Scheduling

L'utilizzo esclusivo della CPU è gestito attraverso l'algoritmo di scheduling dei processi, che determina quale processo nella coda di ready deve essere eseguito successivamente. La scelta del processo dipende dal tipo di sistema , ad esempio batch, interattivo, real-time , in quanto ogni sistema ha uno scopo differente.

Il dispatcher, una componente del sistema operativo, gestisce il context switch che si verifica quando un processo richiede l'utilizzo della CPU. Il context switch coinvolge il salvataggio dello stato del processo corrente e il ripristino dello stato del processo successivo, e ha un costo associato, indicato come latenza di dispatching. Questo costo rappresenta un overhead aggiuntivo legato al tempo che intercorre tra la richiesta della CPU da parte di un processo e il momento in cui il processo inizia effettivamente ad utilizzare la CPU.

In generale, l'obiettivo dell'algoritmo di scheduling dei processi è quello di massimizzare l'utilizzo della CPU e garantire un equo accesso alle risorse di sistema per tutti i processi.

Processi I/O bounded e CPU-bounded

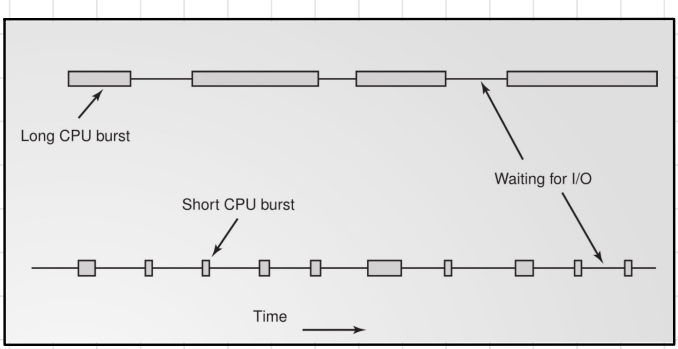
I processi I/O bounded e CPU bounded sono due tipi di processi che si differenziano per la loro attività principale e le conseguenti esigenze di risorse di sistema :

- un processo I/O bounded è un processo che trascorre la maggior parte del suo tempo in attesa di completare operazioni di input/output (I/O) come la lettura o la scrittura di dati su un dispositivo di archiviazione o la ricezione di dati da una rete.

- un processo CPU bounded, al contrario, trascorre la maggior parte del suo tempo eseguendo operazioni che richiedono una notevole quantità di elaborazione da parte della CPU, come il calcolo di algoritmi complessi o l'elaborazione di grandi quantità di dati.

Un processo può manifestare un comportamento I/O bounded e poi mutare il suo comportamento in qualcosa di più elaborativo (CPU-bounded). Non è semplice differenziare i processi tra le due tipologie, anche perchè la situazione può essere cangiante nel tempo. I processi si avvicinano ad essere sempre più I/O bounded in quanto le CPU tendono a diventare sempre più veloci, quindi si necessita di ottimizzare lo scheduling dei processi I/O bound .

Durante la vita di un processo, si possono verificare dei momenti in cui il processo richiede l'utilizzo al 100% della CPU, chiamati **CPU burst**. I processi CPU bounded, ovvero quelli che richiedono molte risorse di elaborazione della CPU, tendono ad avere CPU burst più lunghi. Al contrario, i processi I/O bounded, ovvero quelli che passano la maggior parte del loro tempo in attesa di operazioni di input/output, tendono ad avere CPU burst brevi e utilizzano meno tempo di CPU quando gli viene assegnata.
Inoltre, i tempi di attesa per le operazioni di input/output hanno più o meno la stessa durata, indipendentemente dal fatto che il processo sia I/O bounded o CPU bounded. Questo perché il tempo di attesa è determinato dal tempo impiegato dal dispositivo di input/output per completare l'operazione richiesta, e non dal tipo di processo in esecuzione.



Strategia

Identificare i processi I/O bounded può essere un vantaggio in quanto permette di dedicare la CPU a questa categoria di processi in modo prioritario. I processi I/O bounded richiedono la CPU per brevi periodi di tempo, ma trascorrono la maggior parte del loro tempo in attesa di completare le operazioni di input/output.
Se la CPU viene dedicata ai processi I/O bounded, essi saranno in grado di sfruttarla rapidamente e la rilasceranno presto, permettendo alla CPU di essere utilizzata in modo intensivo per completare l'esecuzione di tutti i processi I/O bounded. Ciò può **migliorare la reattività del sistema operativo** e garantire un utilizzo più efficiente delle risorse di sistema.
Una volta terminati i processi I/O bounded si passa all'esecuzione dei processi CPU-bounded che tutto sommato non si vedono eccessivamente penalizzati da questa scelta in quanto la CPU viene rilasciata in fretta dai processi I/O bounded.
La gestione dell'utilizzo esclusivo della CPU influisce sulla scelta dell'algoritmo di scheduling, nonché la gestione del context switch attraverso il dispatcher, tenendo conto del costo associato alla latenza di dispatching.

