

1. Introduzione

Un Sistema Operativo è quel componente software che ha il compito di assicurare la corretta operatività di un sistema: **controlla e coordina le risorse hardware e software** utilizzate dai programmi in esecuzione, anche detti **processi**.

Esempio di Risorse Hardware:

- Processori (in termini di tempo di CPU che offrono ai vari processi)
- Memorie
- Dispositivi periferici (I/O)

Esempio di Risorse Software:

- Coda dei processi pronti
- Tabella di process table (UNIX)
- Tabelle dei file aperti per ogni singolo processo (nell'ambito di gestione file)

La gestione di questo tipo di risorse avviene da parte del **Kernel**, la parte più importante del sistema operativo.

1.1 Compiti di un Sistema Operativo

Il Sistema Operativo deve:

- Tenere traccia delle risorse (conoscendone anche lo stato, se sono libere o occupate. Nel caso della CPU è necessario sapere se questa è già stata assegnata a un processo o è libera)
- Adottare strategie di assegnazione delle risorse (necessario quando più processi richiedono la stessa risorsa)
 - **A CHI** assegnare la risorsa
 - **QUANDO** assegnarla
 - **PER QUANTO TEMPO**
- Allocare le risorse
- Recuperare le risorse

Quindi, le risorse devono essere assegnate ai processi secondo determinate politiche.

Il Sistema Operativo si deve occupare della **RISOLUZIONE DI POSSIBILI CONFLITTI**, deve garantire l'integrità del sistema e ottimizzarne le prestazioni globali.

1.2 Definizione

Un Sistema Operativo è un **PROGRAMMA DI CONTROLLO**: controlla l'esecuzione di altri programmi per prevenire errori ed usi impropri del calcolatore.

1.2.1 Funzionalità

1. Controllo del/dei processore/i
2. Controllo della memoria principale
3. Controllo dei dispositivi di I/O (anche dispositivi di memorizzazione secondaria)

1.2.2 Obiettivi

- Semplificare l'uso di un sistema di calcolo
- Rendere efficiente l'uso delle risorse (hw/sw) di un sistema di calcolo

1.2.3 Requisiti Fondamentali

Multiplexing

Garantisce che tutti i processi possano eseguire secondo la politica stabilità dal sistema Operativo.

Isolamento

Garantisce che se un processo ha un problema questo problema non infici gli altri processi.

Interazione

Permette ai processi di interagire tra loro. Quando ciò accade, l'*isolamento* non è garantito.

1.3 Storia dei Sistemi Operativi

L'evoluzione dei S.O. è stata influenzata dall'evoluzione dell'Hardware.

1.3.1 Evoluzione Sequenziale

Definizione

Le risorse del sistema vengono dedicate a un singolo programma (non ha importanza definirlo come processo quando è uno solo) finché questo non ha terminato l'esecuzione. Il programma si prende il possesso di tutto il sistema di calcolo.

Primi Sistemi

Scrittura dei programmi in linguaggio macchina (costituito da bit 0 e 1).

PROGRAMMATORE = OPERATORE

Sviluppo dei programmi e caricamento dei sistemi di calcolo completamente manuale.

L'operatore deve essere a contatto con il sistema di calcolo e andare a configurare manualmente il sistema in modo da scrivere il programma durante la configurazione.

PROGRAM COUNTER: indirizzo della prima istruzione dalla quale si sviluppava l'intero programma scritto in linguaggio macchina, che doveva essere scritto manualmente. Si doveva quindi avere accesso al *Program Counter* del Sistema di Calcolo. Si tratta di un **registro della CPU** la cui funzione è quella di conservare *l'indirizzo di memoria della prossima istruzione* (in linguaggio macchina) da eseguire.

- **PRODUTTIVITÀ MOLTO BASSA** sia per la macchina che per l'utente. Lo sviluppo di un programma risultava molto lento.

Evoluzione

- Linguaggi più evoluti (e loro **traduttori**)
- Perforatrici di schede e terminali (**editor**): non c'era più un caricamento manuale ma il programma era scritto diversamente, sempre in modo difficile ma aumentando la produttività.

- Utilizzo di routine standard per trattare le periferiche di I/O (**linker**): in questo modo il programmatore non doveva più scrivere il codice per interfacciarsi alle periferiche di Input / Output.

Esempio

Un programma scritto in C, una volta compilato, genera un file oggetto. Questo non è eseguibile perché all'interno del programma è presente una `printf`, che fa parte della libreria standard C e quindi deve intervenire il linker che prende il formato oggetto del programma e produce, collegandolo con l'oggetto della libreria, il file eseguibile da parte di un processo.

- LOADER: caricatore automatico dei programmi. Parte basilare di un S.O.
 - Controllo diretto della esecuzione da console
 - Correzione in linea (in diretta) degli errori: (debugger)

In caso **MULTIUTENTE** vengono applicate prenotazioni del tempo di uso del sistema.

SEQUENZIALE perché:

- Caricamento editor: permette all'operatore di scrivere il proprio programma. (L'eseguibile viene poi caricato nuovamente sul nastro magnetico)
- Caricamento traduttore: produce il programma nella forma oggetto.
- Caricamento linker
- Caricamento programma eseguibile

Il caricamento, in genere, avveniva da *nastro magnetico*, utilizzato sia come dispositivo di input che di output.

PROBLEMA: scarsa efficienza.

1.3.2 Esecuzione Batch

Si è pensato di aumentare il grado di *sfruttamento delle risorse* e la *produttività* dei programmatori attraverso la riduzione (o eliminazione) dei tempi morti causati dalle operazioni manuali.

Concetto di Batch

Insieme di diversi programmi (JOB), più precisamente comandi per fare eseguire programmi. Vengono raggruppati in base ad esigenze comuni.

Esempio

Diversi programmi FORTRAN vengono raggruppati insieme.

PROGRAMMATORE != OPERATORE

Il Programmatore scrive il programma Fortran e lo passa all'Operatore. Questo raggruppa i diversi programmi Fortran e costruisce un BATCH.

VANTAGGIO → si carica il compilatore FORTRAN una sola volta.

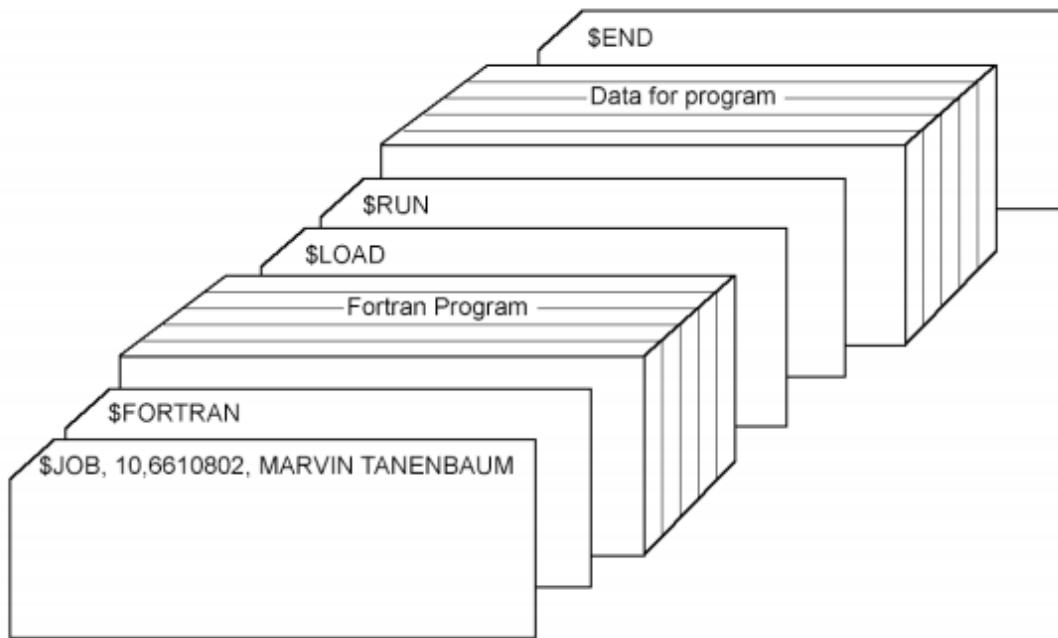
I Programmi venivano consegnati come schede di memoria perforate.

Configurazione schematica del Sistema di Calcolo:

Lettore di Schede → CPU → Stampante (fogli contigui)

Linguaggio Comandi

All'interno di un BATCH, i comandi sono dati in modo automatico tramite un linguaggio comandi **JOB CONTROL LANGUAGE (JCL)**. Azioni che era necessario che il sistema (ma anche l'operatore) facesse con il pacco di schede che veniva consegnato dal programmatore. Ogni linea di codice corrispondeva a una scheda. Il concetto del JCL è attualmente *presente in ogni sistema operativo*. E' possibile considerarlo come l'antenato dei linguaggi delle varie shell.



- **\$JOB** - Inizio di un programma (oltre a costituire il programma, costituiva le schede di controllo e i dati)
- **\$FTN** - Compilatore FORTRAN (quale compilatore caricare)
- **\$LOAD** - Caricamento programma (caricamento esplicito del compilatore)
- **\$RUN** - Esecuzione programma
- **\$END** - Fine di un programma

Questi comandi sono interpretati da una parte del S.O. (sempre residente in memoria)

→ **MONITOR**

SVANTAGGIO DELLA ESECUZIONE BATCH → l'utente non può controllare interattivamente l'esecuzione del programma. Se si riscontra un errore a runtime il sistema fa un DUMP della memoria → immagine della memoria quando il programma si ferma per un errore.

Problema

La velocità della CPU era più grande di 2/3 ordini di grandezza rispetto ai dispositivi I/O a causa della enorme differenza di velocità fra *dispositivi meccanici* (più lenti) come lettori di schede e stampanti rispetto alla velocità di *dispositivi elettronici CPU* (più veloci). La CPU rimane *in attesa* del completamento di operazioni di I/O per una notevole percentuale di tempo.

Soluzione

Sovrapposizione delle operazioni di I/O con le operazioni eseguite dalla CPU tramite una delle seguenti 3 opzioni:

- **OPERAZIONI FUORI LINEA**

Per il lettore di schede e la stampante (lenti nei confronti della CPU ma anche nei confronti del lettore e scrittore di nastro) si utilizza un'ulteriore CPU.

CPU LENTA → per interfacciarsi con un lettore di schede (lento). Viene scritto su nastro ciò che viene letto dal lettore di schede fuori linea (non connesso ancora alla CPU principale). In un secondo momento questa CPU può leggere da nastro i risultati della CPU veloce (principale) e riversare offline i risultati su una stampante.

L'utilizzo di una CPU lenta rappresenta l'interfaccia verso il programmatore, più lenta ma comunque *fuori linea*. Nei confronti del sistema di calcolo, con la CPU VELOCE l'interfacciamento è sempre e solo tramite nastro, un po' più veloce di altri dispositivi come stampanti. Il nastro infatti è un *dispositivo magnetico*, al contrario di una periferica che è un *dispositivo meccanico*.

Un programmatore deve poter sviluppare i propri programmi (il proprio JOB) senza sapere se vengono eseguiti con un S.O che fa uso o meno di operazioni fuori linea e quindi tali programmi devono poter leggere da schede o da nastro e devono poter scrivere su stampante o su nastro.

- INDIPENDENZA DAI DISPOSITIVI FISICI
- DISPOSITIVI LOGICI DI I/O

È il S.O che, sapendo se sta facendo uso o meno di operazioni fuori linea, **collega** il dispositivo logico di I/O con il giusto dispositivo fisico → **concetto base della ridirezione**

- UTILIZZO DI BUFFER DI INPUT - OUTPUT

meccanismo di INTERRUPT → la CPU viene sollecitata a gestire un dato ingresso inserendolo in un buffer, potendo continuare a fare altro fintanto che il dato non risulta disponibile sul dispositivo.

Questi buffer consentono di effettuare una sovrapposizione delle operazioni di input / output (operazioni di lettura del dato successivo) di un JOB con la sua esecuzione.

Quando il JOB ha bisogno del dato successivo lo trova già nel BUFFER (su memoria centrale)

Il concetto vale anche per l'uscita:

- programma produce uscita → BUFFER → stampante

- TECNICA DI SPOOLING

Acronimo per ***simultaneous peripheral operations on-line***. Invece che operare fuori linea, si lavora *on line* utilizzando i *dischi come buffer*. Tutto questo è reso possibile dall'avvento dei dischi magnetici a livello hardware. Consente quindi di caricare più programmi (o JOB) in memoria di massa.

Durante i tempi morti della CPU, venivano lette le schede (relative al JOB successivo) in parallelo all'esecuzione di un certo JOB, *memorizzando il risultato in un buffer su disco*. Quando il JOB successivo va in esecuzione trova i dati già nel buffer.

Lo Spooling è utilizzato nei sistemi operativi moderni per l'Output.

- Quando si chiede di stampare un documento, non si rimane in attesa del completamento della stampa, ma in realtà il documento viene battezzato in un'area di spooling e da questa viene preso dal gestore della stampante e va a riportare i dati che trova in quest'area sul dispositivo di output (stampante).

1.3.3 Multiprogrammazione

Quando un certo programma (JOB corrente) deve attendere un qualche evento, questo JOB viene sospeso. Ciò implica che la CPU risulti libera e in questo modo il sistema esegue un JOB diverso (siccome ci sono altri programmi in memoria di massa, per lo *spooling*). I Programmi rispettano lo schema di **Von Neumann**:

Un programma per essere eseguito deve essere sempre necessariamente caricato sulla memoria centrale.

JOB in memoria di massa = JOB in memoria centrale

Non è detto che sia caricato sempre il programma per intero, in quanto ci sono casi in cui con la memoria sono caricate solo parti di un programma, tuttavia *le istruzioni da eseguire nell'immediato devono essere caricate in memoria centrale*.

Funzionamento

La CPU, potendo avere a disposizione più JOB da eseguire, non ha più il vincolo dell'esecuzione sequenziale, ma può andare ad eseguire (quando il JOB corrente attende un evento) un altro JOB. Questo discorso è applicabile non solo a due programmi, ma a un numero qualunque di programmi. *Finché la CPU ha un JOB da eseguire non rimane mai inutilizzata* (cioè non è mai **IDLE**).

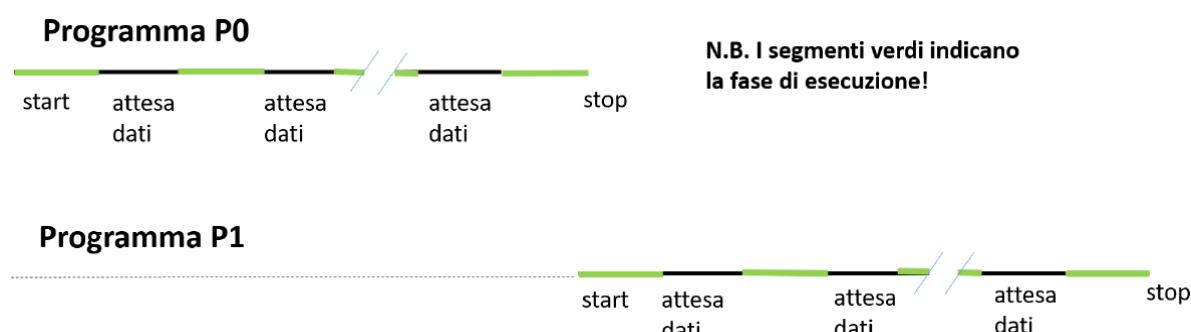
La presenza di diversi JOB nel sistema implica per il S.O:

- Gestione della memoria di massa
- Gestione della memoria centrale
- Gestione della CPU

Analisi Comportamento

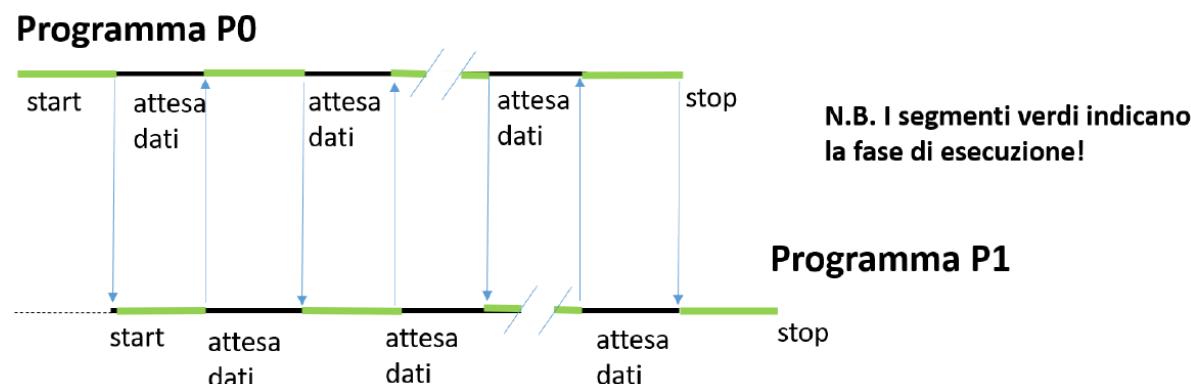
• SENZA MULTIPROGRAMMAZIONE

In caso di assenza di multiprogrammazione, il Sistema Operativo si definisce **SISTEMA OPERATIVO MONOPROGRAMMATO**, poiché *gestisce i programmi in modo sequenziale nel tempo*. Tutte le risorse hardware e software del sistema sono dedicate ad un solo programma per volta → **BASSO GRADO DI UTILIZZO DELLE RISORSE**



• CON MULTIPROGRAMMAZIONE

I programmi danno luogo a due processi concorrenti (o paralleli) le cui esecuzioni si avvicendano sulla CPU.



Confronto

Supponiamo di dovere mandare in esecuzione due programmi P0 e P1 (due processi), ognuno con le seguenti caratteristiche:

- Durata complessiva di 1 minuto (dallo *start* allo *stop*)
- 50% di attività di CPU (*in verde*) e il 50% di attesa di dati (*in nero*)

Caso 1: Assenza di Multiprogrammazione

- Durata complessiva dei due processi: 2 minuti (1 minuto per P0 + 1 minuto per P1)
- Occupazione della CPU: 1 minuto (grado di utilizzo = 50%)
- Throughput della CPU (cioè numero di programmi completati nell'unità di tempo): 1 al minuto

Caso 2: Presenza di Multiprogrammazione

- Durata complessiva dei due processi: 1 minuto
- Occupazione della CPU: 1 minuto (grado di utilizzo = 100%)
- Throughput della CPU (cioè numero di programmi completati nell'unità di tempo): 2 al minuto → caso teorico

Nella realtà: grado di utilizzo e throughput più bassi.

Vantaggi

- Migliore utilizzo delle risorse (riduzione dei tempi morti)

Svantaggi

- Maggiore complessità del S.O.:
 - Algoritmi per la gestione delle risorse (CPU, memoria, I/O)
 - Protezione degli ambienti dei diversi processi (in un sistema *monoprogrammato* c'è un problema di protezione per il sistema operativo solo nei confronti dell'unico programma in esecuzione)

1.3.4 Definizione di Processo

Le attività che vengono svolte da un Sistema Operativo quando un programma è in esecuzione determinano un *flusso di esecuzione* che viene denominato **PROCESSO** (entità attiva definita dal S.O. per eseguire un programma). Prima della sua esecuzione, quando si trova ancora in memoria centrale nella fase precedente a quella di *start*, è possibile fare riferimento ad esso con il termine **PROGRAMMA**.

1.4 Categorie di Sistemi Operativi

1.4.1 Sistemi Operativi Batch

I programmi devono essere sottomessi al sistema da parte dell'operatore attraverso dei comandi di sistema.

programma + comandi di sistema → **JOB**

Problema

- Mancanza di interazione fra utente e JOB in esecuzione → *debugging difficile*

Utilizzo

- Programmi che non richiedono interazioni
- Programmi con tempo di esecuzione molto alto

Esempi:

- buste-paga
- previsioni del tempo
- analisi statistiche

Scheduling dei JOB

In genere, First COME First SERVED (FCFS o anche FIFO).

Gestione Memoria

- Una parte → Monitor BATCH (parte del sistema operativo sempre residente)
- Il resto → JOB corrente in un'esecuzione in linea con quella sequenziale (in assenza di multiprogrammazione)

MONITOR BATCH

M
O
N
I
T
O
R



Protezione

La protezione viene semplificata. E' importante avere un *registro* che indica la *prima locazione di memoria* utilizzabile da un programma utente. Il programma utente viene caricato a partire da questa locazione e per tutto lo spazio di cui ha bisogno deve essere possibile contenerlo nell'area di memoria RAM a disposizione (le dimensioni sono fissate dalla dimensione fisica della memoria). Tutte le volte che viene prodotto un indirizzo dal programma, il S.O. andrà a controllare che questo sia maggiore o almeno uguale a quello segnato nel *registro di protezione*, questo andrà a garantire che il programma sta richiedendo l'accesso a zone di memoria che non sono utilizzate dal S.O. Questo concetto è maggiormente importante per gli *accessi in scrittura*, ma in generale viene realizzato anche per gli *accessi in lettura*, dove l'esecuzione del programma viene bloccata per tentativo di accesso a zone di memoria non autorizzate che non sono appartenenti al S.O.

Ogni volta che il Program Counter si muove per far eseguire il programma viene fatto un controllo sul registro di protezione?

Non viene controllato solo il valore del *Program Counter*. Esso punta a un indirizzo di un'istruzione che è in memoria centrale e sulla quale viene fatto un *fetch*. L'istruzione viene decodificata e poi deve essere eseguita. Nell'esecuzione di questa istruzione devono essere utilizzati dei valori presenti nei registri, che ad un certo punto saranno derivati dal caricamento di un valore che ha in memoria. Tutte le volte che c'è un indirizzamento alla memoria che possa essere, da parte del PC nei confronti di un'istruzione, o da parte dell'esecuzione di un'istruzione nei confronti dell'accesso in un'area di dati viene fatto questo controllo.

Gestione Periferiche

- Controllo di programma (polling sul dispositivo)
- Interrupt → il vettore delle interruzioni serve per recuperare l'indirizzo iniziale della routine di risposta all'interruzione specifica

1.4.2 Sistemi Operativi Multiprogrammati

MULTIPROGRAMMAZIONE → più programmi in memoria centrale

Un S.O. multiprogrammato gestisce *contemporaneamente* più programmi presenti nella memoria principale. Quando un processo è IDLE (inattivo) ne prende un altro e lo comincia a eseguire parzialmente.

La *contemporaneità* può essere:

- **Reale**

Esempio:

Avendo 3 CPU e 3 Programmi in memoria centrale, ciascuna può mandare in esecuzione un programma e avere così un processo attivo in ogni CPU e in questo caso i 3 processi sono *realmente* in esecuzione contemporaneamente.

- **Simulata**

Esempio:

Avendo un numero di CPU minore rispetto al numero di programmi (quindi al *grado di multiprogrammazione* → numero di programmi caricati in memoria centrale). In questo caso le CPU saranno assegnate per brevi lassi temporali, secondo apposite politiche di scheduling, ai vari programmi per eseguirli in parallelo.

Categoria di Processi e Terminologie

Programma in esecuzione → **PROCESSO**

- **PROCESSI I/O BOUND** → attività di I/O rilevante
- **PROCESSI CPU BOUND** → attività di CPU rilevante (sia in stato di *running* che di *ready*)

Prendendo come riferimento le immagini precedenti, per i processi **I/O bound** i tratti *neri* del disegno sarebbero preponderanti, mentre per i processi **CPU bound** quelli preponderanti sono quelli *verdi*.

MULTIPROCESSO → più processi attivi (implica di avere la multiprogrammazione). Da *non confondere* con il termine *multiprogrammazione*. Dal punto di vista teorico (in realtà non è mai così) con la multiprogrammazione si potrebbe anche non avere più processi attivi: il sistema potrebbe anche gestire un processo alla volta. Un programma descrive un *insieme infinito* di processi.

MULTIUTENTE → più utenti possono accedere contemporaneamente (*SISTEMI INTERATTIVI o CONVERSAZIONALI*).

Esempio:

Il sistema Windows non è multiutente, come accesso *contemporaneo* allo stesso sistema Windows nello stesso momento. I sistemi UNIX sono invece multiutente, in quanto permettono l'accesso contemporaneo allo stesso sistema UNIX da parte di più utenti.

Possibili combinazioni:

- Sistema multiprogrammato, ma non multiutente

Esempio

sistema real-time

- Sistema multiprogrammato e multiutente

Esempio

sistema time-sharing

1.4.3 Sistemi Special Purpose

Sistemi *dedicati ad applicazioni specifiche*, spesso con *hardware speciale* (dispositivi di I/O).

Esempio:

- Sistemi Real-Time

1.4.4 Sistemi General Purpose

Sistemi che servono per scopi generali (non specifici).

Esempio:

- Linux
- Windows

Sistemi che non sono stati scritti per applicazioni specifiche.

1.4.5 Sistemi Time-Sharing

Un S.O *time-sharing* fornisce a ciascun utente una **porzione di un sistema di calcolo**.

Ogni utente (→ *multiutenza*) ha i propri programmi in memoria (→ *multiprogrammazione*).

Funzionamento

Il sistema operativo garantisce l'esecuzione dei vari programmi (e quindi dei vari processi) assegnando un **quanto di tempo**, e per questa durata va in esecuzione un determinato processo che esegue lo specifico programma che fa parte di uno specifico utente. In questo modo un programma è in esecuzione per **al massimo un quanto di tempo** (*time slice*), definito dal sistema.

Al completamento del quanto, il controllo passa ad un altro processo (che esegue un altro programma), a rotazione. Se prima della scadenza del suo quanto, il processo/programma fa una richiesta di I/O, perde il controllo.

- Ogni utente ha l'illusione di avere la macchina dedicata completamente a lui (CPU, memoria, dispositivi I/O unicamente appartenenti a lui).

- **Astrazione** di più **macchine virtuali**

MACCHINA VIRTUALE → utilizzo di uno strumento che ci permette di installare un S.O diverso da quello di partenza. Su un hardware è presente un Sistema Operativo *ospite* e su di esso una macchina virtuale sulla quale è possibile farne girare un altro.

Problema

- Scheduling della CPU

1.4.6 Sistemi Operativi Misti

Sono sistemi che forniscono diverse possibilità:

- **Gestione Interattiva** → time-sharing (utenti ai terminali)
 - Una parte del S.O ragiona in termini di time-sharing perché ha bisogno di gestire utenti connessi con diversi terminali fisici (gestione del II tipo).
- **Gestione BATCH** (nei tempi morti)
 - Necessità di eseguire dei JOB che necessitano molto tempo e che quindi vengono eseguiti di notte dal sistema.
- **Gestione Real-Time**
 - Gestione di eventi che non possono essere rimandati.

1.5 Struttura di un Sistema Operativo

Esistono 3 tipi di approccio rispetto alla progettazione di un Sistema Operativo:

- Monolitici
- Layered (stratificati)
- Microkernel

1.5.1 Monolitici

I Sistemi Operativi monolitici sono costituiti da un *insieme di moduli* (non è detto che il programma sia un tutt'uno), *senza alcuna relazione gerarchica*.

- Ogni modulo può chiamare ogni altro modulo
- Tutti i moduli possono accedere indiscriminatamente a tutti i dati presenti nel kernel

Vantaggi

- Il progettista *NON* deve decidere quali parti del S.O. hanno bisogno di avere i privilegi per accedere all'HW (accessibile solo dal Kernel), tutte le parti hanno gli stessi privilegi.
- Parti differenti del S.O. possono cooperare facilmente

Svantaggi

- Cambiando un singolo modulo, potrebbe rompersi il sistema di chiamata dei moduli. Un modulo che chiamava quello che è stato modificato poteva fare delle assunzioni che dopo le modifiche potrebbero non essere più valide.
- L'interfacciamento fra le differenti parti di un S.O. è spesso reso più complicato e uno sviluppatore potrà più facilmente compiere degli errori. In un *Kernel Monolitico*, **un errore è fatale** perché si lavora sempre in *kernel mode* e quindi si blocca tutto.

Esempio:

UNIX. L'intero Sistema Operativo risiede nel kernel e tutte le *system call* eseguono in *kernel mode*.

N.B: Un certo grado di modularità viene garantito dal fatto che i più recenti kernel monolitici (es. *LINUX*) permettono di effettuare il caricamento di moduli eseguibili a runtime → è possibile estendere le potenzialità del kernel, solo su richiesta.

1.5.2 Stratificati

In questi Sistemi Operativi esiste una *relazione gerarchica tra i moduli (layer)* che compongono il Sistema Operativo. Questa stratificazione presenta un *protocollo* che fa sì che:

- Ad ogni modulo è assegnato un numero che indica il livello di appartenenza
- Un modulo di livello N potrà accedere solo a funzionalità e dati presenti al livello sottostante (N-1)

In questo caso il sistema risulta di più facile manutenzione anche se meno efficiente del sistema monolitico. E' possibile cambiare tutti i moduli di un certo livello senza bisogno di cambiare i moduli del livello sottostante, e nemmeno quelli del livello sovrastante se viene garantita che l'interfaccia data ad essi sia sempre la stessa. E' dunque possibile cambiare l'intera implementazione di un livello senza che questa abbia effetto sugli altri, cosa *contraria* a quanto visto nella *struttura monolitica*.

Il Sistema Operativo può essere progettato come una gerarchia di **livelli di astrazione**: per ogni livello possono essere usate tecniche di *information hiding* → **modularità**

1.5.3 Microkernel

In questo tipo di Sistemi Operativi sono contenuti solo ed esclusivamente i *servizi indispensabili per il funzionamento dell'intero sistema*. In particolare, questa struttura gestisce:

- Processi
- Messaggi → meccanismo di comunicazione tra processi

I *restanti servizi* (gestione memoria, file system, gestore finestre, ecc...) sono considerati processi utente: non eseguono a livello di kernel, ma eseguono *sopra alla microkernel*. L'approccio è opposto a quello monolitico dato che si va a minimizzare la parte di S.O. che esegue in kernel mode e la maggior parte del S.O. esegue in user mode.

Vantaggi

- Kernel molto contenuto → manutenzione più facile, meno errori
- Codice eseguito in kernel mode molto contenuto
- S.O. nel complesso più facilmente estendibile e personalizzabile → è sufficiente aggiungere nuovi servizi che sono visti come processi utente.

Svantaggi

Basse prestazioni. Tutte le volte che il gestore della memoria o il file system ha bisogno di operare a livello fisico sull'hardware deve interfacciarsi con il microkernel facendo una richiesta (inviando dunque un messaggio) per poter accedere ad un dispositivo di memorizzazione secondario, piuttosto che alla memoria o altro.

Nei S.O. a microkernel le applicazioni che implementano i diversi servizi possono essere pensate come dei server che forniscono funzionalità agli altri server o alle applicazioni utente che in questo caso operano come client: per questo motivo i sistemi microkernel sono chiamati anche ***client-server OS***.

1.6 Componenti di un Sistema Operativo

Questi componenti devono gestire aspetti specifici del sistema operativo. Svolgono diverse funzioni:

- **PROCESSOR/PROCESS MANAGEMENT (NUCLEO o KERNEL)**
 - Gestione processore tramite processi,
 - Sincronizzazione e comunicazione
- **I/O SYSTEM**
 - gestione dei dispositivi di I/O
- **MEMORY MANAGEMENT**
 - gestione della memoria primaria (centrale → RAM)
- **FILE SYSTEM**
 - gestione della memoria secondaria
- **PROTECTION SYSTEM**
 - gestione degli accessi
- **NETWORKING**
 - gestione di sistemi distribuiti
- **COMMAND INTERPRETER SYSTEM**
 - gestione interfaccia utente → interprete del linguaggio comandi

1.6.1 Gestore dei Processi

Il processo è l'unità di lavoro in un Sistema Operativo multiprogrammato. In generale, un Sistema Operativo gestisce una *collezione di processi*:

- processi del S.O. che *eseguono il codice del sistema*
- processi utente che *eseguono il codice di utente*

Questi due tipi di processi possono essere in esecuzione **concorrentemente** (*reale* con più CPU, *simulata* con una sola CPU).

Gestione del Processore

Il Sistema Operativo si deve occupare della politica di assegnazione del processore (o dei processori) ai vari processi → ***scheduling dei processi***

Funzioni Svolte

- Creazione e distruzione di processi
- Sospensione e riattivazione di processi
- Strumenti per la sincronizzazione e la comunicazione di processi
- Scheduling dei processi → vari algoritmi a seconda delle esigenze del S.O e/o degli utenti

1.6.2 Gestione dei Dispositivi I/O

Nasconde al programmatore i dettagli e le peculiarità dei dispositivi di I/O (periferiche, memoria di massa, ecc...), in particolare:

- Gestione a controllo di programma (polling)
- Gestione ad interruzioni (interrupt) → routine di risposta alle interruzioni

Funzioni Svolte

- Routine di risposta alle interruzioni se il dispositivo è trattato come interrupt
- Gestione dei buffer di I/O

1.6.3 Gestione della Memoria Centrale

Si tratta della *gestione degli spazi di indirizzamento dei processi* (sistemi operativi multi programmati e time-sharing, e quindi multiprocessing). Il grado di multiprogrammazione, considerando la sola memoria RAM, potrebbe essere limitato.

MEMORIA VIRTUALE → il programmatore ha l'**illusione** di avere a disposizione una memoria più grande di quella fisica.

Funzioni Svolte

- Allocazione e deallocazione di memoria
- Gestione degli spazi di indirizzamento di ogni processo
- Gestione della memoria virtuale (non si dovrà utilizzare la sola memoria RAM, ma anche dischi ad accesso veloce)

1.6.4 Gestione della Memoria Secondaria

L'informazione può essere memorizzata su diversi dispositivi (dischi, nastri, cassette, ecc...), all'utente non interessa *dove* l'informazione viene salvata. Ciascun dispositivo di memorizzazione ha le proprie caratteristiche e la propria organizzazione fisica. Il Sistema Operativo **astrae** dalle caratteristiche fisiche introducendo il concetto di file e di file system.

FILE → unità di memorizzazione logica, insieme di informazioni logicamente correlate.

Esempio:

- Programmi (codice sorgente, codice oggetto, codice eseguibile)
- Dati (numerici, alfabetici, alfanumerici)

FILE SYSTEM → organizzazione dei file (directory)

- ad un solo livello
- a più livelli

Funzioni Svolte

- Creazione e cancellazione di directory
- Creazione e cancellazione di file
- Operazioni di trattamento dei file
- Piazzamento dei file sulla memoria secondaria (corrispondenza delle informazioni sulla memoria secondaria, per la quale all'utente viene fornita un'astrazione).

1.6.5 Gestione Interfaccia Utente

Questo livello **interpreta i comandi** dati al Sistema Operativo (richiesti dall'utente) → *costituisce l'interfaccia fra il Sistema Operativo e l'utente.*

I comandi sono relativi a tutte le funzioni viste:

- Gestione dei processi
- Gestione dell'I/O
- Gestione della memoria
- Gestione del file system
- Gestione della protezione
- Gestione dei sistemi distribuiti

A questo livello si collocano anche gli strumenti che sono offerti per il supporto ai linguaggi di programmazione

- COMPILATORI
- INTERPRETI

1.6.6 Gestione della Protezione

Meccanismi e politiche per controllare l'accesso dei processi alle risorse del sistema di calcolo.

Elemento più importante del Sistema Operativo.

MECCANISMI → realizzati via *hardware*. Unità fondamentale del gestore di protezione realizzato via hardware.

In particolare:

Modi differenti di Funzionamento

Il punto di partenza per realizzare la protezione a livello hardware è rappresentato dal **bit di modo**: *monitor* (0) - *utente* (1).

Consente di stabilire se l'istruzione corrente è eseguita per conto del S.O. o dell'utente (si definisce *stato monitor* o *stato utente*). Ogni volta che si verifica un interrupt oppure si richiede un servizio del S.O. (tramite una system call), l'hardware passa da modo utente a modo supervisore (o privilegiato, detto anche kernel mode) → le istruzioni macchina che possono causare danni allo stato del sistema (a livello hardware possono fare qualsiasi cosa) sono definite come **istruzioni privilegiate** che possono essere eseguite solo in modo supervisore.

I/O

Il livello di gestione dell'I/O impedisce l'accesso diretto ai dispositivi da parte degli utenti poiché le istruzioni di I/O sono privilegiate.

Memoria

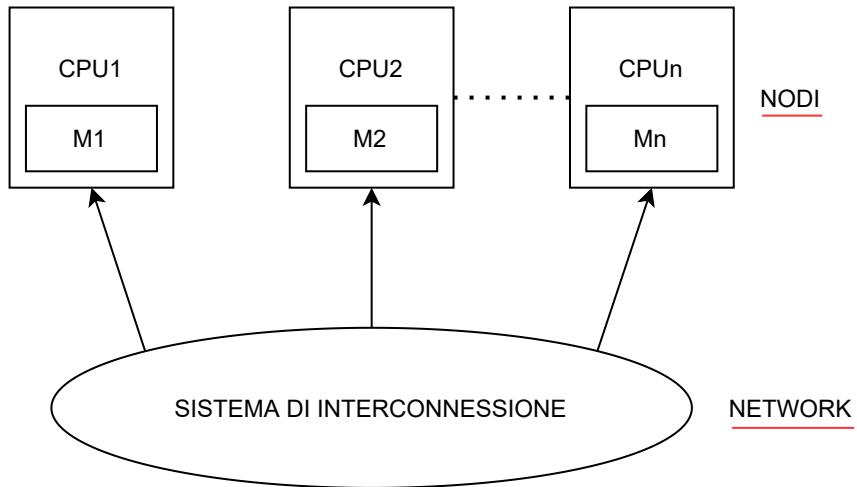
Hardware di indirizzamento della memoria assicura che un processo possa operare solo entro il proprio spazio di indirizzamento → ad esempio con **registri base e limite**.

1.7 Problematiche di Gestione dei Sistemi Distribuiti

Se a livello architettonico è presente una rete, il Sistema Operativo si deve occupare anche della gestione di questa rete.

1.7.1 Architettura Distribuita (Loosely Coupled Architecture)

Insieme di processori (CPU) che **non condividono né la memoria né il clock**. Ogni processore ha la sua memoria locale (e il suo clock) e comunica con gli altri solo mediante linee di comunicazione.

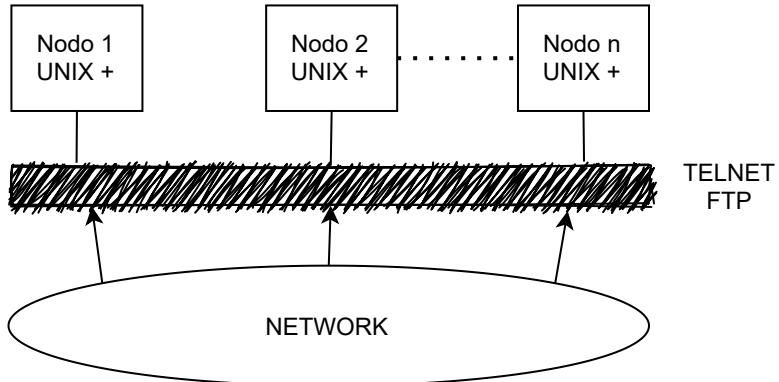


Assenza di una memoria comune. Ciò comporta più problemi: considerando che il Sistema Operativo debba andarsi ad appoggiare sull'hardware (architettura distribuita) quello che si dovrà andare a definire è un *sistema distribuito*. Questo sistema non è di facile progettazione proprio a causa dell'*assenza di una memoria comune*, si hanno *tante memorie locali* alle varie CPU e quindi non è facile stabilire la parte di S.O che deve essere sempre caricata in memoria centrale in quale memoria si trova. La *soluzione migliore* rimane l'implementazione di un **Sistema Operativo di Rete**.

Esempio:

Attuali versioni di UNIX (LINUX) sono *sistemi operativi di rete*.

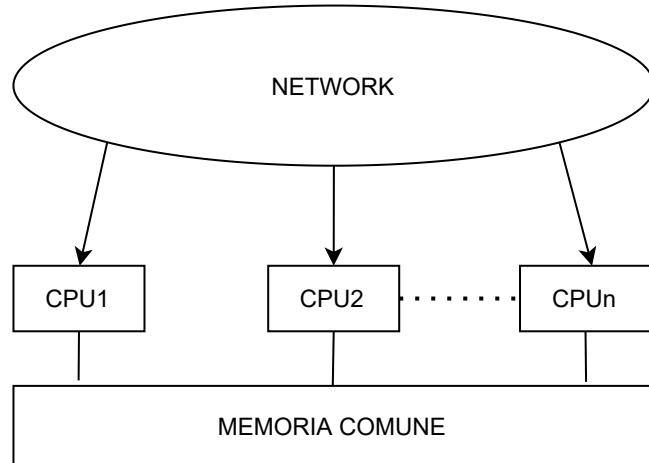
Sistema Operativo di RETE → Si tratta di un normale sistema operativo sviluppato per ragionare su un'unica CPU o più CPU in memoria comune, sul quale si va ad aggiungere uno *strato software* per interfacciarsi con la rete, aggiungendo servizi come *FTP*.



In questo caso il S.O non è propriamente distribuito, in quanto deve nascondere del tutto la natura distribuita dell'architettura, ma sono S.O di rete che possono gestire alcuni aspetti dell'architettura distribuita ma non necessariamente nasconderne la distribuzione. Eventualmente l'utente può essere a conoscenza che il proprio processo stia girando su un certo nodo dell'architettura piuttosto che un altro.

1.7.2 Architettura Multiprocessore (Tightly Coupled Architecture)

Insieme di processori (CPU) che condividono la memoria. Ogni processore comunica con gli altri o tramite la memoria comune (condivisa) oppure mediante linee di comunicazione (network).



Il Sistema Operativo può essere allocato nella memoria comune, e in seguito esso nasconderà tutte le caratteristiche relative alla presenza di più CPU e di una rete.

La rete di comunicazione può essere configurata in modi diversi:

- Bus
- Connessioni punto a punto
 - Rete totalmente connessa
 - Rete parzialmente connessa

In entrambi i casi nascono problematiche di **NETWORKING**. Rispetto a un *sistema centralizzato*, dove il **network non è presente**, che può degenerare avendo un'unica CPU e un'unica memoria, *possono verificarsi i seguenti problemi*:

- Problemi di routing (strategie di connessioni)
- Problemi di sicurezza e di accesso contemporaneo in relazione alla rete di comunicazione
- Problemi di allocazione dei processi ai processori
 - Statica (dove il *processo viene assegnato per sempre allo stesso processore*) o dinamica (dove può essere che il processo cominci a eseguire su un processore e in seguito venga spostato)
 - Trasparenza o meno dell'allocazione: l'utente può non sapere dove viene eseguito il proprio processo, oppure richiedere che l'esecuzione avvenga su un processore specifico (*predilezione del processore*). Se questo concetto viene a meno e può trattare *tutto il S.O* possono essere presenti delle strategie di **LOAD BALANCING** → migrazione di processi. Questi vengono allocati in modo da bilanciare il carico sia a livello di computazione che di comunicazione.

1.8 Studio di un Sistema Operativo

PUNTO DI VISTA DELL'UTENTE → esterno

PUNTO DI VISTA DEL S.O → interno

1.8.1 Punto di Vista dell'Utente

I servizi del Sistema Operativo ad un programmatore sono offerti in modi diversi a seconda se l'utente è:

1. **UN PROGRAMMATORE** (livello di programmazione) → **CHIAMATE DI SISTEMA** (**SYSTEM CALL**)
2. **UN UTENTE** (vero e proprio, livello utente) → **PROGRAMMI DI SISTEMA** (compilatore, programmi che si utilizzano senza conoscenza delle richieste fatte al S.O.)

Chiamate di Sistema (**SYSTEM CALL o SYSCALL**)

In ambito UNIX vengono definite *chiamate primitive*. Sono funzioni implementate nel S.O. che possono essere usate dai processi (programmatori) in *user space*:

- Da un punto di vista concettuale un programmatore che vuole utilizzare una certa funzione del S.O chiama una syscall come una normale funzione.
- La differenza sostanziale (rispetto a una chiamata di funzione) è che l'indirizzo della syscall è presente in una **tabella del kernel** e nel momento in cui si effettua la chiamata il processore entra in *Kernel Mode* → tutte le chiamate di sistema producono così un *cambio del bit di modo*
- Tutti i processori contengono nel loro ISA (*Instruction Set Architecture*, set di istruzioni a basso livello) un'istruzione per effettuare una syscall
 - x86: int/int → chiamata di interrupt software che prevede una *iret*, ovvero un ritorno
 - MIPS: syscall (istruzione hardware)
 - In altri casi: trap (al S.O)

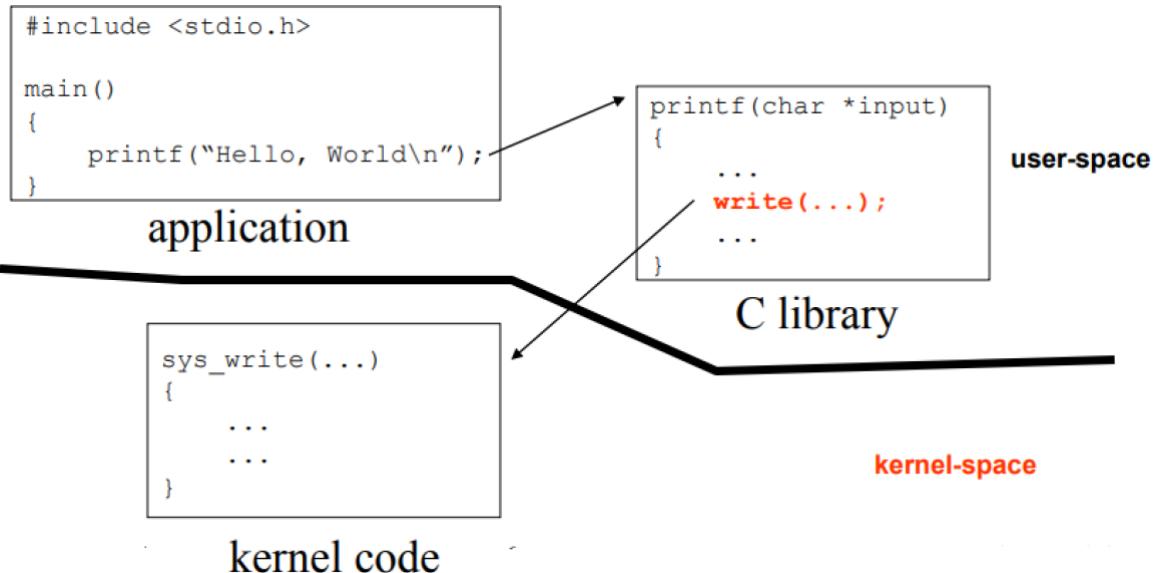
Possono definirsi come ***l'interfaccia*** con cui i processi operanti in *user space* possono:

- Accedere all'hardware e alle altre risorse gestite dal sistema operativo
- Comunicare con il kernel

Pertanto, le *system call* costituiscono uno ***strato tra hardware e processi in user space***, che:

- Fornisce un'interfaccia per semplificare l'uso delle risorse *HW* da parte dei processi *user space*
- Assicura la sicurezza del sistema → È il kernel che decide chi e come può accedere alle diverse risorse del sistema. In questo modo non è possibile che un processo utente possa andare a scrivere in un'allocazione in cui è presente la tabella dei processi del sistema.

La chiamata di una system call da parte di un'applicazione avviene attraverso la chiamata di opportune funzioni di **libreria**, che a loro volta si rifanno all'interfaccia della system call per svolgere il loro compito.



Esempio:

Il programmatore sa che per fare un'azione sullo *standard output* deve chiamare la *funzione di libreria printf*. Questa funzione, invocata in user space, dopo che vengono fatte delle operazioni relative alla formattazione della stringa da stampare, invoca una syscall (funzione *write*), che nel kernel corrisponde all'andare a invocare la *funzione write*.

Ad ogni system call è assegnato un numero che la contraddistingue, e quindi il kernel riconosce le syscall per numero e non per nome → una volta assegnato il *numero* ad una system call, *non può più essere mutato*, altrimenti le applicazioni precedentemente compilate non troverebbero la corretta corrispondenza.

N.B: Se una syscall viene rimossa il suo identificativo non potrà essere “riciclato”.

Esempio:

Il kernel Linux mantiene una lista di tutte le syscall registrate in una tabella memorizzata in *sys_call_table* in *entry.S* (Assembler Source Code)

In generale una System Call ha:

- Uno o più argomenti di input
- Un valore di ritorno che indica la corretta esecuzione o meno → Di solito un valore di ritorno pari a 0 (zero) è indicazione di successo (si ricordi la semantica UNIX), mentre un qualunque *valore diverso da 0 denota un errore*.

Passaggio dei Parametri

- Molte syscall richiedono parametri per poter essere eseguite
- I *parametri sono scritti nei registri di CPU* prima della chiamata della syscall
- Se sono richiesti più argomenti, gli stessi sono passati attraverso stack o puntatori (il valore del puntatore viene caricato nel registro e in seguito si va a considerare la zona di memoria puntata dal puntatore).

Librerie vs System Call

Le system call vengono mappate in un *sottoinsieme delle librerie*. Rispetto alle system call, le funzioni di libreria possono svolgere una delle seguenti funzioni

- *Wrapper* alla system call, cioè fornisce *elaborazioni supplementari* a quelle fornite dalla system call.

Esempio:

La funzione di libreria `printf()` effettua le formattazioni dei dati prima di chiamare la system call `write()` che scrive i dati su standard output.

- Relazione 1-1 con la system call, cioè chiama esclusivamente la system call di competenza

Esempio:

`open()` per i file

- Non chiamano nessuna system call

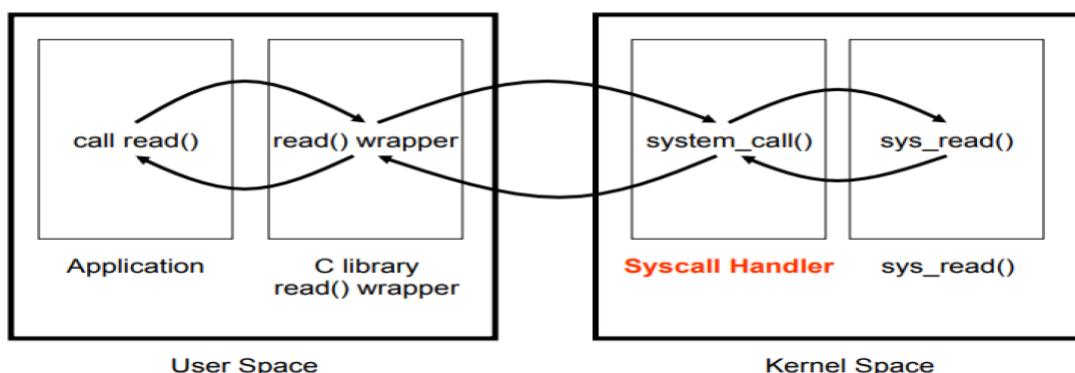
Esempio:

`strcpy()`

Libreria Standard C

La libreria standard del C (`libc`) contiene funzioni di utilità che forniscono servizi general purpose al programmatore queste funzioni *non sono chiamate dirette a servizi del kernel*, sebbene alcune di esse possano fare uso delle system call per realizzare i propri servizi. Questa libreria implementa:

- La principale API (Application Programming Interface) sui sistemi Unix
- L'interfaccia per le chiamate di sistema → `libc` fornisce anche la maggior parte dell'API POSIX (Portable OS interface).



Programmi di Sistema

A livello utente è disponibile una *collezione di programmi di utilità*, in particolare per sviluppare un programma:

- Compilazione
- Collegamento (linking)
- Caricamento
- Esecuzione
- Debugging

L'interfaccia utilizzata dell'utente è la **command interpreter** (*processore comandi*, nel caso di LINUX è la *shell*)

1.8.2 Punto di Vista del Sistema Operativo

Un Sistema Operativo è un **programma event driven** (guidato dagli eventi). Sostanzialmente, non esegue azioni fino a quando non succede effettivamente qualcosa. Non prende iniziativa autonome ma reagisce a delle richieste, si pone in attesa per:

- Lavori da eseguire

- Dispositivi da gestire
- Utenti a cui rispondere

Tipicamente un evento è rappresentato dal verificarsi di un **INTERRUPT** → al verificarsi di un interrupt, il controllo passa al S.O.

INTERRUPT → particolare richiesta al sistema operativo da parte di un processo in esecuzione.

Esistono tipi diversi di interrupt:

- **Asincroni** derivanti da un dispositivo di I/O (quindi da hardware), definito asincrono rispetto alla normale esecuzione del programma in esecuzione sulla CPU
- **Sincroni**
 - Derivanti da errori verificatisi durante la esecuzione

Esempio:

overflow, divide by 0, tentativi di accesso ad aree di memoria non consentite

- Derivanti da una system call

Il trattamento di questo interrupt produce un *cambio di visibilità*: si passa *da modalità utente a modalità supervisore*.

Distinzione fra Meccanismi e Politiche

- Le **politiche** decidono *cosa debba essere fatto*
- I **meccanismi** attuano la decisione stabilita da una certa politica: sono l'oggetto alla quale la politica si riferisce.

È un concetto fondamentale del software engineering: avere due componenti distinte per prendere decisioni sulle *politiche* e per implementare i meccanismi rende possibile cambiare uno senza cambiare l'altro.

Esempio: (non relativo ai S.O.)

meccanismo → denaro

politica → come il denaro viene speso

Esempio:

meccanismo → interrupt asincrono. Un certo dispositivo richiede l'attenzione della CPU, interrompendone l'esecuzione.

politica → routine di risposta all'interruzione. La CPU, sulla base del numero di interrupt, va ad eseguire una routine di risposta all'interruzione.

1.9 Progettazione di un Sistema Operativo

Prima di partire con la progettazione, si deve partire con la definizione del problema:

- Definire gli **obiettivi** del S.O. che si vuole realizzare
- Definire i **vincoli** entro cui il S.O. deve operare

La progettazione sarà *influenzata*:

- A livello più basso, *dal sistema hardware* sul quale si va ad eseguire
- A livello più alto, dalle *applicazioni* che devono essere eseguite dal Sistema Operativo

A seconda di queste condizioni, il S.O. sarà: *batch*, *time-shared*, *single-user*, *multi-user*, *distribuito*, *general-purpose*, *real-time*, ...

Bisogna poi considerare:

- Richieste dell'utente → *comodo da usare* (user friendly), *facile da imparare*, *robusto* (senza crash o interruzioni), *sicuro* (i dati che l'utente va ad utilizzare devono essere mantenuti in modo privato), *veloce*
- Richieste degli sviluppatori → *facile da progettare*, *da mantenere e da aggiornare*, *veloce da implementare*

1.10 Installazione di un Sistema Operativo

Una volta che il S.O. sia stato progettato e quindi implementato bisogna installarlo: normalmente il S.O. è implementato in modo da garantirne la **portabilità**:

- Lo stesso Sistema Operativo viene spesso previsto per architetture hardware differenti
- Possono essere equipaggiati con dispositivi periferici molto diversi, e spesso anche diverse architetture di CPU e BUS → bisogna prevedere meccanismi per la generazione del S.O. specifico per l'architettura utilizzata

I parametri tipici per la generazione di un sistema operativo sono:

- *Tipo di CPU* utilizzata (o di CPU utilizzate)
- *Quantità di memoria centrale*
- *Periferiche utilizzate*
- *Parametri numerici* di vario tipo (numero utenti, processi, ampiezza dei buffer, tipo di processi, ...)

I metodi che possono essere utilizzati per la generazione/installazione sono:

- Rigenerazione (nel senso di *ricombinazione*) del kernel con i nuovi parametri/driver
- Prevedere la gestione di moduli aggiuntivi collegati durante il boot → non si ha una rigenerazione di tutto il kernel ma si parte da una situazione preesistente: all'accensione del sistema si vanno a caricare dei moduli aggiuntivi, in particolare se si devono andare a trattare delle periferiche che prima non c'erano (ciò è reso possibile da alcune tecniche particolari di *caricamento dinamico*, che consentono di caricare determinate librerie come *extension MacOS*, *DLL Windows*, *moduli Linux*).

2.1 Gestione dei Processi (e Processori)

Questo gestore normalmente prende il nome di **KERNEL** o **NUCLEO**. Nel caso in cui la strutturazione sia monolitica tutto collassa nel *nucleo* (che coincide con il sistema operativo). Questa *macchina astratta* che comprende l'hardware a disposizione deve mettere in campo:

- un *meccanismo di esecuzione di processi sequenziali*
- uno o più *meccanismi di sincronizzazione*

Questa macchina (kernel o nucleo) può essere anche realizzata:

- totalmente in *hardware* (in genere non è così)
- *software* (una parte consistente)
- *firmware*

Si analizzano di seguito i concetti essenziali del nucleo per prima cosa dal **punto di vista ESTERNO** (utente programmatore).

2.1.1 Algoritmo, Programma, Processo

ALGORITMO → procedimento logico che deve essere seguito per risolvere il problema in esame

PROGRAMMA → descrizione dell'algoritmo tramite un opportuno formalismo (*linguaggio di programmazione*), che rende possibile l'esecuzione dell'algoritmo da parte di un particolare elaboratore. Parlando di un linguaggio di programmazione, si dovrà passare per le fasi opportune, quali compilazione, collegamento (linking) fino all'ottenimento del formato eseguibile, partendo dal programma scritto in forma sorgente. Se il linguaggio è interpretato come linguaggio shell, mandiamo direttamente in esecuzione il programma senza tutte le fasi necessarie in un linguaggio di programmazione compilato.

Un programma *descrive un insieme* (possibilmente infinito) di processi, ognuno dei quali è relativo all'esecuzione del programma da parte del processore per un determinato insieme di dati in ingresso. *Ogni volta che si richiede l'esecuzione di un programma si genera un nuovo processo.*

PROCESSO (sequenziale) → sequenza di eventi a cui dà luogo un elaboratore quando opera sotto il controllo di un particolare programma.

Differenza fra Programma e Processo

Un programma (entità statica, prima memorizzato come file nel file system e poi caricato in memoria centrale) può essere eseguito da più processi (entità dinamiche).

Un processo è un programma in esecuzione

2.1.2 Rappresentazione di un Processo

Sequenza di **stati** attraverso i quali passa l'elaboratore durante l'esecuzione di un programma.

Esempio:

Massimo Comune Divisore (M.C.D.) di x e y (numeri naturali)

```

a = x;
b = y;
while (a != b) {
    if (a > b)
        a = a - b;
    else b = b - a;
}

```

x	18	18	18	18	18	18
y	24	24	24	24	24	24
a	-	18	18	18	12	6
b	-	-	24	6	6	6

a = 6, b = 6 → M.C.D dei due numeri dati in ingresso: 18, 24

STATO → espresso dai valori delle variabili

La rappresentazione tabellare non è particolarmente comoda, perché se si dovesse rappresentare un programma più complesso questa diventerebbe molto grande e di difficile interpretazione. A questa, vengono preferite altre modalità di rappresentazione.

Grafo di Precedenza

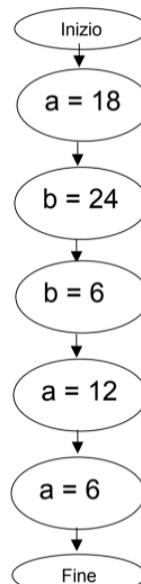
Un processo può essere rappresentato tramite un *grafo orientato* → *grafo di precedenza* del processo.

I *nodi* del grafo rappresentano i singoli eventi del processo, mentre gli *archi* identificano le *precedenze temporali tra tali eventi*.

Processo Sequenziale

Il grafo di precedenza è a *ordinamento totale* cioè ogni nodo (fatta eccezione per quello iniziale e quello finale) *ha esattamente un predecessore ed un successore*.

