Sistemi Operativi I

Gabriele Fioco

a.a. 2024-2025

Indice

Parte I

Introduzione

1 Architetture dei sistemi operativi

1.1 Sistema monolitico

I sistemi monolitici sono sviluppati come un unico blocco che esegue tutte le funzioni del sistema operativo. Non vi è alcun ordinamento tra le funzioni del sistema: tutte le funzioni e le strutture dati sono accessibili da qualsiasi punto del kernel. Ogni funzione risiede in un singolo file binario in esecuzione in singolo spazio di indirizzamento

Manutenzione ed espansione sono molto difficili: un bug in una funzione potrebbe causare il blocco dell'intero sistema, contemporanemaente trovarlo e risolverlo potrebbe essere difficile. Il vantaggio principale è che il sistema è compatto e quindi veloce, con poco overhead per le chiamate di sistema.

1.2 Sistema gerarchico

Il sistema a struttura gerarchica organizza su livelli gerarchici diversi le funzioni. Una funzione di un certo livello può chiamare solo le funzioni di livello inferiore. Non vi è comunque una separazione tra le componenti del SO.

È facile identificare le dipendenze delle gerarchie ma la manutenzione è difficile. Inoltre non scala bene con la complessità dell'hardware.

1.3 Sistema stratificato

Un sistema modulare è scomposto in moduli separati, piccoli e con un compito specifico. L'insieme dei moduli è il kernel. Un modo di creare un sistema modulare è il sistema stratificato: il sistema è scomposto in un certo numero di livelli, il più basso è l'hardware, il più alto l'interfaccia utente. Ogni modulo implementa un componente e nasconde i propri dettagli implementativi, fornendo solo un'interfaccia ben definita a cui possono accedere i livelli superiori.

Il vantaggio principale è che lo sviluppo e il debug del sistema sono facilitati: quando si implementa un nuovo livello, si presume che quelli sottostanti funzionino correttamente, poiché sono già stati testati; e dopo aver implementato e testato anche il nuovo livello, si può passare a un livello superiore in modo analogo. Le prestazioni sono però limitate poichè per accedere ad un livello inferiore occorre attraversare i livelli intermedi. Inoltre è difficile identificare i livelli.

1.4 Sistema microkernel

I sistemi a microkernel sono progettati inserendo nel kernel soltanto le funzioni indispesabili, implementando le altre come servizi a livello utente. Il risultato è un kernel più piccolo. Tipicamente il kernel contiene funzioni di gestione dei processi, comunicazione interprocesso e gestione della memoria.

Un microkernel quindi permette la comunicazione tra i client e i servizi a livello utente, tipicamente con un sistema di scambio di messaggi.

Un primo vantaggio del sistema microkernel è che le funzioni sono a livello utente e quindi sono facilmente intercambiabili, senza passare per il kernel. Vi è inoltre la massima separazione meccanismi e politiche. È sicuro e affidabile: se un servizio fallisce, basta riavviare questi poichè si trova in spazio utente. Le performance di questi sistemi soffrono dell'overhead causato dal sistema di scambio di messaggi che consuma memoria e tempo di esecuzione a causa dello spazio per salvare i messaggi e dai frequenti cambi di contesto.

1.5 Sistema a moduli funzionali

I sistemi a moduli funzionali utilizzano moduli kernel caricabili (Loadable Kernel Modules, LKMs). Il kernel comprende una serie di servizi di base, e può fornirne altri collegando dei moduli sia durante l'avvio che durante l'esecuzione. L'idea alla base, dunque, è fornire servizi di base, mentre gli altri moduli possono essere aggiunti dinamicamente. Così facendo non è necessario ricompilare il kernel per ogni nuovo servizio.

Quest'approccio coinvolge l'uso di tecniche di programmazione a oggetti, ed è il più utilizzato. Come nel modello a strati, le implementazioni di ogni modulo sono nascoste agli altri ma hanno interfacce ben definite per comunicare, tuttavia ogni modulo può chiamare qualunque altro modulo. La comunicazione è diretta, migliorando così le prestazioni rispetto all'approccio a microkernel. Gli altri vantaggi di quest'implementazione sono gli stessi dei sistemi a microkernel.

1.6 Sistema a macchine virtuali

Nei sistemi a macchine virtuali il kernel fornisce una copia esatta dell'hardware sottostante ad ogni macchina virtuale eseguita in modalità utente, costruendo una gerarchia di macchine astratte. Ogni macchina virtuale esegue un SO. L'utente ha l'iillusione di avere una macchina propria. Il SO operativo di base schedula la CPU e utilizza la memeria virtuale per suddividere tempo di CPU e memoria tra le VM, il file system fornisce i dischi virtuali, lo spooling le periferiche.

Ogni VM è isolata dalle altre e dalla macchina fisica, dunque è facile testare modifiche senza causare l'arresto del sistema. È però difficile condividere dati tra VM, due soluzioni sono creare un disco virtuale condiviso o una rete virtuale. Lo svantaggio principale sono il calo delel prestazioni dovuto alla virtualizzazione, una richiesta al SO deve infatti passare prima al kernel della VM poi al software di controllo della VM e infine al kernel della macchina fisica.

Parte II

Processi

2 Processi

2.1 Introduzione

Un **processo** è un programma in memoria in esecuzione.

Lo **stato di evoluzione della computazione** è un insieme di tutti i valori da cui dipende l'evoluzione della computazione del processo, data da:

- Area dati statica
- Stack
- Heap
- Registri del processore
- Informazioni per la gestione di chiamate a procedura: stack pointer, base pointer, indirizzo di ritorno
- Program counter

Durante l'esecuzione il processo può usare il processore, attendere l'uso del processore o attendere la disponibilità di una risorsa. Per questo è caratterizzato da uno stato. Lo **stato del processo** è la modalità di uso del processore da parte del processo. I possibili stati sono:

- Nuovo
- In esecuzione
- In attesa
- Pronto all'esecuzione
- Terminato

Si può disegnare il **diagramma dell'evoluzione dello stato del processo**. Il diagramma rappresenta una macchina a stati finiti, è un grafo orientato che ha come nodi lo stato del processo e archi le transizioni tra lo stato.

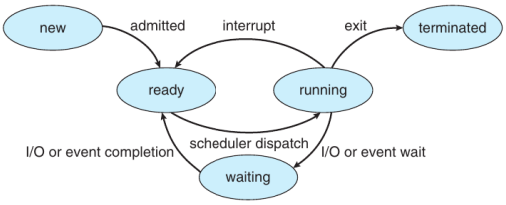

Figura 1: Diagramma dell'evoluzione dello stato del processo

Il sistema operativo rappresenta ogni processo attraverso una struttura dati detta **blocco di controllo del processo** (PCB). Questa serve a memorizzare tutti i dati per mandare in esecuzione o riprendere l'esecuzione di un processo. Contiene:

- L'identificatore del processo
- Lo stato del processo
- Il program counter
- I registri della CPU
- Informazioni di schedulazione
- Informazioni di gestione della memoria
- Informazioni di contabilità (es. uso della CPU, limiti di tempo…)
- Informazioni di I/O (es. lista dei dispositivi I/O usati, lista dei file aperti…)

2.2 Gestione dei processi: code e schedulazione

Un processo entra nel sistema e viene messo in una **coda dei processi pronti**. Una volta uscito dalla coda dei processi pronti e quindi in esecuzione su un core questo può dover effettuare operazioni di I/O, aspettare un figlio o subire un'interruzione quindi essere inserito nella **coda di attesa**. Una volta finita l'attesa il processo viene reimmesso nella coda dei processi pronti, da cui verrà poi inserito di nuovo in esecuzione. Ne consegue che esiste una coda dei processi pronti e un insieme di code di attesa.

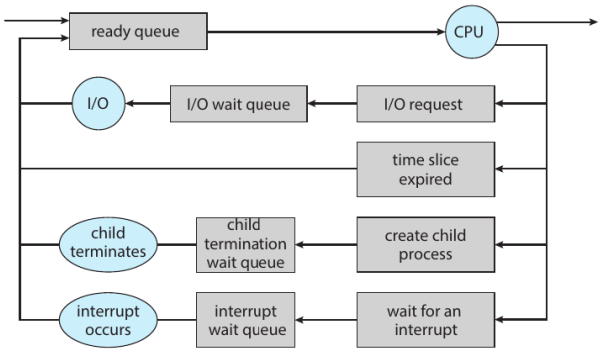

Figura 2: Diagramma di transizione tra le code

La coda dei processi pronti è generalmente realizzata tramite una lista concatenata. La testa della coda punta al primo PCB; ogni PCB contiene un puntatore al prossimo PCB della coda.

Lo **schedulatore della CPU** si occupa di selezionare un processo nella coda dei processi pronti e allocargli un core. Lo schedulatore deve essere eseguito molto spesso. Alcuni sistemi operativi implementano una forma intermedia di schedulatore, detto swapping, che toglie un programma dalla memoria al fine di ridurre il grado di multiprogrammazione. Lo swapping è utile anche se la memoria è piena e va liberata.

Il **dispatching** consiste nel prendere il primo processo nella coda dei processi pronti generata dallo schedulatore e metterlo in esecuzione su un core.

Un processo si dice: **I/O Bound** se esegue operazioni di I/O per la maggior parte del tempo; **CPU Bound** se esegue computazioni per la maggior parte del tempo.

