Sistemi operativi

Giacomo Fantoni
Telegram: @GiacomoFantoni
Filippo Momesso
Telegram: @Momofil31
Github: https://github.com/giacThePhantom/SistemiOperativi

26 maggio 2020

Indice

1 Introduzione **4**
- 1.0.1 Punti chiave nel progetto di calcolatori . . . 4
- 1.1 Storia dei sistemi operativi . . . 5
  - 1.1.1 Prima generazione (1946-1955) . . . 5
  - 1.1.2 Seconda generazione (1955-1965) . . . 6
  - 1.1.3 Terza generazione (1965-1980) . . . 7
  - 1.1.4 Quarta generazione (1980-1990) . . . 8

2 Componenti di un sistema operativo **9**
- 2.1 Servizi di gestione . . . 9
- 2.2 Interprete dei comandi (shell) . . . 10
  - 2.2.1 System calls . . . 10

3 Architettura di un sistema operativo **12**
- 3.1 Modello client-server . . . 13
- 3.2 macchine virtuali . . . 13
  - 3.2.1 Esokernel . . . 13
- 3.3 Processi e thread . . . 13
- 3.4 Processi e thread . . . 14

4 Processi e thread **15**
- 4.1 Processi . . . 15
  - 4.1.1 Immagine in memoria . . . 15
  - 4.1.2 Stati di un processo . . . 15
  - 4.1.3 Scheduling . . . 15
  - 4.1.4 Operazione di dispatch . . . 16
  - 4.1.5 Operazioni sui processi . . . 16
  - 4.1.6 Stati di un processo . . . 17
- 4.2 Threads . . . 17
  - 4.2.1 Multi-threading . . . 17
  - 4.2.2 Vantaggi dei thread . . . 17
  - 4.2.3 Stati di un thread . . . 18
  - 4.2.4 Implementazione dei thread . . . 18
  - 4.2.5 La libreria posix pthreads . . . 18
- 4.3 Relazione tra processi . . . 19
  - 4.3.1 Scambio di messaggi . . . 19
  - 4.3.2 Memoria condivisa . . . 20

4.4 Gestione dei processi del sistema operativo . . . 21
4.4.1 Kernel separato . . . 21
4.4.2 Kernel in processi utente . . . 21
4.4.3 Kernel come processo . . . 21

5 Scheduling CPU **22**
5.1 Tipi di scheduling . . . 22
5.1.1 Caratteristiche degli scheduler . . . 22
5.1.2 Scheduling a medio termine . . . 22
5.2 Scheduling della CPU . . . 22
5.2.1 Dispatcher . . . 23
5.2.2 Modello astratto del sistema . . . 23
5.2.3 Prelazione (preemption) . . . 23
5.2.4 Metriche di scheduling . . . 23
5.3 Algoritmi di scheduling . . . 23
5.3.1 First-Come, First-Served (FCFS) . . . 23
5.3.2 Shortest-Job-First (SJF) . . . 24
5.3.3 Scheduling a priorità . . . 24
5.3.4 Higher response ratio next (HRRN) . . . 24
5.3.5 Round robin (RR) . . . 24
5.3.6 Code multilivello . . . 25
5.3.7 Code multilivello con feedback . . . 25
5.3.8 Scheduling fair share . . . 25
5.3.9 Valutazione degli algoritmi . . . 25

6 Sincronizzazione tra processi **26**
6.0.1 Buffer: modello software . . . 26
6.0.2 Sezione critica . . . 26
6.1 Soluzioni software . . . 27
6.1.1 Algoritmo 2 . . . 27
6.1.2 Algoritmo 3 . . . 27
6.1.3 Algoritmo del fornaio . . . 28
6.2 Soluzioni hardware . . . 28
6.2.1 Test and Set . . . 29
6.2.2 Swap . . . 29
6.2.3 Test and Set con attesa limitata . . . 29
6.2.4 Conclusioni . . . 30
6.3 Semafori . . . 30
6.3.1 Semafori binari . . . 30
6.3.2 Semafori interi . . . 31
6.3.3 Implementazione . . . 32
6.3.4 Applicazioni . . . 32
6.3.5 Limitazioni . . . 33
6.4 Problemi classici dei semafori . . . 33
6.4.1 Produttore consumatore . . . 33
6.4.2 Dining philosophers . . . 34
6.4.3 Sleepy barber . . . 35
6.4.4 Limitazioni dei semafori . . . 36

6.5 Monitor . . . 36
6.5.1 Operazioni del monitor . . . 37
6.5.2 Limitazioni . . . 38
6.6 Sincronizzazione in Java . . . 39
6.6.1 Buffer produttore consumatore . . . 39
6.7 Conclusioni . . . 39
6.8 Problema degli scrittori e dei lettori . . . 39

7 Deadlock **41**
7.0.1 Condizioni necessarie . . . 41
7.1 Modello astratto: resource allocation graph (RAG) . . . 41
7.2 Gestione dei deadlock . . . 42
7.2.1 Prevenzione statica . . . 42
7.2.2 Prevenzione dinamica (avoidance) . . . 42
7.2.3 Rilevamento (detection) e ripristino (recovery) . . . 44
7.2.4 Algoritmo dello struzzo . . . 45
7.3 Conclusioni . . . 45
7.3.1 Partizionamento in classi . . . 46
7.3.2 Algoritmi specifici . . . 46

Capitolo 1

Introduzione

Un sistema operativo è un insieme di programmi che agiscono come intermediario tra l'hardware e l'uomo per facilitare l'uso del computer, rendere efficiente l'uso dell'hardware e evitare conflitti nell'allocazione di risorse tra hardware e software. Offre pertanto un ambiente per controllare e coordinare l'utilizzo dell'hardware da parte dei programmi applicativi. I suoi compiti principali sono di gestore di risorse e di controllore dell'esecuzione dei programmi e il corretto utilizzo del sistema. La struttura dei sistemi operativi è soggetta a notevole variabilità ad è adattabile a criteri di organizzazione estremamente differenti. È pertanto un programma sempre in esecuzione sul calcolatore che generalmente viene chiamato kernel al quale si aggiungono programmi di sistema e programmi applicativi. Nel progettare un sistema operativo si deve tipicamente fare un trade-off tra l'astrazione che semplifica l'utilizzo del sistema e l'efficienza.

Componenti

Un sistema di calcolo si può dividere in 4 componenti:

- Dispositivi fisici: sono composti dall'unità centrale di elaborazione (CPU), dalla memoria e dall'I/O.
- Programmi applicativi: definiscono il modo in cui utilizzare le risorse per risolvere i problemi computazionali da parte degli utenti.
- Sistema operativo: Controlla e coordina l'uso dei dispositivi da parte degli utenti.
- Utenti.

1.0.1 Punti chiave nel progetto di calcolatori

Punto di vista dell'utente

La percezione di un calcolatore dipende dall'interfaccia impiegata. Il più comune metodo di utilizzo è il PC composto da schermo, tastiera e mouse. Il sistema operativo in questo caso si progetta considerando la facilità di utilizzo con qualche attenzione alle prestazioni ma non all'utilizzo delle risorse. Nel caso di un utente che utilizza terminali connessi ad un mainframe condividendo risorse con altri utenti il sistema operativo andrebbe ottimizzato per massimizzare l'utilizzo delle risore.

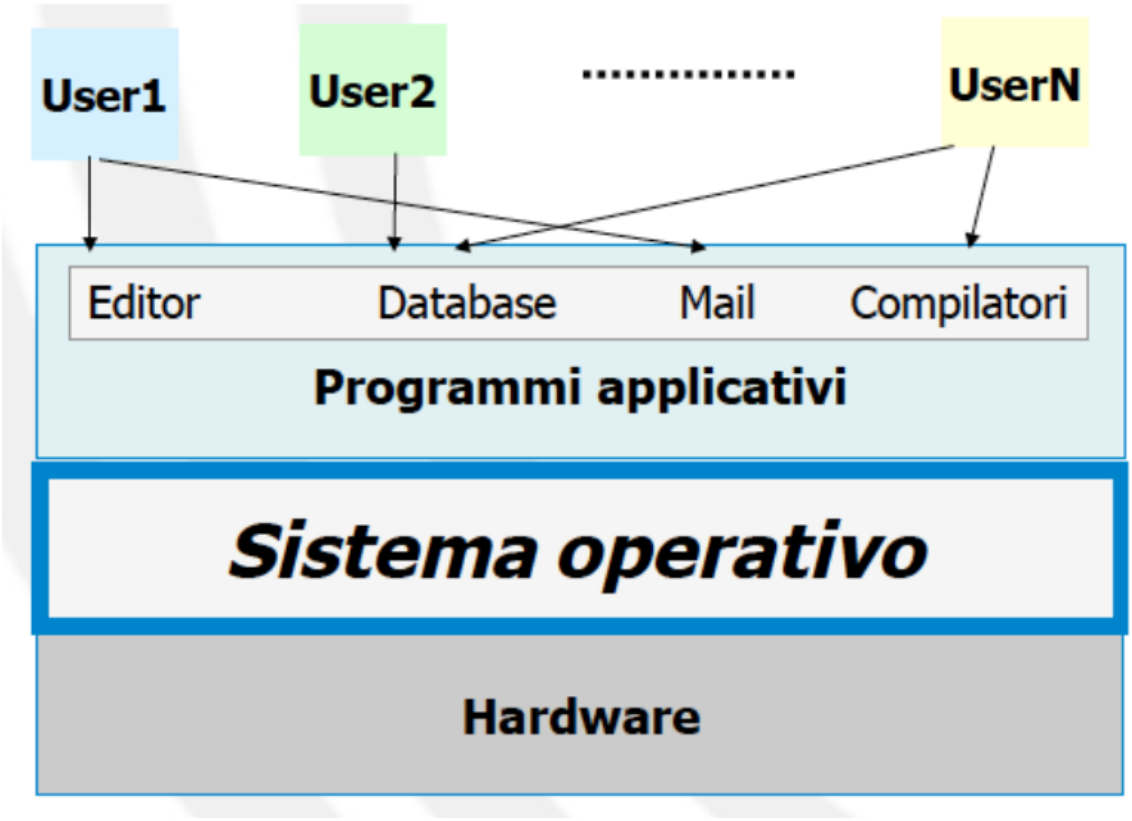

Figura 1.1: Stack del sistema operativo

Punto di vista del sistema

Il sistema operativo è il programma collegato più strettamente ai suoi elementi fisici ed è assimilabile ad un assegnatore di risorse o come programma di controllo che gestisce l'esecuzione dei programmi utente in modo da impedire che si verifichino errori o che il calcolatore sia utilizzato in modo scorretto.

1.1 Storia dei sistemi operativi

Si possono identificare 5 generazioni di calcolatori che riflettono direttamente l'evoluzione dei sistemi operativi dovuta all'aumento dell'utilizzo del processore.

1.1.1 Prima generazione (1946-1955)

In questa generazione i calcolatori erano enormi e a valvole, non esisteva il sistema operativo e l'operatore del calcolatore era equivalente al programmatore. L'accesso alla macchina era gestito tramite prenotazioni e i programmi venivano eseguiti da console caricando in memoria un'istruzione alla volta agendo su interruttori. Il controllo degli errori era fatto attraverso spie della console. Il processing era seriale.

Evoluzione

Durante la prima generazione si diffondono periferiche come il lettore/perforatore di schede, nastri e stampanti che rendono necessari programmi di interazione con periferiche detti device driver. Viene sviluppato del software come librerie di funzioni comuni e compilatori, linker e loader. Queste evoluzioni portano a una scarsa efficienza in quanto pur essendo la programmazione facilitata le operazioni erano complesse con tempi di setup elevati e un basso utilizzo relativo della CPU per eseguire il programma.

1.1.2 Seconda generazione (1955-1965)

In questa generazione si introducono i transistor nei calcolatori. Viene separato il ruolo di programmatore e operatore eliminando lo schema a prenotazione e il secondo permette di eliminare dei tempi morti. I programmi o jobs simili nell'esecuzione vengono raggruppati in batch in modo da aumentare l'efficienza ma aumentando i problemi in caso di errori o malfunzionamenti.

