Alma Mater Studiorum · Università di Bologna

DIPARTIMENTO DI INFORMATICA - SCIENZA E INGEGNERIA
Informatica per il management

Sistemi Operativi

Docente:
Davide Sangiorgi

Studente:
Massimo Rondelli

Anno Accademico 2020/2021

Indice

Capitolo 1

Introduction

1.1 What is an operating System?

Iniziamo a introdurre il concetto di sistema operativo. Attraverso la Figura 1.1 possiamo vedere come si colloca il sistema operativo all'interno di un calcolatore. Al livello più basso abbiamo l'hardware, sopra di esso troviamo il sistema operativo, il quale si appoggia direttamente all'hardware della macchina. Tutto ciò che è sopra al sistema operativo, si appoggia ad esso.

Il sistema operativo crea sull'hardware stesso una visone della macchina che non è più la macchina hardware pura, ma è una macchina più astratta, più facile da utilizzare. Abbiamo bisogno del sistema operativo perché nasconde i dettagli dell'hardware rendendo la macchina più astratta e più uniforme. Non è possibile raggiungere una uniformità universale, purtroppo, perché esistono diversi tipi di sistemi operativi e essi forniscono delle macchine astratte leggermente differenti tra di loro.

Il sistema operativo fornisce un'interfaccia verso il mondo esterno che è costituita da istruzione. Tutti i programmi che sono eseguiti su macchine con uno specifico sistema operativo faranno riferimento a queste istruzioni.

Il sistema operativo ha due obiettivi:

1. Efficienza - il sistema operativo conosce esattamente le risorse hardware della macchina e fa in modo che vengano utilizzate nel miglior modo possibile;

2. Convenienza d'uso - il sistema operativo deve rendere l'uso della macchina più semplice possibile.

Il sistema operativo fornisce ai livelli superiore una interfaccia che viene indicata come **API (Application Programming Interface)**. Essa consiste nelle operazione che il sistema operativo fornisce ai livelli soprastanti. Le operazioni che sono presenti nell'API, sono chiamate **system call**.

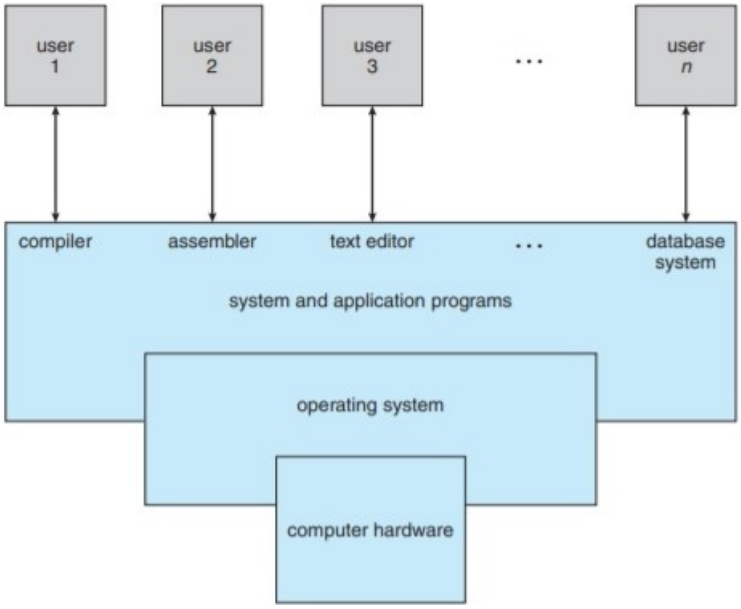

Figura 1.1: Visione astratta dei componenti del calcolatore

Le system call sono le chiamate che fanno i livelli superiori per accedere alla macchina astratta fornita dal sistema operativo. I livelli superiori non possono accedere all'hardware direttamente, ma il sistema operativo fornisce loro una interfaccia attraverso la quale egli possano accedervi.

La macchina astratta fornita dal sistema operativo, oltre a facilitare l'uso dell'hardware, permette anche la *portabilità*. Essendo l'interfaccia del nostro sistema operativo più astratta, il programma che eseguiamo, il quale presuppone l'esistenza di una macchina astratta fornita dal sistema operativo stesso, potrà girare in tutti quei casi in cui il sistema operativo è in funzione. Il sistema operativo nasconde le differenza delle varie macchina, dunque il programma in esecuzione funzionerà anche su macchine molto differenti tra di loro ma che presentano lo stesso sistema operativo; ai livelli superiori viene fornita un'interfaccia comune. Questa è la **portabilità**.

I livelli superiori consistono in programmi applicativi, come per esempio *compiler, assembleer, text editor e database system*. Essi non hanno accesso diretto all'hardware e devono passare dal sistema operativo. Ciò è importante per due motivi:

- Sicurezza - il sistema operativo, con le proprie funzionalità, deve offrire protezione e sicurezza. Se fosse permesso un accesso diretto da parte dei livelli superiori all'hardware della macchina, ci potrebbero essere facilmente degli attacchi in cui un utente malintenzionato manipola l'hardware in modo da mandarlo in condizioni di non operabilità.

- Gestione delle risorse - il sistema operativo deve gestire le risorse della macchina stessa. Egli manipola le risorse per effettuare la gestione; è necessario che conosca in ogni momento *chi sta usando cosa*. Se ci fosse un accesso diretto all'hardware, questo non sarebbe più possibile perché il sistema operativo non saprebbe, per esempio per una periferica di I/O, come viene utilizzata e cosa succede sulla periferica stessa in un certo momento. Questa funzionalità del sistema operativo, di gestione delle risorse, è possibile grazie al fatto che le risorse hardware sono utilizzate unicamente dal sistema operativo stesso.

Dunque, dopo questa descrizione sommaria, possiamo dire che non esiste una vera e propria definizione di sistema operativa. Possiamo indicare però ciò fa:

- Resource allocator - gestisce delle risorse hardware, cioè gestisce le risorse della macchina;
- Control program - controlla l'esecuzione dei programmi utenti e le operazioni sui dispositivi I/O;
- Kernel - viene considerato il programma che è sempre in esecuzione. Una macchina per essere utilizzata deve avere il sistema operativo sempre funzionante.

1.2 Memory Layout for a simple batch system

La memoria centrale in presenza di sistemi operativi primordiali, Figura 1.2, è la memoria che dialoga direttamente con il processore. Il processore è il cervello della macchina. Egli esegue delle istruzioni ma non ha capacità di memoria, quindi egli dialoga con la memoria centrale, in modo che tutto ciò che viene eseguito sul processore viene caricato sulla memoria centrale.

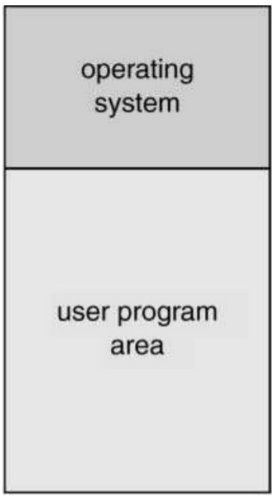

Figura 1.2: Memory Layout

Esiste anche una memoria secondaria, molto più capiente della memoria centrale. Il processore dialoga solamente con la memoria centrale e non quella secondaria perché essa è troppo lenta, circa 3 ordini di grandezza inferiore rispetto a quella centrale.

Inizialmente la memoria centrale era strutturata come si può vedere dalla Figura 1.2, un'area destinata al sistema operativo e un'area destinata ai programmi utente. Era presente un solo utilizzatore della macchina e veniva eseguito un programma per volta, solamente una attività, non c'era ancora il parallelismo.

1.3 Multiprogramming Batch Systems

Col tempo è stato è stato introdotto il parallelismo. Esso permette alla macchina di eseguire più attività nello stesso momento. Il sistema operativo deve assicurarsi che queste diverse attività non si danneggino l'una con l'altra. Facendo riferimento alla Figura 1.3, se viene eseguito *job 1* non vogliamo che egli vada a scrivere all'interno di un'area di memoria che stata assegnata a *job 4*. In questa situazione di parallelismo, dove esistono diverse attività che sono eseguite contemporaneamente viene identificata col nome di *multiprogrammazione*. La multiprogrammazione fa riferimento al fatto che esistono più di una attività in esecuzione per volta.

La multiprogrammazione è stata una svolga nella storia dei sistemi operativo. Una volta era possibile eseguire solamente un programma per volta. In un macchina, il processore è l'unità hardware più importante. Un programma per essere eseguito deve essere caricato sulla memoria centrale. Il processore interagisce solamente con la memoria centrale, questo perché la memoria secondaria è troppo lenta. Una differenza ancora più forte, relativa alle velocità, è la differenza tra il processore rispetto alle periferiche, per esempio la stampante, tastiera. In una macchina di questo genere, Figura 2.1, quando eseguiamo un programma che, a un certo punto dell'esecuzione, necessita dell'uso di una periferiche I/O, come memoria secondaria, stampanti, tastiera, la CPU rimane in attesa per periodi di tempo molto lunghi. Il tempo che serve a una periferica I/O per eseguire il proprio compito, è un tempo morto per il processore. Se non ci fosse la multiprogrammazione, durante questo intervallo di tempo la CPU rimarrebbe inattiva. Può essere un intervallo abbastanza lungo, per esempio mezzo secondo, dove avrebbe potuto eseguire milioni di operazioni.

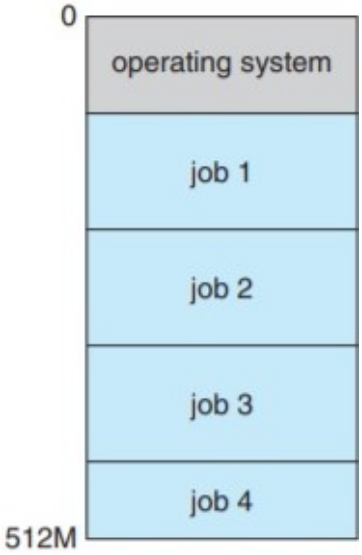

Figura 1.3: Multiprogrammed Batch Systems

L'idea della multiprogrammazione, dal punto di vista di **efficienza** della nostra macchina, è quella di sfruttare al massimo i tempi morti del processore, tenendo conto che la CPU è la risorsa più importante della nostra macchina e quindi è necessario tenere tale risorsa sempre attiva.

Ora facciamo un esempio per comprendere al meglio: se eseguiamo un certo programma, P, quando egli deve eseguire un'istruzione I/O, interrompiamo l'esecuzione del programma P sul processore. Se non esistesse la multiprogrammazione, la CPU sarebbe ferma poiché il programma P sta eseguendo una istruzione I/O. Grazie alla multiprogrammazione, possiamo recuperare un altro programma, per esempio Q, il quale viene caricato sul processore e viene fatta partire l'esecuzione. Quando l'istruzione di I/O del processo P è stata completata, potremmo pensare di voler ripartire con l'esecuzione di

P oppure continuare quella del programma corrente Q. Questo intervallo di tempo è stato occupato dall'esecuzione di Q. Non è tempo sprecato ma è tempo durante il quale abbiamo avanzato con l'esecuzione di un altro processo.

Queste operazioni sono gestite dal sistema operativo. Egli deve controllare le risorse della macchina, dove in questo caso le risorse interessate sono il processore, la memoria centrale e la memoria secondaria. Il sistema operativo effettua le operazioni, in particolare effettua **cambi di contesto** (content switch). I cambi di contesto fanno riferimento alla fatto che sul processore si passa dell'esecuzione di un processo P all'esecuzione di un altro processo Q.

Una volta terminata l'operazione dei I/O per il processo P, sarà il sistema operativo stesso a scegliere se continuare con l'esecuzione di Q oppure ritornare con l'esecuzione del primo processo P.

Il motivo principale per cui si agisce in questo modo è perché il processore è più veloce rispetto alle periferiche di I/O. Tutte le istruzioni che coinvolgono le altre parti periferiche sono, dal punto di vista del processore, una perdita di tempo. Ci sarebbero momenti in cui il processore rimarrebbe fermo e, grazie alla multiprogrammazione, si cercano di riempire questi per eseguire altre attività.

1.4 Time-Sharing Systems-Interactive Computing

I sistemi Time-Sharing costituiscono un passo ulteriore nell'evoluzione dei sistemi operativi. Il concetto del Time-Sharing nella storia dei sistemi operativi è intervenuto quando sono maturate esigenze di usi interattivi della macchina, per esempio sistemi transazionali che sono tipici di molti sistemi gestionali, sistemi bancari. In questi casi c'è un utente che effettua delle interazioni con il sistema stesso. Queste esigenze di sistemi interattivi transazionali sono arrivate quando ci sono stati degli sviluppi tecnologici che le hanno rese possibili. In questi sistemi transazionali, l'obiettivo del sistema operativo cambia.

In un sistema con la multiprogrammazione pura, l'obiettivo principale è l'efficienza della macchina, quindi l'uso ottimale delle risorse della macchina. In un sistema transazionale interattivo, l'obbiettivo principale è la **reattività**. L'utente che effettua delle interazioni con la macchina, una volta eseguito un comando, ci si aspetta di vedere una reazione da parte della macchina stessa in tempi rapidi. Ovviamente questa reattività presuppone un'ottima efficienza da parte del sistema operativo, in quanto deve sempre gestire le risorse della macchina in maniera appropriata.

Il sistema operativo deve imporre delle soglie massimali di permanenza di un processo sulla CPU. Quando un processo viene eseguito, si individua un **quanto di tempo**, cioè il tempo massimale concesso a questo processo per eseguire in maniera continuativa. Se abbiamo un processo P e un quanto di tempo pari a *5*, significa che dopo *5* unità di tempo il processo P viene interrotto.

1.4.1 Interrupt

Sappiamo che il processore sia molto più veloce rispetto alle altre componenti della macchina, perciò abbiamo introdotto il concetto di multiprogrammazione come conseguenza di questo fatto. Un processo che a un certo punto della propria esecuzione si trova a eseguire un'istruzione che fa riferimento alle periferiche di I/O, è un processo che si trova, prima di poter andare avanti nella propria esecuzione, a dover attendere la terminazione di questa istruzione I/O. Dal punto di vista del processore è un periodo di tempo molto lungo. Essendo un periodo molto lungo e siccome il processore è la risorsa principale della macchina, il sistema operativo forza un content switch, dove il processo attualmente in esecuzione perde il controllo del processore e sul processore stesso viene lanciato un nuovo processo che potrà eseguire delle proprie istruzioni. Questo è il concetto base della multiprogrammazione.

Il problema che sorge in questo momento è il seguente: come ci rendiamo conto che una operazione I/O di un processo è terminata? Come il sistema operativo può sapere questo?

Una volta il sistema operativo faceva del **polling**, ovvero di tanto in tanto chiedeva all'unità che sta svolgendo l'istruzione di I/O, se tale istruzione fosse terminata. Le unità di tempo di interrogazione da parte del processore all'unità della macchina che svolge le istruzione I/O non devono essere troppo lunghe, ma non possono essere neanche troppo brevi. Perché un check di questo genere da parte del sistema operativo significa interrompere il sistema operativo, che magari in quel momento sta eseguendo un'altra attività di un altro processo. Perciò gli intervalli di tempo che il sistema operativo aspetta per effettuare questi check non possono essere troppo brevi perché si rischia di fare troppi cambi di contesto, che servono per effettuare i check, e la maggior parte dei check potrebbero presentare un risultato negativo. Quindi il problema principale del polling è capire quando fare il check, quando è il momento più opportuno per farlo.

Nei sistemi operativi delle macchine odierne lo schema è completamente diverso. La verifica sulla terminazione delle operazioni di I/O non avviene tramite il polling o check da parte del sistema operativo, ma ci si affida a un meccanismo di **interrupt.** Il meccanismo di interrupt consiste nel permettere alla periferica stessa di inviare un segnale al processore relativamente alla terminazione dell'operazione che gli era stata impartita.

1.5 Parallel System

Fino ad ora abbiamo parlato di macchine con solamente una CPU, ma le macchine moderne hanno, in generale, più di un processore. Per esempio, un processore dedicato all'attività grafica è del tutto normale nelle macchine odierne. All'interno dei processori stessi c'è parallelismo.

Un singolo processore è diviso in componenti, chiamati **core**. Un core è a tutti gli effetti un processore. Possiamo dire, quindi, che un singolo processore è diviso in sotto unità che sono a loro volta dei processori. La struttura della CPU si trova replicata all'interno di ogni singolo core. Ogni core ha una propria ALU e una propria control unit. All'interno dei core è presente anche una piccola memoria cache, la quale è locale per il singolo core. Nel caso in cui avessimo 4 core, la cache locale del core 3 non è utilizzabile dal core 2.

È presente anche un'altra memoria cache più grande, di livello più esterno ma sempre all'interno della CPU, che è condivisa da ogni singolo core. Per finire, si aggiunge la memoria cache esterna al processore.

Perché per il processore è stata introdotta questa architettura a multi-core? Perché si cerca di velocizzare e rendere una macchina più efficiente. Il modo per farlo è il parallelismo; è molto vantaggioso fare all'interno di un processore più cose contemporaneamente. Il fatto di avere diversi core è molto significativo da questo punto di vista.

1.5.1 Pipelining

Per comprendere meglio questo concetto, facciamo riferimento alla Figura 2.4 del processore. La control unit può essere pensata come tre sotto unità principali:

1. **Fetch Unit:** predisposta al recupero della prossima istruzione che deve essere eseguita;

2. **Decode Unit:** interpreta e comprende come un'istruzione deve essere trattata;

3. **Execute Unit:** impartisce gli ordini precisi per l'esecuzione dell'istruzione stessa. Tipicamente da delle direttive all'ALU per effettuare delle operazioni.

Queste tre sotto unità ci portano facilmente al discorso del pipelining.

Un'istruzione, per essere eseguita nella sua interezza, passa attraverso tutti i tre stadi della control unit. Inizialmente l'istruzione dev'essere recuperata tramite la *fetch unit,* poi decodificata e interpretata attraverso la *decode unit* e, infine, viene eseguita attraverso l'*execute unit.* L'idea del **pipelining** è quella di sfruttare queste tre fasi per introdurre del parallelismo. Per assimilare meglio questo concetto, facciamo un esempio:

Supponiamo di avere quattro istruzioni, I_1, I_2, I_3, I_4. Inizialmente, la prima istruzione che parte è la I_1, di cui se ne occupa la fetch unit. Una volta che la fetch unit ha eseguito il proprio compito con l'istruzione I_1, questa istruzione diventa il soggetto della decode unit, lasciando la fecth unit libera. A questo punto, possiamo pensare che la fetch unit cominci già il lavoro sull'istruzione successava I_2.

Quando il lavoro della fecth e decode unit sulle istruzioni I_1 e I_2 è stato completato, l'istruzione I_1 passa alla execute unit e l'istruzione I_2 diventa il soggetto della decode

unit, lasciando la fetch unit libera di occuparsi di una nuova istruzione, I_3. In questo momento tutte tre le sotto unità sono attive, sono tutte in esecuzione.

Quando l'istruzione I_1 sarà terminata completamente, l'istruzione I_2 potrà essere eseguita nella execute unit, l'istruzione I_3 diventa il soggetto della decode unit e possiamo iniziare a eseguire l'istruzione I_4. Cosi facendo, tutte tre le sotto unità svolgono un'istruzione.

Dato un lavoro da svolgere, il quale richiede una certa unità di tempo, l'idea del **pipeling** è quella di suddividere questa unità di tempo in sotto fasi. Lo scopo principale è quello di voler assegnare a queste sotto fasi delle componenti diverse, in modo da ridurre il tempo complessivo di esecuzione di ciascuna sotto fase.

Se prendiamo un'unità di tempo lunga 6 e la suddividiamo in 3 sotto fasi, ogni sotto fase ha durata 2. Se si avessero n oggetti, senza pipelining, a ogni oggetto servirebbero 6 unità di tempo e il costo complessivo sarebbe $6 \cdot n$ unità di tempo. Avendo pipelining, però, ogni 2 intervalli di tempo, si ha un oggetto la cui lavorazione viene terminata, quindi il costo complessivo è $2 \cdot n$.

In modo più generale, se abbiamo un intervallo di tempo, t, con il pipelining riusciremo a trattare $\frac{t}{2}$ oggetti, piuttosto che $\frac{t}{6}$. Questa grandezza di chiama **throughput.**

1.5.2 Superscalarità

La superscalarità è uno strumento che viene usato per risolvere situazioni in cui nel pipelining ci siano delle fasi che hanno necessità di un tempo maggiore rispetto ad altre.

Facendo riferimento all'esempio visto in precedenza per il pipelining, la sotto unità più lenta è la execute unit, per tanto è ragionevole pensare a un raffinamento dello schema precedente, in modo da avere una fetch unit iniziale che passa i dati alla decode unit e poi, piuttosto che avere il passaggio a un'unica execute unit, si può pensare di avere diverse execute unit, con un **buffer di contenimento**. La funzione del buffer di contenimento è quella di ricevere le istruzioni che arrivano dalla decode unit e smistarle alla execute unit che è libera in quel momento.

Questo schema potrebbe presentare dei problemi di correttezza. Bisogna effettuare un check sulle dipendenze della varie istruzioni, in modo da effettuare l'operazione di parallelismo nel modo più corretto possibile.

Capitolo 2

Computer-System Structures

La figura 2.1 mostra schematicamente l'hardware di una macchina. Abbiamo già parlato più volte della **CPU (Central Processing Unit)** che è il cervello della macchina stessa. La **memory** è la memoria centrale, chiamata anche **RAM (Random Access Memory)**. Le altre parti che compongono la Figura 2.1 sono periferiche I/O. La CPU e la RAM sono, invece, delle strutture particolari che dialogano fra di loro in maniera continua. Sono veramente fondamentali nell'esecuzione di una qualunque attività.

Il **System bus** collega tutte le unità. Il system bus sono dei fili microscopici in cui passano i comandi e i dati. Sono molto importanti perché è attraverso il *bus* che passano le informazioni, ed è molto importante anche il numero di fili. Quando si parta di architettura a 32 o 64 bit, si parla, in realtà, del numero di fili che corrispondo al numero di bit che possono transitare contemporaneamente sul bus. Associato a tutte le unità, tranne il processore, c'è un **controller.**

Il controller è un piccolo circuito elettronico che controlla l'hardware puro. Egli contiene alcuni *registri* che vengono utilizzati dal processore per inviare istruzioni a quella specifica periferica. Il processore è, ovviamente, a conoscenza di questi registri, i quali hanno degli indirizzi particolari dal punto di vista della CPU, grazie ai quali egli sa indirizzarli e utilizzarli.

Come dicevamo in precedenza, il sistema operativo deve nascondere l'hardware ai livelli superiori. Tali livelli non vendono direttamente questi controller, ma vedono la parte del sistema operativo che dialoga con questi controller. Questa parte che dialoga viene chiamata **driver**.

2.1 von Neumann Computer

La Figura 2.2 mostra un'altra visione dell'architettura di un computer. Le architetture dei computer moderni vengono spesso chiamate *"architetture von Neumann"*, facendo riferimento al fisico e matematico ungherese John von Neumann. Egli contribuì

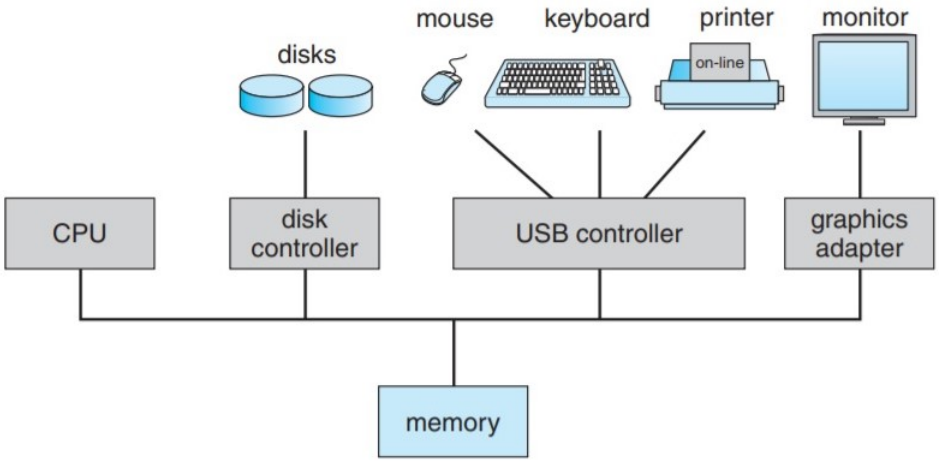

Figura 2.1: A modern computer system

alla creazione di uno dei primi computer che vennero costruito verso la fine dalle seconda guerra mondiale *(eniac)*.

Facendo riferimo alla Figura 2.2, l'**Arithmetic-Logical Unit** e la **control unit** è la CPU. La **primary memory** è la RAM e i **device** sono tutte le periferiche I/O. Abbiamo anche il bus, il quale viene suddiviso in una parte **data** e **address**.

Ora facciamo riferimento alla struttura del processore poiché è sempre la parte più importante della macchina. È la risorsa più importante che il sistema deve gestire. In una CPU possiamo distinguere due parti importanti:

1. Arithmetic-Logical Unit.
2. Control Unit.

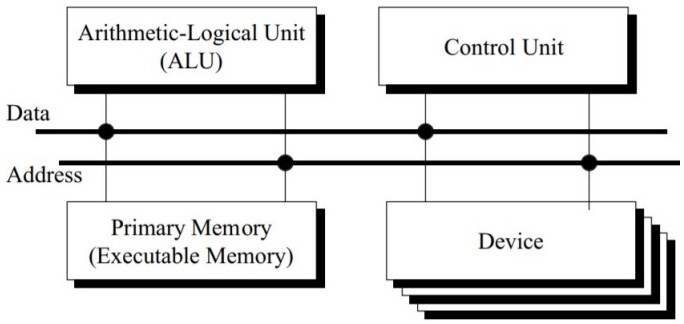

Figura 2.2: Von Neumann Computer

2.1.1 Arithmetic-Logical Unit (ALU)

L'*Arithmetic-Logical Unit* è l'unità, chiamata anche **ALU**, che effettua i calcoli di tipo aritmetica (+, -, x, :) e operazioni logice (and, or, not). Mentre la Control Unit è il vero e proprio cervello del processore. Possiamo dunque dire che è il cervello del cervello, avendo chiamato la CPU il cervello della macchina. La control unit è l'unità che si preoccupa di capire qual è la prossima istruzione che deve essere eseguita; cerca di comprendere cosa significa eseguire un'istruzione e che impartisce gli ordini per eseguire quelle istruzioni che vengono passate. Ovviamente le due parti interagiscono tra di loro.

2.1.2 Primary Memory (RAM)

Come detto in precedenza, il processore dialoga con la memoria centrale. La *RAM* è suddivisa in celle che hanno una numerazione crescente, da 0 a n, dove n è l'indirizzo massimo per la nostra memoria. Ognuna di queste celle è un byte, quindi la memoria centrale è una sequenza di byte che sono indirizzabili dal processore attraverso opportuni registri del processore stesso.

Possiamo vedere la Figura, 2.3 come eseepio. In questa caso abbiamo tre registri. Il **MAR** è il *memory address registrer*. Il **MDR** è il **memory data registrer** e il **command registrer**. Questi sono registri all'interno del processore e vengono utilizzati per dialogare con la memoria centrale. In una configurazione di questo genere, cosa succede? Succede che nella cella 1234, numero contenuto nel registro MAR, si va a scrivere il numero contenuto nel MAD. Si effettua un'operazione di scrittura perché è quello che viene indicato nel registro comandi. Oltre alle procedure di *write*, ci possono anche essere quelle di *read*.

