1 Lezione del 07-10-25

1.1 Nucleo

Il **nucleo** o *kernel* è il cuore di un S/O, il componente che ha il compito di realizzare l'astrazione della *CPU virtuale*. Nel caso di sistemi monoprocessore, vogliamo dividere il tempo fra i processi per dargli l'illusione di essere gli unici in esecuzione sulla macchina.

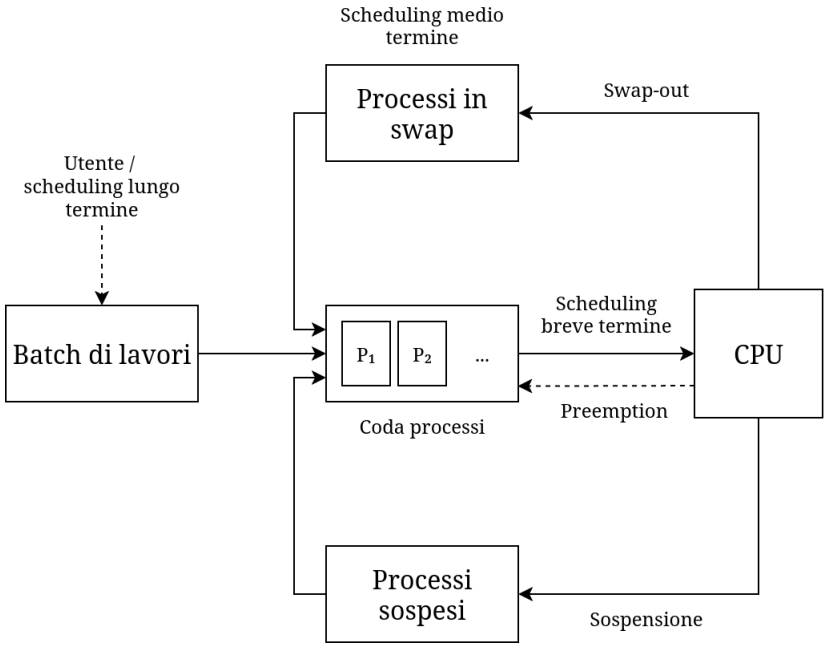
1.1.1 Scheduling

Lo scheduling è l'attività secondo la quale il sistema operativo effettua delle scelte fra quali processi caricare in memoria centrale e a quali assegnare la CPU.

Ci sono 3 diverse attività di scheduling:

- **Breve** termine: lo scheduling propriamente detto, il processo attraverso cui si assegna la CPU. Può essere *preemptive* e *non preemptive* (con o senza diritto di revoca). Solitamente viene invocato molto frequentemente (millisecondi);
- **Medio** termine (*swapping*): il trasferimento temporaneo in memoria secondaria dei processi. Si usa quando la memoria centrale dispone di memoria minore della somma di quella richiesta dai vari processi. Viene invocato più di rado (secondi, minuti);
- **Lungo** termine: la scelta di quali processi caricare dalla memoria secondaria in memoria centrale. Rappresenta un componente importante dei sistemi *batch* multiprogrammati, oggi come oggi quindi sui *server* e meno sulle macchine personali;

Vediamo quindi una schematica che mostra dove queste attività di scheduling avvengono nell'architettura vista:



I processi possono in genere classificarsi in:

- Processi vincolati da **I/O**: passano più tempo a fare I/O burst piuttosto che CPU burst (che sono tanti e piccoli);

- Processi vincolati da **CPU**: passano più tempo a fare calcoli, hanno pochi e lunghi CPU burst.

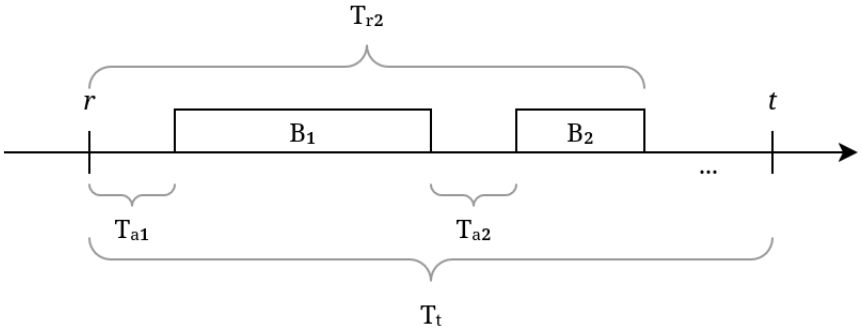
1.2 Algoritmi di scheduling

Gli algoritmi di scheduling che vedremo saranno:

- **FCFS** (*First Come First Served*): è *non prioritario* e *non preemptive*, consiste nel assegnare la CPU sempre al primo processo arrivato;
- **SJF** (*Shortest Job First*): è *prioritario* e *non preemptive*, consiste nell'assegnare la CPU al processo più breve;
- **SRTF** (*Shortest Remaining Time First*): è *prioritario* e *preemptive*, rappresenta sostanzialmente la versione con revoca del precedente;
- **RR** (*Round Robin*): è *non prioritario* e *preemptive*: si basa sull'assegnare quanti temporali ugualmente ad ogni processo;
- Schedulazione **su base prioritaria**: introdurremo qui l'idea di *priorità* per ogni processo;
- Schedulazione **a code multiple**: prevediamo più code, che possiamo distinguere usando gli algoritmi sopra descritti, o come vedremo sarà conveniente, assegnando una *priorità* ad ogni cosa;
- Schedulazione di sistemi **in tempo reale**. Questi sono algoritmi che devono assicurare la terminazione deterministicad ei processi. In questo vedremo gli algoritmi:
  - **RM** (*Rate Monotonic*);
  - **EDF** (*Earliest Deadline First*).

1.2.1 Valutazione degli algoritmi di scheduling

Iniziamo a vedere alcune metriche per la valutazione degli algoritmi di scheduling:



- **Utilizzazione** della CPU o *efficienza*, cioè definito ΔB_i il tempo di burst e T il quanto di tempo totale, vogliamo un efficienza E:

$$E = \frac{\sum \Delta B_i}{T} < 1$$

il più possibile vicina a 1;

- Tempo medio di **completamento** (o tempo di *turnaround*), cioè il tempo che passa prima che il processo possa completare la sua operazione (terminando). Prendiamo l'istante in cui il processo entra in coda pronti come r (da *richiesto*) e quello in cui termina termina come t (da *termina*). Il tempo di completamento T_t sarà ovviamente:
$$T_t = t - r$$
- Tempo di **risposta**, valutato dall'istante in cui un processo entra in coda pronti r all'istante in cui risponde per volta, cioè termina un CPU burst. Purtroppo, non tutto il tempo di completamento T_c è dedicato al processo, ma questo viene eseguito, come sappiamo, in più burst (diciamo B_1, B_2, ...). Il tempo di turnaround T_t sarà allora il tempo trascorso fra r e la fine di un burst B_i, cioè:
$$T_{ri} = \text{end}(B_i) - r$$
- Tempo di **attesa**, cioè la somma dei tempi di attesa posti fra i vari burst:
$$T_a = \sum t_{\alpha i}$$
- **Produttività** (o frequenza di *througput*), definita come il numero medio di processi completati nell'unità di tempo, cioè l'inverso del tempo medio di completamento:
$$P = \frac{1}{T_t}$$
- Rispetto dei **vincoli temporali**, utile principalmente negli algoritmi di scheduling in *tempo reale*.

Chiameremo poi O_v l'**overhead** associato all'esecuzione dello scheduler. Ricordiamo che in ogni caso in questa fase stiamo parlando di scheduling a breve termine.

1.2.2 Algoritmo FCFS

Nell'algoritmo **FCFS** (*First Come First Served*) assegnamo la CPU al primo processo in coda pronti. Sostanzialmente, trattiamo la coda pronti come una coda **FIFO** (*First In, First Out*). Questo lo rende non prioritario e non preemptive.

Quello che otteniamo è un efficienza teorica pari a $E \sim 1$ (c'è un piccolo overhead $O_v \sim 0$ dato dal cambio di contesto), ma generalmente prestazioni piuttosto limitate. Questo è dovuto al fatto che i tempi di attesa (e di conseguenza di completamento) dei processi sono completamente aleatori, e non si fa alcuna scelta informata mirata a minimizzarli.

