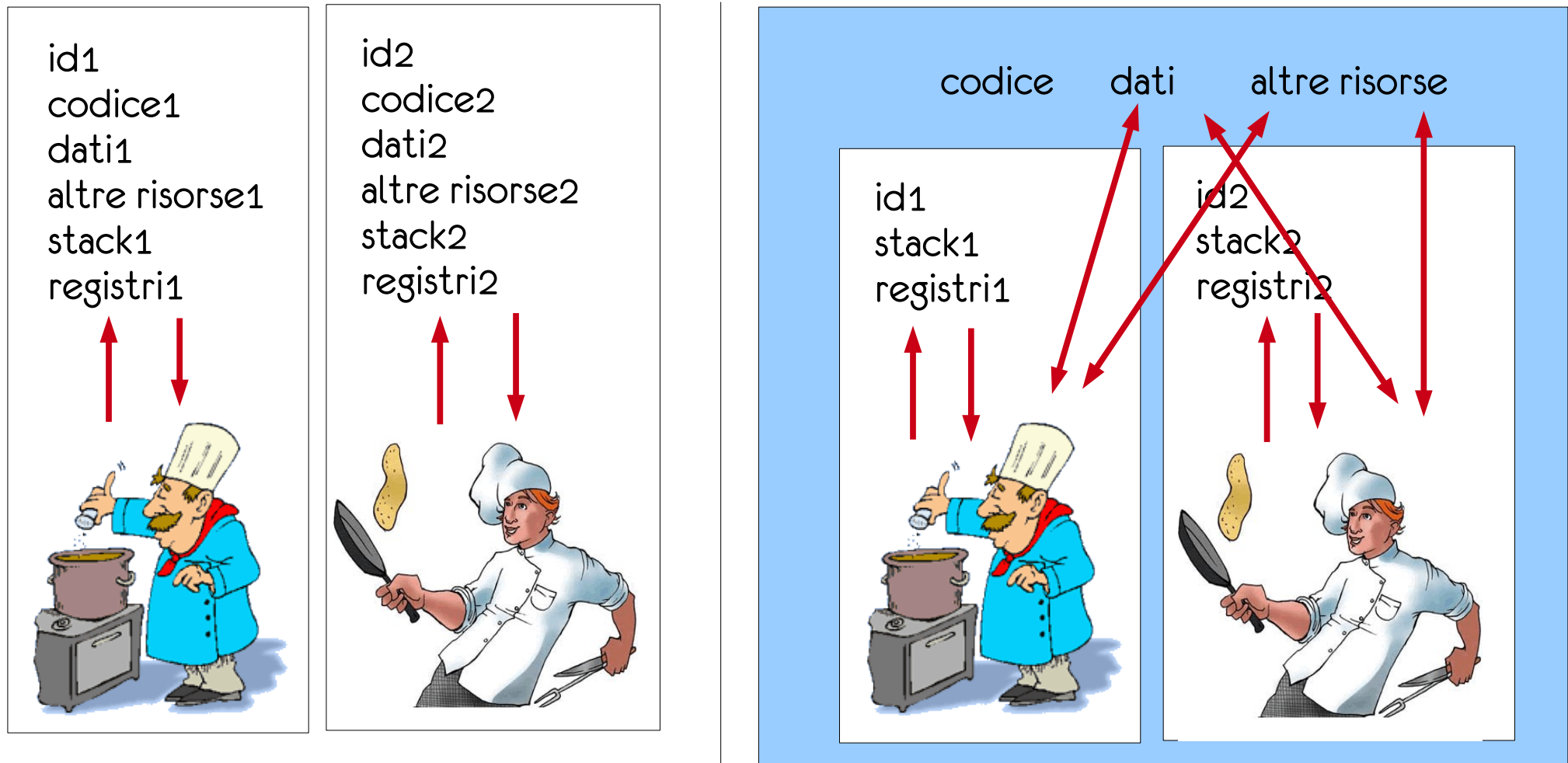
Introduzione

- molti SO moderni considerano come **unità di base d'uso della CPU** non il processo ma il **THREAD**. Un processo può essere organizzato in un insieme di thread cooperanti

Thread:

- è costituito da: un identificatore, un program counter, un insieme di valori di registri, uno stack
 - condivide con gli altri thread dello stesso processo: il codice, la sezione dati, file aperti, segnali e altre risorse di sistema
-
- Un processo costituito da un solo thread è detto **heavyweight process**, processo pesante

Cose proprie e comuni

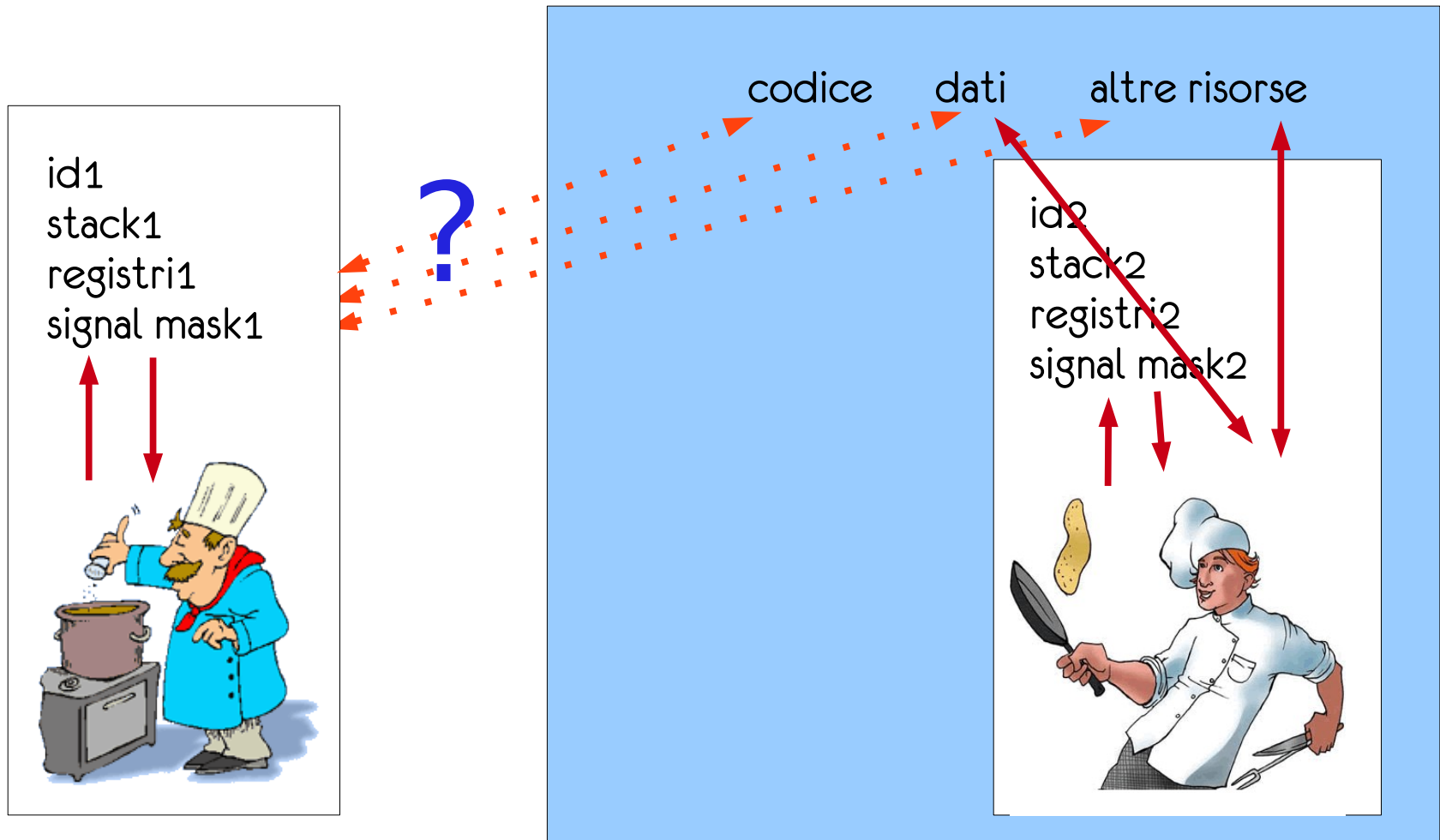


processi

thread di uno stesso processo

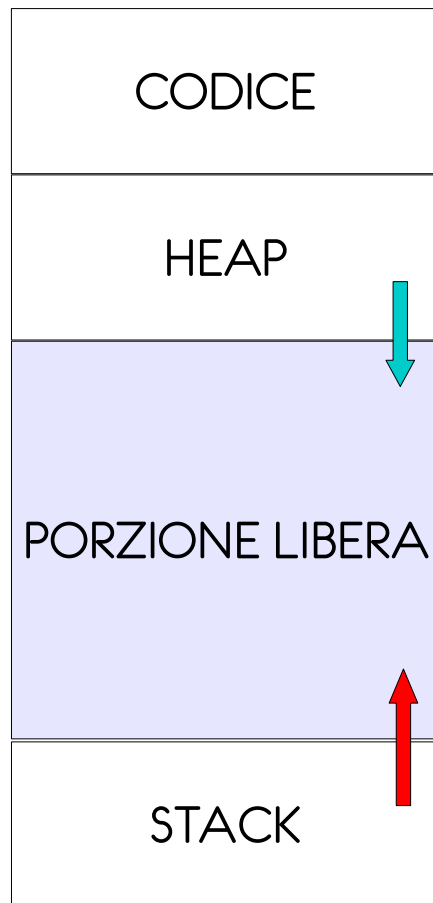
comunicano tramite un meccanismo a
memoria condivisa!!

Thread senza processi?

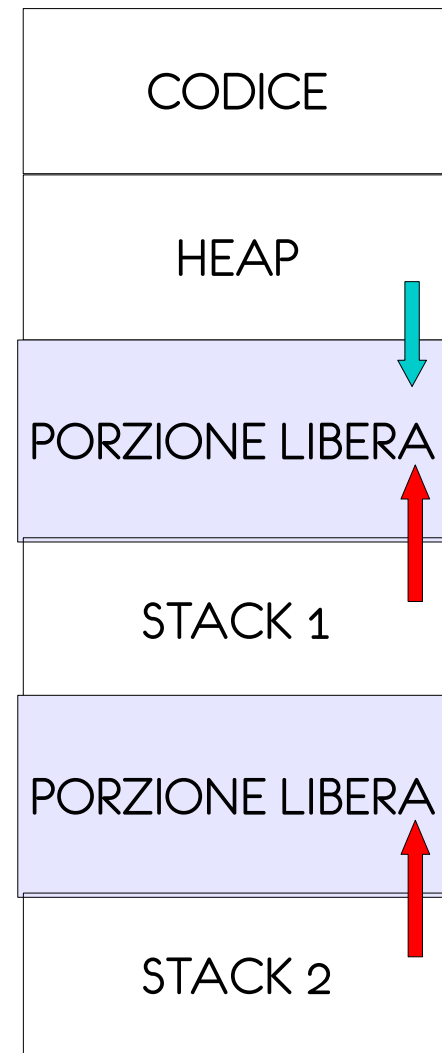


un thread non può esistere al di fuori di un processo perché non contiene tutte le informazioni necessarie per poter effettuare l'esecuzione: codice, dati, risorse fanno parte del processo

Organizzazione della memoria



Processo monolitico



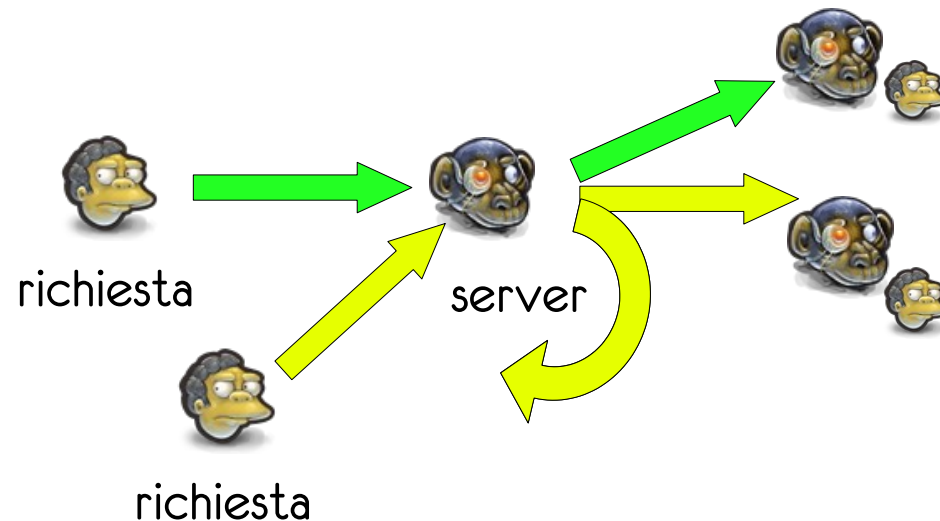
Processo con due thread

Vantaggi

- Una soluzione a thread di solito è più efficiente di una soluzione a processi cooperanti
- Molti programmi contengono sezioni di codice eseguibili indipendentemente dal resto del programma.
- Incremento delle prestazioni:
 - Il **context switch** è più rapido,
 - è richiesta l'allocazione di una **quantità inferiore di risorse** (le condividono),
 - la **comunicazione è più veloce** perché avviene tramite variabili condivise (spesso non occorre appoggiarsi a strutture di IPC)
- in architetture multicore diventa possibile spezzare l'esecuzione di un processo su più processori

Esempio

- molti server (RPC, web, ...) sono implementati a thread:
l'arrivo di una nuova richiesta comporta la **creazione** di un thread servitore, dedicato a soddisfare quella richiesta mentre il thread principale si rimette in attesa di nuove richieste



- Server per cui il tempo di esecuzione è critico, sono talvolta organizzati in un pool di thread creati all'avvio.
- Tali server possono gestire in parallelo al più un numero di richieste pari alla cardinalità del pool di thread.

Thread e linguaggi

- I thread sono definiti dal programmatore, creati e gestiti da programma
 - Molti linguaggi di programmazione offrono specifiche istruzioni per la creazione e il controllo dei thread e per controllare l'accesso alle variabili condivise
 - **Esempio:**
Java, C#, Python Programmazione III
-
- Al contrario i processi sono per lo più generati in modo invisibile all'utente, i linguaggi di programmazione non forniscono costrutti sintattici ad hoc e occorre l'esplicita invocazione di system call
 - Linguaggi come C e C++ sono detti a singolo flusso di controllo
 - **NB:** anche questi linguaggi possono essere usati per scrivere programmi a multithread ma richiedono l'inclusione di apposite librerie

Esempio: Android

- Il SO Android fa dei thread un elemento centrale della programmazione
- Tutte le componenti di ogni applicativo sono realizzate come diversi thread di uno stesso processo
- All'avvio di un applicativo, viene generato il thread main che ha l'importante funzione di effettuare il dispatch degli eventi alle diverse componenti dell'interfaccia grafica



"vedere" I thread

baroglio@rhialto:~\$ **ps m -L | more**

| PID | LWP | TTY | STAT | TIME | COMMAND |
|------|------|------|------|------|--|
| ... | | | | | |
| 1541 | - | tty2 | - | 0:00 | /usr/lib/gdm3/gdm-x-session ... |
| - | 1541 | - | Ssl+ | 0:00 | - |
| - | 1542 | - | Ssl+ | 0:00 | - |
| - | 1555 | - | Ssl+ | 0:00 | - |
| 1543 | - | tty2 | - | 2:01 | /usr/lib/xorg/Xorg vt2 -displayfd 3 -auth... |
| - | 1543 | - | Sl+ | 2:00 | - |
| - | 1544 | - | Sl+ | 0:00 | - |
| 1554 | - | tty2 | - | 0:00 | dbus-daemon --print-address 4 --session |
| - | 1554 | - | S+ | 0:00 | - |
| 1557 | - | tty2 | - | 0:00 | /usr/lib/gnome-session/gnome-session-binary... |
| - | 1557 | - | Sl+ | 0:00 | - |
| - | 1694 | - | Sl+ | 0:00 | - |
| - | 1695 | - | Sl+ | 0:00 | - |
| - | 1697 | - | Sl+ | 0:00 | - |
| 1654 | - | tty2 | - | 0:00 | /usr/lib/gvfs/gvfsd |
| - | 1654 | - | Sl+ | 0:00 | - |
| - | 1655 | - | Sl+ | 0:00 | - |
| ... | | | | | |