Sistemi preemptive e non-preemptive

Un algoritmo di scheduling di tipo non preemptive preleva un processo per eseguirlo e poi lo lascia semplicemente andare finché si blocca (sia su un I/O o in attesa di un altro processo) o finché non rilascia volontariamente la CPU. Nei sistemi chiusi, dove non ci sono interazioni con l'utente, è comune utilizzare un approccio non-preemptive che prevede la collaborazione tra processi, senza ricorrere alla prelazione. In questo contesto, non è un problema che un processo monopolizzi la CPU, poiché i processi collaborano tra loro per garantire l'accesso equo alle risorse di sistema.
Al contrario, un algoritmo di scheduling di tipo preemptive prende un processo e lo lascia eseguire per un tempo massimo prefissato. Se alla fine dell'intervallo è ancora in esecuzione, il processo è sospeso e lo scheduler ne prende un altro da eseguire (se è disponibile). Fare uno scheduling di tipo preemptive richiede che vi sia un interrupt del clock alla fine dell'intervallo per restituire il controllo della CPU allo scheduler.

Sistemi batch, interattivi e real-time

I sistemi batch, interattivi e real-time sono tre tipi di sistemi operativi che differiscono per il loro approccio all'elaborazione delle attività e alla gestione delle risorse di sistema.

I sistemi **batch** sono progettati per elaborare grandi quantità di lavoro in modo automatico e senza interazione utente. In genere, i lavori sono inviati in coda e processati in modo sequenziale, senza intervento umano, fino a quando tutti i lavori in coda non sono stati completati.
I sistemi **interattivi**, al contrario, sono progettati per consentire all'utente di interagire con il sistema in tempo reale, attraverso l'utilizzo di una interfaccia utente. In un sistema interattivo, l'utente può avviare applicazioni, modificare file e utilizzare altre funzionalità del sistema, interagendo con il sistema attraverso l'interfaccia utente.
I sistemi **real-time**, infine, sono progettati per garantire l'esecuzione tempestiva delle attività, in modo che i risultati siano disponibili entro un determinato limite di tempo. In un sistema real-time, le attività sono pianificate in base alle loro priorità e ai tempi di scadenza, e la risposta del sistema deve essere calcolata in modo prevedibile e in tempo reale.

1) Si necessita di avere delle metriche che forniscono modi per valutare l'efficienza di un sistema **batch** , si cerca di minimizzare certi parametri e di massimizzarne altri :

- massimizzare il **throughput** (produttività) che rappresenta la quantità di lavoro completato dal sistema in un determinato periodo di tempo. Tuttavia, il throughput non è utilizzato come parametro assoluto perché potrebbero esserci molti processi con tempi di esecuzione brevi che aumentano il throughput, ma che lasciano i processi più lunghi in attesa per un lungo periodo di tempo.
- minimizzare il **tempo di tournaround**(tempo di completamento) che rappresenta il tempo necessario per completare l'esecuzione di un processo. Questo parametro è importante perché indica quanto velocemente i processi vengono elaborati dal sistema operativo e restituiscono i risultati all'utente.

  → TEMPO ATTESA + DURATA

- minimizzare il **tempo di attesa** che è influenzato dagli algoritmi di scheduling e rappresenta il tempo che un processo trascorre in attesa nella coda di ready prima di ottenere l'accesso alla CPU. Questo parametro è importante perché indica quanto tempo un processo deve aspettare prima di poter utilizzare le risorse del sistema. Spesso viene considerato solo questo unico parametro per valutare la qualità degli algoritmi di scheduling.

2) Gli obiettivi principali dei **sistemi interattivi** riguardano principalmente :

- la massimizzazione dell'esperienza dell'utente, garantendo una risposta rapida e fluida alle richieste dell'utente. Ciò richiede la minimizzazione dei **tempi di risposta** a seguito delle richieste dell'utente, che dipendono dalle scelte degli algoritmi di scheduling nel determinare il processo che va in esecuzione.

3) D'altra parte, i sistemi **real-time** considerano principalmente parametri come :

- **rispetto delle scadenze :** è essenziale che le azioni richieste vengano eseguite entro un tempo prestabilito, al fine di evitare conseguenze negative.

- la **prevedibilità** : è importante per garantire che il sistema sia in grado di gestire in modo affidabile le richieste in tempo reale, senza interruzioni o ritardi imprevisti.

Scheduling nei sistemi batch

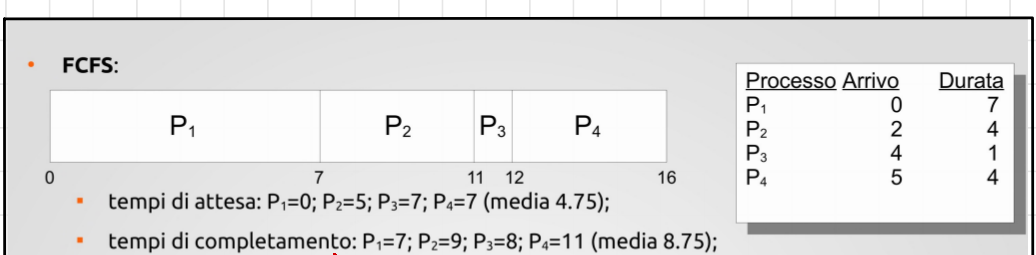
Lo scheduling assume che i processi , una volta presi in carico, non facciano input e output ma che utilizzino la CPU fino al suo completamento.