2.3 Cambio di contesto

Un processo potrebbe essere sospeso per eseguire un interrupt, o far eseguire un altro processo. Quando questo avviene bisogna effettuare un cambio di contesto. L'operazione di cambio di contesto prevede di salvare lo stato di un processore sospendendolo, e successivamente di riesumarlo tramite un caricamento dello stato. Lo stato del processo è salvato dal PCB, ed è generalmente posto nello spazio riservato al kernel.

Il cambio di contesto si dice volontario quando il processo cede il controllo volontariamente; involontario quando il sistema operativo gli toglie la CPU.

2.4 Operazioni sui processi

2.4.1 Creazione

La creazione di un processo avviene con la chiamata a sistema `fork()`. Il processo che crea un altro processo è detto **padre**, i processi creati dal padre sono detti **figli**. A sua volta i figli possono diventare padri, creando una struttura ad albero. Quando il processo padre chiama `fork()` ottiene il pid del figlio.

I sistemi operativi associano ad ogni processo un intero univoco detto **pid** (process id), utilizzato come indice per ottenere informazioni sui processi dal kernel. Nei sistemi UNIX è presente un processo di init con pid 1, padre di tutti gli altri processi e quindi radice dell'albero.

Le risorse del figlio possono essere:

- indipendenti dal padre, ottenute dal sistema operativo;
- parzialmente condivise col padre;
- condivise col padre;
- ottenute con passaggio di dati di inizializzazione.

Quando un processo padre crea un figlio, potrebbero esserci due possibilità di esecuzione:

- Il padre continua l'esecuzione concorrentemente al figlio
- Il padre aspetta che il figlio termini l'esecuzione

Tenendo conto che lo spazio di indirizzamento del figlio è sempre distinto dal padre, vi sono due modalità di indirizzamento possibili:

- Il figlio è un duplicato del padre, dunque ha stesso programma e stessi dati
- Il figlio carica un nuovo programma

2.4.2 Terminazione

La chiamata a sistemi `exit()` termina un processo e restituisce al padre un valore intero, passato alla chiamata. La chiamata è implicita quando le istruzioni terminano. Il padre ottiene il valore ritornato dal figlio attraverso la chiamata a sistema `wait()`. Quando un processo termina tutte le sue risorse sono deallocate.

Un processo può terminare un altro processo attraverso una system call. Alcuni sistemi operativi non ammettono l'esistenza di un figlio se il padre è terminato; questo fenomeno è chiamato **terminazione a cascata**: la terminazione del padre causa la terminazione di tutti i figli.

Un processo padre potrebbe aspettare che il figlio termini attraverso la chiamata `wait()` che restituisce il pid del figlio e che prende un puntatore a intero in cui inserire il valore ritornato dal figlio.

Quando un figlio termina l'esecuzione questi rimane nella tabella dei processi fino a che il padre chiama `wait()` al fine di restituire il valore ritornato dal figlio. Durante questo lasso di tempo il processo figlio è detto zombie. Potrebbe accadere che il padre termini senza chiamare `wait()`, in questo caso il processo figlio è detto orfano. Questo probema comunque è risolto dal processo di init che controlla periodicamente i processi orfani e chiama `wait()` per terminarli.

Un processo potrebbero terminare in caso di anomalia per:

- Eccessivo uso di risorse
- Compito non necessario
- Terminazione a cascata
- Il padre ha terminato, e il sistema operativo non permette al figlio di continuare senza il padre

3 Thread

3.1 Introduzione

Un **thread** è l'unità base di utilizzo della CPU, rappresenta un flusso di controllo dell'esecuzione di istruzioni di un programma. Comprende un program counter, dei registri e uno stack. Un processo multithreading è formato da più thread che cooperano e condividono alcuni dati.

I benefici principali del multithreading sono:

- Reattività: utile in particolare per le interfacce utente.
- Condivisione delle risorse: i thread condividono lo spazio di indirizzamento con il processo principale.

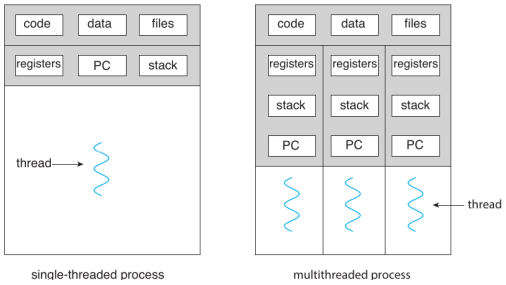

Figura 3: Concetto di thread

- Economia: poichè i thread condividono alcune risorse, il cambio di contesto è più economico.
- Scalabilità e parallelismo: nei sistemi multiprocessore i thread possono essere eseguiti in parallelo.

Il supporto al multithreading può avvenire a livello utente con delle librerie o a livello kernel con chiamate a sistema. I thread a livello utente vengono gestiti senza il supporto del sistema operativo.

3.2 Modelli multithread

Thread utente e thread kernel hanno una relazione, descritta di seguito in base a diversi modellis.

Modello molti a uno Nel modello multi a uno ogni thread di un processo è mappato a un solo kernel del sistema operativo. La gestione avviene solo a livello utente ed è dunque efficiente. Di contro non è possibile l'esecuzione parallela dei thread e se uno di questi dovesse effettuare una chiamata a sistema che blocca l'esecuzione, bloccherebbe gli altri.

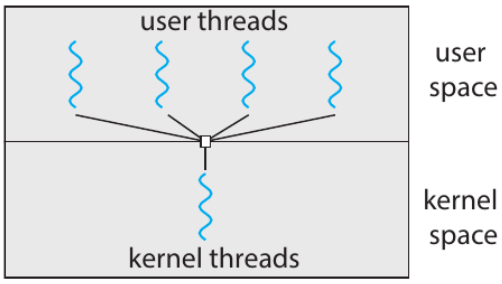

Figura 4: Modello molti a uno

Modello uno a uno Nel modello uno a uno ogni thread utente è mappato a uno e un solo thread kernel. Garantisce il parallelismo e permette a un thread di interrompere l'esecuzione senza bloccare gli altri. Di contro vengono creati molti thread kernel con conseguente overhead.

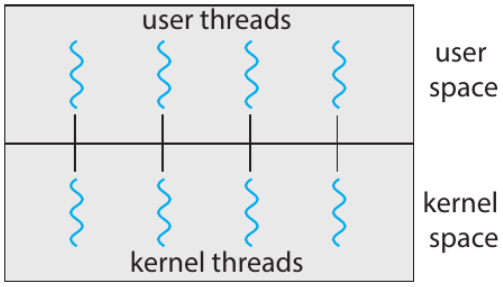

Figura 5: Modello uno a uno

Modello molti a molti Nel modello molti a molti, molti thread utente vengono instradati a un numero minore o uguale di thread kernel. Vi è il massimo sfruttamento delle architetture multiprocessore.

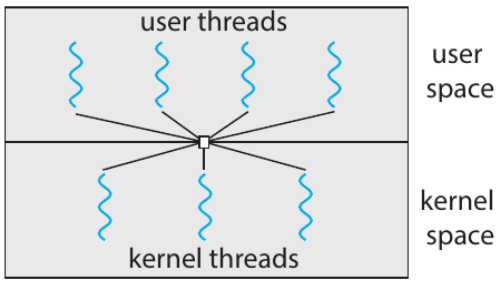

Figura 6: Modello molti a molti

Modello a due livelli Il modello a due livelli instrada molti thread utente a un numero minore o uguale di thread kernel, ma permette anche a un thread utente di essere mappato su un solo thread kernel.

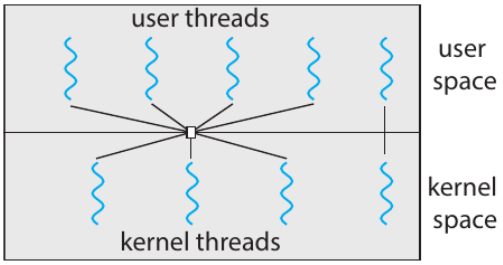

Figura 7: Modello a due livelli

Ricapitolando, nel modello molti a uno il programmatore è libero di creare tutti i thread che vuole ma senza esecuzione parallela. Nel modello uno a uno non vi sono i problemi del molti a uno ed è garantito il parallelismo, ma il programmatore deve stare attento a non creare troppi thread. Il modello molti a molti non ha nessuno di questi problemi, il programmatore può creare quanti thread vuole ed eseguirli in parallelo. Questi è però

molto difficile da implementare, e il gran numero di processori presenti sui sistemi moderni permette di creare molti thread. Molti sistemi operativi, inclusi Linux e Windows, utilizzano il modello uno a uno.

3.3 Modelli di cooperazione tra thread

I thread cooperano organizzando le proprie interazioni secondo tre modelli:

Thread simmetrici Tutti i thread possono svolgere lo stesso insieme di attività, e arrivata una richiesta ne può essere attivato uno qualunque.

Thread gerarchici Vi è una divisione in due livelli tra thread coordinatori e lavoratori. I primi coordinano i lavori (chi fa cosa, come e quando), ricevendo le richieste esterne e decidendo eventualmente a quale thread lavoratore indirizzarle; i secondi eseguono. Questo modello permette di avere il coordinatore virtualmente sempre reperibile, dal momento che evade rapidamente le richiete. A livello implementativo è conveniente mappare il thread coordinatore nel kernel, e i thread lavoratori nel livello utente.

Thread in pipeline Ogni thread è specializzato e svolge rapidamente una porzione del lavoro complessivo, passando il risultato al thread successivo. Vi è dunque una suddivisione del lavoro, come in una catena di montaggio. Il vantaggio principale è l'elevato throughput.

