

Processi, RAM e file system

Processi e ram

- Abbiamo visto che di ogni processo viene mantenuta un'astrazione
- **Dove?** In RAM
- Cosa significa che un processo è in RAM?

Processi e ram

- Abbiamo visto che di ogni processo viene mantenuta un'astrazione
- **Dove?** In RAM
- Cosa significa che un processo è in RAM? Il codice che esegue, il suo stack, in suo heap sono contenuti in RAM
- **Alcuni problemi di rappresentazione:**
 - occorre associare porzioni di RAM ai vari processi
 - occorre un modo per sapere se e quanta RAM è libera qualora venga generato un nuovo processo

Gestione della memoria

principale e secondaria

La memoria principale è limitata

il SO è responsabile delle seguenti attività:

- mantenere l'informazione relativa a chi sta usando quale parte di memoria e quale parte è invece libera
- assegnare/revocare lo spazio a seconda delle necessità
- decidere quali (parti di) processi vanno trasferiti in memoria centrale e quali vanno rimossi

il SO gestisce anche la memoria secondaria (hard disk)

- assegnazione dello spazio / scheduling del disco
- quando la quantità di memoria richiesta dai processi è troppo elevata non è possibile mantenerli tutti in RAM, occorre tenerne alcuni in memoria secondaria

Memoria e file system

- Gli utenti vedono la memoria organizzata in file
- **file: unità logica di archiviazione**, di norma organizzati in una struttura a **directory**
- Gli utenti usano nomi e cammini, il SO deve saper individuare i blocchi di memoria corrispondenti: per ogni file è necessario mantenere diverse proprietà, es. limitazioni all'accesso da parte di utenti diversi, tipo del file, data di creazione, ...

sistemi operativi e utenti

capitolo 2 del libro (VII ed.)

SO come ambiente di lavoro

Classi di servizi offerti 1/2

Interfaccia utente (user interface)

- **command-line**: comandi espressi come stringhe di caratteri
- **interfacce grafiche (GUI)**: sistema grafico a finestre con puntatore e menu

Esecuzione di programmi

- deve essere possibile far eseguire programmi
- i programmi possono richiedere l'accesso a file e ad dispositivi di I/O (es. monitor)
- deve essere possibile rilevare lo stato di terminazione (corretta o errata) di un programma

SO come ambiente di lavoro

Classi di servizi offerti 2/2

Comunicazione fra processi

- modello a memoria condivisa: i processi scrivono/leggono dati in/da un'area comune che deve essere gestita in modo da evitare inconsistenze
- modello a scambio di messaggi: il SO gestisce lo scambio di pacchetti di informazione fra i diversi processi

Protezione

- gli utenti possono desiderare di limitare l'accesso all'informazione di loro proprietà
- non tutti possono eseguire tutti I comandi, tutti gli applicativi, leggere/modificare qualsiasi file

Interfacce utente

- In molti SO (es. Unix, Windows XP) l'interfaccia utente è un **programma particolare** che viene avviato all'atto del login (autenticazione ed accesso alla macchina)
- **Command-line interface** o interprete di comandi o **shell**: esegue un ciclo infinito in cui attende un comando, lo esegue, torna in attesa. I comandi possono essere codificati all'interno dell'interprete stesso oppure possono corrispondere a nomi di programmi che vengono identificati dall'interprete, caricati in memoria e quindi eseguiti
- **Graphical User Interface** (GUI): vige la metafora del tavolo da lavoro (desktop), su cui sono posti oggetti selezionabili ed attivabili tramite puntatore

Struttura di un sistema operativo

capitolo 2 del libro (VII ed.), da 2.6 in
poi

Come si implementa un SO?

- Originariamente scritti in linguaggio assembly di specifici processori
 - Dipendenza dall'hardware → poca portabilità
 - Esempio
 - MS-DOS era scritto nel linguaggio assembly del processore Intel 8088, quindi poteva essere usato solo su processori Intel
- Attualmente SO sono scritti in **linguaggi di programmazione di alto livello**
 - es. Unix, Linux e Windows XP sono scritti in C
- CPU attuali troppo complesse per riuscire a programmarle ancora in assembly direttamente

SCELTA DI UN LINGUAGGIO CHE SUPPORTI LA PORTABILITÀ

Passo 1: definire i criteri

- Per quale scopo e per quale tipo di uso e di utenza verrà progettato il SO?
- Messi in luce questi aspetti si passa alla **definizione dei criteri più opportuni**, cioè tutte le politiche di gestione di tutte le risorse

DEFINIZIONE DELLE POLITICHE AD ALTO LIVELLO

Criteri e meccanismi

- **criterio**: specifica un comportamento, **cosa** fare in una certa circostanza (**es. alle 20:00 accendi il riscaldamento**). Sono anche detti politiche.
- **meccanismo**: strumenti, anche software, neutri rispetto ai criteri (**es. istruzioni per avviare il riscaldamento**)
- è possibile implementare criteri diversi con gli stessi meccanismi, per esempio posso adoperare le stesse istruzioni di accensione del riscaldamento per definire le politiche:
 - alle 6:30 accendi il riscaldamento
 - quando la temperatura scende a 16.5 gradi accendi il riscaldamento

DEFINIZIONE DELLE POLITICHE AD ALTO LIVELLO

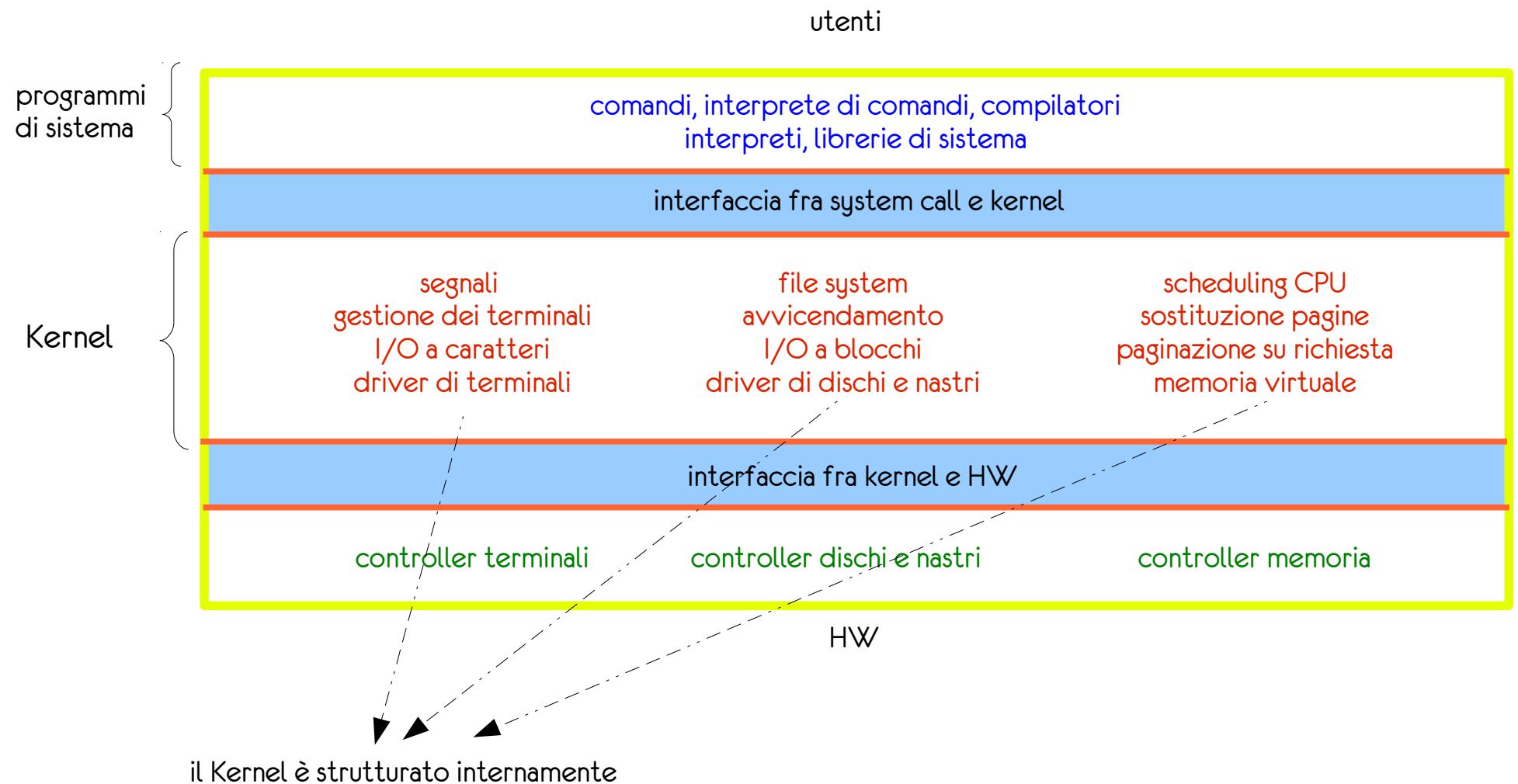
Struttura

- Come ogni programma complesso, **un SO deve essere ben strutturato** per poter essere mantenuto e aggiornato
- **Alcuni noti SO nati come sistemi piccoli e limitati e poi accresciutisi al di là delle previsioni, non avevano una chiara struttura in moduli, con una chiara definizione delle interfacce**
- **Alcuni limiti derivavano dall'HW** in uso. Per esempio l'Intel 8088 non prevedeva dual mode: gli applicativi potevano accedere direttamente all'I/O scrivendo direttamente su video e dischi ... per questo MS-DOS, scritto per I8088, era vulnerabile

Tecniche di modularizzazione

- NB: la modularizzazione è possibile se e solo se supportata dall'hardware
- Tecniche:
 - stratificazione
 - microkernel
 - moduli

Struttura del primo Unix

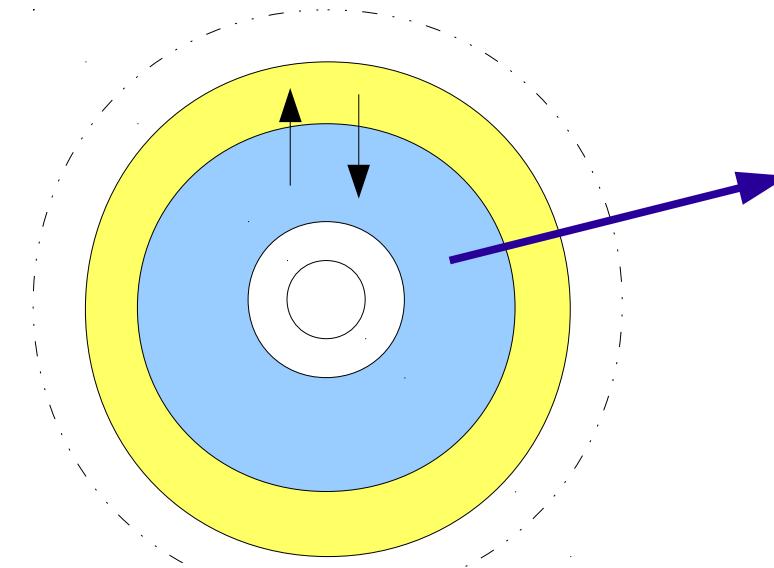


Stratificazione

Tecnica di stratificazione

Il sistema ha una struttura “a cipolla”, lo strato più interno (o strato 0) corrisponde all'HW, quello più esterno all'interfaccia utente

ogni strato realizza un **oggetto astratto, che incapsula dei dati e le operazioni che consentono di elaborare quei dati**, parte delle quali sono rese visibili/accessibili dall'esterno (appartengono all'interfaccia dello strato)



Livello M, contiene:
strutture dati
routine, in parte richiamabili
dal **livello M+1**

Tecnica di stratificazione

Vantaggi e svantaggi

semplicità di progettazione: per realizzare uno strato basta sapere quali funzionalità ha a disposizione, non importa sapere come sono realizzate

semplicità di debug e verifica del sistema: ogni strato usa solo funzionalità messe a disposizione dallo strato immediatamente inferiore, possiamo verificare gli strati uno per volta

difficoltà: definire in modo opportuno gli strati, quale funzionalità sta in quale strato?

minore efficienza in fase di esecuzione, ogni passaggio di stato comporta infatti a chiamata di una nuova funzione, da caricare, che può richiedere parametri, da caricare a loro volta ...

Stratificazione e macchine virtuali

- L'approccio a strati trova il suo vertice massimo nel concetto di **macchina virtuale**: uno strato software che duplica il comportamento di ogni componente hardware del computer
- A che pro? Alcuni scenari:
 - uso di SO diversi su uno stesso computer
 - verifica e debugging di applicativi su SO diversi
 - uso di SO diversi da parte di utenti diversi
 - ...
- Invece di installare un SO direttamente, lo si installa su di una macchina virtuale, che a sua volta lavora su un computer che ha già un proprio SO
- I SO installati sulla macchina virtuale sono detti “ospiti”

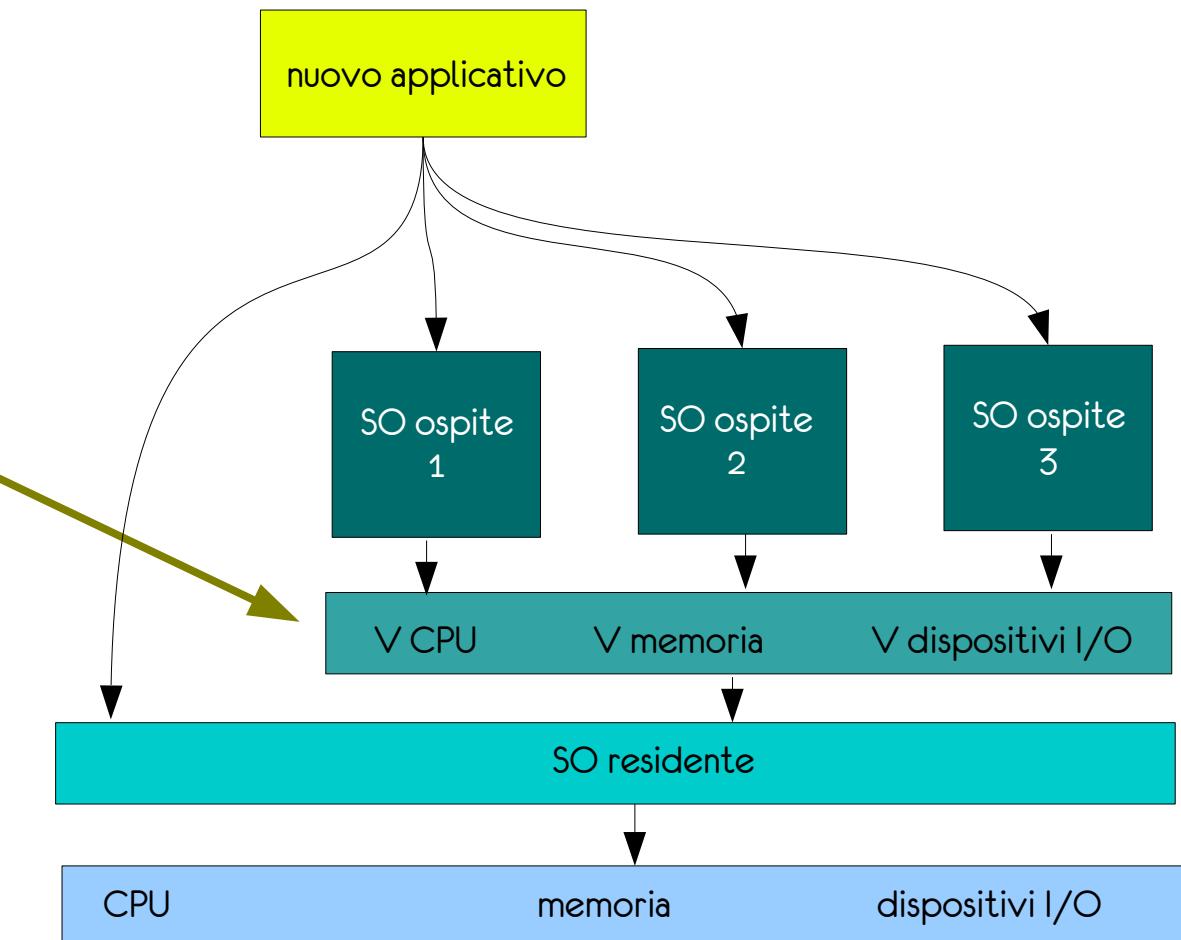
macchine virtuali - Esempi 2/2

Java Virtual Machine

CLR di .Net

Vmware

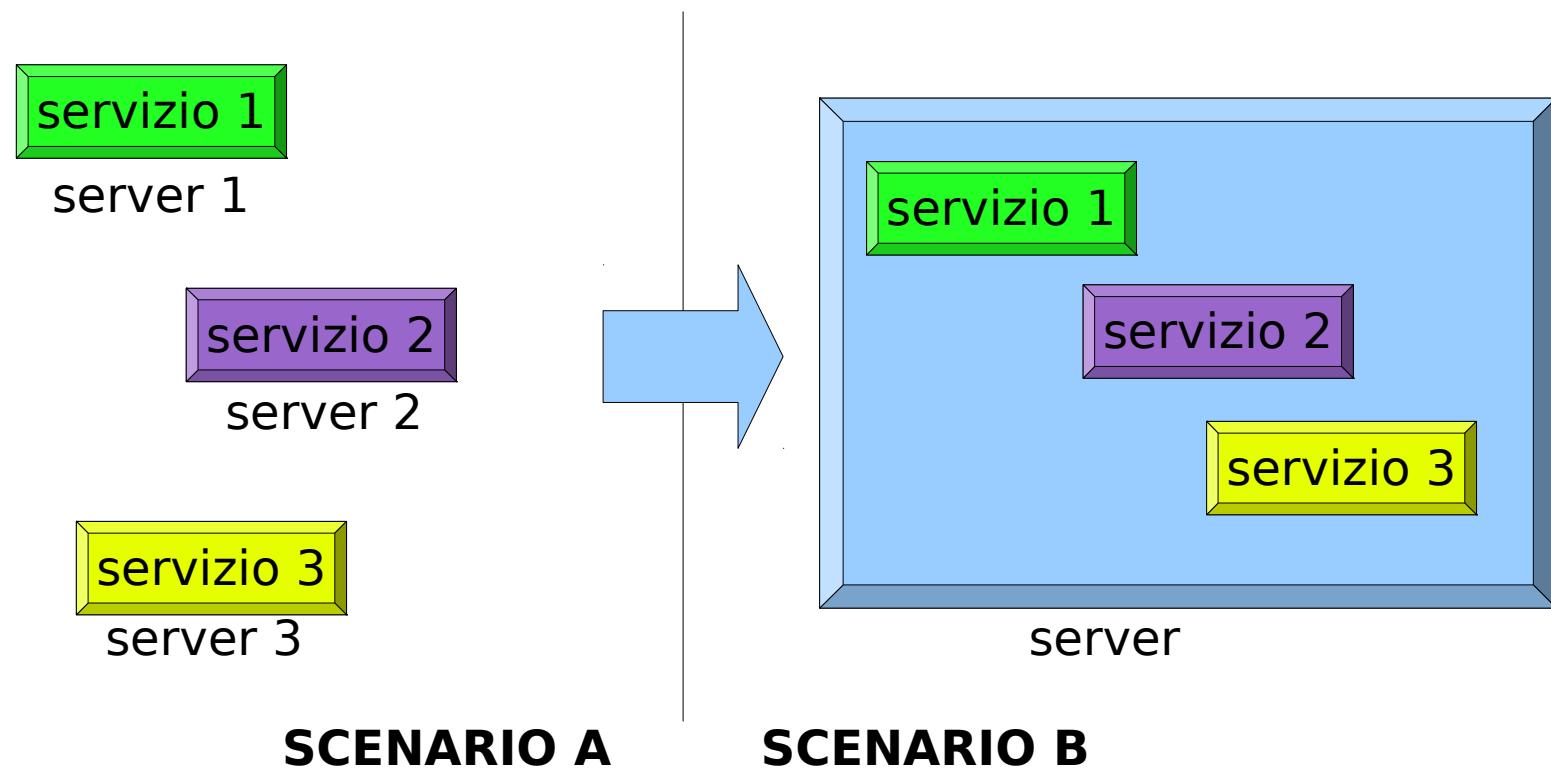
Hypervisor:
elemento di una
macchina virtuale che
fornisce un'astrazione
delle diverse risorse
hardware



Virtualizzazione delle piattaforme

tema attuale, non strettamente legato ai SO ma più all'erogazione di servizi

es. Server Consolidation



Pro/contro virtualizzazione

La riduzione del numero dei server comporta:

- riduzione di occupazione di spazio
- riduzione dei contratti di manutenzione
- riduzione del consumo energetico (funzionamento e refrigerazione)
- se permangono più server posso attuare politiche di bilanciamento del carico

Ridurre il numero dei server ha dei vantaggi ma ridurre eccessivamente il numero dei server comporta una riduzione delle prestazioni

Tecnica a microkernel

Nata verso la metà degli anni '80 con la realizzazione del SO Mach,
presso la Carnegie Mellon University

L'idea è rimuovere dal kernel tutto ciò che non è essenziale,
spostandolo a livello di applicativo utente.

I microkernel in genere contengono servizi minimi per la gestione di
processi, comunicazione (a scambio di messaggi) e **memoria**

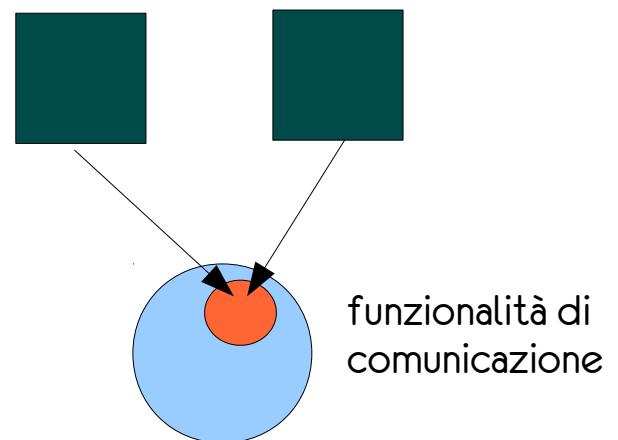
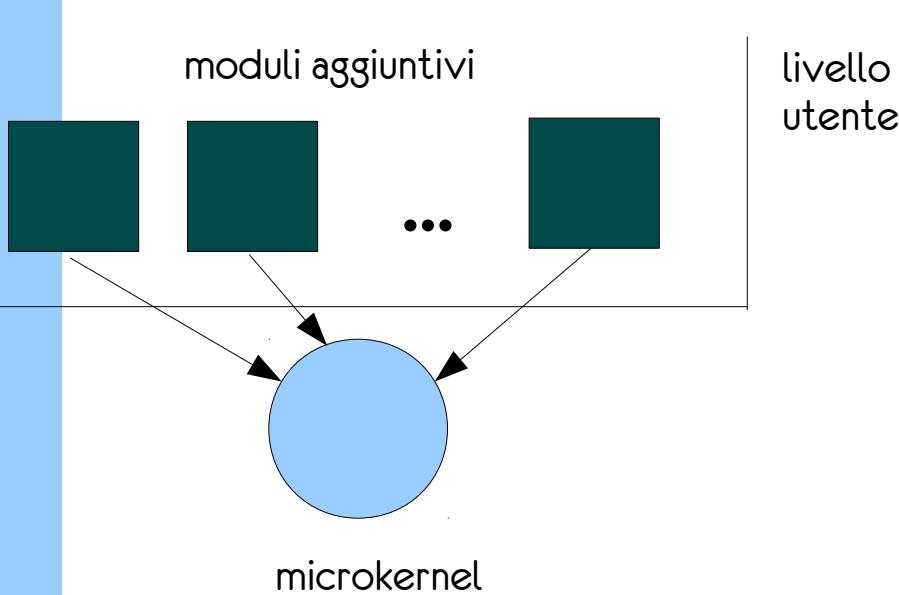


Tecnica a microkernel

Nata verso la metà degli anni '80 con la realizzazione del SO Mach, presso la Carnegie Mellon University

L'idea è rimuovere dal kernel tutto ciò che non è essenziale, spostandolo a livello di applicativo utente.

I microkernel in genere contengono servizi minimi per la gestione di **processi, comunicazione** (a scambio di messaggi) e **memoria**



Inefficienza prestazionale:
lo scambio di messaggi (richiesta/risposta)
appesantisce l'esecuzione col crescere del
numero dei processi

Tecnica a microkernel

vantaggi:

- **facilità di estensione**: i nuovi servizi vengono aggiunti senza modificare il kernel
- il SO risulta più **semplice da adattare** a nuove architetture
- **maggior sicurezza**: un servizio compromesso non incide sull'affidabilità del kernel

svantaggi:

- **sovraffatti** dati dall'esecuzione di processi utente con funzionalità di SO
- in particolare la comunicazione indiretta è un collo di bottiglia

Tecnica a moduli

- **Approccio attualmente considerato il migliore**, adottato da SO come Solaris, Linux, Mac OS
- A un kernel minimale possono essere aggiunti moduli nuovi, in fase di avvio ma soprattutto anche di esecuzione:
 - aggiunta moduli significa aggiunta di system call
 - significa anche aggiunta della capacità di gestire nuovo hardware (per esempio driver per dispositivi specifici)
- Come nei sistemi a microkernel, il kernel gestisce un nucleo ridotto all'osso di funzionalità essenziali, che però possono essere modificate run-time a seconda delle esigenze (per es. per gestire nuovi device o qualche protocollo di networking)
- Ogni modulo ha una propria interfaccia ben definita (come nell'approccio stratificato)

Tecnica a moduli

Differenze e vantaggi:

- a differenza dai sistemi stratificati, ogni modulo può usarne qualsiasi altro (**flessibilità**)
- a differenza dai microkernel, la comunicazione fra moduli è diretta (**efficienza**)

lsmod

per visualizzare I moduli caricati si può utilizzare il comando lsmod:

LSMOD(8)

lsmod

LSMOD(8)

NAME

lsmod - Show the status of modules in the Linux Kernel

SYNOPSIS

lsmod

DESCRIPTION

lsmod is a trivial program which nicely formats the contents of the /proc/modules, showing what kernel modules are currently loaded.

COPYRIGHT

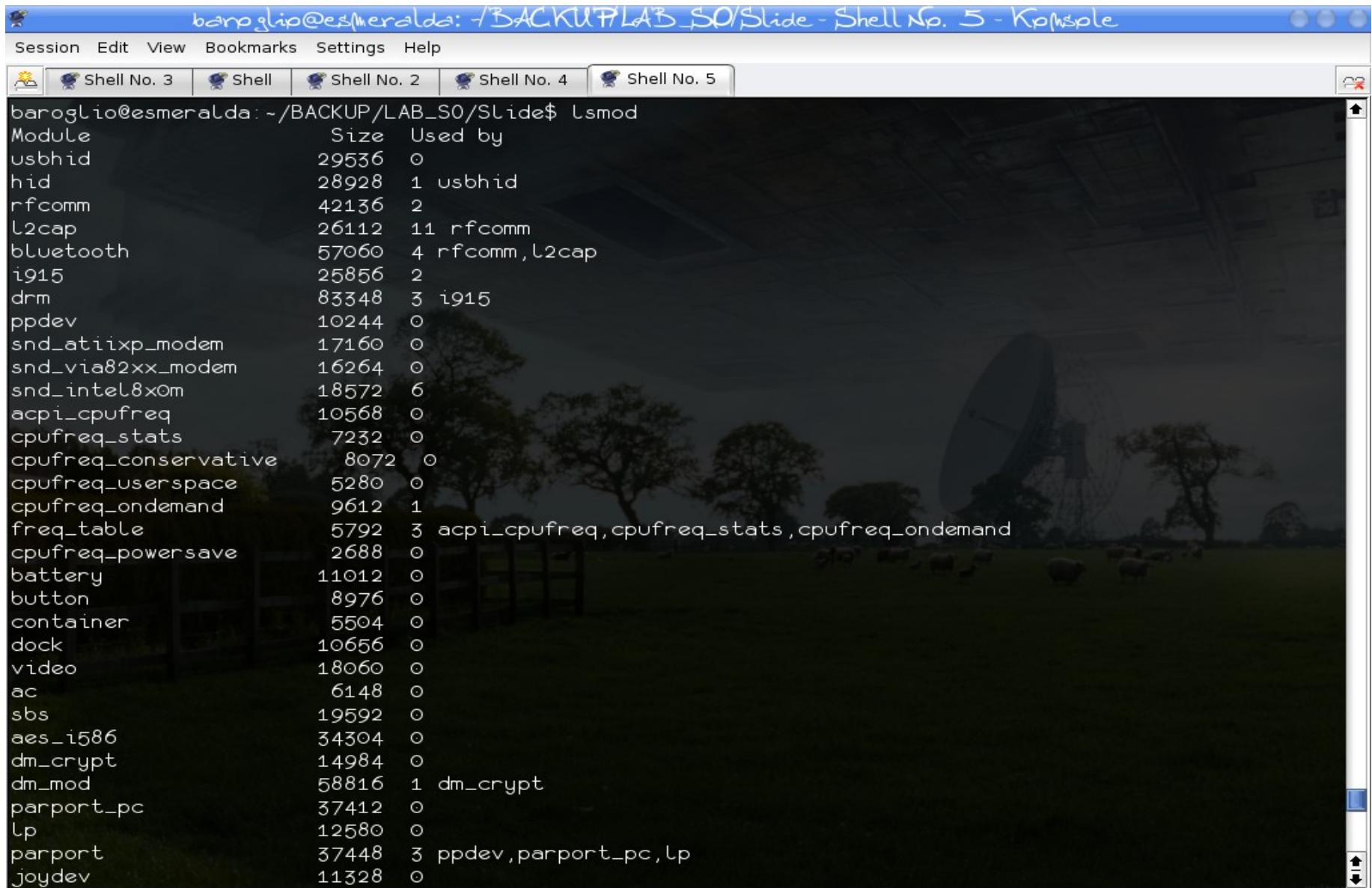
This manual page originally Copyright 2002, Rusty Russell, IBM Corporation. Maintained by Jon Masters and others.

SEE ALSO

insmod(8), modprobe(8), modinfo(8)

lsmod

per visualizzare I moduli caricati si può utilizzare il comando lsmod



The screenshot shows a terminal window titled "baroglio@esmeralda: ~/BACKUP/LAB_S0/Slide - Shell №. 5 - Konssole". The window contains the output of the "lsmod" command, which lists loaded kernel modules along with their sizes and dependencies. The modules listed include usbhid, hid, rfcomm, l2cap, bluetooth, i915, drm, ppdev, snd_atiixp_modem, snd_via82xx_modem, snd_intel8x0m, acpi_cpufreq, cpufreq_stats, cpufreq_conservative, cpufreq_userspace, cpufreq_ondemand, freq_table, cpufreq_powersave, battery, button, container, dock, video, ac, sbs, aes_i586, dm_crypt, dm_mod, parport_pc, lp, parport, and joydev.

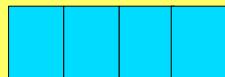
```
baroglio@esmeralda: ~/BACKUP/LAB_S0/Slide$ lsmod
Module           Size  Used by
usbhid          29536  0
hid              28928  1 usbhid
rfcomm          42136  2
l2cap          26112  11 rfcomm
bluetooth      57060  4 rfcomm,l2cap
i915            25856  2
drm             83348  3 i915
ppdev           10244  0
snd_atiixp_modem 17160  0
snd_via82xx_modem 16264  0
snd_intel8x0m   18572  6
acpi_cpufreq    10568  0
cpufreq_stats   7232   0
cpufreq_conservative 8072   0
cpufreq_userspace 5280   0
cpufreq_ondemand 9612   1
freq_table      5792   3 acpi_cpufreq,cpufreq_stats,cpufreq_ondemand
cpufreq_powersave 2688   0
battery          11012  0
button            8976  0
container        5504   0
dock              10656  0
video             18060  0
ac                6148   0
sbs               19592  0
aes_i586         34304  0
dm_crypt         14984  0
dm_mod          58816  1 dm_crypt
parport_pc       37412  0
lp                12580  0
parport         37448  3 ppdev,parport_pc,lp
joydev           11328  0
```



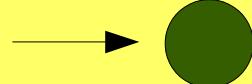
processi

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



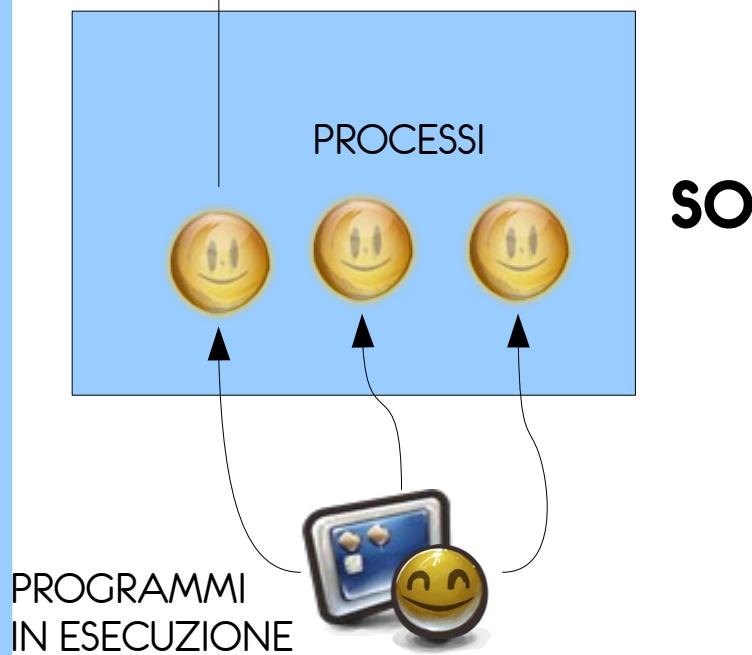
sistema time-sharing:
il tempo di CPU è diviso fra i task di più utenti collegati tramite terminali diversi

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 ESECUSIONE

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

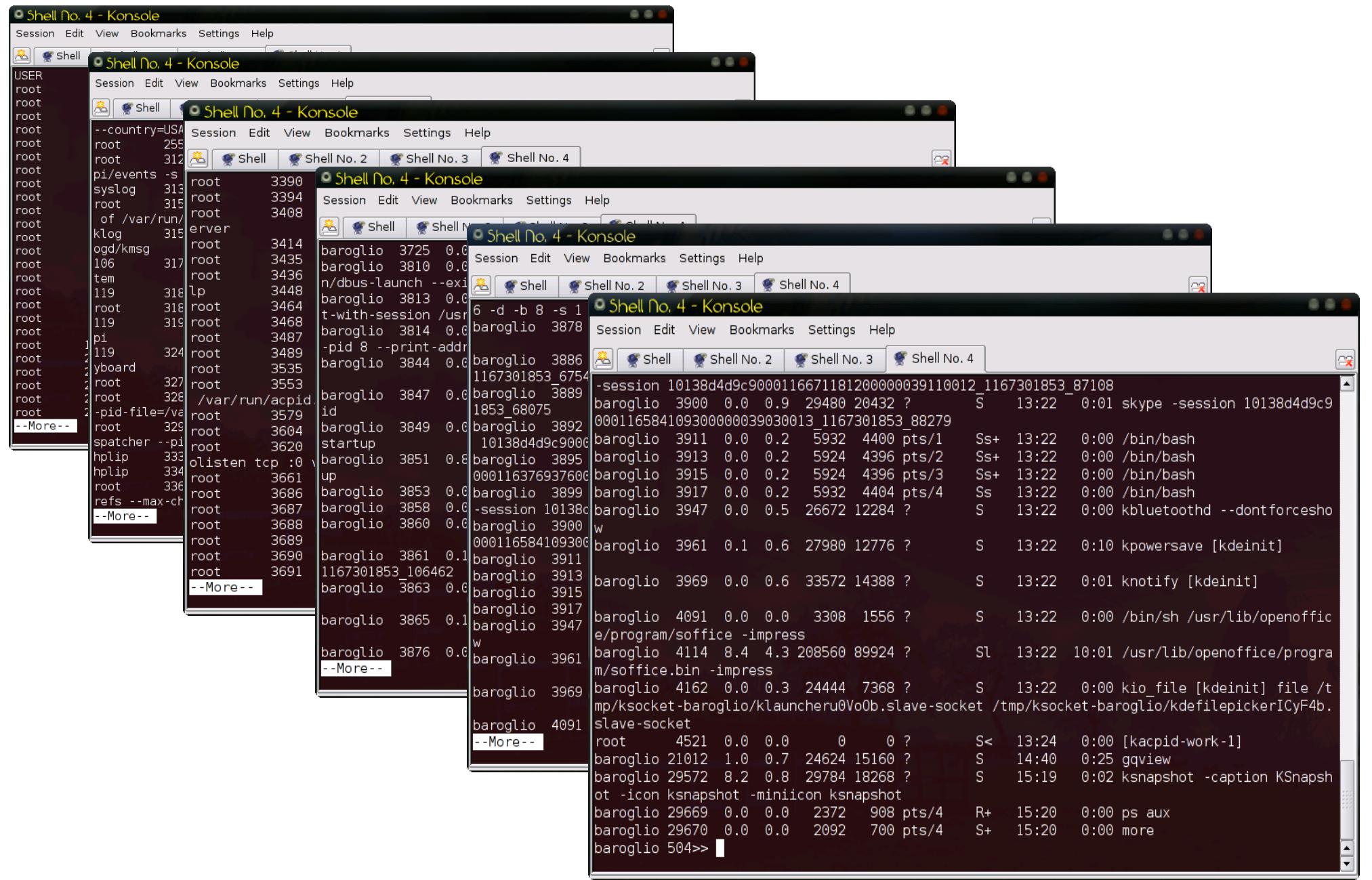
Cristina Baroglio
a.a. 2006/2007

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

Task View

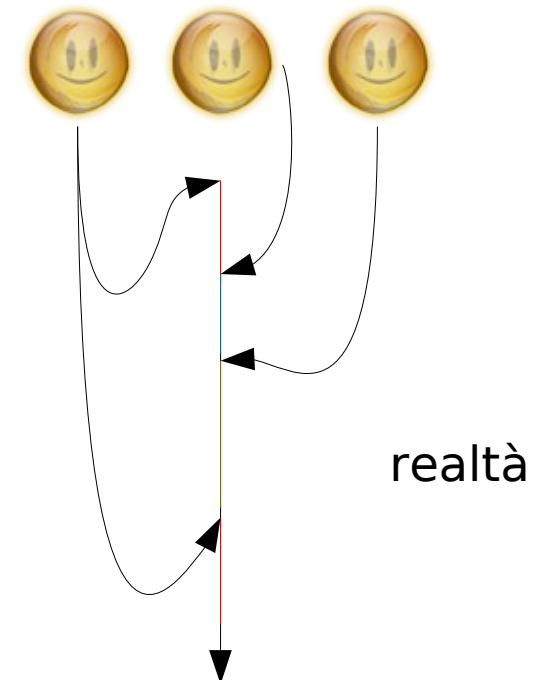
- Master Pages
- Layouts
- Custom Animation
- Slide Transition

103

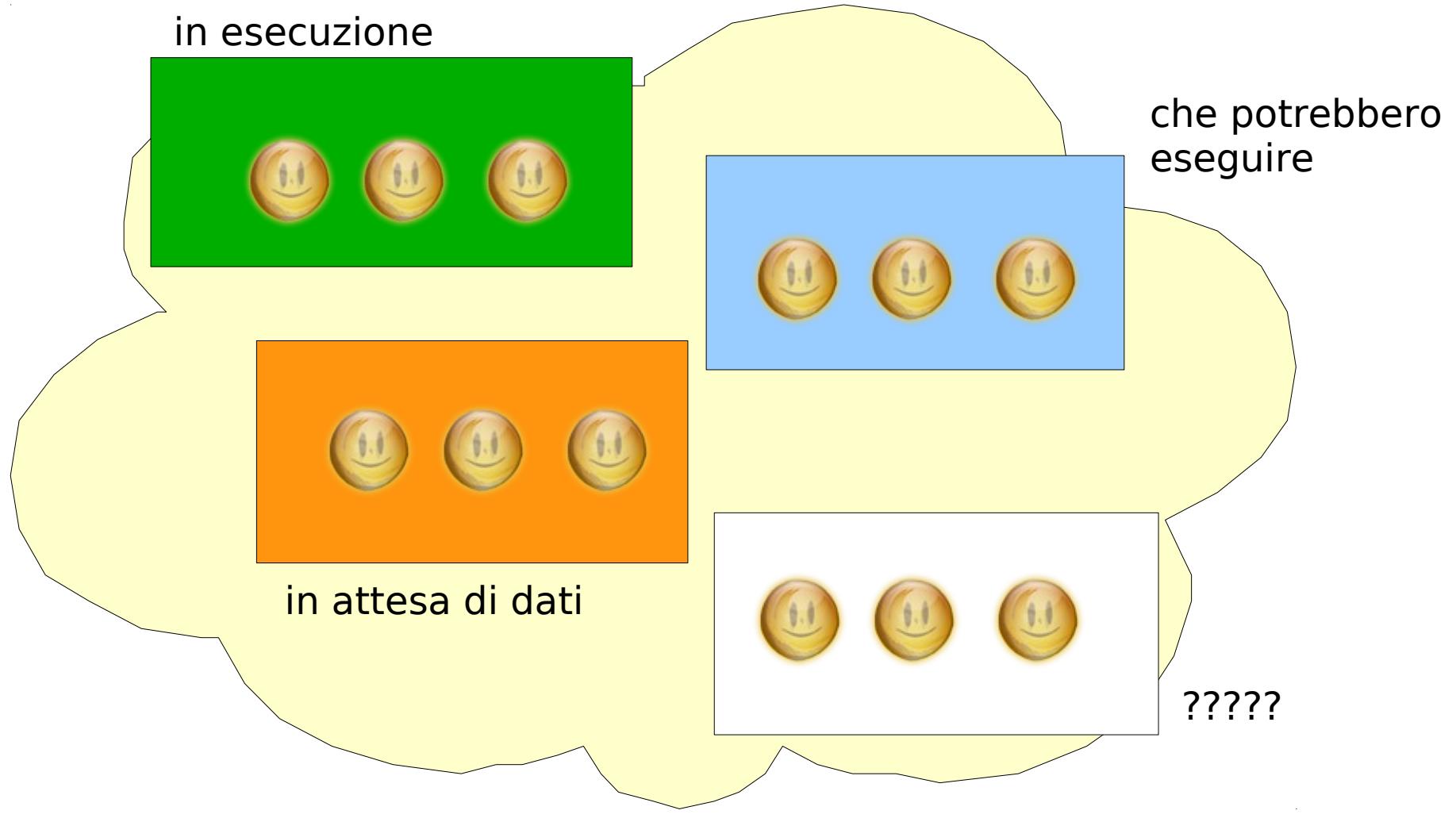


Parallelismo virtuale

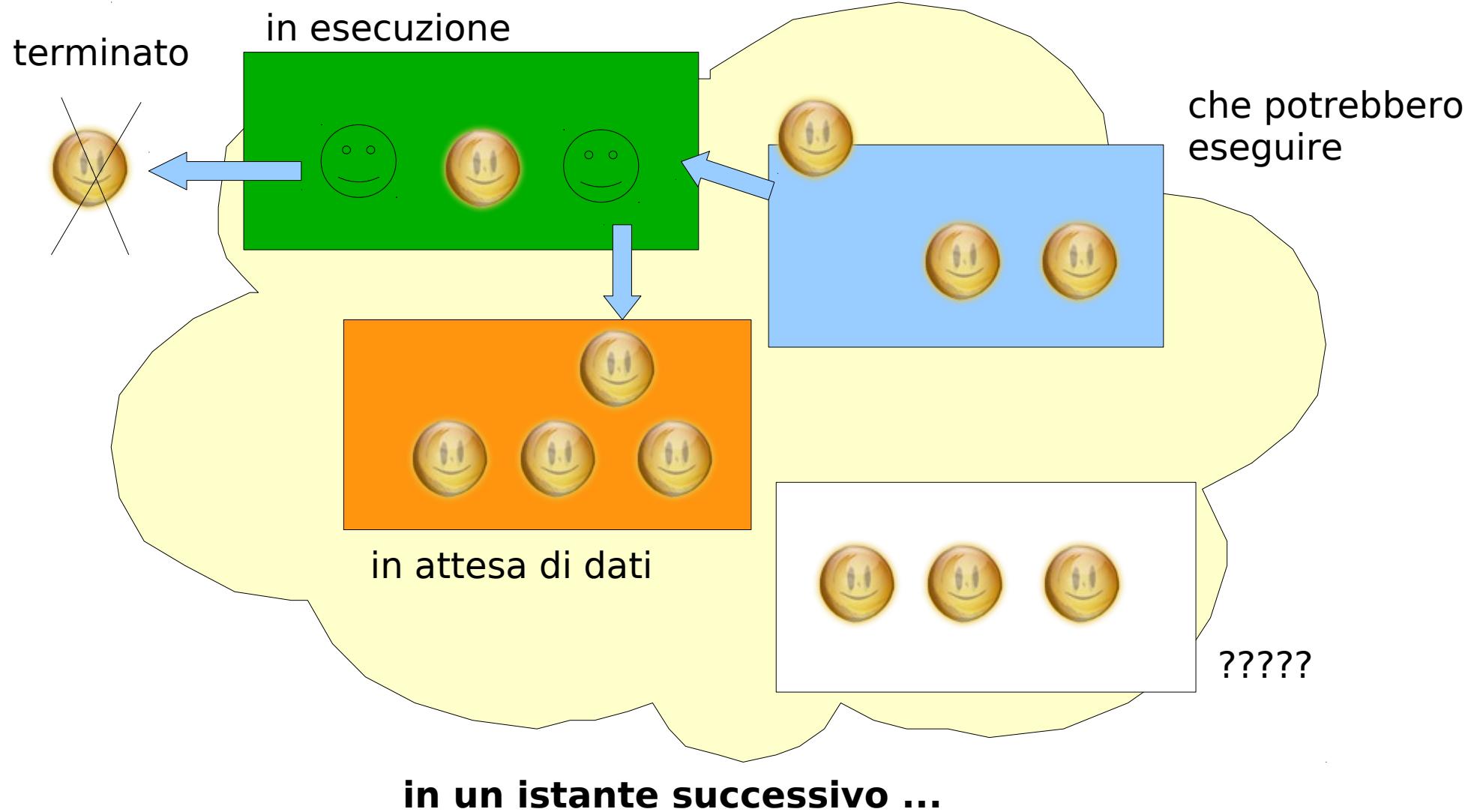
- Attraverso al SO i processi si suddividono l'uso delle risorse in modo tale portare avanti tutti quanti insieme la propria computazione
- Al più un processo per CPU può, in realtà, essere attivo in ogni istante ma gli utenti non se ne accorgono, percepiscono le diverse esecuzioni come parallele



Categorie di processi



I processi cambiano stato

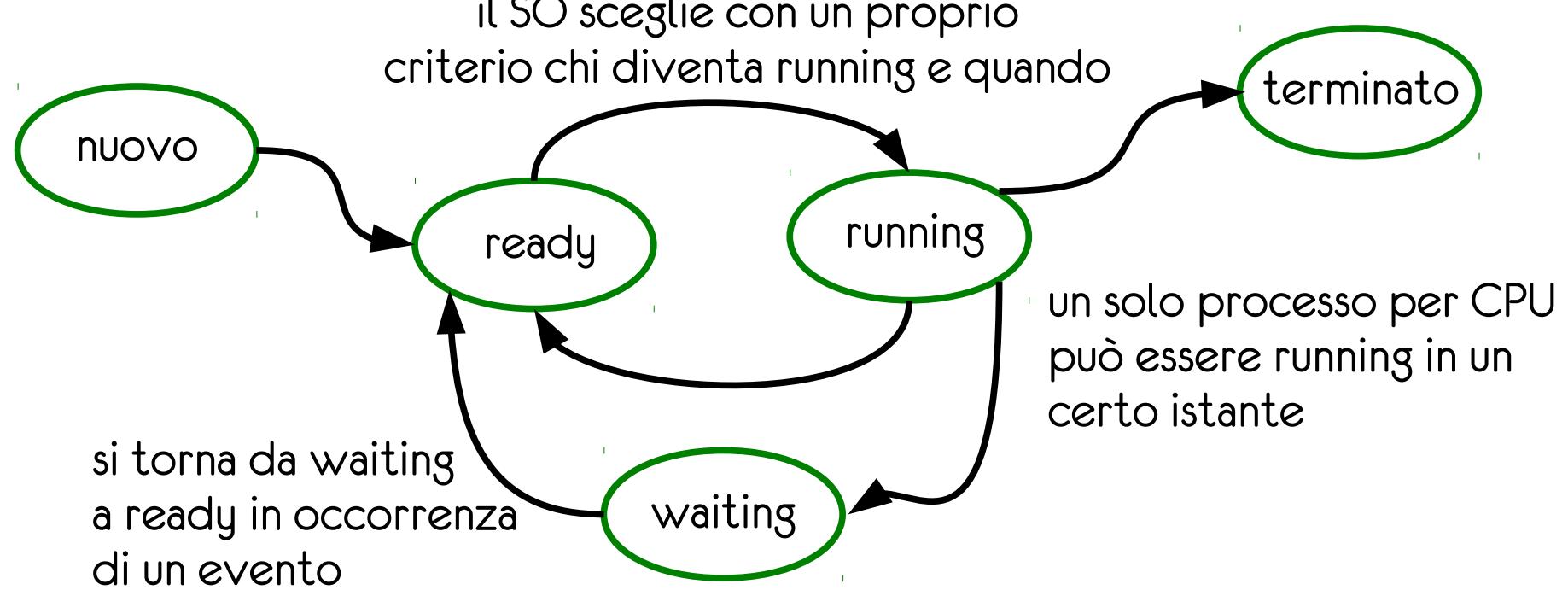


Parallelismo virtuale

- 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

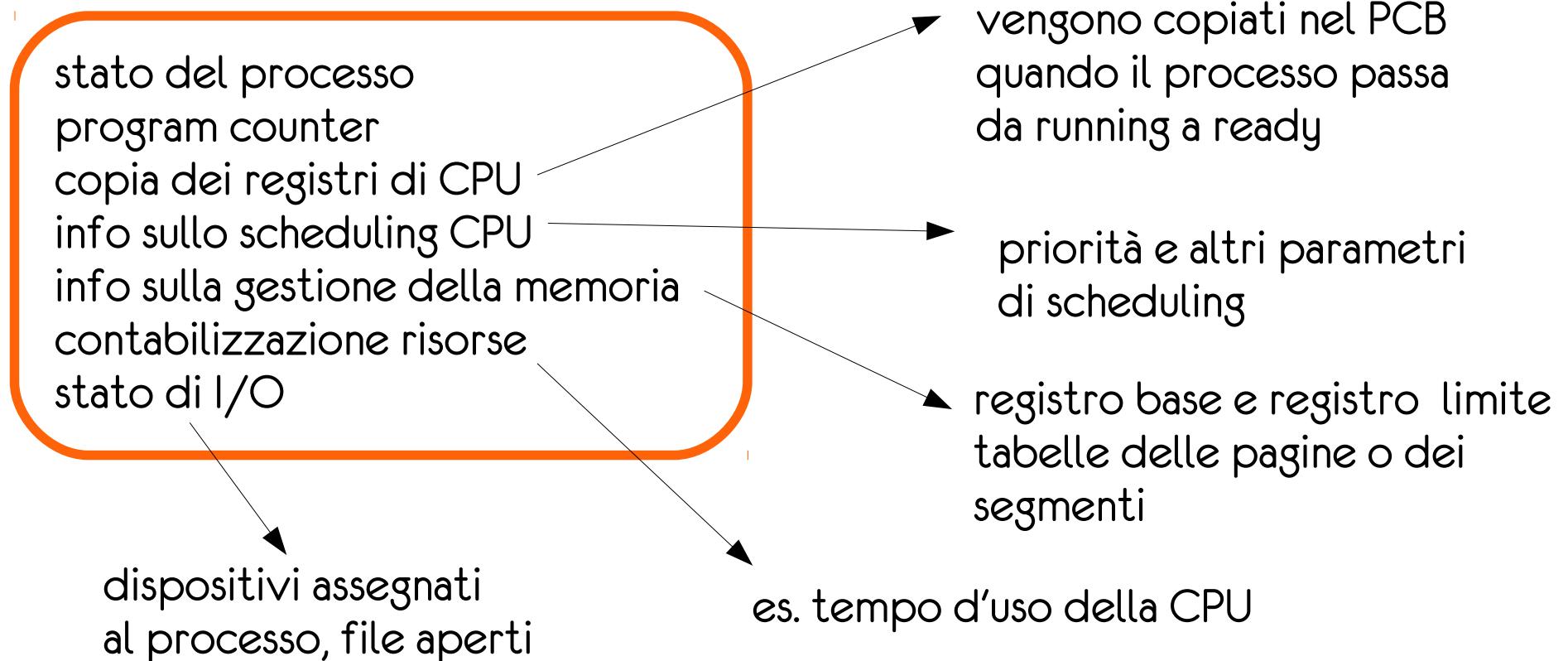
Diagramma di transizione

- Diagramma di transizione degli stati di un processo



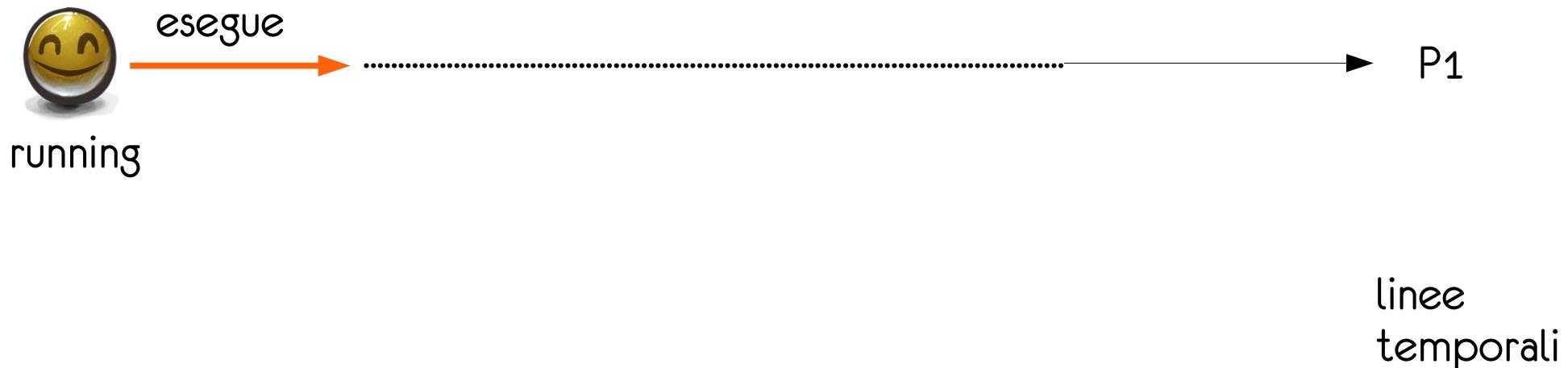
Processi e PCB

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



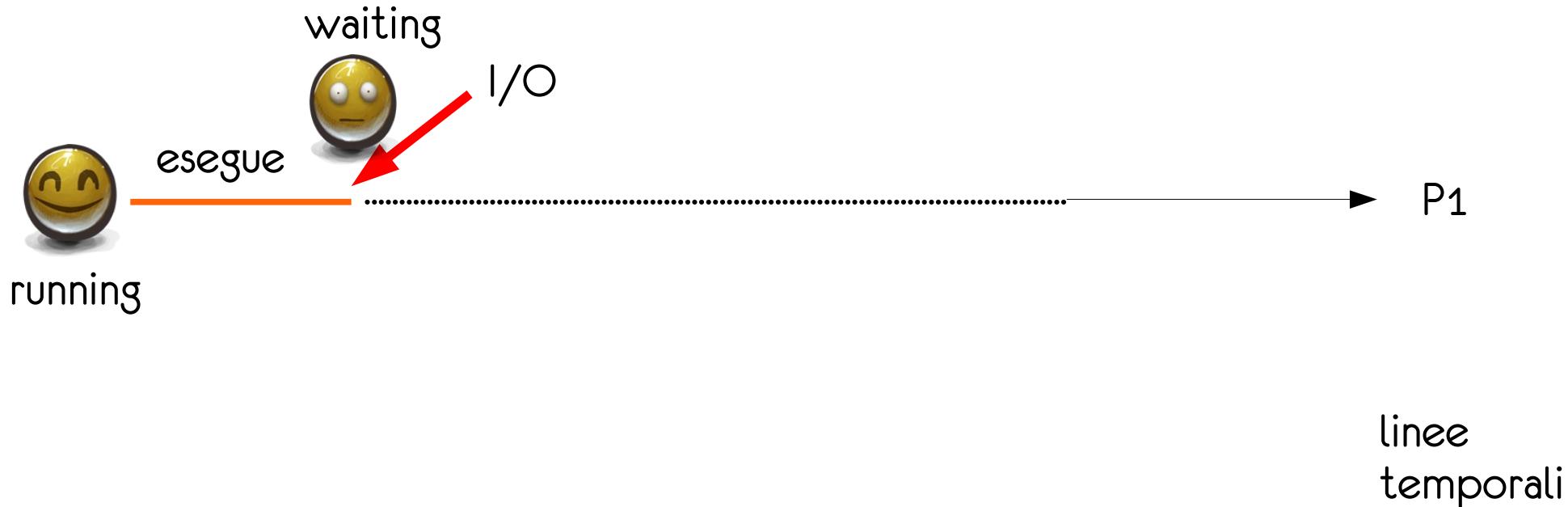
Es. commutazione della CPU

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



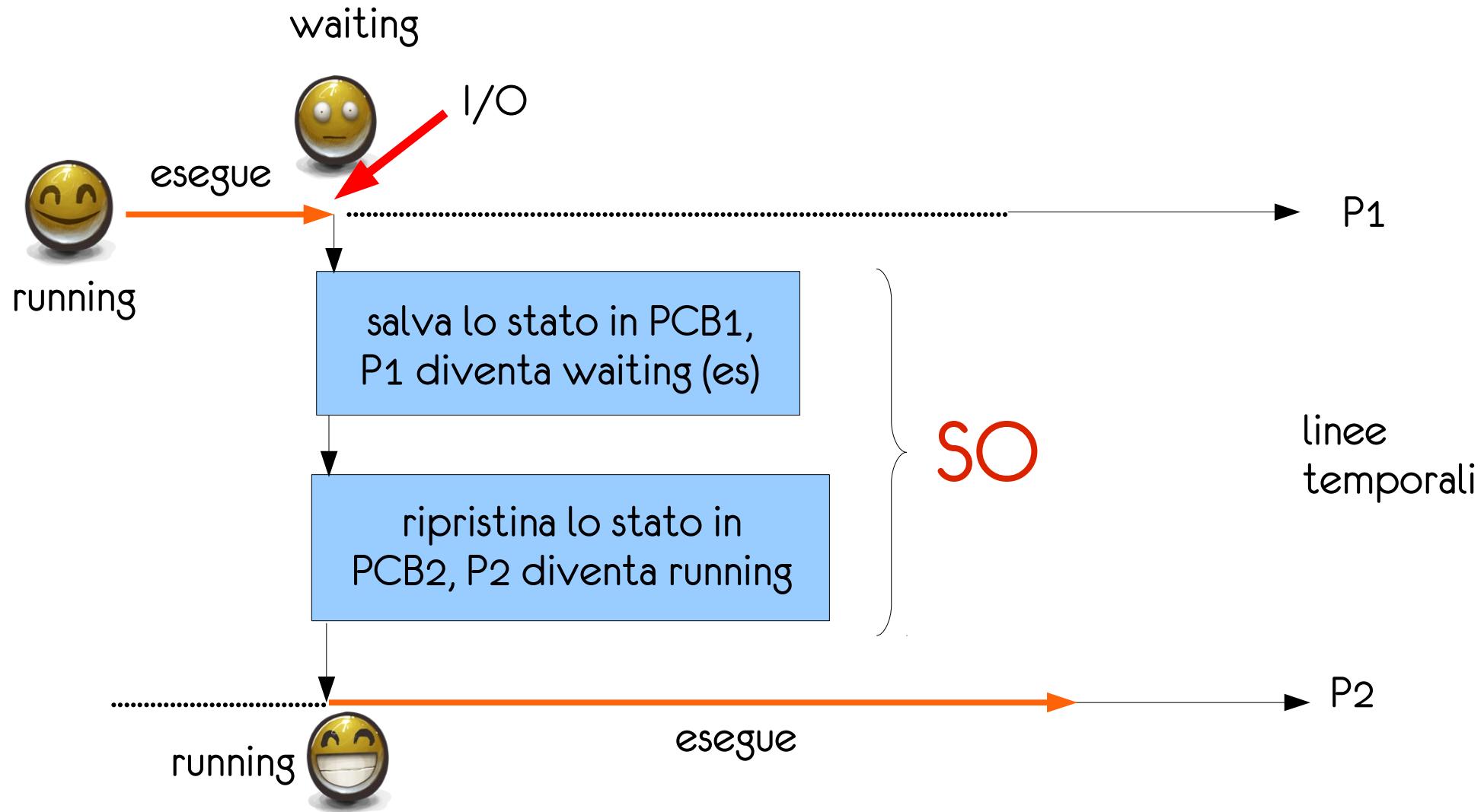
Es. commutazione della CPU

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



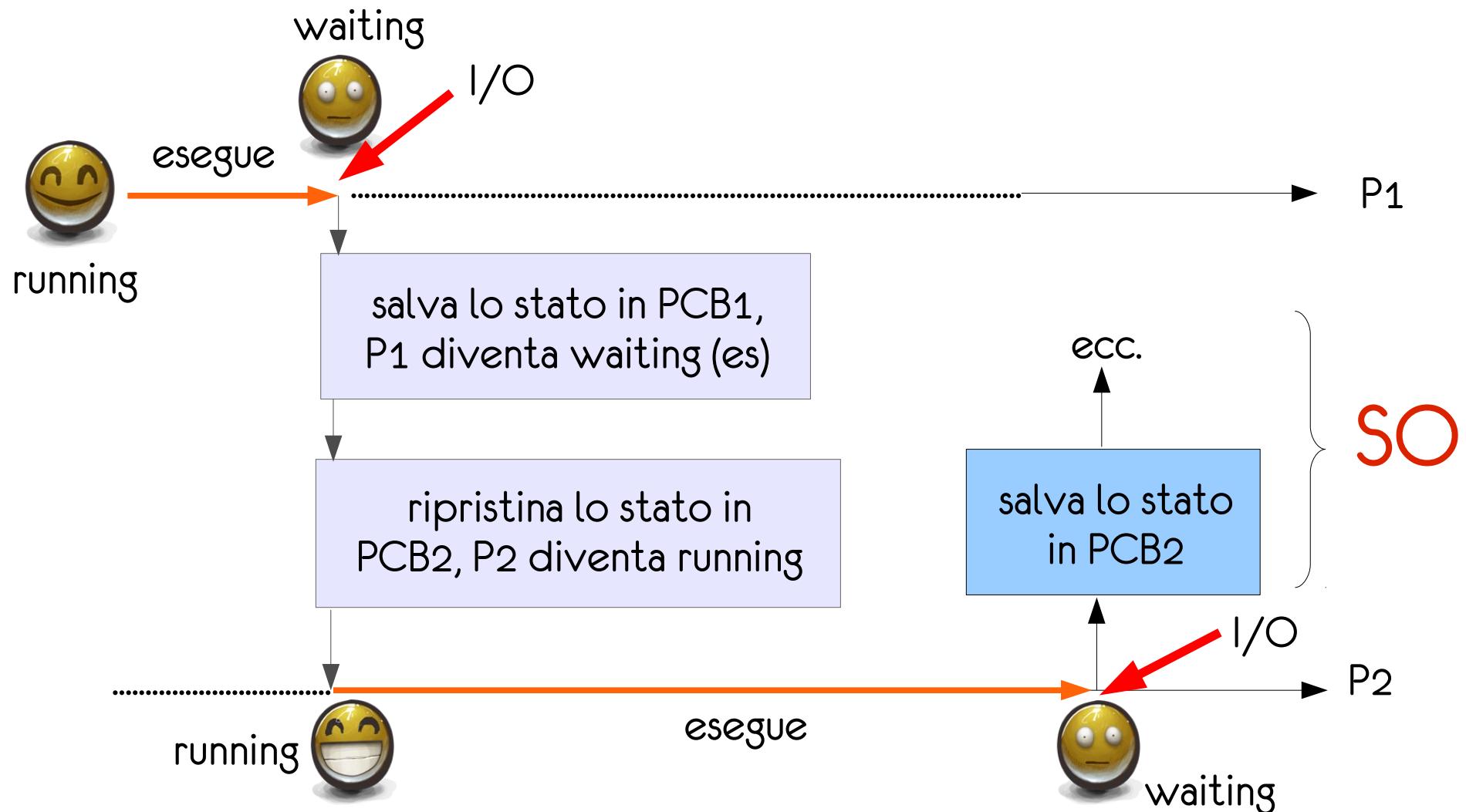
Es. commutazione della CPU

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



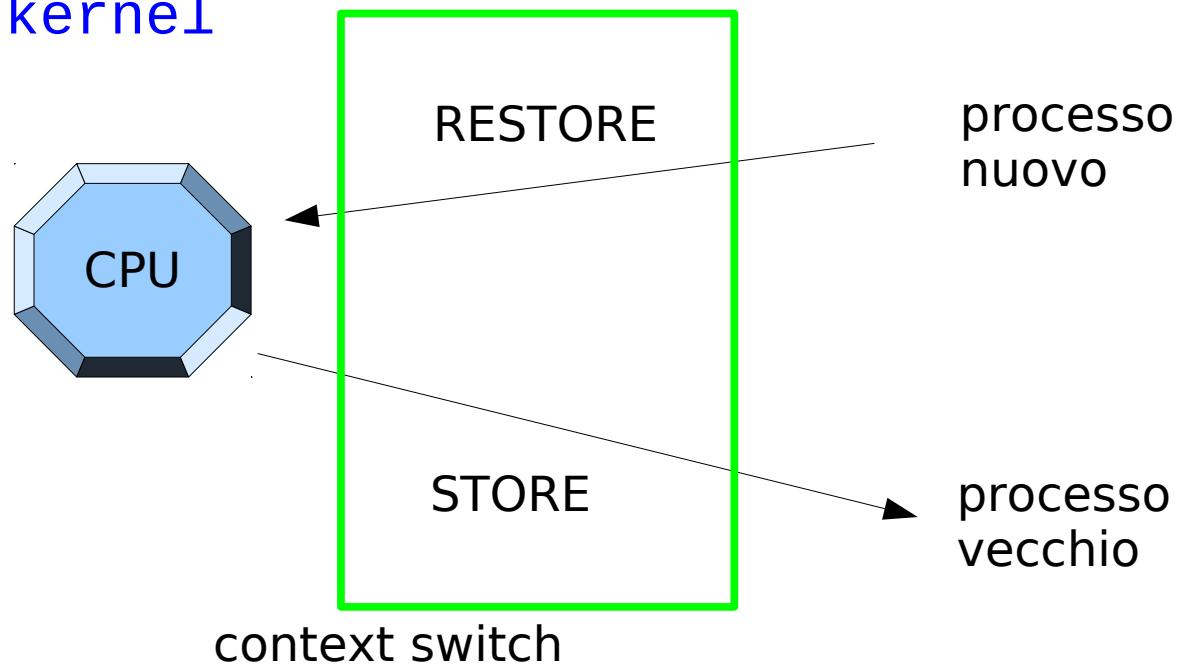
Es. commutazione della CPU

- 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

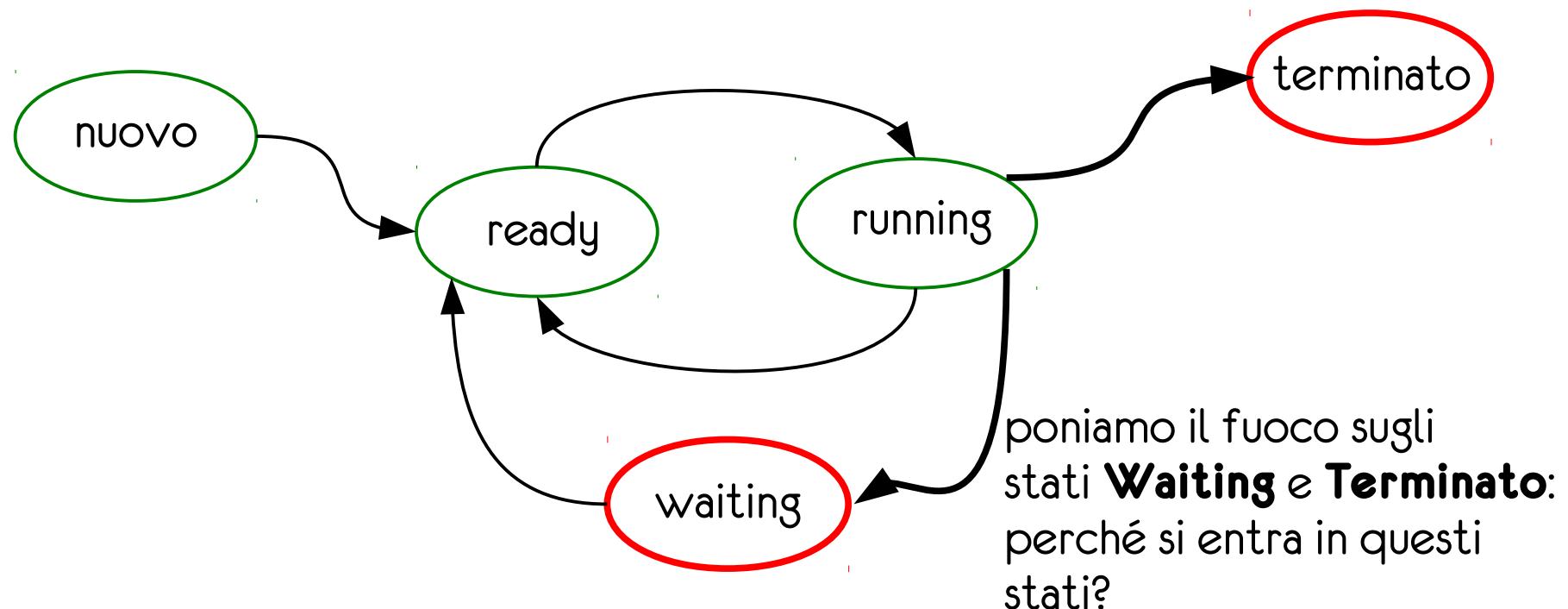
Quanto tempo dura un C.S.?

```
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$ █
```

- $815 \text{ cs / sec} \approx 1 \text{ c.s. ogni } 0.001 \approx 1 \text{ ogni millisecondo } (10^{-3} \text{ sec})$
- 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

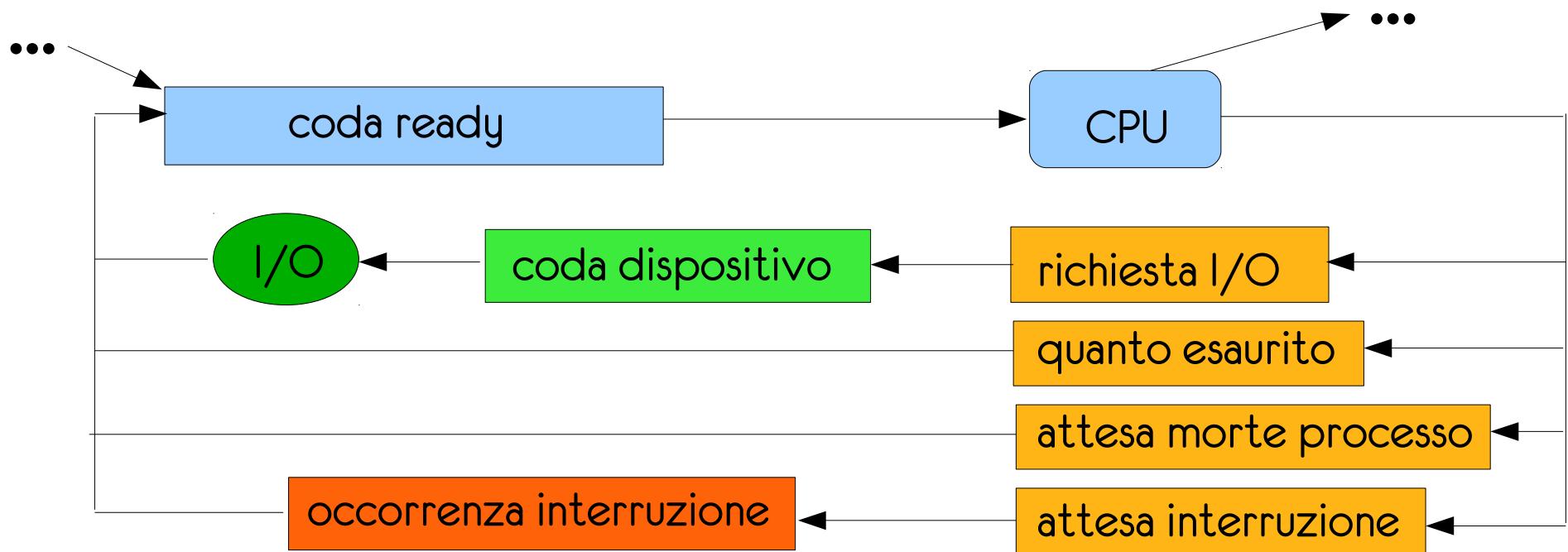
- **waiting:**
 - esecuzione di un comando di I/O
 - sospensione volontaria per un lasso di tempo
 - sospensione volontaria in attesa di un evento (es. morte di un altro processo)
 - sincronizzazione (es. attesa di un messaggio, attesa legata ad altri strumenti di sincronizzazione come i semafori)

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



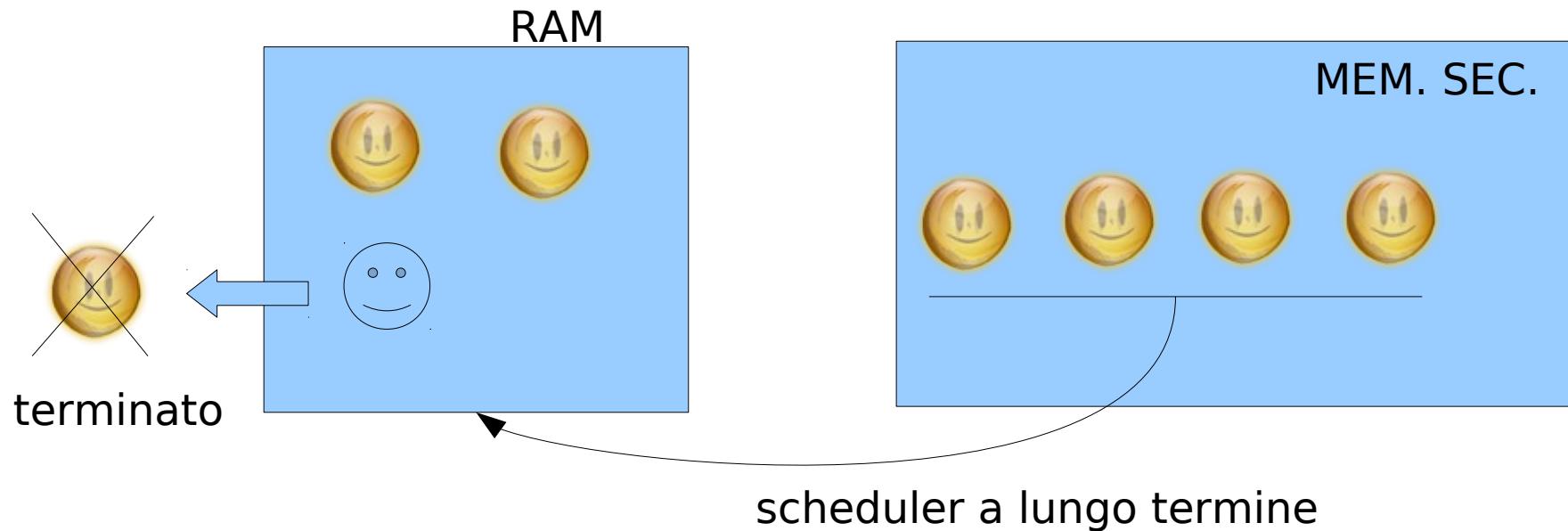
Scheduling

- **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

- **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

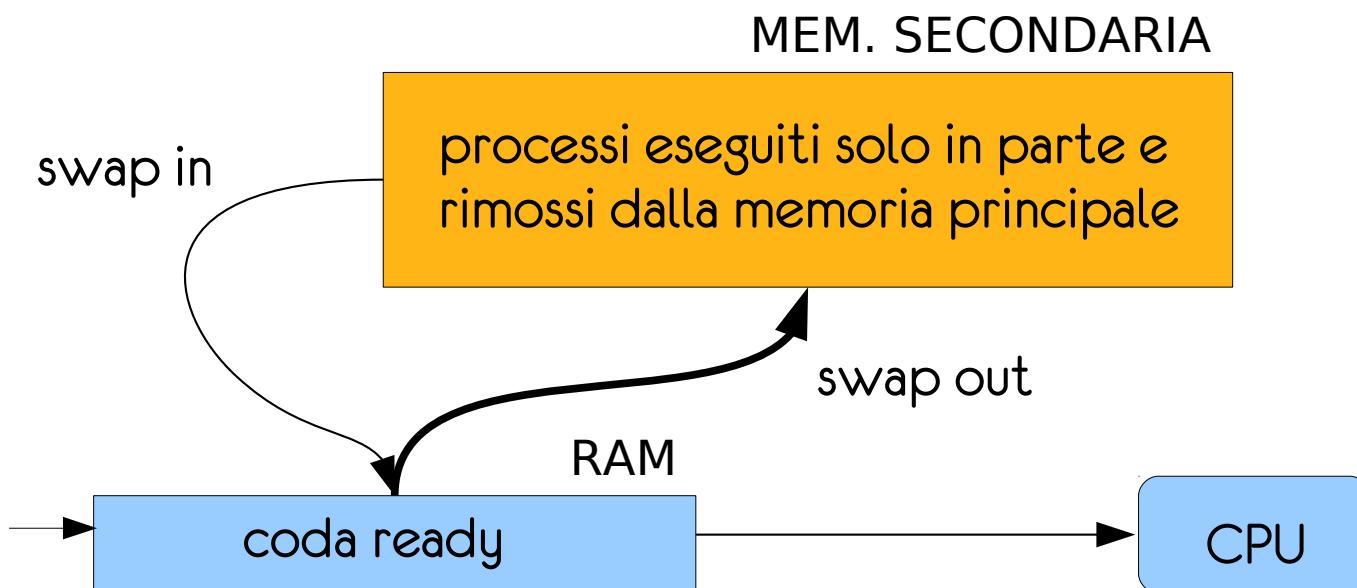
Scheduling a lungo termine



- si limita a sotituire 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**)



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

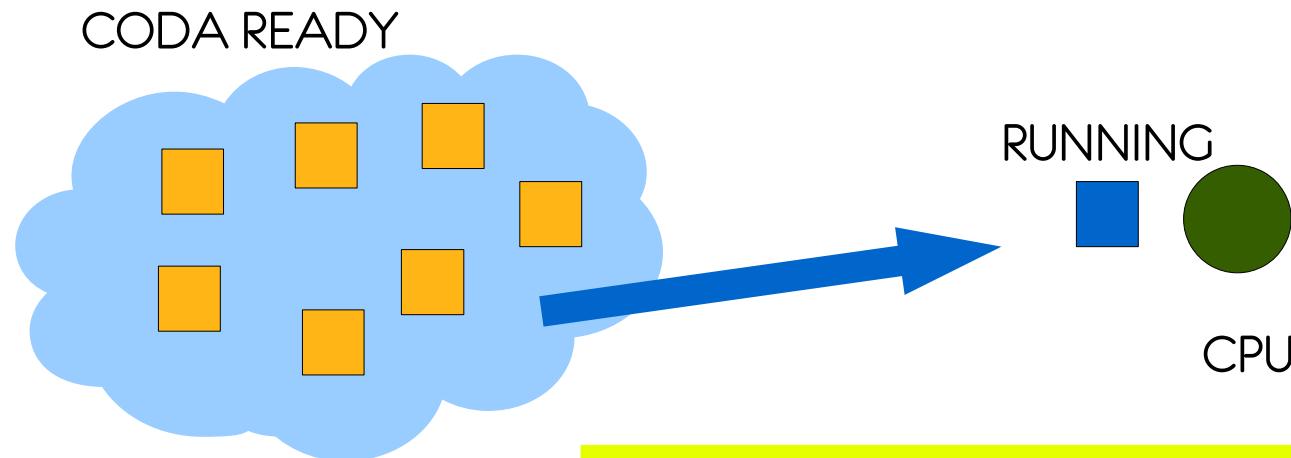
Scheduling della CPU

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

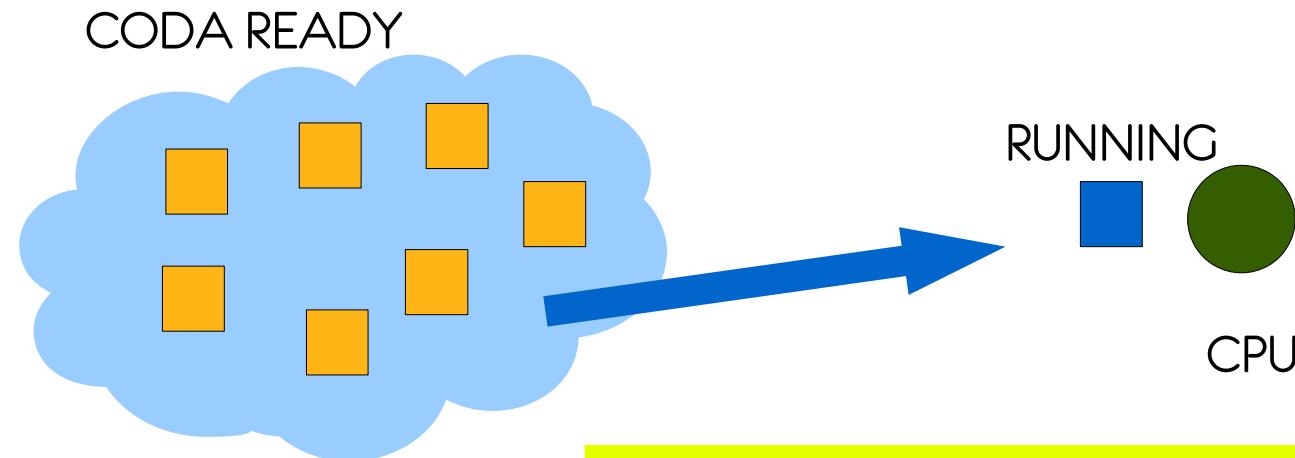
Scheduler della CPU



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
⊕ criterio di selezione ⇒ politica

Algoritmi di scheduling

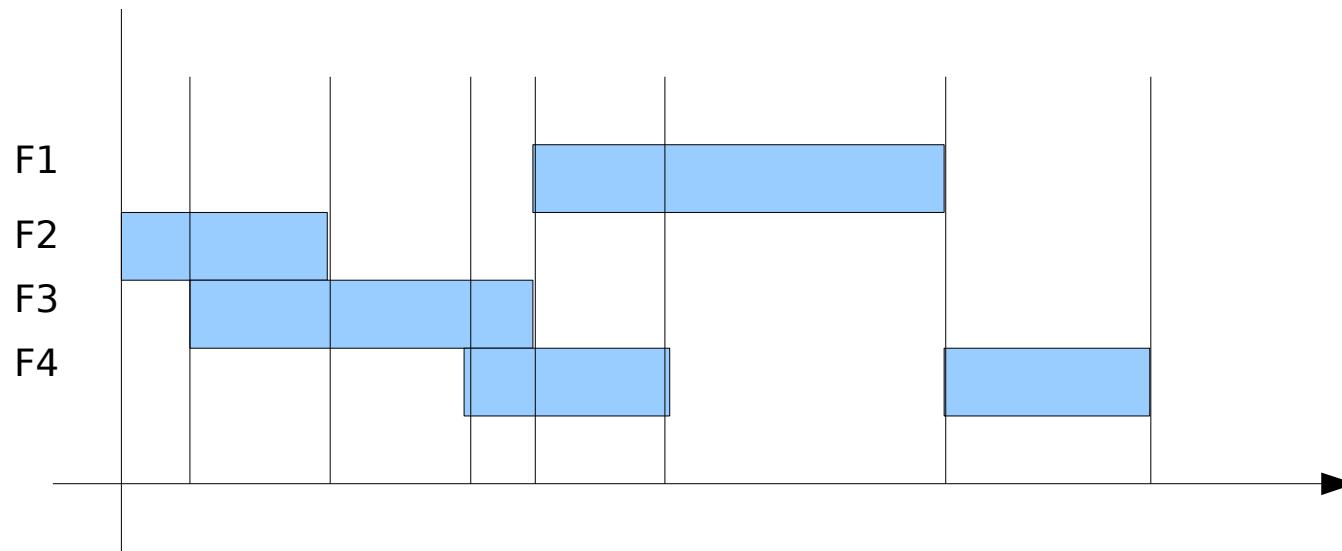
- first-come-first-served (FCFS)
- shortest jobs first (SJF)
- a priorità
- round-robin (RR)
- a code multilivello (multilevel queue scheduling)
- a code multilivello con feedback (multilevel feedback queue scheduling)

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'algoritmo 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



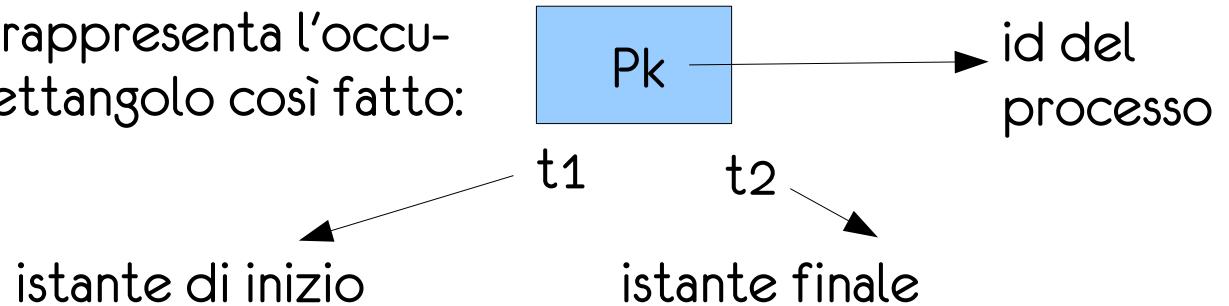
Diagrammi di Gantt

- 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

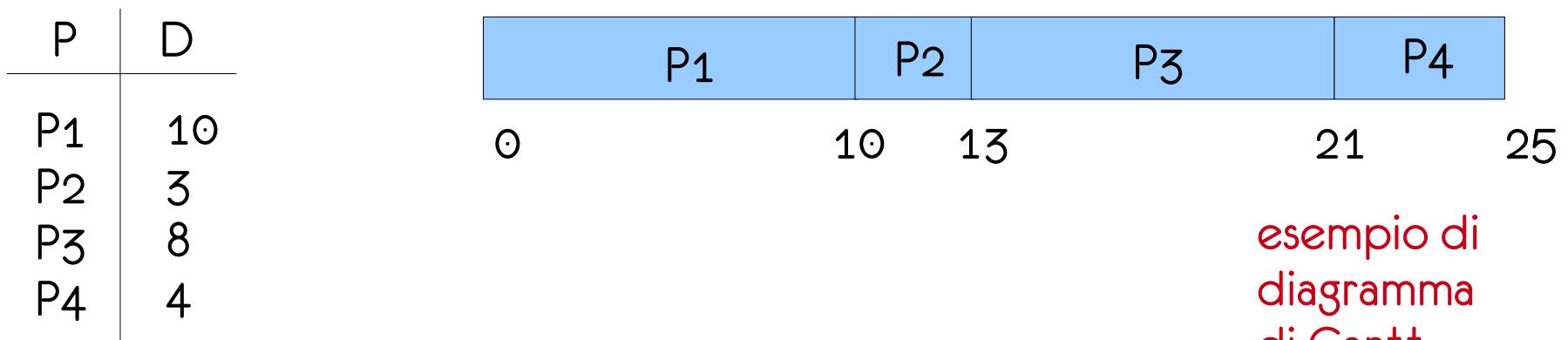


Premessa: diagrammi di Gantt

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

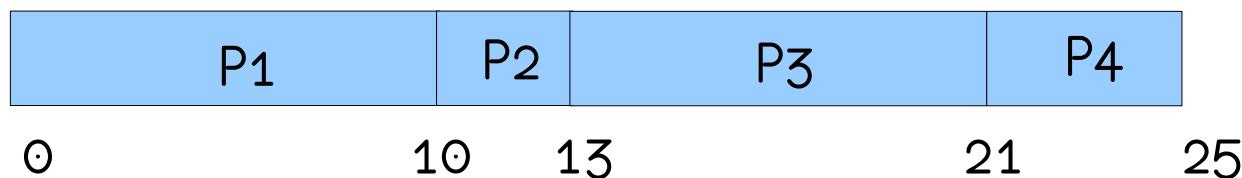


P: processo

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

First-come-first-served

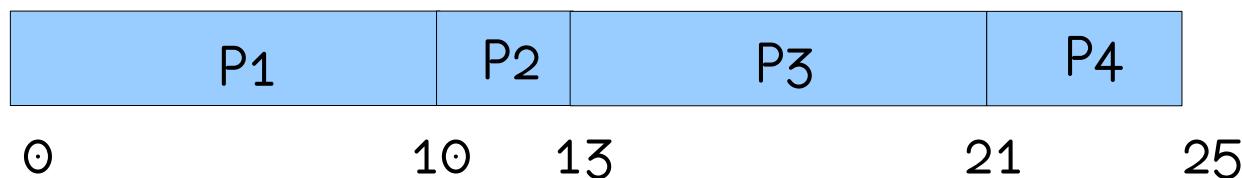
- **FCFS**: PCB organizzati in una coda FIFO, non è preemptive, il primo processo arrivato è il primo ad avere la CPU, la mantiene per un intero CPU burst
- il tempo medio di attesa è abbastanza lungo, es:



- t.m.a. = ???

First-come-first-served

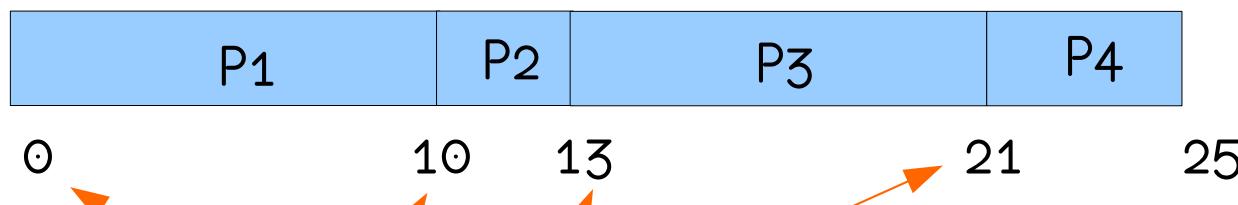
- **FCFS**: PCB organizzati in una coda FIFO, non è preemptive, il primo processo arrivato è il primo ad avere la CPU, la mantiene per un intero CPU burst
- il tempo medio di attesa è abbastanza lungo, es:



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

First-come-first-served

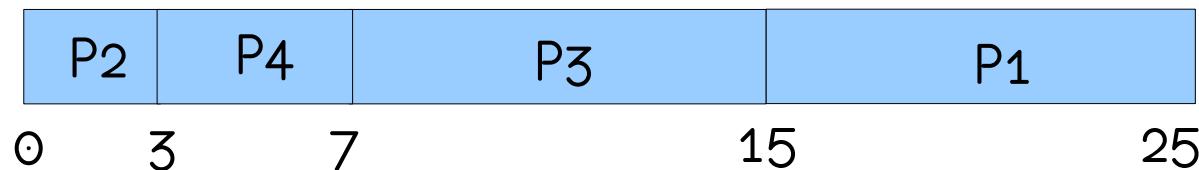
- **FCFS**: PCB organizzati in una coda FIFO, non è preemptive, il primo processo arrivato è il primo ad avere la CPU, la mantiene per un intero CPU burst
- il tempo medio di attesa è abbastanza lungo, es:



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

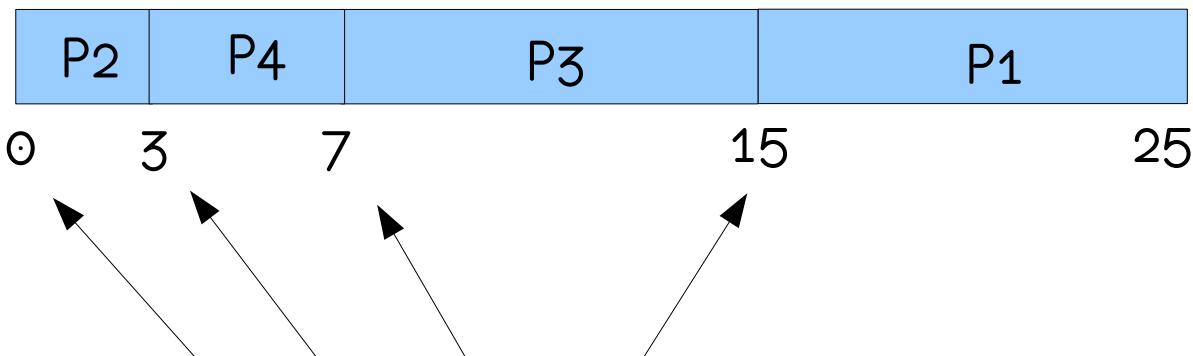
First-come-first-served

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



First-come-first-served

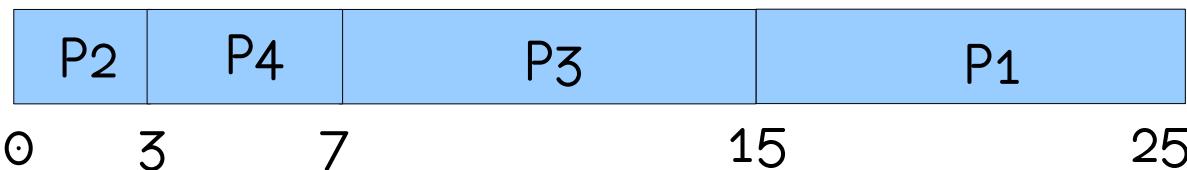
- SÌ:



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

First-come-first-served

- **sì:**



- $tma = (0 + 3 + 7 + 15) / 4 = 25/4 = 6,25$ 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 preeemptive** (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

Scheduling a Priorità

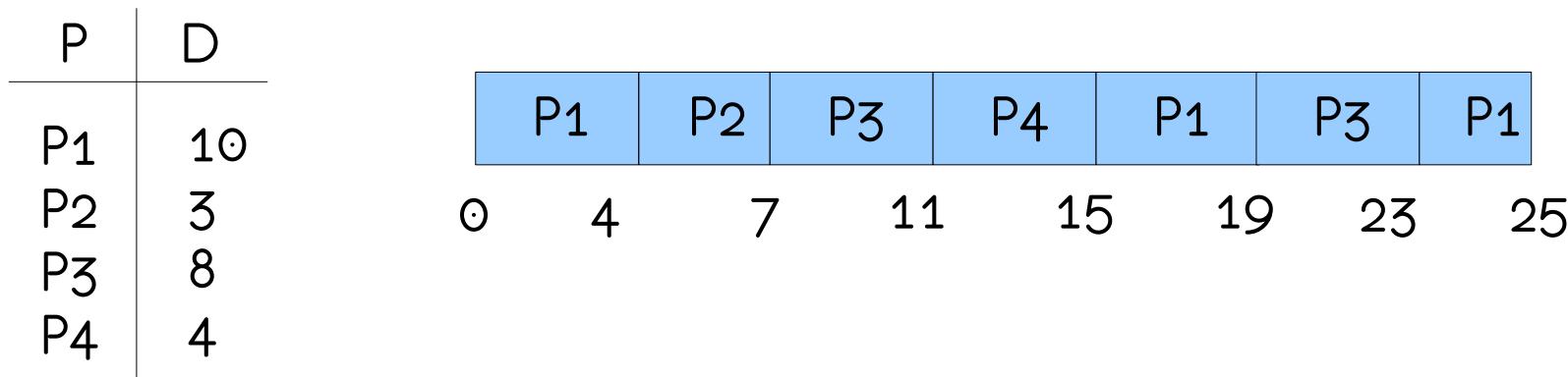
- **Ogni processo ha associata una priorità che decide l'ordine di assegnazione della CPU. Può essere con o senza prelazione**
- La priorità può essere definita:
 - **Internamente**: sulla base di caratteristiche del processo. Es. SJF è un algoritmo di scheduling basato su priorità definita internamente,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

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

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:



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		P1	P2	P3	P4	P1
P2	3	0	4	7	11	15	19
P3	8						23
P4	4						25

$tma = (0 + 4 + 7 + 11 + 11 + 8 + 4) / 4 = 45 / 4 = 11,25$

Round-robin

- il tma è abbastanza alto ma **non si ha starvation**: ogni processo viene servito ogni **(n_proc - 1) * 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

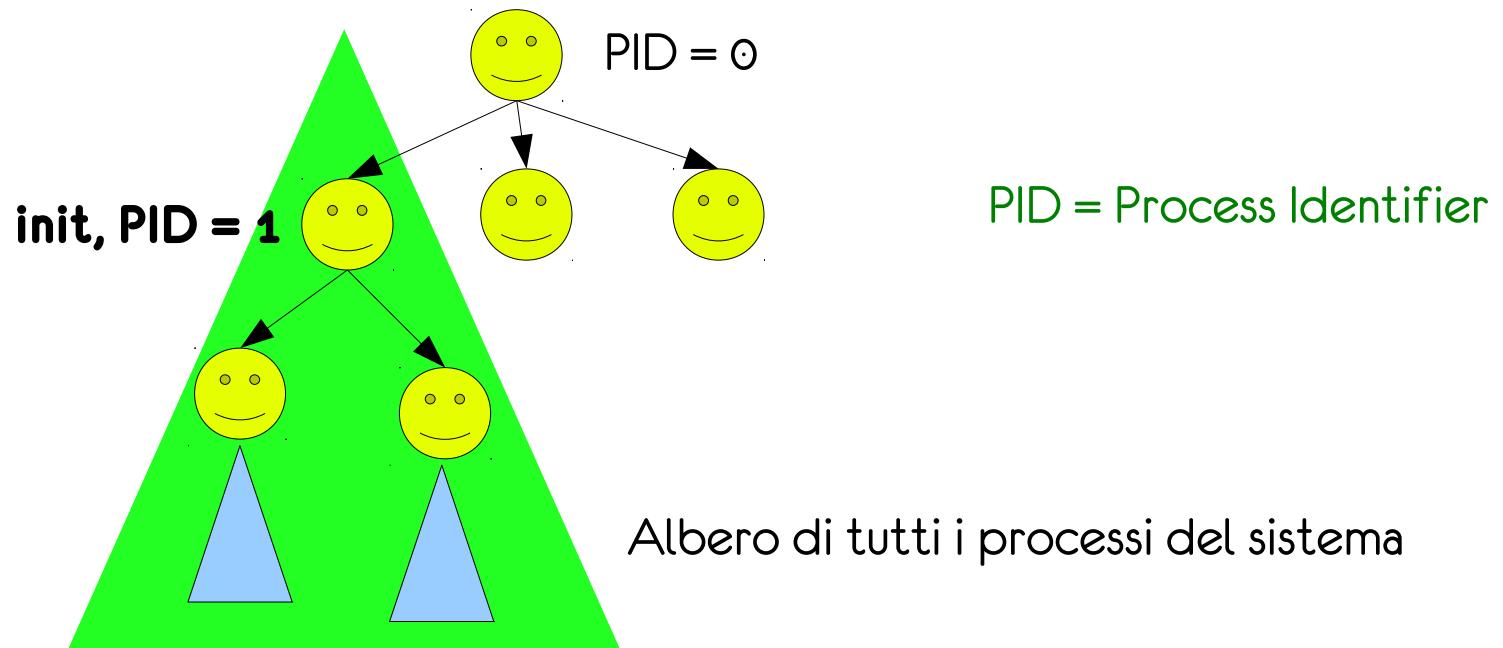
- Algoritmi utili quando è possibile distinguere i processi in categorie legate alla loro natura
- La ready queue è suddivisa in tante code quante sono le categorie
- Ogni coda può avere un algoritmo di scheduling diverso
- Esiste una priorità fra le code (eventualmente associata a meccanismi di invecchiamento 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 SO: **un altro processo**
- con l'avvio del SO si genera un **albero di processi** che cresce e decresce dinamicamente con l'evoluzione dell'elaborazione



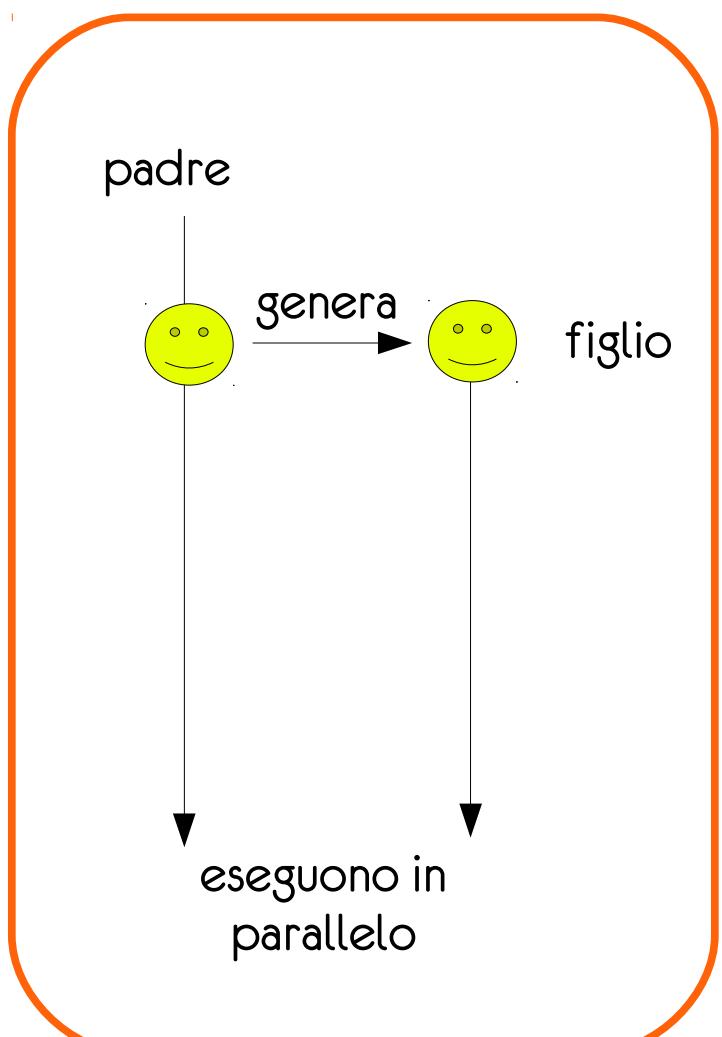
Creazione

- 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

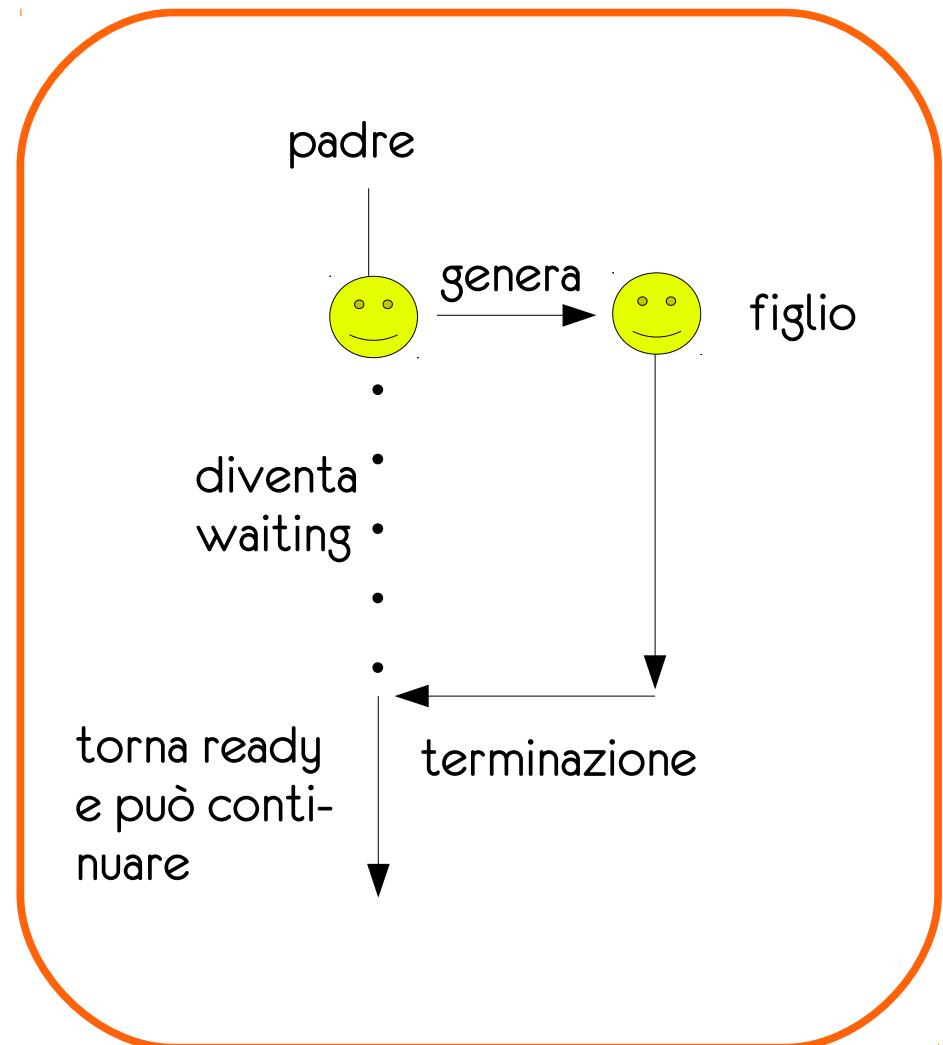
Creazione

- **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 tramita una system call (**exec**)

Creazione: fork



CASO A



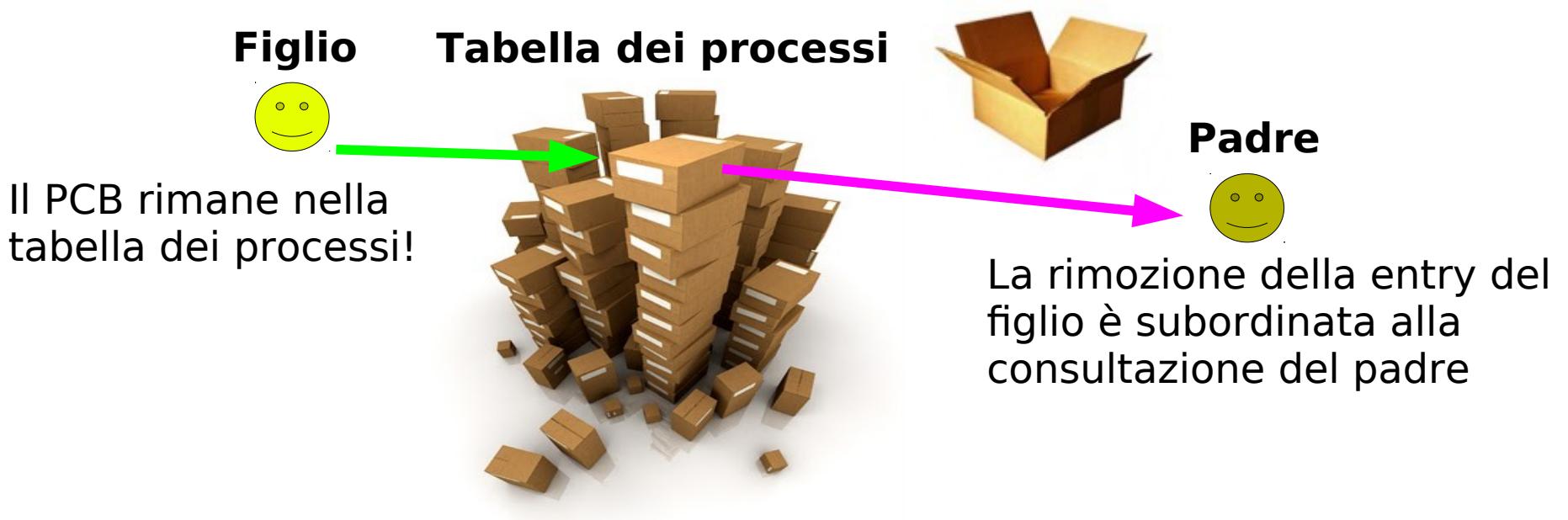
CASO B

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

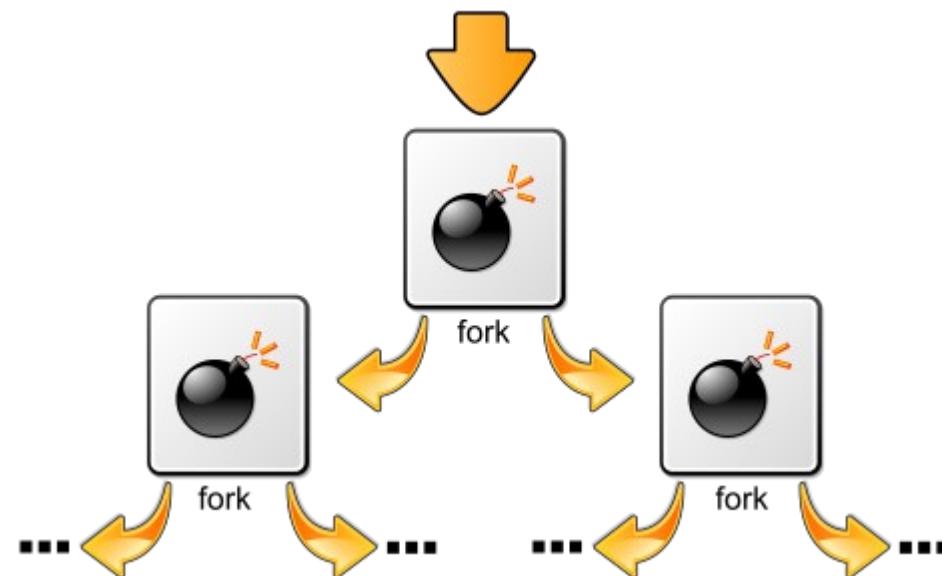
Problemi

- ... 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.



http://en.wikipedia.org/wiki/File:Fork_bomb.svg

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)

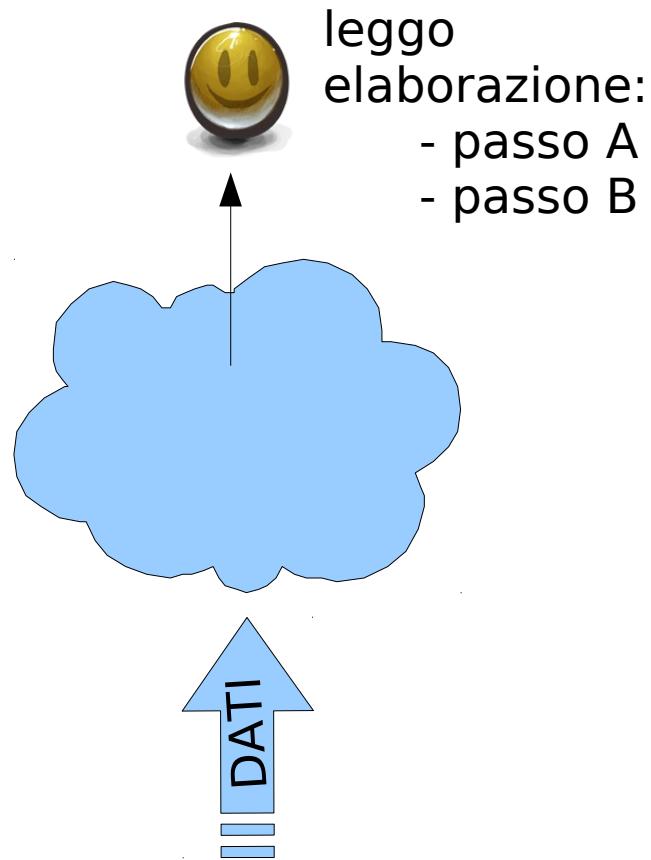
modelli di comunicazione

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

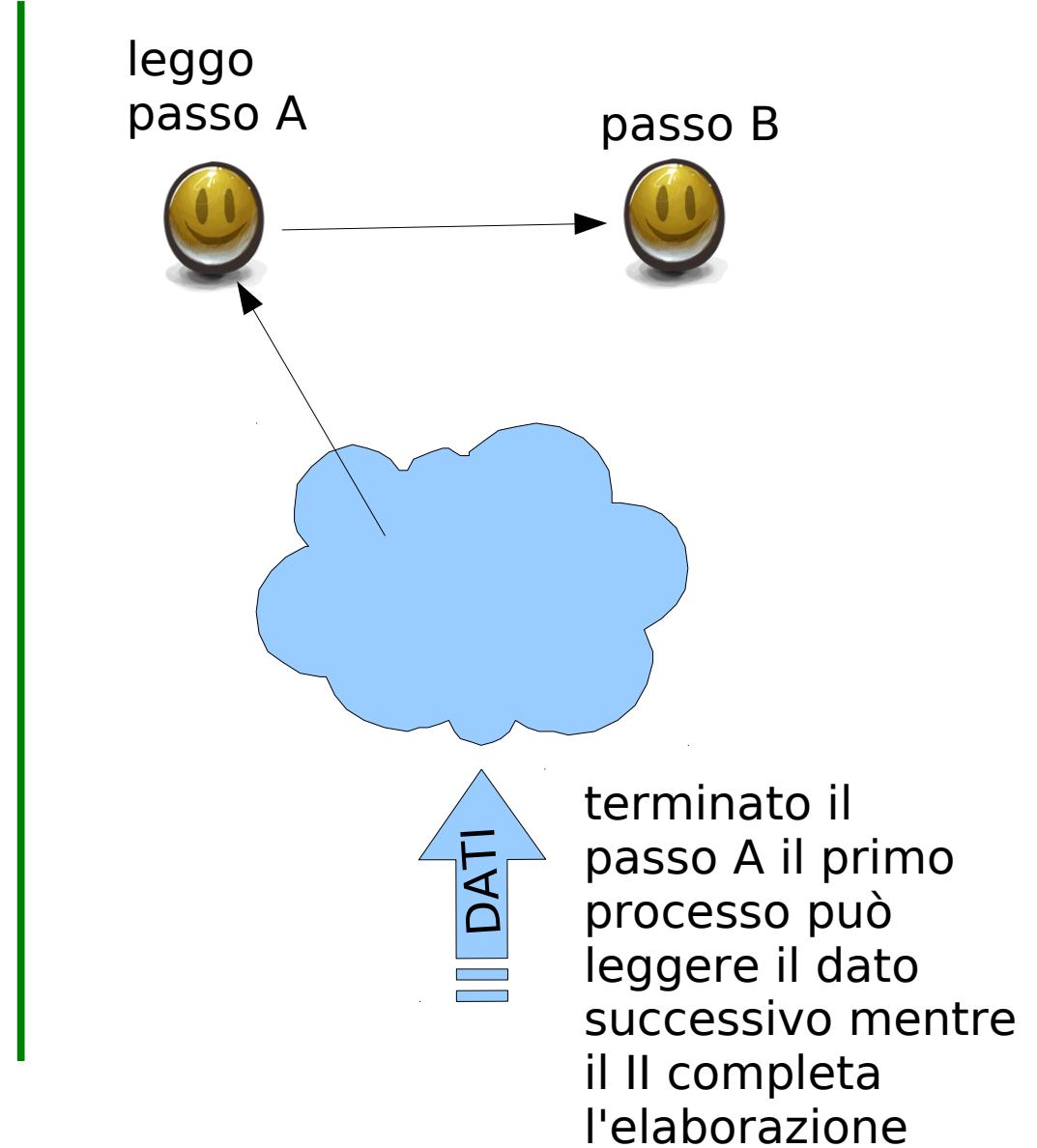
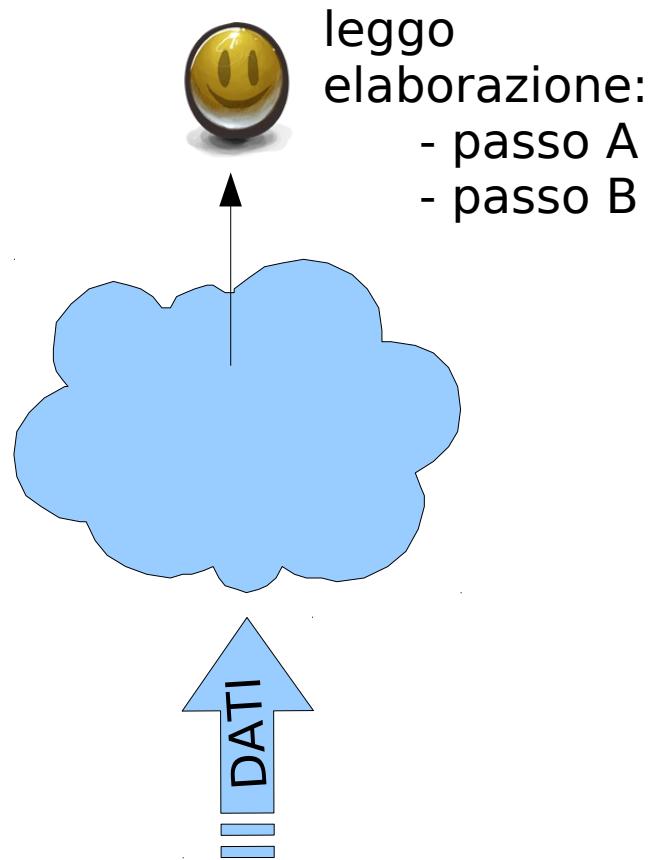
Processi cooperanti

- 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:
 - **maggior 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

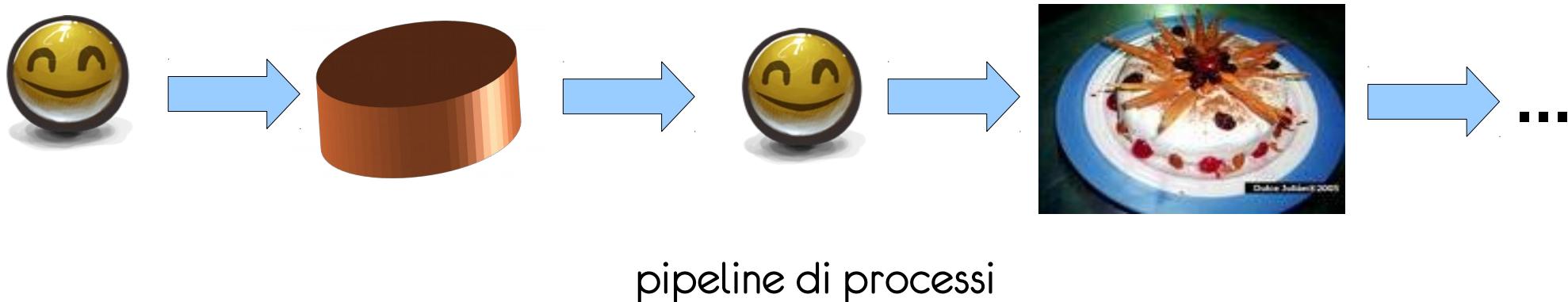


Processi cooperanti



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

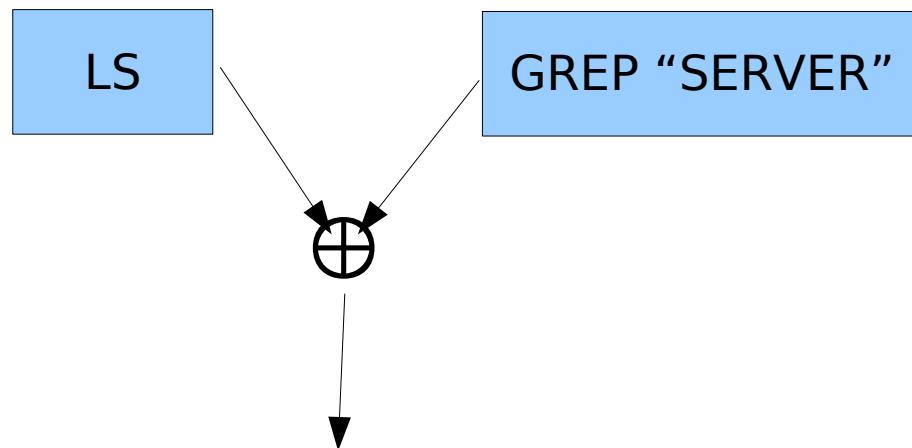


come fanno i vari processi a passarsi i dati semi-lavorati?

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
```

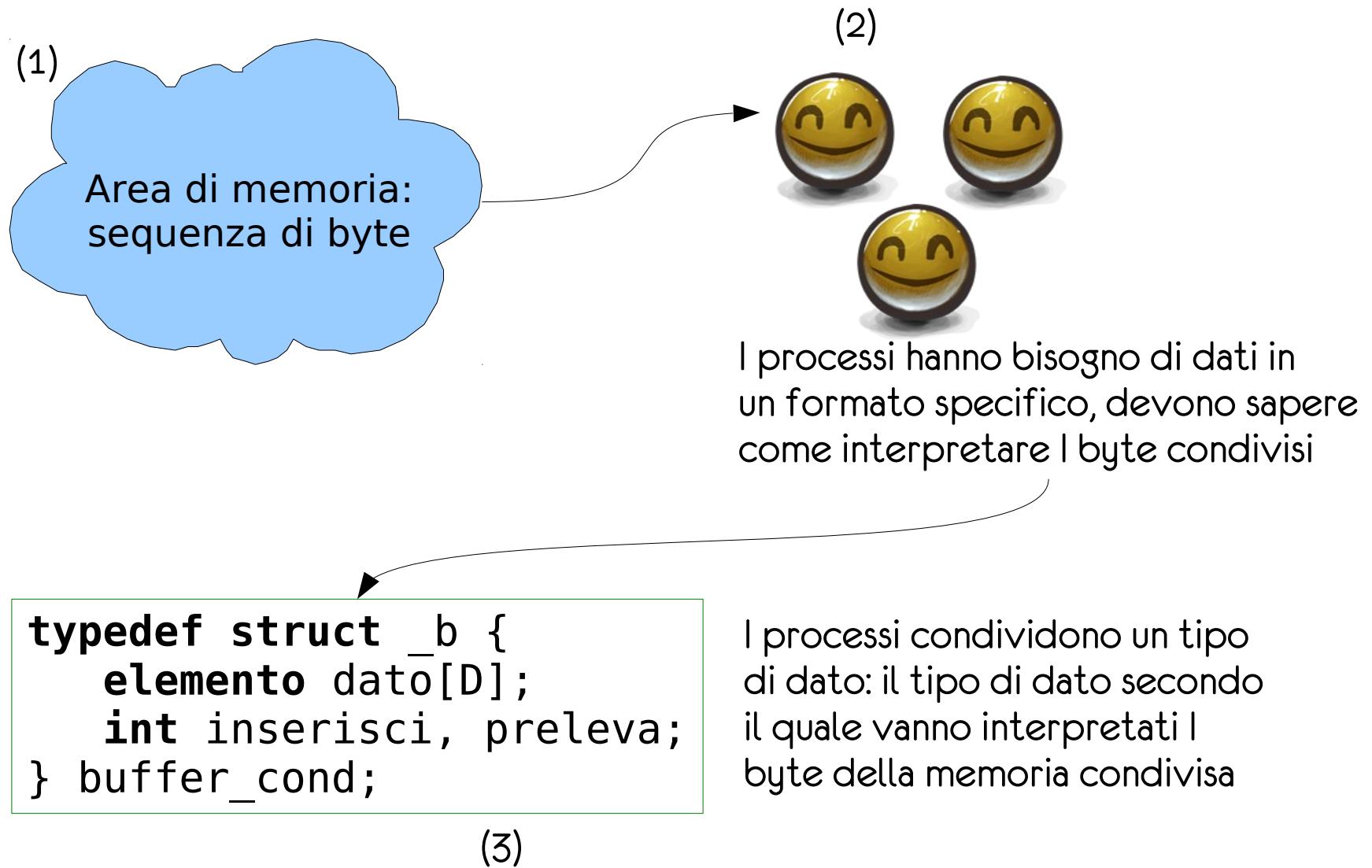
```
typedef ... elemento;
```

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

spazio per i dati
condivisi

entrambi inizializzati a 0

Memoria condivisa



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

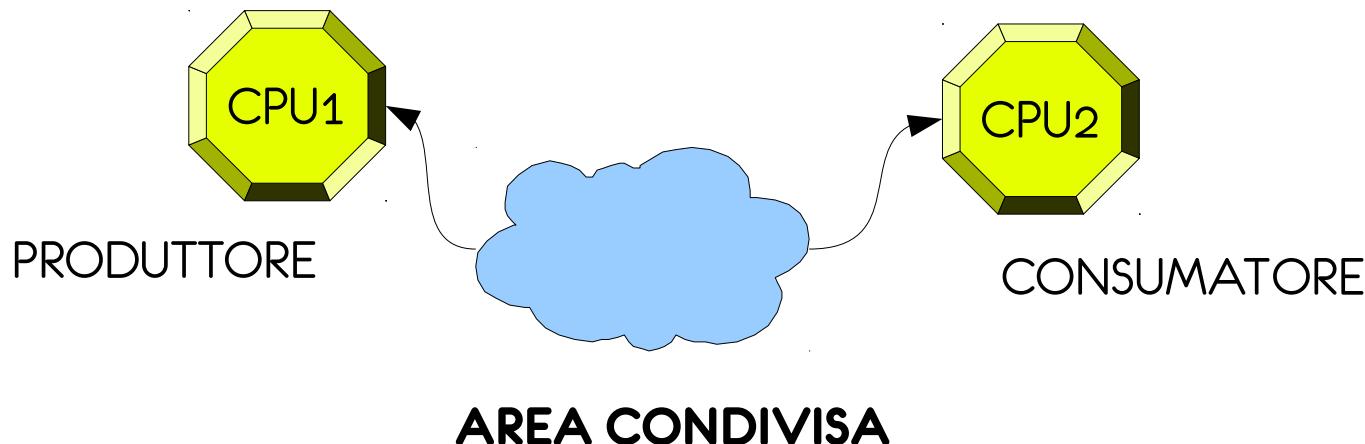
- Gli accessi vanno controllati per evitare inconsistenze!!!

```
 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!!!