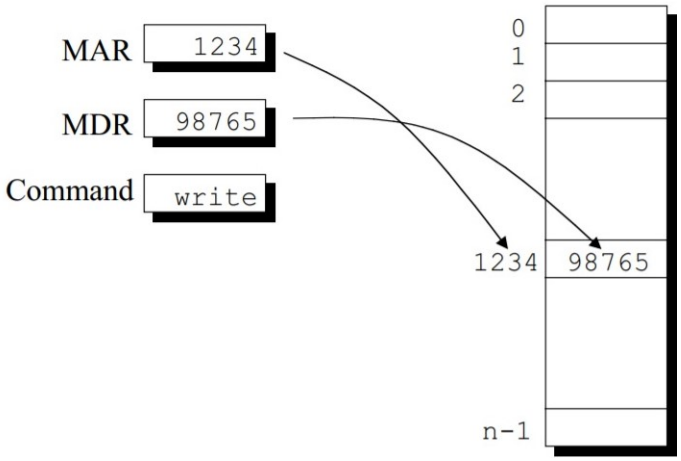

Figura 2.3: Memory Unit - RAM

2.1.3 Control Unit

La *Control Unit* è il cervello del processore. Ha il compito di decidere qual è la prossima istruzione che deve essere eseguita e deve impartire gli ordini per l'esecuzione stessa. Anche nella control unit esistono dei registri. Il registro più importante è **program counter (PC)**. Il program counter è un registro nel quale è registrato il numero della cella di memoria che contiene l'istruzione che deve essere eseguita. Esiste un altro registro che si chiama **instruction register (IR)**. Egli contiene l'istruzione che deve essere eseguita.

La control unit, nella propria attività, si compone di fasi. La prima fase si chiama *fetch*, nella viene recuperata l'istruzione che deve essere eseguita. Nella seconda fase c'è una interpretazione, bisogna comprendere queste istruzioni; cioè bisogna capire che tipo di istruzione è, che cosa comporta e che cosa bisogna fare. L'ultima fase è quella di esecuzione vera e proprio. Si impartiscono gli ordini per l'esecuzione di quella istruzione.

Queste tre fasi all'interno della control unit sono rappresentate da tre parti diverse. Esiste una strutturazione che rispecchia queste fasi di lavorazione.

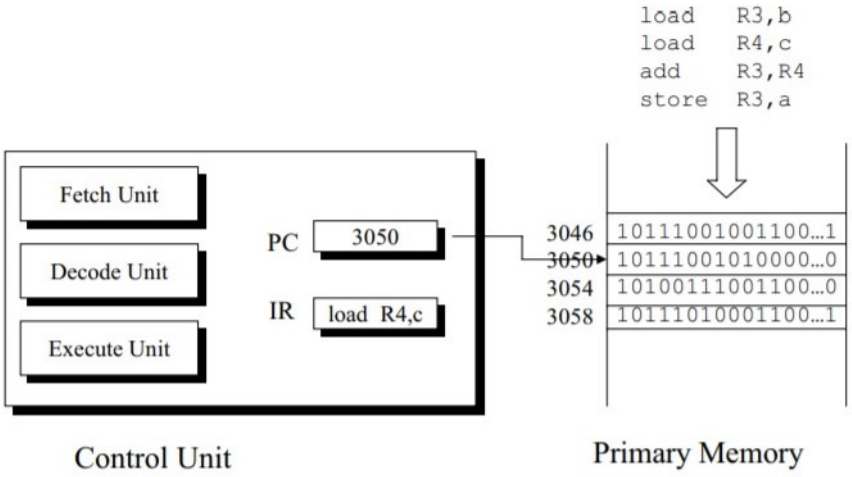

Figura 2.4: Control Unit

2.2 Storage Structure

All'interno di una macchina possiamo trovare diversi tipi di memorie. Le memorie all'interno di una macchina si distinguono per la loro dimensione, velocità e per il fatto di essere permanenti oppure volatili.

2.2.1 Registers

Facendo riferimento alla Figura 2.5, vediamo che la forma di memoria più veloce è rappresentata dai registri che si trovano all'interno del processore. Purtroppo i registri del processore sono disponibili in numero limitato. I registri sono nell'ordine dei KB.

2.2.2 Cache

La memoria cache è una forma di memoria che è intermedia tra i registri del processore e la memoria centrale. Le memorie cache sono nell'ordine dei MB, circa 64 MB.

La cache è una memoria molto veloce, se l'accesso ai registri della CPU avviene nell'ordine 1 ns, l'accesso alla memoria cache potrebbe prendere 2/4 ns. Mentre, per esempio, l'accesso alla RAM potrebbe prendere 10/20 ns. C'è una differenza abbastanza significativa.

Perché esiste una memoria cache? Una memoria cache esiste per memorizzare quelle informazioni che sono usate più frequentemente dal processore. Se ci sono dei dati e delle istruzioni che sono usate di frequente dal processore, esse possono essere copiate all'interno della memoria cache in modo tale che, se devono essere usati una seconda volta, possono essere più facilmente ritrovabili.

Quando il processero lavora e ha bisogno di una certa istruzione, controlla prima la presenza di questa istruzione all'interno della memoria cache. Se questo ha successo, allora siamo apposto. Altrimenti, bisogna cercare l'informazione nel livello di memoria successivo, ovvero la memoria centrale.

2.2.2.1 Principio di località dei processi

Le informazioni e i dati che un certo processo in esecuzione usa in un certo momento, sono molto simili a quelli che userà nell'immediato futuro o che ha usato nell'immediato passato.

L'insieme dei dati e delle istruzioni che un processo usa in un certo momento cambiano con il passare del tempo. Questo principio permette di utilizzare le memorie cache in maniera efficacie. È possibile suddividerlo in punti:

1. Principio di località temporale: insieme di istruzioni e dati in uso per un certo processo che cambia lentamente nel tempo;

2. Principio di località spaziale: le istruzioni e i dati che uso in questo momento, sono in posizione contigua a quelli che userò nell'immediato futuro o che ho usato nell'immediato passato.

Bisogna specificare che esistono diversi livelli di cache.

La cache principale è quella esterna al processore, pur se nelle immediate vicinanze del processore stesso. Esistono spesso anche delle cache interne, ovvero all'interno della

CPU stessa. All'interno del circuito integrato del processore, un po' di transistor vengono utilizzati per creare dei altri livelli di cache. La gestione di tutti questi livelli di memoria è affidata al sistema operativo.

2.2.2.2 Sistema opearitvo e cache

Il sistema operativo deve conoscere queste cache ed è lui che decide cosa portare in cache e quando portarla. Questa situazione, tuttavia, porta a delle domande che non sono affatto banali. La prima domanda che il sistema operativo si pone è quando bisogna portare un certo oggetto all'interno della memoria cache. Successivamente, deve decidere su quale livello di cache portare i dati. Il livello, ovviamente, va in base a quante cache sono presenti.

Le cache sono di dimensioni nettamente inferiori a quelle della memoria centrale, e estremamente più piccole rispetto alla memoria secondaria, quindi si riempono più in fretta. Potrebbe darsi che il sistema operativo vorrebbe portare un dato in cache, ma la cache è già piena, perciò non c'è spazio per contenere il nuovo dato. Dunque, è necessario fare spazio al nuovo dato eliminando qualcosa già presente. Quindi, come terza domanda, è necessario domandarsi quali dati eliminare all'interno della cache.

2.2.2.3 Consistenza dei dati

Il problema della consistenza dei dati si verifica quando all'interno della macchina sono presenti più processori. Utilizzerò un esempio per spiegare questo concetto.

Inizialmente un processo che deve essere eseguito si potrebbe trovare in memoria secondari poi, quando l'ordine dell'esecuzione viene impartito, egli viene caricato sulla memoria centrale e durante l'esecuzione può darsi che alcune parti del processo stesso finiscano in una qualche memoria cache. Dunque, possiamo notare ora che il dato si trova in tre memorie.

Durante l'esecuzione del processo, un dato potrebbe essere modificato. Se inizialmente avevamo un dato D, esso potrebbe cambiare e diventare D^1. Il valore di D potrebbe essere, per esempio, il valore di una variabile, la quale durante l'esecuzione è stata modificata e diventa D^1

A questo punto, si hanno due valori diversi per il dato D; abbiamo il D iniziale e il D^1. Bisogna fare attenzione che quando l'esecuzione del nostro processo sarà terminata, è importante che le repliche del dato siano consistenti tra di loro, cioè abbiamo lo stesso valore. Durante l'esecuzione del processo, questa consistenza non è garantita. Si potrebbero avere valori del dato diversi su livelli diversi di memoria.

Il problema della consistenza dei dati, è reso più complicato dalla presenza di più processori. Facendo sempre riferimento all'esempio precedente, all'interno della stessa macchina potrebbe esserci un altro processore, al quale potrebbe esserci collegata un'altra memoria cache.

Il problema potrebbe sorgere quando il processore 2 viene utilizzato per eseguire un processo,il quale ha bisogno dello stesso dato D utilizzato dal processore 1. Quando il processore esegue un processo e ha bisogno di un certo dato, questo dato viene recuperato dalla memoria centrale, se non è presente già in cache. Per default il dato viene letto dalla memoria centrale, ma ciò potrebbe essere un rischio. Il rischio è che venga letto un valore di D che non è più il valore corrente. Potrebbe essere letto un vecchio valore.

L'esecuzione del processo, sul processore 2, potrebbe continuare lo stesso modificando il dato D, facendolo diventare D^2, il quale è diverso da D^1. A questo punto ci troviamo in una situazione di inconsistenza, cioè si hanno due processi che hanno una visione del dato che è diversa e non concordante ($D^1 \neq D^2$).

2.2.2.4 Main Memory

La memoria centrale, oggi giorno, è disponibile in quantità dell'ordine del MB/GB.

2.2.2.5 Second Memory

Le altre forme di memorie che vediamo nella Figura 2.5, sono tutte forme di memoria secondarie. Esse possono arrivare all'ordine del TB. Ci sono diversi tipi di memoria secondaria. Oggi giorno, quella più diffusa è l'SSD - Solid State Drive - che ha col tempo soppiantato l'HDD - Hard Disk Drive - i quali sono molto più lenti rispetto alle SSD.

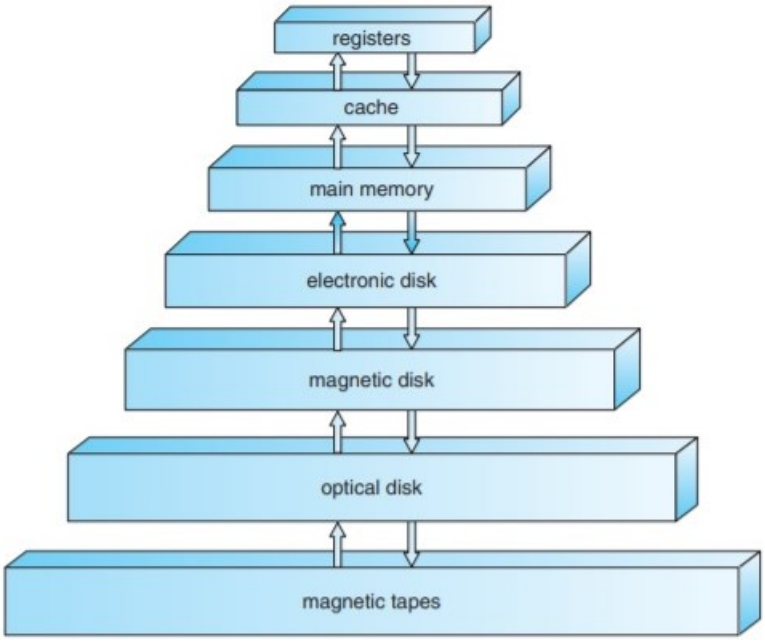

Figura 2.5: Storage-Divice Hierarchy

2.3 Hardware Protection

In questa sezione parleremo di alcune caratteristiche dell'hardware che sono presenti in una macchina per venire incontro alle esigenze del sistema operativo. Esempi in cui il sistema operativo influenza la struttura e lo sviluppo dell'hardware delle macchine.

Iniziamo a introdurre concetti che hanno a che fare con la protezione dell'hardware durante l'esecuzione di un job da parte di una macchina.

- **Dual-Mode Operation:** Le Dual-Mode Operation possiamo immaginarle come un semplice bit che si trova all'interno del processore con lo scopo di differenziare il caso in cui l'istruzione attualmente eseguita dalla CPU sia un'istruzione di un processo, come per esempio di un programma utente, oppure un'istruzione del sistema operativo. Differenziamo quindi due mode:

  1. **User mode**: si riferisce alla situazione in cui stiamo eseguendo istruzioni di un processo, non del sistema operativo.
  2. **Monitor mode**: si riferisce a una situazione in cui il processore sta eseguendo operazioni per conto del sistema operativo.

  Questo concetto sembra al quanto banale, ma bisogna dire che ci sono istruzioni che solo il sistema operativo è in grado di eseguire. Per esempio, le periferiche I/O possono essere usate soltanto dal sistema operativo. Non vogliamo che il programma utente abbia accesso diretto alle periferiche I/O perché potrebbe fare casini. È meglio che il sistema operativo garantisce questa esclusività tramite le system call. Se c'è un programma utente che vuole accedere alle periferiche I/O, sarà il sistema operativo che permetterà l'accesso al programma utente attraverso le system call.

  Le istruzioni che possono essere eseguite soltanto dal sistema operativo prendono il nome di **istruzioni privilegiate.**

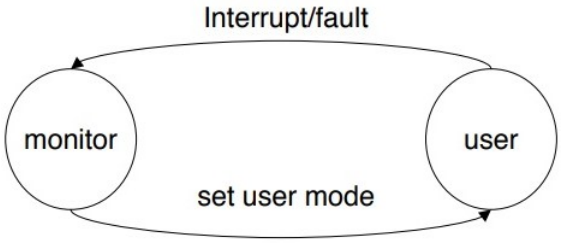

Figura 2.6: Dual-Mode Operation

Come possiamo vedere in Figure 2.6, se lo user mode prova a eseguire una istruzione privilegiata, scatterà una interrupt che farà intervenire il sistema operativo

poiché siamo in una situazione in cui il processo utente cerca di fare qualcosa di potenzialmente pericoloso.

- **Memory Protection:** Questa forma di protezione deve essere assicurata dal sistema operativo. Facendo riferimento alla Figura 2.7, possiamo vedere che la memoria centrale è divisa fra i processi in esecuzione. La prima parte della memoria, *monitor*, è occupata dal sistema operativo. Le altre zone di memoria sono divise fra diverse attività, in questo caso 4 processi utenti. Ogni processo ha una zona di memoria associata ad esso, e questa zona di memoria è individuata dal **base register**, il quale indica l'indirizzo della cella di memoria che il job può utilizzare e un **limit register** che indica la dimensione del job stesso. In questo caso il job 2 occuperà 120900 indirizzi consecutivi di memoria centrale.

  Che cosa significa garantire che un job non interferisca nelle zone di memoria di altri processi? Significa che quando la CPU produce, durante l'esecuzione di job 2 per esempio, un indirizzo di memoria, dovremmo verificare che questo indirizzo prodotto rappresenti un numero che si trova tra la prima e l'ultima cella di memoria allocata a job 2. Se questo controllo dovesse essere fatto esclusivamente dal sistema operativo, quindi un controllo delegato completamente al software, sarebbe piuttosto pesante, in quanto tutte le volte che bisogna accedere alla memoria centrale, bisognerebbe fare due controlli. Controllare che l'indirizzo creatp rappresenti un numero uguale o più grande del base register e inferiore al limit register.

  Quello che succede in realtà è che quando un indirizzo viene prodotto dal processore, questo indirizzo viene confrontato con il l'indirizzo base e anche con l'indirizzo limit. Prima che l'esecuzione di job 2 abbia inizio, il sistema operativo si preoccupa di andare a scrivere su questi registri i valori che sono l'indirizzo della prima cella e la dimensione di job 2. Soltanto se questi due controlli hanno esito positivo, si potrà accedere alla memoria centrale.

- **CPU Protection:** Un altro esempio di protezione che viene ottenuto tramite l'intervento dell'hardware è la CPU protection. Per comprendere questo concetto facciamo riferimento a una situazione di time-sharing. Il time-sharing è una evoluzione della multiprogrammazione, nella quale c'è interattività con la macchina. Quando un processo utente deve interagire con la macchina ci aspettiamo una certa reazione da parte della macchina stessa molto contenuti, la macchina stessa reagisce molto velocemente.

  Un processo non può eseguire per periodo di tempo arbitrario ma nel momento in cui viene fatto partire questo processo gli viene assegnato un quanto di tempo, cioè un periodo massimale di utilizzo del processore, al termine del quale viene fatto un cambio di contesto. Viene fatto ciò per evitare una situazione in cui sia un processo che, per un errore di programmazione o intenzionalmente, rimane sulla CPU per un periodo troppo lungo.

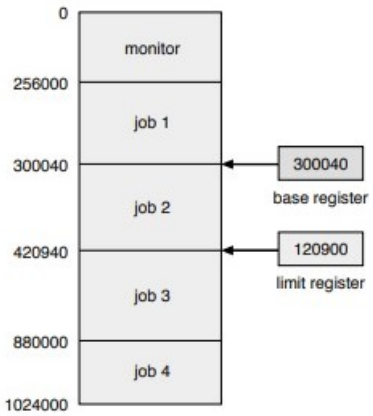

Figura 2.7: Memory Protection

Il sistema operativo, quando fa partire un certo processo, sceglie il quanto di tempo massimale per il job che sta per partire. Successivamente è sempre compito del sistema operativo assicurarsi che il processo rimanga in esecuzione per un periodo che è al più quel quanto di tempo che era stato scelto all'inizio.

Come facciamo il controllo? Come ci assicuriamo che effettivamente questo processo rimanga in esecuzione al più per il quanto di tempo prestabilito? Se questi controlli fossero delegati interamente al software, il sistema operativo utilizzerebbe troppo tempo. Perciò viene delegato l'hardware stesso a controllare che il tempo di esecuzione sia inferiore o uguale al quanto di tempo prestabilito dal sistema operativo. A livello di hardware interviene un timer che viene decrementato, ad ogni click di un orologio presente all'interno dell'hardware stesso. Il valore finale del timer rappresenta il quanto di tempo. Quando il sistema operativo fa partire un processo, dovrà settare il valore del registro timer al valore del quanto di tempo. Quando il timer raggiunge il valore 0, il processo deve essere interrotto; parte un interrupt che fa intervenire il sistema operativo, il quale dovrà effettuare un cambio di contesto.

Capitolo 3

Processes

3.1 Process Concept

Per comprendere al meglio il concetto di processo, è necessario introdurre una distinzione tra il termine concetto e il termine programma.

Algorithm 1 Programma

```
begin
Int var x,y := 0;
x := x+y;
y := y * y;
print y;
end
```

Questo pezzo di codice all'interno del quale vengono effettuate delle operazione, viene chiamato **programma**. Un programma sono linee di codice. È un'entità passiva. Quando questo programma viene lanciato diventa un'entità attiva e, così facendo, diventa un **processo**. Quindi, un processo è un programma in esecuzione. Un processo per poter essere eseguito necessita di una parte di:

- codice: composto dalle istruzioni che il processo stesso deve eseguire;
- dati: dati globali che devono essere aggiornati durante l'esecuzione del processo;
- stack: viene utilizzato dal processo per gestire delle determinate procedure all'intero del codice. Serve per gestire la dinamicità del processo.

Un altro concetto molto importante per quanto riguarda un processo è il punto nel codice in cui sono arrivato durante l'esecuzione del processo. Questo concetto viene chiamato **program counter.** Per definire lo stato di un processo in un certo momento è necessario sapere qual è il program counter.

3.1.1 Process State

Un processo durante il proprio tempo di vita si può trovare in diversi stati. In particolare abbiamo 5 stati:

1. **New:** stato in cui avviene la creazione di un nuovo processo. Il sistema operativo si occuperà del processo appena creato. Egli creerà delle strutture dati apposite per il nuovo processo.

2. **Running:** stato di un processo che è attualmente in esecuzione. Nella situazione di una macchina con un solo processore, c'è un solo processo running, quindi un solo processo potrà essere in esecuzione. Altri possibili processi devono aspettare il loro turno.

3. **Waiting:** stato di un processo che sta aspettando qualche evento. Processo che non è in grado di essere eseguito perché ha bisogno che accada qualcosa. Un caso tipico è quello di un processo che sta aspettando il completamento di una istruzione I/O.

4. **Ready:** stato di un processo che è pronto per essere eseguito ma non può essere fisicamente eseguito perché, in quel momento determinato momento, il processore è utilizzato da un altro processo.Nella situazione in cui ci sono più processi che processori, per esempio una macchina con un solo processore e quindi un solo processo running, gli altri processi rimarranno in attesa finché il processore non si libera.

   Quando il processore si libera, il sistema operativo deve fare una scelta sui processi che sono in stato ready. Dovrò selezionare tra i processi ready il prossimo che deve essere eseguito.

5. **Terminated:** stato di un processo che ha terminato la propria esecuzione. È il complementare rispetto allo stato new. Quando un processo è terminato, il sistema operativo interviene per liberare la zona di memoria che era stata disegnata per quel specifico processo. Vengono eliminate le strutture dati che facevano riferimento al processo terminato.

Dalla Figura 3.1 possiamo notare gli stati sopra elencati. Notiamo che non sono possibili transizioni arbitrarie tra gli stati. Non abbiamo tutte le possibili transizioni tra questi stati, ma soltanto alcune di queste.

Dallo stato **new** è possibile passare soltanto allo stato **ready**. Un processo quando viene creato, non viene mandato direttamente in esecuzione ma viene messo in stato ready, quindi pronto per l'esecuzione.

Dallo stato **ready** l'unica transizione possibile è verso lo stato **running**. Un processo ready ha tutto ciò che gli serve per essere eseguito e quindi aspetta soltanto di ricevere il controllo del processore.

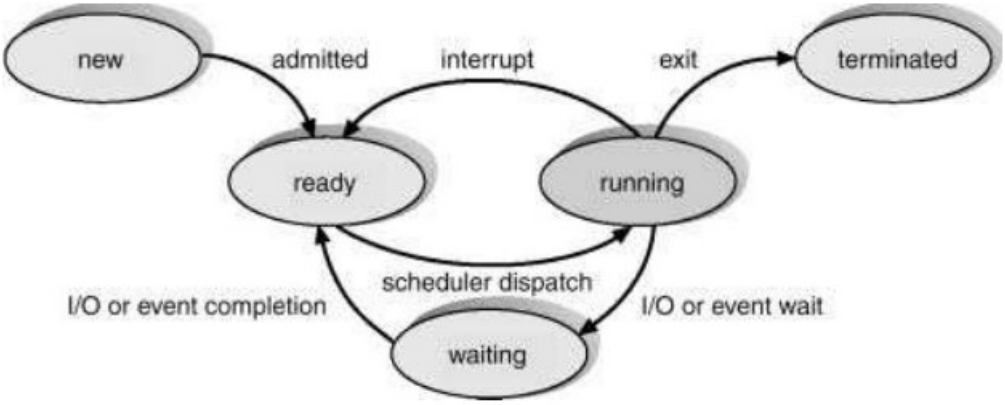

Figura 3.1: Diagram of Process State

Dallo stato **running** ci sono tre transizioni uscenti, verso **waiting**, **ready** e **terminated**. Quando un processo è running, potrebbe terminare la sua esecuzione e passare allo stato terminated. Potrebbe anche ritornare allo stato ready. Nel caso in cui arrivi una interrupt la quale comporta all'intervento del il sistema operativo, il processo in esecuzione viene messo da parte, in stato ready. Da running si può passare anche a waiting. Il caso tipico è l'esecuzione di un'istruzione I/O, quindi un'istruzione molto lenta dal punto di vista del processore e il sistema operativo interviene per far eseguire alla CPU un altro processo.

Dallo stato **waiting** si passa solamente allo stato ready. Come nel caso di new, un processo che ha terminato il periodo di waiting non passa direttamente all'esecuzione, ma passa allo stato ready. Perciò si entra in competizione con altri processi per l'acquisizione del processore.

3.1.2 Process Control Block (PCB)

Il Process Control Block è una struttura tabellare, Figura 3.2, che traccia delle informazioni principali associate a un processo. Il sistema operativo deve manipolare il processo, per tanto avere un modo per sapere quali sono le caratteristiche principali e le risorse utilizzate da quel processo. Inoltre, il sistema operativo ha bisogno di una struttura attraverso la quale gestire i cambi di contesto.

Il Process Control Block è una funzione molto importante nel content switch. Il cambio di contesto è una fase in cui il sistema operativo fa passare sul processore un nuovo processo, togliendo dell'esecuzione il processo precedente. In Figura 3.3 possiamo vedere il funzionamento di un content switch. Inizialmente c'è un processo P_0 che è in esecuzione ma, a un certo punto, arriva in una situazione dove è necessario effettuare un content switch; il processo P_0 viene messo in stato waiting o ready. A questo punto

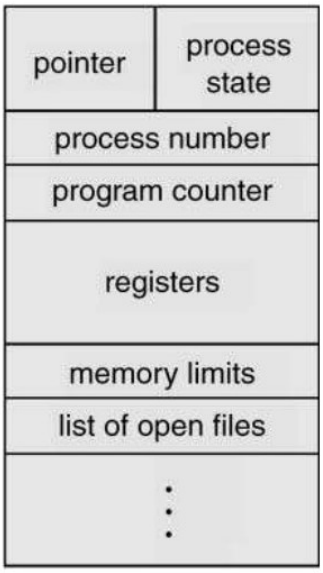

Figura 3.2: Process Control Block

interviene il sistema operativo, in quanto il content switch deve essere gestito da lui stesso.

Il sistema operativo interrompe il processo in questione, P_0, il quale non aveva terminato la propria esecuzione, quindi toglie P_0 dal processore con l'idea di farlo ripartire un momento successivo. Perchè questo sia possibile, occorre salvare lo stato del processo P_0. Per stato non facciamo riferimento alla Figura 3.1, ma al program counter e ai registri. Tutti questi dati vengono salvati nel process control block di ogni processo, PCB_0.

Arrivati a questo punto il processore è libero. Il sistema operativo ora sceglie un altro processo da mandare in esecuzione. In questo caso viene scelto il processo P_1, che supponiamo fosse un processo interrotto precedentemente. Anche se fosse la prima volta che esegue, all'interno del process control block relativo al processo P_1, ovvero PCB_1, bisogna inizializzare tutti i registri necessari al processore per permettere l'esecuzione di P_1. Se P_1 è un processo che era già stato eseguito precedentemente, ci sarà già salvato sul suo process control block, il program counter e tutti i registri rilevanti del processore. Questi registri vengono ripristinati e, al termine di tutto ciò, l'esecuzione di P_1 può riprendere.

