Lezione 6
Schedulazione CPU

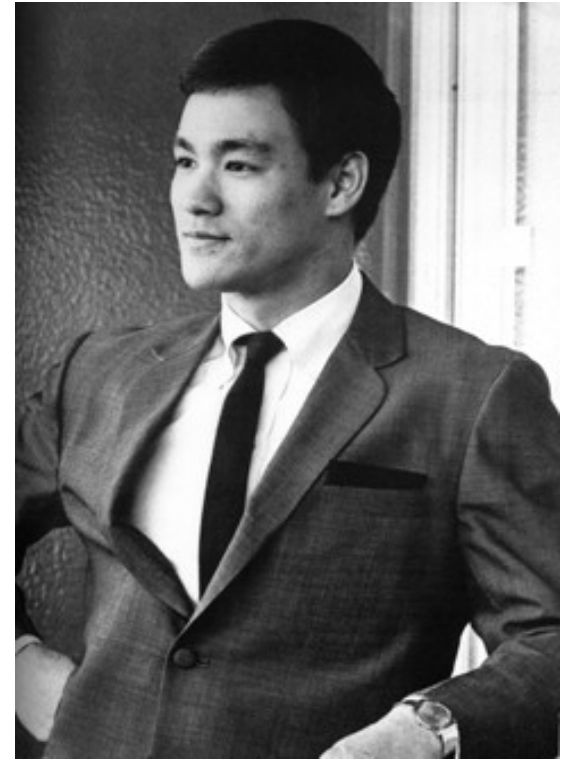
Sistemi Operativi (9 CFU), CdL Informatica, A. A. 2018/2019
Dipartimento di Scienze Fisiche, Informatiche e Matematiche
Università di Modena e Reggio Emilia
http://weblab.ing.unimo.it/people/andreolini/didattica/sistemi-operativi

Quote of the day

(Meditate, gente, meditate...)

“If you spend too much time thinking about a thing, you'll never get it done. Make at least one definite move daily toward your goal.”

Lee Jun Fan (1940-1973)
Filosofo, attore, esperto di arti marziali



INTRODUZIONE

Lo scenario

(SO moderno, multiutente, multiprogrammato)

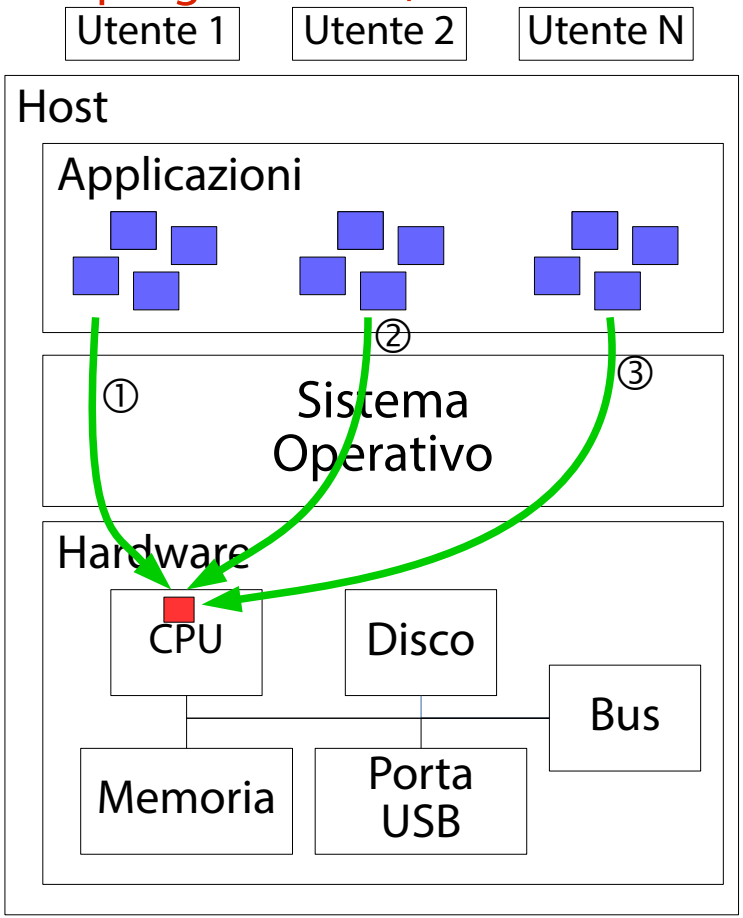
Si consideri un SO moderno, multiutente, multiprogrammato.
I processi in esecuzione sono tanti. Contateli!

```
ps -el | wc -l
```

Le CPU sono poche. Contatele!

```
nproc
```

→ Si rende necessario il **time sharing**.



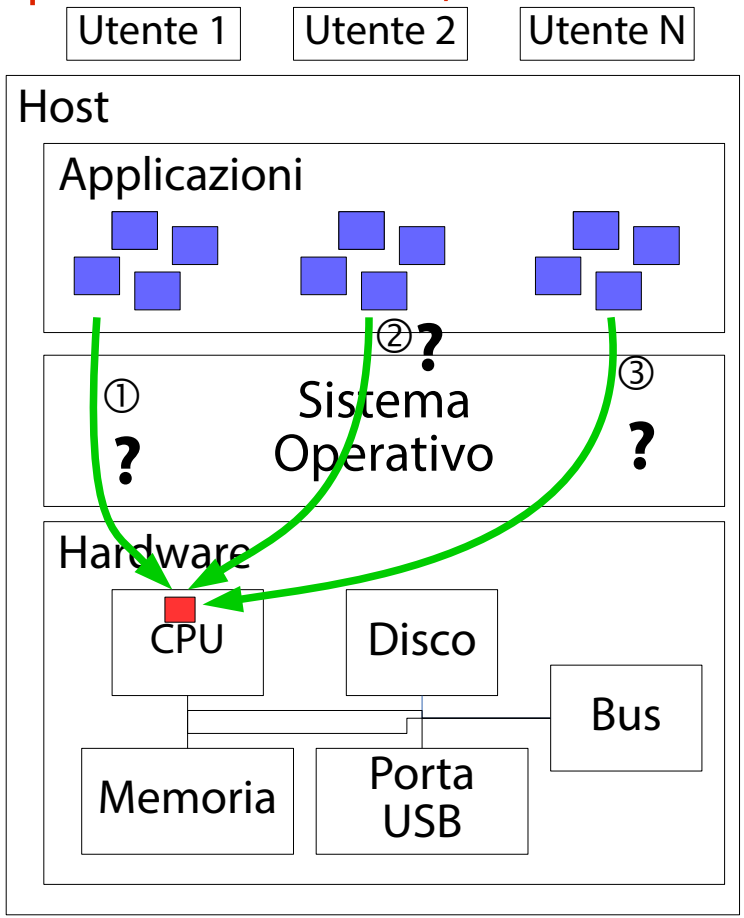
Problema 1/3

(Ripristinare l'esecuzione di un processo sulla CPU)

Il nucleo si trova spesso nella situazione di ripristinare l'esecuzione di un processo Q a scapito di un altro processo P che viene bloccato.

- Scadenza quanto di tempo P.
- P chiede servizio bloccante.

Come fa il SO a ripristinare l'esecuzione di un processo sulla CPU?



Problema 2/3

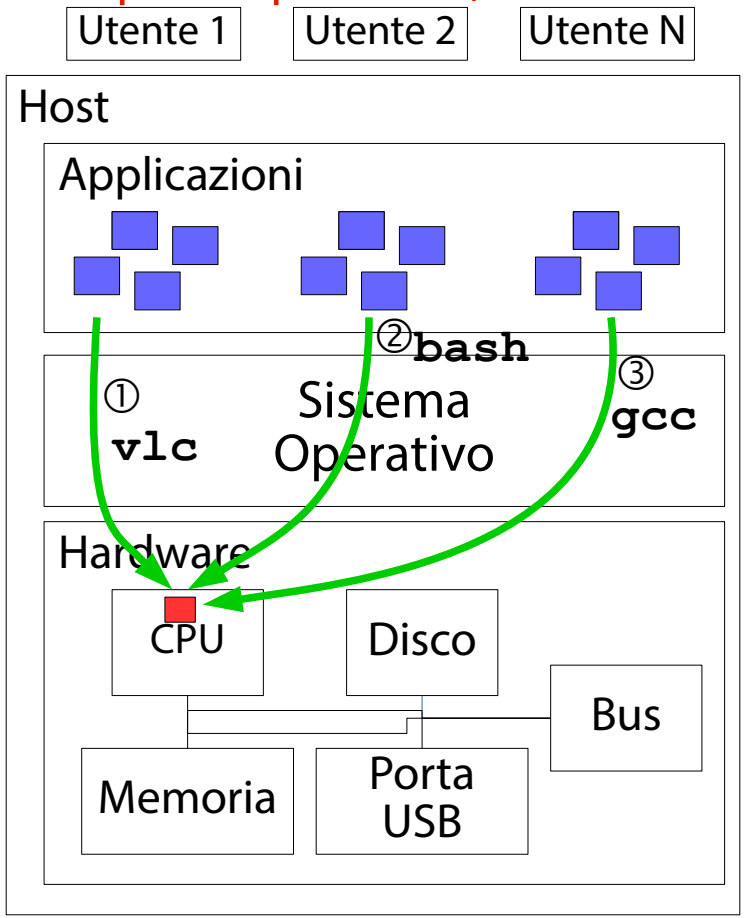
(Scegliere preferibilmente i processi più "importanti")

Alcuni processi sono più "importanti di altri".

- Quelli da cui l'utente si aspetta maggiore interattività: giochi, riprodutori audio e video, …
- Quelli di servizio per il SO.

Il SO è in grado fornire una nozione di "importanza" di un processo?

Il SO sa scegliere per primi i processi più "importanti"?



Problema 3/3

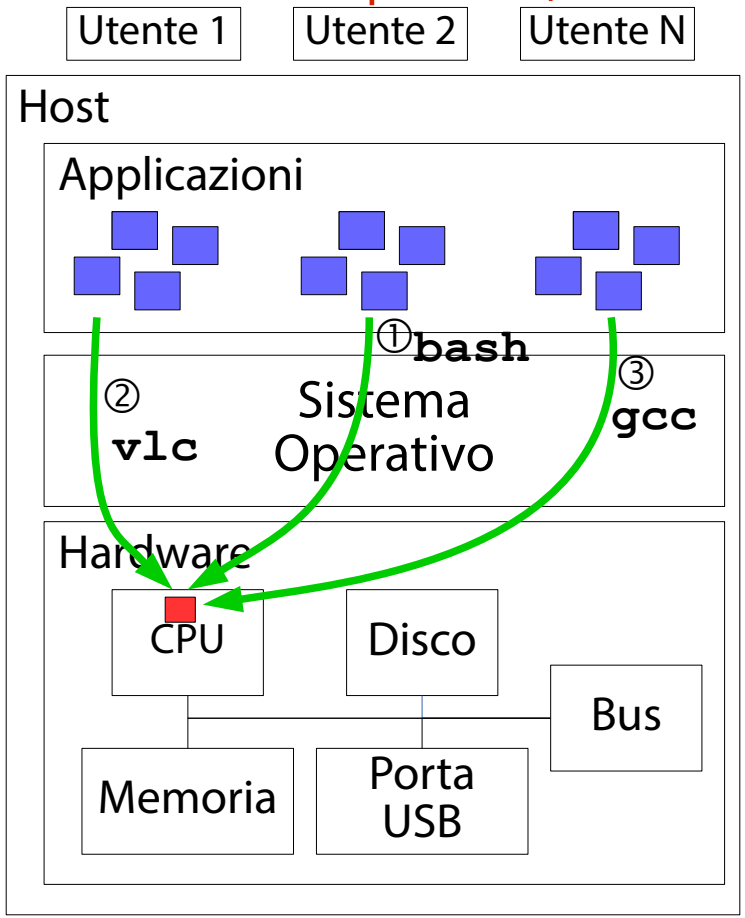
(Monitorare e controllare le procedure di scelta dei processi)

L'utente è interessato a capire e modificare le scelte effettuate dal SO nell'ambito di esecuzione dei processi. Scenari di esempio:

- Chi va in esecuzione più spesso? Perché?
- È possibile alterare in maniera temporanea l'importanza dei processi?

È possibile esaminare tali scelte?

È possibile alterare le scelte?



Soluzione: gestore dei processi

(Affronta e risolve (si spera!) i problemi ora esposti)

Il nucleo del SO implementa una libreria di funzioni, detta **gestore dei processi**.

L'obiettivo del gestore dei processi è di affrontare e risolvere i tre problemi ora visti.

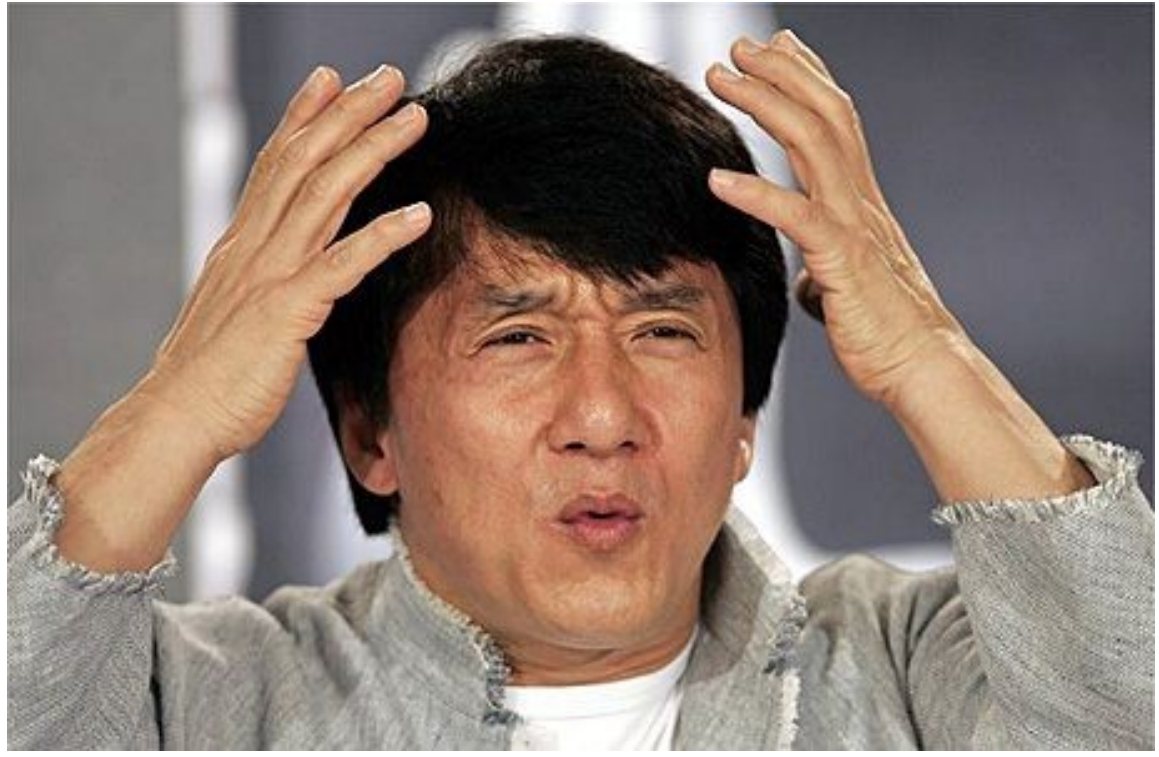
Il gestore dei processi si occupa, nello specifico, dei seguenti aspetti.

- Gestione delle attese.
- Scelta del prossimo processo da eseguire.
- Cambio di contesto fra un processo ed il successivo.

Il gestore dei processi svolge tali attività tramite uno **schedulatore di processi**.

“Schedulatore dei processi”?

(Eh?)



Schedulazione

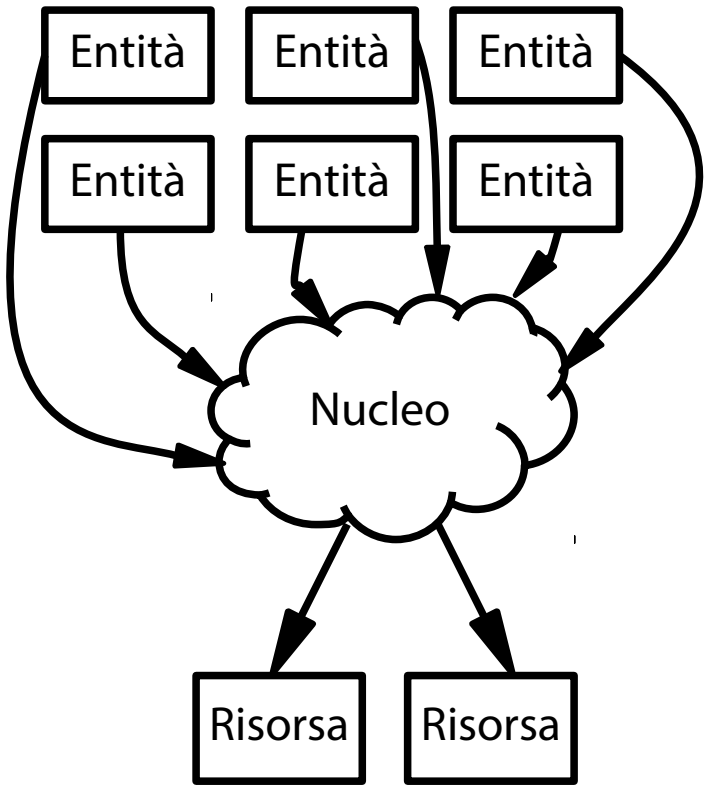
(Risolve il Problema 1/3)

In parecchie situazioni, il nucleo deve assegnare una risorsa ad n entità che la vogliono utilizzare.

Entità: processi utente, altri componenti del nucleo.

<u>Vincolo fondamentale:</u> le entità richiedenti eccedono le istanze di risorsa presenti.

Come assegnare le richieste alle risorse?