“vedere” I thread: top

```
baroglio@rhaltto: ~  
File Edit View Search Terminal Tabs Help  
alpine x baroglio@rhaltto: ~ x baroglio@rhaltto: ~ x  
top - 11:45:57 up 2:34, 1 user, load average: 0,22, 0,13, 0,11  
Tasks: 248 total, 2 running, 245 sleeping, 0 stopped, 1 zombie  
%Cpu(s): 2,9 us, 1,0 sy, 0,0 ni, 95,8 id, 0,3 wa, 0,0 hi, 0,0 si, 0,0 st  
KiB Mem : 8078804 total, 3319236 free, 1880068 used, 2879500 buff/cache  
KiB Swap: 8290300 total, 8290300 free, 0 used. 5557712 avail Mem  
  
  PID USER      PR   RES    SHR S  COMMAND            PPID  nTH  TGID  
 1739 baroglio  20 209264  63320 S  gnome-shell        1557   8   1739  
 1543 baroglio  20 120396 104144 R  Xorg               1541   2   1543  
21378 baroglio  20  37612  26964 S  gnome-screensho    1      5  21378  
 1913 baroglio  20  49344  29860 S  gnome-terminal-    1      4   1913  
 2651 baroglio  20 705588 128260 S  firefox            1     50  2651  
    1 root      20    6116   3988 S  systemd            0      1     1  
   891 root      20   10708   6080 S  polkitd            1      3   891  
 1554 baroglio  20    4968   3484 S  dbus-daemon        1541   1  1554  
 1557 baroglio  20   14824  12708 S  gnome-session-b    1541   4  1557  
 1648 baroglio  20    9340   5408 S  ibus-daemon        1      4  1648  
 1793 baroglio  20   10628   8904 S  mission-control    1      4  1793  
21182 baroglio  20    3856   3152 R  top                20606  1  21182  
21353 baroglio  20   24040  10712 S  dley-na-server-s   1      5  21353  
    2 root      20      0      0 S  kthreadd           0      1     2  
    3 root      20      0      0 S  ksoftirqd/0        2      1     3  
    5 root      0      0      0 S  kworker/0:0H        2      1     5  
    7 root      20      0      0 S  rcu_sched          2      1     7
```

fork exec e thread

- abbiamo visto le system call per i processi: fork ed exec
- un processo multithread può usare queste system call? Che effetto hanno?
- **fork:**
 - ??
- **exec:**
 - ??

fork exec e thread

- abbiamo visto le system call per i processi: fork ed exec
- un processo multithread può usare queste system call? Che effetto hanno?
- **fork:**
 - in certi SO causa la creazione di un nuovo processo con la duplicazione di tutti i thread
 - in altri SO causa la sola duplicazione del thread chiamante
- **exec:**
 - causa la sovrascrittura del codice dell'intero processo con un nuovo programma: NB, tutti i thread vengono sostituiti!

Operazioni sui thread

- **creazione:** implica la creazione di una struttura dati specifica che mantiene le informazioni relative al nuovo thread;
- **terminazione:** è più rapida di quella dei processi perché, per es., non richiede la gestione delle risorse
- sospensione/blocco
- recupero/risveglio
- **join:** specifica per 1 thread, comporta l'attesa da parte di un thread della terminazione di un altro

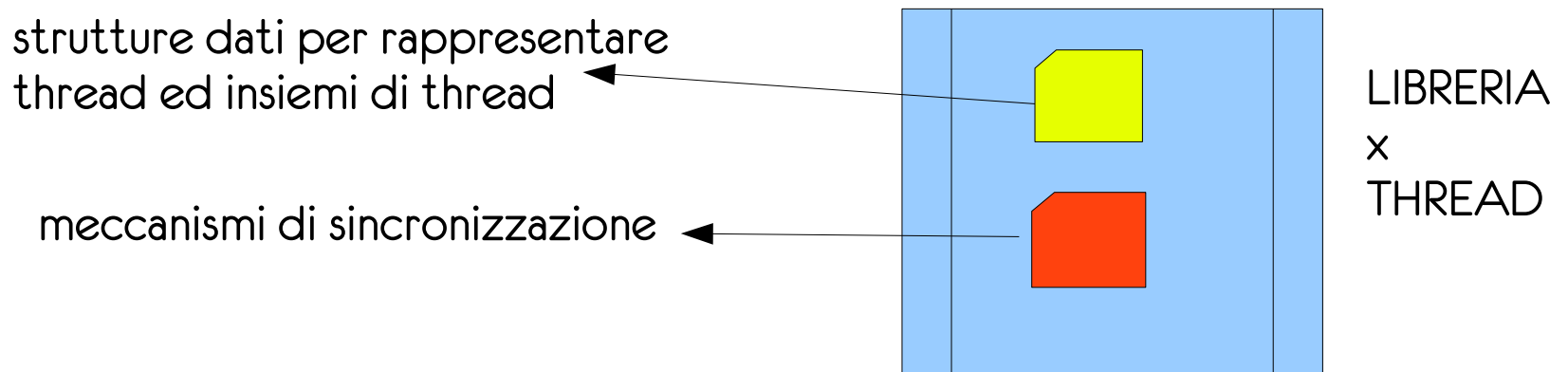
T1 si blocca sulla join fino a quando T2 non termina

```
T1:  
...  
join(T2)  
...
```

```
T2:  
...  
thread_exit()
```

Modelli di thread (a)

- **Primi sistemi operativi:** consentivano un solo contesto di esecuzione per processo -> ogni processo multithread deve gestire le info dei suoi thread, il loro scheduling, la comunicazione fra i suoi thread.
- Si parla di **thread a livello utente**
- Sono creati da funzioni di libreria che non possono eseguire istruzioni privilegiate
- Sono trasparenti al SO, che vede il processo come una sola entità indistinta



Modelli di thread (a)

- **Vantaggi:**

- I thread a livello utente sono portabili anche su SO che non prevedono il multi-threading perché sono gestiti internamente al processo, ad un livello di astrazione più alto;
- I criteri per effettuare lo scheduling possono essere facilmente adattati alle esigenze dello specifico programma;
- esecuzione più rapida in quanto non richiede né l'uso di interruzioni (e dei context switch che conseguono alla loro gestione) né l'invocazione di system call

- **Svantaggi:**

- Non sono adattabili a sistemi multiprocessore
- Se un thread richiede di eseguire un'operazione di I/O tutto il processo rimane bloccato fino al suo termine

Modelli di thread (b)

- Un approccio diverso consiste nell'associare a un processo diversi contesti di esecuzione (corrispondenti ai vari thread), il cui scheduling sulla CPU è gestito esplicitamente dal SO
- **thread a livello kernel**: sono creati e gestiti dal SO, se presenti lo scheduling della CPU è fatto a livello di thread kernel.
- Sono più costosi per il SO perché richiedono che esso mantenga appositi descrittori nonché l'associazione fra tali descrittori e i processi che ne definiscono il contesto di esecuzione
- L'utente genera dei thread a livello utente tramite le apposite librerie e il SO associa a tali thread proprio strutture, che implementano thread a livello kernel

modelli di thread (b)

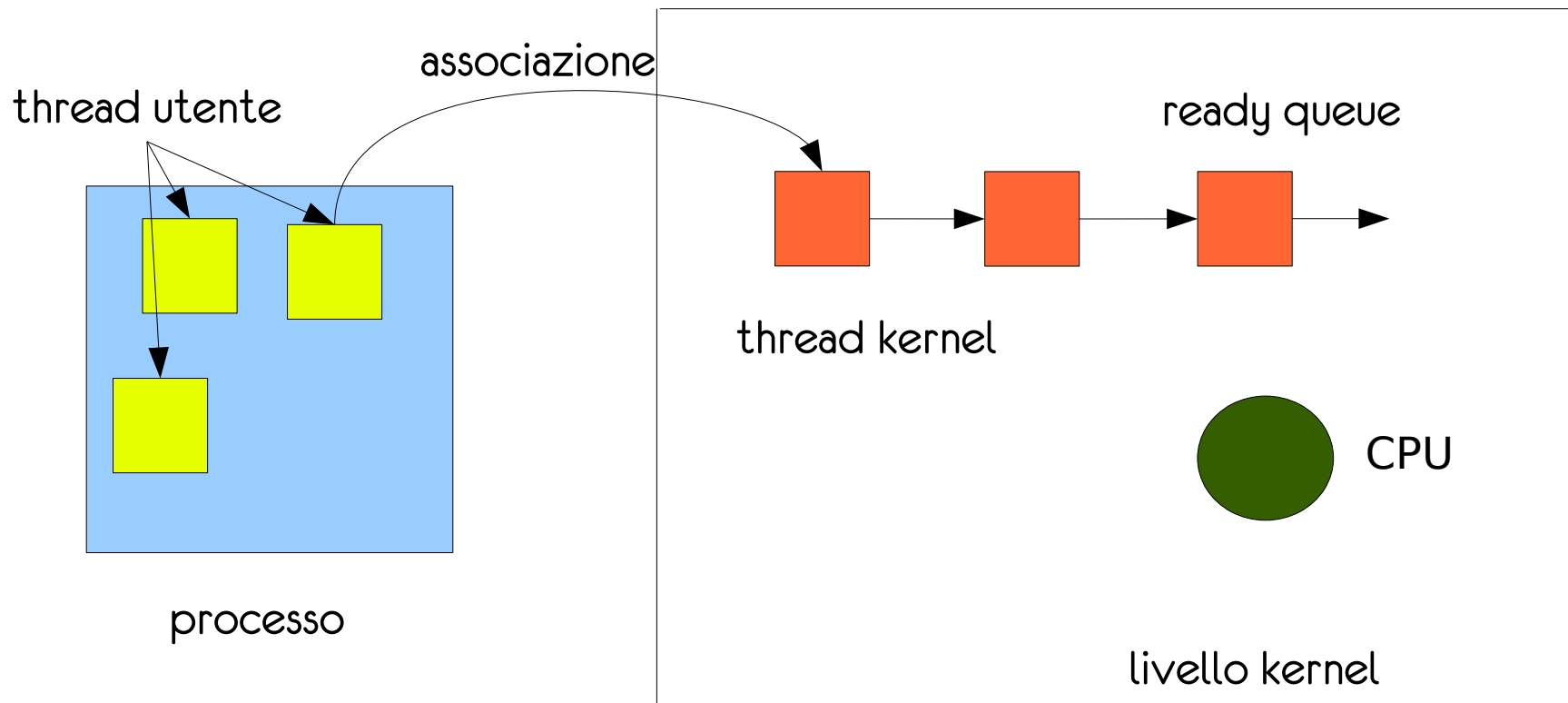
- **Vantaggi:**

- è possibile distribuire i thread di uno stesso processo su più processori (se disponibili)
- singole operazioni di I/O non bloccano processi multithread
- maggiore interattività con l'utente
- migliori prestazioni dei singoli processi

- **Svantaggi:**

- minore portabilità: non portabili su SO non multithread, SO multithread diversi implementano i thread in modo diverso
- in presenza di applicazioni fortemente multithread il SO potrebbe dover gestire migliaia di thread contemporaneamente, ciò potrebbe causare un sovraccarico e un calo di prestazioni

Livello utente ➡ livello kernel



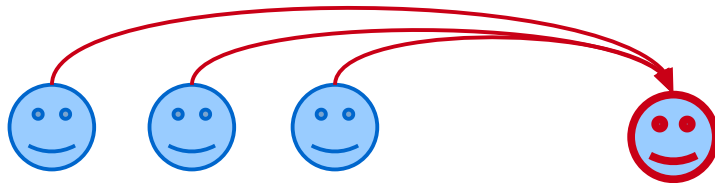
- Per far effettuare lo scheduling della CPU a livello di thread occorre associare thread utente a thread kernel.
- Vi sono **tre modelli** secondo i quali questa associazione può essere fatta: uno a uno (thread kernel), molti a uno (thread utente), molti a molti (soluzione ibrida)

Livello utente e livello kernel

- **uno** (utente) **a uno** (kernel): soluzione adottata da Linux e Windows



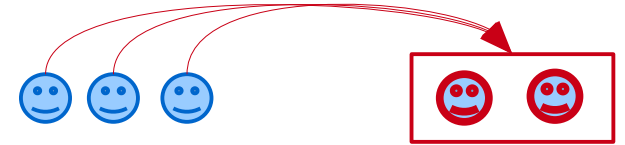
- **molti a uno**: a un pool di thread utente è assegnato un thread kernel. Inefficiente, un solo thread utente per volta può accedere al kernel



- **molti a molti**: a un insieme di thread utente è associato un insieme (di solito + piccolo) di thread kernel. Quando consente anche di vincolare un thread utente a un thread kernel, si parla di modello a due livelli

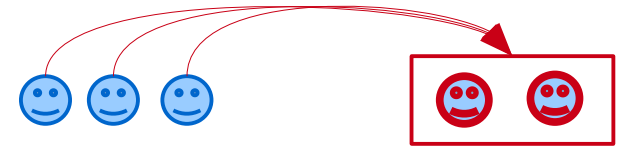


Lightweight process



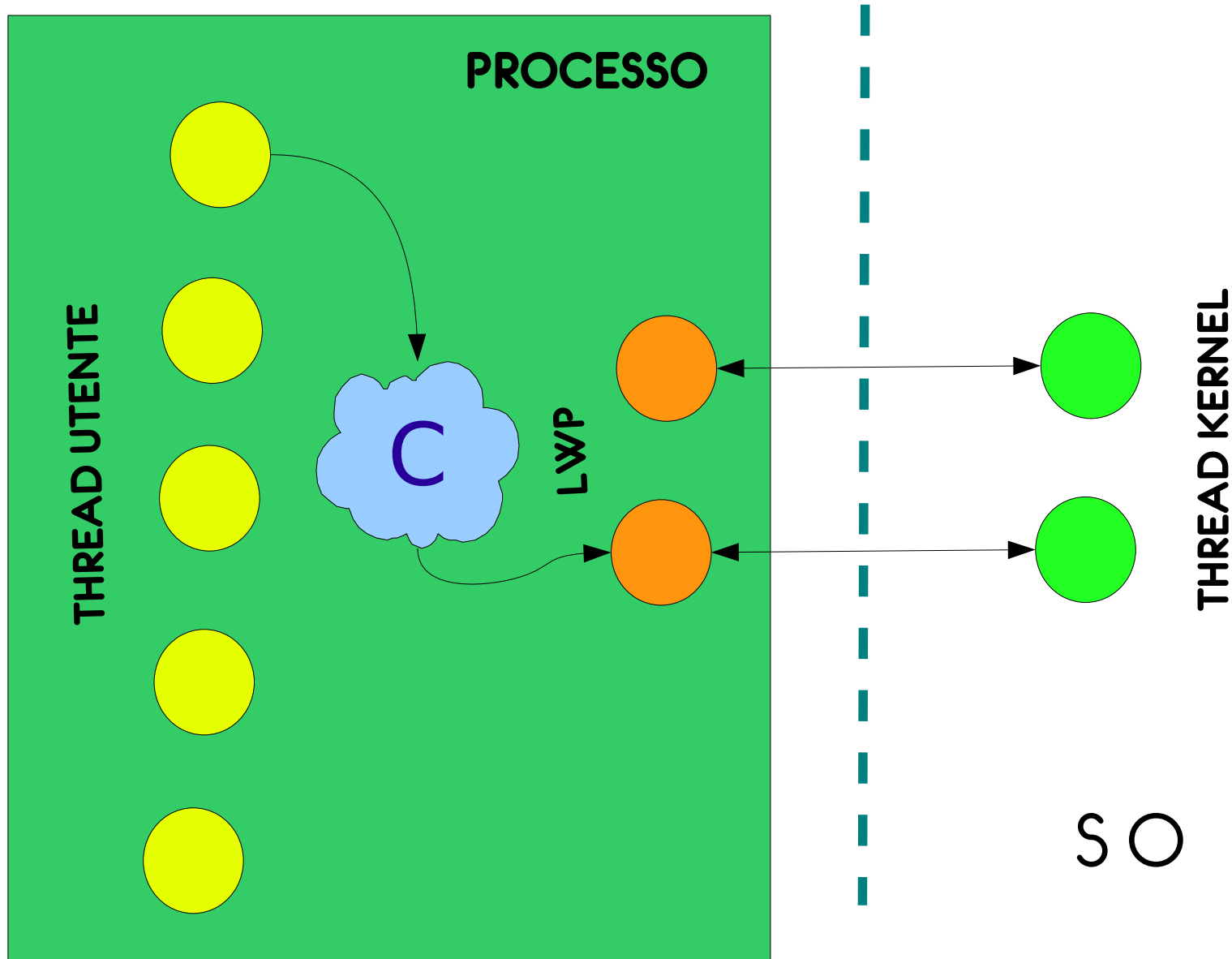
- nei SO che usano il modello molti a molti (o quello a due livelli) si ha la necessità di effettuare uno scheduling dei thread utente per l'accesso ai thread kernel
- questo spesso viene fatto introducendo un nuovo oggetto fra thread utente e thread kernel, detto lightweight process (**LWP**, processo leggero)

Lightweight process

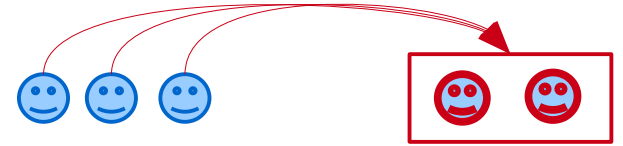


- nei SO che usano il modello molti a molti (o quello a due livelli) si ha la necessità di effettuare uno scheduling dei thread utente per l'accesso ai thread kernel
- questo spesso viene fatto introducendo un nuovo oggetto fra thread utente e thread kernel, detto lightweight process (**LWP**, processo leggero)
- Gli LWP sono visti dagli applicativi utente come **processori virtuali** (un vero e proprio pool di risorse ad essi assegnate) e sono usati per effettuare lo scheduling dei thread utente al kernel. Si parla di **associazione indiretta**.
- **Ogni LWP corrisponde a un thread kernel**
- L'assegnamento di un LWP a un thread utente è gestito in modo esplicito, da programma, dall'applicativo stesso, che deve includere procedure speciali per la gestione di **"upcall"** ...

