memoria virtuale

capitolo 9 del libro (VII ed.), 12.6

Cristina Baroglio

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?

migliore gestione della RAM?

miglioramento della vita del programmatore?

miglioramento della vita dell'utente dei programmi?

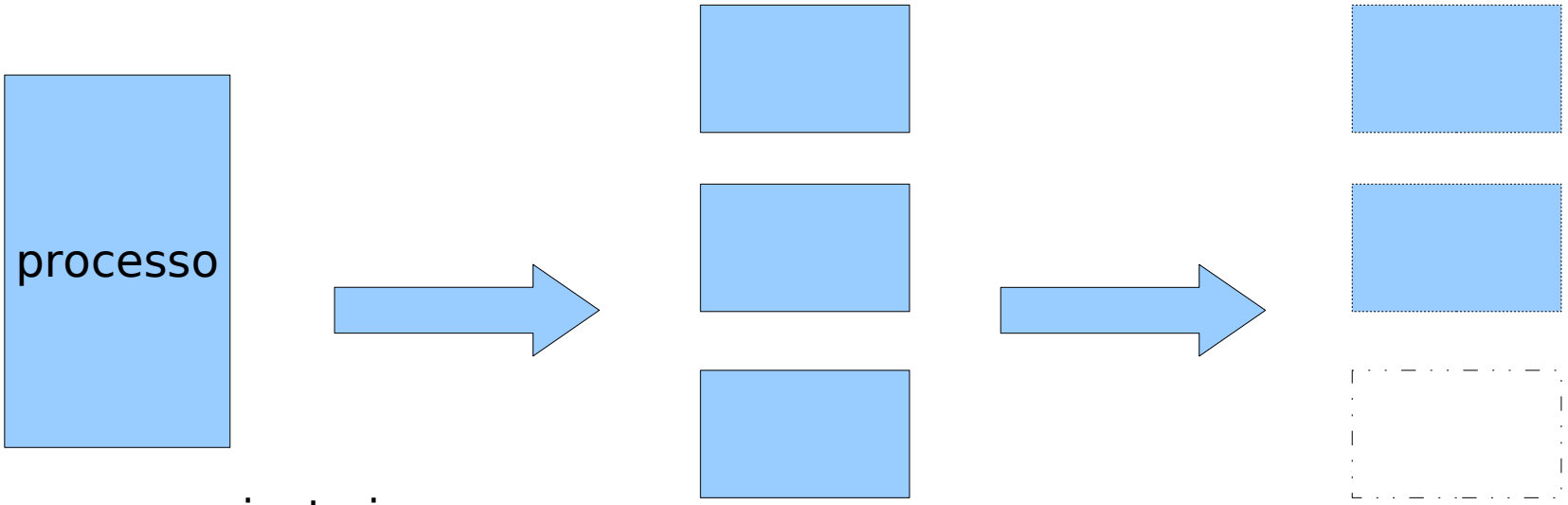
Introduzione

- 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



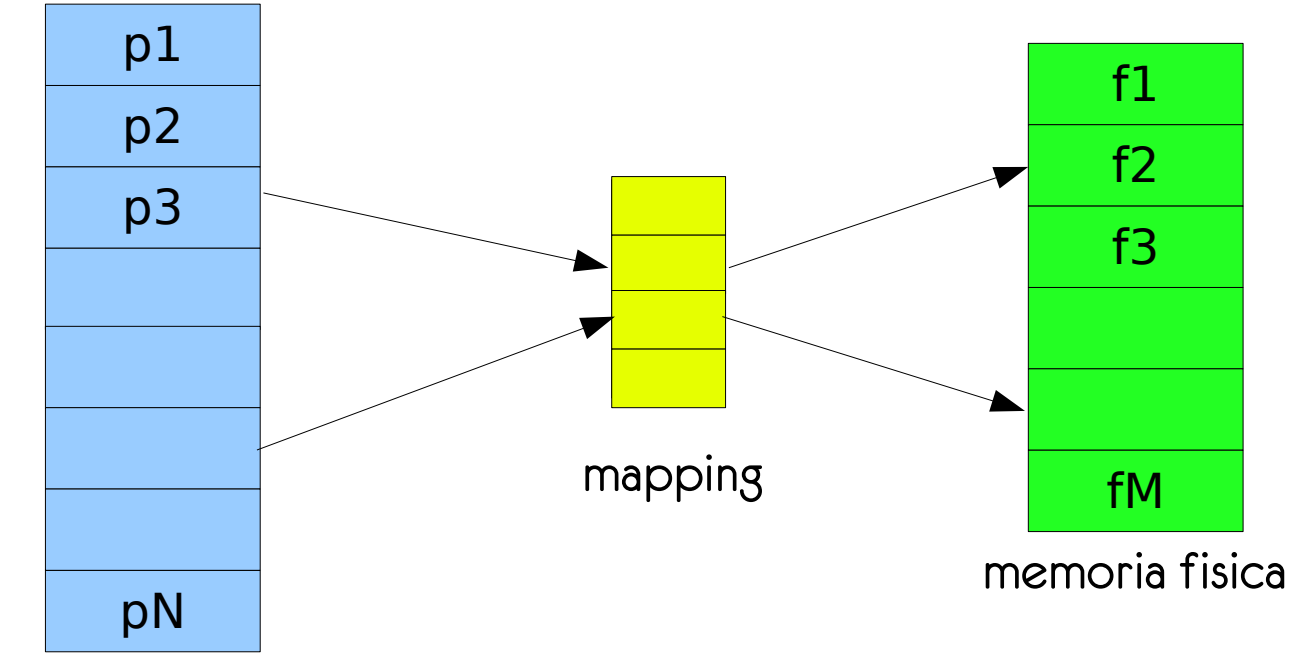
Processo caricato in RAM in modo **contiguo e totale**

Si rilascia il vincolo di contiguità:
- paginazione
- segmentazione

Si rilascia il vincolo di totalità:
- memoria virtuale

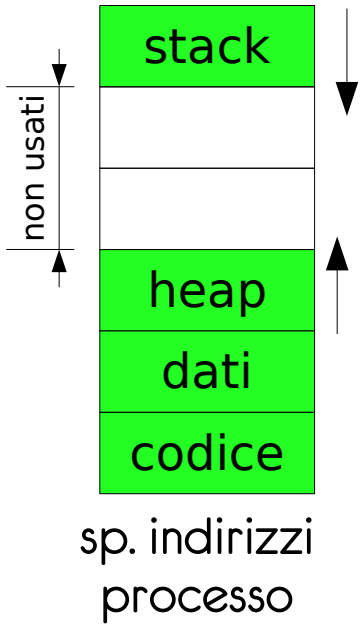
Memoria virtuale

- 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)

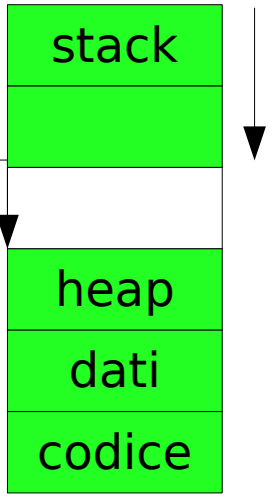
Spazio degli indirizzi di un processo



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.

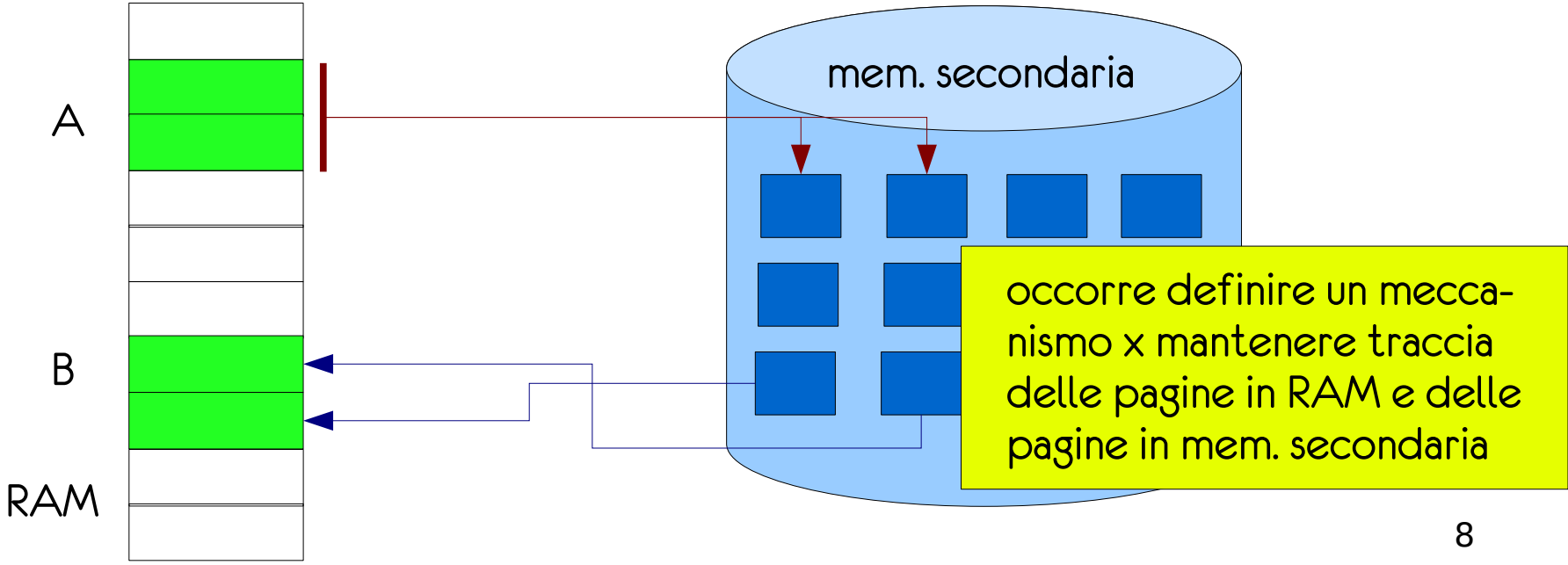
Questo non cambia lo spazio degli indirizzi virtuali del processo, semplicemente una parte di indirizzi prima non corrispondenti ad alcun indirizzo assoluto saranno ora mappati su parole di memoria effettive



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

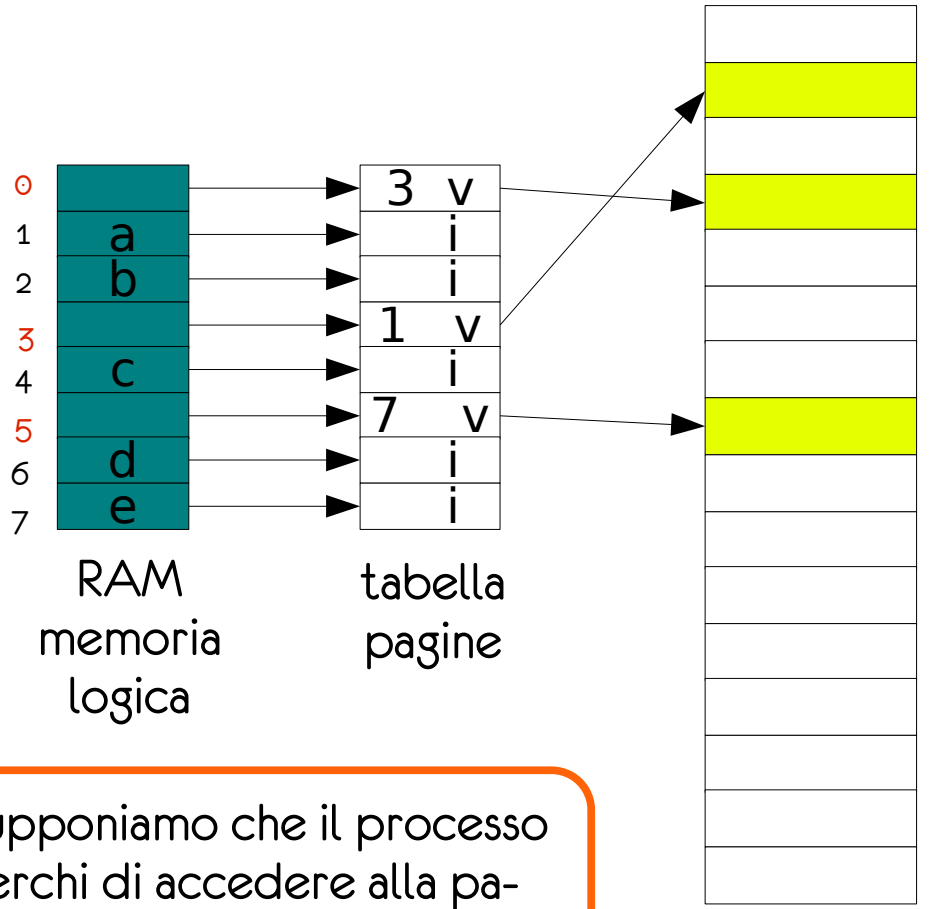
Demand paging - lazy swapping

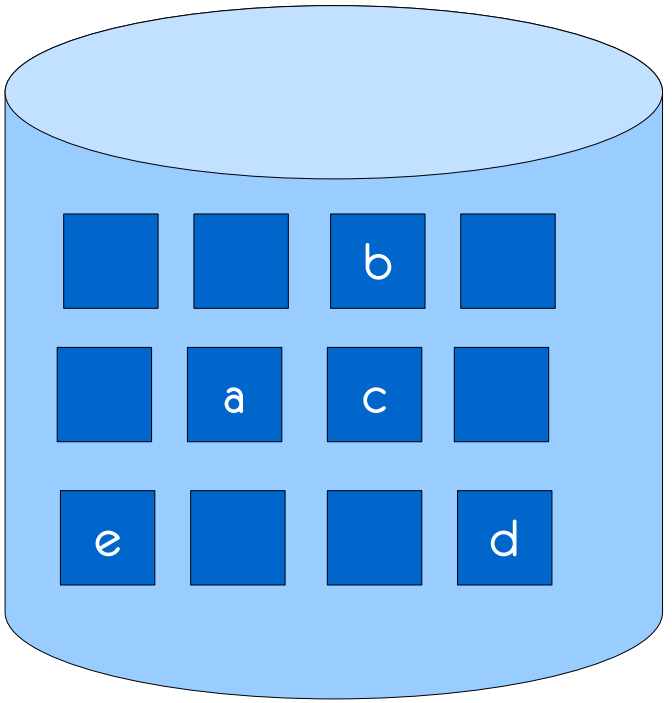
- Ricorda lo swapping ma riguarda porzioni di processo
- I processi vengono caricati in modo parziale:
  quando una pagina diventa utile (perché contiene una parte di codice da eseguire o dei dati da elaborare) la si carica avvicendandola con una pagina precedentemente in RAM
- Questo meccanismo è noto 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





le pagine invalide sono in memoria secondaria

Supponiamo che il processo cerchi di accedere alla pagina "a" ...

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 $\in$ 8msec

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 ge
- salva
- verifi na che si tratta di page fault
- contr nvalida 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'at della CPU a un altro pro
- interrupt ch della pagina
- aggiorname
- quando lo scheduling riavvia il processo sospeso, ripristino dello stato
- riese ta

Ordini di Grandezza

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

più in breve possiamo riassumere queste operazioni in

<1> servizio di interruzione per page fault
<2> lettura della pagina
<3> riavvio del processo

richiede circa 8 millisecondi

dominante

richiedono da 1 a 100 microsecondi

Un po' di numeri ...

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

ma = 200 nsec

tgpf = 8 msec

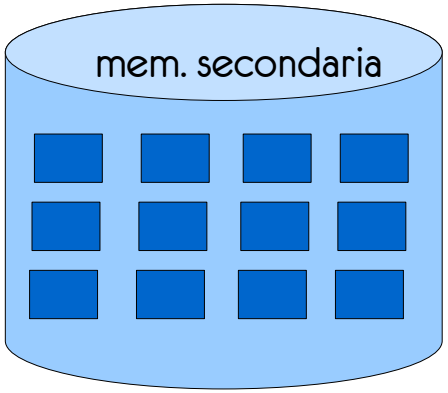
t. a. e. = *(1-p) * 200 + p * 8.000.000*
= *200 – p*200 + p * 8.000.000 =*
= *200 + p * 7.999.800*

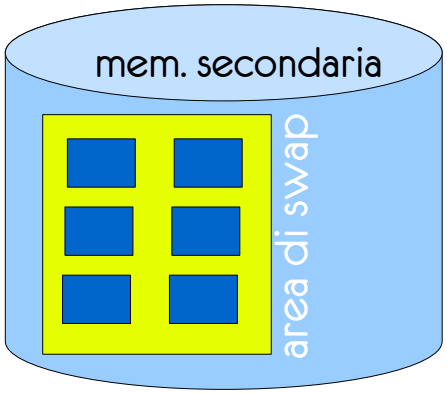
supponiamo di avere un page fault ogni 1000 accessi: p = 0.001

t. a. e. = *200 + 0.001 * 7.999.800 = 200 + 7.999,8 ≈ 8.000 nsec*
(8 microsec)

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 ½ 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 sè, 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

processo P

esegue fork

processo F: figlio

pag. codice 1

...

pag. codice N

pag. dati 1

...

pag. dati M

(1) in RAM

pag. codice 1

...

pag. codice N

pag. dati 1

...