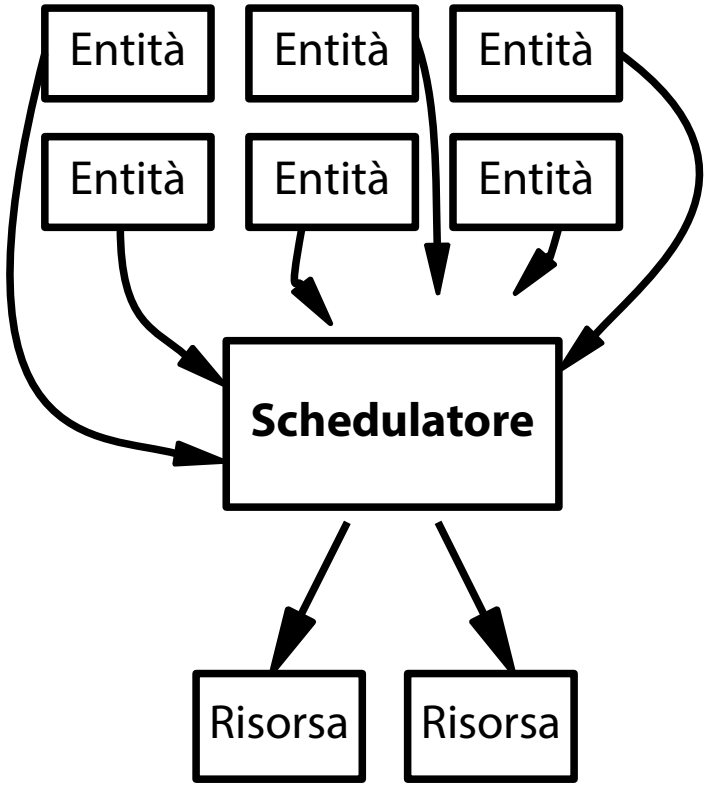


Lo schedulatore

(http://www.youtube.com/watch?v=ptuo3e3u6T8)

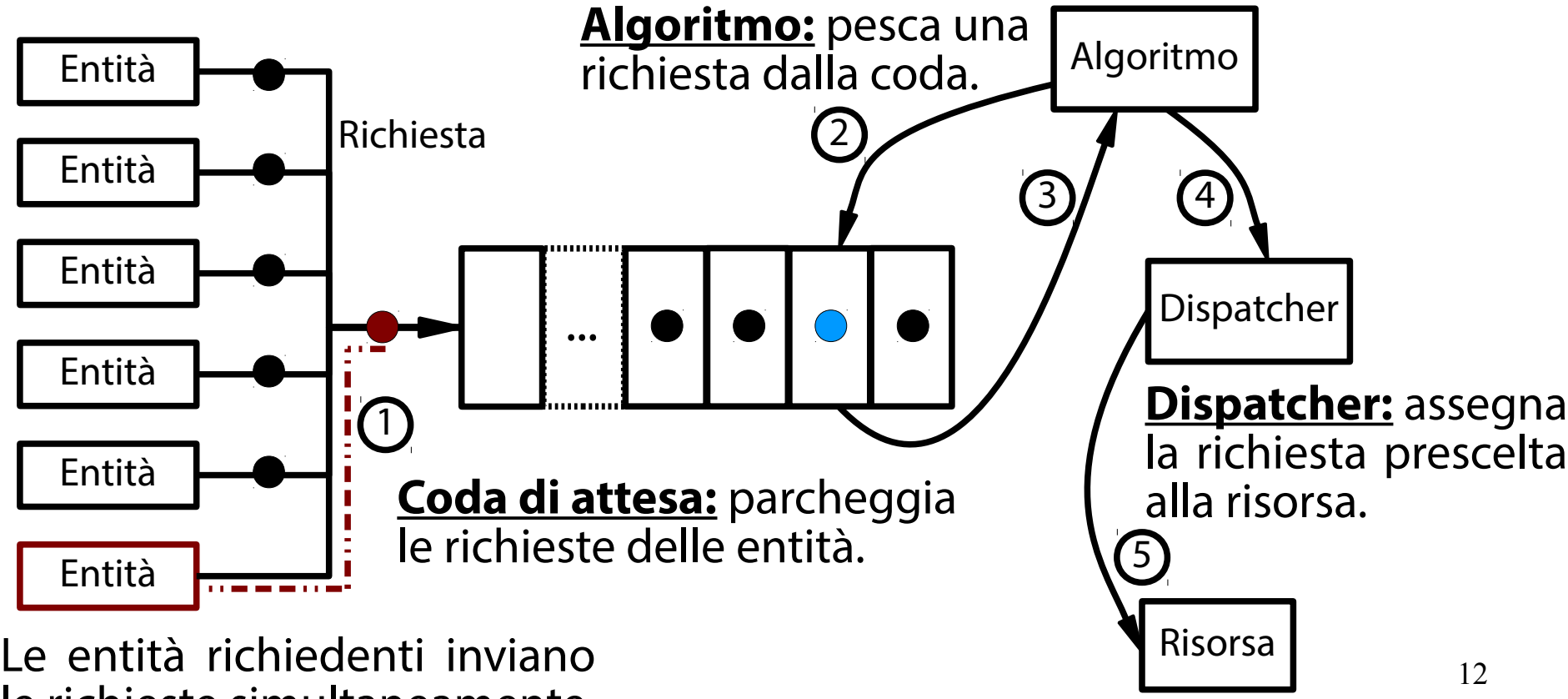
Uno **schedulatore** (**scheduler**) arbitra l'accesso alle risorse.

- **Accodamento:** le richieste sono ricevute e "parcheggiate" in una coda di attesa.
- **Algoritmo:** una richiesta è scelta dalla coda di attesa.
- **Dispatching:** la richiesta scelta è assegnata alla risorsa.



Un semplice modello

(Schedulatore = coda attesa + algoritmo selezione + meccanismo dispatching)



Il ruolo delle astrazioni software

(È bene ribadire questo concetto)

Richieste, coda di attesa, algoritmo, dispatcher sono tutte **astrazioni software** (strutture dati di controllo + funzioni interne di gestione).

- Non esistono in realtà!
- Andate a controllare con il microscopio le periferiche hardware; non ne troverete nemmeno una, di astrazione!

Esempio specifico

(Che cosa sono entità, coda, algoritmo, dispatcher?)

Entità → astrazione software che modella una "richiesta" e/o un processo che la effettua.

Coda → astrazione software che modella una coda di attesa con una specifica disciplina di inserimento e di rimozione. (Ad oggi è implementata tramite un albero rosso-nero per motivi prestazionali)

Algoritmo → astrazione software che fornisce un servizio virtuale di "scelta" di una richiesta dentro una coda.

Dispatcher → astrazione software che fornisce un servizio virtuale di "assegnazione" della risorsa alla richiesta.

Code di attesa

(Wait queue)

Nel nucleo la gestione delle attese è effettuata tramite **code di attesa** (**wait queue**).

È una astrazione software.

- Struttura di controllo: lista con con disciplina di inserimento e cancellazione (tipicamente FIFO).
- Funzioni di gestione della lista: inserimento, controllo lista nulla, cancellazione, recupero elemento con più elevata priorità, …

Gli elementi della coda di attesa sono puntatori alle strutture di controllo delle astrazioni software.

Quante code di attesa esistono?

(Tante; una per ogni possibile ragione di attesa)

Esiste una coda di attesa per ogni possibile **evento** su una risorsa che, non ancora verificatosi, provoca il blocco di un processo.

- Attesa di disponibilità di una periferica di I/O.
- Attesa di un timeout.
- …

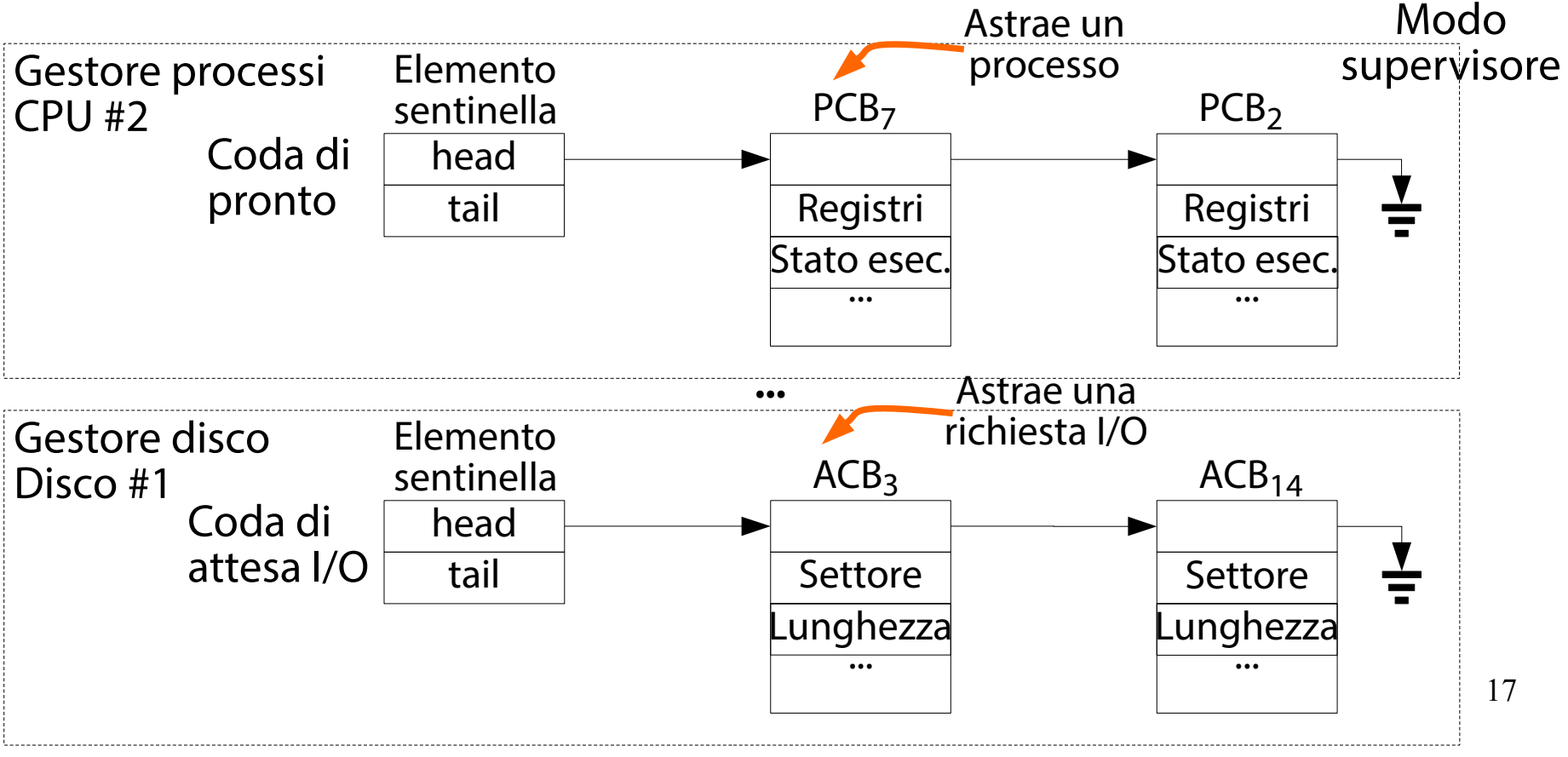
Ogni CPU ha anche una **coda di pronto** (**ready queue**) che contiene tutti i processi sbloccati dal verificarsi di un qualche evento e pronti per l'esecuzione.

Uno schema concettuale

(Non dettagliato; serve solo a far capire l'idea)

Modo utente

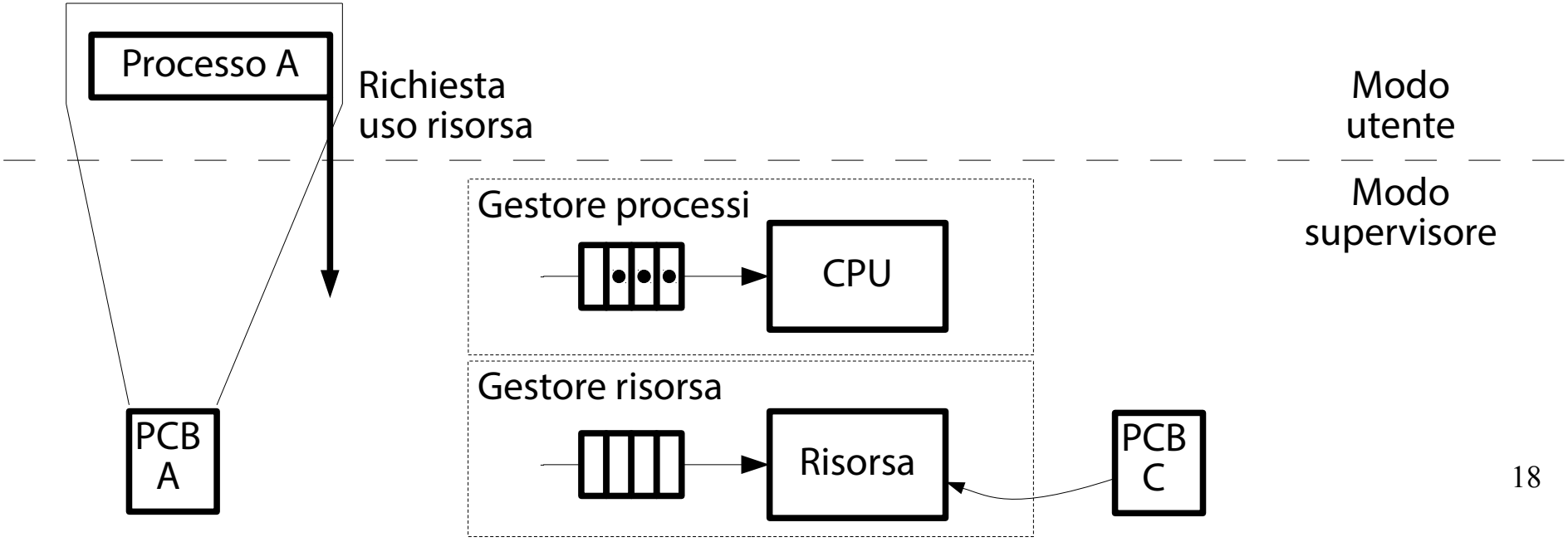
Modo supervisore



Accesso ad una risorsa occupata

(**Richiesta** → attesa → liberazione risorsa → risveglio → uso)

Il processo A vuole usare una risorsa gestita dal nucleo. Pertanto, effettua una richiesta al relativo gestore (tramite una chiamata di sistema).

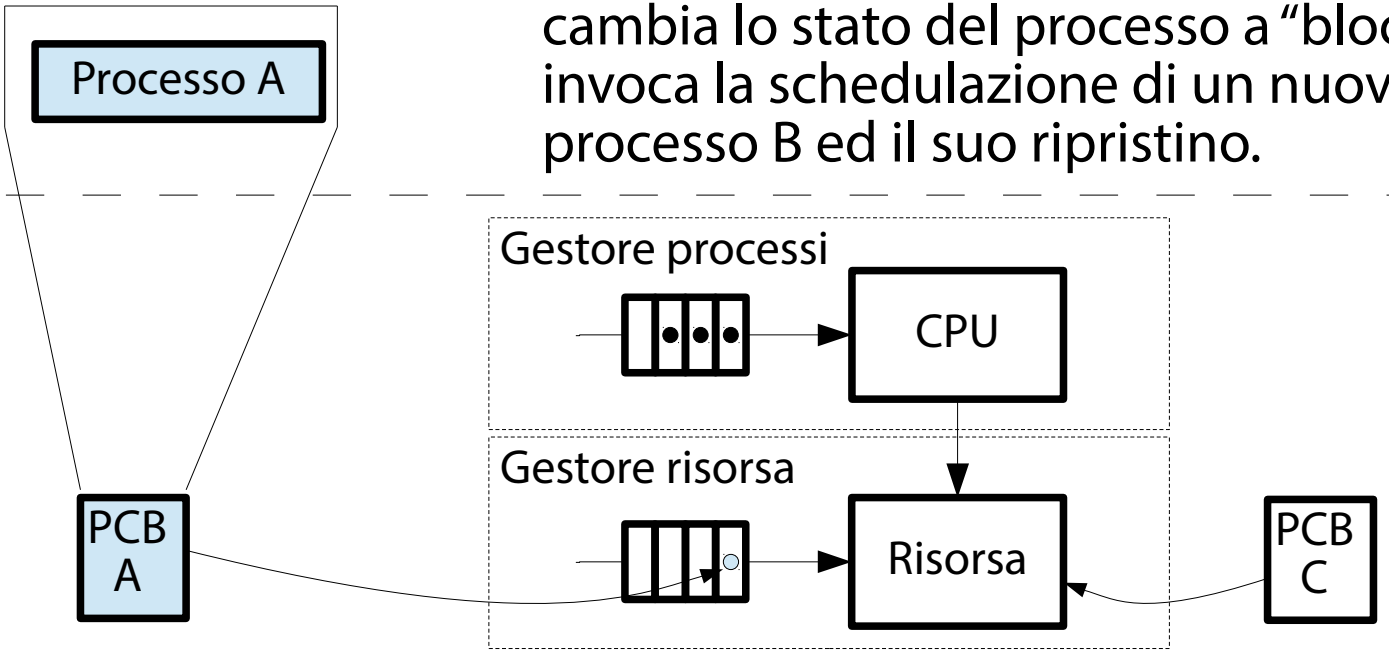


Accesso ad una risorsa occupata

(Richiesta → **attesa** → liberazione risorsa → risveglio → uso)

La risorsa è impegnata dal processo C.
Il gestore della risorsa:

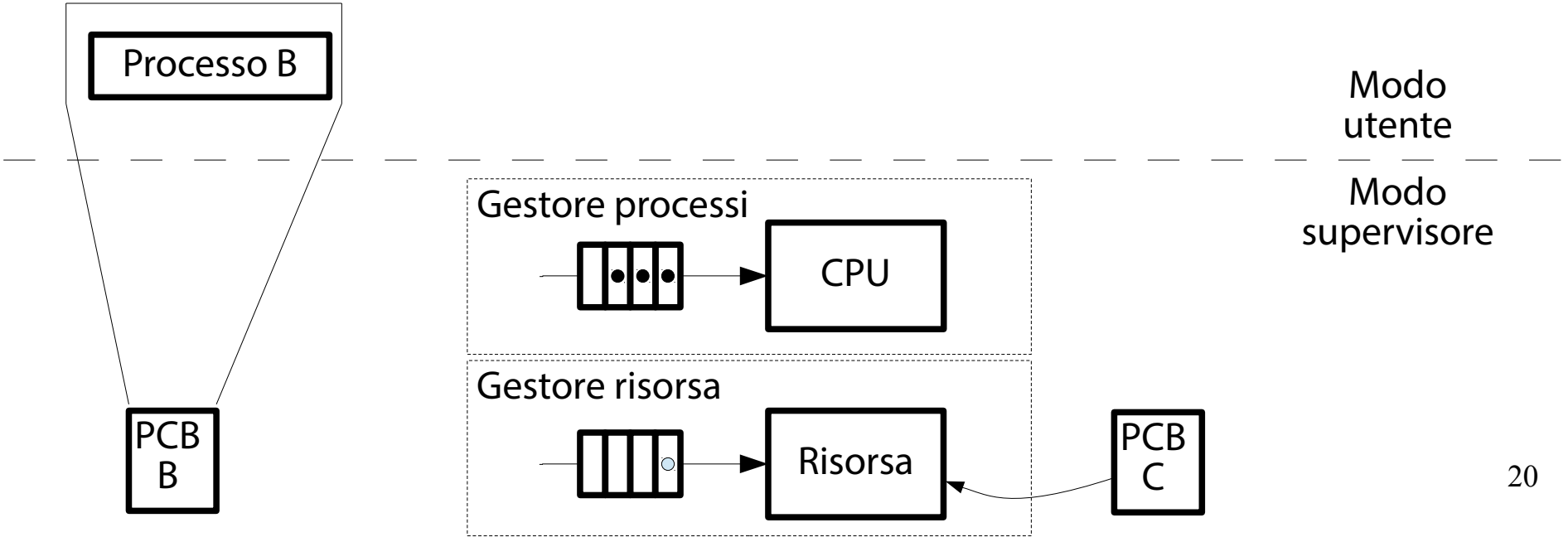
- blocca il processo perché non può avanzare.
- inserisce il PCB del processo A nella coda di attesa.
- cambia lo stato del processo a "bloccato".
- invoca la schedulazione di un nuovo processo B ed il suo ripristino.