Processori virtuali assegnati a un processo

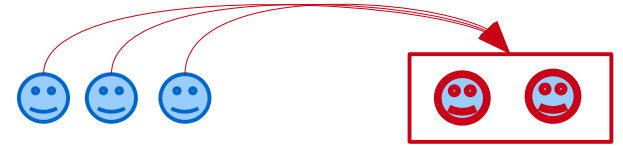


Scheduling



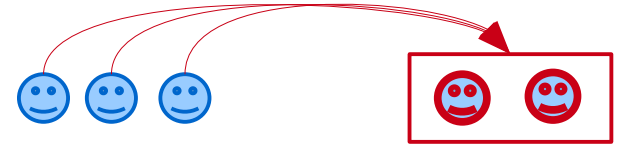
- L'**applicativo utente** esegue il proprio **scheduling dei thread** su un insieme di LWP messi a disposizione dal kernel. Ogni thread dell'applicativo può essere **pronto**, in **esecuzione** o in **attesa**:
 - un thread utente è in esecuzione se ha assegnato un LWP

Scheduling



- L'**applicativo utente** esegue il proprio **scheduling dei thread** su un insieme di LWP messi a disposizione dal kernel. Ogni thread dell'applicativo può essere **pronto**, in **esecuzione** o in **attesa**:
 - **un thread utente è in esecuzione se ha assegnato un LWP**
- Quando un thread esegue una system call bloccante, il SO informa l'applicativo (**upcall**).

Scheduling



- L'**applicativo utente** esegue il proprio **scheduling dei thread** su un insieme di LWP messi a disposizione dal kernel. Ogni thread dell'applicativo può essere **pronto**, in **esecuzione** o in **attesa**:
 - **un thread utente è in esecuzione se ha assegnato un LWP**
- Quando un thread esegue una system call bloccante, il SO informa l'applicativo (**upcall**).
- L'applicativo esegue un **gestore della upcall** che salva lo stato del thread bloccante e rilascia l'LWP su cui era eseguito, che viene riassegnato dall'applicativo stesso a un suo altro thread pronto per l'esecuzione.
- Quando si verificherà l'evento che sveglia il thread sospeso, il SO farà un'altra upcall. L'applicazione segnerà come pronto il thread e lo inserirà nel pool dei thread pronti da assegnare a un LWP.

Lightweight process

NB: non è uno
scheduling della
CPU !!!

lo scheduling della CPU viene
effettuato fra i thread kernel
che vanno in ready queue

thread utente pronti

LWP

thread kernel



un applicativo ha pronti più thread
di LWP a disposizione, deve decidere
a chi assegnarli: due saranno in esecuz.
e uno no

ogni LWP ha associato un
thread kernel



upcall

quando un thread sta per bloccarsi il
SO fa una upcall e l'applicativo riassegna l'LWP



upcall

quando può proseguire il SO fa un'altra upcall e il
thread torna fra quelli pronti

Scheduling della CPU

- in presenza di thread lo **scheduling della CPU** è quindi a due livelli:
 - **process-contention scope** (PCS): è lo scheduling effettuato all'interno di un processo per decidere a quali thread utente vanno assegnati gli LWP a disposizione del processo. I thread kernel associati agli LWP in uso vengono gestiti come i PCB già visti
 - **system-contention scope** (SCS): lo scheduling della CPU viene fatto fra tutti i thread kernel in ready queue (a una granularità più fine rispetto a quanto visto per i PCB), indipendentemente dal processo di appartenenza
- se l'associazione fra thread utente e thread kernel è 1-a-1 allora si ha solo SCS

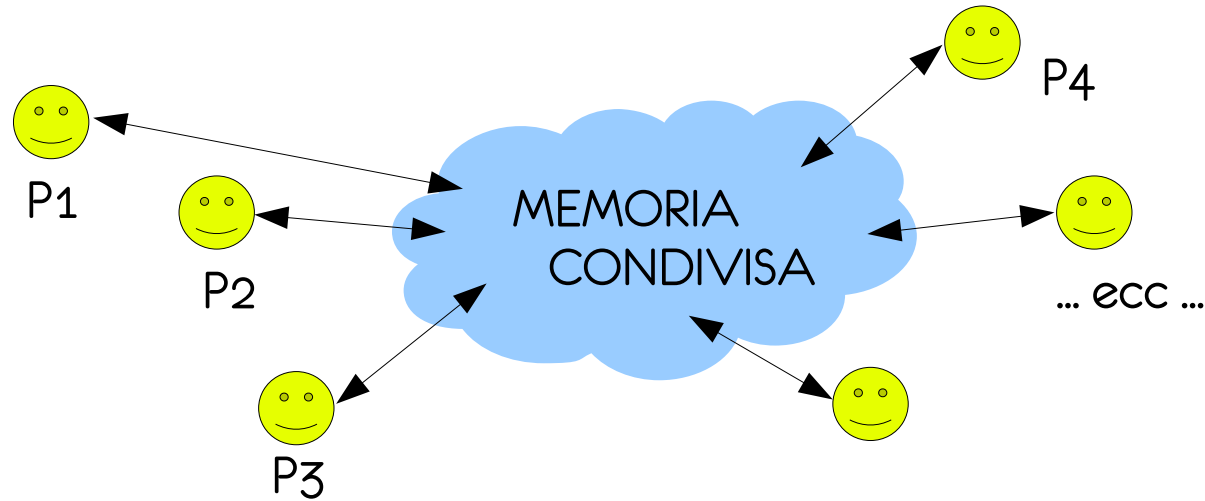
esecuzione concorrente asincrona

capitolo 6 del libro (VII ed.)
e capitolo 5 del libro di Deitel, Deitel e
Choffnes

approfondimento: Dijkstra, E. W. (1971, June).
[Hierarchical ordering of sequential processes.](#)
Acta Informatica 1(2): 115-138.

Chandy, K. M., e Misra, J.
[The drinking philosophers problem](#) (1984), ACM.
Link su moodle

Introduzione 1/2

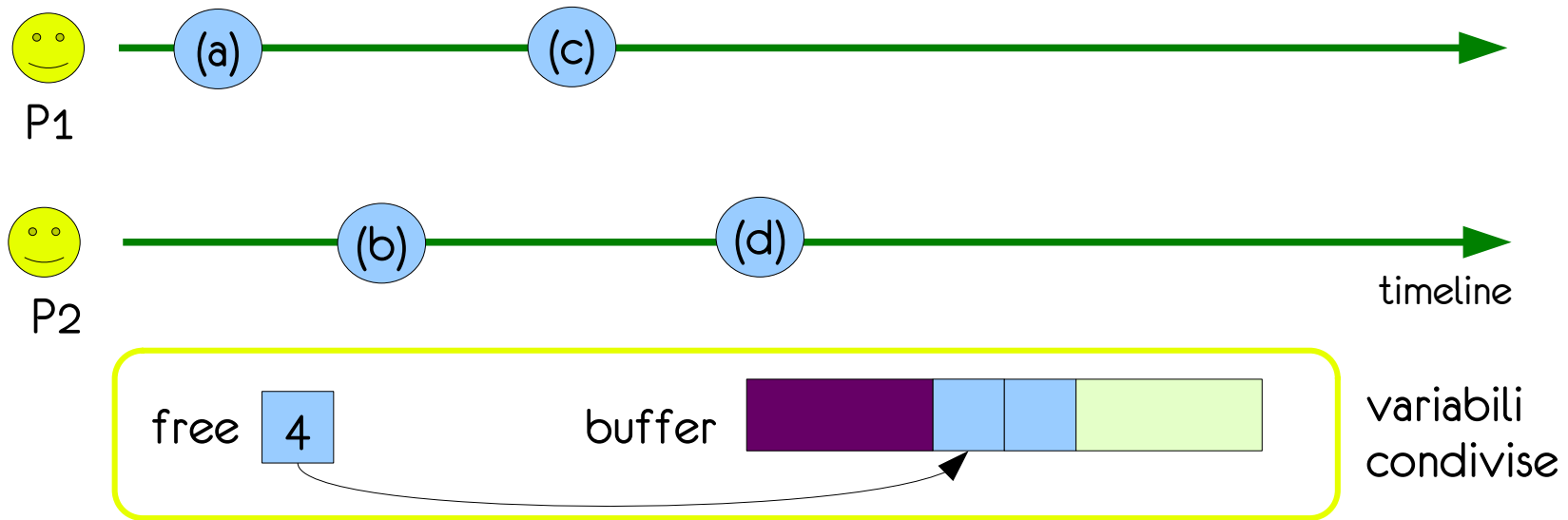


- Consideriamo **due** processi **produttori** che inseriscono dati nella stessa area di memoria condivisa
- L'indice della prima posizione libera è indicato da `free`
- Inserzione:

```
1) buffer[free] = dato  
2) free = (free+1) % D
```

questo codice può creare problemi in un contesto di concorrenza?

Introduzione 2/2



- Non ci sono controlli! L'interleaving potrebbe produrre a una evoluzione dell'esecuzione dove le due istruzioni non sono eseguite in modo atomico, es:

| | | |
|----------------|--|-------------------------|
| (a) P1 esegue: | $\text{buffer}[\text{free}] = \text{dato}$ | $\text{buffer}[4] = 2$ |
| (b) P2 esegue: | $\text{buffer}[\text{free}] = \text{dato}$ | $\text{buffer}[4] = 18$ |
| (c) P1 esegue: | $\text{free} = (\text{free} + 1) \% D$ | $\text{free} = 5$ |
| (d) P2 esegue: | $\text{free} = (\text{free} + 1) \% D$ | $\text{free} = 6$ |



i dati risultano inconsistenti!!!!

Ancora peggio . . .

- Nell'esempio precedente il buffer condiviso era legato a un applicativo utente
- Molte strutture dati condivise (tabella dei file aperti, tabella delle pagine, ecc.) sono strutture usate dal SO!!!
- Un SO con multitasking, in cui possono essere presenti allo stesso tempo diversi processi eseguiti in modalità kernel, può creare inconsistenze nelle tabelle di sistema

Sezione critica 1/4

- Per **sezione critica** si intende una porzione di codice in cui un processo modifica variabili condivise (in generale dati condivisi)
- Requisito: se più processi che condividono una variabile, solo uno di essi per volta può essere nella sua sezione critica (**mutua esclusione**)
- NB: due processi possono essere simultaneamente in sezione critica se tali sezioni si riferiscono a variabili differenti

1) INIZIO SC

2) `buffer[free] = dato`

3) `free = (free+1) % D`

4) FINE SC

Occorre imporre un **meccanismo di controllo** che implementi la mutua esclusione nell'esecuzione di sezioni critiche

Sezione critica 2/4

- Struttura del codice

1) sezione non critica

2) sezione di ingresso

3) `buffer[free] = dato`

4) `free = (free+1) % D`

5) sezione di uscita

6) sezione non critica

corrisponde ad una richiesta, fatta al meccanismo di gestione, di accedere alla sezione critica

sezione critica

avvisa il meccanismo di gestione che è possibile consentire ad un altro processo l'accesso in sezione critica

Sezione critica 3/4

- Una sezione critica:
 - è determinata dalle variabili condivise utilizzate
 - è un segmento di codice che non deve essere eseguito con interleaving di istruzioni di altre sezioni critiche appartenenti alla stessa famiglia (che usano le stesse variabili condivise)
- **Problema:** definire un meccanismo che ne consenta l'utilizzo corretto

Sezione critica 4/4

- Criteri soddisfatti da una soluzione al problema della sezione critica:
 - 1.Mutua esclusione:** Un solo processo per volta può eseguire la propria sezione critica
 - 2.Progresso:** Nessun processo che **non** desideri utilizzare una variabile condivisa può impedirne l'accesso a processi che desiderano utilizzarla. Solo i processi che intendono entrare in sezione critica concorrono a determinare chi entrerà
 - 3.Attesa limitata:** esiste un limite superiore all'attesa di ingresso in sezione critica

Soluzione 1

- Vediamo una prima soluzione al problema, nel caso in cui si abbiano solo **due** processi. La soluzione si appoggia a una variabile **turno**: se $turno == i$ allora P_i può accedere alla sezione critica.
- Codice di P_i :

<sezione non critica>

while (turno != i) do no_op;

▼ sezione di ingresso

<sezione critica>

turno = j;

▼ sezione di uscita

<sezione non critica>

Critica

- **Pro:**

- la soluzione garantisce la mutua esclusione

- **Contro:**

- la soluzione impone una stretta alternanza fra i processi: e se il turno fosse uguale ad i ma il processo P_i non avesse alcuna intenzione di entrare in sezione critica? Allora neppure P_j potrebbe entrarvi, pur volendolo
- si viola quindi la condizione di progresso: P_i è in sezione non critica, non dovrebbe impedire a P_j di entrare in sezione critica
- inoltre il `while (...) no_op` realizza un'attesa attiva: il processo occupa la CPU per non fare nulla, per aspettare (spreco!!)

Idea!

- Per evitare la stretta alternanza:
 - usare **due** variabili anziché una sola
 - ciascuna variabile indica **l'intenzione** di un processo di entrare in sezione critica
 - **problema**: entrambi i processi potrebbero desiderare di entrare in sezione critica in uno stesso momento ...
 - per accedere alla sezione critica devo verificare se la variabile dell'altro processo è impostata

Soluzione 2

- La variabile turno è sostituita da un array, *flag*, di due booleani inizializzati a *false*
- Codice di P_i :

```
flag[i] = true;  
while (flag[j]) do no_op;
```

sezione di ingresso

<sezione critica>

```
flag[i] = false;
```

sezione di uscita

<sezione non critica>

Critica

- **Pro:**

- la soluzione garantisce la mutua esclusione
- la soluzione evita la stretta alternanza fra i due processi

- **Contro:**

- la sezione di ingresso **non è eseguita in maniera atomica**: cosa succede se P_i e P_j desiderano entrambi entrare in sezione critica? `flag[i] == true` e `flag[j] == true`!! I due rimangono bloccati nel while successivo!!
- inoltre il while (...) no_op realizza **un'attesa attiva**: il processo occupa la CPU per non fare nulla, per aspettare (spreco!!)

Idea!

- Per evitare lo stallo:
- posso **usare sia flag che turno**, la variabile turno viene presa in considerazione solo quando entrambi i processi in competizione desiderano entrare in sezione critica ed hanno flag a true (race condition)
- turno, quindi, dirime le dispute che portano allo stallo

Soluzione 3

- La variabile *flag*, di due booleani inizializzati a *false*, è affiancata dalla variabile *turno*, che indica il processo da favorire in caso di competizione

- Codice di P_i :

```
flag[i] = true; turno = j;  
while (flag[j] && turno == j) do no_op;
```

sezione di ingresso

<sezione critica>

flag[i] = false;

sezione di uscita

<sezione non critica>

L'**algoritmo di Peterson** somiglia nell'uso di flag e turno ad un algoritmo precedente, l'algoritmo di Dekker a lungo considerato "la" soluzione al problema però è più semplice

Commento

- Sezione di ingresso:

```
(1) flag[i] = true;  
(2) turno = j;  
(3) while (flag[j] && turno == j) do no_op;
```

- la variabile turno è unica e condivisa, se i due processi eseguono la loro sezione di ingresso in parallelo, uno dei due sarà l'ultimo ad accedere (e settare) il valore di turno. Alcuni possibili interleaving:

- P1-1, P1-2, P2-1, P2-2: turno vale 1
- P1-1, P1-2, P2-1, P1-3, P2-2, P2-3: ... segue ...

