



Cos'è un sistema operativo? APPUNTI S.O. 21/22

Dominick Ferraro

- Per uno studente: software che consente l'accesso a internet.
- Per un programmatore: software che rende possibile sviluppare programmi.
- Per utente di pacchetti applicativi: SW che rende possibile l'uso del pacchetto

Ognuno può percepire il S.O. in maniera diversa.

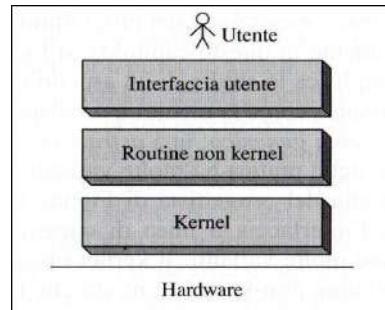
Un sistema operativo è un insieme di routine che semplificano l'esecuzione di programmi e l'uso di risorse.

Gli obiettivi principali di un S.O. sono:

1. **La convenienza per l'utente:** ambiente che soddisfa le necessità degli utenti. Un buon S.O. deve essere conveniente per chi lo usa (veloce, interfaccia grafica intelligente, connessione web, nuove funzioni...)
2. **L'uso efficiente:** gli amministratori si aspettano che il SO sfrutti al meglio le risorse hardware a disposizione in modo da avere buone prestazioni.
3. **L'assenza di interferenze:** i sistemi possono essere usati da più utenti contemporaneamente, è fondamentale avere meccanismi che consentono di non creare interferenze tra utenti: ciò avviene allocando risorse per ogni utente.
- 3.1. **Accesso illegale ai file:** garantire la protezione e l'accesso a un file o una directory.

VISTE DI UN SISTEMA OPERATIVO

La definizione di SO dipende dal tipo di utilizzo che ne fa l'utente.
Un progettista di SO ha una sua visione astratta del sistema operativo stesso:



- **Interfaccia utente:** questa accetta comandi per eseguire programmi e usa le risorse fornite dal SO.
- **Routine non di sistema:** queste implementano i comandi utente relativi all'esecuzione dei programmi e all'uso delle risorse del computer.
- **Kernel:** questo è il cuore del SO, controlla il pc e fornisce un insieme di funzioni e servizi per utilizzare la CPU.

Funzioni di un SO

- Principali:
 - Gestione dei programmi
 - Gestione delle risorse
 - Sicurezza e protezione

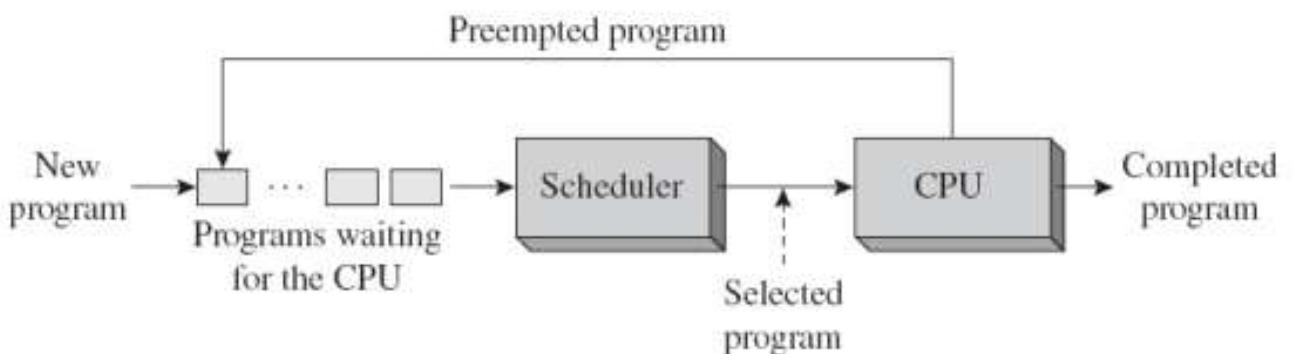
TASK	QUANDO SUCCIDE
Costruire una lista di risorse	Durante la fase di boot
Memorizzare le info per la sicurezza	Mentre vengono registrati nuovi utenti
Verificare l'identità di un utente	Al momento del login
Esecuzione dei programmi	Quando richiesto dall'utente
Mantenere le informazioni di autorizzazione	Quando un utente specifica che i suoi collaboratori

	possono avere accesso ai suoi programmi o dati
Preservare lo stato corrente dei programmi ed effettuare lo scheduling	In maniera continuativa durante l'esecuzione del SO

GESTIONE DEI PROGRAMMI

Le CPU moderne permettono di eseguire più programmi contemporaneamente, la funzione chiave per ottenere ciò è lo **scheduling**, che decide di volta in volta a quale programma deve essere concesso l'uso della CPU.

Lo Scheduler mantiene una lista dei programmi in attesa di essere eseguiti dalla CPU e ne seleziona uno per l'esecuzione. Inoltre specifica anche per quanto tempo potrà utilizzare la CPU. Al termine di questo tempo, il SO sottrae al programma la CPU e la concede ad un altro programma. Questa azione prende il nome di **prelazione**. Un programma che perde la CPU a causa della prelazione ritorna nella lista dei programmi in attesa di essere eseguiti dalla CPU.



Gestione delle risorse

L'allocazione e la de-allocazione possono essere realizzate con una tabella delle risorse. Questa tabella viene generata durante la fase di booting e rimane nel tempo.

Nome della risorsa	Classe	Indirizzo	Stato dell'allocazione
stampante1	Stampante	101	Allocata a P ₁
stampante2	Stampante	102	Libera
stampante3	Stampante	103	Libera
disco1	Disco	201	Allocata a P ₁
disco2	Disco	202	Allocata a P ₂
cdw1	CD writer	301	Libera

STRATEGIA PER L'ALLOCAZIONE DELLE RISORSE PIU' COMUNI

1) **Partizionamento delle risorse:** il SO a priori decide quali risorse allocare ad ogni programma. Una partizione è una collezione di risorse.
Questa è una tecnica facile da implementare ma poco flessibile.

2) **POOL-BASED:** il SO alloca le risorse da un insieme di risorse: consulta la tabella e alloca la risorsa se è libera.

RISORSE VIRTUALI

Una risorsa virtuale è una risorsa fittizia, creata dal SO attraverso l'uso di risorse reali. Un SO può usare la stessa risorsa reale per supportare diverse risorse virtuali. In questo modo può dare l'impressione di avere un numero di risorse maggiore di quelle disponibili.

Ad esempio, avendo a disposizione un solo hard disk, lo si potrebbe partizionare in due parti, avendo così l'impressione di averne due.

Sistemi operativi lezione 2

SO, COMPUTER E PROGRAMMI UTENTI

Richiamo di concetti di architettura di un calcolatore.

Le funzioni svolte da un sistema operativo sono elaborate dalla CPU. La CPU si alterna in primis nell'esecuzione di programmi utente e laddove è necessario eseguire controlli li esegue.

Dobbiamo capire la relazione tra il sistema operativo e i programmi degli utenti. Quando la CPU esegue un programma utente si dice che sta eseguendo un **PROCESSO**. Quando sta facendo ciò e poi la CPU deve essere data a un altro processo, si dice che la CPU **commuta** dalle istruzioni di un processo a un altro processo, si dice che avviene un “**cambio di contesto**”. La CPU si alterna nei processi, la maggior parte sono processi utente e a volte routine del kernel. Qualsiasi cosa la CPU sta facendo, deve essere salvata e deve salvare tutti i processi che svolge.

PRINCIPI FONDAMENTALI

Il kernel de SO è l' un insieme di routine che costituiscono il nucleo del SO. Il kernel svolge diverse funzioni:

1. Implementa funzioni di controllo
2. Fornisce servizi per i programmi utente
3. Si trova in memoria durante il funzionamento di un SO

Un'interruzione(**interrupt**) dirotta la CPU verso l'esecuzione di codice kernel. Meccanismo che notifica al kernel l'occorrenza di un evento.

Un interrupt software è usato dai programmi per comunicare le loro richieste al kernel. Ad esempio l'operazione di scrittura e apertura di un file.

Il kernel deve assicurare che non ci siano mutue interferenze fra i programmi degli utenti e tra un programma utente e il SO. È un aspetto fondamentale per garantire un corretto funzionamento di tutta la macchina. Per prevenire queste interferenze, la CPU ha due modalità operative:

1. **Modalità kernel:** in quel momento la CPU può eseguire tutte le istruzioni. Il kernel opera con la CPU in tale modalità in modo da poter controllare le operazioni del computer
 2. **Modalità utente:** la CPU non può eseguire le istruzioni che possono interferire con altri programmi o con il SO se usata in modo indiscriminato. CPU lavora in tale modalità per eseguire i programmi utenti.
-

IL COMPUTER

Le componenti principali sono la CPU, la memoria (intesa come RAM) e i dispositivi di I/O.

La memoria e la CPU sono direttamente connessi ad un BUS dove fluiscono tutte le informazioni memorizzate e caricate. A questo bus sono controllati i dispositivi I/O collegati tramite un DMA (Direct Memory Access).

LA CPU

Ai programmi utenti o al SO sono visibili due caratteristiche della CPU:

- Registri general-purpose(GPR):
 - Chiamati anche registri accessibili ai programmi
 - Mantengono dati, indirizzi, valori...
- Registri di controllo:
 - Contengono le informazioni che controllano le operazioni della CPU.
 - L'insieme dei registri vengono detti PSW (program status word)
 - Ciascun registro è visto come un campo della PSW

CAMPI PRINCIPALI DEL PSW:

Program counter (PC)	Condition code (CC)	Modalità (M)	Informazioni di protezione della memoria (MPI)	Maschera degli interrupt (IM)	Codice interrupt (IC)
Campo	Descrizione				
Program counter	Contiene l'indirizzo della prossima istruzione da eseguire.				
Condition code (flag)	Indica alcune caratteristiche del risultato di un'istruzione aritmetica (< 0 , $= 0$, > 0). Questo codice viene utilizzato nell'istruzione di salto condizionato.				
Modalità	Indica se la CPU è in esecuzione in modalità kernel o in modalità utente. Si assume un campo di un solo bit con valore 0 per indicare che la CPU è in modalità kernel e 1 per indicare che è in modalità utente.				
Informazioni di protezione della memoria	Informazioni relative alla protezione della memoria del processo in esecuzione. Questo campo è composto di sottocampi che contengono il registro base e il registro limite.				
Maschera degli interrupt	Indica quali interrupt sono abilitati e quali sono "mascherati".				
Codice interrupt	Describe la condizione o l'evento che ha causato l'ultimo interrupt. Questo codice è utilizzato da una routine di servizio dell'interrupt.				

Il campo modalità (M) del PSW contiene 0 se la CPU è nella modalità kernel (privilegiata) e 1 se è nella modalità utente.

Se la CPU opera in:

- Modalità kernel: può eseguire istruzioni privilegiate.
 - Modalità utente: non esegue istruzioni privilegiate.

STATO DELLA CPU

GPR e PSW contengono le info necessarie per sapere cosa sta facendo la CPU.

Il kernel salva lo stato della CPU quando toglie la CPU ad un programma in esecuzione. Quando il programma deve riprendere, ricarica lo stato salvato dalla CPU nei GPR e PSW

Esempio: Stato della CPU (cont.)

Address	Instruction			PSW	PC CC M
	MOVE	A, ALPHA			MPI IM IC
0142	COMPARE	A, 1			
0150	BEQ	NEXT			1 A
	...				
0192	NEXT			Registers	B
	...				X
0210	ALPHA	DCL_CONST	1		SP

- (a) Listato di un programma *assembly* che mostra l'indirizzo associato ad ogni istruzione o dato.
 - (b) Stato della CPU dopo l'esecuzione dell'istruzione **COMPARE**.

UNITA' DI GESTIONE DELLA MEMORIA (MMU)

La memoria è un'illusione di una memoria più grande di una memoria della memoria reale di un computer.

È implementata usando l'allocazione memoria non contigue l'MMU.

La CPU passa l'indirizzo di un dato o dell'istruzione, usato nell'istruzione corrente, alla MMU.

- È chiamato indirizzo logico.

La MMU traduce l'indirizzo logico in un indirizzo fisico.

GERARCHIA DELLA MEMORIA

La gerarchia della memoria fornisce una memoria ampia e veloce, a basso prezzo.

È una organizzazione di più memorie con diverse velocità di accesso e dimensioni.

- La CPU accede solo alla memoria più veloce, ovvero la cache.
- Se un byte richiesto non è presente nella memoria a cui si sta accedendo, esso viene caricato da una memoria più lenta
- Il tempo di accesso effettivo dipende da quante volte si verifica tale situazione in una memoria più veloce.

Più la memoria è piccola, più è veloce e più è costosa.

Un esempio di memoria gerarchica è quello costituito da:

- memoria cache, piccola e veloce
- memoria RAM, più capiente ma più lenta della cache
- hard disk, è la memoria più lenta ma anche quella più capiente

Quando la CPU esegue una ricerca nella cache, può verificarsi un successo (hit) o un fallimento (miss)

- **Lo hit ratio (h)** della cache è la frazione di byte acceduti dalla CPU che comportano un successo della ricerca nella cache.
- **Tempo accesso effettivo (gerarchia con cache e memoria)**

$$\begin{aligned}t_{\text{ema}} &= h \times t_{\text{cache}} + (1 - h) \times (t_{\text{tra}} + t_{\text{cache}}) \\&= t_{\text{cache}} + (1 - h) \times t_{\text{tra}}\end{aligned}$$

Dove

t_{ema} = tempo di accesso alla memoria effettivo,

t_{cache} = tempo di accesso alla cache, e

t_{tra} = tempo impiegato per trasferire un blocco della cache dalla memoria alla cache

Il funzionamento della memoria è analogo al funzionamento della cache.

Sono trasferiti blocchi di byte (pagine) dal disco alla memoria o viceversa.

Ma la gestione della memoria ed il trasferimento dei blocchi tra memoria e disco sono eseguiti via SW, nella cache il trasferimento è eseguito dallo HW.

- La gerarchia di memoria che comprende MMU, memoria e il disco è detta memoria virtuale.

GERARCHIA DI MEMORIA

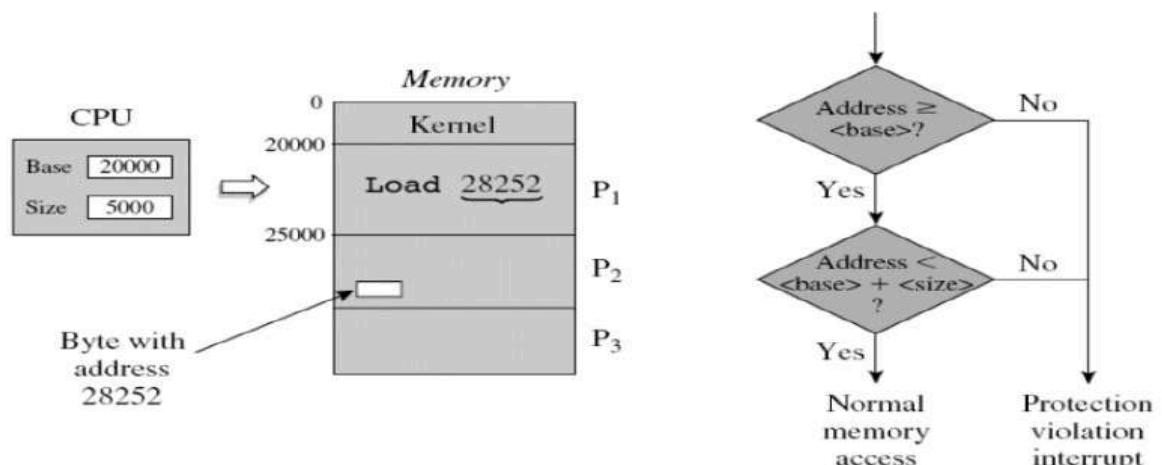
La protezione della memoria è implementata controllando se un indirizzo di memoria usato da un programma si trova fuori dall'area di memoria ad esso allocata.

Registri di controlli usati:

- Base e size (chiamato anche limite): indirizzo del primo byte= <base>. Indirizzo dell'ultimo byte=<base>+<size> -1

Esempio: Fondamenti di Protezione della Memoria

- L'esecuzione di una istruzione `load` causa una violazione della protezione



INPUT/OUTPUT

Modalità di I/O	Descrizione
I/O programmato	Il trasferimento dati tra la periferica di I/O e la memoria avviene attraverso la CPU. La CPU non può eseguire nessun'altra istruzione mentre è in esecuzione un'operazione di I/O.
Interrupt di I/O	La CPU è libera di eseguire altre istruzioni dopo aver eseguito l'istruzione di I/O. Un interrupt viene generato quando un byte di dati deve essere trasferito dalla periferica di I/O alla memoria e la CPU esegue la routine di servizio dell'interrupt che gestisce il trasferimento del byte. Questa sequenza di operazioni viene ripetuta finché tutti i byte sono trasferiti.
I/O basato sul direct memory access (DMA)	Il trasferimento di dati tra la periferica di I/O e la memoria avviene direttamente sul bus. La CPU non è coinvolta nel trasferimento dei dati. Il controller DMA genera un interrupt quando il trasferimento di tutti i byte è stato completato.

INTERRUZIONI

Un evento è una situazione che richiede l'attenzione del SO. Il progettista del computer associa un interrupt ad ogni evento. Lo scopo è di informare il SO dell'occorrenza dell'evento, permettendogli di eseguire delle azioni per la sua gestione. L'azione dell'interrupt salva lo stato della CPU e carica i nuovi contenuti nei PSW e GPR, la CPU inizia ad eseguire istruzioni di una routine di servizio delle interruzioni nel kernel. Ad ogni interrupt è associata una priorità. Se si verificano più interrupt nello stesso istante, la CPU servirà l'interrupt con priorità maggiore, gli altri interrupt restano PENDENTI fino al momento in cui sono selezionati per essere serviti.

Classe	Descrizione
I/O interrupt	Causato da condizioni come il completamento dell'I/O e il malfunzionamento delle periferiche di I/O.
Timer interrupt	Generato a intervalli di tempo o quando è trascorso uno specifico intervallo di tempo.
Program interrupt	(1) Causato da condizioni di eccezioni che si verificano durante l'esecuzione di una istruzione, per esempio eccezioni aritmetiche come overflow, eccezioni di indirizzamento e violazioni di protezione della memoria. (2) Causato dall'esecuzione di una speciale istruzione chiamata <i>istruzione di interrupt software</i> , il cui unico scopo è di generare un interrupt.

CODICE INTERRUZIONE

Quando si verifica un interrupt di un tipo, l'HW imposta un codice d'interrupt nel campo IC del PSW:

- Indica che si è verificato un interrupt di quel tipo
- Utile per conoscerne la causa

I codici dipendono dall'architettura:

- L'interrupt I/O: codice → indirizzo periferica
- L'istruzione per generare un interrupt SW(SI) ha un operando intero usato come codice dell'interrupt.

MASCHERAMENTO INTERRUPT

Il campo **IM** (interrupt mask) del PSW indica a quali interrupt è consentito di verificarsi in quel momento.

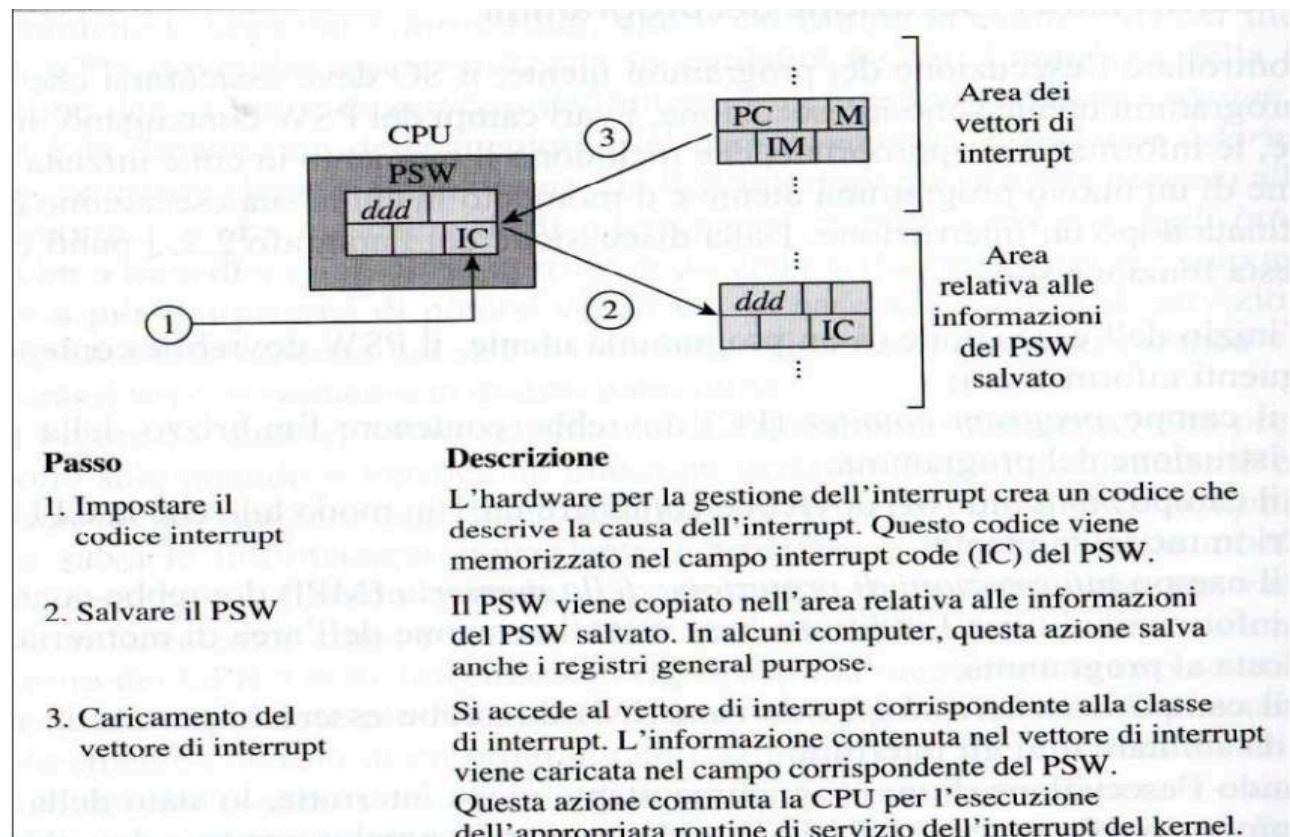
IM può contenere un intero m per cui solo interrupt con priorità maggiore di m possono verificarsi.

Oppure contiene una stringa di bit dove ogni bit indica se un tipo di interrupt è abilitato o meno a verificarsi.

Un interrupt non abilitato è detto mascherato o disabilitato.

Il verificarsi di un interrupt disabilitato non viene perso, esso è pendente fino a che non è abilitato e rilevato.

AZIONE DI UN'INTERRUZIONE



INTERAZIONE DEL SO CON IL COMPUTER E I PROGRAMMI UTENTE

Il SO interagisce con il computer per:

- Conoscere info sugli eventi, in modo da poterli servire
- Ripristinare lo stato della CPU per riprendere l'esecuzione di un programma dopo aver servito un interrupt
- I programmi hanno bisogno di usare i servizi del SO per scopi quali iniziare un'operazione di I/O
 - Il metodo che causa un interrupt e passa la richiesta al SO è noto come CHIAMATA DI SISTEMA(system call)

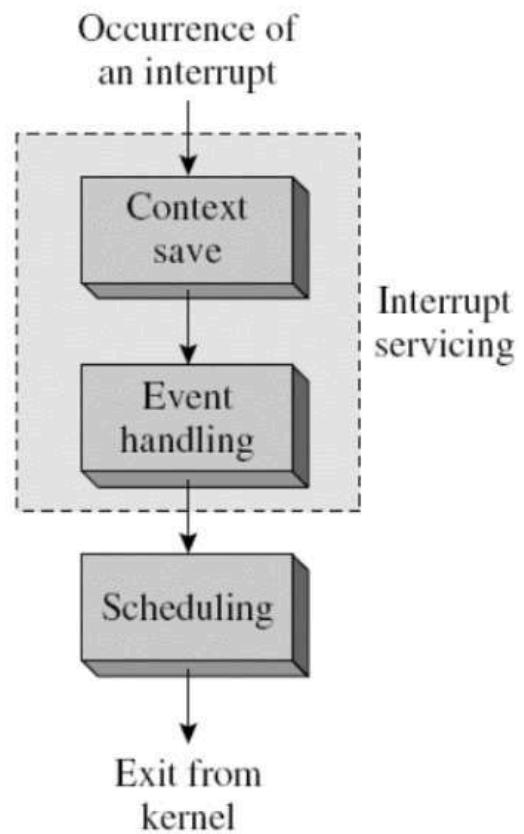
CONTROLLARE L'ESECUZIONE DEI PROGRAMMI

1. Quando inizia un programma utente, il PSW contiene:
 - a. Il campo PC (program counter)
 - b. Il campo MODE(M) impostato a modalità utente (1)
 - c. Il campo Info di protezione della memoria (MPI) con l'indirizzo di partenza in memoria e la dimensione del programma
 - d. Il campo Maschera Interrupt(IM) impostato con tutti gli interrupt abilitati
2. Quando un programma è interrotto, lo stato della CPU dovrebbe essere salvato
3. Quando deve essere ripristinata l'esecuzione del programma interrotto, lo stato salvato della CPU dovrebbe essere caricato nel PSW e nei GPR.

SERVIZIO DELL'INTERRUZIONE

Servizio dell'interruzione

- Il kernel costruisce il vettore degli interrupt, per ogni classe di interrupt, in fase di boot
 - Per semplicità consideriamo lo stesso formato del PSW
 - PC
 - Modalità
 - MPI (0,dim memoria)
 - IM
- Il Context save salva lo stato della CPU del programma
- Lo scheduler seleziona un programma per l'esecuzione



CHIAMATE DI SISTEMA

Un programma usa le risorse del computer come le periferiche di I/O. Tuttavia, le risorse sono condivise tra i programmi utente. Per facilitare la gestione delle risorse, le istruzioni che allocano o hanno accesso a risorse critiche sono implementate come istruzioni privilegiate.

Una system call è una richiesta che un programma fa al kernel attraverso un interrupt software.

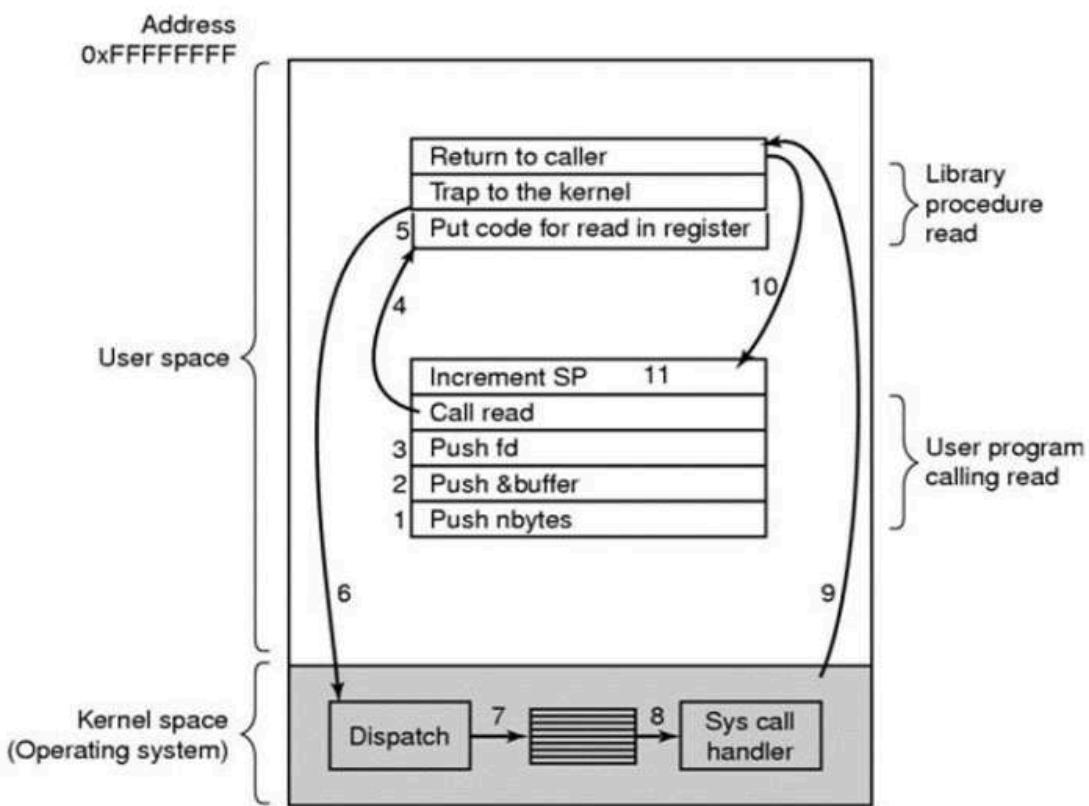
Chiamate di sistema (cont.)

- Alcune system call di Linux

Call number	Call name	Description
1	exit	Terminate execution of this program
3	read	Read data from a file
4	write	Write data into a file
5	open	Open a file
6	close	Close a file
7	waitpid	Wait for a program's execution to terminate
11	execve	Execute a program
12	chdir	Change working directory
14	chmod	Change file permissions
39	mkdir	Make a new directory
74	sethostname	Set hostname of the computer system
78	gettimeofday	Get time of day
79	settimeofday	Set time of day

Esempio: Esecuzione read

read (fd, &buff, nbytes)



LEZIONE 3 15/03/21

Un sistema di calcolo è costituito da un computer, da interfacce con altri sistemi e da servizi che sono messi a disposizione dal SO per i programmi utente. Nel corso degli anni i sistemi si sono evoluti con prestazioni elevate. Gli stessi sistemi si sono dovuti adeguare e sono aumentati in complessità per gestire una serie di compiti via via più complessi.

AMBIENTI DI CALCOLO E NATURA DELLE ELABORAZIONI

Un ambiente di calcolo consiste di:

- Computer
- Interfacce con altri sistemi
- I servizi forniti dal suo SO ai propri utenti ed i loro programmi

Inizialmente i sistemi non erano interattivi con l'utente ed era più complesso usarli. Nel corso degli anni gli utenti hanno avuto la possibilità di interazione di uno o più programmi, ad esempio fornire dati di input o rinominare un file.

Ambienti di calcolo non interattivi: il SO è orientato sull'uso efficiente delle risorse, le varie risorse dovevano essere usate per la maggior parte del tempo, bisognava ridurre il loro tempo di inattività. L'elaborazione era mandare ed eseguire dei moduli chiamati programmi o job (insieme di programmi che si eseguono sequenzialmente, se uno non termina correttamente, l'altro programma non viene eseguito).

Ambiente di calcolo interattivi: il SO è orientato sulla riduzione della quantità media di tempo richiesto per implementare un'interazione tra utente e la propria elaborazione poiché nei

sistemi non interattivi quando si davano in input i dati i tempi di elaborazione erano lunghi. L'esecuzione di un programma è detta processo. L'interazione con un processo avviene mediante una SOTTO RICHIESTA al processo.

Elaborazioni in un SO

Computation	Description
Program	A <i>program</i> is a set of functions or modules, including some functions or modules obtained from libraries.
Job	A <i>job</i> is a sequence of programs that together achieve a common goal. It is not meaningful to execute a program in a job unless previous programs in the job have been executed successfully.
Process	A <i>process</i> is an execution of a program.
Subrequest	A <i>subrequest</i> is the presentation of a computational requirement by a user to a process. Each subrequest produces a single response, which consists of a set of results or actions.

Ambienti real-time, distribuiti ed embedded:

- Un'elaborazione real-time ha specifici vincoli temporali, il SO assicura che le elaborazioni siano completate nei vincoli.
- Ambiente di calcolo distribuito: permette un'elaborazione per usare risorse condivise localizzate in più computer attraverso una rete.
- Ambiente di calcolo embedded: il computer è parte di uno specifico sistema HW. Di solito sono quelli che si trovano negli elettrodomestici. Il computer è tipicamente poco costoso con una configurazione minimale e il SO deve soddisfare i vincoli temporali che derivano dalla natura del sistema da controllare.

Ambienti di calcolo moderni: supportano numerose e diverse applicazioni. Hanno caratteristiche tratte dai diversi ambienti di calcolo appena descritti. Il SO impiega strategie complesse per gestire le elaborazioni dell'utente e le risorse, ad esempio deve ridurre il tempo medio richiesto per implementare un'interazione tra un utente e un'elaborazione e assicurare un uso efficiente delle risorse.

CLASSI DI SO

Caratteristiche chiave delle classi di SO

OS class	Period	Prime concern	Key concepts
Batch processing	1960s	CPU idle time	Automate transition between jobs
Multiprogramming	1960s	Resource utilization	Program priorities, preemption
Time-sharing	1970s	Good response time	Time slice, round-robin scheduling
Real time	1980s	Meeting time constraints	Real-time scheduling
Distributed	1990s	Resource sharing	Distributed control, transparency

EFFICIENZA, PRESTAZIONI DEL SISTEMA E SERVIZI UTENTE

Due tra gli obiettivi fondamentali di un SO:

1. Efficienza d'uso di una risorsa. Le risorse sono CPU, hardware e dispositivi di I/O.

Cercare di capire quanto è inutilizzata una risorsa e minimizzare i tempi. Una CPU è destinata a eseguire programmi utente, chiaramente nel momento in cui un programma utente invoca un'operazione di I/O deve attendere quell'operazione e ci possono essere periodi di tempo dove la CPU non fa niente in attesa che il programma ottenga l'operazione. Tutto ciò rappresenta overhead e deve essere contenuto.

2. Convenienza per l'utente. Ci sono aspetti oggettivi e non.

Posso valutare l'usabilità di un sistema (soggettivo), un altro esempio è il tempo di **turnaround** e **tempo di risposta** introdotti per valutare come si comporta un sistema relativamente all'esigenza di un utente.

Per un amministratore di sistema, sono più importante le prestazioni di un sistema nel suo ambiente, tipicamente misurate come **throughput** cioè la quantità di lavoro svolto in un tempo. Per valutare le prestazioni di un sistema, quando abbiamo deciso una giusta combinazione di efficienza e servizi utenti, è importante conoscere la quantità di lavoro svolta nell'unità di tempo.

DEFINIZIONI:

- Throughput: il numero medio di job, programmi, processi completate da un sistema nell'unità di tempo.
 - Turnaround: il tempo dalla sottomissione di un job, programma o processo da un utente fino all'istante in cui i risultati sono resi disponibili all'utente.
 - Tempo di risposta: il tempo della sottomissione di una sottorichiesta dell'utente all'istante in cui un processo risponde ad essa.
-

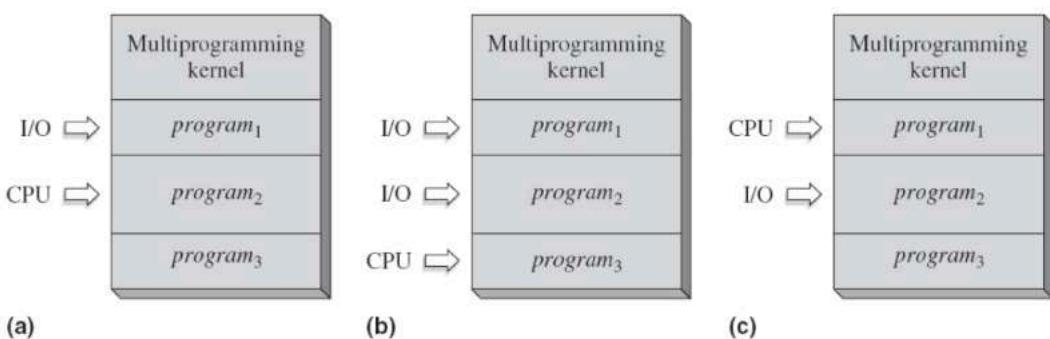
Sistemi di elaborazione Batch

- **Batch**: sequenza di job utente preparati per essere elaborati dal SO.

In tal modo il kernel avvia l'elaborazione dei job senza che sia richiesto l'intervento dell'operatore al computer.

SISTEMI MULTI-PROGRAMMATI

Forniscono un utilizzo efficiente delle risorse in un ambiente non interattivo. Usano la modalità DMA dell'I/O. Possono eseguire le operazioni di I/O di alcuni programmi mentre la CPU sta eseguendo altri programmi. In questi sistemi la misura più adatta per i servizi utente è il tempo di turnaround di un programma.



Funzionamento di un sistema multi-programmato:

- (a) programma₂ è in esecuzione mentre programma₁ sta eseguendo un'operazione di I/O;
- (b) programma₂ avvia un'operazione di I/O, programma₃ è schedulato;
- (c) l'operazione di I/O del programma₁ è completata ed è schedulato.

In un sistema multi-programmato:

- Il DMA consente la multiprogrammazione.
- La protezione della memoria previene eventuali accessi non autorizzati al di fuori dello spazio di indirizzamento definito dal contenuto del registro base e del registro size (limite)
- Le modalità kernel e utente della CPU prevengono le interferenze tra i programmi permettendo l'esecuzione delle istruzioni privilegiate solo al kernel (se un programma utente cercasse di effettuare un'operazione del genere verrebbe generato un interrupt).

Una misura delle prestazioni adeguata a un SO multiprogrammato è il throughput. Il SO mantiene sempre un numero sufficiente di programmi in memoria in modo che la CPU e i dispositivi di memoria non siano inattivi (idle). Non è sufficiente aumentare solo il grado di multiprogrammazione (numero di programmi) ma devo aggiungere in memoria dei programmi che hanno il giusto mix di programmi CPU-bound e I/O-bound.

- I/O bound: sono processi che invocano operazioni perlopiù I/O.
- CPU bound: sono impegnati con la CPU e fanno poche richieste di operazioni I/O.

È fondamentale avere questo mix poiché se avessi tutti processi I/O-bound, la CPU sarebbe ferma e viceversa.

SISTEMI MULTIPROGRAMMATI: PRIORITA' DEI PROGRAMMI

Technique	Description
Appropriate program mix	<p>The kernel keeps a mix of CPU-bound and I/O-bound programs in memory, where</p> <ul style="list-style-type: none"> • A <i>CPU-bound program</i> is a program involving a lot of computation and very little I/O. It uses the CPU in long bursts—that is, it uses the CPU for a long time before starting an I/O operation. • An <i>I/O-bound program</i> involves very little computation and a lot of I/O. It uses the CPU in small bursts.
Priority-based preemptive scheduling	<p>Every program is assigned a priority. The CPU is always allocated to the highest-priority program that wishes to use it. A low-priority program executing on the CPU is preempted if a higher-priority program wishes to use the CPU.</p>

- **Definizione di priorità:** Un criterio di tie-break mediante il quale uno scheduler decide quale richiesta dovrebbe essere schedulata quando molte richieste sono in attesa.

È importante che la priorità l'abbiano i processi I/O-bound.

PRESTAZIONI DEI SISTEMI MULTIPROGRAMMATI

Come migliorare le prestazioni? (**throughput**)

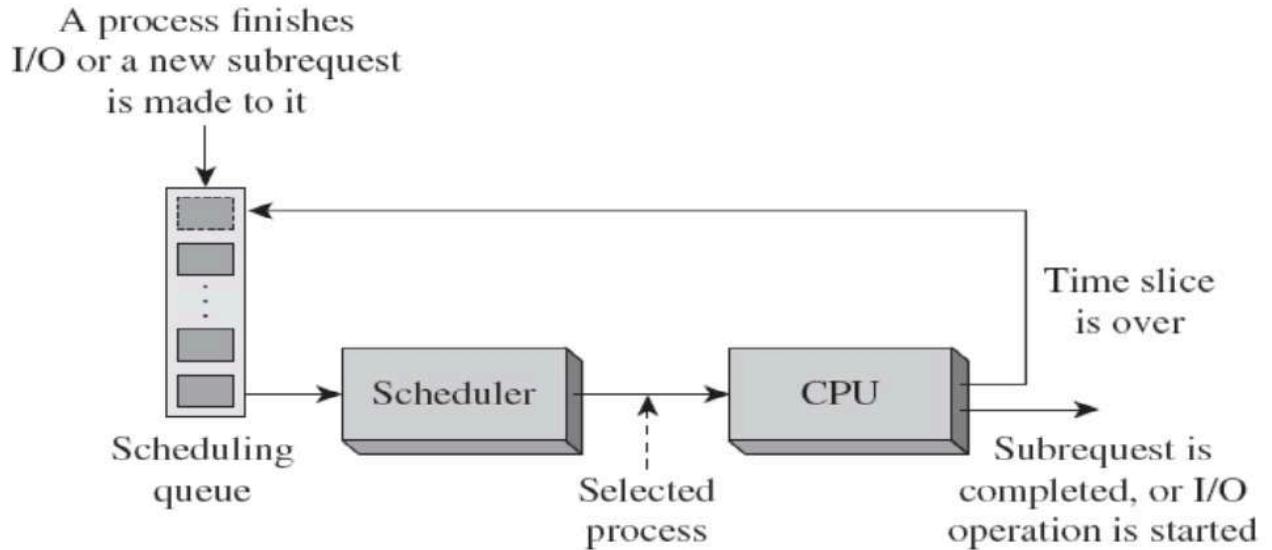
- Aggiungo un programma CPU-bound: schedulo nei periodi di inattività.
 - Aggiungo un programma I/O-bound: la presenza di un nuovo programma può essere utilizzata per migliorare i dispositivi di I/O.
-

SISTEMI TIME SHARING

Fornisce una risposta veloce alla sotto-richieste dell'utente. Gli utenti potevano sottomettere le proprie richieste di processi e la misura di convenienza dell'utente è il tempo di risposta. Per realizzare questi sistemi è stata introdotta una nuova politica di *scheduling*, lo *scheduling round robin con time slicing*. L'idea è che posso avere di un sistema più utenti che possono lanciare più processi e bisogna garantire che tutti i processi vengano garantiti in maniera simultanea in modo tale da assegnare una quantità di tempo per processo. Così tutti i processi possono essere eseguiti. Lo scheduler sceglie quindi un processo dalla coda e possono succedere diverse cose:

- Il processo selezionato viene completato e si passa al prossimo
- Il processo invoca un processo di I/O quindi si blocca e si deve schedulare un altro processo
- Il processo ha esaurito il time-slice ad esso destinato, uguale per tutti i processi. Il time-slice viene implementato mediante un interrupt di un timer.

Un processo specifico per essere completato potrebbe essere schedulato varie volte.



Abbiamo detto che per esprimere la convenienza per un utente consideriamo il tempo di risposta (rt): misura dei servizi utente. Se l'elaborazione di una sotto-richiesta richiede δ secondi di CPU:

$$rt = n \times (\delta + \sigma)$$

$$\eta = \delta / (\delta + \sigma)$$

Dove

- n : numero utenti che usano il sistema,
- δ : tempo richiesto per completare una sotto-richiesta,
- σ : overhead dello scheduling,
- η : efficienza di CPU

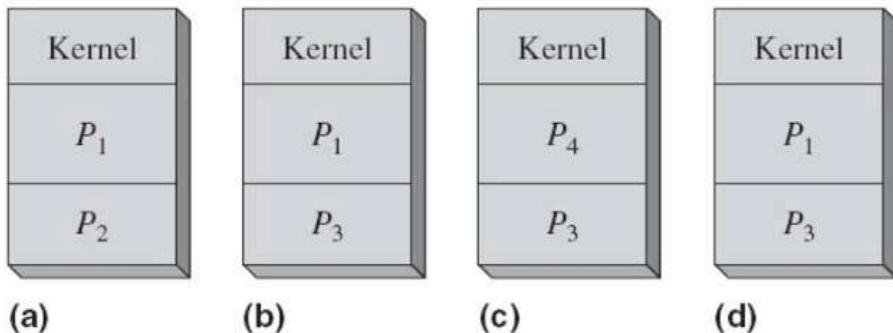
Ovviamente queste formule non sono precise. Il tempo di risposta effettivo potrebbe essere diverso poiché alcuni utenti possono essere inattivi o alcuni programmi possono richiedere più di δ secondi di CPU.

SWAPPING DEI PROGRAMMI

Lo swapping dei programmi è il concetto che si introduce per migliorare le prestazioni in un sistema time-sharing multiprogrammato.

Lo swapping è una tecnica che rimuove temporaneamente un processo dalla memoria di un elaboratore.

Il kernel esegue operazioni di *swap_in* e *swap_out*



Swapping: (a) processi in memoria tra 0 e 105 ms; (b) P_2 è sostituito da P_3 a 105 ms; (c) P_1 è sostituito da P_4 a 125 ms; (d) P_1 è soggetto a swap-in per servire la successiva sotto-richiesta relativa ad esso.

SISTEMI OPERATIVI REAL-TIME

Di questi sistemi quello che interessa è svolgere delle operazioni in un tempo predefinito. I tempi che devono essere rispettati dipendono dal tipo di applicazioni. È possibile distinguere due sistemi:

- Sistemi real-time hard
- Sistemi real-time soft

I sistemi hard sono quelli per cui è necessario assicurare il completamento dell'operazione negli stretti vincoli dettati dall'operazione.

Nei sistemi soft questi requisiti possono anche non essere rispettati però si cerca di ottimizzare il suo completamento almeno in un punto di vista probabilistico (si garantisce che con una buona probabilità possono essere rispettati i vincoli temporali). I sistemi soft possono essere le applicazioni multimediali.

SISTEMI OPERATIVI DISTRIBUITI

Questi sistemi hanno molte risorse distribuite su aree geografiche differenti e devono coordinare e garantire trasparenza di queste.

I MODERNI SO

Nei moderni SO sono presenti tutti i SO di cui abbiamo parlato precedentemente. È impensabile che un SO moderno non abbia le caratteristiche di quelli passati.

Concept	Typical example of use
Batch processing	To avoid time-consuming initializations for each use of a resource; e.g., database transactions are batch-processed in the back office and scientific computations are batch-processed in research organizations and clinical laboratories.
Priority-based preemptive scheduling	To provide a favored treatment to high-priority applications, and to achieve efficient use of resources by assigning high priorities to interactive processes and low priorities to noninteractive processes.
Time-slicing	To prevent a process from monopolizing the CPU; it helps in providing good response times.
Swapping	To increase the number of processes that can be serviced simultaneously; it helps in improving system performance and response times of processes.
Creating multiple processes in an application	To reduce the duration of an application; it is most effective when the application contains substantial CPU and I/O activities.
Resource sharing	To share resources such as laser printers or services such as file servers in a LAN environment.

STRUTTURA DI UN SO

Funzionamento di un SO

Quando un computer è avviato, è eseguita una procedura di boot: viene analizzata la sua configurazione, il tipo di CPU, la dimensione della memoria, i dispositivi I/O e altri dettagli HW.

Carica parte del SO in memoria e inizializza le strutture dati passando ad esso il controllo del sistema.

Durante il funzionamento del computer possono verificarsi delle interruzioni causate da:

- Un evento: completamento di un'operazione di I/O
- Una syscall fatta da un processo

Quando vi è un'interruzione il sistema operativo passa la CPU in modalità kernel, si costruisce il vettore di interrupt nella quale dove c'è la prima istruzione della routine di servizio di quella specifica interruzione, salva il processo e poi si restituisce il controllo in modalità utente dove o si seleziona un nuovo processo o si riprende quello interrotto.