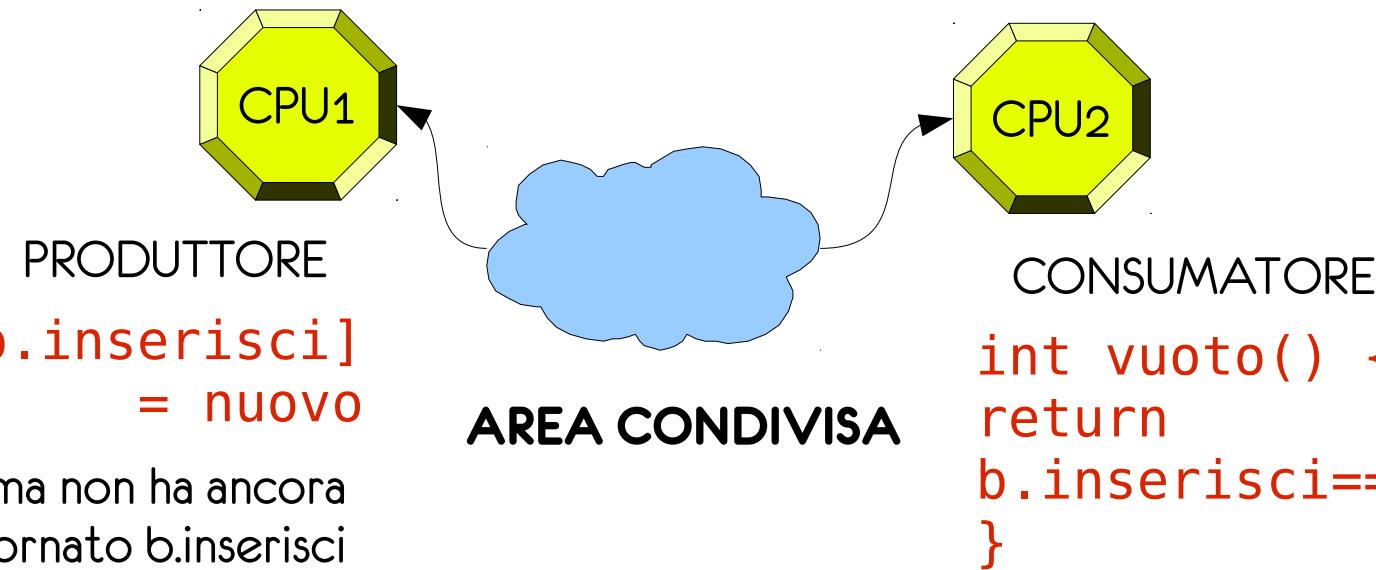
Inconsistenza dei dati

- 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 essere 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 ricevuto (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:



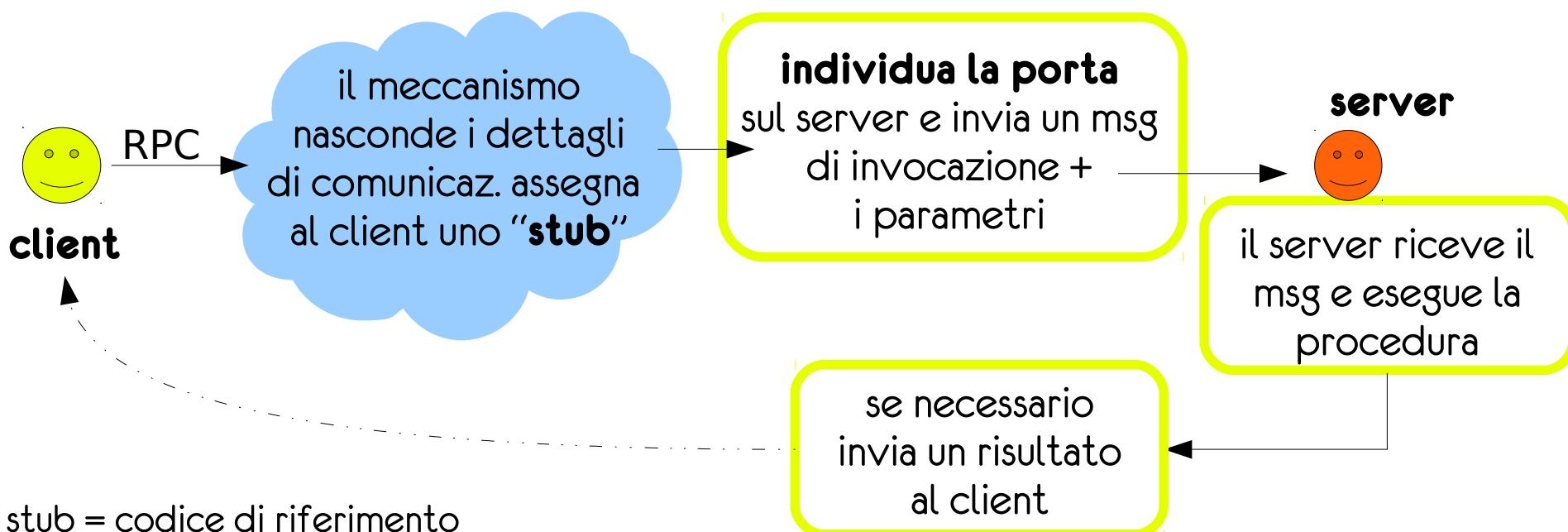
- Esempio di id: **140.228.112:80** la parte in blu (140.228.112) è l'IP di una macchina, quella in rosso (80) un numero di porta, sono concatenati da un due punti.
- I messaggi sono semplici pacchetti dati non strutturati

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

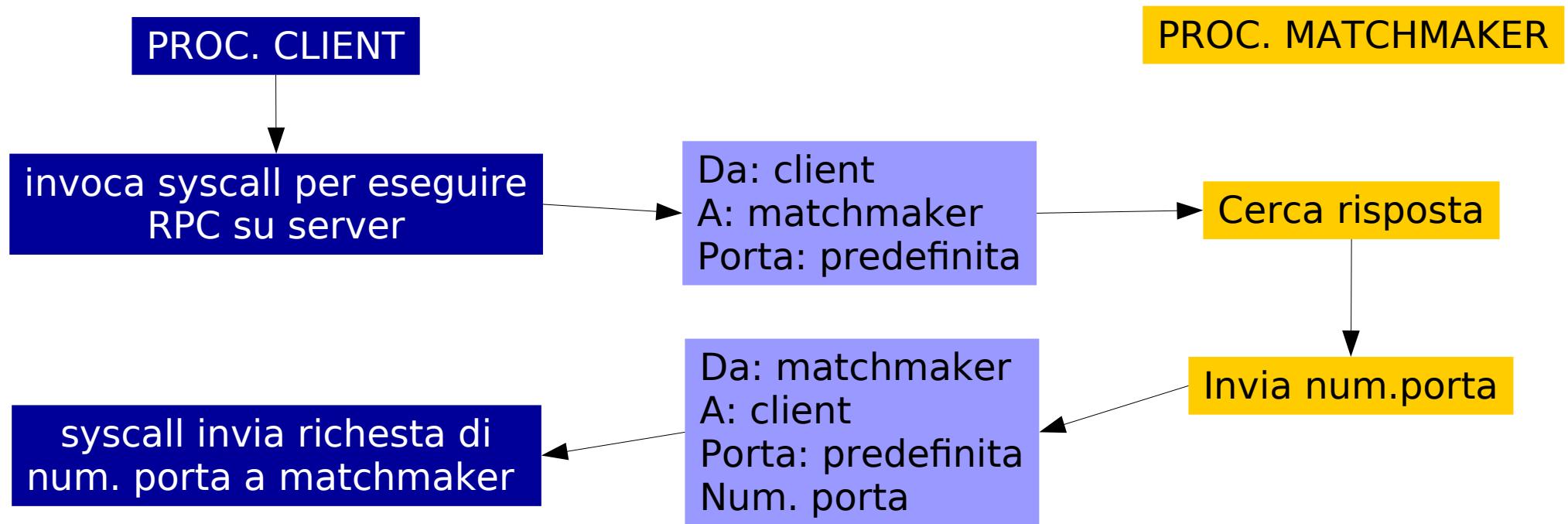
- meccanismo usato in sistemi client-server
- in certi casi è utile consentire a un processo di invocare l'esecuzione di una procedura che risiede su di un'altra macchina connessa in rete: meccanismo noto come **remote procedure call** (RPC)
- i messaggi sono ben strutturati, sono costituiti dall'identificatore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



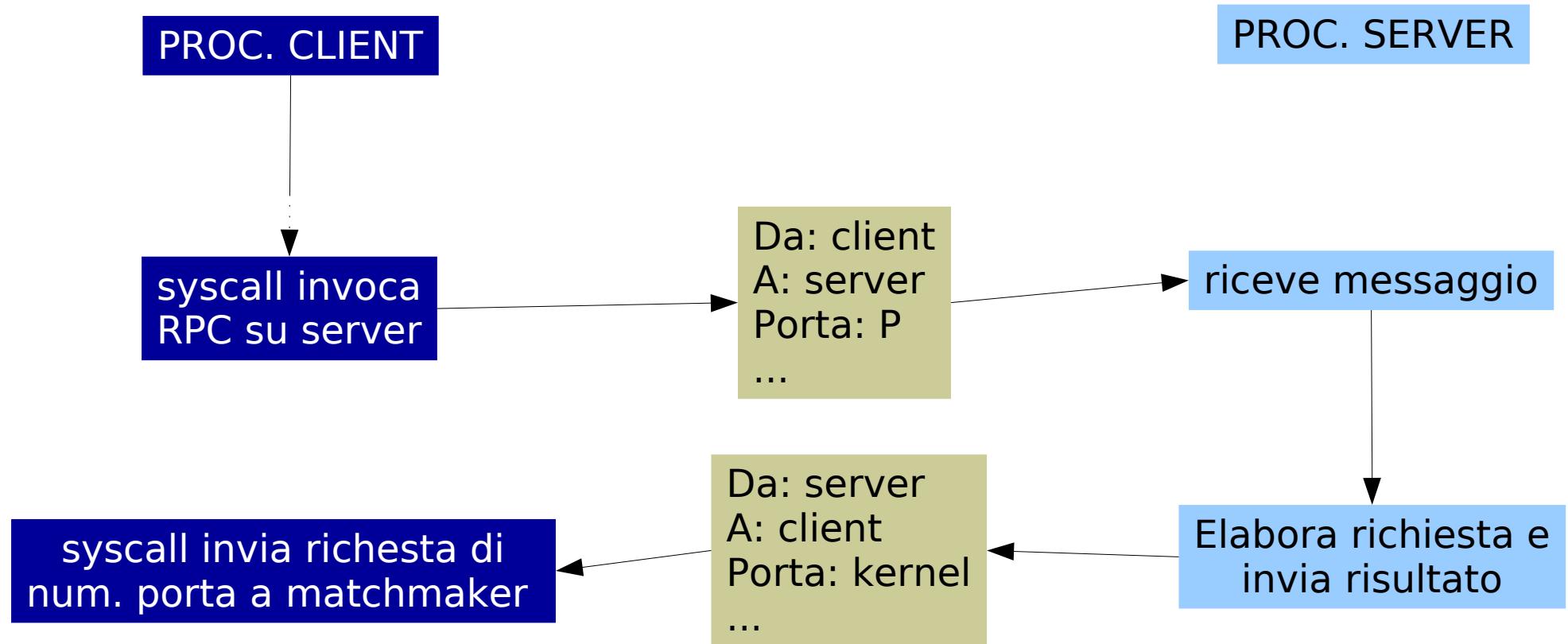
Associazione porte a RPC

- 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



PROCESSO PRODUTTORE

PROCESSO CONSUMATORE

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.)

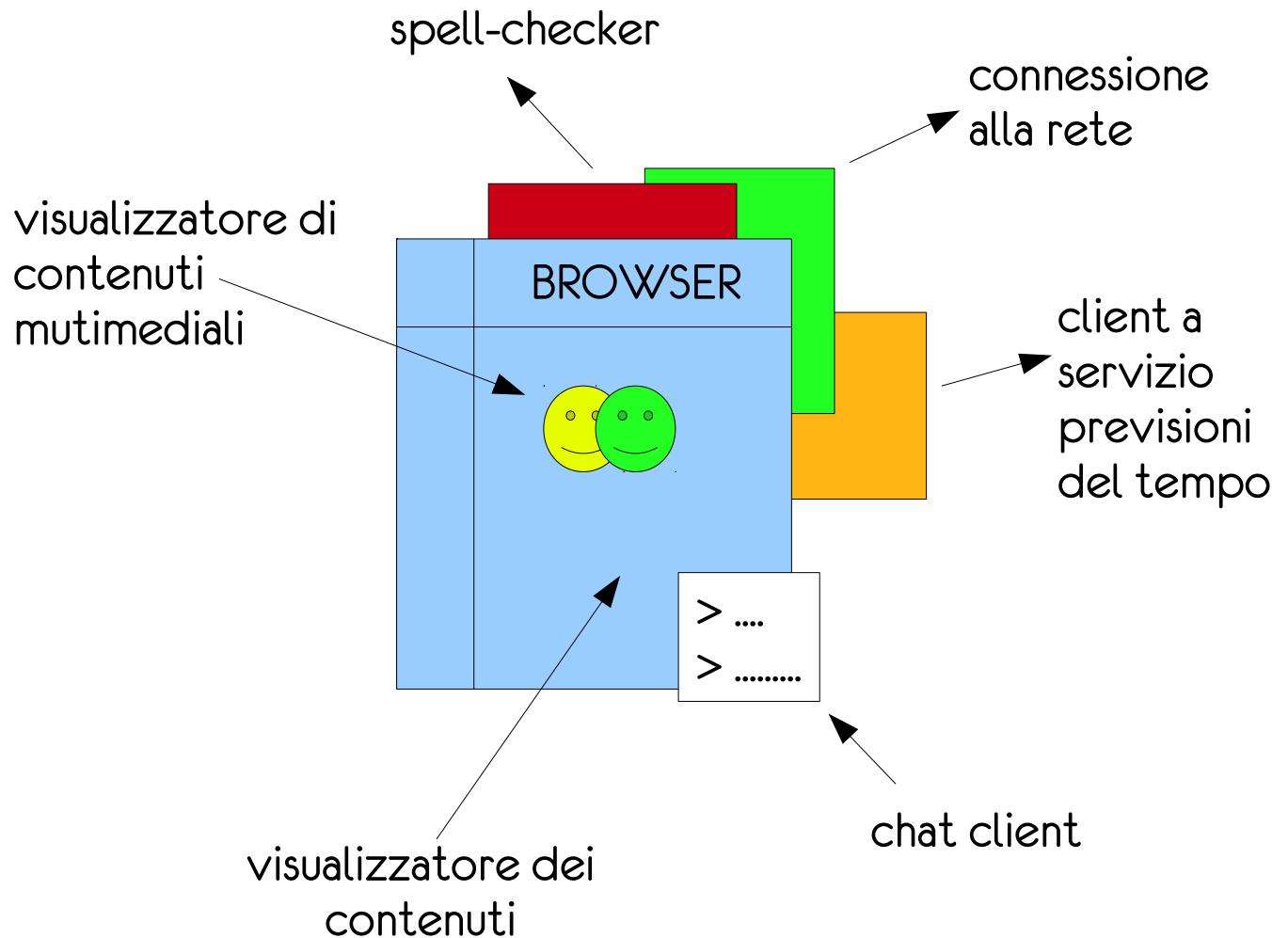
+

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

Introduzione

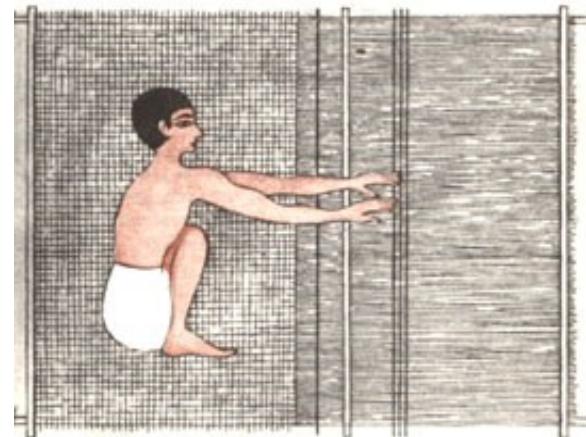
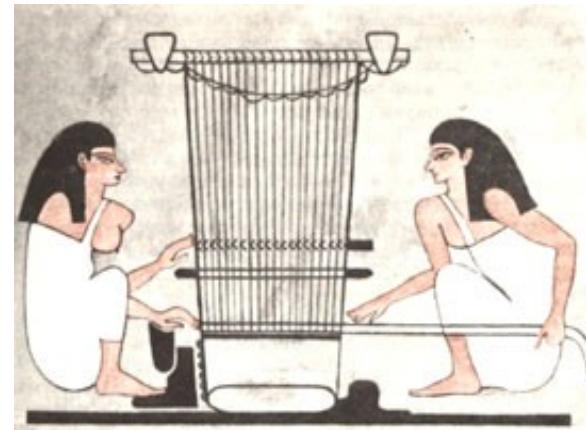
io avvio il programma
con un click oppure
digitando il nome del
browser in una shell ...

**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 f_1 e f_2 non hanno dipendenze, eseguire l'una prima dell'altra non fa differenza. Allora è anche possibile pensare a una loro esecuzione parallela ...

f_1 e f_2 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

id1
codice1
dati1
altre risorse1
stack1
registri1



id2
codice2
dati2
altre risorse2
stack2
registri2



codice

dati

altre risorse

id1
stack1
registri1



id2
stack2
registri2

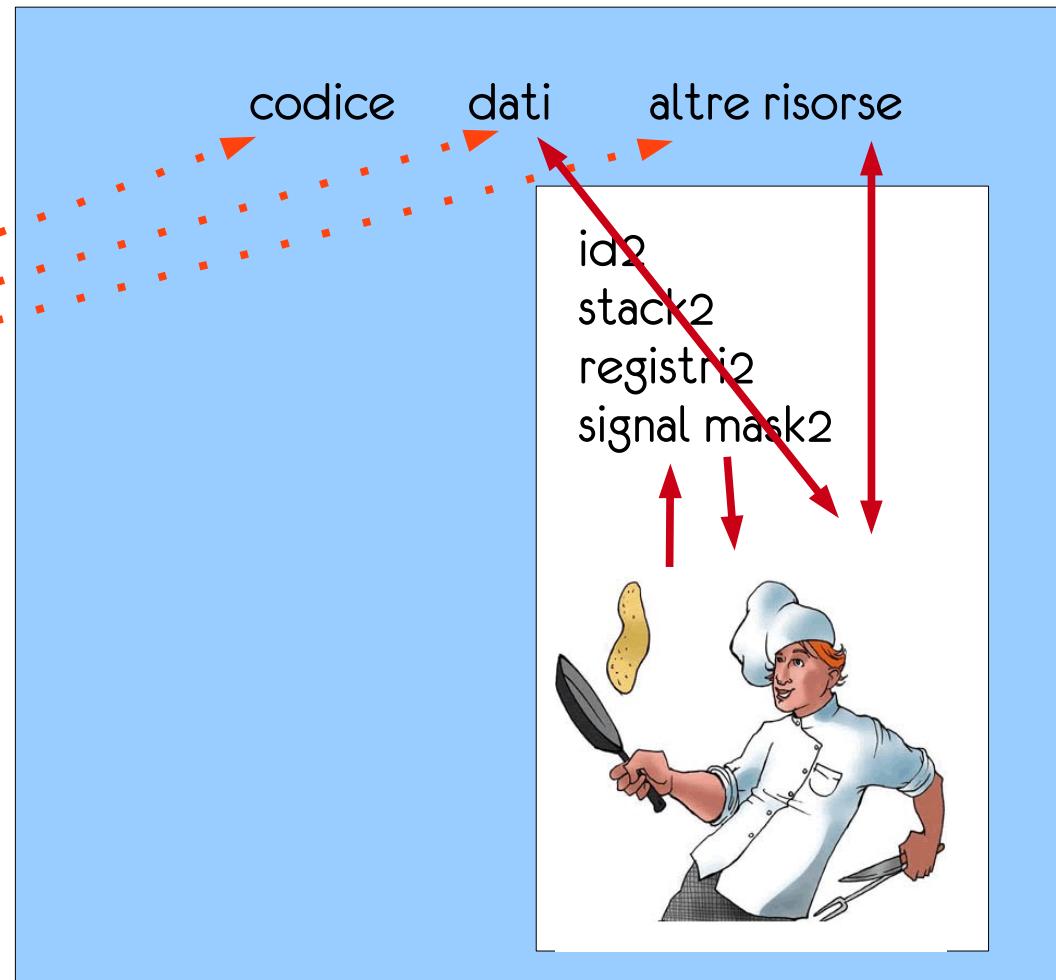
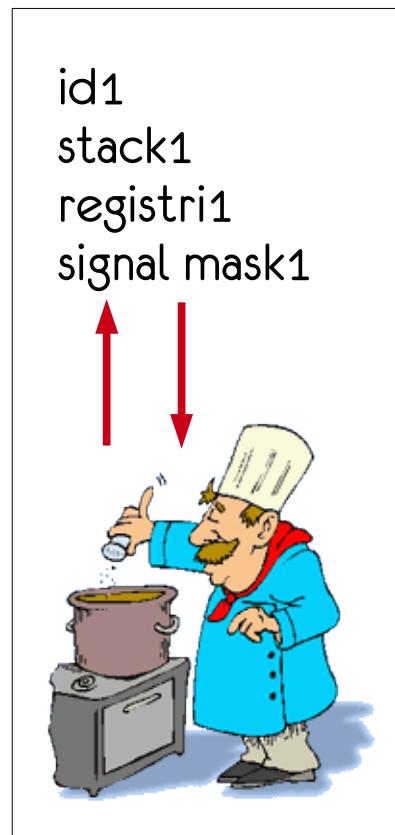


processi

thread di uno stesso processo

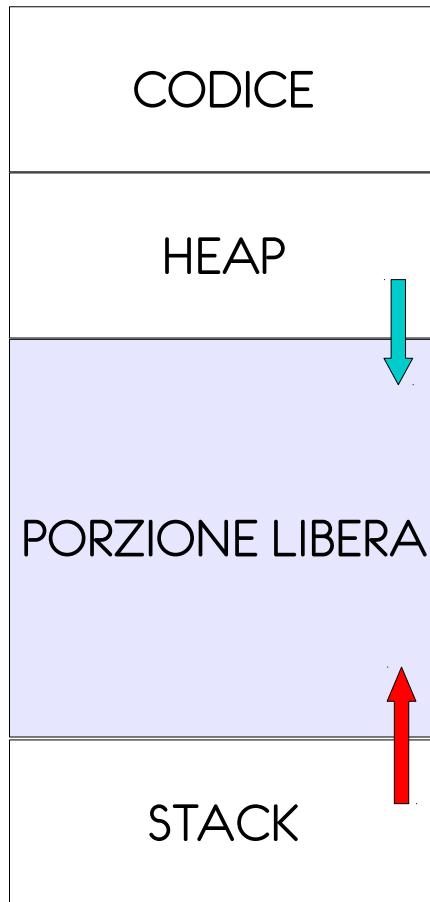
comunicano tramite un meccanismo a memoria condivisa!!

Thread senza processi?

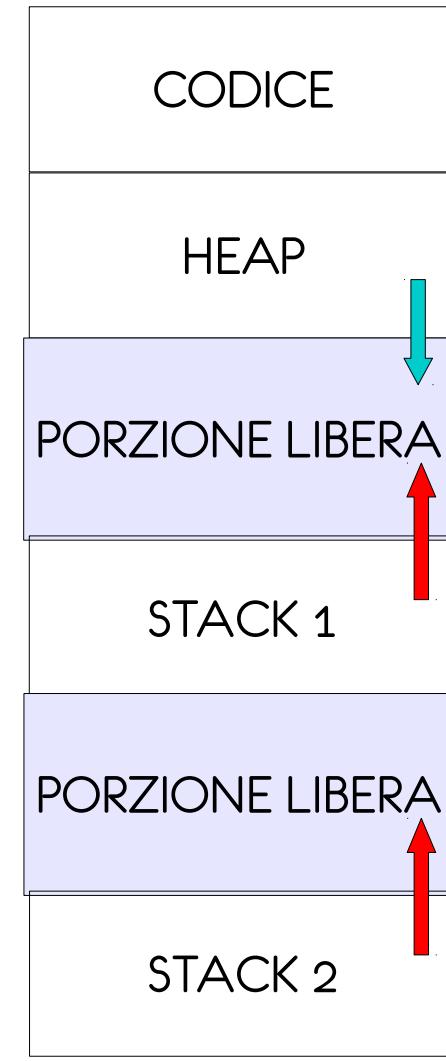


un thread non può esistere al di fuori di un processo perché non contiene tutte le informazioni necessarie per poter effettuare l'esecuzione: codice, dati, risorse fanno parte del processo

Organizzazione della memoria



Processo monolitico



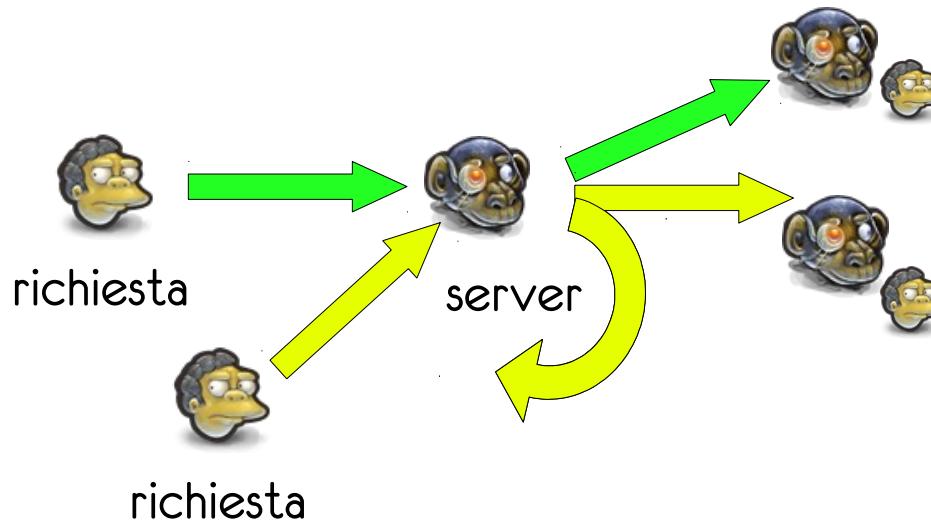
Processo con due thread

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@rhialto: ~
File Edit View Search Terminal Tabs Help
alpine      x  baroglio@rhialto: ~  x  baroglio@rhialto: ~  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 dleyna-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
```

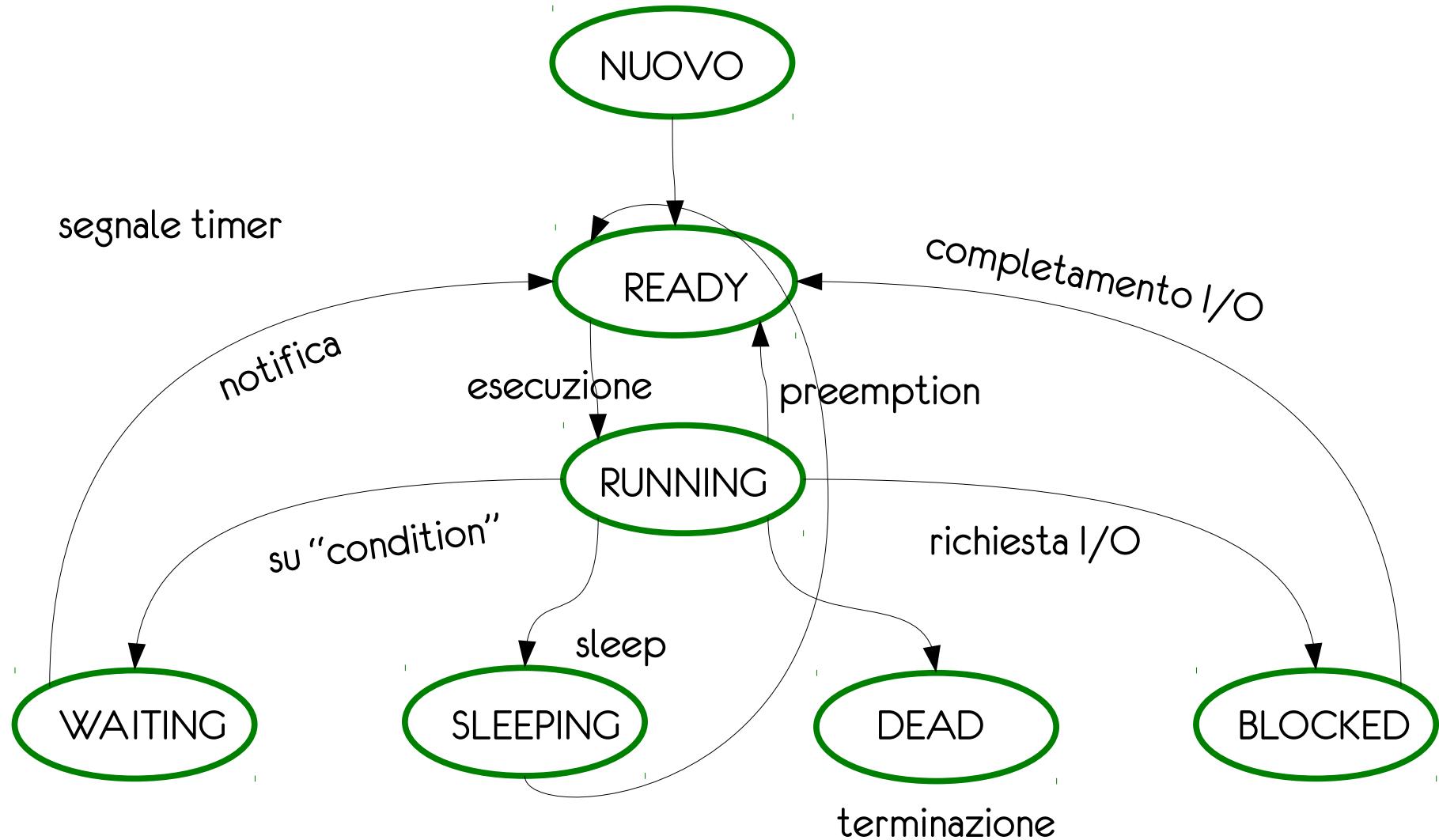
fork exec e thread

- 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!

Diagramma degli stati



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 I thread, comporta l'attesa da parte di un thread della terminazione di un altro

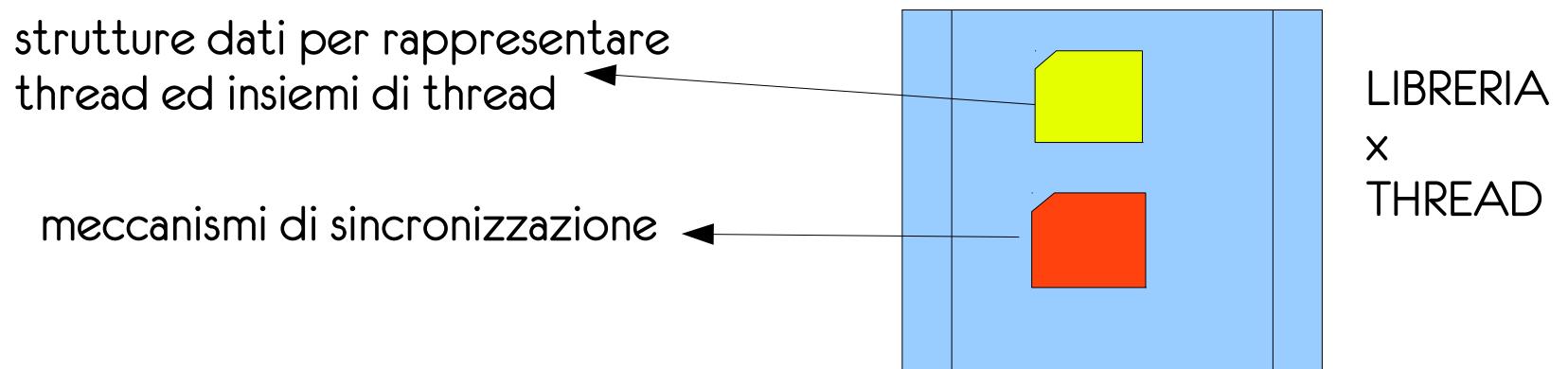
T1 si blocca sulla join fino a quando T2 non termina

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

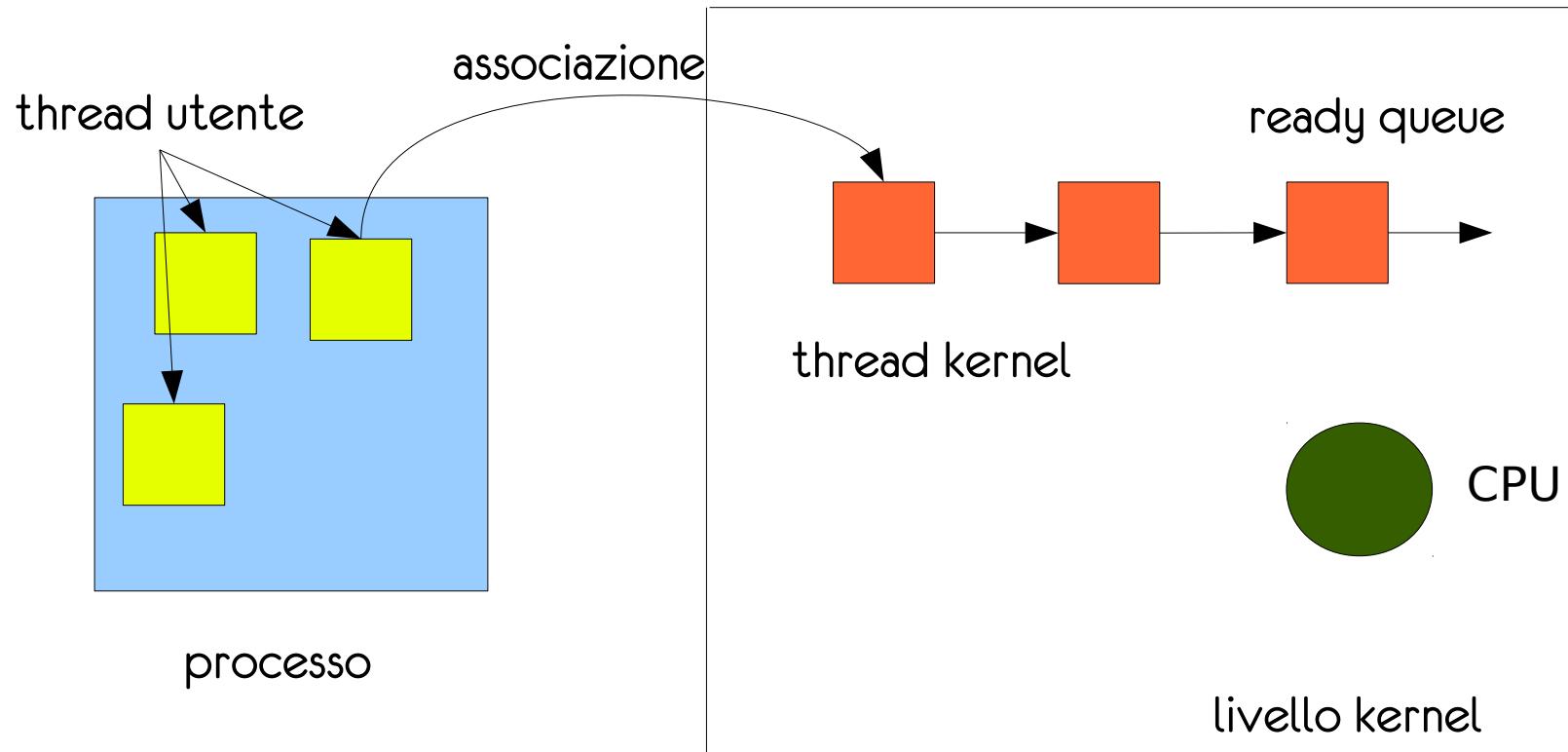
Modelli di thread (b)

- 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



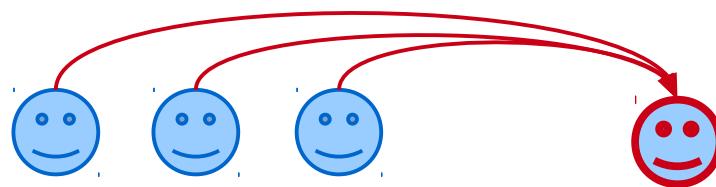
- Per far effettuare lo scheduling della CPU a livello di thread occorre associare thread utente a thread kernel.
- Vi sono **tre modelli** secondo i quali questa associazione può essere fatta: uno a uno (thread kernel), molti a uno (thread utente), molti a molti (soluzione ibrida)

Livello utente e livello kernel

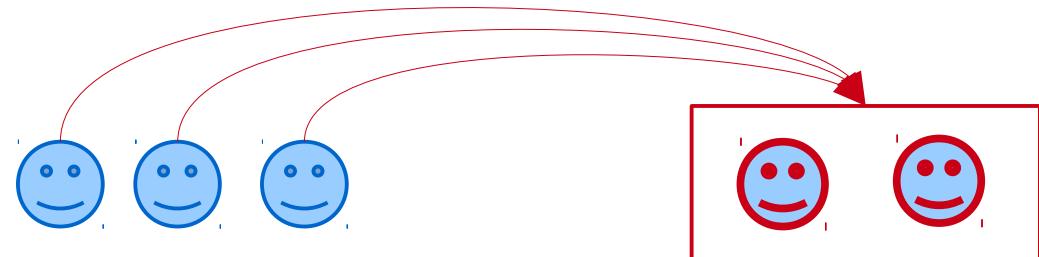
- **uno (utente) a uno (kernel)**: soluzione adottata da Linux e Windows



- **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



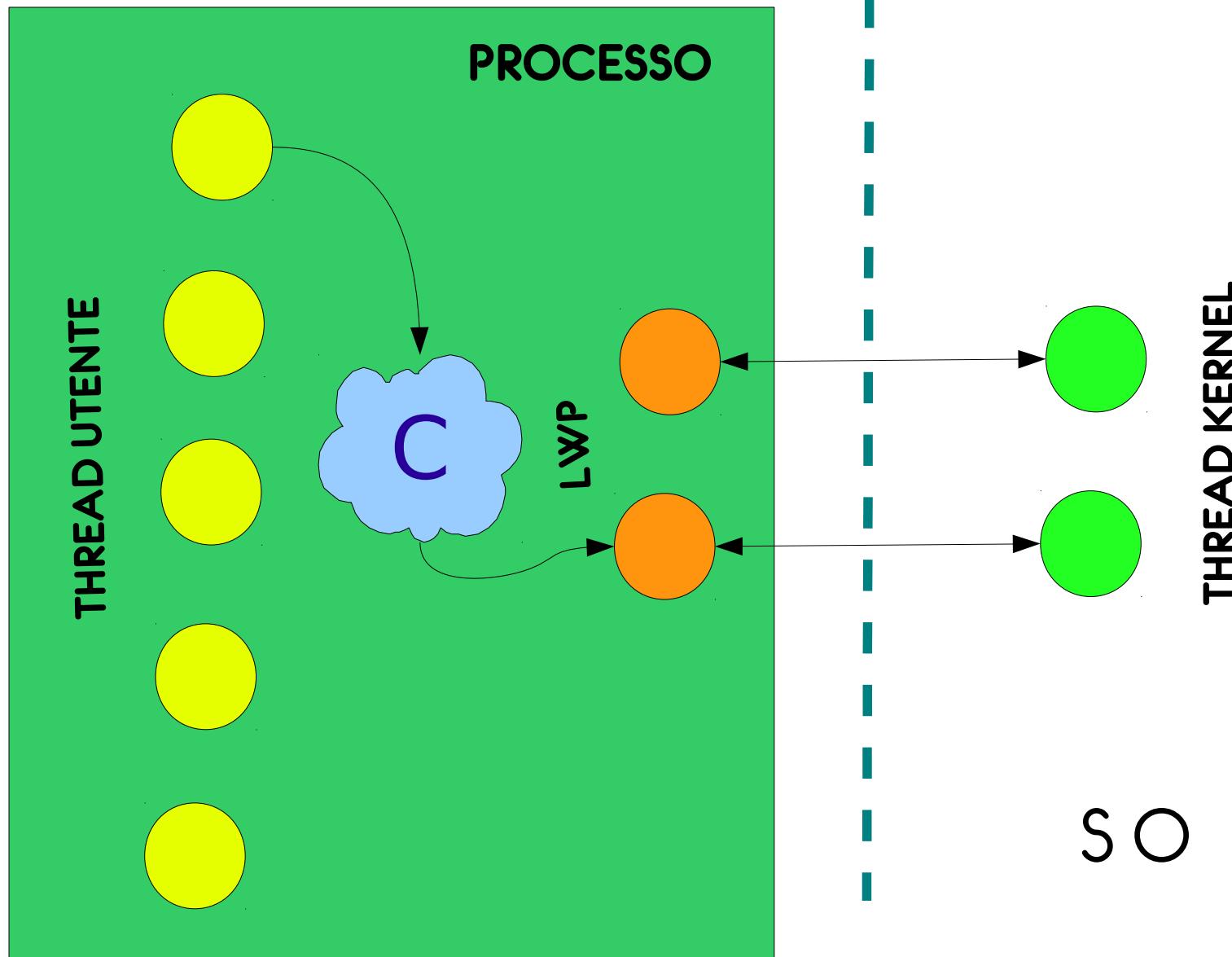
- nei SO che usano il modello molti a molti (o quello a due livelli) si ha la necessità di effettuare uno scheduling dei thread utente per l'accesso ai thread kernel
- questo spesso viene fatto introducendo un nuovo oggetto fra thread utente e thread kernel, detto lightweight process (**LWP**, processo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

thread utente pronti



NB: non è uno
scheduling della
CPU !!!

LWP



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



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

thread kernel



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

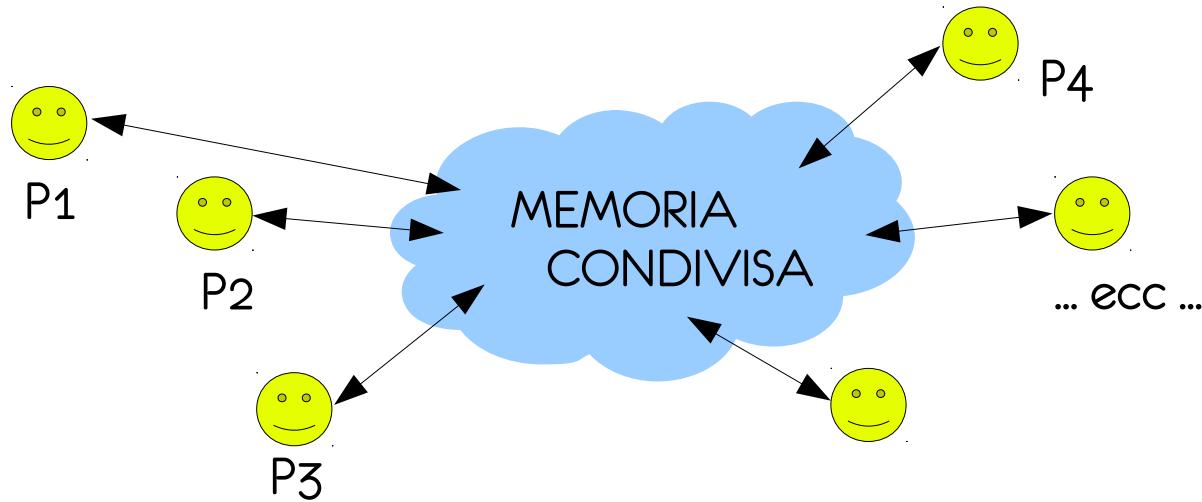
Esecuzione concorrente asincrona

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

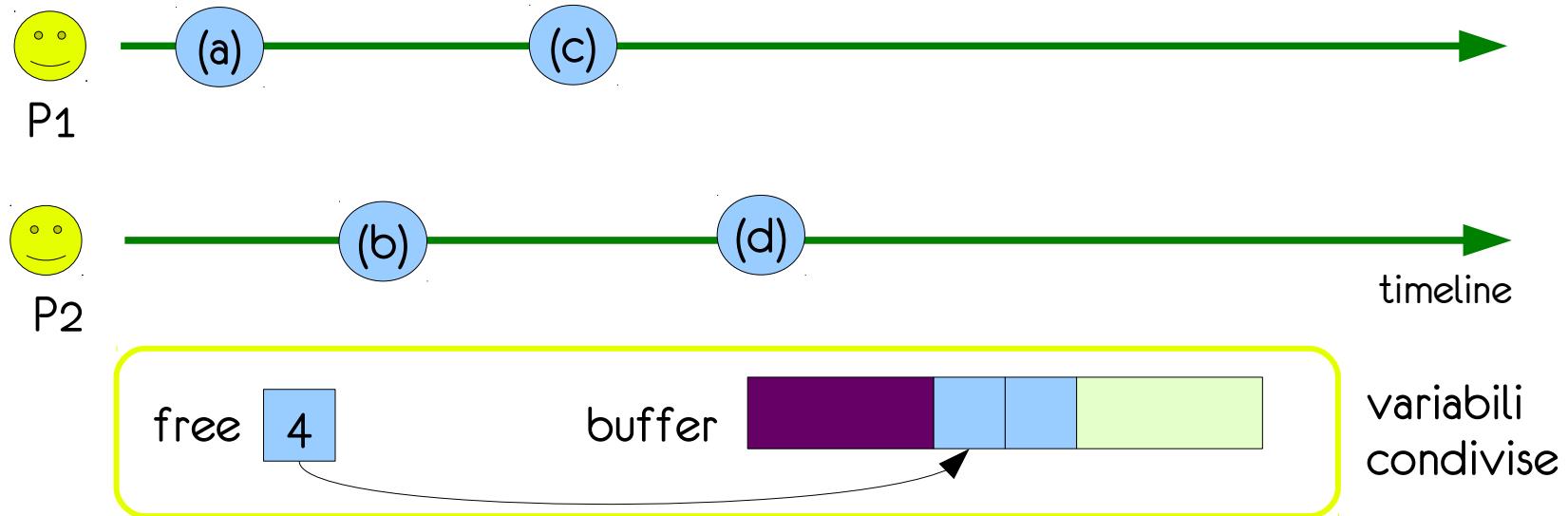


- Consideriamo **due processi produttori** che inseriscono dati nella stessa area di memoria condivisa
- L'indice della prima posizione libera è indicato da free
- Inserzione:

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



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

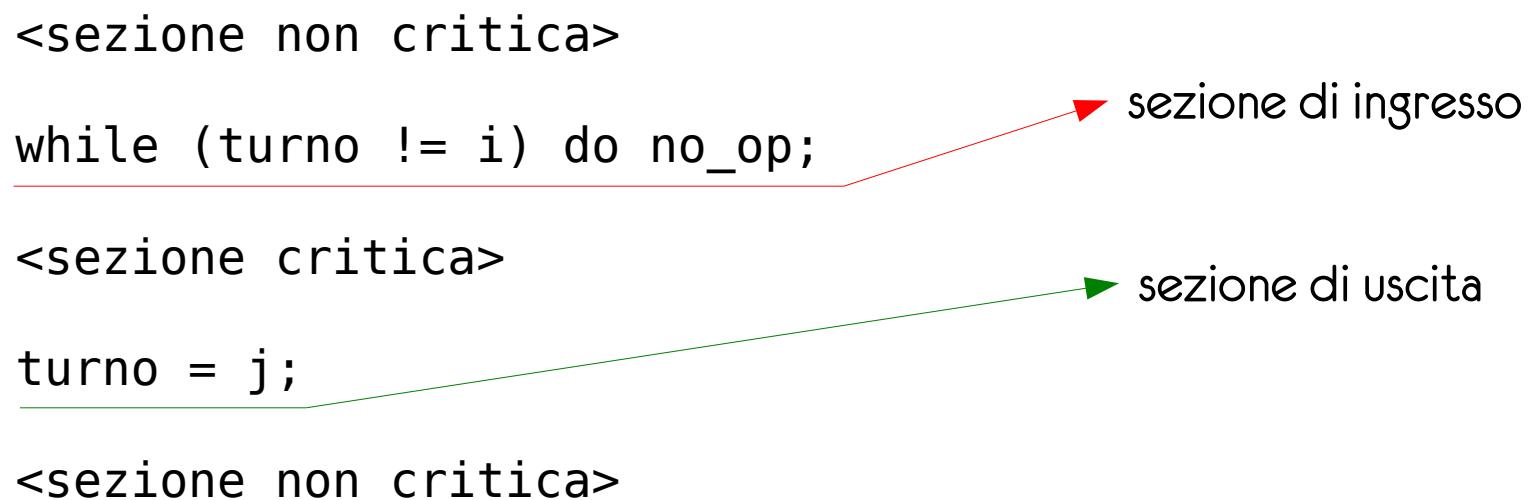
Sezione critica 4/4

- 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 critica>  
turno = j;  
<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 Pi non avesse alcuna intenzione di entrare in sezione critica? Allora neppure Pj potrebbe entrarvi, pur volendolo
- si viola quindi la condizione di progresso: Pi è in sezione non critica, non dovrebbe impedire a Pj 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 critica>

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

<sezione non critica>

→ sezione di ingresso

→ sezione di uscita

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 Pi e Pj 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

- 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 critica>

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

<sezione non critica>

sezione di ingresso

sezione di uscita

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	P2	tempo
P1-1 flag[1]=true		
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.		

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;
```

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;           ➔ sto richiedendo un biglietto  
ticket[i] = max_ticket + 1;    ➔ mi viene dato un biglietto  
choosing[i] = false;          ➔ non sto più richiedendo un biglietto  
  
for (j = 0; j < N; j++) {  
    while (choosing[j]) no_op;  ➔ aspetto eventualmente che gli  
                                venga rilasciato il biglietto  
    while  
        (ticket[j] != 0 &&           ➔ se il processo j desidera entrare in SC  
            ticket[j] < ticket[i] ||     ➔ e ha un biglietto precedente  
            ticket[j] == ticket[i] && j < i) no_op;  
    }  
  
per ogni processo concorrente eseguo un controllo
```

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;  
}
```

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

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 richie-
dere un nuovo ticket per entrare in SC, met-
tendo 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

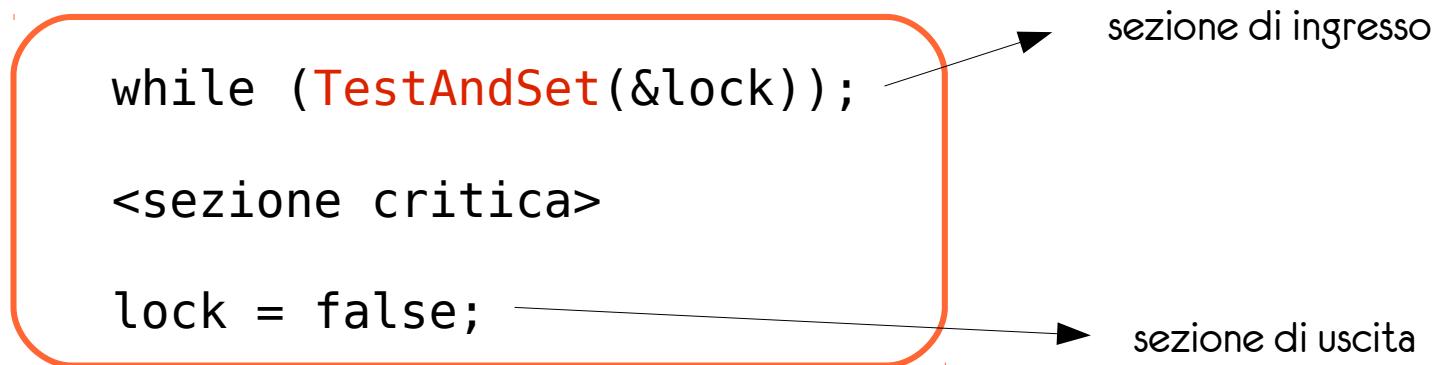
Uso di TestAndSet

- 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;
```



The code示例 shows the use of the TestAndSet operation to manage a critical section. It starts with a while loop that repeatedly calls TestAndSet on the variable lock. If the lock is false (i.e., not set), it enters the critical section (indicated by the text <sezione critica>). Inside the section, the variable lock is set to false, which exits the while loop and returns control to the calling thread.

- 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":

The diagram shows a code snippet enclosed in a red rounded rectangle. The code consists of three lines: a while loop that tests the value of the variable 'lock' using the `TestAndSet` function, a block labeled '`<sezione critica>`' containing some code, and an assignment statement `lock = false;`. Two arrows point from the text 'sezione di ingresso' to the start of the while loop and from the text 'sezione di uscita' to the end of the assignment statement.

```
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[i])  
    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
sezine 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; → valore del semaforo  
    processo *lista; → lista di attesa relativa al semaforo  
} semaforo;
```

```
P (semaforo *s) {  
    S->valore --; → decremento il valore del semaforo  
    if (S->valore < 0) { → se il valore diventa negativo occorre  
        <aggiungi il PCB di questo processo a S->lista>  
        block(); → sospendere il processo  
    }  
}
```

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++; incremento il valore del semaforo  
    if (S->valore<0) { se il valore è negativo, allora ci sono processi  
da risvegliare  
        <scegli un PCB P da S->lista>  
        wakeup(P); 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 $P(\text{mutex}) \dots V(\text{mutex})$, dove mutex è un semaforo di mutua esclusione
- Accesso limitato**: tutti i processi coinvolti parentesizzano le loro sezioni critiche con $P(\text{nrис}) \dots V(\text{nrис})$, dove nrис 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: $P(\text{sem})$ codice	P2: codice $V(\text{sem})$
----------------------------------	----------------------------------

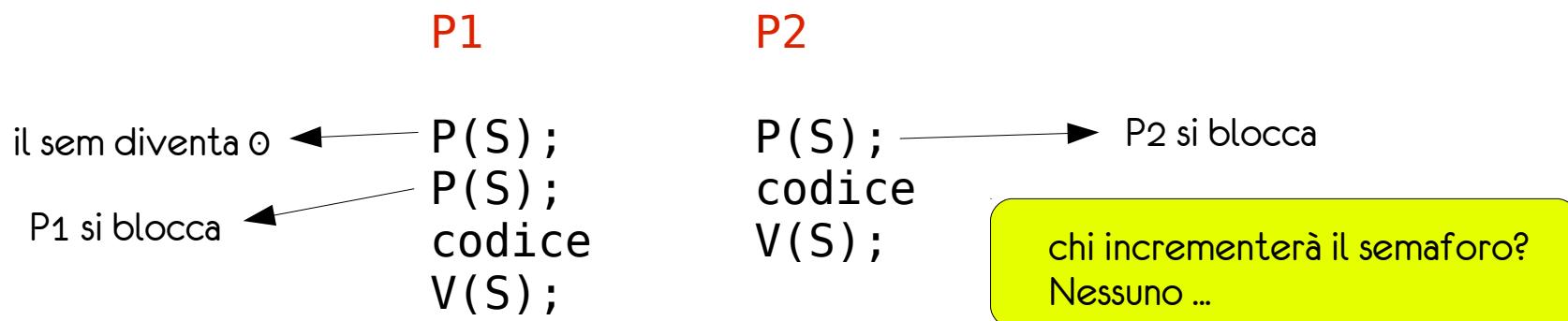
in quale ordine vengono eseguiti P1 e P2?

Operazioni atomiche

- Le operazioni sui **semafori** devono essere eseguite in modo atomico: sono **sezioni critiche** perché i semafori sono variabili condivise
- su sistemi monoprocessoressi, atomicità ottenuta **disabilitando gli interrupt**: non è un problema perché i tempi di esecuzione di P e V sono molto brevi
- su sistemi multiprocessoressi, 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

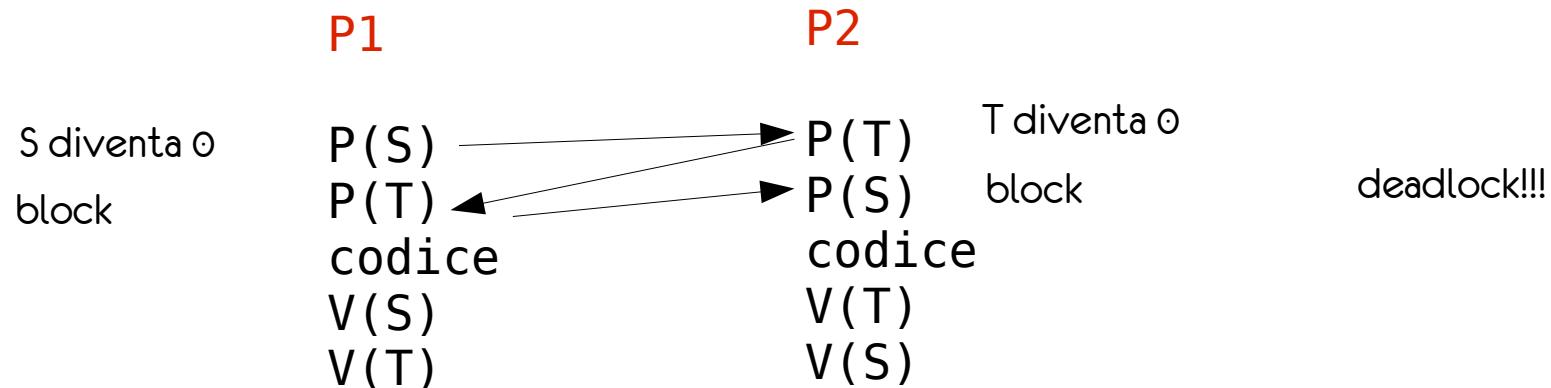
- Situazioni più subdole in cui si può generare deadlock sono causate dall'utilizzo di più semafori

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	P2
$P(S)$	$P(T)$
$P(T)$	$P(S)$
codice	codice
$V(S)$	$V(T)$
$V(T)$	$V(S)$

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

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);
```

```
    <consuma elemento>
```

```
}
```

rimane fermo finché non c'è nulla da consumare, quando occupate viene incrementato da un produttore lo decremente, 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 estraе 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

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

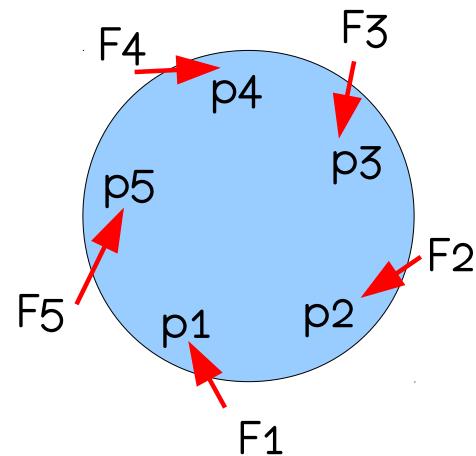
Problemi classici di sincronizzazione

- 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

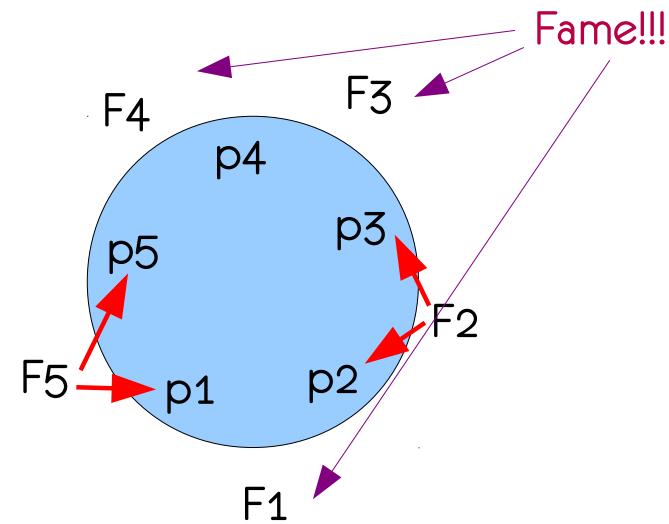
è un problema interessante perché rappresenta un'ampia categoria di situazioni pratiche in cui diversi processi hanno bisogno di più risorse e bisogna evitare situazioni di deadlock e di starvation (es. quando riesco ad avere solo una parte delle risorse necessarie)



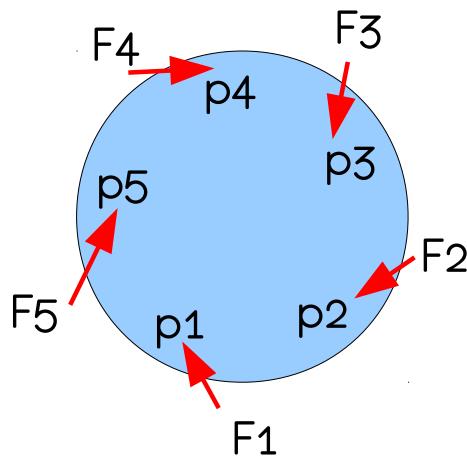
deadlock e altri guai



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

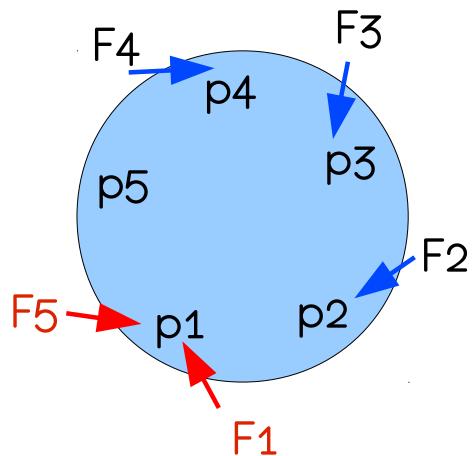
Prevenire il deadlock

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 encapsulare risorse di basso livello, come i semafori, in tipi di dati astratti, come i **monitor**, offerti poi come nuovi costrutti del linguaggio

Monitor

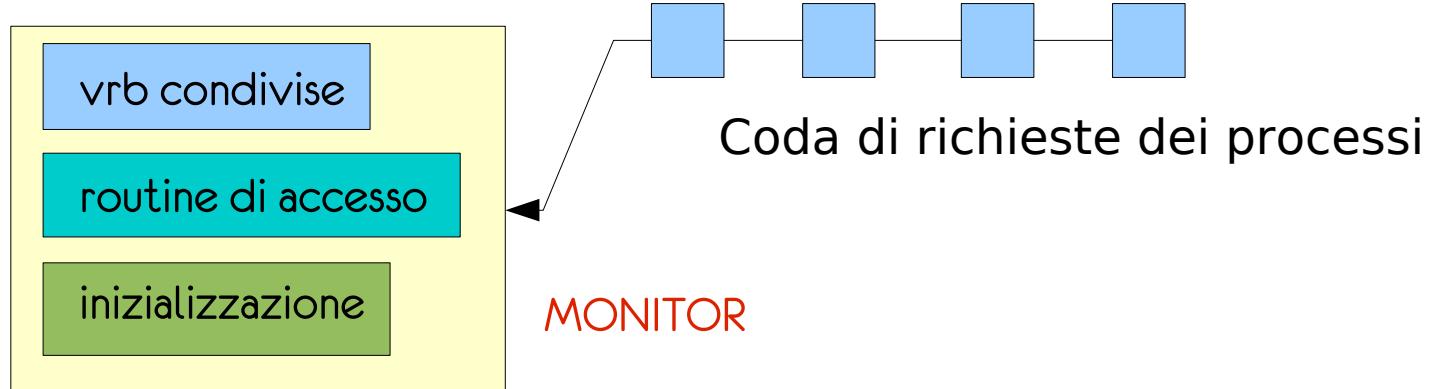
- **Monitor** (Dijkstra, Brinch Hansen) è un costrutto di sincronizzazione contenente i dati e le operazioni necessarie per allocare una risorsa condivisa usabile in modo seriale
- È un **Abstract Data Type**: incapsula dei dati mettendo a disposizione le operazioni necessarie per manipolarli
- Le variabili 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

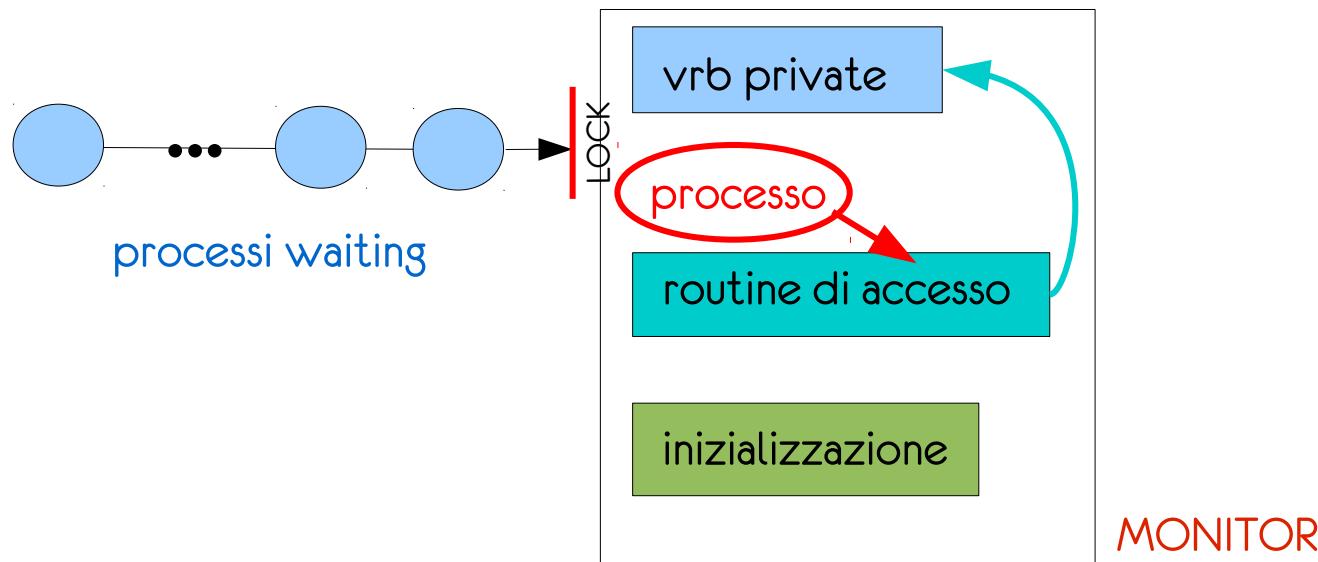
- Brinch-Hansen: “You can imagine the (monitor) calls as a **queue of messages being served one at a time**. The monitor will receive a message and try to carry out the request as defined by the procedure and its input parameters. If the request can immediately be granted the monitor will return parameters . . . 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

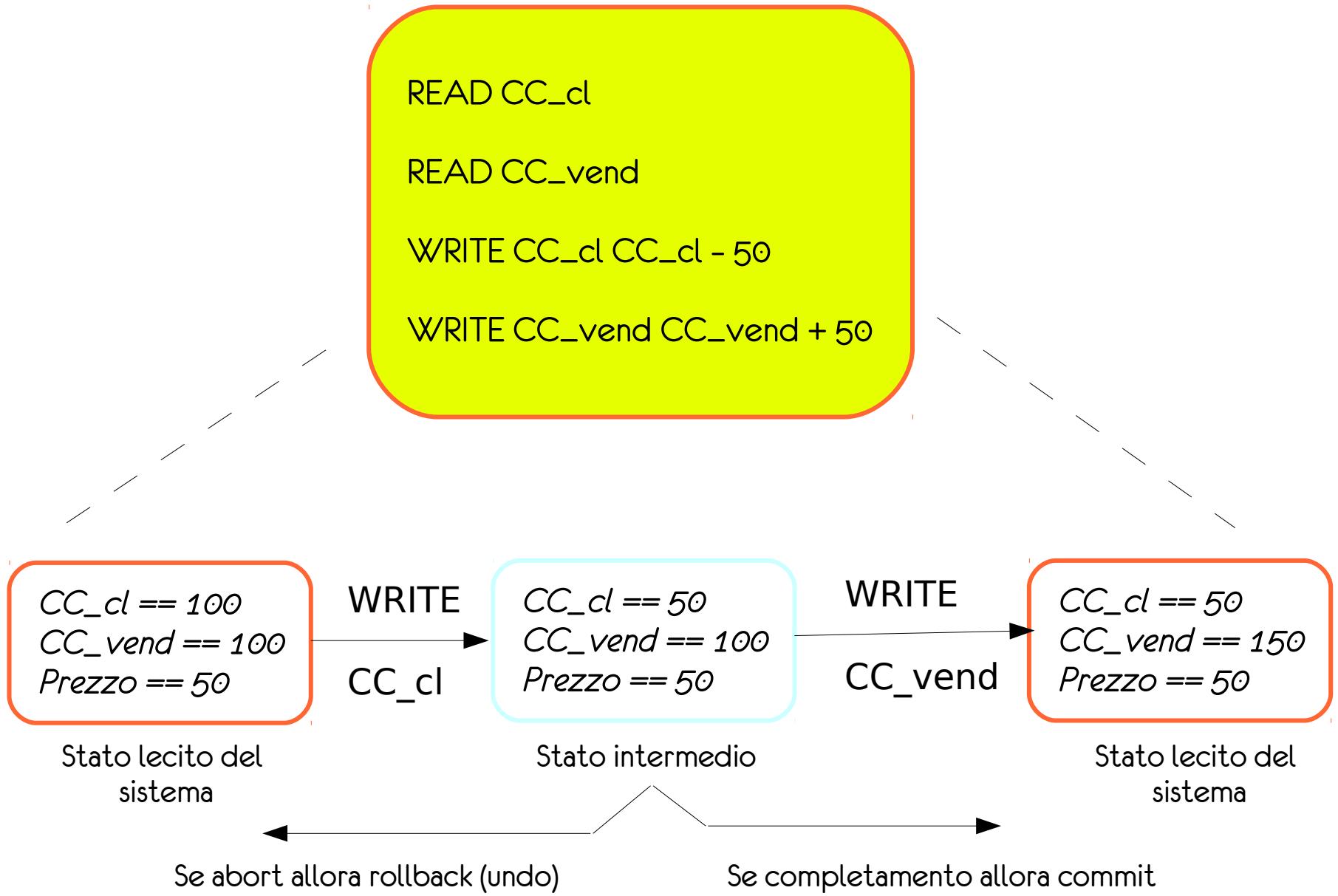
READ CC_vend

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:
 $\langle T, \text{start} \rangle$
- si registra una sequenza di tuple, ciascuna relativa ad una operazione di write prossima da eseguire (**write-ahead logging**), siffatte:
 $\langle \text{ID transazione}, \text{ID dato modificato}, \text{valore precedente}, \text{nuovo valore} \rangle$
- si registra il successo della transazione T:
 $\langle T, \text{commit} \rangle$

Ripristino/abort

- A seguito di un crash di sistema, il SO 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 SO 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 SO 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 SO esegue un $\text{redo}(T)$, attua le modifiche registrate.

Ripristino/abort

- A seguito di un crash di sistema, il SO 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 SO 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 SO 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 SO esegue un $\text{redo}(T)$, attua le modifiche registrate.
- Questo genere di meccanismo è usato dai sistemi di **journaling** proposti dai moderni SO 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

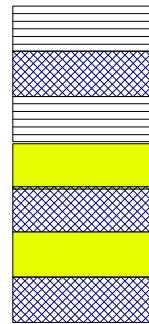
- Quando si ha un crash di sistema, si scorre a ritroso il logfile fino a identificare la prima occorrenza di <checkpoint>
- Tutte le operazioni che precedono questa etichetta possono essere ignorate perché per definizione di checkpoint tutte le modifiche sono già in memoria stabile
- Le operazioni di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



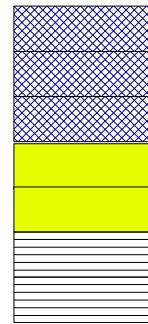
Transazioni atomiche concorrenti

- 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



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

T1

read(A)
write(A)
read(B)
write(B)

T2

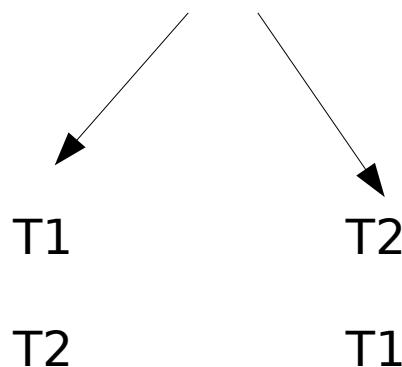
read(A)
write(A)
read(B)
write(B)

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

Esempio



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

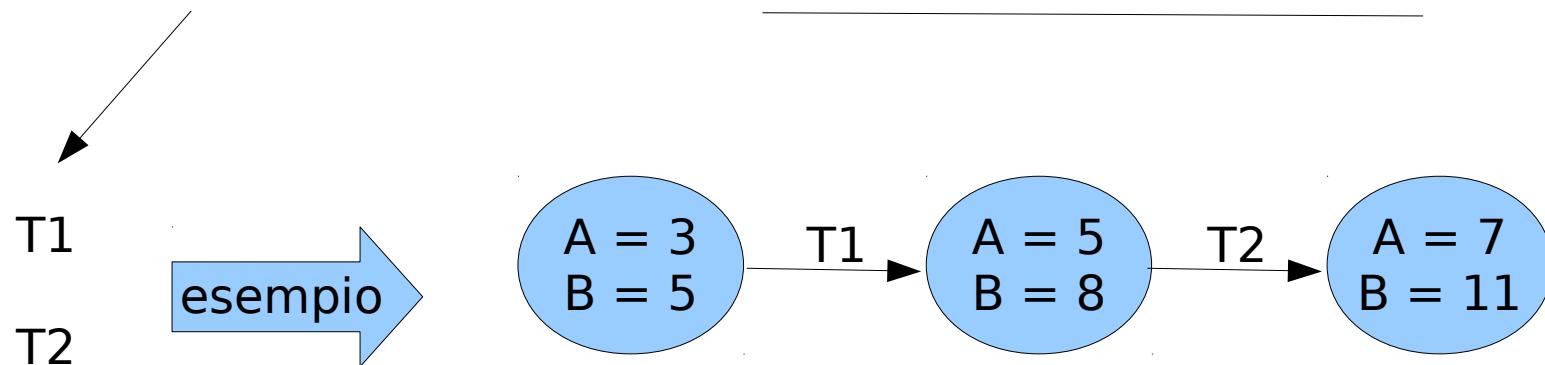


Gli **unici stati corretti** per il sistema sono quelli raggiungibili eseguendo T1 e T2 in modo sequenziale

T1 **T2**

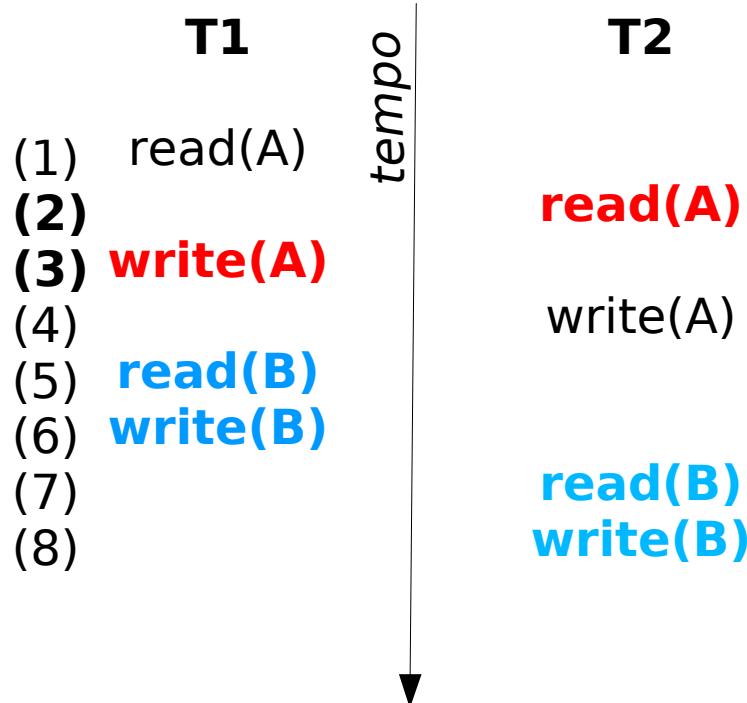
read(A) read(A)
A=A+2 A=A+2
write(A) write(A)
read(B) read(B)
B=B+3 B=B+3
write(B) write(B)

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

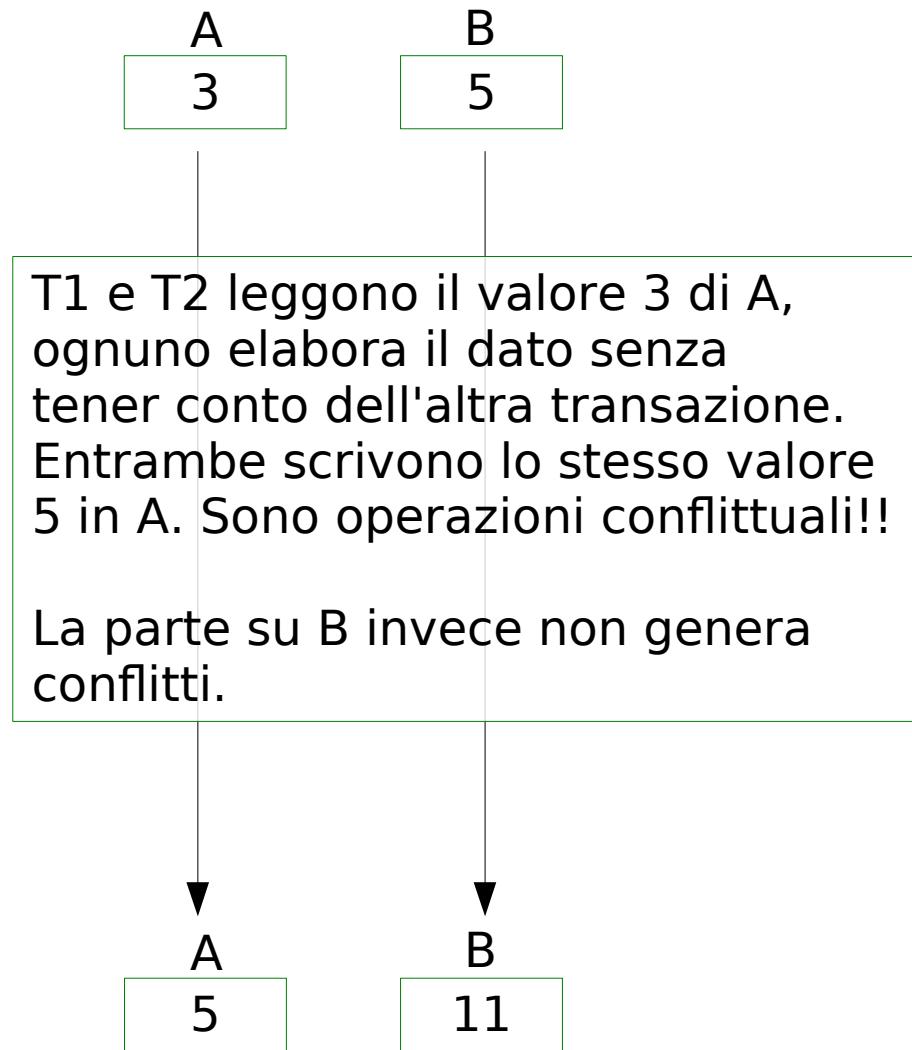


Problema: i sistemi ad alte prestazioni non possono permettersi di proibire l'esecuzione concorrente delle transazioni. Occorre consentire un po' di interleaving però in modo tale che lo stato raggiunto sia uno degli stati leciti (stesso risultato in tempo più rapidi)

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'interleaving? 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'espressione
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'espressione
a partire dal valore 100

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

tempo

Serializzabilità

- In generale 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

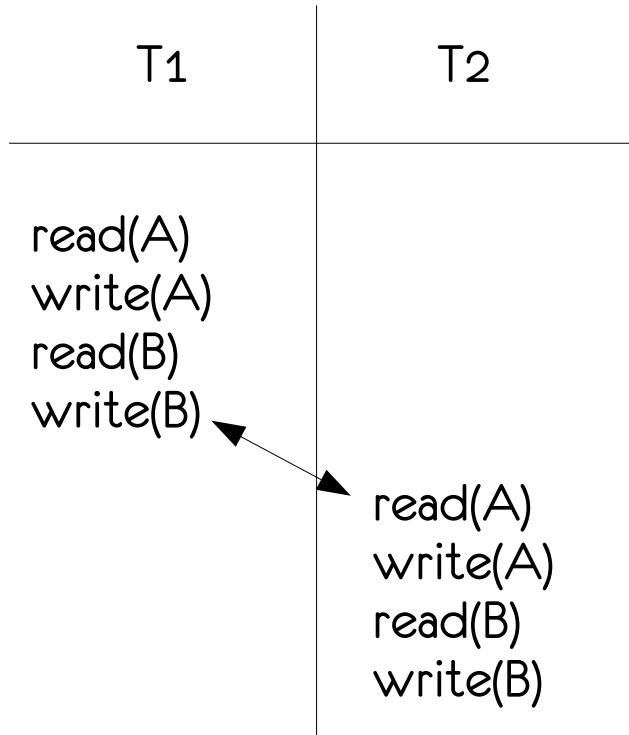
- Introduciamo un concetto nuovo: **operazioni conflittuali**:
date due **transazioni** T1 e T2 e le due **operazioni** 01
(appartenente a T1) e 02 (appartenente a T2), se 01 e 02
appaiono in successione, accedono agli stessi dati e almeno
una delle due operazioni è una **write**, allora 01 e 02 sono
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

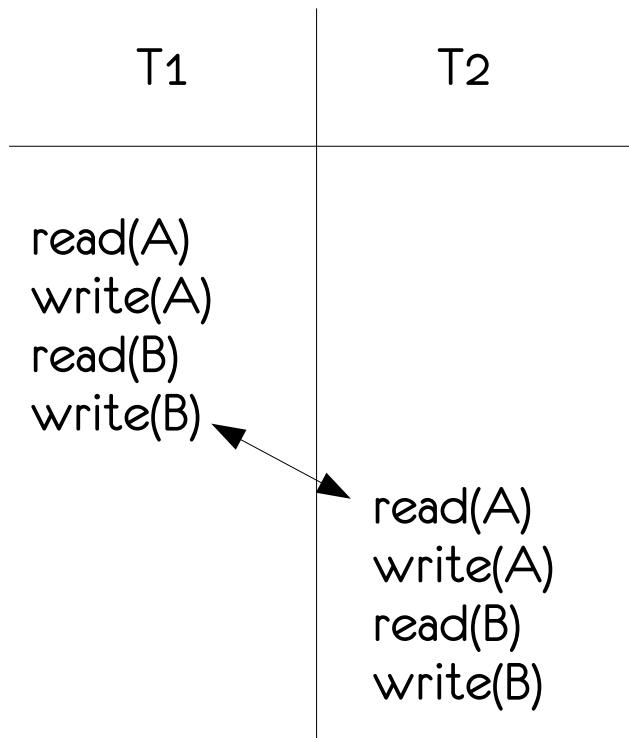
esempio



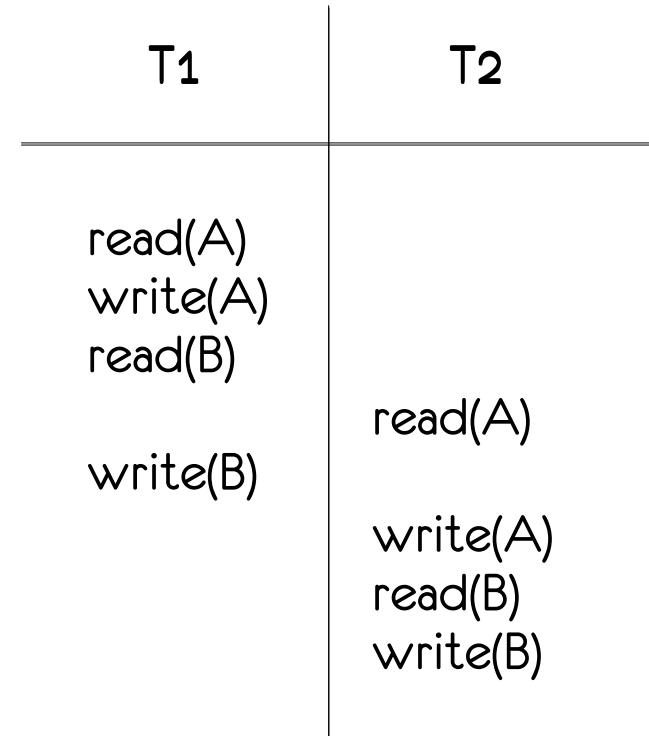
Proviamo a introdurre un po' di interleaving
write(B) e read(A) sono conflittuali?

esecuzione seriale di
T1 e T2

esempio

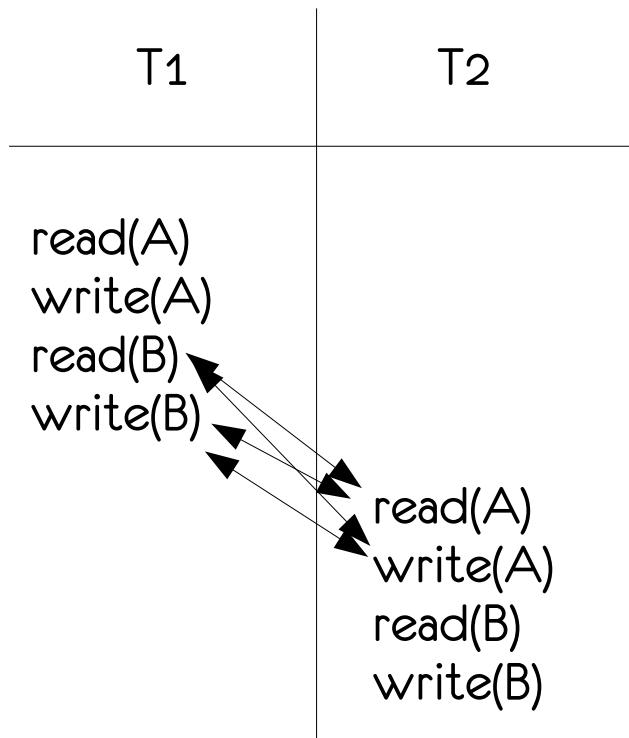


esecuzione seriale di
T1 e T2

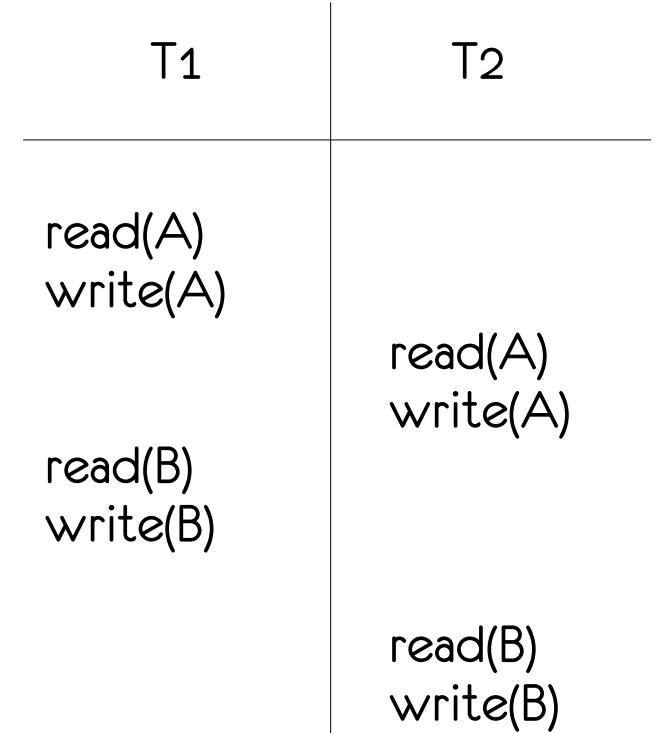


esecuzione concorrente di
T1 e T2 equivalente a
quella seriale

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)

esecuzione seriale di
T1 e T2

conflitto!

Sono operazioni
conflittuali!!!

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

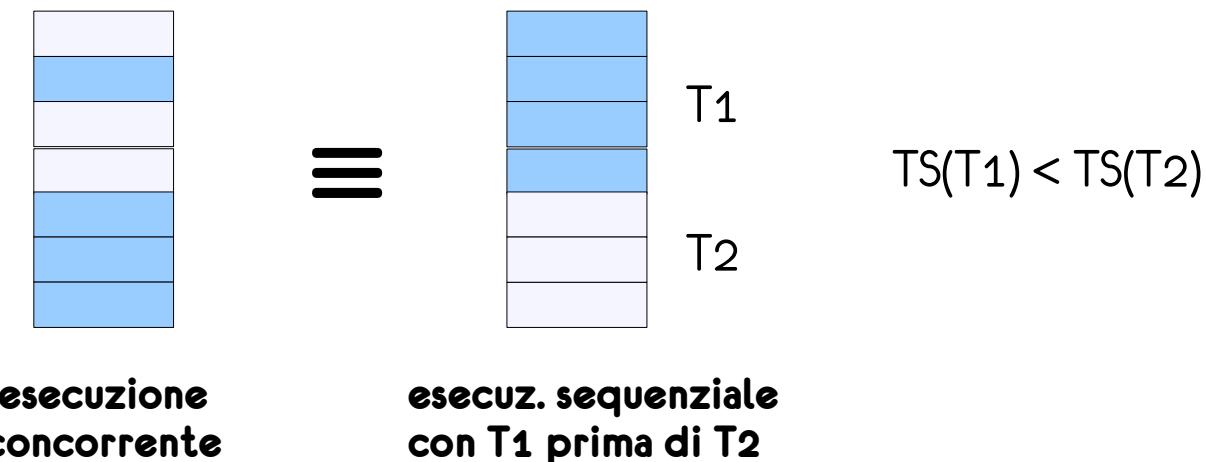
- Per garantire la serializzabilità si può usare un meccanismo di **lock** in cui:
 - a ogni dato soggetto a transazione si associa un lock
 - il lock di un dato può essere:
 - **S** (indica la possibilità di condivisione): la transazione può leggere il dato ma non scriverlo
 - **X** (indica esclusività): 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**
 - ...

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 $R(D)$ assgnando $\max(R(D) , TS(T))$

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)	write(B)
read(B)	read(A)
write(A)	write(A)
read(B)	

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

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

read(B): è eseguibile?

$TS(T1) < W(A)? \quad 1 < 0? \quad \text{no!}$
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 esegibile?

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

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

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

read(A): è esegibile?

$TS(T2) < W(A)?$ $2 < 0?$ 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?

$TS(T1) < R(A)?$ $1 < 2?$ si!

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

deadlock

capitolo 7 del libro (VII ed.)

Introduzione

- **Deadlock:** situazione per cui un insieme di processi sono fermi in attesa di un evento che solo uno dei processi appartenenti all'insieme stesso potrebbe causare
- Noi vedremo il problema del deadlock in relazione al problema della gestione di risorse, anche se si tratta di un **problema più generale**

Introduzione

- **Deadlock:** situazione per cui un insieme di processi sono fermi in attesa di un evento che solo uno dei processi appartenenti all'insieme stesso potrebbe causare
- **Problema della gestione delle risorse:**
 - un sistema fisico può essere visto come un insieme di **risorse**
 - ogni risorsa può essere presente in un certo numero di **istanze equivalenti**, es:
 - CPU: 2 (sistema biprocessore)
 - stampanti: 3
 - CD-reader: 1

Introduzione

- Un processo che vuole usare N istanze di una certa risorsa deve farne richiesta al gestore delle risorse, cioè al SO
- Si distinguono 3 fasi:

- **richiesta**: può causare attesa
- uso
- **rilascio**

sono effettuati tramite system call

-
- Il SO tiene traccia di quali risorse sono assegnate a quale processo
 - Richiesta/rilascio: tramite **system call ad hoc** (es. open e close di file) oppure tramite **strumenti di sincronizzazione** (es. semafori)

Condizioni al deadlock

- Es. di deadlock fra due processi: P1 detiene l'unico lettore di fotografie e vuole la stampante per stampare delle foto, P2 ha la stampante e vuole il lettore perché deve, anch'esso, stampare delle foto ...
- **Condizioni necessarie al deadlock**

<1> **ME**: risorse non condivisibili

<2> **possesso e attesa**: un processo attende le risorse non disponibili, anche detenendo già il possesso di alcune delle risorse a lui necessarie

<3> **no prelazione**: il rilascio non viene forzato

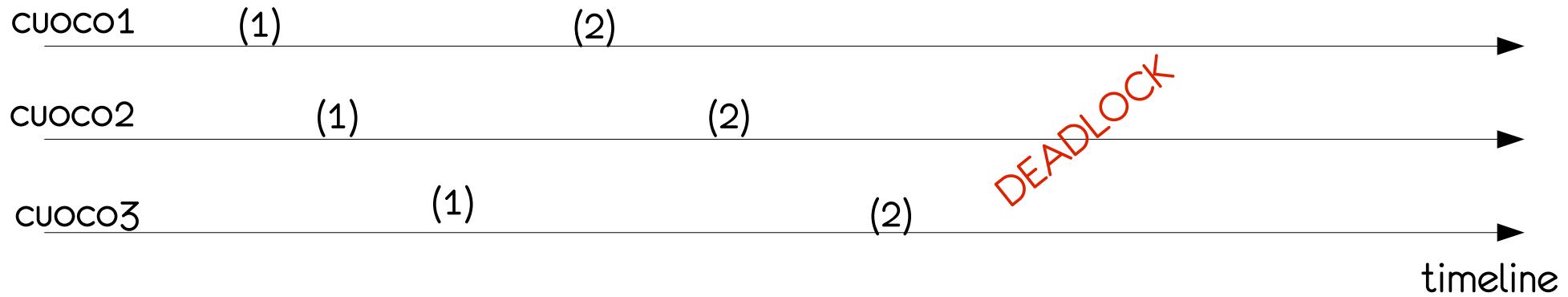
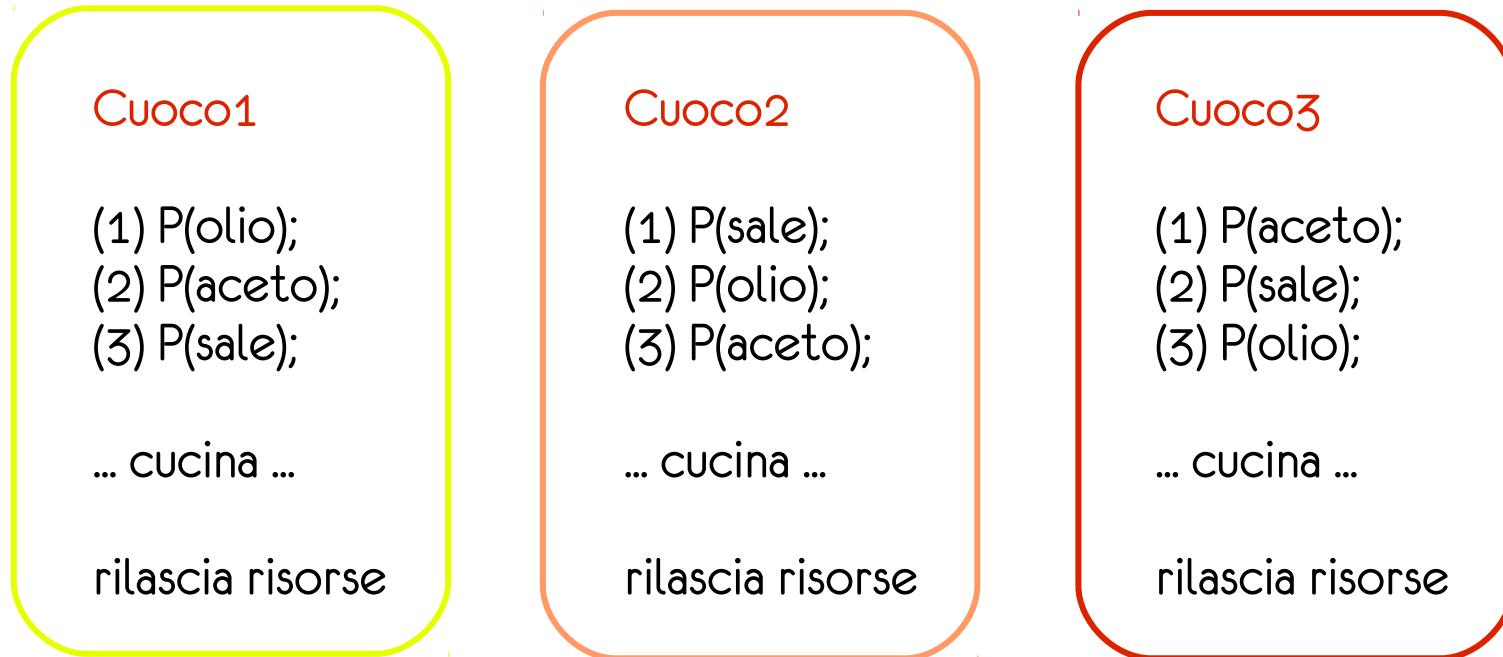
<4> **attesa circolare**: siano i processi in questione P1, ..., Pn, allora P1 attende risorse da P2, che attende risorse da P3, ecc. e Pn attende risorse da P1

basta che una **non** sia vero per evitare il deadlock

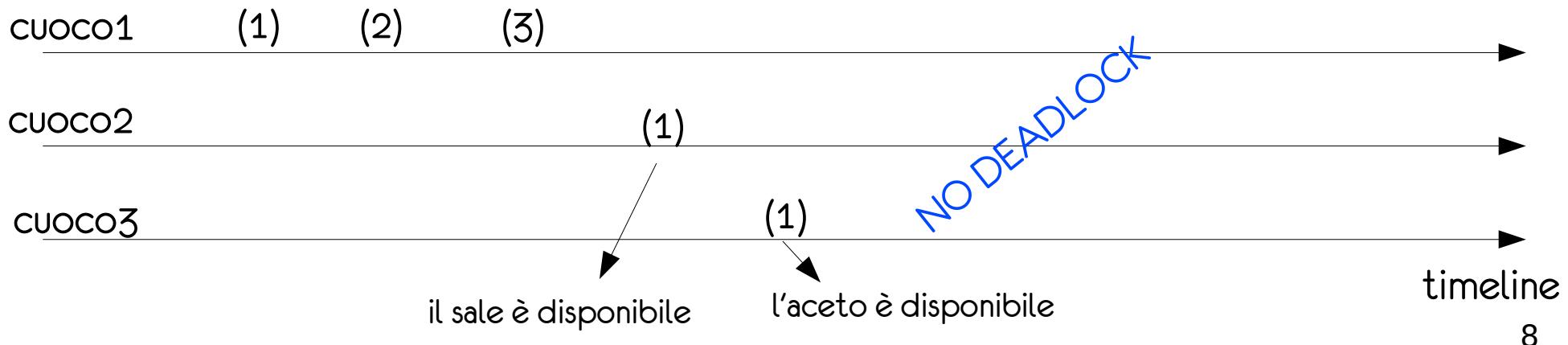
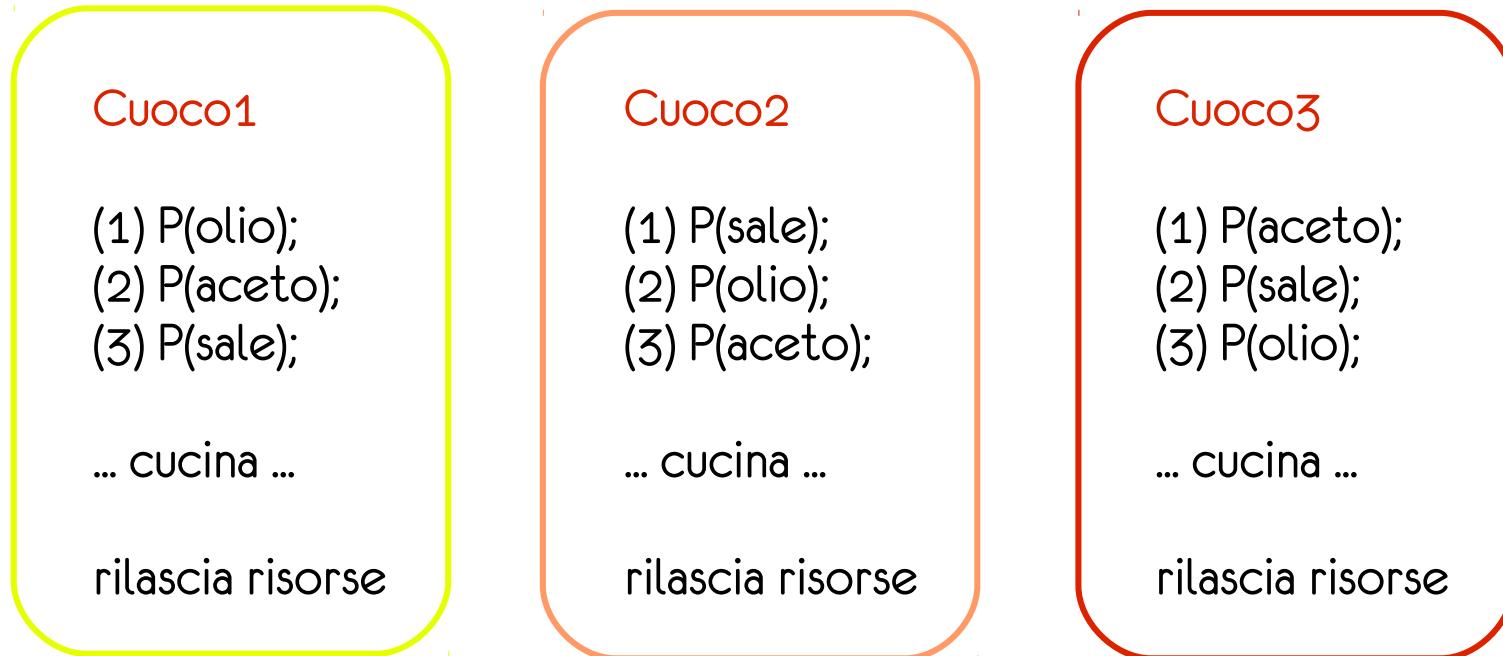
Esempio



In codice



In codice



Grafo di assegnazione delle risorse

- Rappresentazione delle assegnazioni che permette di rilevare situazioni di deadlock
- È un grafo $G = \langle V, E \rangle$ tale per cui:
 - ...

Grafo di assegnazione delle risorse

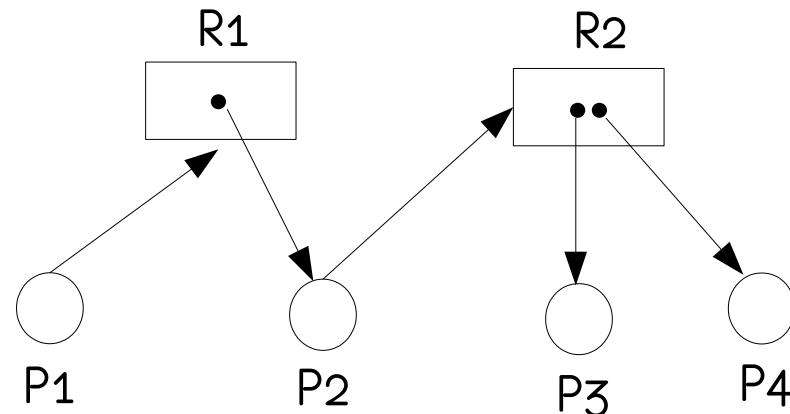
- Rappresentazione delle assegnazioni che permette di rilevare situazioni di deadlock
- È un grafo $G = \langle V, E \rangle$ tale per cui:
 - V è l'insieme dei vertici ed è partizionato in due sottoinsiemi P ed R , $P \cap R = \emptyset$:
 - $P = \{P_1, \dots, P_n\}$ è l'insieme di tutti i processi del sistema
 - $R = \{R_1, \dots, R_m\}$ è l'insieme di tutte le classi di risorse del sistema
 - ...

Grafo di assegnazione delle risorse

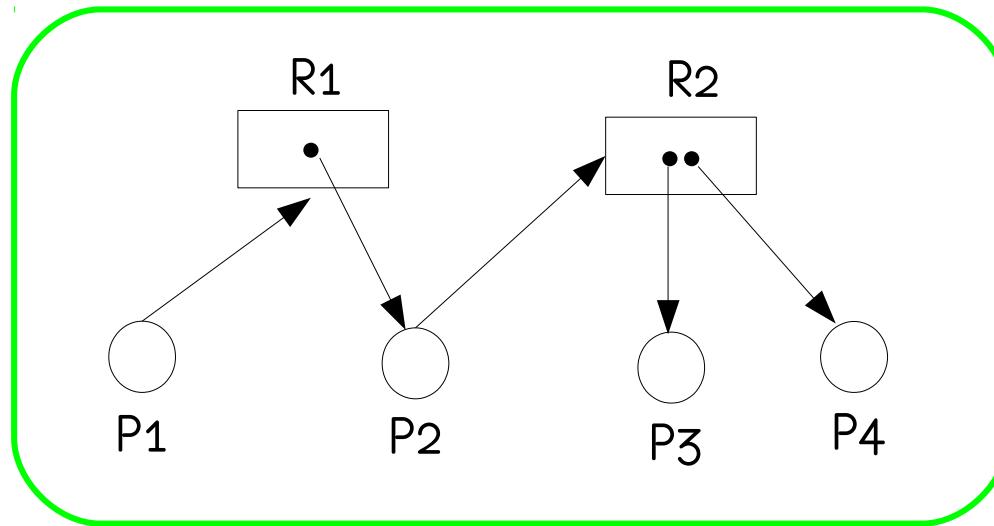
- Rappresentazione delle assegnazioni che permette di rilevare situazioni di deadlock
- È un grafo $G = \langle V, E \rangle$ tale per cui:
 - V è l'insieme dei vertici ed è partizionato in due sottoinsiemi P ed R , $P \cap R = \emptyset$:
 - $P = \{P_1, \dots, P_n\}$ è l'insieme di tutti i processi del sistema
 - $R = \{R_1, \dots, R_m\}$ è l'insieme di tutte le classi di risorse del sistema
 - E è l'insieme degli archi:
 - Un'arco direzionato da R_i a P_j , $R_i \rightarrow P_j$, indica che una risorsa di classe R_i è stata assegnata al processo P_j (**arco di assegnazione**)
 - Un'arco direzionato da P_j a R_i , $P_j \rightarrow R_i$, indica che il processo P_j ha richiesto ed è in attesa di una risorsa di tipo R_i (**arco di richiesta**)

Rappresentazione grafica del grafo di assegnazione delle risorse

- Ogni processo P_i è rappresentato da un cerchietto
- Ogni classe di risorsa R_i è rappresentata da un rettangolo contenente tanti puntini quante sono le sue istanze
- Un **arco di assegnazione** parte da una specifica risorsa (un puntino) ed è diretto a un processo
- Un **arco di richiesta** parte da un processo e termina a un rettangolo (la classe della risorsa)



Notazione grafica



P Processo P

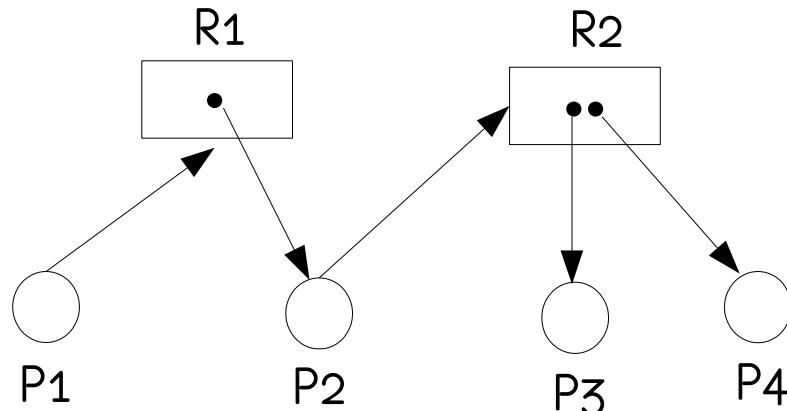
P R Arco di richiesta

R Classe di risorsa R

P R Arco di assegnazione

R Classe di risorsa R con due istanze

Esempio



Abbiamo 4 processi e 2 classi di risorse
R1 ha una sola istanza, R2 ha 2 istanze
P1 ha richiesto una risorsa di tipo R1
L'unica risorsa di questa classe è assegnata a P2,
che ha richiesto una risorsa di classe R2
Nessuna di queste è libera al momento,
essendo esse assegnate a P3 e P4

I 3 cuochi

Cuoco1

- (1) P(olio);
- (2) P(aceto);
- (3) P(sale);

... cucina ...

rilascia risorse

Cuoco2

- (1) P(sale);
- (2) P(olio);
- (3) P(aceto);

... cucina ...

rilascia risorse

Cuoco3

- (1) P(aceto);
- (2) P(sale);
- (3) P(olio);

... cucina ...

rilascia risorse

Quante classi di risorse abbiamo?

Quante istanze per ogni classe?

Quanti processi?

Chi detiene quale risorsa?

Chi richiede quale risorsa?

informazioni di tipo statico

informazioni note a run-time
il grafo cattura l'andamento
dell'esecuzione

I 3 cuochi

Cuoco1

- (1) P(olio);
- (2) P(aceto);
- (3) P(sale);

... cucina ...

rilascia risorse

Cuoco2

- (1) P(sale);
- (2) P(olio);
- (3) P(aceto);

... cucina ...

rilascia risorse

Cuoco3

- (1) P(aceto);
- (2) P(sale);
- (3) P(olio);

... cucina ...

rilascia risorse

Quante classi di risorse abbiamo?

3

Quante istanze per ogni classe?

1

Quanti processi?

3

Chi detiene quale risorsa?

Chi richiede quale risorsa?

I 3 cuochi

Cuoco1

- (1) P(olio);
- (2) P(aceto);
- (3) P(sale);

... cucina ...

rilascia risorse

Cuoco2

- (1) P(sale);
- (2) P(olio);
- (3) P(aceto);

... cucina ...

rilascia risorse

Cuoco3

- (1) P(aceto);
- (2) P(sale);
- (3) P(olio);

... cucina ...

rilascia risorse

Quante classi di risorse abbiamo?

Quante istanze per ogni classe?

Quanti processi?

Chi detiene quale risorsa?

Chi richiede quale risorsa?

[a] Quando un processo P esegue la richiesta di una risorsa aggiungo un arco di richiesta

[b] Quando P ottiene la risorsa, cancello tale arco e ne inserisco uno di assegnazione

[c] Quando P rilascia la risorsa l'arco di assegnazione viene rimosso



I 3 cuochi

Cuoco1

- (1) P(olio);
- (2) P(aceto);
- (3) P(sale);

... cucina ...

rilascia risorse

Cuoco2

- (1) P(sale);
- (2) P(olio);
- (3) P(aceto);

... cucina ...

rilascia risorse

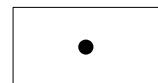
Cuoco3

- (1) P(aceto);
- (2) P(sale);
- (3) P(olio);

... cucina ...

rilascia risorse

olio



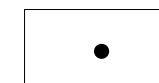
C1

aceto

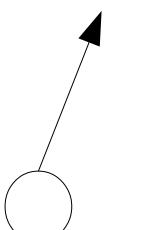
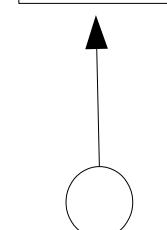
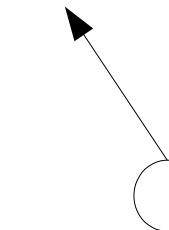


C3

sale



C2



I 3 cuochi

Cuoco1

- (1) P(olio);
- (2) P(aceto);
- (3) P(sale);

... cucina ...

rilascia risorse

Cuoco2

- (1) P(sale);
- (2) P(olio);
- (3) P(aceto);

... cucina ...

rilascia risorse

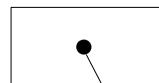
Cuoco3

- (1) P(aceto);
- (2) P(sale);
- (3) P(olio);

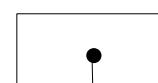
... cucina ...

rilascia risorse

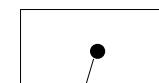
olio



aceto



sale



C1

C3

C2

I 3 cuochi

Cuoco1

- (1) P(olio);
- (2) P(aceto);
- (3) P(sale);

... cucina ...

rilascia risorse

Cuoco2

- (1) P(sale);
- (2) P(olio);
- (3) P(aceto);

... cucina ...

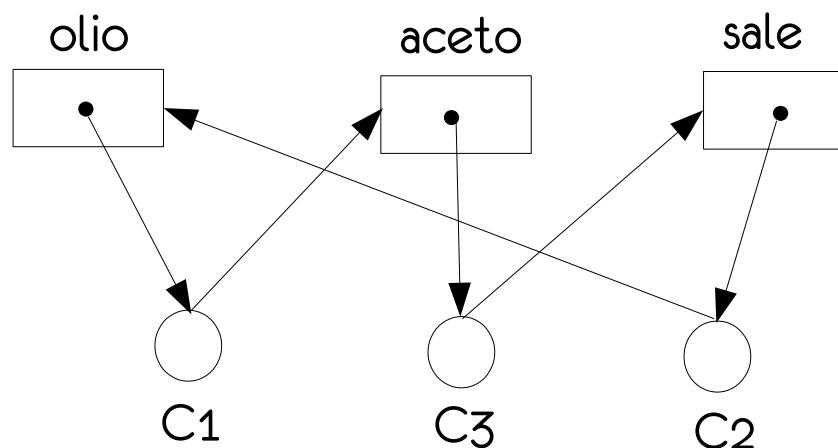
rilascia risorse

Cuoco3

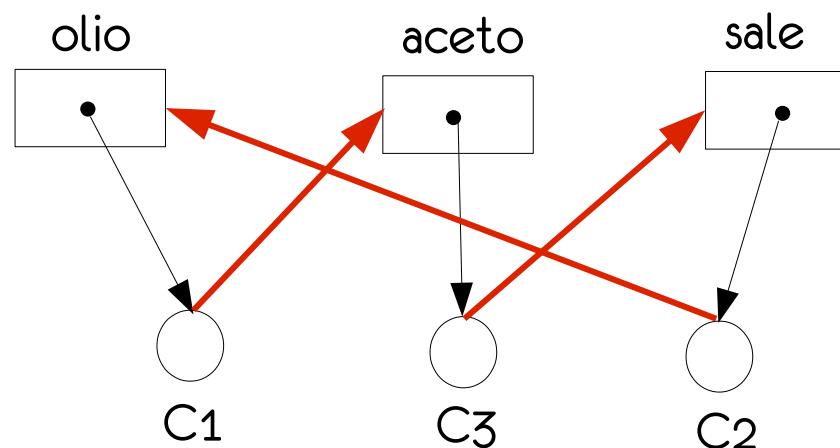
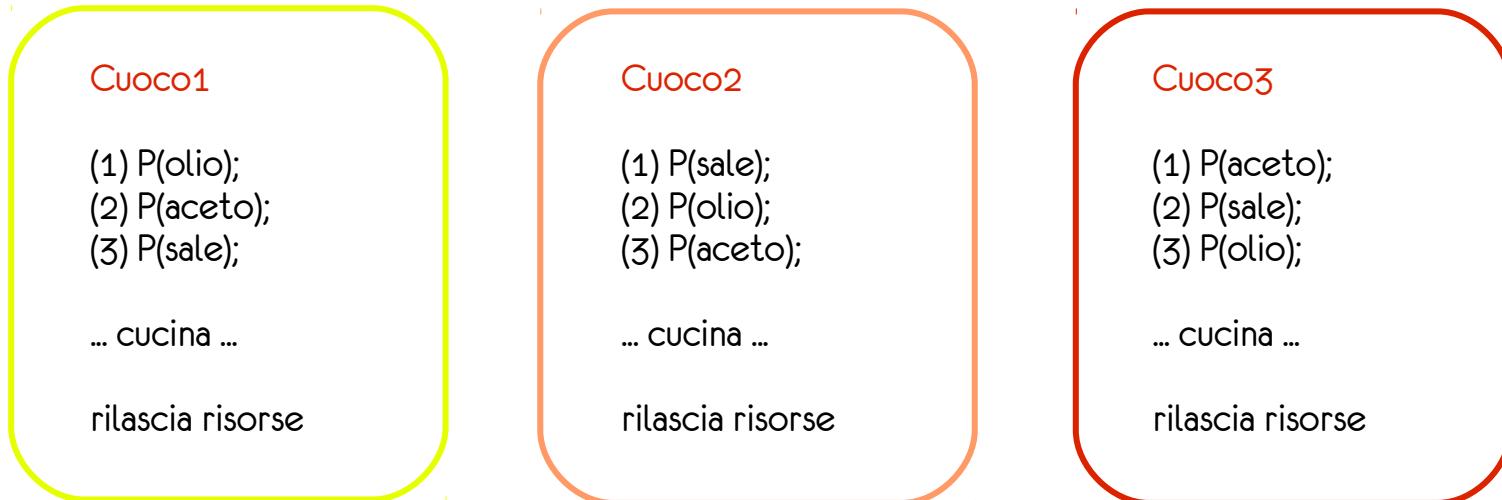
- (1) P(aceto);
- (2) P(sale);
- (3) P(olio);

... cucina ...

rilascia risorse



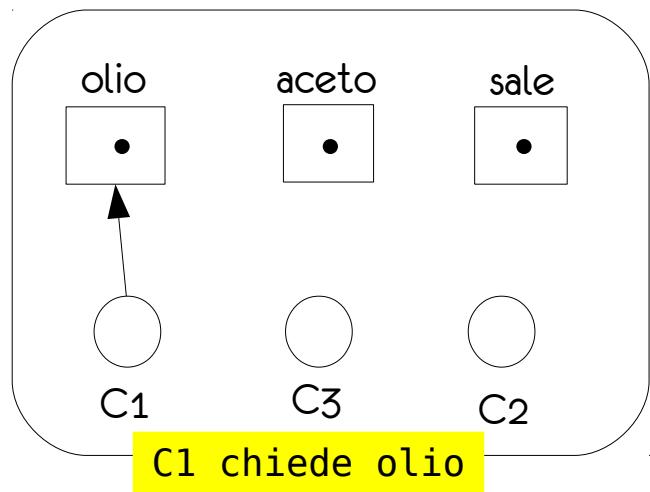
I 3 cuochi



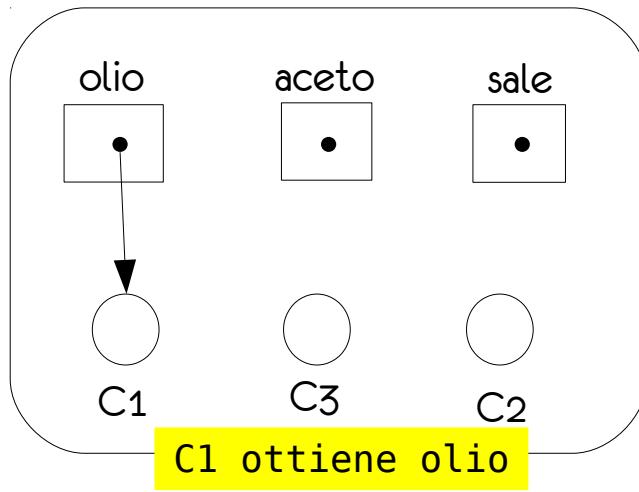
C1 aspetta C3, che aspetta C2, che aspetta C1: **deadlock**

I 3 cuochi

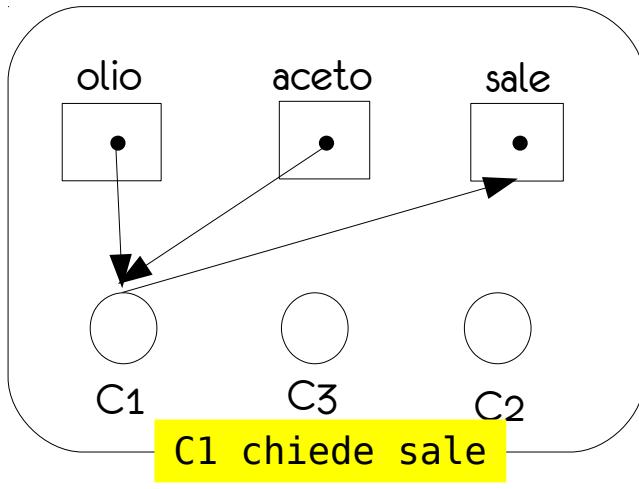
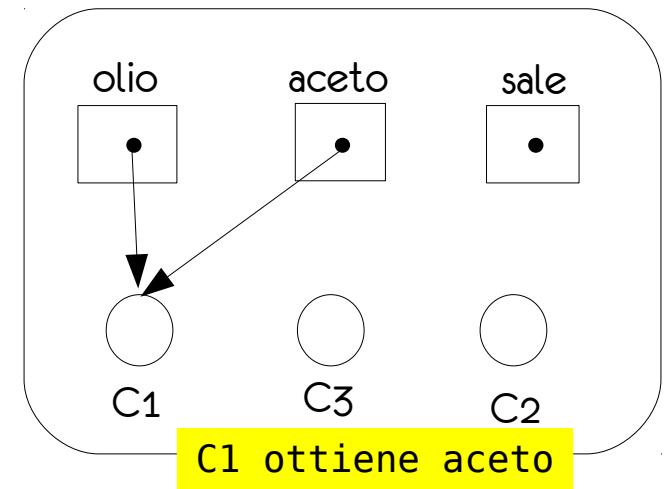
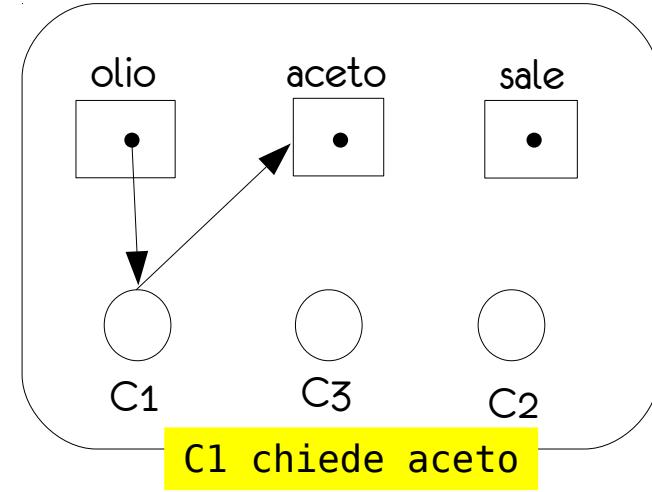
1



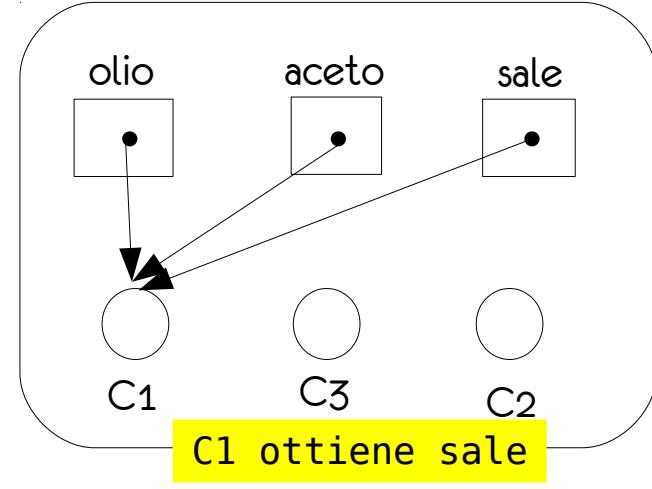
2



3



5



6

eccetera ... esecuzione senza deadlock

Uso del grafo di assegnazione

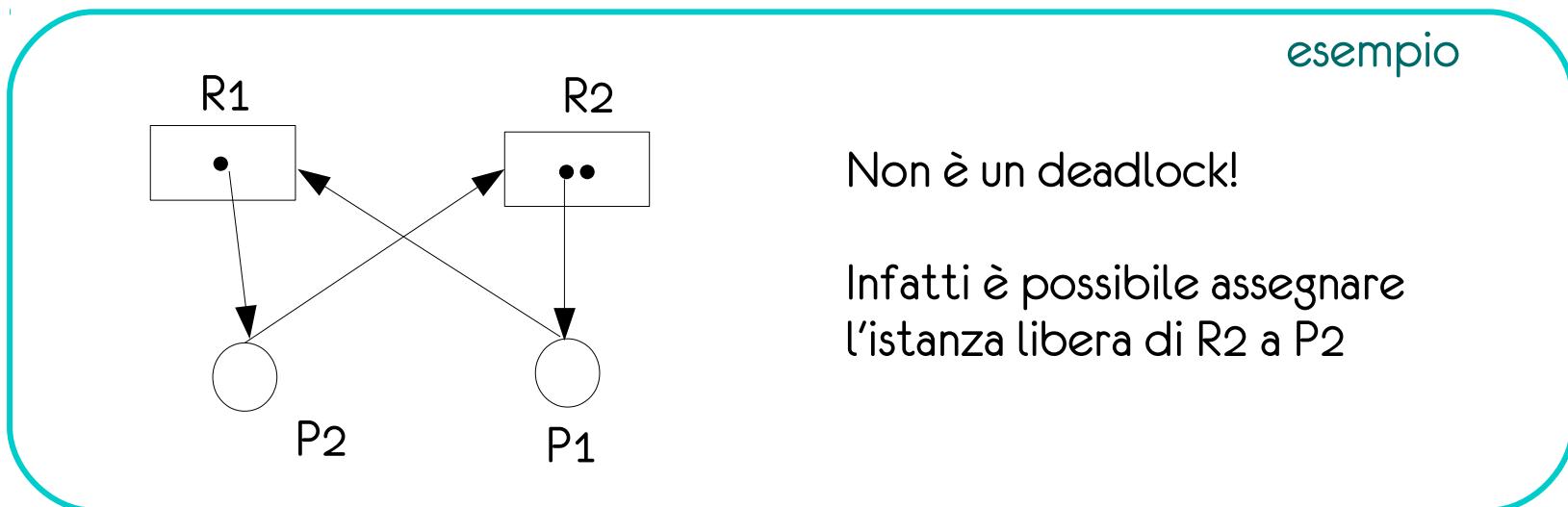
- Se il grafo **non** contiene cicli **non** c'è deadlock
- La presenza di un ciclo è condizione **necessaria** ma **non sufficiente** per avere deadlock

Uso del grafo di assegnazione

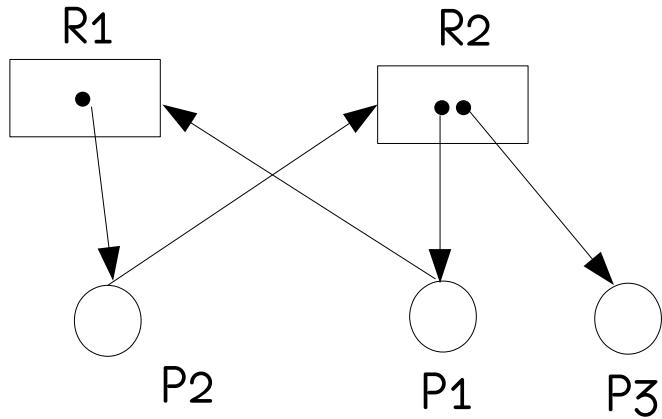
- Se il grafo **non** contiene cicli **non** c'è deadlock
- La presenza di un ciclo è condizione **necessaria ma non sufficiente** per avere deadlock:
 - Se il grafo contiene un ciclo che comprende risorse aventi tutte una sola istanza, allora c'è deadlock
 - Se il ciclo comprende risorse aventi più di una istanza non è detto che vi sia deadlock: **ciclo come risorsa necessaria ma non sufficiente.** Basta che una delle richieste sia soddisfacibile per rompere il ciclo

Uso del grafo di assegnazione

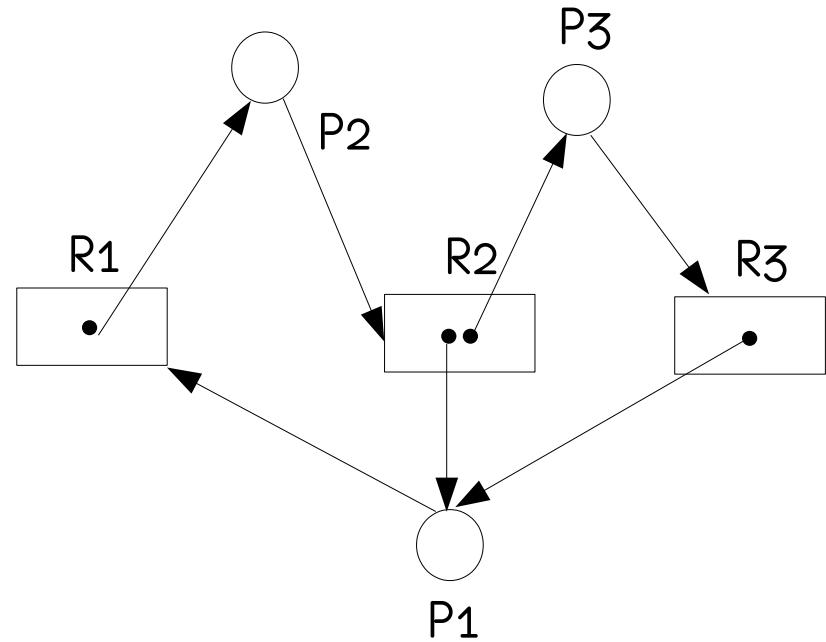
- Se il ciclo comprende risorse aventi più di una istanza non è detto che vi sia deadlock: **ciclo come risorsa necessaria ma non sufficiente**. Basta che una delle richieste sia soddisfacibile per rompere il ciclo



Altri esempi



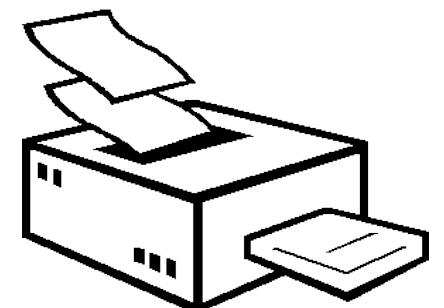
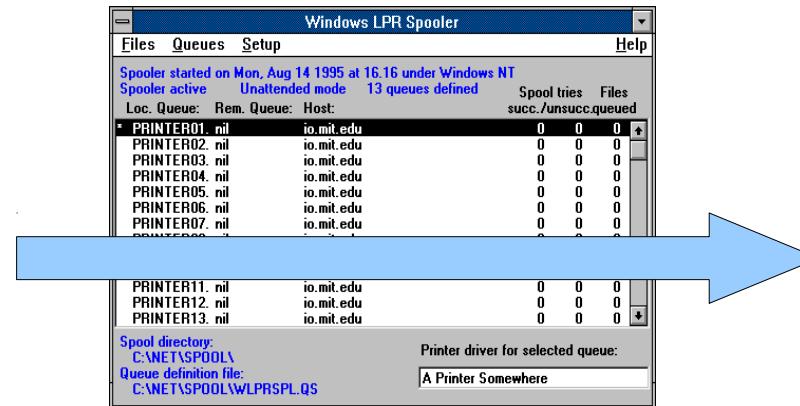
Deadlock? No
Appena P3 libera R2
P2 riparte



Deadlock? Si
P1 aspetta R1 che è di P2
P2 aspetta R2, le cui istanze
sono di P1 e P3 e
P3 aspetta R3 che è di P1

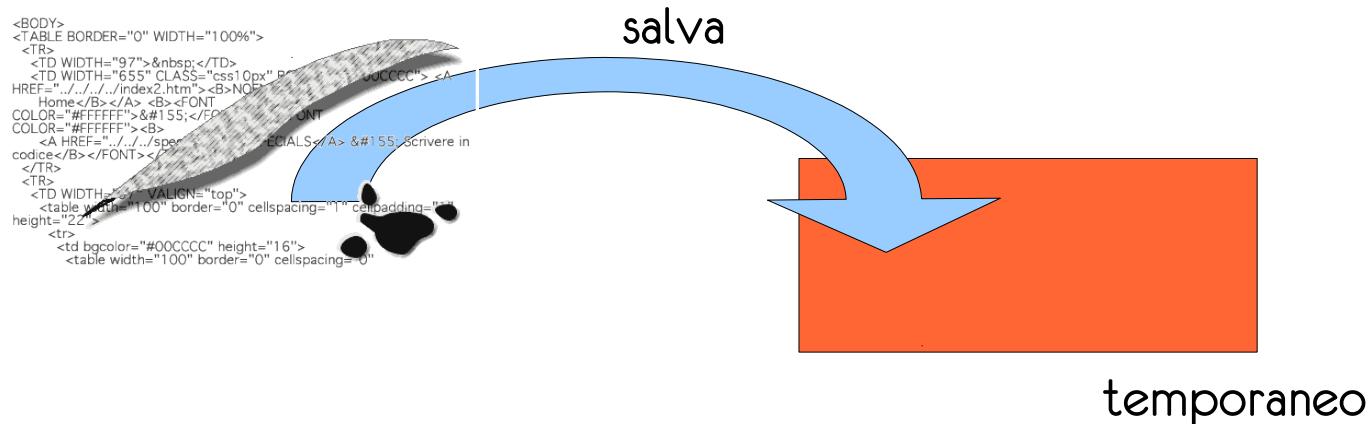
Un caso reale: spooling

- Un processo deve stampare un documento: la stampante gestisce una pagina per volta, quindi il processo dovrebbe attendere la terminazione della stampa della pagina corrente prima dell'invio alla stampante della successiva
- Problema: lentezza!!
- Per migliorare l'esecuzione del processo utente si introduce uno spooler



Spooler

- Uno spooler è un programma che funge da intermediario fra i processi di stampa e la stampante (o altri generi di device)
- Un processo di stampa può terminare subito dopo aver inviato il proprio documento allo spooler
- L'invio può avvenire in modi diversi:
 - caso (1): il documento viene salvato in un file temporaneo poi elaborato dallo spooler



Scenario

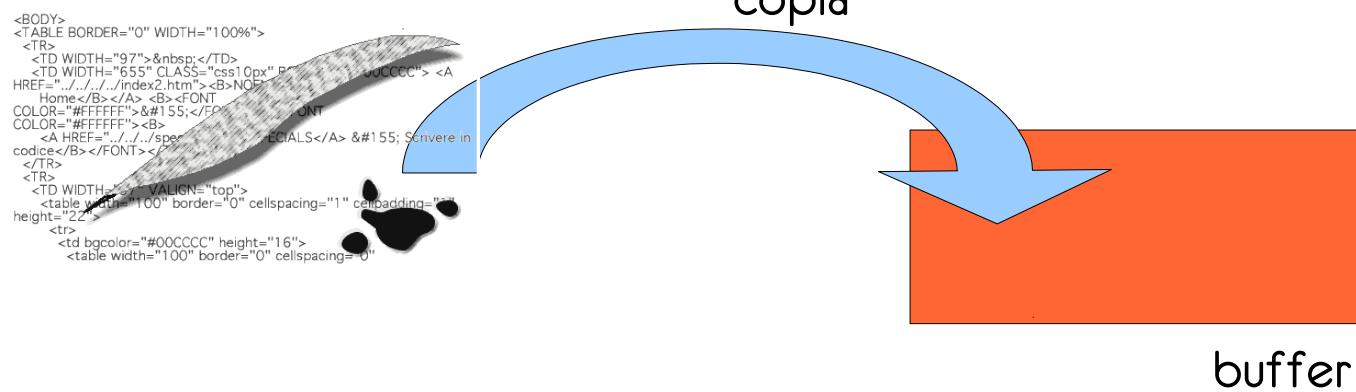
- Molti sistemi di spooling gestiscono solo documenti completamente codificati
- Quindi un processo di stampa deve per es.:
 - convertire il documento in un formato di stampa
 - salvare il risultato in un file temporaneo
- **Risorsa condivisa: memoria**
- Cosa succede se si esaurisce la memoria e nessun processo di stampa ha terminato il proprio lavoro?

