Sistemi operativi

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

28 maggio 2020

Indice

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.

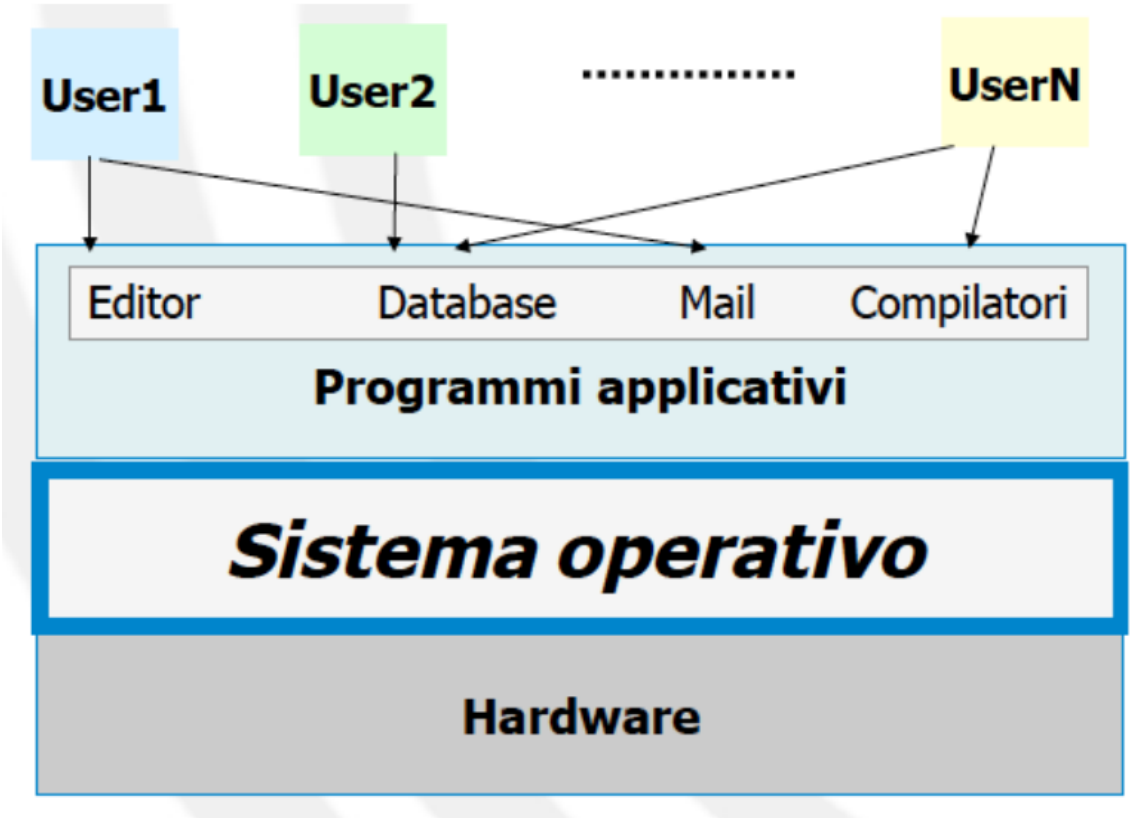

Figura 1.1: Stack del sistema operativo

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

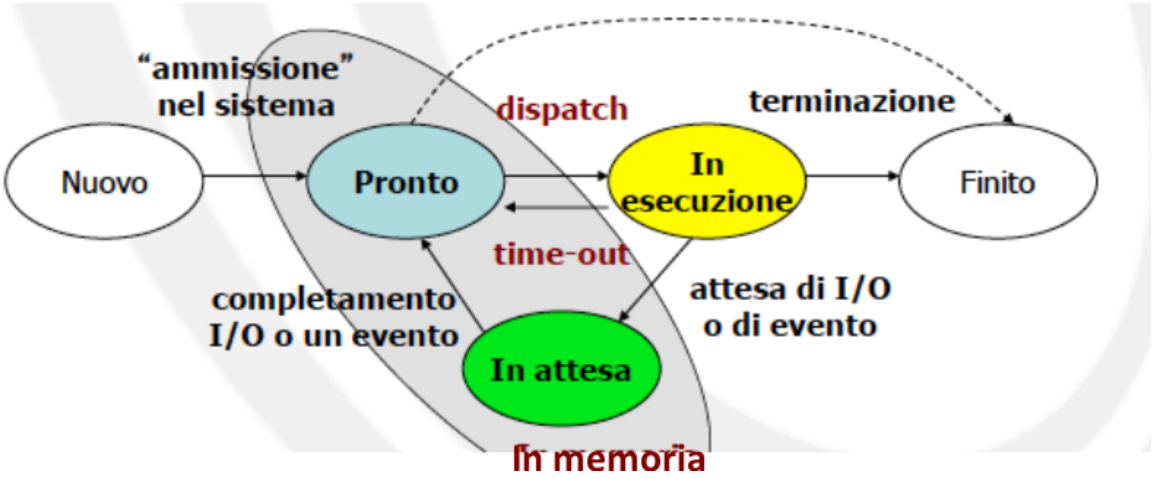

Figura 4.1: Stack del sistema operativo

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]
```

7.2: Allocazione per P_i

Algoritmo di allocazione

```
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.

Capitolo 8

Gestione della memoria

La condivisione della memoria da parte di più processi è essenziale per l'efficienza del sistema. Offre problematiche negli ambiti di allocazione della memoria ai singoli job, per la protezione e condivisione dello spazio di indirizzamento e per la gestione della memoria virtuale (swap). Nei sistemi moderni la gestione della memoria è inseparabile dal concetto di memoria virtuale. Ogni processo deve essere portato in memoria e trasformato in processo per essere eseguito. La CPU preleva le istruzioni da eseguire dalla memoria in base al valore del program counter. L'istruzione viene codificata e può prevedere il prelievo di operandi dalla memoria. Al termine dell'istruzione il risultato può essere scritto in memoria. Quando il processo termina la sua memoria viene rilasciata.

8.1 Da programma a processo

La trasformazione da programma a processo avviene attraverso varie fasi precedenti all'esecuzione: in ogni fase si ha una diversa semantica degli indirizzi (spazio logico e spazio fisico). Gli indirizzi del programma sorgente sono simbolici e trasformati in indirizzi fisici attraverso il compilatore che associa agli indirizzi simbolici indirizzi rilocabili e il linker o il loader associano agli indirizzi rilocabili indirizzi assoluti. Si noti come gli indirizzi hanno diverse rappresentazioni nelle varie fasi di costruzione di un programma. Il collegamento tra indirizzi simbolici e fisici viene detto binding.

8.1.1 Binding

Il binding di dati e istruzioni a indirizzi di memoria può avvenire in tre momenti distinti:

1. Tempo di compilazione: statico, se è noto a priori in quale parte della memoria risiederà il processo è possibile generare codice assoluto, ma se la locazione di partenza cambia sarà necessario ricompilare.

2. Tempo di caricamento: statico, si rende necessario generare codice rilocabile con indirizzi relativi all'inizio del programma. Se cambia indirizzo di riferimento si deve ricaricare.

3. Tempo di esecuzione: dinamico, il binding viene posticipato se il processo può essere spostato durante l'esecuzione in posizioni diverse della memoria. Viene richiesto supporto hardware affinchè l'operazione possa essere fatta efficientemente.

8.1.2 Collegamento

Il collegamento può essere statico in cui tutti i riferimenti sono definiti prima dell'esecuzione e l'immagine del processo contiene una copia delle librerie usate o dinamico in cui il link viene posticipato al tempo di esecuzione e il codice del programma non contiene il codice delle librerie ma solo un riferimento (stub) per poterle recuperare.

8.1.3 Caricamento

Il caricamento può essere statico in cui tutto il codice viene caricato in memoria a tempo dell'esecuzione o dinamico in cui il caricamento dei moduli viene posticipato in corrispondenza del primo utilizzo (risulta utile per codice con molti casi speciali).

8.1.4 Spazi di indirizzamento

Lo spazio di indirizzamento logico è legato a uno spazio di indirizzamento fisico: l'indirizzo logico (o virtuale) è generato dalla CPU mentre quello fisico viene considerato dalla memoria. Nel binding a compile o load-time l'indirizzo fisico e logico coincidono mentre in quello a run-time sono generalmente diversi.

Memory management unit (MMU)

La memory management unit è un dispositivo hardware che mappa indirizzi virtuali in indirizzi fisici: il valore del registro di rilocazione è aggiunto ad ogni indirizzo generato da un processo e inviato alla memoria.