Function	Description
Process management	Initiation and termination of processes, scheduling
Memory management	Allocation and deallocation of memory, swapping, virtual memory management
I/O management	I/O interrupt servicing, initiation of I/O operations, optimization of I/O device performance
File management	Creation, storage and access of files
Security and protection	Preventing interference with processes and resources
Network management	Sending and receiving of data over the network

POLITICHE E MECCANISMI

Quando si progetta un SO queste funzioni principali che il SO deve svolgere devono essere progettate. Questa non è un'operazione banale. Per semplificare si è soliti dividere il livello di progettazione sotto due punti: punti di vista di politica e punto di vista di meccanismo.

- Punto di vista di politica: principio in base al quale il SO esegue la funzione, decidendo cosa dovrebbe essere fatto. Ad esempio potrei introdurre la politica dello scheduler round-robin.

- Punto di vista di meccanismo: azione necessaria per implementare la politica. Se la politica è lo scheduling round-robin, il meccanismo è l'implementazione di quest'ultimo.
-

PORATIBILITÀ DEI SO

I SO principalmente erano funzionante solo su un'architettura. Ora con le nuove innovazioni tutti i SO sono portabili: possono essere eseguiti su differenti architetture. L'azione di ***porting*** significa adattare il software per usarlo in un nuovo sistema di computer mentre la ***portabilità*** è la semplicità con cui un programma può essere sottoposto a porting.

Porting di un SO significa cambiare parti del suo codice che dipendono dall'architettura per farlo funzionare su un diverso HW.

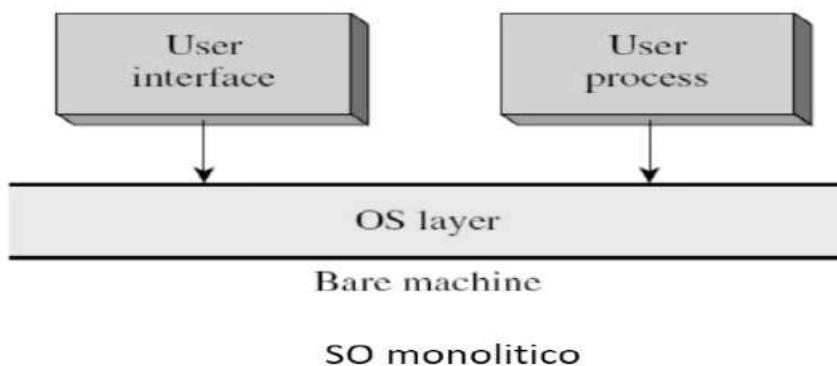
ESTENDIBILITÀ DEI SO

L'estendibilità del sistema è la facilità con cui posso aggiungere nuove funzionalità. Queste possono avvenire per un'aggiunta di un dispositivo alla macchina o aggiunta di nuove funzioni.

Nei moderni sistemi abbiamo la funzione *plug_and_play* dove possiamo aggiungere una nuova periferica e questa verrà riconosciuta.

SO CON STRUTTURA MONOLITICA

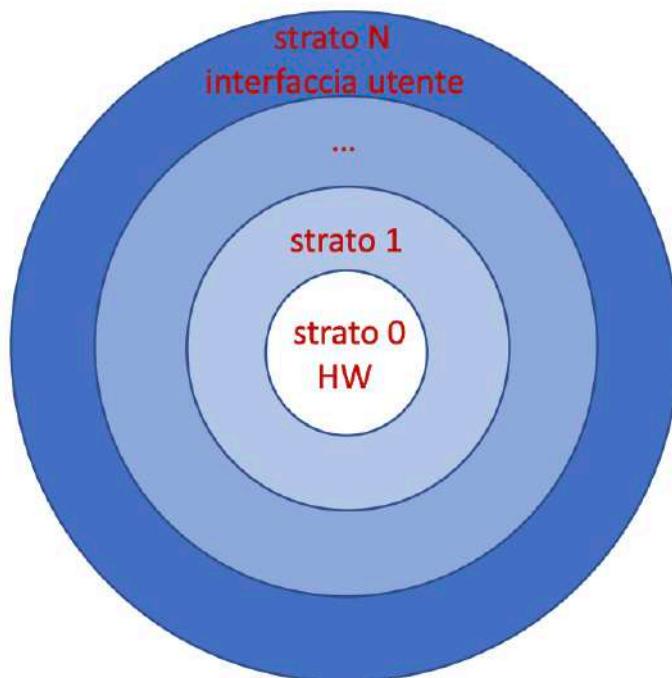
I primi SO avevano una struttura monolitica. Il SO costituiva un singolo strato software. Tra l'utente e la nuda macchina (HW). Immaginiamo un unico eseguibile che si pone sopra la macchina.



La struttura monolitica ha vantaggi e svantaggi.

Problema dei SO monolitici: sono difficili da implementare e da estendere. Erano poi poco portabili poiché le istruzioni essendo in unico file binario, sono tutte distribuite nel codice.

Il vantaggio è la comunicazione tra i vari moduli è molto più efficiente. Si è ricorso in passato a diverse strategie alternative. Un modo per gestire questo sistema è quello di utilizzare un’architettura a strati: dall’ hardware fino ad arrivare all’utente. Uno strato funziona tramite le funzioni degli strati inferiori.

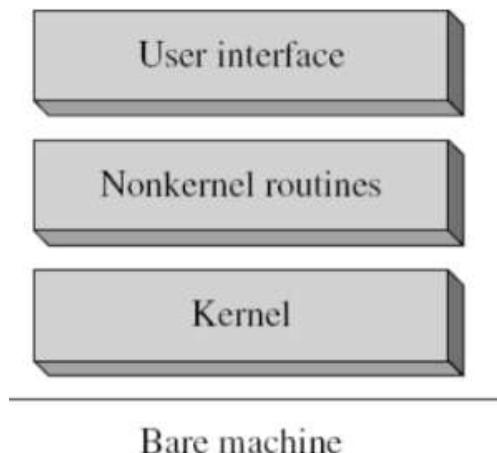


STRUTTURA SISTEMA MULTI-PROGRAMMATO

I problemi di un sistema progettato a strati è che da un punto di vista di efficienza perde molto perché è molto più lento: è possibile che per un'operazione I/O debba attraversare vari strati. Anche se semplifica rispetto alla progettazione monolitica, questa non è banale poiché devo decidere in quale strato mettere delle funzionalità e ognuna deve essere utile a quello sottostante.

SO BASATI SUL KERNEL

L'obiettivo era la portabilità e semplificare la progettazione. L'idea era quella di minimizzare la quantità di codice che dipende dall'architettura (per passarla su un'altra macchina devo riscriverlo). Le funzioni principali venivano implementate sul kernel in poche righe di codice e tutte le altre venivano viste come routine non kernel.



Il kernel degli Unix ha cominciato a crescere: per migliorare le prestazioni, nel kernel venivano messe funzioni anche non kernel. Lo rendevano però meno portabile e più pesante. In seguito questi si sono evoluti verso moduli kernel che possono essere caricati dinamicamente. Il kernel viene progettato come un

kernel con insieme di moduli. Si carica il kernel base, dopo di che gli altri moduli sono ricaricati all'occorrenza.

SO BASATI SU MICRO-KERNEL

I micro-kernel furono sviluppati per ovviare ai problemi di portabilità, estendibilità e affidabilità dei kernel.

Un micro-kernel è un nucleo essenziale del codice del SO: si implementano le funzionalità base e le altre sono implementate come dei server che invocano i servizi del micro-kernel. . Il micro-kernel non include lo scheduler e il gestore della memoria.

Un esempio di micro-kernel è il sistema MAC o quello Darwin dove si basa il sistema OSx di Apple.

Lezione 19/03/2021

PROCESSI

Un processo è un programma in esecuzione.

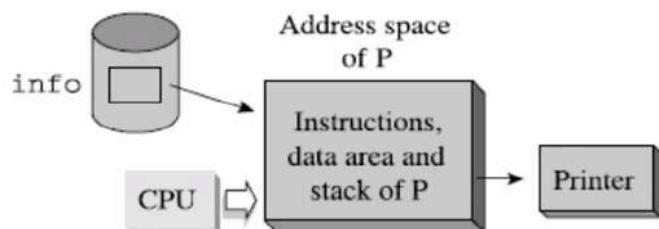
In un SO possiamo avere più processi in esecuzione simultaneamente.

Processo: esecuzione di un programma usando le risorse ad esso allocate.

Program P

```
file info;
int item;
open (info, "read");
while not
    end-of-file (info)
        read (info, item);
        ...
print ...;
stop;
```

(a)



(b)

Il SO deve creare un processo, creare spazio di memoria e creare le risorse associate. Un'altra risorsa è la CPU ma non è costantemente disponibile al programma per motivi già noti.

COMPONENTI DI UN PROCESSO

Un processo comprende sei componenti: id, codice, dati, stack, risorse, stato CPU.

- ID identificativo assegnato dal SO
- Codice → è il codice del programma
- Dati → dati usati durante l'esecuzione
- Stack → contiene parametri delle funzioni e procedure invocate durante l'esecuzione e loro indirizzi di ritorno
- Risorse → è l'insieme di risorse allocate dal SO
- Stato della CPU → composto da PSW e GPR
 - Ricordiamo che PSW

Program counter (PC)	Condition code (CC)	Mode (M)	Memory protection information (MPI)	Interrupt mask (IM)	Interrupt code (IC)
-------------------------	------------------------	-------------	--	------------------------	------------------------

RELAZIONI TRA PROCESSI E PROGRAMMI

Un programma è un insieme di funzioni e procedure. Le funzioni possono essere processi separati o possono costituire la parte di codice di un singolo processo.

Relationship	Examples
One-to-one	A single execution of a sequential program.
Many-to-one	Many simultaneous executions of a program, execution of a concurrent program.

- Relazione 1 a 1 con un singolo programma eseguito attraverso un processo.
 - Relazione Più a 1, abbiamo un programma a cui sono associati più processi: o perché di quel programma mando in esecuzione più istanze (immaginare un'applicazione client server dove mandiamo in esecuzione n volte lo stesso processo con lo stesso codice) o perché durante l'esecuzione del programma, all'interno del programma ci sono istruzioni dove le istruzioni di alcune funzioni sono demandate ad altri processi.
-

PROCESSI FIGLI

Posso costruire un'applicazione costituita da più processi che svolgono sottofunzioni del processo principale. Questo lo si fa istruendo il kernel che deve creare un nuovo processo. Il processo originario può effettuare una chiamata di sistema per creare altri processi. Quando un processo crea un altro processo, sta generando un processo figlio. Processi figli e genitori creano un albero dei processi. Un processo crea uno o più processi figli e attribuisce parte del proprio lavoro a ciascuno. Questo ci permette di creare multi-tasking in un'applicazione e avere più benefici:

- Speedup dell'elaborazione (maggior velocità di esecuzione): Diminuzione del tempo di esecuzione dell'applicazione grazie alla creazione di processi figli. Se il processo primario non creasse processi figli, eseguirebbe le operazioni sequenzialmente; invece creando i processi figli, questi eseguono concorrentemente le operazioni.
- Priorità per le funzioni critiche: Molti SO consentono a un processo genitore di assegnare priorità ai processi figli. Un'applicazione real-time può

assegnare una priorità alta a un processo figlio che deve eseguire una funzione critica in modo tale da soddisfare i suoi requisiti di risposta.

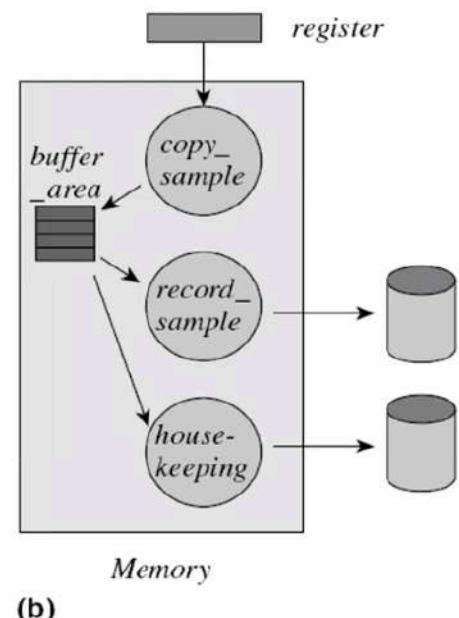
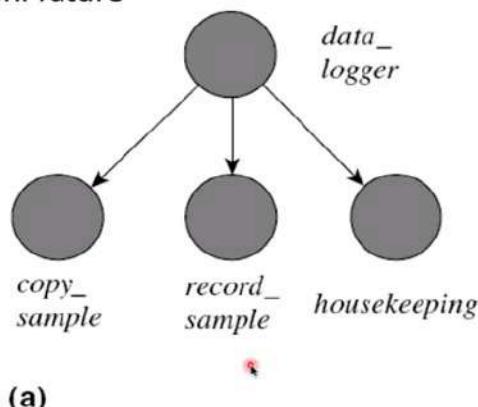
- Proteggere un processo genitore dagli errori:

Il kernel può terminare un processo figlio in caso di errore, ma il padre resta protetto e può avviare un'azione di recupero.

Esempio: processi figli in un'applicazione real-time

Il processo iniziale, `data_logger`, deve eseguire tre funzioni:

1. Copiare il campione da uno speciale registro in memoria
2. Copiare il campione dalla memoria in un file
3. Effettuare alcune analisi del campione e memorizzare il risultato in un altro file usato per elaborazioni future



Albero dei processi e processi

CONCORRENZA E PARALLELISMO

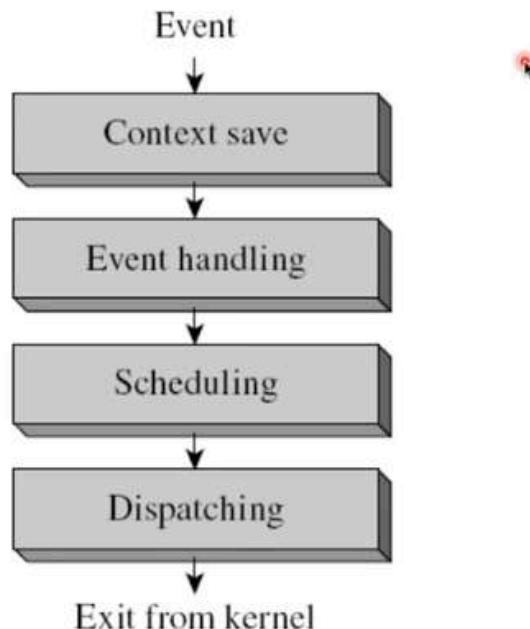
- **Parallelismo:** possibilità di far svolgere diverse azioni contemporaneamente. Lo ottengo solo se il calcolatore ha più CPU, assegnando azioni per ogni CPU

- **Concorrenza:** è un'illusione di parallelismo. Due attività sono concorrenti se c'è un'illusione che esse sono eseguite in parallelo laddove solo una di loro può essere eseguita in un dato momento. In un SO è ottenuta alternando le operazioni di vari processi sulla CPU

Concorrenza e parallelismo possono fornire un migliore throughput

IMPLEMENTAZIONE DEI PROCESSI

Per un SO, un processo è un'unità di lavoro computazionale. Il compito primario del kernel è controllare le operazioni dei processi per assicurare un uso efficace di un sistema di computer.



Funzioni fondamentali del kernel per controllare i processi

1. **Salvataggio del contesto:** salva lo stato della CPU e le informazioni riguardanti il processo interrotto.

2. **Gestione dell'evento:** analizza la condizione che ha generato l'interrupt o la system call ed effettua le azioni appropriate.
3. **Scheduling:** seleziona il prossimo processo da eseguire sulla CPU.
4. **Dispatching:** impostare il controllo della CPU (o l'accesso alle risorse) per il processo schedulato e caricare il suo stato salvato della CPU per iniziare o proseguire l'esecuzione e modifica lo stato del processo in running.

STATO DI UN PROCESSO E TRANSIZIONI DI STATO

Stato di un processo: l'indicatore che descrive la natura dell'attività corrente di un processo.

Stato	Descrizione
Running	Una CPU sta eseguendo le istruzioni di un processo.
Blocked	Il processo deve aspettare finché non viene soddisfatta una sua richiesta per una risorsa o finché non si verifica uno specifico evento.
Ready	Il processo richiede l'uso della CPU per continuare la sua esecuzione; tuttavia, non è stato ancora eseguito il dispatch.
Terminated	L'esecuzione del processo, ovvero, l'istanza del programma che rappresenta, è stata completata correttamente o è stata terminata dal kernel.

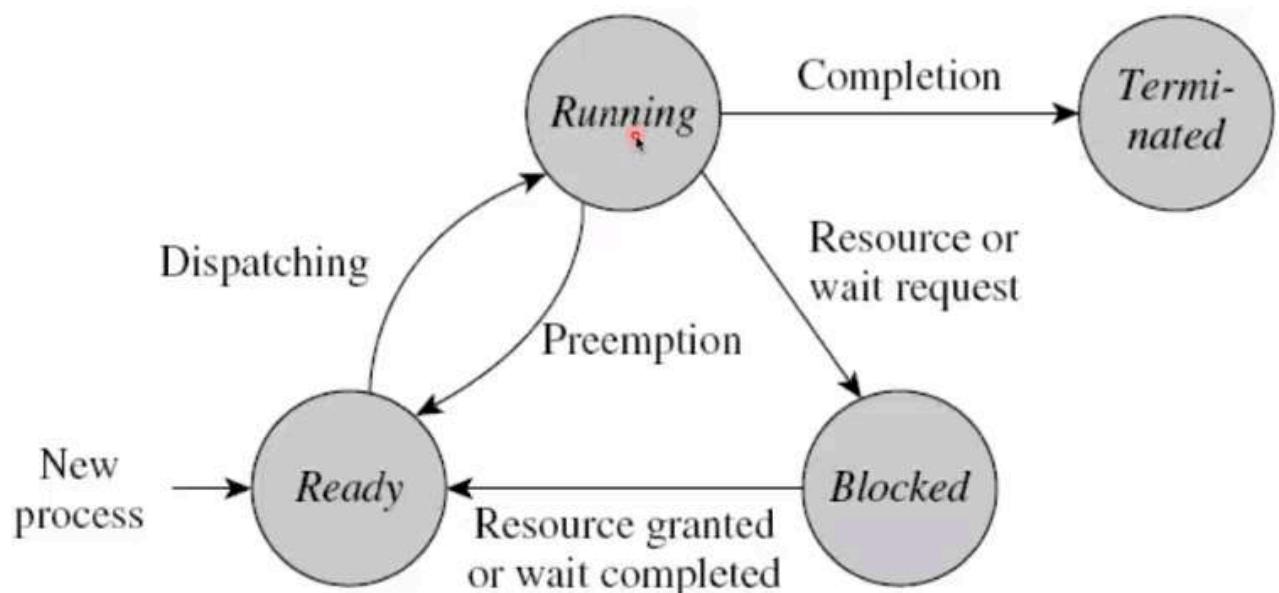
Gli stati di un processo dipendono dai SO. Ogni SO ha la sua astrazione ma di base tutti i SO hanno questi **4** stati base citati.

- Un processo è bloccato (**blocked**) quando è in attesa che una risorsa da lui richiesta gli venga allocata o quando è in attesa del verificarsi di un particolare evento.
- Un processo è pronto (**ready**) quando può essere schedulato, cioè quando la richiesta è soddisfatta o quando si è verificato l'evento che stava attendendo.
- Un processo è in esecuzione (**running**) quando la CPU sta eseguendo le istruzioni del processo stesso.

- Un processo è terminato (**terminated**) quando viene portata a termine la sua esecuzione o se, per qualche motivo, viene terminato dal kernel.

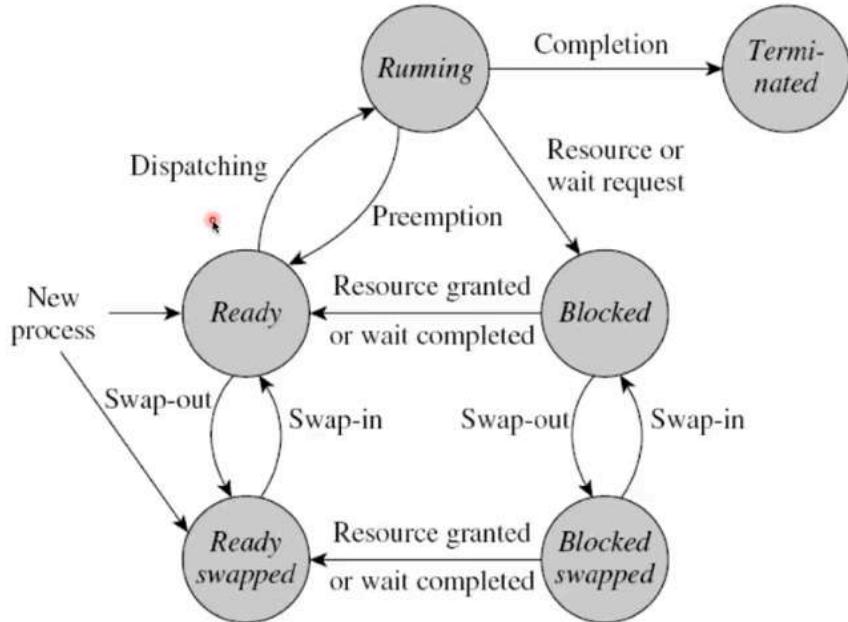
Un computer che ha una sola CPU può avere un solo processo nello stato di running, ma più processi negli stati blocked, ready e terminated.

Una transizione di stato per un processo è un cambio del proprio stato. È causata dall'occorrenza di qualche evento come l'inizio o la fine di un'operazione di I/O



PROCESSI SOSPESI

Un kernel ha bisogno di stati aggiuntivi per descrivere i processi sospesi per lo swapping.



Stati di processo e transizioni di stati usando due stati swapped

Questo succede quando un processo è bloccato per diversi motivi. Il SO monitora il comportamento di tutti i processi in esecuzione e può stabilire che la mia operazione potrebbe essere lunga e a questo punto potrebbe decidere di rimuovermi dalla memoria principale e portarmi alla secondaria (*SWAP OUT*). Un programma in swap out o è ancora bloccato e il sistema mi riporta in principale (*Swap in*) poiché crede che a breve verrò sbloccato oppure durante il periodo della memoria secondaria, la risorsa è stata completata e quindi il sistema mi passa da *swapped* a *ready swapped*. le decisioni di swap in/out dipendono dalla memoria in quel momento.

CONTESTO DI UN PROCESSO E PROCESSO CONTROL BLOCK

Il kernel alloca risorse ad un processo e lo schedula per l'uso della CPU. Il SO deve mantenere le info associate al processo che possiamo dividere in due parti:

- Contesto del processo: tutte le info associate alle risorse allocate al processo
 - Process control block: struttura dati associata a un processo che contiene le info sullo stato del processo, necessarie per poter ripristinare correttamente l'esecuzione del processo quando questo viene prelazionato all'occorrenza di un'interrupt.
-

CONTESTO DI UN PROCESSO

- Spazio di indirizzamento del processo: il blocco di memoria associato al processo con dati, codice e stack
 - Info sulle allocazioni della memoria: le aree di memoria associate al processo, usate dalla MMU.
 - Stato delle attività sui file: info su file in uso
 - Info sulle interazioni del processo: info necessarie per controllare le interazioni del processo con altri processi
 - Info sulle risorse: info sulle risorse allocate al processo
 - Info di miscellanea: mix di info necessarie per il funzionamento del processo
-

PROCESS CONTROL BLOCK

Contiene tre info:

- ID del processo, del genitore e dell'utente che lo ha creato
- Info di stato del processo: stato, contenuto del PSW e GPR
- Info per il controllo delle operazioni: priorità, interazione con altri processi.

Contiene anche un campo puntatore usato dal kernel per costruire la lista di PCB(process control block) per lo scheduling (i processi ready).

Campo del PCB	Contenuto
Id del processo	L'id univoco assegnato al processo al momento della creazione.
Id del genitore e dei figli	Questi id sono usati per la sincronizzazione dei processi, tipicamente per consentire a un processo di verificare se un processo figlio ha terminato la sua esecuzione.
Priorità	La priorità è generalmente un valore numerico. Al momento della creazione, al processo viene assegnata una priorità. Il kernel può cambiare la priorità dinamicamente in base alla natura del processo (CPU-bound o I/O-bound), al tempo di attività e alle risorse utilizzate (solitamente il tempo di CPU).
Stato del processo	Lo stato corrente del processo.
PSW	Questa è un'istantanea, ovvero un'immagine del PSW eseguita l'ultima volta che il processo è stato bloccato o prelazionato. Il caricamento di questa immagine nel PSW ripristina l'esecuzione del processo (vedi Figura 2.2 per i campi del PSW).
GPR	Il contenuto dei registri general purpose salvati l'ultima volta che il processo è stato bloccato o prelazionato.
Informazione sugli eventi	Per un processo nello stato <i>blocked</i> , questo campo contiene l'informazione relativa all'evento per il quale il processo è in attesa.
Informazioni sui segnali	Informazione relativa ai gestori dei segnali (vedi Paragrafo 5.2.6).
Puntatore al PCB	Questo campo è utilizzato per creare una lista di PCB, necessaria per la schedulazione.

SALVATAGGIO DEL CONTESTO, SCHEDULING E DISPATCHING

- La funzione di salvataggio del contesto salva lo stato della CPU del processo interrotto nel PCB e salva le info riguardo al contesto. Si cambia poi lo stato da *running* a *ready*.
- Lo scheduling usa le info sullo stato dai PCB per selezionare un processo ready per l'esecuzione e passa il suo ID alla funzione di dispatching.
- Funzione di dispatching: imposta il contesto del processo selezionato, cambia il suo stato a *running* e carica lo stato salvato della CPU dal PCB nella CPU. Scarica i buffer di traduzione dell'indirizzo usati dalla MMU.

Esempio: Salvataggio del Contesto, Scheduling e Dispatching

- Un SO contiene due processi P_1 e P_2 (priorità $P_2 > P_1$)
 - P_2 bloccato su un'operazione di I/O e P_1 in esecuzione
- Quando l'I/O di P_2 è completata:
 1. E' eseguita la funzione di salvataggio del contesto per P_1 ed il suo stato è cambiato in pronto
 2. Dal campo info evento del PCB, il gestore sa che l'I/O è stato iniziato da P_2 , quindi cambia lo stato di P_2 da bloccato a pronto
 3. E' eseguito lo scheduling che seleziona P_2 poiché ha priorità maggiore tra i processi pronti
 4. Lo stato di P_2 è cambiato in esecuzione ed avviene il dispatch

Commutazione di Processo

- Le azioni 1, 3 e 4 insieme costituiscono la commutazione tra i processi P_1 e P_2
- La commutazione avviene anche quando un processo in esecuzione diviene bloccato in conseguenza di una richiesta o della prelazione al termine di un time slice
- L'occorrenza di un evento non comporta la commutazione se esso
 - Causa una transizione di stato solo in un processo con la cui priorità è più bassa del processo interrotto dall'evento
 - Non causa alcuna transizione di stato, ad esempio, se l'evento è causato da una richiesta immediatamente soddisfatta
- L'overhead dipende dalla dimensione delle informazioni sullo stato del processo
- Alcune architetture di computer forniscono istruzioni speciali per ridurre l'overhead della commutazione
 - Salvano o caricano il PSW e tutti i GPR; scaricano i buffer di traduzione indirizzo usati dalla MMU
- Osservazione:
 - la commutazione può comportare anche un overhead indiretto
 - In nuovo processo potrebbe non avere alcuna parte del suo spazio di indirizzamento nella cache e quindi le sue prestazioni sono scadenti fino a quando sono memorizzate sufficienti info nella cache

GESTIONE DEGLI EVENTI

Il SO gestisce l'esecuzione dei processi ed è sempre pronta ad intercettare eventi sotto forma di interrupt.

Ci sono vari tipi di eventi:

1. *Evento per la creazione di un processo*: viene creato un nuovo processo
2. *Evento per la terminazione di un processo*: un processo termina la sua esecuzione
3. *Evento timer*: si verifica un interrupt del timer
4. *Evento per la richiesta di una risorsa*: un processo effettua la richiesta per una risorsa
5. *Evento per il rilascio di una risorsa*: un processo rilascia una risorsa
6. *Evento per la richiesta di avvio di I/O*: un processo richiede l'avvio di un'operazione di I/O
7. *Evento per il completamento di I/O*: un'operazione di I/O viene completata
8. *Evento per l'invio di un messaggio*: un processo invia un messaggio a un altro processo
9. *Evento per la ricezione di un messaggio*: un processo riceve un messaggio
10. *Evento per l'invio di un segnale*: un processo invia un segnale a un altro processo
11. *Evento per la ricezione di un segnale*: un processo riceve un segnale
12. *Un interrupt da programma*: l'istruzione corrente nel processo *running* non è eseguita correttamente
13. *Evento per il malfunzionamento dell'hardware*: un'unità hardware del computer ha causato un malfunzionamento

EVENTI: CREAZIONE DEI PROCESSI

Quando un utente invoca un comando o un processo, intende creare un processo figlio per eseguire un programma. Invoca una syscall per la creazione del nuovo processo. Questi eventi comportano la creazione di un nuovo processo. A questo punto la routine di gestione dell'evento deve creare tutto ciò che è necessario per rappresentare, eseguire e gestire un processo.

Crea un PCB per il nuovo processo, gli assegna un ID e una priorità e tutte info sono inserite nel PCB, all'interno di questo viene inserito anche l'ID del processo padre (**fork**). Determina poi la quantità di memoria da allocare per lo spazio di indirizzamento del processo e dispone il tutto per l'allocazione della memoria. Il kernel associa al processo alcune risorse standard (canali di I/O standard). Inserisce le info sulla memoria e le altre risorse

allocate nel contesto del nuovo processo e nel PCB viene impostato lo stato ready e lo immette in una lista di PCB.

EVENTI: TERMINAZIONE DEI PROCESSI

Un processo termina per vari motivi. Se parliamo di terminazione normale, un processo invoca una syscall per terminare sé stesso o un processo figlio. Quando il kernel riceve tali info, questo termina il processo quando le operazioni di I/O avviate dal processo sono terminate.

Rilascia la memoria e le risorse allocate. Lo stato del processo passa da running a terminated. Il PCB del processo terminato non viene rimosso fino a quando il processo padre non ne preleva il suo stato di terminazione. Nel caso, il padre era in attesa della terminazione del processo figlio, il kernel lo deve attivare.

- Il kernel prende l'id del processo padre dal PCB del processo terminato e controlla il campo info evento del PCB del processo padre per verificare se il processo padre è in attesa della terminazione del processo figlio

EVENTI: PRELAZIONE DEI PROCESSI

Un processo nello stato running deve essere prelazionato se scade il suo time slice. La funzione di salvataggi contesto cambia lo stato del processo running in ready prima di invocare il gestore dell'evento timer interrupt. Il gestore sposta solo il PCB del processo in un a lista di scheduling. La prelazione deve avvenire anche quando un processo con maggiore priorità diventa ready.

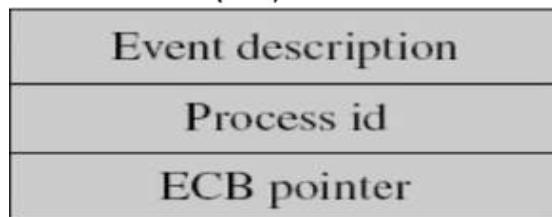
EVENTI: UTILIZZO DELLE RISORSE

Se un processo richiede una risorsa che non è immediatamente disponibile allora il nostro processo deve cambiare stato: da running a blocked. Il gestore dell'evento cambia lo stato e annota l'ID della risorsa richiesta nel campo info evento del PCB.

Se un processo rilascia una risorsa con una syscall, il gestore non deve cambiare lo stato di tale processo: deve controllare se altri processi sono bloccati in attesa di tale risorsa ed allocarla ad uno di essi cambiandone lo stato da blocked a ready.

Una syscall per l'avvio di un'operazione di I/O e un interrupt per segnalare il completamento di un'operazione di I/O comportano azioni di gestione dell'evento simili alle precedenti.

- Quando si verifica un evento, il kernel deve trovare il processo il cui stato è affetto dalla sua occorrenza
 - I SO usano vari schemi per velocizzare il tutto
 - Ad esempio, *Event Control Block* (ECB)



EVENT CONTROL BLOCK (ECB)

Quando si verifica un evento, il kernel deve trovare il processo il cui stato è influenzato dall'evento. L'operazione è effettuata mediante una ricerca nei campi delle informazioni relative agli eventi dei PCB dei processi. Questa ricerca è costosa in termini di tempo, per cui i SO usano vari schemi per velocizzarla. Uno di questi consiste nell'utilizzo dell'Event Control Block (ECB).

Un ECB contiene tre campi:

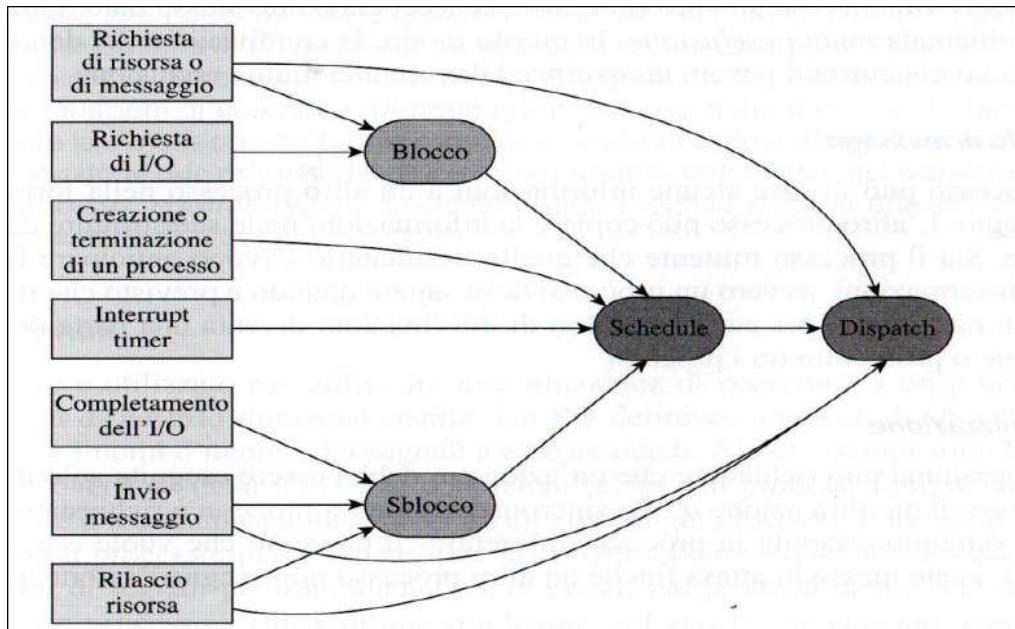
- - campo descrizione dell'evento, che descrive un evento
- - campo ID del processo, che contiene l'ID del processo in attesa dell'evento
- - campo puntatore all'ECB, che è necessario per inserire l'ECB di un processo nella lista appropriata, in

quanto il kernel può mantenere più liste separate di ECB per ogni classe di evento

Quando si verifica un evento, il kernel esamina la lista degli ECB appropriata per trovare un ECB con una descrizione dell'evento

corrispondente. Il campo ID indica quale processo è in attesa dell'evento.

GESTIONE DELL'EVENTO



1. La figura illustra le azioni del kernel per la gestione dell'evento descritte in precedenza.
L'azione block modifica sempre lo stato del processo chiamante da ready a blocked.
L'azione unblock cerca un processo la cui richiesta può essere soddisfatta e cambia lo stato da blocked a ready.
2. Una system call per la richiesta di una risorsa implica un'azione di block (seguita dallo scheduling e dal dispatching) se la risorsa non può essere allocata direttamente al processo che la richiede. Invece, se la risorsa può essere allocata direttamente, allora l'azione di blocco non viene effettuata e il processo viene semplicemente sottoposto a dispatching.
3. Quando un processo rilascia una risorsa, nel caso in cui un altro processo sia in attesa della risorsa rilasciata, viene eseguita un'azione di unblock (seguita da scheduling e

dispatching), poiché il processo sbloccato può avere una priorità maggiore del processo che ha rilasciato la risorsa. Anche in questo caso, lo scheduling non viene effettuato se in conseguenza dell'evento nessun processo è stato sbloccato.

LEZIONE 22/03/2021

CONDIVISIONE, COMUNICAZIONE E SINCRONIZZAZIONE TRA PROCESSI

Durante l'uso di una macchina, in ogni istante abbiamo un elevato numero di processi in esecuzione. Cosa succede quando ho processi concorrenti in esecuzione come preserviamo la correttezza e la coerenza dei dati? Questo significa che dobbiamo avere la possibilità di consentire a ciascun processo di poter accedere a quel dato condiviso in maniera sicura per essere certi che in quel momento solo un processo è in esecuzione e può manipolare i dati.

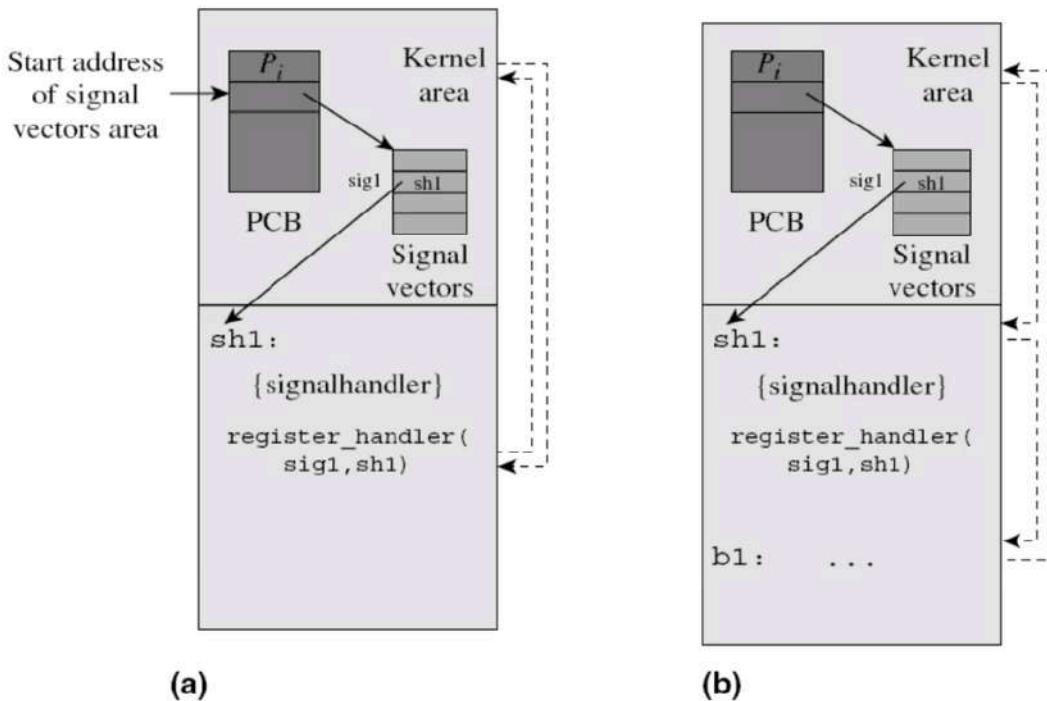
Quando mandiamo in esecuzione n processi, non conosciamo l'ordine di scheduling perché dipende dall'algoritmo di quest'ultimo. Se ho dati condivisi e non so quale processo viene eseguito in un dato momento, si pone il problema di coerenza. Vedere mutua esclusione. " " " " "

Quando abbiamo un certo numero di processi, questi possono interagire tra loro per uno scopo comune.

- Lo scambio di info non è immediato poiché lo spazio degli indirizzi di questi processi è separato: i processi non comunicano se non sfruttando meccanismi specifici come le pipe o scambio di messaggi (prevede assistenza da parte del SO per garantire sincronizzazione tra processi).

- L'altro aspetto è legato appunto alla sincronizzazione. Quando i processi devono cooperare, è fondamentale seguire un ordine di esecuzione. Posso introdurre un ordine di esecuzione al fine di mantenere la coerenza di tutte le operazioni che i processi svolgono e i loro dati.
- Un'altra possibilità per comunicare tra processi sono i SEGNALI. Il segnale è un meccanismo che somiglia alle interruzioni ma viene utilizzato nelle applicazioni utente per mandare info da un processo a un altro. L'uso dei segnali è importante per rappresentare situazioni eccezionali. Ci sono segnali legati ad eventi di HW come overflow della CPU o per notificare alcune info dei processi figli a un processo padre, quando termina un processo figlio viene inviato un segnale al processo padre. In ogni sistema abbiamo un certo numero di segnali, caratterizzati da un nome e da un ID. quando un processo riceve un segnale da un altro processo, il processo ricevente può gestire il segnale in tre modi diversi:
 - Eseguo un'azione di default
 - Ignoro il segnale
 - In conseguenza della ricezione del segnale, l'utente può predefinire una funzione che esegue una serie di compiti. Si dice che si scrive una funzione **handler**, cioè una funzione eseguita quando si riceve un determinato segnale.

Esempio: gestione di un segnale



Gestione segnale del processo P_i : (a) registrazione di un handler; (b) invocare un gestore

THREAD

I processi concorrenti velocizzano l'esecuzione delle applicazioni ma avere un certo numero di processi potrebbe creare overhead (a causa del passaggio di contesto da un processo all'altro il SO deve provvedere a salvare il contesto e il PCB del processo interrotto e caricare quello del processo corrente). Tutto il lavoro che fa il SO che non è eseguire codice utente è overhead.

Ovviamente il context switch crea un overhead fisso a cui non possiamo rimediare, invece possiamo fare qualcosa riguardo il salvataggio di info sulle risorse allocate al processo e sull'iterazione con altri processi.

Supponiamo di avere un processo P con P_i e P_J processi figli. Quando si crea un processo figlio, lo spazio di indirizzamento del padre viene copiato a quello dei figli. Ciò significa che appena creato il figlio è uguale al padre, tutte le info sul contesto sono

uguali se non hanno allocato alcuna risorsa. Quindi P_i e P_J hanno spesso info ridondanti. Se faccio cambio di contesto tra P_i e P_J queste info descritte sono le stesse, introduco overhead per caricare le stesse info identiche. Nascono così i thread. L'idea è di fare in modo che all'interno di ciascun processo, posso definire un'unità elaborativa più leggera (thread) e posso avere la possibilità all'interno di un processo di creare più thread. I thread di un processo condividono il contesto del processo nel quale sono stati creati. Quindi se ho più thread in esecuzione posso commutare da un thread all'altro e questo coinvolge l'overhead di esecuzione, lo stato delle risorse è condiviso. Questo tipo di operazione è molto più snella rispetto alla commutazione tra processi.

Thread: esecuzione di un programma che usa le risorse di un processo.

- Un thread è un modello alternativo di esecuzione di un programma
- Un processo crea un thread attraverso una syscall
- L'uso di thread suddivide lo stato di un processo in due parti:
 - 1- Lo stato della risorsa resta il processo
 - 2- Lo stato della CPU è associato con il thread

La commutazione tra thread comporta un minor overhead rispetto alla commutazione tra processi.

THREAD CONTROL BLOCK

Il TCB contiene info necessarie all'esecuzione del thread:

- L'ID del thread, la loro priorità e il loro stato.
- Stato della CPU: PSW e GPR.
- Puntatore al PCB del processo all'interno del quale è stato creato.
- Puntatore al TCB per creare la lista necessaria alla schedulazione.

STATO DI UN THREAD E TRANSIZIONI DI STATO

Essendo un'unità elaborativa, un thread è caratterizzato da stato e transizioni di stato.

Appena creo un thread questo viene impostato a *ready*.

Un thread può transire nello stato *running* quando è sottoposto a dispatching(caricamento ed esecuzione da parte della CPU).

Entra nello stato *blocked* perché condividendo memoria, potrebbe accadere che la sincronizzazione causa problemi(se t1 non finisce t2 deve aspettare e si blocca).

VANTAGGI THREAD VS PROCESSI

Abbiamo già visto che il primo vantaggio dei thread rispetto ai processi consiste nel minor overhead relativamente alla creazione e alla commutazione. Ci sono però anche altri vantaggi che, in varie situazioni, fanno preferire l'uso dei thread all'uso dei processi:

- Uno di questi è quello di avere una comunicazione più efficiente. In pratica, visto che i thread (diversamente dai processi) condividono lo spazio di indirizzamento del processo genitore, possono comunicare tra loro attraverso dati condivisi anziché mediante messaggi, evitando in questo modo l'overhead di comunicazione dovuto alle system call.
- Un altro vantaggio rispetto ai processi è la progettazione semplificata. Infatti l'uso dei thread può semplificare la progettazione e la codifica delle applicazioni che servono le richieste concorrentemente (es: prenotazioni dei voli online). La creazione e la terminazione dei thread è più efficiente rispetto alle medesime operazioni sui processi; tuttavia, il suo overhead può causare un decadimento delle prestazioni del server nel caso in cui i client effettuassero un

numero elevato di richieste. Per questo motivo si ricorre ad un'organizzazione chiamata thread pool che evita questo overhead: consiste nel riutilizzare i thread invece di distruggerli dopo aver soddisfatto le richieste.

CODIFICA PER USARE I THREAD

Molto importante è la correttezza dei dati su cui si lavora. Se invochiamo funzioni che invocano dati globali e statici, questi sono sorgenti di situazioni di incoerenza nei dati quando abbiamo applicazioni concorrenti. Dobbiamo assicurarci di usare funzioni thread safe che evitano di usare variabili condivise e dati statici.

Un altro problema è la gestione dei segnali: quale thread dovrebbe gestire un segnale?

Questa scelta può essere fatta dal kernel o dall'applicazione. Normalmente ci sono strategie concepite. Tipicamente se il segnale è associato a eccezione HW il thread che gestisce il segnale è lo stesso che ha causato l'eccezione HW : se ho un processo con 10 thread e il 3 ha causato una divisione per zero, il segnale deve essere gestito da quello che ha causato quel problema. Gli altri segnali sono inviati ad un qualsiasi thread del processo.

Il thread con priorità maggiore si occupa di un segnale.

I THREAD IN C: THREAD POSIX

I thread possono essere usati anche in C: dobbiamo usare una libreria <<pthread>>, contiene 60 funzioni che possiamo usare per la gestione dei thread (creazione, terminazione, aggiungere attributi, proprietà ...), per la condivisione dei dati con la mutua esclusione e assistenza per la sincronizzazione.

Un thread è creato con la chiamata a `pthread_create(<data structure>, <attributes>, <start routine>, <arguments>)`.

La sincronizzazione genitore figlio avviene con `pthread_join`

Un thread termina con `pthread_exit`.

