PROCESSI E THREAD: STRUTTURA

Multiprogrammazione e multitasking

Questi concetti sono le tecniche che adottiamo per l'esecuzione di un SO.

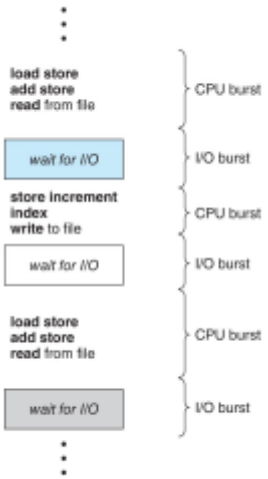
Due obiettivi dei sistemi operativi:

1. **Efficienza**: mantenere impegnata la (o le) CPU il maggior tempo possibile nell'esecuzione dei programmi (se ci sono programmi da eseguire);
2. **Reattività**: dare l'illusione che ogni processo progredisca continuamente nella propria esecuzione, come se avesse una CPU dedicata;
   Ci sono due modi in cui un processo può eseguire:
   a. **Completamente batch**: il programma aspetta che la CPU gli viene dedicata e può aspettare per un tempo anche lungo. Una volta che avrà la CPU, gli verrà completamente dedicata il 100% del tempo e dopodiché esegue completamente per il tempo necessario e poi lascia la CPU. Inefficiente e poco reattivo perché durante un'operazione IO, che sono molto lente, e la CPU durante quell'intervallo di tempo è fermo e non serve al
   b. **Multiprogrammazione** e il **multitasking** (o **time-sharing**).
      • Obiettivo della multiprogrammazione: impedire che un programma che non è in condizione di proseguire l'esecuzione mantenga la CPU.
      • Obiettivo del multitasking: far sì che un programma interattivo possa reagire agli input utente in un tempo accettabile.

Notare che la multiprogrammazione non è una tecnica rilevante per i sistemi puramente batch.

Burst CPU e I/O

Un'applicazione di solito non effettua tutte le operazioni di IO in un solo momento e tutte le operazioni di calcolo in un solo momento, bensì le alterna in burst. La tipica esecuzione di un processo è data da Burst di CPU (principalmente fa calcoli) alternati a Burst di IO (il processo di ferma in attesa di I/O). L'obiettivo della multiprogrammazione è massimizzare l'utilizzo della CPU. Gli algoritmi di scheduling sfruttano il fatto che di norma l'esecuzione di un processo è una sequenza di:



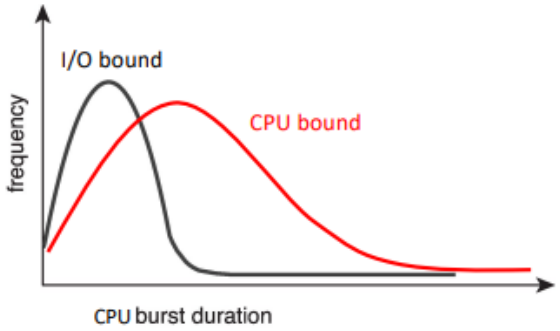
• Burst della CPU: sequenza di operazioni di CPU
• Burst dell'I/O: attesa completamento operazione di I/O

Il tempo che ci impiegano i due burst dipende da quanto tempo ci mette l'utente a fornire queste informazioni e quanto il sistema ci impiega a fornire un output.
Tipica esecuzione di un processo è data da momenti di burst della CPU alternati da burst di I/O, dove il processo si blocca in attesa di input e output.

Distribuzione delle durate dei burst della CPU

La distribuzione delle durante dei burst della CPU determina che tipo di programmi stiamo eseguendo.

Programma con prevalenza di I/O (I/O bound):
• Elevato numero di burst CPU brevi
• Ridotto numero di burst CPU lunghi
Tipico dei programmi interattivi (Ad esempio PowerPoint)



Programma con prevalenza di CPU (CPU bound):
• Elevato numero di burst CPU lunghi
• Ridotto numero di burst CPU brevi
Tipico dei programmi batch (tanta computazione, meno interazione utente)

In entrambi i casi la curva della distribuzione ha la forma riportata accanto, ma:
• I/O bound: il massimo sta più a sinistra
• CPU bound: il massimo sta più a destra

Si cerca di massimizzare la reattività dei programmi I/O bound, quelli CPU bound hanno più bisogno di efficienza, di tenere impegnato la CPU il più possibile.

Multiprogrammazione

Il sistema operativo mantiene in memoria le immagini di tutti i processi da eseguire.
Quando una CPU non è impegnata ad eseguire un processo (non sta eseguendo niente - idle), il sistema operativo seleziona un processo non in esecuzione e gli assegna la CPU. Se esiste un processo da eseguire e la CPU è in idle, la CPU viene assegnato al processo.
Quando un processo non può proseguire l'esecuzione (ad es. perché deve attendere il termine dell'input di dati che gli servono per proseguire), la sua CPU viene assegnata ad un altro processo non in esecuzione.
Come risultato, se i processi sono più delle CPU, queste saranno impegnate nell'esecuzione di qualche processo per la maggior parte del tempo.

