03 - Scheduling della CPU

Introduzione

Per accedere ad una risorsa, formiamo una coda dei processi in attesa di accedere a quella risorsa, e per garantire l'attesa limitata quella coda deve essere gestita in base all'ordine di arrivo; in questo modo siamo sicuri che l'attesa è limitata.

Aver stabilito che la coda di attesa a quella risorsa è gestita in base all'ordine di arrivo vuol dire aver scelto un algoritmo di scheduling.
Gli algoritmi di scheduling sono gli algoritmi con cui andiamo a gestire le code di attesa.

L'esecuzione di un processo è caratterizzata da momenti in cui utilizza la CPU e da altri in cui il sistema utilizza un dispositivo di I/O (in questo tempo il processo rimane sospeso).

- *CPU Burst* : tempo in cui il processo passa ad eseguire istruzioni della CPU
- *I/O Burst* : tempo in cui il processo passa in attesa del completamento di un'operazione di I/O

Perché ci sono molti CPU Burst di breve durata e pochi di lunga durata ?
Un processo che chiede molte operazioni di I/O (interruzioni software) tenderà ad avere dei CPU Burst più brevi.
Un processo che richiede molte operazioni di I/O è un processo altamente interattivo; i processi che lavorano in background tendono ad avere CPU Burst di lunga durata.
Studiare la distribuzione dei CPU Burst ci permette di capire la natura di questi processi.

I processi possono essere:
-
CPU-bound, producono poche sequenze di operazioni della CPU molto lunghe. Impiegano più tempo nelle elaborazioni che per l'I/O. Pochi CPU-burst ma di lunga durata.
-
I/O bound, producono molte sequenze di operazioni della CPU di breve durata. Impiegano più tempo per l'I/O che per le elaborazioni, hanno molti CPU-burst ma di breve durata.

Scheduler

Stato del processo

Introduzione

Lo scheduler della CPU vede l'elenco di tutti i processi che sono in stato di *ready* (gestisce una coda dei processi *ready*, la coda è una coda dei PCB), ne sceglie uno e passa il puntatore di quel PCB ad un altro pezzo del Kernel chiamato "**dispatcher**".

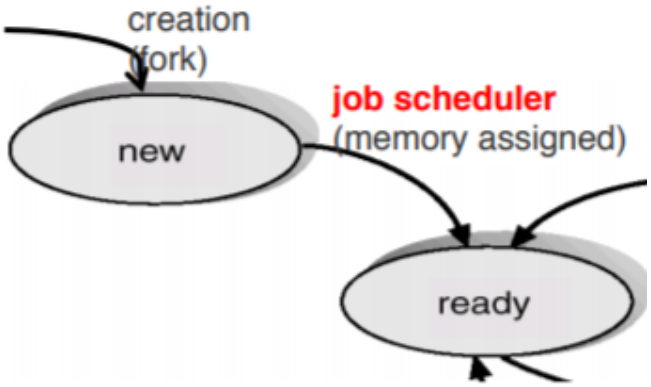
Vi sono diversi tipi di scheduler che si distinguono in base alla frequenza con cui vengono invocati gli algoritmi di scheduling e in base alle operazioni che svolgono.

- **Scheduler a lungo termine** (*job scheduler*) : esso viene invocato di rado e si occupa di selezionare i processi in stato di *new* e caricarli in memoria (passano in stato di *ready*). Esso deve massimizzare il grado di multiprogrammazione senza far degradare le prestazioni del sistema. Viene invocato per allocare nuova memoria o per de-allocarla quando un processo si conclude.
- **Scheduler a medio termine** : si occupa di selezionare tra i processi "scaricati" dalla memoria (*swapped out*) quelli da caricare (*swap in*) nuovamente in memoria. Viene invocato leggermente più di frequente rispetto a quello a lungo termine ed anche lui deve evitare di degradare le prestazioni del sistema. Si occupa di togliere all'occorrenza risorse al fine di non peggiorare le prestazione del sistema. Va quindi a leggere in memoria centrale e se necessario sospende altri processi al fine di mantenere buone prestazioni. Opera a giochi fatti, quindi quando il processo è già stato creato.
- **Scheduler a breve termine** (*CPU scheduler*) : seleziona tra i processi presenti nella *ready queue* il processo a cui allocare la CPU. È invocato di frequente, quindi deve essere molto veloce (millisecondi), e inoltre deve massimizzare l'utilizzo della CPU.

Job Scheduler

Controlla il grado di multiprogrammazione: se è possibile caricare nuovi processi in memoria senza provocare il thrashing, seleziona i processi che devono essere caricati in memoria.

I descrittori di tali processi (PCB) vengono inseriti nella *ready queue*. Cosi lo stato dei processi passa da *new* a *ready*.

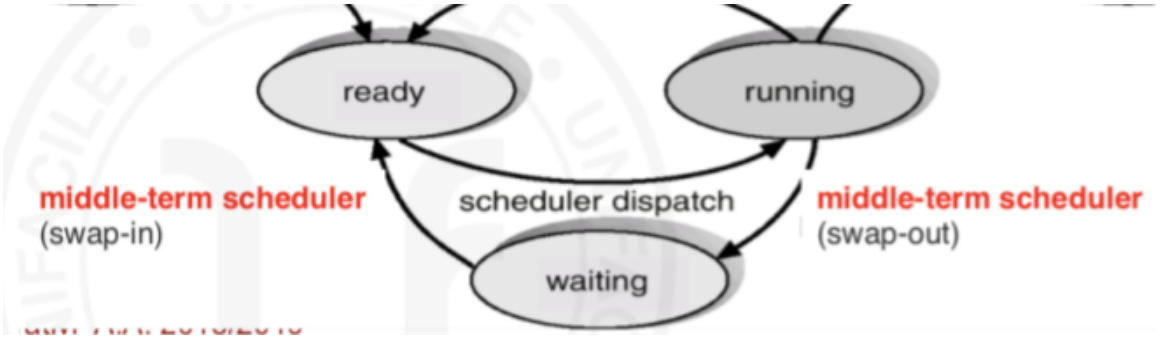


Middle-term Scheduler

Controlla il grado di multiprogrammazione ed il rischio di degrado delle prestazioni del sistema (thrashing).