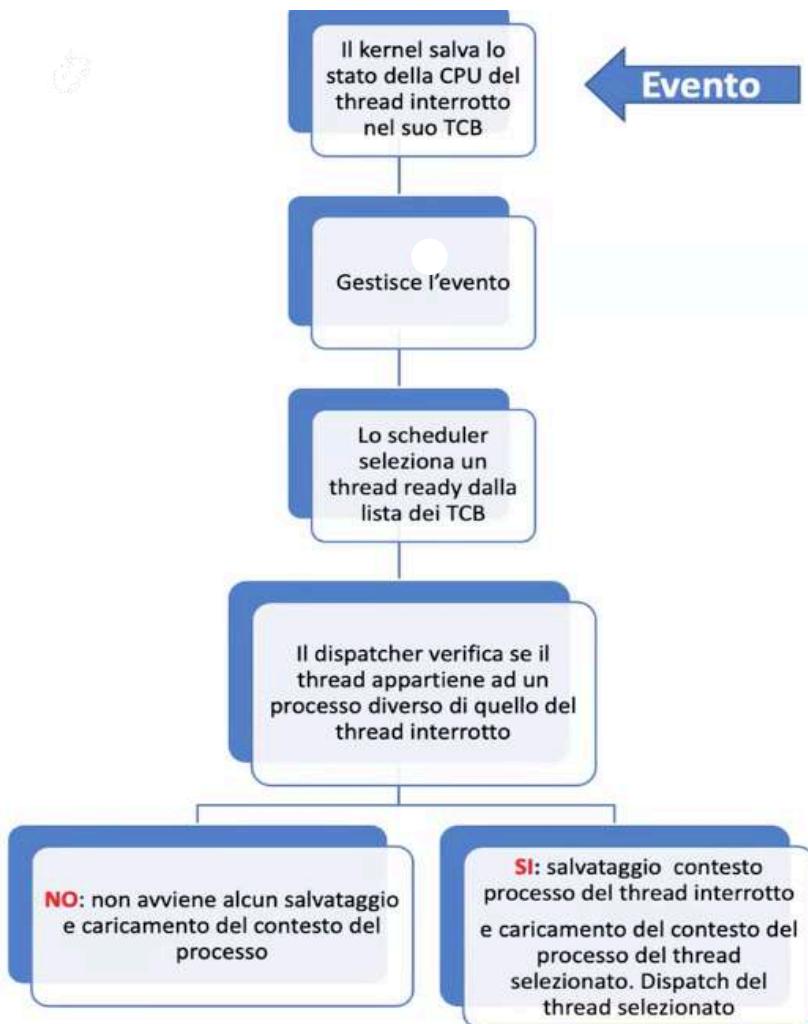
THREAD DI LIVELLO KERNEL, UTENTE E IBRIDI

Esistono diversi tipi di thread:

- Thread di livello kernel dove sono gestiti dal kernel
- Thread di livello utente, i thread sono gestiti dalla libreria dei thread
- Thread ibridi, combinazioni di thread di livello kernel e utente.

THREAD A LIVELLO KERNEL

Con i thread a livello kernel è come avere un processo ma con una quantità di informazioni inferiori. La commutazione tra thread dello stesso processo causa un overhead per la gestione dell'evento. Il kernel va a creare un thread con una syscall. Supponiamo di avere n processi e ogni processo ha m thread, alla creazione di un thread il SO crea un TCB che contiene tutte le info dette prima. Tutte queste info sono mantenute a livello kernel.



PRO E CONTRO THREAD LIVELLO KERNEL

- **PRO**
 - Thread simili ai processi (con meno info)
 - In ambienti multi CPU consentono il parallelismo
 - Più thread di un processo possono essere schedulati contemporaneamente.

- **CONTRO**

- La commutazione tra thread è fatta da kernel quando è gestito un evento
 - C'è un overhead di gestione evento anche se il thread interrotto e quello schedulato sono dello stesso processo
 - Limita il risparmio di overhead nella commutazione tra thread
-

THREAD DI LIVELLO UTENTE

La commutazione tra thread è più veloce perché non comporta nessuna syscall e il kernel non è coinvolto. Se c'è un thread utente che dovesse bloccarsi allora questo bloccherebbe tutti i thread del processo. Inoltre non c'è nessuna concorrenza o parallelismo.

PRO E CONTRO

- **PRO**

- Schedulazione e sincronizzazione gestita dalla libreria che evita le syscall per la sincro tra thread: la commutazione tra thread meno onerosa rispetto ai thread di livello kernel
 - Consente anche ad un processo di scegliere una strategia di scheduling adatta alla sua natura.

- **CONTRO**

- Il kernel non sa della distinzione tra thread e processo
 - Al più un thread alla volta può essere operativo

- I thread di livello utente non forniscono parallelismo
 - La concorrenza è compromessa se un thread fa una syscall che porta a un blocco
-

MODELLO DI THREAD IBRIDI

Un modello ibrido è una combinazione di parallelismo (che manca a livello utente) e basso overhead (che manca a livello kernel). L'idea è avere più modelli ibridi:

- Molti a uno: a più thread di livello utente, è associato un thread di livello kernel.
- Uno a uno: ad ogni thread di livello utente corrisponde uno a livello kernel
- Molti a molti: a più thread di livello utente associo più thread di livello kernel

Il modello ibrido migliore è quello molti a molti teoricamente. Il problema che sorge però è la complessità dello sviluppo della gestione molti a molti.

Linux e UNIX adottano un mapping di uno a uno.

Lezione 26/03/2021

SINCRONIZZAZIONE DEI PROCESSI

Cos'è la sincronizzazione dei processi?

il termine processo è un termine generico usato sia per processi che per thread.

Durante il funzionamento di un SO per diversi motivi, ad ogni momento sono in esecuzione una moltitudine di processi kernel e

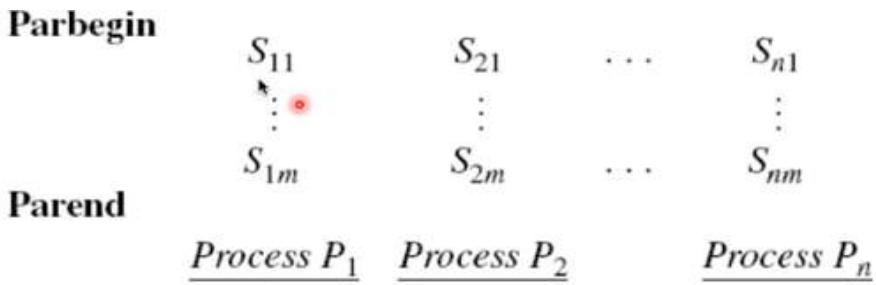
utenti. La cosa fondamentale è evitare che i processi accedano in maniera incontrollata a dati condivisi. Tutto ciò rende necessaria la sincronizzazione: i processi interagiscono tra loro e quando lo fanno devono coordinarsi in qualche modo. Introduciamo una notazione:

- ***read_set_i*** → insieme di dati letti dal processo P_i e messaggi interprocesso o segnali ricevuti da P_i
 - ***write_set_i*** → insieme di dati modificati dal processo P_i e messaggi interprocesso o segnali inviati da P_i
-
- Quando i processi devono scambiarsi info lo fanno con uno scambio di messaggi (*read set_i*).
 - *write_Set_i* → Indica i dati che il processo modifica e i messaggi che il processo invia a altri

Processi interagenti: i processi P_i e P_j sono interagenti se la *write_Set* di uno dei processi si sovrappone con la *write_Set* o la *read_Set* dell'altro. Se l'intersezione di questi insieme è vuota, i due processi sono indipendenti e non ho problemi di sincronizzazione.

CONVENZIONI IN PSEUDOCODICE PER I PROGRAMMI CONCORRENTI

Per rappresentare programmi concorrenti usiamo uno pseudocodice nel quale usiamo un costrutto di controllo chiamato “Parbegin” e “Parend”. Tutte le istruzioni sono eseguite in concorrenza.



Se ci sono variabili condivise su cui i processi operano simultaneamente, queste vanno dichiarate al di fuori del costrutto Parbegin. Le variabili locali sono dichiarate all'inizio di ciascun processo. I commenti sono fatti con le parentesi graffe. L'indentazione è usata per annidare i controlli.

RACE CONDITION

Se c'è un dato condiviso a cui può accedere un certo numero di processi, durante l'accesso dei processi concorrentemente al dato condiviso, può compromettere il valore del dato iniziale. Quando si verifica una situazione del genere si parla di RACE CONDITION su quella risorsa. Supponiamo di avere P_i e P_j e supponiamo che aggiornano il valore di d_s con le operazioni a_i e a_j :

- Operazione a_i : $d_s := d_s + 10$; Sia $f_i(d_s)$ il risultato
- Operazione a_j : $d_s := d_s + 5$; Sia $f_j(d_s)$ il risultato

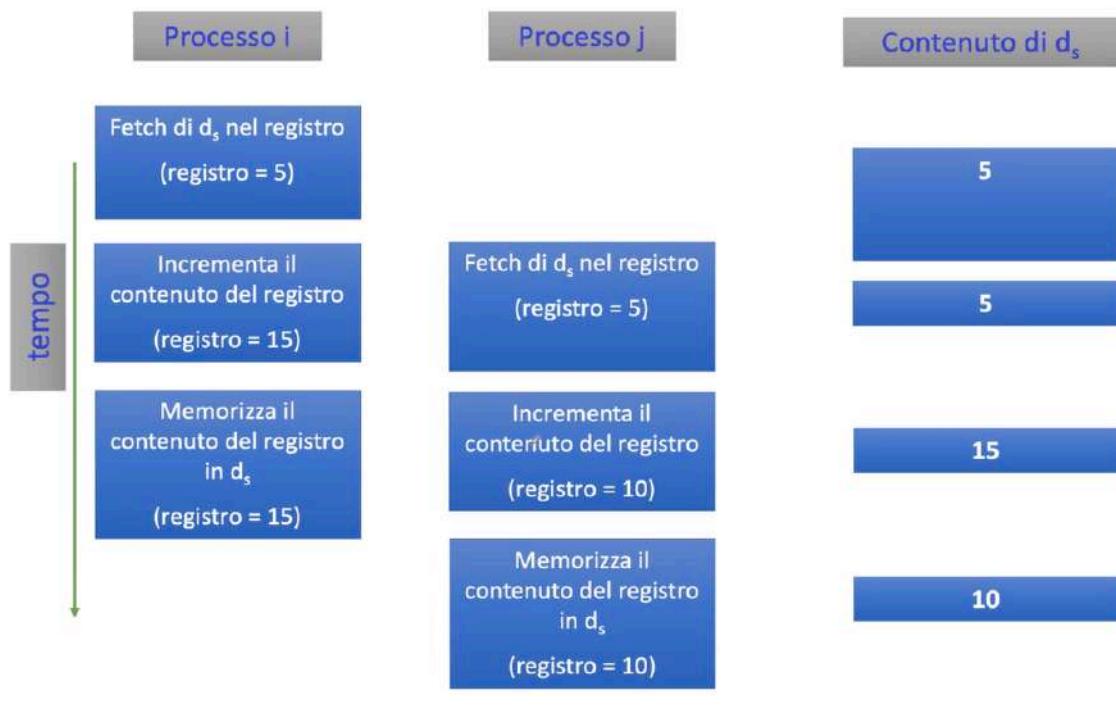
- Cosa succede se tali operazioni sono eseguite concorrentemente? (non si sa chi accedere per primo e non si può prevedere).

Race condition: Una condizione in cui il valore di un oggetto condiviso d_s , risultante dall'esecuzione delle operazioni a_i e a_j su d_s nei processi interagenti, può essere diversa da ambo $f_i(f_j(d_s))$ e $f_j(f_i(d_s))$

Risultato di incremento di p_i modificato poi da p_j

Risultato di incremento di p_j modificato poi da p_i

Due thread non sincronizzati che incrementano la stessa variabile



Esempio di Race Condition

Code of processes	Corresponding machine instructions																
$S_1 \quad \text{if } nextseatno \leq capacity$	$S_{1.1}$ Load $nextseatno$ in reg_k $S_{1.2}$ If $reg_k > capacity$ goto $S_4.1$																
$S_2 \quad \text{then}$																	
$S_3 \quad \text{allotedno} := nextseatno;$	$S_{2.1}$ Move $nextseatno$ to $allotedno$																
$S_3 \quad nextseatno := nextseatno + 1;$	$S_{3.1}$ Load $nextseatno$ in reg_j $S_{3.2}$ Add 1 to reg_j $S_{3.3}$ Store reg_j in $nextseatno$ $S_{3.4}$ Go to $S_5.1$																
$S_4 \quad \text{else}$																	
$S_4 \quad \text{display "sorry, no seats available"}$	$S_{4.1}$ Display "sorry, ..."																
$S_5 \quad \dots$	$S_{5.1}$...																
Some execution cases																	
Case 1 P_i	<table border="1"> <tr> <td>$S_{1.1}$</td><td>$S_{1.2}$</td><td>$S_{2.1}$</td><td>$S_{3.1}$</td><td>$S_{3.2}$</td><td>$S_{3.3}$</td><td>$S_{3.4}$</td><td></td><td>$S_{1.1}$</td><td>$S_{1.2}$</td><td>$S_{4.1}$</td></tr> </table>	$S_{1.1}$	$S_{1.2}$	$S_{2.1}$	$S_{3.1}$	$S_{3.2}$	$S_{3.3}$	$S_{3.4}$		$S_{1.1}$	$S_{1.2}$	$S_{4.1}$					
$S_{1.1}$	$S_{1.2}$	$S_{2.1}$	$S_{3.1}$	$S_{3.2}$	$S_{3.3}$	$S_{3.4}$		$S_{1.1}$	$S_{1.2}$	$S_{4.1}$							
Case 2 P_i	<table border="1"> <tr> <td>$S_{1.1}$</td><td>$S_{1.2}$</td><td></td><td>$S_{2.1}$</td><td>$S_{3.1}$</td><td>$S_{3.2}$</td><td>$S_{3.3}$</td><td>$S_{3.4}$</td><td></td><td>$S_{2.1}$</td><td>$S_{3.1}$</td><td>$S_{3.2}$</td><td>$S_{3.3}$</td><td>$S_{3.4}$</td></tr> </table>	$S_{1.1}$	$S_{1.2}$		$S_{2.1}$	$S_{3.1}$	$S_{3.2}$	$S_{3.3}$	$S_{3.4}$		$S_{2.1}$	$S_{3.1}$	$S_{3.2}$	$S_{3.3}$	$S_{3.4}$		
$S_{1.1}$	$S_{1.2}$		$S_{2.1}$	$S_{3.1}$	$S_{3.2}$	$S_{3.3}$	$S_{3.4}$		$S_{2.1}$	$S_{3.1}$	$S_{3.2}$	$S_{3.3}$	$S_{3.4}$				
Case 3 P_i	<table border="1"> <tr> <td>$S_{1.1}$</td><td>$S_{1.2}$</td><td>$S_{2.1}$</td><td>$S_{3.1}$</td><td></td><td>$S_{3.2}$</td><td>$S_{3.3}$</td><td>$S_{3.4}$</td><td></td><td>$S_{1.1}$</td><td>$S_{1.2}$</td><td>$S_{2.1}$</td><td>$S_{3.1}$</td><td>$S_{3.2}$</td><td>$S_{3.3}$</td><td>$S_{3.4}$</td></tr> </table>	$S_{1.1}$	$S_{1.2}$	$S_{2.1}$	$S_{3.1}$		$S_{3.2}$	$S_{3.3}$	$S_{3.4}$		$S_{1.1}$	$S_{1.2}$	$S_{2.1}$	$S_{3.1}$	$S_{3.2}$	$S_{3.3}$	$S_{3.4}$
$S_{1.1}$	$S_{1.2}$	$S_{2.1}$	$S_{3.1}$		$S_{3.2}$	$S_{3.3}$	$S_{3.4}$		$S_{1.1}$	$S_{1.2}$	$S_{2.1}$	$S_{3.1}$	$S_{3.2}$	$S_{3.3}$	$S_{3.4}$		
Execution of instructions by processes																	
Time →																	

Condivisione dei dati di processi di un'applicazione di prenotazione

Le race condition sono prevenute garantendo che le operazioni a_i e a_j non siano eseguite concorrentemente. Come si prevengono le race condition su azioni svolte concorrentemente?

- Posso introdurre il concetto di **mutua esclusione**: nel momento in cui ho un processo che accede a una variabile condivisa d_s , nel momento in cui quel processo sta operando su quella variabile condivisa, nessun altro processo deve avere la possibilità di accedervi.
 - La sincronizzazione per l'accesso ai dati è il coordinamento dei processi per implementare la mutua esclusione su dati condivisi.
- Con la mutua esclusione è assicurato che il risultato delle operazioni a_i e a_j sarà $f_i(f_j(d_s))$ oppure $f_j(f_i(d_s))$

- Definiamo
 - $update_set_i$ -> insieme di dati aggiornati dal processo P_i (letti, modificati e scritti)
 - Per prevenire una race condition:
 - Si controlla che la logica dei processi causa una race
 - $update_set_i \cap update_set_j \neq \emptyset$
 - Es.: nel sistema di prenotazione aereo
 - $update_set_i = update_set_j = \{\text{nextseatno}\}$
 - Noto il dato su cui c'è una race
 - Si usano tecniche di sincronizzazione che implementano la mutua esclusione per l'accesso ai dati
-

SEZIONI CRITICHE

La mutua esclusione è implementata usando le sezioni critiche di codice.

Sezione critica (SC): una sezione critica per un oggetto d_s è una sezione di codice che è progettata in modo che non possa essere eseguita concorrentemente con sé stessa o con altre sezioni critiche per d_s

Nei codici di esempio, una SC è indicata con un rettangolo grigio.

- Se un processo P_i sta eseguendo una SC per d_s , un altro processo che intendesse eseguire una SC per d_s dovrebbe attendere la fine dell'esecuzione della SC di P_i
 - Una SC per un dato d_s , è una regione di mutua esclusione rispetto agli accessi a d_s
-

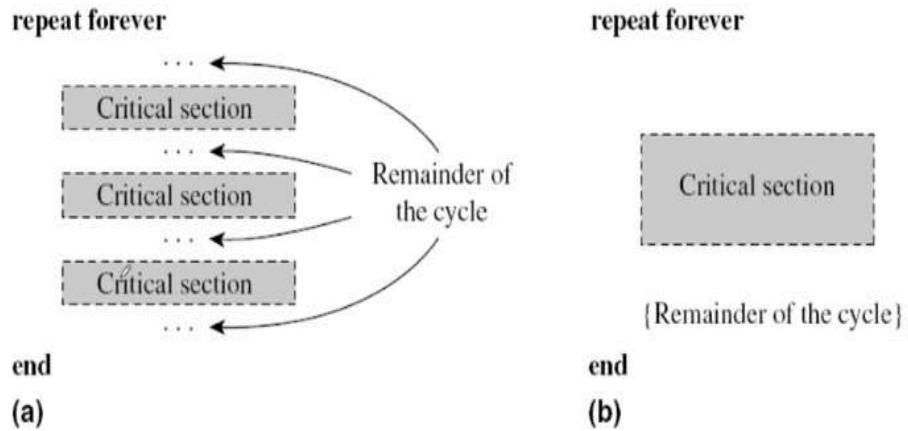


Figure 6.3 (a) A process with many critical sections; (b) a simpler way of depicting this process.

```

if nextseatno ≤ capacity
then
    allottedno:=nextseatno;
    nextseatno:=nextseatno+1;
else
    display "sorry, no seats
            available";

```

Process P_i


```

if nextseatno ≤ capacity
then
    allottedno:=nextseatno;
    nextseatno:=nextseatno+1;
else
    display "sorry, no seats
            available";

```

Process P_j

Uso di sezioni critiche in un sistema di prenotazione aereo

Usare SC causa ritardi nelle operazioni dei processi:

- Un processo non deve essere eseguito a lungo in un a SC
- Il processo non deve invocare sycall che possono portarlo in blocked, all'interno di una SC
- Il kernel non deve prelazionare un processo che è alle prese con le sezione di una SC
 - Il kernel dovrebbe essere sempre informato se un processo è in SC
 - Non è possibile implementarlo nel caso un processo implementi una SC autonomamente (senza che il kernel lo sappia)
- Assumeremo che un processo trascorra poco tempo in una SC.

PROPRIETA' IMPLEMENTAZIONE DI UNA SC

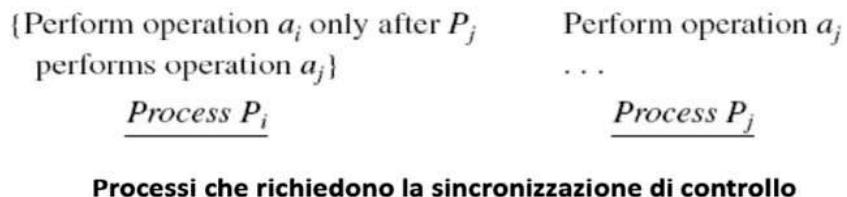
Quando implementiamo una SC dobbiamo assicurarci che questa soddisfi 3 proprietà fondamentali:

- Mutua esclusione: nel momento in cui ho n processi che possono accedere a sezione critica, solo un processo può stare in SC, gli altri devono attendere.
- Progresso: se non c'è nessun processo in SC per un dato d_s , se c'è in quel momento un processo che intende entrare in SC bisogna farlo entrare subito. In poche parole la SC non può essere riservata a nessun processo e deve essere pronta ad accogliere qualsiasi processo il prima possibile.
- Attesa limitata: dopo che un processo P_i ha indicato la volontà di entrare in SC per una variabile condivisa d_S , il numero di volte che altri processi possono entrare nella SC prima di P_i deve essere limitato. Se ho n processi che vogliono entrare in SC devo garantire che al più un numero finito di processi. In poche parole un processo non può aspettare molto se ha espresso volontà di SC.

Il processo e l'attesa limitata insieme prevengono la **starvation** (quando un processo non riesce a entrare)

Sincronizzazione di Controllo e Operazioni Indivisibili

- I processi interagenti devono coordinare la loro esecuzione uno rispetto all'altro, per eseguire le rispettive azioni nell'ordine desiderato
 - Requisito soddisfatto mediante la sincronizzazione di controllo



- La segnalazione è una tecnica generale di sincronizzazione di controllo

```
var
    operation_aj_performed : boolean;
    pi_blocked : boolean;
begin
    operation_aj_performed := false;
    pi_blocked := false;

    Parbegin
        ...
        if operation_aj_performed = false
        then
            pi_blocked := true;
            block (Pi);
            {perform operation aj}
        ...
        ...
    ...
    Parend;
end.
```

Process P_i Process P_j

tentativo di segnalazione naive mediante variabili booleane

Una segnalazione *naive* non funziona. P_i potrebbe bloccarsi indefiniteamente in alcune situazioni.

Race condition nella sincronizzazione di Processi

Time	Actions of process P_i	Actions of process P_j
t_1	if $action_aj_performed = false$	
t_2		{perform action aj }
t_3		if $pi_blocked = true$
t_4		$action_aj_performed := true$
:		
t_{20}	$pi_blocked := true;$	
t_{21}	$block(P_i);$	

- Usare invece operazioni atomiche o indivisibili

Operazione indivisibile: un'operazione su un insieme di oggetti che non può essere eseguita concorrentemente con sé stessa o altra operazione su un oggetto incluso nell'insieme.

Quando inizia un processo questo deve finire e non può essere interrotto in alcun modo. (Operazione Indivisibile)

```

procedure check_aj
begin
  if  $operation\_aj\_performed=false$ 
  then
     $pi\_blocked:=true;$ 
     $block(P_i)$ 
  end;

procedure post_aj
begin
  if  $pi\_blocked=true$ 
  then
     $pi\_blocked:=false;$ 
     $activate(P_j)$ 
  else
     $operation\_aj\_performed:=true;$ 
  end;

```

Operazioni indivisibili $check_aj$ e $post_aj$ per la segnalazione

APPROCCI ALLA SINCRONIZZAZIONE

- CICLARE VS BLOCCARE

Una SC in generale per una data variabile condivisa d_s sostanzialmente è un'operazione equivalente a un'istruzione indivisibile. Quando ho più processi che devono accedere in SC, se uno vuole accedervi ma già c'è un altro, deve aspettare.

Abbiamo due modi di aspettare:

- **Busy wait(Attesa attiva)**: quando un processo è in SC o sta eseguendo un'operazione indivisibile, non fare nulla.

```
while (some process is in a critical section on {ds} or  
       is executing an indivisible operation using {ds})  
{ do nothing }
```

Critical section or
indivisible operation
using {d_s}

- Un'attesa attiva ha molte conseguenze:
 - o Non può fornire la proprietà di attesa limitata
 - o Degrado delle prestazioni a causa del ciclare
 - o Deadlock
 - o Inversione di priorità

Per evitare le attese attive, un processo in attesa di entrare in una SC è posto in blocked.

Verrà cambiato in ready solo quando può entrare nella SC.

```
if (some process is in a critical section on {ds} or  
       is executing an indivisible operation using {ds})  
then make a system call to block itself;
```

Il processo decide se ciclare o bloccarsi: la decisione è soggetta a race condition. Adottiamo approcci algoritmici per poter eseguire

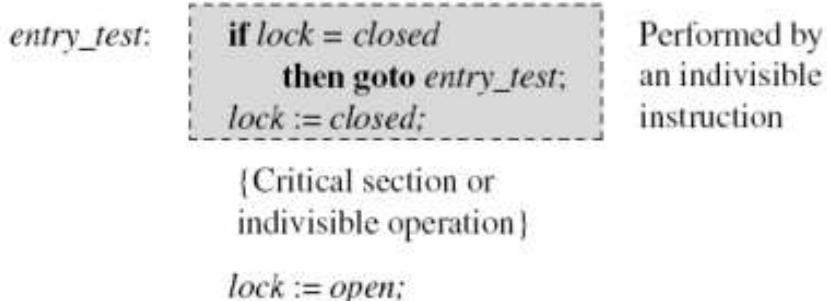
questo tipo di operazioni in mutua esclusione, o possiamo usare istruzioni messe a disposizione dalla macchina a livello HW.

SUPPORTO HW PER LA SINCRONIZZAZIONE DI PROCESSI

L'HW fornisce istruzioni indivisibili che evitano race condition sulle locazioni di memoria.

Queste istruzioni funzionano usando una variabile di LOCK necessaria per implementare la SC e le operazioni indivisibili.

L'idea è: uso la variabile LOCK che può indicare aperto o chiuso (0-1). Se lock è chiuso allora ripeto il test, altrimenti chiudo il lock se era aperta e poi eseguo la sezione critica, quando ho finito, riapro il lock.



- **entry_test** eseguita con un'istruzione indivisibile
 - istruzione Test-and-set (TS)
 - Istruzione swap

ISTRUZIONE TEST AND SET

L'istruzione test and set (TS) imposta il contenuto di una locazione di memoria ad 1 e restituire il valore precedente. Nel caso dell'esempio la TS usa la variabile LOCK assegnandogli il valore '00' che significa OPEN. L'operazione TS LOCK non fa altro che porre LOCK a 1 e ritornare il valore precedente, impostando il valore nel campo CC nel PSW, queste due operazioni sono atomiche(indivisibili). Vado a vedere quant'era il valore

precedentemente di lock: se era 0, il risultato sarà 0 e si procede se invece è 1 significa che il lock era occupato e saltiamo su entry test.

<u>LOCK</u>	DC X'00'	Lock is initialized to open
<u>ENTRY TEST</u>	TS <u>LOCK</u>	Test-and-set lock
	BC 7, <u>ENTRY TEST</u>	Loop if lock was closed
	...	{ Critical section or indivisible operation }
	MVI LOCK, X'00'	Open the lock (by moving 0s)

Vediamo il codice con un linguaggio ad alto livello

Esempio di Uso di TestAndSet

```
bool TestAndSet (bool &target) {
    bool retValue = target;
    target = True;
    return retValue;
}
```

critical section with test and set

Shared state:

```
lock = False
```

Thread one code:

```
1: while test_and_set(lock):
2:   do nothing
3: # critical section
4: lock = False
```

Thread two code: (same)

```
5: while test_and_set(lock):
6:   do nothing
7: # critical section
8: lock = False
```

SWAP

È un'istruzione che usa lock ma invece di operare su lock, usa una variabile temporanea che imposta a 1 e poi swappa lock con temp, questo scambio non deve essere interrotto. Vado a vedere lock prima, se era aperto allora salto e vado alla sezione critica e infine riapro lock, se era chiuso allora salto a entry test e vado a ciclare.

TEMP	DS	1	Reserve one byte for TEMP
LOCK	DC	X'00'	Lock is initialized to open
	MVI	TEMP, X'FF'	X'FF' is used to close the lock
<u>ENTRY_TEST</u>	<u>SWAP</u>	LOCK, TEMP	
	COMP	TEMP, X'00'	Test old value of lock
	BC	1, ENTRY_TEST	Loop if lock was closed
	(...)		{ Critical section or indivisible operation }
	MVI	LOCK, X'00'	Open the lock

Linguaggio ad alto livello:

```
bool lock;
bool key;
void swap(bool &a, bool &b) {
    boolean temp = a;
    a=b;
    b=temp;
}
```

```
while(1) {
    key = true;
    while(key)
        swap(lock, key);
    /* SC */
    lock = false;
}
```

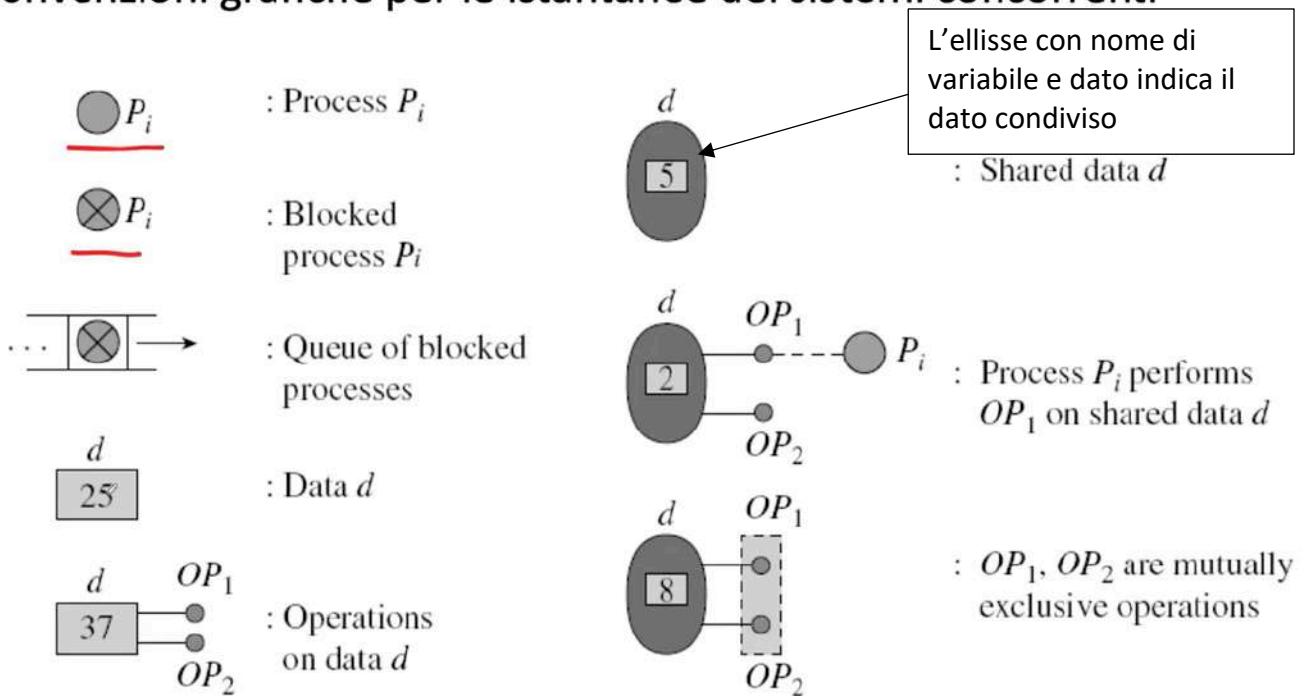
APPROCCI ALGORITMICI, PRIMITIVE DI SINCRONIZZAZIONE E COSTRUTTI DI PROGRAMMAZIONE CONCORRENTE

Strutture dei sistemi concorrenti

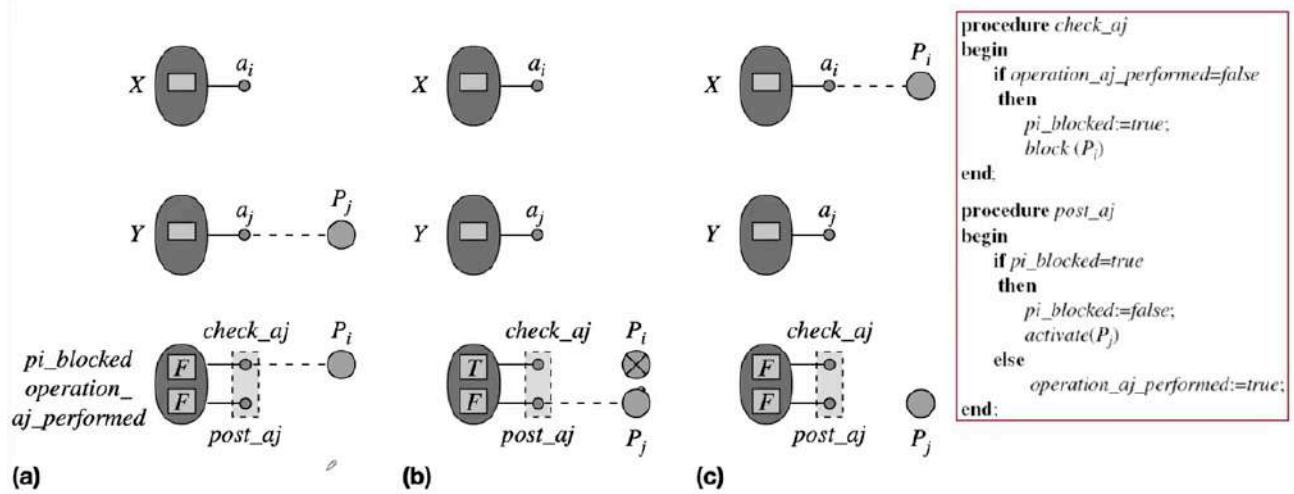
Abbiamo 3 componenti: dati condivisi e dati di sincronizzazioni, le operazioni sui dati condivisi e dei processi interagenti.

Un'istantanea (snapshot) di un sistema concorrente è una vista del sistema in un istante specifico.

Convenzioni grafiche per le istantanee dei sistemi concorrenti



Esempio:



- (a) P_i esegue l'operazione `check_aj`
- (b) P_i è bloccato; P_j esegue l'operazione `post_aj`
- (c) `post_aj` attiva P_i ; P_j esce da `post_aj`

PROBLEMI CLASSICI DI SINCRONIZZAZIONE

La maggior parte dei problemi di sincronizzazione si rifanno alle soluzioni dei problemi classici di cui parleremo. Quando si

risolvono problemi di sincronizzazione bisogna avere una certa esperienza per associare il problema a uno classico. Una soluzione di un problema di sincronizzazione deve soddisfare tre criteri importanti:

1. Correttezza: operazione in mutua esclusione
2. Massima concorrenza: le soluzioni devono garantire che tutti i processi siano in esecuzione, senza lasciare nessun processo
3. Nessuna attesa attiva: evitare il busy waiting

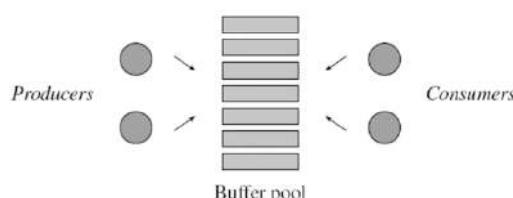
Alcuni problemi classici:

- Produttori-Consumatori con buffer limitati
 - Lettori e scrittori
 - Filosofi a cena
-

PRODUTTORI-CONSUMATORI CON BUFFER LIMITATI

Abbiamo un insieme di buffer e un numero di processi definiti produttori (processi che memorizzano info nei buffer) e un insieme di processi consumatori che leggono le info dei produttori e le prelevano. Questa è una situazione che ha race condition. Il buffer è soggetto a race condition poiché produttori e consumatori modificano i dati. Dobbiamo sincronizzare i due processi e dobbiamo soddisfare:

1. Un produttore non deve sovrascrivere un buffer pieno. Se è pieno non può produrre.
2. Un consumatore non può consumare un buffer vuoto.
3. Produttori e consumatori devono accedere ai buffer in modo mutuamente esclusivo
4. Le info devono essere consumate nello stesso ordine in cui è messa nei buffer



POSSIBILE SOLUZIONE

Quello che si può fare è utilizzare una variabile booleana per indicare se un processo produttore ha prodotto o meno.

Prodotto viene impostato a false. Finché prodotto è false si va a verificare lato produttore se esiste una locazione dell'array buffer vuoto. Se è così produce all'interno di quella locazione e pone produced = true. Questo controllo deve avvenire in mutua esclusione. Dal lato del consumatore: consumed=false e controllo se c'è il buffer pieno. Se è così posso consumare e pongo consumed uguale a true.

Questa prima soluzione non è il massimo poiché quando costruiamo una soluzione di un problema di sincronizzazione dobbiamo garantire una massima concorrenza. In questo caso non la garantiamo perché abbiamo un processo produttore deve visitare tutto l'array per controllare una posizione vuota e chiaramente questa operazione limita la possibilità ai processi consumatori di poter procedere perché se ho locazioni piene queste possono essere consumate indipendentemente dal produttore e viceversa per il controllo fatto dai consumatori. L'altro problema è l' *if* fatto nel producer. Quell'*if* è un'attesa attiva che va a ciclare (busy waiting).

Chiaramente vale anche per l'*if* fatto dal lato consumatore.

```
begin
  Parbegin
    var produced : boolean;
    repeat
      produced := false
      while produced = false
        if an empty buffer exists
        then
          { Produce in a buffer }
          produced := true;
        { Remainder of the cycle }
      forever;
    Parend;
  end.

Producer
```

```
var consumed : boolean;
repeat
  consumed := false;
  while consumed = false
    if a full buffer exists
    then
      { Consume a buffer }
      consumed := true;
    { Remainder of the cycle }
  forever;

Consumer
```

- Come migliorare lo schema precedente?
 - Dobbiamo continuare ad accedere in mutua esclusione all’array di buffer però dobbiamo segnalare al consumatore che un produttore ha prodotto e c’è un elemento a disposizione e viceversa il consumatore deve segnalare al produttore che c’è una locazione libera.

In questo schema per semplicità consideriamo una soluzione con un solo produttore, un solo buffer e un solo consumatore.

SOLUZIONE

Usiamo il seguente schema.

Abbiamo la variabile booleana *buffer full* per indicare che il buffer è stato riempito e usiamo un’altra booleana per indicare quando il processo produttore e consumatore sono *blocked*.

Inizializziamo tutto a false. Devo utilizzare operazioni non interrompibili.

Lato produttore:

Vado a controllare se il buffer è vuoto, se è così posso produrre all’interno del buffer. Fatto ciò, devo segnalare al consumatore che il buffer è pieno tramite *post_b_full*

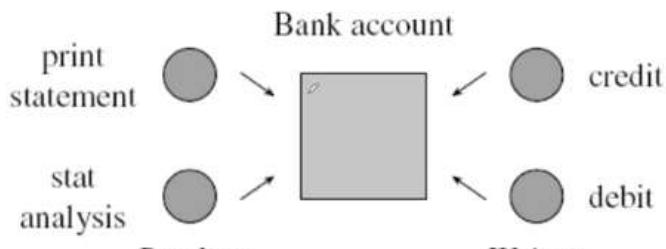
Lato consumatore:

se il buffer è pieno allora può consumare e va a informare il produttore che c’è una locazione vuota.

Nel caso in cui il buffer fosse pieno, il produttore si blocca in attesa di ricevere il messaggio da parte del consumatore e viceversa.

LETTORI E SCRITTORI

Abbiamo un numero di processi che accedono in lettura ad un dato condiviso insieme a uno o più scrittori.



Lettori e scrittori in un sistema bancario

Immaginiamo un conto bancario (dato condiviso), gli scrittori sono i processi che aggiornano il credito e i lettori sono i processi per sapere quant'è il conto.

Per avere una soluzione al problema devo rispettare dei vincoli:

1. I lettori possono leggere concorrentemente. Non c'è problema se più lettori vogliono leggere il dato,
2. La lettura è proibita mentre uno scrittore sta scrivendo
3. Solo uno scrittore può eseguire la scrittura in un dato momento
4. Un lettore ha priorità non prelazionabile sugli scrittori (sistema lettori-scrittori con preferenza ai lettori)

POSSIBILE SOLUZIONE

Lato lettore:

se uno scrittore sta scrivendo allora deve aspettare. Se nessuno sta scrivendo posso andare direttamente a leggere.

se nessun altro sta leggendo, controllo se c'è qualche scrittore in attesa, lo vado ad attivare in modo che possa andare a scrivere.

Lato scrittore:

se i lettori stanno leggendo o uno scrittore sta scrivendo, deve aspettare.

Se non è così può scrivere. L'operazione di scrittura deve avvenire in MUTUA ESCLUSIONE. Se ci sono poi lettori o scrittori in attesa, posso attivare uno scrittore o tutti i lettori e continuo a ciclare.

Parbegin

repeat

If a writer is writing

then

{ wait };

{ read }

If no other readers reading

then

if writer(s) waiting

then

activate one waiting writer;

forever;

Parend;

end.

repeat

If reader(s) are reading, or a

writer is writing

then

{ wait };

{ write }

If reader(s) or writer(s) waiting

then

activate either one waiting
writer or all waiting readers;

forever;

Reader(s)

Writer(s)

La cosa importante è che questa soluzione garantisce la mutua esclusione e ho una attesa limitata poiché dopo un'operazione l'attesa è molto breve.

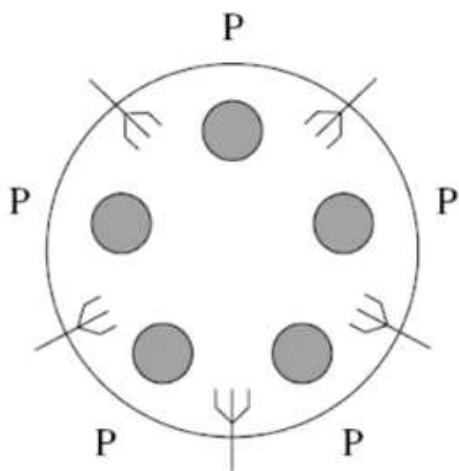
FILOSOFI A CENA

Abbiamo una tavola rotonda dove ho un insieme di filosofi che mangiano e pensano.

Bisogna garantire che se un filosofo ha fame, deve poter mangiare e non muoia di inedia. La possibilità di mangiare è dettata dal fatto che un dato filosofo possa prendere la forchetta

che i trova alla sua destra e alla sua sinistra. Quindi deve avere le due forchette per poter mangiare. Essendo le forchette condivise, è ovvio che ci possa essere una situazione per la quale tutti non hanno la possibilità di prendere le forchette (**deadlock**).

La soluzione non deve incorrere in deadlock (i filosofi non avendo la possibilità di mangiare, si bloccano a vicenda) o **livelock** (non ho i processi bloccati però i processi favoriscono sempre gli altri)



POSSIBILE SOLUZIONE

Ciascun filosofo preleva una forchetta per volta.

Controllo se la forchetta di sinistra è disponibile, se non lo è mi blocco. Se è disponibile la prendo e devo controllare anche quella di destra. Se ho preso anche quella di destra posso mangiare. Una volta mangiato poso ambo le forchette. Se il filosofo alla mia sinistra sta aspettando per la forchetta di destra, lo attivo oppure controllo il filosofo alla destra, se era in attesa della sinistra lo vado ad attivare. Fatto ciò, il mio filosofo si ferma.

Qual è il problema di questo tipo di soluzione?

Se una forchetta non è disponibile, il filosofo si blocca: potrei prendere quella a sinistra e comunque bloccarmi con quella di

destra! Se questo capita a tutti, tutti hanno una forchetta e tutti rimangono bloccati (deadlock).

Se la forchetta di destra non è disponibile, prima di bloccarmi, potrei rilasciare la forchetta di sinistra e poi bloccarmi. In questo caso evito il deadlock ma non evito il livelock perché rischio di favorire sempre gli altri.

SOLUZIONE MIGLIORE

Utilizzo una variabile booleana per verificare se ho acquisito entrambe le forchette. Operazione in mutua esclusione e atomica: finché non prendiamo entrambe le forchette, se entrambe sono disponibile allora le prendo e pongo la booleana true. Fatto ciò, esco dal while e posso mangiare, una volta mangiato, pongo ambo le forchette. Se il mio vicino di destra e/o quello di sinistra stavano in attesa, li attivo e mi metto a pensare. In questo caso evito deadlock e livelock.

Il problema è che questo ciclo è busy waiting perché il loop causa una condizione di attesa attiva.

APPROCCI ALGORITMICI PER LE SEZIONI CRITICHE

Gli approcci algoritmici per implementare le SC non impiegano i servizi del kernel, né le istruzioni HW.

Indipendenti dal SO e dall'HW, tuttavia, usano il busy waiting e usano complesse organizzazioni logiche per evitare la race condition.

ALGORITMI A DUE PROCESSI

Problema con due processi a disposizione

Prima soluzione

```
var      turn : 1 .. 2;
begin
    turn := 1;
    Parbegin
        repeat
            while turn = 2
                do { nothing };
                { Critical Section }
            turn := 2;
                { Remainder of the cycle }
        forever;
    Parenend;
end.
```

Process P₁

turn indica il prossimo processo che entra in SC

```
repeat
    while turn = 1
        do { nothing };
        { Critical Section }
    turn := 1;
        { Remainder of the cycle }
    forever;
```

Process P₂

Questa soluzione usa una variabile “turn” che detta qual è il prossimo processo che deve entrare in sezione critica. Turn può avere valore 1 e 2 poiché ho due processi.

Iniziamo con turn=1.

Processo p1:

p1 controlla turn, se è uguale a 2 allora non deve fare nulla e aspettare (Attesa attiva). Se non è vero allora entro in SC e una volta uscito pongo turn=2 per dare possibilità all'altro processo di entrare in SC

processo p2:

uguale a p1.

Questa soluzione non è ottimale poiché viola la condizione del progresso (proprietà della sezione critica):

supponiamo che p1 sia in SC e supponiamo che p2 sia all'interno del ciclo facendo altro. Se p2 sta lì e p1 esce, p1 imposta turn=2, va anche lui nel ciclo e potrebbe tornare su al while mentre p2 è ancora nel ciclo. Siccome ora turn=2 (while), p1 cicla aspettando p2 che entri ma p2 non è in SC e quindi in questo momento ho un processo p1 che vuole entrare in SC ma non può, anche se non c'è nessun processo in SC. Violata la condizione del progresso.

Seconda soluzione

```

var      c1, c2 : 0..1;
begin
    c1 := 1;
    c2 := 1;
Parbegin
    repeat
        while c2 = 0
            do { nothing };
        c1 := 0;
        { Critical Section }
        c1 := 1;
        { Remainder of the cycle }
    forever;
Parend;
end.

```

Process P₁

c_i variabili di stato.
c_i = 0 indica quando P_i è in SC
c_i = 1 quando è fuori dalla SC

```

repeat
    while c1 = 0
        do { nothing };
    c2 := 0;
    { Critical Section }
    c2 := 1;
    { Remainder of the cycle }
forever;

```

Process P₂

Per evitare problemi di progresso, si utilizza una variabile di stato che indica se un dato processo si trova in SC oppure no. Ogni processo ha la propria variabile di stato c_i.
Se c_i = 0 allora il processo è in SC, se 1 è fuori dalla SC.