8.1.5 Considerazioni

In un sistema multiprogrammato non è possibile conoscere in anticipo dove un processo può essere posizionato in memoria e lo swap impedisce di poter utilizzare indirizzi rilocati in modo statico. Pertanto non è possibile il binding a tempo di compilazione o di caricamento. Si deve far affidamento alla rilocazione dinamica usata per sistemi complessi come la gestione della memoria nel sistema operativo o la rilocazione statica, possibile in sistemi progettati per applicazioni specifiche e con limitata gestione della memoria nel sistema operativo.

8.2 Schemi di gestione della memoria

I metodi principali di gestione della memoria sono:

- Allocazione contigua.
- Paginazione.
- Segmentazione.
- Segmentazione doppia.

Si noti come tutti prevedono che il programma sia interamente caricato in memoria, mentre soluzioni realistiche usano memoria virtuale.

8.2.1 Allocazione contigua

Nell'allocazione contigua i processi sono allocati in memoria in posizioni contigue all'interno di una partizione. Le partizioni possono essere fisse o variabili. La memoria è un insieme di partizioni di dimensioni predefinite e diverse. Nascono problematiche per l'assegnazione di memoria ai job e per il supporto della rilocazione dinamica. L'assegnazione della memoria viene fatta dallo scheduling a lungo termine attraverso o una coda di partizione o una coda singola.

Assegnazione della memoria

Se si trova una coda per partizione il processo viene assegnato alla partizione più piccola in grado di contenerlo. Si noti come è poco flessibile in quanto possono esserci partizioni vuote e job nelle altre code. Nel caso di una coda unica gestita con politica first come first served l'implementazione è facile ma vi è un basso utilizzo della memoria. La scansione della coda può avvenire attraverso best-fit-only in cui la scelta del job avviene scegliendo quello con le dimensioni più simili alla partizione o attraverso first-available-fit in cui viene scelto il primo job che può stare nella partizione.

Supporto per la rilocazione

La MMU consiste di registri di rilocazione per proteggere lo spazio dei vari processi, attivamente e passivamente. Contiene il valore dell'indirizzo più basso (registro base o di rilocazione) e il limite superiore dello spazio logico (registro limite). Ogni indirizzo logico deve essere minore del limite.

Considerazioni

Si noti come questo è un approccio relativamente semplice ma il grado di multiprogrammazione è limitato dal numero di partizioni e nasce frammentazione che porta a spreco di memoria. La frammentazione può essere interna: nella partizione se la dimensione della partizione è più grande della dimensione del job o esterna se vi sono partizioni non utilizzate che non soddisfano le esigenze dei processi in attesa.

Tecnica delle partizioni variabili

In questa tecnica lo spazio utente viene diviso in partizioni di dimensioni variabili identiche alla dimensione dei processi per eliminare la frammentazione interna.

Assegnazione della memoria La memoria viene vista come un insieme di buche e il sistema operativo mantiene informazioni su partizioni allocate e buche. Quando arriva un processo gli viene allocata memoria usando la buca che lo può contenere. Per soddisfare la richiesta di n celle di memoria data una lista di buche libere si possono utilizzare le strategie:

- First-fit: alloca la prima buca grande a sufficienza.
- Best-fit: alloca la più piccola buca grande a sufficienza. Richiede la scansione della lista e fornisce il minimo spreco.
- Worst-fit: alloca la buca più grande. Richiede la scansione della lista e lascia la buca di dimensioni più grandi.

Si noti come first fit è tipicamente la migliore.

Supporto per la rilocazione Come per le partizioni fisse i registri di rilocazione vengono usati per proteggere lo spazio dei vari processi attivamente e passivamente.

Considerazioni Non fornisce frammentazione interna per costruzione ma la frammentazione esterna causa spreco di memoria in quanto nonostante esista lo spazio disponibile non è contiguo. Con first fit dati N blocchi allocati $0.5 \cdot N$ blocchi vanno persi. La frammentazione si può ridurre attraverso compattazione: il contenuto della memoria viene spostato in modo da rendere contigue tutte le partizioni. È possibile solo se la rilocazione è dinamica e la modifica del registro base.

Tecnica del buddy system

In questa tecnica si deve fare il compromesso tra le partizioni fisse e variabili. La memoria viene vista come una serie di liste di blocchi di dimensione 2^k con $L < k < U$, dove 2^L è il più piccolo blocco allocato e 2^U è il più grande blocco allocato. La memoria è disponibile sotto forma di blocchi di dimension 2^k. All'inizio tutta la memoria è disponibile: a lista di blocchi di dimensione 2^U contiene un solo blocco che rappresenta tutta la memoria mentre le altre sono vuote. Quando arriva una richiesta di dimensione s si cerca un blocco libero con dimensione adatta purchè sia pari a una potenza del 2. Se $2^{U-1} < s < 2^U$ l'intero blocco di dimensione 2^U viene allocato. Altrimenti il blocco 2^U viene diviso in due blocchi di dimensione 2^{U-1}. Questa operazione viene ripetuta fino a che è viene allocato il processo o si arriva al blocco di dimensione 2^L. Quando un processo rilascia la memoria il suo blocco torna a far parte della lista dei blocchi di dimensione corrispondente. Se si formano 2 blocchi adiacenti di dimensione 2^k è possibile compattarli ottenendo un unico blocco libero di dimensione 2^{k+1}. Il vantaggio è che la compattazione richiede solo di scorrere la lista dei blocchi di dimensione 2^k ed è veloce, ma nasce la frammentazione interna dovuta solo ai blocchi di dimensione 2^L.

8.2.2 Paginazione

La paginazione nasce per eliminare la frammentazione esterna. Si permette che lo spazio di indirizzamento fisico di un processo sia non-contiguo. Si alloca memoria fisica dove essa è disponibile. La memoria fisica viene divisa in blocchi di dimensione fissa detti frame e la memoria logica viene divisa in blocchi della stessa dimensione detti pagine. Per eseguire un programma avente dimensione n pagine bisogna trovare n frame liberi. Si utilizza una tabella delle pagine (page table) per mantenere traccia di quale frame corrisponde a quale pagina. Esiste una tabella delle pagine per ogni processo e viene usata per tradurre un indirizzo logico in un indirizzo fisico. La frammentazione interna nasce solo nell'ultima pagina.

Traduzione degli indirizzi

L'indirizzo generato dalla CPU viene diviso in due parti:

- Numero di pagina (p): usato come indice nella tabella delle pagine che contiene l'indirizzo di base di ogni frame.
- Offset (d): combinato con l'indirizzo base definisce l'indirizzo fisico che viene inviato alla memoria. Se la dimensione della memoria è 2^m e quella di una pagina è 2^n parole per byte i primi $m - n$ bit sono il numero della pagina e i successivi n il numero di offset.

Implementazione della tabella delle pagine L'efficienza è fondamentale e pertanto la tabella si può implementare attraverso registro o implementazione in memoria (tabella multilivello o invertita).

Implementazione tramite registri Le entry (righe) della tabella delle pagine sono mantenute nei registri. La soluzione è efficiente ma fattibile solo se il numero di entry è limitato e allunga i tempi del context switch in quanto richiede il salvataggio dei registri.

Implementazione in memoria La tabella risiede in memoria e vengono utilizzati due registri: il page-table base register (PTBR) che punta alla tabella delle pagine e l'opzionale page-table length register (PTLR) che contiene la dimensione della tabella delle pagine. Il context switch è più breve in quanto richiede la modifica solo del PTBR (PTLR), ma ad ogni accesso a dati o istruzioni richiede due accessi in memoria per la tabella delle pagine e il risultante dato/istruzione. Il problema del doppio accesso può essere risolto tramite una cache molto veloce detta translation look-aside buffers (TLB). Funzionano confrontando l'elemento fornico con il cambio chiave di tutte le entry contemporaneamente e nella tabella delle pagine la chiave dà il numero di pagina e il valore il numero di frame. Essendo il TLB molto costoso viene memorizzato solo un piccolo sottoinsieme delle entry della tabella delle pagine. Ad ogni context switch il TLB viene ripulito per evitare il mapping di indirizzi errati. Durante un accesso alla memoria se la pagina cercata è nel TLB questo restituisce il numero di frame con un singolo accesso (minore del 10% del tempo richiesto in assenza di TLB), altrimenti è necessario accedere alla tabella delle pagine in memoria. L'hit ratio α è la percentuale delle volte in cui una pagina si trova nel TLB. Si rende necessario definire il concetto di tempo di accesso effettivo:

$$EAT = (T_{MEM} + T_{TLB}) \cdot \alpha + (2 \cdot T_{MEM} + T_{TLB}) \cdot (1 - \alpha)$$

Dove T_{TLB} è il tempo di accesso a TLB e T_{MEM} è il tempo di accesso a memoria.