Scenario

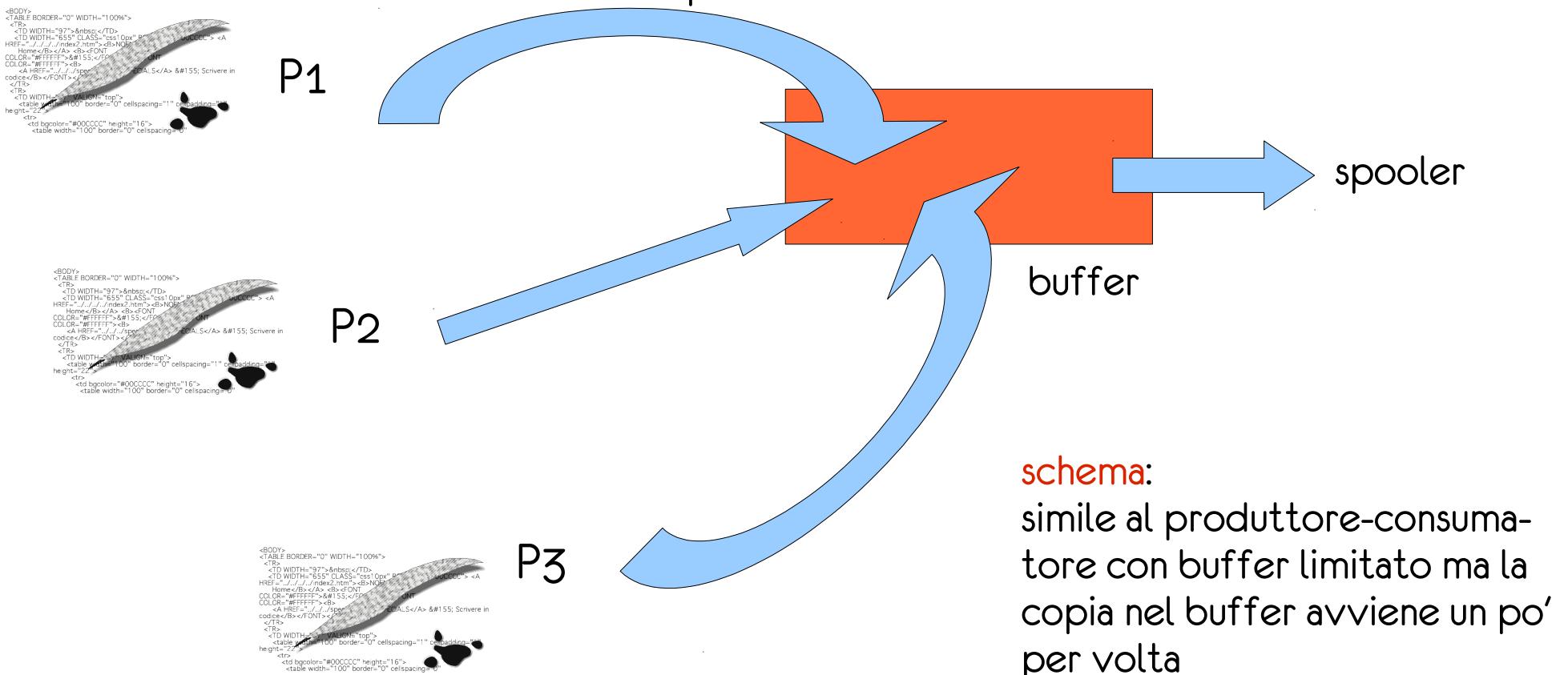
- Molti sistemi di spooling gestiscono solo documenti prodotti in modo completo
- Quindi un processo di stampa deve per es.:
 - convertire il documento in un formato di stampa
 - salvare il risultato in un file temporaneo
- **Risorsa condivisa: memoria**
- Cosa succede se si esaurisce la memoria e nessun processo di stampa ha terminato il proprio lavoro?
Deadlock!
 - I processi di stampa **acquisiscono la memoria un po' per volta**, quindi attendono la memoria mancante
 - lo spooler potrebbe liberare la memoria processando un documento ma non lo fa perché **nessun documento è completo**

Spooler

- Consideriamo un'implementazione alternativa dello spooler che diventa ...
- ... un programma che funge da intermediario fra i processi di stampa e la stampante, legge da un'area di memoria predefinita e di dimensione limitata (buffer) i documenti da stampare e interagisce con la stampante.
- Un processo di stampa può terminare dopo aver inviato il proprio documento allo spooler
- L'invio avviene in questo modo, caso (2): il documento viene copiato nel buffer man mano che viene generato



Scenario



Scenario

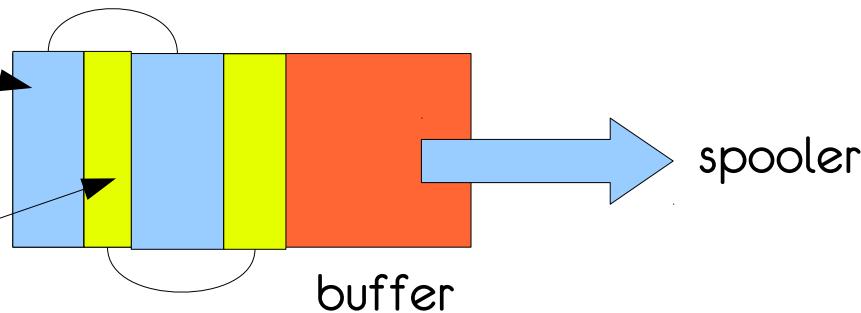
P1

```
<BODY>
<TBODY>
<TR>
<TD WIDTH="97">&nbsp;</TD>
<TD WIDTH="655" CLASS="cst1Opz" style="background-color: #00CCCC;"> <A href="http://www.unicam.it/~b-NCS/index.htm" style="color: #FFFFFF; font-size: 10pt; font-weight: bold; text-decoration: none; font-family: Arial, Helvetica, sans-serif; font-style: italic;">Home</A></TD>
<TD WIDTH="100" style="background-color: #00CCCC; text-align: right; vertical-align: bottom; font-size: 10pt; font-weight: bold; color: #FFFFFF; font-family: Arial, Helvetica, sans-serif; font-style: italic;">Scrivere in  
codice</TD>
<TR>
<TD WIDTH="100" style="vertical-align: top">
<table width="100" border="0" cellspacing="1" cellpadding="0" style="border-collapse: collapse; border: 1px solid black; background-color: #00CCCC; height: 16">
<tr>
<td bgcolor="#00CCCC" style="width: 100%; height: 16; border: 0; padding: 0; margin: 0; font-size: 10pt; font-weight: bold; color: #FFFFFF; font-family: Arial, Helvetica, sans-serif; font-style: italic;">Scrivere in  
codice</td>
</tr>
</table>
</TD>
<TD style="background-color: #FF0000; text-align: center; vertical-align: middle; width: 100px; height: 16px; border: 1px solid black; border-radius: 5px; font-size: 10pt; font-weight: bold; color: #FFFFFF; font-family: Arial, Helvetica, sans-serif; font-style: italic;">Scrivere in  
codice</TD>
</TR>
</TBODY>
</BODY>
```

P2

```
<BODY>
<TBODY>
<TR>
<TD WIDTH="37">&nbsp;</TD>
<TD WIDTH="655" CLASS="cst1Opz" style="background-color: #00CCCC;"> <A href="http://www.unicam.it/~b-NCS/index2.htm" style="color: #FFFFFF; font-size: 10pt; font-weight: bold; text-decoration: none; font-family: Arial, Helvetica, sans-serif; font-style: italic;">Home</A></TD>
<TD WIDTH="100" style="background-color: #00CCCC; text-align: right; vertical-align: bottom; font-size: 10pt; font-weight: bold; color: #FFFFFF; font-family: Arial, Helvetica, sans-serif; font-style: italic;">Scrivere in  
codice</TD>
<TR>
<TD WIDTH="100" style="vertical-align: top">
<table width="100" border="0" cellspacing="1" cellpadding="0" style="border-collapse: collapse; border: 1px solid black; background-color: #00CCCC; height: 16">
<tr>
<td style="width: 100%; height: 16; border: 0; padding: 0; margin: 0; font-size: 10pt; font-weight: bold; color: #FFFFFF; font-family: Arial, Helvetica, sans-serif; font-style: italic;">Scrivere in  
codice</td>
</tr>
</table>
</TD>
<TD style="background-color: #FF0000; text-align: center; vertical-align: middle; width: 100px; height: 16px; border: 1px solid black; border-radius: 5px; font-size: 10pt; font-weight: bold; color: #FFFFFF; font-family: Arial, Helvetica, sans-serif; font-style: italic;">Scrivere in  
codice</TD>
</TR>
</TBODY>
</BODY>
```

Problema: cosa succede se la porzione di buffer libera non basta a contenere nessuna delle porzioni di documento prodotte dai processi di stampa?



ancora una volta: **deadlock**

Nota

La discontinuità delle aree di memoria contenenti l'output di un processo di stampa non è un problema. Quando studieremo la memoria vedremo che si tratta della norma: i file sono memorizzati in aree discontinue, il SO ha strutture adeguate a mantenere/ricostruire la struttura logica e sequenziale dei file

Scenario

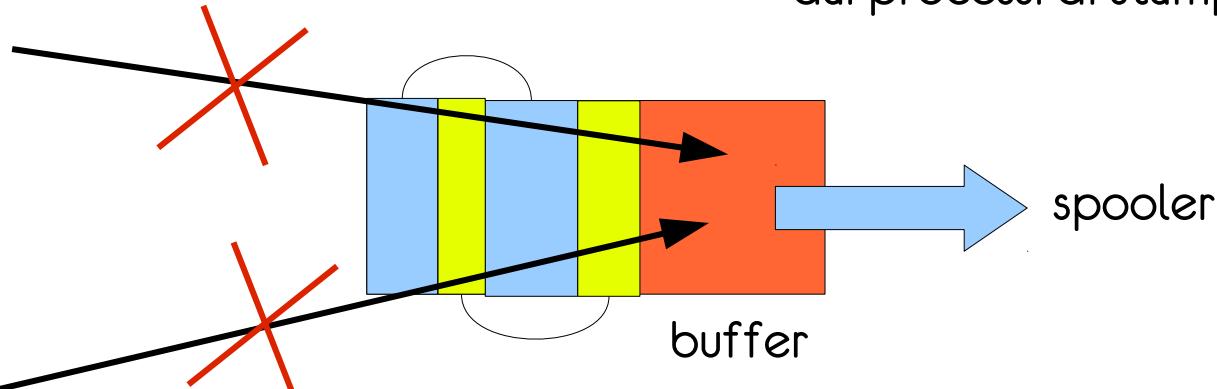
P1

```
<BODY>
<TABLE BORDER="0" WIDTH="100%">
<TR>
<TD WIDTH="97">&nbsp;</TD>
<TD WIDTH="655" CLASS="cst1px" BORDER="1" style="background-color:#00CCCC;"> <A
HREF="#">Home</A></TD><TD><A href="#">Scritta</A><br/><A href="#">Copia</A>
<A href="#">Stampa</A></TD>
<TD align="right" style="background-color:#00CCCC;">Scrivere in
codice</TD></TR>
<TR>
<TD align="center" VALIGN="top">
<table border="0" border="0" cellspacing="1" cellpadding="0">
<tr>
<td bordercolor="#00CCCC" height="16">
<table width="100" border="0" cellspacing="0">
```

P2

```
<BODY>
<TABLE BORDER="0" WIDTH="100%">
<TR>
<TD WIDTH="97">&nbsp;</TD>
<TD WIDTH="655" CLASS="cst1px" BORDER="1" style="background-color:#00CCCC;"> <A
HREF="#">Home</A></TD><TD><A href="#">Scritta</A><br/><A href="#">Copia</A>
<A href="#">Stampa</A></TD>
<TD align="right" style="background-color:#00CCCC;">Scrivere in
codice</TD></TR>
<TR>
<TD align="center" VALIGN="top">
<table border="0" border="0" cellspacing="1" cellpadding="0">
<tr>
<td bordercolor="#00CCCC" height="16">
<table width="100" border="0" cellspacing="0">
```

Problema: cosa succede se la porzione di buffer libera non basta a contenere nessuna delle porzioni di documento prodotte dai processi di stampa?



ancora una volta: **deadlock**

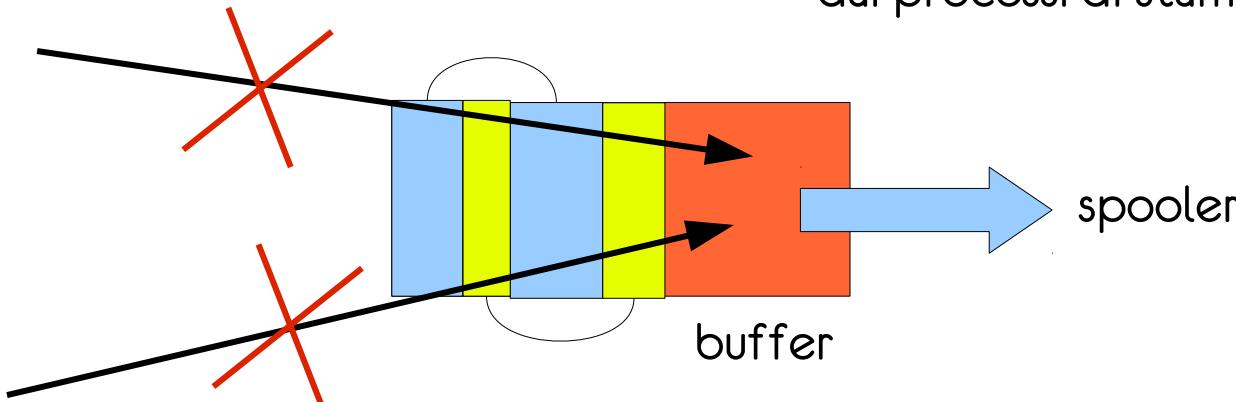
Il deadlock è causato da: (1) attesa circolare fra i processi di stampa, (2) possesso e attesa di porzioni di memoria, (3) mutua esclusione nell'uso di un'area di memoria specifica, (4) no prelazione

Scenario

P1

```
<BODY>
<TABLE BORDER="0" WIDTH="100%>
<TR>
<TD WIDTH="37">&nbsp;</TD>
<TD WIDTH="56%" CLASS="Orp" style="background-color: #CCCCFF; color: #00008B; font-size: 10pt; font-weight: bold; text-align: center;">
Home </a> </td> </tr>
<tr>
<td width="100%" border="0" cellspacing="1" cellpadding="0" style="background-color: #CCCCFF; color: #00008B; font-size: 10pt; font-weight: bold; text-align: center;">
<form>
<table border="1" style="width: 100%; border-collapse: collapse; border: none; margin-bottom: 10px; height: 16px;">
<tr>
<td border="0" style="width: 100%; border: none; background-color: #00CCCC; height: 16px;">
<table width="100%" border="0" cellspacing="0" border="0">

```



P2

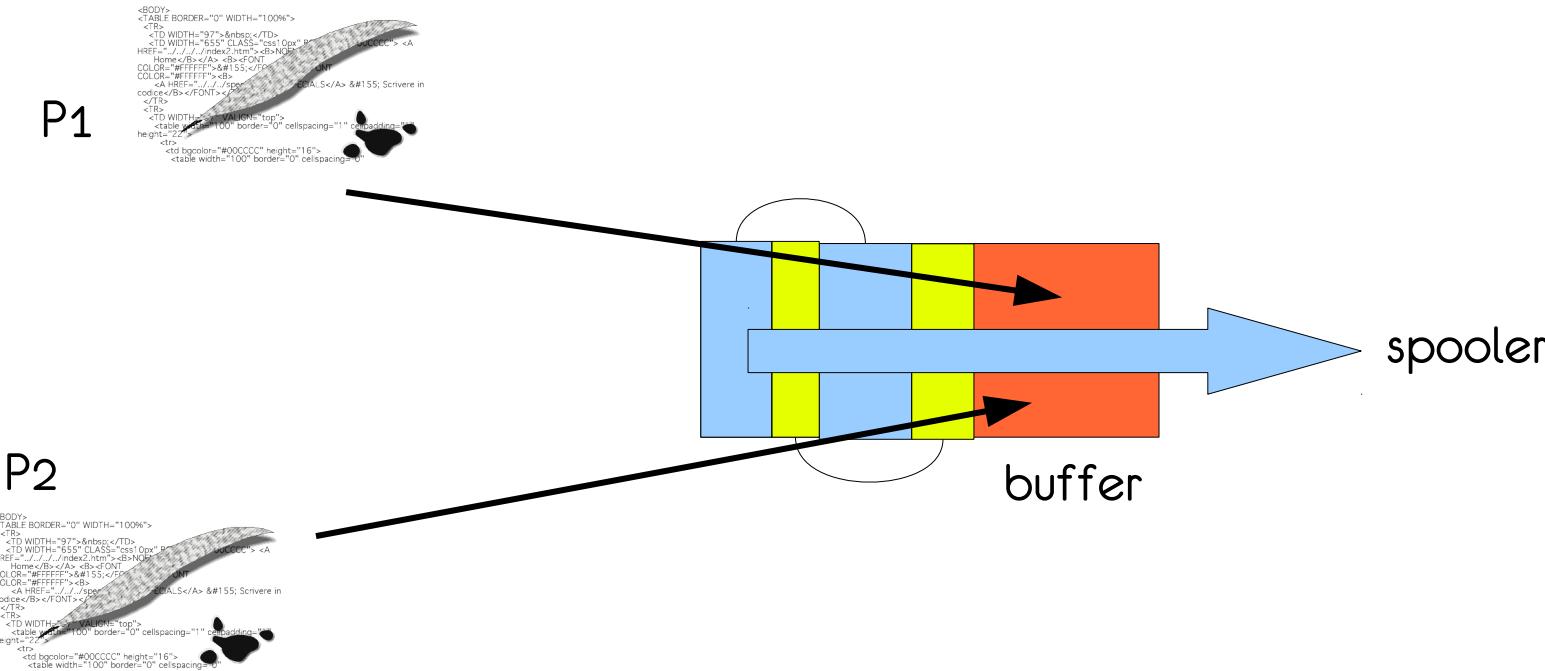
```
<BODY>
<TABLE BORDER="0" WIDTH="100%>
<TR>
<TD WIDTH="97%">&nbsp;</TD>
<TD WIDTH="3%" style="background-color: #CCCCFF; color: #00008B; font-size: 10pt; font-weight: bold; text-align: center;">
Home </a> </td> </tr>
<tr>
<td width="100%" border="0" cellspacing="1" cellpadding="0" style="background-color: #CCCCFF; color: #00008B; font-size: 10pt; font-weight: bold; text-align: center;">
<form>
<table border="1" style="width: 100%; border-collapse: collapse; border: none; margin-bottom: 10px; height: 16px;">
<tr>
<td border="0" style="width: 100%; border: none; background-color: #00CCCC; height: 16px;">
<table width="100%" border="0" cellspacing="0" border="0">
```

Problema: cosa succede se la porzione di buffer libera non basta a contenere nessuna delle porzioni di documento prodotte dai processi di stampa?

ancora una volta: **deadlock**

Può servire una politica del tipo: se il buffer è pieno al 75% nessun nuovo processo di stampa può iniziare a copiare un nuovo documento nel buffer?
No, perché il deadlock non è necessariamente causato da un nuovo processo possono bastare quelli già attivi

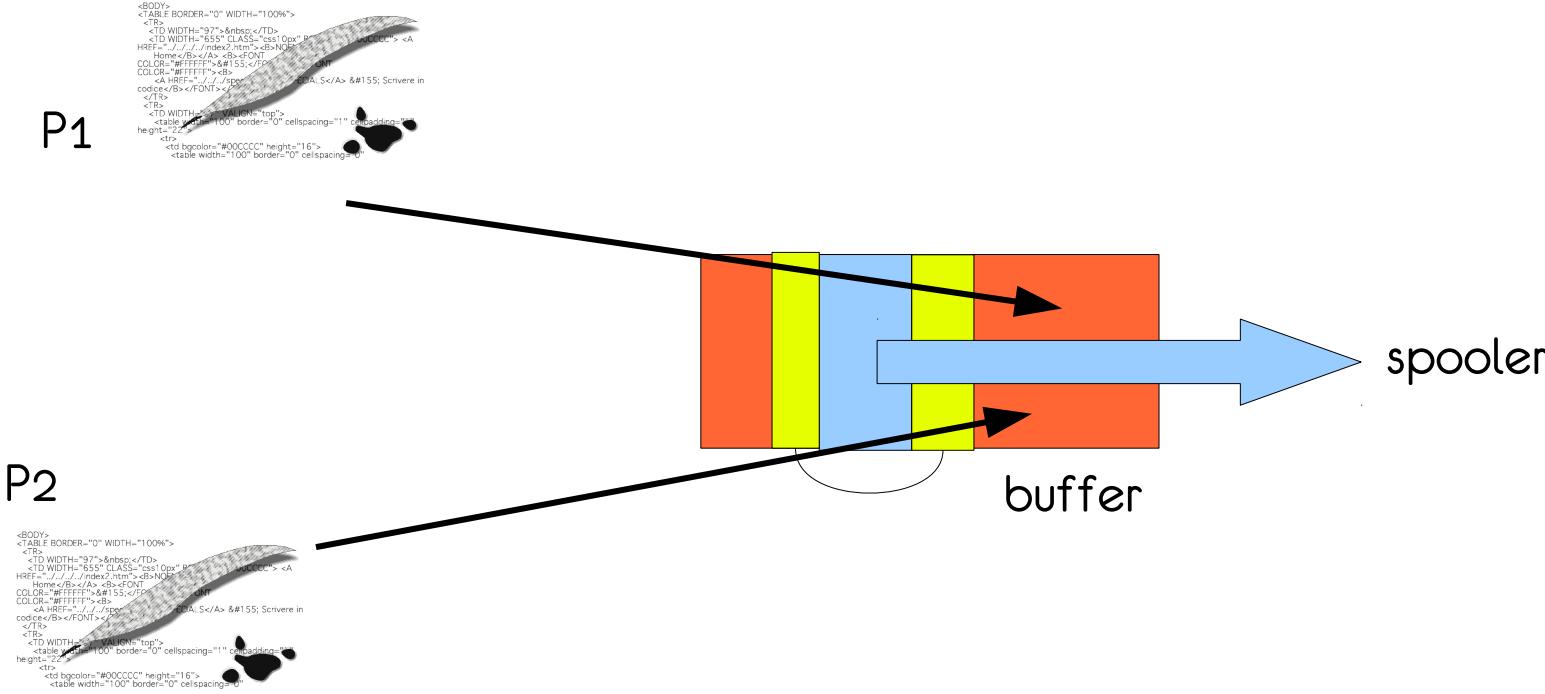
Scenario



Soluzione: **streaming**

lo spooler sottrae a un processo di stampa la memoria che ha acquisito e usato svuotandola, cioè riversandone il contenuto sulla stampante: **prelazione della risorsa memoria**

Scenario



Soluzione: **streaming**

lo spooler sottrae a un processo di stampa la memoria che ha acquisito e usato svuotandola, cioè riversandone il contenuto sulla stampante

Che fare col deadlock?

- **Rilevare il deadlock:**



- è una capacità fondamentale se non abbiamo metodi che a priori ne evitano il generarsi

- **Rompere il deadlock quando si presenta:**



- richiede la capacità di monitorare le richieste/assegnazioni di risorse

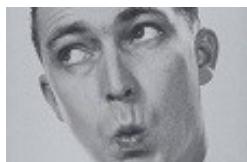
- **Prevenire il deadlock:**



- occorre definire opportuni protocolli di assegnazione delle risorse

- **Far finta che il deadlock sia impossibile:**

- è la tecnica più usata, poco costosa perché non richiede né risorse aggiuntive né l'attuazione di politiche particolari



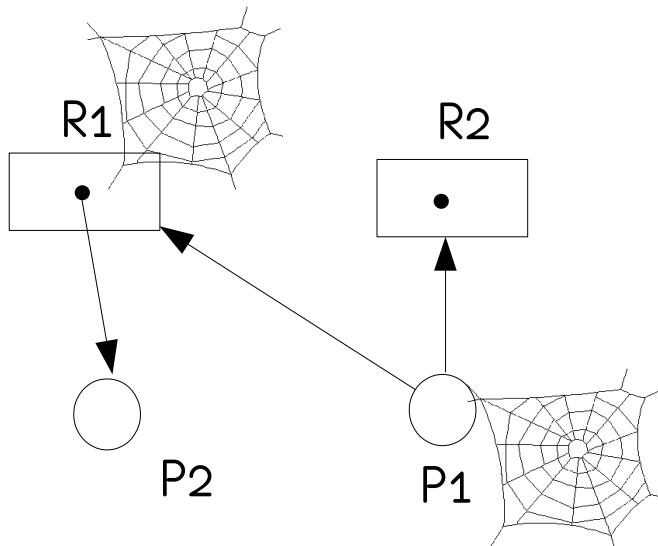
Prevenzione

Rendere impossibile una delle 4 condizioni necessarie al deadlock

- **Mutua Esclusione**: la richiesta di usare le risorse in ME può essere rilasciata solo per alcuni tipi di risorse, es. file aperti in lettura, altre sono intrinsecamente ME, es. CD writer (due processi non possono scrivere sullo stesso CD contemporaneamente)
- **Possesso e attesa**:
 - **possibile strategia**: se un processo ha bisogno di più risorse non può accumularle un po' per volta, o le ottiene tutte insieme o non ne prende nessuna. Nota: Occorre evitare starvation.
- **Consentire la prelazione**:
 - **possibile strategia**: un processo che ha N risorse e ne richiede un'altra o la ottiene subito o (se occorre attendere) rilascia tutte le risorse in suo possesso
- **Attesa circolare**:
 - **possibile strategia**: imporre un ordinamento delle risorse e dei processi

Prima strategia di Havender

- Protocollo di richiesta delle risorse: **tutte le risorse necessarie ad un processo devono essere richieste insieme**
 - se sono tutte disponibili**, il sistema le assegna e il processo prosegue
 - se anche solo una non è disponibile** il processo non ne acquisisce nessuna e si mette in attesa
- Vantaggio**: previene il deadlock
- Svantaggio**: spreco di risorse, ad esempio:



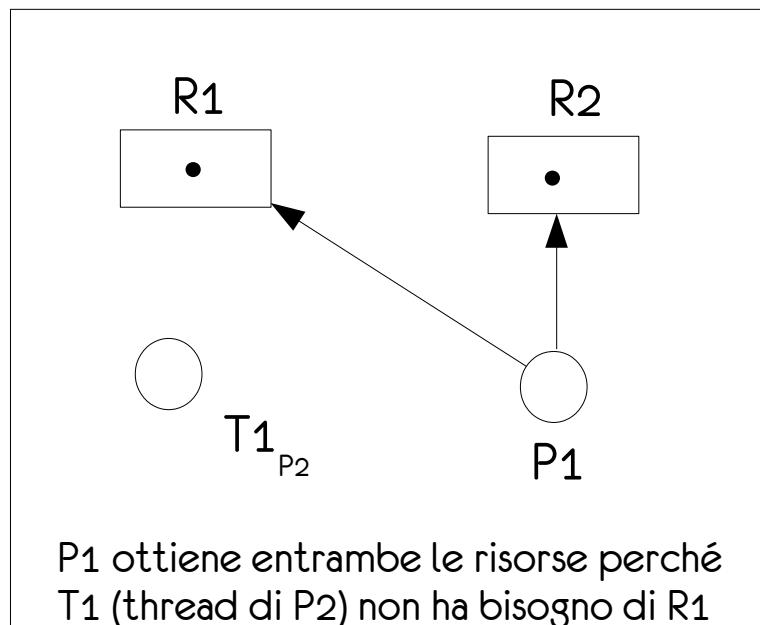
P1 richiede sia R1 che R2 ma l'unica istanza di R1 è assegnata a P2

P2 ha dovuto richiedere R1 ma l'userà solo al termine del proprio lavoro (3 ore dopo!!)

R2 è libera, P1 potrebbe usare sia R1 che R2 prima che a P2 occorra effettivamente R1

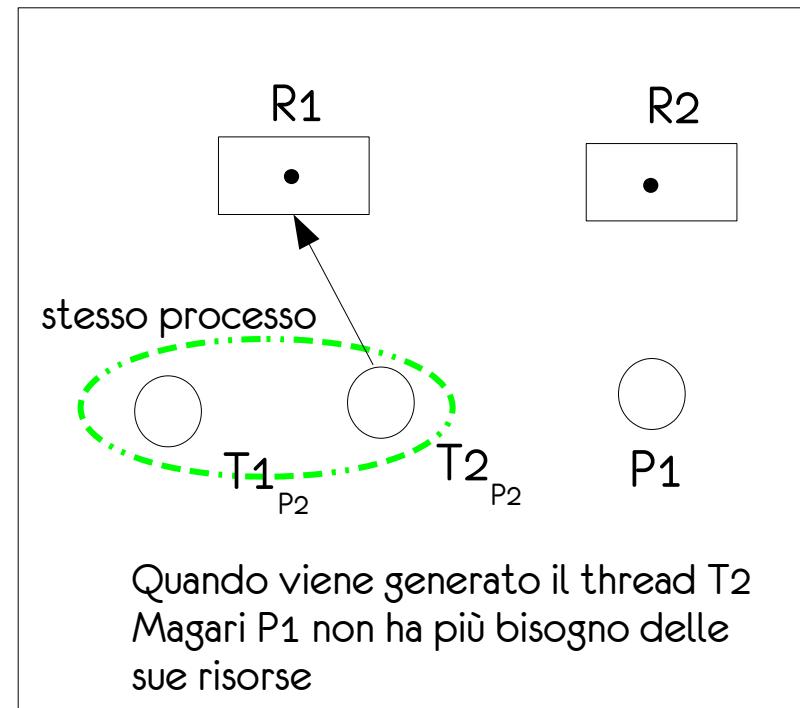
Possibile miglioramento

- Questaa strategia non funziona per **processi heavyweight** però se all'interno del processo riusciamo a distinguere più **thread** di esecuzione, ciascuno dei quali ha bisogno di un sottoinsieme delle risorse ed è generato solo quando occorre ... la strategia può risultare efficace



istante 1

P1 ottiene entrambe le risorse perché
T1 (thread di P2) non ha bisogno di R1



istante 2

Quando viene generato il thread T_2
Magari P1 non ha più bisogno delle
sue risorse

Nota: in generale un processo che richiede molte risorse può essere soggetto a starvation

Seconda strategia di Havender

- **Consentire la prelazione delle risorse**
- Se la prima strategia di Havender non è applicata, un processo potrebbe accumulare risorse via via.
- Supponiamo che a un certo punto il processo effettui una richiesta non esaudibile perché le risorse sono esaurite:
 - il processo non può eseguire il proprio compito ma ...
 - ... se *non rilascia le risorse accumulate* neanche gli altri processi potranno lavorare!!

Seconda strategia di Havender

- **Consentire la prelazione delle risorse**
- Se la prima strategia di Havender non è applicata, un processo potrebbe accumulare risorse via via.
- Supponiamo che a un certo punto il processo effettui una richiesta non esaudibile perché le risorse sono esaurite:
 - il processo non può eseguire il proprio compito ma ...
 - ... se *non rilascia le risorse accumulate* neanche gli altri processi potranno lavorare!!
- **Seconda strategia di Havender**: quando un processo richiede una risorsa che gli viene negata, rilascia tutte le risorse accumulate fino a quel momento
- eventualmente il processo effettua subito dopo una nuova richiesta di tutte le risorse che ha appena perso + quella che non è riuscito ad ottenere

Critica

- È una tecnica **costosa**: perdere delle risorse può significare perdere un lavoro già compiuto in parte (es. se mi viene tolta della memoria perdo i dati eventualmente già inseriti in essa)
- Vale la pena solo se il sistema è tale per cui verrà applicata di rado
- Il suo uso in congiunzione a un **criterio di priorità** che predilige l'assegnazione di risorse a processi che ne richiedono poche, può causare la starvation di quei processi che hanno bisogno di molte risorse
- Inoltre **non tutte le risorse sono preemptible**: per esempio interrompere una stampa non è ragionevole

Attesa circolare

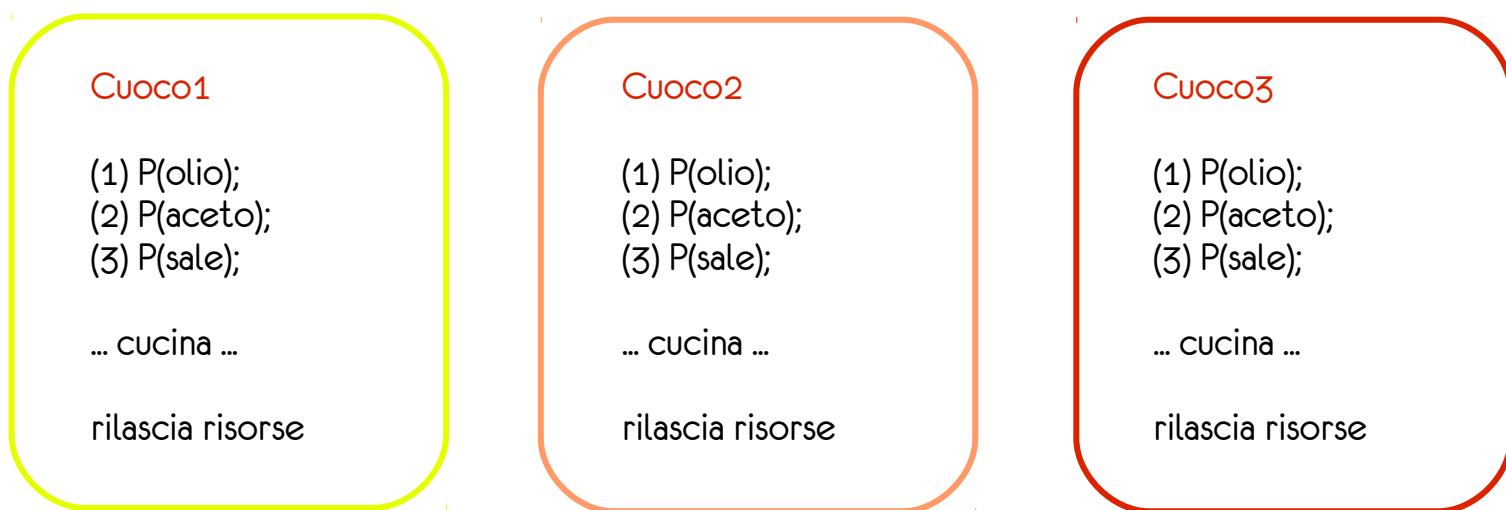
- L'ultima strategia di Havender comporta l'**avoidance dell'attesa circolare**
- Ogni **risorsa** ha assegnato un **numero**, utilizzato per quella risorsa soltanto
- Sulla base di tali numeri le risorse risultano ordinabili in ordine strettamente crescente (**R₁ < R₂ < ... < R_n**)

Attesa circolare

- L'ultima strategia di Havender comporta l'**avoidance dell'attesa circolare**
- Ogni **risorsa** ha assegnato un **numero**, utilizzato per quella risorsa soltanto
- Sulla base di tali numeri le risorse risultano ordinabili in ordine strettamente crescente ($R_1 < R_2 < \dots < R_n$)
- Un processo che abbia bisogno di M risorse le deve **richiedere in ordine crescente**, per esempio:
 - P ha bisogno di R_4 , R_7 ed R_2 allora richiede nell'ordine R_2 , quindi R_4 e infine R_7
- Non si può avere deadlock perché l'ordinamento delle richieste impedisce l'attesa circolare
- È stata usata in alcuni sistemi operativi ma non è molto **flessibile**: chi scrive programmi per il sistema deve essere consapevole dell'ordinamento imposto alle risorse

Esempio

- Proviamo ad applicare la terza strategia di Havender ai tre cuochi
- Numeriamo le risorse: `olio ← 1, aceto ← 2, sale ← 3`
- Tutti i processi che usano queste risorse le devono **richiedere rispettando l'ordinamento**: i tre cuochi diventeranno uguali, nella loro prima parte del programma



- A questo punto i tre cuochi competono per la risorsa olio, che verrà assegnata ad uno solo di loro, che potrà richiedere la risorsa aceto senza entrare in competizione con gli altri, ancora in attesa dell'olio. In breve uno dei processi otterrà tutte le risorse necessarie e non si avrà deadlock

Deadlock avoidance

- Non sempre è possibile inibire a priori una delle condizioni necessarie affinché si abbia il deadlock (applicando le strategie di Havender)
- **questo non significa che non si possa evitare il deadlock**
- i metodi che consentono di fare ciò richiedono alcune informazioni, per esempio che i processi dichiarino quante risorse di un certo tipo hanno bisogno
- L'algoritmo di **deadlock avoidance** esamina lo stato di allocazione delle risorse e garantisce che in futuro non si formeranno attese circolari

Deadlock avoidance

- In certi contesti non è possibile inibire a priori una delle condizioni necessarie affinché si abbia il deadlock (applicando le strategie di Havender)
- **Questo non significa che non si possa evitare il deadlock**
- I metodi che consentono di fare ciò **richiedono alcune informazioni**, per esempio che i processi dichiarino quante risorse di un certo tipo hanno bisogno

Deadlock avoidance

- Occorre introdurre due nozioni nuove:
 - <1> **stato (del sistema) sicuro**: si dice che il sistema è in uno stato sicuro (o safe) se il SO può garantire che ciascun processo completerà la propria esecuzione in un tempo finito

Stato di allocazione delle risorse

- Lo **stato di allocazione delle risorse** cattura il numero di risorse libere, di risorse allocate e se disponibile il numero di risorse ancora richiedibili
- Visualizzabile tramite una **tabella**
- Es. se ho **10 istanze di una classe di risorse R**, tre processi **P1, P2 e P3**, inoltre **P1 ha allocate 3 risorse e ne desidera ancora 2**, **P2 ne ha allocate 4 e ne desidera 1** e **P3 ne ha allocata 1 e ne desidera ancora 4**:

The diagram shows a table representing the state of allocated resources (A) and resources still needed (ancora da richiedere) for three processes (P1, P2, P3). The table has columns for Process (processi) and Resource (R). A pink arrow points to the first column (processi), and a green arrow points to the second column (R). A yellow callout bubble states: "è una fotografia dell'allocazione delle risorse in un certo istante". A red arrow labeled "free 2" points to the bottom row of the table, and another red arrow labeled "risorse libere" points to the value "free 2".

	A	R	ancora da richiedere
processi	P1 P2 P3	3 4 1	2 1 4

NB: un processo non può procedere se non ha tutte le risorse che gli servono

Stato di allocazione delle risorse

	A	R
P1	3	2
P2	4	1
P3	1	4

free 2

	A	R
P1	3	2
P2	5	0
P3	1	4

free 1

P2 ha ricevuto la risorsa mancante
e può procedere

	A	R
P1	5	0
P2	4	1
P3	1	4

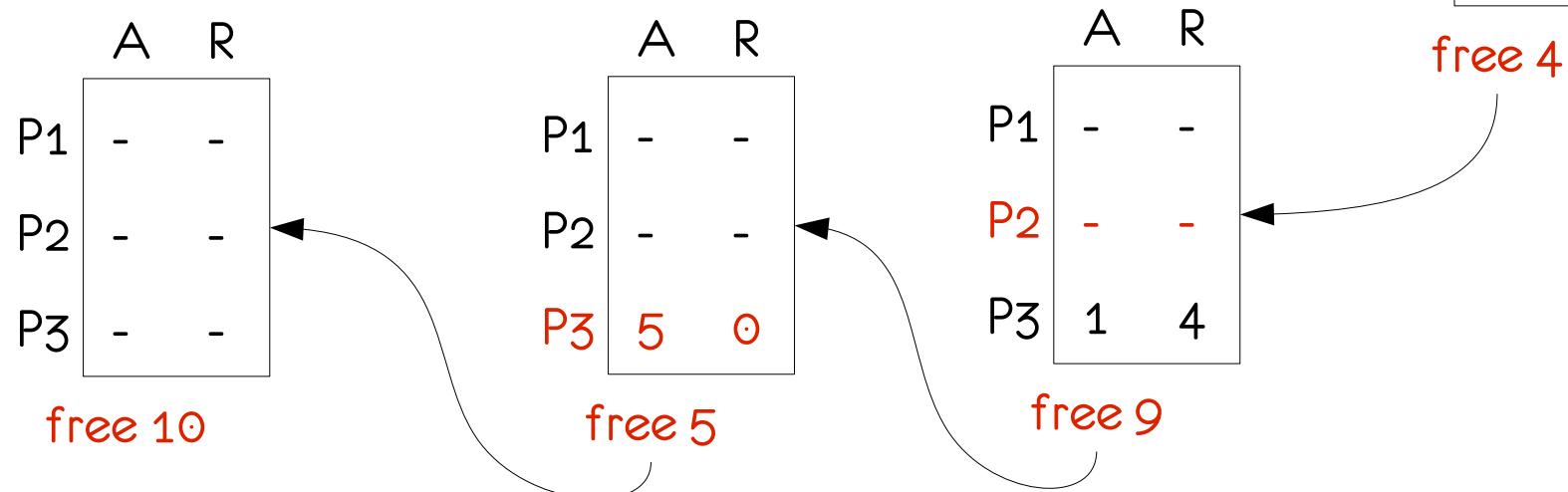
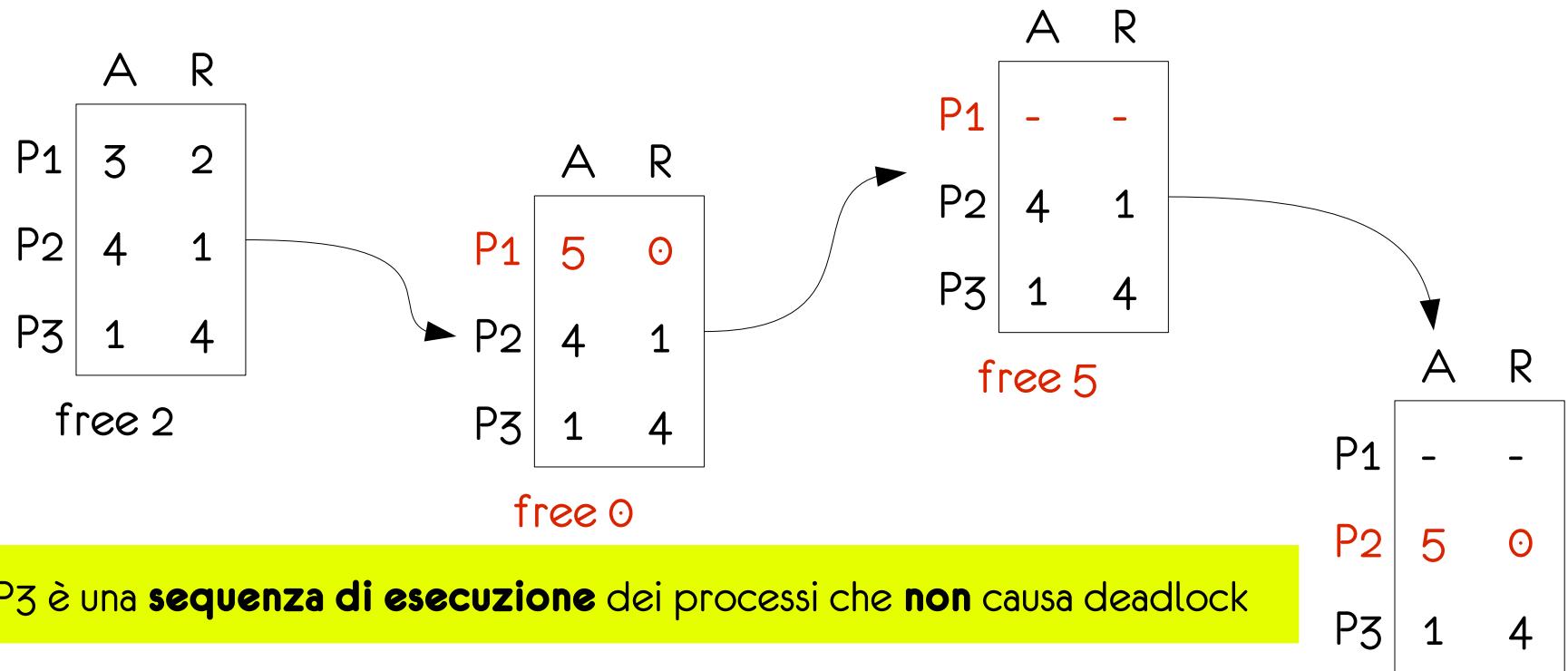
free 0

P1 ha ricevuto le risorse mancanti
e può procedere

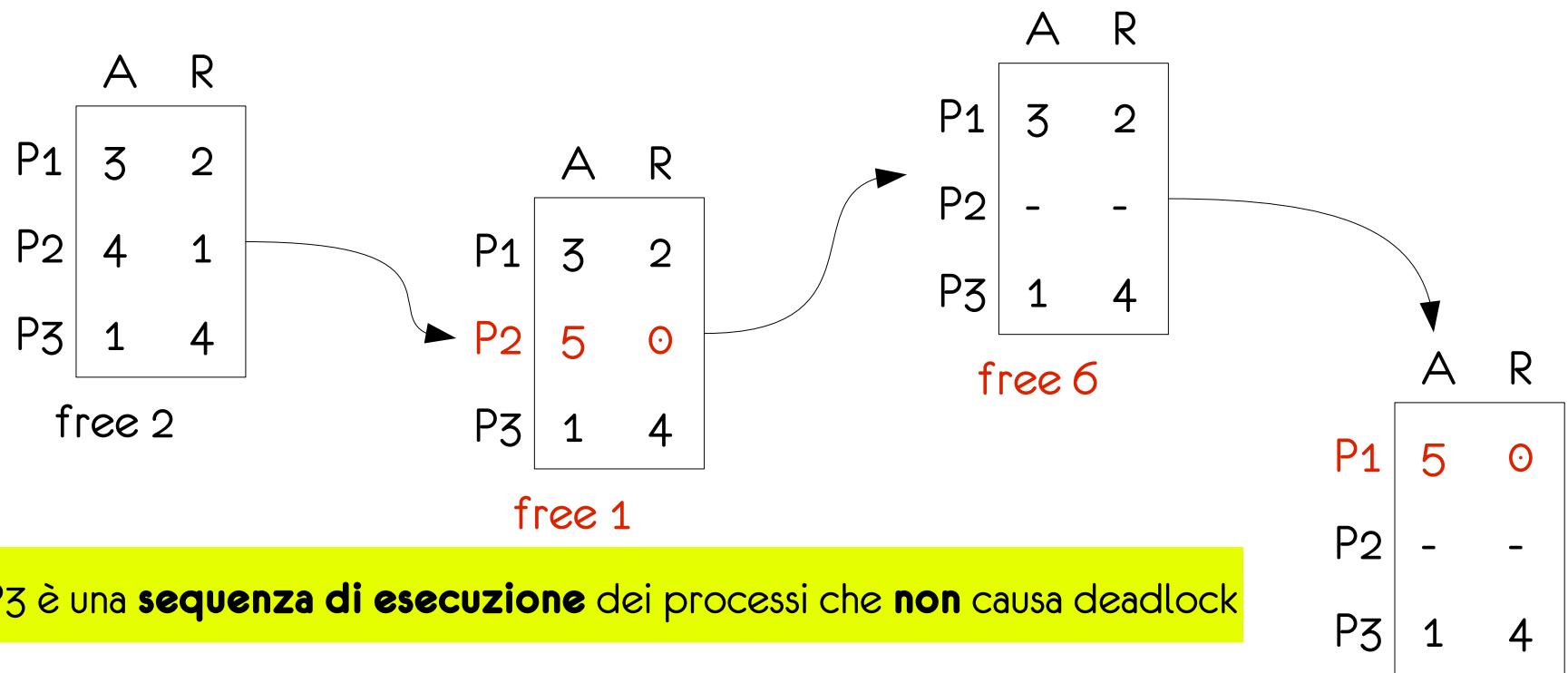
lo stato si evolve a seconda dell'
esecuzione successiva. A seconda
del processo che verrà eseguito
si possono avere diverse evoluz.

**NB: se un processo richiede n
risorse, il SO può decidere di
assegnargliene m<n**

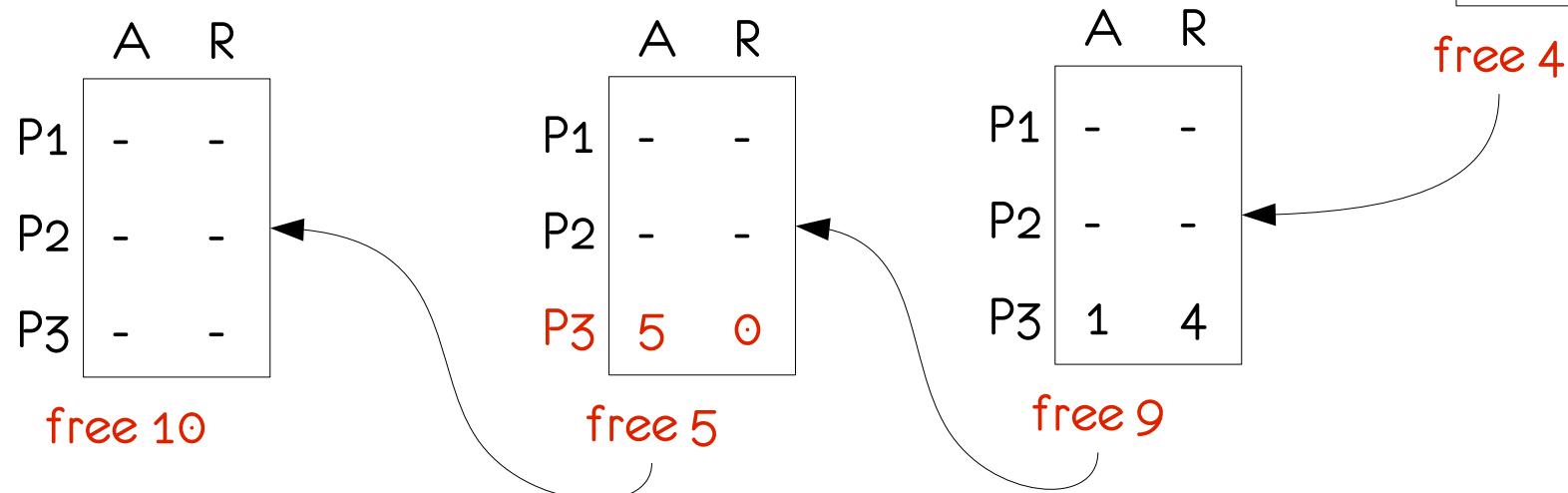
Sequenza di esecuzione



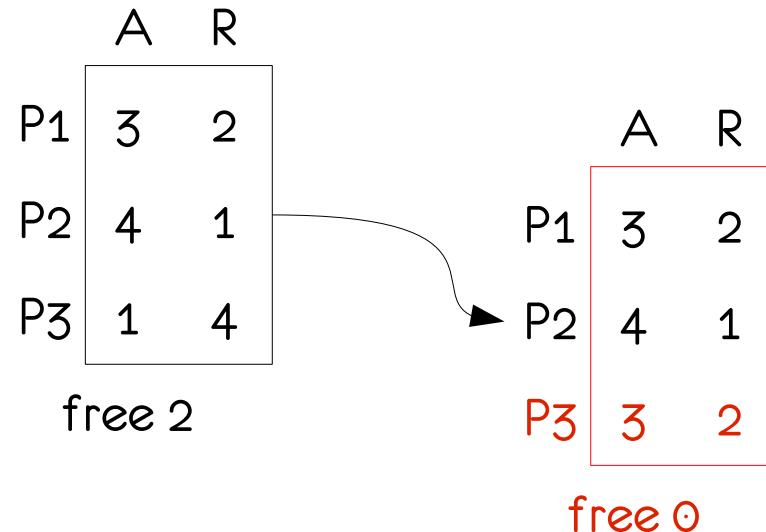
Altra sequenza di esecuzione



P2, P1, P3 è una **sequenza di esecuzione** dei processi che **non** causa deadlock

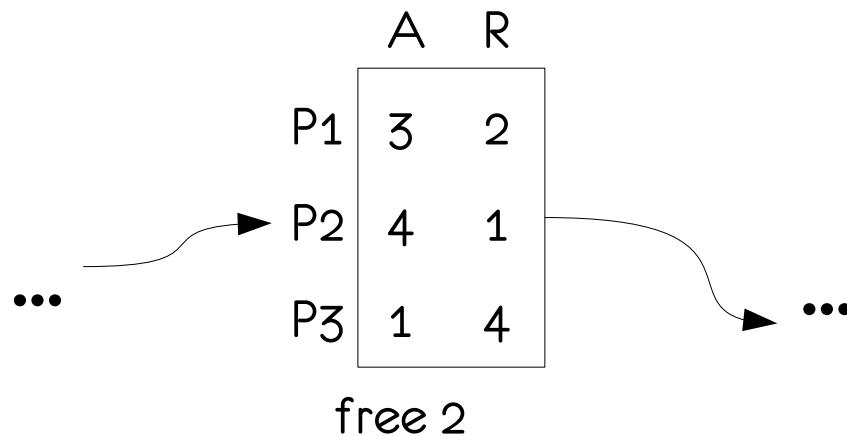


Altra sequenza di esecuzione



La scelta di soddisfare in parte la richiesta di P3 comporta invece il deadlock
infatti nessun processo ha abbastanza risorse per proseguire
e non ci sono più risorse libere

Stato iniziale?



Siamo partiti dallo stato indicato senza precisare che si può trattare di un qualsiasi stato di esecuzione, non necessariamente quello iniziale!!
Le risorse sono in parte già allocate quindi deve essere avvenuta una porzione di esecuzione

Deadlock avoidance

- Occorre introdurre due nozioni nuove:
 - <1> **stato (del sistema) sicuro**: si dice che il sistema è in uno stato sicuro (o safe) se il SO può garantire che ciascun processo completerà la propria esecuzione in un tempo finito
 - <2> **sequenza sicura**: una **sequenza** $\langle P^1, \dots, P^n \rangle$ di processi (parzialmente eseguiti) è detta **sequenza sicura** se le richieste che ogni P^i deve ancora fare sono soddisfacibili usando le **risorse attualmente libere** più le **risorse usate (e liberate) dai processi P^j** aventi $j < i$ (cioè dai processi che lo precedono)

Deadlock avoidance

- Finché il sistema di processi che condividono risorse rimane in uno stato sicuro il SO può evitare il verificarsi del deadlock

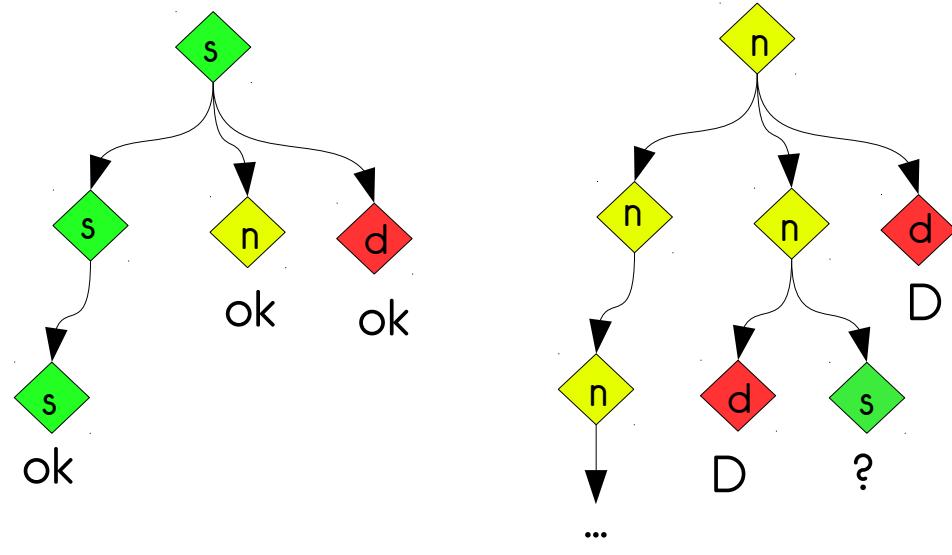
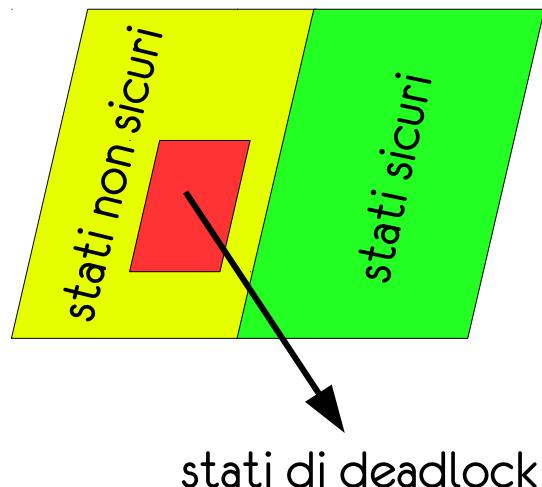
Perché funziona?

- Una sequenza sicura non presenta deadlock perché per ogni processo P^i sono veri i seguenti casi:
 - P^i ha tutte le risorse che gli servono
 - oppure a P^i basta aspettare la terminazione di qualche processo precedente per poter procedere
- può capitare che si generi una catena di attese che non ha mai termine?
 - l'unico caso è l'attesa circolare. L'attesa può risultare circolare?
 - no! Infatti alla peggio si torna indietro lungo la sequenza fino a P^1 :

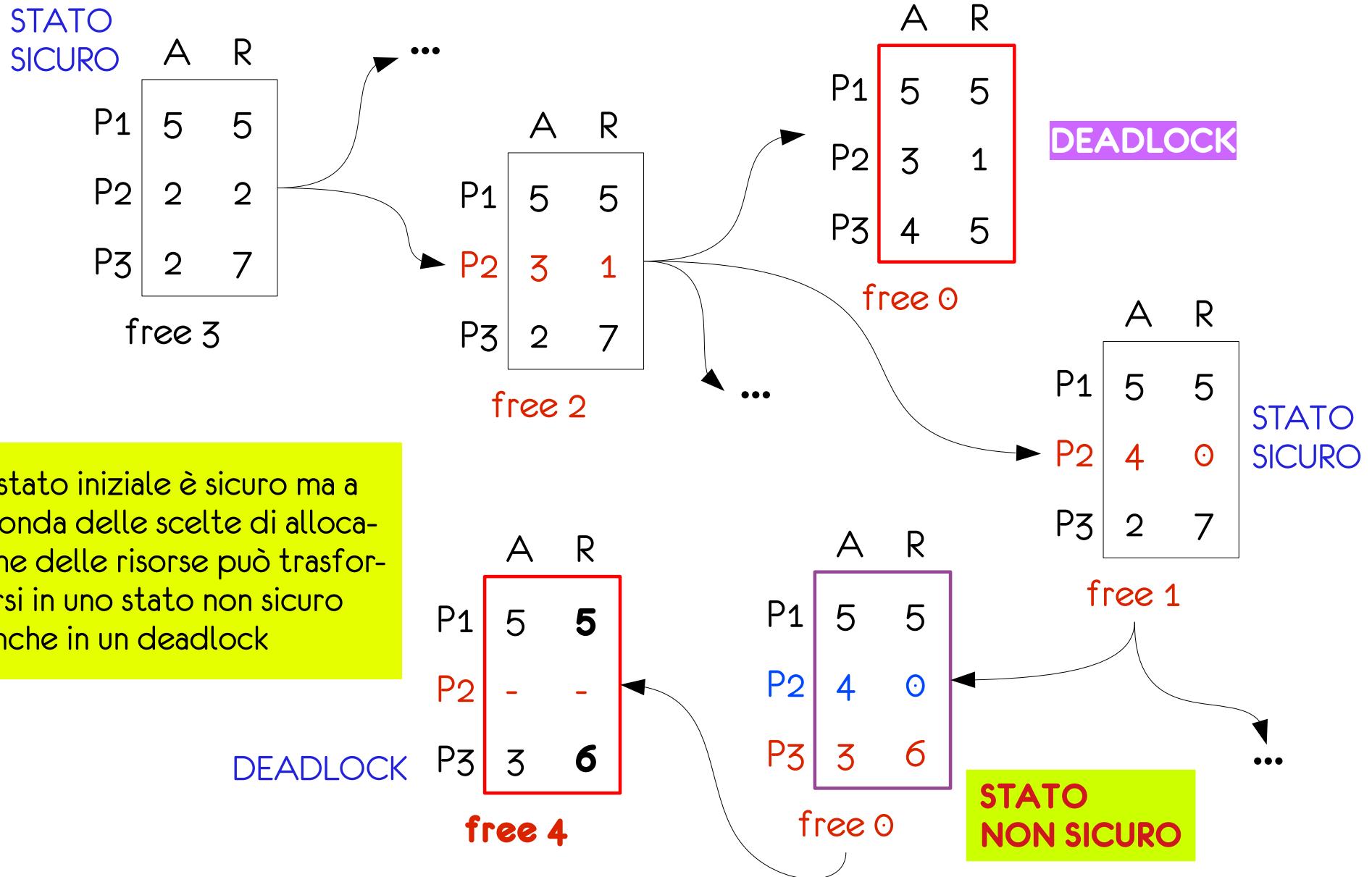
per definizione di sequenza sicura tutte le richieste che il processo P^1 effettuerà da qui alla sua fine devono essere o disponibili oppure in possesso di un processo P^j con $j < 1$, che però non esiste

Stati sicuri e sequenze sicure

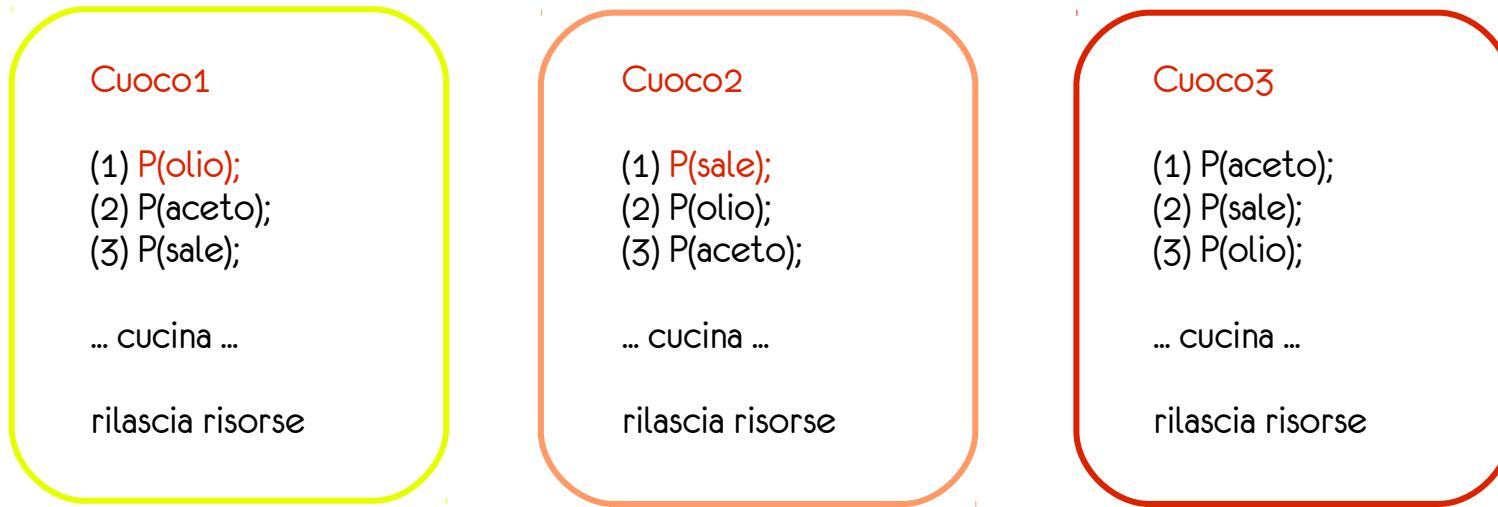
- Uno stato è sicuro se da esso si dirama almeno una **sequenza sicura**, quindi **se esiste** almeno un ordinamento dei processi che è una sequenza sicura
- Uno stato non sicuro non necessariamente è uno stato di deadlock ma può portare al deadlock
- Uno stato non sicuro può evolvere in uno stato sicuro? Se tale evoluzione esistesse per definizione lo stato sarebbe sicuro



Evoluzione degli stati

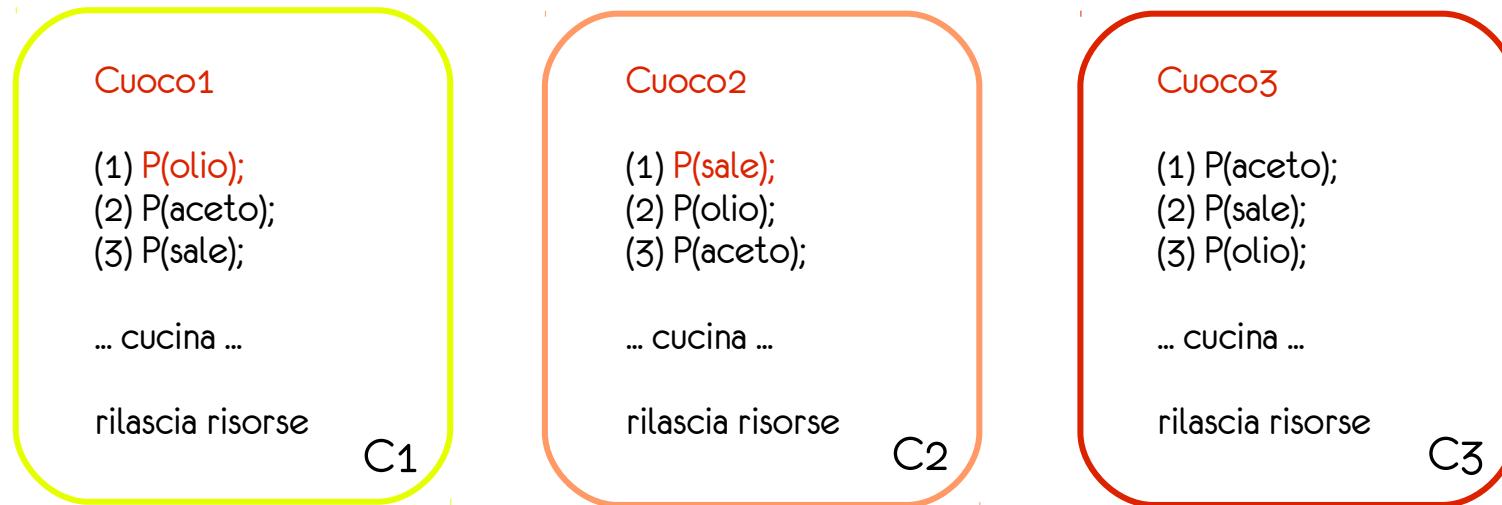


Altro esempio: i 3 cuochi



- Supponiamo che Cuoco1 abbia l'olio e Cuoco2 il sale, il sistema è in uno stato sicuro?
- In questo momento non c'è deadlock ma ...
- ... esiste un ordinamento che è una sequenza sicura?

i 3 cuochi



- Tutti i possibili ordinamenti:

- C1, C2, C3: no C1 dipende da C2 che lo segue
- C1, C3, C2: idem
- C2, C1, C3: no C2 dipende da C1 che lo segue
- C2, C3, C1: idem
- C3, C1, C2: no C1 dipende da C2
- C3, C2, C1: no C2 dipende da C1

Lo stato non
è sicuro

Deadlock avoidance

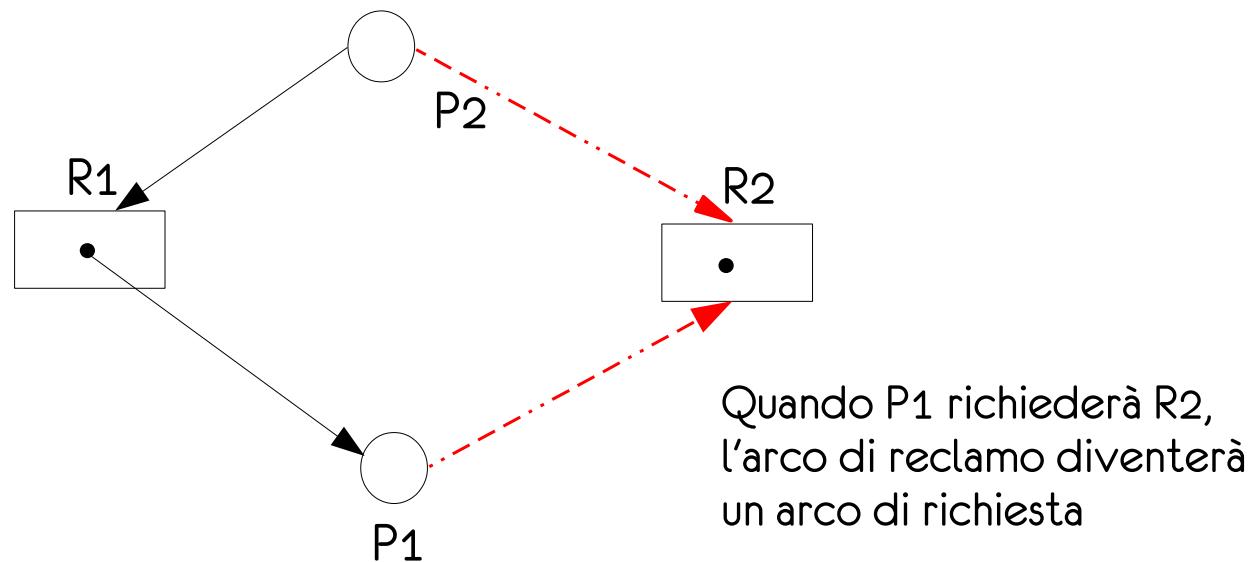
- Per evitare il deadlock il SO cerca di mantenere l'esecuzione in uno stato sicuro
- **E se una scelta sbagliata portasse a uno stato non sicuro?**
- in questo caso non è più possibile riportare il sistema in uno stato sicuro e molto facilmente si genererà un deadlock

Deadlock avoidance

- Per evitare il deadlock il SO cerca di mantenere l'esecuzione in uno stato sicuro
- **E se una scelta sbagliata portasse a uno stato non sicuro?**
- in questo caso non è più possibile riportare il sistema in uno stato sicuro e molto facilmente si genererà un deadlock

Algoritmo di deadlock avoidance

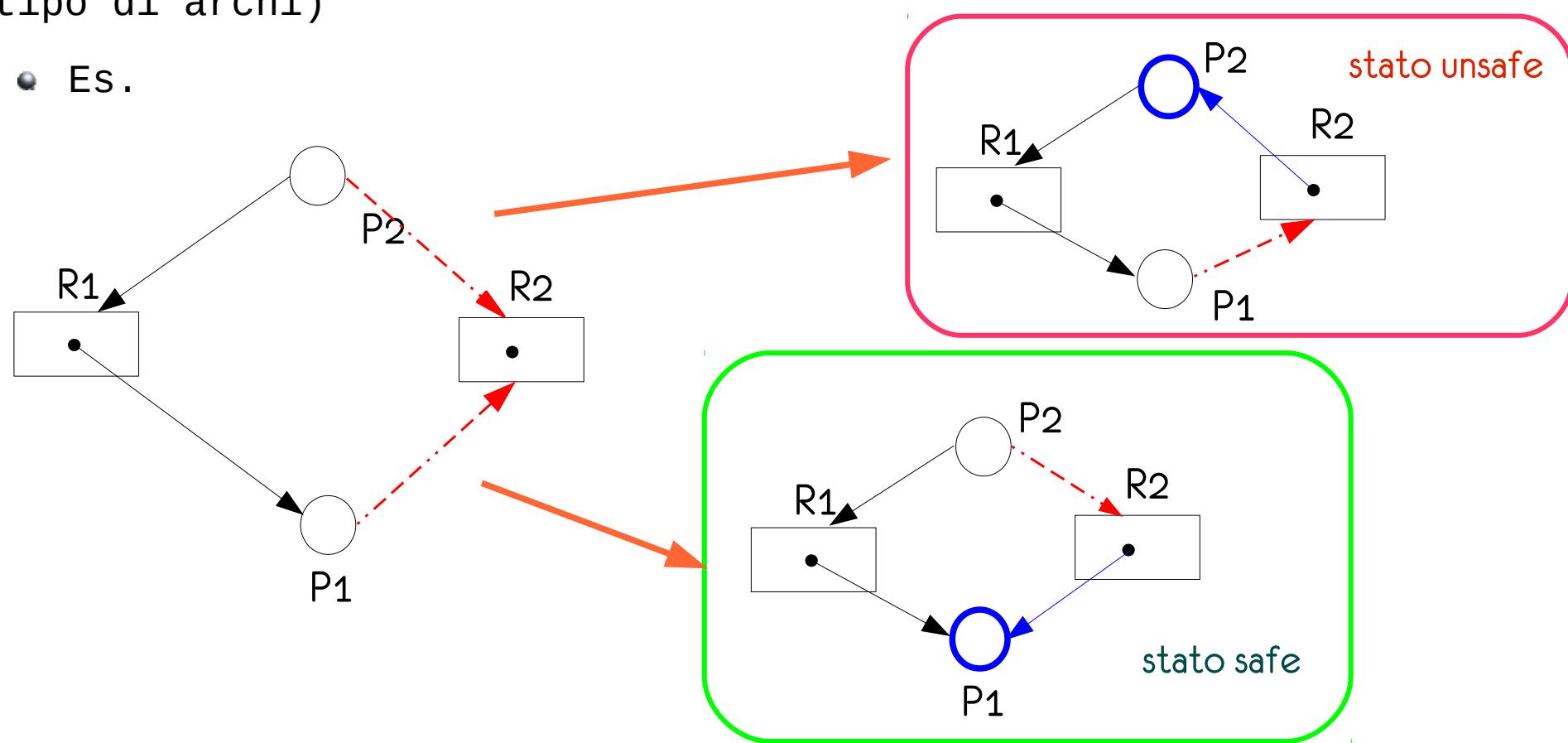
- Questo metodo funziona solo se ogni classe di risorsa ha **una istanza**
- È possibile prevenire il deadlock utilizzando una variante del grafo di assegnazione delle risorse ottenuto introducendo un terzo tipo di arco:
 - **arco di reclamo** (claim edge): $P_i \rightarrow R_j$ indica che P_i richiederà R_j *in futuro*; è rappresentato da una linea tratteggiata



Algoritmo di deadlock avoidance

- All'inizio tutti i processi inseriscono nel grafo di assegnazione un claim edge per ciascuna risorsa di cui avranno bisogno
- È possibile trasformare un arco di reclamo in un arco di richiesta SSE non si genera un ciclo (costituito da qualsiasi tipo di archi)

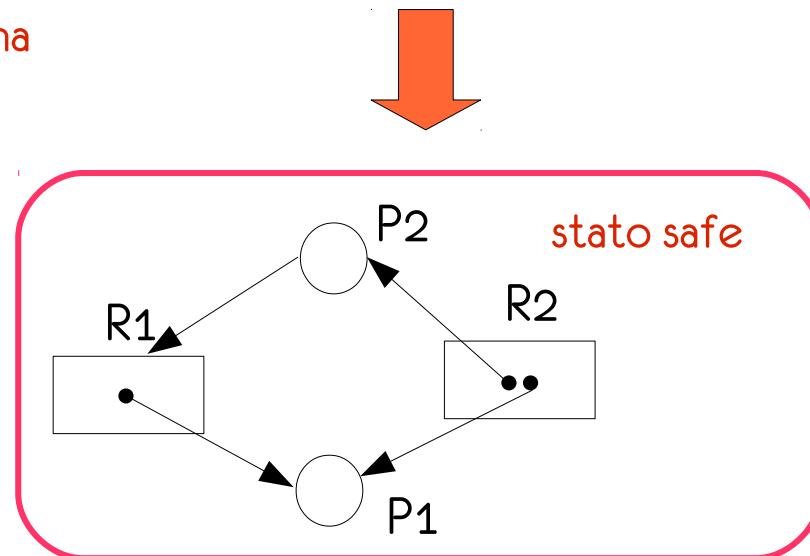
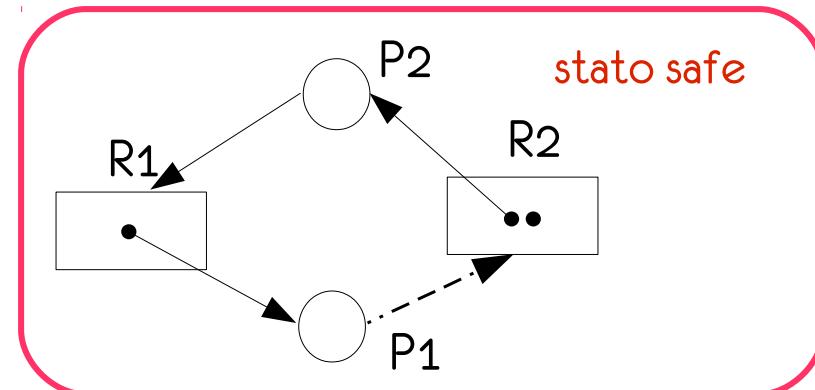
- Es.



Algoritmo di deadlock avoidance

NB: se io avessi due istanze di R2 lo stato sarebbe safe!!
La condizione non è più sufficiente

NOTA: quando un processo rilascia una risorsa l'arco di assegnazione ritorna ad essere un arco di reclamo



Avoidance: algoritmo del banchiere

- Algoritmo più generale, si applica anche quando i processi richiedono $n > 1$ risorse di un certo tipo
- **Metafora**: i processi sono visti come clienti di una banca a cui possono richiedere un prestito fino a un certo massimo
- **Informazione richiesta**: ogni nuovo processo deve dichiarare all'inizio il numero massimo di risorse (dei vari tipi) di cui avrà bisogno
- **M** = numero delle classi di risorsa gestite
- **N** = numero dei processi
- Complessità: $O(N^2M)$



Algoritmo del banchiere: variabili

- **disponibili[M]**: indica la disponibilità per ogni classe di risorsa
- **massimo[N][M]**: per ciascun processo indica il numero massimo di risorse di ciascun tipo che saranno richieste
- **assegnate[N][M]**: indica quante risorse di ciascuna classe sono assegnate a ogni processo
- **necessarie[N][M]**: indica quante risorse di ciascun tipo ancora mancano ai vari processi (necessarie = massimo - assegnate)

Algoritmo del banchiere

- L'algoritmo soddisfa una richiesta di un processo SSE
l'assegnazione delle risorse richieste porta ad uno stato sicuro
- È diviso in due algoritmi:
 - 1) un algoritmo per verificare che uno stato è sicuro
 - 2) un algoritmo di gestione di una richiesta (che utilizza il precedente)
- **Convenzione notazionale:**

Dati due vettori di uguale lunghezza X e Y si indica con $X < Y$ il fatto che per ogni indice i $X[i] < Y[i]$, si indica con $X \leq Y$ il fatto che per ogni indice i $X[i] \leq Y[i]$ e si indica con $Z = X + Y$ il fatto che per ogni indice i $Z[i] = X[i] + Y[i]$

Algoritmo di verifica della sicurezza

1. Siano **Lavoro** e **Fine** due array di lunghezza **M** ed **N**
2. **Lavoro = Disponibili**
3. **Fine[i] = falso**, per ogni $i \in [1, N]$
4. Cerca $i \in [1, N] \mid \text{Fine}[i] == \text{falso} \wedge \text{Necessarie}[i] \leq \text{Lavoro}$
5. se l'hai trovato:
 1. **Lavoro = Lavoro + Assegnate[i]**
 2. **Fine[i] = true**
 3. **goto 4**
6. altrimenti **goto 7**
7. se $\forall i \in [1, N], \text{Fine}[i] == \text{true}$ lo **stato è sicuro**

Esempio con M == 1

STATO
SICURO?

	A	R
P1	5	5
P2	2	2
P3	2	7

free 3

c'è $i \in [1, N]$ |
 fine[i] == false \wedge
 necessarie[i] \leq lavoro?
 Sì $i == 2!!$

↓
 lavoro = lavoro + assegnate[2]
 fine[2] = true

disponibili = 3
 necessarie = {5, 2, 7}
 assegnate = {5, 2, 2}

fine = {false, false, false}

lavoro = disponibili, cioè è uguale a 3

	A	R
P1	5	5
P2	-	-
P3	2	7

free 5

è come se fossi passata
virtualmente nello stato

disponibili = 3
 necessarie = {5, 2, 7}
 assegnate = {5, 2, 2}
 fine = {false, true, false}
 lavoro = 5

Esempio con M == 1

	A	R
P1	5	5
P2	-	-
P3	2	7

```

disponibili = 3
necessarie = {5, 2, 7}
assegnate = {5, 2, 2}

fine = {false, true, false}
lavoro = 5

```

free 5

c'è $i \in [1, N]$ |
 fine[i]==false \wedge
 necessarie[i] \leq lavoro?

Sì $i == 1!!$

\downarrow
 lavoro = lavoro+assegnate[1]
 fine[1] = true

è come se fossi passata
 virtualmente nello stato

	A	R
P1	-	-
P2	-	-
P3	2	7

\downarrow
 disponibili = 3
 necessarie = {5, 2, 7}
 assegnate = {5, 2, 2}
 fine = {true, true, false}
 lavoro = 10

free 10

Esempio con M == 1

	A	R
P1	-	-
P2	-	-
P3	2	7

free 10

c'è $i \in [1, N]$ |
 fine[i]==false \wedge
 necessarie[i] \leq lavoro?
 Sì $i == 3!!$

↓
 lavoro = lavoro+assegnate[3]
 fine[3] = true

disponibili = 3
 necessarie = {5, 2, 7}
 assegnate = {5, 2, 2}
 fine = {true, true, false}
 lavoro = 10

LO STATO VALUTATO
È SICURO

	A	R
P1	-	-
P2	-	-
P3	-	-

è come se fossi passata
virtualmente nello stato

free 12
 disponibili = 3
 necessarie = {5, 2, 7}
 assegnate = {5, 2, 2}
 fine = {true, true, true}
 lavoro = 12

Algoritmo di gestione delle richieste

1. Consideriamo un processo **j**, sia **Richieste** un vettore di **M** elementi, **Richieste[i]** è il num. di risorse di classe **i** richieste da **j** in un certo istante
2. Se **Richieste > Necessarie[j]** **ERRORE!** Il processo viola le sue stesse dichiarazioni iniziali di necessità
3. Se invece **Richieste > Disponibili** aspetta
4. Altrimenti **simula l'esecuzione** della richiesta:
 1. **Disponibili = Disponibili - Richieste**
 2. **Assegnate[j] = Assegnate[j] + Richieste**
 3. **Necessarie[j] = Necessarie[j] - Richieste**
5. **Verifica se lo stato raggiunto è sicuro:**
 1. **Se sicuro:** si effettua l'assegnazione
 2. **Se non è sicuro:** si ripristinano i valori precedenti e si sospende il processo

Esercizio

Applicare l'algoritmo del banchiere per decidere se assegnare o meno le risorse richieste dal processo P2 nella seguente situazione:

Nel sistema ci sono 4 classi di risorse con la seguente disponibilità massima $\{3, 2, 8, 5\}$
Abbiamo 5 processi che necessitano al più delle seguenti quantità di risorse:

P1	0	2	7	3
P2	2	0	5	1
P3	1	2	8	1
P4	3	2	8	3
P5	3	2	7	4

Ci troviamo in questo stato

P1	0	0	2	1	Risorse libere: $\{2, 0, 6, 4\}$
P2	0	0	0	0	
P3	1	2	0	0	
P4	0	0	0	0	
P5	0	0	0	0	

- (a) Verificare se lo stato è sicuro
- (b) Se sì, la richiesta di P2 di ottenere $\{2, 0, 5, 0\}$ risorse è soddisfacibile?
- (c) Se no, indicare uno stato che sarebbe sicuro e verificare se in esso la richiesta di P2 è ok

traccia

Il seguente stato è sicuro?

P1	0	0	2	1
P2	0	0	0	0
P3	1	2	0	0
P4	0	0	0	0
P5	0	0	0	0

lavoro = disponibile = {2, 0, 6, 4}

fine = {false, false, false, false, false}

necessarie[5][4] è la matrice delle risorse che servono ai processi ma non sono ancora state allocate:

P1	0	2	5	2
P2	2	0	5	1
P3	0	0	8	1
P4	3	2	8	3
P5	3	2	7	4

c'è un valore di i tale che fine[i]==false e necessarie[i]<lavoro? Sì: i = 2 infatti $\{2, 0, 5, 1\} \leq \{2, 0, 6, 4\}$

lavoro = {2, 0, 6, 4} non cambia perché P2 non aveva assegnate risorse

fine = {false, true, false, false, false}

c'è un valore di i tale che fine[i]==false e necessarie[i]<lavoro? CONTINUEATE DA SOLI ...

Che fare col deadlock?



- **Rilevare il deadlock:**

- è una capacità fondamentale se non abbiamo metodi, come i precedenti, che a priori ne evitano il generarsi: rilevato un deadlock è possibile attuare una politica di ripristino dalla condizione di stallo. Due casi:
 - istanza singola per ogni classe di risorsa
 - istanze multiple

- Rompere il deadlock quando si presenta:

- richiede la capacità di monitorare le richieste/assegnazioni di risorse

- Prevenire il deadlock:

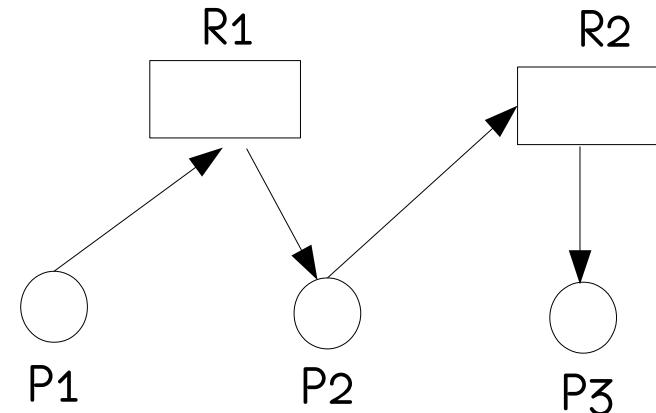
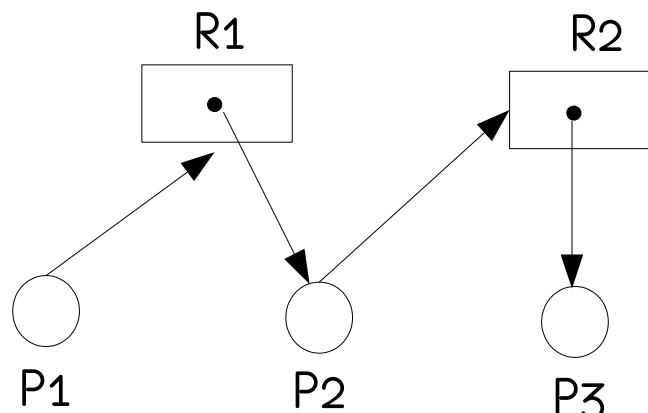
- occorre definire opportuni protocolli di assegnazione delle risorse

- Far finta che il deadlock sia impossibile:

- è la tecnica più usata, poco costosa perché non richiede né risorse aggiuntive né l'attuazione di politiche particolari

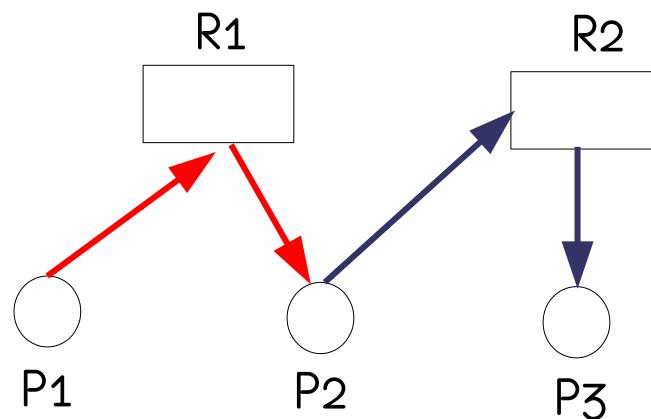
Istanza singola di risorsa

- Caso in parte già discusso
- Si basa su di una semplificazione del grafo di assegnazione delle risorse detto **Grafo d'Attesa**
- Un grafo d'attesa ha un solo tipo di vertici: i **processi**
- **Si può costruire un grafo di attesa partendo da un grafo di allocazione**
- <passo 1> Osserviamo che se l'istanza è singola, è ridondante mantenere una notazione per la classe di risorse e una per le istanze: una classe un'istanza

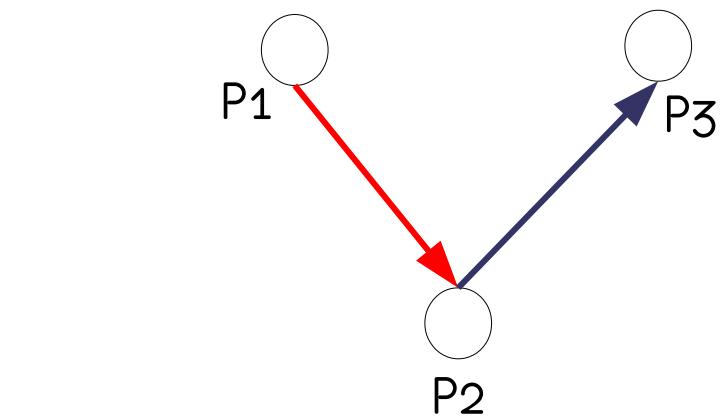


Istanza singola di risorsa

- Un grafo d'attesa è ancora più semplice infatti ha un solo tipo di vertici: i processi
- Un arco $P_i \rightarrow P_j$ indica che P_i è in attesa di una risorsa assegnata a P_j
- <passo 2> Un arco $P_i \rightarrow P_j$ corrisponde a una coppia di archi $P_i \rightarrow R_s$ e $R_s \rightarrow P_j$ nel grafo di allocazione:



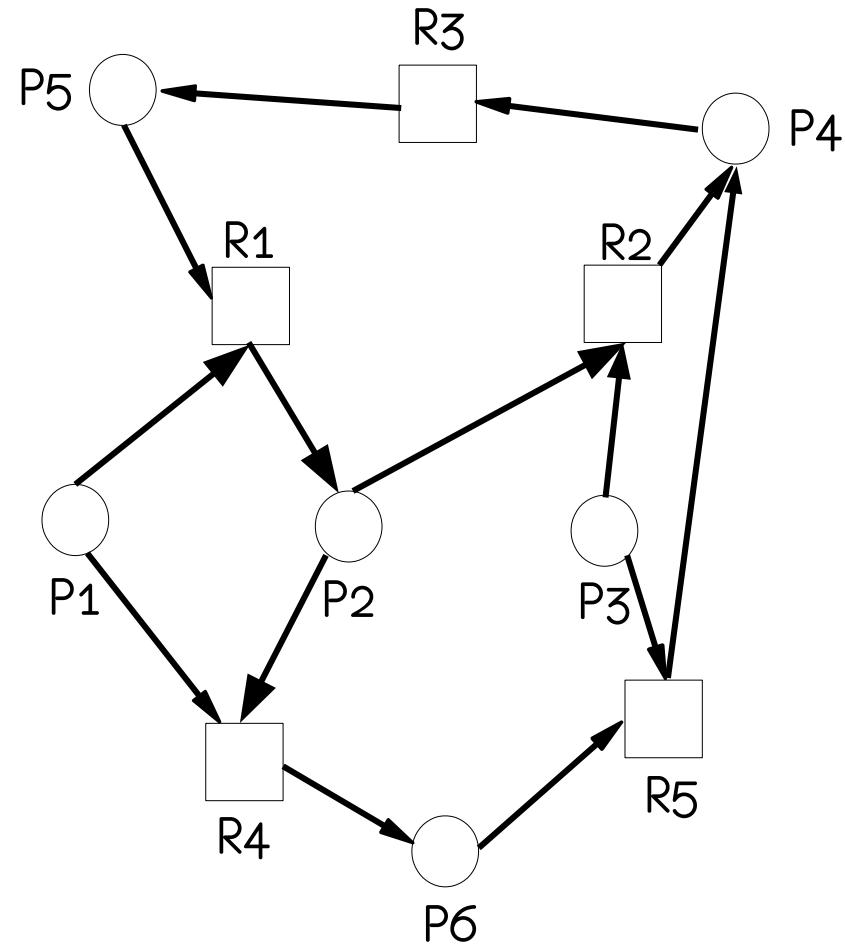
P1 aspetta la risorsa assegnata a P2
P2 aspetta la risorsa assegnata a P3



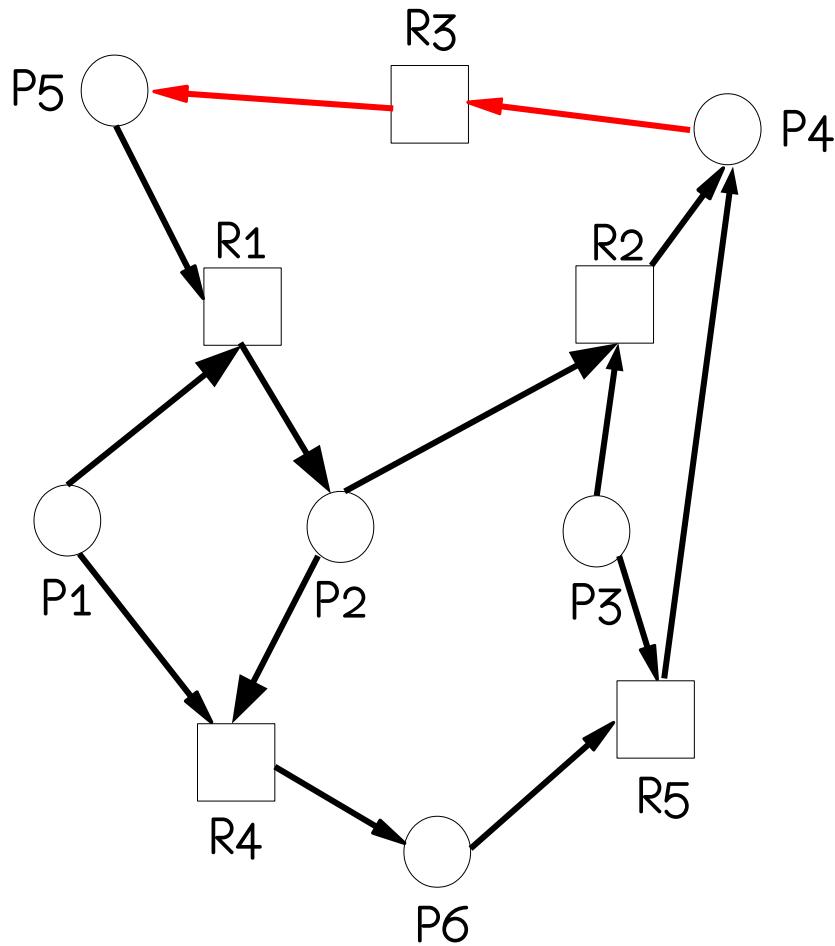
diventa

P1 aspetta P2
P2 aspetta P3

esempio 1/5



esempio 2/5



$P_1 \rightarrow R_1 \rightarrow P_2$
 $P_1 \rightarrow R_4 \rightarrow P_6$

$P_2 \rightarrow R_4 \rightarrow P_6$
 $P_2 \rightarrow R_2 \rightarrow P_4$

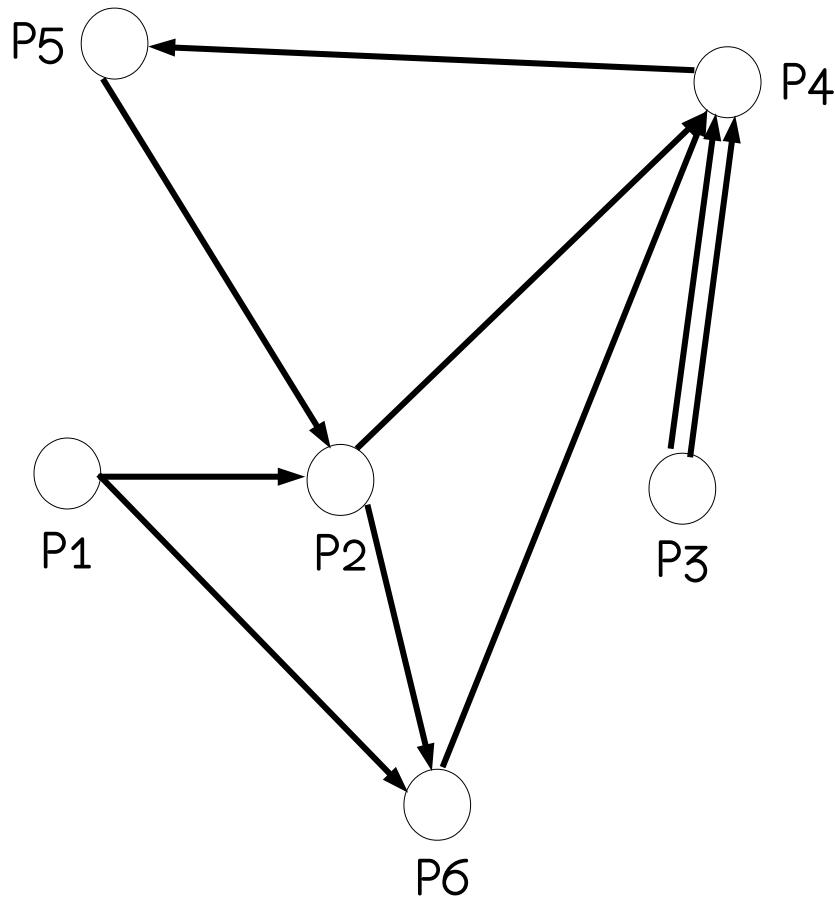
$P_3 \rightarrow R_2 \rightarrow P_4$
 $P_3 \rightarrow R_5 \rightarrow P_4$

$P_4 \rightarrow R_3 \rightarrow P_5$

$P_5 \rightarrow R_1 \rightarrow P_2$

$P_6 \rightarrow R_5 \rightarrow P_4$

esempio 3/5



$P_1 \rightarrow R_1 \rightarrow P_2$
 $P_1 \rightarrow R_4 \rightarrow P_6$

$P_2 \rightarrow R_4 \rightarrow P_6$
 $P_2 \rightarrow R_2 \rightarrow P_4$

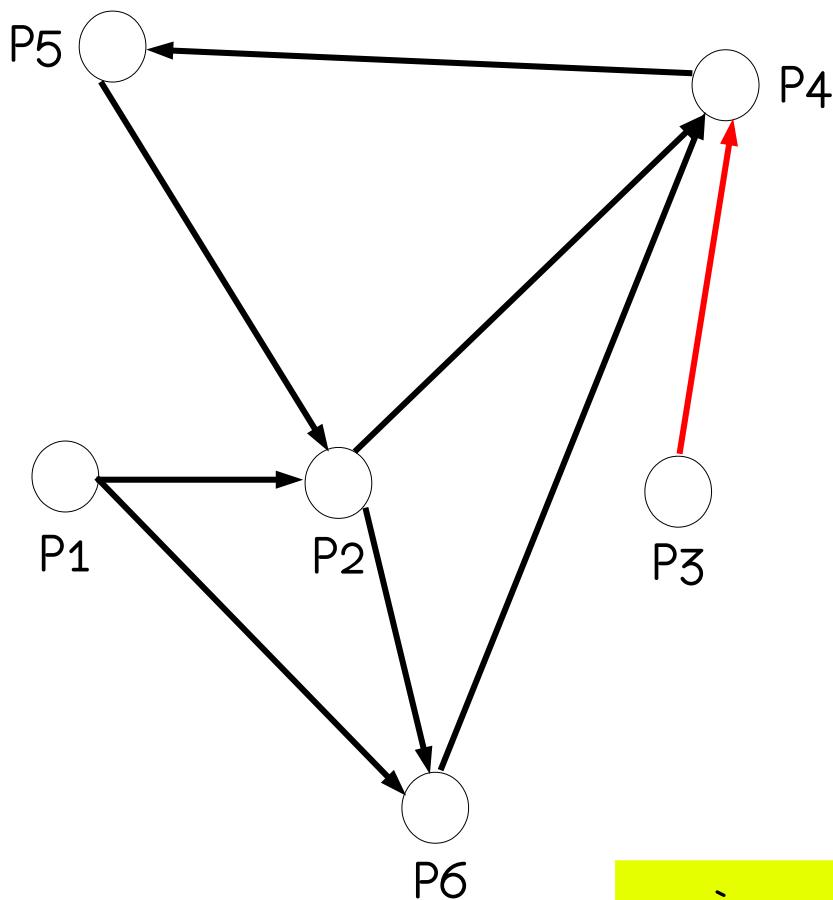
$P_3 \rightarrow R_2 \rightarrow P_4$
 $P_3 \rightarrow R_5 \rightarrow P_4$

$P_4 \rightarrow R_3 \rightarrow P_5$

$P_5 \rightarrow R_1 \rightarrow P_2$

$P_6 \rightarrow R_5 \rightarrow P_4$

esempio 4/5



$P_1 \rightarrow R_1 \rightarrow P_2$

$P_1 \rightarrow R_4 \rightarrow P_6$

$P_2 \rightarrow R_4 \rightarrow P_6$

$P_2 \rightarrow R_2 \rightarrow P_4$

$P_3 \rightarrow R_2 \rightarrow P_4$

$P_3 \rightarrow R_5 \rightarrow P_4$

$P_4 \rightarrow R_3 \rightarrow P_5$

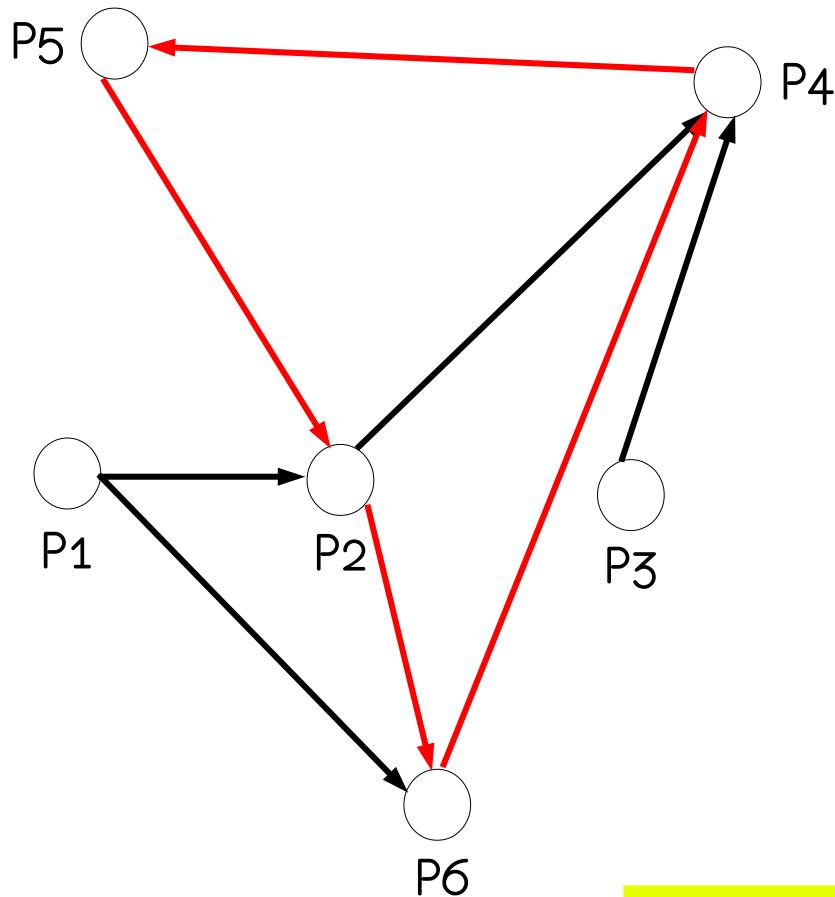
$P_5 \rightarrow R_1 \rightarrow P_2$

$P_6 \rightarrow R_5 \rightarrow P_4$

nella visualizzazione li
fondiamo in un arco solo

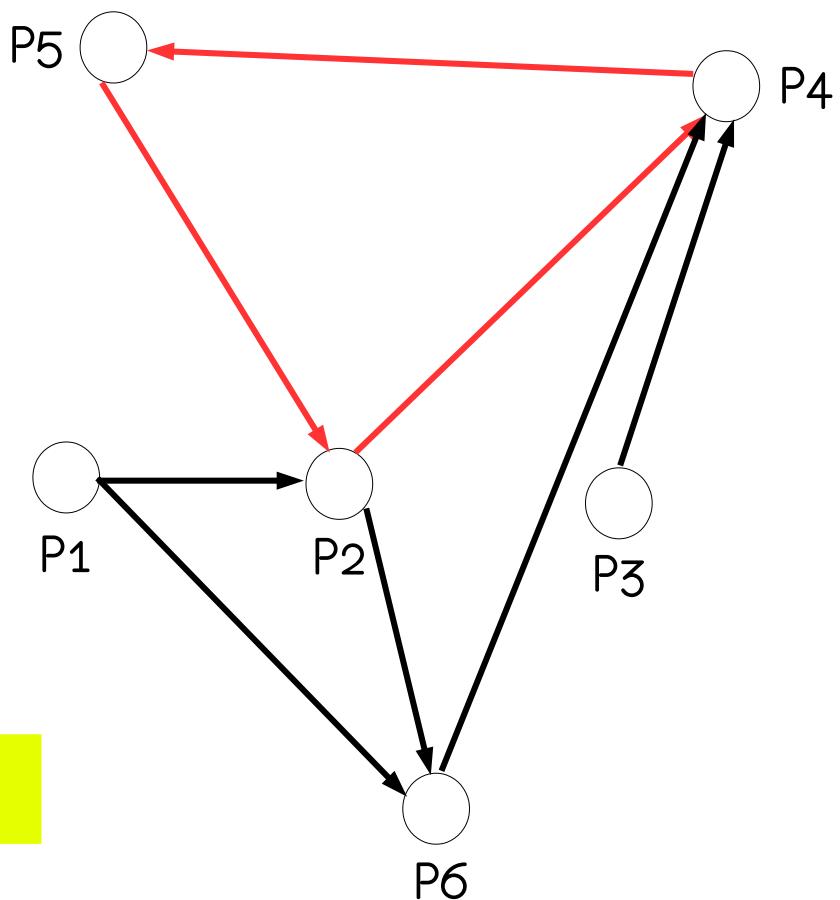
C'È DEADLOCK?
BISOGNA VERIFICARE SE CI SONO CICLI

esempio 4/5



DEADLOCK

CI SONO CICLI?
PIÙ DI UNO ...



Istanze multiple di risorsa

- Il metodo precedente è applicabile se e solo se per ogni classe di risorsa esiste un'unica istanza
- In generale per ogni classe di risorsa R^i si possono avere molte istanze, in formule: $| R^i | = N^i \geq 1$

Istanze multiple di risorsa

- Occorre utilizzare *strutture dati analoghe a quelle usate nell'algoritmo del banchiere*:
- **Disponibili[M] = {n₁, ..., n_k}** mantiene il numero di istanze disponibili di ogni risorsa
- **Assegnate[N][M]**: ogni riga della matrice indica quante istanze di ciascun tipo di risorsa sono state assegnate a un certo processo; **Assegnate[i]** indica l'attuale assegnazione per processo P_i
- **Richieste[N][M]**: ogni riga della matrice indica la **richiesta attuale** di ogni processo, tale richiesta può comprendere istanze di risorse differenti (non si tratta di claim, cioè di richieste future)
- NB: Assegnate e Richieste catturano le situazioni di possesso e attesa correnti

NB: lo scopo è rilevare il deadlock, accorgersi se c'è e non prevenirlo

Esempio

Disponibili == {3, 1, 0, 2}

Assegnate[3][4] ==

0	0	0	0
0	1	1	0
2	0	1	3

Abbiamo 4 classi di risorse di cui sono attualmente disponibili rispettivamente 3, 1, 0 e 2 istanze

Richieste[3][4] ==

1	2	0	0
0	0	0	0
0	1	0	0

Ci sono 3 processi due dei quali hanno assegnate istanze di diversi tipi di risorse (P2 ha complessivamente 2 istanze di due risorse diverse, P3 ha 6 istanze di 3 risorse diverse)

Dei tre processi due hanno una richiesta in corso (P1 richiede 3 istanze di due classi diverse e P3 richiede una sola istanza di una risorsa). Secondo la disponibilità attuale la richiesta di P3 è soddisfacibile quella di P1 no

L'algoritmo che vedremo **individua cicli di processi**, per essere in un ciclo un processo deve avere assegnate alcune risorse ed essere in attesa di altre (avere richieste in corso)

Algoritmo

```
1. int Lavoro[M];
2. boolean Fine[N];
3.
4. /* inizializzazione */
5. Lavoro = Disponibili;
6. for (i in [1,N])
7.     if (Assegnate[1] == {0, 0, ..., 0}) Fine[i] = true;
8.     else Fine[i] = false;
9.
10. /* calcolo */
11. while  $\exists$  un indice i | Fine[i] == false  $\wedge$  Richieste[i]  $\leq$  Lavoro
    1. Lavoro = Lavoro + Assegnate[i]
    2. Fine[i] = true
12.
13. /* test: c'è deadlock? */
14. for (i in [1,N])
    1. if (Fine[i] == false) << c'è deadlock >>
```

F sta per False

NB: tutti i processi per cui Fine[i] = false sono in deadlock

esempio

Lavoro == Disponibili == {3, 1, 0, 2}

Fine[N] = {true, false, false}

Assegnate[3][4] ==

0	0	0	0
0	1	1	0
2	0	1	3

P1 non può essere
parte di un ciclo
non ha risorse
assegnate

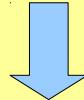
Richieste[3][4] ==

1	2	0	0
0	0	0	0
0	1	0	0

NON C'È DEADLOCK
FINE = {TRUE, TRUE, TRUE}

while ($\exists i \mid \text{Fine}[i] == \text{false} \wedge \text{Richieste}[i] \leq \text{Lavoro}$)

1. ...



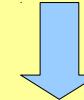
condizione vera per $i == 2$, infatti:

$\text{Fine}[2] == \text{false}$

$\text{Richieste}[2] == \{0, 0, 0, 0\} < \{3, 1, 0, 2\}$

$\text{Lavoro} = \text{Lavoro} + \text{Assegnate}[2] = \{3, 2, 1, 2\}$

$\text{Fine}[2] = \text{true}$



while ($\exists i \mid \text{Fine}[i] == \text{false} \wedge \text{Richieste}[i] \leq \text{Lavoro}$)

1. ...



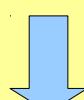
condizione vera per $i == 3$, infatti:

$\text{Fine}[3] == \text{false}$

$\text{Richieste}[3] == \{0, 1, 0, 0\} < \{3, 2, 1, 2\}$

$\text{Lavoro} = \text{Lavoro} + \text{Assegnate}[3] = \{5, 2, 2, 5\}$

$\text{Fine}[3] = \text{true}$



while ($\exists i \mid \text{Fine}[i] == \text{false} \wedge \text{Richieste}[i] \leq \text{Lavoro}$)

LA CONDIZIONE È FALSA

esempio con deadlock

Lavoro == Disponibili == {3, 1, 0, 2}

Fine[N] = {true, false, false}

Assegnate[3][4] ==

0	0	0	0
0	1	1	2
2	0	1	3

P1 non può essere
parte di un ciclo
non ha risorse
assegnate

Richieste[3][4] ==

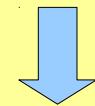
1	0	0	0
2	0	1	0
0	1	0	3

Se volete provare un caso più generale
simulate sulla matrice Assegnate:

1	0	0	0
0	1	1	2
2	0	1	3

while ($\exists i \mid \text{Fine}[i] == \text{false} \wedge \text{Richieste}[i] \leq \text{Lavoro}$)

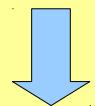
1. ...



condizione falsa, infatti:

Richieste[2] == {0, 1, 1, 2} non è < {3, 1, 0, 2}
infatti una componente di Richieste è > della
corrispondente componente di Lavoro

Richieste[3] == {2, 0, 1, 0} non è < {3, 1, 0, 2}



ESCO DAL WHILE

C'È DEADLOCK?
FINE = {TRUE, FALSE, FALSE}

Sì, i processi in deadlock sono
P2 e P3

Uso degli algoritmi visti

- Quando usare gli algoritmi di rilevamento del deadlock?
- Dipende:
 - dalla frequenza con cui si verificano i deadlock
 - dal numero di processi mediamente coinvolti
- Si possono definire alcune euristiche:
 - effettuare la verifica quando un processo che richiede una risorsa la trova occupata (può capitare di frequente)
 - effettuare la verifica quando l'utilizzo della CPU scende al di sotto di una certa soglia o a intervalli fissi
- In generale, si può dire che esiste un processo responsabile del deadlock?
 - no, tutti i processi coinvolti in un ciclo sono corresponsabili

Che fare col deadlock?

- “Rompere” il deadlock - quando viene identificata una situazione di deadlock:
 - 1) terminare i processi coinvolti:
tutti, uno, qualcuno?
 - 2) effettuare la prelazione delle risorse
 - 3) riassegnare le risorse





Soluzione 1: terminazione

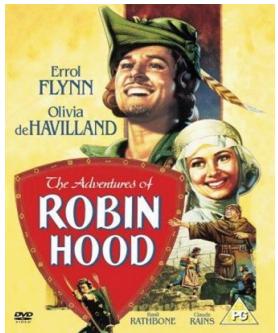
- Terminare un processo è costoso perché il lavoro da esso svolto svolto viene perduto
- Si possono adottare diverse politiche:
 - **Terminare tutti i processi coinvolti**
molto oneroso!!!
 - **Terminare un processo per volta fino alla risoluzione del deadlock**
occorre applicare l'algoritmo di rilevamento dopo l'abort di ciascun processo
- Abort di un processo: può comportare l'insorgere di problemi di consistenza in presenza di transazioni atomiche. In questo caso il SO deve effettuare il rollback delle transazioni interrotte



Come scegliere la vittima?



- **Desiderio**: scegliere il/i processo/i la cui terminazione è meno onerosa
- **Problema**: non esiste una misura precisa a cui fare riferimento
- Alcune misure di riferimento sono:
 - priorità dei processi (scelgo processi a bassa priorità)
 - tempo di computazione effettuata rapportato al tempo stimato di computazione residua
 - processo interattivo / processo batch
 - ...



Soluzione 2: prelazione

- Idea: sottrarre successivamente risorse ad alcuni processi per assegnarle ad altri
- Anche in questo caso occorre identificare una vittima e la scelta è effettuata su criteri economici, la prelazione di risorse deve essere poco costosa
- Che fare dei processi a cui sono state sottratte risorse? Come per la terminazione potrebbe essere necessario riportare lo stato del sistema a una condizione di consistenza
- Occorre evitare che vengano sottratte risorse sempre allo stesso processo impedendogli così di continuare (insorgenza di starvation!!)

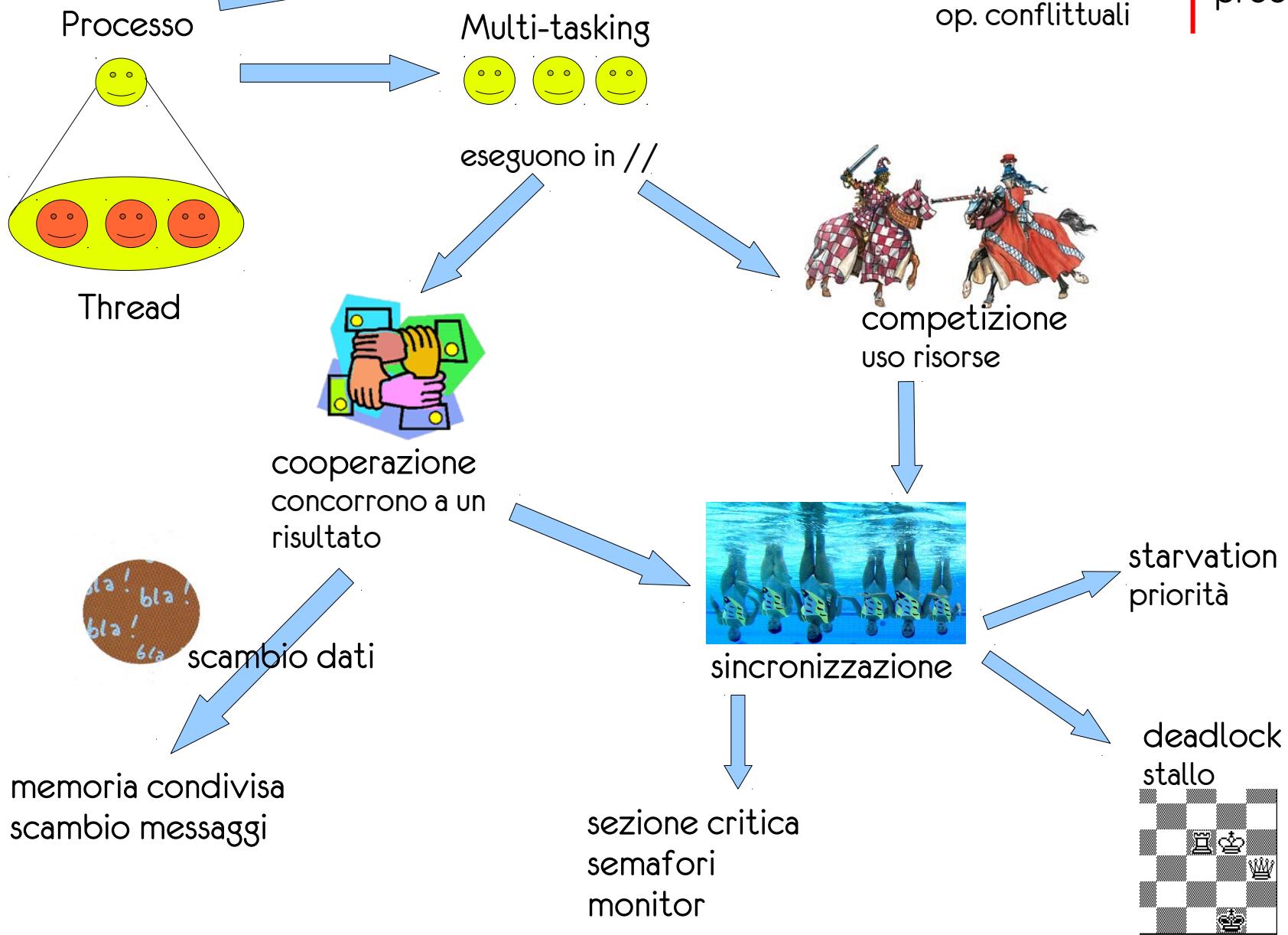
Che fare col deadlock?