Esempio: (riprende *Esempio M.C.D*)



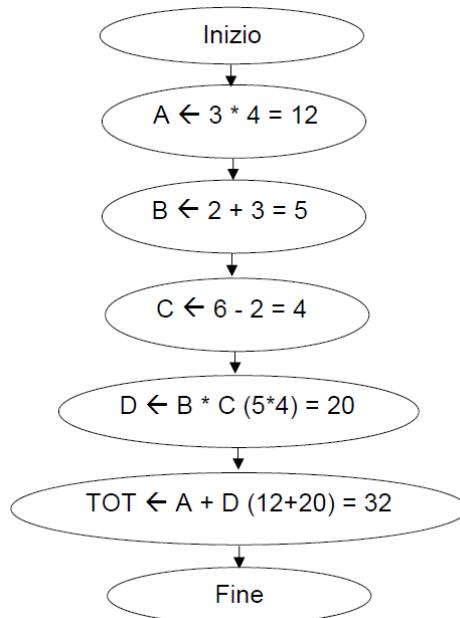
Per vedere che questa non è l'unica possibilità in cui si sviluppa un programma che porti ad ottenere un grafo di precedenza a ordinamento totale e quindi a un processo sequenziale, passiamo a considerare un *formalismo matematico*.

Esempio:

Il programma corrisponde alla valutazione dell'espressione:

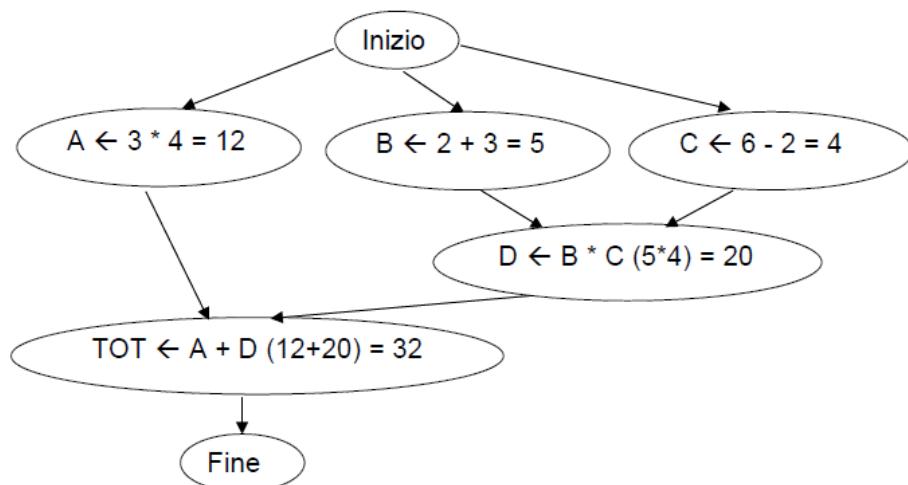
$$(3 * 4) + (2 + 3) * (6 - 2)$$

GRAFO DI PRECEDENZA AD ORDINAMENTO TOTALE



GRAFO DI PRECEDENZA AD ORDINAMENTO PARZIALE

Nel caso in cui si è in di più nella risoluzione dell'espressione. Con una suddivisione dei compiti, l'espressione può essere valutata svolgendo in parallelo i calcoli. Prima di calcolare D deve essere posto il vincolo di aver prima calcolato B e C. Con l'ordinamento parziale, un nodo può avere più di un predecessore o un successore.



ARCO → vincolo di precedenza tra gli eventi

VINCOLO DI SINCRONIZZAZIONE → ordinamento di eventi

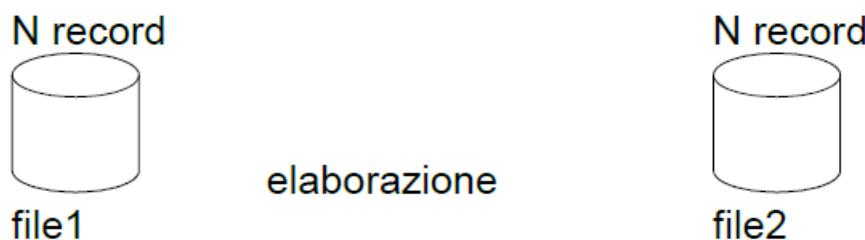
Processo non Sequenziale

L'**ordinamento totale** di un grafo di precedenza può derivare dalla natura sequenziale di un processo, cioè può essere implicito nel problema da risolvere (il calcolo dell'MCD è possibile ottenerlo solo con un grafo a ordinamento totale). Altre volte, invece, l'ordinamento totale può essere imposto dalla natura sequenziale dell'elaboratore e non essere insito nel problema (come nel caso del formalismo matematico precedente, dove la natura del programma è tale per cui potrei ottenere un grafo di precedenza a ordinamento parziale, e quindi un *processo non sequenziale*).

Esistono molti esempi di applicazioni che potrebbero essere più naturalmente rappresentate da processi *non sequenziali* → cioè processi tra i cui eventi non esiste un ordinamento totale, ma solo *parziale*.

Esempio:

Elaborazione di dati su un file sequenziale (composto da N record).

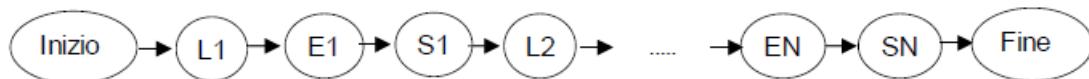


```
T buffer;  
int i;  
  
for (i = 1; i <= N; i++) {  
    lettura(buffer);  
    elaborazione(buffer);  
    scrittura(buffer);  
}
```

Scrivendo il programma in questi termini, senza ragionare attraverso compilatori paralleli, quello che otteniamo è un'esecuzione che da luogo a un processo sequenziale e quindi a un *grafo di precedenza a ordinamento totale*.

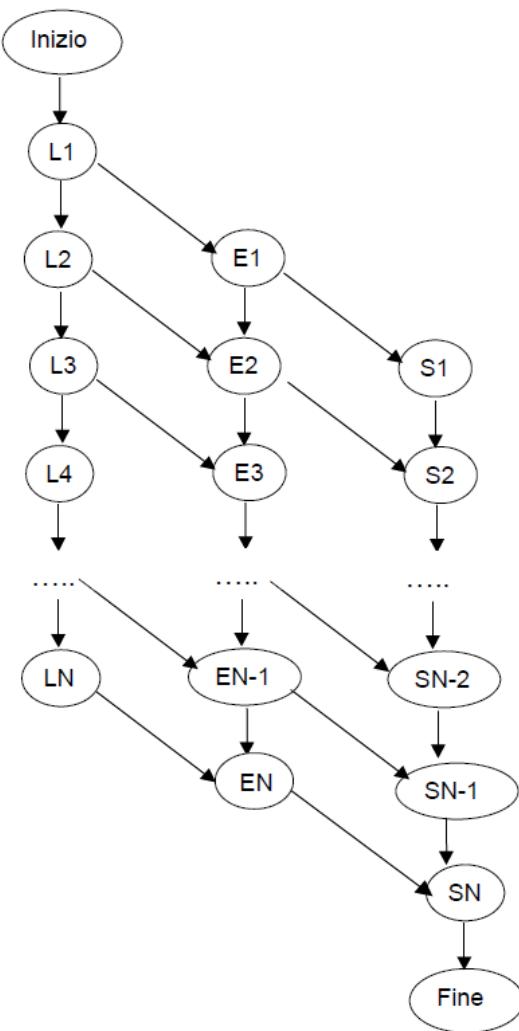
GRAFO DI PRECEDENZA AD ORDINAMENTO TOTALE

- L → lettura
- E → elaborazione
- S → scrittura



GRAFO DI PRECEDENZA AD ORDINAMENTO PARZIALE

In realtà il problema non è necessariamente sequenziale, pertanto è anche possibile ottenere un *grafo a ordinamento parziale* in cui rimangono dei *vincoli di sequenzialità* (sulle letture, sulle elaborazioni e sulle scritture). Una volta fatto partire il tutto, è possibile arrivare alla lettura del terzo dato che può avvenire in parallelo con l'elaborazione del secondo dato e la scrittura del primo: non c'è nessuna ragione insita nella natura del problema che lo vietи.



L'esecuzione di un processo non sequenziale richiede:

- un elaboratore non sequenziale
- un linguaggio di programmazione non sequenziale (oppure degli strumenti che automaticamente producono da un programma scritto in modo sequenziale dal programmatore, un programma non sequenziale)

Elaboratore non Sequenziale

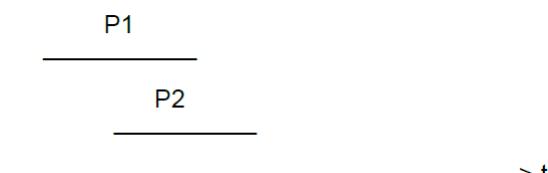
In grado di eseguire più operazioni contemporaneamente → la *contemporaneità* può essere **reale (A)** o **simulata (B)**.

Esempio:

In relazione all'esempio precedente, la *lettura*, *elaborazione* e *scrittura* di dati diversi in contemporanea. E' necessaria una CPU differente per svolgere ciascuno di questi compiti.

A) PIÙ ELABORATORI

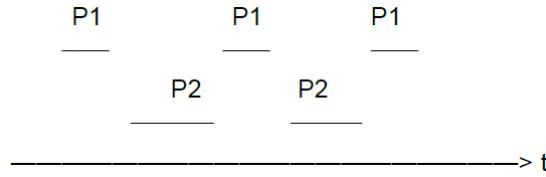
- architettura multiprocessore (più processori, memoria comune)
- architettura distribuita (più processori ognuno con memoria privata)



B) UN SOLO ELABORATORE

- architettura monoprocesso

MULTIPROGRAMMAZIONE → quasi parallelismo



Si tratta di un sistema di elaborazione in grado di eseguire processi non sequenziali. Deve fornire:

- una serie di **unità di elaborazione** (*fisiche*, quindi distinte l'una dall'altra, o *virtuali*)
- **meccanismi di sincronizzazione** per imporre i vincoli di precedenza e per consentire l'interazione tra i processi. Il processo di elaborazione deve ricevere il dato su cui deve operare.

Linguaggi di Programmazione non Sequenziale

Linguaggi CONCORRENTI	Linguaggi SEQUENZIALI + Funzioni di Libreria (Primitive)
Concurrent Pascal (<i>process</i>)	C per UNIX (<i>fork()</i> , ecc...)
ADA (<i>task</i>)	
Java (<i>thread</i>)	

Linguaggi Concurrenti:

Hanno delle parole chiave nei linguaggi che consentono di definire i vari processi non sequenziali (indicati tra parentesi nella tabella).

Linguaggi Sequenziali:

Utilizzando un linguaggio sequenziale su dei sistemi, vengono utilizzate delle *funzioni di libreria specifiche* che si interfacciano con le opportune *system call* (primitive) del Sistema Operativo, le quali consentono di creare dei processi (ad esempio con la parola chiave *fork*).

Per superare le difficoltà dovute alla complessità di un algoritmo non sequenziale → **scomposizione** di un processo non sequenziale in un **insieme di processi sequenziali** eseguiti "contemporaneamente" (contemporaneità reale o simulata)

Ogni processo sequenziale costituente (cioè derivato da tale scomposizione) è *analizzato e programmato separatamente*.

Esempio:

- *processo di lettura* (lettura del secondo dato garantita solo al completamento della lettura del primo)
- *processo di elaborazione*
- *processo di scrittura*

I processi possono essere eseguiti **contemporaneamente**, ma devono rispettare i **vincoli di precedenza** che esistono tra le operazioni dei vari processi → **SINCRONIZZAZIONE**

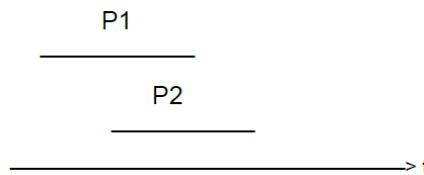
PROCESSI CONCORRENTI INTERAGENTI → processi sequenziali di cui è costituito un *processo non sequenziale*.

Processi Concorrenti

Definizione: Più processi (sequenziali) si dicono **concorrenti** se la loro esecuzione si sovrappone nel tempo. In generale, i processi che compongono un processo non sequenziale.

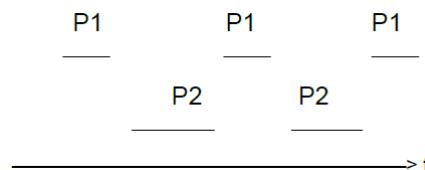
Due processi sono **concorrenti** se la **prima** operazione di uno inizia prima dell'ultima dell'altro. Indichiamo con t_1 l'istante di tempo in cui *finisce* il processo P_1 , mentre t_0 l'istante di tempo in cui *comincia* il processo P_2 . Utilizzando questa definizione non c'è bisogno di chiarire se si tratta di una sovrapposizione reale o simulata.

A) **OVERLAPPING** → *PARALLELISMO REALE* (architettura multiprocessore)



P1 e P2 sono eseguiti su due CPU distinte.

B) **INTERLEAVING** → *QUASI PARALLELISMO* (architettura monoprocessoresso)

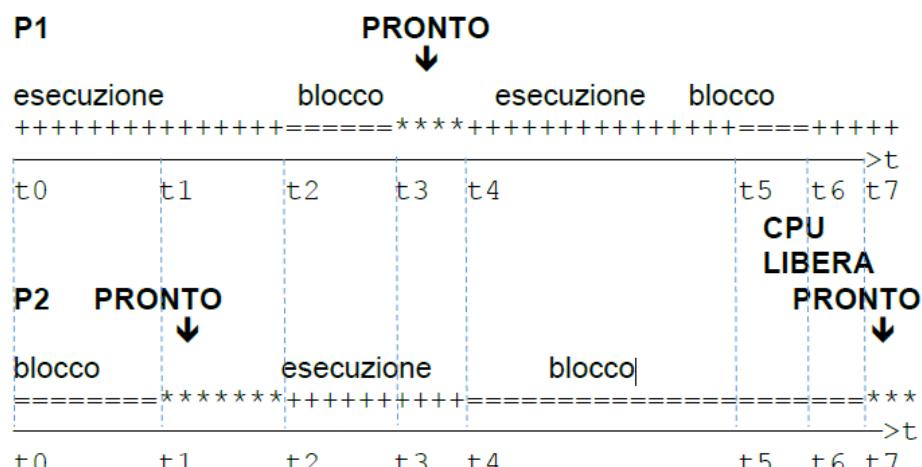


Utilizzando una sola CPU, potrebbe accadere che ad un certo punto P1 richieda un'operazione di I/O, liberi la CPU che viene utilizzata per l'esecuzione di P2, a ciò non è garantito. Se P1 fosse un processo *CPU bound* (si limita a fare i calcoli, senza bisogno di fare interazioni I/O), una volta che prende possesso della CPU non la lascia. In questo caso *dove intervenire il Sistema Operativo* per garantire che anche P2 possa essere eseguito.

Nel caso di un sistema **MULTIPROGRAMMATO**, risulta comune che il processore esegua sequenze di operazioni appartenenti a processi diversi (che quindi determinano *multiprocessing*). Quindi, nel Sistema Operativo sono presenti **contemporaneamente** più processi attivi di cui:

- uno solo è in esecuzione
- gli altri sono:
 - **PRONTI** → in attesa del processore
 - **IN ATTESA** di eventi (ad esempio, completamento di operazioni di I/O)

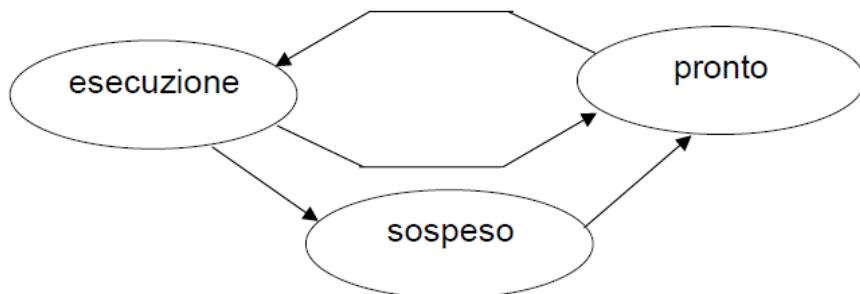
Evoluzione Temporale di Due Processi Concorrenti



Si fa riferimento agli istanti temporali rappresentanti l'evoluzione dei due processi concorrenti. Nell'istante t_5 entrambi i processi sono in blocco e la **CPU** risulta **libera**. Queste situazioni sono possibili, ma si cerca di limitarle in modo di dare alla CPU sempre un compito da eseguire.

2.1.3 Stati di un Processo

- **IN ESECUZIONE (RUNNING)** → sta usando la CPU
- **SOSPESO (SLEEPING)** → in attesa di un evento
- **PRONTO (READY)** → pronto per l'esecuzione



Nel caso in cui un processo fosse di tipo **CPU bound** (non abbandona la CPU autonomamente) può essere gestito da un Sistema Operativo **TIME-SHARING** in modo da forzare la transizione di stato da *esecuzione* a *pronto* andando a sottrarre la CPU, così che questa non venga monopolizzata da un unico processo. I sistemi *time-sharing* **assegnano un quanto di tempo specifico** per l'esecuzione di una parte del processo, una volta esaurito questo quanto *riportano il processo in stato di pronto*, e solo a un giro successivo questi possono tornare ad essere in esecuzione.

2.1.4 Descrittore di un Processo

Siccome quando un processo è in esecuzione utilizza la CPU (*i registri macchina della CPU*), bisogna fare in modo che *quando questo non occupa più la CPU, tutte le informazioni relative allo stato di esecuzione vengano salvate*. Se non si prende questo accorgimento, il processo *non può riprendere la sua esecuzione* dalla situazione in cui era arrivato.

A questo scopo *ogni processo* possiede un **DESCRITTORE di PROCESSO** (*Process Control Block, PCB*) che *mantiene tutte queste informazioni* (e altre) *quando il processo non è in esecuzione* (pronto o sospeso). Questi descrittori sono solitamente *in code*, pertanto si fa riferimento alle *code di processi pronti* e alle *code di processi sospesi*. Di solito presentano almeno un *riferimento al prossimo processo in coda*, potendone presentare uno *anche per il processo precedente*.

2.2 Relazione tra Processi Concorrenti

2.2.1 Processi Concorrenti Disgiunti

PROCESSO INDIPENDENTE → un processo che *non può influenzare o essere influenzato da altri processi*.

Proprietà

- il suo stato *non è condiviso* da altri processi
- la sua esecuzione è *deterministica*: il *risultato della esecuzione dipende solo dai dati di ingresso*
- la sua esecuzione è *riproducibile*: il *risultato della esecuzione è sempre lo stesso* a parità dei dati in ingresso
- la sua esecuzione può essere *bloccata e fatta ripartire* senza provocare danni

2.2.2 Processi Concorrenti Interagenti

PROCESSO INTERAGENTE → un processo che può influenzare o essere influenzato da altri processi.

Proprietà

- il suo *stato* è *condiviso* da altri processi
- la sua esecuzione è *non deterministica*: il *risultato della esecuzione* dipende dalla sequenza di esecuzione relativa e *non è predibile*
- la sua esecuzione è *non riproducibile*: il *risultato della esecuzione non è sempre lo stesso* a parità dei dati in ingresso
- la sua *esecuzione* non può essere *bloccata e fatta ripartire* senza provocare danni

Interazione tra Processi Interagenti

Si distinguono più tipi di interazioni:

- **COMPETIZIONE** → *sincronizzazione* indiretta o implicita
- **COOPERAZIONE** → *sincronizzazione* diretta o esplicita

A queste se ne aggiunge un terzo tipo, che però risulta **indesiderata**:

- **INTERFERENZA**

Competizione (indiretta)

I processi *competono per l'uso di risorse comuni*. Un problema inerentemente non sequenziale può essere risolto da un insieme di processi concorrenti che interagiscono in modo competitivo.

Il problema di base della competizione è quello della **mutua esclusione**, cioè la *competizione per l'uso di risorse comuni che non possono essere usate contemporaneamente*. A questo problema tuttavia possono aggiungersene altri. Questo tipo di interazione introduce dei **vincoli di sincronizzazione** tra gli eventi dei processi concorrenti. Questi vincoli servono per garantire una corretta interazione.

Anche i processi indipendenti competono per l'uso esclusivo di risorse comuni.

Cooperazione (diretta)

Un problema inerentemente non sequenziale può essere risolto da un insieme di processi concorrenti che interagiscono in modo cooperativo.

Con questo tipo di interazione è previsto uno **SCAMBIO DI INFORMAZIONI**. Questo scambio di informazioni può semplicemente essere:

- *invio e la ricezione di un segnale* (senza trasferimento di dati)
- *invio e ricezione di messaggi* (con trasferimento di dati) → in questo caso la *sincronizzazione che è necessaria* (vincoli di sincronizzazione) fra i processi spesso viene detta **COMUNICAZIONE**. È possibile pertanto considerare la comunicazione come un *caso particolare di sincronizzazione*.

Anche con questo tipo di interazione, i processi concorrenti interagenti che compongono un processo non sequenziale introducono dei vincoli di **SINCRONIZZAZIONE** tra gli eventi dei processi concorrenti.

Interferenza

Questo tipo di interazione è *provocata da errori di programmazione*, in particolare può essere provocata da:

- inserimento nel programma di interazioni tra processi non richieste dalla natura del problema
- erronee soluzioni a problemi di interazione (cooperazione e competizione) necessarie per il corretto funzionamento del programma

Si tratta di un'interazione *non prevedibile e non desiderata* che dipende dalla **velocità relativa** tra i processi. Ciò vuol dire che una certa esecuzione in un lasso di tempo potrebbe non portare a problemi di interferenza, ma a parità dei dati di ingresso un'ulteriore esecuzione potrebbe portare al verificarsi di problemi di interferenza. Questi **errori** sono **dipendenti dal tempo**, e prendono anche il nome di *corse critiche* (race condition).

2.2.3 Sincronizzazione

Per far modo che non si verifichino errori dipendenti dal tempo, viene introdotto un **vincolo di sincronizzazione**. La sincronizzazione rappresenta un **vincolo sull'ordine con cui sono eseguite le operazioni** sui processi. Questi vincoli sono diversi a seconda dell'interazione:

- Nel caso di *INTERAZIONE INDIRETTA*, il vincolo deve rendere *impossibile* l'esecuzione contemporanea delle operazioni con le quali i *processi accedono a risorse comuni* (caso della mutua esclusione).
- Nel caso di *INTERAZIONE DIRETTA*, il vincolo deve *imporre* che le operazioni di un processo *non abbiano inizio prima o dopo determinate operazioni di altri processi*.

In ogni caso sono necessari **MECCANISMI DI SINCRONIZZAZIONE** → non usare sincronizzazione o sbagliarla implica introdurre degli *ERRORI DIPENDENTI DAL TEMPO*.

Esempio:

Si riprende in considerazione l'esempio di elaborazione di dati su un file sequenziale (in particolare, il *grafo ad ordinamento parziale*).

In questo caso il processo E (elaboratore), non può svolgere le operazioni di elaborazione sul secondo dato prima che il processo L (lettore) non abbia letto il secondo dato e glielo abbia comunicato. La *freccia* tra i due eventi rappresenta un *vincolo di sincronizzazione diretta*. Il processo lettore dovrà avere un modo per comunicare il secondo dato che è stato letto al processo di elaborazione. Così come il processo di elaborazione dovrà fornirlo al processo scrittore (mediante una comunicazione). Le frecce verticali rappresentano la *normale sequenzialità delle operazioni del singolo processo*. Un qualsiasi processo L (così come gli altri) non può leggere il dato *i* se prima non ha letto il dato *i-1* e così via. Fra i vari processi allo stesso modo esistono delle relazioni di sincronizzazione (comunicazione) per cui il processo lettore, fino a che non ha comunicato il dato letto al processo di elaborazione non può passare a leggere il successivo.

Strumenti di Sincronizzazione

Due diversi modelli logici di riferimento:

- MODELLO AD AMBIENTE GLOBALE → la prima configurazione a livello hardware era di avere una sola CPU con una memoria comune.
 - il primo definito a livello storico
 - detto anche Modello a memoria comune

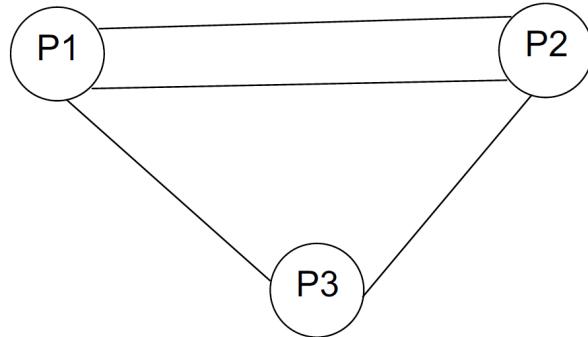
- MODELLO AD AMBIENTE LOCALE → è il modello dei processi utilizzato in UNIX (e quindi anche in Linux), scelto nonostante avessero anch'essi una sola CPU e una sola memoria.
 - il secondo definito a livello storico
 - detto anche Modello a scambio di messaggi
 - maggiore **PROTEZIONE** e **CORRETTEZZA** (ogni processo ha il proprio spazio di indirizzamento che non è condiviso con gli altri processi)

Modello ad Ambiente Locale

Ogni **applicazione** viene strutturata come un **insieme di processi**, ciascuno operante in un *ambiente locale non accessibile direttamente da nessun altro processo*. In questo modello si può avere **solo un tipo di interazione** fra processi: la **COOPERAZIONE**. Ogni forma di interazione tra processi (sincronizzazione) avviene tramite *scambio di messaggi* o *invio di segnali*.

Questo modello è tipico dei Sistemi UNIX (in particolare LINUX).

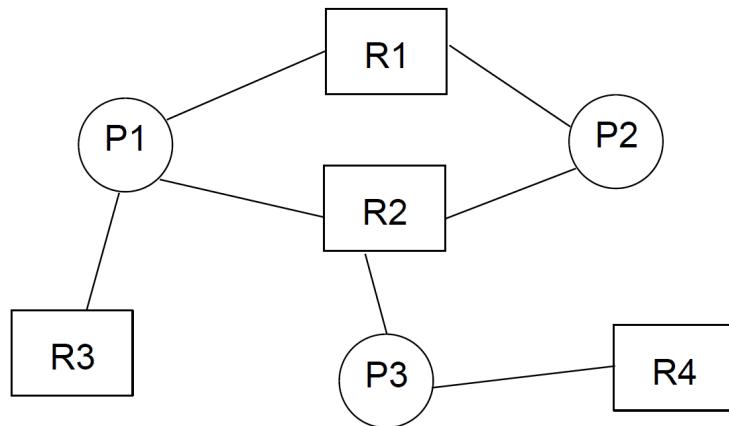
A livello logico l'applicazione è *composta esclusivamente da processi*:



Modello ad Ambiente Globale

Ciascuna applicazione viene strutturata come un **insieme di componenti**, suddivisa in due insiemi disgiunti:

- PROCESSI (componenti attivi)
- RISORSE (componenti passivi)



RISORSA → qualunque oggetto, fisico (es. stampante) o logico (es. memoria) di cui un processo necessita per portare a termine il suo compito. Una risorsa è generalmente utilizzata da più processi. In questo tipo di modelli le risorse sono raggruppate in classi.

Una **classe di risorse** identifica l'*insieme di tutte e sole le operazioni che un processo può eseguire per operare su risorse di quella classe*.

Esempio:

Si ha la classe di risorse denominata "Stampante". Se a livello architetturale si hanno a disposizione 3 stampanti, le operazioni che si possono fare su *una delle specifiche istanze* della classe Stampante, sono le stesse, ma si hanno più istanze appartenenti alla classe.

Nel modello ad ambiente globale si possono avere **due tipi di interazione** fra processi: **COMPETIZIONE** e **COOPERAZIONE**. Gli STRUMENTI DI SINCRONIZZAZIONE sono: semafori (meccanismi semplici, a livello di S.O), monitor (più sofisticati, a livello di programmazione), ecc...

I vari problemi di sincronizzazione nell'ambito dei processi concorrenti interagenti possono essere sia *problemi di competizione* che *problemi di cooperazione*.

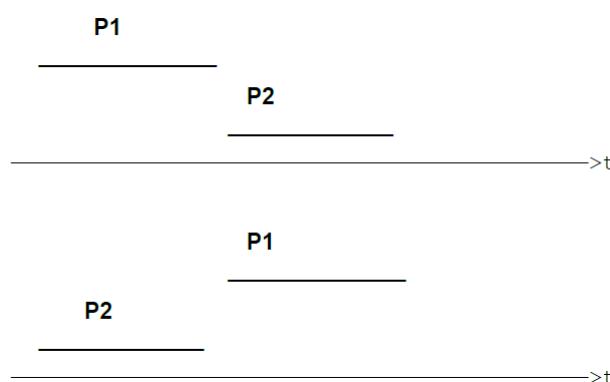
Problemi di Competizione

Mutua Esclusione

Si ha necessità di mutua esclusione quando **non più di un processo alla volta** può accedere ad una risorsa comune → problema base della competizione fra processi (modello ad ambiente globale).

Esempio di Risorsa Comune:

Insieme di variabili contenute in memoria centrale (comune).



Supponendo di avere un solo processore, definiamo i segmenti *P1* e *P2* l'insieme delle istruzioni che vanno a modificare le variabili comuni da parte dei rispettivi processi *P1* e *P2*. La **regola** di mutua esclusione impone che le **operazioni** con le quali i processi accedono alle variabili comuni **non si sovrappongano nel tempo**. In figura vengono riportati i due scenari dove nel primo opera prima *P1*, mentre nel secondo opera prima *P2*. Non viene imposto **nessun vincolo sull'ordine** on il quale le operazioni sulle variabili comuni sono eseguite.

Esempio di Mutua Esclusione (1):

P1 e P2 accedono a due variabili comuni:

- un contatore (*cont*) che deve incrementare ognqualvolta si effettua una determinata azione
- un identificatore (*id*) che deve tenere traccia dell'ultimo processo che ha effettuato l'azione

Al completamento dell'esecuzione dei processi, *cont* deve contenere un valore pari al numero complessivo delle azioni effettuate dai due processi e *id* deve contenere l'identificatore dell'ultimo processo che ha effettuato l'azione. Si consideri che un processo può eseguire un frammento di codice anche più volte (inserendolo ad esempio in un ciclo for).

P1

```
...
<azione>
cont = cont + 1;
id = getpid();
...
```

P2

```
...
<azione>
cont = cont + 1;
id = getpid();
...
```

Con la sintassi si fa riferimento a una parte di codice eseguita da ciascun processo.

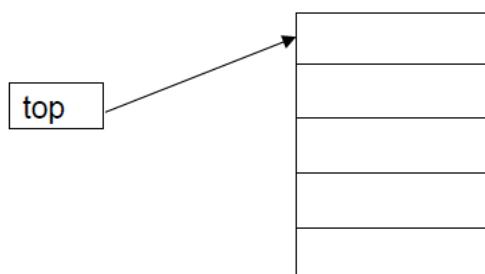
Possibile sequenza di esecuzione:

```
t0:          (P1)
t1:          (P2)
t2: cont = cont + 1;  (P2)
t3: id = P2;         (P2)
t4: cont = cont + 1;  (P1)
t5: id = P1;         (P1)
```

A livello pratico, il contatore è stato incrementato di 2 *unità*, e il codice è stato eseguito 2 *volte*, prima da P1, poi da P2. Per quanto riguarda l'ID, il valore non è corretto in quanto risulta che l'ultimo processo a eseguire l'azione è stato P1, ma è chiaramente visibile dalla sequenza che in realtà l'ultimo ad averlo eseguito è stato P2.

Esempio di Mutua Esclusione (2):

Due processi (P1 e P2) hanno accesso ad una struttura dati organizzata a pila (STACK) per inserire e prelevare informazioni (sempre dalla prima posizione, il top dello stack).



I due processi fanno uso delle due procedure:

- INSERIMENTO
- PRELIEVO

INSERIMENTO (y)

```
...
top = top + 1;
stack[top] = y;
...
```

PRELIEVO (x)

```
...
x = stack[top];
top = top - 1;
...
```

L'esecuzione contemporanea delle due procedure può portare ad un uso scorretto della risorsa STACK.

Possibile sequenza di esecuzione: (peggiore caso possibile, caso di *corsa critica*)

```
t0: top = top + 1  (P1)
t1: x = stack[top]  (P2)
t2: top = top - 1  (P2)
t3: stack[top] = y  (P1)
```

Quando nell'istante *t1* si fa un'operazione di *prelievo* puntando al top, il top risulta ancora vuoto, in quanto alla corrispettiva posizione dell'array non è ancora stata assegnata il valore *y*, pertanto la precedente operazione di inserimento non risulta completata.

Sezione Critica

Una sezione critica è la **sequenza di operazioni** con le quali un processo accede e modifica un insieme di variabili comuni.

Ad un insieme di variabili comuni possono essere associate:

- UNA SOLA sezione critica (usata da tutti i processi)
- PIÙ sezioni critiche (classe di sezioni critiche) → nello scenario più completo la classe potrebbe degenerare e contenere una sola sezione critica.

Per capire meglio questi termini, riprendiamo gli esempi precedenti:

- Nel primo esempio, i processi usavano variabili comuni (cont e id) tramite una sola sezione critica (`cont = cont+1; id = getpid();`)
- Nel secondo esempio, i processi usavano variabili comuni (top e stack) tramite due sezioni critiche, che quindi definivano una classe di sezioni critiche (INSERIMENTO e PRELIEVO).

La REGOLA di MUTUA ESCLUSIONE stabilisce che: **sezioni critiche appartenenti alla stessa classe devono escludersi mutuamente nel tempo**. In altre parole, *una sola sezione critica di una classe può essere in esecuzione ad ogni istante di tempo*.

2.2.4 Soluzioni

Come soluzione al problema della mutua esclusione si devono utilizzare degli **STRUMENTI DI SINCRONIZZAZIONE**. La sezione critica deve essere preceduta da una parte di **PROLOGO** e seguita da una parte di **EPILOGO**.

- PROLOGO → deve assicurarsi che il processo sia l'unico che sta cercando di eseguire la porzione critica. Deve verificare la disponibilità della sezione critica, e in caso affermativo, acquisirla.
- EPILOGO → fa in modo che sia chiaro al sistema operativo che il processo non è più in esecuzione sulla sezione critica. Si deve occupare del rilascio della sezione critica (in modo che altri processi possano eseguirla)

...

PROLOGO

SEZIONE CRITICA

EPILOGO

...

Nel seguito considereremo (se non diversamente specificato) il CASO PARTICOLARE di avere due processi P1 e P2.

Requisiti per una Soluzione Accettabile

1. Mutua esclusione dei processi che eseguono le sezioni critiche
2. Indipendenza dei processi che eseguono le sezioni critiche (l'esecuzione di un processo non deve essere un prerequisito per l'esecuzione di un altro).
3. Assenza di condizioni di stallo (deadlock a livello di strumento di sincronizzazione) per i processi che eseguono le sezioni critiche

DEADLOCK → i processi sono bloccati nell'attesa di verificarsi di situazioni che non si possono verificare

4. Assenza di attese attive per i processi che eseguono le sezioni critiche (conviene mettere in attesa un processo se questo non può eseguire la sezione critica)
5. Assenza di *starvation* per i processi che eseguono le sezioni critiche (i processi messi in attesa devono essere riattivati non appena possibile).
- STARVATION** → dimenticarsi di un processo nella coda dei processi pronti
6. Le sezioni critiche eseguite con interruzioni abilitate

Osservazione: Si suppone che ogni processo sia eseguito ad una velocità diversa da 0 → non si può fare alcuna ipotesi sulla velocità relativa dei processi. Le sezioni critiche non hanno quindi una velocità istantanea di esecuzione.

I requisiti **1, 2 e 3** sono di **CARATTERE LOGICO** e l'implementazione del meccanismo di mutua esclusione è **corretto** solo se presenta questi tre requisiti. Il requisito **5** è di **ordine realizzativo** ed è ancora legato alla **CORRETTEZZA** dell'implementazione.

I requisiti **4 e 6** sono di ordine realizzativo e non riguardano la correttezza del meccanismo, ma l'**EFFICIENZA** della sua implementazione.

Il problema della mutua esclusione può, in prima approssimazione, essere risolto rispondendo solo ai REQUISITI 1, 2, 3, e 5, trascurando i requisiti 4 e 6 → **IPOTESI DI SEZIONI CRITICHE BREVI**

Sezioni Critiche Sufficientemente Brevi

Posso avere una soluzione al problema della mutua esclusione introducendo delle *attese attive*. Con questa soluzione il sistema non è efficiente, in quanto si sprecano dei cicli di clock per nulla, ma potrebbe comunque rappresentare una soluzione per sezioni critiche sufficientemente brevi. In alternativa, posso decidere di rappresentare lo strumento di sincronizzazione attraverso la *disabilitazione delle interruzioni*, perché ciò, nel caso di sistemi monoprocesso, realizza immediatamente la mutua esclusione. Disabilitando le interruzioni non può né intervenire un processo perché lo scheduler ha scoperto che è scaduto il quanto di tempo (che rappresenta un'interruzione) né si può accorgere che è intervenuto un processo a maggiore priorità, in quanto questo nuovo evento (l'avere un nuovo processo nella coda dei processi pronti) non viene evidenziato.

Sia {A, B, C, ...} una classe di sezioni critiche "sufficientemente" brevi

Soluzione in Caso Monoprocesso

PROLOGO → DISABILITAZIONE INTERRUZIONI

violazione requisito n. 6

EPILOGO → ABILITAZIONE INTERRUZIONI

Soluzione in Caso Multiprocesso

USO DI OPERAZIONI INDIVISIBILI (**PRIMITIVE**) chiamate

PROLOGO → LOCK (X)

violazione requisito n. 4

EPILOGO → UNLOCK (X)

```

typedef enum {false, true} Boolean;
void LOCK (Boolean X) {
    while (X);           // fino a che X è true si rimane in attesa attiva
    X = true
}
void UNLOCK (Boolean X) {
    X = false;
}

```

X è un **indicatore** associato alla classe di sezioni critiche (un indicatore per ciascuna classe):

- **X == false** nessuna sezione critica in esecuzione → il processo che trova X a false può metterlo a true e cominciare ad eseguire
- **X == true** una sezione critica in esecuzione → un processo sta eseguendo una funzione critica quindi ogni altro processo che tenta di fare l'epilogo per poi passare ad eseguire la sezione critica viene bloccato

INDIVISIBILITÀ → nessun altro processo può andare ad eseguire la LOCK se questa è in esecuzione sulla stessa variabile X da parte di un altro processo.

Per quanto riguarda l'*UNLOCK* in un singolo ciclo di clock viene posto *X = false*, nello stesso ciclo di clock nessun altro può avere accesso alla memoria centrale e andare a interrompere l'assegnamento. La *LOCK* è più problematica, in quanto presenta una verifica del valore di X e dall'assegnazione *X = true* quando questa ha valore false.

L'**INDIVISIBILITÀ** della *LOCK* (in particolare, per le due funzioni di *LOCK* e *UNLOCK*) viene *garantita* da *istruzioni hardware* :

- TEST-AND-SET
- EXCHANGE

LOCK con TEST-AND-SET

Questa istruzione HARDWARE, **in un solo ciclo di clock**, verifica e assegna valore alla variabile su cui agisce. A livello logico (astratto):

```

typedef enum {false, true} Boolean;
Boolean TEST_AND_SET (Boolean &parametro) {
    Boolean valore = parametro;
    parametro = true;
    return valore;
}

```

L'operazione *LOCK* diviene:

```

void LOCK (Boolean X) {
    while (TEST_AND_SET(X));
}

```

La *TEST-AND-SET* verifica il valore e lo setta contestualmente a *true*.

- se *X == false* torna il valore precedente ma avendo contestualmente cambiato il valore di X a *true*, facendo terminare il ciclo *while*, terminando a sua volta la *LOCK* e quindi l'*EPILOGO*. Terminato l'*epilogo*, il processo può andare ad eseguire la *sezione critica* della classe protetta dal booleano X. Una volta eseguita questa parte dovrà essere eseguita l'*UNLOCK* affinché riporti il valore a *false*.

- se $X == \text{true}$ (è già presente un processo che sta eseguendo la sezione critica della classe X) la funzione setta nuovamente X a true e ritorna true, andando a bloccare in un ciclo di attesa attiva il processo mediante un `while(true)`. Questo rimarrà tale fino a quando il processo che sta eseguendo la sezione critica della classe associata a X andrà ad eseguire l'UNLOCK, ponendo X = false.

Non c'è nessuna garanzia che il primo processo che tenta di accedere alla sezione critica della classe occupata sia il primo ad accedere alla sezione critica.

LOCK con EXCHANGE

Questa istruzione hardware, in un solo ciclo di clock, scambia il valore di due variabili.

```
typedef enum {false, true} Boolean;
void EXCHANGE (Boolean &a, Boolean &b) {
    Boolean temp = a;
    a = b;
    b = temp;
}
```

La funzione *EXCHANGE* si limita a scambiare il valore di due variabili passate per riferimento.

La funzione LOCK diviene:

```
void LOCK (Boolean X) {
    Boolean priv;
    priv = true;
    do
        EXCHANGE(X, priv);
    while (priv == true);
}
```

La funzione LOCK deve definire una *variabile privata*, ovvero allocata sullo *stack della LOCK*, ovvero sullo *stack privato del processo*, non è una variabile condivisa come la X. In questo modo ciascun processo che esegue la LOCK ha una propria istanza della variabile *priv*. Nel ciclo di attesa attivo si va a scambiare il valore di *priv* (inizializzato a true) con X:

- se $X == \text{false}$, X assume il valore true, mentre *priv* assume il valore di false. In questo caso la LOCK viene definita *PASSANTE*. L'EPILOGO ha successo e quindi il processo può eseguire la sezione critica associata alla classe di sezione critica X e contestualmente la X viene messa a false, quindi vengono bloccati eventuali tentativi di accesso con la LOCK.
- se $X == \text{true}$, lo scambio è inutile ma viene comunque fatto, ponendo X a true e *priv* a true. Poiché *priv* è ora uguale a true si rimane nel ciclo di attesa attiva.

Semaforo

Un semaforo S rappresenta una *istanza* di un *tipo di dato astratto* (*Semaphore*).

TIPO DI DATO ASTRATTO → simile al concetto di *classe* per i linguaggi di programmazione a oggetti (come ad esempio Java). In esso è però assente il concetto di ereditarietà.

Un Semaforo presenta la *rappresentazione interna di ogni sua istanza* (oggetto, in termini di linguaggi di programmazione) mediante un *valore intero* (valore del semaforo) e una *coda di descrittori dei processi*, i quali saranno accodati all'interno dei semafori in attesa delle condizioni necessarie per poter proseguire nella loro esecuzione. Un Semaforo *non implica un'attesa attiva*. Il processo all'interno della sua coda si sospende qualora le condizioni per la sua esecuzione non

risultano soddisfatte. Un Semaforo presenta inoltre due operazioni indivisibili chiamate *wait(S)* e *signal(S)*, ciascuna che prende come parametro una certa istanza di semaforo.

La sua struttura si presenta dunque in questo modo:

- DATO
 - valore intero
 - coda descrittori
- OPERAZIONI (INDIVISIBILI)
 - wait (S)
 - signal (S)

Le due funzioni possono anche essere chiamate diversamente:

- WAIT (S) → P(S) (si effettua una verifica che presenta un *if*)
- SIGNAL (S) → V(S)

Ad un Semaforo S è associato un valore intero non negativo *Sv*, con valore iniziale:

$$S_0 \geq 0$$

La *coda di attesa* associata ad un generico Semaforo S verrà riconosciuta con il simbolo Q_S dove S è l'istanza specifica del semaforo. Essendo le due operazioni precedentemente definite *INDIVISIBILI*, mentre un processo esegue un'operazione non deve essere possibile per nessun altro processo eseguire nessun'altra operazione sullo stesso semaforo.

Operazioni su un Semaforo

WAIT(S)

```
void WAIT(Semaphore S) {
    if (Sv == 0) {
        < il processo viene SOSPESO
        e il suo descrittore viene inserito in QS >
    }
    else Sv = Sv - 1;
}
```

- Se $Sv == 0$ (semaforo rosso), la WAIT viene detta SOSPENSIVA, in quanto non ha avuto successo e il processo è stato sospeso. Non può essere eseguita nessun'altra istruzione seguente all'invocazione della WAIT perché il processo risulta bloccato.
- Se $Sv > 0$ (condizione necessaria, un semaforo non può avere valore negativo), allora il valore viene decrementato e la WAIT ha successo. Il processo in questo modo può proseguire con l'istruzione successiva.

SIGNAL(S)

```
void SIGNAL(Semaphore S) {
    if (<esiste un processo P nella coda QS?>) {
        < il suo descrittore viene tolto da QS e
        lo stato di P modificato in PRONTO >
    }
    else Sv = Sv + 1;
}
```

- Se Q_S presenta processi in coda viene estratto il primo processo nella coda (FIFO) e inserito nella coda dei processi pronti. La coda Q_S rappresenta la coda dei processi sospesi, mentre la

Signal cambia lo stato di questi processi, togliendoli della coda e *segnandoli come pronti*. Questa operazione *non può mettere in esecuzione*, poiché in esecuzione c'è chi sta eseguendo la Signal.

- Se Q_S non presenta processi in coda il valore del semaforo viene incrementato.

Invariante dei Semafori

$$Sv = S_0 + ns(S) - nw(S)$$

- **Sv** → valore del semaforo
- **S₀** → valore iniziale del semaforo
- **ns(S)** → numero di volte che è stata eseguita la signal con nessun processo in coda
- **nw(S)** → numero di volte che è stata eseguita la wait con successo

Dato che

$$Sv \geq 0$$

allora

$$nw(S) \leq ns(S) + S_0$$

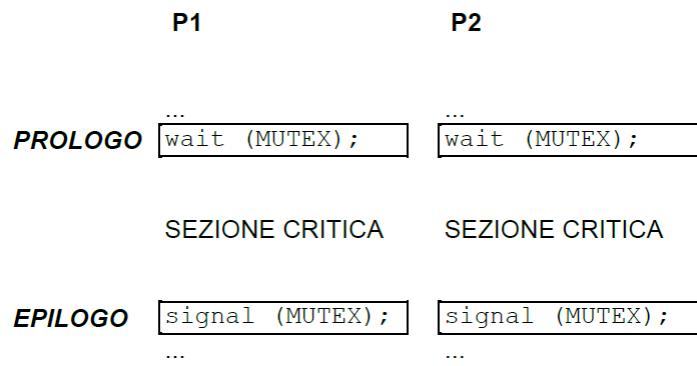
Questa relazione è **INVARIANTE** chiamata anche **INVARIANTE DEL SEMAFORO** cioè è *sempre vera* qualunque sia il numero di primitive (WAIT e SIGNAL) eseguite su un qualunque semaforo.

Utilizzo di un Semaforo per Garantire la Mutua Esclusione

I Semafori rappresentano un **meccanismo** per sospendere e riattivare i processi. Questo meccanismo può essere usato per realizzare ad esempio una **politica** di mutua esclusione.

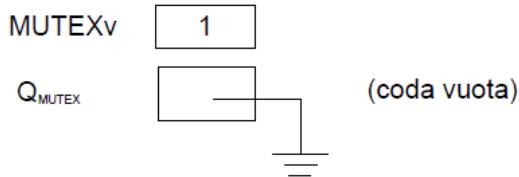
```
Semaphore MUTEX;
/* valore iniziale MUTEX_0 = 1 */
```

In questo ambito i semafori vengono normalmente chiamati *MUTEX (mutual exclusion)*. Questi Semafori devono avere sempre come valore iniziale 1.



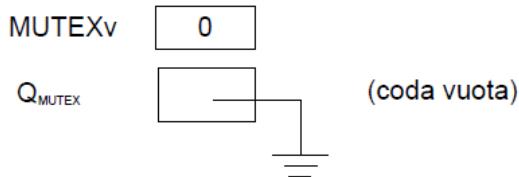
Esempio: Possibile Sequenza di Operazioni di Sincronizzazione

Situazione iniziale



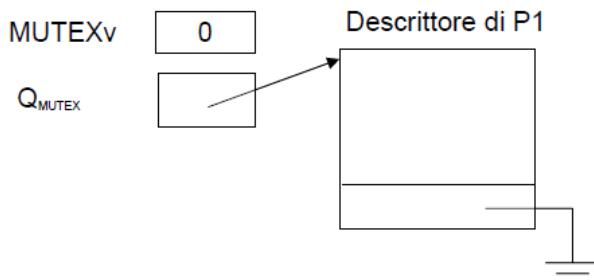
Arriva prima il processo P2

- ➔ wait su **MUTEX** (valore == 1)
 - if ($\text{MUTEXv} == 0$) ...
 - else $\text{MUTEXv} = \text{MUTEXv} - 1 = 0$
- ➔ P2 entra nella sezione critica



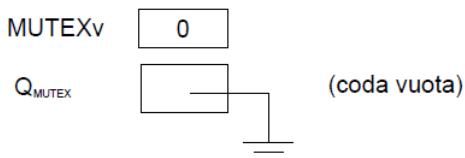
Arriva poi il processo P1

- ➔ wait su **MUTEX** (valore == 0)
 - if ($\text{MUTEXv} == 0$)
 - il descrittore di P1 in coda
- ➔ P1 viene **SOSPESO**



Il processo P2 termina la sezione critica

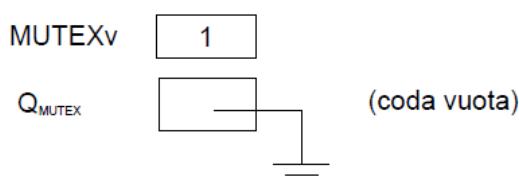
- ➔ signal su **MUTEX** (valore == 0)
 - if (< esiste un processo in coda?>)
 - < il descrittore di P1 viene tolto dalla coda
- ➔ P1 diviene PRONTO >



Nota: Il valore del semaforo è rimasto a 0 perché P1 è dentro la sezione critica. Se arrivasse un processo P3, o lo stesso P2 fosse in un ciclo per tornare dentro la sezione critica questo processo deve essere bloccato *fuori dalla sezione critica*.

Il processo P1 termina la sezione critica

- ➔ signal su **MUTEX** (valore == 0)
 - if (< esiste un processo in coda?>) ...
 - else $\text{MUTEXv} = \text{MUTEXv} + 1 = 1$



Soddisfacimento dei Requisiti

1. Un solo processo alla volta si può trovare nella sezione critica:

$$n = nw(MUTEX) - ns(MUTEX)$$

numero di processi dentro la sezione critica

NOTA: $n \geq 0$

L'invariante

$$nw(MUTEX) \leq ns(MUTEX) + MUTEX_0$$

diventa

$$nw(MUTEX) \leq ns(MUTEX) + 1$$

$$n = nw(MUTEX) - ns(MUTEX) \leq 1$$

Quindi

$$0 \leq n \leq 1$$

2. Un processo può bloccarsi SOLO se la sezione critica è occupata. infatti, se un processo si blocca su un semaforo è perché:

$$MUTEXv == 0$$

Quindi dalla relazione invariante dei semafori:

$$MUTEXv = ns(MUTEX) - nw(MUTEX) + MUTEX0$$

diventa

$$nw(MUTEX) = ns(MUTEX) + 1$$

e quindi

$$n = nw(MUTEX) - ns(MUTEX) = 1$$

3. Assenza di condizioni di stallo → il meccanismo del semaforo non le presenta per lo stesso motivo appena dimostrato.
4. Un semaforo non presenta attese attive.
5. Non c'è *starvation* in quanto perché la coda è gestita *FIFO*, in questo modo viene garantito che il primo processo sospeso è il primo ad essere riattivato.
6. Non si è parlato di disabilitazione delle interruzioni

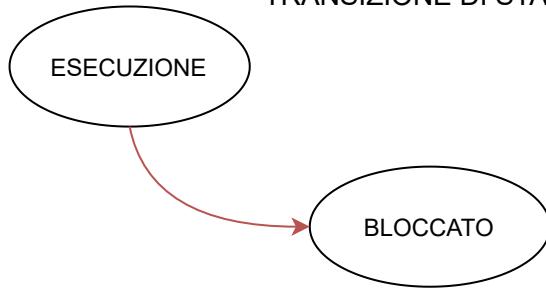
Osservazioni su WAIT e SIGNAL

1. L'esecuzione della wait può essere un'azione sospensiva per il processo (P_X) che la esegue. La sospensione *dipende dal valore del semaforo*: se il valore è 0 (rosso), allora il processo viene sospeso.

P_X esegue una *WAIT SOSPENSIVA*
su un semaforo

1

TRANSIZIONE DI STATO di P_X



Il descrittore di P_X viene inserito nella coda associata al semaforo

- eventuale (se il valore del semaforo è 0) modifica dello *stato del processo* da *ESECUZIONE* → *BLOCCATO*.

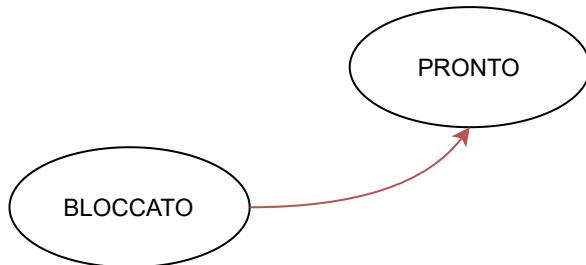
- evita l'*ATTESA ATTIVA*

2. L'esecuzione della signal NON comporta concettualmente nessuna modifica nello stato del processo (P_Y) che la esegue.

P_Y esegue una *SIGNAL RISVEGLIANTE* sul semaforo su cui è sospeso P_X (supponiamo che P_X sia il primo processo della coda)

2

TRANSIZIONE DI STATO di P_X



Quando l'algoritmo scheduling decide che il processo può andare in esecuzione lo porterà nuovamente in esecuzione.

- eventuale MODIFICA DELLO STATO DI UN PROCESSO in coda da BLOCCATO a PRONTO

3) La scelta del processo sospeso da riattivare è fatta tramite una politica **FIFO** → evita la **STARVATION**.

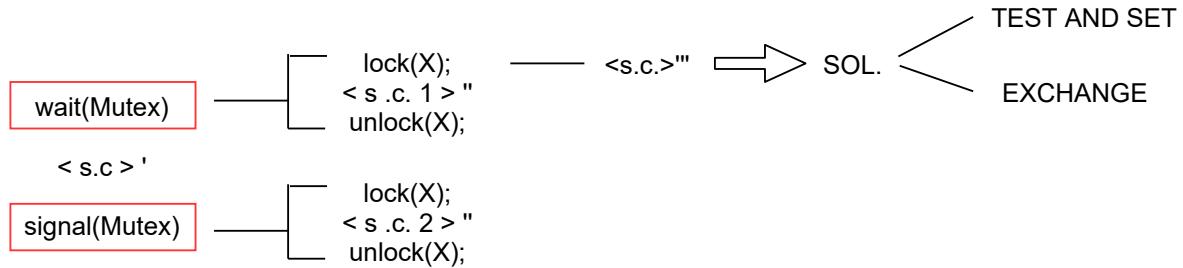
I semafori *non* sono meccanismi utili *esclusivamente per il problema della mutua esclusione*, ma sono dei meccanismi più generali su cui si riesce a implementare delle *politiche di sincronizzazione*, in particolare per risolvere problemi di cooperazione e di competizione.

Se il valore del contatore associato ad un semaforo ha il valore vincolato ad essere sempre 0 o 1 (come nel caso dei semafori usati solo per la mutua esclusione) si usa più propriamente il termine **SEMAFORO BINARIO o MUTEX**, altrimenti si parla di **SEMAFORO CONTATORE o GENERALIZZATO**.

Indivisibilità della WAIT e della SIGNAL

L'implementazione dello *pseudo-codice* della WAIT e della SIGNAL comporta che le due operazioni vadano ad agire sul valore del semaforo (S_Y), sia in lettura che in scrittura e sulla coda associata al semaforo (Q_S) → occorre garantire l'indivisibilità delle operazioni (primitive) sui semafori.

Concettualmente è come avere un problema di mutua esclusione per una classe di sezioni critiche di primo livello (■).



Questo problema si risolve introducendo:

PROLOGO → *wait(Mutex)*

EPILOGO → *signal(Mutex)*

Così facendo si ha la garanzia che la sezione critica venga eseguita in maniera mutualmente esclusiva. In questo modo si sposta il problema a dover realizzare una mutua esclusione di secondo livello (indicata in figura con) perché i codici della WAIT e della SIGNAL presentano a loro volta delle sezioni critiche.

Perché di II Livello? → I codici delle due operazioni vanno a fare accedere un *insieme di processi maggiore di 1* a una struttura dati condivisa (corrispondente al semaforo). I codici indicati con `<s.c. 1>` e `<s.c. 2>` sono codici appartenenti alla classe di sezione critica di secondo livello.

Perché di III Livello? → Per implementare la LOCK è come se andassimo ad introdurre un *problema di sezione critica di terzo livello* (). Con la UNLOCK X viene settato a false. Con la LOCK è necessario verificare il valore di X: nel caso in cui sia *false* bisogna metterlo *true*, mentre nel caso in cui questo sia *true* bisogna continuare in un *ciclo di attesa attiva* verificando quando questo valore cambia. Questo livello è l'ultimo perché in seguito si utilizza una soluzione con *TEST-AND-SET* o con *EXCHANGE* (soluzioni a livello hardware).

INDIVISIBILITÀ' → mentre un processo sta eseguendo la WAIT su un semaforo, un altro processo non può eseguire né la WAIT né la SIGNAL sullo stesso semaforo.

Prima Soluzione

Disabilitazione delle interruzioni → Rappresenta una *soluzione parziale* perché *non vale per i multiprocessori*.

Seconda Soluzione

Uso di LOCK e UNLOCK implementate con TEST-AND-SET o EXCHANGE → Utilizzate esclusivamente per *soluzioni critiche sufficientemente brevi*, per il fenomeno di *attesa attiva* all'interno della LOCK. Il codice della WAIT, così come quello della SIGNAL, essendo dei codici scritti a livello di programmazione *rispettano i requisiti di essere sezioni critiche sufficientemente breve*.

Possiamo dunque proteggere con un PROLOGO di LOCK e un EPILOGO di UNLOCK su una certa variabile booleana X il codice della WAIT e quello della SIGNAL.

```

void WAIT (Semaphore s) {
    LOCK (x);
    < codice della wait >;
    UNLOCK (x); // ATTENZIONE
}

void SIGNAL (Semaphore s) {
    LOCK (x);
    < codice della signal >;
    UNLOCK (x);
}

```

Se si hanno più classi di sezioni critiche di primo livello (livello utente-programmatore) si dovranno avere tanti semafori di mutua esclusione quanti sono le classi che si dovranno proteggere. Il numero di parametri X che servono sono tanti quanti i semafori.

Il codice della UNLOCK va eseguito sia che la esecuzione del codice della WAIT comporti la sospensione del processo (quindi subito prima della sospensione, liberando la sezione critica associata al semaforo, permettendo al sistema l'inserimento del descrittore nella coda) e sia in caso di semplice decremento del valore del semaforo.

2.3 Nucleo: Riepilogo Punti di Vista

PUNTO DI VISTA ESTERNO (utente)	PUNTO DI VISTA ESTERNO (programmatore)	
processo sequenziale	↳ processi concorrenti: - indipendenti - interagenti ↙ ↘	
processo NON sequenziale ⇒ insieme di processi sequenziali + vincoli di sincronizzazione	Interazione indiretta <i>competizione</i>	Interazione diretta <i>cooperazione</i>
NECESSITÀ di: - linguaggio NON sequenziale - elaboratore non sequenziale (multiprogrammazione) ↳	↑ ↑ Modello Globale	↑ Modello Locale
Necessità di strumenti di sincronizzazione		

2.4 Esempi di Uso dei Semafori

Utilizzando il *meccanismo dei semafori*, vediamo di applicare alcune **politiche di sincronizzazione** per risolvere problemi di competizione più articolati rispetto alla semplice mutua esclusione:

2.4.1 Gestione di un Insieme di Risorse Equivalenti

Un insieme di **n** processi **P0, P1, P2, ..., Pn-1** che *competono* per l'uso di una delle **m** risorse (equivalenti) **R0, R1, R2, ..., Rm-1** (si pensi alle risorse come delle stampanti equivalenti). Il generico processo Pi ha necessità di accedere ad una qualunque delle risorse **Rk**.