Lato Processo1

While il processo p2 si trova in SC non devo fare nulla. Quando il processo p2 non è in SC allora p1 va in SC e dopo la SC pongo c₁ = 1 e vado nella parte restante del ciclo.

Lato Processo2

Uguale al processo 1.

In questo caso ho risolto il problema del progresso poiché segnalo la SC vuota o piena in modo che ci sia sempre un processo pronto a entrare nella SC.

-Viene violata però la mutua esclusione: p1 e p2 vogliono entrare contemporaneamente nella SC.

Accade che entrambi i processi hanno la possibilità di entrare e questo non soddisfa la SC.

Potremmo scambiare il while mettendo la verifica di c_1 prima del while ma questo potrebbe portare comunque al deadlock per lo stesso ragionamento.

Entrambi si bloccano sul while.

ALGORITMO DI DEKKER

Combina le soluzioni dei primi due algoritmi.

Utilizza sia la variabile “turn” sia la variabile di stato c_i .

- Se i due processi vogliono entrare contemporaneamente, a chi è consentito viene determinato dalla variabile “turn” se in un istante c’è solo 1 processo che vuole entrare, turn non ha importanza.

- Se entrambi i processi tentano di entrare nelle SC, turn forza uno dei due favorisca l’altro processo.

Algoritmo di Dekker

```
var      turn : 1 .. 2;  
        c1, c2 : 0 .. 1;  
begin  
    c1 := 1;  
    c2 := 1;  
    turn := 1;  
Parbegin  
    repeat  
        c1 := 0;  
        while c2 = 0 do  
            if turn = 2 then  
                begin  
                    c1 := 1;  
                    while turn = 2  
                        do { nothing };  
                    c1 := 0;  
                end;  
                { Critical Section }  
                turn := 2;  
                c1 := 1;  
                { Remainder of the cycle }  
            forever;  
Parend;  
end.
```

turn è efficace solo quando ambo i processi cercano di entrare nella SC nello stesso tempo

```
repeat  
    c2 := 0;  
    while c1 = 0 do  
        if turn = 1 then  
            begin  
                c2 := 1;  
                while turn = 1  
                    do { nothing };  
                c2 := 0;  
            end;  
            { Critical Section }  
            turn := 1;  
            c2 := 1;  
            { Remainder of the cycle }  
        forever;
```

Process P₁

Process P₂

Inizializziamo c₁ e c₂ uguali 1 poiché siamo fuori dalla SC.
Impostiamo turn uguale a 1 e iniziamo.

Processo p1:

c₁=0 però se anche c₂ vuole entrare in SC allora vedo il valore di turn. Se turn=2 allora c₁ resta fuori dalla SC e attende finché turn non va a 2. Quando questo non è più vero allora c₁=0 ed entra in SC. una volta fatto ciò turn=2 e c₁=1 e vado avanti con il processo.

Lo stesso avviene col processo p2.

ALGORITMO DI PETERSON

Utilizza un array booleano di flag (un flag per processo).

Un processo imposta il flag a true quando intende entrare in SC e lo imposta a false quando ne esce.

Turn è usata per evitare i livelock.

Si suppone che i due processi siano P_0 e P_1 e gli id (0 e 1) sono usati per accedere ai flag di stato.

```

var      flag : array [0 .. 1] of boolean;
          turn : 0 .. 1;
begin
    flag[0] := false;
    flag[1] := false;
Parbegin
    repeat                                repeat
        flag[0] := true;
        turn := 1;
        while flag[1] and turn = 1
            do {nothing};
            { Critical Section }
            flag[0] := false;
            { Remainder of the cycle }
            forever;
Parend;
end.

```

Process P_0

Process P_1

Inizialmente i flag sono false per entrambi.

Processo P_0 :

pone il suo flag a true per indicare che vuole entrare in SC. *turn* non è usata per dire semplicemente a chi tocca ma per favorire esplicitamente l'altro. In questo modo se P_1 voleva entrare, ha la possibilità di farlo. Finché l'altro processo vuole entrare e $turn = 1$, aspetto. Quando la condizione non è più vero, P_0 entra in SC e pone il suo flag a 0.

Lo stesso vale per P_1

Soluzioni con n processi.

Algoritmo del panettiere

Le varie soluzioni contemplano che un processo provi a favorire l'altro per entrare in SC.

L'idea dell'algoritmo del panettiere è che il processo che deve entrare in SC deve prendere un “numeretto” e poi il processo con il numero minore entra in SC.

Si dice che il processo viene “servito” quando entra in SC.

Abbiamo bisogno di due strutture dati, 2 array.

- *choosing*[0 ... $n - 1$], array booleano, e l'choosing[i] indica se P_i è impegnato nella scelta
- *number*[0 ... $n - 1$], dove number[i] contiene il numero scelto di P_i .

Per ovviare al problema di due numeri uguali consideriamo ciò:
È servito il processo che ha la coppia (number[i], i) minore, dove:

```
(number[j],j) < number[i],i) se  
    number[j] < number[i], oppure  
    number[j] = number[i] and j < i
```

Algoritmo del Panettiere (cont.)

```
const n = ... ;
var choosing : array [0 .. n - 1] of boolean;
    number : array [0 .. n - 1] of integer;
begin
    for j := 0 to n - 1 do
        choosing[j] := false;
        number[j] := 0;
Parbegin
    process Pi :
        repeat
            choosing[i] := true;
            number[i] := max (number[0], .. ,number[n - 1]) + 1;
            choosing[i] := false;
            for j := 0 to n - 1 do
                begin
                    while choosing[j] do { nothing };
                    while number[j] ≠ 0 and (number[j], j) < (number[i], i)
                        do { nothing };
                end;
                { Critical Section }
                number[i] := 0;
                { Remainder of the cycle }
            forever;
        process Pj : ...
Parend;
end.
```

Cosa accade se non usiamo l'array choosing?

Supponiamo di avere solo 2 processi, P_0 e P_1 . Sono eseguiti simultaneamente ed entrambi arrivano ad eseguire la scelta del numero. Entrambi eseguono max ma non hanno ancora aggiunto 1. Significa che P_0 sceglie max tra number di 0 e 1 ma il max sarà

0, lo stesso anche P_1 . Il massimo di entrambi è zero. Supponiamo che P_1 proceda, viene schedulato e P_0 prelazionato. All'interno di `number[1]` viene incrementato di 1. Il processo P_1 procede e va a controllare se ci sono altri processi con numeri inferiori al suo, siccome P_0 non ha ancora scelto il numero, il processo P_1 salta il ciclo `while` e va in SC. P_1 viene prelazionato e schedulato P_0 che ancora non veniva incrementato. Viene incrementato e anche in `number[0]` c'è 1. Il problema è che P_0 va avanti, esegue il `while` e lui verifica che anche per P_1 c'è un numero diverso da 0 e controlla la seconda condizione, per quanto riguarda P_0 ha la coppia più piccola (P_0 ha 1 e P_1 ha 1 ma P_0 ha l'indice minore), la condizione non è vera, salto il ciclo `while` e vado in SC. La condizione è violata.

L'array `choosing` garantisce la mutua esclusione della SC.

SINCRONIZZAZIONE DEI PROCESSI

Nel corso degli anni sono state inserite primitive di sincronizzazione per semplificare i problemi. La prima primitiva è rappresentata dai SEMAFORI.

Un semaforo è una variabile intera condivisa con valori non negativi che può essere soggetta alle sole operazioni che seguono:

1. Inizializzazione (specificato come parte della sua dichiarazione)

<ol style="list-style-type: none"> 2. Operazioni indivisibili <i>wait</i> e <i>signal</i> (o <i>post</i>), atomiche <pre>procedure wait(S) begin while S<=0 do {nothing}; S:=S-1; end;</pre>	<pre>procedure signal(S) begin S:=S+1; end;</pre>
---	---

Dobbiamo garantire l'atomicità delle operazioni ed evitare la busy wait.

Implementazione dei Semafori (NO valori negativi)

```
wait(semaphore *S) {  
  
    if (S->value > 0)  
        S->value--;  
    else {  
        aggiungi P a S->list;  
        block();  
    }  
}
```

```
signal(semaphore *S) {  
  
    if (qualche P bloccato su S)  
        togli P da S->list;  
        wakeup(P);  
    }  
    else  
        S->value++;  
}
```

```
typedef struct{  
    int value;  
    struct process *list;  
} semaphore;
```

In che modo si usa il semaforo?

Il semaforo è usato associato ad una risorsa condivisa che vogliamo proteggere. Lo si usa sia per la mutua esclusione, si per garantire l'accesso a una risorsa condivisa. Supponiamo di avere un sistema con locazioni condivise. Se le mie locazioni libere sono n, mi basta usare un semaforo inizializzato a n. se ho processi concorrenti che vogliono accedere a queste risorse condivise, invoco una funzione wait sul semaforo: il processo wait controlla il contatore, se è maggiore di 0 significa che ho risorse libere, vi

accedo a queste e decremento il contatore delle risorse. Finché un processo invoca wait e il contatore è maggiore di 0 si può accedere. Quando i miei processi hanno acquisito le n risorse e sono finite, se arriva un altro processo, questo viene bloccato finché non ci sarà nuovamente una risorsa disponibile, quando il valore del semaforo è maggiore di 0 cioè quando i processi rilasciano la risorsa, attraverso la funzione *signal*. Quando uso signal il processo vede se c'è qualche processo bloccato e in attesa, se così lo riattivo oppure se non c'è nessuno in attesa, magari perché tutti hanno fatto accesso e nessun altro ha fatto wait e si è bloccato, semplicemente incremento il valore del contatore per indicare che ora una risorsa è disponibile.

Implementazione dei Semafori (**SI** valori negativi)

```
wait(semaphore *S) {  
    S->value--;  
    if (S->value < 0) {  
        aggiungi P a S->list;  
        block();  
    }  
}
```

```
signal(semaphore *S) {  
    S->value++;  
    if (S->value <= 0) {  
        togli P da S->list;  
        wakeup(P);  
    }  
}
```

Il contatore qui può assumere anche valori negativi. Prima decremento il valore del semaforo e poi vado a vedere il valore. Ovviamente il tutto funziona in maniera equivalente. La signal prima incrementa il contatore e se il valore è minore o uguale di zero, ci sono processi bloccati in attesa, uno lo devo togliere dalla lista e lo devo risvegliare.

USO DEI SEMAFORI NEI SISTEMI CONCORRENTI

- Usati per la mutua esclusione: mi basta inizializzare un semaforo a 1. A questo punto voglio implementare la mutua esclusione per entrare in SC con 2 processi. Il primo processo invoca wait e avrà il contatore a 1. Va in SC in mutua esclusione e viene decrementato il valore del semaforo. Se ho un altro processo questo fa una wait al semaforo ma questo viene bloccato poiché il contatore è 0. Si sblocca solo quando l'altro processo esce dalla SC e invoca una signal sul processo e questo viene risvegliato ed entra in SC. Si usano i semafori binari (non diversi dagli altri). Il valore del contatore varia solo tra 0 e 1.
- Concorrenza limitata: se ho c risorse e ho un numero di processo maggiore di c , consento solo a un numero di processi di lavorare in maniera concorrente con le risorse.
- Segnalazione: la segnalazione è usata quando un processo P_i esegue a_i solo dopo che P_j ha fatto l'operazione a_j . È implementato usando un semaforo inizializzato a 0. P_i chiama una wait sul semaforo prima dell'operazione a_i . P_j esegue una signal sul semaforo dopo aver fatto l'operazione a_j .

Uso: Mutua Esclusione

```

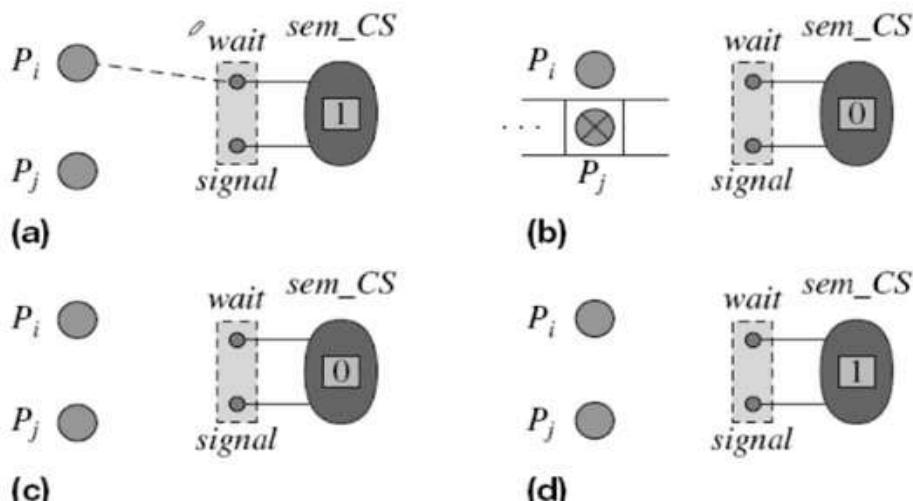
var sem_CS : semaphore := 1;
Parbegin
repeat
    wait (sem_CS);
    { Critical Section }
    signal (sem_CS);
    { Remainder of the cycle }
forever;
Parend;
end.

```

Process P_i

Process P_j

.23 CS implementation with semaphores.



USO: concorrenza limitata

Fino a c processi possono eseguire concorrentemente op_i
 Implementata inizializzando un semaforo sem_c a c .
 Ogni processo che intende eseguire op_i

- fa una ***wait (sem_c)*** prima di eseguire ***op_i***, e
- una ***signal (sem_c)*** dopo averla eseguita

Uso: Segnalazione tra Processi

```
var sync : semaphore := 0;  
Parbegin  
    ...  
    wait (sync);  
    { Performaction  $a_i$  }  
Parend;  
end.  
Process  $P_i$   
    ...  
    { Performaction  $a_j$  }  
    signal (sync);  
Process  $P_j$ 
```

- Non possono verificarsi race condition poiché le operazioni *wait* e *signal* sono indivisibili
 - *Semaforo binario*: può solo avere i valori 0 e 1
-

SEMAFORI BINARI

Definizione: variante dei semafori in cui il valore può assumere solo i valori 0 e 1.

Uso: servono a garantire mutua esclusione, semplificando il valore del programmatore. Hanno lo stesso potere espressivo dei semafori contatore.

Osservazione: la differenza è solo concettuale. Il resto è uguale.

PRODUTTORE CONSUMATORE CON SEMAFORI

Produttore-consumatore con buffer singolo:

evita la busy wait poiché i semafori sono usati per controllare se i buffer sono pieni o vuoti.

La concorrenza totale nel sistema è 1.

La soluzione usa 2 semafori: uno full (locazioni piene) e uno empty. Prima di iniziare abbiamo full a zero. Mentre empty ne ha 1. Il buffer ha 1 elemento.

Il produttore per poter produrre deve assicurarsi che il buffer sia vuoto e fa una wait sul semaforo empty, se trova locazione vuota acquisisce il semaforo, empty viene portato a zero e va ad eseguire in mutua esclusione la sua operazione. Dopo aver finito deve segnalare con una signal su full e incrementare il valore del semaforo o risvegliare un processo bloccato.

Il consumatore per consumare deve accettarsi che ci siano locazioni piene con una wait su full, se sono piene, accede, decremente full e consuma. Alla fine fa una signal su empty.

Produttore-consumatore con buffer singolo

```

type   item = . . .;
var
    full : Semaphore := 0; { Initializations }
    empty : Semaphore := 1;
    buffer : array [0] of item;
begin
Parbegin
repeat           Ⓛ
    wait (empty);
    buffer [0] := . . .;
    { i.e., produce }      ]
    signal (full);
    { Remainder of the cycle }
forever;
Parend;
end.
repeat
    wait (full);
    x := buffer [0];
    { i.e., consume }
    signal (empty);
    { Remainder of the cycle }
forever;

```

Producer Consumer

SUPPONIAMO DI AVERE ORA N BUFFER

In questo caso cambia che l'array è di n elementi e lo implementiamo come un buffer circolare. Si utilizzano sempre full ed empty ma ora empty è inizializzato a n. ho bisogno di due

indici, uno del produttore per scorrere il buffer e uno per il consumatore, inizializzati a zero. Il codice è uguale, solo che dopo la produzione, dobbiamo aggiornare l'indice. Vale anche per il consumatore.

Il produttore vede se ci sono locazioni vuote con una wait su empty. Va a produrre e incrementa il contatore ed esegue una signal su full.

Il consumatore controlla se ci sono locazioni da consumare con una wait, consuma, incrementa il contatore e segnala al produttore che c'è una nuova locazione vuota.

Se avessi più produttori dovrei usare un nuovo semaforo S_C, dopo che posso produrre, acquisisco il semaforo S_C

Lettori scrittori con semafori

Ho m lettori e n processi scrittori ed entrambi i gruppi di processi cercano di accedere ad un dato da leggere o scrivere.

Se ho più lettori che vogliono leggere, possono farlo concorrentemente.

Se ho lettori che scrivono, gli scrittori non possono scrivere e viceversa.'

Gli scrittori possono leggere uno alla volta

Per poter gestire questa situazione, utilizziamo una serie di contatori. In particolare abbiamo:

- Runread: numero di lettori in lettura
- Totread: numero di lettori che intendono leggere o sono in lettura
- Lo stesso per runwrite e totwrite

Parbegin repeat <div style="border: 1px dashed gray; padding: 5px;"> if runwrite ≠ 0 then { wait }; { read } <div style="border: 1px dashed gray; padding: 5px;"> if runread = 0 and totwrite ≠ 0 </div> then <i>activate one waiting writer</i> forever; Parend; </div> <p style="text-align: center;"><u>Reader(s)</u></p>	repeat <div style="border: 1px dashed gray; padding: 5px;"> if runread ≠ 0 or runwrite ≠ 0 </div> then { wait }; { write } <div style="border: 1px dashed gray; padding: 5px;"> if totread ≠ 0 or totwrite ≠ 0 </div> then <i>activate either one waiting writer or all waiting readers</i> forever;
---	--

Writer(s)

Le soluzioni a questo problema sono 2:

1. Una da priorità agli scrittori
2. Una da priorità ai lettori (quella che vedremo): se sono uno scrittore che ha modificato dobbiamo decidere a chi diamo priorità (in questo caso lettori). Lo scrittore verifica se ci sono lettori in procinto di leggere, deve risvegliare tutti i lettori in modo che possano andare a leggere (sfrutta la variabile totread).

Processo lettore:

capire se posso leggere, controllo se runwrite != 0, ovviamente non posso leggere e devo aspettare, l'accesso a runwrite avviene in mutua esclusione, se in runwrite non c'è nulla allora il lettore può leggere. Dopo che ho letto: se ho letto verifico se sono l'ultimo che ha letto (runread=0) e se c'è uno scrittore che vuole scrivere allora attivo uno scrittore in attesa. Se non è così, o perché ci sono altri lettori o perché non ci sono scrittori, allora salto e continuo con le attività

Processo scrittore:

posso scrivere solo se non ci sono lettori che stanno leggendo e se non ci sono scrittori in azione, se non è così allora aspetto oppure scrivo. Se ci sono lettori o scrittori che vogliono agire, o attivo tutti i lettori o attivo uno scrittore.

```

var
    totread, runread, totwrite, runwrite : integer;
    reading, writing : semaphore := 0;
    sem_CS : semaphore := 1;
begin
    totread := 0;
    runread := 0;
    totwrite := 0;
    runwrite := 0;
Parbegin
    repeat
        wait (sem_CS);
        totread := totread + 1;
        if runwrite = 0 then
            runread := runread + 1;
            signal (reading);
            signal (sem_CS);
            wait (reading);
            { Read }
            wait (sem_CS);
            runread := runread - 1;
            totread := totread - 1;
            if runread = 0 and
                totwrite > runwrite
            then
                runwrite := 1;
                signal (writing);
                signal (sem_CS);
    forever;
Parend;
end.

Reader(s)                                Writer(s)

```

Questo è il problema con preferenza ai lettori

Dichiariamo totread e le altre 3 var.

Uso 3 semafori:

- Un semaforo reading per sapere quando posso leggere
- Uno writing per sapere quando posso scrivere
- Uno per la mutua esclusione inizializzato a 1

Inizializzo le 4 variabili tot... a zero.

Codice lettori:

un processo lettore deve dichiarare che vuole leggere, per fare questo deve manipolare la variabile totread per dire che è un potenziale lettore nuovo. Prima di poter operare su questa var devo acquisire il semaforo per la mutua esclusione facendo una wait sul semaforo CS(mutua esclusione): se sono il primo accederò in mutua esclusione a manipolare la var. Incremento la variabile. Dobbiamo verificare se ci sono scrittori che stanno scrivendo prima di leggere. Controllo in mutua esclusione: se non c'è nessuno in scrittura posso leggere divento un lettore effettivo e aggiorno runread. Ora il lettore si auto segnala con una signal su reading, esco dalla sezione critica e a questo punto eseguo wait su reading, trovo che reading è 1 quindi effettivamente vado a leggere. Nel momento in cui sono un lettore che ha finito di leggere, usando lo schema visto in precedenza, deve acquisire il semaforo per la mutua esclusione. Decrementa runread perché ha finito di leggere e in più decrementa totread. Verifica poi se era l'ultimo lettore, se runread=0 e se totwrite>runwrite cioè ci sono scrittori in attesa, allora pongo runwrite a 1 (solo uno scrittore alla volta può scrivere) e do la possibilità allo scrittore di andare a scrivere invocando una signal su writing, fatto ciò, lascio la SC invocando la signal sul semaforo della SC.

Codice Scrittori:

tutte le operazioni devono essere fatte in mutua esclusione, quindi una wait sul semaforo della mutua esclusione.

Sono un potenziale nuovo scrittore quindi aggiorno totwrite.

Per vedere se diventerò un effettivo scrittore, devo verificare innanzitutto se ci sono lettori. Se non ci sono lettori e se non ci sono altri scrittori allora io posso andare a scrivere. Metto runwrite a 1 e abilito me stesso a poter scrivere, con una signal su writing(inizialmente è zero). Dopo, fatto ciò, posso invocare signal sul semaforo della mutua esclusione e posso scrivere con una signal su Writing e vado a scrivere. Una volta che ho scritto succede che devo aggiornare tutti del fatto che ho scritto.

Acquisisco il semaforo della mutua esclusione e decremento poiché c'è uno scrittore in meno e vado a vedere se ci sono lettori che devono leggere e li vado a risvegliare (precedenza ai lettori) Finché il nr degli scrittori è minore del nr dei lettori, incremento runread e faccio una signal sul semaforo reading.

IMPLEMENTAZIONE DEI SEMAFORI

Per l'implementazione dei semafori si considera un semaforo come una struttura dove abbiamo un campo contatore, un campo lista per creare la lista dei processi bloccati sul semaforo e uso una var booleana chiamata lock che uso per utilizzare l'atomicità delle operazioni wait e signal sul semaforo. Per evitare race condition durante l'accesso al valore del semaforo bisogna invocare una funzione che acquisisce il lock. Il lock è un mutex che stabilisce se è acquisito oppure no i m

I MONITOR

È un costrutto di sincronizzazione. Consente di alleviare la quantità di lavoro che il programmatore ha quando si deve accedere alla Sezione Critica. Il monitor garantisce la mutua esclusione. Un programmatore poco esperto potrebbe utilizzare in maniera impropria un semaforo. Questo costrutto è un linguaggio ad alto livello (non è messo a disposizione del sistema operativo, tutta la gestione del monitor è demandata al compilatore del linguaggio che mette a disposizione il monitor). Questo costrutto è assimilabile a una classe dove nella quale si definiscono i dati locali alla classe che sono soggetti a race condition e devono essere regolati in mutua esclusione. Oltre ai dati nel monitor vi sono molte funzioni attraverso le quali è possibile accedere ai dati. La manipolazione dei dati del monitor avviene solo invocando le procedure del monitor stesso e non c'è nessun altro modo da parte dei processi di poter accedere ai dati se non così. Un'altra caratteristica fondamentale che rende il monitor più semplice rispetto ai semafori è il fatto che ogni processo può invocare una procedura del monitor soltanto in mutua esclusione. Questa cosa non va garantita dal programmatore ma il costrutto lo fa automaticamente. Ecco perché risulta più comodo l'utilizzo dei monitor rispetto ai semafori.

È una raccolta di dati, strutture e procedure raggruppate in un modulo o pacchetto.

- I processi possono invocare le procedure di un Monitor in ogni momento
- I processi non possono accedere ai dati di un monitor se non attraverso le procedure del monitor
- In un monitor può essere attivo un solo processo alla volta
 - Un monitor implementa la mutua esclusione

```
monitor monitor-name
{
    // shared variable
    declarations
    procedure P1 (...) { .... }

    procedure Pn (...) {.....}

    Initialization code ...
}
```

// Simile ad una classe C++ o Java

Pseudocodice definizione di un monitor. Somiglia molto a una classe di C++.

SINCRONIZZAZIONE CON MONITOR

Le operazioni che si fanno sui dati devono essere eseguite in mutua esclusione. Non è sufficiente solo questo, oltre alla mutua esclusione abbiamo la necessità della sincronizzazione di controllo che gestisce l'ordine attraverso il quale i processi possono eseguire le proprie operazioni. Supponiamo che un processo invochi un monitor, e all'interno di esso, deve bloccarsi fino a che si verifichi una certa condizione.

È necessario un supporto affinchè il processo quando si blocca rilascia il monitor in modo da consentire ad altri processi l'ingresso.

Quando la condizione sarà soddisfatta ed il monitor di nuovo disponibile, al processo deve essere consentito di sbloccarsi e di rientrare nel monitor nel punto in cui era stato sospeso.

MONITOR: DEFINIZIONE FORMALE

Un monitor ci consente di encapsulare i dati su cui si può avere race condition. Consentono l'esecuzione in mutua esclusione perché definiscono un lock (implicito) e fa sì che quando un processo entri in un monitor lo faccia acquisendo questo lock in maniera esclusiva. Qualsiasi altro processo volesse entrare nel monitor, se vi è il lock, non può farlo. Quando il processo che ha il lock occupato, esce dal monitor, il costrutto farà sì che il processo rilasci il monitor.

All'interno del monitor abbiamo più variabili di condizione che consentono ai processi di bloccarsi finché non si verifica una certa condizione.

SINCRONIZZAZIONE CON MONITOR

Il monitor ci offre la mutua esclusione e la garantisce.

La sincronizzazione invece è deputata al programmatore e la si fa tramite le variabili di condizione.

Queste variabili di condizione mettono a disposizione due operazioni:

- ***cond_wait(c)*** consente ad un processo di bloccarsi sulla variabile di condizione se questa non è verificata.
 - ***cond_signal(c)*** riprende l'esecuzione di un processo bloccato dalla *cond_wait* sulla stessa variabile di condizione.
- **Osservazione**
 - le operazioni *cond_wait()* e *cond_signal()* sono diverse rispetto alle corrispondenti nei semafori
 - Se un processo invoca *cond_signal()* e non c'è nessuno in attesa sulla variabile di condizione il segnale è perso

VARIABILI DI CONDIZIONI

- **condition x;**
 - **x.cond_wait()** un processo che la invoca si blocca fino ad una **x.cond_signal()**
 - Il lock mantenuto dal processo è rilasciato atomicamente quando il processo si blocca
 - **x.cond_signal()** risveglia un processo (se esiste) che ha invocato **x.cond_wait()**
 - Se non ci sono state **x.cond_wait()** sulla variabile, non ha effetto
- Consideriamo P_1 e P_2 che eseguono due istruzioni S_1 e S_2 con il vincolo che S_1 sia eseguita prima di S_2
 - Creiamo un monitor con due procedure F_1 and F_2 invocate da P_1 e P_2 , rispettivamente
 - Una variabile di condizione "x"
 - Una variabile booleana "done" inizializzata a false
 - **F1:**

```
s1;
done = true;
x.signal(); → COND
```
 - **F2:**

```
if done = false
    x.wait()
s2;
```

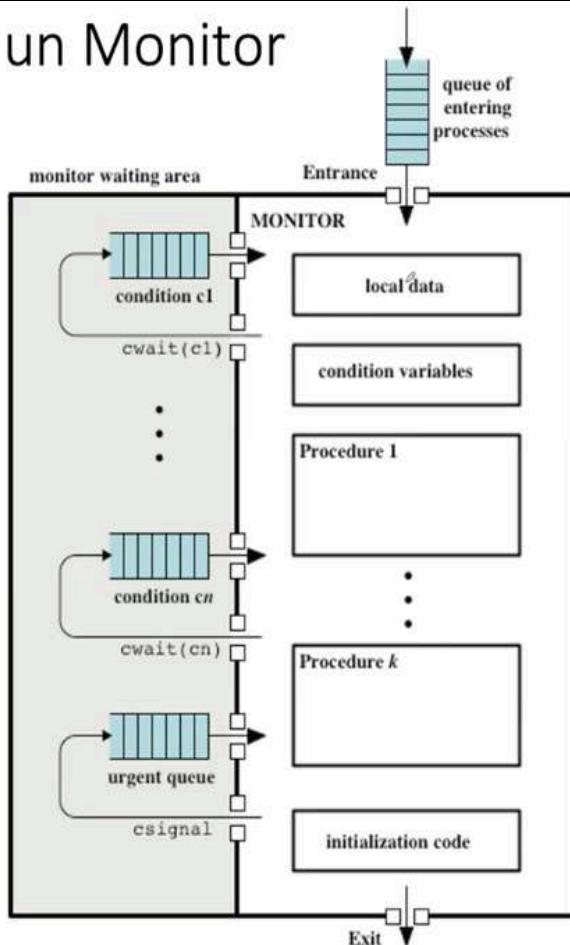
REGOLE PER LE VARIABILI DI CONDIZIONE

Dopo una *cond_signal* è necessario avere un processo attivo nel monitor e abbiamo bisogno di una regola per decidere chi risvegliare.

- Hoare: eseguiamo il processo appena risvegliato, sospendendo l'altro (che ha invocato *cond_signal*)
- Hansen: il processo che invoca c *cond_signal* deve uscire immediatamente
 - *cond_signal* può apparire solo come istruzione finale di una procedura del monitor
 - Concettualmente più semplici da implementare
 - Dopo la *cond_signal*, lo scheduler seleziona uno solo dei processi in attesa

- Mesa: il processo P_i che invoca *cond_signal* continua
 - o il processo in attesa comincia dopo che è + uscito dal monitor
-

Struttura di un Monitor



Il monitor ha un punto di ingresso rappresentato da una coda: se ho più processi che in quel momento cercano di invocare una procedura del monitor solo uno ha la possibilità di entrarci e gli altri verranno accodati nella coda di attesa per entrare nel monitor. Ma questa non è l'unica coda di attesa: ci sono le code associate alle variabili di condizione.

Nel momento in cui un processo è entrato nel monitor e ha invocato una data procedura, esempio un produttore che va a produrre e che il buffer in quel momento è pieno, allora si deve bloccare invocando la cond wait. Il costrutto monitor fa sì che il processo bloccato rilasci il lock in maniera tale che un altro processo in attesa possa essere schedulato. A un certo punto un processo segnalerà al processo bloccato che una data condizione

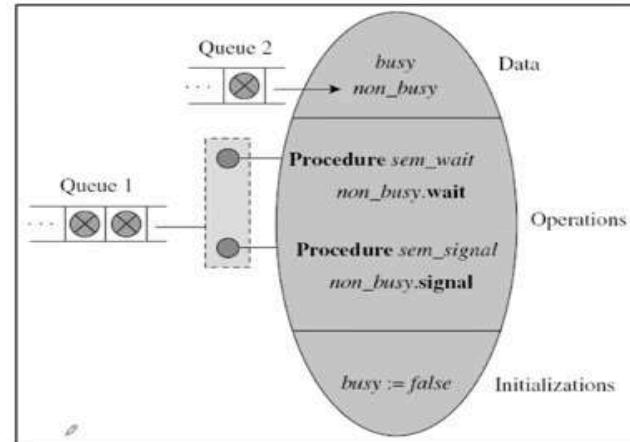
si è verificata con una cond signal, questo significa che il processo si può risvegliare. Chi continua tra quello risvegliato o segnalato dipende dall'approccio usato (Hoare, Hansen, Mesa). Può accedere che ho più processi che potrebbero dover necessitare del verificarsi della stessa condizione e avere più processi bloccati in attesa.

Semaforo Binario con Monitor

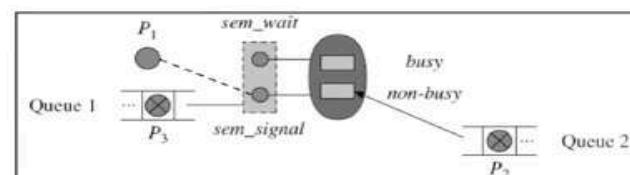
```

type Sem_Mon_type = monitor
var
    busy:boolean;
    non_busy: condition;
procedure sem_wait;
begin
    if busy = true then non_busy.cond_wait;
    busy := true;
end
procedure sem_signal;
begin
    busy := false;
    non_busy.cond_signal;
end
begin { initialization }
    busy := false;
end
end
var binary_sem : Sem_Mon_type;
begin
    Parbegin
        repeat           repeat           repeat
            binary_sem.sem_wait;   binary_sem.sem_wait;   binary_sem.sem_wait;
            { Critical Section } { Critical Section } { Critical Section }
            binary_sem.sem_signal; binary_sem.sem_signal; binary_sem.sem_signal;
            { Remainder of       { Remainder of       { Remainder of
                the cycle } }     the cycle } }     the cycle } }
        forever;         forever;         forever;
    Paren;
    end.
    Process P1           Process P2           Process P3

```



Monitor per il semaforo binario



P₁ accede in SC, P₂ cerca di eseguire sem_wait, P₃ prova ad eseguire sem_wait prima che P₁ finisca di eseguire sem_signal

Esempio: Produttori-Consumatori con Monitor

```

type Bounded_buffer_type = monitor
const
  n = . . .;                                { Number of buffers }
type
  item = . . .;
var
  buffer : array [0..n-1] of item;
  full, prod_ptr, cons_ptr : integer;
  buff_full : condition; ←
  buff_empty : condition; →
procedure produce (produced_info : item);
begin
  if full = n then buff_empty.wait;
  buffer [prod_ptr] := produced_info;          { i.e., Produce }
  prod_ptr := prod_ptr + 1 mod n;
  full := full + 1;
  buff_full.signal;
end;
procedure consume (for_consumption : item);
begin
  if full = 0 then buff_full.wait;
  for_consumption := buffer[cons_ptr];          { i.e., Consume }
  cons_ptr := cons_ptr + 1 mod n;
  full := full - 1;
  buff_empty.signal;
end;
begin { initialization }
  full := 0;
  prod_ptr := 0;
  cons_ptr := 0;
end;

```

Supponiamo di avere n buffer. Abbiamo che la struttura dati su cui c'è race condition è il buffer. Andiamo a definirci un monitor che rappresenta il buffer e che conterrà una costante che ci dice la dimensione del buffer e gli elementi e poi le variabili che sono l'array e gli indici che devo usare per poter scorrere il buffer da parte dei produttori e dei consumatori e usiamo due variabili di condizioni buff_full attraverso la quale attendiamo quando non ci sono buffer pieni, usata dal consumatore quando non ci sono buffer pieni e la variabile buff_empty usata dai produttori quando non ci sono buffer vuoti.

Abbiamo il codice di inizializzazione dove il buffer è inizializzato a zero e anche i suoi indici.

Le procedure usate sono produce e consume.

PRODUCE:

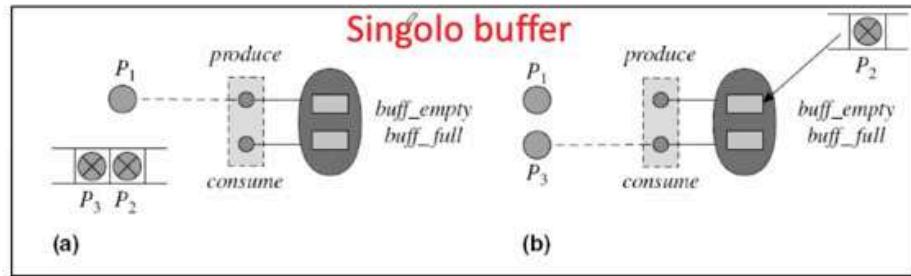
se il buffer è pieno il produttore deve aspettare la condizione per la quale ho almeno un buffer vuoto e vado a invocare la `buff_empty cond_wait`.

Se c'è un buffer vuoto e posso produrre vado a produrre accedendo al buffer, produco e incremento il contatore che mi indica dove può essere prodotto il prossimo elemento, incremento il numero di locazioni piene e vado a segnalare al consumatore in attesa che c'è una locazione piena invocando una `cond_signal` su `buff_full`.

CONSUME:

il consumatore va a controllare `full` per verificare se ci sono buffer pieni da consumare, se è 0 deve aspettare con una `cond_wait` su `buff_full` oppure consuma e invoca una `cond_signal` `buff_empty` per segnalare che c'è una nuova locazione vuota disponibile.

Esempio: Produttori-Consumatori con Monitor



```
begin
var B_buf : Bounded_buffer_type;
```

Parbegin

<pre>var info : item; repeat info := ... B_buf.produce (info); { Remainder of the cycle } forever; Parend; end.</pre>	<pre>var info : item; repeat info := ... B_buf.produce (info); { Remainder of the cycle } forever;</pre>	<pre>var area : item; repeat B_buf.consume (area); { Consume area } { Remainder of the cycle } forever;</pre>
---	--	---

Producer P₁

Producer P₂

Consumer P₃

```

procedure produce (produced_info : item);
begin
  if full = n then buff_empty.wait;
  buffer [prod_ptr] := produced_info;
  prod_ptr := prod_ptr + 1 mod n;
  full := full + 1;
  buff_full.signal;
end;
procedure consume (for_consumption : item);
begin
  if full = 0 then buff_full.wait;
  for_consumption := buffer[cons_ptr];
  cons_ptr := cons_ptr + 1 mod n;
  full := full - 1;
  buff_empty.signal;
end.
```

{ i.e., Producer }

{ i.e., Consumer }

Consideriamo di avere un solo buffer.

Supponiamo di avere due produttori(P1,P2) e un consumatore(P3) e che questi 3 processi vadano in esecuzione simultaneamente. Cercano di produrre e di consumare allo stesso tempo. Supponiamo che ha la meglio il produttore P1, se è così invoca la funzione produce e va a produrre un elemento. Siccome sta eseguendo produce, gli altri due processi che concorrentemente hanno provato ad accedere al monitor, verranno accodati nella coda di ingresso al monitor.

Supponiamo che il produttore ha terminato di produrre e che venga schedulato P2 ma è sempre un produttore e quando va a invocare produce si accorge che il buffer è pieno e si blocca sulla

variabile di condizione che specifica che per procedere ci deve essere una locazione vuota. A questo punto viene schedulato il consumatore che può invocare consume. Quando avrà finito va invocare buff_empty signal e P2 viene risvegliato, vedrà che full è 0 e potrà procedere a produrre.

Notare che abbiamo utilizzato i monitor di Hansen(l'ultima istruzione è la signal).

PROBLEMA DEL BARBIERE ADDORMENTATO (esercizio tipo esame)

- In un negozio di barbiere c'è
 - Un barbiere
 - N sedie per i clienti in attesa
 - 1 poltrona da barbiere
- Specifiche
 - Se non ci sono clienti, il barbiere si addormenta sulla poltrona
 - Quando arriva, un cliente sveglia il barbiere e si fa tagliare i capelli sulla poltrona
 - Se arriva un cliente mentre il barbiere sta tagliando i capelli ad un altro cliente, il cliente attende su una delle sedie libere
 - Se tutte le sedie sono occupate, il cliente, contrariato, se ne va!

```

sem_CS: semaphore := 1;
clienteDisponibile, poltrona : semaphore := 0;
inAttesa : integer;
SEDIE : const;

begin
  SEDIE := N;
  inAttesa := 0;
Parbegin
  repeat

    wait(ClienteDisponibile);
    wait(sem_CS);
    inAttesa := inAttesa - 1;
    signal(poltrona);
    signal(sem_CS);
    {taglia capelli};

    wait(sem_CS);
    if inAttesa < SEDIE then
      inAttesa := inAttesa + 1;
      signal(ClienteDisponibile);
      signal(sem_CS);
      wait(poltrona);
      {taglio dei capelli}

  forever
    else signal(sem_CS);
    {lascia il negozio};

Parend
Barbiere                                Clienti

```

Sincronizzazione dei Thread POSIX

- I thread POSIX forniscono
 - Mutex per la mutua esclusione
 - Un mutex è come un semaforo binario
 - Variabili di condizione per la sincronizzazione di controllo tra processi
 - Un SO può implementare i thread POSIX come thread di livello kernel o thread di livello utente
-

Sincronizzazione di Processi in Unix

- Unix System V fornisce una implementazione di livello kernel dei semafori
 - Il nome di un semaforo è chiamato key
 - Key è associato con un array di semafori
 - I processi condividono un semaforo usando la stessa key
 - Unix SVR4 tiene traccia di tutte le operazioni eseguite da un processo su ogni semaforo che usa
 - Esegue un undo su di essi quando il processo termina
 - Aiuta a rendere i programmi più affidabili
 - Unix 4.4BSD pone un semaforo in un'area di memoria condivisa ed usa un'implementazione ibrida
-

Sincronizzazione di processi Linux

- Linux ha semafori Unix-like per i processi utente
 - Fornisce anche semafori usati dal kernel
 - Semaforo convenzionale
 - Usa uno schema efficiente per un'implementazione di livello kernel
 - Semaforo Lettore-scrittore (read-write lock)
 - I kernel precedenti alla versione 2.6 implementavano la mutua esclusione nello spazio kernel mediante system call
 - Il kernel 2.6 ha un mutex dello spazio utente veloce chiamato futex
 - Usa un'implementazione ibrida
 - Fornisce un'operazione wait limitata temporalmente
-

MESSAGE PASSING

Sia monitor che semafori sono stati costruiti per applicazioni con processi concorrenti e per cooperare per la risoluzione di un problema in cui le macchine hanno 1 o più CPU con memoria condivisa nella quale si possono mettere semafori per gestire uno specifico problema di sincronizzazione. Ci sono una gamma di applicazioni dove non è possibile usare i semafori mentre per i monitor sono messi a disposizione solo da un numero limitato di linguaggi di programmazione. Quello che accade è che ci sono alcuni tipi di applicazioni (più processori con memoria non condivisa, collegati in pc diversi con una rete di comunicazione dove i semafori non possono essere usati) e c'è bisogno di usare altro, i message passing.

È la strategia di comunicazione e sincronizzazione che si può adottare per applicazioni distribuite che giacciono su host diversi e collegate da una rete di comunicazione.

Lo scambio dei messaggi viene usato per realizzare sia la sincronizzazione dei processi applicando la mutua esclusione e implementare la comunicazione in modo da scambiare info tra processi.

La funzionalità è fornita mediante una coppia di primitive

```
send (destination, message)  
receive (source, message)
```

Un processo invia le info in forma di messaggio verso un altro processo designato come destinazione.

Un processo riceve info eseguendo la primitiva *receive* che indica la sorgente e il messaggio

Sincronizzazione



I processi in qualsiasi momento possono essere impegnati a fare altro, quando c'è una send e una receive bisogna determinare quale comportamento adottare alla ricezione di queste primitive.

- **SEND:** quando un processo invia una send ci sono due possibilità. Il processo mittente viene bloccato finchè il mittente non riceve il messaggio oppure no.
- **RECEIVE:** se il messaggio è stato già inviato, viene ricevuto dal processo che ha invocato receive e il processo continua ma può avvenire che il ricevitore ha invocato receive e il messaggio non è stato ancora inviato quindi il processo che ha invocato receive viene bloccato finchè non riceve il messaggio oppure continua l'esecuzione ma rinuncia alla volontà di riceverlo.

SEND E RECEIVE BLOCCANTI

Avere primitive send e receive bloccanti significa che sia il mittente che il destinatario si bloccano finchè il messaggio non viene effettivamente consegnato.

Questo tipo di comunicazione prende nome di **rendezvous**, permette una sincronizzazione stretta tra processi.

Tuttavia possiamo usare combinazioni di queste send e receive, è possibile specificare che queste siano non bloccanti, posso adottare una combinazione di schemi (send bloccante – receive non bloccante ecc...).

Send non bloccante

Send non bloccante, receive bloccante

- Il mittente continua ma il ricevitore è bloccato fino a che il messaggio richiesto arriva
- È la combinazione più utile
- Invia uno o più messaggi a più destinazioni il più velocemente possibile
- Esempio: un processo server il cui compito è fornire un servizio o una risorsa ad altri processi

Send non bloccante, receive non bloccante

- A nessuna delle controparti è richiesto di attendere

Avere una send non bloccante (il processo che invia un messaggio non si blocca) è naturale in molte applicazioni di programmi concorrenti (comune per richiedere una print ecc).

C'è un potenziale pericolo però:

se si verifica un errore può accadere che una send non bloccante porti situazioni in cui il processo invia ripetutamente messaggi provocando uno spreco di risorse.

Inoltre si va ad attribuire al programmatore il compito di verificare se un messaggio è stato inviato o meno e se è stato ricevuto.

!!Rivedere questa parte!!

INDIRIZZAMENTO

Gli schemi per specificare processi

INDIRIZZAMENTO DIRETTO

Nel caso della send il processo va ad indicare in maniera esplicita un identificatore del destinatario.

Per quanto riguarda la receive ci sono due possibilità:

1. È richiesto che il processo vada ad indicare in maniera esplicita il mittente
2. Il ricevitore non ha la possibilità di specificare da chi ricevere il messaggio in anticipo. Pensare a un processo server di stampa che accetta messaggi richiesta da parte di qualsiasi processo, con una receive dovrebbe specificare da quale processo riceve la richiesta, ma è impossibile che il server (ricevente) vada a specificare da chi debba arrivare il messaggio, la soluzione è usare l'indirizzamento indiretto.

Indirizzamento Indiretto



È molto più flessibile e potente. Disaccoppiamo quello che è il processo mittente da quello destinatario e possiamo usare i messaggi in maniera più flessibile.