(provate a fare diverse simulazioni con diversi interleaving, anche spezzando l'esecuzione della condizione del while)

Simulazione

• P1-1, P1-2, P2-1, P1-3,

```
P1
(1) flag[1] = true;
(2) turno = 2;
(3) while (flag[2] &&
           turno == 2) do no_op;
```

```
P2
(1) flag[2] = true;
(2) turno = 1;
(3) while (flag[1] &&
           turno == 1) do no_op;
```

| | P1 | P2 | |
|------|--|-------------------------------|----------------|
| | | | |
| P1-1 | flag[1]=true | tempo ↓ | |
| P1-2 | turno = 2 | | |
| P2-1 | | | flag[2] = true |
| P1-3 | flag[2] &&turno == 2 vera! | | |
| P2-2 | | turno = 1 | |
| P2-3 | | flag[1] &&turno == 1 vera! | |
| P1-3 | flag[2] &&turno == 2 falsa! Entro in S.C. | | |

NB: turno è una variabile
condivisa dai due processi

il meccanismo funziona perché
ogni processo cede (educata-
mente) il turno all'altro

Critica

- **Pro:**
 - la soluzione garantisce la **mutua esclusione**, evitando la stretta alternanza fra i due processi
 - **progresso**: chi esce dalla sezione critica mette a false il proprio flag, quindi l'altro processo avrà modo di uscire dal proprio while
 - **attesa limitata**: l'attesa di un processo nel proprio while dura quanto la sezione critica dell'altro processo
 - l'algoritmo è estendibile a N processi (**algoritmo del fornaio**)
- **Contro:**
 - il while (...) no_op realizza un'attesa attiva: il processo occupa la CPU per non fare nulla, per aspettare (spreco!!)

Soluzione per N processi

- La prima soluzione semplice e veloce per il caso più generale a N processi venne proposta da Lamport ed è nota come **Algoritmo del Fornaio** (o del panettiere)
- L'idea è di utilizzare un **ticket con numero crescente** per dirimere le race condition (la competizione)

Soluzione per N processi

- Codice di P_i :

```
choosing[i] = true;
ticket[i] = max_ticket + 1;
choosing[i] = false;
for (j = 0; j < N; j++) {
    while (choosing[j]) no_op;
    while (ticket[j] != 0 &&
           ticket[j] < ticket[i] ||
           ticket[j] == ticket[i] && j < i) no_op;
}
```

sezione di ingresso



<sezione critica>

ticket[i] = 0;  sezione di uscita

<sezione non critica>

Soluzione per N processi

• Codice di P_i :

```
choosing[i] = true;
ticket[i] = max_ticket + 1;
choosing[i] = false;
```

sto richiedendo un biglietto
mi viene dato un biglietto
non sto più richiedendo un biglietto

```
for (j = 0; j < N; j++) {
    while (choosing[j]) no_op;
    while
        (ticket[j] != 0 &&
         ticket[j] < ticket[i] ||
         ticket[j] == ticket[i] && j < i) no_op;
}
```

aspetto eventualmente che gli venga rilasciato il biglietto
se il processo j desidera entrare in SC
e ha un biglietto precedente il mio oppure ...

per ogni processo concorrente eseguo un controllo

ha lo stesso numero ma il suo id di processo è precedente il mio, allora cedo il passo

Soluzione per N processi

• Codice di P_i :

```
choosing[i] = true;  
ticket[i] = max_ticket + 1;  
choosing[i] = false;
```

due processi possono avere lo stesso ticket a causa di un'interleaving nell'esecuzione di questa linea di codice

```
for (j = 0; j < N; j++) {  
    while (choosing[j]) no_op;  
  
    while  
        (ticket[j] != 0 &&  
         ticket[j] < ticket[i] ||  
         ticket[j] == ticket[i] &&  
         j < i) no_op;  
}
```

so che prima o poi toccherà a me perché i ticket contengono numeri crescenti, per i quali esiste un solo ordinamento (crescente)

Quando un processo è servito deve richiedere un nuovo ticket per entrare in SC, mettendo il proprio ticket a zero

Attendo ciclando i controlli su ticket[j] finché la condizione non diventa vera e posso controllare il ticket del processo successivo

Critica

- **Pro:**
 - la soluzione garantisce la **mutua esclusione**
 - **progresso e attesa limitata**: sono verificate
- **Contro:**
 - codice e dati sono **complessi**
 - i processi fanno **busy-waiting**, cioè anziché sospendersi attendono il turno tenendo occupata la CPU

Ci sono alternative? L'alternativa è introdurre opportuni supporti hardware

Sincronizzazione HW

Sincronizzazione hardware

- **Soluzione 1:** all'ingresso di una sezione critica, **disabilitare gli interrupt** per impedire il context switch.

Sincronizzazione hardware

- **Soluzione 1:** all'ingresso di una sezione critica, **disabilitare gli interrupt** per impedire il context switch.
- Senza context switch non è possibile la prelazione!
- **Problemi:**
 - interferenze pesanti con lo scheduling della CPU (una sezione critica può essere molto lunga ...)
 - gli interrupt notificano eventi da gestire, non possono essere mantenuti disabilitati a lungo

Sincronizzazione hardware

- **Soluzione 2:** introdurre l'uso di “lock” (lucchetti).
- Per entrare in sezione critica un processo deve avere ottenuto il giusto lock, che rilascia al termine

Sincronizzazione hardware

- **Soluzione 2:** introdurre l'uso di “**lock**” (lucchetti).
- Per entrare in sezione critica un processo deve avere ottenuto il giusto lock, che rilascia al termine
 - a questo fine, molte architetture forniscono istruzioni che consentono di
 - (1) controllare e modificare il valore di una cella di memoria oppure
 - (2) scambiare il contenuto di due celle di memoria in modo atomico
 - in particolare: **TestAndSet** e **Swap**

TestAndSet

- **TestAndSet è atomica**: viene eseguita con interrupt disabilitati, quindi se due processi lanciano ciascuno una TestAndSet, le due esecuzioni verranno sequenzializzate (no interleaving delle istruzioni)

TestAndSet

- **Implementazione:**

```
boolean TestAndSet (boolean *variabile) {  
    boolean valore = *variabile;  
    *variabile = true;  
    return valore;  
}
```

- salva in una variabile locale il valore puntato dal parametro, mette a true il valore puntato dal parametro, restituisce il valore salvato
- Cosa c'è di speciale?
- L'atomicità dell'esecuzione dell'intera routine

Uso di TestAndSet

- realizziamo le sezioni di ingresso e uscita da una sezione critica tramite TestAndSet

Uso di TestAndSet

- occorre dichiarare una variabile globale, accessibile ai processi concorrenti, di tipo booleano
- sia essa chiamata "lock"

Uso di TestAndSet

- realizziamo le sezioni di ingresso e uscita da una sezione critica tramite TestAndSet
- occorre dichiarare una variabile globale, accessibile ai processi concorrenti, di tipo booleano. Sia essa chiamata "lock":

```
while (TestAndSet(&lock));
```

```
<sezione critica>
```

```
lock = false;
```

sezione di ingresso

sezione di uscita

- Perché funziona?

Uso di TestAndSet

- realizziamo le sezioni di ingresso e uscita da una sezione critica tramite TestAndSet
- occorre dichiarare una variabile globale, accessibile ai processi concorrenti, di tipo booleano. Sia essa chiamata "lock":

```
while (TestAndSet(&lock));
```

```
<sezione critica>
```

```
lock = false;
```

sezione di ingresso

sezione di uscita

- TestAndSet restituisce il valore precedente di lock che sarà falso sse nessun altro processo è in una sezione critica controllata tramite lock. Solo in questo caso si esce dal while

Swap

- Alternativa a TestAndSet è **Swap**
- Swap scambia in modo atomico i valori dei suoi due parametri

Swap

- Implementazione:

```
Swap(boolean *a, boolean *b) {  
    boolean temp = *a;  
    *a = *b;  
    *b = temp;  
}
```

- Anche in questo caso l'unica particolarità sta nel tipo di esecuzione della routine, che è atomica

Uso di swap

- Usiamo Swap per realizzare l'accesso in mutua esclusione a una sezione critica
- Oltre a lock (variabile condivisa) utilizzeremo anche una variabile booleana (locale) "chiuso":

Uso di swap

- Implementazione:

```
chiuso = true;  
while (chiuso) Swap(&lock, &chiuso);  
  
<sezione critica>  
  
lock = false;
```

- si esce dal ciclo while solo quando lock era false
- All'uscita da Swap lock risulta automaticamente impostato a true

Critica

- Entrambi i metodi visti garantiscono la mutua esclusione ...

Critica

- Entrambi i metodi visti garantiscono la mutua esclusione ...
- ... ma non l'attesa limitata!
- non c'è garanzia che un processo che vorrebbe eseguire TestAndSet oppure Swap non venga sempre prevaricato da altri
- è possibile usare TestandSet per realizzare una soluzione che non presenta il problema dell'attesa illimitata?

Attesa limitata

- Sì, l'algoritmo è complicatissimo!!! (non da studiare)

sezione di ingresso

```
attesa[i] = true;
chiave = true;
while (attesa[i] && chiave)
    chiave = TestAndSet(&lock);
attesa[i] = false
```

sezione di uscita

```
j = (i+1) % n;
while ((j != i) && !attesa[j])
    j = (j+1) % n;
if (j == i) lock = false;
else attesa[j] = false
```

- in ogni caso persiste il problema del **busy waiting** ...

Semafori

Semafori

- Strumenti di sincronizzazione introdotti da Dijkstra nel 1965 per minimizzare il busy-waiting e per semplificare la vita ai programmatori
- **semaforo** = variabile a cui (dopo l'inizializzazione) si può accedere solo tramite due operazioni atomiche:
 - **P** (*proberen*, verificare in olandese) e
 - **V** (*verhogen*, incrementare),
- Sono anche note come wait e signal, down e up

Semafori

- **semaforo** = variabile a cui si può accedere (dopo l'inizializzazione) solo tramite due operazioni atomiche note come P (*proberen*, verificare in olandese) e V (*verhogen*, incrementare), anche note come wait e signal
- Possiamo immaginare lo stato del semaforo come un campo di tipo intero
- in pseudocodice:

```
P (S) {  
    while (S <=0) no_op;  
    S--;  
}
```

definizione
di P e V

```
V (S) {  
    S++;  
}
```

La P non realizza
un'attesa attiva?

mutex inizializzato
al valore 1

```
P(mutex);  
  
<sezione critica>  
  
V(mutex);
```

utilizzo di un semaforo mutex,
P e V per controllare una
sezione critica

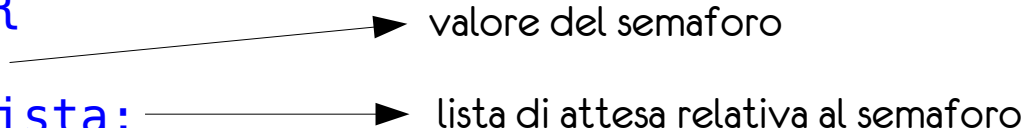
Spinlock e sospensione

- Il tipo di attesa comportato dai semafori implementati come visto è detto **spinlock**: lo spinlock fa un'attesa attiva
- sono possibili altre implementazioni che evitano lo spinlock. Richiedono strutture di appoggio
- ogni semaforo ha associata una **lista di PCB**: quelli dei processi sospesi su quel semaforo
- quando un processo si sospende su di un semaforo, lo scheduler della CPU la assegna a un altro processo
- quando il valore del semaforo viene alzato uno dei processi nella sua coda di attesa viene scelto e fatto passare dallo stato waiting allo stato ready

Semafori con code 1/2

- Vediamo una possibile implementazione del tipo di dato semaforo, nel caso questo includa una coda di attesa, e delle procedure P e V

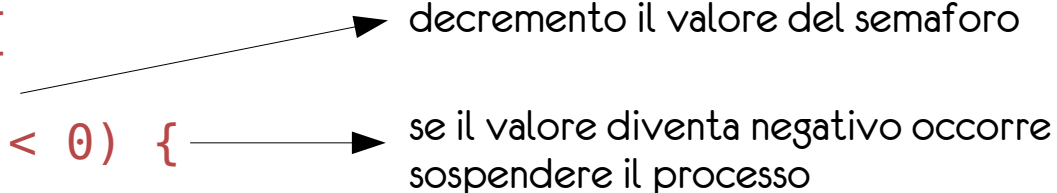
```
typedef struct {  
    int valore;  
    processo *lista;  
} semaforo;
```



valore del semaforo

lista di attesa relativa al semaforo

```
P (semaforo *s) {  
    S->valore - -;  
    if (S->valore < 0) {
```

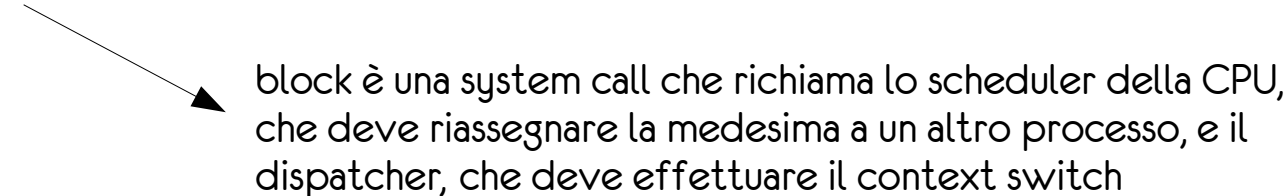


decremento il valore del semaforo

se il valore diventa negativo occorre sospendere il processo

```
        <aggiungi il PCB di questo processo a S->lista>  
        block();  
    }
```

```
}
```



block è una system call che richiama lo scheduler della CPU, che deve riassegnare la medesima a un altro processo, e il dispatcher, che deve effettuare il context switch

Semafori con code 2/2

- Vediamo una possibile implementazione del tipo di dato semaforo, nel caso questo includa una coda di attesa, e delle procedure P e V

```
V (semaforo *S) {  
    S->valore++;  
    if (S->valore < 0) {  
        <scegli un PCB P da S->lista>  
        wakeup(P);  
    }  
}
```

incremento il valore del semaforo

se il valore è negativo, allora ci sono processi da risvegliare

wakeup è una system call che rende il processo P ready

NB: poiché come prima operazione P decrementa sempre il valore del semaforo, quando questo ha valore negativo, il suo valore assoluto indica il numero dei processi in attesa

Inizializzazione e uso

- I semafori di mutua esclusione sono inizializzati a 1 e possono valere al più 1:
 - 1 = risorsa disponibile,
 - 0 (o valore negativo) = risorsa occupata
- I semafori che possono assumere valori > 1 sono detti semafori contatori. Il numero N , valore del semaforo, indica una quantità di risorse disponibili
 - in certe implementazioni un processo può richiedere un numero $M \geq 1$ di risorse
 - il codice di P e V che abbiamo visto non consente di realizzare questo tipo di richieste
 - l'implementazione Unix (che studieremo per la parte di lab) invece lo consente

Tipi di sincronizzazione

- I semafori sono strumenti che consentono di realizzare molti tipi di sincronizzazione diversa: dipende da come si usano
- **Mutua esclusione**: tutti i processi coinvolti parentesizzano le loro sezioni critiche con `P(mutex) ... V(mutex)`, dove `mutex` è un semaforo di mutua esclusione
- **Accesso limitato**: tutti i processi coinvolti parentesizzano le loro sezioni critiche con `P(nris) ... V(nris)`, dove `nris` è un semaforo contatore. Un numero limitato di processi potrà eseguire in parallelo una certa sezione di codice
- **Ordinamento**: si può controllare l'ordine di esecuzione di due processi, ad esempio sia `sem` un semaforo inizializzato a 0:

| | |
|---------------------|---------------------|
| P1: | P2: |
| <code>P(sem)</code> | <code>codice</code> |
| <code>codice</code> | <code>V(sem)</code> |

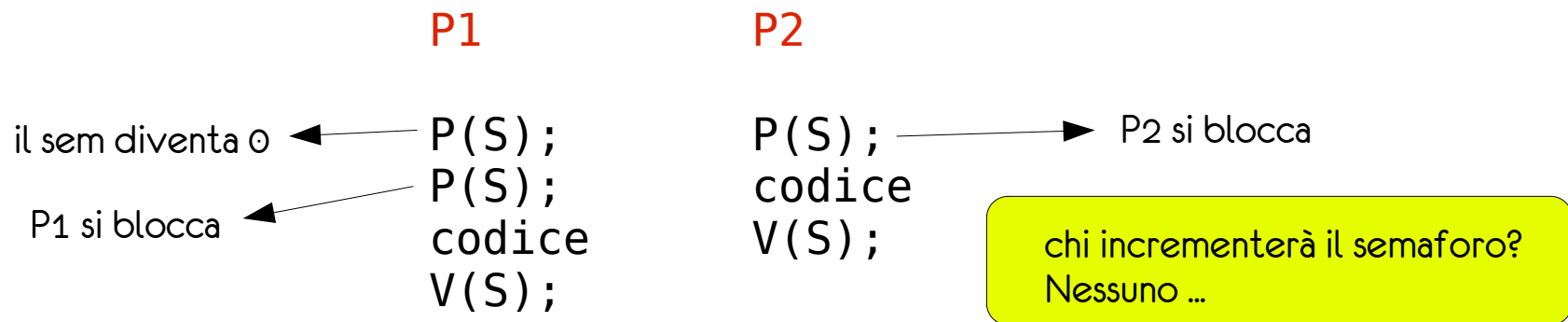
in quale ordine vengono eseguiti P1 e P2?