First-Come, First-Served

L'algoritmo FCFS o per ordine di arrivo è un algoritmo di scheduling dei processi in cui i processi vengono eseguiti nell'ordine in cui sono stati ricevuti dal sistema. I processi vengono accodati in una coda e vengono estratti dalla testa della coda, in ordine FIFO (First-In, First-Out), per essere eseguiti. Questo è un'algoritmo non-preemtive in quanto non viene sottratta la CPU ad un processo che si trova in esecuzione . L'algoritmo FCFS può causare la situazione di starvation, in cui alcuni processi potrebbero dover aspettare a lungo per essere eseguiti, se ci sono processi più lunghi in esecuzione.
Nonostante ciò, l'algoritmo FCFS ha il vantaggio di essere semplice ed efficiente in termini di utilizzo delle risorse di sistema, poiché non richiede una grande quantità di lavoro di scheduling. Inoltre, l'algoritmo FCFS può essere utile in situazioni in cui la priorità dei processi non è critica e il tempo di esecuzione dei processi è simile.

Diagrammi di Gantt

I diagrammi di Gantt sono strumenti utilizzati per la pianificazione e il controllo dei progetti. Si tratta di diagrammi a barre che mostrano le attività del progetto lungo un asse temporale. Ogni attività viene rappresentata da una barra che indica il suo inizio e la sua fine, e la durata dell'attività viene rappresentata dalla lunghezza della barra.
Il diagramma di Gantt aiuta a visualizzare le relazioni tra le attività del progetto e a identificare eventuali sovrapposizioni o ritardi.



I tempi medi di attesa e di completamento **tma** e **tmc** vengono calcolati sommando tutti i tempi di attesa (o di completamento) e la si divide sul numero di processi (nell'esempio n = 3).

Shortest Job First

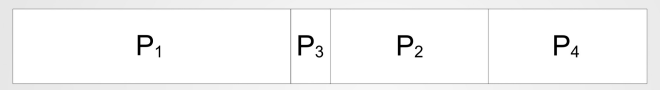
L'algoritmo di scheduling Shortest Job First (SJF) è un algoritmo non pre-emptive che privilegia l'esecuzione dei processi più brevi, migliora le statistiche di esecuzione, come il tempo di attesa medio e il tempo di completamento medio. L'algoritmo SJF presuppone di conoscere la durata dei processi.

L'algoritmo SJF è ottimale solo se tutti i lavori sono immediatamente disponibili (a tempo zero) e non ci sono nuovi processi in arrivo durante l'esecuzione. Questo succede in quanto scambiare i tempi di esecuzione dei processi migliora il numeratore in quanto posso facilmente abbassare notevolmente il tempo di attesa di processi brevi e allungare di poco l'esecuzione di processi lunghi. Questo causa un abbassamento del valore del numeratore in **tma** in questo caso specifico. In caso contrario, l'algoritmo SJF può portare a una situazione di starvation, in cui i processi più lunghi vengono continuamente posticipati in favore dei processi più brevi.

In generale, l'algoritmo SJF è utile in situazioni in cui il tempo di esecuzione dei processi è noto o può essere stimato con precisione, come ad esempio in sistemi batch o in sistemi in cui i processi hanno una durata nota. Tuttavia, l'algoritmo SJF può essere meno adatto in situazioni in cui i tempi di esecuzione dei processi sono variabili o difficili da stimare.

Processo	Arrivo	Durata
P_1	0	7
P_2	2	4
P_3	4	1
P_4	5	4

- **SJF:**



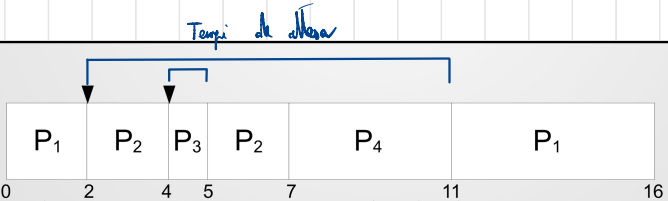
  - tempi di attesa: P_1=0; P_2=6; P_3=3; P_4=7 (media 4);
  - tempi di completamento: P_1=7; P_2=10; P_3=4; P_4=11 (media 8);

