Processi e thread: la struttura

Pietro Braione

Reti e Sistemi Operativi – Anno accademico 2025-2026

Argomenti

- Multiprogrammazione e multitasking
- Implementazione dei processi
- Criteri di valutazione algoritmi di scheduling
- Principali algoritmi di scheduling
- Code multilivello con retroazione

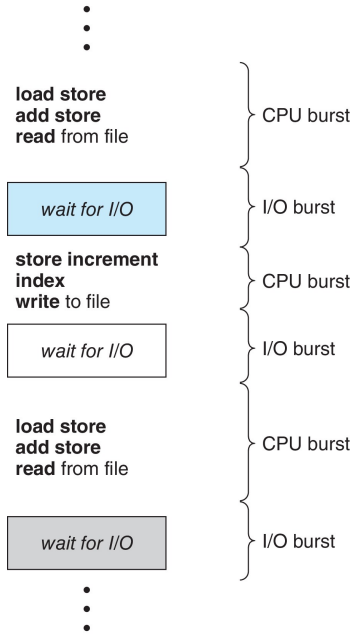
Multiprogrammazione e multitasking

Multiprogrammazione e multitasking

- Due obiettivi dei sistemi operativi:
  - **Efficienza**: mantenere impegnata la (o le) CPU il maggior tempo possibile nell'esecuzione dei programmi (se ci sono programmi da eseguire)
  - **Reattività**: dare l'illusione che ogni processo progredisca continuamente nella propria esecuzione, come se avesse una CPU dedicata; questo è particolarmente importante per i programmi interattivi, che devono reagire in tempi accettabili quando ricevono un input utente
- Le due tecniche adottate nei sistemi operativi per ottenere questi due obiettivi sono la **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 si che un programma interattivo possa reagire agli input utente in un tempo accettabile
  - Notare che il multitasking non è una tecnica rilevante per i sistemi puramente batch

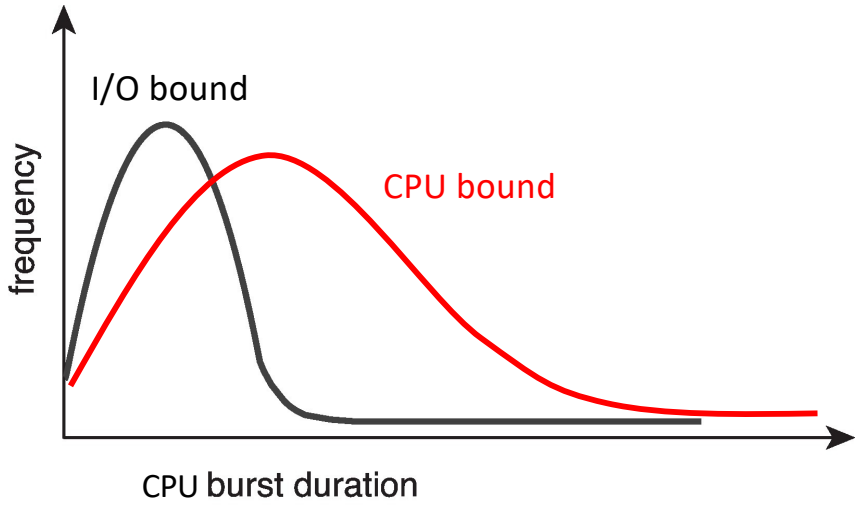
Burst CPU e I/O

- 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



Distribuzione delle durate dei burst della CPU

- 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
- Programma con prevalenza di CPU (**CPU bound**):
  - Elevato numero di burst CPU lunghi
  - Ridotto numero di burst CPU brevi
  - Tipico dei programmi batch
- 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