- **Far finta di niente:**

- è la tecnica più usata, poco costosa perché non richiede né risorse aggiuntive né l'attuazione di politiche particolari
- quando un utente si accorge che si è verificato un deadlock, lo risolve manualmente



mappa dei principali concetti visti

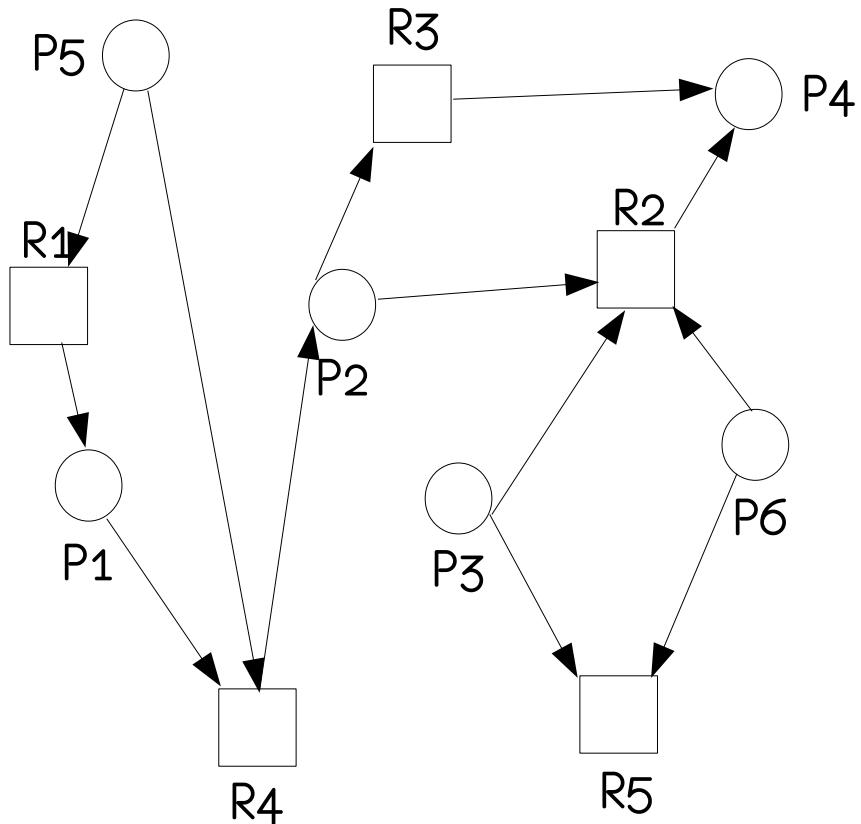


Esercizi

- Grafo allocazione risorse - grafo di attesa - presenza di deadlock
- Evoluzioni dello stato di allocazione delle risorse e deadlock
- Grafi di allocazione con archi di reclamo: generazione di stati sicuri e non sicuri
- Presenza di deadlock in caso di risorse con istanze multiple
- Deadlock avoidance: Algoritmo del banchiere
- Transazioni equivalenti

esercizio

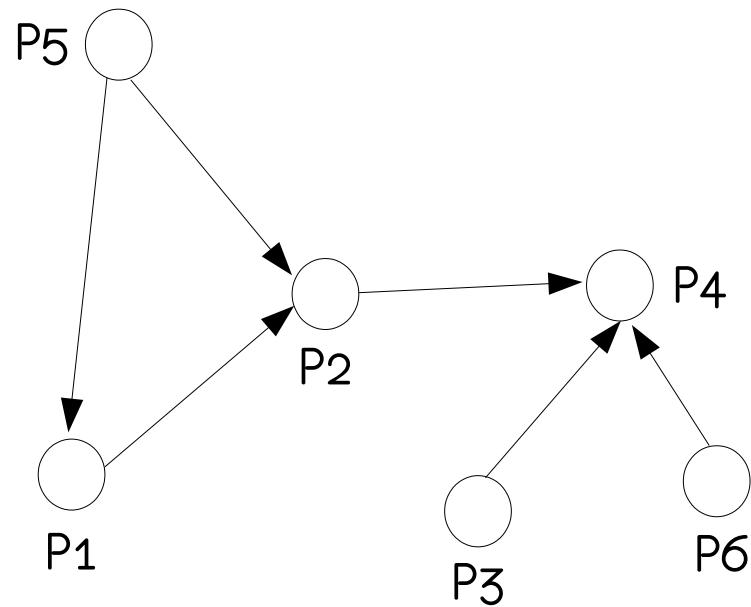
Si consideri il seguente grafo di assegnazione delle risorse, trasformarlo in un grafo di attesa e verificare se vi è deadlock o meno motivando la risposta



soluzione

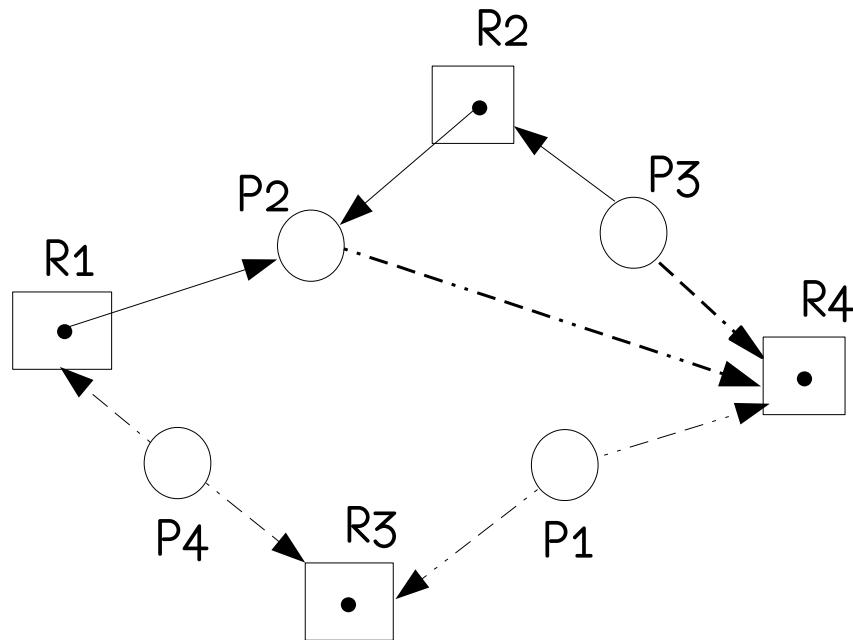
Soluzione:

non c'è deadlock perché il grafo di attesa non contiene cicli

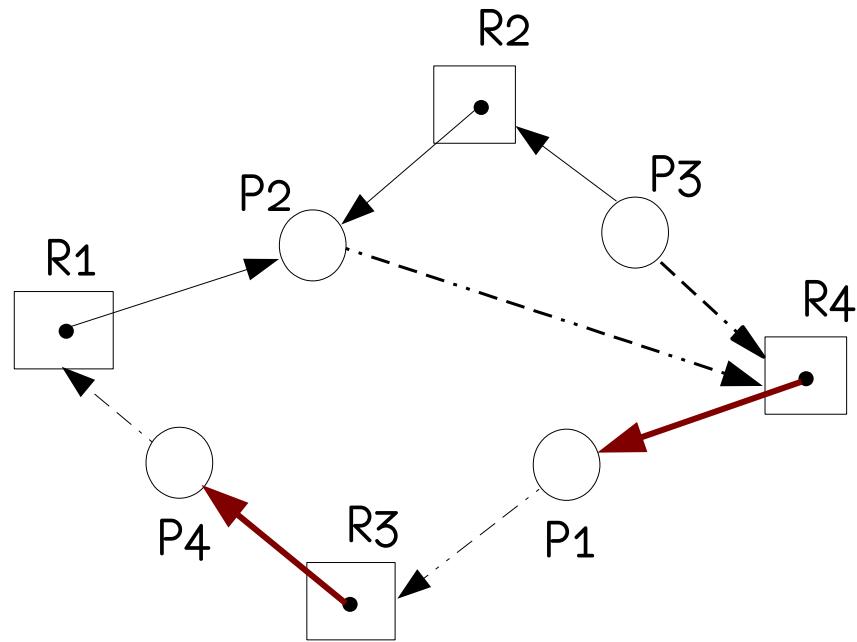


esercizio

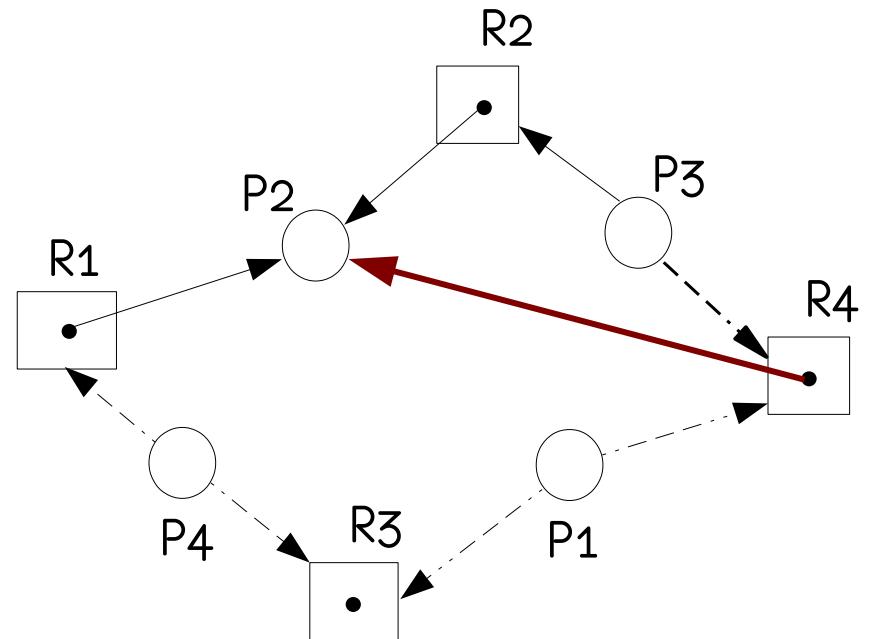
Dato il seguente grafo di assegnazione delle risorse con archi di reclamo individuare un'assegnazione che porta a uno stato non sicuro e un'assegnazione sicura



soluzione



Stato non sicuro: il grafo contiene
un ciclo che coinvolge P_2, P_1, P_4



Stato sicuro: il grafo non contiene alcun ciclo

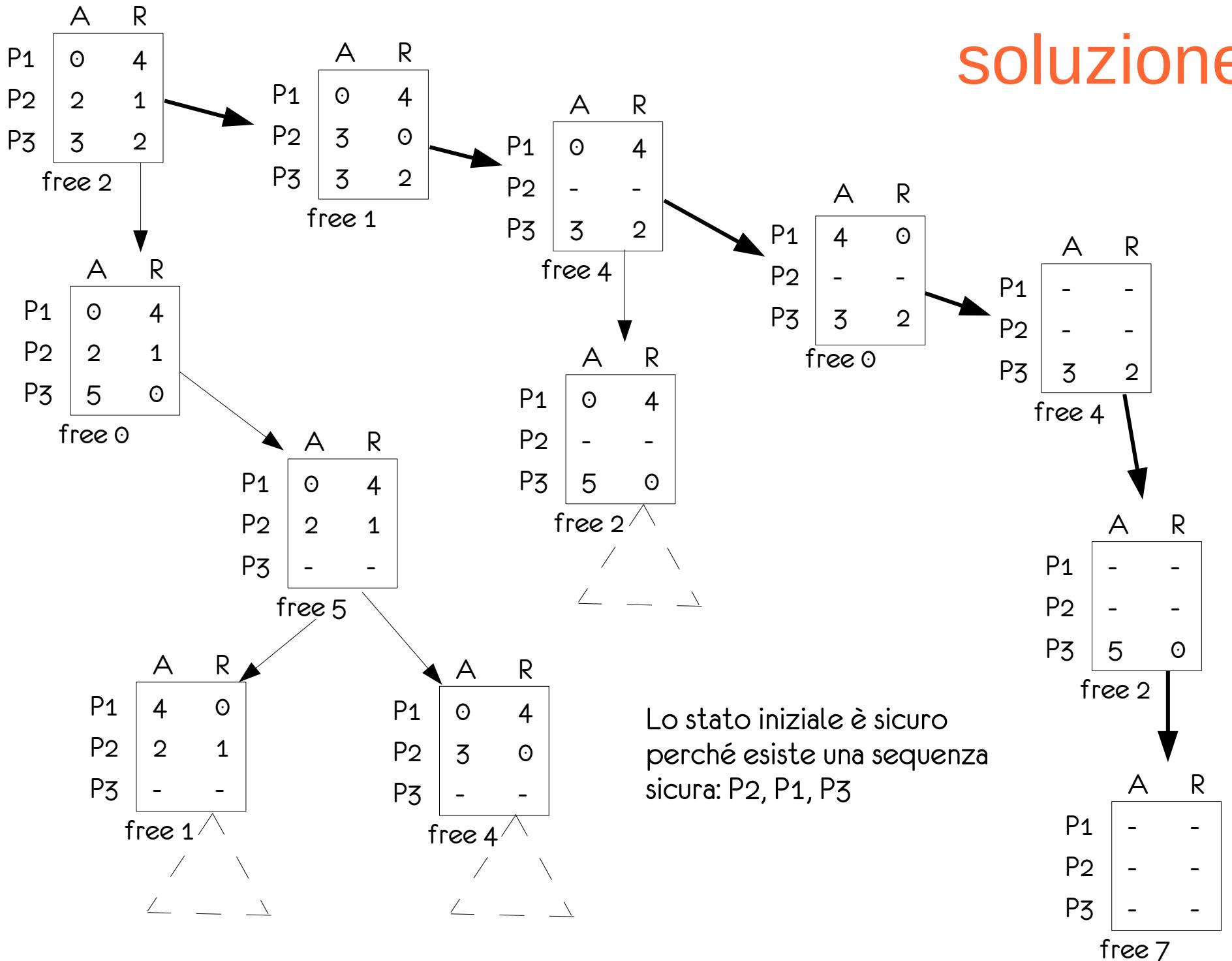
esercizio

Costruire tutte le possibili evoluzioni dello stato di allocazione delle risorse riportato di seguito. La colonna A indica le risorse in possesso dei processi, R indica il numero di risorse necessarie ma non ancora assegnate, free indica il numero di risorse attualmente libere. Al termine dire se lo stato iniziale è sicuro motivando la risposta.

	A	R
P1	0	4
P2	2	1
P3	3	2

free 2

soluzione



esercizio

Dati 3 processi e 3 classi di risorse con istanze multiple applicare l'algoritmo di identificazione del deadlock allo stato descritto nel seguito; in caso di deadlock si specifichi quali processi sono coinvolti

Disponibili == {0, 0, 1}

Assegnate[3][3] ==

1	0	0
0	1	1
2	0	1

Richieste[3][3] ==

1	2	0
0	0	0
0	1	0

soluzione

Disponibili == {0, 0, 1}

Assegnate[3][3] ==

1	0	0
0	1	1
2	0	1

Richieste[3][3] ==

1	1	3
0	0	0
0	1	0

Lavoro = Disponibili
Fine = {false, false, false}

c'è un i | !Fine[i] && richieste[i]<Lavoro? Si, richieste[2]

Lavoro = Lavoro + Assegnate[2] = {0,1,2}

Fine = {false, true, false}

c'è un i | !Fine[i] && richieste[i]<Lavoro? Si, richieste[3]

Lavoro = Lavoro + Assegnate[3] = {2,1,3}

Fine = {false, true, true}

c'è un i | !Fine[i] && richieste[i]<Lavoro? Si, richieste[1]

Lavoro = Lavoro + Assegnate[1] = {3,1,3}

Fine = {true, true, true}

Tutti i processi hanno Fine a true quindi non c'è deadlock

esercizio

Dati 3 processi e 3 classi di risorse con istanze multiple applicare l'algoritmo di identificazione del deadlock allo stato descritto nel seguito; in caso di deadlock si specifichi quali processi sono coinvolti

Disponibili == {0, 0, 1}

Assegnate[3][3] ==

1	0	0
0	1	1
2	0	1

Richieste[3][3] ==

1	2	0
0	1	0
0	1	0

soluzione

Disponibili == {0, 0, 1}

Assegnate[3][3] ==

1	0	0
0	1	1
2	0	1

Richieste[3][3] ==

1	1	3
0	1	0
0	1	0

Lavoro = Disponibili

Fine = {false, false, false}

c'è un i | !Fine[i] && richieste[i]<Lavoro? Si, richieste[2]

No, infatti:

Richieste[1] = {1,1,3} non è < {0,0,1}

Richieste[2] = {0,1,0} non è < {0,0,1}

Richieste[3] = {0,1,0} non è < {0,0,1}

C'è deadlock, i processi coinvolti sono P1, P2 e P3
(hanno tutti Fine a false)

esercizio

Applicare l'algoritmo del banchiere per decidere se lo stato riportato qui di seguito è sicuro o meno. Motivare la risposta.

	A	R
P1	5	1
P2	2	2
P3	1	8
free 1		

soluzione

	A	R
P1	5	1
P2	2	2
P3	1	8
free	1	

```
lavoro=disponibili = 1
```

```
necessarie = {1, 2, 8}
```

```
assegnate = {5, 2, 1}
```

```
fine = {false, false, false}
```

```
c'è i | !Fine[i] && necessarie[i]<=lavoro? Si, i = 1
```

```
lavoro = lavoro + assegnate[i] = 6
```

```
fine = {true, false, false}
```

```
c'è i | !Fine[i] && necessarie[i]<=lavoro? Si, i = 2
```

```
lavoro = lavoro + assegnate[i] = 8
```

```
fine = {true, true, false}
```

```
c'è i | !Fine[i] && necessarie[i]<=lavoro? Si, i = 3
```

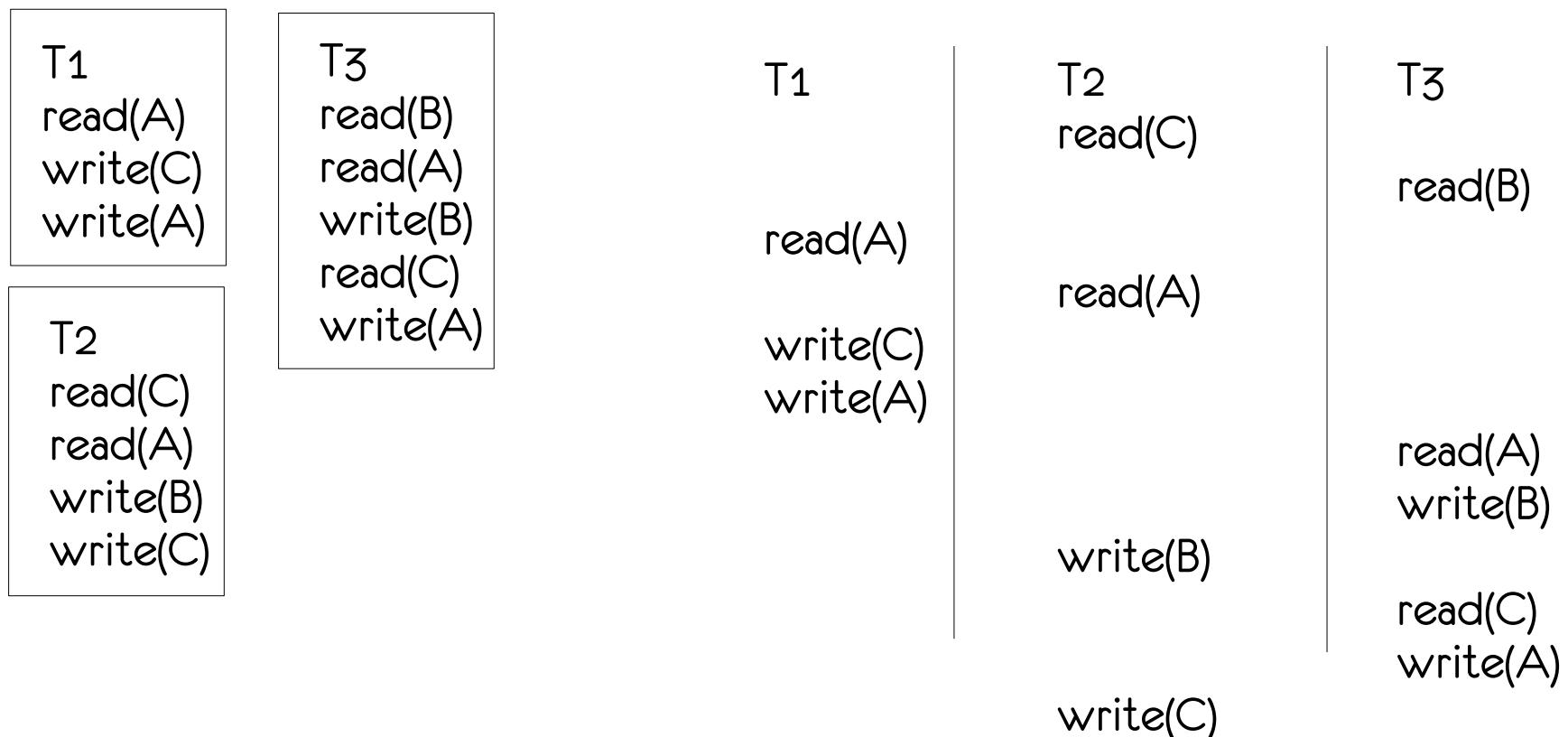
```
lavoro = lavoro + assegnate[i] = 9
```

```
fine = {true, true, true}
```

lo stato è sicuro perché tutti i processi hanno fine a true

esercizio

Date le transazioni T1, T2 e T3 riportate nel seguito dire, motivando la risposta, se l'esecuzione riportata è equivalente all'esecuzione sequenziale di T2, T1, T3 (nell'ordine indicato)



soluzione 1/2

T2	T1	T3	T2	T1	T3
read(C)			read(C)		read(B)
read(A)			read(A)	read(A)	
write(B)				write(C)	
write(C)				write(A)	
	read(A)				read(A)
	write(C)				write(B)
	write(A)				read(C)
		read(B)			write(A)
		read(A)			
		write(B)			
		read(C)			
		write(A)			

Bisogna provare a trasformare l'esecuzione delle 3 transazioni in sequenza (sulla sinistra) in un'esecuzione in cui le operazioni sono eseguite nell'ordine dato a destra. Per farlo occorre vedere, coppia x coppia di operazioni di transazioni diverse, contigue nel tempo e da spostare, se sono conflittuali: se no, si procede, se sì le due sequenze non sono equivalenti.

soluzione 2/2

T2	T1	T3	T2	T1	T3
read(C)			read(C)		read(B)
read(A)			read(A)	read(A)	
write(B)				write(C)	
write(C)				write(A)	
	read(A)				read(A)
	write(C)				write(B)
	write(A)				read(C)
		read(B)			write(A)
		read(A)			
		write(B)			
		read(C)			
		write(A)			

Per es. read(C) di T2 non cambia posizione, procediamo.

read(B) di T3 diventa la seconda operazione eseguita: per avere l'equivalenza, tale operazione non deve essere in conflitto con nessuna di quelle in blu. In effetti non è in conflitto con le 3 operazioni di T1 e neppure con write(C) di T2, però è in conflitto con write(B) di T2, quindi le due sequenze non sono equivalenti.

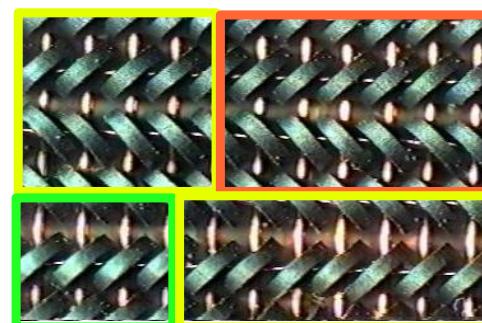
memoria centrale

capitolo 8 del libro (VII ed.)

Introduzione

- La memoria principale, come la CPU, è una **risorsa condivisa** fra i processi
- È necessario imporre dei meccanismi di gestione
- Alcuni metodi vengono messi a disposizione già a livello HW
- Altri metodi sono realizzati a un livello di astrazione superiore, fra questi:
 - paginazione della memoria
 - segmentazione della memoria
- La scelta del metodo da adottare dipende da diversi fattori, *in primis* l'architettura considerata

ripartiamo dal livello 0
(HW) e pian piano
costruiamo le sovra-
strutture necessarie



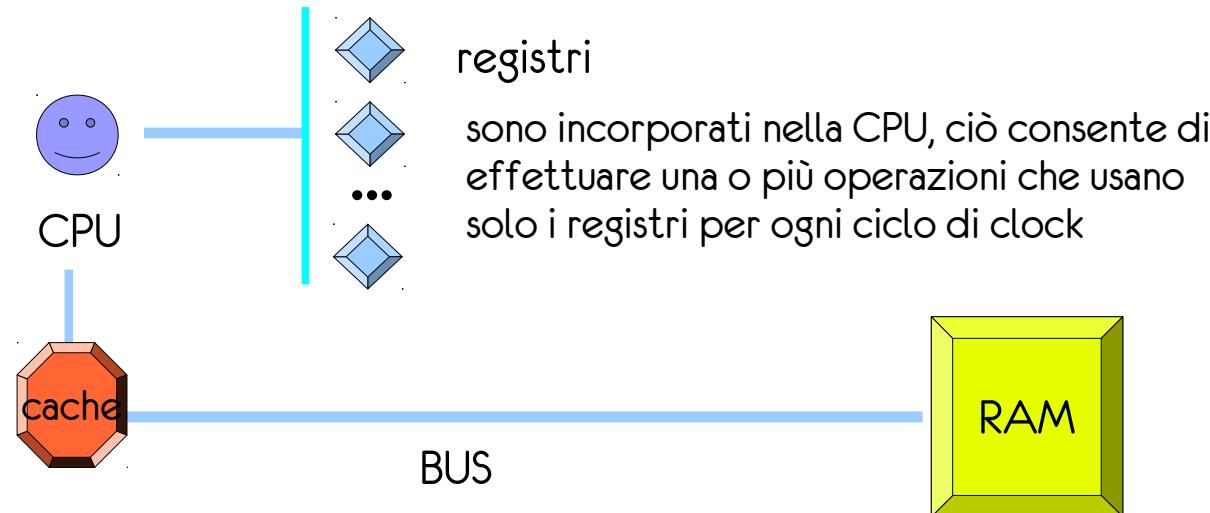
SOVRASTRUTTURA
suddivisione fra i
processi

Core Memory - RAM - 1951

Livello Hardware

- RAM: unica memoria di grandi dimensioni direttamente accessibile alla CPU
- la CPU preleva le istruzioni da eseguire (operazione fetch) e i dati da elaborare (load) dalla RAM e memorizza in essa i risultati dell'elaborazione (store). Ciclo:
 - **fetch**: individua tramite program counter la prossima istruzione e caricala nell'instruction register
 - **decode**: decodifica l'operazione tramite un codice operativo
 - **execute**: individua e carica i dati richiesti ed esegui l'operazione
- **NB**: la CPU riceve ed utilizza un **flusso di indirizzi di memoria**, non sa come questi siano generati né a cosa servano
- Concentriamoci quindi per ora sui **singoli indirizzi**

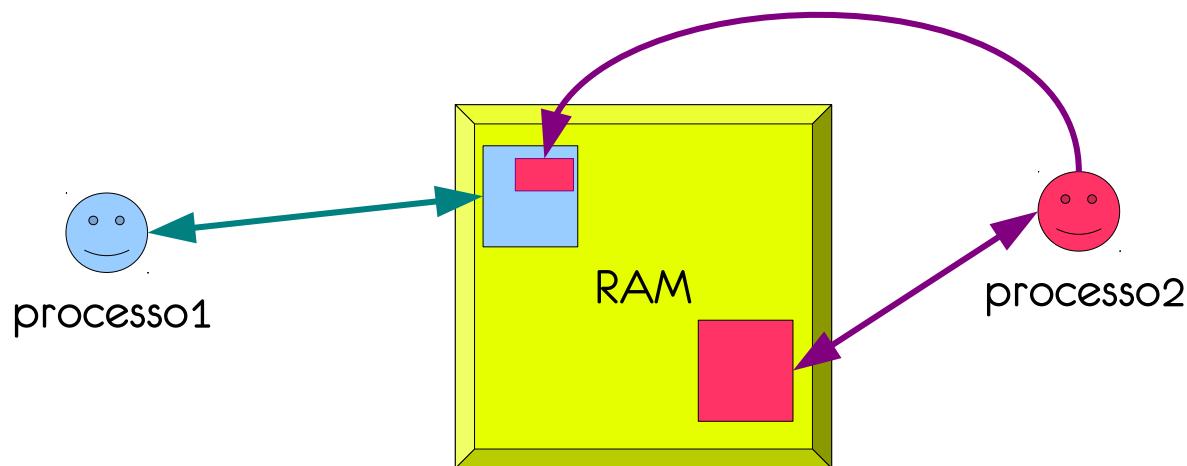
Schema generale: livello 0



La RAM è accessibile via bus
talvolta l'accesso alla RAM richiede diversi cicli di clock durante i quali la CPU è in attesa

Per consentire un uso migliore della CPU spesso si frappone un buffer (cache) fra RAM e CPU. Scopo della cache: mediare la lentezza di interazione con la RAM mettendo a disposizione dati "probabilmente utili"

Memoria e processi

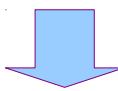


Processo 1 e Processo 2 usano ciascuno una porzione di RAM ...

... e se per errore Processo 2 scrivesse un proprio dato sopra a un'istruzione di Processo 1?

Occorre attuare meccanismi di protezione

La RAM è condivisa da vari processi, per ogni processo la RAM contiene il suo codice (completo o parziale) e i dati da elaborare. È necessario far sì che un processo non vada a sovrascrivere il/i codice/dati di un altro (specie se quest'ultimo è un processo del SO!!!)



In altri termini occorre definire meccanismi che consentono di **assegnare a ogni processo un'area di memoria separata**

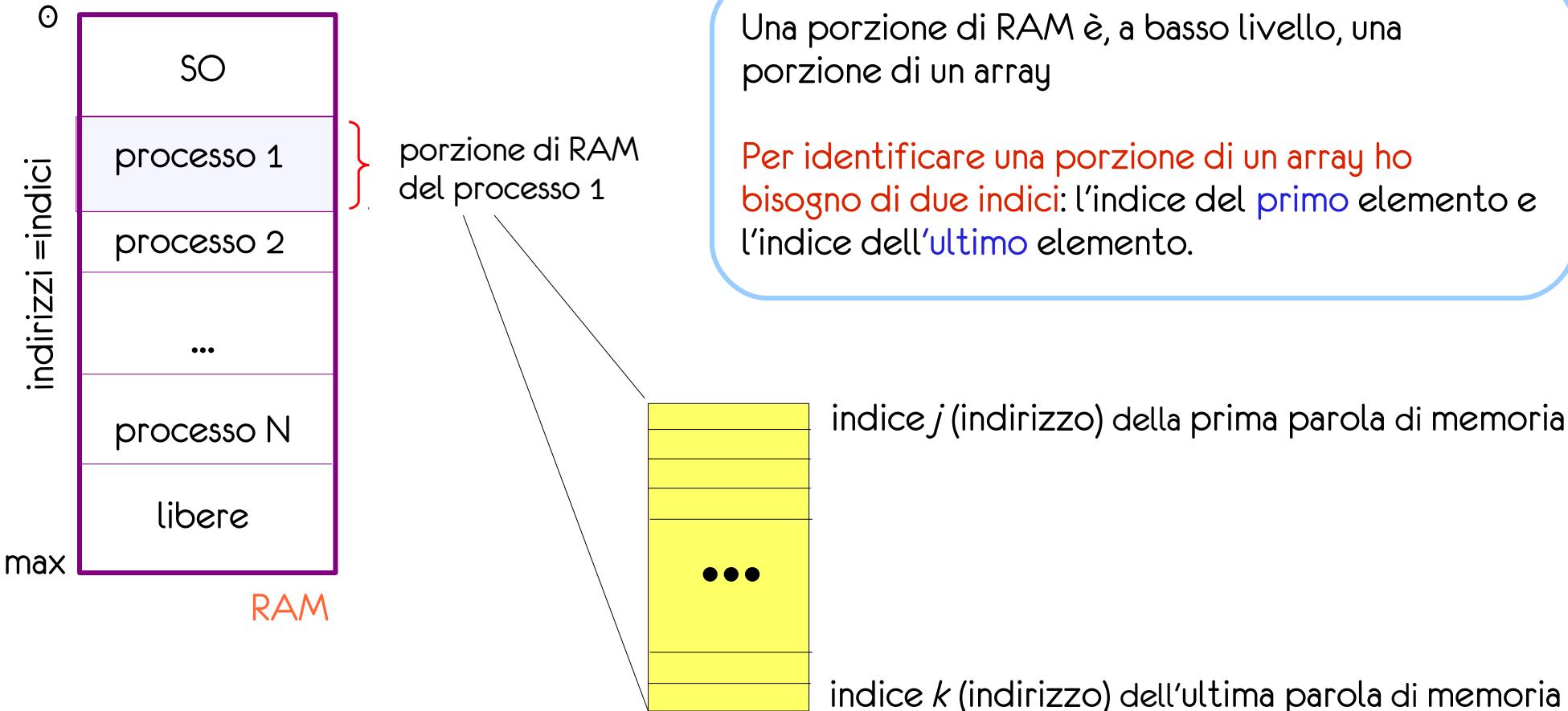
Processi e spazi degli indirizzi

- Un processo in esecuzione è caricato in RAM (codice, dati)
- L'esecuzione di un processo comporta la produzione di indirizzi (di istruzioni, di dati)
- La CPU produce un flusso di indirizzi che consentono al processo l'accesso a elementi contenuti in RAM
- Occorre controllare che un processo acceda solo agli indirizzi appartenenti alla porzione di RAM ad esso assegnata. Esempio di codice che produce errori di accesso in RAM:

```
for (i=0 ; ; i++) mio_array[i] = 0;
```

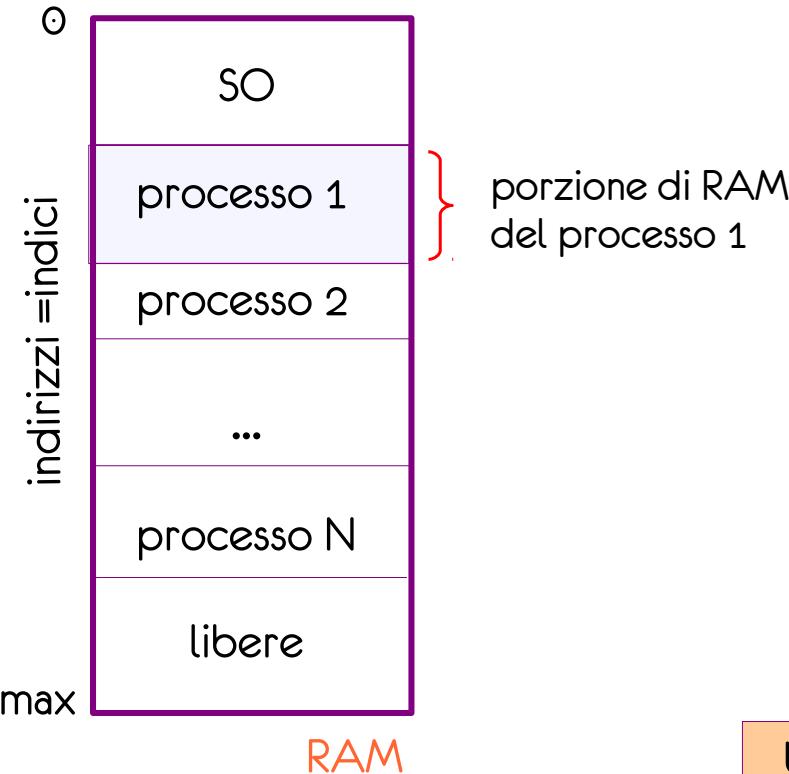
- Vi sono diversi modi di realizzare il mapping processore-RAM, per cominciare consideriamo il più semplice

Registro base e limite 1/3



Posso conservare tale coppia di valori per ciascun processo

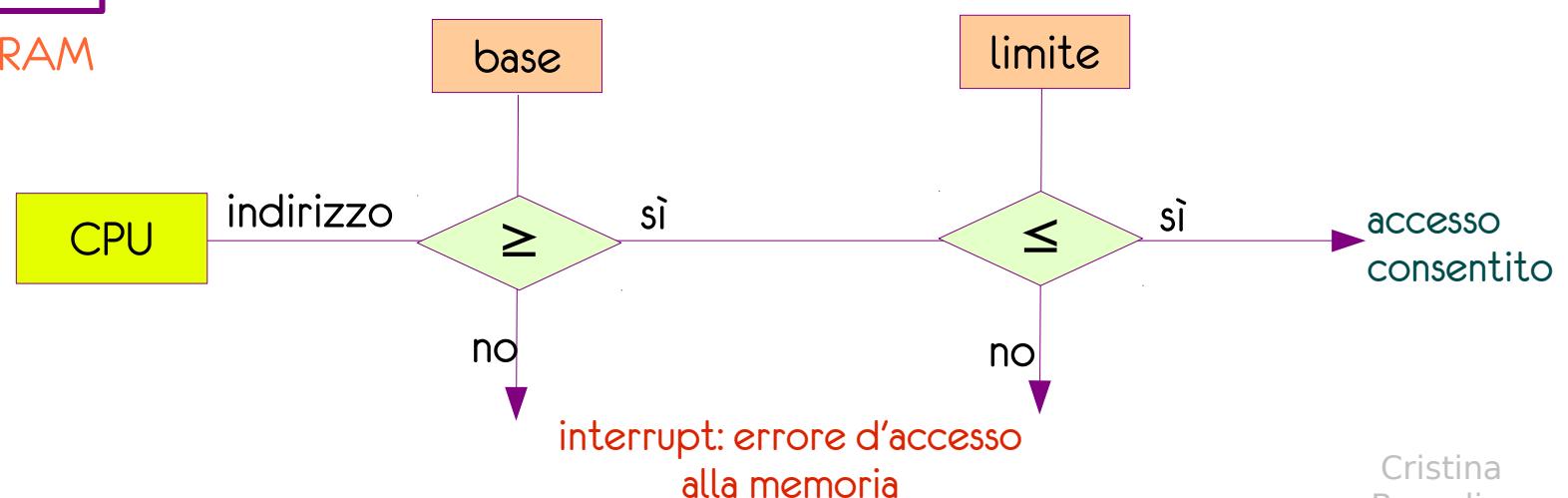
Registro base e limite 2/3



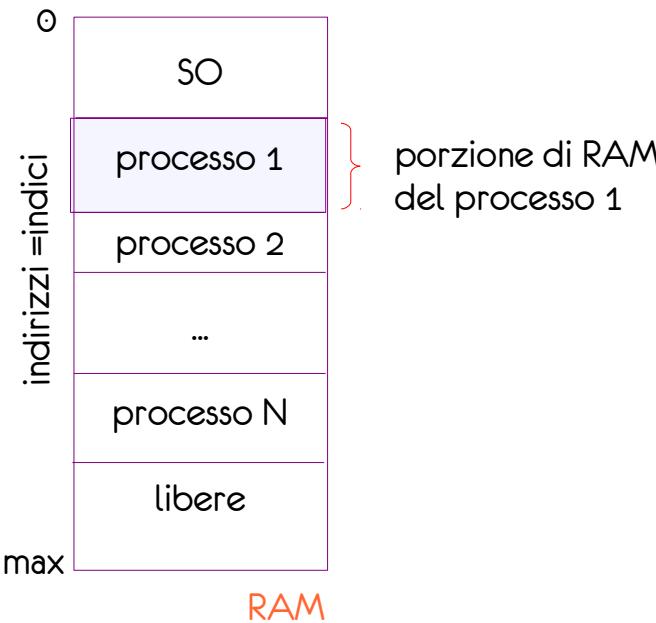
Il SO (e solo il SO) può eseguire un'istruzione che carica tali valori in due registri (registro base e registro limite)

Meccanismo di controllo:

Prodotto un indirizzo
la CPU lo confronta
con il registro base e
il registro limite



Registro base e limite 3/3

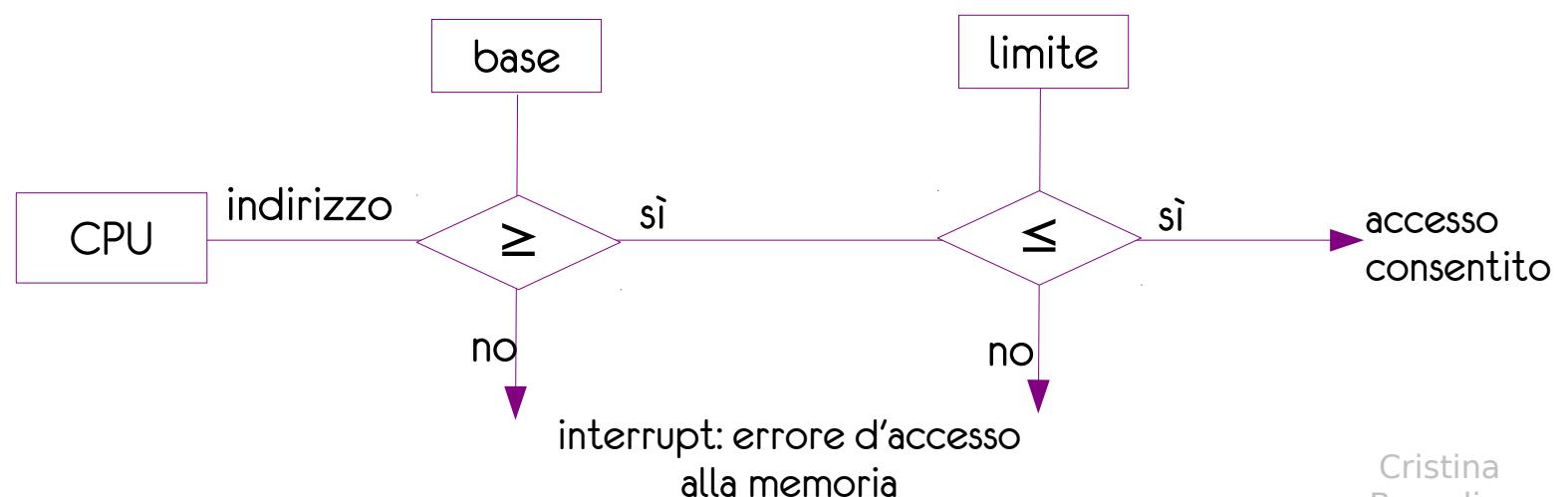


Il SO (e solo il SO) può eseguire un'istruzione che carica tali valori in due registri (registro base e registro limite)

A quale porzione di RAM può accedere il SO?
Tutta, perché per esempio si occupa di caricare in RAM i processi pronti da eseguire

Meccanismo di controllo:

Prodotto un indirizzo
la CPU lo confronta
con il registro base e
il registro limite



Binding fra variabili e indirizzi

```
...  
void inizializzaRandom(graf *G) {  
    int numN, i, j;  
    printf("\n Quanti nodi vuoi creare?  
    scanf("%d", &numN);  
  
    (*G).nodes = (nodo *) malloc(sizeof(nodo) * numN);  
    (*G).numN = numN;  
    (*G).edges = (connessioni)malloc(sizeof(connessioni) * numN);  
  
    for (i=0; i<numN; i++) {  
        (*G).nodes[i].dato = i; /  
        /*C) nodosi il vicinato = 0;  
        uso nomi di variabili */  
        (*G).nodes[i].vicinato = 0;  
    }  
}
```

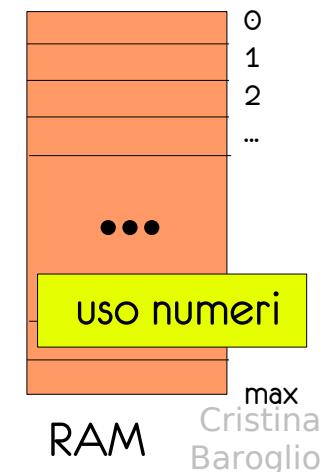
e se ho più esecuzioni contemporanee
dello stesso programma?

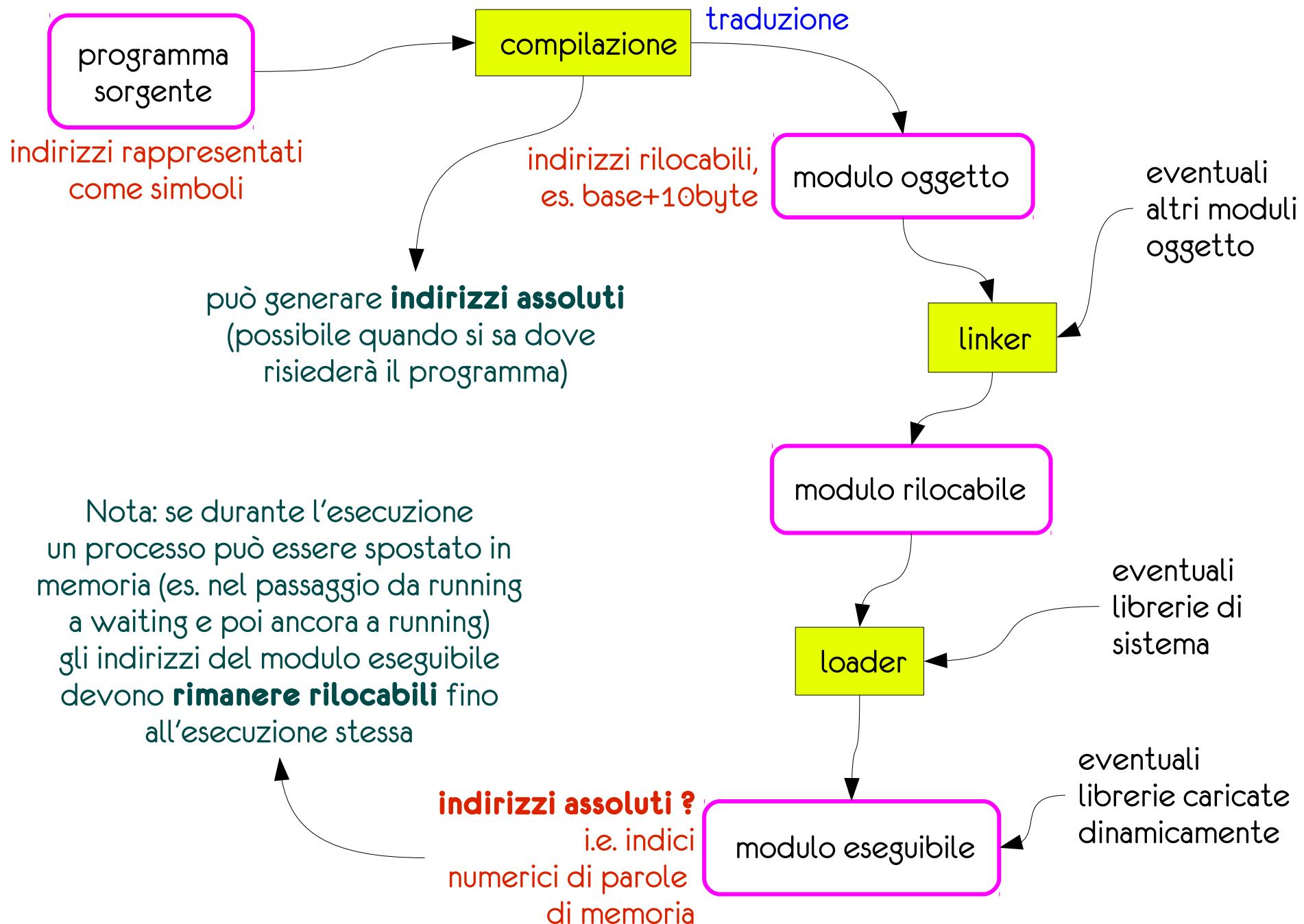
un processo esegue un programma
scritto in un linguaggio di programmazione

all'interno del programma si dichiarano e
si utilizzano delle **variabili** (i, j, G, numN, ...) e
delle **procedure** (somma, cerca, ...)

a livello di RAM si utilizzano degli indirizzi
cioè degli **indici** di parole di memoria

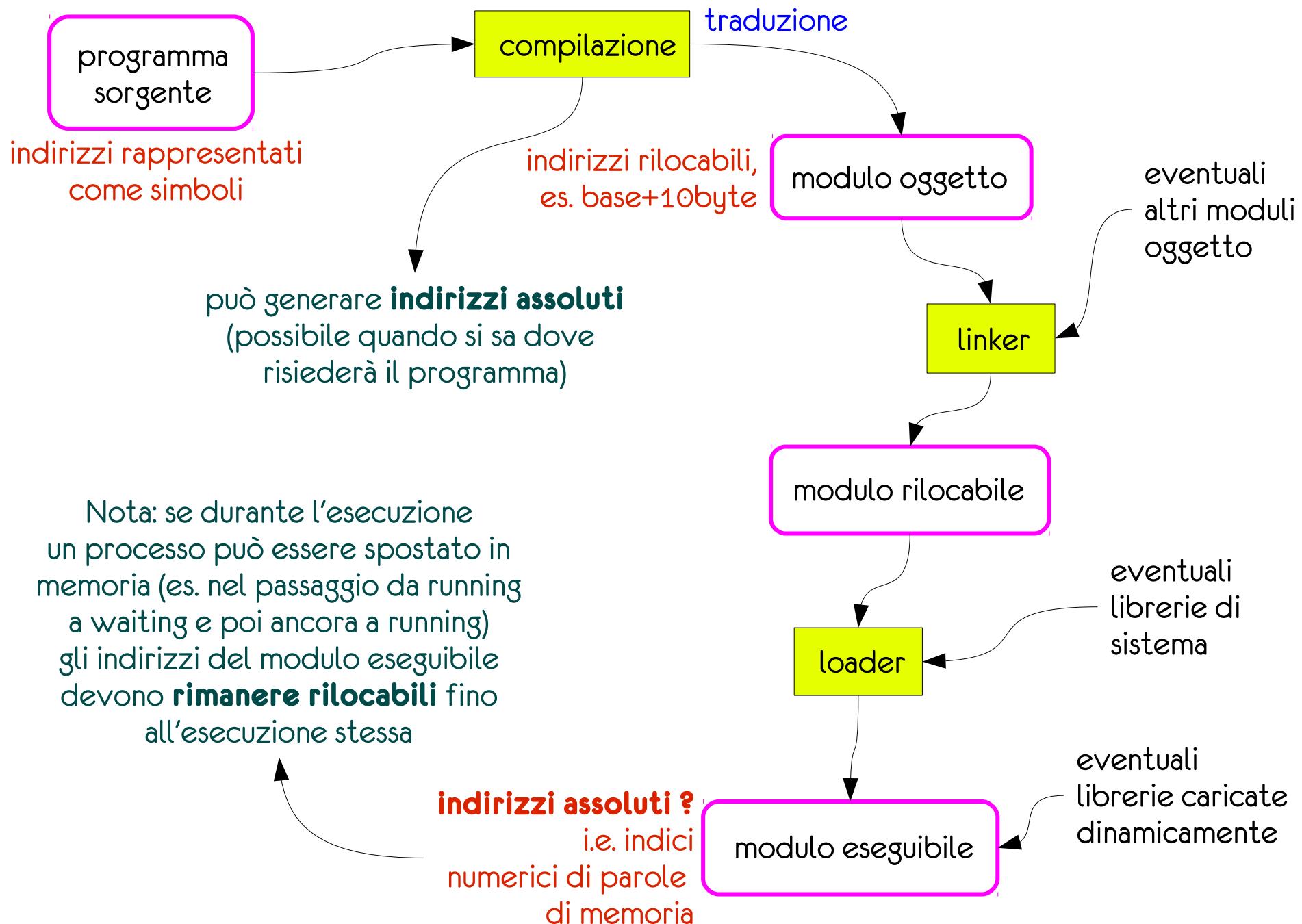
quali passaggi ci sono nel mezzo?





Nota: se durante l'esecuzione un processo può essere spostato in memoria (es. nel passaggio da running a waiting e poi ancora a running) gli indirizzi del modulo eseguibile devono **rimanere rilocabili** fino all'esecuzione stessa

indirizzi assoluti ?
i.e. indici numerici di parole di memoria



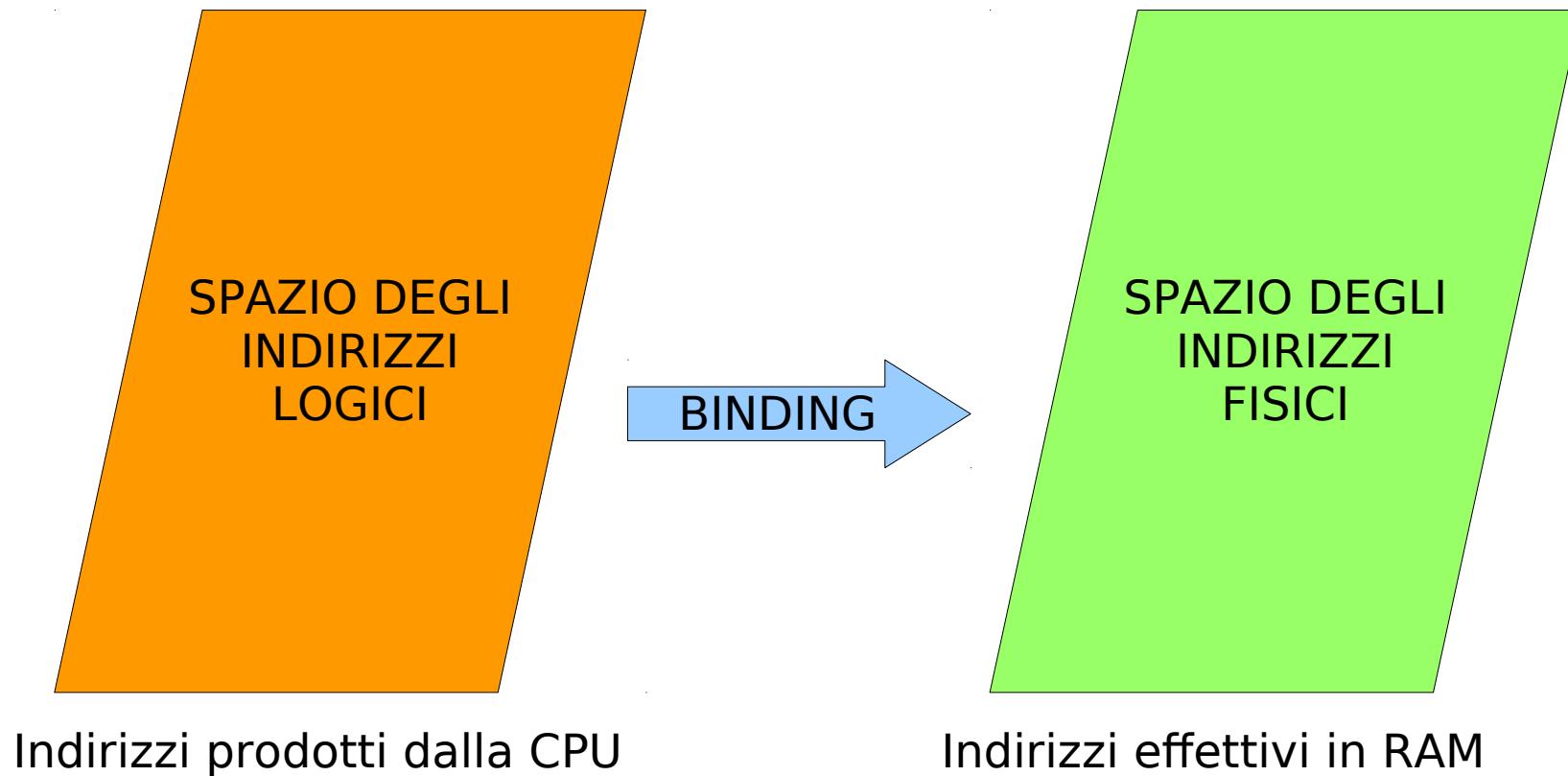
Nota: se durante l'esecuzione un processo può essere spostato in memoria (es. nel passaggio da running a waiting e poi ancora a running) gli indirizzi del modulo eseguibile devono **rimanere rilocabili** fino all'esecuzione stessa

indirizzi assoluti ?
i.e. indici numerici di parole di memoria

Indirizzi logici e fisici

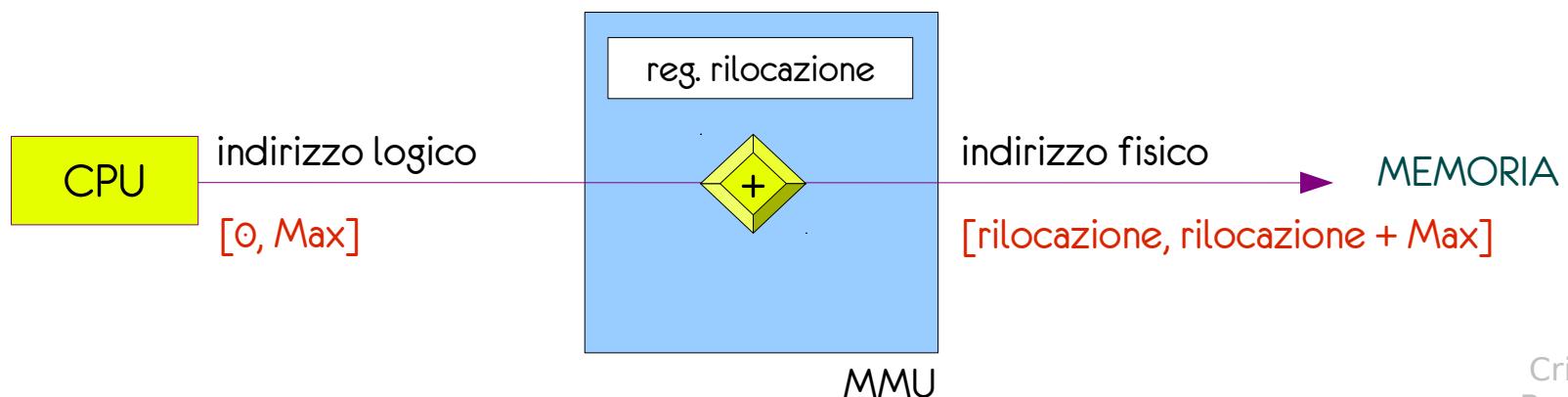
- un indirizzo prodotto dalla CPU è detto **indirizzo logico**
- d'altro canto ogni parola di memoria ha un proprio **indirizzo fisico**

Spazi degli indirizzi di un processo

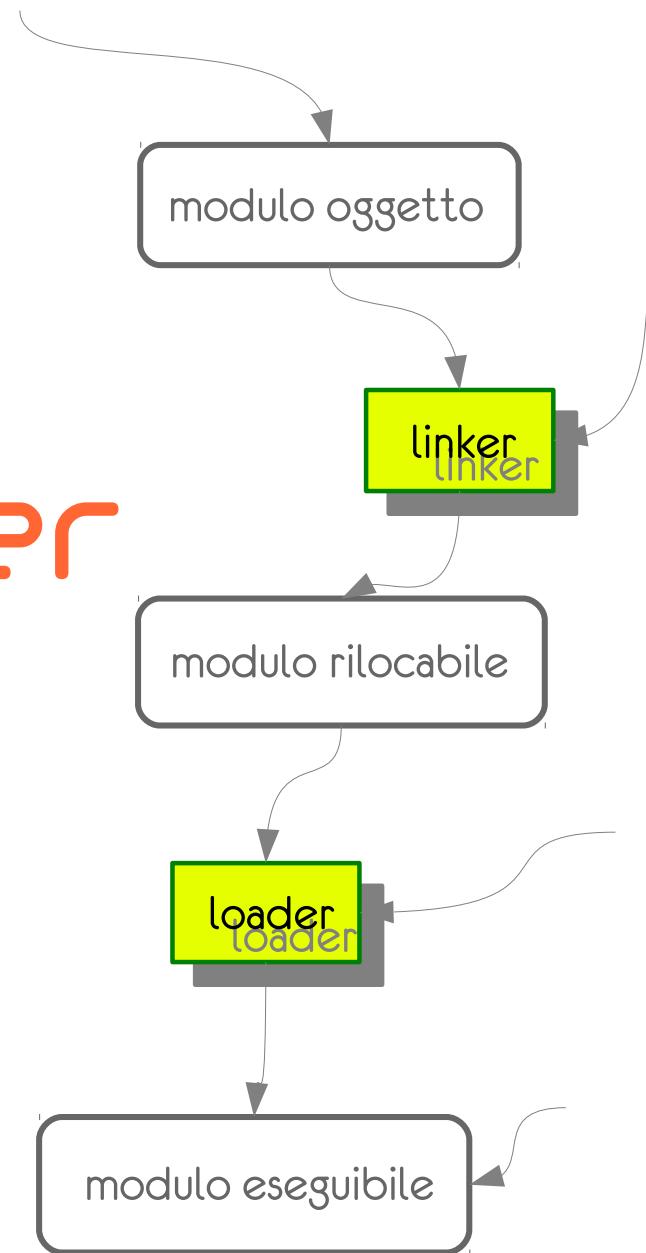


Indirizzi logici e fisici: binding

- **Binding**: mapping dallo spazio degli indirizzi logici di un processo allo spazio dei suoi indirizzi fisici
- quando il **binding** viene fatto a tempo di **compilazione** o di **caricamento**:
 - indirizzo logico = fisico
- quando il **binding** viene fatto a tempo di **esecuzione**:
 - La corrispondenza deve essere calcolata: **lo spazio degli indirizzi logici ≠ dallo spazio degli indirizzi fisici**
- in questo caso il binding è a carico dell'**MMU** (memory management unit). L'MMU può essere realizzato in molti modi il più semplice è una generalizzazione del meccanismo basato sul registro base
- NB: la conversione è fatto SSE serve, cioè SSE si deve accedere alla memoria (lettura/scrittura)

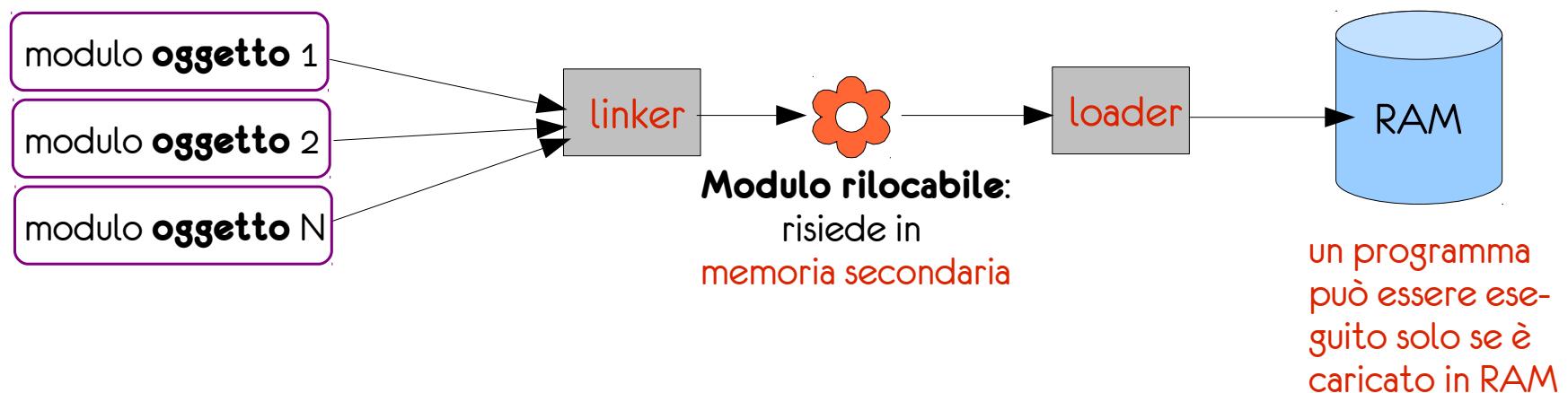


linker e loader

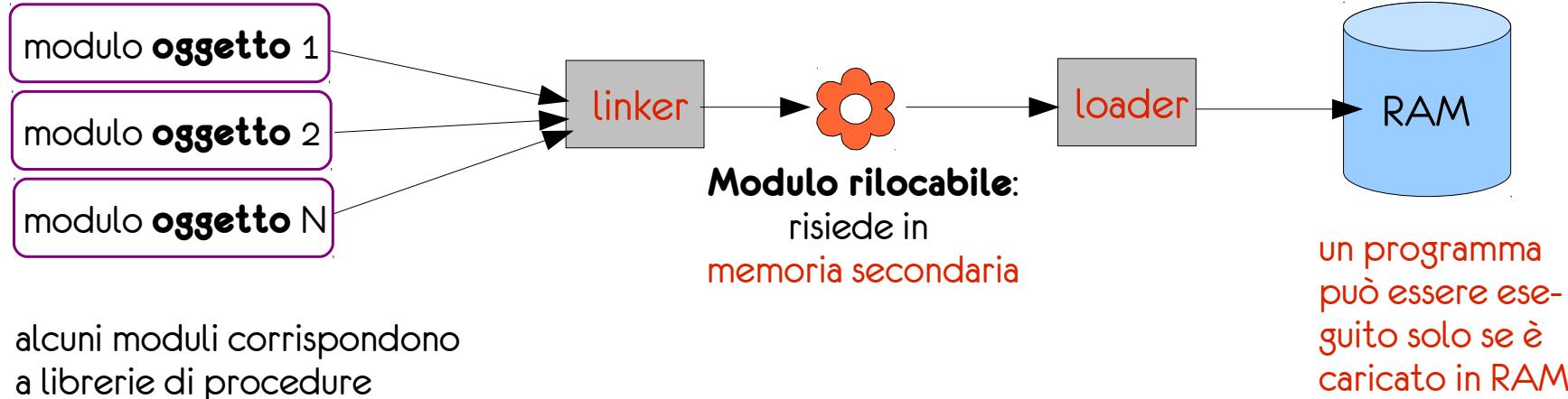


Linking e loading

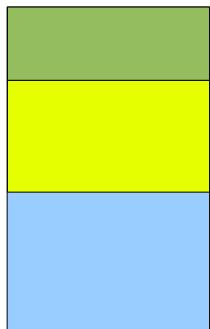
- **linking**: processo di composizione dei moduli che costituiscono un programma; associa ai nomi (di variabili o procedure), utilizzati da ciascun modulo e non definiti in esso, le corrette definizioni
- **loading**: copia un programma eseguibile (o parte di esso) nella RAM
- **linking** e **loading** sono **statici** quando precedono l'esecuzione
- **linking** e **loading** sono **dinamici** quando sono svolti durante esecuzione



Linking, loading e RAM



Approccio tradizionale



inserisco una copia del codice di ogni libreria

L'intero file risultante è caricato in RAM:

- 1) una libreria può essere caricata **molte volte**, una copia per ogni programma che la include
- 2) carico anche le procedure **che non uso**

eseguibile = composizione di

- file oggetto contenente il main
- file oggetti corrispondenti a librerie

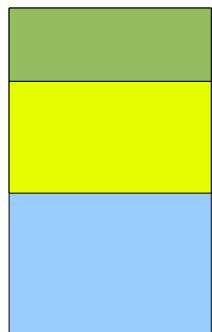
SPRECO !!!

Linking e loading dinamici

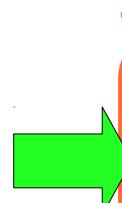
- linking/loading: sono detti dinamici quando sono effettuati nella fase di esecuzione
- **loading dinamico**: una procedura è caricata in RAM quando occorre la sua prima invocazione (al suo primo utilizzo)
- **linking dinamico**: il collegamento del codice di una procedura al suo nome è effettuato **alla sua prima invocazione**. In questo caso il linker statico aggiunge solo uno *stub* della procedura in questione

Loading dinamico

- tutte le procedure risiedono in memoria secondaria sotto forma di **codice rilocabile**
- il codice di una procedura viene caricato nella RAM solo quando la procedura viene chiamata (per la prima volta)
- vantaggio rispetto a caricare un'intera libreria: **si occupa meno RAM**
- nota: occorre che il SO fornisca gli strumenti per realizzare librerie a caricamento dinamico



carico il codice di una procedura solo quando richiamata



In RAM solo una parte di programma:

- una procedura può essere caricata molte volte come parte di programmi diversi
- però carico solo le procedure che uso

eseguibile = composizione di

- file oggetto contenente il main
- rif. ai codici rilocabili

maggior efficienza

Linking dinamico

- Rimanda il collegamento reale di una libreria alla fase di esecuzione
- dopo la compilazione, il linker statico arricchisce il “nucleo” del programma aggiungendo gli *stub* relativi alle procedure appartenenti alle librerie dinamiche usate
- **stub** = codice di riferimento, ha la seguente funzione
 - durante l'esecuzione, lo stub verifica se il codice della procedura è già stato caricato nella RAM:
 - se sì, sostituisce se stesso con l'indirizzo della procedura in questione
 - se no, causa il caricamento del codice della procedura e poi procede con la sostituzione come nel caso precedente
- **NB:** non importa se la procedura di libreria è stata caricata da un altro processo, **tutti i processi fanno riferimento alla stessa copia del codice**, la libreria risulta condivisa

Aggiornamento librerie condivise

- Il linking dinamico ha un notevole vantaggio:
- se **aggiorno una libreria dinamica**, automaticamente tutti i programmi che usano la libreria faranno riferimento alla nuova versione anche **senza ricompilare** (linking e compilazione sono spesso eseguiti entrambi dal compilatore)!
- se il linking fosse solo statico dovrei invece aggiornare il collegamento della libreria al programma, prima di utilizzare il medesimo

allocazione della ram

capitolo 8 del libro (VII ed.), da 8.3

Sommario

- Lasciamo il Livello 0 per **salire d'astrazione**, passiamo al Livello 1
- Affronteremo ora il problema della **gestione della memoria principale** e, in particolare, della **scelta dell'area da assegnare a un processo** (quale memoria, quanta memoria, organizzata come)
- Tre approcci:
 - 1) Allocazione contigua
 - 2) Paginazione
 - 3) Segmentazione
- Ogni approccio fa riferimento a un modello di rappresentazione, che richiede apposite strutture dati e meccanismi di gestione

Allocazione contigua

- Allocazione contigua
 - rilocazione e protezione
 - partizioni multiple
 - frammentazione

Introduzione



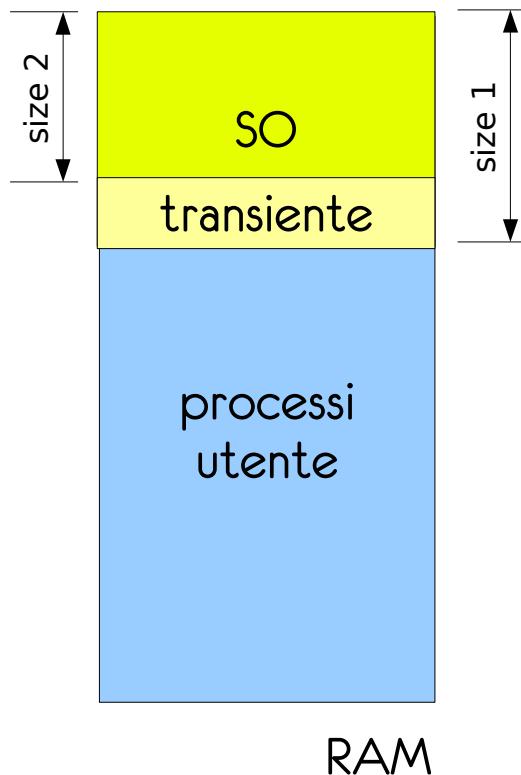
Nel modello ad allocazione contigua si suddivide la RAM in due parti:

- una riservata al SO, posizionata solitamente in memoria bassa (la posizione dipende dalla posizione del vettore delle interruzioni)
- l'altra riservata ai processi utente

occorre proteggere la partizione di memoria riservata al SO da letture/scritture ad opera di processi utente. Inoltre devo proteggere in modo analogo le aree di RAM riservate ai diversi processi utente

Ciò è facilmente realizzabile utilizzando un **registro di rilocazione** per la conversione di indirizzi logici in indirizzi fisici

Introduzione



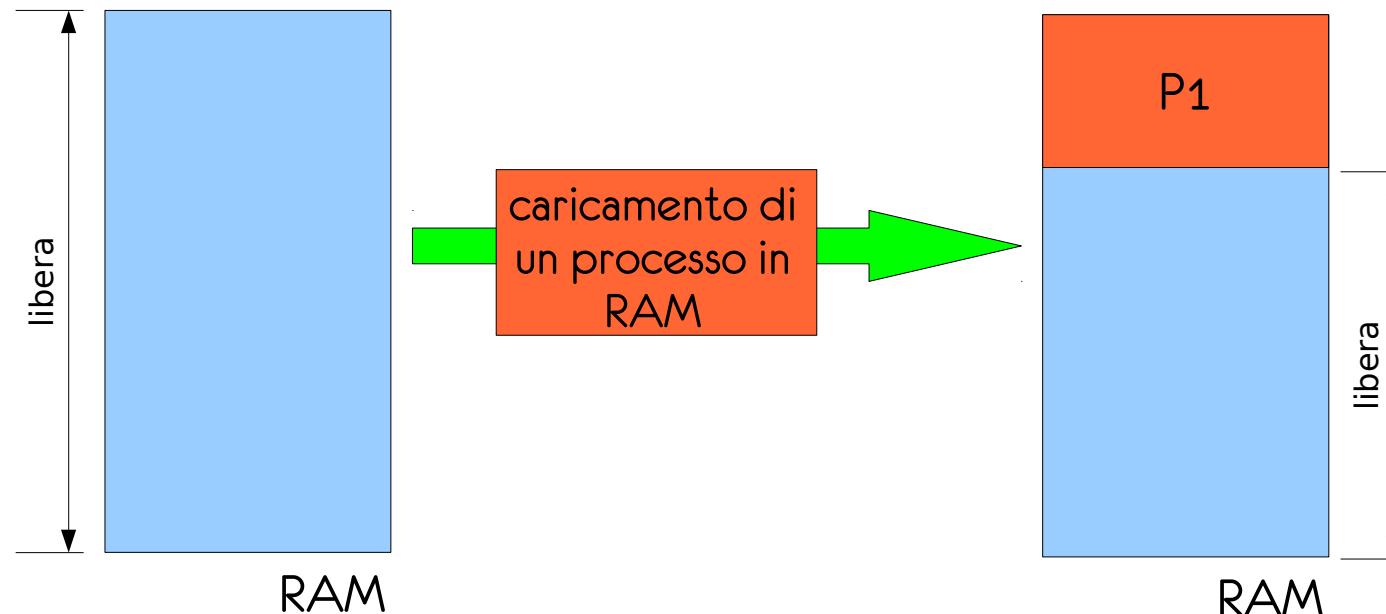
Il codice di SO caricato può a sua volta essere suddiviso in un nucleo di base sempre necessario e una parte che può essere utile o meno a seconda delle circostanze (**codice transiente**).

Il codice transiente può essere rimosso dalla RAM quando non serve e aggiunto quando serve

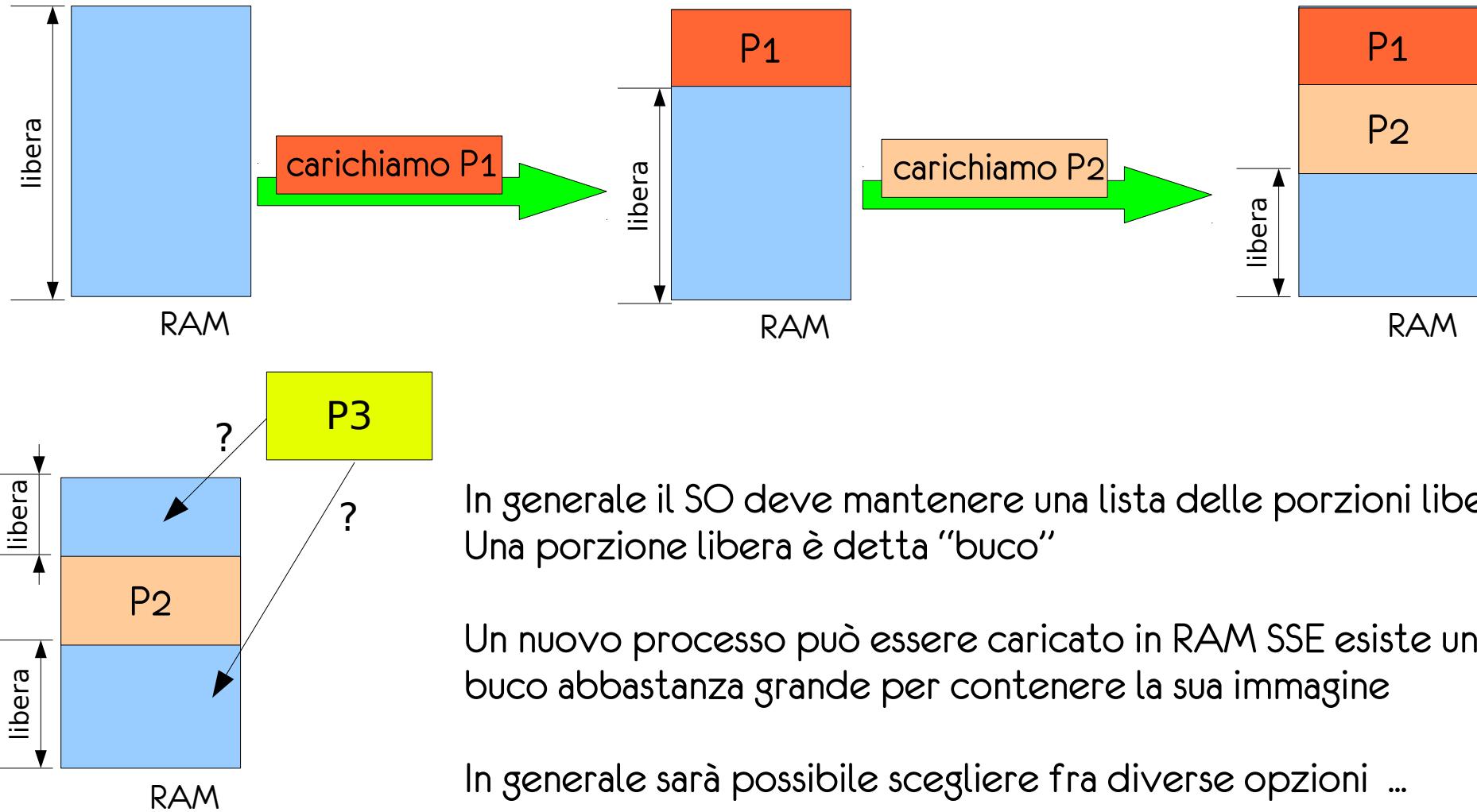
In queste circostanze occorre poter modificare la partizione riservata al SO

Allocazione a partizioni multiple

- Vediamo ora come funziona il meccanismo di allocazione della memoria ai processi nel modello ad allocazione contigua
- Lo schema seguito si chiama “a partizioni multiple”
- All'inizio tutta la RAM (esclusa la porzione per il SO) è libera:



Allocazione a partizioni multiple



Criteri di scelta

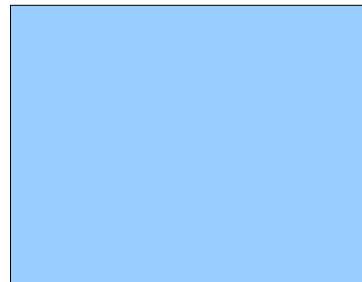
- **Best-fit**: scelgo la porzione più piccola fra quelle adeguate a contenere l'immagine del processo
- **First-fit**: scelgo la prima porzione sufficientemente grande, trovata scandendo la lista dei buchi liberi
- **Worst-fit**: scelgo la porzione più grande fra quelle libere

Criteri di scelta

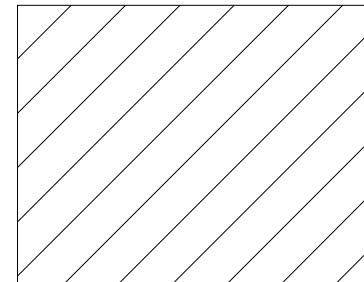
- **Qual'è la migliore?** Per capirlo occorre introdurre la nozione di **frammentazione della memoria**
- Di per sé per **frammentazione** si intende lo spezzettamento della memoria in tante parti. Si dice che si ha:
 - **frammentazione esterna**, se queste parti sono abbastanza grandi da essere utilizzabili (es. per contenere un processo)
 - **frammentazione interna**, se sono molto piccoli, praticamente inutilizzabili. In questo caso il frammento viene unito alla partizione precedente.

Criteri di scelta

- La frammentazione è un problema.
- L'analisi statistica mostra che con il **first-fit**, ogni N blocchi di memoria allocati si perde uno spazio pari a 0.5^*N blocchi a causa della frammentazione (**regola del 50%**)
- In pratica 1/3 della memoria risulta inutilizzabile



RAM ALLOCATA



RAM INUTILIZZABILE

Criteri di scelta

- In generale worst-fit è la strategia peggiore,
- First-fit e best-fit non sono sempre l'uno meglio dell'altro però computazionalmente first-fit è una tecnica meno costosa

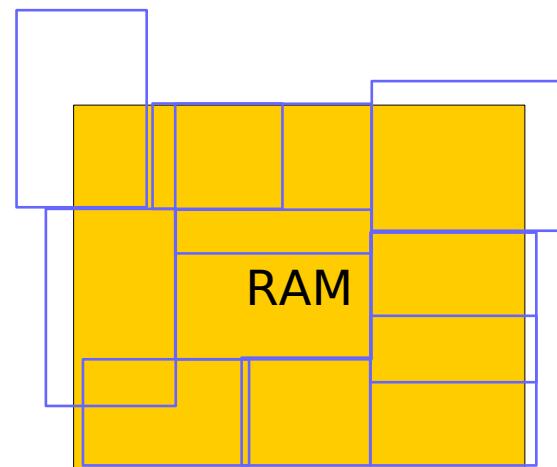
Combattere la frammentazione

- È possibile combattere la frammentazione attuando di tanto in tanto una **politica di compattamento**: spostare le immagini dei processi in memoria dimodoché risultino **contigue**
- Il compattamento è applicabile solo se il binding fra indirizzi logici e fisici è effettuato a tempo di esecuzione

allocazione contigua, binding e rilocazione

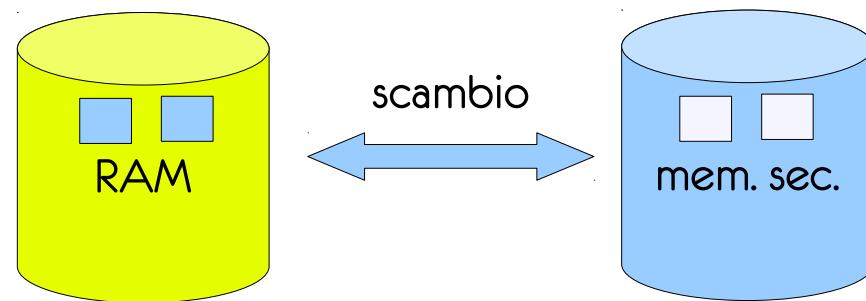
Swapping

- Scheduling di medio termine (già accennato)
- La **RAM ha dimensione limitata**
- Può succedere che i processi running e ready siano così tanti da richiedere complessivamente una quantità di memoria maggiore di quella offerta dalla RAM

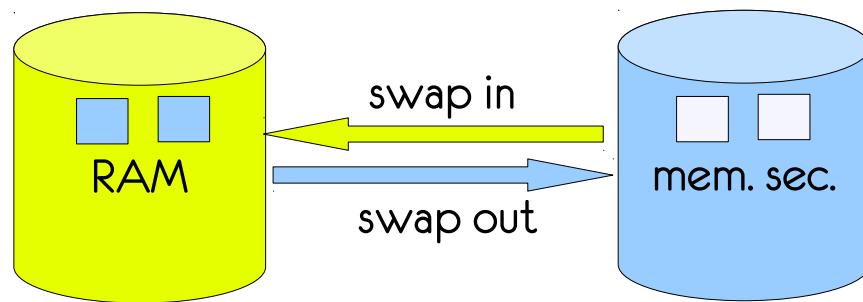


Swapping

- Scheduling di medio termine (già accennato)
- Soluzione: mantenere una parte dei processi ready in memoria secondaria ed effettuare di tanto in tanto lo swapping (lo scambio) fra processi in RAM e processi in memoria secondaria



Swapping



- **swap in:** carico l'immagine di un processo ready da memoria secondaria (anche detta **backing store**) in RAM
- **swap out:** scarico l'immagine di un processo che non è in esecuzione in memoria secondaria
- per motivi di efficienza è importante che i processi in testa alla ready queue siano conservati nella RAM, gli altri possono essere conservati in memoria secondaria

Swapping e binding

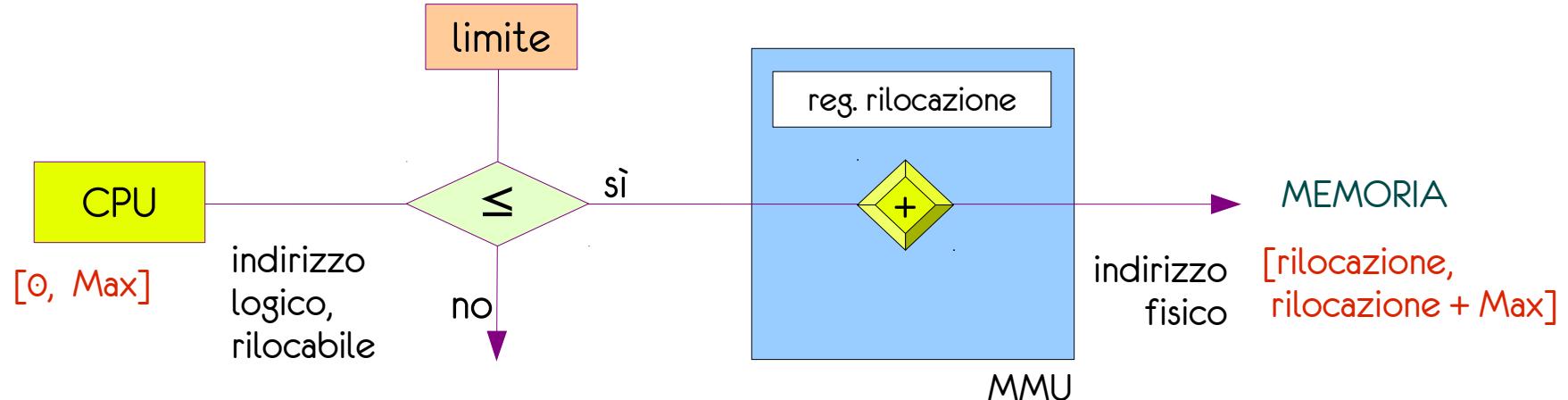
- L'immagine di un processo può fare swap-out e dopo un po' essere ricaricata in RAM (es. round-robin)
- In questo caso posso ricollocarla in una porzione qualsiasi della RAM?

Swapping e binding

- La collocazione dipende dal **quando** viene effettuato il binding delle variabili:
 - se il **codice non è rilocabile** allora l'immagine del processo dovrà rioccupare la stessa sezione di RAM
 - se è **rilocabile** (in particolare, se il binding è dinamico) questo non è necessario

- Si può implementare la rilocazione se la RAM è gestita secondo il modello dell'allocazione contigua?
- Come avviene il binding?

Rilocazione e protezione



registro limite e registro di rilocazione vengono caricati durante il context switch
il contenuto del registro di rilocazione può variare nel tempo

L'uso dei registri limite e di rilocazione è possibile solo se l'HW dispone di queste strutture \Rightarrow la realizzazione dell'approccio a memoria contigua può essere realizzato solo previa presenza di un adeguato supporto HW

qualche dettaglio sullo
swapping

Tempo di swapping

- Il tempo necessario al completamento dello swapping è dato dal **tempo di swap-out + tempo di swap-in**
- dipende dalla **dimensione delle immagini dei processi** coinvolti e dal **tempo di trasferimento da/a memoria secondaria**
- **Esempio:** se il tempo di trasferimento è pari a 1MB/sec, di quanto tempo ho bisogno per trasferire un processo con un'immagine da 100KB?

$$100\text{KB}/(1\text{MB/sec}) = (100\text{KB}/1000\text{KB})\text{sec} = 0.1\text{sec} = 100\text{msec}$$

- di solito si usano i millisecondi come unità di misura
- ...

Tempo di swapping

- Supponendo identici tempo di swap-out e tempo di swap-in complessivamente occorreranno circa 200 msec
- il “circa” è dovuto al tempo necessario a posizionare la testina del disco

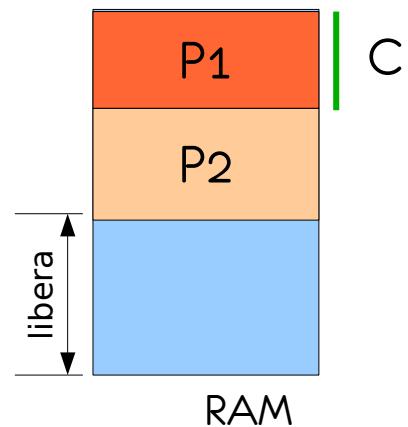
Commenti

- Un processo può essere oggetto di swapping **SSE non ha in atto operazioni di I/O** perché le operazioni di I/O non possono essere effettuate su variabili residenti in memoria secondaria
- **Esempi**
 - Unix (prime versioni): di base lo swapping era disabilitato, si attiva solo quando il carico del sistema è molto elevato
 - Windows 3.1: lo swapping era attivato solo a carico elevato ed era effettuato manualmente dall'utente

■ fine introduzione sulla gestione della RAM

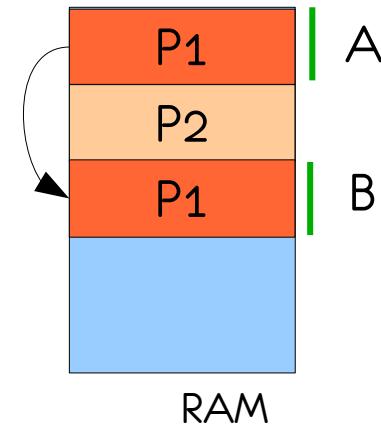
Paginazione della memoria

- La paginazione è un meccanismo di gestione della RAM alternativo all'allocazione contigua
- detto “**spazio degli indirizzi di un processo**” l'**insieme di tutti gli indirizzi a cui il processo ha accesso**, caratteristica fondamentale della paginazione è che essa consente allo spazio degli indirizzi fisici di un processo di non essere contiguo



allocazione contigua

sp. indirizzi di P1 = C



paginazione

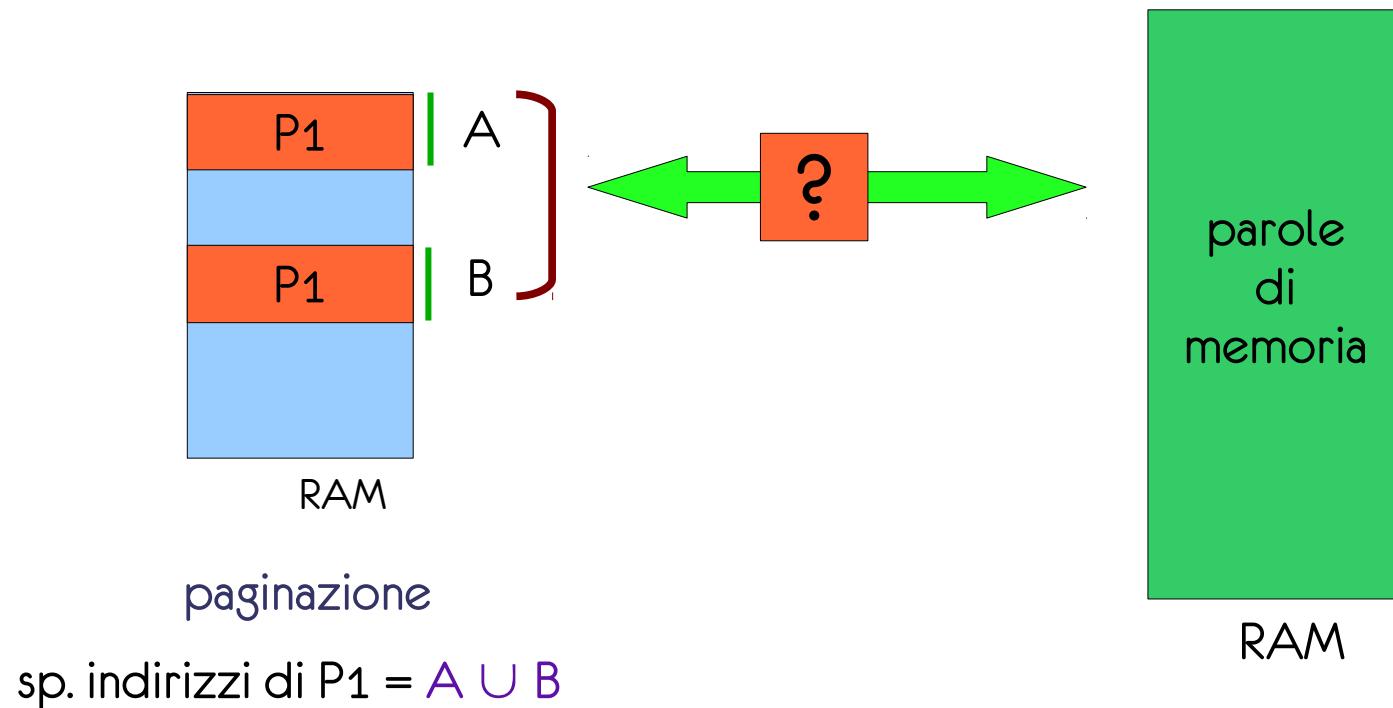
sp. indirizzi di P1 = A \cup B

È UN VANTAGGIO?

Paginazione

- È un **vantaggio**
- Consente di far (de)crescere in modo dinamico lo spazio riservato a un processo, semplicemente (togliendo) aggiungendo delle pagine
- Quindi per esempio posso mantenere in RAM solo una porzione del codice di un processo, aggiungendo via via altre parti utili, con riferimento all'esecuzione corrente
- La gestione del **codice transiente** del SO diventa molto più semplice e naturale
- Paginazione e architettura di una macchina sono strettamente correlate: **la paginazione è possibile solo avendo un opportuno supporto hardware**
- Vediamo ora le strutture necessarie per realizzare questo modello ...

Pagine e strutture di supporto

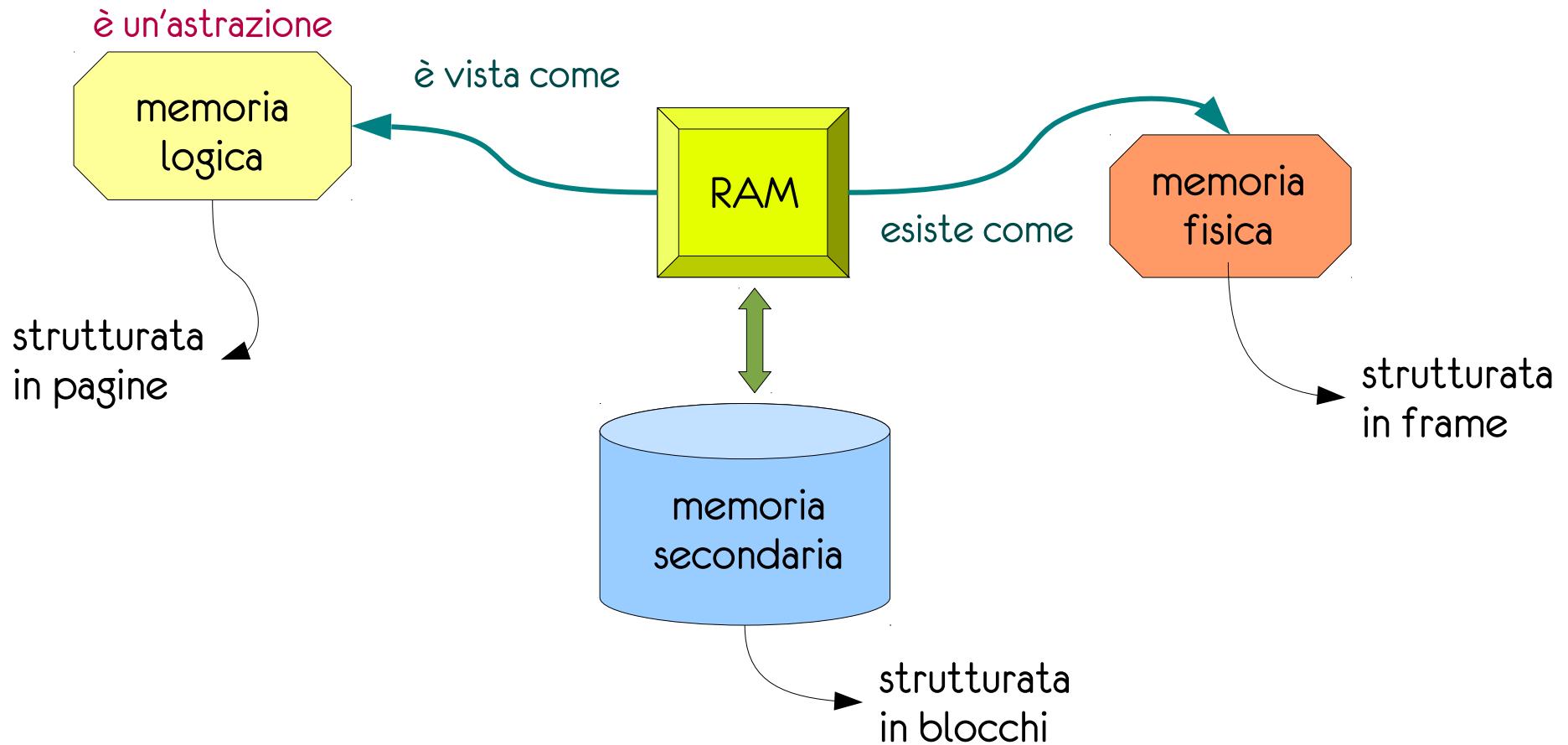


(1) Come posso associare porzioni di processo a porzioni di RAM?

(2) Come posso organizzare le diverse porzioni in un tutt'uno?

Vedere la RAM come array non è più così comodo ...

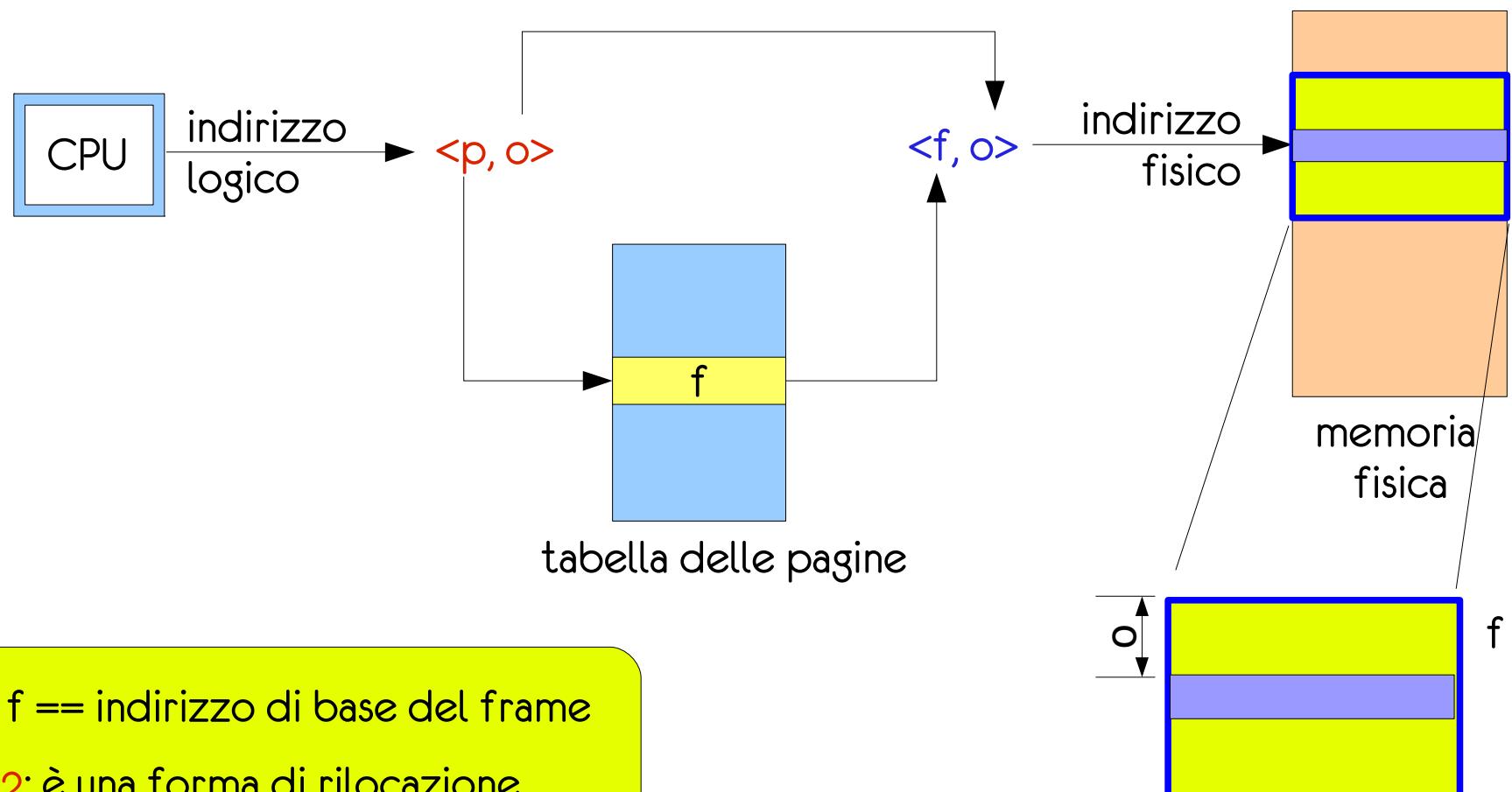
Pagine, frame e blocchi



blocchi, frame e pagine sono tutti termini che fanno riferimento a porzioni di memoria
di **uguali dimensioni** ma appartenenti a viste/elementi diversi

Pagine e frame

Il primo tipo di struttura di cui abbiamo bisogno serve a fare il binding fra indirizzi logici e indirizzi fisici. In questo contesto un **indirizzo logico** è una coppia `<numero_di_pagina, offset>`, un **indirizzo fisico** è una coppia data a `<id_frame, offset>`



Indirizzi logici

- La dimensione è la stessa per tutte le pagine ed è definita dall'architettura; è una potenza di 2 normalmente compresa fra 512 byte e 16 MB
- Supponiamo che la **dimensione di una pagina** sia 2^n e la **dimensione della memoria logica** sia 2^m , in quante pagine sarà suddivisa la memoria logica?

$$2^m / 2^n = 2^{m-n}$$

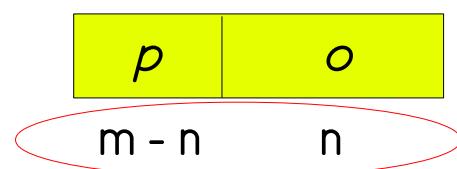
- **quanti bit** servono per rappresentare un numero di pagina?

$$m - n$$

- **quanti bit** occorrono quindi per rappresentare lo scostamento all'interno di una pagina?

$$n$$

- quindi un indirizzo logico:



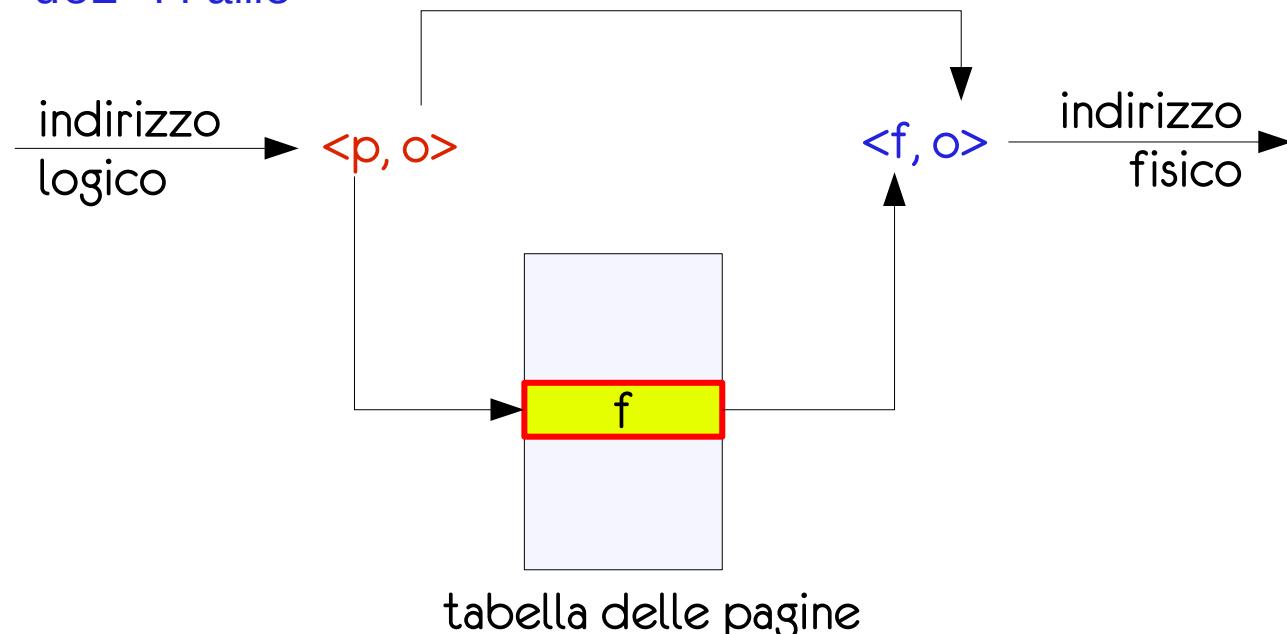
Nota: è giusto che la somma faccia m?

Indirizzi logici: esempio

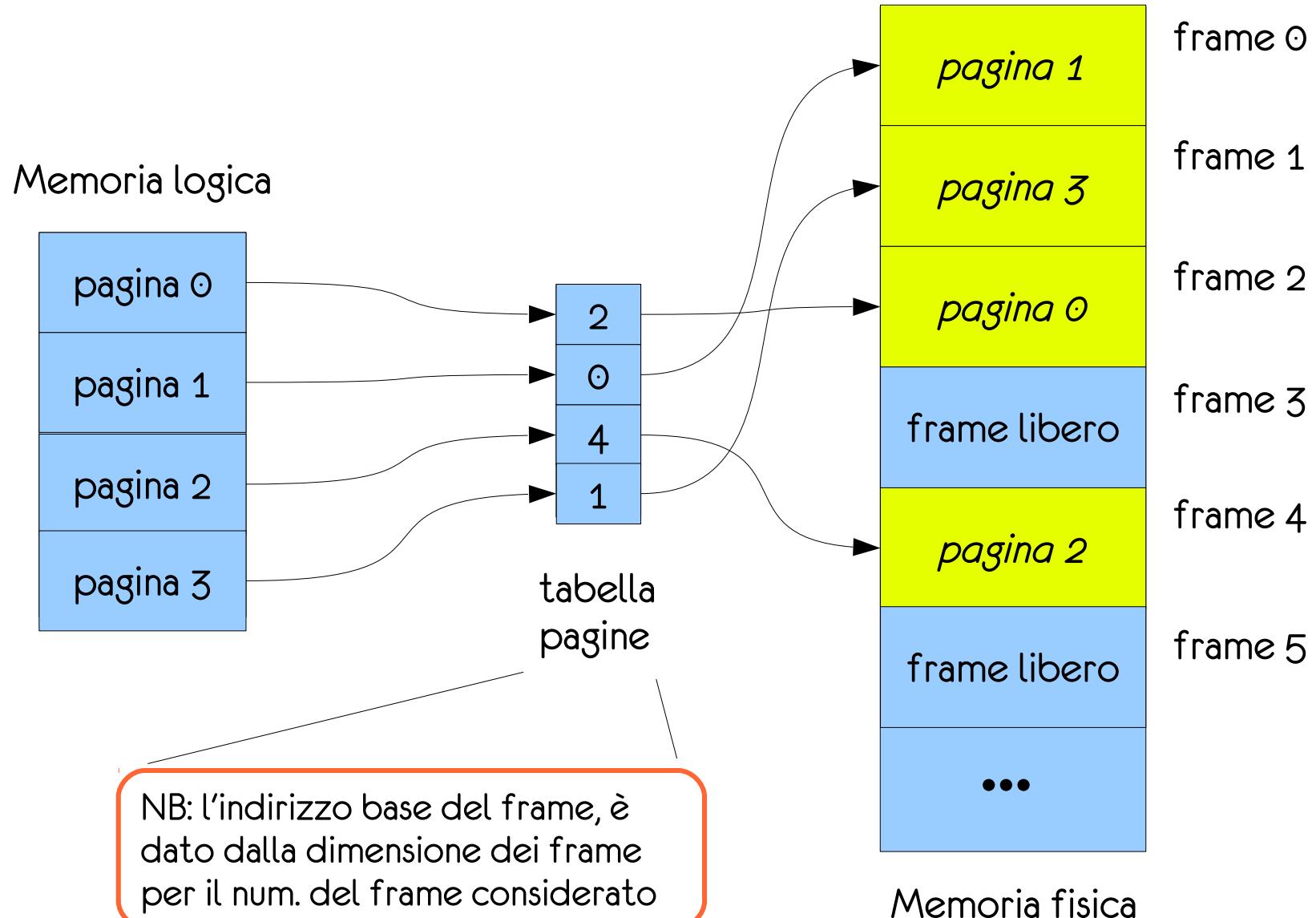
- Supponiamo di avere pagine di dimensione 512 byte e una RAM di dimensione 1MB, quante pagine avremo? Quanti bit occorrono per rappresentare indirizzi logici in questo contesto?
- Innanzi tutto devo riportarmi a potenze di 2 nella stessa unità di misura:
 - **pagina**: 512 byte = 2^9 byte
 - **memoria logica**: 1MB = 2^{20} byte
- Ora possiamo fare i conti:
 - **numero di pagine necessarie**: $2^{20} / 2^9 = 2^{20-9} = 2^{11}$
 - per rappresentare il numero di pagina occorrono **11 bit**
 - per rappresentare l'**offset** occorrono): **9 bit**
 - num bit per le pagine + num bit x l'offset = $11 + 9 = 20$

Paginazione e rilocazione

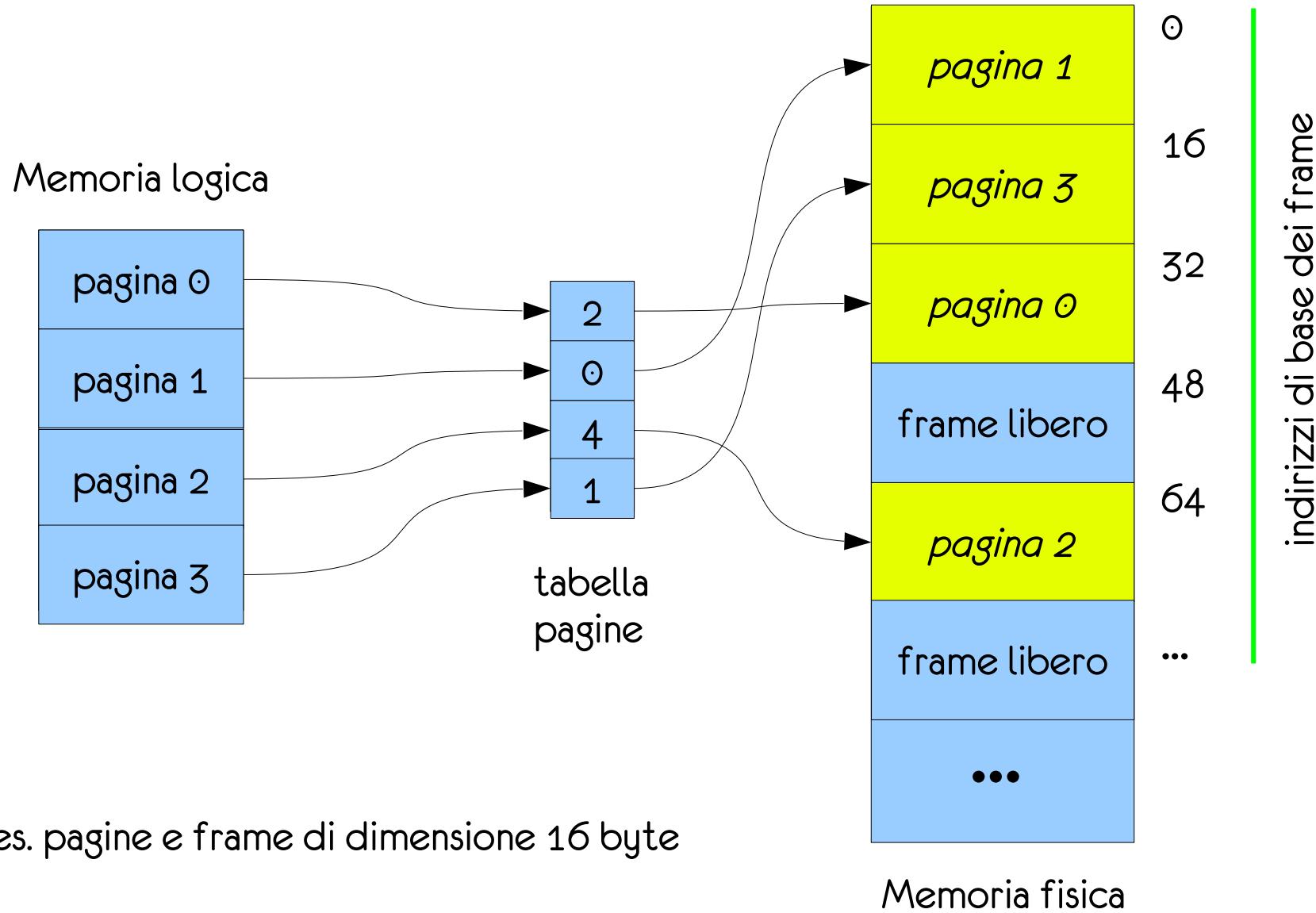
- Cosa vuol dire “**rilocare il codice**” in questo contesto?
consentire l'accesso a una pagina, indipendentemente dal frame in cui è caricata
- **Si può realizzare la rilocazione?** Sì, il registro di rilocazione è sostituito dalla entry nella tabella delle pagine, corrispondente a p. Il valore f individua l'**indirizzo di inizio del frame**



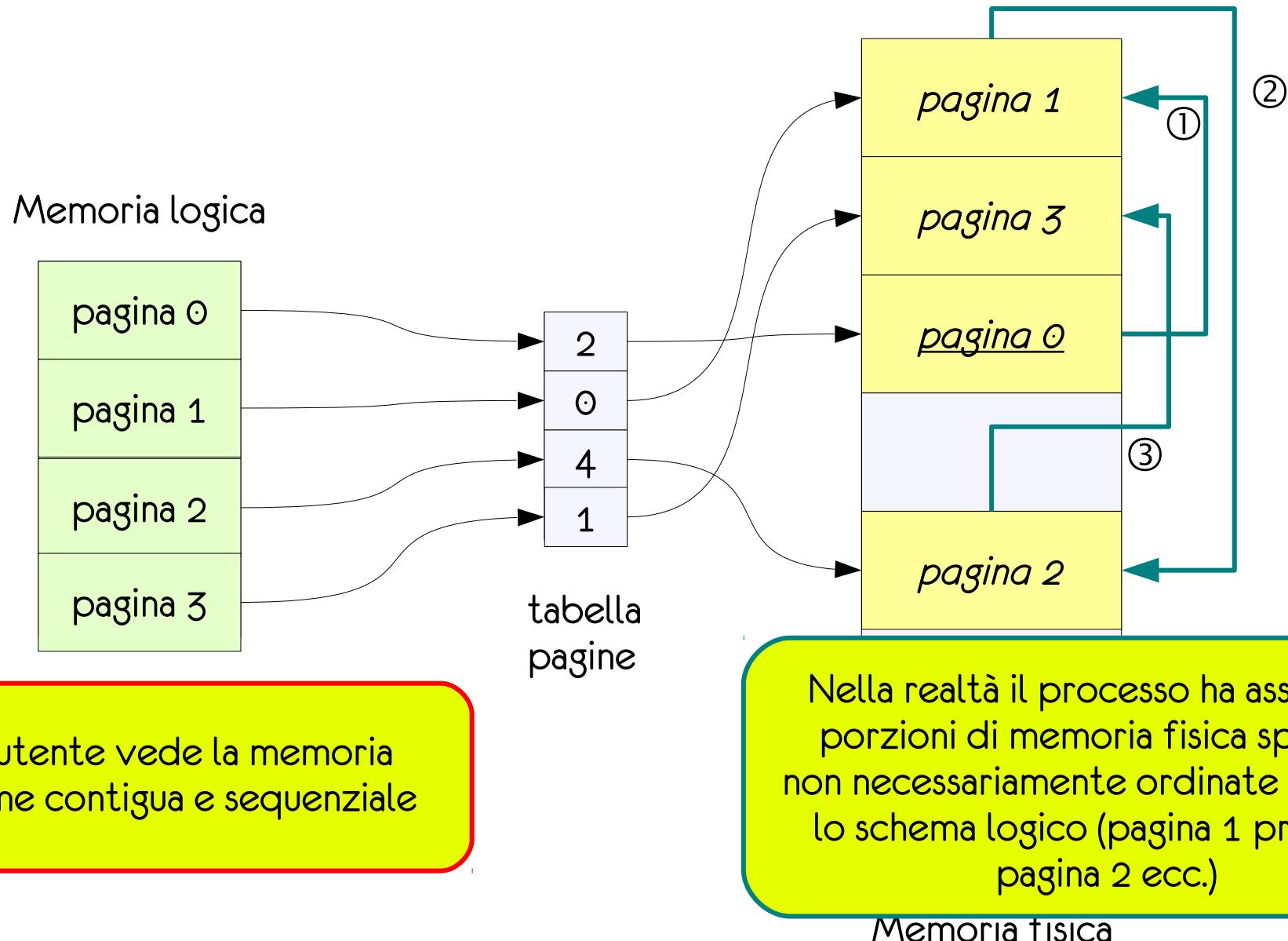
Esempio di paginazione



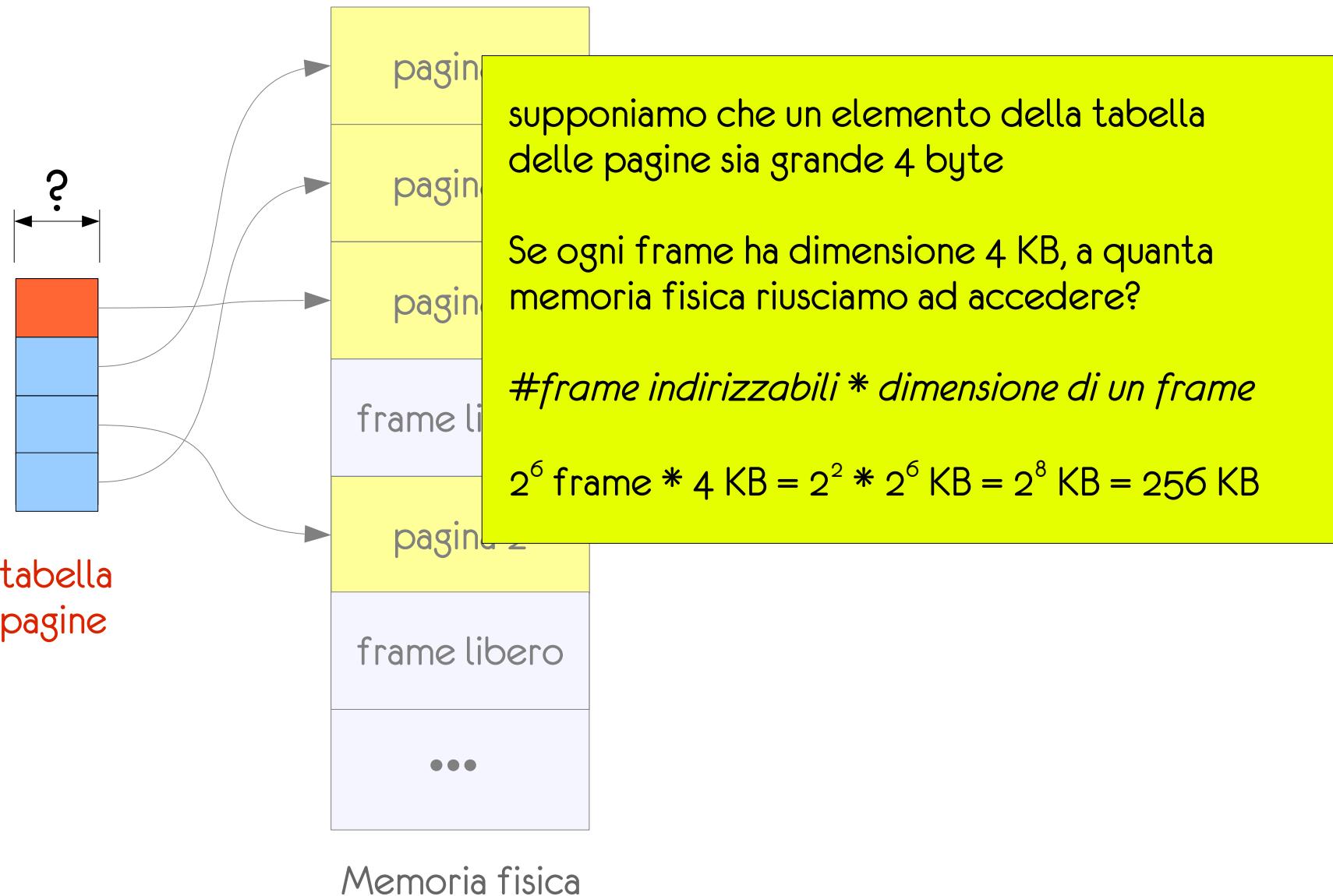
Esempio di paginazione



Esempio di paginazione



Qualche conto



Paginazione e frammentazione

- La paginazione elimina il problema della frammentazione esterna però permane il problema della **frammentazione interna**
- Ogni processo può avere allocate un numero di pagine diverso, quante di queste presenteranno frammentazione interna?
- Soltanto l'ultima perché raramente la dimensione di un processo sarà un multiplo della dimensione di una pagina, quindi in generale avrò bisogno di *N pagine più un pezzetto* per ciascun processo
-

Paginazione e frammentazione

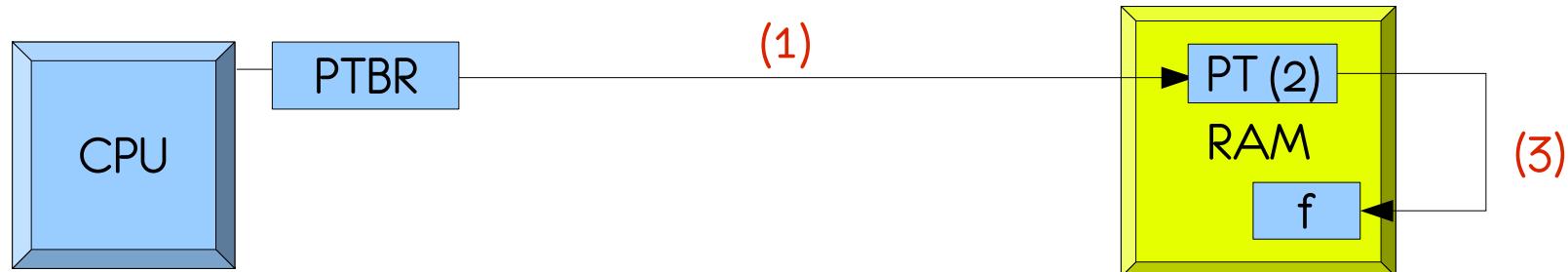
- **Frammentazione interna:** in media possiamo dire che avremo $\frac{1}{2}$ pagina non utilizzata per ogni processo
- **Dimensione ottimale delle pagine:** occorre trovare un compromesso fra limitare il problema della frammentazione e ridurre i costi dell'I/O:
 - **pagine piccole:** riducono la frammentazione
 - **pagine grandi:** migliori quando occorre trasferire da/a memoria secondaria grosse quantità di dati (carico una pagina invece di tante in sequenza)

Quante tavelle delle pagine?

- ogni processo in RAM ha la propria tabella delle pagine ∈ PCB
- inoltre il SO mantiene (in copia unica) numero e lista dei frame liberi (**tabella dei frame**). Quando si carica un processo:

```
if (#frame liberi ≥ #pagine del processo) {  
    foreach (pagina da caricare) {  
        <<crea una entry nella tab. delle pag. del processo>>  
        <<nuova entry = num. di un frame libero>>  
        <<aggiorna tabella dei frame>>  
        <<carica pagina>>  
    }  
}  
  
else ???  
/* per ora diciamo che il processo non verrà caricato */
```

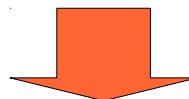
Problema: tempi di accesso



(1) tramite PTBR accedo alla RAM per individuare la page table

(2) tramite la page table costruisco l'indirizzo fisico a cui accede

(3) accedo all'indirizzo fisico di interesse

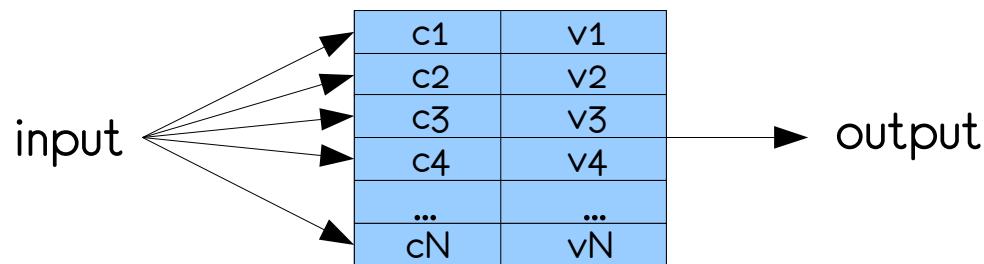


Il numero di accessi alla RAM è raddoppiato perché ogni volta devo prima accedere alla tabella delle pagine (PT) e poi all'indirizzo che mi interessa

Il raddoppio degli accessi è inaccettabile, rallenta troppo l'esecuzione

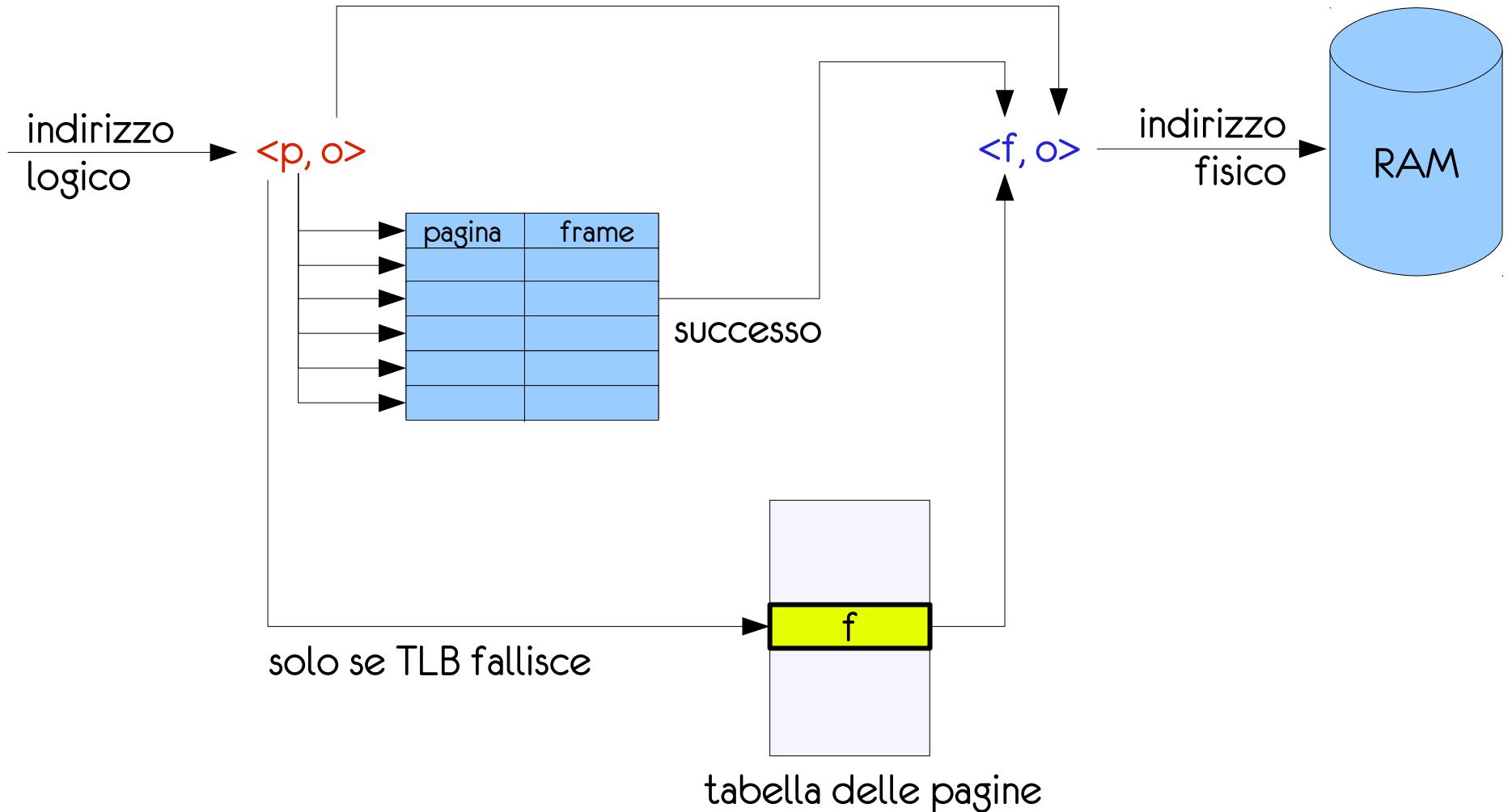
Soluzione: Translation look-aside buffer

TLB (translation look-aside buffer): è una cache molto veloce contenente delle coppie **<chiave, valore>**. Quando riceve un input lo confronta contemporaneamente con tutte le chiavi. Se trova una chiave corrispondente restituisce il valore associato.