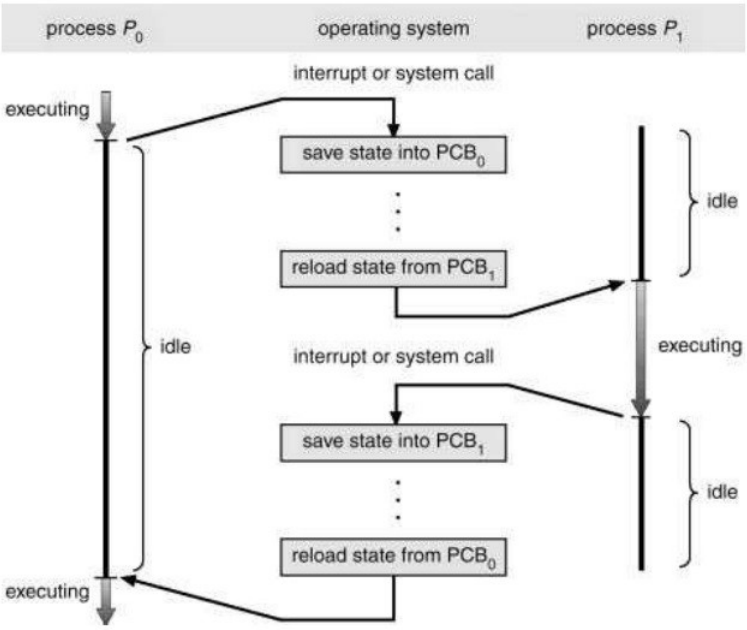

Figura 3.3: Content Switch

Capitolo 4

Threads

Per comprendere il concetto di threads pensiamo al concetto di processo e a cosa ne rappresenta. I processi sono delle entità che posseggono un insieme di risorse (codice, data, memoria, file) e rappresentano un flusso di controllo, cioè un'attività in esecuzione. Il processo utilizza le risorse per eseguire le proprie attività, è un'entità attiva.

Il passaggio da processo a threads, si può pensare come un passaggio nel quale vengono separati questi due elementi: l'elemento *risorse* e l'elemento *flusso di esecuzione*. Questi due elementi caratterizzano l'entità processi. Attraverso i threads all'interno di un processo, è possibile avere più flussi in esecuzione.

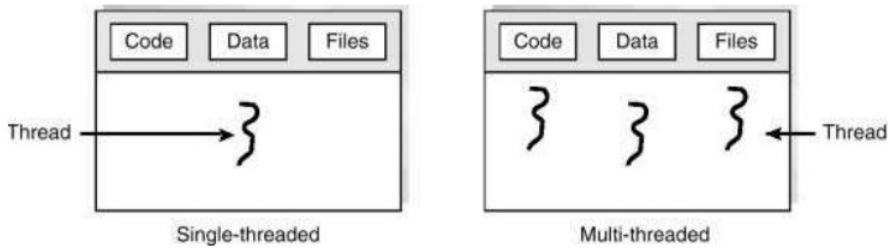

Figura 4.1: Single and Multithreaded Processes

Tramite la Figura 4.1, possiamo vedere la differenza tra la situazione classica, *Single-threaded*, e una situazione più evoluta, *Multi-threaded*.

Perché viene introdotto il concetto di threads? Quali sono i vantaggi e gli svantaggi? Un elemento fondamentale dei threads è che sono degli oggetti più leggeri rispetto ai processi. I processi sono delle strutture grandi, presentano tante risorse. Un processo ha un process control block che è una struttura tabellare piuttosto pesante. Effettuare un content switch a livello di processi, è un'operazione che prende abbastanza tempo. Il tempo che il sistema operativo deve impiegare per passare da un processo P a un processo Q, non è indifferente. La gestione dei processi dal punto di vista del sistema operativo è piuttosto pesante. Una qualunque operazione di un processo, per il sistema

operativo, è un'operazione che richiede una quantità di tempo non indifferente. Questo tempo rende meno efficiente la macchina.

I threads, invece, sono delle strutture più leggere. Sono dei flussi di esecuzione all'interno dello stesso processo. Passare da un threads all'altro significa rimanere sempre all'interno dello stesso processo. Le risorse sono le stesse, condividono lo stesso codice. Gli elementi relativi ai threads sono di meno rispetto agli elementi relativi ai processi, ciò li rende più leggeri.

Facciamo un esempio: pensiamo alla situazione in cui un utente debba scrivere un libro. Egli utilizzerà un text editor per la scrittura e il libro conterrà un migliaio di pagine. Supponiamo che sia alla fase finale della scrittura del libro e si accorge di voler aggiungere qualcosa all'inizio del libro. Se, dopo aver fatto questa modifica, volesse andare dalle pagine iniziai alle pagine finali del libro, il salto non potrebbe essere immediato perché il sistema (text editor) dovrebbe prima rimpaginare tutto il libro, quindi rifare tutta l'impaginatura di tutto il libro poiché si hanno introdotto dei nuovi paragrafi all'inizio e questo potrebbe aver portato a delle conseguenze su tutto il resto. Quindi, l'utente, dovrà attendere la terminazione dell'impaginatura prima di poter modificare qualcosa in fondo al libro. Questa procedura potrebbe prendere un po' di tempo. Se avessimo una struttura a processi, accaderebbe la situazione appena descritta. Con i thread, invece, questo problema di attesa scompare, o comunque viene minimizzato notevolmente. Possiamo pensare che il text editor sia un processo, cioè esiste un processo dietro il nostro text editor. All'interno del processo c'è la possibilità di avere diversi flussi di controllo, ovvero di avere diversi thread. A questo punto potremmo pensare che ci sia un thread che si occupa delle interazioni dirette con l'utente, T_1, il quale esegue i comandi che arrivano dall'utente attraverso tastiera. Ci potrebbe essere un thread separato che si occupa della impaginazione, T_2. Il T_1 sarebbe prioritario rispetto a T_2, poiché quando l'utente invia dei comandi su tastiera, questi devono essere ascoltati immediatamente dalla macchina. Non appena il T_1 è in pausa, quindi non ha un compito immediato da fare, il T_2 potrebbe intervenire e fare il proprio lavoro di impaginazione. In questa situazione potrebbe anche esserci un terzo thread, T_3, che interviene quando gli altri due thread sono fermi, per fare un salvataggio su disco. Il terzo thread servirebbe per salvare lo stato corrente del nostro libro in modo tale che se ci dovesse essere un problema con la macchina, si ha comunque un backup del lavoro. Questi tre thread sono all'interno dello stesso processo e condividono le stesse risorse. Facendo riferimento al nostro esempio, la risorsa più importante è il libro.

4.1 Vantaggi e svantaggi

Il *vantaggio* principale di una struttura thread è:

- **Efficienza:** il fatto di poter strutturare un'applicazione come un insieme di thread cooperanti, permette di avere concorrenza e parallelismo. Attraverso il paralleli-

smo, si ha l'opportunità di avere molteplici attività che avanzano in modo parallelo. Si riescono a sfruttare al meglio le architetture moderne ed è possibile gestire più richieste utente per volta (responsiveness). Un altro aspetto importante per quanto riguarda l'efficienza è che se un thread si blocca, il processo non è bloccato.

Mentre lo *svantaggio* principale di una struttura thread è:

- **Sincronizzazione:** il fatto di avere dei thread che condividono la stessa risorsa, per esempio due thread, T_1 e T_2, che condividono e possono accedere entrambi a dei dati, porta a delle possibilità di interferenza tra thread che sono molto significative. Il passaggio dallo schema processi, dove l'esecuzione di un processo avviene in maniera sequenziale e c'è solo un flusso di controllo, al passaggio a uno schema thread, dove sono presenti più flussi di controllo è un passaggio molto importante.

  Per un utente leggere del codice che esegue in maniera sequenziale può essere molto facile, mentre quando si parla di codice che permette il parallelismo, in cui c'è condivisione dei dati, capire che cosa succede è molto complicato per la mente umana. Ciò potrebbe portare a del non determinismo. Il fatto di avere più thread che condividono gli stessi dati, porta a situazioni in cui si hanno delle difficoltà a capire quello che succederà con i dati in quanto è complicato comprendere quando e come i due thread useranno i dati. Se entrambi thread modificano i dati, questo può portare a del non determinismo, cioè può portare a una situazione finale in cui i dati sono diversi nei due casi.

Nei sistemi operativi moderni, i thread sono implementati a livelo di kernel. Vengono implementati nella parte più importante del sistema operativo, nel nucleo. Il sistema operativo fornisce delle system call per la gestione, creazione e terminazione dei thread; interviene anche per gestire il content swtich tra thread. In questo caso si parla di thread che sono gestiti a livello di *kernel*. Questo è standard nei sistemi moderni.

Tuttavia si trovano ancora dei sistemi operativi in cui i thread sono gestiti a livello *utente*. Il questo caso il sistema operativo si occupa della gestione dei processi e non "vede" i thread. I thread sono implementati al di fuori del nucleo del sistema operativo stesso. Il vantaggio principale di questo approccio è che quando si esegua un'operazione di gestione dei thread, essa viene gestita a livello utente, senza l'intervento del sistema operativo. Non è necessario effettuare un content switch durante il quale è bisogna far intervenire il sistema operativo per eseguire delle operazioni implementate dal sistema operativo, ma si rimane ai livelli superiori.

4.2 Java Threads

Java nasce come linguaggio sequenziale. I thread sono presenti a livello di linguaggio. Esistono due modi fondamentali per introdurre la concorrenza in Java (concorrenza a

livello di thread). Il modo principale per introdurre il concetto di thread è quello di **estendere** la classe predefinita thread.

```
1  class Worker1 extends Thread
2  {
3      public void run(){
4          System.out.println("I am a Worker Thread");
5      }
6  }
```

In questo codice possiamo vedere che la classe *Worker1* eredita, con extends, la classe predefinita *Thread*. Ereditare dalla classe **Thread** comporta all'obbligo di definire un metodo **run()** (si possono ereditare, ovviamente, anche altri metodi della classe ereditata). Il metodo **run()** verrà eseguito quando un thread di questa classe viene fatto partire. In questo caso, il metodo *run()*, stampa *I am a Worker Thread*.

```
1  public class First
2  {
3      public static void main(String[] args){
4          Worker1 runner = new Worker1();
5
6          runner.start();
7
8          System.out.println("I am the main thread");
9      }
10 }
```

Nella classe *First* viene utilizzata la classe predefinita prima, **Worker1**, per definire dei thread.

A riga 4 viene creato un oggetto thread, chiamato runner, creato dalla classe Worker1. A riga 6 si invoca un metodo **start()**. Il metodo *start()*, si trova dentro la classe predefinita **Thread**. Questo metodo inizializza un thread, crea un thread. In questo momento, alla fine dell'istruzione *runner.start()*, si hanno due entità che competono per il processero, il programma principale e il worker1 thread. Quale sarà la prima entità a essere eseguita? Non si sa. Si è in una situazione di **non determinismo.**

In questo momento ci si trova in una situazione di parallelismo. Ci sono due thread attivi: il main thread (programma principale) e il working thread. Essendo in una situazione di non determinismo, non è possibile prevedere in maniera esatta quale tra questi due thread eseguirà. A questo punto possiamo porci un quesito: quale stampa avverrà per prima? Stamperà per prima la stampa della classe *Worker1* o della classe *First*? Non si sa, siamo in una situazione di non determinismo.

Esiste un altro modo in Java per creare i thread. Si possono creare attraverso l'utilizzo dell'**interfaccia Runnable**.

```
1 public interface Runnable
2 {
3     public abstract void run();
4 }
```

Un interfaccia è una classe nella quale vengono specificati solo i metodi e i parametri che entrano in gioco. Nel codice soprastante, diciamo che esiste un'interfaccia **Runnable** che contiene un metodo *run()*. Chi implementa l'interfaccia **Runnable**, dovrà definire il metodo *run()*.

Le interfacce, in Java, sono utili per gestire l'ereditarietà multipla. C'è la possibilità di estendere solamente una classe alla volta, ma è possibile implementare più interfacce.

```
1 public class Worker2 implements Runnable
2 {
3     public void run(){
4         System.out.println("I am a Worker Thread");
5     }
6 }
```

La classe *Worker2* implementa *Runnable* e come tale deve definire il metodo *run()*, che era astratto nell'interfaccia Runnable.

Nella classe *Second*, creiamo il thread. La creazione avviene su tre fasi:

1. Inizialmente, riga 4, si crea un oggetto della classe Worker2 di tipo Runnable perché fa riferimento all'interfaccia Runnable. Al termine di questa fase, si è creato un oggetto ma non c'è ancora un thread.

2. La seconda fase, riga 5, consiste nella conversione dell'oggetto runner in thread. Si invoca la classe Thread predefinita in Java, la quale permette la creazione di thread, e gli si passa come paramentro l'oggetto runner creato in precedenza.

3. A questo punto, avendo creato il thread, si invoca il metodo start(). Il metodo start() crea il nuovo thread e alla fine invoca il metodo run(). In questo punto si hanno due attività concorrenti, in competizione tra di loro, per il processore. Siamo, di nuovo, in una situazione di non determinismo.

```
1 public class Second
2 {
3     public static void main(String[] args){
4         Runnable runner = new Worker2();
5         Thread thrd = new Thread(runner);
6 
7         thrd.start();
8 
```

```
9           System.out.println("I am the main thread");
10      }
11  }
```

Anche i thread hanno uno stato, come i processi. La Figura 4.2 non è da confondere con la Figura 3.1 vista in precedenza. La Figura 4.2 serve per dare una semantica alle operazioni su thread viste fin'ora. C'è uno stato runnable che ingloba i due stati running e ready. Quando si invoca il metodo start(), per fare partire un thread, ci si trova nello stato runnable. In questo stato il thread può girare: può completamente girare oppure si può trovare pronto per girare ma non lo fa ancora perché il processero è occupato con un altro thread.

Un thread che invoca un metodo di sleep(), suspend() o un'operazione I/O, viene bloccato e messo nello stato blocked.

Una operazione resume() permette a un thread di tornare allo stato runnable.

Una operazione di stop() si ha quando il thread è terminato.

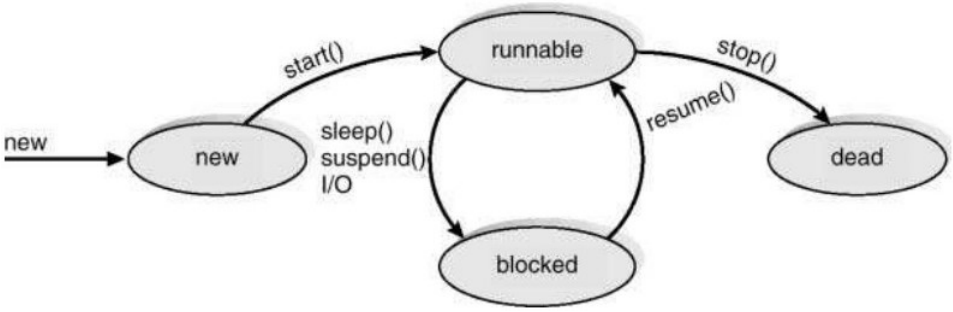

Figura 4.2: Java Thread States

Capitolo 5

CPU Scheduling

La CPU scheduling è quella parte del sistema operativo che si occupa della gestione del processore, in particolare si occupa delle politiche di scelta del processo che deve essere eseguire. Quale processo deve essere eseguito quando il processore è libero.

La CPU scheduling è strettamente legata al discorso del cambio di contesto, quindi multiprogrammazione. Un processo che sta eseguendo non viene lasciato sul processore fino alla fine della propria esecuzione ma, si accetta l'idea che questo processo possa perdere il controllo del processore a favore di un altro processo. Questo è il momento in cui c'è un cambio di contesto.

La Figura 5.1 mostra che cosa succede durante la "vita" di un processo. Un processo attraversa periodi di uso del processore e periodi di uso delle periferiche di I/O. Questi periodi sono di lunghezza variabile. Esistono processi che vengono chiamati **CPU bound** oppure **I/O bound**. Vengono indicati in questo modo perché sono processi che usano prevalentemente il processore o le periferiche di I/O.

Quando parliamo di scheduling possiamo fare una distinzione in tre tipologie:

1. **Scheduling a lungo termine:** Lo scheduling a lungo termine è quel sotto modulo che seleziona i processi che devono essere portati dalla memoria secondaria alla memoria centrale. Quando si vuole far eseguire un processo, esso deve essere presente sulla memoria centrale. Non è detto, però, che si riesca a portare sulla memoria centrale tutti i processi che si vogliono eseguire. Il sistema operativo, quindi, deve fare una scelta dei processi da portare dalla memoria secondaria alla memoria centrale. Questa decisione si chiama scheduling a lungo termine ed è una decisione che viene presa a intervalli di tempo abbastanza lunghi da parte del sistema operativo.

2. **Scheduling a breve termine:** Lo scheduling a breve termine, anche definito come CPU scheduler in senso stretto, è quel sotto modulo che sceglie quale processo far eseguire sulla CPU. Quale processo che si trova nella ready queue deve essere eseguito sulla CPU.

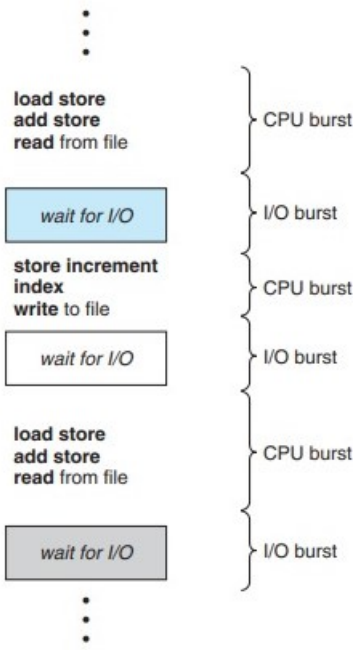

Figura 5.1: Alternating Sequence of CPU and I-O bursts

3. **Scheduling a medio termine (swapper):** Lo scheduling a medio termine, chiamato anche swapper, è quello scheduler che si occupa a trasferire momentaneamente dei processi dalla memoria centrale alla memoria secondaria (swap out) e viceversa (swap in). Questo scheduler può intervenire perché ci si potrebbe trovare in una situazione in cui c'è un numero molto alto di processi in esecuzione sul processore e ci potrebbe essere un nuovo o più di un nuovo processo la cui esecuzione viene richiesta poiché hanno una priorità molto alta, quindi devono essere assolutamente eseguiti.

   Non essendoci abbastanza spazio in memoria centrale e potendo aumentare ulteriormente il livello di multiprogrammazione, il sistema operativo in questo caso decide di sospendere un processo attualmente in esecuzione e di spostarlo temporaneamente su memoria secondaria (swap out). Oppure si può verificare anche il caso contrario, dove un processo viene spostato da memoria centrale alla memoria secondaria (swap in) perché, per esempio inattivo da molto tempo.

L'obiettivo, quindi, del CPU scheduling è di scegliere fra i processi in memoria che sono ready to execute (in stato ready), quali portare in stato running. Il modulo del sistema operativo in questione è il CPU scheduler. Egli deve prendere delle decisioni e quindi essere invocato dal sistema operativo stesso, quando:

1. il processo da running passa in stato waiting;

2. il processo da running passa in stato ready;
3. il processo da waiting passa in stato ready;
4. termina.

Uno scheduling si dice:

- **Nonpreemptive:** una volta selezionato un processo per l'esecuzione, questo processo rimane in esecuzione fintanto che ha bisogno del processore. Fintanto che, o non termina, o va in stato waiting perché, per esempio, ha invocato una periferica I/O.
- **Preemptive:** il processo potrebbe perdere il processore anche in un momento in cui egli vorrebbe rimanere sulla CPU perché ha ancora dei compiti da eseguire.

5.1 Scheduling Criteria

In questa sezione verranno elencati i criteri che devono essere ottimizzati.

- **CPU utilization:** è il criterio principale. L'obiettivo principale del sistema operativo è quello di tenere il processore attivo il più frequentemente possibile. L'utilization è una percentuale di tempo. L'obiettivo è arrivare $> 90\%$.
- **Response time:** è il secondo criterio principale. È importante per i processi di tipo interattivo. I processi interattivi sono quei processi che intergiscono con l'utente. Il response time è il periodo di tempo che passa da quando una richiesta viene inviata da parte dell'utente e quando il processore inizia a prendere in carica la richiesta stessa.
- **Waiting time:** è il periodo di tempo che un processo aspetta nella ready queue. È significativo perché il tempo di attesa in stato ready è un tempo perso dal punto di vista del processo. Il processo è pronto per eseguire ma non sta eseguendo perché abbiamo la multiprogrammazione, quindi possiamo dire che è il prezzo da pagare per la multiprogrammazione.
- **Turnaround time:** è il tempo complessivo che viene preso per eseguire un certo processo. Include il tempo in cui il processo è in esecuzione e il tempo in cui stato ready. (E tutti gli altri tempi).
- **Throughput:** è il numero di processi che completano la loro esecuzione per unità di tempo. Questo punto è significativo quando si hanno delle macchine che devono eseguire tanti progetti simili.

È evidente quali debbano essere i criteri da ottimizzare: bisogna massimizzare la CPU utilization e il throughput e minimizzare il response time, waiting time e turnaround time.

5.2 First-Come, First-Served (FCFS) Scheduling

L'algoritmo First-Come First-Served (FCFS) si basa su una procedura naturale: il primo che arriva è il primo a essere servito. Il primo processo che arriva nell'insieme ready sarà il primo a essere servito quindi il primo a essere eseguito. Inoltre, l'idea è quella di eseguire un processo fino in fondo. Il sistema operativo non interverrà per interrompere il processo mentre egli utilizza il processore. Questo è un algoritmo estremamente semplice. È un algoritmo naturale che, però, non è necessariamente quello migliore. Se la FCFS si comportasse in maniera efficiente rispetto ai criteri elencati precedentemente, si adotterebbe questo algoritmo. Purtroppo però la First-Come First-Served non è una politica ottimale sotto tanti punti di vista. In particolare è molto sensibile il waiting time a una scelta fra processi che vengono fatti partire in ordine diverso. L'obiettivo è quello di minimizzare il waiting time.

5.3 Shortest-Job First (SJR) Scheduling

Le problematiche del First-Come, First-Served portano a utilizzare un algoritmo e politica differente. La nuova politica utilizzata, SJR, porta a eseguire per primo il processo più corto, più breve. Quando si parla di lunghezza di un processo, si fa riferimento alla lunghezza dei CPU burst raffiguranti nella Figura 5.1.

La politica del Shortest-Job First viene anche chiamata politica ottimale, perché il waiting time medio è il migliore possibile, è quello ottimale. Non si può fare meglio. È ottimale sia in ambito *nonpreemptive* che in ambiot *preempitve*.

Il problema principale dello Shortest-Job First è non si conosce il burst time di un processo. È impossibile, dato un programma P scritto in un linguaggio di programmazione, determinare se il programma terminerà. È impossibile scrivere un programma e infatti possiamo dire che tale politica non è implementabile. Questa politica viene utilizzata per confrontare delle politiche realistiche, delle politiche che sono effettivamente implementabili.

Un altro problema della SJR è denominato **starvation**. Per starvation si intende una situazione in cui esiste un processo che non riesce ad andare avanti nella propria esecuzione. La macchina non è blocca, essa sta eseguendo ma c'è un processo che non riesce ad andare avanti. Il discorso di starvation è spesso menzionato in contrapposizione rispetto a quello dei **deadlock**, dove tutti i processi sono bloccati e nessuno riesce ad avanzare nella propria esecuzione.

Nel caso del SJR, queste eventualità sono legate al fatto che se esistono tanti processi brevi che si rendono disponibili per l'esecuzione e un processo più lungo, con periodo di CPU burst ampio, egli non riuscirà mai a eseguire.

5.3.1 Principio di località per lo Shortest-Job First

Il principio di località dice se il comportamento di un processo cambia molto lentamente nel tempo. Le istruzioni e i dati che il processo utilizza in un certo momento, definito come working set del processo stesso, saranno molto simili a quelli utilizzati nel periodo precedente o successivo. Il working set cambia molto lentamente nel tempo.

Questa località di tipo temporale viene utilizzata anche per approssimare la SJR. Lo Shortest-Job First funziona su un comportamente futuro dei nostri processi, quindi si fa riferimanto al periodo di CPU burst, cioè quello che succedere a questo processo nel futuro.

Il futuro non è determinabile, ma il passato si perché il processo ha eseguito sulla macchina. La località temporale ci serve per determinare la lunghezza del CPU burst futuro. Esso sarà simile alla lunghezza del CPU burst relativo al processo eseguito nell'imminente passato.

5.3.2 Prority Scheduling

Il sistema operativo tiene conto della priorità dei processi. Un modo per tenere conto della priorità è adottare una politica di scheduling su processore basato sulla priorità. Il prossimo processo che andrà sul processore sarà quello che viene considerato avere la priorità più alta. Ogni processo ha un numero di priorità associato. In base ai sistemi operativi, un numero di priorità basso significa che quel processo ha una priorità alta, ma può anche verificarsi il caso contrario, dipende dai sistemi operativi stessi.

La politica di priorità, come la politica SJF, può trovarsi nelle versioni preempitve e non preempitve. Nel caso preemptive, se un processo è in esecuzione sulla CPU e mentre sta girando arriva un nuovo processo, appartenente all'insieme dei processi ready, con una priorità più alta del processo attualmente in esecuzione, il sistema operativo può forzare un content switch e cambia il processo in esecuzione con il nuovo arrivato.

La politica di priorità presenta, anch'essa come la politica SJF, il problema della starvation. La SJF si può vedere come una scheduling di priorità, dove la priorità è data dalla lunghezza del prossimo periodo di CPU burst. In maniera astratta la SJF si può vedere in questo modo. Se un processo arriva nella coda dei processi ready e ha una priorità bassa, cioè un processo considerato poco importante, esso rischia la starvation. Per starvation, ricordiamo, la situazione in cui un processo, pur essendo pronto per l'esecuzione quindi all'interno dell'insieme ready, non ottiene mai fisicamente il controllo del processore. Rimane sempre in attesa.

La soluzione standard per la starvation si chiama **aging**. Questa tecnica consiste nell'aumentare la priorità di un processo quando egli resta nella coda di processi ready per un periodo prolungato di tempo. Se il sistema operativo si accorge che all'interno della coda dei processi ready c'è un processo che è rimasto in attesa per molto tempo, il sistema operativo stesso aumenta la priorità del processo. Se questo aumento di priorità

non è sufficiente ed egli rimane lo stesso in una situazione di starvation, viene aumentata ulteriormente la priorità finché non riuscirà ad eseguire. Questa tecnica di aging non si applica nel caso SJF.

Come viene determinata la priorità? La priorità può essere di due tipologie:

1. **Statica:** non viene modificata durante i periodi di vita dei processi;
2. **Dinamica:** durante i periodi di vita dei processi, la priorità può essere modificata in funzione del tempo di utilizzo della CPU.

Tipicamente nei sistemi operativi odierni, la priorità è un misto di queste due tipologie. Inizialmente, quando un processo viene fatto partire per la prima, ha una priorità assegnata dal sistema operativo stesso. Durante la vita del processo, la priorità può essere modificata. Il meccanismo di aging può cambiare, in parte, questa priorità.