Shortest Remaining Time Next

SRTN è un algoritmo di scheduling dei processi che utilizza la politica di esecuzione del processo più breve rimasto da eseguire. In pratica, SRTN è una versione preemptive dell'algoritmo SJF (Shortest Job First).
In SRTN, forzo l'esecuzione del processo arrivato con tempo di esecuzione più breve, valutandolo in funzione del tempo di esecuzione residuo del processo corrente , non del suo tempo originale, in quanto questo processo è in esecuzione da un pò. Questo significa che un processo in esecuzione può essere interrotto se un nuovo processo con una durata minore arriva e deve essere eseguito. Ciò garantisce una maggiore efficienza e riduce il tempo di attesa medio.
SRTN richiede di conoscere il tempo di esecuzione rimanente di ogni processo, il che può essere difficile da stimare in situazioni reali. Inoltre, come tutti gli algoritmi pre-emptive, può causare un alto overhead di sistema, poiché richiede frequenti interruzioni e cambi di contesto.
Il calcolo del tempo di esecuzione totale tiene conto del reale tempo di arrivo del processo, anche se la sua esecuzione è stata interrota per prelazione. L' algoritmo SRTN è efficace in situazioni in cui il tempo di esecuzione dei processi può essere stimato con precisione e i processi hanno una durata variabile. L'algoritmo SRTN può essere utilizzato in sistemi interattivi in cui i processi sono di breve durata e la risposta rapida è essenziale.

Processo	Arrivo	Durata
P_1	0	7
P_2	2	4
P_3	4	1
P_4	5	4

- **SRTN:**



  - tempi di attesa: P_1=9; P_2=1; P_3=0; P_4=2 (media 3);
  - tempi di completamento: P_1=16; P_2=5; P_3=1; P_4=6 (media 7).

Spostandoci nel contesto dei **processi interattivi** si necessita di massimizzare la reattività alle richieste dell'utente, valutando metriche differenti. Valutiamo un algoritmo che necessita di prelazione ed inoltre **si abbandona l'idea di poter fare considerazione sui tempi di esecuzione dei processi** ed inoltre consideriamo i picchi di CPU burst.

Round-Robin

RR è un algoritmo di scheduling preemtive dei processi che prevede l'esecuzione dei processi in modo rotatorio. In pratica, ogni processo viene eseguito per un breve periodo di tempo (chiamato "quanto" o "time slice"), dopodiché viene interrotto e posto in fondo alla coda dei processi in attesa, mentre il prossimo processo in cima alla coda viene eseguito.

Valori tipici per i time-slice sono 20-50 ms. Con n processi ed un quanto quanto di q ms, ogni processo avrà diritto a 1/n della CPU e attenderà al più (n-1)q ms.

L'algoritmo Round Robin è utile in situazioni in cui i processi hanno una durata variabile e non è possibile prevedere con precisione il tempo di esecuzione di ogni processo. In questo modo, ogni processo riceve una quota equa del tempo di CPU disponibile e tutti i processi vengono eseguiti in modo tempestivo.
Round Robin può causare un alto overhead di sistema, poiché richiede frequenti interruzioni e cambi di contesto. Inoltre, se il quanto è impostato troppo basso, aumenta la reattività del sistema ma potrebbe rallentarlo a causa di un overhead eccessivo dato dai context-switch. Se invece il quantum è impostato troppo alto, esso potrebbe essere più grande di qualsiasi altro CPU-burst causando un comportamento di fatto identico a quello dell'algoritmo FCFS perchè mi ritrovo ad eseguire per intero il CPU-burst di un processo dato che dura meno del time-slice e la prelazione non agisce .

In generale, l'algoritmo Round Robin è utile in sistemi interattivi in cui la risposta rapida è essenziale e i processi hanno una durata variabile.

Abbiamo due scenari possibili dipendenti dalla natura del processo (CPU-bounded o I/O bounded) :

- Nei processi CPU-bound, il quanto di tempo viene consumato interamente durante il burst di CPU del processo. Questo significa che il processo utilizza l'intero quantum di tempo per eseguire operazioni sulla CPU, senza essere interrotto. Nel caso dei processi CPU-bound, il quanto di tempo deve essere impostato in modo da garantire che i processi non trascorrano troppo tempo in attesa di ottenere una nuova allocazione di CPU.

- Nei processi I/O-bound, il quanto di tempo viene consumato solo parzialmente, poiché i processi possono sospendersi durante il loro quantum di tempo per effettuare una richiesta di I/O. In questo caso, il processo viene temporaneamente sospeso fino al completamento dell'operazione di I/O e il quantum di tempo rimanente viene utilizzato solo dopo la ripresa dell'esecuzione del processo.

Distinguere in questo senso l'occupazione della CPU serve a differenziare i processi che al momento si comportano da CPU bound e da I/O bound e servirà a gestire i processi interattivi assegnando priorità differenti.

Priorità

Un processo si vede assegnata una sorta di etichetta che da significato all'urgenza con cui deve essere eseguito. I processi con un'alta priorità necessitano di un esecuzione privilegiata rispetto a quelli con bassa priorità.
Distinguiamo tra priorità statiche e dinamiche.

La priorità **statica** è assegnata ai processi in fase di creazione e non varia durante l'esecuzione del processo. In questo caso, la priorità viene assegnata in base alle caratteristiche del processo, come l'importanza, l'urgenza e la criticità. Ad esempio, i processi di sistema possono essere assegnati una priorità più alta rispetto ai processi utente.

