

# INTEGRAZIONE SISTEMI OPERATIVI BARBAGALLO

scritto amatoriale unione degli appunti di Scionti, Farina, Casano e Terranova – 2021

\*per dubbi su spiegazioni ed esercizi consultare i testi originali e/o materiale esterno

## 1. Introduzione

### 1.1 Modalità di Privilegio

In un SO si distinguono spesso diverse modalità di privilegio. In alcuni sistemi sono diverse, ma principalmente sono due:

- Modalità Kernel(super user): Pieni poteri, la CPU funziona senza nessuna preclusione, accesso a qualsiasi componente hardware, possibilità di utilizzo di ogni istruzione prevista dalla architettura.
- Modalità Utente(user): Poteri limitati, i processi in questa modalità hanno limitazioni e gli è negato l'uso di operazioni riservate e di istruzioni privilegiate, accesso ad alcune periferiche precluso, non si può accedere alla unità di mappatura della memoria(definisce le aree di memoria utilizzabili dai diversi processi).

Lo scopo di questa doppia modalità di privilegio è principalmente per motivi di sicurezza, affinché nessun processo possa inficiare la stabilità del sistema. Il SO è ovviamente eseguito in modalità Kernel, mentre i vari programmi sono eseguiti in user mode per questioni di sicurezza, anche la UI (User Interface) è eseguita in modalità utente.

### 1.2 Approcci Descrittivi

La definizione di Sistema Operativo segue un concetto di astrazione (strutturazione della descrizione di sistemi informatici complessi):

- Il SO come Macchina Estesa: L'hardware (HW) ha una propria interfaccia che è però a basso livello, è quindi di difficile interazione con i livelli soprastanti. Tramite il SO possiamo interagire più facilmente con HW, infatti questo trasforma "l'ugly interface" dell'HW in un interfaccia più "user friendly" e quindi più fruibile dalle applicazioni attraverso le chiamate di sistema (System Call). Agevola l'operato dell'utente rendendo le operazioni più facili ed intuitive.

- Il SO come Gestore di Risorse: Il SO mette a disposizione dell'utente una comoda interfaccia, gestisce tutti i pezzi che compongono un sistema più complesso, è colui che ha il pieno controllo. Ha il compito di distribuire in maniera efficace ed efficiente le risorse disponibili ai vari programmi che competono tra loro per usarle. Deve gestire e proteggere le componenti che sono sfruttate in concorrenza da vari utenti/programmi. I sistemi odierni sono Multi-core, quindi riescono a gestire più richieste contemporaneamente, e Multi-utente, quindi può essere utilizzato contemporaneamente da più utenti. Il SO deve saper gestire bene la CPU, deve cederne l'uso ciclicamente a processi diversi così da soddisfare le varie richieste. Il suo compito principale è tenere traccia di chi sta usando quale risorsa e soddisfare le richieste degli utenti/programmi mediandole se risultano conflittuali.

### **1.3 Chiamate di Sistema**

Ogni SO nasce con lo scopo di fornire servizi semplificati ai programmatori. Questi servizi sono fruibili tramite delle chiamate di sistema (Syscall). Quando si invoca una chiamata di sistema, si ha un passaggio da modalità User a modalità Kernel. Questo passaggio di modalità è gestito dall'istruzione TRAP, che garantisce che una volta eseguita la routine di sistema si rientri in modalità Utente. Questo meccanismo è oneroso, per questo i sistemi moderni tendono a fare il più possibile in modalità utente, circoscrivendo i compiti da eseguire a pieni poteri.

#### **1.3.1 Interrupt Hardware**

Un interrupt è un evento asincrono che fa scattare l'esecuzione di una procedura legata a sé stesso, che svolge un compito.

Esiste un gestore hardware di interrupt, una sorta di coda nella quale vengono inseriti gli interrupt che devono essere segnalati al processore.

Al verificarsi di un interrupt, si salvano i registri del processore in uno stack, e si esegue la routine corrispondente all'interrupt.

Per invocare la routine corrispondente ad un interrupt, l'hardware predispone una tabella di 256 indirizzi, che contiene le diverse procedure.

Al verificarsi di un interrupt l'hardware stesso sa dove andare a prendere la procedura corrispondente.

## **1.4 Processore (CPU):**

Il sistema operativo collabora a stretto contatto con l'hardware. Nel corso della storia, in alcuni casi l'hardware è venuto incontro alle esigenze dei SO, in altri casi sono stati i sistemi ad adattarsi.

La risorsa da amministrare in modo più preciso possibile è senz'altro la CPU. Il ciclo di base di un processore prevede il prelievo delle informazioni (fetch), la loro decodifica (decode) e la loro esecuzione (execute).

Tutte le CPU contengono al loro interno alcuni registri per mantenere momentaneamente risultati temporanei, variabili e chiavi.

### **1.4.1 Registri:**

- Program Counter (PC): Contiene l'indirizzo di memoria della prossima istruzione, una volta che l'istruzione viene recuperata il PC viene aggiornato e punterà all'istruzione successiva.
- Program Status Word (Parola di stato del programma, PSW): È un registro che contiene vari flag e valori che rappresentano lo stato del processo in un determinato momento. Inoltre contiene uno o più bit legati allo stato della CPU (modalità kernel/user) e vari bit di controllo. Non è consentito di agire su tutti i campi del PSW, ma solo su determinati bit.
- Stack Pointer (Puntatore a pila, SP): È un puntatore che punta alla cima dello stack corrente in memoria. Ogni stack contiene una struttura (frame) per ogni procedura ancora non terminata, contiene le variabili di ingresso, le variabili locali e temporanee che non vengono contenute in altri registri.

### **1.4.2 Progetti:**

Esistono diversi progetti per la CPU:

- Pipeline: È un organizzazione che consente alla CPU di eseguire più azioni diverse in maniera contemporanea: al contrario del ciclo base (dove fetch, decode e execute sono eseguiti uno dietro l'altro un processo alla volta), la CPU potrebbe avere unità separate per il fetch, il decode e l'execute. Per esempio, mentre il processo N è in execute, la CPU fa il decode del processo N+1 e contemporaneamente la fetch del processo N+2.
- Superscale: Consiste in unità multiple di esecuzione, vengono recuperate due o più istruzioni per volta, vengono decodificate e riservate in un buffer in attesa di essere eseguite. Non appena un'unità di esecuzione si libera,

pesca dal buffer un'istruzione che è in grado di gestire e la esegue. In questo modo le istruzioni non saranno eseguite secondo l'ordine pensato dal programmatore.

### **1.4.3 Multithreading (Hyperthreading)**

Permette di mantenere in contemporanea due stati di diversi processi nella stessa CPU, questo avviene quando un processo effettua una chiamata TRAP ed è in attesa di risposta dal SO, mentre ciò avviene il SO commuta il processore per servire un altro processo. Il Multithreading migliora l'efficienza, non si tratta di esecuzione parallela, ma abbatte i tempi del non uso della CPU.

Nei sistemi Multi-processor (sia fisici che virtuali) ogni processore ha il suo set completo, è indipendente dagli altri, può utilizzare tutto il set messo a disposizione dal nostro sistema. I sistemi Multi-processor sono stati soppiantati da sistemi Multi-core.

Nella GPU possiamo trovare anche centinaia di core, questo perché deve essere una componente efficace ma che comunque non ha la necessità di avere un eccessivo numero di task disponibili per funzionare al meglio, lavora su un insieme limitato di dati, ma ha una capacità di elaborazione elevatissima.

## **1.5 Concetti di Memorizzazione**

MMU ("Memory Management Unit"): è una classe di componenti hardware che gestisce le richieste di accesso alla memoria generate dalla CPU.

- Registri della CPU: costruiti con la stessa tecnologia della CPU, quindi sono veloci quanto questa e non c'è nessun ritardo di accesso. La capacità di memorizzazione è di 32x32 bit (CPU a 32 bit) o di 64x64 bit (CPU 64 bit).
- Memoria Centrale (Random Access Memory, RAM): memoria rappresentativa delle capacità di lettura-scrittura della macchina. E' meno costosa, quindi ve ne è maggiore quantità.
- Altri dischi (Hard disk, SSD): memorie di dimensione massiccia per il salvataggio dei dati. Costi molto bassi, quindi enormi quantità.

### 1.5.1 Cache

Serve a mantenere indirizzi di dati frequentemente utilizzati, pronti per essere recuperati in fretta. Possono esistere diversi livelli di cache (generalmente L1 ed L2), più piccoli sono, più veloce è l'accesso, ma tutti sono molto costosi. Se l'esito della ricerca sulla cache è nullo, si passa il dato alla memoria centrale. La stratificazione può essere implementata in diversi modi in sistemi multi-Core: cache condivise implicano uso di meccanismi di lock, cache dedicate implicano gestione della coerenza dei dati. 1° Livello - Cache intrinseca al core.

2° Livello - Può essere condivisa dai core (INTEL) oppure suddivisa/dedicata nei vari core (AMD).

Nel caso la cache sia suddivisa e un core modifica un dato su cui sta lavorando un altro core, sorge un problema di sincronizzazione interna che deve essere risolto dalla progettazione HW.

### 1.6 Dispositivi di I/O

\* Controller: Tra ogni dispositivo ed il software c'è di mezzo un controller, a cui viene delegata la complessità della gestione di una tipologia di dispositivi. Un controller è un componente che ha una sua sorta di CPU ed uno spazio di memorizzazione interni.

caratteristiche:

-- più uscite verso più dispositivi dello stesso tipo

-- i dispositivi controllati vengono comandati con uno standard (esempio dischi ottici: standard SATA).

-- una interfaccia singola verso il software: è una astrazione per il SO, che può comunicare con il controller attraverso specifiche "porte di comunicazione".

\* Driver: L'interfaccia tra il controller e il SW non è standardizzata, e per questo esistono i driver. Un driver è un insieme di procedure utili alla gestione del controller, esso è fornito dai costruttori hardware del controller.

### **1.6.1 Dialogo tra SO e Controller:**

L'interfaccia esposta al SO dal controller consiste in una serie di registri accessibili dal SO. Questi registri devono essere impostati in maniera corretta per eseguire determinate operazioni, e la codifica delle operazioni utilizzata è appunto conosciuta dal driver.

Tuttavia, ci sono due modi di comunicare con un controller:

- Istruzioni IN/OUT
- Mappatura in memoria dei registri I/O del controller (lavoro con dei classici load/store in memoria): in tal caso bisogna stare attenti ai processi utente, mappando solo i registri strettamente necessari nel loro spazio di indirizzamento.

### **1.6.2 Modalità di I/O**

Le operazioni di I/O devono essere eseguite in modalità Kernel. I controller comunicano con il SO attraverso il PCI EXPRESS, che usa dei BUS seriali (e non condivisi).

- \* Busy waiting: : Un programma dà il via ad una chiamata di sistema che il kernel traduce in una chiamata di procedura al driver appropriato. Dopodiché il driver avvia l'operazione di I/O, entrano in un ciclo finché il dispositivo di I/O non finirà il lavoro restituendo qualcosa. Fatto ciò il driver darà i risultati al kernel (se ce ne sono) e restituisce il controllo al chiamante. Questo metodo ha lo svantaggio di tenere occupata la CPU anche quando non è necessario farlo, privandola agli altri processi.
- \* Interrupt: Alla richiesta di dati il processo in attesa lascia la CPU ad altri processi. All'arrivo dei dati il controller lancia un interrupt a lui dedicato per notificare il sistema, e la CPU va a prendere i dati e li salva nello spazio di indirizzamento del processo che li aveva richiesti.
- \* DMA: Un controller ausiliario che ha il compito di prendere i dati e salvarli nello spazio di indirizzamento del processo che li ha richiesti. Quando un processo richiede dati ad una periferica, esegue una chiamata bloccante e fornisce informazioni riguardo dove vuole trovare questi dati una volta avuti. Una volta salvati i dati nel suo spazio di indirizzamento, il sistema viene notificato avendo il ritorno dalla chiamata bloccante stessa (meccanismo simile agli interrupt). Questo solleva la CPU da compiti del genere.

## **1.7 Tipologia di Sistemi Operativi**

### **1.7.1 Sistemi per Mainframe**

Devono lavorare con una grande quantità di dati ed è possibile installare applicativi supplementari. E' sicuramente il caso dove si gestisce meglio l'I/O.

### **1.7.2 Sistemi per Server**

Lavorano con una quantità minore di dati, ma sempre molti, ed è possibile installare applicativi supplementari. Si deve rispondere a molte richieste: si sfrutta il parallelismo.

### **1.7.3 Sistemi per Personal Computer**

Meno dati, meno carico. Possibilità di installare applicativi supplementari. Sono comunque multiprogrammati, non ostante noi spesso usiamo una applicazione alla volta, il sistema ha in esecuzione diversi processi che si contendono la CPU. Le esigenze in questo caso, sono concentrate sulla reattività di utilizzo, per garantire una buona esperienza d'uso.

### **1.7.4 Sistemi per Smartphone**

Ridurre al minimo il consumo di batteria, e quindi regolamentare la potenza computazionale richiesta dai processi. Possibilità di installare applicativi supplementari, con molte restrizioni sul controllo che hanno dell'hardware.

### **1.7.5 Sistemi Embedded(integrati)**

Sono sistemi chiusi e progettati per funzionare come sono stati costruiti. Questo implica una facile gestione ed una ottima efficienza. Sono molto semplici a livello hardware e software (esempio: elettrodomestico).

### **1.7.6 Sistemi Real-Time**

Si riferiscono a sistemi industriali (hard-real-time), o multimediali (soft-real-time). Sono calcolatori che controllano macchinari per linee di produzione, catene di montaggio e così via dicendo.

Bisogna garantire che le operazioni legate agli eventi siano Tempestive, per evitare danni fisici. Si raggiunge la tempestività grazie alla collaborazione tra i processi, che, si sa a priori essere “buoni” perché scritti appositamente. Pochi controlli di sicurezza.

## **1.8 Struttura di un Sistema Operativo**

Nella sua accezione più generica un SO è una collezione di procedure. Sono così tante che nel corso degli anni ha avuto senso pensare di suddividere queste procedure, con diversi modelli:

### **1.8.1 Struttura Monolitica**

E' la struttura più difficile, codice poco strutturato e progetto molto grosso. Struttura interna quasi inesistente e difficilmente modificabile: il SO consiste in un insieme di procedure che possono richiamarsi a vicenda. Per costruire il vero programma oggetto del SO prima tutte le procedure vengono compilate e in seguito sono legate tra loro tramite un linker di sistema. Non esiste codifica né mascheramento: tutte le procedure sono visibili le une alle altre e possono liberamente interagire tra loro. Non ha supporto HW ed è poco gestibile.

### **1.8.2 Struttura a Livelli/Strati**

E' possibile creare un sistema a livelli che utilizza funzioni degli strati inferiori per offrire servizi a strati superiori che li richiedono tramite chiamate TRAP (ogni volta effettuata la chiamata TRAP se ne controlla la validità). È un modello più semplice da controllare e da sviluppare (usa un incapsulamento tipo OOP). Principalmente ha problemi di prestazioni, dovuti alle numerose chiamate nidificate.

I livelli possono essere rappresentati da cerchi concentrici, che operano come nella struttura a livelli ma con una limitazione maggiore.

Vi è una separazione hardware forzata tra i livelli: il livello  $x$ , per usufruire di un servizio offerto dal livello  $x-2$ , deve passare per il livello  $x-1$ . Questo è garantito dall'HW con un meccanismo simile a quello della TRAP: diverse modalità di privilegio, una per ogni livello, ed ogni area di memoria ha una maschera che indica il privilegio minimo per accedervi.



### **1.8.3 Microkernel**

E' un kernel(nucleo del SO) ridotto ai minimi termini, contenente solo le procedure che hanno bisogno di stare a stretto contatto con l'HW.

Tutto il resto dei compiti possono essere svolti da normali processi SW in modalità utente.

Diventa quindi necessaria la comunicazione tra il kernel, ed i processi generici, che avviene tramite messaggi.

Ha un miglior design ed è semplice da sviluppare e gestire.

Essendo ogni processo isolato dall'altro, se ci fossero problemi con una parte del SO, basta terminare il processo, senza mettere in ginocchio il sistema. Quindi maggiore robustezza e stabilità.

Come svantaggio vi è l'overhead dovuto alla comunicazione tramite messaggi elevato. (esempio Windows NT che aveva la GUI poco reattiva)

### **1.8.4 Struttura a moduli**

Il kernel principale ha funzionalità ridotte e, sfruttando i concetti base della programmazione ad oggetti, dispongo di moduli che posso caricare dinamicamente, in base alle mie esigenze. Essi poi possono comunicare tra loro ogni modulo ha il suo utilizzo specifico e la propria interfaccia definitiva per comunicare con l'esterno.

Tutti i moduli stanno all'interno del kernel, ma vengono caricati dinamicamente all'occorrenza. Dato ciò, la comunicazione tra moduli non prevede un passaggio di modalità, garantendo minore overhead ed efficienza, però potrebbero fare danno e per questo motivo non è possibile inserire moduli terzi.

### **1.8.5 Virtualizzazione**

La virtualizzazione nasce dall'estremizzazione dell'astrazione.

La nostra idea è quella di costruire più macchine astratte complete su una macchina fisica. L'obiettivo è la separazione dei servizi: questo modello è molto utilizzato per via della possibilità di separare diversi servizi offerti.

Infatti, delegando una diversa macchina astratta alla gestione di un servizio, si ottiene una enorme sicurezza, in quanto ogni servizio risulta essere totalmente separato dagli altri. Non stiamo facendo altro che isolare dei sistemi operativi, ognuno con una certa mansione.

A cosa può servire una macchina virtuale?

1. La progettazione dei sistemi operativi: sviluppare, testare un SO è tendenzialmente legato ad una macchina virtuale, ove, con gli opportuni metodi è persino possibile effettuare un “dump” (debug).
2. Per uno sviluppatore si può testare virtualmente come si comporta un sw su più sistemi operativi (sviluppo multi-piattaforma).
3. Il “cloud”, difatti, è relativo ad una macchina virtuale.

L'Hypervisor è il SW che offre l'astrazione della virtualizzazione e gestisce il tutto.

### **1.8.5.1 Tipi di Virtualizzazione**

- Virtualizzazione: in generale, la MV è di per sé una copia del sistema operativo. Può avvenire in modalità kernel sull'hardware o in modalità utente su un SO ospitante.
- Paravirtualizzazione: il sistema ospitato è conscio del fatto di essere ospitato. Qui è più facile gestirne le risorse.
- Simulazione: qui il procedimento è molto simile alla virtualizzazione. Viene simulata una vera e propria macchina fisica. I software girano sulla CPU fisica.

### **1.8.5.2 Tipi Hypervisor**

- Tipo 1: Hypervisor che gira sull'hardware. È una sorta di “mini sistema operativo” (dual-boot).
- Tipo 2: Sistema di virtualizzazione più comune, è quello che quotidianamente viene utilizzato più frequentemente. Qui il SO ospitato verrà utilizzato in modalità utente. (esempio Virtualbox)

### **1.8.5.3 Criticità kernel/utente nella virtualizzazione effettiva**

L'Hypervisor è un processo utente che gira su un SO ospitante. Il SO virtualizzato non sa di esserlo. Esso ha le sue sysCall, e presuppone l'utilizzo della modalità kernel per le procedure interne al kernel. In tal caso si ha un passaggio alla modalità kernel del SO virtuale, che non presuppone necessariamente un passaggio alla modalità kernel del SO fisico.

#### 1.8.5.4 Passaggio a modalità kernel effettiva

Lo si ha quando una procedura kernel del SO virtuale ha necessariamente bisogno di comunicare con l'HW (I/O). In tal caso l'hypervisor si occupa di fare l'opportuna richiesta al SO ospitante, che passerà in modalità kernel e soddisferà la richiesta.

#### 1.8.5.5 Java Virtual Machine

Una macchina virtuale, diversa da quelle viste fin ora, è la Java Virtual Machine(JVM). Questa è stata sviluppata da Sun Microsystem per poter interpretare il codice Java in tutte le macchine provviste della JVM, così da non avere problemi di esecuzione anche nel caso di invio dati.

### ESERCIZIO

*Il tempo di overhead è un parametro fondamentale per lo studio delle prestazioni di un SO. Esso rappresenta il tempo medio di CPU necessario per eseguire i moduli del kernel. È spesso fornito in percentuale rispetto al tempo totale di utilizzo della CPU. Si può calcolare con:*

$$\text{Overhead \%} = \frac{\text{Tempo CPU per l'esecuzione dei moduli del kernel}}{\text{Tempo totale di utilizzo CPU}}$$

*Ovviamente, più il tempo è basso, maggiore sarà la quantità di tempo CPU che si può utilizzare per i processi utente.*

*Nei sistemi a livelli l'inserimento di più strati implica un sostanziale aumento dell'Overhead, diminuendo l'efficienza del sistema stesso*

## 2. Processi, Thread, IPC e Scheduling

### 2.1 Definizione Processo

Un processo è una istanza di esecuzione di un programma.

Un programma è invece il risultato della compilazione di un codice.

Lo stato di un processo (Come insieme di risorse) è definito principalmente da due cose: dati nel suo spazio di indirizzamento e metadati contenuti nel PCB.

### 2.2 Spazio di indirizzamento

È l'area di memoria all'interno della quale il processo può lavorare detta "area di attivazione". Questa area di attivazione ha diverse sezioni:

- Sezione text: nella quale si trova il codice eseguibile, ad indirizzo 0.
- Area di dati statici: contiene tutte le variabili globali, attive dall'inizio dell'esecuzione, alla fine. Questa area è spesso un tutt'uno con la sezione text.
- Sezione Stack: usata per l'allocazione automatica, variabili locali, passaggio parametri ecc. Esso cresce verso il basso, andando verso la successiva sezione.
- Heap: sezione di allocazione dinamica. L'Heap cresce verso l'alto, andando verso lo stack. Se Stack e Heap si toccano, abbiamo esaurito lo spazio in RAM. È quello che succede quando facciamo ad esempio ricorsione infinita. Tendenzialmente nell'heap, con operazioni di allocazione e deallocazione, si finisce per creare dei buchi.

### 2.3 Process Control Block (PCB)

È una tabella che contiene dei dati satellite utili al SO per la gestione del processo. Ce n'è uno per processo. Per quanto riguarda lo scheduling tra i processi da parte del SO, è fondamentale questa tabella, per poter ripristinare lo stato di un processo che era andato in "background".

I dati nel PCB sono generalmente:

- PID( Process Identifier)
- Utente proprietario
- Tabella dei file aperti

- Copia della PSW
- Copia dei contenuti dei registri del processore
- Altre informazioni di gestione.

Il SO mantiene una tabella dei processi contenente i vari PCB. Essi transitano poi tra le varie code di sistema in base agli eventi che si verificano.

## **2.4 Multiprogrammazione e pseudo-parallelismo**

La multiprogrammazione consiste nel voler eseguire più processi contemporaneamente, o far apparire che sia così.

In un sistema moncore, ad un dato istante è in esecuzione uno ed un solo processo. Si può pensare che ogni processo possenga la propria CPU e venga eseguito in parallelo agli altri processi, ma sappiamo che in realtà la CPU switcha continuamente processo e li fa avanzare in “falso-parallelismo”. Un primo approccio alla multiprogrammazione è quello di eseguire diversi processi uno dopo l’altro. Potrebbe essere adeguato per sistemi batch, ma non per sistemi interattivi in cui vogliamo far apparire che i processi vengano eseguiti in parallelo.

## **2.5 Creazione di processi**

Un processo viene creato:

\* All’inizializzazione del sistema: all’ inizializzazione del SO si avvieranno in automatico dei processi di background che offrono servizi al sistema, questi sono detti Demoni(Daemons). Questi processi possono essere analizzati usando il Task Manager di Windows.

- Da parte di un processo Padre o da parte dell’utente: spesso un processo effettua delle chiamate di sistema per avviare processi secondari (figli) che lo aiutino nel suo lavoro (questo è utile quando il lavoro può essere facilmente distribuito ed effettuato in “parallelo”).

Quando un utente apre un programma o una cartella, si crea un nuovo processo che gestisce il tutto.

### 2.5.1 Comandi Creazione di processi

Un processo viene creato attraverso dei comandi particolari:

- \* In UNIX si usano fork ed exec: fork crea una copia del processo chiamante, il processo figlio ha la stessa immagine di memoria, gli stessi dati, le stesse stringhe di ambiente del processo genitore. Normalmente il processo figlio esegue una exec per cambiare la sua immagine di memoria ed eseguire un nuovo programma.
- \* In Windows si usa CreateProcess: questa chiamata ha 10 parametri e gestisce la creazione di un processo da 0, gli assegna la memoria, il programma da eseguire, informazioni di priorità, attributi di sicurezza, bit di controllo, puntatori a strutture, ecc.

### 2.6 Terminazione di processi:

Una volta eseguito il suo compito, un processo deve essere terminato, questo succede quando:

- \* Uscita normale (volontario): exit (UNIX), ExitProcess (Win). Una volta che il processo ha terminato il proprio compito, effettua una system call per far notare al SO che ha finito. (exit status=0)
- \* Uscita su errore (volontario): il processo riscontra un errore fatale, qualcosa che non riesce a gestire, un componente che manca per proseguire l'esecuzione. Quindi, in certi cASI, avverte l'utente con un pop-up e si auto-termina. (exit status>0)
- \* Errore critico (involontario): errori di programmazione, bug, divisioni per 0, accesso a memoria inesistenti, ecc. Sono errori che il processo non può gestire da solo, interviene quindi il SO: se è un errore che riesce a gestire e risolvere non è un grande problema, ma esistono errori non contemplati dal SO e che comportano un invio di segnale di chiusura al processo che si auto-termina.
- \* Terminato da un altro processo (involontario): Un processo con autorità necessaria invia sua system call richiedendo la terminazione di un altro processo attraverso dei comandi (kill (UNIX), TerminateProcess (Win)).

## 2.7 Stati di un processo

Nella sua vita, ogni processo può trovarsi in 5 stati differenti (il primo e l'ultimo sono opzionali):

- Creazione: Il processo viene creato, aspetta l'ammissione nel ciclo della CPU.
- Pronto: il processo è pronto per essere scelto dallo schedatore ed essere eseguito
- In esecuzione: il processo è in esecuzione, è stato scelto dallo schedatore, rimane in questo stato o finché termina il suo lavoro o finché viene interruptato.
- Bloccato: il processo ha subito un interrupt, è in attesa di un qualche evento esterno per continuare il suo lavoro
- Terminato: il processo ha finito il suo lavoro, non ha bisogno di compiere altre azioni e si termina.

### 2.7.1 Context Switch tra processi

E' la fase in cui un processo lascia la CPU in favore di un altro.

Per implementare il modello a processi il SO mantiene una tabella (vettore di strutture) chiamata tabella dei processi (Process Table) dove ogni elemento di questa struttura rappresenta un processo ed è il PCB.

Come detto prima, ogni PCB contiene le informazioni utili al processo e alla sua esecuzione in modo tale che, quando ne sarà ripresa l'esecuzione, sembrerà che non sia mai stato fermato.

Com'è possibile dare l'illusione che una sola macchina dotata di una sola CPU faccia lavorare una molteplicità di processi contemporaneamente e sequenzialmente? Ogni classe di dispositivi I/O ha associata una locazione detta vettore delle interruzioni (Interrupt Vector) che contiene l'indirizzo della procedura per la gestione delle interruzioni. Se un dispositivo chiama un'interruzione, il PC, lo stato e alcuni registri del processo corrente sono salvati nello stack corrente dell'HW dedicato alle interruzioni e la CPU salta all'indirizzo specificato dall'interrupt vector.

### 2.7.2 Scheduler

Lo Scheduler è lo strumento che sceglie, tra i possibili processi in “pronto” quale sarà il candidato ad utilizzare la CPU. Come lavora? Non guarda la tabella dei processi, poiché dinamica e scomoda. Bensì considera la coda dei processi pronti (coda di priorità). È un elemento dinamico, una lista linkata composta dai PCB dei processi nello stato di “pronto”. È chiaro che all’interno della coda possono essere presenti alcuni processi più importanti di altri, la coda di priorità interviene a tal proposito.

### 2.7.3 Context Switch Schema

Schematizzando (esistono comunque variazioni da SO a SO):

- 1) Salvataggio dei registri;
- 2) L’HW carica un nuovo PC dal vettore delle interruzioni;
- 3) Salvataggio registri e impostazione di un nuovo stack;
- 4) Esecuzione procedura di servizio per l'interrupt (scritta in C);
- 5) Interrogazione dello scheduler per sapere con quale processo proseguire;
- 6) Ripristino dal PCB dello stato di tale processo (registri, memoria);
- 7) Ripresa nel processo corrente.

## ESERCIZIO

Supponiamo che sia in esecuzione il processo utente 3, quando si verifica un’interruzione dal disco: il program counter di questo processo, la parola di stato del programma e magari uno o più registri vengono messi sullo stack corrente dall’HW dedicato alle interruzioni e la CPU salta all’indirizzo specificato nell’interrupt vector. Questo è tutto quello che fa l’HW; da qui in poi è tutto in mano al SW. Azioni come il salvataggio dei registri o l’inizializzazione dello stack pointer, vengono svolte da una piccola routine in linguaggio assembler. Quando questa routine termina, chiama una procedura C per terminare il resto del lavoro per lo specifico tipo di interrupt.

Quando questa ha svolto il suo compito, viene richiamato lo scheduler per vedere qual è il prossimo processo da eseguire. Dopodiché il controllo viene passato al codice in linguaggio assembler perché vengano caricati i registri e la mappa di memoria per il nuovo processo corrente. Tutte queste azioni di salvataggio e ripristino vengono chiamate **context switching**.

## 2.8 Thread

I processi dei SO odierni sono organizzati a thread, dei “processi leggeri” che cooperando portano avanti il lavoro del processo (flussi di esecuzione). Esistono sia processi con un solo thread sia processi che ne contengono molteplici. Il termine multithreading è utilizzato per descrivere la situazione



in cui ad un solo processo sono associati più thread.

Ogni thread ha:

- Il suo PC, che tiene traccia della prossima istruzione da eseguire;
- I suoi registri, che mantengono le variabili utili;
- Uno stack, che contiene la storia dell'esecuzione e un elemento per ogni procedura ancora in esecuzione.

In definitiva i processi radunano le risorse e i thread sono le entità schedate per l'esecuzione nella CPU. Thread dello stesso processo condividono risorse, spazio di indirizzamento e file aperti.

### **2.8.1 Multithreading**

Quando un processo con thread multipli viene eseguito su un sistema con una sola CPU, i thread vengono eseguiti a turno; la CPU passa rapidamente avanti e indietro per i thread e questo è un lavoro più complesso il cambiare il contesto tra processi rispetto al cambiarlo tra thread dello stesso processo. Thread diversi in un processo non sono indipendenti come processi diversi, perché tutti i thread hanno esattamente lo stesso spazio di indirizzamento, cioè condividono anche le stesse variabili globali. Dal momento che ogni thread può accedere ad ogni indirizzo di memoria dello spazio di indirizzamento del processo, un thread può leggere, scrivere o persino cancellare completamente lo stack di un altro thread: non c'è protezione tra i thread perché è impossibile realizzarla e non dovrebbe essere necessaria. Oltre alla condivisione dello spazio di indirizzamento, tutti i thread condividono lo stesso insieme di file aperti, processi figli, allarmi eccetera. Quindi quando si hanno dei lavori da eseguire correlati fra loro, è meglio utilizzare un unico processo con diversi thread, invece se i lavori non sono correlati è preferibile utilizzare diversi processi.

### **2.8.2 Operazioni tipiche sui thread:**

- `thread_create`: un thread ne crea un altro;
- `thread_exit`: il thread chiamante termina;
- `thread_join`: un thread si sincronizza con la fine di un altro thread;
- `thread_yield`: il thread chiamante rilascia volontariamente la CPU.

