Lezione 18
Algoritmi di scheduling

Sistemi Operativi (9 CFU), CdL Informatica, A. A. 2020/2021
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...)

“Fools ignore complexity.
Pragmatists suffer it.
Some can remove it.
Geniuses remove it.”

Alan J. Perlis (1922-1990)
Inventore del linguaggio ALGOL
Vincitore del primo Turing Award (1966)

INTRODUZIONE

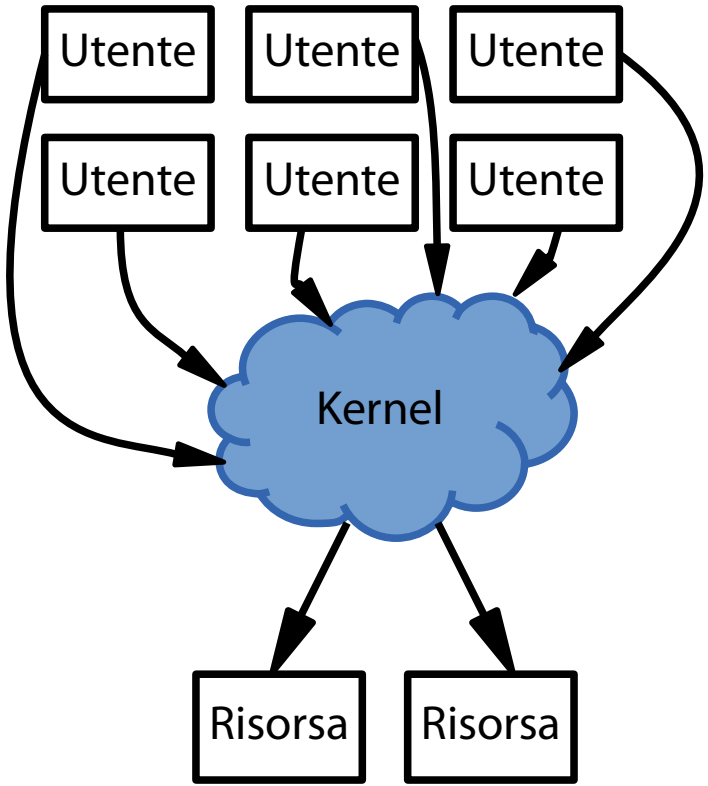
Lo scenario

(http://www.youtube.com/watch?v=AN7M4ULqz8Y)

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

Vincolo fondamentale: le entità richiedenti eccedono le istanze di risorsa presenti.

Come assegnare le richieste alle risorse?

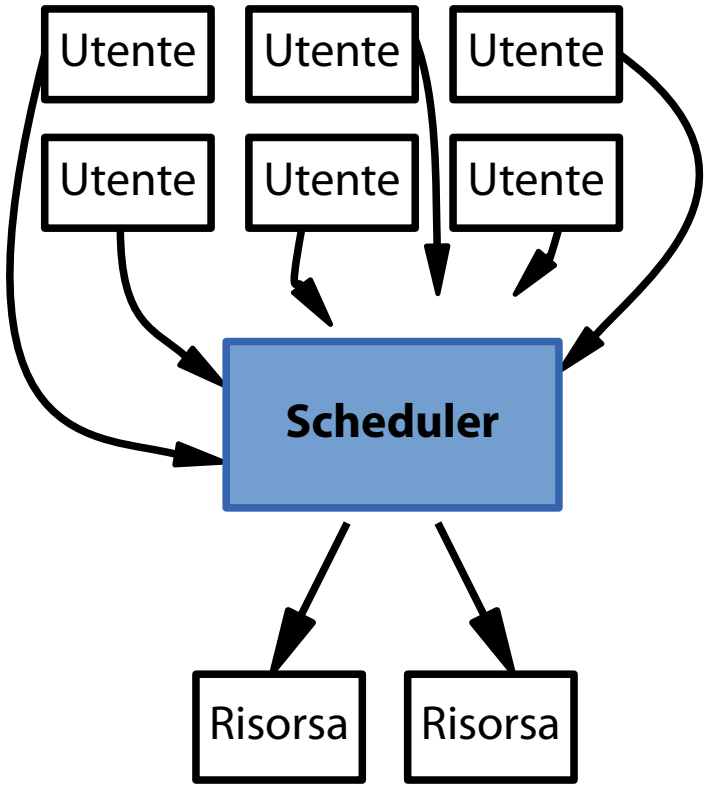


Lo scheduler

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

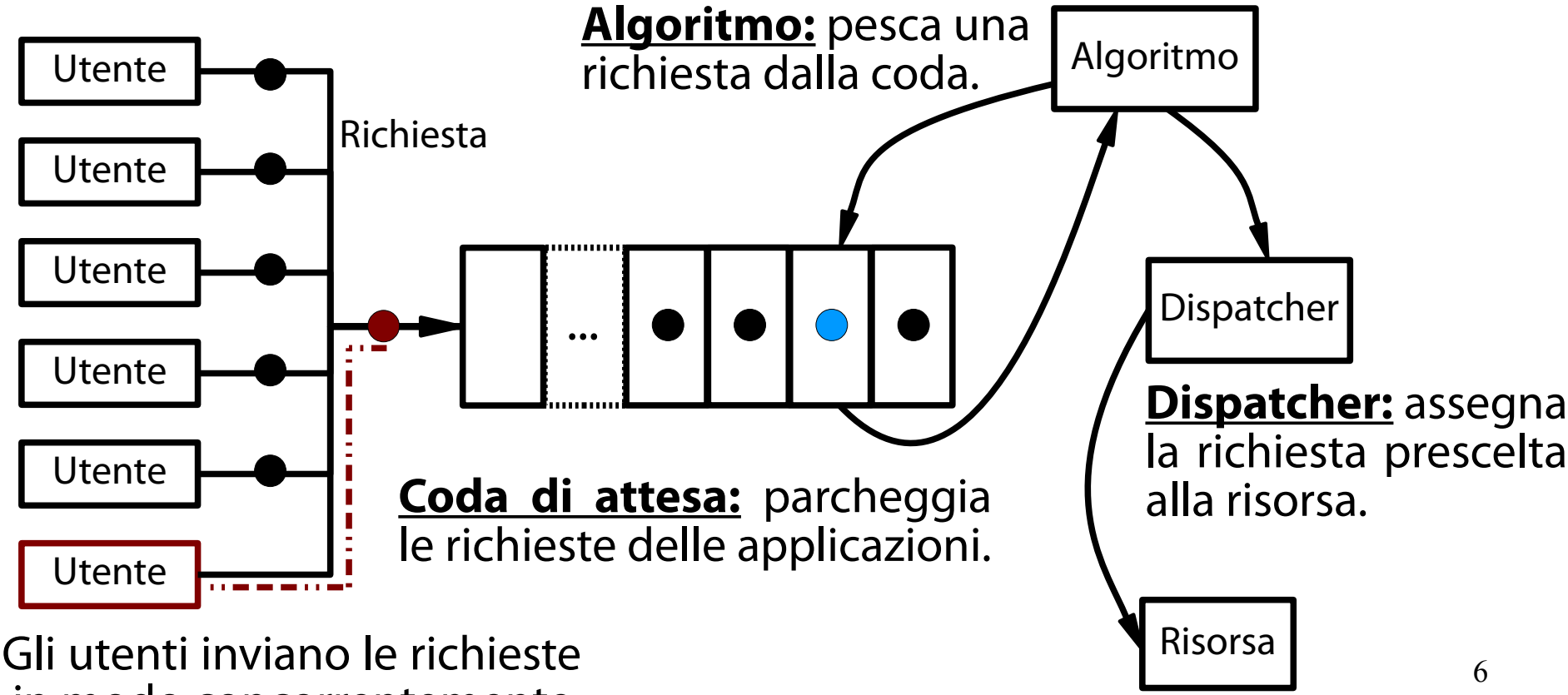
Uno **scheduler** arbitra l'accesso alle risorse.

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



Un semplice modello

(Scheduler = code di attesa + algoritmo + meccanismo di dispatch)



Algoritmo: pesca una richiesta dalla coda.

Coda di attesa: parcheggia le richieste delle applicazioni.

Dispatcher: assegna la richiesta prescelta alla risorsa.

Gli utenti inviano le richieste in modo concorrentemente.

Alcuni scenari di riferimento

(Lo scheduler è decisivo in alcuni di questi)

CPU scheduling: scelta del prossimo processo da eseguire. Entità=processi, risorsa=CPU.

Job scheduling: scelta del prossimo job (stampa, posta, cron) da eseguire. Entità=job, risorsa=CPU.

I/O scheduling: scelta della prossima richiesta di disco (lettura/scrittura) da soddisfare. Entità=richieste di I/O, risorsa=disco.

Memory scheduling: scelta di una pagina di memoria da spostare in swap. Entità=pagine di memoria, risorsa=memoria.

Obiettivi

(A cosa deve mirare uno scheduler?)

1. Evitare periodi di inattività della risorsa; più in generale, aumentarne l'utilizzo.
2. Aumentare il numero di assegnazioni di risorse per unità di tempo.
3. Ridurre l'attesa per una risorsa.
4. Assegnare la risorsa in maniera equa fra i richiedenti.

Scheduling con e senza prelazione

(Meglio con prelazione)

Con prelazione. Lo scheduler interrompe la fruizione della risorsa da parte di un utente per favorirne un altro.

Senza prelazione. Lo scheduler non interrompe la fruizione della risorsa. L'utente continua ad usufruire della risorsa fino a quando

- non termina la sua funzione.
- non necessita di un'altra risorsa.