Protezione

La protezione viene associata associando bit di protezione ad ogni livello come il bit di validità (valid-invalid bit) per ogni entry della tabella delle pagine che risulta utile per la memoria virtuale:

- Valid: la pagina associata è nello spazio di indirizzamento logico del processo.
- Invalid: la pagina associata non è nello spazio di indirizzamento logico del processo.

Bit di accesso:

- Per marcare una pagina modificabile o meno (read-only).
- Per marcare una pagina eseguibile o meno.

Pagine condivise

Per il codice condiviso si trova un'unica copia fisica ma più copie logiche (una per processo), il codice read-only rientrante che non cambia mai durante l'esecuzione può essere condiviso tra processo. I dati in generale saranno diversi da processo a processo con più copie fisiche e logiche.

Spazio di indirizzamento

Nelle architetture moderne lo spazio di indirizzamento virtuale è molto maggiore dello spazio fisico. Sono necessari meccanismi per gestire il problema della dimensione della tabella delle pagine attraverso la paginazione della tabella delle pagine (tabella multilivello) o la tabella delle pagine invertita.

Pagine multilivello Questo metodo è equivalente a paginare la tabella delle pagine: solo alcune parti della tabella sono memorizzate in memoria, altre si trovano sul disco. Sono possibili versioni da 2 a 4 livelli. Ogni livello è memorizzato come una tabella separata in memoria, la conversione dell'indirizzo logico in quello fisico può richiedere anche 4 accessi a memoria e il TLB mantiene le prestazioni a livelli ragionevoli.

Tabella delle pagine invertita Si trova una tabella unica nel sistema con un'entry per ogni frame contenente la coppia <process-id, page-number> dove il primo è l'identificativo del processo che possiede la pagina e il secondo è l'indirizzo logico della pagina contenuta nel frame corrispondente a quella entry. Ogni indirizzo logico generato dalla CPU è una tripla: <process-id, page-number, offset>. È necessario cercare il valore desiderato e pertanto un aumento del tempo necessario per cercare un riferimento ad una pagina. Per questioni di ricerca si deve ridurre il tempo di ricerca a $O(1)$ attraverso una tabella hash e si crea la necessità di un meccanismo per gestire le collisioni quando diversi indirizzi virtuali corrispondono allo stesso frame.

8.2.3 Segmentazione

Nella segmentazione la memoria viene gestita nello stesso in cui l'utente la percepisce: un programma è una collezione di segmenti, unità logiche come main, procedure, funzioni, variabili, stack, tabella dei simboli e vettori. L'indirizzo logico è dato dalla coppia <numero di segmento, offset> e la tabella dei segmenti mappa gli indirizzi logici bidimensionali in indirizzi risifici unidimensionali. Ogni entry contiene una base, l'indirizzo fisico di partenza del segmento in memoria e il limite, la lunghezza del segmento.

Tabella dei segmenti

La tabella dei segmenti è simile alla tabella delle pagine: viene salvata in memoria e il processo possiede il segment-table base register (STBR) che punta alla locazione della tabella dei segmenti e il segment-table length register (STLR) che indica il numero di segmenti utilizzati da un programma. Un indirizzo logico <s, d> è valido se $s < STLR$. $STBR + s$ è l'indirizzo dell'elemento della tabella dei segmenti da recuperare. Il TLB viene usato per memorizzare le entry maggiormente usate come per la paginazione.

Protezione

La segmentazione supporta naturalmente la protezione e la condivisione di porzioni di codice in quanto il segmento è un'entità semantica ben definita. Ad ogni segmento sono associati un bit di modalità (read/write/execute) e il bit di validità (0 se il segmento non è legale).

Condivisione

A livello di segmento se si vuole condividere qualcosa basta inserirlo in un segmento. Si rende pertanto possibile condividere parti di un programma come funzioni di libreria.

Frammentazione

Il sistema operativo deve allocare spazio in memoria per tutti i segmenti di un programma che hanno lunghezza variabile. Periste il problema di allocazione che viene risolto con first o best fit. Esiste possibilità di frammentazione esterna per segmenti di dimensione significativa.

Confronto con paginazione

Paginazione:

- Vantaggi
  - Non esiste frammentazione (minima interna).
  - L'allocazione dei frame non richiede algoritmi specifici.
- Svantaggi
  - Separazione tra vista utente e vista fisica della memoria.

Segmentazione:

- Vantaggi
  - Consistenza tra vista utente e vista fisica della memoria.
  - Associazione di protezione e condivisione segmenti.
- Svantaggi
  - Richiesta di allocazione dinamica dei segmenti.
  - Potenziale frammentazione esterna.

8.2.4 Segmentazione paginata

È possibile combinare paginazione e segmentazione per migliorarli: i segmenti vengono paginati: ogni segmento viene diviso in pagine e possiede la propria tabella delle pagine. La tabella dei segmenti contiene l'indirizzo base delle tabelle delle pagine per ogni segmento. Elimina il problema dell'allocazione dei segmenti e della frammentazione esterna.

Capitolo 9

Memoria virtuale

Si noti come negli schemi precedenti per la gestione della memoria l'intero programma deve essere caricato in memoria per essere eseguito, cosa che può non essere necessaria e solo una parte del programma può essere in memoria: lo spazio degli indirizzi logici può essere pertanto molto più grande di quello degli indirizzi fisici e più processi possono essere mantenuti in memoria. Si deve pertanto avere la possibilità di swappare le pagine da e verso la memoria e non l'intero processo. La memoria virtuale permette la separazione della memoria logica dalla memoria fisica in quanto è data dalla memoria fisica e dal disco. Può essere implementata attraverso paginazione su domanda (demand paging) o segmentazione su domanda (demand segmentation).

9.1 Paginazione su domanda

In questo metodo una pagina viene caricata in memoria solo quando necessario in modo da ridurre le richieste di I/O quando è necessario lo swapping ottenendo una risposta più rapida e un minore consumo di memoria per permettere a più processi di accedere alla memoria. È fondamentale sapere se una pagina è presente o no in memoria.

9.1.1 Valid/invalid bit e page fault

Ad ogni entry della page table è associato un bit (1, valid, in memoria, 0, invalid, non in memoria). Inizialmente sono tutti 0 e se durante la traduzione di un indirizzo logico in un indirizzo fisico si ha una entry con tale bit a 0 si ha un page fault.

Gestione dei page fault

Il page fault causa un interrupt al sistema operativo che verifica una tabella associata il processo: se il riferimento è invalido fa un *abort*, altrimenti attiva il caricamento della pagina. Carica un frame vuoto, fa lo swap della pagina nel frame da disco, modifica le tabelle ponendo nella page table il valid bit a 1 e nella tabella interna del processo la pagina in memoria. Infine ripristina l'istruzione che ha causato il page fault. Il primo accesso in memoria di un programma risulta sempre in un page fault nel demand paging puro.

9.1.2 Prestazioni

La paginazione su domanda influenza il tempo di accesso effettivo alla memoria (effective access time, EAT). Con un tasso di page fault p tale che $0 \leq p \leq 1$ per cui $p = 0$ nessun page fault e $p = 1$ ogni accesso è un page fault:

$$EAT = (1 - p) \cdot t_{mem} + p \cdot t_{page\ fault}$$

Dove $t_{page\ fault}$ è dato da tre componenti: il servizio dell'interrupt, lo swap in (lettura della pagina), costo di riavvio del processo e lo swap out opzionale.

9.1.3 Rimpiazzamento delle pagine

Se non ci sono pagine libere si cercano pagine (frame) in memoria e si fa lo swap su disco di tali pagine. La realizzazione richiede un algoritmo che massimizzi le prestazioni minimizzando il numero di page fault. In caso di assenza di frame liberi nel caso di un page fault il sistema operativo verifica su una tabella associata al processo se si tratta di page fault o violazione di accesso. Cerca un frame vuoto e se non c'è usa un algoritmo di rimpiazzamento delle pagine per scegliere un frame vittima, fa lo swap della vittima su disco, lo swap della pagina nel frame da disco, modifica le tabelle e ripristina l'istruzione che ha causato il page fault. Si noti come in assenza di frame liberi il tempo di page fault si raddoppia. Pertanto si ottimizza utilizzando un bit nella page table detto bit di modifica o dirty bit che vale 1 se la pagina è stata modificata dal momento in cui viene ricaricata e solo le pagine che vengono modificate vengono scritte su disco quando diventano vittime.