Multiprogrammazione

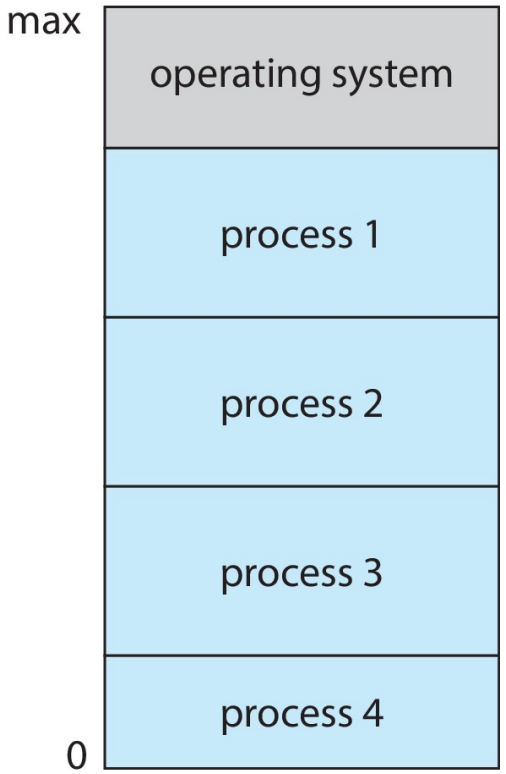
- Quando una CPU non è impegnata ad eseguire un processo, il sistema operativo seleziona un processo non in esecuzione e gli assegna la CPU
- Quando un processo non può proseguire l'esecuzione (ad es. perché deve attendere il termine dell'input di dati che gli servono per proseguire), la sua CPU viene 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
- Il sistema operativo mantiene in memoria le immagini di tutti i processi da eseguire

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
- 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 si che i programmi batch, che sono CPU-bound, non monopolizzino la CPU a discapito dei programmi interattivi, che sono I/O-bound

Multiprogrammazione e memoria

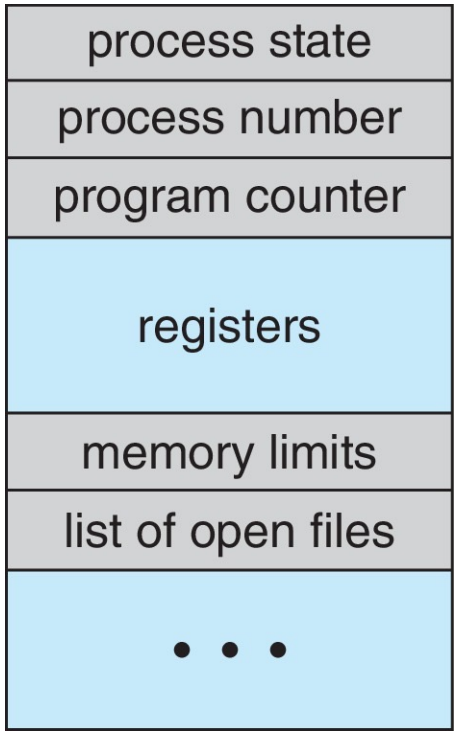
- La multiprogrammazione richiede che tutte le immagini di tutti i processi siano in memoria perché questi possano essere eseguibili
- Se i processi sono troppi la tecnica dello **swapping** può essere usata per spostare le immagini dentro/fuori dalla memoria
- Se l'immagine di un processo è troppo grande, la **memoria virtuale** è un'ulteriore tecnica che permette di eseguire un processo la cui immagine non è completamente in memoria
- Queste tecniche aumentano il numero di processi che possono essere eseguiti in multiprogrammazione, ossia il **grado di multiprogrammazione**
- Parleremo più avanti di tutti questi aspetti



Implementazione dei processi

Process Control Block (PCB)