Scheduling senza prelazione
(Meglio con prelazione)

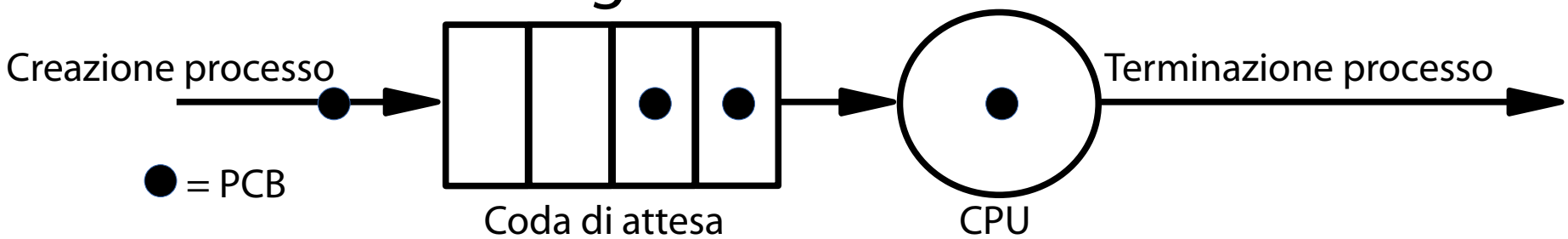
Scheduling cooperativo. L'utente stesso può decidere di assegnare la risorsa direttamente ad un altro utente, con una operazione che prende il nome di **yield**.
→ Scheduler "distribuito" fra kernel ed utenti.
Scheduling non cooperativo. L'utente non influisce sulle scelte decisionali dello scheduler.

SCHEDULING DELLA CPU

Modello di riferimento

(Senza prelazione)

Nel resto della presentazione si considerano algoritmi di scheduling di processi (**scheduling della CPU**). I processi non effettuano richieste ad altre risorse. Non esiste prelazione. Il modello di riferimento è il seguente.



Indici di prestazione

(Lo scheduler si comporta bene o male?)

Un **indice di prestazione** è una grandezza in grado di rivelare il livello di prestazione di un componente hw/sw.

Nel caso di uno scheduler della CPU, la prestazione può essere valutata dai seguenti punti di vista:

- attesa di un processo (ad iniziare o a completare la elaborazione).
- produttività dello scheduler.
- utilizzo del processore.

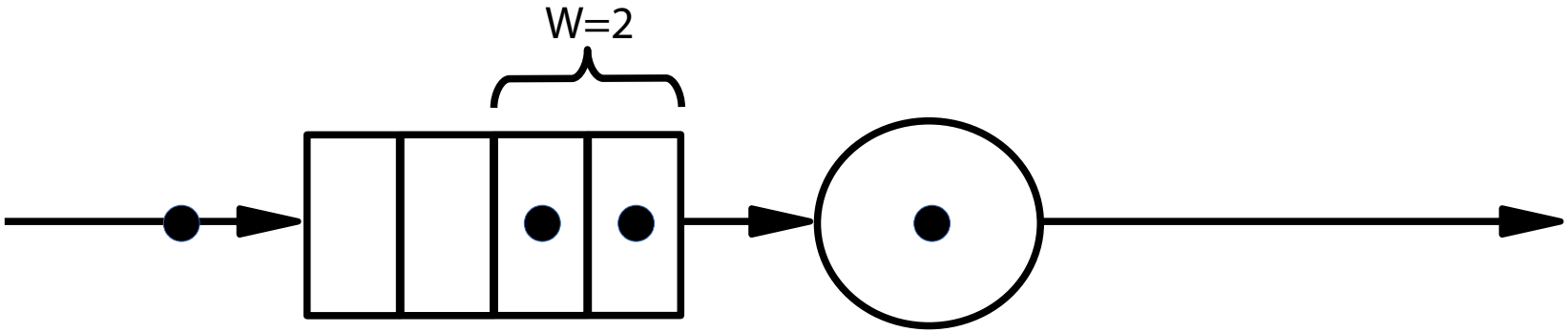
Numero di processi in coda

(Quante persone ho davanti a me in fila allo sportello?)

Definizione: è il numero di processi W accodati in uno specifico istante $t=t_1$.

A che serve: misura l'attesa di un processo ad iniziare la elaborazione e lo stato di congestione del processore.

Valore ideale: più basso è, meglio è.



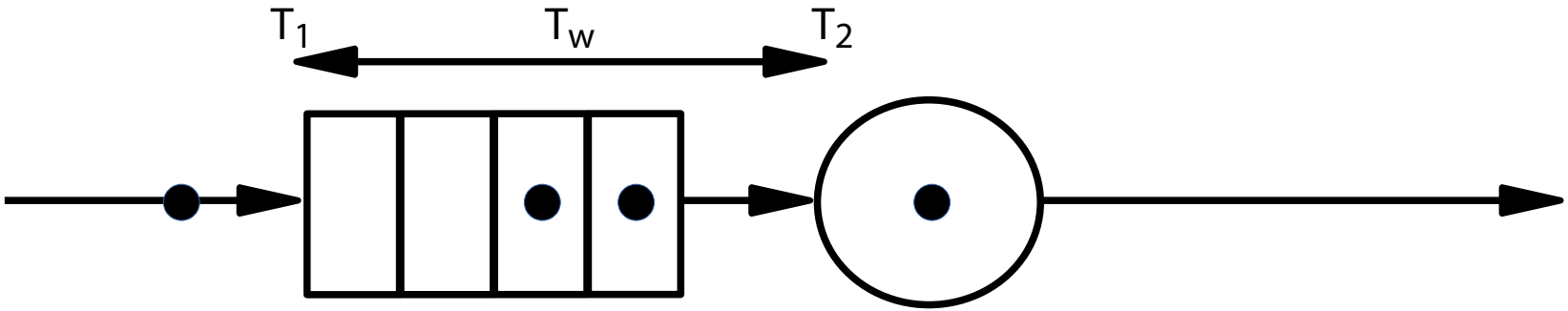
Tempo di attesa in coda

(Quanto è lunga l'attesa in fila?)

Definizione: è l'intervallo temporale fra l'ingresso di un processo in coda e l'associazione al processore: $T_w=T_2-T_1$.

A che serve: misura l'attesa di un processo ad iniziare la elaborazione e lo stato di congestione del processore.

Valore ideale: più basso è, meglio è.



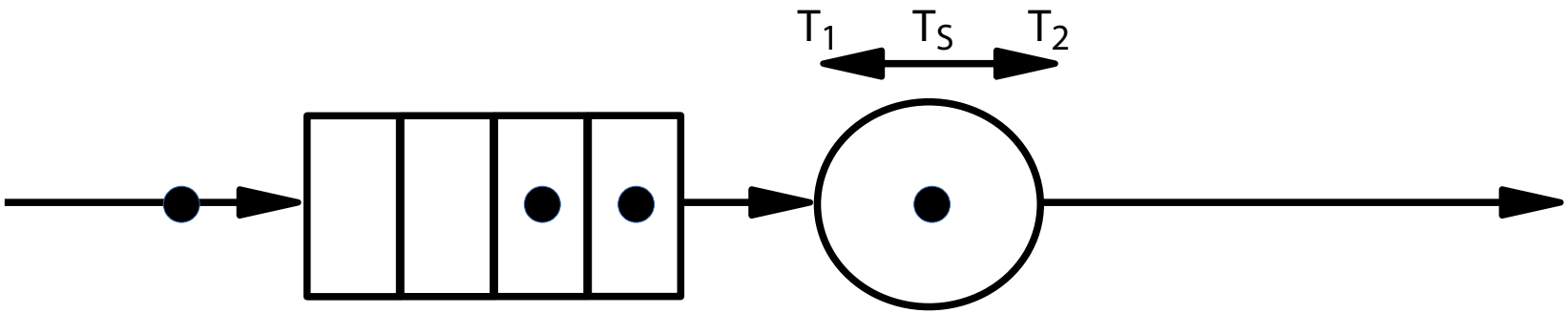
Tempo di servizio

(Quanto è veloce il tipo allo sportello?)

Definizione: è l'intervallo temporale in cui il processore esegue codice del processo: $T_S=T_2-T_1$.

A che serve: misura l'attesa di un processo a completare l'elaborazione.

Valore ideale: più basso è, meglio è.



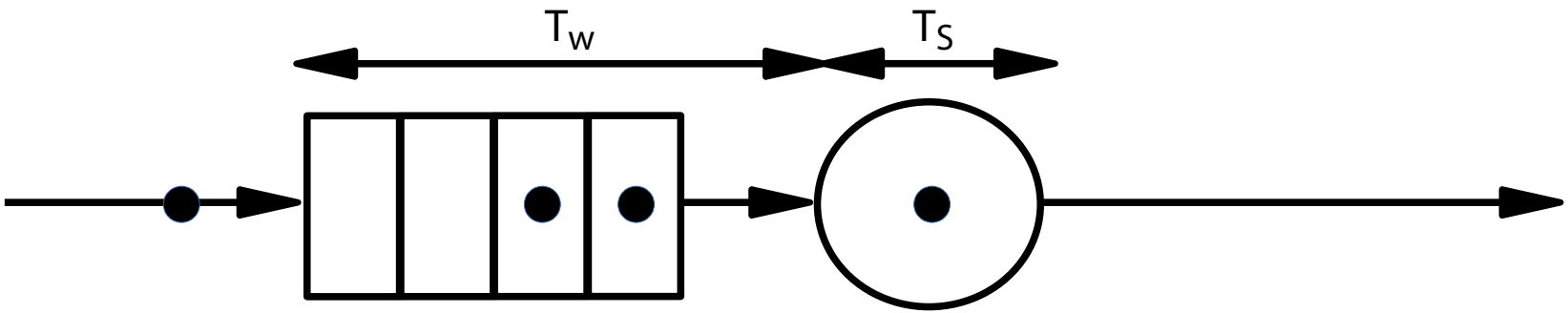
Tempo di completamento

(Quanto ci metto a finire la pratica, dall'ingresso all'uscita?)