3.4 Gestione dei thread

3.4.1 Creazione

La semantica di `fork()` ed `exec()` potrebbe cambiare nei sistemi multithread. La maggior parte dei sistemi hanno deciso di mantenere due versioni di `fork()`: una che copia solo il thread che la chiama; una che copia l'intero processo. `exec()` mantiene lo stesso comportamento: una volta chiamato sostituisce l'intero processo, inclusi tutti i thread. Quindi se successivamente a una chiamata a `fork()` si chiama `exec()` ha senso utilizzare la versione che copia solo il thread corrente, in quanto copiarli tutti sarebbe inutile se poi vengono cancellati.

3.4.2 Gestione dei segnali

Un segnale notifica un evento a un processo. Un segnale può essere ricevuto in modo asincrono o sincrono. I segnali sincroni sono generati dal processo stesso che li riceve in seguito a determinate operazioni (es. accesso illegale a memoria); i segnali asincroni sono inviati da un processo a un altro. I segnali vengono gestiti con un gestore di default definito dal kernel o da un gestore definito dall'utente.

Nei sistemi multithread è più complesso inviare segnali. Dato un segnale, ci si chiede a quale thread inviarlo. Vi sono diverse opzioni:

- Inviare il segnale solo al thread ricevente
- Inviare il segnale a più thread
- Inviare il segnale a tutti i thread del processo
- Assegnare uno specifico thread del processo come destinatario di tutti i segnali

3.4.3 Terminazione

Cancellare un thread vuol dire terminarne l'esecuzione prima che abbia concluso. Il thread da eliminare è detto target thread, la sua cancellazione può avvenire secondo due modalità:

- Cancellazione asincrona: un thread cancella il target thread immediatamente.
- Cancellazione differita: il target thread controlla periodicamente se vi è una richiesta di cancellazione e termina quando può terminare.

Il problema della cancellazione asincrona è che il thread potrebbe non liberare delle risorse necessarie, o potrebbe essere cancellato mentre sta aggiornando dei dati condivisi tra thread. Viceversa con la cancellazione differita il target thread può eseguire alcune operazioni prima di uscire, per terminare dolcemente.

3.4.4 Attivazione dei thread: processi leggeri

Nella gestione dei thread utente è necessario che sistema operativo e librerie si coordinino. A questo scopo nei modelli molti a molti e a due livelli il sistema operativo crea un intermediario detto processo leggero. La libreria vede questo processo leggero come un processo virtuale su cui appoggiarsi e creare thread. Ogni LWP è collegato a un thread utente; il kernel schedula sui LWP.

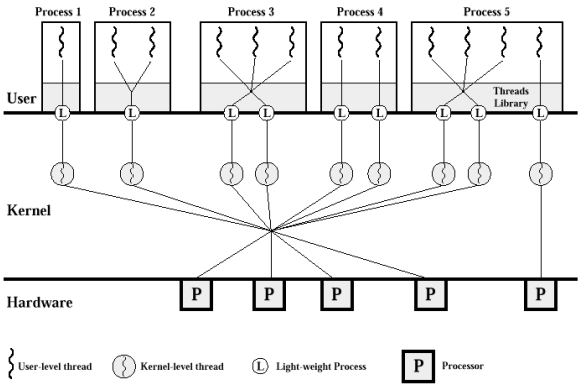

Figura 8: Modello di multithreading a due livello con LWP

4 Scheduling

4.1 Introduzione

La **multiprogrammazione** permette di caricare più programmi in memoria con l'obbiettivo di tenere la CPU sempre occupata; il **multi tasking** permette di avere una turnazione dei processi sul processore quando il processo in esecuzione è in attesa di una periferica. Multiprogrammazione e multitasking permettono al processore di gestire più flussi contemporaneamente ed eseguire programmi apparentemente in parallelo. Se non vi fosse multitasking, quando un processo entra nella coda di attesa la CPU non farebbe nulla.

Lo **schedulatore** della CPU deve gestire la turnazione dei processi sul processero, definendo

politiche di ordinamento e ordinando la coda dei processi pronti. Il **dispatcher** si occupa di mettere in esecuzione il primo processo della coda: questi deve cambiare contesto, mettere l'esecuzione in modalità utente e saltare al punto giusto del processo appena riesumato. Il tempo per eseguire queste operazioni è detto **latenza di dispatching**.

4.2 Proprietà

Lo scheduling si basa su una proprietà dei processi: l'esecuzione di un processo inizia con un ciclo di operazioni della CPU, è poi seguito da un ciclo di operazioni I/O, poi un ciclo di CPU e così via. Il ciclo di operazioni della CPU è detto **CPU burst**, il ciclo di operazioni I/O è detto **I/O burst**. Si osserva che la curva della durata dei CPU burst è esponenziale: vi sono un gran numero di CPU burst corti e un piccolo numero di CPU burst lunghi.

Lo schedulatore viene sempre eseguito quando un processo cambia stato da esecuzione a attesa e quando un processo termina. Se lo schedulatore viene attivato anche quando un processo cambia stato da attesa a pronto e quando un processo cambia stato da esecuzione a pronto, si dice che lo scheduling è **pre-emptive** (o con rilascio anticipato); viceversa si dice **non pre-emptive**. Uno schedulatore pre-emptive forza il rilascio della CPU assegnata a un processore, l'attivazione dello schedulatore si dice asincrona con l'evoluzione della computazione del processo; uno schedulatore non pre-emptive lascia il processore al processo fino a quando questi non lo rilascia di propria volontà, in questo caso l'attivazione dello schedulatore si dice sincrona con l'evoluzione della computazione del processo. Lo scheduling pre-emptive porta diverse problematiche in quanto potrebbe avvenire una corsa critica tra processi che condividono alcuni dati. Un kernel può quindi essere progettato pre-emptive o meno. Un kernel pre-emptive richiede meccanismi quali la mutua esclusione e la disabilitazione degli interrupt.

4.3 Livelli di schedulazione

Vi sono tre livelli di schedulazione:

- Schedulazione a breve termine: lo schedulatore a breve termine, detto schedulatore della CPU, schedula i processi in memoria centrale, già nella coda dei processi pronti. Il processo che è nella prima posizioe è quello inserito dal dispatcher. Questo tipo di schedulatore deve essere eseguito frequentemente per garantire la turnazione rapida ei processi, deve essere veloce per minimizzare il sovraccarico di gestione e deve usare algoritmi semplici.
- Schedulazione a lungo termine: lo schedulatore a lungo termine, detto job scheduler, ordina iinoltre riportare i processi dalla memoria centrale alla area di swap (swap out) e riportarli ancora dentro processi attivati nel sistema identificando il gruppo di processi che devono essere caricati in memoria centrale e posti nello stato di esecuzione. Il gruppo è costruito mescolando processi CPU bound e I/O bound. Questo tipo di schedulazione può assere anche assente o minimale, usa algoritmi complessi e deve essere eseguito poco frequentemente per non sovraccaricare il sistema.
- Schedulazione a medio termine: ordina i processi selezionato dallo schedulatore a breve termine per portarrli effettivamente in memoria centrale, scegliendone l'ordine. Potrebbe rimuovere temporanemanete rimuvoere i processi dalla memoria centrale e viceversa. Modifica dinamicamente il gruppo di processi caricati in memoria centrale e posti nello stato di pronto dallo schedulatore a lungo termine.

4.4 Criteri di valutazione

Ogni algoritmo di scheduling ha diverse caratteristiche, la scelta ricade in base a diversi criteri. L'algoritmo non cambia il tempo di esecuzione o di I/O ma qulelo speso nella coda dei processi pronti.

- **Utilizzo processore**: si vuole utilizzare il processore il più possibile. Se non si usa tutto il processore si perde capacità computazionale, quindi soldi.
- **Throughput**: numero di processi completati in una certa unità di tempo.
- **Turnaround time**: tempo tra l'inserimento di un processo nel sistema e la sua uscita, considerando i tempi morti. È dato dalla somma del tempo in esecuzione, tempo nella coda di attesa e tempo nella coda dei processi pronti.
- **Tempo d'attesa**: somma dei tempi spesi nella coda dei processi pronti.
- **Tempo di risposta**: tempo da quando viene fatta una richiesta alla prima risposta.

L'obbiettivo della schedulazione è massimizzare l'utilizzo del processore e il throughput, e minimizzare turnaround, tempo d'attesa e tempo di risposta. Generalmente si vogliono migliorare i valori medi, ma in alcuni casi si vogliono migliorare i valori di minimo/massimo. Nei sistemi interattivi è importante minimizzare la varianza tra i tempi d'attesa più che minimizzare il valore medio, così che l'utente abbia un sistema più predicibile.

4.5 Metodi di valutazione dei criteri

4.5.1 Modellazione deterministica

Una classe di valutazione è la valutazione analitica: data una formula, la si applica su un certo carico di lavoro per valutarne le performance.

La modellazione deterministica è una delle più semplici valutazioni analitiche. Con questo metodo si prende un certo carico di lavoro e si applica ciascun algoritmo, ottenendone le performance.

Questo modello è semplice, veloce e preciso, tuttavia analizza un carico di lavoro predeterminato senza generalizzazione. Pertanto viene utilizzato solo per fornire esempi, o per sistemi con carichi di lavoro costanti.

4.5.2 Modello a reti di code

Nella maggior parte dei sistemi il carico di lavoro non è costante, pertanto non si possono valutare le performance con un modello deterministico. Si può però determinare la distribuzione dei cicli di CPU burst e I/O burst e la distribuzione di tempo in cui i processi entrano nel sistema. Da questi dati si possono calcolare, con un modello statistico, i criteri di valutazione.