5.3.3 Round Robin (RR)

La politica di Round Robin implementa il concetto di time-sharing. L'idea è quella di assegnare a ogni processo un quanto di tempo, nell'ordine dei millisecondi, e quando questo tempo è scaduto il processo viene tolto dal processore e messo sulla coda dei processi ready. L'insieme dei processi ready è gestito come una coda. È la stessa coda che viene utilizzata quando si parla della FCFS, cioè si inserisce da una parte e si recupera dall'altra, ma la differenza è il concetto di quanto di tempo. Gli spostamenti nella coda stessa sono derivati anche, ma non solo, dal termine del quanto di tempo. Di seguito vediamo un esempio del Round Robin con un quanto di tempo pari a 4:

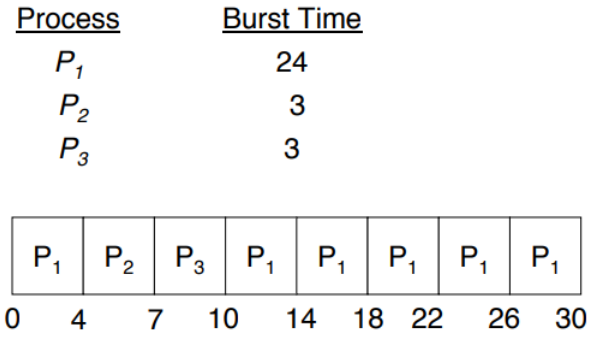

Figura 5.2: Round Robin

Inizialmente si prende il processo P_1 e lo si manda in esecuzione sulla CPU. Dopo 4 unità di tempo è ancora in esecuzione e, quindi, avviene un content switch perché il quanto di tempo per P_1 è terminato, anche se egli ha ancora del lavoro da svolgere. Ora si recupera dalla testa il processo P_2 e lo si manda in esecuzione. Egli esegue per 3 unità

di tempo e al tempo 7 viene terminato. Il processo P_2 non viene reinserito nella coda in quanto ha già terminato tutto il suo lavoro. Stessa cosa accade con P_3.

A questo punto l'unico processo che deve terminare la propria esecuzione è il processo P_1. Al tempo 10 si recupera P_1, il quale dopo 4 unità di tempo finisce il suo quanto di tempo e avviene un content switch. Il sistema operativo interviene per fare un content switch ma, quello che fa semplicemente, è rimettere in esecuzione P_1 poiché è l'unico rimasto. Questo operazione avviene in modo ciclico finché il processo P_1 non termina il proprio lavoro.

Se ci sono n processi nella ready queue e il quanto di tempo è q, allora ogni processo ottiene, supponendo che tutti i processi sono più o meno simili, $1/n$ del tempo di CPU in pezzetti che sono grandi q ciascuno. Ogni processo prima di poter partire con la propria esecuzione dovrà aspettare al massimo $(n-1)q$ unità di tempo.

La scelta del quanto di tempo q è fondamentale. Si possono verificare due casi:

- q **grande:** se il quanto di tempo è molto grande, la politica di Round Robin degenera in una FIFO.

- q **bassa:** se il quanto di tempo è molto basso, si avrà un numero molto alto di content switch. Si riesca che la somma degli overhead causati dai content switch, diventino molto significativi.

5.3.4 Multilevel Queue

Arrivati a questo punto ci possiamo chiedere quale sia la CPU scheluding più utilizzata. La Round Robin potrebbe essere la scelta migliore, ma diamo una risposta più concreta.

I sistemi operativi moderni cercano di combinare le varie politiche viste in precedenza per cercare di prendere i benefici di queste politiche e evitare, per quanto possibile, i loro inconvenienti. Il metodo che si utilizza per implementare questa idea, non è grazie a una singola queue per i processi ready ma un sistema a multi-livelli, cioè un sistema con tante queue.

L'insieme dei processi ready viene suddiviso in sottoinsieme, cioè in diverse code. Queste nuove code sono organizzate in base ai livelli di priorità. La priorità viene usata per stabilire su quale di questi livelli un processo viene inserito. Nella Figura 5.3 si hanno 5 livelli. La priorità interviene per scegliere il livello nel quale un processo venga messo, e i vari livelli possono essere gestiti con politiche differenti. Ovviamente i processi interattivi vengono gestiti con delle politiche di Round Robin. I processi bach di background potrebbero essere gestiti con una politica FCFS. Si possono avere politiche diverse su livelli diversi.

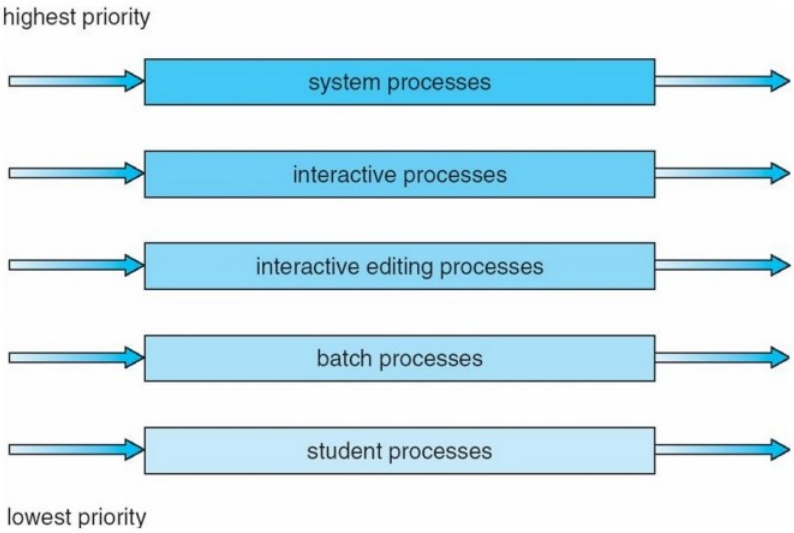

Figura 5.3: Multilevel queue

5.3.5 Processor Affinity

La processor affinity riguarda il discorso di macchine con più processori. Si immagini di avere un insieme di processi ready, dove all'interno dell'insieme c'è un processo P in testa alla coda, e 2 processori, CPU_1 e CPU_2. Un aspetto banale da tenere in considerazione è che il processo P non dev'essere richiesto contemporaneamente dai entrambi i processori. Un altro aspetto, meno banale, è quello che riguarda le memorie cache. Sappiamo che ci sono diversi livelli di cache e sappiamo che ogni processore ne può avere una.

Ora, supponiamo che il processo P venga eseguito sulla CPU_1, la memoria cache del processore 1 si popolerà di elementi che hanno a che fare con il processo P. Se, successivamente, il processo P perde il controllo del processore, quindi viene rimesso all'interno della coda nell'insieme dei processi ready, una volta che tornerà in esecuzione non ha molto senso che venga fatto eseguire sulla CPU_2. Perché?

Perché quando si ha un content switch, la cache non viene completamente svuotata, cioè non vengono eliminati tutti i dati relativi al processo P. Quindi, nel momento in cui P viene scelto di nuovo per l'esecuzione, c'è una buona probabilità che sulla cache per processore 1 esistano ancora degli elementi che hanno a che fare con P. Per questo motivo non avrebbe senso farlo eseguire sulla CPU_2. Questa caratteristica viene chiamata **processor affinity.**

Possiamo pensare che l'insieme dei processi ready possa essere diviso in due, in modo da creare due processi ready: $ready_1$ e $ready_2$. all'interno dell'insieme $ready_1$ verranno eseguiti i processi su CPU_1 e all'interno di dell'insieme $ready_2$ verranno eseguiti i processi su CPU_2. Questa situazioni funziona bene se i due insiemi sono equilibrati, se hanno

una cardinalità simile. Nel caso in cui ciò non si verifichi, c'è il rischio che un processore sia poco utilizzato. In questo caso si effettua una *migrazione*, **process migration**. Cioè si fa migrare un processo da un insieme ready a un altro.

Capitolo 6

Synchronization

In questo capitolo andremo a parlare di sincronizzazione e interferenze tra processi. In generale parleremo di concorrenza.

Nei capitoli precedenti abbiamo già parlato della differenza tra parallelismo apparente e parallelismo reale. Quando parliamo di **parallelismo/concorrenza apparente** ci riferiamo alla situazione in cui, dal punto di vista utente si ha l'impressione che ci siano diverse attività che la macchina sta svolgendo in contemporanea ma, dal punto di vista concreto, in realtà, siccome c'è un solo processore, in ogni istante c'è una sola istruzione che sta eseguendo.

Quando si parla di **parallelismo/concorrenza reale** ci sono più unità di calcolo e più processori. Ciò comporta alla possibilità che entrambi i processori siano attivi in un certo istante, e durante tale periodo ci saranno più istruzioni che sono fisicamente eseguite. Un caso particolare del parallelismo reale sono i sistemi distribuiti.

6.1 Che cos'è la concorrenza?

Un esecuzione concorrente è una situazione in cui ci sono due programmi in esecuzione contemporaneamente. Con concorrenza, in senso ampio, è l'insieme delle notazioni per descrivere l'esecuzione concorrente di due o più programmi. È l'insieme di tecniche per risolvere i problemi associati all'esecuzione concorrente, quali **comunicazione** e **sincronizzazione**. La concorrenza è stata per i sistemi operativi per permettere la multiprogrammazione. Il fatto che il processore possa essere condiviso tra più processi indipendenti.

Il caso tra parallelismo apparente e parallelismo reale, da un punto di vista della concorrenza, non presentano differenze. I concetti e le problematiche sono piuttosto simili. Per comprendere a pieno il concetto e i problemi di interferenza, vediamo il seguente esempio:

Si consideri il seguente codice:

```
1 void modifica(int valore){
2     totale = totale + valore
3 }
```

Questa procedura, **modifica**, prende in input un intero parametro **valore**, e tutto quello che fa questa procedura è aggiungere a una variabile globale, **totale**, il valore intero che è stato preso in input come parametro. Supponiamo che:

- Esiste un processo P_1 che esegue **modifica(+10).**
- Esiste un processo P_2 che esegue **modifica(-10).**
- P_1 e P_2 vengono lanciati in contemporanea. Supponiamo di avere un linguaggio di programmazione, per esempio Java, che ci permetta di fare ciò.
- **totale** è una variabile condivisa tra i due processi, con valore inziale 100.

A questo punto ci chiediamo: quanto varrà **totale** alla fine dell'esecuzione dei processi P_1 e P_2?

Ci aspettiamo che il risultato di totale sia sempre uguale a 100, ma non è il caso. Nel senso che effettivamente, se si provasse a fare questo esempio, nella maggior parte dei casi avremo come valore finale 100, ma questo valore non è garantito. Ci potrebbero essere situazione in cui i due processi P_1 e P_2 terminano con **totale** cne non vale 100.

Per comprendere questo concetto dobbiamo pensare a ciò che si era detto quando abbiamo parlato di CPU. Il processore non esegue le istruzioni di un linguaggio ad alto livello, come può essere C o Java, ma esegue le istruzioni del linguaggio macchina, Assembley, di quel processore. Ogni CPU ha il proprio linguaggio. Per eseguire dei codici di un qualunque linguaggio esso sia su quel processore, bisogna prima tradurre il linguaggio sorgente nel linguaggio macchina. Una istruzione come quella della procedura **modifica**, quando viene tradotta in linguaggio macchina potrebbe essere tradotta nel codice seguente:

Algorithm 2 In Assembly:

```
.text
modifica:
lw $t0, totale
add $t0, $t0, $a0
aw $t0, totale
```

Bisogna pensare che totale sia una variabile globale, quindi che una variabile che si trova in memoria centrale. Come già detto, un processo per essere eseguito deve avere il proprio codice e dati presenti sulla memoria centrale. L'operazione di somma

viene effettuata all'interno del processore, in particolare dalla ALU. L'ALU permette di eseguire le operazioni aritmetiche e booleane. Questo spiega perché questa singola istruzione venga divisa in tre istruzioni.

La prima istruzione è una istruzione di **load**, in cui il valore di totale viene recuperato dalla memoria centrale e portato su un registro t_0 del processore. Il parametro valore viene inserito in un registro dell'ALU, un registro che si chiama a_0. A questo punto effettuiamo la somma tra i due registri. L'istruzione di somma è data dall'istruzione **add**. Assumiamo che l'output venga depositato di nuovo sul registro t_0. Terminata la somma, bisogna riportare il valore finale sulla varibile **totale** che è presente in RAM, istruzione **sw**.

P1	`lw $t0, totale`	totale=100, $t0=100, $a0=10
P1	`add $t0, $t0, $a0`	totale=100, $t0=110, $a0=10
P1	`sw $t0, totale`	totale=110, $t0=110, $a0=10
S.O.	`interrupt`	
S.O.	`salvataggio registri P1`	
S.O.	`ripristino registri P2`	totale=110, $t0=? , $a0=-10
P2	`lw $t0, totale`	totale=110, $t0=110, $a0=-10
P2	`add $t0, $t0, $a0`	totale=110, $t0=100, $a0=-10
P2	`sw $t0, totale`	totale=100, $t0=100, $a0=-10

Figura 6.1: Scenario 1 - multiprogramming (corretto)

Nella Figura 6.1, possiamo vedere uno scenario corretto. Porta a una situazione finale corretta. In questa caso abbiamo un solo processore, quindi ci può essere una sola istruzione che può essere eseguita in un certo momento. Sulla prima colonna è presente l'entità per la quale il processore sta eseguendo l'istruzione. Nella seconda colonna è presente l'istruzione che viene eseguita. Nell'ultima colonna, invece, si può il valore di certe celle quando l'istruzione nella riga corrispondente è terminata. Possiamo vedere il valore della cella totale di RAM, il valore di t_0 e di a_0. Questi due ultimi valori sono registri del processore.

Quando viene eseguita la prima istruzione, load totale su t_0. Inizialmente totale vale 100 e l'esecuzione di questa istruzione non modifica il valore di totale. t_0 prende il valore 100 e il registro a_0, cioè il registro nel quale abbiamo scritto il valore del parametro della procedura modifica che è stata invocata.

La seconda istruzione è una istruzione di somma. Si somma t_0 e a_0 e si registra tale somma su t_0. Alla fine si avrà totale sempre pari a 100 perché non viene interpellato da questa istruzione. Il valore finale di t_0 è $t_0 + a_0$, quindi $100 + 10 = 110$ e a_0 non viene modificata e rimane a 10.

L'ultima istruzione di P_1 è l'istruzione con la quale scriviamo su totale il valore di t_0. Viene modificato solamente totale che diventa uguale a 110, mentre t_0 e a_0 rimangono invariati.

A questo punto supponiamo che ci sia un content switch e mandiamo in esecuzione il processore P_2. Effettuando il content switch, salviamo il valore dei registri del processore relativi all'esecuzione di P_1 sul Process Control Block.

A questo punto, prima di far partire l'esecuzione di P_2, bisogna ripristinare i valori dei registri del processore con i valori che si avevano quando il processo P_2 era stato interrotto precedentemente. (Supponiamo che P_2 si una processo che aveva già eseguito e che era stato precedentemente interrotto). Il risultato del content switch porta ad avere: totale rimarrà pari a 110 in quanto un content switch non va mai a cambiare il valore dei dati del programma. I registri interni del processore, invece, cambiano. Per t_0 non si sa che valore potrà avere. Dipenderà dal valore che t_0 aveva quando è stato interrotto precedentemente. Mentre a_0 ha valore -10, perché il processo P_2 invoca la procedura **modifica(-10)**.

A questo punto si può far partire P_2. Le istruzioni che vengono eseguite sono le stesse che del processo P_1 eseguite prima, ma con i valori relativi al processo P_2.

Si termina, correttamente, con il valore di totale pari a 100.

Di seguito verrà mostrato un nuovo scenario di multiprogramming. Questo tipo di situazione, però, è errata. Le operazioni svolte sono le medesime dello scenario precedente.

P1	`lw $t0, totale`	totale=100, $t0=100, $a0=10
S.O.	`interrupt`	
S.O.	`salvataggio registri P1`	
S.O.	`ripristino registri P2`	totale=100, $t0=? , $a0=-10
P2	`lw $t0, totale`	totale=100, $t0=100, $a0=-10
P2	`add $t0, $t0, $a0`	totale=100, $t0= 90, $a0=-10
P2	`sw $t0, totale`	totale= 90, $t0= 90, $a0=-10
S.O.	`interrupt`	
S.O.	`salvataggio registri P2`	
S.O.	`ripristino registri P1`	totale= 90, $t0=100, $a0=10
P1	`add $t0, $t0, $a0`	totale= 90, $t0=110, $a0=10
P1	`sw $t0, totale`	totale=110, $t0=110, $a0=10

Figura 6.2: Scenario 2 - multiprogramming (errato)

6.2 Producer-Consumer Problem

Il problema Producer-Consumer è un problema classico di concorrenza. È un problema estremamente semplice, ma permette di individuare una serie di problematiche a livello di interferenza tra processi.

Il produttore è un processo che produce dei dati che vengono consumati dal consumatore. Il produttore e il consumatore sono dei processi separati; ognuno svolge un'attività molto semplice. Il produttore produce dei dati in sequenza e il consumatore consuma questa sequenza di dati. Ogni dato creato dal produttore deve essere consumato dal consumatore. Un problema che si pone da questa situazione è che il produttore potrebbe produrre dati a una velocità diversa da quella con cui il consumatore li consuma. Se vogliamo che ogni dato prodotto dal produttore sia consumato dal consumatore, è necessario che la velocità dei due processi sia sincronizzata.

Un modo per ovviare a questa situazione è introdurre una struttura tra i due processi chiamata **buffer**. Il buffer ha un certo numero di posizioni e un puntatore, chiamato **in**, che indica in quale cella del buffer il produttore debba depositare i propri elementi. Anche il consumatore ha un proprio indice, chiamato **out**, che indica l'indice della cella dalla quale il consumatore deve prelevare il dato. Il buffer è una struttura circolare. Quando il produttore ha riempito tutte le celle, egli riparte dalla cella iniziale del buffer, se ovviamente esse è libera.

Il buffer funge da mediatore per cercare di alleviare i problemi che derivano dalle diverse velocità dei produttori e consumatori. Questa funzionalità, però, non elimina definitivamente questo problema.

Il codice del produttore è il seguente: La variabile count tiene traccia del numero di dati depositati nel buffer che non sono ancora stati consumati. BUFFER_SIZE indica la taglia del buffer. Se count è uguale a BUFFER_SIZE, il produttore non ha più spazio per scrivere nel buffer e deve andare avanti. Altrimenti, l'item prodotto dal produttore viene inserito nella casella **in** del buffer. **in** viene incrementato di 1 modulo BUFFER_SIZE. E infine count viene incrementato.

```
1  /* CODICE PRODUTTORE */
2  while(true){
3      /* produce an item and put in nextProduced */
4      while(count == BUFFER_SIZE){
5          ;// do nothing
6          buffer[in] = nextProduced;
7          in = (in + 1) % BUFFER_SIZE;
8          count++;
9      }
10 }
```

Il codice del consumatore è pressoché simile a quello del produttore. Se *count* == 0, non faccio nulla. In caso contrario, si preleva un elemento dal buffer, **buffer[out]**, e lo si registra nella variabile nextConsumed. Si incrementa out di 1 modulo BUFFER_SIZE. E infine si decrementa la variabile count, in quanto si è estratto un dato dal buffer.

```
/* CODICE CONSUMATORE */
while(true){
    while(count == 0){
        ;// do nothing
        nextConsumed = buffer[out];
        out = (out + 1) % BUFFER_SIZE;
        count--;
        /* consume the item in nextConsumed */
    }
}
```

Di seguito vediamo una classe in Java che implementa un BoundedBuffer: Questa classe ha due metodi fondamentali: il metodo **enter()** e il metodo **remove()**. Il metodo enter() viene invocato dal produttore per depositare un elemento sul buffer. Il metodo remove() viene invoca dal consumatore per eliminare un elemento dal buffer.

```
public class BoundedBuffer {
    public void enter(Object item) {
        // producer calls this method
        public void enter(Object item) {
            while (count == BUFFER_SIZE)
                ; // do nothing
            // add an item to the buffer
            ++count;
            buffer[in] = item;
            in = (in + 1) % BUFFER_SIZE;
}
    }
    public Object remove() {
        // consumer calls this method
        public Object remove() {
            Object item;
            while (count == 0)
                ; // do nothing
            // remove an item from the buffer
            --count;
            item = buffer[out];
            out = (out + 1) % BUFFER_SIZE;
            return item;
```

```
24    }
25        }
26        // potential race condition on count
27        private volatile int count;
28    }
```

I produttori invocano **enter()** e i consumatori invocano **remove()**. Che cosa potrebbe andare male? Per andare male, in questo caso, significa che i dati prodotti dal produttore non sono consumati o sono duplicati. Capita qualcosa di incorretto, qualcosa che non vogliamo che accada.

La variabile count potrebbe essere una variabile critica, in quanto è l'unica variabile condivisa da entrambi i processi. Il produttore la incrementa mentre il consumatore la decrementa. Ciò potrebbe portare qualche tipo di problema. Nel caso in cui ci trovassimo in una situazione con più consumatori e produttori, anche le variabili **in** e **out** diventano condivise. Anch'esse possono creare delle situazioni di criticità.

Un'altra situazione critica potrebbe essere quando ci sono due produttori che invocano il metodo enter() contemporaneamente. In questo caso il problema si presenta se avviene un content switch prima dell'incremento della variabile **in**, perché entrambi i produttori vanno a scrivere sullo stesso elemento del buffer. (Tra riga 9 e 10). Analogamente, si presenta lo stesso problema nel caso in cui ci siano due consumatori che invoca il metodo remove().

I problemi possono verificarsi anche solamente con un produttore e un consumatore. Si supponga che il buffer sia vuoto e il produttore e consumatore invoca in contemporanea il metodo enter() e remove(). Che cosa potrebbe andare storto? Avendo un solo processore, può essere eseguita una sola istruzione per volta. Si supponga che il primo a eseguire sia il produttore attraverso il metodo enter(), il quale, dopo un certo quanto di tempo determinato dal sistema operativo, deve effettuare un content switch. A che livello di codice il content switch potrebbe creare dei problemi?

La situazione problematica si trova tra la riga 8 e 9. Nel caso in cui il content switch dovesse avvenire dopo l'incremento della variabile count ma prima dell'inserimento dell'item nel buffer. Il produttore ha incrementato count, dunque count passa da 0 a 1. A questo punto il sistema operativo effettua un content switch e il consumatore inizia l'esecuzione del metodo remove(). Egli "passa" il while, in quanto che la variabile count è stata incrementata precedentemente dal produttore. Il problema è che la variabile count vale 1 ma è il buffer è ancora vuoto poiché il content switch è avvenuto prima dell'inserimento dell'item nel buffer, quindi il consumatore non riesce a rimuovere alcun item perché il buffer è ancora vuoto.

6.3 Critical Region

La critical region è la parte di un processo che accede a delle strutture dati condivise. Rappresenta un potenziale punto di interferenza. Se si ha un processo P e un processo Q che condividono una variabile x. Una regione critica è una zona di codice di un processo che prevede l'utilizzo di una variabile/risorsa condivisa. Se un processo, P per esempio, accede a una variabile condivisa x, è necessario che nessun altro processo, in quel momento, acceda alla variabile x. Questa caratteristica viene chiamata **mutual exclusion.** Indica il fatto che quando un processo opera sulla variabile condivisa, nessun altro può fare altrettanto.

6.3.1 Solution to Critical-Section Problem

Una soluzione al problema della sezione critica deve soddisfare alcune condizioni:

1. **Mutual Exclusion**: se un processo P esegue in sezione critica, quindi si trova all'interno della sezione critica cioè sta manipolando le variabile condivise, allora nessun altro processo può essere in quel momento nella sezione critica. Questa è la condizione più importante.

2. **Progress**: la condizone di progesso dice che se nessun processo attualmente è in esecuzione nella sezione critica, e ci sono dei processi che vorrebbero entrarci, la selezione del processo che dovrà entrarci non può essere rimandata all'infinito.

3. **Bounded Waiting**: la condizione di progresso, in se, non è sufficiente perché potrebbe esserci una situazione in cui un processo P ha l'uso esclusivo della variabile condivisa x, (esempio precedente), e la usa in continuazione. In questo caso un processo Q che vorrebbe utilizzare la variabile condivisa x non riesce ad avanzare nel proprio codice. La condizione di bounded waiting dice che ci dev'essere un limite di attesa per un processo, prima di accedere alla sua sezione critica. Sostanzialmente se un processo vuole accedere a una risorsa condivisa, prima o poi ce la farà. Non può rimanere in attesa infinita.

A questo punto vediamo dei meccanismi che ci permettono di garantire queste tre condizioni. Vediamo come implementare questo problema per essere risolto.

Immaginiamo due thread che hanno delle sezioni critiche. Cerchiamo di capire come potrebbe essere scritto il codice di questi thread per garantire il problema della sezione critica.

Il codice seguente è il codice della classe **Worker** con la quale si creano i thread. Nei capitoli precedenti abbiamo visto che il modo standard per creare i thread è quello di estendere *Thread*. Il costruttore della classe prende in input 3 parametri. Una stringa, un intero e MutualExclusion.

MutualExclusion è un arbitro presente in un'altra classe. La classe Worker implementa un arbitro il cui compito è quello di per risolvere il problema della sezione critica. Il worker thread viene creato attraverso un intero, che è il proprio identificativo, e un oggetto arbitro *s*, che il thread potrà usare per gestire la propria sezione critica.

I thread sono dentro in ciclo dove, di tanto in tanto, utilizzano la sezione critica. Ciò avviene all'interno del metodo run(). Prima di entrare in sezione critica, il thread invoca un metodo **enteringCriticalSection(id)** dell'arbitro shared e quando ha finito la sezione critica invoca il metodo **leavingCriticalSection(id)**. In entrambi i casi fornisce come parametro il proprio identificativo, **id**, perché l'arbitro deve sapere qual è il thread che ha fatto la richiesta.

Invocare il metodo **enteringCriticalSection(id)** significa che il thread chiede all'arbitro il permesso di poter entrare in sezione critica. Mentre, quando si invoca l'istruzione **leavingCriticalSection(id)**, il thread dice all'arbitro che ha concluso la sezione critica.

```
1  public class Worker extends Thread {
2      public Worker(String n, int i, MutualExclusion s){
3          name = n;
4          id = i;
5          shared = s;
6      }
7
8      public void run() {
9          while (true) {
10             shared.enteringCriticalSection(id);
11             // in critical section code
12             shared.leavingCriticalSection(id);
13             // out of critical section code
14         }
15     }
16
17     private String name;
18     private int id;
19     private MutualExclusion shared;
20 }
```

A questo punto vediamo com'è definita la classe **MutualExlcusion**, ovvero l'arbitro.

```
1  public abstract class MutualExclusion {
2      public static void criticalSection() {
3          // simulate the critical section
4      }
5      public static void nonCriticalSection() {
6          // simulate the non-critical section
7      }
```

```
8       public abstract void enteringCriticalSection(int t);
9       public abstract void leavingCriticalSection(int t);
10      public static final int TURN_0 = 0;
11      public static final int TURN_1 = 1;
12  }
```

I metodi **enteringCriticalSection(int t)** e **leavingCriticalSection(int t)** sono i metodi cruciali della classe. Sono i metodi che i thread invocano per l'entrata in sezione critica e per segnale l'uscita da essa.