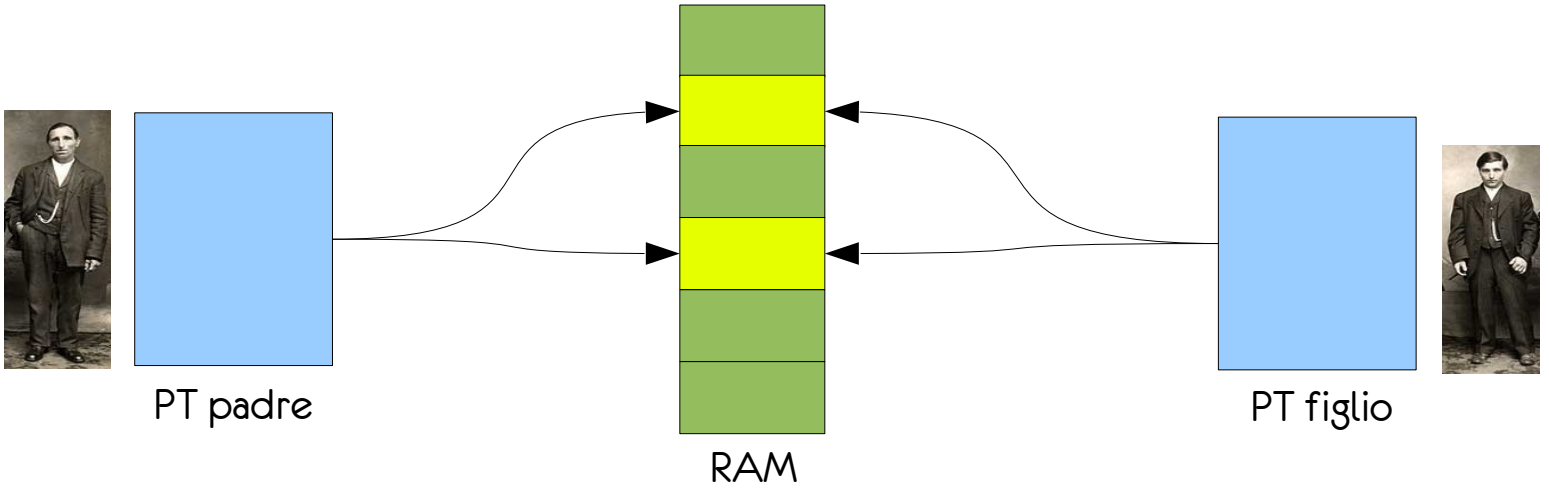
pag. dati M

esegue lo stesso codice del padre

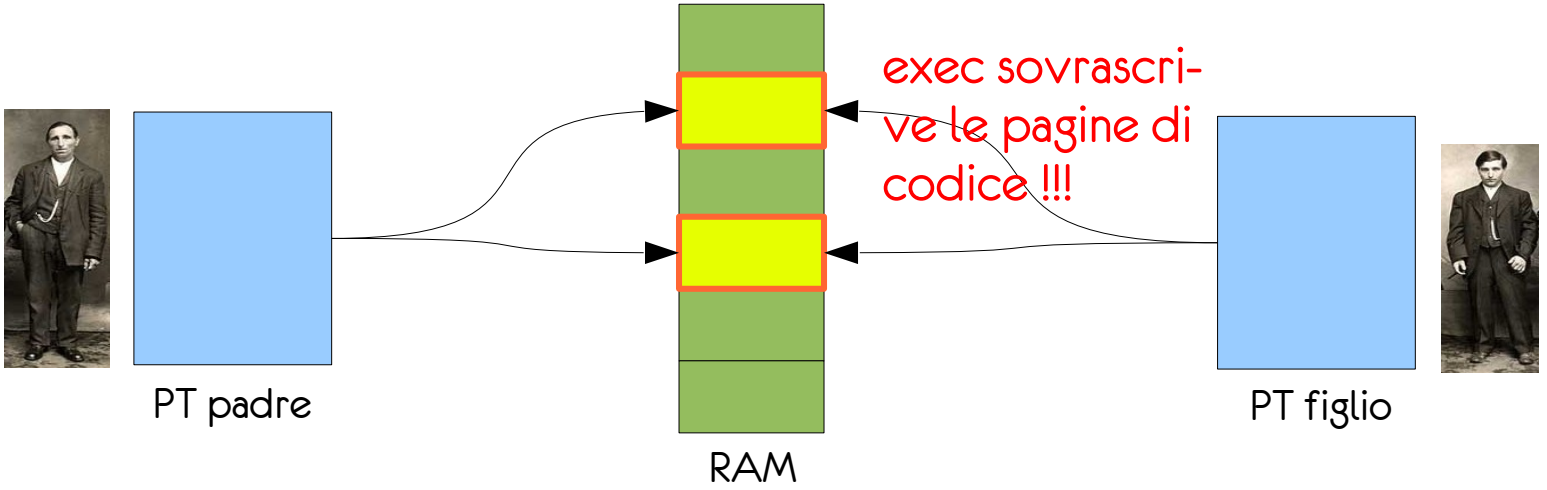
ha una copia delle vrb. del padre e dello stack

(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

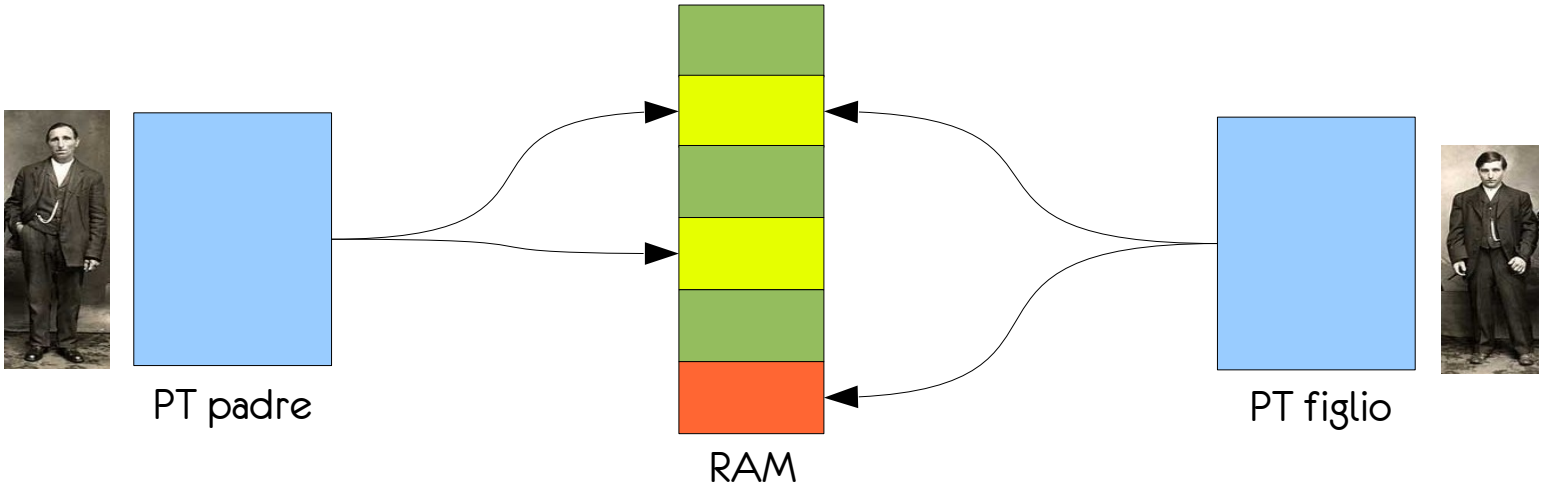
Fork, exec e paginazione



(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



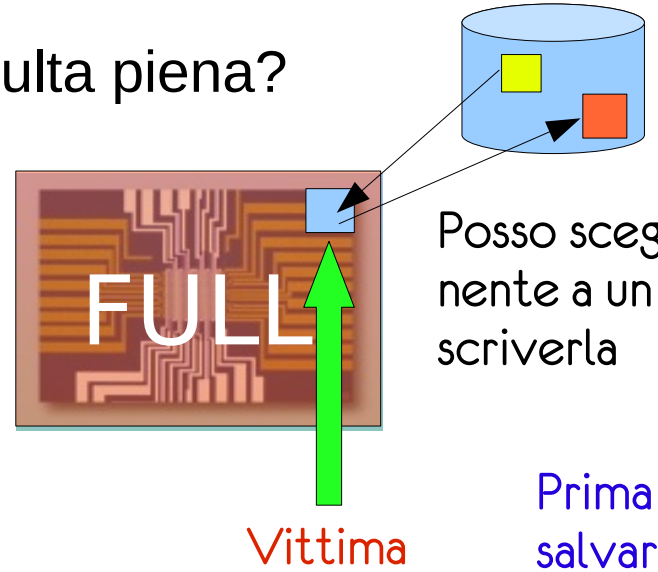
(2) per ovviare al problema della sovrascrittura delle pagine di codice del processo padre si adotta la tecnica di "copiatura su scrittura": quando uno dei due processi esegue una "exec" si allocano delle nuove pagine per il processo chiamante e si carica il nuovo codice in esse

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?



Posso scegliere una pagina appartenente a un processo in RAM e sovrascriverla

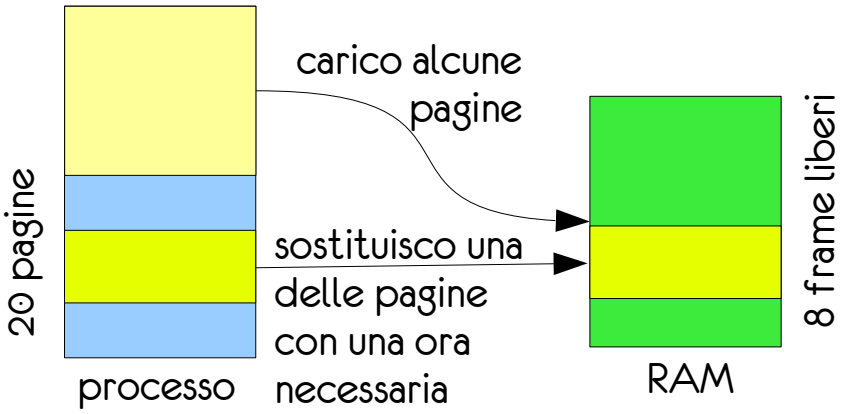
Prima di sovrascrivere occorre salvare la vittima in memoria secondaria

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

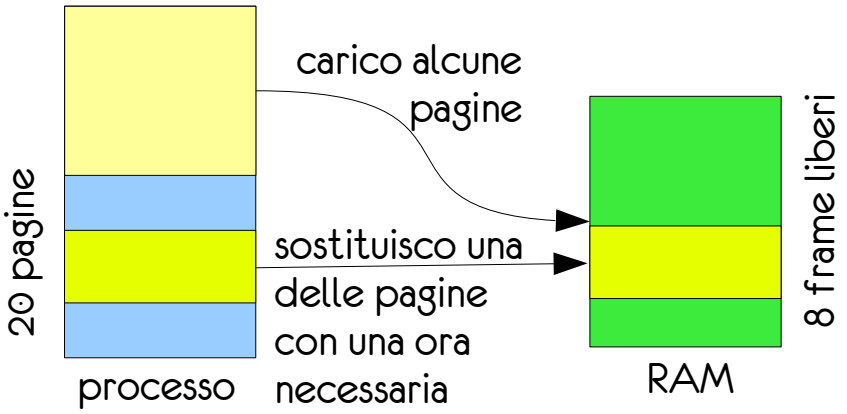
Commenti

- 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



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

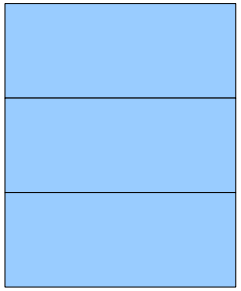
- Implementazione la paginazione su richiesta significa sviluppare due algoritmi:
  - algoritmo di allocazione dei frame:
    - spartisce M frame liberi fra N processi
  - algoritmo di sostituzione delle pagine:
    - seleziona le pagine da sostituire
    - occorre definire il criterio di selezione
    - 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

Sostituzione FIFO

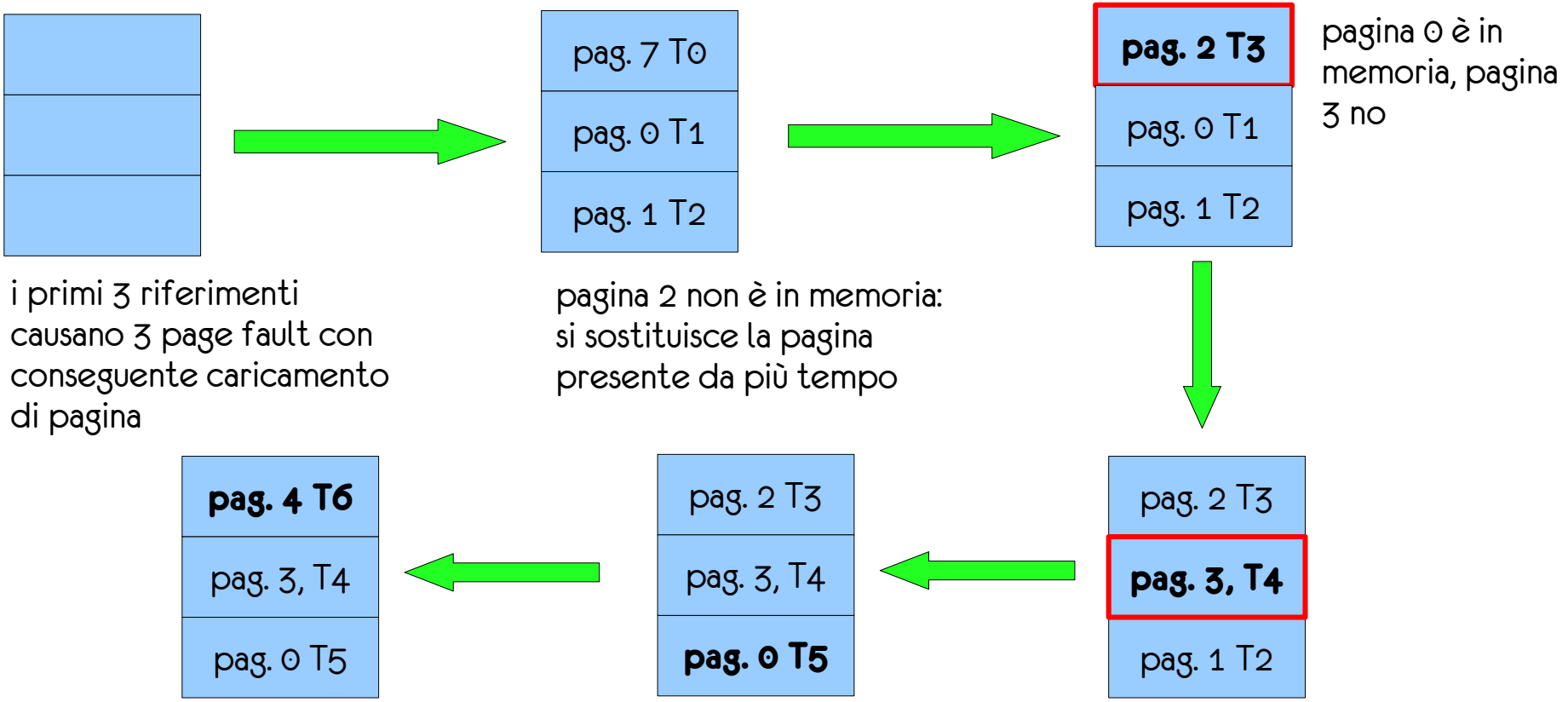
- a ogni pagina viene associato un marcatore temporale: l'istante in cui è stata caricata in memoria
- si sceglie di sostituire la pagina caricata meno di recente (quella in memoria da più tempo)
- basta organizzare le pagine in una coda FIFO
- si sostituisce sempre la pagina corrispondente al primo 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 inzialmente vuoti

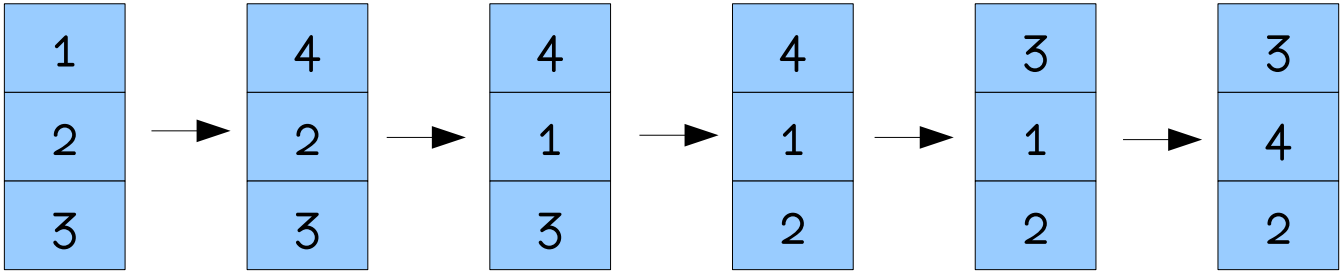
Sostituzione FIFO

- 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

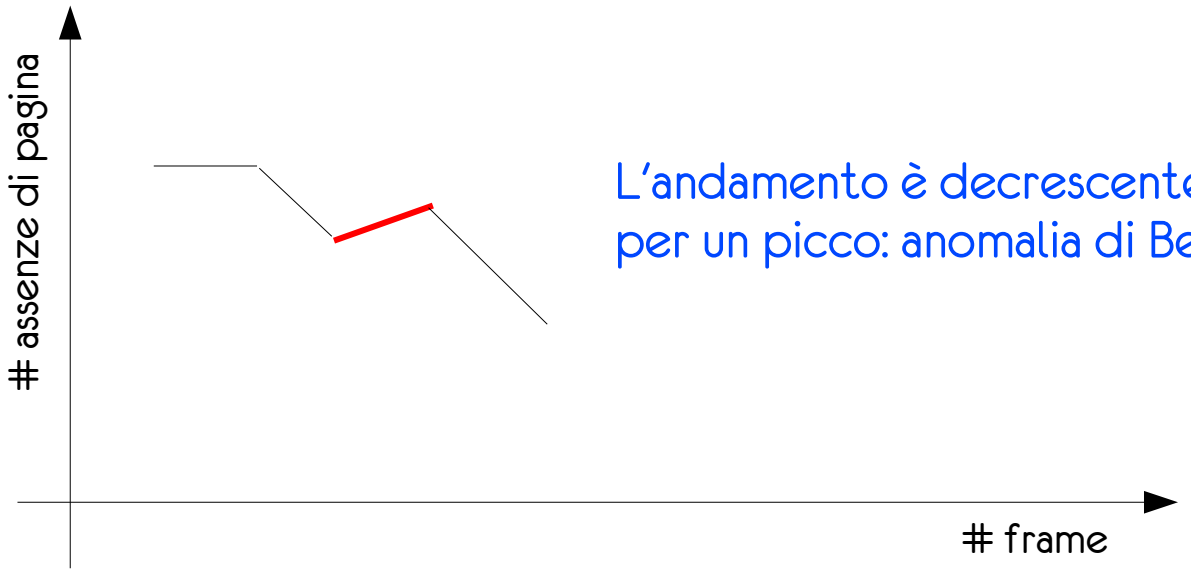
- FIFO è una tecnica semplice da realizzare ma non si comporta sempre bene
- il fatto che una pagina sia stata caricata tempo fa non significa che non sia più in uso, al contrario potrebbe essere una pagina di uso frequente: rimuoverla causerebbe sicuramente un successivo page fault
- consideriamo la 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:



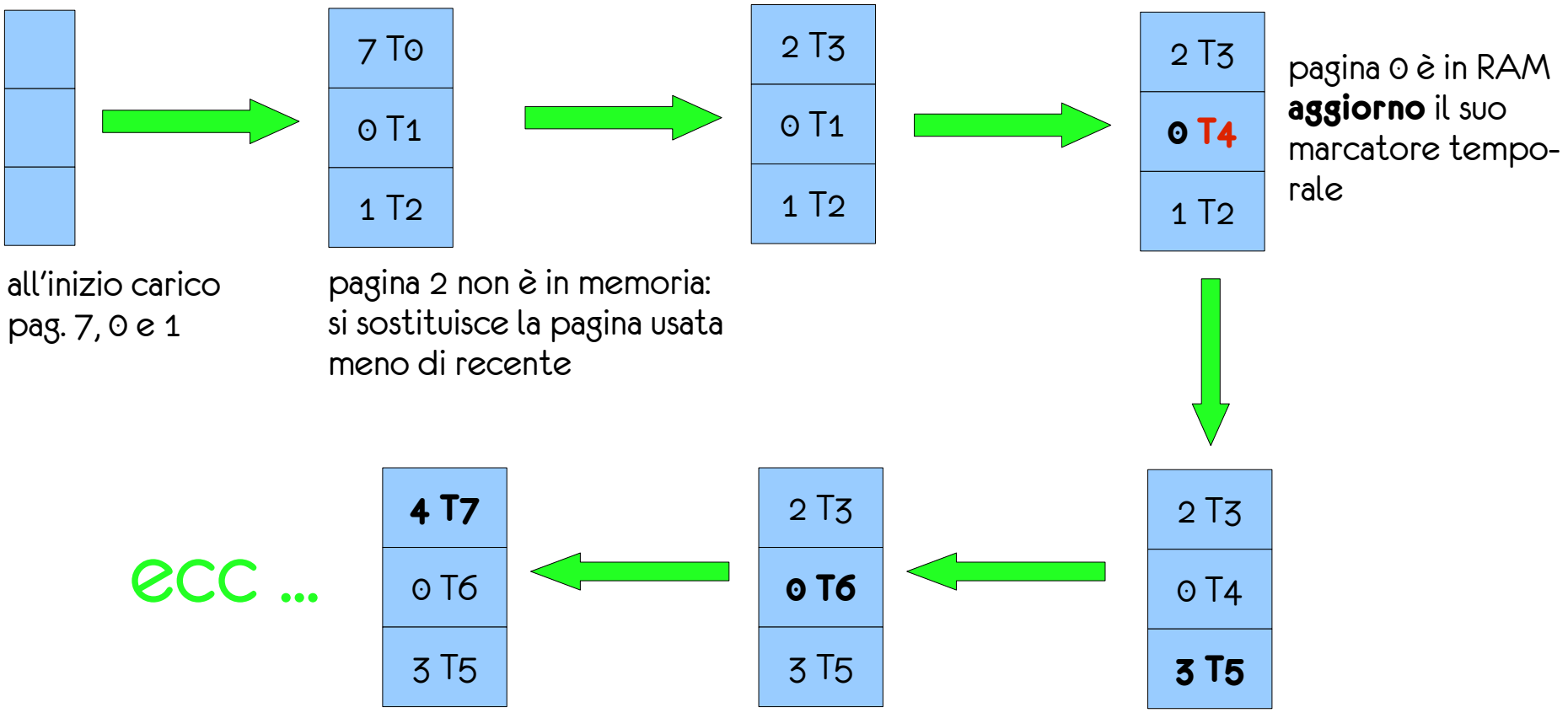
Algoritmo ottimale

- 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 marcatore 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 marcatore temporale
  - ogni volta che si accede a una pagina si aggiorna il suo marcatore usando il valore del contatore
- Problemi:
  - overflow del contatore
  - la ricerca della vittima nella tabella delle pagine è sequenziale

Commenti

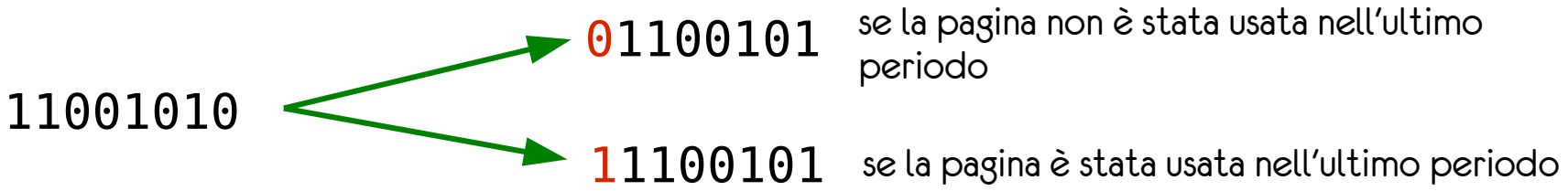
- Soluzione 2:
  - si mantiene uno stack di pagine
  - ogni volta che si fa riferimento a una pagina la si pone in cima allo stack (eventualmente togliendola da altra posizione nello stack)
  - la pagina in fondo allo stack è sempre quella usata meno di recente
- Altri commenti:
  - LRU non è soggetta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

Approssimazione a LRU

- Poche architetture consentono di applicarel'LRU
- Alcune architetture consentono di applicarne un'approsimazione 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

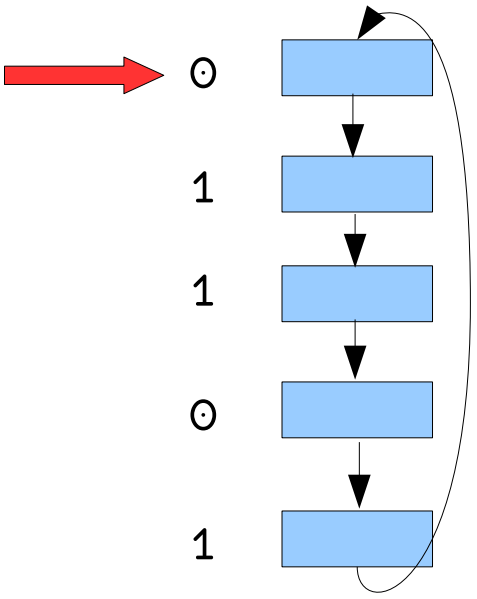
- È una tecnica di approssimazione della LRU in cui si associa a ogni elemento nella tabella delle pagine una serie di bit che realizzano dei registri a scorrimento
- A intervalli regolari un timer passa il controllo al SO, che sposta i bit nella sequenza traslandoli a destra di 1, copia il bit di riferimento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

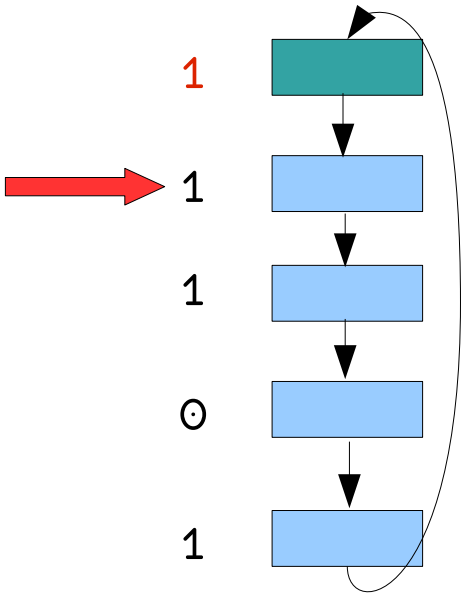
- è un algoritmo che unisce il metodo LRU all'approccio FIFO
- ogni pagina ha associato un bit di riferimento
- le pagine sono mantenute in una coda circolare FIFO



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

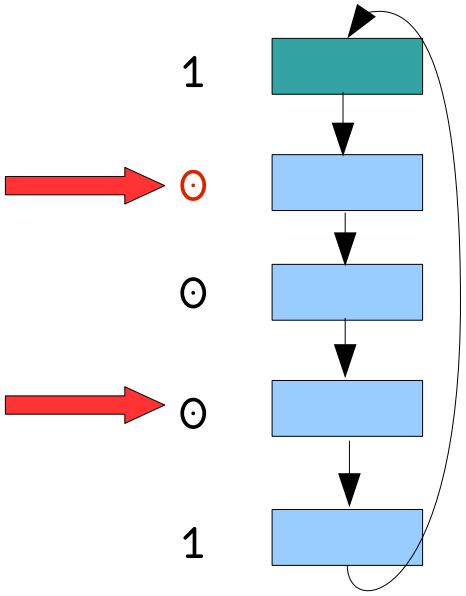
Algo. di seconda chance



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

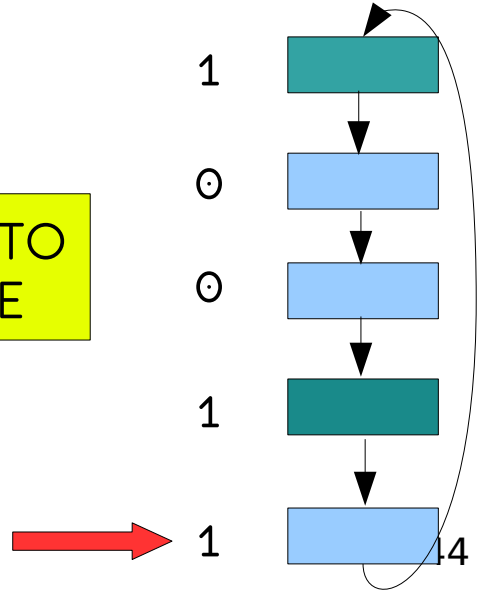
Algo. di seconda chance



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



Cristina Baroglio

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 ...

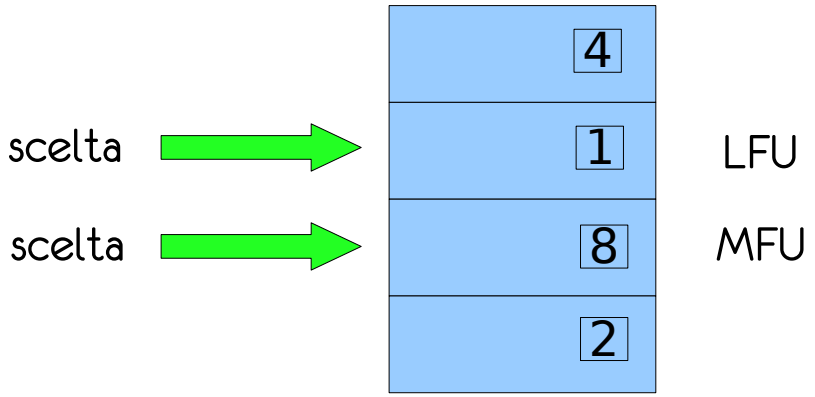
Algo. di seconda chance migliorato

- Ogni pagina ha associata una coppia di bit <r, m>:
  - 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:
  - <0, 0>: né usata di recente né modificata
  - <0, 1>: non usata di recente ma modificata
  - <1, 0>: usata di recente ma non modificata
  - <1, 1>: 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

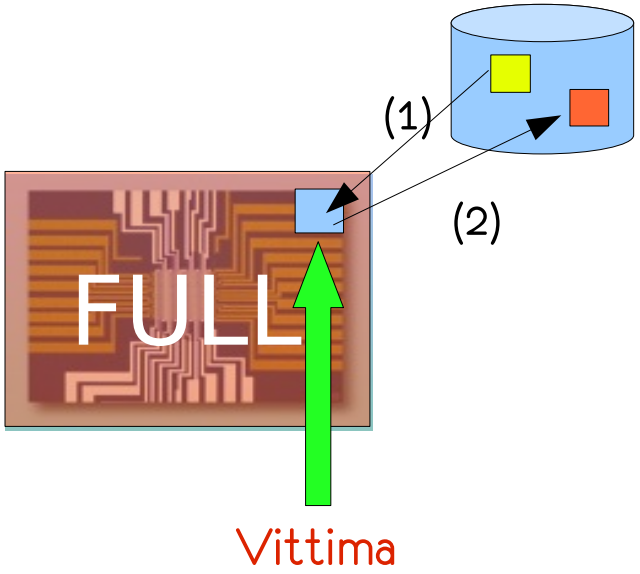


Commenti

- 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

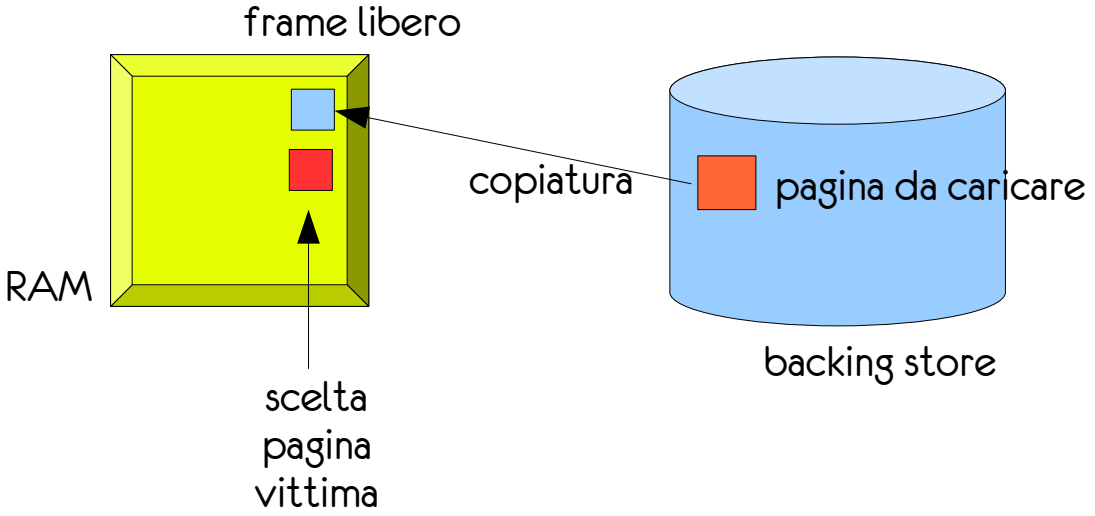
- Spesso gli algoritmi di sostituzione delle pagine sono affiancati da altre procedure finalizzate a incrementare le prestazioni del sistema
- Fra queste una tecnica che consente di iniziare la copiatura in RAM della pagina necessaria per proseguire (1) **prima** che la pagina vittima sia stata ricopiata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

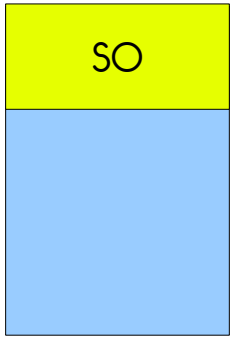
Allocazione dei frame

- 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

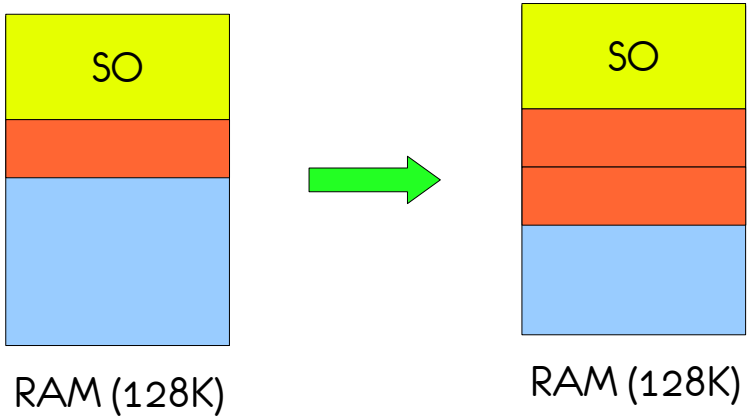
Allocazione dei frame



Una porzione deve comunque essere riservata al sistema Operativo

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

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

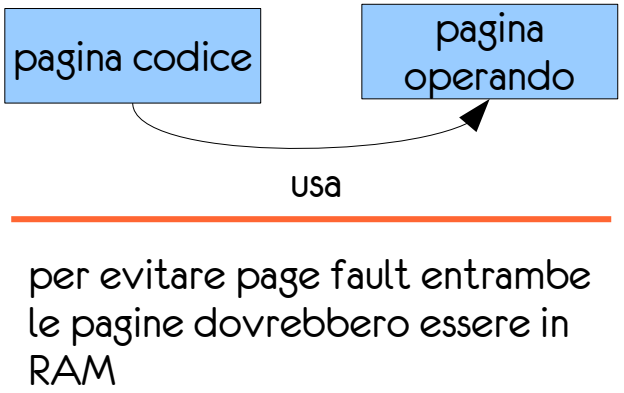
- Allocare inizialmente 0 frame per processo e poi 1 per volta quando necessario è l'unica scelta possibile? È la migliore?