Operazioni atomiche

- Le operazioni sui semafori devono essere eseguite in modo atomico: sono sezioni critiche perché i semafori sono variabili condivise
- su sistemi monoprocesso, atomicità ottenuta disabilitando gli interrupt: non è un problema perché i tempi di esecuzione di P e V sono molto brevi
- su sistemi multiprocesso, invece, si utilizzano prevalentemente spinlock perché disabilitare le interruzioni di tutti i processori crea cali di prestazione
- => busy-waiting ridotta ma non eliminata

Attenzione

- Il cattivo uso dei semafori da parte di un programmatore può creare **deadlock** (stallo), es. sia S un semaforo che inizialmente vale 1:



Attenzione

- Situazioni più subdole in cui si può generare deadlock sono causate dall'utilizzo di più semafori

Attenzione

- Situazioni più subdole in cui si può generare deadlock sono causate dall'utilizzo di più semafori
- Esempio: siano S e T due semafori inizializzati ad 1 e siano P1 e P2 due processi che eseguono le seguenti operazioni:

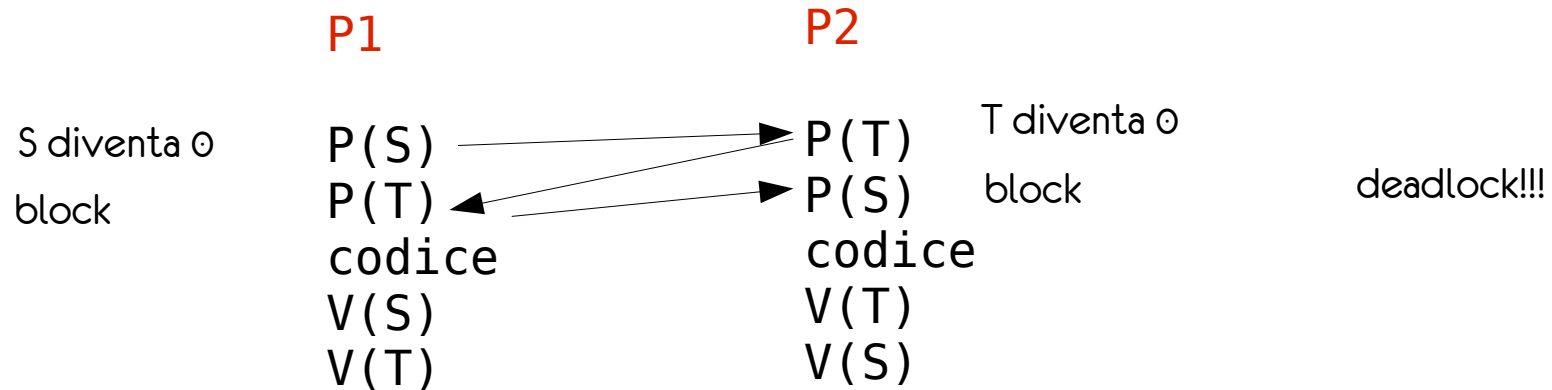
P1

P(S)
P(T)
codice
V(S)
V(T)

P2

P(T)
P(S)
codice
V(T)
V(S)

Attenzione



- Se P1 esegue P(S) poi P2 esegue P(T), quindi P1 esegue P(T), bloccandosi, e P2 esegue P(S), bloccandosi: si ha un deadlock
- NB: il deadlock non si ha ad ogni esecuzione, **dipende dall'interleaving delle istruzioni!!!** Es. se P1 esegue P(S) e subito dopo P(T) entra in sezione critica e poi alza il valore dei due semafori, lasciando entrare P2

Problemi classici di sincronizzazione

- Produttori – consumatori con buffer limitato (in parte già visto)
- Lettori – scrittori
- Cinque filosofi

Problemi classici di sincronizzazione

- **Produttori – consumatori con buffer limitato** (in parte già visto): si usano 3 semafori:
 - **mutex**: mutua esclusione
 - **libere**: numero di caselle libere del buffer
 - **occupate**: numero di caselle occupate del buffer
- Lettori – scrittori
- Cinque filosofi

produttori - consumatori

Produttore

```
forever {  
    <produci un nuovo elemento>  
  
    P(libere);  
    P(mutex);  
  
    <inserisci nuovo elemento>  
  
    V(mutex);  
    V(occupate);  
}
```

Consumatore

```
forever {  
  
    P(occupate);  
    P(mutex);  
  
    <estrai elemento>  
  
    V(mutex);  
    V(libere);  
  
    <consuma elemento>  
}
```

Inizialmente **mutex** vale 1, **libere** vale N (capienza del buffer) e **occupate** vale 0

produttore

Produttore

```
forever {
```

```
    <produci un nuovo elemento>
```

```
    P(libere);
```

```
    P(mutex);
```

```
    <inserisci nuovo elemento>
```

```
    V(mutex);
```

```
    V(occupate);
```

```
}
```

se il buffer è pieno libere vale 0 quindi il produttore si sospende e rimane sospeso finché non c'è qualche cella a disposizione
NB: la P decrementa libere

l'inserzione vera e propria va effettuata in mutua esclusione per mantenere la consistenza dei dati nel buffer, che è un'area di memoria condivisa

alla fine incremento il numero delle celle occupate, attivando eventualmente un consumatore

consumatore

Consumatore

```
forever {
```

```
    P(occupate);
```

```
    P(mutex);
```

```
    <estrai elemento>
```

```
    V(mutex);
```

```
    V(libere);
```

```
    <consumo elemento>
```

```
}
```

rimane fermo finché non c'è nulla da consumare, quando occupate viene incrementato da un produttore lo decrementa, indicando che sta per consumare un oggetto, quindi procede

come prima l'accesso al buffer va fatto in mutua esclusione per evitare che si produca un'inconsistenza nei dati

quando si estrae un elemento si libera una cella. Il processo segnala questo evento incrementando il semaforo "libere". Se il buffer era pieno e un produttore era in attesa, ritorna ready

l'elaborazione dell'oggetto estratto, la sua consumazione, non viene fatta in ME, per minimizzare l'esecuzione in sezione critica. Infatti non richiede ulteriori accessi al buffer

Problemi classici di sincronizzazione

- Produttori - consumatori con buffer limitato
- **Lettori - scrittori**: dei processi lettori e scrittori usano una stessa area di memoria. Gli scrittori modificano la risorsa, quindi accedono in ME con tutti. I lettori non modificano la risorsa, quindi possono accedere alla memoria condivisa in parallelo.
 - Si utilizzano:
 - **scrittura**: semaforo di ME
 - una variabile intera condivisa **numlettori** che conta quanti processi stanno leggendo in questo momento
 - **mutex**: semaforo di ME per l'accesso a **numlettori**
- Cinque filosofi

Problemi classici di sincronizzazione

- Produttori - consumatori con buffer limitato
- **Lettori - scrittori**: dei processi lettori e scrittori usano una stessa area di memoria. Gli scrittori modificano la risorsa, quindi accedono in ME con tutti. I lettori non modificano la risorsa, quindi possono accedere alla memoria condivisa in parallelo.
 - Si utilizzano:
 - **scrittura**: semaforo di ME
 - una variabile intera condivisa **numlettori** che conta quanti processi stanno leggendo in questo momento
 - **mutex**: semaforo di ME per l'accesso a **numlettori**
- Cinque filosofi

lettori - scrittori

Lettore

```
forever {  
    P(mutex);  
    numlettori++;  
    if (numlettori == 1)  
        P(scrittori);  
    V(mutex);  
  
    <legge>  
  
    P(mutex);  
    numlettori--;  
    if (numlettori == 0)  
        V(scrittura);  
    V(mutex);  
}
```

Scrittore

```
forever {  
    P(scrittura);  
  
    <scrive>  
  
    V(scrittura);  
}
```

Inizialmente **mutex** vale 1, **scrittura** vale 1 e **numlettori** vale 0

scrittore

Scrittore

```
forever {
```

```
    [ P(scrittura);  
      <scrive>  
      V(scrittura); ]
```

uno scrittore accede all'area condivisa in ME con chiunque altro (lettori o scrittori) perché modifica la risorsa. Scrittura vale inizialmente 1

```
}
```

lettore

Lettore

```
forever {
```

```
[A] { P(mutex);  
      numlettori++;  
      if (numlettori == 1)  
        P(scrittori);  
      V(mutex);
```

```
<legge>
```

```
[B] { P(mutex);  
      numlettori--;  
      if (numlettori == 0)  
        V(scrittura);  
      V(mutex);
```

```
}
```

[B] decrementa numlettori ed eventualmente incrementa il semaforo scrittura

un lettore usa due classi di sezione critica: una relativa all'area di memoria da cui legge e una relativa alla variabile condivisa numlettori. Mutex controlla l'accesso a quest'ultima

[A] e [B] parentesizzano la sezione critica relativa all'area da cui si legge. Implementano un controllo per cui un lettore può accedervi a patto che nessuno scrittore stia operando.

[A]:
se `numlettori > 1` allora qualcun altro ha già fatto in modo tale che nessuno scrittore sia attivo. Il lettore può procedere nella lettura.

se `numlettori == 1` sono l'unico lettore, tramite il semaforo scrittura controllo ed eventualmente attendo che non ci siano scrittori attivi. Poiché uso una P, quando riesco a passare blocco l'accesso ad eventuali scrittori

Problemi classici di sincronizzazione

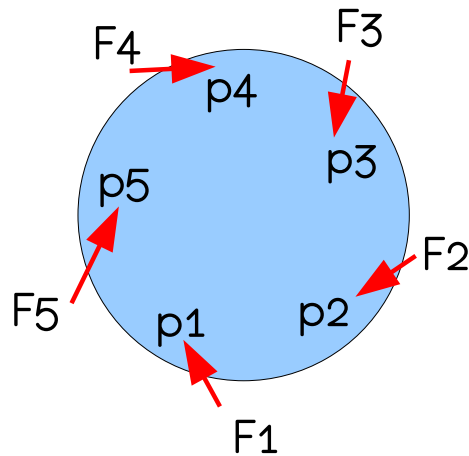
- Produttori - consumatori con buffer limitato
- Lettori - scrittori
- **Cinque filosofi**: 5 filosofi passano il tempo seduti intorno a un tavolo pensando e mangiando a fasi alterne. Per mangiare un filosofo ha bisogno di due posate ma ci sono solo cinque posate in tutto

è un problema interessante perché rappresenta un'ampia categoria di situazioni pratiche in cui diversi processi hanno bisogno di più risorse e bisogna evitare situazioni di deadlock e di starvation (es. quando riesco ad avere solo una parte delle risorse necessarie)

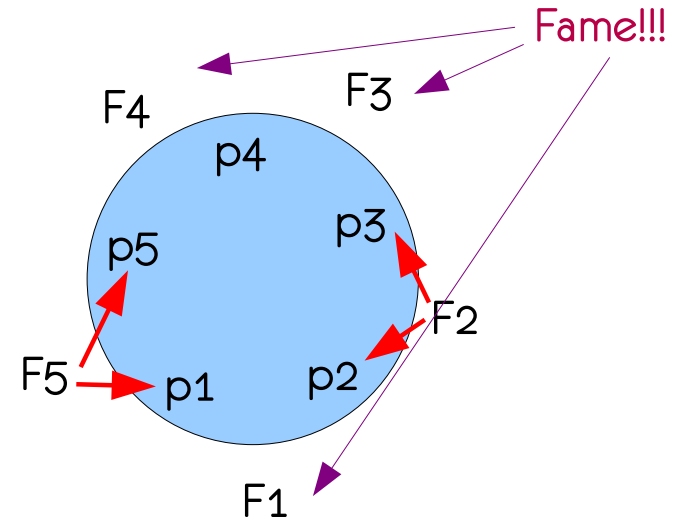


```
forever {  
  think  
  eat  
}
```

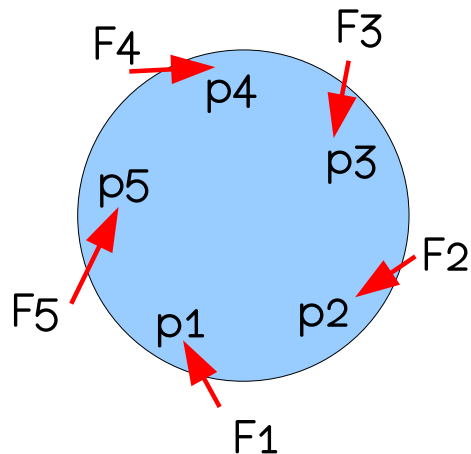
deadlock e altri guai



Deadlock: tutti hanno preso la posata di sinistra ma nessuno può prendere quella di destra



Starvation: due filosofi si accaparrano sempre le posate necessarie impedendo agli altri di mangiare



Livelock: tutti prendono la posata di sinistra ma nessuno può prendere quella di destra, quindi tutti rilasciano la posata di sinistra, poi ricominciano da capo. I processi non sono propriamente bloccati ma non c'è progresso.

una soluzione

Filosofo F_i

```
forever {  
    P(posata[i]);  
    P(posata[i+1]%5);  
  
    <mangia>  
  
    V(posata[i]);  
    V(posata[i+1]%5);  
  
    <pensa>  
}
```

si usano 5 semafori, uno per ciascuna posata;
tutti richiedono la posata di sinistra e poi
quella di destra.

Deadlock? Sì

Starvation? Sì

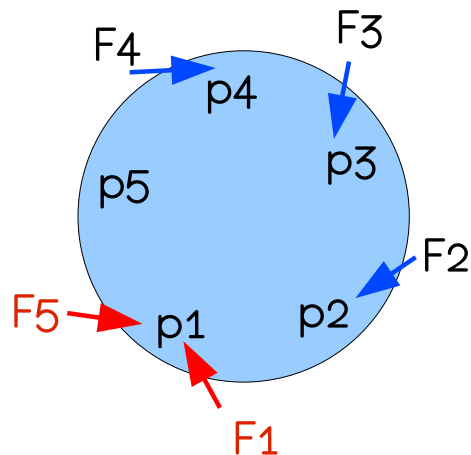
Prevenire il deadlock

Dijkstra propone una soluzione che consente di evitare il deadlock. A questo fine occorre rompere la simmetria nell'accesso alle posate. Introduzione di una semplice regola:

ogni filosofo deve prendere per prima la posata con indice minore

F1 prenderà quindi p1 prima di p2, ..., F4 prenderà p4 prima di p5.

Apparentemente al contrario F5 prenderà prima p1 e poi p5!!!