Banale Problema di Mutua Esclusione → l'accesso alla risorsa deve avvenire in modo mutualmente esclusivo. Se la stampante viene assegnata a un processo che stampa i suoi risultati non deve essere assegnata a nessun'altro, altrimenti le informazioni di stampa si andrebbero a mescolare.

Prima Soluzione

Si definisce *1 semaforo di mutua esclusione per ogni risorsa (m semafori)*. → *Pi* deve specificare a quale risorsa vuole accedere, facendo la *WAIT* su un *Mutex_K* specificando il valore K.

→ **NON ACCETTABILE**

Problema

Pi potrebbe rimanere bloccato sulla richiesta della risorsa *Rk* (facendo su questa una *WAIT*) mentre altre sono libere. La risorsa *Rk* potrebbe già essere stata presa in carico perché i processi non si coordinano tra loro e scelgono la risorsa da utilizzare in modo random. In questo caso dunque *un altro processo* potrebbe aver fatto la *WAIT* sul *semaforo di mutua esclusione* che protegge *Rk* e aver cominciato ad utilizzarla.

Seconda Soluzione

Per non trovarsi di fronte al problema precedente viene introdotto un **Gestore delle Risorse** (è un'entità passiva, non un altro processo), che *gestisce la allocazione dinamica delle risorse ai processi*:

- assegna una risorsa libera (qualunque) ad ogni processo che ne fa richiesta
- libera la risorsa non appena il processo non ne ha più bisogno

Per fare ciò introduciamo:

- **STRUTTURA DATI** → per sapere se la risorsa è libera o occupata. Questa può essere un *array di dimensione m* costituito da valori booleani, dove per ogni risorsa corrispondente a un indice abbiamo i seguenti valori:
 - *true* : risorsa libera
 - *false* : risorsa occupata
- **OPERAZIONI** → utilizzate dai processi.
 - *richiesta di una risorsa libera* : per acquisire la risorsa libera, se esiste: solitamente *n* è un numero a volte *molto maggiore di m* pertanto un processo potrebbe arrivare a fare una richiesta senza che ci siano risorse libere.
 - *rilascio di una risorsa* : per liberare una risorsa occupata

Schema del Generico Processo *Pi*

Dato un **processo Pi** (con *i* = 0, 1, 2, ..., *n*-1). Questo schema può anche essere inserito in un *loop senza fine*.

```
int x; /* x può assumere valori nel range 0..m-1 */  
...  
x = RICHIESTA(); /*      il parametro effettua una richiesta  
                      x rappresenta l'indice della risorsa assegnata */  
<uso della risorsa Rx>  
RILASCIO (x);  
...
```

Schema del Gestore

Descriviamo nel dettaglio le operazioni di *Richiesta* e *Rilascio* elencando cosa è necessario implementare a livello di *struttura dati*:

- Serve un semaforo per rappresentare il numero di risorse (equivalenti) R0, R1, R2, ... Rm-1 → il semaforo verrà chiamato *RISORSE* (semaforo *contatore* o *generalizzato*, con valore iniziale pari a M). Questo valore viene decrementato mano a mano che le risorse vengono assegnate e dunque incrementato quando queste tornano disponibili.
- Serve una struttura dati (*array*) per mantenere lo stato di ognuna delle risorse (rappresentato dal booleano true se la risorsa è libera o dal booleano false se la risorsa è occupata) → l'array verrà chiamato *LIBERO*
- Poiché abbiamo bisogno dell'array LIBERO (che diventa una variabile *condivisa fra più processi*) abbiamo bisogno di garantire l'accesso mutuamente esclusivo e quindi serve un semaforo binario → *MUTEX*

```
typedef enum {false, true} Boolean;
Semaphore MUTEX;      /* valore iniziale MUTEX0=1 */
Semaphore RISORSE;    /* valore iniziale RISORSE0=m */
Boolean LIBERO[m];   /* valore iniziale LIBERO[i]=true, tutte le risorse libere */
*/
int RICHIESTA() {
    int i;           /* i può assumere valori nel range 0..m-1*/
    wait(RISORSE);
    wait(MUTEX);                // PROLOGO
    for (i = 0; ! LIBERO[i]; i++);
    /* trovato indice i per cui libero[i] è true */ //
    LIBERO [i] = false;          // <s.c>
    signal(MUTEX);              // EPILOGO
    return i; // otteniamo l'indice della risorsa
}

void RILASCIO(int x){
    wait(MUTEX);                // PROLOGO
    LIBERO[x] = true;            // <s.c>
    signal(MUTEX);              // EPILOGO
    signal(RISORSE);
}
```

Richiesta

Fino a quando il valore del semaforo *Risorse* è maggiore di 0, la *wait()* va a decrementare il valore del semaforo. Quando il numero di risorse disponibili arriva a 0, il processo viene bloccato sulla stessa *wait()*, facendo passare il processo dallo stato di esecuzione a bloccato.

Sia che la *wait()* sia bloccante, sia che il processo ad un certo punto sia stato risvegliato tornando in stato di esecuzione, questo va a avanti nella richiesta, cercando quale tra le risorse disponibili è libera. Per fare questa operazione, deve risolvere il problema della mutua esclusione (sezione critica). Una volta entrati nel ciclo for è sicuro che ci sia almeno una risorsa libera per il processo, in quanto o la *wait* era passante o questo è tornato in esecuzione dopo che almeno una risorsa si è liberata.

Rilascio

Anche su questa operazione è presente una sezione critica appartenente alla stessa classe della sezione critica precedente. Si utilizza lo stesso *MUTEX* facendo una *wait()* che ci permette di accedere in modo mutualmente esclusivo all'array di risorse. La *signal(RISORSE)* avrà come effetto quello di far passare un processo da bloccato a pronto (se la coda dei processi sospesi non è vuota), oppure e la questa coda è vuota andrà ad incrementare il valore del semaforo risorse, conteggiando una risorsa in più a disposizione dei processi che ne devono fare richiesta.

Osservazioni

1. L'**ordine delle due primitive** *wait(RISORSE)* e *wait(MUTEX)* è **vitale**. Se l'ordine delle due PRIMITIVE WAIT nella procedura RICHIESTA venisse così invertito:

```
wait(MUTEX);  
wait(RISORSE);
```

La soluzione precedentemente introdotta *garantisce un'implementazione esente da DEADLOCK*, tuttavia ciò *non è garantito per l'utilizzo di più semafori*, facendo errori di programmazione come nell'osservazione corrente, con i semafori con i seguenti valori:

```
MUTEX = 1  
RISORSE = 0  
  
wait(MUTEX) → passante  
wait(RISORSE) → bloccante
```

Nessun processo può più entrare nella sezione critica protetta da MUTEX → **DEADLOCK**

2. Se il numero di risorse (*m*) si riduce a 1, la soluzione si semplifica: non ci sarebbe più bisogno di un array di booleani → basta un semaforo R inizializzato ad 1.

Il codice degenere in questo modo (le risorse sono in realtà *un'unica risorsa*):

```
RICHIESTA → wait(RISORSE)  
RILASCIO → signal(RISORSE)
```

2.4.2 Problema Lettori-Scrittori

Più processi possono accedere allo stesso insieme di dati (come ad esempio, un *file*), visto come una **risorsa condivisa tra i processi**. Nell'esempio precedente non era rilevante l'utilizzo che un processo faceva della risorsa, mentre qui lo è, per questo si distinguono due categorie di processi:

- Processi che possono **SOLO leggere i dati** → **PROCESSI LETTORI**
- Processi che possono **ANCHE modificarli** → **PROCESSI SCRITTORI**

Nella realtà è difficile avere processi che hanno un comportamento univoco nei confronti di tutte le risorse. Piuttosto, è più facile avere a che fare con processi che si comportano come lettori in alcune fasi, poi come scrittore in altre. In questo esempio definiamo degli *intervalli di tempo* in cui i processi agiscono come lettori o come scrittori.

Vincoli

1. I processi **lettori** possono accedere **contemporaneamente** alla risorsa
2. I processi **scrittori** hanno **accesso esclusivo** alla risorsa
3. I processi **lettori e scrittori** si **escludono mutuamente** nell'accesso alla risorsa

Schema Processo Lettore

```
Inizio_lettura();
<lettura>;
Fine_lettura();
```

Schema Processo Scrittore

```
Inizio_scrittura();
<scrittura>;
Fine_scrittura();
```

Le funzioni di *Inizio* possono essere funzioni che *permettono l'accesso alla risorsa*, ma anche funzioni che *provoca la sospensione del processo in esecuzione*, consentendo l'accesso alla risorsa solo quando le condizioni lo permettono.

Questo problema noto è stato usato per verificare tutti gli strumenti di sincronizzazione in ambiente globale. I suoi vincoli possono considerarsi "*una prova del 9*".

Prima Soluzione

- Serve un semaforo per la sincronizzazione nell'accesso alla risorsa degli scrittori *fra di loro* e di uno scrittore *nei confronti dei lettori* → semaforo *SYNCH* inizializzato a 1 che consentirà di rispettare i vincoli 2 e 3.
- Serve un contatore per il numero di lettori in accesso alla risorsa → *num_lettori* inizializzato a 1. Serve ai lettori per sapere quando è il momento di sbloccare un eventuale scrittore bloccato su *SYNCH*: a *Inizio_lettura* gli scrittori *incrementano* questo numero, mentre a *Fine_lettura* lo decrementano, in modo da sbloccare uno scrittore rimasto bloccato su *SYNCH* quando questo è uguale a 0.
- Poiché abbiamo bisogno del contatore *num_lettori* (che diventa una variabile condivisa fra più processi) si deve garantire l'accesso mutuamente esclusivo → *MUTEX*

Anche *SYNCH* è tecnicamente un *MUTEX* (semaforo binario), ma gli si associa un nome diverso per la sua funzione.

```
Semaphore MUTEX; /* valore iniziale 1 */
Semaphore SYNCH; /* valore iniziale 1 */
int num_lettori = 0; /* valore iniziale 0 */

void Inizio_lettura() {
    wait(MUTEX);
    num_lettori++; // 
    if (num_lettori == 1) // < s.c >
        wait(SYNCH); // 
    signal(MUTEX);
}

void Fine_lettura() {
```

```

    wait(MUTEX);
    num_lettori--;
    if (num_lettori == 0)      // < s.c >
        signal(SYNCH);      //
        signal(MUTEX);
}

void Inizio_scrittura() {
    wait(SYNCH);
}

void Fine_scrittura() {
    signal(SYNCH);
}

```

Inizio lettura

Se quello che tenta di accedere alla risorsa è *il primo lettore*, allora questo primo lettore deve fare una *wait(SYNCH)*, che andrà a verificare il valore del semaforo *SYNCH* che sarà *uguale a 1 (WAIT PASSANTE)* se e solo se non c'è alcun processo scrittore che ha preso possesso della risorsa desiderata in scrittura. Il primo lettore potrà accedere alla *signal(MUTEX)* solo se la *wait(SYNCH)* non è bloccante.

Fine lettura

Qui è presente l'operazione di decrementazione, complementare a quella fatta a *Inizio lettura*. Se quello che tenta di accedere alla risorsa è *l'ultimo lettore del blocco*, la risorsa dopo questo non è più utilizzata da nessun lettore, quindi va a fare la *signal(SYNCH)* e se c'è un processo scrittore in coda viene liberato, mentre se non è presente viene fatto un *incremento* del semaforo (decrementato in precedenza con la *wait* passante del semaforo, che aveva occupato la risorsa).

Osservazioni

- Il **semaforo MUTEX** garantisce la ***mutua esclusione sull'accesso alla variabile num_lettori***
- Il **semaforo SYNCH** garantisce la ***mutua esclusione fra processi lettori e processi scrittori*** e fra i processi scrittori
- Se c'è ***un processo scrittore che sta accedendo*** alla risorsa e ***n processi lettori in attesa***, ***allora uno (il primo) attende sul semaforo SYNCH***, mentre ***gli altri n-1 processi lettori sono in attesa sul semaforo MUTEX***. Una volta terminata la scrittura, il processo scrittore effettua una *signal(SYNCH)* che permetterà al primo lettore di tornare in esecuzione.
- Questa soluzione ***dà priorità ai processi lettori***. Appena un lettore riesce a superare un *wait(SYNCH)*, tutti gli altri lettori non effettuano più un controllo e passano. Se il *lotto di lettori* non arriva mai al termine, *il numero di lettori non arriva mai a 0*, e non permette di eseguire la *signal(SYNCH)*. I processi scrittori possono essere bloccati indefinitamente → ***STARVATION***. Il semaforo, come meccanismo a livello implementativo, non ha un problema di starvation, in quanto la coda dei processi dei semafori è gestita in modalità FIFO.

Seconda Soluzione

Analoga alla precedente ma anch'essa non definitiva. Introduce ***STARVATION***, ma per i lettori.

Regole per la STARVATION

- Un processo lettore non deve accedere alla risorsa se c'è un processo scrittore in attesa
→ *evita starvation scrittori*. Bloccando l'accesso ai nuovi lettori, il numero di lettori può tornare a 0.
- Tutti i processi lettori in attesa, alla fine di una scrittura, devono avere priorità sull'accesso alla risorse rispetto all'accesso di un altro processo scrittore → *evita starvation lettori*.

Terza Soluzione

Questa terza soluzione risolve il problema della starvation sia dei lettori che degli scrittori.

La soluzione per tale ragione risulta più complicata:

- Serve una variabile *contatore* che conta *quanti lettori sono in accesso alla risorsa* (ricordiamo che ci può essere un insieme di processi lettori contemporaneamente in accesso) → *lettori_attivi*
- Serve una variabile *contatore* che conta *quanti lettori sono bloccati in attesa della risorsa* → *lettori_bloccati*

N.B.1 - Il blocco di un lettore (che incrementa questo contatore) ci sarà sia se la risorsa è impegnata già da uno scrittore (si bloccherà il primo lettore di un 'lotto') e sia se si è formata coda di scrittori (si bloccheranno tutti i lettori di un 'lotto') → evita la starvation degli scrittori!

N.B.2 - Il valore di lettori bloccati servirà nella *Fine_scrittura* per fare in modo che lo scrittore che sta liberando la risorsa possa verificare la presenza di lettori in attesa e quindi dare il controllo a loro invece di darlo ad uno scrittore → evita la starvation dei lettori!

- Serve una *variabile booleana* (inizializzata a *false*) che indica, se è vera, che c'è uno *scrittore in accesso alla risorsa* (non serve un contatore) → *scrittori_attivi*
- Serve una variabile contatore che conta quanti scrittori sono bloccati in attesa della risorsa → *scrittori_bloccati*
- Poiché abbiamo bisogno di tutte queste variabili (che sono tutte variabili condivise fra più processi) si deve garantire l'accesso mutuamente esclusivo → *MUTEX*
- Servono due semafori, uno per sospendere i lettori e uno per sospendere gli scrittori → *S_LETTORI* e *S_SCRITTORI*

```
typedef enum {false, true} Boolean;
Semaphore MUTEX;           /* valore iniziale 1 */
Semaphore S_LETTORI;        /* valore iniziale 0 */
Semaphore S_SCRITTORI;      /* valore iniziale 0 */
int lettori_attivi = 0;      /* valore iniziale 0 */
Boolean scrittori_attivi = false; /* valore iniziale false */
int lettori_bloccati = 0;     /* valore iniziale 0 */
int scrittori_bloccati = 0;   /* valore iniziale 0 */

void Inizio_lettura() {
    wait(MUTEX);
    if (!scrittori_attivi && scrittori_bloccati==0){ // 
        signal(S_LETTORI);                         //
        lettori_attivi++;                          // < s.c >
    }
    else lettori_bloccati++;                      //
    signal(MUTEX);
    wait(S_LETTORI);
}
```

```

void Fine_lettura() {
    wait(MUTEX);
    lettori_attivi--;
    if (lettori_attivi==0 && scrittori_bloccati>0){ // 
        scrittori_attivi = true; // < s.c >
        scrittori_bloccati--; // 
        signal(S_SCRITTORI); // 
    }
    signal(MUTEX);
}

void Inizio_scrittura() {
    wait(MUTEX);
    if (lettori_attivi == 0 && !scrittori_attivi) { // 
        signal(S_SCRITTORI); // 
        scrittori_attivi = true; // < s.c >
    }
    else scrittori_bloccati++;
    signal(MUTEX);
    wait(S_SCRITTORI);
}

void Fine_scrittura() {
    wait(MUTEX);
    scrittori_attivi = false; //
    if (lettori_bloccati > 0)
        do {
            lettori_attivi++; //
            lettori_bloccati--; //
            signal(S_LETTORI); //
        } // < s.c >
        while (lettori_bloccati != 0);
    else if (scrittori_bloccati > 0) {
        scrittori_attivi = true; //
        scrittori_bloccati--; //
        signal(S_SCRITTORI); //
    }
    signal(MUTEX);
}

```

Inizio_lettura

Il generico lettore si deve chiedere *se non ci sono scrittori attivi* (vincolo numero 3) e *se non ci sono scrittori bloccati* (per evitare la starvation di quest'ultimi).

- Se le *condizioni sono vere*, questo lettore diventerà un lettore attivo (incrementando il relativo valore) e farà una *signal(S_LETTORI)*. Questa non andrà mai a liberare un lettore, ma al contrario *andrà ad incrementare il valore del semaforo* (inizializzato a 0). In questo modo la *wait(S_LETTORI)* fatta a valle della sezione critica risulterà *passante* incrementando la relativa variabile.
- Se le *condizioni sono false*, il lettore incrementa il valore dei lettori bloccati e la *wait(S_LETTORI)* risulterà *bloccante*.

Fine lettura

A fine lettura, si decrementa il valore dei lettori attivi e in seguito il processo controlla se è l'ultimo lettore, e se essendolo è presente una coda di scrittori.

- Se le condizioni sono vere, il primo scrittore in coda deve essere liberato mediante una `signal(S_SCRITTORI)`. Prima di fare ciò, è necessario sistemare le variabili relative alla scrittura in modo che siano coerenti con questo tipo di operazione, mettendo a *true* il valore di *scrittori attivi* e *decrementando* il valore di *scrittori bloccati*.
- Se le condizioni sono false, viene *incrementato* il valore di *scrittori bloccati*.

Inizio scrittura

Lo scrittore deve controllare che non ci siano lettori attivi (vincolo numero 3) e che non ci siano scrittori attivi (vincolo numero 2).

- Se le condizioni sono vere il codice sarà in parte simile a quello di *Inizio lettura()*. La variabile *scrittori_attivi* verrà messa a *true* e attraverso una `signal(S_SCRITTORI)` ai va ad incrementare la il valore del semaforo inizializzato a 0.
- Se le condizioni sono false, il processo diventa un nuovo scrittore bloccato, andando a incrementare il valore della variabile *scrittori bloccati*.

La `wait(S_SCRITTORI)` ha il compito di iniziare la scrittura mettendo in coda il semaforo.

Fine scrittura

Con la fine scrittura si ha nuovamente una sezione critica. In essa, si controlla se ci sono lettori bloccati prima degli scrittori bloccati, per evitare la starvation dei lettori.

- Se la condizione è vera, fino a quando non si termina la serie di lettori bloccati, questi vengono a turno messi in esecuzione, andando prima a incrementare il valore di *lettori attivi* e a decrementare quello di *lettori bloccati*, facendo in seguito una `signal(S_LETTORI)`.
- Se la condizione è falsa, si procede controllando se sono presenti scrittori bloccati. Se sono presenti scrittori bloccati (non è presente un ciclo while come nel caso precedente perchè al massimo ci può essere un solo scrittore in accesso), la varabile *scrittori attivi* viene settata a *true*, si decrementano gli scrittori bloccati e si manda in esecuzione il primo di essi facendo una `signal(S_SCRITTORI)`.

Osservazione

I semafori *S_LETTORI* e *S_SCRITTORI* (binari, sempre inizializzati a 0) sono detti **semafori PRIVATI**:

- solo i processi lettori/scrittori possono fare la *wait* sul corrispondente semaforo
- ogni altro processo può fare la *signal*

Simulazione di Esecuzione

```
// Processo Lettore L1 -> Inizio_lettura();
wait(Mutex);
if(!scrittori_attivi && scrittori_bloccati == 0) {
    signal(S_LETTORI); // S_LETTORI, precedentemente uguale a 0, diventa
S_LETTORI = 1
    lettori_attivi++;
}
wait(S_LETTORI); // siccome S_LETTORI = 1 la wait è passante, e riporta
S_LETTORI = 0
<lettura>
```

```

// Mentre L1 è in lettura, arriva un secondo lettore L2 -> Inizio_lettura()
// L2 -> stesso codice -> <lettura>
// ...
// Ln -> ... -> <lettura> (ottenendo lettori_attivi = n)

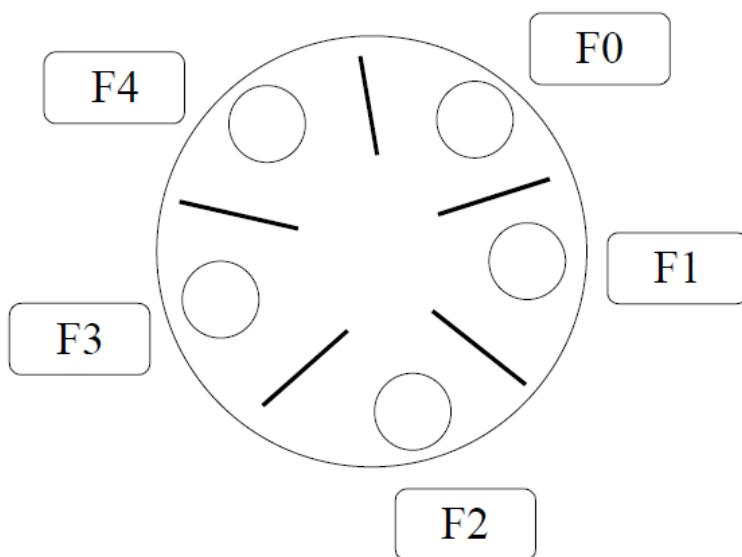
// S1 -> Inizio_scrittura()
wait(Mutex);
if(lettori_attivi == 0 && ... ){ // condizione falsa in quanto abbiamo in esecuzione n>0 lettori
    /* codice non rilevante */
}
else scrittori_bloccati++;
signal(Mutex);
wait(S_SCRITTORI); // WAIT BLOCCANTE -> il semaforo non è stato incrementato (perchè non è entrato nell'if)
                    // quindi S_SCRITTORI = 0

// Supponiamo che via via i vari Li completino la lettura, Li -> Fine_lettura()
lettori_attivi--; // mano a mano che i processi finiscono, questo valore arriva a 0
if(lettori_attivi == 0 && scrittori_bloccati > 0){ // condizione vera
    scrittori_attivi = true;
    scrittori_bloccati--; // essendoci un solo scrittore bloccato in coda, questo il valore torna a 0
}
signal(S_SCRITTORI); // lo scrittore S1 viene sbloccato e può eseguire -> <scrittura>

```

2.4.3 Problema della Cena dei Filosofi

Cinque filosofi passano il tempo a **pensare** (senza bisogno di risorse) e a **mangiare** I filosofi si trovano davanti ad una tavola rotonda su cui c'è una ciotola di riso e hanno **solo cinque bastoncini** (rappresentano le risorse, due di questi sono necessari per mangiare). Ogni filosofo alterna una fase di meditazione e una in cui vuole mangiare.



Problema

Poiché un filosofo per mangiare ha bisogno di due bastoncini, i *filosofi non possono mangiare tutti assieme contemporaneamente*. → **Problema di Mutua Esclusione nell'Accesso alle Risorse Bastoncini**.

Se F2 sta impiegando le risorse per mangiare, allora non possono mangiare né F1 né F3.

Soluzione

Associare un semaforo ad ogni bastoncino.

```
Semaphore S_BASTONCINI [5]; /* array di semafori */  
/* valore iniziale di ognuno = 1 (risorsa libera) */
```

Schema Processo Filosofo i-esimo

```
while (true) {  
    wait(S_BASTONCINO[i]);  
    wait(S_BASTONCINO[(i+1) % 5]);  
    <mangia>;  
    signal(S_BASTONCINO[i]);  
    signal(S_BASTONCINO[(i+1) % 5]);  
    <pensa>  
}
```

Il filosofo alterna, in un ciclo senza fine, una fase dove mangia e una in cui pensa.

Problemi

Supponiamo che i cinque filosofi decidano di mangiare contemporaneamente

- ognuno acquisisce il bastoncino alla sua sinistra
- ognuno tenta di acquisire il bastoncino alla sua destra
- blocco di ogni filosofo
- **DEADLOCK**

Il deadlock non è certo, ma si verifica se due filosofi adiacenti decidono di mangiare nello stesso momento.

Soluzioni

- usare una **soluzione asimmetrica**:
 - un **filosofo dispari** acquisisce prima il bastoncino alla sua sinistra e poi quello alla sua destra
 - un **filosofo pari** acquisisce prima il bastoncino alla sua destra e poi quello alla sua sinistra
- permettere ad un filosofo di acquisire i **due bastoncini solo se entrambi sono disponibili** → questo deve essere fatto in una sezione critica. In questo modo si ragiona sulla *coppia di risorse, non più sulla singola risorsa*.

2.5 Cooperazione tra Processi Concorrenti

La cooperazione può avvenire mediante:

- **scambio di un segnale** che indica il verificarsi di un certo evento
- **scambio di messaggi** generati da un processo e letti da un altro (Esempio: lettore, elaboratore scrittore visto in precedenza).

La **cooperazione tra processi** prevede che l'esecuzione di alcuni di essi risulti **condizionata** dall'informazione prodotta da altri → VINCOLI SULL'ORDINAMENTO NEL TEMPO DELLE OPERAZIONI DEI PROCESSI

Nel caso del *modello dei processi ad ambiente globale*, anche i **problemi di cooperazione** possono essere risolti usando i **semafori**.

2.5.1 Esempio Scambio Segnale

n processi P1, P2, ... Pn attivati dal processo P0.

Regole

- l'esecuzione di Pi non può iniziare prima che sia giunto il segnale da P0
- ad ogni segnale inviato da P0 deve corrispondere una attivazione di Pi

Soluzione tramite Semafori

UN SEMAFORO PER OGNI PROCESSO Pi → Si con valore iniziale Si0 = 0

```
// processo Pi
{
    ...
    while (true){
        wait (Si); // eventuale attesa SEGNALE
        < azioni di Pi >
    }
}

// processo P0
{
    ...
    while (true){
        ...           // scelta processo Pi
        signal (Si); // invio SEGNALE sul corrispondente semaforo
        ...
    }
}
```

Dall'invariante dei semafori, abbiamo:

$$nw(Si) \leq ns(Si) + Si_0$$

Dato che $Si_0 = 0$

$$nw(Si) \leq ns(Si)$$

Quindi, il *numero di volte che il processo Pi è stato attivato* è minore o uguale al *numero di segnali inviati da P0*.

2.5.2 Problema Produttore - Consumatore

Per illustrare in generale il concetto di cooperazione tra processi, si consideri il problema del **PRODUTTORE** e del **CONSUMATORE**. → Un processo **PRODUTTORE** produce informazioni che sono consumate da un processo **CONSUMATORE**.

Un Consumatore è un *processo che consuma il dato* (i processi lettori si limitano a leggere il file, pertanto non possono essere considerati consumatori).

Esempio:

Un processo di stampa che produce caratteri (*produttore*) che sono consumati dal processo gestore della stampante (*consumatore*).

Esempio:

Un processo compilatore che produce codice assembler (*produttore*) consumato da un processo Assemblatore (*consumatore*).

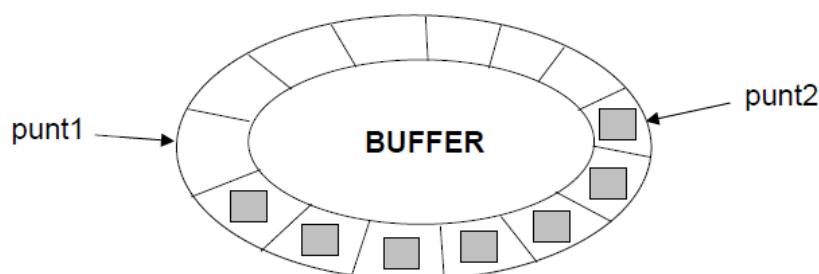
Osservazione: Per consentire una esecuzione concorrente dei processi produttore e consumatore occorre disporre di un buffer (di dimensione maggiore di 1), in modo che un produttore possa inserire un elemento, mentre il consumatore ne preleva un altro. Il suo utilizzo conferisce *flessibilità in fase di esecuzione*. Il processo consumatore, in particolare, può avere i dati necessari alla sua esecuzione *già pronti in un buffer*, andandoli a consumare rispettando i propri tempi.

Il buffer per la comunicazione può essere codificato esplicitamente facendo uso di *memoria condivisa* (nel caso di **modello ad ambiente GLOBALE**) oppure *essere fornito dal S.O.* attraverso l'uso di una *funzione di comunicazione fra processi* (implicito quindi negli strumenti di comunicazione nel caso di **modello ad ambiente LOCALE**).

Nel seguito si supporrà di essere nel primo caso, cioè con un buffer (e altre variabili) condiviso fra i processi produttore e consumatore.

Implementazione del Buffer

I processi produttore e consumatore condividono per comunicare un *buffer circolare a N posti* e due variabili (*punt1* e *punt2*). Queste variabili identificano la posizione *dove il produttore deve inserire* il dato e da cui il *consumatore deve estrarre* il dato.



Regole

- il *consumatore* non può prelevare un messaggio dal buffer se questo è vuoto
- il *produttore* non può depositare un nuovo messaggio nel buffer se questo è pieno
- Il processo produttore e quello consumatore **non devono mai contemporaneamente alla stessa posizione** del buffer (accesso mutuamente esclusivo alla posizione del buffer)

$$0 \leq D - E \leq N$$

D = numero di messaggi depositati

E = numero messaggi estratti

Si consideri una semplice implementazione del *buffer* come un array e di *punt1* e *punt2* come indici "circolari" di tale array.

Inserimento

```
buffer[punt1] = messaggio;
punt1 = (punt1 + 1) % N // aggiornamento puntatore
```

Prelievo

```
messaggio = buffer[punt2];
punt2 = (punt2 + 1) % N // aggiornamento puntatore
```

Se *punt1* == *punt2* possiamo trovare due casistiche:

- buffer tutto pieno → può agire solo il consumatore (blocco produttore)
- buffer tutto vuoto → può agire solo il produttore (blocco consumatore)

Soluzione tramite Semafori

Si introducono due semafori:

- ***spazio_disponibile***
 - con valore iniziale uguale a N
 - all'inizio c'è spazio per N messaggi
- ***messaggio_disponibile***
 - con valore iniziale uguale a 0
 - all'inizio il buffer è vuoto

```
processo PRODUTTORE{
    while (true){
        <produzione messaggio>
        wait(spazio_disponibile); // passante al primo tentativo, decrementerà
        <deposito messaggio>      // il valore del semaforo
        signal(messaggio_disponibile); // deve sbloccare eventuali consumatori
    }                                // oppure incrementare il semaforo
}

processo CONSUMATORE{
    while (true){
        wait (messaggio_disponibile);
        <prelievo messaggio>
        signal(spazio_disponibile);
        <consumo messaggio>
    }
}
```

In questo caso, i processi produttore e consumatore possono accedere **contemporaneamente** al BUFFER (utilizzando le due variabili *punt1* e *punt2*), se non si è in una situazione limite (buffer pieno o buffer vuoto).

2.5.3 Problema Produttori - Consumatori

In questo esempio si hanno **più produttori e più consumatori** (anche con molteplicità diverse, H e K). Le informazioni preparate da un produttore possono essere utilizzate da un qualunque consumatore.

Così facendo, la variabile *punt1* è utilizzata da *tutti i produttori*, mentre *punt2* da *tutti i consumatori* → **necessità di mutua esclusione sugli indici punt1 e punt2**. Sono necessari **due semafori** *MUTEX1* e *MUTEX2* (inizializzati a 1).

```
//processo PRODUTTOREi (con i che varia da 1 a H)
{
    while (true){
        <produzione messaggio>
        wait(spazio_disponibile);
        wait(MUTEX1);
        <deposito messaggio> // <s.c>
        signal(MUTEX1);
        signal(messaggio_disponibile);
    }
}

// processo CONSUMATOREj (con j che varia da 1 a K)
{
    while (true){
        wait (messaggio_disponibile);
        wait(MUTEX2);
        <prelievo messaggio> // <s.c>
        signal(MUTEX2);
        signal(spazio_disponibile);
        <consumo messaggio>
    }
}
```

In questo modo *non è possibile avere più produttori in accesso al buffer*, così come *non è possibile avere più consumatori in accesso al buffer*. Il produttore *i-esimo* può andare *in parallelo ad operare sul buffer* insieme con il consumatore *j-esimo*.

2.6 Costrutti di Sincronizzazione in Ambiente Globale

2.6.1 La Necessità

L'uso di **semafori** e delle primitive *WAIT* e *SIGNAL* consentono di risolvere *ogni problema di sincronizzazione in ambiente globale*, ma sono strumenti di **basso livello**. Inoltre, un loro uso scorretto produce **errori** difficilmente rilevabili.

Primo Esempio

Supponiamo di avere una semplice sincronizzazione:

```
wait(mutex)
SEZIONE CRITICA
signal(mutex)
```

e di aver invertito l'ordine delle due primitive:

e di aver invertito l'ordine delle due primitive:

```
signal(mutex)
SEZIONE CRITICA
wait(mutex)
```

In particolare supponiamo di avere tre processi P_1 , P_2 e P_3 : P_1 con il codice sbagliato come indicato precedentemente; P_2 e P_3 con il codice giusto.

Supponiamo ora di avere questo ordine di esecuzione:

- P_1 signal(mutex) \rightarrow $mutex_v = 2$
 - SEZIONE CRITICA \rightarrow 1 processo nella SC
- P_2 wait(mutex) \rightarrow $mutex_v = 1$
 - SEZIONE CRITICA \rightarrow 2 processi nella SC
- P_3 wait(mutex) \rightarrow $mutex_v = 0$
 - SEZIONE CRITICA \rightarrow 3 processi nella SC

Errore \rightarrow 3 processi sono all'interno della sezione critica.

Secondo Esempio

Supponiamo in questo caso di avere:

```
wait(mutex)
SEZIONE CRITICA
signal(mutex)
```

```
wait(mutex)
SEZIONE CRITICA
wait(mutex)
```

In particolare, supponiamo di avere due processi P_1 e P_2 : P_1 con il codice sbagliato come indicato precedentemente; P_2 con il codice giusto.

Supponiamo ora di avere questo ordine di esecuzione:

- P_1 wait(mutex) \rightarrow $mutex_v = 0$
 - SEZIONE CRITICA
 - wait(mutex) \rightarrow $mutex_v = 0 \rightarrow P_1$ bloccato
- P_2 wait(mutex) \rightarrow $mutex_v = 0 \rightarrow P_2$ bloccato

Errore \rightarrow 2 processi bloccati *all'esterno della sezione critica*: **DEADLOCK**

La possibilità di commettere errori usando i semafori cresce al crescere della complessità del problema di sincronizzazione da risolvere \rightarrow necessità di introdurre dei **costrutti di sincronizzazione** nei linguaggi di programmazione concorrenti.

2.6.2 Monitor

Il Monitor è stato introdotto per la prima volta nel linguaggio Concurrent Pascal consiste in un **tipo di dato astratto** (A.D.T., pensiamo al concetto di classe ma senza ereditarietà) viene applicato anche in ambito concorrente **estendendolo** con dei **VINCOLI DI SINCRONIZZAZIONE**.

Qualcosa di simile ad un monitor si può ottenere in Java utilizzando la keyword **synchronized** a *livello di metodo* (operazione di una classe) o a *livello di blocco di codice*.

Nei Monitor troviamo delle **strutture dati comuni a più processi**, e delle **operazioni uniche** invocabili su un'istanza di tipo Monitor. Su un'istanza di tipo Monitor viene garantito un **vincolo di mutua esclusione**, che fa sì che un solo processo alla volta può eseguire una delle operazioni del monitor. Se un altro i-esimo processo vuole eseguire la stessa operazione o una delle altre definite dal monitor, vengono sospesi all'esterno.

Sintassi del Monitor (Pascal-like)

```
type <nome_monitor> = monitor;
<dichiarazione di variabili locali al monitor> // corrispondenti a 'private'
// STRUTTURA DATI stato risorsa (raramente la risorsa stessa)

// interfaccia delle istanze del monitor che si sta definendo
procedure entry Proc1 (...);
begin ... end;

.....
procedure entry ProcN (...);
begin ... end;

procedure Proci1 (...);
begin ... end;
.....
procedure ProciM (...);
begin ... end;

begin <inizializzazione> end;
```

La parola chiave *entry* corrisponde al `public` di Java. Si tratta delle **sole entità visibili all'esterno di un monitor** → i processi le dovranno invocare per poter acquisire e rilasciare la risorsa.

La *Proci1* o la *ProciM*, invece possono essere invocate solo all'interno del monitor (*Proc1*, *ProcN*). Senza *entry* la variabile è come se fosse *privata*. L'*inizializzazione* viene eseguita ogni volta che si dichiara una variabile di questo tipo.

Le parole evidenziate sono *parole chiave* del linguaggio:

- `type` serve per definire un nuovo tipo
- `monitor` definisce che quel nuovo ADT è un monitor
- `procedure` definisce una procedura, la quale *non ritorna un valore*
- `function` definisce una funzione, la quale *ritorna un valore*
- `begin` e `end` definiscono il *corpo di una procedura/funzione* o la *parte di inizializzazione* (corrispondono alle *parentesi graffe* del linguaggio C)

Una volta definito il tipo monitor, bisogna crearne una **istanza**:

```
var <nome_istanza>: <nome_monitor>;
```

La chiamata ad una procedura entry del monitor presenta la sintassi *dot notation*:

```
<nome_istanza>.<nome_procedura_entry> (parametro1, ...);
```

Il **compilatore** controlla che le strutture dati, definite all'interno di un monitor, siano accedute solo attraverso le procedure/function `entry` e mai direttamente.

Primo Livello di Controllo

Un solo processo alla volta può essere attivo all'interno di un monitor→ le **procedure** di un monitor sono **eseguite in maniera mutuamente esclusiva**.

Se un processo invoca una procedura di un monitor, mentre un altro processo sta eseguendo una procedura dello stesso monitor, **viene sospeso** in attesa che il monitor sia liberato. Questi processi in attesa della liberazione di un monitor sono chiamati **processi sospesi all'esterno del monitor**.

I problemi di sincronizzazione *non si limitano alla mutua esclusione*, possono esserci ad esempio *problemi competitivi e cooperativi* nell'esecuzione di un processo. È dunque necessario un ulteriore livello di controllo.

Secondo Livello di Controllo

Mediante l'introduzione di **VARIABILI SPECIALI**, che consentono al processo di *sospendersi* nel caso non siano rispettate le **condizioni** per la sua esecuzione, consentendo ad un altro processo di tentare l'accesso. Con queste variabili è possibile anche controllare l'ordine di accesso al monitor e quindi realizzare particolari politiche di sincronizzazione.

Le *condizioni* sono ottenute dal valore sia di variabili interne al monitor che da variabili proprie di ogni processo (passate come parametri nelle procedure entry). Poiché vengono utilizzate a valle per *verificare una condizione* sono chiamate **Condition Variables**.

I processi che hanno le condizioni soddisfatte, entrano nel monitor e, in generale, modificano alcune variabili per fare sì che i processi sospesi possano riprendere la esecuzione.

Sintassi

```
var x: condition;
```

Le uniche operazioni che si possono eseguire sulle variabili *condition* si chiamano **WAIT** e **SIGNAL**.

Attenzione: Da non confondere con le omonime operazioni primitive che si usano per i semafori (hanno semantica diversa).

- `x.WAIT;`

Questa operazione è *sempre sospensiva*.

Esempio di Utilizzo:

Supponiamo che <condizione> sia la condizione che consente ad un processo di utilizzare una risorsa o un insieme di risorse. All'interno di una procedure/function del monitor dovremo scrivere qualcosa tipo:

```
if NOT <condizione> then x.WAIT;
```

In questo modo che se la <condizione> non è verificata il processo si sospende sulla variabile condition x (introdotta appunto per accogliere una coda di processi per i quali la condizione non è verificata). I processi sospesi su variabili condition a seguito della esecuzione di una WAIT vengono detti **processi sospesi all'interno del monitor**.

- `x.SIGNAL;`

Questa operazione ***risveglia*** un processo sospeso, se non c'è nessun processo sospeso, non ha nessun effetto.

Problema da Risolvere:

Sia P il processo sospeso sulla *variabile condition* x (per effetto di una WAIT su x), e Q il processo che esegue la SIGNAL su x, allora sia Q (il segnalante) che P (il segnalato) si troverebbero ad essere attivi all'interno del monitor → questo violerebbe il vincolo di MUTUA ESCLUSIONE insito nel costrutto MONITOR.

P → Processo Segnalato

Q → Processo Segnalante

L'implementazione della `x.SIGNAL` deve decidere ***a chi lasciare il controllo***. Per risolvere questo problema sono state definite ***due semantiche***:

1. Segnala e Aspetta (Soluzione di Hoare):

Il processo P (il processo segnalato) riprende immediatamente l'esecuzione, mentre Q (il processo segnalante) viene sospeso:

- Q potrà andare in esecuzione solo quando P sarà uscito dal monitor oppure si sarà sospeso su una variabile condizione
- in ogni modo Q torna in esecuzione prima di qualunque altro processo che arriva dall'esterno del monitor
- il monitor risulterà libero solo quando anche Q sarà uscito

Nel seguito vedremo l'uso del costrutto monitor e la sua implementazione tramite semafori con questa semantica.

2. Segnala e Continua:

Il processo Q (il processo segnalante) continua la propria esecuzione, mentre P (il processo segnalato) viene sospeso in attesa che Q liberi il monitor:

- P potrà andare in esecuzione solo quando Q sarà uscito dal monitor oppure si sarà sospeso su una variabile condizione
- in ogni modo P torna in esecuzione prima di qualunque altro processo che arriva dall'esterno del monitor
- il monitor risulterà libero solo quando anche P sarà uscito

Il problema nell'utilizzo di questa semantica è che il programmatore deve essere consapevole di essa dato che, quando il controllo viene finalmente dato al processo P, non c'è nessuna garanzia che le condizioni che avevano portato Q a segnalare P siano ancora soddisfatte e quindi *P deve per forza riverificarle* prima di poter procedere.

Primo Esempio Uso Monitor: Gestione Risorse Equivalenti

Supponiamo di avere un insieme di processi che facciano uso della stessa risorsa, che quindi va usata in moto *mutuamente esclusivo* → lo *stato della risorsa* (non la risorsa) va protetto in un monitor. Si tratta di un problema semplice affrontato con l'utilizzo di un monitor.

Il monitor è ***uno strumento software***, pertanto non è possibile salvare in esso la risorsa fisica, ma solo *lo stato di essa*.

```
type GESTORE = monitor;
var occupato: boolean; libero: condition;
```

```

procedure entry RICHIESTA;
begin
    if occupato then libero.WAIT; // la wait che entra nella condition è sempre
    sospensiva
    occupato := true; // non è un 'else'
end;

procedure entry RILASCIO;
begin
    occupato := false; libero.SIGNAL;
end;

begin { inizializzazione } occupato := false; end;

```

I **processi** dovranno prevedere una fase di *acquisizione* ed una di *rilascio* tramite le due **procedure entry** invocate su una istanza del monitor *GESTORE*:

```
var R: GESTORE;
```

Allora i processi dovranno avere tutti questo schema:

```

// Processo Pi
begin
    R . Richiesta;
    <uso della risorsa>
    R . Rilascio;
end;

```

In questo modo si è realizzato il corrispondente di un **semaforo binario**.

Secondo Esempio di Uso di Monitor: Lettori-Scrittori

Riprendiamo l'esempio dei processi lettori e scrittori.

Variabili del Monitor:

- `num_lettori` → numero di lettori attivi;
- `occupato` → indica se la risorsa è occupata da uno scrittore;
- `ok_lettura` → variabile CONDIZIONE su cui si sospendono i processi lettori
- `ok_scrittura` → variabile CONDIZIONE su cui si sospendono i processi scrittori

Per semplificare la codifica si supponga di avere un'ulteriore operazione applicabile su una variabile condizione `COND . QUEUE` che ritorna *true* se esistono dei processi sospesi, *false* altrimenti.

I **processi lettori e scrittori** dovranno prevedere una fase di *acquisizione* ed una di *rilascio* tramite le due **procedure entry** invocate su una istanza del monitor *LETTORI_SCRITTORI*

```
var LS: LETTORI_SCRITTORI;
```

Allora i processi dovranno avere questi schemi:

SCHEMA PROCESSO LETTORE	SCHEMA PROCESSO SCRITTORE
<pre>{ LS.Inizio_lettura; <lettura>; LS.Fine_lettura; }</pre>	<pre>{ LS.Inizio_scrittura; <scrittura>; LS.Fine_scrittura; }</pre>

```

type LETTORI_SCRITTORI = monitor;
var num_lettori: integer;
occupato: boolean;
ok_lettura, ok_scrittura: condition;

procedure entry Inizio-lettura;
begin
  if occupato or ok_scrittura.QUEUE
    then ok_lettura.WAIT;
  num_lettori := num_lettori + 1;
  ok_lettura.SIGNAL; // vedi osservazione
end;

procedure entry Fine-lettura;
begin
  num_lettori := num_lettori - 1;
  if num_lettori = 0 then ok_scrittura.SIGNAL;
end;

procedure entry Inizio-scrittura;
begin
  if num_lettori <> 0 or occupato
    then ok_scrittura.WAIT;
  occupato := true;
end;

procedure entry Fine-scrittura;
begin
  occupato := false;
  if ok_lettura.QUEUE then ok_lettura.SIGNAL; // no starvation lettori
  else ok_scrittura.SIGNAL;
end;

begin { inizializzazione }
  num_lettori := 0; occupato := false;
end;

```

Osservazione: In questa soluzione si utilizza una *politica di risveglio con meccanismo A CASCATA*. Ciò vuol dire che il processo scrittore (alla fine della sua esecuzione), una volta che si accorge che c'è coda di processi lettori *libera il primo*. Questo, cominciando la lettura, andrà a risvegliare il seguente processo lettore in attesa di eseguire.

Simulazione di Esecuzione

```

// L1
occupato = false;
numero_lettori = 0;
ok_scrittura.QUEUE = false;
// L2 e L3

```

Con queste condizioni il *Processo lettore L1* può leggere. Se in seguito arrivano i processi *L2* e *L3* questi possono iniziare anch'essi la lettura.

```
// S1  
numero_lettori =< 0; // S1 -> sospeso su ok_scrittura  
// S2  
numero_lettori =< 0; // S2 -> sospeso su ok_scrittura
```

Con l'arrivo di due *processi scrittori* durante la lettura dei 3 precedenti processi, questi vengono sospesi e messi in attesa su *ok_scrittura*.

```
// L4  
occupato = false;  
ok_scrittura.QUEUE = true; // L4 -> sospeso su ok_lettura
```

Con l'arrivo di un *nuovo processo lettore L4* questo trova *ok_scrittura.QUEUE true*, in quanto sono ora presenti dei processi scrittori in attesa di essere eseguiti. In questo modo il nuovo processo L4 viene a sua volta *sospeso su ok_scrittura*.

```
// L2 finisce la lettura  
numero_lettori = 2; // 3-1 = 2  
numero_lettori = 0; // false  
// L1 finisce la lettura  
numero_lettori = 1; // 2-1 = 1  
numero_lettori = 0; // false  
// L3 finisce la lettura  
numero_lettori = 0; // true  
ok_scrittura.SIGNAL // S1 READY  
  
// S1 riprende l'esecuzione  
occupato = true;  
  
// arriva L5, il quale trova  
occupato = true; // L5 si sospende su ok_lettura  
  
// S1 finisce la sua esecuzione  
occupato = false;  
if (ok_lettura.QUEUE) // true, sono presenti processi lettori in attesa  
    ok_lettura.SIGNAL; // L4 READY  
  
// L4 comincia la sua esecuzione  
numero_lettori = 1  
ok_lettori.SIGNAL // L5 READY  
  
// L5 esegue  
numero_lettori = 2  
ok_lettori.SIGNAL // non ci sono processi sospesi su questa variabile, nessun  
effetto  
  
// L5 termina la lettura  
numero_lettori = 1 // 2-1 = 1  
  
// L4 termina la lettura  
numero_lettori = 0  
if numero_lettori = 0 // true
```

```

ok_scrittura.SIGNAL // S2 READY

// S2 esegue
occupato = true;
// finisce la scrittura
occupato = false;
if (ok_lettura.QUEUE) // false
else ok_scrittura.SIGNAL // non ci sono scrittori in attesa, nessun effetto

```

Al termine si ritorna alla *situazione iniziale*.

Terzo Esempio di Uso di Monitor: Cena dei Filosofi

Riprendiamo l'esempio dei *filosofi*: la soluzione risolve il problema del deadlock, imponendo che un filosofo acquisisca contemporaneamente i *due bastoncini* di cui ha bisogno. I **bastoncini** rappresentano la **risorsa** il cui stato deve essere protetto da un monitor mediante una particolare politica di sincronizzazione.

Variabili del Monitor

- **STATO** → array che mantiene lo stato di ogni filosofo (*thinking, hungry, eating*): un filosofo può passare nello stato di *eating* se e solo se i suoi vicini non stanno mangiando
- **COND** → array di variabili condizione su cui si sospendono i processi filosofi quando sono "affamati", ma i due bastoncini di cui hanno bisogno sono occupati

```

type CENA_FILOSOFI = monitor;
var STATO: array[0 .. 4] of
    (thinking, hungry, eating); // enumerazione
COND: array[0.. 4] of condition;

procedure entry Pick-up(i: 0..4);
begin
    STATO[i] := hungry;
    test (i);
    if STATO[i] <> eating then
        COND[i].WAIT;
end;

procedure entry Put-down(i: 0..4);
begin
    STATO[i] := thinking;
    test(i+4 mod 5);
    test(i+1 mod 5);
end;

// Controlla se i filosofi vicini stanno mangiando
procedure test(k: 0..4);
begin
    if STATO[k+4 mod 5] <> eating and
        STATO[k] = hungry and
        STATO[k+1 mod 5] <> eating
    then begin
        STATO[k] := eating;
        COND[k].SIGNAL; // nel caso della Pick-Up non fa nulla, il filosofo
    end; // k è l'unico nella coda di esecuzione
end;

```

```

end;

begin { inizializzazione }
    for i:= 0 to 4 do STATO[i] := thinking;
end;

```

Lo stato iniziale dei filosofi è quello di *thinking*, nel quale a *nessuno* occorre avere *i bastoncini*. Indicando con CF una istanza del MONITOR:

```
var CF: CENA_FILOSOFI;
```

Il processo filosofo i-esimo avrà il seguente schema:

```

// Processo FILOSOFOi
begin
    repeat
        CF . Pick-up(i); // se ci sono le bacchette esegue
        <mangia>
        CF . Put-down(i); // se altri sono in attesa li libera
        <pensa>
    until false;
end;

```

La sintassi del *repeat until false* è la corrispondente del *while(true)*.

Simulazione di Esecuzione

I filosofi si trovano attorno a un tavolo con 5 bastoncini a disposizione.

```

// Comincia F1
Pick-up(1);
s[1] := H;
test(1);
if s[1] <> E { // non eseguita, il primo filosofo riesce a mangiare
    ...
}
}

```

Chiamando *test(1)* viene eseguito il seguente codice:

```

if s[k+4mod5] <> E // con k=1 -> 5mod5 = 0, stato del filosofo s0 (true)
and
s[1] = H; // true, lo stato di F1 è HUNGRY (true)
and
if s[k+1mod5] <> E // con k=1 -> 2mod5 = 2, stato del filosofo s2 (true)

```

La condizione è verificata, pertanto *F1 può mangiare*:

```
s[1] = E;
```

Supponiamo che arrivi *F0* (i filosofi sono 5, *F0 ... F4*)

```
// F0
Pick-up(0) {
    s[0] := H;
    test(0);
    if s[0] <> E { // vedi implementazione sotto
        ...
    }
}
```

Chiamando *test(0)* viene ripresa la funzione precedente:

```
if s[k+4mod5] <> E // con k=0 -> 4mod5 = 4, stato del filosofo s4 (true)
and
s[0] = H; // true, lo stato di F0 è HUNGRY (true)
and
if s[k+1mod5] <> E // con k=0 -> 5mod5 = 1, stato del filosofo s1 (false)
```

Poiché ci sono già filosofi adiacenti che stanno mangiando, *F0* non può prendere le bacchette che gli servono per cominciare, pertanto, riprendendo il codice precedente:

```
if s[0] <> E { // true
    COND[0].WAIT; // F0 sospeso
}
```

Se a questo punto arrivasse anche *F2* verrebbe ugualmente sospeso. Supponiamo ora che *F1* finisca di mangiare:

```
// F1 finisce di mangiare
Put-down(1) {
    s[1] := T;
    test(i+4mod5);
    test(i+1mod5);
}
```

Riprendendo la *test(1+4mod5)*:

```
// test(i+4mod5) -> con i=1 test(5mod5) -> test(0)
if s[k+4mod5] <> E // con k=0 -> 4mod5 = 4, stato del filosofo s4 (true)
and
s[0] = H; // true, lo stato di F0 è HUNGRY (true)
and
if s[k+1mod5] <> E // con k=0 -> 5mod5 = 1, stato del filosofo s1 (true)
```

Si è verificato che il *primo filosofo adiacente a F1* (*F0*, a sinistra) fosse *affamato* (in attesa dunque delle bacchette), in seguito si è verificato che nessun filosofo adiacente ad esso stesse mangiando. Poiché tutte le condizioni sono rispettate, *F0* può *mangiare*:

```
// Condizione precedente con k = 0
if s[k+4mod5] <> E and s[0] = H and if s[k+1mod5] {
    s[0] = E;
    COND[0].SIGNAL; // F0 RIATTIVATO, READY
}
```

F0, precedentemente sospeso e in possesso di una sola bacchetta, può *raccogliere la seconda e cominciare a mangiare*, terminando la fase di *Pick-up*. La *Put-down(1)* continua, facendo ora la *test(1+4mod5)* sul secondo filosofo adiacente a *F0*: quello *alla sua destra*:

```
// test(i+1mod5) -> con i=1 test(2mod5) -> test(2)
if s[k+4mod5] <> E // con k=2 -> 6mod5 = 1, stato del filosofo s1 (true)
and
s[2] = H; // false, lo stato di F2 non è HUNGRY (false)
and
if s[k+1mod5] <> E // con k=2 -> 3mod5 = 3, stato del filosofo s3 (true)
```

Poiché la condizione interna alla *if clause* non è rispettata la *Pick-up* termina.

Quarto Esempio di Uso di Monitor: Produttori-Consumatori

A differenza dei precedenti, si tratta di un esempio in *ambito cooperativo*. Si riprende l'esempio di più processi *PRODUTTORE* e di più processi *CONSUMATORE* che comunicano utilizzando un *buffer circolare*. La **risorsa buffer viene protetta da un monitor**.

I processi Produttore hanno bisogno di una *procedura entry* con questa interfaccia:

```
procedure entry DEPOSITA(x: messaggio);
```

I processi produttore della seguente *procedura entry*:

```
procedure entry PRELEVA(var x: messaggio);
```

Le condizioni per cui un processo può attendere possono essere rappresentate con **due variabili condizione**:

- `non_pieno` → per i Produttori che si sospendono in attesa che il *buffer non sia più pieno*
- `non_vuoto` → per i Consumatori che si sospendono in attesa che il *buffer non sia più vuoto*

```
type BUFFER_CIRCOLARE = monitor;

var buff: array[0..N-1] of messaggio; // formato definito in precedenza
punt1, punt2: 0..N-1; // variano da 0 a N-1 in modo circolare
inseriti: 0..N;
non_pieno, non_vuoto: condition;

procedure entry DEPOSITA(x: messaggio);
begin
  if inseriti = N then non_pieno.WAIT;
  buff[punt1] := x;
  punt1 := (punt1 + 1) mod N; □
  inseriti := inseriti + 1;
  non_vuoto.SIGNAL; //sveglia il primo consumatore in attesa, se esiste
end;

// con la sintassi 'var' si fa riferimento a un parametro di uscita
procedure entry PRELEVA(var x: messaggio);
begin
  if inseriti = 0 then non_vuoto.WAIT;
  x := buff[punt2];
  punt2 := (punt2 + 1) mod N;
```

```

inseriti := inseriti - 1;
non_pieno.SIGNAL; //sveglia il primo produttore in attesa, se esiste
end;

begin {inizializzazione}
  inseriti := 0; punt1 := 0; punt2 := 0;
end;

```

Il valore della variabile `inseriti` nell'implementazione precedente attraverso *semafori* era rappresentato sia dal valore del semaforo per sospendere i produttori (N), sia da quello utilizzato per sospendere i consumatori (0). Il valore delle variabili `punt1` e `punt2` quando vengono inizializzati potrebbe essere qualsiasi rappresentante gli N indici del buffer, trattandosi di un *buffer circolare*.

Note su sintassi Pascal-like:

1. La parola-chiave `mod` realizza l'operazione modulo (resto intero della divisione)
2. La parola-chiave `var` nel passaggio dei parametri serve a ottenere il passaggio per riferimento.

Indicando con *BUFFER* una *istanza del MONITOR*:

```
var BUFFER: BUFFER_CIRCOLARE;
```

I processi produttore e consumatore avranno i seguenti schemi:

```

// Processo PRODUTTOREi
begin
  repeat
    <produzione messaggio m>
    BUFFER . DEPOSITA(m);
  until false;
end;

// Processo CONSUMATOREj
begin
  repeat
    BUFFER . PRELEVA(m);
    <consumo messaggio m>
  until false;
end;

```

Osservazione: L'aver inserito il monitor nel buffer fa sì che il *parallelismo che si aveva nella soluzione con i semafori* (in particolare nel caso di più produttori e più consumatori) *non si possa avere*. Non è possibile che produttore e consumatore accedano in contemporanea al buffer, anche se concettualmente *potrebbero farlo*. In realtà, basterebbe che i due processi non agissero contemporaneamente sulla stessa locazione del buffer.