Due idee:

1. Mantenere in memoria le immagini di tutti i processi
2. Sottrarre la CPU ad un processo che non lo sta usando perché impegnato in un burst di I/O e assegnarlo ad un processo che in quel momento non ha la CPU ma ha la necessità di fare burst di CPU.

Multitasking

È un'estensione della multiprogrammazione per i sistemi interattivi.
La CPU viene «sottratta» periodicamente al programma in esecuzione ed assegnata ad un altro programma. Con una certa frequenza, seppur la CPU è impegnato in un Burst di CPU, viene sottratto e lo assegno ad un altro programma. In questo modo tutti i programmi progrediscono in maniera continuativa nella propria esecuzione, anziché solo nei momenti in cui il programma che detiene la CPU si mette in attesa. Questo fa sì che i programmi batch, che hanno burst CPU lunghi e pochi burst I/O, non monopolizzino la CPU a discapito dei programmi interattivi.

Se fosse solo multiprogrammazione e abbiamo solo un programma estremamente CPU bound. In questo caso quando questo processo tiene la CPU, se ne avessimo solo uno, tale processo lo utilizzerebbe e lo terrebbe impegnato fino a quando non termina e se la durata è molto lunga, i processi I/O bound non potrebbero andare avanti perché la CPU verrebbe impegnata nell'esecuzione di lunghissimi CPU bound, processi batch.

Multiprogrammazione e memoria

La multiprogrammazione richiede che tutte le immagini di tutti i processi siano in memoria perché questi possano essere eseguibili.

Se i processi sono troppi non possono essere contenuti tutti in memoria: la tecnica dello **swapping** può essere usata per spostare le immagini dentro/fuori dalla memoria e spostarlo in una memoria secondaria.

Se l'immagine di un processo è troppo grande, la **memoria virtuale** è un'ulteriore tecnica che permette di eseguire un processo la cui immagine non è completamente in memoria.

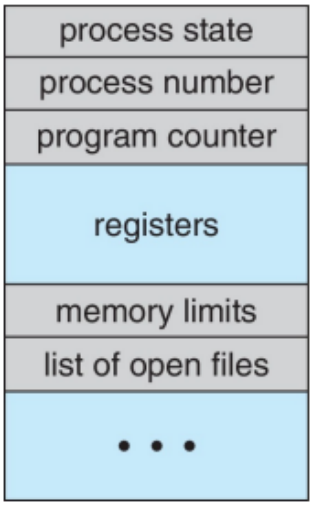
Queste tecniche sono complementari e aumentano il numero di processi che possono essere eseguiti in multiprogrammazione, ossia il **grado di multiprogrammazione**.

Swapping -> Per avere più immagini di quante la memoria ne possa contenere;

Memoria virtuale -> Per avere immagini di un processo molto più di grandi di quanto la memoria possa contenere.

Implementazione dei processi

Process Control Block (PCB)



Detto anche task control block, è la struttura dati del kernel che contiene tutte le informazioni relative ad un processo:
• Process state: ready, running…
• Process number (o PID): identifica il processo
• Program counter: contenuto del registro «istruzione successiva»
• Registers: contenuto dei registri del processore
• Informazioni relative alla gestione della memoria: memoria allocata al processo
• Informazioni sull'I/O: dispositivi assegnati al processo, elenco file aperti…
• Informazioni di scheduling: priorità, puntatori a code di scheduling…
• Informazioni di accounting: CPU utilizzata, tempo trascorso

Stato di un processo

Durante l'esecuzione, un processo cambia più volte stato.

Gli stati possibili di un processo sono:
• Nuovo (**new**): il processo è creato, ma non ancora ammesso all'esecuzione;
• Pronto (**ready**): il processo può essere eseguito (è in attesa che gli sia assegnata una CPU);
• In esecuzione (**running**): le sue istruzioni vengono eseguite da qualche CPU;
• In attesa (**waiting**): il processo non può essere eseguito perché è in attesa che si verifichi qualche evento (ad es. il completamento di un'operazione di I/O);
• Terminato (**terminated**): il processo ha terminato l'esecuzione.