La priorità **dinamica**, invece, viene assegnata durante l'esecuzione del processo e può variare in base alle condizioni del sistema e dello stesso processo. In questo caso, la priorità viene assegnata in base alle prestazioni del processo, come il tempo di attesa, il tempo di esecuzione e il tempo di completamento. Ad esempio, un processo che ha aspettato a lungo per l'allocazione della CPU può ricevere una priorità più alta per garantire una rapida esecuzione(aging).

Per favorire i processi I/O bounded si assegna all'etichetta di priorità l'inverso della frazione dell'ultimo quanto di tempo utilizzato : processi che hanno utilizzato una frazione più piccola del quanto di tempo avranno una priorità più alta e questo corrisponde a favorire i processi che mostrano un comportamento I/O bounded. Un processo CPU-bounded quando riceve la CPU ha un'occupazione più lunga della CPU a causa di questo meccanismo di preferenza della priorità ma non rischia di monopolizzarla se si applica la **priorità con prelazione** , infatti la CPU verrà tolta al processo CPU bounded e verrà assegnata ai processi I/O bounded in quanto hanno una priorità più alta . Questo è un modo elegante per gestire al meglio le risorse di sistema.

Prelazione vs Senza prelazione

Applicando questo meccanismo senza la prelazione si ha un effetto del tutto analogo all'esecuzione dello SJF, che predilige l'esecuzione di processi brevi, assegnando una priorità pari a l'inverso tempo di esecuzione del processo , così da favorire i processi con tempo di esecuzione più bassa. Implementando un meccanismo di prelazione si ottiene l'algoritmo SRTN.

Starvation risolto con aging

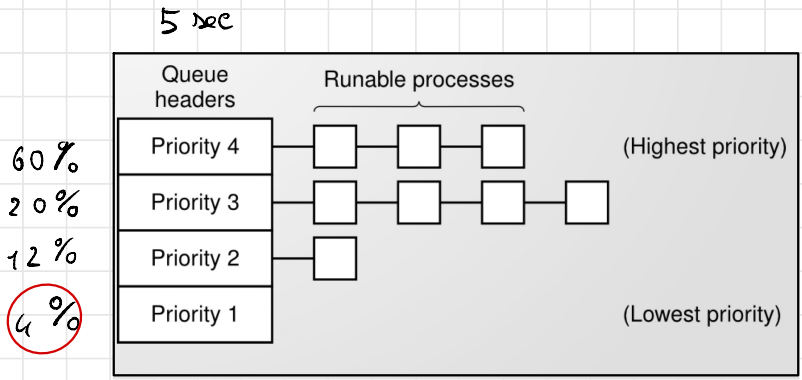
Un evento di starvation si verifica quando **un processo con una priorità bassa non riesce a ottenere la CPU perché viene costantemente superato da processi con una priorità più alta**. In questo caso, il processo a bassa priorità potrebbe dover attendere a lungo prima di poter eseguire, causando un ritardo significativo nella sua esecuzione. Questo problema può verificarsi soprattutto nei sistemi interattivi, dove i processi hanno bisogno di essere eseguiti rapidamente per garantire una risposta tempestiva all'utente.
Per risolvere il problema della starvation, è possibile utilizzare una tecnica chiamata **aging**. L'aging consiste nell'aumentare gradualmente la priorità dei processi a bassa priorità man mano che trascorre il tempo. In questo modo, i processi a bassa priorità che attendono da molto tempo possono ricevere una priorità più alta e avere maggiori possibilità di essere eseguiti. L'aging è particolarmente utile nei sistemi interattivi, poiché garantisce che tutti i processi ricevano una quota equa del tempo di CPU disponibile.

Scheduling a code multiple (code di priorità)

Lo schema prevede l'esistenza di una coda per ogni classe di priorità e si inseriscono i PCB dei processi all'interno delle classi rispettando le priorità dei processi. Si schedulano i processi attraverso l'utilizzo di due algoritmi di scheduling in verticale ed in orizzontale. Nelle code più alte ci saranno i PCB dei processi I/O bounded e nelle code più basse quelli CPU-bounded. Scelgo di applicare in senso verticale un algoritmo che prevede di schedulare i processi nella coda più alta non vuota , in modo da favorire i processi con priorità alta. Una volta scelta la coda da smaltire si può pensare di applicare un algoritmo come il Round Robin, trattandosi di processi interattivi. Viene assegnato un quanto di tempo differente tra le varie code, in particolare assegno quanti di tempo brevi nelle code più alte, per massimizzare la reattività del sistema senza ottenere un impatto grave sull'overhead , mentre si alzano i quanti di tempo scendendo nelle code a bassa priorità in quanto se si schedulano i processi in basso è perchè non vi sono più processi altamente interattivi da eseguire. Aumentando il quanto di tempo si ottiene un comportamento del tutto analogo al FCFS per i processi altamente CPU-bounded , ma non avere prelazione nei processi CPU-bound non causa la monopolizzazione della CPU perchè se rientra un processo a priorità più alta esso verrà schedulato dall'algoritmo verticale che invece ha la prelazione.

