09 - Scheduling della CPU

Scheduling della CPU

Obiettivo della multiprogrammazione: massimizzazione dell'utilizzo CPU

- **Scheduling della CPU**: commuta l'uso della CPU tra i vari processi
- **Scheduler della CPU (a breve termine)**: e' quella parte del SO che seleziona dalla coda dei processi pronti il prissimo processo al quale assegnare l'uso della CPU

Coda dei processi pronti (ready queue):

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

Strategia di gestione della ready queue e' realizzata mediante **politiche** (algoritmi) di scheduling

Terminologia: CPU burst & I/O burst

Ogni processo alterna:

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

--> quando un processo e' in I/O burst, la CPU non viene utilizzata: in un sistema multiprogrammato, short-term 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

- **I/O bound**: prevalenza di attivita' I/O
  - molti CPU burst di breve durata, intervallati da I/O burst di lunga durata
- **CPU bound**: prevalenza di attivita' di 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**): CPU rimane allocata al processo running finche' esso non si sospende volontariamente o non termina

- con prelazione (**pre-emptive**): processo running puo' essere prelazionato, cioe' SO puo' sottrargli CPU per assegnarla ad un nuovo processo

--> i sistemi a divisione di tempo hanno sempre uno scheduling **pre-emptive**

Politiche e meccanismi

- Scheduler: decide quale processo assegnare la CPU
- A seguito della decisione, viene attuato il cambio di contesto (context-switch)
- Dispatcher: e' la parte di SO 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 indicatori di performance:

- **Utilizzo della** CPU: percentuale media di utilizzo CPU nell'unita' di tempo
- **Throughput** (del sistema): numero di processi completati nell'unita' di tempo
- **Tempo di attesa** (di un processo): tempo totale trascorso nella ready queue
- **Turnaround** (di un processo): 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 velocita' dei dispositivi di I/O)

In generale:

- devono essere **massimizzati**
  - utilizzo della CPU
  - Throughput
- Invece, devono essere **minimizzati**
- Turnaround (sistema batch)
- Tempo di attesa
- Tempo di risposta (sistemi interattivi)

Non e' possibile ottimizzare tutti i criteri contemporaneamente

A seconda del tipo di SO, gli algoritmi di scheduling possono avere diversi obiettivi

- nei sistemi batch:
  - massimizzare throughput
  - minimizzare turnaround

- nei sistemi interattivi
    - minimizzare il tempo medio di risposta dei processi
    - minimizzare il tempo di attesa

Algoritmo di scheduling FCFS

Firts-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** (nella versione "pura" iniziale)

Problemi dell'algoritmo FCFS

Non e' possibile influire sull'ordine dei processi:

- nel caso di processi in attesa dietro ai processi lunghi con CPU burst (processi CPU bound), il tempo di attesa e' alto
- Possibilita' di effetto convoglio, se molti processi I/O bound seguono un processo CPU bound: scarso grado di utilizzo della CPU

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 piu' corto (Shortest Job First)

SJF puo' essere:

- non pre-emptive
- pre-emptive: (Shortest Remaining Time First, SRTF) se nella coda arriva un processo (Q) con CPU burst minore della CPU burst rimasto al processo running (P) --> pre-emption

Problema: e' difficile stimare la lunghezza del prossimo CPU-burst di un processo (di solito, uso del passato per predirre il futuro

Stimare la lunghezza di CPU burst

Unica cosa ragionevole: stimare probabilisticamente la lunghezza in dipendenza dai precedenti CPU burst di quel processo.

Possibilita' molto usata: exponential averaging

Algoritmo di scheduling Round Robin

E' tipicamente usato in sistemi time sharing:

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

Round Robin (RR)

Obiettivo principale e' la minimizzazione del tempo di risorsa (adeguato per sistemi interattivi). Tutti i processi sono trattati allo stesso modo (assenza di starvation)

Problemi:

- dimensione del quanto di tempo
    - Dt piccolo (ma non troppo piccolo: Dt >> $T_{context\ switch}$) tempi di risposta ridoti, ma alta frequenza di context switch
    - Dt grande, overhead di context switch ridotto, ma tempi di risposta piu' alti
- trattamento equo dei processi
    - possibilita' di degrado delle prestazioni del SO, che deve avere maggiore importanza dei processi utente

Scheduling con priorita'

Ad ogni processo viene assegnata una priorita':