Definizione: è la somma dei tempi di attesa in coda e di servizio: $T_C = T_W + T_S$.

A che serve: misura l'attesa complessiva di un processo.

Valore ideale: più basso è, meglio è.



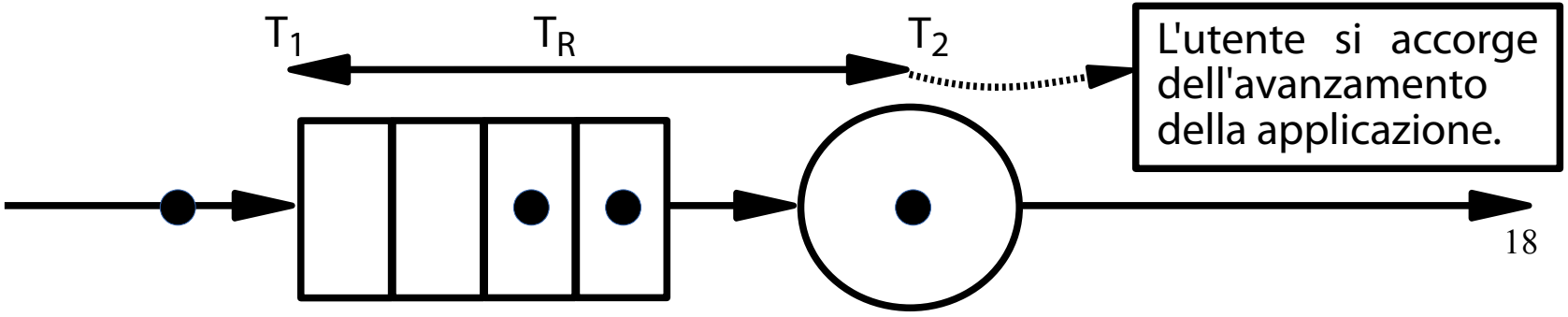
Latenza

(Quando inizia il tipo alla cassa a rispondermi?)

Definizione: è l'intervallo temporale fra l'ingresso di un processo in coda ed il calcolo del primo byte della risposta utile all'applicazione: $T_R=T_2-T_1$.

A che serve: misura l'attesa di un processo ad iniziare la elaborazione e lo stato di congestione del processore. È l'attesa percepita dall'utente che ha lanciato il processo.

Valore ideale: più basso possibile per i processi interattivi.



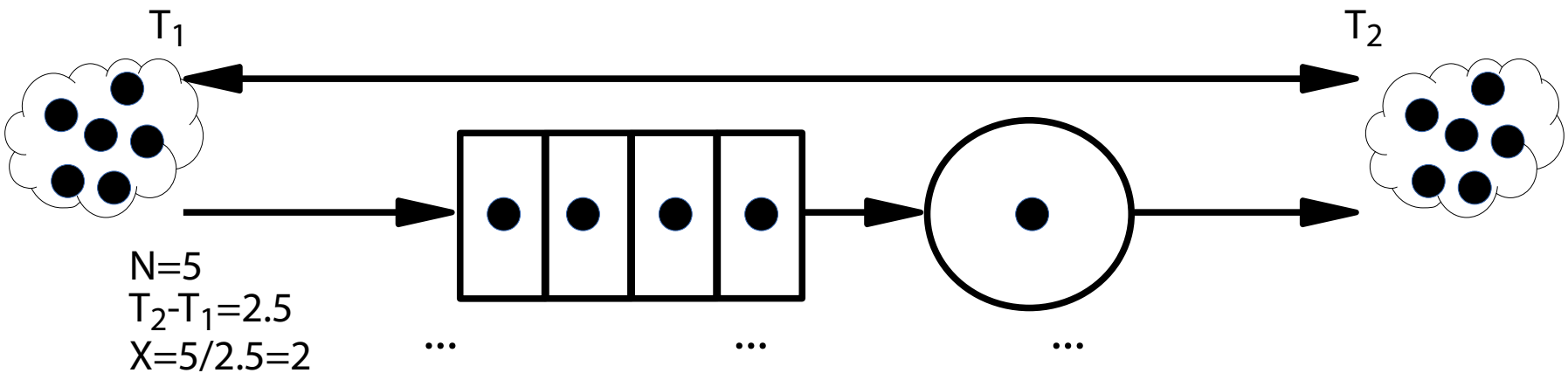
Throughput

(Quante persone serve il tipo alla cassa ogni secondo/minuto/ora?)

Definizione: è il numero di processi portati a termine da uno scheduler nell'unità di tempo: $X=N/(t_2-t_1)$.

A che serve: misura la produttività dello scheduler.

Valore ideale: più alto possibile per i processi non interattivi.



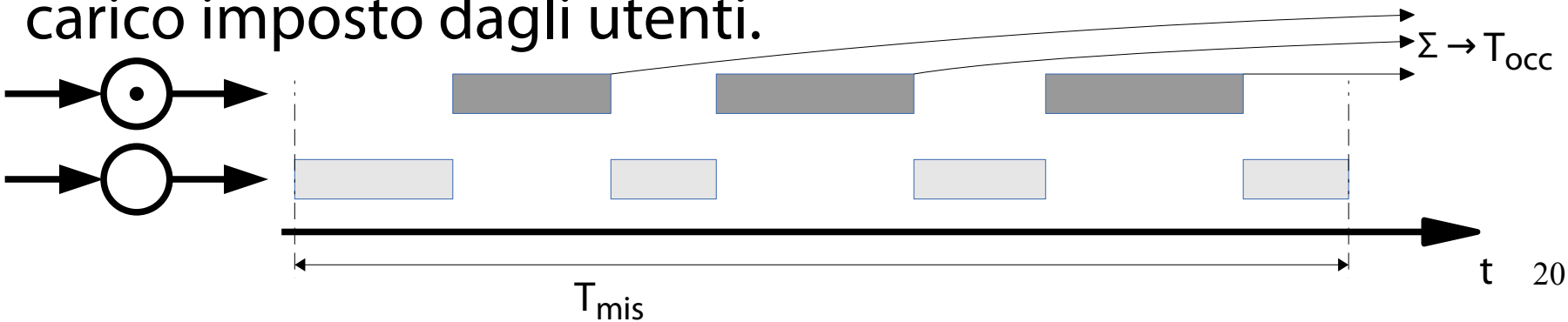
Utilizzazione

(Quanto tempo è stato occupato, in percentuale, il tipo alla cassa?)

Definizione: è frazione di tempo in cui il processore è occupato (T_{occ}) in un dato intervallo di misurazione (T_{mis}): $\rho = T_{occ}/T_{mis}$.

A che serve: misura l'utilizzo del processore (probabilità di trovarlo occupato in un intervallo di tempo).

Valore ideale: più alto possibile, compatibilmente con il carico imposto dagli utenti.



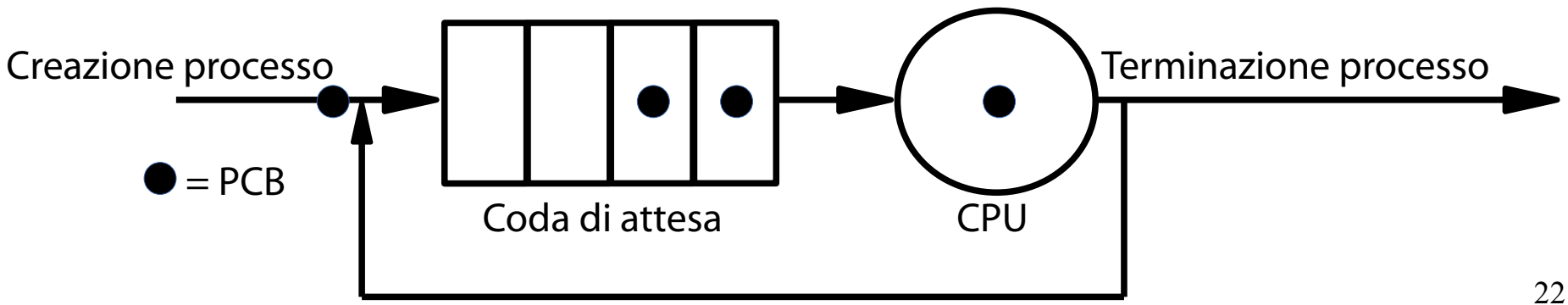
Indici di prestazione: quali scegliere?

(Quale aspetto dello scheduler si vuole valutare?)

Aspetto	**Indice di prestazione**
Interattività fornita ai processi	Latenza
Produttività dello scheduler	Throughput
Produttività dei processi	Throughput Utilizzazione del processore
Livello di congestione	Tempo di attesa Latenza Numero di utenti in coda Tempo di completamento

Modello di riferimento
(Con prelazione)

Nel modello con prelazione si deve prevedere il rientro di un processo nella coda di attesa.
Un processo può subire più attese in coda e ricevere più volte servizio dal processore.



Indici riassuntivi

(Statistica serve)

Per ogni schedulazione possono esistere più misurazioni degli indici di prestazioni.

In generale: si ha almeno una misura per ogni processo coinvolto (una sola in assenza di I/O).

Schema con prelazione: si ha una misura per ogni processo coinvolto e, in alcuni casi (ad es. tempo di attesa) per ogni rientro del processo in coda.

→ Necessità di **riassumere** tali misurazioni in un **unico valore complessivo**.

Indici riassuntivi

(Statistica serve)

Esempi di **indici riassuntivi** sono i seguenti.

Valori minimo, massimo: quantificano la prestazione migliore e peggiore ottenuta.