La classe **TestAlgorithm** è il programma principale. La classe test per testare diversi algoritmi che spiegeremo a breve. A riga 4 si crea un arbitro, quindi si crea un oggetto della istanziazione algoritmo della classe astratta MutualExclusion. A riga 6 e 7 si creano i due thread, first e second. Prendo in input come parametri un identificativo e l'arbitro. Nelle due istruzioni finali i due thread vengono effettivamente creati. Sono attivi e possono eseguire in contemporanea.

```
1   public class TestAlgorithm
2   {
3    public static void main(String args[]) {
4       MutualExclusion alg = new Algorithm_1();
5
6       Worker first = new Worker("Runner 0", 0, alg);
7       Worker second = new Worker("Runner 1", 1, alg);
8
9       first.start();
10      second.start();
11   }
12  }
```

Ora vediamo differenti algoritmi di arbitro.

6.3.1.1 Algoritmo 1

Questo algoritmo è una prima definizione di arbitro. La classe algoritmo 1 estende la classe estratta MutualExclusion. La cosa che ci interessa è come vengono definiti i due metodi **enteringCriticalSection(int t)** e **leavingCriticalSection(int t)**.

Nel momento in cui viene creato l'oggetto di tipo algorimo, la variabile **turn** assume un valore che è dato dalla costante **TURN_0**, cioè valore uguale a 0. Attraverso i metodi **enteringCriticalSection(int t)** e **leavingCriticalSection(int t)** l'arbitro gestisce la sezione critica.

Nel metodo enteringCriticalSection(int t), il comportamento dell'arbitro è il seguente: l'arbitro ha la variabile turn, che può valere 0 o 1, ed egli consulta il valore di turn per determinare se un il thread che invoca il metodo abbia il diritto di entrare in sezione

critica. **Thread.yield()** è un'istruzione che dice alla Java Virtual Machine che il thread seguente può essere interrotto in favore di altri thread.

```
public class Algorithm_1 extends MutualExclusion {
    public Algorithm_1() {
        turn = TURN_0;
    }
    public void enteringCriticalSection(int t) {
        while (turn != t)
        Thread.yield();
    }
    public void leavingCriticalSection(int t) {
        turn = 1 - t;
    }
    private volatile int turn;
}
```

La variabile turn si modifica nel metodo leavingCriticalSection(int t) che viene invocato dal thread che ha finito di eseguire nella propria sezione critica. Quindi lo scopo principale di questo metodo è quello di modificare la variabile turn. A questo punto l'arbitro può dare il permesso al thread successivo di entrare nella propria sezione critica.

Qual è il problema di questo arbitro? È sempre accettabile oppure no? È soddisfacente?

Il problema principale è che l'arbitro impone un'alternanza esatta sulla sezione critica da parte dei due thread. Significa che, secondo l'arbitro, i due thread avranno l'uso del tutto simile della sezione critica e, fra l'altro, un uso che prevede un'alternanza che inizia con 0, decisione presa dall'arbitro senza nessuna ragione valida. Non si può assumere che i due thread abbiamo un uso simile della sezione critica.

Questo algoritmo rispetta sicuramente la condizione Mutual Exclusion ma potrebbe portare alla violazione delle altre due. In particolare la condizione di progresso.

6.3.1.2 Algortimo 2

Vediamo un altro algoritmo. Nell'algortimo 1 l'arbitro decideva in maniera autonoma senza ascoltare il thread, cioè senza avere una idea di quale fossore le esigenze dei thread stessi. Ciò non è corretto.

```
public class Algorithm_2 extends MutualExclusion {
    public Algorithm_2() {
        flag[0] = false;
        flag[1] = false;
    }
    public void enteringCriticalSection(int t) {
```

```
7           int other = 1 - t;
8           flag[t] = true;
9           while (flag[other] == true){
10              Thread.yield();
11          }
12      }
13      public void leavingCriticalSection(int t) {
14          flag[t] = false;
15      }
16      private volatile boolean[] flag = new boolean[2];
17  }
```

Anche nell'algoritmo 2, ovvero l'arbitro 2, la parte cruciale della classe sono i due metodi enteringCriticalSection(int t) e leavingCriticalSection(int t).

Quando un thread invoca il metodo enteringCriticalSection(int t) cosa succede? La varibile **other** indica il puntatore relativo a un altro thread considerato. L'arbitro si segna su flag[t] la richiesta del thread per entrare in sezione critica, quindi si segna a true. A questo punto per dare effettivamente il permesso al thread t di eseguire nella sezione critica, si consulta il flag dell'altro thread, **flag[other]**, e vediamo se esso è uguale a true. Nel caso in cui sia uguale a true, significa che in un momento precedente, il flag di other è stato settato a true e quindi in questo momento quel thread si trova in sezione critica.

Quando un thread invoca il metodo leavingCriticalSection(int t), cioè ha terminato la propria sezione critica, il flag viene settato a false.

Anche questo algoritmo presenta un problema. Ci potrebbe essere una violazione di una delle condizioni precedentemente elencate.

Il problema è che si potrebbe arrivare a una situazione di stallo. Viene violata, di nuovo, la condizione di progresso.

Si supponga che all'inizio i due flag siano settati a false. Arriva il primo thread, T_0, e inizia a eseguire il codice setta il flag del thread a true. Il problema si registra se avviene un content switch prima del while, (tra riga 8 e riga 9). Nel caso in cui ci sia un content switch, il thread T_1 va dunque in esecuzione e invoca il metodo enteringCriticalSection(). Pertanto invoca il metodo con il valore del parametro t pari a 1. Facendo ciò, succede che anche il thread T_1 mette il proprio flag uguale a true. Ora entrambi i thread, prima del while, sono settati a true e quindi sono bloccati sul while. Qualunque dei due thread esegua, non riescono a superare il while perché l'altro thread risulta essere sempre uguale a true.

6.3.1.3 Algoritmo 3

Algoritmo di Peterson.

L'idea di questo algoritmo è quella di mettere assieme di due algoritmi precedenti. L'idea è quella di modificare il while del metodo enteringCriticalSection, attraverso l'aggiunta di una condizione. Si aggiunge una variabile **turn** condivisa che deve avere lo stesso valore, in un certo momento, per entrambi i thread. Questa variabile turn permette di andare avanti.

All'interno del while si va a controllare se il flag dell'altro thread sia uguale a true, e se turn sia uguale a other. Consideriamo due situazioni importanti:

1. **Situazione standard**: pensiamo a una situazione standard in cui si due flag, $Flag[0]$ e $Flag[1]$ settati inizialmente a false e la variabile $turn = 0$. Si supponga che, dati i thread T_0 e T_1, a un certo punto il thread T_0 entri in critical section e invochi il metodo enteringCriticalSection(int t). Ora il $Flag[0]$ viene settato a true e la variabile *turn* viene settata al valore dell'altro thread, quindi $turn = 1$. A questo punto T_0 riesce a superare il while, perché il $thread[other] = false$ e $turn = other$, e quindi egli riesce a entrare in sezione critica.

   Se durante l'uso della sezione critica arriva T_1 e cerca anch'esso di entrarci, succede che: egli setta il $Flag[1] = true$ e $turn = 0$. A questo punto però il while risulta essere bloccante perché il thread T_1 si trova sul while con il $flag[other] = true$ e $turn = other$. Accade ciò perché non c'è ancora stato un content switch e all'interno della sezione critica c'è ancora il thread T_0. Dunque il thread T_1 non può ancora accedervi.

2. **Situazione problematica**: pensiamo sempre di avere una situazione iniziale con i $Flag[0]$ e $Flag[1]$ settati a false e $turn = 0$. Una situazione problematica è quando i due thread cercano di entrare quasi in contemporanea nella sezione critica.

   Pensiamo a una situazione in cui arrivi il thread T_0 che setta il $Flag[0] = true$ e $turn = 1$. Se il sistema operativo effettuasse in questo momento il content switch, cioè prima del while, a questo punto arriverebbe anche il thread T_1 che setterebbe anch'esso il proprio $Flag[1] = true$ e cambierebbe di nuovo la variabile turn, riportandola alla situazione iniziale, cioè $turn = 0$. A questo punto ci si trova nella situazione in cui entrambi i thread vogliono entrare in sezione critica. La condizione $turn = other$ sarà vera soltanto per uno dei due thread. In particolare sarà vera per il thread che ha settato *turn* l'ultima volta, quindi nel nostro caso T_1. Per entrambi i thread la prima parte del while è true ma la seconda condizione è vera solo per il thread T_1.

   Quindi il thread T_0 trova globalmente il while false e dunque riesce a proseguire entrando nella propria sezione critica. Quando avrà terminato setterà il $Flag[0] = false$ e, a questo punto, anche il thread T_1 sarà in grado di uscire dal while perché la prima condizione del while è $false$ e quindi anche il T_1 sarà in grado di entrare nella propria sezione critica.

Quando anch'esso avrà terminato, setterà il $Flag[1] = false$ e ci si ritroverà in una situazione che è esattamente la stessa della situazione iniziale: cioè con $Flag[0] = false$, $Flag[1] = false$ e $turn = 0$. Tutto è andato correttamente.

```
1  public class Algorithm_3 extends MutualExclusion {
2      public Algorithm_3() {
3          flag[0] = false;
4          flag[1] = false;
5          turn = TURN_0;
6      }
7      public void enteringCriticalSection(int t) {
8          int other = 1 - t;
9          flag[t] = true;
10         turn = other;
11         while ((flag[other] == true) && (turn == other))
12             Thread.yield();
13         }
14     public void leavingCriticalSection(int t) {
15         flag[t] = false;
16     }
17     private volatile int turn;
18     private volatile boolean[] flag = new boolean[2];
19 }
```

6.4 Semaphore

Fino ad ora abbiamo esaminato i problemi di interferenza tra processi. In particolare abbiamo visto il problema della sezione critica e abbiamo cercato di risolvere questo problema dal punto di vista software. La soluzione software non è poi così banale. Se la gestione della sezione critica fosse interamente affidata al programmatore, sarebbe molto complicato. Però, sono stati introdotti dei costrutti per facilitare il compito del programmatore.

Edsger Dijkstra introdusse il concetto di **semaforo.** L'idea del semaforo è quella di introdurre nel sistema operativo dei costrutti specifici per aiutare il programmatore a gestire i problemi di concorrenza. Il semaforo svolge una funzione simile a quella dell'arbitro. Il semaforo è una variabile intera. L'unico uso di questa variabile è l'incremento e il decremento. L'incremento e decremento sono determinate da due operazioni chiamate $P(S)$ e $V(S)$.

- $P(S)$**:** while $S \leq 0$ do no-op;

  $S--$;

- $V(S)$**:** $S++$;

L'idea generale di semaforo è che se ci si trova all'interno di un codice nel quale c'è un pezzo di sezione critica, introduciamo un semaforo, S, che gestire una risorsa condivisa x e prima di entrare nella sezione critica effettuiamo un decremento su S. Uscendo dalla sezione critica effettuiamo un incremento su S. Le operazioni di incremento e decremento vanno pensate, facendo riferimento agli algoritmi visti precedentemente, a delle richieste verso il semaforo per l'accesso alla sezione critica. Un semaforo funge simile all'arbitro, quindi chiediamo all'arbitro di accedere alla sezione critica.

Il sistema operativo fornisce dei costrutti per accedere a delle risorse condivise. Il sistema operativo fornisce dei semafori, il caso classico è quello del **semaforo binario**, dove è possibile assumere solo valori 0 e 1.

Il sistema operativi implementa i semafori e li rende disponibili ai livelli superiori. I programmatori potranno utilizzare i semafori per gestire e risolvere i problemi di concorrenza. Si supponga che il sistema operativo implementi i semafori, nel modo visto precedentemente, cioè attraverso le operazioni **P(S)** e **V(S)**. Questo tipo di implementazione può causare problemi?

Si immagini di avere un semaforo $S = 1$ e due processi P e Q che eseguono il codice di $P(S)$ in contemporanea. Cosa potrebbe succedere? Entrambi i processi eseguono lo stesso codice dell'operazione $P(S)$ in parallelo, che cosa può andare storto? Si possono creare dei problemi di interferenza?

Un problema di interferenza possibile è quello in cui la variabile S assuma valore -1. Inizialmente $S = 1$ e uno dei due processi, per esempio P, inizia a eseguire. Essendo $S = 1$, il processo P supera il while. Il problema d'interferenza si può verificare se prima di effettuare il decremento della variabile S, avviene un content switch. Ciò porterebbe in esecuzione il processo Q, con il valore di S ancora pari a 1. A questo punto anche il processo Q supera il while e decrementa S portandolo uguale a 0. Ora il processo Q può effettivamente entrare in sezione critica.

Se, però, una volta che il processo Q è entrato in sezione critica, dovesse avvenire un altro content switch, il processo P decrementerebbe a sua volta la variabile S portandola pari a -1, poiché egli aveva superato il while precedentemente. Così facendo anche il processo P entra nella sezione critica, con tutti i problemi del caso. In una situazione di questo tipo è stata violata la condizione della mutual exclusion sulla sezione critica.

Un altro problema di interferenza potrebbe accadere quando due processi eseguono l'operazione $V(S)$ contemporaneamente. Essendo l'incremento una operazione non atomica, ci sarebbero due processi che manipolano una variabile condivisa in contemporanea. Ciò porta ai problemi visti precedentemente.

A questo punto si potrebbe pensare che i semafori non risolvono il problema della sezione critica, però non è così. Vediamo come si possono chiudere tali problemi. Il passo mancante è semplice.

Ricordiamoci che i semafori sono implementati dal sistema operativo, perciò non è un qualunque programma utente che esegue il codice. Il sistema operativo può decidere quando le interrupt e i content switch hanno luogo. Egli può decidere che, mentre si esegue del codice a livello di sistema operativo, non avvengano dei content switch. Egli può decidere che durante l'esecuzione di incremento e decremento su S, non ci siano content switch. Può fare in modo che tali operazioni siano eseguite in maniera atomica. Il sistema operativo si permettere di farlo perché si sta eseguendo codice di sistema operativo e non codice utente.

Ora vediamo delle implementazioni di semafori:

6.4.1 Semaphore Eliminating Busy-Waiting

```
1  P(S){
2      S--;
3      if(S < 0){
4          // add this process to list
5          block
6      }
7  }
8
9  V(S){
10     S++;
11     if(S <= 0){
12         // remove a process P from list
13         wakeup(P);
14     }
15 }
```

L'idea di questa implementazione è quella di prendere un semaforo S, non soltanto come una variabile intera, ma anche come una lista di processi. Processi che sono in attesa sul semaforo.

Nell'operazione $P(S)$ viene inizialmente decrementato il semaforo e successivamente analizzato il suo valore. Se il valore del semaforo è strettamente minore di 0, il processo che ha effettuato la $P(S)$ deve essere bloccato, e viene inserito in una lista. I processi all'interno della lista sono tutti processi in stato waiting. Processi bloccati. Stanno aspettando che il semaforo sia disponibile.

Nell'operazione di incremento $V(S)$ si fa l'incremento del semaforo. Se il valore del semaforo S ha un valore minore o uguale di 0, allora bisogna togliere un processo dalla lista. Se $S \leq 0$, significa che c'è almeno un processo bloccato sul semaforo.

In questa implementazione, il valore del semaforo indica, in un certo momento, quanti processi sono bloccati sul semaforo stesso.

6.4.2 Synchronization Using Semaphores

```
public class FirstSemaphore {
    public static void main(String args[]) {
        Semaphore sem = new Semaphore(1);
        Worker[] bees = new Worker[5];
        for (int i = 0; i < 5; i++){
            bees[i] = new Worker(sem, "Worker " + (new Integer(i)).
               toString();
        }
        for (int i = 0; i < 5; i++){
            bees[i].start();
        }
    }
}

public class Worker extends Thread {
    public Worker(Semaphore) { sem = s;}
        public void run() {
        while (true) {
            sem.P();
            // in critical section
            sem.V();
            // out of critical section
        }
    }
    private Semaphore sem;
}
```

Fino ad adesso abbiamo sempre parlato di **semafori binari**. Tuttavia ci sono anche **semafori ennari**. I semafori ennari sono semafori che possono raggiungere un valore generico n, dove $n > 1$. L'utilizzo di questo semaforo ha senso nelle situazioni in cui esiste una risorsa condivisa, la quale può essere condivisa da un numero di processi che è maggiore di 1.

6.4.3 Bounded-Buffer Problem

Il Bounded-Buffer corrisponde al problema dei produttori e consumatori. Ora analizzeremo tale problema e cercheremo di trovare una soluzione attraverso l'utilizzo dei semafori. Il Bounded-Buffer è oggetto Java con due metodi, **enter()** e **remove()**. Il primo invocato dai produttori e il secondo invocato dai consumatori.

```
/* CODICE PRODUTTORE */
```

```
while(true){
    /* produce an item and put in nextProduced */
    while(count == BUFFER_SIZE){
        ;// do nothing
        buffer[in] = nextProduced;
        in = (in + 1) % BUFFER_SIZE;
        count++;
    }
}
```

Attraverso i semafori vediamo come eliminare tutte le forme di interferenza relative ai Bounded-Buffer.

Il codice del metodo enter() e del metodo remove() sono codici in cui viene manipolata una risorsa condivisa, cioè il buffer. Si potrebbe pensare di introdurre un semaforo binario **mutex** in modo da effettuare una operazione $mutex.P$ prima di entrare nel buffer e una operazione $mutex.V$ una volta usciti dal buffer. L'idea del semaforo è simile a quella di una chiave. Chiunque voglia agire sul buffer, deve impossessarsi della chiave. Il produttore o consumatore cercano di prendere la chiave attraverso l'operazione $mutex.P$. Quando la P ha successo, quindi quando la chiave viene acquisita, si può agire sul buffer. Una volta terminata l'esecuzione viene rilasciata la chiave attraverso l'operazione $mutex.V$ e un altro processo potrà eseguire.

È sufficiente un meccanismo di questo genere per risolvere il problema di interferenza? Una soluzione di questo tipo porta a una situazione di **stallo**.

Si supponga di avere il buffer pieno e il semaforo uguale a 1. Siccome il buffer è pieno, è inutile che intervenga il produttore. È necessario che intervenga il consumatore. Se, però, arriva lo stesso un produttore che vuole impossessarsi del buffer e riesce ad acquisire la chiave, ciò ci porta in una situazione di stallo perché effettuando l'operazione **mutex.P**, il valore del semaforo S è stato decrementato a 0 e il while() all'interno del metodo enter() risulta **true**, in quanto count == buffer_size.

Così facendo, il produttore rimane bloccato sul while e non succede nient'altro. Il consumatore è l'unico che potrebbe sbloccare il produttore dal while, poiché è l'unico che può decrementare il count. Però, quando arriva un consumatore, la prima operazione che fa è quella di cercare di impossessarsi della chiave per accedere al buffer, ma essa è in possesso dal produttore, il quale è bloccato sul while. Quindi possiamo dire che il produttore non rilascia mai la chiave e, di conseguenza, non potrà mai essere sbloccato poiché la chiave rimane inaccessibile a tutti gli altri.

È necessario utilizzare un meccanismo più sofisticato. Bisogna utilizzare dei semafori per implementare il meccanismo di bloccaggio tramite il while. Non si utilizzerà più il while ma si useranno dei semafori ennari per gestire questa situazione.

Nel codice seguente vediamo tre semafori: il semaforo **mutex**, **empty** e **full**.

Il semaforo **empty** indica quante sono le caselle libere sul buffer, mentre il semaforo **full** indica quante sono le caselle piene. Questi due nuovi semafori prendono il posto della variabile count, che precedentemente teneva conto degli elementi all'interno del buffer e, essendo semafori, implementano anche il meccanismo di bloccaggio per i produttori e consumatori sul buffer stesso.

```
1  public class BoundedBuffer {
2      public BoundedBuffer() {
3          // buffer is initially empty
4          count = 0;
5          in = 0;
6          out = 0;
7          buffer = new Object[BUFFER_SIZE];
8          mutex = new Semaphore(1);
9          empty = new Semaphore(BUFFER_SIZE);
10         full = new Semaphore(0);
11     }
12     public void enter() {
13         empty.P();
14         mutex.P();
15
16         // add an item to the buffer
17         buffer[in] = item;
18         in = (in + 1) % BUFFER_SIZE;
19         mutex.V();
20         full.V();
21     }
22     public Object remove() {
23         full.P();
24         mutex.P();
25
26         // remove an item from the buffer
27         Object item = buffer[out];
28         out = (out + 1) % BUFFER_SIZE;
29
30         mutex.V();
31         empty.V();
32
33         return item;
34     }
35
36     private static final int BUFFER_SIZE = 2;
37     private Semaphore mutex;
38     private Semaphore empty;
```

```
39      private Semaphore full;
40      private int in, out;
41      private Object[] buffer;
42  }
```

Analizziamo ora il metodo enter(). La prima cosa che succede all'interno del metodo è l'operazione **empty.P()**. Si verifica se un produttore può inserire un nuovo elemento nel buffer. Nel caso in cui il buffer sia pieno, quindi il numero di caselle vuote è uguale a zero, l'operazione $P()$ diventa bloccante. Quando empty() raggiunge il valore 0, questa operazione è bloccante. Il produttore si blocca su quella precisa empty() e rimarrà bloccato fin tanto che empty() non verrà incrementata. Nel caso in cui il buffer non sia pieno, il produttore inserirà il proprio elemento nella cella **in**, incrementerà la variabile **in** con l'aritmetica modulo e, infine, effettuerà un incremento sulla variabile full, attraverso l'operazione **full.V()**.

Il metodo remove() svolge, circa, le stesse operazioni del metodo enter(). Il consumatore inizialmente cerca di decrementare full. Nel caso in cui il buffer sia vuoto, tale operazione diventa bloccante poiché non ci sono elementi presenti sul buffer. Il numero di celle presenti sul buffer è uguale a zero. Se l'operazione $full.P()$ ha luogo, il consumatore può svolgere il proprio lavoro. Preleva dalla casella out, incrementa la variabile out attraverso l'aritmetica modulo e, infine, viene fatto l'incremento su $empty()$. Il consumatore, avendo prelevando un elemento dal buffer, ha aumentato di 1 il numero di celle libere del buffer.

Il semaforo mutex(), in questo caso, serve perché quando c'è una sezione critica è bene proteggerla con un semaforo adeguato. I semafori empty() e full() servono a disciplinare l'ingresso sul buffer e il semaforo mutex() serve per assicurarsi che le manipolazioni sul buffer avvengano in maniera mutual exclusion.

6.4.4 Readers-Writers Problem

Prima di parlare del problema Readers-Writers, facciamo una distinzione tra due variabili:

- **variabili condivise solo in lettura:** Si supponga, inizialmente, di avere una variabile $x = 5$ e due processi P_1 e P_2. Il processo P_1 legge la variabile x creando una variabile $y = x$ e il processo P_2 legge anch'esso la variabile x creando una variabile $z = x$.

  Si ha una condivisione della variabile x ma è una condivisione solo in lettura. Ci interessa il valore di x ma solo in lettura e non in scrittura. I due processi utilizzano la variabile x ma non la modificano.

  In questo caso una contemporaneità da parte di P_1 e P_2 nell'accesso a x non causa nessun problema.

- **variabili condivise solo il scrittura:** Si supponga, inizialmente, di avere una variabile $x = 5$ e due processi P_1 e P_2. Il processo P_1 scrive sulla variabile x modificandone il valore. Dopo questa operazione $x = y$. Allo stesso modo anche il processo P_2 scrive sulla variabile x, facendo diventare $x = z$.

  In questa caso x è usato in pura scrittura. Ci possono essere problemi? Dipende dalla rappresentazione interna. Per rappresentazione in interna di x si intende se le operazioni di scrittura possano avvenire in maniera atomica oppure no.

  Vediamo due esempi per le variabili condivise solo in scrittura:

6.4.4.1 Esempio 1: Caso Boolean

Supponiamo che x è sia valore boolean settato inizialmente a true. Dati due processi concreti P_1 e P_2 dove:

- P_1 setta $y = false$
- P_2 setta $z = true$

Se il processo P_1 effettua un'operazione di scrittura $x = y$ e il processo P_2 effettua l'operazione $x = z$ e, infine, viene effettuata una stampa finale delle tre variabile x, y, e z, ci potremmo chiedere quali potrebbero essere i valori possibili stampati.

Il valore della potrebbero essere:

- **x** potrebbe essere sia true che false;
- **y** è false;
- **z** è true.

Dal momento che la variabile x può avere valori sia true che false, possiamo dire di essere in una situazione di **non determinismo.** Una situazione di questo tipo indica che, quando viene eseguito il codice concorrente, si possono avere più sequenze di esecuzioni possibili.

Nonostante ciò, una situazione di questo tipo non crea problemi. Siccome x è una variabile boolean, quindi ha una rappresentazione interna molto semplice, la scrittura di x avviene in un colpo solo, cioè c'è un'istruzione macchina unica in cui questa scrittura avrà luogo.

6.4.4.2 Esempio 2: Caso reali

Supponiamo ora che le variabili x, y e z siano delle variabili reali (float). Facendo riferimento all'esempio precedente ma con il nuovo tipo di variabili, si inizia ad avere

una situazione un po' più complicata. Essendo le variabili in questione **variabili real**, in generale, la rappresentazione in memoria di queste variabili sta più su celle. La rappresentazione del numero reale prende più **byte**.

Nel caso in cui per la scrittura della variabile x vengano utilizzate due celle di memoria, sono necessarie **due istruzioni macchina**. Una prima istruzione per modificare la prima cella e una seconda istruzione per modificare la seconda cella. Concludendo possiamo dire che, in una situazione di questo tipo, la scrittura non è più atomica. Si rischia una situazione di **interleaving.**

Ora possiamo iniziare a parlare del problema Readers-Writers. Suddividiamo il problema in tre punti:

1. **Database in possesso di un lettore.**
2. **Database in possesso di uno scrittore.**
3. **Database libero.**

I Readers-Writers sono dei thread che, quando vengono creati, hanno accesso a un database chiamato **db**. I Readers-Writers effettueranno delle operazioni, rispettivamente, di lettura e scrittura.

La classe Reader estende la classe Thread. Nel costruttore della classe viene utilizzato l'oggetto **db**. L'oggetto db funge da arbitro. Serve per garantire l'accesso corretto al database. Il database ha due metodi, **startRead()** e **endRead()** che vengono invocati dal lettore prima di iniziare e alla fine della lettura.

```
1  public class Reader extends Thread {
2
3      public Reader(Database db) {
4          server = db;
5      }
6
7      public void run() {
8          int c;
9          while (true) {
10             c = server.startRead();
11             // now reading the database
12             c = server.endRead();
13     }
14  }
15  private Database server;
16 }
```

Il Writer è un thread analogo al Reader. Anche il thread dello scrittore viene creato con un parametro del database e invoca i suoi due metodi, **stardWrite()** e **endWrite()**.

Si lo scrittore che il lettore, prima di effettuare operazioni sul database, invocano rispettivamente i propri metodi.

```
1  public class Writer extends Thread {
2
3      public Writer(Database db) {
4          server = db;
5      }
6
7      public void run() {
8          while (true) {
9              server.startWrite();
10             // now writing the database
11             server.endWrite();
12         }
13     }
14  private Database server;
15 }
```

L'aspetto importante è comprendere come funziona l'arbitro del database. Egli da l'accesso al database.

Analizziamo i metodi all'interno del codice:

- **startWrite():** Il semaforo binario **db** funge da lock. Serve per avere accesso al database. Uno scrittore che vuole entrare nel database deve effettuare una **P()** su db.

  Nel caso in cui db sia libera, lo scrittore, attraverso l'operazione dp.P(), può acquisire il lock.