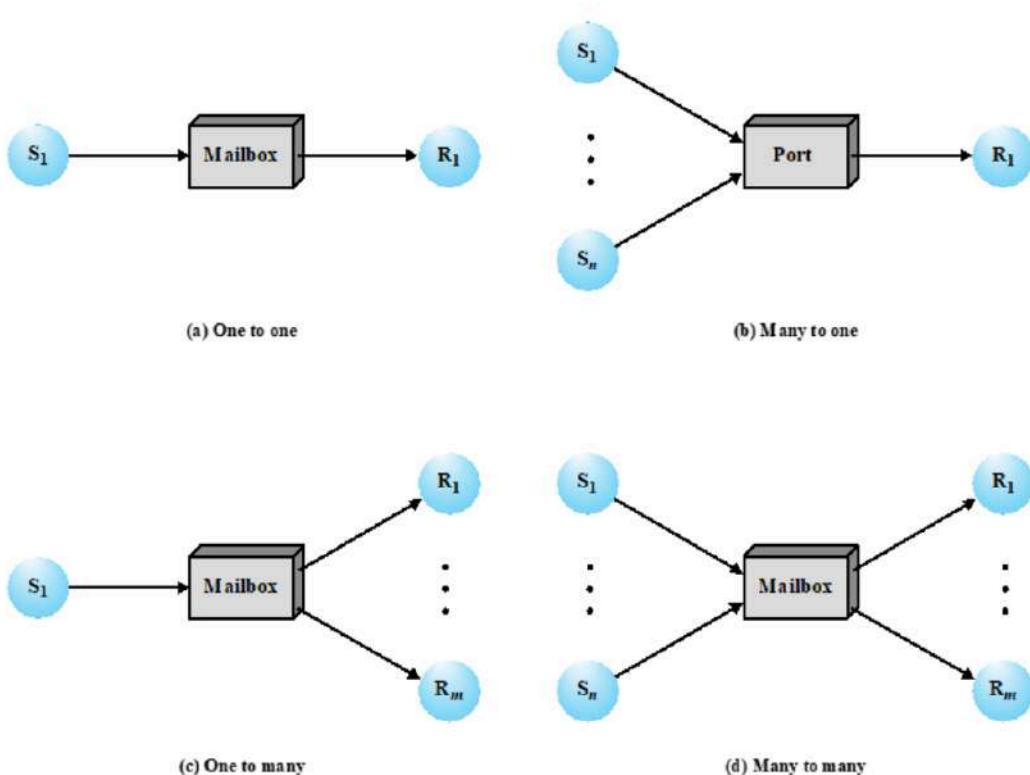
L'idea è quella che i messaggi devono essere inviati all'interno di una struttura dati condivisa e rappresentata da una sorta di coda nella quale i messaggi inviati sono temporaneamente memorizzati, in questo modo il processo ricevitore potrà ricevere il messaggio attingendo da questa coda senza sapere chi è il mittente, queste code vengono chiamate **mailbox** (recipienti di messaggi). In questo caso non ho bisogno di specificare il processo destinatario e il ricevitore attinge da queste **mailbox**.

MAILBOX

Nel momento in cui viene creata ha un nome univoco, il proprietario della **mailbox** è il processo che la crea e con l'id di

questa mailbox, tutti i processi che intendono inviare messaggi usano l'id. Solo il proprietario può ricevere i messaggi. Il kernel può fornire diversi modi per supportare le mailbox, permettendo ai processi utenti di creare nomi o dare dei nomi standard.

Indirizzamento Indiretto con Mailbox



- Relazione uno a uno: esiste un ricevitore a cui è associato un mittente. È una sorta di comunicazione privata tra due processi.
- M a 1: applicazioni di tipo client server dove abbiamo un ricevitore (server) e un certo numero di client. La mailbox viene chiamata porta.

- 1 a M: un mittente e più destinatari. Comoda quando è necessario inviare un messaggio in broadcast a più destinatari simultaneamente.
 - M a N: più processi server che devono fornire più processi client.
-

ASSOCIAZIONE E PROPRIETA' DELLE MAILBOX

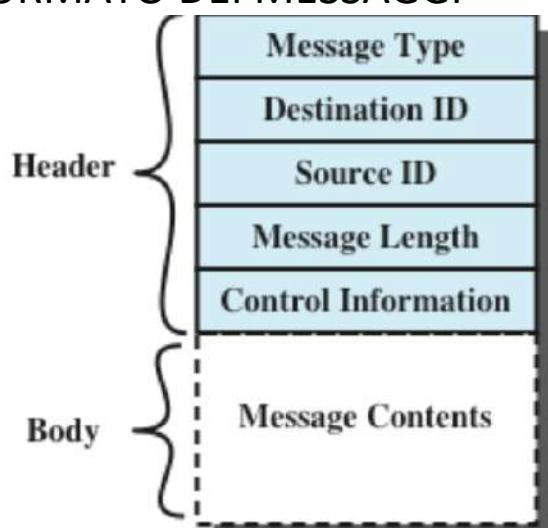
Associazione statica:

se ho M a 1 in questo caso l'associazione è creata in maniera permanente al processo che ha creato la mailbox, lo stesso per le 1 a 1.

Nel caso in cui abbiamo una M a N l'associazione di un mittente alla mailbox avviene in maniera dinamica, in questo caso bisogna usare altre primitive tipo *connect e disconnect*

Proprietà: quando il processo termina, la porta è eliminata(M a1). In generale il SO mette a disposizione un servizio di creazione mailbox e sono di proprietà del processo che le crea, quando questo termina terminano le mailbox ma il SO può far sì che la proprietà di una mailbox diventi del SO e servono delle primitive per cancellare le mailbox.

FORMATO DEI MESSAGGI



L'header contiene info riguardo il messaggio:

- Tipo
- Destinazione
- Sorgente
- Lunghezza
- Altre info

Il body contiene il messaggio effettivo.

Mutua Esclusione con Scambio Messaggi

```
/* program mutualexclusion */
const int n = /* number of processes */;
void P(int i)
{
    message msg;
    while (true) {
        receive (box, msg);
        /* critical section    */;
        send (box, msg);
        /* remainder    */;
    }
}
void main()
{
    create_mailbox (box);
    send (box, null);
    parbegin (P(1), P(2), . . . , P(n));
}
```

Si crea una mailbox condivisa da tutti i processi che devono andare in sezione critica. La mailbox è inizializzata con un messaggio null. Un processo che vuole entrare in SC deve provare a ricevere un messaggio, una volta ricevuto può andare in SC e una volta che ha svolto le sue operazioni, prende il messaggio e con una send lo invia alla mailbox. Se nella mailbox c'è un messaggio, il processo prende ed entra in SC, se non c'è il

processo si blocca. Dei vari processi bloccati su una receive solo uno di questi sarà risvegliato e riceverà il messaggio tramite la send di un processo che è uscito dalla SC.

Produttore-Consumatore con Scambio messaggi

```
const int
    capacity = /* buffering capacity */ ;
    null /* empty message */ ;
int i;
void producer()
{
    message pmsg;
    while (true) {
        receive (mayproduce, pmsg);
        pmsg = produce();
        send (mayconsume, pmsg);
    }
}
void consumer()
{
    message cmsg;
    while (true) {
        receive (mayconsume, cmsg);
        consume (cmsg);
        send (mayproduce, null);
    }
}

void main()
{
    create_mailbox (mayproduce);
    create_mailbox (mayconsume);
    for (int i = 1; i <= capacity; i++) send (mayproduce,
null);
    parbegin (producer, consumer);
}
```

Abbiamo una mailbox per produrre e un'altra per poter consumare.

Appena il produttore può generare dati, questi vengono inviati al consumatore che gli indicano che c'è disponibilità per consumare. I dati sono organizzati come una coda di messaggi, il primo inviato sarà anche il primo estratto ed elaborato.

La dimensione del buffer è determinato dalla variabile capacity.

I produttori hanno la mailbox inizializzata a null, pari alla capacità del buffer. Posso avere più produttori e consumatori e il sistema costituito da processi consumatori e produttori può essere distribuito: un host per produttori e un host per consumatori.

STRATEGIA DI ACCODAMENTO

La più semplice è di tipo FIFO ma inappropriata se qualche messaggio è più urgente di altri.

Le alternative comportano la possibilità di assegnare priorità ai messaggi oppure dare la possibilità al destinatario di accedere alla coda dei messaggi e selezionare un messaggio.

Message passing in Unix

- Tre supporti alla comunicazione interprocesso
 - Pipe:
 - Funzione per trasferimento dati
 - Pipe senza nome può essere usata solo da processi che appartengono allo stesso albero dei processi (hanno un antenato in comune)
 - Code di messaggi (*message queue*):
 - Usate dai processi nel dominio del sistema Unix
 - I permessi di accesso indicano quali processi possono inviare o ricevere messaggi
 - Socket: un'estremità di un percorso di comunicazione
 - Può essere usata per impostare dei percorsi di comunicazione tra processi nel dominio Unix e all'interno di alcuni domini Internet

Message Passing in Windows

- Pipe con nome: comunicazione affidabile e bidirezionale di byte/messaggi tra server e client
 - Implementata attraverso l'interfaccia del file system
 - Passaggio di messaggi asincrono e sincrono
- Local Procedure Call (LPC) esegue il passaggio dei messaggi tra processi nello stesso host
 - Scelta di tre modi di passaggio dei messaggi
- Socket Windows (Winsock)
 - Integrate con il Windows message passing
- RPC: sincrono e asincrono

SCHEDULING DEI PROCESSI

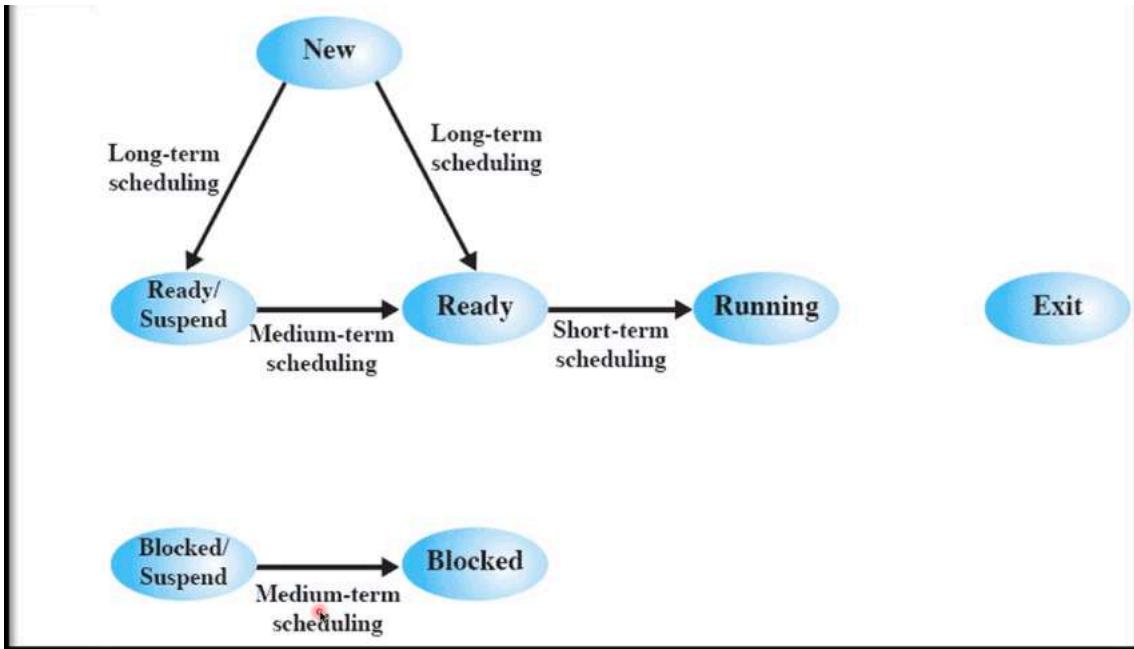
Nei sistemi multi programmati, dove ci sono più processi concorrenti in esecuzione, accade che questi processi devono alternarsi nell'uso della CPU (vale anche per i mono processore) quindi un processo si alterna con altri scambiandosi la CPU. In ogni istante la CPU è impegnata nell'esecuzione di un processo mentre gli altri sono in attesa. Lo scheduling serve a decidere quale processo deve essere eseguito. Le strategie di scheduling sono importanti per le prestazioni del sistema.

La chiave per la multi programmazione è lo scheduling, per questo motivo all'interno dei sistemi si adottano diverse strategie di scheduling:

- Lungo termine: viene invocato per decidere quale processo deve essere eseguito
- Medio termine: decide quali processi devono essere presenti parzialmente o totalmente in memoria
- Breve termine: quale dei processi pronti è assegnato alla CPU
- I/O: quale richiesta di I/O pendente di un processo deve essere gestita da un dispositivo di I/O disponibile.

Nel momento in cui abbiamo un nuovo processo (creato o schedulato) lo scheduler a lungo termine decide quali dei processi deve essere ammesso, portato in memoria e schedulato dallo scheduler a breve termine.

In alcuni sistemi quando viene creato un nuovo processo, parte direttamente nello stato swap out e lo scheduler a medio termine dovrà deciderlo di portarlo in swap in.



TERMINOLOGIA E CONCETTI DI SCHEDULING

Quando parliamo di scheduling in generale, lo possiamo definire come un'attività in base alla quale vado a selezionare una richiesta (esecuzione di un programma o job) che deve essere soddisfatta dalla CPU.

TERMINI IMPORTANTI

- **TEMPO DI ARRIVO:** indica il momento nel quale un utente sottomette un job o un processo
- Tempo di ammissione: indica il tempo nel quale il sistema inizia a considerare un job o un processo per schedularlo.
- Tempo di completamento: tempo nel quale viene completato un processo
- Deadline: istante temporale limite entro il quale un processo deve essere completato

- **Tempo di servizio:** tempo totale di CPU e I/O richieste per essere completato un processo
- **Prelazione:** deallocazione di un processo dalla CPU forzata
- **Priorità:** meccanismo dove è possibile selezionare un processo con una strategia quando abbiamo più processi che devono essere eseguite.

Questi concetti possono essere basati sulla visione del sistema e sull'utente: vediamo

VISIONE DELL'UTENTE

2. **Deadline overrun:** quantità di tempo ecceduta di un processo rispetto alla quantità fissata inizialmente
3. **Fair share:** una condivisione di tempo di CPU specificata che deve essere devoluta all'esecuzione di un processo o di un gruppo di processi
4. **Response ratio:** rapporto tra (il tempo trascorso del processo in quel momento rispetto al tempo di arrivo+ tempo di servizio di quel processo)/tempo di servizio del processo
5. **Tempo di turnaround:** tempo che intercorre tra la sottomissione di un processo e il suo completamento.
Questo è un concetto che ha a che fare con i sistemi non interattivi
6. **Tempo di turnaround pesato:** rapporto tra tempo di turnaround e il tempo di servizio di un processo

Abbiamo altri concetti

- Tempo medio di servizio : media di tutte le richieste da parte di un processo utente
- Tempo medio di turnaround

Concetti legati alla prestazione del sistema

- Lunghezza della schedulazione
 - Throughput: numero medio di processi completati dal sistema nell'unità di tempo
-

TECNICHE DI SCHEDULING

Quello di cui si deve preoccupare il sistema è ottenere buone prestazioni e buoni servizi utente.

Ciò richiede l'adozione di diverse strategie.

1. Scheduling basati su priorità: si migliora il throughput del sistema dando una priorità diversa ai processi I/O bound e CPU bound e se la diamo ai processi I/O bound allora avremo migliori prestazioni.
2. Riordino delle richieste: immaginare un certo numero di processi che vanno nella coda dei processi per essere selezionati, arriveranno in un certo ordine. Potremmo sia schedularli per ordine di arrivo ma potremmo anche selezionarli a nostro piacimento. Questo è implicito nella prelazione. Ciò migliora il servizio utente e il throughput.
3. Variazione dello slot temporale:
 - Ricordiamo $\eta = \delta / (\delta + \sigma)$, con η efficienza di CPU, σ overhead dello scheduling, δ slot temporale

Valori più piccoli degli slot temporali forniscono migliori tempi di risposta ma una minore efficienza della CPU. Quello che si può fare è cercare di usare slot temporali diversi per richieste differenti (slot piccoli per I/O bound e grandi per CPU bound).

RUOLO DELLE PRIORITA'

La priorità che impostiamo va fatta in due modi:

1. Statica: imposto priorità a priori sulla base di alcune considerazioni
2. Dinamica: la priorità può variare sulla base di criteri impostati di conseguenza

Sulla base delle priorità posso fare il riordino di cui abbiamo parlato prima, servendo prima processi brevi o quelli che non vengono serviti da tempo.

- Cosa accade se i processi hanno la stessa priorità?

In questo caso usiamo lo scheduling round-robin, assegnando una quantità temporale di CPU alla fine della quale selezione un successivo processo. Ci sono però pericoli potenziali: la **STARVATION**, è possibile che processi con priorità bassa sono ritardati nell'essere selezionati per l'esecuzione e alcuni processi possono non essere mai eseguiti ma con strategie di aging è possibile risolvere la starvation.

Aging: se ho un processo che non viene eseguito da tempo, col passare del tempo la priorità aumenta sempre più in modo tale da poter essere eseguito.

SCHEDULING SENZA PRELAZIONE

Quando viene selezionato un processo, questo viene completato totalmente dalla CPU.

L'unico vantaggio è quello di essere estremamente semplice da realizzare, una volta selezionato un processo, lo assegno alla CPU

fino al suo completamento. Anche se usiamo uno scheduling senza prelazione, posso riordinare le richieste comunque. Tra gli algoritmi senza prelazione abbiamo:

- Alcune politiche di scheduling senza prelazione:
 - Scheduling *First-Come, First-Served* (FCFS)
 - Scheduling *Shortest Job First* (SJF)
 - Scheduling *High Response Ratio* (HRN)

Process	P_1	P_2	P_3	P_4	P_5
Admission time	0	2	3	4	8
Service time	3	3	5	2	3

Processi per lo Scheduling

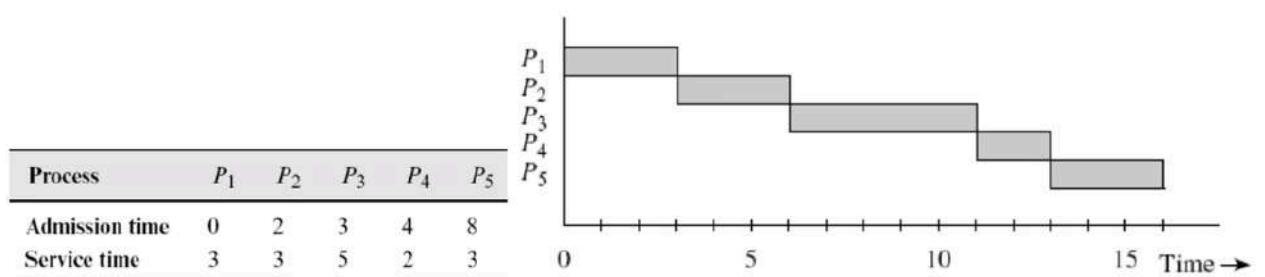
Scheduling First Come, First Served (FCFS)

- Le richieste sono schedulate sempre nell'ordine in cui giungono al sistema

Time	Completed process			Processes in system (in FCFS order)	Scheduled process
	<i>id</i>	<i>ta</i>	<i>w</i>		
0	-	-	-	P_1	P_1
3	P_1	3	1.00	P_2, P_3	P_2
6	P_2	4	1.33	P_3, P_4	P_3
11	P_3	8	1.60	P_4, P_5	P_4
13	P_4	9	4.50	P_5	P_5
16	P_5	8	2.67	-	-

$$\overline{ta} = 6.40 \text{ seconds}$$

$$\overline{w} = 2.22$$



Per gli esempi consideriamo la tabella in basso a sinistra della foto.

In questo algoritmo le richieste sono elaborate sulla base dell'ordine di arrivo. Costruiamo la tabella mettendo:

- Time: istanti temporali e vediamo per ogni istanti quale processo viene schedulato
- ID: consideriamo quale processo viene completato
- TA: tempo di turnaround (tempo dall'arrivo del processo al suo completamento)
- W: tempo di turnaround pesato (TA / tempo di servizio)
- Processes in system: ordine all'interno coda
- Scheduled process: processi schedulati in un tale istante

Vediamo l'esempio:

Al tempo zero abbiamo solo il processo P1, nessun processo è completato, non abbiamo né TA e né W, la coda di processi pronti è costituita solo dal processo P1 e quindi l'algoritmo schedula P1, essendo schedulato P1, avendo una schedulazione non prelazionata, P1 viene eseguito fino al suo completamento, alla seconda riga della tabella vediamo che P1 richiede 3 unità di tempo per essere completato, al tempo 3 abbiamo che P1 è stato completato, il TA è 3, il W è dato da $3/3$ quindi 1. All'istante 3 la coda dei processi è P2, P3 e quindi l'algoritmo schedula P2 e richiede 3 unità di servizio. Alla terza riga vediamo che P2 si completa nell'istante 6, nell'id c'è P2, il processo P2 è arrivato nell'istante 2, è stato completato a 6 e il TA quindi è 4, il W è 1.33. All'istante la coda dei processi contiene P3 e P4 e viene

schedulato quindi P3 che, come vediamo dalla riga 4, impiega 8 unità di tempo per essere completato. All’istante 11 infatti il processo P3 è completato, il TA è 8, il W è 1.60 e nella coda dei processi abbiamo P4, P5, viene quindi schedulato P4 che impiega 9 unità di tempo. Al tempo 13 P4 è completato, il TA è 9, il W è 4.50 e nella coda dei processi abbiamo solamente P5 che viene schedulato e impiega 8 unità di tempo per essere completato, all’istante 16 è completato, il TA è 8, il W 2.67 e la coda dei processi in attesa è vuota e non viene schedulato nulla.

Possiamo calcolare il TA medio che è 6.40 secondi e il W medio (2.22).

Possiamo osservare che c’è un’ampia variabilità di W ed è dovuta dal fatto che ad esempio P4, che è un processo breve, per essere schedulato ha dovuto aspettare un po’ e questo va a impattare sul W. Se ho processi brevi per questo algoritmo, il tempo di turnaround è elevato quindi potremmo cercare di migliorare questa strategia cercando di fare in modo che i processi brevi non attendano troppo.

Quello che si può fare è migliorare l’algoritmo facendo in modo da riordinare le richieste cercando di dare una possibilità di esecuzione ai processi più brevi.

Vediamo quindi la Shortest Job First

Scheduling Shortest Job First (SJF)

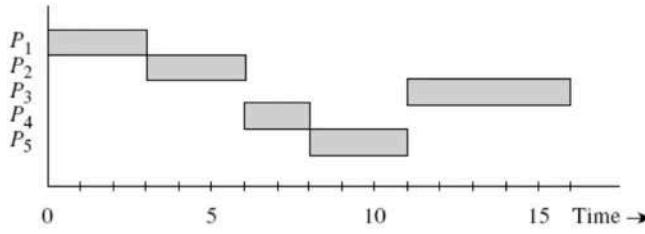
- Schedula sempre la richiesta con il minimo tempo di servizio

Time	Completed process			Processes in system	Scheduled process
	<i>id</i>	<i>ta</i>	<i>w</i>		
0	—	—	—	{P ₁ }	P ₁
3	P ₁	3	1.00	{P ₂ , P ₃ }	P ₂
6	P ₂	4	1.33	{P ₃ , P ₄ }	P ₄
8	P ₄	4	2.00	{P ₃ , P ₅ }	P ₅
11	P ₅	3	1.00	{P ₃ }	P ₃
16	P ₃	13	2.60	{}	—

$$\overline{ta} = 5.40 \text{ seconds}$$

$$\overline{w} = 1.59$$

Può causare starvation
dei processi più lunghi



Process	P ₁	P ₂	P ₃	P ₄	P ₅
Admission time	0	2	3	4	8
Service time	3	3	5	2	3

Diamo in qualche modo una priorità rispetto a quanto tempo è necessario per poter completare un processo.

Schedula il processo con tempo di servizio minore.

Ricostruiamo la tabella e analizziamola:

Non ci interessa l'ordine di arrivo nel sistema ma solamente chi c'è.

I processi sono gli stessi dello scenario precedente.

Al tempo 0 ho solo P₁ e l'algoritmo seleziona e schedula P₁, ricordando che è uno schema senza prelazione.

All'istante 3 P₁ viene completato, il TA è 3 e il W è 1. Nella coda dei processi in attesa abbiamo l'insieme {P₂, P₃}, viene schedulato P₂ poiché ha il tempo di servizio più breve, al tempo 6 P₂ è completato, il TA è 4, il W è 1.33. Al tempo 6 abbiamo l'insieme {P₃, P₄} e viene schedulato P₄ perché ha tempo di

servizio minore di P3. P4 impiega 2 unità di tempo per essere completato, al tempo 8 infatti finisce, il TA è 4, il W è 2 e nell'insieme abbiamo {P3,P5} e viene schedulato P5. Al tempo 11 viene completato, il TA è 3, il W è 1 e nella coda rimane solo P3 che viene schedulato, impiega 5 unità di tempo per essere completato, il TA è 13 e W è 2.60, l'insieme è vuoto e non viene schedulato nulla.

- Il TA medio è 5.40 secondi e il W medio è 1.59.

È più basso dell'algoritmo precedente proprio perché abbiamo dato la possibilità ai processi brevi di essere completati prima.

- Il problema dello SJF che sorge è la STARVATION dei processi più lunghi. Se continuano ad arrivare processi brevi i processi lunghi non avranno mai la possibilità di essere schedulati.
- Inoltre dobbiamo conoscere i tempi di servizio dei processi che non conosciamo effettivamente. È possibile con una simulazione o con una definizione di tempi da parte dell'utente ma questo è molto complesso.

SCHEDULING HIGH RESOIBSE RATIO NEXT (HRN)

Scheduling High Response Ratio Next (HRN)

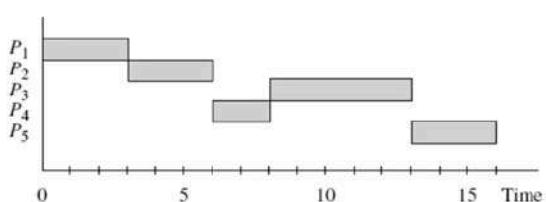
- Calcola i rapporti di risposta di tutti i processi nel sistema e seleziona il processo con il rapporto più elevato

$$\text{Response ratio} = \frac{\text{time since arrival} + \text{service time of the process}}{\text{service time of the process}}$$

Time	Completed process			Response ratios of processes					Scheduled process
	<i>id</i>	<i>ta</i>	<i>w</i>	<i>P₁</i>	<i>P₂</i>	<i>P₃</i>	<i>P₄</i>	<i>P₅</i>	
0	—	—	—	1.00					<i>P₁</i>
3	<i>P₁</i>	3	1.00		1.33	1.00			<i>P₂</i>
6	<i>P₂</i>	4	1.33			1.60	2.00		<i>P₄</i>
8	<i>P₄</i>	4	2.00			2.00		1.00	<i>P₃</i>
13	<i>P₃</i>	10	2.00					2.67	<i>P₅</i>
16	<i>P₅</i>	8	2.67						—

$$\overline{ta} = 5.8 \text{ seconds}$$

$$\overline{w} = 1.80$$



L'uso del tasso di risposta contrasta la starvation

Process	<i>P₁</i>	<i>P₂</i>	<i>P₃</i>	<i>P₄</i>	<i>P₅</i>
Admission time	0	2	3	4	8
Service time	3	3	5	2	3

Tutti i SO moderni usano la prelazione.

L'ultimo algoritmo senza prelazione che andremo a considerare è HRN, va a calcolare il rapporto di risposta di tutti i processi all'interno del sistema e seleziona quello con rapporto più elevato.

Il rapporto di risposta si calcola: rapporto tra la somma del tempo in cui arriva il processo e il tempo di servizio, diviso il tempo di servizio.

Questo rapporto di risposta indica il ritardo dell'espletamento di un processo rispetto al suo tempo di servizio.

Seleziono quello con il rapporto più elevato, significa che riesco a tener conto di quei processi che stanno da più tempo, oltre a considerare i tempi di servizio del processo.

Creiamo la solita tabella:

All'istante 0 nessuno è completato, piuttosto di considerare la coda dei processi, a ogni istante vado a calcolare il response ratio per ognuno dei processi, il response ratio per P1 è 1 e, essendo senza prelazione, P1 viene schedulato e dopo 3 unità di tempo viene completato. Il TA è 3, il W è 1, all'istante 3 abbiamo 2 processi, P2 e P3, calcoliamo il response ratio. P2=1.33 e P3=1.00, selezioniamo il maggiore con il response ratio e quindi P2 viene schedulato. E cos' via fino all'istante 16 dove abbiamo solo P5 che viene schedulato e non ci sono altri processi in coda e non viene schedulato più nulla.

Il TA medio è 5.8 secondi e il W medio è 1.80.

Quello che osserviamo è che il rapporto di risposta di un processo breve si incrementa più velocemente rispetto a uno più lungo e quindi comunque si privilegia i processi brevi (importante per migliorare il throughput) ma è anche vero che alla lunga il

response ratio dei processi lunghi cresce con il passare del tempo e introduce un effetto simile all'aging.

POLITICHE DI SCHEDULING CON PRELAZIONE

La differenza è che la CPU (server) può commutare ad un altro processo anche se il processo attualmente in esecuzione non è stato completato secondo un certo criterio.

Ogni volta che schedulo un processo, non è detto che se tolgo la CPU è stato completato, se viene prelazionato torna nella coda dei processi pronti per essere selezionato in un secondo momento.

Un dato processo, prima di essere completato, viene prelazionato più volte, questo comporta che lo schema di scheduling ha un overhead maggiore rispetto agli schemi senza prelazione.

È usato nei SO multi programmati e time sharing.

SCHEDULING ROUND-ROBIN

L'obiettivo del round robin, non è tanto posto sulle prestazioni del Sistema, piuttosto sono i tempi di risposta per l'utente, se ho tanti processi sottomessi dagli utenti, cerca di fare il meglio.

Si fissa come parametro uno slot temporale δ , che è la massima quantità di tempo per cui, schedulato un dato processo, questo può essere eseguito. Nel frattempo possono succedere tre cose:

1. Il processo durante quello slot di tempo viene completato
2. Prima che scada lo slot si può verificare un evento che potrebbe prelazionarlo.
3. Allo scadere dello slot temporale, non abbiamo completato l'esecuzione del processo, ritorna in coda alla coda e attenderà il suo turno quando sarà schedulato per δ secondi.

Lo scheduling RR fornisce dei tempi di servizio comparabili a tutti i processi che sono CPU bound quindi se facciamo una valutazione, vedremo che i W sono comparabili per tutti i processi che sono schedulati con una strategia RR.

Il valore di W dipende anche da quanti processi ci sono nel sistema (più ne sono più è elevato).

Abbiamo detto che in un sistema multiprogrammato per migliorare il throughput devo favorire i processi più brevi, il RR da questo punto di vista non fa testo perché il suo obiettivo non è migliorare le prestazioni quanto piuttosto dare risposte brevi ai processi e migliorare l'esperienza dell'utente.

Scheduling Round-Robin (RR) con Time-Slice

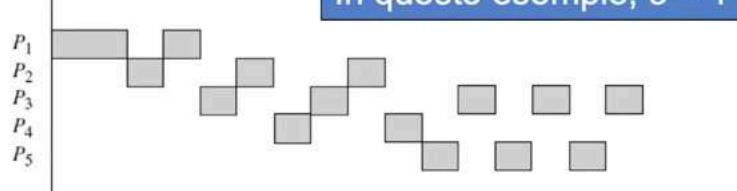
Time of scheduling		0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
Position of	P_1	1	1	2	1												
Processes in	P_2			1	3	2	1	3	2	1							
ready queue	P_3				2	1	3	2	1	4	3	2	1	2	1	2	
(1 implies	P_4					3	2	1	3	2	1						
head of queue)	P_5									3	2	1	2	1	2	1	
Process scheduled	P_1	P_1	P_2	P_1	P_3	P_2	P_4	P_3	P_2	P_4	P_5	P_3	P_5	P_3	P_5	P_3	

c	t_a	w
4	4	1.33
9	7	2.33
16	13	2.60
10	6	3.00
15	7	2.33

$$\bar{t}_a = 7.4 \text{ seconds}, \bar{w} = 2.33$$

c: completion time of a process

In questo esempio, $\delta = 1$



Process	P_1	P_2	P_3	P_4	P_5	0	5	10	15	Time →
Admission time	0	2	3	4	8					
Service time	3	3	5	2	3					

Abbiamo i soliti processi con i tempi di arrivo e di servizio.

Nella tabella rappresentiamo la posizione dei processi all'interno della coda dei processi pronti. Abbiamo detto che ogni processo viene eseguito per una certa quantità di tempo quando viene

schedulato, allo scadere viene prelazionato e rimesso all'ultima posizione della coda dei processi pronti. Ogni volta dobbiamo quindi vedere qual è la posizione di processi nella loro coda. Dobbiamo sottolineare una cosa: se prelaziono un processo e contemporaneamente ne arriva uno nuovo nella coda, quale va in coda all'ultimo posto? **Assumiamo che all'ultimo posto va quello prelazionato.**

Il δ è posto a 1 in questo esempio.

➤ Analisi della tabella:

Al tempo 0 abbiamo un solo processo in coda, P1, e la sua posizione all'interno dei processi è la prima e viene schedulato, P1 per completarsi impiega 3 unità, quando scatta $\delta = 1$, P1 non è stato completato, viene prelazionato e viene messo nella coda dei processi pronti, all'istante 1 però non ci sono altri processi e P1 si trova nuovamente al primo posto e viene schedulato. Passa il tempo 1 e arriviamo all'istante 2 dove arriva il processo P2, P1 va all'ultimo posto della coda e P2 si trova al primo posto della coda che viene schedulato, all'arrivo dell'istante 3, P2 viene schedulato, arriva P3 e nella coda dei processi al primo posto c'è P1 che viene schedulato e completato.

All'istante 4, P1 non è più in coda e arriva P4 all'ultima posizione mentre al primo posto in coda abbiamo P3 che viene schedulato e così via...

A questo punto per ogni processo calcoliamo TA e W.
Per poterli calcolare dobbiamo calcolare i tempi di completamento c . Il processo P1 viene completato all'istante 4, siccome era arrivato a 0, il TA è 4 mentre il W è $4/3$ (tempo di servizio) quindi 1.33.

P2 ha $c=9$, TA è 7 e W è 2.33.

P3 ha c=16, TA 13, W 2.60.

P4 ha c=10, TA 6, W 3.

P5 ha c=15, TA 7, W 2.33

Osserviamo che il TA medio è alto rispetto agli schemi senza prelazione ma abbiamo detto che non è questo l'aspetto che il RR privilegia. Invece W è molto basso e simile agli schemi senza prelazione.

È chiaro che P1 ha il miglior W perché quando arriva viene subito schedulato, questo ci suggerisce che il W dipende dal carico del sistema (quanti processi sono in coda per essere schedulati).

VARIAZIONE DEL TEMPO DI RISPOSTA NELLO SCHEDULING RR

- Ricordiamo che per un processo il tempo di risposta $rt = n \times (\delta + \sigma)$, dove n è il numero di processi nel sistema

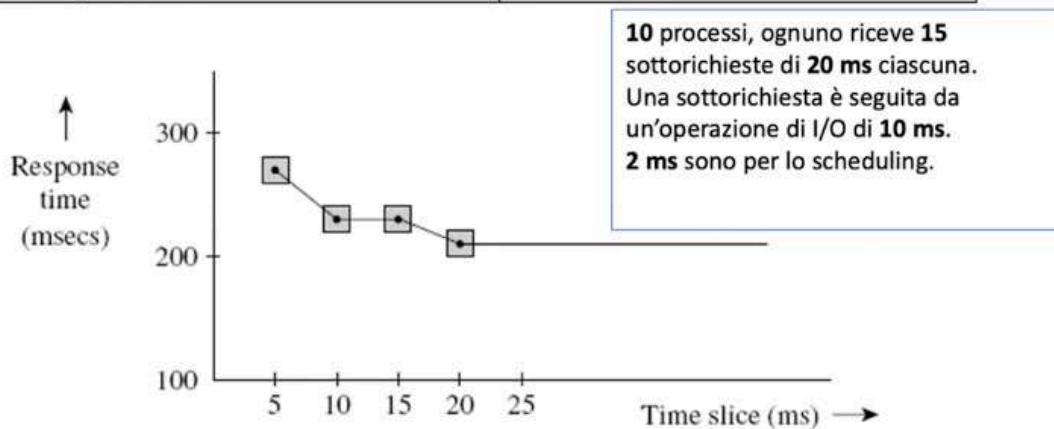
In realtà la relazione tra **rt** e **δ** è più complessa.

- Alcuni processi saranno bloccati per operazioni di I/O o per l'attesa di azioni degli utenti
 - Quindi **rt** è governato dal numero di processi attivi piuttosto che da **n**
- Se una richiesta ha bisogno più di **δ** secondi di tempo di CPU, sarà schedulata più di una volta prima di produrre una risposta
 - Per cui per piccoli valori di **δ** , i valori dei tempi di risposta possono essere più elevati.

Esempio: Variazione del Tempo di Risposta nello Scheduling RR

- Per piccoli valori di δ , l' rt di una richiesta può essere maggiore

Time slice	5 ms	10 ms	15 ms	20 ms
Average rt for first subrequest (ms)	248.5	186	208.5	121
Average rt for subsequent subrequest (ms)	270	230	230	210
Number of scheduling decisions	600	300	300	150
Schedule length (ms)	4200	3600	3600	3300
Overhead (percent)	29	17	17	9



Supponiamo di avere il RR in un sistema in cui ipotizziamo 10 processi che si comportano alla stessa maniera: ognuno costituito da 15 sottorichiesta dove ognuna di esse viene completata in 20 millisecondi. In questo scenario supponiamo che una sotto richiesta viene seguita da un'operazione di I/O che richiede 10 millisecondi e supponiamo che l'algoritmo di scheduling, sigma, richiede 2 millisecondi.

Vogliamo vedere quanto variano i tempi di risposta per la prima sottorichiesta e poi per tutte le altre successive alla prima media in funzione della variazione dello slot temporale δ .

Se considero lo slot a 20 millisecondi avremo un tempo breve mentre più diminuisce, più impiega tempo.

Il valore medio dei tempi di risposta, nel caso dello slot a 20 millisecondi abbiamo un valore di 210 mentre più diminuisce, più

aumentano i valori. Lo stesso vale per il numero delle decisioni di scheduling e per la lunghezza dello scheduling e per la percentuale di overhead.

LEAST COMPLETED NEXT (LCN)

Uno schema con prelazione alternativo è LCN, cioè seleziono quel processo che è stato eseguito fino a quel momento per minor tempo.

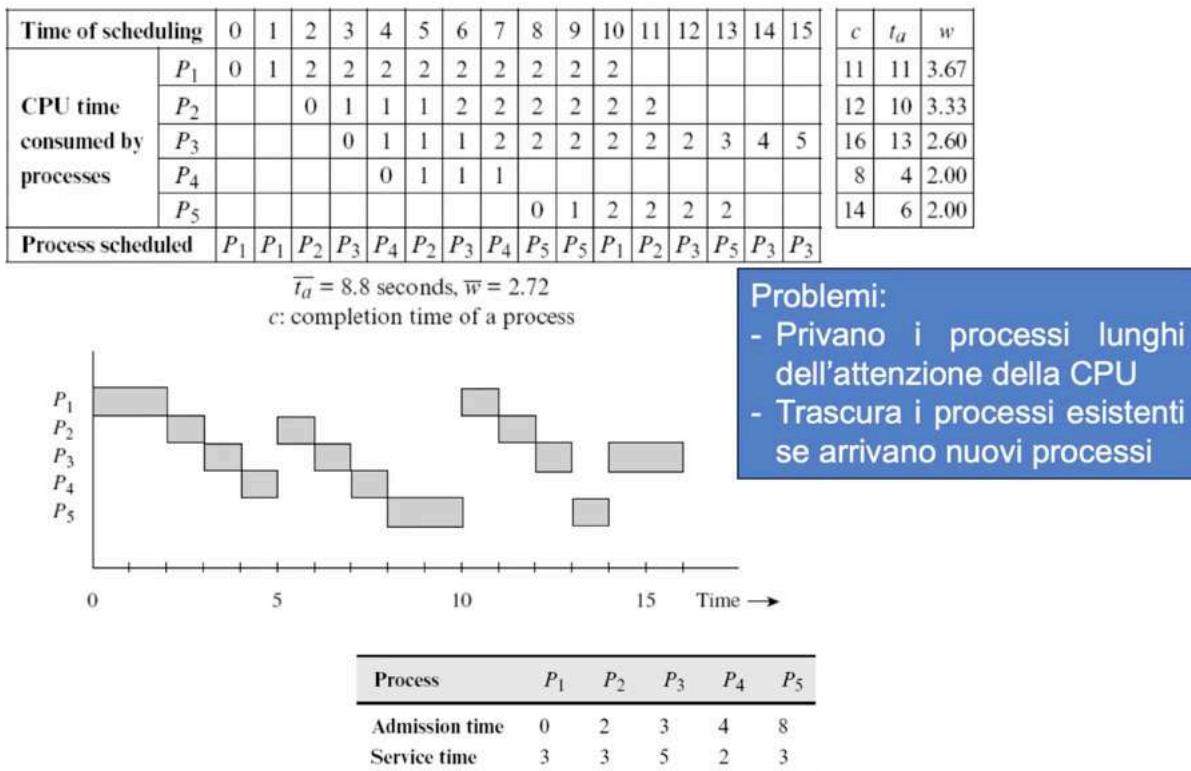
In questo tipo di schema, la natura del processo non impatta sulla schedulazione e sul suo avanzamento. Più o meno progetzano tutti i processi in maniera uguale in termini di CPU usata, ciò significa che i processi che durano meno comunque finiscono prima e a differenza del RR, il LCN fornisce migliori prestazioni complessive.

Come inconveniente ha una possibile starvation dei processi lunghi per motivi che già conosciamo.

Questo schema favorisce i processi appena arrivati: se un processo arriva significa che ha usato la CPU meno di tutti e in qualche modo viene selezionato il processo nuovo.

Può accadere che in ogni istante potrebbero esserci un uguaglianza di valori tra più processi che hanno gli stessi valori nell'utilizzo dei tempi di CPU, in quel caso vado a scegliere quello che non è elaborato da più tempo.

Least Completed Next (LCN)



Abbiamo da considerare, rispetto al RR, il tempo di CPU usato dai processi. Abbiamo i soliti processi con tempo di ammissione e servizio.

➤ Analisi della tabella:

All’istante 0 il processo P_1 è l’unico presente nel sistema, ha usato per 0 istanti la CPU e viene schedulato, ora all’istante 1 P_1 continua a essere il processo presente da solo e il nostro processo ha usato per 1 istante di tempo la CPU, il processo viene schedulato di nuovo.

All’istante 2 arriva P_2 che ha usato per 0 istanti la CPU e quindi viene schedulato.

All’istante 3 arriva P_3 che ha usato per 0 la CPU e viene schedulato.

All’istante 4 arriva P_4 che ha usato per 0 la CPU e viene schedulato.

All'istante 5 non arriva nessun altro processo e ho P1 con 2 istanti di CPU mentre P2, P3, P4 e viene schedulato quello che non viene eseguito da più tempo: P2. E così via.

Osserviamo che sono favoriti i processi appena arrivati e i più lunghi rischiano di andare in starvation e i processi in generali sono trascurati in favore dei nuovi arrivati.

- I tempi di TA medio elevato e W abbastanza basso ma più elevato per i processi P1 e P2 perché sono seguiti da nuovi processi che arrivano e devono attendere per molto tempo
-

SHORTEST TIME TO GO (STG)

È una versione con prelaziono dello SJF.

L'idea è quella di selezionare il processo che è il minor requisito di tempo restante cioè quello lì a cui manca meno per essere completato.

È in grado di favorire i processi che sono più brevi e favorisce quei processi che sono prossimi a essere completati, anche rispetto ai nuovi arrivati.

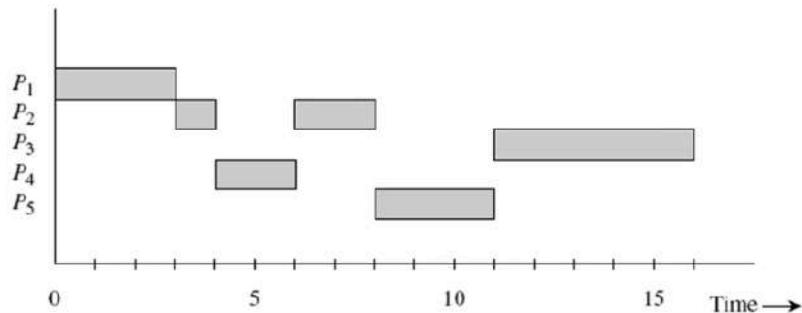
Migliora i tempi di TA e W ma in ogni caso i processi lunghi possono soffrire di starvation.

Shortest Time to Go (STG)

Time of scheduling		0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	c	t_a	w
Remaining CPU time requirement of a process	Process scheduled	P_1	3	2	1												3	3	1.00	
	P_2			3	3	2	2	2	1								8	6	2.00	
	P_3				5	5	5	5	5	5	5	5	5	4	3	2	16	13	2.60	
	P_4					2	1										6	2	1.00	
	P_5									3	2	1					11	3	1.00	

$$\bar{t}_a = 5.4 \text{ seconds}, \bar{w} = 1.52$$

c : completion time of a process



Analogo alla politica SJF -> processi più lunghi possono andare in starvation

Abbiamo i soliti processi da analizzare.

Costruiamo la tabella e analizziamola:

Il criterio di schedulazione è rappresentato dal tempo restante per il completamento del processo e quindi indichiamo questo criterio sulla parte sinistra.

All'istante 0 abbiamo solo il processo P1 e vediamo quanto resta P1 per essere completato, in questo caso 3. È l'unico processo presente e viene schedulato nell'istante 0. All'istante 1 P1 ha altre 2 unità di tempo, è ancora l'unico processo e quindi viene rischedulato. Al tempo 2 entra in gioco anche P2. Per completare P1 ci vuole 1 tempo e per P2 ci vogliono 3 unità. Scheduliamo di nuovo P1 perché impiega meno tempo e all'istante 3, P1 ha completato la sua attività perché richiedeva 3 unità di servizio. Arriva anche P3 che ne richiede 5. Al tempo 3 abbiamo P2 e P3 e

viene schedulato P2. All'istante 4 arriva il processo P4 che ha 2 unità per essere completato per cui viene schedulato P4 (anche P2 ha 2 unità ma viene schedulato quello che non viene schedulato da più tempo). All'istante 5 P4 gli rimane 1 unità e quindi viene schedulato e completato.

All'istante 6 abbiamo P2 e P3 e viene schedulato P2 che ha 2 unità.

All'istante 7 abbiamo P2 di nuovo e all'istante 8, P2 è completato e arriva anche P5 che viene schedulato perché ha 3 unità di tempo.

Visto che rimangono solo questi due e non ne arriveranno altri, viene schedulato P5 prima e poi P3 fino al completamento.

Possiamo ora calcolarci il tempo di completamento: P1 termina a 3.

P2 termina a 8. P3 termina a 16. P4 termina 6. P5 termina a 11.

c	t_a	w
3	3	1.00
8	6	2.00
16	13	2.60
6	2	1.00
11	3	1.00

Ci calcoliamo i tempi di TA e W.

Osserviamo che il TA medio è migliorato e a ogni istante quello favorito è un processo a cui resta poco tempo per essere completato. Anche se arrivassero processi più brevi, verrebbero favoriti comunque quelli già presenti se hanno tempi minori.