Valore medio (con annessa **deviazione standard**): quantifica la prestazione in media (e la sua variabilità)

- fra diversi processi.
- dello stesso processo nei suoi diversi rientri in coda di attesa (presenza di prelazione).

OSS.: più è alta la deviazione standard, più la prestazione è oscillante (pessimo!).

Valore complessivo: quantifica l'attesa ed il servizio totali del processo durante la sua esecuzione.

Throughput vs. Latenza

(Sorry, you can't have it all)

Throughput e latenza sono, in generale, **obiettivi contrastanti** fra loro.

- Uno scheduler ad alto throughput serve tutto ciò che si presenta in coda → Non si fa distinzione fra processi interattivi e non → La latenza di tali processi aumenta.
- Uno scheduler a bassa latenza tende a scegliere i processi interattivi → Gli altri processi non avanzano → Il throughput dello scheduler cala.

ALGORITMI DI BASE

FIRST COME, FIRST SERVED

(Il processo che si accoda per primo è servito per primo)

Opera senza prelazione.

→ Un processo esegue fino all'I/O o alla fine.

Funzionamento:

I processi accodano con disciplina First In.

Si assegna il processore al processo in cima alla coda (disciplina First Out).

18-simulazione-FCFS.pdf

FCFS: vantaggi
(Perché implementare ed usare FCFS)

Algoritmo molto semplice.
- Il **più** semplice.

Eseguibile su piattaforme non dotate di clock hardware.
- Non implementa prelazione.
- → i processi non devono essere interrotti bruscamente.
- → non serve un timer per avviare periodicamente il meccanismo di interruzione.

FCFS: svantaggi 1/2

(Perché NON implementare e NON usare FCFS)

L'attesa di un processo non è, in generale, minima. Essa varia anche sensibilmente al variare della composizione dei processi (lunghezza, ordine di accodamento).

Si consideri, ad esempio, il tempo di attesa medio $\overline{T}_w=(T_w(P_1)+T_w(P_2)+T_w(P_3))/3$.

Arrivo $(P_1,P_2,P_3) \rightarrow \overline{T}_w=(0+24+27)/3=17$

Arrivo $(P_2,P_3,P_1) \rightarrow \overline{T}_w=(0+3+6)/3=3$ (!)

FCFS: svantaggi 2/2
(Perché NON implementare e NON usare FCFS)

FCFS è senza prelazione. Un processo CPU-bound monopolizza il processore e stalla l'esecuzione degli altri processi (starvation).
→ Pessimo nei sistemi time sharing interattivi, specie se il processo soggetto a starvation è un terminale dell'interfaccia utente (testuale o grafica).

SHORTEST JOB FIRST

(Il processo "più piccolo" è servito per primo)

Opera senza prelazione.

→ Un processo esegue fino all'I/O o alla fine.

Funzionamento:

I processi accodano con disciplina First In.

Si assegna il processore al processo con il CPU burst successivo più piccolo fra quelli in coda. Se esistono più processi con CPU burst minimo, si applica il criterio FCFS.

18-simulazione-SJF.pdf

SJF: vantaggi
(Perché implementare ed usare SJF)

In assenza di prelazione, si può dimostrare che SJF è l'algoritmo di scheduling che fornisce tempo di attesa medio minimo. Ossia, dati:

n processi $P_1, P_2, \ldots, P_n$;

$T_w(P_j)$ tempo di attesa del processo P_j;

$\overline{T}_w=(1/n)*[T_w(P_1)+T_w(P_2)+...+T_w(P_n)]$ tempo medio di attesa dei processi al termine della schedulazione.

→ $\overline{T}_w$ fornito da SJF è il minimo fra tutti gli scheduler senza prelazione.

SJF: svantaggi 1/2
(Perché NON implementare e NON usare SJF)

SJF richiede la conoscenza della durata del prossimo CPU burst di ciascun processo.

- In generale, non la si può conoscere a priori (il SO non è dotato di palla di vetro). La durata va stimata in qualche modo.

Lo scheduler eseguito non è SJF, bensì un "simil-SJF" con stima dei CPU burst.

SJF: svantaggi 2/2
(Perché NON implementare e NON usare SJF)

SJF considera l'intera durata del processo. Tuttavia, in uno scheduler con prelazione, i processi sono eseguiti “per piccoli pezzi” e le durate residue divergono rapidamente dalla durata iniziale.

→ L'informazione su cui fa conto SJF diventa molto rapidamente obsoleta.

Stima durata prossimo CPU burst

(Ovvero come rendere utilizzabile SJF)

Una stima buona della durata del prossimo CPU burst è la **media esponenziale** τ_n dei CPU burst passati e presente.

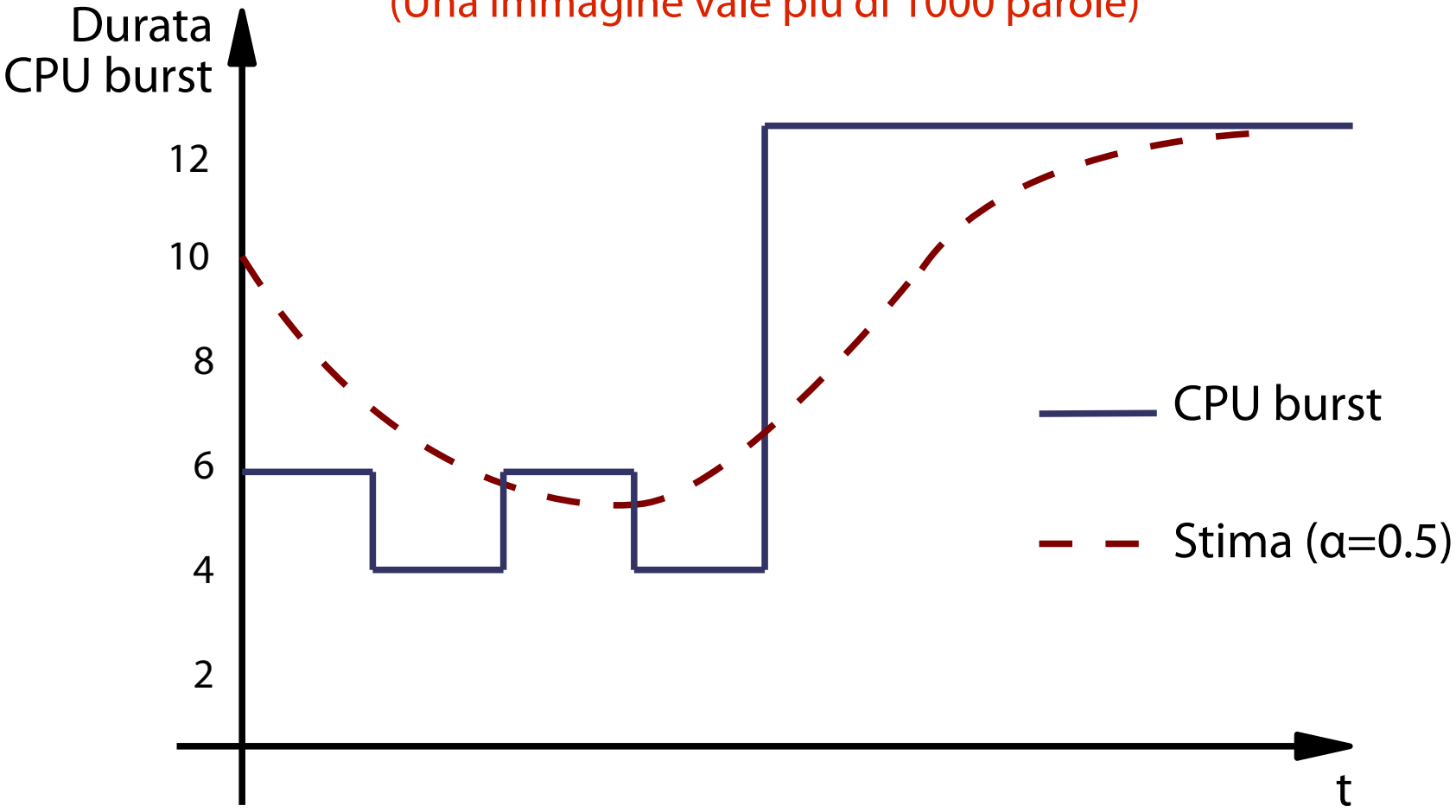
$$\tau_{n+1} = \alpha * t_n + (1 - \alpha) * \tau_n \, , \alpha \in [0, 1]$$

Il parametro α "tara" il quantitativo di storia passata da prendere in considerazione.

- α=0: valore recente non ha effetto
- α=1: assenza di storia
- α=1/2: stesso peso per valore recente e storia passata

Stima durata prossimo CPU burst

(Una immagine vale più di 1000 parole)



SHORTEST REMAINING TIME FIRST

(Il processo con la rimanenza più piccola è servito per primo)

Opera con prelazione.

→ Un processo può essere interrotto dallo scheduler.

Funzionamento:

I processi accodano con disciplina First In.

Si assegna il processore al processo con il CPU burst residuo più piccolo fra quelli in coda e quello in esecuzione al momento della decisione.

Variante di SJF pensata per i sistemi time sharing.

18-simulazione-SRTF.pdf

SRTF: vantaggi
(Perché implementare ed usare SRTF)

SRTF diminuisce il tempo di attesa medio rispetto a SJF.

In generale, la prelazione accorcia il tempo di attesa.