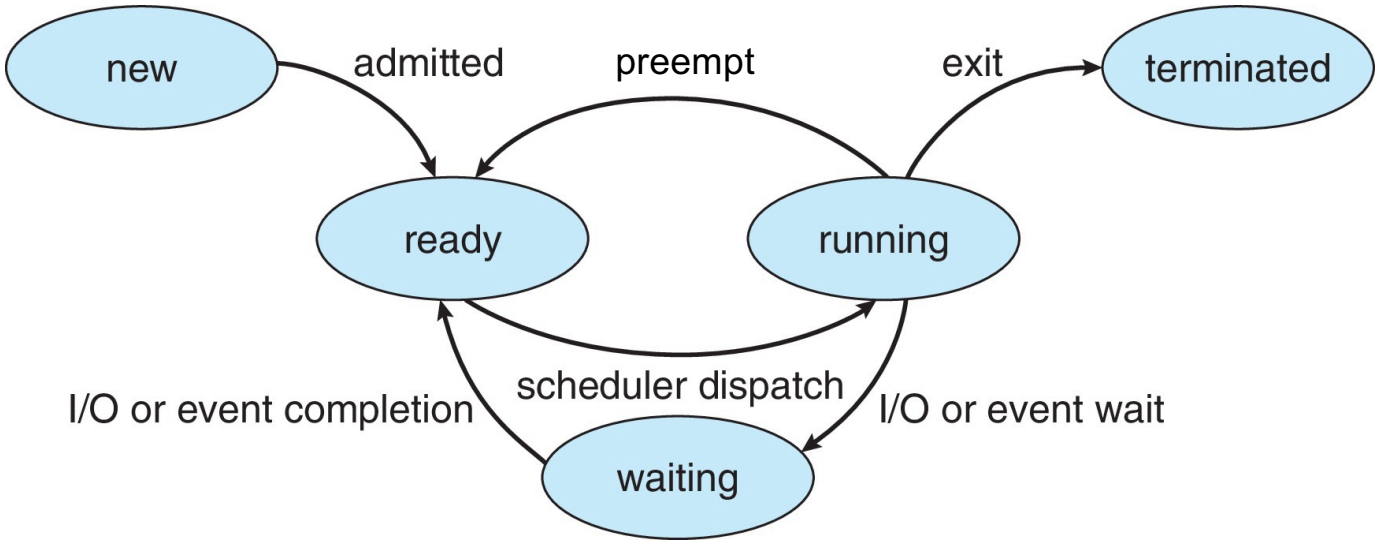
- Detto anche task control block
- È la struttura dati del kernel che contiene tutte le informazioni relative ad un processo:
  - Process state: ready, running…
  - Process number (o PID): identifica il processo
  - Program counter: contenuto del registro «istruzione successiva»
  - Registers: contenuto 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

