Scheduling della CPU

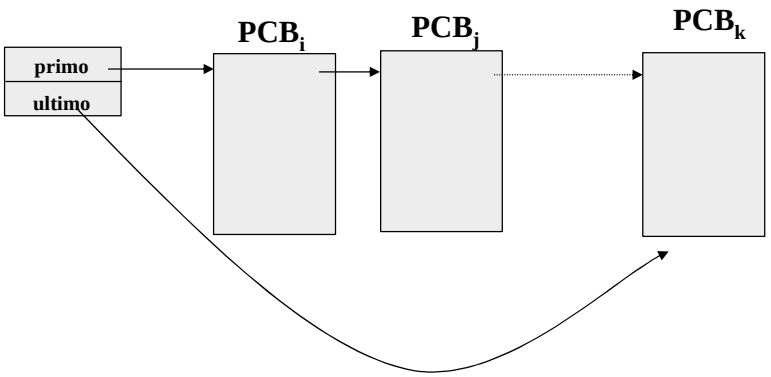
Scheduling della CPU

Obiettivo della multiprogrammazione: massimizzazione dell'utilizzo della CPU.

- **Scheduling della CPU:** commuta l'uso della CPU tra i vari processi.

- **Scheduler della CPU (a *breve termine*):** è quella parte del S.O. che seleziona dalla coda dei processi pronti il prossimo processo al quale assegnare l'uso della CPU.

Coda dei processi pronti (*ready queue*):



contiene i descrittori (process control block, PCB) dei processi pronti.

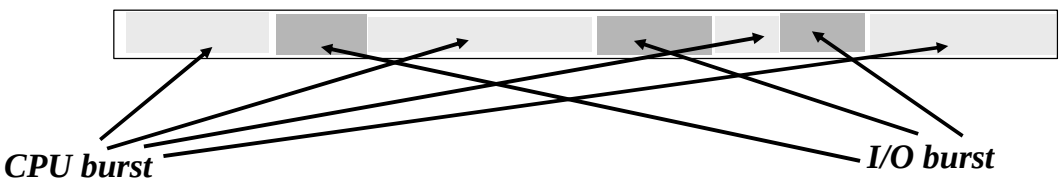
- la strategia di gestione della ready queue è realizzata mediante le **politiche** (algoritmi) di scheduling

Terminologia: CPU burst & I/O burst

Ogni processo alterna:

- **CPU burst**: fasi in cui viene impiegata soltanto la CPU senza I/O
- **I/O burst:** fasi in cui il processo effettua input/output da/verso una risorsa(dispositivo) del sistema

(burst = raffica)



- Quando un processo è in I/O burst, la CPU non viene utilizzata: in un sistema multiprogrammato, lo scheduler assegna la CPU a un nuovo processo.

Terminologia: Processi I/O bound & CPU bound

- A seconda delle caratteristiche dei programmi eseguiti dai processi, e` possibile classificare i processi in:
  - Processi ***I/O bound***: prevalenza di attivita` di I/O
    - ➔Molti CPU burst di *breve* durata, intervallati da I/O burst di *lunga* durata.
  - Processi ***CPU bound***: prevalenza di attivita` computazione:
    - ➔CPU burst di *lunga* durata, intervallati da pochi I/O burst di *breve* durata.

Terminologia: Pre-emption

Gli algoritmi di scheduling si possono classificare in due categorie:

- **senza prelazione** (***non pre-emptive***): la CPU rimane allocata al processo *running* finchè esso non si sospende volontariamente (ad esempio, per I/O), o non termina.
- **con prelazione** (***pre-emptive***): il processo *running* può essere prelazionato, cioè: il S.O. può sottrargli la CPU per assegnarla ad un nuovo processo.

➢ I sistemi a divisione di tempo (*time sharing*) hanno uno scheduling ***pre-emptive.***

Politiche & Meccanismi

Lo scheduler ***decide*** a quale processo assegnare la CPU.

- A seguito della decisione, viene attuato il **cambio di contesto** (*context-switch*).
- **Dispatcher**: è la parte del S.O. che realizza il cambio di contesto.