Evoluzione

Nasce l'automatic job sequencing in cui il sistema si occupa di passare da un job all'altro: il sistema operativo fa il lavoro dell'operatore e rimuove i tempi morti. Nasce pertanto il monitor residente, il primo esempio di sistema operativo, perennemente caricato in memoria. Le componenti del monitor erano i driver per i dispositivi di I/O, il sequenzializzatore dei job e l'interprete delle schede di controllo (per la loro lettura ed esecuzione). La sequenzializzazione avviene tramite un linguaggio di controllo o job control language attraverso schede o record di controllo.

Limitazioni

L'utilizzo del sistema risulta ancora basso a causa del divario di velocità tra I/O e CPU. Una soluzione è la sovrapposizione delle operazioni di I/O e di elaborazione. Nasce così l'elaborazione off-line grazie alla diffusione di nastri magnetici capienti e veloci. La sovrapposizione avviene su macchine diverse: da scheda a nastro su una macchina e da nastro a CPU su un'altra. La CPU viene limitata ora dalla velocità dei nastri, maggiore di quella delle schede.

Sovrapposizione tra CPU e I/O

È possibile attraverso un opportuno supporto strutturale far risiedere sulla macchina le operazioni off-line di I/O e CPU.

Polling Il polling è il meccanismo tradizionale di interazione tra CPU e I/O: avviene l'interrogazione continua del dispositivo tramite esplicite istruzioni bloccanti. Per sovrapporre CPU e I/O è necessario un meccanismo asincrono o richiesta I/O non bloccante come le interruzioni o interrupt e il DMA (direct memory access).

Interrupt e I/O In questo caso la CPU programma il dispositivo e contemporaneamente il dispositivo controllore esegue. La CPU, se possibile prosegue l'elaborazione. Il dispositivo segnala la fine dell'elaborazione alla CPU. La CPU riceve un segnale di interrupt esplicito e interrompe l'istruzione corrente salvando lo stato, salta a una locazione predefinita, serve l'interruzione trasferendo i dati e riprende l'istruzione interrotta.

DMA e I/O Nel caso di dispositivi veloci gli interrupt sono molto frequenti e porterebbero a inefficienza. Si rende pertanto necessario creare uno specifico controllore hardware detto DMA controller che si occupa del trasferimento di blocchi di dati tra I/O e memoria senza interessare la CPU. Avviene pertanto un solo interrupt per blocco di dati.

Buffering Si dice buffering la sovrapposizione di CPU e I/O dello stesso job. Il dispositivo di I/O legge o scrive più dati di quanti richiesti e risulta utile quando la velocità dell'I/O e della CPU sono simili. Nella realtà i dispositivi di I/O sono più lenti della CPU e pertanto il miglioramento è marginale.

Spooling Si dice spooling (simultaneous peripheral operations on-line))la sovrapposizione di CPU e I/O di job diversi, Nasce un problema in quanto i nastri magnetici sono sequenziali e pertanto il lettore di schede non può scrivere su un'estremità del nastro mentre la CPU legge dall'altra. Si devono pertanto introdurre dischi magnetici ad accesso causale. Il disco viene utilizzato come un buffer unico per tutti i job. Nasce il paradigma moderno di programma su disco che viene caricato in memoria, la pool di job e il concetto di job scheduling (la decisione di chi deve o può essere caricato su disco).

1.1.3 Terza generazione (1965-1980)

In questa generazione viene introdotta la multiprogrammazione e i circuiti integrati. La prima nasce dal fatto che un singolo job è incapace di tener sufficientemente occupata la CPU e pertanto si rende necessaria una loro competizione. Sono presenti più job in memoria e le fasi di attesa vengono sfruttate per l'esecuzione di un nuovo job. Con la presenza di più job nel sistema diventa possibile modificare la natura dei sistemi operativi: si passa ad una tendenza a sodisfare molti utenti che operano interattivamente e diventa importante il tempo di risposta di un job (quanto ci vuole perchè inizi la sua esecuzione). Nasce pertanto il multitasking o time sharing, estensione logica della multiprogrammazione in cui l'utente ha l'impressione di avere la macchina solo per sè e si migliora l'interattività con la gestione di errori e l'analisi di risultati. Nascono i sistemi moderni con tastiera che permette decisioni dell'evoluzione del sistema in base ai comandi dell'utente e un monitor che permette un output immediato durante l'esecuzione. Il file system inoltre è un'astrazione del sistema operativo per accedere a dati e programmi.

Protezione

Con la condivisione si rende necessario introdurre delle capacità di protezione per il sistema:

- I/O: programmi diversi non devono usare il dispositivo contemporaneamente, viene realizzata tramite il modo duale di esecuzione: modo user in cui i job non possono accedere direttamente alle risorse di I/O e modo supervisor o kernel in cui il sistema operativo può accedere a tali risorse. Tutte le operazioni di I/O sono privilegiate: le istruzioni di accesso invocano una system call, un interrupt software che cambia la modalità da user a supervisor e al termine della system call viene ripristinata la modalità utente.
- Memoria: un programma non può leggere o scrivere ad una zona di memoria che non gli appartiene: realizzata associando dei registri limite ad ogni processo, che possono essere modificati unicamente dal sistema operativo con istruzioni privilegiate.
- CPU: prima o poi il controllo della CPU deve tornare al sistema operativo, realizzata attraverso un timer legato ad un job, al termine del quale il controllo passa al monitor.

1.1.4 Quarta generazione (1980-1990)

- Diffusione di sistemi operativi per PC e workstation, utilizzo personale degli elaboratori e nascita delle interfacce grafiche (GUI).
- Sistemi operativi di rete in cui esiste una separazione logica delle risorse remote in cui l'accesso alle risorse remote è diverso rispetto a quello delle risorse locali.
- Sistemi operativi distribuiti: le risorse remote non sono separate logicamente e l'accesso alle risorse remote e locali è uguale.

Quinta generazione (1990- oggi)

Sistemi real-time vincolati sui tempi di risposta del sistema, sistemi operativi embedded per applicazioni specifiche, per piattaforme mobili e per l'internet of things.

Capitolo 2

Componenti di un sistema operativo

Un sistema operativo offre un ambiente on cui eseguire i programmi e fornire servizi che naturalmente variano in base al sistema operativo. Si possono comunque identificare alcune classi di servizi comuni.

2.1 Servizi di gestione

Gestione dei processi

Si intende per processo un programma in esecuzione che necessita di risorse e viene eseguito in modo sequenziale (un'istruzione alla volta). Si differenzia tra processi del sistema operativo e quelli utente. Il sistema operativo è responsabile della creazione, distruzione, sospensione, riesumazione e ella fornitura di meccanismi per la sincronizzazione e la comunicazione tra processi e fornisce meccanismi per la sincronizzazione.

Gestione della memoria primaria

La memoria primaria conserva dati condivisi dalla CPU e dai dispositivi di I/O. Un programma deve essere caricato in memoria prima di poter essere eseguito. Il sistema operativo è responsabile della gestione dello spazio di memoria (quali parti e da chi sono usate), ovvero della decisione su quale processo caricare in memoria in base allo spazio disponibile e dell'allocazione e rilascio dello spazio di memoria.

Gestione della memoria secondaria

Essendo la memoria primaria volatile e piccola si rende necessaria una memoria secondaria per mantenere grandi quantità di dati in modo permanente. È formata tipicamente da un insieme di dischi magnetici (che stanno per essere sostituiti dai dischi a stato solido -SSD- più veloci e performanti). Il sistema operativo è responsabile della gestione dello spazio libero su disco, dell'allocazione dello spazio su disco e dello scheduling degli accessi su disco.

Gestionde dell'I/O

Il sistema operativo nasconde all'utente le specifiche caratteristiche dei dispositivi di I/O per motivi di efficienza e protezione. Viene impegato un sistema per accumulare gli accessi ai dispositivi

(buffering), una generica interfaccia verso i device driver, con device driver specifici per alcuni dispositivi.

Gestione dei file

Le informazioni sono memorizzate su supporti fisici diversi controllati da driver con caratteristiche diverse. Si crea pertanto un file, un'astrazione logica per rendere conveniente l'uso di memoria non volatile grazie alla raccolta di informazioni correlate. Il sistema operativo è responsabile della creazione e cancellazione di file e directory, del supporto di operazioni primitive per la loro gestione (copia, incolla, modifica), della corrispondenza tra file e spazio fisico su disco e del salvataggio delle informazioni a scopo di backup.

Protezione

Si intende con protezione un meccanismo per controllare l'accesso alle risorse da parte di utenti e processi. Il sistema operativo deve definire quali sono gli accessi autorizzati e quali no, i controlli da imporre e fornire gli strumenti per verificare le politiche di accesso. La sicurezza di un sistema operativo comincia con l'obbligo di identificazione di ciascun utente che permette l'accesso alle risorse.

Rete (sistemi distribuiti)

Si intende per sistema distribuito una collezione di elementi di calcolo che non condividono nè la memoria nè un clock: le risorse di calcolo vengono connesse tramite una rete. Il sistema operativo è responsabile della gestione in rete delle varie componenti.

2.2 Interprete dei comandi (shell)

Vi sono due modi fondamentali per gli utenti di comunicare con il sistema operativo: un primo basato su un'interfaccia a riga d comando (o interprete dei comandi) e un secondo bastato su un'interfaccia grafica o GUI. Il primo lascia inserire direttamente agli utenti le istruzioni che il sistema deve eseguire. L'interprete dei comandi è più comunemente conosciuto come shell. La funzione principale dell'interprete è quella di prelevare ed eseguire il successivo comando impartito dall'utente. A questo livello si utano nuovi comandi per la gestione dei file che possono essere implementati internamente all'interprete o attraverso programmi speciali. Nel secondo caso l'interprete non capisce il comando in sè ma prende il nome per caricare l'opportuno file in memoria per eseguirlo.

Interfaccia grafica

L'interfaccia grafica è una modalità di comunicazione tra utente e il sistema operativo. È più intuitiva della riga di comando e la GUI è l'equivalente del desktop e rimane strettamente legata a mouse, tastiera e schermo. Puntando le icone col mouse è possibile accedere a file, cartelle e applicazioni.

2.2.1 System calls

Le chiamate di sistema costituiscono l'interfaccia di comunicazione tra il processo e il sistema operativo. Sono tipicamente scritte in linguaggi di alto livello come *C* o *C++*. I programmatori non si devono preoccupare dei dettagli di implementazione delle *sys.call* in quanto solitamente utilizzano

un'API (application program interface) che specifica un'insieme di funzioni a disposizione dei programmatori e dettaglia i parametri necessari all'invocazione di queste funzioni e i valori restituiti. Le due API più comuni sono *win32* e *POSIX API*, rispettivamente per Windows e UNIX. I parametri delle system calls possono essere pasati per valore o riferimento, ma vanno fisicamente messi da qualche parte: vengono pertanto posizionati nei registri (molto veloci, ma pochi e di dimensione fissa) nello stack del programma o in una tabella di memoria il cui indirizzo è passato in un registro o nello stack. Le system calls possono essere implementate in due modi:

- L'interprete legge il comando e cerca all'interno della shell per cercare il programma da avviare, non viene utilizzata in quanto rende necessario modifiche al kernel e non è efficiente.
- L'interprete legge il comando e possiede una tabella che collega tale comando al programma da avviare.

Le system calls si differiscono in cotrollo dei processi, gesione dei file, dei dispositivi, delle comunicazioni e della protezione. Un processo inoltre deve essere sia in grado di essere chiuso normalmente (*end*) che in modo anomalo (*abort*), con la conseguente generazione di un messaggio di errore e copia dello stato del processo abortito.