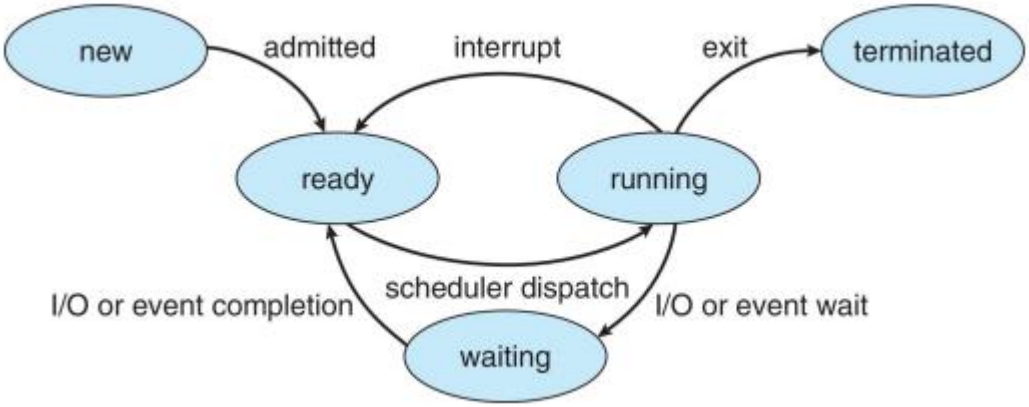


Figura 1: Diagramma di transizione di stato dei processi

- Un processo quando viene creato non viene subito ammesso alla esecuzione, perché potrebbero esserci già troppi processi all'intero del sistema operativo.
- Il processo appena creato viene mantenuto in questo stato new ma non è ancora ammesso allo scheduling.
- Quando la situazione è tranquilla, quando il numero totale di processi che l'SO si deve alternate in questi tre stati: ready, running e waiting, è sufficientemente basso da ammettere un altro, il processo viene ammesso e entra nel ready, ma non ha ancora CPU.
- Prima o poi una CPU deve darlo a questo processo pronto e poi ci sarà la scheduler dispatch e andrà a running.
- Interrupt: transizione in cui un processo ha eseguito per un po' di tempo (multitasking), CPU viene ceduto ad un altro per un momento e quindi il processo che prima usava quella CPU subisce un interrupt (sarebbe meglio chiamarlo preemption: capacità del sistema di interrompere l'esecuzione di un processo in un'esecuzione per assegnare la CPU a un altro processo).
- Se un processo in stato "running" deve aspettare un evento I/O, si mette in uno stato chiamato "waiting" e non può andare avanti. Quando è in stato di attesa, la CPU non gli serve e quindi viene ceduto ad un altro in cui magari è stato "ready".
- Quando l'evento I/O si verifica, il processo dallo stato di waiting e torna allo stato "ready" ma non riceve subito la CPU perché potrebbe star effettuando altre operazioni.
- Una volta che il processo ha finito di effettuare le sue operazioni, il SO libererà tutte le risorse del processo all'interno SO in modo tale che possano essere usati per altri processi.

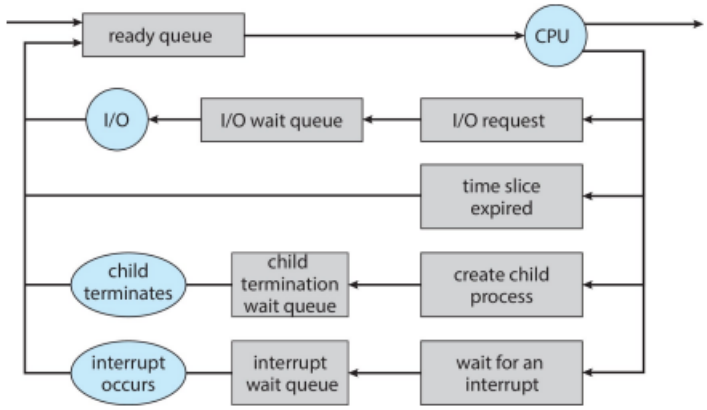
Lo scheduler della CPU

Si occupa delle transizioni.

Lo scheduler della CPU, o scheduler a breve termine sceglie il prossimo processo da eseguire tra quelli in stato ready ed alloca un core (CPU) libero ad esso. È una politica di quale processo mando in esecuzione adesso.
Mantiene diverse code di processi (Process Control Block: oggetto kernel che viene utilizzato per rappresentare un processo):

- Ready queue: processi residenti in memoria e in stato ready;
- Wait queue: code per i processi che sono residenti in memoria e in stato wait; una coda diversa per ciascun diverso tipo di evento di attesa.

Durante la loro vita, i processi (i loro PCB) migrano da una coda all'altra a seconda dello stato del processo stesso.



Quando un processo viene ammesso e quindi entra in stato "ready", viene messo nella ready queue. Dopo un po' viene estratto dalla ready queue e gli viene assegnato la CPU. Una volta finito il "time slice expired" che è la preemption, viene direttamente rimesso nella ready queue. Stessa cosa viene effettuato per l'attesa di I/O. Se fa una fork, si mette in wait per l'attesa del processo figlio, dove il processo padre non prosegue e aspetta che il figlio termini.

Digressione sulla "time slice expired": Il concetto di "time slice expired" (scadenza del time slice) è legato all'allocazione del tempo di CPU nei sistemi operativi che supportano il multitasking o la multiprogrammazione. In tali sistemi, più processi possono essere in esecuzione contemporaneamente, e ciascun processo ottiene un periodo di tempo chiamato "time slice" o "quantum" durante il quale può eseguire istruzioni sulla CPU.

Quando si verifica la scadenza del time slice, significa che il periodo di tempo assegnato a un determinato processo è terminato. A questo punto, il sistema operativo interrompe l'esecuzione di quel processo e decide quale processo assegnare alla CPU successivamente. Questo processo di cambio di contesto è fondamentale per garantire che tutti i processi abbiano la possibilità di eseguire le loro istruzioni in modo equo e che nessun processo monopolizzi la CPU. L'utilizzo efficace del time slice è parte integrante della gestione della concorrenza e della multitasking nei sistemi operativi moderni.