Quando è presente il multithreading, normalmente i processi partono con solo un thread presente, che ha la capacità di creare nuovi thread

richiamando una procedura dalla libreria, come ad esempio `thread_create` e un parametro che tipicamente specifica il nome della procedura che il nuovo thread deve eseguire. Non è necessario né possibile specificare qualcosa di nuovo sullo spazio di indirizzamento del thread appena creato, dal momento che questo, automaticamente, è in esecuzione nello spazio di indirizzamento del thread che l'ha creato (lo spazio di indirizzamento è condiviso). Con o senza relazione gerarchica (poche volte c'è una gerarchia), al primo thread viene restituito l'identificatore del thread creato, che permette di riferirsi per nome al nuovo thread. Mentre i thread spesso si rivelano utili, introducono un certo numero di complicazioni nel modello di programmazione.

### **2.8.3 Filosofia dei Thread**

Ma non ci bastavano i processi che erano in “falso-parallelismo”?  
Dobbiamo metterci i thread che sono un “falso-parallelismo del falso-parallelismo”?

Ebbene, come abbiamo visto i processi paralleli semplificano non poco il lavoro. Ma adesso abbiamo entità parallele che condividono gli stessi dati e lo stesso spazio di indirizzamento, rendendo possibile un'elaborazione molto più veloce efficiente di prima. Creazione/distruzione di un thread e cambiamenti di stato sono molto più facili e veloci di quelli di un processo, il lavoro parallelo di thread permette una più efficace sovrapposizione di attività, infine nei processori multi-core i thread possono usare il “parallelismo puro”.

### **2.8.4 Hyper-Threading e Multicore**

- \* MonoCore: Unica CPU. Necessario multiplexing temporale. (interleaved)
- \* Hyper-Threading: Presenza di due banchi di registri per CPU, in modo da non dover riprogrammare i registri ma poter mantenere lo stato di due Thread ed eseguire immediati context-switch. Una CPU in Hyper-Threading viene vista dal SO come due CPU virtuali.
- \* Multi-Core: Diverse CPU fisiche in un unico Socket. Spalmiamo i Thread tra le CPU.

## 2.8.5 Programmazione Multicore

La progettazione di SW multi-Threading non è per nulla banale.

Principi di base:

- Individuazione e separazione dei task
- Bilanciamento tra i task
- Suddivisione dei dati e gestione delle dipendenze di dati con mecc. di lock
- Test e debugging consapevole della possibilità di errori saltuari dovuti a sfortunati interlacciamenti tra i Thread

## 2.9.1 Thread a Livello Utente

È detto “modello uno a molti”, utile nel caso il kernel non supporti i thread (molti SO del passato), infatti eseguendoli in questo modo il kernel non sarà a conoscenza della loro esistenza. I thread sono eseguiti sopra un Sistema a tempo di esecuzione (Run-time System) che altro non è che una collezione di procedure che gestiscono i kernel (thread\_create, thread\_exit, ecc..).

Ogni processo ha la propria tabella dei suoi thread (thread table) che tiene traccia dei thread (stato corrente del thread, PC, registri, ecc..).

Quando un thread deve essere bloccato, perché in attesa di dati o di un altro thread, invia una richiesta di blocco al Run-time system. Se questa viene accolta lo stato (PC, registri, ecc...) del thread da bloccare viene salvato nella tabella dei thread, il Run-time system cerca un altro thread pronto all'avvio e ricarica i registri macchina (che si occupano dell'esecuzione delle istruzioni) con quelli del nuovo thread. Fatto ciò scambia lo stack pointer con il program counter e il thread appena pescato entrerà in esecuzione.

Pro: Il livello utente non effettua TRAP e lo scheduling è personalizzato e gestito tra thread dello stesso processo;

Contro: Nel livello utente le chiamate bloccanti sono un grosso problema, è possibile renderle non-bloccanti, ma è molto complicato; inoltre, essendo in modalità utente, se un thread non rilascia spontaneamente la CPU è impossibile bloccarlo, così facendo un unico thread blocca tutti gli altri thread del processo;

### 2.9.2 Thread a Livello Kernel

È detto “modello uno a uno”, in questo caso non serve un Run-time System, e la thread table è una sola ed è contenuta nel kernel. Quando un thread ha bisogno di creare, distruggere un altro thread esegue una richiesta al kernel che tramite TRAP la soddisfa. Tutte le tabelle contenute prima nei Run-time System sono ora nel kernel.

Se un thread si blocca il kernel (a sua discrezione) può eseguire un thread dello stesso processo oppure di un altro processo.

Pro: nel livello kernel un thread bloccante non rallenta gli altri;

Contro: nel livello kernel le chiamate TRAP sono molto più costose e dispendiose, stesso discorso per creazione, distruzione dei thread e commutazione di contesto.

### 2.9.3 Thread Ibrido

Un modo per risolvere i problemi dei due modelli è creare un modello ibrido che ne prenda il meglio: esistono thread a livello kernel che gestiscono vari thread al livello utente. Il kernel gestisce solo una parte dei thread, gli altri sono al livello utente e son gestiti in gruppo dai thread al livello kernel. La maggior parte dei SO moderni supporta i thread a livello kernel, e supportano quelli a livello utente attraverso determinate librerie.

## ESERCIZIO

4 thread da gestire su sistema single core (pseudo parallelismo)

T1 -> T2 -> T3 -> T4 -> T1 -> T2 -> T3 -> T4 -> ...

4 thread da gestire su sistema multi core (parallelismo puro)

core 0 T1 -> T3 -> T1 -> T3 -> ...

core 1 T2 -> T4 -> T2 -> T4 -> ...

### 2.10 Comunicazione fra Processi

C'è la necessità che i processi comunichino tra loro in modo ben strutturato e senza utilizzare le interruzioni, cioè che in definitiva ci sia Interprocess Communication (IPC).

Ci sono tre questioni da considerare:

- Come un processo scambia informazioni con l'altro.
- Essere sicuri che due o più processi non si accavallino l'uno sull'altro.
- Mettere in coordinazione i processi in modo adeguato.

### **2.10.0.1 Pipe e Strutture**

Una struttura semplice e monodirezionale per permettere la comunicazione tra processi.

I dati in output da un processo diventano dati in input di un altro processo. Non distinguono tipi di dati, “inviano” flussi di byte.

Un processo per estrarre dati o inviare dati esegue delle chiamate alla pipe bloccanti. Grazie alle chiamate bloccanti, niente problemi di sincronizzazione. Un altro modo per comunicare è quello di leggere e scrivere sulla stessa struttura dati. Infatti, se allochiamo una struttura dati e la mappiamo nello spazio di indirizzamento di due processi, possiamo farli comunicare, ma qui nascono problemi di sincronizzazione.

### **2.10.1 Corse Critiche**

In certi cASI, processi diversi potrebbero condividere una parte di memoria dove possono leggere-scrivere contemporaneamente. Se due o più processi stanno leggendo o scrivendo su un qualche dato condiviso ed il risultato finale dipende dall'ordine in cui vengono eseguiti i processi, si presenta una Corsa critica (Race Condition).

L'unico modo per evitare le corse critiche è impedire a più di un processo alla volta di lavorare su risorse condivise. Abbiamo bisogno della mutua esclusione (Mutual Exclusion) per fare in modo che una risorsa non sia più accessibile se già in uso da qualcuno. Per semplificarci la vita basterà isolare le parti in cui i programmi accedono alla memoria comune e fare in modo che non collidano tra loro. Queste sezioni sono dette Regioni critiche (Critical Region) o Sezioni Critiche (Critical Section).

### **2.10.2 Esempio Corse Critiche Versamento Conto-Corrente**

Supponiamo di avere due processi P1 e P2, ed una variabile condivisa tra questi due processi.

Supponiamo che sia P1 che P2, si occupano solo di incrementare il valore di questa variabile. Questo significa che leggeranno il valore, gli aggiungeranno 1 e scriveranno il valore.

Considerato che sia P1 che P2 eseguiranno questo codice Assembly

```
MOVE RX,[c]
```

```
Inc RX  
MOVE [c],RX
```

In una iterazione ci aspettiamo che la variabile c sia uguale a 2.

\*Corsa critica:

Prima che P1 riesca a fare l'ultimo Move lascia la CPU per via della prelazione. La variabile condivisa resta quindi a 0.

Viene schedulato P2, che incrementerà la variabile ad 1.

Viene schedulato nuovamente P1, che completa l'istruzione di Move, settando la variabile ad 1.

Con due incrementi, la variabile è arrivata ad 1.

L'interlacciamento compiuto per via della prelazione ha portato ad una corsa critica.

### **2.10.3 Corse Critiche nel Codice del Kernel**

Un problema analogo potrebbe presentarsi anche per le routine del kernel. Se i Kernel sono con o senza prelazione (Kernel preemptive vs. Kernel non-preemptive), quindi se sono in grado di temporaneamente interrompere e portare al di fuori della CPU il processo, senza alcuna cooperazione da parte del processo stesso, al fine di permettere l'esecuzione di un altro processo. Infatti anche tra le routine del kernel è desiderabile lo pseudoparallelismo raggiungibile con l'utilizzo della prelazione.

Il problema è la prelazione?

Niente prelazione: kernel senza prelazione non ricadono in corse critiche, ma sarebbe un grosso limite. Scheduler senza prelazione, non fanno ricadere i processi in corse critiche, ma anche questo è un grosso limite. Quanto detto vale solo per sistemi mono-core.

Conclusioni: Il vero problema non è la prelazione, ma il parallelismo, che è in un certo modo simulato dalla prelazione.

### **2.10.4 Condizioni di Dijkstra**

Per avere processi che cooperino tra loro e che usino dati condivisi in maniera efficiente, si devono soddisfare quattro condizioni:

- 1) Due processi non devono mai trovarsi contemporaneamente all'interno della sezione critica.
- 2) Non si deve fare alcuna ipotesi sulle velocità e sul numero delle CPU.
- 3) Nessun processo in esecuzione fuori dalla sua sezione critica può bloccare altri processi.

4) Nessun processo deve aspettare indefinitivamente per poter entrare nella sua sezione critica.

## **2.11 Mutua Esclusione con Attesa Attiva**

### **2.11.1 Disabilitare le interruzioni (interrupt)**

La soluzione più semplice è di permettere a ciascun processo di disabilitare le interruzioni non appena entra nella sua regione critica, e di riabilitarle non appena ne esce. Questo approccio ha però dei difetti: e se un processo disabilitasse le interruzioni e non le riabilitasse più? Potrebbe essere la fine del sistema. In più, in un elaboratore multiprocessore, con due o più CPU, la disabilitazione delle interruzioni avrebbe effetto solo sul processore che esegue l'istruzione disable, mentre gli altri continuerebbero l'esecuzione e potrebbero accedere alla memoria condivisa. D'altra parte, però, è spesso conveniente che lo stesso kernel disabiliti le interruzioni per poche istruzioni, mentre sta aggiornando variabili o liste importanti, quindi il disable è uno strumento utile ma pericoloso che non è adatto al nostro scopo.

### **2.11.2 Variabili di lock**

Come secondo tentativo, vediamo una soluzione di tipo SW. Supponiamo di avere una sola variabile condivisa (di lock), con valore iniziale a 0. Quando un processo vuole entrare nella sua regione critica, controlla prima la sua variabile di lock e se vale 0, la mette a 1 ed entra, altrimenti aspetta che sia 0. Supponiamo che un processo legga la variabile di lock e veda che contiene 0, ma prima che possa metterla a 1, viene schedato un altro processo che va in esecuzione e setta la variabile di lock a 1. Quando il primo processo torna in esecuzione, imposterà, a sua volta, la variabile a 1 e i due processi saranno contemporaneamente nella loro regione critica non garantendo la mutua esclusione. Il ricontrollo della variabile prima di eseguire le operazioni nella zona critica non funziona.

### **2.11.3 Alternanza Stretta**

Si utilizza una variabile turn, inizialmente posta a 0 che tiene traccia del processo al quale tocca entrare nella sezione critica. Inizialmente, il processo A legge turno (=0) ed entra nella propria regione critica; anche il

processo B trova turno a 0 ed entra in un piccolo ciclo, testando in continuazione turno per vedere quando sarà uguale a 1. Il testare continuamente una variabile si dice busy waiting (in questo caso si parla di Spin Lock). Un lock che usa l'attesa attiva si dice SPIN LOCK. Quando il processo A lascia la sezione critica, pone turno a 1 per permettere al processo B di entrare. Il processo 1 entra e termina rimettendo il turno a 0. Vi è un problema. A però non aveva bisogno di entrare nella sua area critica. B ha di nuovo bisogno di entrare in fase critica ma il turno è di A che non lo rilascia perché esegue codice non critico. Questa proposta non è ottima se un processo è molto più lento dell'altro. Questa situazione viola la terza condizione, ossia, un processo (B) è bloccato da un altro che non è nella propria sezione critica(A).

```
int N=2
int turn

function enter_region(int process)
  while (turn != process) do
    nothing

function leave_region(int process)
  turn = 1 - process
```

```
while (true) do
  while (turn != 0) do
    nothing
  critical_region()
  turn = 1
  noncritical_region()
```

```
while (true) do
  while (turn != 1) do
    nothing
  critical_region()
  turn = 0
  noncritical_region()
```

### 2.11.4 Soluzione di Peterson

Dati N processi, ognuno eseguirà una routine (enter\_region) prima di entrare nella sezione critica e una routine (leave\_reagion) quando ne uscirà, il processo passerà un proprio identificativo alla funzione.

/I campi:

Other -> identificativo dell'altro processo (1 se 0, 0 se 1).

Interested[process] -> manifestazione dell'interesse di entrare.

Turn -> variabile di turno che indica l'identificativo del processo che entra (la imposto col mio ID)./

Il controllo indica che se il processo A vuole entrare nella sezione critica e se B è già nella sua sezione critica, A cicla senza fare nulla. In realtà la variabile "Interested" non è quella fondamentale, quella che lavora è il "Turn": i processi A e B vorrebbero entrare nella sezione critica, impostano "interested" a true.

Se il processo A imposta il proprio turno, e immediatamente il processo B imposta il suo, nel while A vedrà che B è interessato e che il turno non è



suo e quindi entrerà nella sezione critica mentre B ciclerà finché A non si imposterà il suo interested su false, a quel punto entrerà.

Come problema c'è ancora busy waiting, può avere problemi nei moderni multiprocessori a causa del riordino degli accessi alla memoria centrale. La soluzione è facilmente applicabile a 2 processi, aumentandone il numero i controlli diventano molto più complessi (aumentano esponenzialmente).

```
int N=2
int turn
int interested[N]

function enter_region(int process)
    other = 1 - process
    interested[process] = true
    turn = process
    while (interested[other] = true and turn = process) do
        nothing

function leave_region(int process)
    interested[process] = false
```

### 2.11.5 Test and Set Lock (TSL) e XCHG (eXCHanGe)

Questa è una proposta che richiede un piccolo aiuto anche dall'hardware. Molti calcolatori hanno un'istruzione TSL, RX e LOCK.

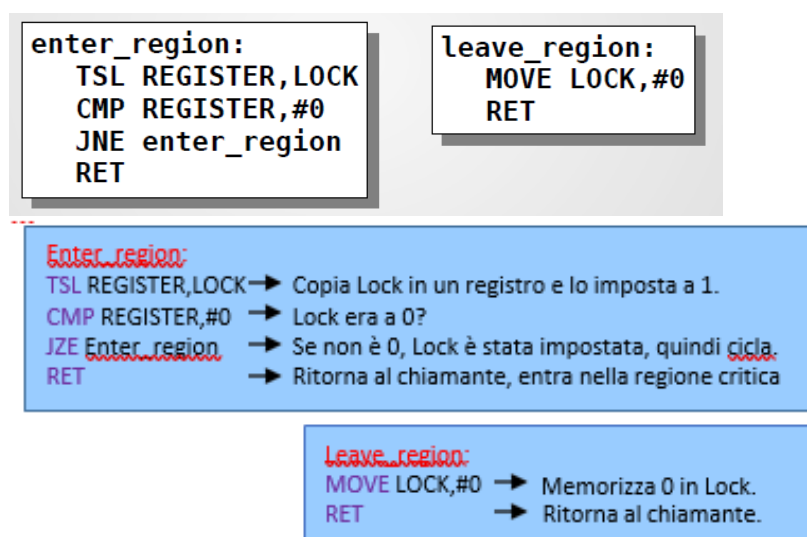
TSL mette il contenuto di una parola di memoria lock nel registro RX e poi memorizza un valore diverso da 0 all'indirizzo di memoria di lock. Le operazioni di lettura e memorizzazione della parola sono garantite indivisibili: nessun altro processore può accedere alla parola finché l'istruzione non è finita (è un'istruzione atomica). La CPU che esegue l'istruzione TSL blocca il bus di memoria per impedire che altre CPU accedano alla memoria.

Per usare l'istruzione TSL, useremo una variabile condivisa, lock, per coordinare l'accesso alla memoria condivisa. Quando lock è a 0, qualunque processo può metterla a 1 usando l'istruzione TSL e poi leggere o scrivere nella memoria condivisa. Quando ha finito, il processo mette lock di nuovo a 0 utilizzando una normale istruzione move. Come può essere usata questa istruzione per garantire la mutua esclusione? La prima istruzione copia il vecchio valore di lock su un registro e mette il valore di lock a 1, dopodiché il vecchio valore di lock viene confrontato con 0. Se non è uguale a 0, il blocco era già impostato, quindi il programma torna all'inizio e

continua a controllare il valore. Per risolvere il problema della sezione critica, un processo chiama Enter\_region che fa attesa attiva finché il blocco non è libero, poi acquisisce il controllo e termina; dopo la sezione critica, il processo chiama Leave\_region che memorizza uno 0 in lock. L'istruzione TSL è una istruzione non privilegiata e quindi eseguibile anche in modalità utente e non solo kernel.

In sintesi: TSL Evoca un restrizione ad una locazione di memoria; Leggo le informazioni del REGISTER le salvo in una locazione, e setto il LOCK a 1. Nel caso sfortunato dove 2 TSL siano eseguite contemporaneamente entrambe setteranno il lock a 1, ma solo una delle due ha letto 0 nel registro, e questa andrà avanti.

- XCHG ("eXCHanGe"): scambia i valori del registro con quelli della variabile di lock. È, difatti, la stessa identica cosa. eXCHanGe gira a x86 32bit.



### 2.11.5.1 Problema dell'Inversione di Priorità

Sia la soluzione di Peterson che quella che usa TSL, sono corrette, ma entrambe hanno il difetto di richiedere busy waiting, il che può provocare dei comportamenti inaspettati: consideriamo un calcolatore con due processi, H con priorità alta e L con priorità bassa; le regole di schedulazione sono tali che H viene mandato in esecuzione non appena si trova in ready. Ad un certo istante, mentre L si trova nella propria regione critica, H passa nello stato di ready. H comincia quindi l'attesa attiva, ma poiché L non viene mai scelto quando H è in esecuzione, L non ha mai la possibilità di lasciare la sezione critica, così H cicla all'infinito.

Questa situazione si chiama problema dell'inversione di priorità. Bisogna dare la possibilità al processo di bloccarsi in modo passivo (rimozione dai processi pronti).

### **2.11.6 Sleep e Wakeup**

Il SO usa l'algoritmo di scheduling per decidere quale sia il processo successivo ad usare la CPU (bassa-alta priorità). Le due primitive:

Sleep -> Un processo chiede al SO di sospendere la propria esecuzione, è una pausa passiva che chiede di eliminare il suo nome dalla coda dei processi pronti e farsi inserire in un'altra coda in riferimento dell'avvenimento che si aspetta.

Wakeup -> Risveglio un altro processo

### **2.11.7 Problema Produttore-Consumatore con Sleep e Wakeup**

Ho N processi o thread che si scambiano informazioni. I processi produttori producono e inseriscono dati, i processi consumatori estraggono i dati dal buffer, tutto ciò in modo concorrente. L'unica variabile condivisa è il count, inizialmente impostato a 0. Quando il buffer è pieno, i produttori dovranno sospendere il proprio lavoro, e aspettare che i consumatori svuotino il buffer per risvegliarsi, analogamente se il buffer è vuoto i consumatori si addormenteranno in attesa dei dati dei produttori.

Il problema anche in questo caso risiede nelle Corse Critiche: se in un certo momento il consumatore legge count=0 si sospenderà, ma tra la lettura e la sospensione il controllo viene dato al produttore che produce ed aggiorna il count a count=1 e manderà un wakeup al consumatore. In questo modo il wakeup verrà perso dato che il consumatore non è ancora sospeso, ma quando il controllo sarà restituito al consumatore, questo, avendo letto 0 e non essendosi accorto dell'aggiornamento, si sospenderà. Una volta che il buffer verrà riempito, il produttore andrà in sleep. Entrambi saranno sospesi per sempre.

Per risolvere il problema si potrebbe aggiungere un bit di riserva dove, se viene mandato un wakeup mentre il processo è sveglio, il wakeup viene salvato e usato in seguito. Questo funziona con 2 processi, ma diventa veramente complicato con più thread e ha un grosso costo di bit di riserva.

```
function producer()
while (true) do
    item = produce_item()
    if (count = N) sleep()
    insert_item(item)
    count = count + 1
    if (count = 1)
        wakeup(consumer)
```

```
function consumer()
while (true) do
    if (count = 0) sleep()
    item = remove_item()
    count = count - 1
    if (count = N - 1)
        wakeup(producer)
    consume_item(item)
```

#### PRODUCER:

produce\_item → crea un elemento

N → elementi massimi del buffer

Count → numero elementi

Insert\_item → inserimento

Count=1 → prima era 0, il buffer era vuoto e quindi il consumer era in sleep

#### CONSUMER:

count=0 → buffer vuoto, entra in sleep.

Remove\_item → prelievo l'item.

Count = N-1 → prima il buffer era N cioè pieno, quindi sveglio il producer che era in sleep.

## 2.12 Semafori

È un miglioramento del sistema Sleep e Wakeup, vi è una variabile intera positiva condivisa tra i processi che può essere modificata tramite due operazioni:

- Up: incrementa la variabile.
- Down: decrementa la variabile. Controlla che il valore sia maggiore di 0, se è così decrementa il valore (cioè consuma una delle sveglie che erano state salvate) e semplicemente continua, mentre se il valore è 0, il processo viene sospeso, per il momento senza completare la down.

Queste operazioni sono implementate in modo atomico, viene garantito che, se il semaforo è occupato, nessun altro processo possa prendere il controllo e accedervi, finché la prima operazione non è terminata.

L'atomicità è assoluta ed inevitabile per evitare le corse critiche.

I semafori non sono previsti dall'architettura, quindi devono appoggiarsi a funzioni e strutture dati, la garanzia di atomicità è data da operazioni effettuate dal kernel in modalità kernel, si disabilitano gli interrupt ad opera del SO per il tempo necessario dell'esecuzione dell'operazione dell'up-down. Nel caso si utilizzino CPU multiple, ciascun semaforo deve essere protetto da una variabile di lock e si deve utilizzare l'istruzione TSL. La disabilitazione degli interrupt non crea alcun problema dato che le operazioni di Up-Down sono veloci da eseguire e occupano veramente poco tempo alla CPU.

### 2.12.1 Tipologie di Semafori

- Semafori Mutex: Usati per garantire la mutua esclusione. In questo caso il semaforo viene inizializzato ad uno ed ogni processo, prima di entrare in sezione critica, esegue una `down()` e poi una `up()` subito dopo essere uscito dalla sezione critica.
- Semafori Conteggio Risorse: Usati per la sincronizzazione tra processi. Ci permettono di contare il numero di risorse, e mandare in pausa un processo non appena non ci sono più risorse disponibili, attendendo quindi un altro processo che si occupi di reimmettere risorse.

### 2.12.2 Problema Produttore-Consumatore con i Semafori

Questa soluzione utilizza tre semafori:

- Pieno (Full): controllare il numero di elementi occupati (inizializzato a 0)
- Vuoto (Empty): controllare il numero di elementi vuoti (inizializzato a N)
- Mutex: garantire la mutua esclusione (inizializzato a 1).

I semafori inizializzati a 1 e che vengono usati da due o più processi per assicurarsi che di volta in volta solo uno di essi possa entrare nella propria sezione critica, sono chiamati semafori binari o semafori Mutex. Se ogni processo chiama una `down` subito prima di entrare nella regione critica ed un `up` subito dopo esserne uscito, è garantita la mutua esclusione. Il semaforo mutex è utilizzato per la mutua esclusione, mentre l'altro uso dei semafori è per garantire la sincronizzazione. I semafori vuoti e pieni sono necessari per garantire che certe sequenze di eventi si verifichino o no in un certo ordine. Questa soluzione reintroduce il busy waiting, ma limitato agli accessi al semaforo e riguarda frammenti di codice di granularità inferiore rispetto

alle regioni critiche generali e di conseguenza ha durata breve e prevedibile. TSO o XCNG hanno il solo problema dello spin lock.

```
int N=100
semaphore mutex = 1
semaphore empty = N
semaphore full = 0
```

```
function producer()
while (true) do
    item = produce_item()
    down(empty)
    down(mutex)
    insert_item(item)
    up(mutex)
    up(full)
```

```
function consumer()
while (true) do
    down(full)
    down(mutex)
    item = remove_item()
    up(mutex)
    up(empty)
    consume_item(item)
```

### 2.12.3 Mutex

I mutex sono ottimi per gestire la mutua esclusione di una risorsa o di codice condiviso, sono facili e semplici da implementare. Un mutex è una variabile che può assumere i valori di “bloccato” o “non bloccato”. Di fatto basterebbe un bit per rappresentarlo, ma in pratica lo si rappresenta con un intero, si usa intendere lo 0 come non bloccato

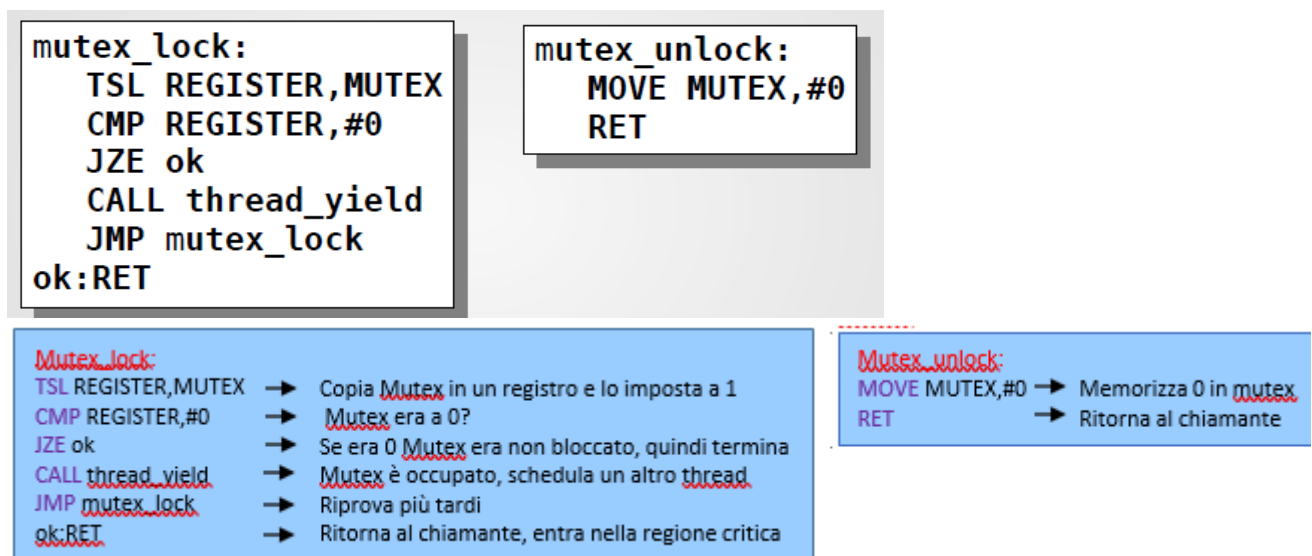
e tutti gli altri valori come bloccato. Ci sono due procedure:

- \* **Mutex\_lock**: quando un thread deve accedere ad una regione critica e il mutex non è già bloccato da altro, richiama questa procedura e si assicura il pieno controllo della risorsa.

- \* **Mutex\_unlock**: quando il thread che è nella regione critica ha terminato, richiama la procedura rendendo il comando agli altri thread.

Dato che i Mutex sono tanto semplici, sono facilmente implementabili nello spazio utente, ma cosa li rende migliori delle soluzioni viste fin ora?

Nei Mutex, se il processo non può entrare perché il Mutex è occupato, non entra in busy waiting, ma lascia il posto e viene schedato un altro thread.



### 2.12.4 Fast User Space Mutex (Futex)

Una variante sono i futex, uno strumento ben preciso usato nei sistemi linux. Ciò nasce da un'osservazione: l'idea è che l'operazione di sincronizzazione è sempre più frequente ed avere l'implementazione più snella possibile migliora le prestazioni. Abbiamo già visto un modello più snello, il TSL e XCHG che non si appoggia sul SO, ma che ha il problema

dello spin lock, problema risolto col semaforo. È possibile usare le TRAP, ma ha i suoi costi. Nel blocco del Mutex troviamo lo spin lock. Nel caso dei semafori quando non c'è contesa in una struttura dati, usandoli si effettua di fatto una chiamata di sistema che cambia la modalità della CPU sprecando tempo.

I futex sono un connubio di entrambi gli scenari. Sono implementati con una doppia componente, una di kernel e l'altra di libreria. Nei futex, in modalità utente posso verificare se c'è contesa tramite una variabile LOCK gestita come visto prima, ma se mi accorgo che c'è contesa e non riesco a prendere il controllo del LOCK, chiedo aiuto alla componente kernel del futex che blocca il thread che ha fatto la chiamata al futex. Quindi se ho poca contesa, non ho particolare bisogno del kernel: basta controllare la variabile LOCK, ma se per caso sono in una contesa, allora chiedo sostanzialmente al SO di bloccarmi e di rimuovere il mio PCB dalla coda dei thread in esecuzione.

## ESERCIZIO

*Supponiamo di avere 3 processi che condividono una variabile x e che i loro pseudo-codici siano i seguenti:*

P1:	P2:	P3:
wait(S)	wait(R)	wait(T)
x=x-2	x=x+2	if (x<0) signal(R)
signal(T)	signal(T)	wait(T)
wait(S)	wait(R)	print(x)
x=x-1		
signal(T)		

*Determinare l'output del processo P3 assumendo che il valore iniziale di x è 1 e che i 3 semafori abbiano i seguenti valori iniziali: S=1, R=0, T=0.*

P2 e P3 attendono su due variabili (R e T) che sono inizialmente poste a zero, dunque vengono sospesi fino a una successiva signal che li faccia uscire dalla coda di attesa. P1, invece, attende sulla variabile S, che inizialmente è posta a uno, dunque non entra in sospensione, ma semplicemente pone il semaforo associato a 0, e continua l'esecuzione portando x, dal valore 1 al valore  $1-2=-1$ . Adesso P1 sblocca un processo dalla coda di T (cioè P3), con signal(T), e torna immediatamente ad attendere su S (su cui, almeno in questo codice, non verrà mai più fatta una signal, e che quindi resterà in attesa per sempre...). Il processo sbloccato (P3) controlla il valore di x; lo trova pari a -1, e  $-1 < 0$ , quindi entra nel ramo then dell'if, eseguendo signal(R), che sblocca un processo dalla coda di R (P2), e torna in attesa su T. Il processo sbloccato dalla coda di R (P2), porta x da -1 a  $-1+2=1$  e immediatamente sblocca un processo dalla coda di T. L'unico che era presente in tale attesa era P3 da pochissimo fa, che, adesso, può finalmente fare la print(x): siccome x contiene 1, P3 stamperà 1, e questa è la risposta.