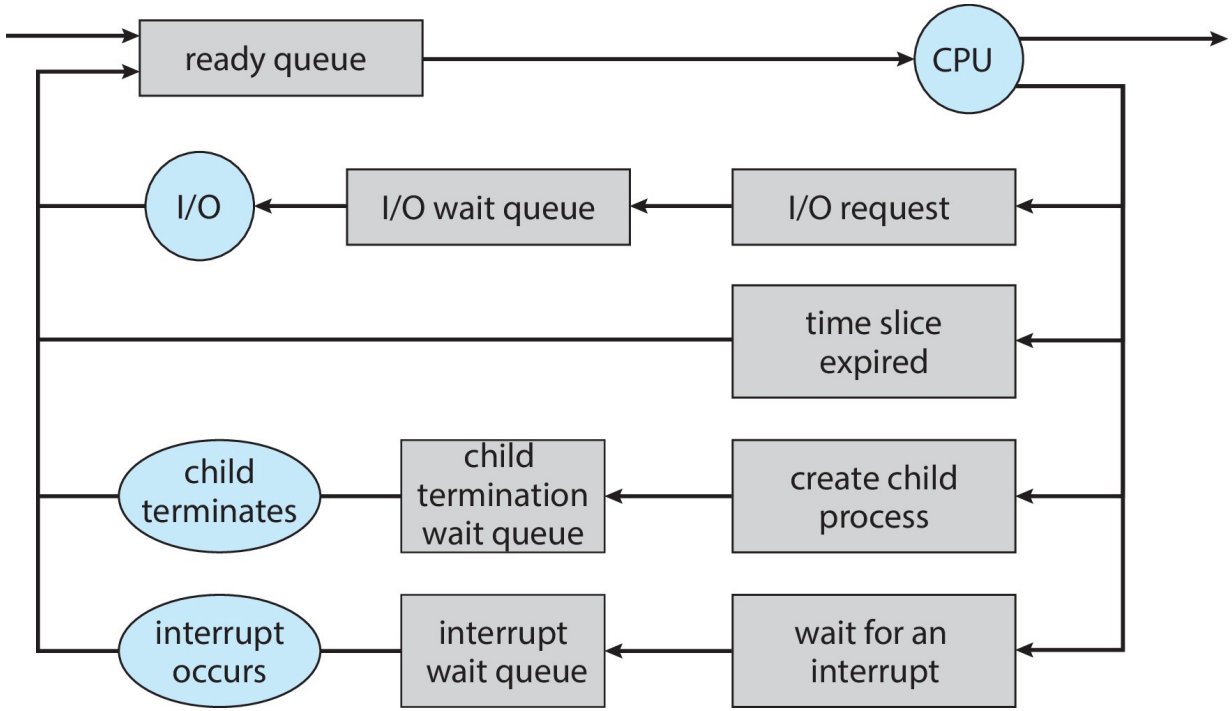
Diagramma di transizione di stato dei processi



Lo scheduler della CPU

- Lo **scheduler della CPU** , o **scheduler a breve termine** sceglie il prossimo processo da eseguire tra quelli in stato ready ed alloca un core libero ad esso
- Mantiene diverse code di processi:
  - **Ready queue**: processi residenti in memoria e in stato ready
  - **Wait queues**: 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

Rappresentazione delle code di scheduling



Schemi di scheduling (1)

- 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 effettuati 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 (**nonpreemptive**) 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

Schemi di scheduling (2)

- 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)?
- Di contro lo schema di scheduling cooperativo presuppone che nessun processo monopolizzi il processore:
  - Che succede se un processo non rinuncia al processore per lungo tempo?
  - E se contiene un bug che lo fa andare in loop infinito?
- I sistemi operativi possono differire notevolmente per come i loro scheduler decidono di scegliere il prossimo processo da mandare in esecuzione (*politica* di scheduling)

Commutazione di contesto e dispatcher

- Le informazioni che permettono di ripristinare l'esecuzione di un processo precedentemente interrotto vengono dette il **contesto** del processo
- 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**)
- La commutazione di contesto viene effettuata dal **dispatcher**, che:
  - Salva il contesto del processo da interrompere nel suo PCB (in particolare lo stack pointer)
  - Carica il contesto del processo da eseguire dal suo PCB (in particolare lo stack pointer)
  - Ritorna nel punto corretto del programma del processo selezionato (ossia, dove era stato precedentemente interrotto)
- Notare che il dispatcher implementa un tipico *meccanismo*, mentre lo scheduler implementa una tipica *politica*