Se il grado di multiprogrammazione è troppo elevato (il sistema rischia il *thrashing*) seleziona quindi i processi da scaricare (*swap out* → il contenuto di un determinato processo in memoria centrale viene salvato sulla memoria di livello più basso nella gerarchia) e il loro stato passa da *run* a *wait* (e gli vengono tolte le risorse).

Se il grado di multiprogrammazione è sufficientemente basso, seleziona i processi swapped-out che devono essere caricati in memoria (*swap in* → il contenuto di un processo passa da una memoria di livello più basso alla memoria centrale) e il loro stato passa da *wait* a *ready*. I PCB di tali processi vengono inseriti nella ready queue.

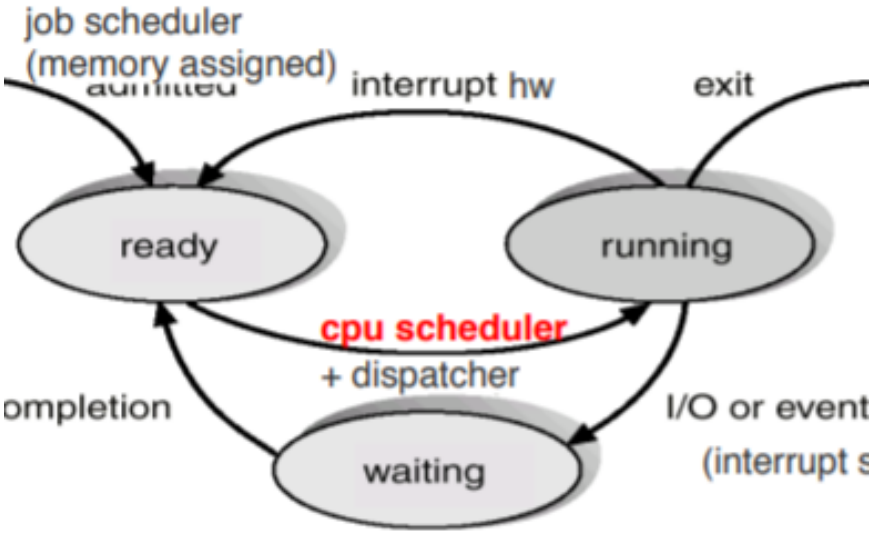


CPU Scheduler

Seleziona un processo tra quelli presenti in memoria che sono pronti per l'esecuzione e alloca la CPU a tale processo.

Le decisioni riguardanti lo scheduling della CPU si possono prendere nelle seguenti circostanze:
1) un processo passa dallo stato di *running* a *waiting*.
2) un processo passa dallo stato di *running* a *ready*.
3) un processo passa dallo stato di *waiting* a *ready*.
4) un processo termina.

Lo scheduling si divide in due categorie :
-
senza diritto di prelazione (*non-preemptive*) → quando si assegna la CPU ad un processo, questo rimane in possesso della CPU fino a quando non la rilascia spontaneamente (*FIFO*, *SJF senza prelazione*).
-
con diritto di prelazione (*preemptive*) → quando il SO può obbligare un processo a rilasciare la CPU anche se il processo ne ha ancora bisogno (*RR*, *SJF con prelazione*).



Dispatcher

Il *Dispacher* assegna il controllo della CPU al processo selezionato dal CPU scheduler (scheduler a breve termine).

Questo comporta le seguenti operazioni:
-
commutazione di contesto (context switching)
-
passaggio da modalità kernel a modalità utente
-
salto alla locazione di memoria appropriata per far ripartire il programma utente.

Questo comporta però un dispendio di tempo (latenza di Dispatch) necessario a fermare un processo e a far partire un altro.

Quando un processo termina l'utilizzo della CPU il processo successivo non parte subito. Parte la routine di gestione della chiamata di sistema che ha messo il processo precedente in stato di attesa, dopodiché parte lo scheduler, poi il dispatcher; quando il dispatcher ha terminato il suo lavoro inizia il processo successivo. Quindi il nuovo processo inizia all'istante in cui termina quello precedente + la durata del context switch Δ .

Abbiamo trascurato la durata dei context switch (Δ) perché :

1) partiamo dal requisito che questo Δ deve essere piccolo, trascurabile rispetto alla durata media dei CPU Burst (thrashing).

2) vogliamo concentrarci su cosa succede ai processi, decidendo di considerare trascurabile la durata Δ; vogliamo fare un ragionamento sulle durate dei CPU Burst dei nostri processi e come questi possono influenzare i

parametri che ci siamo chiesti prima.

3) dal punto di vista degli esercizi è molto più facile trascurare la durata dei context switch (considerata pari a 0), consapevoli che è una approssimazione grossolana di quello che succede nella realtà.

CPU Scheduling nei Sistemi Monoprocessore

Criteri di Scheduling Monoprocessore

Valgono anche per lo scheduling multiprocessore con l'aggiunta di altri.

Diversi algoritmi di scheduling della CPU hanno proprietà differenti. Per il confronto tra gli algoritmi sono stati suggeriti alcuni criteri:

- Utilizzo della CPU: deve essere più attiva possibile.
-
Produttività: misura il numero dei processi completati nell'unità di tempo.
-
Tempo di completamento: è l'intervallo di tempo che intercorre tra la sottomissione del processo e il completamento dell'esecuzione.
-
Tempo di attesa: è la somma degli intervalli d'attesa passati nella coda dei processi pronti. (tempo totale in cui un processo è rimasto nella coda di *ready*).
-
Tempo di risposta: è il tempo che intercorre tra la sottomissione di una richiesta e la prima risposta prodotta (non l'output)

FCFS (First-Come First-Served)

I processi sono serviti in ordine di arrivo.

Finché un processo non termina la sua esecuzione, nessuno è in grado di togliergli l'utilizzo della CPU (algoritmo senza prelazione).

Il tempo medio di attesa si riduce se i processi che arrivano sono ordinati per tempo di utilizzo della CPU crescente (prima i processi più brevi e poi quelli più lunghi).