Capitolo 3

Architettura di un sistema operativo

Un principio importante è la separazione tra meccanismi e criteri o policy: i primi determinano come eseguire qualcosa, mentre i secondi cosa si deve fare. Questa distinzione è importante ai fini della flessibilità in quanto i criteri sono soggetti a cambiamenti di luogo e tempo.Principi importanti da tenere a mente durante lo sviluppo di sistemi operativi è il KISS (keep it small and simple), semplice dal punto di vista del codice, per mantenere affidabilità e mantenibilità e il POLA (principle of least privilege): un programm deve poter accedere unicamente ai dati strettamente necessari, fondamentale per il mantenimento di sicurezza e affidabilità.Questi cambiamenti devono richiedere il cambio di meccanismi solo nel caso pessimo. Le principali tipologie di architettura di sistemi operativi sono:

- Sistemi monolitici: sistemi senza gerarchia e con un unico strato software tra utente e hardware, tutti i componenti sono sullo stesso livello e possono chiamarsi vicendevolmente. In questa tipologia il codice dipende direttamente dall'architettura hardware e rende test e debugging complesso.
- Sistemi a struttura semplice: si ha un minimo di gerarchia e di struttura, non esiste ancora la suddivisione modalità utente e modalità kernel. Questa struttura è mirata a ridurre i costi di sviluppo e manutenzione.
- Sistema operativo a livelli. I servizi sono organizzati su livello gerarchici con al livello più alto l'interfaccia utente e al più basso l'hardware. Ogni livello può utilizzare funzioni di livelli inferiori. La modularità rende più semplice la manutenzione, ma diminuisce l'efficienza e richiede un'attenta definizione dei livelli.
- Sistemi basati su kernel: vengono utilizzati due livelli, i cui servizi sono distinti tra kernel e non-kernel. Presenta i vantaggi del sistema a livelli come modularità ma senza avere troppi livelli. Tra i servizi al di fuori del kernel non si trova nessuna organizzazione e si tende a pensare al kernel come a una struttura monolitica.
- Sistemi a micro-kernel: i micro-kernel sono un insieme di piccoli kernel che svolgono poche funzioni fondamentali. Occupano meno memoria e sono più affidabili e mantenibili, ma presentano scarse prestazioni: ogni volta che si deve accedere ad un programma applicativo si deve fare un cambio tra modalità kernel a modalità utente e viceversa una volta terminato il processo. Vengono utilizzati da quando le prestazioni di CPU e memoria sono sufficienti a non far percepire all'utente il cambio di modalità. La modularità offre inoltre maggiore sicurezza e portabilità.

3.1 Modello client-server

Una variazione del'idea del microkernel è quella di distinguere due classi di processi: i server che forniscono un servizio e i client che lo utilizzano. Spesso il livello più basso è un microkernel, ma non è richiesto. La comunicazione tra client e server avviene tramite scambio di messaggi: per ottenere un servizio il client deve costruire un messaggio e inviarlo al server, che quando lo riceve restituisce la risposta. Se client e server operano sulla stessa macchina sono possibili delle ottimizzazioni.

3.2 macchine virtuali

Le macchine virtuali sono introdotte nel 1972 da IBM come estremizzazione dell'approccio a livelli persata per offrire un sistema di timesharing multiplo ovvero che permette la multiprogrammazione e una macchina estesa che abbia un interfaccia più semplice del solo hardware. La base della macchina virtuale è la separazione di questi due aspetti. La sua parte centrale era il virtual machine monitor che permette la multiprogrammazione offrendo diverse macchine virtuali al livello superiore. Un tipo di macchina virtuale utilizza un type 1 hypervisor, usato comuniemente che si trova sull'hardware e permette di eseguire diversi sistemi operativi sulla stessa macchina. Il type 2 hypervisor viene utilizzato su un sistema operativo host nel quale l'hypervisor installa il sistema operativo guest in un disco virtuale che il sistema host vede come un file di grandi dimensioni. La differenza tra i due tipi di hypervisor sta nel fatto che quello di tipo 1 si trova direttamente sull'hardware, mentre il tipo 2 viene creato in un sistema operativo host.

3.2.1 Esokernel

Piuttosto che clonare la macchina, un'altra strategia per ottenere un sottinsieme delle risorse è partizionarla. Al livello più basso si trova un programma eseguito in modalità kernel che alloca risorse alle virtual machine e controlla le loro prove di utilizzarle. Il vantaggio dell'esokernel è che evita un livello di mappatura: in altri metodi è necessaria una mappatura tal disco virtuale a quello fisico, come per tutte le altre risorse.

3.3 Processi e thread

Attributi (Process Control Block): contiene un puntatore alla cella di memoria che contiene l'immagine, contiene lo stato del processo in un determinato momento, contiene i registri, le informazioni relative allo stato dell'I/O. Un processo può essere in diversi stati. All'inizio il processo viene creato, poi può essere in esecuzione se gli viene assegnata la CPU o non in esecuzione se non gli viene assegnata la CPU. Il Dispatcher assegna la CPU ai processi che sono pronti, ma non in esecuzione (posso avere diversi processi in memoria, ma solo uno per volta usa la CPU e quindi è effettivamente in esecuzione, a meno di CPU multicore). Quando un processo è pronto e viene creato viene messo nella ReadyQueue (oppure nella coda di un dispositivo cioè la coda in cui viene messo un processo che sta aspettando di accedere a un determinato dispositivo). In realtà esistono diverse code in cui può essere messo un processo. Dispatch e Scheduler sono due componenti diversi, lo scheduler sceglie mentre il dispatcher implementa questa cosa. Context-Switch: salvo tutto quello che c'era nel PCB, tutto quello che stavo utilizzando al momento dell'esecuzione. Due operazioni fondamentali che fa un sistema operativo è la creazione e la terminazione del processo. Ogni processo può creare altri processi e questi prendono il nome di processi figli. Il figlio può essere creato in modalità sincrona, cioè finché il figlio è in vita io non faccio niente, oppure in modalità asincrona cioè io creo il figlio

e continuo la mia esecuzione. Con la fork il figlio lo creo esattamente uguale al padre, con la exec posso caricare sul figlio un programma diverso rispetto al padre. Con la wait creo un'esecuzione sincrona tra padre e figlio. Una fork può fallire perché o il padre non ha abbastanza memoria o perché non ha i privilegi per creare un figlio. L'exec cambia l'immagine in memoria del figlio.

3.4 Processi e thread

Attributi (Process Control Block): contiene un puntatore alla cella di memoria che contiene l'immagine, contiene lo stato del processo in un determinato momento, contiene i registri, le informazioni relative allo stato dell'I/O. Un processo può essere in diversi stati. All'inizio il processo viene creato, poi può essere in esecuzione se gli viene assegnata la CPU o non in esecuzione se non gli viene assegnata la CPU. Il Dispatcher assegna la CPU ai processi che sono pronti, ma non in esecuzione (posso avere diversi processi in memoria, ma solo uno per volta usa la CPU e quindi è effettivamente in esecuzione, a meno di CPU multicore). Quando un processo è pronto e viene creato viene messo nella ReadyQueue (oppure nella coda di un dispositivo cioè la coda in cui viene messo un processo che sta aspettando di accedere a un determinato dispositivo). In realtà esistono diverse code in cui può essere messo un processo. Dispatch e Scheduler sono due componenti diversi, lo scheduler sceglie mentre il dispatcher implementa questa cosa. Context-Switch: salvo tutto quello che c'era nel PCB, tutto quello che stavo utilizzando al momento dell'esecuzione. Due operazioni fondamentali che fa un sistema operativo è la creazione e la terminazione del processo. Ogni processo può creare altri processi e questi prendono il nome di processi figli. Il figlio può essere creato in modalità sincrona, cioè finché il figlio è in vita io non faccio niente, oppure in modalità asincrona cioè io creo il figlio e continuo la mia esecuzione. Con la fork il figlio lo creo esattamente uguale al padre, con la exec posso caricare sul figlio un programma diverso rispetto al padre. Con la wait creo un'esecuzione sincrona tra padre e figlio. Una fork può fallire perché o il padre non ha abbastanza memoria o perché non ha i privilegi per creare un figlio. L'exec cambia l'immagine in memoria del figlio.

Capitolo 4

Processi e thread

4.1 Processi

Si dice processo un istanza di un programma in esecuzione, un concetto dinamico eseguito in modo sequenziale. In un sistema multiprogrammato i processi evolvono in modo concorrente a causa delle risorse fisiche e logiche limitate. Il sistema operativo stesso consiste di più processi.

4.1.1 Immagine in memoria

Il processo consiste di istruzioni (sezione Codice o Testo, la parte statica del codice), dati (sezione Dati, le variabili globali), lo Stack (chiamate a procedure e paremetri e variabili locali), lo Heap (la memoria allocata dinamicamente) e gli attributi (id, stato, controllo).

Attributi

All'interno del sistema operativo ogni processo è rappresentato dal processo control block (PCB) che contiene le informazioni riguardanti lo stato del processo, il program counter, i valori dei registri, le informazioni sulla memoria, informazioni sullo stato dell'I/O, informazioni sull'utilizzo del sistema e informazioni di scheduling come la priorià.

4.1.2 Stati di un processo

Durante la sua esecuzione un processo evolve attraverso diversi stati con un diagramma specifico al sistema operativo. Lo schema base contiene un nuovo processo che deve essere ammesso nel sistema, lo stato non in esecuzione che viene messo in esecuzione dall'operazione di dispatch (l'operazione inversa avviene dalla pausa). La stato finito viene ottenuto dopo la terminazione. Un processo può essere messo in pausa a causa di un time out, caso nel quale viene messo direttamente tra i processi pronti o essere in attesa del completamento di un altro processo che compie un evento per cui diventa pronto.

4.1.3 Scheduling

Lo scheduling è l'operazione di selezione del processo da eseguire nella CPU al fine di garantire la multiprogrammazione (con l'obiettivo di massimizzare l'uso della CPU con più di un processo in memoria) e il time-sharing (con l'obiettivo di commutare frequentemente la CPU tra processi

in modo che ognuno creda di avere la CPU tutta per sè). Il long term scheduler o job scheduler seleziona quali processi devono essere trasferiti nella coda dei processi pronti, mentre lo short-term scheduler o CPU scheduler seleziona quali sono i prossimi processi ad essere eseguiti e alloca la CPU di conseguenza. Il secondo è invocato molto di frequente e pertanto deve essere veloce, mentre il primo viene invocato meno frequentemente e può essere lento. Il long-term scheduler controlla il grado di multiprogrammazione.

Code di scheduling

Ogni processo è inserito in una serie di code: la coda dei processi pronti (ready queue) che è la coda dei processi pronti per l'esecuzione e la coda di un dispositivo ovvero la coda dei processi in attesa che il dispositivo si liberi. All'inizio il processo è nella ready queue fino a quando non viene selezionato per essere eseguito (dispatched). Durante l'esecuzione può accadere che il processo necessiti di I/O e venga inserito in una coda di un dispositivo, il processo termina il quanto di tempo, viene rimosso forzatamente dalla CPU e reinserito nella ready queue, il processo crea un figlio e ne attende la terminazione, il processo si mette in attesa di un evento.

4.1.4 Operazione di dispatch

Durante il dispatch deve avvenire un cambio di contesto: salvataggio del PCB del processo che esce e caricamento del PCB del processo che entra. Dopo questo avviene il passaggio alla modalità utente (all'inizio della fase di dispatch il sistema si trova in modalità kernel). Infine avviene il salto all'istruzione da eseguire del processo appena arrivato nella CPU.

Cambio di contesto

Il passaggio della CPU a un nuovo processo causa la registrazione dello stato del processo vecchio e il caricamento dello stato del nuovo. Il tempo necessario al cambio di contesto è un puro overhead: il sistema non compie alcun lavoro utile e la durata del cambio di contesto dipende molto dall'architettura.