F1 ed F5 concorrono all'uso di p1:

se vince F1, F5 aspetterà che p1 si liberi prima di richiedere p5, lasciando a F4 la possibilità di usarla

se vince F5, F1 aspetterà che p1 si liberi, lasciando a F5 la possibilità di mangiare richiedendo anche p5

NB: la starvation persiste

Chandy-Misra

Chandy-Misra propongono una soluzione “equa”, nel senso che quando due processi sono in competizione l'algoritmo non favorisce sempre lo stesso (causando starvation). Viene introdotto un concetto di precedenza fra processi, che però cambia nel tempo, è dinamica. Il dinamismo è ottenuto associando un concetto di stato alla risorsa.



1. Ogni forchetta può essere “sporca” o “pulita”, inizialmente sono tutte “sporche” e ogni filosofo (quello con id più basso fra i due vicini) ne ha una..
2. Quando un filosofo vuole mangiare invia ai suoi vicini dei messaggi per ottenere le forchette che gli mancano.
3. Quando un filosofo, che ha una forchetta, riceve una richiesta: se sta pensando, la cede altrimenti se la forchetta è pulita ne mantiene il possesso, se è sporca la cede. Quando passa una forchetta ne pone lo stato a “pulita”.
4. Dopo aver mangiato, tutte le forchette di un filosofo diventano “sporche”. Se risultano richieste pendenti per qualche forchetta, il filosofo la pulisce e la passa.

Uso errato dei semafori

- È molto facile introdurre errori involontari in programmi che fanno uso di semafori. consideriamo la semplice mutua esclusione:
 - `V(mutex) <sez. cr.> P(mutex)`
 - `P(mutex) <sez. cr.> P(mutex)`
 - `<sez. critica>`
- sono tutti esempi di programmi sbagliati. Basta un solo processo sbagliato per creare deadlock!!!
- una soluzione è incapsulare risorse di basso livello, come i semafori, in tipi di dati astratti, come i **monitor**, offerti poi come nuovi costrutti del linguaggio

Monitor

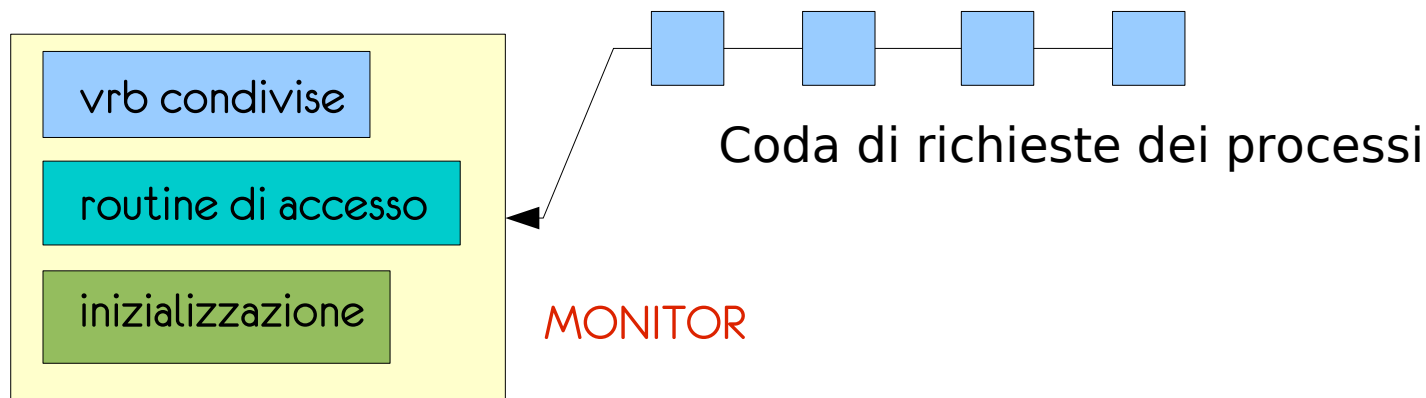
- **Monitor** (Dijkstra, Brinch Hansen) è un **costrutto di sincronizzazione** contenente i dati e le operazioni necessarie per allocare una risorsa condivisa usabile in modo seriale
- È un **Abstract Data Type**:
incapsula dei dati mettendo a disposizione le operazioni necessarie per manipolarli
- Le variabili di un monitor sono **condivise** dai processi che usano quel monitor
- Un solo processo per volta può essere attivo all'interno di un certo monitor (**Mutua Esclusione**)

Monitor

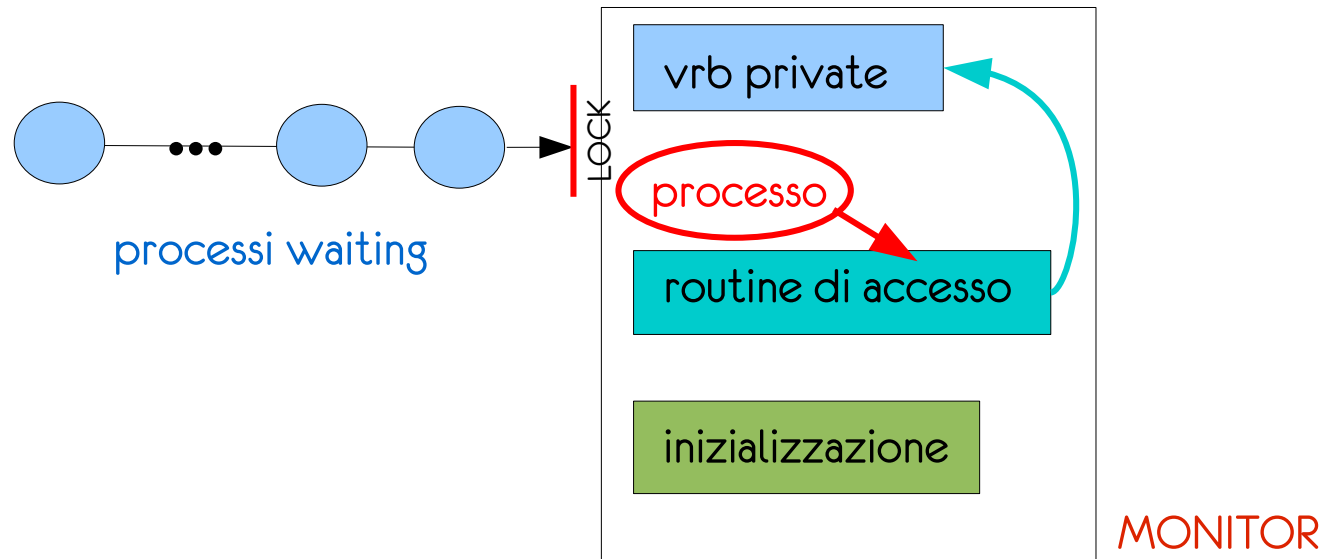
- Brinch-Hansen: "You can imagine the (monitor) calls as a **queue of messages being served one at a time**. The monitor will receive a message and try to carry out the request as defined by the procedure and its input parameters. If the request can immediately be granted the monitor will return parameters . . . and allow the calling process to continue. However, if the request cannot be granted, the monitor will prevent the calling process from continuing, and enter a reference to this transaction in a queue local to itself. **This enables the monitor [...] to inspect the queue and decide which interaction should be completed now.** From the point of view of a process a monitor call will look like a **procedure call**. The **calling process will be delayed** until the monitor consults its request. The monitor then has a set of scheduling queues which are completely local to it, and therefore protected against user processes."
- **Fonte:** P. Brinch Hansen, Monitors and Concurrent Pascal: A personal history. 2nd ACM Conference on the History of Programming Languages, Cambridge, MA, April 1993. In c 1993, Association for Computing Machinery, Inc.

Monitor

- Per accedere alle variabili condivise un processo deve eseguire una routine di accesso al monitor (possono essercene diverse)
- un solo processo per volta può essere attivo all'interno di un certo monitor (Mutua Esclusione)
- es. i monitor sono usati da Java per realizzare la ME dei thread (prog. III)



Monitor



un processo che riesce ad eseguire una routine di accesso al monitor acquisisce un **lock** sul monitor. Tutti gli altri processi che cercano di eseguire routine di accesso vengono sospesi in una coda di attesa esterna al monitor. Quando il lock viene rilasciato uno dei processi waiting viene riattivato

Monitor

- oltre che la ME, i monitor consentono anche di effettuare alcuni tipi di **sincronizzazione fra i processi**
- es. monitor per produttori-consumatori: un consumatore che ha avuto accesso al monitor si accorge che non c'è nulla da consumare ... non c'è un modo per far sì che i consumatori possano entrare solo se è il caso?
- vengono introdotte variabili di tipo **condition**, su cui si possono eseguire solo le operazioni:
 - **wait(x)**: l'esecutore viene sospeso se la condition x è falsa
 - **signal(x)**: se qualche processo è sospeso sulla condition x, uno viene scelto e risvegliato, altrimenti non ha effetto
 - NB: signal è diversa dalla V dei semafori: la V incrementa sempre il valore del semaforo, ha sempre un effetto!

Monitor: esempio

monitor per un allocatore di risorse

```
boolean inUso = false;
```

```
condition disponibile;
```

```
monitorEntry void prendiRisorsa() {  
    if (inUso) wait(disponibile);  
    inUso = true;  
}
```

```
monitorEntry void rilasciaRisorsa() {  
    inUso = false;  
    signal(disponibile);  
}
```

se una qualche risorsa di interesse risulta usata da qualcun altro, allora mi sospendo sulla condition "disponibile". Quando mi risveglio continuo dalla linea di codice successiva: setto a true inUso.

Memento: il monitor è eseguito in ME quindi tutte le monitorEntry sono atomiche, a meno di sospensioni volontarie

quando rilascio una risorsa, faccio una notifica sulla condition "disponibile". Se qualcuno era in attesa si risveglia, altrimenti la notifica viene dimenticata

Signal e M.E.

Quando un processo (thread) esegue signal rischiamo di avere due processi (thread) attivi nel monitor!! Quello che ha eseguito signal e quello risvegliato



SOLUZIONI

Segnalare e Attendere

- P attende
- Q riprende ed esegue

Segnalare e Proseguire

- P continua
- Q aspetta che P finisca

5 filosofi realizzati coi monitor 1/2

Monitor 5filosofi {

Può essere “mangia” solo se i due vicini
non sono settati a “mangia”

enum { pensa, affamato, mangia } stato[5];
condition aspetta[5];

Prende (int i) {
 stato[i] = affamato;
 verifica(i);
 if (stato[i] != mangia) **aspetta[i].wait()**;
}

Consente ai filosofi di sospendersi
quando non possono mangiare

Posa (int i) {
 stato[i] = pensa;
 verifica((i+4) % 5);
 verifica((i+1) % 5);
}

...

5 filosofi realizzati coi monitor 2/2

...

```
Verifica (int i) {  
    if (stato[(i+4)%5] != mangia) &&  
        stato[i] == affamato &&  
        (stato[(i+1)%5] != mangia)  
    {  
        stato[i] = mangia;  
        aspetta[i].signal();  
    }  
}
```

```
Codice di inizializzazione {  
    for (int i=0; i<5; i++)  
        stato[i] = pensa;  
}
```

```
}
```

M.E. e transazioni

- In talune circostanze applicative (e.g. basi di dati, contabilizzazione) il concetto di esecuzione in ME non è sufficiente. Si vorrebbe realizzare un'idea di **atomicità più forte**, per cui o si riesce ad eseguire un intero insieme di istruzioni con successo oppure non se ne deve eseguire nessuna. Devono essere eseguite come una sola operazione non interrompibile.
- Esempio:
 - acquisisci stampante,
 - contabilizza pagine da stampare,
 - stampa,
 - rilascia stampante

in questo caso vorrei poter effettuare il **rollback** delle istruzioni eseguite, disfarne gli effetti, tornare allo stato precedente: *cancella pagine stampate, sottrai loro numero da totale*

e se la carta finisce prima che io abbia potuto stampare tutte le pagine desiderate?

Mi verranno fatte pagare anche quelle non stampate?

Cosa me ne faccio delle sole prime 2 (per es.) pagine?

Es. journaling del file system

Esempio: transazione commerciale

CC_cl == 100 // conto corrente del cliente
CC_vend == 100 // c.c. del venditore
Prezzo == 50 // prezzo oggetto acquistato

READ CC_cl

READ CC_vend

WRITE CC_cl CC_cl - 50

WRITE CC_vend CC_vend + 50

Esempio: transazione commerciale

CC_cl == 100 // conto corrente del cliente
CC_vend == 100 // c.c. del venditore
Prezzo == 50 // prezzo oggetto acquistato

READ CC_cl

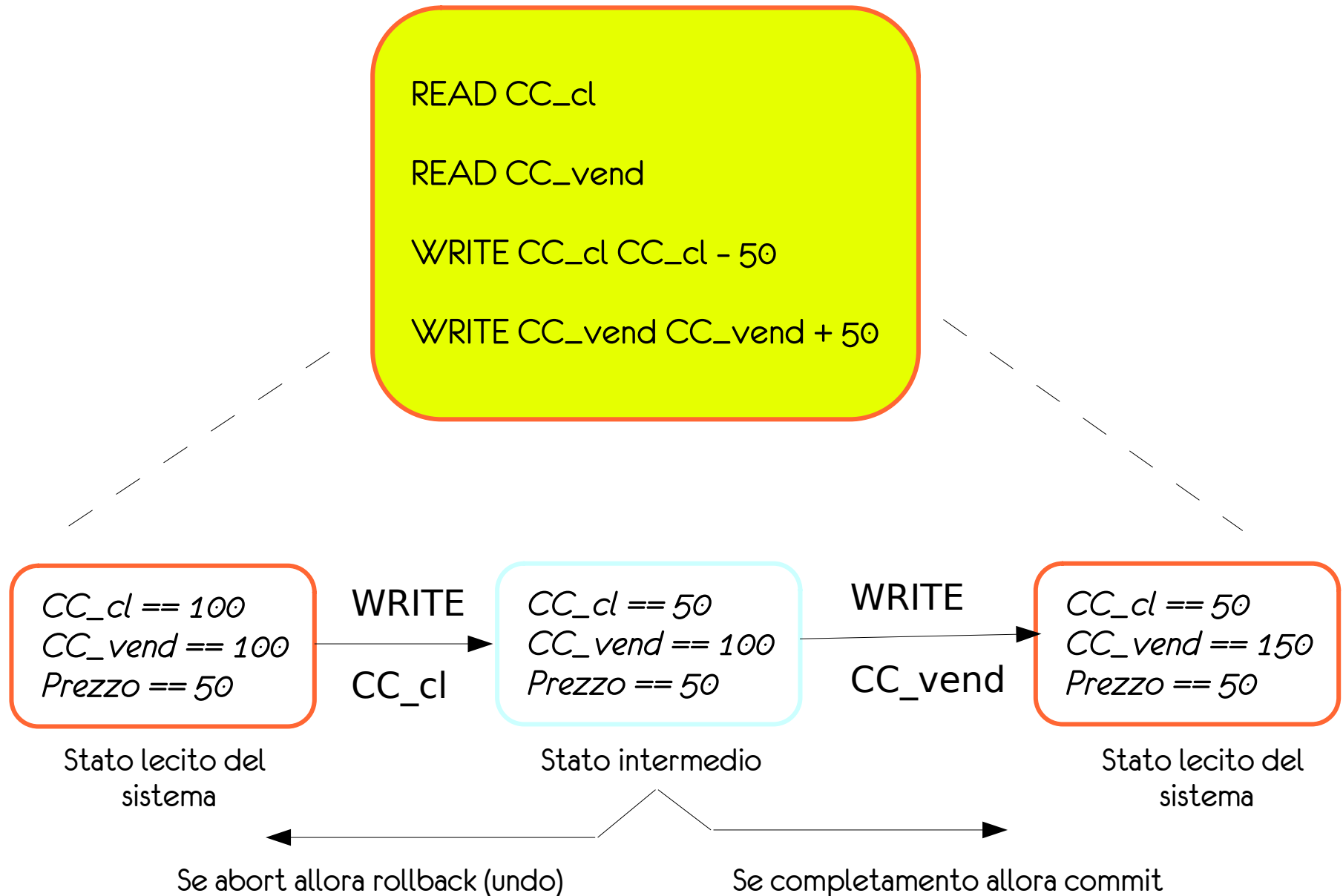
READ CC_vend

WRITE CC_cl CC_cl - 50

WRITE CC_vend CC_vend + 50

Se l'esecuzione si interrompesse a questo punto avremmo uno stato inconsistente: il CC del cliente è stato decrementato ma la cifra non è stata aggiunta al CC del venditore

Esempio: transazione commerciale



Transazione

- Transazione concetto che indica un insieme di istruzioni che esegue una singola funzione logica
- Ovvero una sequenza di **read** e **write** che si conclude con un **commit** o con un **abort**:
 - **commit**: la transazione si è conclusa con successo
 - **abort**: la transazione è fallita. Potrebbe avere alterato dei dati! Bisogna disfarne gli effetti (*rollback* o *compensazione*)

Transazione

- Come si fa a disfare una transazione abortita?
- Ovvero come si fa a garantire l'atomicità di una transazione?
- Dipende dai dispositivi di memoria usati per mantenere i dati elaborati dalla medesima:
 - **memorie volatili** (es. cache e registri): staccando la corrente si cancellano
 - **non volatili** (es. dischi, EEPROM): sono persistenti ma non danno garanzie di eternità
 - **stabili**: sono supporti di memorizzazione a cui si aggiungono politiche e strumenti di duplicazione che rendono “eterni” i dati memorizzati

Memoria volatile

- Vedremo solo il caso di transazioni abortite per **cancellazione di memoria volatile** (es. crash di sistema)
- Per ripristinare uno stato precedente occorre tener traccia delle operazioni eseguite => memorizzare in un file (mantenuto in memoria stabile e detto **logfile** o **log**) cosa viene fatto
- **Contenuto del log:**

- per ogni transazione T si registra il suo inizio:

<T, start>

- si registra una sequenza di tuple, ciascuna relativa ad una operazione di **write** prossima da eseguire (**write-ahead logging**), siffatte:

<ID transazione, ID dato modificato, valore precedente, nuovo valore>

- si registra il successo della transazione T:

<T, commit>

Ripristino/abort

- A seguito di un crash di sistema, il S0 controlla il log e per ogni transazione T registrata nel log:
 - se a $\langle T, \text{start} \rangle$ non corrisponde un $\langle T, \text{commit} \rangle$ il S0 esegue l'operazione $\text{undo}(T)$ che ripristina tutti i valori modificati dalla transazione eseguita in modo parziale
 - se a $\langle T, \text{start} \rangle$ corrisponde $\langle T, \text{commit} \rangle$ il S0 verifica che le modifiche registrate siano state effettivamente eseguite. È possibile che al crash alcune modifiche registrate nel log non fossero ancora state apportate ai dati veri e propri, in questo caso il S0 esegue un $\text{redo}(T)$, attua le modifiche registrate.