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

Priorita' possono essere definite

- **internamente**: SO attribuisce ad ogni processo una priorita' in base a politiche interne
- **esternamente**: criteri esterni al SO (es: nice in UNIX)

--> Le priorita' possono essere costanti o variare dinamicamente

In ogni istante e' in esecuzione il processo pronto **a priorita' massima** (algoritmi preemptive e non preemptive)

Problema: starvation dei processi
Starvation: si verifica quando uno o piu' processi di priorita' bassa vengono lasciati indefinitamente nella coda dei processi pronti, perche' vi e' sempre almeno un processo di priorita' piu' alta

Soluzione: invecchiamento (aging) dei processi, ad esempio

- la priorita' cresce dinamicamente con il tempo di attesa del processo
- la priorita' decresce al crescere del tempo di CPU gia' utilizzato

Approcci misti

Nei SO reali, spesso si combinano diversi algoritmi di scheduling

Esempio: **Multiple Level Feedback Queue**

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

Esempio di Multi Level Feedback Queue

3 code

- Q_0 - RR con time quantum=8ms
- Q_1 - RR con time quantum=16ms
- Q_2 - FCFS

Scheduling

- Un processo nuovo entra in Q_0; quando acquisisce la CPU ha 8ms per utilizzarla; se non termina nel quanto di tempo viene spostato in Q_1
- In Q_1 il processo e' servito ancora RR e riceve 16ms di CPU; se non termina nel quanto di tempo, viene spostato in Q_2

--> Priorita' elevata a processi con breve uso di CPU

Scheduling in UNIX (BSD 4.3)

Obiettivo: **privilegiare i processi interattivi**

Scheduling MLFQ: Multilevel Feedback Queue Scheduling

- piu' livelli di priorita' (circa 160): piu' grande e' il valore, piu' bassa e' la priorita'
- Viene definito un valore di riferimento pzero
  - Priorita' >= pzero: processi di utente ordinari
  - Priorita' < pzero: processi di sistema (ad es. esecuzione di system call), non possono essere interrotti da segnali (kill)
- Ad ogni livello e' associata una coda, gestita Round Robin (quanto di tempo: 100ms)
- Aggiornamento dinamico delle priorita': ad ogni secondo viene ricalcolata la priorita' di ogni processo
- La priorita' di un processo decresce al crescere del tempo di CPU gia' utilizzato
  - feedback negativo
  - di solito, processi interattivi usano poco la CPU: in questo modo vengono favoriti
- L'utente puo' influire sulla priorita': comando **nice** (ovviamente soltanto per decrescere la priorita')
- Quando un processo esegue una system call, puo' venire bloccato (es.: richiesta di uso del disco)
- A verificarsi dell'evento atteso, il processo diventa pronto e viene inserito nel relativo livello di priorità
  (priorita' negativa = priorita' alta)
- Favorisce i **processi interattivi** (attesa per un terminale) e i processi che eseguono I/O

Linux scheduling (da v2.5)

Due algoritmi: **time sharing** e **real-time**

- **Time-sharing**
  - Con priorita' dinamiche, basato su crediti - processi con piu' crediti schedulati prima
  - Crediti vengono decrementati in base a timer
  - Quando crediti = 0, il processo viene deschedulato
  - Si rialza il credito di tutti quando tutti i processi arrivano a credito=0
- **Real-time**
  - Soft real-time con priorita' statiche
  - Conforme a POSIX.1b compliant - due classi
    - FCFS e RR all'interno della stessa priorita'
    - proesso a priorita' maggiore esegue sempre per primo

Scheduling dei thread Java

Java Virtual Machine (JVM) usa **scheduling con prelazione e basato su priorita'**

- FCFS tra thread con stessa priorita'

JVM mette in stato di running un thread quando:

1. il thread che sta usando la CPU esce dallo stato Runnable
2. un thread a priorita' piu' alta entra nello stato Runnable

*NB: le Java Specifications non indicano se i thread hanno un quanto di tempo oppure no. JVM puo' adottare una propria politica di scheduling oppure delegare lo scheduling dei thread al sottostante sistema operativo

Time-Slicing

Siccome JVM non garantisce time-slicing, andrebbe usato il metodo yield() per trasferire il controllo ad altro thread di uguale priorita':

```
while (true) {
    // perform CPU-intensive task
    ...
    thread.yield();
}
```

Si possono assegnare valori di priorita' tramite metodo `setPriority()`.