Realizzazione Monitor (Segnala e Aspetta)

Il costrutto di monitor viene trattato dal compilatore *come un semaforo*. Il compilatore si assicura che le **procedure entry** di un monitor vengano eseguite in mutua esclusione: per farlo, associa ad ogni istanza di un monitor un semaforo **MUTEX** inizializzato a 1. Quindi, per ogni dichiarazione:

```
procedure entry Proc (...); ... end;
```

il compilatore genera una *wait(MUTEX)* (prologo) all'inizio della procedura. Come ci si aspetta, alla fine genera una *signal(MUTEX)* (epilogo). Questa operazione (liberare il monitor mediante la *signal*) non deve essere fatta qualora si hanno processi sospesi mediante la **semantica segnala e aspetta**. Questi processi segnalanti infatti, non devono essere "scavalcati" da altri processi al di fuori del monitor, ma hanno precedenza rispetto agli altri. Questi processi segnalanti vengono sospesi su un **semaforo URGENT** (inizializzato a 0), e ci sarà un **contatore URGENTCOUNT** che rappresenta il numero di processi sospesi su URGENT, ovvero quelli che hanno fatto una *SIGNAL* su una variabile condizione e che per effetto della semantica segnala e aspetta sono stati sospesi su *URGENT*. Al posto della semplice *signal(MUTEX)*, come epilogo della procedura ritroviamo il seguente codice:

```
if URGENTCOUNT > 0 then signal(URGENT);
else signal(MUTEX);
```

Per ogni variabile condizione *COND* viene associato:

- un semaforo **CONDSEM**, inizializzato a 0, sul quale si sospendono i processi che effettuano la *COND.WAIT*. Nel caso di *array di variabili condition* deve esserci *un array di semafori per ogni variabile condizione*.
- un contatore **CONDCOUNT**, inizializzato a 0, per tenere traccia del numero di processi sospesi sulla variabile condizione

Vediamo ora come vengono realizzate le operazioni **WAIT** e **SIGNAL** su una variabile condizione **COND**:

```
// COND.WAIT
CONDCOUNT := CONDCOUNT + 1; // il processo viene sospeso
if URGENTCOUNT > 0 then signal(URGENT);
else signal(MUTEX);
wait(CONDSEM);
CONDCOUNT := CONDCOUNT - 1;
```

Prima di sospendersi un processo deve eseguire il codice visto in precedenza alla fine di una procedure entry. Deve controllare che non ci siano processi sospesi segnalanti da risvegliare. Se non ci sono può liberare il monitor andando a risvegliare eventualmente un processo in attesa all'esterno di esso. Una volta che questo processo viene risvegliato dalla sospensione esso potrà andare a *decrementare il valore del contatore indicante il numero di processi sospesi*.

```
// COND.SIGNAL
if CONDCOUNT > 0 then begin
    URGENTCOUNT := URGENTCOUNT + 1;
    signal(CONDSEM);
    wait(URGENT);
    URGENTCOUNT := URGENTCOUNT - 1;
end;
```

Se sono presenti processi sospesi, il processo che esegue la *SIGNAL* si deve sospendere (sul semaforo *URGENT*). Prima di farlo, risveglia un processo in coda di attesa (per forza presente se si esegue la *SIGNAL*).

Priorità nelle Code delle Variabili Condizione

Negli esempi precedenti, si è supposto che la gestione delle code di processi associate alle variabili condizione fosse di tipo **FIFO** → con l'operazione **SIGNAL** eseguita su una variabile condizione, viene *risvegliato il processo in attesa da più tempo*. Questo tipo di gestione **garantisce** che **non ci siano dei fenomeni di ATTESA INDEFINITA** da parte dei processi sospesi (**NO STARVATION**). Ci sono casi in cui la politica *FIFO* non risulta adeguata. Introducendo una **priorità** è tuttavia possibile avere starvation, altrimenti occorre *complicare la soluzione* per fare in modo di gestire un *numero massimo di processi che hanno accesso alla risorsa*. Superato questo numero, la coda di processi in attesa di essa deve essere gestito, resettando il contatore e ripartendo con una **nuova serie di ingressi gestiti a priorità**.

Soluzione

Occorre una modifica della sintassi della **WAIT**. Consiste nell'introduzione di un parametro che rappresenta la **priorità di accodamento**.

```
cond . WAIT(p);
```

Nel momento che viene eseguita una

```
cond . SIGNAL;
```

viene risvegliato il processo a priorità maggiore e quindi, ad esempio, con minore valore di **p**. Solitamente un processo con priorità maggiore è quello con il minore valore di p.

Primo Esempio: Gestione di Risorse Equivalenti

Consideriamo ancora una volta il problema di più processi che facciano uso della stessa risorsa R. Supponiamo che la **politica di gestione** della risorsa si basi su questa considerazione: "quando R viene rilasciata, la risorsa viene assegnata al processo che la usa per il periodo di tempo più breve". Se la risorsa è libera, l'*informazione sul tempo* risulta inutile, ma se la risorsa è occupata, il tempo serve per accodare con questo valore sulla variabile condition **Libero**, in modo che una volta attuato il *rilascio*, la **SIGNAL** vada a risvegliare il processo che ha intenzione di utilizzare la risorsa per meno tempo possibile. L'esempio precedente si modifica in questo modo:

```
type GESTORE = monitor;
var occupato: boolean;
    libero: condition;

procedure entry RICHIESTA(tempo: integer);
begin
    if occupato then libero.WAIT(tempo);
    occupato := true;
end;

procedure entry RILASCIO;
begin
    occupato := false;
    libero.SIGNAL;
end;

begin { inizializzazione }
    occupato := false;
end;
```

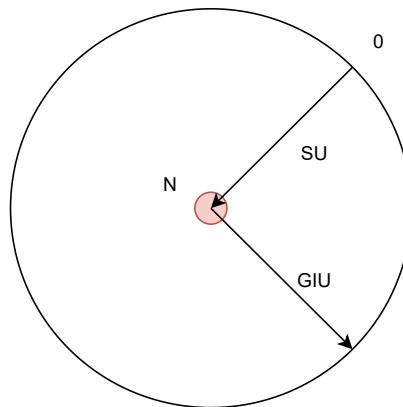
Il Processo avrà la seguente struttura:

```
var t: integer;
begin
  <definizione di t> // tempo per cui usa la risorsa
  R . Richiesta(t);
  <uso di R>
  R . Rilascio;
end;
```

Secondo Esempio: Accesso Disco a Testine Mobili

La **politica di gestione** di questa risorsa deve tendere a *minimizzare il numero di spostamenti e di cambiamenti di direzione del braccio del disco*. Inizialmente il *braccio del disco* è posizionato sulla traccia 0 (quella più *esterna*) e si muove verso l'interno. Quando viene fatta una richiesta di accesso e il *disco* è *occupato* si confronta il numero della *traccia corrente (POS)* con quella richiesta (**dest**). Se il disco è libero viene immediatamente eseguita la richiesta, qualunque essa sia.

Nel disco la *traccia più esterna* è la *traccia 0*, mentre la *più interna* è la *traccia N*.



Sono possibili tre casi:

1. Se si ha che `POS < dest`, allora il processo viene inserito (con l'indicazione della traccia richiesta) in una coda associata alla direzione *SU* (cioè verso *l'interno*)
2. Se si ha che `POS > dest`, allora il processo viene inserito (con l'indicazione della traccia richiesta) in una coda associata alla direzione *GIÙ* (cioè verso *l'esterno*)
3. Se si ha che `POS = dest`, allora la richiesta viene accodata nella direzione corrente (che può essere *SU* o *GIÙ*, a seconda di dove si trova il disco)

Nella **politica di risveglio**, viene servita la richiesta più prossima nella coda relativa alla direzione corrente: se questa coda è vuota, allora viene cambiata direzione e viene servito il processo nella coda relativa alla nuova direzione e con la richiesta più prossima.

Variabili del Monitor

Le variabili locali del monitor comprendono:

- `POS` → posizione corrente del braccio (numero della traccia corrente)
- `DIR` → direzione corrente del braccio (SU o GIU)
- `OCCUPATO` → lo stato del disco

Sono inoltre definite **due variabili condizione**, `DIR_SU` e `DIR_GIU`, per la sospensione dei processi

```

type traccia = 0 .. N;
type GESTORE_DISCO = monitor;
var POS: traccia;
    DIR: (giu, su); // enumerazione
    OCCUPATO: boolean;
    DIR_GIU, DIR_SU: condition;

procedure entry RICHIESTA(dest: traccia);
begin
    if OCCUPATO then
        if (POS < dest) or (POS = dest and DIR = su)
            then DIR_SU.WAIT(dest);
        else DIR_GIU.WAIT(N - dest);
    OCCUPATO := true; // caso in cui siamo stati risvegliati
    POS := dest;
end;

procedure entry RILASCIO;
begin
    OCCUPATO := false;
    if DIR = su then
        if DIR_SU.QUEUE then DIR_SU.SIGNAL;
        else begin
            DIR = giu; DIR_GIU.SIGNAL; // cambio direzione
            end;
    else
        if DIR_GIU.QUEUE
            then DIR_GIU.SIGNAL;
        else begin
            DIR = su; DIR_SU.SIGNAL;
            end;
    end;
begin {inizializzazione}
    POS := 0; DIR := su; OCCUPATO := false;
    // oppure POS := N; DIR := GIU
end;

```

Supposto di aver definito

```
var DISCO: GESTORE_DISCO;
```

Allora i processi che vogliono accedere al disco dovranno avere questo schema:

```

// PROCESSO Pi
var t: traccia;
begin
    <calcolo di t>
    DISCO . Richiesta(t);
    <uso della traccia richiesta>
    DISCO . Rilascio;
end;

```

t corrisponderà a `dest` (a livello generale).

Chiamate Innestate di Monitor

Supponiamo di avere *due monitor*, un monitor *A* e un monitor *B*, e supponiamo che una procedura *entry Proc1* del monitor *A* *invochi una procedura entry Proc3* del monitor *B*. Nel caso che la procedura *Proc3* di *B* abbia come *effetto di bloccare il processo corrente*, il monitor *B* viene *liberato ma il monitor A no* → possibilità di **blocco critico (DEADLOCK)**

Soluzioni

- evitare le *chiamate innestate di monitor*, così come le *chiamate ricorsive*
- in caso di sospensione di un processo, rilasciare tutti monitor implicati nelle chiamate → occorre garantire la consistenza delle strutture dati di tutti i monitor

2.7 Costrutti di Sincronizzazione in Ambiente Locale

Nel modello ad ambiente locale si può avere *solo un tipo di interazione* fra processi: la **COOPERAZIONE**. In questo modello *non è presente COMPETIZIONE dato che non ci sono risorse in comune* (e quindi problemi di mutua esclusione). Ogni processo opera sui dati propri e il principale strumento di sincronizzazione è lo **scambio di messaggi** (lo scambio di segnali è il caso degenero) → si utilizza un *sistema di comunicazione fra processi* chiamato **IPC - Inter Process Communication**.

Il modello ad ambiente globale è il *primo ad essere stato introdotto*, e rappresenta la naturale **astrazione** del funzionamento di un **sistema multiprogrammato a memoria comune** che si appoggia su una architettura:

- **monoprocesso con memoria comune**
- **multiprocessore con memoria comune** → ad ogni processore può essere associata anche una memoria privata, MA OGNI interazione avviene tramite risorse contenute nella memoria comune

Con il passare del tempo si cominciarono ad usare anche *architetture distribuite* e quindi è stato introdotto il **Modello ad Ambiente Locale**. Questa architettura è simile alla precedente: sono sempre presenti più CPU ciascuna con una memoria privata, ma *senza una memoria comune*, dotata semplicemente di una *rete di interconnessione*.

Il Modello ad Ambiente Locale pertanto rappresenta la naturale **astrazione** del funzionamento di un **sistema multiprogrammato privo di memoria comune**. A livello di programmazione si deve essere in grado di risolvere ogni problema con ciascuno dei due modelli introdotti. I modelli hanno la **stessa capacità espressiva**. Il modello a scambio di messaggi può essere utilizzato e realizzato sia in presenza di memoria comune che in assenza.

In questo modello **non esiste più il concetto di risorsa** accessibile direttamente (e quindi condivisa) dai processi.

Sono possibili due schemi:

1. Alla risorsa è associato un *processo servitore* → un processo dedicato a servirla. Ogni **processo CLIENTE** che ha bisogno di accedere alla risorsa chiede, *tramite scambi di messaggi*, al servitore di farlo. I processi non vedono le risorse, ma vedono **i servizi** per accedervi.

Esempio:

Risorsa STAMPANTE → processo GESTORE

Generico processo Pi → manda un messaggio di richiesta di STAMPA

2. La risorsa (*logica*) viene *passata* da un processo ad un altro *sotto forma di messaggio*

Esempio:

Problema Produttori-Consumatori.

Non sarà più presente *un buffer condiviso*, ma la comunicazione tra processi avverrà attraverso **due primitive di comunicazione: SEND e RECEIVE**. Attraverso l'utilizzo di queste due funzioni, si va a creare un **canale logico di comunicazione** fra i processi che vogliono comunicare.

2.7.1 Caratteristiche di un Canale Logico

Le caratteristiche di un canale logico dipendono dalle scelte implementative del S.O, e rispondono alle seguenti domande:

1. Come viene stabilito un canale?
2. Un canale può essere associato a più di una coppia di processi sender-receiver?
3. Un canale è unidirezionale o bidirezionale?
4. Qual è la capacità di un canale?
5. Che dimensione hanno i messaggi? Un canale può "trasportare" messaggi di dimensione fissa o variabile?

Possibili Scelte Implementative

1. Designazione della coppia di processi comunicanti
RISPONDE A: 1, 2, 3
2. Uso di BUFFERIZZAZIONE
RISPONDE A: 4
3. Lunghezza dei messaggi
RISPONDE A: 5

Queste **scelte** sono **ortogonali**. Ovvero, sono *indipendenti l'una dall'altra*.

Designazione della Coppia di Processi Comunicanti

CASO A: Identificazione Diretta

Comunicazione Simmetrica

Comunicazione SIMMETRICA → i processi *SENDER* e *RECEIVER* devono *indicare esplicitamente con chi vogliono comunicare*:

- il processo *SENDER* DEVE indicare esplicitamente il processo *RECEIVER* cui vuole spedire un messaggio
- il processo *RECEIVER* DEVE indicare esplicitamente il processo *SENDER* da cui vuole ricevere un messaggio

Le primitive di comunicazione diventano:

- `SEND (R, message)` → spedisce il messaggio '*message*' al processo R
- `RECEIVE (S, message)` → riceve il messaggio '*message*' dal processo S

Proprietà del Canale

1. Il canale viene *stabilito automaticamente* (nel momento in cui si fanno le operazioni di *SEND* e *RECEIVE*) fra la coppia di processi che desidera comunicare
2. Il canale è associato solo alla coppia S-R
3. Il canale è **unidirezionale** → se i due processi comunicanti invertono i propri ruoli, viene creato un nuovo canale

Un processo che è *SENDER* nei confronti di un altro processo può essere, in altri momenti della sua vita, *SENDER* o *RECEIVER* per altri processi. Lo schema di comunicazione è detto ***uno-a-uno***.

Svantaggio: Non è possibile realizzare un processo *gestore di una risorsa* → si dovrebbero conoscere i nomi (identificatori) di tutti i clienti. A questa comunicazione, viene preferita quella ***asimmetrica***.

Comunicazione Asimmetrica

Comunicazione ASIMMETRICA → identificazione diretta solo per il *RECEIVER*

Le primitive di comunicazione diventano:

- `SEND (R, message)` → spedisci il messaggio *message* al processo *R*
- `RECEIVE (ID, message)` → ricevi il messaggio *message* da qualunque processo l'identità del processo MITTENTE viene riportata nel valore del parametro di uscita *ID*. Il *Receiver* non conosce l'identità del mittente del messaggio, ma viene accettato purché questa venga specificata come parametro *ID*.

Lo schema di comunicazione viene detto in questo caso ***molti-a-uno***. Utile per implementare un processo *SERVITORE* di una risorsa che *riceve richieste* da molti altri processi *CLIENTI*:

- ogni *CLIENTE* specifica il *SERVITORE* a cui spedire la richiesta
- il *SERVITORE* non sa (a priori) da chi gli può arrivare la prossima richiesta

Altre Comunicazioni Asimmetriche

- ***uno-a-molti***
- ***molti-a-molti*** → un generico cliente che ha bisogno di un servizio non manda la richiesta ad uno specifico servitore, ma ad un insieme di fornitori.

Le primitive di comunicazione diventano:

- `SEND (set-of-receivers, message)` → spedisci il messaggio *message* all'*insieme di servitori* specificato. La singola primitiva attua la spedizione *in contemporanea a tutti i receiver*.
- `RECEIVE (ID, message)` → ricevi il messaggio *message* da qualunque processo l'identità del processo MITTENTE viene riportata dal valore di *ID*

La comunicazione *molti-a-molti* viene chiamata **Protocollo di MULTICAST**, e può essere ampliata come **Protocollo di BROADCAST**.

SVANTAGGIO → *modularità limitata*

Se cambia il nome di un processo implicato in una comunicazione tutti i processi che sono interessati devono essere avvertiti, in quanto devono utilizzare un identificatore differente.

CASO B: Identificazione Indiretta

I processi *SENDER* e *RECEIVER* non devono indicare con chi vogliono comunicare. La comunicazione avviene attraverso un *elemento intermedio*: ***mailbox*** (o porta).

- il processo *SENDER* spedisce un messaggio ad una *mailbox*
- il processo *RECEIVER* riceve un messaggio da una *mailbox*

Le primitive di comunicazione diventano:

- `SEND (mbx, message)` → spedisci il messaggio *message* alla *mailbox* *mbx*
- `RECEIVE (mbx, message)` → ricevi il messaggio *message* dalla *mailbox* *mbx*

Nota: Se l'identità del mittente è importante andrà inserita come parte del messaggio.

Proprietà del canale

1. il canale viene stabilito solo se la coppia di processi utilizza la stessa mailbox canale <====> mailbox
2. il canale può essere associato a più coppie di processi. Viceversa, fra una coppia di processi ci possono essere più canali
3. il canale può essere unidirezionale o bidirezionale

Schemi di Comunicazione

- uno-a-uno
- uno-a-molti
- molti-a-uno (tipica implementazione del gestore della risorsa)
- molti-a-molti

Caso Particolare: Gestore di una Risorsa

Ogni processo viene creato con la propria **mailbox "privata"**. Solo questo processo può ricevere da questa mailbox, mentre ogni altro processo che vuole comunicare con quel processo spedisce un messaggio a questa mailbox.

Esempio:

Gestione di una stampante da parte di un processo *P1*.

Uso di BUFFERIZZAZIONE

CASO A: Capacità = 0

In questo caso il *buffer di comunicazione* è assente. Il canale di comunicazione *non può accodare* messaggi. Il processo **SENDER deve aspettare** un **RECEIVER** che dall'altra parte faccia la ricezione del messaggio. La primitiva **SEND** è in questo caso una **primitiva SINCRONA** (o **bloccante**). Se dall'altra parte il **RECEIVER** sta aspettando il messaggio, e pertanto arriva su una **RECEIVE bloccante** → **rendez-vous**.

A livello implementativo, una **RECEIVE** viene *sempre definita bloccante*.

Esempio:

Il **SENDER** invia un messaggio. Non essendoci subito pronto il **RECEIVER**, il processo **S** viene sospeso. Quando il **RECEIVER** effettua la *receive* il processo **S** viene rimesso in esecuzione.

Vantaggi

- facilità di realizzazione
- il mittente dopo la **SEND** è sicuro che il **RECEIVER** ha ricevuto il messaggio

Svantaggi

- minore grado di parallelismo
- se il **SENDER** o il **RECEIVER** non arrivano al **rendez-vous** → blocco

Soluzioni per il Blocco

- Introduzione di un time-out nelle primitive di comunicazione (con identificazione diretta o indiretta). Lo stato sospeso viene sospeso per un massimo di tempo in termini di *s/ms*.

```
send(X, message, time-out) // dove X può essere R oppure Mbx
```

se dopo il tempo specificato il messaggio non è arrivato, il processo mittente viene sbloccato

```
receive(Y, message, time-out) // dove Y può essere S, ID oppure Mbx
```

se dopo il tempo specificato il messaggio non è arrivato, il processo ricevente viene sbloccato

- Introduzione di **asincronicità** nelle primitive di comunicazione
 - **buffering** per il lato **mittente**
 - **ricezione non bloccante** per il lato ricevente (si tratta di un'ulteriore primitiva di receive non bloccante)

CASO A: Capacità = N

Il canale può accodare (con politica *FIFO*) al massimo N messaggi. Il processo *SENDER*, se la *coda del canale non è piena*, può *spedire il messaggio* e poi **continuare** la propria esecuzione → la primitiva *SEND* è in questo caso una primitiva *ASINCRONA* (o non bloccante). Il processo *SENDER* *dove aspettare*, se la *coda è piena*, fino a che il (un) processo *RECEIVER* non ha effettuato la ricezione.

CASO C: Capacità = ∞

Si può considerare una capacità illimitata se il sistema può aumentare la capacità del canale per propria comodità. La capacità illimitata *non rappresenta una situazione di partenza*, ma si verifica nel caso in cui un processo *SENDER* finisce lo spazio a disposizione e ha quindi la necessità di accodare più messaggi.

Lo **schema di comunicazione** anche in questo caso è **asincrono**.

Caso di **identificazione diretta** → è il *sistema operativo* che deve garantire il *buffering* (in genere, teoricamente infinito)

Caso di **identificazione indiretta** → è la *mailbox* che ha associata la *coda dei messaggi (dimensione finita)*

Vantaggi Casi B/C

- maggiore grado di parallelismo
- nel caso capacità infinita nessuna possibilità di blocco del *SENDER*

Svantaggi Casi B/C

- difficoltà di realizzazione
- il mittente non sa se se il messaggio è stato ricevuto → se per il mittente è importante sapere se il messaggio è stato ricevuto si utilizza un *messaggio di acknowledgement*

Ricezione non bloccante lato ricevente

Un altro grado di **asincronicità** può essere introdotto dal lato *ricevente* (sia nel caso di buffering o meno dal lato mittente). La semantica della *RECEIVE* viene modificata in modo da non essere bloccante: se un messaggio è presente, allora il processo ricevente lo riceve altrimenti prosegue l'esecuzione. Questo può risultare utile nel caso di *un receiver che aspetta risposta da più servitori*, dei quali gli basta il primo che risponde.

Lunghezza dei Messaggi

CASO A: Messaggi di Dimensione Fissa D

Vantaggio

- maggiore semplicità per il sistema (*nei canali di comunicazione*). Si creano array di dimensione fissa.

Svantaggio

- messaggi corti (minori di D) → sprecano memoria
- messaggi più lunghi vanno "spacchettati" dal lato mittente in diversi messaggi di lunghezza D
→ problema di "rimpacchettamento" dal lato ricevente

CASO B: Messaggi di Dimensione Variabile

Vantaggio

- maggiore flessibilità per il programmatore

Svantaggio

- maggiore complessità per il sistema

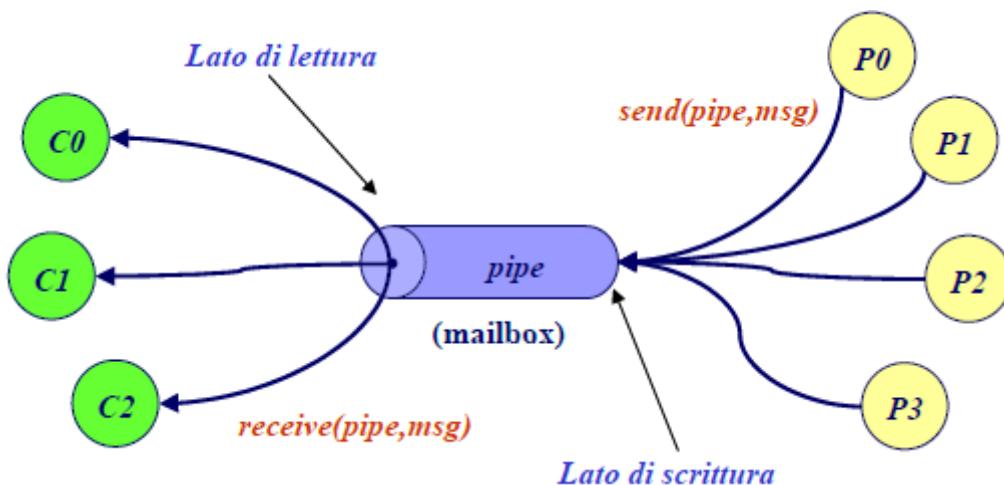
2.7.2 Interazione tra Processi UNIX

I processi UNIX normalmente non condividono la memoria. Per fare interagire questi processi è necessario utilizzare *strumenti di Inter Process Communication*.

Comunicazione tramite PIPE

Le PIPE sono un *tipo di comunicazione* che consentono l'*identificazione indiretta* (corrispondenti delle *mailbox*). Una pipe è un canale logico di comunicazione tra processi:

- **unidirezionale**: → accessibile ad un estremo in lettura ed all'altro in scrittura
- **capacità limitata**: → la pipe è in grado di gestire l'accodamento di un numero limitato di messaggi (consente bufferizzazione, *SEND* asincrona, non bloccante), gestiti in modo FIFO: il limite è stabilito dalla dimensione della pipe. La dimensione dei messaggi è variabile, ma deve essere *rispettata una dimensione minima*.
- **molti-a-molti**:
 - più processi possono spedire messaggi attraverso la stessa pipe
 - più processi possono ricevere messaggi attraverso la stessa pipe



I rispettivi nomi delle funzioni primitive di *SEND* e *RECEIVE* sono ***write*** e ***read***.

2.7.3 Implementazione di un Canale di Comunicazione

Il canale può essere realizzato in vari modi:

1. Tramite un BUS o altro collegamento hardware

Per fare comunicare *processi che risiedono su processori distinti* (*collegati tramite hardware*) in un sistema distribuito → è necessario effettuare una ***COPIA del messaggio*** dallo spazio di indirizzamento del processo mittente allo spazio di indirizzamento del processo ricevente.

2. In memoria condivisa

Per fare comunicare processi che risiedono sullo stesso processore di un sistema distribuito oppure se viene usato in sistemi a memoria comune → la ***COPIA del messaggio non è più necessaria***: il processo mittente ***trasferisce*** al processo ricevente ***l'indirizzo della zona di memoria*** dove si trova il messaggio

Svantaggio: i processi non sono più disaccoppiati cioè non operano più in AMBIENTE strettamente LOCALE → il processo mittente non può alterare il messaggio fino a che questo non è stato elaborato dal ricevente.

Vantaggio: maggiore efficienza

UNIX e LINUX realizzano le pipe in memoria condivisa in locazioni di memoria del KERNEL. Viene effettuata la copia del messaggio in una mailbox: il messaggio viene copiato dalla spazio di indirizzamento del produttore (*write su pipe*) dentro la pipe e quindi il messaggio viene poi copiato dalla pipe allo spazio di indirizzamento del consumatore.

2.7.4 Considerazioni su SEND e RECEIVE

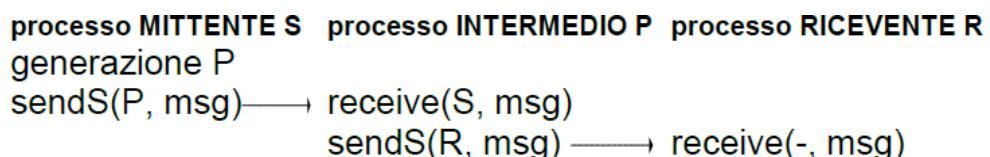
- Le operazioni *SEND* e *RECEIVE* sono operazioni primitive del S.O. e quindi è garantita la loro indivisibilità → le operazioni di *read* e *write* su una pipe avvengono *in modo atomico*.
- Le primitive di *SEND sincrona* e *asincrona* hanno la ***stessa capacità espressiva***. Se un S.O offre solo una send sincrona o asincrona si riesce sempre a realizzare quella di semantica opposta. Vediamo come, supponendo le *receive bloccanti*:

 - Send sincrona con send asincrona (che indichiamo con *sendA* per maggior chiarezza; non importa se usiamo identificazione diretta o indiretta):



Dopo aver inviato la *send sincrona*, il mittente si blocca in attesa dell'*acknowledgement*.

- Send asincrona con send sincrona (che indichiamo con *sendS* per maggior chiarezza; usiamo identificazione diretta per semplicità); è necessario introdurre un'entità intermedia: ***processo P***



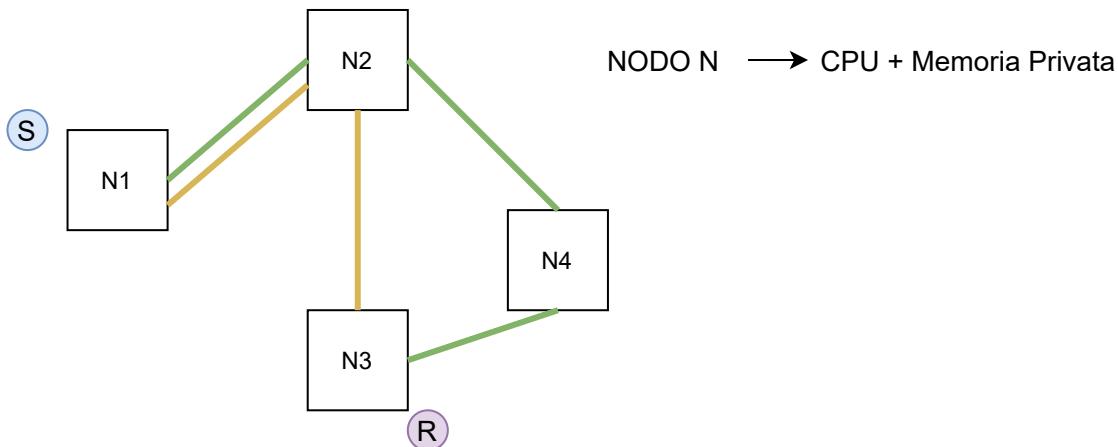
Sarà il processo intermedio, eventualmente, a subire il blocco sulla *send*.

2.7.5 Problema Ordinamento dei Messaggi

Viene garantito l'ordinamento dei messaggi?

1. solo fra la stessa coppia di processi (*sender* e **receive**)

- comunicazione sincrona → AUTOMATICAMENTE GARANTITO. Se il processo *Mittente S* prima di ogni altra *SEND* aspetta che il *Ricevente R* riceva il messaggio, **non c'è un problema di ordinamento**.
- comunicazione asincrona → dipende dal sistema di routing
 - **statico** → percorso giallo, nessun problema di ordinamento. Il percorso è sempre lo stesso.
 - **dinamico** → percorso giallo e verde, possono esserci problemi di ordinamento. I messaggi possono arrivare a *R* in ordine diverso da come sono stati inviati da *S*. Se l'ordine è importante, a livello implementativo è necessario aggiungere *un numero di ordine nei messaggi*.



Se è importante per l'applicazione e il sistema di routing non lo garantisce

- si DEVE inserire nel messaggio un numero d'ordine
- il ricevente deve crearsi uno spazio locale di memorizzazione dei messaggi ricevuti "fuori" ordine

2. fra tutti i processi

problema di nozione di tempo globale e ordinamento totale degli eventi

2.7.6 Possibili Condizioni di Errore

Terminazione di un Processo

Un processo SENDER o RECEIVER può *terminare la propria esecuzione* (per un errore o naturalmente) e *lasciare bloccato un processo in comunicazione*.

Soluzioni

- comunicazione sincrone con *time-out*
- comunicazioni asincrone
- intervento del sistema che può anche *terminare il processo bloccato* o semplicemente *notificargli l'accaduto*

Comportamento di UNIX/LINUX: in caso di pipe senza scrittore/produttore la read (receive), normalmente bloccante, si tramuta in *non bloccante* e torna 0, sbloccando il processo lettore/ricevente. In caso di pipe senza lettore/consumatore, nonostante la *send sia asincrona* (e quindi non toccata da questa evenienza), la pipe ha comunque *capacità limitata*, pertanto è inutile che lo scrittore continui a scrivere sulla pipe: il kernel invia il segnale *SIGPIPE* al processo scrittore/produttore, provocandone la morte. Insieme a *SIGPIPE* può essere inviata una *specificazione* che permette al processo di passare a fare altro, anzichè morire.

Come si risolve il problema in caso di uso di schemi di sincronizzazione in ambiente globale? È possibile solo *un intervento del sistema* che può anche *terminare il processo bloccato o semplicemente notificargli l'accaduto*. Si può inoltre fare *ritornare alla WAIT* un valore in modo che si sappia se questa ha avuto successo (risveglio in seguito a una *SIGNAL*) o se è stata semplicemente terminata dal S.O in seguito alla terminazione (improvvisa o non) del processo che doveva effettuare la *SIGNAL*.

Messaggi Persi e Messaggi "Corrotti"

Solo in caso di Sistemi Distribuiti. Nel caso di sistema *monoprocessore* o *multiprocessore a memoria comune* se si hanno problemi nei banchi di memoria il problema è un *problema di Gestione della Memoria*. Nel caso dei Sistemi distribuiti il problema è relativo al **sottosistema di comunicazione**.

Soluzioni

- Il **sistema operativo** è **responsabile di accorgersi** di questi errori e di **effettuare la ritrasmissione** → bisogna che abbia *una copia di tutti i messaggi* (schema molto pesante per il sistema)
- Il **mittente** è **responsabile** di accorgersi di questi errori e di **effettuare la ritrasmissione**, se vuole. Possibile solo:
 - **comunicazioni sincrone** con *time-out*
 - **comunicazioni asincrone** con *messaggio di ACK da parte del ricevente*
- Soluzione intermedia tra le due precedenti: il **sistema operativo è responsabile di accorgersi** di questi errori e notifica al **mittente** l'accaduto il mittente può, se vuole, **effettuare la ritrasmissione**

2.7.7 Esempi di Utilizzo di Messaggi in Ambiente Locale

Soluzione del Problema Produttori e Consumatori

Ipotesi:

- identificazione → indiretta (uso di *mailbox*)
- buffering → sì
- lunghezza messaggi → variabile

Schema asincrono di comunicazione: *SEND asincrona - RECEIVE sincrona*

In questo caso si sfrutta la potenzialità dei messaggi di *trasportare dati* (informazioni)

- **Caso A) buffering → limitato a N**

È possibile realizzare *solo una soluzione* al problema Produttori/Consumatori analoga a quella vista in ambiente globale con un buffer, di dimensione N, in memoria comune protetto da semafori

- **Caso B) buffering → con capacità illimitata**

È possibile realizzare una *soluzione* al problema Produttori/Consumatori *più ampia* di quella vista in ambiente globale che prevedeva un buffer, di dimensione N, perché è come se potessimo avere un buffer di dimensione infinita

Vediamo quale può essere la soluzione in questi due casi.

Si definisce *una sola mailbox* → **BUFFER** su cui si mandano e si prelevano le informazioni. Si sfrutta la capacità limitata o illimitata della mailbox per simulare il buffer di dimensione N o infinita.

```
// Struct rappresentante un MESSAGGIO
typedef struct {
    prcIdent id; /* identità processi mittenti */
    union { ... } dati /* record variante */
} Message;
```

```
// Generico Processo PRODUTTOREI
{
    Message msg; // variabile locale, privata
    Information info;
    while (true) {
        <produzione messaggio info>
        msg.dati = info;
        // msg.id = id; // se è importante può mettere l'ID
        send(BUFFER, msg);
    }
}
```

```
// Generico Processo CONSUMATOREI
{
    Message msg; // variabile locale, privata
    Information info;
    while (true) {
        receive(BUFFER, msg);
        info = msg.dati;
        <consumo messaggio>
    }
}
```

- **Caso C) buffering → con capacità illimitata**

In questo caso vogliamo realizzare una soluzione al problema Produttori/Consumatori analoga a quella vista in ambiente globale che prevedeva un buffer, di dimensione N. A livello si S.O si ha un buffer con capacità illimitata, ma i Produttori/Consumatori hanno a disposizione per comunicare un buffer di capacità illimitata. Si parte da una capacità illimitata a livello di bufferizzazione che si vuole *andare a limitare*.

Si definiscono due MAILBOX:

- DATIPROD → da cui i Produttori prelevano un **token** per poter inserire nel buffer
- DATICONS → su cui si mandano e si prelevano le informazioni (buffer precedente)

DATIPROD viene *riempita* inizialmente di *un certo numero di messaggi vuoti* in numero pari alla dimensione del buffer → *tanti token quanti sono i possibili elementi del buffer*. Ciò garantisce che **nell'applicazione** ci siano **al massimo N messaggi**.

```
// Generico Processo PRODUTTOREj
{
    Message msg;
    Information info;
    while (true) {
        <produzione messaggio info>
        receive(DATIPROD, msg);
        msg.dati = info; // il token viene trasformato in messaggio
        send(DATICONs, msg);
    }
}
```

```
// Generico Processo CONSUMATOREj
{
    Message msg;
    Information info;
    while (true) {
        receive(DATICONs, msg);
        info = msg.dati;
        send(DATIPROD, msg);
        <consumo messaggio>
    }
}
```

Realizzazione di un Semaforo

In un *Sistema ad Ambiente Locale puro* non si hanno processi che condividono memoria né alcun tipo di risorse, ma l'implementazione effettiva di un modello in un sistema operativo può portare ad avere eccezioni: una necessità del genere può manifestarsi ad esempio in UNIX/LINUX. I processi non possono condividere zone di memoria, ma una grossa *eccezione* è rappresentata dal **File System**, una **macro-risorsa** (costituita da file e directories) condivisa da tutti i processi. Poiché questi sistemi operativi non adottano nessun politica per garantire un **accesso corretto** (tipo lettori-scrittori) è **compito del programmatore** attuarla utilizzando ad esempio una implementazione dei *semafori*.

Prima Ipotesi

- identificazione → indiretta (uso di mailbox)
- buffering → sì

Un messaggio vuoto corrisponde ad un invio di un SEGNALE. Una **sezione critica** viene **protetta** da un meccanismo di **RECEIVE** e **SEND su una mailbox** (semaforo di mutua esclusione).

Inizialmente, bisogna creare la mailbox e poi riempirla con un messaggio (vuoto) → **NULL**

```
create-mailbox (mutex); // Creazione della mailbox Mutex
send(mutex, NULL); // Deposito di un token (messaggio vuoto)
```

Quindi il codice di un generico processo Pi che vuole accedere alla risorsa in mutua esclusione è:

```

Message msg;
{
    ...
    receive(mutex, msg); /* continua solo se c'è un messaggio e lo estrae */
    <sezione critica>
    send(mutex, msg); /* restituisce il token per consentire ad altri l'entrata
*/
    ...
}

```

- Il messaggio vuoto *NULL* rappresenta una sorta di "gettone di entrata" (*TOKEN*) che consente l'ingresso di un processo per volta
- SEND e RECEIVE sono primitive (INDIVISIBILI)
- Semaforo binario → Capacità = 1

Per realizzare un **Semaforo Generalizzato** si spedisce inizialmente un certo numero di messaggi *NULL* pari al valore che deve avere il semaforo all'inizio.

Seconda Ipotesi

- identificazione → diretta asimmetrica
- buffering → no

In questo c'è bisogno di *un processo che rappresenta il semaforo* e una **sezione critica** viene **protetta** da un meccanismo di **SEND verso questo processo**. Inizialmente, bisogna creare il processo semaforo PS:

```

while (true) {
    prcIdent P;
    receive(id, message);
    P = id; // forzo la ricezione da uno specifico processo
    receive(P, message); // epilogo s.c. può tornare a ricevere altri msg.
}

```

Quindi il codice di un generico processo *Pi* che vuole accedere alla risorsa in mutua esclusione è:

```

Message msg; /* inizializzato a NULL */
{ ...
    send(PS, msg); /* continua solo se PS riceve
il messaggio cioè se è bloccato sulla prima
receive*/
    <sezione critica>
    send(PS, msg); /*questa send serve solo per
fare in modo che il processo semaforo PS si
sblocchi dalla seconda receive */
    ...
}

```

Non c'è bufferizzazione, pertanto la prima *send* effettuata dal processo *Pi* è bloccante. Il processo procede solo se il processo riceve il messaggio.

2.8 Linguaggi di Programmazione Concorrenti

Oltre al linguaggio di programmazione concorrente *Concurrent Pascal* che si basa sul modello dei processi *GLOBALE* (e che ha introdotto il MONITOR per la sincronizzazione), ci sono altri linguaggi di programmazione concorrente che invece hanno deciso di basarsi sul modello dei processi *LOCALE*.

2.8.1 Linguaggio CSP

Scelte a livello di scambio di messaggi:

- identificazione diretta e simmetrica
- *SEND* con schema sincrono (no buffering)

Processo Mittente S Processo Ricevente R

R ! message S ? message

- **!** per inviare il messaggio
- **?** per ricevere il messaggio

2.8.2 Linguaggio OCCAM

Definito inizialmente come linguaggio 'Assembler', anche se di alto livello, per Transputer (tentativo europeo di progettare 'microprocessori', chip per un'architettura parallela). Derivato da CSP (come sintassi).

Scelte a livello di scambio di messaggi:

- identificazione indiretta (uso di *mailbox* chiamate channel), quella diretta risultava troppo limitante
- *SEND* con schema sincrono (no buffering)

CH: channel

Processo Mittente S Processo Ricevente R

CH ! message CH ? message

2.8.3 Linguaggio ADA

ADA, prima programmatrice donna. Il linguaggio nasce da un progetto del *Department of Defense (DoD)* Americano. Per alcuni versi mescola modello locale e globale (per il quale però non fornisce meccanismi di sincronizzazione).

Scelte a livello di scambio di messaggi:

- identificazione diretta e *asimmetrica*
- *SEND* con schema sincrono (no buffering)

Processo Mittente S

```
task S is
end S;

task body S is
...
  R.servizio(...);
...
end R;
```

Processo Ricevente R

```
task R is
  entry servizio(...);
end R;
task body R is
...
  accept servizio (...);
...
end R;
```

Il **rendez-vous** si dice **esteso** perché non solo il processo mittente S e il processo ricevente R rimangono sincronizzati fino a che R non ha ricevuto il messaggio, ma anche fino a che R non ha svolto la procedura/servizio richiesto da S. Il mittente S infatti, più che spedire semplicemente un messaggio, invia la *richiesta di esecuzione di una procedura che non fa parte del suo spazio di*

indirizzamento. Il Ricevente R deve accettare il messaggio ed eseguirne l'operazione.

2.9 Punto di Vista ESTERNO

2.9.1 Utente o Programmatore di Sistema

Le informazioni che devono essere note a un programmatore di sistema sono più dettagliate di quelle note all'utente. (vedere *Schema 2.3 Nucleo: Riepilogo Punti di Vista* per analizzarne meglio il dettaglio)

Partendo da un *problema non sequenziale*, si scomponete il processo (non sequenziale) derivante dalla sua soluzione in un *insieme di processi sequenziali*. Questi processi, derivando da questo scomposizione hanno bisogno di un certo **tipo di interazione** per determinare la soluzione del problema:

- *COMPETIZIONE* e *COOPERAZIONE* → Ambiente Globale
- *COOPERAZIONE* → Ambiente Locale

Perché l'interazione tra i processi possa avvenire, sono necessari degli **strumenti di sincronizzazione**:

- *SEMAFORI* (meccanismi di basso livello) o *MONITOR* → Ambiente Globale
- *SCAMBIO DI MESSAGGI* (messaggi di contenuto nullo se i processi devono scambiarsi segnali)
→ Ambiente Locale

3.1 Punto di Vista INTERNO

3.1.1 Sistema Operativo

NUCLEO → *primo livello di astrazione* del Sistema Operativo

Il nucleo dovrebbe, per quanto possibile, *contenere solo meccanismi* consentendo a livello dei processi di utilizzare tali meccanismi *per la realizzazione di politiche di gestione*, diverse a seconda del tipo di applicazione.

3.2 Progettazione del Nucleo di un Sistema Operativo Multiprogrammato

Ipotesi: Architettura Monoprocesso

Il nucleo deve fornire il *supporto* per:

1. Processi Sequenziali che possono essere in vari stati:
 - 1 Processo in Esecuzione (perché monoprocesso) → *RUNNING*
 - In una coda di processi pronti → *READY QUEUE*. Il loro stato di esecuzione viene salvato in un *Descrittore di Processo* e sono mandati in esecuzione attraverso specifici *ALGORITMI DI SCHEDULING*.
2. Meccanismi di Sincronizzazione, avendo prima definito il *modello di riferimento*:
 - Modello ad Ambiente Globale
 - *SEMAFORI* con *coda dei processi sospesi*
 - *OPERAZIONI* sui semafori (*Inizializzazione, WAIT e SIGNAL*)
 - Modello ad Ambiente Locale
 - *CANALI DI COMUNICAZIONE* con *coda dei processi sospesi*
 - *OPERAZIONI (SEND e RECEIVE)*

In entrambi i casi le operazioni di sincronizzazione (sia sui semafori che sui canali) sono *PRIMITIVE*. Ogni volta che un processo esegue una di queste operazioni si transita da:

STATO UTENTE* ↔ *STATO SUPERVISORE (SISTEMA)

La *visione complessiva* che deve avere il nucleo è quella dello ***stato globale del sistema***, ovvero lo stato di tutti i processi e, nel caso del modello ad Ambiente Globale, di tutte le risorse (occupate/disponibili).

3.3 Progettazione delle Strutture Dati del Nucleo

1. **Registro che identifica il processo in esecuzione** (*RUNNING*). Il processo running può essere o meno mantenuto nella *coda dei processi pronti*. Il registro non è necessariamente hardware.
2. Un **DESCRITTORE** per ogni processo → **Process Control Block (PCB)**
3. Una **coda dei processi pronti** (*READY QUEUE*), a livello logico. A livello implementativo possono esserci più code.

4. Una **coda dei processi sospesi** (possono anche essere più code)

- *code eventi esterni*
- *code semafori* (in Ambiente Globale)
- *code canali* (in Ambiente Locale)

Le code, sia quella dei processi pronti che quelle dei processi sospesi, possono essere gestite in modalità *FIFO*. Se a livello di progetto, si associa una *PRIORITÀ ai processi* che influenza l'algoritmo di scheduling, queste possono essere gestite tramite essa.

Bisogna quindi decidere se la READY QUEUE è:

- Realizzata con *una sola coda con i descrittori accodati in ordine di priorità* → nel momento in cui un processo può essere messo in stato di *READY*, deve essere messo in coda nella posizione corrispondente al suo *grado di priorità*, cosa non semplice.
- Se è realizzata con *più code di descrittori* → *una per ogni priorità*

Oltre che stabilire il *range di valori* che può assumere la priorità e se un valore basso significa priorità alta o bassa, è bene distinguere tra priorità:

- statica
- dinamica

3.3.1 Descrittore di Processo

Un descrittore di processo deve riportare le seguenti informazioni:

- **Nome** del processo, solitamente un *numero identificativo (PID)* per identificare univocamente un descrittore.
- **Stato** del processo; in una ipotesi di minima:
 - pronto
 - bloccato
 - in esecuzione (solo se il PCB resta nella coda di processi pronti, altrimenti basta un riferimento *RUNNING* nel registro).
- **PROGRAM COUNTER** → indirizzo della prossima istruzione da eseguire. Serve per quando il processo riprende la propria esecuzione dallo stato di *pronto* o *bloccato*.
- **Registri della CPU** → serve sempre quando un processo riprende la propria esecuzione. Serve per riprendere l'esecuzione in modo corretto. A seconda dell'hardware sottostante si possono trovare le seguenti tipologie di registri:
 - accumulatori
 - registri indice
 - registri generali
- **Informazioni di Scheduling:**
 - priorità
 - puntatore al prossimo processo in coda → se la lista presenta un solo elemento al puntatore sarà assegnato il valore *NULL*.
- **Informazioni di Accounting**
 - tempo di CPU usato
 - limiti di tempo (correlati agli algoritmi di scheduling in *time sharing*)
- **Informazioni sullo Stato di I/O**
 - richieste non soddisfatte di I/O
 - dispositivi assegnati
 - lista file aperti

- **Informazioni per la Gestione della Memoria**

Nel momento in cui un processo viene creato nel S.O (e quindi si lancia un programma in esecuzione), deve essere creato il suo descrittore. A livello progettuale si hanno due scelte.

1. Quando ho *necessità di avere un nuovo processo attivo nel sistema* si crea anche una memoria (*struttura dati*), come quella appena specificata nell'elenco sopraffatto. Il tempo che il sistema impiega ad allocare questa memoria viene denominato **tempo di overhead**. In questo caso, dopo la creazione del processo, è necessario un tempo supplementare per *creare il descrittore di processo*.
2. Allocazione di una **Pool di Descrittori Liberi**. Il nucleo in questo modo mantiene una coda di *descrittori disponibili* per non perdere in seguito del tempo ad allocarli. Questi descrittori vengono mano a mano associati ai processi creati e tornano disponibili alla loro distruzione. Nel caso non ci siano descrittori disponibili, il nucleo può segnalare un errore all'utente (UNIX/LINUX) oppure allocarne uno nuovo.

Esempi di Descrittori di Processi

In UNIX

Il *descrittore esiste per tutta la vita del processo* e contiene tutte le informazioni importanti per il processo (sia quando è in esecuzione e sia quando è pronto o sospeso).

- **pid** (process identifier) → nome del processo
- **ppid** (parent process identifier) → nome del processo padre
- **real uid** (user identifier)
- **real gid** (group identifier)

Un processo è creato da un utente (con un UID e un GID) di cui si tiene traccia nel descrittore.

- **effective uid** (user identifier)
- **effective gid** (group identifier)

Un processo può eseguire un programma che ha il SUID e/o il SGID settato che appartiene ad un utente (con UID e GID) di cui si tiene traccia nel descrittore.

- locazioni delle aree utente e di kernel → rappresentano le *Informazioni per la Gestione della Memoria*. Lo spazio di indirizzamento in UNIX è diviso in un'area utente e un'area di codice.
- **stato** del processo
- **priorità** del processo
- **puntatore al prossimo processo in coda**

In LINUX

Linux usa due termini diversi a seconda del punto di vista:

- a) **Punto di vista del KERNEL** → *task = entità con dati e codice*
- b) **Punto di vista ESTERNO** → *processo = parte dei task eseguita in user mode*

Per quanto risulta dal codice scritto in C per la versione del *Kernel 2.0* abbiamo:

- Tabella dei Processi (statica), corrispondente al Pool a cui si è fatto riferimento precedentemente
- Stato del Processo
- Identificatore del Processo
- User ID, Effective User ID, Saved User ID
- Group ID, Effective Group ID, Saved Group ID
- Tempo di Esecuzione

- Priorità Statica
- Codice di Terminazione
- Task *Seguente* e Task *Precedente*
- Padre Originale
- Padre Attuale
- Informazioni sui File Aperti

3.4 Progettazione delle Operazioni del Nucleo

Queste operazioni sono rappresentate dalle **chiamate di sistema (system calls)** o **primitive**, che funzionano sempre in STATO SUPERVISORE. Prendendo come riferimento un processo UNIX, questo può eseguire in due modi diversi:

- "processo di utente" (**user mode**) → modo di esecuzione NORMALE
- "processo di sistema" (**kernel mode**) → modo di esecuzione con MAGGIORE VISIBILITÀ

La transizione tra i "modi" user e kernel avviene mediante l'invocazione delle primitive. L'esecuzione delle primitive avviene in **stato kernel** con una visibilità maggiore, in quanto devono andare ad agire su strutture dati non accessibili in *user mode*.

3.4.1 Operazioni Primitive

Creazione di un Processo

Parametri:

- tipo (di sistema o di utente)
- priorità
- programma da mandare in esecuzione (da cui ricavare dimensioni e quantità di memoria per DATI e CODICE)
- informazioni di protezione

Effetto:

- creazione di un nuovo descrittore (o suo recupero dal POOL di descrittori liberi), sua inizializzazione con anche allocazione della memoria che rappresenta lo spazio di indirizzamento del processo, infine transizione dello stato (**STATO = PRONTO**)
- inserimento nella coda dei processi pronti, o eventualmente nella coda di priorità se c'è gestione della priorità

Creazione in UNIX

In UNIX si utilizza la **primitiva FORK**. In questo caso tutti i parametri che servono al Kernel per effettuare la creazione sono *impliciti*. Il tipo del processo, così come la priorità, dipende dal tipo che siamo nel momento in cui effettuiamo la creazione. Unix usa lo stato *IDLE* (esplicito) fino a che la fase di inizializzazione non è terminata e poi lo stato *READY*.

Distruzione di un Processo

Questa operazione viene usata per:

- terminazione o autodistruzione
- uccisione (un processo può volerne distruggere un altro)

Parametri:

- identificatore processo

Effetto:

- recupero risorse assegnate
- eliminazione dal sistema (se è in una coda, viene estratto)
- distruzione del suo descrittore (o reinserimento nel POOL di descrittori liberi)

Distruzione in UNIX

In UNIX si utilizza la **primitiva EXIT** (per terminazione o autodistruzione) e **primitiva KILL** con *SIGKILL* (uccisione di un processo da parte di un altro processo). Unix usa lo stato *TERMINATED* e se necessario lo stato *ZOMBIE* per indicare uno stato padre in attesa della terminazione del processo figlio, in questo modo viene mantenuto il descrittore del processo figlio che può essere utile al processo padre.

Sospensione di un Processo

Usata da:

- richieste di I/O
- In *Ambiente Globale* → *WAIT* sospensiva su semaforo
- In *Ambiente Locale* → *RECEIVE* se manca messaggio, *SEND SINCRONA* se manca il rendez-vous e *SEND ASINCRONA* con canale pieno se bufferizzazione limitata

Parametri:

- identificatore processo

Effetto:

- spostamento del suo descrittore dalla coda dei processi pronti, (se come per ipotesi precedente è questa presenta il processo in esecuzione) alla coda di sospensione e transizione dello stato in (**STATO = SOSPESO**).
- selezione del prossimo processo pronto come processo *RUNNING* (secondo l'algoritmo di *SCHEDULING*) → *PROCESS SWITCHING* (o cambio di contesto)

Riattivazione di un Processo

Usata da:

- completamento di richieste I/O → gestione interrupt
- In *Ambiente Globale* → *SIGNAL* su semaforo con coda
- In *Ambiente Locale* → *RECEIVE* quando arriva il messaggio, *SEND SINCRONA* quando arriva il receiver e *SEND ASINCRONA* quando si libera spazio nel canale limitato

Parametri:

- identificatore processo

Effetto:

- spostamento del suo descrittore dalla coda di sospensione alla coda dei processi pronti. La riattivazione di un processo *non implica mai che esso passi direttamente in stato di esecuzione*. Transizione dello stato in (**STATO = PRONTO**).
- **eventuale** selezione del prossimo processo pronto come processo *RUNNING* (algoritmo di *SCHEDULING*) → *PROCESS SWITCHING*. L'arrivo di un processo nella coda dei processi pronti

può far sì che quel processo debba immediatamente andare in esecuzione. Se è già presente un processo in fase di *running* questo deve ritornare nello stato *pronto*.

Sospensione Temporizzata di un Processo

Usata per:

- autosospensione (ad esempio per uso di time-out)
- sospensione di un altro processo → collegata alla gestione di un **TIMER**

Parametri:

- identificatore processo
- durata

Effetto:

- spostamento del suo descrittore dalla coda dei processi pronti, (se come per ipotesi precedente questa presenta il processo in esecuzione) alla **coda del TIMER** e transizione dello stato (**STATO = SOSPESO**)
- selezione del prossimo processo pronto come processo RUNNING (algoritmo di SCHEDULING) → PROCESS SWITCHING

Sospensione Temporizzata in UNIX

In UNIX, si utilizza la **primitiva SLEEP** per l'autosospensione, mentre *non è prevista una primitiva che permetta di sospendere un altro processo*. Allo scadere del tempo di sospensione → riattivazione (con la stessa primitiva indicata precedentemente). La riattivazione infatti è *indipendente da qualsiasi sia stata la causa di sospensione*.

Lettura degli Attributi di un Processo

Visto che i *descrittori di un processo* fanno parte dello spazio di indirizzamento accessibile solo al nucleo, se si vogliono rendere disponibili delle informazioni ai programmati di sistema è necessario prevedere delle *primitive che vadano a leggere alcune informazioni presenti nei descrittori dei processi*.

Lettura degli Attributi in UNIX

Queste primitive in UNIX sono costituite dalle operazioni: GETPID, GETPPID, GETUID, GETGID, GETEUID, GETEGID.

Modifica degli Attributi di un Processo

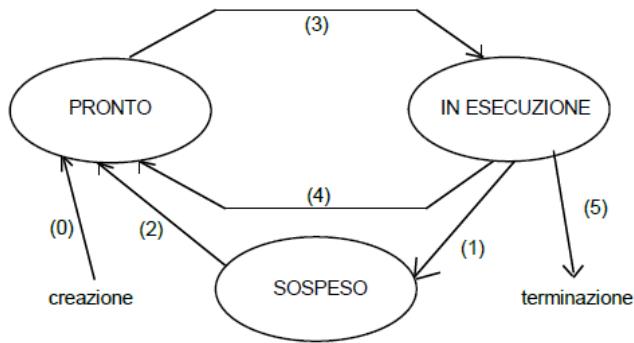
Un esempio di attributo di un processo che può essere soggetto a modifica nel corso della sua vita è la *priorità*.

Modifica degli Attributi in UNIX

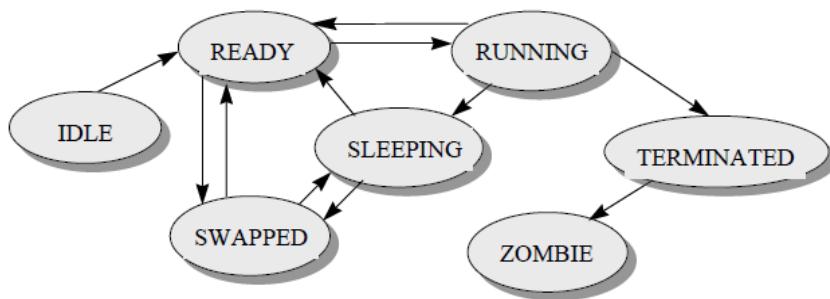
In UNIX esiste una primitiva chiamata *NICE* che permette di modificare la priorità di un processo (a livello utente), *abbassandola*.

I progettisti devono considerare che le *chiamate di sistema possano fallire* e quindi devono prevedere **CODICI DI ERRORE**.

3.4.2 Transizioni di Stato



0. **da programma a processo**
1. **da esecuzione a sospeso**
2. **da sospeso a pronto**
3. **da pronto a esecuzione** → effetto di una azione dello SCHEDULER
4. **da esecuzione a pronto** → effetto di una azione dello SCHEDULER dovuta a uso di PREEMPTION
5. **da esecuzione a terminazione**



- **SWAPPED** immagine copiata su disco
SWAP OUT → si applica preferibilmente a processi sospesi
- **ZOMBIE** terminato, ma presente in attesa di consegna del risultato al padre (che non ha ancora aspettato il figlio)

3.4.3 Cambio di Processo

Il *cambio di processo* (detto anche *cambio di contesto*, **context switching**) è necessario quando si effettua la commutazione della CPU dal processo *running* al prossimo processo che deve andare in esecuzione (in base all'algoritmo di scheduling).

N.B: Si ha a causa delle transizioni (1) e (4) e ha come effetto finale la transizione (3)

Quando il nucleo deve effettuare il cambio di processo sono necessarie le seguenti azioni:

- *salvataggio del contesto del processo in esecuzione*, dai registri macchina, *nel suo descrittore*
- *inserimento del suo descrittore nella coda appropriata* (di processi sospesi o dei processi pronti)
- *caricamento dell'identificatore del nuovo processo che deve andare in esecuzione* nel registro **RUNNING**
- *ripristino del contesto del nuovo processo running* nei registri macchina

3.4.4 Dispatcher

Il DISPATCHER è quel modulo del nucleo che passa effettivamente il controllo della CPU ai processi scelti dallo scheduler (con il quale non deve essere confuso). Quindi si occupa di effettuare:

- il *cambio di contesto* (con le azioni sopra descritte)

- il passaggio da modo *SUPERVISORE* a modo *UTENTE*
- il salto alla giusta posizione del programma utente per riattivarne l'esecuzione

3.4.5 Creazione di un Processo

Quando un processo crea un nuovo processo, ci sono **due possibilità** per quanto riguarda lo **spazio di indirizzamento** del nuovo processo:

- il processo figlio è un duplicato del processo genitore
- nel processo figlio si carica un diverso programma, da specificare a lato della creazione

Esempi nei Sistemi Operativi

UNIX

In UNIX, un nuovo processo si crea per mezzo della **primitiva FORK**. Il nuovo processo è composto da una copia dello spazio di indirizzamento del processo padre per quanto riguarda i dati (sia utente che kernel, il codice viene condiviso e i due processi eseguono lo stesso programma). Molto spesso dopo la creazione, uno dei due processi (il padre o, più normalmente, il figlio) impiega una delle primitive della famiglia **EXEC** per sostituire lo spazio di memoria del processo con un nuovo programma.