Accesso ad una risorsa occupata

(Richiesta → **attesa** → liberazione risorsa → risveglio → uso)

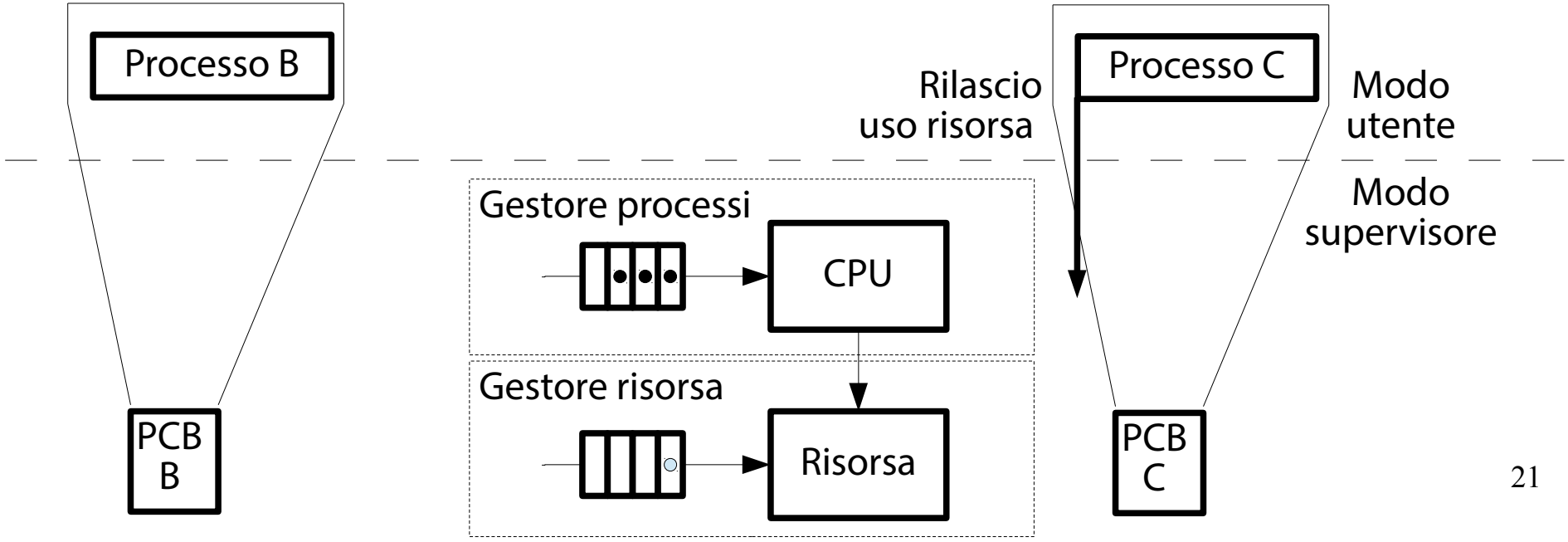
Ora è in esecuzione il processo B, che non usa la risorsa.
Il processo A non può più essere schedulato per l'esecuzione fintantoché non si libera la risorsa.
Pertanto, esso attende.



Accesso ad una risorsa occupata

(Richiesta → attesa → **liberazione risorsa** → risveglio → uso)

Successivamente un altro processo C (in esecuzione) finisce di usare la risorsa e la libera (ad esempio tramite una chiamata di sistema).



Accesso ad una risorsa occupata

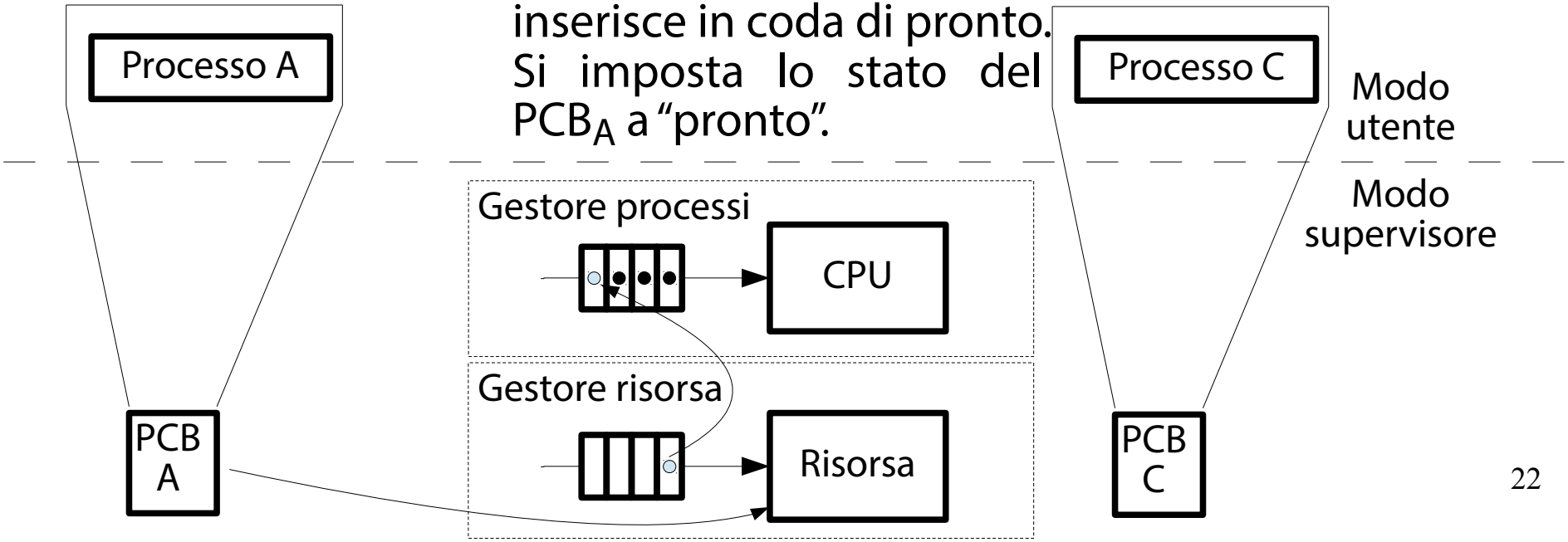
(Richiesta → attesa → liberazione risorsa → **risveglio** → uso)

Tale chiamata di sistema invoca il gestore della risorsa e provoca il **risveglio** di un processo in attesa (A).

Si disassocia il PCB di C dalla risorsa.

Si seleziona il primo PCB in coda di attesa e lo si inserisce in coda di pronto.

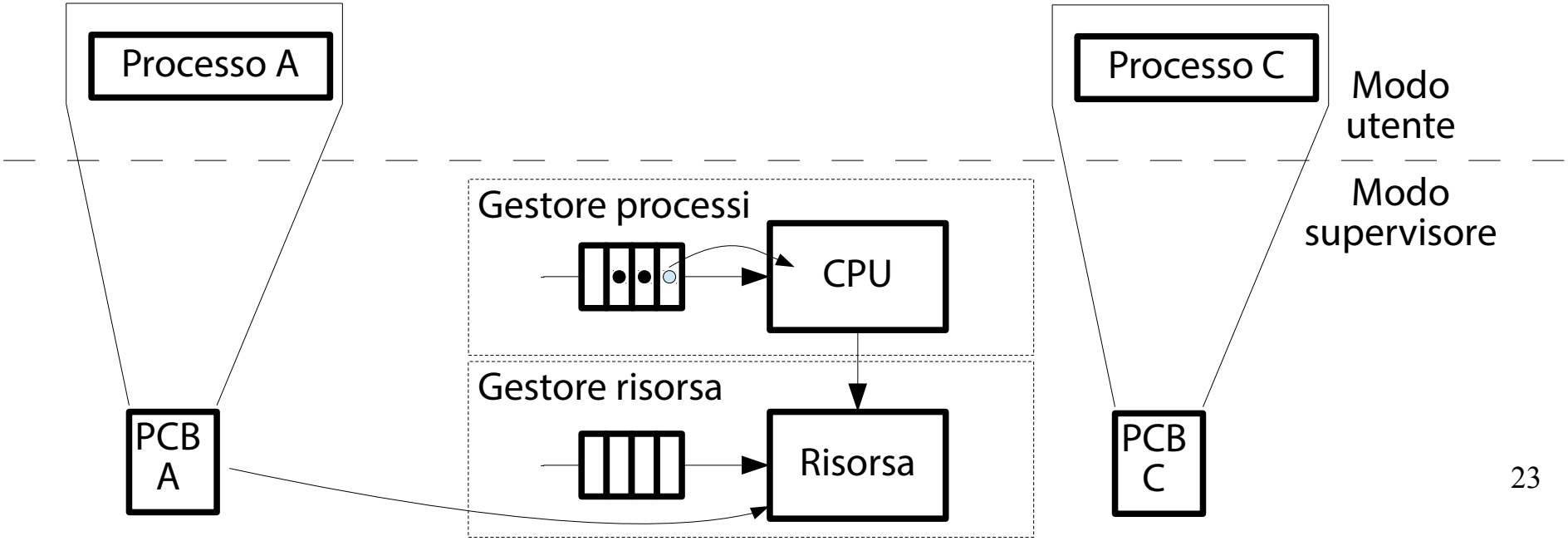
Si imposta lo stato del PCB_A a "pronto".



Accesso ad una risorsa occupata

(Richiesta → attesa → liberazione risorsa → risveglio → **uso**)

Dopo un po' il processo A sarà selezionato per l'esecuzione.
Il processo A acquisisce la risorsa.



Schedulatore della CPU

(Discusso in questa lezione)

Il nucleo implementa diversi schedulatori, uno per ciascuna risorsa offerta.

- CPU, disco, memoria, scheda di rete, …

In questa lezione si considera lo **schedulatore della CPU**.

- Entità richiedenti: processi, nucleo.
- Richieste: rappresentate dai PCB dei processi pronti.
- Risorsa assegnata: tempo di CPU.

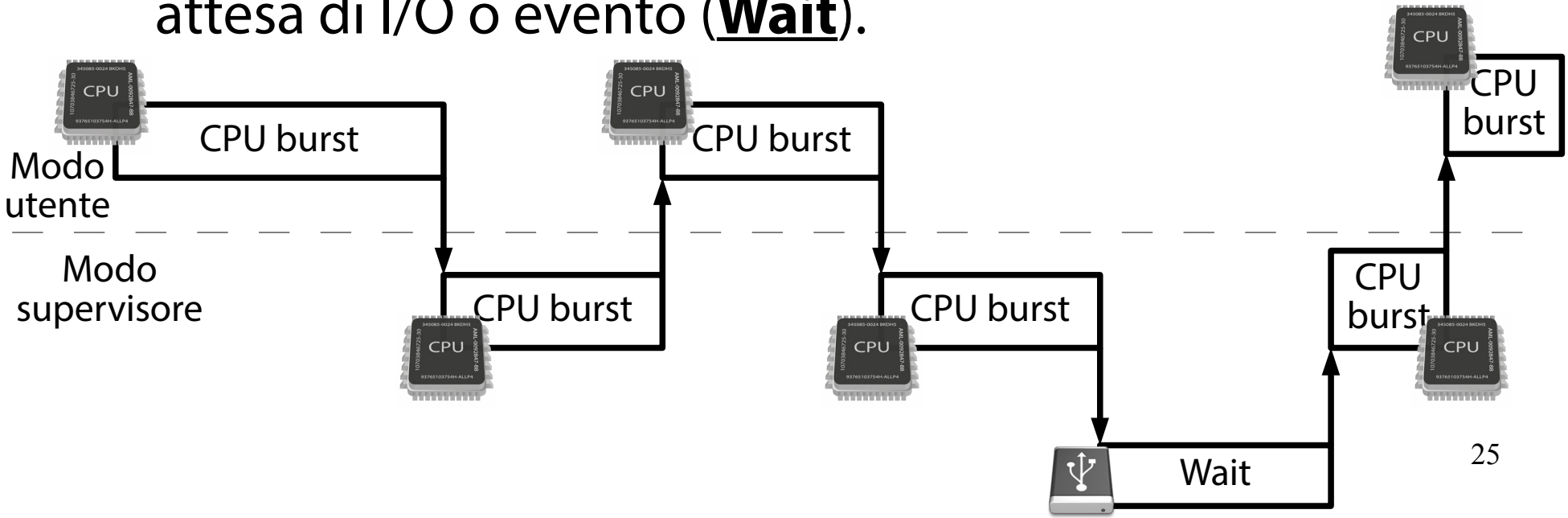
Che obiettivi concreti persegue lo schedulatore della CPU?

Modello di comportamento processi

(Una visione diversa rispetto all'alternanza Calcolo - Servizio)

Durante la sua esecuzione, un processo si alterna in due fasi:

- fase di elaborazione utente/supervisore (**CPU burst**).
- attesa di I/O o evento (**Wait**).



Processi vincolati alle risorse

(Resource boundedness)

Un processo si dice **vincolato** (**bound**) ad una risorsa (**resource**) oppure ad un evento (**event**) se la sua prestazione è correlata direttamente alla disponibilità della risorsa o al verificarsi dell'evento.

Se la risorsa scarseggia o l'evento è raro, il processo va a singhiozzo.

Più e disponibile/performante la risorsa, tanto più performante è il processo.

Processi CPU-bound e I/O-bound

(I più comuni)

Il comportamento di un processo si posiziona in genere fra questi due estremi.

Processo CPU-bound.

Tende a produrre poche sequenze di CPU burst lunghe.
Tende a fare poche richieste di I/O.

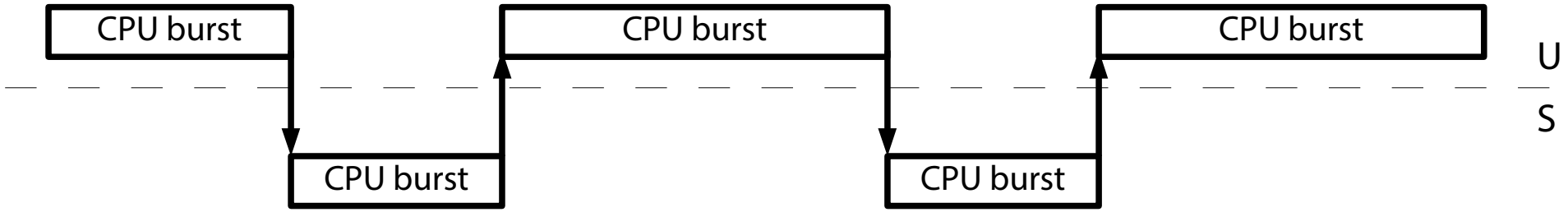
Processo I/O-bound.

Tende a produrre tante sequenze di CPU burst brevi.
Tende a fare molte richieste di I/O.

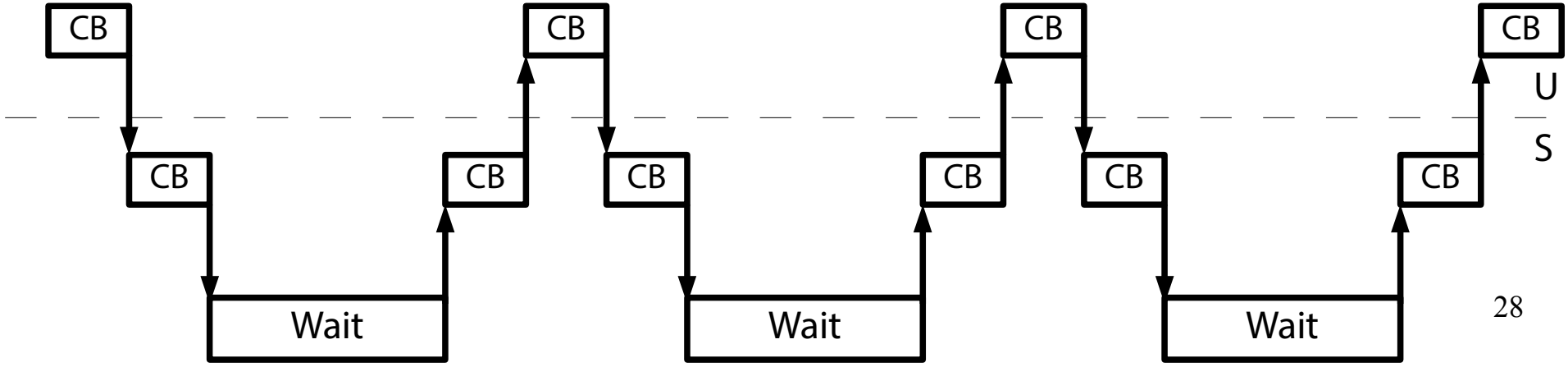
Processi CPU-bound e I/O-bound

(Diagrammi temporali di esecuzione tipici)

Processo CPU-bound `while true; do :; done`



Processo I/O-bound `dd if=/dev/sda of=/dev/sdb bs=1M`



Quali processi sono più importanti?

(Bella domanda; dipende dall'uso che si fa del SO)

Sistema desktop.

Elevata interattività con l'utente.

Frequenti richieste di I/O a periferiche.

→ Si dovrebbero favorire i processi I/O-bound.

Sistema di calcolo batch.

Interattività pressoché inesistente.

Necessità di completare quanti più processi di calcolo possibile.

→ Si preferiscono i processi CPU-bound.

Grado di multiprogrammazione

(Quanti processi attivi sono pronti per l'esecuzione?)

Il **grado di multiprogrammazione** è la somma di due contributi:

- il numero di processi in esecuzione.
- il numero di processi pronti per l'esecuzione.

Tale numero dipende da diversi fattori:

- la frequenza con cui un utente fa partire processi.
- la frequenza con cui i processi si bloccano.
- la frequenza con cui i processi terminano.

Obiettivi dello schedulatore della CPU

(Se non sono rigorosamente mantenuti, la macchina ha prestazioni basse)

Mantenere, nei limiti del possibile, il grado di multiprogrammazione imposto dall'utente senza degradare le prestazioni proprie e dei processi.