Molte architetture adottano un TLB in associazione alla tabella delle pagine per limitare il problema del doppio accesso alla RAM, chiavi = numeri di pagina, valori = # frame

Schema d'uso



Hit ratio

- Col termine **hit ratio** si intende la percentuale di **successo** relativo al reperimento di una pagina tramite TLB (la pagina è accessibile tramite il TLB)
- Dire che l'hit ratio è pari al 92% significa che in 92 casi su 100 il frame è stato individuato attraverso il TLB e solo nei restanti 8 si è dovuto accedere alla tabella delle pagine
- L'hit ratio consente di calcolare il **tempo medio effettivo di accesso** a una pagina

Hit ratio

Esempio

- Supponiamo che il tempo di accesso al TLB sia di 20 nsec mentre l'accesso alla RAM richiede 100 nsec
- Proviamo a calcolare il tempo medio effettivo necessario per accedere a una pagina nel caso in cui l'hit ratio è pari all'82% e nel caso in cui è pari al 95%

Hit ratio

- **Ipotesi:**

- tempo di accesso al TLB = 20 nsec
- tempo di accesso alla RAM = 100 nsec
- hit ratio = 82% = 0.82
(oppure 95%=0.95)
- percentuale di accessi tramite page table = 18% = 0.18
(oppure 5%=0.05)

Hit ratio

- **Calcolo**

- tempo di accesso a una pagina tramite TLB:
 - $T_{tlb} = \text{accesso a TLB} + \text{accesso RAM} = 20 + 100 \text{ nsec} = 120 \text{ nsec}$
- tempo di accesso a una pagina tramite tabella delle pagine:
 - $T_{pt} = 2 \times \text{accesso RAM} = 100 + 100 \text{ nsec} = 200 \text{ nsec}$
 - 82%: t.di a. medio = $(0,82 * T_{tlb}) + (0,18 * T_{pt}) = 0,82 \times 120 + 0,18 \times 200 = 134,4 \text{ nsec}$
 - 95%: t.di a. medio = $(0,95 * T_{tlb}) + (0,05 * T_{pt}) = 0,95 \times 120 + 0,05 \times 200 = 124 \text{ nsec}$

Aggiornamento del TLB

- Quando si inserisce una nuova coppia nel TLB?
- Quando non trovo una pagina di interesse (*TLB miss*):
 - se c'è spazio nel TLB si inserisce la nuova coppia <pagina, frame>
 - se non c'è spazio, si sostituisce una coppia già presente con quella nuova, di solito viene scelta per la sostituzione la coppia usata meno di recente
- l'idea di fondo è che quando accedo ad una nuova pagina, verosimilmente l'avrò bisogno per un po' di tempo
- **NB:** alcune delle pagine usate come chiavi nel TLB corrispondono ai processi principali del SO, le loro entry sono considerate non sovrascrivibili -> **ho pagine di processi diversi!!!**

Context switch

- Il TLB contiene dati (quasi) esclusivamente concernenti il processo running
- Quando si effettua un context switch occorre aggiornare il contenuto del TLB? Vorrei attuare un meccanismo di protezione che impedisca a un processo di accedere alle pagine di un altro

Context switch

- Due possibili soluzioni:
 - Alcuni TLB arricchiscono l'informazione contenuta in essi, aggiungendo un **ASID** (identificatore univoco dello spazio degli indirizzi di un processo). Un ASID identifica in modo univoco un processo. Uso l'informazione relativa a una pagina nel TLB, solo se l'ASID del processo running corrisponde all'ASID della pagina
 - Soluzione alternativa: il dispatcher **svuota il TLB ad ogni context switch**

Protezione

- Un aspetto ulteriore concerne la **modalità di accesso** alle pagine, memorizzata nella tabella delle pagine del processo
- Una pagina può essere accessibile in **lettura**, **lettura-scrittura**, **esecuzione**, oppure essere **invalida** (non appartiene allo spazio degli indirizzi del processo).
- Si possono avere pagine non valide quando la tabella delle pagine di un processo è più piccola dello spazio riservato ad essa nella RAM
- Se un processo cerca di accedere a una pagina non valida, viene generato un interrupt
- La **modalità di accesso** viene aggiunta alla tabella delle pagine

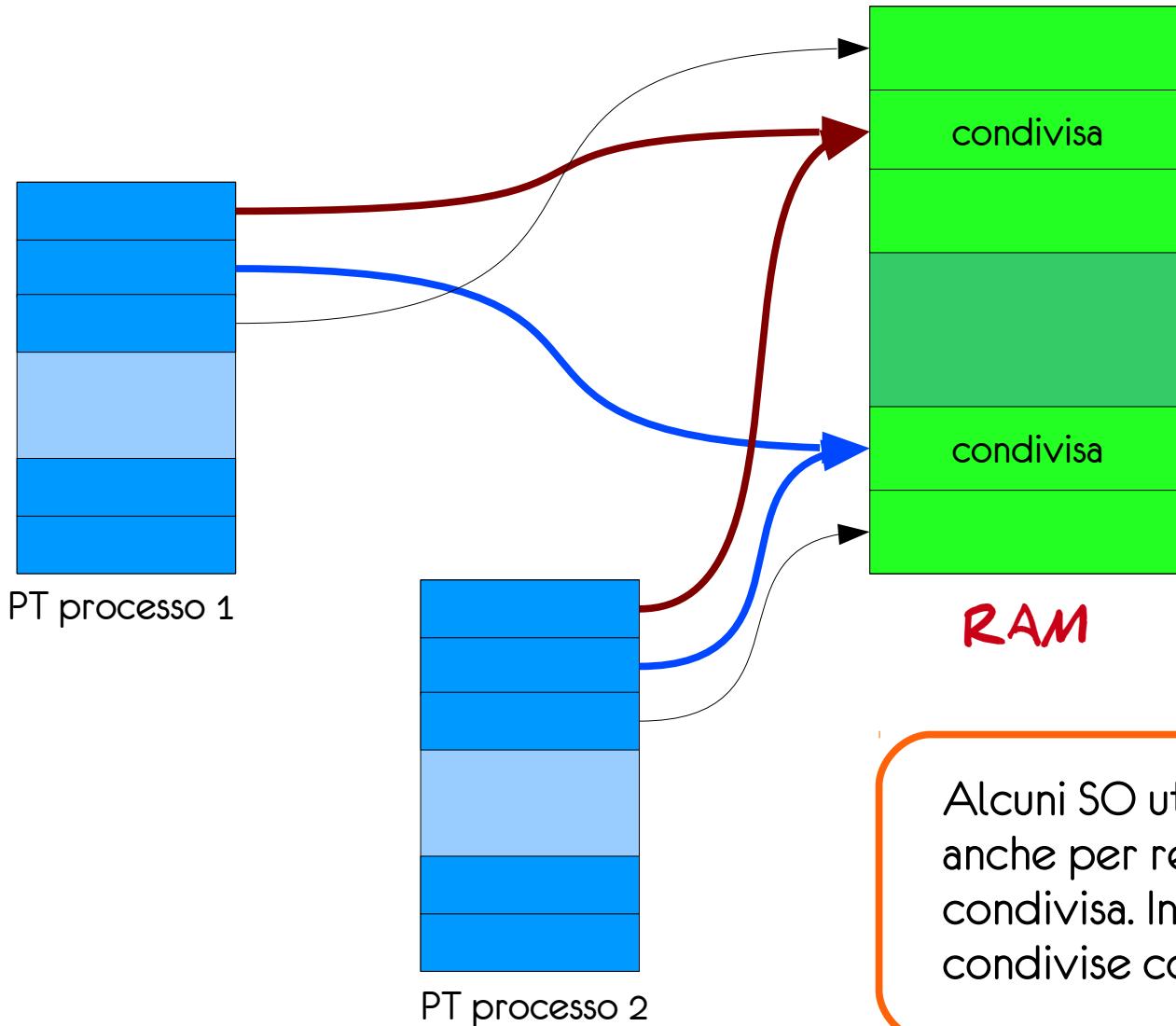
Condivisione delle pagine

- Spesso nei sistemi in time-sharing diversi utenti usano lo **stesso programma contemporaneamente**
- In un contesto di multi-programmazione in senso lato, succede spesso che più processi utilizzino la **stessa libreria contemporaneamente**

Condivisione delle pagine

- Supponiamo che un programma di questo tipo (es. compilatore) occupi 200KB, che 4 utenti lo stiano usando, che i loro processi abbiano 1 pagina di dati e che le pagine siano grandi 50KB: complessivamente saranno occupate 20 pagine di RAM: 1000KB
- se fosse possibile far condividere il codice sarebbero invece sufficienti 400KB di RAM (50×4 per i dati + 200 per il codice in copia unica)
- NB: solo le **pagine di codice** possono essere condivise, pagine accessibili in sola lettura. Si parla di **codice rientrante**

Condivisione delle pagine



Alcuni SO utilizzano questo sistema anche per realizzare la memoria condivisa. In questo caso le pagine condivise contengono dati

Struttura della tabella delle pagine

capitolo 8 del libro (VII ed.), da 8.5

Elenco

- paginazione multi-livello:
 - struttura gerarchica ad albero
- hash table:
 - spesso usata per architetture oltre i 32 bit
- tabella delle pagine invertita:
 - approccio diverso in cui si ha una sola tabella delle pagine globale anziché una per ciascun processo

Paginazione multi-livello

- La dimensione della tabella delle pagine dipende dalla **dimensione dello spazio degli indirizzi logici**
- Consideriamo:
 - indirizzi a 32 bit (permettono di fare riferimento a 2^{32} parole di memoria),
 - con pagine grandi 4KB (cioè 2^{12})
- in quante pagine verrà suddiviso lo spazio degli indirizzi logici?

Paginazione multi-livello

- In quante pagine verrà suddiviso lo spazio degli indirizzi logici?

$$2^{32} / 2^{12} = 2^{20}$$

- cioè la tabella delle pagine potrà avere fino a 2^{20} entry (precisamente 1,048,576)

Paginazione multi-livello

- In quante pagine verrà suddiviso lo spazio degli indirizzi logici?

$$2^{32} / 2^{12} = 2^{20}$$

- cioè la tabella delle pagine potrà avere fino a 2^{20} entry (precisamente 1,048,576)
- se ogni entry occupa 4byte, una tabella delle pagine potrà occuperà **4MB !!!**

Paginazione multi-livello

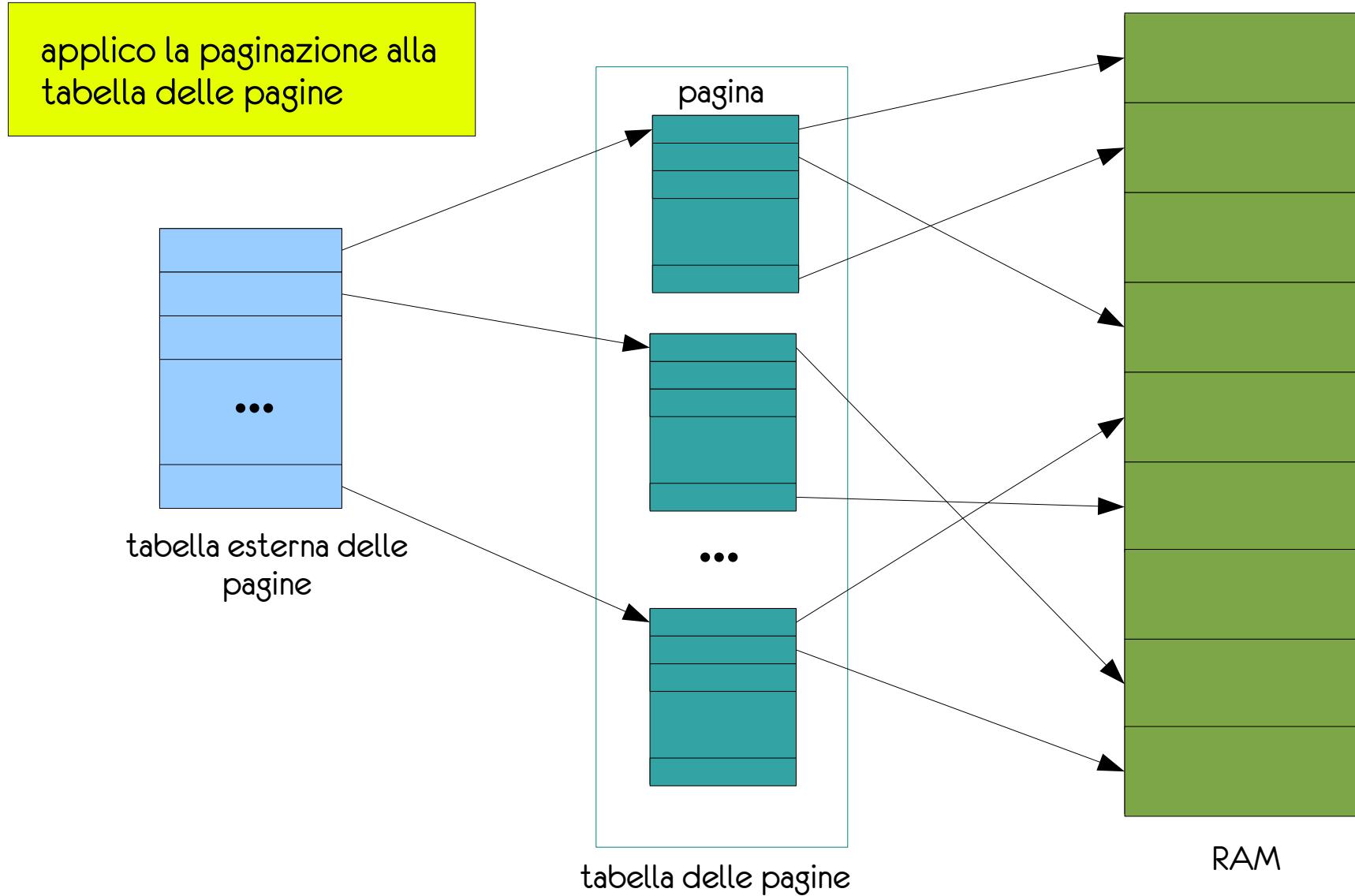
- **Osservazione:**

nella realtà nessun processo usa l'intero spazio degli indirizzi, ne usa molto meno
la maggior parte della tabella sarebbe inutilizzata, tuttavia i processi possono avere **tabelle delle pagine molto pesanti**

Paginazione multi-livello

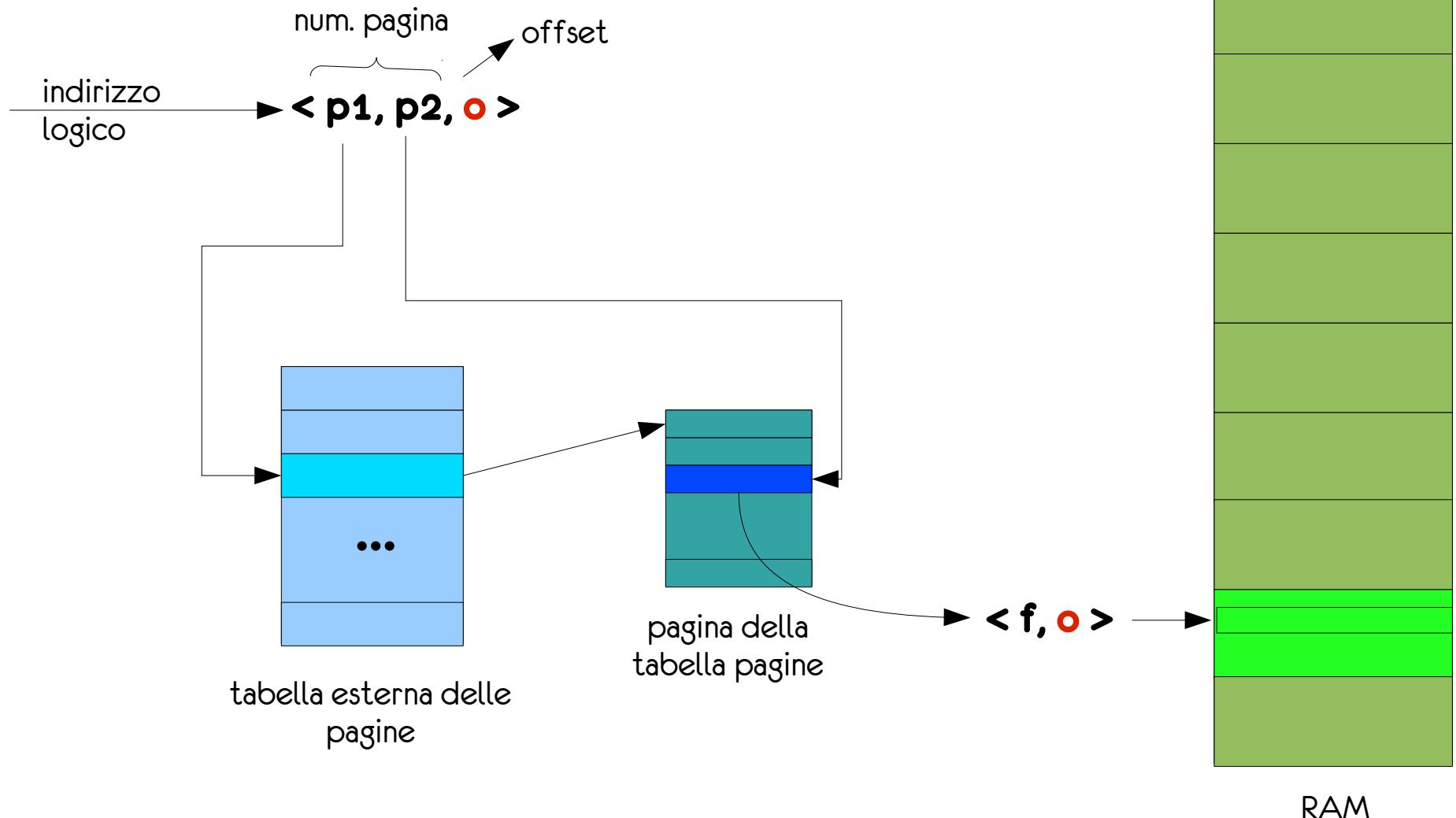
- **Conclusione:** l'allocazione della tabella delle pagine non può essere contigua ma va suddivisa in tante parti organizzate in una struttura
- **Nella paginazione multi-livello si usa un albero**

Paginazione a due livelli



Indirizzi logici

p1	p2	o
10 bit	10 bit	12 bit



Esempi e commenti

- Architettura SPARC (di Sun):
paginazione a 3 livelli
- Motorola 68030 a 32 bit:
paginazione a 4 livelli

Esempi e commenti

- La paginazione multilivello non è appropriata per architetture a 64 bit: la tabella esterna risulterebbe troppo grande
- Si dovrebbero aggiungere troppi livelli ulteriori di paginazione (UltraSPARC ne richiederebbe 7!) e ciò renderebbe **troppo costoso l'accesso** alla RAM qualora si verificasse un TLB miss
- Diverse architetture a 64 bit (es. Intel, HP ma anche Apple PowerPC) ad alte prestazioni usano tabelle delle pagine inverse (vedremo fra poco) **unitamente** a tabelle hash

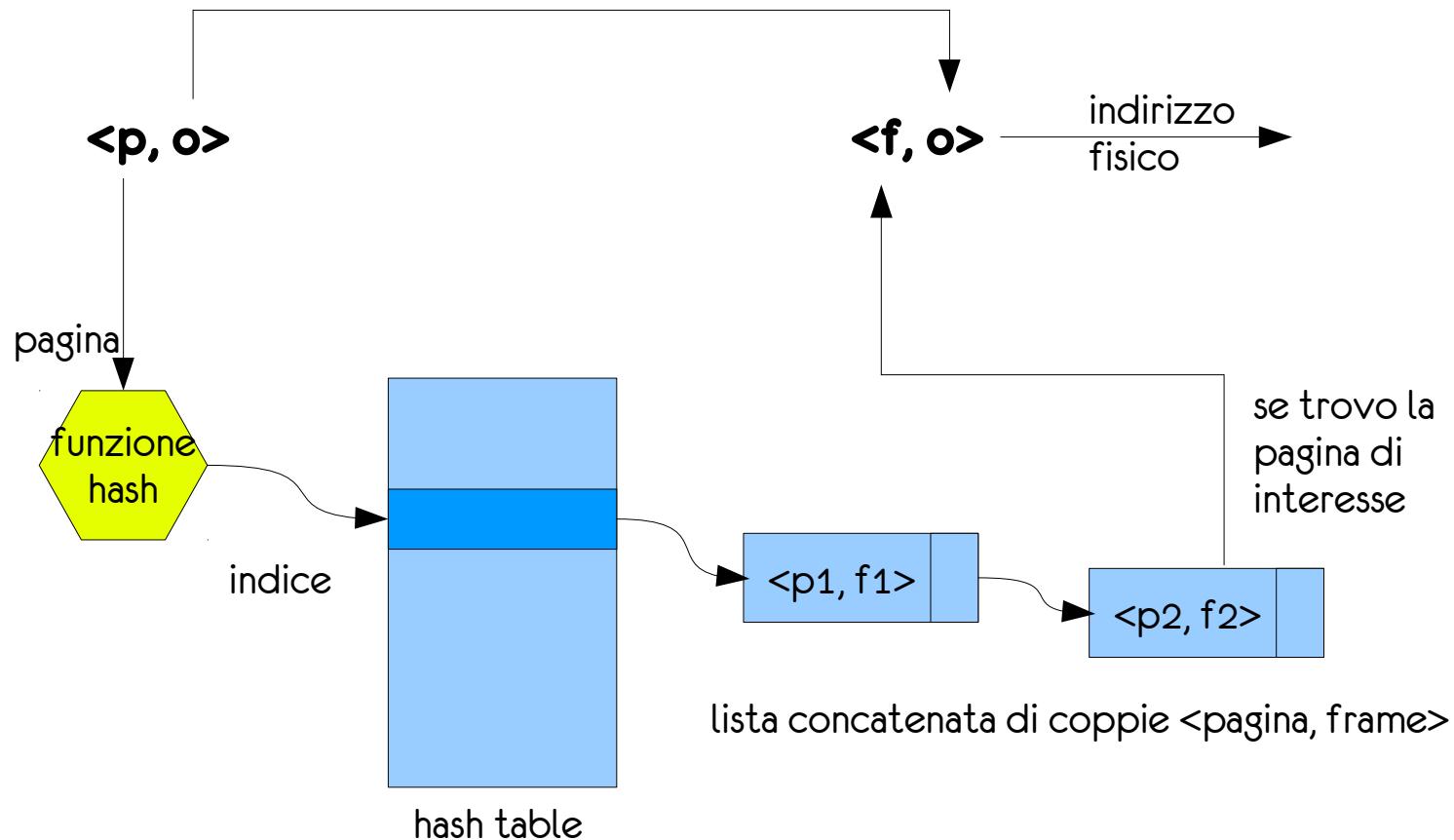
Esempi e commenti

- Architettura SPARC (di Sun):
paginazione a 3 livelli
- Motorola 68030 a 32 bit:
paginazione a 4 livelli

Esempi e commenti

- La paginazione multilivello non è appropriata per architetture a 64 bit: la tabella esterna risulterebbe troppo grande
- Si dovrebbero aggiungere troppi livelli ulteriori di paginazione (UltraSPARC ne richiederebbe 7!) e ciò renderebbe **troppo costoso l'accesso** alla RAM qualora si verificasse un TLB miss
- Diverse architetture a 64 bit (es. Intel, HP ma anche Apple PowerPC) ad alte prestazioni usano tabelle delle pagine inverse (vedremo fra poco) **unitamente** a tabelle hash

Hash table: accenno



Vantaggi e svantaggi delle tabelle delle pagine

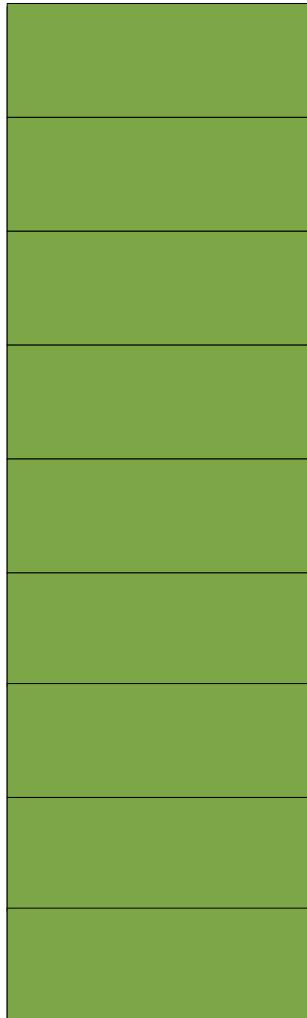
- **Vantaggi**

- rappresentazione naturale se si adotta un **approccio processo-centrico**
- la tabella delle pagine è ordinata in modo tale che sia semplice per il SO identificare il punto in cui si trova l'indirizzo del frame di interesse

Vantaggi e svantaggi delle tabelle delle pagine

- **Svantaggi**
 - una tabella delle pagine può essere grandissima (un elemento per ogni pagina del processo), contenere milioni di elementi e occupare troppo spazio di quella RAM che dovrebbe contribuire a gestire
- **Esistono approcci alternativi?**
- **invertiamo il fuoco e ricominciamo il discorso a partire dalla RAM anziché dai processi**

Tabella invertita



RAM

La RAM è suddivisa in frame, ogni frame può essere libero oppure contenere la pagina di un processo

Posso quindi pensare di mantenere una sola tabella con una entry di questo tipo per ogni frame:

< pid, p >

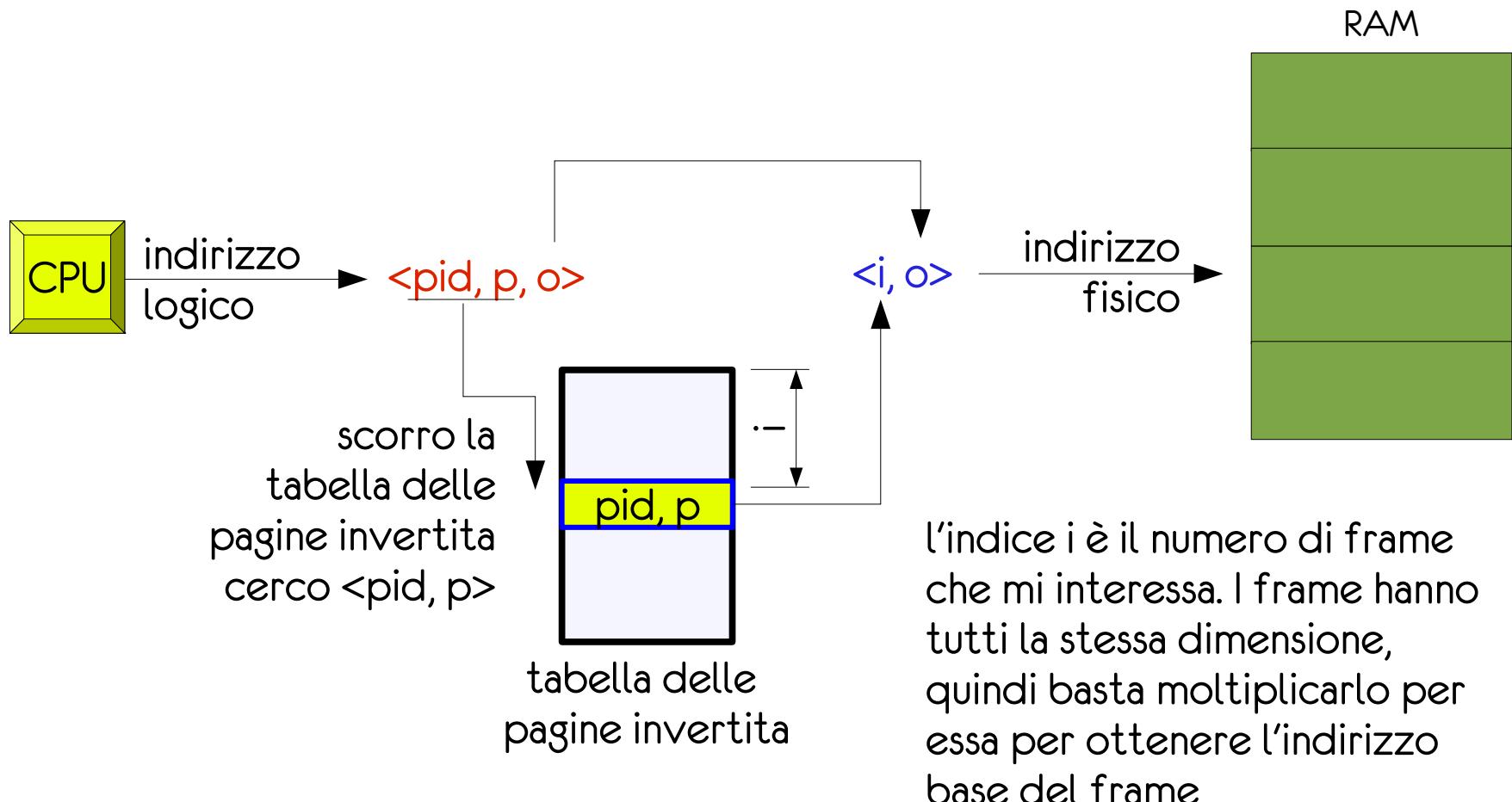
pid = processo, p = pagina

Gli indirizzi logici saranno quindi triple di questo tipo:

< pid, p, o >

pid = processo, p = pagina, o = offset

Struttura



Vantaggi e svantaggi

- **Vantaggi**

- rappresentazione più compatta
- richiede meno spazio in memoria

- **Svantaggi**

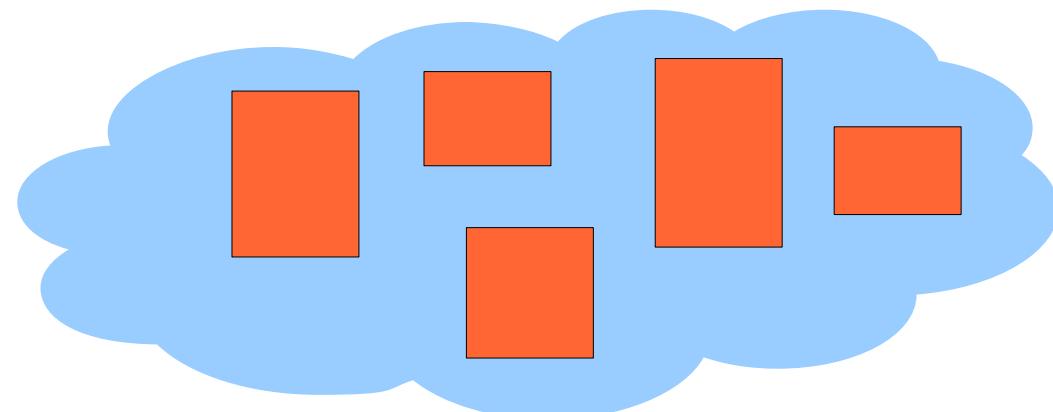
- tempi d'accesso aggravati dalla ricerca in tabella
- per ovviare a questo limite talvolta la tabella delle pagine invertita è implementata come una hash table talvolta ci si appoggia a un TLB

segmentazione

capitolo 8 del libro (VII ed.), da 8.6

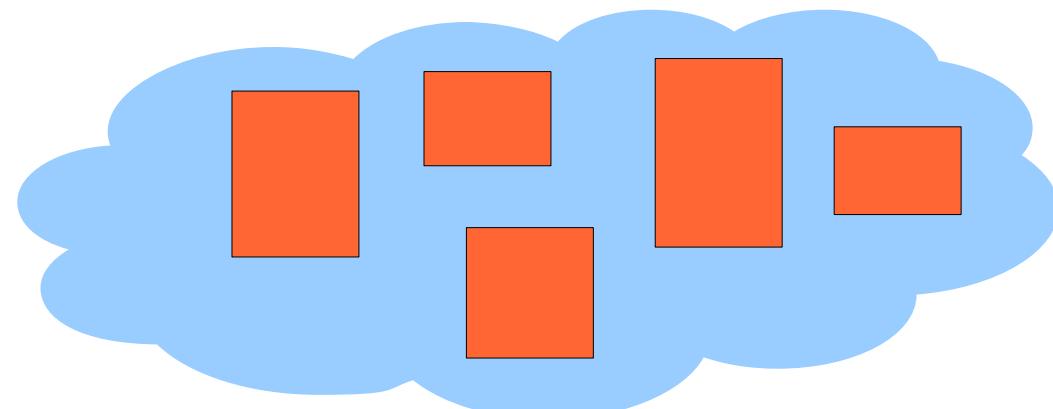
Introduzione

- La paginazione struttura la memoria in un insieme di elementi, il cui contenuto può essere indifferentemente costituito da codice e/o dati
- Questa organizzazione è molto diversa dal modo in cui i *programmatori* vedono i programmi, cioè come strutturate in parti con un preciso *valore funzionale* (il codice, le librerie, lo stack, l'heap)



Introduzione

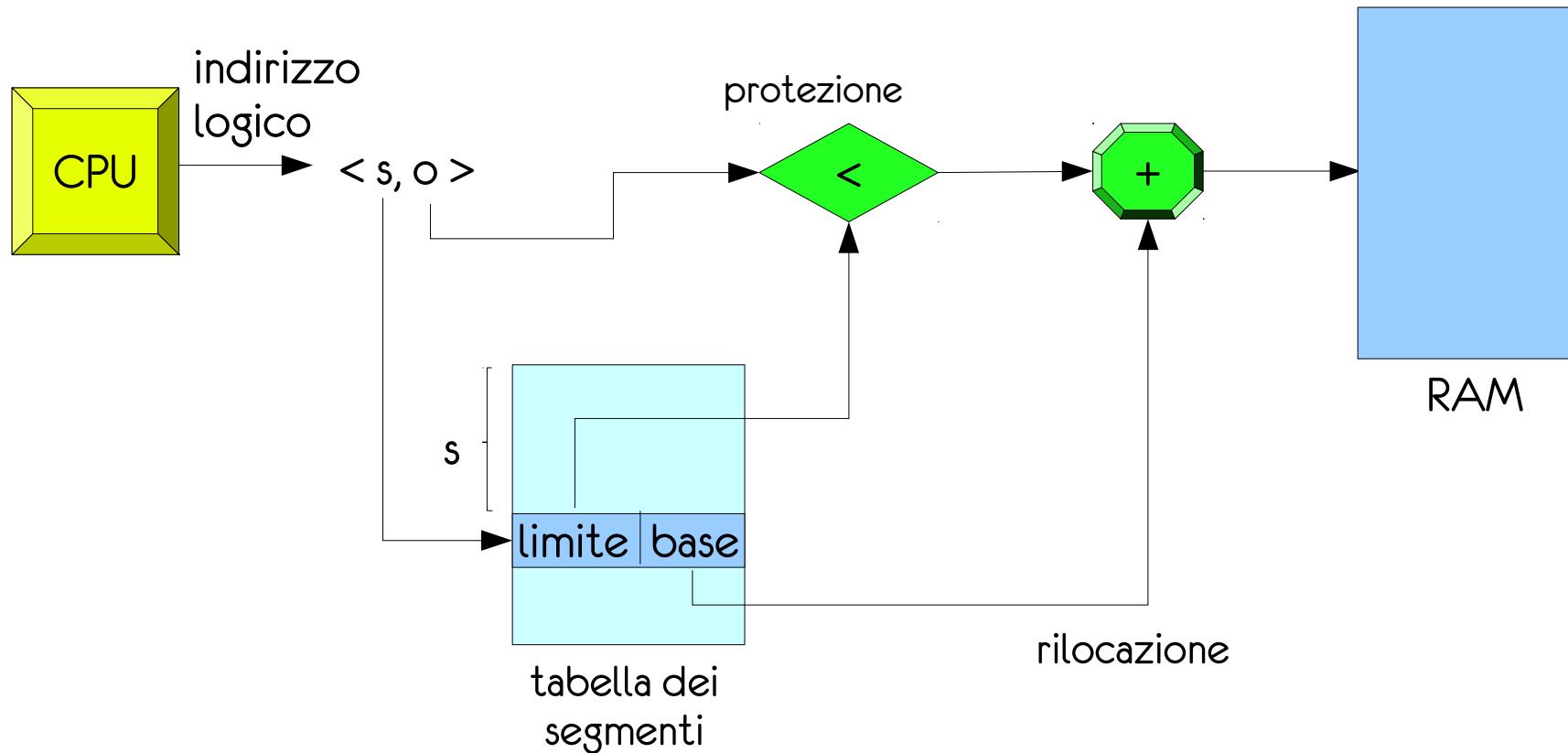
- La segmentazione modella la memoria secondo uno schema più vicino al modo in cui il programmatore vede il programma
- Ogni processo ha una porzione di RAM organizzata come un insieme di segmenti di memoria, di dimensione variabile



Segmenti

- Per adottare questo approccio è necessario che il programma risulti già organizzato in segmenti. Tale strutturazione viene effettuata dal compilatore
- Es. compilatore C può creare questi segmenti per un programma:
 - codice
 - vrb globali
 - stack per ciascun thread
 - heap
 - libreria standard del C
 - le altre librerie avranno assegnati segmenti in un tempo successivo
- Indirizzo logico: < **id_di_segmento, offset** >

Architettura



s = indice della entry relativa al segmento

ogni segmento è caratterizzato da due valori: **indirizzo di base** e **indirizzo limite** perché la loro dimensione è varia

Commenti

- La segmentazione è spesso più efficiente della paginazione
- I segmenti possono avere dimensione estremamente varia
- Quando lo scheduler a medio o lungo termine carica un processo, deve trovare uno spazio adeguato alle dimensioni dei suoi segmenti.
- I problemi sono simili a quelli dello schema a partizioni multiple, anche se la segmentazione è un meccanismo molto più flessibile
 - occorre gestire la memoria libera in modo dinamico
 - si ha frammentazione esterna
 - possibile attuare tecniche di compattazione per ridurre la frammentazione

paginazione + segmentazione

- Alcune architetture sfruttano una tecnica che si basa sull'unione di paginazione e segmentazione
- Esempio: Pentium Intel**



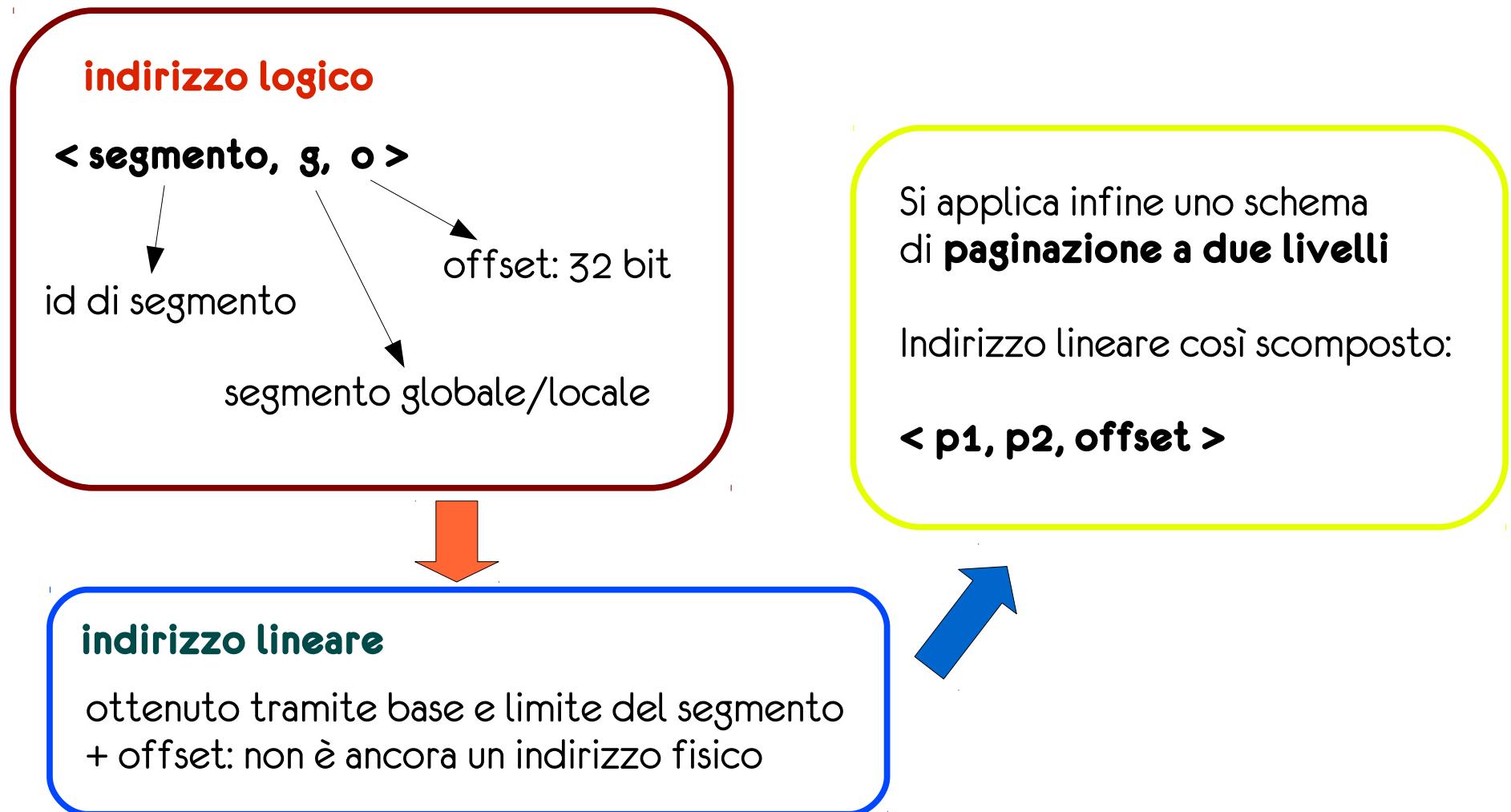
Ogni segmento è strutturato in pagine

dimensione max segmento 4GB
numero max di segmenti per processo 16K

Il processore ha 6 registri di segmento che permettono a un processo di fare riferimento a 6 segmenti contemporaneamente

8K riservati ∈ tabella locale dei descrittori
8K condivisi ∈ tabella globale dei descrittori

Intel Pentium



Conclusioni

- Esistono diversi modelli per la gestione della RAM
- La scelta dipende fortemente dall'HW a disposizione
- Ogni HW è progettato in modo tale da supportare uno (o più) modelli specifici
- Nel valutare un modello/HW occorre tenere presente:
 - se consente la rilocazione
 - se consente lo swapping
 - se consente la condivisione
 - se attua meccanismi di protezione

supporti alla
multi-programmazione

Esercizi

- esercizi sull'allocazione contigua
- esercizi sulla paginazione

Allocazione contigua

- Dati i seguenti processi (da caricare in RAM nell'ordine con cui sono indicati) e le rispettive occupazioni di memoria, e date i seguenti buchi di memoria, dire quali verranno caricati in quali buchi, quali non possono essere caricati al momento e quale saranno i buchi liberi di memoria rimanenti alla fine. Considerare il caso in cui si fa uso della strategia best-fit, quello in cui si usa worst-fit e quello in cui si usa first-fit

P1: 11K buchi: 8K -> 4K -> 20K -> 18K -> 7K -> 10K -> 12K -> 15K

P2: 20K

P3: 9K

Allocazione contigua

- best-fit
 - buchi: 8K -> 4K -> 20K -> 18K -> 7K -> 10K -> **12K** -> 15K
 - P1: 11K
 - P2: 20K
 - P3: 9K
- buchi: 8K -> 4K -> **20K** -> 18K -> 7K -> 10K -> **1K** -> 15K
- P2: 20K
- P3: 9K
- buchi: 8K -> 4K -> 18K -> 7K -> **10K** -> **1K** -> 15K
- P3: 9K
- buchi: 8K -> 4K -> 18K -> 7K -> **1K** -> **1K** -> 15K

Allocazione contigua

buchi: 8K -> 4K -> **20K** -> 18K -> 7K -> 10K -> 12K -> 15K

- first-fit

P1: 11K

P2: 20K

P3: 9K

buchi: 8K -> 4K -> **9K** -> 18K -> 7K -> 10K -> 12K -> 15K

P2: 20K: non si può più allocare

P3: 9K

buchi: 8K -> 4K -> **9K** -> 18K -> 7K -> 10K -> 12K -> 15K

P3: 9K

buchi: 8K -> 4K -> 18K -> 7K -> 10K -> 12K -> 15K

Allocazione contigua

buchi: 8K -> 4K -> **20K** -> 18K -> 7K -> 10K -> 12K -> 15K

- worst-fit

P1: 11K

P2: 20K

P3: 9K

buchi: 8K -> 4K -> **9K** -> 18K -> 7K -> 10K -> 12K -> 15K

P2: 20K: non si può più allocare

P3: 9K

buchi: 8K -> 4K -> 9K -> **18K** -> 7K -> 10K -> 12K -> 15K

P3: 9K

buchi: 8K -> 4K -> 9K -> **9K** -> 7K -> 10K -> 12K -> 15K

Paginazione

- Sia data un'architettura che supporta la paginazione, in cui le pagine hanno dimensione pari a 2^{10} byte. Quante pagine saranno necessarie per gestire una RAM di 1GB? Quanti bit saranno necessari per rappresentare il numero di pagina? quanti per l'offset? Di quanti bit sarà costituito complessivamente un indirizzo?

Paginazione

- Sia data un'architettura che supporta la paginazione, in cui le pagine hanno dimensione pari a 2^{10} byte. Quante pagine saranno necessarie per gestire una RAM di 1GB?
- $1\text{GB} = 2^{30}$ byte
- sono necessarie $2^{30} / 2^{10}$ pagine = 2^{20}
- Sono necessari 20 bit per rappresentare il numero di pagina
- Sono necessari 10 bit per rappresentare l'offset
- complessivamente un indirizzo sarà costituito da 30 bit

Paginazione

- Sia data un'architettura che supporta la paginazione, in cui le pagine hanno dimensione pari a 2^{12} byte. Quante pagine saranno necessarie per rappresentare un processo della dimensione di 2MB?

Paginazione

- Sia data un'architettura che supporta la paginazione, in cui le pagine hanno dimensione pari a 2^{12} byte. Quante pagine saranno necessarie per rappresentare un processo della dimensione di 2MB?
- $2\text{MB} = 2 * 2^{20} \text{ byte} = 2^{21} \text{ byte}$
- $2^{21} / 2^{12} = 2^9 \text{ pagine}$

memoria virtuale

capitolo 9 del libro (VII ed.), 12.6

Introduzione

- Nella descrizione delle tecniche di gestione della RAM abbiamo glissato sul problema del **caricamento parziale dei processi**, lasciando intendere (non troppo esplicitamente) che i processi fossero sempre caricati interamente
- è utile rilasciare questo vincolo?



Introduzione

- Nella descrizione delle tecniche di gestione della RAM abbiamo glissato sul problema del **caricamento parziale dei processi**, lasciando intendere (non troppo esplicitamente) che i processi fossero sempre caricati interamente
- è utile rilasciare questo vincolo?

Introduzione

- Nella descrizione delle tecniche di gestione della RAM abbiamo glissato sul problema del **caricamento parziale dei processi**, lasciando intendere (non troppo esplicitamente) che i processi fossero sempre caricati interamente
- è utile rilasciare questo vincolo?
- **Sì:**
 - Consente di creare programmi che (da soli o complessivamente) occupano più spazio di quanto disponibile in RAM
 - Si liberano i programmatori dai “vincoli di memoria”: posso portare un grosso programma su un computer con meno memoria e vederlo comunque funzionare
 - Si migliora l'uso della RAM: (1) molte procedure vengono usate in circostanze rare, perché caricarle sempre? (2) spesso array e matrici sono sovradimensionati, perché non aggiungere spazio solo se serve davvero?
 - Riducendo la RAM necessaria a ciascun processo, posso eseguire molti più programmi contemporaneamente! Maggiore multiprogrammazione
 - meno tempo per fare swap
 - meno I/O per caricare i processi

Percorso



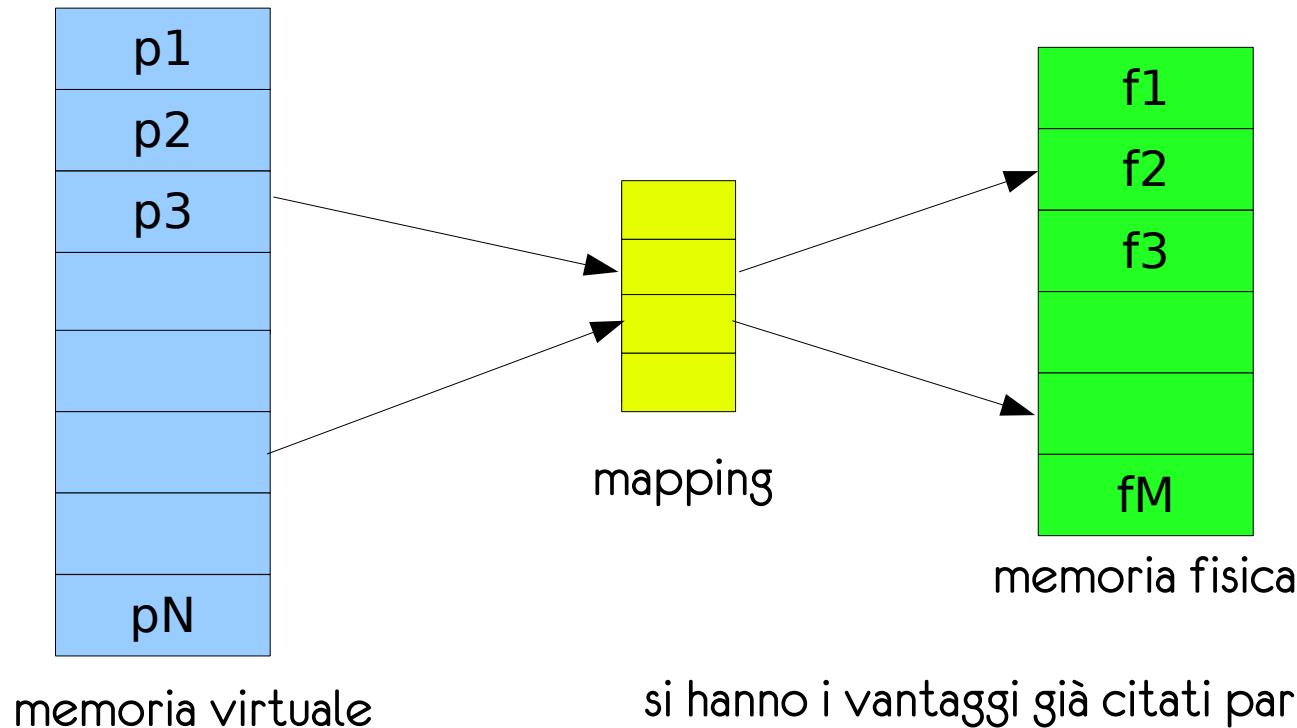
Processo caricato in
RAM in modo
contiguo e totale

**Si rilascia il vincolo
di contiguità:**
- paginazione
- segmentazione

**Si rilascia il vincolo
di totalità:**
- memoria virtuale

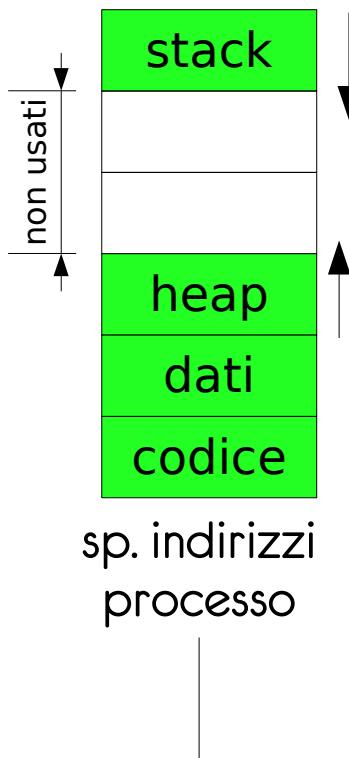
Memoria virtuale

- Il concetto di “memoria virtuale” nasce dalla separazione della memoria logica (la memoria come percepita dall'utente) dalla memoria fisica, a disposizione



si hanno i vantaggi già citati parlando della paginazione, fra le altre cose la condivisione di pagine (librerie / memoria condivisa)

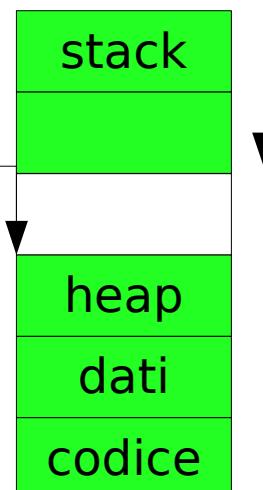
Spazio degli indirizzi di un processo



Lo spazio degli indirizzi di un processo è predefinito e, in genere, è molto più grande dello spazio realmente occupato dal processo

Con l'esecuzione stack e heap cresceranno l'uno verso l'altro. Per contenere i nuovi dati potranno essere allocati nuovi frame.

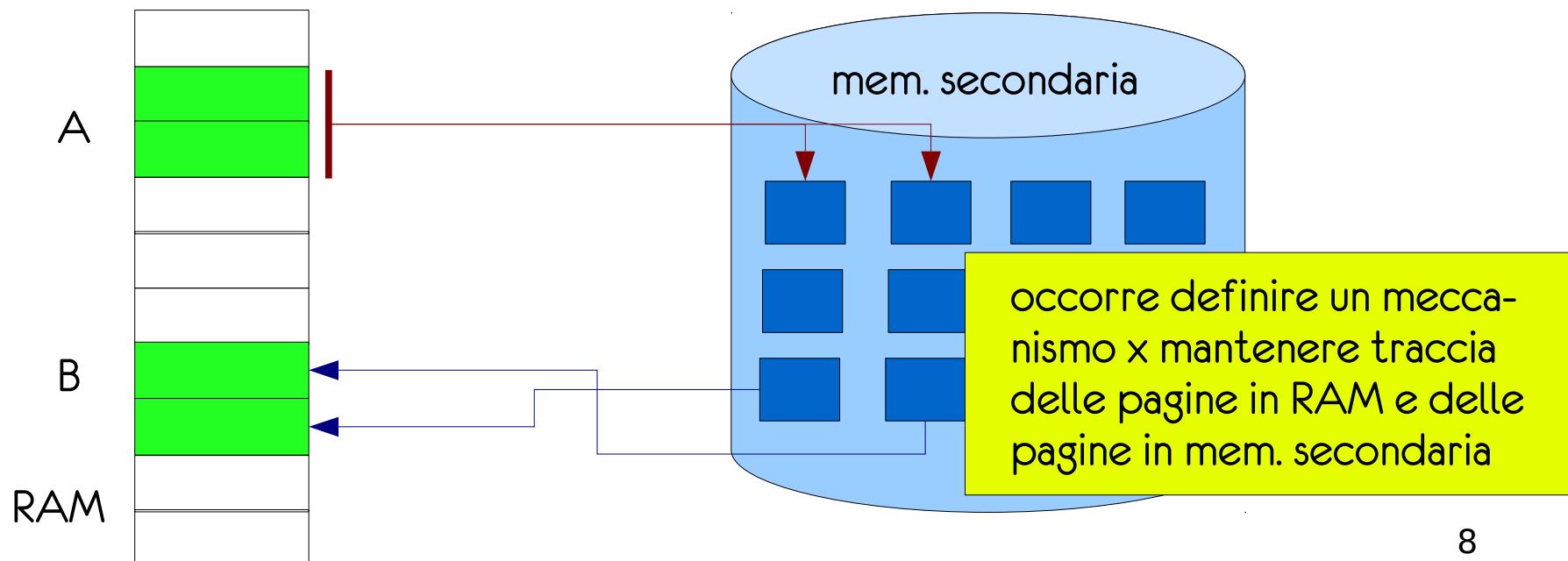
Questo non cambia lo spazio degli indirizzi virtuali del processo, semplicemente una parte di indirizzi prima non corrispondenti ad alcun indirizzo assoluto saranno ora mappati su parole di memoria effettive



Di qui in avanti pensiamo al processo come a un insieme di pagine

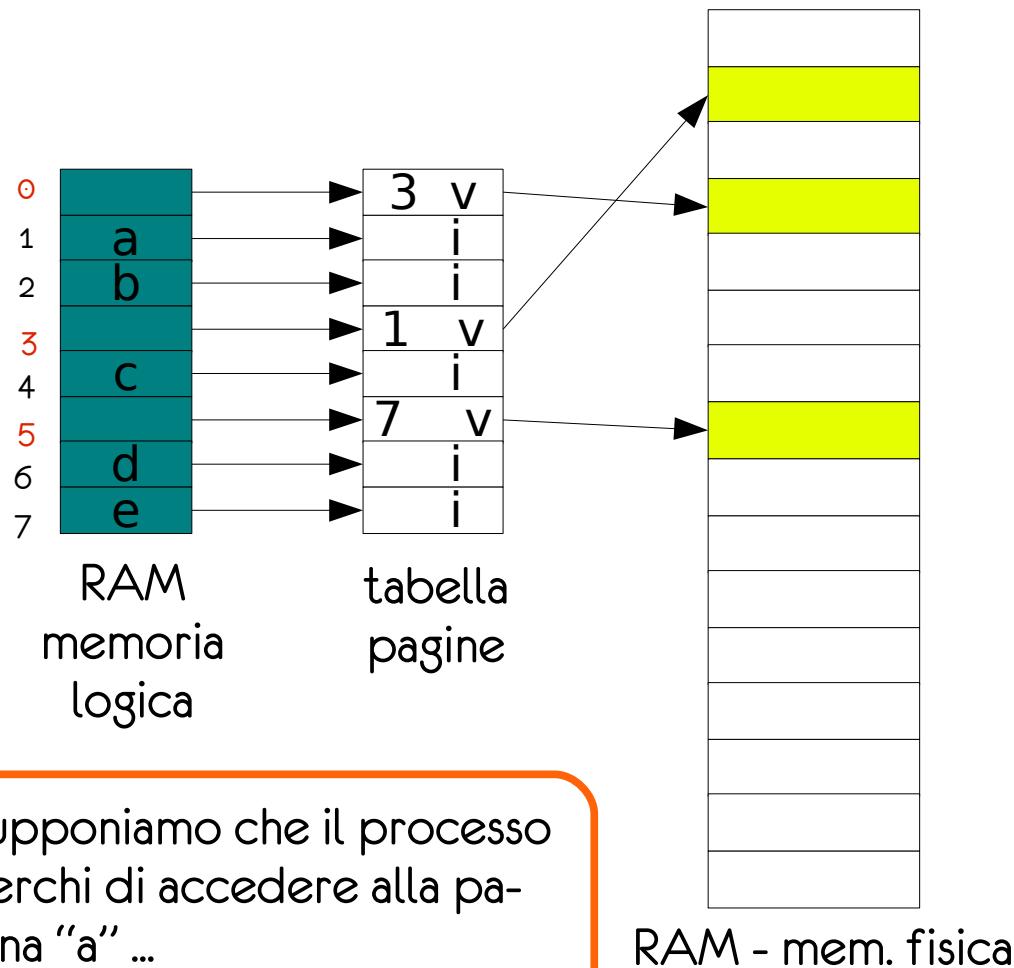
Demand paging - lazy swapping

- Ricorda lo swapping ma riguarda porzioni di processo
- I processi vengono caricati in modo parziale:
quando una pagina diventa utile (perché contiene una parte di codice da eseguire o dei dati da elaborare) la si carica avvicinandola con una pagina precedentemente in RAM
- Questo meccanismo è noto come **lazy swapping**
- La parte del SO che gestisce il caricamento delle pagine è detto **pager**

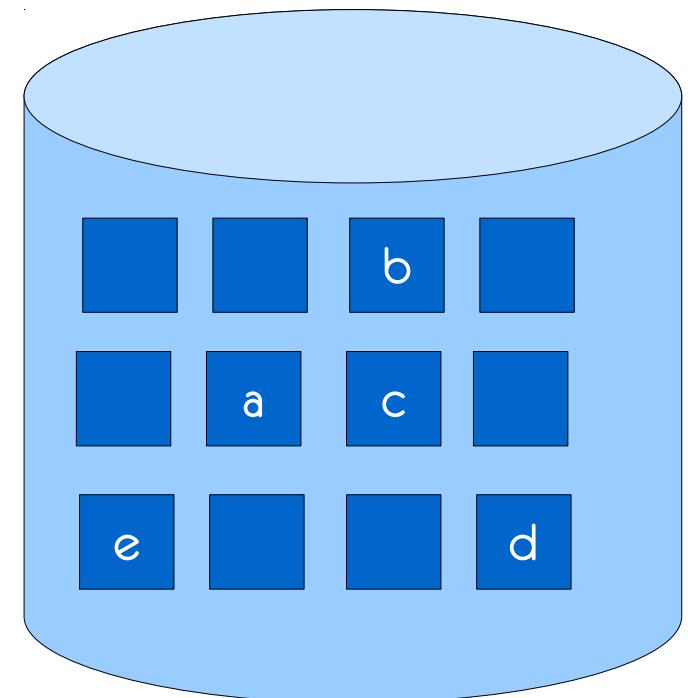


Bit di validità

nella tabella delle pagine di un processo si interpreta il bit di validità in questo modo:
se è falso la pagina o appartiene a un altro spazio degli indirizzi oppure appartiene al
processo ma al momento risiede in memoria secondaria



Supponiamo che il processo
cerchi di accedere alla pa-
gina "a" ...



le pagine invalide sono in
memoria secondaria

Page fault

- Quando un processo cerca di accedere a una pagina che non valida si genera un interrupt (**page fault exception**)
- Il sistema accede a una tabella conservata nel PCB del processo per verificare se si tratta di una pagina del processo, ancora in memoria secondaria (altrimenti si tratta di una pagina invalida):
- **Se sì:**
 - individua un **frame libero**
 - è richiesta un'operazione di **copiatura da disco** della pagina desiderata
 - a lettura completata si **aggiorna la tabella delle pagine**
 - si **riavvia l'operazione interrotta** dall'interrupt e il processo riprende come se nulla fosse
- **Se no** si termina il processo

Page fault: esempio 1

- L'interruzione per page fault può occorrere in momenti diversi dell'esecuzione di un'istruzione
- Consideriamo per es. l'istruzione ADD A B (somma A e B memorizzando il risultato in qualche locazione C non menzionata nell'istruzione):
 - <1> page fault all'atto del **caricamento dell'istruzione**
 - <2> page fault quando si cerca di **accedere a un operando**
 - <3> page fault quando si deve **salvare il risultato** in C
- Cosa significa “riavviare l'istruzione” in questi tre casi?
- In generale l'interruzione del ciclo fetch-decode-execute causa il riavvio del ciclo stesso per l'istruzione interrotta

Page fault: esempio 1

- <1> page fault all'atto del caricamento dell'istruzione:
 - l'istruzione da eseguire non è in RAM: viene caricata la pagina contenente l'istruzione, si carica l'istruzione, la si esegue
- <2> page fault quando si cerca di accedere a un operando:
 - l'istruzione è già stata caricata e decodificata, siamo in fase di esecuzione: si carica la pagina mancante, si ricarica l'istruzione, la si esegue
- <3> page fault quando si deve salvare il risultato in C:
 - l'istruzione è già stata parzialmente eseguita: si carica la pagina mancante, si ricarica l'istruzione, la si riesegue (salvando il risultato)

NB: una stessa istruzione potrebbe causare diversi page fault

Paging on demand

- Anche detta “paginazione a richiesta” (o demand paging), è un meccanismo per cui una pagina viene caricata in RAM SSE è stata richiesta
- (ipotesi estrema) Un processo viene avviato *senza caricare alcuna sua pagina*
- Il caricamento della prima istruzione causa un page fault e quindi il caricamento della prima pagina, ecc.
- In generale, si tratta di una tecnica costosa? Ogni istruzione può causare diversi page fault, il tempo di esecuzione potrebbe moltiplicarsi a dismisura ...
- È una tecnica che richiede particolari supporti HW?



Paging on demand

- Le uniche strutture HW di supporto richieste sono quelle già viste per realizzare la **paginazione** e lo **swapping** (cap. 8)
- Riguardo l'esecuzione:
 - in generale vale il **principio di località dei riferimenti** (riprenderemo il concetto parlando di paginazione degenere)
 - proviamo però a fare un calcolo x valutare il peso della paginazione su richiesta sul **tempo di accesso effettivo** alle pagine:

$$\text{tempo di accesso effettivo} = (1-p) * ma + p * tgpf$$

- dove p = probabilità che si verifichi un page fault, $p \in [0, 1]$
- ma = tempo medio di accesso alla RAM $\in [10, 200]$ nsec
- $tgpf$ = tempo di gestione di un page fault, comprende il tempo di lettura e caricamento della pagina da memoria secondaria $\in 8$ msec

Gestione del page fault

- si genera un'interruzione
- salvataggio dei registri e dello stato del processo
- verifica dell'interruzione: in questo caso si determina che si tratta di page fault
- controllo della correttezza del riferimento (pagina invalida perché?) e individuazione della locazione occupata dalla pagina mancante su disco
- lettura e copiatura della pagina da memoria secondaria in RAM (operazione di I/O)
- durante l'attesa per il completamento di questa operazione, allocazione della CPU a un altro processo
- interrupt che segnala il completamento dell'operazione di caricamento della pagina
- aggiornamento della tabella delle pagine
- quando lo scheduling riavvia il processo sospeso, ripristino dello stato
- riesecuzione dell'istruzione interrotta

Gestione del page fault

- si genera un'eccezione
- salvare lo stato del programma
- verificare se la pagina è valida (è stata cancellata? è stata modificata da un altro processo?)
- controllare la tabella di traduzione (valida perché?) e individuazione della posizione occupata dalla pagina mancante su disco

Ordini di Grandezza

1 secondo	10^0	unità di misura
1 millisecondo	10^{-3}	
1 microsecondo	10^{-6}	
1 nanosecondo	10^{-9}	

- lettura e copiatura della pagina da memoria secondaria in RAM (operazione di I/O)
- durante l'attesa di I/O possiamo eseguire altre operazioni (es. calcolo della CPU a bassa priorità)
- più in breve possiamo riassumere queste operazioni in un solo interrupt che viene invocato quando la pagina è pronta
- interrupt che viene invocato quando la pagina è pronta
 - <1> servizio di interruzione per page fault
 - <2> lettura della pagina
 - <3> riavvio del processo
- quando lo scheduling riavvia il processo sospeso, il ripristino dello stato richiedono da 1 a 100 microsecondi
- riesce a gestire circa 8 millesimi di secondi

richiede circa 8 millesimi di secondi

dominante

Un po' di numeri ...

tempo di accesso effettivo = $(1-p) * ma + p * tgpf$

$$ma = 200 \text{ nsec}$$

$$tgpf = 8 \text{ msec}$$

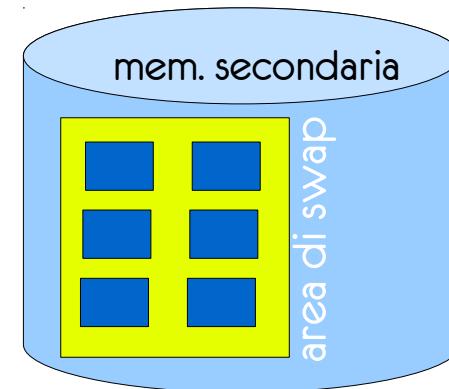
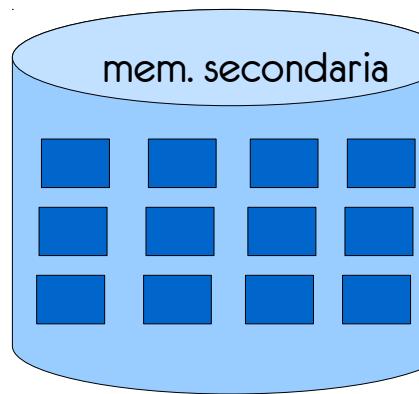
$$\begin{aligned} t. a. e. &= (1-p) * 200 + p * 8.000.000 \\ &= 200 - p * 200 + p * 8.000.000 = \\ &= 200 + p * 7.999.800 \end{aligned}$$

supponiamo di avere un page fault ogni 1000 accessi: $p = 0.001$

$$\begin{aligned} t. a. e. &= 200 + 0.001 * 7.999.800 = 200 + 7.999,8 \approx 8.000 \text{ nsec} \\ &\quad (8 \text{ microsec}) \end{aligned}$$

Area di swap

- Abbiamo detto che le pagine di un processo che non sono in RAM sono contenute in **memoria secondaria** (backing store)
- In particolare, tali pagine sono conservate in una porzione speciale della memoria secondaria, detta **area di swap**



- L'area di swap è vista come un'estensione della RAM, anche se i tempi di accesso sono molto maggiori

Area di swap

- La gestione dell'area di swap varia molto da SO a SO sia per quel che riguarda la sua **implementazione** sia per quel che riguarda il suo **dimensionamento** sia per quel che riguarda i suoi **contenuti**
- dimensionamento
 - Linux: suggerisce di allocare il doppio della quantità di RAM a disposizione
 - es. se ho $\frac{1}{2}$ GB di RAM -> riservo 1GB di area di swap
 - Solaris: suggerisce di allocare una quantità pari alla differenza fra la dimensione dello spazio degli indirizzi logici e la dimensione della RAM
 - es. 2^{32} , sp. ind. indirizzi logici, 2^{10} spazio di RAM, $2^{32} - 2^{10}$ dimensione dell'area di swap
- **Consiglio generale:** *melius abundare quam deficere*, se l'area di swap si esaurisce il SO dovrà terminare qualche processo per liberare memoria

Area di swap

- **Implementazione**

- **come file**: l'area di swap è un file speciale del file system
 - **pro**
 - si evita il problema del dimensionamento, un file cresce/diminuisce a seconda delle necessità
 - **contro**
 - la gestione del file system introduce delle sovrastrutture che rallentano ulteriormente l'accesso
- **come partizione a sé, non formattata**
 - si usa un gestore speciale che manipola direttamente pagine e adotta algoritmi ottimizzati per ridurre i tempi di accesso
 - **pro**: maggiore velocità
 - **contro**: per ridimensionarla occorre ripartizionare il disco

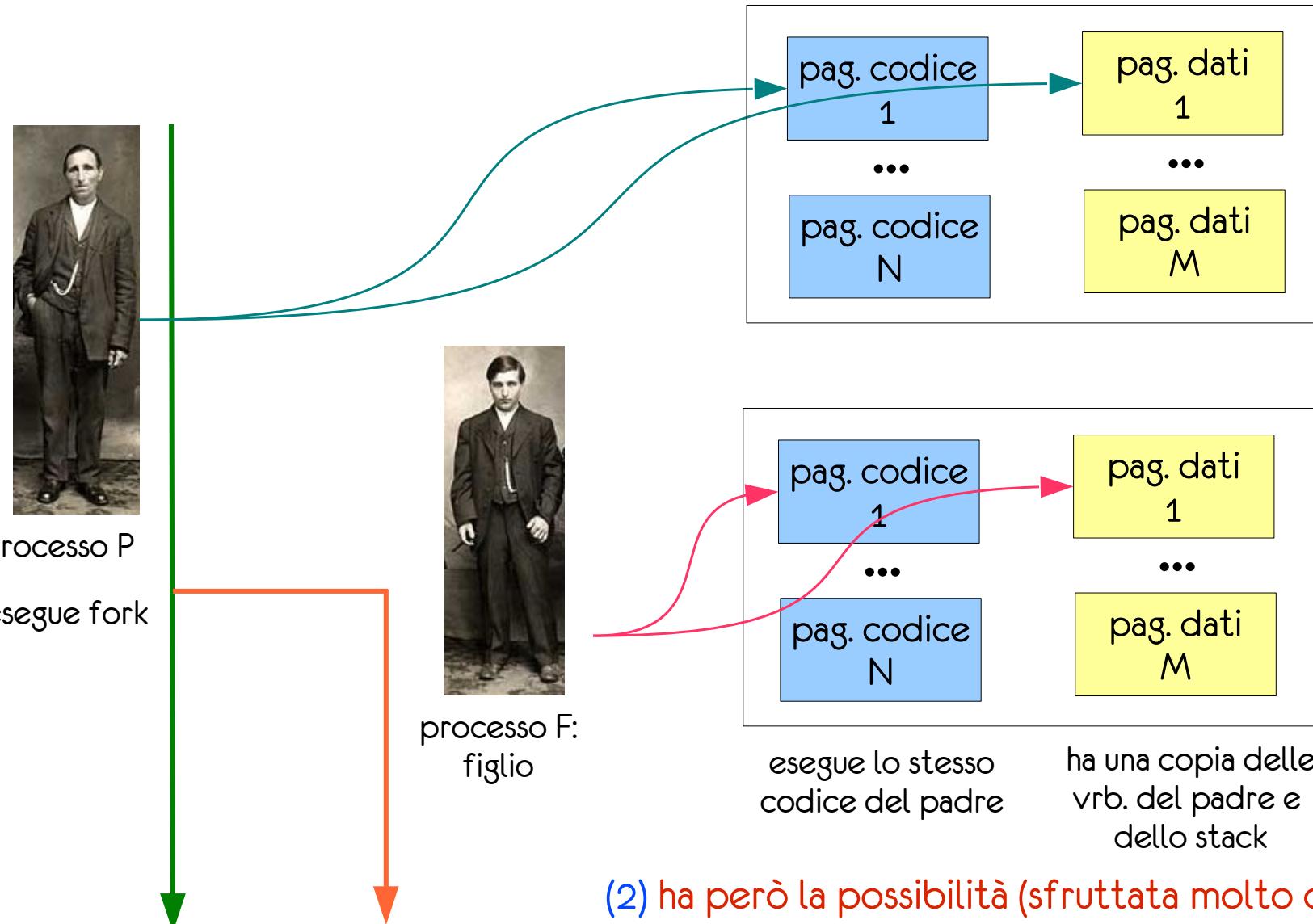
Area di swap

- Contenuti
- SO di qualche anno fa mantenevano nell'area di swap pagine di codice e pagine di dati ...
- ... però dato che le pagine di codice **non sono modificate** dall'esecuzione dei programmi e i tempi di accesso all'area di swap non sono poi di molto inferiori a quelli di accesso al resto del disco ...
- ... oggi si preferisce **mantenere nell'area di swap per lo più pagine di dati** (stack/heap dei processi) prelevando le pagine di codice direttamente dal file system
- Esempi di SO che adottano questa tecnica: **Solaris** e **BSD**

processi padri e figli

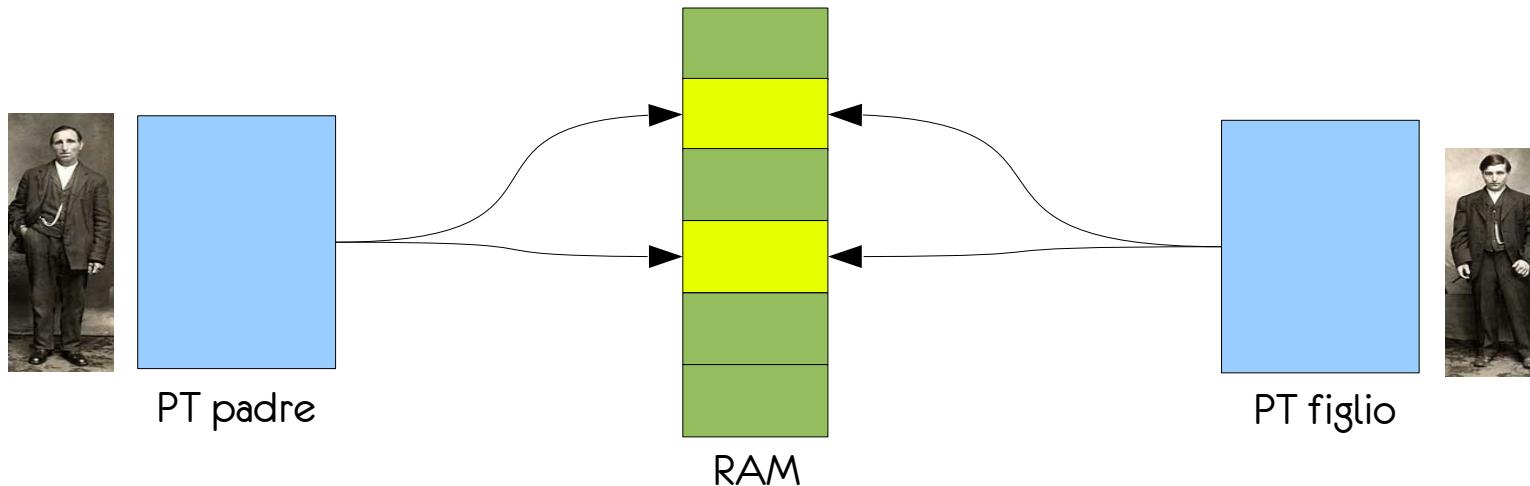
capitolo 9 del libro (VII ed.)

Processi padri e figli

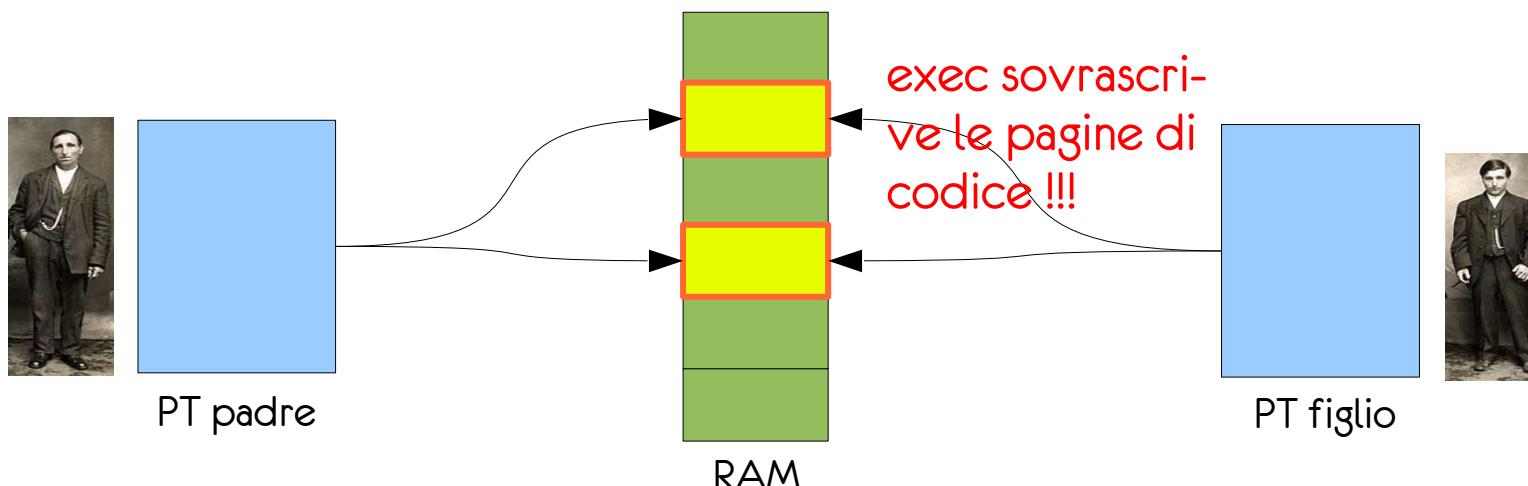


(2) ha però la possibilità (sfruttata molto di frequente) di cambiare il codice da eseguire tramite la system call "exec": l'effetto è una sovrascrittura delle pagine di codice del processo

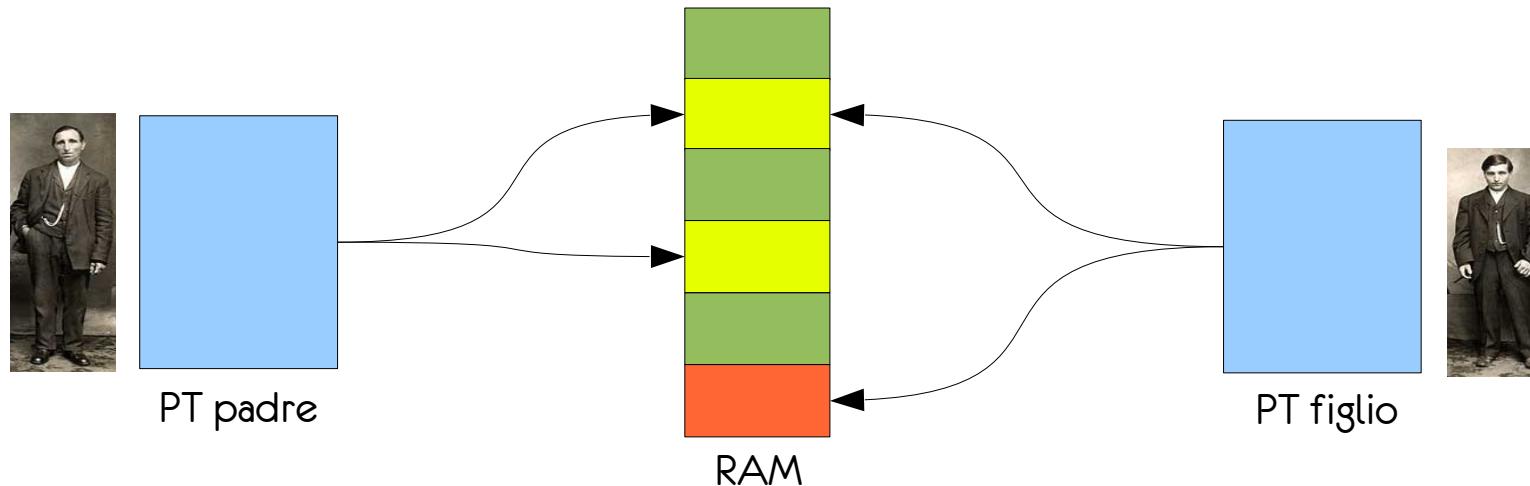
Fork, exec e paginazione



- (1) per ovviare al problema della duplicazione delle pagine di codice del processo padre e del processo figlio, è possibile far condividere tali pagine ai 2 processi



Copiatura su scrittura



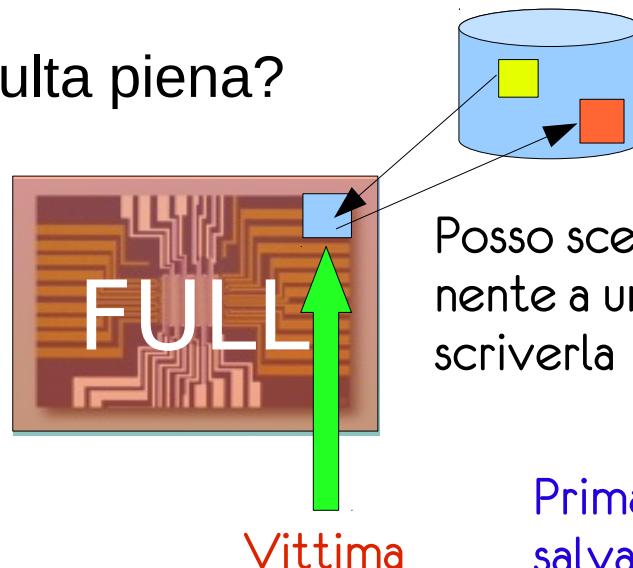
(2) per ovviare al problema della sovrascrittura delle pagine di codice del processo padre si adotta la tecnica di “**copiatura su scrittura**”: quando uno dei due processi esegue una “exec” si allocano delle nuove pagine per il processo chiamante e si carica il nuovo codice in esse

Le ragioni del nome “copiatura su scrittura” derivano dal caso (più generale) in cui un processo deve **modificare solo in parte** il contenuto di una pagina condivisa. In questo caso la pagina viene prima copiata e poi si consente la modifica della copia.

Es. si possono far condividere ai due processi anche stack e heap, almeno inizialmente. Quando uno dei due cerca di modificare una porzione (es.) di stack si duplica solo la pagina coinvolta

Sostituzione di pagine

- La paginazione come tecnica di realizzazione della memoria virtuale aumenta il livello di multi-programmazione dell'elaboratore
- Abbiamo affrontato il problema del page fault ...
- ... ma non ci siamo ancora posti il problema se, in occorrenza di un page fault, sia sempre possibile individuare un frame libero in cui caricare la pagina richiesta ...
- Cosa fare se la RAM risulta piena?



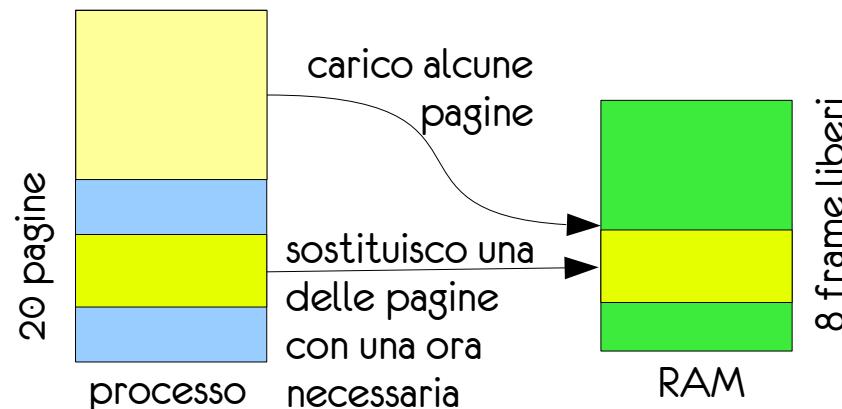
Prima di sovrascrivere occorre salvare la vittima in memoria secondaria

Dirty bit

- Il meccanismo descritto richiede la **copiatura di due pagine**: la vittima sarà copiata in memoria secondaria, la nuova pagina sarà copiata in RAM
- per ridurre l'inefficienza comportata da questa duplice scrittura, si cerca di limitarla ai casi in cui è effettivamente necessaria:
 - la pagina nuova va necessariamente copiata in RAM
 - ma è sempre necessario copiare la vittima in memoria secondaria?
- È necessario solo se la pagina **è stata modificata rispetto a una sua copia già conservata su disco**
- Basta mantenere un bit (**dirty bit**) per ogni pagina: il bit viene settato non appena la pagina è modificata
- **if (! dirty bit) → la pagina non va ricopiata**

Commenti

- Paginazione:
 - memoria logica e memoria fisica completamente separate
- i processi hanno a disposizione uno spazio degli indirizzi più grande di quello fisico offerto dalla RAM
- realizzazione di un **meccanismo dinamico** tramite il quale caricare / sostituire pagine a seconda delle esigenze di esecuzione
- introduzione di meccanismi finalizzati ad aumentare l'efficienza, contrastando in parte gli appesantimenti imposti dalle moltiplicate letture/copiature di pagine
- incremento del livello di multiprogrammazione



Implementazione

- Implementazione la paginazione su richiesta significa sviluppare due algoritmi:
 - **algoritmo di allocazione dei frame:**
 - spartisce M frame liberi fra N processi
 - **algoritmo di sostituzione delle pagine:**
 - seleziona le pagine da sostituire
 - occorre definire il criterio di selezione
 - occorre arricchire l'informazione con i dati necessari per applicare il criterio di selezione
- è importante poter valutare i diversi algoritmi proposti
 - di norma si sceglie l'algoritmo che causa la **frequenza di assenza delle pagine** (page fault rate) minore

Algoritmi di sostituzione

- FIFO
- ottimale
- LRU (least-recently used)
- per approssimazione a LRU
 - seconda chance
 - algo. con bit supplementari di riferimento
- basato su conteggio:
 - least frequently used
 - most frequently used

Implementazione

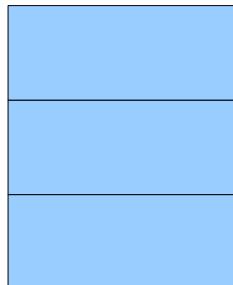
- Implementazione la paginazione su richiesta significa sviluppare due algoritmi:
 - **algoritmo di allocazione dei frame:**
 - spartisce M frame liberi fra N processi
 - **algoritmo di sostituzione delle pagine:**
 - seleziona le pagine da sostituire
 - occorre definire il criterio di selezione
 - occorre arricchire l'informazione con i dati necessari per applicare il criterio di selezione
- è importante poter valutare i diversi algoritmi proposti
 - di norma si sceglie l'algoritmo che causa la **frequenza di assenza delle pagine** (page fault rate) minore

Algoritmi di sostituzione

- FIFO
- ottimale
- LRU (least-recently used)
- per approssimazione a LRU
 - seconda chance
 - algo. con bit supplementari di riferimento
- basato su conteggio:
 - least frequently used
 - most frequently used

Sostituzione FIFO

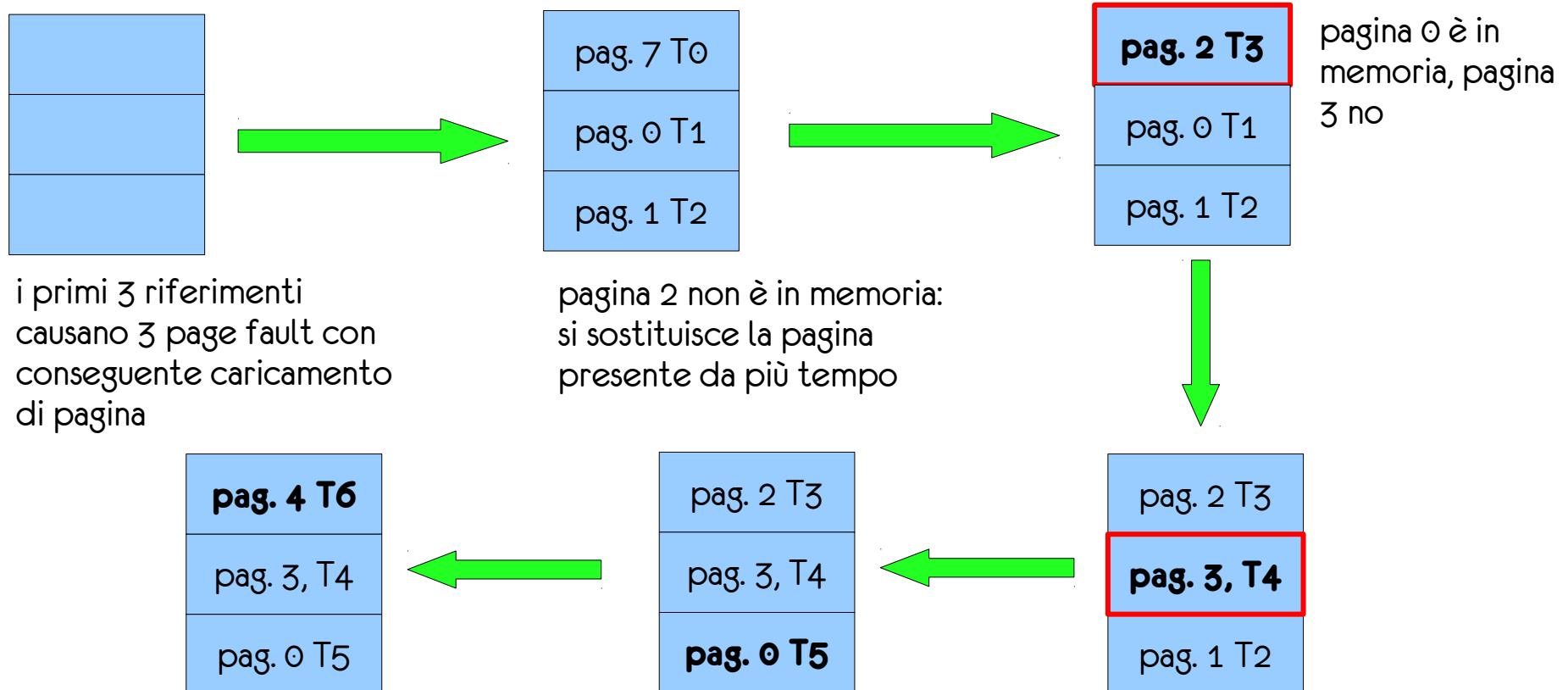
- a ogni pagina viene associato un marcatore temporale: l'istante in cui è stata caricata in memoria
- si sceglie di sostituire la pagina caricata meno di recente (quella in memoria da più tempo)
- basta organizzare le pagine in una coda FIFO
- si sostituisce sempre la pagina corrispondente al primo elemento della coda
- supponiamo di avere una RAM con 3 soli frame e di avere la seguente sequenza di riferimenti: 7, 0, 1, 2, 0, 3, 0, 4



i tre frame sono inizialmente vuoti

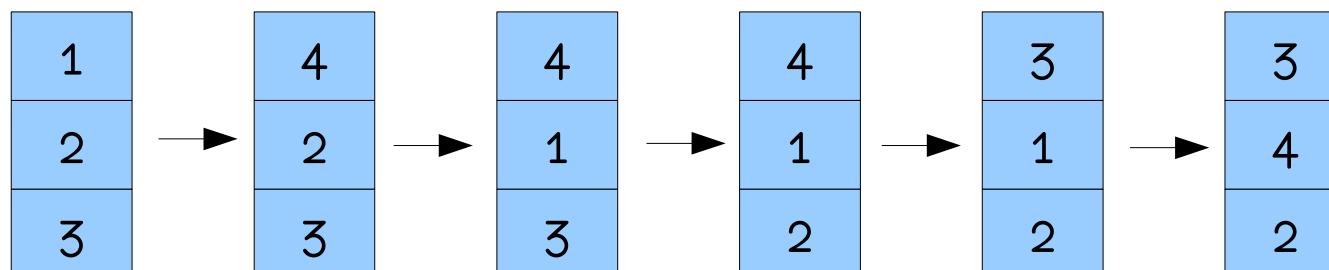
Sostituzione FIFO

- supponiamo di avere una RAM con 3 soli frame e di avere la seguente sequenza di riferimenti: 7, 0, 1, 2, 0, 3, 0, 4, 2, 3, 0, 3, 2, 1, ...



Commenti 1/2

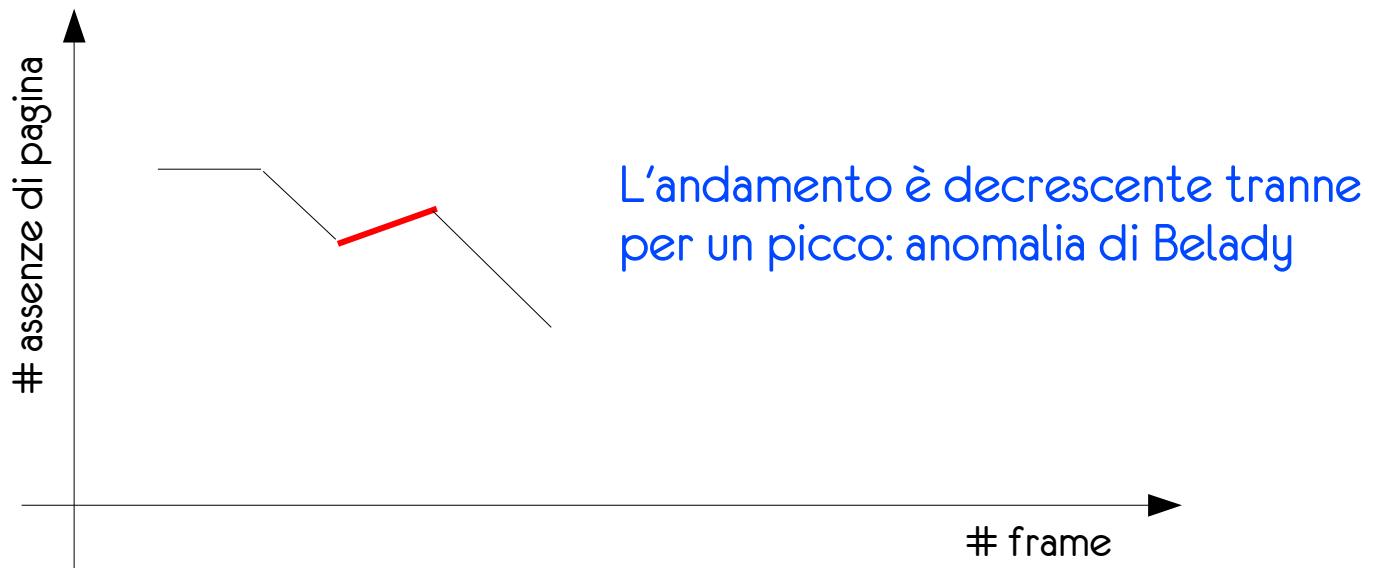
- FIFO è una tecnica semplice da realizzare ma non si comporta sempre bene
- il fatto che una pagina sia stata caricata tempo fa non significa che non sia più in uso, al contrario potrebbe essere una pagina di uso frequente: rimuoverla causerebbe sicuramente un successivo page fault
- consideriamo la sequenza **1, 2, 3, 4, 1, 2, 3, 4, 1, 2, 3, 4** sempre nel contesto di una RAM con tre frame:



ogni riferimento causa un page fault !!!

Commenti 2/2

- Il numero di page fault dipende dal numero di frame che compongono la RAM: in generale, maggiore il numero di frame, minore la frequenza di page fault
- Tuttavia l'algoritmo di sostituzione delle pagine FIFO presenta l'**anomalia di Belady**: la frequenza delle assenze può aumentare con l'aumentare del numero di frame in RAM
- Sul libro è riportato un esempio in cui si osserva questo andamento:



Algoritmo ottimale

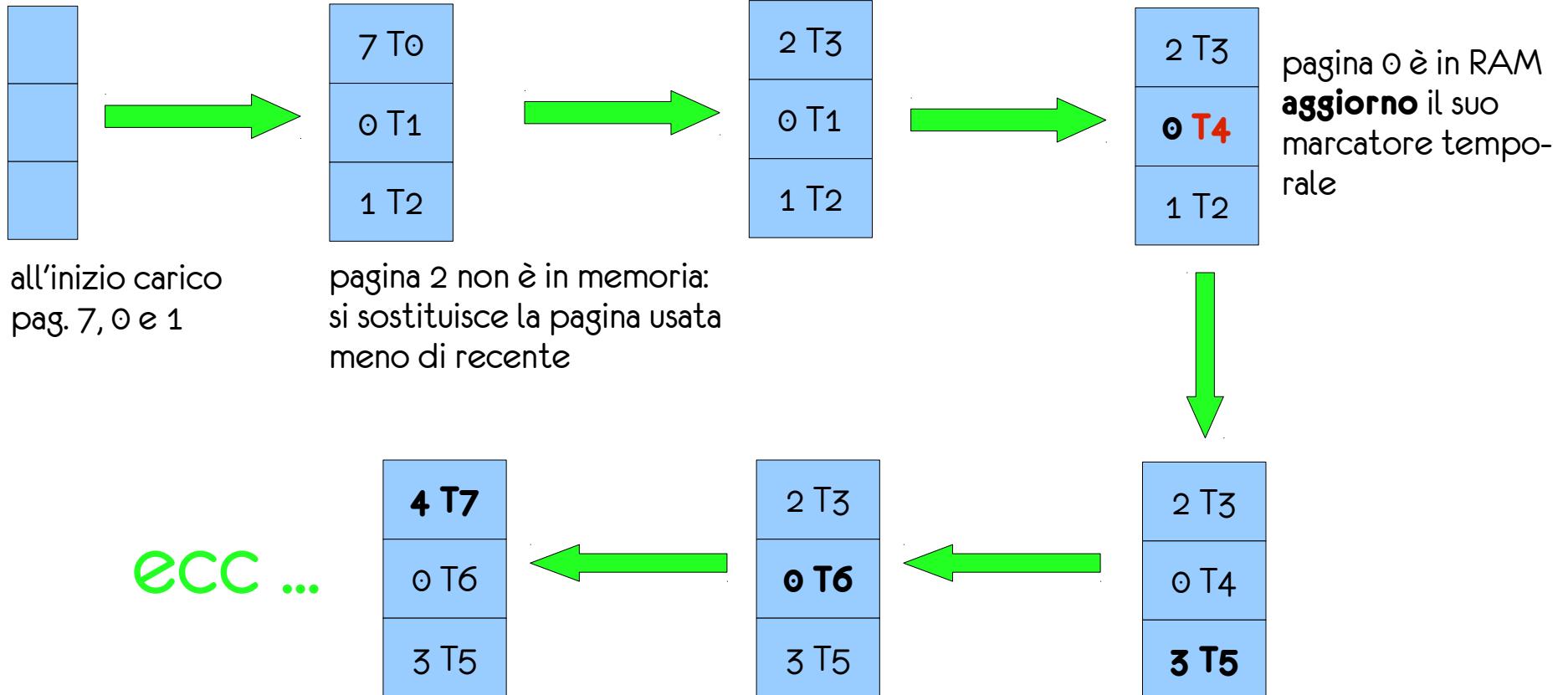
- L'algoritmo ottimale per la sostituzione delle pagine è noto come OPT o MIN
- Non presenta anomalia di Belady
- presenta la frequenza di page fault più bassa
- **criterio adottato:** si sostituisce la pagina che non verrà usata per il periodo di tempo più lungo
- Sfortunatamente **non è applicabile** in quanto non è possibile sapere a priori quando una pagina verrà richiesta in futuro



Least-recently used

- Simile all'algoritmo FIFO, ma anziché basare la scelta sul tempo di caricamento la basa sul tempo di utilizzo
- LRU è un'approssimazione di OPT, in cui si basa la stima del momento in cui una pagina sarà usata sulla base di quanto accaduto finora
- A ogni pagina si associa un marcatore temporale che corrisponde all'*istante di ultimo utilizzo*
- criterio: si sceglie la pagina usata meno di recente
- Proviamo ad applicare l'algoritmo sull'esempio visto per FIFO: 3 frame in RAM con sequenza dei riferimenti 7, 0, 1, 2, 0, 3, 0, 4, 2, 3, 0, 3, 2, 1, ...

7, 0, 1, 2, 0, 3, 0, 4, 2, 3, 0, 3, 2, 1, ...



Commenti

- LRU è un algoritmo molto considerato anche se poche architetture forniscono l'HW per applicarlo
- L'unico problema (per risolvere il quale occorre un supporto HW) è determinare la pagina da sostituire, infatti gli aggiornamenti del marcitore temporale causano un continuo rimescolio delle pagine nella sequenza
- Soluzione 1:
 - si aggiunge alla CPU un contatore incrementato a ogni riferimento alla memoria
 - associo a ogni elemento della tabella delle pagine un campo x mantenere il marcitore temporale
 - ogni volta che si accede a una pagina si aggiorna il suo marcitore usando il valore del contatore
- Problemi:
 - overflow del contatore
 - la ricerca della vittima nella tabella delle pagine è sequenziale

Commenti

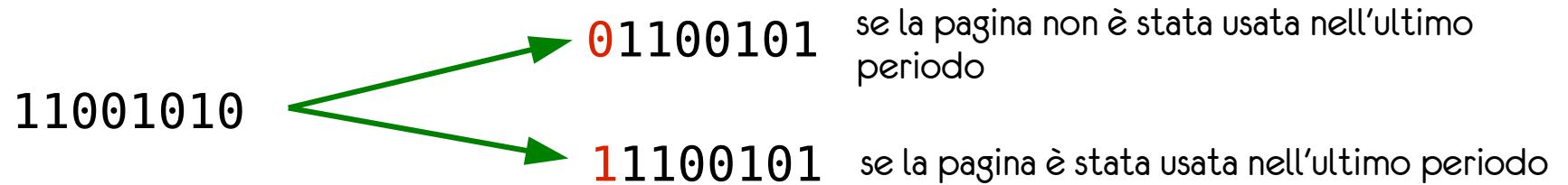
- **Soluzione 2:**
 - si mantiene uno stack di pagine
 - ogni volta che si fa riferimento a una pagina la si pone in cima allo stack (eventualmente togliendola da altra posizione nello stack)
 - la pagina in fondo allo stack è sempre quella usata meno di recente
- **Altri commenti:**
 - LRU non è soggetta ad anomalia di Belady

Approssimazione a LRU

- Poche architetture consentono di applicare l'LRU
- Alcune architetture consentono di applicarne un'approssimazione che si basa sull'uso del “**bit di riferimento**”
- anziché mantenere un contatore e un insieme di timestamp, si associa semplicemente a ogni elemento della tabella delle pagine **un bit**, che viene **impostato a 1 ad ogni accesso** (in lettura o in scrittura) alla pagina a cui fa riferimento
- all'inizio tutti i bit sono a 0
- dopo un po' alcuni bit saranno diventati 1
- manca l'informazione relativa a quando le varie pagine sono state **usate**
- esistono diversi metodi che si basano su questo supporto

Algo. con bit supplementare di riferimento

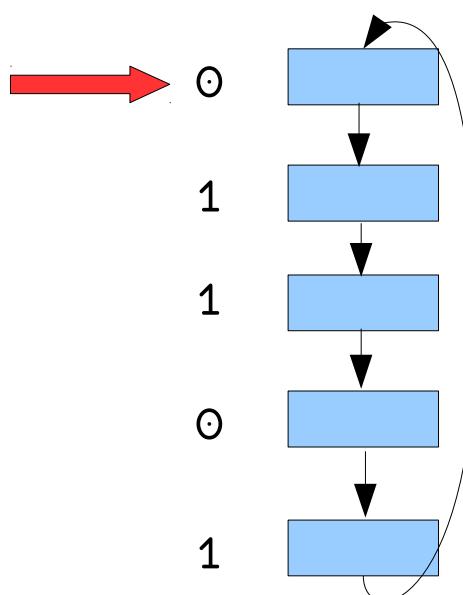
- È una tecnica di approssimazione della LRU in cui **si associa a ogni elemento nella tabella delle pagine una serie di bit che realizzano dei registri a scorrimento**
- A intervalli regolari un **timer** passa il controllo al SO, che sposta i bit nella sequenza traslandoli a destra di 1, copia il bit di riferimento nel bit più significativo della sequenza e scarta il bit meno significativo, es:



- quindi i **bit di riferimento** delle pagine vengono **azzerati**
- una pagina che ha il suo registro a **00000000** non è stata usata negli ultimi 8 periodi di tempo

Algo. di seconda chance

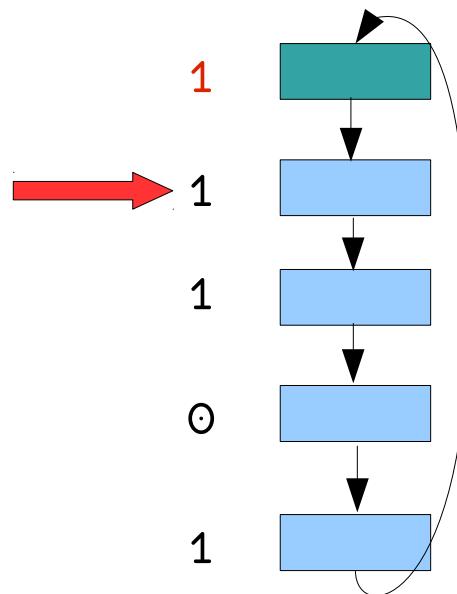
- è un algoritmo che unisce il metodo LRU all'approccio FIFO
- ogni pagina ha associato un bit di riferimento
- le pagine sono mantenute in una coda circolare FIFO



La freccia rossa indica il punto corrente di inizio della ricerca della vittima

La lista viene percorsa alla ricerca della prima pagina avente bit di riferimento a 0. Nell'esempio la pagina corrente diventa la vittima

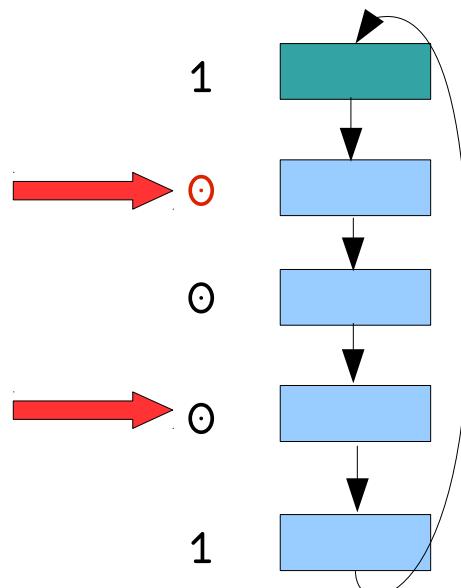
Algo. di seconda chance



Carico la pagina e le associo un bit di riferimento impostato ad 1

Quando diventerà necessario caricare una nuova pagina, la ricerca partirà dalla posizione successiva, in questo caso il bit di rif. ora vale 1

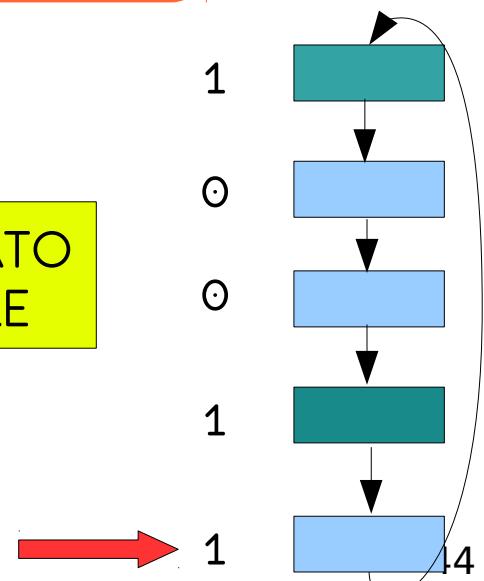
Algo. di seconda chance



Quando il bit di riferimento è impostato a 1: la pagina viene saltata ma il suo bit di riferimento viene azzerato.
Idem per la pagina successiva

A questo punto si incontra una pagina con bit di riferimento a 0 e la si sovrascrive mettendo il bit di riferimento a 1

RISULTATO FINALE



Osservazioni

- L'algoritmo di seconda chance è un'approssimazione dell'LRU in quanto mantiene l'informazione relativa alle pagine usate più di recente (bit di riferimento)
- **la struttura circolare della coda è un modo per concedere un po' di tempo in RAM a ciascuna pagina:** infatti una pagina con bit che passa da 1 a 0 non viene rimossa subito ma solo quando si tornerà alla sua locazione dopo aver percorso tutta la lista (avendo considerato e eventualmente scelto come vittima altre pagine)
- **Se tutti i bit di riferimento sono impostati a 1, l'algoritmo di seconda chance si trasforma nell'algoritmo FIFO**
- l'algoritmo può essere migliorato ...

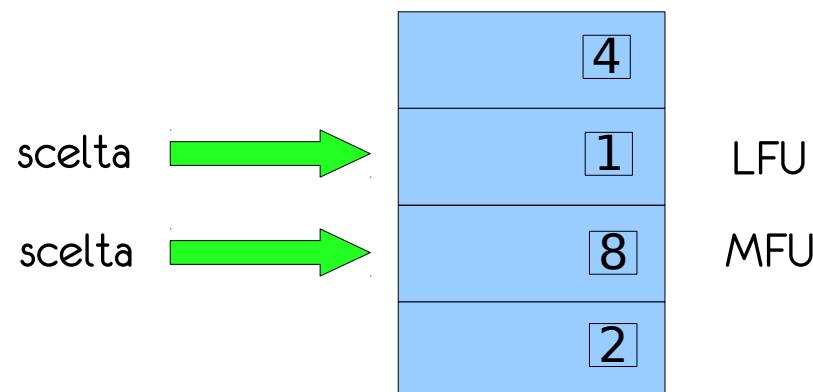
Algo. di seconda chance migliorato

- Ogni pagina ha associata una coppia di bit $\langle r, m \rangle$:
 - **bit di riferimento**: indica se la pagina è stata usata di recente
 - **bit di modifica**: indica se la pagina è stata modificata di recente
- Si individuano 4 classi di pagine:
 - $\langle 0, 0 \rangle$: né usata di recente né modificata
 - $\langle 0, 1 \rangle$: non usata di recente ma modificata
 - $\langle 1, 0 \rangle$: usata di recente ma non modificata
 - $\langle 1, 1 \rangle$: usata di recente e modificata
- Criterio di scelta: le classi sono ordinate sulla base del valore della coppia di bit, si sceglie una pagina appartenente alla classe in posizione inferiore fra quelle rappresentate in quel momento

00 < 01 < 10 < 11

Sostituzione su conteggio

- Fra i tanti algoritmi per la sostituzione delle pagine che sono stati ideati, particolarmente rilevanti quelli basati sul **conteggio del numero di riferimenti** fatti a ciascuna pagina
- Appartengono a questa categoria:
 - l'algoritmo **LFU** (least frequently used): sostituisce la pagina con il minor numero di riferimenti
 - l'algoritmo **MFU** (most frequently used): sostituisce la pagina con il maggior numero di riferimenti

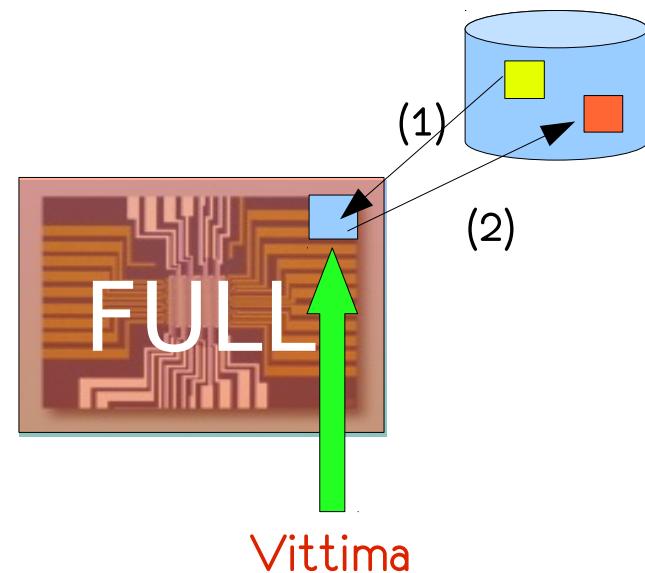


Commenti

- **LFU** si basa sull'idea che una pagina molto usata ha un conteggio alto, mentre una pagina che serve poco avrà un conteggio basso
- **problema**: le pagine sono solitamente usate per un certo periodo di tempo. Se non si ha un modo per ridurre nel tempo il conteggio degli accessi, non si riesce a distinguere fra una pagina che è stata molto usata ma ora non lo è più e una pagina che è attualmente molto usata
- **MFU**: si basa sul principio opposto che una pagina con un contatore basso è stata probabilmente appena caricata, quindi è utile

Pool of free frames

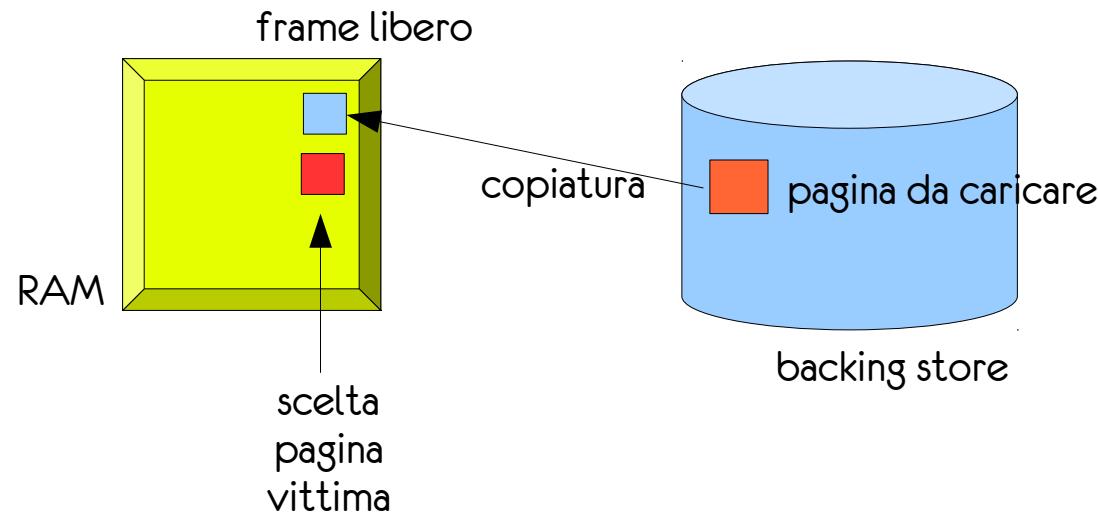
- Spesso gli algoritmi di sostituzione delle pagine sono affiancati da altre procedure finalizzate a incrementare le prestazioni del sistema
- Fra queste una tecnica che consente di iniziare la copiatura in RAM della pagina necessaria per proseguire (1) **prima** che la pagina vittima sia stata ricopiata in memoria secondaria (2)



Pool of free frames

- L'idea è associare a ogni processo un piccolo **pool di frame liberi**
- quando diventa necessario caricare una pagina nuova:
 - la si copia in un frame libero associato al processo
 - durante la copiatura, si sceglie una vittima e la si copia in backing store
 - in questo modo si libera un frame che viene aggiunto al pool di frame liberi del processo
- **NB:** la vittima non viene cancellata!
- fino a quando non viene sovrascritta, si tiene memoria della sua presenza in RAM, in questo modo non è necessario ricopiarla in caso di un successivo tentativo di accesso

Pool of free frames



La pagina vittima va ad arricchire il pool dei frame liberi assegnati al processo ma non viene cancellata o sovrascritta, al contrario rimane accessibile attraverso la tabella delle pagine

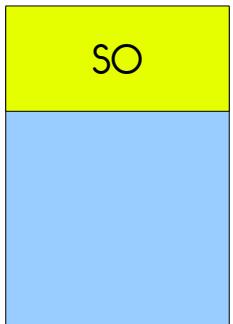
Allocazione dei frame

- per i processi utente
- per i processi kernel

Allocazione dei frame

- L'algoritmo di allocazione dei frame completa l'implementazione della memoria virtuale, iniziata con la definizione di un algoritmo di sostituzione delle pagine
- Questo algoritmo viene applicato quando si hanno a disposizione N frame liberi, occorre caricare uno o più processi e **bisogna decidere quanti frame assegnare a ciascuno** (come spartirli)
- Partiamo da uno **scenario** semplice, in un contesto di **paginazione su richiesta pura**:
 - 1 utente
 - RAM da 128K
 - pagine da 1K
 - un processo

Allocazione dei frame



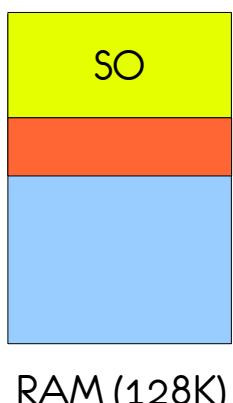
Una porzione deve comunque essere riservata al sistema Operativo



La porzione rimanente può essere allocata per il processo del singolo utente

RAM (128K)

Supponiamo che il SO occupi 28K e che si attui una gestione di paginazione a richiesta pura: il processo utente viene avviato senza caricare alcuna pagina, poi al primo page fault si effettua il primo caricamento



Quando ce ne sarà bisogno si caricherà la seconda pagina, ecc. ecc. ecc. fino a un massimo di 100K

La strategia di allocazione dei frame adottata consiste nel **non** assegnare inizialmente alcun frame libero ai processi

Allocazione dei frame

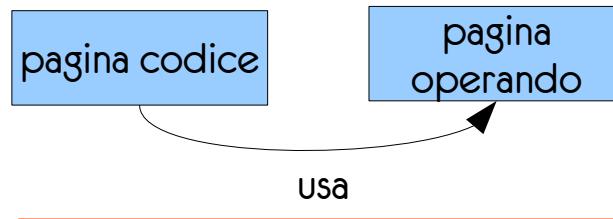
- Allocare inizialmente 0 frame per processo e poi 1 per volta quando necessario è l'unica scelta possibile? È la migliore?