SJF: $\overline{T}_w(P_1, P_2, P_3, P_4)=0+4+(4+5)+(4+5+8)]/4 = \mathbf{7.75}$

SRTF: $\overline{T}_w(P_1, P_2, P_1, P_1, P_3) =$

$[\overline{T}_w(P_1) + \overline{T}_w(P_2) + \overline{T}_w(P_3) + \overline{T}_w(P_4)] / 4 =$

$[\ 0+(10-1) + (1-1) + (17-2) + (5-3)] / 4 =$

6.5

SRTF: svantaggi
(Perché NON implementare e NON usare SRTF)

SRTF richiede la conoscenza della durata del CPU burst residuo di ciascun processo.

- In generale, non la si può conoscere a priori (il SO non è dotato di palla di vetro). La durata va stimata in qualche modo.

Lo scheduler eseguito non è SRTF, bensì un "simil-SRTF" con stima dei CPU burst.

PRIORITY

(Il processo più importante è servito per primo)

Opera con o senza prelazione.

Funzionamento:

- I processi accodano con disciplina First In.
- Si associa ad ogni processo una priorità.
- Si assegna il processore al processo con la priorità più alta.
- Se due processi hanno uguale priorità, si applica il criterio FCFS.

18-simulazione-PRIO.pdf

PRIORITY: vantaggi

(Perché implementare ed usare PRIORITY)

PRIORITY favorisce i processi ritenuti più importanti. Tale modo di operare è fondamentale nei sistemi interattivi.

→ Processo importante: processo che interagisce "spesso" e direttamente con l'utente (terminale testuale o grafico, editor, gioco, riproduttore multimediale, …).

PRIORITY: svantaggi

(Perché NON implementare e NON usare PRIORITY)

PRIORITY è soggetto a starvation dei processi.
Nella simulazione ora vista, P_4 ha tempo di attesa pari a 18 → starvation.
Se P_4 fosse un terminale, stallerebbe per lungo tempo e l'utente non riuscirebbe ad interagire con il SO.

Come combattare la starvation

(Il processo anziano è importante e merita rispetto)

Gli scheduler con priorità necessitano di un ulteriore meccanismo per combattere l'attesa indefinita.

Aging: aumento graduale della priorità dei processi in attesa da lungo tempo.

PRIORITY: scelta del valore di priorità

(Che numeri scegliere? Chi li imposta?)

Le priorità sono solitamente **numeri interi** con un ordinamento. Possono essere definite in due modi.

Internamente: dal SO, in base a grandezze misurabili.

Limiti di tempo, memoria, CPU burst.

Rapporto (avg I/O burst) / (avg CPU burst).

Esternamente: in base a criteri esterni al SO (importanza del processo).

Si associa ad ogni processo una priorità.

Sistemi UNIX: valore intero in [-20, 19] (più è basso, più è alta la priorità).

ROUND ROBIN

(Giro giro tondo...)

Opera con prelazione.

→ Un processo può essere interrotto dallo scheduler.

Funzionamento:

I processi accodano con disciplina First In.

Viene scelto il processo in cima alla coda.

Ciascun processo può eseguire al più un per un intervallo di tempo prestabilito, dopdodiché si passa al processo successivo.

FIFO con prelazione.

18-simulazione-RR.pdf

RR: vantaggi
(Perché implementare ed usare RR)

L'algoritmo RR fornisce, in generale, attese molto basse.

RR: $\overline{T}_w(P_1, P_2, P_3) =$
$[\overline{T}_w(P_1) + \overline{T}_w(P_2) + \overline{T}_w(P_3)] / 3 =$
$[\ 6 \quad + 4 \quad + 7 \quad] / 3 =$
5.66

Con n processi ed un quanto di tempo pari a q, l'attesa di un processo è limitata superiormente a:

(n-1) * q

RR: svantaggi
(Perché NON implementare e NON usare RR)

L'algoritmo Round Robin fornisce tempi di completamento di processi CPU-bound più lunghi rispetto a FCFS.

- FCFS esegue i CPU burst “in toto”, fino alla fine, mentre RR si alterna fra processi.

Quanti di tempo fissi e variabili

(Quanta vs. time slice)

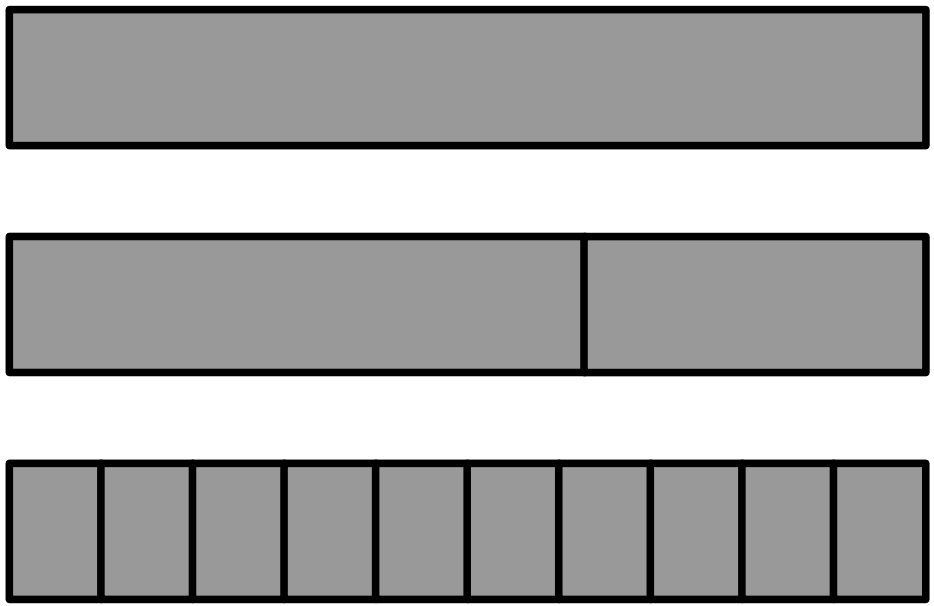
Quanto fisso: il quanto è scelto una volta per tutte e non cambia mai. È molto semplice da gestire, ma un quanto fisso non va mai bene per tutte le categorie di processi.

Time slice variabile: il kernel calcola dinamicamente l'intervallo massimo di esecuzione, in funzione della tipologia di processo e della sua priorità di esecuzione.

Impatto del quanto sulle schedulazioni

(Quanto piccolo → tante schedulazioni)

Tempo di esecuzione del processo = 10



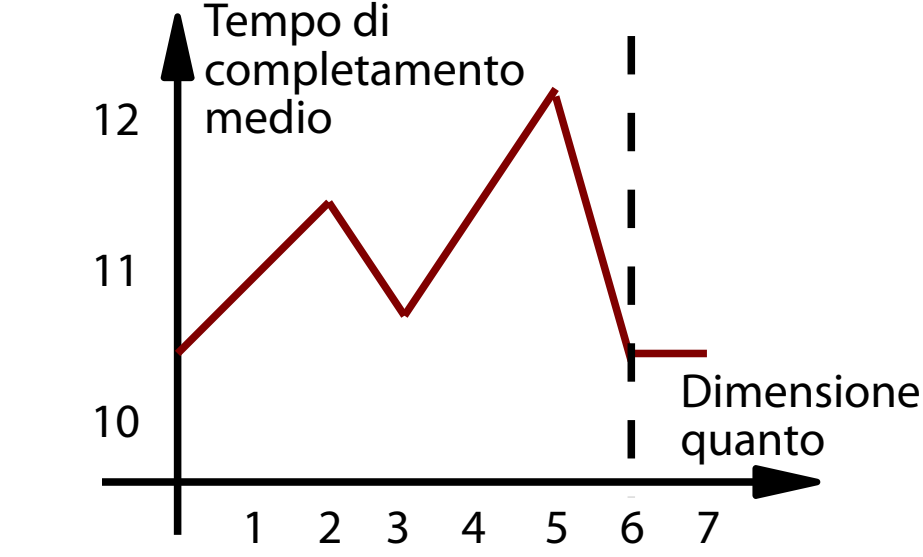
Quanto	Cambi di contesto
12	0
6	1
1	9

Impatto del quanto sul completamento

(Quanto grande → i processi finiscono in un sol quanto)

Il tempo di completamento medio tende a diminuire se la maggior parte dei processi finisce il CPU burst entro lo scadere del quanto.

Regoletta empirica: l'80% dei processi (3 su 4 nell'esempio) dovrebbe avere CPU burst minori del quanto → quanto=6.



Processi	CPU burst medi
P_1	6
P_2	3
P_3	1
P_4	7

80% dei processi (3 su 4) ha quanto ≤6.

Riassumendo

("In medio stat virtus")

Il quanto di tempo va scelto

- molto più grande della durata del dispatching (almeno 10-100 volte più grande).
- abbastanza grande da contenere l'80% dei CPU burst medi dei processi.
- non troppo più grande, altrimenti degenera in un FCFS che è disastroso nei sistemi interattivi.

SCHEDULING MULTILIVELLO

Categorie di processi nei SO moderni
(Foreground, background, batch)

Processi interattivi (**foreground**): devono andare in esecuzione il più presto possibile.
Processi non interattivi brevi (**background**): devono andare in esecuzione meno spesso degli interattivi.
Processi non interattivi lunghi (**batch**): vanno in esecuzione solo se non ne esistono di interattivi e non interattivi brevi.