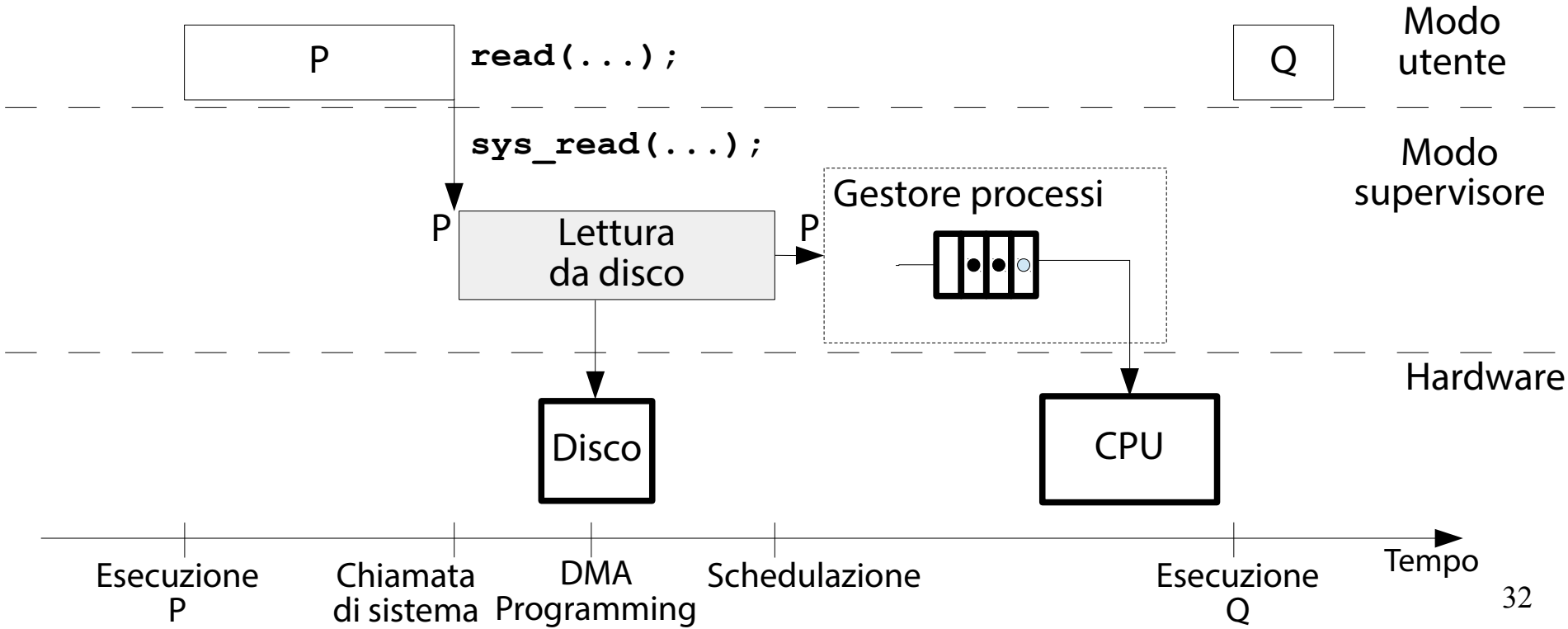
Favorire i processi "giusti" per il tipo di SO considerato:

- Desktop → I/O-bound.
- Server → CPU-bound.

Non lasciare mai inutilmente inoperosa la CPU.

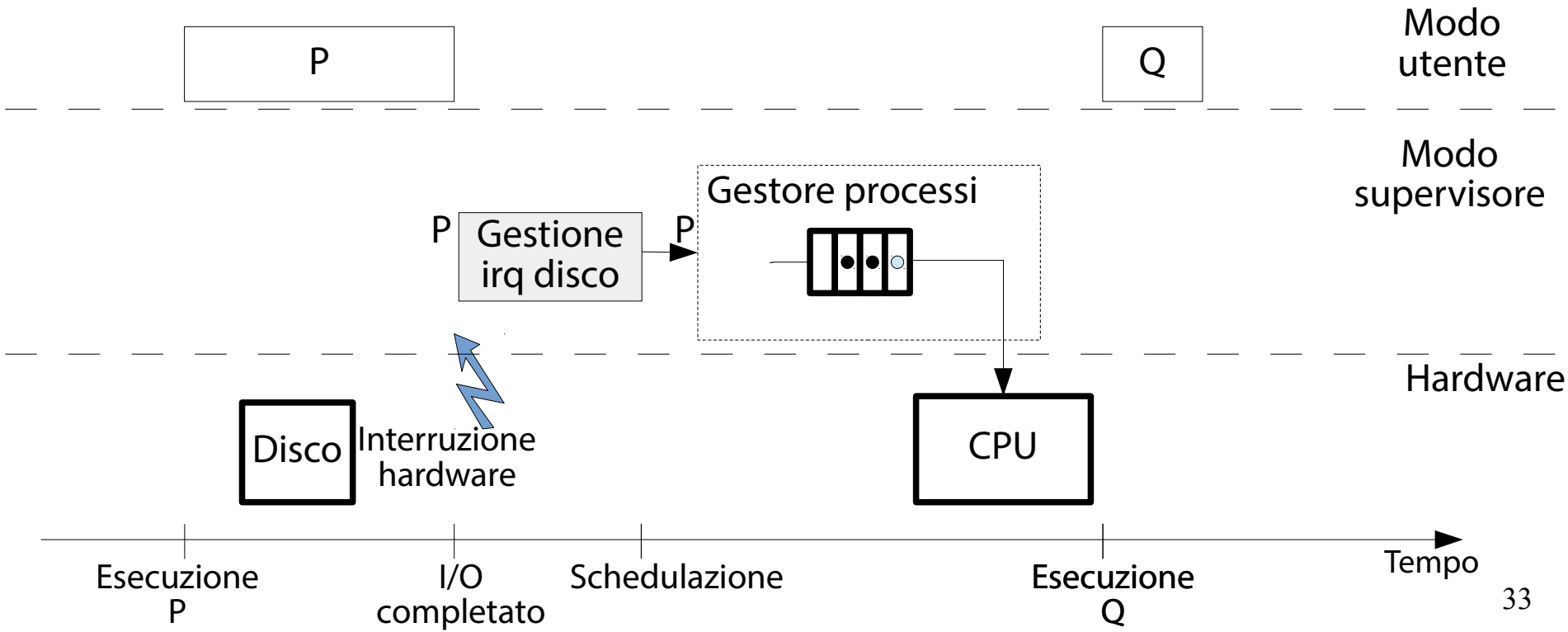
Quando schedula Linux?

(P→ Q: esecuzione di una richiesta bloccante)



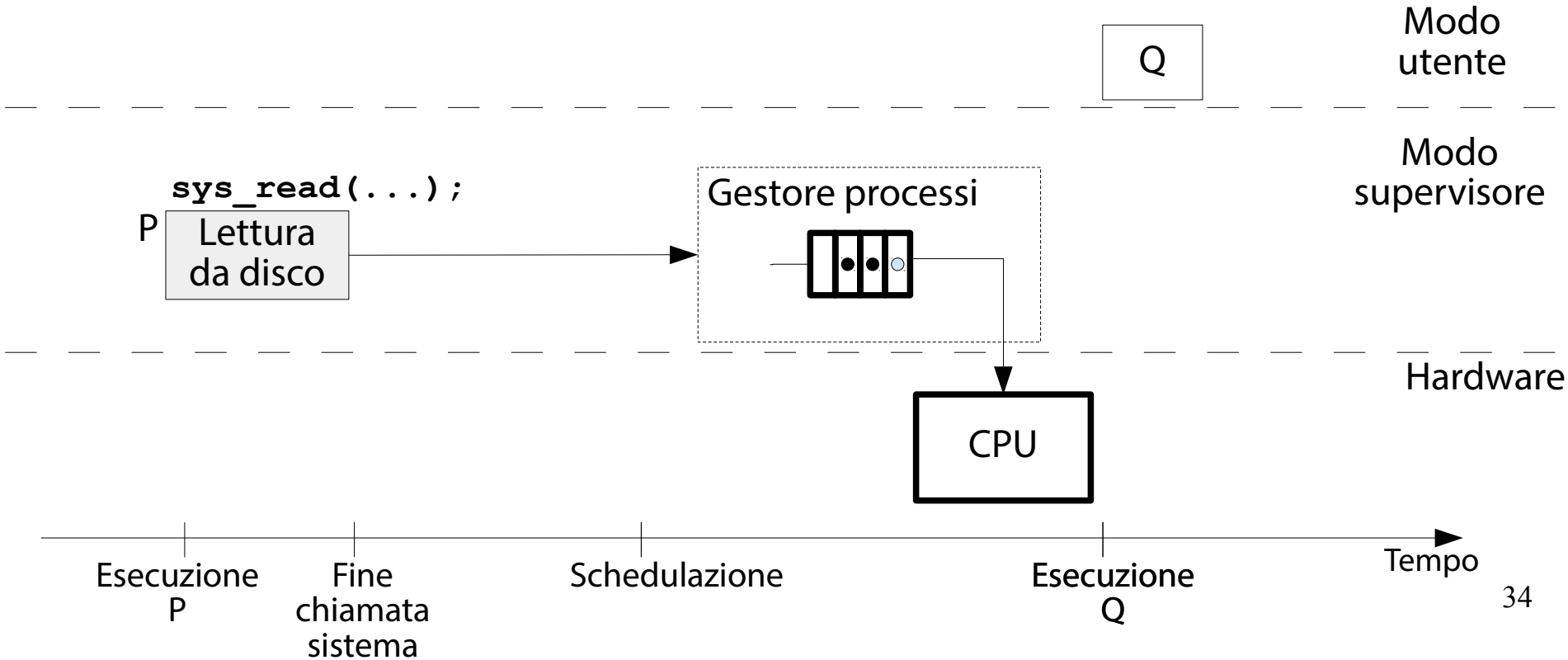
Quando schedula Linux?

(P→ Q: arrivo di una interruzione sulla CPU, non necessariamente dal disco)



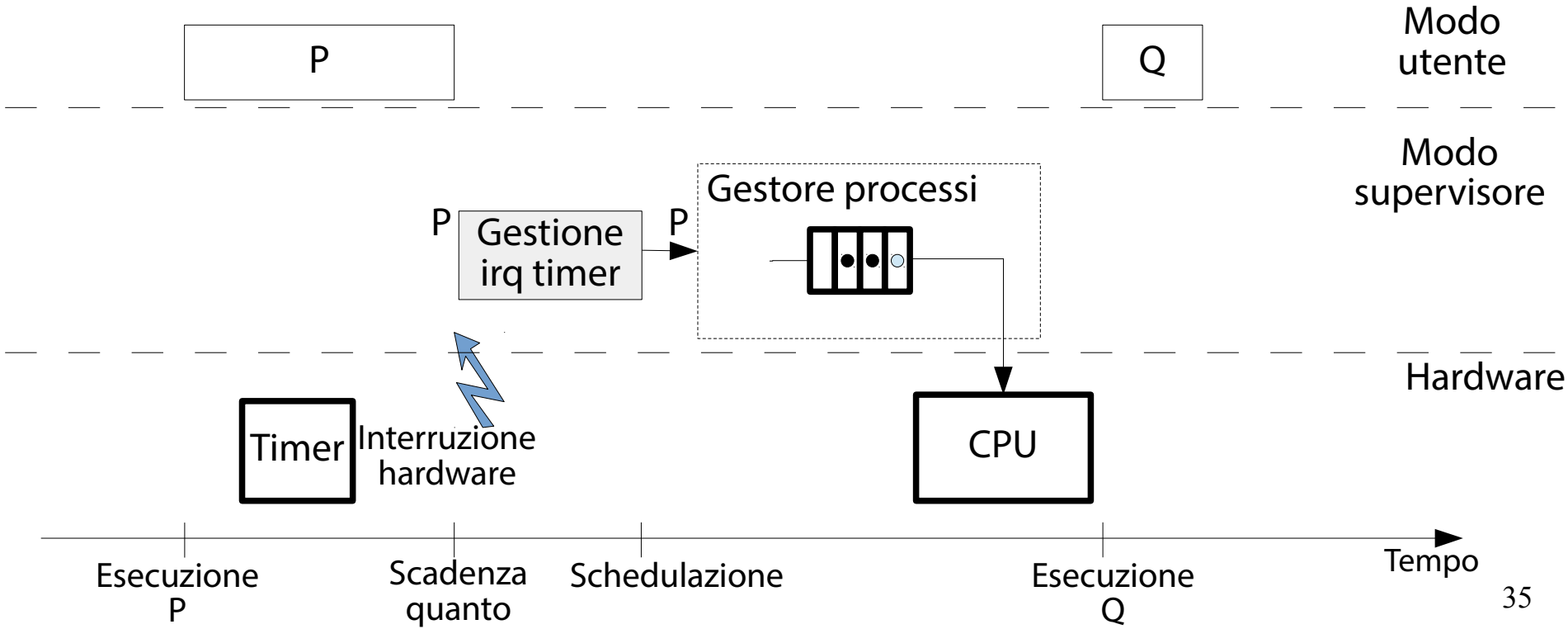
Quando schedula Linux?

(P→ Q: termine di una chiamata di sistema)



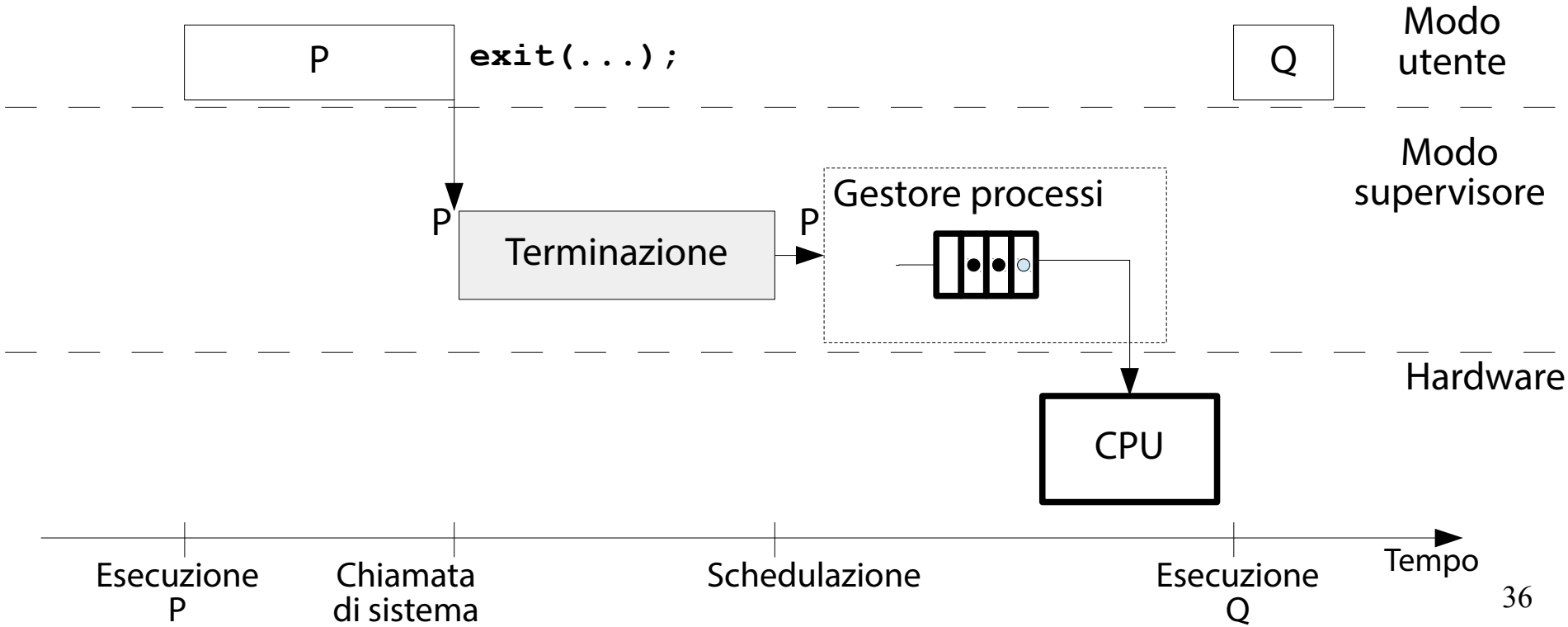
Quando schedula Linux?

(P→ Q: scadenza del quanto di tempo)



Quando schedula Linux?

(P→ Q: terminazione di un processo)



Una domanda (im)pertinente

(Di quelle che fanno guadagnare punti bonus)

Schedulare un nuovo processo Q quando:

- P esaurisce il suo quanto di tempo;
- P esegue una richiesta bloccante;
- P termina;

è naturale, ed è ciò che è stato sempre fatto dai sistemi time sharing.

Ma perché si dovrebbe schedulare un nuovo processo Q anche nelle seguenti situazioni?

- P termina una chiamata di sistema;
- arriva una interruzione hardware (non timer).

La risposta
(Semplice e diretta)

Il motivo è semplice: si aumentano le occasioni in cui il nucleo può schedulare un nuovo processo.

Pro:

aumenta la probabilità di schedulare un processo più importante che, nel frattempo, è diventato nuovamente pronto per l'esecuzione!
→ Migliora l'interattività del SO!

Contro:

il processo P viene "scippato" della CPU quando invece potrebbe ancora eseguire.

Prelazione

(Il nucleo sottrae la CPU ad un processo e la assegna ad un altro)

Si chiama **prelazione** (**preemption**) la capacità del nucleo di sottrarre inaspettatamente la CPU ad un processo (anche quando potrebbe andare avanti) per assegnarla ad un altro.

Per "inaspettatamente" si intende, tipicamente:

- all'arrivo di una interruzione hardware (non timer).
- al termine di una chiamata di sistema.

Lo schedulatore della CPU di Linux è con prelazione → Favorisce l'interattività.

Priorità di un processo

(Risolve il Problema 2/3)

Ad ogni processo è associato un numero intero non negativo detto **priorità**. La priorità (salvata nel descrittore di processo) definisce l'importanza del processo nel SO.

Tra due processi con priorità differenti, un algoritmo di schedulazione tende sempre a pescare quello con la priorità più alta.

“Quando un uomo con la pistola...”

(“...incontra un uomo col fucile, quello con la pistola è un uomo morto!”)

Priorità, starvation ed aging

(Altrimenti il SO stallerebbe in presenza di diversi processi CPU-bound)

Se uno scheduler decidesse sempre esclusivamente in base alla priorità, un processo a bassa priorità sarebbe spacciato.

- Non riceverebbe mai tempo di CPU (il suo "cibo").
- Morirebbe di fame (**starvation**).

Per evitare ciò, il gestore dei processi individua i processi in starvation ed alza loro gradualmente la priorità (**aging** o **priority boost**).

→ Prima o dopo (molto dopo) esegue anche il processo a priorità più bassa.

Priorità in Linux

(Sono ben 140!)

Linux usa 140 livelli distinti di priorità, mappati nell'intervallo [0, 139].

Occhio: le priorità sono invertite numericamente!

- 139 è la priorità più bassa.
- 0 è la priorità più alta.

Un processo "normale" inizia la sua esecuzione con la priorità 120.

Una domanda (im)pertinente

(Di quelle che fanno guadagnare punti bonus)

Linux schedula i processi tenendo conto di:

- priorità assegnata (la più alta vince).
- boost di priorità anti-starvation.

Come fa lo schedulatore della CPU a scegliere un processo nel caso in cui tutti i processi in coda di pronto abbiano la **stessa priorità**?

La risposta

(Molto semplificata, ma rende l'idea)

Nel caso in cui tutti i processi in coda di pronto abbiano la stessa priorità l'algoritmo di schedulazione sceglie il processo che ha ricevuto meno tempo di CPU.