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

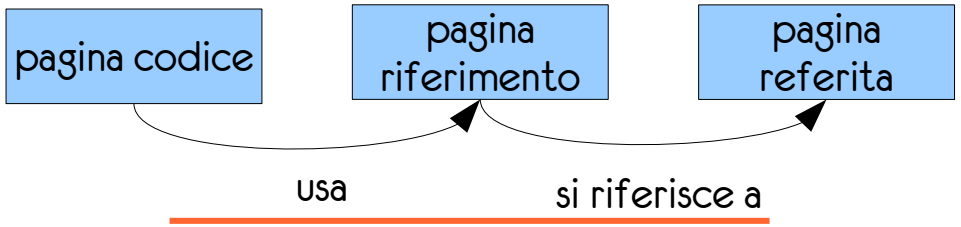
Numero minimo di frame

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



Numero minimo di frame

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

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

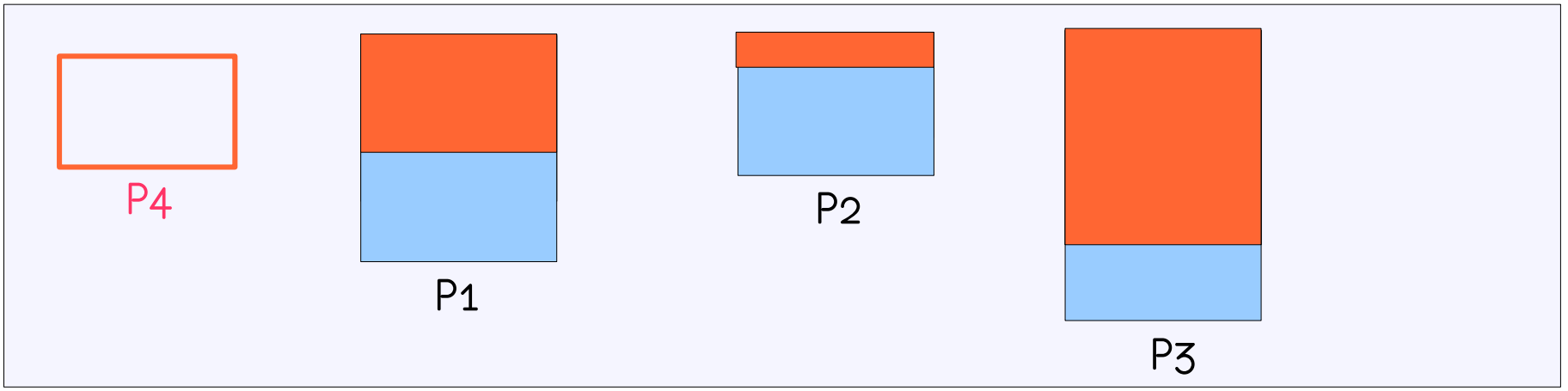
- Se abbiamo definito un num. minimo di frame necessari per caricare un'istruzione, potrebbe essere necessario incrementare tale valore di qualche unità per i processi più leggeri, decrementando di conseguenza i valori assegnati ai processi più pesanti

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

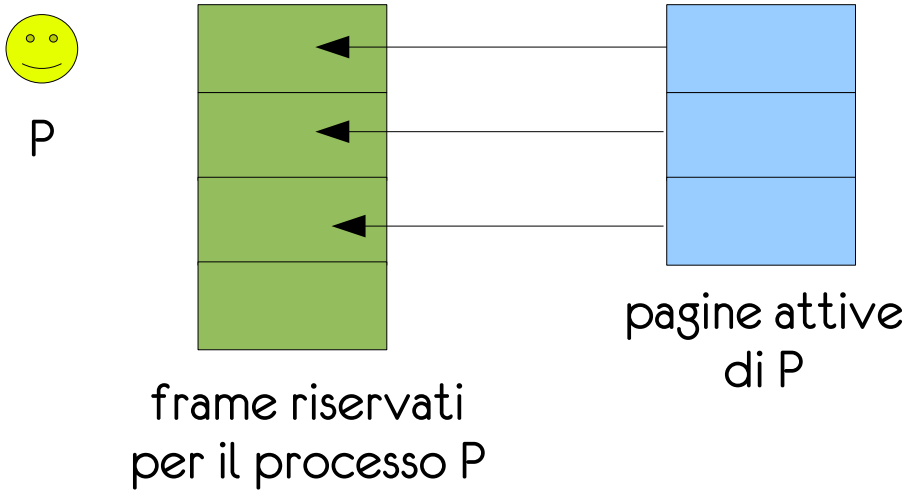
Di tanto in tanto l'ingresso di un nuovo processo causerà una riassegnazione dei frame rimasti liberi

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

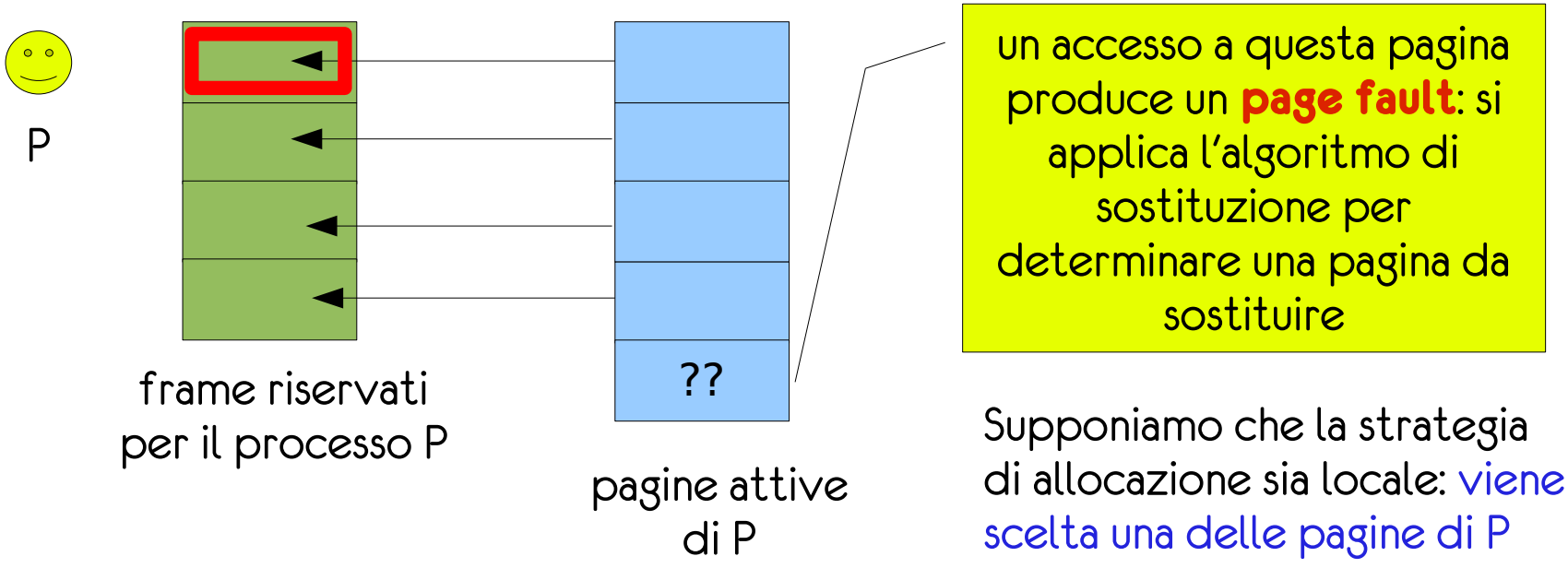
- un altro aspetto da discutere è un fenomeno noto come thrashing
- il thrashing è un fenomeno degenerativo che si può verificare nella gestione della memoria virtuale
- ogni processo ha un numero minimo di frame riservati
- l'esecuzione di ciascun processo si appoggia in ogni istante a un certo numero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

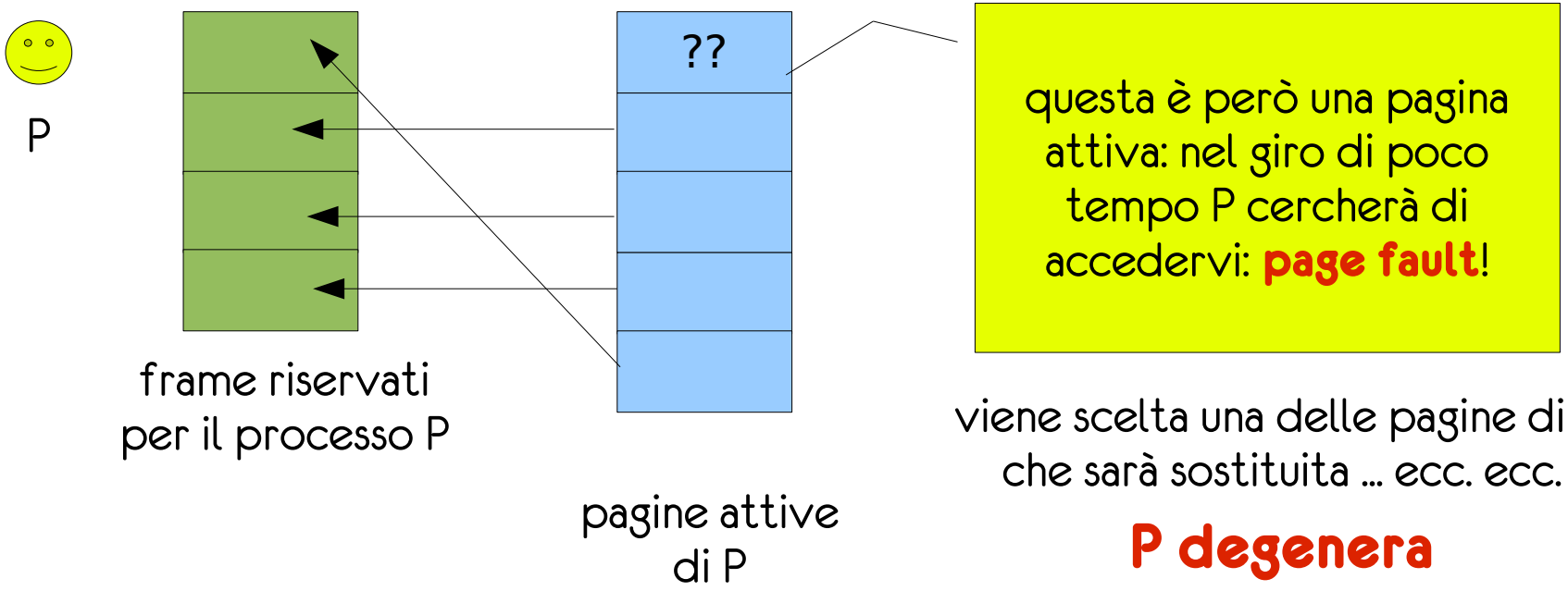
Thrashing

- 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”



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”

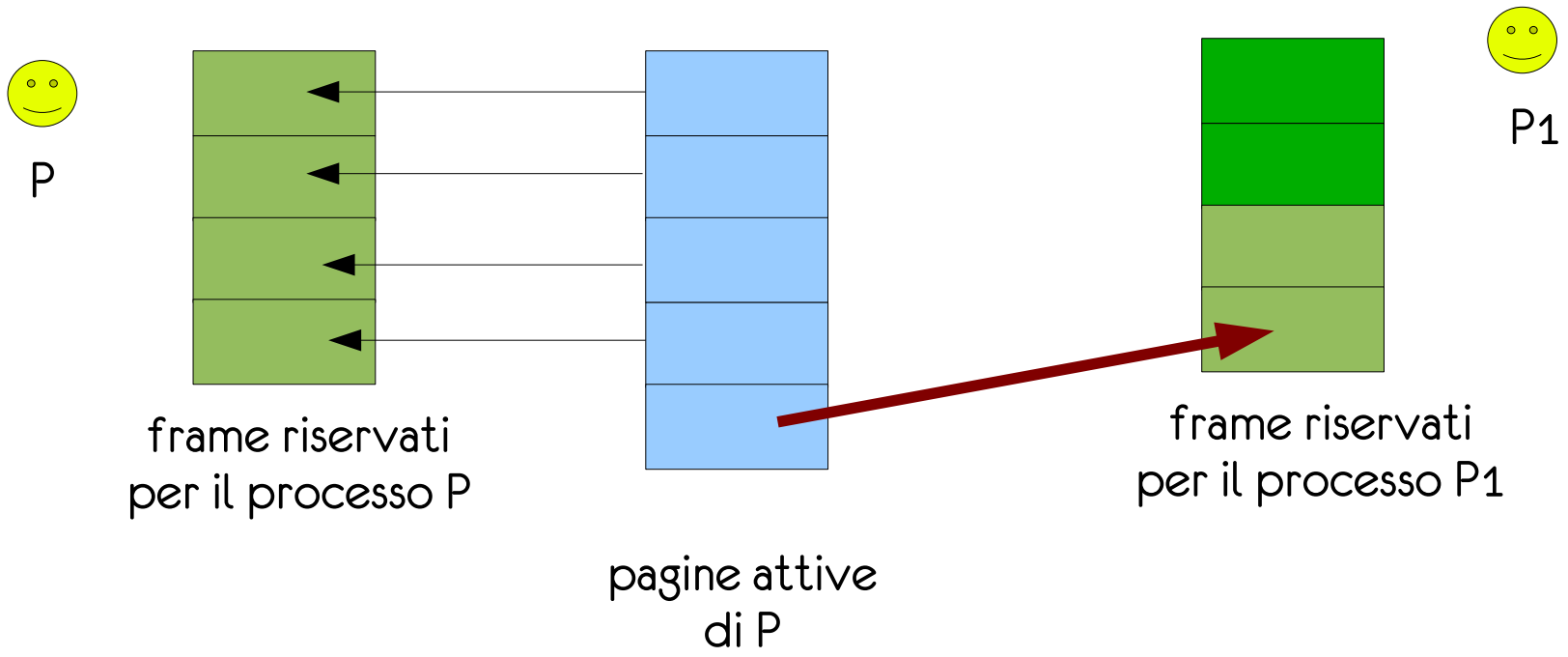


Thrashing

- 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

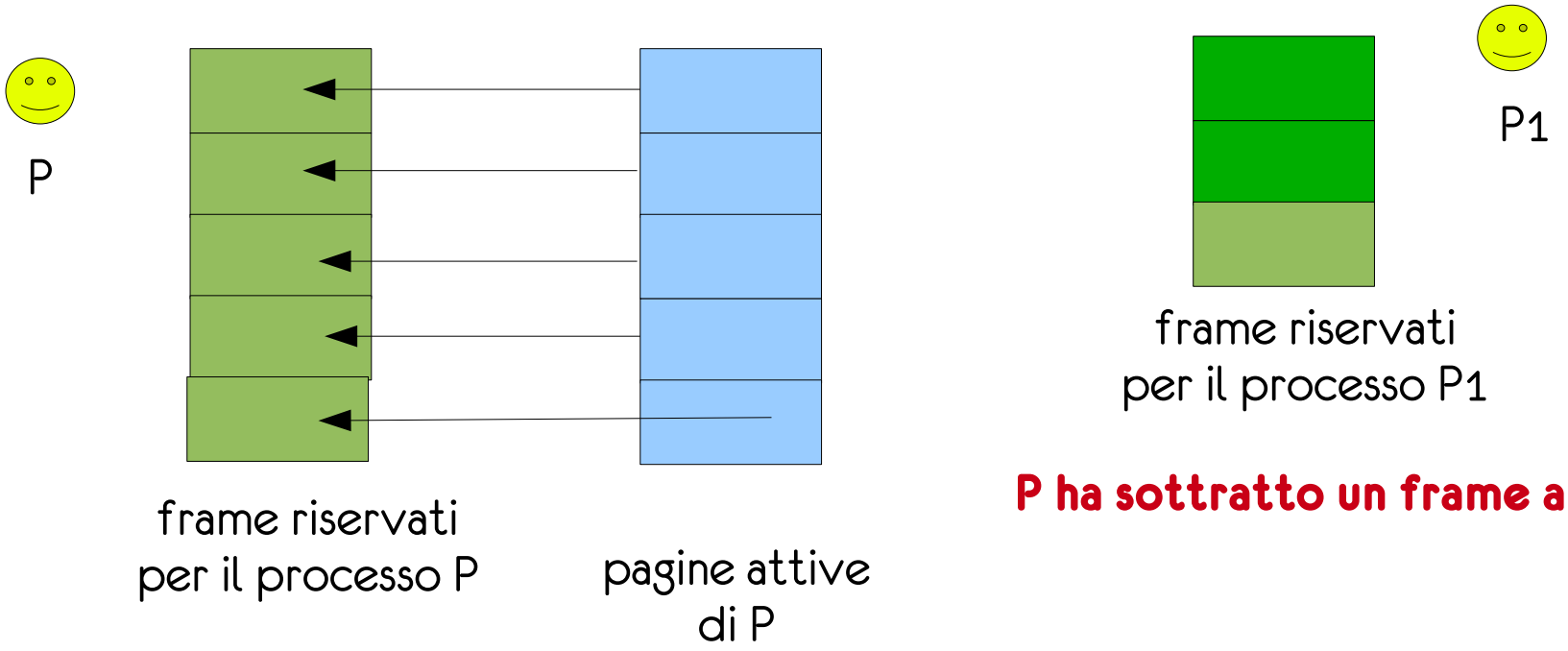
Thrashing

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



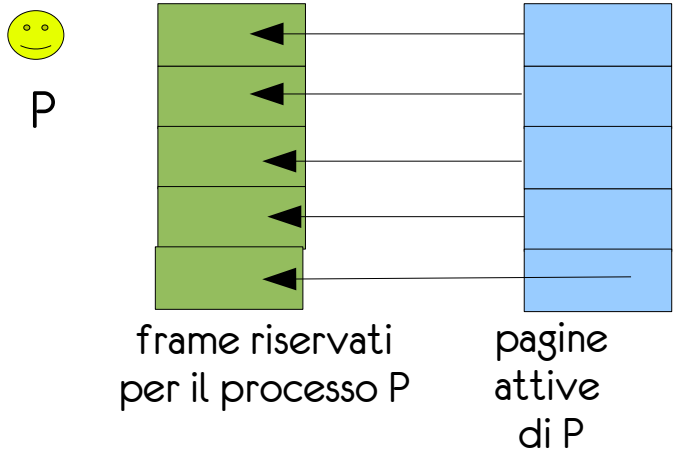
Thrashing

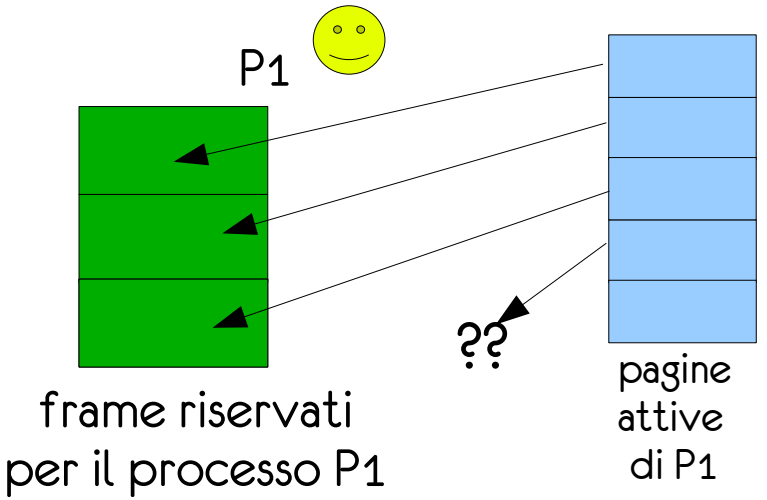
- e se il meccanismo di allocazione fosse globale?
- in questo caso potrebbe essere scelto un frame di un'altro processo



Thrashing

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

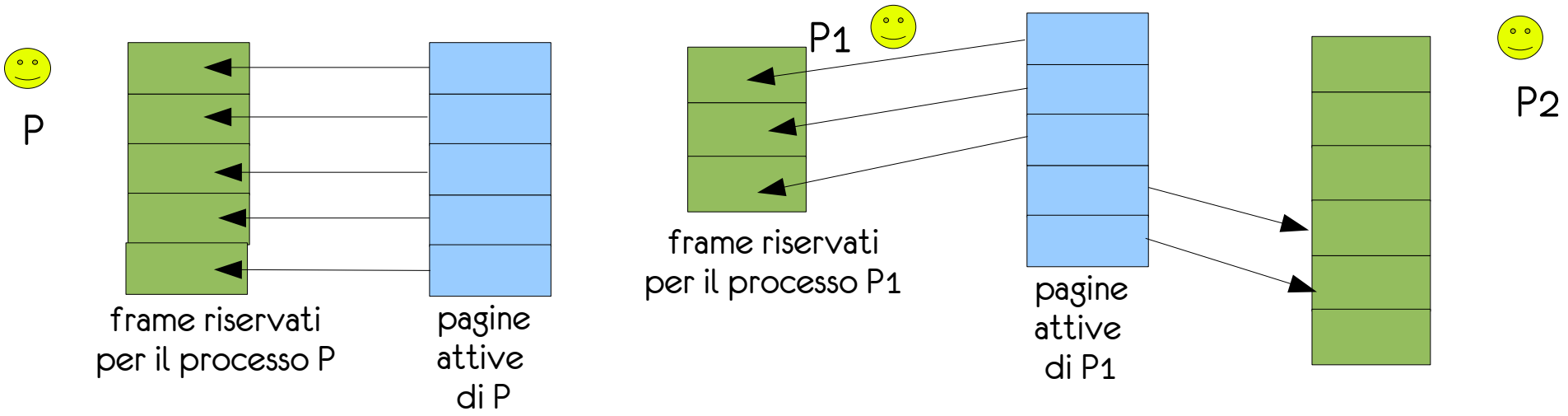




A questo punto se il numero di pagine attive per P1 cresce, P1 riempirà i propri frame residui ...