Problematiche

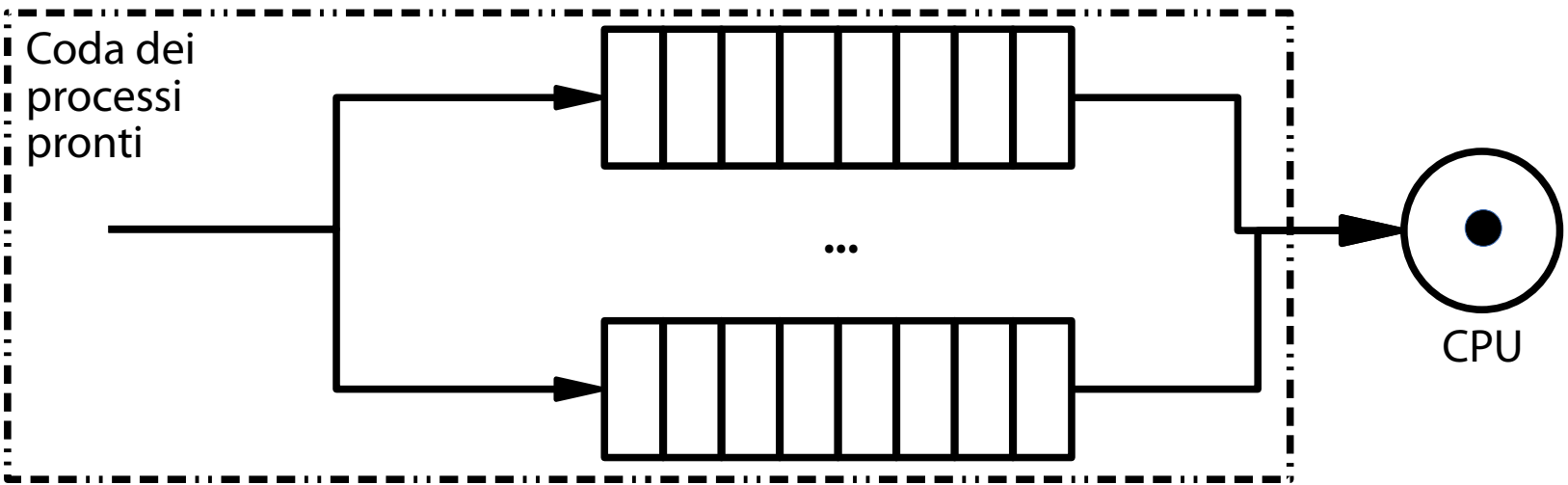
(Quali difficoltà aggiuntive comporta la presenza di diverse categorie?)

È necessario **classificare i processi**, in modo da sapere a quale categoria appartiene ciascuno di essi.
È necessario **saper schedulare prima** i processi appartenenti alle **classi più importanti**.
Come si possono risolvere questi due problemi?
→ **Scheduler multilivello** (**multilevel queue**)

Scheduler multilivello

(Si usano più code, una per ogni categoria di processi)

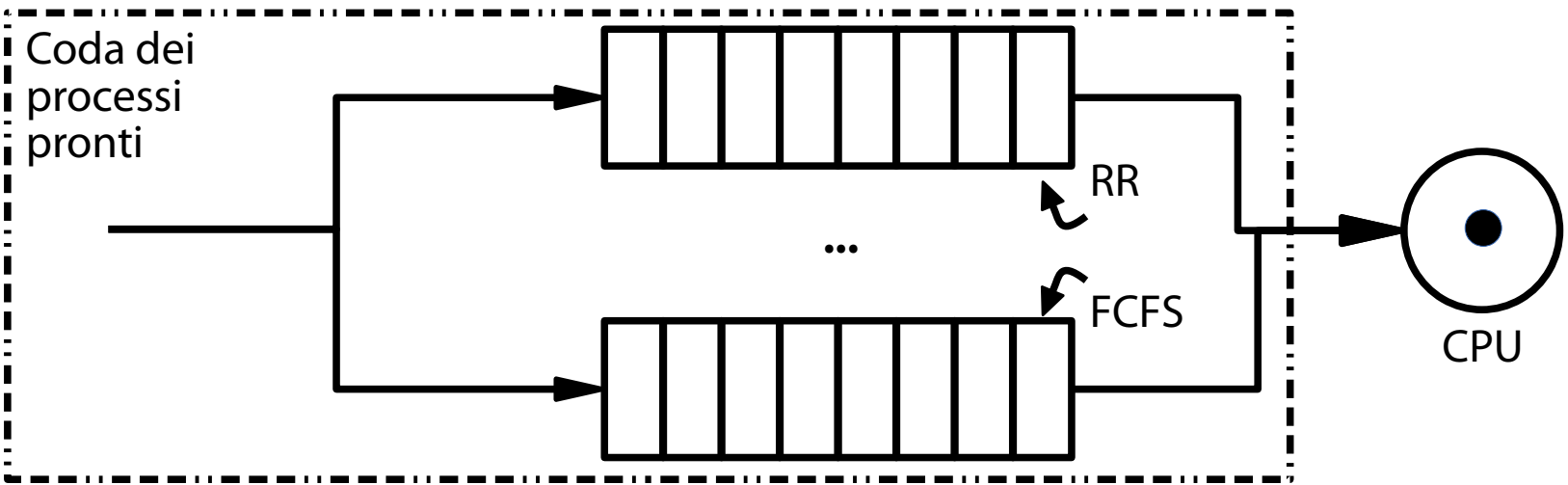
La coda di attesa dei processi pronti è suddivisa in più code distinte.



Scheduler multilivello

(Ogni coda pesca nel modo più consono per la categoria)

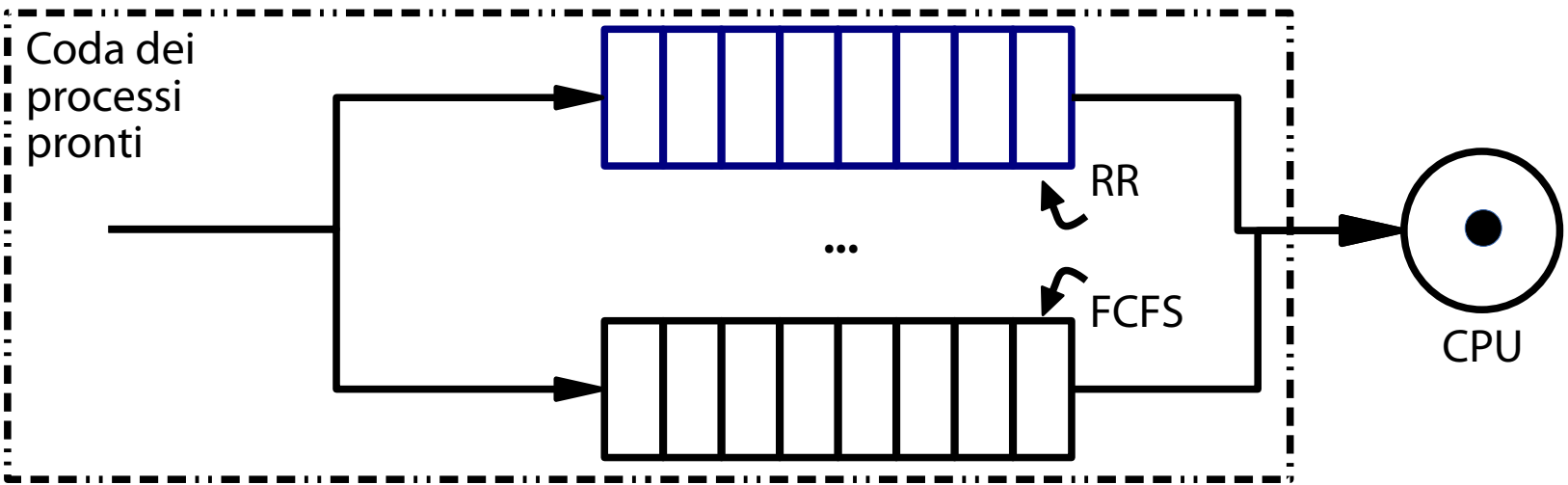
Ciascuna coda ha un proprio algoritmo di scheduling classico (ad esempio RR, FCFS).



Scheduler multilivello

(Prima si sceglie la coda...)

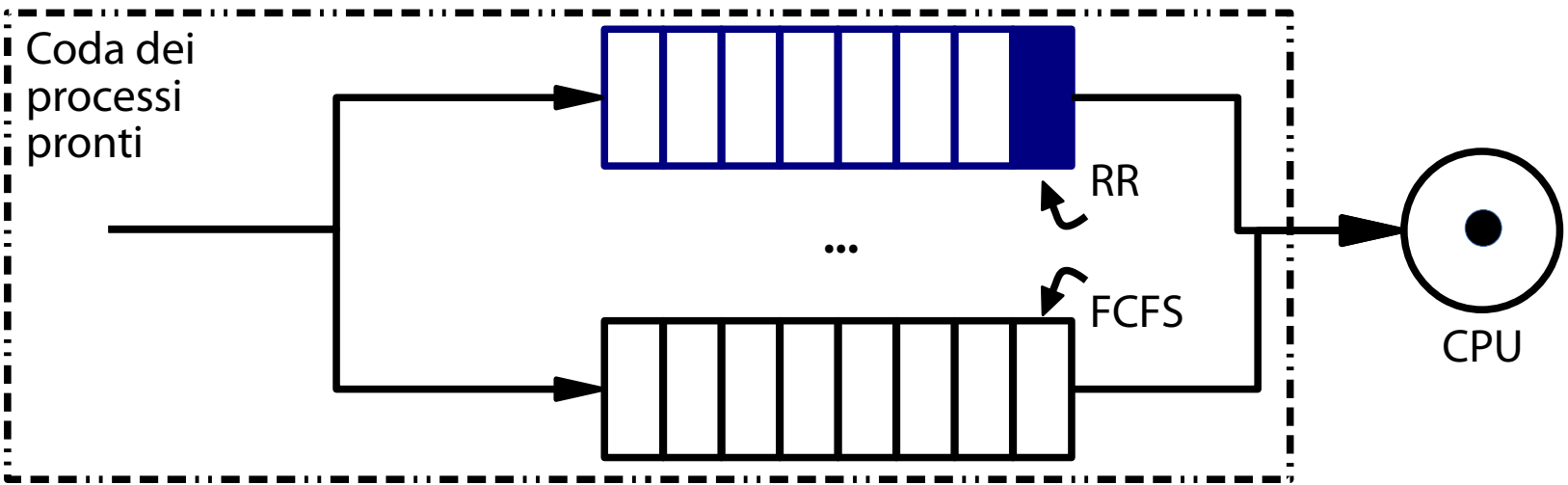
L'algoritmo di scheduling opera a due livelli.
Primo passo: sceglie la coda da cui pescare un processo.



