09-05-2023

Per gli algoritmi di scheduling con priorità di distinguono algoritmi preemptive e non preemptive.

Per differenziare i processi interattivi dai processi in background si potrebbe agire nel seguente modo:
Scheduling con coda multipla: la coda dei processi pronti si suddivide in:
- coda dei **processi interattivi** -> algoritmo roundrobin
- coda dei **processi in background** -> algoritmo FCFS
- Si possono applicare scheduling differenti sulle due code (con priorità su quelli interattivi)

Si può anche giocare sul timeslice:

- 80% del tempo dedicato ai processi interattivi
- 20% del tempo dedicato ai processi in background

Gestione dei processi interattivi ed eventuali algoritmi

SRTN permette di superare il limite di SJF ma se quest'ultimo conosce a priori tutti i tempi, risulta essere un algoritmo ottimale.

- Si usa SJF nella gestione dei processi interattivi: si basa su un approccio `attesa della richiesta -> esecuzione della richiesta`
- I comandi più brevi vengono eseguiti all'inizio ma comunque si deve sapere quale, fra tutti i processi, è il processo più breve.

Il problema è l'identificazione della durata del successivo processo. Si potrebbe fare con una **stima**. E si stima sulla base delle esecuzioni precedenti svolte (***burst precedenti***).

Shortest Process Next

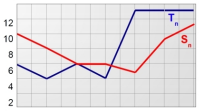
- Tenta di simulare SJF in processi interattivi
- Il processo successivo da eseguire viene stimato tramite la media esponenziale delle lunghezze (durate) delle precedenti esecuzioni di occupazione della CPU.

Si definiscono:

- T_n -> lunghezza dell'n-esimo CPU-burst
- S_{n+1} tempo richiesto per il prossimo CPU-burst
- a -> valore fisso fra 0 e 1 e **pesa**, nella **stima**, le **un ricordo di quello che è successo** nel *breve/medio/lungo* tempo.

Lastima del processo n+1 è:

$$S_{n+1} = S_n(1-a) + T_n a$$



- se $a = 0$ vuol dire che la storia recente non ha alcun effetto e quindi ottiene $S_{n+1} = S_n$

- se $a = 1$ vuol dire che il recente CPU-burst è quello cruciale che determina la stima del prossimo ma non viene considerata la cronologia di quello che è successo in precedenza e quindi si ottiene $S_{n+1} = T_n$
- Si da un peso ponderato ad $a = \frac{1}{2}$ fra la cronologia passata e l'evento presente.
- Il problema è il punto di partenza S_0. Si può rendere costante

> *In generale: le scelte per il bilanciamento del carico che si prendono sono basate sull'esperieza passata e presente dei fatti*

Funzionamento SPN

- Ci si trova nel caso n dove $S_n = 10$ mentre $T_n = 6$
- Ne stimo 8 ma in reatà ne ha fatti solo 4

T_n	6	4	6	4	13	13	13
S_n	10	8	6	6	5	9	11

- Usualmente **a = 1/2** e questo algoritmo stima il tempo del **CPU-burst del successivo processo** e in questo caso si considera una **cronologia di media lunghezza**

> *Se il tempo effettivo di un processo T_n è più alta della stima, quest'ultima si incrementa (ovviamente)*

Algoritmo di Scheduling Garantito

E' un approccio diverso e **stabilisce una percentuale di uso della CPU** che deve essere **RISPETTATA**.

- Va **calcolata la percentuale di uso di CPU spettante per ogni processo**: considera quanto tempo è stato utilizzato dal timeslice è stato usato e, in base a questa quantità, determina quanta quantità di CPU un processo ha diritto ad utilizzare. Viene scelto il processo con rapporto minore
- Inoltre, il rapporto deve essere minore rispetto al consumo
- fare promesse reali e mantenerle utenti connessi avranno 1/n della potenza CPU (idem processi)
- tenere traccia quanta CPU ricevuta
- (Tc / n) = tempo dalla creazione diviso n
- l'algoritmo esegue il processo con rapporto minore