Allocazione dei frame

- Allocare inizialmente 0 frame per processo e poi 1 per volta quando necessario è l'unica scelta possibile? È la migliore?
- Considerazione
 - La scelta operata è guidata dai page fault
 - Ogni page fault introduce grossi ritardi nell'esecuzione
- Scopo
 - page fault = ritardo, ritardo = inefficienza → noi desideriamo minimizzare i ritardi quindi cerchiamo di minimizzare la frequenza dei page fault
- Approccio
 - in generale il *#page fault* è inversamente proporzionale al *# di frame allocati* per ciascun processo
 - **idea:** cercare di mantenere in memoria un *numero minimo di frame* per processo tale da ridurre la probabilità che si generi un page fault

Numero minimo di frame

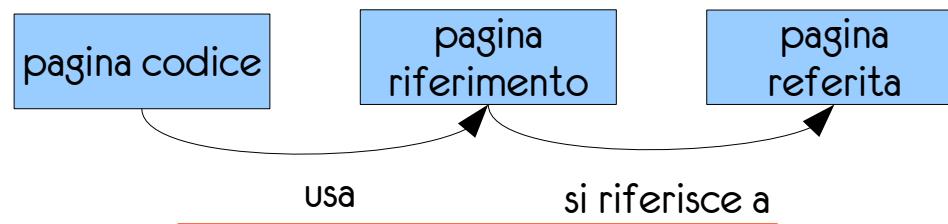
- Consideriamo un'**istruzione con un solo operando**
- È probabile che per caricarla ed eseguirla **occorrono almeno due pagine**:
 - una pagina di codice, contenente l'istruzione
 - una pagina di dati, contenente l'operando



per evitare page fault entrambe le pagine dovrebbero essere in RAM

Numero minimo di frame

- Se l'operando è un **riferimento in memoria** allora il numero di pagine sale a **tre**:
 - una pagina di codice, contenente l'istruzione
 - una pagina di dati, contenente l'operando
 - una pagina contenente il dato puntato dall'operando



per evitare page fault tutte e tre le pagine dovrebbero essere in RAM

Numero minimo di frame

- ...
- **Inoltre:**
 - un'istruzione può occupare uno spazio abbastanza grande da stare a **cavallo di due pagine**. Se ciò accade occorreranno almeno due pagine per il solo caricamento dell'istruzione
 - si possono avere **più livelli di indirizzamento indiretto**: nel caso peggiore tutta la memoria virtuale dovrebbe essere caricata in RAM per evitare page fault

Numero minimo di frame

- Il numero minimo di frame necessari al caricamento e all'esecuzione di un'istruzione dipende dall'architettura
- Supponiamo di avere M frame liberi ed N processi:
 - invece di applicare una **paginazione su richiesta pura** posso attuare una politica diversa e decidere di riservare ad ogni processo un certo numero di frame, caricando subito più pagine in RAM:
 - tornando ai nostri 100K di RAM liberi (frame da 1 K), se dobbiamo caricare 4 processi, possiamo pensare di assegnare a ciascuno 24 frame e di lasciarne 4 come pool di frame liberi (**allocazione uniforme**)
 - **in alternativa:** poiché i processi hanno dimensione diversa, alloco per ciascun processo un #frame proporzionale alla sua dimensione (**allocazione proporzionale**)

Allocazione proporzionale

- Indichiamo con VM^i la dimensione della memoria logica occupata dal generico processo P^i
- La quantità di memoria logica occupata da N processi sarà $V = \sum_{i \in [1, N]} VM^i$
- Indicando con M il numero di frame disponibili, il numero di frame allocati al processo P^i sarà:

$$m = \frac{VM^i}{V} \times M$$

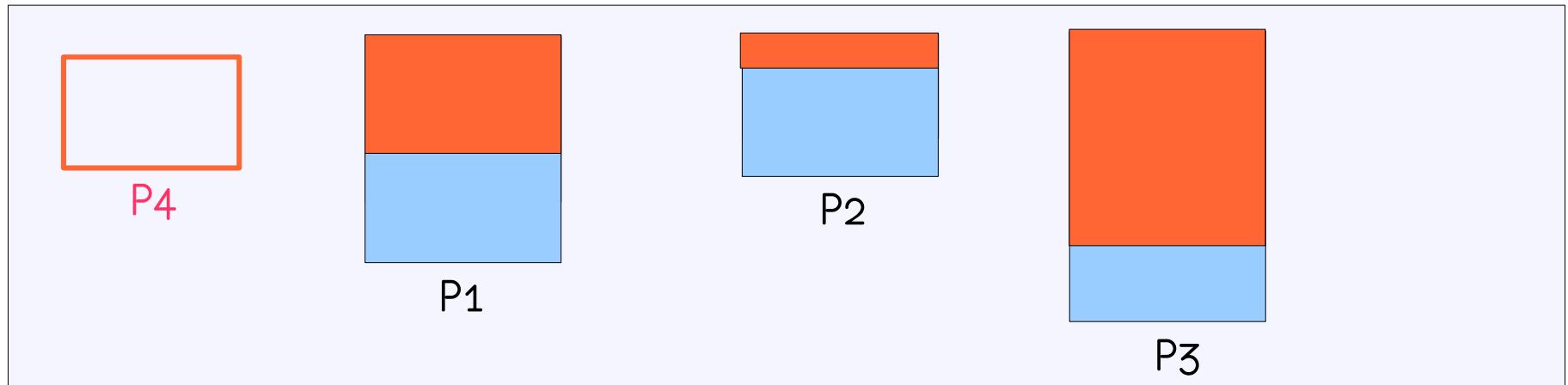
- Se abbiamo definito un num. minimo di frame necessari per caricare un'istruzione, potrebbe essere necessario incrementare tale valore di qualche unità per i processi più leggeri, decrementando di conseguenza i valori assegnati ai processi più pesanti

Dinamicità

- NB: il numero di processi in RAM è una funzione del tempo
- se il livello di multiprogrammazione cresce (num. processi in RAM aumenta) occorrerà redistribuire un certo numero di frame ancora liberi ai nuovi processi
- se il livello di multiprogrammazione diminuisce, sarà possibile assegnare ai processi in RAM un numero maggiore di frame

Riassumendo

In generale la RAM è suddivisa a priori fra i processi secondo un certo criterio: ogni processo ha un tot di frame



In ogni istante una parte dei frame riservati x un processo sarà occupata e una parte libera

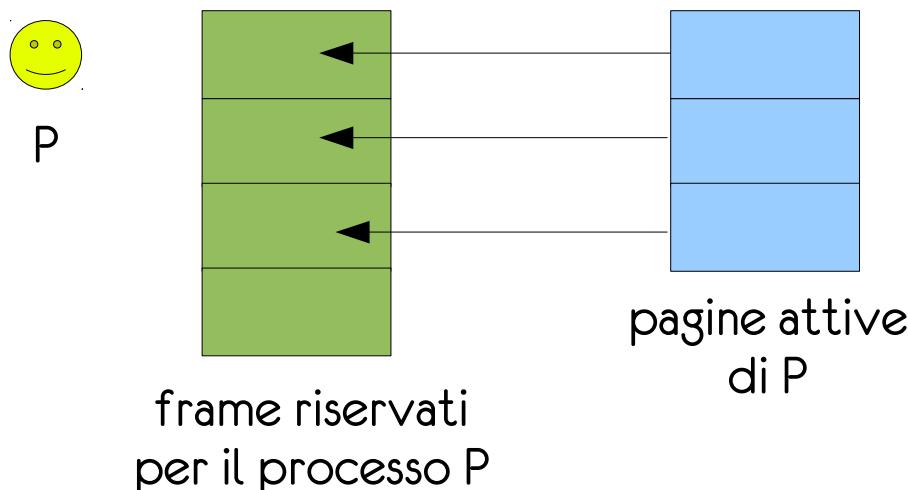
Di tanto in tanto l'ingresso di un nuovo processo causerà una **riassegnazione dei frame rimasti liberi**

Allocazione globale/locale

- Una questione che non abbiamo ancora considerato è la **priorità dei processi** in connessione all'**allocazione dei frame**
- Aspettativa: processi a priorità maggiore hanno a disposizione un maggior numero di frame
- Le strategie viste sono “eque” da questo punto di vista: **i processi hanno tutti la stessa importanza**
- Consideriamo il caso particolare in cui un processo ad alta priorità esaurisce il proprio pool di frame. Ci sono due possibilità:
 - si sacrifica **una delle sue pagine**, sostituendola con quella di interesse (**allocazione locale** delle pagine: uso solo le pagine riservate per il processo)
 - si sacrifica **un altro processo**, che ha un frame libero, sottraendoglielo (**allocazione globale** delle pagine, posso usare qualsiasi frame libero)

Thrashing

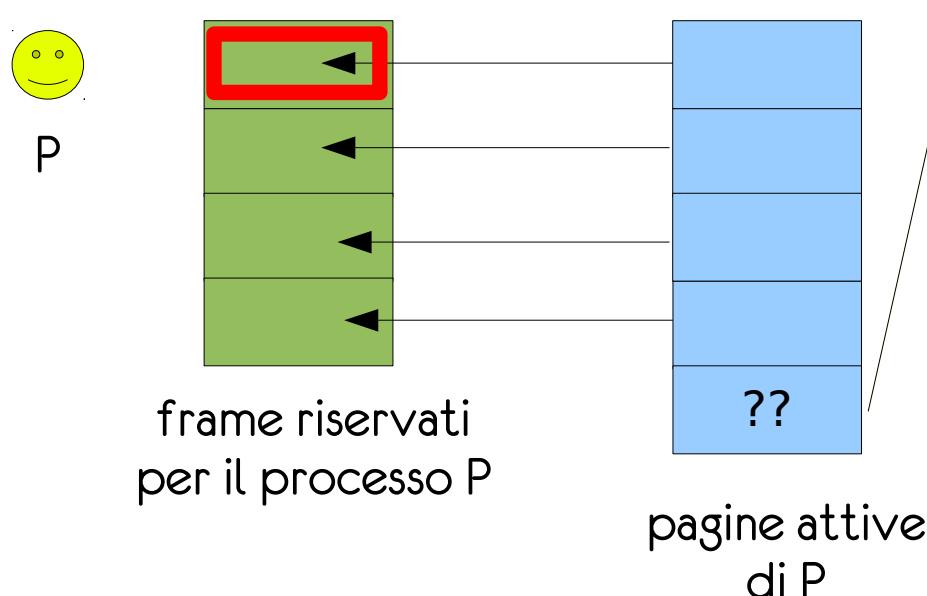
- un altro aspetto da discutere è un fenomeno noto come **thrashing**
- il thrashing è un **fenomeno degenerativo** che si può verificare nella gestione della memoria virtuale
- ogni processo ha un **numero minimo di frame** riservati
- l'esecuzione di ciascun processo si appoggia in ogni istante a un certo numero di pagine “attive”



Il numero di pagine attive cambia nel tempo: supponiamo che cresca: può succedere che a un certo punto vi siano più pagine attive che frame a disposizione

Thrashing

- un altro aspetto da discutere è un fenomeno noto come **thrashing**
- il thrashing è un **fenomeno degenerativo** che si può verificare nella gestione della memoria virtuale
- ogni processo ha un **numero minimo di frame** riservati
- l'esecuzione di ciascun processo si appoggia in ogni istante a un certo numero di pagine “attive”

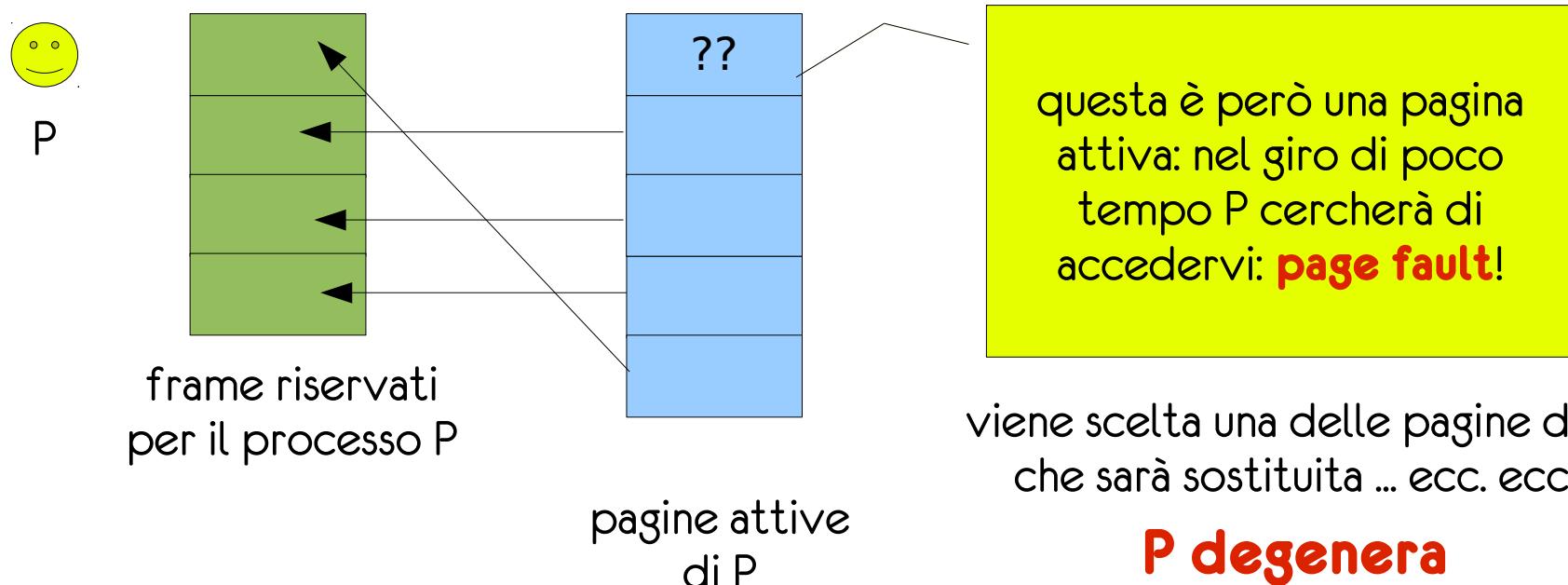


un accesso a questa pagina produce un **page fault**: si applica l'algoritmo di sostituzione per determinare una pagina da sostituire

Supponiamo che la strategia di allocazione sia locale: viene scelta una delle pagine di P

Thrashing

- un altro aspetto da discutere è un fenomeno noto come **thrashing**
- il thrashing è un **fenomeno degenerativo** che si può verificare nella gestione della memoria virtuale
- ogni processo ha un **numero minimo di frame** riservati
- l'esecuzione di ciascun processo si appoggia in ogni istante a un certo numero di pagine “attive”

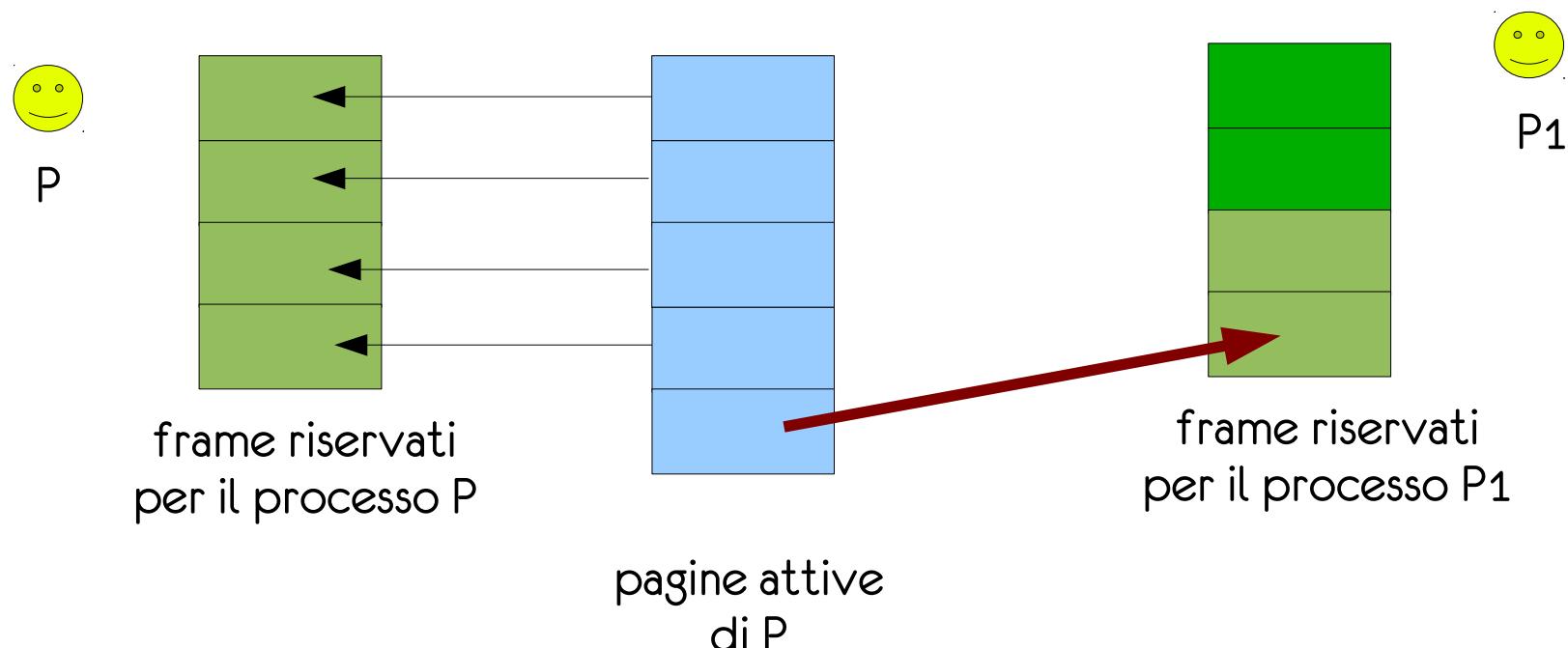


Thrashing

- Quando il numero di pagine attive è maggiore del numero minimo di frame riservato per il processo ed occorre caricare una nuova pagina (page fault), il normale processo di sostituzione sceglierà come vittima una pagina attiva (per forza di cose)
- a questo punto si innesta un meccanismo perverso: per proseguire il processo si produce un page fault, che genera una sostituzione che rimuove una pagina utile quasi immediatamente, si produce un nuovo page fault, che sostituisce una pagina attiva, e così via
- in breve si spende più tempo nel sostituire pagine che nell'eseguire il processo: si parla di thrashing

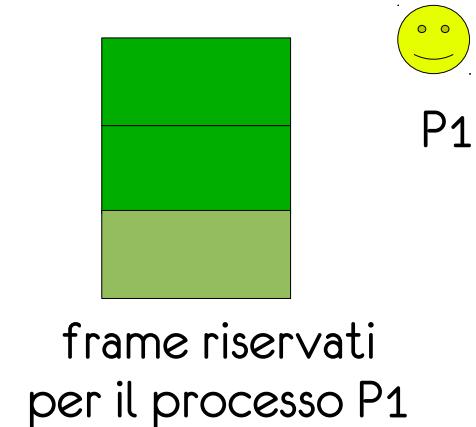
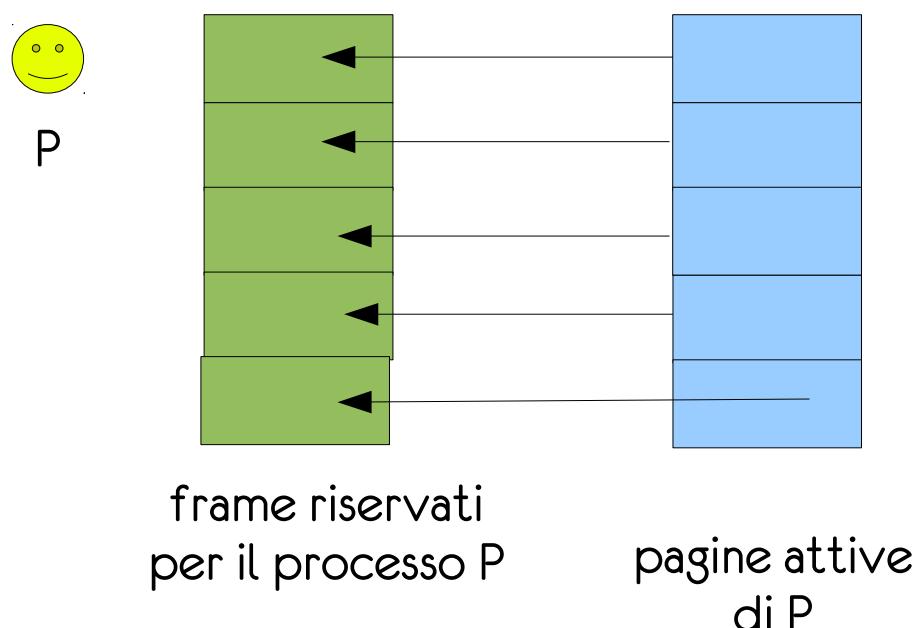
Thrashing

- Possibili alternative?
- e se il meccanismo di allocazione fosse globale?
- in questo caso potrebbe essere scelto un frame di un'altro processo



Thrashing

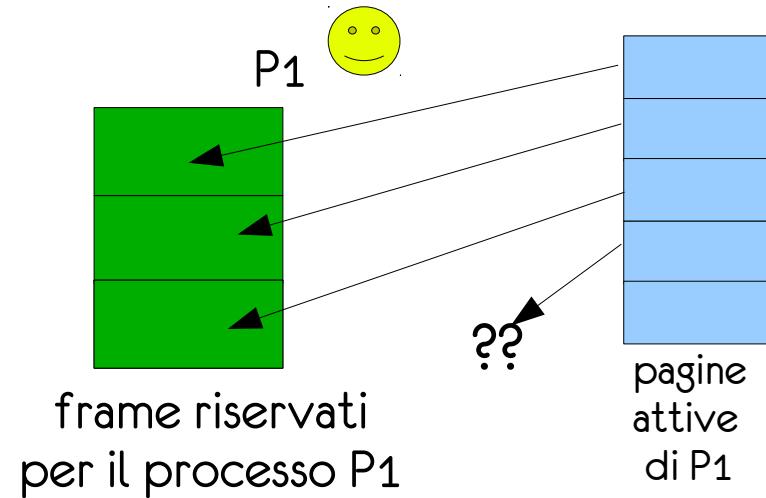
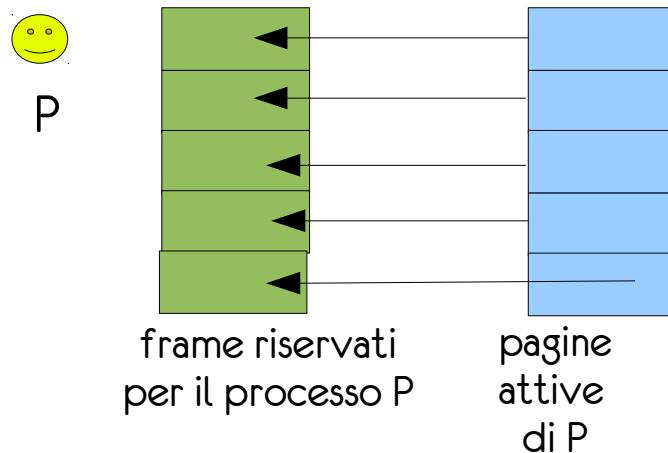
- e se il meccanismo di allocazione fosse globale?
- in questo caso potrebbe essere scelto un frame di un'altro processo



P ha sottratto un frame a P1

Thrashing

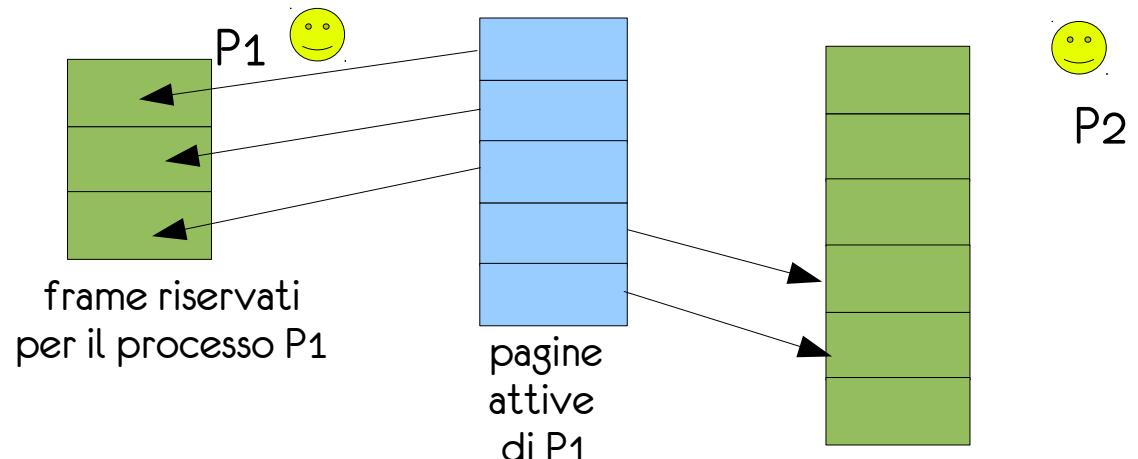
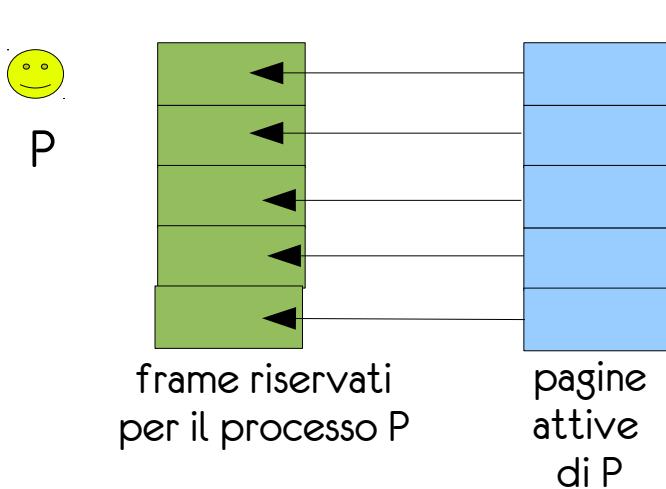
- e se il meccanismo di allocazione fosse globale?
- in questo caso potrebbe essere scelto un frame di un'altro processo



A questo punto se il numero di pagine attive per P1 cresce, P1 riempirà i propri frame residui ...

Thrashing

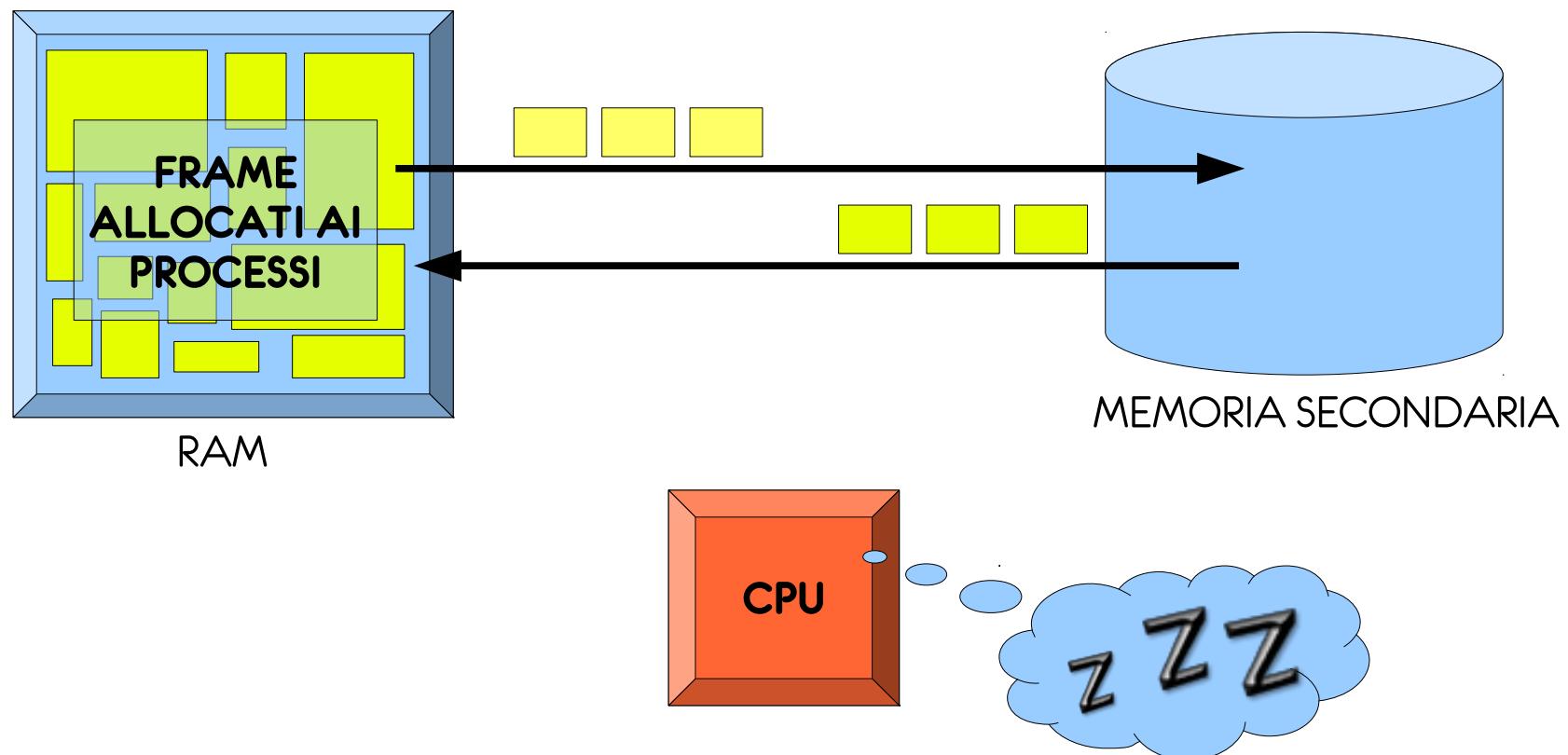
- e se il meccanismo di allocazione fosse globale?
- in questo caso potrebbe essere scelto un frame di un'altro processo



A questo punto se il numero di pagine attive per P2 cresce, P2 riempirà i propri frame residui ...

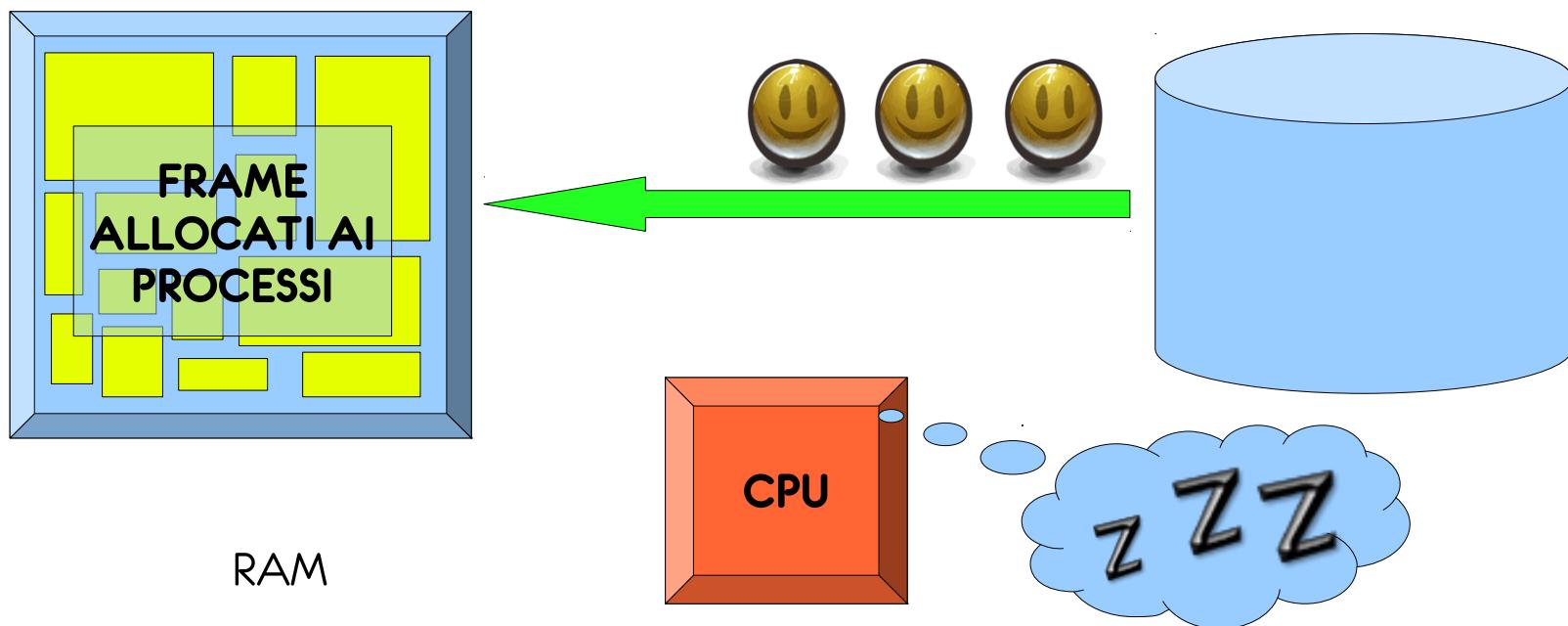
Effetti del thrashing

- quando si ha thrashing perché i processi hanno a disposizione meno frame del numero di pagine attive, l'attività della CPU tende a diminuire: i processi tendono a rimanere in attesa del completamento di operazioni di I/O



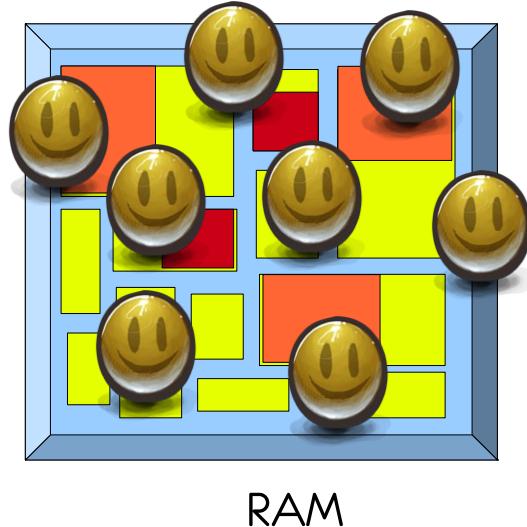
Thrashing e multiprogrammazione

- Molti SO monitorizzano l'uso della CPU: quando questo cala, se ci sono processi conservati in memoria secondaria, portano in RAM qualche processo in più ...



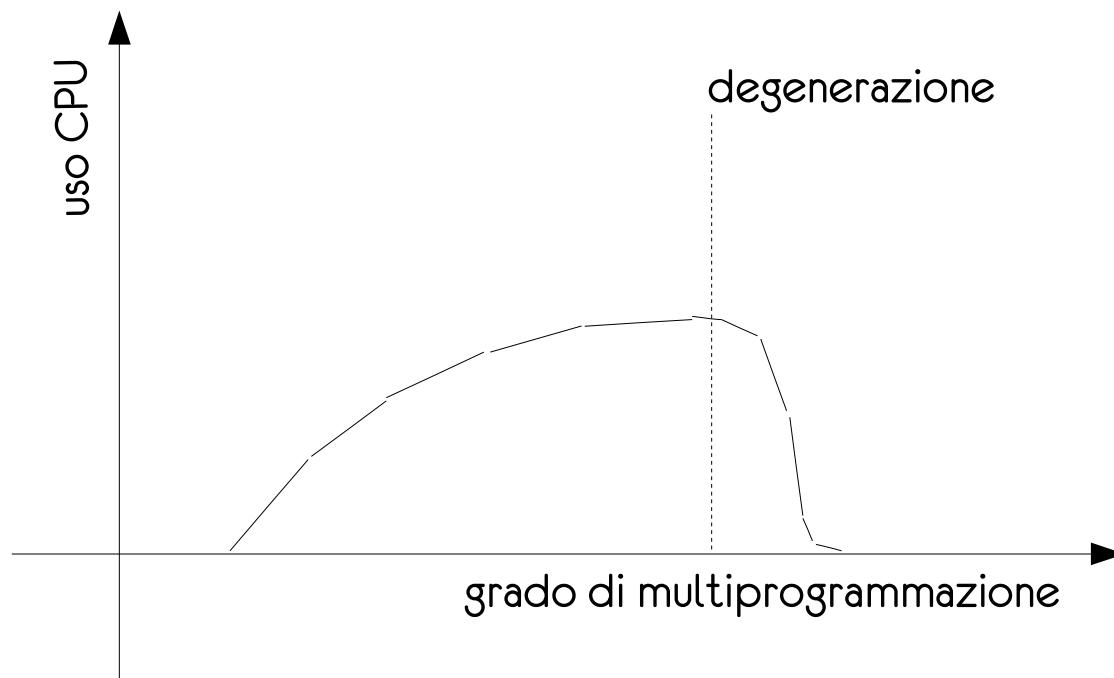
Thrashing e multiprogrammazione

- quando si ha thrashing perché i processi hanno a disposizione meno frame del numero di pagine attive, l'attività della CPU tende a diminuire (molto I/O)
- Molti SO monitorizzano l'uso della CPU: quando questo cala, se ci sono processi conservati in memoria secondaria, portano in RAM qualche processo in più ...
- Ai nuovi processi vengono allocati alcuni frame ...
 - ... sottratti ad altri processi in RAM se ...
 - ... la strategia di allocazione è globale
 - aumenta la frequenza di page fault
 - diminuisce l'utilizzo della CPU
 - ecc. fino al blocco totale



Andamento

- se si monitora l'uso della CPU, in presenza di thrashing si ha questo andamento



Cambiando strategia ...

- Si ha thrashing perché la **politica di allocazione dei frame è globale**: il SO può sottrarre frame ad un processo per assegnarli ad un altro processo
- un processo degenere, sottraendo frame agli altri processi, renderà degeneri pure questi
- se adottassimo un **algoritmo locale**, un processo degenere non potrebbe rendere degeneri altri processi perché non potrebbe sottrarre loro dei frame
- anche in questo caso però, la frequenza di page fault del processo degenere influenzerebbe il tempo di accesso effettivo degli altri processi a causa del sovraccarico causato al dispositivo di paginazione
- ... è il meglio che possiamo fare ?

Pagine attive

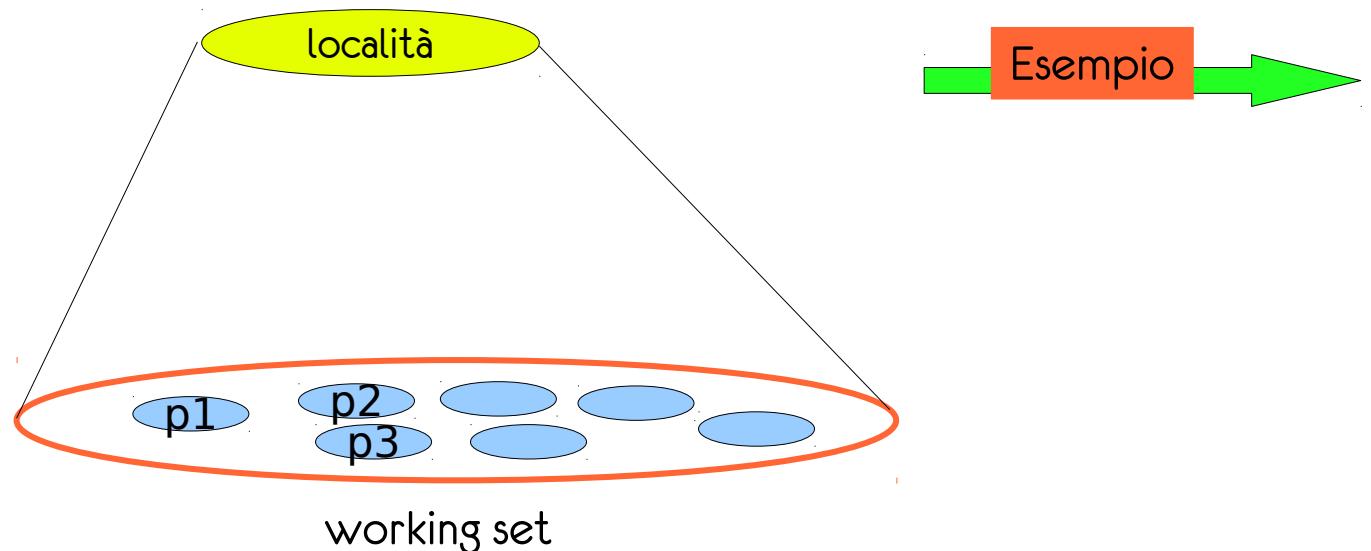
- l'ideale è cercare di prevedere di quante pagine avrà bisogno un processo e assegnargli un numero di frame sufficiente
- L'allocazione non sarà di tipo uniforme né sarà di tipo proporzionale ma dipenderà dal numero di pagine attive per ciascun processo
- non si distribuiscono tutti i frame liberi fra i processi, solo quelli ad essi necessari
- poiché il numero di pagine attive varia nel tempo:
 - se cresce si allocano nuovi frame
 - se decresce si liberano frame
- questo approccio si realizza nel modello a **Working Set**

Working set

- l'esecuzione dei processi può essere descritta sulla base di un **principio di località** che cattura il fatto che ogni processo accede, per periodi di tempo di durata consistente, solo a un **sottoinsieme delle variabili globali**, a un certo **insieme di variabili locali** e a un **sottoinsieme delle istruzioni** che compongono il suo codice
- con il termine **working set** si intende l'insieme delle pagine attive di un processo, cioè l'insieme delle pagine che il processo sta usando
- tale insieme varia nel tempo, per es. invocando una nuova procedura si può focalizzare l'esecuzione su un diverso working set

Working set

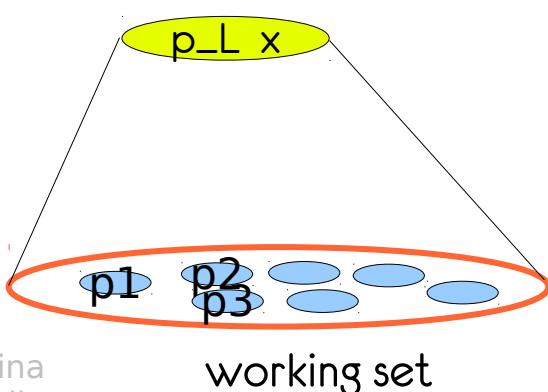
- il working set può essere visto come l'insieme delle pagine che implementa una località del processo
- il concetto di località può essere visto come un'astrazione del concetto di working set



Località e working set

```
void push(lista *p_L, double x)
{
    printf("push %.2f \n",x);
    if(p_L == NULL)
    {
        *p_L = (lista) malloc (sizeof(struct nodo));
        if(p_L == NULL) return;
        (*p_L) -> info = x;
    }
    else ...
}
```

Finché permango in una località
se il SO ha caricato tutto il work-
ing set relativo non si avranno
page fault

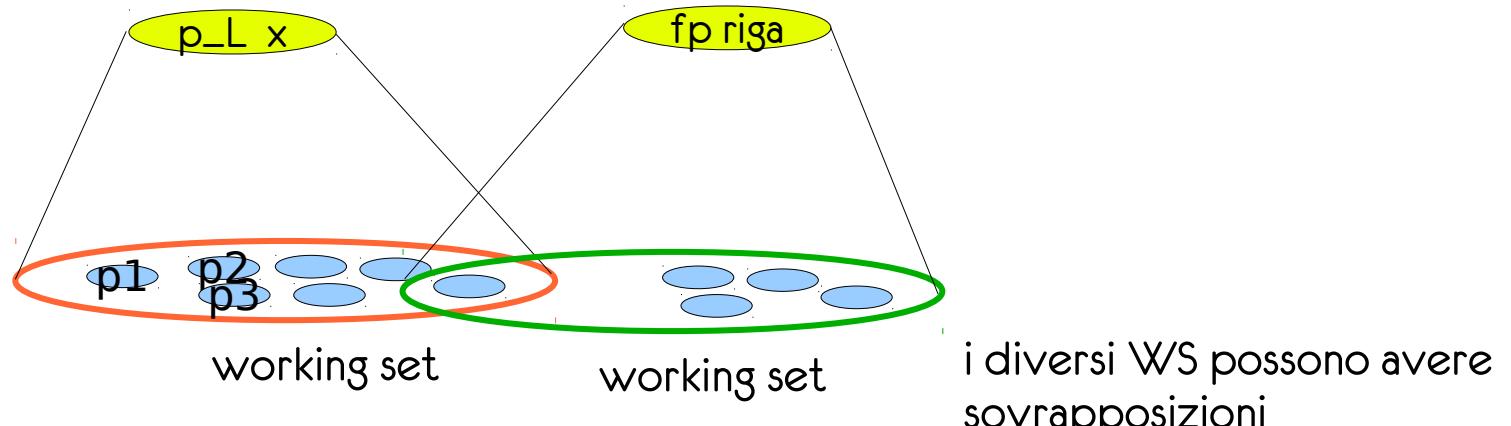


Quando eseguo la funzione push uso le variabili locali `p_L` e `x` ed eseguo le istruzioni che in parte vediamo: la località è data dal codice di push e dalle variabili `p_L` e `X`. Tale località si implementa in un insieme di pagine che contiene effettivamente il codice in que-
stione e le variabili locali menzionate

Località e working set

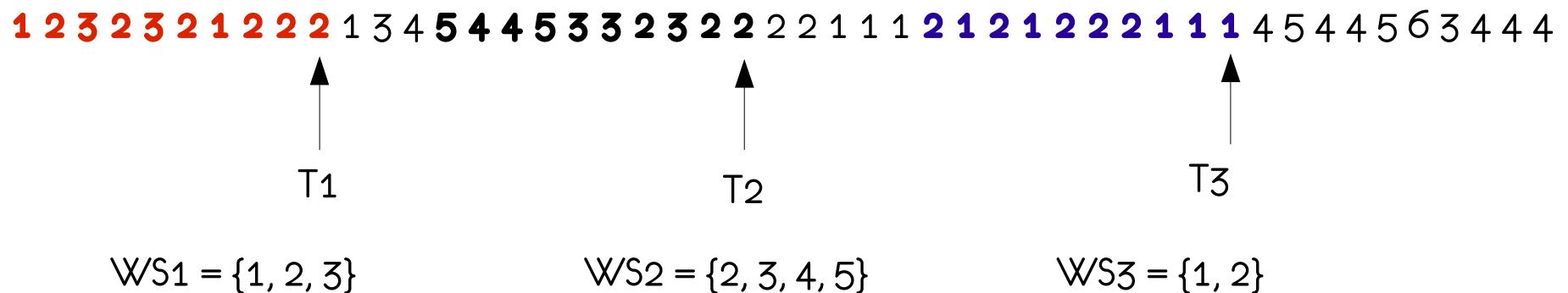
```
FILE *fp = fopen("dati", "r");  
char riga[128];  
  
fread(riga, 128, 1);  
while(!feof(fp)) {  
    printf("%s\n", riga);  
    fread(riga, 128, 1);  
}
```

cambiando località
cambierà l'insieme delle pagine
coinvolto nell'esecuzione



Modello del working set

- si basa sul principio di località
- viene definito un **parametro δ** (ampiezza della finestra di analisi)
- per ogni processo si analizzano gli **ultimi δ riferimenti alle pagine**
- le pagine individuate costituiscono il **working set corrente**
- es. sia $\delta = 10$



Working set e δ

1 2 3 2 3 2 1 2 2 2 1 3 4 5 4 4 5 3 3 2 3 2 2 2 2 1 1 1 2 1 2 1 2 2 2 1 1 1 4 5 4 4 5 6 3 4 4 4

\uparrow \uparrow \uparrow

T_1 T_2 T_3

$\delta = 10$

$WS1 = \{1, 2, 3\}$ $WS2 = \{2, 3, 4, 5\}$ $WS3 = \{1, 2\}$

1 2 3 2 3 2 1 2 2 2 1 3 4 5 4 4 5 3 3 2 3 2 2 2 2 1 1 1 2 1 2 1 2 1 2 2 2 1 1 1 4 5 4 4 5 6 3 4 4 4

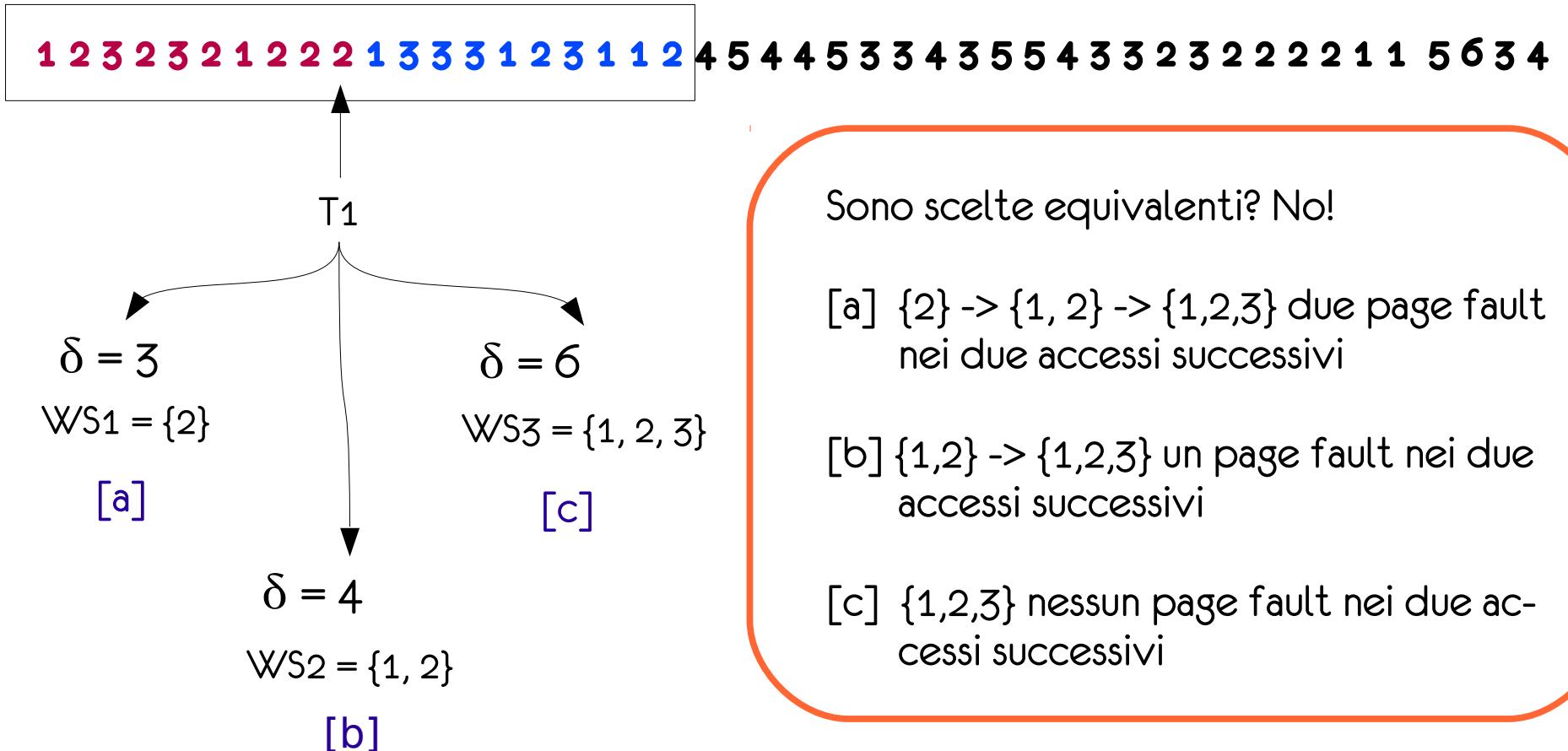
\uparrow \uparrow \uparrow

T_1 T_2 T_3

$\delta = 5$

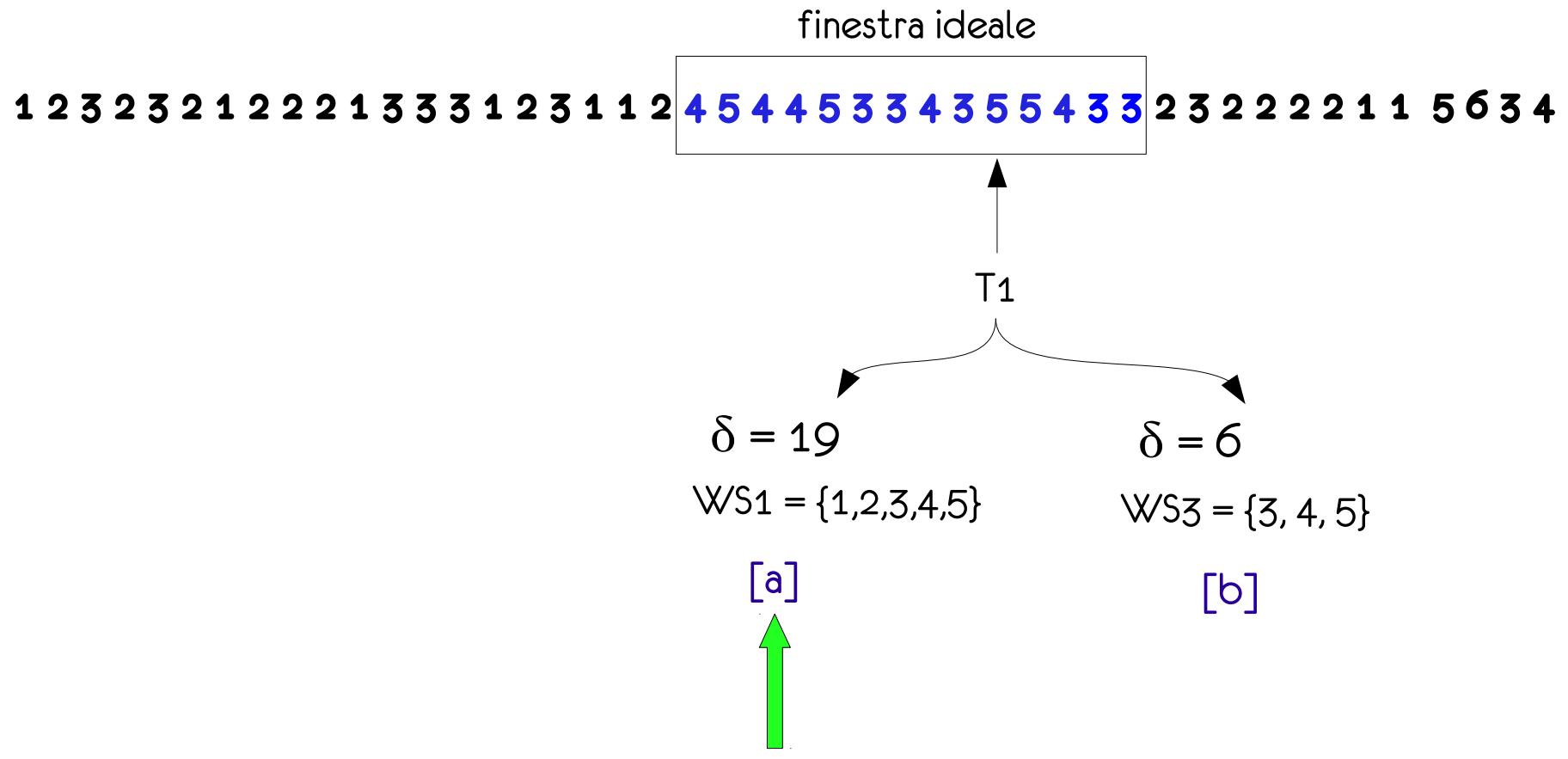
$WS1 = \{1, 2\}$ $WS2 = \{2, 3\}$ $WS3 = \{1, 2\}$

Working set e δ



δ troppo piccolo non coglie l'intera località

Working set e δ



δ troppo grande \rightarrow spreco di RAM, si sovrappongono più località

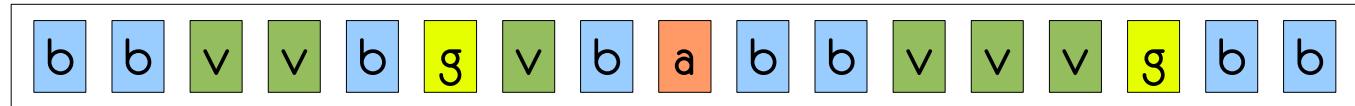
Uso del working set

- Definita la dimensione della finestra δ è possibile calcolare il WS di ogni processo e quindi calcolare la quantità di frame attualmente necessari ai processi in esecuzione:

$$D = \sum_{i \in [1, N]} |WS^i|$$

- Quando D diventa maggiore del numero di frame liberi, si genera il thrashing perché qualche processo non ha a disposizione un numero sufficiente di frame
- Se i frame sono invece sufficienti il SO può assegnare a ciascun processo una quantità di frame adeguata
- Se c'è un surplus di frame liberi è possibile avviare nuovi processi
- Quando il WS di un processo cresce, se non ci sono frame liberi, il SO sceglie uno dei processi, lo copia in memoria secondaria, lo sospende e assegna (parte dei) suoi frame al processo richiedente: così si evita il thrashing

Tener traccia del Working set



- Una pagina fa parte di un WS se esiste **almeno un riferimento ad essa** all'interno della finestra di analisi
- La verifica viene fatta a **intervalli regolari** (timer), es. ogni 1000 riferimenti
- δ nella realtà è un valore $\approx 10.000 / 15.000$
- Occorre determinare un **modo efficiente** per calcolare il WS di un processo
- **Soluzione:** si possono usare i bit di riferimento e i registri a scorrimento già descritti parlando di algoritmi di sostituzione delle pagine

Prepaginazione

- **problema inevitabile:**
 - per via del modo in cui sono definite, a ogni cambiamento di località si ha un certo numero di page fault (caricamento della nuova località):
 - es. **1 2 3 2 2 1 1 2 3 2 3 4 5 6 5 5 4 6 6 4 5 5 4**
- **problema evitabile:**
 - un problema a cui si può invece ovviare, legato alla **paginazione pura** (*carico una pagina solo quando viene riferita*), riguarda l'**eliminazione dei page fault ad ogni swap in** del processo
 - se si memorizza, associato al PCB del processo, anche il WS del processo al momento della sospensione, è possibile caricare subito in memoria tutte le pagine utili, senza dover passare attraverso a una serie di page fault
 - questa tecnica è nota come **prepaginazione**

Page fault frequency

- L'approccio a working set ha riscosso un notevole successo (anche perché consente di effettuare la prepaginazione)
- per evitare il fenomeno del thrashing in sé, esistono tecniche alternative
- in particolare la strategia basata sulla frequenza delle assenze di pagina (**page fault frequency**)
- La frequenza dei page fault aumenta considerevolmente in presenza di thrashing, un modo per evitare la degradazione è porre un limite superiore alla frequenza di page fault accettata
- se un processo supera il limite gli si allocherà un nuovo frame
- d'altro canto, il fatto che tale frequenza scenda molto può essere sinonimo di un WS sovradimensionato

Esercizio

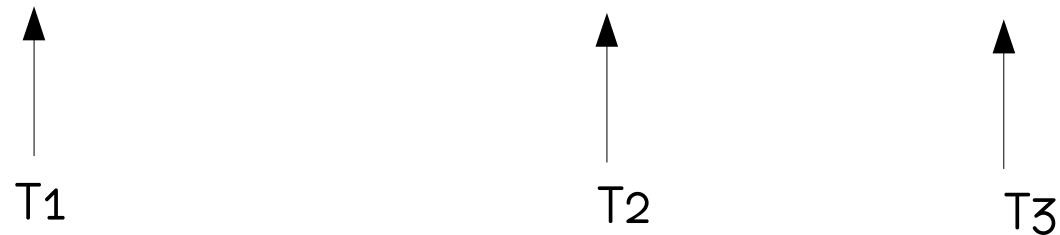
- Quante page fault può creare un'istruzione a due operandi (ISTR OP1 OP2) nel caso sia possibile un indirizzamento indiritto a due livelli?

soluzione

- *Quante page fault può creare un'istruzione a due operandi ($ISTR\ OP1\ OP2$) nel caso sia possibile un indirizzamento indiritto a due livelli?*
- un page fault per l'istruzione in sé
- tre page fault per ogni operando (OP stesso + il primo eventuale riferimento + il secondo eventuale riferimento)
- se occorre salvare un risultato, un page fault per il risultato
- totale: 8 page fault

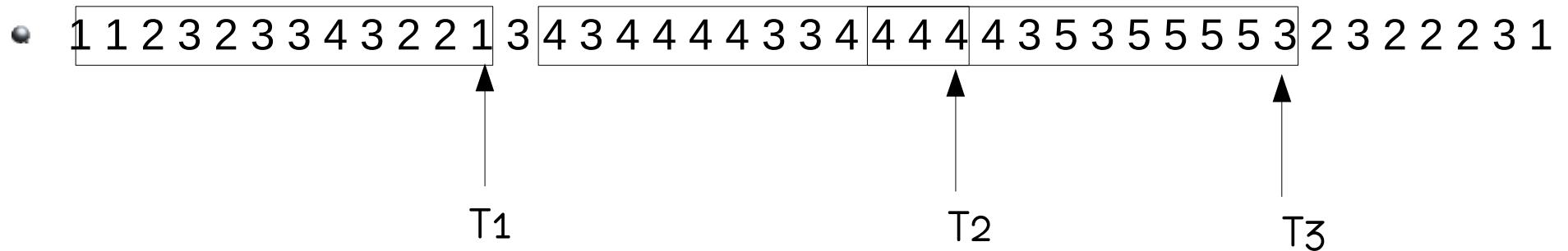
Esercizio

- Calcolare il WS agli istanti T_1 , T_2 e T_3 per un processo P che richiede la sequenza di accessi alle pagine indicata sotto, nel caso in cui δ sia alternativamente uguale a 8, 12 e 4
- 1 1 2 3 2 3 3 4 3 2 2 1 3 4 3 4 4 4 4 3 3 4 4 4 4 4 4 3 5 3 5 5 5 5 3 2 3 2 2 2 3 1



Esercizio

- Calcolare il WS agli istanti T_1 , T_2 e T_3 per un processo P che richiede la sequenza di accessi alle pagine indicata sotto, nel caso in cui δ sia alternativamente uguale a 8, 12 e 4



$$\delta = 12$$

$WS_1 = \{1,2,3,4\}$
 $WS_2 = \{3,4\}$
 $WS_3 = \{3,4,5\}$

$$\delta = 8$$

$WS_1 = \{1,2,3,4\}$
 $WS_2 = \{3,4\}$
 $WS_3 = \{3,5\}$

$$\delta = 4$$

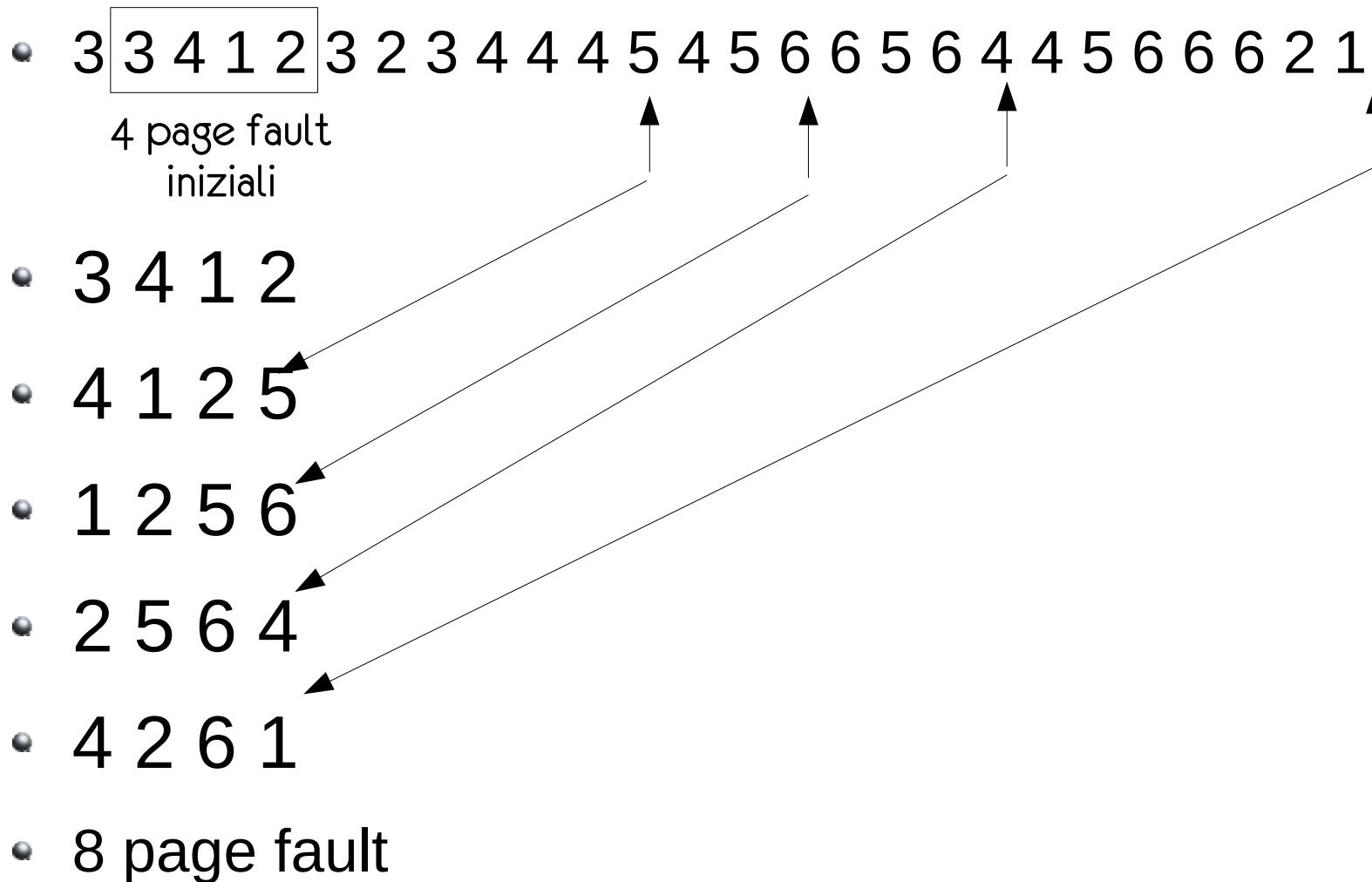
$WS_1 = \{1,2,3\}$
 $WS_2 = \{4\}$
 $WS_3 = \{3,5\}$

esercizio

- Supponendo di avere a disposizione 4 frame, indicare il numero di page fault che si avrebbero nell'eseguire accessi, secondo la sequenza di pagine indicata sotto, nel caso in cui si utilizzi l'algoritmo FIFO e nel caso in cui si utilizzi l'algoritmo LRU:
- 3 3 4 1 2 3 2 3 4 4 4 5 4 5 6 6 5 6 4 4 5 6 6 6 2 1

soluzione 1

- Algoritmo FIFO

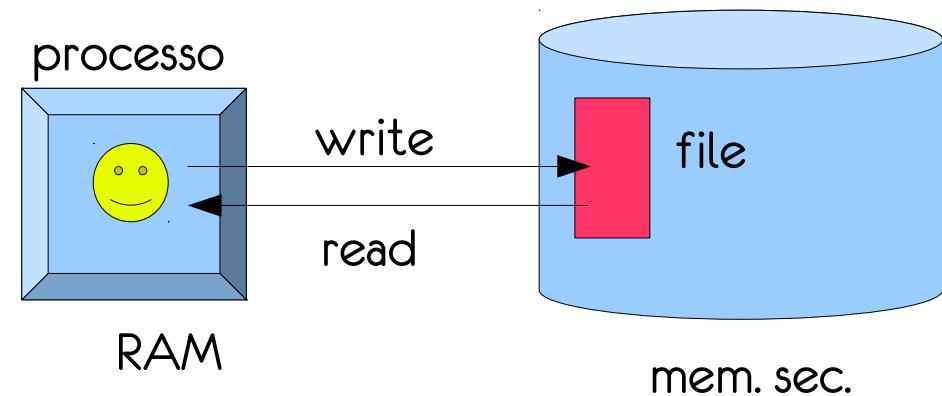


Soluzione 2 --- SISTEMARE

- Algoritmo LRU (tempo inizia a t0)
- 3 3 4 1 2 3 2 3 4 4 4 5 4 5 6 6 5 6 4 4 5 6 6 6 2 1
2 accessi a 3 iniziali
- 3 (t1) 4 (t2) 1 (t3) 2 (t4) -> 3 (t7) 4 (t10) 1 (t3) 2 (t6)
- 3 (t7) 4 (t10) 5 (t11) 2 (t6) -> 3 (t7) 4 (t12) 5 (t13) 2 (t6)
- 6 (t14) 4 (t12) 5 (t13) 2 (t6) -> 6 (t22) 4 (t18) 5 (t19) 2 (t23)
- 6 (t22) 1 (t24) 5 (t19) 2 (t23)
- 7 page fault

Memoria virtuale e file

- Fino ad ora abbiamo parlato di memoria virtuale facendo esclusivo riferimento all'esecuzione di processi
- Quando un processo **legge un file** non si può dire che il file faccia parte del suo spazio di indirizzi: non è parte del suo codice, non contiene le sue variabili ... si tratta di dati che vengo successivamente acquisiti per l'elaborazione
- l'accesso avviene tramite le system call **read()** e **write()**
- In questo contesto eseguire una lettura (scrittura) richiede un accesso alla memoria secondaria, su cui il file risiede

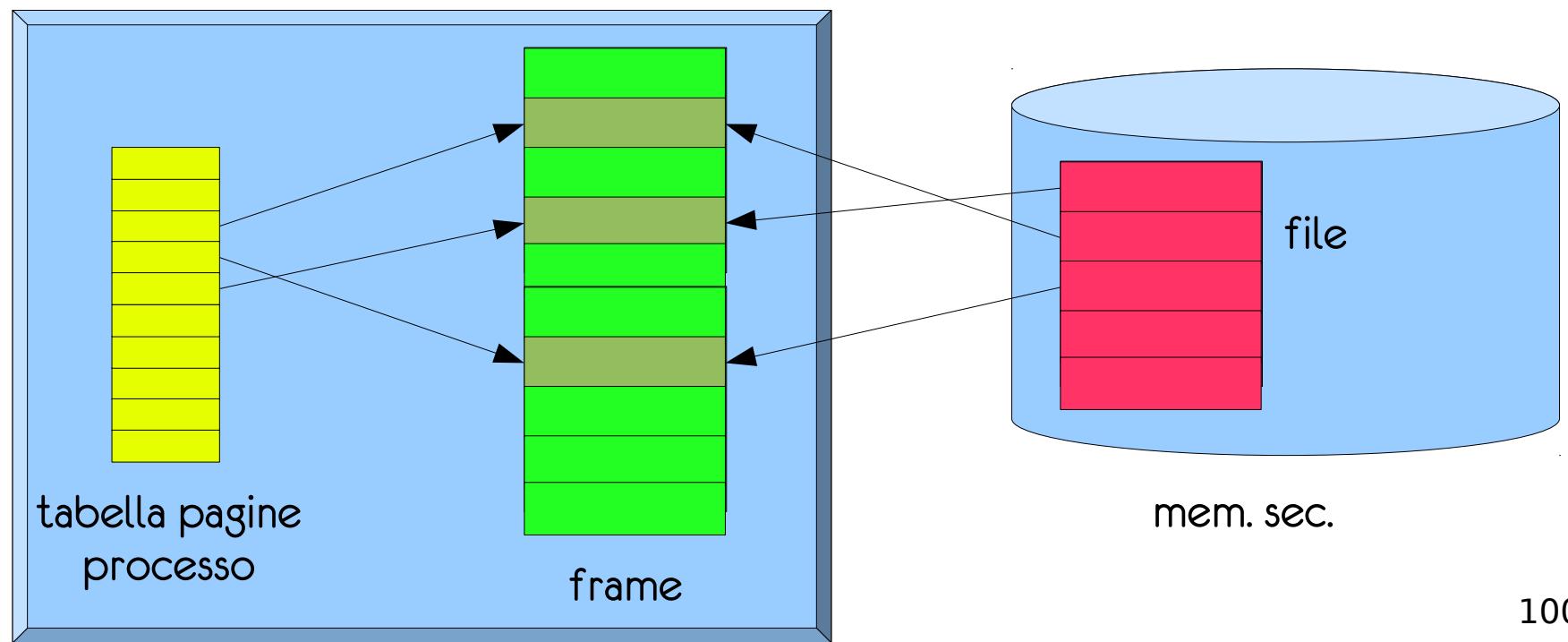


Memoria virtuale e file

- Fino ad ora abbiamo parlato di memoria virtuale facendo esclusivo riferimento all'esecuzione di processi
- Quando un processo legge un file non si può dire che il file faccia parte del suo spazio di indirizzi: non è parte del suo codice, non contiene le sue variabili ... si tratta di dati che vengo successivamente acquisiti per l'elaborazione
- l'accesso avviene tramite le system call read() e write()
- In questo contesto eseguire una lettura (scrittura) richiede un accesso alla memoria secondaria, su cui il file risiede
- di solito i processi non eseguono accessi sporadici ai file bensí sequenze di accessi, però effettuare ogni operazione direttamente sul disco comporta bassa efficienza: sarebbe meglio effettuare un tot di operazioni in RAM e riportare su backing le modifiche di tanto in tanto
- soluzione: **mappatura dei file**

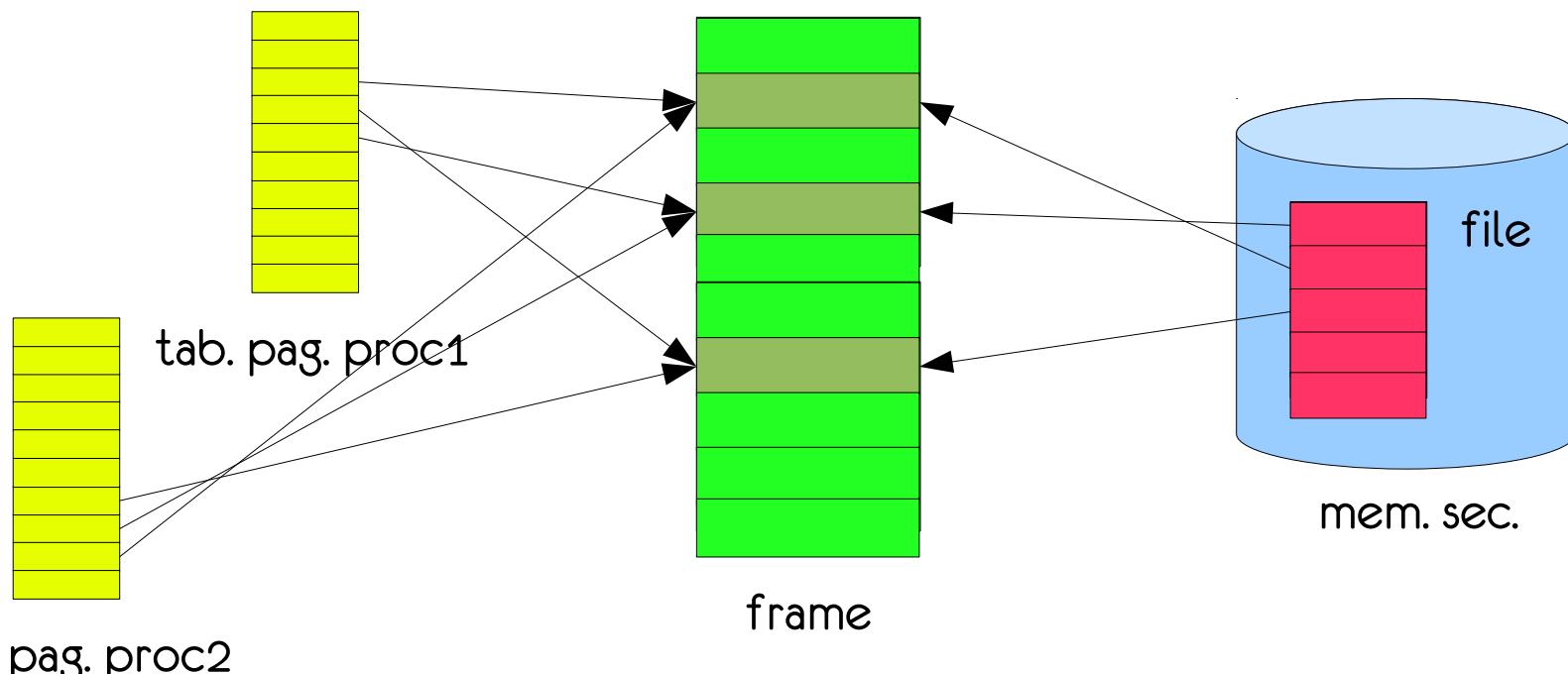
Mappatura dei file

- procedimento tramite il quale si associa una parte degli indirizzi virtuali di un processo a una parte di un file a cui questo ha accesso, sfruttando le strutture per implementare la memoria virtuale
- in pratica ad alcuni blocchi del file vengono associate alcune pagine del processo che vi accede

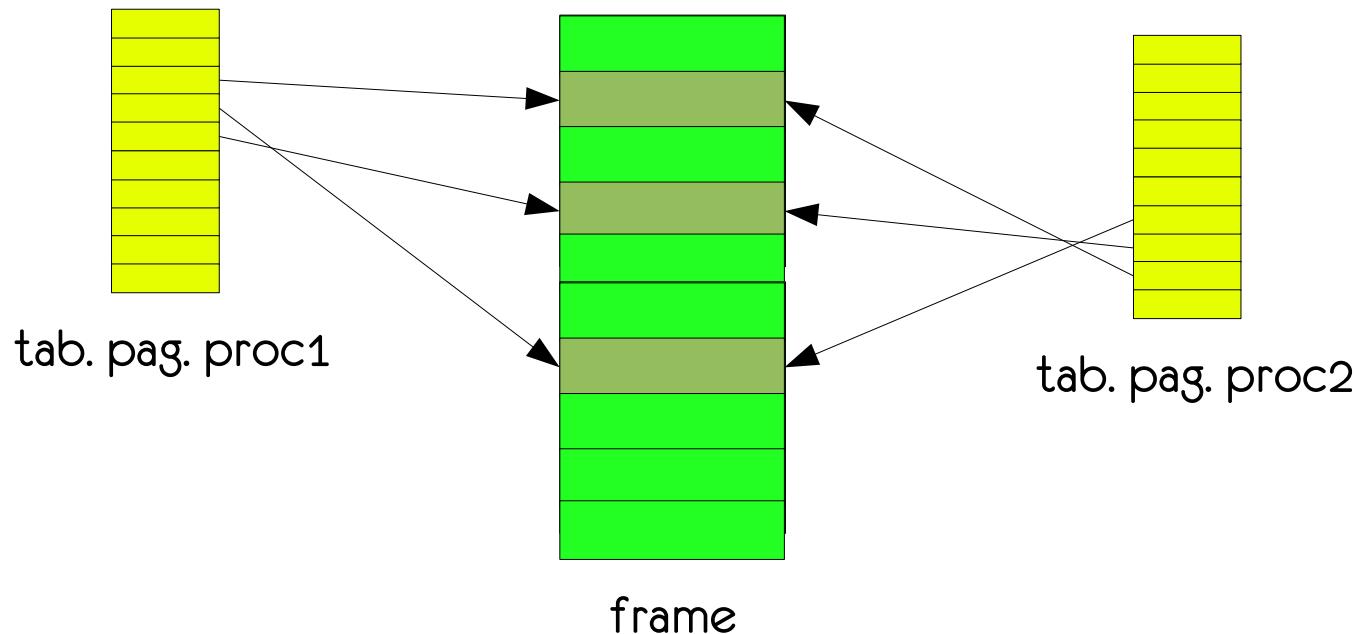


Effetti

- un processo svolgerà una serie di letture/scritture sulla copia del file che è nella RAM
- la minore richiesta di operazioni di I/O rende più rapide lettura/scrittura
- se un file è usato da più processi, basta caricarlo in RAM una volta sola e sfruttare il meccanismo per la condivisione delle pagine



Nota bene



Gli stessi blocchi caricati in frame di RAM sono accessibili da processi diversi attraverso indirizzi logici diversi: gli indirizzi logici sono infatti definiti sulla base delle pagine a cui si fa riferimento. Processi diversi useranno numeri di pagina diversi per accedere a uno stesso file

Da ram a disco

- quando si modifica la copia in RAM di un file occorre, prima o poi, riportare le modifiche effettuate anche sull'originale in memoria secondaria
- alcuni sistemi adottano come strategia un **controllo periodico**: se durante tale controllo una pagina risulta modificata, la si copia in memoria secondaria
- in altri casi si riversa la copia modificata sull'originale solo a **chiusura del file**
- talvolta è consentito all'utente di effettuare da programma operazioni di **flush** (riversamento) esplicite

Esempi

- **Windows NT, XP e 2000** utilizzano questo meccanismo per realizzare la memoria condivisa
 - memoria condivisa = file mappato in RAM e accessibile a un insieme di processi cooperanti
 - un processo apre il file,
 - esegue una system call per creare un suo mapping in memoria
 - poi esegue una system call per agganciare le pagine caricate al proprio spazio degli indirizzi
 - altri processi possono a questo punto agganciare lo stesso file caricato al proprio spazio degli indirizzi
- **Linux e Unix** utilizzano questo meccanismo per gestire solo i file, cioè non realizzano la memoria condivisa attraverso la mappatura di file

Mappatura I/O

- Oltre ai file, alcuni SO effettuano la mappatura in memoria principale dei registri dei controllori dei dispositivi di I/O
- esempi: controller video, porte seriali e parallele per modem/stampanti
- In questi casi l'interazione con i device avviene esclusivamente scrivendo/leggendo le porzioni di memoria principale associate ai device

Utente

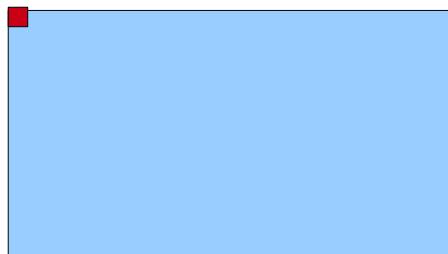
- i meccanismi di gestione della memoria sono trasparenti all'utente
- tuttavia, lo stile di programmazione può influenzare il numero di pagine usate e il numero di page fault di un programma. Avendo un'idea dei meccanismi sottostanti si possono scrivere programmi migliori, più efficienti
- **esempio 1**
 - consideriamo dei processi che fanno uso di un'area di memoria condivisa per scambiarsi dati. Tale area è gestita come un buffer circolare, la cui definizione richiede l'uso di 3 variabili:
 - `dato_t buffer[250]`
 - `int testa`
 - `int coda`

Esempio 1-a

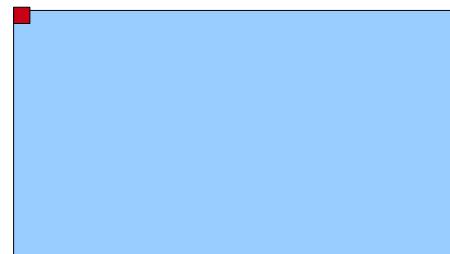
- **Soluzione dell'utente Sloppy**

- `alloc(a(testa)`
- `alloc(a(coda)`
- `alloc(a(buffer)`

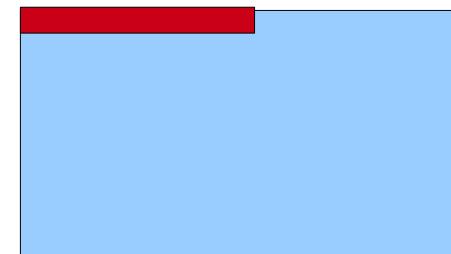
Il SO non ha modo di sapere che le tre strutture allocate fanno parte della stessa struttura, nella mente dell'utente. Non ha idea che gli insiemi di processi che si sincronizzeranno su di esse sono gli stessi.



pagina 1



pagina 2



pagina 3

Supponiamo che ogni pagina sia da 500K
avremo tre pagina quasi vuote

Esempio 1-b

- **Soluzione dell'utente Sly**

- `alloc(testa+coda+buffer)`

Il SO alloca una sola pagina contenente l'intera struttura: si risparmiano due pagine più tutto il tempo che occorre a un processo per richiedere ed ottenere di agganciarsi ad esse



pagina 1

La pagina non è molto piena perché l'oggetto condiviso è piccolo però lo spreco è diminuito molto

Esempio 2

- Supponiamo di dover inizializzare una matrice di interi così definita:

```
int mat[128][128];
```

- useremo quindi due indici i e j per scorrere la matrice e accedere alle sue celle, a cui assegneremo il valore 0

Esempio 2-a

- **Soluzione dell'utente Sloppy**

```
for (j=0; j<128; j++) // per ogni colonna  
    for (i=0; i<128; i++) // per ogni riga  
        mat[i][j] = 0;
```

- il codice è corretto
- il problema è che le matrici sono memorizzate per riga ...
- supponiamo che le pagine siano da 128 parole, ogni riga della matrice è contenuta in una pagina diversa (128 pagine): il ciclo for esterno può arrivare a produrre 128 page fault ad ogni iterazione ($128 \times 128 = 16.384$ p.f.)
- questo perché in generale il processo avrà disposizione un numero di frame ridotto

Esempio 2-b

- **Soluzione dell'utente Sly**

```
for (i=0; i<128; i++) // per ogni riga  
    for (j=0; j<128; j++) // per ogni colonna  
        mat[i][j] = 0;
```

- il codice è corretto
- abbiamo al più 128 page fault, uno per ogni riga

Allocazione dei frame

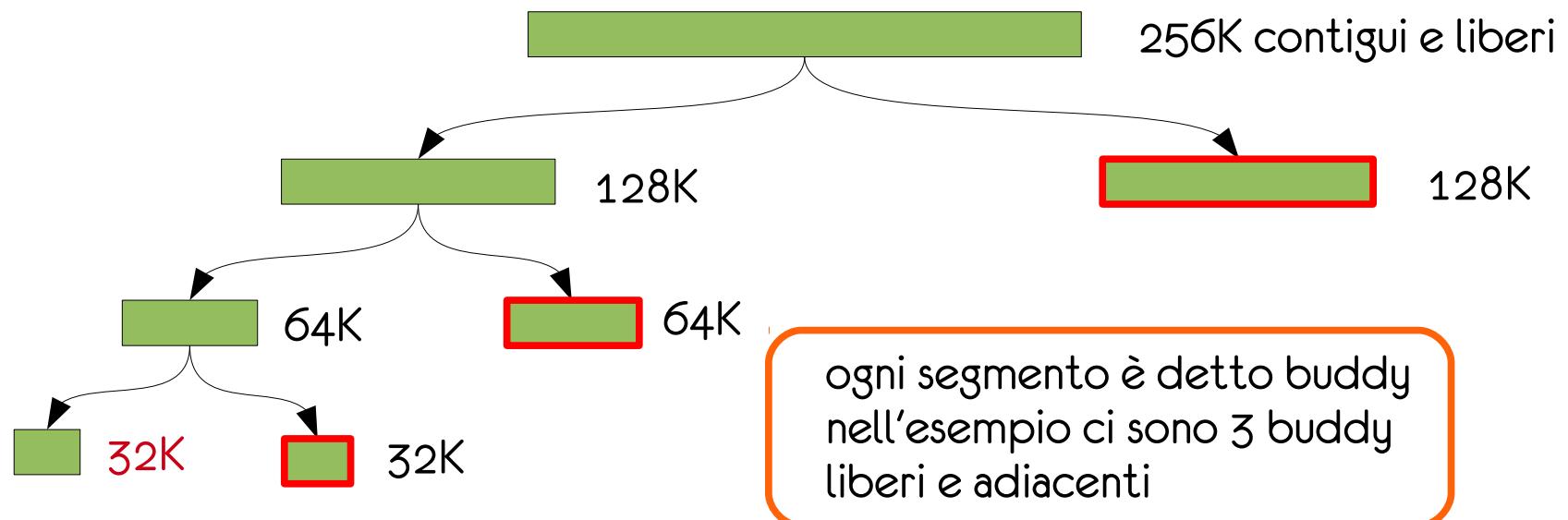
- per i processi utente
 - per i processi kernel
-

Allocazione per il kernel

- L'allocazione di memoria per i processi kernel sottosta a **meccanismi in parte dissimili** da quella vista per i processi utente
- Ciò è motivato da alcune **particolari caratteristiche dei processi kernel** e dalla necessità di rendere molto efficiente la loro esecuzione
 - necessità di strutturare dati di dimensioni variabili, spesso molto più **piccoli di una pagina** -> si desidera evitare lo spreco di una pagina per lo più vuota
 - alcuni dispositivi interagiscono direttamente con la RAM, se necessitano di una porzione di memoria maggiore di una pagina, **l'area richiesta deve essere sempre contigua**
- **codice e dati del kernel non sono sottoposti a paginazione**

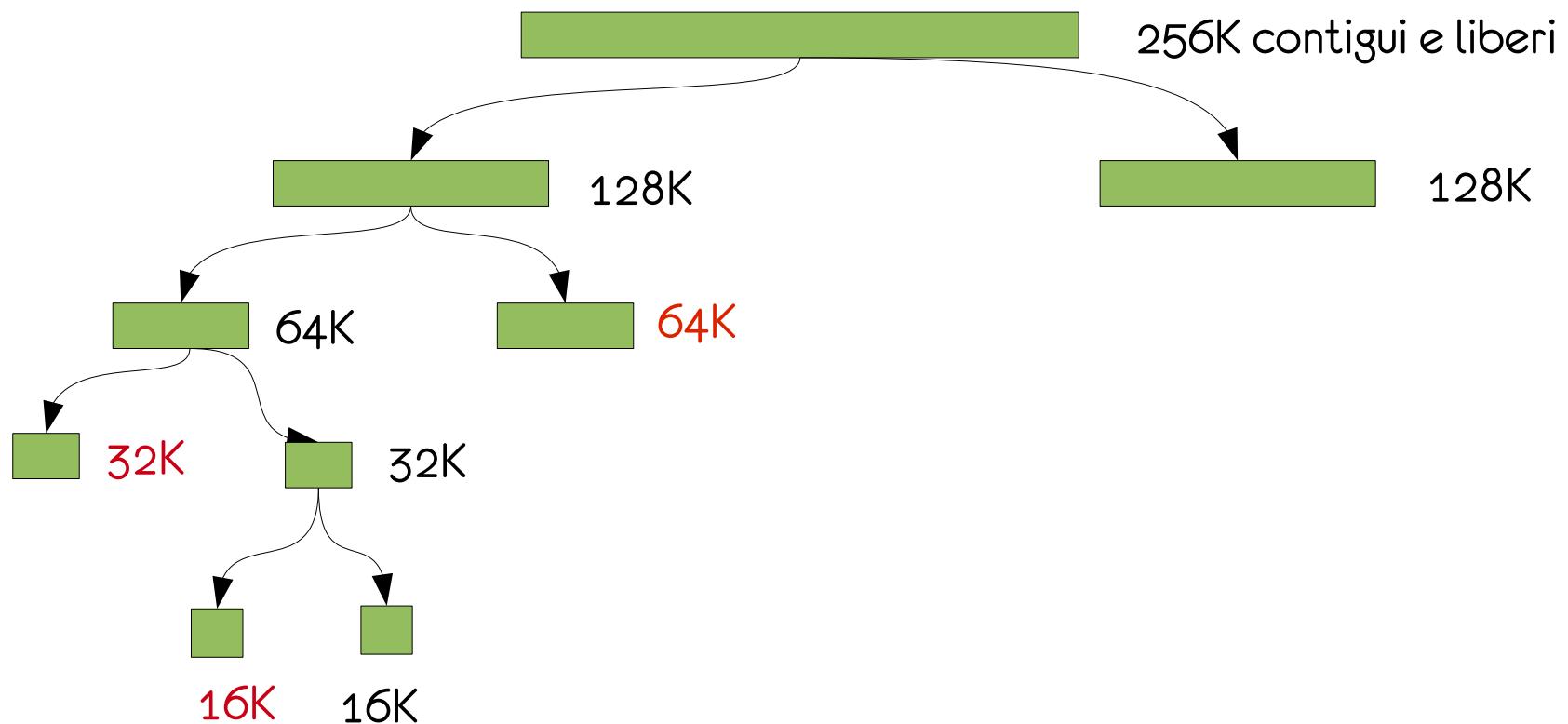
Sistema buddy

- meccanismo di allocazione della memoria, anche noto come **sistema gemellare**
- usa un segmento di pagine fisicamente contigue e **alloca la memoria in unità di dimensioni pari a potenze di 2** (4KB, 8KB, 16KB, ecc.), arrotondando per eccesso le richieste (se chiedo 6KB, l'allocatore ne restituisce 8)
- Esempio il SO richiede **30K**:



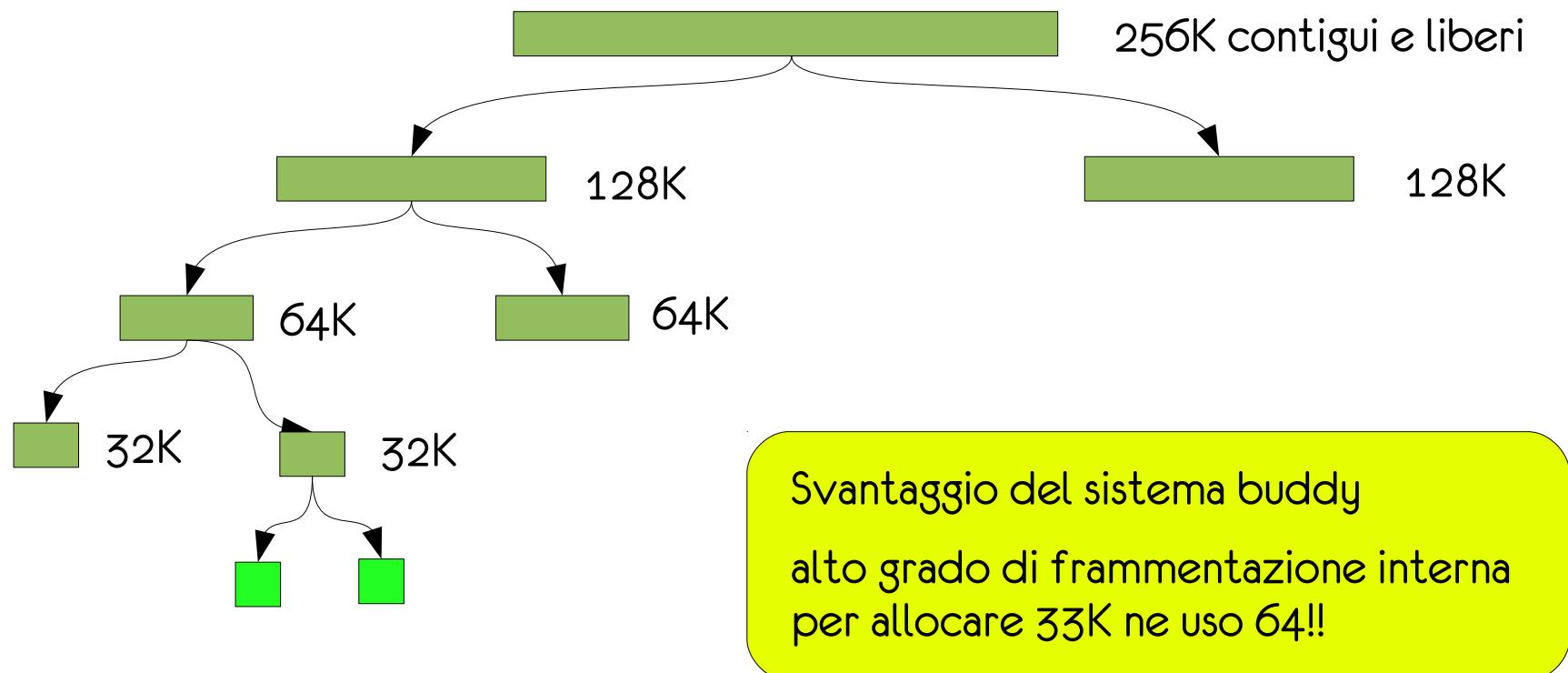
Sistema buddy

- Supponiamo che vengano richiesti 30K, 55K, 12K in sequenza



Sistema buddy

- coppie di gemelli liberi possono essere fuse in un unico buddy di dimensione doppia

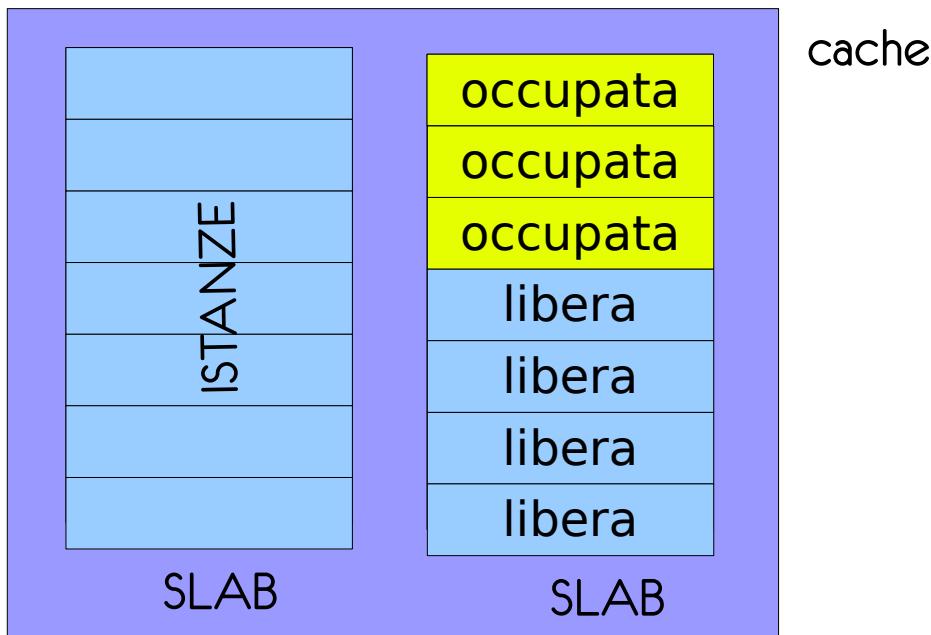


Allocazione a slab

- **slab** (lastra): sequenza di pagine fisicamente contigue
- **cache**: insieme di slab
- viene mantenuta **una cache per ogni tipo di strutture dati usate dal SO**, es.
 - una cache per i semafori
 - una cache per i PCB
 - una cache per i descrittori di file
 - ecc.

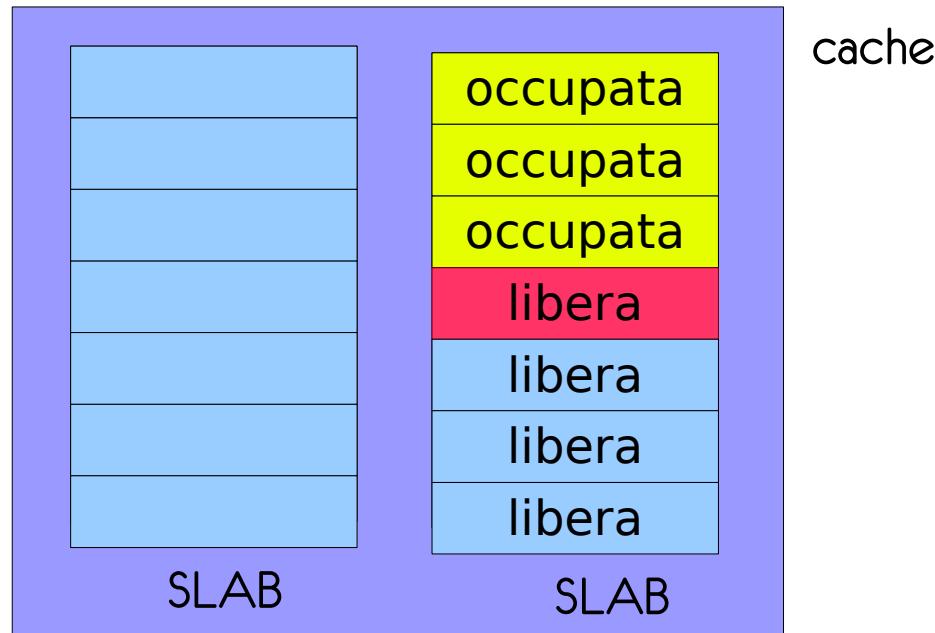
Allocazione a slab

- ogni cache contiene delle istanze del tipo di dato ad essa associato, il numero di istanze dipende dalla dimensione della cache e delle istanze stesse
 - es. posso avere 10 semafori
- ogni **istanza** può essere nello stato **libera** oppure **occupata**
 - es. tutti i semafori sono inizialmente liberi, cioè non utilizzati, quando un pool di processi richiede l'allocazione di un semaforo, un'istanza diventerà occupata



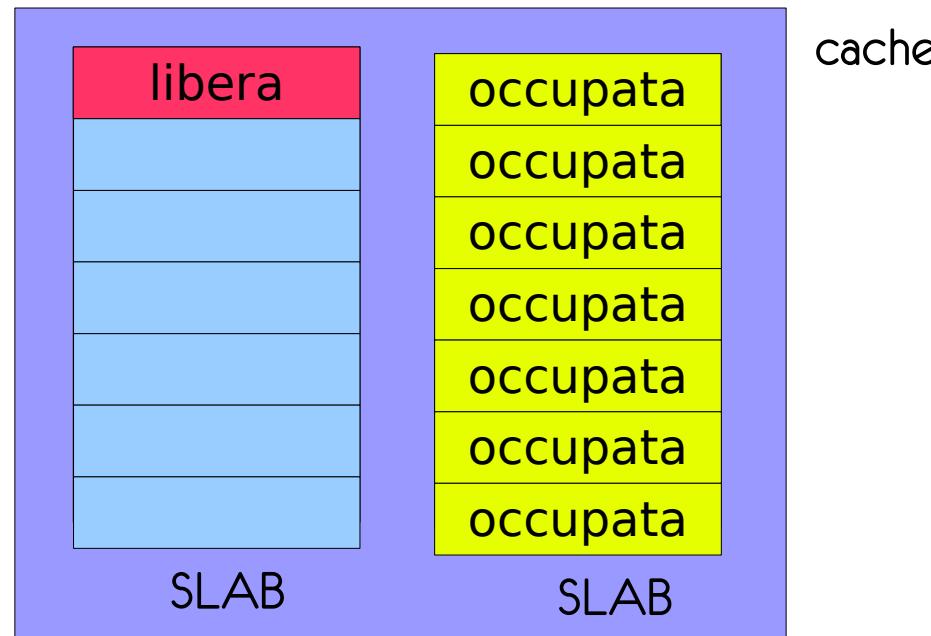
Allocazione

- quando viene richiesta una nuova istanza, il SO cerca la cache in questione
 - <1> all'interno di questa cerca una slab solo parzialmente occupata, e alloca un'istanza libera contenuta da quest'ultima



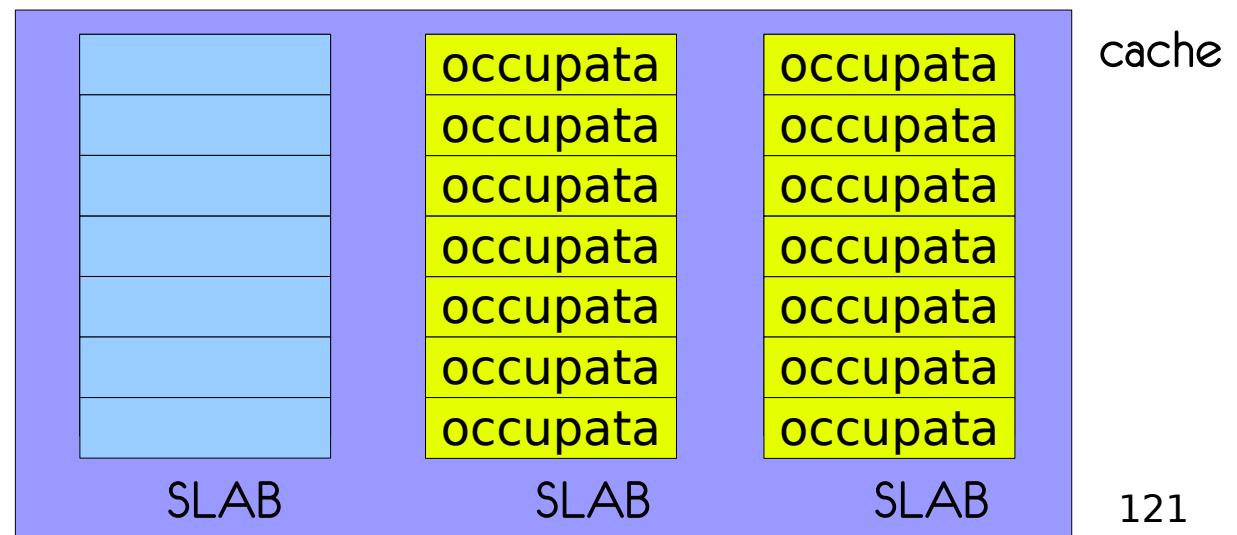
Allocazione

- quando viene richiesta una nuova istanza, il SO cerca la cache in questione
 - <1> all'interno di questa cerca una slab solo parzialmente occupata, e alloca un'istanza libera contenuta da quest'ultima
 - <2> se non ne trova, cerca una slab libera e alloca un'istanza da essa contenuta



Allocazione

- quando viene richiesta una nuova istanza, il SO cerca la cache in questione
 - <1> all'interno di questa cerca una slab solo parzialmente occupata, e alloca un'istanza libera contenuta da quest'ultima
 - <2> se non ne trova, cerca una slab libera e alloca un'istanza da essa contenuta
 - <3> se non ne trova crea una nuova slab a partire da un insieme di frame contigui e l'assegna alla cache poi ripete <2>



Allocazione

- quando viene richiesta una nuova istanza, il SO cerca la cache in questione
 - <1> all'interno di questa cerca una slab solo parzialmente occupata, e alloca un'istanza libera contenuta da quest'ultima
 - <2> se non ne trova, cerca una slab libera e alloca un'istanza da essa contenuta
 - <3> se non ne trova crea una nuova slab a partire da un insieme di frame contigui e l'assegna alla cache poi ripete <2>
- è una tecnica efficiente che elimina il problema della frammentazione interna perché una slab è suddivisa in spazi adatti a contenere un certo tipo di oggetti e gli oggetti sono allocati in toto o per nulla
- introdotta da Solaris, usata anche in Linux (che prima utilizzava il sistema buddy)

Top

- top è un comando presente in alcune versioni di Unix e Linux che consente di:
 - <1> vedere una gran quantità di informazioni riguardo l'uso di CPU e RAM
 - <2> eseguire comandi per la gestione dei processi
- si lancia da linea di comando
- vediamo insieme com'è strutturata l'interfaccia

Shell - Konsole <2>

Session Edit View Bookmarks Settings Help



Shell

```
top - 15:17:32 up 8 min, 1 user, load average: 0.56, 0.46, 0.25
Tasks: 104 total, 1 running, 102 sleeping, 0 stopped, 0 zombie
Cpu(s): 16.3% us, 5.6% sy, 0 %ni, 78.1% id, 0.0% wa, 0.0% hi, 0.0% si
Mem: 2067208k total, 630600k used, 1436608k free, 15932k buffers
Swap: 4192956k total, 0k used, 4192956k free, 327116k cached
```

STATISTICHE GENERALI**LINEA DI COMANDO**

PID	USER	PR	NI	VIRT	RES	SHR	S	%CPU	%MEM	TIME+	COMMAND
3986	baroglio	15	0	33448	17m	13m	S	11.6	0.9	0:05.35	kicker
3738	root	5	-10	310m	38m	4844	S	6.0	1.9	0:32.94	Xorg
3972	baroglio	17	0	31812	15m	12m	S	2.0	0.8	0:09.44	kded
3982	baroglio	15	0	29788	13m	10m	S	0.3	0.7	0:01.24	kwin
5973	baroglio	16	0	2204	1056	828	R	0.3	0.1	0:00.03	top
1	root	16	0	1564	504	436	S	0.0	0.0	0:00.61	init
2	root	34	19	0	0	0	S	0.0	0.0	0:00.00	ksoftirqd/0
3	root	RT	0	0	0	0	S	0.0	0.0	0:00.00	watchdog/0
4	root	10	-5							0:00.07	events/0
5	root	10	-5							0:00.00	khelper
6	root	10	-5	0	0	0	S	0.0	0.0	0:00.00	kthread
8	root	10	-5	0	0	0	S	0.0	0.0	0:00.16	kblockd/0
9	root	10	-5	0	0	0	S	0.0	0.0	0:00.00	kacpid
10	root	10	-5	0	0	0	S	0.0	0.0	0:00.03	kacpid-work-0
127	root	10	-5	0	0	0	S	0.0	0.0	0:00.00	khubd
180	root	20	0	0	0	0	S	0.0	0.0	0:00.00	pdflush
181	root	15	0	0	0	0	S	0.0	0.0	0:00.00	pdflush

INFORMAZIONI DI DETTAGLIO

STATISTICHE GENERALI

```
top - 15:17:32 up 8 min,  1 user,  load average: 0.56, 0.46, 0.25
Tasks: 104 total,   1 running, 103 sleeping,   0 stopped,   0 zombie
Cpu(s): 16.3% us,  5.6% sy,  0.0% ni, 74.1% id,  4.0% wa,  0.0% hi,  0.0% si
Mem: 2067208k total, 630600k used, 1436608k free,    15932k buffers
Swap: 4192956k total,        0k used, 4192956k free, 327116k cached
```

numero di task totale, e divisi in percentuale secondo il loro stato (in esecuzione, dormienti -ready-, bloccati e zombie -morti non morti-)

statistiche d'uso delle CPU: se l'elaboratore ne ha più d'una si possono vedere le statistiche CPU x CPU oppure come sommario generale (è il caso rappresentato) Sono mostrate le percentuali d'uso di diverse tipologie di task, fra queste "us" rappresenta i processi utente e "sy" i processi di sistema (kernel)

Mem e Swap contengono info sulla RAM e sull'area di swap

ID PROCESSO

UTENTE PROPRIETARIO

MEM. VIRTUALE COMPLESSIVA

STATO DEL TASK

TEMPO DI CPU

PID	USER	PR	NI	VIRT	RES	SHR	S	%CPU	%MEM	TIME+	COMMAND	NOME CMD
3986	baroglio	15	0	33448	17m	13m	S	11.6	0.9	0:05.35	kicker	
3738	root	5	-10	310m	38m	4844	S	6.0	1.9	0:32.94	Xorg	
3972	baroglio	17	0	31812	15m	12m	S	2.0	0.8	0:09.44	kded	
3982	baroglio	15	0	29788	13m	10m	S	0.3	0.7	0:01.24	kwin	
5973	baroglio	16	0	2204	1056	828	R	0.3	0.1	0:00.03	top	
1	root	16	0	1564	504	436	S	0.0	0.0	0:00.61	init	
2	root	PRIORITÀ	9				S	0.0	0.0	0:00.00	ksoftirqd/0	
3	root		RT	0			S	0.0	0.0	0:00.00	watchdog/0	
4	root						S	0.0	0.0	0:00.07	events/0	
5	root	10	-5		0	0	S	0.0	0.0	0:00.00	khelper	
6	root	10	-5		0	0	S	0.0	0.0	0:00.00	kthread	
8	root	10	-5		0	0	S	0.0	0.0	0:00.16	kblockd/0	
9	root	10	-5		0	0	S	0.0	0.0	0:00.00	kacpid	
10	root	10	-5		0	0	S	0.0	0.0	0:00.03	kacpid-work-0	
127	root	10	-5		0	0	S	0.0	0.0	0:00.00	khubd	
180	root	20	0		0	0	S	0.0	0.0	0:00.00	pdfflush	
181	root	15	0		0	0	S	0.0	0.0	0:00.00	pdfflush	

PRIORITÀ

RENICE

DIM. TASK RESIDENTE

% RAM USATA

% CPU USATA

DIM PORZIONE CONDIVISA



Current Fields: AEHIOQTWKNMbcdfgjplrsuvyzX for window 1:Def

Toggle fields via field letter, type any other key to return

- * A: PID = Process Id
- * E: USER = User Name
- * H: PR = Priority
- * I: NI = Nice value
- * O: VIRT = Virtual Image (kb) *
- * Q: RES = Resident size (kb)
- * T: SHR = Shared Mem size (kb)
- * W: S = Process Status
- * K: %CPU = CPU usage
- * N: %MEM = Memory usage (RES)
- * M: TIME+ = CPU Time, hundredths
- b: PPID = Parent Process Pid
- c: RUSER = Real user name
- d: UID = User Id
- f: GROUP = Group Name
- g: TTY = Controlling Tty
- j: P = Last used cpu (SMP)
- p: SWAP = Swapped size (kb)
- l: TIME = CPU Time
- r: CODE = Code size (kb)
- s: DATA = Data+Stack size (kb)

u: nFLT = Page Fault count

v: nDRT

y: WCHAN

z: Flags

X: COMMAND = Command name/line

Flags field:

0x00000001 PF_ALIGNWARN

0x00000002 PF_STARTING

0x00000004 PF_EXITTING

CAMPI CHE POSSONO ESSERE ATTIVATI/DISATTIVATI

0x00000200 PF_DUMPCORE

0x00000400 PF_SIGNALLED

0x00000800 PF_MEMALLOC

0x00002000 PF_FREE_PAGES (2.5)

0x00008000 debug flag (2.5)

fra le altre cose: (2.5)

dimensione della sezione testo 2.5)

dimensione della sezione dati 2.4)

Shell - Konsole <2>

Session Edit View Bookmarks Settings Help

Shell

```
top - 15:50:29 up 41 min, 1 user, load average: 0.75, 0.30, 0.17
Tasks: 106 total, 1 running, 105 sleeping, 0 stopped, 0 zombie
Cpu(s): 9.6% us, 3.0% sy, 0.0% ni, 87.4% id, 0.0% wa, 0.0% hi, 0.0% si
Mem: 2067208k total, 667504k used, 1399704k free, 18012k buffers
Swap: 4192956k total, 0k used, 4192956k free, 343296k cached
```

PID	VIRT	SHR	%CPU	%MEM	PPID	CODE	DATA	nFLT	COMMAND
3738	313m	4852	6.0	2.0	3702	1476	46m	73	Xorg
3972	31812	12m	1.7	0.8		1	40	2508	23 kded
3986	33564	13m	1.3	0.9		1	40	3280	38 kicker
4076	30124		ID PROC	0.7	3965	40	2116	1	konsole
13142	28860		PADRE	0.7	3965	96	1540	0	ksnapshot
3974	2872	898	0.5	0.1		1	84	780	1 g #PAGE FAULT
3982	29788	10m	0.3	0.7	3965	40	2496	45	kwin
4000	25004	7492	0.3	0.4		40	1	ss	
12961	2204	836	0.3	0.1	4079	52			
1	1564	436	0.0	DIM SEZIONE TESTO		28	244	13	init
2	0	0	0.0			0	0	0	ksoftirqd/0
3	0	0	0.0			0	0	0	watchdog/0
4	0	0	0.0			1	0	0	events/0
5	0	0	0.0			1	0	0	khelper
6	0	0	0.0			1	0	0	kthread
8	0	0	0.0		6	0	0	0	kblockd/0
9	0	0	0.0		6	0	0	0	kacpid

VMSTAT

vmstat è un comando che consente di ottenere informazioni sulla memoria virtuale sugli eventi sui processori e su diversi tipi di attività della CPU

vmstat -f dice quante fork sono state eseguite a partire dall'avvio della macchina

vmstat -m mostra informazioni sulle slab

vmstat -s riporta una tabella con il conteggio di quanti eventi di diversi tipi si sono verificati + alcune info generali

Sinossi di vmstat

baroglio 514>> vmstat --help

usage: vmstat [-V] [-n] [delay [count]]

-V prints version.

-n causes the headers not to be reprinted regularly.

-a print inactive/active page stats.

-d prints disk statistics

-D prints disk table

-p prints disk partition statistics

-s prints vm table

-m prints slabinfo

-S unit size

delay is the delay between updates in seconds.

unit size k:1000 K:1024 m:1000000 M:1048576 (default is K)

count is the number of updates.

baroglio 515>>

vmstat -f

Il risultato mostrato a video è semplicissimo, es:

```
baroglio 503>> vmstat -f  
          14991 forks  
baroglio 504>>
```

```
baroglio 516>> vmstat -s  
2067208 total memory  
672768 used memory  
378732 active memory  
200000 inactive memory  
1394440 free memory  
21764 buffer memory  
348156 swap cache  
4192956 total swap  
0 used swap  
4192956 free swap  
71678 non-nice user cpu ticks  
60 nice user cpu ticks  
10587 system cpu ticks  
494691 idle cpu ticks  
5877 IO-wait cpu ticks  
294 IRQ cpu ticks  
259 softirq cpu ticks  
330014 pages paged in  
92188 pages paged out  
0 pages swapped in  
0 pages swapped out  
1780101 interrupts  
4393343 CPU context switches  
1172671730 boot time  
25266 forks
```

vmstat -s

vmstat -m

baroglio 504>> vmstat -m

Cache	Num	Total	Size	Pages
ip_conntrack_expect	0	0	92	42
ip_conntrack	7	17	232	17
wrap_mdl	0	0	44	84
dm_tio	0	0	16	203
dm_io	0	0	16	203
uhci_urb_priv	1	92	40	92
ip_fib_alias	9	113	32	113
ip_fib_hash	9	113	32	113
clip_arp_cache	0	0	192	20
UNIX	250	250	384	10
ip_mrt_cache	0	0	128	30
tcp_bind_bucket	10	203	16	203
inet_peer_cache			

Mostra le cache di RAM usate dal SO, per ogni cache sono riportate informazioni sulla sua struttura e sul suo uso

Ci sono 144 cache su questo portatile

vmstat -m

baroglio 504>> vmstat -m

Cache	Num	Total	Size	Pages
ip_conntrack_expect	0	0	92	42
ip_conntrack	7	17	232	17
wrap_mdl	0	0	44	84
dm_tio				
dm_io				
uhci_urb_priv				
ip_fib_alias				
ip_fib_hash				
clip_arp_cache				
UNIX				
ip_mrt_cache	0	0	128	30
tcp_bind_bucket	10	203	16	203
inet_peer_cache			

Descrittori dei campi

Cache	nome della cache
Num	numero di oggetti attualmente attivi
Total	numero totale di oggetti disponibili
Size	dimensione del singolo oggetto
Pages	numero di pagine

Ci sono 144 cache su questo portatile

Shell - Konsole <2>

Session Edit View Bookmarks Settings Help



Shell

slabtop

Active / Total Objects (% used) : 114783 / 119200 (96.3%)
Active / Total Slabs (% used) : 5889 / 5890 (100.0%)
Active / Total Caches (% used) : 81 / 137 (59.1%)
Active / Total Size (% used) : 23365.37K / 23680.78K (98.7%)
Minimum / Average / Maximum Object : 0.01K / 0.20K / 128.00K

OBJS	ACTIVE	USE	OBJ SIZE	SLABS	OBJ/SLAB	CACHE SIZE	NAME
44341	44341	100%	0.13K	1529	29	6116K	dentry_cache
14904	13950	93%	0.05K	207	72	828K	buffer_head
10656	10630	99%	0.46K	1332	8	5328K	ext3_inode_cache
8778	8742	99%	0.27K	627	14	2508K	radix_tree_node
6380	6260	98%	0.09K	145	44	580K	vm_area_struct
6072	6048	99%	0.04K	66	92	264K	sysfs_dir_cache
4181	4181	100%	0.03K	37	113	148K	size-32
3288	3288	100%	0.32K	274	12	1096K	inode_cache
2700	2540	94%	0.19K	135	20	540K	filp

File system

capitolo 10 del libro (VII ed.)

Introduzione

- Programmi e dati sono conservati in memoria secondaria: tipicamente un insieme di dischi. Gli utenti (voi, io) vedono la memoria secondaria come organizzata in **file** e **directory**. Qualcuno ha sentito il termine “**file system**”
- dopo alcuni anni di uso inconsapevole di questi termini un utente potrebbe cominciare a porsi alcune domande ...



File

- Il termine **file** rappresenta un **concetto logico** (astratto)
- Un file è un **insieme di informazioni correlate**, definito da un creatore (a volte tramite l'ausilio di un programma, es. editor), a cui è associato un **nome**
- Dal punto di vista dell'utente i file sono gli **elementi di base in cui è organizzata la memoria**, sono indifferenziati
- Però esistono *file e file* ...



dvi



testo



shell



crittato



html



binario



ascii

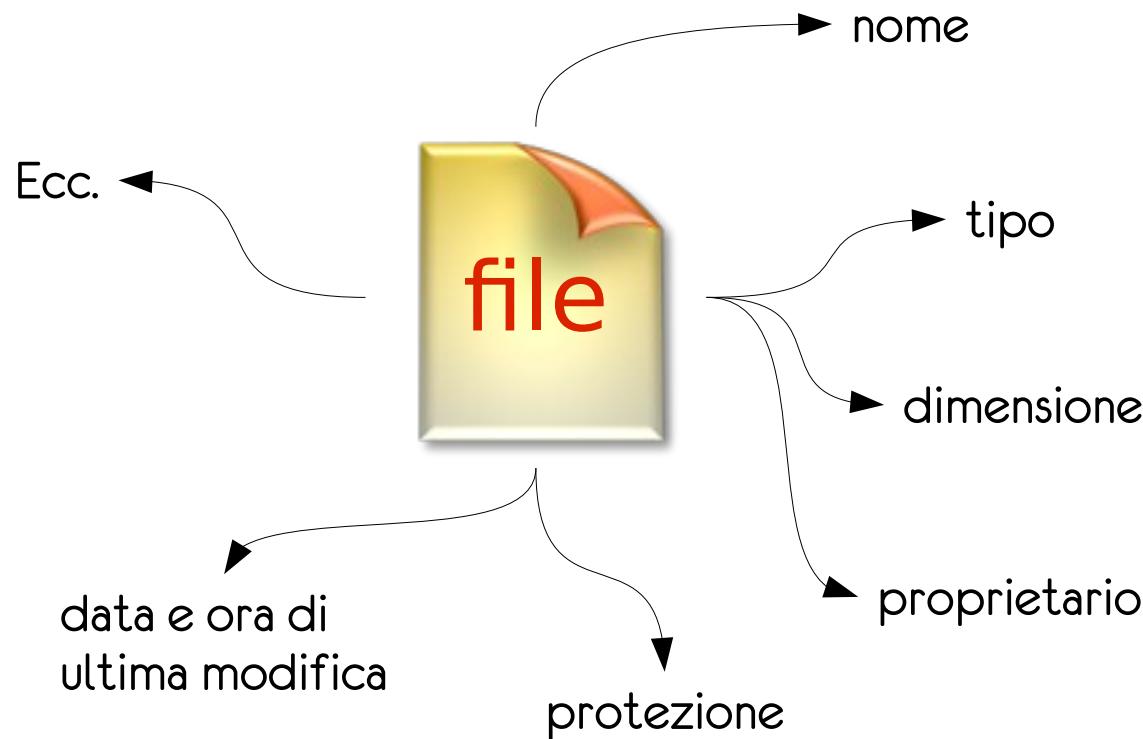
Metadati

- il fatto che esistano tipi di file, con estensioni e icone differenti implica che un file non sia soltanto un insieme di dati correlati
- un **file** è **in parte contenuto e in parte oggetto** descritto attraverso un insieme di **caratteristiche** che lo riguardano. Tali caratteristiche sono anche dette **attributi** o **metadati** (*dati inerenti i dati*)

Esempio

- provate ad applicare, in linux, il comando **file** a un file qualsiasi (es. *clock.png*), otterrete una risposta tipo:
- clock.png*: PNG image data, 128 × 128, 8-bit/color RGBA, non-interlaced
- provate a **rinominare il file** in modo da ingannare il comando, per es. rinominiamolo semplicemente **clock**:
- clock*: PNG image data, 128 × 128, 8-bit/color RGBA, non-interlaced
- queste informazioni devono essere associate al file stesso indipendentemente dalla sua **estensione**, non sono dei contenuti, sono metadati

Metadati



Posso associare programmi di gestione a tipi di file, es. acroread o xpdf ai file di tipo pdf: cliccando sull'icona che rappresenta un file pdf viene avviato dal SO il gestore di default.

Basta mantenere una tabella di associazioni <tipo, applicativo>

Fra i diversi modi per associare a un file il suo tipo, particolarmente rilevante l'uso di "magic number"

Altri SO si affidano alle estensioni dei file

Magic Number

MAGIC NUMBER: un codice conservato all'interno del file stesso. Unix adotta questo meccanismo

ESEMPI:

- **Java class file:**
CAFEBABE (esadecimale)
- **Linux script:**
"shebang" (#!, 23 21) seguito dal path dell'interprete necessario
- **File postscript:**
%!
- **File PDF:**
%PDF
- **File GIF:**
codice ASCII per "GIF89a", cioè 47 49 46 38 39 61

File di tipo diverso

- perché ho bisogno di diversi tipi di file? Perché tutte queste diverse estensioni?
- Ogni tipo/estensione corrisponde a un particolare **formato** con cui i dati sono organizzati, es.



file immagine

matrice di pixel colorati

però posso avere png, gif,
tiff, eps, raw, svg, jpg, xcf, ...

ACROREAD
KGHOSTVIEW
si aspettano i dati in questo formato

```
%%Page: 1 1
% Translate for offset
14.173228346456694 14.173228346456694 translate
% Translate to begin of first scanline
0 63.97795275590552 translate
63.97795275590552 -63.97795275590552 scale
% Image geometry
64 64 8
% Transformation matrix
[ 64 0 0 64 0 0 ]
% Strings to hold RGB-samples per scanline
/rstr 64 string def
/gstr 64 string def
/bstr 64 string def
{currentfile /ASCII85Decode filter /
  /I85Decode filter /RunLengthDecode filter gstr readstring pop}
  /I85Decode filter /RunLengthDecode filter bstr readstring pop}
```

FORMATO EPS

12861 ASCII Bytes

l\mn%en\srqo\Tuu~>
|Mh%en'8Qfo\TUu~>

File di tipo diverso

- perché non posso usare Microsoft Word (o OpenOffice o un editor di pagine HTML) per scrivere un programma C?
- perché se lo faccio il compilatore si arrabbia?

```
void push(lista *p_L, double x)
{
    printf("push %.2f \n",x);
    //stack vuoto
    if(p_L == NULL)
    {
        *p_L = (lista) malloc (sizeof(struct
            if(p_L == NULL) return;
            (*p_L) -> info = x;
    }
else ...
```

Grassetto, colori, indentazione ecc. sono esplicitamente rappresentati da questi programmi, quindi il contenuto del file è il codice + tutta l'informazione relativa alla sua rappresentazione

File di tipo diverso

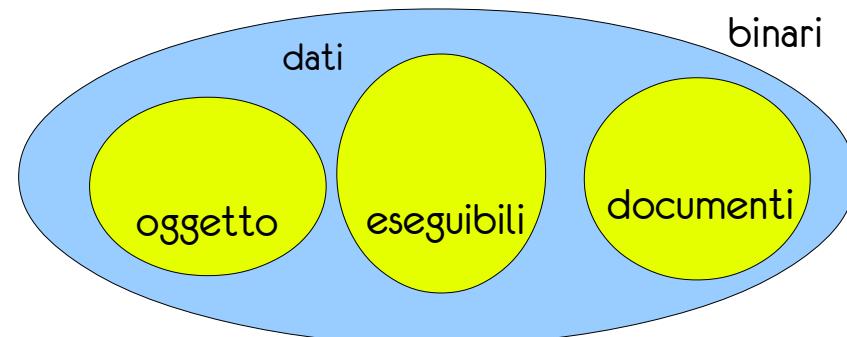
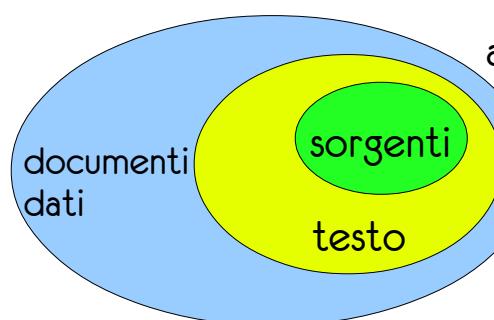
- perché non posso usare Microsoft Word (o OpenOffice o un editor di pagine html) per scrivere un programma C?
- perché se lo faccio il compilatore si arrabbia?

```
<font color="#0000ff"> //push sullo stack</font>
<font color="#298a52"><b>void</b></font> push(lista *p_L, <font
color="#298a52"><b>double</b></font> x)
{
    printf(<font color="#ff00ff">&quo
    <font color="#6b59ce">% .2f</font>
    <font color="#ff00ff"> </font><font color="#6b59ce"> </font>
    <font color="#ff00ff">&quot;</font>,x);
    <font color="#0000ff"> //stack vuoto</font>
    <font color="#a52829"><b>if</b></font>
    (p_L == <font color="#ff00ff">NULL</font>
    {
        *p_L = (lista) malloc (<font color="#a52829"><b>sizeof</b>
        </font>(<font color="#298a52"><b>struct</b></font> nodo)); ...
```

in HTML lo stesso codice sarebbe
rappresentato in questo modo

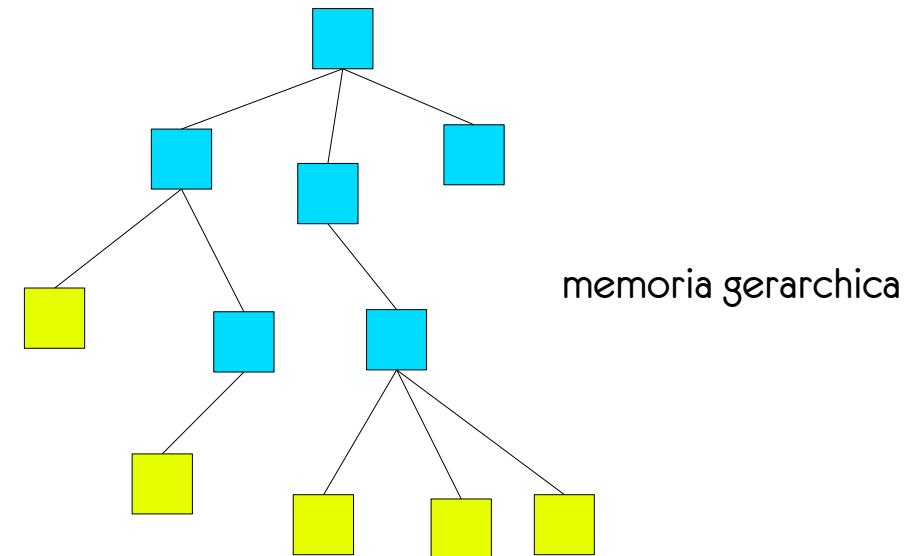
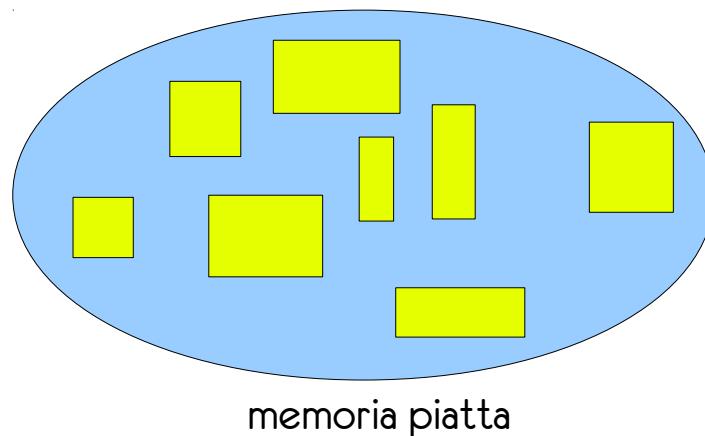
Tipi di file

- I file possono essere caratterizzati anche da **tipologie più generali** rispetto alla specifica applicazione che li gestirà, in particolare:
 - file alfanumerici**, contengono sequenze di caratteri. Fra questi:
 - file di testo: sequenze di caratteri divise in righe
 - file sorgenti: sequenze di procedure e funzioni strutturate
 - file binari**, contengono byte organizzati secondo una struttura precisa, non sono visualizzabili come testo. Fra questi:
 - file oggetto: sequenze di byte comprensibili per il linker
 - file eseguibili: sequenze di byte comprensibili per il loader
 - altri tipi di documenti: es. file compressi, documenti word, immagini png, ecc.