  Il semaforo db è libero quando ha un valore > 1. È necessario che sia maggiore di 1 per poter effettuare il decremento perché l'operazione P() porta a decrementare il semaforo stesso e, nel caso in cui il semaforo abbia valore 0, una operazione P() sarebbe bloccante.

- **endWrite():** L'operazione di endWrite() serve per mettere a disposizione nuovamente il lock del database. Questo mettodo effettua una V() sul database dp, quindi effettua un incremento.

- **startRead():** Il metodo startRead() è analogo a quello dello scrittore, quindi si può pensare di avere una db.P(). Una soluzione di questo tipo però ci permette di avere un solo lettore per volta sul database. Tuttavia, per avere un sistema più efficiente, è necessario avere più lettori per volta sul database.

Il momento in cui il lock da parte dei lettori viene acquisito è un momento particolare. Il primo lettore che arriva acquisisce il lock per tutti i lettori. I lettori successivi possono entrare senza doversi preoccupare di acquisire il lock perché è già stato acquisito.

Attraverso la variabile readerCount si tiene conto del numero di lettori che sono, in un certo momento, presenti sul database. Nel caso in cui il readerCount sia uguale a 1, significa che è entrato nel database il primo lettore. Egli deve acquisire il lock e, infatti, effettuare una P() su db.

- **endRead():** Come per il metodo startRead(), anche l'endRead() è analogo a quello dello scrittore. Si può pensare di avere una db.V(). Però, anche in questo caso, ci sarebbe un solo lettore.

  La soluzione ottimale è quella in cui l'ultimo lettore che esce la database rilascia il lock. Anche in questo caso abbiamo la variabile readerCount. Quando readerCount è uguale a 0 viene effettuata una V() su db, in modo che l'ultimo lettore rilascia il lock.

Nei metodi startRead() e endRead() è presente anche una variabile mutex. Tale variabile deve essere acquisita dai lettori prima di effettuare il codice poiché, all'interno del codice, vengono manipolate delle variabili condivise con altri lettori. Nel caso in cui non venisse utilizzata la variabile mutex potrebbero crearsi dei problemi.

```
1  public class Database
2  {
3      public Database() {
4          readerCount = 0;
5          mutex = new Semaphore(1);
6          db = new Semaphore(1);
7      }
8
9      public int startRead() {
10         mutex.P();
11         ++readerCount;
12
13         // if I am the first reader tell all others
14         // that the database is being read
15         if (readerCount == 1)
16             db.P();
17
18         mutex.V();
19         return readerCount;
20     }
21
```

```
22      public int endRead() {
23          mutex.P();
24          --readerCount;
25
26          // if I am the last reader tell all others
27          // that the database is no longer being read
28          if (readerCount == 0)
29          db.V();
30
31          mutex.V();
32          return readerCount;
33      }
34
35      public void startWrite() {
36          db.P();
37      }
38
39      public void endWrite() {
40          db.V();
41      }
42
43   private int readerCount; // number of active readers
44   Semaphore mutex; // controls access to readerCount
45   Semaphore db; // controls access to the database
46  }
```

6.4.5 Sincronizzazione tra processi

Vediamo ora dei problemi di sincronizzazione tra processi dove è possibile utilizzare i semafori. Per comprendere a pieno questi problemi utilizzeremo degli esempi.

6.4.5.1 Esempio 1

Supponiamo di avere due processi P_1 e P_2. Il processo P_1 cerca di stampare $print(A)$ e il processo P_2 cerca di stampare $print(B)$.

La domanda che ci si pone è: se lanciamo in contemporanea i due processi, quale sarà la stampa che viene stampata per prima? Le stampe possibili sono **AB** o **BA**. Se imponiamo che l'unica stampa possibile sia **AB**, che cosa dobbiamo fare? Come possiamo assicurarci che la prima lettera stampata sia A e la seconda sia B attraverso i semafori binari?

Come visto precedentemente, sappiamo che quando un semaforo binario è settato a 0 e si effettua una operazione $P()$, essa risulta bloccante. L'idea in questo caso è di

utilizzare un semaforo $S = 0$ e effettuare un'operazione $P(S)$ sul processo P_2 prima della stampa di $print(B)$. Successivamente aggiungiamo una $V(S)$ dopo la stampa di $print(A)$ perché vogliamo che la chiave venga messa rimessa a disposizione.

P_1	P_2
	$P(S)$
$print(A)$	$print(B)$
$V(S)$	

6.4.5.2 Esempio 2

Si supponga di avere due processi P_1 e P_2. Nel caso in cui il processo P_1 voglia effettuare una $print(A)$ e una $print(C)$ e il processo P_2 una $print(C)$ e una $print(D)$, che cosa può succedere? Se si lanciano i due processi in parallelo, le stampe possono venire in ordine arbitrario. Supponiamo, però, di volere la stampa **ABCD**. Come possiamo combinare i due processi attraverso i semafori binari per creare la stampa di questo tipo?

Inizializziamo due semafori $S = 0$ e $T = 0$. È necessario bloccare il processo P_2 per poter stampare $print(A)$, quindi la prima operazione da effettuare è aggiungere un $P(S)$ nel processo P_2, prima della stampa $print(B)$. Dopo aver effettuato la stampa di A e incrementato il semaforo S attraverso l'operazione $V(S)$ è necessario introduttore il semaforo T. Bisogna aggiungere una $P(T)$ nel processo P_1 in modo da bloccare $print(C)$ per stampare $print(B)$ e, successivamente, bisogna aggiungere una $V(T)$ nel processo P_2 per incrementare il semaforo T. Infine, bisogna aggiungere una $V(S)$ dopo la stampa $print(C)$ per incrementare il semaforo S per stampare la $print(D)$.

P_1	P_2
	$P(S)$
$print(A)$	$print(B)$
$V(S)$	$V(T)$
$P(T)$	$P(S)$
$print(C)$	$print(D)$
$V(S)$	

6.4.5.3 Esempio 3

Si supponga di avere due processi P_1 e P_2 e un loop, (a livello di codice i loop potrebbe essere inteso come un while), nei due programmi per stampare $print(A)$ nel processo P_1 e $print(B)$ nel processo P_2. Attraverso il loop vogliamo ottenere una sequenzializzazione perfetta fra le stampe che i due processi cercano di fare **(AB, AB, ...)**.

P_1	P_2
loop	loop
$P(T)$	$P(S)$
$print(A)$	$print(B)$
$V(S)$	$V(T)$

Dati due semafori $S = 0$ e $T = 1$, inizialmente l'unico processo che può partire è il processo P_1. Il processo P_2, se cerca di eseguire il decremento, rimane bloccato perché il semaforo $S = 0$. P_1 riesce a superare l'istruzione $P(T)$, quindi il semaforo T viene decrementato a 0 e si stampa $print(A)$. Dopo la stampa di A viene fatto un incremento su S che porta il semaforo $S = 1$ e il processo P_1 rientra nel loop. In questa situazione l'unico processo che può avanzare è il processo P_2 perché il semaforo $T = 0$ e la $P(T)$ all'inizio del processo P_1 risulta bloccante. P_2 avanza perché il semaforo $S = 1$.

Il processo P_2 riuscirà a completare la propria $P(S)$ portando $S = 0$ e stampando la $print(B)$. Dopo la stampa di B l'unica possibilità è quella di continuare su P_2, dove viene eseguita la $V(T)$, portando il semaforo $T = 1$. Ora il processo P_1, torna runnable.

A questo punto siamo tornati nella situazione iniziale, i semafori hanno gli stessi valori partenza, però, è già stata stampata una sequenza **AB**.

6.4.5.4 Esempio 4

Si supponga di avere due processi P_1 e P_2 in loop. In questo caso l'occorrenza di stampa finale tra A e B dev'essere arbitraria, ovvero si accetta del non determinismo arbitrario.

In ogni istante:

$$|\#(A) - \#(B)| \leq 1 \tag{6.1}$$

In altre parole significa inizialmente la stampa può avvenire con una **A** o con una **B**, ma il qualunque momento la differenza fra il numero di A e il numero di B dev'essere ≤ 1. Ciò significa che se la stampa è iniziata con una A, necessariamente dopo occorre una B, e viceversa. Bisogna sincronizzare i due processi sui loop.

In una esecuzione possibile di questo genere, settando due semafori S e T a 0, si entra nel loop e, inizialmente, è possibile stampare sia A che B. Supponiamo che si inizi stampando A. A questo punto è possibile continuare sul processo P_1 stampando di nuovo una A o continuare sul processo P_2 stampando una B. Tuttavia il processo P_1 si fermerà quando dovrà eseguire la $P(S)$ poiché il semaforo $S = 0$, dunque, per poter stampare un'altra A è necessaria la $V(S)$ nel processo P_2 dopo la stampa di B. Concludendo, quindi, dopo la stampa di una A è assolutamente necessaria la stampa di una B. Fin tanto che il processo P_2 non ha eseguito la $V(S)$, il processo P_1 si trova bloccato sulla $P(S)$.

Alla fine di P_2 entrambi i processi possono rientrare nel loop, decrementando i semafori riportandoli a 0. Questa è un'esecuzione corretta.

P_1	P_2
loop	loop
$print(A)$	$print(B)$
$V(T)$	$V(S)$
$P(S)$	$P(T)$

6.5 Java Synchronization

Molti linguaggi di alto livello hanno costrutti per gestire la concorrenza. Sono costrutti che raccolgono l'idea dei **monitor.** I costrutti di sincronizzazione forniti da Java permettono una gestione della concorrenza molto più immediata e semplice di quella possibile utilizzando i semafori. Qual è l'idea della gestione della concorrenza in Java?

Introduciamo la parola **synchronized**. Essa può essere aggiunta come etichetta ai metodi di un oggetto con lo scopo di gestire una risorsa condivisa implementata da quell'oggetto. L'idea di base è che ogni oggetto abbiamo un **lock** associato ad esso. Il lock è una specie di chiave di cui bisogna impossessarsi per operare sul quel determinato oggetto.

Vediamo come esempio il metodo **enter()** del bounder buffer. Quando un metodo che viene chiamato **synchronized**, prima di poter invocare il metodo enter() sull'oggetto bounded buffer, i soggetti desiderati, i produttori nel nostro caso, devono prima impossessarsi del lock per poter eseguire fisicamente le operazioni descritte nel metodo. Il lock è una chiave unica e c'è un solo thread in java per volta che può impossessarsi del lock di un certo oggetto e, quindi, c'è un solo thread che sarà in grado liberamente di agire sull'oggetto.

```
1 public synchronized void enter( Object item ){
2     while( count == BUFFER_SIZE ){
3         Thread.yield();
4     }
5     ++count;
6     buffer[ in ] = item;
7     in = ( in + 1 ) % BUFFER_SIZE;
8 }
```

Se abbiamo un thread P_1 che vuole eseguire il metodo enter(), inizialmente egli dovrà verificare che il lock sia disponibile e, in caso affermativo, lo potrà acquisire per operarare sul bounded buffer. Nel caso in cui, in questo momento, arrivi un altro thread P_2 che vorrebbe anch'esso eseguire il metodo enter(), egli non può farlo poiché il lock è occupato da P_1. Il thread P_2 deve aspettare che P_1 termini l'esecuzione. Durante questo periodo il thread P_2 viene messo in un insieme chiamato **entry set**, il quale è associato all'oggetto in questione cioè al bounded buffer, con l'idea che P_2 rimanga in attesa fin tanto che

il lock non diventa disponibile. Il lock diventerà disponibile quando P_1 ha terminato il codice del metodo enter().

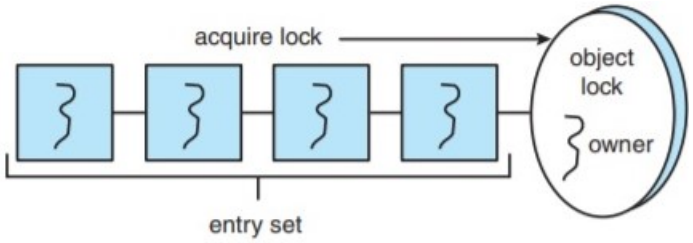

Figura 6.3: Entry Set

Nella Figura 6.3 viene illustrata la situazione. Vediamo che c'è un lock associato a un thread, il quale sta eseguendo un metodo etichettato synchronazied su un determinato oggetto. Ci sono anche altri thread che hanno cercato di eseguire un metodo synchronized dell'oggetto ma, non trovando il lock disponibile, sono stati messi nell'entry set. Quando il thread attualmente in esecuzione terminerà il codice del metodo synchronized, il lock sarà libero. A questo punto ci sarà competizione tra gli altri thread per chi riuscirà ad impossessarsene. Non si riesce a stabilire chi tra i thread se ne impossesserà.

Tuttavia, il codice visto sopra non è del tutto corretto per far funzionare un procedimento di questo genere perché, nel caso in cui un produttore, cioè un thread che ha invocato il metodo synchronized enter(), arriva ed esegue il metodo enter() e si trova bloccato sul while() perché il buffer è pieno, purtroppo l'operazione **yield()** non rilascia il lock. Dunque ci si ritrova nella situazione in cui il produttore rimane in attessa sul while senza rilasciare il lock, quindi non da la possibilità a un altro thread di intervenire e eseguire operazioni sul buffer. In questo caso sarebbe necessario che un consumatore effettuasse un'operazione di remove() sul buffer in modo da liberare un posto, ma ciò non può accadere. Sono necessarie delle operazioni che permetto di dire al thread di rilasciare al lock e di passare in stato waiting, quindi di essere bloccati in attesa della condizione di azione sul buffer per loro diventi vera.

Questo ci porta a introdurre due primitive: **wait()** e **notify()**. Il produttore che entra nel while, quindi che effettua il metodo enter() con il buffer pieno, esegue una wait(). Ciò comporta all'inserimento del produttore in un nuovo insieme chiamato **wait set**. Precedentemente avevamo introdotto l'**entry set**, cioè quell'insieme di thread che vorrebbero eseguire un metodo synchronized dell'oggetto in questione, ma non sono ancora riusciti a farlo. Il wait set, invece, è un insieme di thread che avevano precedentemente acquisito il lock, ma poi, all'interno del metodo synchronized che avevano invocato, si sono trovati a eseguire una wait(). Se hanno eseguito una wait(), significa che non più potevano operare sulla risorsa e quindi hanno preferito rilasciare il lock. I thread nel wait set sono una specie di thread in stato waiting. Stanno aspettando un qualcosa che

permetta loro di operare. Nel caso del produttore che trova il buffer pieno, il qualcosa è la liberazione di alcune caselle del buffer.

La notify() è il contrario della wait(). È necessaria per sbloccare i thread che si trovano all'interno del wait set. Quando un thread esegue una notify(), viene preso dal wait set e messo nell'entry set.

```
public synchronized void enter( Object item ){
    while( count == BUFFER_SIZE ){
        try{
            wait();
        } catch( InterruptedException e ){ }
    }
    ++count;
    buffer[ in ] = item;
    in = ( in + 1) % BUFFER_SIZE;
    notify();
}
```

Tuttavia, dal momento che il wait set è popolato sia da produttori che consumati, attraverso un'operazione di notify() non si è in grado di dire chi voler svegliare. Per questo motivo Java mette a disposizione un'altra primitiva chiamata **notifyAll()**. Attraverso questa operazione tutti i thread che sono all'interno del wait set vengono svegliati e spostanti nell'entry set. Ovviamente, la scelta di utilizzare una notify() o una notifyAll() è significativa, soprattutto dal punto di vista di memoria.

Vediamo l'esempio del Reader-Writer per comprendere a meglio questa differenza. Sappiamo che esiste un database che può essere usato sia dai lettori che dagli scrittori. Ci sono due tipi di thread che utilizzano il database, i reader solo in lettura e i writer solo in scrittura. Quando un writer utilizza il database, il suo utilizzo è esclusivo. Viceversa, quando un reader utilizza il database, è possibile che vi siano anche altri lettori sul database.

Vediamo di seguito il codice Java della classe Database:

```
public class Database {

    public Database() {
        readerCount = 0;
        dbReading = false;
        dbWriting = false;
    }

   public synchronized int startRead() {
        while (dbWriting == true) {
            try {
                wait();
```

```
13              }
14              catch (InterruptedException e) { }
15              ++readerCount;
16              if (readerCount == 1)
17                  dbReading = true;
18              return readerCount;
19          }
20      }
21
22      public synchronized int endRead() {
23      --readerCount
24      if (readerCount == 0)
25          dbReading = false;
26          notifyAll();
27      return readerCount;
28      }
29
30      public synchronized void startWrite() {
31          while (dbReading == true || dbWriting == true)
32              try {
33                  wait();
34              }
35              catch (InterruptedException e) { }
36              dbWriting = true;
37          }
38      }
39
40      public synchronized void endWrite() {
41          dbWriting = false;
42          notifyAll();
43      }
44
45      private int readerCount;
46      private boolean dbReading;
47      private boolean dbWriting;
48  }
```

A riga 45 e 46 vengono inizializzate due variabili booleane che servono per gestire le tre situazioni possibili.

1. Database libero;
2. Database in possesso del reader;
3. Database in possesso del writer.

I metodi sono dichiarati synchronized perché sono metodi all'interno dei quali vengono manipolate delle risorse condivise. Le risorse condivise manipolate sono le variabili inizializzate in fondo alla classe: readerCount e le variabili booleane. Analizziamo i metodi:

- **startRead():** l'aspetto più importante per un lettore è quella di comprendere se il database è in possesso degli scrittori oppure no. Le variabli *dbReading* e *dbWriting* inizialmente sono settate a false e indica che il database è libero. Se *dbReading* viene successivamente settata a true indica che il database è in possesso degli scrittori. Viceversa, se *dbWriting* viene settata a true indica che il database in possesso dei lettori.

  L'aspetto significativo per i lettori è verificare che *dbWriting* sia uguale a true. Nel caso in cui lo sia, il lettore deve aspettare e invoca l'operazione **wait().** Il lettore in questione viene messo sull'insieme wait set dell'oggetto database.

  Se si riese a superare il while, significa che *dbWriting* è uguale a false e non ci sono scrittori su database. A questo punto si è sicuri che si può andare avanti e si può iniziare a leggere. Se **readerCount == 1**, viene settato *dbReading = true*. Il booleano *dbReading* essendo uguale a true significa che è il database è in possesso dei lettori.

- **endRead():** ogni lettore che ha finito la propria lettura sul database invoca il metodo endRead(). Nel caso in cui il lettore uscente sia l'ultimo di tutti i lettori, egli esce e segnalata che il database è nuovamente libero. Altrimenti, il lettore in questione esce senza fare nulla e un altro lettore prenderà possesso del database. La variabile **readerCount** è fondamentale.

- **starWriting():** uno scrittore arriva sul database se e solo se esso è completamente libero. Se *dbWriting* o *dbReading* è uguale a true, lo scrittore dev'essere stoppato, quindi effetta una wait(). Altrimenti, se entrambe sono false, lo scrittore può procedere e setterà *dbWriting = true*.

- **endWriting():** quando la scrittura è terminata, *dbWriting* viene settato a false e viene utilizzato effettuata una notifyAll().

Facciamo un altro esempio:
Joe e John, due fratelli, condividono un conto in banca. Vogliamo visualizzare il saldo, depositare e ritirare del denaro. Vediamo la classe Account di Java che permette questi metodi: Il metodo **getBalance()** permette di sapere quanto denaro c'è attualmente nel conto. Il metodo **deposit(double amount)** permette di aggiungere al conto l'amount. Il metodo **withdraw(double amount)** permette di ritirare e eliminare il parametro amount dal conto. L'if all'interno del metodo serve per fare in modo che il conto non vada mai in rosso.

```
class Account {
    private double balance;
    public Account(double initialDeposit) {
        balance = initialDeposit;
    }
    public double getBalance() {
        return balance;
    }
    public void deposit(double amount) {
        balance += amount;
    }
    public void withdraw(double amount) {
        if ( balance >= amount ) { balance -= amount; }
    } // no negative balance allowed
}
```

Una implementazione di questo genere potrebbe dare dei problemi? Due soggetti che possono invocare in contemporanea i metodi, possono dare dei problemi di interferenza?

Un problema di interferenza è dovuto al fatto che la variabile **balance** è condivisa. Gli incrementi e decrementi della variabile possono causare problemi. Non essendo atomica questa operazione, quindi richiede più istruzioni a livello macchina, potrebbe causare delle problematiche. Un altro problema di interferenza si verifica nel caso in cui avviene un content switch dopo l'if nel metodo **withdraw()**. Anche il metodo **getBalance()** può creare dei problemi in quanto la variabile balance può essere modificata. Ciò potrebbe creare dei problemi se uno dei due fratelli effettua tale metodo mentre l'altro sta modificando il valore della variabile stessa.

L'unico modo per risolvere questi problemi di interferenza è attraverso l'attributo **synchronized.** Grazie al lock di synchronized si riesce a eliminare queste problematiche.

Capitolo 7

Memory Management

In questo modulo andremo a parlare della gestione della memoria centrale. La RAM è importante perché è la memoria che dialoga direttamente con il processore. Un processo per essere eseguito dev'essere caricato su memoria centrale. La RAM è disponibile in quantità inferiore rispetto alla memoria secondaria, va gestita con cura.

Uno degli obiettivi di questo modulo, infatti, è fare in modo che la memoria centrale sia gestita nella maniera migliore possibile, permettendo un livello di multiprogrammazione molto alto. Bisogna fare in modo che la memoria centrale non vada sprecata e che sia assegnata a processi importanti. Un altro importante obiettivo è rendere l'accesso alle risorse della memoria centrale nel modo più rapido possibile.

Parleremo anche di:

- Swapping;
- Contiguous Memory Allocation;
- Paging;
- Structure of the Page Table.

Quando abbiamo introdotto il concetto di multiprogrammazione, avevamo visto l'organizzazione della memoria centrale. La RAM è organizzata in blocchi dove, il blocco più in alto viene assegnato al sistema operativo, e negli altri blocchi vengono allocati i processi. Avevamo menzionati i **base register** e **limit register**, che rappresentano gli indirizzi della prima e ultima cella di memoria assegnata a un certo processo. Quando la CPU manda in esecuzione un generico processo P, produce un indirizzo di memoria x. Per garantire la protezione della memoria per il processo P, l'indirizzo x deve essere contenuto tra il base e limit register della zona di memoria dedicata al processo P. Avevamo visto anche la Memory Managament Unit (MMU), la quale, a livello hardware, implementa la circuiteria per effettuare controlli, per esempio il controllo dell'indirizzo x.

7.1 Swapping

Quando si parla della gestione della memoria centrale, l'aspetto più importante da comprendere è la differenza tra **indizzi logici** e **indirizzi fisici**. Vediamo di comprendere questi due concetti:

Dato un **programma** composto da linee di codice, variabili e istruzioni, quando viene mandato in esecuzione viene creato un **processo.** All'interno della sua struttura, un processo contiene del codice eseguibile, una parte di dati e una parte di heap/stack che serve per gestire la dinamicità del processo stesso. Se durante l'esecuzione del programma si hanno delle variabili globali x e y, queste variabili saranno destinate ad avere uno spazio di riferimento nella zona **dati** nella struttura del processo. Infine si ha la **RAM**. La RAM è suddivisa in celle numerate in ordine crescente. Un processo per essere eseguito deve essere caricato memoria centrale. Quindi si può pensare che le variabili x e y vengano caricate su delle celle della RAM. Per esempio la variabile x può essere caricata sulla cella m e la variabile y sulla cella $m + 1$.

Dunque possiamo dire che il processo è un'**entità virtuale**, nel senso che non ha una sua esistenza concreta fisica come la memoria centrale. È un'entità virtuale divisa in celle, come la RAM. La suddivisione in celle è importante perché il processo deve essere caricato su memoria centrale, quindi è necessario che al suo interno ci sia una strutturazione del tutto simile in modo da effettuare il caricamento.

Le operazioni che le variabili di un programma possono fare sul processo sono le operazioni di **load** e di **write**. Una variabile per essere caricata dal programma al processo deve effettuare un'operazione di load. I nomi delle variabili x e y all'interno del programma sono **riferimenti simbolici**. Tuttavia tali riferimenti non vengono mantenuti quando si passa dal programma al processo e dal processo alla memoria centrale. Le variabili x e y quando vengono caricate sulle celle del processo e della RAM prendono dei "nomi" differenti. Per esempio, le celle del processo possono essere identificate come k e $k + 1$ e le celle della memoria centrale m e $m + 1$. Quindi possiamo dire che c'è una relazione differente tra **x**, **k** e **m**. L'associazione di indirizzi di istruzioni e dati agli indirizzi di memoria può avvenire in tre fasi:

- **Compile time:** Se in fase di compilazione si conosce dove risiderà il processo in memoria, è possibile generare il **codice assoluto**. Ad esempio, se un processo utente risiederà a partire della posizione **r**, il codice generato dal compilatore inizierà da quella posizione e si estenderà da lì. Lo svantaggio di questa procedura è che se in un secondo momento la posizione di partenza cambia, sarà necessario ricompilare il codice.

- **Load time:** Se non è noto in fase di compilazione dove risiederà il processo in memoria, il compilatore deve generare un codice trasferibile. In questo caso, l'associazione finale viene ritardata fino al momento del caricamento. Se l'indirizzo

di partenza cambia, è sufficiente ricaricare il codice utente per incorporare questo valore modificato.

- **Execute time:** quando il processo in questione viene caricato sulla memoria centrale, i riferimenti interni rimangono quelli iniziali. Quando viene eseguito il load(k) all'interno del processo, la CPU chiede di accedere alla cella k. Per fare ciò è necessaria un'operazione, r, per accedere alla cella m di RAM, dove $m = k + r$.

Quando si parla di **indirizzo fisico** si tratta dell'indirizzo di una cella di memoria sul quale si vuole operare. Quando si parla di **indirizzo logico** si fa riferimento all'indirizzo prodotto dalla CPU.

Per **swapping** si intende lo spostamento di un processo da memoria centrale a memoria secondaria. Quando un processo rimane inattivo per un periodo prolungato di tempo, si sposta momentaneamente dalla RAM alla memoria secondaria. Durante la propria vita un processo, può essere spostato più volte. Ovviamente, quando è su memoria secondaria un processo non può essere eseguito, rimane in stand-by.

7.2 Contiguous Memory Allocation

Lo schema della memoria contigua è una tecnica per l'allocazione di memoria. I processi vengono messi su memoria centrale in dei blocchi contigui di memoria. Il processo viene considerato come blocco unico di memoria e viene allocato come tale.

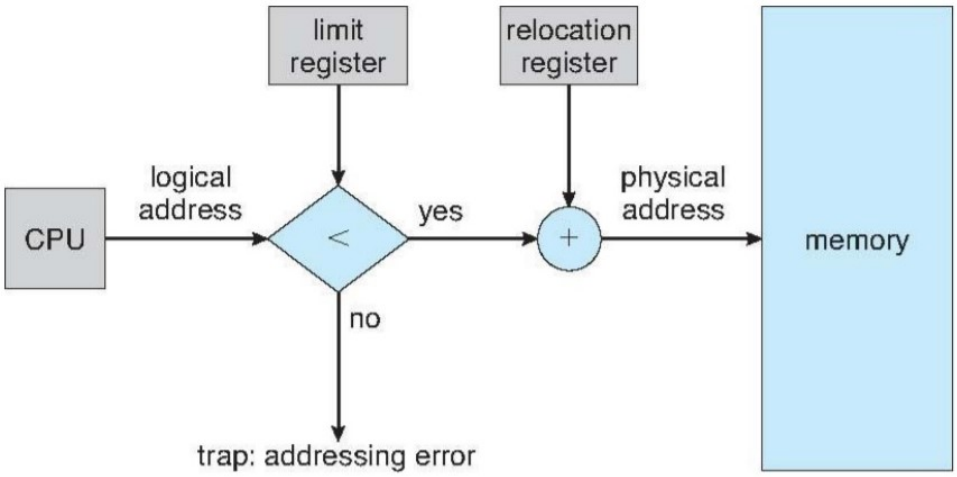

Figura 7.1: Hardware Support for Relocation and Limit Registers

In Figura 7.1 possiamo vedere che cosa deve essere effettuato dal momento in cui la CPU produce un indirizzo logico, quindi stiamo facendo riferimento all'execution time.