4.1.5 Operazioni sui processi

Creazione di un processo

Un processo può creare un figlio che ottiene risorse dal sistema operativo o dal padre (sparizione, condivisione). Il figlio può essere eseguito in modalità sincrona (il padre attende la terminazione dei figli) o in modalità asincrona (evoluzione parallela e concorrente di padre e figli). In UNIX per creare un figlio si usa:

- System call *fork*: crea un figlio che è un duplicato esatto del padre.
- System call *exec*: carica sul figlio un programma diverso da quello del padre.
- System call *wait*: per l'esecuzione sincrona tra pare e figlio.

Terminazione di un processo

Un processo può terminare in tre casi: quando finisce la sua esecuzione, quando è terminato forzatamente dal padre per eccesso nell'uso delle risorse, il compito richiesto al figlio non è più necessario e il padre termina e il sistema operativo non permette ai figli di sopravvivere al padre o quando il

processo è terminato forzatamente dal sistema operativo in caso di chiusura da parte dell'utente o a causa di errori.

4.1.6 Stati di un processo

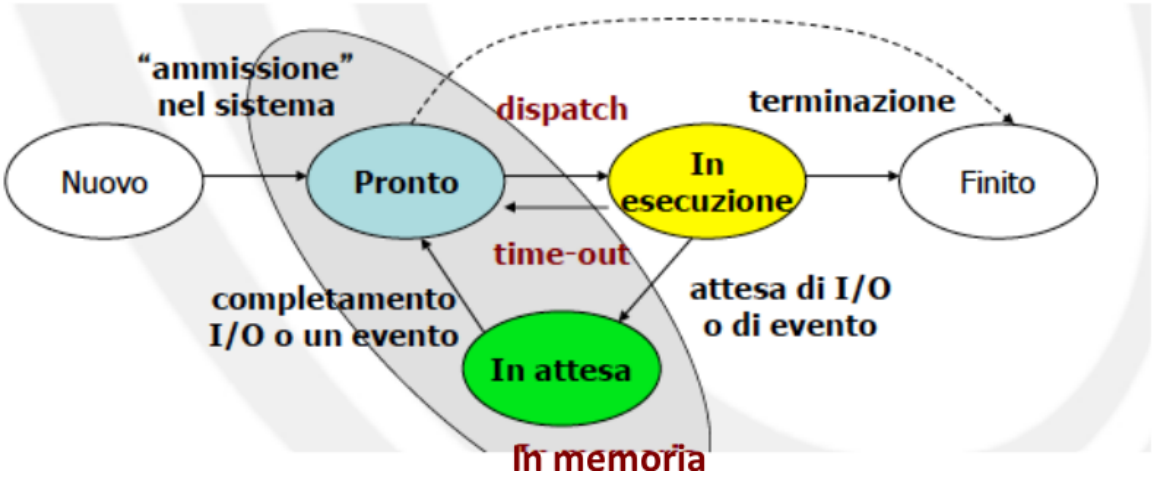

Figura 4.1: Stack del sistema operativo

4.2 Threads

Si noti come un processo unisce i concetti del possesso delle risorse e utilizzo della CPU, caratteristiche indipendenti e possono essere considerate separatamente: si considera il thread come l'unità minima di utilizzo della CPU e il processo come unità minima di possesso delle risorse. Ad un processo sono dunque associati spazio di indirizzamento e risorse del sistema, mentre a un singolo thread lo stato di esecuzione, contatore del programma, insieme di registri e lo stack. Le thread condividono tra di loro lo spazio di indirizzamento e risorse e stato del processo.

4.2.1 Multi-threading

Il multithreading rappresenta la capacità di un sistema operativo di supportare più thread per un singolo processo e ha come conseguenza la separazione tra flusso di esecuzione e spazio di indirizzamento: i processi con thread multipli hanno pi˘u flussi associati ad un singolo spazio di indirizzamento.

4.2.2 Vantaggi dei thread

I thread permettono di ridurre il tempo di risposta in quanto il blocco di un thread permette comunque ad altri thread di continuare la propria esecuzione. Oltre a quello permettono di condividere delle risorse in quanto thread di uno stesso processo condividono la memoria senza dover introdurre tecniche esplicite di condivisione (necessario per i processi) e pertanto la sincronizzazione e comunicazione risulta agevolata. La creazione, la terminazione e il contex switch risulta più veloce tra i thread rispetto che tra i processi. Presentano infine grandi vantaggi per quanto riguarda l'aumento di parallelismo se l'esecuzione avviene su multiprocessore.

4.2.3 Stati di un thread

Un thread possiede gli stessi stati di un processo: pronto, in esecuzione e in attesa, ma in generale lo stato del processo può non coincidere con lo stato di un suo thread.

4.2.4 Implementazione dei thread

I thread possono essere implementati a user-level (one to many), a kernel-level (one to one) o ibrida (m to n).

Thread user-level

La gestione viene affidata alle applicazioni, il kernel ignora l'esistenza dei thread e le funzionalità sono disponibili tramite una libreria di programmazione. I vantaggi di questa implementazione riguardano la non necessità di passare in kernel-mode per utilizzare i thread prevenendo due mode switch aumentando l'efficienza, il meccanismo di scheduling può variare in base alle necessità applicative e possono essere eseguiti su qualsiasi sistema operativo senza modificare il kernel. Come svantaggi presentano il fatto che il blocco di un thread blocca l'intero processo (superabile con accorgimenti specifici) e non è possibile sfruttare il multiprocessore: lo scheduling di un thread è presente sempre sullo stesso processore: un solo thread è in esecuzione per ogni processo.

Thread kernel-level

La gestione viene affidata al kernel e le applicazioni usano i thread tramite system call. I vantaggi sono lo scheduling a livello thread in quanto il blocco di un thread non blocca l'intero processo, più thread dello stesso processo possono essere in esecuzione in contemporanea su CPU diverse e le funzioni del sistema operativo stesse possono essere multithreaded. Come svantaggi presenta la scarsa efficienza in quanto il passaggio tra i thread implica un passaggio attraverso il kernel.

4.2.5 La libreria posix pthreads

Per usare i pthreads in un programma C è necessario includere la libreria *<pthread.h>* e occorre linkare la libreria libpthread usando l'opzione *-pthread*.

Creazione di un thread

Un thread ha vari attributi che possono essere cambiati come la sua priorità e la dimensione del suo stack contenuti in un oggetto di tipo *pthread_ attr_ t* e la system call *int pthread_ attr_ init(pthread_ attr_ t *attr);* inizializza con i valori di default un contenitore di attributi **attr* che può essere passato alla system call per creare un nuovo thread. La creazione vera e propria avviene tramite la system call *pthread_ create* che accetta quattro argomenti:

- Una variabile di tipo *pthread_ t* che contiene l'identificativo del thread che viene creato.
- Un oggetto **attr* che contiene gli attributi da dare al thread creato (*NULL* per i default).
- Un puntatore alla routine che contiene il codice che deve essere eseguito dal thread.
- Un puntatore all'argomento che si vuole passare alla routine stessa.

terminazione di un thread

Un thread termina quando finisce il codice della routine specificata all'atto della creazinoe del thread stesso o quando nel codice della routine si chiama la system call di terminazione *void pthread_ exit(void *value_ ptr);*. Quando termina il thread restituisce il valore di return specificato nella routine o se chiama la system call il valore passato a quella come argomento.

Sincronizzazione fra thread

Un thread può sospendersi in attesa della terminazione di un altro thread chiamando la system call *int pthread_ join(pthread_ t thread, void **value_ ptr);* dove il primo argomento è l'identificativo del thread di cui si vuole attendere la terminazione (appartenente allo stesso processo) e il secondo valore restituito dal thread che termina.

Sincronizzazione fra thread

Condivisione dello spazio logico Due thread dello stesso processo condividono lo stesso spazio di indirizzamento e le stesse variabili, cosa che nei processi tradizionali è possibile solo usando esplicitamente un segmento di memoria condivisa.

Sincronizzazione dell'esecuzione La sincronizzazione può avvenire attraverso diversi strumenti come i semafori (non disponibili nell'ultima versione dello standard) e meccanismi di sincronizzazione strutturati come le variabili condizionali.

4.3 Relazione tra processi

Nei processi indipendenti si nota un'esecuzione deterministica che dipende solo dal proprio input che non influenza, nè viene influenzata da altri processi con i quali non avviene nessuna condivisione dei dati. Nei processi cooperanti invece un processo può essere influenzato e influenzare altri processi e l'esecuzione risulta non deterministica e non riproducibile. I processi cooperanti sono naturalmente necessari per la condivisione di informazioni, permettono di accelerare il calcolo eseguendo parallelamente subtask su multiprocessore, permettono la modularità con funzioni distinte su vari processi e sono convenienti. Si devono naturalmente stabilire dei modelli di comunicazione rigorosi: lo scambio di messaggi e la condivisione della memoria.

4.3.1 Scambio di messaggi

Il passaggio di messaggi è un meccanismo utilizzato dai processi per comunicare e sincronizzare le loro azioni e in questo caso i processi comunicano tra loro senza condividere variabili. Le IPC forniscono le operazioni di *send(message)* e di *receive(message)* con la lunghezza del messaggio fissa o variabile. Se P e Q desiderano comunicare devono stabilire un canale di comunicazione e scambiarsi messaggi attraverso send/receive. Il canale di comunicazione deve essere implementato naturalmente sia in maniera fisica che logica.

Comunicazioni dirette

Nella comunicazione i processi devono nominarsi esplicitamente e può essere simmetrica: *send(P1, message)* e *receive(P2, message)* dove P1 e P2 sono gli id rispettivamente di destinatario e mittente o può essere asimmetrica: *send(P1, message)* e *receive(id, message)* dove in id viene salvato il nome

del processo che ha eseguito la send. SSi nota come se un processo cambia nome si devono ricodificare tutti i messaggi di send e receive.

Comunicazioni indirette

I messaggi sono spediti e ricevuti da mailboxes o porte tali che ogni mailbox ha un unico id e i processi possono comunicare solo se condividono una mailbox. Le operazioni permettono di creare una nuova mailbox, *send* e *receive* tramite mailbox ed eliminare una mailbox. Le primitive sono definite come *send(A, message)*: spedire un messaggio alla mailbox A e *receive(A, message)*, ricevere un messaggio dalla mailbox A. Il canale di comunicazione può essere stabilito solo se i processi condividono una mailbox comune, ogni canale può essere associato a molti processi, ogni coppia di processi può condividere molti canali di comunicazione e i canali possono essere unidirezionali o bi-direzionali. Le concorrenze rispetto alla ricezione si risolvono permettendo ad un canale di essere associato al più con due processi, permettendo a un solo processo alla volta di eseguire la ricezione o permettendo al sistema di selezionare in modo arbitrario il ricevente: il mittente è notificato di chi ha ricevuto il messaggio.

Sincronizzazione

Lo scambio di messaggi può essere bloccante o non bloccante.

Bloccante (sincrono) La send bloccante blocca il mittente fino a che il messaggio è ricevuto e la receive bloccante blocca il ricevente fino a che il messaggio è disponibile.

Non bloccante (asincrono) La send non bloccante permette al mittente di spedire il messaggio e continuare la sua esecuzione e la receive non bloccante permette al ricevente di ricevere un messaggio valido o nulla permettendogli di continuare la sua esecuzione in ogni caso.

4.3.2 Memoria condivisa

In POSIX il processo crea prima il segmento di memoria condivisa *segment id = shmget(IPC_PRIVATE, size, S_IRUSR | S_IWUSR);* il processo che vuole accedere alla memoria condivisa deve attaccarsi: *shared memory = (char *) shmat(id, NULL, 0);* ora il processo può scrivere nel segmento condiviso attraverso *sprintf* e quando ha finito il processo stacca il segmento di memoria dal proprio spazio di indirizzi: *shmdt(shared memory);* e per rimuovere il segmento di memoria si usa *shmctl(shm_id, IPC_RMID, NULL);*

Pipe

Le pipe agiscono come condotte che permettono a due processi di comunicare.