Strumenti a disposizione degli utenti

(Risposta al Problema 3/3)

Il SO mette a disposizione strumenti per la gestione degli aspetti di schedulazione.

- Impostazione della CPU su cui esegue un processo.
- Impostazione della priorità di un processo.

Il SO mette a disposizione strumenti per il monitoraggio dell'attività di schedulazione.

- Numero di schedulazioni al secondo.
- Attesa di un processo.

ANALISI DELLE ATTESE

Scenario e domande

(Come sono gestite le attese? Quante code sono necessarie?)

Scenario: il SO time sharing sta eseguendo una moltitudine di processi su poche CPU.
Si consideri uno specifico processo.

Domande:
È possibile quantificare l'attesa del processo?
È possibile capire i motivi legati all'attesa?

Misurazione utilizzazione CPU

(Il processo tende ad attendere?)

Si monitorano le utilizzazioni di CPU del processo in modo utente e supervisore. Ad esempio, per il processo **`bash`** più recente:

```
pidstat -u -p $(pgrep -n bash) 1
```

Si leggono i campi **`%usr`** e **`%system`**.

Valori delle utilizzazioni bassi:

→ Il processo tende ad eseguire per breve tempo.

→ Il processo attende eventi.

Esercizi (3 min.)

1. Aprite un terminale ed eseguite il comando **`top`**. Monitorate le utilizzazioni di CPU in modo utente e supervisore per tale comando. Quali valori riscontrate? Cosa potete dire sul processo: attende o no?

Lettura del WCHAN

(Su quale funzione del nucleo è bloccato il processo?)

Una volta individuato un processo in attesa, è possibile individuare rapidamente la funzione interna del nucleo su cui è bloccato.

Si legge il campo **`WCHAN`** (**Wait Channel**) del descrittore dei processi. Ad esempio, per il processo **`bash`** più recente:

Stampa i primi 16 caratteri

```
ps -p $(pgrep -n bash) -o pid,wchan:16
```

Si scopre che **`bash`** attende sulla implementazione nel nucleo (**`do_wait()`**) della chiamata di sistema **`wait()`**.

Il padre **`bash`** attende che il figlio esca.

Un problema tecnico

(WCHAN non è sempre disponibile)

L'ISA Intel usa il registro **Frame Pointer** **`RBP`** per memorizzare il puntatore al record di attivazione corrente del processo attivo.

Affinché sia possibile identificare il **`WCHAN`**, è necessario che **`RBP`** sia usato come puntatore all'ultimo record di attivazione, e non come registro generico.

Purtroppo, Debian compila i suoi software con ottimizzazioni spinte, fra le quali l'uso di **`RBP`** come registro generico.

→ **`WCHAN`** non è disponibile in Debian su architettura INTEL x86 (32 bit) o x86_64 (64 bit).

Una soluzione di rimedio

(Il file /proc/*PID*/stack)

Per tale motivo il nucleo Linux annota l'insieme dei record di attivazione in modo supervisore e lo rende disponibile nel file **`/proc/PID/stack`**.
La lettura del file richiede i diritti di amministratore. Per il processo **`bash`** più recente:

```
su -
cat /proc/$(pgrep -n bash)/stack
```

Si scopre (nuovamente) che **`bash`** attende sulla implementazione nel nucleo (**`do_wait()`**) della chiamata di sistema **`wait()`**.

Esercizi (3 min.)

2. Con riferimento al comando **`top`** lanciato nell'Esercizio 1, individuate la funzione del nucleo su cui **`top`** è bloccato.

Un metodo alternativo con ltrace

(Su quale funzione del nucleo è bloccato il processo?)

In alternativa è possibile tracciare l'esecuzione del processo con il comando **`ltrace`**.

Occhio! Il comando **`ltrace`** è utile se l'evento bloccante deve ancora verificarsi! Se l'evento bloccante si è già verificato, non siete in grado di vederlo!

Ad esempio, per vedere il blocco del processo **`bash`** più recente:

```
ltrace -STtttn4 -p $(pgrep -n bash)
```

Esercizi (3 min.)

3. Con riferimento al comando **`top`** lanciato nell'Esercizio 1, eseguite una tracciatura del processo corrispondente con le opzioni usate nella slide precedente. Individuate la funzione di libreria/chiamata di sistema che provoca il blocco di **`top`**.

ANALISI DELLE SCHEDULAZIONI

Scenario e domande

(È possibile quantificare l'attività di schedulazione della CPU?)

Scenario: il SO time sharing sta eseguendo una moltitudine di processi su poche CPU.

Domande:

È possibile misurare l'attività di schedulazione dell'intero sistema?

È possibile misurare l'attività di schedulazione di un singolo processo?

Cambio di contesto
(Context switch)

Nel gergo dei SO, l'avvicendamento di un nuovo processo sulla CPU (risultato dell'operazione di dispatching) prende il nome di **cambio di contesto** (**context switch**).
Il numero di cambi di contesto al secondo fornisce una misura dell'attività di schedulazione di un sistema.

Il comando vmstat

(Fornisce molti indici di prestazione interessanti)

Il comando esterno **`vmstat`** (fornito dal pacchetto software Debian **`procps`**) fornisce informazioni sull'attività del sistema di gestione della memoria virtuale.

In cui sono coinvolti anche CPU e disco, come vedrete.

Il comando **`vmstat`** richiede solitamente una opzione (**`-w`**, per una migliore formattazione) ed un solo argomento (l'intervallo di campionamento):

```
vmstat -w 1
```

Attività globale di schedulazione

(Il comando vmstat 1)

Il campo **`cs`** del comando **`vmstat`** il numero di cambi di contesto effettuati in un secondo dalla CPU.

Più è alto **`cs`**, più intensa è l'attività di schedulazione.

- Esistono processi I/O-bound.
- Arrivano diverse interruzioni.
- Terminano diversi processi.

Esercizi (3 min.)

4. Aprite un terminale e lanciate il comando:

```
vmstat -w 1
```

Aprite un altro terminale e lanciate un processo CPU-bound:

```
while true; do :; done
```

Misurate i valori di **`cs`**. Come cambiano dopo l'esecuzione del processo CPU-bound?

Esercizi (3 min.)

5. Aprite un terminale e lanciate il comando:

```
vmstat -w 1
```

Aprite un altro terminale e lanciate un processo I/O-bound:

```
su -
dd if=/dev/sda of=/dev/null bs=4K
```

Misurate i valori di **`cs`**. Come cambiano dopo l'esecuzione del processo I/O-bound?

Confrontate i valori di cs ottenuti con quelli dell'esempio precedente. Che cosa osservate?

Attività di schedulazione per processo

(Il comando pidstat -w)

L'opzione **`-w`** del comando **`pidstat`** fornisce informazioni utili sull'attività di schedulazione.

```
pidstat -w 1
```

L'indice fornito è ancora una volta il numero di cambi di contesto effettuati al secondo.
Tuttavia, ne sono fornite due varianti, discusse nel seguito.

Cambi di contesto volontari e non

(Due misure molto utili)

Cambi volontari. Il processo esegue di sua sponte una operazione bloccante ed è costretto a bloccarsi.

Campo **`cswch/s`**.

Cambi non volontari. Il processo esegue per tutta la durata del quanto di tempo ed il nucleo lo blocca contro la sua volontà.

Campo **`nvcswch/s`**.

Il significato dei due indici

(Il processo tende ad essere CPU-bound o I/O-bound? Se I/O bound, su cosa?)

Considerate la coppia di valori:

(x, y) = (**`cswch/s`**, **`nvcswch/s`**).

Si hanno quattro casi distinti.

(x basso, y basso). Il processo fa pochi calcoli ed ha una scarsa attività di I/O.

(x alto, y basso). Il processo fa tante chiamate di tipo bloccante (I/O, allarmi, …) e pochi calcoli.

(x basso, y alto). Il processo tende ad usare tutto il quanto di tempo disponibile (CPU-bound).

(x alto, y alto). Il processo fa tante chiamate di I/O su periferiche basate su DMA. Esso viene viene interrotto dalle rischedulazioni seguenti le interruzioni hw.

Esercizi (5 min.)

6. Ripetete gli Esercizi 4 e 5, usando il comando **`pidstat -w 1`** come strumento per il monitoraggio dell'attività di schedulazione.

Che cosa osservate?

GESTIONE DELLE PRIORITÀ

Scenario e domande

(Come sono gestite le attese? Quante code sono necessarie?)

Scenario: il SO time sharing sta eseguendo una moltitudine di processi su poche CPU.
Si consideri uno specifico processo.

Domande:

È possibile modificare la priorità di esecuzione del processo?
È possibile far eseguire un processo su una specifica CPU?
È possibile quantificare tali aspetti?

Priorità statiche in GNU/Linux

(Intervallo [100, 139])

I processi avviati normalmente hanno una priorità nell'intervallo [100, 139]. Le priorità in [100, 139] si chiamano **priorità statiche**.

Non è possibile assegnare le priorità in [0, 99] con mezzi normali. Servono le maniere forti…

- Comandi speciali dati da amministratore.
- Altrimenti, chiunque innalzerebbe la priorità dei propri processi.

Lettura della priorità statica

(Possibile con diversi comandi: top, ps)

Diversi comandi permettono di leggere il valore di priorità di un processo normale:

```
top
ps
```

Sfortunatamente, ciascun comando sembra avere una propria rappresentazione delle priorità.

Usare l'intervallo [0, 139] era troppo facile…

Lettura della priorità statica

(Possibile con diversi comandi: **top**, ps)

Il comando **top** mostra la priorità del processo nel campo di nome **PR**.

I valori di priorità statica mostrati da **top** variano nell'intervallo [0, 39].

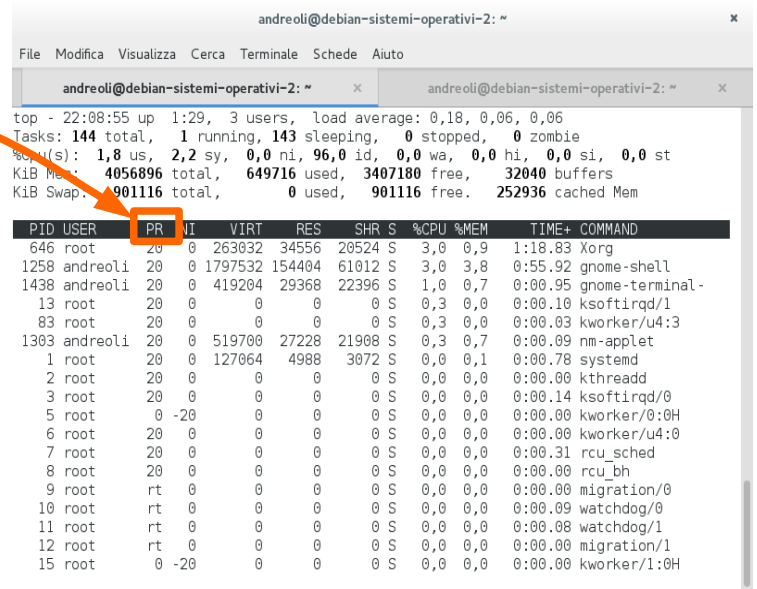
Priorità mostrata da **top**

= Priorità del nucleo

- 100

La priorità più comune tra i processi è quella di default.

120 (nucleo), 20 (**top**)



Lettura della priorità statica

(Possibile con diversi comandi: top, **ps**)

Il comando **`ps -l`** mostra la priorità del processo nel campo di nome **`PRI`**.

I valori di priorità mostrati da **`ps`** variano nell'intervallo [-40, 99].

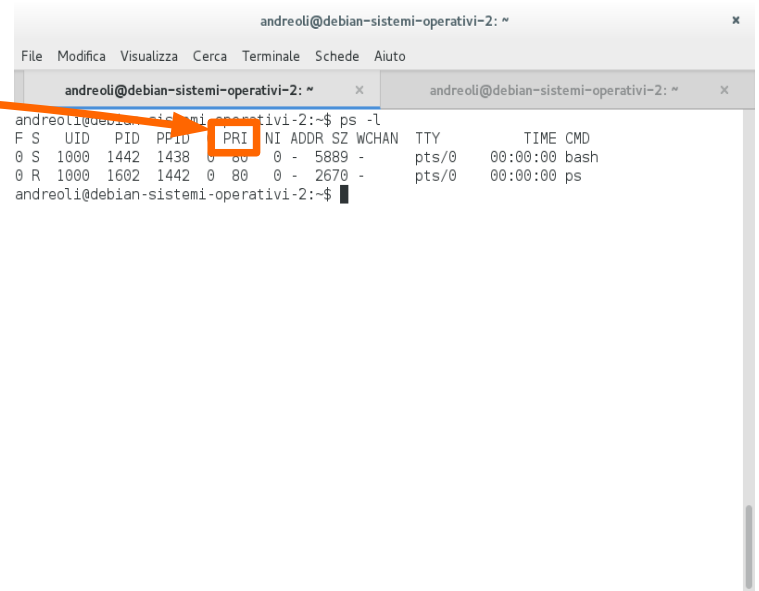
Priorità mostrata da **`ps`**

= Priorità del nucleo

- 40

La priorità più comune tra i processi è quella di default.

120 (nucleo), 80 (**`ps`**)



Una piccola osservazione

(Non ve la chiederò mai all'esame; ve la propongo così, giusto per curiosità)

Il comando **ps** è in grado di mostrare le priorità in ben sette tipologie di intervallo diverse!

Provare per credere:

```
ps -o pid,priority,opri,pri_foo,pri_bar,pri_baz,pri,pri_api,comm
```

L'intervallo usato di default da **ps** corrisponde al campo **opri**.

```
andreoli@debian-sistemi-operativi-2:~$ ps -o pid,priority,opri,pri_foo,pri_bar,pri_baz,
pri,pri_api,comm
  PID PRI  PRI FOO BAR BAZ PRI API COMMAND
 1442  20   80   0  21 120  19 -21 bash
 1657  20   80   0  21 120  19 -21 ps
```

Vi rivelo un segreto…

(Tutti i migliori (programmi) sono matti!)

Esercizi (3 min.)

7. Aprite un terminale e lanciate il comando:

```
sleep 1000
```

Misurate la priorità del processo associato al comando, usando sia **`top`** sia **`ps`**.
Riportate i valori ottenuti.

Il comando nice

(Permette di modificare la priorità statica di default)

Il comando esterno **`nice`** permette di lanciare un programma con una priorità modificata.

Tanto per cambiare, **`nice`** usa la propria rappresentazione della priorità statiche: l'intervallo [-20, 19] dei **valori di cortesia** (**nice value**, **niceness**).

Valore di cortesia

= Priorità del nucleo

\- 120

Perché abbassare la priorità?

(Per poter vedere i film mentre ricompilate il kernel! Che domande…)

Immaginate di dover eseguire una applicazione con le caratteristiche seguenti:

- tempo di esecuzione elevato.
- nessun requisito di interattività.
- nessun requisito sul tempo di completamento.

Classici esempi:

- backup di un file system.
- compilazione di un software grosso.

Voi non vorreste che tale applicazione sottraesse troppe risorse al SO mentre ~~guardate un film videogiocate chattate su FB~~ lavorate, giusto?

Stampa del valore di cortesia di default

(nice, senza opzioni, senza argomenti)

Il comando **`nice`**, lanciato senza opzioni e senza argomenti, stampa il valore di cortesia di default. Tale valore dovrebbe in teoria essere pari a 0 (si veda la relazione introdotta nella slide precedente).

Avviate un terminale e scrivete:

```
nice
```

Il valore di cortesia di default è, in effetti, 0.

Avvio di comandi a priorità più bassa

(nice *comando*, nice -n *nicelevel comando*)

Per lanciare un comando (ad esempio, **`ls`**) ad una priorità più bassa, basta lanciare il comando con un livello di cortesia più alto di 0 (10, se non lo si specifica):

```
nice ls
```

È possibile specificare un valore di cortesia con l'opzione **`-n`**. Ad esempio, per lanciare **`ls`** con la priorità 139 (valore di cortesia 19):

```
nice -n +19 ls
```

Lettura del valore di cortesia

(Possibile con diversi comandi: top, ps)

Diversi comandi permettono di leggere il valore di cortesia di un processo normale:

```
top
ps
```

A differenza del caso precedente, qui la rappresentazione dei valori di cortesia è una sola.

Lettura del valore di cortesia

(Possibile con diversi comandi: **top**, ps)

Il comando **top** mostra il valore di cortesia del processo nel campo di nome **NI**.

I valori di cortesia mostrati da **top** variano nell'intervallo [-20, 19].

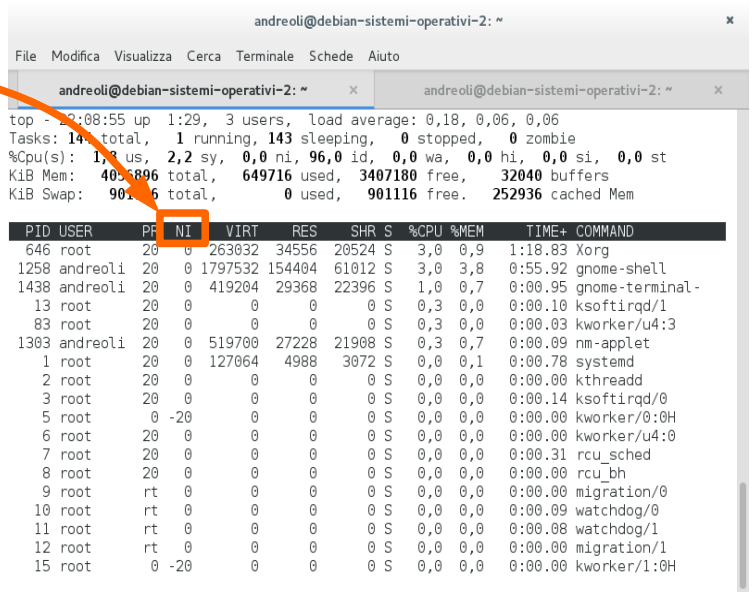
Valore di cortesia

= Priorità del nucleo

\- 120

Il valore di cortesia più comune tra i processi è quello di default.

120 (nucleo), 0 (**top**)



Lettura del valore di cortesia

(Possibile con diversi comandi: top, **ps**)

Il comando **`ps -l`** mostra il valore di cortesia del processo nel campo di nome **`NI`**.

I valori di cortesia mostrati da **`ps`** variano nell'intervallo [-20, 19].