Algoritmo di Scheduling a Lotteria

- **Ogni processo ha un biglietto assegnato** e quando si deve decidere chi deve occupare la CPU, l'algoritmo **estrae** un **numero casuale** e il processo che va in esecuzione è il processo che ha il numero precedentemente assegnato
- Una volta che il **processo estratto va in esecuzione**, viene **consumato** il numero assegnatoli
- In caso di **esigenze diverse** si possono **assegnare biglietti extra** ad un processo e vuol dire **aumentare la probabilità** che il processo venga scelto dopo l'estrazione.
- I processi possono **cooperare**: se `A` si trova nella sezione critica, `B` non può accedere. `A` prende i suoi e li cede a `B` in modo da aumentare la probabilità che `B` venga estratto e viceversa verrà fatto da `B`

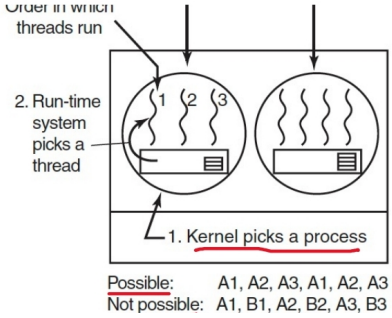
Algoritmo di Scheduling Fair-Share

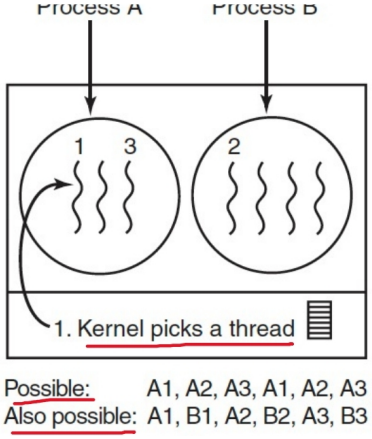
- *Se l'utente A esegue 9 processi mentre l'utente l'utente B ne esegue 1, allora la CPU è dedicata quasi interamente ad A e questa non è una corretta distribuzione della CPU.*
- Questo algoritmo **tiene in considerazione i proprietari dei processi** e determina **un'equo uso della CPU fra gli utenti del sistema** a prescindere dal numero di processi che ogni singolo utente esegue
- Si assegna una **proporzione di uso della CPU** ai vari utenti **in base al numero di processi da eseguire**

> *Classicamente questi algoritmi non vengono usato ma si usa, come detto prima, l'algoritmo ROUND-ROBIN in caso di parità di priorità (stessa classe di priorità)*

Scheduling dei Thread

I thread possno essere implementati in vari e i pro dell'uno corrispondono ai contro dell'altro e viceversa:

- **A livello Utente** (*il **kernel conosce i thread** e se uno di essi si blocca, allora viene bloccato l'intero processo. Il cambio fra thread è veloce*):
  - 

  - Lo **scheduling** è **PERSONALIZZATO** e sceglie di **assegnare** un ***timeslice*** da assegnare **al processo stesso**
  - Il **processo stesso** usa un **sistema di scheduling** con il quale fa l'assegnazione dei tempi ai singoli thread (*distribuendo il timeslice assegnato al processo*)
  - *Interrupt di clock* fra thread **non sono possibili** quindi i thread lavorano in modalità **non preemptive** e ognuno di loro *lasciano la CPU quando finoscono interamente il tempo assegnato* (*e questo rappresenta un aspetto negativo e un limite*).
  - Di conseguenza *potrebbe capitare* che un thread **consuma l'intero timeslice** assegnato al processo
- **A livello Kernel** (*il kernel riconosce i thread quindi può stoppare il singolo thread se necessario. Un thread di un processo potrebbe essere cambiato con un thread di un altro processo*):
  - 

  - Il **timeslice è assegnato al thread** dal kernel
  - Adesso lo **scheduling dei thread è gestito dal kernel**

- I **thread vengono considerati tutti uguali** e ciò potrebbe **compromettere le performance** e quindi si potrebbe perdere tempo nel **cambio di contesto**
- Piuttosto che **sostituire un thread con un altro di un altro processo**, si sfrutta l'informazione del processo e si cerca di **forzare**, se possibile, lo **scambio** fra thread **con un altro thread dello stesso processo** (*per evitare di scambiare thread fra processi diversi*)

Differenze fra Thread Utente e Thread Kernel (scheduling)

Le prestazioni sono dfiferenti:

- Il **cambio di contesto** è più lento
- A livello kernel il thread bloccato non sospende il processo -> non c'è livello utente
- Il grosso vantaggio dei Thread Utente è il permesso di usare uno ***SCHEDULING PERSONALIZZATO*** (*quindi dipendente dal tipo di applicazione*)

Di norma si usa un sistema **MISTO**.