SJF (Shortest-Job-First)

Algoritmo per brevità, **la CPU viene assegnata al processo col CPU Burst più breve (che si trova in coda).**

Ci sono due possibili realizzazioni:

- *Senza prelazione*: il processo mantiene il possesso della CPU fino a quando non completa il CPU burst attuale. L'interruzione hardware arriva ogni volta che arriva un nuovo processo in coda; in questo caso però riparte il processo che era in esecuzione. Terminata la sua esecuzione lo scheduler andrà a scegliere, tra i processi in coda, quello con il CPU burst minore.
- *Con prelazione*: assegnata la CPU ad un processo, se durante la sua esecuzione dovesse arrivare in coda un altro processo con un CPU burst minore, tolgo la CPU al processo in esecuzione e la assegno al processo con minore durata. In questo caso viene anche chiamato **SRTF (Shortest-Remaining-Time-First).** Ad ogni interruzione hardware (all'arrivo di ogni nuovo processo) verranno esaminati nuovamente tutti i CPU burst, per poi assegnare la CPU al processo con CPU burst minore. Il CPU burst lo stimo andando a guardare come si è comportato nel passato utilizzando la media esponenziale.

SJF è ottimale in quanto minimizza il tempo medio di attesa.

Lunghezza del CPU Burst Successivo

- Può essere solo stimata.
- Si può utilizzare la media esponenziale dei CPU burst precedenti.

– t_n = lunghezza effettiva dell'n-esimo CPU burst
– τ_n = lunghezza stimata dell'n-esimo CPU burst
– α = peso, dove $0 \leq \alpha \leq 1$
– c = stima della durata del primo CPU burst (condizione al contorno)

$$\begin{cases} \tau_{n+1} = \alpha\, t_n + (1-\alpha)\, \tau_n \\ \tau_1 = c \end{cases}$$

- α =0
  –
  $\tau_{n+1} = \tau_n = \ldots = \tau_1$
  – La storia recente del processo non conta (algoritmo con priorità statica).
- α =1
  –

$\tau_{n+1} = t_n$
– Conta solo l'ultimo CPU burst.

Scheduling per Priorità

Ad ogni processo viene associata una priorità (un numero intero). La CPU viene allocata al processo con priorità più alta (numero piccolo = alta priorità).

Anche in questo caso si distinguono i casi con e senza prelazione. Un SJF è uno scheduling per priorità dove la priorità è la lunghezza del CPU burst successivo.

I problemi che hanno gli algoritmi per priorità riguardano la **starvation**: processi con bassa priorità potrebbero non ricevere mai la CPU.

La soluzione adottata a questo problema è l'**aging**: con il passare del tempo nella coda di ready, la priorità del processo viene aumentata.

RR (Round-Robin)

Implementa il TIME SHARING, ovvero **a rotazione ogni processo riceve la CPU per una piccola unità di tempo** (quanto di tempo = q).

Allo scadere del quanto, il processo è prelazionato ed aggiunto alla coda dei processi ready. La coda dei processi è gestita mediante coda FIFO circolare.

Se ci sono n processi nella coda di ready e il quanto di tempo è q allora ogni processo attenderà al più
$(n-1) * q$ unità di tempo.

Le prestazioni dipendono da q:

- q molto grande ⇒ RR diventa FCFS
- q molto piccola ⇒ nessun processo potrà accedere alla CPU (q deve essere molto maggiore di Δ = durata del context switch)

Tipicamente ha un tempo medio di completamento maggiore di SJF, ma ha un tempo di risposta migliore.

MFQ (Multilevel Feedback Queue)

L'algoritmo Multilevel Queue suddivide la coda di ready in diverse code distinte. Ogni coda adotta un proprio algoritmo di scheduling.

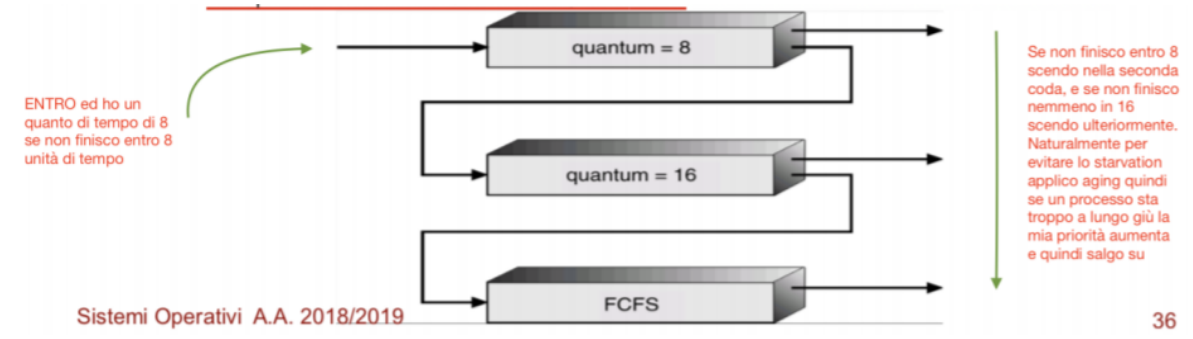
È inoltre necessario avere uno scheduling tra le code .

I processi in background sono quelli in cui l'utente può interagire con la macchina mentre quei processi sono in esecuzione.
I processi foreground invece sono quelli che prevedono l'interazione con l'utente, quindi non posso eseguire altri processi mentre questi ultimi sono in esecuzione.
Quindi divido le code in processi background e processi foreground.

Un processo può passare da una coda ad un'altra: ad esempio un processo che usa troppo la CPU si può spostare in una coda con priorità minore; analogamente un processo che attende da troppo tempo si può spostare in una coda con priorità maggiore (ovvero questo è un modo per implementare l'aging).

Come funziona praticamente questa tecnica di scheduling della CPU?

Un processo entra nella coda Q_0, quando ottiene la CPU riceve n millisecondi, se non finisce entro quel lasso di tempo, viene spostato in un'altra coda con priorità minore. E cosi si procede fino al suo completamento o a una sospensione. Naturalmente se un processo finisce in fondo e rimane troppo a lungo vale il discorso di aging che lo riporta su.