File system

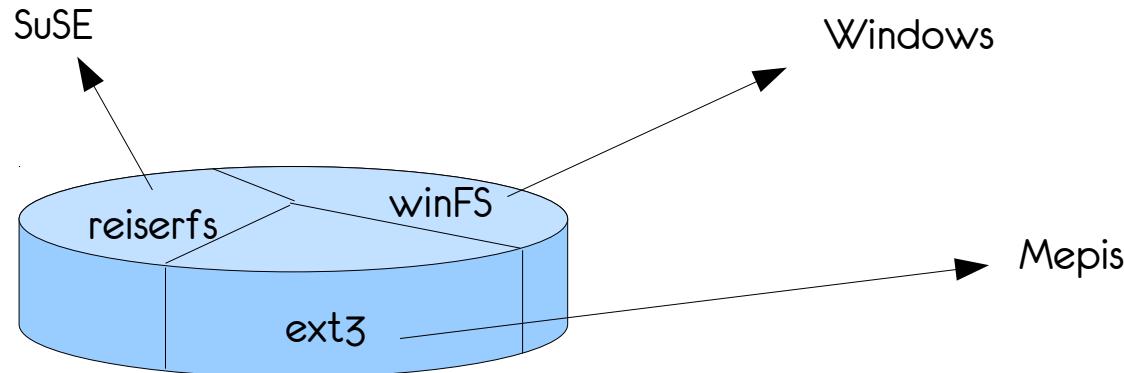
- I file possono essere organizzati in vari modi
- La **struttura logica** secondo la quale sono organizzati i file è detta **file system**
- Due visioni classiche:
 - organizzazione piatta**: la memoria secondaria è strutturata a livello logico in un insieme di file piatto, non posso avere due file con lo stesso nome
 - organizzazione gerarchica**: la memoria secondaria è organizzata ad albero, i nodi intermedi fungono da contenitori e sono detti **directory**



Tipi di file system

- Un **file system** è mantenuto attraverso l'utilizzo di **strutture dati interne**, preposte al mantenimento dei **metadati** (dati inerenti i file e dati inerenti le directory)
- La scelta dei metadati da gestire, la scelta di particolari strutture atte a implementare tali metadati e l'organizzazione in generale della struttura in cui i file sono organizzati porta alla realizzazione di uno specifico file system
- Esistono molti tipi di file system:
 - ext2, ext3 (Tweedie, 1999), ext4 gestisce volumi fino a 1 exabyte (exa: 10^{18} byte, trilioni di byte)
 - ReiserFS, Reiser4 (Namesys 2004)
 - FAT, WinFS
 - Amiga Fast File System
 - ADFS (Acorn computers)
 - file system per flash memory, es: JFFS, YAFFS, smxFFS

Disco e file system



La memoria secondaria può essere suddivisa in partizioni ciascuna delle quali è un diverso file system

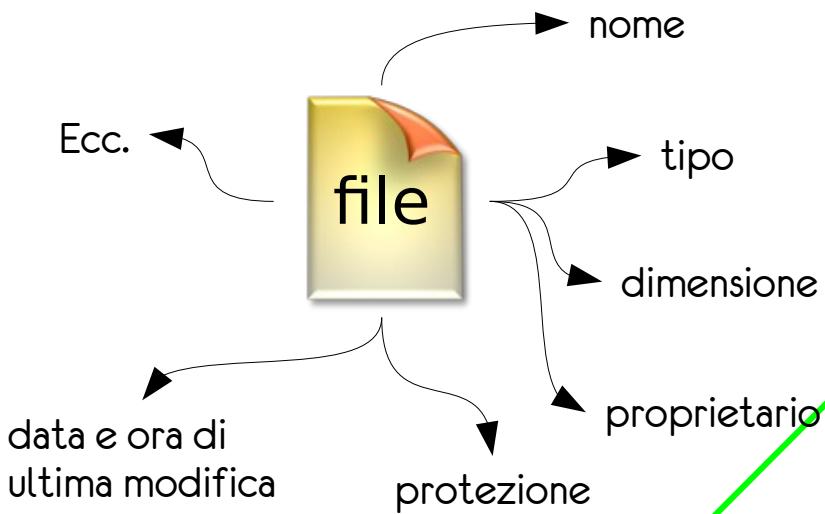
I file system delle varie partizioni possono essere di tipo differente

Questa possibilità è molto utile quando si desidera installare su di uno stesso computer più di un SO

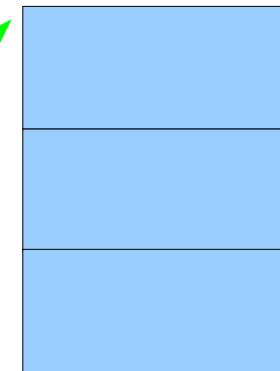
Studiare un file system

- Livello logico (cap. 10):
 - file e directory
 - organizzazione delle directory
 - operazioni consentite
 - protezione
- Implementazione (cap. 11):
 - implementazione delle directory
 - rappresentazione dei file
 - file system virtuale
 - allocazione e gestione dello spazio libero
 - ripristino
 - mounting, file system di rete

File



File come contenuto



Blocchi che l'utente immagina come sequenziali

File come oggetto gestito dal FS: descrittore



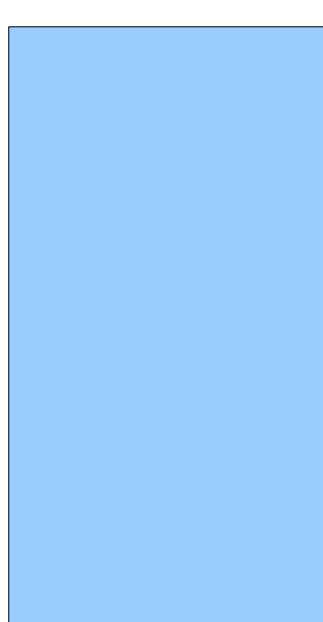
è l'analogo di un PCB (descrittore di processo)

File: operazioni

- Un file è una struttura dati astratta utilizzabile solo attraverso operazioni predefinite
- **creazione**: occorre trovare spazio sufficiente nel disco, occorre registrare i metadati relativi al file in un'apposita struttura (**file system**)
- **scrittura**: occorre identificare la posizione occupata dal file, occorre trovare la posizione in cui scrivere, occorre aggiornare i metadati
- **lettura**: occorre identificare la posizione occupata dal file, occorre trovare la posizione da cui leggere, occorre aggiornare i metadati
- **riposizionamento**: si assegna un nuovo valore a un puntatore ai contenuti del file; non è richiesto alcun accesso effettivo in lettura o scrittura, quindi si lavora solo a livello di meta-dati
- **cancellazione**: occorre individuare il file, aggiornare le strutture di sistema che mantengono informazioni sulla memoria libera, aggiornare le strutture di sistema che mantengono l'organizzazione dei file (**file system**)
- **troncamento**: vengono cancellati i contenuti di un file ma vengono mantenuti i metadati

Esempio

- Consideriamo, per es., un processo che debba leggere un intero file ed elaborare i dati così acquisiti



File



P

Lettura1:
cerca il file
ottieni puntatore
esegui lettura

Lettura2:
cerca il file
ottieni puntatore
esegui lettura

Lettura3:
cerca il file
ottieni puntatore
esegui lettura

Più efficiente

Apri il file:
cerca il descrittore
del file
ottieni puntatore

Lettura1:
esegui lettura

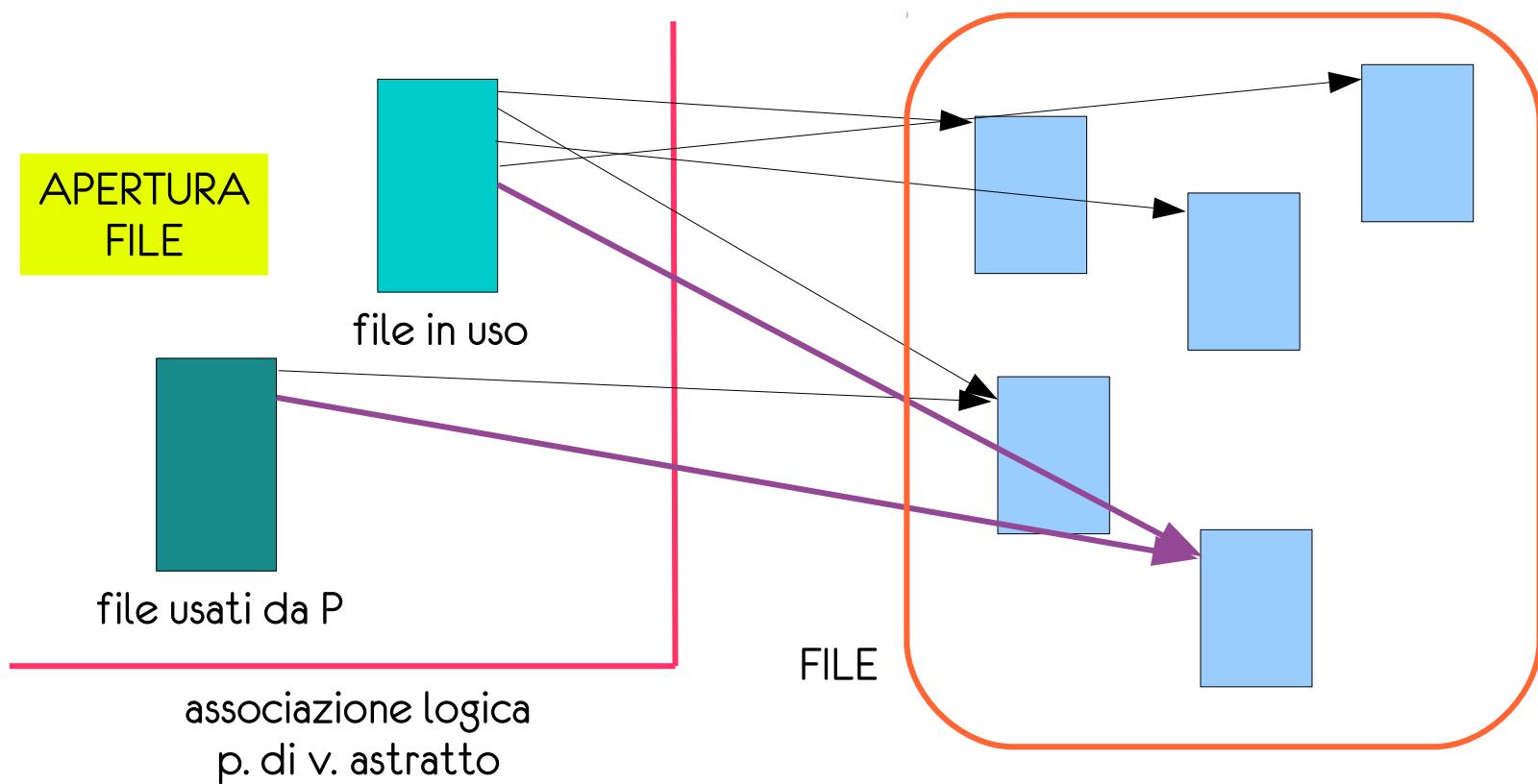
Lettura2:
esegui lettura

Lettura3:
esegui lettura

Chiudi il file

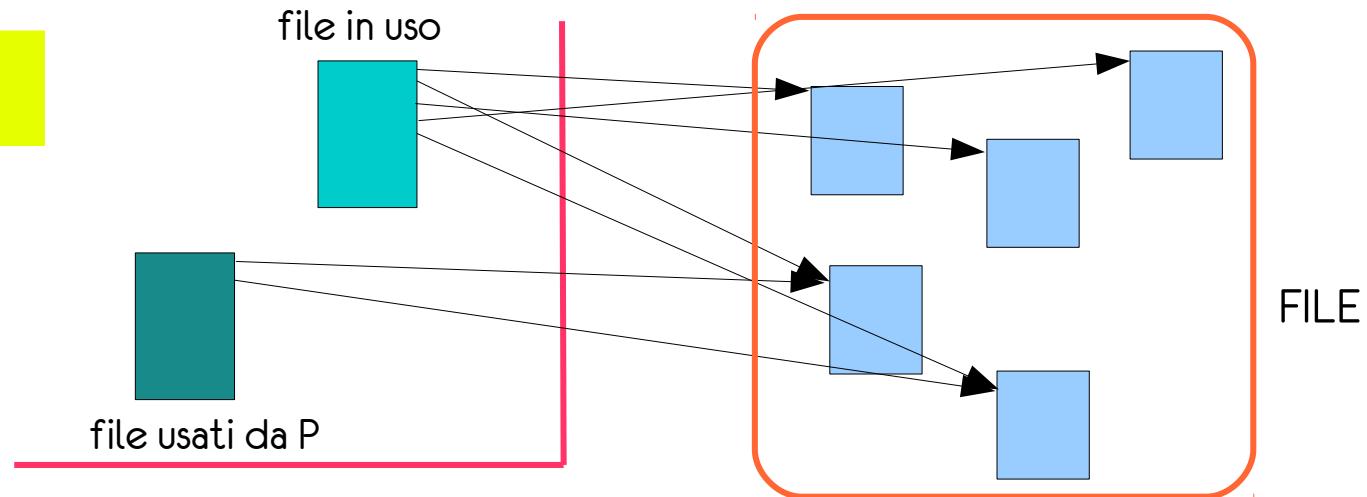
Apertura e chiusura di un file

- L'operazione di **apertura di un file** consente di ottenere un **handle** del file, cioè un riferimento che consente di operare effettivamente sul file stesso
- Questa operazione modifica alcune strutture gestite dal SO, che tracciano quali file sono in uso, in generale, e quali file sono in uso da ogni specifico processo

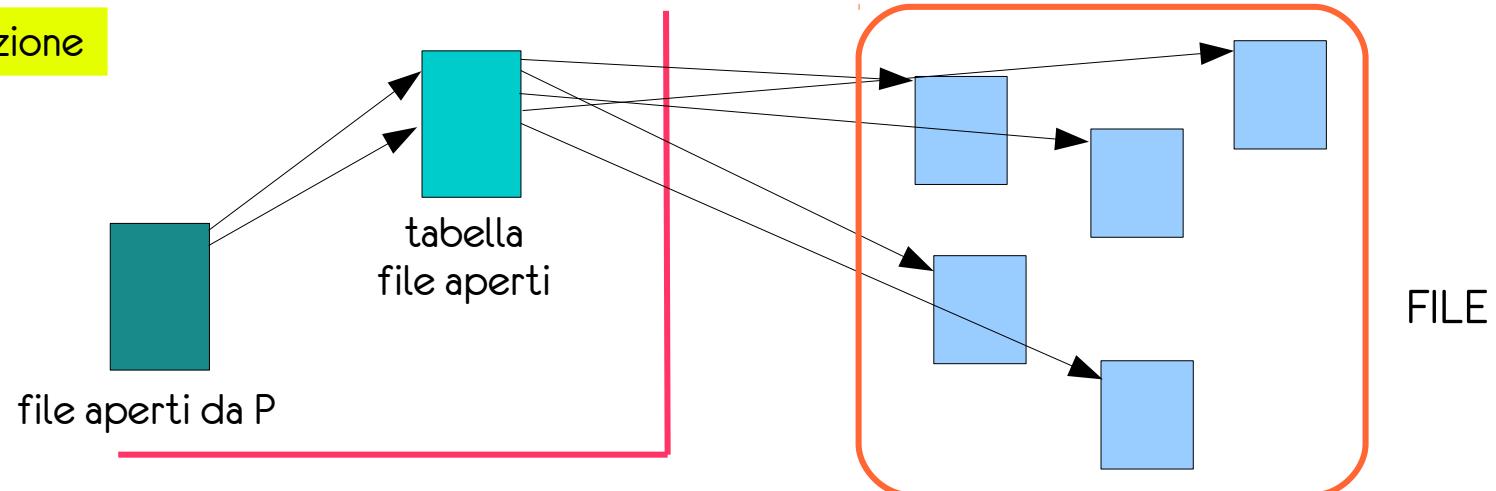


Apertura e chiusura di un file

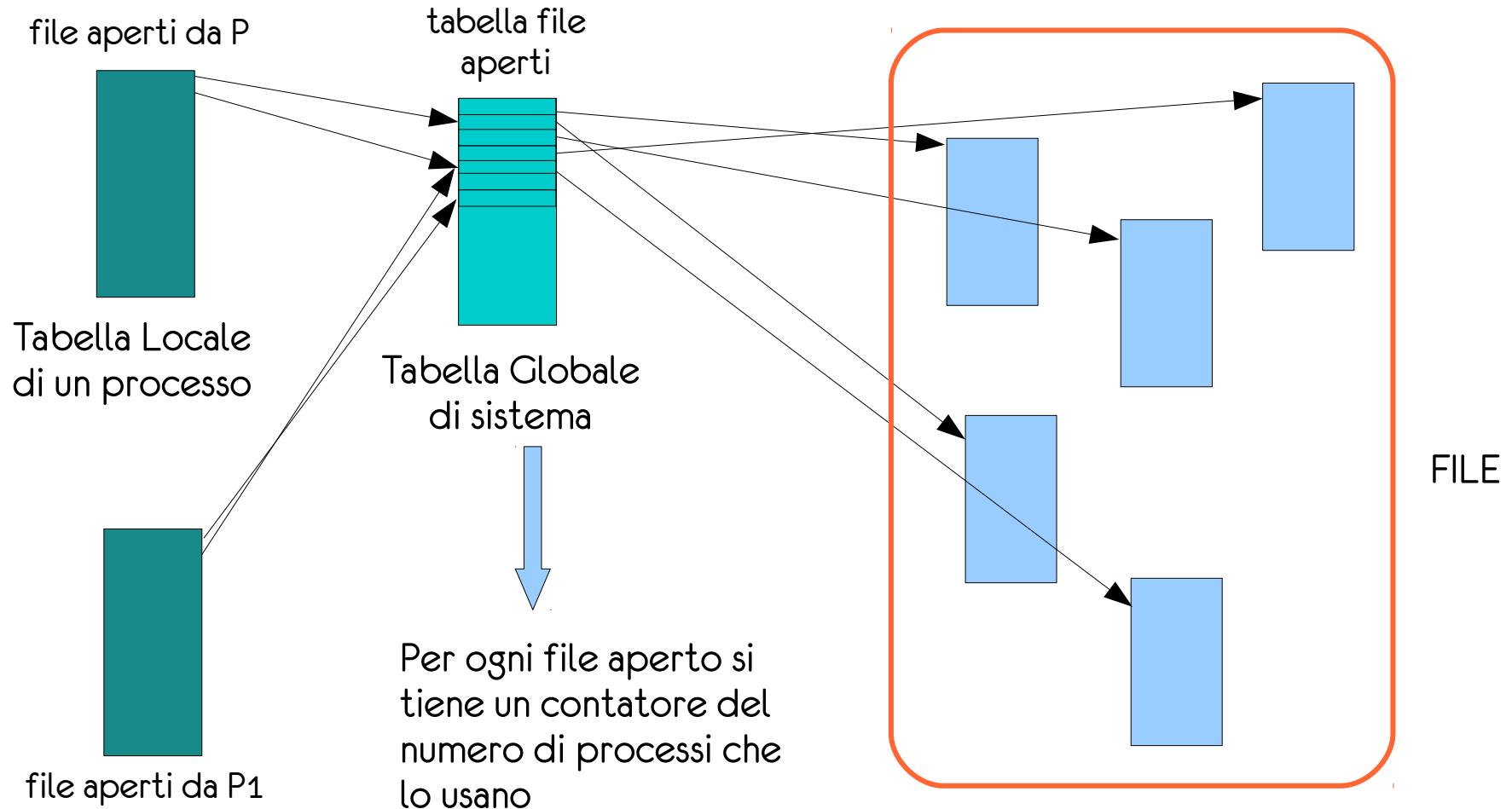
associazione logica
p. di v. astratto



implementazione



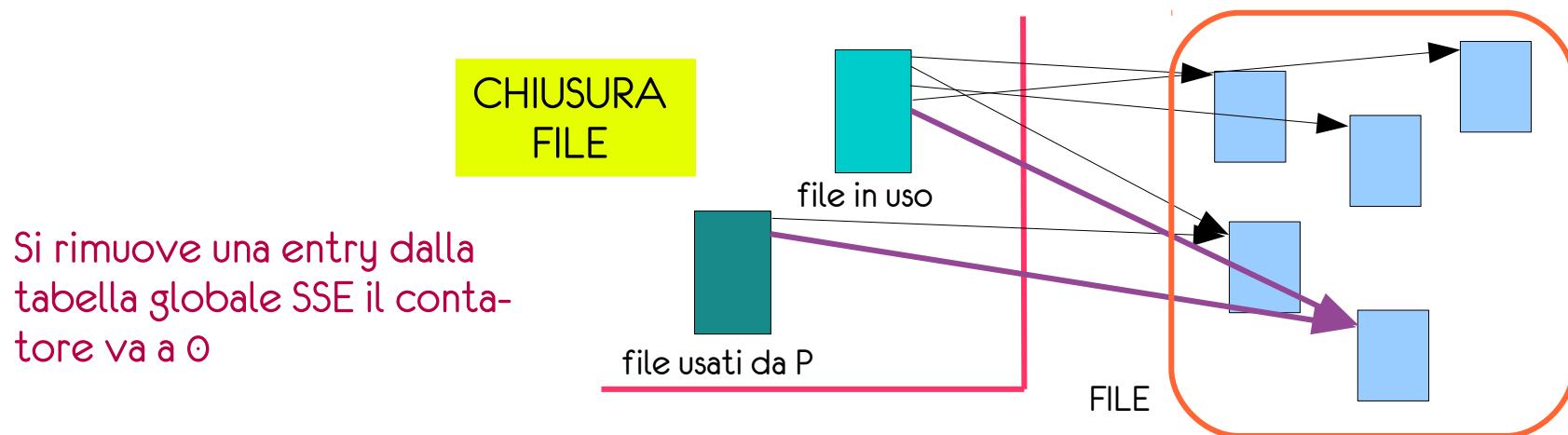
Apertura e chiusura di un file



implementazione

Apertura e chiusura di un file

- Più in generale l'operazione di apertura di un file consente di ottenere un handle del file, cioè un riferimento che consente di operare effettivamente sul file stesso
- Questa operazione modifica alcune strutture gestite dal SO, che tracciano quali file sono in uso, in generale, e quali file sono in uso da ogni specifico processo
- Un file aperto da un processo è inteso come una **risorsa** in uso da parte del processo
- L'apertura di un file comporta (come vedremo in dettaglio) l'**allocazione di un insieme di risorse** di file system che consentono l'accesso al file stesso
- Chiudere un file significa rilasciare le risorse di file system allocate



Apertura e chiusura di un file

- Più in generale l'operazione di apertura di un file consente di ottenere un handle del file, cioè un riferimento che consente di operare effettivamente sul file stesso
- Questa operazione modifica alcune strutture gestite dal SO, che tracciano quali file sono in uso, in generale, e quali file sono in uso da ogni specifico processo
- Un file aperto da un processo è inteso come una risorsa in uso da parte del processo
- L'apertura di un file comporta (come vedremo) l'allocazione di un insieme di risorse di file system che consentono l'accesso al file stesso
- **Chiudere un file** significa rilasciare le risorse di file system allocate
- La chiusura comporta un insieme di operazioni sulle strutture gestite dal SO per indicare che un certo processo ha terminato di usare un file.
- Se il processo **era l'unico utilizzatore del file**, le pagine di RAM usate per consentire un'elaborazione efficiente dei dati possono essere liberate
- A differenza da altri tipi di risorse, in genere i file sono risorse condivisibili

Apertura di un file

- **Esempio**, in C posso usare:

- **fopen**: funzione di libreria, restituisce come handle un FILE *
- **open**: system call, restituisce come handle un numero intero detto file descriptor

```
struct _IO_FILE {
    int _flags;      /* High-order word is _IO_MAGIC; rest is flags. */

    /* The following pointers correspond to the C++ streambuf protocol. */
    /* Note: Tk uses the _IO_read_ptr and _IO_read_end fields directly. */
    char* _IO_read_ptr; /* Current read pointer */
    char* _IO_read_end; /* End of get area. */
    char* _IO_read_base; /* Start of putback+get area. */
    char* _IO_write_base; /* Start of put area. */
    char* _IO_write_ptr; /* Current put pointer. */
    char* _IO_write_end; /* End of put area. */
    char* _IO_buf_base; /* Start of reserve area. */
    char* _IO_buf_end; /* End of reserve area. */

    /* The following fields are used to support backing up and undo. */
    char *_IO_save_base; /* Pointer to start of non-current get area. */
    char *_IO_backup_base; /* Pointer to first valid character of backup area */
    char *_IO_save_end; /* Pointer to end of non-current get area. */
    ... ecc ...
}
```

chiusura di un file

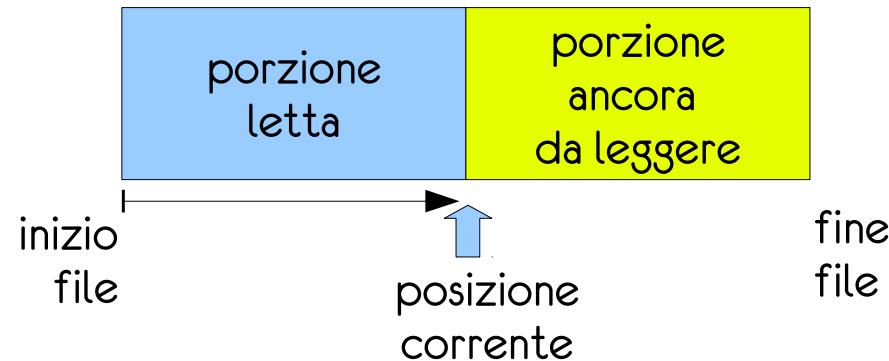
- Per esempio in C si utilizzano
 - `fclose`: funzione di libreria, ha come parametro un FILE *, l'`handle` ottenuta tramite `fopen`
 - `close`: system call, ha come parametro un file descriptor (un numero intero), l'`handle` ottenuta tramite `open`

Lettura e scrittura su file

- Anche lettura e scrittura richiedono di solito una handle come argomento
- Inoltre occorre definire il punto del file da cui leggere o in cui scrivere
 - **accesso sequenziale:** il programmatore non deve specificare la posizione in modo esplicito, il SO manterrà un puntatore alla posizione corrente di lettura/scrittura per il processo. Questa informazione è mantenuta nella tabella di file aperti locale (del processo).
 - **accesso diretto:** si può leggere/scrivere da/in specifiche posizioni del file, che vanno indicate espressamente

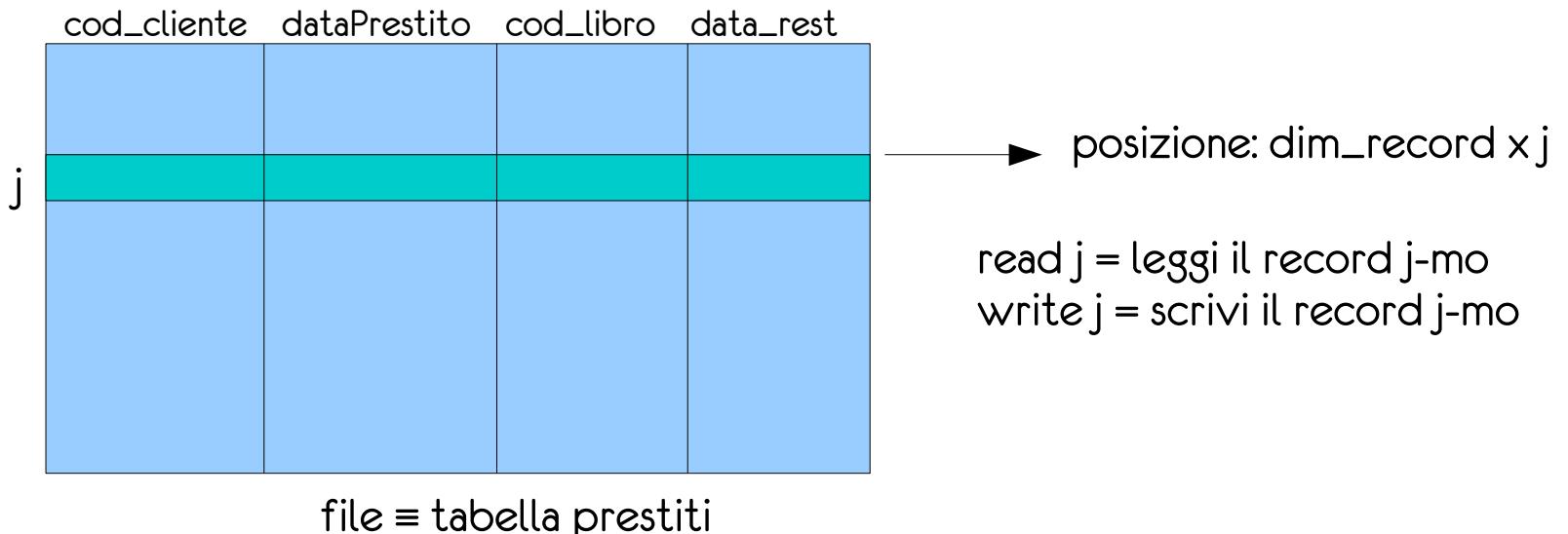
Accesso sequenziale

- è il metodo di accesso a file più comune
- i contenuti di un file vengono percorsi secondo l'ordine logico sequenziale dei dati stessi
- ogni **operazione di lettura** fa avanzare un puntatore che indica la posizione raggiunta correntemente all'interno del file
- ogni **operazione di scrittura** aggiunge del contenuto in fondo al file
- alcuni SO consentono una terza operazione: riportare il puntatore alla posizione corrente all'inizio del file



Accesso diretto

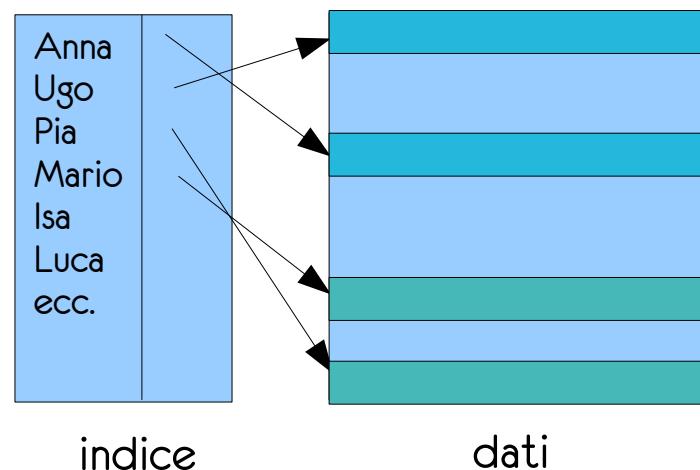
- **presupposto:** i dati contenuti in un file hanno un formato prestabilito
- il file è visto come una **sequenza di record di pari dimensione**
- conoscendo tale dimensione e la posizione del record di interesse è possibile accedervi senza scorrere l'intero file
- **es.** tabella di una base di dati



Accesso a indice

- definito sulla base del precedente
- un **file indicizzato** è in realtà costituito da due file:
 - il **file dei contenuti** veri e propri, memorizzati secondo un preciso formato
 - un **file indice**, contenente riferimenti ai record
- l'indice non è necessariamente numerico

mantenendo
l'indice ordinato
si possono effettuare ricerche
veloci



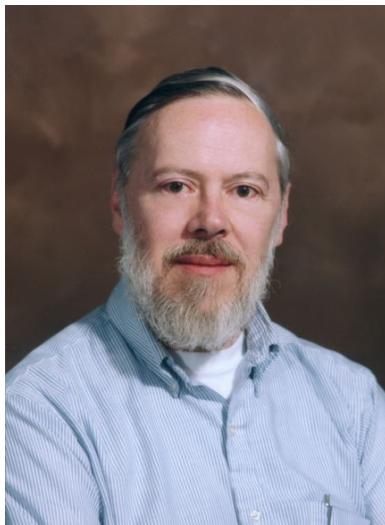
Es. molti cosiddetti database contenenti dati relativi a **esperimenti medici/biologici** sono in realtà file indicizzati

Protezione

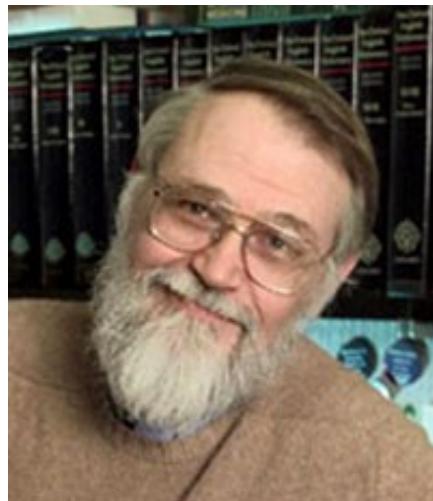
- Alcuni SO richiedono di indicare la **modalità di apertura di un file** (di base lettura o lettura/scrittura), la scelta può disabilitare determinate operazioni
- Certi SO consentono di associare ai file dei **diritti di accesso** che indicano le modalità di apertura del file consentite agli utenti (es. un utente non può sovrascrivere il codice del SO)
- Certi SO consentono di associare ai file dei **lock**:
 - **lock condiviso**: analogo a un lock di lettura, consente a un insieme di processi di effettuare determinate operazioni (anche in parallelo) sullo stesso file
 - **lock esclusivo**: analogo a un lock di scrittura, solo il processo che detiene il lock può usare il file
- I lock possono essere gestiti come avvertenze o obblighi
 - **lock consigliato**: se un processo cerca di usare un file il cui lock è detenuto da un altro processo, il SO lo avvisa ma non impedisce l'operazione
 - **lock obbligatorio**: l'accesso viene impedito

Directory

- è un'entità del file system (FS) preposta a contenere file o altre directory
- in passato molti file system erano piatti, nel senso che i file erano definiti tutti allo stesso livello, senza alcuna organizzazione; alcuni FS consentono (anche oggi) l'uso di un solo livello di directory. In questo caso il FS è partizionato è insiemi di file
- Il primo FS gerarchico è nato con Unix ed è stato realizzato da Dennis Ritchie (Turing Award nel 1983 come creatore di Unix, insieme a Ken Thompson)



Dennis Ritchie



Brian Kernighan

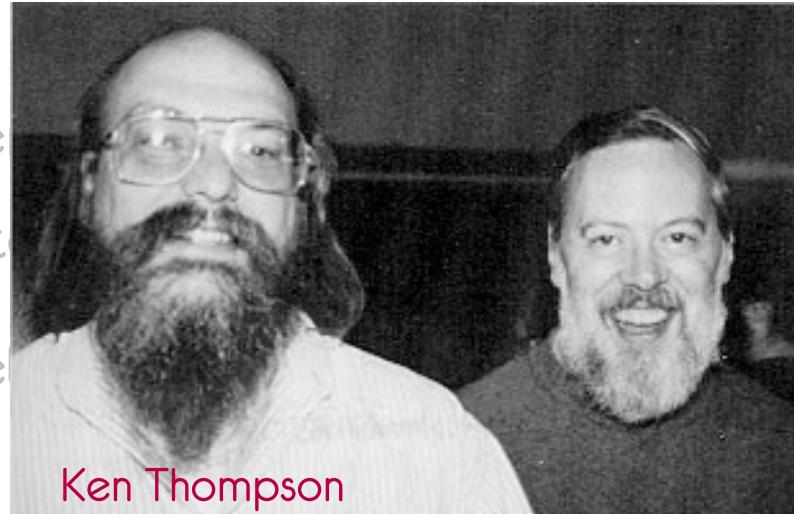
Directory

Richard Stallman

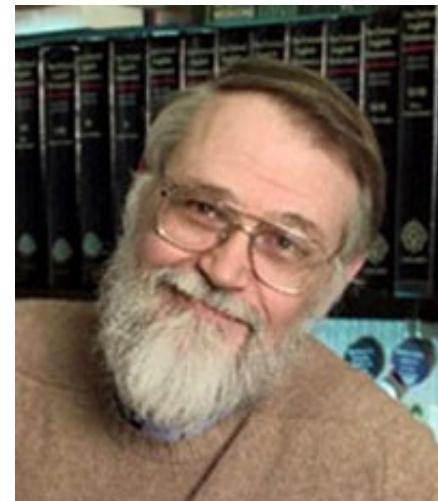


- è un attivista per i diritti dei programmi liberi
- in particolare ha fondato la FSF (Free Software Foundation) insieme a Bruce Perens
- oggi è ancora attivo nel campo della software freedom
- insieme a Ken Thompson ha creato il linguaggio C
- Il primo FS gerarchico è nato con Unix ed è stato realizzato da Dennis Ritchie (Turing Award nel 1983 come creatore di Unix, insieme a Ken Thompson)

Ken Thompson



Dennis Ritchie



Brian Kernighan

?

forse
qualcuno
di voi



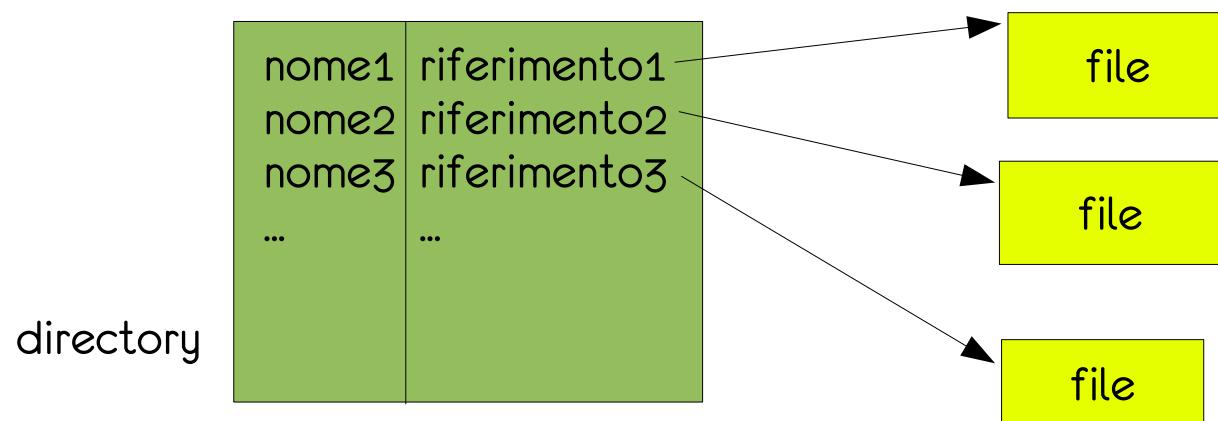
Frances Elizabeth Allen
(Pr. Turing 2006)



Dijkstra (Pr. Turing, 1972
linguaggio Algol)

Directory

- Da un punto di vista astratto una directory è una **tabella** che consente di accedere ai contenuti di un file a partire dal suo nome
- Operazioni possibili
 - **scrittura**: aggiungere/rimuovere file (se organizzazione gerarchica anche directory)
 - **lettura**: listare i contenuti
 - **ricerca** di un file (se organizzazione gerarchica anche directory)
 - **attraversamento**: percorrere la struttura definita dalle directory



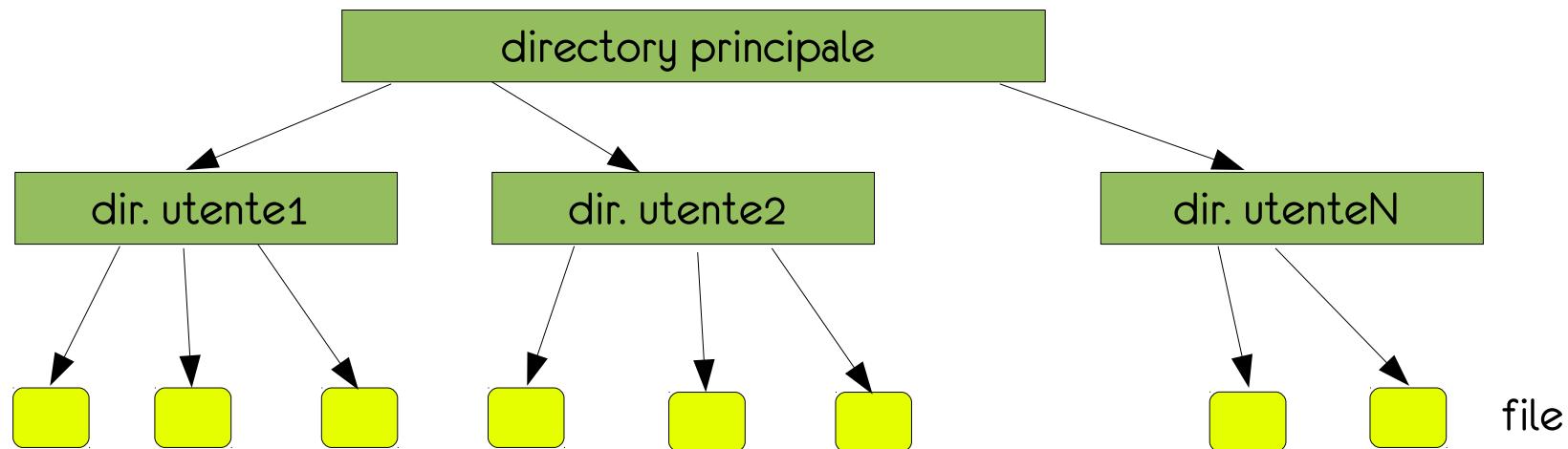
Directory a un livello

- Tutti i file sono contenuti all'interno della stessa directory
- Limiti: non posso avere due file con lo stesso nome, molto difficile gestire la multi-utenza



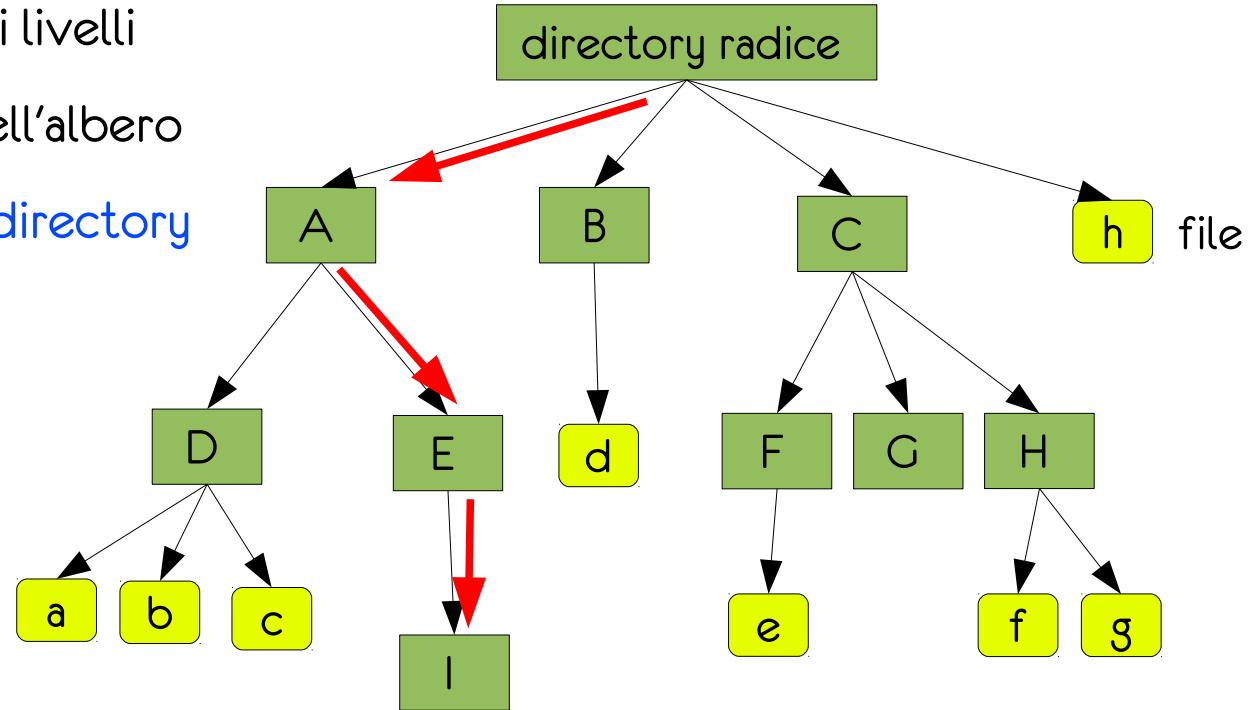
Directory a due livelli

- Ogni utente inserisce tutti i file in una propria directory (non ulteriormente strutturabile)
- Tutte le directory utente sono riferite da una directory superiore detta directory principale
- Limiti: poco flessibile, un utente può avere molti file, due utenti potrebbero voler condividere alcuni file ma in questo modello ciò non è consentito



Directory ad albero

- ho un numero qualsiasi di livelli
- i **file** sono tutti **foglie** dell'albero
- tutti i **nodi interni** sono **directory**



- due elementi dell'albero possono avere lo stesso nome se non sono fratelli
- ogni nodo è identificato in modo univoco dall'unico **cammino assoluto** che lo collega alla radice

Esempio: **radice/A/E/I**

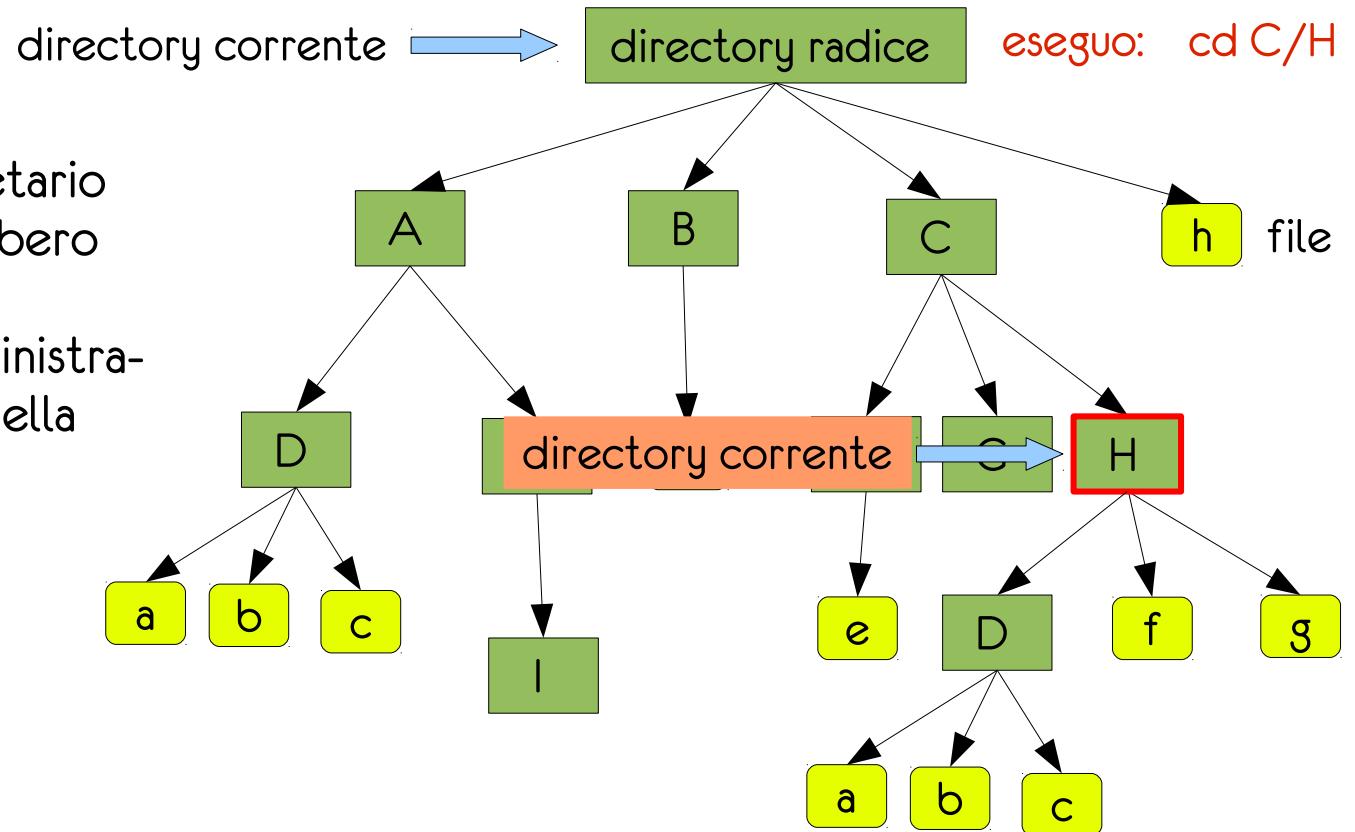
Directory ad albero

- Usare cammini assoluti per identificare i file è scomodo se i cammini sono lunghi, es: `/home/utente_pippo/documenti/2007/relazioni/rel1.txt`
- Di norma si consente di definire per ogni utente una “directory di lavoro” (**directory corrente**): si tratta di un riferimento ad una delle directory dell’albero
- L’utente è virtualmente posizionato in quella directory:
 - per accedere ai contenuti della cartella ne digita il solo nome
 - per accedere ad altri contenuti dell’albero può utilizzare, a scelta, **cammini relativi** oppure cambiare directory di lavoro e poi usarne i soli nomi

Directory ad albero

Ogni utente sarà proprietario di un particolare sottoalbero

Un utente speciale (amministratore) sarà proprietario della parte di sistema



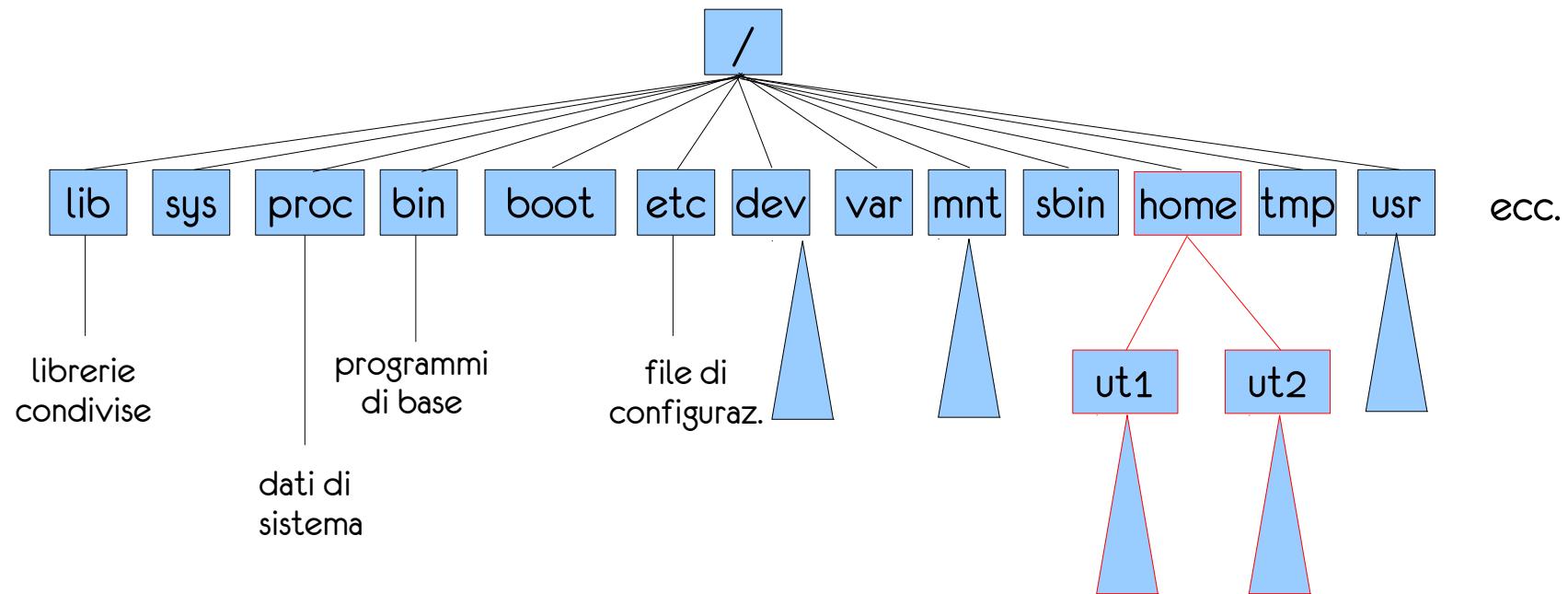
Per riferirmi al file "f" basta che ne digitai il nome

Per riferirmi al file "a" della sottodirectory "D" digiterò D/a

Posso indicare che intendo risalire l'albero userò un simbolo speciale, per es. ".."

..../..//h è un riferimento al file "h" contenuto nella directory radice

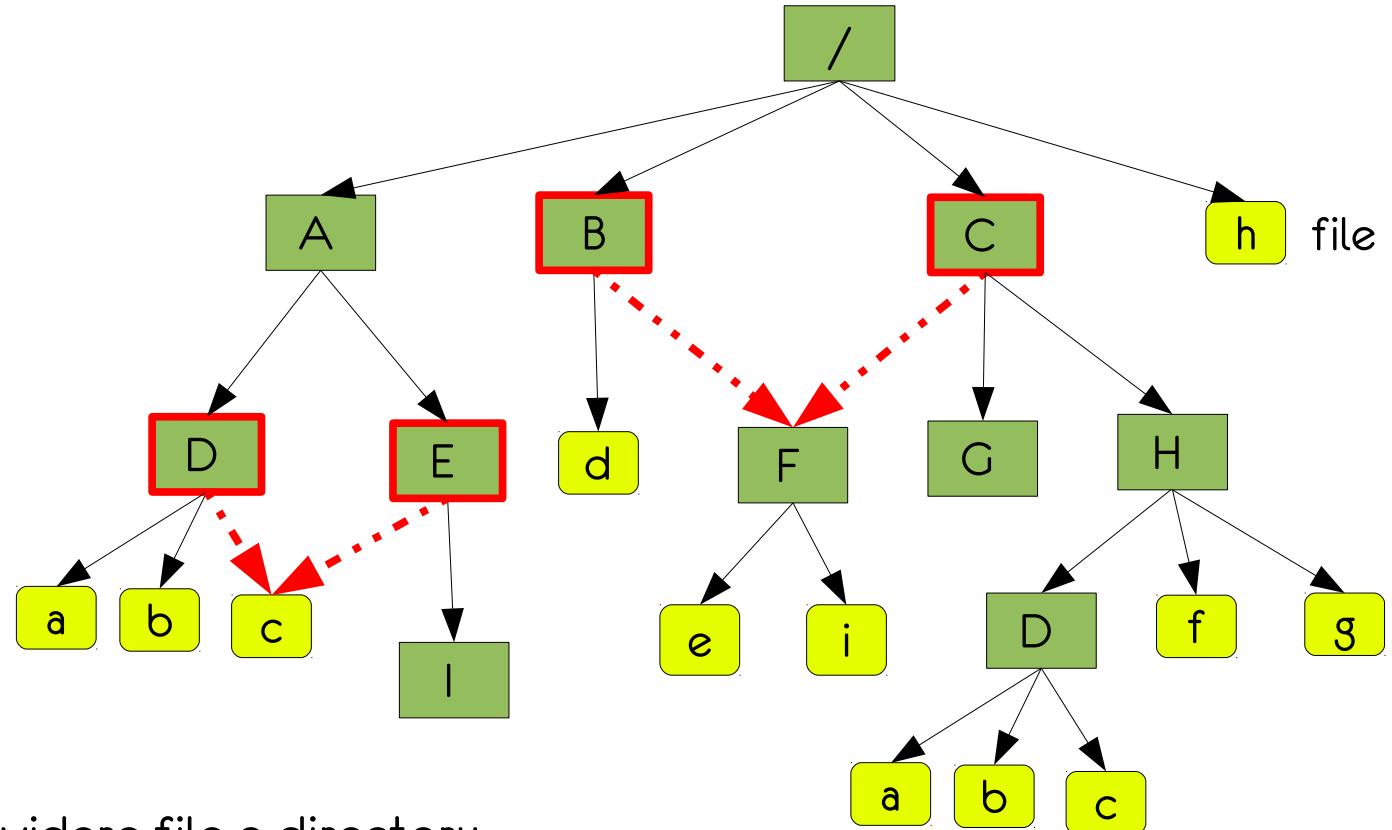
Esempio



Mai scrivere programmi che identificano i file attraverso un cammino assoluto
Ogni SO ha una propria struttura delle directory, l'installazione del programma
posiziona l'eseguibile in un punto del file system che non necessariamente
ricalca la struttura usata sulla macchina su cui il programma è stato sviluppato

C:\Documents\Progetto\Versione3\ppp.dat non è collocabile nell'albero riportato qui sopra

Directory con grafo aciclico

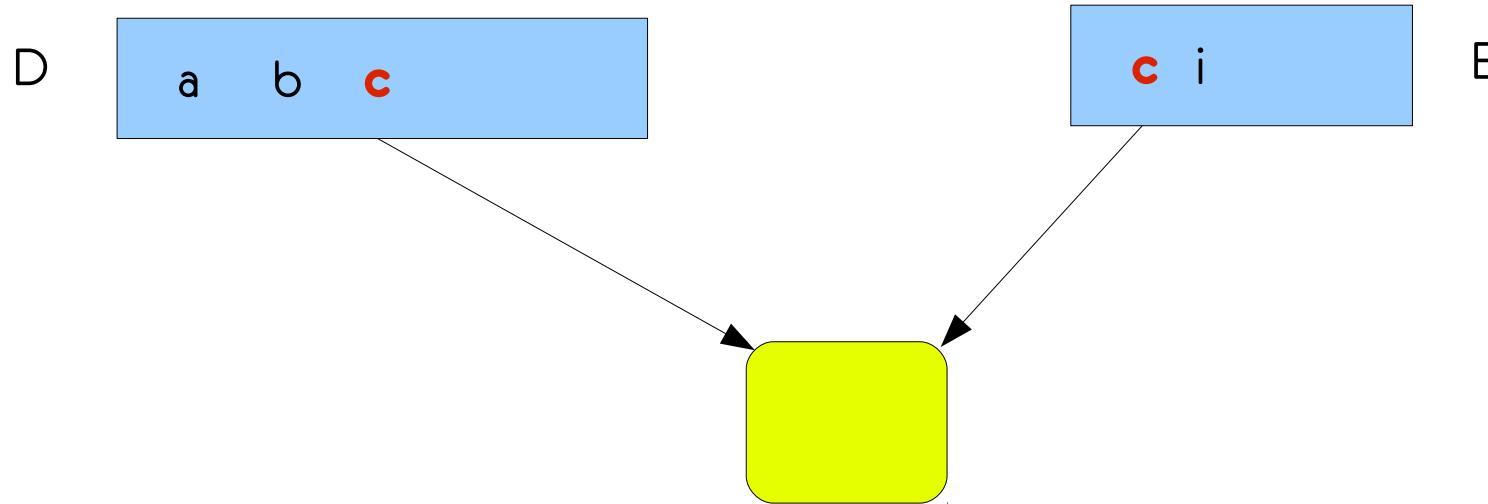


È possibile condividere file o directory

Ciò significa che un file/directory sono accessibili
attraverso cammini assoluti differenti

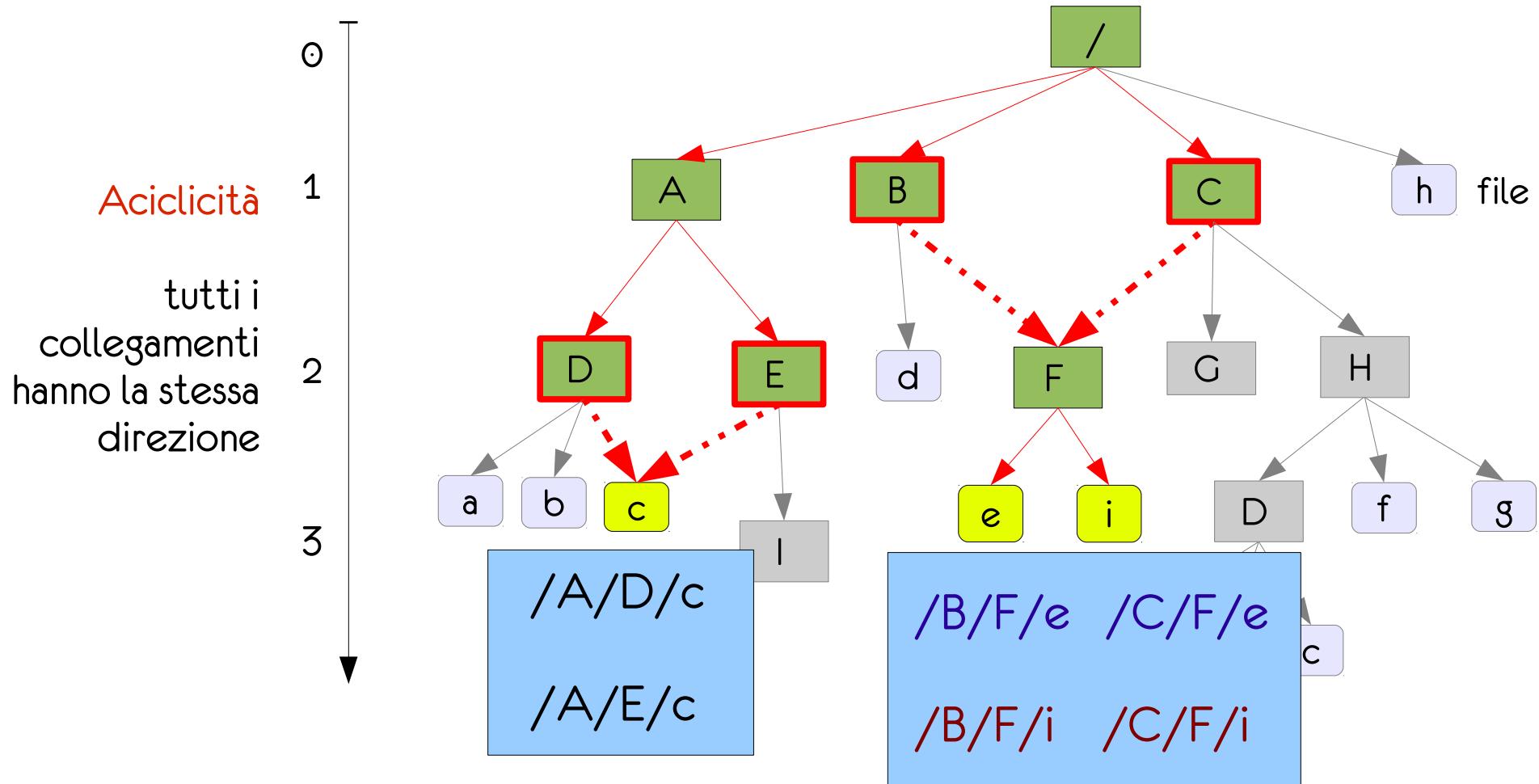
Es. "c" è contenuto sia in E che in D

Collegamenti multipli



Le directory D ed E hanno un'intersezione non nulla
entrambe contengono il nome "c" con associato un
riferimento allo stesso oggetto

Directory con grafo aciclico



Directory con grafo aciclico

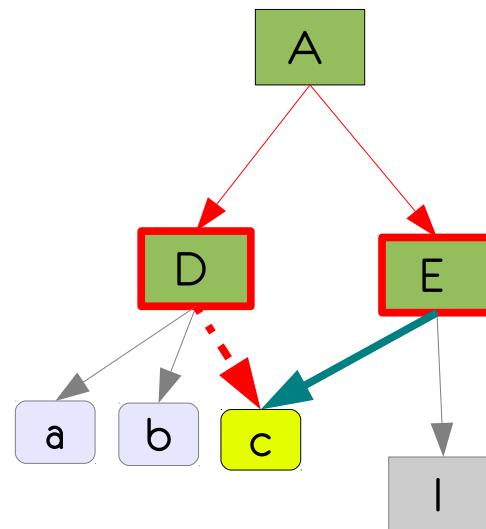
Cosa significa eseguire la cancellazione di /A/D/c?
Il file verrà cancellato?

Es. supponiamo che D ed E siano le radici dei sottoalberi di proprietà di due utenti diversi
“c” è un file condiviso

Cancellare “c” da “D” può significare:

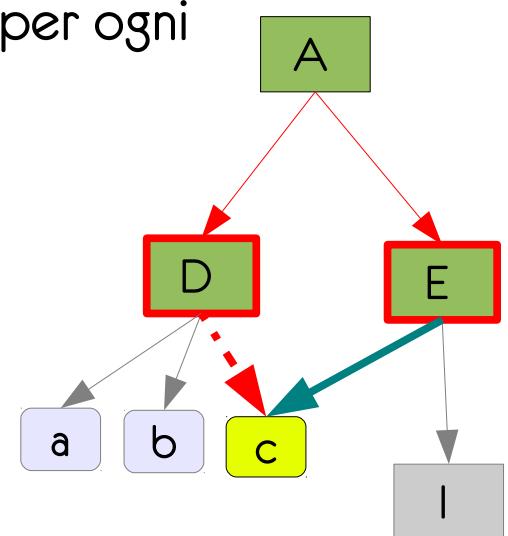
<a> “c” non serve più a nessuno dei due utenti
-> “c” va rimosso

**** uno dei due utenti rinuncia ad usare “c”
-> soltanto uno dei collegamenti a “c” va rimosso

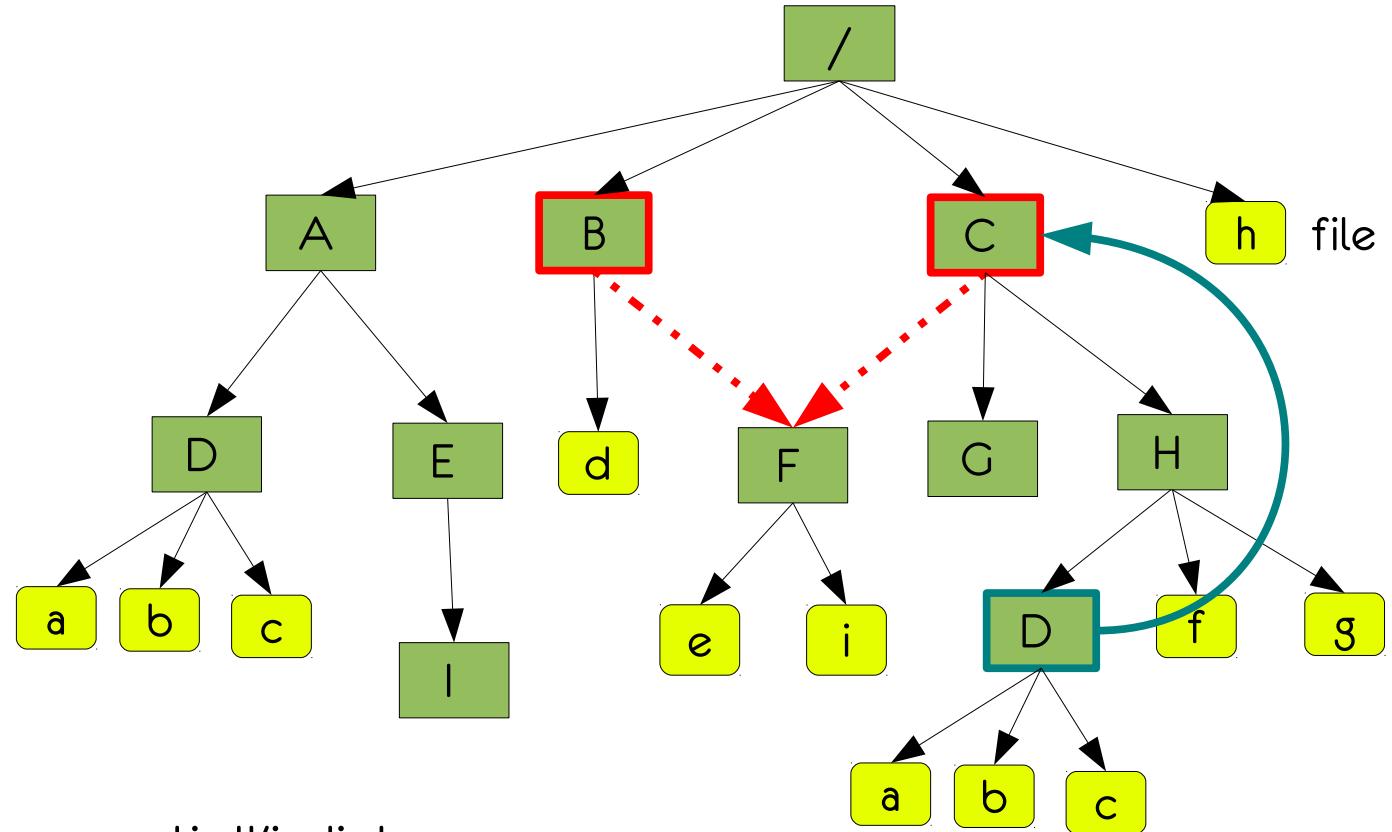


Link

- **Soluzione demandata all'utente** che può decidere come accedere a un file/directory ed effettuare collegamenti di tipo diverso fin dall'inizio. La relazione contenuto-contenitore creata con la creazione del file è forte: "rm c" rimuove il file. Invece nel caso dei link:
 - **link simbolico**: il collegamento rappresenta l'interesse ad usare un file, cancellare il collegamento significa dire che non ci serve più usarlo (non che vada rimosso)
 - **link fisico**: il collegamento ha una valenza più forte cancellarlo vuol dire cancellare il file
- **Soluzione globale**: mantenere un numero di riferimenti per ogni file/directory:
 - La creazione di un link incrementa il valore
 - cancellare un collegamento comporta decrementare tale numero
 - solo quando si raggiunge il valore zero il file viene rimosso

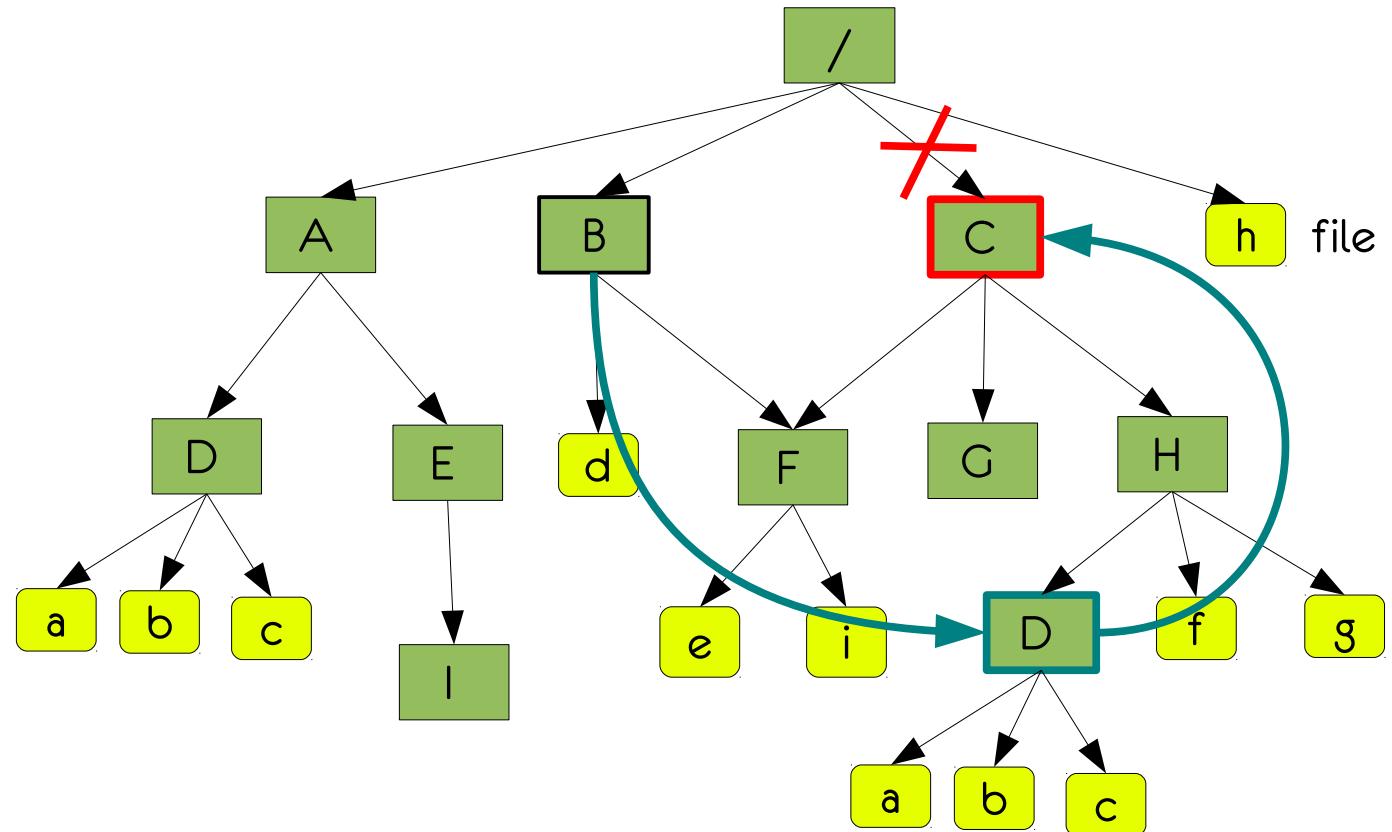


Directory a grafo generale



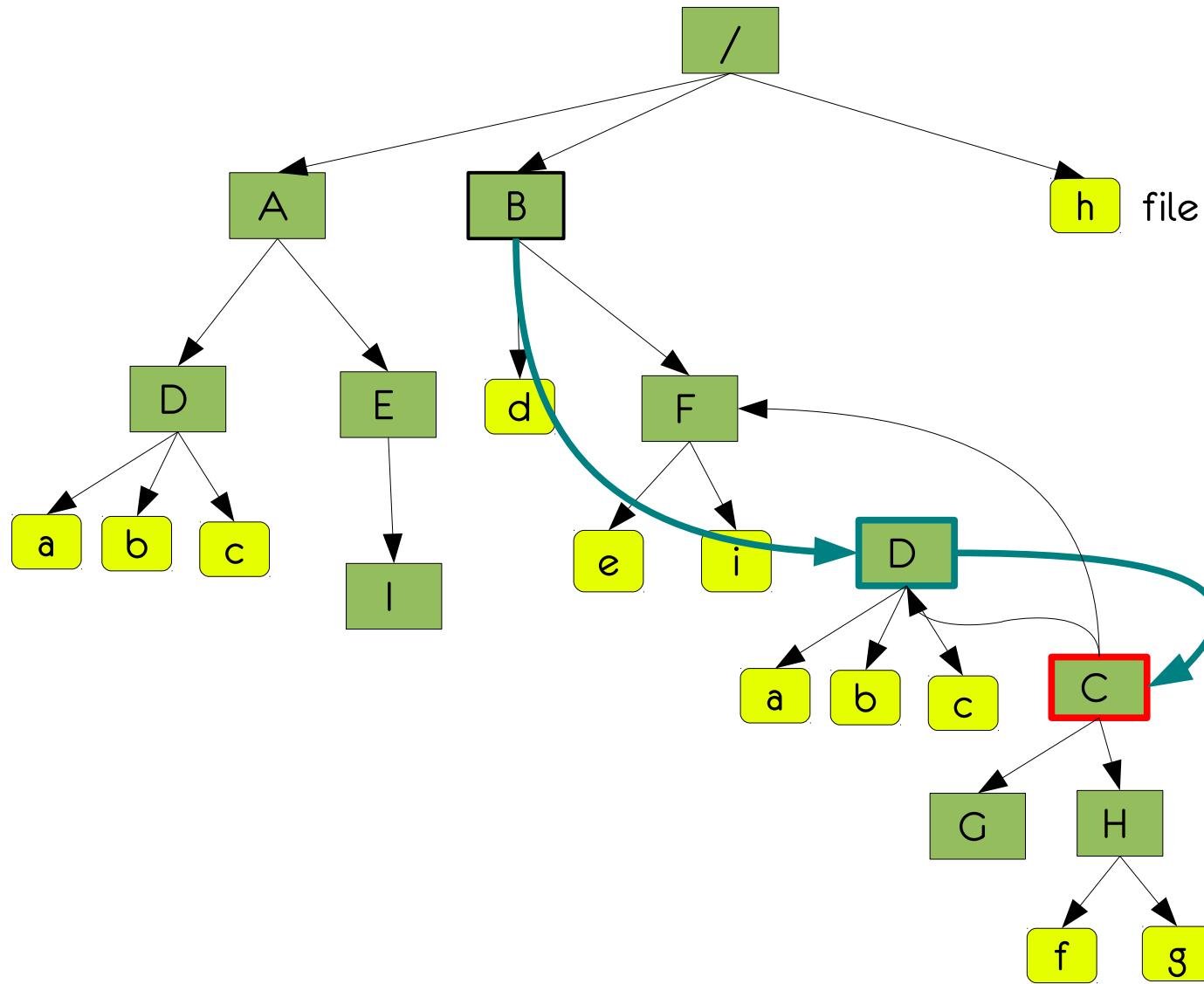
Posso avere collegamenti all'indietro
con la creazione di cicli nel grafo

Cancellazione

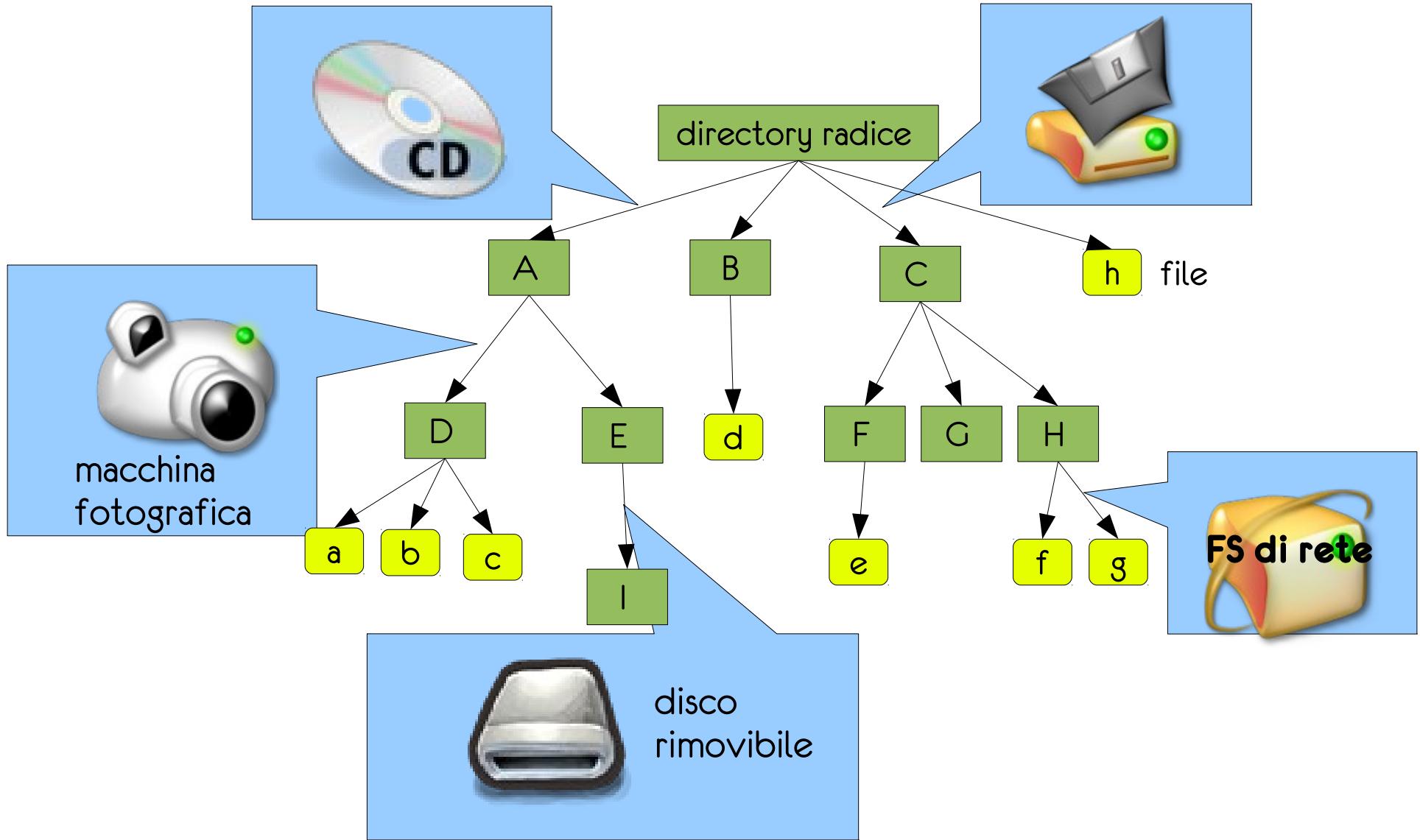


Se cancelliamo il collegamento dalla radice a C
il sottoalbero con radice C è ancora accessibile
attraverso B/D. È come se il grafo cambiasse
forma ...

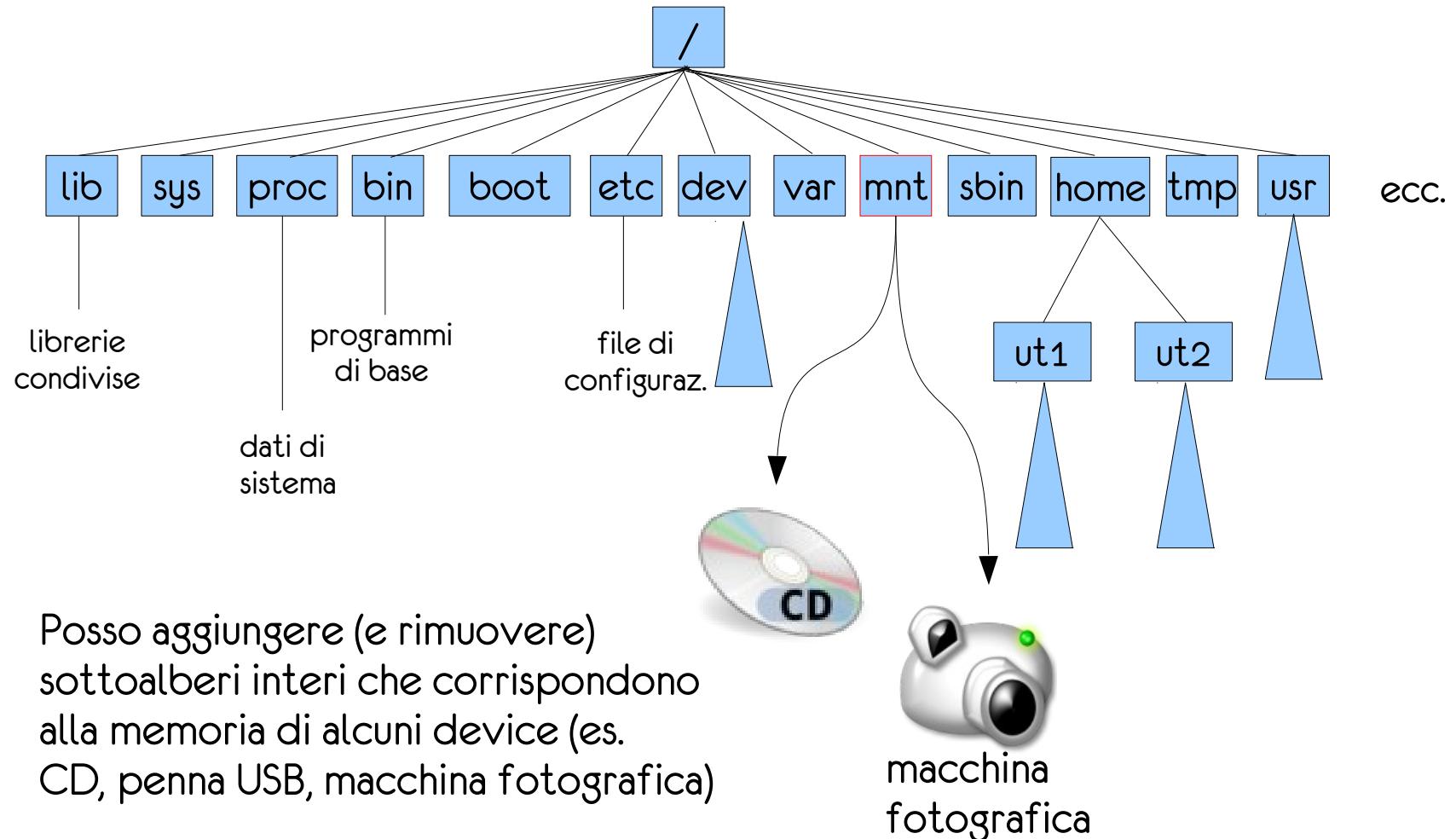
Cancellazione



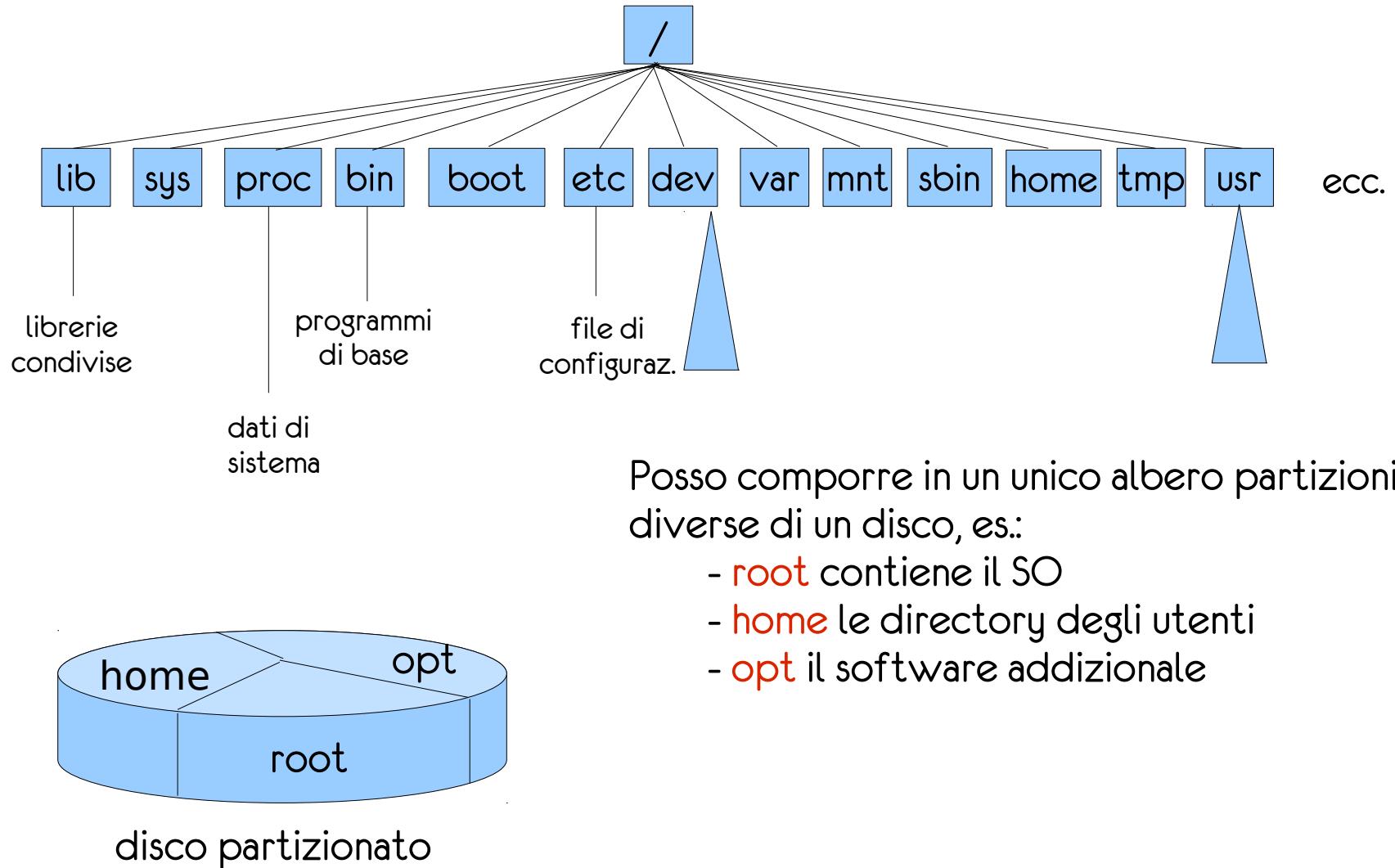
Mounting del file system



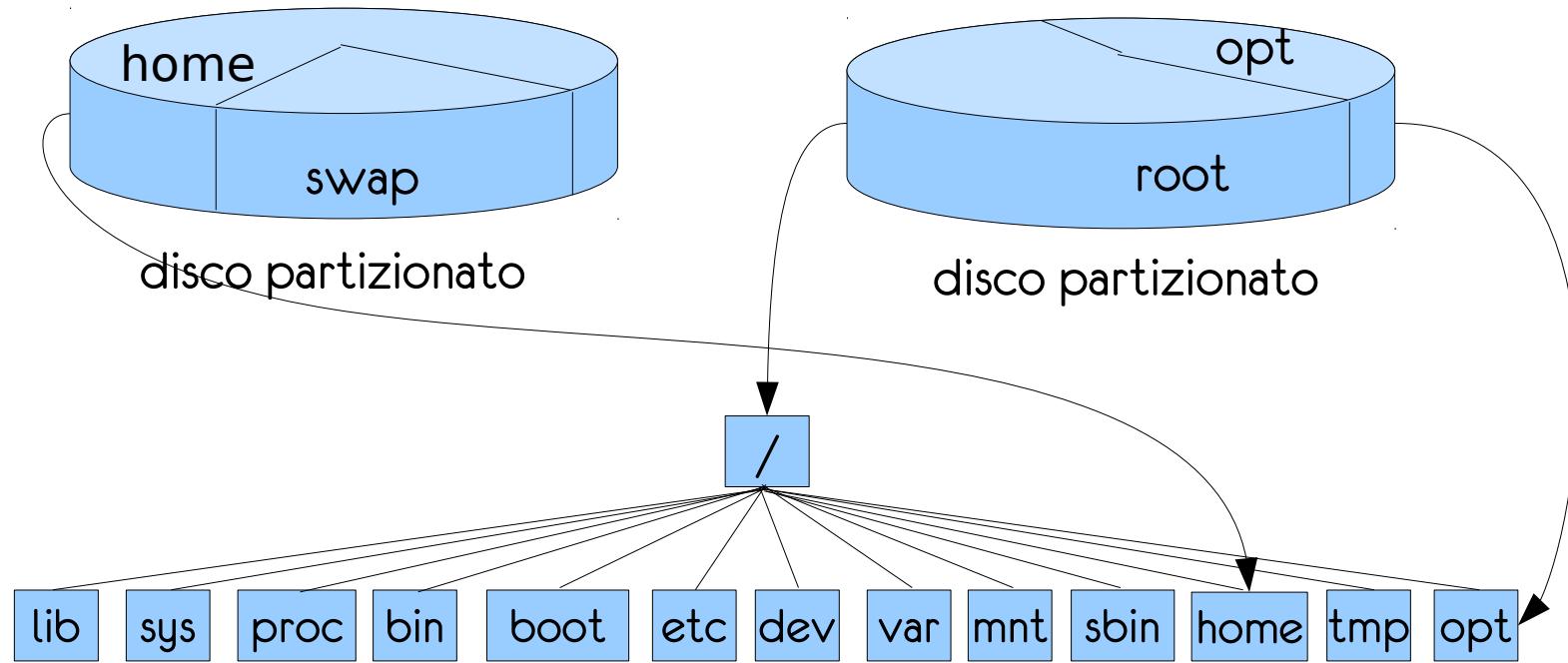
Mount



Mount



Mount



Posso comporre più dischi

L'utente accederà alla memoria senza rendersi conto dell'esistenza di diversi supporti distinti

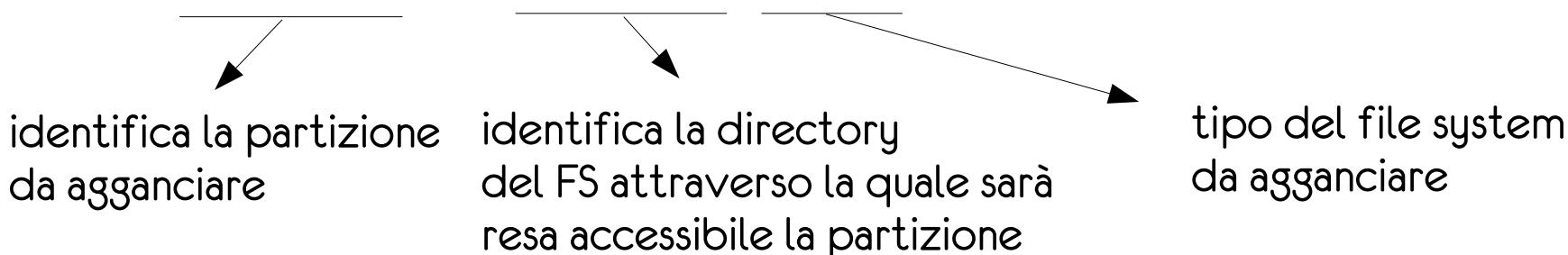
Mount

- Il mount di una penna USB è diverso dal mount di una partizione: il primo viene fatto **run-time** quando il device è agganciato fisicamente all'elaboratore; il secondo viene fatto **all'avvio** della macchina
- La specifica di cosa va montato all'avvio è contenuta in un **file speciale**. In Unix/Linux si chiama fstab e si trova nella directory /etc

Esempio di /etc/fstab

Pluggable devices are handled by uDev, they are not in fstab

/dev/hda3	/	ext3	defaults,noatime 1 1
/dev/hda2	swap	swap	sw,pri=1 0 0
/dev/hda4	/home	ext3	defaults,noatime 1 2
/dev/hda1	/mnt/hda1	reiserfs	noauto,users,exec 0 0



Mount: esempio ubuntu

```
# /etc/fstab: static file system information.
#
# <file system> <mount point>  <type>          <options>
proc            /proc           proc            defaults
/dev/sda2        /               ext3            defaults,errors=remount-ro
/dev/sda4        /home           ext3            defaults
/dev/sda3        /opt             ext3            defaults
/dev/sda1        none            swap            sw
/dev/hda         /media/cdrom0  udf,iso9660    user,noauto
```

Esempio script di mount

```
#!/bin/sh

DEF_MOUNT_POINT=/mnt/usbkey/
DEF_DEV=/dev/sdc1
echo "Default usbPen device: $DEF_DEV"
echo "Default mount point: $DEF_MOUNT_POINT"
umask 0022

# Get local user
#LOCALUSER=`id -gn`

# Get local user's uid
INUID=`mygetuid`

# Get local user's gid
INGID=`mygetgid`

echo "Your local user's password may be requested ..."
(sudo /bin/mount -t auto -o uid=$INUID,gid=$INGID $DEF_DEV $DEF_MOUNT_POINT)
echo "Mounted $DEF_DEV on $DEF_MOUNT_POINT"
```

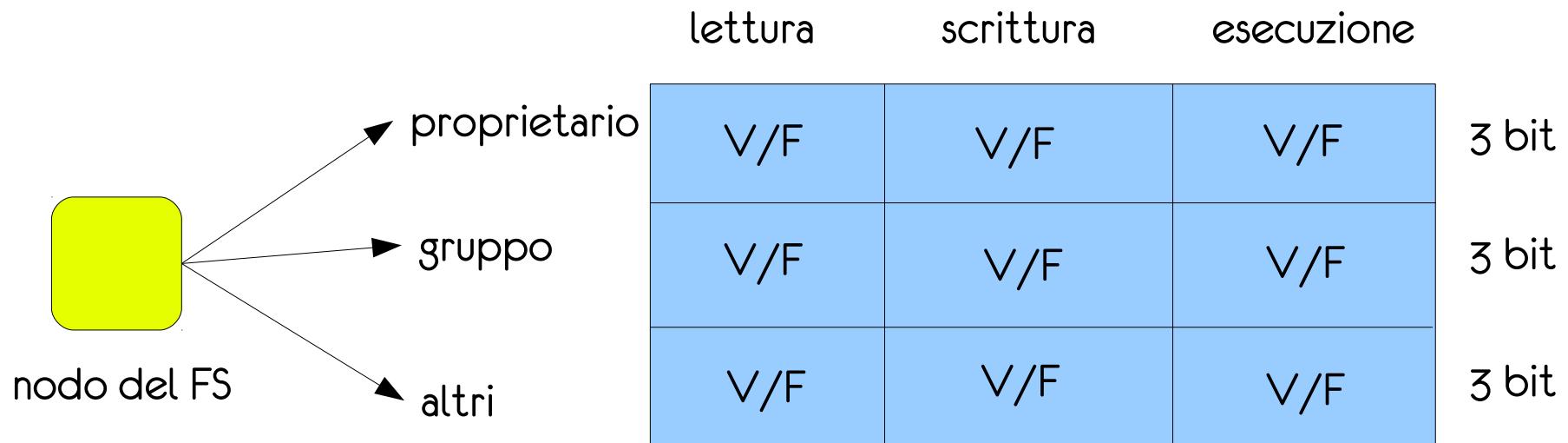
File System distribuiti

- I file a cui si accede attraverso un FS possono anche **risiedere su macchine diverse** collegate fra di loro in rete
- Es. in laboratorio alcuni programmi di uso comune non sono installati macchina per macchina ma sono installati su una sola macchina il cui FS è collegato a quelli delle macchine-utenti
- **Modello client-server**
 - Il calcolatore che contiene fisicamente i file condivisi è detto **server**: mantiene il sottoalbero condiviso in una directory convenuta
 - le macchine-utenti sono **client**; spesso sono attuati meccanismi di protezione per cui solo certe macchine possono fungere da client
 - NB: un calcolatore può svolgere contemporaneamente funzione di client e di server
- è possibile **montare FS distribuiti** esattamente come si fa per le partizioni di una stessa macchina o con un chiavettaq USB

Protezione

- con questo termine si intende la salvaguardia delle informazioni contenute nel FS da accessi impropri
- **Soluzione 1:** uso di ACL, Access Control List
 - ad ogni nodo del FS viene associata una lista di autorizzazioni che specifica tutti gli utenti che possono accedere al file
 - questa soluzione complica l'implementazione delle directory, in quanto ogni directory dovrebbe mantenere associata a ciascun utente una ACL e le ACL sono di lunghezza diversa
 - inoltre chi compila la lista per ogni file? Il creatore del file?
- **Soluzione 2:** proprietario, gruppo, altri
 - ogni nodo ha un proprietario
 - tutti gli utenti sono divisi in gruppi di lavoro

Soluzione 2



V = diritto concesso

F = diritto non concesso

Es. se documento.odp ha associati i diritti 110 100 000 il proprietario può leggere e modificare il file, gli utenti del suo gruppo possono solo leggerlo, gli altri non possono fare alcuna operazione su questo documento

110 100 000

- se associamo a ciascuno dei 3 bit un diritto di accesso (lettura, scrittura o esecuzione) avremo che:
 - **100** (4 in decimale) abilita la lettura
 - **010** (2 in decimale) abilita la scrittura
 - **001** (1 in decimale) abilita l'esecuzione
- Composizioni di diritti di accesso sono date da somme di questi valori: non esistono due combinazioni diverse che, sommate, danno lo stesso risultato
- es: 110 è la somma di 100 e 010 abilita lettura e scrittura, 101 (5) abilita lettura ed esecuzione, 011 (3) abilita scrittura ed esecuzione, 111 (7) abilita tutto quanto

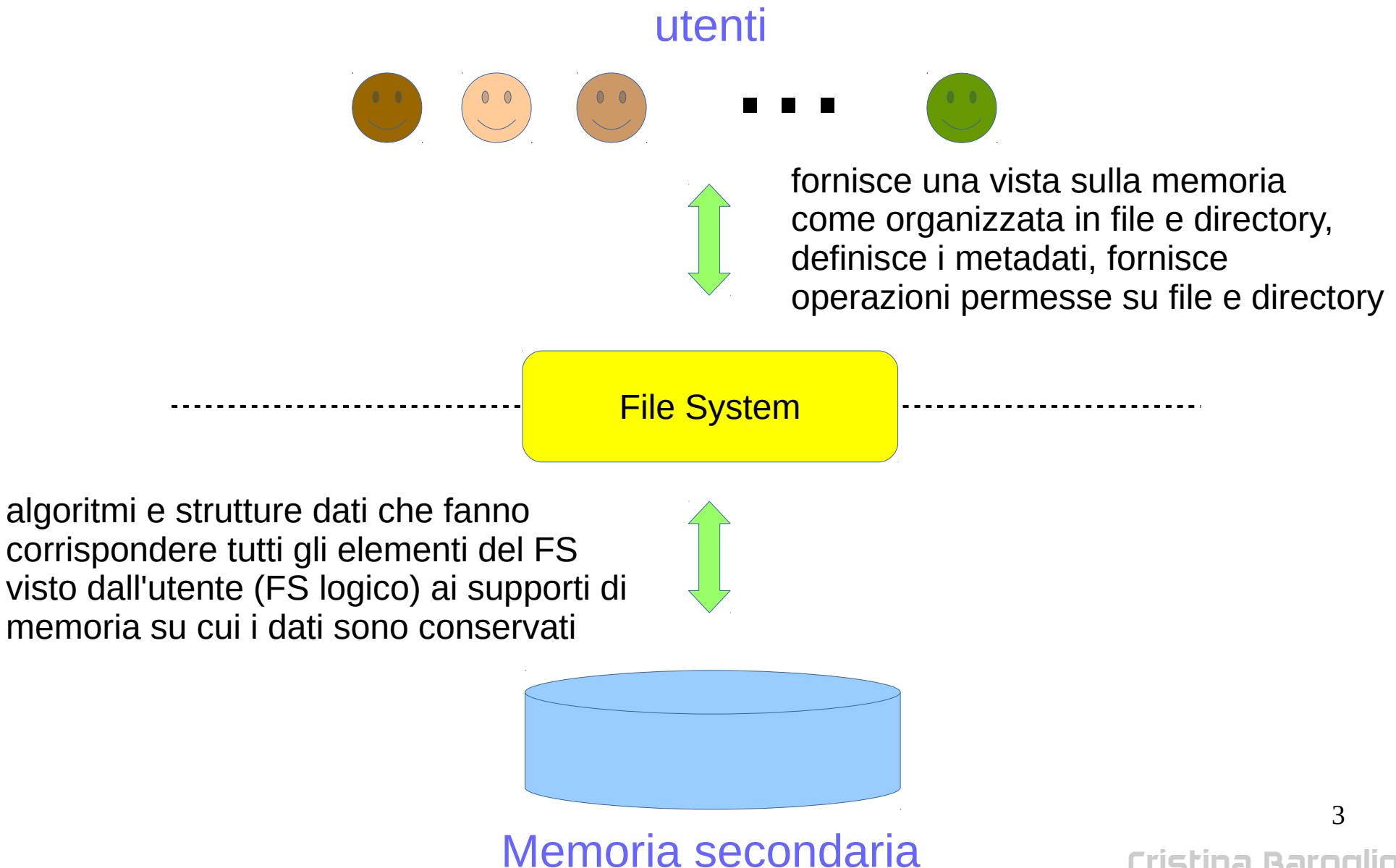
Implementazione del file system

Capitolo 11 del libro (VII ed.)

Introduzione

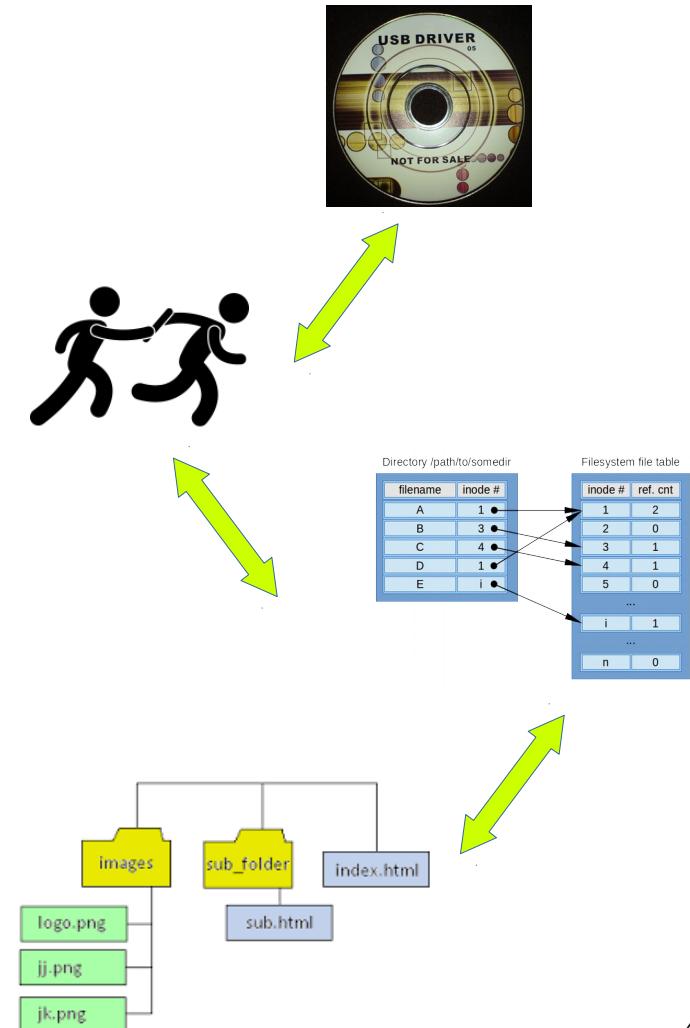
- Programmi e dati sono conservati in **memoria secondaria**
- la memoria secondaria ha le seguenti caratteristiche:
 - è **organizzata in blocchi** (un blocco può comprendere più settori)
 - è possibile **accedere direttamente** a qualsiasi blocco
 - è possibile leggere un blocco, modificarlo e riscriverlo esattamente **nella stessa posizione** di memoria
 - Si accede alla memoria secondaria attraverso un **FILE SYSTEM**

Il FS sta fra gli utenti e la memoria secondaria

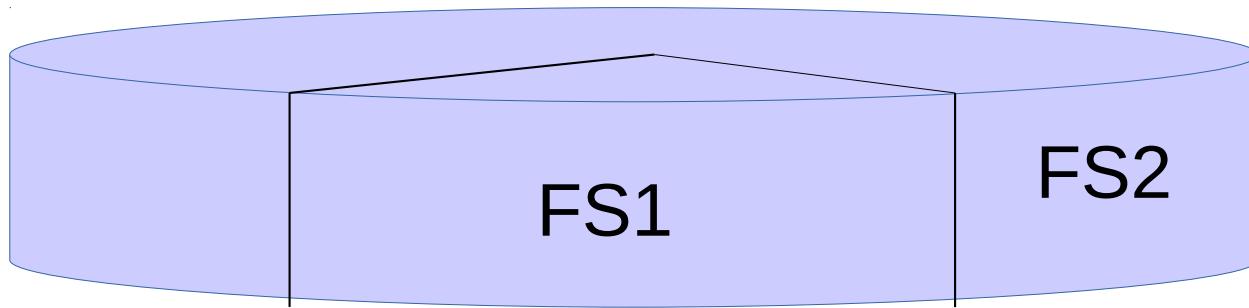


Livelli del file system

- il FS di solito è strutturato in una sequenza di livelli, dal basso verso l'alto:
 - **driver di dispositivo**: si occupa del trasferimento dei dati da dispositivo di memoria secondaria a RAM e viceversa. È il gestore del dispositivo
 - **file system di base**: è proposto a passare comandi al driver di dispositivo
 - **modulo di organizzazione dei file**: è a conoscenza di come i file sono memorizzati su disco, è in grado di tradurre indirizzi logici in indirizzi fisici. Mantiene e gestisce anche l'informazione relativa ai blocchi liberi
 - **file system logico**: gestisce il FS a livello di metadati. Ogni file è rappresentato da un **File Control Block** (FCB)



Struttura del disco



- Un disco può essere diviso in più **partizioni**
- Ogni partizione è organizzata in **un solo FS**
- Un FS contiene il S0 e/o i file degli utenti
- I diversi FS sono composti in una sola struttura (operazione **mount**)

Struttura su disco



- Un FS è una sequenza di blocchi di memoria secondaria
- Blocchi speciali:
 - **boot control block**: se il FS non contiene un S0 è un blocco vuoto, se invece contiene un S0, questo blocco contiene info necessarie in fase di bootstrap

Struttura su disco

- Blocchi speciali:
 - boot control block: ...
 - **volume control block** (o **superblocco**): descrive lo stato del FS stesso, quant'è grande, quanti file può contenere, ecc. ... In particolare, contiene il numero dei blocchi appartenenti alla partizione, la loro dimensione, il numero di blocchi liberi (e loro riferimenti)

Struttura su disco

- Blocchi speciali:
 - boot control block: ...
 - volume control block (o superblocco): ...
 - **struttura delle directory**: organizza i file, è implementata in modi diversi e contiene coppie <nome file, FCB>

Struttura su disco

- Blocchi speciali:
 - boot control block: ...
 - volume control block (o superblocco): ...
 - struttura delle directory: ...
 - una **lista di FCB** (file control block): contengono i metadati relativi ai file e consentono di accedere ai loro contenuti; gli FCB sono anche detti **inode** (index-node), ciascuno identificato da un numero

INODE / FCB

- Un **FCB** (o **inode**) mantiene le informazioni relative ai file
- Gli inode sono conservati in **memoria secondaria**
- **Esempio:** in Unix un inode su disco può contenere queste informazioni

- Identità del proprietario del file
- Tipo del file
- Diritti di accesso
- Tempi di accesso e modifica
- Dimensione del file numero di byte
- Tabella per l'accesso ai dati

(User ID)
(regolare, directory, link, device, ...)
(r w x per proprietario, gruppo, altri)
(data/ora ultimo accesso/ultima modifica) # di link al file

(indirizzi dei blocchi di mem. secondaria contenenti i dati)

NB: per semplificare l'accesso gli inode hanno **dimensione fissa** -> es. è prevista una lunghezza massima per i nomi dei file, l'indirizzamento della memoria avviene attraverso parole di lunghezze prefissata

Esempio: FS del SO MINIX

- Minix è un SO scritto in C da Andrew Tanenbaum (Vrije University) per scopi didattici (fine anni '80)
- I nomi di file sono al più di 14 caratteri
- i blocchi possono essere riferiti da parole di 16 bit (2^{16} blocchi) - questo impone un limite importante, se i blocchi sono grandi 1KB non è possibile gestire memorie secondarie più grandi di 64 MB
- Linux è nato come tentativo di superare questi due limiti di Minix

FS di MINIX



- 1 Boot block: contiene un codice di bootstrap usato per avviare il SO
- 2 Super block: descrive lo stato del FS (quant'è grande, quanti file può contenere, ecc.)
- 3 Lista degli inode: ogni inode corrisponde a un file (esistente o potenziale), uno in particolare corrisponde alla radice del FS
- 4 Blocchi dati: ogni blocco può appartenere a uno e un solo file

NB: in Unix le **directory** sono implementate come **file speciali**, quindi alcuni inode corrispondono a directory

Strutture dati in RAM

- All'avvio di un elaboratore viene “montato il FS” e vengono caricate in RAM delle informazioni che riguardano la sua struttura
- Lo scopo è consentire l'accesso (efficiente) ai file
- Le strutture mantenute in RAM sono:
 - ★ **mount table**: mantiene informazioni su ciascuna partizione montata, ogni operazione di mount/unmount la modifica
 - ★ **directory**: mantiene informazioni sulle directory a cui si è avuto accesso di recente (utile in particolare per quei SO che usano descrittori diversi per i file e per le directory. Alcuni SO, e.g. Unix, invece implementano le directory come file speciali)
 - ★ **tabella dei file aperti**: mantiene una copia arricchita dei FCB di tutti i file aperti al momento tabelle dei file dei processi: si ha una tabella di questo tipo per ogni processo; per ogni file aperto viene mantenuto un riferimento all'opportuno elemento della tabella (globale) dei file aperti

In-core inode (FCB in RAM)

Quando un file viene aperto per la prima volta il suo inode viene caricato in RAM ed arricchito da qualche informazione aggiuntiva

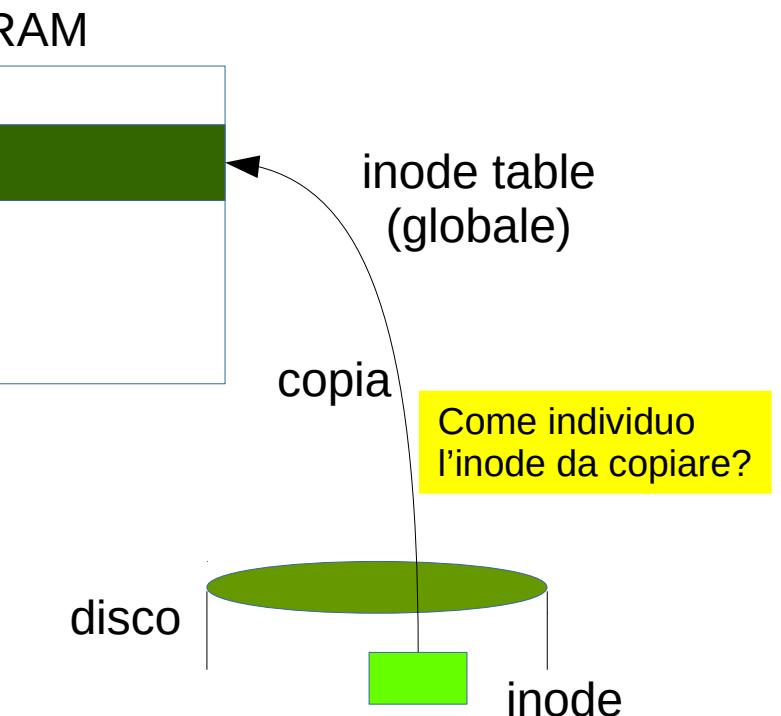
Esempio: in Unix un inode su ram può contenere queste informazioni

- ✓ **INODE COPIATO DA DISCO**
- ✓ **Stato dell'in-core inode:** dice se
 - ✓ l'inode è locked (il file non è disponibile)
 - ✓ la copia dell'inode in RAM è diversa da quella su disco,
 - ✓ il file è un punto di mount
 - ✓ ...
- ✓ **device number:** identificatore del FS a cui appartiene il file
- ✓ **inode number:** identificatore dell'inode nella struttura dati su disco
- ✓ **contatore** del numero di riferimenti all'inode (#utilizzi del file attuali)
- ✓ ...

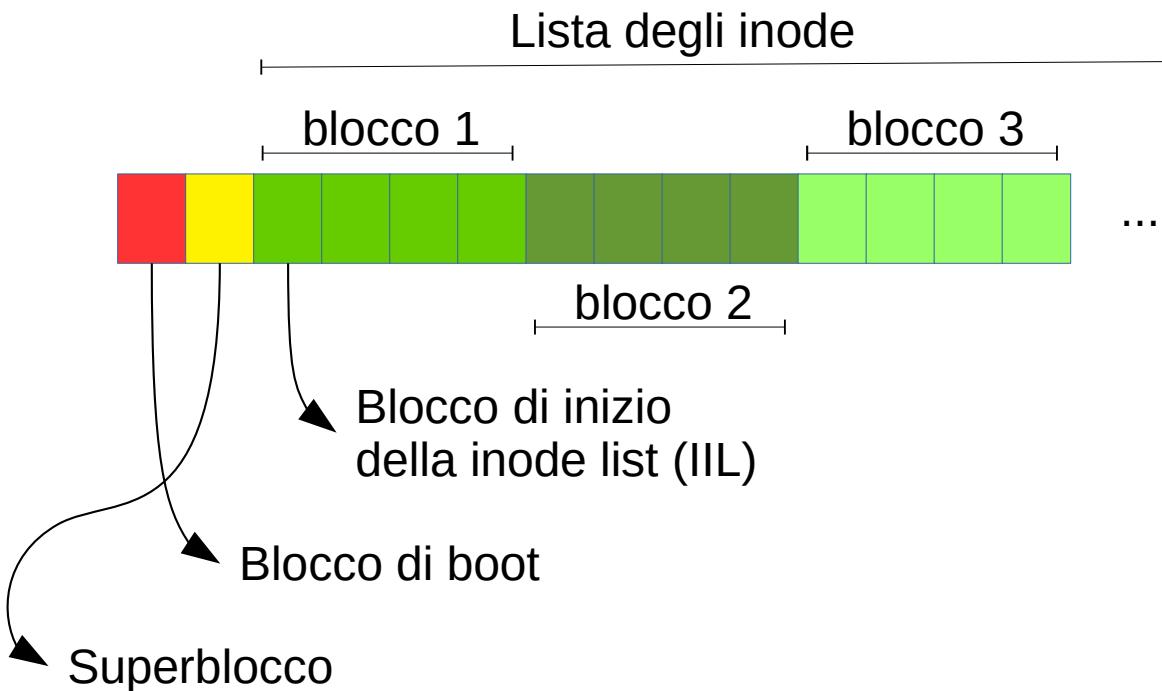
In Unix gli inode caricati sono detti **incore inode** e sono conservati in una tabella globale che si chiama **inode table**

Apertura di un file

- supponiamo che un processo esegua la sys. call `open("prog.c", O_RDONLY)`
- Il kernel mantiene una **inode table** di dimensione finita
- Nell'eseguire la open: controlla se l'inode corrispondente al file è già stato caricato, **se sì userà l'incore inode trovato** altrimenti:
 - Se c'è spazio, crea un **nuovo incore inode**, lo "locka" e va a copiarvi l'inode su disco corrispondente al file da usare
 - **FAIL** ← Se non c'è spazio l'operazione fallisce e viene restituito un errore (così il processo richiedente non rischia di rimanere sospeso per tempi lunghi)
 - *se esisteva già lo vedremo fra poco*
- **Nota:** il lock (esclusivo e obbligatorio) serve per evitare inconsistenze. Si attiva all'inizio di certe system call e si disattiva al loro termine



Esempio: UNIX



... La lista degli inode è contenuta in una **sequenza di blocchi**

Gli inode hanno dimensione fissa

Tutti i blocchi contengono lo stesso numero di inode

Esempio #inode-per-blocco: 4

Per accedere alla porzione di disco che memorizza un inode basta conoscerne: 1) il numero, 2) la dimensione degli inode (la stessa per tutti) e 3) quella dei blocchi

Il SO ha tutte queste informazioni

Proviamo a identificare l'indirizzo di inizio dell'inode con indice 5 (partendo a contare dall'indice 0)

Esempio: UNIX



L'inode è identificato da un indirizzo $\langle B, D \rangle$:

B = blocco
S = displacement nel blocco

- #inode-per-blocco: 4
- inode-number: 5
- B: IIL + 1
- D: 1 x dim-inode

[1] Individuare il blocco che contiene l'inode

$$B = (\text{inode-number} / \# \text{inode-per-blocco}) + \text{IIL}$$

/ : divisione intera

[2] Individuare il displacement dell'inode nel blocco

$$D = (\text{inode-number} \% \# \text{inode-per-blocco}) \times \text{dim-disk-inode}$$

\% : resto della divisione intera

Algoritmo NAMEI

- Nei lucidi precedenti abbiamo implicitamente supposto di avere a disposizione il numero dell'inode corrispondente al file di interesse
- Di solito però avremo il **nome del file** ... o più in un cammino, es.

mieiFile/Documenti/anno2020/slidesisOp.odp

- Per identificare l'inode relativo a un file devo avere a disposizione un algoritmo che è in grado di percorrere il cammino fino a identificare il numero dell'inode:

- 1) accedo alla directory *mieiFile*
- 2) cerco fra i suoi contenuti *Documenti*
- 3) accedo alla directory *Documenti*
- 4) cerco fra i suoi contenuti *anno2020*
- 5) accedo alla directory *anno2020*
- 6) cerco fra i suoi contenuti
slideSisOp.odp
- 7) trovando il numero di un inode

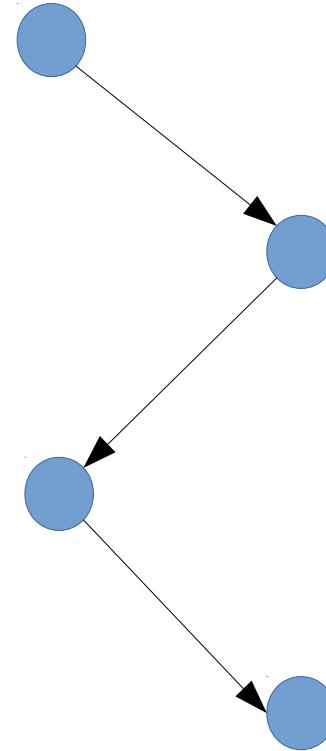
Algoritmo NAMEI

inode-number NAMEI(string cammino)

```
if (la prima directory del cammino è "/")
    current = root inode
else current = inode della working directory
```

```
repeat
    el = leggi prossimo elemento dall'input;
    if (el is null) return current;
    else if (el è contenuto in current)
        current = inode associato a el
    else return (no inode)
until (el is null);
```

```
return current;
```



Per verificare se el è contenuto in current occorre accedere ai blocchi dati relativi a current

Open di un file in Unix

- Un processo esegue **open(pathname, flags)**:
 - Il S0 verifica se il file è in uso da parte di qualche processo;
 - **Se no:**
 - (1) Utilizza l'algoritmo **namei** per identificare il numero di **inode** del file
 - (2) Calcola l'indirizzo su disco **<B, D>**, che permette di accedere **all'inode** conservato su disco
 - (3) Invia al **controller** del disco un comando di lettura che si concluderà con la copiatura dell'inode in una entry della tabella degli inode in RAM (**in-core inode**)
 - (4) Aggiorna le **tabelle di sistema** di conseguenza
 - **Se sì:**
 - (1) Identifica l'in-core inode e lo rende accessibile al processo che ha eseguito open modificando le **tabelle di sistema** contenute in RAM

Open di un File (UNIX)

La system call `open("prog.c", O_RDONLY)` restituisce come handle del file un **file descriptor**, cioè un numero intero. Cosa rappresenta questo numero?

In Unix vengono mantenute in RAM 3 tavelle:

- 1) **tabella degli incore inode (globale)**: contiene gli inode dei file aperti
- 2) **tabella dei file (globale)**: contiene un riferimento alla tabella precedente, un contatore e un puntatore alla locazione del file a cui si è arrivati a leggere (o in cui scrivere)
- 3) **tabella dei file del processo (locale, 1 tabella per ogni processo)**: contiene riferimenti alla tabella (globale) dei file, uno per ogni file aperto, più 3 riferimenti presenti per ogni processo, anche se non ha file aperti:
 - (a) *standard input*
 - (b) *standard output*
 - (c) *standard error*

Tabella dei file del processo

- Alla creazione di un processo viene creata la **tabella dei file aperti** specifica del processo
- Tale tabella contiene 3 entry corrispondenti a **stdin**, **stdout**, **stderr**:

0	stdin
1	stdout
2	stderr

Flussi standard che permettono l'interazione del processo con il suo ambiente (default: lettura da tastiera, scrittura su terminale, scrittura su terminale)

Tabella dei File del
processo

Open

la system call **open(...)** restituisce come **handle del file** un **file descriptor**, cioè un numero intero

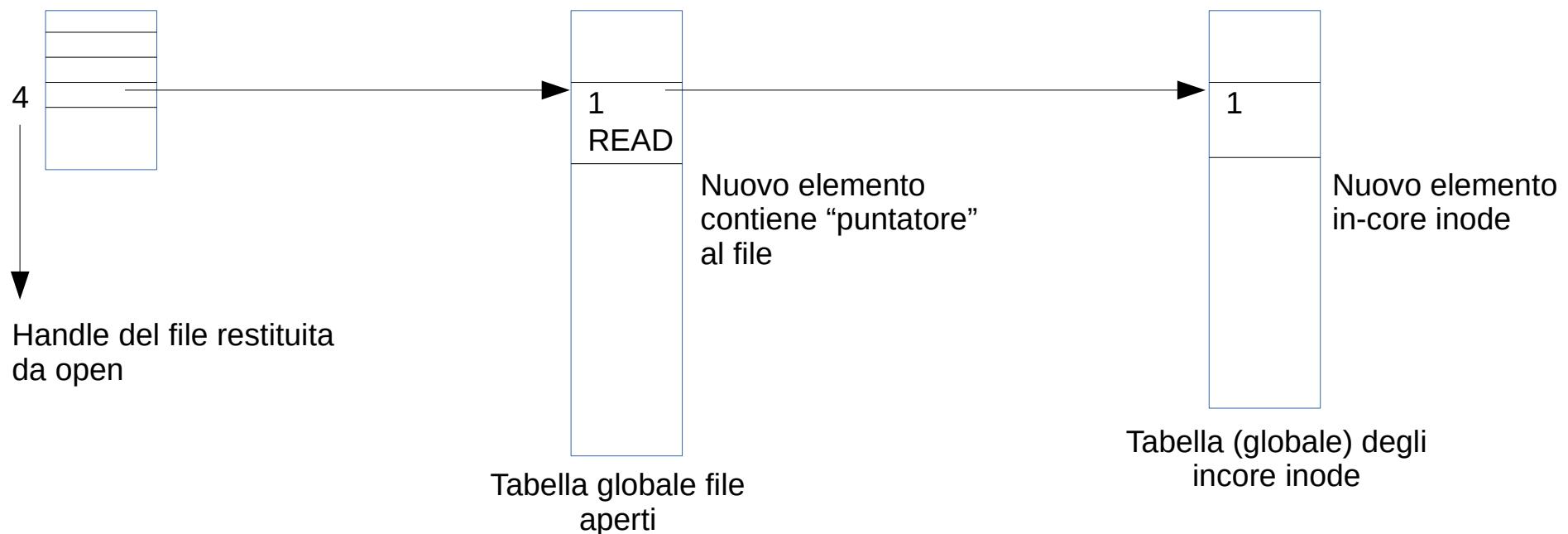
Per ogni open:

- (1) Se il file era già stato aperto si aggiorna la entry corrispondente nella **tabella degli inode (globale)**, altrimenti si crea una nuova entry
(NB: se non c'è spazio: interrupt);
- (2) Viene creata una entry nella **tabella dei file (globale)**:
conterrà un riferimento all'inode più un displacement che indica l'indirizzo della prossima lettura (o scrittura). Il default per il displacement è zero (file aperto nel suo punto di inizio); è invece uguale alla dimensione del file se l'apertura avviene in modalità write/append
- (3) Viene aggiunta una entry alla **tabella dei file (privata del processo)** che esegue la open. Questa entry contiene un riferimento alla entry nella tabella dei file globale.

NB: il suo indice in tabella è il file descriptor restituito da open

Il processo esegue una open

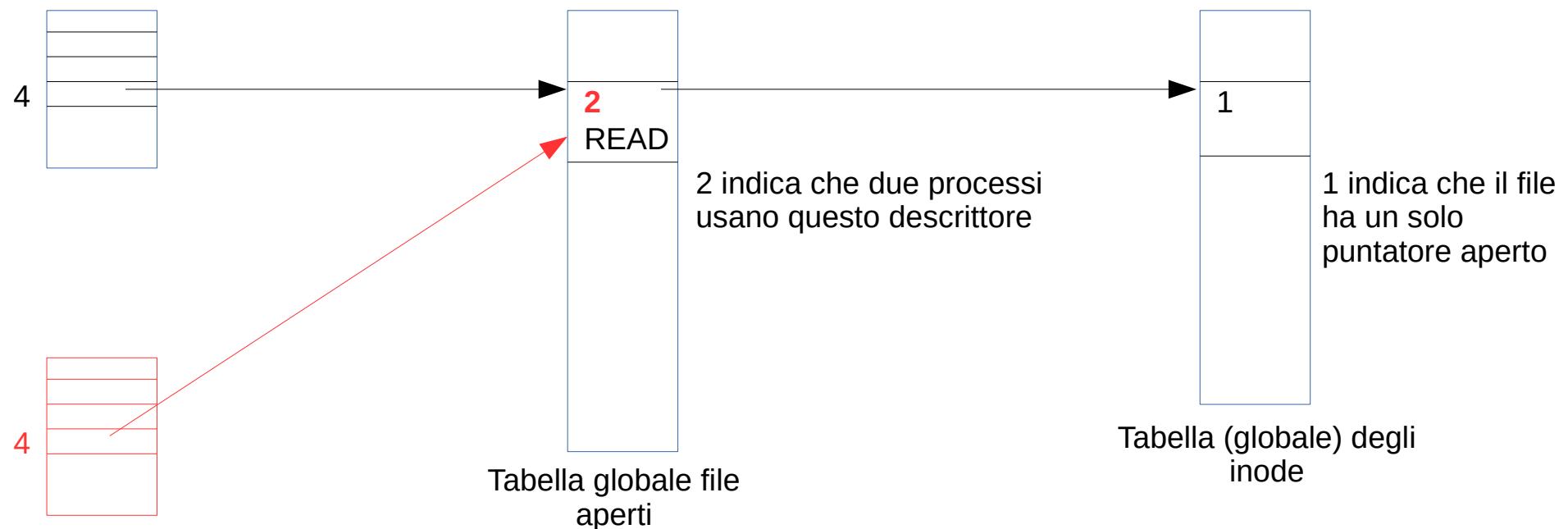
- Il processo esegue `open(0_RDONLY, "prova")`. Supponiamo che il file **non** sia in uso da parte di altri processi: il suo inode deve essere caricato in RAM



Ricorda: queste tabelle hanno una capienza limitata! Potrebbe non esseri spazio per un nuovo descrittore!