9.1.4 Problematiche

Le problematiche della paginazione su domanda sono la decisione del rimpiazzamento delle pagine e l'allocazione del numero di frame ad un processo al momento dell'esecuzione.

9.2 Algoritmi di rimpiazzamento delle pagine

L'obiettivo di questi algoritmi è di minimizzare il minimo tasso di page fault. Si valuta l'esecuzione di una particolare stringa di riferimenti a memoria detta reference string e si calcolano il numero di page fault sulla stringa. È sempre necessario sapere il numero di frame disponibili per il processo. Il tasso di page fault è inversamente proporzionale al numero di frame.

9.2.1 Algoritmo FIFO (first-in-first-out)

In questo algoritmo la prima pagina introdotta è la prima ad essere rimossa. L'algoritmo è cieco in quanto non viene valutata l'importanza (frequenza di riferimento) della pagina rimossa. Tende ad aumentare il tasso di page fault e soffre dell'anomalia di Belady.

Anomalia di Belady

Nell'anomalia di Belady il numero di page fault non può decrescere all'aumentare del numero di frame usando FIFO, mentre a volte più frames equivalgono a più page fault.

9.2.2 Algoritmo ideale

Garantisce il minimo numero di page fault rimpiazzando le pagine che non saranno usate per il periodo di tempo più lungo. Questa informazione richiede una conoscenza anticipata della stringa dei riferimenti e ci si ritrova in una situazione simile a SJF e pertanto l'implementazione è impossibile a meno di approssimazioni. Rimane comunque un utile riferimento per altri algoritmi.

9.2.3 Algoritmo least recently used (LRU)

Questo algoritmo è un'approssimazione dell'algoritmo ottimo e usa il passato recente come previsione del futuro rimpiazzando la pagina che non viene usata da più tempo. È di difficile implementazione in quanto non è banale ricavare il tempo dell'ultimo utilizzo e può richiedere notevole hardware addizionale.

Implementazioni

Tramite contatore Ad ogni pagina è associato un contatore e ogni volta che la pagina viene referenziata il clock di sistema è copiato nel contatore. Si rimpiazza la pagina con il valore più piccolo del contatore. Si noti come tale pagina vada cercata.

Tramite stack Viene mantenuto uno stack di numeri di pagina e a ogni riferimento ad una pagina questa viene messa in cima allo stack. L'aggiornamento richiede l'estrazione di un elemento interno allo stack. Al fondo dello stack si trova la pagina LRT. Si noti come in questo caso non è necessaria alcuna ricerca.

Uso del bit di reference Che viene associato ad ogni pagina inizialmente a 0. Quando la pagina è referenziata messo a 1 dall'hardware. Per il rimpiazzamento si sceglie una pagina che ha il bit a *o*. Si nota come è approssimato in quanto non viene verificato l'ordine di riferimento delle pagine.

Alternativa Si usano più bit di reference (registro di scorrimento) per ogni pagina. I bit vengono aggiornati periodicamente e si usano i bit come valori interi per scegliere la LRU, ovvero quella con il valore più basso di registro di scorrimento.

Approssimazioni

Algoritmo LFU (Least Frequently Used) Mantiene un conteggio del numero di riferimenti fatti ad ogni pagina e rimpiazza quella con il conteggio più basso.

Algoritmo MFU (Most Frequently Used) L'opposto di LFU, si basa sul concetto che la pagina con il conteggio più basso è molto probabilmente stata appena caricata e dovrà presumibilmente essere usata ancora.

Rimpiazzamento second chance (clock) Si basa su una FIFO circolare basata su bit di reference: se il bit è a 0 si rimpiazza, se a 1 si mette a 0 e si analizza la pagina successiva.

Variante Si usano più bit di reference.

9.3 Allocazione dei frame

Data una memoria con N frame e M processi è importante scegliere quanti frame allocare ad ogni processo non violando il fatto che ogni processo necessita di un minimo numero di pagine per poter essere eseguito in quanto l'istruzione interrotta da un page fault deve esser fatta ripartire. Pertanto il numero di pagine deve essere uguale al massimo numero di indirizzi specificabile in un'istruzione. I valori tipici vanno da 2 a 4 frame. Inoltre lo schema di allocazione può essere fisso, in cui un processo ha sempre lo stesso numero di frame o variabile, in cui il numero di frame allocati può variare durante l'esecuzione.

9.3.1 Contesto del rimpiazzamento

Nel caso di page fault le vittime possono essere scelte localmente: il processo seleziona vittime solo tra i propri frame o globalmente in cui un processo sceglie in frame dall'insieme di tutti i frame di altri processi. Migliora il throughput e pertanto è più utilizzato.

9.3.2 Allocazione fissa

Allocazione in parti uguali

Dati m frame e n processi si alloca ad ogni processo $\frac{m}{n}$ frame.

Allocazione proporzionale

Alloca secondo la dimensione del processo, parametro non sempre significativo in quanto la priorità può essere più significativa.

9.3.3 Allocazione variabile

Permette di modificare dinamicamente le allocazioni ai vari processi. Le basi della modifica avvengono in base al calcolo del working set o della page fault frequency (PFF).

Calcolo del working set

Un criterio per rimodulare l'allocazione dei frame consiste nel calcolare quali sono le richieste effettive di ogni processo in base al modello della località: un processo passa da una località di indirizzi all'altra durante la sua esecuzione come array, procedure e moduli. Idealmente un processo necessita di un numero di frame pari alla sua località.

Modello del working set Il frame sufficiente a mantenere in memoria il suo working set $WS_i(t, \Delta)$ è il numero di pagine referenziate nell'intervallo di tempo $[t - \Delta, t]$ più recente detto finestra del working set. Se Δ è troppo piccolo è poco significativo, se troppo grande copre varie località e se vale infinito comprende tutto il programma. Il working set viene calcolato approssimando tramite timer e bit di reference: si usa un timer che interrompe periodicamente la CPU, all'inizio di ogni periodo i bit di reference vengono posti a 0 e ad ogni interruzione del timer le pagine vengono scansionate e quelle con bit di reference a 1 si trovano nel working set, quelle con valore 0 vengono scartate. L'accuratezza aumenta in base al numero di bit e alla frequenza di interruzioni. La richiesta totale di frame è $D = \sum_i WSS_i$. Se D è maggiore del numero totale di frame si verifica il thrashing in cui un processo spende tempo di CPU continuando a swappare pagine da e verso la memoria è la conseguenza di un basso numero di frame e di un circolo vizioso.

Thrashing

Se il numero di frame allocati ad un processo scende sotto un certo minimo il tasso di page fault tende a crescere portando a un abbassamento dell'utilizzo della CPU dovuto al fatto dell'attesa per la gestione dei page fault e il sistema operativo tende ad aumentare il grado di multiprogrammazione aggiungendo processi che rubano frame ai vecchi processi aumentando il tasso di page fault. A un certo punto il throughput precipita e si deve pertanto stimare con esattezza il numero di frame necessari a un processo per non entrare in thrashing.

Frequenza dei page fault

Una soluzione alternativa e più accurata del working set è la frequenza dei page fault: si stabilisce un tasso di page fault accettabile. Se quello effettivo è troppo basso il processo rilascia dei frame, se è troppo alto ne ottiene altri.

9.4 Conclusioni

La selezione della dimensione della pagina deve essere fatta accuratamente in quanto si deve sempre fare un trade-off:

Pagine piccole:

- Frammentazione.
- Molte entry nella page table.
- Costo di lettura e scrittura non ammortizzato.
- Località.

Pagine grandi:

- Frammentazione interna significativa.
- Grande dimensione della page table.
- I/O overhead.
- Grande granularità, si deve anche trasferire ciò che non è necessario.

Naturalmente la struttura dei programmi influisce sul numero di page fault e in alcuni casi esistono frame che non devono essere mai rimpiazzati come frame corrispondenti a pagine del kernel e altri corrispondenti a pagine usate per trasferire dati da e verso I/O. Questi subiscono il blocco di frame (frame locking).

Capitolo 10

Gestione della memoria secondaria

10.1 Tipologie di supporto

10.1.1 Nastri magnetici

I nastri magnetici sono formati da una sottile striscia di materiale plastico rivestita di un materiale magnetizzabile. Furono usati per memorizzare dati digitali per la prima volta nel 1951. La massima capienza è di circa $5TB$ nel 2011. Sono ad accesso sequenziale e molto più lenti della memoria principale e dei dischi rigidi in termini di tempo di accesso. Il riposizionamento della testina di lettura richiede decine di secondi. Sono stati rimpiazzati da dischi magnetici e memorie a stato solido e usati solo per backup.