Pipe ordinarie Le pipe ordinarie permettono la comunicazione in stile produttore-consumatore: il produttore scrive ad un'estremità, mentre il consumatore legge all'altra estremità. Sono unidirezionali e richiedono una relazione parent-child tra i processi comunicanti.

Pipe con nome Le pipe con nome permettono comunicazioni bidirezionali senza relazione parent-child. Sono utilizzabili da più processi.

4.4 Gestione dei processi del sistema operativo

Il sistema operativo è un programma e il kernel può essere eseguito separatamente, all'interno di un processo utente o come processo.

4.4.1 Kernel separato

Il kernel viene eseguito al di fuori di ogni processo con uno spazio riservato in memoria, prende il controllo del sistema ed è sempre in esecuzione in modo privilegiato. Il concetto di processo viene applicato solo ai processi utente.

4.4.2 Kernel in processi utente

Si considerano i servizi del sistema operativo come procedure chiamabili da programmi utente accessibili in modalità protetta (kernel mode). L'immagine dei processi prevede il kernel stack per gestire il funzionamento di un processo in modalità protetta e il codice e dati del sistema operativo condiviso tra i processi. Offre un vantaggio di efficienza in quanto dopo interrupt o trap durante l'esecuzione è necessario un solo mode switch in cui il sistema passa da user mode a kernel mode e viene eseguita la parte di codice relativa al sistema operativo senza context switch. Dopo il completamento del suo lavoro il sistema operativo può decidere di riattivare lo stesso processo utente (mode switch) o un altro (context switch).

4.4.3 Kernel come processo

Si considerano i servizi del sistema operativo come processi individuali eseguiti in modalità protetta, una minima parte del sistema operativo deve essere eseguita al di fuori di tutti i processi (lo scheduler). È vantaggioso per sistemi multiprocessore dove processi del sistema operativo possono essere eseguiti su un processore ad hoc.

Capitolo 5

Scheduling CPU

Si intende per scheduling l'assegnazione di attività nel tempo: l'utilizzo della multiprogrammazione impone l'esistenza di una strategia per regolamentare l'ammissione dei processi nel sistema e l'ammissione dei processi all'esecuzione.

5.1 Tipi di scheduling

Gli scheduler si dividono in a lungo termine (job scheduler) che seleziona quali processi devono essere portati dalla memoria alla ready queue e gli scheduler a breve termine (CPU scheduler) che seleziona quale processo deve essere eseguito dalla CPU.

5.1.1 Caratteristiche degli scheduler

Lo scheduler a breve termine è invocato spesso e pertanto deve essere veloce, mentre lo scheduler a lungo termine è invocato più raramente e può essere più lento in quanto controlla il grado di multiprogrammazione e il mix di processi. Il secondo può essere I/O bound con molto I/O e molti brevi burst di CPU o CPU-bound, con molti calcoli e pochi lunghi burst di CPU. In sistemi con risorse limitate lo scheduler può essere assente.

5.1.2 Scheduling a medio termine

I sistemi operativi con memoria virtuale prevedono un livello intermedio di scheduling per la momentanea rimozione forzata (swapping) di un processo dalla CPU per ridurre il grado di multiprogrammazione.

5.2 Scheduling della CPU

Il CPU scheduler è un modulo del sistema operativo che seleziona un processo tra quelli in memoria pronti per l'esecuzione e gli alloca la CPU e data la frequenza di invocazione è una parte critica del sistema operativo in quanto necessita algoritmi di scheduling.

5.2.1 Dispatcher

Il dispatcher è un modulo del sistema operativo che passa il controllo della CPU al processo scelto dallo scheduler: fa lo switch del contesto, il passaggio alla modalità user e il salto alla opportuna locazione nel programma per farlo ripartire. Nasce una latenza di dispatch, il tempo necessario al dispatcher per fermare un processo e farne ripartire un altro. Deve pertanto essere la più bassa possibile.

5.2.2 Modello astratto del sistema

Nel sistema si trova un alternanza di burst (sequenza) di CPU e I/O. Nel modello a cicli di burst CPU-I/O l'esecuzione di un processo consiste dell'alternanza ciclica di un burst di CPU e di uno di I/O. I CPU burst sono distribuiti esponenzialmente con molti burst brevi o pochi lunghi.

5.2.3 Prelazione (preemption)

Si intende per prelazione il rilascio forzato della CPU. Quando non è presente il processo che detiene la CPU non la rilascia fino al termine del burst, mentre quando è presente può essere forzato a rilasciarla prima del termine.

5.2.4 Metriche di scheduling

Per misurare la qualità dello scheduling si considerano:

- Utilizzo della CPU, con l'obiettivo di tenerla più occupata possibile.
- Throughput, il numero di processi completati per unità di tempo.
- Tempo di attesa (t_w), la quantità di tempo totale spesa da un processo nella coda di attesa influenzato dall'algoritmo di scheduling.
- Tempo di completamento (turnaround t_t), il tempo necessario ad eseguire un particolare processo dal momento della sottomissione al momento del completamento.
- Tempo di risposta (t_r), il tempo trascorso da quando una richiesta è stata sottoposta al sistema fino alla prima risposta del sistema spesso.

Per ottimizzare lo scheduling si deve pertanto massimizzare l'utilizzo della CPU e il throughput e minimizzare i tempi di turnaround, attesa e risposta.

5.3 Algoritmi di scheduling

5.3.1 First-Come, First-Served (FCFS)

La coda dei processi è una coda FIFO e il primo processo arrivato è il primo ad essere servito, ha un'implementazione semplice. Uno svantaggio ovvio è il cosiddetto effetto convoglio: i processi brevi si accodano a processi lunghi preesistenti e sorgono problemi in contesti interattivi.

5.3.2 Shortest-Job-First (SJF)

Associa ad ogni processo la lunghezza del prossimo burst di CPU e il processo con il burst più breve viene selezionato per l'esecuzione. Ne esistono due schemi uno non preemptive e uno preemptive in cui se arriva un nuovo processo con un burst di CPU più breve del tempo che rimane da seguire al processo un esecuzione questo viene rimosso dalla CPU per fare spazio a quello appena arrivato (algoritmo di shortest-remaining-time-first SRTF). SJF è ottimo in quanto permette il minimo tempo medio di attesa.

Calcolo del prossimo burst di CPU

Di questo è possibile solo una stima utilizzando le lunghezze dei burst precedenti come proiezione di quelli futuri e utilizzando una media esponenziale: sia t_n la lunghezza reale dell'n-esimo burst, τ_{n+1} il valore stimato per il prossimo burst e α un coefficiente tale che $0 < \alpha < 1$, allora:

$$\tau_{n+1} = \alpha \cdot t_n + (1 - \alpha) \cdot \tau_n$$

Se $\alpha = 0$ la storia recente non viene utilizzata e se $\alpha = 1$ conta solo l'ultimo burst reale. Si nota come ogni termine successivo pesa meno del predecessore.

5.3.3 Scheduling a priorità

In questo algoritmo viene associata una priorità a ogni processo e la CPU viene allocata al processo con priorità più alta. Può essere preemptive o non-preemptive. Si noti come SJF è uno scheduing a priorità data da $\frac{1}{\text{lunghezza del burst successivo}}$. Il comando *nice* in Linux modifica la priorità. Le politiche di assegnamento della priorità possono essere interne al sistema operativo (limiti di tempo, requisiti di memoria, numero di file aperti) o esterne (importanza del processo, motivi economici o politici). Può nascere il problema di starvation in quanto processi a bassa priorità possono non essere mai eseguiti, risolti con un aumento della priorità con il passare del tempo.

5.3.4 Higher response ratio next (HRRN)

È un algoritmo a priorità non-preentive: la priorità

$$R = \frac{t_{attesa} + t_{burst}}{t_{burst}} = 1 + \frac{t_{attesa}}{t_{burst}}$$

è maggiore per valori di R più alti e dipende anche dal tempo di attesa e pertanto va ricalcolata al termine di un processo se nel frattempo ne sono arrivati altri o alt ermine di un processo. Supera il favoritismo di SJF verso i job corti favorendo i processi che completano in poco tempo o quelli che hanno atteso molto.

5.3.5 Round robin (RR)

Questo algoritmo basa lo scheduling su time-out: a ogni processo viene assegnato un quanto del processo di CPU tra i 10 e i 100 millisecondi e al termine del quanto il processo è prelazionato e messo nella ready queue, una coda circolare. Se ci sono n processi nella coda e il quanto è q ogni processo ottiene $\frac{1}{n}$ del tempo di CPU in blocchi di q unità di tempo alla volta e nessun processo attende più di $(n-1)1$ unità di tempo. È di semplice implementazione (FCFS con prelazione) il quanto va scelto con cura: se troppo grande diventa equivalente a FCFS, se troppo piccolo nasce

troppo overhead per il context switch, un valore ragionevole è uno tale che l'80% dei burst siano minori di q. Il tempo di turnaround è maggiore o uguale di SJF e quello di risposta minore o uguale di SJF.

5.3.6 Code multilivello

È una classe di algoritmi in cui la ready queue è partizionata in più code e ogni coda ha il suo algoritmo di scheduling. Si rende necessario anche uno scheduling tra code, che può essere a priorità fissa con la possibilità di starvation per le code a priorità bassa o basato su time slice in cui ogni coda ottiene un quanto di tempo di CPU che può usare per schedulare i suoi processi.

5.3.7 Code multilivello con feedback

Sono code multilivello in cui un processo può spostarsi da una coda all'altra a seconda delle sue caratteristiche (usato anche per implementare l'aging). Lo scheduler ha come parametri il numero delle code, gli algoritmi per ogni coda, criteri per la promozione o degradazione di un processo e i criteri per definire la coda di ingresso di un processo.

5.3.8 Scheduling fair share

Si noti come le politiche precedenti di scheduling sono orientate al processo ma non alle applicazioni (che possono essere composte da più processi). Fair share tenta di fornire equità alle applicazioni le vengono suddivise tra gruppi di processi (le applicazioni).

5.3.9 Valutazione degli algoritmi

Modello deterministico (analitico)

Questa valutazione è basata sull'algoritmo e su un preciso carico di lavoro, definisce le prestazioni di ogni algoritmo per tale carico specifico. Vengono utilizzate per illustrare gli algoritmi e richiedono conoscenze troppo specifiche sulla natura dei processi.

Modello a reti di code

Non esiste un preciso gruppo di processi per utilizzare il modello deterministico ma è possibile determinare le distribuzioni di CPU e I/O burst. Il sistema di calcolo è descritto come una rete di server ognuno con la propria coda e si usano formule matematiche che indicano la probabilità che si verifichi un determinato CPU burst e la distribuzione dei tempi di arrivo nel sistema dei processi da cui è possibile ricavare utilizzo, throughput medio, tempi di attesa e altri parametri.

Simulazione

Si programma un modello del sistema utilizzando dati statistici o reali (precisa ma costosa).

Implementazione

È l'unico modo sicuro per valutare un algoritmo di scheduling: lo si codifica, inserisce nel sistema operativo e si vede come funziona.

Capitolo 6

Sincronizzazione tra processi

Il modello astratto della sincronizzazione tra processi è quello del produttore-consumatore: il primo produce un messaggio e il secondo lo consuma in esecuzione concorrente. Essendo il buffer limitato ci sono dei vincoli: non si può aggiungere in buffer pieni e non si può consumare da buffer vuoti.

6.0.1 Buffer: modello software

In software il buffer è circolare di N posizioni, con in che indica la prima posizione libera e out la prima posizione occupata. Il buffer è vuoto quando $in = out$ e pieno quando $out = (in + 1)\%n$. Per semplicità si utilizza una variabile counter per indicare il numero di elementi nel buffer. Il produttore svolgerà la sua operazione incrementando il counter se e solo se il counter è minore di N e il consumatore lo decrementa solo se è maggiore di 0. Essendo le operazioni di incremento e decremento non atomiche (sono più istruzioni assembly) non è noto l'ordine di interleaving tra i processi e possono nascere delle inconsistenze in quanto produttore e consumatore possono modificare *counter* contemporaneamente. Si rende necessario proteggere l'accesso alla sezione critica.