Priorità fisse

Tuttavia si soffre ancora di starvation in quanto le code con priorità più bassa sono penalizzate in favore delle code a priorità più alta. Per risolvere il problema della starvation, viene proposta l'assegnazione di un tempo di esecuzione alle intere code. In questo modo, ogni coda utilizza una percentuale del tempo totale concesso, garantendo che anche le code a bassa priorità abbiano un tempo di esecuzione assegnato e non vengano penalizzate rispetto alle code a priorità più alta.
In pratica, ad ogni coda di priorità viene assegnata una percentuale del tempo totale di CPU disponibile per l'esecuzione dei processi. Ad esempio, se ci sono tre code di priorità e il sistema dispone di un totale di 100 unità di tempo di CPU, le tre code potrebbero utilizzare rispettivamente il 60%, il 25% e il 15% del tempo totale di CPU. In questo modo, anche le code a bassa priorità avranno un tempo di esecuzione assegnato e non saranno penalizzate rispetto alle code a priorità più alta. Questa soluzione rappresenta una strategia efficace per mitigare il problema della starvation.
Tuttavia, l'approccio con l'assegnazione di un tempo di esecuzione alle intere code non è molto flessibile.



Feedback (o retroazione)

L'approccio con feedback è un'altra soluzione data per mitigare la starvation mediante l'aggiustamento dinamico delle priorità dei processi in base alle loro prestazioni. In questo approccio, i processi vengono monitorati costantemente per individuare quelli che richiedono più o meno risorse di sistema rispetto alle attese. Il sistema può quindi utilizzare un meccanismo di upgrade e downgrade delle classi di processi per adattare dinamicamente le priorità dei processi in base al loro utilizzo del time-slice.
Ad esempio, se un processo a bassa priorità viene eseguito per un lungo periodo di tempo, il sistema può decidere di aumentare la sua priorità per garantire un utilizzo più efficiente della CPU. Questo aggiustamento dinamico delle priorità garantisce che i processi non siano bloccati o ritardati in modo permanente, prevenendo il problema della starvation.
Complessivamente, l'approccio con feedback fornisce una soluzione flessibile e adattabile al problema della starvation, garantendo che tutti i processi ricevano una giusta quota di risorse di sistema e prevenendo il blocco indefinito di qualsiasi processo.

Si pensa di applicare il principio per brevità utilizzato per i sistemi batch adattandolo alle metriche valutabili per i processi interattivi, scegliendo il processo che , una volta mandato in esecuzione, restituirà la CPU per primo.

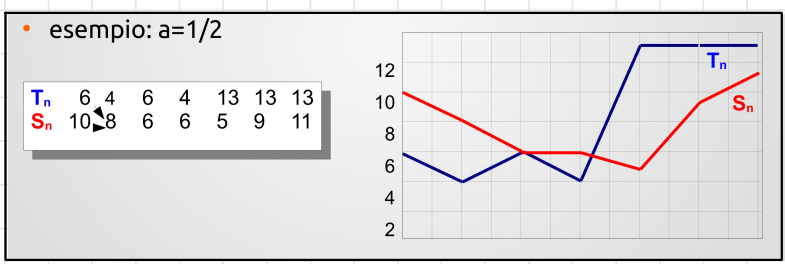
Shortest Process Next

SPN è un algoritmo di scheduling utilizzato nei sistemi operativi per allocare il tempo della CPU ai processi. In SPN, il processo con il tempo di elaborazione stimato più piccolo viene selezionato per l'esecuzione successiva. L'idea di base di SPN è quella di minimizzare il tempo medio di attesa per tutti i processi. Quando un processo arriva, viene stimato il suo tempo di elaborazione previsto. **La scelta ricade sul processo col CPU burst più breve in quanto manifesta un comportamento più I/O bounded**. L'algortimo si basa sul calcolo della durata del prossimo CPU burst (Tn+1) valutandolo in relazione alla durata dei CPU burst precedenti (Tn-2, Tn-1, Tn). Si calcola una media pesata che attribuisce un peso maggiore alle stime (Sn) più recenti in modo da adattare le stime al comportamento osservato. Se si implementa una versione che sfrutta la prelazione di SPN , se arriva un nuovo processo con un tempo di elaborazione previsto più piccolo rispetto al tempo residuo del processo in esecuzione, il processo in esecuzione viene prelazionato e il nuovo processo viene eseguito.

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

STIMA DEL PROSSIMO BURST — STIMA ATTUALE — REALE DURATA DEL BURST

La scelta di un parametro di peso compreso tra 0 e 1 fornisce una stima meno prona a reagire rispetto ai cambiamenti ma fornisce una curva che cambia comportamento in maniera meno repentina. Un valore di a prossimo a 1 pesa molto meno lo storico delle stime precedenti, mentre se a è prossimo a 0 sto valutando molto meno la stima attuale.



L'algoritmo SPN è ottimale nel senso che minimizza il tempo medio di attesa per tutti i processi. Tuttavia, **richiede la conoscenza del tempo di elaborazione previsto di ciascun processo**, che potrebbe non essere disponibile in pratica. Inoltre , se si pensa di implementare una versione con prelazione, SPN potrebbe comportare un gran numero di switch di contesto, che possono essere costosi in termini di prestazioni. SPN può essere un algoritmo di scheduling efficace quando sono disponibili stime accurate dei tempi di elaborazione e il costo degli switch di contesto è bilanciato dal beneficio di minimizzare il tempo medio di attesa per tutti i processi. Si nota che , a fronte di una stima molto accurata dei tempi di elaborazione, l'algoritmo applica fedelmente il principio per brevità anche sui sitemi interattivi.