## 2.12.4 Monitor

Costruito ad alto livello disponibile su alcuni linguaggi, più evoluto dei semafori, ma con limitazioni, sono una versione più semplificata.

I monitor rappresentano un tipo astratto di dato, hanno un proprio stato interno, con variabili che lo rappresentano, e una serie di procedure che operano su questo. Si avvicinano molto al concetto di oggetto.

La mutua esclusione nei monitor si ottiene perché ci può essere solo un flusso di esecuzione che accede al monitor, con una coda di attesa interna.

Il primo difetto è proprio il vincolo di protezione ai dati, per cui le variabili interne al monitor possono essere usate solo dalle procedure interne.

Sebbene i monitor forniscano una maniera semplice per ottenere la mutua esclusione, grazie alla loro proprietà, questo non è sufficiente, perché abbiamo bisogno di un modo per bloccare i processi quando non possono proseguire. La soluzione sta nell'introduzione di variabili di condizione, accessibili solo dall'interno del monitor. Una variabile di condizione è una variabile con l'attributo condizione ed è associata a una condizione nel monitor. Quindi i dati, le variabili e i metodi sono tutti dentro il monitor e i metodi non possono accedere ad eventuali dati esterni e così via.

È un servizio offerto dai linguaggi e non dal SO, ma è implementato usando primitive offerte dai SO (Semafori). Per evitare l'accesso ed avere più flussi che accedono al monitor, basta usare un LOCK sul monitor. Posso avere funzioni `down(L)` e `up(L)` per escludere l'accesso al monitor.

I monitor sono strettamente legati alla programmazione multithread, ma nei monitor abbiamo un vincolo dello strumento stretto. I monitor si possono usare all'interno di un singolo processo. Seppur solitamente, nel modello a più processi che condividono uno spazio di memoria, due processi scritti da persone differenti con linguaggi differenti possono comunicare, ciò non è vero nel caso dei monitor perché è strettamente connesso al linguaggio e chi lo può usare è solo il codice del programma stesso, quindi vi è un modello più restrittivo.

### 2.12.4.1 Comandi Monitor

La `wait()` mette in attesa il Thread che la chiama.

La `signal()` sveglia un Thread che aveva fatto `wait()` sulla stessa variabile.

Quando viene eseguita la `signal` però, potrebbe accadere che il Thread che



l'ha eseguita sia ancora dentro il monitor. Per questo la primitiva signal() può avere diverse semantiche:

- Soluzione tecnica Hoare(signal & wait)

L'idea è quella di bloccare T1 che fa la signal(), e così si sveglia T2 senza possibilità che T1 sia ancora in esecuzione. (autoaddormentamento)

- Soluzione pratica Mesa(signal & continue)

T1 esegue la signal(), ma l'effetto verso T2 si concretizza solo quando finisce l'esecuzione di T1.

- Soluzione Pascal(signal & return)

T1 fa la signal() come ultima operazione e quindi return subito dopo.

## 2.12.4.2 Problema Produttore-Consumatore con i Monitor

```
monitor pc_monitor
condition full, empty;
integer count = 0;

function insert(item)
    if count = N then wait(full);
    insert_item(item);
    count = count + 1;
    if count = 1 then signal(empty)

function remove()
    if count = 0 then
        wait(empty);
    remove_item();
    count = count - 1;
    if count = N-1 then signal(full)

function producer()
    while (true) do
        item = produce_item()
        pc_monitor.insert(item)

function consumer()
    while (true) do
        item = pc_monitor.remove()
        consume_item(item)
```

## 2.12.5 Scambio di Messaggi

E' uno strumento analogo ai semafori offerto dal SO, non ha vincoli e se ho due processi, P1 e P2, possono avere due linguaggi diversi. Questo metodo di comunicazione tra processi usa due primitive, send e receive, che, come i semafori e a differenza dei monitor, sono chiamate di sistema invece che costrutti del linguaggio. Come tali, possono essere facilmente messe in procedure di libreria, come send(destinazione, messaggio) o receive(sorgente, messaggio). SEND spedisce un messaggio ad una determinata destinazione e RECEIVE riceve un messaggio da una determinata sorgente (o da ANY, qualunque sorgente, se al ricevente non interessa una particolare sorgente). Se non è disponibile nessun messaggio, il ricevente potrebbe bloccarsi finché non arriva un messaggio; in alternativa, può terminare immediatamente con un codice d'errore.

### **2.12.5.1 Problemi Scambio di Messaggi**

I sistemi a scambio di messaggi però hanno molti problemi e problematiche di progetto che non si presentano con i semafori o i monitor, in particolare se i processi che comunicano sono su macchine diverse connesse attraverso una rete; ad esempio, i messaggi possono essere persi dalla rete. Per prevenire la perdita di messaggi, il mittente e il destinatario possono concordare che non appena un messaggio viene ricevuto, il destinatario spedisce indietro uno speciale messaggio di acknowledgement (conferma dell'avvenuta ricezione); se il mittente non ha ricevuto la conferma entro un certo intervallo di tempo, ritrasmette il messaggio.

Consideriamo ora cosa succede se il messaggio stesso è stato ricevuto correttamente, ma la conferma viene persa. Il mittente ritrasmetterà il messaggio, così che il destinatario lo riceverà due volte: è essenziale che il destinatario possa distinguere un nuovo messaggio dalla ritrasmissione di uno vecchio. Di solito questo problema viene risolto mettendo numeri di sequenza consecutivi in ogni messaggio originale; se il destinatario ottiene un messaggio che porta lo stesso numero di sequenza del messaggio precedente, saprà che è un duplicato che può essere ignorato.

I sistemi a scambio di messaggi devono anche trattare il problema dell'assegnazione dei nomi ai processi, in modo che il processo specificato in una chiamata send o receive non sia ambiguo.

Anche l'autenticazione è un problema nei sistemi a scambio di messaggi: come fa il cliente a dire se sta comunicando con il file server reale, e non con un impostore?

Ci sono anche problematiche di progetto che sono importanti quando il mittente e il destinatario sono sulla stessa macchina, tra le quali ci sono le prestazioni. La copia di un messaggio da un processo all'altro è sempre più lenta di un'operazione su un semaforo o di un accesso al monitor.

### **2.12.5.2 Indirizzamento e Mailbox Buffer**

\* Indirizzamento con Mail box:

Si ha un buffer detto Mail box, gestito dal kernel.

La receive() estrae messaggi dal buffer ed è una procedura bloccante in mancanza di dati.

La send() diventa bloccante se la mail box ha raggiunto dimensione N, in tal

caso attenderà eventuali estrazioni prima di scrivere messaggi nella mailbox.

I processi comunicano quindi con il S.O. All'interno del kernel risiede una mailbox, in cui vengono copiati i messaggi inviati con la send, e da cui vengono copiati i messaggi dal processo ricevente con la receive.

Svantaggi: Overhead alto sia per il continuo passaggio a modalità kernel, sia per le copie di messaggi effettuate da send e receive.

\* Indirizzamento Diretto:

I processi comunicano direttamente tra loro, senza passare per una zona di memoria condivisa. Può essere più efficiente, ma è una gestione scomoda specialmente quando abbiamo più di due processi che devono comunicare tra loro.

### 2.12.5.3 Libreria MPI

E' possibile estendere questo modello anche a più macchine, facendo comunicare processi remoti, utilizzando i socket. Ovviamente sorgono diversi problemi legati alla comunicazione di rete, come quello di gestire eventuali perdite di messaggi.

```
function producer()
  while (true) do
    item = produce_item()
    build_msg(m,item)
    send(consumer, msg)
```

```
function consumer()
  while (true) do
    receive(producer, msg)
    item=extract_msg(msg)
    consum_item(item)
```

### 2.12.6 Problema dei 5 Filosofi

Il problema può essere formulato abbastanza semplicemente come segue: cinque filosofi sono seduti attorno ad un tavolo tondo e ciascun filosofo ha un piatto di spaghetti. Gli spaghetti sono così scivolosi che per mangiarli ogni filosofo deve avere due forchette e fra ogni coppia di piatti vi è una forchetta. La vita dei filosofi alterna periodi in cui essi pensano ad altri in cui mangiano. Quando un filosofo comincia ad avere fame, cerca di prendere possesso della forchetta che gli sta a sinistra e di quella che gli sta a destra, una alla volta ed in un ordine arbitrario. Qualora riesca a prendere entrambe le forchette, mangia per un po' e, successivamente, depone le forchette e continua a pensare.

Soluzione ovvia: la procedura `prendi_forketta` aspetta fino a che la forchetta specificata è disponibile e ne prende possesso.

Sfortunatamente, la soluzione ovvia è sbagliata. Supponete che tutti e cinque i filosofi prendano la loro forchetta sinistra nello stesso istante: nessuno di loro sarà più in grado di prendere la forchetta di destra e, di conseguenza, ci sarà uno stallo.

```
int N=5
function philosopher(int i)
    think()
    take_fork(i)
    take_fork((i+1) mod N)
    eat()
    put_fork(i)
    put_fork((i+1) mod N)
```

Potremmo modificare il programma in modo tale che dopo aver preso la forchetta di sinistra, il programma esegua un controllo per vedere se la forchetta di destra è disponibile. Se non lo è, il filosofo rimette a posto quella di sinistra, aspetta per un po' e poi ripete l'intero processo. Anche questa proposta non funziona, sebbene per un motivo diverso. Con un po' di sfortuna, tutti i filosofi potrebbero cominciare l'algoritmo contemporaneamente, prendere le loro forchette di sinistra, accorgersi che la forchetta di destra non è disponibile, rimettere giù la forchetta di sinistra, aspettare, riprendere la forchetta di sinistra simultaneamente e continuare così per sempre. Una situazione come questa, nella quale tutti i programmi continuano ad essere in esecuzione per un tempo indefinito, ma nessuno di essi compie dei veri progressi, è nota come *starvation* (letteralmente, morte per fame).

Adesso si potrebbe pensare: "E se i filosofi aspettassero un tempo casuale anziché un tempo fisso fra un tentativo e l'altro di accedere alle forchette, la probabilità che tutto possa andare avanti a bloccarsi anche per una sola ora sarebbe molto bassa." Questa osservazione è vera e in quASI tutte le applicazioni riprovare più tardi non è un problema. Comunque, in alcune applicazioni si vuole una soluzione che funzioni sempre e che non possa fallire a causa di una serie sfortunata di numeri casuali.

La soluzione ultima appare la seguente: prima di prendere possesso delle

forchette un filosofo dovrebbe fare una down su mutex (semaforo). Dopo aver deposto le forchette dovrebbe fare una up su mutex. Da un punto di vista pratico, questa soluzione presenta un problema: solo un filosofo alla volta può mangiare.

La soluzione seguente presenta, invece, il massimo grado di parallelismo per un numero arbitrario di filosofi. Usa un vettore, affamato, per tenere traccia se un filosofo sta mangiando, pensando o sia affamato. Un filosofo può passare nello stato in cui mangia se nessuno dei suoi vicini sta mangiando. Il programma usa un vettore di semafori (uno per ogni filosofo) cosicché i filosofi affamati possano bloccarsi se le loro forchette risultano bloccate.

Una soluzione del tutto equivalente può essere implementata con l'utilizzo di un monitor.

## Problema dei 5 filosofi: soluzione basata sui semafori

```
int N=5; int THINKING=0
int HUNGRY=1; int EATING=2
int state[N]
semaphore mutex=1
semaphore s[N]={0,...,0}
```

```
function philosopher(int i)
  while (true) do
    think()
    take_forks(i)
    eat()
    put_forks(i)
```

```
function left(int i) = i-1 mod N
function right(int i) = i+1 mod N
```

```
function test(int i)
  if state[i]=HUNGRY and state[left(i)]!=EATING and state[right(i)]!=EATING
    state[i]=EATING
    up(s[i])
```

```
function take_forks(int i)
  down(mutex)
  state[i]=HUNGRY
  test(i)
  up(mutex)
  down(s[i])
```

```
function put_forks(int i)
  down(mutex)
  state[i]=THINKING
  test(left(i))
  test(right(i))
  up(mutex)
```

## Problema dei 5 filosofi: soluzione basata sui monitor

```
int N=5; int THINKING=0; int HUNGRY=1; int EATING=2
```

```
monitor dp_monitor  
  int state[N]  
  condition self[N]
```

```
  function take_forks(int i)  
    state[i] = HUNGRY  
    test(i)  
    if state[i] != EATING  
      wait(self[i])
```

```
  function put_forks(int i)  
    state[i] = THINKING;  
    test(left(i));  
    test(right(i));
```

```
  function test(int i)  
    if ( state[left(i)] != EATING and state[i] = HUNGRY  
        and state[right(i)] != EATING )  
      state[i] = EATING  
      signal(self[i])
```

```
function philosopher(int i)  
  while (true) do  
    think()  
    dp_monitor.take_forks(i)  
    eat()  
    dp_monitor.put_forks(i)
```

### 2.12.7 Problema dei Lettori-Scrittori

Oltre il problema dei 5 filosofi, un altro problema è quello della gestione di lettura e scrittura su un Database, conosciuto comunemente come problema dei lettori e scrittore. Come possiamo gestire chi scrive e chi legge su una stessa struttura dati?

La soluzione è fare in modo che quando un lettore ottiene l'accesso, esegue una down sul semaforo DB, i lettori successivi incrementano un contatore RC. Uscendo i lettori decrementano il contatore, l'ultimo eseguirà una UP sul semaforo, consentendo l'accesso ad uno scrittore, se esiste. In questo modo più lettori possono operare in parallelo poiché il loro operato non collide. In sostanza quello che accade è che una volta entrato un lettore, verranno poi soddisfatti tutti i lettori, prima che si possa ripassare agli scrittori. Questa soluzione privilegia i lettori e può portare ad una attesa indefinita per gli scrittori, nel caso in cui arrivassero lettori di continuo. Per evitare che lo scrittore attenda tempo indefinito ad aspettare che indefiniti lettori leggano, basterà mettere in coda i lettori dietro lo scrittore.

Il database sta al di fuori del monitor, metterlo all'interno significherebbe garantire la mutua esclusione sul DB, proprio perché il monitor stesso è esclusivo, e non vogliamo ciò perché renderebbe incompatibili 2 letture.

**\*Prima Soluzione con Monitor:**

Corrisponde alla soluzione iniziale proposta con i semafori, ma basata sui monitor: non entrano scrittori se ci sono dentro lettori.

Questa soluzione è del tutto equivalente a quella vista con i semafori.

**\*Seconda Soluzione con Monitor:**

Aggiungendo OR in\_queue(write) nella start\_read()

Entra prima un blocco di lettori, poi appena arriva uno scrittore vengono smaltiti i lettori già entrati, ed entra lo scrittore. Rispetta quindi la coda di attesa in ordine d'arrivo.

**\*Terza Soluzione con Monitor:**

Invertendo l'if nella procedura end\_write(), si privilegiano gli scrittori in coda, piuttosto che i lettori. Privilegiando la parte degli scrittori si rischia di creare starvation sui lettori, e quindi una coda indefinitamente lunga di lettori. Potrebbe essere accettabile in qualche scenario reale.

## Problema dei lettori e scrittori: soluzione basata sui semafori

- Problema classico che modella l'accesso ad un data-base;

```
function reader()
  while true do
    down(mutex)
    rc = rc+1
    if (rc = 1) down(db)
    up(mutex)
    read_database()
    down(mutex)
    rc = rc-1
    if (rc = 0) up(db)
    up(mutex)
    use_data_read()
```

```
semaphore mutex = 1
semaphore db = 1
int rc = 0
```

```
function writer()
  while true do
    think_up_data()
    down(db)
    write_database()
    up(db)
```



## Problema dei lettori e scrittori: soluzione n.1 basata sui monitor

```
monitor rw_monitor
  int rc = 0; boolean busy_on_write = false
  condition read, write
```

```
  function start_read()
    if (busy_on_write) wait(read)
    rc = rc+1
    signal(read)
```

```
  function end_read()
    rc = rc-1
    if (rc = 0) signal(write)
```

```
  function start_write()
    if (rc > 0 OR busy_on_write) wait(write)
    busy_on_write = true
```

```
  function end_write()
    busy_on_write = false
    if (in_queue(read))
      signal(read)
    else
      signal(write)
```

```
function reader()
  while true do
    rw_monitor.start_read()
    read_database()
    rw_monitor.end_read()
    use_data_read()
```

```
function writer()
  while true do
    think_up_data()
    rw_monitor.start_write()
    write_database()
    rw_monitor.end_write()
```

## Problema dei lettori e scrittori: soluzione n.2 basata sui monitor

```
monitor rw_monitor
  int rc = 0; boolean busy_on_write = false
  condition read, write
```

```
  function start_read()
    if (busy_on_write OR in_queue(write)) wait(read)
    rc = rc+1
    signal(read)
```

```
  function end_read()
    rc = rc-1
    if (rc = 0) signal(write)
```

```
  function start_write()
    if (rc > 0 OR busy_on_write) wait(write)
    busy_on_write = true
```

```
  function end_write()
    busy_on_write = false
    if (in_queue(read))
      signal(read)
    else
      signal(write)
```

```
function reader()
  while true do
    rw_monitor.start_read()
    read_database()
    rw_monitor.end_read()
    use_data_read()
```

```
function writer()
  while true do
    think_up_data()
    rw_monitor.start_write()
    write_database()
    rw_monitor.end_write()
```



## Problema dei lettori e scrittori: soluzione n.3 basata sui monitor

```
monitor rw_monitor
  int rc = 0; boolean busy_on_write = false
  condition read, write

  function start_read()
    if (busy_on_write OR in_queue(write)) wait(read)
    rc = rc+1
    signal(read)

  function end_read()
    rc = rc-1
    if (rc = 0) signal(write)

  function start_write()
    if (rc > 0 OR busy_on_write) wait(write)
    busy_on_write = true

  function end_write()
    busy_on_write = false
    if (in_queue(write)) signal(write)
    else signal(read)
```

```
function reader()
  while true do
    rw_monitor.start_read()
    read_database()
    rw_monitor.end_read()
    use_data_read()

function writer()
  while true do
    think_up_data()
    rw_monitor.start_write()
    write_database()
    rw_monitor.end_write()
```

### 2.13 Scheduling

Il compito dello Scheduling è scegliere il prossimo processo ad essere eseguito, e dedicargli quali e quante risorse (CPU, memoria, dispositivi I/O)

- CPU Burst-> La CPU viene assegnata ad un determinato processo finché questo non decide di cederla oppure finché il SO non la riprende. È il tempo di utilizzo della CPU.

- CPU-bounded -> Processi che usano molta CPU rispetto all'uso di disp. I/O.

- I/O-bounded -> Processi che usano molto i dispositivi di I/O rispetto CPU.

In un sistema reale, abbiamo processi di entrambi i tipi, quindi conviene avere in esecuzione un mix di processi in modo tale che non si abbia uso intensivo di CPU, o uso intensivo dell'I/O, ma un uso moderato di entrambi contemporaneamente.

#### 2.13.1 Quando si Schedula

Quando è necessario schedulare?

- Quando nasce un nuovo processo: in questo caso è necessario sapere quando eseguirlo, subito o dopo il processo padre.

- Quando muore un processo: quale deve essere il prossimo ad essere eseguito?
- Quando il processo in esecuzione si blocca: quando questo è in attesa del I/O, quando è bloccato ad un semaforo, quando riscontra un errore.

### **2.13.2 Algoritmi di Scheduling**

Se il clock della CPU fornisce interruzioni periodiche si deve prendere una decisione di schedulazione ogni interruzione di clock o ogni K interruzioni. Come per i kernel, esistono due tipi di algoritmi di schedulazione:

- NONPREEMPTIVE (senza Prerilascio): L'esecuzione del processo può essere interrotta o dalla CPU, o dal processo stesso che cede il controllo. In questo caso ad ogni interruzione di clock viene ripreso il processo precedente.
- PREEMPTIVE (con Prerilascio): Il processo è lasciato in esecuzione per un certo periodo di tempo ( l'unità di tempo che usa lo schedulatore per regolare l'esecuzione dei processi è il clock della CPU ), dopo questo lo schedulatore deciderà cosa eseguire

#### **2.13.2.1 Dispatcher**

Il Dispatcher è la componente del SO che si occupa del cambio di contesto (CPU – I/O). E' responsabile del salvataggio del processo uscente ed entrante. Componente che si occupa di salvataggio e ripristino dei contesti. Serve a trasferire il file dalla coda dei processi pronti alla CPU.

Nota: nel momento in cui la CPU esegue una procedura di scheduling ed una di dispatching, essa esegue difatti una parte del kernel. Pertanto, sarebbe corretto dire che c'è una sorta di "spreco" (overhead), allo stesso tempo però necessario per quanto riguarda i processi eseguiti in modalità kernel. È però importante notare che se l'algoritmo impiega tanto tempo per effettuare la sua scelta, questo implica uno spreco di tempo notevole. La latenza di dispatch descrive la quantità di tempo impiegata da un sistema per rispondere a una richiesta di avvio delle operazioni di un processo.

### 2.13.2.2 Obiettivi

Ambienti differenti: batch, interattivi e real-time.

- Obiettivi comuni:

- o Equità nell'assegnazione della CPU;
- o Bilanciamento nell'uso delle risorse;

- Obiettivi tipici dei sistemi Batch:

- o Massimizzare il throughput (o produttività);
- o Minimizzare il tempo di turnaround (o tempo di completamento, tra la richiesta e la conclusione);
- o Minimizzare il tempo di attesa (tenere la CPU in costante utilizzo);

- Obiettivi tipici dei sistemi interattivi:

- o Minimizzare il tempo di risposta;

- Obiettivi tipici dei sistemi real-time:

- o Rispetto delle scadenze (evitare la perdita di dati);
- o Prevedibilità.

È normale che ambienti diversi richiedano diversi algoritmi di schedulazione, poiché ciò che lo scheduler deve ottimizzare dipende molto dal tipo di sistema.

### 2.13.3 Scheduling nei Sistemi Batch

Fondamentalmente nei sistemi Batch non è necessaria una pronta risposta dato che non ci sono utenti in attesa al terminale: si eseguono job in cascata, solitamente sempre dello stesso tipo. E' possibile usare sia algoritmi preemptive che non-preemptive e si preferirà un approccio che riduce gli scambi tra processi migliorando le prestazioni.

#### 2.13.3.1 First-Come First-Served (FCFS)

Sicuramente il più semplice tra tutti gli algoritmi, è senza prerilascio, in cui ai processi viene assegnata la CPU nell'ordine in cui l'hanno chiesta fino a quando non la rilasciano. Il problema è che senza prerilascio un processo potrebbe monopolizzare la CPU rallentando l'esecuzione di processi potenzialmente veloci. Scegliamo sempre il processo in testa alla coda dei processi pronti.

## ESERCIZIO

Abbiamo i seguenti processi da elaborare con l'algoritmo FCFS:

Processo	Arrivo	Durata
P1	0	7
P2	2	4
P3	4	1
P4	5	4

P1 arriva al s0 e andrà subito in elaborazione e ci rimarrà fino al complemento in s7. P2 arriva a s2 e attenderà i 5s del rimanente completamento di P1 per andare in esecuzione ed essere completato all's9. Così via per gli altri, dove P3 e P4 attendono entrambi s7 e vengono completati in P3=8, P4=11.

Tempi di attesa: P1=0, P2=5, P3=7, P4=7 (media 4,75)

Tempi di completamento: P1=7, P2=9, P3=8, P4=11 (media 8,75)

### 2.13.3.2 Shortest Job First (SJF)

È un algoritmo senza prerilascio che presuppone la conoscenza dei tempi di esecuzione dei singoli processi. Se in un certo momento un nuovo processo (o più processi) deve essere schedulato, questo non sarà schedulato in relazione ai processi già presenti, ma solo in relazione a quelli che devono essere schedulati nello stesso momento. Eseguendo prima i job più corti abbassiamo il Tempo medio d'attesa e miglioriamo il throughput. E' dimostrato essere l'algoritmo ottimale in quanto a tempo medio d'attesa, ma solo se tutti i processi da eseguire sono in coda a tempo 0.

## ESERCIZIO

Abbiamo i seguenti processi da elaborare con l'algoritmo SJF:

Processo	Arrivo	Durata
P1	0	7
P2	2	4
P3	4	1
P4	5	4

P1 arriva al s0 e andrà subito in elaborazione e ci rimarrà fino al complemento in s7. In quest'ultimo istante sono arrivati tutti gli altri processi, si sceglie quindi quello con durata minore che è P3 e viene eseguito, e così via.

Ordine di elaborazione: P1, P3, P2, P4

Tempi di attesa: P1=0, P2=6, P3=3, P4=7 (media 4)

Tempi di completamento: P1=7, P2=10, P3=8, P4=11 (media 8)

### 2.13.3.3 Shortest Remaining Time Next (SRTN)

È una versione preempitiva dell'algoritmo precedente SJF.

Quando il ciclo di clock termina il processo con meno tempo di esecuzione rimanente viene eseguito, questo permette a nuovi processi di entrare in gioco se hanno tempi di esecuzioni minori rispetto a quello del processo in esecuzione. Sostanzialmente si mette a schedulare il processo con durata residua più breve. Il concetto di durata residua nasce dal fatto che questo algoritmo fa intervenire la prelazione all'arrivo di nuovi processi in coda. Raggiunge l'ottimo anche quando non ci sono tutti i processi in coda a tempo 0.

## ESERCIZIO

Supponiamo di avere i seguenti cinque processi con relative durate e tempi di arrivo:

processi	P1	P2	P3	P4	P5
durata	7	5	2	4	4
tempo di arrivo	0	3	4	6	9

Qual è il processo che sarà schedato per ultimo?

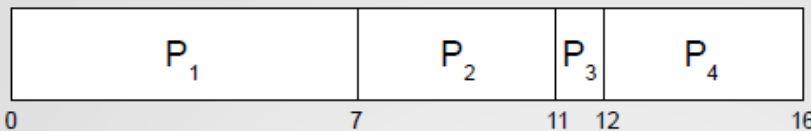
Abbiamo detto che l'SRTN esegue prima i processi più veloci; essendo preemptive, nel momento in cui sta elaborando un processo ne arriva uno più veloce, sospende quello in corso e passa al più veloce. Quindi lo schema di esecuzione sarà (in grassetto, il processo a cui passa l'esecuzione):

t=0	t=3	t=4	t=6	t=9
P1=7	P1=4, P2=5	P1=3, P2=5, <b>P3=2</b>	P1=3, P2=5, P3=0, P4=4	P1=0, P2=5, P3=0, <b>P4=4</b> , P5=4

t=13	t=17	t=22
P1=0, P2=5, P3=0, P4=0, <b>P5=4</b>	P1=0, <b>P2=5</b> , P3=0, P4=0, P5=0	P1=0, P2=0, P3=0, P4=0, P5=0

Il processo schedato per ultimo sarà P2.

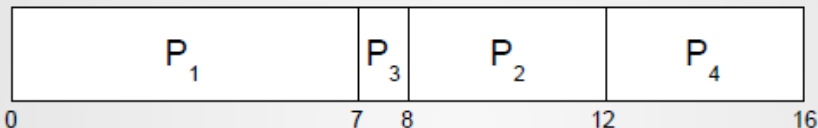
### FCFS:



- tempi di attesa:  $P_1=0$ ;  $P_2=5$ ;  $P_3=7$ ;  $P_4=7$  (media 4.75);
- tempi di completamento:  $P_1=7$ ;  $P_2=9$ ;  $P_3=8$ ;  $P_4=11$  (media 8.75);

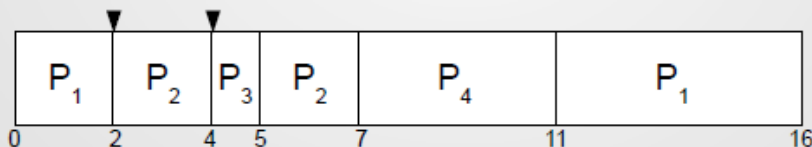
Processo	Arrivo	Durata
P <sub>1</sub>	0	7
P <sub>2</sub>	2	4
P <sub>3</sub>	4	1
P <sub>4</sub>	5	4

### SJF:



- tempi di attesa:  $P_1=0$ ;  $P_2=6$ ;  $P_3=3$ ;  $P_4=7$  (media 4);
- tempi di completamento:  $P_1=7$ ;  $P_2=10$ ;  $P_3=4$ ;  $P_4=11$  (media 8);

### SRTN:



- tempi di attesa:  $P_1=9$ ;  $P_2=1$ ;  $P_3=0$ ;  $P_4=2$  (media 3);
- tempi di completamento:  $P_1=16$ ;  $P_2=5$ ;  $P_3=1$ ;  $P_4=6$  (media 7).

## 2.13.4 Scheduling nei Sistemi Interattivi

In un ambiente Interattivo il prerilascio è essenziale per evitare che un processo si impossessi della CPU impedendo l'accesso agli altri. Bisogna privilegiare la reattività nell'interazione con l'utente. Minimizzare il tempo

di risposta, ovvero il tempo che passa tra il completamento di un I/O e l'istante in cui la CPU viene riassegnata al processo che si riprende dalla chiamata bloccante dell'evento I/O.

#### **2.13.4.1 Scheduling RoundRobin (RR)**

È uno degli algoritmi più vecchi, semplici, imparziali e facili di implementare, è una versione con prelazione del FCFS. Ad ogni processo è assegnato un tempo di esecuzione detto quanto di tempo.

Se al termine del suo quanto il processo è ancora in esecuzione la CPU viene rilasciata e riassegnata, se il processo entra in blocco prima della fine del suo quanto accade lo stesso.

Il RR è semplice da implementare:

c'è una lista di processi in coda, appena un processo termina il suo quanto viene sospeso e spostato alla fine della coda.

L'unica variabile da settare è quindi il quanto (timeslice): se lo setto troppo breve (1 ms) i tempi di esecuzione saranno soverchiati dai tempi di cambio del contesto (cambio registri, aggiornamento liste e tabelle, ricarica della cache, ecc..), se lo setto troppo lungo (100 ms) sarebbe come se i processi monopolizzassero la CPU. Un buon compromesso di durata del quanto è tra i 20 e i 50 ms.

La prelazione è implementata con un interrupt clock. Non avrebbe senso far scattare la prelazione ad ogni interrupt clock, infatti il quanto di tempo  $q$  è multiplo del tempo che intercorre tra due interrupt clock, in modo che ogni  $n$  interrupt clock, uno cada su  $q$ , prelazionando il processo in esecuzione.

Vi è scheduling equo, supponendo di avere  $n$  processi, ogni processo avrà diritto ad una parte pari ad  $1/n$  della CPU. Con  $n$  processi in coda, e un quanto di tempo  $q$ , un processo può attendere al massimo  $n-1(q)$  ms. Infatti sono  $n-1$  i processi che ci sono al massimo davanti a lui, ed ognuno verrà eseguito per  $q$  tempo.

## ESERCIZIO

Abbiamo i seguenti processi in coda con relativa durata in millisecondi, e quanto di tempo stabilito di 20 ms:

Processi	P1	P2	P3	P4
Durata (millisecondi)	30	15	60	45

Verranno eseguiti nel seguente ordine:

- P1 (interrotto dopo 20 ms, ne rimangono altri 10)
- P2 (termina la propria esecuzione perché dura meno di 20 ms)
- P3 (interrotto dopo 20 ms, ne rimangono altri 40)
- P4 (interrotto dopo 20 ms, ne rimangono altri 25)
- P1 (termina la propria esecuzione perché necessitava di meno di 20 ms)
- P3 (interrotto dopo 20 ms, ne rimangono altri 20)
- P4 (interrotto dopo 20 ms, ne rimangono altri 5)
- P3 (termina la propria esecuzione perché necessitava di esattamente 20 ms)
- P4 (termina la propria esecuzione)

## ESERCIZIO

Supponiamo di utilizzare un algoritmo di scheduling preemptive come il Round-Robin: assumendo di avere un context switch effettuato in 5 ms e di usare quanti di tempo lunghi 50 ms, a quanto ammonta la percentuale di overhead per la gestione dell'interlacciamento dei processi?

$$\frac{5}{50 + 5} = 0,09 \cdot 100 = 9\%$$

### 2.13.4.2 Scheduling a Priorità

Esistono processi con priorità differente, questa dipende da vari fattori: caratteristiche, modalità di utilizzo delle risorse (CPU bounded meno priorità, I/O bounded più priorità).