SCHEDULING DEI PROCESSI

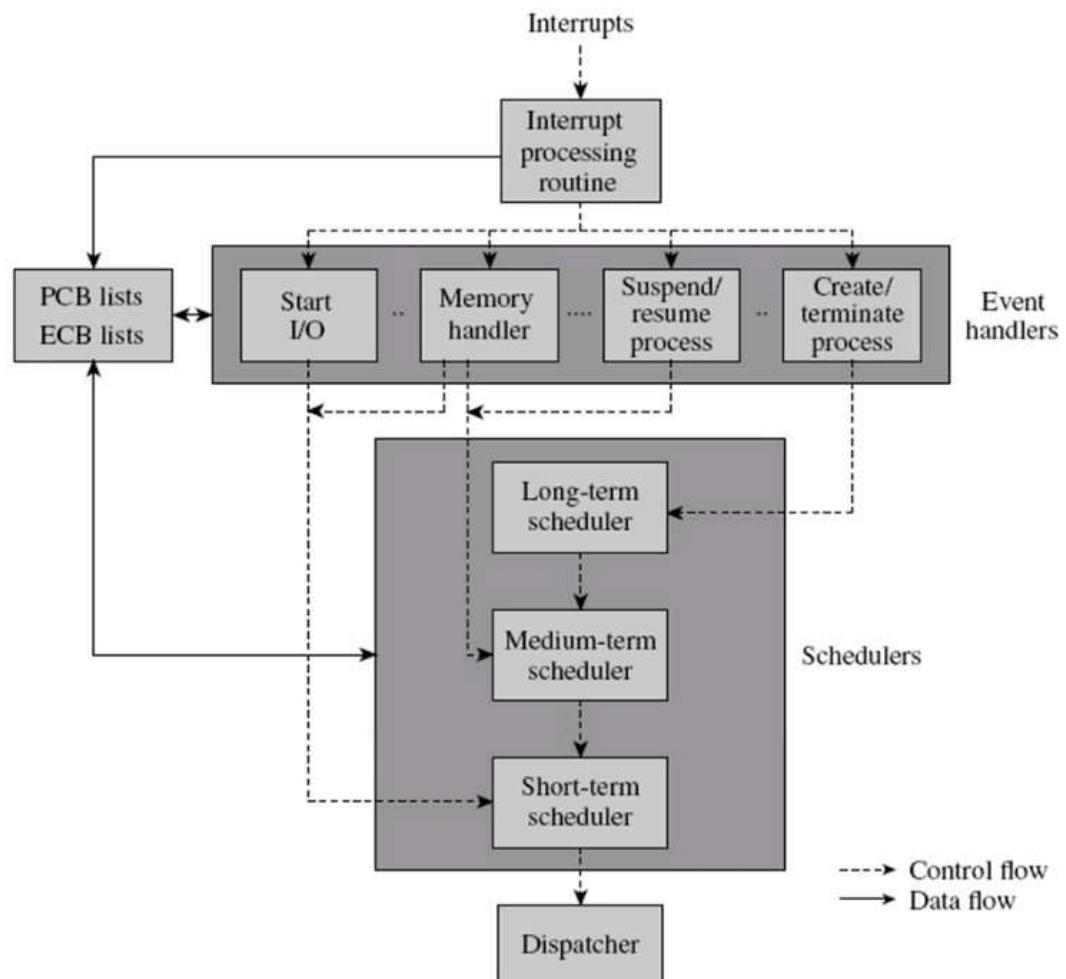
- SCHEDULING IN PRATICA

Abbiamo visto che lo scheduling è importante perché attraverso esso il SO è in grado di garantire una buona prestazione del sistema e di servizio utente. Per poter fare questo, il SO deve essere in grado di adattare il suo funzionamento sulla base dei processi presenti, a quante richieste ci sono da parte degli utenti e sulla disponibilità delle risorse del sistema.

Questo giustifica che adattarsi a tutto ciò non è necessario un solo tipo di scheduler ma è necessario usarne diversi. Nei SO abbiamo visto che è possibile usare 3 scheduler:

- a. Lungo termine: decide quando ammettere un processo appena arrivato allo scheduling, a seconda della natura (CPU,I/O bound) e disponibilità delle risorse.
- b. Medio termine: decide quando fare swap-out di un processo dalla memoria e quando ricaricarlo in modo che ci sia un numero sufficiente di processi ready in memoria.
- c. Breve termine: decide quale prossimo processo *ready* servire con la CPU e per quanto tempo. Chiamato anche scheduler di processo o scheduler.

Gestione degli eventi e Scheduling



Il nostro SO è guidato da eventi e che questi quando si verificano sono verificato per mezzo di interrupt.

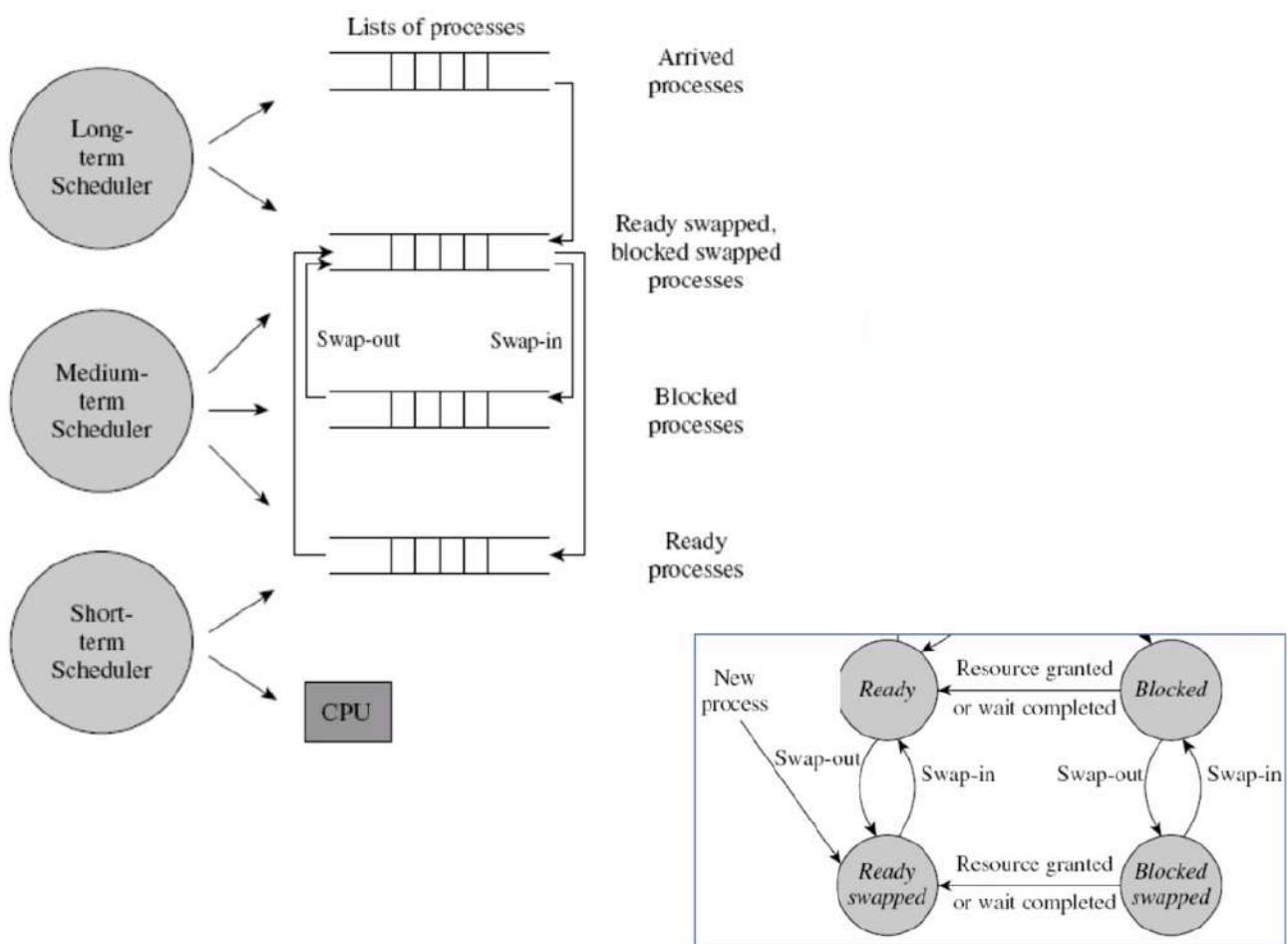
Il sistema è sempre pronto ad identificare l'occorrenza di questi eventi, e quando si verifica un interrupt, si richiede l'attenzione del kernel che serve la routine di gestione dell'interrupt. Questa routine fa un'operazione di salvataggio del contesto perché il processo viene interrotto dall'evento e va a invocare il gestore dell'evento.

Il gestore dell'evento innanzitutto va a cambiare lo stato del processo che è influenzato dall'evento che è occorso (es processo da blocked a ready) e poi invoca uno dei 3 scheduler.

Significa che se l'evento è un evento che ha creato un nuovo processo, questo gestore invoca lo scheduler a lungo termine che decide se il processo appena creato può essere messo in coda. Se l'evento è la sospensione di un processo o un ripristino del processo, il gestore richiama lo scheduler a medio termine che deve decidere se quel processo deve essere sottoposto a swap in o swap out.

Se non c'è disponibilità di memoria, il gestore richiama lo scheduler a breve termine che richiama in swap out il processo.

Esempio: Scheduling a Lungo, Medio e Breve Termine in un Sistema Time-Sharing

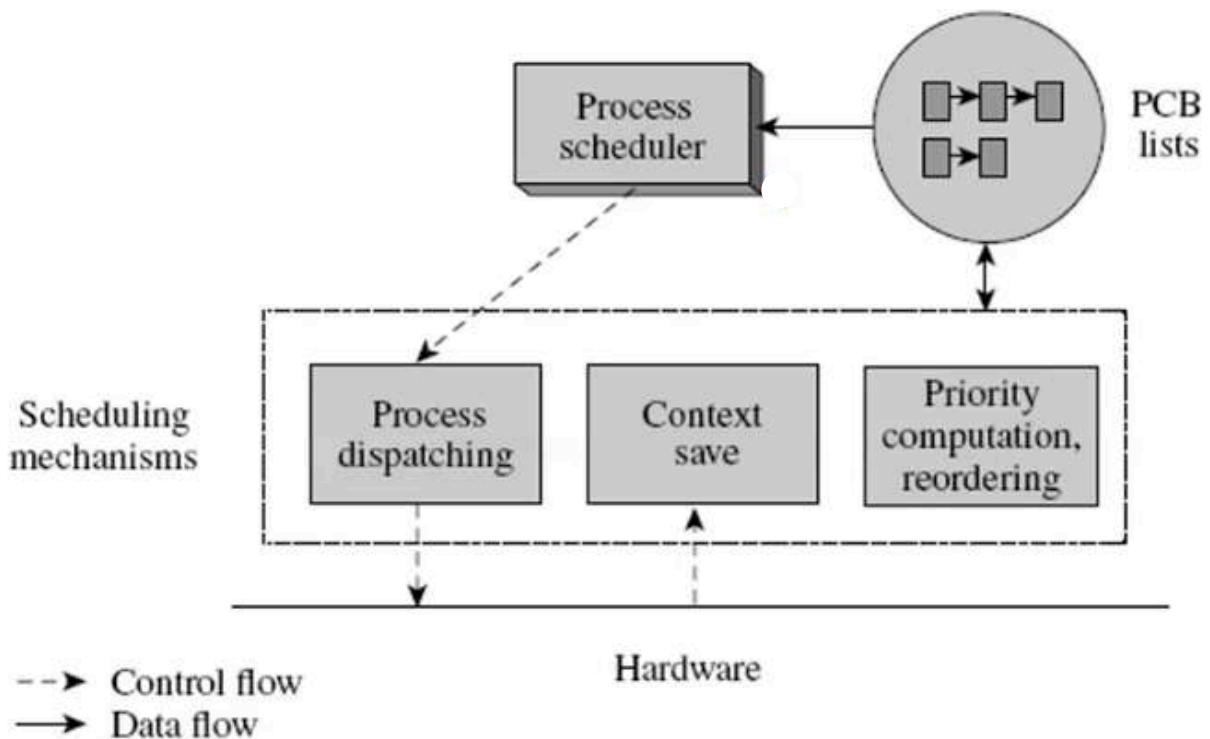


Transizioni di stato legate alle attività dei tre scheduler analizzati.

Quando arriva un nuovo processo, tipicamente, lo scheduler a lungo termine decide di ammetterlo e quando lo fa lo inserisce nella lista dei processi pronti ma swapped. Fa una copia del codice del processo all'interno dello spazio di swap in maniera tale da averlo nella coda dei processi ready swapped.

Lo scheduler a medio termine interviene sulla coda dei processi bloccati o pronti nello stato swap out. Le transizioni che possono avvenire dipendono dagli stati in cui si può trovare un processo. Se mi trovo in ready swapped, posso passare in ready a causa di un'operazione di swap in (fatta dallo scheduler a medio termine), lo scheduler può anche portare un processo dallo stato blocked swapped a uno blocked. Lo swap out fa le transizioni inverse. Lo scheduler a breve termine porta il processo al dispatcher e poi viene eseguito dalla CPU.

SCHEDULING IN PRATICA: STRUTTURE DATI E MECCANISMI DI SCHEDULING



Lo scheduler preleva un processo dalla lista e lo invia al dispatcher che si deve preoccupare di caricare il PSW e il GPR(general purpose) del processo selezionato.

Se la lista è vuota si implementa un *loop idle*, un ciclo dove non si fa nulla.

Questo loop comporta uno spreco di risorse da parte della macchina ma per la maggior parte dei sistemi desktop e portatile non è un grosso problema, al contrario di sistemi embedded o reti di sensori, avere un loop idle per andare a intercettare gli eventi non è l'ideale come soluzione. In questi sistemi si presuppone che la CPU possa lavorare in diversi stati operativi ed essere totalmente in stato di sleep.

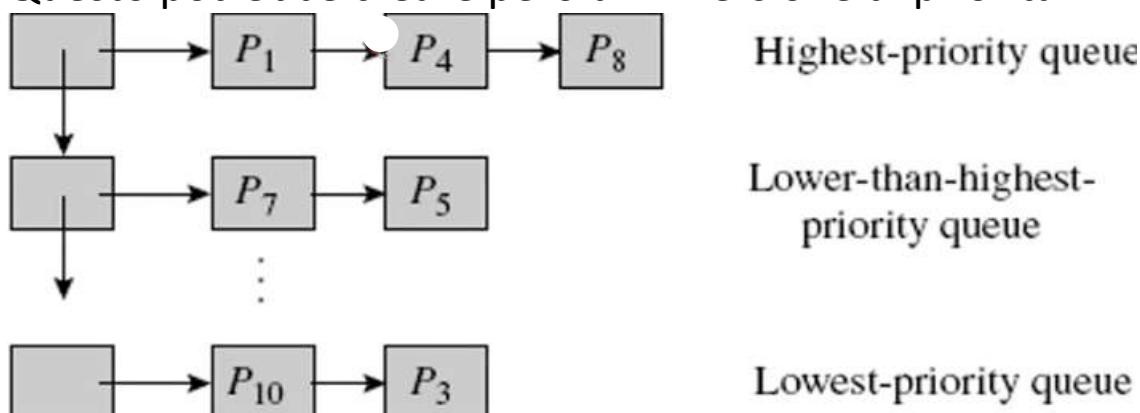
SCHEDULING BASATO SU PRIORITA'

È mantenuta una lista separata di processi ready per ogni valore di priorità. È organizzata come coda di PCB.

C'è l'intestazione della lista che ha due puntatori: uno punta al primo elemento della coda e l'altro punto all'elemento di intestazione della coda di priorità inferiore. L'overhead della gestione della coda non è costituito dal numero di processi attivi quanto dal numero di code.

Può portare a starvation dei processi a bassa priorità che può essere gestita attraverso l'aging che man mano aumenta le priorità dei processi.

Questo potrebbe creare però un'inversione di priorità



SCHEDULING RR CON TIME SLICING

Può essere implementato con una lista singola di PCB dei processi ready, la lista è organizzata come una coda.

Lo scheduler rimuove il primo PCB dalla coda e schedula il processo corrispondente:

- Se il time slice scade, il PCB è messo alla fine della coda
- Se il processo avvia un'operazione di I/O, il suo PCB è aggiunto alla fine della coda quando la sua operazione di I/O è completata

Un processo col passare del tempo migra dal fondo alla testa della coda per essere schedulato.

SCHEDULING MULTIVELLO

È uno schema molto adottato nei sistemi moderni.

Prevede di combinare lo scheduling di priorità con uno RR.

Si prevedono diverse code di scheduling dove ciascuna coda di un fissato livello di priorità viene gestita con una strategia RR. L'idea è che si può attribuire alle code di priorità maggiore i processi interattivi (quelli con tempi di risposta immediati) mentre ai processi non interattivi si attribuiscono code di priorità inferiore. All'interno di ciascuna coda si può fissare un time slice adeguato. La coda ad alta priorità ha un piccolo time slice. La coda a bassa priorità ha un time slice maggiore.

Un processo in testa ad una coda è schedulato solo se le code per tutti i livelli più elevati di priorità sono vuote.

Le priorità sono determinate a priori e non c'è possibilità di spostare i processi tra le code: determina 2 contro indicazioni:

1. Sistema non tollerabile ai guasti e se si fissa male le priorità influenza il sistema
2. Situazioni come starvation non possono essere risolte

SCHEDULING ADATTIVO MULTILIVELLO

Lo scheduler varia la priorità di un processo in modo che esso abbia un time slice consistente con il suo requisito di CPU.

Lo scheduler determina il livello di priorità corretto tenendo traccia i comportamenti del processo in modo da poter spostare il processo da una coda all'altra ovviando al problema dello scheduler multilivello.

SCHEDULING REAL-TIME

Il problema cambia poiché uno scheduling real-time tiene conto di due vincoli e deve soddisfare le Deadline delle applicazioni.

La differenza tra scheduling ordinario e real time è che abbiamo vincoli temporali da rispettare.

Nella maggior parte delle applicazioni real-time i processi sono interattivi e hanno delle scadenze importanti e una strategia di scheduling deve avere modi per trasformare la deadline di un'applicazione in una per i processi.

Inoltre i processi possono essere periodici, le differenti istanze di un processo possono arrivare a intervalli fissi e tutte devono soddisfare le rispettive deadline.

PRECEDENZE DI PROCESSO E SCHEDULAZIONI AMMISSIBILI

Abbiamo un certo numero di processi che interagiscono tra di loro in maniera tale da garantire un ordine di esecuzione (qualcuno viene eseguito prima di un altro ecc...). In questa situazione nel momento in cui lo scheduler deve decidere quale processo eseguire, deve tener conto delle deadline e della dipendenza dei vari processi interagenti: un processo non può iniziare la propria attività senza che l'altro abbia completato il suo lavoro.

Per fare questo prendiamo in considerazione queste dipendenze che prendono il nome di precedenze e dobbiamo vedere come una tecnica di scheduling real time tiene traccia di queste precedenze.

Lo fa mediante un grafo ma prima di definirlo introduciamo una notazione $P_i \rightarrow P_j$ ciò significa che P_i precede P_j .

Un grafo delle precedenze dei processi (PPG) è un grafo orientato $G \equiv (N, E)$ tale che $P_i \in N$ rappresenti un processo, ed un arco $(P_i, P_j) \in E$ implica $P_i \rightarrow P_j$. Quindi, un cammino P_i, \dots, P_k in PPG implica $P_i \rightarrow^* P_k$. Un processo P_k è un discendente di P_i se $P_i \rightarrow^* P_k$.

\rightarrow^* significa discendenza diretta.

Si usano i grafi per stabilire se esiste una schedulazione tale per cui i vincoli temporali sono soddisfatti. Il modo in cui posso soddisfare i vincoli dipende dal tipo di sistema realtime.

Se ho un sistema hard realtime, devo assicurare il rispetto delle deadline.

Un sistema soft realtime può anche non garantire qualche rispetto delle deadline e i requisiti sono soddisfatti probabilisticamente.

Una schedulazione ammissibile è una sequenza di decisioni di scheduling che permette ai processi di operare in accordo con le precedenze e soddisfare i requisiti dell'applicazione.

Uno scheduling real time deve individuare una sequenza ammissibile se ne esiste una.

APPROCCI ALLO SCHEDULING REAL-TIME

Ci sono diverse strategie per implementare lo scheduling realtime:

1. Scheduling statico: tipo di schedulazione che viene determinata prima che il sistema funzioni, per determinarla si vanno a considerare le operazioni tra processi, le periodicità, i vincoli legati alle risorse e le deadline
2. Scheduler basato su priorità: si analizza le specifiche delle applicazioni real time in modo da assegnare le priorità ai processi.
3. Schedulazione dinamica: lo scheduling si esegue alla richiesta da parte di un processo: creo il processo in seguito alla richiesta solo se il processo può garantire il soddisfacimento dei vincoli temporali. (situazione tipica delle applicazioni multimediali: quando si richiede un servizio quello che si può fare è richiedere sia il momento di inizio, tempo per eseguire, risorse necessarie e tempo entro cui completare le operazioni, in questi sistemi multimediali c'è una richiesta così fatta, si vede se è possibile rispettare questi vincoli oppure non viene creato nemmeno).

Scheduling ottimista → ammette tutti i processi ma può perdersi qualche deadline.

SCHEDULING CON DEADLINE

Si basa sul selezionare sempre il processo con la deadline più prossima alla scadenza.

Quando abbiamo una deadline associata a un processo dipende come definiamo la deadline:

- Deadline di inizio: specifica entro cui iniziare il processo
- Deadline di completamento: tempi entro i quali un processo deve essere completato

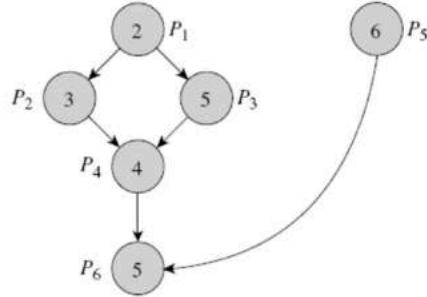
Il punto di partenza è un'applicazione realtime che ha un suo requisito globale di risposta e a partire da questo posso usare il grafo delle precedenze per calcolarmi per ogni processo qual è la sua deadline attraverso questa formula:

$$D_i = D_{application} - \sum_{k \in descendant(i)} x_k$$

dove $D_{application}$ è la scadenza dell'applicazione e x_k è il tempo di servizio del processo P_k

Ho la deadline complessiva D_i , per sapere la deadline del processo i -simo prendo la deadline della applicazione complessiva e gli sottraggo i tempi di servizio che dipendono da tale processo e faccio questo per ciascun processo.

Esempio: determinare le deadline dei processi



- Il totale dei tempi di servizio dei processi è 25 secondi
- Se l'applicazione deve produrre una risposta in 25 secondi, le deadline dei processi sarebbero:

Process	P ₁	P ₂	P ₃	P ₄	P ₅	P ₆
Deadline	8	16	16	20	20	25

Supponiamo di avere tale grafo delle dipendenze con 6 processi.

P₂ non può iniziare se non finisce prima P₁, lo stesso come p3.
P₄ non inizia se non finiscono P₂ e P₃. P₆ non inizia se non finisce P₄ ecc...

P₁ → *P₄ (P₄ è discendente di P₁).

I numeri all'interno di ciascun nodo rappresentano i tempi di servizio di ciascun processo.

$D_{application} = 25$ secondi.

Come calcolo D₁, D₂... D₆? Applicando la formula di prima.

$D_1 = 25 - (\text{somma dei processi che senza il completamento di } P_1 \text{ non possono iniziare}) \rightarrow 25 - (3+5+4+5) = 8$

$$D_2 = 25 - (4+5) = 16$$

$$D_3 = 25 - (4+5) = 16$$

$$D_4 = 25 - 5 = 20$$

$$D_5 = 20$$

$$D_6 = 20$$

Una volta che ho le deadline devo avere una strategia che seleziona i processi sulla base delle deadline.

Usiamo la **Earliest Deadline First (EDF)** che seleziona sempre il processo con la deadline più prossima. Si considera la sequenza di scheduling dei vari processi.

Sia **seq** la sequenza in cui i processi sono elaborati, se **pos(P_i)** è la posizione di P_i nella sequenza delle decisioni di scheduling, lo sfioramento delle deadline di P_i , ovvero D_i , non avviene se:

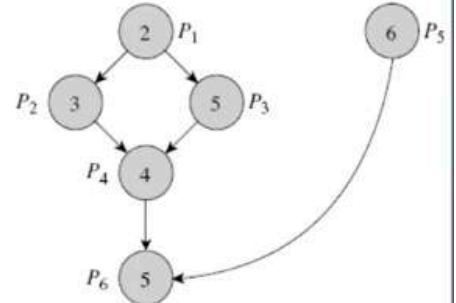
$$\sum_{k: pos(P_k) \leq pos(P_i)} x_k \leq D_i$$

- La condizione vale quando esiste una schedulazione ammissibile
- **Vantaggi:** semplicità e natura senza prelazione
- Buona politica per lo scheduling statico

Scheduling con Deadline (cont.)

Time	Process completed	Deadline overrun	Processes in system	Process scheduled
0	-	0	$P_1 : 8, P_2 : 16, P_3 : 16, P_4 : 20, P_5 : 20, P_6 : 25$	
2	P_1	0	$P_2 : 16, P_3 : 16, P_4 : 20, P_5 : 20, P_6 : 25$	P_1
5	P_2	0	$P_3 : 16, P_4 : 20, P_5 : 20, P_6 : 25$	P_2
10	P_3	0	$P_4 : 20, P_5 : 20, P_6 : 25$	P_3
14	P_4	0	$P_5 : 20, P_6 : 25$	P_4
20	P_5	0	$P_6 : 25$	P_5
25	P_2	0	-	-

- P4: 20 indica che P4 ha la deadline a 20
- P2, P3 e P4, P5 hanno le stesse deadline
 - Sono possibili altre tre schedulazioni
 - Nessuna incorre nello sforamento delle deadline



Siamo nella situazione precedente con lo stesso grafo e le relative deadline.

Al tempo 0 abbiamo i 6 processi con le relative deadline. In questo momento andiamo a schedulare è quello con la deadline più vicina che è P1. P1 viene eseguite per due unità di tempo e viene completato all'istante 2. Vado a selezionare P2 che richiede 3 unità, a 5 ha finito e seleziono P3 e così via. Alla fine seleziono l'ultimo processo che verrà completato secondo la deadline.

Quello che vediamo è che P2,P3,P4,P5 hanno stesse deadline quindi vi sono diverse schedulazioni ammissibili (avrei potuto scegliere prima P3 e poi P2 ecc...).

Questo è un algoritmo che ha un problema: nel momento in cui abbiamo schedulazioni ammissibili, le possiamo identificare

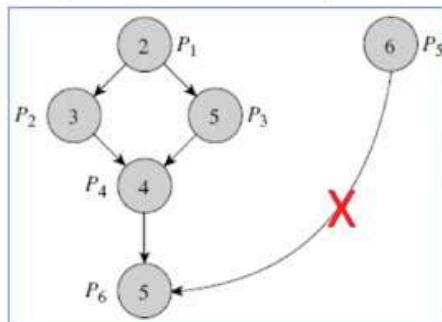
tranquillamente. È però un algoritmo che non è in grado di anticipare quanti processi non rispettano la rispettiva deadline se ci sono processi non ammissibili.

Esempio: problemi con lo scheduling EDF

- Se rimuoviamo l'arco (P5, P6) dal PPG
 - Due applicazioni indipendenti: P1-P4 e P6, e P5
 - Se tutti i processi devono essere completati in 19 secondi
 - Non esiste una soluzione ammissibile
 - Deadline dei processi

Process	P_1	P_2	P_3	P_4	P_5	P_6
Deadline	2	10	10	14	19	19

- Lo scheduling EDF può schedulare i processi come segue:
 - P1, P2, P3, P4, P5, P6 oppure P1, P2, P3, P4, P6, P5
 - Quindi il numero dei processi che non rispettano le rispettive deadline non è predicibile



In questo caso P5 è un processo indipendente e abbiamo quindi due applicazioni indipendenti (P1,p2,p3,p4,p6) e (P5).

Supponiamo che il requisito sia $D_{applicazione} = 19$.

Non esiste alcuna schedulazione ammissibile tale per cui tutti i processi si completano nelle rispettive deadline garantendo che l'applicazione termini in 19 secondi.

Seguo la procedura di prima e ricalcolo le deadline.

Se applichiamo EDF succede che posso avere 2 schedulazioni diverse:

posso schedulare P1,P2,P3,P4,P5,P6 OPPURE P1,P2,P3,P4,P6,P5. Quindi il numero dei processi che non rispettano le deadline non è predicibile.

AMMISSIBILITÀ DELLA SCHEDULAZIONE PER PROCESSI PERIODICI

Schedulazione real time in presenza di processi periodici.

I processi periodici sono processi che possono ripetersi nel tempo ad intervalli regolari. Quando ho processi di questo tipo devo considerare l'intervallo di tempo nel quale si ripete e quelle sono le deadline dei processi.

Supponiamo che i processi non eseguano operazioni di I/O e che siano indipendenti e non abbiamo un grafo delle dipendenze.

Process	P_1	P_2	P_3
Time period (ms)	10	15	30
Service time (ms)	3	5	9

Per illustrare la procedura abbiamo la tabella con 3 processi periodici.

In questo caso si va a considerare la frazione di tempo di CPU usata da un processo, facendo il rapporto tra il tempo di servizio e il periodo $\rightarrow P_i = x_i/T_i$

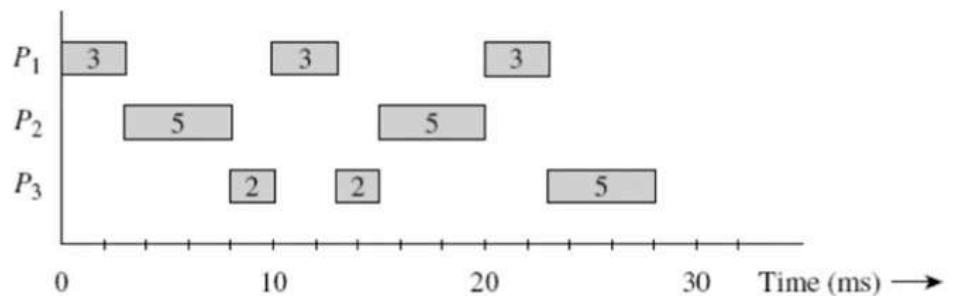
$$P1 = 10/3; \quad P2=15/5; \quad P3=30/9;$$

se sommo questi tempi di servizio, la somma è 0.93. In pratica, se l'overhead della CPU è trascurabile, è possibile servire i tre processi.

In generale, se abbiamo n processi periodici, che non eseguono I/O operation, è possibile determinare una schedulazione ammissibile se la somma di CPU di ciascun processo è ≤ 1 .

Scheduling Rate Monotonic (SRM)

- Determina i tassi a cui il processo deve ripetersi
 - Tasso di $P_i = 1 / T_i$
- Assegna lo stesso rate come priorità del processo
 - Un processo con un periodo più piccolo ha maggiore priorità
- Impiega uno scheduling basato su priorità
 - Può completare le sue operazioni prima



Algoritmo di scheduling per processi periodici.

Lo SRM va a determinare i tassi a cui il processo deve ripetersi.

- Tasso di $P_i = 1 / T_i$

Questo rate viene usato per determinare la priorità dei processi.

Il processo che ha frequenza maggiore ha priorità maggiore.

La priorità la si calcola facendo $\frac{1}{1 / T_i}$, determinate le priorità dei processi sulla base del loro tasso, si usa uno scheduling con priorità.

Nel caso dei tre processi visti in precedenza, P1 P2 P3.

P1 ha priorità maggiore mentre P3 è il minore.

Vado a schedulare P1 inizialmente, a questo punto lo eseguo per 3 unità di tempo, dopo di che vado a selezionare quello con tasso maggiore, cioè P2. Eseguo P2 per 5 unità, dopo di che eseguo P3 che dovrebbe durare 9 unità ma dopo 2 unità arrivo a 10 e quindi tocca a P1 che viene usato per 3 unità e completato e così via...

Il processo P3 viene completato a 28.

Quello che può succedere è che in ogni caso questo scheduling non garantisce uno scheduling ammissibile in tutte le situazioni. In questo esempio P3 è terminato a 28ms, supponiamo che P3 si ripeta a ogni 27 ms invece di 30. Succede che le priorità singole cambiano ma quelle relative restano fisse.

La sequenza di schedulazione è la stessa, l'unico problema è che abbiamo sforato di 1 la sua deadline e la procedura di schedulazione non garantisce uno scheduling ammissibile.

In generale è stato dimostrato che se una applicazione ha molti processi allora è possibile che non SRM non rispetti alcune deadline se non viene rispettata questa relazione:

$$\sum_{i=1}^m \frac{x_i}{T_i} \leq m(2^{\frac{1}{m}} - 1) \rightarrow 0.69 \text{ per } m \rightarrow \infty$$

Dove $\frac{x_i}{T_i}$ sono uso della CPU e m sono i processi del sistema

Al crescere di $m \rightarrow \inf$, questo numero arriva a 0.69. Significa che questo scheduling non garantisce che tutti i processi siano eseguiti nelle deadline se la somma dei tempi di utilizzo sfiora il 69%.

La variante scheduling guidato da deadline assegna dinamicamente le priorità ai processi sulla base delle loro deadline correnti. Con questa variante si raggiunge un tasso di

uso di CPU del 100% ma le prestazioni degradano alla lunga perché modificare priorità in corso d'opera aggiunge overhead.

LEZIONE 3/05/21

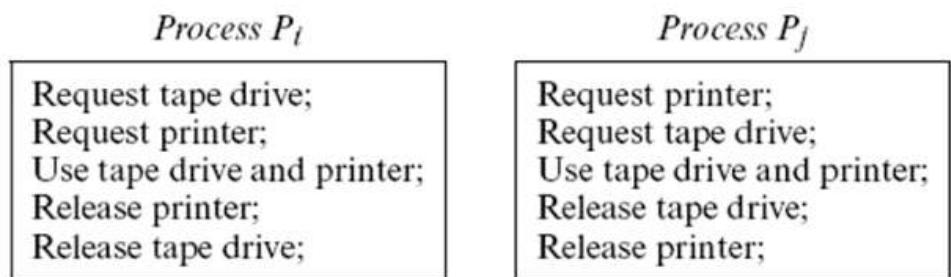
DEADLOCK

Cos'è un deadlock?

Un deadlock è una situazione dove abbiamo un certo numero di processi in insieme D dove ciascun processo soddisfa due condizioni:

1. Il processo P_i è bloccato in qualche evento E_j
2. L'evento E_j può essere generato solo da un altro processo in D. Siccome sono tutti bloccati, gli eventi non verranno mai generati.

- **Deadlock di risorsa -> preoccupazione primaria di un SO**



I deadlock possono verificarsi anche nella sincronizzazione e la comunicazione di messaggi <- **preoccupazione utente**

Esempio deadlock se si può verificare al momento di allocazione di risorse:

supponiamo che il processo P_i richiede un'unità a nastro, una stampante, usa la stampante e nastro, rilascia stampante e poi nastro.

P_j fa la stessa cosa ma con diverso ordine di richiesta: chiede prima stampante e poi nastro, le usa e le rilascia.

Questa è una situazione di deadlock:

se P_i richiede nastro, P_j richiede stampante: in questo momento vengono allocate le rispettive unità senza problema, successivamente P_i richiede la stampante ma si blocca perché è allocata a P_j che si blocca a sua volta poiché il nastro lo ha P_i . Entrambi sono bloccati in attesa di risorsa e si verifica uno stallo.

DEADLOCK NELL'ALLOCAZIONE DI RISORSE

Per definire il problema dobbiamo definire tutti gli ingredienti che giocano un ruolo nel deadlock:

in un elaboratore ci sono diversi classi di risorse (disco, memoria centrale) e nell'ambito di ciascuna classe ci sono più istanze.

Indichiamo con R_i una classe e una R_j un'istanza di risorsa della classe.

Definite le risorse in classe e istanze, quando si alloca una risorsa a un sistema si possono verificare 3 eventi:

1. Richiesta di una risorsa
2. Effettiva allocazione della risorsa
3. Rilascio della risorsa
 - a. La risorsa rilasciata può essere allocata a d un altro processo

- Eventi legati all'allocazione di risorse

Event	Description
Request	A process requests a resource through a system call. If the resource is free, the kernel allocates it to the process immediately; otherwise, it changes the state of the process to <i>blocked</i> .
Allocation	The process becomes the <i>holder</i> of the resource allocated to it. The resource state information is updated and the state of the process is changed to <i>ready</i> .
Release	A process releases a resource through a system call. If some processes are blocked on the allocation event for the resource, the kernel uses some tie-breaking rule, e.g., FCFS allocation, to decide which process should be allocated the resource.

Ci sono 4 condizioni per un deadlock di risorsa (simultanee)

1. Risorse non condivisibili: quella risorsa specifica deve essere una risorsa allocata in maniera esclusiva a un dato processo, significa che solo un processo può avere questa risorsa.
2. Non prelazionabilità della risorsa: quando una risorsa è allocata ad un processo non può essere prelazionata
3. Possesso e attesa: se un processo è bloccato in attesa di un'altra risorsa ma già possiede un'altra risorsa, questa non viene rilasciata finché non ottiene la risorsa che necessita
4. Attesa circolare: una catena di possesso e attesa per una serie di processi: P_i possiede una risorsa e attende risorse da P_j che a sua volta attende risorse da P_k che a sua volta attende le risorse di P_i .

È essenziale anche un'altra condizione per il deadlock:

- Nessun annullamento di richieste di risorse: un processo bloccato su una richiesta di risorsa non può annullarla.

MODELLARE LO STATO DI ALLOCAZIONE DELLE RISORSE

Il sistema deve avere un modo per rappresentare le info che riguardano le risorse in qualsiasi momento a quelli processi sono allocati.

Abbiamo due rappresentazioni:

1. Modello a grafo: ciascun processo può chiedere un'istanza per ogni classe di risorse disponibili
2. Modello a matrice: un processo può richiedere un qualsiasi numero di istanza di una classe di risorse.

Modello A Grafo RRAG

In un RRAG abbiamo nodi ed archi.

I nodi sono di due tipi:

1. Un cerchio, rappresenta i processi
2. Rettangolo, rappresenta classe di risorse
3. Puntini indicano le istanze di risorse.

È chiaro che ci sono archi di 2 tipi:

- Arco che va da un cerchio a un rettangolo: il processo ha richiesto una risorsa di una classe
 - Arco che va da rettangolo a rettangolo: al processo è stato allocato un'istanza di una classe
-
- Un arco di **allocazione** (R_k, P_j) è cancellato quando il processo P_j rilascia un'unità di risorsa della classe R_k
 - Un arco di **richiesta** (P_i, R_k) è cancellato ed è aggiunto un arco di **allocazione** (R_k, P_i) quando è concessa una richiesta in attesa del processo P_i per un'unità di una classe R_k

GRAFO DI ATTESA WFG

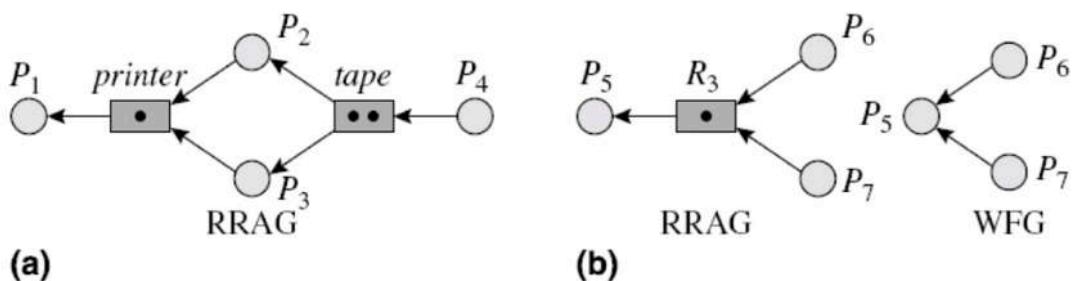
Può essere usato per descrivere lo stato delle risorse di un sistema in cui ogni classe di risorsa contiene solo un'istanza di risorsa.

C'è un unico nodo che rappresenta il processo.

- **Un arco è una relazione wait-for tra processi**

- **Un arco wait-for (P_i, P_j)** indica che
 - Il processo P_j occupa l'unità di risorsa della classe di risorsa
 - Il processo P_i ha richiesto la risorsa e si è bloccato su di essa
 - In sostanza, P_i attende che P_j rilasci la risorsa

RRAG vs WFG



(a) Grafo di richiesta e allocazione di risorse (RRAG)

(b) Equivalenza di RRAG e WFG quando ogni classe di risorsa contiene una sola unità

Nel grafo a abbiamo una classe nastro con due puntini (2 istanze) e classe stampante. Questo mostra che la stampante è stata allocata a P1 e P2 e P3 sono bloccati in attesa per la stampa. P2 e P3 hanno avuto allocato un nastro e P4 è in attesa di una delle due istanze del nastro per essere allocato.

Nel grafo b abbiamo un grafo RRAG che rappresenta una classe di risorse che contiene 1 istanza. La risorsa è allocata a P5 e P6 e P7 sono bloccati in attesa di R3. Il grafo RRAG può essere semplificato con la WFG come vediamo in figura.

CAMMINI NEI WFG E RRAG

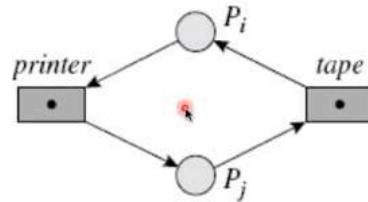
Un cammino è una sequenza di archi tali che il nodo destinazione di un arco è il nodo sorgente dell'arco seguente.

- Consideriamo un cammino in un RRAG: $P_1 - R_1 - P_2 - R_2 \dots P_{n-1} - R_{n-1} - P_n$.
Questo cammino indica che
 - Al processo P_n è stata allocata un'unità di R_{n-1}
 - Al processo P_{n-1} è stata allocata un'unità di R_{n-2} ed aspetta un'unità di R_{n-1} , ecc.
 - Nel WFG, lo stesso cammino sarebbe $P_1 - P_2 - \dots P_{n-1} - P_n$
 - Supponiamo che ogni classe di risorsa contenga un'unica unità
 - I cammini $P_1 - R_1 - P_2 - R_2 \dots P_{n-1} - R_{n-1} - P_n$ nel RRAG e $P_1 - P_2 - \dots P_{n-1} - P_n$ nel WFG sono privi di deadlock
-

MODELLI DI GRAFO

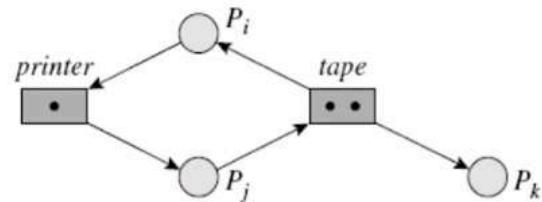
Nel caso si considerano classi di risorse con un'istanza, allora un ciclo determina uno stato di deadlock.

- Non può esistere un deadlock se un RRAG o un WFG **NON** contiene un ciclo



- Un ciclo in un RRAG non implica necessariamente un deadlock se una classe di risorse ha unità multiple

Quando P_k finisce,
La sua unità a nastro
può essere allocata a P_j



MODELLO A MATRICE

Lo stato di allocazione è rappresentato da due matrici:

- risorse allocate
- risorse richieste

Se un sistema ha n processi e r classi di risorse, ciascuna di tali matrice ha dimensioni $n \times r$.

	Printer	Tape
P_i	0	1
P_j	1	0
P_k	0	1

Allocated
resources

	Printer	Tape
P_i	1	0
P_j	0	1
P_k	0	0

Requested
resources

Total resources	Printer	Tape
Free resources	1	2
	0	0

{ Al processo P_i non è stata allocata una stampante
 { A P_j solo una stampante
 { A P_k solo nastro

P_i ha richiesto stampante

P_j ha richiesto nastro

P_k non ha richiesto nulla

Struttura dati “total resources” indica quante stampanti abbiamo in totale.

Struttura dati “free resources” indica quante risorse libere ci sono al momento.

GESTIONE DEI DEADLOCK

Abbiamo tutto ciò che ci serve.

Ci sono 3 tipologie di strategie:

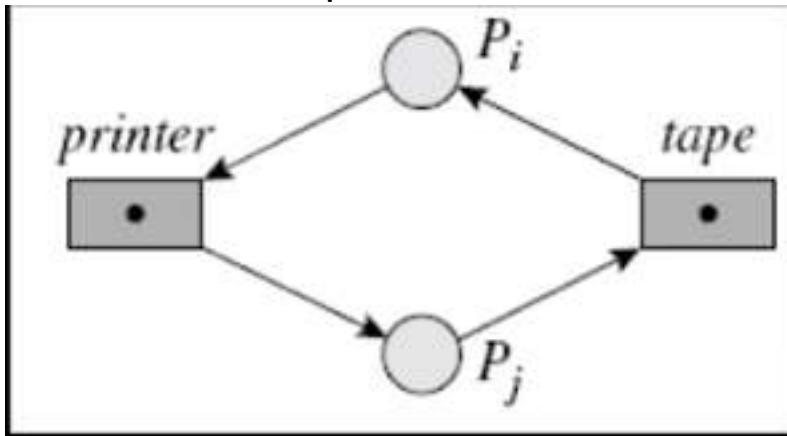
1. ammettere che in un momento ci possa essere un deadlock e risolverlo
2. Prevenire un deadlock
3. Evitare un deadlock

Abbiamo più strategie che comportano un overhead diverso e non c’è una migliore dell’altra. Ognuna ha pro e contro.

1 Strategia: do la possibilità ai processi di andare in deadlock. Il kernel analizza le varie risorse per verificare se esiste un deadlock, se esiste, queste strategie terminano 1 o più processi in stallo con lo scopo di liberare le risorse occupate ed allocarle ad altri processi.

Una contro indicazione di questo metodo è l’overhead che dipende dal numero di processi in stallo e il fatto che quando termine un processo in stallo poi lo devo riavviare e ciò è overhead. C’è da dire che questo algoritmo deve verificare periodicamente se c’è un deadlock.

Se applichiamo questo algoritmo all'esempio precedente dopo i primi due passi, l'algoritmo individua il deadlock e uno dei due verrà terminato per liberare il deadlock



2 Strategia: il kernel usa una strategia di allocazione in modo da assicurare che le 4 condizioni per cui si possa verificare un deadlock, non si verifichino contemporaneamente, se una delle 4 proprietà non sussiste allora non ci sarà nessun deadlock.
Una strategia semplice prevede che un dato processo nel momento in cui richiede una risorsa, le richiede tutte e subito in modo che il processo non si blocchi, si completa e rilascia le risorse per gli altri processi.

Contro: se un processo necessita di risorse in momenti diversi, è obbligato a tenere risorse dall'inizio senza usarle e blocca altri.

3 Strategia: il kernel analizza lo stato di allocazione delle risorse e verificare se concedendo le risorse a un processo, questo può portare un deadlock, se succede, a quel processo non verranno date risorse e così non ci sarà un deadlock in futuro. Se le risorse possono essere attribuite, le attribuisco subito se disponibili o blocco il processo in attesa delle risorse. Questo lo si fa con l'algoritmo del banchiere.

Contro: un dato processo può aspettare a lungo prima di vedersi concedere l'allocazione delle risorse.

STRATEGIA 1: INDIVIDUAZIONE E RISOLUZIONE DEI DEADLOCK

Un processo che è bloccato non è coinvolto in un deadlock insieme ad altri processi se la sua richiesta può essere soddisfatta da una sequenza di eventi del tipo:

completamento processo - rilascio risorsa - allocazione risorsa

Ciò significa che ho un processo bloccato ma non è in deadlock perché se ho un altro processo che detiene delle risorse richieste da questo processo bloccato, la terminazione di quel processo causa il rilascio delle risorse che possono essere riallocate al processo che era bloccato e quindi il processo bloccato di prima non è in deadlock.