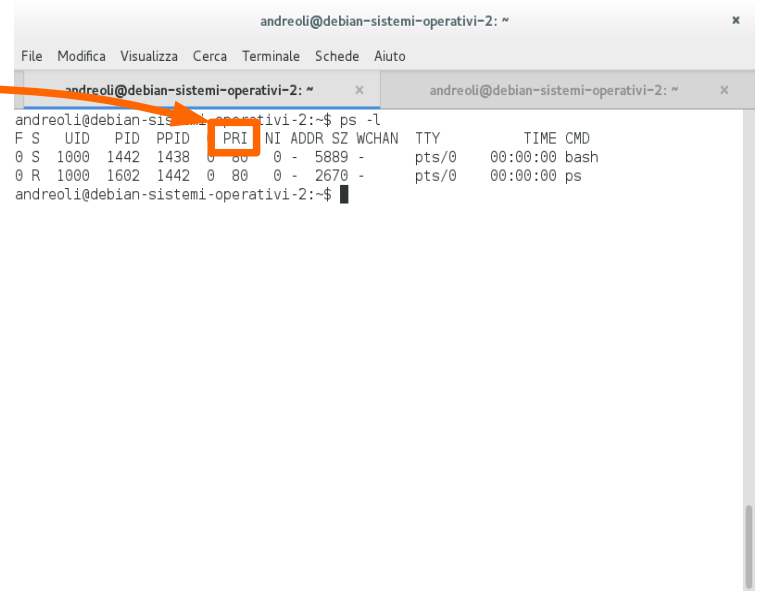
Valore di cortesia

= Priorità del nucleo

\- 120

Il valore di cortesia più comune tra i processi è quello di default.

120 (nucleo), 0 (**`ps`**)



Esercizi (2 min.)

8. Aprite un terminale e lanciate il comando:

```
nice -n +19 ls -lR /
```

Misurate il valore di cortesia del processo associato al comando **`ls`**.

Riportate i valori ottenuti.

L'effetto della priorità più bassa

(Un rallentamento consistente nell'esecuzione del processo)

Aprite un terminale e lanciate 10 processi di natura CPU bound al suo interno.

```
for i in $(seq 1 10); do
   bash -c 'while true; do :; done' &
done
```

Nello stesso terminale lanciate prima:

```
ls -lR /
```

e dopo:

```
nice -n +19 ls -lR /
```

Notate delle differenze nell'esecuzione?

Una domanda (im)pertinente

(Di quelle che fanno guadagnare punti bonus)

È fortissima la tentazione di eseguire il comando con un livello di cortesia più basso.
Una cosa del genere, per capirsi:

```
nice -n -20 ls -lR /
```

Sarà possibile eseguire **`ls`** con priorità più alta (ovvero, con un livello di cortesia più basso)?

La risposta

(Dovrebbe essere sufficientemente chiara)

NO!*

*O meglio, l'utente può provare a lanciare il comando con priorità più alta, ma ben difficilmente riuscirà a raggiungere lo scopo prefisso.

Qual è il problema?

(Non si può alzare la priorità di un processo se non si è amministratori)

Il comando **`nice`** notifica l'impossibilità di alzare la priorità con il seguente messaggio:

```
nice: cannot set niceness: Permission denied
```

Tuttavia, il comando viene eseguito lo stesso (con la priorità di default).

Servono i diritti di amministratore per poter elevare la priorità di un processo.

PS: si scopre ora perché il comando si chiama **`nice`**…

Be nice to others!

(By lowering your process priority)

Esercizi (2 min.)

9. Eseguite **`ls -lR /`** alla priorità più alta di cui siete capaci finora.

Misurate i valori di priorità e di cortesia del processo associato al comando **`ls`**.

Riportate i valori ottenuti.

Modifica della priorità statica

(Il comando renice)

Il comando esterno **`renice`** modifica la priorità statica di un processo in esecuzione.
Per il resto, valgono le considerazioni svolte in precedenza sul comando **`nice`**. Se, ad esempio, un processo eseguente

```
ls -lR /
```

ha PID 1234, il seguente comando modifica la sua priorità statica a 139:

```
renice -n +19 -p 1234
```

Esercizi (2 min.)

10. Aprite un terminale ed eseguite il comando seguente:

```
top
```

Misurate il valore di priorità del processo associato al comando.

Modificate la priorità statica del processo al valore 139, senza riavviarlo nuovamente.

Misurate nuovamente il valore di priorità del processo associato al comando.

Classi di schedulazione

(Una classe → Un insieme di processi condividenti un algoritmo di scheduling)

Finora si è considerato il solo algoritmo di schedulazione "di default" che, nel prendere le sue decisioni, considera due fattori:

- il valore di priorità (statica);
- il tempo di esecuzione sulla CPU (ovvero, la sua mancanza).

Tuttavia in Linux non esiste solo questo algoritmo.

In generale, i processi sono suddivisi in **classi di scheduling** disgiunte.

- Una classe → un insieme di processi che condividono lo stesso algoritmo di schedulazione.

Perché diverse classi?

(Per accomodare i requisiti sui tempi di completamento)

Perché esistono diverse classi di schedulazione?
Perché processi diversi possono avere requisiti diversi sui tempi di completamento.

Un backup del vostro disco rigido può terminare anche fra 4 ore; non muore nessuno.

Se il nucleo lancia un servizio di swap della memoria, tale servizio deve sempre eseguire subito e terminare nel più breve tempo possibile (pena il non funzionamento della macchina).

Classi ed algoritmi
(SCHED_NORMAL: processi normali)

SCHED NORMAL: processi normali gestiti con l'algoritmo di schedulazione di default (**Completely Fair Scheduler**, **CFS**).

Si usano le priorità in [100, 139].

Viene schedulato per primo chi ha la priorità più alta.

In caso di priorità uguale, viene schedulato chi ha usufruito della CPU per meno tempo.

C'è prelazione; il nucleo può strappare il processo alla CPU per favorirne un altro.

Classi ed algoritmi
(SCHED_FIFO: FIFO senza prelazione)

SCHED FIFO: processi “real time” gestiti con l'algoritmo **First in First Out** (**FIFO**).

Si usano le priorità in [0, 99] → Precedenza rispetto ai processi normali.

Viene schedulato per primo chi ha la priorità più alta.

In caso di priorità uguale, l'ordine di schedulazione è quello di attivazione.

NON C'È PRELAZIONE. Il nucleo esegue il processo fino a quando non esce o non fa I/O!

Ottimo per applicazioni batch ad alta priorità.

Una osservazione

(Si può rallentare la macchina, ma non stallarla completamente)

Occhio! Potete rallentare pesantemente il SO se lanciate tanti processi

- CPU-bound
- di classe SCHED_FIFO
- ad alta priorità

quanti sono i processori disponibili.

Non riuscirete tuttavia ad impallare il SO del tutto.

Il SO ha dei meccanismi di protezione contro questo problema.

Per saperne di più su questi meccanismi, leggete il link di approfondimento.

Classi ed algoritmi

(SCHED_RR: Round Robin con prelazione)

SCHED_RR: processi "real time" gestiti con l'algoritmo **Round Robin** (**RR**).

Si usano le priorità in [0, 99] → Precedenza rispetto ai processi normali.

Viene schedulato per primo chi ha la priorità più alta.

In caso di priorità uguale, l'ordine di schedulazione è circolare: 1, 2, 3, 4, …, N, 1, 2, 3, 4, …, N, …

C'è prelazione; il nucleo può strappare il processo alla CPU per favorirne un altro.

Ottimo per applicazioni interattive ad alta priorità.

Classi ed algoritmi

(SCHED_BATCH: CFS su processi a bassissima priorità)

SCHED_BATCH: processi poco importanti gestiti con l'algoritmo di default (CFS).

Viene schedulato per primo chi ha la priorità più alta.

In caso di priorità uguale, viene schedulato chi ha usufruito della CPU per meno tempo.

C'è prelazione; il nucleo può strappare il processo alla CPU per favorirne un altro.

Ottimo per applicazioni non interattive a bassa priorità.

Una domanda (im)pertinente

(Di quelle che fanno guadagnare punti bonus)

Riassumendo: non esiste un solo schedulatore, bensì diversi, ciascuno associato ad una specifica classe di processi che lo condivide.

Come funziona allora, in generale, lo scheduler dei processi?

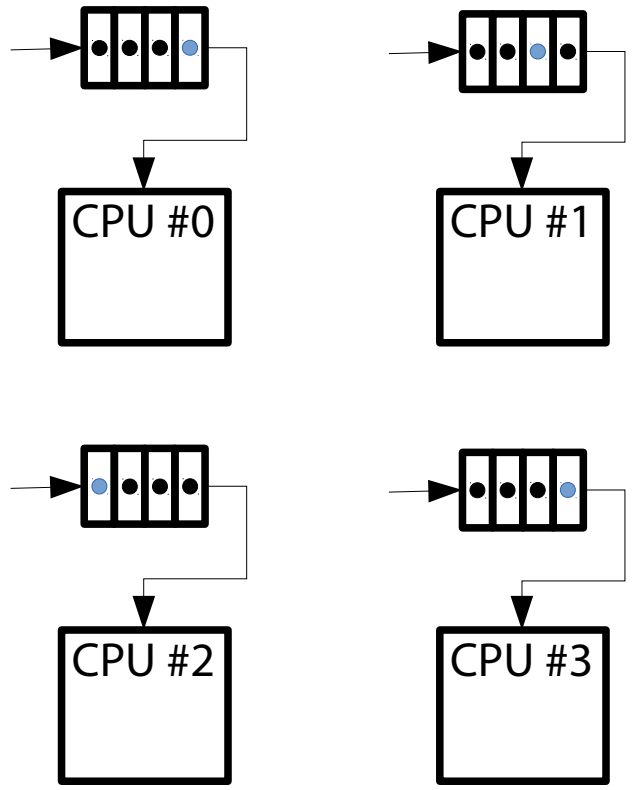
- In che ordine sono invocati gli schedulatori di processo delle classi distinte?
- Se due processi SCHED_FIFO e SCHED_RR sono alla stessa priorità, chi viene schedulato per primo?

La struttura dello schedulatore di Linux

(A closer look)

Lo schedulatore dei processi è distribuito sulle CPU disponibili.

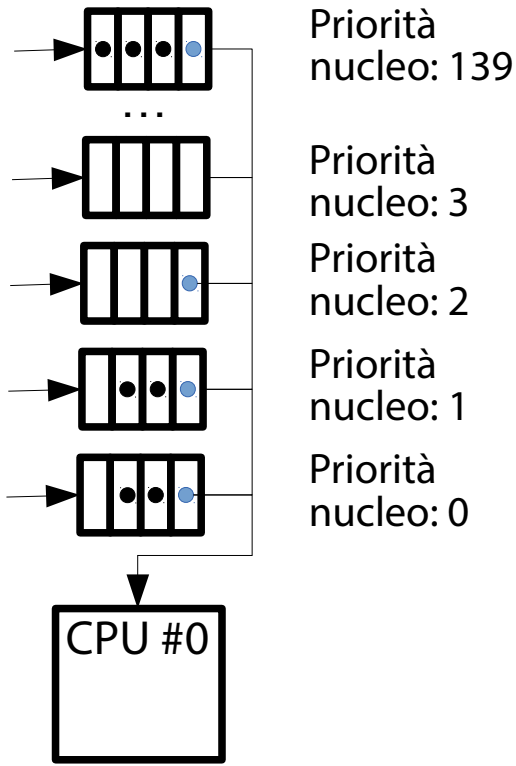
- Ogni CPU ha una coda dei processi pronti.
- Il codice dello schedulatore esegue concorrentemente sulle CPU.



Le code dei processi pronti
(Una per ogni priorità del nucleo)

La coda dei processi pronti è, in realtà, un insieme di 140 code di pronto.

- Una coda per ogni priorità del nucleo.



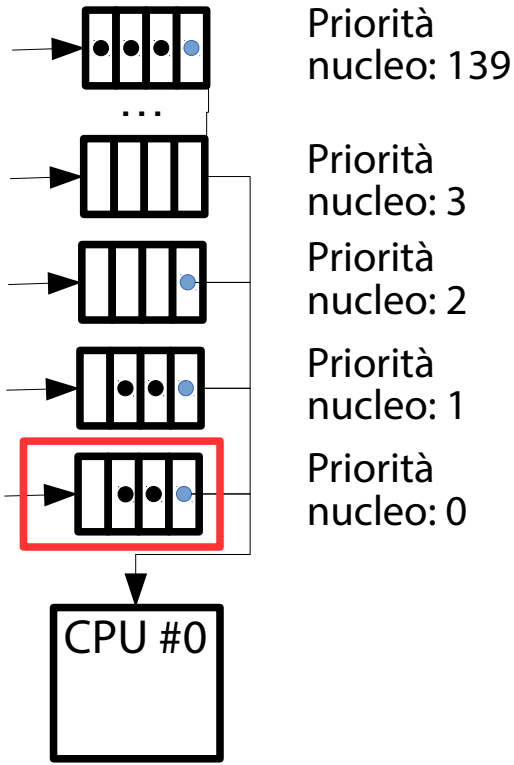
Scelta della coda di processi

(La prima non vuota)

L'algoritmo di schedulazione sceglie innanzitutto una coda da cui pescare processi.

- Parte dalla coda di pronto a priorità di nucleo 0.
- Sale fino alla prima coda di pronto con almeno un processo in attesa di esecuzione.

La prima coda popolata è la 0.

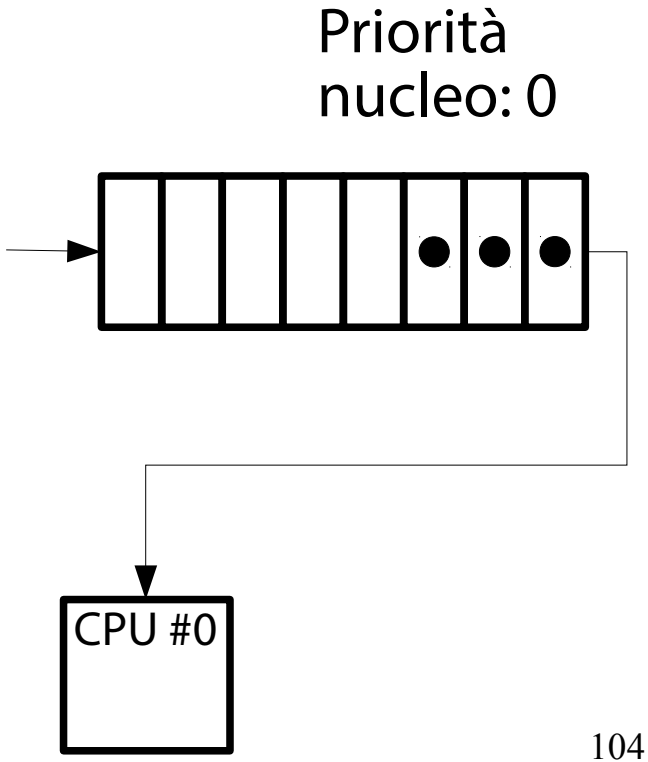


Scelta della classe di schedulazione

(**Prima SCHED_FIFO**, poi SCHED_RR)

All'interno della coda è presente almeno un processo. Nell'ordine, lo schedulatore vede se esistono processi di classe:

- SCHED_FIFO
- SCHED_RR



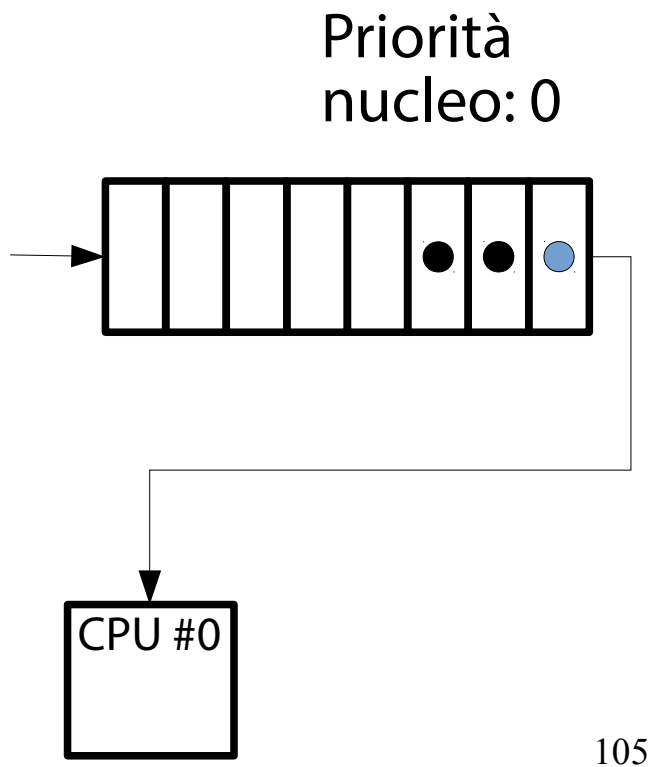
Schedulazione

(FIFO schedula il primo processo di classe SCHED_FIFO)

Se ne trova uno SCHED_FIFO, schedula tutti i processi nella classe SCHED_FIFO.

Schedula il primo usando l'algoritmo FIFO.

Non c'è prelazione. Il processo va avanti fino all'I/O, alla fine o all'arrivo di un processo con una priorità più elevata (non è questo il caso).



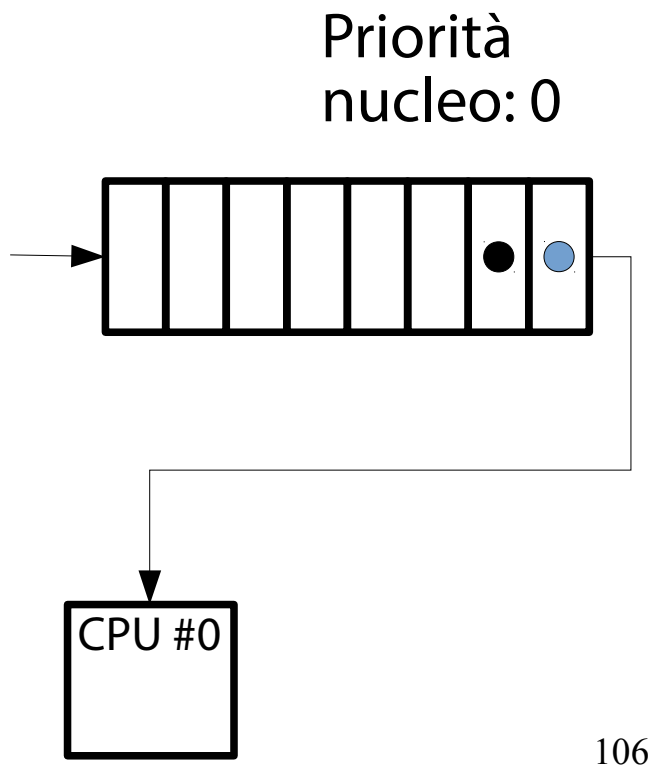
Schedulazione

(FIFO schedula il secondo processo di classe SCHED_FIFO)

Se ne trova uno SCHED_FIFO, schedula tutti i processi nella classe SCHED_FIFO.

Poi schedula il secondo con la classe FIFO.