Schemi di scheduling

Lo scheduler della CPU ha il compito di decidere a quale processo tra tutti quelli nella ready queue assegnare un core libero. Tali **decisioni di scheduling** possono essere effettuate in diversi momenti, corrispondenti a cambi di stato dei processi:

1. Quando un processo passa da stato running a stato waiting;
2. Quando un processo passa da stato running a stato ready;
3. Quando un processo passa da stato waiting a stato ready;
4. Quando un processo termina.

Se il riassegnamento viene fatto solo nelle situazioni 1 e 4 lo schema di scheduling è detto senza prelazione (non preemptive) o cooperativo, dal momento che un core è sempre liberato da un processo che volontariamente rinuncia ad esso.
Altrimenti è detto con prelazione (preemptive), dal momento che un core può essere anche liberato perché un core viene forzatamente sottratto dal kernel ad un processo che lo sta usando.

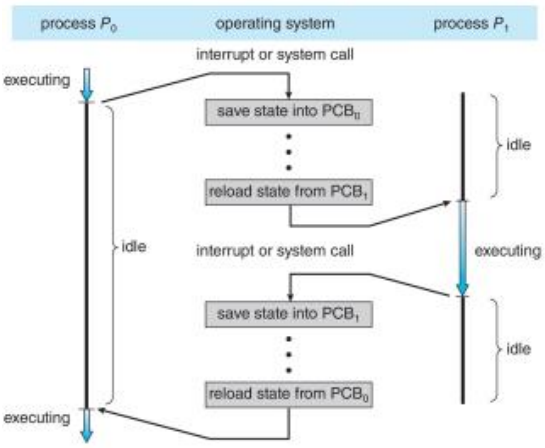
Lo schema di scheduling preemptive è più complicato da implementare:

- Che succede se due processi condividono dati?
- Che succede se un processo sta eseguendo in modalità kernel (system call o IRQ)?

Il processo può essere interrotto in maniera asincrona ed è molto difficile far s' che un processo possa essere interrotto in qualsiasi momento bloccato nel suo stato di esecuzione e dopodiché ripreso da quello stato e proseguire nella sua esecuzione come se non fosse mai stato interrotto prima. In uno schema cooperativo ci sono schemi precisi dove è possibile rinunciare la CPU (API in cui si metterà in attesa), mentre in uno schema con prelazione può essere interrotto in qualsiasi momento. C'è un problema non di politica ma di meccanismo. Qual è il meccanismo che permette di bloccare il suo stato, salvarlo da qualche parte in maniera che sia congelato nel tempo e in maniera che dopo possa essere rimesso in campo e farlo ripartire dallo stesso punto come se non fosse mai stato interrotto?

Commutazione di contesto e dispatcher

Quando il controllo della CPU deve passare da un processo ad un altro processo scelto dallo scheduler a breve termine avviene una commutazione di contesto (context switch). Il contesto è il processo che sta attualmente eseguendo su un certo core. Quando avviene un interrupt, viene invocata la routine di interrupt del timer e va a vedere se è scaduto il tempo. Quando scade il tempo, viene invocato lo scheduler e implementa una politica che decide quale processo deve essere eseguito. Quando lo scheduler ha deciso quale processo deve essere eseguito, evoca il dispatcher e gli dice "Cambia dal processo X al processo Y" e il dispatcher effettua questa operazione.



La commutazione di contesto viene effettuata dal dispatcher, che:

- Salva il contesto (stato, registri…) del processo da interrompere nel suo PCB;
- Carica il contesto del processo da eseguire dal suo PCB;
- Passa in modalità utente;
- Salta nel punto corretto del programma del processo selezionato (ossia, dove era stato precedentemente interrotto).

Problema: in che ordine si salvano le cose? Qual è il punto giusto in cui salvare le cose? (Sarebbe alla fine, quando c'è proprio l'ultimo valore altrimenti salverei troppo presto).

Solitamente si decide di utilizzare una politica di fairness, permettendo a tutti i processi di effettuare la loro esecuzione equamente. Non esiste un processo al quale non do tempo e do lo stesso tempo a tutti.

Notare che il dispatcher implementa un tipico meccanismo, mentre lo scheduler implementa una tipica politica.

Latenza di dispatch

Definizione: La latenza di dispatch è il tempo impiegato dal dispatcher per fermare un processo ed avviarne un altro (tempo impiegato per effettuare una commutazione di contesto). La latenza di dispatch è puro overhead: non viene eseguito alcun lavoro utile (il lavoro utile è l'esecuzione di un programma utente). Più è complesso il sistema operativo, più è complesso il PCB (il contesto), maggiore è la latenza di dispatch. Alcuni processori offrono supporto speciale per minimizzare la latenza di dispatch (es. banchi di registri multipli).

Implementazione del multithreading

Thread a livello utente e kernel

Thread a livello utente: i thread disponibili nello spazio utente dei processi; sono quelli offerti dalle librerie di thread ai processi.

Thread a livello del kernel: i thread implementati nativamente dal kernel; sono utilizzati per strutturare il kernel stesso in maniera concorrente. Il kernel usa uno scheduler dei thread e utilizza i thread del kernel per strutturare sé stesso in maniera concorrente. È utile per sfruttare la presenza di più core all'interno del nostro sistema.