La priorità può essere considerata come un numero (0 bassa priorità – 100 alta priorità per esempio), e può essere statica o dinamica:

- o Statica: Nasce e muore col processo.
- o Dinamica: Dipende da altri fattori (ereditarietà, determinati momenti della vita del processo, altri processi in funzione, ecc..)

Per evitare che possa subentrare una Starvation dei processi minori si usa la priorità dinamica (con Starvation si intende quando un processo non riceve mai il controllo della CPU a causa della sua bassa priorità e diventa “affamato”).

#### 2.13.4.2.1 Priorità

Potremmo considerare una priorità dinamica legata al quanto di tempo utilizzato precedentemente dal processo: se il processo ha sfruttato al massimo il tempo e le risorse la priorità non varierà, altrimenti se ne sfrutta meno la priorità si abbasserà.

Altro modo è dare più priorità a task più veloci da completare. In un sistema basato su prelazione potrebbe essere che il processo perda o acquisti priorità in base se si aggiungono o vengono a mancare i presupposti di esecuzione. Se si considerasse come priorità  $1/t$ , dove  $t$  è il tempo di durata del processo, si otterrebbe esattamente l'algoritmo di scheduling SJF. Possiamo vedere SJF come uno scheduling per priorità, dove la priorità maggiore è assegnata al job più breve.

Nei sistemi interattivi si usano solitamente priorità dinamiche, e si intende favorire i processi I/O-bounded, e svantaggiare quelli CPU-Bounded. Per favorire i processi I/O-Bounded dobbiamo riuscire a distinguerli.

Valutiamo come il processo usa il suo quanto di tempo:

- Processi I/O-Bounded si bloccano spesso prima di essere prelazionati.
- Processi CPU-Bounded sfruttano tutto il loro quanto di tempo.

Valutando una serie di "quanti test" possiamo quindi stimare il comportamento transitorio del processo, ed assegnare priorità maggiore a quelli I/O-Bounded.

In tal caso i processi CPU-Bounded verranno schedulati solo in mancanza di processi I/O-Bounded pronti, e questo potrebbe causare Starvation.

I processi CPU-Bounded potrebbero attendere all'infinito se si presentassero sempre processi I/O-Bounded pronti.

Soluzione: Euristiche basate sull'invecchiamento (Aging)

Aumentare nel tempo la priorità dei processi che non usano la CPU: prima o poi la priorità del processo che attende da tanto supererà le altre, e sarà eseguito. Ovvero più la CPU viene usata da parte di un processo HP (High Priority), più la sua priorità scenderà man mano, quando un processo a bassa priorità in coda supera il processo in esecuzione, si effettua uno switch e processi con meno priorità potranno entrare in esecuzione.

Esiste una variante detta scheduling a code multiple (classi di priorità).

Consiste nell'avere "n code" rappresentative di una determinata priorità (esempio: coda 1=processi con priorità 1, coda 2=processi con priorità 2,...).

Composto da priorità fisse (in alto: processi I/O bounded, in basso: processi



CPU bounded) e con feedback (o retroazione).

È possibile partizionare la coda dei processi pronti in base alle loro priorità, suddividendo dinamicamente: nelle classi più alte ci sono i programmi interattivi, nelle classi più basse ci sono servizi minori. Nelle classi più alta si imposta un RR breve, nelle classi più basse si può usare un FCFS.

Una buona implementazione può essere fatta così:

Round robin in ogni coda, variando la durata del quanto di tempo.

Più è alta la priorità, meno dura il quanto di tempo. Viceversa, più è bassa la priorità, più dura il quanto di tempo.

Nell'ultima fascia abbiamo praticamente un FCFS(quantità di tempo infiniti).

I processi a bassa priorità(CPU-Bounded) saranno serviti con un FCFS.

Non usando aging torna il problema della starvation.

Può essere risolto garantendo ad ogni priorità una percentuale di utilizzo della CPU, dato un intervallo di tempo: più bassa è la priorità, minore è la percentuale concessa. Possiamo attuare due politiche per le classi di priorità: un processo rimane nella sua coda di priorità, dalla creazione alla terminazione. Oppure, in base a come si comporta un processo transitoriamente, posso declassarlo o promuoverlo ad una classe più alta. Queste operazioni vengono fatte in concomitanza con lo scheduling.

### 2.13.4.3 Shortest Process Next (SPN)

Noi non possiamo accedere alla lunghezza del task, ma possiamo confrontare la lunghezza del CPU burst cercando prevedere statisticamente la lunghezza del prossimo CPU burst (che dovrebbe essere simile alla lunghezza di quello precedente), possibilmente schedulando meglio la lista dei processi. Ancora meglio è calcolare una media esponenziale che aggiorna la stima del prossimo CPU burst basandosi su tutti i precedenti valori. " $S_{n+1} = S_n(1 - a) + T_n a$ "

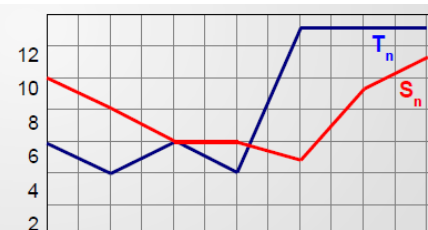
$S_n$  -> Media ponderata delle Stime precedenti;

$T_n$  -> Durata effettiva dell'ultimo CPU burst;

$a$  -> Valore arbitrario compreso tra 0 e 1, più è alto meglio si adatterà all'ultima stima ma oscillerà di più;

• esempio:  $a=1/2$

$T_n$	6	4	6	4	13	13	13
$S_n$	10	8	6	6	5	9	11



#### **2.13.4.4 Scheduling Garantito**

Garantisce un uso equo della CPU stabilendo una percentuale di utilizzo. Avendo N processi, assegno ad ogni processo  $1/N$  tempo di esecuzione.

#### **2.13.4.5 Scheduling a Lotteria**

Concretizza e semplifica l'idea precedente, crea periodicamente una lotteria e distribuisce un tot di ticket in base alla priorità che verranno estratti ciclicamente, la CPU verrà usata da un ticket alla volta, e all'estrazione il ticket viene rimosso. Rende possibile l'utilizzo della CPU da parte di tutti, si possono avere processi cooperanti che si scambiano i ticket, dando priorità ad uno tra essi.

#### **2.13.4.6 Scheduling Fair-Share**

Ripartisce equamente la CPU tra tutti i gruppi di processi (utenti) del sistema. Utenti che lanciano più processi rispetto ad altri avranno la stessa CPU dedicata, indipendentemente dal numero di processi.

#### **2.13.5 Sistemi Real-Time**

Rispetto delle scadenze: poca sicurezza e sistema semplice.

Scheduling semplice e overHead quasi nullo. Responsabilità delegate ai programmatori, che scrivono programmi consapevoli di dover collaborare con altri programmi.

#### **2.13.6 Scheduling dei Thread**

- \* Thread Utente:

- o Ignorati dallo scheduler del kernel;

- o Per lo scheduler del sistema run-time vanno bene tutti gli algoritmi non-preemptive visti;

- o Possibilità di utilizzo di scheduling personalizzato.

- \* Thread Kernel:

- o O si considerano tutti i thread uguali, oppure

- o Si pesa l'appartenenza al processo (priorità ai thread fratelli);

- o Lo switch su un thread di un processo diverso implica la riprogrammazione della MMU.

Nota: infatti, generalmente, è opportuno che se si sta utilizzando un

processo contenuto all'interno di un Thread, si continui con lavorare con altri processi sullo stesso Thread. Questo perché si va a risparmiare un'operazione di rimappatura della memoria, si evita di "saltare" da un thread all'altro. I thread fratelli condividono una parte dello spazio di indirizzamento, quindi non serve effettuare context-switch.

### **2.13.7 Scheduling su Sistemi Multicore**

In questi sistemi abbiamo più core a disposizione e possiamo sfruttarli contemporaneamente, possiamo attuare diverse strategie:

- Multielaborazione asimmetrica:

In questo modello abbiamo un CORE MASTER (SERVER) che gestisce le strutture dati interne, i dispositivi di I/O, ecc. Gli altri core sono SLAVE, che eseguono i compiti del SERVER.

- Multielaborazione simmetrica (Symmetric Multi Processing "SMP"):

Tutti i core sono uguali e si ripartisce il carico equamente sui processori. In questo caso subentra la gestione delle Race Condition dato che le strutture dati sono condivise. C'è un'unica coda dei processi pronti (l'accesso alla coda deve essere effettuato in maniera oculata dato che più core potrebbero voler pescare un processo contemporaneamente, inoltre potrebbe succedere che un core debba aspettare altri processi e si iberni) oppure code separate per ogni core (più usato ma con il problema del bilanciamento del carico).

#### **2.13.7.1 Affinità**

Con più core ho più cache private, se un processo viene schedulato in momenti diversi in core diversi, questi non possono accedere alla cache di altri core (tipo cache miss), perdendo tempo e lavoro per ripopolare la propria cache. Detto ciò è preferibile rischedulare un processo sullo stesso core, stabilendo un'affinità del processo con un determinato core.

- Affinità debole: Il SO non conta molto l'affinità processo-core, è possibile violarla.

- Affinità forte: Il SO prende in serio conto l'affinità processo-core, non la viola. Sorge il problema del bilanciamento del carico: nel caso le code siano separate e in un core ci sia una lunga coda di processi con affinità forte il carico è fortemente sbilanciato.

### **2.13.7.2 Migrazione**

Per questo motivo esistono dei meccanismi di migrazione di processi da una coda all'altra.

- Migrazione guidata (push): Esiste un servizio del SO che verifica il bilanciamento del carico, sposta i processi nel caso di uno sbilanciamento.
- Migrazione spontanea (pull): Il core stesso esamina le varie code e sottrae i processi ad una coda più carica.

In genere si implementano entrambe le strategie contemporaneamente poiché solo una delle due non riesce a gestire tutti i problemi.

## **2.14 Scheduling su Windows 8**

Il sistema Windows è abbastanza semplice è Preemptive basato su classi di priorità. Api Win32 per la gestione delle priorità dei thread:

- SetPriorityClass: chiamata di sistema che permette di assegnare una classe di priorità ad un processo (contenitore di thread).
- SetThreadPriority: Permette ad un thread di assegnare a se stesso o ad un thread fratello una priorità relativa all'interno del processo, creando una gerarchia di priorità all'interno del processo. La priorità relativa è temporanea e può solo incrementare rispetto alla priorità base.

La priorità è compresa da 0 a 31, l'algoritmo di selezione del processo da eseguire prima identifica la classe non vuota di priorità più alta e poi il processo candidato con maggiore priorità secondo un Round-Robin interno alla classe. Ovvero mappatura sulla priorità di base (numerica), disposte in classi: a numeri più alti corrispondono priorità più alte. Nella tabella abbiamo: nelle Colonne ci sono le priorità per i processi e nelle righe quelle per i Thread all'interno di un dato processo.

### **2.14.1 Motivi Priorità Relativa Modificata**

I motivi per i quali la priorità relativa viene modificata sono:

- 1) L'utilizzo del quanto del tempo: se questo viene usato parzialmente o il thread si blocca.
- 2) Ripresa da un I/O: si blocca in attesa di un I/O, si ha un salto di +3 di priorità relativa.
- 3) Processi in primo piano: se la GUI è in primo piano rispetto ad altri oltre ad avere più priorità, ha un quanto di tempo aumentato.

### 2.14.2 Inversione Priorità e AutoBoost

Gli ultimi thread sono fittizi, lo “zero page thread” (thread a priorità zero) è una routine periodica che entra in gioco quando il processore è libero. A causa della scelta netta della schedulazione di determinati processi potrebbe dar luogo ad un inversione di priorità: se un processo di HP è bloccato da un processo a LP che non è in esecuzione perché è troppo LP, quest'ultimo subisce un Auto Boost temporaneo che ne incrementa la priorità sbloccando il primo processo.

## 2.15 Scheduling su Linux

Tutto è schedulato per “task”, che può essere o un processo o un thread. È in realtà un agglomerato di flussi che condividono qualcosa.

### 2.15.1 Chiamata di Sistema Clone

Chiamata di sistema clone: duplica il flusso in esecuzione, questo può o meno condividere cose col padre, ha diversi flag che le donano granularità e potenzialità.

VM -> condivide la memoria se impostato ad 1.

FS -> avranno working directory condivisa se impostato ad 1.

FILES -> files condivisi se impostato ad 1.

PARENT -> si eredita l'ID del genitore se impostato ad 1.

Fork -> Duplica fedelmente il padre condividendone i flussi, è il caso in cui tutti i flag sono settati ad 1.

Linux gestisce i thread come task, posso identificarli tramite degli ID:

- Process identifier (PID): per retrocompatibilità.
- Task identifier (TID): univoco per ogni task.

Thread distinti ma dello stesso processo hanno TID diversi ma PID uguali, un processo clonato ha sia PID che TID diverso dal padre

### 2.15.2 Scheduler O(1) di Linux

È stato abbandonato perché era difficilmente scalabile e le sue prestazioni non erano all'altezza dei processori multicore.

Usa 3 classi di priorità:

- Real-time FIFO (non preemptive): lascia ai processi il controllo.
- Real-time round-robin (preemptive): tenta la mediazione tra i due

sistemi, è una real-time con un quanto di tempo.

- Timesharing (preemptive): è la classe usata.

Usa delle priorità numeriche:

- Real-time da 0 (più alta) a 99;
- Timesharing da 100 a 139 (più bassa);

Che possono variare in modo:

- Statico (nice value: -20 a +19) l'utente può solo diminuire la propria priorità (aumentando il numero) dando più spazio agli altri, gli amministratori possono alzarla e diminuirla.
- Dinamico (da -5 per I/O boundness a +5 per CPU boundness) ai processi I/O boundness, che hanno quanti più piccoli rispetto ai CPU boundness, viene data più priorità.

Per ogni CPU si ha una RUNQUEUE, questa gestisce una serie di code, 140 code attive (active) e 140 code scadute (expired).

L'algoritmo di scheduling va tra le code attive e prende la coda con priorità più alta selezionandone il primo processo e eseguendolo fino all'esaurimento del suo quanto di tempo. Nel caso succeda qualcosa e non riesca a completare il suo quanto di tempo, al riavvio il quanto rimanente viene ripristinato (non succede su WIN). Nel caso esaurisca il quanto viene spostato nella coda expired. Una volta consumati i processi nella coda attiva, si swappano i due array e le code expired diventano code active rinnovando i proprio quanti di tempo.

I quanti oscillano tra i 5ms (corti) ai 800ms (lunghi)

Le classi ad alta priorità avranno quanti più lunghi, le classi a bassa priorità ce li avranno più corti (si parla sempre di processi in Timesharing).

La complessità è costante perché non dipende dagli elementi e non bisogna effettuare ricerche.

È una runqueue specifica per ogni CORE, è prevista la migrazione spontanea (dal processo) o guidata (attraverso demone) da CORE a CORE. È necessario proteggere la struttura dati attraverso uno SpinLock, c'è quindi il busywaiting, ma è necessario per la consultazione della struttura dati.

### 2.15.3 Scheduler CFS di Linux

Questo scheduler è stato criticato per la gestione delle code, assegnazione e gestione delle priorità e fumosità. Un tentativo di avere un progetto di qualità superiore ha portato allo scheduler CFS (Completely Fair Scheduler). Questo cerca di dare risorse equamente, tenendo traccia delle risorse usate per un processo e delle risorse che gli spetterebbero. Usa il Virtual Runtime (VRT), cioè il conteggio della quantità di tempo in cui il processo ha usato le risorse, tutti i processi debbono poter utilizzare la CPU con la stessa quantità di tempo, di volta in volta si esegue quindi il processo con virtual runtime più basso.

La struttura dati usata è un albero di ricerca rosso-nero (autobilanciante; inserimenti, ricerche, cancellazioni con complessità logaritmica).

L'elemento minimo starà nel PCB in basso a SX.

È uno scheduler con prelazione.

Man mano che il processo utilizza la CPU il suo VRT cresce e se non è più il minimo si cambia il posto e si switcha il processo in esecuzione. C'è un tempo minimo di esecuzione e una granularità del controllo configurabile. Viene costantemente mantenuto il VRT più basso, periodicamente viene fatta una normalizzazione di tutte le etichette, sottraendo il minimo a tutti, evitando di far crescere le variabili in modo incontrollato. Quando entra un nuovo thread questo avrà un VRT uguale al minimo o al valore medio (dipende dall'implementazione).

Il sistema dà la possibilità ai processi che si bloccano spesso di avere un'alta priorità (quando si interrompe il suo VRT resterà inalterato, in modo naturale si troverà vicino alla parte SX dell'albero, se non proprio sarà il prossimo processo in esecuzione).

Le priorità sono applicate come fattori di decadimento, sono delle costanti moltiplicative che influiscono il VRT. I processi a bassa priorità li avranno più alti cosicché il VRT cresca più rapidamente lasciando prima la CPU agli altri. Nella versione 2.6.24 è stato introdotto lo scheduling di gruppo, il fair share. Il sistema ha la possibilità di conteggiare e fare scelte di scheduling per gruppi di processi. Nei sistemi multicore è tutto molto in linea con ciò che abbiamo visto, ogni CORE ha il suo albero e la migrazione può essere spontanea o guidata.

## 3. Gestione della Memoria

### 3.1 Gerarchia di Memoria

L'ideale per un PC sarebbe avere una memoria veloce, non volatile, spaziosa e non costosa. Tutto ciò però non esiste e non è fattibile, quindi le macchine odierne si appoggiano a tre tipi di memoria

- 1) Una piccola quantità di memoria molto veloce e molto costosa (CACHE).
- 2) Gigabyte di memoria principale volatile a velocità media e prezzo medio (RAM).
- 3) Terabyte di disco di memorizzazione non volatile, lento ed economico (HD).

La parte del SO che gestisce l'interazione tra queste memorie è il gestore della memoria, il suo compito è tener conto della memoria utilizzata e libera, allocare memoria quando un processo lo richiede, deallocarla quando non è più utile. La RAM è il livello più basso direttamente utilizzabile dalla CPU.

### 3.2 Sistemi Gestione Memoria Centrale (RAM)

Esistono due sistemi della gestione della memoria:

- 1) Sistemi che non effettuano swapping, più semplici.
- 2) Sistemi che effettuano Swapping o paginazione, cioè che spostano i processi avanti e indietro dalla memoria centrale alla memoria di massa.

### 3.2 Sistemi Senza Swapping e Senza Paginazione

All'inizio dello sviluppo dei primi sistemi operativi l'astrazione era un concetto per lo più praticamente inesistente. Ci si "accontentava" di ciò che le prestazioni hardware offrivano, senza badare troppo a sistemazioni software. Il modello senza astrazione è pertanto un modello utilizzato sui primi mainframe (anni '60) e sui primi PC (primi anni '80). Erano per lo più computer con 1 sola CPU e memoria fisica nell'ordine di qualche kb.

I programmi utilizzano direttamente indirizzi fisici, sono eseguiti uno alla volta e solo al termine del primo può entrare in esecuzione il successivo.

Non vi è astrazione della memoria a favore dei processi.

Il sistema può essere organizzato in tre modi differenti:



- 1) Il SO sta nella parte bassa della RAM.
- 2) Il SO sta nella ROM (Read Only Memory).
- 3) Il SO sta nella RAM e il Gestore dei dispositivi (drivers) nella ROM.

Nel caso di Multiprogrammazione in questi sistemi si presenta il problema di far coesistere più processi in esecuzione concorrente: mentre un processo è in attesa di un'operazione di I/O, un altro che richiede la CPU può essere eseguito. In questo caso suddividiamo la memoria in  $n$  partizioni, ognuna con grandezza diversa: processi piccoli saranno assegnati a partizioni piccole e processi che richiedono più memoria saranno assegnati a partizioni più capienti. Ci sarà una coda di processi che attendono che gli sia assegnata una partizione, nel caso di numerosi processi piccoli però non riusciremo a sfruttare le aree di memoria riservate ai processi più pesanti. Per risolvere questo problema potremmo usare un contatore interno al processo che indica quante volte è stato scartato perché la memoria richiesta da lui era troppo piccola rispetto alle aree disponibili, appena si raggiunge un valore  $K$ , il processo dovrà entrare in un'area di memoria (la più piccola possibile) ed essere eseguito.

### **3.2.1 Multiprogrammazione Senza Astrazione**

Cosa succede se devo avviare un processo? Dentro il processo ci sono allocazioni di memoria che rimandano ad altre del processo stesso (es una variabile  $X$  rimanda all'indirizzo di memoria 22 del processo), ma appena devo eseguirlo e lo sposto in memoria centrale, gli indirizzi non corrisponderanno più (se sposto il processo all'indirizzo 10000, la variabile  $X$  dovrebbe puntare al 10022, ma in realtà punta a 22 che è un'area esterna al processo!). Questo può comportare gravi problemi sia al processo che al resto del sistema (ad esempio incrementare, decrementare, cancellare una variabile potrebbe distruggere il SO). La soluzione è sommare a tutti i rimandi a variabili l'indirizzo MIN in cui il processo è stato allocato ( $22+10000=10022$  OK!).

### **3.2.2 Rilocalizzazione Compile-time e Statica**

Il compilatore, in fase di creazione del codice, deve decidere gli indirizzi e, in alcuni casi, questi indirizzi non posso averli o stabilire a priori su dove e quante volte dovrò eseguire la mia applicazione.

\*Rilocazione a compile-time:

Il compilatore dovrebbe conoscere il range di indirizzi su cui lavorerà il codice una volta istanziato (deve sapere che indirizzi saranno dati al processo), in modo da fargli generare gli indirizzi corretti in base alla sezione di RAM assegnata. E' una gestione molto scomoda, quasi infattibile.

\*Rilocazione statica in fase di caricamento, con rallentamento del loader:

Ovvero, il compilatore funziona in modo basilare, usa gli indirizzi partendo dal primo utile (0). Il Loader attua la rilocazione, ovvero traduce gli indirizzi sapendo che zona di memoria è stata assegnata all'istanza di quel programma(processo), in fase di avvio del processo. Questo viene fatto attuando uno shift degli indirizzi con un offset. Richiede un appesantimento nella fase di Loading.

### **3.2.3 Protezione della Memoria Lock&Key**

Blocchi di memoria con delle chiavi di protezione e PSW con la chiave del processo in esecuzione.

Sapendo che i primi sistemi avevano una memoria fisica dell'ordine di qualche kb (da 16 a 64), l'idea era quella di suddividere questa memoria in blocchi di dimensioni fisse (2kb). Per ogni blocco veniva associata una chiave (combinazione di un numero a 4 bit). L'idea era quella che, nel momento in cui un processo richiedeva la CPU, se la chiave della PSW era uguale a quella del blocco della CPU allora il processo aveva accesso, altrimenti no.

### **3.2.4 Spazio degli Indirizzi e Compilazione Dinamica**

Dal punto di vista della protezione ciò non è abbastanza, il processo può facilmente riuscire ad arrivare fuori dalla memoria a lui assegnata (basta che punti già inizialmente ad un indirizzo fuori dal suo range), per ovviare a questo problema si definisce un limite superiore oltre il quale il processo non può accedere (che corrisponde al MAX del processo, ovvero il Registro Limite), se il processo tenta di accedervi viene bloccato, se invece è dentro il range, viene sommato il valore del Registro BASE ed il valore sarà rimesso in range. E' una rilocazione praticamente uguale a quella di prima, ma che avviene a tempo di esecuzione ("rilocazione a runtime") ed è svolta dall'MMU (Memory Management Unit). Questa tecnica permette di

risolvere il problema della rilocalizzazione “standard”. Vengono definiti così degli indirizzi logici letti dalla CPU. I registri in ogni caso vanno aggiornati ad ogni context-switch.

### **3.3 Swapping**

Con i sistemi Batch organizzare la memoria in partizioni fisse è semplice ed efficiente, ma nei sistemi Timesharing (i più diffusi) non è disponibile abbastanza RAM per mantenere tutti i processi attivi, quindi si ricorre ad una tecnica di Swapping (svolto dallo Swapper, uno scheduler di medio termine), che consiste nel caricare interamente un processo dal disco, eseguirlo per un certo periodo di tempo e rispostarlo nuovamente sul disco (sia nel caso che sia terminato o meno), evitando di buttare via il suo stato, salvando il suo spazio di indirizzamento e liberando RAM .

Un'altra strategia, detta Memoria Virtuale, rende possibile caricare solo una parte del processo sulla RAM e eseguirlo ugualmente (la vedremo più avanti). La forza dello Swapping sta nell'allocazione dinamica: non essendoci aree prestabilite di memoria, posso riempire la RAM con più processi possibili, siano essi tutti piccoli, tutti grandi o misti. L'unico problema da tenere in mente è che potrebbe succedere che a processi già in memoria potrebbe servire ulteriore memoria nel corso dell'esecuzione: per evitare spostamenti di processi già allocati in altre aree di memoria è bene allocare una quantità extra di memoria per ogni processo, prevedendo così ulteriori richieste. Inoltre è preferibile fare in modo che le aree di memoria disponibile siano contigue, così da evitare frammentazione.

#### **3.3.1 Problema Richieste I/O Pendenti**

Supponiamo che un processo P lancia una richiesta I/O, e prima che venga completata viene spostato in HD. Magari viene inserito un nuovo processo nello spazio di indirizzamento del vecchio processo P, e quando il DMA otterrà i dati, si occuperà di scriverli nell'indirizzo di memoria fornitogli in fase di richiesta. Si crea quindi una inconsistenza, in quanto il DMA senza poter sapere nulla del cambio di processo, scrive dati in un nuovo processo che non aveva lanciato richieste I/O. Quindi il sistema dovrebbe annullare la richiesta I/O, prima di schedulare a medio termine un processo.

### **3.3.2 Strategie di Allocazione e Frammentazione**

- dimensione fissa: molto limitativa e poco utilizzata.
- dimensione dinamica: molto più utilizzata attualmente.
- f. interna: fenomeno di spreco, dovuto ad allocazioni che faccio ad un processo, stanziando memoria di dimensione fissa. Ogni processo occupa lo stesso spazio di RAM. In prima analisi non è una soluzione buona perché potrebbe non bastare lo spazio ad un processo mentre magari ne stiamo sprecando per un altro.
- f. esterna: problema legato all'allocazione contigua. E' necessario fare delle previsioni sullo spazio che verrà usato dal processo relativo ad un programma, e non è semplice né preciso. Inoltre in questo caso si viene a creare Frammentazione. Il problema consiste nell'avere delle partizioni piene e vuote, che causano una miriade di "buchi" che sostanzialmente sprecano solo memoria.

### **3.3.3 Memory Compaction Problema**

Le previsioni potrebbero essere errate o imprecise, ed in tal caso ci toccherà trovare spazio contiguo da aggiungere allo spazio di indirizzamento concesso al processo. Inoltre potremmo non riuscire ad allocare più processi, pur avendo tanto spazio libero in RAM, ma diviso in piccoli buchi.

La Memory Compaction è una sorta di deframmentazione applicata alla memoria centrale. Consiste nello spostare opportunamente le locazioni di memoria contigue tra di loro, in modo tale da "compattare" ("accorpate") gli spazi vuoti non utilizzati. Viene usata poco perché incide molto negativamente sulle prestazioni del sistema.

### **3.4 Gestione dell'Allocazione Contigua**

Con un assegnazione dinamica della memoria è necessaria una gestione adeguata dal parte del SO, in particolare è necessario sapere quali aree di memoria sono allocate e quali sono libere, visti i pericoli di frammentazione interna quando allochiamo la memoria per Blocchi. Infatti, in seguito alla richiesta di un certo numero di byte di un processo, si eseguirà un arrotondamento per eccesso al multiplo più vicino, in relazione alla

dimensione minima del blocco decisa. Ci può essere uno spreco, che nel caso peggiore coincide con quando la richiesta sfiora di un solo byte rispetto ad un multiplo della dimensione minima, costringendo il sistema ad arrotondare al multiplo successivo.

E' possibile gestire l'allocazione in due modi:

- Mappe di bit (Bitmap):

La memoria è divisa in unità di allocazioni, che possono essere di varia grandezza in base alle scelte progettuali: più è piccola l'allocazione, più grande sarà la Bitmap (mappa di bit); più è grande l'allocazione più saranno presenti sprechi di memoria dato che non allocherò la memoria in modo preciso. Una volta stabilita la grandezza dell'unità di allocazione la nostra Bitmap avrà una sua grandezza fissa, infatti la dimensione della Bitmap dipende solo dalla grandezza della memoria fisica e dalla dimensione dell'unità di allocazione. Quando si deve caricare un processo di K unità, si deve scansionarla e trovare una sequenza di almeno K unità di memoria liberi nella Bitmap (rappresentati come 0). Scansionare la Bitmap è meno dispendioso di scansionare l'intera RAM.

Il problema principale di questo metodo è quando la memoria necessaria è a cavallo tra due parole e questo rallenta molto il processo.

- Lista concatenata dei blocchi:

Ogni nodo della lista, corrisponde ad una sequenza di blocchi (Unità di allocazione).

\* Implementazione con lista dei blocchi liberi e occupati:

Potremmo usare un'unica lista doppiamente linkata sia per i blocchi liberi che per i blocchi occupati, e tenerla in ordine di indirizzo. Questo ci permette di fare la coalescenza in tempo costante dei blocchi liberi contigui, unificandoli in un unico nodo della lista. Ogni nodo della lista contiene l'indice del blocco iniziale, ed il numero di blocchi contigui. Per questo più la RAM ha zone libere contigue, più faccio coalescenza, più risparmio spazio in lista.

\* Implementazione con singola lista dei blocchi liberi:

Si potrebbe anche implementare una sola lista dei blocchi liberi, che potrebbe anche semplificare l'istanziamento di processi, ma la lista sarebbe

più difficile da mantenere. Infatti non potremmo però fare coalescenza così facilmente: potrebbero esserci blocchi occupati tra due blocchi liberi. E' utile ordinare la lista per indirizzi di memoria così da facilitarne l'aggiornamento quando un processo termina o si arresta.

Per allocare la memoria ad un certo processo possiamo usare diversi metodi:

- o First Fit: viene considerato il primo posto abbastanza capiente disponibile, è molto veloce, ma potrebbe comportare sprechi di memoria.
- o Next Fit: identico al First Fit, si ricorda della prima allocazione libera trovata. Quando sarà necessario allocare memoria per un nuovo processo riprenderà la ricerca da lì e non dall' inizio della lista. Sorprendentemente poco più lento del first fit.
- o Best Fit: Cerca in tutta la lista l'allocazione più adatta, cioè quella che offre la memoria più piccola necessaria. È più lento del first fit, e sorprendentemente spreca più memoria: il best fit lascia libere piccole porzioni di memoria inutilizzabili.
- o Quick Fit: si mantengono due liste, quella dei processi allocati e quella delle aree di memoria libera ordinate per dimensione. Al momento dell'allocazione si cerca l'area più adatta. In questo metodo in particolare si mantengono dei puntatori alle aree di memoria più richieste (es: 4K, 12K, 21K, ecc...) e si istanza un area in base alla grandezza del processo.