Il modello a reti di code è un modello statistico. Il sistema è visto come una rete composta da server ognuno con una propria coda dei processi in attesa di ottenere il servizio. Durante la sua vita, un processo è un client che richiede una sequenza di servizil. Le transizioni tra code rappresentano flussi di richieste.

Per costruire un modello a reti di coda, bisogna:

- Specificare la topologia del grafo della rete
- Specificare le caratteristiche di ogni servizio

- Analizzare la rete usando la teoria delle code

Il risultato dell'analisi statistica permette di ottenere dati quali l'utilizzo di una risorsa, la lunghezza media della coda e il tempo medio d'attesa.

La complessità matematica rende necessarie alcune semplificazioni, rendendo però la trattazione irrealistica.

4.5.3 Simulazione

Un altro modello statistico è la simulazione.

La simulazione prevede di programmare un modello di un sistema. I dati in input alla simulazione sono generati casualmente seguendo una distribuzione della probabilità, forniti empiricamente oppure definiti matematicamente.

Una simulazione basata sulle distribuzioni potrebbe non essere accurata a causa delle relazioni tra gli eventi in un sistema reale.

4.5.4 Implementazione

La modalità di valutazione più accurata è implementando l'algoritmo, scrivendone il codice e testandolo in un caso reale.

Vi sono dei costi relativi alla scrittura dell'algoritmo e all'ambiente di testing, solitamente una macchina virtuale. Inoltre, il sistema potrebbe cambiare nel tempo.

4.6 Politiche di schedulazione

L'algoritmo di schedulazione deve decidere quale processo nella coda dei processi pronti mettere in esecuzione. Per semplicità, in questa sezione si considerano sistemi con un singolo processore.

4.6.1 First-Come, First-Served

L'algoritmo **first-come first-served** (FCFS) alloca la CPU al primo processo che fa richiesta. L'implementazione avviene tramite una coda FIFO: l'ingresso di un processo alla coda avviene collegando il suo PCB in coda, quando la CPU è libera viene preso il primo processo in coda. FCFS quindi non è un algoritmo pre-emptive.

FCFS è un algoritmo semplice da implementare ma presenta diverse problematiche. In primo luogo, il tempo di attesa medio è molto lungo. In secondo luogo, processi CPU bound potrebbero monopolizzare il processore, che lascia processi I/O bound in attesa e di conseguenza lasciando inutilizzati i dispositivi.

4.6.2 Round-Robin

Round-Robin (RR) cambia i processi in esecuzione a rotazione, dunque simile a FCFS ma pre-emptive. Stabilito un certo quanto di tempo, i processi vengono immessi a turno in maniera circolare, ogni processo viene eseguito per un quanto prima di essere sostituito. Quindi la coda dei processi pronti è trattata come una lista circolare. Se un processo finisce

il CPU burst in meno di un quanto allora rilascia volontariamente la CPU, altrimenti il sistema operativo esegue un cambio di contesto involontario.

Il tempo di attesa in RR è solitamente molto alto. Le performance dell'algoritmo e il tempo di turnaround dipendono fortemente dalla lunghezza del quanto di tempo. Se la lunghezza del quanto è troppo grossa allora RR tende ad essere come FCFS, se il quanto è troppo corto allora avvengono troppi cambi di contesto. Come regola generale l'80% dei CPU burst devono essere inferiori del quanto di tempo. RR risulta utile per processi omogenei.

4.6.3 Schedulazione con priorità

Lo **scheduling con priorità** assegna un livello di priorità ad ogni processo, ed esegue prima i processi con priorità elevata. La priorità può essere assegnata internamente al sistema operativo (es. durata del CPU burst) o esternamente (es. livello di importanza). Processi con stessa priorità impiegano FCFS o round-robin. Solitamente si decide un certo range di valori interi per assegnare le priorità, si può poi considerare un processo più importante di un altro se ha assegnato un numero maggiore o minore.

Il problema della schedulazione per priorità è che processi con priorità bassa potrebbero rimanere bloccati indefinitivamente (**starvation**). Una soluzione adottata è detta **aging** e consiste nell'innalzare gradualmente il livello di priorità dei processi per una certa unità di tempo. Si tenga conto che è necessario ripristinare periodicamente la priorità oppure si innalzerebbero tutti i livelli di priortà allo stesso livello, diventando di fatto un FCFS.

La schedulazione con priorità può essere pre-emptive o non pre-emptive.

4.6.4 Shortest-Job-First

L'algoritmo **shortest-job-first** (SJF) associa ad ogni processo la durata del prossimo cpu burst, viene poi allocato alla CPU quello col ciclo più corto. Se due processi hanno stesso CPU burst allora viene impiegato FCFS. È quindi un caso particolare di schedulazione con priorità, dove la priorità è l'inverso della durata del CPU burst.

SJF garantisce il tempo minimo di attesa, ma viene difficile calcolare la durata del prossimo CPU burst. Si utilizza una stima detta media esponenziale. Siano t_n il tempo reale al tempo n, τ_n il tempo stimato al tempo n, α un valore t.c: $0 \leq \alpha \leq 1$, allora si calcola $\tau_{n+1} = \alpha t_n + (1 - \alpha)\tau_n$. Il valore α è il peso relativo dei tempi reali e stimati precedenti.

SJF può essere sia pre-emptive che non pre-emptive. SJF pre-emptive è detto anche **shortest-remaining-time-first**.

4.6.5 Coda a più livelli

Nella **coda a più livelli** si raggruppano i processi per tipologia omogenea. Ogni coda ha esigenze diverse e quindi algoritmi di scheduling diversi, solitamente RR. Lo scheduling tra l'insieme di code ha un algoritmo dedicato, solitamente pre-emptive con priorità.

4.6.6 Coda a più livelli con retroazione

La **coda a più livelli con retroazione** è una coda a più livelli che permette ai processi di cambiare tra una coda e l'altra. I processi vengono raggruppati per durata del CPU burst,

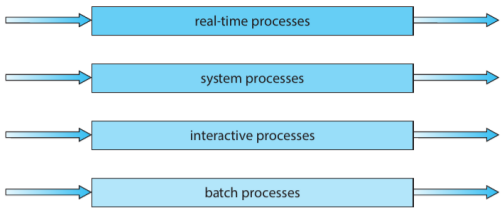

Figura 9: Code a più livelli per tipologia

lasciando i più brevi in code con priorità alta. Un processo che aspetta troppo viene messo in una coda con priorità più alta, risolvendo il problema della starvation.

Una coda a più livelli con retroazione è definita da:

- Un insieme di code
- Un algoritmo di scheduling per ogni coda
- Un criterio per decidere quando innalzare la priorità di un processo
- Un criterio per decidere quando diminuire la priorità di un processo
- Un criterio per decidere a quale coda assegnare un processo appena entrato nella coda dei processi pronti

La coda a più livelli è l'algoritmo di scheduling più generale, può essere configurato per soddisfare diverse esigenze. Tuttavia è difficile da implementare.

4.7 Schedulazione dei thread

Il sistema operativo schedula i kernel thread nell'ambito dei processi di sistema (**System-Contention Scope, SCS**); le librerie thread a livello utente schedulano i thread utente nell'ambito del singolo processo (**Process-Contention Scope, PCS**), il sistema operativo è ignaro dell'esistenza di questa schedulazione. Solitamente le librerie thread schedulano i thread utente secondo una politica di priorità pre-emptive, con priorità assegnata dal programmatore.

I sistemi operativi con modello uno a uno schedulano soltanto nell'ambito dei processi di sistema.

4.8 Schedulazione in sistemi in tempo reale

Un sistema in tempo reale è un sistema in cui la correttezza delle operazioni dipende sia dal risultato logico sia dal rispetto di una scadenza entro cui le operazioni devono essere completate. I sistemi in tempo reale si classificano in base alle conseguenze della mancata scadenza:

- Sistemi in tempo reale rigido: le attività devono necessariamente essere completate entro la scadenza.
- Sistemi in tempo reale moderato: l'utilità del risultato è nulla se viene superata la scadenza. Comunque il sistema può tollerare un certo numero di scadenze perse, con conseguente degrado della qualità.

- Sistemi in tempo reale lasco: von vi è nessuna garanzia che le attività rispettino le scadenze. Si garantisce solo che i processi critici abbiano priorità più alta rispetto a quelli non critici.

Tempi di latenza Si definisce latenza di un evento il tempo tra la richiesta di un servizio e il momento in cui viene erogato. Eventi diversi hanno tempi di latenza diversi. Nei sistemi in tempo reale vi sono due tipi di latenza influenzano i sistemi in tempo reale:

- Latenza di interrupt: tempo tra l'arrivo di un interrupt e il momento in cui la CPU lo gestisce. Questo tempo è influenzato dal tempo in cui gli interrupt vengono disabilitati per aggiornare le strutture dati del kernel.
- Latenza di dispatching: tempo di esecuzione del dispatcher per feramre il processo corrente e avviare quello successivo. Il miglior modo per limitare la latenza di dispatching è usare kernel preemptive.

Processi periodici Un processo periodico è un processo che viene eseguito periodicamente a intervalli costanti di tempo. Ogni processo periodico ha un tempo di esecuzione t, una scadenza d e un periodo p. La frequenza del processo è $1/p$. Gli schedulatori tengono conto di queste caratteristiche per schedulare i processi. Un processo periodico annuncia la sua scadenza all'algoritmo di schedulazione, che seguendo un algoritmo detto di controllo dell'ammissione, ammette il processo se e solo se può garantire che questo può essere servito entro la sua scadenza.