VMS

La creazione di un processo carica il programma specificato in tale processo e ne avvia l'esecuzione.

Windows

Sono previste entrambe le possibilità: la *semantica simile alla FORK*, e la *semantica simile a VMS*.

3.4.6 Tabelle per Processi in UNIX

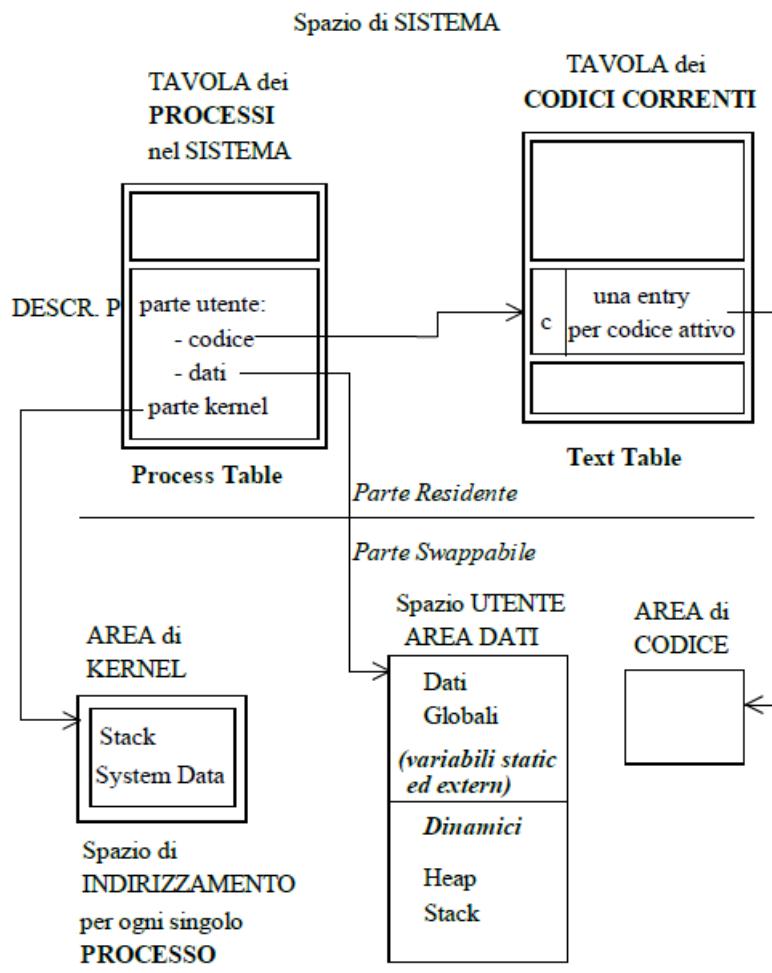


Tabelle del KERNEL - Parte Residente

Process Table

La Process Table rappresenta la *pool di descrittori* sia *liberi* che *in uso* per i processi attivi nel sistema, specificato da un campo apposito. Presenta dunque un **descrittore di processo** per **ogni processo attivo nel sistema**.

Text Table

In questa tabella è presente *un elemento per ogni codice attivo*, presentando un'ulteriore tabella per rappresentare in modo facile la condivisione di codice tra processo padre e i vari processi figli (anche solo uno).

Queste due tabelle appartengono al *kernel* e si trovano nella parte della *Parte Residente*, sempre accessibile a prescindere dalla gestione della memoria fatta sui processi utente.

Spazio di Indirizzamento dei Processi

Si trova nella *Parte Swappabile*. Se il processo non è *in stato swapped* allora in memoria centrale, nello spazio utente avremo la presenza di *aree dedicate ad ogni singolo processo*:

Area di KERNEL

In genere accessibile solo durante la esecuzione delle primitive del Sistema Operativo, nella quale troviamo:

- Stack → utilizzato per le *system calls*
- System Data → a sua volta suddivisa in:
 - *enviroment*, dove oltre all'ambiente di esecuzione ci sono anche *argc* e *argv* (parametri passati all'invocazione del *main*)
 - *user area* (o *user data*) dove troviamo la *tabella dei file aperti* e la *tabelle di trattamento dei segnali*

Area DATI

Suddivisa in:

- Dati Globali → che nel linguaggio C sono le *variabili static ed extern*
- Dati Dinamici → dati gestiti a *Stack*, come variabili definite a livello di singole istruzioni nella funzione *main* e dati gestiti a *Heap*, come le zone di memoria che si vanno ad allocare con primitive tipo *malloc()*

Area di CODICE

Ne è solo una ma più processi possono fare riferimento alla stessa tramite il loro descrittore di processo.

3.4.7 Primitive in UNIX/LINUX

Creazione di un Processo

FORK - UNIX

Se si verifica un qualsiasi errore, la fork restituisce al processo parent il valore -1.

Se la fork ha successo viene creato un processo figlio:

- inserisce una nuova entry utilizzando uno degli elementi pre-allocati nella **tabella dei processi** → descrittore per il nuovo processo con attributi ereditati dal padre: ad esempio stesso *UID* e *GID*
- l'area dati utente del processo figlio viene copiata dall'area dati utente del processo padre
- il nuovo processo esegue lo stesso codice del padre → aggiornamento contatore nell'elemento della text table, perché padre e figlio condividono lo stesso codice
- l'area kernel del processo figlio viene duplicata dall'area kernel area del processo padre → tabella dei file aperti e situazione nello stato dei segnali uguale a quella del processo padre
- inserisce il descrittore del figlio nella coda dei processi pronti

Durante le prime 4 fasi il processo è in stato *IDLE*.

FORK - LINUX (versione 2.0)

L'unica differenza a livello implementativo dalla precedente è che non viene fatta una copia iniziale dell'area utente. Questo rappresenta un'**ottimizzazione**. Nell'area del padre viene messa un flag:

- area dati marcata *write-protected*
- una scrittura genera un *page-fault* (eccezione per tentativo di accesso)

- il *kernel* fa una copia dell'area dati e dà al processo che deve scrivere il permesso di modificarla

L'ottimizzazione di non fare immediatamente la copia dati è dovuta al fatto che spesso un figlio, appena dopo essere stato creato, va a fare una *EXEC* per poter eseguire un nuovo programma, andando così a creare *una nuova area dati*.

Esecuzione di un Programma

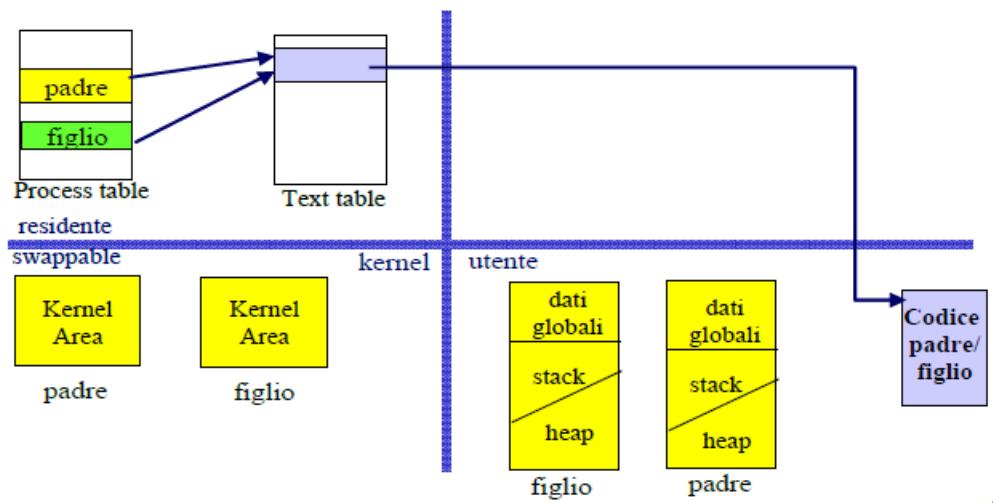
EXEC

UNIX consente di cambiare il programma che un processo sta eseguendo usando una delle primitive della famiglia *EXEC*.

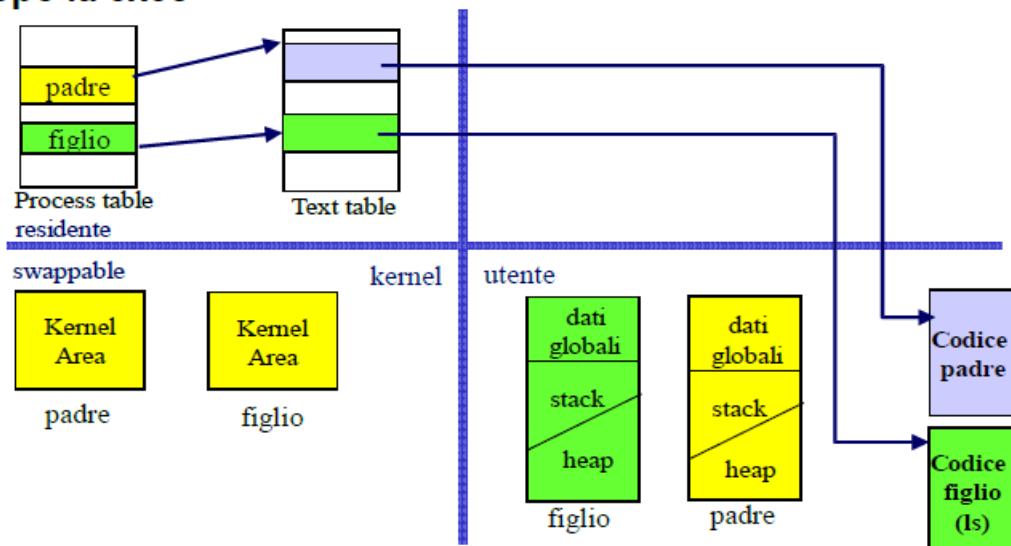
N.B.1: è questa primitiva che, eventualmente, cambia *EUID* e/o *EGID*.

N.B.2: come effetto collaterale (non visibile nella figura), viene cambiata la tabella che specifica come vengono trattati i segnali. I segnali che precedentemente erano ignorati rimangono tali così come quelli collegati alle azioni di default. Tutti i segnali che prevedevano un'azione di trattamento di quel segnale vengono tutti riportati all'azione di default: le azioni specifiche appartengono al codice, pertanto è bene svincolarle dal codice precedente.

Prima della exec



Dopo la exec



Si consideri la generazione di un certo processo. Supponiamo che il figlio abbia ottenuto come copia dal padre una *Kernel area*, un'area *Dati* costituita da *Stack* e *Heap* e ha ottenuto per condivisione il *codice del padre*. Se il figlio effettua una *EXEC*, l'area dati viene *deallocated* e ne viene allocata una nuova che serve per eseguire il nuovo programma da mandare in esecuzione. Poiché

il codice che si esegue è diverso, deve essere caricato o recuperato (se era già in uso) l'elemento della *Text table* opportuno. In questo modo il figlio non punta più all'elemento della Text table a cui puntava il padre, ma punterà al codice nuovo.

3.5 Processo VS Thread

PROCESSO → programma in esecuzione

PROGRAMMA = entità passiva

PROCESSO = entità attiva

Caratteristiche:

- Un processo è l'**unità di esecuzione** all'interno di un Sistema Operativo
- L'esecuzione di un processo è *sequenziale*: le istruzioni vengono eseguite in sequenza, secondo l'ordine specificato nel programma

Un S.O. *MULTIPROGRAMMATO* consente l'esecuzione concorrente di più processi.

Un processo è costituito da:

- Codice del programma eseguito
- Program Counter (PC)
- altri registri della CPU
- Stack (per parametri e variabili locali di funzioni/procedure)
- Dati (per variabili globali)

Quindi un processo può essere rappresentato dalla ennupla:

processo = {PC, registri, codice, dati, stack, ...}

I "..." rappresentano *dati variabili a seconda del S.O.* In *UNIX* ad esempio per ogni processo viene allocata una *Kernel area*.

3.5.1 Processi Pesanti VS Leggeri (Thread)

- processi diversi eseguono codici distinti
- processi diversi accedono a dati distinti (e quindi diversi)
- i processi (normalmente) non condividono memoria

Processi con queste caratteristiche vengono anche chiamati **Processi Pesanti** (si ispirano al **modello locale**). Unix, di base, ragiona in termini di processi pesanti.

Un **thread** (o **Processo Leggero**) è un'*unità di esecuzione* che *condivide codice e dati con altri processi leggeri* correlati. La condivisione di dati non è possibile tra due thread qualunque: i due devono appartenere allo stesso task. Un *thread* può essere definito con la ennupla:

thread = {PC, registri, stack, ...}

Queste sono le informazioni minime che consentono a un flusso di esecuzione di eseguire all'interno di una CPU. L'insieme di tutti i *thread* che *condividono codice e dati* viene detto **TASK**, che può essere definito con la ennupla:

task = {thread1, thread2, ..., threadN, codice, dati}

Per un *thread* valgono i ragionamenti fatti utilizzando il termine *processo*. Un *thread* può essere in stato di esecuzione, in stato *ready* o *suspended*. Come i processi, devono avere un descrittore che deve tenere traccia delle informazioni proprie del *thread* e che deve avere un riferimento al codice e ai dati tramite l'appartenenza al *TASK*.

N.B: In un caso degenero, un **processo pesante** equivale a un **task** con **un solo thread**.

Osservazioni

Modello dei Processi

I **Processi Pesanti** si ispirano al **modello locale**:

- **ASSENZA di condivisione di memoria**
- a differenza dei thread, un processo *NON può condividere variabili con altri processi*

I **Thread** si ispirano al **modello globale**

- **Condivisione di memoria**
- a differenza dei processi pesanti, un thread *può condividere variabili con altri thread appartenenti allo stesso task*

Costi Creazione e Context Switching (che è puro overhead)

- Per i S.O. che usano **Processi Pesanti**, il *tempo per creare un processo* e per *effettuare il process switching* risulta *molto elevato* (dipende dalla dimensione del descrittore)
- Per i S.O. che usano **Processi Leggeri**, il *tempo per creare un thread* e per *effettuare il process switching* è *molto inferiore rispetto a quello di processi pesanti* → il descrittore di un thread non contiene nessuna informazione relativa a codice e dati. *I Thread partono con un task con già allocati i Dati e il Codice.*

Ad esempio, creare un thread in *Solaris* richiede 1/30 del tempo richiesto per creare un nuovo processo.

Thread nella Quotidianità

- Browser web → Un thread per la rappresentazione sullo schermo di immagini e testo e contemporaneamente è presente un thread (che si interfaccia con un server web) per il reperimento dell'informazione in rete da visualizzare.
- Server web → Un thread per il servizio di ciascuna richiesta da parte dei client. Sarà presente un *TASK* in attesa della richiesta *HTTP/S* sulla port dedicata, e questo task una volta ricevuta la richiesta la assegna a uno specifico thread.
- LINUX → un thread per la gestione della memoria libera

3.5.2 Realizzazione di un Thread

La Realizzazione di un Thread può avvenire:

A livello di Kernel:

- il S.O. gestisce direttamente i cambi di contesto (process switching) sia tra thread dello stesso task che tra task
- il S.O. fornisce strumenti per la sincronizzazione nell'accesso a variabili comuni (secondo schemi di competizione e di cooperazione) da parte di thread diversi dello stesso task

A livello Utente

- il passaggio da un thread al successivo nello stesso task non richiede interruzioni al S.O. → l'implementazione risultante è molto efficiente
- il S.O. tratta solo processi pesanti → qualsiasi *thread che effettua una chiamata di sistema bloccante* (che si pone in attesa di I/O) *causa il blocco dell'intero processo/task*

Osservazioni

I thread a livello kernel sono supportati da tutti i S.O. attuali: Windows, Solaris, Linux, Mac OS X.

Per provare il funzionamento di una applicazione che faccia uso di processi leggeri si può:

- in **Linux**, usare la libreria *Pthread* che aderisce allo standard POSIX (implementata a livello kernel o a livello utente)
- in **Java**, o definire una sottoclasse della classe Thread o definire una classe che implementa l'interfaccia *Runnable* → più flessibile, in quanto consente di definire un thread che è sottoclasse di una classe diversa dalla classe Thread

3.5.3 Modelli di Multithreading

Molti sistemi supportano sia *kernel thread* che *user thread*: si vengono così a creare tre **differenti modelli di multithreading**:

Many-to-One

Un certo numero di *user thread* vengono mappati su un solo *kernel thread*.

Modello generalmente adottato da S.O. che non supportano kernel thread multipli.

One-to-One

Ogni *user thread* viene mappato su un *kernel thread* → thread nativi

Esempi: Windows, Linux e Solaris (dalla versione 9)

Many-to-Many

Molti user threads possono essere associati a diversi kernel threads.

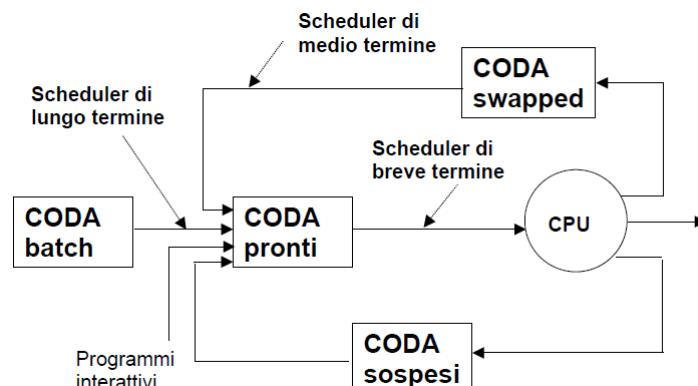
Il S.O. può creare il numero di kernel threads minore o uguale rispetto agli user threads.

3.6 Scheduling

Con il termine **Scheduling** si fa riferimento a un insieme di tecniche e di meccanismi del S.O. che stabiliscono l'ordine di esecuzione dei processi. Lo **SCHEDULER** è un modulo del S.O. e ha come obiettivo l'**ottimizzazione delle prestazioni**.

Esistono **3 tipi** di scheduler:

1. scheduler di lungo termine
2. scheduler di medio termine
3. scheduler di breve termine



Lo scheduling è una funzione fondamentale dei S.O: *si sottopongono a scheduling quasi tutte le risorse di un calcolatore*. Naturalmente la CPU è una delle risorse principali e il suo scheduling è alla base della progettazione dei S.O.

3.6.1 Tipi di Scheduler

Scheduler di Lungo Termine

Agisce sulla coda dei lavori *batch*. Dalla coda dei programmi caricati sulla *memoria secondaria* **decide chi caricare in memoria centrale**, attuando lo scheduling dalla coda dei processi batch (che contiene i **JOB**) a quella dei *processi pronti*. Questo tipo di scheduler va a determinare il **grado di multiprogrammazione** (numero di programmi in memoria centrale).

Obiettivo: Questo tipo di scheduler deve compiere la sua scelta in modo da inserire *nella coda dei processi pronti* un numero *bilanciato* di *processi CPU bound* e di *processi I/O bound*. Avendo solo processi *CPU bound*, avremmo un monopolio della CPU da parte di questi processi, mentre con solo processi *I/O bound* si hanno dei grandi intervalli di tempo con *CPU IDLE*.

La frequenza con cui agisce lo scheduler di lungo termine è più bassa di quella degli altri due tipi di scheduler.

Nei **sistemi time sharing** (come *Unix* o *Linux*) è l'utente che stabilisce direttamente il grado di multiprogrammazione e quindi lo scheduler a lungo termine non è presente: tutti i nuovi processi creati vengono direttamente caricati in memoria.

Scheduler di Medio Termine

Lo *Scheduler a Medio Termine* si incarica di gestire i processi rimossi dalla memoria centrale. In particolare, quando un processo viene sospeso effettua lo **swapping-out** (sfratto) del processo dalla memoria centrale. Il processo viene inserito nella *coda SWAPPED* e lo spazio che questo processo occupava in memoria centrale viene così salvato in *memoria di massa*. Quando un processo precedentemente sospeso diventa *pronto*, lo *riporta in memoria centrale* effettuando lo **swapping-in**.

Per subire lo *swapping-out* un processo non deve necessariamente essere sospeso, in alcuni S.O come *UNIX* è possibile arrivare allo *swapping* anche attraverso la *coda dei processi pronti*. In generale è più usato per quelli sospesi.

Scheduler di Breve Termine

Viene anche chiamato **CPU Scheduler** e rappresenta quella *parte del Sistema Operativo che decide a quale dei processi pronti* (pesanti o leggeri) presenti nel sistema *assegnare il controllo della CPU*. Questo scheduler lavora dunque sulla coda dei processi pronti, occupandosi della transizione di stato da *PRONTO* a *ESECUZIONE*.

Viene invocato *ogni volta che è necessario effettuare una commutazione di processo*, quindi ogni volta che un *EVENTO* (esterno o interno) produce un cambiamento nello stato globale del sistema. Come ad esempio:

- interrupt per completamento operazioni di I/O
- interrupt dal TIMER
- chiamate di sistema

3.6.2 Politiche VS Meccanismi

Scheduler e **Dispatcher** sono due elementi distinti del nucleo.

Lo **SCEDULER** (della CPU) decide a quale processo assegnare la CPU. Si parla spesso di *algoritmi, strategie di scelta* → **POLITICA**

Dopo questa decisione, il **DISPATCHER** passa effettivamente il controllo della CPU al processo
→ **MECCANISMO**

3.6.3 Valutazione delle Prestazioni

I parametri che i progettisti degli algoritmi di SCHEDING considerano per massimizzare le prestazioni di un S.O. (dal punto di vista del sistema o dal punto di vista utente) sono:

1. sfruttamento della CPU
2. lavoro utile
3. tempo di ricircolo
4. tempo di attesa
5. tempo di risposta

I primi due sono parametri *system-oriented*, mentre gli altri tre sono *user-oriented*.

Parametri

Sfruttamento della CPU (utilizzo della CPU)

Percentuale del tempo di CPU durante il quale la CPU è impegnata (da 0% a 100%)

La CPU si considera impegnata:

- quando non è inattiva
- quando esegue lavoro utile (escluso funzioni del S.O, *overhead*)

Obiettivo: sfruttamento della CPU del 100% → riduzione del tempo in cui la CPU è *IDLE*

Valori tipici: da 40% a 90%

Lavoro Utile (Produttività)

Questo parametro è anche detto **throughput**: numero di processi completati per unità di tempo.

Tempo di Ricircolo (Completamento)

Anche chiamato **turnaround time**: tempo che impiega un processo per essere completato.

Questo tempo rappresenta l'*intervallo di tempo* da quando un processo viene immesso nel sistema a quando termina ed esce. Risulta essere la somma dei tempi passati nell'attesa di caricamento in memoria (attesa nella *coda batch*), nella coda dei processi pronti, durante la esecuzione nella CPU e nel compiere operazioni di I/O.

Tempo di Attesa

Si riferisce al **tempo di attesa della CPU**, anche chiamato **waiting time**: rappresenta il *tempo speso da un processo in attesa per ottenere la CPU*. Risulta essere la somma degli intervalli di attesa nella coda dei processi pronti.

Tempo di Risposta

Chiamato anche **response time**. Il significato di questo parametro dipende dal tipo di Sistema Operativo:

Sistemi Time-Sharing (sistemi interattivi):

Intervallo di tempo da quando un processo viene immesso nel sistema a quando esce il primo risultato.

Sistemi Real-Time

Intervallo di tempo da quando viene notificato un evento a quando il processo che lo gestisce termina.

Ottimizzazione

L'ottimizzazione delle prestazioni consiste nel:

- **MASSIMIZZARE**
 - lo sfruttamento della CPU
 - il lavoro utile
- **MINIMIZZARE**
 - il tempo di ricircolo
 - il tempo di attesa
 - il tempo di risposta

Ciò è **IMPOSSIBILE** da ottenere, poiché i parametri sono *interdipendenti* e alcuni sono *in contrasto tra loro*. Per attuare una buona ottimizzazione delle prestazioni è necessario un **compromesso tra i vari requisiti**. La scelta deve essere fatta *tenendo conto per cosa deve essere utilizzato il Sistema Operativo*.

Quindi, nella pratica, i progettisti selezionano i parametri in base alle specifiche esigenze:

- Nei *sistemi batch*, massimizzare il lavoro utile e minimizzare il tempo medio di ricircolo.
- Nei *sistemi interattivi*, minimizzare il tempo medio di risposta e il tempo medio di attesa (coda dei processi pronti).

Bisognerebbe però anche considerare la **VARIANZA**, che dovrebbe essere piccola. Nei sistemi interattivi, sarebbe più importante

MINIMIZZARE la varianza del tempo di risposta piuttosto che il tempo medio di risposta. *Un S.O. con un tempo di risposta prevedibile è meglio di un S.O. mediamente rapido, ma molto variabile.*

3.7 Scheduling Con e Senza Preemption

Per lo *scheduler di breve termine* un progettista deve fare una scelta iniziale:

- algoritmi di scheduler che *NON prevedono PREEMPTION* → senza prelazione (non interrompenti)
- algoritmi di scheduler che *prevedono PREEMPTION* → con prelazione (interrompenti)

NON PREEMPTION significa:

Il processo in esecuzione *mantiene il possesso della CPU* fino a che non decide volontariamente di *sospendersi* oppure *termina*.

OSSERVAZIONE: Un algoritmo di scheduling **NON PREEMPTIVE** è *l'unico che può essere utilizzato su certe piattaforme HW*, perché non richiede speciali caratteristiche come la presenza di timer.

PREEMPTION significa:

In riferimento allo Scheduler: al processo in esecuzione può essere sottratta la CPU. Il processo in esecuzione viene riportato in stato di PRONTO.

Esempi: Un processo più prioritario diventa pronto oppure è scaduto il quanto di tempo.

VANTAGGIO:

maggiori velocità di risposta agli eventi o velocità di risposta per gli utenti

SVANTAGGIO:

l'algoritmo di scheduling DEVE essere attivato OGNI VOLTA che ci sono EVENTI che possono modificare lo STATO del SISTEMA

Osservazione: Le prime versioni di Windows (fino alla 3.1) utilizzavano algoritmi di scheduling NON PREEMPTIVE, mentre da Windows 95 in poi si usa uno scheduler PREEMPTIVE.

La possibilità di effettuare PREEMPTION della CPU ha delle ripercussioni a livello di progettazione del nucleo di un S.O.

Infatti, durante una chiamata di sistema per conto di un processo, il nucleo può essere in fase di modifica di strutture dati essenziali del nucleo stesso. Se durante tale system call il processo subisce PREEMPTION, le strutture dati del nucleo potrebbero risultare in uno stato inconsistente!

Alcuni S.O., fra cui la maggior parte delle versioni di UNIX, risolvono questo problema attendendo il completamento delle chiamate di sistema prima di effettuare la PREEMPTION

COME SI OTTIENE QUESTO?

Si consideri che una system call (o parte di essa) può essere considerata una sezione critica sufficientemente breve e quindi basta usare le soluzioni viste cioè:

- Nei S.O. implementati su architettura monoprocesso, basta usare la disabilitazione (come PROLOGO) e la riabilitazione (come EPILOGO) delle interruzioni.
- Nei S.O. implementati su architettura multiprocesso basta usare un meccanismo di lock (LOCK come PROLOGO e UNLOCK come EPILOGO)

SVANTAGGIO: questa soluzione non si concilia con eventuali esigenze real-time!

3.8 CPU Burst - I/O Burst

Durante la sua attività, un processo alterna le seguenti fasi:

- **CPU burst** nella quale il processo usa solo la CPU
- **I/O burst** nella quale il processo effettua operazioni di I/O

3.9 Algoritmi di Scheduling

Questi algoritmi sono applicabili in linea di principio ad ogni tipo di scheduler. Nel seguito si fa specifico riferimento al *CPU scheduling* e si considerano per semplicità, per ogni processo, una sola sequenza di operazioni della CPU (cioè un solo CPU burst).

3.9.1 FCFS

L'acronimo sta per **First Come First Served** detto anche **FIFO** (First In First Out)

È la tecnica più semplice di scheduling per i processi: i processi vengono eseguiti in ordine di arrivo SENZA uso di PREEMPTION. Quando un processo acquisisce la CPU, resta in esecuzione fino a quando si blocca volontariamente o termina

Osservazioni:

- NON PREEMPTION
- Non c'è il concetto di *priorità*

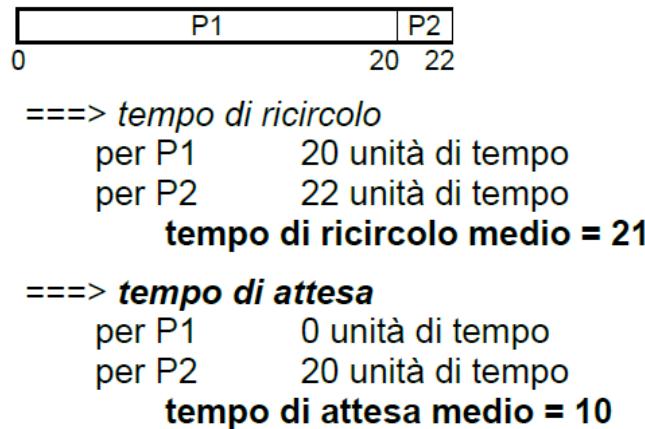
PRESTAZIONI basse, specialmente per processi corti che arrivino dopo processi lunghi:

- basso sfruttamento della CPU
- basso lavoro utile
- elevati tempi di ricircolo
- elevati tempi di risposta

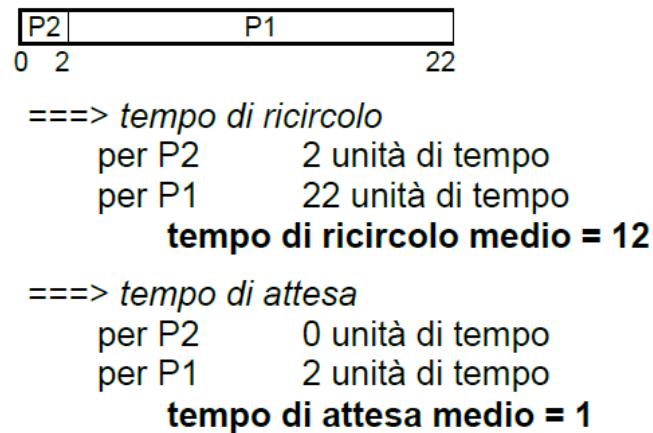
Esempio

- P1 → durata del *CPU burst* 20 unità di tempo
- P2 → durata del *CPU burst* 2 unità di tempo

Caso A: P1 - P2



Caso B: P2 - P1



L'algoritmo FCFS *non consente di variare l'ordine dei processi*. Si può incorrere nell'effetto

CONVOGLIO: processi corti che sono accodati dopo un processo con CPU burst lungo, il che porta a una riduzione del grado di utilizzo della CPU e riduzione del grado di utilizzo dei dispositivi di I/O.

3.9.2 SJF

L'acronimo sta per ***Shortest Job First***, tuttavia viene eseguito *per primo il processo con la più breve lunghezza della successiva sequenza di operazioni CPU* (CPU burst).

Osservazioni:

- *NON PREEMPTION*
- Concetto di priorità particolare

PRESTAZIONI migliori dell'algoritmo *FCFS*.

Svantaggio:

La *difficoltà* nella implementazione dell'algoritmo SJF consiste nel *conoscere la durata della successiva richiesta della CPU*.

Soluzioni:

1. Per lo scheduler di lungo termine, *si può usare come durata il limite di elaborazione indicato dall'utente* per l'intero JOB
2. Per lo scheduler a breve termine, si può predirne il valore usando la storia della esecuzione del processo → ci si riferisce alle durate dei CPU-burst precedenti usando una media esponenziale.

3.9.3 SRTN

L'acronimo sta per ***Shortest Remaining Time Next***. Si tratta della versione dell'algoritmo ***SJF CON PREEMPTION***.

Se arriva nella coda dei processi pronti un processo P1 con lunghezza della successiva sequenza di operazioni della CPU più breve di quello in esecuzione P2, viene mandato in esecuzione P1 e P2 subisce PREEMPTION.

Osservazioni:

- *POSSIBILITÀ DI PREEMPTION*

Vantaggio:

SOLUZIONE OTTIMA in termini di ***minimizzazione del tempo medio di attesa***.

Svantaggio:

Non sempre è possibile sapere a priori la durata del prossimo CPU burst di un processo, al limite è possibile fare delle *stime*.

3.9.4 Round - Robin

Questo algoritmo è ***utilizzato dallo scheduler di breve termine nei sistemi in time-sharing*** (interattivi).

L'algoritmo ha come obiettivo ***avere un buon tempo di risposta*** e una ***equa suddivisione delle risorse tra i processi***.

Utilizzo della *PREEMPTION* → ad ogni processo viene assegnato un *TIME SLICE*, ovvero un ***QUANTO DI TEMPO***.

Regole:

1. Alla scadenza del quanto di tempo il controllo della CPU passa al prossimo processo nella READY QUEUE
2. Se un processo si sospende (volontariamente) prima dello scadere del suo quanto di tempo, il resto non consumato viene perso dal processo (a maggior ragione se il processo termina)

...)

La *READY QUEUE* può essere considerata una coda circolare di processi: in realtà, la coda dei processi pronti è gestita *FIFO*, ma ad ogni processo è assegnata la CPU per un *QUANTO DI TEMPO* (*QT*) prefissato.

NECESSITÀ di un TIMER DEDICATO

Due casi:

1. *TIMER* → interrupt ad ogni quanto → chiamata al CPU SCHEDULER
 - salvataggio processo in esecuzione nel descrittore
 - ripristino stato del prossimo processo nella READY QUEUE
 - passaggio del controllo al nuovo processo RUNNING
2. *Processo RUNNING si sospende* ==> chiamata al CPU SCHEDULER
 - salvataggio processo nel descrittore e accodamento
 - ripristino stato del prossimo processo PRONTO
 - azzeramento timer
 - passaggio del controllo al nuovo processo RUNNING

N.B: Se un processo termina il caso 2 è semplificato dato che non c'è la fase di salvataggio/accodamento del processo.

OSSERVAZIONI:

- processi con CPU burst corti possono richiedere un solo quanto di tempo
- processi con CPU burst lunghi dureranno proporzionalmente alle richieste di risorse

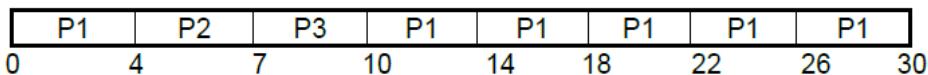
Esempio

Q = 4 unità di tempo

P1 → durata del CPU burst 24 unità di tempo

P2 → durata del CPU burst 3 unità di tempo

P3 → durata del CPU burst 3 unità di tempo



Tempo di Ricircolo:

P1 → 30 unità di tempo

P2 → 7 unità di tempo

P3 → 10 unità di tempo

Tempo di Ricircolo Medio = $47/3 \approx 16$

NB: Con N processi pronti e quanto di tempo Q ogni processo aspetta al massimo T_M la sua prossima fase di esecuzione.

$$T_M = (N - 1) * Q$$

Performance

La PERFORMANCE del ROUND-ROBIN dipende da Q:

- se $Q \approx \infty \rightarrow FCFS$
- se $Q \approx 1$ microsecondo → processor sharing

Gli utenti hanno l'impressione che ciascuno degli N processi abbia un proprio processore in esecuzione a $1/N$ della velocità del processore reale. Ciò è possibile solo se realizzato via HW.

Valori tipici di TIME-SLICE: 10-100 millisecondi.

Ad esempio: **Unix** → 100 millisecondi e **Linux** → 200 millisecondi

Da un punto di vista di principio, **più basso è Q, migliore è il tempo di risposta.**

Bisogna fissare il valore di Q in base al tempo che dura il PROCESS SWITCHING (che è solo overhead).

Esempio

P1 → durata del CPU burst 10 unità di tempo

- **CASO A: Q = 12 unità di tempo**
 - P1 esegue in un singolo quanto di tempo
- **CASO B: Q = 6 unità di tempo**
 - P1 ha bisogno di DUE quanti di tempo
 - NECESSITÀ di UN PROCESS SWITCHING
- **CASO C: Q = 1 unità di tempo**
 - P1 ha bisogno di 10 quanti di tempo
 - NECESSITÀ di NOVE PROCESS SWITCHING

Conclusione:

Il valore di Q deve essere **molto maggiore del process switching time.**

Tempo di Ricircolo

Anche il tempo di ricircolo è influenzato dal valore di Q, ma INVERSAMENTE. Infatti, da un punto di vista di principio, **più alto è Q e migliore è il tempo di turnaround.**

Esempio

P1 → durata del CPU burst 10 unità di tempo

P2 → durata del CPU burst 10 unità di tempo

P3 → durata del CPU burst 10 unità di tempo

CASO A: Q = 1 unità di tempo

I processi eseguono sequenzialmente fino al completamento per 1 unità di tempo ciascuno:

$P1 \rightarrow P2 \rightarrow P3 \rightarrow P1 \rightarrow P2 \rightarrow \dots$

Tempo di ricircolo di P1 = 28

Tempo di ricircolo di P2 = 29

Tempo di ricircolo di P3 = 30

Tempo di Ricircolo Medio = 29 unità di tempo

CASO B: Q = 10 unità di tempo

Tempo di ricircolo di P1 = 10

Tempo di ricircolo di P2 = 20

Tempo di ricircolo di P3 = 30

Tempo di Ricircolo Medio = 20 unità di tempo

REGOLA EMPIRICA: Il QUANTO di TEMPO dovrebbe essere scelto in modo da essere **maggior**e **dell'80% dei CPU burst.**

Osservazione: Spesso la coda dei processi pronti contiene un processo fittizio (detto dummy process) che va in esecuzione quando tutti i processi sono bloccati. Il processo dummy rimane in esecuzione sino a quando qualche altro processo diventa pronto, eseguendo un ciclo senza fine.

3.9.5 A Priorità

L'algoritmo viene detto anche **EVENT-DRIVEN**. Viene **utilizzato dallo scheduler di breve termine** nei sistemi real-time dove è importante un basso tempo di risposta agli eventi esterni.

Ad ogni processo viene assegnata una **PRIORITÀ** (di solito un *numero intero, in un range prefissato*)

Osservazione: in un certo senso anche SJF/SRTN possono essere visti come algoritmi a priorità dove la priorità è la *lunghezza del prossimo CPU burst*.

Il progettista del S.O. deve stabilire le caratteristiche di questa proprietà assegnata ai processi:

1. statica/dinamica
2. assegnata dal sistema/assegnata dall'utente
3. chiarire il range (fisso) dei valori
4. valore basso → priorità alta oppure valore basso → priorità bassa

Si sceglie di implementare in questo caso la **coda dei processi READY con più code**, una per ogni valore di priorità.

Regola: Lo scheduler sceglie *sempre il processo pronto che ha la MASSIMA PRIORITÀ*. Il sistema può fare uso o meno della PREEMPTION.

Con PREEMPTION

Ogni volta che un processo P diventa PRONTO si confronta il suo valore di priorità con quello del processo RUNNING: se è maggiore, va in esecuzione P

Problema:

Un processo a bassa priorità potrebbe NON RIUSCIRE mai ad eseguire → **STARVATION**

Possibile Soluzione (fattibile solo in caso di priorità dinamica e gestita dal S.O.):

Priorità che si innalza più il processo è PRONTO in attesa della CPU → **aging priority** (priorità per anzianità).

3.9.6 Sistemi HARD Real-Time

Presenza di processi CRITICI che devono eseguire **prima di una certa scadenza TEMPORALE (DEADLINE)**.

- **EARLIEST-DEADLINE first scheduler**

- scheduler con prelazione
- seleziona il processo che ha la scadenza più prossima

- **MINIMUM LAXITY first scheduler**

- La *laxity* (rilassabilità) è una *misura della NON urgenza di un processo*: se T è il tempo corrente, D la deadline e C il tempo di esecuzione rimanente, allora:

$$L = (D - T) - C$$

Quindi L è il *tempo che un processo può attendere prima di andare in esecuzione e rispettare la deadline*.

- seleziona il processo che ha minore laxity

3.9.7 Code Multiple

I progettisti di un S.O possono decidere di progettare lo **scheduler sulla base dei diversi "tipi" di processi** che chiaramente hanno **esigenze diverse** → combinando diverse tecniche di SCHEDULING.

- processi di sistema
 - scheduler a priorità (EVENT-DRIVEN)
- processi di utenti interattivi
 - scheduler ROUND-ROBIN
- processi/JOB corrispondenti a lavori BATCH
 - scheduler FCFS o SRTN

Ognuno di questi scheduler lavora su *una sua CODA dei PROCESSI PRONTI*, da qui il nome **CODE MULTIPLE**. Dall'alto verso il basso, lo scheduler ottiene in questo modo le code ad *alta, media e bassa priorità*.

Problema:

SCHEDULING fra gli SCHEDULER di solito basato sulla priorità o su quanti di tempo (non uguali per le varie code).

Osservazione: un processo viene assegnato STATICAMENTE ad una certa CODA

Code Multiple e RETROAZIONE (feedback)

Questo algoritmo rappresenta *un'alternativa al precedente*. Con questo algoritmo:

- un processo può migrare da una CODA all'altra in base al suo comportamento
- si evitano situazioni di starvation o uso eccessivo della CPU

In questo modo, progettando un **sistema con più code**, a ciascuna coda viene assegnata *una certa priorità ed è gestita in ROUND-ROBIN*. A una **coda di priorità minore**, corrispondono **quanti di tempi sempre maggiori**.

Conclusione

Si riserva un **trattamento PREFERENZIALE** ai **processi con CPU burst brevi**. Quelli più lunghi (cioè che consumano più risorse)

"sprofondano" in basso, e sono eseguiti nei tempi morti della CPU. C'è anche un meccanismo di **PROMOZIONE**: se un processo non esaurisce il suo quanto di tempo perché si sospende può risalire la "CHINA".

L'algoritmo di scheduling a *code multiple con retroazione* è il più generale, ma anche *il più complesso da realizzare*.

3.10 Scheduling per Sistemi Multiprocessore

A livello architetturale, i progettisti possono trovarsi di fronte a due casi:

3.10.1 Processori Eterogenei

In questo caso, l'unica scelta possibile è adottare una associazione **STATICA**. Ogni processore ha la propria *coda di processi pronti* e il suo *algoritmo di SCHEDULING*.

3.10.2 Processori Omogenei

Decidendo di adottare la stessa scelta di prima, dove ogni processore ha la propria coda di processi pronti e il suo algoritmo di SCHEDULING, si può incorrere nel seguente problema: *un processore è IDLE con la READY QUEUE vuota*, mentre *un altro è sovraccaricato*. La **soluzione** consiste nell'adottare **un'associazione DINAMICA**.

In questo modo è possibile ottenere una **CONDIVISIONE DEL CARICO (LOAD SHARING)**. Viene implementata **una sola coda di processi pronti** da cui tutti i processori prelevano.

Problema

Il problema è dato dall'**accesso ad una struttura dati comune**.

Possibile Soluzione 1: un processore (MASTER) fa le funzioni di SCHEDULER. Questo è il solo che accede alla coda dei processi pronti e assegna il processo scelto ad un altro processore → **ASYMMETRIC MULTIPROCESSING**

Possibile Soluzione 2: **SYMMETRIC MULTIPROCESSING (SMP)**, assicurandosi che:

- lo stesso processo non venga selezionato da due processori
- alcuni processi non siano persi dalla coda

Questa soluzione che non scala benissimo con il numero di processori: *aumentano le contese nell'accesso alla READY QUEUE*. Potrebbe sembrare che sia meglio tornare ad avere una coda dei processi pronti per ogni CPU, ma così facendo si avrebbe sempre il *problema della distribuzione dei processi nelle varie code*.

3.10.3 Ulteriori Problematiche

L'utilizzo delle **SPIN-LOCK** (anche chiamata **SPINNING**, tecnica che consiste nel verificare periodicamente se il **LOCK** è stato sbloccato) può far nascere l'esigenza di avere uno **smart scheduling**. Se è in esecuzione uno **SPIN-LOCK** lo scheduler può tenere in considerazione di non far scadere il quanto di tempo, in modo da dare al processo che mantiene la SPIN-LOCK più tempo per terminare la propria sezione critica, in modo che possa poi *rilasciare il LOCK* consentendo ad altri processi di accedere a quella sezione critica resettando il *flag*.

Un altro problema è quello della **predilezione** (o *affinity*) per il processore. Se un processo durante la sua vita può cambiare coda per bilanciare il carico dipende se a livello di progettazione si è optato per una *predilezione debole* o una *predilezione forte*:

- **Predilezione DEBOLE** (*soft affinity*):

Il SO tenta di mantenere il processo sullo stesso processore, ma la *migrazione* non è *interdetta se è necessario un bilanciamento* delle READY QUEUE. In particolare, questa *migrazione* può essere di due tipi:

- **GUIDATA** (*push migration*) → un processo dedicato controlla periodicamente la lunghezza delle code;
Es. Linux: eseguito ogni 200ms
- **SPONTANEA** (*pull migration*) → lo scheduler sottrae un PCB ad una coda sovraccarica
Es. Linux: invocata quando la coda si svuota
- **Predilezione FORTE** (*hard affinity*):
Il processo è vincolato all'esecuzione su uno specifico processore → decide di mantenere un *processo in esecuzione sempre sullo stesso processore*, si ha il *riutilizzo del contenuto della cache per burst* successivi.

Linux implementa sia la predilezione debole (*default*) che quella forte, disponendo per quest'ultima chiamate di sistema con cui specifica che un processo non può abbandonare un dato processore.

3.11 Scheduling in LINUX

Prima della **versione 2.5 del kernel**, era una variante dell'**algoritmo tradizionale di UNIX** (*code multiple con feedback* gestite perlopiù con *Round Robin* e quanti di tempo differenziati). Il problema di questo algoritmo è che **era poco scalabile** e poco adatto ai sistemi SMP. La **complessità O(n)** faceva sì che l'**algoritmo non fosse costante**.

Dalla **versione 2.6 del kernel** la **complessità diventa O(1)** e il **tempo è costante** a prescindere dal numero di TASK nel sistema.

Ogni processore ha una *runqueue* che contiene due priority array:

- **active** → insieme dei task che non hanno ancora esaurito il loro quanto di tempo
- **expired** → insieme dei task che hanno eseguito per un intero quanto di tempo

Dalla **versione 2.6.23** lo scheduler viene aggiornato al **Completely Fair Scheduler (CFS)**, che consiste in una classe di scheduler con diverse caratteristiche, di cui la più importante è che viene calcolato un **tempo di esecuzione virtuale per ogni TASK**. Non vengono assegnate le priorità, ma si registra per quanto tempo è stato eseguito ogni TASK (con un **vruntime**) e per scegliere il prossimo TASK da eseguire lo scheduler sceglie quello con il **vruntime minore**.

3.12 Deadlock

3.12.1 Introduzione

DEADLOCK → situazione nella quale uno o più processi rimangono **indefinitamente bloccati** (sospesi), attendendo ciascuno il verificarsi delle condizioni necessarie per il proprio proseguimento.

Da non confondere con la *starvation*, che deriva dalla cattiva gestione della politica di risveglio dei processi, che possono non essere gestiti in maniera *FIFO*.

Si hanno situazioni di deadlock sia nel caso:

- interazione indiretta (risorse riusabili)
- interazione diretta (risorse consumabili)

Di conseguenza, il deadlock può verificarsi sia che si adotti il modello dei processi ad ambiente globale (che li presenta entrambi) sia che si adotti quello locale (in cui si ha solo l'interazione diretta).

Nel caso di *modello globale e interazione indiretta*: più processi possono entrare in competizione per ottenere l'uso di risorse. Se una delle risorse richieste non è disponibile il processo viene posto in condizione di attesa. Se un processo in attesa non cambia più il suo stato cioè le risorse richieste sono trattenute da altri processi essi pure in attesa, si ha una *situazione di deadlock*.

Risorse Riusabili

Proprietà

- devono essere usate in modo esclusivo
- in genere non possono essere sottratte al processo durante l'uso
- numero fisso (e limitato)

Esempio

- **Risorse fisiche** (tempo di CPU, spazio di memoria, file, dispositivi di I/O stampanti, etc.)
- **Risorse logiche** (code, tabelle, etc.)

Esempi di DEADLOCK

- Sistema con due processi (P1 e P2) e quattro unità di TIPO stampante:
Si suppone che ognuno dei due processi (P1 e P2) acquisisca e mantenga due stampanti, ma ne ha bisogno di tre per svolgere per intero il proprio compito → si ha una situazione di DEADLOCK. Sia P1 che P2 attendono di poter acquisire la terza stampante, ma non c'è nessuna stampante libera. Entrambi i processi sono bloccati in attesa del verificarsi di una condizione che *non si può* verificare!
- Sistema con due processi (P1 e P2), una stampante di TIPO1 e una stampante di TIPO2:
Si suppone che il processo P1 acquisisca la stampante di TIPO1 (R1) e il processo P2 acquisisca la stampante di TIPO2 (R2), ma entrambi hanno bisogno dell'altra risorsa → si ha una situazione di DEADLOCK. Ciascun processo attende di acquisire la stampante dell'altro.

Esempi di DEADLOCK utilizzando Semafori

In riferimento al secondo esempio, si utilizzano due semafori *s1* e *s2* per proteggere la sezione critica associata a *R1* e a *R2*:

```
wait(s1);
<S.C1-a>
  wait(s2);
    <S.C2>
      signal(s2);
    <S.C1-b>
      signal(s1);
```

Si ha uno pseudocodice di questo tipo, per ciascuno dei due processi. Ciascuno dei due processi fa una prima wait sul proprio semaforo per accedere alla prima stampante, per poi rimanere in attesa sulla seconda wait, essendo l'altra stampante ancora occupata dal processo rivale. La soluzione è corretta dal punto vista del problema della sezione critica e il meccanismo del semaforo in sé non presenta problemi di deadlock.

Risorse Consumabili

Le risorse consumabili sono rappresentate dai *segnali e messaggi* che vengono scambiati tra i processi.

Caso Modello Globale

Si hanno due processi *P* e *Q* e due buffer *A* e *B*. *P* inserisce un messaggio nel buffer *A* e preleva un messaggio dal buffer *B*. *Q* inserisce un messaggio nel buffer *B* e preleva un messaggio dal buffer *A*. In questo caso si ha una situazione di deadlock quando *P* attende di inserire un messaggio nel buffer *A* pieno (rimanendo così in stato *bloccato*) e *Q* attende di inserire un messaggio nel buffer *B* pieno. Entrambi i processi operano per inserire un messaggio senza che nessuno cominci a prelevarli. Oppure, in caso contrario, si può verificare quando entrambi attendono di prelevare un messaggio senza che nessuno inserisca qualcosa nel buffer.

Caso Modello Locale

- *P* spedisce un messaggio a *Q*
- *Q* spedisce un messaggio a *P*
- *P* riceve un messaggio da *Q*
- *Q* riceve un messaggio da *P*

Se le **send sono sincrone**. Nel caso di *P*, per andare ad eseguire l'istruzione seguente deve effettuare il *rendez vous* con *Q*, il quale a sua volta deve fare il *rendez vous* con *P*.

Se le **send sono asincrone** ma il *buffer* è *limitato*, può verificarsi comunque *deadlock* quando i *canali* sono saturi.

La deadlock si avrà sempre anche partendo con delle *receive*, in quanto sono *bloccanti*.

3.12.2 Schema di Uso di un Processo - Caso Risorse Riusabili

Consideriamo risorse di diverso tipo: *R₁*, *R₂*, ..., *R_m*.

Ciascun tipo di risorsa *R_i* ha *W_i* istanze (con *W_i ≥ 1*). Il protocollo di accesso alle risorse da parte dei processi prevede che il processo deve rispettare la seguente sequenza:

- **RICHIESTA**: Se la richiesta non può essere soddisfatta immediatamente (perché la risorsa è già utilizzata da un altro processo) il processo richiedente deve attendere fino al rilascio della risorsa da parte del processo che la detiene
- UTILIZZO
- RILASCIO

La richiesta ed il rilascio sono realizzate da apposite SYSTEM CALL e possono essere realizzate ad esempio con *WAIT* e *SIGNAL* su semafori. In caso di system call, il S.O. controlla per ogni utilizzo che la risorsa sia stata richiesta ed assegnata. Una tabella di sistema registra lo stato di ogni risorsa e, se occupata, il processo che la sta utilizzando.

3.12.3 Condizioni per il Deadlock

Si hanno *n* processi *P₁*, *P₂*, ..., *P_n* che usano *m* tipi di risorse *R₁*, *R₂*, ..., *R_m*. Si verifica una condizione di deadlock solo se **sono vere tutte le seguenti quattro condizioni**:

1. Le risorse possono essere **utilizzate da un solo processo** alla volta → MUTUA ESCLUSIONE
2. I processi trattengono le risorse che già possiedono mentre chiedono risorse addizionali (condizione di allocazione PARZIALE) → TRATTENIMENTO DELLE RISORSE E ATTESA

3. Le risorse già assegnate ai processi non possono essere sottratte a questi → MANCANZA DI PREEMPTION
4. Esiste un insieme di processi (P_i, P_{i+1}, \dots, P_k) tali che P_i è in attesa di una risorsa posseduta da P_{i+1} , P_{i+1} è in attesa di una risorsa posseduta da P_{i+2}, \dots, P_k è in attesa di una risorsa posseduta da P_i → ATTESA CIRCOLARE

Le quattro condizioni **non sono indipendenti fra di loro**: ad esempio, la condizione di ATTESA CIRCOLARE implica la condizione di TRATTENIMENTO E ATTESA.

3.12.4 Grafo di Allocazione delle Risorse

Si definisce grafo di allocazione (o assegnazione) delle risorse:

$$G = (V, E)$$

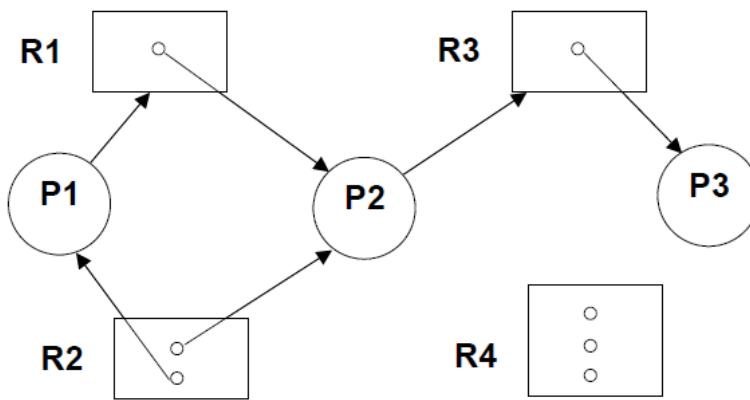
- $V \rightarrow$ insieme di vertici (o nodi), partizionato in:
 - $P = \{P_1, P_2, \dots, P_n\}$ insieme di PROCESSI
 - $R = \{R_1, R_2, \dots, R_m\}$ insieme dei TIPI di risorse
- $E \rightarrow$ insieme di lati (o archi), dove ogni suo elemento è una coppia ordinata:
 - $(P_i, R_j) \rightarrow$ LATO RICHIESTA
 - $(R_j, P_i) \rightarrow$ LATO ASSEGNAZIONE

Esempio di Ciclo (con deadlock)

Risorse:

- R_1 ha una sola istanza
- R_2 ha due istanze
- R_3 ha una sola istanza
- R_4 ha tre istanze

$$E = \{(P_1, R_1), (P_2, R_3), (R_1, P_2), (R_2, P_1), (R_2, P_2), (R_3, P_3)\}$$



Condizioni per il Deadlock

Se esiste un ciclo nel grafo di allocazione delle risorse allora PUÒ esistere una situazione di DEADLOCK. Se il grafo di allocazione delle risorse NON CONTIENE cicli allora non esiste una situazione di DEADLOCK. Ci sono **due cicli**:

1. $P_1 \rightarrow R_1 \rightarrow P_2 \rightarrow R_3 \rightarrow P_3 \rightarrow R_2 \rightarrow P_1$
2. $P_2 \rightarrow R_3 \rightarrow P_3 \rightarrow R_2 \rightarrow P_2$

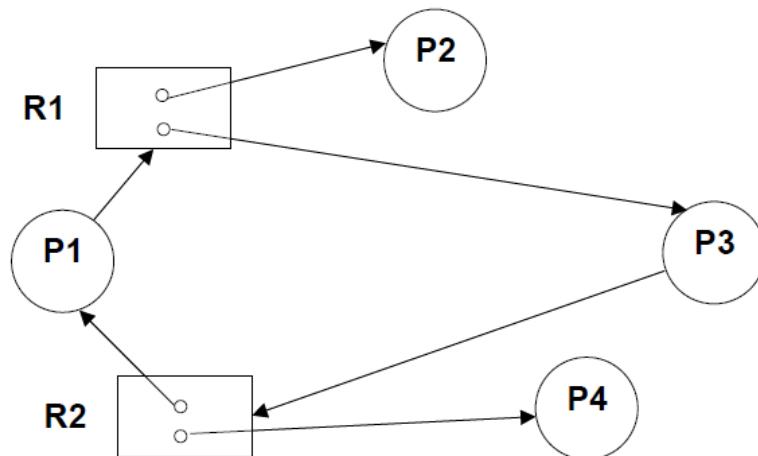
P1, P2 e P3 sono in una situazione di DEADLOCK: P2 aspetta l'istanza della risorsa R3 che è usata da P3, ma P3 aspetta perché P1 o P2 liberino una istanza della risorsa R2 e P1 aspetta perché P2 liberi l'istanza della risorsa R1.

Esempio di Ciclo (senza deadlock)

Risorse:

- R1 ha due istanze
- R2 ha due istanze

$$E = \{ (P1, R1), (P3, R2), (R1, P2), (R1, P3), (R2, P1), (R2, P4) \}$$



C'è **un ciclo**:

$$P1 \rightarrow R1 \rightarrow P3 \rightarrow R2 \rightarrow P1$$

P1, P2, P3 e P4 non sono in una situazione di deadlock, infatti:

- P4 può rilasciare una istanza della risorsa R2, che può essere allocata a P3 "rompendo" il ciclo
- P2 può rilasciare una istanza della risorsa R1, che può essere allocata a P1 "rompendo" il ciclo

3.12.5 Metodi Trattamento Deadlock

- Il primo metodo di trattamento del deadlock è la **PREVENZIONE**, che consiste nell'assicurarsi che il sistema **non entri mai** in una situazione di DEADLOCK. Si fa distinzione tra:
 - **PREVENZIONE STATICÀ** assicurarsi che *almeno una delle quattro condizioni necessarie non si verifichi* (in particolare, ATTESA CIRCOLARE) → restrizioni nell'uso delle risorse
 - **PREVENZIONE DINAMICA** *verificare*, sulla base dello stato corrente di allocazione delle risorse e delle richieste, se soddisfare una richiesta di allocazione di una risorsa libera può portare ad una possibile situazione di deadlock → occorrono delle informazioni aggiuntive
- Si tratta di un **metodo a posteriori**. Accorgersi del DEADLOCK e "rimediare" al problema → **DETECTION** e **RECOVERY**. Si effettuano delle azioni per ELIMINARLO: "uccisione" di processi oppure PREEMPTION delle risorse.

Prevenzione Statica

Detta anche *DEADLOCK PREVENTION*, consiste nell'assicurarsi all'atto della scrittura del programma che almeno una delle quattro condizioni necessarie non si verifichi. **Prevenire** i deadlock ha come svantaggio un basso utilizzo delle risorse e uno throughput ridotto. Si considerano le quattro condizioni separatamente:

Mutua Esclusione

Se il tipo di risorsa è non-condivisibile cioè se una istanza deve essere usata da un processo alla volta allora questa condizione *non può essere eliminata*. Invece, se il tipo di **risorsa è condivisibile** cioè se ogni istanza può essere usata da più processi per volta, (come ad esempio un file read-only), allora questa condizione *può essere eliminata* e l'uso di questo tipo di risorsa **non produce mai deadlock**.