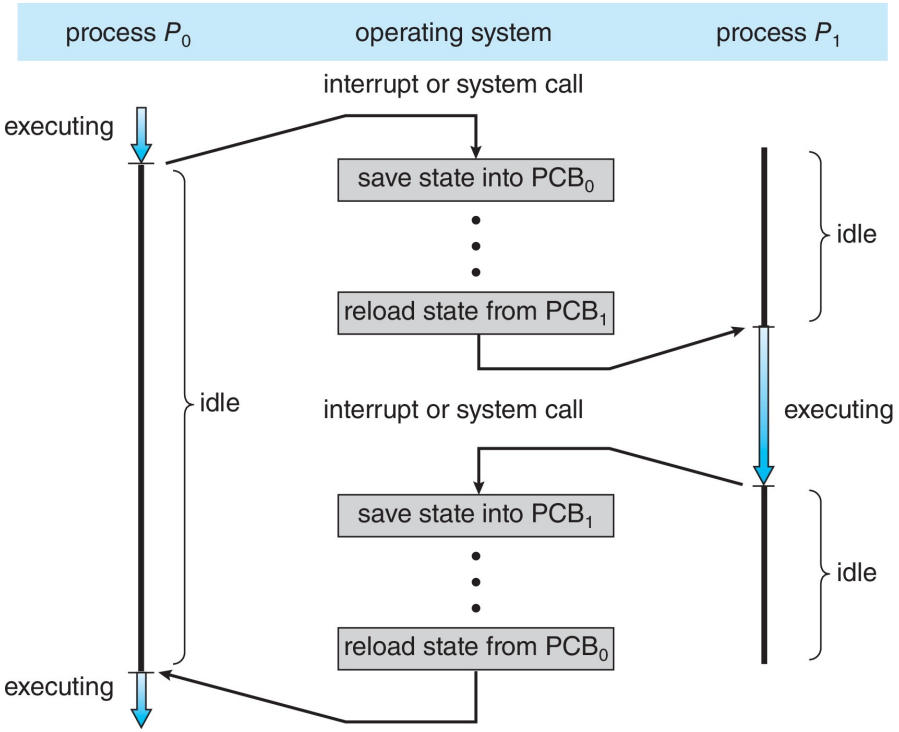
La «magia» della commutazione di contesto

```
#   void swtch(struct context *old, struct context *new);
#
# Save current registers in old. Load from new.

.globl swtch
swtch:
        sd ra, 0(a0)
        sd sp, 8(a0)
        sd s0, 16(a0)
        sd s1, 24(a0)
        ...
        sd s11, 104(a0)

        ld ra, 0(a1)
        ld sp, 8(a1)
        ld s0, 16(a1)
        ld s1, 24(a1)
        ...
        ld s11, 104(a1)

        ret
```



Latenza di dispatch

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

Criteri di valutazione algoritmi di scheduling

Confrontare algoritmi di scheduling

- Praticamente ogni sistema operativo ha i propri algoritmi di scheduling
- Questo è indizio del fatto che non esiste una politica di scheduling «migliore» di tutte le altre
- Si, ma come confrontiamo diverse politiche di scheduling?

Criteri di valutazione algoritmi di scheduling (1)

- 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
- A noi interessano essenzialmente tempo di completamento e di attesa, e su quelli svolgeremo i nostri esercizi

Criteri di valutazione algoritmi di scheduling (2)

- 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
- Vantaggio: implementazione molto semplice (coda FIFO, nessuna prelazione)
- Svantaggio: tempo di attesa medio può essere lungo («effetto convoglio»)

Ordine di arrivo

Processo	Durata burst CPU	Tempo attesa
P_1	24	0
P_2	3	24
P_3	3	27

Quando iniziano ad eseguire P1, P2, P3

Tempo di attesa medio = (0 + 24 + 27) / 3 = 17

Tempo di completamento medio = (24 + 27 + 30) / 3 = 27

Quando terminano di eseguire P1, P2, P3

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
- Vantaggio: Implementazione molto semplice (coda FIFO, nessuna prelazione)
- Svantaggio: tempo di attesa medio può essere lungo («effetto convoglio»)

Ordine di arrivo

Processo	Durata burst CPU	Tempo attesa
P_2	3	0
P_3	3	3
P_1	24	6

Tempo di attesa medio = (0 + 3 + 6) / 3 = 3 **(ridotto a circa 1/6)**
Tempo di completamento medio = (3 + 6 + 30) / 3 = 13 **(ridotto a circa 1/2)**

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?)
- 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