### **3.4.1 Problemi dell'Allocazione Contigua**

La necessità di mantenere più processi in memoria dà luogo ad una serie di problematiche che devono essere risolte in modo efficiente ed efficace:

- \* problemi di protezione: bisogna cercare di rendere gli spazi in qualche modo isolati, evitare che i processi possano "sforare" e "pasticciare" il lavoro che altri processi stanno svolgendo.
- \* problemi di rilocalizzazione: far coincidere i processi in partizioni di memoria separate deve far funzionare il tutto in termini di indirizzi utilizzati. Tramite il principio di "allocazione contigua" abbiamo cercato di risolvere più o meno efficacemente questo problema.
- \* problema di ripartizionamento: il problema principale sta nel fatto che la memoria viene suddivisa in partizioni di memoria tra di loro contigue.

Questo fa sì che, al momento in cui un processo viene allocato, si creino degli algoritmi di “compattazione della memoria”, non sempre però efficienti al 100%.

### **3.5 Memoria Virtuale**

Inizialmente, quando i programmi divennero troppo grandi per essere caricati in memoria centrale, il programmatore doveva dividere il suo programma in moduli da far girare uno dopo l'altro sulla RAM, ciò ovviamente era complicato e molto lungo (overlay). Il metodo per ovviare a questo problema divenne noto come Memoria Virtuale. L'idea di base consiste di caricare le parti del programma che devono eseguire job, lasciando il resto sul disco, le parti del programma vengono quindi continuamente caricate e scaricate dalla RAM in base alle necessità. Il concetto di memoria virtuale si basa sul concetto di paginazione. La paginazione ci permette di abbandonare il modello stretto della allocazione contigua, suddividendo la memoria in pagine. La dimensione delle pagine è solitamente fissa, si rischia frammentazione interna che si presenta quando la pagina è troppo grande. E' utile però ad evitare quella esterna, che possiede un vincolo notevole (quello della contiguità). Alcune pagine possono anche non risiedere in memoria, venendo all'occorrenza caricate dalla memoria secondaria (vengono poste in un'area di paging, simile all'area di swap). Vi è una protezione implicita tra processi.

#### **3.5.1 Spazio di Indirizzamento Virtuale**

La Paginazione usa lo spazio di indirizzamento virtuale per ogni processo. Ogni programma lavora con indirizzi da 0 a max. Max dipende dall'architettura. In un sistema a 32 bit, gli indirizzi generati dalla CPU sono a 32 bit. Quindi  $\text{max} = 2^{32} - 1$  byte  $\rightarrow$  4 gb. Quindi ogni processo lavora come se avesse 4gb di RAM a disposizione, nonostante magari non abbiamo nemmeno 4gb di RAM fisica installata. Sarà il sistema stesso che si occuperà di concedere lo spazio promesso, scegliendo quali dati mantenere in RAM e quali no.

## ESERCIZIO

Consideriamo un sistema che fa uso di memoria virtuale con le seguenti caratteristiche: uno spazio di indirizzamento virtuale da 1 GB, un numero di pagina virtuale a 22 bit e un indirizzo fisico a 20 bit. Determinare esattamente quanti frame fisici ci sono in memoria.

**Spazio di indirizzamento virtuale**  $1\text{ GB} = 1024^3 \text{ byte} = (2^{10})^3 = 2^{30} \text{ byte}$

Con un **numero di pagina virtuale a 22 bit** possiamo individuare un numero di pagine pari a:  
 **$2^{22}$  pagine**

Adesso possiamo calcolare la dimensione di una pagina con  $\frac{\text{dimensione indirizzo virtuale}}{\text{numero di pagine}}$

$$\text{dimensione di una pagina} = \frac{2^{30}}{2^{22}} = 2^8 = 256$$

L'indirizzamento fisico è a 20 bit quindi la dimensione totale della RAM è di  $2^{20}$  byte Sapendo che la dimensione di una pagina è 256 byte il numero di frame in memoria RAM è:

$$\text{numero di frame} = \frac{2^{20}}{256} = \frac{2^{20}}{2^8} = 2^{12} = 4096$$

### 3.6 Paginazione

E' un'astrazione ad alto livello molto comoda, fortemente supportata dall'hardware e utile a far combaciare lo spazio di indirizzamento virtuale con il sistema di indirizzi fisici nella memoria centrale.

Infatti il principale problema della gestione della memoria è raccordare gli indirizzi virtuali con gli indirizzi fisici. Questa mappatura è gestita dalla Memory Management Unit (MMU), che possiamo immaginare frapposta tra CPU e memoria. Si suddivide la memoria fisica in frame (o pagine fisiche) e la memoria virtuale in pagine, ad ogni pagina corrisponde un frame. Per tener traccia della mappatura si usa la tabella delle pagine, una funzione che mappa le pagine virtuali con il numero di frame che contiene la pagina. Esiste un bit di presenza che dice se la pagina è mappata in memoria. Nel caso la pagina incontri l'errore di Page Fault, subentra il SO che si occupa di portare il contenuto della Page Fault in uno slot libero.

\*Page Fault: eccezione generata quando un processo cerca di accedere a una pagina che è presente nel suo spazio di indirizzamento virtuale, ma che non è presente nella memoria fisica poiché mai stata caricata o perché precedentemente spostata su disco.

Di tabelle delle pagine ne esistono tante quante sono i processi.



Usualmente il comando per prendere il contenuto di uno spazio di memoria e assegnarlo a REG è: MOV REG, X

Dove X è l'indirizzo di memoria virtuale.

Quando l'MMU riceve questo comando calcola a quale allocazione di memoria fisica corrisponde e manda quest'indirizzo in uscita sul BUS.

### **3.6.1 Mappatura Funzionamento**

Alla MMU viene passato un indirizzo (virtuale) da prendere, per esempio gli viene passato il comando MOV REG, 20, cioè un programma cerca di accedere alla memoria di indirizzo 20, che cade nella pagina (virtuale) 0 (da 0 a 4095 K) la quale, secondo una funzione di traduzione (che potrebbe ricercare l'indirizzo libero più vicino), corrisponde alla pagina fisica 2 (8192 K – 12287 K). Di conseguenza trasforma l'indirizzo in  $(8192+20=)$  8212, mandandolo in uscita sul BUS. Effettivamente l'MMU ha trasformato gli indirizzi virtuali da 0 a 4K in indirizzi fisici tra 8K e 12K. In questo modo è evidente che non tutte le frame possono essere mappate nelle pagine, quindi abbiamo bisogno di un bit di controllo presente/assente che ci indica se la pagina è mappata o meno. Nel caso di un page fault (nel caso si tenti di accedere ad una pagina non mappata) l'errore è gestito dal SO e questo prende una delle pagine meno usate, ne salva il contenuto sul disco, la rialloca, fa riferire la pagina nella memoria appena liberata, cambia i bit di controllo e quindi fa ripartire l'istruzione interrotta. Per esempio se cerco di accedere all'indirizzo virtuale 25K, questo avrà bit di presente/assente = 0 (infatti non è mappato ad un indirizzo fisico).

### **3.6.2 Tabella delle Pagine [voce]**

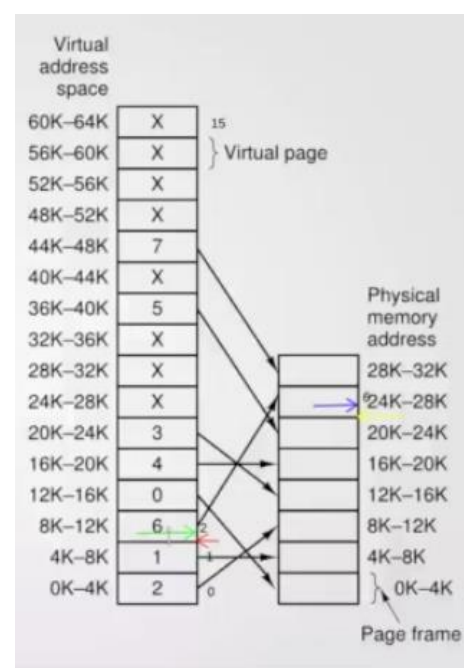
Non in tutti i sistemi gli elementi delle tabelle sono uguali, ma seguono pressappoco tutti lo stesso schema.

- Numero di pagina fisica (Page Frame Number): Il campo più importante, lega la frame alla pagina.
- Bit presente/assente (Present/absent): Se è uguale ad 1 l'elemento è valido, se è 0 si andrà incontro ad un Page Fault.
- Bit Protezione (Protection): Indica i tipi di permessi: se è 0 si ha diritto di lettura/scrittura, se è 1 solo lettura.

- Bit Modificato (Dirty Bit): Inizialmente la pagina ha questo bit settato a 0, comunica se la pagina è stata mai modificata dopo essere stata portata in memoria, cioè se la copia della pagina della memoria centrale è uguale a quella della pagina che sta leggendo (se è “sporca”). Viene modificato quando l’MMU vede che la pagina viene aperta in modalità scrittura. Quando è 0 la pagina può essere scartata tranquillamente, se nel caso fosse 1, il SO deve prima preservare il vecchio contenuto, sincronizzando il contenuto tra la memoria centrale e secondaria.
- Bit di Referenziamento (referenced): inizialmente posto a 0, se si fa un qualunque referenziamento (lettura/scrittura) a quella pagina. Serve a decidere quali pagine in memoria sacrificare e quali no, servono a fare una statistica su quali pagine sono utilizzate e quali no. L’informazione è relativa ad un giro di clock.
- Bit per disabilitare la cache (Chaching disabled): Può essere utile per disabilitare una pagina in cui avvengono mappature di I/O.
- Bit di validità o allocazione: un processo può utilizzare tutto lo spazio di indirizzamento, ma materialmente nell’allocazione già si perde della memoria. Il bit di validità indica se quella pagina è già stata allocata.

## ESERCIZIO //traduzione da indirizzo virtuale a indirizzo fisico

- 1) Supposto di avere un indirizzo virtuale di una page (8196)
- 2) Si divide l’indirizzo per la dim. di ogni page (4096K)
- 3)  $8196/4096=2,009$ . -> quoziente=2 resto=4 (offset)
- 4) Il quoziente indica l’indice della pagina virtuale (partendo da 0).
- 5) Il resto indica l’offset, ovvero la distanza dalla word che è stata richiesta e l’inizio della pagine che ospita quella word.
- 6) Nella pagina virtuale 2 trovo l’indirizzo del Frame relativo (6)
- 7) Moltiplico  $6 * \text{dimFrame}$  (es 4096) = 24576.
- 8) Questo valore è l’indirizzo fisico che identifica dove inizia la word nella tabella dei Frame.
- 9) A questo valore sommo l’offset e trovo, partendo da un indirizzo virtuale, il relativo indirizzo fisico nel frame.



## ESERCIZIO

1 gb = ( $2^{30}$ ) spazio di indirizzamento virtuale.

22 bit (indice)#pagina virtuale.

20 bit indirizzo fisico.

Quanti frame fisici abbiamo a disposizione?

In altri termini ci interessa capire quanti slot ho nello schema risultante.

8 di 34

---

**Primo metodo risolutivo:** Lavoriamo con le dimensioni degli indirizzi e indici  
Se  $2^{30}$  è il nostro spazio di indirizzamento virtuale( indirizzo virtuale di 30 bit), e l'indice #pagina virtuale è di 22 bit, allora il nostro indirizzo virtuale sarà così composto:

[ #paginaVirtuale (22bit) | offset (x)], quindi  $22+x = 30 \Rightarrow x = 8$ .

Se un indirizzo fisico è da 20bit, allora essendo l'indirizzo fisico composto così:

[#indiceFrame(x) | offset(8bit) ] quindi  $8+x = 20 \Rightarrow x = 12$

Quindi abbiamo  $2^{12} = 4096$  frame fisici, considerato che l'indice dei frame è di 12 bit.

**Secondo metodo risolutivo:** Lavoriamo con le dimensioni degli elementi

Quando diciamo che abbiamo 1GB( $2^{30}$  byte) di spazio di indirizzamento virtuale, dividendolo per la quantità di pagine che abbiamo( $2^{22}$  byte), posso determinare la dimensione della singola pagina. ( $2^{30}/2^{22}$ ), è uguale a  $2^8$  byte. Ovvero la dimensione di una singola frame o pagina sarà  $2^8 = 256$  byte.

## ESERCIZIO

*Si supponga che una macchina abbia indirizzi virtuali a 48 bit e indirizzi fisici a 32 bit. (a) Se le pagine sono di 4 KB, quante voci ci sarebbero nella tabella delle pagine se fosse a un solo livello? Si dia una spiegazione. (b) Si supponga che lo stesso sistema abbia un TLB con 32 voci. Inoltre, si assuma che un programma contenga istruzioni che stanno esattamente in una pagina e legga sequenzialmente elementi interi lunghi da un array che si estende su migliaia di pagine. Quanto è indicato il TLB in questo caso?*

(a) Se le pagine sono di 4 KB, servono 12 bit per rappresentare l'offset all'interno di una di esse ( $2^{12} = 4096$ ). Quindi restano  $48 - 12 = 36$  bit per il numero di pagina virtuale: conseguentemente ci saranno  $2^{36} = 64G$  entry nella tabella delle pagine nel caso di una paginazione ad un livello.

(b) È poco indicato perché la lettura (sequenziale) degli elementi dell'array coinvolge più di 32 voci e quindi si avranno continui fallimenti accedendo al TLB, rendendo di fatto inutile la presenza della memoria associativa se non per la singola voce facente riferimento alla pagina delle istruzioni.

## ESERCIZIO

*Calcolare il numero di pagine virtuali e offset per una pagina a 4KB e una a 8KB, per ognuno dei seguenti indirizzi virtuali decimali: 20000, 32768 e 60000*

La formula è:  $(\text{indirizzo virtuale}) / (4KB * 1024) = 4,88 \rightarrow$  il valore intero "4" è il numero di bit e  $0,88 * 4096 =$  l'offset

### 3.6.3 Tabelle dei Frame

Ha tante voci quanti sono i frame, ogni voce dice se un frame è allocato o meno e rende possibile distinguere i frame liberi da quelli occupati e sapere da cosa sono occupati. Viene chiamata in causa nel momento in cui è necessario allocare un nuovo processo o ulteriore memoria di un processo già esistente. Gli aspetti da curare sono la dimensione e la velocità di consultazione.

- Per mantenere un accesso rapido si potrebbe prevedere un vettore di registri ad alta velocità interni all' MMU idoneo a contenere la tabella delle pagine per fare le traduzioni. Ma questo modello appesantisce il context switch man mano che le tabelle crescono di grandezza.

- Un'altra idea è avere un registro PTBR (Page-Table Base Register) ed un puntatore che punta all'indirizzo fisico ove si sviluppa la tabella in memoria fisica. L' MMU calcola l' l-esimo elemento e crea il fetch per la traduzione di quell'elemento. Ogni volta che devo accedere alla memoria, devo fare un referenziamento al registro per accedere e un altro tramite l' MMU per prelevare, rendendo l'accesso molto più lento dato che sono 2 accessi alla memoria. Il context switch è però molto veloce.

### 3.6.4 Memoria Associativa (TLB)

La Translation LookASIDe Buffer (TLB) è una piccola tabella contenuta all'interno dell'MMU che si frappone tra MMU e Memoria Centrale ed ha un meccanismo simile alla cache. Essendo che in genere un programma usa un gran numero di riferimenti ad un piccolo numero di pagine, essa contiene un numero limitato di voci che rappresentano i registri più utilizzati. Ciò ci permette di evitare costi extra di accesso alla memoria ad ogni richiesta di traduzione.

Quando viene consultata dall'MMU: questa prima vede se ciò che cerca è nella TLB (dove la ricerca è ottimizzata), altrimenti fa un fetch alla memoria centrale. La TLB viene gestita dall'HW e dovrebbe contenere il set di pagine attualmente in uso dal processo in esecuzione.

In una TLB trovo:

- Numero di pagina virtuale: è il campo di ricerca chiave.
- Numero di frame: ciò che cerco.
- Maschera dei diritti: sempre a disposizione dell'MMU.
- Dirty Bit: può succedere che il bit non sia propriamente sincronizzato, e quindi devo tenerlo in conto.
- Bit di validità

#### 3.6.4.1 TLB Hit or Miss

La sua presenza, in casi fortunati, permetterà di evitare di dover accedere alla tabella delle pagine. Questo processo viene formalmente chiamato "TLB hit" (quando l'MMU trova la page). L'idea è quella che se l'MMU, una volta ottenuto l'indirizzo da tradurre, se trova le informazioni che gli servono all'interno della TLB può evitare di accedere alla memoria centrale. ALTRIMENTI, si parla di un "TLB miss", caso in cui cioè non si trovano le informazioni che servono. Caso sicuramente "meno performante", cioè in cui si ha necessità di accedere alla memoria centrale per consultare la tabella delle pagine. Entrambe hanno un ruolo simile quindi alla "memoria cache", per il fatto che, rispettivamente, si ha o meno la possibilità di accedere alla memoria centrale per prelevare informazioni.

### 3.6.4.2 Ricerca Parallelizzata e Possibilità di Voci Vincolate

- La ricerca viene effettuata tramite hardware, si parla infatti di “ricerca parallelizzata”, che consente una ricerca molto più efficiente e veloce.
- Può succedere che, cambiando spazio di indirizzamento di lavoro (cambiando processo), per far funzionare le cose correttamente, sarebbe necessario azzerare la TLB nell’ambito del context-switch. Ovvero, cambiando tabelle di processi, si è costretti ad azzerare tutte le tabelle a 0: questo perché mantenere le “vecchie voci” risulterebbe pericoloso. Fare in sostanza operare la CPU, in concomitanza con l’MMU, su delle informazioni relative ad un’altra tabella, vecchia e praticamente non utile, risulterebbe rischioso e potrebbe creare ambiguità.

### 3.6.4.3 ASID vs FLUSH

- address-space identifiers (ASID): è un termine molto simile (come concetto) a quello del PID (Process ID). Non è altro che un termine identificativo del processo, che ci aiuta a distinguere le pagine dei singoli processi. Con questo identificativo si fa anche riferimento al numero della pagina a cui il processo fa riferimento, concetto molto importante perché evita di creare ambiguità e di azzerare praticamente le tabelle a 0.
- flush della TLB: quando viene azzerata la TLB.

### 3.6.4.1 Effective Access Time (EAT)

Indica la stima di quanto tempo ci vorrà per un accesso in TLB, il quale dipende da:

- quanto tempo ci vuole per accesso a memoria e accesso a TLB.
- se ci sono TLB miss il tempo di accesso alla memoria raddoppia, perché si fanno 2 accessi in memoria centrale.
- dal TLB ratio (percentuale di successi)

In generale, siano:

- $\alpha$  : Tempo di accesso alla memoria
- $\beta$  : Tempo di accesso a TLB
- $\epsilon$  : TLB ratio (rapporto tra TLB hit e numero di accessi alla TLB).

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

### 3.6.5 Tabella Pagine Multilivello

Questo tipo di tabella è utile in quanto consente di risparmiare spazio, le pagine inutili infatti non vengono allocate e quindi non appesantiscono il sistema. È possibile suddividere la tabella in gruppi omogenei trattati come figli di un nodo radice. La struttura è gestita tramite una tabella, che contiene dei puntatori alle pagine virtuali dei gruppi più bassi. È possibile che elementi della tabella non puntino a nulla (così da risparmiare memoria). Il numero di accessi alla memoria sono tanti, data l'elevata complessità della struttura dati: un accesso alla tabella di primo livello, un altro accesso per la tabella di secondo livello ed un ulteriore accesso al dato. La tabella delle pagine multilivello permette quindi di ridurre e rendere quASI proporzionale al numero effettivo di pagine allocate il numero di tabelle effettivamente usate. Mantenere tutte le tabelle in memoria centrale risulterebbe pesante, maggiormente nei moderni sistemi a 64bit. Soluzione? Non mantenere l'intera tabella in memoria.

Ma invece utilizzare una struttura gerarchica, una sorta di albero.

Anziché avere ad esempio 1 milione di pagine in memoria, ho una tabella da 1024B in memoria centrale, collegata ad ulteriori 1024 tabelle di pagine che risiedono in memoria secondaria. La tabella alla radice dell'albero ha i riferimenti alle tabelle delle pagine di livello 2.

Come raggiungere, dalla radice, un indirizzo a livello due?

La voce 0 della tabella radice punta a 1024 voci nella tabella di secondo livello. La voce 1 della tabella radice punta a 1024 voci nella tabella di secondo livello e così via...

Preso un indirizzo a 32bit della tabella radice, si fa il calcolo come sopra.

- i 10 bit più significativi (PT1) mi dicono qual è il gruppo.
- i 10 bit meno significativi (PT2) mi dicono qual è la posizione della pagina in quel gruppo.
- i restanti 12 bit danno l'offset, informazioni su quale pagina specifica riferirci.

### 3.6.6 Tabella Pagine Invertita

Esistono altri approcci che gestiscono gli spazi di indirizzamento a 64 bit.

Immaginiamo una tabella con tante voci quanti sono i frame e non le pagine virtuali. Al suo interno deve contenere un bit di controllo (se il

frame è allocato o meno), il numero di pagina virtuale e un identificativo univoco che distingue gli spazi di indirizzamento (PID). Questo è il numero minimo di bit di referenziamento che mi servono. Per supportare la ricerca è necessaria una struttura interna decente, altrimenti la ricerca non sarebbe efficiente, si utilizza quindi una Tabella Hash: una tabella più snella dove più voci della tabella corrispondono ad una sola, quindi vengono messe in lista. La TPI mi permette di effettuare una traduzione da indirizzo virtuale ad indirizzo fisico effettuando una ricerca all'interno della tabella (è indicizzata per indici fisici, quindi devo ricercare nella tabella l'indice virtuale e l'ASI, non posso accedervi direttamente).

Ovviamente è necessaria solo una tabella delle pagine invertita.

Ha come contro che la ricerca è decisamente inefficiente, il che significa che se dovessimo avere un blocco molto grande di informazioni risulterà sempre più difficile andare alla ricerca del processo a cui si vuole fare riferimento. In questo caso viene cambiata la visualizzazione, anche se di base ogni processo viene comunque indicizzato. Inoltre il fatto che si abbia soltanto una sola tabella non significa che l'Nesima (l'ultima) abbia informazioni su tutte le altre tabelle precedenti. Questo significa, in soldoni, che aggiornando la tabella si perdono informazioni sul "passato", si perdono informazioni necessarie per gestire i "page fault". Ma dove mantengo l'informazione riguardo dove prendere la pagina in HD?

Ho ancora bisogno delle n tabelle legate ai singoli processi, ma solo per il caso dei page fault. Potrei paginare queste tabelle su disco, rallentando un poco solo il meccanismo dei Page fault. E' un meccanismo macchinoso e scomodo, inoltre la tabella delle pagine invertite ha un problema con le pagine condivise, per cui non si possono mantenere più ASI. Bisogna usare tabelle di corrispondenze alias ausiliarie e questo è scomodo.

A differenza della Tabelle Multilivello utilizzata tutt'oggi, essa ultimamente non viene mai presentata, è una forma di architettura poco efficiente.

### **3.6.7 Cache Memoria vs Memoria Virtuale**

Questo è il caso in cui la "memoria cache" viene inserita dopo l'MMU.

Come deve comportarsi?

- non serve invalidarla sul context-switch.
- caching poco efficace.



Abbiamo una CPU che genera indirizzi virtuali, questi successivamente vengono passati all'MMU (eventualmente con l'aiuto della TLB). Viene quindi fatta un'interrogazione alla cache per indirizzo fisico. Una cache è formata da linee di cache, identificate dall'indirizzo fisico a partire dal quale inizia questa linea di cache e da una chiave, che difatti permette di capire se una determinata richiesta ricade o no in una delle linee di cache contenuta all'interno dell'unità buffer-cache. In questo caso non sono obbligato ad azzerare la cache perché non ho pericolo di confusione. Questa ricerca, in questo caso, avviene per indirizzo fisico, teoricamente senza problemi. Il problema, però, sta nel fatto che la cache risulta quasi penalizzata poiché inserita in cascata (dopo) l'MMU. In sostanza, per capire se una word è o meno presente all'interno della CPU, dobbiamo "pagare" il costo di una traduzione da indirizzo virtuale ad indirizzo fisico. Questo, a parità di condizioni, rappresenta una sorta di limite, di "collo di bottiglia".

Questo è il caso in cui la "memoria cache" viene inserita prima dell'MMU. Come deve comportarsi?

- servono gli ASID per non invalidarla ogni volta: scala male su L2.
- maggiore efficacia

L'idea di porre la cache prima dell'MMU permettere idealmente di migliorarne l'efficacia, perché non si dovrebbe "pagare" il costo della traduzione a priori; eventualmente andrebbe pagato solo dopo (nel caso in cui la cache non avrebbe aiutato in tale processo). Il non azzerare questo tipo di cache è un problema, siamo obbligati ad azzerarla poiché in ogni linea di cache ho un identificativo indicativo, e nel caso di cambio di contesto, gli spazi di indirizzamento di un altro processo esistono, ma non contengono i dati giusti e ciò dà vita a problemi. Per ovviare a ciò è necessario identificare ogni spazio non solo con l'ID, ma anche con un identificativo del processo (ASID). Potenzialmente dovrebbe essere più efficace perché, se si è fortunati, ci si può trovare in un caso di "cache hit" senza dover pagare il costo di una traduzione, poiché questa viene posposta e limitata solo ai "casi sfortunati" (i cosiddetti "cache miss"). Una cache siffatta dovrebbe essere idealmente azzerata ad ogni context-switch, nel caso ad esempio di un passaggio da un processo ad un altro.

Cosa si usa in pratica?

Cache L1 basata su indirizzi virtuali, Cache L2 può essere di entrambi i tipi. La CPU potrebbe passare la richiesta sia alla cache L1 sia alla MMU (così da anticipare un possibile cache-miss). Tipicamente questi approcci si usano entrambi. La cache di livello L1, quella che sta più vicina alla CPU, non pone il problema dell'azzeramento quindi non creando confusione, mentre la cache di secondo o terzo livello è spesso dopo l'MMU e lavora con ind. fisici

## ESERCIZIO

*In un sistema che usa paginazione, l'accesso al TLB richiede 150ns, mentre l'accesso alla memoria richiede 400ns. Quando si verifica un page fault, si perdono 8ms per caricare la pagina che si sta cercando in memoria. Se il page fault rate è il 2% e il TLB hit il 70%, indicare l'EAT ai dati.*

Convertendo tutti i tempi in ms, abbiamo:

- tempo di accesso al TLB:  $150 \text{ ns} = 150 \cdot 10^{-6} \text{ ms}$ ;
- tempo di accesso alla memoria:  $400 \text{ ns} = 400 \cdot 10^{-6} \text{ ms}$ ;
- tempo di gestione del page fault: 8 ms;
- page fault rate:  $2\% = 0,02$ ;
- TLB hit:  $70\% = 0,7$ .

Quindi

$$\begin{aligned} EAT &= TLB \text{ hit} \cdot (150 \cdot 10^{-6}) + (1 - \text{page fault rate}) \cdot (150 \cdot 10^{-6} + 400 \cdot 10^{-6}) + \text{page fault rate} \cdot \\ & (150 \cdot 10^{-6} + 8 + 400 \cdot 10^{-6}) \\ &= 0,7 \cdot (150 \cdot 10^{-6}) + 0,28 \cdot (150 \cdot 10^{-6} + 400 \cdot 10^{-6}) + 0,02 \cdot (150 \cdot 10^{-6} + 8 + 400 \cdot 10^{-6}) \\ &= 0,7 \cdot (1,5 \cdot 10^{-4}) + 0,28 \cdot (1,5 \cdot 10^{-4} + 4 \cdot 10^{-4}) + 0,02 \cdot (1,5 \cdot 10^{-4} + 8 + 4 \cdot 10^{-4}) \\ &= 1,05 \cdot 10^{-4} + 1,54 \cdot 10^{-4} + 0,160011 \\ &= 0,1627 \text{ ms} \end{aligned}$$

### 3.7 Algoritmi di Sostituzione delle Pagine

Un aspetto della paginazione che coinvolge prettamente il sistema operativo è quello del page fault. Che cosa succede quando l'hardware (l'MMU), ricercando all'interno della tabella delle pagine non trova riscontro? O meglio, trova sì il record della pagina, ma trova un record che attraverso il bit di presenza/assenza comunica all'MMU che la traduzione non si può fare, perché la pagina virtuale in cui ricade la word da ricercare non si trova all'interno della memoria centrale.

L'hardware è la componente, in questo caso, che rileva il problema. Se però non riesce ad risolvere il problema "se ne lava le mani", nel senso che cede il tutto al sistema SW che (si spera) sia in grado di soddisfare la richiesta. Bisogna però allo stesso tempo evitare che quest'ultimo "trattenga" per troppo tempo il processo, per evitare che il sistema operativo sostanzialmente rallenti o perda tempo inutilmente.

Cosa accade nel caso di page fault?

Se si vuole andare a ricercare dove si trova un determinato processo, quest'ultimo si ricerca all'interno di una pagina. Una volta trovato, è però altresì importante capire dove inserirlo, cercando di scegliere la posizione migliore.

Esistono 2 casi, relativi all'inserimento di un processo:

- caso 1: è il più fortunato. È il caso in cui il processo viene subito trovato ed inserito all'interno della CPU. Viene modificato il bit di presenza/assenza e "tutto fila liscio".

- caso 2: caso "sfortunato", ma tutt'altro che raro. Caso in cui tutti i frame/gli slot della memoria fisica ad un certo punto sono occupati, pieni. In questo caso il sistema operativo, tramite delle metodologie particolari, dovrà cercare di scegliere tra le pagine virtuali, quella che diventerà una "pagina vittima" da spostare dalla memoria centrale al disco. È un problema simile al cache-miss, ma questo è da gestire da lato SW (il cache-miss è gestito dall'HW).

\*ATTENZIONE: l'astrazione deve funzionare garantendo che il contenuto della pagina vittima non si cancelli. Questo significa che, in sostanza, nessun dato sarà cancellato: non vi è perdita di informazione. Questo significa anche che, se in futuro il sistema dovesse avere nuovamente bisogno di quella determinata pagina, il suo contenuto sarà nuovamente raggiungibile. In questo contesto, entra in gioco quello che prima abbiamo definito come "bit di modifica": bit presente all'interno della tabella delle pagine che mi dice, nel caso in cui ci sia una copia su disco, se questa è sincronizzata con la copia che è presente in memoria centrale.

- il sistema operativo, nel caso in cui questo bit sia "0" (la pagina risulta "pulita"), può quindi anche scegliere di "non fare operazioni". Meglio, concettualmente la pagina viene spostata su disco, ma praticamente non viene fatta alcuna operazione di I/O.

- se invece il bit è posto a "1" (quindi la pagina risulta "sporca", già utilizzata), si presenta una soluzione un po' più pesante (in termini di complessità), per il fatto che il sistema operativo in questo caso dovrà preoccuparsi di non eliminare il contenuto della pagina su cui si vuole sostituire l'informazione. Questo caso comporta un'operazione di I/O in più. Questa scelta non è implementabile in modo efficiente: bisognerebbe numerare le pagine indicando il numero di istruzioni che verranno eseguite

prima di riferirsi nuovamente alla pagina, pagine con numeri più alti sarebbero potenziali pagine vittime.

Ora vediamo soluzioni reali e implementabili.

### **3.7.1 Algoritmo OPT**

E' la soluzione ottimale.

E' importante tenere conto del fatto che, quanto più tempo l'algoritmo di risoluzione è pesante (impiega tempo), quanto più la gestione del page fault sarà rallentata.

Scegliamo la pagina che verrà referenziata in un futuro più lontano: in generale riduce la probabilità e il costo della penalizzazione che si presenterà. E' una scelta ottimale ma difficilmente realizzabile: avrebbe dei costi che supererebbero i benefici di una simile soluzione.

Rappresenta comunque un termine di paragone: è però importante pensare di tenere conto questa soluzione come una "soluzione simulata", in modo tale da utilizzarla come "metrica di confronto" per altre soluzioni. In generale, meno page fault saranno presenti, più veloce sarà l'algoritmo a completare la soluzione.