Trattenimento delle Risorse e Attesa

Per evitare questa condizione esistono due metodi:

1. Un processo acquisisce tutte le risorse di cui ha bisogno all'inizio, le usa e poi le rilascia tutte in blocco

Problemi

- bisogna fare una STIMA delle necessità di un processo. Più facile per JOB meno facile per processi INTERATTIVI → SOVRASTIMA. Se un processo ha bisogno di più risorse, bisogna fare sì che o le prenda entrambe o nessuna.
- alcune risorse *preallocate* potrebbero essere utilizzate solo per periodi di tempo molto limitati

2. Un processo può acquisire una risorsa R1 se e solo se non ha acquisito nessuna altra risorsa R2. In caso contrario, deve rilasciare R2 se vuole acquisire R1.

Problemi

- più riacquisizioni e rilasci
- non è detto che per tutte le risorse sia facile rilasciarle e poi riacquisirle

Vantaggi

- stima non necessaria
- non trattiene le risorse quando non SERVE

Svantaggi di Entrambe le Soluzioni

- abbassamento del grado di utilizzo delle risorse e la concorrenza
- possibile l'attesa indefinita per processi con forti richieste di risorse

Mancanza di Preemption

Un processo che si pone in attesa di una risorsa R1, mentre ha già acquisito un'altra risorsa R2, si vede sottrarre R2. Quando R1 si sarà liberata, dovrà essere riassegnata anche R2.

Problema

Non è detto che per tutte le risorse sia facile il rilascio e poi la riassegnazione.

Osservazione

Questo protocollo può essere adatto per risorse il cui stato si può salvare e recuperare facilmente (registri, memoria; ma non per dispositivi di I/O).

Attesa Circolare

Per evitarla è necessario imporre un **ordinamento totale** (possibilmente logico) su tutti i tipi di risorse.

$$R = \{ R_1, R_2, \dots, R_m \}$$

Associamo un **numero intero positivo a ogni tipo di risorsa**

$$F : R \rightarrow \mathbb{N}$$

Nell'allocazione delle risorse ai processi bisogna seguire questo protocollo: ogni processo può richiedere solo risorse in ordine crescente di numerazione. Quindi, inizialmente, un processo P può richiedere un qualunque numero di istanze del tipo di risorsa R_i . In seguito, P può richiedere istanze del tipo di risorsa R_j se e solo se $F(R_j) > F(R_i)$.

Esempio:

$$F(\text{disco}) = 5$$

$$F(\text{stampante}) = 12$$

Il processo P vuole usare il disco e la stampante, allora deve richiedere prima il disco e poi la stampante. In alternativa, un processo P può acquisire istanze del TIPO R_j se ha rilasciato ogni risorsa di tipo R_i tale che $F(R_i) \geq F(R_j)$. Seguendo questo protocollo non si può mai instaurare una condizione di attesa circolare. La funzione F deve essere definita secondo l'ordine d'uso normale delle risorse del sistema. Ad esempio, poiché generalmente l'unità disco si usa prima della stampante è sensato definire: $F(\text{disco}) < F(\text{stampante})$.

Prevenzione Dinamica

Detta anche **DEADLOCK AVOIDANCE**. **Evitare i deadlock** presuppone che il sistema conosca a priori informazioni addizionali sulle richieste future dei processi. Si tratta di **tecniche di prevenzione che agiscono a tempo di esecuzione** e verificano, sulla base dello stato corrente di allocazione delle risorse e delle richieste complessive dei processi, se il soddisfacimento di una richiesta per una risorsa può portare ad una possibile situazione di deadlock. Si soddisfa una richiesta solo se questa non porta ad un possibile blocco critico. Non c'è sempre la certezza che si arrivi al deadlock, si ottiene solo un *campanello d'allarme*.

Occorrono delle **informazioni aggiuntive** sulle *richieste* delle risorse. La tecnica più semplice di questa categoria si basa sull'*informazione del massimo numero di risorse di ciascun tipo* di cui ogni processo ha bisogno durante la sua esistenza.

STATO DI ALLOCAZIONE DELLE RISORSE → numero di risorse allocate e disponibili e il massimo richiesto dai processi. Quando un processo richiede una risorsa disponibile il S.O. deve decidere se la sua (loro) allocazione immediata mantenga il sistema in stato sicuro. Uno **stato** si dice **SICURO** se il sistema può allocare le risorse ad ogni processo, in un qualche ordine, evitando possibili situazioni di deadlock, e quindi **esiste una sequenza sicura** di esecuzione dei processi. Non è detto tuttavia che uno **stato non sicuro** porti necessariamente al deadlock.

Facendo in modo che il *sistema transiti sempre in uno stato sicuro si evita certamente il deadlock*.

Esempio

Si considera un sistema con **un solo tipo di risorsa**, con 12 istanze e 3 processi: P_0, P_1, P_2 .

Processo	Necessità Massime	Allocazioni Correnti
P0	10	5
P1	4	2
P2	9	2

Sono ancora presenti 3 *istanze libere*. Il **sistema è in uno stato sicuro**, infatti esiste la sequenza $\langle P1, P0, P2 \rangle$ che è *sicura*.

- P1 può acquisire le altre 2 istanze di cui necessita e terminare e quindi liberare le istanze: 5 istanze libere ($4 + 1$)
- P0 può acquisirle e terminare: libera 10 istanze
- P2 può acquisirne 7 e terminare

Si può passare ad uno **stato non sicuro** se al tempo T1 P2 richiede un'ulteriore istanza e gli viene allocata:

Processo	Necessità Massime	Allocazioni Correnti
P0	10	5
P1	4	2
P2	9	3

Sono ancora presenti 2 *istanze libere*.

- Solo P1 può acquisire le altre 2 istanze di cui necessita e terminare: libera 4 istanze
- P0 e P2 hanno bisogno entrambi di un numero maggiore di istanze → **DEADLOCK**

Le tecniche di **DEADLOCK AVOIDANCE** devono **stabilire se lo stato di allocazione** cui si perviene è **sicuro o meno**.

Lo svantaggio di questa tecnica è dato dall'*abbassamento del grado di utilizzo delle risorse*.

Evitare il Deadlock: Algoritmo del Banchiere

Si utilizzano diverse strutture dati per **memorizzare lo stato di allocazione del sistema**. I processi sono visti come *clienti* che possono richiedere credito (*necessità di un processo*). L'algoritmo del banchiere non è svolto da un processo, ma dal sistema durante una *supervisor call*, qualora esso voglia fare *prevenzione dinamica*. Si definiscono i seguenti elementi:

- ***n*** → numero di processi
- ***m*** → numero di *tipi di risorse*
- ***DISPONIBILI*** → vettori di lunghezza *m* per memorizzare il numero di *risorse disponibili per ogni tipo*.

$DISPONIBILI[j] = k \rightarrow k$ istanza del tipo R_j dove $1 \leq j \leq m$

- ***NECESSITÀ_MAX*** → matrice $n \times m$ per memorizzare le *necessità massime di ogni processo*.

$NECESSITÀ_MAX[i, j] = k \rightarrow$ al processo P_i servono al massimo k istanze di TIPO R_j $1 \leq i \leq n$ e $1 \leq j \leq m$

- ***ALLOCAZIONE*** → matrice $n \times m$ per memorizzare le *situazione attuale di allocazione di ogni processo per ciascun tipo di risorsa*.

$ALLOCAZIONE[i, j] = k \rightarrow$ al processo P_i sono state allocate k istanze di TIPO R_j $1 \leq i \leq n$ e $1 \leq j \leq m$

- **NECESSITÀ** → matrice $n \times m$ per memorizzare le *necessità rimaste di ogni processo per ciascun tipo di risorsa*.

$NECESSITÀ[i, j] = k \rightarrow$ processo P_i servono ancora k istanze di TIPO R_j $1 \leq i \leq n$ e $1 \leq j \leq m$

La *necessità* è data da $NECESSITÀ_{MAX}$ - $ALLOCAZIONE$ (vettori).

- **RICHIESTA_K** → vettore delle richieste del Processo P_K

- $RICHIESTA_K \leq NECESSITÀ_K \rightarrow$ altrimenti errore
- $RICHIESTA_K \leq DISPONIBILI \rightarrow$ altrimenti c'è almeno una risorsa NON DISPONIBILE e il processo P_K deve aspettare

- **STATO S* (STATO FUTURO)**, definito da:

- $DISPONIBILI := DISPONIBILI - RICHIESTA_K$
- $ALLOCAZIONE_K := ALLOCAZIONE_K + RICHIESTA_K$
- $NECESSITÀ_K := NECESSITÀ_K - RICHIESTA_K$

Uno stato viene considerato *SICURO* se tutti i processi che hanno già ottenuto delle risorse sono in grado di completare la propria esecuzione, anche se ciascun processo intendesse usare tutte le risorse di cui ha bisogno.

Provare che uno STATO S* È SALVO

Definiamo ogni richiesta del processo P_K come $RICHIESTA_K$. Memorizziamo lo stato attuale (S_0) e introduciamo il vettore:

- **FINITO** → $FINITO[i] := false$ $1 \leq i \leq n$

1. valutare lo stato S^* che si otterrebbe soddisfacendo la richiesta $RICHIESTA_K$

2. trovare un indice i tale per cui

- $FINITO[i] = false$
- $NECESSITÀ_i \leq DISPONIBILE$

Se non esiste andare al passo 4

3. valutare il nuovo stato S^{*+} che ottenuto da:

$DISPONIBILI = DISPONIBILI + ALLOCAZIONE_i$

$FINITO[i] := true$

Quindi tornare al passo 2

4. se $FINITO[i] = true$ per ogni i allora lo stato è *sicuro*, altrimenti lo stato non è *sicuro*

Se **lo stato S è sicuro*** il sistema **soddisfa la richiesta $RICHIESTA_K$** e quindi **S diventa il nuovo stato*** del sistema (S_1), altrimenti se **S* non è sicuro** si **ripristina lo stato iniziale S_0** e la **$RICHIESTA_K$ del processo P_K non è soddisfatta**.

Esempio 1

- 3 processi
- 1 tipo di risorsa con 10 istanze

Tempo T0

	ALLOCAZIONE	NECESSITÀ_MAX	DISPONIBILI	NECESSITÀ
P1	4	8	2	4
P2	2	3		1
P3	2	9		7

In queste condizioni, lo **STATO S0 è sicuro**, infatti esiste la sequenza: $\langle P2, P1, P3 \rangle$ che è SICURA. Supponiamo che all'istante T1 arrivi una richiesta di un'altra risorsa da parte di P3.

$$RICHIEDA_3 = (1) < DISPONIBILI = (2)$$

Si calcola lo STATO S^* a cui si perverrebbe con il soddisfacimento di questa richiesta:

Si definisce un vettore FINITO[$false_1, false_2, false_3$].

Tempo T1 → Passo 1

	ALLOCAZIONE	NECESSITÀ_MAX	DISPONIBILI	NECESSITÀ
P1	4	8	1	4
P2	2	3		1
P3	3	9		6

Passo 2 → Si cerca un valore i tale per cui $FINITO[i] = false$: in questo caso tutti rispettano questa condizione. Inoltre, per quell'i-esimo processo la sua necessità deve essere minore o uguale a disponibile. L'unico per cui ciò avviene è il processo P2.

Passo 3 → $DISPONIBILI = DISPONIBILI + ALLOCAZIONE_i = 1 + 2 = 3$ e il vettore diventa FINITO[$false_1, true_2, false_3$].

Ritorno al Passo 2 → Gli ultimi due processi rimasti corrispondono sempre a valori dell'indice i per cui $FINITO[i] = false$. Per entrambi, la necessità supera la disponibilità, pertanto non è possibile proseguire.

Lo **STATO S^* non è sicuro**. Non esiste più una sequenza sicura, pertanto si ripristina lo STATO S0.

Esempio 2

- 5 processi
- 3 tipi di risorse
 - A con 10 istanze
 - B con 5 istanze
 - C con 7 istanze

Tempo T0

	ALLOCAZIONE	NECESSITÀ_MAX	DISPONIBILI	NECESSITÀ
	ABC	ABC	ABC	ABC
P0	010	753	332	743
P1	200	322		122
P2	302	902		600
P3	211	222		011
P4	002	433		431

Applicando l'algoritmo del banchiere, questo **STATO SO è sicuro**. Esiste infatti la sequenza: $\langle P1, P3, P4, P2, P0 \rangle$ che è *sicura*. Supponiamo che all'istante T1 arrivi una richiesta di un'altra risorse: 1 istanza di A e 2 istanze di C da parte di P1. Calcoliamo dunque lo **STATO *** di questa richiesta.

Tempo T1

	ALLOCAZIONE	NECESSITÀ_MAX	DISPONIBILI	NECESSITÀ
	ABC	ABC	ABC	ABC
P0	010	753	230	743
P1	302	322		020
P2	302	902		600
P3	211	222		011
P4	002	433		431

Questo stato è **ancora sicuro**, in quanto esiste la sequenza: $\langle P1, P3, P4, P0, P2 \rangle$ che è *SICURA*. Si può soddisfare *RICHIEDA1* e lo stato **S* diventa il nuovo stato effettivo S1**.

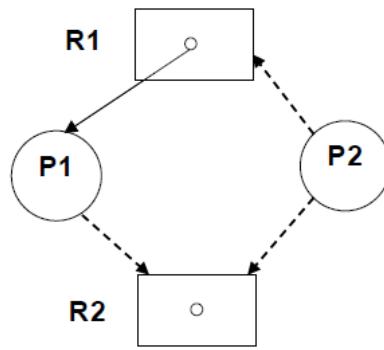
Svantaggi del Metodo

- bisogna conoscere le necessità massime di ogni processo
- presuppone che, in un certo istante, ogni processo richieda tutte le ulteriori risorse di cui ha bisogno e che le mantenga fino a che non ha terminato → CASO PEGGIORE
- molto pesante computazionalmente → mxn^2 operazioni

Caso Particolare

Si consideri **una singola istanza per ogni tipo di risorsa**. Questo caso rappresenta l'**ALGORITMO PIÙ EFFICIENTE**.

Si utilizza il *grafo di allocazione delle risorse* con in più un *nuovo tipo di arco* (indicato con una linea tratteggiata da Pi a Rj (P_i, R_j), il quale rappresenta una **RICHIESTA FUTURA**, e che quindi indica che P_i avrà bisogno di R_j in futuro).



In questo caso *P1 ha chiesto e ricevuto la Risorsa R1*. Come necessità residue, *P1 ha richiesto l'utilizzo di R2*, così come *P2 ha richiesto l'utilizzo di R1 e R2*. Se un processo P_i richiede una risorsa R_j allora questa **richiesta potrà essere esaudita SE E SOLO SE questo non porta alla formazione di un ciclo** nel grafo.

Vantaggio

La ricerca di un ciclo in un grafo richiede n^2 operazioni (perché ogni risorsa ha una singola istanza).

Deadlock Detection e Recovery

Consiste nell'individuazione del fenomeno di deadlock e della sua eliminazione: la prima fase viene detta *DETECTION*, mentre la seconda *RECOVERY*.

Detection

Un algoritmo per scoprire se è presente una situazione di deadlock si basa su strutture dati simili a quelle usate per l'algoritmo del banchiere, ma agisce a posteriori.

- $n \rightarrow$ numero di PROCESSI
 - $m \rightarrow$ numero di TIPI di RISORSE
 - ***DISPONIBILI*** → vettore di lunghezza m per memorizzare il numero di risorse disponibili per ogni TIPO
 - ***ALLOCAZIONE*** → matrice $n \times m$ per memorizzare la situazione attuale di allocazione di ogni PROCESSO per ogni TIPO di risorse
 - ***RICHIESTE*** → matrice $n \times m$ per memorizzare le richieste correnti di ogni PROCESSO per ogni TIPO di risorse
- $RICHIESTE[i, j] = k \rightarrow$ il processo P_i richiede ancora k istanze di TIPO R_j $1 \leq i \leq n$ e $1 \leq j \leq m$

Algoritmo

L'algoritmo procede **verificando** tutte le **possibili sequenze dei processi che devono essere completati**. Tra queste *deve esistere almeno una sequenza che porta al completamento*. Non si parla più di *stato sicuro*, ma semplicemente di *deadlock o non deadlock*.

Memorizziamo lo stato attuale S_0 e introduciamo il vettore: $FINITO \rightarrow FINITO[i] := false \quad 1 \leq i \leq n$

1. trovare un i tale per cui

- $FINITO[i] = false$
- $RICHIESTE_i \leq DISPONIBILI$

Se non esiste andare al passo 3.

2. valutare il nuovo stato che si ottiene da

- $DISPONIBILI = DISPONIBILI + ALLOCAZIONE_i$
- $FINITO[i] := true$

[Tornare al passo 1](#)

3. se $FINITO[i] = \text{false}$ per un qualche i allora il sistema è in una situazione di DEADLOCK: il processo P_i è in DEADLOCK.

Viene ripristinato lo stato iniziale S_0 .

Visione Ottimistica

Si suppone che i processi non richiedano altre risorse. Se questi dovessero farlo, un sistema che nello stato S_0 non presenta deadlock potrebbe nel futuro presentare un DEADLOCK a causa delle nuove richieste di risorse da parte dei processi.

Esempio

- 5 processi
- 3 tipi di risorse
 - A con 7 istanze
 - B con 2 istanze
 - C con 6 istanze

Tempo T0

	ALLOCAZIONE	RICHIESTE	DISPONIBILI
	ABC	ABC	ABC
P0	010	000	000
P1	200	202	
P2	303	000	
P3	211	100	
P4	002	002	

Poiché il sistema ha esaurito tutte le risorse disponibili, questo potrebbe rappresentare un *campanello d'allarme per una situazione di deadlock* che fa scattare l'*algoritmo di detection*. Lo **STATO S0** non presenta nessuna situazione di DEADLOCK. Infatti, esiste la sequenza $\langle P_0, P_2, P_1, P_3, P_4 \rangle$ che porta $FINITO[i]=\text{true}$ per ogni i . P_0 e P_2 sono gli unici processi che possono pensare di eseguire, a differenza degli altri che sono sospesi. Supponiamo ora che il processo P_2 faccia una richiesta addizionale per una istanza di tipo C.

Tempo T1

	ALLOCAZIONE	RICHIESTE	DISPONIBILI
	ABC	ABC	ABC
P0	010	000	000
P1	200	202	
P2	303	001	
P3	211	100	
P4	002	002	

Questo **STATO S1** presenta una situazione di DEADLOCK. Infatti, anche supponendo che P0 finisce e rilasci quindi le sue risorse, queste non sono sufficiente per gli altri processi P1, P2, P3, P4 sono in DEADLOCK.

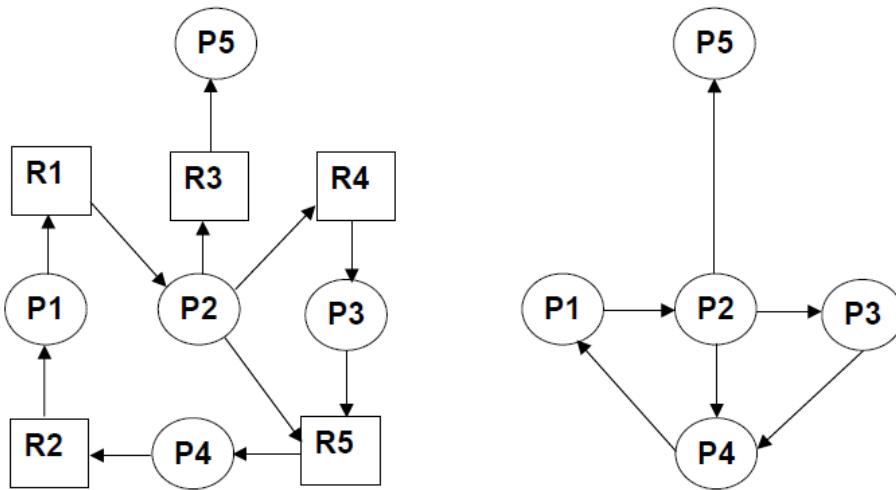
Svantaggio

Si tratta di un *algoritmo computazionalmente pesante: mxn^2 operazioni.*

Caso Particolare

Si crea una singola istanza per ogni tipo di risorsa. Questo metodo rappresenta l'**ALGORITMO PIÙ EFFICIENTE**.

Si utilizza una *variante del grafo di allocazione delle risorse: GRAFO DI ATTESA* → non ci sono più i nodi risorsa se nel grafo di allocazione esistono gli archi (P_i, R_j) e (R_j, P_k) , allora esiste nel grafo di attesa l'arco (P_i, P_k) che significa che P_i sta aspettando una risorsa posseduta da P_k .



Esiste un DEADLOCK se e solo se esiste un ciclo nel grafo di attesa.

Vantaggio

La ricerca di un ciclo in un grafo richiede n^2 operazioni.

Scelto l'algoritmo di DETECTION, bisogna stabilire: ogni **quanto attivarlo**:

- *attivazione frequente*: solo se i DEADLOCK si presentano frequentemente → alto overhead
- *attivazione più sporadica*: i DEADLOCK passano inosservati per molto tempo → basso overhead

Al limite: attivarlo ogni volta che una richiesta per una risorsa non può essere soddisfatta.

Oppure: basarsi sul valore dell'utilizzo della CPU. ATTIVAZIONE dell'algoritmo di DEADLOCK DETECTION solo se questo valore scende sotto una certa soglia (ad esempio al di sotto del 40%).

Recovery

Una volta scoperta una situazione di deadlock bisogna *porvi rimedio*. Per farlo si hanno *due modi*:

1. *Uccidere uno o più processi* che sono *coinvolti* nel deadlock: si recuperano le risorse dei processi uccisi.
2. *Togliere le risorse a uno o più processi* che sono in deadlock (**RESOURCE PREEMPTION**)

Uccisione di Processi

Non è sempre semplice terminare un processo, talvolta un processo potrebbe stare aggiornando un file o stampando, ad esempio. Abbiamo due metodi per terminare i processi:

- uccisione di TUTTI i processi in DEADLOCK (molto drastico)
- uccisione di un processo coinvolto alla volta fino a che non si elimina il deadlock (meno drastico). Ciò implica che dopo ogni uccisione si verifichi tramite l'algoritmo di detection se esiste ancora il deadlock.

Come si fa ad eleggere la vittima?

Bisognerebbe scegliere quelli che implicano "costo minimo". Per farlo si hanno a disposizione diversi fattori:

- priorità del processo
- quanto ha eseguito e quanto deve ancora eseguire (se è prossimo al termine non conviene)
- quante e quali tipi di risorse ha usato (non ha senso uccidere un processo che libera poche risorse)
- quante risorse gli servono per terminare
- quanti processi saranno coinvolti in questo ROLLBACK

Resource Preemption

Le *risorse tolte a uno* o più processi vengono *date agli altri* fino a che non si elimina il deadlock.

Per farlo è necessario stabilire una vittima: come prima il discorso è ancora legato a una valutazione del "costo minimo". La vittima, una volta che gli sono state sottratte le risorse, deve tornare in uno *state consistente* → **ROLLBACK**.

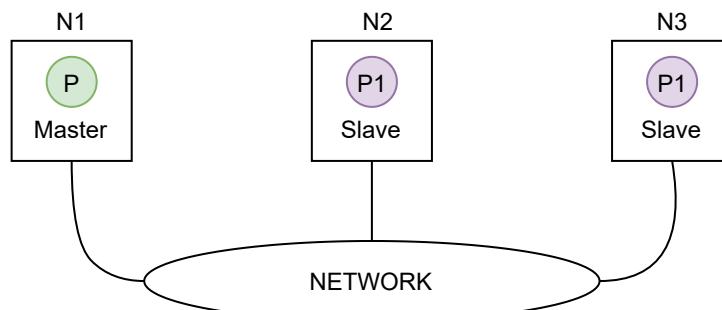
Anche il *rollback* può essere implementato in modi diversi:

- **ROLLBACK totale** → il processo ricomincia dall'inizio (caso più semplice)
- **Mantenere dei CHECKPOINT** in un processo → punti che ricordano lo stato di esecuzione di un processo, a cui si può "tornare" in caso di necessità.

Anche nel deadlock è bene **evitare situazioni di STARVATION**: ogni volta che si scopre un deadlock è bene **evitare che venga selezionato sempre lo stesso processo come "vittima"**. Per farlo, si può pensare ad un *numero massimo di volte* che un processo può subire un *rollback*.

Tolleranza ai Guasti con Politica Master-Slave (o a Copie Passive)

I checkpoint sono anche utilizzati per realizzare politiche di tolleranza ai guasti *master-slave*. Anziché un'architettura con *un solo nodo di esecuzione*, si considera *un'architettura distribuita*, in modo da avere *ridondanza dei nodi*, con ad esempio altri due nodi. Solitamente sono altri due perché la tolleranza ai guasti è maggiormente garantita con una triplice ridondanza.



Il Processo P *master* ogni tanto esegue un *checkpoint* (salvataggio dello stato di esecuzione) e lo manda tramite la rete a P1 e P2. Se il nodo N1 su cui risiede P ha un *fault*, il primo slave, cioè P1 parte ad eseguire dall'ultimo checkpoint ricevuto e assume il ruolo di *master*. Questo comincia a fare quello che faceva P in termini di esecuzione, andando ogni tanto a effettuare dei checkpoint su P2. Nel frattempo, il *nodo N1 viene riparato* e viene creato un *secondo slave* in modo che ci sia sempre una ***triplice ridondanza***.

Metodo Misto

Dato che nessun metodo di quelli visti è il migliore, si può adottare un approccio misto. Le ***risorse*** vengono ***suddivise in classi***:

- Le classi vengono ordinate (in modo da non avere problemi di attesa circolare)
- Ad ogni classe viene applicato il metodo più opportuno

Classe 1 - Risorse Interne usate dal S.O

Prevenzione statica tramite *ordinamento delle risorse*.

Esempio:

PCB, Buffer I/O

Classe 2 - Memoria Centrale

Prevenzione statica tramite *PREEMPTION della risorsa* tramite *swapping in memoria secondaria*.

Esempio:

Blocchi di Memoria

Classe 3 - Risorse di Processo

Prevenzione dinamica tramite *DEADLOCK AVOIDANCE*. La richiesta massima di risorse è nota a priori.

Esempio:

File

Classe 4 - Area di Swap

Prevenzione statica tramite *preallocazione di tutto lo spazio necessario a priori*.

Esempio:

Blocchi su Disco

Memoria

Per poter essere eseguiti i programmi devono essere caricati, almeno parzialmente, in memoria. Nella fase di **fetch** la cpu preleva le istruzioni dalla memoria in base al contenuto del registro Program Counter **PC**. Le istruzioni possono determinare ulteriori letture e scritture di dati in specifici indirizzi di memoria. Le strategie di indicizzazione della memoria sono un componente importantissimo di un sistema operativo. Abbiamo già visto come tramite diverse politiche di scheduling si possono ottenere aumenti di performance, in particolare attraverso la multiprogrammazione, bisogna quindi che la memoria possa ospitare più processi contemporaneamente per garantire tali prestazioni.

1. Punto di vista utente

Per quanto riguarda l'utente, colui che ha più contatto con la memoria è il programmatore, questo tipo di utente ragiona in termini di variabili e codice utilizzando linguaggi di programmazione di alto livello. Tramite nomi simbolici si fa riferimento a variabili, dietro le quante queste vengono tradotte in indirizzi **logici** o **virtuali**. Il codice sorgente, una volta compilato in formato oggetto, viene sottoposto alla fase di linking, dove vengono collegate le librerie necessarie e a questo punto viene prodotto l'eseguibile il quale viene caricato in memoria nel momento in cui vogliamo eseguirlo. Bisogna che il sistema operativo effettui un **binding**, una corrispondenza tra quelli che sono gli indirizzi fisici effettivi e quelli logici che usa il programmatore. Questo binding avviene in 3 modi diversi:

1. A tempo di compilazione o assemblaggio se la posizione in memoria del programma è nota a priori allora può essere generato un **codice binario assoluto** in questo caso allora gli indirizzi logici coincidono con quelli fisici.
2. A tempo di caricamento, in particolare in questo caso si parla di **rilocazione statica**, il compilatore genera degli indirizzi relativi che poi vengono convertiti in indirizzi fisici dal loader (caricatore). Se si deve spostare il programma in un'area diversa di memoria allora va ricaricato.
3. A tempo di esecuzione, in questo caso si parla di **rilocazione dinamica**, il programma viene caricato nella sua forma **rilocabile** il binding avviene quando l'eseguibile viene messo in esecuzione ed è a carico della **MMU** (Memory Management Unit).

In entrambi i casi 2 e 3 l'eseguibile vinario si dice **rilocabile**.

Fino ad ora abbiamo preso in considerazione casistiche di linking statico, ovvero in cui il linking con le librerie, che avviene subito dopo la compilazione crea un eseguibile finale con al suo interno tutto quello che serve per eseguirlo. Una funzionalità molto importante e utile che il binding a tempo di esecuzione rende possibile è il **linking dinamico**, ovvero l'uso di librerie dinamiche (file .dll in Windows .so in Unix) in questo caso le librerie vengono "linkate" solo al momento del bisogno durante l'esecuzione del programma portando i seguenti vantaggi:

- Dimensioni ridotte degli eseguibili
- Facile aggiornamento delle librerie senza dover ricompilare interamente i programmi che le utilizzano
- Ottimizzazione della memoria: se una procedura di una libreria è utilizzata da più processi possiamo tenerne una sola copia in memoria e renderla disponibile a tutti i processi che la richiedono.

Come avviene il linking dinamico?

Nel caso in cui il sistema operativo supporti il linking dinamico, piccole porzioni di codice, dette **stub**, vengono inserite dove necessario e permettono di localizzare la routine appropriata nella libreria residente in memoria. Lo stub rimpiazza se stesso con l'indirizzo della routine e la esegue. Il sistema operativo deve verificare se la routine si trova già nello spazio di indirizzamento del processo e, in caso negativo, provvedere a renderla disponibile.

Uno degli **svantaggi** introdotti dal linking dinamico è che gli stub potrebbero potenzialmente rimandare a codice malevolo.

2. Punto di vista del Sistema Operativo

Come abbiamo capito nella sezione precedente ci interessa in particolare analizzare la gestione della memoria nel caso di programmi **rilocabili** a run-time. La memoria centrale è una delle risorse più importante (dopo la cpu) di un sistema, in particolare se multiprogrammato. Il gestore della memoria si occupa di:

1. Allocare la memoria fisica ai processi che ne fanno richiesta
 - o Un processo per eseguire **deve** avere una certa quantità di memoria, facciamo riferimento alla memoria allocata per un processo come il suo **spazio di indirizzamento**
2. **Isolamento** degli spazi di indirizzamento proteggere lo spazio di indirizzamento di ogni processo da sconfinamenti voluti o erronei da parte di altri processi
3. Consentire la **condivisione** di aree di memoria fra processi interagenti (in ambito globale) o fra processi che utilizzano gli stessi servizi

3. Modalità di gestione della memoria

Vediamo ora quali sono i diversi approcci riguardanti la gestione della memoria:

1. Nessuna politica: Applicabile solo nel caso di programmi assoluti o rilocabili staticamente.
2. Politiche di allocazione contigua: ogni entità di un processo occupa locazioni **contigue** dello spazio **fisico** di memoria, gli indirizzi fisici sono quindi consecutivi.
3. Politiche di allocazione non contigua: una entità di un processo può essere allocata in locazioni **non contigue** dello spazio fisico di memoria.

I criteri di valutazione di cui teniamo conto quando valutiamo una politica di gestione della memoria sono i seguenti:

1. Spreco di memoria, risolvibile tramite la **frammentazione**
2. Sovraccarico temporale: complessità computazionale delle operazioni di allocazione/deallocazione della memoria
3. Sovraccarico nelle operazioni di accesso alla memoria: durata delle operazioni aggiuntive in rapporto al tempo di accesso fisico

3.1 Politiche di allocazione contigua

Iniziamo ad analizzare le politiche di allocazione contigua, i primi esempi sono giusto cenni storici non più utilizzati nei sistemi operativi moderni.

3.1.1 Monitor Monoprocesso

Abbiamo già visto questa politica in precedenza, è la politica che veniva utilizzata in MS-DOS, sistema operativo monoprogrammato. La memoria veniva suddivisa in 2 aree contigue, la prima era adibita alla porzione del sistema operativo denominata **monitor** ovvero quella parte che doveva rimanere sempre in esecuzione, mentre il resto della memoria era allocato per i processi

(del sistema operativo e non). La gestione di questa memoria avveniva in maniera molto semplice: il sistema operativo teneva traccia della prima e ultima locazione disponibile per allocare i processi, la parte di monitor veniva allocata alla testa o coda (parte alta o bassa) della memoria in base alle configurazioni degli interrupt hardware, che in genere facevano parte del sistema operativo. Altre parti del S.O. in particolare il loader e l'interprete dei comandi venivano allocati all'estremo opposto e sovrascritti se necessario. Un grosso problema di questo tipo di politica è la **protezione del monitor** da accessi indesiderati, risolta tramite uno dei seguenti modi:

1. Impostare la porzione del monitor in **sola lettura**: questo metodo non era particolarmente utilizzato perché così facendo non si può aggiornare il S.O.
2. Utilizzo di appositi **registro barriera** (fence register): ogni indirizzo generato dal programma in esecuzione viene comparato con il valore del fence register, se tale indirizzo risulta maggiore (al di fuori dello spazio di memoria riservato al monitor) allora l'accesso viene permesso, altrimenti no.
3. Associazione di un **bit di protezione** ad ogni locazione di memoria, settando a 1 quelli non accessibili (zona del monitor) e a 0 quelli accessibili.

Le soluzioni 2 e 3 hanno senso solo nel caso in cui il sistema operativo possa fare la distinzione tra le modalità di funzionamento **utente** e **supervisore**.

3.1.2 Partizionamento statico della memoria

Approccio molto simile al precedente con la differenza che questo viene applicato a sistemi **multiprocesso**, in sostanza quello che prima era un grande blocco unico di memoria adibito al processo in esecuzione sul sistema viene frammentato per poterne ospitare molteplici. Il partizionamento è **statico**, avviene al momento di caricamento del sistema operativo, e suddivide la memoria in un **numero fisso** di partizioni dalla **dimensione fissata**. In base al grado di multiprogrammazione, alla dimensione della memoria fisica e alla dimensioni tipiche dei processi, si decide che tipo di suddivisione applicare. Esiste una **tabella delle partizioni** dove il gestore della memoria tiene traccia dello stato delle partizioni, della loro dimensione e del loro indirizzo di partenza. Per la creazione di un processo è necessario trovare una partizione libera che ha dimensione sufficiente per contenere il codice e i dati del programma, nel descrittore del processo (PCB) includeremo dunque anche la partizione di memoria che gli è stata assegnata. A questo punto per effettuare il **binding** tra gli indirizzi logici e quelli fisici basta che il **registro base** detto anche di rilocazione venga caricato con il valore contenuto nel descrittore del processo, ovvero l'indirizzo di partenza della zona di memoria in cui si trova il programma. La Memory Management Unit a questo punto non farà altro che calcolare gli indirizzi fisici come la somma del valore nel registro base più l'indirizzo logico. Quando un processo termina invece, bisogna liberare la memoria, il sistema operativo cambierà quindi lo stato di quella partizione da allocata a libera. I **problem**i di questa politica sono i seguenti:

1. Strategia di allocazione delle partizioni: bisogna stabilire una strategia nel caso in cui ci fossero più partizioni libere che soddisfano i requisiti del nostro processo per decidere quale allocare.
 1. Metodo **first-fit**: scelgo la prima partizione libera che soddisfa i requisiti, questa opzione è più veloce.
 2. Metodo **best-fit**: scelgo la più piccola disponibile, questa opzione spreca meno memoria e causa una **minore frammentazione**.
2. Strategia in caso di non avere partizioni adatte
 1. Non c'è nessuna partizione ne libera ne occupata che può contenere il processo, questo caso può essere gestito come:
 1. **Errore progettuale**, serve ridefinire le partizioni.

2. Tramite l'uso di tecniche di **overlay**: si sovrappongono nella stessa partizione parti dello stesso programma assoluto.
2. Non c'è nessuna partizione libera
3. Non c'è nessuna partizione libera che può contenere il processo

I casi ii e iii, invece, possono essere gestiti nei seguenti modi:

1. **Aspettare** che si liberi una partizione adeguata, **N.B.** si puo interferire con le scelte fatte dallo scheduler
2. Costringere un processo a lasciare libera una partizione adeguata: lo **swapping**.

Swapping

Lo **swapping out** consiste nella **rimozione dalla memoria** di processi (di solito sospesi) per essere messi nel backing store (partizione disco ad accesso rapido), dal quale in seguito vengono portati nuovamente in memoria per proseguire l'esecuzione **swapping in**. Questa operazione permette di avere uno spazio logico della memoria che risulta superiore a quello fisico. Possiamo quindi fare le seguenti considerazioni riguardo allo swapping:

- La maggior parte del tempo di swap è dovuta al trasferimento di dati (swap in/out). Queste operazioni risultano **lente** in quanto accedono alla memoria secondaria; bisogna valutare se l'utilizzo di queste tecniche è conveniente o meno.
- Il S.O. mantiene una coda dei processi che sono stati spostati nel backing store.
- Lo swapping è utilizzato in molti S.O. attuali.
- Non tutti i processi possono essere swappati, i processi critici ad esempio non sono swappabili. Il S.O. deve essere quindi in grado di marcare alcuni processi come non swappabili. Per far sì che lo swapping risulti utile bisogna che il numero di processi "fissati" in memoria sia limitato e che tali processi possano essere allocati solo da utenti privilegiati e non da chiunque abbia accesso al sistema.
- La **associazione** fra processo swapped out e partizione può essere di due tipi:
 1. **statica**: il processo deve tornare alla partizione di residenza precedente allo swap, questo è necessario nel caso in cui il codice sia assoluto o rilocabile solo staticamente.
 2. **dinamica**: il processo può essere rilocato in una qualunque partizione, questo è possibile se il codice è rilocabile dinamicamente, è sufficiente modificare il valore del registro base.
- Potrebbero esserci problemi nel caso in cui ci siano delle operazioni di I/O pendenti, in questi casi possiamo scegliere di non usare lo swapping o di usare aree di I/O del S.O.

Protezione

Bisogna garantire l'isolamento degli spazi di indirizzamento dei processi:

- Protezione del S.O. nei confronti degli altri processi
- Protezione di ogni processo nei confronti degli altri processi, possibile solo tramite un **meccanismo hw**

Possibili soluzioni:

1. Se a livello hw oltre il registro base c'è anche un **registro limite** il cui valore sarà posto uguale al più alto indirizzo **logico** utilizzato nel programma. Anche il valore di questo registro deve essere salvato nel descrittore del processo.
2. **Diritti di accesso**: Un insieme di bit associati a blocchi di memoria, questo approccio consiste nell'avere dei bit che vengono riservati per specificare il diritto di accesso delle partizioni, con il grosso limite che con un numero limitato di bit posso salvare le informazioni di un numero limitato di partizioni.

Condivisione

Questo concetto è opposto a quello di protezione appena visto. Come mettiamo in condivisione tra processi le risorse nel caso del partizionamento statico?

1. Entità condivise nel S.O.:
 - vantaggio: semplicità
 - svantaggio: crescita del S.O. e staticità
2. Copie multiple della entità che deve essere condivisa
 - vantaggio: metodo diretto
 - svantaggio: costoso in termini di tempo, problemi in caso di processi swapped out e grosso spreco di memoria
3. Allocare una **partizione dedicata** e comune:
 - vantaggio: metodo intuitivo
 - svantaggio: ogni accesso viene considerato una violazione in termini di protezione
(soluzione: modifica temporanea delle chiavi, insiemi separati di registri base e limite)

Conclusioni sul partizionamento statico

Come abbiamo visto il partizionamento statico è un metodo **semplice**, necessita un modesto contributo HW, è adatto a **ambienti statici**, il suo problema più critico è la **frammentazione interna**, ovvero la zona di memoria residua che si viene a creare quando un processo ha una dimensione inferiore a quella della partizione che gli viene allocata. Questo frammento che si viene a creare è **non utilizzato** e **non utilizzabile**. Questo tipo di problematica in alcuni casi si fa molto sentire, ad esempio quando esistono processi molto grandi che non vengono eseguiti di frequente. Un altro problema di questo tipo di politiche è che la necessità di memoria deve essere **nota staticamente**, andando a limitare operazioni relative alla crescita dello stack. Infine è evidente che il **numero fisso** di partizioni limita il grado di multiprogrammazione, una soluzione parziale è quella dello swapping che però in termini I/O è molto costosa.

3.1.3 Partizionamento Dinamico della Memoria

Vediamo ora come modificare il precedente approccio per migliorarlo, in particolare vedremo come far sì che la suddivisione della memoria avvenga in base alle esigenze dei processi in **modo dinamico**. Il **partizionamento dinamico** consiste nell'avere un **numero variabile** di partizioni con **dimensioni variabili**. Questo tipo di approccio di gestione della memoria ci consente di effettuare **MVT**: Multiprogramming with Variable number of Tasks. In questo caso non si parla più solo di partizioni, con partizioni infatti ora faremo riferimento alle aree di memoria allocate, mentre chiameremo il resto **memoria libera**, viceversa quando la partizione termina di essere utilizzata viene liberata e torna ad essere libera. Il gestore della memoria a questo punto deve mantenere una **tavella di descrizione delle partizioni** come in precedenza ma con la differenza che anziché tenere traccia di tutte le zone di memoria terrà traccia solo delle partizioni, ovvero delle zone di memoria allocate. Le zone di memoria libere sono salvate all'interno di una apposita lista, questa **lista di aree di memoria libera** ha una struttura ben precisa, innanzitutto il puntatore alla testa di questa lista risiede nel S.O. mentre il resto dei suoi elementi risiedono in memoria centrale, proprio nelle zone libere a cui si riferiscono. Ogni elemento di questa lista contiene un puntatore al prossimo elemento (linked list, può essere anche double linked), e informazioni riguardo alla dimensione dell'area libera a cui fanno riferimento.

Esempio

Supponiamo di avere la seguente suddivisione della memoria:

0 100 400 500 750 900 1000

[S.O. | ... | Pi | Pj | Pk | ...]

Come possiamo vedere abbiamo il S.O. negli indirizzi da 0 a 100, poi libero da 100 a 400, 3 processi contigui Pi, Pj, Pk e infine di nuovo una zona non allocata da 900 a 1000. Come detto in precedenza il puntatore alla testa della lista si trova nel S.O. e punta in questo caso all'indirizzo 100, qua troveremo una semplice struttura dati, ad esempio una `struct`, che contiene un intero e un puntatore: `[900, 300]` l'intero ci indica la dimensione della zona libera ovvero 300 indirizzi, mentre il puntatore punta il successivo elemento della lista, che è anche la successiva area libera di memoria ovvero 900. All'indirizzo 900 troveremo la seguente struttura dati: `[-, 100]` perché la zona non allocata è spazia per 100 indirizzi ed è l'ultima della lista quindi ha il puntatore settato a `NULL`.

Creazione di un processo

Vediamo ora come avviene l'allocazione di memoria nel caso della creazione di un processo. Supponiamo che il processo richieda 120KB, il gestore della memoria va a cercare all'interno della tabella di descrizione delle partizioni una partizione libera sufficientemente grande per accogliere il processo, quando trova tale partizione alloca il processo e effettua le opportune modifiche alla lista di aree di memoria libera ridimensionando la partizione che ora è occupata dal nuovo processo. Facendo riferimento all'esempio di prima, se questo nuovo processo fosse stato allocato all'indirizzo 100 allora la struct a quell'indirizzo sarebbe stata cambiata a `[900, 180]`. Vediamo nello specifico l'**algoritmo di allocazione** della nuova partizione:

1. Si cerca nella lista delle aree libere un'area con dimensione superiore o uguale a quella richiesta dal processo. Se non esiste si può:
 - o aspettare che si liberi
 - o costringere un processo a lasciare libera un'area adeguata → **swapping**
2. Una volta trovata la partizione si effettua la differenza tra la dimensione richiesta e quella trovata, in base al confronto di questa differenza con una costante c:
 - o `diff <= k`: si alloca tutta la partizione
 - o `diff > c`: si crea una partizione e il resto rimane memoria libera

Strategia per la costante c:

- o Valore minimo: dipende dalle dimensioni delle informazioni da mantenere per la lista, lo spazio restante deve essere abbastanza per contenere la `struct` di cui si parlava prima.
 - o Valore massimo: dipende da scelte di progetto, determina una minima frammentazione interna la cui entità può essere trascurata.
3. In una riga libera della tabella si memorizzano i dati della nuova partizione
 4. Nel descrittore di processo si registra il numero di riga della tabella

Strategia di selezione dell'area libera

1. Metodo first-fit, o la sua variante next-fit, sono i metodi più popolari:
 - o first-fit: si sceglie la prima area libera che soddisfa i requisiti
 - o next-fit: si memorizza il puntatore all'area libera dopo una allocazione e la prossima ricerca parte da lì
2. Metodo best-fit: si sceglie la area libera più piccola che soddisfa i requisiti, più lento ma spreca meno memoria, crea una minore frammentazione esterna
3. Metodo worst-fit: si sceglie la area libera più grande che soddisfa i requisiti, tenta di non creare piccole aree per ridurre lo spreco di memoria

Algoritmo di Deallocazione

Vediamo ora cosa avviene nel momento in cui il gestore della memoria deve liberare una partizione:

1. Nel descrittore del processo bisogna individuare il numero di riga della tabella e si marca come deallocata
2. Si restituisce la partizione alla lista delle aree libere **unificandola** se possibile con aree libere adiacenti
 - o Essendo la lista ordinata per indirizzi questa operazione risulta piuttosto semplice, infatti basta cambiare all'interno della struct la dimensione della nuova partizione liberata. Se la lista è doppiamente linkata questo algoritmo diventa molto più immediato.
3. Si cancella la riga della tabella corrispondente alla partizione deallocateda

Frammentazione Esterna

La frammentazione esterna è un termine con il quale si indica quelle aree di memoria libere che si vengono a creare tra le partizioni. Allocando e deallocando la memoria queste aree che si vengono a creare possono diventare talmente piccole e **frammentate** da non essere sufficienti a soddisfare richieste di allocazione, anche se globalmente lo spazio in memoria c'è. La frammentazione viene ridotta quando una partizione liberata può essere inglobata in un area libera adiacente, detto ciò non sempre è possibile deallocare aree di memoria vicine quindi man mano che il tempo passa e la memoria viene allocata e deallocated la frammentazione aumenta sempre di più. Una possibile **soluzione** è la **compattazione**, ovvero compattare tutte le piccole aree frammentate in un'unica area. Il **problema** di questa soluzione chiaramente è che bisogna sospendere tutti i processi coinvolti per rilocarli in modo dinamico e una volta rilocali bisogna aggiornare la lista delle aree libere e la tabella delle partizioni. La compattazione può essere fatta appena possibile, ad esempio ogni volta che si libera una partizione, oppure solo quando è indispensabile, ad esempio quando non si riesce ad allocare spazio per un processo.

Nota Bene

Non tutti i processi possono essere rilocali in modo dinamico, ad esempio i processi con I/O pendenti non possono essere spostati; se il gestore della memoria non vuole essere vincolato a questo aspetto, si deve garantire che l'I/O avvenga solo nello spazio kernel (approfondiremo questo argomento in seguito).

Vediamo ora quali sono i 2 metodi principali che possiamo applicare quando andiamo ad effettuare la compattazione:

1. Compattazione incrementale e selettiva: si va ad applicare il **minimo** numero di spostamenti che soddisfano i nostri requisiti, ovvero una area di memoria contigua di dimensione d.
2. Compattazione globale: si spostano tutti i processi da un lato della memoria, ad esempio quello iniziale, creando un'unica grande area contigua libera.

Osservazioni e Conclusioni sul partizionamento dinamico

1. La **traduzione** da indirizzo **logico** a **fisico** viene ottenuta analogamente al partizionamento statico (uso del registro base o registro di rilocazione)
2. Le problematiche di **protezione** e di **condivisione** sono analoghe al caso di partizionamento statico e quindi anche le soluzioni (per protezione uso del registro limite).
3. Richiede lo stesso hw del caso statico
4. Metodo adatto ad ambienti dinamici

5. Non esiste più (o è molto limitato) il problema della **frammentazione interna**
6. Tutta la memoria libera può essere allocata ad un processo
7. Un processo potrebbe avere dinamicamente bisogno di più memoria rispetto a quella allocata inizialmente, **soluzioni:**
 1. Si fissa la costante **c** in modo che al processo viene allocata più memoria di quella strettamente necessaria all'inizio → la frammentazione interna potrebbe però non essere più trascurabile se poi quella memoria il processo non la usa
 2. Si interviene solo quando necessario:
 - Se il processo è allocato vicino ad un'area libera → si ingloba tale area libera adiacente
 - Il processo viene spostato a run-time in un'area di memoria libera più grande
8. Richiede una **gestione più complessa**
9. Il problema più critico è la **frammentazione esterna**, risolvibile tramite la **compattazione**
10. Metodo misto: partizionamento statico e dinamico → statico per allocare parti del sistema che risultano critiche; dinamico per allocare processi utente.

3.1.4 Segmentazione

Questo metodo è una via di mezzo tra politica di allocazione contigua e quella non. **Dal punto di vista del S.O.** la frammentazione esterna (problema critico del partizionamento dinamico) si può ridurre se si riducono le **dimensioni** delle zone da allocare. Se allochiamo zone piccole allo stesso tempo lasciamo libere zone più grandi ed è quindi difficile imbattersi nel caso in cui non ci sia una partizione libera sufficientemente grande. La soluzione consiste quindi nel suddividere i programmi in parti: **segmentare** i processi, da qui il nome **segmentazione**. **Dal punto di vista esterno** la segmentazione è uno schema di gestione della memoria che asseconda la visione utente. L'utente generalmente non pensa alla memoria come ad un array lineare di byte → Un programma è una collezione di segmenti, ogni **segmento** è un'unità logica cioè una raccolta di entità che sono **logicamente correlate**. Il compilatore crea un **segmento** separato (con dimensioni diverse) per ogni componente del programma, ad esempio: un segmento per il **codice** uno per i **dati** e uno per lo **stack**. Ogni **segmento** viene allocato in modo **contiguo**, un programma nel suo complesso non ha tutte le sue entità allocate in modo contiguo. Per questo motivo la segmentazione è una **via di mezzo** tra allocazione contigua e non contigua.

All'interno del singolo segmento gli indirizzi **partono da zero**, all'interno di un programma ogni entità viene identificata dalla coppia `[segmento, offset]`. Al momento del caricamento di un programma segmentato **bisogna** allocare ogni segmento → si procede in modo analogo al caso di partizionamento dinamico → bisogna trovare una zona di memoria libera di dimensione sufficiente ad allocare ogni singolo segmento (lista delle aree libere e metodi first-fit, next-fit, best-fit o worst-fit). Per ogni segmento si ha un **descrittore di segmento** in cui si memorizza l'indirizzo fisico iniziale (indirizzo base) e la dimensione del segmento → **tabella dei descrittori di segmento (TDS)** con l'accortezza che **deve** esserci una TDS per ogni processo attivo nel S.O. Questa tabella dei segmenti è fondamentalmente un vettore di coppie di registri Base e Limite → La dimensione di ogni segmento serve per controllare che i riferimenti siano tutti all'interno del segmento specificato (no accessi out of range).

Osservazioni

1. Dal punto di vista del S.O. la **segmentazione** è simile alla gestione della memoria partizionata dinamicamente → la differenza è che l'unità di allocazione è il segmento e che abbiamo più coppie basi e limiti (una per ogni segmento).
2. Le TDS devono essere modificate ogni volta che c'è uno swap o una rilocazione per esigenze di compattazione.
3. La dimensione di ogni TDS dipende dalle dimensioni dello spazio logico di un processo.

Realizzazione della TDS

- mantenere la TDS sui registri della cpu ha alcuni svantaggi:
 - bisogna limitare la dimensione della TDS
 - overhead al momento del process switching
- Se le TDS hanno grosse dimensioni, allora non possono essere memorizzate nei registri macchina, ma devono essere allocate in memoria; in questo caso, le TDS fanno parte di un **segmento di memoria speciale** → quindi c'è bisogno di un registro hw che punta alla base della TDS del processo corrente detto **registro base della TDS**. Dato che le dimensioni di una TDS possono variare, si deve usare un altro registro hw dedicato detto **registro limite** della tds. N.B. I valori dei registri appena menzionati **devono** essere salvati nel **descrittore del processo**.

A questo punto sorge un **problema**: nel secondo caso infatti, un accesso ad un dato **implica** due accessi in memoria: per accedere alle informazioni della TDS e poi per accedere alla locazione fisica effettiva. Questo causa un **overhead** nella **traduzione** degli indirizzi da logici a fisici, che **non è** accettabile. Per porre rimedio a questo problema possiamo ricorrere a 2 soluzioni:

1. Una **cache** (TLB) su cui caricare le sole informazioni sui segmenti usati più recentemente (approfondiremo questo discorso con la memoria virtuale)
2. **Registri particolari** detti registri **di segmento** su cui caricare le informazioni sui segmenti utilizzati **più di frequente**.

Protezione

Fra processi **diversi** → analoga al caso di partizionamento dinamico: uso dell'indirizzo base e della dimensione del segmento. Inoltre, è possibile avere un livello di protezione anche all'interno dello stesso processo. Un ulteriore strato di protezione può essere aggiunto tramite la tipizzazione dei segmenti di codice, dati e stack e tramite i **diritti di accesso** per ogni tipo di segmento:

- Per segmenti di tipo stack sia lettura che scrittura
- Per segmenti di tipo codice solo esecuzione o lettura
- Per segmenti di tipo dati o solo lettura o solo scrittura oppure entrambe

In questo modo possiamo prevenire errori dovuti a tentativi di esecuzione di dati e crescita eccessiva dello stack con conseguente sconfinamento in aree codice o dati adiacenti. Le informazioni dei diritti di accessi vengono stabilite tramite degli appositi bit e inseriti nel descrittore di segmento.

Pro e Contro

Vediamo ora vantaggi e svantaggi della segmentazione partendo dai primi:

- Flessibilità
- Facilità di condivisione
- Condivisione:
 - Le informazioni da condividere sono inserite in **segmenti** dedicati e separati.
 - Ogni processo che deve accedervi avrà nella propria TDS un **descrittore per quel segmento**, possibilmente avendo anche accessi diversi

Gli svantaggi della segmentazione si presentano principalmente nel caso dello swap: il S.O. deve accorgersi se un segmento condiviso da un processo swapped in è già in memoria. Infine rimane ancora il discorso della **frammentazione esterna** anche se ridotta rispetto al partizionamento dinamico. Vedremo ora come risolvere queste problematiche tramite le politiche di allocazione

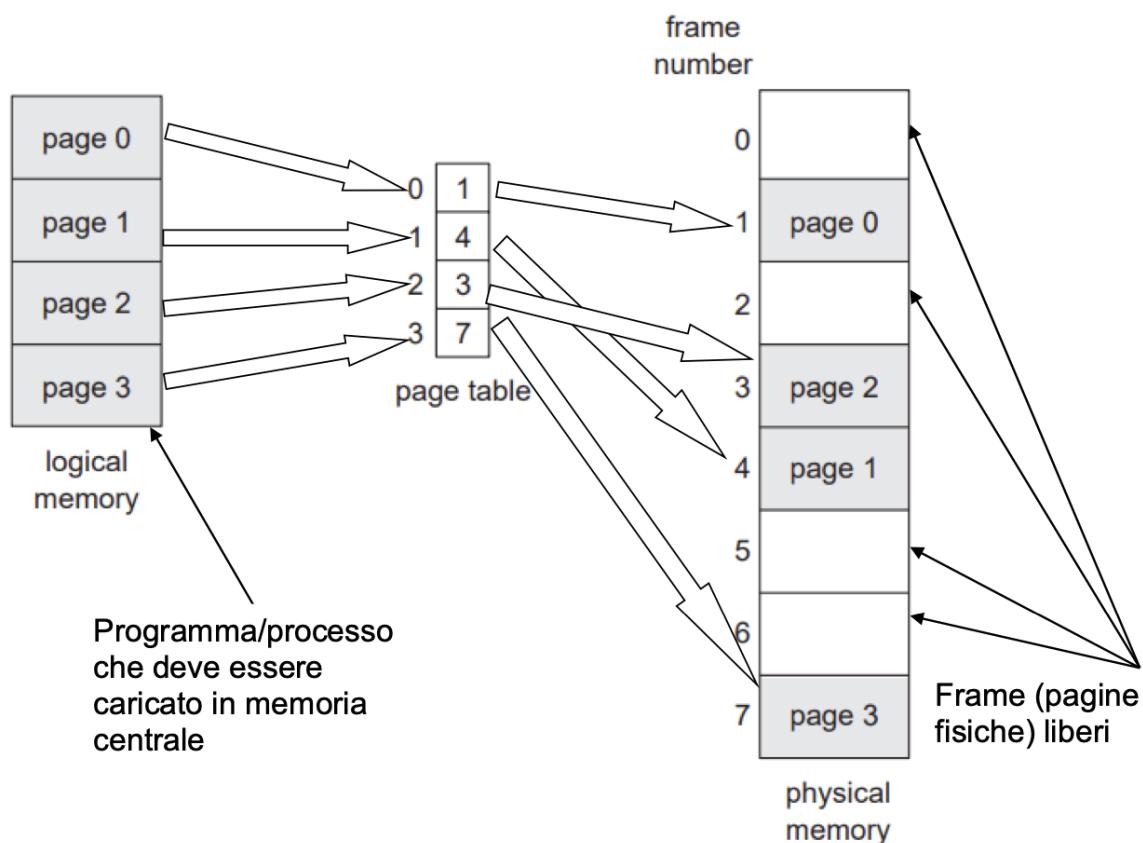
non contigua.