Scheduler = POLITICHE
Dispatcher = MECCANISMI

Criteri di Scheduling

Per analizzare e confrontare i diversi algoritmi di scheduling, vengono considerati alcuni parametri:

- **Utilizzo della CPU:**esprime la percentuale media di utilizzo della CPU nell'unita` di tempo.
- **Throughput** (del sistema): esprime il numero di processi completati nell'unità di tempo.
- **Tempo di Attesa** (di un processo): tempo totale trascorso nella ready queue.
- **Turnaround** (di un processo): è l'intervallo di tempo tra la sottomissione del job e il suo completamento.
- **Tempo di Risposta** (di un processo): intervallo di tempo tra la sottomissione e l'inizio della prima risposta (a differenza del turnaround, non dipende dalla velocità dei dispositivi di I/O)

Criteri di Scheduling

In generale:

- devono essere **massimizzati**:
  - **Utilizzo della CPU (**al massimo: 100%**)**
  - **Throughput**

- invece, devono essere **minimizzati**:
  - **Turnaround** (sistemi *batch*)
  - **Tempo di Attesa**
  - **Tempo di Risposta** (sistemi *interattivi*)

Criteri di Scheduling

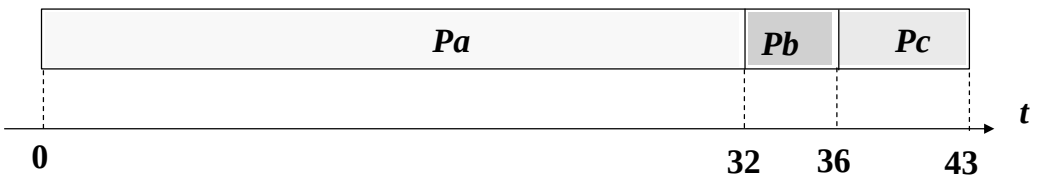
Non è possibile rispettare tutti i criteri contemporaneamente.

- A seconda del tipo di S.O., gli algoritmi di scheduling possono avere **diversi obiettivi**; tipicamente:
  - nei **sistemi *batch***:
    - **massimizzare throughput** e **minimizzare turnaround**
  - nei **sistemi *interattivi***:
    - **minimizzare il tempo medio di risposta** dei processi
    - **minimizzare il tempo di attesa**

Algoritmo di scheduling FCFS

- ***First-Come-First-Served***: la coda dei processi pronti viene gestita in modo FIFO:
  - i processi sono schedulati secondo **l'ordine di arrivo** nella coda
  - algoritmo **non pre-emptive**

Esempio: tre processi [Pa, Pb, Pc] (diagramma di ***Gantt***)

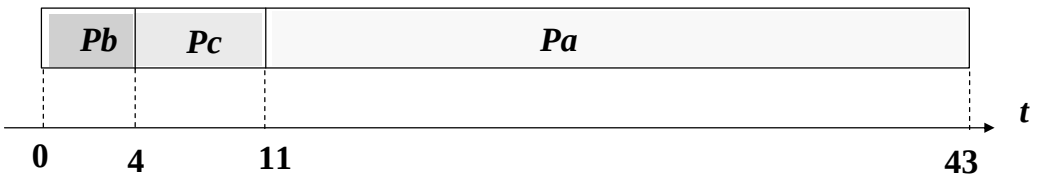


$$T_{\text{attesa medio}} = (0 + 32 + 36)/3 = 22{,}7$$

Algoritmo di scheduling FCFS

Esempio: se cambiassimo l'ordine di scheduling:

[Pb, Pc, Pa]



$$T_{\text{attesa medio}} = (0 + 4 + 11)/3 = 5$$

Problemi dell'algoritmo *FCFS*

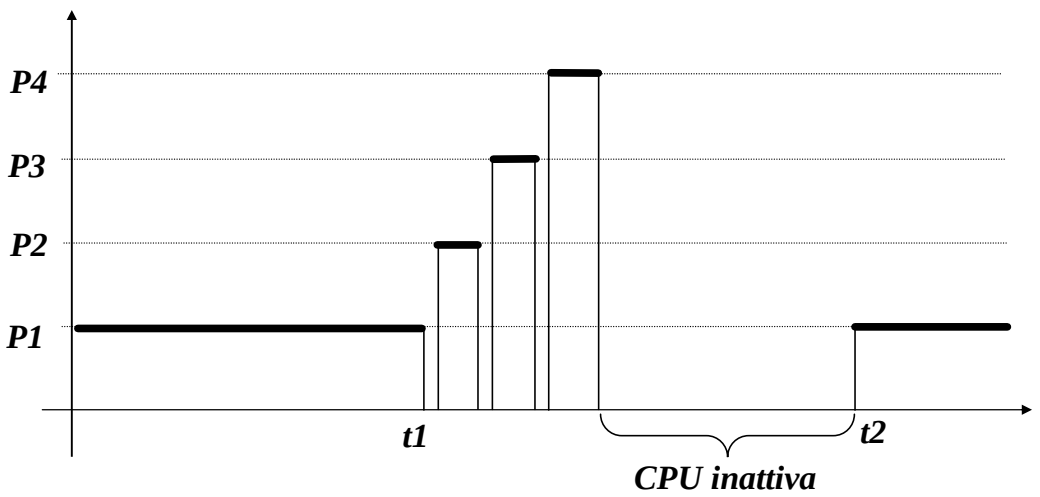
Non è possibile influire sull'ordine dei processi:

- **nel caso di processi in attesa dietro a processi con lunghi CPU burst (processi CPU bound), il tempo di attesa è alto.**
- **Possibilità di effetto *convoglio:*** se molti processi I/O bound seguono un processo CPU bound : **scarso grado di utilizzo della CPU.**

Algoritmo di scheduling FCFS: effetto convoglio

Esempio: [P1, P2, P3, P4]

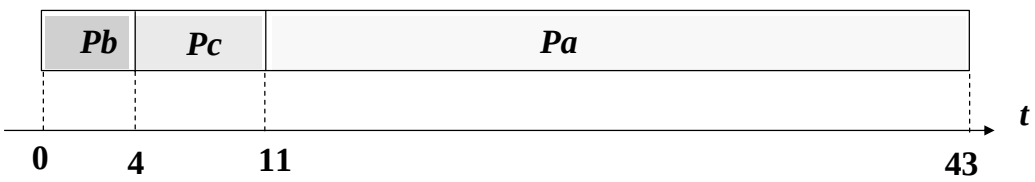
- P1 e` CPU bound; P2,P3,P4 sono I/O bound
- P1 effettua I/O nell'intervallo [t1,t2]



Algoritmo di scheduling SJF (*Shortest Job First*)

Per risolvere i problemi dell'algoritmo FCFS:

- per ogni processo nella ready queue, viene stimata la lunghezza del prossimo CPU-burst
- viene schedulato il processo con il CPU burst più piccolo (*Shortest Job First*)



- si può dimostrare che il tempo di attesa è **ottimale**

Algoritmo di scheduling SJF (*Shortest Job First*)

SJF può essere:

- **non pre-emptive**
- ***pre-emptive*: (*Shortest Remaining Time First, SRTF)*** se nella coda arriva un processo (Q) con CPU burst minore del CPU burst rimasto al processo running (P) ➔ ***pre-emption***: scambio tra P e Q.

Problema:

- è difficile stimare la lunghezza del prossimo CPU burst di un processo (di solito: uso del passato per predire il futuro)

Scheduling con Priorità

Ad ogni processo viene assegnata una priorità:

- lo scheduler seleziona il processo pronto con **priorità massima**
- processi con uguale priorità vengono trattati in modo FCFS

Priorità: possono essere definite

- **internamente**: il S.O. attribuisce ad ogni processo una priorità in base a politiche interne
- **esternamente**: criteri esterni al S.O (es: `nice` in Unix).

➢ Le priorità possono essere **costanti** o **variare** dinamicamente.