Algoritmi di schedulazione in un sistema in tempo reale Lo schedulatore di un sistema in tempo reale deve supportare un algoritmo con coda con priorità preemptive. Per i sistemi in tempo reale lasco è sufficiente, per sistemi in tempo reale stretto bisogna usare altri algoritmi:

- Schedulazione a frequenza monotona: si usa una priorità statica con preemption. La priorità è inversamente proporzionale alla frequenza del processo: al diminuire della frequenza aumenta la priorità. L'obbiettivo è allocare la CPU ai processi che la richiedono più spesso. Questo algoritmo di schedulazione assume che ogni processo abbia un proprio tempo di esecuzione che è sempre lo stesso.
- Schedulazione a scadenza più urgente: la schedulazione a scadenza più urgente assegna la priorità dinamicamente alla scadenza dei processi. L'assegnazione di priorità è inversamente proporzionale alla scadenza: più presto è la scadenza, più è alta la priorità. Funziona anche per processi non periodici e non richiede che un processo annunci il suo tempo di elaborazione. Teoricamente è ottimale ma le latenze di interruzione e di dispatching abbassa l'utilizzo della CPU sotto il 100%.

Nei sistemi con processi omogenei si usa Round-Robin. Nei sistemi in tempo reale rigido vi è un algoritmo detto di ammissione che accetta il processo se e solo se può garantire che il processo può essere schedulato e terminato entro la scadenza. Nei sistemi in tempo reale lasco la priorità è statica per processi critici, dinamica per processi non critici.

5 Comunicazione interprocesso (IPC)

5.1 Processi cooperanti

Due processi si dicono cooperanti se lavorano congiuntamente per uno scopo applicativo comune, scambiandosi informazioni e se uno può influenzare l'altro. Viceversa si dicono

indipendenti.

I vantaggi della cooperazione sono:

- Condivisione delle informazioni
- Specializzazione
- Parallelizzazione delle attività.
- Moularità
- Scalabilità
- Qualità della realizzazione del progetto

I processi cooperanti hanno bisogno di meccanismi e politiche di comunicazione e sincronizzazione.

5.2 Caratteristiche dei metodi IPC

Le entità coinvolte nella comunicazione interprocessi sono il processo mittente P, il processo destinatario Q e il canale di comunicazione che può essere unidirezionale o bidirezionale. La scelta di un metodo di comunicazione va scelto in base a diverse caratteristiche:

- Quantità di informazioni da trasmettere
- Velocità di esecuzione
- Scalabilità: quanti processi possono comunicare
- Semplicità di utilizzo
- Omogeneità nelle comunicazioni: si vuole evitare di usare più di un metodo di IPC.
- Integrazione nei linguaggi di programmazione
- Affidabilità
- Sicurezza
- Protezione

I metodi IPC si distinguono in:

- Comunicazione diretta: i processi comunicanti conoscono l'uno il nome dell'altro. Richiede che entrambi i processi siano attivi.
- Comunicazione indiretta: i processi conoscono solo dove prelevare il messaggio.

5.3 Comunicazione con memoria condivisa

La comunicazione con memoria condivisa può avvenire con variabili globali. P e Q hanno il proprio spazio di indirizzamento, ma anche una parte comune che si sovrappone. Questa può essere realizzata in due modi:

- Area di memoria fisicamente condivisa: il SO mappa parte dello spazio di indirizzamento logico dei processi su uno spazio fisico condiviso.
- Buffer con copiatura gestita dal sistema operativo: il sistema operativo esegue una doppia copiatura. Prima dal processo mittente nel proprio spazio di indirizzamento, poi dal proprio spazio di indirizzamento a quello del processo destinatario. Questo metodo richeide un grosso overhead di gestione a causa della doppia copiatura.

Oltre alle varaibili globali è possibile comunicare con condivisione di buffer. Il processo mittente scrive le informazioni nel buffer, e il ricevente le preleva. Anche questo si può

realizzare con un'area di memoria fisicamente condivisa o con un buffer con copiatura gestita dal sistem operativo.

Le problematiche princpali sono: l'identificazione dei processi comunicanti; garantire la consistenza delle operazioni; sincronizzare l'accesso in mutua esclusione.

5.4 Comunicazione con scambio di messaggi

La comunicazione con scambio di messaggi fornisce un meccanismo che permette ai processi di comunicare con messaggi scambiati dal sistema operativo, senza passare dai propri spazi di indirizzamento. Sono sempre fornite le primitive `send(message)` e `receive(message)`. I messaggi possono essere a dimensione fissa o variabile, e contengono: processo mittente, processo destinatario, informazioni da trasmettere ed eventuali informazioni di gestione.

5.4.1 Denominazione: comunicazione diretta e indiretta

Lo scambio con messaggi può avvenire a comunicazione diretta o indiretta. In un sistema con comunicazione diretta simmetrica i processi devono nominare esplicitamente il recipiente oppure il mittente del messaggio; in un sistema con comunicazione diretta assimmetrica o il destinatario o il mittente non sono nominati esplicitamente, in questo caso invio e ricezione sono verso/da un processo qualunque o facente parte di un gruppo specificato.

Lo svantaggio della comunicazione diretta è che cambiare l'identificatore di un processo necessita di esaminare tutti gli altri processi per effettuare il cambio.

Nella comunicazione indiretta i messaggi sono mandati da un recipiente e prelevati dal recipiente, detto mailbox o porta. Le primitive messe a disposizione sono `send(A, message)` e `receive(A, message)` dove `A` è l'identificatore della mailbox. Il canale di comunicazione dunque: è stabilito tra due processi se e solo se i due hanno una mailbox in comune; può essere usato da più processi; un processo può usare più canali di comunicazione.

Se la mailbox è usata da più processi, potrebbe accadere che più di uno chiami `receive()`. A questo punto, per decidere a quale processo inviare il messaggio, il sistema operativo potrebbe:

- Permettere a una mailbox di essere usata al massimo da due processi
- Permettere a un solo processo di chiamare `receive()`
- Selezionare l'ordine dei processi a cui inviare i messaggi utilizzando una politica come FIFO, scadenza o priorità. Si può avere lo stesso criterio per tutte le mailbox o uno diverso per ognuna.

Una mailbox può essere posseduta dal sistema operativo o da un processo, ovvero essere parte del suo spazio di indirizzamento. In una mailbox posseduta da un processo, il possessore può solamente ricevere messaggi, gli utenti possono soltanto mandarne. Inoltre, se il processore possessore termina allora termina anche la mailbox. Se la mailbox è posseduta dal sistema operativo, questi deve fornire anche le primitive per creare, distruggere e mandare/ricevere messaggi sulla mailbox. Inizialmente il possessore della mailbox è chi la crea, ma il privilegio può essere esteso anche agli altri permettendo a più processi di ricevere messaggi.

La comunicazione può avere cardinalità:

- Molti a uno: più processi client chidono un servizio a un processo server.

- Uno a molti: un processo client chiede un servizio a più processi server, il primo disponibile eroga il servizio.
- Molti a molti: più processi client chiedono un servizio a più processi server.

5.4.2 Sincronizzazione

Le primitive `send()` e `receive()` possono essere sincrone (bloccanti) o asincrone (non bloccanti):

- `send()` bloccante: il processo mittente viene bloccato fino a quando il messaggio non è ricevuto dalla mailbox o dal destinatario.
- `receive()` bloccante: il ricevente viene bloccato fino a quando non vi è un messaggio disponibile.
- `send()` non bloccante: è la chiamata `send_cond()`, se l'invio non può essere completato allora non blocca il processo ma restituisce un messaggio d'errore
- `receive()` non bloccante: è la chiamata `receive_cond()`, se non vi sono messaggi non blocca il processo ma restituisce un messaggio d'errore.l

5.4.3 Buffering

I messaggi della comunicazione risiedono in una coda temporanea. Un buffer può essere assegnato a ciascuna coppia di processi o può essere ad uso generale. La quantità di buffer assegnati può essere:

- Nulla: il mittente non può depositare nessun messaggio, quindi l'invio è bloccante fino a quando il ricevente non legge. Si osserva che la comunicazione è sincrona.
- Limitata: vi sono una quantità limitata di messaggi, l'invio è bloccante se e solo se non ci sono buffer disponibili.
- Illimitata: il mittente deposita immediatamente il messaggio in un buffer, quindi l'invio non è mai bloccante.

5.5 Comunicazione con file e pipe

La comunicazione con file può avvenire con file condiviso o con pipe.

Comunicazione con file condiviso Vi è un file su disco su cui vengono depositati e prelevati i messaggi. Con questo tipo di comunicazione la comunicazione può avvenire uno a uno, molti a uno, uno a molti e molti a molti.

Comunicazione con pipe La comunicazione si può realizare in maniera smile con le pipe, nello spazio di indirizzamento del sistema operativo. Una pipe è un file in memoria centrale con scrittura soltanto in aggiunta e lettura unica sequenziale. I messaggi contengono: il processo mittente; le informazioni da trasmettere; eventuali informazioni di gestione. Possono essere a dimensione fissa o variabile.

Le funzioni fornite sono creazione e cancellazione della pipe o del file, lettura e scrittura da e per la pipe o il file.

Le caratteristiche di questo metodo sono:

- la sincronizzazione in lettura e scrittura è effettuata dal filesystem;
- l'ordinamento dei messaggi nei file dipende dal filesystem, nelle pipe è sempre FIFO;
- l'ordinamento dei processi nei file dipende dal filesystem.