Come si individua un deadlock correntemente?

Se il sistema è un sistema per cui ogni classe di risorsa contiene 1 sola istanza, vado a verificare se utilizzo la rappresentazione su grafi verifico se c'è un ciclo e ho individuato il deadlock. Nel caso in cui le classi avessero più istanze, non si usa il grafo di attesa, non conviene nemmeno usare il RRAG quindi si usa il modello a matrice. Questa rappresentazione si applica in generale.

Quando si usa la rappresentazione a matrice si prova a cercare una sequenza di eventi ammissibile dove tutti i processi bloccati possono sbloccarsi e ottenere le risorse, se una sequenza di questo tipo esiste non c'è deadlock, se non esiste invece c'è.

ESEMPIO: INDIVIDUAZIONE DEADLOCK

- Lo stato di allocazione di un sistema che contiene 10 unità di una classe di risorse R_1 e tre processi P_1 , P_2 e P_3 :

	R_1		R_1	
P_1	4		P_1	6
P_2	4		P_2	2
P_3	2		P_3	0
Allocated resources			Requested resources	
				Total resources
				R_1
				10
				Free resources
				0

- Il processo P_3 è nello stato *running*
 - Simuliamo il completamento di P_3
 - Allocchiamo le sue risorse a P_2
 - Tutti i processi in questo modo possono completare
 - Non esistono processi bloccati quando la simulazione termina
 - Quindi nessun deadlock

Quando si completa questa procedura, tutti i processi sono terminati e non ci sono processi bloccati.

Non c'è deadlock.

Un Algoritmo di Individuazione Deadlock

Inputs

$\rightarrow n$: Number of processes;
$\rightarrow r$: Number of resource classes;
$\rightarrow Blocked$: set of processes;
$\rightarrow Running$: set of processes;
$\rightarrow Free_resources$: array [1..r] of integer;
$\rightarrow Allocated_resources$: array [1..n, 1..r] of integer;
$\rightarrow Requested_resources$: array [1..n, 1..r] of integer;

Data structures

$\rightarrow Finished$: set of processes;
------------------------	---------------------

1. **repeat until** set *Running* is empty
 - a. Select a process P_i from set *Running*;
 - b. Delete P_i from set *Running* and add it to set *Finished*;
 - c. **for** $k = 1..r$

$$Free_resources[k] := Free_resources[k] + Allocated_resources[i,k];$$
- d. **while** set *Blocked* contains a process P_l such that
 - a. **for** $k = 1..r$, $Requested_resources[l,k] \leq Free_resources[k]$

$$Free_resources[k] := Free_resources[k] - Requested_resources[l,k];$$

$$Allocated_resources[l,k] := Allocated_resources[l,k] + Requested_resources[l,k];$$
 - b. Delete P_l from set *Blocked* and add it to set *Running*;
2. **if** set *Blocked* is not empty **then**
 - a. declare processes in set *Blocked* to be deadlocked.

Ripeto il ciclo finchè l'insieme *Running* è vuoto:
 prendo uno dei processi *running* P_i e una volta selezionato lo devo togliere da *running* e metterlo ai processi terminati. Nel momento in cui un processo termina, deve rilasciare le risorse; scandisco tutte le classi di risorse e per ogni classe di risorse libero le risorse che P_i aveva impegnato e le aggiungo alle risorse già libere.

Dopo averle liberate devo esaminare l'insieme dei processi bloccati.

Finchè esiste un processo bloccato P_I tale che le risorse che lui richiede sono inferiori a quelle disponibili gli devo attribuire le risorse: prendo il processo P_I e gli do tutte le risorse. Aggiorno la matrice di risorse allocate.

Elimino P_I dall'insieme dei processi bloccati, lo metto in running e il ciclo si ripete.

Quando non ci sono più processi running da scegliere si esce dal ciclo. Se nell'insieme bloccato c'è qualche processo allora sono in deadlock oppure non c'è nessun deadlock.

Esempio: Funzionamento Algoritmo Individuazione Deadlock

	R_1	R_2	R_3
P_1	2	1	0
P_2	1	3	1
P_3	0	1	1
P_4	1	2	2

Allocated resources

	R_1	R_2	R_3
P_1	2	1	3
P_2	1	4	0
P_3			
P_4	1	0	2

Requested resources

↑
Stato prima che P_3 faccia richiesta di 1 unità di R_1

	R_1	R_2	R_3
Total resources	5	7	5
Free resources			
	R_1	R_2	R_3
R_1 R_2 R_3	1	0	1

(a) Initial state

(b) After simulating allocation of resources to P_4 when process P_3 completes

	R_1	R_2	R_3
Allocated resources	2	1	0
Free resources	0	1	0
R_1 R_2 R_3	1	4	0
P_1 P_2 P_3 P_4	1	0	2

(c) After simulating allocation of resources to P_1 when process P_4 completes

	R_1	R_2	R_3
Allocated resources	4	2	3
Free resources	0	2	1
R_1 R_2 R_3	1	4	0
P_1 P_2 P_3 P_4	0	0	0

(d) After simulating allocation of resources to P_2 when process P_1 completes

	R_1	R_2	R_3
Allocated resources	0	0	0
Free resources	3	0	4
R_1 R_2 R_3	0	0	0
P_1 P_2 P_3 P_4	2	7	1

Abbiamo 4 processi con matrice di allocazione e risorse richieste, 3 classi di risorse e l'array di risorse libere.

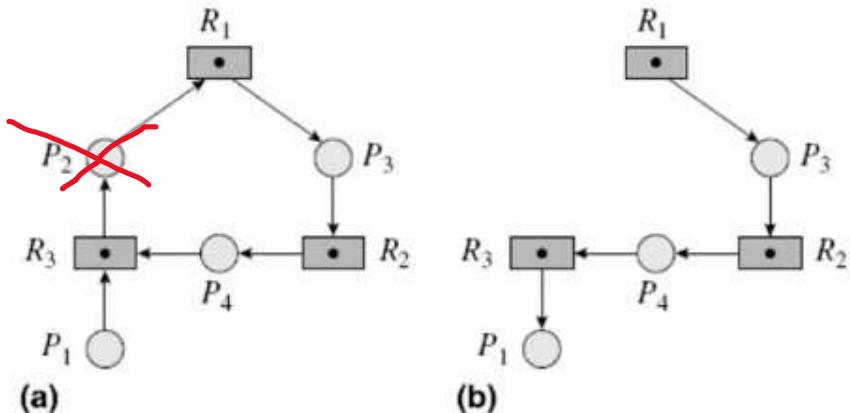
RISOLUZIONE DEADLOCK

Una volta individuato il deadlock, va risolto.

La risoluzione del deadlock per un insieme D di processi in deadlock consiste nello spezzare il deadlock per assicurare il progresso per alcuni processi in D:

- Si ottiene forzando la terminazione di uno o più processi in D
 - o Ogni processo terminato viene detto vittima
 - Le risorse della vittima sono allocate ad altri
 - o La scelta della vittima è fatta sulla base di criteri quali priorità processo, risorse consumate dal processo ecc...

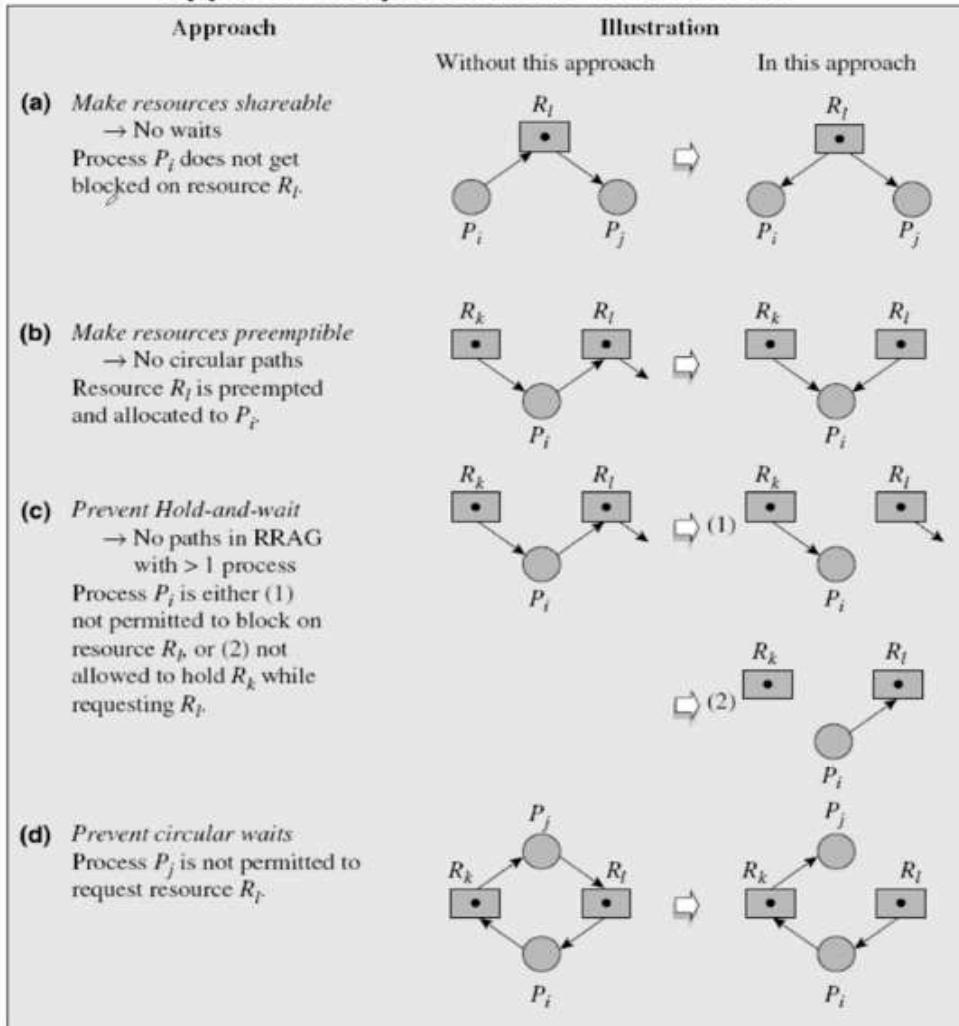
No
CICLO



Risoluzione deadlock. (a) un deadlock; (b) stato allocazione risorse dopo la risoluzione del deadlock

PREVENZIONE DEADLOCK

Approcci alla prevenzione del deadlock



La strategia di prima prevede che noi possiamo avere un deadlock, lo individuiamo e lo risolviamo. Con questo approccio non dobbiamo avere il deadlock prevenendolo.

Abbiamo detto che ci sono 4 condizioni che danno luogo al deadlock.

Questa strategia prevede di andare a risolvere singolarmente una di questa proprietà in maniera tale che se una non si verifica, non si possono verificare le altre.

In cosa consiste andare a prevenire. Spiegazione figura sopra.

- Proprietà 1: rendo la risorsa condivisibile. Gli archi diventano entrambi diretti. Non ci sarà più deadlock.

- Proprietà 2 (non prelazionabilità): rendiamo le risorse prelazionabili.
 - Proprietà 3 (possesso e attesa): posso intervenire in due modo per violarla:
 - o Faccio in modo che il processo P_i non si possa bloccare nella richiesta di una risorsa già impegnata
 - o Non consento a P_i fintantoché bloccato di mantenere la risorsa che manteneva R_k
 - Proprietà 4: non do la possibilità a un processo di richiedere una risorsa già impegnato: significa che devo introdurre un vincolo di validità alla richiesta.
-

PREVENZIONE DEADLOCK: ALLOCAZIONE SIMULTANEA

Possiamo operare sulle singole 4 proprietà di prima, basta che una non si verifichi e non ci può essere deadlock.

La strategia più semplice che si può implementare è quella di far sì che ciascun processo quando necessita di un numero di risorse vada a richiederle tutte simultaneamente in maniera tale che le dà tutte al processo e non potrà mai bloccarsi. Di fatto si prevengono le 4 proprietà.

Un processo bloccato non possiede alcuna risorsa. Questa è una strategia interessante per piccoli SO. Ha un solo difetto: pregiudica l'efficienza delle risorse.

PREVENZIONE DEADLOCK: RANKING DELLE RISORSE

Si può adottare un'altra strategia che previene le attese circolari. Ad ogni classe di risorsa si associa un rank(punteggio) di risorsa. Alla richiesta di una specifica risorsa, il kernel applica un vincolo di validità per decidere se questa richiesta deve essere soddisfatta o meno. Se il vincolo è TRUE concedo la risorsa al processo, se FALSE non gliela concedo e termino il processo.

Il vincolo è: il rank della risorsa che richiedo deve essere **maggior**e della risorsa con rank più elevato correntemente allocata al processo.

Funziona al meglio quando tutti i processi richiedono le rispettive risorse in ordine crescente di rank di risorsa: nel caso peggiore, la strategia può degenerare nella strategia di “allocazione simultanea” perché se c’è una richiesta potrei fare in modo che vado a richiedere una risorsa con rank minore per primo per poi richiedere quelle più elevate. Questo alla lunga degenera alla strategia vista in precedenza.

- Perché se attribuisco il rank alle varie risorse e sono un processo che possiede una risorsa, faccio richiesta a un’altra risorsa, il vincolo è vero se il rank della risorsa che richiedo è maggiore della risorsa che già possiede?

Supponiamo il processo P_i che possiede la risorsa R_k e a un certo punto richiede R_l . Se R_l ha rank > del rank R_k allora posso soddisfare la richiesta. Se R_l è subito disponibile gliela assegno oppure il processo si blocca in attesa che la risorsa sia disponibile. Supponiamo di avere P_i che ha R_l e richiede R_k . Sappiamo che rank $R_k <$ rank R_l , il vincolo è false, il processo viene terminato e l’attesa circolare non si verifica e R_l viene rilasciata e attribuita a P_i .

EVITARE I DEADLOCK

Abbiamo visto diverse modalità per evitare deadlock:

- La prima strategia consiste nell’individuare un deadlock in cui si è verificato ed applicar un metodo che risolve il deadlock.
- La seconda è quella della prevenzione. Queste strategie hanno come obiettivo di fare un’allocazione che non

consenta il verificarsi contemporaneo delle 4 proprietà del deadlock.

- L'ultima consiste nell'evitare i deadlock. Queste strategie vanno a soddisfare una richiesta di risorse solo se si stabilisce che dando le risorse non si verificherà un deadlock o immediato o futuro. Verificare che non occorra un deadlock nell'immediato è semplice (algoritmo in base al quale considero un insieme di processi con le loro richieste e vado a verificare se esiste un processo che le cui richieste di risorse sono inferiori alle risorse disponibili; così eseguo il processo che rilascia risorse allocabili ad altri processi. Se alla fine esaurisco tutti i processi attivi allora non ho deadlock, se mi rimane qualche processo attivo significa che c'è), il problema si pone per la verifica futura (il kernel non sa il comportamento futuro dei processi e non ha modo di stabilire se ci sarà deadlock futuro). La strategia adottabile per evitare deadlock è l'approccio conservativo: ogni processo è obbligato a dichiarare il numero max di risorse che può richiedere nella sua esecuzione, il kernel può consentire ai processi di richiedere le proprie risorse in fasi (non le dà tutte subito), sempre rispettando il vincolo delle risorse massime che ha dichiarato. In questo modo si può applicare un'analisi del caso peggiore (attraverso una simulazione si verifica se, concedendo risorse a un processo e rispettando i vincoli determinati dal max numero che il processo chiede, passo da uno stato sicuro a un successivo stato sicuro. È conservativo perché si fa in modo che il processo dichiari il numero max ma il processo può completare il suo ciclo senza richiedere effettivamente il numero massimo di risorse che ha dichiarato (il max è solo una stima, potrebbe essere anche minore).

L'algoritmo del banchiere si basa sull'ultimo tipo di strategia. Funziona all'analogia dei banchieri che devono stabilire se concedere mutui o meno. Il sistema valuta se la richiesta può essere soddisfatta attraverso una simulazione e questa è tesa a stabilire se al completamento si verifica un deadlock o no. Se definisco stato sicuro una situazione dove posso garantire concessione di risorse e non provocare deadlock, questo algoritmo garantisce che si può transire da uno stato sicuro all'altro.

NOTAZIONI DELL'ALGORITMO DEL BANCHIERE

Notation	Explanation
$Requested_resources_{j,k}$	Number of units of resource class R_k currently requested by process P_j
$Max_need_{j,k}$	Maximum number of units of resource class R_k that may be needed by process P_j
$Allocated_resources_{j,k}$	Number of units of resource class R_k allocated to process P_j
$Total_alloc_k$	Total number of allocated units of resource class R_k , i.e., $\sum_j Allocated_resources_{j,k}$
$Total_resources_k$	Total number of units of resource class R_k existing in the system

1. Richieste risorse j,k : k fa riferimento alle risorse della classe k richieste dal processo P_j
2. Numero massimo di risorse che P_j può chiedere alla classe k
3. Risorse allocate: risorse di classe k allocate a P_j
4. Totale risorse allocate della classe k
5. Totale delle risorse della classe k

Stato di allocazione sicuro: è uno stato di allocazione in cui è possibile costruire una sequenza di eventi **completamento processo, rilascio risorsa e allocazione risorsa** con cui ogni processo P_j nel sistema può ottenere $Max_need_{j,k}$ risorse per ogni classe di risorsa R_k e completare le proprie operazioni.

Se ho due classi di risorse disco e stampante e ho 2 e 2 risorse disponibili e il mio processo ne richiede 3, non è una richiesta fattibile. Siccome in un dato momento oltre alle risorse massime ci sono altri processi in ballo, si vede se il massimo numero di risorse meno quelle allocate è inferiore a quelle disponibile.

- Schema dell'approccio:
 1. Quando un processo fa una richiesta, si verifica se è una richiesta che può essere soddisfatta. Questa simulazione si chiama *proiezione dello stato di allocazione* (prima di concedere la risorsa)
 2. Se la proiezione è sicura, concede le risorse e aggiorna *Allocated_resources* e *Total_alloc*, altrimenti mantiene la richiesta pendente
 - a. La sicurezza è verificata con una simulazione
 - b. È assunto che un processo completa le sue operazioni solo se può prendere il massimo richeistro di ciascuna risorsa simultaneamente, ovvero, per tutti i k.
 3. Quando un processo rilascia una qualsiasi risorsa o termina le operazioni, esamina le richieste pendenti e alloca quelle che pongono il sistema in un nuovo stato sicuro
-

Esempio: algoritmo del Banchiere per una sola classe di risorse

Un sistema contiene 10 unità di risorse di una singola classe R_k

- Il requisito di risorse massime dei tre processi è 8, 7, 5 e l'allocazione corrente è 3, 1, 3
- P1 fa una richiesta di una risorsa, quindi in total_alloc ci saranno 8 risorse

P_1	8	P_1	3	P_1	1	Total alloc	7
P_2	7	P_2	1	P_2	0	Total resources	10
P_3	5	P_3	3	P_3	0		
Max need		Allocated resources		Requested resources			

- Consideriamo ora le seguenti richieste:

$$\begin{aligned} \text{Total_resource}(k) - \text{Total_alloc}(k) &\geq \\ \text{Max_need}(l,k) - \text{Allocated_resource}(l,k) \end{aligned}$$

In questo caso abbiamo un sistema che contiene 10 istanze di un'unica classe R. Abbiamo tre processi e le richieste massime di ciascun processo sono 8,7,5.

Dobbiamo tenere traccia delle risorse allocate ai tre processi: P1 3, P2 1 e P3 3.

L'altra cosa è tenere traccia del numero totale di risorse: 10 e delle risorse allocate totali correntemente: 7.

Supponiamo che P1 faccia una richiesta di risorsa quindi se il processo P1 fa questa richiesta uno deve verificare se questa richiesta può essere soddisfatta: siccome il numero totale di risorse allocate è 7 e ho ancora $10-7=3$ risorse disponibili, se concedo la risorsa a P1 ho che P1 avrà un'altra risorsa allocata (4 in totale) e il numero totale di risorse allocate è ora 8. Ho ancora $10-8=2$ risorse disponibili. Dobbiamo capire se questa risorsa può essere soddisfatta. Vediamo che P3 al più potrà richiedere ancora massimo 2 risorse che è compatibile con le 2 rimanenti.

Concedendo P1 potrei soddisfare anche P3 che al suo completamento rilascerebbe 5 risorse in modo da allocare le sue

risorse a P1 che potrebbe concludere le sue operazioni e quando P1 completa rilascia 8 risorse in modo che P2 possa completare le sue operazioni. Detto ciò, è possibile allocare 1 risorsa a P1. Supponiamo che abbiamo assegnato questa risorsa a P1. Ci troviamo in un nuovo stato:

P_1	8	P_1	2	4	P_1	x	0	Total alloc	✓	8
P_2	7	P_2	1		P_2	0		Total resources	✓	10
P_3	5	P_3	3		P_3	0				
Max need		Allocated resources			Requested resources					

Andiamo a considerare le seguenti richieste:

1. P1 fa una richiesta per 2 unità di risorsa. Dobbiamo verificare che il tot delle risorse disponibili – il totale delle risorse allocate è maggiore o uguale del massimo richiesto dal processo P1 – le risorse che gli sono attualmente allocate:

$$10 - 8 = 2 \text{ risorse. } \geq 8 - 4 = 4$$

$2 \geq 4$? No. La richiesta di P1 non sarebbe concessa.

Total_resource(k) – Total_alloc(k) \geq Max_need(l,k) – Allocated_resource(l,k) <- usiamo questa regola

2. P2 fa una richiesta per 2 unità di risorsa:

$2 \geq 6$? No. La richiesta P2 viene scartata.

3. P3 fa una richiesta per 2 unità di risorsa:

$2 \geq 2$? SI. La richiesta viene accettata.

Algoritmo del Banchiere

Inputs

n	:	Number of processes;
r	:	Number of resource classes;
$Blocked$:	set of processes;
$Running$:	set of processes;
$P_{requesting_process}$:	Process making the new resource request;
Max_need	:	array [1..n, 1..r] of integer;
$Allocated_resources$:	array [1..n, 1..r] of integer;
$Requested_resources$:	array [1..n, 1..r] of integer;
$Total_alloc$:	array [1..r] of integer;
$Total_resources$:	array [1..r] of integer;

Data structures

$Active$:	set of processes;
$feasible$:	boolean;
$New_request$:	array [1..r] of integer;
$Simulated_allocation$:	array [1..n, 1..r] of integer;
$Simulated_total_alloc$:	array [1..r] of integer;

1. $Active := Running \cup Blocked;$
for $k = 1..r$

$$New_request[k] := Requested_resources[requesting_process, k];$$
2. $Simulated_allocation := Allocated_resources;$
for $k = 1..r$ /* Compute projected allocation state */

$$Simulated_allocation[requesting_process, k] :=$$

$$Simulated_allocation[requesting_process, k] + New_request[k];$$

$$Simulated_total_alloc[k] := Total_alloc[k] + New_request[k];$$
3. $feasible := true;$
for $k = 1..r$ /* Check whether projected allocation state is feasible */
if $Total_resources[k] < Simulated_total_alloc[k]$ **then** $feasible := false;$
4. **if** $feasible = true$
then /* Check whether projected allocation state is a safe allocation state */
while set $Active$ contains a process P_l such that

$$\text{For all } k, Total_resources[k] - Simulated_total_alloc[k]$$

$$\geq Max_need[l, k] - Simulated_allocation[l, k]$$

$$\text{Delete } P_l \text{ from } Active;$$

for $k = 1..r$

$$Simulated_total_alloc[k] :=$$

$$Simulated_total_alloc[k] - Simulated_allocation[l, k];$$
5. **if** set $Active$ is empty
then /* Projected allocation state is a safe allocation state */
for $k = 1..r$ /* Delete the request from pending requests */

$$Requested_resources[requesting_process, k] := 0;$$

for $k = 1..r$ /* Grant the request */

$$Allocated_resources[requesting_process, k] :=$$

$$Allocated_resources[requesting_process, k] + New_request[k];$$

$$Total_alloc[k] := Total_alloc[k] + New_request[k];$$

COMPONENTI ALGORITMO:

Dobbiamo considerare le strutture dati e le variabili:

- N numero di processi
- R numero di classi di risorse
- Blocked → insieme dei processi
- Running → insieme dei processi
- P_requesting_process → processo che fa richiesta delle risorse
- MAX_need → Matrice n x r dove per ciascun processo si tiene traccia del numero max di risorse per ciascuna classe
- Allocated_resources → matrice che tiene traccia delle risorse allocate per ogni classe
- Requested_resources → matrice
- Total_alloc → vettore
- Total_resources

STRUTTURE DATI:

- Active → insieme dei processi
- Feasible → bool
- New_request → richieste del processo che va a fare
- Simulated_allocation
- Simulated_total_alloc → allocazione tot delle risorse simulato per ogni classe di risorse.

SVOLGIMENTO ALGORITMO:

- I processi attivi sono quelli in stato running e stato blocked e quindi ne faccio l'unione.
- Vado a raccogliere le nuove richieste che un processo può fare per ciascuna classe di risorse e le metto in

`New_request[k]` dove k va da $1 \dots r$, determino questo valore dalla matrice `Requested_resources`.

PASSO 2:

- Predisposizione della simulazione: si verifica se la richiesta che è arrivata è una richiesta fattibile verificando il totale delle risorse disponibile per ogni classe rispetto a quelle allocate correntemente. Siccome si deve simulare, si lavora sull'array `simulated_allocation`. Per ogni classe di risorse, vado a verificare se quella risorsa risulta fattibile o no cioè vado anzitutto ad aggiornare l'allocazione delle risorse: all'interno della matrice `simulated_allocation` vado a mettere per il processo che fa la richiesta e per ogni classe di risorse, l'allocazione corrente del processo + la nuova richiesta, questo lo devo fare per ogni classe di risorse. Una volta aggiornato `simulated_allocation`, dobbiamo aggiornare il vettore che tiene traccia delle risorse allocate effettivamente per ciascuna classe.

`Simulated_total_allocated` è uguale al totale allocato + la nuova richiesta che è stata fatta.

PASSO 3:

- Controlliamo se la nostra proiezione è fattibile oppure no e parte con `feasible = true`. Cosa determina se è fattibile o meno? se il totale disponibile di risorse per ogni classe è minore delle risorse simulate, allora quella richiesta non è fattibile (se concede la risorsa si allocherebbe un numero maggiore di risorse rispetto a quelle disponibili).

PASSO 4: fondamentale

- Controlla se è possibile costruire una sequenza completamento-rilascio-allocazione per ogni processo all'interno del sistema. Supponiamo che *feasible* = *true*. Se la proiezione è sicura bisogna verificare se lo stato proiettato è uno stato sicuro. Finchè c'è l'insieme **attivo** (running e blocked) contiene un processo P_l tale che per tutte le classi di risorse devo verificare che il totale di risorse di quella classe – le risorse allocate già di quella classe sia maggiore o uguale del massimo che richiede il processo per quella classe – quelle che gli sono state già allocate. Se esiste un tale processo allora posso selezionarlo, eliminarlo dall'insieme degli attivi e aggiorno gli array simulati.

PASSO 5:

- Quando esco dal while devo verificare se l'insieme Active è vuoto o no. Se è vuoto significa che concedendo la richiesta fattibile riesco a trovare una sequenza di passi << completamento, rilascio, alloca >> che mi porta al completamento di tutti i processi all'interno dei processi attivi. Se active non è vuoto significa che non sono in grado di garantire che in futuro qualche processo non andrà in deadlock, la richiesta non viene soddisfatta.

Esempio: algoritmo del Banchiere per più classi di risorse

$$\begin{aligned} \text{Total_resource}(k) - \text{Total_alloc}(k) &\geq \\ \text{Max_need}(l,k) - \text{Allocated_resource}(l,k) \end{aligned}$$

(a) State after Step 1

	R_1	R_2	R_3	R_4		R_1	R_2	R_3	R_4		R_1	R_2	R_3	R_4
P_1	2	1	2	1		1	1	1	1		0	0	0	0
P_2	2	4	3	2		2	0	1	0		0	1	1	0
P_3	5	4	2	2		2	0	2	2		0	0	0	0
P_4	0	3	4	1		0	2	1	1		0	0	0	0
	Max need					Allocated resources					Requested resources			
											Total alloc	5	3	5
											Total exist	6	4	8
											Active	{ P_1, P_2, P_3, P_4 }		

Abbiamo un sistema in cui $n=4$ processi. 4 classi di risorse.

Dei 4 processi in questione abbiamo la matrice del massimo delle risorse che un processo può chiedere.

Abbiamo poi le risorse allocate e le risorse richieste.

Dobbiamo supporre che concediamo a P_2 le sue richieste e se gliele concedo supponiamo che nella matrice simulata gli sto aggiungendo 1 e 1.

	R_1	R_2	R_3	R_4
P_1	1	1	1	1
P_2	2	4	2	0
P_3	2	0	2	2
P_4	0	2	1	1
	Allocated resources			

Total alloc	R_1	R_2	R_3	R_4
	5	4	6	4

Il totale simulato allocato sarà.

Vediamo se il totale è maggiore di ciascun'allocazione simulata per sapere se è ammissibile. Dopo dei confronti vediamo che la richiesta è ammissibile ($\text{Total alloc(simulated)} \leq \text{Total exist}$ e *feasible = true*).

PASSO SUCCESSIVO: (nel libro correggere dopo in prima)

(b) State before while loop of Step 4

P_1	2 1 2 1	P_1	1 1 1 1	P_1	0 0 0 0	Simulated total_alloc	5 4 6 4
P_2	2 4 3 2	P_2	2 1 2 0	P_2	0 1 1 0	Total exist	6 4 8 5
P_3	5 4 2 2	P_3	2 0 2 2	P_3	0 0 0 0		
P_4	0 3 4 1	P_4	0 2 1 1	P_4	0 0 0 0		
	Max need		Simulated allocation		Requested resources	Active	$\{P_1, P_2, P_3, P_4\}$

Visto che è ammissibile vediamo se riusciamo a completare la sequenza completa processo, rilascia risorse, alloca risorse in modo da completare tutti i processi senza deadlock.

Prima dell'esecuzione del while vediamo l'esecuzione nella figura in alto.

Eseguiamo il ciclo while che porta a completamento il processo P_1 . Per verificare ciò basta soddisfare la proprietà.

$$\text{Tot_risorse} - \text{Tot_allocato_simulato} = 1 \ 0 \ 2 \ 1$$

$$\text{Max_need} - \text{Allocated_resources} = 1 \ 0 \ 1 \ 0$$

1	0	2	1
---	---	---	---

\geq

1	0	1	0
---	---	---	---

SI. P_1 va al completamento.

STATO DOPO AVER SIMULATO IL COMPLETAMENTO DI P_1

(c) State after simulating completion of Process P_1

P_1	2 1 2 1	P_1	1 1 1 1	P_1	0 0 0 0	Simulated total_alloc	5 4 5 4
P_2	2 4 3 2	P_2	2 1 2 0	P_2	0 1 1 0	Total exist	4 3 5 3
P_3	5 4 2 2	P_3	2 0 2 2	P_3	0 0 0 0		
P_4	0 3 4 1	P_4	0 2 1 1	P_4	0 0 0 0		
	Max need		Simulated allocation		Requested resources	Active	$\{P_2, P_3, P_4\}$

P_1 viene rimosso dall'insieme dei processi attivi e va a completamento. P_1 rilascia le risorse che gli sono state allocate 1 1 1 1.

Vediamo per P4: facciamo la differenza tra total_exist e simulated total alloc, avremo $\boxed{2 \ 3 \ 1 \ 2}$

Facciamo il confronto max_need ecc e il risultato è $\boxed{0 \ 1 \ 3 \ 0}$

P4 soddisfa le condizioni e viene completato e rilascia le sue risorse allocate $0 \ 2 \ 1 \ 1$.

(d) State after simulating completion of Process P_4

P_1	2	1	2	1
P_2	2	4	3	2
P_3	5	4	2	2
P_4	0	3	4	1

Max
need

P_1	1	1	1	1
P_2	2	1	2	0
P_3	2	0	2	2
P_4	0	2	1	1

Simulated
allocation

P_1	0	0	0	0
P_2	0	1	1	0
P_3	0	0	0	0
P_4	0	0	0	0

Requested
resources

Simulated total_alloc $\boxed{4 \ 1 \ 4 \ 2}$

Total exist $\boxed{6 \ 4 \ 8 \ 5}$

Active $\{P_2, P_3\}$

P4 non è più nei processi attivi.

P2 soddisfa la proprietà e lo completiamo e rilascia $2 \ 1 \ 2 \ 0$.

(e) State after simulating completion of Process P_2

P_1	2	1	2	1
P_2	2	4	3	2
P_3	5	4	2	2
P_4	0	3	4	1

Max
need

P_1	1	1	1	1
P_2	2	1	2	0
P_3	2	0	2	2
P_4	0	2	1	1

Simulated
allocation

P_1	0	0	0	0
P_2	0	1	1	0
P_3	0	0	0	0
P_4	0	0	0	0

Requested
resources

Simulated total_alloc $\boxed{2 \ 0 \ 2 \ 2}$

Total exist $\boxed{6 \ 4 \ 8 \ 5}$

Active $\{P_3\}$

Rimane solo P3 che soddisfa le proprietà e va al completamento.

Effettivamente ora active è vuoto e a questo punto vengono effettivamente allocate le risorse al processo che ha fatto la richiesta (P2).

Complessità dell'algoritmo: $n^2 \times r$.

Algoritmo del Banchiere con allocazioni parametriche

- Supponiamo di avere 5 processi, P_0, P_1, P_2, P_3 , e P_4 e 4 classi di risorse A, B, C e D, nella seguente configurazione

Processi/Risorse	A	B	C	D
P_0	6	4	5	6
P_1	10	7	6	8
P_2	6	2	0	8
P_3	0	3	4	2
P_4	9	1	6	9

Max Risorse

Processi/Risorse	A	B	C	D
P_0	4	X-1	3	2
P_1	8	0	Y-2	2
P_2	4	0	0	0
P_3	0	0	3	2
P_4	2	1	Z+1	4

Risorse allocate

A	B	C	D
2	2	10	4

Risorse disponibili

- Determinare
 - gli intervalli di X, Y e Z per cui il sistema si trova in uno stato sicuro e l'eventuale sequenza sicura
 - Se la richiesta di (2, 0, 0, 2) risorse di P_2 può essere soddisfatta
- L'obiettivo è determinare se esiste una sequenza sicura in funzione di X, Y e Z
- Se esaminiamo la condizione

$$\text{Risorse_disponibili}_k \geq \text{Max_risorse}_{l,k} - \text{Risorse_Allocate}_{l,k}$$

l'unico processo che può essere soddisfatto è P_0 ed in particolare

$$\begin{cases} 4 - (X - 1) \leq 2 \\ 4 - (X - 1) \geq 0 \end{cases} \quad \text{da cui si ricava} \quad 3 \leq X \leq 5,$$

Dopo che P_0 termina l'esecuzione rilascia le sue risorse e le risorse disponibili diventano

$$[6, X + 1, 13, 6]$$

- A questo punto può essere soddisfatta solo la richiesta di P_3 . Le risorse disponibili al termine di P_3 sono:

$$[6, X + 1, 16, 8]$$

- Ora si può soddisfare solo la richiesta di P_2 . Al termine di P_2 le risorse disponibili diventano

$$[10, X + 1, 16, 8]$$

- Quindi, si può soddisfare la richiesta di P_4 se

$$\begin{cases} 6 - (Z + 1) \leq 16 \\ 6 - (Z + 1) \geq 0 \end{cases} \quad \text{da cui si ricava} \quad -11 \leq Z \leq 5,$$

- Infine, le richieste di P_1 possono essere soddisfatte se

$$\begin{cases} 6 - (Y - 2) \leq Z + 17 \\ 6 - (Y - 2) \geq 0 \end{cases} \quad \text{da cui si ricava} \quad -14 \leq Y \leq 8,$$

- In definitiva, la sequenza sicura è

P_0, P_3, P_2, P_4, P_1 con $3 \leq X \leq 5, -11 \leq Z \leq 5, -14 \leq Y \leq 8$

P2 non può essere soddisfatta

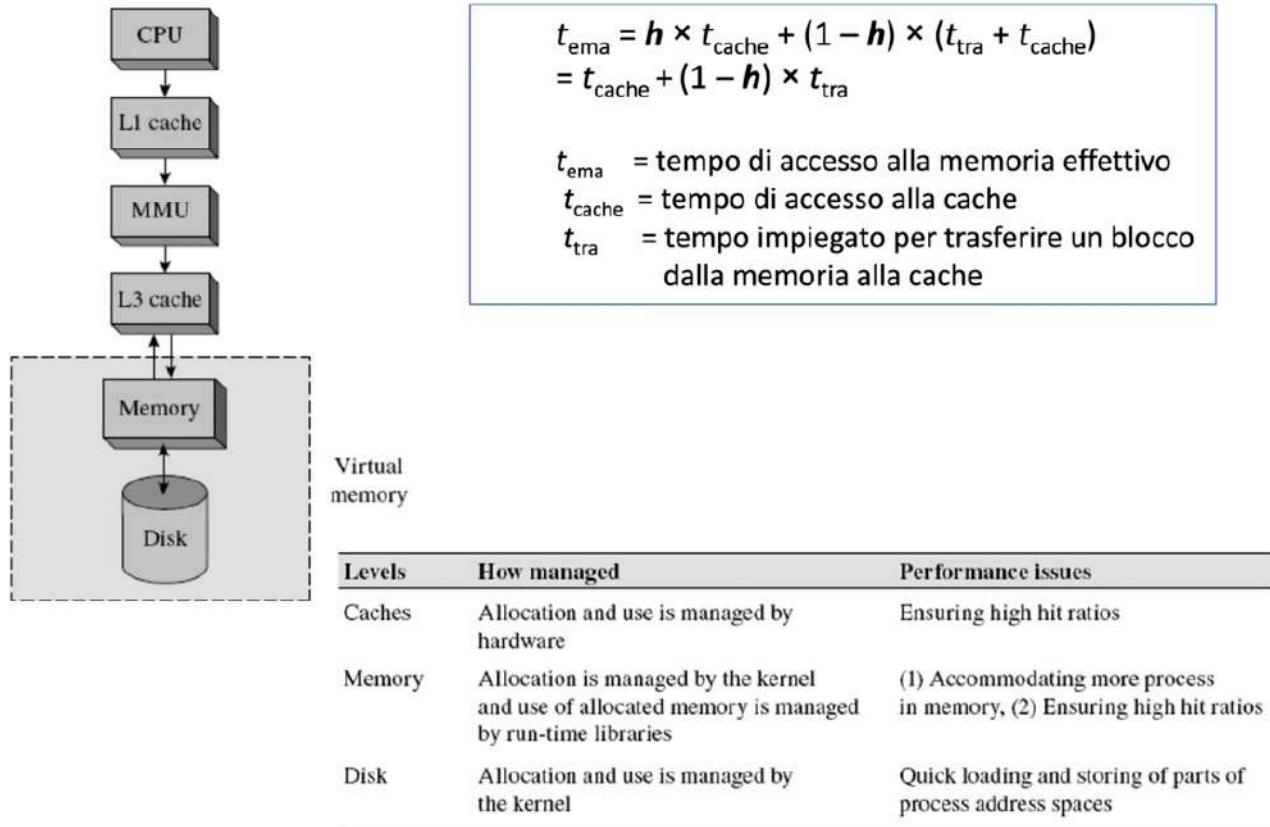
Lezione 10/05

GESTIONE DELLA MEMORIA

INTRODUZIONE

- Gerarchia della memoria
- Allocazione statica e dinamica della memoria
- Esecuzione dei programmi
- Gestione dello Heap

Gestione della Gerarchia di Memoria



La memoria è organizzata come una gerarchia perché l'obiettivo di un SO è duplice:

- Ottimizzazione delle prestazioni del sistema (numero di processi simultaneamente eseguiti). Quando è necessario aumentare prestazioni conviene usare sistemi multiprogrammati, questo comporta situazioni che coinvolgono la CPU che si dedica all'esecuzione di processi ma anche la memoria. Abbiamo visto che la CPU quando opera con indirizzi di memoria, opera con indirizzi che fanno riferimento sono i registri o la memoria centrale. Gli indirizzi di un disco non sono accessibili direttamente. Quando la CPU opera su istruzioni che si trovano nei registri, l'accesso è molto veloce mentre l'accesso alla memoria centrale è più lento perché avviene prima attraverso l'accesso a un BUS e poi si accede alla memoria. Ci si può trovare in situazioni

dove la CPU si ritrova inattiva e non può completare un'operazione.

- Per ottimizzare l'uso della CPU e della memoria la soluzione è stata quella di introdurre una memoria intermedia che si chiama "cache". La cache va a colmare questo gap dal punto di vista della velocità di accesso della CPU agli operandi in memoria centrale. La cache può essere di vari livelli: L1 si trova sul chip della CPU, L2 si trova al di fuori, più grande e lenta.
- Memoria centrale
- Memoria secondaria: dischi molto lenti e molto capienti. Memoria permanente e non volatile al contrario delle altre.

La gerarchia ottimizza l'esecuzione dei programmi e lo sfruttamento della memoria secondaria può essere utile per ottimizzare il numero di processi che simultaneamente devono trovarsi in memoria per aumentare il grado e le prestazioni del nostro sistema.

A questo proposito può accadere che se ho un processo nella memoria centrale che per un motivo o è bloccato a lungo o ha una bassa priorità rispetto ad altri processi, potrebbe essere sottoposto a swap out. Questa gestione è finalizzata a trovare il giusto mix tra processi I/O e CPU bound per ottimizzare le prestazioni. La gestione della memoria prevede un supporto hardware. La cache viene usata per cercare di minimizzare gli accessi in memoria centrale e senza supporto HW non esisterebbe.

La memoria e il disco fanno parte della memoria virtuale. Tutti i sistemi moderni implementano una memoria virtuale.

In che modo è possibile allocare spazio di memoria a un processo?

I primi sistemi erano uni programmati (1 solo processo per volta) e bastava caricare un processo nella sezione di memoria uno alla volta. Quando si è passato ai sistemi multiprogrammati questa gestione non andava più bene, come specifico a quale porzione di memoria ogni processo deve essere memorizzato?

Si poteva prevedere di attribuire spazi predefiniti a ciascun processo e fare in modo che fossero caricati in quelle posizioni. In questa maniera non è possibile avere una gestione ottimale dei processi dal punto di vista delle prestazioni. Nei sistemi moderni quando allochiamo spazio, questi possono essere spostati continuamente dalla memoria secondaria alla centrale. È necessario che un processo possa essere eseguito in una qualsiasi area di memoria. Lo spazio disponibile deve essere gestito opportunamente e gli indirizzi del processo devono fare riferimento a degli indirizzi che in quel momento sono astratti che richiedono un mapping dallo spazio logico a quelli fisici in un secondo momento perché dovremmo conoscere dall'inizio le locazioni di memoria in cui un processo verrà eseguito ma non ci è dato di saperlo.

BINDING DEGLI INDIRIZZI

Quello che avviene quando compiliamo un programma per creare un eseguibile, avvengono traduzioni degli indirizzi a seconda del modo in cui l'eseguibile viene generato. La maggior parte dei sistemi moderni consente ai vari processi di richiedere qualsiasi parte della memoria fisica anche se ipotizziamo che lo spazio del nostro calcolatore ha indirizzi che iniziano a 0000. L'indirizzo di partenza di ciascun processo non deve essere necessariamente 0000 se voglio che quel processo sia eseguito in qualsiasi parte della memoria. In che modo posso evitare questo vincolo e

introdurre questa flessibilità per far sì che un processo possa essere eseguito in qualsiasi parte della memoria?

Quello che avviene è il fatto che i programmi in fase di compilazione attraversano diverse fasi di traduzione perché lavorano in uno spazio di indirizzi detto “spazio degli indirizzi logici”: per poter ottenere questa cosa non possiamo fare direttamente riferimento alla memoria fisica, dobbiamo far riferimento a una memoria logica dove ciascun indirizzo viene usato come se fossero dei simboli (quando in un sorgente troviamo una variabile count, avrà un indirizzo in memoria fisico). Durante la traduzione dal codice sorgente al codice oggetto, andiamo di volta in volta a cambiare questi indirizzi perché teniamo conto del fatto che una porzione del processo può cambiare posizione in memoria e bisogna fare in modo che questi indirizzi siano rilocabili cioè che il loro indirizzo sia calcolato a partire da una posizione iniziale. Viene creato un eseguibile attraverso il linking e il loading e anche questi fanno questa associazione tra indirizzi logici fino ad arrivare a indirizzi fisici.

ASSOCIAZIONE (BINDING) DI ISTRUZIONI E DATI IN MEMORIA

L'associazione degli indirizzi da logici a fisici può avvenire in tempi diversi durante la generazione di un file eseguibile:

1. Compilazione: se sappiamo la posizione da cui deve partire un dato processo possiamo specificare dall'inizio un indirizzo assoluto di memoria, se la posizione di partenza cambia non possiamo specificarlo
2. Caricamento: si genera del codice rilocabile quando la posizione di memoria non è nota
3. Esecuzione: viene generato del codice rilocabile durante l'esecuzione di un programma. In questa fase a run time è necessario un supporto HW costituito da un unità MMU.

Detto ciò, il binding può essere di due tipi:

- a. Statico: avviene quando questa corrispondenza prima dell'esecuzione del programma
 - b. Dinamico: avviene durante l'esecuzione del programma.
-

ALLOCAZIONE STATICÀ E DINAMICA DELLA MEMORIA

L'allocazione della memoria ha a che fare con il binding, è un aspetto del binding degli indirizzi.

Allocare memoria significa destinare una porzione fisica di memoria di un dato processo.

- L'allocazione statica è eseguita dal compilatore, linker o loader essendo statica avviene prima che il programma venga eseguito. Comporta che le dimensioni delle strutture dati del nostro eseguibile devono essere note a priori e questo non è sempre fattibile.
 - L'allocazione dinamica è più flessibile, viene fatta a run time. Quando è necessario avere memoria è possibile allocarla. Chiaramente va da sé che nella gestione dinamica si introduce un overhead per gestire l'allocazione durante l'esecuzione
-

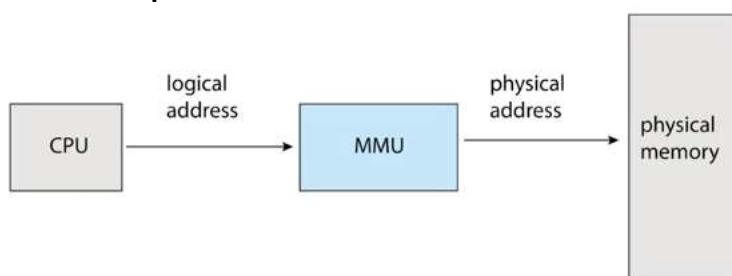
SPAZI DEGLI INDIRIZZI LOGICO E FISICO

Lo spazio di indirizzamento è diviso in parte logica e parte fisica. Siccome quando si scrivono programmi si vuole liberare il programmatore di conoscere la memoria fisica sottostante per cui la separazione la si fa proprio per questo motivo e per fare in modo di avere una tecnica in base alla quale i processi possono essere caricati in qualsiasi porzione della memoria e in qualsiasi momento.