Modelli di supporto al multithreading

I thread a livello del kernel vengono utilizzati dalle librerie di thread per implementare i thread a livello utente di un certo processo. A tale scopo possono essere adottate diverse strategie (modelli di multithreading):

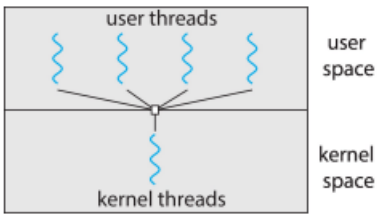
- Molti-a-uno: i thread a livello utente di un certo processo sono implementati su un solo thread a livello del kernel;
- Uno-a-uno: ogni thread a livello utente è implementato su un singolo, distinto thread a livello del kernel;
- Molti-a-molti: i thread a livello utente di un certo processo sono implementati su un insieme di thread a livello del kernel possibilmente inferiore di numero, e l'associazione thread utente / thread kernel è dinamica, stabilita da uno scheduler interno alla libreria di thread

Molti-a-uno

Ho un thread del kernel e tanti thread utente e ogni processo per il kernel è single threaded. La libreria del thread alterna i thread utente sull'unico thread del kernel che c'è. È di fatto single threaded, ma la libreria del thread implementa tanti thread e li alterna sull'unico thread che esiste.

Vantaggio: usabile su ogni sistema operativo (unica soluzione possibile se il sistema operativo non è multithreaded).
Svantaggi:

- Se un thread utente fa una chiamata di sistema bloccante blocca tutti i thread utente dello stesso processo;
- Non sfrutta la presenza di più core.
- È una soluzione semi-parallela.

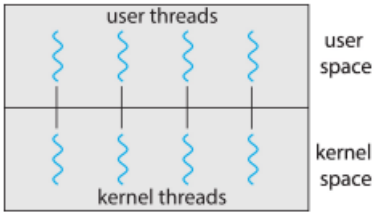


Poco usato in pratica.

Esempi: Solaris Green Threads oppure GNU Portable Threads.

Uno-a-uno

Modello più usato. Ogni thread utente viene creato un corrispondente thread del kernel.



Vantaggi:

- Permette un maggior grado di concorrenza;
- Permette di sfruttare il parallelismo nei sistemi multicore.

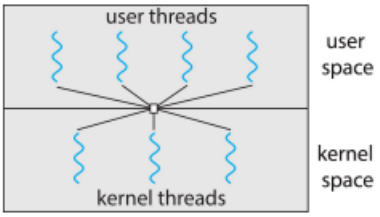
Svantaggi:

- Minore performance rispetto al modello molti-a-uno;
- Stress del kernel

Esempio: Linux, Windows

Molti-a-molti

Cerca di combinare i vantaggi dei modelli molti-a-uno e uno-a-uno. Cerca di ridurre il numero di thread del kernel necessari per schedulare i thread utente. I thread del kernel diventano come CPU virtuali.



Svantaggio: complesso da implementare (la libreria di thread deve dinamicamente alternare l'esecuzione dei thread a livello utente sui thread a livello del kernel disponibili).

Modello a due livelli: permette agli utenti di creare dei thread che hanno un mapping uno-a-uno con un thread a livello del kernel. Inoltre, un processo bloccante non blocca altri processi.

Esempio: Solaris, AIX

Thread control blocks

Attenzione! Nei kernel multithreaded, siccome ogni thread viene schedulato, lo scheduler non schedula solo i processi, ma sia processi che thread.

Nei kernel multithreaded il kernel mantiene delle strutture dati analoghe ai PCB, i thread control blocks (TCB); un TCB memorizza il contesto di un thread e le sue informazioni di contabilizzazione. Il PCB contiene solo le informazioni di contesto e contabilizzazione comuni (ad esempio lo spazio di memoria). Il PCB è di norma collegato ai TCB dei thread kernel utilizzati dal processo, e viceversa i TCB dei thread

kernel utilizzati da un processo sono collegati al PCB del processo.
Se un thread è kernel puro, allora non ha un PCB. Ma se siamo in un modello Uno-a-uno e il kernel thread corrisponde ad un user thread, allora si memorizza anche a che processo appartiene il thread.

Criteri di valutazione algoritmi di scheduling

Politiche che permettono di capire quale processo deve essere eseguito (PROCESSO NON THREAD).

Criteri di valutazione algoritmi di scheduling

Misure che servono per confrontare le caratteristiche dei diversi algoritmi (purtroppo non dipendono solo dall'algoritmo, ma anche dal carico).

Principali criteri:

- Utilizzo della CPU: % di tempo in cui la CPU è attiva nell'esecuzione dei processi utente (dovrebbe essere tra il 40% e il 90%, in funzione del carico);
- Throughput: # di processi che completano l'esecuzione nell'unità di tempo (dipende dalla durata dei processi);
- Tempo di completamento: tempo necessario per completare l'esecuzione di un certo processo (dipende da molti fattori: durata del processo, carico totale, durata dell'I/O…);
- Tempo di attesa: tempo trascorso dal processo nella ready queue (meglio del tempo di completamento, meno dipendente da durata del processo e dell'I/O);
- Tempo di risposta: negli ambienti interattivi, tempo trascorso tra l'arrivo di una richiesta al processo e la produzione della prima risposta, senza l'emissione di questa nell'output.