5.6 Comunicazione con socket

Il modello di comunicazione con socket è la generalizzazione della comunicazione in rete con pipe. Una macchina ha una socket (o porta) identificata dall'indirizzo di rete e dalla porta. È la classica comunicazione client-server dove il mittente chiede il servizio inviando la richiesta su una macchina specifica a una porta specifica.

I messaggi hanno dimensione fissa o variabile. Le funzioni fornite sono creazione e cancellazione della socket, lettura e scrittura.

Il canale di comunicazione è l'insieme della rete e della porzione di socket da cui si prendono i messaggi. L'ordinamento dei messaggi è FIFO, i processi in attesa sono FIFO. La connessione può essere con gestione della connessione (ovvero vi sono meccanismi che garantiscono l'arrivo del messaggio) o senza gestione della connessione. Può con multicast.

6 Sincronizzazione tra processi

6.1 Processi concorrenti e corse critiche

Due processi si dicono concorrenti quando chiedono l'accesso a una risorsa condivisa usabile solo in mutua esclusione. Con mutua esclusione si intende che non possono essere compiute più operazioni incompatibili da parte di più processi.

Quando il risultato dell'esecuzione dipende dall'ordine in cui sono eseguite le istruzioni, sta avvenendo una corsa critica. Il segmento di codice in cui avviene una corsa critica è detto sezione critica. Vi sono poi i segmenti di codice di ingresso della sezione critica, uscita dalla sezione critica e sezione non critica.

La sincronizzazione è l'insieme di politiche e meccanismi che garantiscono la mutua esclusione per risorse condivise, quindi l'integrità dei dati. Le risorse condivise possono essere sia fisiche che condivise. Un protocollo di sincronizzazione, ovvero che risolve il problema della sezione critica, deve garantire:

- Mutua esclusione: se un processo sta eseguendo la sua sezione critica, nessun altro può eseguire la propria.
- Progresso: se nessun processo è nella sezione critica e alcuni processi voglio entrare, solo questi possono partecipare alla decisione sul prossimo e questa non può essere ritardata indefinitivamente.
- Attesa limitata: deve esserci un limite al numero di volte in cui un processo possa entrare nella sezione critica dopo che un altro processo ha chiesto di entrare.

6.2 Supporto hardware alla sincronizzazione

L'hardware potrebbe fornire delle primitive per costruire soluzioni al problema della sincronizzazione o come strumenti stessi per la sincronizzazione.

6.2.1 Barriere della memoria

Un'architettura potrebbe riordinare le istruzioni e portare ad instabilità nei dati. Pertanto vi deve essere un modello di memoria che determina quando la memoria si considera garantita. Vi sono due tipi di modelli:

- Fortemente ordinata: una modifica in memoria è immediatamente visibile.
- Debolmente ordinata: una modifica in memoria non è immediatamente visibile.

Le architetture pertanto forniscono delle istruzioni che forzano la propagazione di qualunque cambiamento in memoria su tutti i processori. Queste istruzioni sono dette barriere della memoria e garantiscono che tutte le operazioni di `load` e `store` siano completate prima delle prossime.

6.2.2 Istruzioni hardware: test and set, compare and swap

Le architetture forniscono istruzioni hardware utili a testare e modificare il contenuto di una parola o scambiare il contenuto di due parole atomicamente, dunque senza interruzioni.

L'operazione di test and set setta a true la variabile in input e ne restituisce il valore precedente.

```
boolean test_and_set(boolean *target) {
    boolean rv = *target;
    *target = true;
    return rv;
}
```

L'operazione di compare and swap opera su tre operandi: se il valore di `value` è uguale ad `expected`, allora setta `value` come `new_value` e restituisce il vecchio valore.

```
int compare_and_swap(int *value, int expected, int new_value) {
    int temp = *value;
    if (*value == expected) {
        *value = new_value;
    }
    return temp;
}
```

6.2.3 Variabili atomiche

6.3 Variabili di turno e di lock

L'approcio con variabili di turno è un approcio a livello di istruzioni ovvero che non richiede nè l'intervento del sistema operativo nè dell'hardware. Una variabile di turno è una variabile condivisa che definisce il turno della risorsa, ovvero a quale processo spetta il diritto d'uso in un certo istante.

Dati due processi P_i e P_j con $j = 1-i$, allora un algoritmo con variabile di turno utilizza una variabile intera `turn` tale che se `turn = i` allora il processo P_i ha diritto a usare la risorsa, e un vettore booleano di due elementi `flag` tale che se `flag[i] = true` allora il processo P_i vuole la risorsa. A questo punto, l'accesso alla sezione critica per P_i è così eseguito:

```
while (true) {
    flag[i] = true
    turn = j
    while (flag[j] && turn == j);
    /* Sezione critica */
```

```
    flag[i] = false
    /* Sezione non critica */
}
```

Si tenga presente che a causa del riordino delle istruzioni e del funzionamento di `load` e `store` nelle architetture moderne, quest'approcio potrebbe non funzionare. Pertanto, è meglio usare il supporto hardware descritto precedentemente.

Un altro approcio a livello di istruzioni utilizza le variabili di lock. Una variabile di lock è una variabile che definisce lo stato di uso di una risorsa. Questa è posta uguale a 0 se la risorsa è libera, uguale a 1 se è occupata. La variabile di lock può essere modificata disabilitando le interruzioni e rendendo la modifica atomica. Un altro modo di leggere e modificare la variabile di lock atomicamente è con l'istruzione hardware test and set.

Gli approci a livello di istruzione richiedono attenzione da parte del programmatore.

6.4 Semafori

Gli approci a livello di istruzione potrebbero portare ad errori. Si vuole innalzare il livello di astrazione, portando la sincronizzazione come funzionalità del sistema operativo, garantendo la corretta gestione della gestione delle variabili.

Un semaforo è una struttura variabile intera S, vi sono due tipi di semafori con semantica e dominio diverso:

- Semaforo binario $S \in \{0, 1\}$: è una variabile binaria che definisce se una risorsa è libera ($S = 1$) o in uso ($S = 0$).
- Semaforo generalizzato $S \in D \subseteq \mathbb{N} \cup \{0\}$: è una variabile intera che rappresenta lo stato d'uso di un insieme di risorse omogenee. Se $S = n$ allora vi sono n risorse libere.

Il SO mette a disposizione due primitive:

- `release(S)` (o `signal`): rilascia la risorsa su semaforo S.
- `acquire(S)` (o `wait`): ottiene la risorsa su semaforo S, aspettando se non disponibile.

6.4.1 Implementazione

L'implementazione dei semafori con la sola variabile intera vuol dire mettere in attesa attiva un processo che sta aspettando una risorsa. Pertanto si ricorre a una struttura dati che utilizza anche una coda dei processi in attesa. Quando un processo richeide una risorsa e questa non è pronta, viene inserito nella coda e sospeso; quando un processo rilascia la risorsa attiva il primo processo della coda. Lo schedulatore dei processi decide l'ordine di ottenimento della risorsa in base alla politica scelta.

6.5 Monitor

Con i semafori la responsabilità dell'uso è lasciata al programmatore, che deve chiamare in modo corretto le funzioni `acquire` e `release` e definire la sezione critica. Per evitare errori si innalza utleriormente il livello di astrazione.

Un monitor è un costrutto di sincronizzazione formulato a livello di linguaggio di programmazione, si tratta di un costrutto linguistico trasformato nelle corrette chiamate a

sistema dal compilatore. Solo un processo alla volta può essere attivo in un monitor. In particolare è un tipo di dato astratto che include:

- struttura dati condivisa con lo spazio per memorizzare il semaforo ed altre informazioni;
- le operazioni definite dal programmatore dei processi nel monitor, ovvero le sezioni critiche;
- del codice di inizializzazione del modulo, ad esempio per inizialzizare il semaforo;
- la coda dei processi in attesa di entrare nel monitor, schedulata secondo la politica adeguata.

I monitor potrebbero non essere adatti a certe situazioni. Pertanto vi è un costrutto detto `condition`. Questi prevede di dichiarare una o più variabili di tipo `condition`. Su queste variabili si possono effettuare le operazioni di `wait()` e `signal()`. Si suppong `condition x`, allora un processo che chiama `x.wait()` viene sospeso fino a quando un altro processo chiama `x.signal()`. Si consideri un processo P che chiama `x.signal()`, allora esiste un processo sospeso Q che ha chiamato `x.wait()`. A questo punto se Q riprende l'esecuzione, allora P dovrebbe aspettare, altrimenti si avrebbero due processi nel monitor. Vi sono due possibilità di esecuzione:

- Segnala e attendi: P attende che o Q esca dal monitor o per un'altra condizione.
- Segnala e continua: Q aspetta cheo P esce dal monitor o per un'altra condizione.

Un compromesso tra le due possibilità è far uscire P non appena esegue `x.signal()`, quindi Q viene riattivato.

6.6 Problemi di sincronizzazione: starvation e deadlock

La sincronizzazione potrebbe portare a due problemi: starvation e deadlock.

Starvation (blocco indefinito) Si parla di starvation quando un processo rimane indefinitivamente in attesa di usare una risorsa poichè altri processi ottengono prima tale risorsa. Questo stato è causato dall'uso di una politica di schedulazione della coda di attesa che non garantisce a tutti i processi di ottenere la risorsa in tempo finito. Si risolve scegliendo una politica adatta.