6.0.2 Sezione critica

Si intende per sezione critica una porzione di codice in cui si accede ad una risorsa condivisa. La soluzione al suo problema deve rispettare:

- Mutua esclusione: unicamente un processo alla volta può accedere alla sezione critica.
- Progresso: solo i processi che stanno per entrare nella sezione critica possono decidere chi entra e la decisione non può essere rimandata all'infinito.
- Attesa limitata: deve esistere un massimo numero di volte per cui un processo può aspettare di seguito.

Un generico processo che accede ad una risorsa condivisa ha una struttura che contiene un'operazione ripetuta contenente prima delle operazioni sulla sezione critica le regole di ingresso alla sezione critica, le operazioni su di essa, una sezione di uscita dalla stessa e successivamente le operazioni sulla sezione non critica.

Soluzione

Assumendo una sincronizzazione in ambiente globale con condivisione di celle di memoria, le soluzioni software riguardano aggiunta di codice alle applicazioni senza nessun supporto hardware o del sistema operativo, mentre le soluzioni hardware riguardano codice alle applicazioni con supporto hardware.

6.1 Soluzioni software

6.1: PROCESS i

```
int turn; /*se turn = i allora entra il processo i*/
while(1){
    while(turn != i); /*sezione di entrata*/
    sezione critica
    turn = j; /*sezione di uscita*/
    sezione non critica
}
```

Questo algoritmo richiede stretta alternanza tra i processi: possono entrare nella zona critica unicamente alternativamente. Inoltre non rispetta il criterio del progresso in quanto non esiste alcuna nozione di stato.

6.1.1 Algoritmo 2

6.2: PROCESS i

```
boolean flag[2]; /*inizializzato a FALSE*/
while(1){
    flag[i] = true; /*vuole entrare in SC*/
    while(flag[j] == true); /*sezione di entrata*/
    sezione critica
    flag[i] = false; /*sezione di uscita*/
    sezione non critica
}
```

Risolve il problema dell'algoritmo 1 ma l'esecuzione in sequenza dell'istruzione *flag[] = true* da parte dei due processi porta a deadlock. Invertendo le istruzioni della sezione di entrata si viola la mutua esclusione in quanto entrambi i processi possono trovarsi nella sezione critica se eseguono in sequenza il *while* prima di impostare la *flag* a *true*

6.1.2 Algoritmo 3

6.3: PROCESS i

```
int turn;    /*di chi e' il turno?*/
boolean flag[2]; /*inizializzato a FALSE*/
while(1){
    flag[i] = TRUE; /*voglio entrare*/
    turn = j; /*tocca a te, se vuoi*/
```

```
    while(flag[j] == TRUE && turn == j);
    sezione critica
    flag[i] = FALSE;
    sezione non critica
}
```

È la soluzione corretta in quanto entra il primo processo che esegue $turn = j$ oppure $turn = i$

Dimostrazione

Mutua esclusione P_i entra nella sezione critica se e solo se $flag[j] = false$ o $turn = i$. Se P_i e P_j sono entrambi nella sezione critica allora $flag[i] = flag[j] = true$, ma P_i e P_j non possono aver superato entrambi il *while* perchè $turn$ vale i oppure j, pertanto solo uno dei due P è entrato.

Progresso e attesa limitata Se P_j non è pronto per entrare nella sezione critica allora $flag[j] = false$ e P_i può entrare. Se P_j ha impostato $flag[j] = true$ e si trova nel *while* allora $turn = i$ oppure $turn = j$. Se $turn = i$ P_i entra nella sezione critica, se $turn = j$ vi entra P_j. In ogni caso quando P_j esce dalla sezione critica imposta $flag[j] = false$ e quindi P_i può entrare nella sezione critica e P_i entra nella sezione critica al massimo dopo un'entrata di P_j.

6.1.3 Algoritmo del fornaio

Risolve il problema con N processi: ogni processo sceglie un numero ($choosing[i] = l$), il numero più basso viene servito per primo e per situazioni di numero identico si usa un confronto a due livelli ($numero,\ i$). L'algoritmo è corretto.

6.4: PROCESS i

```
boolean choosing[N]; /*il processo sceglie un numero*/
int number[N]; /*ultimo numero scelto*/
while(1){
    choosing[i] = TRUE;
    number[i] = Max(number[0], ..., number[N - 1]) + 1;
    choosing[i] = FALSE;
    for(j = 0; j < N; j++){
        while(choosing[j] == TRUE);
        while(number[j] != 0 && (number[j] < number[i] || (number[j] ==
            ↪ number[i]) && (j < i)));
    }
    sezione critica
    number[i] = 0;
    sezione non critica
}
```

6.2 Soluzioni hardware

Un modo hardware per risolvere il problema della sezione critica è di disabilitare gli interrupt mentre una variabile condivisa viene modificata. Da questo nasce un problema in quanto se il test per

l'accesso è lungo gli interrupt devono essere disabilitati per troppo tempo. Un'alternativa è che l'operazione per l'accesso alla risorsa deve occupare un unico ciclo di istruzione non interrompibile, con istruzioni atomiche di *Test-and-set* e *swap*

6.2.1 Test and Set

6.5: Test and Set

```
bool TestAndSet(boolean &var){ //il valore di var viene modificato
    boolean temp;    //Tutte le operazioni della funzione
    temp = var; //sono atomiche.
    var = true;
    return temp;
}
```

Il valore di ritorno è il valore di *var* a cui viene assegnato *true*. Si passa a questa funzione un lock che permette il controllo della sezione critica a un processo alla volta in quanto passa solo il primo processo che arriva e trova *lock* = *false*. Quando il processo termina le operazioni sulla sezione critica pone *lock* = *false*.

6.2.2 Swap

6.6: Swap

```
void Swap(boolean &a, boolean &b){
    boolean temp;    //Queste operazioni sono
    temp = a;    //svolte atomicamente
    a = b;
    b = temp;
}
```

Lo *swap* viene utilizzato sul *lock* e una variabile locale, che permette l'accesso solo quando quella locale diventa *false* in modo da eliminare la competizione sul *lock*. Si noti come *TestAndSet* e *Swap* non rispettano attesa limitata in quanto manca l'equivalente della variabile *turn* e sono pertanto necessarie variabili addizionali.

6.2.3 Test and Set con attesa limitata

6.7: PROCESS i

```
/*Variabili globali*/
boolean waiting[N];
boolean lock;
/*Programma locale*/
while(1){
    waiting[i] = true;
    key = true;
    while(waiting[i] && key){
        key = TestAndSet(lock);
```

```
    }
    wating[i] = false;
    sezione critica
    j = (i + 1) % N;
    while(j != i && !waiting[j]) //primo processo in attesa
        j = (j + 1) % N;
    if (j == i)
        lock = false;    //abilita se' stesso
    else
        waiting[j] = false; //abilita il processo j in attesa
    sezione non critica

}
```

6.2.4 Conclusioni

I vantaggi delle soluzioni hardware sono la scalabilità in quanto indipendenti dal numero di processi coinvolti e il fatto che l'estensione a N sezioni critiche è immediato. Offrono però maggiore complessità al programmatore rispetto alle soluzioni software e serve busy waiting con spreco di CPU.

6.3 Semafori

Si nota come le soluzioni precedenti non sono banali da aggiungere a programmi e sono basate su busy waiting. I semafori offrono una soluzione generica che funziona sempre. Si intende per semaforo una variabile intera S a cui si accede attraverso due primitive atomiche:

- Signal: *V(s)* che incrementa il valore di S di 1.
- Wait: *P(s)* che tenta di decrementare il valore di S di 1, se è 0 non si può ed è necessario attendere.

Esistono i semafori in versione binaria (con S 0 o 1) o generica (con S a valori interi maggiori o uguali di 0).

6.3.1 Semafori binari

Si noti come i semafori binari abbiano lo stesso potere espressivo di quelli a valori interi.

6.8: Implementazione concettuale

```
P(s):
    while(s == FALSE);//attesa
    s = FALSE;
V(s):
    s = TRUE;
```

Con busy waiting

6.9: Semaforo binario

```
/* s inizializzato a TRUE */
P(bool &s){
    key = FALSE;
    do{
        Swap(s, key);
    }while(key == FALSE);
}
```

6.10: Semaforo binario

```
V(bool &s){
    s = TRUE;
}
```

6.3.2 Semafori interi

Il problema dei semafori interi è garantirne l'atomicità.

6.11: Implementazione concettuale

```
P(s):
    while(s == 0); //attesa
    s--;
V(s):
    s++;
```

Con busy waiting

Sia *bool mutex* un semaforo binario inizializzato a *TRUE* e *bool delay* un semaforo binario inizializzato a *FALSE*. Sia in *P* che in *V* l'operazione *P(mutex)* protegge *S* da un'altra modifica. In *V* *V(delay)* permette la liberazione di un processo in attesa. In *P* se qualcuno occupa il semaforo si attende, altrimenti lo si passa.

6.12: Semaforo intero

```
P(int &s){
    P(mutex);
    s = s - 1;
    if(s < 0){
        V(mutex);
        P(delay);
    }
    else
        V(mutex);
}
```

6.13: Semaforo intero

```
V(int &s){
    P(mutex);
    s = s + 1;
    if(s <= 0){
        V(delay);
    }
    V(mutex);
}
```

Senza busy waiting

Nei semafori interi senza busy waiting si necessita di *bool mutex*, un semaforo binario inizializzato a *TRUE*. Inoltre il semaforo non è più un intero ma una *struct* contenente un valore intero (con semantica analoga a quello con busy wating) e una lista contenente i PCB (process control block). Inoltre l'operazione di *sleep()* mette il processo nello stato di waiting mentre *wakeup* lo mette nello stato ready. Si noti come si deve decidere l'ordine di *wakeup* dei processi.

6.14: Semaforo intero

```
typedef struct{
    int value;
    PCB *List;
} Sem;
```

6.15: Semaforo intero

```
P(Sem &s){
    P(mutex);
    s.value = s.value - 1;
    if(s.value < 0){
        V(mutex);
        append(processs i, s.List);
        sleep();
    }
    else{
        V(mutex);
    }
}
```

6.16: Semaforo intero

```
V(Sem &s){
    P(mutex);
    s.value = s.value + 1;
    if(s.value <= 0){
        V(mutex);
        PCB *p = remove(s.List);
        wakeup(p);
    }
    else{
        V(mutex);
    }
}
```

In questo caso il busy waiting viene eliminato dalla entry section ma rimane nella P e V del mutex che essendo veloce porta a poco spreco di CPU. Un alternativa è disabilitare gli interrupt durante P e V in cui istruzioni di processi diversi non possono essere eseguite in modo alternato.

6.3.3 Implementazione

Si noti come l'implementazione reale è diversa da quella concettuale: il valore di s può diventare minore di zero per i semafori interi in quanto conta quanti processi ci sono in attesa. La lista dei PCB può essere FIFO (strong semaphore) garantendo così attesa limitata. Nella modalità di implementazione con busy waiting (spinlock) la CPU controlla attivamente il verificarsi della condizione di accesso alla sezione critica. È una soluzione scalabile e veloce, CPU-intensive e adatta per attese brevi, come l'accesso a memoria. Nella modalità senza busy waiting con sleep (mutex, semaforo) il processo viene messo inattesa che si verifichi la condizione di accesso alla sezione critica. È più lento e adatto per attese lunghe come l'I/O.

6.3.4 Applicazioni

Un semaforo binario con valore iniziale 1 (mutex) viene utilizzato per la protezione di una sezione critica per n processi. Un semaforo binario con valore iniziale 0 viene utilizzato per la sincronizzazione del tipo di attesa di evento tra processi.

Sezione critica

Si dice mutex un semaforo binario di mutua esclusione. In questo caso n processi condividono la variabile s.