Obiettivi:

- Massimo utilizzo della CPU;
- Massimo throughput;
- Minimo tempo di completamento (medio);
- Minimo tempo di attesa (medio);
- Minimo tempo di risposta (medio)

Naturalmente nessun algoritmo di scheduling può ottimizzare tutti i criteri contemporaneamente.

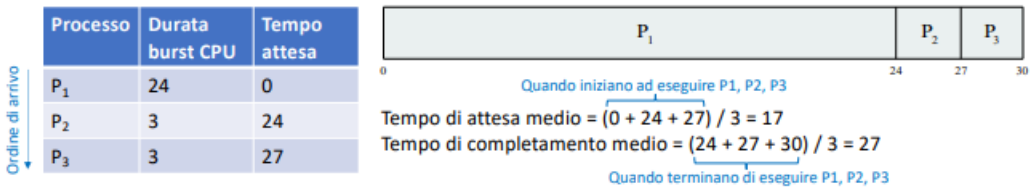
Principali algoritmi di scheduling

Scheduling in ordine di arrivo

Scheduling in ordine di arrivo, o first-come-first-served (FCFS): la CPU viene assegnata al primo processo che la richiede. I processi arrivano in ordine, e l'ordine di arrivo è il tempo in cui un processo viene ammesso nella ready queue.

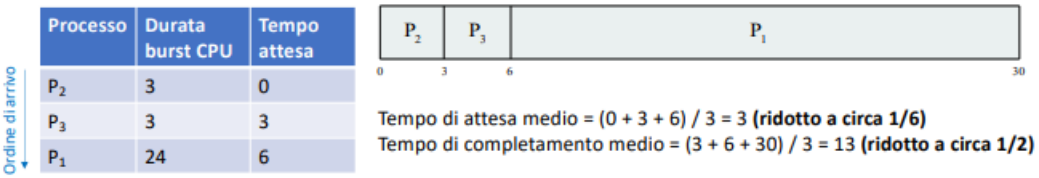
Vantaggio: implementazione molto semplice (coda FIFO, nessuna prelazione).

Svantaggio: tempo di attesa medio può essere lungo («effetto convoglio»).



Schema senza prelazione: il processo utilizza la CPU fino a quando non termina, non viene interrotto per far eseguire altri processi. Proviamo a confrontarlo con un altro algoritmo di scheduling che ha lo stesso carico ma diverso ordinamento.

Si crea l'effetto convoglio: se un certo processo è molto lungo, rallenta quelli dietro.



È un algoritmo di scheduling che però ha molti svantaggi.

Scheduling per brevità

Scheduling per brevità, o shortest-job-first (SJF): la CPU viene assegnata al processo che ha il successivo CPU burst più breve.
Vantaggi: implementazione quasi identica a FCFS, ma minimizza il tempo di attesa medio (è ottimale)
Svantaggio: di solito non si sa in anticipo qual è il processo che avrà il CPU burst più breve (quanto durerà il prossimo CPU burst?).

Ordine di arrivo

Processo	Durata burst CPU	Tempo attesa
P_1	6	0
P_2	8	6
P_3	7	14
P_4	3	21

Tempo di attesa medio FCFS = (0 + 6 + 14 + 21) / 4 = 10,25
Tempo di completamento medio FCFS = (6 + 14 + 21 + 24) / 4 = 16,25

Ordine per brevità

Processo	Durata burst CPU	Tempo attesa
P_4	3	0
P_1	6	3
P_3	7	9
P_2	8	16

Tempo di attesa medio SJF = (0 + 3 + 9 + 16) / 4 = 7
Tempo di completamento medio SJF = (3 + 9 + 16 + 24) / 4 = 13

Idea: riordinare i processi in base alla durata del burst di CPU.

Scheduling per SRTF

L'algoritmo **shortest-remaining-time-first** (SRTF) utilizza la prelazione per gestire il caso in cui i processi non arrivino tutti nello stesso istante: se nella ready queue arriva un processo con un burst più corto di quello running, quest'ultimo viene prelazionato dal nuovo processo.

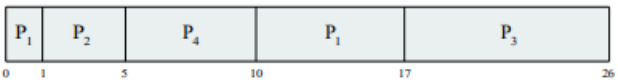
$$T\ attesa\ processo = ist\ terminazione\ processo - (T\ arrivo + durata\ burst)$$

$$T\ completamento\ processo = ist\ terminazione\ processo - T\ arrivo$$

Esempio:

Processo	Tempo di arrivo	Durata burst CPU
P_1	0	8
P_2	1	4
P_3	2	9
P_4	3	5

Determina l'ordine di arrivo



Tempo di attesa medio = ((17-8) + (5-5) + (26-11) +(10-8)) / 4 = 6,5
Tempo di completamento medio = ((17-0) + (5-1) + (26-2) +(10-3)) / 4 = 13

Scheduling circolare

Nello scheduling circolare, o round-robin (RR) ogni processo ottiene una piccola quantità fissata di tempo di CPU (quanto di tempo), di solito 10-100 millisecondi, per il quale può essere in esecuzione. Trascorso tale tempo il processo in esecuzione viene interrotto e messo in fondo alla ready queue, che è gestita in maniera FIFO. In tal modo la ready queue funziona essenzialmente come un buffer circolare, e i processi vengono scanditi dal primo all'ultimo, per poi ripartire dal primo nello stesso ordine. Timer che genera un interrupt periodico con periodo q per effettuare la prelazione del processo corrente (passaggio del processo da stato running a ready).