Thrashing

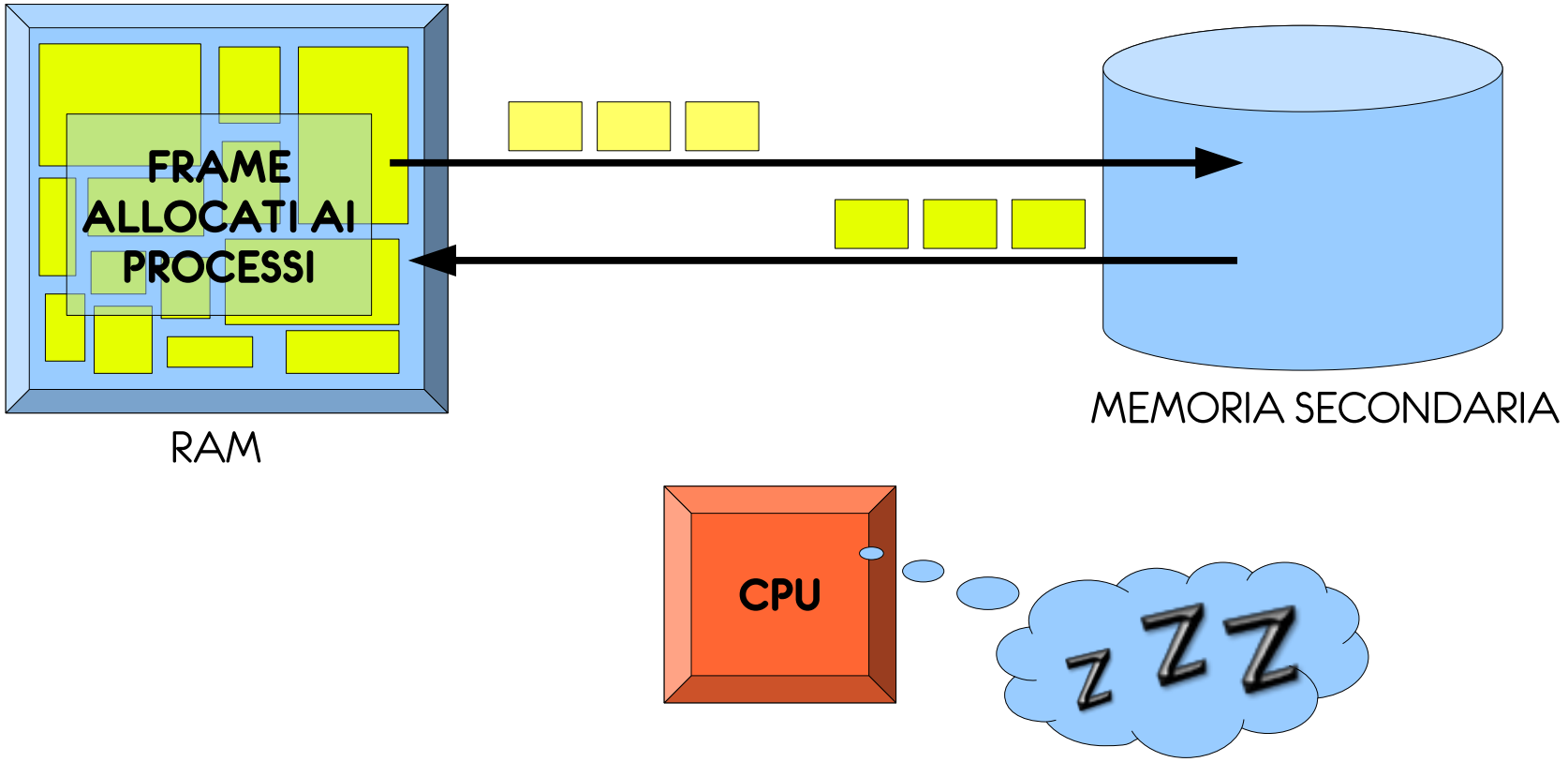
- e se il meccanismo di allocazione fosse globale?
- in questo caso potrebbe essere scelto un frame di un'altro processo



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

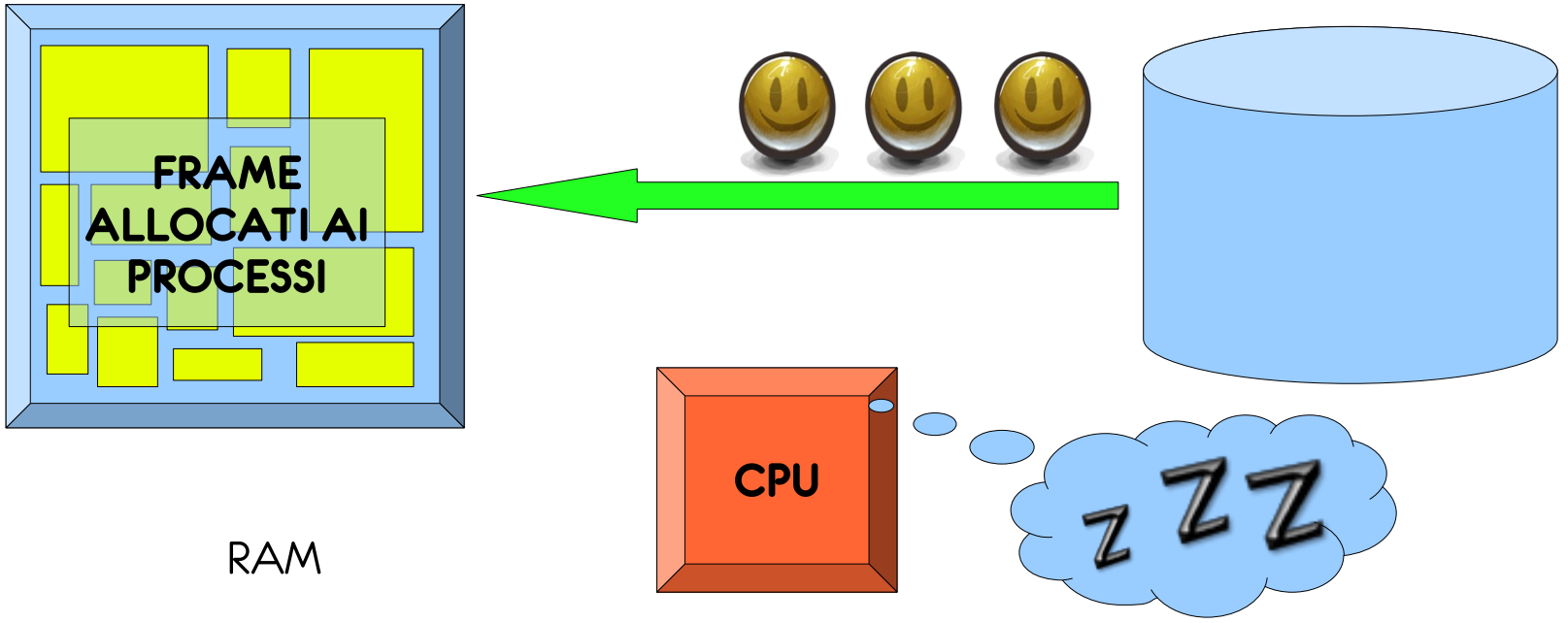
Effetti del thrashing

- quando si ha thashing 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



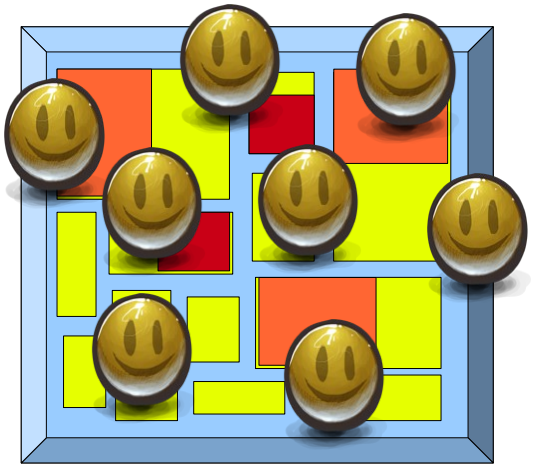
Thrashing e multiprogrammazione

- 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

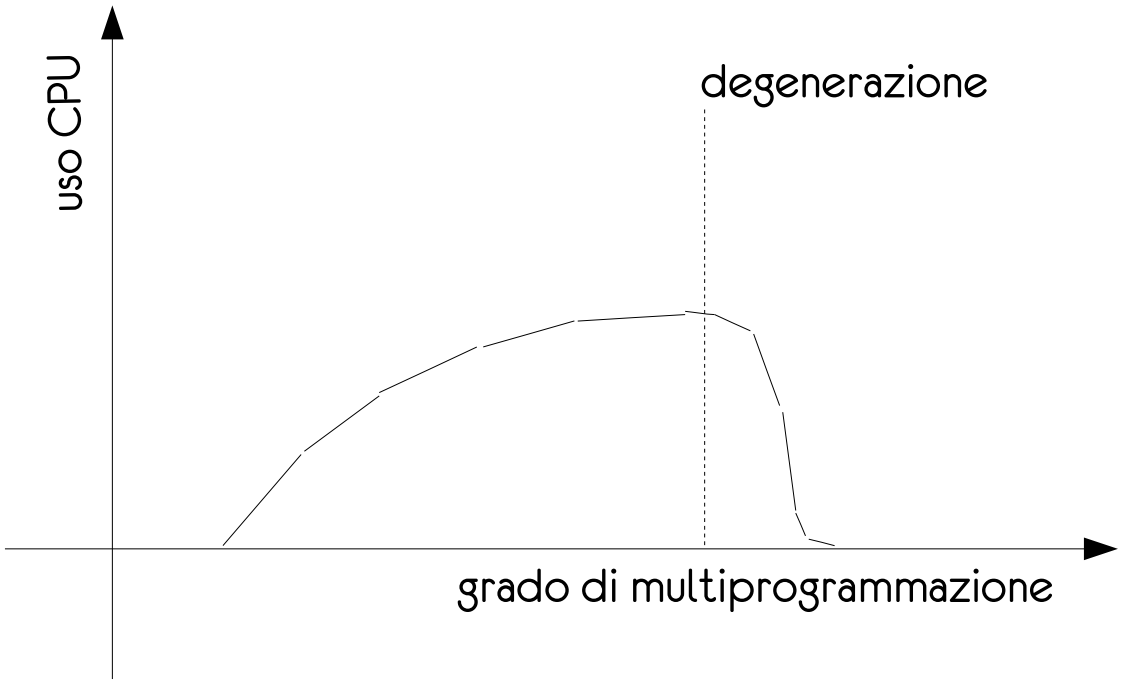
- quando si ha thashing perché i processi hanno a disposizione meno frame del numero di pagine attive, l'attività della CPU tende a diminuire (molto I/O)
- Molti SO monitorizzano l'uso della CPU: quando questo cala, se ci sono processi conservati in memoria secondaria, portano in RAM qualche processo 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



RAM

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, sotteaendo 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

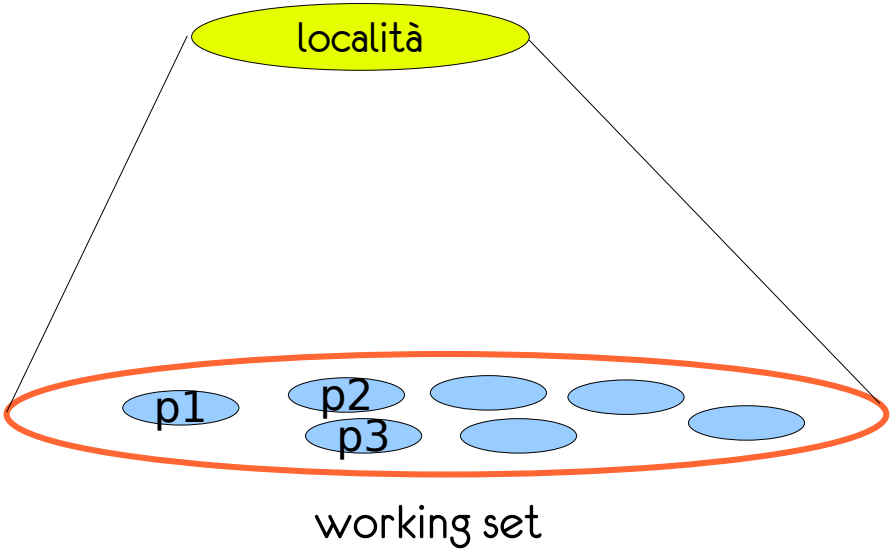
- l'ideale è cercare di prevedere di quante pagine avrà bisogno un processo e assegnargli un numero di frame sufficiente
- L'allocazione non sarà di tipo uniforme né sarà di tipo proporzionale ma dipenderà dal numero di pagine attive per ciascun processo
- non si distribuiscono tutti i frame liberi fra 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

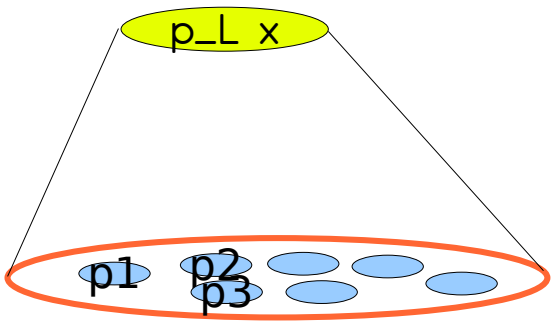
```
void push(lista *p_L, double x)
{
    printf("push %.2f \n",x);

    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 working 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 questione e le variabili locali menzionate



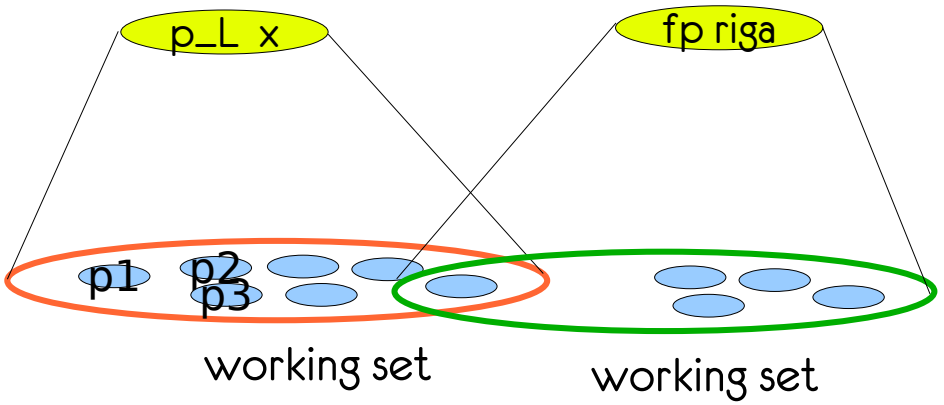
working set

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);
}
```

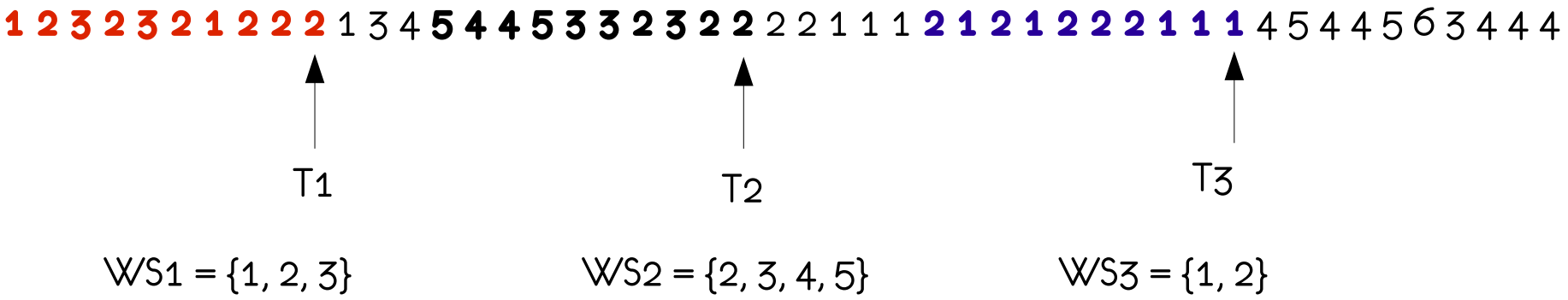
cambiando località
cambierà l'insieme delle pagine
coinvolto nell'esecuzione