Scheduler multilivello

(… e poi si sceglie la richiesta)

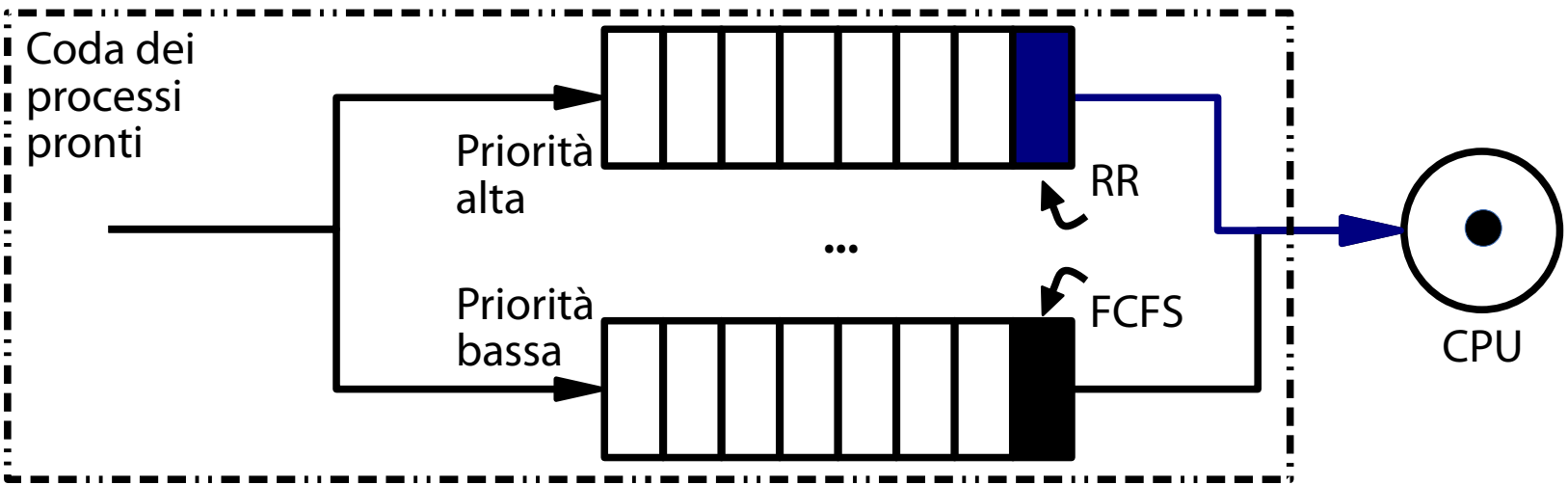
L'algoritmo di scheduling opera a due livelli.
Secondo passo: sceglie un processo dalla coda.



Scelta della coda

(Come avviene?)

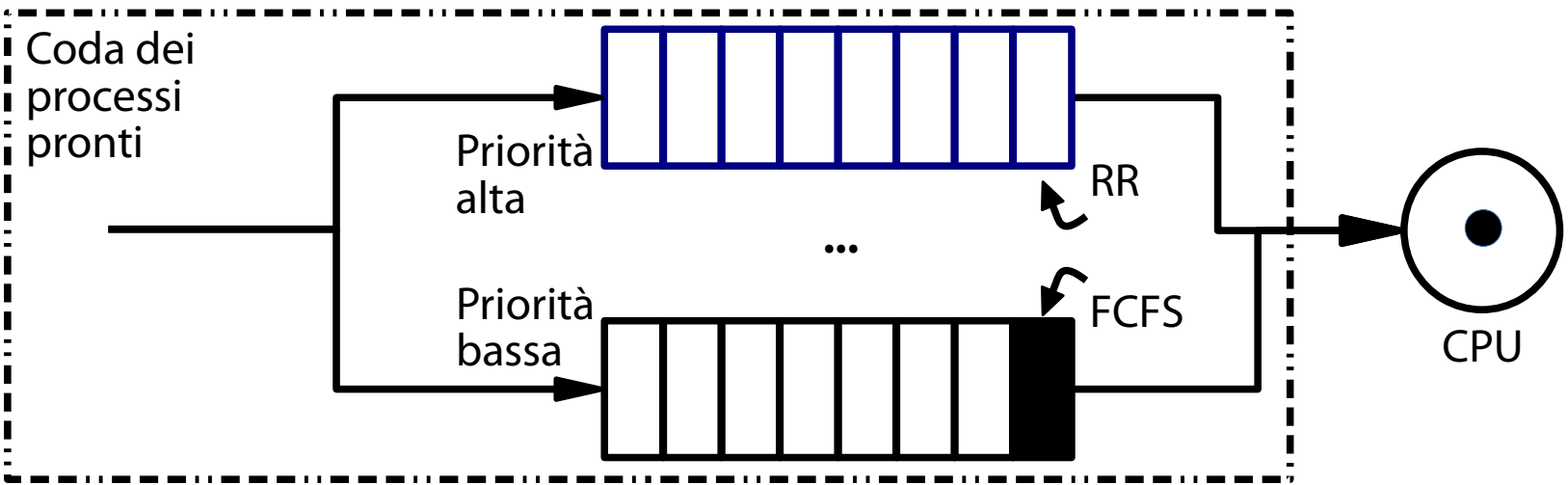
Si associano diversi livelli di priorità (fissi) alle code. Si sceglie la coda a priorità più alta in cui è presente un processo.



Scelta della coda

(Come avviene?)

Dopo l'esecuzione, il processo è inserito di nuovo nella stessa coda da cui proviene.



Parametri di progetto

(Da definire nel progetto di uno scheduler multilivello)

Numero di code.
Algoritmo di scheduling per ciascuna coda.
Criterio usato per scegliere la coda in cui inserire inizialmente un processo.

Un esempio concreto

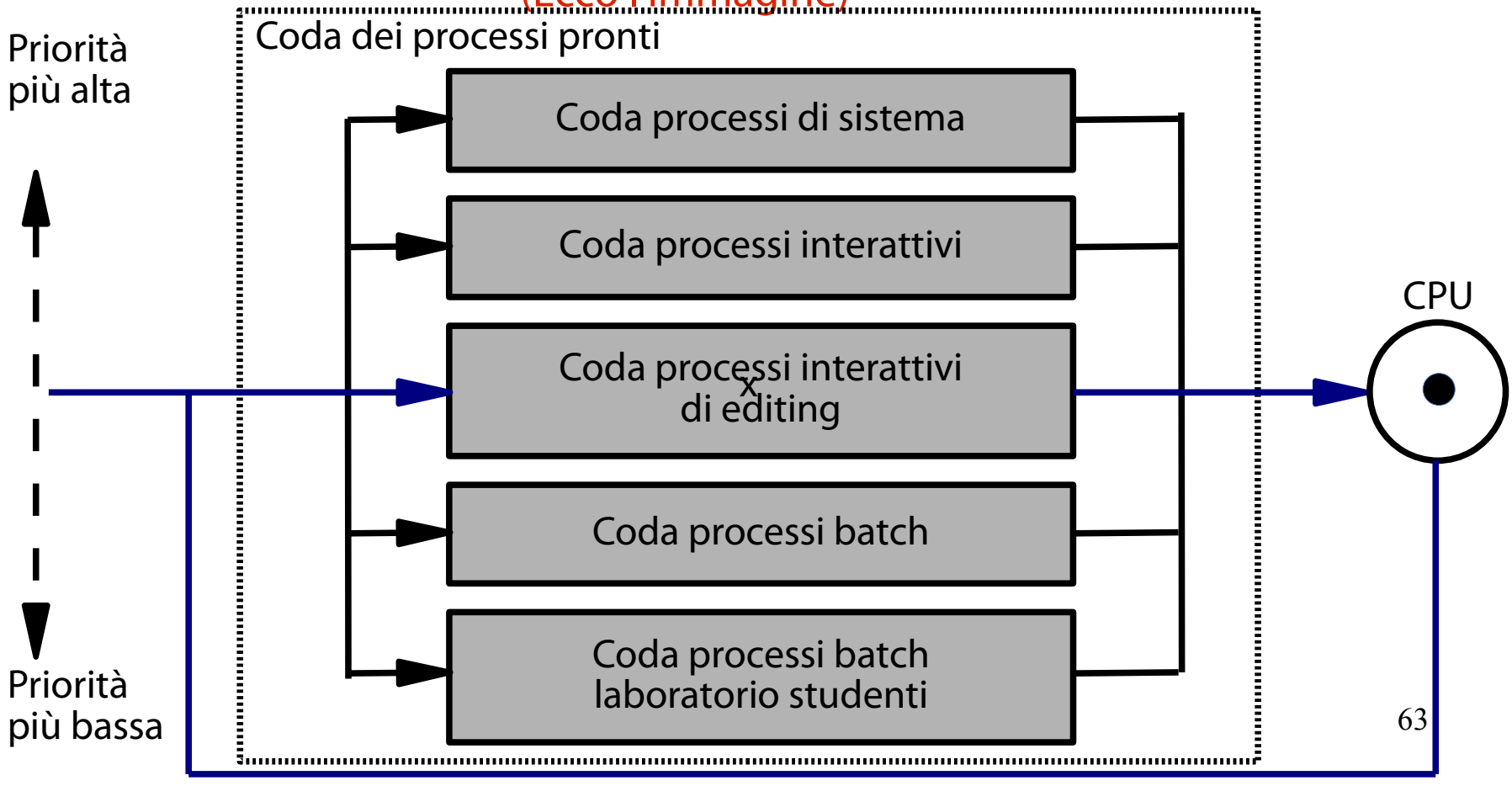
(Vale quanto un'immagine, ovvero 1000 parole)

Si suddivide la coda di pronto in cinque code distinte, con priorità decrescente in termini di importanza.