10.1.2 Dischi magnetici

Sono piatti di alluminio o di altro materiale ricoperti di materiale ferromagnetico con una massima capienza di $4TB$ nel 2011. Il fattore di forma o diametro è sempre più piccolo in modo da raggiungere maggiori velocità di rotazione. Parte da 3.5 pollici per i sistemi desktop e fino a 1 pollice per i mobili. La lettura e la scrittura avvengono tramite una testina sospesa sulla superficie magnetica. Nella scrittura il passaggio di corrente positiva o negativa attraverso la testina magnetizza la superficie, mentre nella lettura il passaggio sopra un'area magnetizzata induce una corrente positiva o negativa nella testina.

Struttura di un disco

Da un punto di vista fisico il disco è costituito da superfici (platter, i dischi), cilindri (cylinder, le tracce di tutti i dischi con la stessa posizione), tracce (cerchi di settori all'interno di un disco) e settori (sottoaree del disco disposte in cerchi concentrici).

Settore Il settore è l'unita più piccola di informazione che può essere letta o scritta su disco. Ha una dimensione variabile tra i 32 byte e i $4KB$ (tipicamente di 512 byte). Sono generalmente raggruppati logicamente i cluster per aumentare l'efficienza. La dimensione dei cluster dipende dalla grandezza del disco e dal sistema operativo (da $2KB$ a $32KB$). Un file occupa sempre almeno un cluster. Per accedere a un settore si necessita di sapere la superficie, la traccia e il settore stesso.

Tempo di accesso al disco

Il tempo di accesso è la somma del seek time (tempo necessario a spostare la testina sulla traccia), del latency time (latenza, il tempo necessario a posizionare il settore desiderato sotto la testina che dipende dalla velocità di rotazione) e del transfer time (il tempo necessario al settore per passare sotto la testina). Il seek time va da $3ms$ fino a $15ms$, mediamente $9ms$ per dischi desktop. La latenza, considerando in media mezza rotazione del disco per ogni accesso è $0.5 \cdot (\frac{60}{\text{velcità di rotazione}})$. Per un disco da $7200rpm$ è di $4.16ms$. Il transfer time è il rapporto tra la dimensione del blocco e la velocità di trasferimento. Per un disco da $7200rpm$ si ha: disk-to-buffer $1030\frac{Mbits}{sec}$ e buffer-to-computer $300\frac{MB}{sec}$. Si noti come il seek time è dominante e pertanto dato il grande numero di processi che accedono al disco è necessario minimizzare il seek time tramite algoritmi di scheduling per ordinare gli accessi al disco in modo da minimizzare il tempo di accesso totale (riducendo lo spostamento della testina) o massimizzare la banda, il numero di byte trasferiti nell'unità di tempo, che misura la velocità effettiva.

10.1.3 Dispositivi a stato solido

Utilizzano chip NVRAM o memorie flash NAND per memorizzare dati in modo non volatile. Usano la stessa interfaccia dei dischi fissi e pertanto possono rimpiazzarli facilmente. Hanno una capacità fino a $2TB$.

Performance delle flash memory

Hanno letture simili a DRAM negli ordini dei nanosecondi e scritture simili ai dischi magnetici nell'ordine dei millisecondi. Rispetto ai dischi fissi sono meno soggette a danni, ma con un limite superiore sul numero di write, più silenziosi, più efficienti nel tempo di accesso, non necessitano di essere deframmentate e sono più costose.

10.2 Scheduling degli accessi a disco

Logicamente il disco può essere considerato come un vettore unidimensionale di blocchi logici. Un blocco o cluster è l'unità minima di trasferimento. Il vettore è mappato sequenzialmente sui settori del disco. Il settore 0 è il primo settore della prima traccia del cilindro più esterno e la numerazione procede gerarchicamente per settore, tracce e piatti.

10.2.1 Disk scheduling

Un processo che necessita I/O esegue una system call e il sistema operativo usa la vista logica del disco e una sequenza di accessi sono una sequenza di indici del vettore. Nelle richieste sono anche contenute informazioni come il tipo di accesso (R/W), l'indirizzo di memoria destinazione e la quantità di dati da trasferire.

10.2.2 Algoritmi di disk scheduling

In questi algoritmi bisogna sempre tenere conto del compromesso tra costo ed efficacia. Sono valutati tramite una sequenza di accessi.

First-come-first-served (FCFS)

In questo algoritmo le richieste sono processate nell'ordine di arrivo.

Shortest seek time first (SSTF)

In questo algoritmo si fa la scelta più efficiente localmente: si seleziona la richiesta con il minimo spostamento rispetto alla posizione attuale della testina. Simile al SJF con possibilità di starvation di alcune richieste. Non è ottimo.

SCAN

La natura dinamica delle richieste porta all'algoritmo SCAN in cui la testina parte da un'estremità del disco, si sposta verso l'altra estremità servendo tutte le richieste correnti. Arrivato all'altra estremità riparte nella direzione opposta servendo le nuove richieste. Viene detto algoritmo dell'ascensore.

SCAN circolare (CSCAN)

Come SCAN, ma quando la testina arriva ad un'estremità riparte immediatamente da 0 senza servire altre richieste. Il disco viene considerato come lista circolare e il tempo di attesa è più uniforme rispetto a SCAN. Alla fune di una scansione ci saranno più richieste all'altro estremo.

(C-)LOOK, variante dello SCAN

La testina non arriva fino all'estremità del disco. Cambia direzione (LOOK) o riparte dalla prima traccia (C-LOOK) non appena non ci sono più richieste in quella direzione.

N-step SCAN, variante dello SCAN

Per evitare che la testina rimanga sempre nella stessa zona la coda delle richieste viene partizionata in più code di dimensione massima N. Quando una coda viene processata per il servizio gli accessi in arrivo riempiono altre code. Dopo cue una coda è stata riempita non è possibile riordinare le richieste. Le code sature vengono servite nello scan successivo. Per N grandi degenera in SCAN, mentre per $N = 1$ degenra in FCFS. FSCAN è simile ma con due sole code.

Last-in-last-out (LIFO)

In certi casi può essere utile schedulare gli accessi in base all'ordine inverso di arrivo. È utile nel caso di accessi con elevata località ma offre possibilità di starvation.

Analisi degli algoritmi

Nessuno degli algoritmi considerati è ottimo in quanto quello ottimo sarebbe poco efficiente. L'analisi è influenzata dalla distribuzione di numero e dimensione degli accessi in quanto più l'accesso è grande più il peso relativo del seek time diminuisce e dall'organizzazione delle informazioni su disco (il file system) e l'accesso alla directory essenziale tipicamente lontane dalle estremità del disco. In generale SCAN e C-SCAN sono migliori per sistemi con molti accessi a disco. Il disk-scheduling viene spesso implementato come modulo indipendente dal sistema operativo con diverse scelte di algoritmo.

10.3 Gestione del disco

10.3.1 Formattazione dei dischi

Nella formattazione di basso livello o fisica il disco viene diviso in settori che il controllore può leggere o scrivere. Per usare un disco come contenitore di file il sistema operativo deve memorizzare le proprie strutture del disco che viene partizionato in uno o più gruppi di cilindri detti partizioni. Avviene una formattazione logica per creare un file system e il programma di boot per inizializzare il sistema. Il programma di bootstrap memorizzato nella ROM (bootstrap loader) carica il bootstrap dal disco (dai blocchi di root), carica i driver dei dispositivi e lancia l'avvio del sistema operativo. Nella formattazione fisica si suddivide il disco in settori, si identificano e si aggiunge lo spazio per la correzione degli errori (ECC). Nella formattazione logica si inizializza il file system, la lista di spazio occupato e libero e le directory vuote.

10.3.2 Gestione dei blocchi difettosi

L'ECC (error correction code) serve a capire in lettura o scrittore se un settore contiene dati corretti o meno. Per la lettura il controllore legge un settore insieme all'ECC e calcola l'ECC per i dati appena letti. Se il risultato è diverso si trova un errore (bad block). L'errore va segnalato e i bad block si possono gestire offline o online.

Gestine offline dei bad block

Durante la formattazione logica si individuano i bad block e si mettono una lista, si rimuovono e si marca l'entry nella FAT. Successivamente si può eseguire un'utility per isolare i bad block come. Questo algoritmo è tipico dei controllori IDE.

Gestione online dei bad block

Il controllore mappa il bad block su un blocco buono non usato. Si deve partano disporre di blocchi di scorta riservati (sector sparing). Quando il sistema operativo richiede il bad block il controllore mappa in modo trasparente l'accesso al bad block sul blocco di scorta. Potrebbe inficiare le ottimizzatone fornite dallo scheduling se non si allocassero spare block in ogni cilindro.