### **3.7.2 Algoritmo Not Recently Used (NRU)**

La scelta è ponderata e mirata per cercare di scegliere tra le pagine presenti in memoria, "quella che non è stata recentemente utilizzata".

Si tratta di un algoritmo sicuramente migliore e più efficiente rispetto all'algoritmo che scarta le pagine utilizzate di recente, per il fatto che se tot. pagine sono state utilizzate di recente è probabile che queste vengano riutilizzate nell'immediato futuro. A tal proposito quindi, se queste vengono scartate, si creerebbe una situazione di spreco ulteriore dovuta al "ripescaggio" delle pagine cancellate.

Raccogliamo un po' di statistiche sull'uso delle pagine caricate:

- bit di referenziamento (R) e di modifica (M): è un bit che in realtà è in parte gestito dall'HW e in parte gestito dal SW.

L'HW si occupa di aggiornarlo e portarlo ad "1" (se non lo è già), nell'ambito di un referenziamento della pagina. Il SW, invece, si occupa di azzerare il bit di referenziamento, periodicamente.

Si possono però distinguere delle classi di pagine, in particolare 4, ognuna

delle quali corrispondono ad una possibile combinazione tra il bit di referenziamento e il bit di modifica.

In ordine di efficienza:

- classe 0: (classe “ideale” da scegliere) non referenziato e non modificato;
- classe 1: non referenziato, modificato;
- classe 2: referenziato, non modificato;
- classe 3: referenziato, modificato.

Note:

- se il bit di referenziamento è trovato a “1” significa che, nell’ultimo secondo, la pagina è stata usata. Il che significa che, chiaramente, questa pagina non va scartata. Le pagine da scartare sono invece quelle che presentano come bit di referenziamento “0”.
- in generale, viene scelta una pagina dalla classe non vuota di numero più basso.

## ESERCIZIO

*Supponiamo le pagine attualmente in memoria siano le seguenti (senza nessun ordine particolare):*

*$A(r=1, m=0)$ ,  $B(r=1, m=1)$ ,  $C(r=0, m=1)$ ,  $D(r=1, m=1)$ ,  $E(r=1, m=0)$ .*

*I bit indicati tra parentesi sono rispettivamente i bit di referenziamento e di modifica.*

*Quale sarebbe la pagina selezionata per la sostituzione dall’algoritmo NRU?*

*La pagina C, che appartiene alla classe 1, più bassa delle altre.*

### 3.7.3 Algoritmo First In First Out (FIFO)

Viene rimossa la pagina più vecchia: si pensi ad una lista concatenata in cui ho all’estrema sinistra le pagine più vecchie e sulla destra quelle più recenti. L’ordine è legato all’età delle pagine. La lista viene aggiornata inserendo le pagine appena arrivate in memoria sulla dx e sulla sx il candidato scartato poiché “vecchio”. La scelta non è semplice: può rimuovere pagine, sì vecchie, ma magari molto usate.

Il fatto che una pagina non sia da diverso tempo in memoria non significa che quella determinata pagina non è più necessaria. Si potrebbe infatti trattare di una porzione di dati per il mio processo sistematicamente utilizzati per lo stesso.

### 3.7.4 Algoritmo della Seconda Chance

Si prevede sempre l’utilizzo di una lista che operi mediante il meccanismo di una lista FIFO, ma si opera con una “seconda chance”. Questo significa che, la pagina che sta sul punto di essere eliminata, ovvero quella più

vecchia e non usata di recente, può essere “risparmiata” se e solo se possiede il bit di referenziamento ad 1;  
se è a 0 la tira fuori ed azzerava il bit di referenziamento.

## ESERCIZIO

*Supponiamo di avere una coda FIFO delle pagine attualmente in memoria:*

A (r=1, m=0), B (r=1, m=1), C (r=0, m=0), D (r=0, m=1), E (r=1, m=0)

*Tale lista è ordinata secondo l'ordine d'arrivo (A è quella più vecchia). I bit indicati tra parentesi sono rispettivamente il bit di referenziamento e il bit di modifica. Quale sarebbe la pagina selezionata per la sostituzione dall'algoritmo della Seconda chance?*

La pagina C perché è la prima che nello scorrimento della coda ha R=1.

### 3.7.5 Algoritmo di Clock

E' lo stesso criterio di scelta applicato nell'algoritmo della Seconda Chance, applicato però in altra maniera. L'idea è quella che è presente una coda circolare, tramite cui con una “lancetta” (puntatore) mi sposto tra gli elementi e si aggiornano opportunamente i bit di referenziamento del SO con una complessità  $O(n)$ . Si ha un notevole risparmio di risorse.

### 3.7.6 Algoritmo Least Recently Used (LRU)

Guarda alle pagine come sono usate di recente, stila una classifica e sceglie la pagina che è stata usata meno di recente. Mantiene un contatore associato alle pagine (potrebbe essere il numero di istruzioni eseguite dalla CPU dopo aver consultato la pagina), pagine con contatori maggiori saranno potenziali vittime. È un'ottima approssimazione dell'algoritmo ottimale, ma è complicata e dispendiosa: ad ogni nuovo referenziamento le istruzioni devono essere aggiornate. Vediamo ora varie soluzioni, sia HW che SW.

- Con supporto HW:

\*Con un contatore nella CPU e campi relativi nella tabella delle pagine:

L'idea potrebbe essere quella di, ogni volta che la CPU fa un accesso ad una determinata pagina (quindi prima ad una word relativa alla pagina), possiamo immaginare che sia l'MMU a leggere il valore del contatore che si trova attualmente nella CPU e va a copiare questo valore all'interno del campo corrispondente per quella determinata pagina. Questo contatore è una sorta di timestamp, quindi, più è alto più indica un evento recente.

Ogni volta che l'MMU fa accesso ad una determinata pagina questo contatore viene copiato all'interno dell'apposito campo della pagina relativa. Si andrebbero ad attenzionare tutti i contatori delle matrici, per vedere qual è il più basso da eliminare. A tal proposito quindi non si terrebbe conto del bit di referenziamento: l'ordine dell'LRU permette di dare una visione molto più precisa e dettagliata dell'ordine in cui i processi sono più o meno utilizzati. Quest'ultima è però una soluzione molto dispendiosa e praticamente irrealizzabile, pertanto si utilizza il bit di referenziamento.

\*Con tramite una matrice di bit:

Il numero di righe e colonne corrisponde al numero di frame effettivi. Ogni volta che avviene l'accesso all'i-esimo frame vengono posti a 1 l'intera i-esima riga e posta a 0 l'intera i-esima colonna.

Si sceglie il frame che presenta il numero minore di bit posti a 1.

### **3.7.6.1 Algoritmo Not Frequently Used (NFU)**

Approssimazione LRU, meno dispendiosa. Uso un contatore associato ad ogni pagina, li aggiornano periodicamente in modo SW con frequenza bassa sfruttando il bit di referenziamento: prima di resettarlo periodicamente tramite la procedura, lo sommo al contatore della rispettiva pagina, cosicché pagine che sono rilevate con il bit di referenziamento pari ad 1 avranno il contatore maggiore, quindi le pagine vittime saranno quelle col contatore minore. Può erroneamente privilegiare pagine che sono state molto utilizzate in passato ma che invece sono scarsamente usate adesso.

### **3.7.6.2 Algoritmo di Aging**

Approssimazione LRU, più precisa rispetto a NFU.

Ad ogni scadenza del clock:

- fa uno shift a destra del contatore associato ad ogni pagina.
- accostamento a sinistra (come bit più significativo) del bit R.

Ogni bit del contatore di una pagina rappresenta un intervallo di tempo in cui il rispettivo bit di referenziamento è risultato 0 o 1 in base a come è stata utilizzata la pagina. In sostanza, si tiene conto di un vero e proprio storico della pagina a cui si fa riferimento. Viene quindi scelto in maniera precisa la pagina che si vuole scartare.

### 3.7.7 Confronto delle Prestazioni

Per confrontare l'andamento e le prestazioni di un determinato algoritmo è importante tenere conto di quella che è la **metrica**: ossia conteggiare, a parità delle altre condizioni, quanti page fault provoca utilizzare un algoritmo piuttosto che un altro.

A parità di condizioni, fissiamo:

- **numero di frame**: 3 frame.
- sequenza degli indirizzi virtuali a cui accedere: non è in particolare tenere traccia di ogni singolo indirizzo. L'algoritmo tiene conto principalmente di come sono state effettivamente utilizzate le pagine, di quali pagine contengono quei determinati indirizzi.

#### 3.7.7.1 Prestazioni OPT

- algoritmo **OPT**: non funziona guardando al passato, ma al futuro. Bisogna capire, guardando "al futuro" quale pagina sarà utilizzata e quindi di cui avrò sicuramente bisogno in un secondo momento. Quella pagina non è da cancellare.

Viene utilizzato come paragone perché è l'algoritmo ottimale, è difficile a livello pratico da realizzare ma dà un limite inferiore: **bisogna trovare un algoritmo che abbia un numero di page fault basso**, il più possibile confrontabile con il numero di page fault ottenuti applicando OPT.



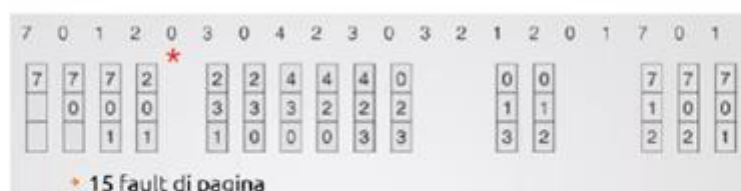
Esempio nella figura (\*): qui, si tiene in considerazione il futuro del programma. Nel caso indicato la pagina che meno si presenta nel futuro è la 7. Pertanto, è la prima ad essere sostituita dalla pagina 2.

In generale, **più basso è il numero di page fault, meglio si sta comportando l'algoritmo**.

Tendenzialmente, non esistono algoritmi che operano meglio dell'algoritmo OPT (ottimale). È importante però tenere comunque conto delle prestazioni degli altri algoritmi.

#### 3.7.7.2 Prestazioni FIFO

- algoritmo **FIFO**:

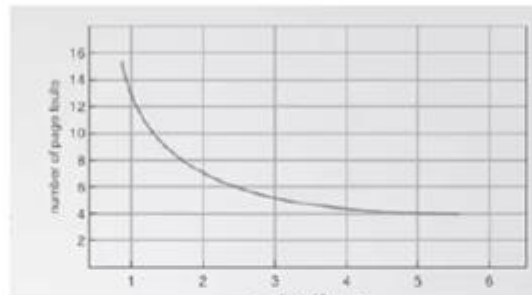


Esempio nella figura (\*): anche qui si tiene in considerazione il futuro del programma. Nel caso indicato la pagina che meno si presenta nel futuro è sempre la 7 (caso identico a quello di prima). Vi è però una sostituzione decisamente poco opportuna, in cui al posto del 7 viene sostituito il 2.



### 3.7.7.2.1 Anomalia di Belady


Un ricercatore che si è occupato dello studio di questi algoritmi, **Belady**, ha osservato un **fenomeno particolare**: alcuni algoritmi, in particolare **l'algoritmo FIFO**, non segue il comportamento desiderato secondo i frame:

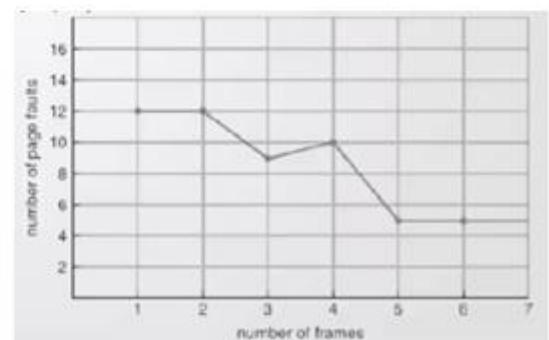


In realtà **Belady** ha scoperto alcuni “casi limite” in cui il comportamento ha un andamento scostante, indesiderato. In particolare, accade che **aumentando il numero di frame disponibili, aumentano il numero di page fault**.

Su un esempio specifico, fornita la sequenza:

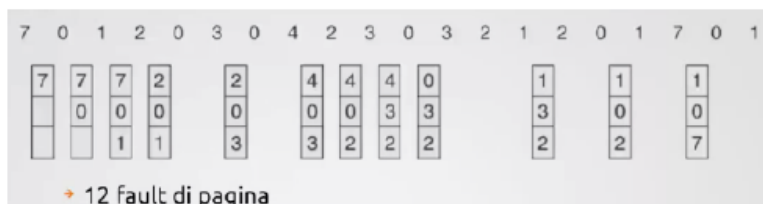
1, 2, 3, 4, 1, 2, 5, 1, 2, 3, 4, 5.

Con  un solo frame in più si presenta l'anomalia (vedi grafico).



### 3.7.7.3 Prestazioni LRU

- algoritmo **LRU**: che ricordiamo essere una soluzione rapida perché “guarda al passato”



Strategia migliore dell'algoritmo FIFO, ma chiaramente non bene tanto quanto l'algoritmo ottimale.

Nella pratica, però, è importante dire che questo è l'algoritmo che nella pratica è frequente e utilizzato: vedi i **meccanismi utilizzati per le cache**, anche hardware.

È un algoritmo detto “**a stack**”, poichè nella sua implementazione è sempre possibile individuare un'implementazione che utilizza uno stack come struttura d'appoggio.

- la cosa molto positiva, invece, sta nel fatto che questo algoritmo **non presenta anomalie di Belady**: si tratta quindi di un algoritmo molto stabile.

### 3.7.7.3.1 Proprietà di Inclusione

**Proprietà di inclusione:** l'insieme di pagine caricate avendo "n" frame è incluso in quello che si avrebbe avendo "n+1" frame. In sostanza questo significa che **il numero di page fault può solo ridurre, non può aumentare.**

$$B_t(n) \subseteq B_t(n+1) \quad \forall t, n$$

### 3.7.7.4 Riepilogo

- altri algoritmi visti: abbiamo visto che molti di questi sono in realtà sfumature, o algoritmi comunque basati su quelli visti in precedenza. Pertanto, questi algoritmi, si comporteranno in maniera consona a quelli su cui si basano o "prendono spunto".

- **NFU, aging:** godono della proprietà di inclusione (appross. LRU).
- **seconda chance, clock:** soffrono dell'anomalia (riducono a FIFO).
- **NRU:** soffre dell'anomalia (riduce a FIFO).
- **OPT:** non implementabile, ma utile come termine di paragone;
- **NRU\*:** approssimazione rozza dell'LRU;
- **FIFO\*:** può portare all'eliminazione di pagine importanti;
- **Seconda chance\*:** un netto miglioramento rispetto a FIFO;
- **Clock\*:** come S.C. ma più efficiente;
- **LRU:** eccellente idea (vicina a quella ottima) ma difficilmente realizzabile se non in hardware;
- **NFU:** approssimazione software abbastanza rozza dell'LRU;
- **Aging:** buona approssimazione di LRU con implementazione software efficiente.

#### Cosa si usa nei sistemi reali?

\* soffre dell'anomalia di Belady.

- Varianti della Seconda Chance più ottimizzate
- NFU come variante di LRU.

## 3.8 Allocazione dei Frame

Vediamo ora che logica segue il SO per l'allocazione della memoria, e le strategie migliori.

\*Paginazione su richiesta (pure demand paging) è una strategia che risolve il problema di capire quali pagine allocare in fase di avvio del processo: inizialmente non ne viene caricata nessuna: il sistema affida al meccanismo del page fault il compito di allocare le pagine a cui il processo si riferisce. In questo modo avremo un picco di page fault frequency all'inizio dell'allocazione della memoria, che poi si stabilizzerà, è una strategia lazy, ma efficace.

Ad un processo vengono assegnate dei frame:

c'è un Minimo numero di frame che devono essere assegnati al processo, che dipende dal processo stesso e dall'architettura, poiché l'esecuzione di un'istruzione può interessare più pagine e se non ho abbastanza frame disponibili non posso eseguire l'istruzione. Le pagine generalmente sono:

- 1 per contenere l'istruzione.
- 1 per ogni operando.
- N per gli accessi indiretti alla memoria.

Potremmo anche assegnare il massimo numero di frame tramite la memoria libera. Nel caso il SO non possa offrire il minimo dei frame richiesti, si sospende il processo e si fa swapping su disco, rilasciando la memoria a lui dedicata.

### 3.8.1 Strategie e Problemi

Esistono diversi modi per suddividere le allocazioni di memoria tra i processi:

- Allocazione equa: Ogni processo ha N frame su cui può allocare memoria, se genera un page fault il frame da riallocare viene pescato tra quelli assegnatigli. Ovviamente il SO si riserva più memoria rispetto ai processi.
- Allocazione proporzionale: Più realisticamente al processo viene allocata memoria in proporzione alla sua taglia, cioè al processo  $i$  di dimensione  $s_i$  vengono assegnati  $a_i = s_i / S \times m$  frame (dove  $S$  è la dimensione dei vari processi e  $m$  è il numero di frame)
- Allocazione per priorità: Si favoriscono processi a priorità più elevata, così da rendere più efficiente l'esecuzione. In realtà mitigando l'allocazione proporzionale all'allocazione per priorità si ottiene un ottimo risultato di allocazione.

\*Per la scelta della pagina vittima posso scegliere o pagine locali (dello stesso processo, allocazione locale) o pagine appartenenti anche ad altri frame (allocazione globale). Nella stragrande maggioranza si usa l'allocazione globale, in questo modo il numero di page fault non dipenderà più dalla bontà dell'algoritmo o dalle pagine dedicate inizialmente, ma anche dalla probabilità che un altro processo "rubi" pagine fisiche.

\*Nel caso un processo disponga di pochi frame sopra il suo minimo indispensabile, entra in Trashing, cioè perde molto tempo a causa di eccessivi page fault, il tempo di overhead è maggiore del tempo effettivo di

lavoro e si rallenta moltissimo il comparto di I/O, coinvolgendo anche gli altri processi. Per risolverlo è necessario dargli più allocazioni di memoria, ma se non ho abbastanza frame per soddisfare questa richiesta l'unica soluzione è fare uno swap-out o del processo in trashing oppure di un processo vittima che ha una più bassa priorità.

Il problema sorge quando ho più processi in trashing, cioè quando un eccessivo grado di multiprogrammazione porta la stragrande maggioranza dei processi ad andare in trashing, generando un sovraccarico. A questo punto è necessario terminare qualche processo manualmente.

\*Per decidere quanta memoria allocare ad un processo ci si rifà al Principio di località: una porzione ben precisa di dati e di risorse che il processo necessita per essere eseguito correttamente in un preciso momento. È conveniente che una località si carichi interamente in memoria, così da minimizzare i page fault. Le località sono molte e possono essere condivise o cambiate: quando cambio località avrò un alto numero di page fault causati dal caricamento delle nuove pagine appartenenti alla località. Quando è necessario allocare i frame ad un processo bisogna adeguare il loro numero alla località attuale, cioè deve essere abbastanza grande per contenerli (meglio eccedere di un po' in quanto la località non ha sempre dimensione fissa).

### **3.8.2 Working Set (WS)**

Un tentativo di approssimare il concetto di località è il Working Set (WS) che corrisponde, in un dato istante, ad un insieme di pagine ben preciso. Fissato un parametro  $\Lambda$ , questo indica le pagine usate negli ultimi  $\Lambda$  accessi alla memoria. Se scelgo opportunamente  $\Lambda$ , il Working Set contiene il concetto di località del processo in maniera più adeguata.

Se riesco a calcolare la dimensione dell' $i$ -esimo working set (Working Set Size  $i$  (WSS $_i$ )) di tutti i processi, riesco a calcolare la richiesta globale di frame, che posso comparare con la memoria disponibile, rendendomi più facile allocare più frame o prevenire il trashing abbassando il grado di multiprogrammazione (facendo lo swap-out di qualche processo eccessivamente pretenzioso di frame).

Posso usare quindi la tecnica della prepaginazione, che consiste nel riportare in memoria il WS prima dello swap out, per poi essere ripristinato

più rapidamente, questo abbassa il numero di page fault. Un tipo simile di prepaginazione può avvenire all'avvio del processo, si raccolgono delle statistiche delle pagine (relative alle risorse e ai dati più che altro) tipicamente utilizzate dall'applicazione in fase di avvio, così da velocizzarne l'inizializzazione.

Il working set si può calcolare tramite:

- Interrupt periodici
- Bit di referenziamento R
- Log che conserva la storia del processo in base alle istruzioni effettuate nel tempo di  $\Delta$

\*La richiesta globale D di frame in un certo intervallo di tempo, può essere definita come somma delle richieste di frame di ogni processo e quindi

$$D = \sum |WS|(P_i), \text{ per } i \text{ che va da } 0 \text{ a } n.$$

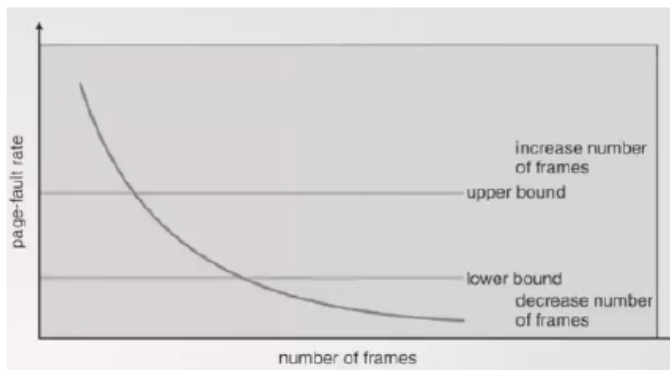
(Sommatoria della cardinalità del working set di ogni processo)

Se D è maggiore del numero di frame che abbiamo (memoria fisica), il sistema potrebbe essere in stato di trashing.

### 3.8.3 Page Fault Frequency

Quando il WS è totalmente in memoria avrò poche page fault, mentre se non è totalmente in memoria ne riscontrerò tanti. Quindi posso mettere in relazione il WS con il Page Fault Frequency (PFF): processi con bassa PFF (Lower Bound) possono cedere frame a processi con PFF più alta (Upper Bound), così da adeguare la quantità di frame alla dimensione del working set, e di fatto approssima esattamente ciò di cui abbiamo bisogno.

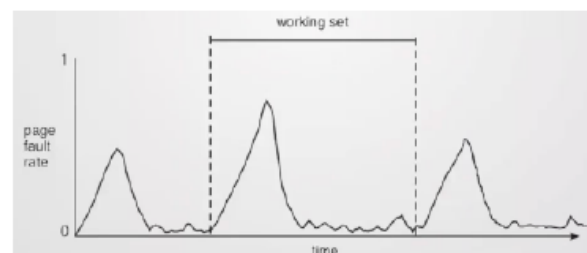
Ovviamente se ci sono troppi processi con PFF troppo alta questo modello non può fare miracoli. Working Set e PFF sono in relazione, infatti tramite la PFF posso individuare i working set del processo.



Abbiamo già visto nel contesto degli **algoritmi di sostituzione delle pagine** che, in generale, se aumento il numero di frame, il fenomeno page fault diminuisce come frequenza in cui si manifesta (avendo più memoria ci si aspetta che il fenomeno decrementa...).

Il sistema va a misurare, in un lasso di tempo abbastanza breve e definito, quanti sono il numero di page fault presenti. Si osserva che: **più è alto il numero di processi, più ampia**

**sarà l'esigenza in termini di numero di frame** (per far sì che il processo non vada in trash).



In cui i "picchi" non sono altro che l'innesco di una transizione tra un working set e il successivo.

### 3.8.4 Politica di Pulitura

Quando avviene un page fault, il caso migliore è trovare immediatamente un frame libero dove assegnare il nostro processo. Per far in modo di trovare questo caso spesso, si usa la strategia della Pulitura per mantenere sempre un numero pool minimo di frame liberi in memoria (devo far in modo di non dover ricorrere al cercare una pagina sporca da usare). In realtà sto anticipando il lavoro di liberazione di memoria di una pagina vittima (se è sporca si deve pulire prima sincronizzandola e poi marcandola come frame libero). Perché farlo? Perché riduce i tempi di reazione al page fault e posso applicarla tramite un Paging Daemon che entra in funzione nei momenti di I/O, cioè di scarso uso della CPU. Alcuni SO effettuano solo la sincronizzazione e non la liberazione del frame, anticipando del lavoro, ma rendendo meno efficiente la pulizia. Nel caso la pagina appena liberata sia oggetto di un page fault, può essere ripescata e ritorna subito disponibile senza necessità di I/O dato che materialmente i dati all' interno della frame non sono stati cancellati.

### 3.8.5 Dimensione della Pagina

Come si deve scegliere la dimensione della pagina e del frame? Questa è molto importante ed ha delle ripercussioni, può essere fissa o variabile (di pochi valori) ed è stabilita come taglia dai progettisti dell'HW.

Comunemente può essere scelto dal SO in un range tra 512Byte - 64Kbyte a oltre (sempre a potenze di due ovviamente). Quali sono i vantaggi?

- Pagine Grandi:

- \*La tabella delle pagine è più piccola: più sono grandi i frame, più memoria occupa ognuno di questi, meno elementi in tabella troverò;

- \*Migliora il modo in cui il sistema utilizza l'I/O: se lavoro con pagine di 1Kb e devo leggere questa pagina dal disco: le tempistiche dipendono dal seek time (tempo necessario per posizionare la testina nell'area di interesse, circa 20ms), dal latency time (tempo di rotazione del disco 6-8 ms) e dal transfer time (tempo di trasferimento dei dati 0.1ms per blocchi da 1K), l'unica in relazione con la dimensione della pagina è il transfer time: grandezze più grandi implicano trasferimenti più grandi. Pagine grandi fanno in modo che seek e latency time siano pagate meno volte per il trasferimento degli stessi dati rispetto a pagine più piccole;

- \*Al crescere della dimensione si riducono i page fault: questo perché se ho variazioni nella grandezza dei dati contenuti nella frame non ho problemi nel dover allocare più frame e quindi generare meno page fault.

- Pagine Piccole:

- \*C'è minore frammentazione interna: se ho bisogno di più memoria rispetto a quella contenuta in un frame ho bisogno di allocarne un altro, se alloco piccoli frame posso arrotondare lo spreco al minimo, se uso frame più grossi avrò sprechi più importanti;

- \*Migliore risoluzione nel definire il working set: spreco meno memoria dato che uso frame più piccoli, agevole così la multiprogrammazione.



### 3.8.6 Pagine Condivise

Vediamo come viene implementata la possibilità di andare a violare il modello di isolamento dei processi, cioè di condividere delle pagine:

- Sola lettura: Nel caso si istanzino due o più processi uguali, questi avranno in comune il codice (ed X), quindi è inutile istanziarlo più volte (codice rientrante). Questa soluzione è perfetta quando nessuno dei due processi ha necessità di effettuare scrittura.

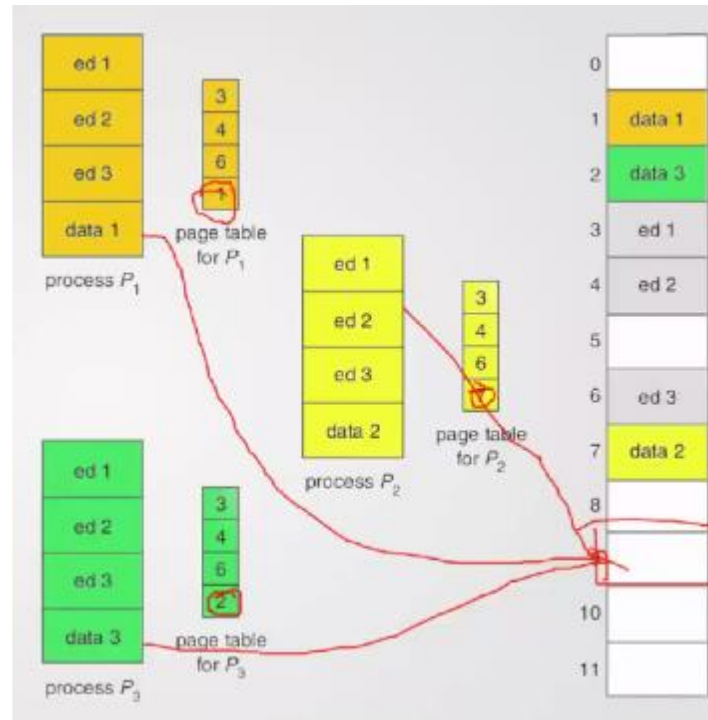
Nel caso dell' esempio le tabelle delle pagine dei processi P1 e P3 hanno gli stessi indirizzi nelle aree del codice (ed1, ed2, ed3), mentre puntano a pagine differenti nel caso dei dati (data1, data3). Questo "riutilizzo" di pagine consente un grande risparmio di memoria nei sistemi multiutente.

- Lettura/scrittura: Questo rappresenta una violazione del modello di isolamento dei processi, ma consente una comunicazione IPC. Quando un processo tenta una scrittura sui dati, la violazione della protezione Read Only causa una trap al SO, viene quindi fatta una copia della pagina cosicché ogni processo abbia la sua copia privata, quindi entrambe le copie sono impostate in Read-Write, così da non dare problemi in successive operazioni di scrittura. Facendo ciò non duplico tutte le pagine comuni, ma solo quelle che vengono man mano modificate.

\*Si può pensare anche ad un'implementazione che non necessita di duplicazioni, dove i due processi condividono i dati, anche modificandoli.

L'idea è che i processi possano interagire tramite segmenti di memoria condivisa, per scambiarsi dati. Nei due processi avrò due pagine che si riferiscono all'area di memoria condivisa, entrambi hanno i diritti di scrittura e lettura (race condition), se scrivo qualcosa nella pagina del processo, si aggiornerà anche l'area di memoria.

\*Implementazione su tabella delle pagine ordinaria o multilivello: semplice ed efficiente. \*Aliasing: tecnica che consiste nell'associare più indirizzi virtuali (In spazi di indirizzamento virtuali differenti) alla stessa pagina fisica.





### 3.8.6.1 Pagine Condivise Difficoltà

\* Gestione della cache: possono presentarsi problemi di sincronizzazione con cache basate su indirizzi virtuali (anche se usano gli ASID per lo stesso frame fisico). Potrebbe succedere che il processo P1, al momento in cui accede alla world N (quella condivisa), la cache porta la linea di cache che copre quella word al suo interno. In sostanza, d'ora in poi il processo P1 quando accederà a quella world non farà accesso alla memoria centrale, ma alla linea di cache che sta al suo interno. Supponendo che la cache passi al processo P2, poiché l'alias (il nome) di quel contenuto è diverso, la cache non si accorgerebbe del fatto che c'è già quel contenuto all'interno della cache stessa. In realtà lo tratterebbe come un cache miss: andrebbe a leggere all'interno della memoria centrale e andrebbe a popolare un'altra linea di cache (differente dalla precedente) al suo interno. La cache, poiché non ha individuato il fatto che si sta parlando della stessa word: all'interno della cache io avrò due linee di cache che riguardano lo stesso pezzo di memoria fisica. Questo è un serio problema, che va cercato di risolvere "a priori" tramite un utilizzo di cache con ricerca basata su indirizzi virtuali e tag fisici (possiamo anche disabilitare):

- la cache cerca in parallelo con la TLB sulla base dell'indirizzo virtuale.
- per capire se si tratta di un duplicato dobbiamo aspettare che la TLB dia in output l'indirizzo fisico.

\* Tabella delle pagine invertita: è una tabella globale che prevede tante entry quanti sono i frame e ne contiene delle informazioni. Nel caso di due processi che condividono memoria (P1 e P2) hanno sicuramente ASID diverso (uno è P1 e l'altro è P2) e anche indirizzo virtuale diverso, ma entrambe dovrebbero riferirsi alla medesima memoria, ciò che non può succedere dato che la tabella delle pagine invertita si riferisce all'ASID e indirizzo virtuale. Per risolvere il problema basta modificare le voci della entry cosicché se P1 sta lavorando si riferiscono a questo, ma quando si effettua un context switch si modificano in modo che si riferiscano a P2. È una toppa ma una buona soluzione. Questo approccio è impraticabile nei sistemi multicore: se P1 e P2 lavorano contemporaneamente sulla tabella da due core diversi, si avrebbero troppi context switch. Teoricamente sarebbe possibile risolvere il problema con tabelle con corrispondenze moti-a-uno, mai realizzate in realtà.