Scheduling con Priorità

Algoritmi di scheduling con priorità possono essere:

- **non-preemptive**
- **pre-emptive**: se arriva in coda un processo con priorità maggiore del processo running ⇨ ***pre-emption***

Esempio di algoritmo con priorità: *SJF*

- per ogni processo, la priorità è **$1/CPU_{burst}$**!
- la priorità è variabile

Scheduling con priorità

Problema: ***starvation*** dei processi.

Starvation: si verifica quando uno o più processi di priorità bassa vengono lasciati **indefinitamente** nella coda dei processi pronti, perchè vi è sempre almeno un processo pronto di priorità più alta.

Soluzione: invecchiamento (***aging***) dei processi:

- **ad esempio:**
  - **la priorità cresce dinamicamente** con il tempo di attesa del processo.
  - **la priorità decresce** con il tempo di CPU gia` utilizzato

Algoritmo di Scheduling *Round Robin*

É tipicamente usato in sistemi *Time Sharing*:

- La ready queue viene gestita come una coda **FIFO** circolare (v. FCFS)
- ad ogni processo viene allocata la CPU per un **intervallo di tempo costante** Δt (*time slice* o, *quanto di tempo*):
  - il processo usa la CPU per Δt (oppure si blocca prima)
  - allo scadere del quanto di tempo: **prelazione** della CPU e re-inserimento in coda

➢ l'algoritmo RR può essere visto come un'estensione di FCFS con pre-emption periodica.

Round Robin

- L’obiettivo principale è la minimizzazione del tempo di risposta:
  - adeguato per sistemi interattivi
- Tutti i processi sono trattati allo stesso modo:
  - non c’è starvation

Round Robin

Problemi:

- dimensionamento del quanto di tempo
  - Δt ***piccolo*** **(**ma non troppo: D**t>>T**$_{\textbf{context switch}}$)
    - ✓ tempi di risposta ridotti, ma
    - ✓ alta frequenza di context switch => overhead
  - Δt ***grande***:
    - ✓ overhead di context switch ridotto, ma
    - ✓ tempi di risposta più alti
- trattamento *equo* dei processi: processi di S.O. e processi utente sono trattati allo stesso modo:
  - possibilità di degrado delle prestazioni del S.O.

Approcci misti

- Nei sistemi operativi reali, spesso si combinano diversi algoritmi di scheduling.

Esempio: *Multiple Level Feedback Queues*

- più code, ognuna associata a un tipo di job diverso (batch, interactive, CPU-bound, etc.)
- ogni coda ha una diversa priorità: scheduling delle code con priorità
- ogni coda viene gestita con scheduling FCFS o Round Robin
- i processi possono muoversi da una coda all'altra, in base alla loro storia:
  - passaggio **da priorità bassa ad alta**: processi in attesa da ***molto*** tempo (feedback *positivo*)
  - passaggio **da priorità alta a bassa**: processi che hanno già utilizzato ***molto*** tempo di CPU (feedback **negativo**)

Scheduling in Unix

Obiettivo: privilegiare i processi interattivi

Scheduling MLFQ:

- **più livelli di priorità** (circa 160): più grande è il valore, più bassa è la priorità.
- Viene definito un valore di riferimento **`pzero`**:
  - Priorità ³ **`pzero`**: processi di utente ordinari.
  - Priorità < **`pzero`**: processi di sistema (ad es. Esecuzione di system call), non possono essere interrotti da segnali (**`kill`**).
- Ad ogni livello è associata una coda, gestita con ***Round Robin*** **(**quanto di tempo: 0,1 s)

Scheduling in Unix

- Aggiornamento dinamico delle priorità: ad ogni secondo viene ricalcolata la priorità di ogni processo.
- La priorità di un processo decresce al crescere del tempo di CPU già utilizzato
  - feedback negativo.
  - di solito, processi interattivi usano poco la CPU: in questo modo vengono favoriti.
- L'utente può influire sulla priorità: comando **`nice`** **(soltanto per decrescere la priorita`)**