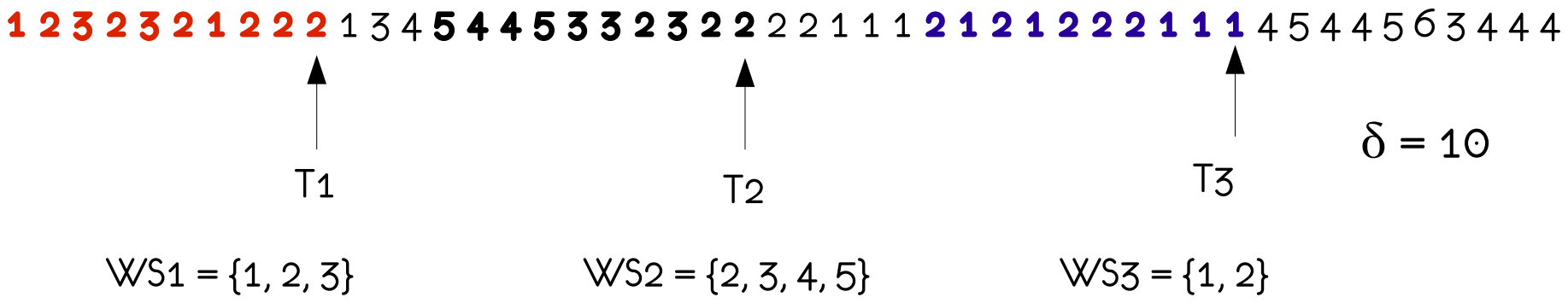
i diversi WS possono avere sovrapposizioni

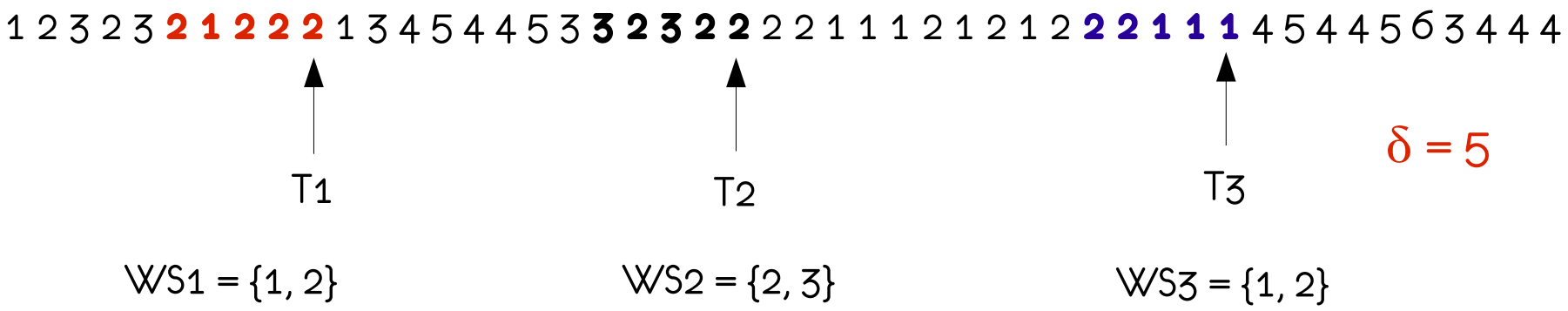
Modello del working set

- 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 $\delta = 10$

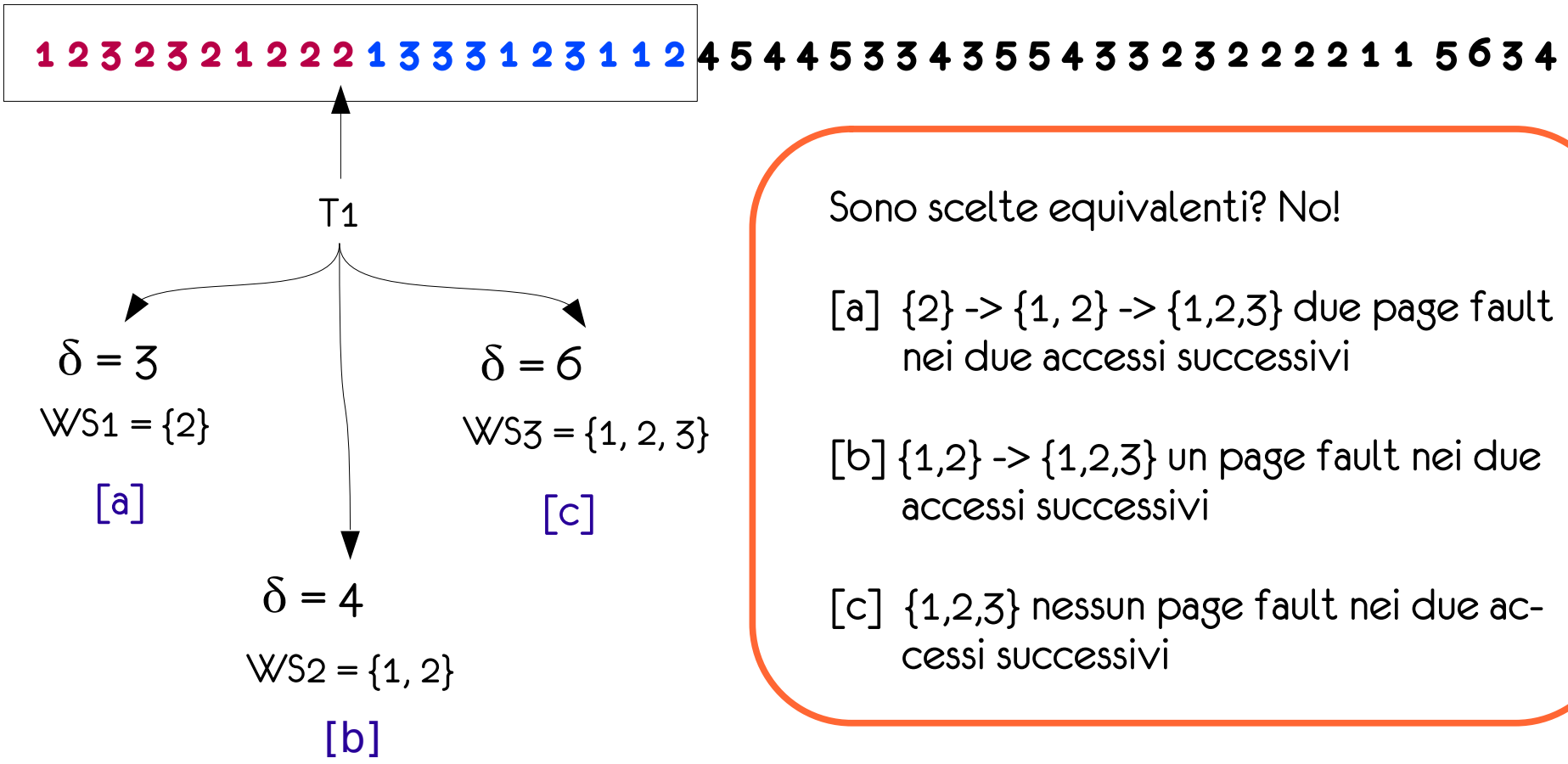


Working set e δ





Working set e δ



Sono scelte equivalenti? No!

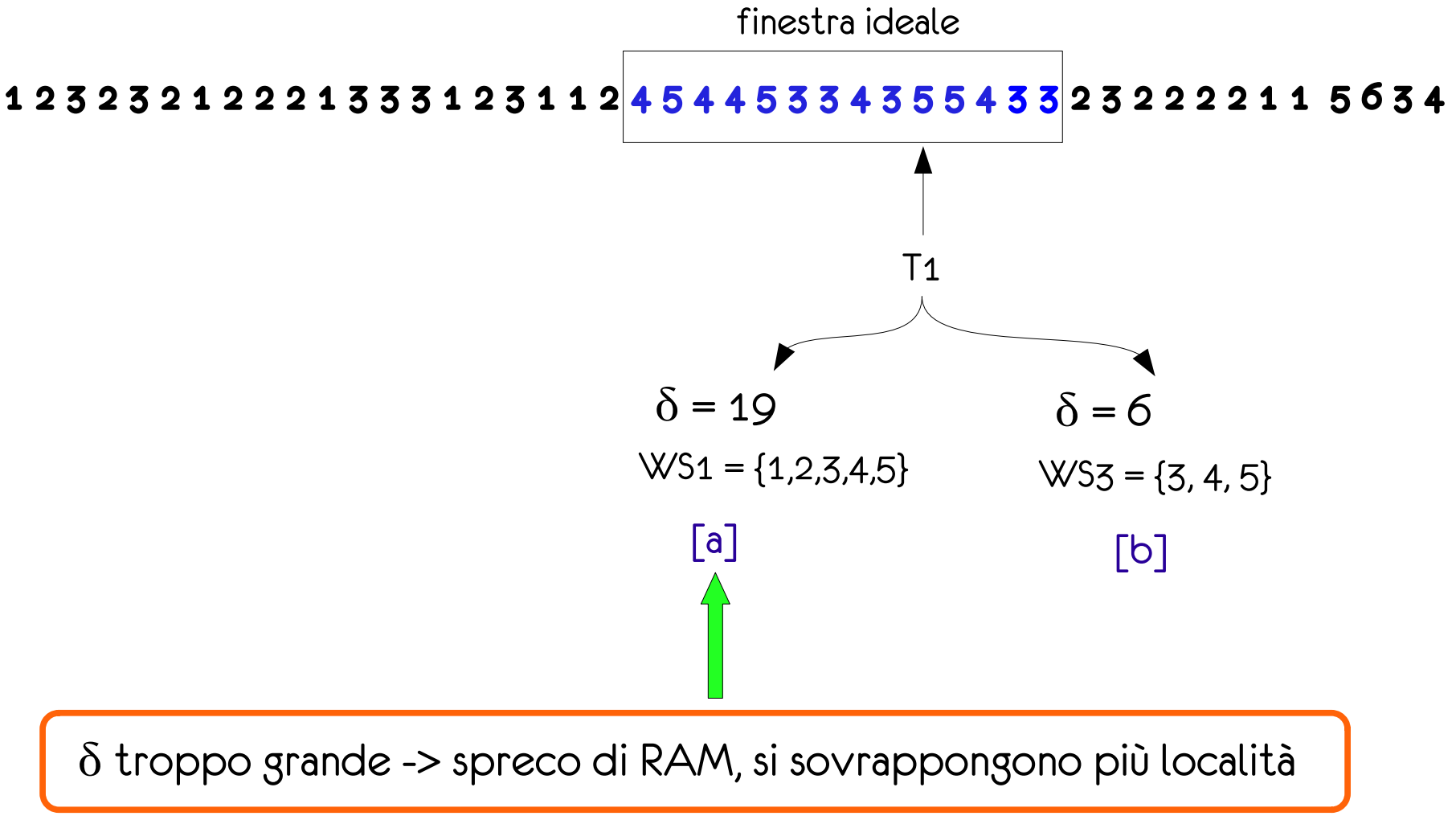
[a] {2} -> {1, 2} -> {1,2,3} due page fault nei due accessi successivi

[b] {1,2} -> {1,2,3} un page fault nei due accessi successivi

[c] {1,2,3} nessun page fault nei due accessi successivi

δ troppo piccolo non coglie l'intera località

Working set e δ



δ troppo grande -> spreco di RAM, si sovrappongono più località

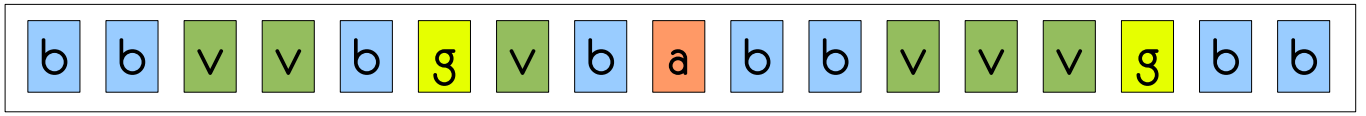
Uso del working set

- 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 degegnerazione 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

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 x 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

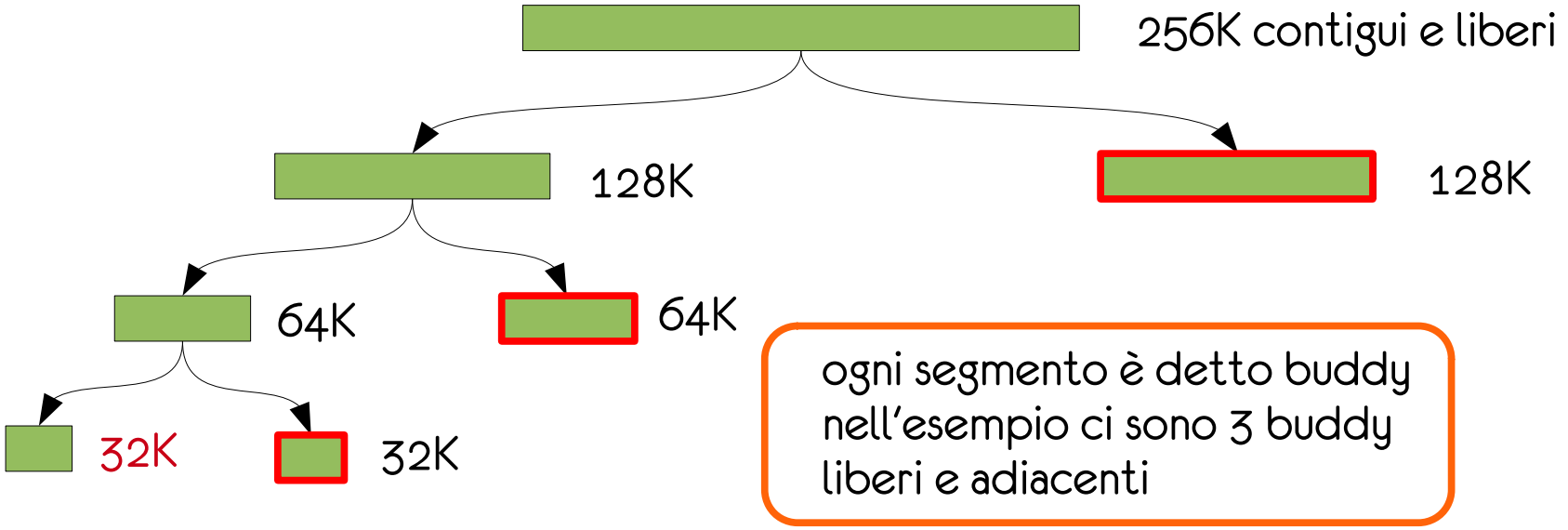
- per i processi utente
- per i processi kernel

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**

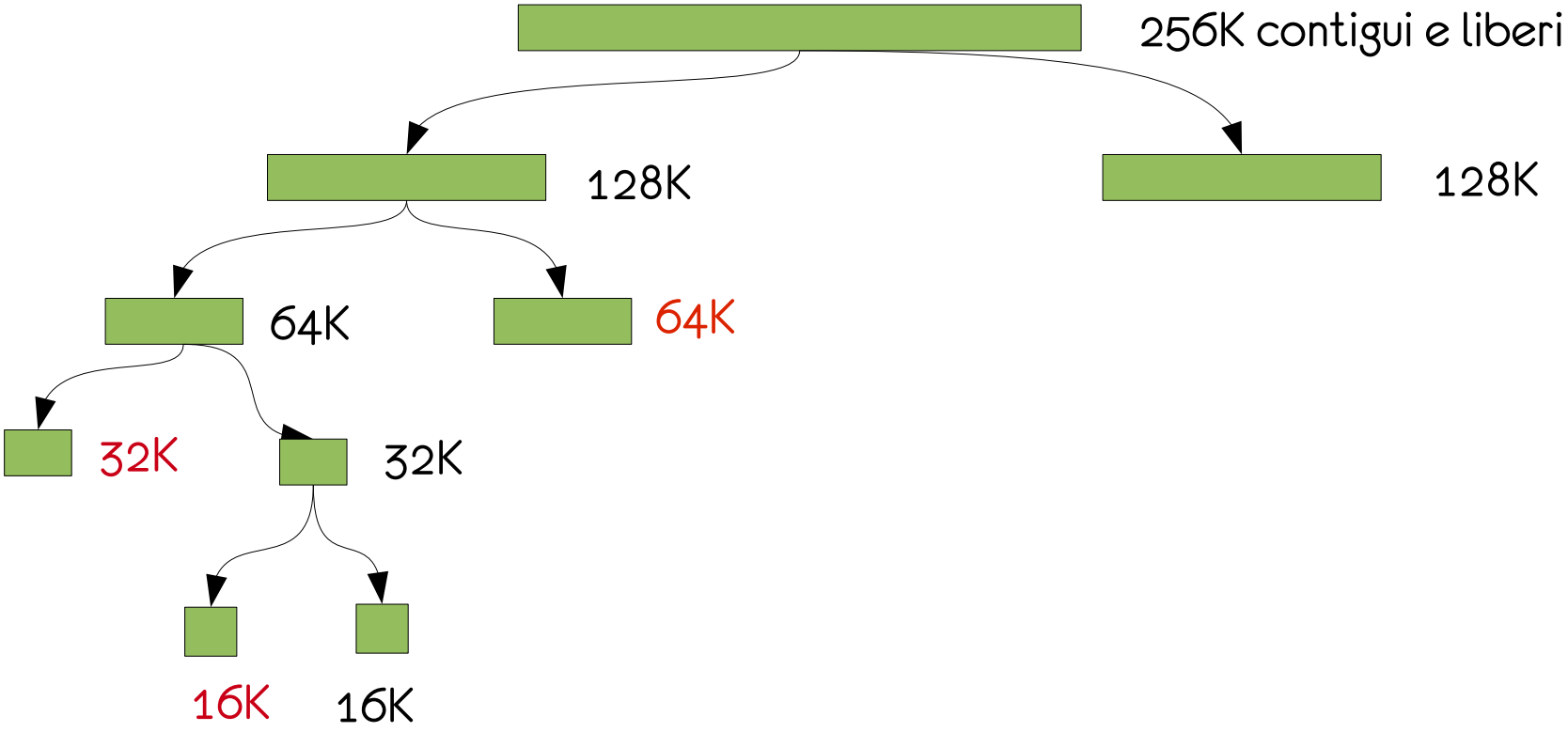
Sistema buddy

- 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:



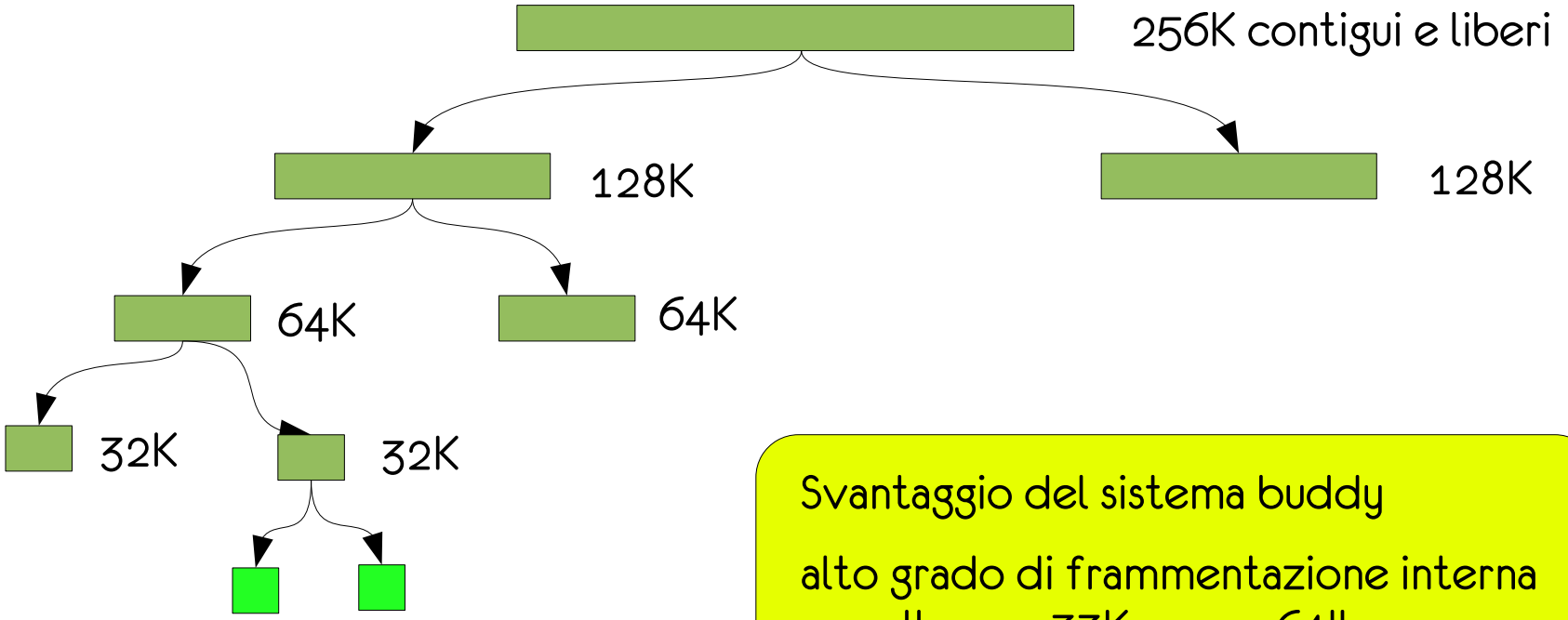
Sistema buddy

- 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



Svantaggio del sistema buddy

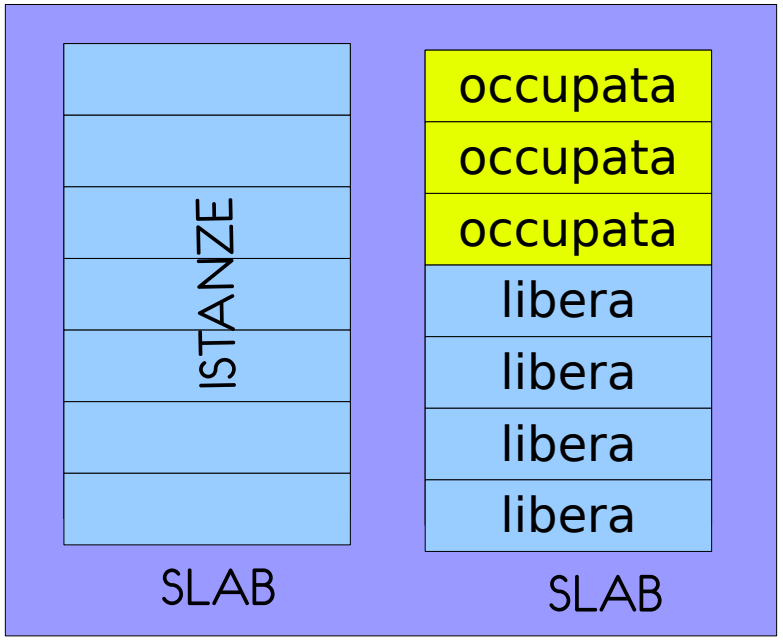
alto grado di frammentazione interna per allocare 33K ne uso 64!!

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.