Scheduling garantito

Lo "scheduling garantito" è un modello di scheduling che garantisce che un certo tipo di processo abbia sempre accesso alla CPU, anche se ci sono altri processi in esecuzione. Questo tipo di scheduling è spesso utilizzato per garantire che i processi ad alta priorità siano eseguiti in modo tempestivo e affidabile. L'obiettivo del scheduling garantito è quello di garantire che i processi ad alta priorità ricevano sempre l'accesso necessario alla CPU per soddisfare i loro requisiti di tempo reali.
Si effettua una scelta (che sta alla base dell'algoritmo di scheduling) che tiene conto del tempo "promesso" al processo per quanto riguarda l'utilizzo della CPU e facendo un rapporto tra il tempo promesso ed il tempo realmente fornito ai processi, si fanno delle considerazioni che influenzano la scelta fatta valutando diversamente i processi in base al tempo di utilizzo di CPU loro concesso, in funzione di quanto stato promesso. Si valuta anche la qualità del servizio concesso valutando questo rapporto.

Scheduling a lotteria

Si distribuiscono ai vari processi vari “ticket” rappresentanti quanti di tempo in cui è concesso loro l’utilizzo della CPU. Una scelta potrebbe essere assegnare la CPU al processo con il ticket corrispondente all’identificativo che è stato estratto. Le estrazioni vengono effettuate in modo randomico e quindi i processi ai quali sono stati assegnati più ticket, hanno una “promessa” più importante in quanto si vedranno assegnati più quanti di CPU. Un comportamento collaborativo prevede che un processo potrebbe cedere parte dei propri ticket ad un processo figlio per dare sussistenza e per fornire un boost di priorità per la sua esecuzione.

Scheduling fair-share

Lo scheduling fair-share è un modello di scheduling utilizzato per garantire che ogni utente o gruppo di utenti abbia un "quota equa" di CPU, in un contesto dove i processi abbiano tutti uno stesso status. L’idea è di applicare una priorità ad un gruppo di processi e non al singolo processo.

Il sistema di scheduling fair-share utilizza un algoritmo di pianificazione che tiene conto dell'uso delle risorse di ogni utente o gruppo di utenti nel tempo e assegna loro una "quota equa" di risorse in base alla loro attuale utilizzazione delle risorse. In questo modo, ogni utente o gruppo di utenti ha un accesso equo alle risorse del sistema. Ad esempio, se un utente richiede più CPU rispetto ad un altro, gli verrà assegnata una quota di CPU maggiore.

Per fissare le idee si possono immaginare due utenti , uno con 1 solo processo ed un’altro con 9 processi, allora si potrebbe pensare di assegnare il 10% di CPU al primo utente ed il 90% di CPU al secondo utente. L’idea è invece quella di gestire le risorse per gruppi di utenze e non per singoli processi, quindi di fatto si assegna il 50% della CPU ad ogni utente se si implementa l’equità data dallo scheduling per gruppi (es. fair-share).
Una ripartizione così precisa della CPU si appoggia alla metodologia dello scheduling garantito, nell’esempio si forniscono il 50% dei ticket ad ognuno dei due gruppi .

Scheduling nei thread

C’è una differenza nella gestione dello scheduling nei thread a seconda dell’implementazione del thread stesso, se a livello utente o a livello kernel. A livello utente la convivenza tra i vari thread all’interno dello stesso processo è puramente collaborativa, per tanto non si hanno meccanismi di prelazione.
Spostandoci sui thread a livello kernel, avendo l’aiuto del sistema operativo, si può pensare a dei meccanismi di prelazione per favorire lo scambio di thread imparentati che hanno un context switch più veloce rispetto a quello di processi differenti.

Scheduling nei sistemi multicore

Quando si hanno a disposizione N CPU per gestire M processi la situazione cambia consistentemente in quanto si può ottenere il reale parallelismo dell’esecuzione di istruzioni. Supponiamo di avere 4 core , bisogna di avere un modo per smistare i processi tra le quattro unità di elaborazione in modo da sfruttare al meglio le risorse di sistema. Quello che si vuole ottenere è una ripartizione del carico con il minimo overhead possibile.
Si hanno due approcci differenti in base alla tipologia di processi gestiti dalle CPU :

- **Multielaborazione asimmetrica** : un possibile approccio potrebbe prevedere che uno dei quattro core (amministratore) gestisca i processi del Kernel, tenendo una coda di processi pronti e si occupa di gestire tutte le routine del Sistema Operativo, mentre gli altri 3 si occupano della gestione dei processi utente. Questo approccio si adatta male in termini di scalabilità al caso in cui ci sono molti più core , infatti all’aumentare del numero di processori averne uno solo che fa da master farà da collo di bottiglia, per tanto questo approccio è caduto in disuso.