3.2 Politiche di allocazione non contigua

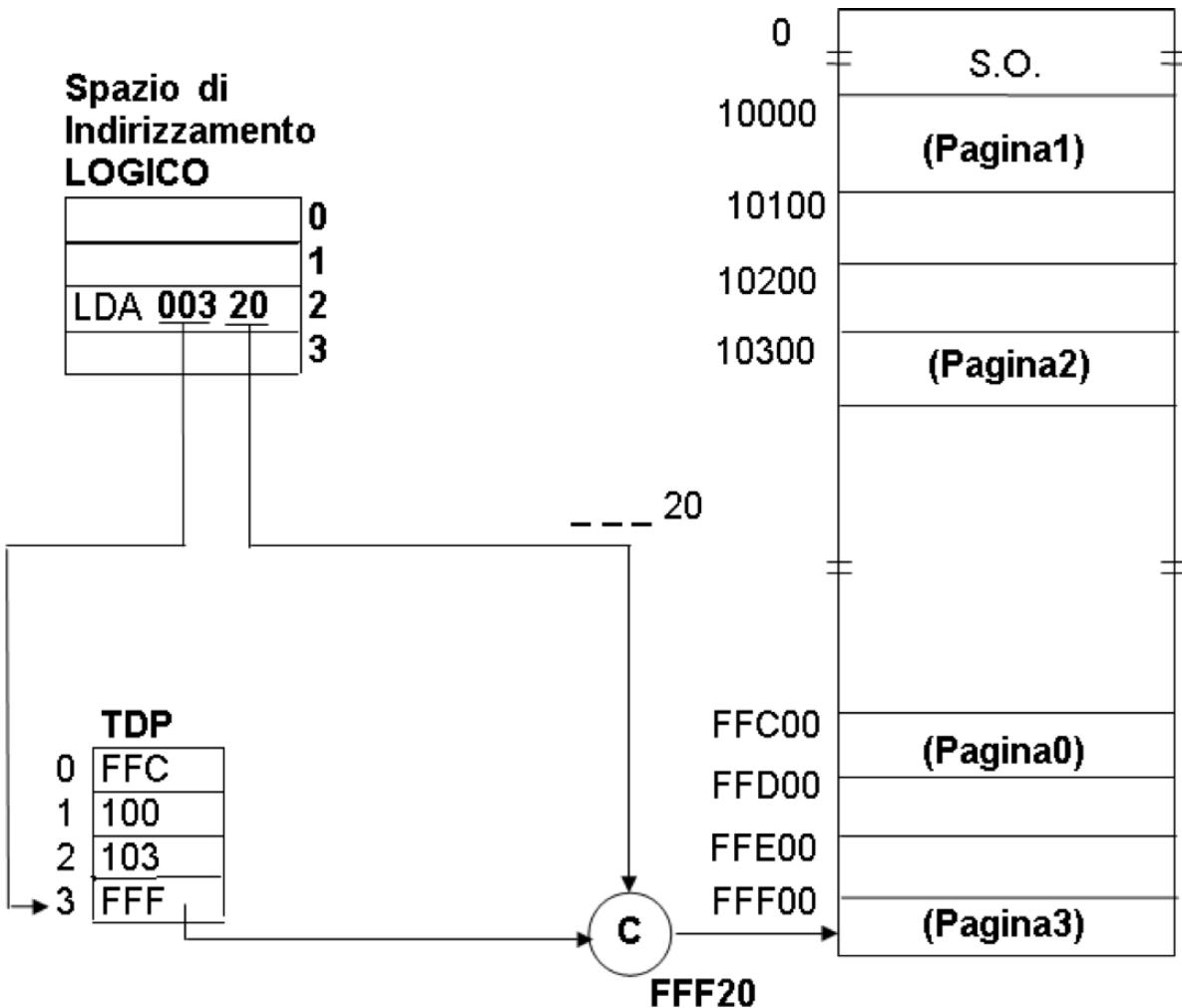
Le entità di un processo possono essere allocate in **locazioni non contigue** dello spazio fisico di memoria → approccio opposto rispetto a quello delle politiche di allocazione contigua.

3.2.1 Paginazione

La paginazione è uno dei concetti base delle politiche di allocazione non contigua. La **memoria fisica** viene **suddivisa** in un certo numero di **blocchi** di dimensione fissa chiamati **pagine fisiche** detti anche frame, la cui dimensione è una potenza di 2. Lo spazio fisico diventa quindi un insieme di pagine di dimensione fissata. Anche lo **spazio logico** di ogni processo viene suddiviso in blocchi di dimensioni fissa detti anche **pagine logiche**. Le pagine fisiche e quelle logiche hanno **la stessa dimensione**. A questo punto bisogna tenere traccia di tutti i frame liberi → l'allocazione di un processo consiste nel trovare un numero di pagine fisiche libere pari al numero di pagine logiche di cui necessita. Le pagine fisiche che servono per allocare un processo possono essere **non fisicamente contigue**. Si ha solo frammentazione interna (relativa all'ultimo frame) → mediamente il 50% dell'ultima pagina fisica allocata! Il sistema di traduzione dell'indirizzo si basa su una **tavella delle pagine** (TDP) **per ogni processo**. Vediamo un esempio di questa tecnica: supponiamo di avere un processo di dimensione pari a 4 pagine logiche (numerate da 0).



Una **TDP** ha tante righe quante sono le pagine logiche: il contenuto di ogni riga è il numero della pagina fisica (frame) dove è stata allocata la corrispondente pagina logica. Ogni indirizzo logico generato dalla cpu è suddiviso automaticamente in un **numero di pagina** **p** e un **offset** **o** all'interno della pagina. **p** serve da indice nella page table per trovare il numero del frame contenente la pagina logica. Vediamo ora come avviene la traduzione, effettuata dalla MMU, da indirizzo **logico** a indirizzo **fisico**. Consideriamo un sistema che usa indirizzi logici e fisici di 20 bit, con pagine da 256 byte (quindi un offset di 8 bit) e infine 12 bit per il numero di pagina.



La numerazione nell'immagine è esadecimale, vediamo come a partire dalla coppia di valori `<page number, offset>` numero pagina, offset pari a `<003, 20>` si passa attraverso la TDP per recuperare l'indirizzo fisico della pagina `FFF` e quindi vi si aggiunge l'offset per arrivare all'indirizzo fisico `FFF20`. Dato che la dimensione della pagina fisica e di quella logica coincidono, il valore dell'offset **rimane invariato** nel passaggio da fisico a logico. Quindi il contenuto della **TDP** sono i 12 bit più significativi dell'indirizzo fisico a cui comincia la pagina → se il numero di pagina fisica è (in hex) `xxx` allora il suo indirizzo iniziale sarà `xxx00`.

Prime Osservazioni

- Per semplificare la traduzione, la dimensione delle pagine è sempre una potenza intera di 2. Inizialmente le **dimensioni tipiche** dei sistemi commerciali variavano da 256 byte a 4 KB, attualmente è compresa fra 512 byte e 1GB ma i valori più tipici sono compresi tra 8KB-4KB
- Il S.O. Può tenere traccia delle pagine libere e allocate tramite una **tabella della memoria** (TDM). Questa tabella avrà tante righe quante pagine, quindi $\frac{\text{dim\{fisica\}}}{\text{dim\{pagina\}}}$.
- Ogni volta che deve essere allocato un processo di dimensione $\text{dim\{processo\}}$ **bisogna allocargli un numero di pagine libere pari a $\frac{\text{dim\{processo\}}}{\text{dim\{pagina\}}}$** . Il S.O. **alloca sempre un numero intero** di pagine ad un processo. Se la $\text{dim\{processo\}}$ non è un multiplo esatto della dimensione della pagina allora l'ultima pagina fisica risulterà utilizzata solo parzialmente → **frammentazione della pagina**
- Non esiste la necessità di particolari strategie per trovare il numero sufficiente di pagine libere per un singolo processo dato che qualunque insieme di pagine va bene:
 - Non è necessario che siano contigue
 - Non ci sono problemi di dimensioni in quanto hanno tutte la stessa dimensione (uguale a quella delle pagine logiche)

- Non ha senso prendere in considerazione politiche first-fit o best-fit

Allocazione Delle Pagine

L'efficienza di una strategia a pagine dipende dalla velocità con cui si riescono ad individuare le pagine fisiche libere. Se utilizziamo la tabella della memoria vista prima e se supponiamo di avere una distribuzione casuale delle pagine libere, il numero medio x di righe della TDM che è necessario esaminare per trovare n pagine libere è

$$x = \frac{n}{q}$$

dove q è la probabilità che una certa pagina fisica sia libera, cioè

$$q = \frac{u}{100}$$

dove u è la percentuale di memoria non utilizzata; quindi, x è proporzionale a n secondo un fattore.

$$k = \frac{1}{q} \geq 1$$

In conclusione, il numero di righe x della TDM da analizzare cresce al crescere della memoria utilizzata (a parità di richiesta, cioè di n). In alternativa alla TDM possiamo collegare i numeri delle pagine fisiche libere in una **lista**. Ogni volta che abbiamo bisogno di n pagine, togliamo dalla lista i primi n numeri.

Deallocation Delle Pagine

Con la TDM, bisogna marcare come libere le n pagine che un processo libera, con la lista invece inseriamo all'inizio i numeri delle n pagine liberate. **In entrambi i casi**, il tempo che si impiega è proporzionale ad n

Ulteriori Osservazioni

- La gestione con la lista delle pagine libere presenta un **vantaggio** al momento della allocazione nei confronti della TDM → **non dipende** dal grado di utilizzo della memoria
- La gestione con la lista delle pagine libere presenta però uno **svantaggio** in termini di gestione di una struttura dinamica al posto di una statica come è la TDM → complessità temporale e spaziale

Realizzazione della TDP

Mantenere la tabella delle pagine sui registri della cpu ha alcuni svantaggi: bisogna limitare la dimensione della tdp e si viene a creare un overhead al momento del process switching. Invece se le tdp hanno grosse dimensioni bisogna allocarle in memoria, in questo caso abbiamo bisogno di un solo registro che punta alla base della tdp del processo corrente, detto **registro base** della tdp. Dato che le dimensioni di una TDP possono variare, dovrebbe esistere un altro registro hw dedicato detto **registro limite** della tdp oppure deve esistere un **bit di validità** che indica se la pagina è nello spazio di indirizzamento logico del processo (vedremo questo concetto meglio quando affronteremo la memoria virtuale). **N.B.:** I valori di questi due registri **devono** essere salvati nel **descrittore del processo**.

TLB

A questo punto individuiamo il seguente **problema**: se salviamo in memoria le TDP allora l'accesso ad **un** dato implica **due** accessi in memoria: uno per accedere alle informazioni della TDP e uno per accedere alla locazione fisica effettiva. Questo **overhead** nella traduzione degli indirizzi da logici a fisici **non è accettabile**. Per risolvere questo problema bisogna disporre come supporto hw di una **cache** delle pagine su cui caricare un **sottoinsieme** della TDP. Per questo supporto hw solitamente si impiega della **memoria associativa** ad alta velocità, detta anche **TLB** o (**Translation Look-aside Buffer**), il numero di elementi di una TLB è di norma compreso fra 64 e 1024. Alcune TLB memorizzano un Address Space Identifier, per ciascun elemento, così da identificare univocamente il processo cui appartiene la corrispondenza `<#pagina, #frame>`: l'ASID consente alla TLB di contenere, nello stesso istante elementi di diversi processi (approfondiremo questo argomento quando vedremo la memoria virtuale).

Gestione TLB

Inizialmente la TLB è vuota, mentre l'esecuzione procede, viene gradualmente riempita con indirizzi di pagine già accedute.

Hit-Ratio

definiamo **hit-ratio** la percentuale che indica quante volte una pagina viene trovata in TLB, dipende dalla dimensione della TLB, ad esempio per Intel 486 l'hit-ratio è 98%.

Se il numero di pagina non è presente nella cache delle pagine si parla di insuccesso della cache: **TLB miss**. Vediamo come si calcola il tempo di accesso effettivo (EAT), se indichiamo con α l'hit-ratio allora:

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

Dove T_{mem} è il tempo di accesso a memoria e T_{TLB} è il tempo di accesso alla TLB. Per far sì che l'uso della TLB sia conveniente è necessario che la **probabilità** che la pagina che serve si trovi nella cache sia molto alta. Si usano quindi **algoritmi probabilistici** per caricare nella cache le pagine che hanno la più alta probabilità di servire al processo nel prossimo futuro, come ad esempio **LRU** (Last Recently Used), approfondiremo questo tipo di algoritmi quando vedremo la memoria virtuale. Nel caso della segmentazione invece, la scelta di quali segmenti caricare nella cache può basarsi anche sulla informazione relativa al tipo dei segmenti cosa che non può avvenire nel caso della paginazione dato che la **suddivisione in pagine** viene effettuata in modo **automatico** e quindi una pagina non ha nessuna correlazione con le entità logiche che contiene. Una singola pagina infatti può contenere variabili, stack, codice...

Protezione

La protezione degli spazi di indirizzamento di un processo dagli altri viene garantita dal meccanismo di traduzione, grazie ai due registri hw menzionati in precedenza (o al bit di validità). La presenza di tabelle TDP potrebbe consentire **protezioni più granulari** → ad una pagina potrebbero essere associati dei diritti di accesso (r, w, x) questo meccanismo anche se sembra simile a quello introdotto per la segmentazione in questo caso è **meno flessibile** dato che la suddivisione in pagine è **completamente trasparente** al programmatore.

Condivisione

La **condivisione** è immediata in un sistema a pagine, i riferimenti alle pagine che devono essere **condivise** saranno **presenti in più TDP**; è il sistema che **deve** riconoscere e realizzare questa condivisione, visto che la paginazione è completamente trasparente al programmatore.

Organizzazione TDP

L'occupazione di memoria dovuta alle strutture dati (TDP) necessarie per gestire la paginazione può divenire eccessivamente grande!

Esempio

Consideriamo uno spazio di indirizzi logici a 32 bit, con pagine di 4KB; la tabella delle pagine potrebbe essere costituita da un milione di elementi. Se ciascun elemento occupasse 4 byte potrebbe risultare fino a 4MB di occupazione di memoria per una sola tabella delle pagine! Meglio evitare di collocare la tabella delle pagine in modo contiguo in memoria centrale e ricorrere a una di queste 2 soluzioni:

1. Paginazione gerarchica: suddividere la paginazione su livelli
2. Tabella delle pagine invertita

Paginazione gerarchica

Vediamo come funziona la paginazione gerarchica tramite un esempio che implementa questa soluzione con 2 livelli: Un indirizzo logico, in architetture a 32 bit con dimensione della pagina di 4KB, viene suddiviso in:

- Un numero di pagina a 20 bit
- Un offset all'interno della pagina di 12 bit

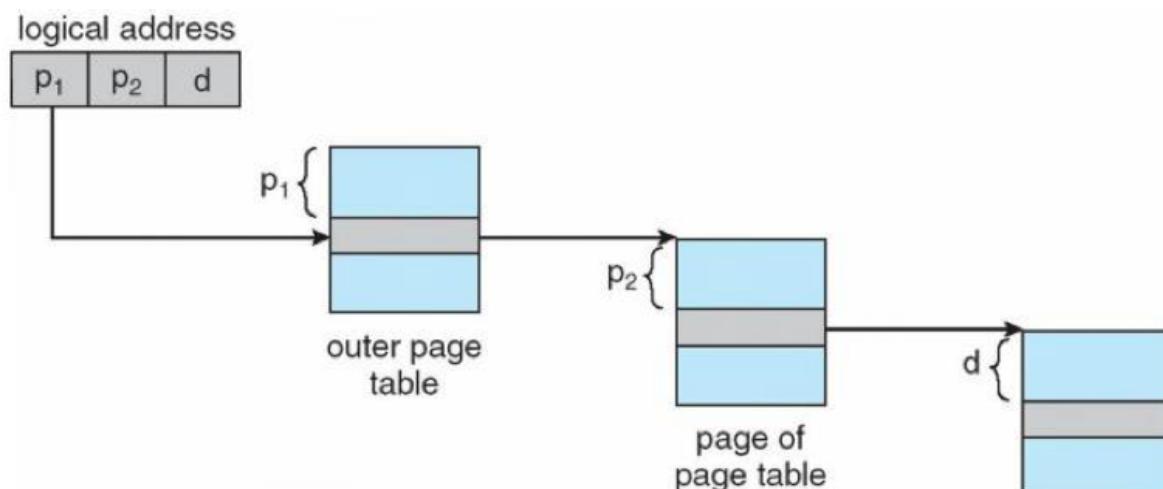
Dato che la tabella delle pagine è paginata il numero di pagina viene ulteriormente suddiviso in:

- Un numero di pagina di 10 bit (tabella esterna)
- Un offset di 10 bit (tabella delle pagine)

Pertanto un indirizzo logico sarà suddiviso in questo modo:

[n pagina		offset]
[p1		p2 d]
[10b		10b 12b]

Dove p1 sono i 10 bit che fanno da indice nella tabella delle pagine (esterna) detta anche page directory, e p2 sono i 10 bit che indicano l'offset all'interno della pagina della page table entry (interna). In questa immagine vediamo nello specifico lo schema logico di traduzione attuato dalla MMU, da indirizzo logico (generato dalla cpu) a quello fisico (senza introdurre la TLB che sarà invece essenziale).

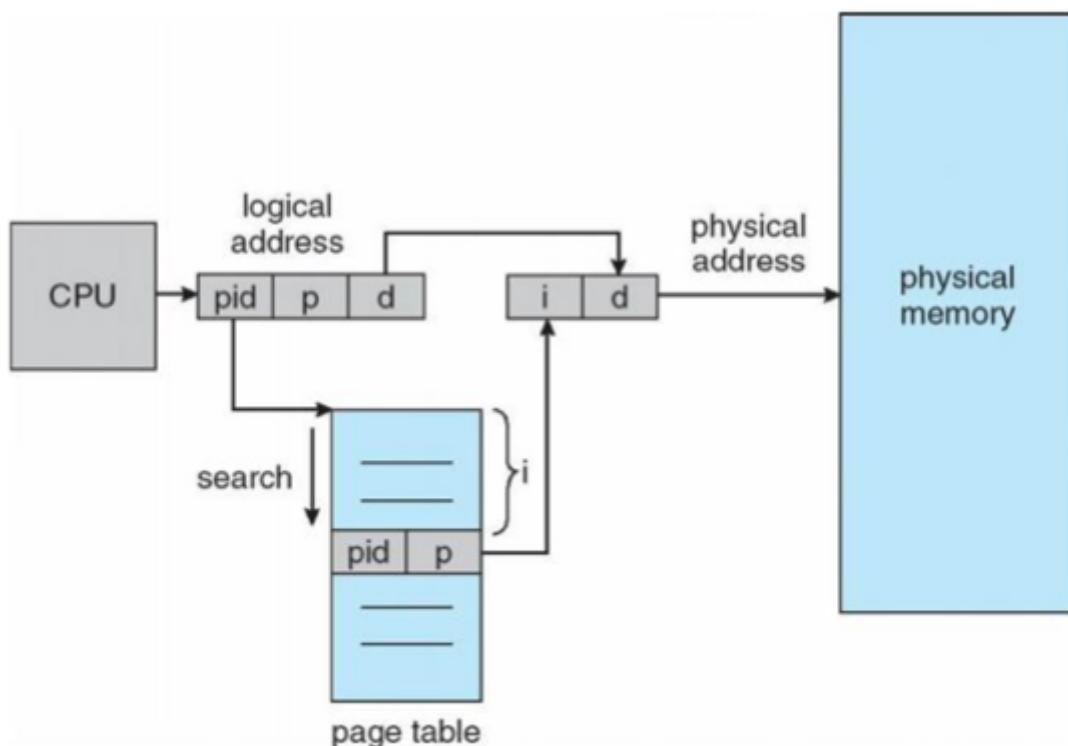


Dato che ciascun livello viene memorizzato come una tabella separata in memoria, la traduzione di un indirizzo logico nel corrispondente indirizzo fisico può richiedere **tre** accessi alla memoria: anche se il tempo richiesto per un accesso alla memoria sembra triplicato, la presenza di **TLB** consente di mantenere prestazioni ragionevoli. In pratica, solo alcune pagine della tabella delle pagine sono memorizzate in memoria, le altre sono su disco (come vedremo nella memoria virtuale). Lo schema di paginazione a due livelli non è più adeguato nel caso di sistemi con spazio di indirizzi a **64 bit** infatti: la tabella delle pagine più esterna avrebbe 2^{42} elementi, mentre quella interna delle pagine potrebbe essere costituita da 2^{10} elementi da 4 byte ciascuno. Si deve quindi optare per una soluzione a più livelli di paginazione (solitamente da 3 a 7).

Chiaramente l'aggiunta di livelli aggiunge tabelle in memoria e quindi il numero di accessi in memoria cresce. In conclusione, per le architetture a 64 bit la paginazione gerarchica è da considerarsi **inadeguata** a causa dei costi proibitivi di accesso alla memoria in caso di TLB miss. Per risolvere il problema nel caso dei 64 bit bisogna adottare la **tabella delle pagine invertita** che vedremo tra poco. Generalmente, si associa una tabella delle pagine ad ogni processo, che contiene un elemento per ogni pagina virtuale che il processo sta utilizzando, dando forma ad una rappresentazione naturale, dato che i processi si riferiscono alle pagine per mezzo di indirizzi logici ed il S.O. si occupa della traduzione in indirizzi fisici. Abbiamo quindi visto come il **problema** di questo approccio sta nel fatto che ogni tabella delle pagine può contenere milioni di elementi ed occupare molta memoria.

Tabella Delle Pagine Invertita

Questa soluzione consiste nell'utilizzare un'**unica** struttura dati globale che ha un elemento per ogni frame. Ogni elemento della tabella delle pagine invertita rappresenta un frame e, in caso di frame allocato, contiene: **[pid, p, d]** l'identificatore del processo a cui è assegnato il frame, il numero di pagina logica del processo e l'offset all'interno della pagina. Ecco come cambia quindi lo schema di traduzione attuato dalla MMU, da indirizzo logico a fisico:



Gli svantaggi di questo approccio sono 2:

1. Risulta difficile la realizzazione della **condivisione**, dovuta alla presenza di un solo elemento indicante la pagina logica corrispondente ad ogni pagina fisica.
2. Incremento del tempo necessario per ricercare nella tabella quando si fa un riferimento a pagine → è necessario ricercare su tutta la tabella → ricerca sequenziale → inefficiente!

Per ridurre il tempo di ricerca, da un punto di vista implementativo, si possono impiegare tabelle hash per limitare la ricerca ad uno, o al più a pochi, elementi della tabella, riducendo così la complessità da $O(n)$ a $O(1)$. In pratica, è necessario un meccanismo per gestire le collisioni quando diversi indirizzi logici corrispondono allo stesso frame. Ogni accesso alla tabella hash aggiunge comunque un riferimento alla memoria, per migliorare le prestazioni si usa una **TLB**.

Conclusioni sulla paginazione

- La paginazione è un processo gestito **interamente dal S.O.** diversamente dalla segmentazione
- Elimina il problema della frammentazione esterna, non servono algoritmi di compattazione
- Allocazioni e deallocazioni sono **semplici** e generano un sovraccarico modesto
- Il grado di utilizzo della memoria è molto elevato, se la dimensione della pagina è piccola: la dimensione della pagina piccola produce TDP troppo grandi
- Resta il problema della **frammentazione interna**, per ogni processo sull'ultima pagina allocata. Questa aumenta con l'aumentare della pagina.
- C'è necessità di hw dedicato molto sofisticato, abbiamo parlato ad esempio di **memoria associativa** per le TLB.

Differenze tra Segmentazione e Paginazione

Usando la paginazione lo spazio di indirizzamento logico è **lineare**, così come lo è quello fisico. Il punto di vista dell'utente è quindi **uguale** al punto di vista del S.O. (a meno della rilocazione). La paginazione è completamente **trasparente** al programmatore → l'utente specifica un indirizzo logico che è diviso automaticamente dall'hardware in due parti, un numero di pagina logica e offset all'interno della pagina. L'indirizzo fisico si ottiene da numero di pagina fisica combinato con l'offset. La dimensione delle pagine è fissa e uguale per quelle logiche e quelle fisiche: in genere una potenza di 2 → non è necessaria una politica di ricerca di pagine libere per allocare le pagine logiche. Entrambi gli approcci sono affetti dal problema della frammentazione interna. Dato che segmentazione e paginazione presentano vantaggi e svantaggi, un'idea è quella di combinarli assieme → **segmentazione con paginazione** vedremo questa tecnica nella memoria virtuale.

3.3 Memoria Virtuale

Abbiamo visto ripetutamente che un programma per essere eseguito ha bisogno di essere caricato in memoria, in realtà **non tutto** deve essere per forza in memoria.. Alcune parti quali porzioni di codice correlate alla gestione di errori, array liste e tabelle sovradimensionati rispetto all'utilizzo standard... Tutte queste casistiche occupano spazio **inutilmente**: non è necessario che esse vengano caricate in memoria per un corretto funzionamento del programma. Dobbiamo quindi stabilire una **classificazione ortogonale**:

1. L'intero spazio di indirizzamento logico di un processo **deve** essere in memoria perché il processo possa eseguire
2. **Non è necessario** che l'intero spazio di indirizzamento logico di un processo sia in memoria perché il processo possa eseguire

La **memoria virtuale** consente di eseguire **processi** che sono stati caricati **solo parzialmente** aumentando il grado di multiprogrammazione: la somma degli **spazi di indirizzamento virtuali** (o logici) di tutti i **processi attivi** in un sistema a memoria virtuale **può** superare la capacità della memoria fisica. In un **sistema a memoria virtuale** lo spazio di indirizzamento virtuale di un processo viene mantenuto in memoria secondaria (**Backing store**) e vengono caricate secondo necessità sono alcune parti in memoria centrale. Chiaramente tutti i dettagli della gestione della VM (virtual memory) sono completamente **trasparenti** al programmatore. Il programmatore ha l'**illusione** di avere a disposizione una memoria fisica che è maggiore di quella effettivamente

presente. Il **punto centrale** in questo tipo di gestione è che il S.O. deve sapere quali parti caricare in memoria centrale senza penalizzare l'esecuzione con lunghi tempi di attesa di caricamento da memoria secondaria. La VM può essere vista come **estensione** della paginazione e della segmentazione, quindi la traduzione da indirizzi logici a fisici può essere basata su tabella delle pagine e/o tabella dei segmenti. La **differenza** è che con la VM, alcune delle parti (pagine o segmenti) che costituiscono lo spazio di indirizzamento virtuale di un processo in esecuzione possono essere mancanti dalla memoria fisica. Nei sistemi a memoria virtuale l'hardware di traduzione (MMU) deve essere in grado di riconoscere se la parte richiesta si trova o meno in memoria reale. La parte mancante può essere una pagina o un segmento a seconda dello schema scelto. Principalmente faremo riferimento allo schema a pagine, nello specifico la **paginazione su richiesta**.

3.3.1 Paginazione su Richiesta

Di solito, la VM, è realizzata mediante tecniche di **paginazione su richiesta**:

- Tutte le pagine di ogni processo risiedono in memoria secondaria
- Durante l'esecuzione alcune pagine vengono trasferite in memoria centrale all'occorrenza:
su richiesta

Pager

Il pager è quella parte del S.O. che si occupa dei trasferimenti delle pagine da/verso memoria secondaria/centrale. Il pager non va confuso con lo **swapper** che si occupa di gestire i trasferimenti da/verso memoria secondaria/centrale di **interi** processi. Il pager è **lazy** cioè trasferisce in memoria centrale una pagina soltanto se ritenuta necessaria e quindi se viene acceduta.

3.3.2 Gestione delle Pagine

Uno dei temi principali della VM è la gestione delle pagine, infatti è importante che le pagine che servono siano presenti in memoria in modo di non dovere andare a recuperarle sul disco; nei casi in cui la pagina non è in memoria invece, bisogna cercare di recuperarla il più velocemente possibile.

Riconoscere la mancanza di pagina

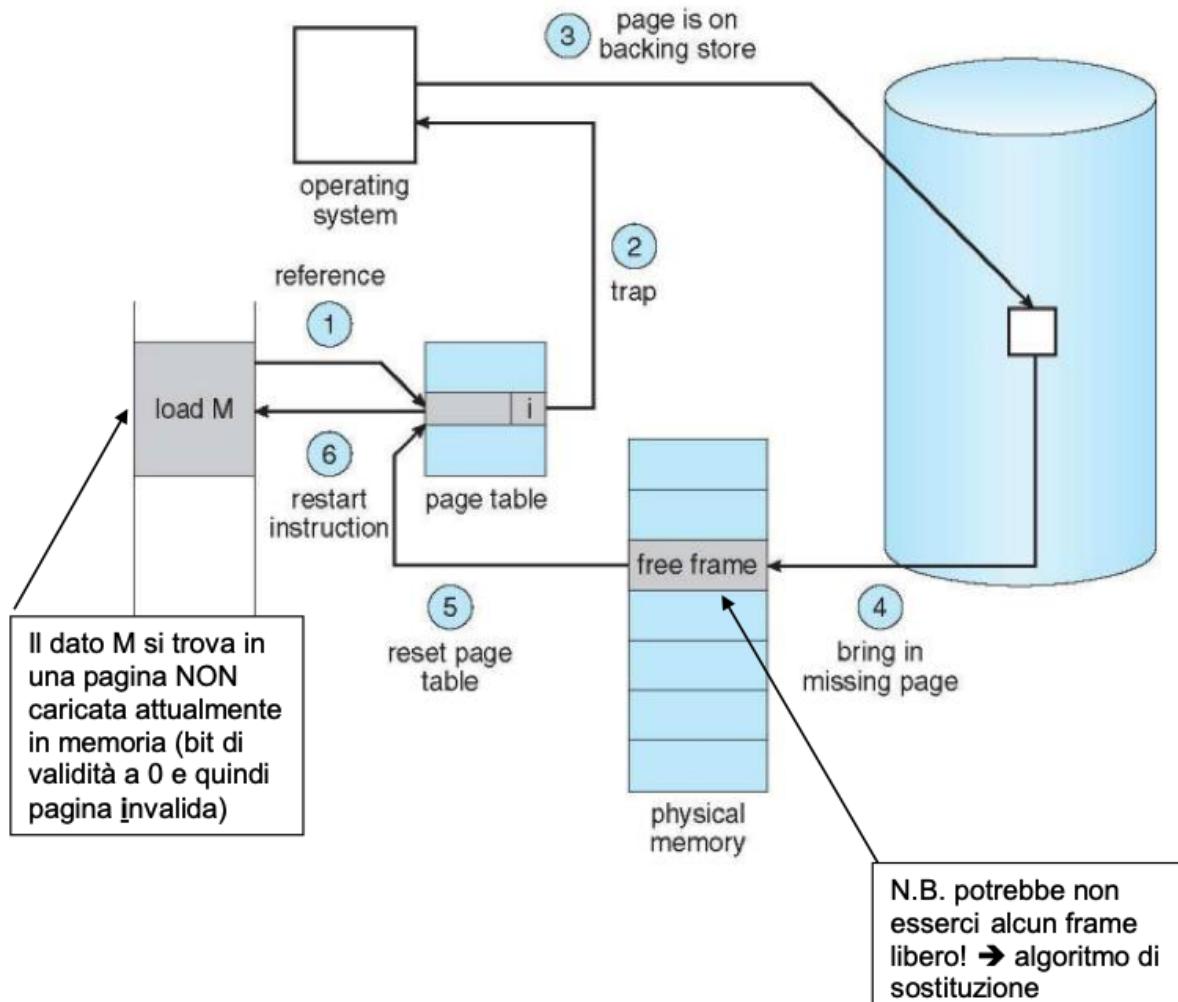
Una possibile strategia per riconoscere la mancanza di pagina è quella di utilizzare un **bit di presenza**, detto anche bit di validità, in ogni riga della tabella delle pagine. se vale 1 la pagina è presente in memoria, altrimenti no. Il S.O. deve mantenere una tabella interna per decidere se si tratta di un riferimento non valido e tutti gli indirizzi sul disco di dove si trovano le pagine (in particolare quelle mancanti). In teoria, alcuni programmi possono accedere a diverse pagine con una singola istruzione provocando più page fault.

Gestione della Eccezione di Mancanza di Pagina (Page Fault Trap)

Il processo **viene sospeso** e il controllo torna al S.O., che può tramite l'algoritmo di scheduling mandare in esecuzione un altro processo pronto. Il processo che ha subito il **page fault** rimane sospeso finché la pagina mancante non viene caricata in memoria. Quando c'è un page fault (bit di presenza = 0) il gestore della memoria deve:

1. Trovare una pagina di memoria reale libera (o liberarne una)
2. caricare la pagina mancante leggendola dalla memoria secondaria, modificando la tabella delle pagine (e caricando le informazioni anche nella TLB)
3. Settare a **pronto** il processo che era stato sospeso

Chiaramente l'hw deve essere in grado di riconoscere una interruzione durante l'esecuzione di una istruzione e, quando il processo che genera la Page Fault Trap viene interrotto il S.O. deve garantire che, una volta recuperata la pagina che serve, l'esecuzione di quel processo riprenda **esattamente** dal punto dove si era interrotto. Vediamo ora con la seguente immagine le varie fasi di gestione di un page fault:



In alcuni casi è anche possibile avviare l'esecuzione di un processo senza pagine in memoria, questa casistica si dice **paganazione su richiesta pura**. Quando il S.O. carica nel contatore di programma l'indirizzo della prima istruzione del processo, si verifica un page fault → il processo continua la propria esecuzione, provocando page fault, fino ad ottenere il caricamento in memoria di tutte le pagine "necessarie". Il punto due della precedente immagine è un evento cruciale nel funzionamento della VM, vediamo quindi più nello specifico cosa avviene quando una pagina non è presente in memoria:

1. Trap al S.O.
2. Salvataggio dei registri utente e dello stato del processo
3. Verifica che l'interruzione sia dovuta o meno ad un page fault
4. Controllo della correttezza del riferimento alla pagina e localizzazione della pagina su disco (punto 3 in figura)
5. Lettura dal disco e trasferimento (4 della figura)
6. Attesa nella coda del dispositivo di I/O
7. Attesa dovuta a posizionamento e latenza
8. Trasferimento della pagina in un frame libero
9. Durante l'attesa: allocazione della cpu ad altri processi utente
10. Ricezione dell'interrupt da disco (I/O completato)
11. Context switch (se è stato eseguito il passo 6)
12. Verifica della provenienza dell'interruzione da disco

13. Aggiornamento della tabella delle pagine (e dei frame) per segnalare la presenza in memoria della pagina richiesta (5 della figura)
14. Attesa nella ready queue
15. Context switch per la ripresa della esecuzione sulla cpu (6 della figura)

3.3.3 Strategie relative alla memoria virtuale

A livello logico (indipendentemente da paginazione o segmentazione) ci sono alcuni compiti per i quali si applicano **strategie** o politiche volte a definire **come** svolgere questi compiti in modo corretto ed efficiente:

1. Strategia di allocazione: **quanta** memoria reale (pagine o segmenti) allocare inizialmente ad ogni processo attivo? Se si usa la paginazione su richiesta pura, la risposta è chiaramente 0, ma non tutti i S.O. la usano.
2. Strategia di ricerca: **quali** parti (pagine o segmenti) caricare, e **quando** caricarle, dal disco in memoria?
3. Strategia di sostituzione (o rimpiazzamento): **quando** è necessario portare una nuova parte (pagina o segmento) in memoria centrale e non c'è spazio a sufficienza (**sovrallocazione**) **quale** parte (vittima) rimuovere per creare spazio per la nuova?
4. Strategia di posizionamento: **dove** mettere una parte (pagina o segmento) che entra in memoria centrale?

Inizieremo dai punti 2 e 4 in quanto sono quelli più intuitivi e banali.

Strategie di Ricerca

Quando una parte (pagina o segmento) che serve ad un processo non si trova in memoria c'è il bisogno di recuperarla. Nel momento in cui riceviamo un eccezione di page fault dobbiamo quindi andare a recuperare la pagina mancante. A seconda dello schema su cui si basa la Memoria Virtuale si ha quindi: richiesta di parti (pag o seg) e paginazione/segmentazione su richiesta. In alternativa si potrebbero pensare anche delle strategie anticipative, ma è difficile prevedere in modo accurato le necessità future di un processo.

Strategia di Posizionamento

Si adottano le stesse strategie di allocazione già descritte per la paginazione e la segmentazione. Nel caso di paginazione basta trovare una pagina libera; nel caso della segmentazione il discorso è più complicato: bisogna trovare una zona di memoria libera sufficientemente grande per far entrare nella memoria il segmento mancante.

Strategia di Sostituzione

Affrontiamo ora le strategie di sostituzione, concentrando l'attenzione sull'implementazione della VM con uno **schema a paginazione**. Quando è necessario portare una nuova pagina in memoria centrale, **ma** non c'è spazio a sufficienza si parla di **sovrallocazione** queste casistiche rappresentano un **problema** e sono un tipo di situazione **molto comune** perché, in particolare aumentando il grado di multiprogrammazione, ci sono molti processi caricati parzialmente in memoria. L'unica **soluzione** che abbiamo è **liberare** una pagina allocata, una sorta di swap out di una pagina, per fare posto a quella nuova. Il punto quindi è **quale** pagina rimuovere? Si tratta di selezionare una **vittima** che verrà spostata su disco. Bisogna adottare una **politica di sostituzione**. Questo tipo di operazione richiede 2 trasferimenti da/a disco, uno per caricare la pagina mancante e uno per spostare la vittima. In genere esiste una **copia in memoria secondaria** (nel backing store) di tutte le pagine, per migliorare le performance, il backing store deve essere il più veloce possibile, è quindi meglio tenerlo separato dal file system possibilmente anche su un device dedicato ed accedervi direttamente. Analizziamo i seguenti due casi:

- Se la pagina che viene selezionata come vittima **non** è stata **modificata** era in memoria reale, allora semplicemente possiamo sovrascriverla, non è necessario effettuare la copia nel backing store e quindi le interazioni con il disco passano da 2 a 1. Quando servirà di nuovo verrà caricata la **copia** dal backing store.
- Se la pagina che viene selezionata come vittima è stata **modificata** mentre era in memoria allora si deve ricopiarla sul backing store, quindi 2 interazioni con il disco.

A volte è presente un meccanismo hw che tiene traccia delle modifiche alle pagine chiamato **dirty bit** questa flag viene alzata ad ogni scrittura sulla pagina, questo meccanismo permette al sistema, quando in idle, di salvare su disco tutte le pagine con il dirty bit settato. Se non esiste il dirty bit allora bisogna scrivere la pagina su disco ogni volta, abbassando le prestazioni.

Definiamo il tempo medio di accesso (**EAT**) come

$$EAT = (1 - p) * T_{mem} + p * (T_{page_fault})$$

Dove p è il page fault rate indicato come probabilità tra 0 e 1.

Politiche di Sostituzione

Questo tipo di algoritmi viene valutato eseguendoli su una particolare stringa di riferimenti a memoria (detta **reference string** o **sequenza degli accessi**) e contando il numero di page fault su tale sequenza. La sequenza è costituita solo da numeri di pagine logiche, non da indirizzi completi, perché a generare il page fault è il tentativo di trovare un certo numero di pagina fisica dove è stata caricata una certa pagina logica. Gli accessi multipli alla stessa pagina non provocano page fault, dato che una pagina viene tolta dalla memoria **solo** quando serve fare spazio, possiamo pensare che gli ulteriori accessi (dopo il primo) non provocheranno eccezioni, quindi nella sequenza non troveremo mai due numeri di pagina uguali adiacenti. I risultati ottenuti dipendono significativamente dal numero di frame a disposizione, in generale ci si aspetta che il numero di page fault cali all'aumentare del numero di frame liberi disponibili. Le politiche di sostituzione devono tendere, a parità di numero di frame a disposizione, a minimizzare il numero di page fault. In tutti gli esempi seguenti la sequenza degli accessi (reference string) che verrà considerata è la seguente:

```
7, 0, 1, 2, 0, 3, 0, 4, 2, 3, 0, 3, 2, 1, 2, 0, 1, 7, 0, 1
```

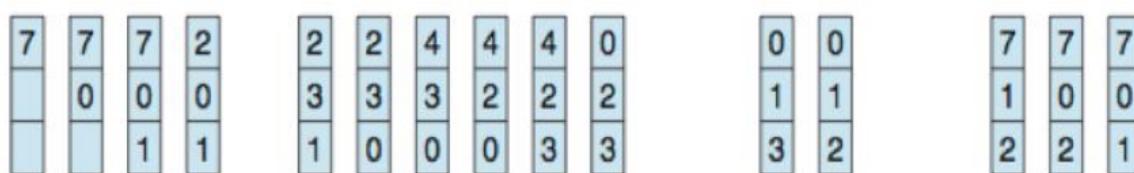
Inoltre, consideriamo, sempre in tutti gli esempi, una situazione di partenza in memoria che vede 3 pagine libere.

Politica di sostituzione First In First Out (FIFO)

Questa politica seleziona come **vittima** la pagina che si trova da più tempo in memoria reale. Per effettuare questa politica, il gestore della memoria deve tenere traccia dell'ordine di caricamento delle pagine in memoria in una coda fifo. Se analizziamo la situazione di esempio presentata in precedenza con 3 pagine libere e la suddetta reference string otteniamo un totale di 15 page faults.

reference string

```
7 0 1 2 0 3 0 4 2 3 0 3 2 1 2 0 1 7 0 1
```



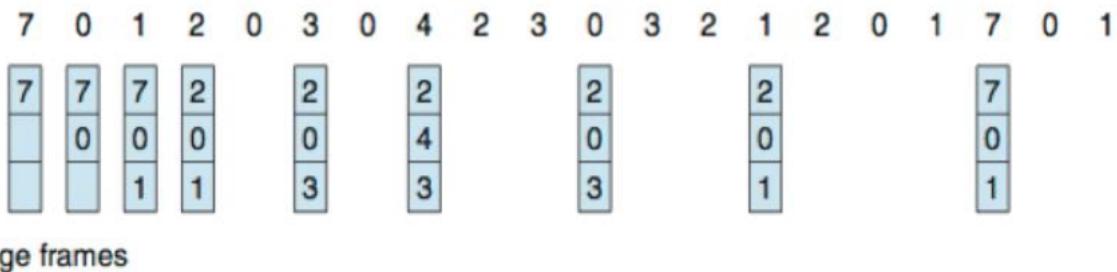
page frames

Questa politica risulta molto semplice da implementare ed intuitiva, ha prestazioni non sempre buone dato che tende a scartare pagine indipendentemente dalla loro **frequenza di utilizzo** ma soprattutto è affetta da un problema denominato **belady**: al crescere del numero di pagine libere può crescere il numero di page fault!!! Dipende dalla sequenza di accesso.

Politica di sostituzione Ottima (OPT)

Dopo la scoperta della anomalia di **belady** la ricerca si è focalizzata sulla definizione di un algoritmo che prevedesse il **minore** numero di page fault. Questa politica seleziona come vittima la pagina che (nel futuro) verrà utilizzata dopo il maggior tempo possibile. Sempre considerando l'esempio iniziale con questo metodo si ottengono 9 page fault.

reference string



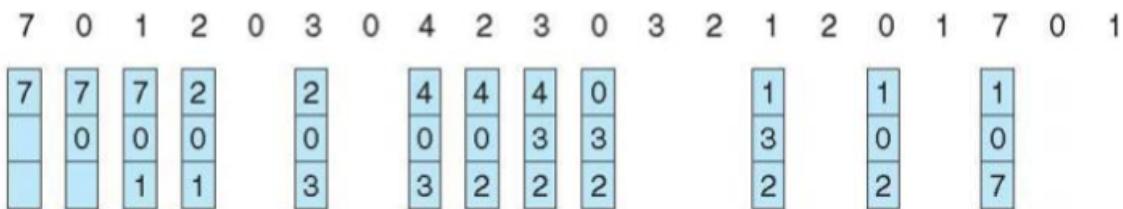
page frames

Il grosso **svantaggio** di questa politica è la **difficoltà di implementazione** dato che comporta avere una conoscenza futura, viene solamente usata come metro di confronto per valutare le altre politiche.

Politica di sostituzione Least Recently Used (LRU)

Questa politica seleziona come vittima la pagina che è stata utilizzata meno recentemente. L'algoritmo **LRU** si comporta **meglio** di quello FIFO, si considera il comportamento del programma sulla base dell'ipotesi che la pagina utilizzata meno recentemente ha **minore probabilità** di essere referenziata nel prossimo futuro. Per effettuare questa politica il gestore della memoria deve tenere traccia dell'ultimo istante di tempo in cui le pagine sono state usate. Vediamo che nell'esempio si ottengono un totale di 12 page fault.

reference string



page frames

OPT e **LRU** sono algoritmi "a pila" che non soffrono dell'anomalia di Belady perché se $S(n)$ è l'insieme delle pagine mantenute in memoria con n frame, allora:

$$S(n) \subseteq S(n+1)$$

La politica LRU è usata molto spesso, il **problema** maggiore è **come** implementarla dato che bisogna avere un modo per determinare l'**ordine di accesso** alle pagine che richiede hw aggiuntivo. Ci sono principalmente due soluzioni:

1. Uso di un **contatore**: questa soluzione è la più semplice, consiste nell'aggiungere un campo **tempo** per ogni riga (pagina) della tabella delle pagine per mantenere l'indicazione dell'istante di tempo dell'ultimo accesso alla pagina e, nel momento in cui bisogni rimuoverne una, si sceglie la pagina con il valore **più piccolo** del campo tempo. Gli

svantaggi di questo approccio sono facili da individuare: serve effettuare una ricerca lineare sulla tabella delle pagine e aggiornare il contatore **ad ogni** accesso in memoria, con la possibilità di overflow.

- Aggiunta di uno **stack** con i numeri di pagina: ogni volta che si effettua l'accesso ad una pagina il suo numero viene inserito alla **testa** dello stack; così facendo la pagina in cima è quella usata più di recente mentre quella in **coda** è la **vittima** → Least Recently Used. Implementando questa soluzione non è necessario fare ricerche per la scelta della pagina, ma mantenere lo stack aggiornato ha un costo, soprattutto nel caso in cui non bisogni fare una semplice operazione di push o di pop ma, ad esempio, si debba riordinare gli elementi all'interno dello stack per riportarne in cima alcuni che si trovano al centro.

Politica di sostituzione che Approssimano LRU

La politica di sostituzione LRU è un algoritmo lento e, per il suo supporto completo, richiede hw non reperibile su tutte le architetture. Alcune architetture introducono un meccanismo hw detto **reference bit** che può servire ad **approssimare** la politica LRU. Si usa un **reference bit** nella tabella delle pagine per ogni riga (pagina), questo bit inizialmente è zero e viene settato **ogni volta che la pagina viene acceduta**. Vediamo quindi utilizzando questo metodo quali approssimazioni di LRU possono essere implementate:

1. Memorizzazione Dei Reference Bit

Questa politica, per ogni pagina, memorizza **ad intervalli regolari** in un registro a scorrimento (tipicamente di 1 Byte) il suo reference bit. Il reference bit viene memorizzato nel bit più significativo, effettuando uno shift a destra e quindi scartando quello meno significativo. Il reference bit infine viene azzerato. Quindi, in questo Byte di **storia** abbiamo la situazione degli ultimi 8 intervalli di tempo. A questo punto, per scegliere la **vittima** basta confrontare i byte di ogni pagina, considerandoli **interi senza segno** e scegliendo la pagina a cui è associato il valore più basso. Chiaramente non si può garantire l'unicità dei valori dei byte, se ci sono più pagine con lo stesso valore possiamo decidere di eliminarle tutte oppure usare una selezione FIFO tra di esse. Infine, si può pensare di aumentare o diminuire la dimensione del registro a scorrimento riducendolo fino a zero (**reference bit**).

2. Algoritmo di Seconda Chance: Not Recently Used (NRU)

Questa politica si basa sulla politica FIFO, con una variante: quando una pagina viene selezionata come **vittima** dall'algoritmo FIFO, viene verificato il suo **reference bit** se è **0** allora si conferma la selezione, se è **1** allora si dà a quella pagina una **seconda chance** e si passa a considerare la prossima pagina nell'ordinamento FIFO. Quando a una pagina è data una seconda chance, il suo **reference bit** viene **azzerato** e la pagina diviene l'ultima pagina inserita della coda FIFO → implementazione con una coda circolare detta anche **clock**. Nel caso peggiore, quando tutti i bit sono a 1, si esegue un ciclo su tutta la coda dando a ogni pagina una seconda chance → NRU **degenera** nella gestione FIFO se tutti i bit sono 0. Se una pagina è acceduta ciclicamente con frequenza sufficientemente elevata, ottiene sempre una seconda chance e non viene mai sostituita!

3. Classi di Pagine

Questa politica considera oltre al **reference bit** anche il **dirty bit** e classifica ogni pagina in una delle seguenti classi:

- [0,0] Né usata recentemente né scritta → migliore pagina da sostituire
- [0,1] Non usata recentemente, ma scritta → non così buona poiché la pagina deve essere scritta su disco
- [1,0] Usata recentemente, ma non scritta → non buona poiché la pagina probabilmente verrà ancora usata

4. [1,1] Usata recentemente e scritta → la peggiore, la pagina verrà ancora usata e prima di essere sostituita deve essere scritta su disco

Quando è necessario un rimpiazzamento di pagina, viene scelta quella che è nella classe (non vuota) più bassa. Se ci sono più pagine, si può usare una politica FIFO o random per scegliere la pagina vittima. Questo algoritmo viene anche detto **seconda chance migliorato** poiché è lo stesso algoritmo di prima, ma invece ce controllare solo il reference bit controlla anche il dirty bit.

4. Least Frequently Used (LFU)

Questa politica mantiene un **contatore** del numero di riferimenti che sono stati fatti ad ogni pagina, seleziona come **vittima** la pagina che ha il contatore settato al valore **minore** basandosi sull'assunzione che una pagina attivamente usata dovrebbe avere un numero di riferimenti maggiore. Un **problema** di questa politica è che se una pagina viene usata molto frequentemente solo inizialmente il suo contatore le permette di permanere in memoria anche quando non è più necessario, una possibile **fix** è effettuare sul contatore uno shift a destra di un bit ad intervalli regolari.

5. Most Frequently Used (MFU)

Questa politica è **complementare** rispetto a LFU: considera che se una pagina ha il minor valore del contatore allora è stata probabilmente appena caricata in memoria reale e quindi è ancora in uso. Le politiche **LFU** e **MFU** sono **poco usate** poiché la loro implementazione risulta onerosa e non approssimano bene la sostituzione OPT.

Osservazioni

1. Di solito i progettisti dei S.O. nella gestione della memoria virtuale fanno in modo di mantenere sempre un **pool di frame liberi** che vengono usati "in parallelo" all'algoritmo di sostituzione: più nello specifico quando si verifica un page fault, si seleziona un frame vittima, ma prima di trascriverlo in memoria secondaria (dirty bit = 1), si procede alla copia della pagina richiesta in un frame del pool. In questo modo il processo può essere riattivato rapidamente senza attendere la fine dell'operazione di salvataggio del frame vittima sulla memoria di massa → il frame che risulterà libero dopo lo sfratto della vittima andrà a fare parte del pool di frame liberi (senza cancellarne il contenuto). Quando si verifica un page fault, prima di accedere alla memoria di massa, si controlla se la pagina richiesta è ancora presente nel pool dei frame liberi & rarr; soft page fault, non sono necessarie operazioni di I/O.
2. La tecnica di copiatura su scrittura, **copy-on-write (COW)**, permette alla coppia di processi padre-figlio di condividere inizialmente le stesse pagine di memoria. Se uno dei due processi modifica una pagina condivisa, e solo in quel caso, viene creata una copia della pagina. La tecnica COW garantisce una modalità di creazione dei processi più efficiente, grazie alla copia delle sole pagine (condivise) modificate. Questa tecnica è usata nella primitiva `fork()` di Linux, dato che normalmente un processo figlio dopo essere stato creato effettua una primitiva `exec()` si pensi ad esempio ad un processo sotto-shell che deve eseguire un comando.

Fino ad ora abbiamo implicitamente supposto di attuare una sostituzione andando ad operare sui frame allocati al processo che ha provocato il page fault → **sostituzione locale** con il **vantaggio** che si tende a **localizzare** gli effetti della strategia di sostituzione sulla politica di allocazione (il numero di pagine allocate ad un processo rimane costante) e lo **svantaggio** che non vengono rese disponibili ai processi che ne facciano richiesta pagine di altri processi scarsamente utilizzate (possibile sottoutilizzo della memoria). Chiaramente la sostituzione locale non è l'unica alternativa, se un processo p subisce un page fault e c'è bisogno di un rimpiazzamento di pagina l'algoritmo potrebbe considerare per selezionare la vittima **tutte** le pagine presenti in memoria

(sostituzione globale). Questa alternativa comporta alcuni **svantaggi**: il numero di pagine allocate ad un processo **varia** durante l'esecuzione, in questo modo si tende ad aumentare il **grado di accoppiamento** fra la strategia di sostituzione e la politica di allocazione dei vari processi → le pagine che l'algoritmo di allocazione sceglie **inizialmente** per un certo processo potrebbero essere selezionate dall'algoritmo di sostituzione, attivato per "colpa" di un altro processo. Un processo può avere **performance** molto diverse in differenti esecuzioni a causa dell'intrusione dell'algoritmo **globale**: il tempo di esecuzione di ciascun processo può variare in modo significativo, un processo non può controllare la propria frequenza di page fault. I **vantaggi** della sostituzione comportano un **maggior throughput**.

Strategia di Allocazione

Vediamo ora nello specifico quanta memoria reale serve allocare ad ogni processo attivo. Dal punto di vista del singolo processo, **maggior** è la memoria allocata **minore** sarà il numero di page fault. Chiaramente non possiamo esagerare in questo perché se no allocando troppa memoria per ogni processo, andremmo a diminuire troppo il grado di multiprogrammazione. Dal punto di vista del sistema **minore** è la memoria allocata e **maggior** sarà il grado di multiprogrammazione, con lo svantaggio complementare a prima che allocando troppa poca memoria per processo si rischia di incappare in troppi page fault. Si può impostare un **limite minimo** sul numero di pagine che devono essere allocate ad ogni processo, questo numero viene fissato in base all'architettura secondo il numero massimo di pagine che ogni singola istruzione può referenziare, solitamente tra le 3 e le 17. Alcuni S.O. applicano la **paginazione su richiesta pura** cioè senza avere caricato nessuna delle pagine logiche in memoria e quindi senza effettuare **pre-paging**. Un altro concetto fondamentale di cui tener conto quando si parla di strategie di allocazione è come viene gestito il **limite massimo** di memoria per ogni processo.

Politica di Allocazione Uguale

Questa semplice politica consiste nell'allocare un numero uguale di pagine a tutti i processi, una vera e propria **assegnazione uniforme** dividendo il numero di pagine disponibile per il numero di processi che ne hanno bisogno, ad esempio con 5 processi e 100 pagine ne assegna 20 ad ognuno.

Politica di Allocazione Diseguale

Questa politica riconosce che i vari processi hanno necessità diverse di memoria e quindi adotta una **assegnazione proporzionale**: alloca le pagine in base alla dimensione del processo. Si sommano le dimensioni di tutti i processi e si assegnano le pagine in maniera proporzionale alla loro dimensione rispetto al totale, le pagine allocate dovranno essere \geq del valore minimo fissato dall'architettura e \leq delle pagine libere. Una variante di questa politica tiene conto anche della **priorità**, andando a prelevare pagine da processi a bassa priorità per allocarle a quelli ad alta priorità.

Trashing

Un sistema si dice in **trashing** quando spende più tempo a gestire la memoria virtuale che ad eseguire i processi. Le cause possono essere spiegate illustrando il funzionamento dei primi sistemi di memoria virtuale paginata (che non usavano tecniche per prevenirlo):

1. Se un processo ha un numero di pagine inferiore a quelle necessarie, il tasso di page fault cresce
2. Il processo (con la sostituzione globale) prenderà i frame da altri processi e similmente faranno altri processi
3. Se molti processi entrano in questa coda, la ready queue si svuota velocemente, l'efficienza della cpu cala, quindi il S.O. aumenta il livello di multiprogrammazione

4. I nuovi processi hanno bisogno di acquisire frame, che però non sono liberi, quindi entrano nella coda del dispositivo usato per il backing store si torna quindi al punto 3

Il thrashing viene anche detto **paginazione degenero** ed è estremamente dannoso per le prestazioni: l'inattività della cpu ed i carichi di I/O che vengono generati possono prendere il sopravvento sul normale funzionamento del sistema. In casi estremi, il sistema potrebbe non funzionare utilmente, spendendo tutte le sue risorse spostando le pagine dal backing store alla memoria e viceversa dalla memoria al backing store. La paginazione su richiesta **tuttavia funziona** perché i processi rispettano il **principio di località degli accessi**: i programmi hanno una forte tendenza a favorire un sottoinsieme del loro spazio di indirizzamento durante l'esecuzione, che costituisce la loro **località**. Una **località** è un insieme di pagine che vengono accedute insieme e quindi sono contemporaneamente in uso attivo:

- Il processo passa da una località ad un'altra
- Le località possono essere (parzialmente) sovrapposte

Esempio

Quando viene invocata una funzione si definisce una nuova **località** vengono fatti riferimenti alle sue istruzioni, alle sue variabili locali ed a un sottoinsieme delle variabili globali; quando la funzione termina il processo lascia la località corrispondente.

Principio di Località Degli Accessi

- Un processo, durante qualsiasi intervallo di tempo, favorisce un sottoinsieme delle proprie pagine
- La sequenza di accessi alla memoria di un processo mostra un'alta correlazione tra immediato passato e immediato futuro;
- La frequenza con cui un certa pagina viene referenziata è una funzione che varia lentamente nel tempo

Quindi la **località** di un processo è rappresentata da un **insieme di pagine** all'interno delle quali, durante un certo periodo di tempo, si trova la maggior parte degli accessi in memoria di quel processo. La località presenta caratteristiche **dinamiche**:

- L'identità delle pagine che ne fanno parte varia dinamicamente
- Quando un programma è in esecuzione, il processo **passa lentamente** attraverso diverse località, che possono sovrapporsi

Quindi, quando avviene il **thrashing**?

1. Il **thrashing di un processo** avviene quando la memoria assegnatagli è inferiore a quella richiesta dalla sua località
2. Il **thrashing del sistema** avviene quando la memoria fisica è inferiore alla somma delle località dei processi attivi

Gli effetti del thrashing possono essere limitati utilizzando tecniche di sostituzione locale. Il principio di località degli accessi porta a considerare che ci sia un **elevata probabilità** che le pagine referenziate recentemente siano referenziate nel prossimo futuro. Il modello di località è il principio non dichiarato sottostante all'uso della TLB (caching), se gli accessi fossero casuali invece che strutturati in località il caching sarebbe pressoché inutile.

Politica di Allocazione Basata sulla Teoria del Working Set

La teoria del working set affronta i problemi della allocazione e della sostituzione cercando di dare risposta al problema del thrashing. Il **working set** (WS) di un processo è l'insieme delle pagine che sono state accedute in una **finestra temporale** di ampiezza Δ nell'immediato passato. È una **approssimazione** della località di un processo. Ci sono **due regole** che stanno alla base della teoria del working set:

1. Un programma deve andare in esecuzione **se e solo se** il suo WS è in memoria reale
2. Una pagina **non deve** essere rimossa dalla memoria **se appartiene** al WS del processo corrente; questo vale sia per l'allocazione che la sostituzione

Se **non** si segue questa teoria e si elimina una pagina che è nel WS di un processo, la **frequenza** dei page fault **aumenterà**. Una caratteristica fondamentale di questa politica è la **dimensione** dei WS. Chiaramente per far sì che questa politica funzioni bisogna avere abbastanza spazio in memoria per contenere tutti i WS altrimenti bisogna diminuire il numero di processi attivi, ovvero liberare memoria tramite **swapping** di processi. Quindi per evitare il thrashing si deve diminuire il grado di multiprogrammazione attuando uno swapping-out di alcuni processi e non appena si libera nuovamente spazio bisognerà effettuare lo swap-in. Un'altra caratteristica fondamentale di questa politica è la dimensione della finestra temporale Δ : se è **troppo piccola** allora non comprenderà tutta la località del processo, se è **troppo grande** può comprendere troppe località del processo. La finestra temporale può essere definita anche in termini di **accessi in memoria** per esempio potremmo decidere di definire località del processo i suoi ultimi 10 accessi in memoria e quindi continuare a cambiare dinamicamente il suo WS ad ogni nuovo accesso in memoria.

Politica di Allocazione Basata sulla Frequenza dei Page Fault

In generale la **relazione** fra la frequenza dei page fault e la quantità di memoria allocata ad un processo **non è lineare**, si stabilisce una frequenza di page fault "accettabile" e si utilizza una politica di sostituzione globale:

- Se la frequenza effettiva di page fault è troppo bassa, il processo rilascia dei frame
- Se la frequenza effettiva di page fault è troppo alta, il processo acquisisce dei nuovi frame

In particolare:

1. Per ogni processo esiste una **soglia** nel rapporto fra numero di pagine **fisiche** e numero di pagine **logiche** al di sotto della quale il numero delle page faults aumenta **molto rapidamente**.
2. Per ogni processo esiste un **limite** al numero di pagine fisiche al di sopra del quale si ha un modesto aumento delle prestazioni

La politica di allocazione deve scegliere un valore compreso fra questi due estremi: si fissano due soglie entro cui deve stare la frequenza di page fault. Quando la frequenza di page fault supera la soglia superiore si devono allocare altre pagine, mentre quando diminuisce al di sotto di quella inferiore si deve fermare la allocazione di nuove pagine.