10.3.3 Gestione dello spazio di swap

Lo spazio di swap viene usato dalla memoria virtuale come estensione della RAM. Si può ricavare dal file system attraverso comuni primitive di accesso ai file, ma è inefficiente in quanto esiste un costo addizionale per l'attraversamento delle strutture dati per le directory e i file. Si può porre in una partizione separata che non contiene informazioni e strutture relative al file system e usa uno speciale gestore detto swap daemon. Lo spazio di swap può essere allocato quando il processo viene creato e viene assegnato spazio per pagine di istruzioni e di dati. Il kernel usa due mappe di swap per tracciare l'uso dello spazio di swap. Oppure può essere allocato quando una pagina viene forzata fuori dalla memoria fisica.

Capitolo 11

File System

Il file system fornisce il meccanismo per la memorizzazione e l'accesso di dati e programmi. Consiste di una collezione di file e della struttura di cartelle (directory)

11.1 Interfaccia del file system

11.1.1 Concetto di file

Il sistema operativo astrae dalle caratteristiche fisiche dei supporti di memorizzazione fornendo una loro visione logica. Si considera un file come uno spazio di indirizzamento logico e contiguo rappresentante un insieme di informazioni correlate identificate da un nome. I File possono essere dati di qualsiasi tipo e programmi.

11.1.2 Attributi di un file

Su disco nella struttura della directory sono salvati per ogni file il nome (unica informazione in formato leggibile), tipo, posizione (puntatore allo spazio fisico sul dispositivo), dimensione, protezione (controllo sui permessi), tempo, data e identificazione dell'utente.

11.1.3 Operazioni sui file

- Creazione: si cerca lo spazio necessario su disco e si crea un nuovo elemento sulla directory per gli attributi.
- Scrittura: una system call che specifica il nome del file e i dati da scrivere, è necessario che sia noto il puntatore alla locazione della prossima scrittura.
- Lettura: una system call che specifica il nome del file e dove mettere i dati letti in memoria, è necessario conoscere il puntatore alla locazione della prossima lettura, lo stesso della scrittura.
- Riposizionamento all'interno di file: aggiornamento del puntatore alla posizione corrente.
- Cancellazione: libera lo spazio associato al file e l'elemento corrispondente nella directory.
- Troncamento: mantiene inalterati gli attributi ma cancella il contenuto del file.

- Apertura: ricerca il file nella struttura della directory su disco, copia il file in memoria e inserisce un riferimento nella tabella dei file aperti. In sistemi multi utente si trovano 2 tabelle: una per ogni processo che contiene i riferimenti per i file aperti relativi a esso e una per tutti i file aperti da tutti i processi che contiene i dati indipendenti dal processo.
- Chiusura: copia del file in memoria su disco.

11.1.4 Struttura dei file

I tipi di file possono essere usati per usare la struttura interna del file. In Unix non esiste, si considera un file una sequenza di parole o bytes. Può essere una struttura a record semplice doe un record è una riga di lunghezza fissa o variabile. Può essere una struttura complessa come accade dei documenti formattati e i formati ricaricabili (load module). Le ultime due si possono emulare con la prima usando caratteri di controllo.

11.1.5 Metodi di accesso

Sequenziale

Questo metodo viene usato da editor e compilatori, permettono le operazioni *read next*, *write next*, *reset* (rewind). Non è permesso il rewrite in quanto si rischia inconsistenza se si scrive qualcosa a metà del file in quanto si potrebbe eliminare ciò che sta dopo.

Diretto

Come accade nel database in cui un file è una sequenza numerata di blocchi detti record. Le operazioni permesse sono *read n*, *write n*, *position to n*, *read next*, *write next*, *rewrite n*.

11.2 Struttura delle directory

Le directory sono una collezione di nodi contenente informazioni sui file. Si trovano entrambi sul disco e determinano la struttura del file system. Per ogni file contengono il nome, tipo, indirizzo, lunghezza attuale, massima lunghezza, data di ultimo accesso, data di ultima modifica, proprietario e informazioni di protezione.

11.2.1 Operazioni sulla directory

- Aggiungere un file.
- Cancellare un file.
- Visualizzare il contenuto della directory.
- Rinominare un file.
- Cercare un file.
- Attraversare il file system.

11.2.2 Organizzazione logica

Gli obiettivi dell'organizzazione logica delle directory sono l'efficienza (rapido accesso ad un file), nomenclatura conveniente agli utenti in quanto permette stessi nomi per file e utenti diversi e nomi diversi per lo stesso file e raggruppamento: i file vengono classificati logicamente per un criterio come il tipo o la protezione.

Directory a un livello

In questo metodo si trova una singola directory per tutti gli utenti. Nascono problemi di nomenclatura in quanto è difficile ricordare se un nome esiste già e inventarne sempre di nuovi. Nascono inoltre problemi di raggruppamento.

Directory a due livelli

In questo metodo si trova una directory separata per ogni utente. Nasce il concetto di percorso (path), è possibile usare lo stesso nome di file per utenti diversi e la ricerca è efficiente. Non esiste comunque raggruppamento e si pone il problema di dove mettere i programmi di sistema condivisi dagli utenti (in Unix si usa *PATH*).

Directory ad albero

In questo metodo la directory viene implementata come un albero permettendo una ricerca efficiente e la possibilità di raggruppamento. Nasce il concetto di working directory e i nomi di percorso possono essere assoluti o relativi.

Directory a grafo aciclico

In questo metodo si espande la versione ad albero a grafo aciclico per permettere la condivisione di file. La condivisione avviene in Windows attraverso i collegamenti e in Unix attraverso i link. I link possono essere simbolici: contengono il pathname del file reale e se si cancella il file il link rimane pendente o hard: si trova un contatore che mantiene il numero di riferimenti decrementato per ogni cancellazione di un riferimento. Il file può essere cancellato solo se il contatore vale 0.

Directory a grafo generico

Questo metodo nasce come estensione della directory a grafo aciclico e si deve garantire che le ricerche terminino per evitare loop infiniti nell'attraversamento del grafo. Per risolvere questo problema si può permettere di collegare solo file non directory o usare un algoritmo di controllo di esistenza di un ciclo ogni volta che si crea un nuovo collegamento, che può essere costoso.

11.2.3 Mount di file system

Si possono creare dei file system modulari con la possibilità di attaccare (mount) e staccare (unmount) interi file system a file system precedenti. In generale un file system deve essere montato prima di potervi accedere. Ci si riferisce al punto in cui viene montato come al mount point.

11.2.4 Condivisione di file

La condivisione di file è importante in sistemi multiutente ed è realizzabile tramite uno schema di protezione. Nei sistemi distribuiti i file possono essere condivisi attraverso una rete. Il Network File System (NFS) è un tipico schema di condivisione di file via rete.

11.2.5 Protezione

Il possessore di un file deve poter controllare cosa è possibile fare su un file e da parte di chi. Per farlo si può implementare una lista di accesso per ogni file o directory che lista chi può fare cosa. Può essere lunga. Gli utenti si possono raggruppare in tre classi con una perdita di generalità: il proprietario, il gruppo (utenti appartenenti allo stesso gruppo del proprietario) e gli altri. Per ogni classe i permessi possono essere di lettura, scrittura o esecuzione.

11.3 Implementazione del file system

Per gestire un file system si usano diverse strutture dati in parte su disco e in parte in memoria. Le caratteristiche, pur avendo una base comune sono fortemente dipendenti dal sistema operativo e dal tipo di file system.

11.3.1 Strutture su disco

- Blocco di boot: contiene le informazioni necessarie per l'avvio del sistema operativo.
- Blocco di controllo delle partizioni: contiene i dettagli riguardanti la partizione come numero e dimensione dei blocchi, la lista dei blocchi liberi, dei descrittori liberi.
- Strutture di directory che descrivono l'organizzazione dei file.
- Descrittori dei file che contengono vari dettagli sui file e i puntatori ai blocchi dati.

11.3.2 Strutture in memoria

- Tabella delle partizioni: informazioni sulle partizioni montate.
- Strutture di directory: copia in memoria delle directory a cui si è fatto accesso di recente.
- Tabella globale dei file aperti con le copie dei descrittori dei file.
- Tabella dei file aperti per ogni processo contenente il puntatore alla tabella globale e le informazioni di accesso.

11.3.3 Allocazione dello spazio su disco

I blocchi su disco possono essere allocati ai file o alle directory in modi diversi con l'obiettivo di minimizzare i tempi di accesso e massimizzare l'utilizzo dello spazio.