Possiamo ipotizzare queste n code con una struttura piramidale, ovvero la priorità scende a mano a mano che scendo giù dalla piramide.

CPU Scheduling nei Sistemi Multiprocessore

Introduzione

Asymmetric multiprocessing

Nel momento in cui abbiamo un solo processore, dobbiamo usare quello sia per le operazioni del kernel, sia per i nostri processi.
Nel caso ne avessimo di più, possiamo attribuirne uno per il kernel, ed uno per in nostri processi; questo significa che il kernel è sempre in esecuzione, dato che ha sempre una CPU dedicata.

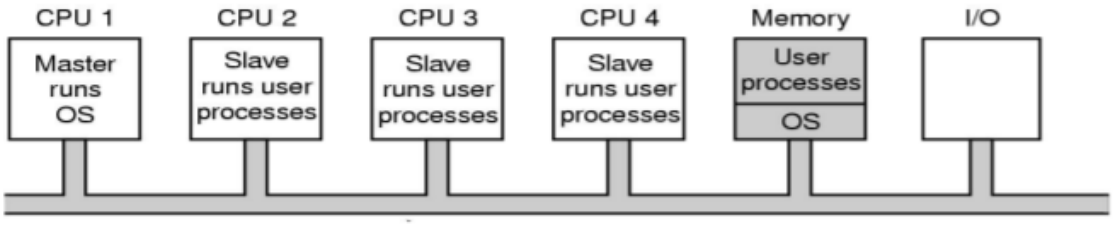
Si tratta della **multielaborazione asimmetrica** che riduce la necessità di condividere dati grazie all'accesso di un solo processore alle strutture dati del sistema. Solo un processore (Master) accede alle strutture dati di sistema.

Si dice asimmetrica perché una CPU è per il kernel, mentre le altre sono per i processi.

- Vantaggi: mantiene un buon bilanciamento del carico (tra le CPU Slave).

  Il kernel può monitorare costantemente se ci sono anomalie, o se c'è una CPU che è particolarmente succube di lavoro. Interviene immediatamente perché ha già la CPU a sua disposizione.

- Svantaggi: ogni system call/interrupt deve essere diretto verso la CPU Master (perché è l'unica ad eseguire il codice del SO in questo schema); quindi se il numero di CPU é alto, il Master può diventare un collo di bottiglia.



Symmetric multiprocessing (SMP)

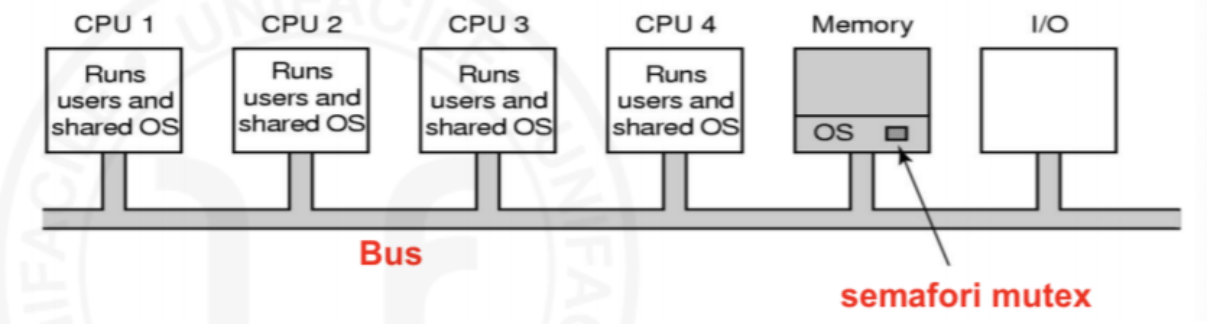
Ogni CPU esegue il codice del SO, quindi non c'è più la distinzione tra master e slave.

Si chiama **multielaborazione simmetrica** perché ogni CPU può essere usata sia per il kernel, sia per i processi.

Si evita così il collo di bottiglia; bisogna avere però più cura della memoria condivisa, in quanto più porzioni di un processo del SO potrebbe essere

eseguiti su più processori contemporaneamente.

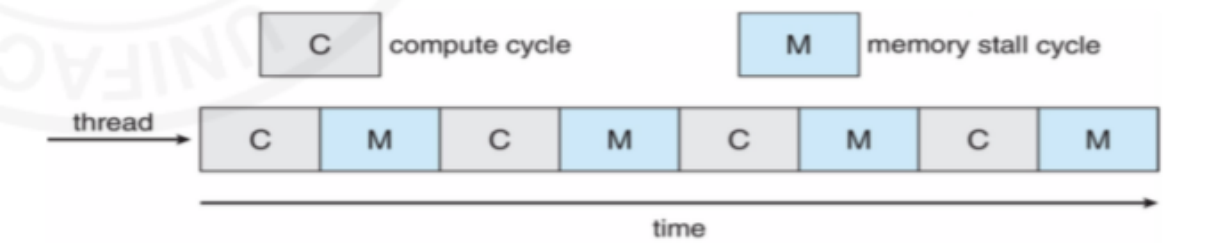
Potrebbe esserci il problema che il kernel è in esecuzione su diverse CPU. L'accesso alle strutture del kernel deve avvenire in modalità mutualmente esclusiva. Dobbiamo controllare l'accesso con un semaforo, in modo che se il kernel è utilizzato da qualche processore, un altro processore potrà eseguire un processo. Dobbiamo bloccare anche il bus, altrimenti ci sarà un blocco causato da collisioni.
Se ad un job assegno un processore, tutti i thread li faccio passare per quel processore.



Multicore Processors

Le architetture multicore permettono la presenza di thread multipli per ogni core.