Deadlock Si parla di deadlock quando in un gruppo di due o più processi ciascun processo aspetta una risorsa detenuta in modo mutuamente esclusivo da un altro processo del gruppo

7 Deadlock

7.1 Risorse condivise e caratterizzazione del deadlock

Un sistema ha un numero finito di risorse distribuite a più processi concorrenti. Le risorse sono partizionate in più tipi, ognuno con una o più istanze; quando un processo richiede un tipo di risorsa, gli deve essere assegnata una qualsiasi istanza della stessa. Un processo utilizza le risorse nelle seguenti fasi:

1. Richiesta: quando richiede la risorsa, se questa non è disponibile, va in attesa.
2. Uso.
3. Rilascio.

Un insieme di processi è in deadlock quando tutti i processi dell'insieme sono in attesa di una risorsa che può essere rilasciata solo da uno degli altri processi in attesa. Specificatamente, un deadlock si verifica se e solo se avvengono simultaneamente le seguenti condizioni:

- Mutua esclusione
- Possesso e attesa: un processo possiede una risorsa ed è in attesa per un'altra risorsa.
- Nessun rilascio anticipato: la risorsa posseduta dal processo non può essere rilasciata se non per volontà del processo stesso.
- Attesa circolare: vi è un insieme di N processi $\{P_0, P_1, \cdots, P_N\}$ tali che P_0 attende una risorsa posseduta da P_1, P_1 attende una risorsa da $P_2, \ldots, P_N$ attende una risorsa da P_1.

I metodi per evitare il deadlock sono:

- Ignorare il deadlock.
- Prevenire il deadlock.
- Evitare il deadlock.
- Rilevare e recuperare il deadlock.

Ignorare il deadlock è la soluzione più utilizzata in quanto più economica. Inoltre, in alcuni sistemi il deadlock è molto raro.

7.2 Grafo di allocazione delle risorse

Si può disegnare un grafo di allocazione delle risorse $G = (V, E)$, orientato, i cui elementi sono:

- Insieme di nodi V: i nodi sono partizionati nell'insieme dei processi di sistema $P = \{R_1, \cdots, R_n\}$ e delle risorse $R = \{R_1, \cdots, R_m\}$ con le relative istanze.
- Archi E: un arco diretto $P_i \to R_j$ da un processo P_i a una risorsa R_j è detto arco di richiesta ed indica che il processo P_i ha richiesto la risorsa R_j; un arco diretto $R_j \to P_i$ da una risorsa R_j a un processo R_i è detto arco di assegnazione e indica che la risorsa R_j è assegnata al processo P_i.

Ogni processo P_i è rappresentato con una circonferenza; ogni tipo di risorsa R_j è rappresentata da un rettangolo con all'interno delle piccole circonferenze che rappresentano un'istanza della risorsa. Un arco di richiesta punta al rettangolo; un arco di assegnazione parte dall'istanza, quindi dal punto.

Se il grafo non contiene cicli allora l'insieme di processi non è in stato di deadlock. Se il grafo contiene un ciclo allora l'insieme di processi potrebbe essere in deadlock. Quindi un ciclo è condizione necessaria ma non sufficiente affinchè vi sia un deadlock. Se il ciclo coinvolge risorse con una sola istanza allora vi è un deadlock e il ciclo è condizione necessaria e sufficiente.

7.3 Tecniche di prevenzione del deadlock

L'approcio nelle tecniche di prevenzione del deadlock è prevenire il deadlock impedendo che almeno una delle condizoni per cui si verifica non sia soddisfatta.

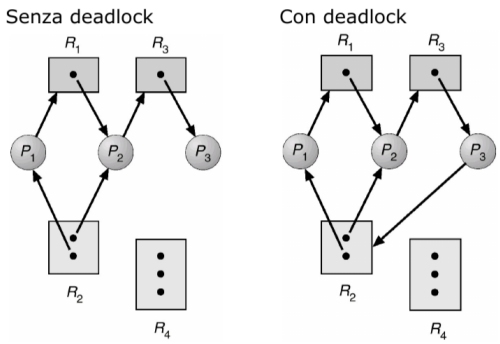

Figura 10: Grafi di allocazione delle risorse: a sinistra grafo ciclico con deadlock; a destra grafo ciclico senza deadlock

7.3.1 Eliminazione della condizione di mutua esclusione

La mutua esclusione deve essere garantita. Alcune risorse sono condivisibili (es. file in sola lettura) e quindi non causano deadlock; altre è necessario che vengano usate in mutua esclusione, e il sistema operativo deve garantire ciò.

7.3.2 Eliminazione della condizione di possesso e attesa

Per assicurare che la condizione di posseso e attesa non occorra mai, bisogna far si che qualora un processo richieda un ricorsa, non ne possegga un'altra. Si possono usare due protocolli:

- Richiedere che un processo richieda tutte le sue risorse prima dell'esecuzione.
- Richiedere che un processo possa chiedere una risorsa solo quando non ne ha nessuna.

In entrambi gli approci l'utilizzo delle risorse potrebbe essere basso, un processo potrebbe detenere una risorsa per tanto tempo ma utilizzarla per poco; inoltre potrebbe manifestarsi starvation, in quanto un processo che ha bisogno di molte risorse popolari potrebbe aspettare indefinitivamente.

7.3.3 Eliminazione della condizione di nessun rilascio anticipato

La condizione può essere invaldiata mediante rilascio anticipato delle risorse il cui stato di uso all'atto del rilascio è ripristinabile.

Si possono usare due protocolli:

- Se un processo P possiede alcune risorse e ne chiede altre non disponibili, allora tutte le risorse di P vengono rilasciate e inserite nelle lista delle risorse che P sta aspettando. P riparte solo quando le risorse vecchie e quelle nuove sono disponibili.
- Se un processo P chiede delle risorse non disponibili, si controlla se sono allocate a un altro processo Q che sta aspettando altre risorse, queste vengono rilasciate da Q e allocate a P, altrimenti va in attesa. Un processo viene ricaricato solo quando sono disponibili le vecchie, che potrebbe aver perso mentre in attesa, e le nuove risorse.

7.3.4 Eliminazione della condizione di attesa circolare

Per invalidare la condizione di attesa circolare si impone un ordinamento totale tra le risorse e si richiede che un processo possa richiederne soltanto nell'ordine di enumerazione. Formalmente si definisce una funzione $F : R \to \mathbb{N}$ che forma un insieme di risorse enumerate $R = \{R_1, ..., R_m\}$.

Se un processo richiede k istanze di una risorsa R_j, vi sono due casi:

- Il processo detiene solo risorse R_i con $i < j$: se le k istanze della risorsa R_j sono disponibili le ottiene, altrimenti attende.
- Il processo detiene risorse R_i con $i > j$: il processo deve rilasciare tutte le risorse R_i, chiedere le istanze delle risorse R_j e chiedere le istanze delle risorse R_i che deteneva precedentemente.

7.4 Tecniche per evitare il deadlock

Le tecniche per evitare il deadlock verificano a priori se la sequenza di richieste e rilasci di risorse di un processo portano al deadlock, tenendo conto dei processi già presenti nel sistema. Le informazioni necessarie a priori sono:

- Numero massimo di risorse per ogni processo
- Risorse attualmente alloccate
- Risorse attualmente disponibili
- Future richieste e rilasci di risorse

L'insieme di numero massimo di risorse per ogni processo, risorse attualmente allocate e risorse attualmente disponibili è detto stato di allocazione. Gli algoritmi per evitare il deadlock esaminano dinamicamente lo stato di allocazione delle risorse per garantire che non possa mai verificarsi l'attesa circolare, chiedendo ad ogni processo il numero massimo di risorse che userà e assegnandole solo se non potrà verificarsi uno stallo.

Stato sicuro Gli algoritmi più utilizzati si basano sul concetto di stato sicuro. Uno stato si dice sicuro se il sistema può allocare le risorse richieste in un certo ordine garantendo che non si verifichi deadlock. Formalmente sequenza di processi $(P_1, \cdots, P_m)$ si dice sicura per lo stato di allocazione corrente se e solo se le richieste che ogni processo P_i può fare possono essere soddisfatte dalle risorse attualmente disponibili più tutte le risorse detenute dai processi P_j per $j < i$. Uno stato è sicuro se esiste una sequenza sicura.

Quindi bisogna garantire che il sistema passi da uno stato sicuro a un altro stato sicuro quando un processo chiede una risorsa. Inizialmente si è in uno stato sicuro, la richiesta di una risorsa viene soddisfatta se la risorsa è disponibile e il sistema entra in uno stato sicuro, altrimenti il processo deve attendere.

7.4.1 Algoritmo del grafo di allocazione delle risorse

Se il sistema ha solamente istanze singole per ogni risorsa si può usare l'algoritmo del grafo di allocazione delle risorse. Questi modifica il grafo di allocazione delle risorse descritto in precedenza, aggiungendo un nuovo tipo di arco chiamato arco di prenotazione. Un arco di prenotazione $P_i \to R_j$ è un arco diretto da un processo P_i a una risorsa R_j la cui semantica è che P_i potrà richiedere in futuro la risorsa R_j. L'allocazione di una risorsa corrisponde a

trasformare l'arco di prenotazione $P_i \to R_j$ in un arco di assegnazione $R_j \to P_i$. Dunque, l'algoritmo controlla se la conversione dell'arco $P_i \to R_j$ nell'arco $R_j \to P_i$ forma un ciclo. Se così fosse, P_i è messo in attesa.

Si osserva che un algoritmo di verifica dei cicli richiede $\Theta(n^2)$ operazione dove n è il numero di processi del grafo.