- **Multielaborazione simmetrica (SMP)** : l’idea è che tutti i core si occupano dell’elaborazione degli stessi processi e si ha quindi una equità di ruolo. Si può pensare che una ripartizione equa del carico di lavoro garantisca che ogni core abbia un carico uguale agli altri. Un problema potrebbe essere decidere se ogni core debba avere una coda propria dei processi pronti o se ci debba essere un’unica coda condivisa tra tutti i core. In quest’ultimo scenario di **coda unica** si deve garantire la mutua esclusione per gestire l’accesso concorrente alla struttura (race condition).

Si può pensare di ottenere la mutua esclusione tramite una variabile di Lock , ma anche questo si traduce in un problema di scalabilità nel caso di molte CPU. Il collo di bottiglia stavolta è dato dal grado di concorrenza di processi sull'unica struttura dati condivisa , infatti il meccanismo di lock blocca tutte le CPU che continueranno a fare spin lock per verificare la disponibilità della risorsa.
L'approccio valido è dato dal primo scenario descritto ovvero quello con **coda dedicata**, avere una coda per ogni core . Questo modello è sicuramente scalabile in quanto ogni core non si bloccherà mai a seguito delle scelte di altre CPU ma si crea uno sbilanciamento del carico (che non avevo con l'approccio a coda unica) , infatti si deve scegliere dove inserire i processi in arrivo ed in ogni caso la situazione è molto dinamica (processi che muoiono ecc).
Il bilanciamento viene creato da meccanisimi di **migrazione**, ovvero lo spostamento di processi da una coda più piena ad una più vuota :

- migrazione **guidata** (di tipo push) : si attiva periodicamente su una qualunque delle CPU e visiona lo stato delle varie code e migra in situazioni di sbilanciamento. La scansione e la migrazione hanno un costo in quanto si deve bloccare una coda, fare un'estrazione e proseguire con l'inserimento in un'altra coda e farla periodicamente causa un forte overhead. Lo spostamento dei processi viene fatto da un servizio che è in esecuzione su tutte le CPU.

- migrazione **spontanea** (di tipo pull) : scatta in seguito a meccanismi differenti , infatti quanto una coda si svuota del tutto si va a ricercare una coda che deve smaltire dei processi dalla quale vengono prelevati i processi ed inseriti nella coda vuota. Questo tipo di migrazione scatta su una precisa CPU che si fa assegnare un processo prelevato dalla coda di un altra CPU.

Le due strategie di migrazione vengono utilizzate insieme in quanto agiscono in momenti diversi e a seguito di meccanismi diversi.

Predilezione di un processo per una CPU

Questo è un vantaggio che è reso in modo **implicito nell'approccio a coda dedicata** e non in quello a coda unica. Predilezione di un processo, ovvero c'è un vincolo che lega un processo ad essere eseguito su una sola CPU. Nel caso di code dedicate questo meccanismo viene implicitamente applicato a meno di migrazione in quanto un processo viene eseguito solo dalla CPU a cui appartiene la coda contenente quel processo (il che risulta un vantaggio per il modello a code dedicate). Analizzando il caso di coda unica condivisa tra le varie CPU invece non è così scontato ed applicare l'affiliazione risulta vantaggioso. Ogni CPU mantiene una propria cache di primo livello, che mantiene i dati relativi ai processi che ha eseguito e verranno rimossi solo se i processi correnti necessitano di vedersi dedicate più risorse. Quando viene eseguito un nuovo processo quindi, il core ha in cache dei dati relativi ai processi che ha eseguito precedentemente. Il vantaggio è creato perchè forzando l'esecuzione di un processo sempre nella stessa CPU, tra un'avvicendamento e l'altro di esecuzione dello stesso processo sulla stessa CPU (non viene eseguito tutto in una volta) ho bisogno di dati che trovo già in cache e quindi avrò con molta probabilità eventi di cache hit piuttosto che cache miss che fanno perdere tempo a causa di un accesso alla memoria centrale. La predilezione viene distinta in :

Predilezione debole : il legame che c'è tra un processo e la CPU può essere spezzato per un buon motivo , ad esempio migrazione dato che viene prediletto il bilanciamento del carico all'affiliazione di un processo con quella CPU.

Predilezione forte : non è predisposta di default per nessun processo ma se viene forzata, attraverso delle system call, il legame che c'è tra CPU e processo non verrà spezzato da fattori esterni.

Cosa usano i nostri Sistemi Operativi?

- elementi comuni:
  - thread, SMP, gestione priorità, predilezione per i processi IO-bounded
- **Windows**:
  - scheduler basato su code di priorità con varie;
  - euristiche per migliorare il servizio dei processio interattivi e in particolare di foreground;
  - euristiche per evitare il problema dell'inversione di priorità.
- **Linux**:
  - scheduling basato su task (generalizzazione di processi e thread);
  - Completely Fair Scheduler (CFS): moderno scheduler garantito;
- **MacOS**:
  - Mach scheduler basato su code di priorità con euristiche.