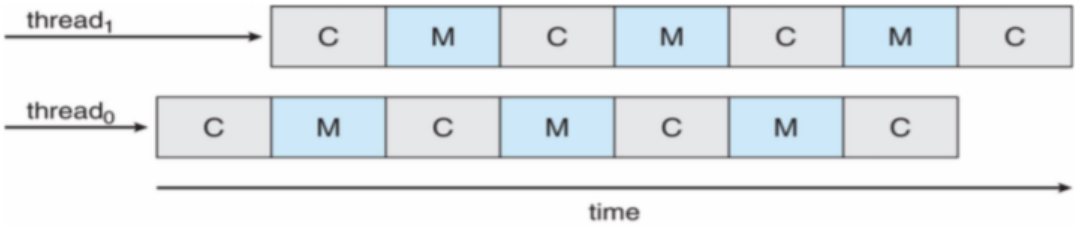
Un CPU burst di un thread può essere a sua volta visto come un'alternanza di operazioni che coinvolgono la CPU (compute cycle) e operazioni che coinvolgono la memoria (memory stall).



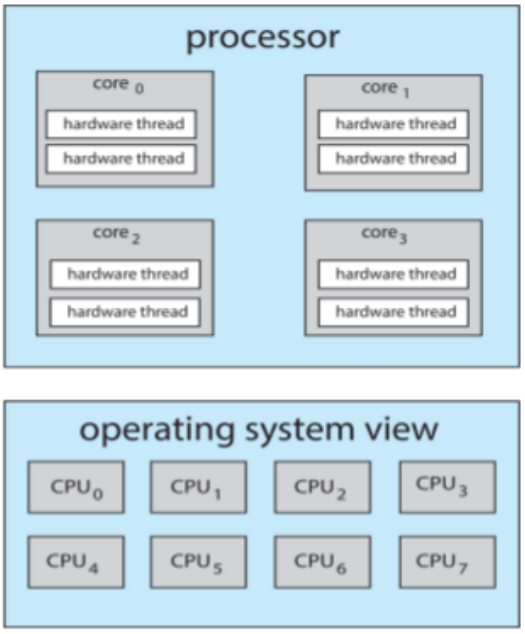
Multithreaded Multicore System

Se si viene a creare una situazione di stallo della CPU o meglio il thread finisce in memory stall, la CPU può essere ceduta ad un'altro thread. Si procede così

quindi in parallelo, ovvero quando un processo va in M, faccio partire un altro processo in fase di C.

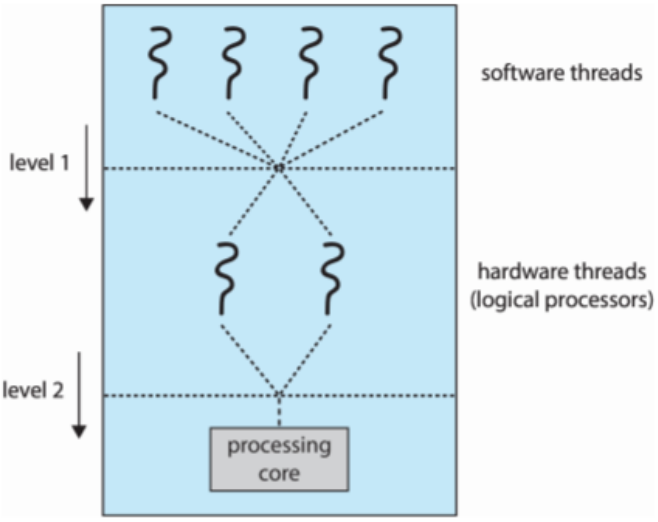


Assegnare hardware-thread multipli ad un unico core si chiama CHIP-MULTITHREADING.



Si applicano così due livelli di scheduling:

1) Il SO decide quale software thread assegnare a una CPU logica;
2) Ogni core decide quale hardware-thread assegnare ad un dato core fisico (a cui fanno riferimento un insieme di CPU logiche).



Criteri di Scheduling Multiprocessore

Si passa da un problema a una dimensione (*quale fra i processi ready rendere running?*) ad un problema bidimensionale (*e su quale CPU?*).

CRITERI :

-

Gli obiettivi dello scheduling monoprocessore rimangono validi.

-

Parallelismo dei JOB : massimizzare il parallelismo per sfruttare la concorrenza del JOB, bisogna massimizzare questo parametro. In alcuni casi per esempio non è utile minimizzare il tempo di esecuzione di un processo nel gruppo se non si minimizza anche quello di tutti gli altri (se il lavoro complessivo risulta terminato solo dopo la terminazione di tutti i processi nel gruppo).

-

Affinità con il processore : è la relazione di un JOB con un particolare processore, la sua memoria locale (sistemi NUMA) e la sua cache. Bisogna massimizzare questo parametro. Ovvero molteplici thread di un JOB vengono eseguiti sulla stessa CPU, evitando così il lasso di tempo necessario a svolgere il context switch. In alcuni casi nella cache potrebbero esserci ancora suoi dati (e nel TLB - potrebbero esserci ancora elementi della sua tabella delle pagine).

-

Bilanciamento del carico: il carico di lavoro deve essere equamente distribuito tra i processori. Bisogna massimizzare questo parametro. Non devono esserci thread in attesa quando ci sono processori inattivi.

JOB → insieme di thread

Affinità con il processore

Quando si parla di **affinità** con il processore bisogna fare la distinzione tra:

- soft affinity : il SO prova a mantenere i thread di uno stesso job nello stesso processore, ma senza garanzia.
- hard affinity : permette ad un job di indicare l'insieme dei processori dove devono girare i suoi thread.

Bilanciamento del carico

Si hanno algoritmi di bilanciamento del carico quando il SO si accorge che ci sono delle CPU che sono sovraccariche oppure scariche di lavoro e quindi può decidere di riequilibrare il carico di lavoro, tramite la migrazione di un certo numero di thread da una CPU ad un'altra per ristabilire l'equilibrio.

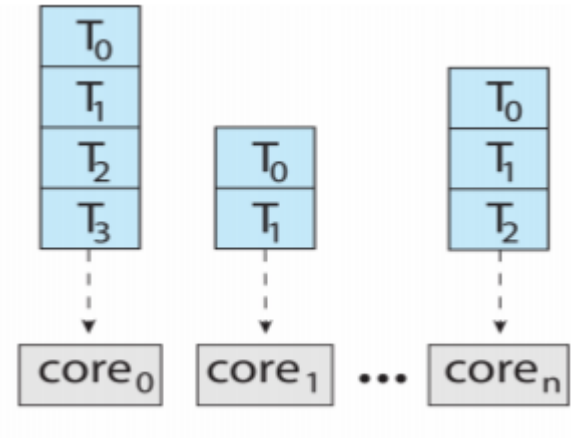
Due possibili strategie :

- push migration : si va alla ricerca di CPU sovraccariche per toglierle un po' di carico.
- pull migration : si va alla ricerca di CPU inattive a cui destinare il carico delle altre CPU sovraccariche.