Scheduling per brevità: esempio

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

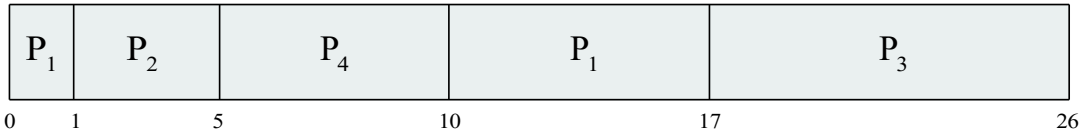
Tempo di attesa medio SJF = (0 + 3 + 9 + 16) / 4 = 7
Tempo di completamento medio SJF = (3 + 9 + 16 + 24) / 4 = 13

Shortest-remaining-time-first: esempio

- Con preemption e tempo di arrivo
- Il tempo di attesa di un processo è:
  istante terminazione processo - (tempo di arrivo + durata burst)
- Il tempo di completamento di un processo invece è:
  istante terminazione processo - tempo di arrivo

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

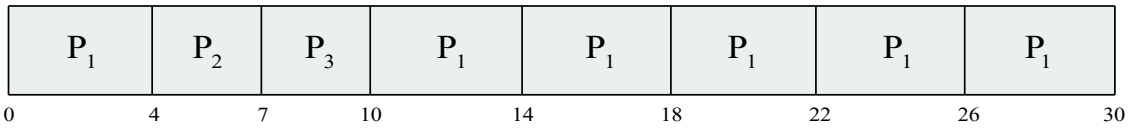
- Nello **scheduling circolare**, o **round-robin** (RR) ogni processo ottiene una piccola quantità fissata di tempo di CPU (**quanto di tempo**), di solito 10-100 millisecondi, per il quale può essere in esecuzione
- Trascorso tale tempo il processo in esecuzione viene interrotto e messo in fondo alla 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)

Scheduling circolare: esempio

- Ricordiamo che il tempo di attesa di un processo è:
  istante terminazione processo - (tempo di arrivo + durata burst)
- E il tempo di completamento di un processo è:
  istante terminazione processo - tempo di arrivo
- In questo esempio il tempo di arrivo è 0 per tutti i processi

Ordine di arrivo

Processo	Durata burst CPU
P_1	24
P_2	3
P_3	3



q = 4

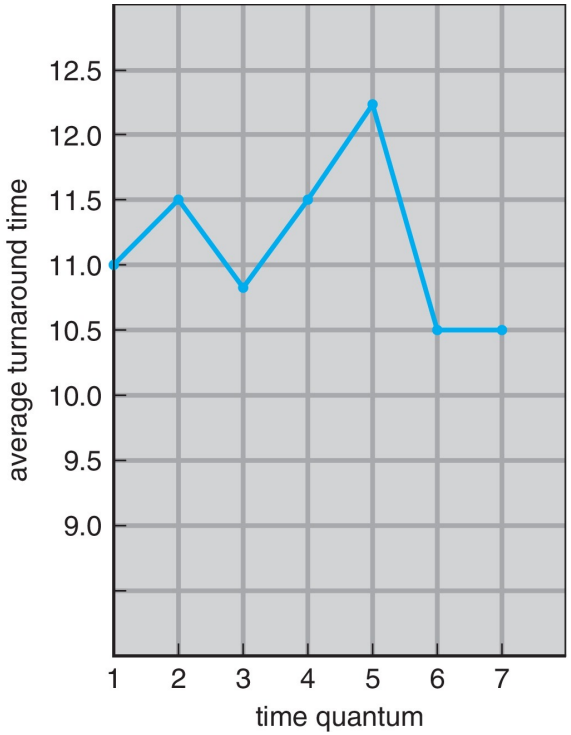
Tempo di attesa medio = ((30-24) + (7-3) + (10-3)) / 3 = 5,67