Viene cambiato l'indirizzo da logico a fisico attraverso il relocation register e viene effettuato un controllo di protezione poiché occorre verificare che l'indirizzo logico non superi il limit register. Questo è il supporto hardware presente nella MMU che interviene in caso di allocazione contigua.

7.2.1 Dynamic Storage-Allocation Problem

Vediamo ora delle differenti strategie per salvare dei buchi di memoria:

- **First-fit:** attraverso una strategia first-fit, il processo viene allocato nel primo hole che è abbastanza grande per contenerlo. All'interno della first-fit si ha una lista di elementi che si scorre e, quando l'elemento selezionato può essere contenuto all'interno dell'hole corrente, si usa per allocare il processo in questione.

- **Best-fit:** attraverso la best-fit, si cerca di fare attenzione sull'elemento da scegliere. Si scegli in base alla taglia del processo dato, in modo da cercare l'hole che meglio approssima il processo. Ovviamente il buco dev'essere grande almeno quanto la taglia del processo, ma tra tutti i buchi che hanno una grandezza almeno quanto quella del processo si sceglie quella più piccola. In questo modo si cerca di creare un buco rimanente più piccolo possibile, in modo da risparmiare più memoria possibile.

- **Worst-fit:** la worst-fit è la strategia opposta alla best-fit. Quello che si cerca di fare è allocare il processo nel buco più grande possibile. Si sceglie sempre il buco più grande di tutti.

Si è visto che la disciplina migliore, soprattutto nelle macchine moderne, è la **first-fit.**

Il problema principale dell'allocazione contigua è chiamato **frammentazione**. Dopo un po' di utilizzo del sistema, quindi dopo un po' in cui alcuni processi sono terminati e alcuni processi sono partiti, ci si ritrova nella situazione in cui ci sono tanti piccoli buchi all'interno della memoria centrale che non sono utilizzati. L'unica soluzione a questo problema è quella di **compattare**. Per compattare s'intende spostare i processi in questione, quindi i processi attualmente su memoria centrale, in modo da compattare la memoria e lasciare un unico hole che ha come taglia la somma delle taglie dei buchi da cui si era partiti. Durante questo periodo la macchina è inutilizzabile. Non avrebbe senso far partire dei processi perché la memoria centrale è occupata a fare questi spostamenti e non potrebbe gestire un processo in entrata. Il tipo di frammentazione che interviene in questo caso si chiama **frammentazione esterna**. Gli sprechi di memoria avvengono su zone di memorie esterne ai nostri processi, cioè zone di memoria che non erano state allocate ai nostri processi.

7.3 Paging

Nell'allocazione contigua si assegna un area contigua di memoria ai processi, dove non è possibile spezzettare i processi. Con la paginazione è possibile, invece, uno spezzettamento. L'idea è quella di suddividere la memoria centrale in dei blocchi della stessa taglia, chiamati **frame**, e un processo anch'esso diviso in blocchi della stessa taglia, chiamati **pagine**. La taglia delle pagine e dei frame è la stessa. Le varie pagine in cui è diviso un processo possono essere posizionate su dei frame che non sono necessariamente contigui, in modo arbitrario. Nell'allocare le pagine in RAM, bisogna tenere conto degli indirizzi logici e fisici, tramite 3 punti:

1. il numero della pagina;
2. il numero del frame collegato alla pagina;
3. l'**offset** d che indica lo scostamento, cioè dove trovare un determinato dato all'interno del frame.

Ciò porta alla creazione di una **page table**, che indica come i blocchi di un processo siano stati allocati in memoria centrale, associando blocchi di RAM alle pagine del processo. In questo schema non serve compattare perché non serve che i frame siano contigui, superando il problema della frammentazione esterna. Per tradurre un indirizzo iniziale i in uno finale f, si utilizza il **mapping** del frame. Per esempio la pagina numero 1 è mappata nel frame 3 e conoscendo lo scartamento (offset), si avrà:

$$f = 3 * k + d$$

dove, $k = taglia$ e $d = offset$.

Vediamo ora i passi fondamentali per determinare l'impaginazione:

- Inizialmente si determina lo spazio degli *indirizzi logici* di un processo in modo non contiguo. Al processo viene allocata la memoria fisica ogni volta che quest'ultimo è a disposizione;
- Si divide la memoria fisica in blocchi di dimensioni fisse chiamati **frame** della stessa taglia (la dimensione è una potenza del 2, tra 512 byte e 8192 byte);
- Si divide la *memoria logica* in blocchi della stessa dimensione chiamati **pagine**. Il sistema operativo tiene traccia di tutti i frame liberi.
- Per eseguire un processo di dimensioni **n** pagine, è necessario trovare **n** frame liberi e caricare il programma;

- Si imposta una tabella delle pagine per tradurre gli indirizzi logici in indirizzi fisici (**page table**, contenuta in RAM). Per memorizzare la tabella delle pagine è sufficiente memorizzare la tabella degli indirizzi, perché è implicito l'ordine numero delle pagine (si sa che il primo elemento della tabelle, l'uno, corrisponde con la pagina zero.
- Si esegue una frammentazione interna.

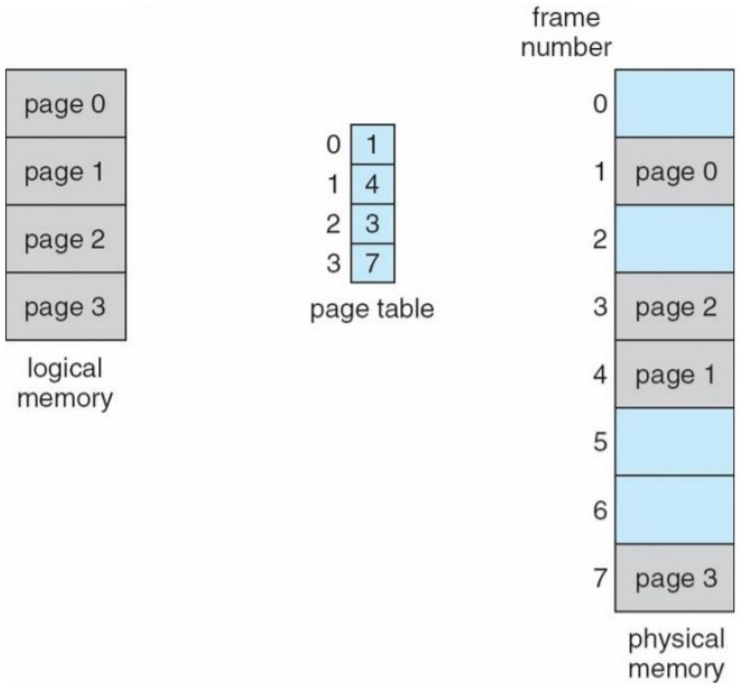

Figura 7.2: Paging model of logical and physical memory

Introduciamo ora il concetto di **taglia**. Si sa che la taglia k dev'essere una potenza del 2, in quanto in numeri manipolati dalla CPU sono in binario (aritmetica binaria). Si prende per esempio che la CPU lavori in numeri decimali, dove in questo caso la tagli k sarebbe una potenza del 10, (per esempio $k = 10^2 = 100$) e anche le pagine sarebbero in blocchi da 100 $(0 - 100 - 200)$. Supponendo di prendere un'informazione all'interno della cella 135, che è contenuta nella cella numero 1 di P, tra 100 e 200, posizionato nel frame 4 della RAM. si avrà una **page table** $1 \rightarrow 4$, dove il 135 di P si troverà shiftato a 435 nella RAM. Per determinare il passaggio da indirizzo logico (135) a fisico (435), si divide 135/100 che da come risultato 1 con resto 35 e, infine, si applica la formula: $4 * 100 + 35 = 435$.

Per ottenere un metodo più rapido con un numero decimale, si tagliano le ultime due cifre di un numero, che sono cifre di scartamento: per esempio il numero 135 in binario,

si taglia alla posizione 100 con scartamento 35, copiando l'offset 35 nel indirizzo fisico e il "4" di 435 viene inserito, consultando la page table che collega la cella "1" di 135 nella corrispondente fisica (in questo caso "4"). In questo modo non ci sono operazioni aritmetiche con un numero binario, risparmiando molto tempo di calcolo. L'indirizzo logico quindi si divide in:

- **Numero pagina:** la prima cifra;
- **Offset:** le due cifre finali.

mentre l'indirizzo fisico si ottiene:

- **Il numero di frame:** la prima cifra è ottenuta consultando la page table che mappa la cella da logica a fisica;
- **Offset:** sono le ultime due cifre ottenute trasportandole dall'indice logico.

Vediamo ora,invece, come determinare la **taglia in binario**. Per determinare la taglia di un numero binario, per esempio 5 (101), si tagliano sempre la prima cifra con il primo numero in binario e le altre due dell'offset, dove ci saranno due posizioni per esso in quanto la taglia è 2^2. Per tradurre l'indirizzo logico 101 in fisico, si trascrive lo stesso scostamento e si consulta la page table per il numero di frame; in questo caso la pagina 1 è all'indirizzo 6, che tradotto in binario sarà 110, con indirizzo finale 11001 = 25 in decimale. Se per esempio si prende in esame l'indirizzo logico \11" etichettato con \l", si rappresenta il numero in binario:

$$8 + 2 + 1 = 1011$$

Si traduce in indirizzo fisico tenendo fisso 11 per l'offset e il numero di frame è tradotto dalla page table in 001, arrivando ad un numero finale (00)111 = 7 in decimale. Il calcolo è dimostrato dalla tabella, Figura 7.3.

In un sistema a 64 bit, il numero di bit a disposizione sarebbe $2^n + 2$ per la RAM. La copertura di 4 Byte alla volta, non si riesce a indirizzare il singolo Byte, ma solo a gruppi. Questo corso tiene conto per semplicità di 2^n Byte per la RAM e che la taglia massima degli indirizzi logici è 2^m bit.

7.3.1 Memoria associativa (TLB)

La memoria associativa è una memoria tampone che l'MMU usa per velocizzare la traduzione degli indirizzi virtuali. Il TLB possiede un numero fisso di elementi della tabella delle pagine, la quale viene usata per mappare gli indirizzi virtuali in indirizzi fisici. La memoria virtuale è lo spazio visto da un processo che può essere più grande della memoria fisica reale. Questo spazio è catalogato in pagine di dimensioni prefissate e

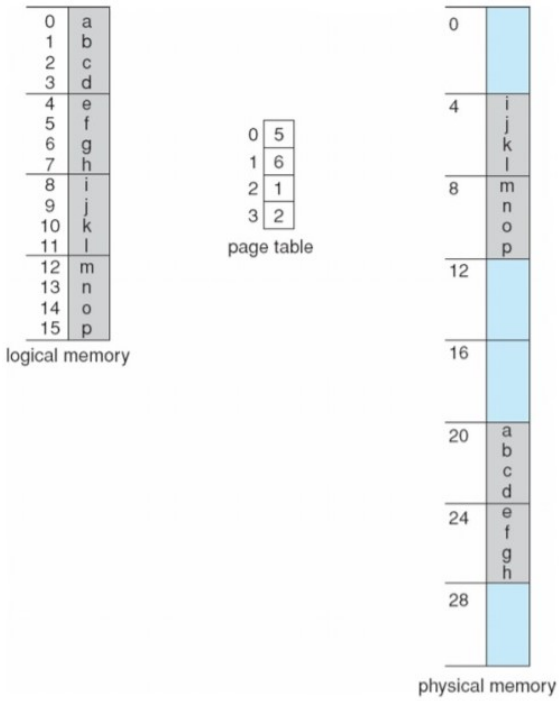

Figura 7.3: Paging example

generalmente solo alcune pagine vengono caricate nella memoria fisica in zone dipendenti dalla politica di rimpiazzo pagine.

La memoria associativa (TLB) serve a caricare dei pezzi della page table e la memoria complessivamente coperta è data da numero di entry, moltiplicata per la taglia di pagina (kb, mb, ...). Per esempio la dimensione della memoria associativa con 64 entry è nell'ordine del kilobyte. Per il principio dei processi, si capisce il come il caching eseguito con il TLB sia fondamentale per la velocizzazione dell'esecuzione di un processo. Un processo con milioni di pagine, sfrutta sempre un piccolo sottoinsieme di pagine (working set) che cambia lentamente nel tempo e quindi serve una cache per memorizzare efficientemente i dati più usati (TLB hit del 90/90%).

L'implementazione di una TLB ha anche dei costi di overhead (gestione), relativo alla modifica della tabella e sul content switch. Le entry precedenti al content switch non servono più e fanno riferimento a pagine di un processo precedente. Modificando la TLB con le pagine di un nuovo processo, sarebbe un appesantimento del content switch, quindi si lasciano un po' di pagine del processo precedente in modo da sfruttarle quando il processo tornerà in esecuzione. In alternativa il sistema operativo potrebbe creare una TLB per ogni processo.

7.3.2 Effective Accesse Time

Vediamo ora il costo dell'accesso, detto **EAT**.

$$EAT = (1 + \epsilon)\alpha + (2 + \epsilon)(1 - \alpha)$$

con α = hit ratio, e ϵ =unità di tempo

per esempio quindi con un hit ratio del 80% e $20ns$ per acceder al TLB e $100ns$ per accedere alla RAM, si ha un EAT di: $(120ns) * 0.80 + (220ns) * 0.20 = 140ns$ che corrisponde ad un 40% di *slowdown* (milgiore rispetto al 100% senza TLB). Portando l'hit ratio al 98% e mantenendo gli stessi dati, si riduce drasticamente il tempo di accesso ai dati.

Sicurezza dei processi: l'ultimo processo non riempie totalmente l'ultima pagina della page table, ma è praticamente vuota, in quanto le pagine devono essere tutte uguali in termini di dimensioni. Se il processo ha 6 pagine ma viene diviso in 8 entry, le ultime due pagine non vengono utilizzate e sono bit di qualunque genere, perché non hanno rilevanza. In tutti i casi 45 dove si recupera la memoria di un altro processo, non scrivendola fino in fondo, si possono presentare attacchi alla sicurezza dei processi (recupero di password...). Un modo per proteggere i dati dei processi è quello di scrivere fino in fondo la page table, scrivendo un bit di validità per indicare se la riga corrispondente è significativa o no. Un processo maligno, prima di leggere le informazioni di una page, legge se è una riga della pagina valida o invalida e il processo protegge l'accesso in caso di validità della riga, non facendolo leggere al processo maligno.

7.3.3 Inverted Page Table

La inverted page table viene utilizzata perché nelle macchine moderne la dimensione della page table è molto grande. La page table può portare via uno spazio significativo della memoria centrale. Tale spazio diventa ancora più significativo se viene raffinata la tecnica della paginazione usando la memoria virtuale. Per ovviare a questa grande perdita di spazio, è stata sperimentata un tecnica alternativa chiamata **inverted page table**.

L'idea è quella di invertire gli schemi visti fin'ora. L'inverted page table funziona nel seguente modo: invece di avere una page table per ogni processo, in questo caso si ha un'unica page table per tutta la memoria centrale. Questa nuova page table da informazioni relative ai frame in cui è divisa la RAM e non alle pagine dei processi. Un frame della memoria centrale viene utilizzato per contenere una certa pagina di un certo processo.

Tuttavia questo schema presenta un problema ovvio sull'efficienza della ricerca. Bisogna scorrere tutta la page table per effettuare una ricerca. Per risolvere questo problema, si può utilizzare una **memoria associativa**. Il sottoinsieme dell'inverted page table, che in un dato momento fa parte del working set del processo in esecuzione, dev'essere

caricato sulla memoria associativa. Nonostante ciò, utilizzando solo la tecnica della memoria associativa, non è sufficiente per risolvere il problema. Si può utilizzare la tecnica dell'**hashing**. Tale tecnica utilizza una hash table.

L'hash table è l'implementazione di una hash function, cioè una funzione che, nel caso della page table, prende come argomento un'informazione che si vuole ricercare. Questa funzione restituisce una entry che riesce a fornire informazioni precise e immediate sulla ricerca.

7.4 Virtual-Memory Management

Un concetto importante da sottolineare fin da subito è che la *paginazione* è lo schema visto fin'ora. La *memoria virtuale* è una raffinazione dello schema di paginazione, ma non è parte della paginazione stessa.

L'idea della memoria virtuale è quello di poter eseguire un processo avendo caricato sulla memoria centrale soltanto un sottoinsieme delle sue pagine. Fin'ora si era detto che per eseguire un processo, esso doveva essere fisicamente caricato tutto su memoria centrale. Ora, con la memoria virtuale, si accetta che sia caricato su RAM solamente un sottoinsieme del processo. Grazie a ciò, si ha un aumento del livello di multiprogrammazione e un processo può essere eseguito senza dover attendere che l'intero processo sia caricato su RAM. È sufficiente caricare un numero limitato di pagine.

Quando viene prodotto l'indirizzo logico, a seguito dell'esecuzione di un processo, potrebbe darsi che la parte interessata di un processo non sia caricata effettivamente su RAM, quindi bisogna andarla a cercare sulla memoria secondaria. Questo meccanismo viene chiama **page fault**. Un page fault viene prodotto da un processo che cerca di accedere a una pagina che in quel dato momento non è presente su memoria centrale.

La Figura 7.4 mostra i passi che devono essere prodotti dal sistema operativo quando capita un page fault, nell'ordine dei numeri visti in figura. Il page fault viene prodotto da una referenza che fa riferimento a una pagine che, in quel dato momento, non è presente sulla memoria centrale. Quindi quell'indirizzo logico fa riferimento a una pagine che non c'è, non esiste nella RAM. Ciò porta a una interrupt e il sistema operativo dovrà gestire questa la situazione.

Il sistema operativo deve capire qual è la pagina mancante e la deve recuperare dalla memoria secondaria. Una volta recuperata, tale pagine, viene spostata sulla RAM. Se nella memoria centrale ci sono dei frame liberi, la nuova pagina viene inserita. Se, invece, non ci sono frame liberi, bisogna trovare una **pagina vittima**. Una pagina vittima è una pagina che si trova su memoria centrale e che si decide di eliminare per fare spazio alla nuova pagina. Terminata la fase 4, bisogna aggiornare la page table del processo che era in esecuzione. Bisogna aggiornare la entry della nuova pagina. L'ultima fase, la fase 6, è la *restart instruction*.

Per quanto riguarda il page fault esistono due algoritmi molto importanti:

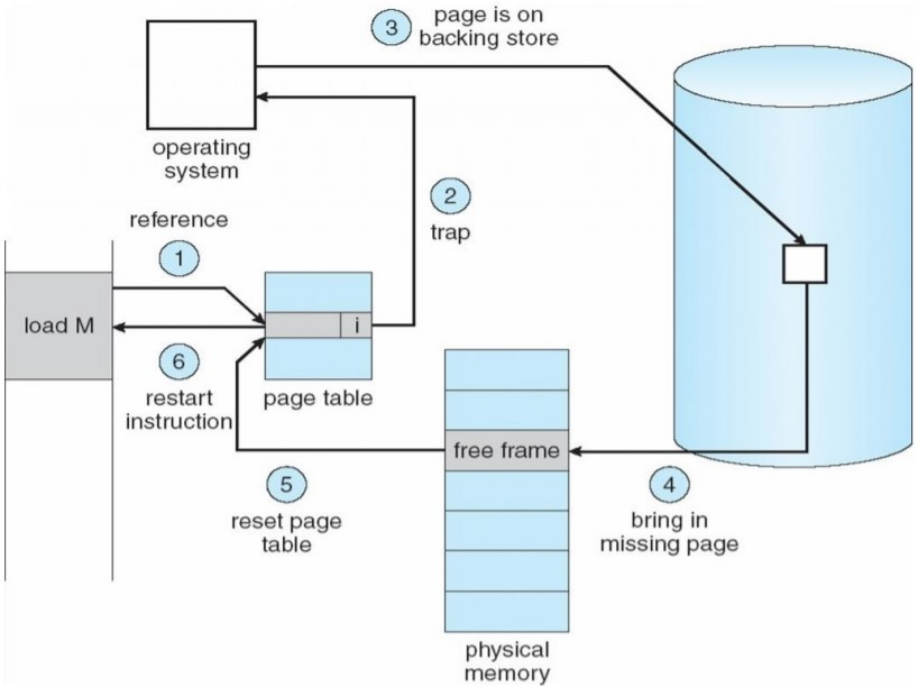

Figura 7.4: Step page fault

1. **Page replecement**: questo algoritmo ha a che fare con la scelta della pagina vittima (avviene nel possibile swap out). Usa il **dirty bit/modify dirtper** ridurre l'overhead del trasferimento delle pagine. Con questo algoritmo completiamo la separazione tra memoria logica e fisica. L'algoritmo di page replecement ha il compito di fare in modo che la percentuale di page fault sia la più bassa possibile.

2. **Frame allocation;** questo algoritmo determina quanti frame associare ai singoli processi.

7.4.1 First-In-First-Out (FIFO) Algorithm

Un primo algoritmo di page replacement è quello First-In-First-Out. L'algoritmo FIFO tiene in considerazione le pagine possibili che ha a disposizione il processo e considera l'ordina con le quali tali pagine sono state caricate su memoria centrale. Quando c'è da scegliere una pagina vittima, si sceglie la pagina che è stata caricata in RAM per prima.

7.4.2 Optimal Page Replacement Algorithm

Una politica ottimale produce la percentuale più bassa di page fault. Questo è l'obiettivio principale del page replacement. Tale politica sceglie come vittima la pagina

sarà utilizza più in là nel tempo. Guarda al futuro.

Ovviamente, questa politica, ha un problema di fondo. Implementarla significherebbe poter essere in grado di guardare al futuro e ciò non è possibile. È un algoritmo ottimale ma, purtroppo, non è possibile implementarlo. Sono necessarie delle approssimazioni.

7.4.2.1 Least Recently Used (LRU) Algorithm

Invece di guardare, in caso di page fault, il futuro, si può pensare di guardare al passato. Si effettua lo stesso procedimento svolto dall'algoritmo ottimale, ma invece di guardare al futuro si guarda al passato. Una politica che guarda al passato invece che al futuro si chiama Least Recently Used (LRU).

La scelta della pagina vittima, secondo questo algoritmo, cade sulla pagina usata meno recentemente. L'implementazione di questo algoritmo è possibile poiché si guarda al passato però bisogna sempre valutare che l'overhead derivante dall'applicazione di questa politica sia vantaggioso.

Una possibile implementazione di questa politica viene chiamata **counter implementation**. Tutte le volte che capita un page fault, bisogna sapere quando è stata, per le varie pagine presenti in memoria centrale, l'ultima volta che è stata utilizzata. Bisogna cercare l'ultimo momento temporale di utilizzo di quella pagina. Per fare ciò si associa ad ogni pagina un counter, il quale ha il compito di dire quando quella pagina è stata utilizzata l'ultima volta. A questo punto, grazie ai counter, si riesce a determinare la pagina vittima. Lo svantaggio di questo algoritmo è che ogni volta che si accede alla RAM, bisogna aggiornare la variabile counter. Tali counter rischiano di diventar pesanti, soprattutto dal punto di vista software.

Un'altra possibile implementazione di queta politica viene chiamata **stack implementation**. In questo caso si utilizza una lista che rappresenta l'ordine temporale di utilizzo delle pagine. Il grosso vantaggio di questo schema è se si utilizza più volte una stessa pagina, non è necessario aggiornarle il counter. Essa sarà sempre al top dello stack. Utilizzando uno stack, si può accedere al primo e all'ultimo elemento della lista stessa. Esistono anche dei puntatori grazie ai quali ci si può muovere in entrambi sensi all'interno della lista stessa.

Concludendo, anche utilizzare l'approssimazione Least Recently Used si hanno dei costi implementativi troppo elevati. A questo punto vediamo l'approssimazione dell'approssimazione, cioè un'approssimazione della Least Recently Used. Questo ci porta a vedere le politiche che si trovano effettivamente sui computer moderni.

7.4.2.2 Reference Bit e Second Chance

La prima politica si chiama **Reference Bit**. L'idea è quella di associare ad ogni pagina un reference bit, cioè un bit, che indica se la pagina è stata utilizzata nell'ultimo intervallo di tempo. In questa politica, il tempo viene suddiviso in intervalli. Il reference

bit viene settato se la pagina in questione è stata usata durante l'intervallo di tempo attuale. L'idea del reference bit è quella di selezionare una pagina che abbia il reference bit a 0. La pagina potrebbe diventare una pagina vittima.

La seconda politica si chiama **Second Chance**. Questa politica mette assieme l'idea del reference bit e l'idea della politica FIFO. La precisone di questa politica dipende da quanti bit si hanno, cioè dipende da quanti periodi di tempo si pensa di coprire con le matrici di bit. Con questa politica si ha una lista di pagine, come nella First-In-First-Out, raffinata attraverso l'aggiunta di un solo bit, cioè il reference bit.

L'idea di questa politica è di sceglie la pagina vittima guardando il reference bit della pagina stessa. Se questo è settato a 0, quella pagina diventerà la pagina vittima, altrimenti se il bit è settato a 1 gli viene data una "seconda possibilità" e il bit viene modificato a 0. Quindi la prima pagina nella lista con il reference bit a 0 diventa la pagina vittima.

7.4.3 Frame allocation Algorithm

Vediamo ora alcuni concetti e algoritmi per l'allocazione dei frame.

La questione, in questo caso, è chiedersi quanti frame bisogna assegnare a ciascun processo. Non è possibile effettuare una scelta arbitraria del numero di pagine da associare a un certo processo, è comunque necessario un numero minimo. Uno primo modo di allocazione è **l'equal allocation.** Quando si utilizza un'allocazione fissa, si da a tutti i processi lo stesso numero di frame. Se si hanno 100 frame e 5 processi, ogni processo prenderebbe 20 frame. Questo metodo risulta eccessivo perché non tutti i processi sono uguali, alcuni hanno esigenze maggiore e necessitano di un numero di frame maggiore.

Un secondo metodo è quello della **proportional allocation**. In questo caso i frame vengono allocati in maniera proporzionale alla taglia del processo stesso. Se la taglia del processo p_i è s_i, la somma di tutte le taglie dei processi è $S = \sum s_i$ ed m è il numero totale di frame, allora l'allocazione a_i per il processo p_i dev'essre proporzionale alla taglia di p_i, quindi si ha che $p_i = \frac{s_i}{S} * m$.