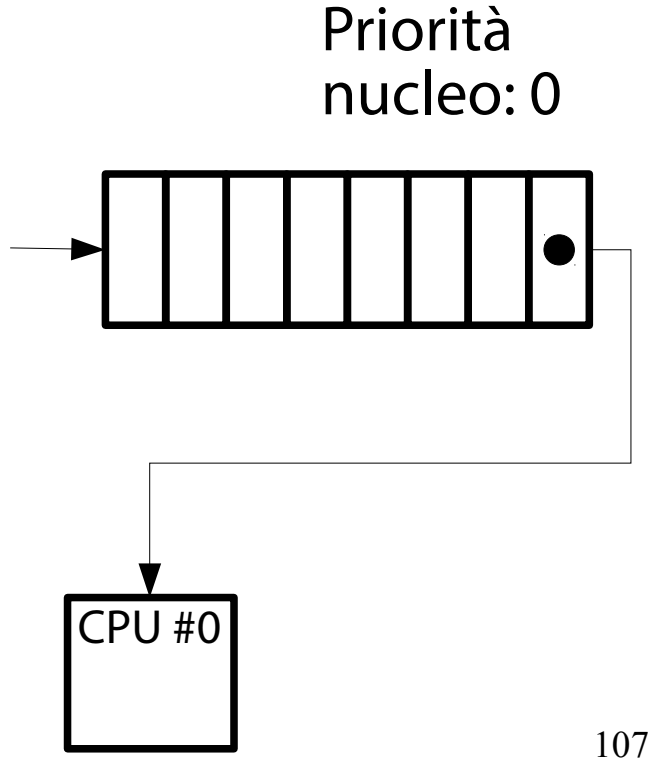
Non c'è prelazione. Il processo va avanti fino all'I/O, alla fine o all'arrivo di un processo con una priorità più elevata (non è questo il caso).



Scelta della classe di schedulazione

(Prima SCHED_FIFO, **poi SCHED_RR**)

Una volta schedulati tutti i processi all'interno della classe SCHED_FIFO, si procede con la classe SCHED_RR.
Esiste un processo con in tale classe?
Sì, esiste.

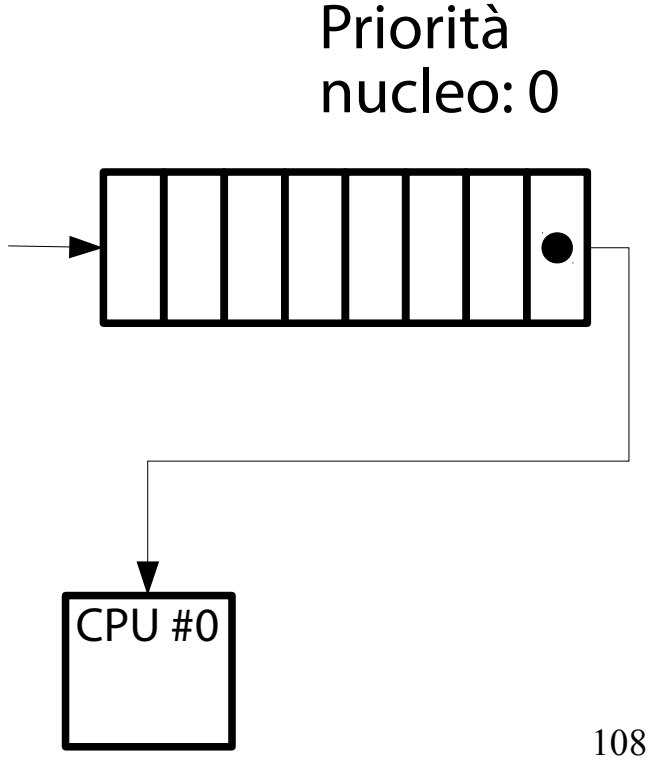


Scelta della classe di schedulazione

(Prima SCHED_FIFO, **poi SCHED_RR**)

Il processo all'interno della classe SCHED_RR è soggetto all'algoritmo Round Robin.

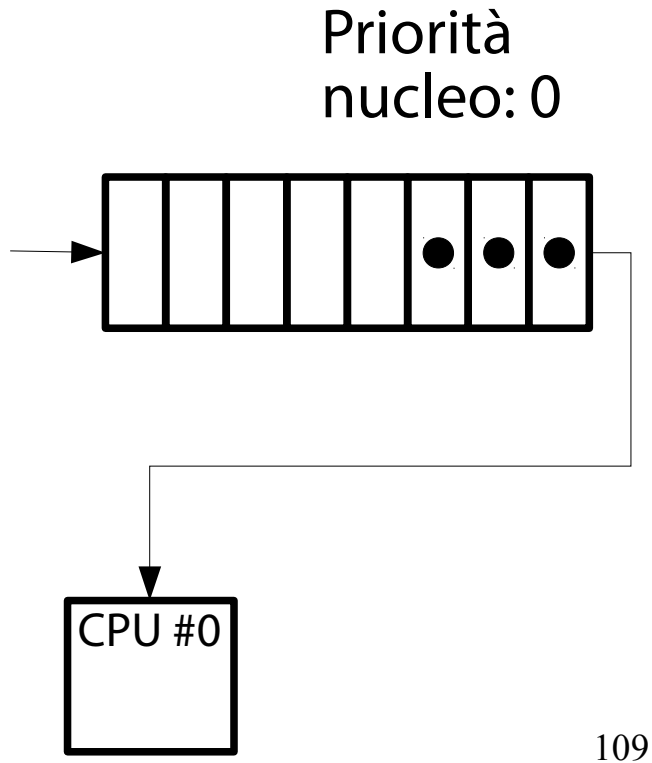
C'è prelazione. Il processo può essere interrotto non solo da un processo a priorità più elevata (non è questo il caso), bensì anche al termine di un quanto di tempo.



Arrivo di altri processi in priorità

(La causa primaria di starvation dei processi normali)

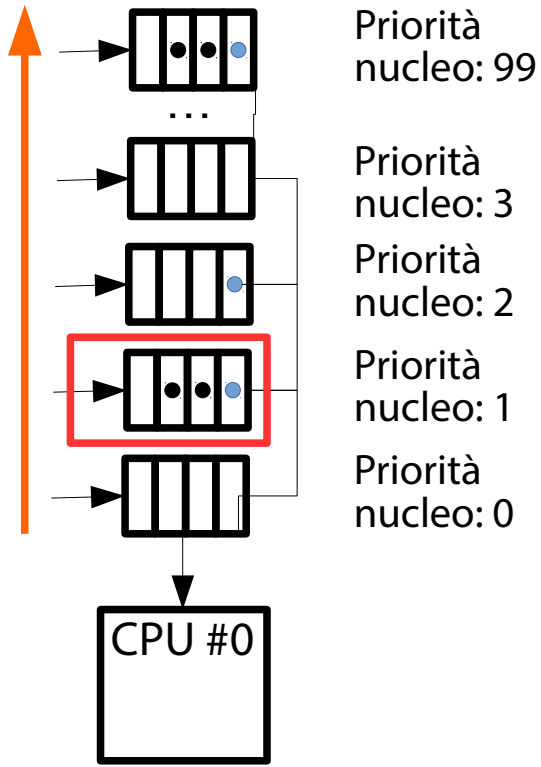
Se nel frattempo si accodano altri processi, non si passa alle priorità successive.
E ricordate: FIFO può sottrarre la CPU a RR.



Smaltimento delle code a priorità più alta

(Sempre dalla prima non vuota)

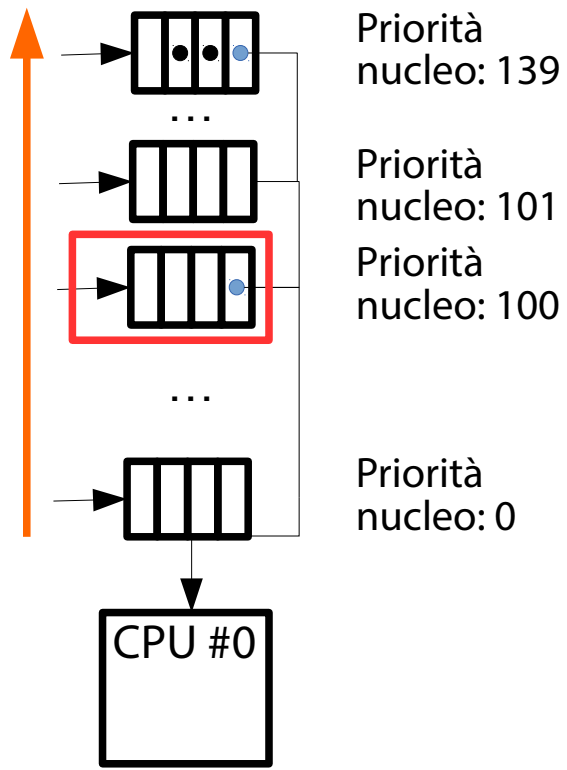
Una volta svuotata la coda a priorità 0, si passa alla prima coda non vuota (la 1, nel caso in questione).
Si prosegue allo stesso modo fino alla coda di priorità 99 (l'ultima delle code riservate ai processi real time).



Smaltimento delle code "normali"

(Sempre dalla prima non vuota)

Successivamente si passa alle code di priorità nell'intervallo [100, 139].
Anche qui, si mira a svuotare progressivamente le code a partire dalle priorità più alte fino a quelle più basse.
Si parte dalla prima coda popolata (la 100).



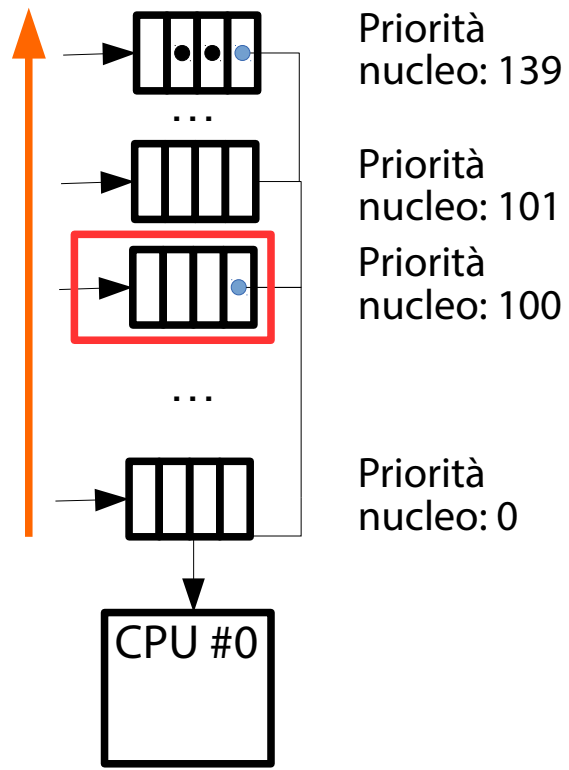
Smaltimento delle code "normali"

(Sempre dalla prima non vuota)

All'interno della coda è presente almeno un processo. Nell'ordine, lo schedulatore vede se esistono processi di classe:

- SCHED_NORMAL
- SCHED_BATCH

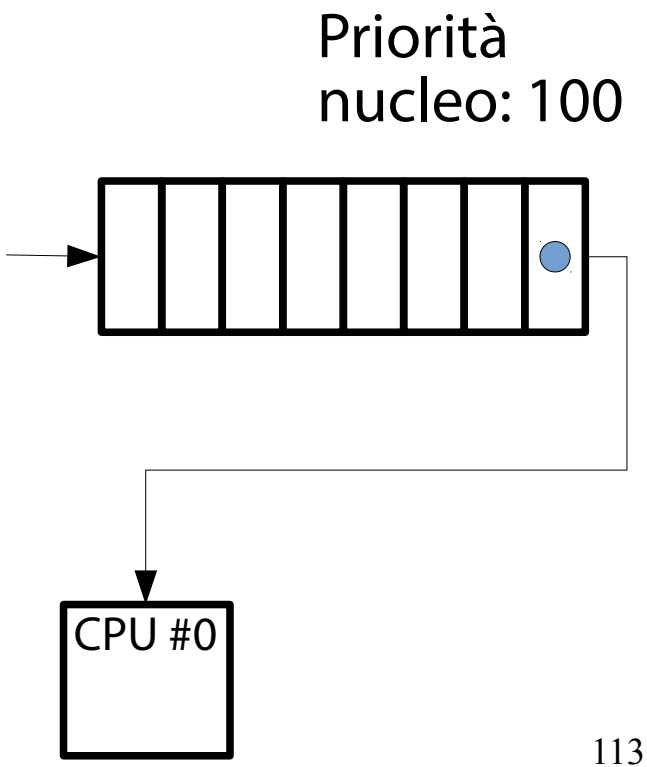
Occhio! Esistono molte più classi, che sono omesse nella spiegazione per semplicità.



Scelta della classe di schedulazione

(**Prima SCHED_NORMAL**, poi SCHED_BATCH)

Se ne trova uno di classe SCHED_NORMAL, schedula tutti i processi nella classe SCHED_NORMAL.

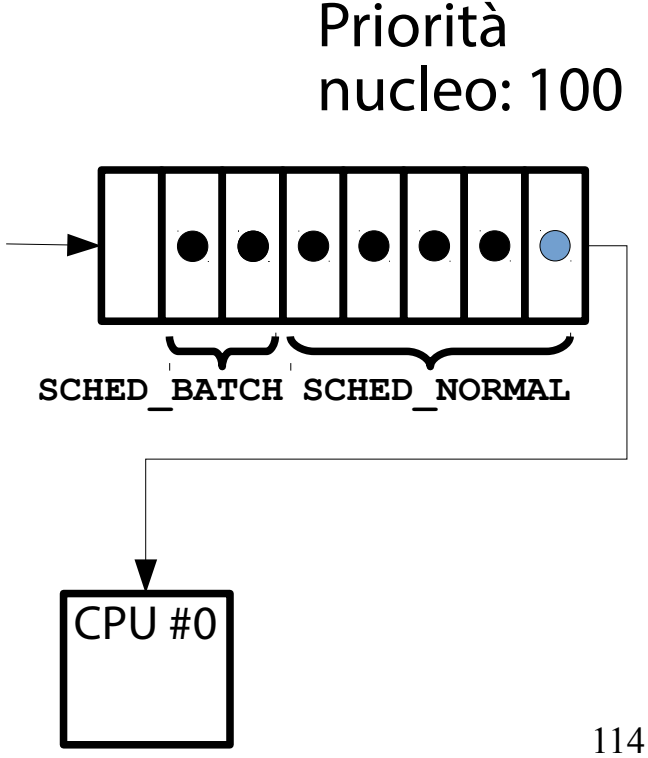


Scelta della classe di schedulazione

(Prima SCHED_NORMAL, **poi SCHED_BATCH**)

Solo quando tutti i processi di classe SCHED_NORMAL sono stati eseguiti può eseguire un processo appartenente alla classe SCHED_BATCH.

- Sempre che nel frattempo non arrivino processi a priorità più alta…



Gestione di classi di scheduling e priorità

(Il comando schedtool)

Il comando esterno **`schedtool`** gestisce le classi di priorità dei processi.

- Recupero informazioni di schedulazione.
- Impostazione di classe e priorità.

Il comando **`schedtool`** è fornito dal pacchetto software **`schedtool`**.

Diventate amministratore.

```
su -
```

Installate il pacchetto software **`schedtool`**:

```
apt-get install schedtool
```

Stampa informazioni di schedulazione

(schedtool, lanciato senza opzioni)

Se lanciato

senza opzioni;

con tanti argomenti numerici rappresentanti PID;

`schedtool` stampa le informazioni di schedulazione di tutti i processi associati a tali PID.

Ad esempio, per stampare le informazioni di schedulazione dei processi di PID 1 e 10 e 1234:

```
schedtool 1 10 1234
```

Rappresentazione della priorità
(Prima fregatura)

L'intervallo di priorità usato da **`schedtool`** è [0, 99].
"0" è la priorità più bassa (e non la più alta, come l'intervallo delle priorità del nucleo [0, 139] lascerebbe pensare).
Analogamente, "99" è la priorità più alta.

Rappresentazione della priorità
(Seconda fregatura)

Il valore di priorità ha senso solo per le classi di schedulazione real time, ovvero:

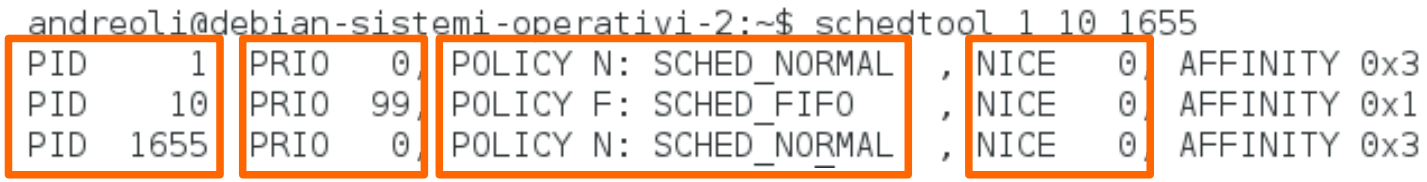
- SCHED_FIFO
- SCHED_RR

In tutti gli altri casi, il valore di priorità è posto a 0 e non va considerato.

Stampa informazioni di schedulazione

(schedtool, lanciato senza opzioni)

Un possibile output è il seguente:



PID del processo

Priorità (`schedtool`)

Algoritmo di schedulazione

Livello di cortesia

Stampa informazioni di schedulazione

(Comando pidstat, opzioni -R e -p)

L'opzione **`-R`** del comando **`pidstat`** fornisce informazioni sulle priorità di uno specifico processo, selezionabile con l'opzione **`-p`**.

`PID`: PID del processo

`prio`: priorità statica/real time

`policy:` classe di scheduling

Ad esempio, per stampare le informazioni di schedulazione del processo di PID 1234 ogni secondo:

```
pidstat -R -p 1234 1
```

Esercizi (5 min.)

11. Stampate le informazioni di schedulazione di tutti i processi in esecuzione.

Che cosa osservate?

Impostazione delle priorità real time

(schedtool, lanciato con le opzioni -R/-F e -p)

È possibile impostare una qualsiasi delle classi di schedulazione tramite una specifica opzione.
Ad esempio:

- **`-F`**: algoritmo real time FIFO.
- **`-R`**: algoritmo real time Round Robin.

Queste classi richiedono l'impostazione di una priorità real time nell'intervallo [0, 99].

- **`-p` *`prio`***: imposta la priorità real time *prio* per gli algoritmi Round Robin o FIFO.

Selezione del processo

(Argomento PID → processo esistente; -e "comando" → un nuovo processo)

Il processo può essere uno fra quelli già esistenti, selezionabile specificando il suo PID come argomento. Ad esempio, per il PID 1234:

```
schedtool -R -p 40 1234
```

In alternativa è possibile aggiungere l'opzione **-e** e specificare un comando come argomento. Tale comando eseguirà con la priorità scelta.

```
schedtool -R -p 40 -e top
```

Esercizi (3 min.)

12. Lanciate due istanze di **`top`**.

In seguito modificate le priorità dei due processi nel modo seguente:

un top → SCHED_NORMAL, nice 10
l'altro top → SCHED RR, prio 50

Misurate le priorità durante l'esecuzione.

AFFINITÀ

Scenario e domande

(Come eseguire processi su CPU specifiche?)

Scenario: il SO time sharing sta eseguendo una moltitudine di processi su poche CPU.

Domande:

È possibile assegnare un processo ad una CPU specifica (ad esempio, per farlo eseguire indisturbato)?

È possibile vedere su quali CPU stanno eseguendo i processi?