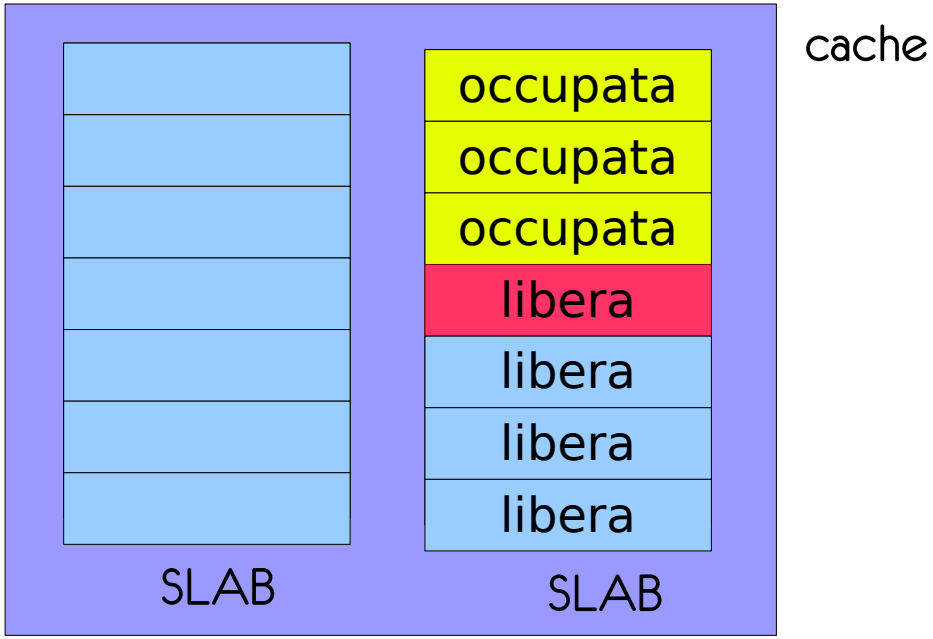
Allocazione a slab

- 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



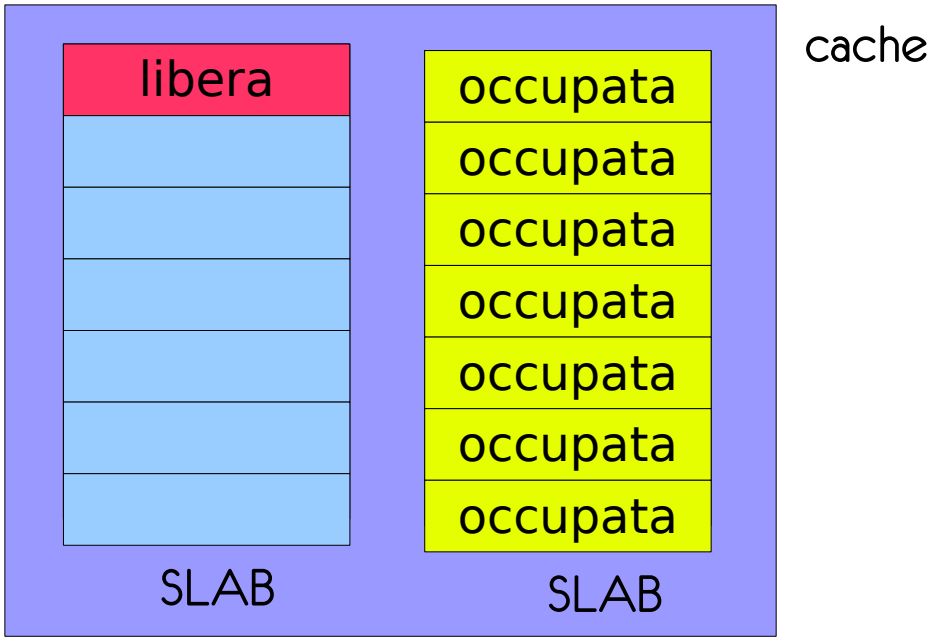
Allocazione

- 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



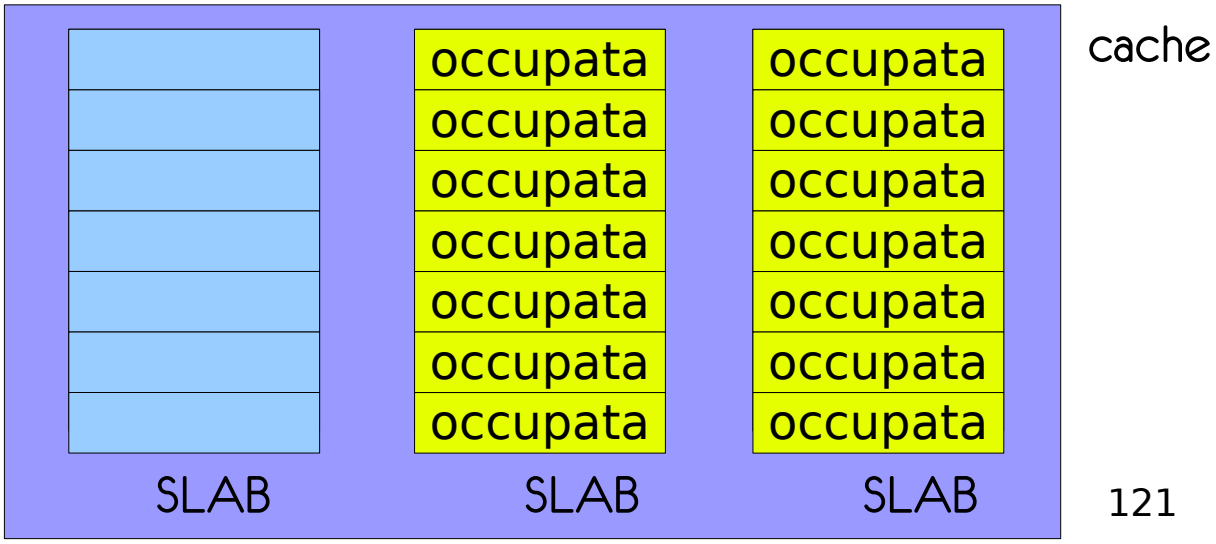
Allocazione

- 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>**

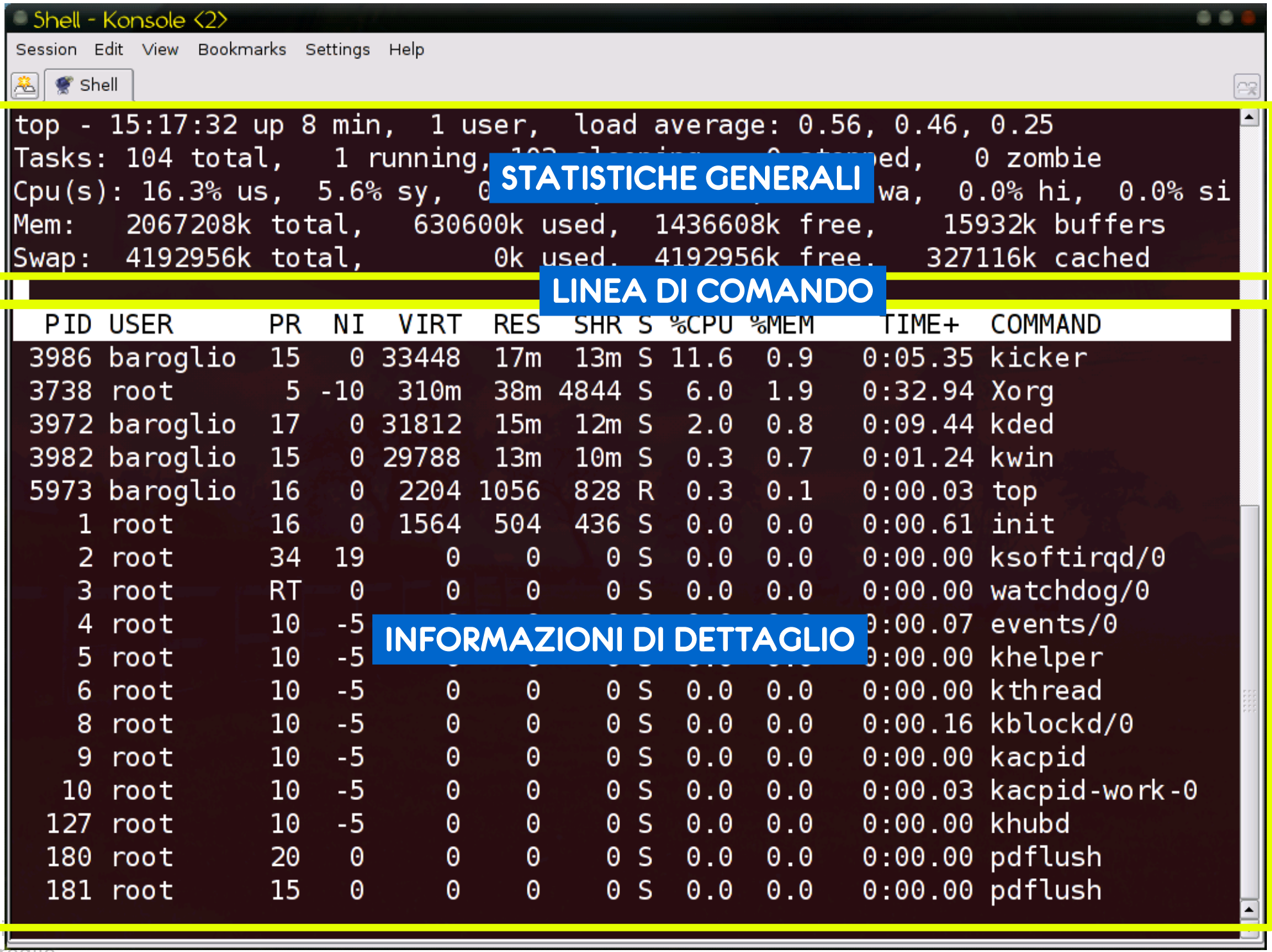


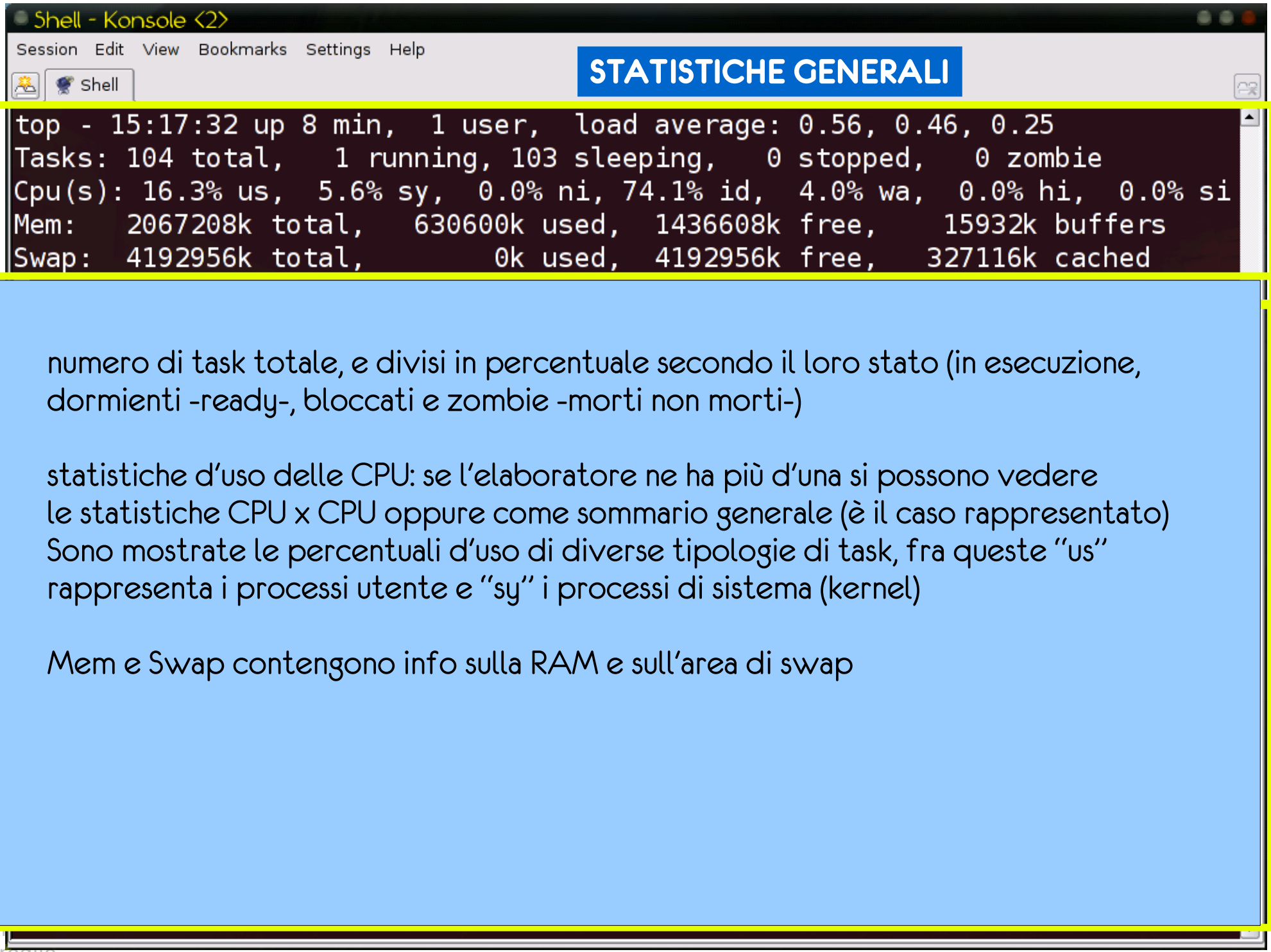
Allocazione

- 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

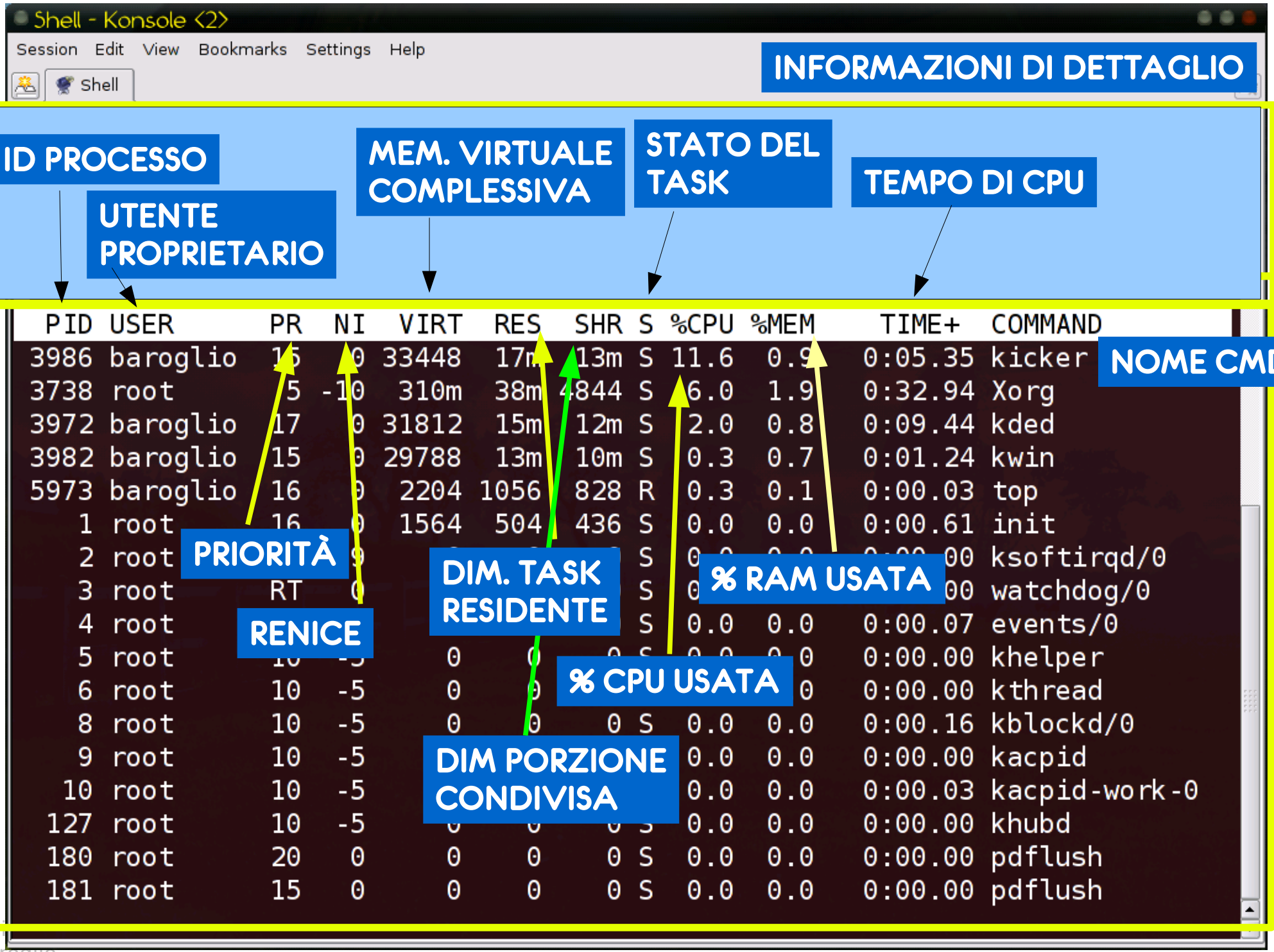

Shell - Konsole <2>
Session Edit View Bookmarks Settings Help
Shell
top - 15:17:32 up 8 min, 1 user, load average: 0.56, 0.46, 0.25
Tasks: 104 total, 1 running, 0 zombie
Cpu(s): 16.3% us, 5.6% sy, 0.0% hi, 0.0% si
Mem: 2067208k total, 630600k used, 1436608k free, 15932k buffers
Swap: 4192956k total, 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
INFORMAZIONI DI DETTAGLIO
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

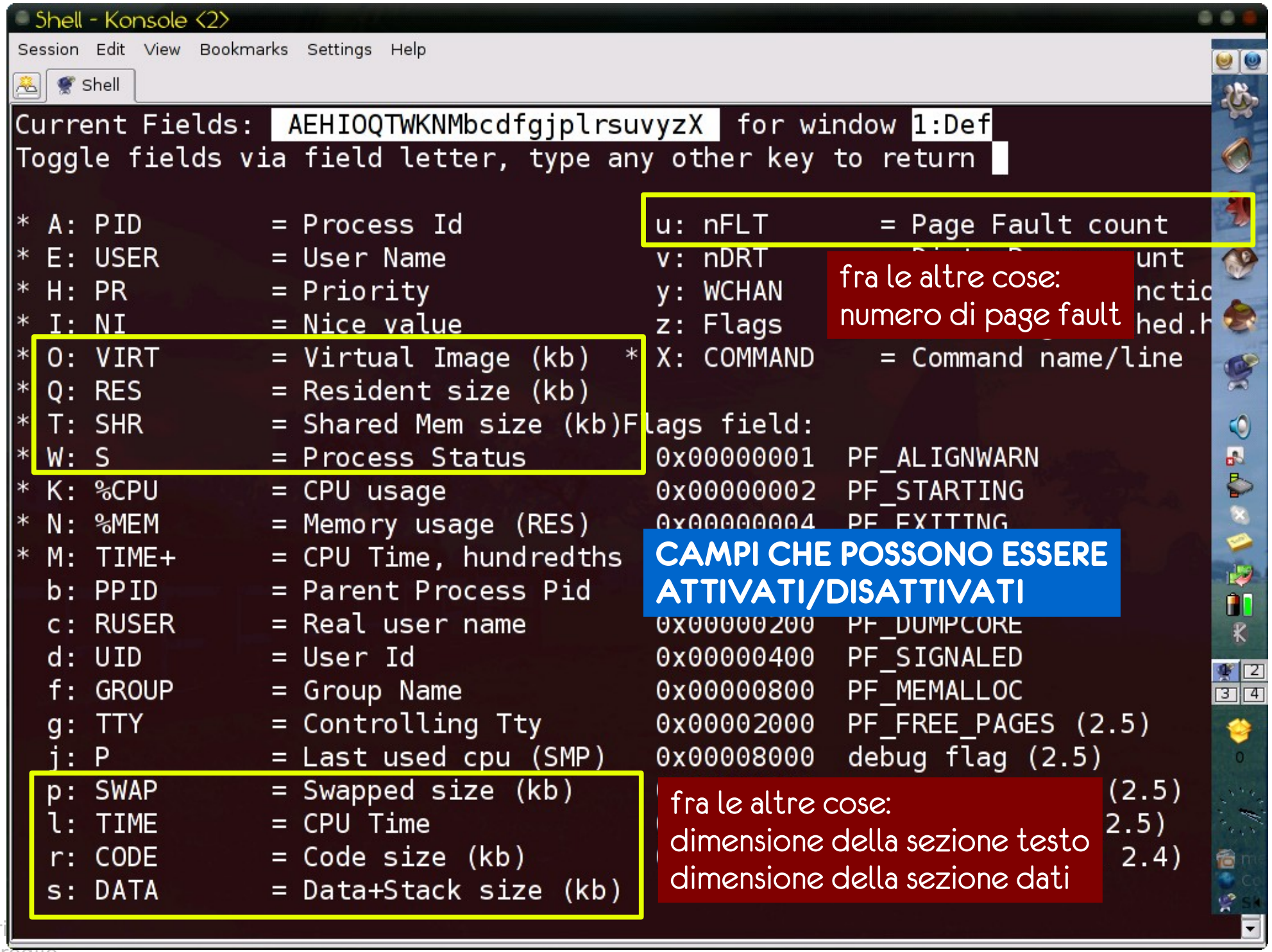


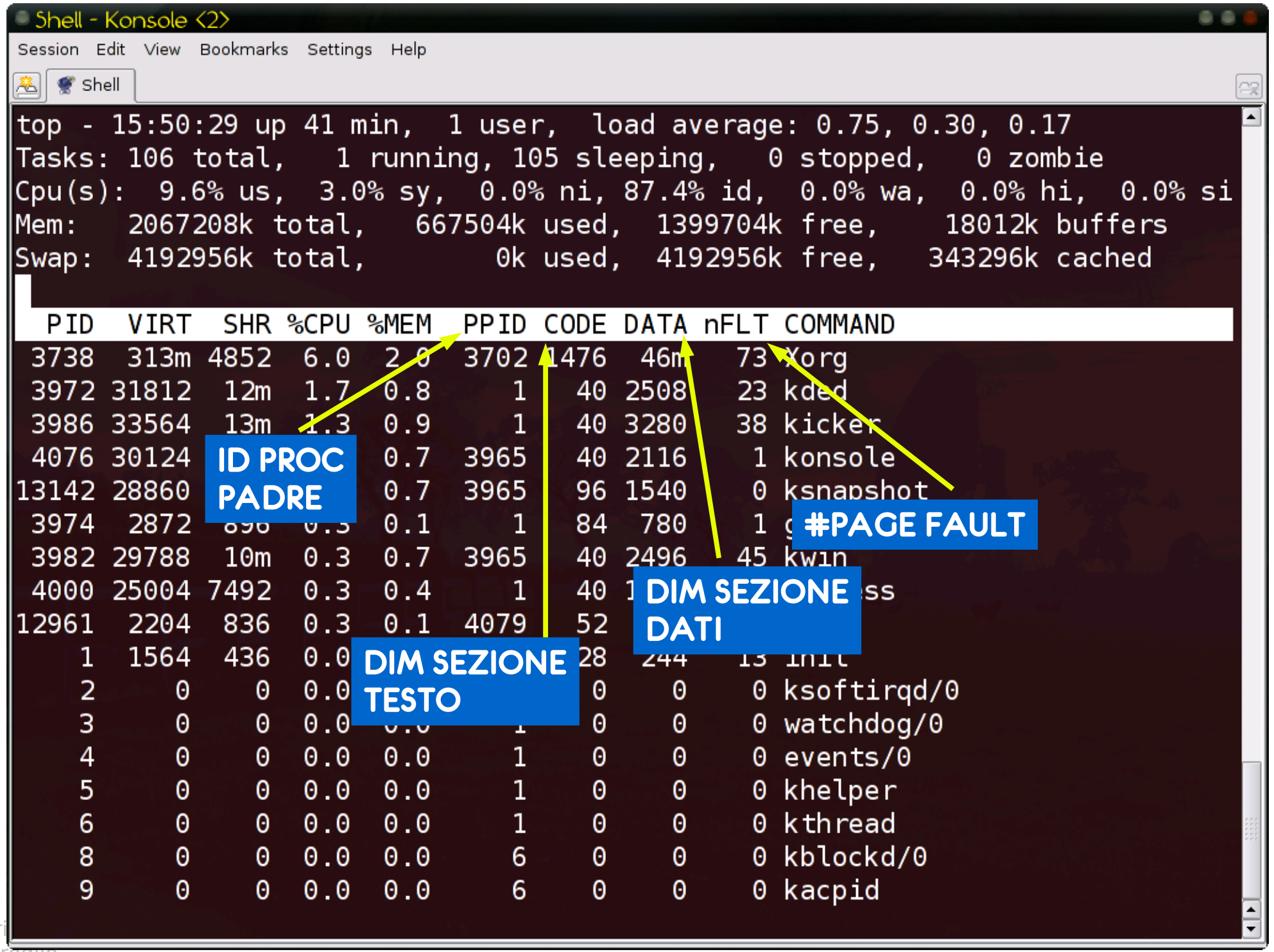
numero di task totale, e divisi in percentuale secondo il loro stato (in esecuzione, dormienti -ready-, bloccati e zombie -morti non morti-)

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


Shell - Konsole <2>
Session Edit View Bookmarks Settings Help
Shell
INFORMAZIONI DI DETTAGLIO
ID PROCESSO
UTENTE PROPRIETARIO
MEM. VIRTUALE COMPLESSIVA
STATO DEL TASK
TEMPO DI CPU
NOME CMD
PRIORITÀ
RENICE
DIM. TASK RESIDENTE
% RAM USATA
% CPU USATA
DIM PORZIONE CONDIVISA
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 S 00 ksoftirqd/0
3 root RT S 00 watchdog/0
4 root S 0.0 0.0 0:00.07 events/0
5 root 0 0 0 0 0:00.00 khelper
6 root 10 -5 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 0.0 0:00.00 kacpid
10 root 10 -5 0.0 0.0 0:00.03 kacpid-work-0
127 root 10 -5 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


Shell - Konsole <2>
Session Edit View Bookmarks Settings Help
Shell
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_EXITING
0x00000200 PF_DUMPCORE
0x00000400 PF_SIGNALED
0x00000800 PF_MEMALLOC
0x00002000 PF_FREE_PAGES (2.5)
0x00008000 debug flag (2.5)
fra le altre cose:
numero di page fault
CAMPI CHE POSSONO ESSERE ATTIVATI/DISATTIVATI
fra le altre cose:
dimensione della sezione testo
dimensione della sezione dati


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 0.7 3965 40 2116 1 konsole
13142 28860 0.7 3965 96 1540 0 ksnapshot
3974 2872 0.1 1 84 780 1
3982 29788 10m 0.3 0.7 3965 40 2496 45 kwin
4000 25004 7492 0.3 0.4 1 40
12961 2204 836 0.3 0.1 4079 52
1 1564 436 0.0
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 0.0 1 0 0 0 events/0
5 0 0 0.0 0.0 1 0 0 0 khelper
6 0 0 0.0 0.0 1 0 0 0 kthread
8 0 0 0.0 0.0 6 0 0 0 kblockd/0
9 0 0 0.0 0.0 6 0 0 0 kacpid
ID PROC PADRE
DIM SEZIONE TESTO
DIM SEZIONE DATI
#PAGE FAULT

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>>
```

vmstat -s

```
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 -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

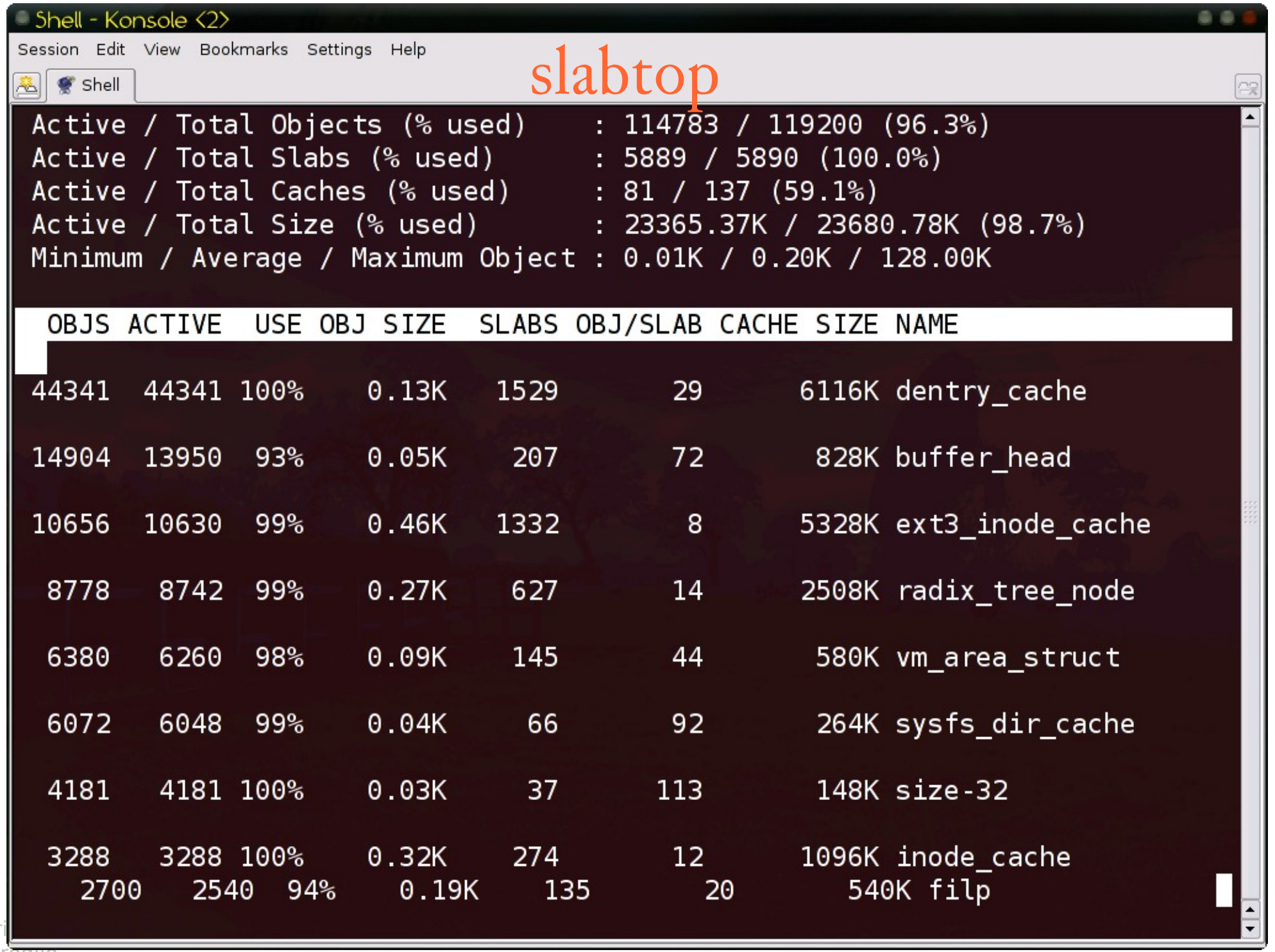
vmstat -m

```
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