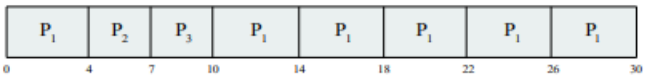
$$T\ attesa\ processo = ist\ terminazione\ processo - (T\ arrivo + durata\ burst)$$

$$T\ completamento\ processo = ist\ terminazione\ processo - T\ arrivo$$

Esempio:

Ordine di arrivo ↓

Processo	Durata burst CPU
P_1	24
P_2	3
P_3	3



q = 4
Tempo di attesa medio = ((30-24) + (7-3) + (10-3)) / 3 = 5,67
Tempo di completamento medio = (30 + 7 + 10) / 3 = 15,67

In questo esempio il tempo di arrivo è 0 per tutti i processi.

Caratteristiche del scheduling circolare

Se ci sono n processi nella ready queue e il quanto temporale è q:

- Nessun processo attende più di q (n - 1) unità di tempo nella ready queue prima di ridiventare running per un altro quanto di tempo (rispetto a SJF e SRTF non c'è bisogno di sapere la durata del burst);
- Ogni processo ottiene 1/n del tempo totale di CPU, in maniera perfettamente equa (rispetto a SJF e SRTF vengono ottenute solo q unità di tempo per volta); Se c'è un processo da schedulare, viene schedulato e quindi la CPU viene sempre utilizzato finché c'è un processo nella ready queue. È un algoritmo che è distribuisce in modo equo il tempo.

Comportamento in funzione di q:

Il quanto di tempo è un parametro di configurazione, possiamo aumentarlo (quanto è conveniente?) o diminuirlo.

- q elevato: se q è di regola maggiore della durata di tutti i burst, RR tende al FCFS (First Come First Served). RR diventa una coda dove il primo processo che viene schedulato è il primo in ordine di arrivo. Ma a quel punto non viene mai preempted perché il suo burst è più corto del quanto di tempo e quindi esegue per tutta la durata del suo burst. Fa cessare l'utilità del RR.
- q basso: deve comunque essere molto più lungo della latenza di dispatch, altrimenti questa sì «mangia» un tempo comparabile al tempo di esecuzione dei processi utente e l'utilizzo della CPU diventa inaccettabilmente basso. Fa entrare dei problemi di non idealità dell'algoritmo perché non stiamo tenendo conto di un elemento che è la latenza di dispatch, tempo che ci mette il dispatcher a cambiare contesto. Quando il quanto di tempo diventa troppo basso e diventa della durata del cambio di contesto, l'utilizzo della CPU diventa inaccettabilmente basso (Uso della CPU = Tempo rubato dalla latenza di dispatch). L'utilizzo della CPU è quindi del 50%.

Il q ideale sarebbe quindi sufficiente lungo da essere molto più lungo delle non-idealità ma altrettanto sufficientemente corto da essere più corto del medio burst di CPU dei processi.

Performance:

- Rispetto a SJF tipicamente RR ha un tempo di completamento medio più alto;
- Ma un tempo di risposta medio più basso (va bene per i processi interattivi);
- Il tempo di completamento medio non necessariamente migliora con l'aumento di q (è un comportamento non lineare che dipende tutto dal carico).

Tempo di completamento

Il tempo di completamento medio migliora se la maggioranza (~80%) dei CPU burst è più breve di q.

Con carichi strani (guarda la tabella), se proviamo a disegnare il tempo del completamento secondo RR, siamo arrivati al FCFS.

Scheduling con priorità

Nello scheduling con priorità ad ogni processo è associato un numero intero che indica la sua priorità.

Priorità in:

- **Unix**: priorità 1 > priorità 2 (Utilizzato in questo corso)
- **Windows**: priorità 1 < priorità 2

Viene eseguito il processo con priorità più alta, gli altri aspettano.

Può essere preemptive o no (Cooperativo: non rinuncia alla CPU fino a quando non lo fa volontariamente) (Se arriva un processo con una priorità più alta di quella attualmente in corso, viene eseguito prima quello con una priorità più alta).

Possono essere permessi più processi con pari priorità o no; in caso positivo occorre stabilire un secondo algoritmo di scheduling per arbitrare tra i processi a pari priorità (di solito si utilizza RR).

SJF è scheduling con priorità, dove la priorità è l'inverso della durata del CPU burst (più è corto il burst, maggiore è la priorità)

C'è il problema della **attesa indefinita (starvation):** un processo a priorità troppo bassa potrebbe non venir mai schedulato. (Se un processo ha una priorità particolarmente bassa, non è garantito che possa andare in esecuzione).