### **3.8.6.2 Copy-on-write e Zero-fill-on-demand**

Cosa succede quando effettuo una fork? Clono un processo (a meno dell'ID) e ne copio dati, codice, indirizzi, ecc... entrambi quindi condivideranno i frame nella memoria centrale, il problema sorge quando effettuerò una scrittura, altererò una word e quindi il sistema di condivisione del medesimo frame non funzionerà più, bisognerà trovare un nuovo frame, allocarlo e mapparla per far in modo che un processo non modifichi i dati dell'altro. Dobbiamo evitare che si creino delle copie superflue dello stesso processo. Le pagine che contengono il codice, vengono banalmente condivise in sola lettura, mentre quelle che contengono sezioni di stack e heap, che devono poter essere modificate, vengono condivise con la tecnica di Copy-on-write ed il processo nuovo viene allocato con il meccanismo zero-fill-on-demand.

\* Copy-on-write: I processi hanno diritti in sola lettura, quando cercano di scrivere si attiverà il meccanismo di copy-on-write che allocherà un nuovo frame personale del processo a cui andrà a puntare e ne verrà garantito il diritto di scrittura.

\* Zero-fill-on-demand: Quando un processo necessita di allocare nuova memoria è necessario allocarla al processo e mapparla su un frame che deve essere azzerato. Le pagine sono riferite ad un frame vuoto fisso ed in modalità lettura, appena qualcuna cerca di scrivere, le viene allocata una pagina dedicata solo a lei (read-only static zero page, come prima). Si può creare un pool di pagine libere e vuote pronte per essere allocate: questo permette di risparmiare risorse (si libera nei momenti di I/O).

### **3.8.7 Librerie Condivise (Linking) e File Mappati**

Le librerie sono una serie di procedure che si usano per risolvere problemi comuni ed unificano e condividono il codice tra più applicazioni.

\*Linking statico: In fase di compilazione includo il codice della libreria nel codice sorgente, non ha bisogno di elementi esterni e di caricare parti di librerie in corso di esecuzione. (poco usato). Il programma diventa "autonomo" e "autocontenuto". Creare programmi eseguibili più grossi comporta un maggiore storage su disco. Non permette al SO di risparmiare memoria RAM. I programmi A e B, se magari utilizzano lo stesso codice, ne utilizzano una porzione che è sostanzialmente ridondante.

\*Linking dinamico: Quando l'applicazione ha bisogno di una certa funzionalità, la libreria viene caricata in memoria. Ciò che avveniva in fase di compilazione ora avviene a runtime. Ciò si traduce in risparmio in memoria nel caso altre applicazioni usino medesime librerie DLL (Dynamic Linked Library) dato che ne tengo in memoria solo una copia. È possibile aggiornare la libreria, in questo caso è necessario specificare la versione della libreria a cui vogliamo accedere. Risparmio di spazio sul disco ed in RAM. Sviluppo indipendente e facilità di aggiornamento.

-- Lo Stab è pezzetto di codice molto piccolo, invocato quando la libreria viene richiesta. Ha lo scopo preciso di provocare il caricamento della libreria invocata.

\*Vi è anche la possibilità di mappare un file su una porzione dello spazio di indirizzamento virtuale, un modello alternativo di I/O su file: il SO offre una chiamata di sistema che mappa un file che si trova sul disco in un area di indirizzamento. Vengono mappate una serie di pagine virtuali consecutive che fanno corrispondere ai vari pezzetti del file. L'intero file, che potrebbe essere anche molto grosso, viene interamente mappato in qualche area dello spazio di indirizzamento virtuale. Le librerie condivise possono essere gestite tramite mappatura dei file.

Gestiscono automaticamente:

- librerie condivise;
- caricamento codice eseguibile;
- caricamento dei dati statici.

-- Il DSS corrisponde ad una porzione del file eseguibile che viene mappata all'interno dello spazio di indirizzamento virtuale. È un segmento che contiene i dati statici preesistenti nel codice. La mappatura avviene in sola lettura, non avviene nessuna modifica.

### **3.8.8 Allocazione Memoria Kernel (Buddy e Slab)**

Non è sempre possibile utilizzare paginazione per il Kernel. Bisogna cercare di avere frammentazione interna minima, o possibilmente nulla.

Memoria dei processi utente:

- paginata ma con frammentazione interna.

Memoria interna al kernel:

- miriamo a frammentazione interna minima o nulla;

- impossibilità di paginare l'allocazione in alcuni casi.

Quando il SO usa delle strutture dati deve allocare memoria, è necessario quindi adottare un sistema che sia veloce, stabile, senza sprechi o frammentazione.

\*Buddy System: Supporta le richieste di allocazione del SO limitando il meno possibile la frammentazione interna. L'idea è quello di usare segmenti di memoria (potenza di 2) che è possibile dividere a metà: ogni richiesta viene arrotondata alla potenza di 2 più vicina e viene allocato il segmento di memoria disponibile uguale all'arrotondamento.

\*Slab Allocation: Ho una cache per ogni struttura dati, che sia libera o occupata. Posso suddividere le richieste in "taglie".

Ogni cache è una collezione di slab.

Uno slab è un pezzo di memoria dedicato a contenere strutture dati di una certa grandezza, è contiguo con altri slab, deve essere un multiplo della dimensione di base della struttura dati e del frame

(esempio: struttura=3K, frame=4K -> slab=12K)

All'inizio uno slab è vuoto, appena è necessario si alloca il primo record libero e lo si alloca. Se si riempie lo stato di uno slab è possibile ricorrere a slab che appartengono alla stessa cache. L'allocazione e deallocazione è rapida, non c'è frammentazione interna né esterna.

### 3.8.9 Segmentazione

La segmentazione è una strategia simile alla paginazione che cerca di far coincidere il modello fisico con quello logico: la memoria è una collezione di segmenti che contengono oggetti di dimensione variabile.

Il processo corrisponde ad una collezione di segmenti, ognuno dei quali è dedicato e specializzato ad un dato contenuto. Possiamo quindi vedere un processo come una collezione di segmenti:

- asseconda la visione logica dell'utente (programmatore);
- un segmento per ogni oggetto: procedura, stack, libreria;

Implementazione:

\* indirizzi bidimensionali: composto da due coordinate: # segmento a cui si fa riferimento e offset, che indica di quanto bisogna spostarsi a partire dall'indirizzo del segmento;

\* tabella dei segmenti: # segmento -> (registri base, registri limite): è

un'allocazione contigua dei segmenti dei vari processi. È l'elemento che, per ogni segmento, ci dice dove si trova in memoria fisica e quanto occupa. Problema: un segmento può crescere o diminuire di dimensioni, questa tecnica presenta quindi frammentazione esterna; Soluzione: approccio misto (segmentazione & paginazione).

### **3.8.10.1 Segmentazione su Pentium Intel (32bit)**

E' interessante notare come in passato è stato proposto un modello utile per usare entrambi i modelli: paginazione e segmentazione.

All'occorrenza si potevano usare entrambi o uno per volta.

Usando segmentazione e paginazione, contemporaneamente:

- \* Segmentation Unit: che aveva il compito di linearizzare l'indirizzo (come visto in precedenza). Tramite una tabella dei segmenti, permette di trasformare l'indirizzo logico bidimensionale in un indirizzo logico lineare (monodimensionale).

- \* Paging Unit: svolge un po' il lavoro dell'MMU, ha il compito di trasformare l'indirizzo lineare (virtuale) in indirizzo fisico.

Qui la paging unit lavora a cascata, subito dopo la segmentation unit.

Questo permette di organizzare meglio la memoria virtuale evitando, con meccanismi come la paginazione, problemi di frammentazione o di altro tipo.

- circa 16.000 segmenti con indirizzamenti a 4GB (32 bit): locali e globali.

- 2 tipi di tabelle per i segmenti (con circa 8000 voci ciascuna):

- Local Descriptor Table (LDT): segmenti dei programmi. Una per ogni processo.

- Global Descriptor Table (GDT): segmenti del sistema operativo. Una per tutti i processi.

- 6 registri segmento: utili a capire a quale segmento di volta in volta ci si vuole riferire. Sono dei registri della CPU, che contengono l'identificativo di alcuni segmenti.

1 = CS (Code Segment): contiene identificativo del processo che conteneva gli indirizzi dei processi da eseguire.

0 = DS (Data Segment) per prelevare istruzioni generiche.

- 4 livelli di protezione: a partire dal kernel fino al livello utente.

L'indice del segmento ha la seguente struttura a 16bit, a cui si somma

l'offset, che ha teoricamente 32bit ma nella pratica dimensione minore. La somma dà un numero lineare a 32bit, passato alla Paging Unit. Overlap: intersezioni tra i processi, chiaramente non utile.

### **3.8.10.2 Paginazione su Pentium Intel (32bit)**

Qui la paginazione è gestita tramite l'utilizzo di tabelle delle pagine multilivello: la tabella di 1° livello ("page directory") contiene registri (generici) e, per ogni blocco, un vero e proprio indirizzo fisico della tabella di 2° livello ("page table") a cui si fa riferimento.

- possibilità di fare swapping delle tabelle delle pagine (solo il 2° livello).
- impiega memoria associativa (TLB).
- architettura x86-64: Evoluzione dell'architettura soprastante, con opportuni accorgimenti. Supporto a tabelle con 4 livelli e seg. limitata.

### **3.8.10.3 Gestione della Memoria su Linux**

L'idea è sempre quella di evitare che il SW venga "stravolto" dal SO che lo ospita, bisogna sempre far sì che il SO gestisca al meglio il SW.

1 sistema operativo vs. tante architetture:

- uso limitato della segmentazione (quasi sparito su x86-64);
- solo 2 livelli di protezione (kernel mode, user mode);

Tabella delle pagine fino a 4 livelli (dalla versione 2.6.11) e recente supporto a 5: bisogna cercare di evitare la complessità che le tabelle multilivello possono creare. Pertanto si limitava (inizialmente) a 4 quelli che sono i livelli della tabella delle pagine, successivamente si è arrivati (attualmente) a 5.

- i386 (32 bit): VM da 4 GB (3 GB in user-mode e 1 GB in kernel-mode);
  - x86-64 (64 bit): indirizzi virtuali effettivi da 48 bit (hack!)
- VM da 256 TB (128 TB user-mode e 128 TB in kernel-mode).

Struttura tipica dello spazio di indirizzamento di un processo:

- copy-on-write sulla pagina statica zero;
- file mappati per sezione codice e librerie condivise;
- allocazione dinamica dell'heap con chiamata di sistema brk() per aumentare la dimensione della heap;
- algoritmo sostituzione delle pagine: ibrido tra l'alg. dell'orologio e NRU;
- slab allocation per la memoria del kernel (dalla versione 2.20).

## 4. File System e Dischi

### 4.1 Introduzione

Problema di base: gestire grandi quantità di informazioni, in modo persistente e condiviso tra più processi. Modello astratto di gestione della memoria secondaria.

- implementazione mediante file e directory;
- i dettagli di gestione ed implementazione costituiscono il File system;
- esempi di dettagli:
  - nomenclatura: es. Linux è CaseSensitive, Windows no, oppure lunghezza massima (in termini di caratteri di ogni file).
  - tipi di file;
  - tipi di accesso;
  - metadati (o attributi): come ad esempio la dimensione del file, la data oraria dell'ultima modifica, appartenenza all'utente, ...
  - operazioni supportate sui file;
  - accesso condiviso ai file: i lock:
    - shared (lock condiviso) vs. exclusive (lock esclusivo):
      - Shared Lock: Vengono acquisiti quando si ha intenzione di leggere dal file. Possono essere acquisiti più shared lock contemporaneamente in quanto le letture sono compatibili tra loro. Esso non può essere acquisito quando è attivo un Exclusive Lock.
      - Exclusive Lock: Esso viene acquisito quando si intende scrivere sul file. Una volta acquisito da un processo, nessun altro può acquisirlo. Inoltre esso può essere acquisito solo se non ci sono Shared Lock attivi.
    - mandatory vs. advisory:
      - Mandatory (Windows): Un processo per lock ha una risorsa in maniera esclusiva, dopo, un processo y prova a fare un lock condiviso ma riceve un blocco da parte del SO, garantendo mutua esclusione.
      - Advisory (Linux): Non blocca immediatamente un processo che vuole accedere ad una risorsa lockata, ma "suggerisce" al processo che arriva dopo di "astenersi" dall'accedere alla risorsa.
- strutture dati per la gestione dei file: globale e per processo.

## 4.2 Struttura di un File System

Il disco, di base, non fa altro che memorizzare una sequenza di blocchi, in modo non volatile, richiamabile e con dimensioni (ogni blocco) di qualche kb (nei moderni SO anche fino a 32kb). Spesso quindi, un disco non viene organizzato in un blocco univoco di dati/metadati ma viene strutturato in partizioni, composto da “contenitori” in cui possiamo memorizzare i File System. Scopo di questa procedura:

1. da un punto di vista logico, l'organizzazione è meglio strutturata.
2. si potrebbe mirare a minore spreco, a un'ottimizzazione in termini di memoria.
3. potrei voler utilizzare file system diversi.
4. potrei voler utilizzare sistemi operativi diversi.

### 4.2.1 Master Boot Record (MBR)

E' un blocco del disco particolare, corrisponde al primo blocco di qualunque disco. Ha il compito di individuare la partizione attiva e di avviare il SO di quella partizione.

Ha dentro una serie di informazioni:

- tabella delle partizioni: serve per gestire il partizionamento del disco.

Inizialmente era pensato per gestire piccoli blocchi di dati, prevedeva infatti al più 4 voci (riferite a 4 partizioni) e 1 voce riferita a quale fosse la partizione attiva.

- boot record (o “boot block”): è un elemento “non volatile”, che ha lo scopo di gestire l'avvio del sistema tramite un codice (binario) eseguibile e contenuto all'interno della BIOS (definito dallo standard ESI).

Nota: ogni architettura è tendenzialmente diversa ma basata su questo modello. In Linux e molti altri sistemi operativi (la maggior parte), non esiste in realtà un “boot record” unico: il più utilizzato è sicuramente il “grub”. Questo permette di caricare la stragrande maggioranza dei sistemi operativi.

- superblocco: contiene, anche in questo caso, un piccolo pezzetto di codice. Ha il compito di rendere disponibili informazioni come il tipo del file system utilizzato in quella partizione, identificato dal cosiddetto “magic byte” (un valore binario che identifica il tipo del file system). È importante perché in generale il SO va difatti a guardare prima l'utility del superblocco



per capire che tipo di file system ha davanti e poi opera in base al file system che ha letto (anche capendo se è supportato o no).

Note:

- L'MBR, al giorno d'oggi, è difatti non più utilizzato. Con l'avvento di uno standard, noto come GPT (GUID Partition Table), si ha la possibilità di effettuare più partizioni di dischi (partizioni estese) di dimensioni molto più grandi. Il tutto risulta quindi molto più efficiente.
- le allocazioni di partizioni possono essere fatte per cluster: che può essere della dimensione di un blocco, o multiplo dello stesso.

## **4.3 Implementazione dei File**

Come memorizzare un generico file A di dimensioni X su disco?

L'idea è quella di "spezzettare" il contenuto in unità, sequenze di byte.

Come? Utilizzando un certo numero di blocchi.

File: sono la comparte astratta più importante del FS. Ha dimensione dinamica (la sua dimensione aumenta e decrementa nel tempo). Si possono implementare in varie strategie.

### **4.3.1 Allocazione Contigua [alloc. sequenziale]**

Al giorno d'oggi è una strategia quasi completamente abbandonata, se non per i dischi ottici (CD, DVD). Questo perché i dischi ottici hanno solitamente il FS in sola lettura (una volta masterizzati non è più possibile modificarne il contenuto), dove quindi l'allocazione contigua ha senso per rendere la lettura veloce.

PRO:

- semplicità nella gestione dei file.
- accesso sequenziale (1) e diretto (2):
  1. prevede, sapendo dove si trova il primo blocco, si possono leggere i blocchi successivi banalmente dal fatto che sono allocati in modo contiguo.
  2. prevede, una volta individuato da dove inizia il file, di calcolare in modo diretto dove si trova l'iesimo blocco del file.
- nel caso dei dischi elettromeccanici: essendo che i blocchi di memoria sono contigui tra di loro, viene consentito un maggiore risparmio in termini proprio di "spostamento" della testina sul disco.

Nota: questo vantaggio, nei dischi elettronici, in realtà questo vantaggio

non è presente. Anche se, tuttavia, i dischi elettromeccanici sono comunque tutt'ora ampiamente utilizzati. Causa frammentazione interna, spreco, quindi gli stessi problemi che abbiamo già visto in questo caso.

#### **4.3.2 Allocazione con Liste Collegate [alloc. concatenata]**

Sono presenti vari blocchi, varie liste, l'una collegata logicamente all'altra. Strategia pessima, difatti mai utilizzata.

PRO:

- ci siamo sdoganati dall'allocazione contigua: se vogliamo inserire, rimuovere o modificare un dato blocco non abbiamo problemi sullo spostamento che il blocco subirà all'interno del disco. Questo perché le liste sono collegate logicamente tra di loro, ma non sono contigue.
- non è presente frammentazione esterna: qualsiasi buco può essere sfruttato per inserire qualsiasi quantità di dato che ci interessa.

CONTRO:

- spesso i file fanno operazioni per potenze di 2: questo può causare una sorta "disallineamento" dovuto al fatto che, se un file non dovesse ricadere per intero in un blocco, andrebbe nel successivo. Questo potrebbe causare anche fenomeni di spreco, per questo motivo, è difatti una forma di allocazione non molto utilizzata.
- problema legato alle prestazioni: accesso diretto non presente, per leggere il blocco *i*-esimo devo accedere a quelli precedenti. Spreco ingente di operazioni di I/O molto grande.

#### **4.3.3 Allocazione con Liste Collegate su una tabella di allocazione dei file (File Allocation Table – FAT) [alloc. tabellare]**

L'idea è quella di implementare (anziché dei blocchi) una tabella di metadati che avrà il solo scopo di tenere traccia dei collegamenti logici (precedente, successore) tra i vari blocchi. Conoscendo il primo blocco, si può ricostruire semplicemente la sequenza dei blocchi allocati a quel determinato file. È una struttura dati presente su disco (non volatile), composto da un vettore di interi. Si hanno tante voci quanti sono i blocchi allocabili all'interno della partizione gestita dal file system.

L'*i*-esima voce della tabella FAT indica qual è il blocco successivo al

blocco i-esimo: è infatti una collezione di puntatori a next. È una struttura dati che permette di controllare, di volta in volta, il successore di un blocco.

PRO: non vi è spreco.

CONTRO: diventa difficilmente gestibile nel caso in cui si volessero utilizzare dischi della grandezza di gb/tb. La tabella, in questo caso, potrebbe pesare anche centinaia di mb, il che chiaramente potrebbe diventare un problema per il fatto di mantenerla sempre in memoria.

#### 4.3.4 Allocazione con Nodi Indice [alloc. indicizzata]

Composto da 2 elementi:

- Metadati: serie di informazioni collegate all'entità file: grandezza del file, chi è l'utente proprietario, i vari timestamp (ultimo accesso e modifica).

NON è però presente il nome del file.

- Directory: in cui è contenuto il nome del file e il numero di collegamenti fisici che referenziano l'i-node stesso.

### ESERCIZIO

*Facendo uso di una lista concatenata, che ha blocchi da 4kb e un numero di blocco di 32bit, quanti blocchi occorrono per memorizzare un file da 40kb?*

Un blocco della lista è di 4KB, ovvero  $4 \cdot 2^{10}$  byte.

Ogni blocco della lista concatenata ha un solo puntatore che specifica il numero del blocco successivo.

Questo numero è di 32bit, ovvero di 4byte.

Dunque ogni blocco della lista ha uno spazio per i dati di:

$(4 \cdot 2^{10} - 4) \text{ byte} = (4 \cdot 1024 - 4) \text{ byte} = (4096 - 4) \text{ byte} = 4092 \text{ byte}.$

Il file è di 40KB, quindi  $(40 \cdot 2^{10}) \text{ byte} = 40960 \text{ byte}.$

La dimensione del file diviso lo spazio disponibile per ogni blocco è:  $40960 / 4092 = 10,01.$

Sono serviti poco più di 10 blocchi, cioè 11 blocchi.

#### 4.3.4.1 Allocazione con I-Node

È l'approccio UNIX. Anche esso è un File System basato sull'allocazione non contigua. In genere l'i-node contiene l'identificativo, le informazioni del file ed una decina di voci che si riferiscono ai blocchi. Questa soluzione va bene per blocchi piccoli, ma per file troppo grandi ci vuole una alternativa. Vi sono diverse strategie:

\* Strategia Collegata:

Utilizzo le 8 voci dell'inode per caricare il file, ed utilizzo l'ultima voce per riferirsi ad un blocco aggiuntivo (il blocco permette di mantenere molti più

dati rispetto le voci del inode). Il blocco aggiuntivo potrebbe essere collegato ad un altro blocco, e così via ricorsivamente. Se il file è piccolo è una soluzione che va bene, ma se il file è pesante non è comodo, perché si rischierebbe di avere una lista di blocchi troppo lunga.

\* Strategia Multilivello:

Ogni voce dell'inode punta ad un blocco. Si potrebbero avere più livelli. L'accesso ai dati è veloce, poiché l'accesso all'i-esimo blocco avviene senza bisogno di scorrere una lista, ma basta ricercare la foglia desiderata (perché la ricerca della lista è  $O(n)$ , mentre la ricerca dell'albero è  $O(\log n)$ ).

Approccio utile per file grandi ma pesante da utilizzare per file piccoli.

- Strategia Ibrida:

Soluzione ibrida tra i due precedenti, basato su una struttura ad albero, gerarchica. Prende gli aspetti positivi da entrambe le strategie. Ne è un esempio il file system V7 di Unix.

#### **4.3.4.2 Allocazione con I-Node nei File System UNIX (V7)**

È basato su i-node a 13 voci.

- le prime 10 voci individuano indirizzi diretti (strategia collegata), ottimo per i file piccoli.

- le ultime 3 voci estendono le prime 10:

se necessario, estendo le 10 voci con la 11esima, che aggiunge un blocco indiretto singolo (blocco pieno di indirizzi diretti). Se questa lista viene riempita (quindi si utilizzano  $(10 + 256) \cdot \text{dim}$ ), si utilizza la 12esima voce.

La 12esima voce è un indirizzo indiretto doppio, che punta ad un blocco pieno di indirizzi indiretti singoli (ovvero blocchi simili alla 11esima voce).

Se questa lista viene riempita (quindi si utilizzano  $(10 + 256 + 256^2) \cdot \text{dim}$  del blocco), si utilizza la 13esima voce.

La 13esima voce è un indirizzo indiretto triplo, che punta ad un blocco pieno di indirizzi indiretti doppi, i quali puntano a indirizzi indiretti singoli....

In questa maniera si possono utilizzare fino a  $(10 + 256 + 256^2 + 256^3) \cdot \text{dim}$  del blocco, il che è un upper bound per il numero di letture che bisognerà fare (al più 4 letture).

Il design originale si ferma a 3 livelli, poiché gestisce file molto grossi, in maniera dinamica e perfettamente adattata alla crescita del file.

Recentemente si utilizzano liste di extent, ovvero liste di sequenze di

blocchi di lunghezza variabile.

Ideale per file molto grossi (ES video editing), è utile perché si vuole avere spezzoni lunghi e contigui di file

Approccio utilizzato attualmente.

- extent: modo per cercare di memorizzare sequenze contigue di blocchi, cercando di mantenere, anziché una lista blocco per blocco, una lista di extent. L'idea è quella di usare le stesse strutture usate fino ad adesso, ma con liste di blocchi di lunghezza variabile.

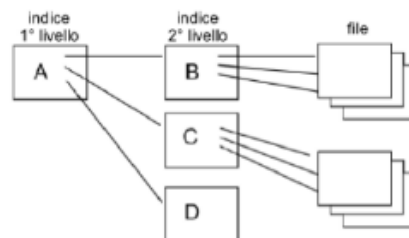
## ESERCIZIO

Supponiamo di utilizzare un file system UNIX basato su i-node particolari che contengono i seguenti campi: 12 indirizzi diretti a blocchi di dati, 1 indirizzo ad un blocco indiretto singolo e 1 indirizzo ad un blocco indiretto doppio. Supponendo di avere numeri di blocchi a 32 bit e blocchi su disco da 1 kb, indicare esattamente la dimensione massima (in kb) supportata da un simile i-node. Esplicitare il calcolo utilizzato.

L'indirizzamento indiretto ha blocchi da 1kb pieno di indirizzi a 32 bit, quindi sono 12 blocchi di indirizzo diretto + 256 di indiretto +  $256^2$  di indiretto doppio = 65804 blocchi.

## ESERCIZIO

Illustrare il meccanismo di allocazione indicizzata dei file. Si faccia riferimento ad un file system con indici a due livelli, dimensione del blocco logico pari a 512 byte e indirizzi di 4 byte. Come si alloca un file di 1 Mb? Quanto blocchi servono? Come si accede al suo 400° blocco? Come si accede al byte 236.448? Qual è la dimensione massima di un file? Quanti blocchi occupa complessivamente?



L'allocazione indicizzata a due livelli segue questo schema:

Se il blocco logico è di 512 byte e gli indirizzi di 4 byte si hanno  $512 / 4 = 128$  indirizzi per blocco.

Un file di 1 Mbyte occupa  $1M / 512 = 2K = 2048$  blocchi ( $1M = 1024 \times 1024$ )

2048 blocchi richiedono 2048 indici =  $2048 / 128 = 16$  blocchi di 2° livello.

Complessivamente il file occupa  $2048 + 16 + 1 = 2065$  blocchi.

Il blocco n. 400 è indicizzato dal blocco indice di 2° livello n.  $400 / 128 = 3$ , l'indirizzo è il  $400 \bmod 128 = 16$ -esimo del blocco.

In generale, l'n-esimo indirizzo occupa i byte da  $(n - 1) \times 4$  a  $(n - 1) \times 4 + 3$ .

Il blocco indice di secondo livello a sua volta è indicizzato dal 4° indirizzo (indirizzo n. 3) dell'indice di 1° livello, che si trova nei byte 12-15 del blocco.

L'accesso al byte 236.448 si risolve così: il byte occupa il blocco  $236.448 / 512 = 461$  (contando da 0, quindi è il 462-esimo), e si trova all'offset  $236.448 \bmod 512 = 416$ . Trovato il blocco si procede come sopra.

La dimensione massima di un file è data dal numero massimo di indirizzi utilizzabili per indicizzarlo.

Nell'indice di 1° livello ci sono al massimo 128 puntatori, quindi ci sono al massimo 128 blocchi indice di 2° livello, ciascuno dei quali contiene 128 puntatori ai blocchi del file, per un totale di  $128 \times 128 = 16384$  puntatori, e quindi blocchi del file. Il file può avere una dimensione massima di  $16384 \times 512 = 8$  Mbyte (8.388.608 byte), e occupa complessivamente  $16384 + 128 + 1 = 16513$  blocchi

## 4.4 Implementazione delle Directory

Directory: è una collezione di riferimenti a file.

Dove memorizzare i metadati/attributi?

Nei FS con i-node la directory contiene solo il nome del file e l'inode.

Nei FS che non usano inode la directory contiene anche tutti i metadati ed il numero del primo blocco.

La lunghezza dei nomi dei file è dinamica, il limite massimo ad oggi è circa 256 caratteri ma potrebbero anche essere un paio: allocare lo spazio massimo sarebbe uno spreco.

\*Una possibile soluzione potrebbe essere di immaginare la directory come una sequenza di entry con dimensione fissa (lunghezza della entry, attributi/metadati del file) e la parte che contiene il nome di dimensione variabile. Il nome viene allineato alla word, ci sarà quindi uno piccolo spreco, trascurabile. Il problema è che la directory ha una struttura dinamica: i file potrebbero crescere, essere aggiunti, tolti..

\*Un approccio migliore si ha posizionando le parti fisse della directory nella parte alta, mentre la parte dinamica (nome) viene salvato in una heap, che può crescere dinamicamente, può essere ricompattata ed è più rapida.

Generalmente si esegue ricerca lineare tra questi record. Nei sistemi moderni si usano meccanismi di caching e ogni directory viene composta da una tabella hash piuttosto che una normale tabella, così da velocizzare la ricerca.

## 4.5 Condivisione di File su un File System (Hard Link e Soft Link)

- usando una FAT:

duplicare la lista con i riferimenti ai blocchi + problemi in caso di append;

- usando i-node:

\*hard-link: contatore dei link + anomalia con accounting;

Un hardlink è invece un oggetto del FS che punta ad un i-node. Esso è quindi un file, ed in generale avere più hardlink che puntano ad uno stesso i-node corrisponde ad avere più copie (sincronizzate) dello stesso file.

Queste copie vengono fatte in tempo costante proprio perchè consistono in un semplice, ulteriore puntatore allo stesso i-node.

Ogni i-node mantiene un conteggio del numero di hardlink che puntano ad esso, e infatti esso viene eliminato se questo contatore arriva a 0 in seguito

alle chiamate “unlink()” per ogni hardlink che esisteva. Non è possibile creare un hardlink per una directory, e non è possibile creare un hardlink a cavallo di due diversi FS, in tal caso sarà necessario eseguire una vera e propria copia.

\*soft-link: universale e permettere di fare riferimenti al di fuori del file system + appesantimento nella gestione;

Un link soft (sym-simbolico) è un file, contenente all'interno un collegamento ad un altro file (o directory). Esso può essere creato anche a cavallo di due FS. Se viene eliminato il file o la directory originale, esso diventa inconsistente.

--eventuali problemi in fase di attraversamenti e backup.

## 4.6 Gestione Blocchi Liberi

- Bitmap (allocazione contigua): in cui ogni singolo bit rappresenta un blocco allocabile all'interno del file. In particolare: 0=blocco allocabile, 1=blocco allocato. L'idea è quella di cercare più blocchi allocabili possibili (setti a “0”), per cercare di gestire meglio gli spazi.

PRO e CONTRO sono gli stessi riscontrati dell'allocazione contigua.

Strategie di allocazione in memoria:

- tutta in memoria;
- un blocco alla volta: paginata con VM.

È l'approccio difatti più utilizzato;

- Liste concatenate: richiede più spazio, ma si sfruttano gli stessi blocchi liberi.

PRO:

- lo spazio ha una dimensione variabile: a differenza della bitmap che viene allocata in maniera fissa qui lo spazio viene allocato in maniera dinamica.
- possibilità di inserire contatori per blocchi contigui.

CONTRO:

- l'ordinamento dei blocchi risulta difficile: proprio per la dinamicità della struttura, il posizionamento e il riferimento risulta un po' più complicato (in termini di calcolo).
- altri contro soliti dell'allocazione con liste.

## 4.7 Controlli di Consistenza

E' opportuno saper gestire al meglio queste strutture dati, in caso di situazioni non poco frequenti e non di certo rare, come ad esempio l'esecuzione di semplici macrooperazioni (come l'inserimento, la modifica, l'eliminazione di un file nel blocco di allocazione). A seguito di crash del sistema i file-system possono diventare inconsistenti: in generale, lavorare con blocchi di dati inconsistenti è una pessima idea.