3.3.4 File Mappati in Memoria

La mappatura di file in memoria permette il trattamento dell'accesso a file come un normale accesso alla memoria, la mappatura si realizza associando un blocco del disco ad una o più pagine residenti in memoria. L'accesso iniziale al file avviene tramite una richiesta di pagina che produce un page fault, quindi una porzione del file pari ad una pagina viene caricata dal S.O. in una pagina fisica. Ogni successiva lettura e scrittura del file viene gestita come un accesso ordinario alla memoria. Questo approccio comporta alcuni **vantaggi**:

1. Si alleggerisce il lavoro del sistema di I/O
2. Si ha accesso rapido alle informazioni contenute nel file
3. Più processi possono mappare contemporaneamente lo stesso file, garantendo la condivisione di pagine di memoria.

E introduce alcuni **problemi**:

1. Quando aggiornare il contenuto del file su disco?
 1. periodicamente e/o all'atto della chiusura del file con la chiamata `close()`
 2. Quando il pager scandisce le pagine per controllarne il **dirty bit** (modifiche immediatamente visibili a tutti i processi che condividono il file)

La condivisione dei file in memoria presenta analogie con la **memoria condivisa** e, infatti, la memoria condivisa può essere realizzata utilizzando file mappati in memoria. L'accesso al contenuto del file mappato in memoria deve essere controllato utilizzando un meccanismo di sincronizzazione (di quelli già visti) per garantire l'integrità dei dati

3.3.5 Allocazione di Memoria al Kernel

Il Kernel, per allocare la propria memoria, attinge ad una riserva di memoria libera **diversa** dalla lista dei frame usata per soddisfare i processi utente. Il kernel richiede memoria per strutture dati dalle dimensioni variabili, di solito molto più piccole di un frame. Bisogna fare un uso oculato della memoria per evitare gli sprechi: in molti S.O. il kernel non è soggetto né a paginazione né a memori virtuale. Parti della memoria del kernel devono essere contigue perché alcuni dispositivi accedono direttamente alla memoria fisica senza l'interfaccia della memoria virtuale, come ad esempio i **dispositivi di I/O**. Vediamo ora due metodi per allocare la memoria per il **kernel**: il metodo Buddy e il metodo a Lastre.

Metodo Buddy

Utilizza una partizione di dimensione fissa per l'allocazione della memoria, composta da pagine fisicamente contigue. La memoria viene allocata attraverso un allocatore-potenza-di-2:

- Alloca memoria in blocchi di dimensione pari a potenze di 2
- La quantità richiesta viene arrotondata alla più piccola potenza di 2 che la contiene
- Quando si richiede meno memoria di quella che costituisce la partizione corrente, questa viene divisa in due partizioni gemelle di identica dimensione
- Il procedimento continua fino ad ottenere la partizione minimale per l'allocazione richiesta

Vantaggio: Al momento della deallocazione si possono ricongiungere rapidamente buddy (partizioni) adiacenti a formare partizioni di memoria contigua più lunghe

Svantaggio: L'arrotondamento della dimensione del blocco alla potenza del 2 più piccola che lo contiene può generare **frammentazione interna**.

Metodo a Lastre

- Una lastra (o slab) è composta da uno o più frame fisicamente contigui
- Uno o più slab contigui formano una cache, ciascuna cache è un contenitore di oggetti.
- Un oggetto è un'area di memoria di una specifica dimensione: ad esempio PCB, descrittori di file, etc... Vi è una sola cache per ciascuna categoria di strutture dati del kernel.

Quando una struttura dati del kernel deve essere allocata, si sceglie dalla cache opportuna un qualunque oggetto libero e lo si marca come "*in uso*". Una slab può assumere uno dei seguenti stati:

- Piena: tutti gli oggetti sono contrassegnati come usati

- Vuota: tutti gli oggetti sono contrassegnati come liberi
- Parzialmente occupata - la lastra contiene oggetti sia usati sia liberi

Quando una lastra diventa piena, un nuovo oggetto viene allocato in una slab completamente vuota. Se non vi sono lastre vuote, si procede all'allocazione di una nuova lastra (cache grow) e la si appende in fondo alla cache opportuna.

Vantaggi

1. Si annulla lo spreco di memoria derivante dalla frammentazione
2. Le richieste di memoria vengono soddisfatte rapidamente

3.3.6 Considerazioni Varie

Dimensione della Pagina

Abbiamo visto come la **dimensione** della pagina gioca un ruolo chiave nei S.O. a memoria virtuale. Tipicamente la dimensione delle pagine varia tra 4KB e 4MB. Una pagina con **dimensioni piccole** può ridurre la frammentazione di pagina, ridurre la quantità di I/O nel trasferimento memoria-disco e viceversa e può fare un uso migliore della memoria dato che pagine piccole si adattano maggiormente alla località dei programmi. **MA** aumentano le dimensioni delle tabelle delle pagine aumentano in proporzione i tempi di trasferimento da disco e il numero di page fault. Riassumendo:

Meglio Piccola	Meglio Grande
Frammentazione	Dimensioni della page table
Quantità di I/O	Tempo di I/O
Località	Numero di page fault

Non esiste una risposta ottimale al problema, la tendenza storicamente è verso l'incremento.

Cache delle Pagine

Nei S.O. che fanno uso di VM basati sulla paginazione, un altro elemento di progetto molto importante riguarda la cache delle pagine o **TLB**. Un parametro importante della TLB, oltre l'hit ratio, è la **TLB reach**: la quantità di memoria accessibile via TLB. Idealmente, il working set di ogni processo dovrebbe essere contenuto nella TLB, altrimenti si verificano molti page fault ed il tempo di esecuzione diviene proibitivo. Per aumentare la portata della TLB si può aumentare la sua dimensione, questo approccio è tanto efficace quanto **costoso**. In alternativa si può aumentare la dimensione delle pagine, questa ultima scelta però potrebbe portare ad un incremento della frammentazione, date che non tutte le applicazioni richiedono pagine grandi.

Soluzione: prevedere pagine di diverse dimensioni: permette l'utilizzo di pagine grandi alle applicazioni che lo richiedono senza aumento della frammentazione. L'uso di dimensioni diverse delle pagine richiede che la gestione della TLB si svolta dal S.O. e non dall'hw, uno dei campi della TLB deve indicare la dimensione della pagina fisica cui fa riferimento. La TLB deve essere gestita (via hw o sw) da strategie di allocazione e sostituzione che consentano un migliore uso della capacità limitata di questa memoria associativa. I problemi da affrontare sono simili a quelli discussi per il problema generale della memoria virtuale. Un aspetto peculiare riguardo alla **allocazione** nella cache delle pagine è se in ogni istante debba contenere **solo** pagine del processo corrente (cache **dedicata** al processo corrente) oppure se possa contenere pagine di vari processi (cache **non dedicata**). Lo **svantaggio** di avere la cache **dedicata** è che quando si passa da un processo all'altro bisogna che la TLB sia svuotata e il nuovo processo subisce una serie di TLB miss fino a che non conterrà le informazioni relative alle sue pagine, causando

overhead eccessivo nel caso di esecuzione di una routine di risposta ad una interruzione. In ambito **multiprocesso** in genere si sceglie l'implementazione della cache **non dedicata**. Bisogna che nella cache ci sia l'indicazione anche del processo cui la pagina appartiene (Address-Space-Identifier **ASID**). Riguardo al problema di **rimpiazzamento** nella cache delle pagine, questo si applica solitamente ogni volta che un riferimento è adatto "a vuoto" (TLB miss): si può utilizzare nel caso di una cache **non dedicata** sia selezioni della vittima locali che globali. In genere, la scelta ricade su una politica **locale** basata sull'algoritmo **LRU**. Un altro aspetto riguardante la cache delle pagine è quello della **coerenza** delle informazioni rispetto alla situazione attuale (pagine in memoria reale o meno). Ci devono essere azioni (hw o sw) che consentono di azzerare il contenuto della cache o di invalidarne alcuni elementi.

Protezione e Condivisione

Per quanto riguarda la **condivisione** in genere si adotta la soluzione che **esclude** la **rimozione** di pagine/segmenti che sono **condivise**, questo perché è complesso mantenere la **consistenza** di più tabelle.

I/O

Un Problema analogo a quello della condivisione si può avere nel caso di **operazioni I/O**, ad esempio durante l'attesa di una procedura di I/O un processo. Immaginiamo un processo **p** che fa una richiesta di I/O e quindi si mette in **attesa** del completamento. Durante questa attesa potrebbe essere che avvengano page fault generati da altri processi e quindi la pagina che contiene **p** venga **selezionata come vittima**, una volta ottenuti i dati dall'operazione di I/O questi verrebbero scritti su una pagina fisica che non è la giusta pagina logica!!! Per rimediare a questo tipo di problema:

1. I dispositivi di I/O scrivono/leggono da buffer di sistema (questi non vengono **mai** sfrattati dalla memoria) con lo **svantaggio** che serve una doppia copia dallo spazio utente a quello di sistema e viceversa.
2. Le pagine di memoria interessate da operazioni di I/O sono bloccate (**pinned**), queste pagine vengono marcate con un **bit di vincolo** e non possono essere rimosse dalla memoria con il **problema** che una pagina pinnata potrebbe non essere mai rilasciata.

Elaborazione in Tempo Reale

Sfruttando la memoria virtuale si incrementa la produttività del sistema, però i singoli processi possono risentire di questa gestione: durante la loro esecuzione possono essere soggetti a un numero maggiore di page faults. Un processo in tempo reale ha bisogno di ottenere il controllo della cpu e di completare la propria elaborazione con un ritardo minimo. Quindi la memoria virtuale rappresenta l'antitesi dell'elaborazione in tempo reale.

3.3.7 Memoria Virtuale Basata sulla Segmentazione

Vantaggi:

- Maggiore facilità nella strategia di allocazione → bisogna avere per ogni processo almeno un segmento di codice, uno di dati e quello dello stack (working set di segmenti)
- Maggiore flessibilità a livello di **protezione e condivisione**
- Maggiore controllo sulla **località**

Svantaggi:

- Gestione della memoria reale (strategia di **posizionamento**) e di quella secondaria **complicata** a causa delle dimensioni variabili dei segmenti

Visto che le realizzazioni di **VM** con **paginazione** o **segmentazione** presentano vantaggi e svantaggi si cerca di ottenere il meglio di entrambe **combinando** i due approcci tramite la **segmentazione con paginazione**.

Segmentazione con Paginazione

L'utente usa la **segmentazione** e il sistema, trasparentemente, usa la **paginazione** per facilitare la allocazione sia in memoria reale che in quella secondaria, eliminando il problema della **frammentazione esterna**. Lo schema di traduzione di un indirizzo virtuale si basa quindi su due tabelle:

1. Tabella dei segmenti, contenente indirizzo di inizio (base), dimensione della tabella delle pagine da utilizzare per quel segmento e diritti di accesso
2. Tabella delle pagine, una per ogni segmento.

Variante: nel caso in cui non esiste nessuna pagina di un certo segmento presente in memoria reale (facendo check sul **bit di presenza segmento**) si può prevedere una **eccezione di mancanza di segmento**, che fa sì che si portino in memoria reale diverse pagine di quel segmento. Lo **svantaggio** della segmentazione con paginazione è il numero di accessi in memoria:

1. Primo accesso alla tabella dei segmenti
2. Secondo accesso alla tabella delle pagine del segmento corrente
3. Terzo accesso alla locazione di memoria

Nasce quindi la **necessità** sia di registri di segmento (o cache di segmento) che di cache di pagine.

3.3.8 Gestione della Memoria in Unix

Per concludere il discorso sulla memoria virtuale vediamo nel concreto come le prime versioni di unix gestiscono la memoria.

Prime Versioni

In Unix lo spazio logico è segmentato in segmenti di codice e segmenti di dati (globali, stack e heap). Nelle **prime versioni** l'allocazione dei segmenti era:

- Segmentazione pura con politica **first fit**
- Non c'era uso di memoria virtuale
- In caso di difficoltà di allocazione dei processi, swapping dell'intero spazio degli indirizzi **N.B.** condivisione di codice.

I principali problemi con questo tipo di gestione erano la frammentazione esterna, la stretta influenza della dimensione dello spazio fisico sulla gestione dei processi in multiprogrammazione e la crescita dinamica dello spazio logico, che comportava la necessità di rilocazione di processi già caricati in memoria.

Swapping

In assenza di memoria virtuale, lo **swapper** ricopre un ruolo chiave per la gestione delle contese di memoria. Periodicamente esso viene attivato per provvedere ad eventuali **swap-out** di processi sospesi o ingombranti o da molto tempo in memoria e **swap-in** di processi piccoli o da molto tempo swappati.

Versioni Moderne

Da BSD v.3 in poi:

- Segmentazione paginata: in particolare il segmento di codice e il segmento di dati
- Memoria virtuale tramite paginazione su richiesta ottimizzata gestita con core map: struttura dati interna al kernel che descrive lo stato di allocazione dei frame e che viene consultata in caso di page fault

La paginazione su richiesta ottimizzata:

1. Una prima ottimizzazione viene ottenuta mantenendo sempre disponibili un certo numero di pagine fisiche tramite la sostituzione operata dal **pagedaemon**: all'atto di un page fault, la sua gestione può essere effettuata più velocemente.
2. Una seconda ottimizzazione deriva da pre caricare pagine non strettamente necessarie nei frame mantenuti liberi: quando avviene un page fault, se la pagina è già in un frame libero, basta soltanto modificare la tabella delle pagine e la lista dei frame liberi senza necessità di effettuare un trasferimento da disco

Le versioni di unix moderne usano una politica di sostituzione delle pagine che approssima la politica LRU, simile all'algoritmo di **seconda chance**.

Sostituzione della vittima:

- La pagina viene resa invalida
- Il frame selezionato viene inserito nella lista dei frame liberi
 - Se a livello hw risulta presente e settato il **dirty bit** allora la pagina va anche copiata in memoria secondaria
 - Se il dirty bit non risulta presente, la pagina va sempre copiata in memoria secondaria

L'algoritmo di sostituzione viene eseguito da un **processo ciclico attivato in background** (anche detto **deamon** o demone) detto **pagedaemon**. In Unix, la sostituzione delle pagine viene attivata quando il numero totale di frame liberi è ritenuto insufficiente. I **parametri** dell'algoritmo sono:

- **lotsfree**: numero minimo di frame liberi per evitare la paginazione su richiesta
- **minfree**: numero minimo di frame liberi necessari per evitare lo swapping dei processi
- **desfree**: numero minimo di frame desiderabili

$$lots_{free} > des_{free} > min_{free}$$

Lo **scheduler** attiva l'algoritmo di sostituzione cioè il pagedaemon se il numero di frame liberi è sovraccarico, ovvero:

- Carico elevato
- Numero di frame liberi < \$min_{free}\$
- Numero medio di frame nell'unità di tempo < \$des_{free}\$

Allora lo **scheduler** attiva lo **swapper**.

Il sistema evita che il pagedaemon usi più del 10% del tempo totale di cpu, per evitare trashing: attivazione (al massimo) ogni 250ms.

3.3.9 Conclusioni

Punto di Vista dell'Utente

- La VM risulta dà l'**illusione** di avere memoria **illimitata**: lo spazio di indirizzamento logico può essere **maggior**e di quello fisico.
- Il **tempo di ricircolo** migliora con l'aumentare della disponibilità di memoria fisica

Punto del Sistema Operativo

- Abbiamo la possibilità di **variare** la **quantità** di memoria allocata ad ogni processo in modo da variare il grado di multiprogrammazione e quindi la percentuale di utilizzo della cpu
- Siamo in grado di utilizzare **maggiormente** la memoria: le parti non utilizzate dei processi non vengono caricate
- Tramite la paginazione, frammentazione esterna e compattazione vengono praticamente annullati

Svantaggi

- **Maggior** complessità sia hw che sw
- Problema del **thrashing**
- Il tempo di ricircolo **medio** tende ad aumentare

5. Gestione dei file - Livello Logico

La parte del S.O. che si occupa della gestione dei file è la parte più visibile di un sistema operativo: fornisce meccanismi per la memorizzazione (su memoria secondaria), l'accesso e la protezione di dati e programmi del S.O. e degli utenti. **Il gestore dei file** è quella parte del S.O. che si occupa dell'organizzazione generale dei dati che risiedono in memoria secondaria (**permanente**). Anche il file system "virtualizza" dispositivi di memorizzazione permanente fornendone una visione logica uniforme. Quindi il sistema di gestione dei file **deve nascondere** all'utente tutti i dettagli relativi ai dispositivi di memorizzazione che sono usati come memoria secondaria, infatti ogni dispositivo ha caratteristiche fisiche diverse, il sistema operativo deve fornire una **astrazione** di lavoro con una visione logica e metodi di accesso uniformi tramite una interfaccia comune ed efficiente allo stesso tempo. La struttura logica con cui il sistema operativo gestisce i file prende il nome di **file system (logico)**

5.1 File System Logico

Vedremo più in dettaglio, quando parleremo del punto di vista **interno**, che il gestore dei file è stratificato. In particolare, lo strato più esterno è quello che viene chiamato **file system logico** che è responsabile per:

- Struttura dei dati nei file
- Organizzazione in directory
- Fornire all'utente un insieme di funzioni di alto livello per operare su di essi, mascherando le operazioni che vengono realmente effettuate per allocare la memoria di massa e per accedervi in lettura/scrittura
- Garantire protezione e condivisione di file

Il **file system logico** garantisce una gestione dei file indipendente dalle caratteristiche fisiche dei dispositivi che costituiscono la memoria secondaria: astrazione utile sia per l'utente sia per i programmi.

5.2 File

Un file rappresenta un insieme di informazioni correlate e memorizzate nella memoria secondaria → **persistenza**. Dal **punto di vista dell'utente**, un file è la più piccola porzione di memoria secondaria indirizzabile logicamente: i dati possono essere scritti nella memoria secondaria soltanto all'interno di un file. Dal **punto di vista del S.O.** i file vengono allocati su dispositivi fisici di memorizzazione non volatili (memoria secondaria o di massa). Un file viene identificato sempre tramite un **nome**:

- Una sequenza limitata di caratteri: la massima lunghezza dipende dal S.O. come l'insieme di caratteri leciti
- Il S.O. può fare o meno distinzione fra maiuscole e minuscole: case-sensitive o case-insensitive.
- Alcuni S.O. dividono il nome in due parti separate da un punto '.' la parte dopo il punto si definisce **estensione**. In alcuni S.O. l'estensione rappresenta il tipo del file.

5.2.1 Attributi dei File

Oltre al **nome**, che identifica il file per l'utente, ogni file ha **altri attributi** detti **metadati**:

- Tipo
- Locazione
- Dimensione
- Identificatori dell'utente che ha creato/possiede il file
- Protezione: permessi d'accesso in lettura, scrittura ed esecuzione
- Ora, data di creazione, modifica, ultimo accesso: dati per il controllo d'uso

5.2.2 Tipi di File

Il tipo e' un attributo di un file che ne indica la struttura logica interna e permette di interpretarne correttamente il contenuto. Alcuni S.O. gestiscono diversi tipi di file: conoscendo il tipo del file, il S.O. potrebbe evitare alcuni errori comuni, un S.O. che riconosce il tipo di un file può manipolarlo in maniera corretta. Esistono tre tecniche principali per identificare i tipi di file:

- **Estensioni**: il tipo viene indicato da un suffisso (estensione) del nome (MS-DOS)
- **Attributo tipo** associato al file (MacOS)
- **Magic number**: valore posto all'inizio del file (unix)

5.2.3 Struttura Logica dei File

A seconda del S.O. i file possono avere una delle seguenti strutture logiche:

- Nessuna: sequenza di parole o byte
- Sequenza a record semplici: linee, record di lunghezza fissa variabile
- Strutture più complesse: documenti formattati, archivi (ad albero, con chiavi, etc...), eseguibili rilocabili, etc...

La **struttura** viene decisa o dal S.O. o dall'applicativo che genera il file. Il tipo di un file e la corrispondente struttura logica possono essere riconosciuti e gestiti in modi diversi nei diversi S.O.

- Se il S.O. gestisce molti formati:
 - Codice di sistema ingombrante
 - Incompatibilità di programmi con file di formato differente
 - Gestione efficiente per i formati supportati
- Altrimenti il S.O. **non** gestisce specifici formati:
 - Codice di sistema più snello
 - Format gestiti dal programmatore
 - I file sono considerati semplici stream di byte
 - Solamente i file eseguibili hanno un formato predefinito dal S.O.

5.2.4 Directory

L'accesso ai file avviene attraverso l'uso di nomi simbolici gestiti dal File System. I file sono **oggetti persistenti**, bisogna preservare il collegamento fra nome del file e le informazioni contenute in esso. Un **descrittore** associa il nome di un file con i suoi attributi (metadati) in particolare la sua locazione sul dispositivo fisico di memorizzazione. I descrittori dei file sono mantenuti in strutture dette **directory**: una directory è un **insieme di descrittori** di file. Queste sono le operazioni che si possono effettuare su una directory:

- Ricerca di file al suo interno

- Elenco totale o parziale del contenuto
- Creazione di un file
- Cancellazione di un file
- Creazione/cancellazione di una directory

5.3 Punto di Vista Utente

Per gli utenti di un S.O. la parte relativa ai file è il **servizio** più visibile. La maggior parte dei comandi forniti dallo strato più esterno del S.O. riguarda i file.

5.3.1 Organizzazione delle Directory

L'organizzazione delle directory caratterizza un S.O. soprattutto dal punto di vista dell'utente: deve garantire **efficienza** nel reperire i file, regole di **nomenclatura** dei file ed infine condivisione e protezione dei file.

Directory ad un solo livello (flat)

Questa rappresenta la più semplice struttura che può avere una directory. Tutti i descrittori di file sono contenuti in un'unica directory, questo approccio era usato specialmente nei primi S.O. monoutente con il **vantaggio** di condivisione immediata ma i seguenti **svantaggi**:

- **Non adatti** a sistemi multiutente
- **Lunghi elenchi** di nomi di file, poco efficiente
- Problema dell'unicità dei nomi che porta ad un naming complesso

Directory a Due Livelli

Per superare le difficoltà legate all'uso di un'**unica** directory in ambito multiutente, **ogni utente** ha una propria directory (ad un singolo livello):

- **1° Livello:** Master File Directory (**MFD**)
- **2° Livello:** User File Directory (**UFD**)

Ogni utente quando accede al sistema, viene posto nel suo UFD, questa directory prende il nome di **home**. Questo approccio comporta i seguenti **vantaggi**:

- Adatto a sistemi multiutente
- Elimina il problema della unicità dei nomi fra utenti diversi
- Semplifica la protezione

E **svantaggi**:

- Lungo elenco di nomi di file, poco efficiente
- Unicità dei nomi nelle relative home directory
- Condivisione non immediata

Condivisione

Gli utenti, come abbiamo appena visto, sono completamente separati e confinati nelle loro home directory. Come fanno quindi a condividere un file nel caso in cui vogliono farlo? L'utente deve avere un modo di referenziare un file che non è nel suo UFD. Ad esempio tramite `username + filename`. Qui nasce il concetto di **path** ovvero percorso. Gli utenti **devono** poter usare **strumenti comuni**, ad esempio, compilatori linker, ecc... Viene definito un UFD **speciale** che contiene gli strumenti comuni, per non dover sempre far riferimento al path completo di questi strumenti lo spazio di ricerca per un eseguibile viene **ampliata**: prima si cerca all'interno del UFD dell'utente e poi in quello speciale, da qui nasce il concetto di **search path**.

Directory ad Albero

Se consideriamo la struttura a due livelli un caso particolare di un albero possiamo facilmente pensare di **estendere** questa struttura ad un numero qualsiasi di livelli, si parla di organizzazione gerarchica a N livelli in cui ogni directory può contenere file e altre directory. Questo consente ad ogni utente di poter **creare sottodirectory**. Vediamo anche per questo approccio i vantaggi:

- Elenco corto di nomi di file, maggiore efficienza
- Eliminato il problema dell'unicità dei nomi per ogni singolo utente, naming semplificato

Gli svantaggi:

- Condivisione ancora non immediata, ma risolvibile tramite **path** e **search path**

E le sue proprietà:

- L'albero ha un'**unica** radice
- Ogni file ha un **unico** path name
- Ogni directory può contenere sia file che **directory**

Ogni utente può definire una directory corrente con la possibilità di cambiarla. I **path name** possono essere relativi (alla directory corrente) o assoluti (a partire dalla root).

Directory a Grafo Aciclico

Per facilitare la condivisione, se due utenti vogliono lavorare sugli stessi file, si concede la possibilità di "vederli" direttamente. Questo è diverso da avere due copie dello stesso file. Lo stesso file viene riferito con path name diversi da utenti diversi. Possiamo generalizzare lo schema ad un **grafo diretto aciclico** (DAG) esistono quindi path name molteplici per ogni file. Con una organizzazione delle directory a **DAG** si ha quello che viene anche detto **aliasing**: due o più path possono venire utilizzati per identificare lo stesso file o directory.

Implementazione Base della Condivisione dei File

1. Link simbolici: l'utente crea una entry particolare nella sua directory, questa identifica il path name completo del file condiviso. `ln -s`
2. Duplicazione descrittori: questo è un link hw, crea un clone dell'originale la quale riferisce il file condiviso `ln`

Nota Bene

I link simbolici sono indispensabili per creare link fra file system fisici diversi e sono necessari per creare link a directory.

Come bisogna gestire la cancellazione di un file condiviso?

1. Nel caso di link **simbolici** se viene eliminato il link non succede nulla all'originale. Se viene eliminato il file il link rimane **dangling** → **problema**
2. Si cancella effettivamente il file solo se non ci sono più link ad esso. In tal caso dobbiamo tenere un contatore dei link per ogni file.

Mantenere il grafo **Aciclico**, facilita l'implementazione di algoritmi necessari per attraversarlo, per esempio durante un backup. Per assicurare l'assenza di cicli bisogna ammettere **solo** link a file, e non a sottodirectory e ogni volta che viene aggiunto un nuovo link, verificare la presenza di cicli. Se si ammette la presenza di cicli, in fase di scansione del file system, occorre marcare, man mano che si percorre il ciclo, ciò che ho già visitato.

Volumi

Un volume è una partizione contenente un file system a livello fisico: **file system fisico**. Nei S.O. semplici, come ad esempio MS-DOS, l'utente doveva conoscere in quale volume era memorizzato il file su cui voleva operare. S.O. più evoluti come unix, rendono **trasparente** all'utente qualunque nozione sui volumi: più volumi vengono montati in un unico **file system logico**.

File System Logico

Un file system logico è in generale composto da più file system fisici: uno di partenza e altri che devono essere **montati** in modo che la loro root sia un nodo e quindi una directory del grafo totale.

5.3.2 Protezione dei File

Soprattutto in sistemi multi-utente e' molto importante **controllare l'accesso** ai file. In genere, per la rappresentazione di politiche di protezione e' necessario introdurre due concetti:

1. **Risorsa**: Le risorse sono le entità da proteggere, ad esempio, file directory, dispositivi... Ad ogni risorsa si associa un utente proprietario, che può concedere o negare ad altri utenti il permesso di accedere ad essa
2. **Dominio di protezione**: Il dominio di protezione viene definito come un insieme di coppie **<risorse, diritti>**. In genere si associa univocamente ad un utente e rappresenta l'insieme delle risorse alle quali e' abilitato ad accedere.

Le possibili operazioni su un file sono lettura (r), scrittura (w), esecuzione (x). Mentre su una directory lettura nomi di file (r) e creazione/cancellazione di un file (w). Ogni processo viene associato ad un particolare dominio, in genere, quello dell'utente che ha richiesto la sua creazione e quindi eredita i suoi diritti (quelli specificati nel suo dominio di accesso). In alcuni S.O. durante l'esecuzione i processi possono cambiare dominio, quindi **variare dinamicamente**. Chiaramente e' compito del S.O. tenere traccia in opportune strutture dati le relazioni tra risorse e domini di protezione. Questo avviene nella cosiddetta **matrice di protezione** dove ogni riga rappresenta un dominio di protezione associato ad un certo utente e ogni colonna rappresenta una risorsa, (file, directory, dispositivo...). Questa matrice a causa della sua vasta dimensione ed elevati tempi di accesso non e' l'approccio consigliato. Gli approcci più comuni sono: liste di controllo e liste di capability.

Liste di Controllo

Una Access Control List rappresenta la politica di protezione associata ad una particolare risorsa (la colonna della matrice di protezione). Quindi per ogni risorsa **R**, vengono elencati i permessi concessi ad ogni utente **U**. Il S.O. autorizza o meno una operazione su un risorsa **R** ricercando nella corrispondente ACL se il processo dell'utente **U** che la sta eseguendo ha i diritti corretti. Sia unix che Windows utilizzano tecniche basate su ACL, in unix ad esempio per ogni file sono tracciati i permessi r w x per: user, group e other, per un totale di 9 bit.

Liste di Capability

Per ogni utente **U**, il S.O. compila una lista di permessi associati alle diverse risorse cui l'utente e' abilitato ad operare (la riga della matrice di protezione). Questa tecnica offre maggiore efficienza rispetto ad ACL: per accedere ad un risorsa al processo viene assegnata, di norma, una **capability**, una specie di riferimento che mantiene anche le informazioni di quali operazioni sono consentite su quella risorsa. Questo approccio ha lo **svantaggio** che una operazione di revoca dei diritti di accesso associati ad una risorsa risulta una operazione onerosa, dato che richiede la ricerca e l'aggiornamento di tutte le c-list o di tutte le capability.

5.4 Punto di Vista del Programmatore di Sistema

Questa categoria di utenti, accede ai file tramite **chiamate di sistema** e vede un file come un **tipo di dato astratto**.

5.4.1 Operazioni

Iniziamo con il vedere quali sono le **operazioni** possibili su un file:

1. Creazione `create(filename)`
 - o Bisogna trovare spazio per la memorizzazione
 - o Creazione di un nuovo descrittore nella directory e quindi associazione nome/locazione
2. Scrittura `write(filename, data)`
 - o Bisogna cercare la locazione del file: nome→descrittore→locazione
 - o Si scrive il dato
3. Lettura `read(filename)`
 - o Bisogna cercare la locazione del file: nome→descrittore→locazione
 - o Si legge e ritorna il dato
4. Cancellazione `delete(filename)`
 - o Bisogna cercare la locazione del file: nome→descrittore→locazione
 - o Bisogna recuperare lo spazio
 - o Cancellazione o invalidazione del descrittore nella directory, distruzione associazione nome/locazione

Osservazione

In tutte le operazioni (a parte la `create`), il S.O. deve ricercare il descrittore del file: se si devono eggerre una serie di operazioni di lettura o scrittura questo risulta molto **inefficiente**. Introduciamo quindi due operazioni che agiscono su una struttura dati mantenuta in memoria centrale: la **tabella dei file aperti** per raggiungere una maggior efficienza.

5. Apertura `open(filename)`
 - o Bisogna cercare la locazione del file: nome→descrittore→locazione
 - o Si inserisce questa informazione (file control block) nella tabella dei file aperti
6. Chiusura `close(filename)`
 - o Si elimina l'informazione relativa al file dalla tabella dei file aperti

In alternativa la `open` ritorna un indice nella tabella dei file aperti (detto **file descriptor**) e le funzioni che ne hanno bisogno usano quel file descriptor al posto del filename. Una operazione di `open` può servire anche per stabilire la modalità di accesso che si vuole usare: lettura, scrittura, entrambe. La modalità richiesta deve essere concessa dal meccanismo di protezione dei file a seconda dei diritti che si hanno su tale file. In base alla modalità dichiarata, nel caso di accesso contemporaneo di più utenti e/o processi allo stesso file il S.O. può adottare schemi per garantire la **consistenza** delle informazioni.

Metodi di Accesso

In base al metodo di accesso definito dal S.O. o scelto dal programmatore, le scritture e le letture avvengono in maniera diversa:

1. Sequenziale
 - o Metodo di accesso ai file più semplice

- Simile come principio alla lettura di un nastro
- Risulta impossibile la lettura oltre l'ultima informazione scritta, la scrittura aggiunge informazioni in fondo al file

Con il metodo di accesso sequenziale le operazioni di lettura e scrittura agiscono sul dato riferito dalla posizione corrente all'interno del file (**file pointer** o **I/O pointer**) all'atto della apertura in lettura, il file pointer viene posizionato all'inizio del file, con la creazione invece alla `end_of_file`. In genere tramite l'operazione **rewind** e' possibile tornare all'inizio.

2. Accesso diretto (casuale)

- Metodo di accesso a file che si "ispira" al disco, il file viene visto come un array di record (di dimensione fissa) si può accedere direttamente al **record numero N** → accesso diretto in uno dei seguenti modi:
 1. Indicando esplicitamente a quale informazione si vuole accedere
 2. Introducendo un'altra operazione **seek** che sposta la posizione corrente del file pointer

Il **vantaggio** di questo approccio e' che necessita di solo 3 primitive: due per leggere/scrivere con accesso sequenziale e una per spostare il file pointer. In questo modo le operazioni di read e write rimangono **inviate**: operano sempre in modo sequenziale.

Esistono **altri** metodi di accesso costruiti a partire dal metodo di accesso diretto: ad esempio, **accesso ad indice** usato per la gestione di basi di dati.

5.5 Punto di Vista del Sistema Operativo

Il gestore dei file e' stratificato, cioè organizzato in livelli: ogni livello si serve delle funzioni dei livelli inferiori per creare nuove, impiegate dai livelli superiori.

5.5.1 Strati del File System

Partendo dall'alto abbiamo:

1. **Logical File System**: quello che abbiamo già trattato
 - Gestisce i metadati, cioè tutte le strutture del file system eccetto i dati veri e propri memorizzati nei file
 - Mantiene le strutture di file tramite i descrittori dei file, che contengono le informazioni sui file quali proprietario, permessi, posizione del contenuto...
 - Gestisce le directory
 - Gestisce protezione e sicurezza
2. **File Organization Module**
 - Traduce gli indirizzi logici di blocco in indirizzi fisici
 - Contiene il modulo per la gestione dello spazio libero
3. **Basic File System**
 - Invia comandi generici I driver di dispositivo per leggere/scrivere blocchi fisici su disco
 - Gestisce il buffer del dispositivo e la cache che conserva i metadati
4. **I/O Control**
 - Traducono comandi di altolivello in sequenze di bit che guidano l'hw di I/O a compiere una specifica operazione in una data locazione

La struttura a strati e' utile per ridurre la complessità e la ridondanza, ma aggiunge overhead nel e può diminuire le performance.

5.5.2 Struttura Interna dei File

Dispositivi di memorizzazione usati come memoria secondaria: **disco rigido**, dischi ottici, unità a stato solido, memorie flash... Trattiamo in particolare dei **dischi**. In un disco si indirizzano **blocchi** di dati, non singole word o byte come in memoria centrale, la dimensione dei blocchi, detti anche **record fisici** e' fissata a 4KB. La dimensione del **record logico** e' diversa da quella del bocco e possibilmente anche variabile. Bisogna risolvere il problema della corrispondenza fra record logici e record fisici. Questo problema viene risolto attraverso il **packing** ovvero l'impacchettamento di più record logici in un unico blocco. Allocare lo spazio disco in termini di blocchi vuol dire che l'ultimo blocco di un file non sarà completamente pieno causando **frammentazione interna**, maggiore e' la dimensione del blocco e maggiore sarà la frammentazione interna.

5.5.3 Directory

Come abbiamo già visto le directory devono essere mantenute in **memoria secondaria** si copiano in memoria centrale solo quelle in utilizzo. Anche le directory quindi vengono trattate come file, possono essere **organizzate** dal S.O. in diversi modi:

1. Ogni entry di una directory contiene il nome e gli altri attributi del file
2. Ogni entry di una directory contiene il nome e un riferimento ad una struttura separata che contiene gli altri attributi del file (vedremo questa nello specifico)

Organizzazione di Una Directory

L'organizzazione più flessibile e' quella che prevede che le informazioni che a livello astratto devono essere memorizzate all'interno di un directory siano collocate in due strutture dati separate.

1. Directory dei Nomi dei File (**DNF**) mantiene la corrispondenza fra **nome logico** del file e un identificatore interno assegnato dal S.O. In unix l'id interno si chiama i-number e quindi in una directory per ogni file si ha: <filename, i-number> Esiste una DNF per ogni directory definita dall'utente.
2. Directory Base dei File (**DBF**) Mantiene tutte le altre informazioni di un descrittore astratto di file, in unix e' l'insieme degli i-node. L'indicatore interno che si trova nella DNF, viene usato come indice all'interno della DBF.

Vantaggio: questa organizzazione consente di realizzare facilmente il meccanismo di **link hw**. Due entry differenti in due DNF che fanno riferimento allo stesso identificatore, una sola entry nella DBF e contatore d'uso nella entry della DBF

Osservazione: la DNF della directory radice e la DBF hanno un indirizzo fisico fisso all'interno del disco

Nel momento in cui si deve **accedere** ad un file, bisogna tramite il suo nome accedere al DNF interessato e ritrovare l'**identificatore interno**, trovare nella DBF l'indirizzo fisico di partenza, oltre a tutte le altre informazioni. **Ottimizzazione:** con l'operazione open, si porta, si porta in memoria il valore dell'id e la riga corrispondente del DBF in una struttura dati mantenuta in memoria centrale → File Control Block (FCB). Questi FCB vengono raccolti in una struttura dati detta **tabella dei file aperti**. In unix si hanno:

1. Una tabella dei file aperti per ogni processo
2. Due tabelle di kernel (globali):
 1. Tabella dei file aperti di sistema
 2. Tabella dei file/i-node attivi di sistema

Realizzazione delle Directory (DNF)

1. Lista lineare di nomi di file
 - o **Vantaggio:** semplice da implementare
 - o **Svantaggio:** esecuzione onerosa dal punto di vista del tempo di ricerca, complessità lineare nel numero di elementi contenuti nella directory
2. Lista ordinata
 - o **Vantaggio:** migliora il tempo di ricerca. Utile per produrre l'elenco ordinato dei file contenuti nelle directory
 - o **Svantaggio:** l'ordinamento deve essere mantenuto a fronte di ogni inserimento/cancellazione
3. Tabella hash: lista lineare con struttura hash
 - o **Vantaggio:** migliora il tempo di ricerca nella directory, inserimento e cancellazione costano O(1), se non si verificano collisioni
 - o **Svantaggio:** dimensione fissa e necessità di rehash

5.5.4 Montaggio di un File System Fisico

Un file system fisico deve essere montato prima di poter essere acceduto dagli utenti e quindi dai processi di un S.O. La posizione dove viene montato viene detta mount point o punto di montaggio. Alcuni S.O. richiedono che i file system fisici che possono essere montati siano solo di un tipo prefissato, altri ne supportano diversi e sondano le strutture del dispositivo per determinare il tipo di file system fisico presente.

Procedura di montaggio

Innanzitutto si deve fornire al S.O. il nome del dispositivo da montare e il punto di montaggio, normalmente il punto di montaggio è una directory vuota cui sarà agganciato il file system fisico che deve essere montato. La directory su cui viene montato un file system fisico punto anche non essere vuota, ma nel momento in cui si effettua il montaggio, i dati ivi contenuti non sono più visibili fino all'operazione di **umount**. Una volta montato, il file system fisico risulta accessibile agli utenti (e ai loro processi). In unix Linux e MacOS diventa parte integrante del file system logico. Il montaggio può avvenire in maniera automatica (MacOS, Windows) oppure manuale (Linux).

6. Gestione dei File - Livello Fisico

6.1 Realizzazione del File System

Le funzioni del file system vengono realizzate tramite chiamate di sistema (API), che utilizzano **dati** gestiti dal S.O. residenti sia su disco che in memoria.

1. Strutture dati del file system residenti su disco:
 1. Blocco di controllo di avviamento (Boot Control Block): contiene le informazioni per l'avviamento di un S.O. da quel volume, ad esempio boot block per Unix e partition boot sector per Windows. Questo blocco è necessario se il volume contiene un S.O. e normalmente occupa il primo posto nel volume.
 2. Blocchi di controllo dei volumi (Volume Control Block): contengono dettagli riguardanti la partizione, quali numero totale dei blocchi e loro dimensione, contatore dei blocchi liberi e relativi puntatori; ad esempio superblocco UFS in Unix e master file table in NTFS

3. Strutture delle directory: usate per organizzare i file, ad esempio in UFS comprendono i nomi dei file e i numeri di i-node
4. Descrittori dei file: contengono dettagli sul file come permessi, dimensione, date di creazione/ultimo accesso/modifica, puntatori ai blocchi di dati. NTFS, memorizza questi dati nella master file table utilizzando una struttura stile DB relazionale
2. Strutture dati del file system residenti in memoria:
 1. Tabella di montaggio: contiene le informazioni relative a ciascun volume montato
 2. Directory cache: contiene informazioni relative a tutte le directory cui i processi hanno avuto accesso di recente
 3. Tabella dei file attivi (di sistema): contiene una copia del descrittore di file per ciascun file aperto nel sistema insieme con altre informazioni **FCB**.
 4. Tabella dei file aperti per ciascun processo: contiene un puntatore all'elemento corrispondente nella tabella di sistema, più informazioni di accesso specifiche del processo
 5. Buffer per la lettura scrittura

6.2 Partizioni e Montaggio

Una partizione può essere un volume contenente un S.O. o essere dotata della sola formattazione di basso livello cioè una sequenza non strutturata di blocchi. Nella fase di avviamento, il S.O. non ha ancora caricato i driver dei dispositivi e quindi non può usare i servizi messi a disposizione dal file system (non può interpretarne il formato). La partizione di avviamento è una serie sequenziale di blocchi, che si carica in memoria come un'immagine. Questa immagine può contenere più informazioni di quelle necessarie al caricamento di un unico S.O. (boot loader). Nella fase di caricamento del S.O., si esegue il montaggio della partizione radice (root partition), che contiene il kernel del S.O. ed eventualmente altri file di sistema. A seconda del S.O., il montaggio degli altri volumi avviene automaticamente in questa fase o si può compiere successivamente in modo esplicito.

6.3 Gestione dello Spazio su Disco

Abbiamo visto come viene gestita la memoria centrale di sistema, chiaramente anche quella secondaria ha bisogno di logiche e politiche per la sua gestione. Bisogna tenere traccia dei **blocchi liberi** e di quelli **allocati** ai file. I fattori di cui deve tenere conto per una buona strategia di allocazione sono:

1. Velocità di esecuzione dell'accesso sequenziale, dell'accesso casuale e dell'allocazione/deallocazione dei blocchi: data la maggiore frequenza degli accessi nei confronti delle operazioni di allocazione/deallocazione, la velocità degli accessi risulta più importante
2. Possibilità di utilizzare trasferimenti multipli di settori: i trasferimenti multipli di settori consentono di limitare il tempo di accesso
3. Grado di utilizzo del disco: si intende la percentuale di spazio totale del disco allocabile agli utenti, la frammentazione (esterna) abbassa il grado di utilizzo
4. Quantitativo di memoria centrale necessario per ogni algoritmo: alcuni algoritmi necessitano di strutture dati mantenute in memoria centrale

La gestione dello spazio su disco assomiglia al problema di gestione della memoria centrale ma con **due nuovi aspetti**:

1. Gli **accessi** ai dischi sono di alcuni ordini di grandezza **più lenti**
2. Il **numero di blocchi** è almeno di un ordine di grandezza **più grande**

Inoltre c'è una elevata variabilità nelle dimensioni dei file, risulta quindi impossibile fare delle previsioni sulle richieste delle risorse. La gestione dei blocchi su disco comporta una **frammentazione interna** indipendente dalla politica di allocazione che viene adottata: dipende dalla dimensione del blocco e da quella del file. Lo spreco medio risulta di mezzo blocco per file, osserviamo la similitudine con il problema di frammentazione interna nel caso di paginazione. Alcuni sistemi adottano uno schema di allocazione che si basa su **cluster** di allocazione (numero intero di blocchi) riducendo il tempo per l'allocazione ma aumentando la frammentazione interna. Come abbiamo visto per la memoria centrale i metodi di allocazione per quella secondaria si suddividono in:

1. Allocazione contigua
2. Allocazione non contigua

6.3.1 Allocazione Contigua

Ogni file viene allocato su blocchi di disco contigui, per reperire il file occorrono solo la locazione iniziale e la lunghezza.

Vantaggi

1. Questo approccio comporta un **accesso sequenziale** e diretto **facilitato**
2. Possibilità di sfruttare trasferimenti multi-settore

Svantaggi

1. Tende a generare **frammentazione esterna**, la formazione di cluster troppo piccoli, ma che globalmente sarebbero sufficienti per le richieste di uno o più file.

Per far fronte a questo problema bisogna effettuare una **compattazione periodica** (squeeze o deframmentazione), deve essere effettuata fuori linea per evitare accessi ai file e risulta essere piuttosto costosa in termini di risorse.

2. La allocazione contigua richiede che le **dimensioni** del file siano dichiarate **in anticipo** al momento della creazione, questo risulta facile nel caso di creazione di file per effettuare una copia ma difficile in caso di creazione di file in generale, ad esempio, per una fase di **editing**.

Per risolvere questo problema si effettua una **stima iniziale** della dimensione del file e se poi questa viene superata:

- Si genera un errore: questo comporta perdita di tempo per l'utente e di spazio per il sistema in caso di sovrastime.
 - Si ricopia il file in un cluster di maggiore dimensione: perdita di tempo per il sistema
 - Si usa uno (o più) cluster non contiguo: bisogna fare attenzione ai casi di **file overflow**, periodicamente (ad esempio durante il compattamento) si può ricostruire la contiguità fisica.
3. Settori difettosi: l'allocazione contigua rende difficile "aggirare" i settori di disco difettosi, aumentando la frammentazione esterna.

6.3.2 Allocazione non Contigua

Per quanto riguarda le politiche di allocazione non contigua vedremo principalmente le varianti di **concatenamento** e **indicizzazione**.

Concatenamento

Ogni file e' una **lista** concatenata **di blocchi**, sparsi ovunque nel disco, vengono riservati alcuni byte in ogni blocco (o in ogni cluster di allocazione) per **puntare** al blocco successivo (viene inserito il numero del prossimo blocco). Quando un file deve essere **creato** si crea un descrittore del file (in una directory), che punta a NIL.

Vantaggi

1. Eliminato il problema della **frammentazione esterna**, non e' più necessaria la compattazione
2. Risulta facile "aggirare" i settori di disco difettosi, basta eliminarli dalle liste dei file oppure dalla lista dei blocchi liberi
3. Accesso sequenziale facilitato
4. Possibilità di modificare la dimensione del file su necessità

Svantaggi

1. Impossibilità di usare trasferimenti multisettore
2. Accesso diretto **impossibile**: richiederebbe di accedere a tutti i blocchi che precedono nella lista
3. Possibile deterioramento dei puntatori

I punti 2 e 3 possono essere risolti mantenendo i puntatori in una tabella dedicata su disco, quindi per quanto riguarda il 2, basta copiare questa tabella in memoria centrale per velocizzare gli accessi casuali, mentre per il 3, basta avere copie ridondanti della tabella su disco, come garanzia per eventuali deterioramenti. Questo metodo nello specifico viene utilizzato in Windows e si chiama **File Allocation Table (FAT)** e per contenere la FAT si riserva una sezione del disco all'inizio di ciascun volume.

Allocazione File con FAT

FAT indica la File Allocation Table, **cluster** invece un aggregazione di settori adiacenti (uno o più blocchi adiacenti). La FAT non e' altro che una tabella che contiene i riferimenti ai cluster liberi ed occupati, quindi al suo interno ha un elemento per ogni cluster. In ogni disco ci sono **due copie** della FAT. Sia la FAT che la directory radice () sono memorizzati in posizioni fisse del disco.

Indicizzazione

L'allocazione indicizzata e' simile al metodo precedente dato che mantiene i puntatori ai blocchi allocati. La differenza consiste nel **miglioramento** nella velocità di accesso diretto: i puntatori vengono raggruppati in **blocchi indice**. Il descrittore del file (sempre considerato dal punto di vista astratto) **non** contiene l'indirizzo del primo blocco su disco, ma l'indirizzo del blocco indice. Il blocco indice copre una funzione simile a quella della tabella delle pagine. Quando un **file** deve essere **creato** si crea un descrittore del file (in una directory), che punta al blocco indice che contiene tutti i puntatori a **NIL**.

Vantaggi

Come nel concatenamento viene eliminata la frammentazione esterna e si possono aggirare i settori difettosi. In più l'**accesso sequenziale e diretto** sono possibili solo se il blocco indice viene portato in memoria.

Svantaggi

1. File corti: nel caso in cui ci siano file di piccole dimensioni si spreca una grande quantità di memoria con il blocco indice, per questo il blocco indice deve essere il **più piccolo** possibile tenendo conto però del punto 2.

Un'alternativa per risolvere questo problema consiste nel mantenere in ogni descrittore di file un certo numero di puntatori ai primi blocchi del file (sufficienti per file corti) e un puntatore ad una lista di blocchi indice o vari puntatori a blocchi indice di primo e secondo livello (vedi punto 2)

2. Dimensione massima dei file: La dimensione massima dei file dipende dalla dimensione del blocco e da quella dei puntatori. In genere la dimensione del blocco indice è quella di un blocco.

Esempio

Con un blocco indice di 512 Byte e una dimensione dei puntatori di 4 Byte (128 puntatori) si hanno $128 * 512 = 64KB$ di dimensione massima di un file. Limite chiaramente **inaccettabile**.

Per rimediare a questo tipo di problema si ricorre all'indicizzazione a più livelli con una delle seguenti implementazioni:

1. Il blocco indice contiene come ultimo puntatore NIL se il file è piccolo (singolo livello di indicizzazione), altrimenti contiene il puntatore ad un altro blocco indice (secondo livello di indice), etc...
2. Il primo blocco indice contiene i puntatori a blocchi indice (secondo livello), etc... In genere due livelli di indici sono sufficienti

6.3.3 Conclusioni

In conclusione il miglior metodo per l'allocazione dei file dipende dal tipo di accesso:

- Il metodo di allocazione contigua ha ottime prestazioni sia per l'accesso sequenziale che casuale
- Il metodo di allocazione concatenata si presta in modo molto naturale per l'accesso sequenziale

Alcuni S.O. combinano questi due metodi di allocazione in base alla modalità di accesso. Questo implica che al momento della creazione si specifichi anche se la modalità di accesso è casuale (allocazione contigua) oppure sequenziale (allocazione concatenata)

- Il metodo di allocazione indicizzata risulta essere il più complesso: l'accesso ai dati può richiedere più accessi al disco (tre, nel caso di un indice a due livelli), i blocchi indice devono essere caricati in memoria centrale e necessita ottimizzazioni nel caso di file corti

La soluzione più semplice è quella di mantenere in un **array di bit** lo **stato** del blocco corrispondente su disco: **libero** (1) o **allocato** (0) → **bit map**. In questo modo, per trovare gli spazi liberi basta iterare il vettore cercando bit con valore 1, copiando il vettore in memoria centrale si queste operazioni possono essere effettuate velocemente. Per i dischi attuali ci possono essere difficoltà nel mantenere la bitmap in RAM. Questo metodo si addice molto bene alla gestione dei file contigui, in alternativa all'array si potrebbe implementare come:

1. Lista concatenata: si usa lo **stesso** schema di allocazione dei file con allocazione non contigua a concatenamento. Quindi si collegano tutti i blocchi liberi mediante puntatori (l'ultimo blocco indica in modo opportuno il termine della lista) e si mantiene un puntatore alla testa della lista in una zona riservata del disco, solo questo puntatore alla testa viene copiato in memoria centrale minimizzando gli sprechi di spazio.

Osservazioni: quando si cerca un solo blocco libero la lista così realizzata risulta efficiente (si stacca il primo blocco libero e si riconcatena il puntatore alla testa col secondo). Quando invece si cercano n blocchi liberi consecutivi si rischia di dover scorrere tutta la lista, con tempi di attesa piuttosto lunghi (bassa efficienza). Nella FAT, l'informazione sui blocchi liberi viene inclusa nella struttura dati per l'allocazione e non richiede quindi un metodo di gestione separato. Vediamo come risolvere questo problema di efficienza nei prossimi punti

2. Il primo blocco libero contiene gli indirizzi di altri \$n-1\$ blocchi liberi più un puntatore al successivo (anch'esso con la stessa struttura), la lista si trasforma quindi in un albero a 2 livelli
3. Counting: se ci sono n blocchi liberi consecutivi viene memorizzato un puntatore al primo e poi il numero di blocchi, si mantiene una lista contenente un indirizzo del disco, che indica un blocco libero, ed un contatore (che indica da quanti altri blocchi liberi contigui e' seguito. La lista si accorta drasticamente

6.4 Osservazioni Generali

6.4.1 Efficienza

L'**efficienza** dipende da:

- Tecniche di allocazione del disco e algoritmi di realizzazione/gestione delle directory
- Tipi di dati conservati nel descrittore del file
- Preallocazione delle strutture necessarie a mantenere i metadati
- Strutture dati a lunghezza fissa o variabile

6.4.2 Prestazioni

Le **prestazioni** dipendono da:

- Mantenere dati e metadati "vicini" nel disco
- Disporre di **buffer cache**, cioè sezioni dedicate della memoria centrale in cui si conservano i blocchi usati di frequente. Se devono essere effettuate scritture sincrone risulta impossibile usare il buffering/caching dato che l'operazione di scrittura su disco deve essere completata prima di proseguire l'esecuzione. Le scritture asincrone, che sono le più comuni, sono invece bufferizzabili.

Sempre in merito alle prestazioni bisogna sottolineare che l'I/O mappato in memoria impiega una cache delle pagine, mentre l'I/O da file system utilizza la buffer cache del disco (in memoria centrale). Molti S.O. unificano la cache del disco con la cache delle pagine, una buffer cache unificata prevede l'utilizzo di un'**unica cache** per memorizzare sia le pagine dei file mappati in memoria che i blocchi trasferiti per operazioni di I/O ordinario da file system. L'algoritmo più ragionevole per questo tipo di cache e' **LRU** a meno che non si tratti di letture **sequenziali** in qual caso si ricorre ai metodi **free-behind** e **read-ahead** che consistono nel rimuovere una pagina dalla cache non appena si verifica la richiesta della pagina successiva e nel caricare in anticipo alcune delle pagine successive a quella appena richiesta. Un ulteriore metodo per aumentare le prestazioni di lettura e scrittura su disco e' creare **dischi RAM** che consiste nel adibire parte della RAM ad un disco virtuale, il cui contenuto viene periodicamente riscritto su un supporto permanente.

6.4.2 Ripristino (Recupero)

I file e le directory sono mantenuti sia in memoria RAM (parzialmente) che su disco, bisogna assicurarsi che malfunzionamenti del sistema non comportino la perdita di dati o la loro incoerenza. Se **blocchi critici** vengono modificati ma non salvati mai (perché acceduti frequentemente) si rischia l'inconsistenza in seguito ai crash, si devono quindi dividere i blocchi in due categorie:

1. Blocchi **critici**: blocchi necessari a garantire la consistenza del file system, ogni modifica deve essere immediatamente trasferita al disco
2. Blocchi **dati**: se riutilizzato a breve, rimane nella cache

Le modifiche ai blocchi dati devono essere riportate su disco anche prima della loro sostituzione, in modo:

- Asincrono: ogni 20-30 secondi (Unix, Windows)
- Sincrono: ogni scrittura viene immediatamente trasferita anche al disco (write through cache)

Ci possono essere altri problemi se il crash si verifica in situazioni di modifica di metadati. Se si ha un crash si possono avere incoerenze fra le strutture, il contatore dei descrittori liberi potrebbe indicare un descrittore allocato, ma la directory non contenere ancora un puntatore all'elemento relativo! Esistono due approcci al problema della consistenza del file system:

1. Curare le inconsistenze dopo che si sono verificate, con programmi di controllo della consistenza. Si deve utilizzare un **verificatore di coerenza** come **fsck** in Linux e **chkdsk** in Windows. Questo tipo di programma confronta i dati nella struttura di directory con i blocchi di dati sul disco e tenta di fissare le eventuali incoerenze, sono lenti e non sempre funzionano.
2. Prevenire le inconsistenze tramite i **Journaled File System** (JFS).

I dispositivi di memoria di massa hanno un Mean Time Between Failures (MTBF) relativamente breve, quindi i sistemisti hanno dovuto implementare soluzioni per aumentarne l'affidabilità:

1. Aumentare l'affidabilità dei dispositivi, ad esempio tramite configurazioni **RAID**
2. **Backup** (automatico o manuale) dei dati dal disco ad altro supporto attraverso:
 1. Dump fisico: direttamente i blocchi del file system, veloce ma difficilmente incrementale e non selettivo
 2. Dump logico: porzioni del file system, può essere completo, differenziale o incrementale; può essere più selettivo, ma a volte troppo astratto (link, file con buchi, etc...). In ogni modo, il recupero dei file perduti (o interi file system) dal backup può essere fatto o dall'amministratore o direttamente dall'utente.

6.5 Generalizzazione del Concetto di File

Il concetto di file può essere usato anche come astrazione del sistema di ingresso/uscita. Questo vuol dire che anche le periferiche come ad esempio mouse e tastiera sono trattate come file. L'utente può usare un unico ed uniforme insieme di servizi per trattare i file e l'I/O che è indipendente dalle periferiche. I programmi usano un'**astrazione** dei dispositivi → standard input e standard output. Il collegamento fra l'**astrazione** e il dispositivo avviene a tempo di esecuzione usando la **ridirezione** mediante appositi metacaratteri (<,>,|) e chiamate di sistema (close, open, create). Chiaramente esistono associazioni di **default** come standard input → tastiera e standard output → video. In Unix, per fornire questa astrazione unica le periferiche stesse sono viste come **file**, in particolare sono situate nella sottodirectory `/dev` in questa directory troviamo file

organizzati a blocchi (dischi) e file organizzati a caratteri (terminali e stampanti). Su tutti questi dispositivi/file si utilizzano le operazioni di open, close, read e write. In alcuni S.O. viene fornito un meccanismo detto **pipe** che permette di fare comunicare due processi, stabilisce un canale di comunicazione unidirezionale con bufferizzazione. Da un lato della pipe, un processo scrive dei dati e , dall'altro lato un altro processo li può leggere. Si usano le stesse operazioni che si usano per file e dispositivi.

6.5.1 Network File System

Il Network File System (NFS) e' un buon esempio di **file system di rete**. Definito e implementato dalla SUN sulla base dei protocolli TCP/IP risulta disponibile nella maggior parte delle distribuzioni Unix e Linux e in alcuni S.O. per PC. Nel contesto del NFS, si considera un insieme di stazioni di lavoro interconnesse in rete come un insieme di stazioni indipendenti con file system indipendenti. Lo scopo dell'NFS e' quello di consentire un certo grado di condivisione fra questi file system, sulla base di richieste esplicite, ma **in modo trasparente all'utente**. La condivisione e' basata su una **relazione client-server** una stazione può essere sia un client che un server. Affinché una directory remota sia accessibile in modo trasparente a una certa stazione cliente, questa deve prima effettuare una operazione di **montaggio** nel proprio file system locale, la directory remota si specifica come argomento dell'operazione di montaggio in modo esplicito. Dopo tale operazione la directory remota risulta accessibile in **modo trasparente** tramite il file system locale. Potenzialmente ogni file system o ogni directory in un file system, nel rispetto dei diritti di accesso, si può montare in modo remoto in una qualsiasi directory locale. Se si monta un file system condiviso in corrispondenza delle directory iniziali degli utenti in tutte le stazioni in una rete locale, un utente può aprire una sessione di lavoro e usare i propri file da una qualunque delle stazioni in rete → **mobilità degli utenti**.