Ripristino/abort

- A seguito di un crash di sistema, il S0 controlla il log e per ogni transazione T registrata nel log:
 - se a $\langle T, \text{start} \rangle$ non corrisponde un $\langle T, \text{commit} \rangle$ il S0 esegue l'operazione $\text{undo}(T)$ che ripristina tutti i valori modificati dalla transazione eseguita in modo parziale
 - se a $\langle T, \text{start} \rangle$ corrisponde $\langle T, \text{commit} \rangle$ il S0 verifica che le modifiche registrate siano state effettivamente eseguite. È possibile che al crash alcune modifiche registrate nel log non fossero ancora state apportate ai dati veri e propri, in questo caso il S0 esegue un $\text{redo}(T)$, attua le modifiche registrate.
- Questo genere di meccanismo è usato dai sistemi di **journaling** proposti dai moderni S0 per verificare velocemente la consistenza del file system dopo un crash
- La registrazione dei dati introduce un overhead di lavoro, rende un po' meno efficiente l'esecuzione, però fornisce garanzie irrinunciabili in certe applicazioni

Tutto il logfile?

- Una domanda lecita è se ad ogni crash sia proprio necessario scorrere l'intero logfile, prendendo in considerazione anche transazioni ormai felicemente concluse da tempo ...
- Minore è il numero di transazioni considerate maggiore l'efficienza del ripristino!

Tutto il logfile?

- Una domanda lecita è se ad ogni crash sia proprio necessario scorrere l'intero logfile, prendendo in considerazione anche transazioni ormai felicemente concluse da tempo ...
- Minore è il numero di transazioni considerate maggiore l'efficienza del ripristino!
- **Checkpoint:**
 - si introduce una entry `<checkpoint>` nel logfile dopo quelle transazioni tali che:
 - 1) tutti i record del logfile relativi alla transazione sono stati riportati in memoria stabile
 - 2) tutte le operazioni di scrittura registrate nel logfile sono state applicate con successo

Uso dei checkpoint

- Quando si ha un crash di sistema, si scorre a ritroso il logfile fino a identificare la prima occorrenza di **<checkpoint>**
- Tutte le operazioni che precedono questa etichetta possono essere ignorate perché per definizione di checkpoint tutte le modifiche sono già in memoria stabile
- Le operazioni di undo e redo vengono solo applicate alle transazioni successive al checkpoint

logfile

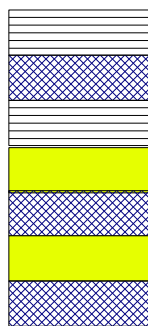
```
<T, start>  
<...>  
<T, end>  
<checkpoint>  
<S, start>  
<...>
```

Posso ignorare tutte queste entry
del logfile: sono già state completate!

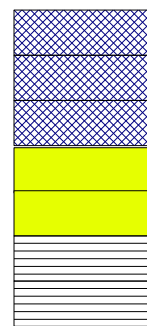
Transazioni atomiche concorrenti

- Per ora abbiamo trattato le transazioni senza preoccuparci del fatto che possano essere concorrenti, **la concorrenza => interleaving delle istruzioni ...**
- Come si combinano **concorrenza** ed **atomicità** (**atomicità a livello logico**, non di esecuzione sulla CPU)?
- **Idea**: garantire in qualche modo la **proprietà di serializzabilità**!
- **Serializzabilità di un insieme di transazioni**: proprietà per cui la loro esecuzione concorrente è equivalente alla loro esecuzione in una sequenza arbitraria

a ogni colore
corrisponde
una transazione
diversa: ogni
rettangolo è un'
operazione



concorrenza



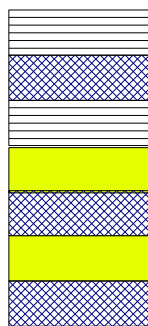
esecuz. sequenziale

si potrebbero eseguire le transazioni in ME, serializzandole sul serio tramite un semaforo mutex ma si tratta di una soluzione troppo rigida

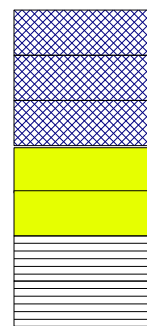
Transazioni atomiche concorrenti

- Per ora abbiamo trattato le transazioni senza preoccuparci del fatto che possano essere concorrenti, **la concorrenza => interleaving delle istruzioni ...**
- Come si combinano **concorrenza** ed **atomicità** (**atomicità a livello logico**, non di esecuzione sulla CPU)?
- **Idea**: garantire in qualche modo la **proprietà di serializzabilità**!
- **Serializzabilità di un insieme di transazioni**: proprietà per cui la loro esecuzione concorrente è equivalente alla loro esecuzione in una sequenza arbitraria

a ogni colore
corrisponde
una transazione
diversa: ogni
rettangolo è un'
operazione



concorrenza



esecuz. sequenziale

si potrebbero eseguire le
transazioni in ME, serializ-
zandole sul serio tramite
un semaforo mutex ma si
tratta di una soluzione
troppo rigida

Esempio

| A | B |
|---|---|
| 3 | 5 |

Siano T1 e T2 due transazioni concorrenti che utilizzano gli stessi dati condivisi, A e B, aventi valori iniziali 3 e 5

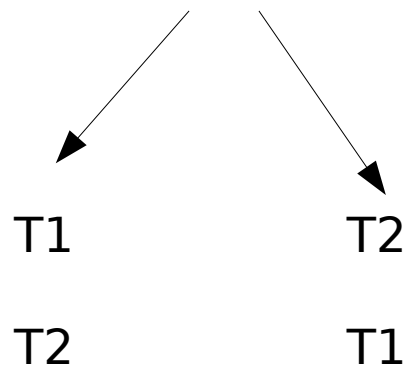
| T1 | T2 |
|----------|----------|
| read(A) | read(A) |
| write(A) | write(A) |
| read(B) | read(B) |
| write(B) | write(B) |

Sono **funzionalmente atomiche**: a livello logico le quattro operazioni costituiscono una funzionalità unica

Esempio

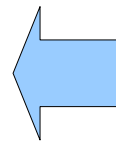
| | |
|---|---|
| A | B |
| 3 | 5 |

Gli **unici stati corretti** per il sistema sono quelli raggiungibili eseguendo T1 e T2 in modo sequenziale



Possibili alternative

| T1 | T2 |
|----------|----------|
| read(A) | read(A) |
| write(A) | write(A) |
| read(B) | read(B) |
| write(B) | write(B) |



Sono **funzionalmente atomiche**: a livello logico le quattro operazioni costituiscono una funzionalità unica

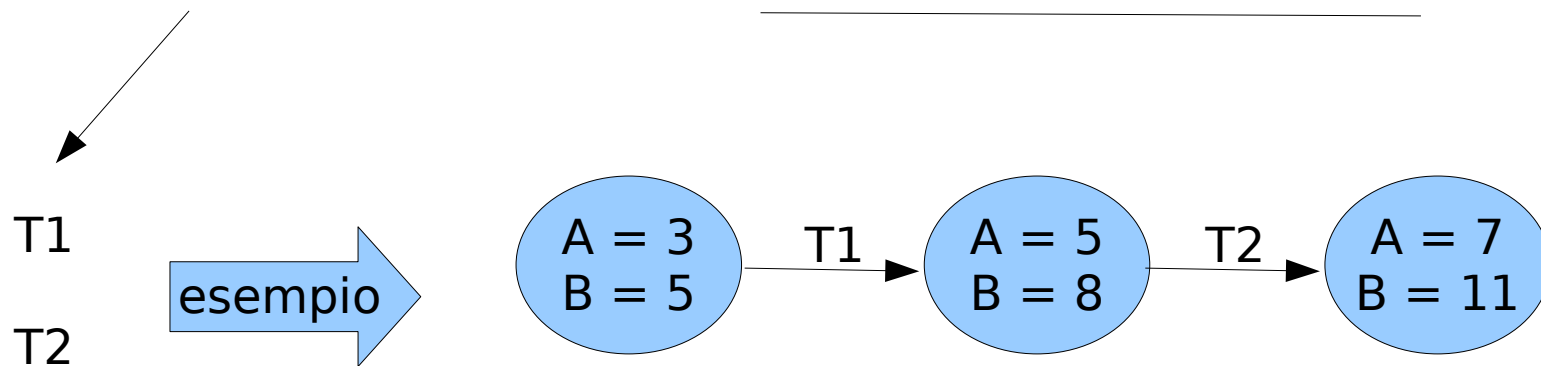
Esempio

| | |
|---|---|
| A | B |
| 3 | 5 |

Gli **unici stati corretti** per il sistema sono quelli raggiungibili eseguendo T1 e T2 in modo sequenziale

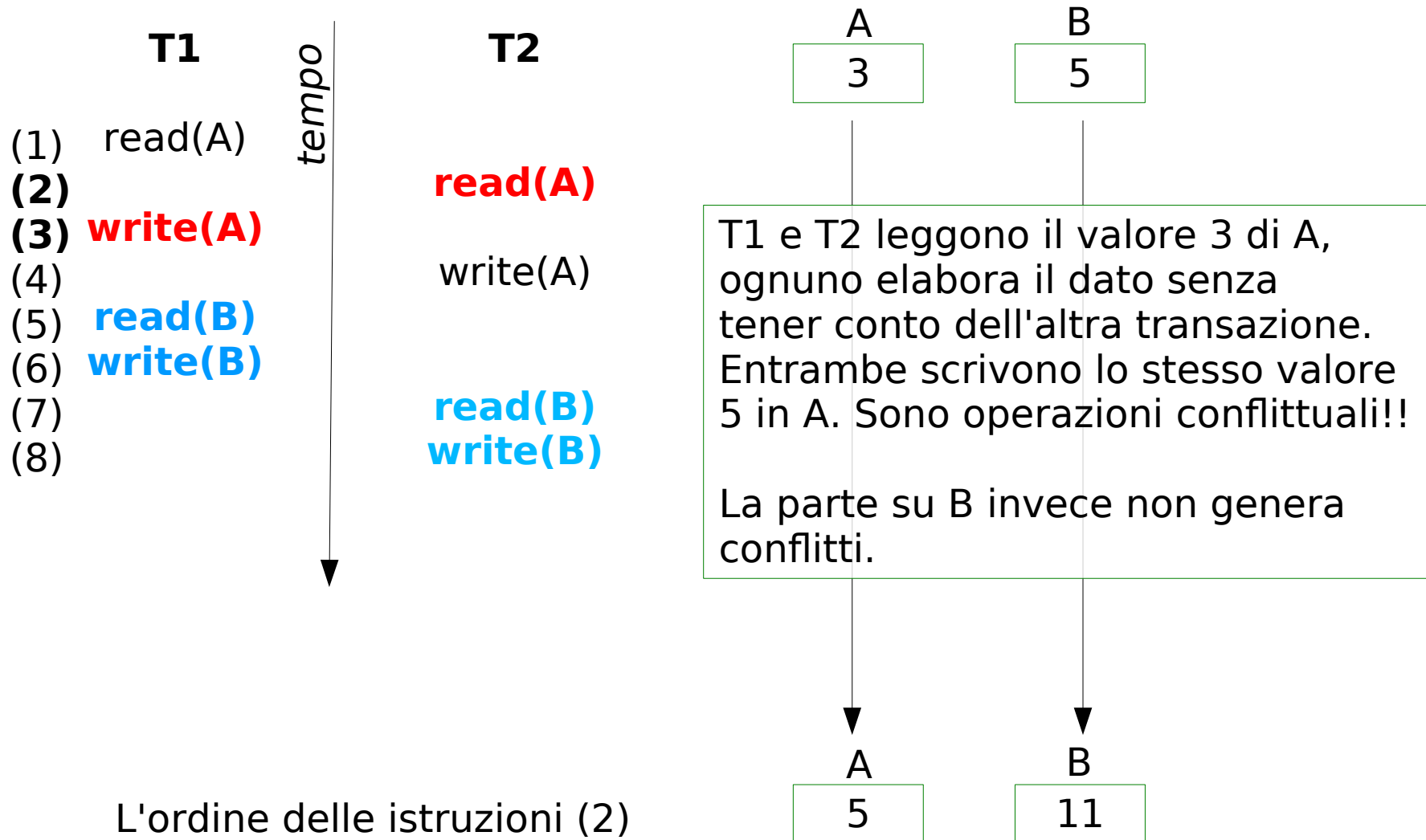
| T1 | T2 |
|----------|----------|
| read(A) | read(A) |
| A=A+2 | A=A+2 |
| write(A) | write(A) |
| read(B) | read(B) |
| B=B+3 | B=B+3 |
| write(B) | write(B) |

Esempio di elaborazione fatta dai processi che eseguono T1 e T2 su A e B



Problema: i sistemi ad alte prestazioni non possono permettersi di proibire l'esecuzione concorrente delle transazioni. Occorre consentire un po' di interleaving però in modo tale che lo stato raggiunto sia uno degli stati leciti (stesso risultato in tempo più rapidi)

Esempio



L'ordine delle istruzioni (2) e (3) non doveva essere invertito

Non è uno stato lecito!!!!

Esempio

CC_cl1 == 100
Prezzo == 50

READ CC_cl1
READ CC_vend
WRITE CC_cl1 CC_cl1 - 50
WRITE CC_vend CC_vend + 50

T1

CC_cl2 == 100
Prezzo == 30

READ CC_cl2
READ CC_vend
WRITE CC_cl2 CC_cl2 - 30
WRITE CC_vend CC_vend + 30

T2

T1 e T2 sono due esecuzioni dello stesso codice, catturano l'acquisto di oggetti diversi da parte di due clienti, effettuato presso lo stesso venditore, il cui CC vale inizialmente 100.

Senza interleaving si raggiunge lo stato finale (corretto) in cui CC_vend vale 180. E se consento l'inteleaving? Consideriamo un esempio ...