Affinità

(Se il processo esegue sempre sulla stessa CPU, è CPU cache friendly)

L'**affinità di CPU** (**CPU affinity**) è la tendenza di un processo a rimanere in esecuzione sempre sulla stessa CPU.

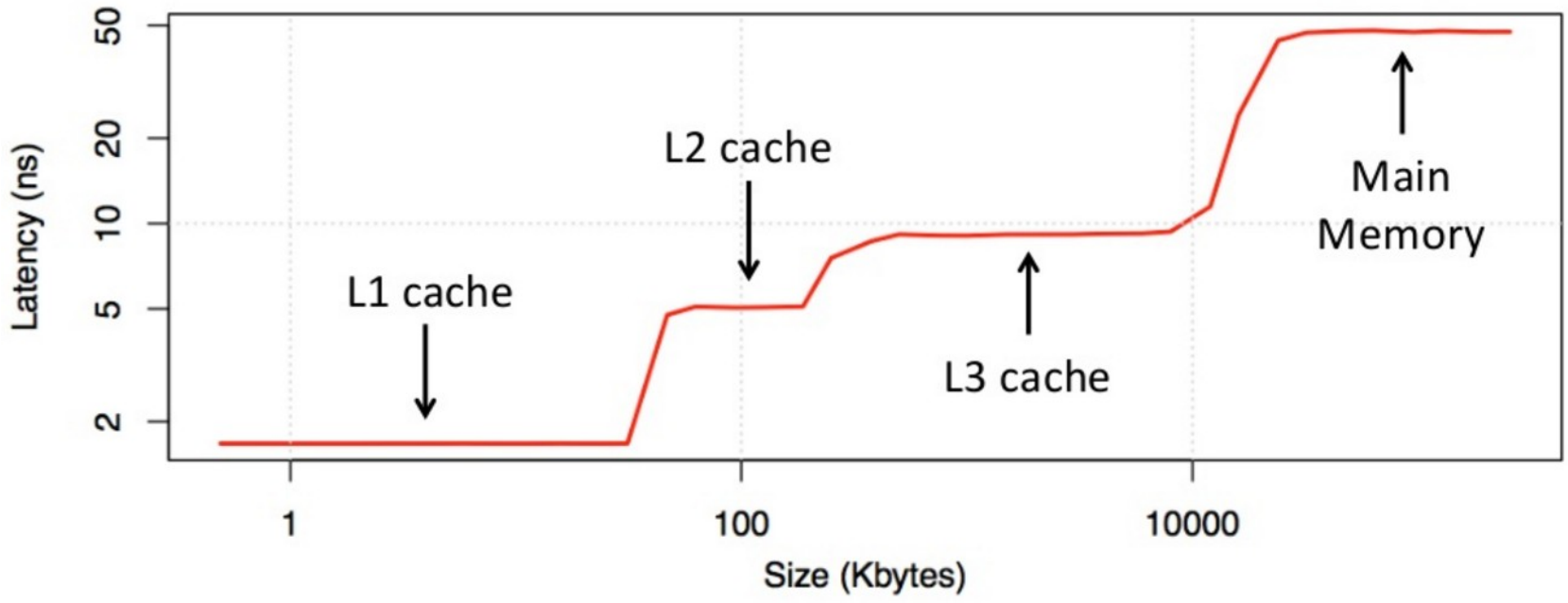
Motivazione: se il processo rimane sempre sulla stessa CPU, il suo codice ed i suoi dati sono, con alta probabilità, nelle cache hardware.

→ Gli accessi alla memoria sono effettuati alla velocità dei registri, e non a quella della RAM.

Qual è il divario di prestazioni?

Tempi di risposta di cache L1, L2 e RAM

(In funzione della dimensione del blocco di memoria; $T_{RAM} \sim 50 * T_{L1}$)



Convinti?

Affinità debole e forte

(Debole → Imposta dallo schedulatore; Forte → Forzata dall'utente)

Predilezione debole (**soft affinity**): lo schedulatore della CPU rischedula il processo sempre sulla stessa CPU, a meno di un ribilanciamento dei processi sulle diverse code di pronto.

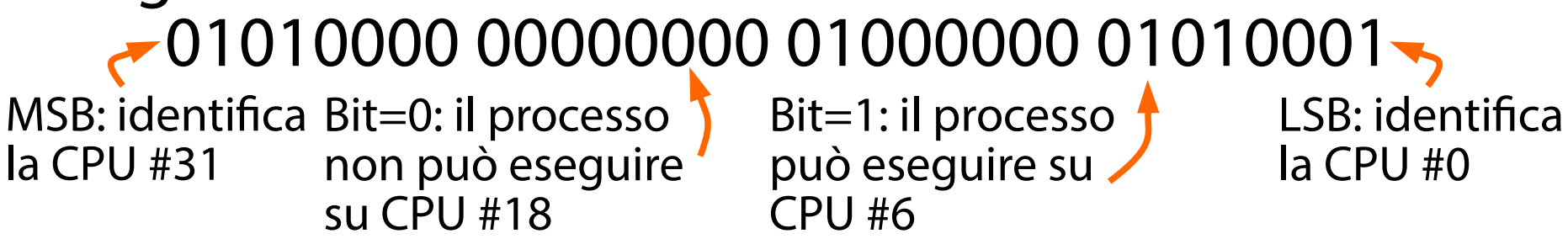
Predilezione forte (**hard affinity**): l'utente impone che il processo esegua sempre su una data CPU.

Gestione della predilezione

(Il comando taskset)

Il comando esterno **`taskset`** (contenuto nel pacchetto software **`util-linux`**) recupera ed imposta la maschera di affinità di un processo.

Maschera di affinità: maschera variabile di n bit che definisce le CPU su cui il processo può eseguire.



Recupero della maschera di affinità

(Comando taskset con l'opzione -p)

L'opzione **`-p`** di **`taskset`** stampa il valore esadecimale della maschera di affinità del processo di PID pari a **`PID`**.

```
taskset -p 1
```

Il valore ritornato è, solitamente, **`0xff`** (o **`0xf`** su macchine con poche CPU).

→ Il processo può essere schedulato su tutte le CPU.

Recupero della maschera di affinità

(schedtool, lanciato senza opzioni)

Il comando **`schedtool`**, lanciato senza opzioni e con argomento il PID del processo di interesse, mostra nel suo ultimo campo l'affinità, espressa tramite maschera di bit in esadecimale.

0x3 = 0x1 + 0x2 → Bit 0 (CPU #0) e bit 1 (CPU #1)

```
andreoli@debian-sistemi-operativi-2:~$ schedtool 1 10 1655
PID     1: PRIO   0, POLICY N: SCHED_NORMAL  , NICE   0, AFFINITY 0x3
PID    10: PRIO  99, POLICY F: SCHED_FIFO    , NICE   0, AFFINITY 0x1
PID  1655: PRIO   0, POLICY N: SCHED_NORMAL  , NICE   0, AFFINITY 0x3
```

Affinità
(maschera esadecimale)

Identificazione della CPU di esecuzione

(Comando pidstat, con l'opzione -u)

Per individuare la CPU su cui il processo sta eseguendo si esegue il comando seguente:

`pidstat -u 1`

e si legge il valore del campo **`CPU`**.

```
andreoli@debian-sistemi-operativi-2:~$ pidstat -u 1
Linux 3.16.0-4-amd64 (debian-sistemi-operativi-2)       21/10/2015      _x86_64_      (
2 CPU)

20:58:56      UID       PID    %usr %system  %guest    %CPU   CPU  Command
20:58:57        0       694    1,98    0,99    0,00    2,97     1  Xorg
20:58:57     1000      1262    0,00    1,98    0,00    1,98     0  gnome-shell
20:58:57     1000      2286    0,00    1,98    0,00    1,98     0  pidstat
```

Esercizi (3 min.)

13. Individuate il processo **`bash`** corrente.
Recuperate la sua maschera di affinità.
Individuate la CPU su cui sta eseguendo.

Impostazione affinità di un processo

(Si potrebbe usare anche schedtool con le opzioni -a e -p)

Il modo più semplice di impostare la maschera di affinità è mediante l'opzione **`-c`** di **`taskset`**. L'argomento è una elenco di numeri interi identificatori di CPU nell'intervallo [0, #CPU – 1].
Ad esempio, su una macchina con 4 CPU, per far eseguire il processo di PID 1234 sulle prime 2 CPU:

```
taskset -p -c 0,1 1234
```

Occhio. L'opzione **`-p`** va scritta per prima, seguita dall'opzione **`-c`** con il suo argomento, ed infine si scrive il PID del processo interessato.

Esecuzione comando su CPU specifiche

(Si potrebbe usare anche schedtool con le opzioni -a e -e)

È possibile eseguire on comando su specifiche CPU omettendo l'opzione **`-p`**.

```
taskset -c 0,1 top
```

Esercizi (5 min.)

14. Impostate 2 CPU per il vostro sistema guest e riavviatelo.

Lanciate il comando **`top`** sulla CPU #0.

Lanciate il comando **`sleep 1000`** sulla CPU #1.

Fate in modo che i due comandi eseguano sempre sulla stessa CPU.

Recuperate la maschera di affinità per entrambi i processi.

SCHEDULAZIONE GERARCHICA (CENNI)

Scenario e domande

(Quali sono i "meccanismi di protezione" dalla starvation?)

Scenario: il SO time sharing sta eseguendo una moltitudine di processi su poche CPU.

Domande:

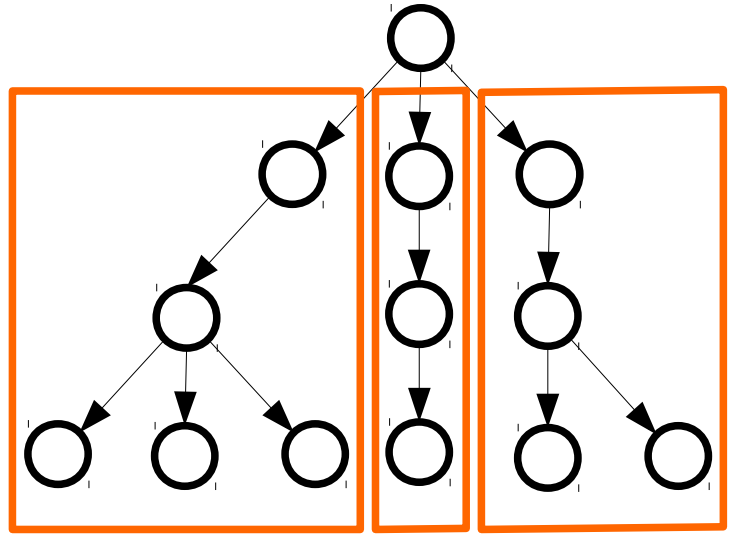
Come fa il nucleo a proteggersi da un processo CPU-bound in classe FIFO che succhia tutte le risorse di calcolo a disposizione?

Come mai, in presenza di processi CPU-bound, un comando lanciato con priorità più bassa su un altro terminale non provoca rallentamenti?

Schedulazione gerarchica

(https://www.youtube.com/watch?v=6hL1pFK2wgA)

Lo schedulatore dei processi adotta una **schedulazione gerarchica**. L'albero dei processi è diviso in tanti **gruppi di processi**.



Schedulazione gerarchica

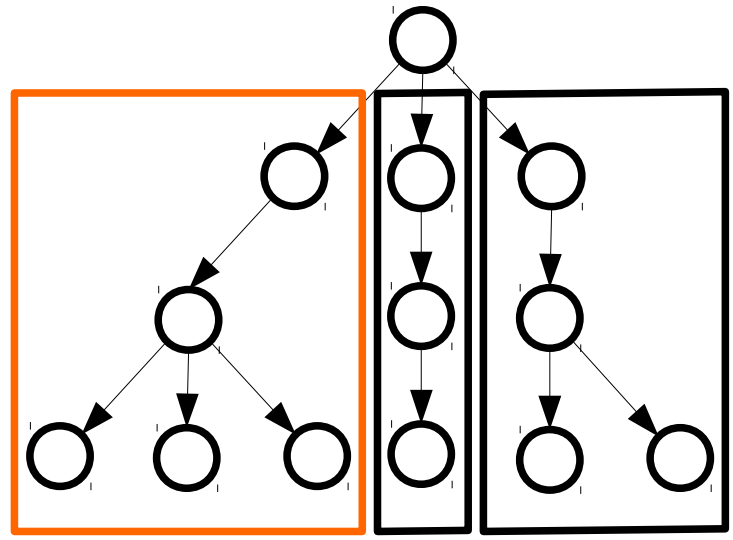
(https://www.youtube.com/watch?v=6hL1pFK2wgA)

Prima di schedulare i processi, lo schedulatore schedula i gruppi!

- In maniera proporzionale al loro peso.

Viene scelto per primo il gruppo 1.

Per 1/N del tempo si schedulano esclusivamente processi del suo gruppo.

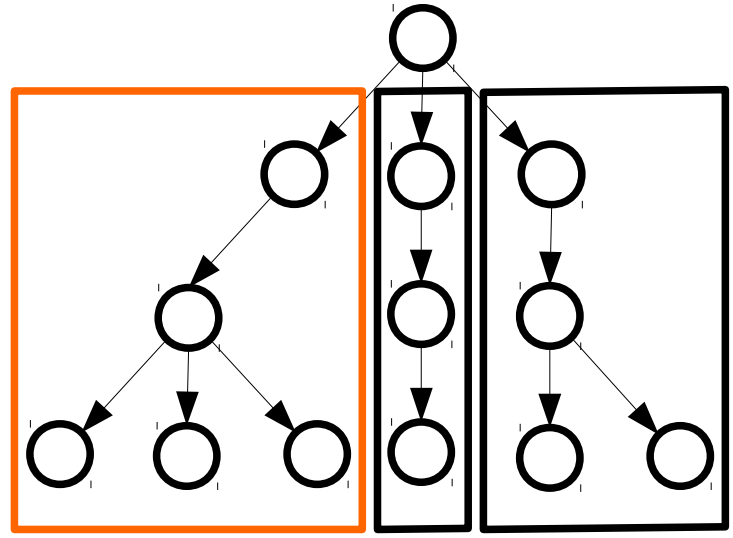


Schedulazione gerarchica

(https://www.youtube.com/watch?v=6hL1pFK2wgA)

All'interno del gruppo:

- si scandiscono le code dei processi pronti.
- si schedulano le classi.
- si schedulano I processi delle classi.

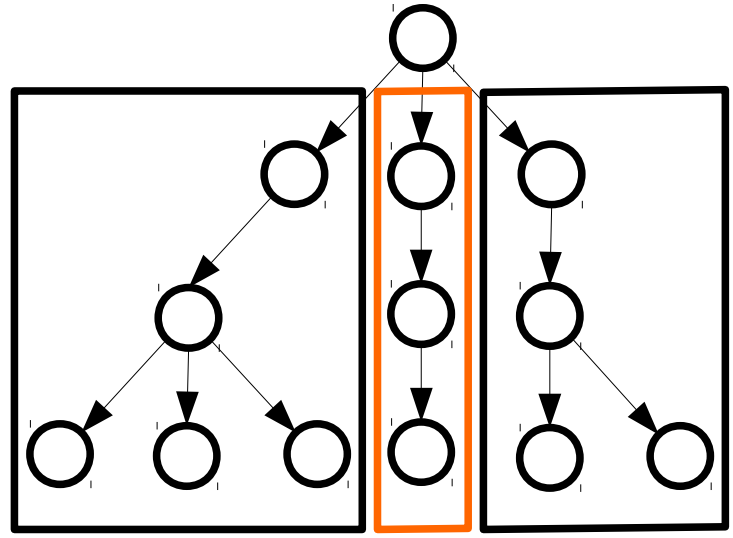


Schedulazione gerarchica

(https://www.youtube.com/watch?v=6hL1pFK2wgA)

Poi si passa al gruppo 2.
All'interno del gruppo:

- si scandiscono le code dei processi pronti.
- si schedulano le classi.
- si schedulano I processi delle classi.



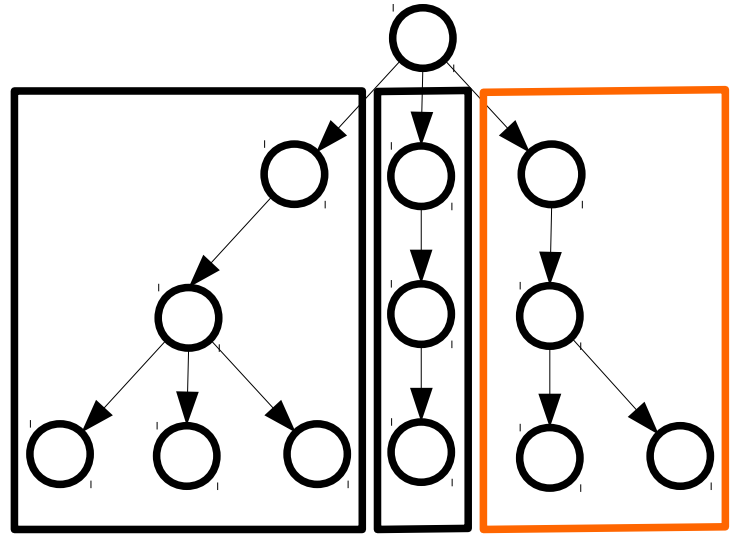
Schedulazione gerarchica

(https://www.youtube.com/watch?v=6hL1pFK2wgA)

A ciascun gruppo viene assegnata una **frazione di tempo di CPU** (**CPU share**).

Solitamente, 1/N del tempo di CPU per ciascun gruppo.

Può essere reimpostata da un amministratore.



Come sono organizzati i gruppi?

(Vi sono tanti modi)

I gruppi di processi possono essere costituiti in svariati modi. Ecco i più popolari.

Tutti i processi figli di un server fanno parte dello stesso gruppo.

Tutti i processi agganciati ad uno specifico terminale fanno parte dello stesso gruppo.

Tutti i processi di una sessione grafica fanno parte dello stesso gruppo.

L'utente può raggruppare a mano processi in gruppi.

Inefficacia di nice su terminali diversi

(https://www.youtube.com/watch?v=NHKiNDLB45I)

Grazie allo scheduling gerarchico, se lanciate:

- su un terminale 10 processi CPU-bound
- su un altro terminale solo un **`ls`** a priorità bassa

`ls` eseguirà per la stessa frazione di tempo dei 10 processi CPU-bound!

- Un emulatore di terminale e **`ls`** costituiscono un gruppo. Dentro il gruppo, il terminale attende e **`ls`** esegue.
- Un emulatore di terminale e 10 processi CPU-bound costituiscono un altro gruppo. Dentro il gruppo, il terminale attende ed i 10 processi eseguono.

Protezione contro la starvation

(https://www.youtube.com/watch?v=NHKiNDLB45I)

Ogni tentativo di sottrarre CPU tramite esecuzione infinita è destinato a fallire.

- Ad esempio, un processo CPU-bound di classe SCHED_FIFO lanciato a priorità elevata.

La schedulazione per gruppi confina le applicazioni ghiotte di CPU ad una frazione del tempo di CPU (ad esempio, 1/N).