Se un processo Figlio è creato

- Il processo esegue fork: il figlio eredita i descrittori dei file aperti contenuti nella tabella globale dei file aperti perché la sua tabella locale dei file è una copia di quella del padre

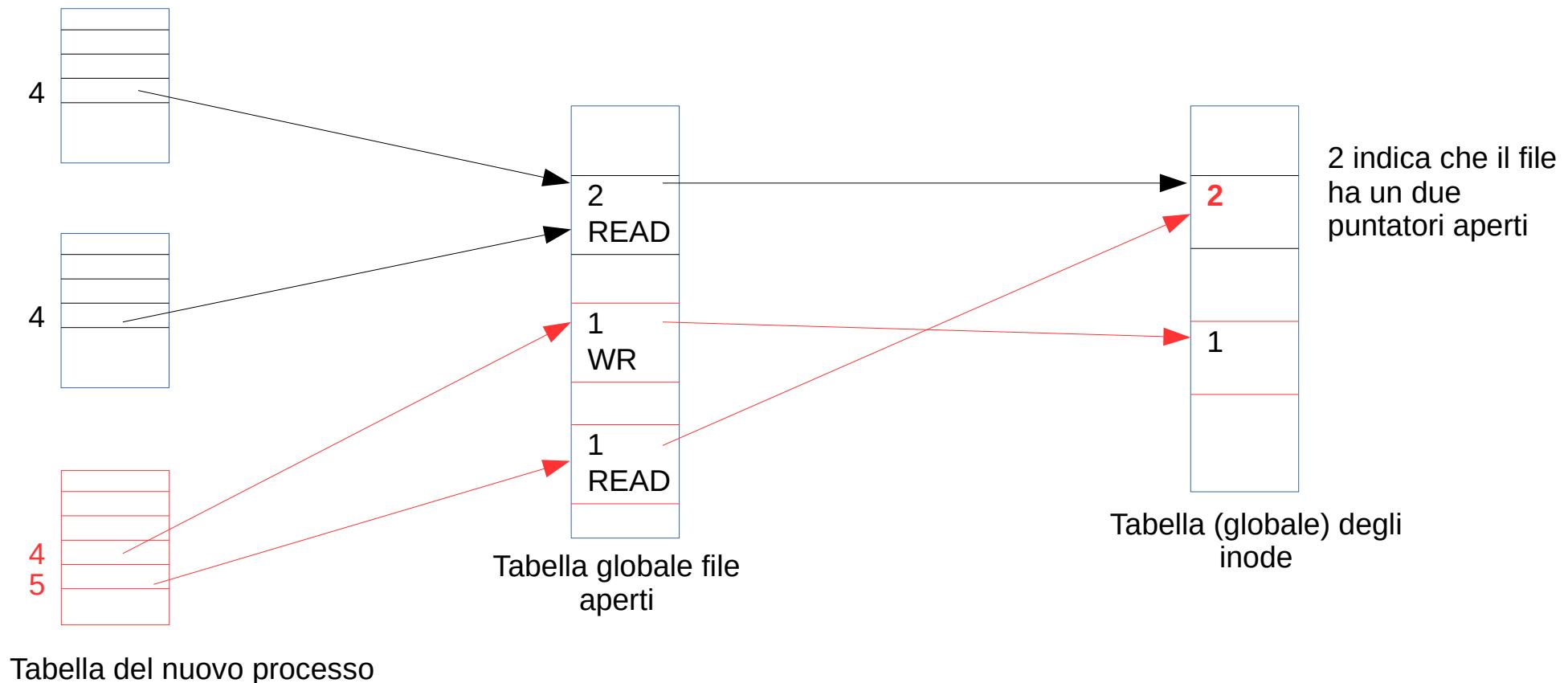


Handle del file restituita
da open

25

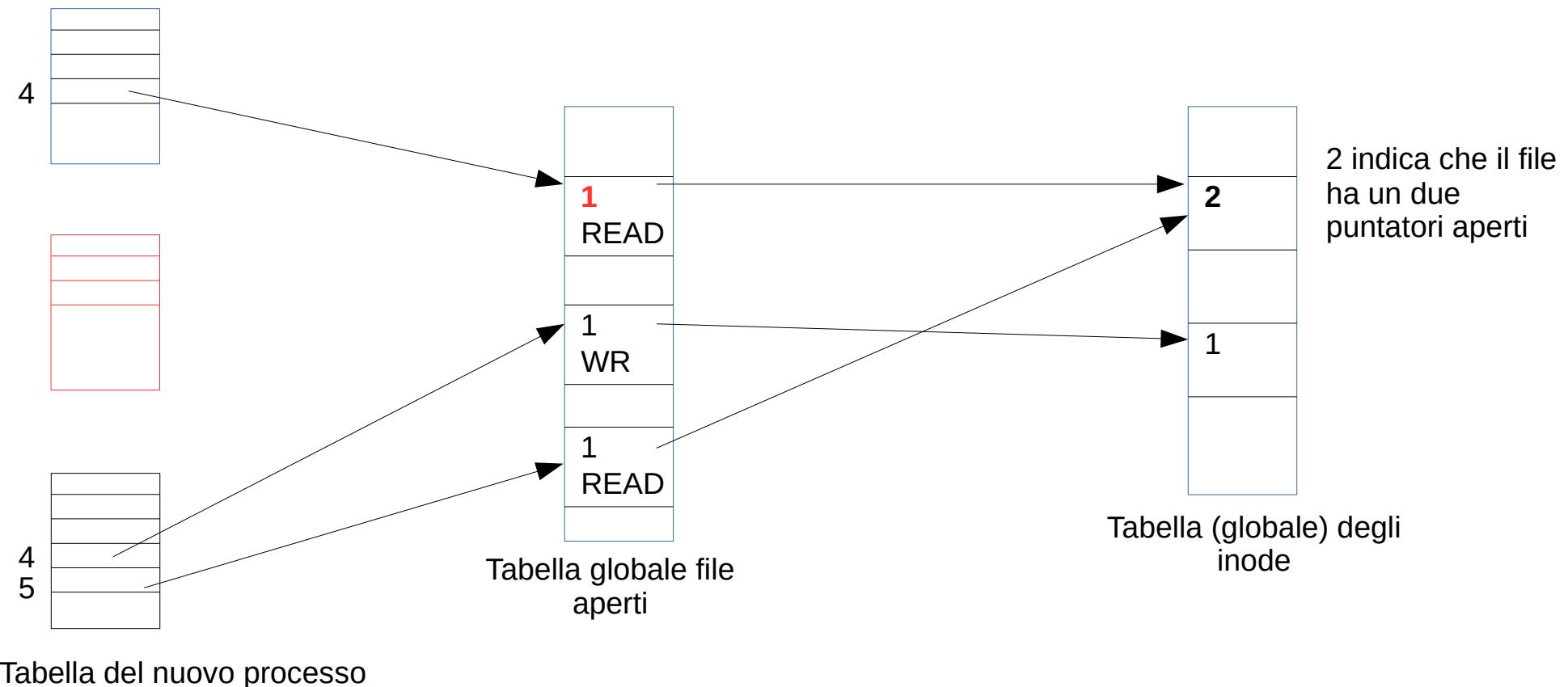
Un altro processo esegue open

- Un processo non in relazione di parentela con i precedenti esegue open(0_RDONLY, "prova"), dopo averne aperto un altro



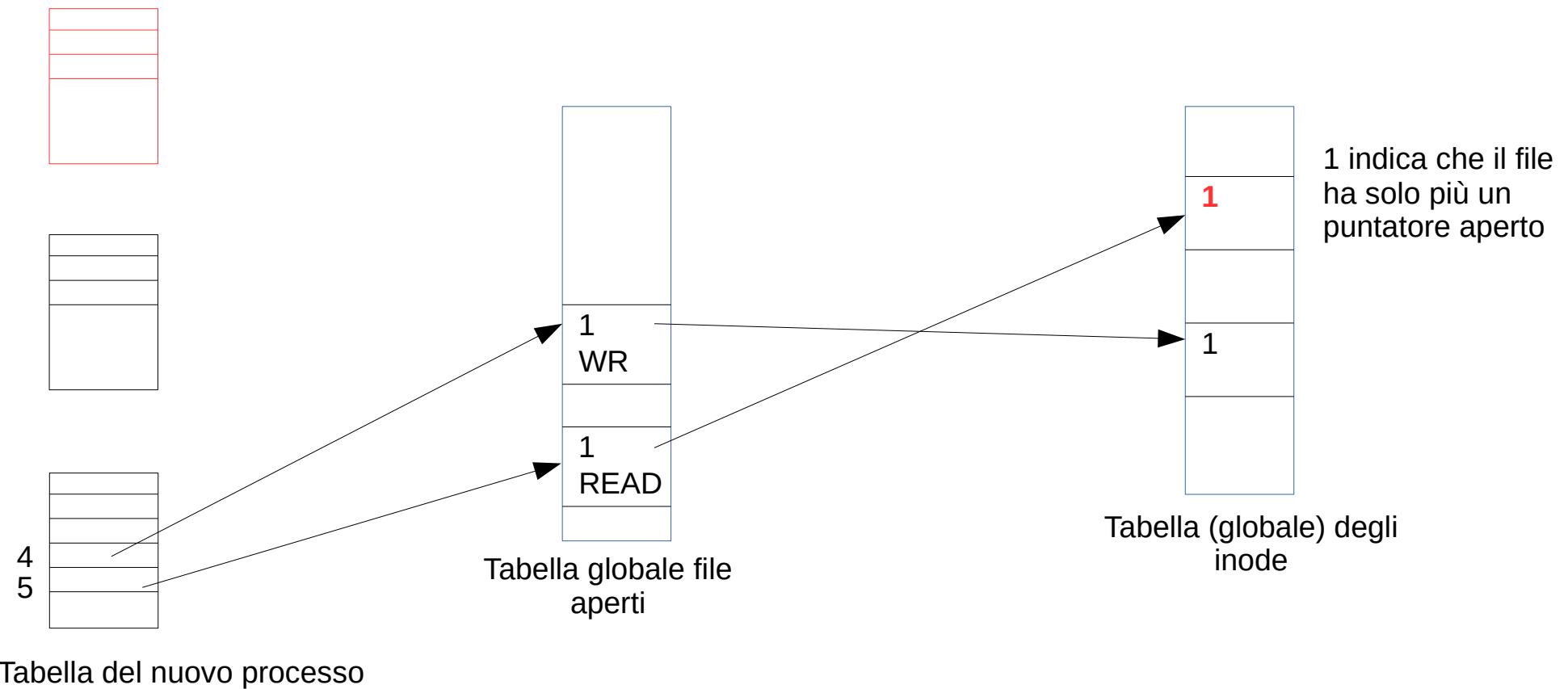
Il processo figlio esegue close

- Il processo figlio esegue `close(4)`. Il puntatore in lettura NON viene rimosso perché è ancora usato dal processo padre



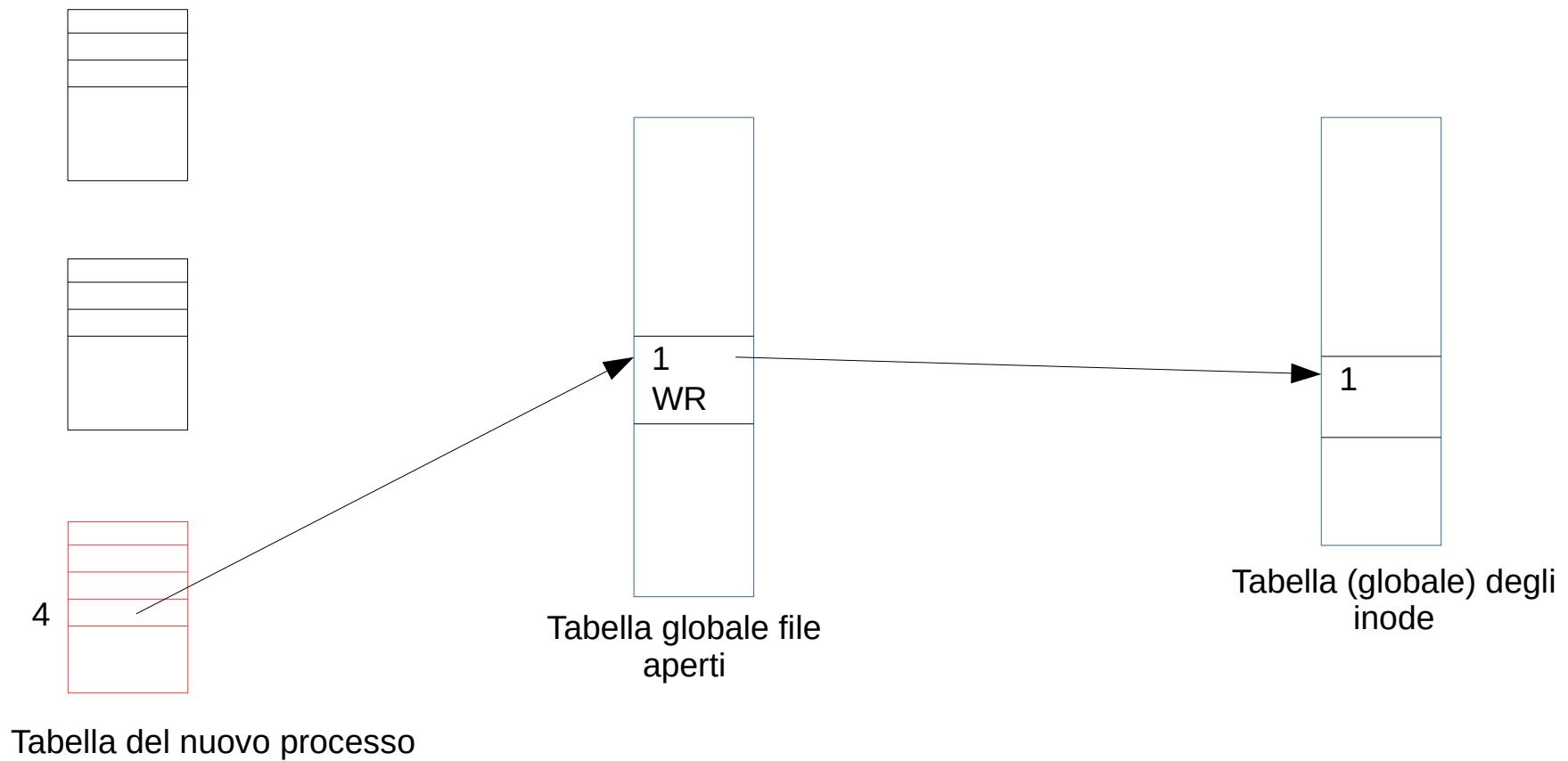
Il processo padre esegue close

- Il processo padre esegue `close(4)`. Il puntatore in lettura VIENE rimosso perché serve più. Il numero di riferimenti all'inode è decrementato



Il terzo processo esegue close

- Il terzo processo esegue `close(5)`. Anche l'inode è rimosso perché nessun processo usa più il file. I frame di RAM occupati dai suoi dati sono liberati.



Implementazione delle directory

- La scelta di come implementare le directory è un punto critico nella progettazione di un File System
- Fra le varie tecniche:
 - **lista lineare (alla Unix)**: una directory è una sequenza di coppie <nome_file, inode number>
 - **B-tree**
 - **tabella hash**

Sequenza lineare

Di facile realizzazione

Limiti

- per verificare se un file è contenuto nella directory occorre scorrere il contenuto della directory
- ricerca di tipo lineare (lenta)
- è difficile mantenere ordinata la lista senza appesantire la gestione delle directory
- occorre avvalersi di strutture d'appoggio e implementare particolari algoritmi per gestire i buchi che si creano nelle directory con la
- cancellazione di file

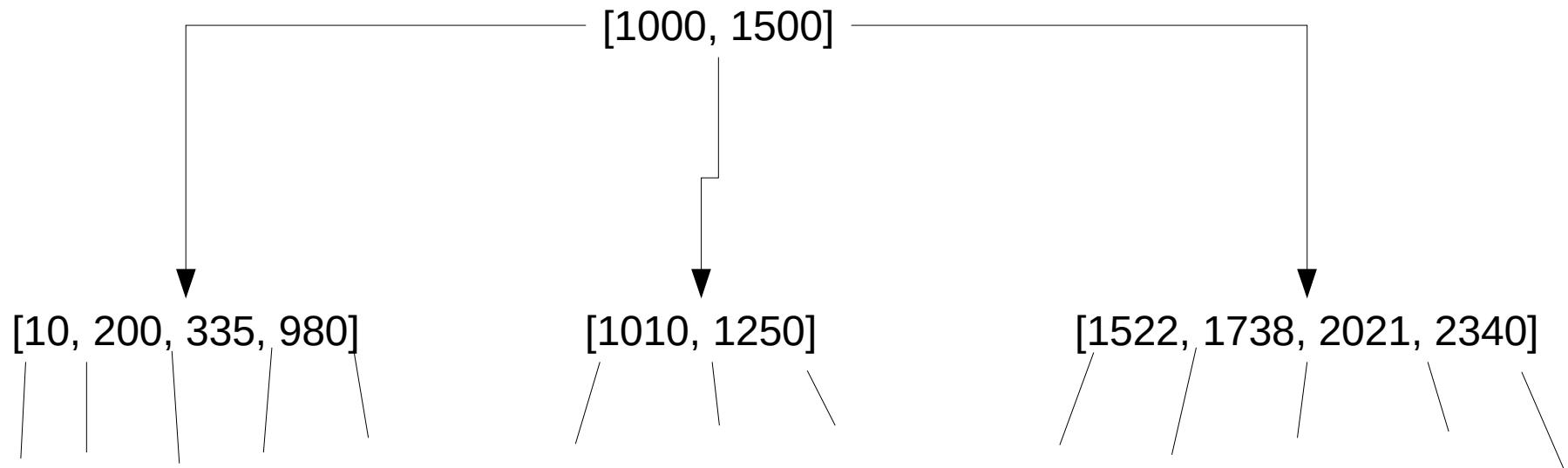
Possibili migliorie?

- appoggiarsi a strutture non lineari, in particolare alberi

B-Tree

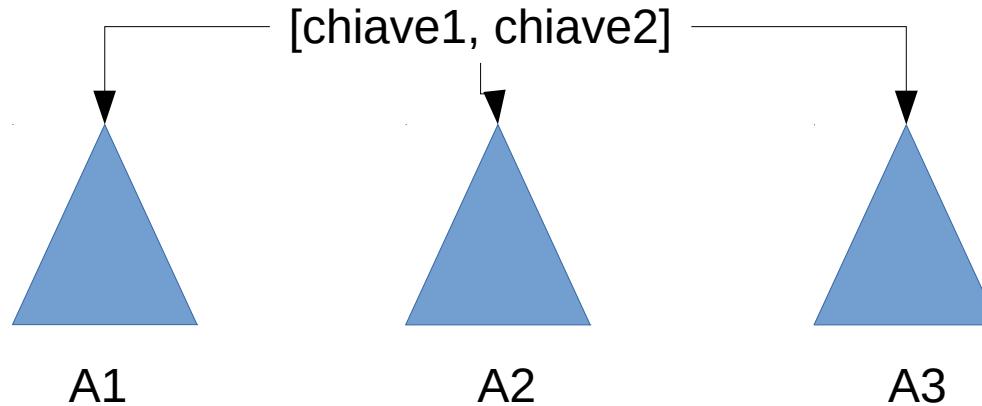
- un B-tree è una struttura ad albero ordinato.
- Ogni nodo contiene da N a $2N$ elementi detti **chiavi**.
- Se un nodo contiene K elementi, allora avrà **$K+1$ puntatori** a nodi del livello successivo.
- Il numero N è detto **ordine dell'albero**. Per esempio il successivo è un albero di ordine 2 ...

B-Tree di ordine 2: esempio



Se i numeri fossero identificatori di inode con associati i relativi metadati questo albero potrebbe rappresentare i contenuti di una directory

In generale



Tutte le chiavi del sottoalbero A1 sono minori di chiave1

Tutte le chiavi del sottoalbero A2 sono comprese fra chiave1 e chiave2

Tutte le chiavi di A3 sono maggiori di chiave2

In un FS le chiavi possono essere identificatori di inode; ad ogni identificatore è associato anche l'inode relativo

B-tree

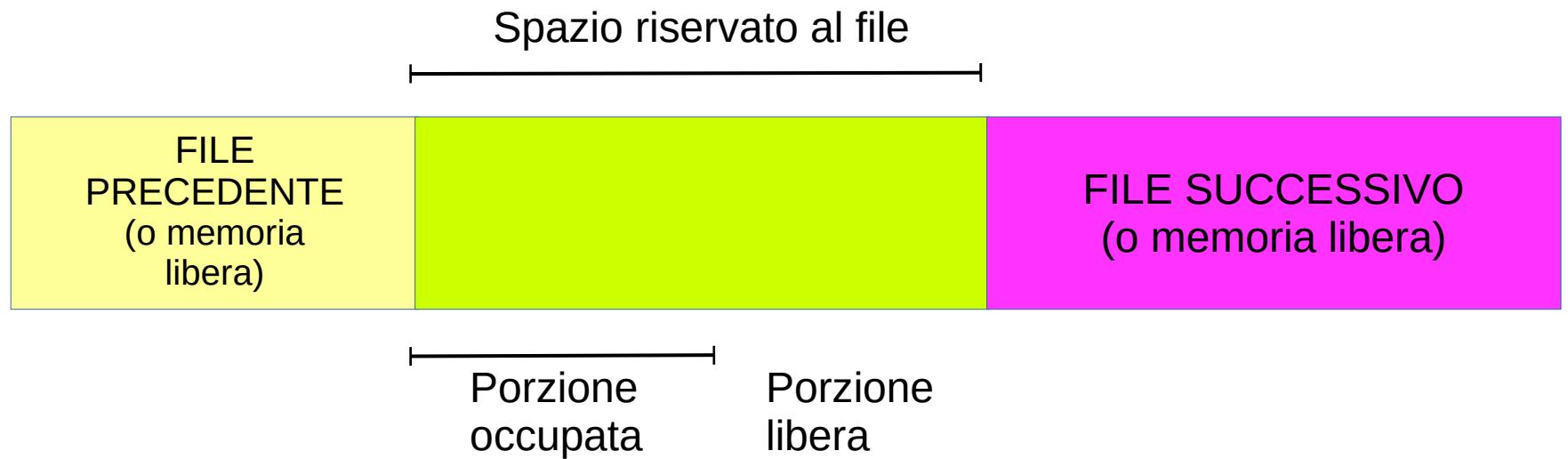
- La ricerca comincia sempre dalla radice del B-tree
- l'ordinamento dell'albero consente di trovare qualsiasi elemento in tempi rapidissimi per es.:
in un B-tree di ordine 25 con 10.000.000 di nodi possiamo individuare qualsiasi chiave attraversando al più 4 nodi, se l'albero è bilanciato
- Bilanciamento (#nodi sottoalberi sinistri = #nodi sottoalberi destri) e fan-out (apertura dell'albero) sono due caratteristiche fondamentali delle strutture ad albero usate per la ricerca

Allocazione dello spazio disco ai file

- Com'è organizzata la memorizzazione dei dati su disco?
 - allocazione contigua
 - allocazione concatenata (variante: FAT)
 - allocazione indicizzata

Allocazione CONTIGUA

Ogni file è allocato in una **sequenza contigua di blocchi**



Allocazione CONTIGUA

Ogni file è allocato in una **sequenza contigua di blocchi**

Vantaggi

- **rapidità di accesso al file**: il tempo di seek (posizionamenti della testina sulla traccia giusta) è trascurabile.
- Quando si accede a un file si tiene traccia dell'ultimo blocco letto, quindi se abbiamo appena letto il blocco B, sia l'**accesso sequenziale** (al blocco B+1) sia l'**accesso diretto** (al blocco B+k) sono immediati

Allocazione CONTIGUA

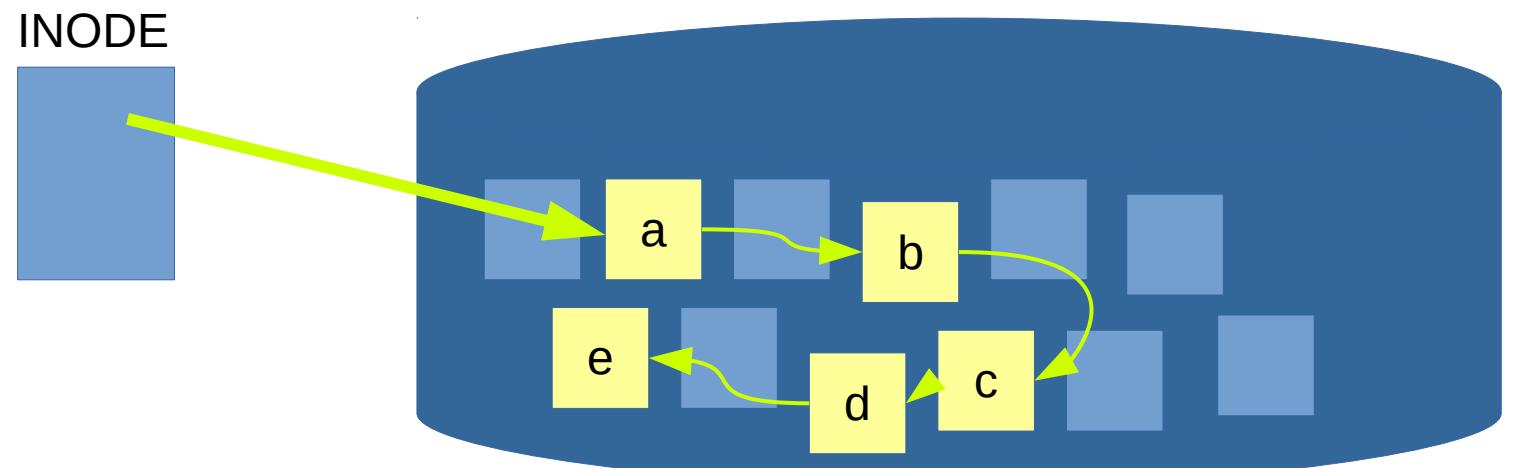
Ogni file è allocato in una **sequenza contigua di blocchi**

Svantaggi

- quanta memoria riservo per un file?
- se ne riservo troppa e il file cresce lentamente spreco memoria
- se ne riservo troppo poca? Necessità di estensioni!
- gestione dei buchi di memoria libera (best-, first-, worst-fit)
- frammentazione esterna
- necessità di deframmentare il disco di tanto in tanto

Allocazione concatenata

- **Alternativa:** spezzare il file in parti che possono essere allocate in modo non contiguo (blocchi dati). Occorre della sovrastruttura per mantenere l'informazione che certi blocchi dati sparsi sul disco costituiscono le parti di uno stesso file
- L'allocazione concatenata consente di fare proprio questo un file è costituito da una sequenza di blocchi sparsi per il disco ma mantenuti in una lista concatenata



Allocazione concatenata

Vantaggi

- non è necessario preallocare memoria per i file
- la lista concatenata è dinamica per natura
- non è necessario effettuare alcuna deframmentazione

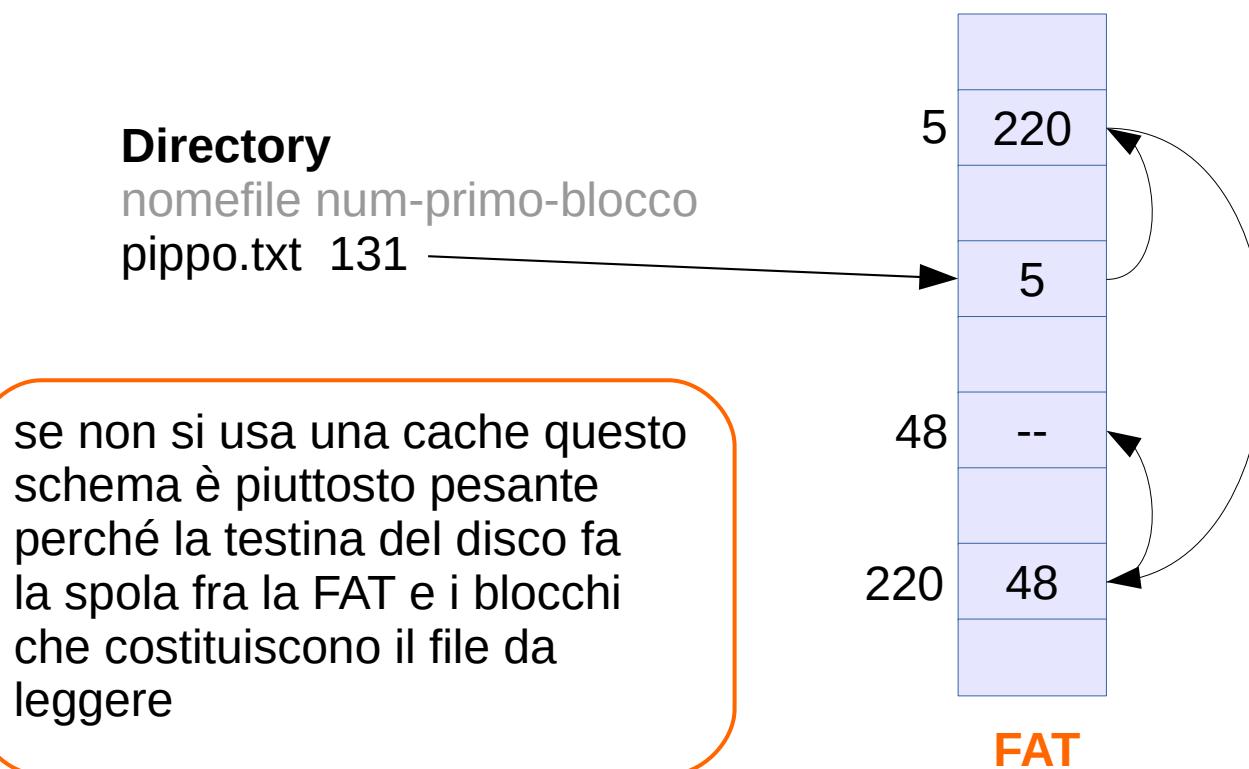
Allocazione concatenata

Svantaggi

- solo l'accesso sequenziale è efficiente, accesso diretto e indicizzato richiedono di scorrere la lista dei blocchi comunque
- i puntatori ai blocchi sono sparsi per il disco ogni salto di blocco comporta un tempo di latenza (tempo sprecato)
- occorre spazio per mantenere i puntatori ai blocchi successivi
- se un puntatore si corrompe si perde tutta la porzione successiva di file (!!!)

Allocazione concatenata: File Allocation Table (FAT)

- usata nei sistemi MS-DOS e successori
- si riserva una sezione della partizione per mantenere una tabella che ha tanti elementi quanti blocchi. Se un blocco fa parte di un file, il contenuto della entry corrispondente in tabella è un riferimento al blocco successivo:

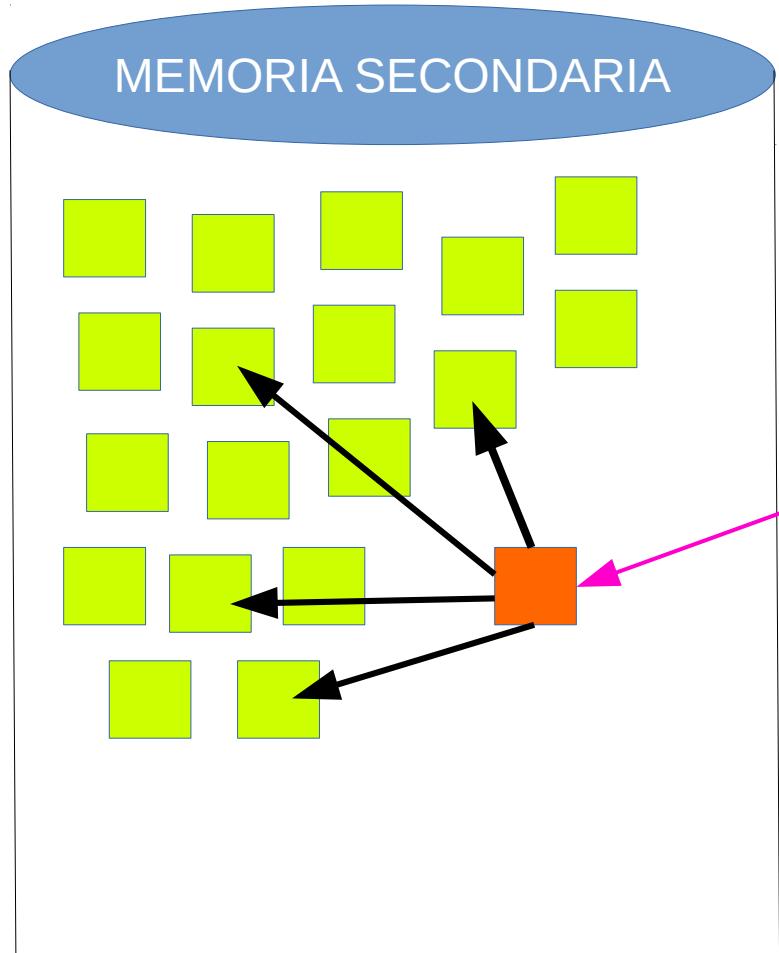


Una FAT mantiene la struttura a lista concatenata esternamente ai blocchi dei file veri e propri

Allocazione indicizzata

- Tipo di allocazione che tenta di superare i limiti delle soluzioni precedenti introducendo un blocco indice
- Ogni file ha un **blocco indice**: un array contenente gli indirizzi dei blocchi che costituiscono il file
- I riferimenti ai blocchi indice dei file sono mantenuti nelle **directory**
- **Accesso:**
 - tramite la directory recupero il blocco indice
 - tramite i riferimenti contenuti nel blocco indice posso accedere ai vari blocchi dati

Blocco indice



Directory

File1 bloccoindice1
File2 bloccoindice2
...

Quando il file viene creato, tutti gli elementi del suo blocco indice sono NULL

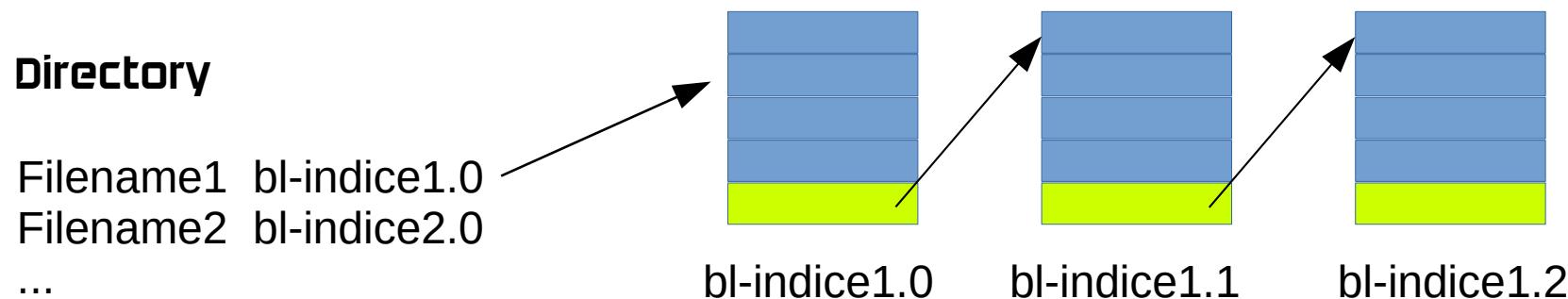
Man mano che il file cresce di dimensioni vengono allocati nuovi blocchi e i loro riferimenti sono inseriti in ordine nel blocco indice

Blocco indice

- La soluzione a blocco indice **elimina il problema della frammentazione esterna** in modo del tutto analogo all'organizzazione della RAM in pagine di pari dimensione, allocate quando c'è bisogno
- Si può avere **frammentazione interna** però solo a livello dell'ultima pagina che costituisce il file
- L'unico problema è il **dimensionamento del blocco indice**:
 - se è troppo piccolo il file potrebbe richiedere più blocchi dati quanti è possibile riferire
 - se è troppo grande si ha frammentazione interna al blocco indice stesso (spreco di memoria)
 - **soluzione ideale**: poter ridimensionare il blocco indice a seconda delle circostanze ...

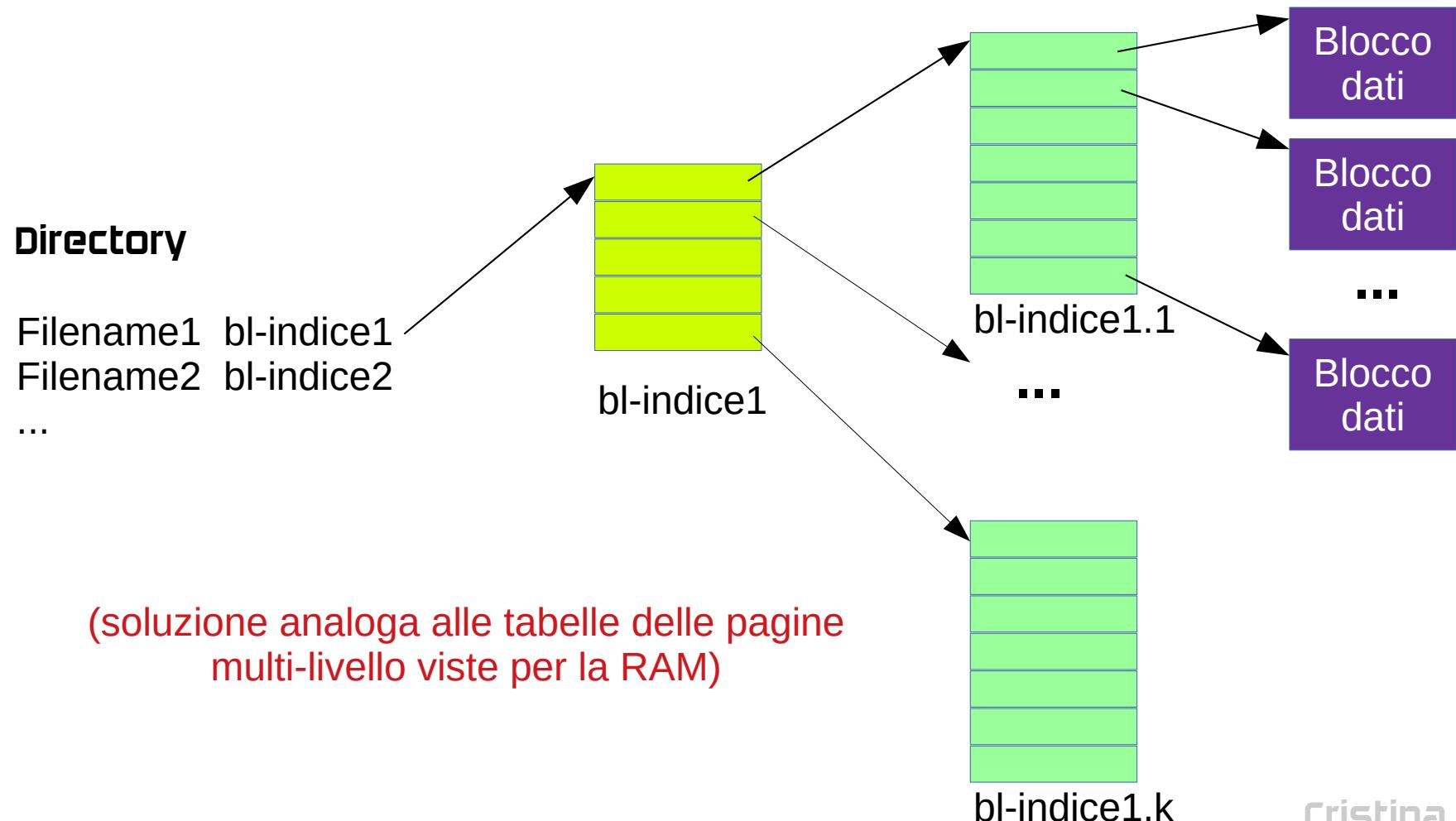
Implementazione a schema concatenato

Utilizzare blocchi indice piuttosto piccoli e interpretare l'ultimo indirizzo contenuto nel blocco indice come riferimento a un'estensione del medesimo, da usare solo se serve



Implementazione con indice a più livelli

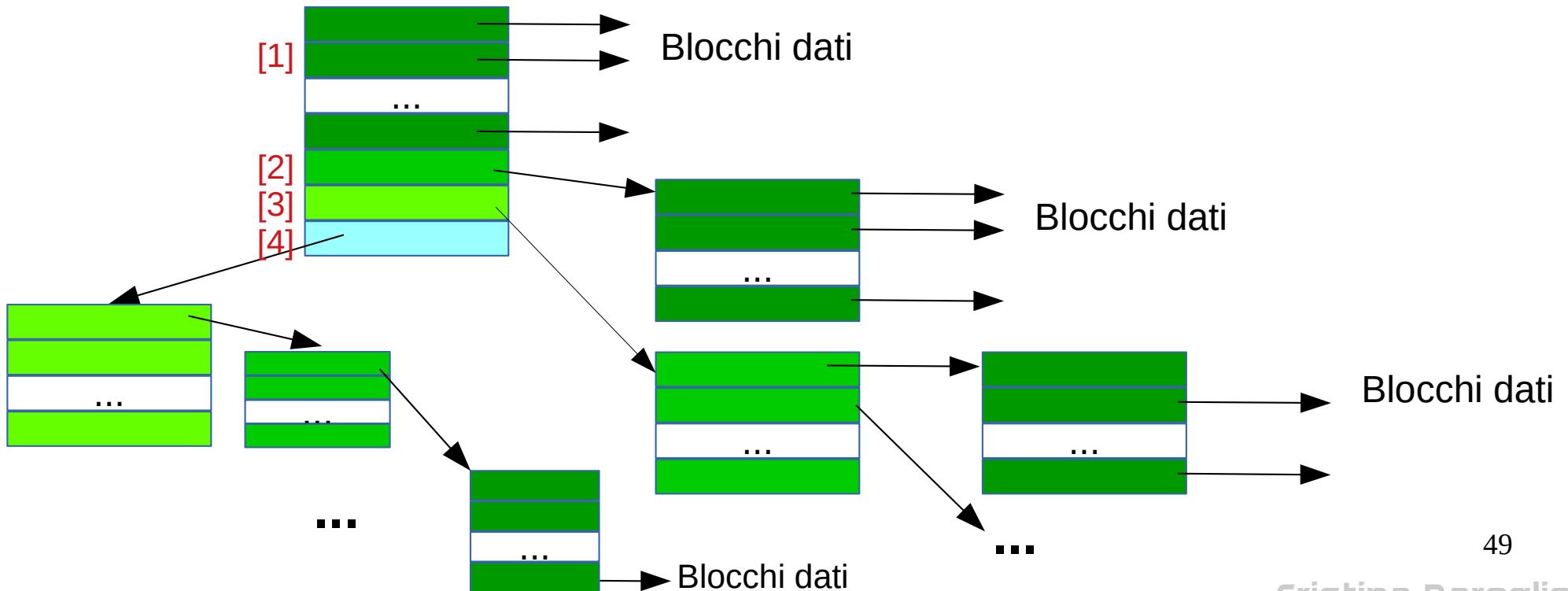
Struttura gerarchica. Il primo livello punta a una serie di blocchi indice, ciascuno dei quali consente l'accesso a blocchi dati. I blocchi indice/dati vengono aggiunti solo se serve



Soluzione ibrida

Viene mantenuta una tabella di accesso ai dati, una parte dei cui elementi consente l'accesso diretto a blocchi mentre un'altra parte consente l'accesso attraverso un indice a più livelli

Soluzione adottata in Unix (per esempio), dove la tabella è inclusa nell'inode



Soluzione ibrida

- [1] ogni entry è un riferimento a un blocco dati
- [2] ogni entry è un riferimento a una tabella di riferimenti a blocchi dati
- [3] ogni entry è un riferimento a una tabella di riferimenti a una tabella di riferimenti a blocchi dati
- [4] ogni entry è un riferimento a una tabella di riferimenti a una tabella di riferimenti a una tabella di riferimenti a blocchi dati

si hanno da 15 a 19 blocchi a indirizzamento diretto [2], [3] e [4] sono allocati solo se serve

Soluzione ibrida

Vantaggio

- con questa tecnica si possono costruire file che raggiungono dimensioni molto grandi (terabyte)

Svantaggio

- laddove si usino i livelli di indicizzazione indiretta questa tecnica soffre un po' dei limiti della tecnica di concatenazione

Commenti generali

- **Allocazione concatenata:** più adeguata a accessi sequenziali
- **Allocazione a indice:** più adeguata ad accessi diretti
- L'allocazione indicizzata richiede di mantenere in RAM una parte dei blocchi indice, possono occorrere due o più accessi al disco se la RAM non è sufficiente:
 - uno (o più) per accedere al blocco indice giusto
 - uno per raggiungere il dato di interesse
- Alcuni sistemi combinano allocazione contigua e indicizzata: finché il file rimane di piccole dimensioni si usa l'allocazione contigua, oltre un certo limite si comincia ad usare un indice

Gestione dello spazio libero

- Occorre una struttura che consente di tener traccia dei blocchi liberi
- **Vettore di bit**: viene mantenuto un array di bit, ognuno dei quali corrisponde a un blocco. Se il blocco è libero il bit è impostato a 1, es:

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

il quarto e quinto blocco sono liberi, idem dal settimo al decimo e così via

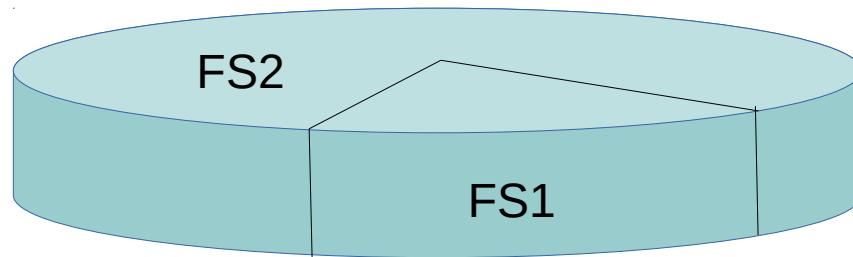
- NB: con questa soluzione è anche facile identificare sequenze di blocchi contigui liberi. Se so che mi servono tre blocchi liberi e li trovo contigui, meglio, perché i tempi di accesso saranno inferiori

Gestione dello spazio libero

- **Lista concatenata**: i blocchi liberi sono concatenati in una lista. Poiché i blocchi vengono allocati ai file uno per volta, basta pescare dalla testa della lista il primo blocco libero ed aggiornare il puntatore
- **Raggruppamento**: concatenazione di blocchi indice. Si usa un blocco libero (di dimensione N) per mantenere N-1 puntatori ad altrettanti blocchi liberi. Si usa l'ultimo puntatore per fare riferimento a un eventuale ulteriore blocco dello stesso tipo
- **Conteggio**: variante del precedente, in presenza di sequenze di blocchi liberi contigui si mantiene un riferimento al primo di tali blocchi e il numero di blocchi liberi ad esso consecutivi. Es. <K, 4> dove K è il riferimento a un blocco libero e 4 il numero di blocchi liberi consecutivi a partire da esso

Quantità di memoria massima gestibile

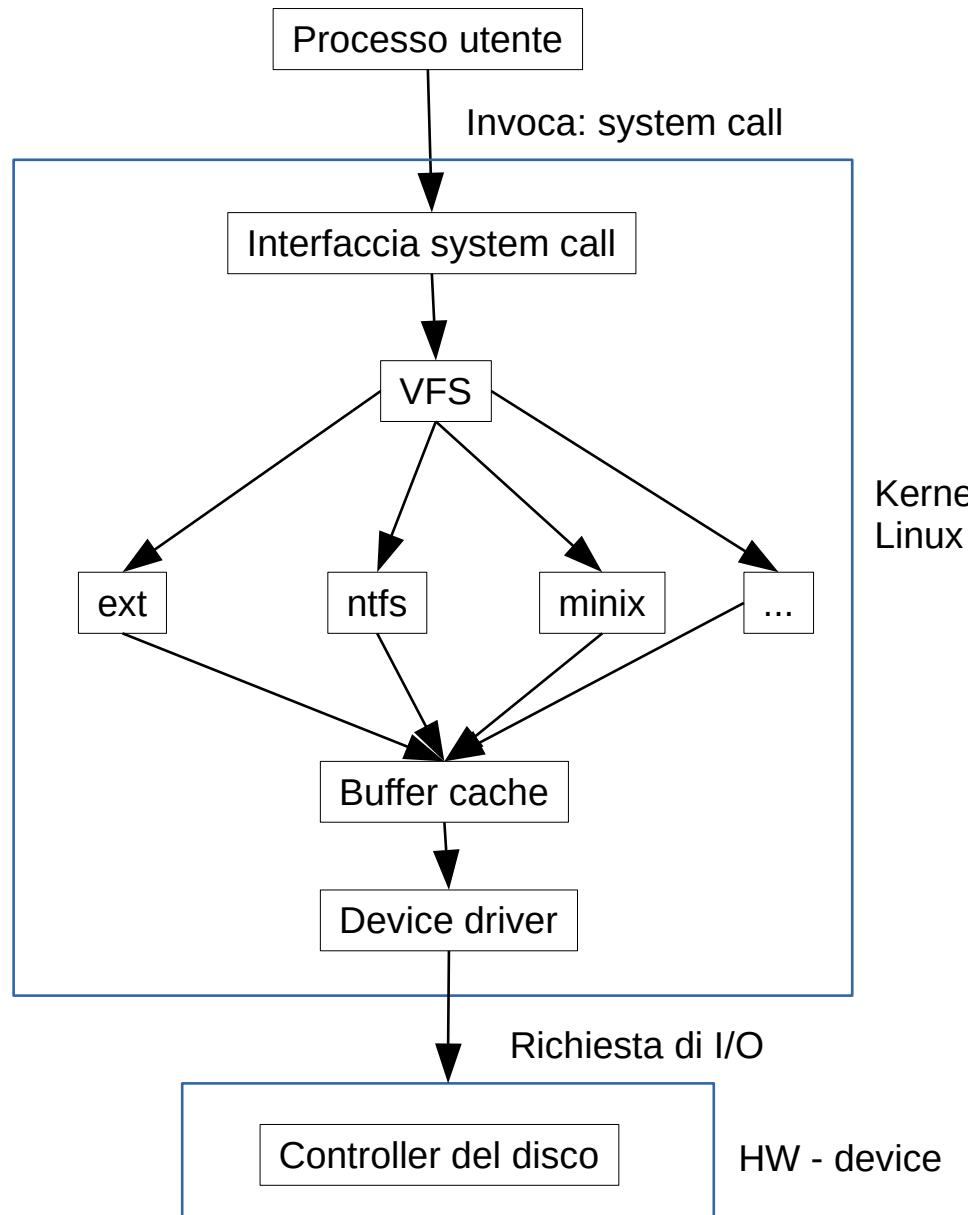
- La quantità massima di memoria gestibile da un file system dipende dalle strutture del file system
- **Esempio:** una FAT a 8bit consente di indicizzare al più 2^8 blocchi (256 blocchi), se ogni blocco è grande 1KB potrò al più gestire una memoria pari a 256KB
- Se devo gestire un disco di dimensioni superiori alla quantità di memoria gestibile da un FS dovrò partizionare il disco in FS diversi o una parte della memoria sarà inaccessibile!!



File System Virtuale

- Un disco può essere partizionato e ogni partizione può contenere FS di tipodifferenti, eppure è possibile montare i diversi FS in un'unica struttura a cui l'utente ha accesso attraverso lo stesso insieme di comandi e system call, senza accorgersi delle diverse implementazioni
- Analogamente è possibile accedere a FS di rete senza rendersi conto che i file usati risiedono su una macchina diversa da quella in uso
- Ciò è possibile perché l'interazione fra i processi e il FS è mediata da un'**interfaccia astratta** che permette di prescindere dallo specifico tipo di FS usato
- Questa interfaccia è il **Virtual File System**

File system virtuale



I processi utente invocano delle system call offerte dal VFS

Il VFS sta a un livello di astrazione più alto dei FS veri e propri ed esegue quella parte del lavoro che è indipendente dalla realizzazione fisica

Il VFS è un livello di indirezione che gestisce le sys. call che lavorano su file, preoccupandosi di richiamare le necessarie routine dello specifico FS fisico per attuare l'I/O richiesto

I diversi FS fisici devono specificare l'implementazione di tutte le sys. call offerte dal VFS

Oggetti di un VFS

- Il VFS gestisce **vnode**:
un vnode è un'**astrazione degli inode**, rappresenta un file, una directory, un link, un socket, un block o un character device, ecc.
- **NB**: mentre un inode ha un identificatore numerico unico per uno specifico FS, un vnode ha un id numerico unico per tutto l'insieme dei FS montati
- I vnode possono essere di diversi tipi a seconda di ciò che rappresentano, per esempio:
 - **superblock**: rappresenta un intero FS
 - **dentry**: rappresenta un generico elemento di una directory
 - **inode**: rappresenta un generico file
 - ...

Operazioni consentite da un VFS

- Un VFS definisce e mette a disposizione del livello utente un insieme di operazioni (**system call**) di base, fra cui troviamo per esempio:
 - open di un file
 - close di un file
 - read/write di un file
- uno specifico FS fisico, es. ext3 o winFS, per essere gestibile attraverso il VFS deve implementare tali funzioni, esattamente come accade per le interfacce in un linguaggio object-oriented.
- L'implementazione è talvolta indicata con il termine: **driver del file system**

Buffer-cache?

- Occorre fare una distinzione fra due tipi di memorie d'appoggio, definiti in base all'uso che se ne fa:
 - **buffer**: è una memoria in cui i dati vengono inseriti in attesa di essere consumati. I dati in un buffer vengono usati una volta sola.
 - **cache**: è una memoria in cui i dati vengono conservati per un po' di tempo in previsione di ulteriori utilizzzi
- I dischi hanno associata una **memoria di appoggio in RAM** detta **buffer cache** ...

Buffer Cache

- **Passi di una scrittura su file:**
 - **Processo:** esegue una **write**
 - Il blocco da modificare, appartenente al file in questione, viene aggiornato **in RAM**
 - La modifica viene **propagata** al blocco del file conservato su disco
- Normalmente un'operazione di scrittura su di un file si conclude solo quando la modifica su disco è stata completata
- **Write-ahead:** meccanismo tramite il quale un'operazione di scrittura viene considerata conclusa al termine dell'aggiornamento della copia del blocco contenuta in RAM

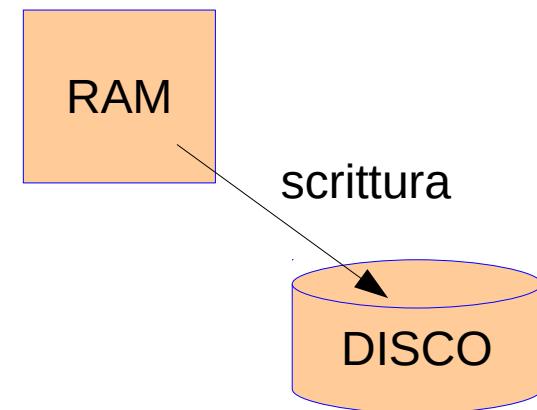
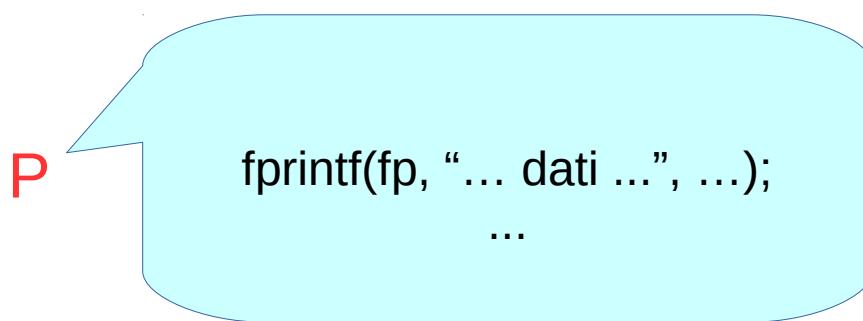
Read ahead – read behind

Quando il controller riceve una richiesta di lettura, individua un settore del disco e poi lo legge tutto (non solo il dato richiesto). La parte extra è inserita nella RAM e lasciata a disposizione per usi futuri.

Questo approccio applica una sorta di criterio di località negli accessi al disco è un'attività nota come **read ahead** o **read behind**

Buffer Cache

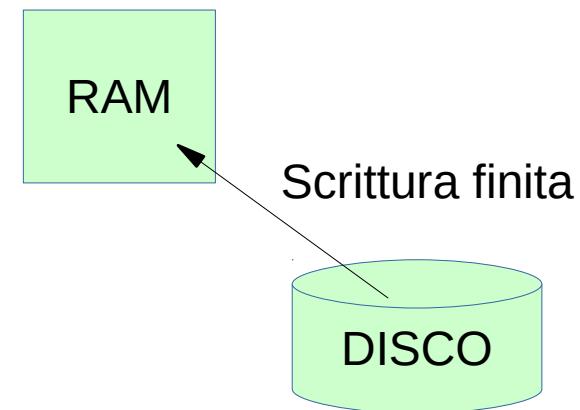
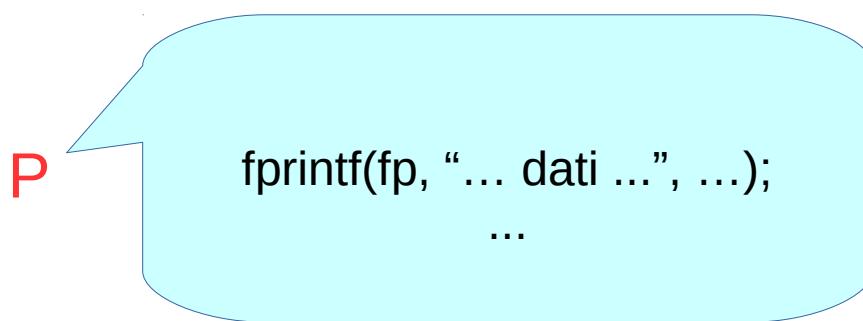
- Perché il write ahead incrementa l'efficienza?
- Consideriamo un processo che esegue la scrittura senza write ahead:



- Per eseguire questo codice il processo è nello stato **RUNNING**
- La scrittura è un'operazione di output, come tutte le operazioni di I/O fa cambiare lo stato del processo in **WAITING**

Buffer Cache

- Perché il write ahead incrementa l'efficienza?
- Consideriamo un processo che esegue la scrittura senza write ahead:



- P rimane **WAITING** finché non termina la scrittura su disco
- Dopo P diventa **READY**
- Infine dopo un tempo che dipende dallo scheduling della CPU, P torna **RUNNING** e prosegue la sua esecuzione

Buffer Cache

- Perché il write ahead incrementa l'efficienza?
- Consideriamo lo stesso processo che esegue la scrittura con il write ahead:

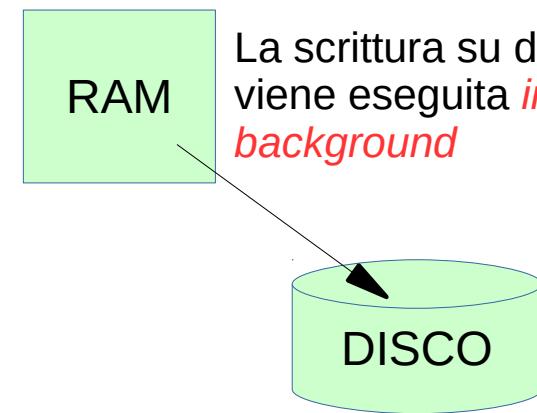
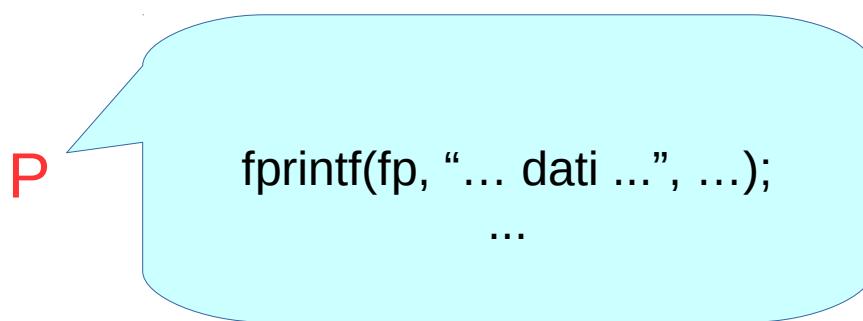
P

```
fprintf(fp, "... dati ...", ...);  
...
```

- Per eseguire questo codice il processo è nello stato **RUNNING**
- Poiché si usa il write-ahead, La scrittura è considerata terminata quando è completata in RAM
- Il processo continua la sua esecuzione senza cambiare stato

Buffer Cache

- Perché il write ahead incrementa l'efficienza?
- Consideriamo lo stesso processo che esegue la scrittura con il write ahead:



- Per eseguire questo codice il processo è nello stato **RUNNING**
- Poiché si usa il write-ahead, La scrittura è considerata terminata quando è completata in RAM
- Il processo continua la sua esecuzione senza cambiare stato

È necessario che l'interazione di scrittura con il device non richieda l'intervento della CPU (es. DMA)

Buffer Cache

- **Perché il buffer cache incrementa l'efficienza?**
- Consideriamo un processo che voglia leggere il file prima scritto da P:

P1

```
fscanf(fp, "... dati ...", ...);  
...
```

- Per eseguire questo codice il processo è nello stato **RUNNING**
- La lettura è un'operazione di input, dovrebbe far cambiare lo stato del processo a **WAITING**

Buffer Cache

- **Perché il buffer cache incrementa l'efficienza?**
- Consideriamo un processo che voglia leggere il file prima scritto da P:

P1

```
fscanf(fp, "... dati ...", ...);  
...
```

- Poiché si usa il buffer cache,
i blocchi dati modificati sono in RAM
- Il processo vi accede e continua la sua esecuzione
senza cambiare stato

Esempio

- Un utente scrive il programma **codice.c** usando **editor di testo** e **compilatore**. Supponiamo che l'utente modifichi il file, lo salvi e chiuda l'editor e solo dopo compili il file modificato:
 - L'**editor** è un esempio di processo che scrive un file
 - senza write ahead, ad ogni save l'editor rimarrebbe **WAITING** fino al termine della copiatura su disco
 - Con il write ahead a ogni save l'editor consente all'utente di continuare a lavorare non appena la richiesta di scrittura è stata notificata al controller del disco
 - La scrittura su disco vera e propria avverrà in background

Esempio

- Un utente scrive il programma **codice.c** usando **editor di testo** e **compilatore**. Supponiamo che l'utente modifichi il file, lo salvi e chiuda l'editor e solo dopo compili il file modificato:
- Il **compilatore** è un esempio di programma che legge un file per elaborarlo. Il file risiede su disco ma la presenza del buffer cache rende **più efficiente** l'accesso ai blocchi dati:
 - se **codice.c** è appena stato salvato, anche se è stato chiuso dal processo editor, i suoi blocchi dati sono (probabilmente) ancora nel buffer cache
 - non serve aspettare il caricamento del file da disco per effettuare la compilazione

Ripristino del file system

- Quando si avvia un elaboratore, il SO compie una verifica della **consistenza del file system**. **FSCK** (**file system check**): è una routine che controlla se all'ultimo spegnimento il FS è stato smontato (operazione di unmount) correttamente.
- L'**unmount** forza l'esecuzione delle scritture pendenti
- Il modo in cui viene effettuato il controllo dipende dal FS. Es. in ext2 si verifica un campo del superblocco, che può assumere i valori **clean** e **unclean**
 - 1) quando si fa il mount del FS il campo è settato ad unclean
 - 2) quando si fa l'unmount viene settato a clean
 - 3) se c'è un crash, il campo rimane unclean

Ripristino del file system

- Se fsck scopre che il FS non è stato smontato correttamente, avvia un controllo minuzioso (che non studiamo) del disco intero, individuando le inconsistenze e risolvendole per quanto possibile
- **La durata di questo controllo dipende dalla dimensione del disco**
- **PROBLEMI:**
 - Se il problema si verifica su un server che mantiene i dati di qualche servizio critico (es. banca, compagnia di volo, ...) il tempo trascorso nella verifica è un costo perché durante tutto il tempo il FS è off-line e non può essere usato
 - altro problema: transazioni. Se il crash ha interrotto una transazione, occorrerà avviare delle operazioni di ripristino di uno stato consistente
- **Soluzione:** mantenere qualche informazione in più che consenta di andare ad effettuare verifiche mirate; perché controllare porzioni di file modificate un'ora fa? Soltanto le modifiche effettuate poco prima del crash possono non essere state applicate ...

Journal

- Il **journal** è un **logfile** mantenuto su disco, in cui si tiene nota di tutte le operazioni di scrittura su file
- viene gestito in modo simile ai logfile per le transazioni quando un processo effettua una scrittura, prima il SO registra l'operazione che si va ad effettuare sul logfile, poi esegue la scrittura vera e propria del dato su file

Journaling del file system

- Le annotazioni presenti nel **journal** consentono di focalizzare il recovery sui soli file su cui sono state effettuate scritture **negli ultimi istanti prima del crash**
- Prima viene scritta l'annotazione nel journal, poi viene modificato il file oggetto della scrittura
- Consente di distinguere fra i seguenti casi, a seconda del momento in cui è avvenuto il disastro da recuperare:
 - **Crash successivo alla scrittura su disco del file:** l'annotazione nel journal corrisponde a quanto presente nel file, nessuna azione viene intrapresa
 - **Crash occorso fra la scrittura del journal e la scrittura su disco del file:** l'annotazione nel journal non corrisponde al contenuto del file, il contenuto del file viene reso consistente rispetto all'annotazione
 - **Crash occorso prima dell'annotazione nel journal:** la scrittura non ha mai avuto luogo

Journaling

Vantaggio

- La durata del ripristino non dipende dalle dimensioni del FS, bastano pochi secondi per riportare la consistenza dei dati

Svantaggio

- La gestione del journal appesantisce la gestione delle operazioni sui file. Nonostante ciò oggi come oggi i journal sono considerati essenziali

memoria secondaria

capitolo 12 del libro (VII ed.)

wikipedia:
GRUB, LILO, NTLDR, RAID

Introduzione

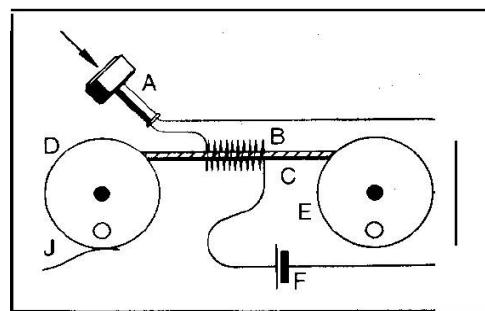
- dopo aver visto il FS dal punto di vista logico e dal punto di vista delle strutture di SO necessarie per la sua gestione, vediamo la struttura dei supporti di memorizzazione su cui il FS risiede
- Vari tipi di supporti
 - memoria secondaria: dischi magnetici
 - memoria terziaria: dischi rimovibili (ottici o magnetici), flash memory (eventualmente accessibile tramite supporto USB), nastri magnetici, ...

Un po' di storia

- La memoria su supporto magnetico viene proposta nel 1888, ad opera di **Oberlin Smith**, come supporto per la registrazione della voce
- solo nel 1898, **Valdemar Poulsen** brevetta il primo registratore magnetico e lo chiama *telegrafono*
- nel 1928 **Fritz Pfeumler** progetta il primo registratore su nastro magnetico
- In origine tutte le registrazioni su supporto magnetico sono di tipo analogico, cioè invece di registrare bit venivano registrati valori compresi in un intervallo continuo
- Oggi la memoria secondaria degli elaboratori è costituita da uno o più dischi magnetici su cui i dati sono registrati in formato digitale



Oberlin Smith



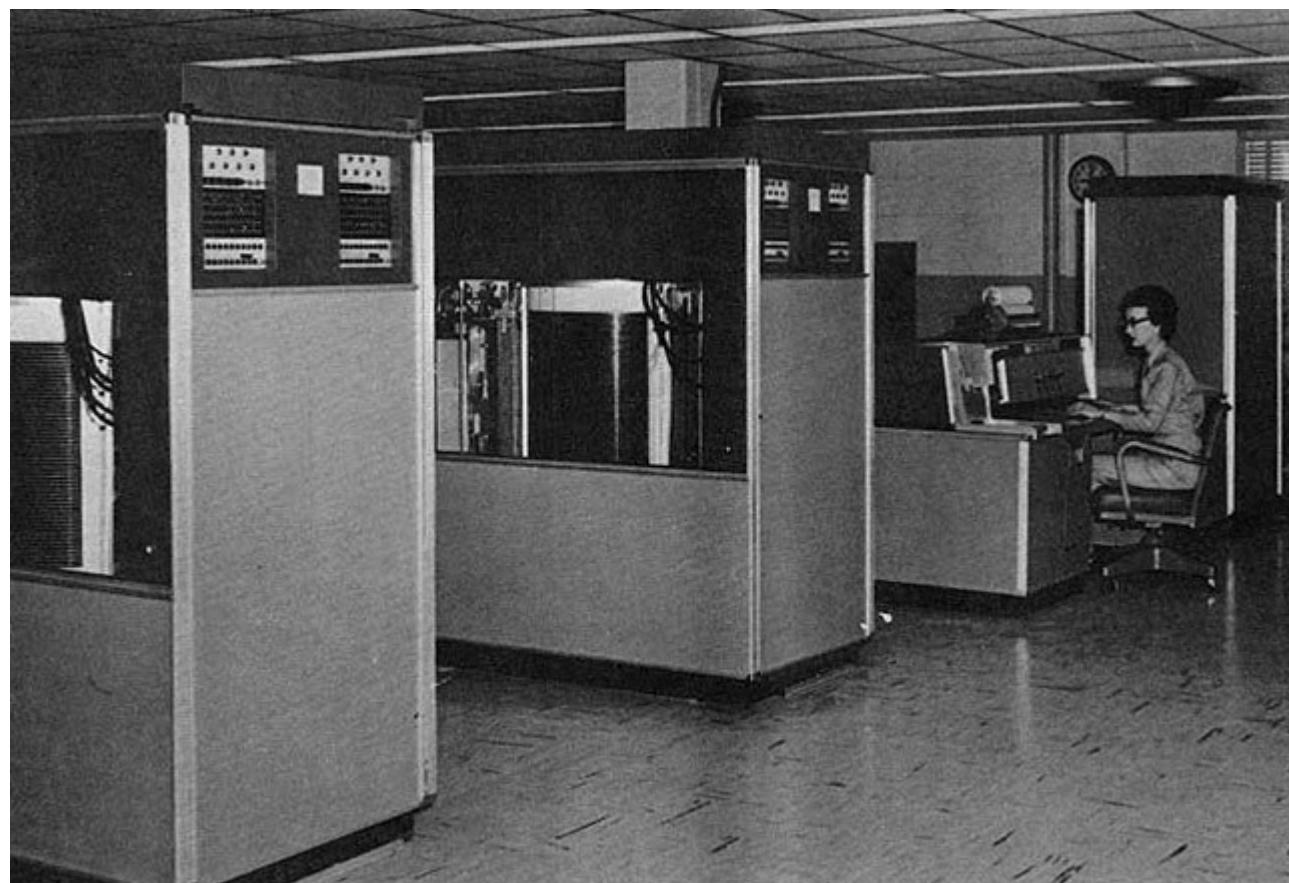
il suo progetto



il telegrafono di Poulsen

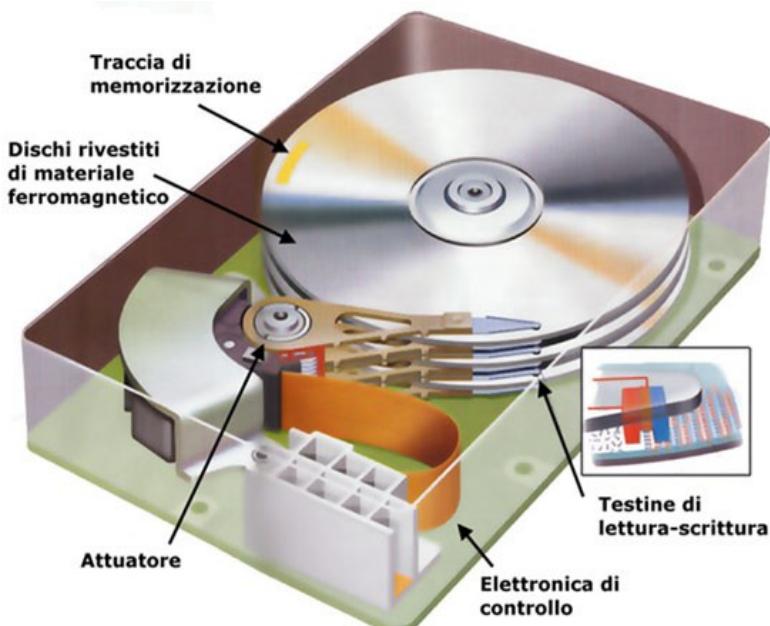
Un po' di storia

- Disco magnetico come memoria secondaria, sviluppato da IBM nel 1956
- Nome “hard disk” usato a partire dagli anni ‘80 in contrapposizione a Floppy Disk

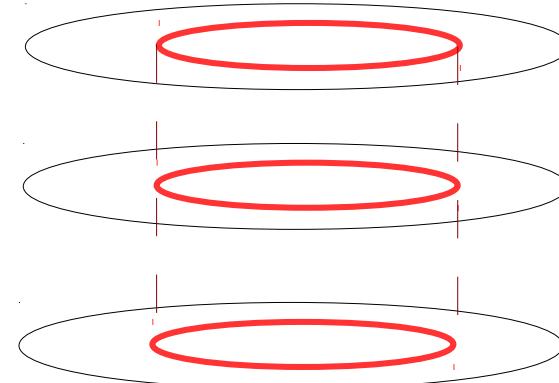


Struttura di un disco

- Un disco è costituito da un insieme di piatti le cui due superfici sono ricoperte di materiale magnetico
- Ogni disco ha una coppia di testine di lettura/scrittura che sono sospese sul relativo piatto (non lo toccano ma l distanza è minima, dell'ordine dei micron) e operano ciascuna su una superficie. Se una testina plana sul disco lo ara è occorrerà sostituirlo.



Ogni superficie è suddivisa in un insieme di tracce circolari, divise in porzioni dette settori. L'insieme di tracce corrispondenti appartenenti ai diversi piatti è detto cilindro



Principio CHS

- CHS = Cylinder, Head, Sector
- I dati organizzati secondo questo principio hanno come indirizzo fisico un'ennupla contenente i seguenti id:
 - Id piatto
 - Id traccia
 - Id cilindro
 - Id settore
 - Id blocco
 - Id testina R/W
 - Id cluster

Funzionamento

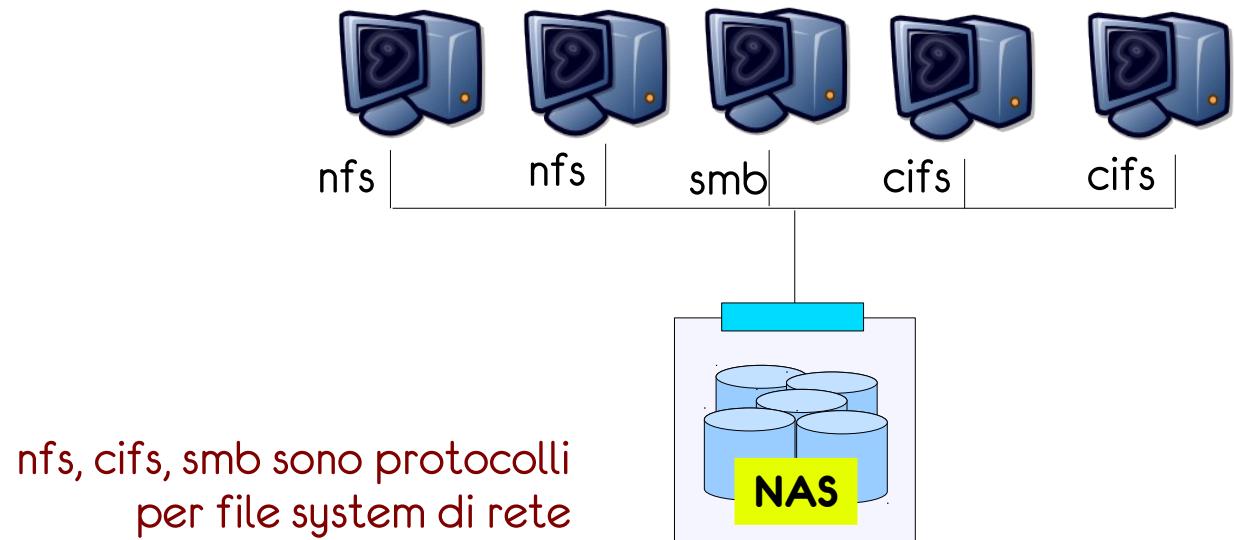
- Quando è in funzione un disco ruota a una velocità oggi compresa fra i 4000 e i 15000 giri al secondo
- Tutte le operazioni che abbiamo menzionato (lettura, scrittura, ricerca di un blocco) richiedono la **ricerca** di qualche settore su una delle superfici di qualche piatto del disco
- Il tempo necessario, detto **tempo di posizionamento**, comprende:
 - **tempo di seek**: richiesto per posizionare il braccio sul cilindro giusto
 - **latenza di rotazione**: richiesto affinché il settore giusto si posizioni, a seguito della rotazione, sotto alla testina del braccio
- a ciò si aggiunge il **tempo di trasferimento** da RAM a disco o viceversa

Connessione al calcolatore

- Un disco è connesso direttamente a un calcolatore attraverso un bus di I/O
- Esistono diverse tecnologie di connessione, es:
 - EIDE
 - ATA/SATA (già menzionate)
 - SCSI (già menzionata)
 - USB
- Il trasferimento dei dati è gestito da una coppia di controllori:
 - un adattatore, posto sul lato computer
 - un controllore del disco, incorporato nel disco stesso
- un comando di I/O viene inserito in RAM in una pagina mappata sull'adattatore, viene quindi trasferito all'adattatore, che lo passa al controller del disco, che esegue il comando utilizzando il buffer cache

Connessione via rete

- i dischi possono essere fisicamente separati dagli elaboratori e connessi alle postazioni di lavoro tramite una rete
- si parla di **NAS**, network-attached storage, accessibili tramite chiamate di procedura remota (RPC)
- la memoria secondaria in questi casi è spesso organizzata come una batteria di dischi **RAID** (es. in laboratorio)



Struttura e formattazione

- Dal punto di vista logico un disco è un array di blocchi di pari dimensioni, tipicamente 512 byte
- L'array è mappato sul disco a partire dal primo settore della prima traccia del primo cilindro (quello più esterno). Completato un cilindro ci si sposta internamente, al successivo
- Appena prodotto però un disco non ha questa struttura, che viene imposta con un'operazione detta **formattazione fisica**, solitamente operata dal costruttore
- L'acquirente/utilizzatore può operare un altro tipo di formattazione detta **formattazione logica**, che consiste nella creazione di un file system

Formattazione fisica

- la formattazione fisica crea una struttura dati speciale per ogni settore, contenente un'intestazione, un'area dati e una coda.
- intestazione e coda sono usati dal controller del disco, contengono fra le altre cose:
 - numero del settore
 - un codice per la correzione degli errori: tramite questo codice il controller può verificare se il settore è integro e, se solo pochi bit risultano alterati, li può identificare e correggere
- la dimensione dei settori è tipicamente di 512 byte

Formattazione logica

- Quando un utente installa un SO effettua un altro tipo di formattazione, legato al partizionamento dei dischi e alla scelta dei file system da installare
- Si parla di **formattazione logica**
- La formattazione logica crea sul disco stesso le strutture dati per la gestione di quel tipo di file system (es. tabella FAT) impostandone i valori iniziali
- Se si hanno più partizioni si installeranno più file system, con la conseguente scrittura delle relative strutture dati, file system per file system

Scheduling del disco

- Il SO è il gestore di tutte le risorse fisiche dell'elaboratore, dischi compresi
- Gestire bene un disco significa (1) minimizzare i tempi di posizionamento (2) massimizzare l'ampiezza di banda:
 - **tempo di seek** (o di ricerca)
 - **tempo di latenza** (latenza di rotazione)
 - **ampiezza di banda**: numero di byte trasferiti fratto il tempo intercorso fra la prima richiesta e il termine dell'ultimo trasferimento
- tutte questi parametri sono migliorabili adottando opportune strategie

Scheduling del disco

- **richiesta**: è una system call effettuata dal SO
- le **richieste** richiedono tempo per essere evase
- di solito ogni disco ha una **coda di richieste pendenti** piuttosto lunga
- lo scheduling del disco è un'implementazione di un criterio di selezione tramite il quale si decide quale richiesta pendente verrà eseguita come successiva
- la scelta è fatta tenendo conto dei dati che costituiscono la richiesta stessa:
 - operazione di lettura o di scrittura
 - **indirizzo su disco**
 - indirizzo in RAM
 - quantità di byte da trasferire

Politiche di scheduling

- FCFS: first come first served
- SSTF: shortest seek time first
- SCAN: algoritmo per scansione o dell'ascensore
- C-SCAN: scansione circolare
- LOOK/C-LOOK

First come First Served

- le richieste sono processate secondo l'ordine di arrivo
- è una strategia equa ma non particolarmente veloce
- supponiamo di avere questa sequenza di richieste, i vari numeri rappresentano i cilindri contenenti le tracce di interesse:

98, 183, 37, 122, 14, 124, 65, 67

- supponiamo che inizialmente la testina sia posizionata sul cilindro **53**, complessivamente verrà percorsa una distanza, misurata in cilindri attraversati, pari a $(98-53)+(183-98)+(183-37)+(122-37)+(122-14)+(124-14)+(124-65)+(67-65) = \textcolor{red}{640}$

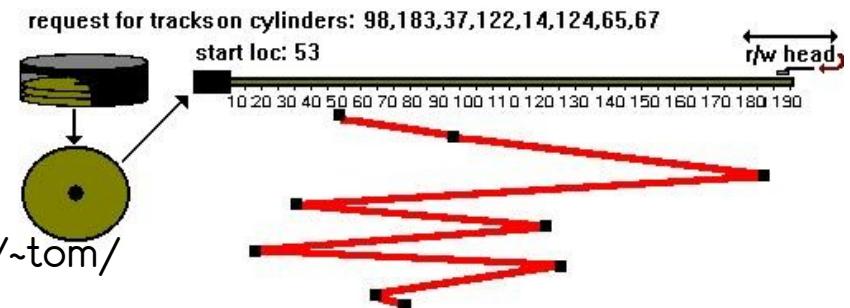


immagine tratta da <http://www.cs.wayne.edu/~tom/guide/os.html>

Shortest seek time first

- per migliorare le prestazioni del disco è possibile adottare politiche per privilegiano le richieste inerenti cilindri vicini alla posizione corrente della testina
- shortest-seek time first privilegia appunto la richiesta che minimizza il tempo di seek
- nel nostro esempio la sequenza

98, 183, 37, 122, 14, 124, 65, 67

- verrebbe gestita in quest'ordine (sempre partendo da 53):
65, 67, 37, 14, 98, 122, 124, 183
- la distanza coperta risulta: $(65-53) + (67-65) + (67-38) + \dots + (183-124) = 236$

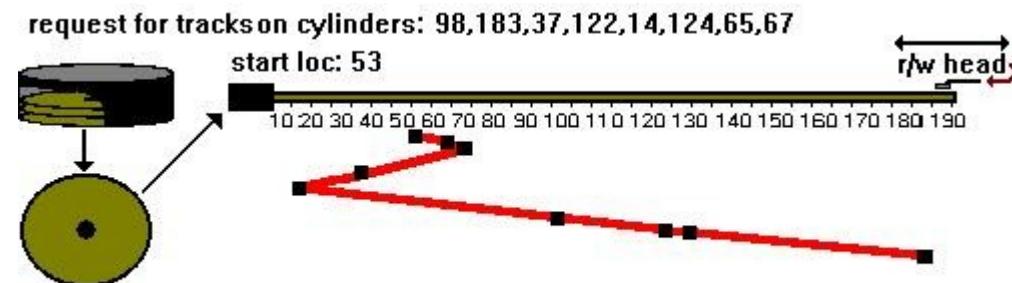


immagine tratta da <http://www.cs.wayne.edu/~tom/guide/os.html>

SSTF

- SSTF ha gli stessi difetti di tutte le tecniche a priorità: possibilità di generare starvation, cioè produrre attese indefinite per qualche richiesta
- Si osservi che il risultato ottenuto, pur essendo migliore di FCFS, non è ottimale infatti incrementando il “costo a breve termine” è possibile ridurre ulteriormente quello “a lungo termine”
- **Esempio:** se mi sposto da 53 a 37 e poi a 14 prima di invertire la direzione e attraversare 65, 67, 98, 122, 124 e 198 percorro complessivamente 208 cilindri!!
- 37 è più lontano da 53 di 65, quindi a breve termine la scelta sembra peggiore, invece sul lungo termine questa strategia risulta molto migliore

SCAN

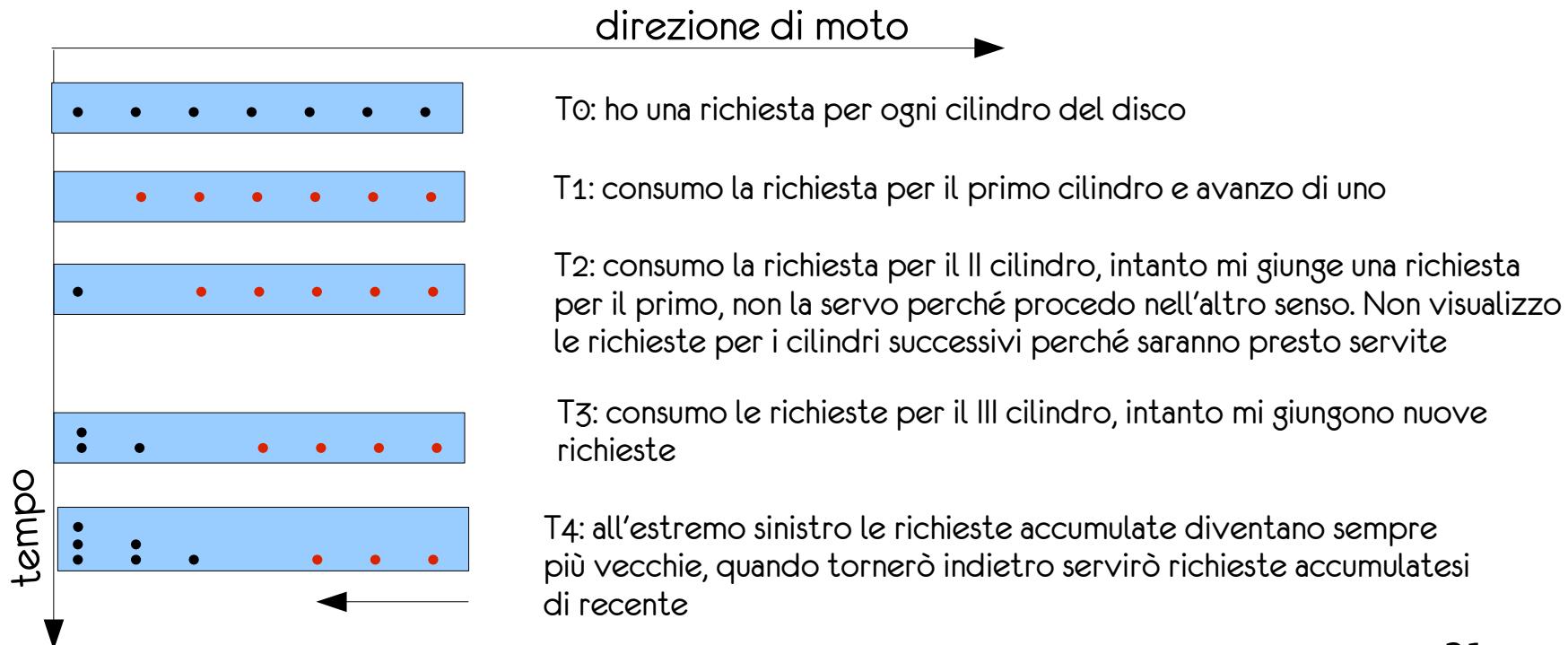
- anche detto **algoritmo dell'ascensore**:
 - il braccio parte da un **estremo del disco** e lo percorre tutto servendo le richieste man mano che attraversa i cilindri richiesti
 - arrivato all'altro **estremo** torna indietro ripetendo la procedura in senso opposto
- consideriamo il solito esempio e **supponiamo che la testina si stia muovendo verso il cilindro 0** (esterno), verranno servite nell'ordine:

37, 14, 65, 67, 98, 122, 124, 183

- per un totale di **208 cilindri** traversati
- se però la direzione fosse stata opposta avremmo ottenuto 65, 67, 98, 122, 124, 183, 37, 14 per un totale di **299 cilindri** traversati (!!)

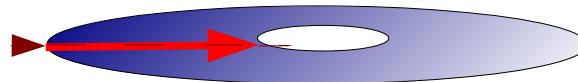
SCAN: commenti

- questo algoritmo presenta un problema
- se le richieste riguardanti i vari cilindri giungono secondo una distribuzione uniforme si avrà un accumulo di richieste non soddisfatte verso l'estremo del disco opposto a quello a cui si trova o si sta muovendo la testina
- si ha qualcosa del genere:



C-SCAN

- C-SCAN (circular scan) cerca di ovviare all'inconveniente appena descritto
- si comporta come SCAN percorrendo il disco in una direzione e servendo via via le varie richieste
- quando raggiunge l'estremo opposto la testina viene riportata direttamente alla posizione di inizio, senza servire alcuna richiesta nel viaggio di ritorno, e si ricomincia
- è un algoritmo circolare perché tratta il primo e l'ultimo cilindro come se fossero adiacenti



- Il vantaggio rispetto a prima è che i tempi di attesa delle richieste risultano più uniformi
- verranno servite nell'ordine: 65, 67, 98, 122, 124, 183, 199 (fine piatto), 0 (inizio piatto), 14, 37
- totale: $2+21+24+2+59+[16]+[199]+14+23= 145+[215] = 360$ di cui 215 senza letture quindi attraversati velocemente

Look

- LOOK e C-LOOK sono le due varianti SCAN e C-SCAN effettivamente usate
- la differenza è che il braccio procede in una direzione finché ci sono richieste per cilindri che si raggiungono spostandosi in quella direzione
- non necessariamente si raggiunge l'estremo opposto
- per esempio C-LOOK si comporta in questo modo: 65, 67, 98, 122, 124, 183, 14, 37
- da 183 non si va a fine piatto e quando si salta indietro si salta al primo cilindro richiesto (14)
- in totale avremo $131 + [169] = 300$

Scelta dell'algoritmo di scheduling

- fino ad ora abbiamo considerato solo il tempo di seek però il tempo di posizionamento su di un blocco dipende anche dalla **latenza di rotazione**, che può essere altrettanto lunga
- l'associazione fra blocchi e settori di solito non è nota al SO quindi non si possono attuare strategie particolari
- in genere se la coda delle richieste è piuttosto scarica (es. PC di casa) il sovraccarico computazionale dato dall'ordinamento della coda delle richieste non è giustificato, quindi un **FCFS** va benissimo
- se invece la coda è lunga (es. server) **SSTF** e **LOOK** sono una scelta migliore
- un impatto importante sui tempi di accesso ai blocchi dati dei file è comunque dato dalla **struttura delle directory** e dalla **tecnica di allocazione dei blocchi disco**. Il tempo di reperimento dei dati sarà tanto maggiore quanti più salti occorre fare nel disco (es. allocazione concatenata)

Commenti

- lo scheduling effettuato dal SO deve inoltre tener conto di altri aspetti riguardanti il tipo di blocchi richiesto e il tipo di operazioni richieste
- **Esempi**
 - il caricamento di una pagina di memoria virtuale (di codice) potrebbe avere priorità sull'input/output dei processi
 - una sequenza di scritture sullo stesso file dovrebbe mantenere l'ordine con cui le scritture sono state effettuate per mantenere la consistenza dei dati in caso di crash

Bootstrap: codice d'avvio

- Abbiamo visto a inizio corso che il bootstrap del SO è effettuato da un programma residente in una piccola memoria speciale ROM o EEPROM
- questa memoria ha un indirizzo noto alla CPU, che viene consultato all'avvio del computer
- il fatto che si tratti di una ROM fa sì che il **codice d'avvio non sia modificabile**, per esempio, da virus: la modifica richiede una sostituzione fisica del circuito

Bootstrap: codice d'avvio

- Abbiamo visto a inizio corso che il bootstrap del SO è effettuato da un programma residente in una piccola memoria speciale di tipo ROM
- questa memoria ha un indirizzo noto alla CPU, che viene consultato all'avvio del computer
- il fatto che si tratti di una ROM fa sì che il codice d'avvio non sia modificabile, per esempio, da virus: la modifica richiede una sostituzione fisica del circuito
- un utente può però installare diversi SO, magari più d'uno sulla stessa macchina, come fa il codice di avvio (un programma molto semplice, installato prima di qualsiasi SO) a sapere quanti SO reperire il loro codice?
- molti produttori risolvono il problema inserendo nella ROM soltanto un programma, detto **boot loader**, che carica **il vero e proprio programma di avviamento, residente su disco e modificabile**

Disco di avvio

- il programma di avviamento risiede in alcuni blocchi di una partizione del disco detti blocchi di avviamento
- il disco contenente tale partizione è detto **disco di sistema** o disco di avvio
- il codice in ROM è solo un loader del programma di avviamento vero e proprio, che è a sua volta un loader del SO
- **Esempi**
 - NTLDR: Windows NT Loader
 - Linux Loader: LILO
 - GRUB
- **NB:** alcuni SO sono così strettamente legati all'HW su cui girano (es. Macintosh pre-1995) che è quasi impossibile far caricare sullo stesso HW un SO diverso

Boot-loader primario e secondario

- il BIOS è il boot-loader primario, oltre a caricare un boot-loader secondario, funzione del BIOS è eseguire il POST (power-on self-test, verifica che i device abbiano corrente) e inizializzare tutti i device che servono prima che il SO sia stato avviato, es:
 - monitor e tastiera
 - mouse,
 - controller SCSI, USB, CDROM,
 - ...
- la ricerca del boot-loader secondario viene fatta secondo la sequenza di device di memoria contenuta in una lista definita dall'utente
- **esempio:** se la sequenza è CDROM, HD cercherà prima se è presente un CD (Live CD) e se questo contiene un SO, se non lo trova passa a cercare sull'HD
- sull'HD la posizione del loader secondario è spesso indipendente dal SO installato. Si fa riferimento a una locazione nota come **MBR, master boot record**

Hard e soft reboot

- l'avvio di un computer dipende in parte dalla modalità di spegnimento attuata in precedenza. Si parla di **hard** (o **cold**) **reboot** e **soft** (o **warm**) **reboot**
- **Hard reboot**: il computer si è spento a causa di un'interruzione di corrente (o di un errore) oppure lo spegnimento forzoso è stato attuato deliberatamente per es. per contrastare l'azione di un virus
 - in questo caso il **contenuto di tutti i buffer e le cache viene perso**, il FS **può risultare in uno stato di inconsistenza** e occorre attuare procedure speciali di ripristino
- **Soft reboot**: segue uno spegnimento avvenuto attraverso una corretta procedura di shutdown

Boot-loader secondario

- GRUB, LILO, NTLDR sono esempi di boot-loader secondari
- **NT LoaDeR**: viene installato nell'MBR ed esegue queste operazioni nell'ordine:
 - accede al file system (di tipo FAT o NTFS)
 - verifica se esiste un'immagine dello stato di esecuzione relativa all'ultimo spegnimento (immagine di **ibernazione**), se sì carica l'immagine per consentire il proseguimento dell'esecuzione dal dove si era interrotta
 - se non la trova esegue il programma **boot.ini** e segue le scelte dell'utente
 - se è stato scelto un OS NT verifica l'HW presente e avvia il programma Ntoskml.exe che avvia NT
 - altrimenti cede il controllo al file associato alla scelta fatta nel file boot.ini, un file non editabile direttamente dall'utente
- **NB**: non è propriamente multiboot perché non consente di avviare SO sviluppati da altri produttori

Difetti Fisici dei dischi

- nessun disco è completamente sano, quando escono dalla fabbrica i dischi presentano sempre un certo numero di blocchi difettosi
- altri possibili guasti possono riguardare le parti mobili (braccio, testine)
- alcuni guasti sono irreparabili: una testina che plana su un piatto causa un danno non recuperabile. L'unico modo per recuperare i dati è attraverso una copia di backup (es. delle home degli utenti)
- per i blocchi difettosi sono invece state studiate tecniche che consentono di usare un disco nonostante la loro presenza
- i blocchi difettosi (1) vanno individuati e (2) occorre “saltarli” quando si allocano blocchi ai file

Gestione manuale

- es. dischi con **interfaccia IDE**
 - i **blocchi difettosi fin dall'origine** vengono individuati dalla procedura di **formattazione logica** e opportunamente marcati come difettosi all'interno delle strutture usate per tener traccia dei blocchi liberi
 - **es.** format di MS-DOS segna i blocchi guasti nella FAT
 - i **blocchi che si guastano nel tempo** causano una perdita dei dati conservati in essi
 - esistono procedure di verifica che possono essere attuate dall'utente senza formattare il disco intero
- questa soluzione va bene per computer di uso “domestico”

Gestione automatica

- sono strategie più raffinate, attuate ad es. dai **controllori SCSI**
- in questo caso è il controller stesso a mantenere una lista dei blocchi difettosi, inizializzata con la **formattazione fisica** del dispositivo
- questa lista viene aggiornata ogni volta che il SO richiede l'accesso a un blocco che risulta difettoso, sfruttando il codice per la correzione degli errori memorizzato con i settori durante la formattazione fisica
- è possibile attuare una **strategia di recupero** ...

Gestione automatica

- sono strategie più raffinate, attuate ad es. dai **controllori SCSI**
- in questo caso è il controller stesso a mantenere una lista dei blocchi difettosi, inizializzata con la **formattazione fisica** del dispositivo
- questa lista viene aggiornata ogni volta che il SO richiede l'accesso a un blocco che risulta difettoso, sfruttando il codice per la correzione degli errori memorizzato con i settori durante la formattazione fisica
- è possibile attuare una **strategia di recupero** ...
- il controller tiene da parte un piccolo insieme di **settori di riserva**, che non vengono trattati come settori liberi e non vengono allocati assecondando le richieste dei processi
- per **limitare i tempi di seek**, vengono tenuti liberi alcuni settori di ciascun cilindro a cui si aggiungono un cilindro intero (o più)

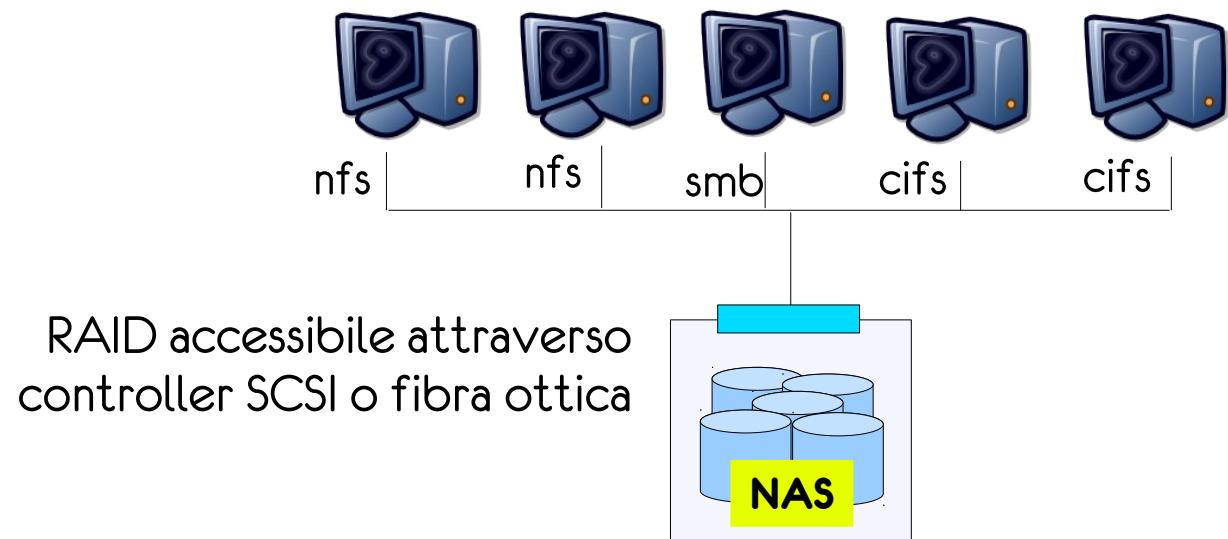


Gestione automatica

- i settori che risultano difettosi a run-time vengono **rimappati su settori accantonati**
- si cerca di utilizzare sempre **settori di riserva appartenenti allo stesso cilindro** per non stravolgere eventuali ordinamenti della coda delle richieste d'accesso
- problema: un settore difettoso conteneva probabilmente dei dati che sono andati persi, **a che serve rimapparlo su un settore vuoto?**
- *... lo scopriremo fra poche slide ...*

RAID

- RAID significa Redundant Array of Independent Disks
- si tratta di una tecnologia che consente di organizzare una batteria di dischi usandola come un'unica memoria
- adatta soprattutto a realizzare file system accessibili via rete (NAS, network attached storage)



Vantaggi

- **Costo:**

- può essere economicamente vantaggioso comperare più dischi di capacità ridotta rispetto a un disco di grande capacità;
- i dischi possono essere aggiunti un po' per volta

- **Affidabilità:**

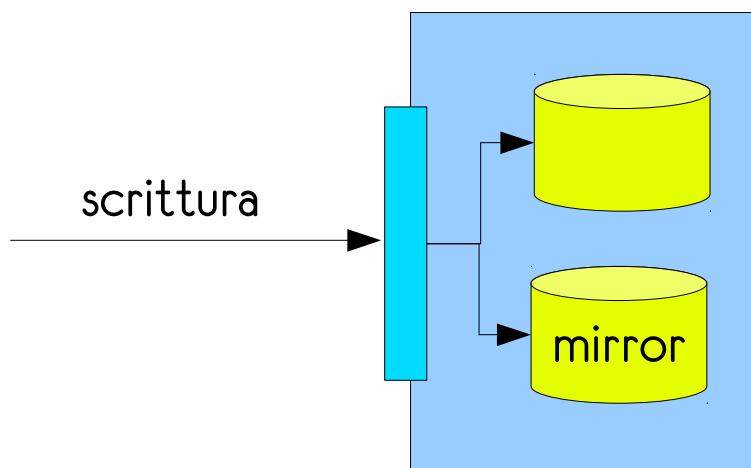
- se uno dei dischi del RAID si rompe, gli altri continuano a funzionare senza problemi. Se al contrario un disco di grande capacità si rompe nessuna parte dei suoi contenuti è più accessibile

- **Ridondanza:**

- con più dischi diventa possibile attuare tecniche di memorizzazione dei file che consentono il **ripristino automatico** dell'informazione in caso di rottura di un disco

Ridondanza

- **idea di base:** mantenere informazioni extra, non necessarie finché il sistema funziona correttamente, che però si possono sfruttare per i ripristini
- **esempio, mirroring (o shadowing)**
 - si trattano due dischi fisici come un solo disco logico
 - ogni disco fisico è l'esatta copia dell'altro
 - ogni scrittura viene effettuata su entrambi i dischi
 - per perdere i dati occorre che si rompano entrambi i dischi



Perdita dei dati

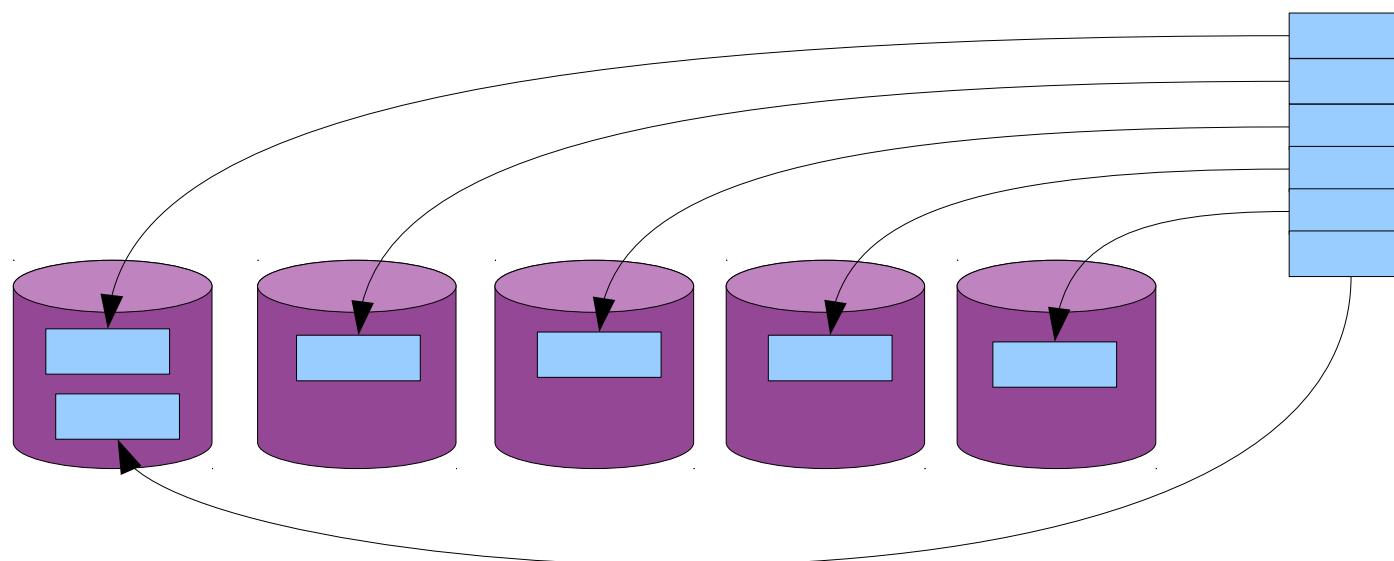
- nel caso ideale in cui le roture dei dischi siano indipendententi si può calcolare che potrebbero trascorrere diverse migliaia di anni prima che i dati memorizzati da un sistema con mirroring vengano persi
- di norma però due dischi usciti di fabbrica insieme, usato uno come mirror dell'altro, tendono a decadere in modo omogeneo quindi quando uno dei due si rompe la probabilità che anche l'altro si rompa dopo poco tempo è piuttosto alta
- se un disco si rompe non si può pensare che i dati memorizzati solo sul mirror siano al sicuro a lungo
- però difficilmente la rottura sarà simultanea e ciò consente il ripristino dei dati
- il mirroring è una tecnica costosa perché gli utenti hanno a disposizione metà della memoria effettivamente presente

Prestazioni

- la duplicazione dei dati su due dischi fisicamente indipendenti, consente di **servire in parallelo due richieste di lettura**, la frequenza delle letture raddoppia
- è anche possibile **incrementare la frequenza di scrittura** sezionando i dati
- si possono sezionare i dati a diversi livelli
 - **sezionamento dei blocchi**
 - **sezionando i byte in singoli bit**, occorre un numero di dischi multiplo di 8

Prestazioni

- la duplicazione dei dati su due dischi fisicamente indipendenti, consente di **servire in parallelo due richieste di lettura**, la frequenza delle letture raddoppia
- è anche possibile **incrementare la frequenza di scrittura** sezionando i dati
- si possono sezionare i dati a diversi livelli
 - **sezionamento dei blocchi**: i blocchi di un file vengono sparsi sui vari dischi, il blocco j di un file viene memorizzato sul disco di numero $(j \% N)+1$ dove N è il numero dei dischi del RAID



Prestazioni

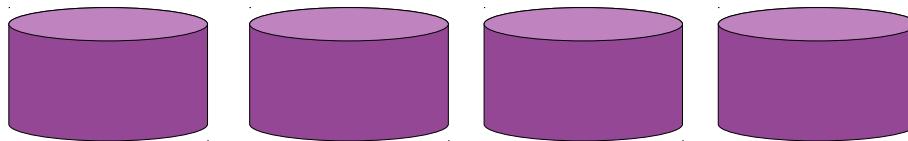
- la duplicazione dei dati su due dischi fisicamente indipendenti, consente di **servire in parallelo due richieste di lettura**, la frequenza delle letture raddoppia
- è anche possibile **incrementare la frequenza di scrittura** sezionando i dati
- si possono sezionare i dati a diversi livelli
 - **sezionamento dei blocchi**: i blocchi di un file vengono sparsi sui vari dischi, il blocco j di un file viene memorizzato sul disco di numero $(j \% N)+1$ dove N è il numero dei dischi del RAID
 - **sezionando i singoli bit**: ogni bit di ogni singolo byte viene salvato su un disco diverso
- in generale si riescono ad ottenere ottimi tempi di risposta in sistemi con accesso a grandi quantità di dati
- questa tecnica non migliora l'affidabilità

Livelli RAID

- sono stati proposti diversi schemi di RAID che realizzano compromessi diversi fra affidabilità e costo
- **Livelli RAID:**
 - livello 0: sezionamento dei blocchi, senza ridondanza
 - livello 1: mirroring
 - livello 2: aka ECC (error correcting codes)
 - livello 3: bit di parità intercalati
 - livello 4: blocchi di parità intercalati
 - livello 5: blocchi intercalati a parità distribuita
 - livello 6: aka schema di ridondanza P+Q
 - livello 0+1: combina i primi due approcci

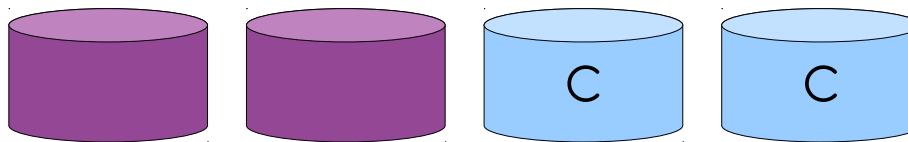
Livelli RAID

- livello 0: **sezionamento dei blocchi**, senza ridondanza



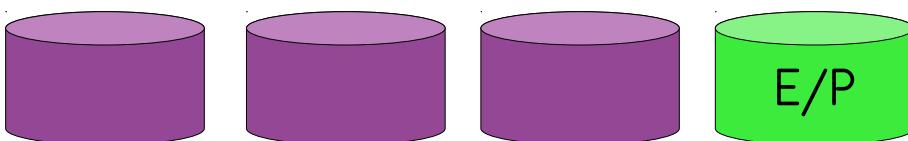
i blocchi dei file sono distribuiti sui diversi dischi

- livello 1: **mirroring**



metà dei dischi sono usati per replicare i dati

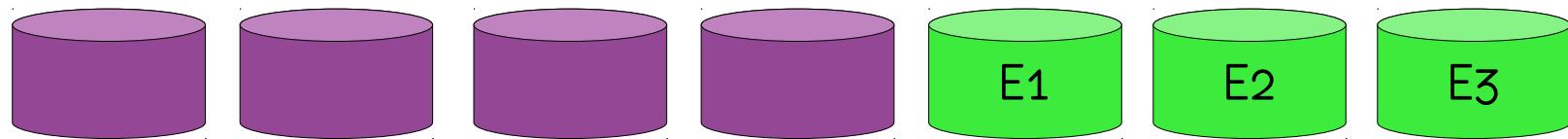
- livello 2 e 3: **ECC (error correcting codes)** e **con bit di parità intercalati**



si applica il sezionamento dei byte, si applica un controllo di correttezza basato su bit di controllo conservati su dischi a parte

Livello 2

- **ECC (error correcting codes)**

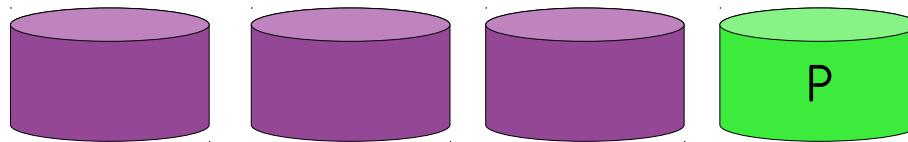


- l'informazione viene prima **codificata** e poi **memorizzata**
- la codifica arricchisce l'informazione di ulteriori dati che servono a verificarne la correttezza e/o a ricostruirla in caso una parte venga persa
- sfrutta **codici di Hamming** per la verifica della correttezza
- richiede almeno 3 dischi extra per mantenere i codici
- in pratica non è usato proprio per i suoi costi
- una sua variante molto meno costosa è la seguente



Livello 3

- **bit di parità intercalati**



- ogni byte ha associato un bit (**bit di parità**) che indica se il byte contiene un numero pari (even parity) o dispari (odd parity) di 1:

11010001 parità: 0 (# pari di 1)

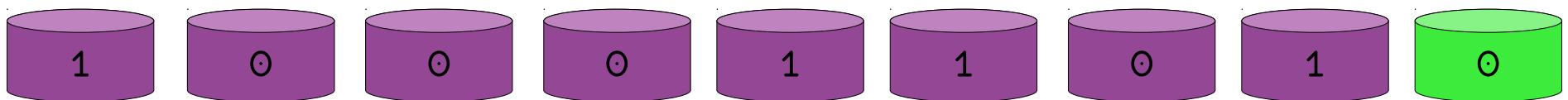
- quindi se un bit viene alterato per errore è possibile accorgersene perché il bit di parità è settato in modo discordo rispetto ai bit del byte:

01010001 parità: 0 -> errore!

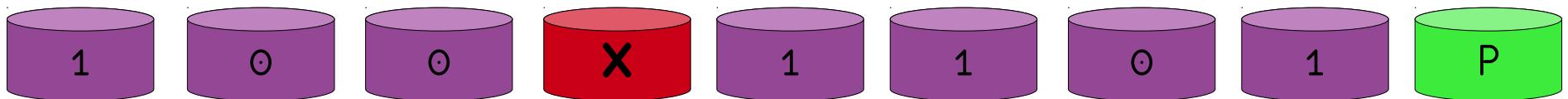
- se ad essere alterato è il bit di parità, si avrà un'analogia discrepanza
- limite delle tecniche a parità: catturano un numero dispari (1, 3, 5, ...) di fault

Livello 3

- **bit di parità intercalati**



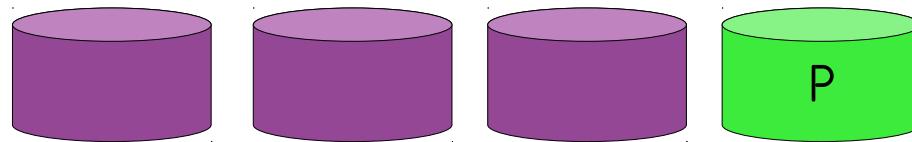
- si sezionano i bit di ogni byte distribuendoli, per esempio su **N** dischi diversi (es. N=8), mentre sull'**(N+1)-mo** disco si memorizzano i **bit di parità**
- se uno dei dischi si guasta, **è possibile ricostruire la sua immagine dai dischi restanti**: N dei dischi contengono i bit di ciascun byte mentre i bit di parità è recuperati dal disco extra (quello con la P in figura)



Se P == 1 devo avere un numero dispari di 1, altrimenti un numero pari

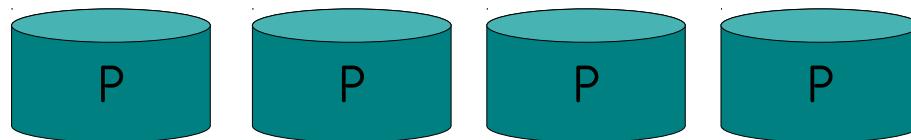
Livelli RAID 4 e 5

- livello 4: **blocchi di parità intercalati**



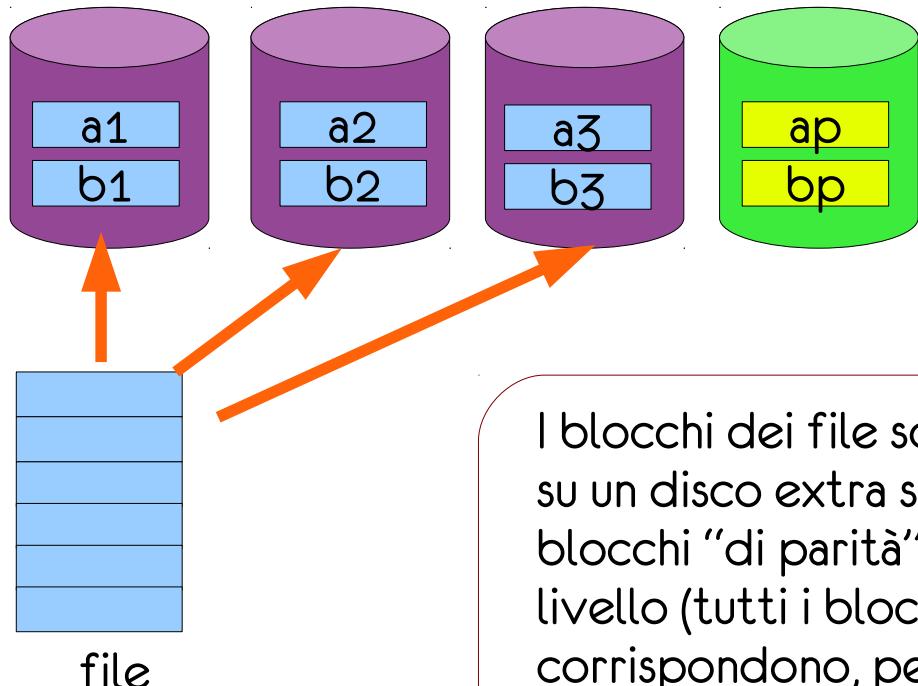
sfrutta il sezionamento dei blocchi, non dei byte, mantiene
l'informazione sulla parità su un disco a parte

- livello 5: **blocchi intercalati a parità distribuita**



blocchi dati e di parità sono distribuiti fra N+1 dischi

Livello 4



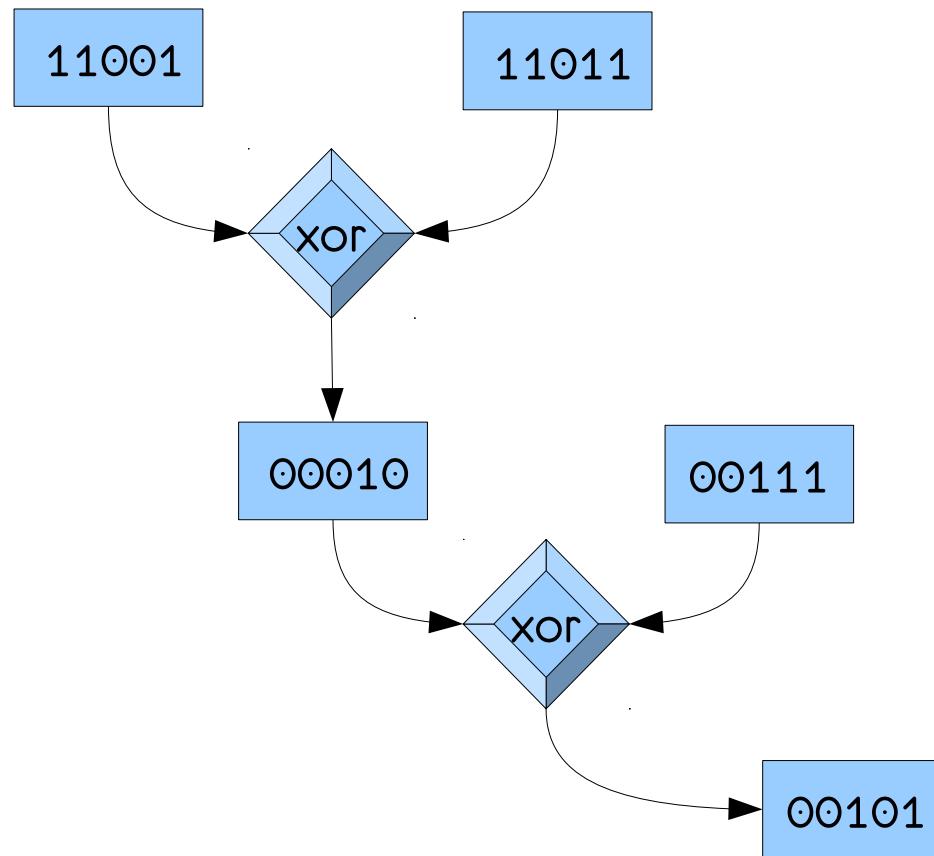
I blocchi dei file sono distribuiti su N dischi
su un disco extra sono contenuti ulteriori
blocchi "di parità" per i blocchi dello stesso
livello (tutti i blocchi dei vari dischi che si
corrispondono, per es. che hanno una medesima
posizione)

Se si brucia un disco, è possibile ricostruirlo usando gli altri N-1 dischi
e il disco di parità

Esempio: uso di xor

calcolo i bit del blocco di parità tramite XOR successivi

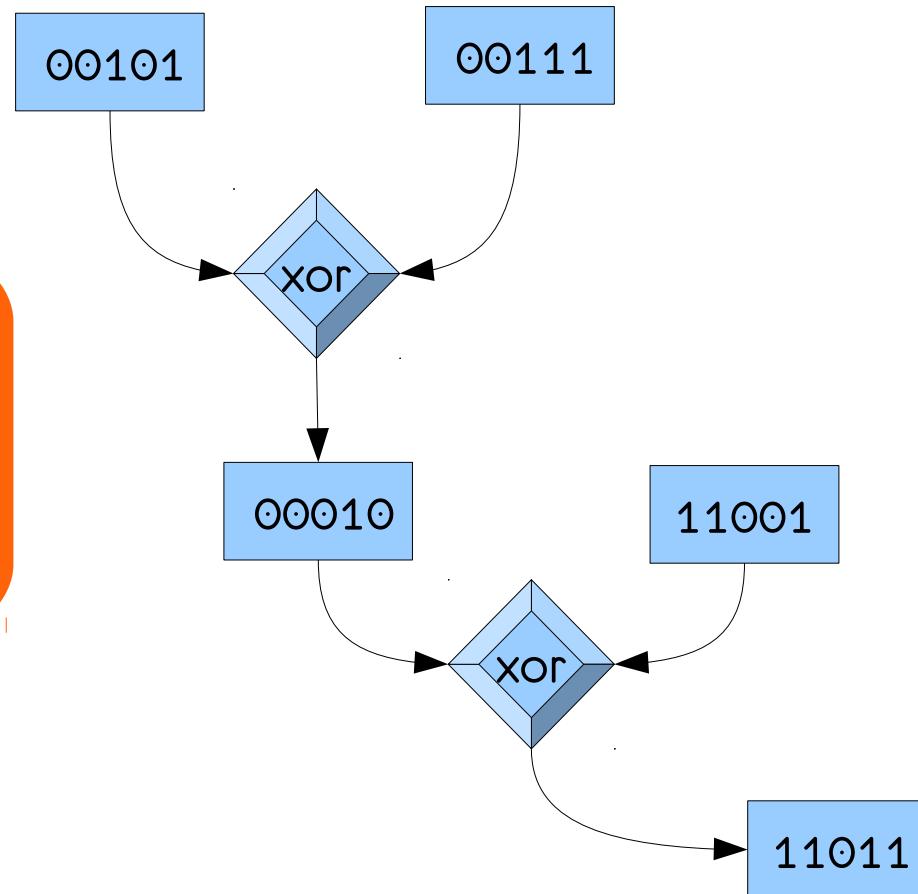
$$11001 \text{ XOR } 11011 \text{ XOR } 00111 = 00101$$



Esempio

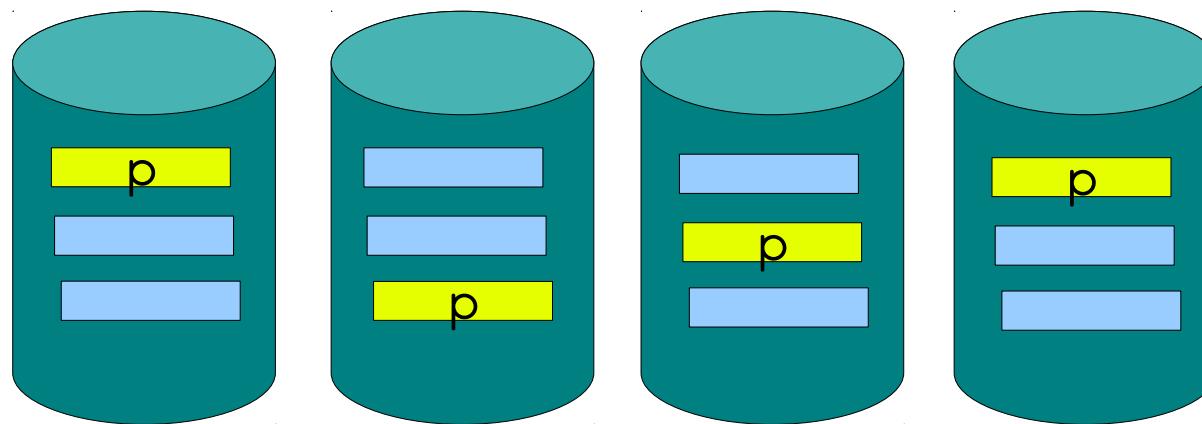
$$11001 \text{ XOR } ?? \text{ XOR } 00111 = 00101$$

Proviamo a usare i dati a nostra disposizione per ricostruire il secondo blocco



Livello 5

- simile al precedente ma i blocchi di parità sono distribuiti nei vari dischi anziché raccolti in un disco a parte, esempio:



- i rettangoli gialli rappresentano i blocchi di parità
- **Vantaggi:**
 - miglior rapporto costo/prestazione per reti orientate alle transazioni, prestazioni molto elevate, protezione dei dati molto elevata, supporte letture e scritture simultanee

GUASTO DI GIOVEDÌ

- giovedì 8 marzo di qualche anno fa si è verificato un **guasto in laboratorio**
- dei 9 dischi che costituivano il RAID **uno** è andato in fault
- non è stato possibile attuare il ripristino automatico perché:
 - degli altri 8 dischi **uno** è risultato avere alcuni settori contenenti blocchi di parità guasti;
 - non in numero sufficiente a mandare in fault il disco ma abbastanza da impedire il ripristino automatico dopo la sostituzione del disco rotto

GUASTO DI GIOVEDÌ

- **Intervento:**

- interruzione dei servizi (web, ssh, file system, ...)
- sostituzione del disco guasto
- trasformazione del disco malfunzionante in disco di spare
- formattazione di tutti i dischi
- ricorso ai backup fatti la notte precedente per recuperare i file



Livelli RAID 6 e 0+1

- livello 6: aka **schema di ridondanza P+Q**. È simile al livello 5 ma memorizza codici più complessi che consentono di risolvere guasti contemporanei su più dischi. Sfrutta dei codici di errore noti come **codici di Reed-Solomon**
- livello 0+1: combina i primi due approcci (sezionamento dei blocchi più mirroring). Buone prestazioni ma costi elevati perché richiede il raddoppio dei dischi

Realizzazione raid

- la struttura RAID può essere realizzata a **livello software** dal SO con un supporto HW minimale. La soluzione SW è tendenzialmente più lenta, per cui si preferisce realizzare in questo caso RAID di livello 0 oppure 1
- la struttura RAID si può appoggiare ad un **HW apposito**, in particolare può essere implementato dal controller di un “disk array”: un dispositivo di memoria per aziende che comprende una batteria di dischi



Scelta del livello raid

- **livello 0:** usato in sistemi ad alte prestazioni in cui la perdita di dati non è critica
- **livello 0+1:** usato per piccole basi di dati, dove occorrono sia affidabilità che prestazioni
- **livello 5:** preferito quando si devono gestire grosse quantità di dati perché richiede meno spazio dell'equivalente soluzione ottenuta adottando livelli più bassi
- **livello 6:** non ancora disponibile in molti sistemi RAID, richiede due dischi in più ma consente ricostruzione automatica in occasione di fault più complessi