Allocazione contigua

In questo metodo ogni file occupa un insieme di blocchi contigui su disco. L'entry della directory è semplice in quanto contiene l'indirizzo del blocco di partenza e la lunghezza del file. Permette un accesso semplice in quanto l'accesso al blocco $b+1$ non richiede lo spostamento della testina rispetto al blocco b a meno che b non sia l'ultimo blocco di un cilindro. Supporta sia l'accesso sequenziale che casuale in quanto per leggere il blocco i di un file che inizia al blocco b basta leggere $b + i$. Presenta però problemi simili a quelli dell'allocazione dinamica: si deve scegliere tra algoritmi best-fit, first-fit o worst-fit e porta a uno spreco di spazio dovuto a frammentazione esterna e richiede una compattazione periodica dello spazio. La decisione della dimensione dello spazio da allocare è un problema in quanto i file potrebbero crescere dinamicamente. Si può fare in modo che se il file deve crescere e non c'è spazio nasce un'errore e una terminazione del programma e si riesegue. In questo modo si tende a sovrastimare lo spazio necessario che porta a uno spreco. Un'altra soluzione consiste di trovare un buco più grande e ricopiare tutto il file in questo, operazione trasparente per l'utente ma che rallenta il sistema.

Variante Alcuni file system moderni usano uno schema modificato di allocazione contigua come Linux ext2fs basato sull'extent, una serie di blocchi contigui su disco. Il file system alloca extent invece di blocchi singoli che in generale non sono contigui e si prestano pertanto per l'allocazione a lista.

Allocazione a lista

In questo metodo ogni file si trova su una lista di blocchi che possono essere sparsi ovunque nel disco. La directory contiene puntatori al primo e all'ultimo blocco e ogni blocco contiene un puntatore al blocco successivo. Considerando X l'indirizzo logico e N la dimensione del blocco $\frac{X}{N-1}$ è il numero del blocco nella lista, mentre $X \mod (n-1)$ è l'offset all'interno del blocco. L'allocazione a lista permette una semplice creazione di un nuovo file in quanto basta cercare un blocco libero e creare una nuova entry nella directory che punta ad esso. Anche l'estensione del file è semplice: si cerca un blocco libero e lo si concatena alla fine del file. Non c'è spreco se non si considera lo spazio per il puntatore in quanto si può usare qualunque blocco e non c'è frammentazione esterna. Non permette però accesso casuale in quanto bisogna scorrere tutti i blocchi a partire dal primo e richiede tanti riposizionamenti sparsi che portano a scarsa efficienza. È inoltre scarsamente affidabile in quanto la perdita di un puntatore o un errore causa il prelevamento del puntatore sbagliato si devono introdurre metodi di recupero con overhead come le liste doppiamente concatenate e memorizzare il nome del file e il numero di blocco in ogni blocco del file.

Esempio - FAT (file allocation table) Si trova una FAT per ogni partizione che contiene un elemento per ogni blocco del disco usata come lista concatenata che migliora l'accesso causale, che rimane comunque con bassa efficienza.

Allocazione indicizzata

In questo metodo ogni file ha un blocco indice (index block) che contiene la tabella degli indirizzi dei blocchi fisici. La directory contiene l'indirizzo del blocco indice. L'accesso casuale è efficiente e quello dinamico è senza frammentazione esterna con overhead del blocco indice per la index table, maggiore di quello richiesto per l'allocazione concatenata. Se X è l'indirizzo logico e N la dimensione del blocco $\frac{X}{N}$ è l'offset nella index table e $X \mod N$ è l'offset all'interno del blocco dati. Si noti

come la dimensione del blocco limita la dimensione del file, pertanto per i file di grandi dimensioni si usa uno schema a più livelli.

Indici multilivello Una tabella più esterna contiene puntatori ad ulteriori index table. Sia X l'indirizzo logico e N la dimensione del blocco in parole $\frac{X}{n \cdot N}$ è il blocco della index table di primo livello e $X \mod (N \cdot N) = R$, dove $\frac{R}{N}$ è l'offset nel blocco della index table di secondo livello e R mod N è l'offset nel blocco dati.

Schema concatenato Si trova una lista concatenata di blocchi indice: l'ultimo degli indici id un blocco indice punta a un altro blocco indice e $\frac{X}{N(N-1)}$ è il numero del blocco indice all'interno della lista dei blocchi indice e $X \mod (N(N-1)) = R$, dove $\frac{R}{N}$ è l'offset nel blocco indice e $R \mod N$ è l'offset nel blocco dati.

Unix e i-node

Unix usa lo schema combinato in cui ad ogni file è associato u i-node (index-node) che contiene gli attributi del file e fa da blocco indice. Gli i-node sono gestiti dal sistema operativo e memorizzati in modo permanente in una porzione riservata al sistema operativo di solito all'inizio del disco. Per memorizzare i blocchi di dati del file ogni i-node contiene:

- 10 puntatori diretti ai blocchi di dati di file.
- Un puntatore single indirect che punta ad un blocco indice che contiene puntatori a blocchi di dati di file.
- Un puntatore double indirect che punta ad un blocco indice che contiene puntatori a blocchi indice contenenti puntatori a blocchi di dati di file.
- Un puntatore triple indirect che punta ad un blocco indice che contiene puntatori a blocchi indice che contengono puntatori a blocchi indice che contengono puntatori a blocchi di dati di file.

11.3.4 Implementazione delle directory

Le directory vengono implementate con lo stesso meccanismo usato per memorizzare i file. Non contengono file ma la lista di file e directory che contengono. Questo spazio può essere memorizzato come lista lineare di nomi di file con puntatori ai blocchi dati di facile implementazione ma poco efficiente in quanto lettura, scrittura e rimozione di file richiedono ricerca per trovare il file. Può essere anche implementato come tabella hash con tempo di ricerca migliore ma che richiede la gestione delle collisioni.

11.4 Gestione dello spazio libero

Per tenere traccia dello spazio libero su disco si mantiene una lista dei blocchi liberi. Per creare un file si cercano blocchi liberi nella lista e per rimuovere un file si aggiungono i suoi blocchi a tale lista.

11.4.1 Alternative

- Vettore di bit: si trova un elemento per ogni blocco tale che $Bit[i] = 0$ implica che il blocco i è libero e $Bit[i] = 1$ implica che è occupato. La mappa di bit richiede spazio extra ed è efficiente solo se il vettore è mantenibile tutto in memoria. Risulta comunque facile ottenere blocchi contigui.
- Lista concatenata di blocchi liberi (free list): permette spreco minimo in testa alla lista e lo spazio contiguo non è ottenibile.
- Raggruppamento: modifica la lista linkata in modo che il primo blocco libero contiene gli indirizzi di $n - 1$ blocchi liberi e l'ultima entry del blocco l'indirizzo del primo blocco del gruppo successivo di blocchi liberi. Fornisce rapidamente un gran numero di blocchi liberi.
- Conteggio: mantiene il conteggio di quanti blocchi liberi seguono il primo in una zona di blocchi liberi contigui. La lista risulta più corta se il contatore è maggiore di 1 per ogni gruppo di blocchi liberi.

11.5 Efficienza e prestazioni

11.5.1 Efficienza

Il disco è un collo di bottiglia e l'efficienza dipende da un algoritmo di allocazione dello spazio su disco: in Unix si cerca di tenere i blocchi di un file vicini al suo i-node, viene richiesta la preallocazione i-node distribuiti sulla partizione e dal tipo di dati contenuti nella directory: la data di ultimo accesso di un file richiede che la lettura di un file richieda la lettura e scrittura del blocco della directory.

11.5.2 Prestazioni

Il controller del disco possiede una piccola cache che è in grado di contenere un'intera traccia ma non basta per garantire prestazioni elevate pertanto si usano dischi virtuali (RAM disk) e cache del disco o buffer cache

Dischi virtuali

Parte della memoria viene gestita come se fosse un disco: il driver di un RAM disk accetta tutte le operazioni standard dei dischi eseguendole in memoria. È veloce ma supporta solo file temporanei. Viene gestito dall'utente che scrive sul RAM disk invece che sul disco.

Cache del disco

È una porzione di memoria che memorizza blocchi usati di frequente simile alla cache tra memoria e CPU. Viene gestita dal sistema operativo e sfrutta il principio di località spaziale e temporale. Il trasferimento di dati nella memoria del processo utente non richiede spostamento di byte. Nascono problematiche riguardanti la politica di rimpiazzamento attraverso LRU (poco efficiente, meglio rilascio indietro e lettura anticipata), LFU, RANDOM. Se l'operazione è una scrittura si può fare write-back: si scrive solo quando si deve rimuovere il blocco dalla cache (può generare problemi di affidabilità in caso di crash) o write-through: si scrive sempre, meno efficiente con la cache in sola lettura.