\*Apposite utility possono effettuare dei controlli di consistenza:

- sui blocchi: l'idea è quella di avere due liste inizialmente vuote

1. "block in use" che viene aggiornato a "1" ogni volta che viene incontrato il blocco i-esimo, a "0" altrimenti. Se il blocco supera il valore "1", vuol dire che più file stanno utilizzando lo stesso blocco di memoria, il che non è un bene. Per costruzione nelle bitmap, un numero di blocco può essere incontrato una sola volta.

2. "free blocks", vale l'opposto di 1. In questo caso potrebbero comunque presentarsi dei problemi di inconsistenza, ma "meglio gestibili".

\*Nell'immagine "d", ad esempio, sono 2 i file che utilizzano lo stesso blocco di memoria. Qui l'idea potrebbe essere quella di:

1) cercare un nuovo blocco libero (ad esempio il blocco 15).

2) duplicare il contenuto del blocco 5 nel blocco 15.

3) far sì che un file utilizzi il blocco 5 e l'altro la copia del blocco 5 (nel blocco 15). Tipicamente in questi casi l'utility permette di dire che il file X o Y è stato danneggiato.

\* sui riferimenti agli i-node: quando un i-node è allocato, è citato in una directory (può essere citato più volte lo stesso i-node in directory diverse, in caso di hard link, ma vi è un contatore).

In caso di errore in tal

caso, dovrei

scansionare il

numero di i-node

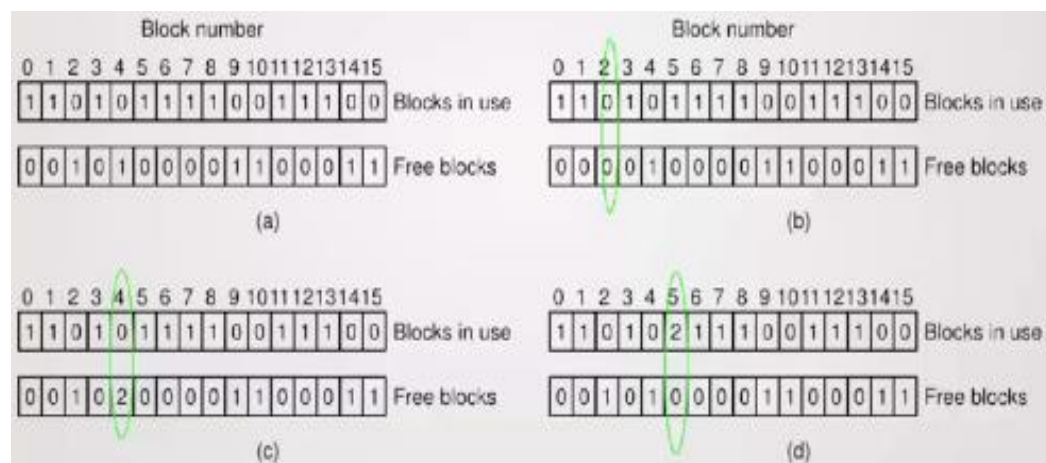
all'interno delle

directory e

controllare nei singoli

i-node (il contatore

deve essere uguale).





### 4.7.1 Journaling

L'idea qui è quella di avere un "grado di resistenza" maggiore di fronte a fenomeni di crash. Le macro-operazioni devono essere pensate come un ordine ben preciso di micro-operazioni. Si ha quindi l'obiettivo di perdere meno informazioni possibili.

Strategia: utilizziamo un Log, una sorta di "diario di bordo" (non volatile) che tiene conto di ciò che si sta facendo sul file system in quel momento. L'idea è quella per cui, in caso di ripristino da crash le operazioni possono essere "ripescate" dal log, in modo da non avere (o avere, ma in minima parte) perdita di informazioni. Affinché tutto funzioni bisogna operare con operazioni idempotenti: le operazioni devono essere fatte in modo tale che, reiterate, non fanno cambiare il risultato.

Situazioni:

- quando il file system non è in uso: il log è vuoto.
- esecuzione: l'idea è quella di appuntare tutte le varie operazioni che il file system sta per effettuare prima ancora di eseguirle.
- a posteriori: viene poi ripulito.

PRO:

- maggiore robustezza dei metadati;
- veloce ripristino da crash/reboot;

Implementazione:

- tramite un file stesso all'interno del file system.
- (unix) esterna al file system: il journal stava all'interno di una partizione separata (non più utilizzato).

## 4.8 Cache sul Disco

Per migliorare le prestazioni dei dischi si fa spesso uso di una cache sul disco (o buffer cache), struttura basata su tabelle hash, dove mantenere alcuni file in memoria.

Implementazione:

\* LRU modificata: lista doppiamente linkata, vengono ordinati i blocchi in modo da avere più a destra il blocco più utilizzato (MRU) e a sinistra il meno utilizzato (LRU), il più ideale da scartare. Ogni qualvolta che un blocco viene letto viene spostato più a destra (aumenta il suo peso). Si utilizza poi una tabella hash che ad ogni coppia (disco, numero di blocco) contiene una

lista linkata (sottoinsieme della lista doppiamente linkata generale). In questa lista ci sono tutti i blocchi che collidono con la stessa ricerca hash.

- read-ahead: (specialmente per dischi elettromeccanici) consiste nel leggere i blocchi successivi al blocco interessato. Teoricamente implicherebbe un overhead ma, essendo un disco elettromeccanico, la testina si trova comunque lì: se nel futuro il blocco successivo avrà bisogno di essere letto, sarà già in cache e questa “scommessa” avrà portato vantaggi. Si spera, in soldoni, che i file siano allocati in maniera contigua su disco.

- free-behind: quando un blocco viene scritto (referenziato) viene portato a destra nella cache, e vi starà per un determinato quanto di tempo; consiste nello spostare in memoria secondaria la parte LRU. Cioè quando un blocco che sta in cache viene scritto, lo levo dalla cache o quantomeno lo metto in posizione meno favorevole per la ricerca. E' difficile che serva di nuovo ciò che è appena stato scritto.

Anche in questo caso c'è scrittura sincrona e asincrona.

\*Scrittura:

- sincrona: non si ritardano le scritture ma vi è maggiore spreco di tempo nell'I/O. - asincrona: scrittura che non avviene in maniera immediata, minore overhead però.

#### **4.8.1 Altre tecniche per migliorare le prestazioni**

Tecniche di allocazione che permettono la riduzione dei movimenti del braccio del disco:

\* scelta di “blocchi vicini” nell'allocazione di un file.

I dischi elettromeccanici sono suddivisi in cilindri. La testina si muove tra i cilindri, la rotazione permette invece di leggere/scrivere in un cilindro.

Idea di base: Le operazioni da eseguire nell'apertura di un File sono legate alla lettura dell'i-Node corrispondente al file. L'utilizzo del file implica la lettura dei blocchi spesso in modo sequenziale, partendo dal primo.

\* posizionamento degli i-node: l'idea è quella di posizionarli il più al centro possibile, in modo tale da essere più velocemente accessibili.

Un certo numero di i-Node vengono preallocati e popolati quando serve allocare file. Gli i-Node preallocati sono posizionati nel cilindro di centro,

quindi la testina deve spostarsi al massimo di  $\text{\#numCilindri}/2$ . Gli i-node vengono organizzati in gruppi: ogni gruppo è composto da una serie di i-Node e blocchi dati. In un cilindro vengono inseriti diversi gruppi. Viene mantenuta una bitMap di i-Node liberi per ogni gruppo. Per allocare, cercherò un i-Node libero e leggerò ad esso quelli dello stesso gruppo. Se non ne trovo, cerco nei gruppi vicini. Questa tecnica non garantisce contiguità ma garantisce vicinanza, cosa che velocizza notevolmente lettura e scrittura dei File, e ha permesso a Linux di essere un sistema che non ha mai avuto bisogno di deframmentazione.

PRO:

- non richiede deframmentazione: nei sistemi Linux particolarmente.
- accesso più veloce.

## 4.9 File System in Linux (V7)

\*V7 È il primo file system unix, è basato su i-node. In particolare qui sono presenti i cosiddetti “gruppi di blocchi”, equivalenti ai cilindri visti sopra. In particolare, ogni gruppo di blocchi è formato da:

- un superblocco: da una serie di informazioni su quel gruppo (quanti elementi ci sono, dove iniziano le varie parti, ...)
- i-node: sono preallocati e organizzati in un “i-node bitmap”. Ogni gruppo di blocchi ha un gruppetto di i-node.
- blocchi di dati: organizzata anche questa in una “block bitmap”, che da informazioni su se sono allocati i blocchi o meno (come la bitmap vista a inizio lezione).

Il numero di accessi è limitato anche per file molto grandi. C’era la limitazione sulla partizione, sulla lunghezza del nome del file. Il numero di i-node era un intero a 16 bit.

Man mano si è evoluto passando da v7 a ext, a ext2, a ext3, fino a ext4.

\*Con ext2 si è introdotta la possibilità di aumentare i caratteri del file, estesi i limiti sulla dimensione del file, si è evoluta la strategia di allocazione: i blocchi sono suddivisi in gruppi, ogni gruppo ha il suo i-node. Si tende ad allocare i blocchi tutti vicini. (come detto prima). Nella creazione di un nuovo file, oltre alla ricerca di un i-node disponibile, si ricerca spazio nel gruppo. Ogni gruppo ha una bitmap per tener conto dei blocchi liberi. L’ideale sarebbe conoscere a priori la dimensione del file e

allocarlo contigualmente. Per cercare di perseguire questo obiettivo (pur non conoscendo a priori la dimensione del file) si cercano tipicamente 8 blocchi liberi contigui e si inizia ad allocare lì. Il sistema alloca più spazio del necessario nella speranza che il file al massimo li utilizzi tutti ma non di più.

\*Con ext3 si mantiene la retrocompatibilità con ext2 e si introduce il journaling.

\*Con ext4 gli i-node sono lenti per file molto grossi, per ovviare a ciò si affiancano gli extent, un segmento contiguo di blocchi allocati appresso al file, questi funzionano appieno se il file è contiguo.

Introduce l'allocatore multiblocco, se il processo richiede un grande numero di blocchi, inizialmente si allocavano man mano, col multiallocatore si ricerca un'area libera del disco e si alloca tramite extent (deve essere specificato). Con l'allocazione ritardata cerca di allocare i blocchi in maniera ritardata nel tempo: quando un processo richiede blocchi per usarli, il SO apparentemente soddisfa la richiesta, ma in realtà usa il buffer cache per ritardare la scrittura e l'allocazione, in questo modo li scrive o quando scade il timer dei 30 sec per mantenere i dati integri oppure quando finisce il processo e così sceglie un'area contigua più precisa possibile. Si può fare una deframmentazione on-line cioè mentre il file system sta ancora lavorando (utile nei server).

\*\*Il BTRFS (B-Tree FS o Butter FS) è il futuro dei file system. È del tutto nuovo, è retrocompatibile ed è rivoluzionario.

-L'idea fondamentale è l'uso del copy-on-write, si ha la possibilità di clonare un file o un gruppo di questi (cartelle o volumi) in particolare permette di prendere un intero disco (o più) e creare un pullBTRFS da cui posso creare volumi (distribuito su più dischi) che posso clonare a loro volta (snapshot) creandone uno storico. Occupa lo spazio solo necessario ed è basato sui B-Tree.

-È possibile utilizzare le primitive send/receive sui volumi.

-Il file System sa le differenze tra i vari snapshot, se il disco si guasta si perde tutto: ciò per cui si usa è copiare i dati in un altro disco.

-Si può inviare tramite la rete un'istantanea del volume, quando ne creo un'altra e devo backupparla non invio l'intera istantanea, ma grazie a questa primitiva capisco le differenze tra gli snapshot ed invio solo le differenze risparmiando lavoro e traffico in rete ecc.

- Si possono effettuare operazioni pesanti anche on-line (deframmentazione, aggiunta nuovo disco, rimozione vecchio, ecc..).
- Tutti i dati prevedono un checksum (CRC) per avere una ridondanza dei dati per accorgersi se si sono subite manipolazioni o presenza di errori.
- Permette di comprimere trasparentemente i file.
- Tramite RAID aggrega più dischi in un pull.

## ESERCIZIO

*In un disco con blocchi di 2 Kbyte ( $= 2^{11}$  byte), è definito un file system FAT 16. Ogni elemento ha lunghezza di 2 byte e indirizza un blocco del disco. La copia permanente della FAT risiede nel disco a partire dal blocco di indice 0 e una copia di lavoro viene caricata in memoria principale all'avviamento del sistema operativo. Supponendo che la FAT sia dimensionata in base alla massima capacità di indirizzamento dei suoi elementi si chiede:*

- 1) il numero di blocchi dati indirizzabili dalla FAT.
- 2) il numero di byte e di blocchi del disco occupati dalla FAT,
- 3) l'indice del primo blocco dati nel disco,
- 4) quale capacità (in blocchi e in byte deve avere il disco) per contenere tutti i blocchi dati indirizzabili.

1. Il numero di blocchi dati indirizzabili dalla FAT è  $2^{16}$
2. La FAT ha  $2^{16}$  elementi di 2 byte pertanto occupa  $2^{17}$  byte è  $2^6 = 64$  blocchi
3. Il primo blocco dati nel disco è quello di indice 64 (i blocchi precedenti sono riservati alla FAT)
4. Per contenere tutti i blocchi indirizzabili, il disco deve avere una capacità di almeno  $2^{16} + 26$  blocchi e  $227 + 217$  byte.

## 4.10 File System in MS-DOS (FAT)

Nato come MS-DOS si è evoluto nella famiglia FAT, è tuttora utilizzato nei sistemi windows ed è consigliato sulle memorie flash, macchine fotografiche, sistemi embemmed, smartphone, ecc. per la sua compatibilità e fruibilità data la diffusione.

Noto il numero di blocco iniziale, riesco a seguire la catena per identificare qualunque blocco appartenente al file.

- FAT-12: 12 è la capienza del numero di blocco, influiva sulla dimensione del blocco che poteva supportare.
- FAT-16: Utilizza blocchi più grandi, la singola voce è 16 bit.
- FAT-32: Penultima versione, in cui le voci fisiche sfruttavano 28 bit su 32, il limite è 2TB, questo perché le dimensioni della partizione dei master boot record non possono descrivere partizioni superiori ai 2TB.
- FAT-64, exFAT: Supportata dalle ultime versioni di Windows, è un'ulteriore evoluzione della FAT, mette pezze su tanti aspetti. È protetta da brevetti, quindi non è utilizzabile su Linux. È sconsigliato per file system interni, usato per memorie flash e HD esterni.

## **4.10 File System NTFS**

È uno dei file system più evoluti e complicati sul mercato, è nato con Windows-NT. Un Volume corrisponde a una partizione o più partizioni aggregate. Un Cluster è l'unità più piccola leggibile e scrivibile sul disco da parte del SO.

La Master File Table (MFT) è una tabella che contiene la lista dei file sul disco, è composta da record di dimensione fissa.

Un FILE è una collezione di più attributi di varia natura, possono essere metadati, flussi di dati, ecc... e non necessita di un nome. Gli Attributi del File possono essere contenuti nello stesso record (residenti) o in record diversi (non residenti).

Le directory sono basate sulla struttura dati B+ Tree per ricerche efficienti. La gestione blocchi liberi è basata su bitmap (visione d'insieme sullo stato di allocazione).

Supporta hard-link, soft-link e montaggio di altri File system (nelle ultime versioni). Supporta il journaling; Può fare compressione e cifratura selettiva dei file, di recente anche la cifratura a livello di volume.

Posso fare copie shadow di volumi con tecnica copy-on-write (ridondanza dei dati e ripristino di file vecchi).

## **4.12 Scheduling del disco**

Una tecnica per ottimizzare i movimenti elettromeccanici della testina è lo scheduling del movimento di questa sul disco. L'obiettivo è il massimizzare il numero di richieste soddisfatte nell'unità di tempo e il minimizzare il tempo di accesso. Il SO può ignorare dove si trovi il blocco da leggere, ma può richiedere dove si trovi fisicamente la testina per massimizzare la lettura. Il SO gestisce una coda di richieste che possono essere esaudite una alla volta. Man mano che il controller si libera ne esaudisce un'altra. Il SO può modificare la coda avendone una visione globale in modo da ottimizzare i costi di posizionamento.

### 4.12.1 Ottimizzare il Seek-time

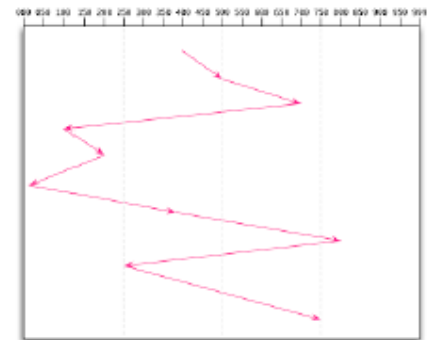
Seek-time: tempo necessario per spostare il blocco delle testine da un dato “cilindro” a un altro.

#### FCFS (First Come First Served)

Come dice il nome vengono servite le richieste nell'ordine in cui arrivano.

Nel nostro esempio l'ordine di chiamata è: 500, 700, 100, 200, 10, 390, 800, 250, 750. Quindi:  $(500-400) + (700-500) + (700-100) + (200-100) + (200-10) + (390-10) + (800-390) + (800-250) + (750-250) = 3030$ . Il tempo per soddisfare le richieste è: 3030ms

Tale algoritmo è equo, semplice da realizzare ma inefficiente.



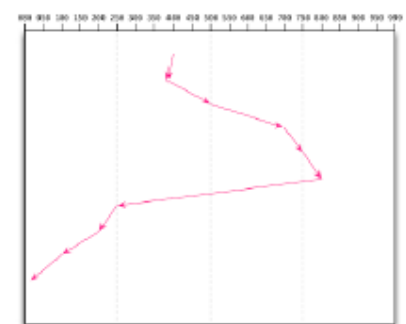
#### SSTF (Shortest Seek Time First)

Questo algoritmo di scheduling sceglie la richiesta con seek time minore rispetto alla posizione corrente (prima soddisfa quelle vicine alla posizione iniziale della testina sul cilindro).

*Nota:* Per comodità riordiniamo la lista delle richieste [10, 100, 200, 250, 390, 500, 700, 750, 800]. Nel nostro esempio l'ordine di chiamata è: 390, 500, 700, 750, 800, 250, 200, 100, 10. Il tempo per soddisfare le richieste è:  $10+110+200+50+50+550+50+100+90 = 1210$ ms

Tale algoritmo ha buone prestazioni (non ottimali) ma non è equo (rischio starvation).

SSTF (Shortest Seek Time First)



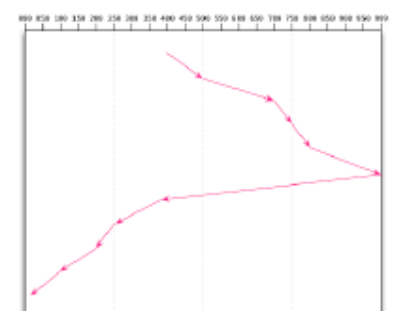
#### SCAN (scheduling per scansione)

L'algoritmo SCAN serve in una direzione e quando arriva al bordo del disco cambia direzione servendo i rimanenti elementi (detto anche algoritmo dell'ascensore).

Nel nostro esempio l'ordine di chiamata è: 500, 700, 750, 800, (999), 390, 250, 200, 100, 10. Il tempo per soddisfare le richieste è: 1588ms

Tale algoritmo garantisce comunque una attesa massima per ogni richiesta, ma si può fare di meglio.

SCAN



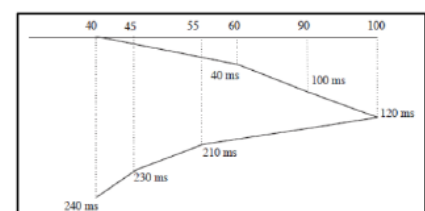
## ESERCIZIO

(Es. con tempi di arrivo diversi) Si consideri un disco con un intervallo di tracce da 0 a 100, gestito con politica SCAN. Inizialmente la testina è posizionata sul cilindro 40; lo spostamento ad una traccia adiacente richiede 2 ms. Al driver di tale disco arrivano richieste per i cilindri 90, 45, 40, 60, 55, rispettivamente agli istanti 0 ms, 20 ms, 30 ms, 40 ms, 80 ms. Si trascuri il tempo di latenza. (1) In quale ordine vengono servite le richieste? (2) Il tempo di attesa di una richiesta è il tempo che intercorre dal momento in cui è sottoposta al driver a quando viene effettivamente servita. Qual è il tempo di attesa medio per le cinque richieste in oggetto?

Assumendo una direzione di movimento ascendente all'istante 0, le richieste vengono servite nell'ordine 60, 90, 55, 45, 40

Il tempo di attesa medio per le cinque richieste in oggetto è

$$\frac{(40-40)+(100-0)+(210-80)+(230-20)+(240-30)}{5} = \frac{0+100+130+210+210}{5} = 130 \text{ ms}$$



#### C-SCAN (scheduling per scansione circolare)

La testina segue le richieste in ordine (da una parte o dall'altra, in genere si procede seguendo un ordine crescente) finché non arriva all'estremo del disco. Una volta arrivato all'estremo del disco torna all'inizio (senza servire richieste) e riparte. Considera le posizioni come collegate in modo circolare, e arrivato alla fine del disco torna sul primo cilindro senza servire alcuna richiesta.

Nel nostro esempio l'ordine di chiamata è: 500, 700, 750, 800, (999), (0), 10, 100, 200, 250, 390. Il tempo per soddisfare le richieste è: 989ms.

Tale algoritmo garantisce un tempo medio di attesa più uniforme.

#### LOOK

Il funzionamento è identico all'algoritmo di SCAN. L'unica differenza è che con questo algoritmo non è necessario raggiungere i bordi del disco.

Nel nostro esempio l'ordine di chiamata è: 500, 700, 750, 800, 390, 250, 200, 100, 10. Il tempo per soddisfare le richieste è: 1190ms.

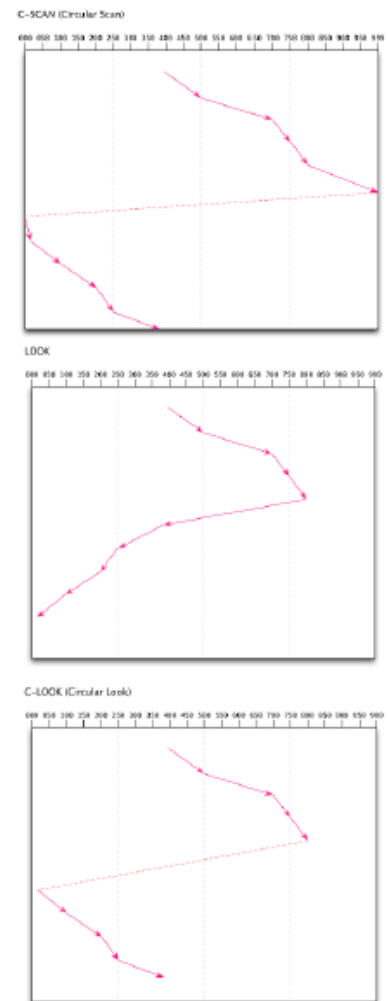
#### C-LOOK (Circular LOOK)

Questo algoritmo è una variante dell'algoritmo C-SCAN. Con questo algoritmo non è necessario arrivare fino ai bordi del disco.

Nel nostro esempio l'ordine di chiamata è: 500, 700, 750, 800, (790), 10, 100, 200, 250, 390. Il tempo per soddisfare le richieste è: 790ms.

Come scegliere?

- C-LOOK per alto carico;
- LOOK o SSTF per basso carico.



## 4.13 Sistemi RAID (Redundant Array of Inexpensive Disks)

Sono sistemi che sfruttano l'esistenza di più dischi (minimo di 2 dischi, anche 4 o 5). È un'astrazione che consente di suddividere i dati relativi ad una unità logica su più dischi, allo scopo di:

- migliorare le prestazioni.
- migliorare la resistenza ai guasti.

Un altro modo per aumentare le prestazioni è sfruttare il parallelismo anche per l'I/O su disco:

- servono più dischi indipendenti;
- striping: si suddividono i dati relativi ad una unità logica (un file, in generale un volume) su più dischi, ovvero compie l'astrazione.

L'idea è quella di "spezzettare" (con una strategia Round-Robin) un file in ogni disco fisico presente in memoria. Questo viene fatto perché, se i dischi hanno la possibilità di leggere contemporaneamente il file, allora lo possono anche caricare contemporaneamente, risparmiando notevolmente



in termini di caricamento.

PROBLEMA: aumenta la probabilità che si verifichi un guasto sul volume logico RAID;

- soluzione: aggiungiamo ridondanza per ottenere migliore affidabilità;
- sostituzione automatica: dischi spare;

Implementazione:

- RAID 0 (striping): suddivide semplicemente i file su più dischi, “apprezzati” molto nei sistemi server, dove operazioni di I/O sono molto frequenti.

Vantaggi:

- Nella lettura di una serie di strip ho una grande probabilità di sfruttare il parallelismo ed aumentare le prestazioni.
- Maggiore capienza in quanto la dimensione del volume logico corrisponde alla somma delle dimensioni dei dischi fisici.

Svantaggi:

- Più dischi ho più aumento la vulnerabilità, aumentando il numero di dischi che possono guastarsi.
- Nessuna ridondanza

- RAID 1 (mirroring): i file vengono anche duplicati (mirroring) sui vari dischi. L’idea è quella di avere una “copia di tutto”. Si devono aggiungere un numero di dischi pari a quelli che si avevano nel RAID-0, che verranno usati per ricopiare tutti gli strip. Se avevamo 4 dischi, abbiamo bisogno di 8 dischi fisici in totale. Il Sistema riallinea in automatico eventuali dischi sostituiti ricopiando i dati.

Vantaggi:

- Letture ancora più agevolate: Nel caso peggiore con il RAID-0 una lettura poteva ricadere tutta in un disco. In questo modo ne abbiamo di sicuro almeno 2 contenenti gli stessi dati.
- Fault Tolerance: E’ garantita di per sé dal mirroring. Se si rompe un disco perdo la Fault tolerance per quel disco fin quando non ne viene riallineato uno che sostituisca quello guasto.

Svantaggi:

- Troppi dischi: La dimensione del volume logico è la metà della somma dei

dischi fisici.

-overHead per la scrittura alto: dobbiamo sempre ricopiare dati, anche se viene fatto in parallelo.

- RAID 2 (striping a livello di bit con ECC):

Si fa striping in modo diverso: ogni strip corrisponde ad un singolo bit e non ad una sequenza di blocchi. Aggiungo bit di ridondanza alle informazioni usando un codice di correzione degli errori ECC, in questo caso il Codice di Hamming: presi 4 bit ad esempio, vengono aggiunti 3 bit di ridondanza. I 7 bit totali vengono divisi nei diversi dischi fisici.

Vantaggi:

-Ottima Fault Tolerance: Supponendo di avere 7 dischi, se si rompe un disco perdo un bit su 7. Posso ricostruire il bit con Hamming. Risparmio un disco rispetto a Raid-1 grazie ad Hamming.

Svantaggi:

-Ancora troppi dischi: La dimensione del volume logico è pari alla somma delle dimensioni di 4 dischi fisici, i restanti 3 sono usati per Hamming.

-E' necessario sincronizzare la rotazione dei dischi.

-Striping a livello di bit oneroso se non gestito in Hardware.

- RAID 3 (striping a livello di bit con bit di parità):

Anche qui si fa striping a livello di bit. Piuttosto che usare Hamming, usiamo un bit di parità. Si dedica un disco ai bit di parità, e tutti gli altri dischi come archiviazione effettiva.

Osservazione: il bit di parità non permette solitamente di capire dove si è creato l'errore, ma solo di capire che una sequenza di bit è corrotta. In questo caso però, se si rompe un disco sappiamo già dove si trova l'errore proprio perché possiamo vedere quale disco si è fisicamente guastato.

Vantaggi:

-Stessa fault tolerance di RAID-2

-Molti meno dischi: Con soli 5 dischi posso ottenere la stessa archiviazione di un RAID-2 con 7 dischi.

Svantaggi:

-Ancora necessaria la sincronizzazione della rotazione

-Fare striping a livello di bit è comunque pesante se non gestito da hardware

- RAID 4 (striping a livello di blocchi con XOR sull'ultimo disco):

Facciamo un RAID-0 e aggiungiamo un disco per la parità. Facciamo striping a livello di blocchi e non di bit (RAID-0). Non abbiamo più un bit di parità ma un blocco di parità, costruito facendo lo XOR degli strip della stessa riga degli altri dischi. Se si guasta un disco posso ricostruire gli strip di ogni riga.

Vantaggi:

- basato su stripe a blocchi, evitiamo di dover fare striping a livello di bit;
- disco extra = XOR degli stripe.
- non necessita di sincronizzazione.
- ottima fault tolerance.

Svantaggi:

- Modifiche molto complesse: Anche se modifichiamo un solo strip dobbiamo ricalcolare tutto il blocco di parità. Per velocizzare possiamo calcolare il nuovo blocco usando il vecchio blocco di parità e lo strip sovrascritto.

## ESERCIZIO

*Dati i seguenti dischi:*

Disco 1	Disco 2	Disco 3
0010	1001	101x
1001	0101	110x
0110	0100	001x

*Trovare i bit di parità x*

x = 011 dato che con lo xor risultano tali valori

- RAID 5 (striping a livello di blocchi con informazioni di parità distribuite)  
Lavora come RAID 4, ma anziché tenere le informazioni di parità in un solo disco, qui vengono distribuite tra i vari dischi, in modo da alleggerire il peso di I/O sull'ultimo disco bilanciando la probabilità di guasti.

## ESERCIZIO

*Dati i seguenti dischi:*

101	010	110	100	110
100	111	100	010	110
110	100	110	-	111
111	010	001	101	1100

*Trovare gli stripe del disco con il simbolo -.*

Per ricostruire il blocco mancante, devo fare solamente lo XOR fra gli elementi delle terza (3a) riga; quindi:

$$111 \text{ XOR } 110 = 001 \text{ XOR } 100 = 101 \text{ XOR } 110 = 011$$

## 4.14 Memorie Flash e File System

Le memorie flash basate su porte NAND lavorano più efficacemente ma hanno delle limitazioni sul numero massimo di cancellazioni. Essendo che bisogna cancellare completamente dei blocchi prima di poterci riscrivere sopra, anche le scritture sono limitate. Riscrivere sempre una stessa zona porta più facilmente ai guasti. Se si vuole modificare una pagina, bisogna cancellare l'intero blocco e riscriverlo. Inoltre ogni blocco ha un limite di cancellazioni che può fare nel suo arco di vita.

I classici File System non sono pensati per gestire problemi del genere. Ad esempio se pensiamo alla FAT, la sua tabella è una tabella fortemente utilizzata, e quindi è una di quelle zone che in una memoria del genere si guasterebbe subito.

Sono nati dei File System appositi, come F2FS (Flash Friendly File System). Dove vi è l'uso di controller che gestiscono il disaccoppiamento tra volume logico e fisico. Tenendo conto del numero di cancellazioni per zona, il volume logico resta invariato all'utente ma viene rimappato rispetto a quello fisico, utilizzando altre zone del disco. Infatti il sistema di mappatura, che si occupa di decidere dove posizionare ogni blocco, anziché sovrascrivere un vecchio contenuto ne crea uno nuovo e quello vecchio è "disabilitato", ovvero temporaneamente non visibile e successivamente verrà richiamato un Garbage Collector che avrà il compito di ripulirlo e ricompattare il disco.

Il GC mantiene il numero di cancellazioni più uniformemente possibile, velocizzando anche le operazioni di lettura e scrittura, evitando spesso di eseguire le cancellazioni, ma marcando dei blocchi come "spazzatura" e scrivendo i dati altrove. Il degrado delle prestazioni potrebbe arrivare nei vecchi file system perché il controller non era informato dei blocchi che non erano più in uso. Ad oggi c'è l'operazione TRIM, che ha il compito di comunicare al controller quali blocchi devono venire cancellati e non devono essere presenti nel file system.