Ovviamente, quando si parla di processi, bisogna parlare sempre di priorità. Un processo che ha priorità alta è un processo a cui si vuole dare precedenza, per quanto possibile. Un altro modo per prendere in considerazione la priorità è quello di assegnare una allocazione di frame nei processi che prende in osservazione la priorità stessa. Se un processo ha più frame a disposizione, avrà anche un numero minore di page fault. Quindi diciamo che l'**allocazione per priorità** consiste nell'assegnare una quantità di frame ai vari processi che è proporzionale alla priorità dei processi stessi. Si effettuano gli stessi calcoli visti in precedenza per la taglia, ma in questo caso al posto che la taglia viene presa in considerazione la priorità. Un altro punto importante per quanto riguarda la allocazione di priorità è che se un processo genera un page fault, egli può scegliere come pagina vittima un processo che ha una priorità più bassa. Quindi un processo con

priorità più alta può rubare dei frame da un processo con priorità più bassa. Questo comportamento porta a dei rischi per i processi con priorità bassa.

7.4.4 Thrashing

Un rischio della memoria virtuale si chiama **thrashing**. Si arriva al thrashing quando il sistema operativo ha esagerato con il livello di multiprogrammazione. Quando il livello di multiprogrammazione diventa troppo elevato rispetto a quello che sono le esigenze dei singoli processi, allora c'è il rischio che il numero di page fault sia elevato.

Quando ci sono troppi page fault, il processore potrebbe essere poco utilizzato. Accade ciò perché il momento in cui capita un page fault è un momento in cui si prende tempo per operazioni di trasferimento di byte fra le memorie secondarie e la memoria centrale. Se, in generale, su tutti i processi si ha un numero di page fault alto, si rischia che il processore si trova ad essere poco utilizzato. Il sistema operativo, essendo il processore la risorsa più importante del calcolatore, fa moltissima attenzione a queste problematiche.

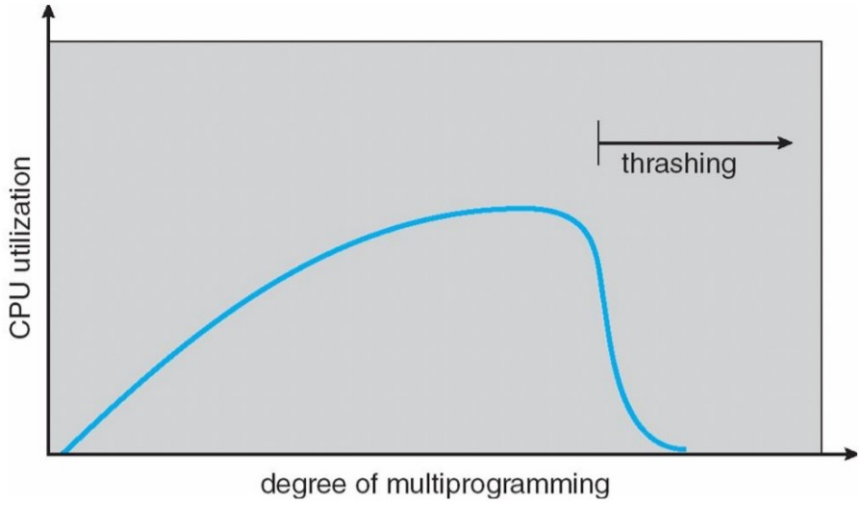

Figura 7.5: Thrashing

Per ovviare al problema del thrashing, è necessario diminuire il livello di multiprogrammazione.

Capitolo 8

Security and Cryptography

Quando si parla di sicurezza i problemi che vengono affrontati sono quelli che insorgono quando ci sono comunicazioni a distanza, per esempio messaggi tra persone, acquisti online, operazioni bancarie, etc. Il primo concetto importante quando si parla di sicurezza è la **confidenzialità.** In uno scambio di messaggi tra due entità, soltanto chi invia e chi riceve è in grado di comprendere il contenuto del messaggio stesso. Un'altra esigenza quest'oggi è quella dell'**autenticazione**. Quando si comunica con un altro soggetto, si vuole essere sicuri effettivamente dell'entità dell'altro interlocutore. **L'integrità dei messaggi** è un'altra importante proprietà. Si vuole essere sicuri che i messaggi che riceviamo siano effettivamente i messaggi che erano stati inviati dal sender. Non si vuole un soggetto centrale che prenda e modifica tali messaggi. L'ultima proprietà è **access and availability** e consiste nel fatto che i servizi restino accessibili e disponibili agli utenti. Per queste quattro proprietà sopra elencate l'uso della crittografia è **essenziale.**

In termini di sicurezza vengono utilizzate le seguenti notazioni: si indica con K_A la **encryption key** e con K_B la **decryption key**. Il messaggio in chiaro chiamato **plaintext**, tra due soggetti, viene identificato con una m.

Vediamo ora quali sono gli scenari più probabili: indiciamo con i nomi **Alice** e **Bob** gli utenti che vogliono scambiarsi messaggi e con **Trudy** l'attanccante malintenzionato.

- **Cipher-text only attack:** in questa situazione, la trudy vede passare la ciphertext, la acquisisce e l'analizza. Essa è l'unica informazione a propria disposizione. Dal punto di vista dell'attaccante questa è la situazione più povera, egli ha meno informazioni su cui poter giocare.

- **Known-plaintext attack:** in questa situazione l'attaccante conosce alcune coppie di associazione tra plaintext e ciphertext.

- **Chosen-plaintext attack:** questa situazione è più sofisticata rispetto alle precedenti. In questo caso Trudy è capace di sperimentare come un certo plaintext possa essere tradotto. Trudy stesso ha accesso all'algoritmo per il criptaggio ed

è in grado di provare, per un certo plaintext n, quale sia il cybertext d che viene prodotto.

8.1 Symmetric Key Cryptography

La crittografia classica è la crittografia a chiave simmetrica. In questo tipo di crittografia, la chiave usata dal sender **Alice** e dal recevier **Bob** è identica. Il cifrario di Cesare era un'applicazione concreta di questo tipo di crittografia. Tuttavia, non era molto sicura perché le chiavi possibili erano in numero molto basso per poter sopravvivere ad un attacco di tipo bruteforce. Un'implementazione migliorativa è utilizzare un **substitution cipher** in modo da trasformare le lettere dell'alfabeto non usando come chiava un numero soltanto ma usando una sostituzione arbitraria. La chiave, per il cifrario di Cesare, diventa una sequenza di lettere la cui lunghezza è pari alla lunghezza dell'alfabeto stesso. L'alfabeto può anche includere i numeri e i segni di punteggiature. Con un sistema di questo tipo, un bruteforce attack non funziona. Le combinazioni possibili, se utilizzato l'alfabeto inglese, sarebbero 26!. Tuttavia, in questo caso il problema non è il bruteforce attack, ma è l'**analisi statistica.** Questa analisi funziona correttamente quando i messaggi inviati sono particolarmente lunghi, in modo da avere una casistica ampia su cui fare riferimento. L'idea è quella di provare a indovinare alcune combinazioni in modo da ridurre il numero di combinazioni possibili che restano sulle altre lettere. Viene usata assieme a un attacco di tipo bruteforce con l'idea di ridurre lo spazio di possibilità dell'insieme degli elementi che devono essere analizzati.

Che cosa viene fatto oggi giorno in ambito della crittografia a chiave simmetria? Iniziamo introducendo il concetto di **cifratura a blocchi**. Con questo tipologia di cifratura si ha un plaintext che viene suddiviso in blocchi della stessa dimensione. L'idea è quella di codificare separatamente i vari blocchi. Se prendiamo come dimensione $n = 3$ e un plaintext del tipo 010110001111. Avendo $n = 3$ bisogna suddivide il plaintext in gruppi da 3 bit ciascuno: 010 110 001 111. A questo punto bisogna individuare una tabella di codifica per questi bit.

input	output	input	output
000	110	100	011
001	111	101	010
010	101	110	000
011	100	111	001

Table 8.1 ♦ A specific 3-bit block cipher

Facendo riferimento alla tabella sottostante, avendo scelto una lunghezza di blocchi pari a 3, bisogna considerare tutte le traduzioni di tutte le possibili sequenze di 3 bit. I possibili input sono $2^3 = 8$ e gli output mostrano come sarà tradotto il blocco in questione. Questa corrispondenza input/output rappresenta una funzione bigettiva, cioè una funzione che prende un input e restituisce un output, quindi sia input che output sono sequenze di 3 bit e la funzione è bigettiva perciò ogni input dev'essere mappato su un output diverso. Quindi, dato il plaintext precedente, il ciphertext sarà 101 000 111 001.

In generale possiamo dire che si hanno delle tabelle di 2^n entry, cioè si hanno 2^n possibili caratteri. Le possibili tabelle, quindi le possibili permutazioni, sono $(2^n)!$. Gli input possibili sono 2^n e sapendo che le permutazioni possibili su un insieme di k elementi sono $k!$, si ha $(2^n)!$ tabelle.

Una situazione molto frequente è avere $n = 64$ o $n = 128$. In questa situazione si hanno tabelle di 2^{64} entry (o addirittura 2^{128}) e, essendo numeri altissimi, creano problemi. La tabella rappresenta la chiave sia di codifica che di decodifica e se sono composte da un numeri di entry elevatissimo sorgono dei problemi. Il sender e recevier devono possedere una rappresentazione della tabella.

L'algoritmo più nodo per la crittografia a chiave simmetrica è il DES (Data Encription Standard), la cui versione più aggiornata si chiama AES (Advanced Encription Standard). Con questi algoritmi si parte con un chiave composta da un numero di 64/56 bit nel DES e per l'AES con almeno 128 bit. Il secondo passo è quello di prendere il blocco composto dai bit e lo si suddivide in blocchi più piccoli. Oltre a ciò, è necessario avere una chiave che svolge differenti compiti:

- la chiave dice come tradurre 3 bit;
- la chiave dice come effettuare degli "scramble", ovvero delle permutazioni di bit.

Dato un plaintext, si utilizza la tabella per ottenere un primo passaggio. Facendo riferimento al plaintext precedente, si avrebbe come output dopo la prima applicazione della tabella la seguente sequenza: 101 000 111 001. A questo punto è possibile effettuare degli **scramble**. Un esempio di scramble potrebbe essere il seguente: dato un numero che si trova in posizione i dispari, egli viene spostato e posizionato nella posizione $i + 4$. L'output dopo il primo scramble è 100 010 110 011. A questo punto è possibile applicare nuovamente la tabella sull'output dopo il primo scramble e, come nuovo risultato, si ha la sequenza 011 101 000 100. Questo è l'output dopo la seconda applicazione della tabella.

L'aspetto interessante è che un pezzo dell'output, per esempio i primi 3 bit, dopo la prima applicazione della tabella dipendevano solamente dai primi 3 bit del plaintext originale. Ora dipendono anche dall'applicazione del primo scramble. Si può andare avanti e fare altri scramble in modo da rendere più sofisticata la sequenza di bit. L'obiettivo è che nell'output finale, un bit qualunque dipenda da tutti i bit del plaintext iniziale.

Con una situazione di questo tipo, l'unico attacco consiste nel provare tutte le chiavi possibili. Usando chiavi da 64 bit, si hanno 2^{64} chiavi possibili.

Un problema derivante da questa applicazione è che traducendo blocchi di bit, c'è la possibilità che se 2 blocchi sono identici allora anche l'output prodotto da essi sarà identico. L'idea è quella di introdurre della randomness (casualità). Lo XOR è un'operazione fra due bit che da come risultato 0 se i due bit sono identici e 1 se i due bit sono diversi. Una proprietà importante dello XOR è che se si parte da un bit b, si applica lo XOR con un bit b^1 e poi si riapplica nuovamente, si ottiene il bit di partenza b. $(b\#b^1)\#b^1 = b$.

Il problema di questo schema è che si raddoppiano le quantità di informazioni da inviare. Nello schema di base, senza randomness si invia la sequenza $c(1)$, $c(2)$ e $c(3)$, quindi 3 blocchi da 3 bit ciascuno, mentre ora questi 3 blocchi bisogna alternarli con i 3 blocchi casuali, $c(1)$, $r(1)$, $c(2)$. $r(2)$, $c(3)$ $r(3)$. In generale se si ha un linguaggio di lunghezza m, questa tecnica porta ad inviarne il doppio. Questo potrebbe essere uno svantaggio, dipende sempre dalla taglia.

Tuttavia applicando la tecnica **Cipher Block Chaining (CBC)** si riesce ad semplificare questo problema. Nello Cipher Block Chaining si ha un solo blocco casuale che viene introdotto e tutti gli altri blocchi casuali vengono evitati perché l'idea è quella di sfruttare il blocco random e il blocco di ciphertext precedente. (c(i) = K_s(m(i)#c(i-1))). Utilizzando questa tecnica sia aumenta solo di 1 il numero di blocchi da inviare.

Concludendo le difficoltà e debolezze principali della crittografia a chiave simmetrica sono:

1. La **chiave** deve rimanere **segreta** ma dev'essere nota ad entrambi i partecipanti (sender, recevier);

2. **Distribuzione della chiave**, come assicurarsi che la chiave possa essere recapitata a entrambi i partecipanti. O se si assume che sia uno dei due a produrla, bisogna capire come recapitarla all'altro;

3. Il numero di chiavi necessarie aumenta in base al quadrato del numero delle persone che vogliono scambiarsi messaggi; Se ci sono n persone e se c'è la possibilità di effettuare scambi segreti tra coppie di persone, allora servono circa n^2 chiavi.

8.2 Public Key Cryptography

La crittografia a chiave pubblica si basa su funzioni matematiche piuttosto che su operazioni di sostituzione e permutazione come la crittografia a chiave simmetrica. Ancora più importante è il fatto che la crittografia a chiave pubblica è asimmetrica. Prevede l'uso di due chiavi distinte.

Gli elementi che vengono utilizzati quando si parla di crittografia a chiave pubblica sono:

- una coppia di chiavi(pubblica K^+, privata K^-);
- il plaintext m;
- il ciphertext c;

I requisiti per gli algoritmi a chiave pubblica sono:

- Dev'essere computazionalmente facile produrre delle coppie (K^+, K^-);
- Dev'essere computazionalmente facile criptare, cioè calcolare il risultato dell'applicazione della chiave pubblica su un certo plaintext. $K^+(m)$.
- Dev'essere computazionalmente facile anche decriptare, cioè calcolare $K^-(c)$;
- Dev'essere computazionalmente impossibile per un attaccante risalire alla chiave privata K^- conoscendo la chiave pubblica K^+;
- Dev'essere computazionalmente impossibile per un attaccante risalire al plaintext m conoscendo la chiave pubblica K^+ e il ciphertext c.
- Le chiavi pubbliche e private devono potersi applicare in entrambe le direzioni, cioè $K^-(K^+(m)) = K^+(K^-(m))$.

Quando si dice **computazionalmente facile** si intende il calcolo di qualcosa attraverso un algoritmo polinomiale, cioè con un tempo che cresce come un polinomio al crescere della taglia dell'input, del tipo n^k dove n è la taglia dell'input e k è una costante. Mentre **computazionalmente impossibile** è un problema per il quale non si conoscono algoritmi polinomiali. Si conoscono solo algoritmi esponenziali con un tempo di calcolo k^n.

8.2.1 RSA Algorithm

Nel caso dell'algoritmo RSA si sfrutta da un lato il problema della decomposizione di un numero nei suoi fattori primi e dall'altro l'aritmetica modulo. L'aritmetica modulo viene indicata nel seguente modo: x mod n. Rappresenta il resto di x quando viene diviso per n. L'aritmetica modulo ha delle proprietà algebriche molto ricche:

- **Proprietà di distributività sulla somma:** [(a mod n) + (b mod n)] mod n = (a+b) mod n;
- **Proprietà di distributività sulla sottrazione:** [(a mod n) - (b mod n)] mod n = (a-b) mod n;
- **Proprietà di distributività sulla moltiplicazione:** [(a mod n) * (b mod n)] mod n = (a* b) mod n;

- **Proprietà di distributività sull'esponente:** (a mod n)^d mod n = a^d mod n;

In generale diciamo che x mod $n = k \rightarrow$ esiste un h tale che $x = (n \cdot h) + k$.

Vediamo ora due teoremi importanti per l'algoritmo RSA. Il primo teorema è il **teorema di Fermat**. Dato un numero primo p e a con $a < p$, allora possiamo dire che $a^{(p-1)}$ mod $p = 1$. Una conseguenza di questo teorema è che se si prende

$$\begin{aligned} &(a^{(p-1)} \times a) \text{ mod p} = \\ &(a^{(p-1)} \text{ mod p} \times \text{ a mod p}) \text{ mod p} = \\ &1 \times a = a \end{aligned} \tag{8.1}$$

Il secondo teorema è il teorema principale dell'algoritmo ed è il **teorema di Eulero**. Questo teorema parla di una funzione chiamata **funzione toziente**. La funzione toziente di n, $F(n)$, dice la cardinalità dell'insieme degli interi più piccoli di n tali che m non ha fattori comuni con n. F(n) = # {m < n t.c. m non ha fattori comuni con n}. Per fattori comuni con n s'intende che nella decomposizione di m e n in fattori primi, esiste un fattore a comune. Il massimo comune divisore fra n e m non è 1. Esiste un numero maggiore di 1 che divide sia n che m.

Proposizone: se n è un numero prima, allora la funzione toziente di n è uguale a $n - 1$. ($F(n) = n - 1$).

Teorema: se p,q sono numeri primi, $F(p \cdot q) = (p - 1) \cdot (q - 1)$.

La proposizione e il teorema precedente ci servono per definire e comprendere a pieno il teorema di Eulero. Vediamo ora l'enunciato: dati due numeri a e n primi tra loro, tranne 1, allora

$$a^{F(n)} \text{ mod } n = 1.$$

Per esempio se si ha $n = 10$, quindi $F(10) = 4$, e $a = 3$, allora

$$3^4 \text{ mod } 10 = 1$$

infatti,

$$3^4 = 3^2 \cdot 3^2 = 9 \cdot 9 = 81 \text{ mod } 10 = 1$$

Conseguenza: se si eleva $a^{(F(n)+1)}$ mod n si ha:

$$\begin{aligned} &a^{(F(n)+1)} \text{ mod n} \\ &= a^{(F(n)+1)} \text{ mod n} = (a^{F(n)} \text{ mod n} \cdot \text{ a mod n}) \text{ mod n} \\ &= 1 \cdot \text{ a mod n} \\ &= a \text{ (assumendo che a < n)} \\ &= a^{(F(n)+1)} \text{ mod n} = \text{a} \end{aligned} \tag{8.2}$$

Moltiplicando $(F(n)+1)$ per un numero k qualsiasi, possiamo vedere che il risultato non cambia:

$$\begin{aligned} & a^{(k\cdot F(n)+1)} \text{ mod n} = a \\ & a^{(k\cdot F(n)+1)} \text{ mod n} = \\ & (a^{k\cdot F(n)} \cdot a) \text{ mod n} \\ & (a^{F(n)^k} \cdot a) \text{ mod n} \\ & ((a^{F(n)} \text{ mod n})^k \cdot \text{ a mod n}) \text{ mod n} \\ & = 1 \cdot \text{ a mod n} = a \end{aligned} \tag{8.3}$$

Vediamo ora l'ultima proprietà necessaria per descrivere l'algoritmo RSA.

Se a non ha fattori comuni con n, allora esiste un inverso moltiplicativo di a. Cioè esiste un numero d tale che $(a \cdot d) \text{ mod } n = 1$. Un inverso moltiplicativo di un numero rispetto a n è un altro numero chiamato d tale che se si moltiplica $(a \cdot d) \text{ mod } n = 1$. Questo significa che esiste un k tale che $(a \cdot d) = n \cdot k + 1$.

Ora che sono state illustrate tutti i teoremi e le proprietà, possiamo vedere il funzionamento dell'algoritmo RSA. Vediamo i passi:

- Inizialmente si parte con due numeri primi p e q;
- Si moltiplicano i due numeri primi ottenendo $n = p \cdot q$;
- Si scegli un numero e tale che e non abbia fattori comuni con la $F(n) = (p-1) \cdot (q-1)$. Ciò significa che non esiste un numero diverso da 1 che sia divisibile per e e per $F(n)$.

  Se questo è vero, si può applicare l'ultima proprietà sopra elencata. Quindi possiamo dire che esiste un inverso moltiplicativo d tale che $(e \cdot d) \text{ mod } F(n) = 1$.
- A questo punto si è in grado di creare la chiave pubblica, la quale è costituita dalla coppia $[n, e]$. La chiave privata invece è costituita dalla coppia $[n, d]$, dove l'informazione segreta è d.

Dato un plaintext m dove $(m < n)$, la codifica consiste nel creare un ciphertext che si ottiene elevando il plaintext alla e e successivamente effettuare il tutto per il modulo n, quindi $c = m^e \text{ mod } n$. La decodifica, invece, consiste nel prendere il ciphertext, elevarlo alla d ed effettuare il tutto per il modulo n. ($c^d \text{ mod } n$).

Questo procedimento funziona perché

$$
\begin{aligned}
c^d \text{ mod n} &= (m^e \text{ mod n})^d \text{ mod n - [espandendo la definizione di c]} \\
&= (m^e)^d \text{ mod n - [sfrutto la distributività del mod]} \\
&= m^{(e \cdot d)} \text{mod n - [sfrutto semplice proprietà dell'elevamento a potenza]} \\
&= m^{(F(n) \cdot k+1)} \text{ mod n - [espandendo la def. di d come inverso moltiplicativo di e]} \\
&= m \text{ - [proprietà di Eulero]}
\end{aligned}
\tag{8.4}
$$

Per determinare il procedimento si utilizza la proprietà derivata dal teorema di Eulero e la proprietà dei fattori comuni.

8.3 Digital signatures

Un problema della crittografia a chiave pubblica è che quando Alice invia un messaggio a Bob utilizzando la chiave pubblica di Bob. Tuttavia chiunque potrebbe aver potuto inviare il messaggio pretendendo di essere Alice. Non c'è nulla che possa garantire a Bob, una volta decifrato il messaggio, che esso sia stato effettivamente inviato da Alice. Per superare questo ostacolo, si può utilizzare la tecnica delle **firme digitali**.

Supponiamo che Bob deve inviare un messaggio ad Alice. Bob firma il plaintext m criptandolo con la sua chiave privata, K_B^-. Dall'altra parte Alice riceve una coppia costituita dal messaggio m stesso e dal messaggio criptato con la chiave privata. A questo punto Alice, conoscendo la chiave pubblica di Bob K_B^+, la applica al ciphertext e verifica il messaggio inviato. Con questo procedimento Alice è sicura che il messaggio decriptato è stato inviato da Bob.

Il problema di questo schema è che la crittografia a chiave pubblica è una crittografia piuttosto costosa. Se i messaggi che vengono scambiati sono lunghi, questo schema richiederebbe il criptaggio di messaggi lunghi e sarebbe inefficiente. Vediamo ora una tecnica per ridurre la tagli del messaggio che dev'essere tratto.

L'idea è quella di utilizzare delle funzioni hash di tipo crittografico. Queste funzioni traducono un messaggio n, a volte anche molto lungo, in un messaggio m molto più corto, quindi necessariamente, per una questione di cardinalità, sono funzioni che non possono essere iniettive quindi c'è la probabilità che due messaggi inizialmente diversi, possono essere mappati tramite la funzione hash sullo stesso output. La particolarità delle funzioni hash usate nell'ambito crittografico è che se si vedono transitare uno degli output x, che è stato ottenuto applicando la funzione hash a un messaggio m, ovvero $(H(m))$, dev'essere impossibile per un attaccante trovare un altro messaggio m^1 che produca lo stesso output x.

Capitolo 9

File System

Un file system è un metodo del sistema operativo usato per immagazzinare, catalogare e per recuperare file (directory). È opportuno distinguere la struttura/organizzazione in logica e fisica.

Nella **struttura logica** il sistema operativo deve definire dal punto di vista logico delle operazioni per manipolare il file system usando come unità di trasferimento le informazioni tra un file ed un processo viene chiamato **record logico**. Nel caso standard viene preso come record logico il Byte. I metodi di accesso sono di due tipi:

- **Sequenziale:** il record logico successivo che dobbiamo utilizzare è il record logico successivo o precedente, dal momento che ci si può spostare sequenzialmente nel file.
- **Casuale/diretto:** ci si può muovere liberamente all'interno del recordo logico.

Nella **struttura fisica**, i record logici sono allocati nella memoria di allocazione secondaria (disco rigido). Lo spazio disponibile su memoria secondaria viene divisa in blocchi fisici, per esempio qualche MB. Dato un certo file costituiti da un insieme di record logici, bisogna capire dove sono posizionati e come si riesce a tenere traccia degli stessi: Si hanno tre metodi di allocazione:

- **contiguo;**
- **concatenato;**
- **indicizzato.**

9.1 Allocazione

I blocchi fisici sono un'unità di allocazione la cui taglia dipende dal sistema operativo che si utilizza facendo riferimento alla velocità e allo spazio che si ha a disposizione. Si

indica con **TBF** la taglia del blocco fisico e con **TRL** la taglia del record logico. Detto ciò possiamo dire allora che il **numero** di record logici dentro un blocco fisico è data dalla divisione tra **TBF/TRL**. Quando si effettua l'operazione di allocazione si possono avere problemi di frammentazione, motlo simili a quelli della memoria centrale.

9.1.1 Allocazione contigua

Nell'allocazione contigua ogni file occupa blocchi contigui sul disco. In questo si potrebbero avere problemi di frammentazione esterna perché si ha della memoria inutilizzabile esterna rispetto a quella assegnata per i file. Anche per i file system si utilizza il decompattamento o la deframmentazione, all'interno della quale si spostano i file system già allocati per ricavara, da tanti piccoli blocchi liberi, un blocco molto più grande. Tuttavia, esiste anche un problema di frammentazione interna.

Nell'allocazione contigua, se il blocco iniziale del file è in posizione x, allora per accedere al record logico i bisogna accedere al blocco fisico $x + (i/n)$. Nell'allocazione contigua è facile implementare sia un accesso sequenziale sia un accesso secondo aspetti logici.

9.1.2 Allocazione concatenata

Evitare la frammentazione esterna, andando a frammentare i vari file utilizzando un puntatore per riconoscere il blocco successivo del file system. Si cerca di evitarla perché non si va più a considerare il **vincolo di contiguità**.

Tuttavia questa allocazione risulta poco efficiente, perché se si avesse bisogno dell'ultimo elemento del blocco, bisogna scorrerlo tutto. Un altro svantaggio consiste nel fatto che ci potrebbero essere problemi di affidabilità. Nel caso in cui perdessimo un puntatore, si perdono tutti i blocchi che seguono per sempre. Per ovviare a questo problema, si possono aggiungere dei puntatori nella direzione opposta per vedere da quali blocchi derivino, ma ciò posta ad un raddoppio della dimensione per i puntatori.

È possibile utilizzare anche la **File Allocation Table (FAT)**. La FAT è una tabella che contiene un elemento per ogni blocco e indica se il blocco è libero oppure no. Inoltre indica anche qual è il blocco successivo.

9.1.3 Allocazione ad indice

Per ogni fily system si usa un blocco separato per scrivere la sequenza in cui il file è stato memorizzato. All'interno si trovano i puntatori dei blocchi del file. Questa tecnica evita la frammentazione esterna. Entrambi i tipi di accesso sono implementati correttamente. Lo svantaggio consiste nel fatto che bisogna utilizzare un blocco indice, particolarmente evidente con file molto piccoli.