- Processi di sistema (RR).
- Processi interattivi (RR).
- Processi interattivi di editing (RR).
- Processi in background (FCFS).
- Processi batch degli studenti di laboratorio (FCFS).

La coda di pronto risultante

(Ecco l'immagine)



Scheduler multilivello: vantaggi

(Perché implementarlo?)

I processi importanti sono serviti per prima, essendo incanalati nelle code a priorità più alta.

→ Per tali processi, la latenza è molto bassa.

Scheduler multilivello: svantaggi

(Perché non implementarlo?)

I processi non importanti soffrono di starvation.

→ Per tali processi, non vi è garanzia alcuna sulla latenza.

→ È necessario integrare lo scheduler con un meccanismo di aging.

Se la natura del processo (CPU-bound, I/O-bound) è variabile nel tempo, lo scheduler multilivello non è efficace.

Scheduler multilivello con retroazione
(Feedback)

Detto anche **multilevel feedback queue**. È una variante dello scheduler multilivello in cui il processo può rientrare in una coda diversa da quella in cui si trovava precedentemente.

→ Adattamento dinamico della priorità del processo (eseguito in seguito ad un cambio di natura del processo).

→ Aging dei processi a bassa priorità.

Rationale del feedback

(Perché si implementa?)

La retroazione tende a raggruppare i processi con caratteristiche di CPU burstiness simili simili nelle stesse code.

- Processi con CPU burst lunghi si muovono verso le code a priorità più bassa.
- Processi con CPU burst brevi si muovono verso le code a priorità più alta.

La retroazione è tanto più efficace quanto meno frequente è il cambio di natura dei processi.

Parametri di progetto

(Da definire nel progetto di uno scheduler multilivello con retroazione)

Numero di code.
Algoritmo di scheduling per ciascuna coda.
Criterio usato per spostare un processo in una coda con priorità maggiore.
Criterio usato per spostare un processo in una coda con priorità minore.
Criterio usato per scegliere la coda in cui inserire inizialmente un processo.

Un esempio concreto

(Vale quanto un'immagine, ovvero 1000 parole)

Si usano tre code di scheduling di priorità 0, 1, 2.
Priorità 0: Round Robin con quanto 8
Priorità 1: Round Robin con quanto 16
Priorità 2: FCFS
La coda a priorità i ha sempre prelazione sulla coda a priorità j (i < j).
Un processo entra a priorità 0.
Se un processo non finisce entro il suo quanto assegnato, viene spostato nella coda successiva.
Se per 10 volte il processo non esaurisce il suo quanto di tempo, viene spostato nella coda precedente

Cosa ci si aspetta da questo scheduler?

(Ovvero: perché le regole astruse ora introdotte funzionano?)

I processi con CPU burst brevi (I/O bound, interattivi con il terminale) sono serviti molto rapidamente.

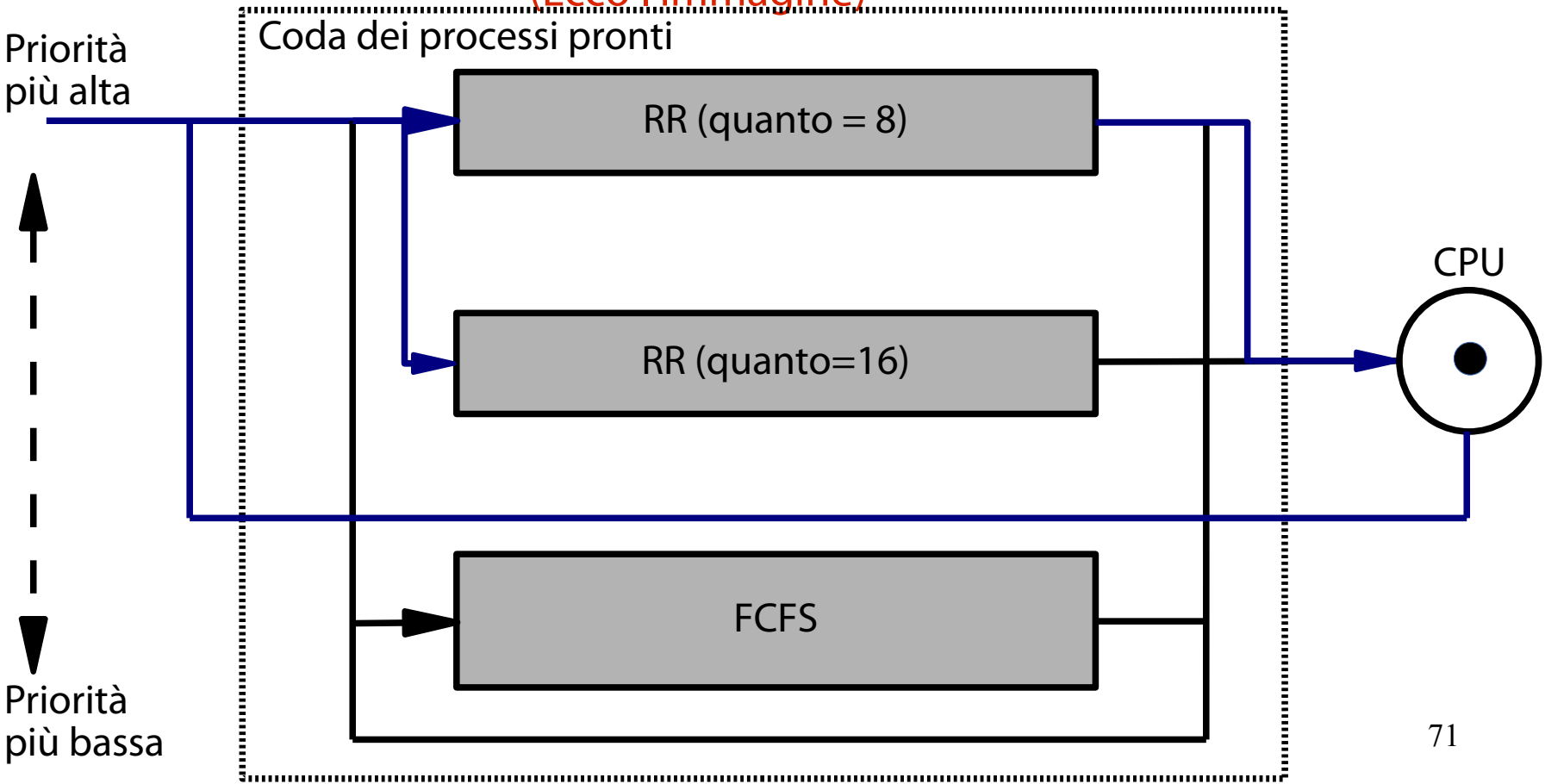
I processi con CPU burst un po' più lunghi sono serviti rapidamente.

I processi con CPU burst lunghi (batch) sono serviti solo se il sistema non è altrimenti impegnato.

Un sistema con CPU burst variabile si adatta lentamente alla sua coda ottimale.

La coda di pronto risultante

(Ecco l'immagine)



SCHEDULING IN SISTEMI SMP

Coda di pronto globali
(Semplice da implementare, performa male)

Coda globale: unica coda di pronto condivisa da tutti i processori.

Lo scheduler pesca dalla coda di pronto e sceglie un processore secondo un dato criterio.

Non scala benissimo con il numero di processori (aumentano le contese dovute agli accessi concorrenti).

Coda di pronto per CPU

(Più complicata da implementare, ma più CPU cache friendly)

Coda di pronto distribuita per CPU: ne esiste una per ogni processore della macchina.

- Lo scheduler può operare su più code allo stesso istante.
- Non esistono ritardi dovuti a misuso delle cache del processore.
- Soluzione preferita dai SO moderni su architetture SMP.

Problema: come bilanciare i processi sulle diverse code?

Predilezione

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

Predilezione (**CPU affinity**): lo scheduler dei processi cerca di eseguire il processo sempre sullo stesso processore.

Stesso processore → dati in CPU cache → prestazione.

Predilezione debole (**soft affinity**): lo scheduler rischedula il processo sempre sulla stessa CPU, a meno di un bilanciamento delle code di scheduling.

Implementata nello scheduler di Linux.

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

In GNU/Linux: **`schedtool -a cpu_id PID`**

Bilanciamento del carico

(Tutte le code dovrebbero avere lo stesso numero di processi)

In uno scheduler con code di pronto locali, la tecnica di predilezione rischia di sbilanciare fortemente il carico fra processori. Si rende necessario il **bilanciamento del carico** fra i diversi processori.

I processi sono spostati (**migrati**) da un processore all'altro, in modo tale che il carico sui processori sia grossomodo lo stesso.

Attenzione a bilanciare troppo spesso, però: **il bilanciamento annulla i benefici della predilezione**!

Migrazione del carico

(Ovvero come spostare processi da una coda ad un'altra)

Migrazione guidata (**push migration**): un processo dedicato controlla periodicamente la lunghezza delle code. In caso di sbilanciamento, sposta i processi in modo da bilanciare il carico.

Linux: bilanciamento controllato ed eseguito ogni 200ms.

Migrazione spontanea (**pull migration**): lo scheduler sottrae un PCB ad una coda sovraccarica.

Linux: invocata quando la coda si svuota