Classificazione degli algoritmi

Si distinguono due casi:

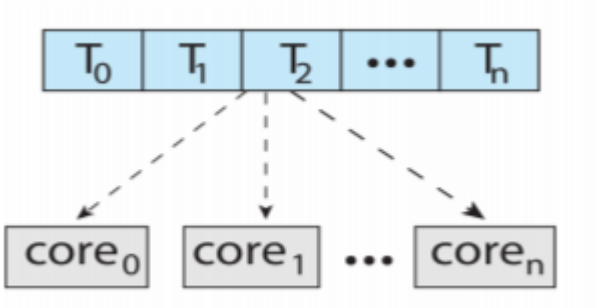
- **una coda di *ready* per processore** : nel momento in cui un thread diventa *ready* bisogna scegliere in quale coda inserirlo; a quel punto verrà schedulato quando l'algoritmo di scheduling di quella coda decide che è arrivato il suo turno. Avrò però necessità di un algoritmo aggiuntivo per il bilanciamento.



- **una coda di *ready* unica.**

  Posso applicare algoritmi come JOB-BLIND e JOB-AWARE

  Nel *job-blind* tutti i thread hanno la stessa importanza indipendentemente dal job a cui appartengono (l'algoritmo di scheduling non sa a quale job appartengono; è difficile garantire parallelismo e affinità).
  Nel *job-aware* l'algoritmo di scheduling è consapevole dell'appartenza di un thread al relativo job.
  Il bilanciamento deve essere garantito dagli algoritmi di scheduling stessi: non appena una di queste CPU si libera, l'algoritmo di scheduling va a pescare da quest'unica coda per decidere qual è il thread da assegnare a quella CPU che si è appena liberata.



Job-blind scheduling

Si possono estendere tutti gli algoritmi per i sistemi monoprocessore.

Essi sono semplici da realizzare e con un sovraccarico minimo, di contro però

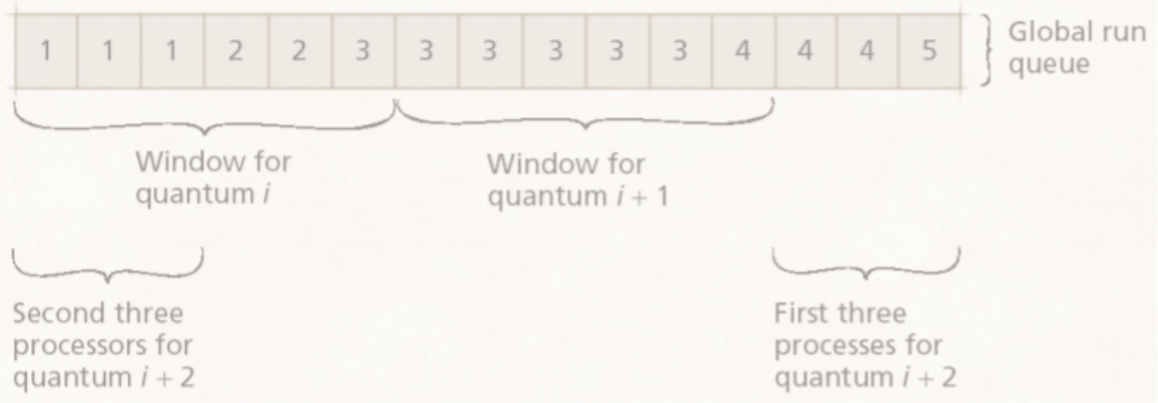
non tengono conto né del parallelismo né dell'affinità. Non si considera l'effettiva presenza di più processori.

Smart scheduling

È una ottimizzazione del RR. Si introduce un flag che permette di riconoscere un processo in sezione critica: quando un processo è in sezione critica e il quanto di tempo è scaduto, si lascia la CPU a quel processo fino a quando non rilascia la sua sezione critica.

Gang Scheduling (Coscheduling)

Tutti i thread di un job sono posti in elementi adiacenti della ready queue, ogni job adiacente all'altro. Lo scheduler mantiene una finestra di dimensioni pari al numero di processori. I thread interni alla finestra sono eseguiti in parallelo per al massimo un quanto. Al termine del quanto la finestra si sposta al gruppo successivo. Se un thread di una finestra passa a *waiting*, la finestra viene estesa di un thread a destra.

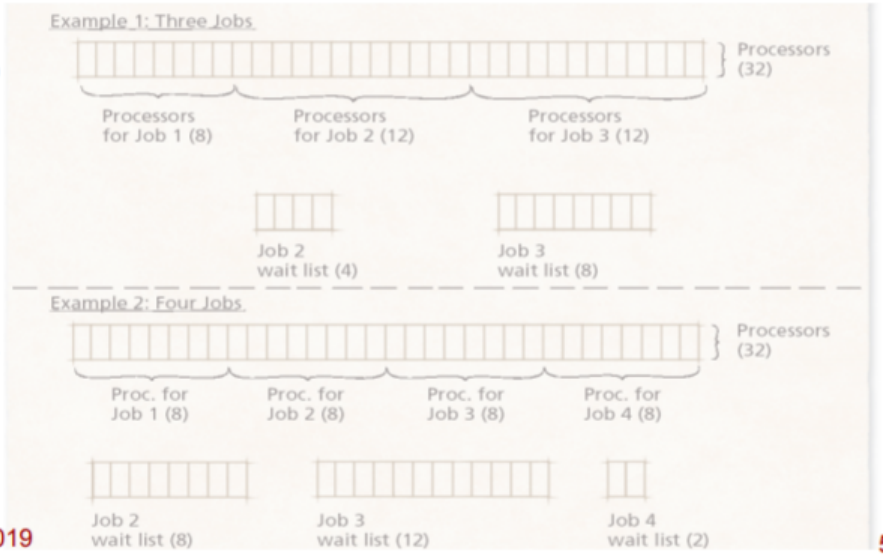


Job-aware scheduling

Partizionamento dinamico

Lo scheduler distribuisce equamente i job sui processori del sistema. Il numero di processori allocati ad un job è sempre minore o uguale al numero di thread eseguibili del job. Quando entra un nuovo job, il sistema aggiorna dinamicamente l'allocazione dei processori.