6.17: Mutex

```
/* valore iniziale di s = 1 (mutex) */
while(1){
    P(s);
    sezione critica
    V(s);
    sezione non critica
}
```

Semafori per attesa evento

Si consideri il caso di due processi $P1$ e $P2$ che devono sincronizzarsi rispetto all'esecuzione di due operazioni A e B in modo che $P2$ possa eseguire B solo dopo che $P1$ ha eseguito A. In questo caso si usa un semaforo binario s inizializzato a 0 tale che $P1$ svolga *V(s)* dopo A e $P2$ svolga B dopo *P(s)*.
Si consideri il caso di due processi $P1$ e $P2$ che devono sincronizzarsi rispetto all'esecuzione di un'operazione A in maniera alternata. Vengono pertanto utilizzati due semafori $s1$ inizializzato a 1 e $s2$ inizializzato a 0 in modo che $P1$ svolga *P(s1)* prima di A e *V(s2)* dopo, mentre $P2$ *P(s2)* prima e *V(s1)* dopo.

6.3.5 Limitazioni

I semafori possono generare deadlock, in cui il processo viene bloccato in attesa di un evento che solo lui può generare o starvation, in cui avviene un'attesa indefinita all'interno del semaforo.

6.4 Problemi classici dei semafori

6.4.1 Produttore consumatore

Il problema del produttore consumatore consiste di due processi con accesso sullo stesso buffer di dimensione limitata. Il produttore scrive nel buffer mentre il consumatore legge e lo svuota. IL problema richiede tre semafori:

- Mutex, un semaforo binario inizializzato a *TRUE* che garantisce la mutua esclusione per il buffer.
- Empty, un semaforo intero inizializzato a N che blocca P se il buffer è pieno.
- Full, un semaforo intero inizializzato a 0 che blocca C se il buffer è vuoto.

6.18: Produttore

```
while(1){
    produce item
    P(empty);
    P(mutex);
    deposit item
    V(mutex);
    V(full);
}
```

6.19: Consumatore

```
while(1){
    P(full);
    P(mutex);
    remove item
    V(mutex);
    V(empty);
    consume item
}
```

6.4.2 Dining philosophers

Si considerino N filosofi che passano la vita mangiando e pensando. Si trova 1 tavola con N bacchette e una ciotola di riso. Se un filosofo pensa non interagisce con gli altri. Se un filosofo ha fame prende 2 bacchette e inizia a mangiare. Il filosofo può prendere solo le bacchette che sono alla sua destra e alla sua sinistra e può prenderne solo una alla volta. Se non ci sono due bacchette libere non può mangiare. Quando un filosofo termina di mangiare rilascia le bacchette.

Prima soluzione

Dati condivisi

- Ci sono n semafori $s[N]$ inizializzati a 1.
- $P(s[j])$ vuol dire che si cerca di prendere la bacchetta j;
- $V(s[j])$ rilascia la bacchetta $j>$

La soluzione è incompleta C'è un possibile deadlock se tutti i filosofi tentano di prendere la bacchetta alla loro destra (sinistra) contemporaneamente.

6.20: Dining philosopher

```
do{
    P(s[i]);
    P(s[(i + 1) % N]);
    ...
    //mangia
    ...
    V(s[i]);
    V(s[(i + 1) % N]);
    ...
    //pensa
    ...
}while(1);
```

Deadlock Questa soluzione presenta un problema in quanto nasce un deadlock nel caso in cui ogni filosofo prende la bacchetta di destra e si trovano tutti ad aspettare che si liberi quella di sinistra che nessuno può rilasciare.

Possibili soluzioni parziali

- Si permette solo a quattro filosofi di mangiare contemporaneamente.
- Soluzione asimmetrica in cui i filosofi in posizione pari prendono la bacchetta sinistra seguita dalla destra mentre i filosofi in posizione dispari fanno il contrario.
- I filosofi si passano un token.

- Si permette ai filosofi di prendere la bacchetta solo se sono entrambe disponibili.

Soluzione corretta

In questa soluzione ogni filosofo si può trovare in tre stati: pensante (*THINKING*), affamato (*HUNGRY*) e mangiante (*EATING*). Le variabili condivise sono un semaforo mutex inizializzato a 1 N semafori $f[N] = 0$ e lo stato di ogni filosofo $stato[N]$.

6.21: Dining philosopher

```
void philosopher(int i){
    while(1){
        think();
        take_fork(i);
        eat();
        dropfork(i);
    }
}
```

6.22: Test

```
void test(int i){
    if(stato[i] == HUNGRY &&
       stato[i - 1] != EATING &&
       stato[i + 1] != EATING){
        stato[i] = EATING;
        V(f[i]);
    }
}
```

6.23: Drop fork

```
void drop_fork(int i){
    P(mutex);
    stato[i] = THINKING;
    //I vicini possono mangiare
    test((i - 1) % N);
    test((i + 1) % N);
    V(mutex);
}
```

6.24: Take fork

```
void take_fork(int i){
    P(mutex);
    stato[i] = HUNGRY;
    test(i);
    V(mutex);
    P(f[i]);
}
```

6.4.3 Sleepy barber

Un negozio ha una sala d'attesa con N sedie ed una stanza con la sedia del barbiere. In assenza di clienti il barbiere si addormenta. Quando entra un cliente se le sedie sono occupate il cliente se ne va, se il barbiere è occupato si siede e se è addormentato lo sveglia.

Soluzione

Dati condivisi

- Semaforo intero customer inizializzato a zero che sveglia il barbiere.
- Semaforo binario barbers inizializzato a zero che rappresenta lo stato del barbiere.
- Semaforo binario mutex inizializzato a 1 che protegge la sezione critica.
- *int waiting = 0* che conta i clienti in attesa.

6.25: Barber

```
while(1){
    P(customers);
    P(mutex);
    waiting--;
    V(barbers);
    V(mutex);
    cut hair
}
```

6.26: Consumer

```
P(mutex);
if(waiting < N){
    waiting++;
    //sveglia il barbiere
    V(customers);
    V(mutex);
    //pronto per il taglio
    P(barbers);
    get haircut
}
else{
    V(mutex);
}
```

6.4.4 Limitazioni dei semafori

L'utilizzo dei semafori presenta delle difficoltà in quanto risulta difficile scrivere programmi e la correttezza delle soluzioni è difficilmente dimostrabile. In alternativa vengono pertanto utilizzati specifici costrutti forniti da linguaggi di programmazione ad alto livello come monitor, classi synchronized in Java e CCR (conditional critical region) e altri.

6.5 Monitor

Sono costrutti per la condivisione sicura ed efficiente di dati da processi. Sono simili al concetto di classe.

6.27: Monitor

```
monitor xyz{
    //dichiarazione di variabili (stato del monitor)
    entry P1(...){
        ...
    }
    ...
    entry Pn(...){
        ...
    }
    {
        //codice di inizializzazione
    }
}
```

Le variabili del monitor sono visibili solo all'interno del monitor stesso e le procedure del monitor accedono solo alle variabili definite nel monitor. Un solo processo alla volta risulta attivo in un monitor in modo che il programmatore non debba codificare esplicitamente la mutua esclusione.

6.5.1 Operazioni del monitor

Per permettere ad un processo di attendere all'interno del monitor si rendono necessari opportune sincronizzazioni: le variabili condition dichiarate all'interno del monitor e accessibili solo tramite due primitive analoghe a quelle dei semafori: *wait()* come *P()* e *signal()* come *V()*. Il processo che invoca *x.wait()* è bloccato fino all'invocazione della corrispondente *x.signal()* da parte di un altro.

Wait

La *wait* blocca sempre il processo che la chiama e si deve pertanto prestare attenzione alla logica che regola tale chiamata.

Signal

La *signal* sveglia esattamente un processo e se ne trovano più in attesa lo scheduler decide quale processo può entrare. Se nessun processo si trova in attesa non c'è nessun effetto. Successivamente a questa operazione il processo che la invoca si può bloccare passando l'esecuzione al processo sbloccato o esce dal monito, caso in cui signal deve essere l'ultima istruzione di una procedura.

Buffer produttore consumatore

6.28: Produttore

```
Producer(){
    while(TRUE){
        //crea nuovo item
        make_item();
        //chiamata alla funzione
            ↪ enter
        ProducerConsumer.enter();
    }
}
```

6.29: Consumatore

```
Consumer(){
    while(TRUE){
        //chiamata alla funsione
        ProducerConsumer.remove();
        //consuma item
        consume_item();
    }
}
```

6.30: Monitor per buffer

```
monitor ProducerConsumer{
    condition full, empty;
    int count;
    entry enter(){
        if(count == N){
            //se il buffer e' pieno blocca
            full.wait();
        }
        //mette l'item nel buffer
        put_item();
        //incrementa count
        count = count + 1;
        if(count == 1){
            //se il buffer era vuoto
```

```
            //sveglia il consumatore
            empty.signal();
        }
    }
    entry remove(){
        if(count == 0){
            //se il buffer e' vuoto blocca
            empty.wait();
        }
        //rimuove item dal buffer
        remove_item();
        //decrementa count
        count = count - 1;
        if(count == N - 1){
            //se il buffer era pieno
            //sveglia il produttore
            full.signal();
        }
    }
    //inizializzazione di count
    count = 0;
    end monitor;
}
```

Semaforo binario nel monitor

6.31: Semaforo binario nel monitor

```
monitor BinSem{
    boolean busy; //inizializzato a FALSE
    condition idle;
    entry void P(){
        if(busy)
            idle.wait();
        busy = TRUE;
    }
    entry void V(){
        busy = FALSE;
        idle.signal();
    }
    busy = FALSE //inizializzazione
}
```

6.5.2 Limitazioni

Pur essendo meno proni ad errori rispetto ai semafori i monitor sono forniti da pochi linguaggi e richiedono sempre presenza di memoria condivisa.

6.6 Sincronizzazione in Java

La sezione critica viene indicata dalla keyword synchronized e i metodi synchronized possono essere eseguiti da un solo thread alla volta e vengono realizzati mantenendo un singolo lock detto monitor per oggetto. I metodi static synchronized presentano un solo lock per classe, mentre per i blocchi synchronized è possibile mettere un lock su qualsiasi oggetto per definire una sezione critica. Altri metodi di sincronizzazione sono *wait() notify()* e *notifyAll()*, che vengono ereditati da tutti gli oggetti.

6.6.1 Buffer produttore consumatore

6.32: Bounded buffer

```
public class BoundedBuffer{
    Object[] buffer;
    int nexin;
    int nextout;
    int size;
    int count;
    //costrutture
    public BoundedBuffer(int n){
        size = n;
        buffer = new Object[size];
        nextin = 0;
        nextout = 0;
        count = 0;
    }
}
```

6.33: Deposit

```
public synchronized deposit(Object
    ↪ x){
    while(count == size) wait();
    buffer[nextin] = x;
    nextin = (nextin + 1) % N;
    count = count + 1;
    notifyAll();
}
```

6.34: Remove

```
public synchronized Object remove()
    ↪ {
    Object x;
    while(count == 0) wait;
    x = buffer[nexout];
    nextout = (nextout + 1) % N;
    count = count - 1;
    notifyAll();
    return x;
}
```

6.7 Conclusioni

IL problema della soluzione critica è un'astrazione della concorrenza tra processi. Esistono soluzioni con diversi compromessi tra complessità e difficoltà di utilizzo. Si deve prestare attenzione alla gestione del blocco critico di un insieme di processi (deadlock) dipendente dalla sequenza temporale degli accessi.

6.8 Problema degli scrittori e dei lettori

C'è un area di dati condivisa ed esistono dei lettori che possono solo leggere i dati e scrittori che possono solo scriverli. Più lettori possono leggere il file contemporaneamente, solo uno scrittore

alla volta può scrivere nel file, se uno scrittore sta scrivendo nel file nessun lettore può leggerlo, gli scrittori non possono leggere e i lettori non possono essere anche scrittori.