Recupero

I problemi di consistenza tra disco e cache possono essere controllati attraverso un controllo di consistenza che confronta i dati nella directory con i dati su disco e sistema le inconsistenze o attraverso l'uso di programmi di sistema per fare il backup del disco su memoria di massa come i nastri che permettono il recupero di file persi tramite il restore dei dati di backup.

File system log structured

Si registra ogni cambiamento del file system come una transazione. Tutte le transazioni sono scritte su log e considerate completate quando viene scritta su esso (anche se il file system può non essere aggiornato). Le transazioni sono scritte in modo asincrono nel file system e quando il file system è modificato la transazione viene cancellata dal log. Se il sistema va in crash le transazioni non avvenute sono quelle presenti sul log. In questo modo si ottimizza il numero di seek.

Capitolo 12

Sistemi RAID

L'evoluzione logica ha permesso di avere dischi sempre più piccoli e meno costosi, pertanto è più facile equipaggiare un sistema con molti dischi in modo da raggiungere maggiori prestazioni attraverso letture e scritture in parallelo e maggior affidabilità tramite ridondanza. Il RAID nasce nel 1988 come Redundant Array of Independent Disks, con l'obiettivo di migliorare l'affidabilità incrementando le prestazioni. La struttura software dei dispositivi RAID si basa su più dischi indipendenti collegati al bus e la funzionalità RAID implementata dal sistema operativo. La struttura hardware si basa su un controllore intelligente che gestisce diversi dischi collegati alla macchina. Una batteria RAID è un'unità a sè stante composta da controllore, cache e dischi autonomi collegati a una macchina,

12.1 Concetti di base

Le strutture RAID si basano su copiatura speculare dei dati (mirroring) e sezionamento dei dati (data striping) per implementare parallelismo che garantisce aumento di prestazioni e affidabilità.

12.1.1 Affidabilità

Un guasto a un disco comporta la perdita di dati e per migliorare l'affidabilità si deve ricorrere alla ridondanza: memorizzare informazioni non strettamente necessarie ma utili per ricostruire le informazioni perse in caso di guasto.

Copiatura speculare

Il modo più semplice per implementare la ridondanza è il mirroring o shadowing in cui un disco logico corrisponde a due dischi fisici, ogni scrittura avviene su entrambi i dischi e i dati si perdono solo se si guastano entrambi i dischi. Il tempo medio di perdita dei dati dipende dal tempo medio di guasto di ogni singolo disco e il tempo medio di riparazione. Si noti come offra protezione solo contro i guasti indipendenti e per cause esterne entrambi i dischi potrebbero guastarsi contemporaneamente. Con mirroring la frequenza di gestione delle letture raddoppia perchè si può leggere da uno qualunque dei due dischi e il tempo di trasferimento rimane inalterato.

12.1.2 Sezionamento dei dati

Usando più dischi è possibile migliorare la capacità di trasferimento distribuendo i dati in sezioni su più dischi (data striping). Il sezionamento può avvenire a livello di bit: i bit di ciascun byte vengono distribuiti su più dischi o a livello di blocco in cui i blocchi dei file sono distribuiti su più dischi. Il parallelismo tramite il bilanciamento del carico aumenta la produttività per accessi multipli a piccole porzioni di dati e riduce il tempo di risposta relativo agli accessi a grandi quantità di dati.

12.1.3 Codici per la correzione di errori

Ad ogni byte si associa un bit di parità che gli indica se gli 1 presenti nel byte sono in numero pari (parità 0), o in numero dispari (parità 1). Identificano tutti gli errori su un singolo bit. Usando più bit supplementari si riescono a individuare e correggere un maggior numero di bit.

12.2 Livelli di RAID

Il mirroring garantisce alta affidabilità ma è costoso, mentre il data striping permette alta capacità di trasferimento dati ma non migliora l'affidabilità. Spesso si usano tecniche basate sui bit di parità. L'utilizzo combinato di queste tecniche è stato schematizzato in 6 livelli di RAID.

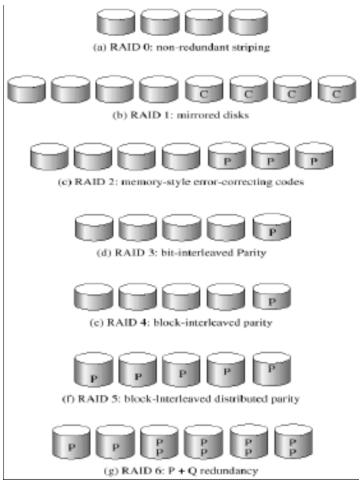

Figura 12.1: Livelli di RAID

12.2.1 Livello RAID 0

Avviene il sezionamento a livello di blocco senza ridondanza. È economico e garantisce alte prestazioni grazie al parallelismo delle operazioni di lettura e scrittura. Non ha ridondanza e l'affidabilità cala all'aumentare del numero di dischi impiegati.

12.2.2 Livello RAID 1

Avviene mirroring senza sezionamento di blocco, aumenta l'affidabilità linearmente con il numero di copie. Si aumentano le prestazioni in lettura (se il disco è occupato si può leggere dall'altro). Ha un alto costo e bassa scalabilità.

12.2.3 Livello RAID 2

Avviene il sezionamento a livello di bit e usa codici per la correzione degli errori (ECC). I bit di correzione sono memorizzati singolarmente in dischi separati diversi rispetto a quelli usati per i dati. Se un disco si guasta i bit rimanenti del byte dati e i bit di correzione associati vengono usati per ricostruire il dato danneggiato. Il RAID 2 richiede solo 3 dischi in più per 4 dischi dati rispetto ai 4 richiesti dal RAID 1. Utilizza il codice Hamming che codifica 4 bit dati in 7 bit, aggiungendone 3 di parità.

12.2.4 Livello RAID 3

Avviene il sezionamento a livello di byte con un disco dedicato al bit di parità noto come organizzazione con bit di parità. I controllori dei dischi sono in grado di rilevare se un settore è stato letto correttamente. Se un settore è danneggiato per ogni bit del settore è possibile determinare se deve valere 0 o 1 calcolando la parità dei bit corrispondenti dai settori degli altri dischi. Ha la stessa efficienza del RAID 2 ma usa un solo disco per i bit di parità. La velocità di trasferimento è pari a n volte quella del RAID 1. Rispetto al RAID 1 fornisce però meno operazioni di I/O al secondo in quanto ogni disco è coinvolto da tutte le richieste e il tempo più lungo per le scritture in quanto è necessario calcolare il bit di parità. Pertanto il controllore RAID sarà capace di gestire il calcolo della parità lasciando la CPU libera.

12.2.5 Livello RAID 4

Avviene il sezionamento a livello di blocco con un disco dedicato alla parità detto organizzazione con blocchi di parità intercalati. È simile al RAID 0 con un blocco di parità in un disco separato. Ha più tolleranza ai guasti e letture più veloci grazie al parallelismo. Il disco usato per la parità può esser un collo di bottiglia e le scritture sono lente a causa del calcolo della parità.

12.2.6 Livello RAID 5

Avviene il sezionamento a livello di blocco con bit di parità distribuiti tra tutti i dischi del RAID o organizzazione con blocchi intercalati a parità distribuita. Un blocco di parità non può contenere informazioni di parità per blocchi che risiedono nello stesso disco. Ha gli stessi vantaggi del RAID 4 senza il collo di bottiglia del disco di parità e come esso ha scritture lente.

12.2.7 Livello RAID 6

È simile al RAID 5 ma con maggiori informazioni di ridondanza per gestire guasti contemporanei su più dischi. Al posto della parità usa altri codici per la correzione degli errori come Reed-Solomon. Ha un altissima ridondanza ma è molto costoso e le scritture sono lente per la gestione dei codici per la correzione degli errori.

12.2.8 Livello RAID 0 + 1

Combina 0 e 1 per fornire affidabilità e alte prestazioni: si trovano due RAID 0 messi in RAID 1. Ha prestazioni migliori rispetto al RAID 5 e alta affidabilità ma richiede il raddoppio del numero di dischi necessari alla memorizzazione dei dati e non supporta la rottura simultanea di due dischi se non appartengono allo stesso stripe.

12.2.9 Livello RAID 1 + 0

Combina 1 e 0 per fornire affidabilità e alte prestazioni: dati n dischi si trovano $\frac{n}{2}$ RAID 1 in RAID 0 è più robusto del RAID 0 + 1 in quanto ogni disco di ogni stripe può guastarsi senza far perdere dati al sistema ma è costoso.