job 1 = 8 thread eseguibili job 2 = 16 thread eseguibili
job 3 = 20 thread eseguibili job 4 = 10 thread eseguibili



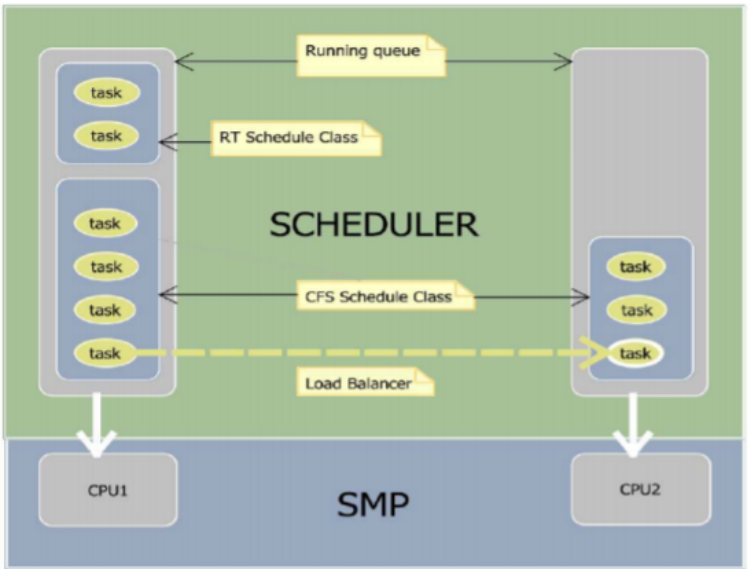
Scheduling della CPU in Linux

Linux SMP Load Balancing

- SMP : approccio simmetrico → ognuna delle CPU che Linux vede può ospitare il kernel; il kernel può andare in esecuzione su una qualsiasi CPU e può essere in esecuzione su più CPU.
- Linux adotta una soluzione multi-coda, ovvero una coda per ciascun processore (ogni CPU ha la sua coda di task ready). Così quando un processore si libera, può selezionare un task senza bloccare la coda agli altri processori (visto che ognuno ha la sua).
- Linux ha una componente aggiuntiva (oltre all'algoritmo di scheduling) che è data dalla parte che si occupa del bilanciamento del carico di lavoro: ogni volta che un task diventa ready entra in gioco l'algoritmo di bilanciamento del carico per stabilire su quale coda deve essere messo quel task.

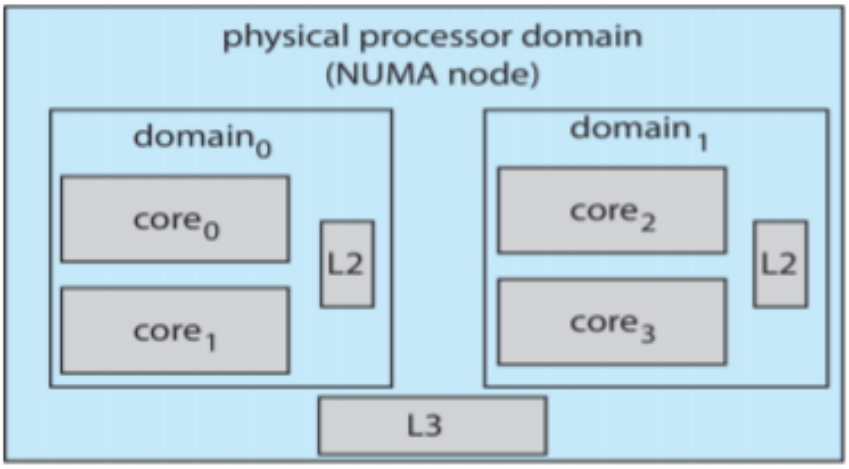
  Periodicamente questo algoritmo va a vedere se le code sono bilanciate oppure no (confronta la lunghezza delle code). Una volta che viene individuato un eventuale sbilanciamento, le code in questione vengono momentaneamente bloccate (semaforo rosso su queste code) per evitare problemi di corsa critica; i task vengono fatti migrare da una coda all'altra e

alla fine rimette a verde i semafori e quindi gli algoritmi di scheduling che agiscono su quelle code possono riprendere a lavorare.



Linux implementa il *Load Balancing*, ma è anche **NUMA-aware**: è l'algoritmo di bilanciamento del carico di Linux → cerca di garantire l'affinità col processore. Comunque lui individua una serie di CPU, ad un determinato job assegna un determinato insieme di CPU; quindi i suoi task verranno messi in attesa preferibilmente sulle code associate a quelle CPU, in modo da garantire che nelle memorie di cache ci siano le pagine di quei task che appartengono ad un determinato job.

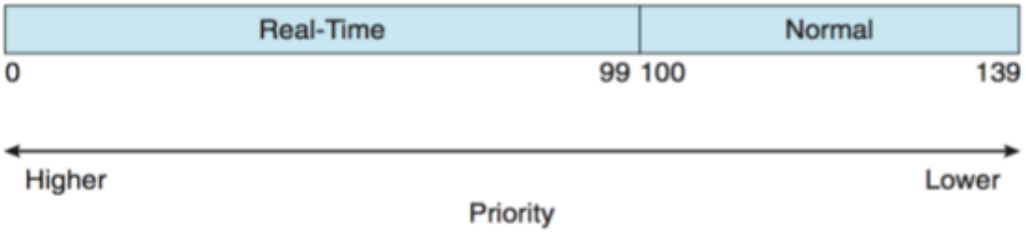
Quando si parla di **Scheduling domain** si intende un insieme di CPU-core.
I domain sono organizzati sulla base di quello che condividono (cioè memoria cache). L'obiettivo è impedire che i thread migrino da un dominio all'altro, mamtenendo così l'affinità.