Abbiamo quindi che l'algoritmo è utile per sistemi batch, dove l'unica cosa che ci interessa è uso massimo della CPU (che ci assicura), ma largamente da evitare per sistemi interattivi, e sopratutto per sistemi real-time. Questo è vouto all'aleatorietà legata al momento della richiesta dei processi, che rende impossibile rispondere celeremente o fare qualsiasi tipo di promessa sul tempo di turnaround.

1.2.3 Algoritmo SJF

L'algoritmo **SJF** (*Shortest Job First*) implementa una **priorità statica**: si fa l'ipotesi di conoscere il **tempo di CPU** utilizzato da ogni processo, e assegnare priorità maggiori a processi con tempo di esecuzione minore. Di contro, non è preemptive, cioè una volta assegnata la CPU non può revocarla. Su come il S/O conosce il tempo di esecuzione non facciamo per adesso ipotesi.

Facciamo una nota sulla priorità statica: ad ogni chiamata dello scheduler questo può sapere solo i tempi di esecuzione dei processi attualmente in esecuzione, cioè si potrebbe mandare in esecuzione un processo con tempo di esecuzione maggiore quando ne entra in coda pronti ne entra uno con tempo minore. In questo caso, per la natura *non preemptive* dell'algoritmo, bisogna lasciare che questo esegua prima di mettere il nuovo arrivato in esecuzione.

Adoperando questo algoritmo si minimizza (nel senso matematicamente ottimo) il tempo di attesa medio dei processi, in quanto si cerca di arrivare il prima possibile al processo sucessivo (svolgendo adesso il più veloce). Uno svantaggio sarà chiaramente che i processi che dimostrano tempi di esecuzione lunghi verranno eseguiti sempre per ultimi.

1.2.4 Algoritmo SRTF

Abbiamo introdotto l'algoritmo **SRTF** (*Shortest Remaining Time First*) come una versione *preemptive* del SJF (in questo rimane comunque prioritario).

Visto che è preemptive, viene eseguito *ogni volta* che cambiano le condizioni di scelta (non soltanto quando la CPU è libera, come nel caso dei non preemptive, ma ogni che un nuovo processo entra in esecuzione). Può per questo motivo eseguire senza innescare cambi di contesto.

Dovremmo adattare la nostra ipotesi di conoscenza del tempo di CPU ad un'ipotesi di conoscenza del **tempo rimanente** per ogni processo: in questo caso se il processo appena entrato ha tempo rimanente minore di quello del processo attualmente in esecuzione, conviene sfruttare la preemption. Di nuovo, per adesso non facciamo assunzioni su come ricaviamo tale euristica.

L'SRTF migliora la risposta in tempo reale del SJF, permettendo una riduzione sia dei tempi di turnaround che dei tempi di attesa medi in caso di richieste di esecuzione di processi non ottimali.

Un problema del SRTF, come avevamo visto nel SJF, e la **process starvation**: in genere, negli algoritmi in base prioritaria, si rischia che i processi a priorità minore (in questo caso quelli con tempo rimanente maggiore) non vengano mai serviti e rimangano a lungo in coda pronti, rendendo il sistema meno responsivo.

1.2.5 Stima dei tempi in SJF e SRTF

Chiariamo la questione di come si possono fare previsioni informate sul **tempo di esecuzione** (in *SJF*) e il **tempo rimanente** (in *SRTF*).

Un'approccio, preso ad esempio il caso del **tempo di esecuzione**, è quello di usare la tecnica della **esponenziale**. Si fa una stima iniziale s_i del tempo di burst t_i esimo. Preso un parametro a con $0 < a < 1$, si aggiorna ad ogni terminazione del processo (quindi facendo delle *osservazioni* per ogni esecuzione del processo) la stima come:

$$s_{n+1} = at_n + (1 - a)s_n$$

Presi $a \sim 0$ si ha che le stime deviano malvolentieri da quella iniziale, mentre con $a \sim 1$ si ha che le stime sono molto volubili rispetto alle osservazioni fatte.

Modeli statistici più complessi possono dare previsioni più accurate, sempre tenendo conto del fatto che lo scheduler deve eseguire con overhead $O_v \sim 0$, o almeno il più piccolo possibile.

Una volta noto il *tempo di esecuzione*, il **tempo rimanente** si può stimare considerando il tempo che il processo ha impiegato finora e sottraendolo dal tempo di esecuzione totale (se non rinunciando al tener conto se tale tempo è stato usato in CPU o I/O burst, purtroppo un'euristica è un'euristica).