Tempo di completamento medio = (30 + 7 + 10) / 3 = 15,67

Scheduling circolare: caratteristiche

- 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)
- Comportamento in funzione di q:
  - q elevato: se q è di regola maggiore della durata di tutti i burst, RR tende al FCFS
  - q basso: deve comunque essere molto più lungo della latenza di dispatch, altrimenti questa si «mangia» un tempo comparabile al tempo di esecuzione dei processi utente e l'utilizzo della CPU diventa inaccettabilmente basso
- 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

Tempo di completamento scheduling circolare



process	time
P_1	6
P_2	3
P_3	1
P_4	7

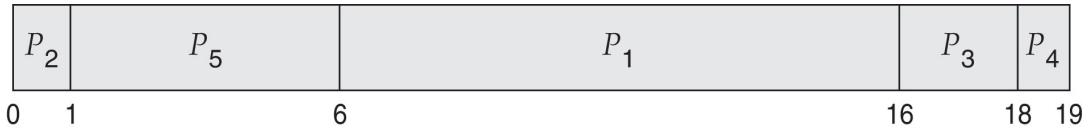
Il tempo di completamento medio migliora se la maggioranza (~80%) dei CPU burst è più breve di q

Scheduling con priorità

- Nello scheduling con priorità ad ogni processo è associato un numero intero che indica la sua priorità
- Viene eseguito il processo con priorità più alta, gli altri aspettano (in Unix-like numero più basso = priorità più alta, in Windows il contrario)
- Può essere preemptive o no
- 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
- Problema della **attesa indefinita** (**starvation**): un processo a priorità troppo bassa potrebbe non venir mai schedulato
- Soluzione: **invecchiamento** (**aging**), ossia aumento automatico di priorità di un processo al crescere del tempo di permanenza nella ready queue

Scheduling con priorità: esempio

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

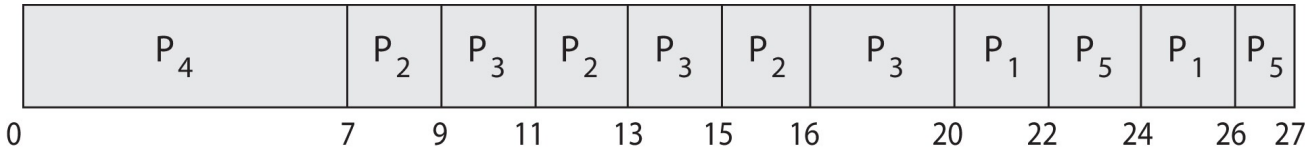


Tempo di attesa medio = (0 + 1 + 6 + 16 + 18) / 5 = 8,2
Tempo di completamento medio = (1 + 6 + 16 + 18 + 19) / 5 = 12

Scheduling con priorità + RR: esempio

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



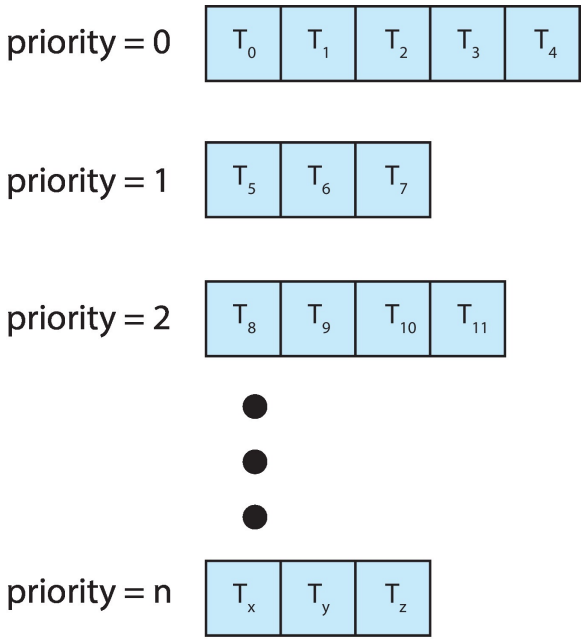
q = 2 per processi con stessa priorità

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

- Lo scheduling con priorità usa ready queue code separate per ogni priorità
- Viene schedulato il processo nella coda non vuota con priorità maggiore