Gestione scheduling in un sistema multiprocessore

Esistono diversi modi per gestire lo scheduling di un multiprocessore:

1. ***ASIMMETRICO***: si hanno diversi processori di cui **UNO** si occupa dello scheduling, elaborazione I/O, smistamento dei processo -> ***MASTER SERVER***. Gli altri, ***SLAVE*** eseguono i processi.
2. ***SIMMETRICO***: Ogni processore esegue un determinato processo e ogni processore fa la stessa identica cosa. Esiste una coda e un algoritmo di scheduling che smista i processi fra i processori. La ***coda dei processi pronti*** è **UNICA** (il kernel gestisce lo scheduling. si deve evitare che 2 processori si aggiudicano la CPU) per tutti i processori oppure **DIVERSA** per ogni processore (si cerca di *sfruttare al massimo la CACHE dello **stesso processore** per migliorare i **tempi di esecuzione***

Politiche di scheduling

- Si cerca di eseguire i processi negli stessi processori per permettere di usare i dati della cache di quel processore e quindi evitando di trasportare dati dalla memoria principale.
- Presenza o assenza di predilezione per i processori. Un processore cerca di prediligere un particolare processo e può essere:
  - ***DEBOLE***: il sistema **tenta di eseguire un processo** in quel determinato processore dove è stato eseguito in precedenza ma **non lo garantisce**
  - ***FORTE***: Il sistema **forza e garantisce** che un processo vada in esecuzione in un **determinato processore** (*o insieme di processori*)

> *In generale il sistema SIMMETRICO viene usato su Windows/Linux.*

Bilanciamento del carico

Lo scheduling deve eseguire il **bilancio fra processi** e processori che hanno al proprio interno delle code (visto che non c'è un'*unica coda*).

- Il carico di lavoro deve essere ugualmente distribuito
- si applica una pseudo-migrazione:
  - un processore che ha una coda piena, allora un processo migra verso un altra coda più vuota di un altro processore.

Si distinguono:

- ***migrazione guidata*** (push): un'attività (*demone*) che periodicamente controlla il carico di lavoro di ciascun processore. Quando vi è uno squilibrio di attività e carichi di lavoro, allora si ridistrbuisce in maniera equilibrata i carichi di lavoro ed effettua la migrazione su un processore la quale coda di processi è più vuota.
- ***migrazione spontanea*** (pull): la coda di un processore diventa vuota. Spontaneamente questa coda viene riempita

> *Spesso questi 2 approcci sono usati insieme in parallelo* (*UNIX*)

Il bilanciamento del carico di lavoro e la predilezione del processore sono due politiche contrastanti e quindi bisogna trovare un compromesso in base alle attività. Non esiste una regola dove si predilige una di queste politiche ma in generale si cerca di eseguire un mix fra le politiche (*ove possibile*)

Cosa usano i nostri Sistemi Operativi

Elementi comuni: thread, SMP, gestione priorità, predilezione per i processi I/O bounded
Windows:

- scheduler basato su code di priorità con varie euristiche per migliorare il servizio dei processi interattivi e in particolare di foregrount
- evita il problema dell'inversione delle priorità
  Linux:
- Scheduling basato su task (generalizzazione di processi e thread) usato con **alberi rosso-neri** ordinati in base al tempo di esecuzione:
  - Si garantisce un corretto bilanciamento grazie alla **proprietà dei *RB-Tree***
- Moderno scheduler garantito e si basa su *Completely Fair Scheduler* (CFS)
- Non considera processo/thread o altro ma sono tutte ***TASK***
  MacOS:
- scheduler basato su code di priorità (*Mach Scheduler*)