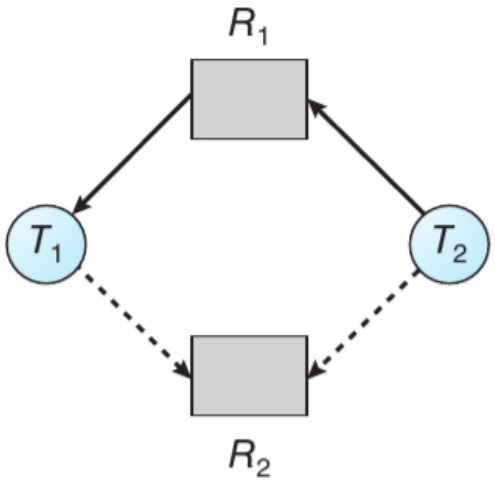

Figura 11: Grafo di allocazione delle risorse con archi di richiesta. Un arco di richiesta è rappresentato da una linea tratteggiata

7.4.2 Algoritmo del banchiere

L'algoritmo del banchiere è meno efficiente dell'algoritmo del grafo di allocazione delle risorse, ma può essere usato su sistemi con più istanze per ogni risorsa. Un processo deve dichiarare il numero massimo di risorse di cui avrà bisogno e restituirle in un tempo finito, quando poi le risorse vengono richieste l'algoritmo deve determinare se l'assegnazione delle risorse lascia il sistema in uno stato sicuro. Si illustrano le strutture dati necessarie, successivamente la procedura per riconoscere se uno stato è sicuro e la procedura per determinare se una richiesta può essere soddisfatta.

L'algoritmo prende il nome dalle banche, che si assicurano di dare prestiti solo se possono soddisfare tutti i clienti.

Strutture dati Le strutture dati necessari modellano lo stato di allocazione. Siano n il numero di processi e m il numero di risorse, l'algoritmo del banchiere richiede le seguenti strutture dati:

- `Avaible`: vettore di lunghezza m che indica per ogni risorsa il numero di istanze disponibili. Se `Avaible[i] = k` allora sono disponibili k istanze della risorsa R_i.
- `Max`: matrice $n \times m$ che indica per ogni risorsa il numero di istanze massimo per ogni processo. Se `Max[i][j] = k` allora il processo P_i richiederà al massimo k volte la risorsa R_j.
- `Allocation`: matrice $n \times m$ che indica per ogni risorsa il numero di istanze allocate ad ogni processo. Se `Allocation[i][j] = k` allora al processo P_i sono attualmente

allocate k istanze della risorsa R_j.
- `Need`: matrice $n \times m$ che indica il numero di risorse ancora da richiedere per ogni processo. Se `Need[i][j] = k` allora il processo P_i dovrà ancora chiedere k istanze della risorsa R_j. Si osserva `Need[i][j] = Max[i][j] - Allocation[i][j]`.

Procedura per riconoscere lo stato sicuro L'algoritmo per determinare se il sistema è in uno stato sicuro è il seguente:

```
// Vettore delle risorse disponibili
Work[1..m] <- Avaible
// Vettore dei processi che hanno terminato la computazione
Finish[1..n]
FOR i <- 0 TO n DO Finish[i] <- True

ciclo:
// Cerca un processo non terminato e le cui richieste di risorse possono essere
// soddisfatte da quelle disponibili in questo momento.
i <- indice i tale che Finish[i] = False e Need[i][j] <= Work[j] per ogni j
IF tale i non esiste THEN
    goto fine
// Una volta che il processo termina l'esecuzione, allora le risorse che gli erano state assegnat
Finish[i] <- True
FOR j <- 0 TO m DO Work[j] <- Work[j] + Allocation[i][j]
goto ciclo

fine:
IF Finish[i] = True per ogni i THEN
    RETURN Stato Sicuro
ELSE
    RETURN Stato non sicuro
```

Questa procedura richiede un numero di operazioni pari a $\Theta(n \cdot m^2)$.

Procedura di richiesta delle risorse L'algoritmo per determinare se una richiesta può essere soddisfatta è il seguente:

```
// Vettore delle richieste del processo. Se Request[i][j] = k
// allora il processo P_i vuole k istanze della risorsa R_j
Request[1..n][1..m]

// Controlla che il processo non ecceda il numero massimo di richieste
IF Request[i] > Need[i] THEN
    Errore: Richiesta maggiore del massimo dichiarato
// Controlla che le risorse richieste siano disponibili
IF Requesti[i] > Avaible[i] THEN
    Attendi: risorse non disponibili

// Si ipotizza che le risorse siano allocate. Se lo stato rimane sicuro
// assegna le risorse, altrimenti ripristina lo stato di allocazione
Avaible <- Avaible - Request[i]
```

```
Allocation[i] <- Allocation[i] + Request[i]
Need[i] <- Need[i] - Request[i]
IF sicuro(Avaible, Allocation, Need) THEN alloca risorse
ELSE attendi e ripristina stato di allocazione
```

7.5 Tecniche di rilevazione e ripristino del deadlock

In un sistema che non prevede sistemi per prevenire o evitare il deadlock, questi ptorebbero verificarsi. Il sistema deve prevedere un metodo per rilevare il deadlock e, dopo averlo rilevato, ripristinare una situazione di corretto funzionamento eliminando il deadlock.

L'algoritmo di rilevazione viene evocato in base a quanto è probabile che un deadlock si presenti e dal numero di processi solitamente coinvolti. Generalmente, vi sono due approci:

- Invocare la rilevazione ogni volta che un processo richiede una risorsa e questa non può essere assegnata: la rilevazione è immediata e poche risorse verranno coinvolte nello stallo, ma vi è un alto sovraccarico computazionale.
- Invocare la rilevazione a intervalli di tempo prestabiliti: la rilevazione è pù complessa e potrebbero esserci molti processi coinvolti, rendendo difficile determinare quale processo è bloccato. Vi è però meno sovraccarico computazionale.

7.5.1 Rilevazione del deadlock in sistemi con singole istanze delle risorse

Nei sistemi con singole istanze per ogni risorsa si utilizza una variante del grafo di allocazione delle risorse, detta grafo di attesa, ottenuto rimuovendo i nodi di tipo risorsa e collassando opportunamente gli archi.

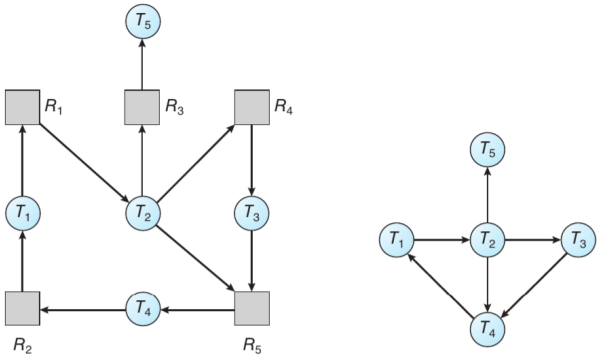

Figura 12: A sinistra grafo di allocazione delle risorse, a destra grafo di attesa ottenuta dal grafo di sinistra

Il sistema è in una situazione di stallo se e solo se il grafo di attesa contiene un ciclo. I processi coinvolti sono quelli presenti nel ciclo.

7.5.2 Rilevazione del deadlock in sistemi con istanze multiple delle risorse

L'algoritmo di rilevazione del deadlock in sistemi con multiple istanze è simile a quello usato dall'algoritmo del banchiere.

```
// Vettore che indica il numero di risorse disponibili per tipo
Avaible[1..m]
// Matrice che definisce il numero di risorse per tipo allocate ad ogni processo
Allocation[1..n][1..m]
// Matrice che indica le richieste attuali per tipo di risorsa per ogni thread
Request[1..n][1..m]

Work[1..m] <- Avaible
Finish[1..n] <- Se allocation[i] != 0 allora Finish[i] = False; altrimenti Finish[i] = True\

ciclo:
i <- trova indice i tale che Finish[i] = False e Request[i] <= Work
IF tale i non esiste THEN
    goto fine
Work <- Work + Allocation[i]
Finish[i] = True
goto ciclo

fine:
IF esiste i t.c. Finish[i] = False THEN
    RETURN C'è deadlock e per i t.c. Finish[i] = False allora il processo i è in deadlock
```

L'algoritmo richiede $\Theta(m \cdot n^2)$ operazioni.

7.5.3 Ripristino dello stallo con terminazione dei processi

Per eliminare lo stallo terminando i processi, vi sono due metodi:

- Terminare tutti i processi coinvolti: porta un costo elevato, troppi processi terminati e spreco di risorse computazionali che dovranno essere riutilizzate.
- Terminare un processo alla volta fino a quando non vi sono più cicli: pochi processi terminati a sovraccarico determinato dall'invocazione multipla dell'algoritmo di rilevazione. L'ordine di eliminazione dei processi può essere dato da:
  - priorità
  - tempo di elaborazione
  - risorse utilizzate
  - risorse richieste per terminare l'elaborazione
  - numero di processi da terminare

7.5.4 Ripristino dello stallo con rilascio delle risorse

Si rilasciano le risorse di un processo fino a quando il ciclo non è rotto. Vi sono tre problemi:

- Selezione della vittima: bisogna scegliere la vittima a costo minimo.
- Rollback: bisogna fare il roll back del processo vittima e portarlo in uno stato sicuro. Determinare lo stato sicuro è difficile, quindi solitamente si riavvia il processo.
- Starvation: scegliendo una vittima in base al costo, può accadere che questa venga scelta più volte e vi sia un problema di starvation. La soluzione più comune è imporre un numero massimo di volte per cui il processo può essere scelto come vittima,

includendo questa metrica nel costo.