6.35: Scrittore

```
Sem scrittura = 1;

scrittore{
    while(1){
        P(scrittura);
        < modifica i dati >
        V(scrittura);
    }
}
```

6.36: Lettore

```
int num_lettori = 0;
semaphore mutex = 1;

Lettore{
    while(1){
        ...
        P(mutex); //protegge num_lettori e li accoda
        num_lettori++;
        if(num_lettori == 1) //primo lettore
            P(scrittura); //acquisisce la mutua esclusione in scrittura
        V(mutex);
        < legge i  dati >
        P(mutex); //protegge num_lettori
        num_lettori--;
        if(num_lettori == 0) //ultimo lettore
            V(scrittura); //rilascia la mutua esclusione in scrittura
        V(mutex);
        ...
    }
}
```

Capitolo 7

Deadlock

In una sequenza di utilizzo dei processi che utilizzano risorse si trovano tre fasi:

- Richiesta: se non può essere immediatamente soddisfatta il processo deve attendere.
- Utilizzo.
- Rilascio.

Un insieme di processi si definisce in deadlock quando ogni processo è in attesa di un evento che può essere causato solo da un processo dello stesso insieme. Un deadlock si può risolvere attraverso preemption e rollback, con pericolo di starvation.

7.0.1 Condizioni necessarie

Affinchè si verifichi un deadlock devono essere vere contemporaneamente:

- Mutua esclusione: almeno una risorsa deve essere non condivisibile.
- Hold and Wait: deve esistere un processo che detiene una risorsa e che attende di acquisirne un'altra detenuta da un altro.
- No preemption: le risorse non possono essere rilasciate se non volontariamente dal processo che le usa.
- Attesa circolare: deve esistere un insieme di processi che attendono ciclicamente il liberarsi di una risorsa.

7.1 Modello astratto: resource allocation graph (RAG)

Sia un RAG un grafo $G(V, E)$ tale che i nodi V rappresentati attraverso cerchi sono i processi mentre i nodi rappresentati come rettangoli sono le risorse. Nei rettangoli si trovano tanti cerchi quante sono le istanze della corrispondente risorsa. Gli archi E da processi a risorse indicano un processo che richiede una risorsa, mentre da risorse a processi indicano che il processo detiene la risorsa. Se il RAG non contiene cicli non ci sono deadlock, mentre se ne contiene se si ha una sola istanza per risorsa allora si ha deadlock, mentre se ci sono più istanze dipende dallo schema di allocazione.

7.2 Gestione dei deadlock

7.2.1 Prevenzione statica

Nella prevenzione statica si tenta di evitare che si possa verificare una delle quattro condizioni.

Mutua esclusione

Si noti come la mutua esclusione è irrinunciabile per certi tipi di risorsa e pertanto non è risolvibile.

Hold and Wait

Una soluzione consiste nel fatto che un processo alloca all'inizio tutte le risorse che deve utilizzare o può ottenerne una solo se non ne ha altre. Con queste soluzioni si ottiene un basso utilizzo delle risorse con possibilità di starvation in caso di richiesta di molte risorse molto popolari. Si deve inoltre conoscere il numero di risorse richieste.

No preemption

Una soluzione consiste nel fatto che un processo che richiede una risorsa non disponibile deve cedere tutte le altre risorse che detiene. In alternativa può cedere risorse che detiene su richiesta di un altro processo. Si noti come è fattibile solo per risorse il cui stato può essere ristabilito facilmente.

Attesa circolare

Si assegna una prioriità, un ordinamento globale ad ogni risorsa in modo che un processo può richiedere risorse solo in ordine crescente di priorità, pertanto l'attesa circolare diventa impossibile. La priorità deve seguire il normale ordine di richiesta.

7.2.2 Prevenzione dinamica (avoidance)

Questo metodo si basa sull'allocazione delle risorse e non viene mai utilizzata poichè richiede una conoscenza troppo approfondita delle richieste di risorse. Si noti come le tecniche di prevenzione statica possono portare a un basso utilizzo delle risorse perchè mettono vincoli sul modo in cui i processi possono accedere alle risorse. L'obiettivo è la prevenzione in base alla richieste: un analisi dinamica del grafo delle risorse per evitare situazioni cicliche. Richiede come requisito la conoscenza del caso peggiore: il massimo numero di istanze di una risorsa richieste per processo.

Stato Safe

Lo stato di assegnazione delle risorse viene calcolato come il numero di istanze disponibili e allocate e le richieste massime dei processi. Il sistema si trova in uno stato sicuro se esiste una sequenza safe ovvero se usando le risorse disponibili può allocare risorse ad ogni processo in qualche ordine in modo che ciascuno di essi possa terminare la sua esecuzione.

Sequenza Safe

Una sequenza di processi $(P_1, \ldots, P_N)$ è safe se per ogni P_i le risorse che P_i può richiedere possono essere esaudite usando le risorse disponibili e quelle detenute da P_j, $j < i$, ovvero attendendo che P_j termini. Se non esiste tale sequenza si trova in uno stato unsafe. Si noti come non tutti gli stati unsafe sono deadlock, ma da stato unsafe si può arrivare ad un deadlock.

Metodo della prevenzione dinamica

Si usano algoritmi che lasciano il sistema sempre in uno stato safe. All'inizio il sistema è in uno stato safe. Ogni volta che P richiede R, R viene assegnata a P se e solo se si rimane in uno stato safe. L'utilizzo delle risorse sarà sempre minore rispetto al caso in cui non si usano tecniche di prevenzione dinamica.

Algoritmo con RAG

Questo algoritmo funziona solo se c'è una sola istanza per risorsa. Il RAG viene esteso con archi di rivendicazione: $P_i \rightarrow R_j$ se P_i può richiedere R_j in futuro. Questi archi vengono indicati con una freccia tratteggiata. All'inizio ogni processo deve rendere noto quali risorse vorrebbe usare durante la sua esecuzione. Una richiesta viene soddisfatta se e solo se l'allocazione della risorsa non crea un ciclo nel RAG. Serve un algoritmo per la rilevazione dei calci di complessità ($O(n^2)$, dove n è il numero di processi)

Algoritmo del banchiere

Pure essendo meno efficiente dell'algoritmo con RAG funziona qualunque sia il numero di istanze. In questo algoritmo il banchiere non deve mai distribuire tutto il denaro che ha in cassa in quanto altrimenti non potrebbe più soddisfare successivi clienti. Ogni processo dichiara la sua massima richiesta e ogni volta che un processo richiede una risorsa si determina se soddisfarla lascia uno stato safe. Tale algoritmo è costituito da un algoritmo di allocazione e uno di verifica dello stato.

7.1: Strutture dati per n processi e m risorse

```
int available[m];    //numero di istanze di Ri disponibili
int max[n][m];       //matrice delle richieste di risorse
int alloc[n][m];     //matrice di allocazione corrente
int need[n][m];      //matrice bisogno rimanente
            //need[i][j] = max[i][j] - alloc[i][j]
```

Algoritmo di allocazione

7.2: Allocazione per P_i

```
void request(int req_vec[]){ //richieste del processo Pi
    if(req_vec[] > need[i][])
        error(); //superato il massimo preventivato
    if(req_vec[] > available[])
        wait(); //attendo che si liberino risorse
    //simulo l'assegnazione
    available[] = available[] - req_vec[];
    alloc[i][] = alloc[i][] + req_vec[];
    need[i][] = need[i][] - req_vec[];
    if(!state_safe()){ //se non e' safe ripristino il vecchio stato
        //rollback
        available[] = available[] + req_vec[];
        alloc[i][] = alloc[i][] - req_vec[];
        need[i][] = need[i][] + req_vec[];
        wait();
```

```
    }
}
```

7.3: Verifica dello stato

Algoritmo di verifica dello stato

```
bool state_safe(){
    int work[m] = available[];
    bool finish[n] = {FALSE};
    int i;
    while(finish != {TRUE}){
        /* cerca Pi che non abbia terminato e
         * possa completare con le risorse
         * disponibili in work */
        for(i = 0; (i < n) && (finish[i] || (need[i][] > work[])); i++);
        if(i == n)
            return FALSE; //non esiste e' unsafe
        else{
            work[] = word[] + alloc[i][];
            finish[i] = TRUE;
        }
    }
    return TRUE;
}
```

Si noti come questo algoritmo abbia complessità $O(m \cdot n^2)$.

7.2.3 Rilevamento (detection) e ripristino (recovery)

In questo metodo si permette che si verifichino deadlock e prevede metodi per riportare il sistema al funzionamento normale. Nasce in quanto prevenzione statica e dinamica sono conservativi e riducono eccessivamente l'utilizzo delle risorse. Ci sono due approcci alternativi: rilevamento del ripristino tramite il grafo di attesa calcolato attraverso il RAG e l'algoritmo di rilevazione.

Attraverso il RAG

Funziona solo con una risorsa per tipo e consiste nell'analizzare periodicamente il RAG< verificare se esistono deadlock ed iniziare il ripristino. Non è necessaria una conoscenza anticipata delle richieste e permette un loro utilizzo migliore ma presenta il costo del recovery.

Algoritmo di rilevamento

L'algoritmo di rilevamento si basa sull'esplorazione di ogni possibile sequenza di allocazione per i processi che non hanno ancora terminato. Se la sequenza va a buon fine (è safe) non avviene deadlock.

7.4: Strutture dati per n processi e m risorse

```
int available[m];    //numero di istanze di Ri disponibili
int alloc[n][m];     //matrice di allocazione corrente
```

```
int req_vec[n][m];   //matrice della richiesta
```

7.5: Algoritmo di rilevamento

```
int work[m] = available[m];
bool finish[] = {FALSE}, found - TRUE;
while(found){
    found = FALSE;
    for(int i = 0; i < n && !found; i++){
        //cerca un Pi con richiesta soddisfacibile
        if(!finish[i] && req_vec[i][] <= work[]){
            //assume ottimisticamente che Pi esegua
            //fino al termine e che quindi restituisca
            //le risorse (catturato alla prossima
            //esecuzione)
            work[] = work[] + alloc[i][];
            finish[i] = TRUE;
            found = TRUE;
        }
    }
}//se finish[i] = FALSE per un qualsiasi i, Pi e' in deadlock
```

Ripristino

L'algoritmo di rilevamento può essere chiamato dopo ogni richiesta, ogni N secondi o quando l'utilizzo della CPU scende sotto una soglia T e può uccidere i processi coinvolti o fare prelazione delle risorse dai processi bloccati nel deadlock.

Uccisione dei processi Questo approccio è costoso in quanto tutti i processi devono ripartire e perdono il lavoro svolto. Uccidere selettivamente fino alla scomparsa del deadlock è costoso in quanto invoca l'algoritmo di rilevazione dopo ogni uccisione e si deve decidere accuratamente l'ordine.

Prelazione delle risorse Il problema è che il processo che subisce la prelazione non può continuare normalmente e pertanto si deve fare rollback in uno stato safe da cui si riparte (eventualmente ripartendo da zero). È possibile starvation se tolgo le risorse sempre agli stessi processo. Si deve pertanto considerare il numero di rollback nei fattori di costo.

7.2.4 Algoritmo dello struzzo

In questo metodo non si fa nulla in quanto i deadlock sono rari e gestirli costa troppo.

7.3 Conclusioni

Ognuno degli approcci ha vantaggi e svantaggi, nessuno superiore agli altri. Si può avere una soluzione combinata che partiziona le risorse in classi usando una strategia di ordinamento con l'algoritmo più appropriato per la classe.

7.3.1 Partizionamento in classi

1. Risorse interne usate dal sistema.
2. Memoria.
3. Risorse di processo.
4. Spazio di swap.

7.3.2 Algoritmi specifici

1. Prevenzione tramite ordinamento delle risorse.
2. Prevenzione tramite prelazione: un job può essere swappato.
3. Prevenzione dinamica: richiesta massima di risorse nota a priori.
4. Prevenzione tramite preallocazione: richiesta massima nota a priori.

È sempre possibile ignorare i deadlock in quanto sono eventi rari, la prevenzione è costosa come il recovery e gli algoritmi sono spesso sbagliati.