Scheduling

Linux implementa due classi di scheduling (e per gestirle utilizza un algoritmo a code multiple) :

- **sched_fair** (time-sharing) : è la classe di scheduling dei task in time-sharing, con priorità compresa tra 100 e 139. Utilizza l'algoritmo di scheduling *Completely Fair Scheduler (CFS)*.
  I Processi vengono selezionati solo quando non ci sono task in *sched_rt*.
- **sched_rt** (real time) : è la classe di scheduling dei task (soft-) real-time, con priorità compresa tra 0 e 99. Utilizza un algoritmo a code multiple (100 code, una per priorità da 0 a 99).



Sched_rt

Si chiama **soft-real time** perché fa tutto ciò che è possibile per far concludere il task in un determinato intervallo di tempo, ma non garantisce nulla.

Un algoritmo di scheduling con priorità implementa una soft-real time, in quanto non ha modo di assicurare la fine nel intervallo di tempo.

- Come stabilire in quale coda deve essere inserito un task quando diventa *ready*?
  →
  Ad ogni task viene assegnata staticamente una priorità, quindi ogni volta che diventa ready viene messo nella coda corrispondente alla sua priorità.
- Come stabilire un algoritmo di scelta della coda?

  → Finché un task è in attesa sulla coda 0 non verranno presi in esame gli altri task; quando la coda 0 è vuota si va a vedere quali sono i task in attesa sulla coda 1. Il task in attesa sulla coda 1 ha a disposizione un quanto di tempo di 200 ms; nel frattempo arriva un task nella coda 0.
  Una volta che il task nella coda 1 ha terminato il suo quanto di tempo, siccome la coda 1 non è più vuota, si ritorna a servire i task sulla coda 0.

Dunque se non ho più processi nella coda di priorità n, vado nella coda con priorità n+1.

- Una volta scelta la coda, come viene gestita?

  → Viene implementato uno scheduling a singola coda di tipo RR con quanto di tempo 200ms.

Sched_fair

L'algoritmo si basa sull'idea di dare il processore al task che ne ha maggiore necessità.
Ha priorità di default pari a 120, ma può essere modificata mediante il **nice**: è un valore compreso tra -20 e +19; dunque la priorità può variare tra 100 e 139. Un task è tanto più *nice* quanto più è disponibile a cedere tempo di CPU.

Si parla di Fair Scheduling in quanto la priorità viene usata come fattore di accrescimento del tempo di runtime (tempo passato in stato di *running*).

La priorità viene usata come decadimento del quanto di tempo quando il task è in running: più è bassa la priorità, più è alto il fattore di decadimento.

Per valutare le prestazioni viene utilizzato un albero rosso-nero al posto di una

coda:

- tempo di selezione di task constante $O(1)$
(gli alberi rosso-neri hanno un tempo di ricerca pari a
$O(log_2 n)$, ma in Linux esiste un puntatore all'elemento che ha il vruntime (virtual runtime) minimo);
- tempo di inserimento di un task pari a
$O(log_2 n)$;
- tempo di cancellazione di un task pari a
$O(log_2 n)$.

Il quanto di tempo non è uniforme per tutti i task, ma dipende dal fattore di nice.

Il **Target time (T)** serve per controllare il tempo di risposta. Il suo obiettivo è quello di servire i vari task entro l'intervallo di tempo.

Il **peso (w)** dipende dal valore di nice.

D

nice > 0 → w_i < 1024 nice < 0 → w_i > 1024

1024 =
2^{10} → serve per spostare 10 bit

Il quanto di tempo viene calcolato in questo modo :

$$q_i = \frac{T \cdot w_i}{\sum_j w_j}$$

j varia tra tutti i processi ready

Stimiamo il tempo di attesa: un task, quando viene messo in coda, sa che nella peggiore delle ipotesi dovrà attendere un tempo pari a T prima che gli venga assegnata la CPU, perché a tutti gli altri task prima di lui è stata assegnata comunque una frazione di T.

Nell'arco di un Target time tutti i task ricevono la CPU almeno una volta.

Si cerca di capire quale processo sarà più lungo (quindi più lento); per fare ciò ci basta conoscere il
runtime, ovvero il tempo di CPU speso dal task dal momento in cui è stato creato fino ad ora (quanto tempo di CPU ha realmente utilizzato), al fine di una stima. Un processo che ha usufruito della CPU più degli altri, si suppone in maniera grossolana che abbia un CPU Burst elevato.

A tal proposito viene introdotto il **Virtual runtime,** il quale viene utilizzato per stimare indirettamente la durata di un CPU Burst.

Consideriamo il fattore di accrescimento $f_i = \frac{1024}{w_i} = (1.25)^{+nice_i}$

Il virtual runtime si calcola come:

$$vruntime_i = runtime_i \cdot f_i$$

- Nice < 0 ⇨ f < 1 ⇨ vruntime cresce più lentamente del tempo reale
- Nice = 0 ⇨ f = 1 ⇨vruntime cresce alla stessa velocità del tempo reale
- Nice > 0 ⇨ f > 1 ⇨vruntime cresce più velocemente del tempo reale

Si cerca di privilegiare i task creati più di recente, partendo dall'idea che i task più anziani sono poco interattivi.

Si premia la brevità con dei meccanismi che evitano l'attesa indefinita.

In un RB-Tree, il valore di un nodo è il suo vruntime. Lo scheduler seleziona sempre il nodo più a sinistra.
Di conseguenza il task con il runtime più piccolo viene selezionato per primo.
Il nodo del task selezionato viene rimosso dall'albero (lo stato del task è running).
Quando lo stato del task ritorna a ready, il suo nodo viene reinserito nell'albero.
Il nuovo nodo più a sinistra viene selezionato e la procedura si ripete.
Se un task passa gran parte del tempo in waiting (interattivo), il suo vruntime è basso e quindi la sua priorità sarà più alta rispetto a quella di task che sono costantemente in running (batch).

I task possono essere raggruppati in gruppi (la nozione di gruppo è equivalente a quella di job). Tutti i task dello stesso gruppo hanno lo stesso vruntime.