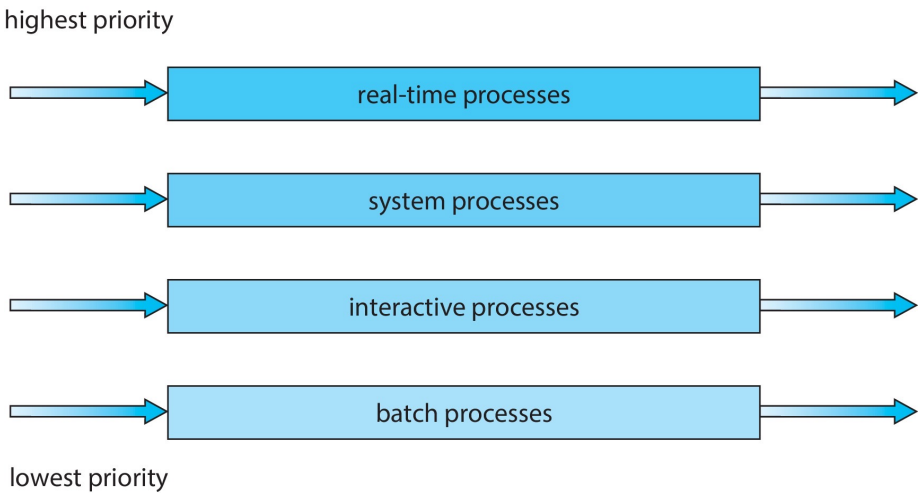


Code multilivello con retroazione

Una soluzione complessa

- Gli algoritmi SJF e SRTF minimizzano il tempo di attesa medio, il che va bene per i processi CPU bound (batch)
- Il principale problema è che il sistema operativo non conosce qual è la durata dei burst di CPU, conoscenza necessaria per tali algoritmi
- L'algoritmo RR, d'altra parte, ha un ottimo tempo di risposta medio, il che va bene per i processi I/O bound (interattivi)
- Il principale problema è che RR ha un pessimo tempo di attesa medio, ed ancora, non conoscendo la durata dei burst di CPU, non siamo in grado di distinguere i processi I/O bound da quelli CPU bound
- Idea: cercare di imparare nel tempo se un processo è I/O o CPU bound e trattarlo diversamente

Code multilivello: priorità basata sul tipo di processo



- Viene utilizzato uno scheduling con priorità + RR (code multilivello)
- La priorità di un processo dipende dal suo tipo:
  - Priorità più alta ai processi interattivi o cyber-fisici che devono reagire rapidamente all'I/O (tipicamente I/O bound)
  - Priorità più bassa ai processi che effettuano lunghe computazioni, o processi batch (tipicamente CPU bound)

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
- Questo già avviene con l'invecchiamento (spostamento di un processo verso una coda a priorità più alta)
- Idea: l'identificazione dinamica di un processo come CPU bound determina un cambio di priorità, stavolta verso il basso
- Uno scheduler di questo tipo è detto con **code multilivello con retroazione**
  - Introdotto nel sistema operativo CTSS (Corbatò et al., 1962)
  - Questo tipo di scheduler è adottato da moltissimi sistemi operativi moderni (Windows, macOS, FreeBSD, Solaris)

Code multilivello con retroazione: esempio

- Code:
  - Q_0: RR con q_0 = 8 msec
  - Q_1: RR con q_1 = 16 msec
  - Q_2: RR con q_2 = ∞ (ossia FCFS)
- Q_0 ha priorità alta, Q_1 intermedia, Q_2 bassa
- Alla creazione un processo entra in Q_0
- Se un processo non termina il suo burst entro il suo quanto di tempo viene messo nella coda immediatamente più bassa, 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