Soluzione: modifica allo scheduling, aggiungendo invecchiamento (aging), ossia aumento automatico di priorità di un processo al crescere del tempo di permanenza nella ready queue.

Esempio di scheduling con priorità

Processo	Durata burst CPU	Priorità (UNIX)
P_1	10	3
P_2	1	1
P_3	2	4
P_4	1	5
P_5	5	2

Tempo di attesa medio = (0 + 1 + 6 + 16 + 18) / 5 = 8,2
Tempo di completamento medio = (1 + 6 + 16 + 18 + 19) / 5 = 12

Non c'è preemption perchè non c'è nessun algoritmo in esecuzione che abbia una priorità minore del successivo.

Esempio di scheduling con priorità + RR

Processo	Durata burst CPU	Priorità (UNIX)
P_1	4	3
P_2	5	2
P_3	8	2
P_4	7	1
P_5	3	3

Ordine di arrivo

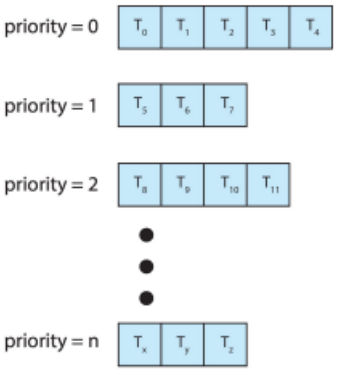
q = 2 per processi con stessa priorità
Tempo di attesa medio = ((26 - 4) + (16 - 5) + (20 - 8) + (7 - 7) + (27 - 3)) / 5 = 13,8
Tempo di completamento medio = (26 + 16 + 20 + 7 + 27) / 5 = 19,2

Code multilivello con retroazione

Permette di integrare insieme e fare un'unica politica di scheduling per tutto un insieme di processi all'interno di un sistema operativo.

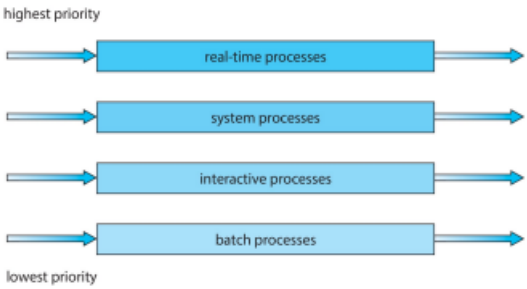
Code multilivello

Lo scheduling con priorità usa ready queue separate per ogni livello di priorità. Viene schedulato il processo nella coda non vuota con priorità maggiore.



Nello scheduling con priorità in base a quale criterio possiamo assegnare le priorità ai processi?

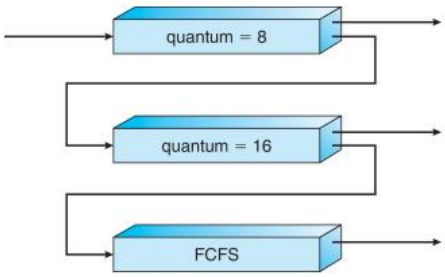
Un criterio comunemente usato è quello di basare la priorità di un processo sul suo tipo:



- **Priorità più alta ai processi interattivi o cyber-fisici (o real-time)** che devono reagire rapidamente all'I/O (tipicamente I/O bound, burst di CPU molto corti);
- **Priorità più bassa ai processi (di sistema)** che effettuano lunghe computazioni, o processi batch (tipicamente CPU bound, burst di CPU molto lunghi).

Code multilivello con retroazione

La priorità di un processo può variare dinamicamente: è sufficiente spostarlo da una ready queue ad una certa priorità ad un'altra. L'invecchiamento è di solito implementato in questo modo (spostamento di un processo verso una coda a priorità più alta). Anche la identificazione di un processo come I/O bound o CPU bound può essere effettuata dinamicamente, e determinare un cambio di priorità, stavolta con uno spostamento verso il basso. Uno scheduler di questo tipo è detto con code multilivello con retroazione ed è stato introdotto nel sistema operativo CTSS. Questo tipo di scheduler è adottato da moltissimi sistemi operativi moderni (Windows, macOS, FreeBSD, Solaris).



Code:

- Q_0: RR con q0 = 8 msec
- Q_1: RR con q1 = 16 msec
- Q_2: RR con q2 = ∞ (ossia FCFS)

Q_0 ha priorità alta, Q_1 intermedia, Q_2 bassa
Alla creazione un processo entra in Q_0.
Se quando un processo diventa running non termina il suo burst entro il suo quanto di tempo, avviene preemption e viene messo all'inizio della coda immediatamente più bassa (Sempre più CPU bound che I/O bound), altrimenti rimane nella sua coda.
Per evitare starvation l'invecchiamento sposta i processi in direzione opposta, verso le code più alte, se passano troppo tempo in una coda senza essere eseguiti.
Effetto ricercato: mantenere i processi I/O bound (con burst della CPU corti) nelle code ad alta priorità e quelli CPU bound (con burst della CPU lunghi) nelle code a bassa priorità. Unisce priorità, RR, FCFS.