Esempio

CC_cl1 == 100
Prezzo == 50

READ CC_cl1

READ CC_vend

WRITE CC_cl1 CC_cl1 - 50
WRITE CC_vend CC_vend + 50

CC_cl2 == 100
Prezzo == 30

READ CC_cl2
READ CC_vend

WRITE CC_cl2 CC_cl2 - 30
WRITE CC_vend CC_vend + 30

▼ tempo

Esempio

CC_cl1 == 100
Prezzo == 50

READ CC_cl1

Ha letto il valore 100

READ CC_vend

WRITE CC_cl1 CC_cl1 - 50
WRITE CC_vend CC_vend + 50

Calcola l'espression
a partire dal valore 100

CC_cl2 == 100
Prezzo == 30

READ CC_cl2
READ CC_vend

Ha letto il valore 100

WRITE CC_cl2 CC_cl2 - 30
WRITE CC_vend CC_vend + 30

Calcola l'espression
a partire dal valore 100

**ERRORE! Valore finale di
CC_vend pari a 150 !**

tempo

Serializzabilità

- In gnerale dati N elementi, possiamo costruire $N!$ (N fattoriale) sequenze diverse
- Quindi date N transazioni possiamo quindi trovare $N!$ sequenze di esecuzione seriale
- Vogliamo consentire un po' di concorrenza per aumentare l'efficienza complessiva dell'esecuzione però ci interessano delle sequenze di esecuzione non seriale che sono equivalenti (dal punto di vista degli effetti) a quelle seriali, che ci danno la garanzia di produrre risultati consistenti

Serializzabilità

- Introduciamo un concetto nuovo: **operazioni conflittuali**:

date due **transazioni** T1 e T2 e le due **operazioni** O1 (appartenente a T1) e O2 (appartenente a T2), se O1 e O2 appaiono in successione, accedono agli stessi dati e almeno una delle due operazioni è una **write**, allora O1 e O2 sono **operazioni conflittuali**

- **NB:**

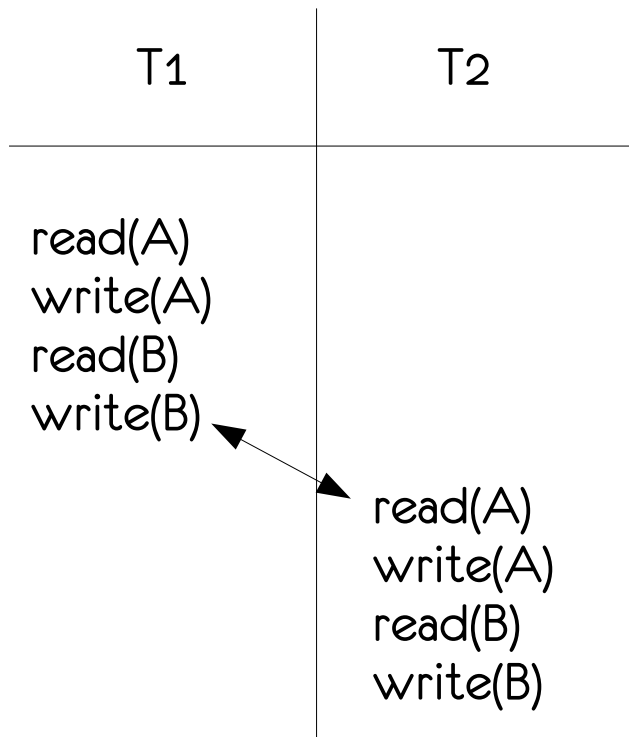
- 1) l'ordine di esecuzione di due operazioni conflittuali incide sul risultato
- 2) due operazioni non conflittuali possono essere scambiate

esempio

| T1 | T2 |
|--|--|
| read(A) write(A) read(B) write(B) | read(A) write(A) read(B) write(B) |

esecuzione seriale di
T1 e T2

esempio

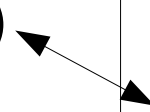


esecuzione seriale di
T1 e T2

Proviamo a introdurre un po' di interleaving
write(B) e read(A) sono conflittuali?

esempio

| T1 | T2 |
|--|--|
| read(A) write(A) read(B) write(B) | read(A) write(A) read(B) write(B) |

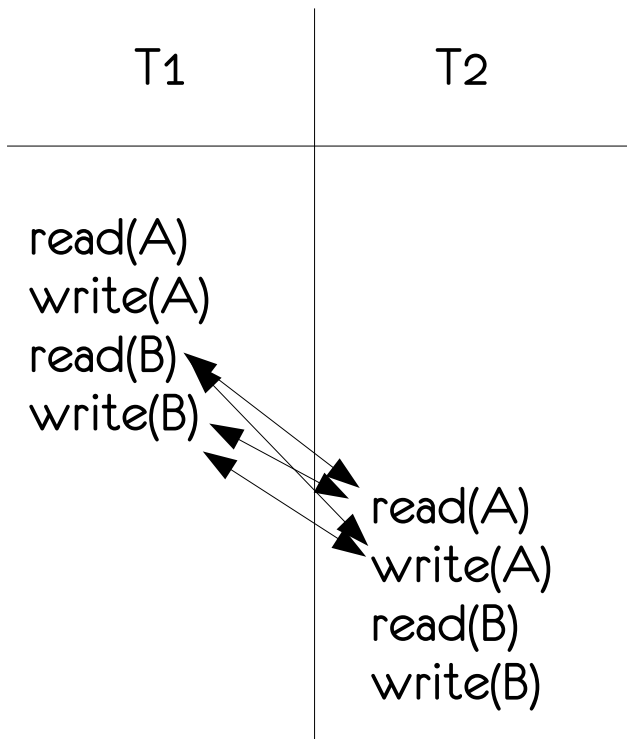


esecuzione seriale di
T1 e T2

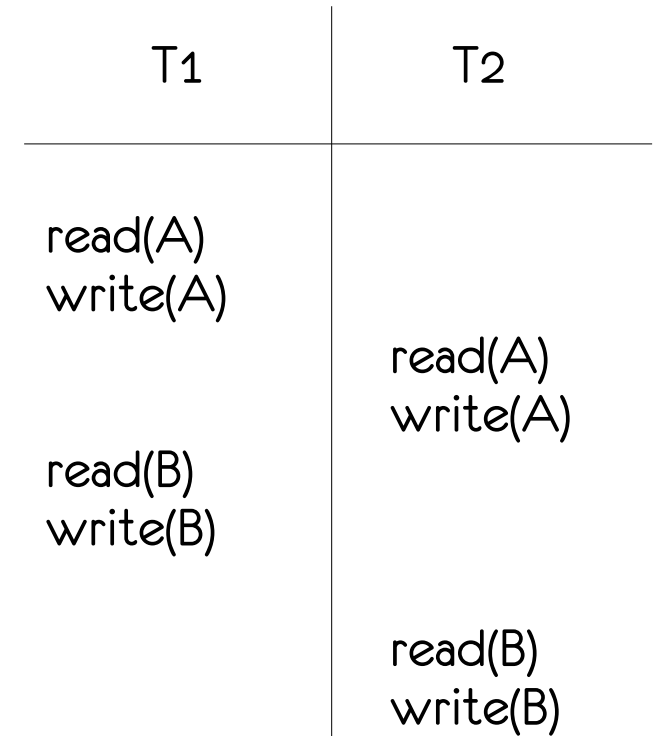
| T1 | T2 |
|--|--|
| read(A) write(A) read(B) write(B) | read(A) write(A) read(B) write(B) |

esecuzione concorrente di
T1 e T2 equivalente a
quella seriale

esempio



esecuzione seriale di
T1 e T2



esecuzione concorrente di
T1 e T2 equivalente a
quella seriale

Dopo qualche iterazione ...

esempio

| T1 | T2 |
|---|---|
| read(A) write(A) read(B) write(B) | read(A) write(A) read(B) write(B) |

conflitto!

Sono operazioni
conflittuali!!!

esecuzione seriale di
T1 e T2

| T1 | T2 |
|--|--|
| read(A) write(A) read(B) write(B) | read(A) write(A) read(B) write(B) |

esecuzione concorrente di
T1 e T2 equivalente a
quella seriale

Protocollo di gestione dei lock

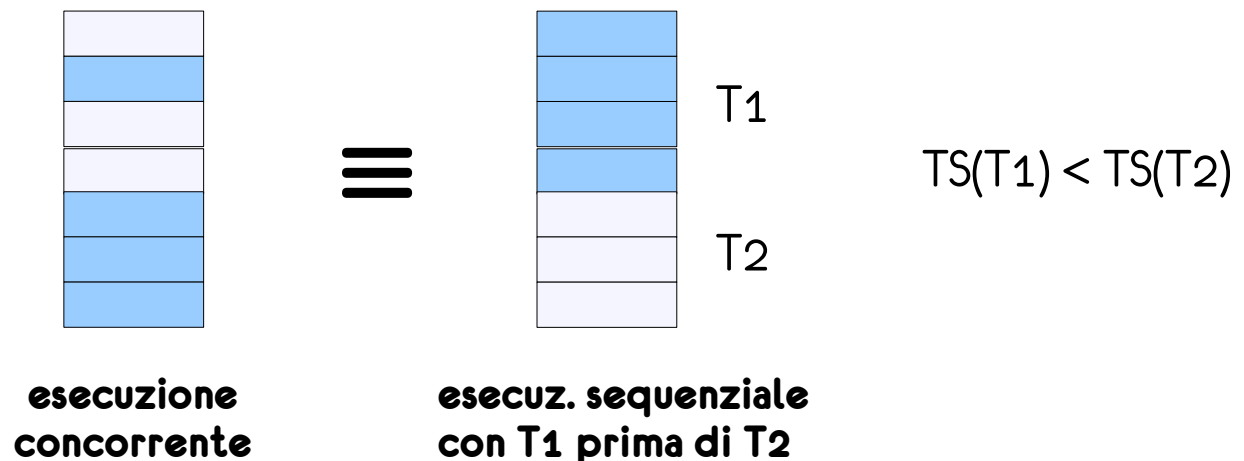
- Per garantire la serializzabilità si può usare un meccanismo di **lock** in cui:
 - a ogni dato soggetto a transazione si associa un lock
 - il lock di un dato può essere:
 - **S** (indica la possibilità di condivisione): la transazione può leggere il dato ma non scriverlo
 - **X** (indica esclusività): la transazione può sia leggere che modificare un certo dato
- una transazione che intenda usare un certo dato deve richiederne il lock appropriato, ed eventualmente attendere che un'altra transazione lo rilasci
- <1> protocollo di gestione dei lock a due fasi
- <2> protocolli di ordinamento dei timestamp

Gestione dei lock a due fasi

- una transazione si può trovare in fase di crescita oppure di riduzione
- inizialmente tutte le transazioni sono in fase di crescita
- una transazione in fase di crescita può ottenere nuovi lock ma non ne rilascia mai nessuno (acquisisce tutte le risorse necessarie)
- una transazione in fase di riduzione può rilasciare lock ma non può richiederne di nuovi (rilascia via via quelle che non le servono più)
- **Commenti:**
 - si può avere deadlock
 - non tutte le sequenze di esecuzione lecite delle operazioni possono essere trovate

Protocolli basati su timestamp

- Cos'è un **timestamp**? È una rappresentazione univoca di un istante temporale, è anche detta marker temporale
- Il sistema assegna a ogni transazione un timestamp prima che questa inizi ad eseguire. Indichiamo il timestamp associato a T con $TS(T)$
- **Idea che vogliamo realizzare**: se due transazioni hanno timestamp $TS(T1) < TS(T2)$ allora il sistema deve garantire che la sequenza di esecuzione delle istruzioni di T1 e T2 sia equivalente all'esecuzione sequenziale in cui T1 viene eseguita prima di T2



Esempio di protocollo

- **Premessa**
- in questo protocollo ogni dato D soggetto a transazione ha due timestamp:
 - $R(D)$: denota il valore più alto di timestamp associato a una transazione che ha letto il dato D
 - $W(D)$: denota il valore più alto di timestamp associato a una transazione che ha eseguito un'operazione di scrittura su D
- questi due valori cambiano nel tempo a seconda degli accessi effettuati a D
- **Regole che definiscono il protocollo**
 - . . .

Regole che definiscono il protocollo

- **<1>** se una transazione **T** desidera **leggere D**:
 - **<1.a>** se $TS(T) < W(D)$: ho una transazione vecchia che cerca di leggere un valore sovrascritto da una più recente -> **azione**: si annulla la lettura richiesta e si esegue una sequenza di undo per annullare T
 - **<1.b>** se $TS(T) > W(D)$: ok -> **azione**: si effettua la lettura e si aggiorna $R(D)$ assegnando $\max(R(D) , TS(T))$

Regole che definiscono il protocollo

- **<2>** se una transazione T desidera **scrivere D** :
 - **<2.a>** se $TS(T) < R(D)$: problema, si vuole fare un aggiornamento che avrebbe dovuto precedere l'ultima lettura -> **azione**: si annulla la write e si effettuano degli undo per annullare T
 - **<2.b>** se $TS(T) < W(D)$: problema, si vuole fare un aggiornamento obsoleto -> **azione**: si annulla la write e si effettuano degli undo per annullare T
 - **<2.c>** si esegue la write e si aggiorna $W(D)$ a $TS(T)$

NB: a una transazione fallita e disfatta viene assegnato un nuovo timestamp, poi la si riavvia

Esempio

Partenza: abbiamo due transazioni, **T1 con $TS(T1) = 1$ e T2 con $TS(T2) = 2$**
Le due transazioni sono descritte qui sotto:

| T1 | T2 |
|---|---------------------------------|
| read(A) read(B) write(A) read(B) | write(B) read(A) write(A) |

Memento:

anche se non eseguite in modo atomico, vogliamo lo stesso risultato ottenuto eseguendole in modo atomico!!!

Proviamo a simularne un'esecuzione (immaginando un certo interleaving) e vediamo come si comporta l'algoritmo basato su timestamp

Vogliamo che l'effetto finale sia identico a quello che si avrebbe in una esecuzione sequenziale di T1 e T2, sia esse T1 e poi T2 oppure T2 e poi T1

Simulazione 1/3

| T1 | T2 |
|----------|----------|
| read(A) | write(B) |
| read(B) | read(A) |
| write(A) | write(A) |
| read(B) | |

Dati da gestire con le transazioni:

A inizialmente $R(A) = 0$ $W(A) = 0$
B inizialmente $R(B) = 0$ $W(B) = 0$

inizia T1

read(A): è eseguibile? Applichiamo l'algoritmo

$TS(T1) < W(A)?$ $1 < 0?$ no!
eseguo read(A) e pongo $R(A) = \max(R(A), TS(T1)) = 1$

read(B): è eseguibile?

$TS(T1) < W(A)?$ $1 < 0?$ no!
eseguo read(B) e pongo $R(B) = 1$

Supponiamo che ora subentri T2 ...

Simulazione 2/3

| T1 | T2 |
|----------|----------|
| read(A) | write(B) |
| read(B) | read(A) |
| write(A) | write(A) |
| read(B) | |

Dati da gestire con le transazioni:

A attualmente $R(A)=1$ $W(A)=0$
B attualmente $R(B)=0$ $W(B)=0$

inizia T2

write(B): è eseguibile?

$TS(T2) < R(B)?$ $2 < 0?$ no!

$TS(T2) < W(B)?$ $2 < 0?$ no!

eseguo write(B) e pongo $W(B) = TS(T2) = 2$

read(A): è eseguibile?

$TS(T2) < W(A)?$ $2 < 0?$ no!

eseguo read(A) e pongo $R(A) = 2$

Simulazione 3/3

| T1 | T2 |
|----------|----------|
| read(A) | write(B) |
| read(B) | read(A) |
| write(A) | write(A) |
| read(B) | |

Dati da gestire con le transazioni:

A attualmente $R(A) = 2$ $W(A) = 0$
B attualmente $R(B) = 0$ $W(B) = 2$

continua T1

write(A): è eseguibile?

$TS(T1) < R(A)?$ $1 < 2?$ sì!

sto cercando di assegnare ad A un valore ormai obsoleto

<1> non eseguo write(A)

<2> rollback di T1

<3> assegno a T1 un nuovo TS (es. 3) e la riavvio

Osservazioni

- L'algoritmo non impone un particolare ordinamento (considerato corretto) fra le transazioni
- L'unico scopo è far sì che tutte le volte che una transazione legge un dato questo abbia o il valore che aveva all'inizio della transazione stessa oppure sia stato modificato dalla transazione stessa
- In questo consiste la garanzia di atomicità funzionale, diversa dall'atomicità di esecuzione:
 - **atomicità funzionale:** interleaving consentito
 - **atomicità di esecuzione:** interleaving disabilitato
- **dove si ha il guadagno?** Transazioni che non interferiscono l'una con l'altra perché o usano dati diversi o l'una non modifica i dati usati dall'altra possono essere eseguite in modo concorrente
- **l'algoritmo non è preventivo:** identifica situazioni problematiche e le aggiusta