- Indirizzo logico(o anche indirizzo virtuale): viene generato dalla CPU ed è l'insieme di tutti gli indirizzi logici generati da un programma. Tutti gli indirizzi necessari per memorizzare variabili, vettori, ecc... sono indirizzi logici (non hanno corrispondenza diretta con la memoria ma sono manipolati e alla fine vengono tradotti in indirizzi fisici dal supporto HW costituito dalla MMU).
 - Indirizzi fisici: insieme di tutti gli indirizzi fisici generati da un programma.
-

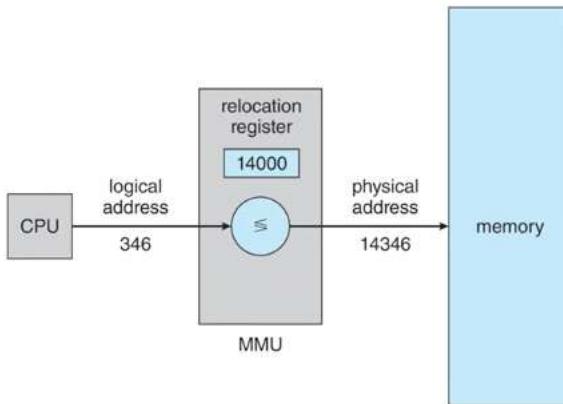
MEMORY-MANAGEMENT UNIT (MMU)

L'MMU prende indirizzi di tipo logico generati dalla CPU e con un meccanismo converte questi indirizzi logici in indirizzi fisici. La cosa importante è che se cambiano le condizioni per cui un processo viene rimosso dalla memoria, portato in memoria secondaria e riportato di nuovo in quella centrale, è possibile ricaricare il processo in una qualsiasi zona di memoria che fosse libera in quel momento.



L'MMU quando deve convertire un indirizzo logico in fisico usa i registri base che vengono usati per la protezione. L'MMU sfrutta questi registri che poi prendono nome di registro di rilocazione, si carica il valore del registro base (rilocazione) con un indirizzo che deve essere aggiunto a un indirizzo di tipo logico generato dalla CPU. La conversione è tale per cui alla fine l'indirizzo di memoria fisico viene generato dall'operazione che svolge l'MMU: prende

l'indirizzo logico, aggiunge il valore del registro base e crea l'indirizzo fisico che viene effettivamente usato.



ESECUZIONE DEI PROGRAMMI

Tutto ciò ha a che fare con l'esecuzione dei programmi.

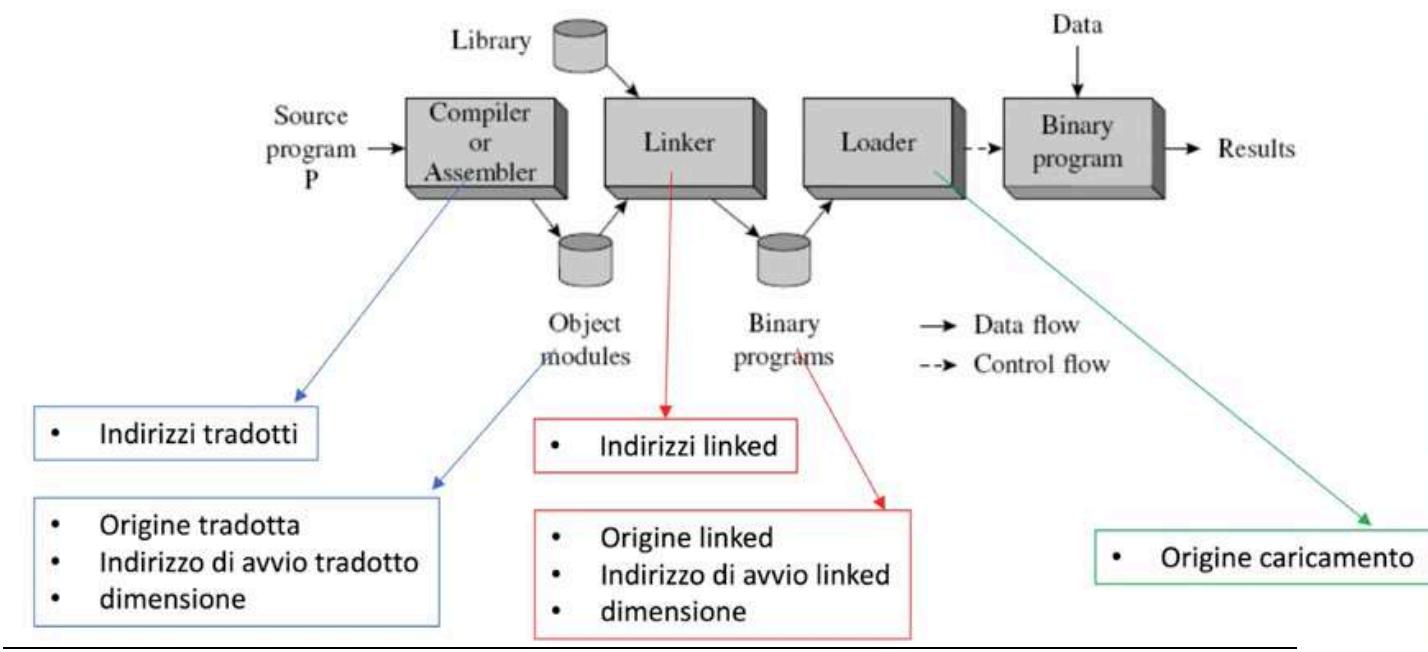
Gli indirizzi logici che vengono creati quando si compila un programma che aspetto hanno e come vengono generati?

Quando si va a creare un eseguibile, compiliamo il nostro programma.

Il compilatore prende le istruzioni e lo traduce in codice oggetto. Questo codice oggetto (binario) può essere specificato dall'utente o è un indirizzo di default e il compilatore assegna indirizzi alle istruzioni e a i dati di conseguenza, usa poi questi indirizzi come operandi nelle istruzioni del programma. L'indirizzo di avvio è l'indirizzo con cui parte il nostro eseguibile e gli indirizzi assegnati dal compilatore in questa fase sono chiamati "indirizzi tradotti". Le istruzioni nelle quali abbiamo operazioni, operandi e istruzioni sono generati dal compilatore e fanno riferimento a un punto di partenza, il compilatore associa istruzioni e dati del programma agli indirizzi tradotti. Dopo di che accade che il codice oggetto potrebbe richiamare funzioni libreria e queste istruzioni devono essere collegate al codice oggetto del programma.

Entrano in gioco il linker che non fa altro che includere l'indirizzo delle librerie all'interno del codice oggetto. Anche il linking può avvenire in modo statico e dinamico.

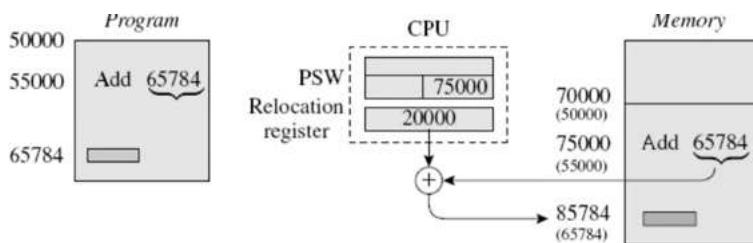
Dopo il linking entra in gioco il loader che è il componente che va a caricare l'eseguibile in una locazione di memoria specifica e questa locazione dipende se il linking è stato fatto in modo statico o dinamico, di conseguenza anche il loading avviene statico o dinamico, nel caso in cui avvenga in maniera dinamica c'è bisogno di una rilocazione.



RILOCAZIONE STATICÀ E DINAMICA

- Statica: avviene prima che un programma venga mandato in esecuzione. Consiste nel caricare il programma nella memoria dall'indirizzo specificato dalla porzione di memoria indicato dal kernel e a seconda se dopo il linking gli indirizzi corrispondono non avviene alcuna rilocazione, se sono diversi il loader riloca il programma che sta caricando.
- Dinamica: avviene durante l'esecuzione del processo. Può avvenire in due modi: se devo rilocare dinamicamente un processo il primo modo è quello di sospendere l'esecuzione di un processo, rilocarlo come se stessi rilocalo staticamente e riprendendo l'esecuzione; durante

l'esecuzione il sistema dovrebbe conoscere dei vari indirizzi prodotti dalla compilazione, linking e loading e chiaramente si produce overhead per sospenderli e riprenderli e questa gestione non è quella prediletta. Quella usata è un'allocazione dinamica dove si utilizza il contenuto del registro in modo che qualsiasi tipo di memoria usata dal programma viene sommata al contenuto del registro e viene individuato l'indirizzo effettivo dove caricare la memoria.



In questa figura è mostrato come funziona la rilocazione dinamica usando il registro di rilocazione: supponiamo di aver un programma che viene caricato a partire dalla locazione 50000 e supponiamo che stiamo eseguendo l'istruzione 55000 che è la ADD. Se il programma deve essere rilocalizzato viene caricato a partire dalla locazione 70000 il registro di rilocazione conterrà il valore 20000 e qualsiasi tipo di indirizzo a cui faceva riferimento il nostro programma deve essere sommato al registro di rilocazione, significa che nella memoria principale il nostro programma non è più caricato nella locazione 50000 ma nella 70000 e ciascun indirizzo di operando viene sommato al registro di rilocazione e posso tranquillamente far sì che durante l'esecuzione un programma sia spostato in memoria.

LOADING DINAMICO

Una cosa importante è che il caricamento del programma avviene in maniera dinamica nei sistemi moderni: si fa in modo che non

risieda tutto in memoria per poter essere caricato ed eseguito ma si può applicare un loading dinamico nel quale non si carica nella memoria centrale una routine che fa parte di una libreria finchè non viene invocata all'interno dell'eseguibile. Così ho un migliore spazio di memoria utilizzato. Se carichiamo tutto in memoria centrale è possibile che alcune routine non vengano mai invocate ma le abbiamo collegate in memoria comunque e abbiamo uno spreco di memoria. Il loading dinamico fa sì che si può caricare una data libreria solo quando è invocata. Tutte le routine sono tenute su disco in formato rilocabile. Non è richiesto un supporto speciale del SO.

LINKING DINAMICO

- **Linking statico:** le librerie di sistema e il codice del programma sono combinati dal loader nell'immagine del programma binario
- **Linking dinamico:** il linking è posticipato fino all'esecuzione.

Con il linking dinamico sono usate dei pezzi di codice detti **stub**, per andare a localizzare. Le routine di libreria residenti in memoria. Lo stub sostituisce sé stesso con l'indirizzo della routine ed esegue la routine. Il SO controlla se la routine si trova nell'indirizzo di memoria del processo, se non è là lo aggiunge allo stesso.

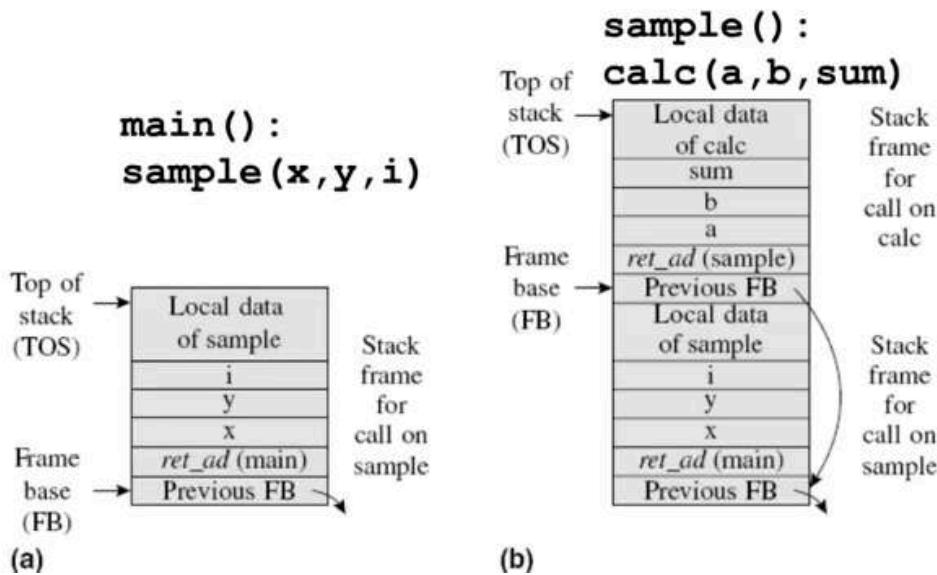
Questo sistema è noto anche come **librerie condivise (shared)**.

Nella gestione della memoria, oltre al kernel che si occupa di allocare memoria a un processo, la gestione riguarda anche il compilatore: quando abbiamo l'eseguibile in esecuzione, all'interno del codice si può avere necessità di memoria a run time, questo succede quando richiamiamo delle funzioni.

ALLOCAZIONE DI MEMORIA AD UN PROCESSO: STACK E HEAP

Quando allochiamo memoria a un processo, il supporto a run time alloca due tipi di memoria quando il programma viene eseguito: lo stack e l'heap.

Lo stack serve per memorizzare tutte le variabili automatiche del codice (variabili definite all'interno delle funzioni senza parole chiavi ulteriori es int i, int count, ...) e per gestire le chiamate a funzioni e gestire il contesto di una funzione e ritornare all'istruzione immediatamente successiva. Lo stack è un tipo di memoria LIFO (è una pila) e l'allocazione avviene quando si ha una chiamata di procedura e quando si esce dalla procedura si va a deallocare quella porzione di memoria.



Abbiamo una funzione `main` che invoca una funzione `sample`.

Abbiamo che le info relative alla chiamata della funzione `sample` prendono il nome di stack frame (sono memorizzati i dati di `sample`, gli argomenti e l'indirizzo di ritorno alla funzione `main` successiva all'invocazione di `sample`), ci sono puntatori per la gestione dello stack, il puntatore top e frame base che è un'entrata dello stack che contiene un altro puntatore che punta al precedente frame base che si trovava nello stack. L'allocazione e la deallocazione è molto veloce con lo stack perché ad esso viene allocato una quantità di memoria contigua e si gestisce

come una pila usando i puntatori che vengono allocati e deallocati. Nella figura b viene mostrata la configurazione dello stack quando nella funzione sample viene richiamata una seconda funzione. Si aggiunge uno stack frame relativo alla nuova funzione richiamata in sample.

Lo stack contiene due stack frame: quello della sample e quello della funzione richiamata.

HEAP

Quello che ci interessa di più però è l'heap. Lo stack è una gestione che coinvolge una porzione di memoria ma che viene gestita come un'area contigua. Cosa diversa è quando in un codice si invocano funzioni per l'allocazione dinamica della memoria. In quel caso si opera sull'heap. Quando si alloca spazio all'interno dell'heap è possibile farlo in qualsiasi posizione dell'heap. Qui la gestione è particolare.

Cosa succede quando allochiamo dinamicamente la memoria? In figura abbiamo due puntatori float e uno intero. Quando invochiamo allocazioni dinamiche, facendo la calloc di 5 float va ad allocare spazio per 5 elementi di tipo float. Ipotizzando che per 1 float servono 4 byte significa che andiamo a riservare per il primo puntatore 20 byte all'interno della memoria in una posizione qualsiasi. Lo stesso vale per il secondo puntatore che riserva 16 byte e con l'ultimo andiamo ad allocare 40 byte. Sulla sinistra della figura abbiamo il contenuto dell'heap a seguito dell'invocazione delle funzioni di allocazione dinamica. La strategia è la stessa che adotta il kernel. La figura b mostra l'heap dopo l'istruzione free puntatore 2, l'aria di memoria puntata da esso viene liberata. Per poter allocare opportunamente e deallocare queste aree i primi byte di ciascun area di memoria allocata contengono info relative alla lunghezza dell'area di memoria allocata.

```

float *floatptr1, *floatptr2;
int *intptr;
floatptr1 = (float *) calloc (5, sizeof (float));
floatptr2 = (float *) calloc (4, sizeof (float));
intptr = (int *) calloc (10, sizeof (int));
free (floatptr2);

```

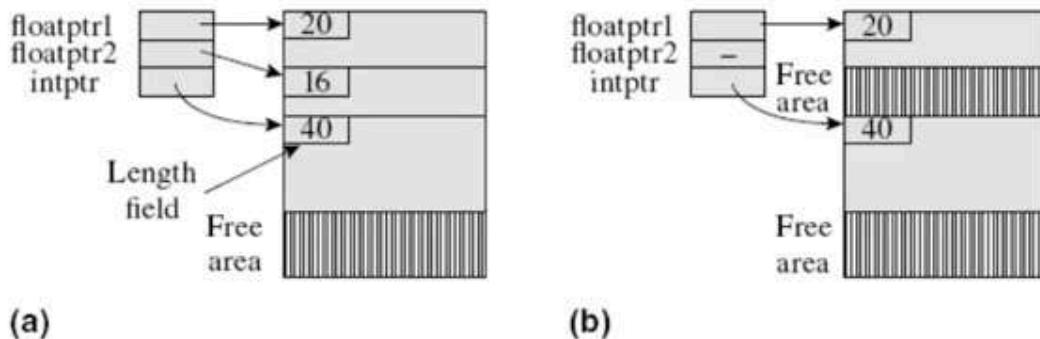
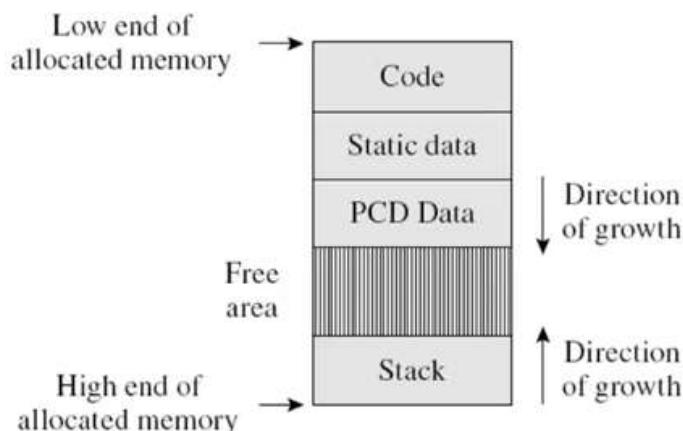


Figure 11.8 (a) A heap; (b) A “hole” in the allocation when memory is deallocated.

ALLOCAZIONE DELLA MEMORIA AD UN PROCESSO: MODELLO DI ALLOCAZIONE DELLA MEMORIA

A seguito di queste considerazioni, cioè che ciascun eseguibile mandato in esecuzione il kernel destina una quantità di memoria per il processo, questo viene caricato con eventuali rilocazioni dinamiche, com’è fatta l’immagine?

L’immagine del processo è organizzato come in figura:



In pratica nella memoria viene allocato spazio per le diverse componenti del programma. In particolare abbiamo la porzione destinata al codice, una porzione destinata alla memorizzazione

dei dati statici (quando si dichiara una var int ecc...) e c'è il problema della gestione dello stack e dell'heap. Entrambi dovrebbero avere una dimensione predefinita ogni volta che il processo va in esecuzione e deve eseguire il nostro codice. Come facciamo a sapere la dimensione giusta per questi? Durante l'esecuzione posso avere un certo numero di chiamate ma risulta difficile allocare precisamente lo spazio dello stack, il kernel non sa quanto spazio allocare precisamente, si può supporre quale siano quelle massime ma è rischioso. Quello che si fa è che stack e heap condividono una grossa area di memoria e crescono in direzione opposte: lo stack cresce verso l'alto e l'heap verso il basso. La dimensione è tale per cui sono sufficienti a gestire l'heap del nostro processo.

ALLOCAZIONE DELLA MEMORIA AD UN PROCESSO: PROTEZIONE DELLA MEMORIA

Per garantire la protezione si utilizzano i registri **base** e **size** per ciascun processo, in modo che sappiamo qual è l'indirizzo logico di partenza e la sua dimensione. Una volta che abbiamo queste info, nel momento in cui ci sono indirizzi generati dalla CPU si verifica che questo non fuoriesca dal limite massimo dedicato al processo. Questo limite lo si conosce perché avendo le due info non devo andare oltre la dimensione + l'indirizzo base.

GESTIONE DELL'HEAP

Vediamo in che modo viene gestito l'heap dal supporto a runtime che però sono operazioni che può svolgere anche il kernel per gestire le locazioni esterne al processo quando deve decidere lo spazio da destinare a un processo quando viene creato. Tutto ciò ha a che fare con il riuso della memoria: una gestione dell'heap è una gestione che deve ottimizzare l'utilizzo della memoria. Significa che dobbiamo avere la possibilità di allocare

memoria quando si usa una variabile e nel momento in cui questo spazio viene inutilizzato deve essere gestito opportunamente in modo che possa essere utilizzato di nuovo successivamente.

Viene usata una lista chiamata “**Free list**” cioè lista dei blocchi liberi che possono essere di dimensioni variabili.

RIUSO DELLA MEMORIA

Table 11.2 Kernel Functions for Reuse of Memory

Function	Description
Maintain a free list	The <i>free list</i> contains information about each free memory area. When a process frees some memory, information about the freed memory is entered in the free list. When a process terminates, each memory area allocated to it is freed, and information about it is entered in the free list.
Select a memory area for allocation	When a new memory request is made, the kernel selects the most suitable memory area from which memory should be allocated to satisfy the request.
Merge free memory areas	Two or more adjoining free areas of memory can be merged to form a single larger free area. The areas being merged are removed from the free list and the newly formed larger free area is entered in it.

- La free list contiene tutte le info per ogni blocco di memoria libero all'interno dell'heap e se un processo fa richiesta di memoria l'allocatore va a consultare la lista e alloca il primo blocco libero che è sufficiente a soddisfare quella richiesta. Nel momento in cui c'è un blocco che si libera, deve essere nuovamente inserito nella free list.
- Quando allochiamo blocchi di memoria che sono poi liberati, una cosa importante per rendere efficiente il riutilizzo è fare

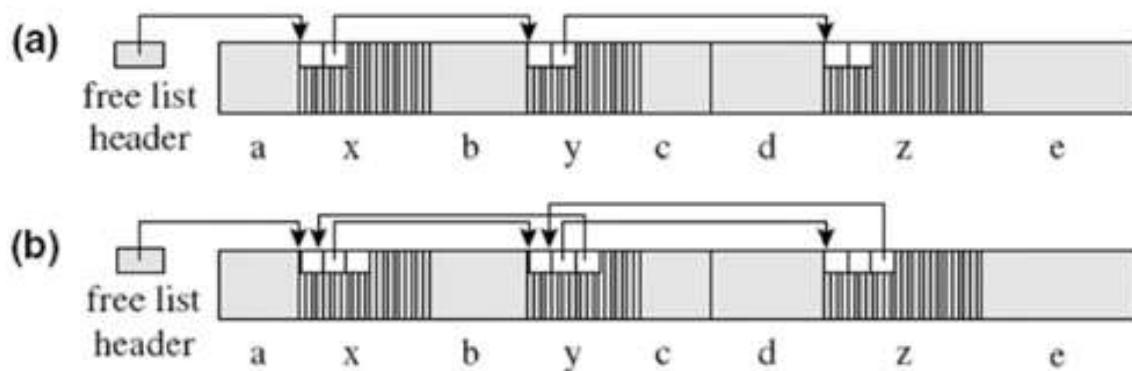
in modo di fondere blocchi di memoria adiacenti in blocchi più grandi per soddisfare maggiori richieste di memorie

GESTIONE DELLE FREE LIST

Vediamo come funziona la gestione basata sulla free list.

Per ogni area di memoria contenuta nella free list, il kernel o il supporto a run time della libreria, viene gestito la dimensione dell'area di memoria e i puntatori usati per formare la lista.

Il kernel memorizza questa info in pochi byte all'inizio della stessa area di memoria libera.



ESECUZIONE DI NUOVE ALLOCAZIONI MEDIANTE FREE LIST

Si usano tre tecniche:

1. First-fit → cerco la prima area di memoria sufficientemente grande a soddisfare la richiesta di n byte. Una volta individuato divido il blocco in due parti: la prima in esattamente n byte per soddisfare la richiesta e quello che avanza va nella free list. Quello che può succedere è una suddivisione delle prime aree della lista e col tempo le aree diminuiscono di dimensione perché ogni volta si spezza e diventando più piccole diventano insufficienti a soddisfare le richieste. Questo tipo di fenomeno viene detto "frammentazione esterna".

2. Best-fit → si evita di dividere aree di memorie grandi. Quello che fa questa strategia è cercare la più piccola area di memoria all'interno della free list di dimensione maggiore o uguale di n in modo che il blocco libero si avvicina alla richiesta di memoria. Tutto ciò che è in più viene spezzato e messo nella free list. Anche il best fit produce alla lunga aree molto più piccole anche della first fit e abbiamo una frammentazione esterna. In più, siccome si effettua un'operazione di ricerca, si introduce un overhead.
3. Next-fit → è come il first fit con una differenza: la ricerca del blocco libero parte da dove si era fermato in precedenza. Si tiene memoria del blocco precedente allocato nella lista e si parte dal blocco successivo e in questo modo si spezzetta in modo più uniforme i blocchi della lista al contrario del first fit che spezzettava solo i primi blocchi.

Quello che funziona meglio è First-fit e Next-fit.

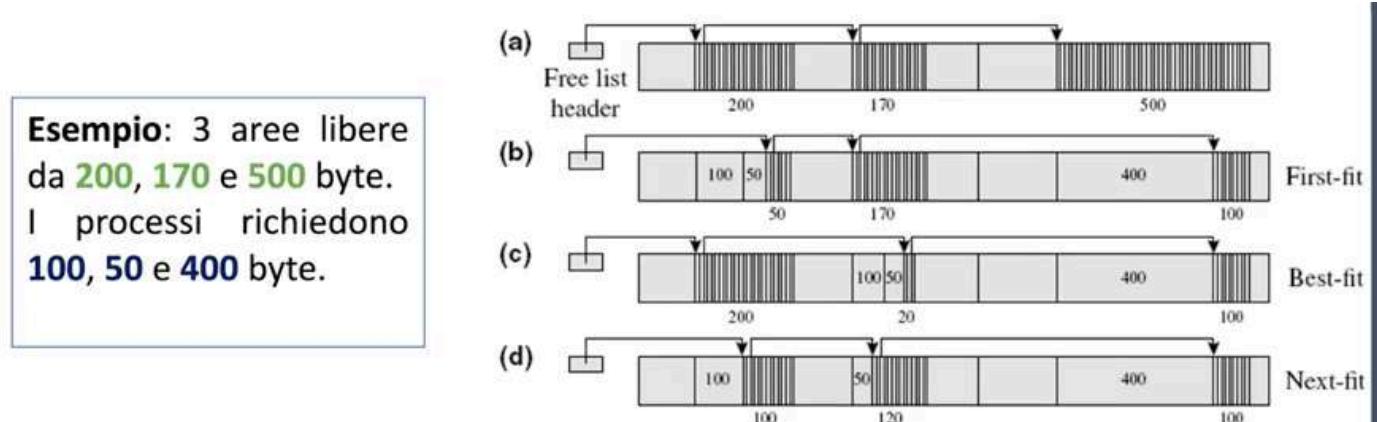


Figure 11.11 (a) Free list; (b)–(d) allocation using first-fit, best-fit and next-fit.

FRAMMENTAZIONE DELLA MEMORIA

Quando allochiamo con best, next, first si crea della frammentazione: spezzo ogni volta e quello che avanza va nella free list ma via via sono sempre più piccoli e faccio difficoltà ad allocarli a richieste successive. Questo viene detta frammentazione esterna.

C'è un'altra frammentazione chiamata **interna**: frammentazione che avviene quando vado a destinare alla richiesta di un processo una quantità maggiore rispetto a quella richiesta, quindi all'interno della quantità destinata viene usata una parte e il rimanente è inutilizzata (esempio: alloco 100 byte a un processo che ne richiede 50, gli altri 50 sono allocati ad esso ma non verranno mai usati).

La frammentazione deve essere minimizzata al meglio.

Come è possibile evitarla?

FUSIONE DI AREE DI MEMORIA LIBERE

È possibile evitare la frammentazione andando a fondere le aree di memorie libere, se nella free list creo blocchi liberi piccoli, li fondo per creare blocchi più grandi ed evito la frammentazione esterna.

Ci sono due modi che consistono in:

- **Boundary tag:** un tag è un descrittore di stato di un'area di memoria, consiste in una parte che ci dice se l'area di memoria è libera o meno e poi c'è un altro campo che ci dice quanto è grande quel blocco di memoria. Il boundary tag sono due tag identici posti all'inizio e alla fine di un blocco libero. In figura abbiamo l'area libera o allocata, la porzione di boundary tag che consiste nello stato di allocazione e la dimensione e abbiamo il puntatore alla free list, alla fine dello stesso blocco ho il b. tag.

- Compattazione della memoria.

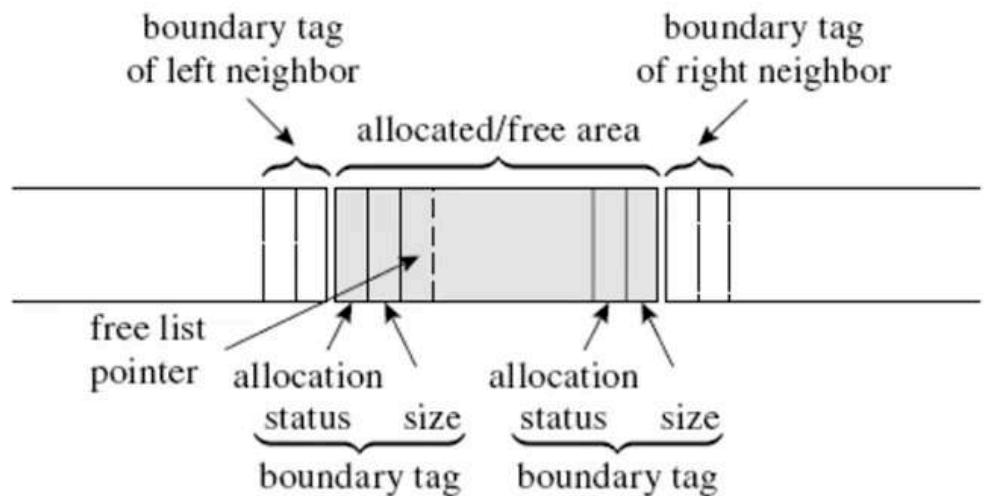


Figure 11.12 Boundary tags and the free list pointer.

- Un tag è un descrittore di stato di un'area di memoria
 - Quando un'area di memoria diviene libera, il kernel controlla i tag boundary delle sue aree vicine
 - Se un'area vicina è libera, è unita con l'area appena liberata

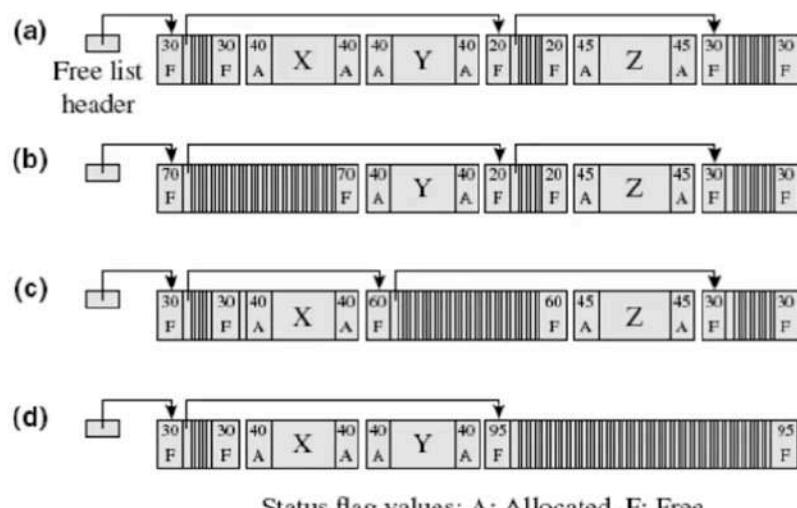


Figure 11.13 Merging using boundary tags: (a) free list; (b)–(d) freeing of areas X, Y, and Z, respectively.

A un certo punto, data la mia free list con le indicazione dei tag, che si libera un'area di memoria, si controlla il boundary tag della memoria che si sta liberando e dei vicini. Quando si libera un'area di memoria si va a verificare se un'area vicina è libera, se è così si

va unire il vicino con l'area appena liberata, si aggiornano i tag e i puntatori all'aria libera successiva.

Quando è eseguita l'unione, è rispettata la “regola del 50%” perché o si libera un'area con 2 vicini liberi, o con 1 vicino libero o con nessun vicino libero, a seconda di queste situazioni la lista cresce, decresce o rimane invariata.

Quando si libera un'area: il numero totale aumenta di 1 se ha 0 vicino, diminuisce di 1 se ha 2 vicini e rimane immutata se ha 1 vicino. Il numero di aree allocate è dato dal numero di aree di tipo A, B,C mentre quelle libere sono dato dal (doppio di A + il numero di B liberi)/2. Alla lunga avrò che $\#A = \#C$ quindi $m = n/2$



$$\text{Number of allocated areas, } n = \#A + \#B + \#C$$

$$\text{Number of free areas, } m = \frac{1}{2}(2 \times \#A + \#B)$$

$$\text{In the steady state } \#A = \#C. \text{ Hence } m = n/2$$

- L'altro metodo è quello della compattazione: prendo le aree di memoria, le compatto verso la fine della free list. Questo richiede una rilocazione.

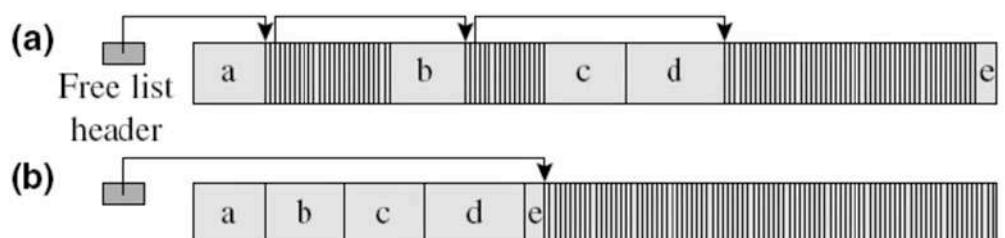


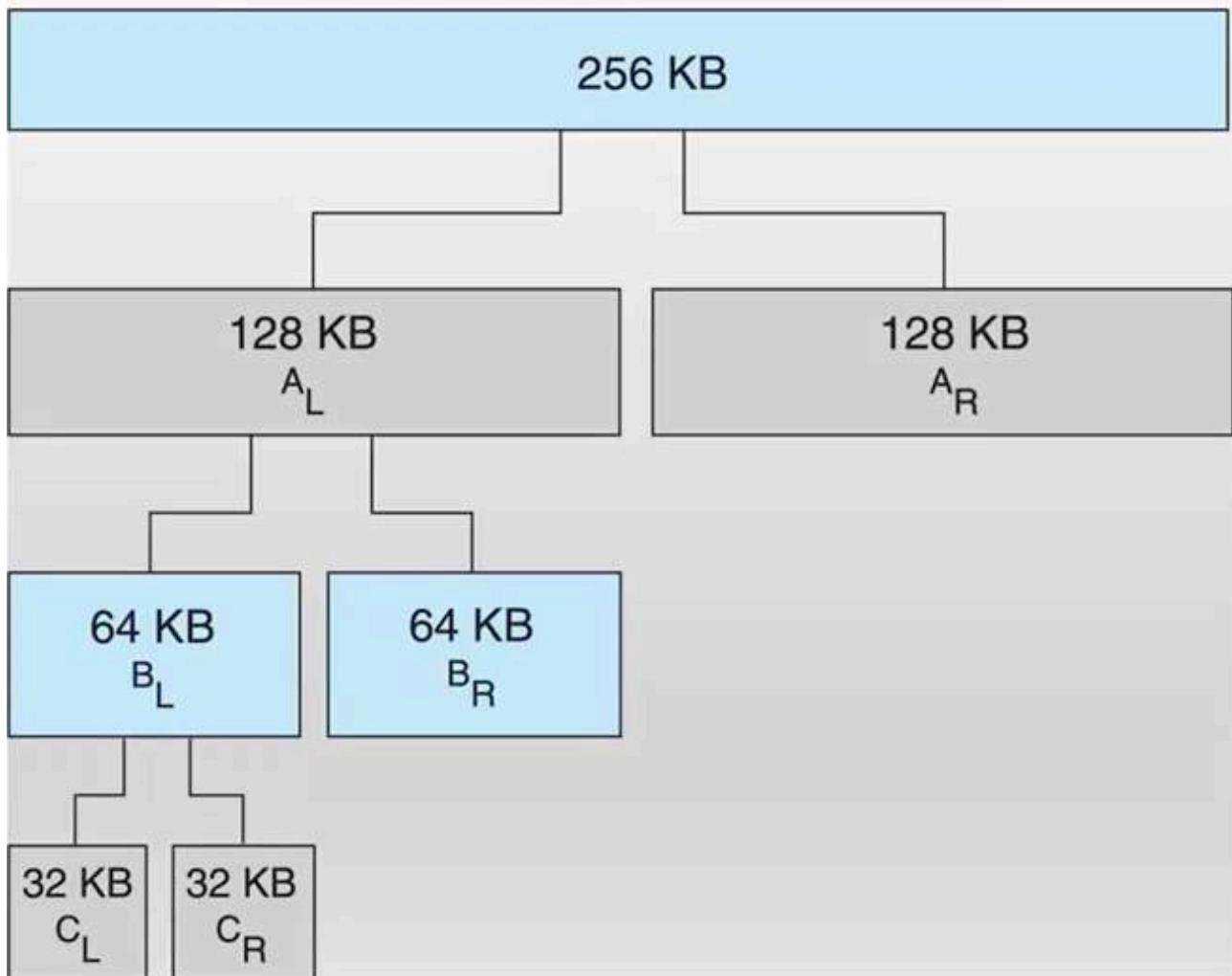
Figure 11.14 Memory compaction.

BUDDY SYSTEM E ALLOCATORI DELLE POTENZE DI 2

Questi allocatori attenuano la frammentazione soprattutto esterna ma comporta frammentazione interna.

BUDDY SYSTEM

Blocchi contigui di memoria



Il buddy system parte con un blocco di memoria, spezza la memoria in due parti creando una sorta di albero con i rispettivi pesi che vediamo in figura. Quando si spezzetta, i blocchi adiacenti spezzettati prendono il nome di buddy (amici). L'idea è che mantengo una lista di blocchi liberi di ciascun livello, quando arriva una richiesta, vedo qual è il blocco più piccolo in grado di soddisfare la richiesta, se non fosse libero devo andare al blocco di livello superiore e trovare quello libero disponibile. Assegno il blocco alla richiesta che potrebbe essere soddisfatta da un blocco molto più grande (frammentazione interna) e quando un blocco

si libera, il blocco e il suo buddy vengono fusi insieme creando un nuovo blocco di livello superiore.

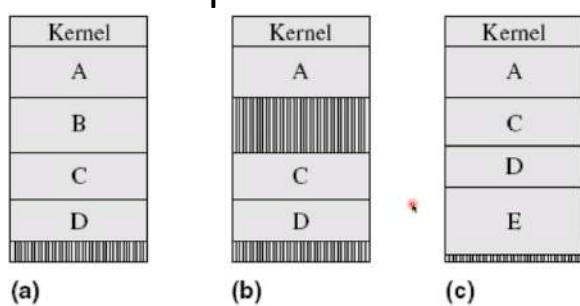
ALLOCATORE DI POTENZE DI 2

Anche qui va ad allocare blocchi di memoria di potenze di 2. La differenza è che non viene fatta nessuna fusione: quando un blocco si libera si mette nella lista e quando c'è una richiesta si alloca un blocco che soddisfa la richiesta. Quando viene rilasciato un blocco viene restituito nella free list corrispondente. Quando l'allocatore non ha più blocchi di una certa dimensione fissata, si crea un nuovo blocco.

Lezione 14/05/2021

ALLOCAZIONE CONTIGUA DELLA MEMORIA

Ad ogni processo il kernel alloca un unico blocco di memoria sufficientemente grande per contenere tutti i dati e le info relative a esso. Tutta la memoria allocata è contigua (quantità di byte sequenziali all'interno della memoria). È uno schema poco flessibile: il kernel non può stimare la dimensione effettiva del processo e quello che può accadere sono problemi di frammentazioni: se ho frammentazioni interne non si può fare nulla ma si possono adottare tecniche per minimizzare la frammentazione esterna che può verificarsi se alloco diversi blocchi ai vari processi; alla lunga la memoria disponibile sarà costituita da piccoli blocchi che non potranno soddisfare le richieste dei processi. Utilizzando le tecniche di riuso si rischia di ritardare qualche processo in attesa che si liberi un blocco per soddisfare il processo in questione.

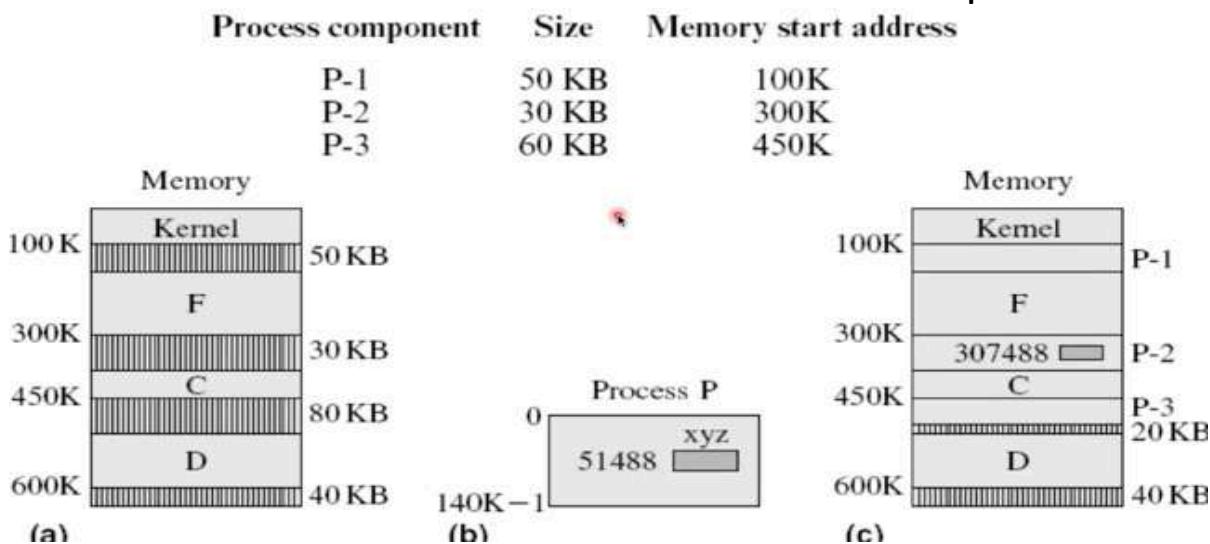


ALLOCAZIONE NON CONTIGUA DELLA MEMORIA

Un'allocazione non contigua è adottata da tutti i SO moderni. Si fa in modo che un dato processo si possa vedere allocato dei blocchi di memoria non contigui nella memoria principale ma sono sparsi ma che insieme hanno una dimensione tale da poter memorizzare l'intero spazio di indirizzamento di un processo. In uno schema di allocazione non contigua, si considera un processo come costituito da un insieme di componenti che possono essere memorizzate in diverse porzioni di memoria.

In questo modo si riduce la frammentazione esterna perché non essendo vincolato a usare blocchi contigui di memoria, si cercano blocchi con dimensioni adatte a contenere una componente del processo. può accadere però frammentazione interna.

- Osserviamo che abbiamo un processo P che è composto da 3 componenti di 50kb, 30kb, 60kb, supponiamo di avere informazioni sugli indirizzi di memoria di partenza di ciascun blocco che memorizza le componenti P1, P2, P3.
- Inizialmente abbiamo un blocco da 50kb, da 30, 80 e da 40. Se abbiamo uno schema non contiguo col processo P rappresentato con 140kb, si assegna il blocco da 50k a P1, quello da 30k a P2 e quello da 80 a P3. Quello che avviene lo vediamo nella figura c: c'è una flessibilità nella memorizzazione della memoria nei vari componenti.



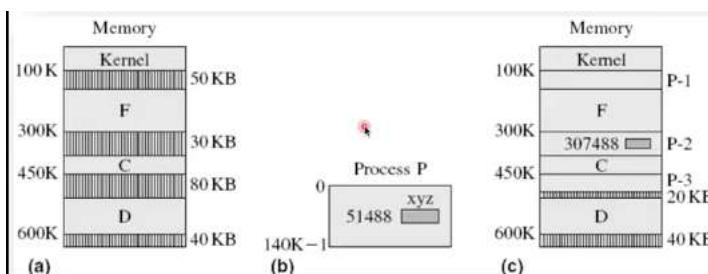
- Ci deve essere un meccanismo attraverso il quale si sale dagli indirizzi logici ai fisici.

INDIRIZZI LOGICI E FISICI, TRADUZIONE INDIRIZZI

Se posso allocare diverse componenti in diverse zone di memoria, c'è bisogno di avere meccanismi che consentono di far corrispondere gli indirizzi logici con gli indirizzi fisici. Questa procedura prende nome di **"Traduzione degli indirizzi"**

Per questo motivo abbiamo due spazi di indirizzamento diverso:

1. Logico: è l'indirizzo di un'istruzione o dato usato in un processo P
 - a. L'insieme degli indirizzi logici in P costituiscono il suo spazio di indirizzamento logico
2. Fisico: indirizzo di memoria dove è memorizzata un'istruzione o un dato
 - a. L'insieme di indirizzi fisici nel sistema costituiscono il suo spazio di indirizzamento fisico.

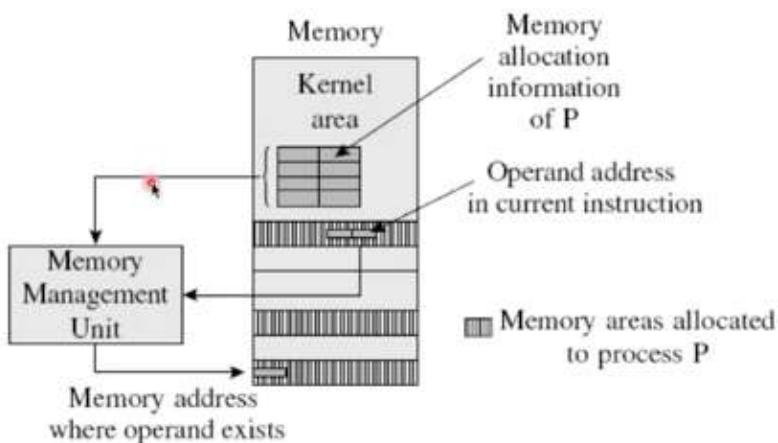


- P-1 ha dimensione 50KB \rightarrow 51200 byte
 - xyz si trova in P-2
 - #Byte di P-2: $51488 - 51200 = 288$
- P-2 è caricato a partire da 300KB (307200)
 - indirizzo fisico di xyz è 307488

Il blocco P1 è un blocco da 50kb \rightarrow 51200 byte.

$51488 > 51200$ quindi l'operando si troverà in P2. Quello che si fa è che lo scostamento del blocco è dato dall'indirizzo logico di partenza *meno* P1= $51488 - 51200 = 288$ byte. Significa che xyz che si deve trovare al byte 288simo a partire dal blocco di P2. P2 è caricato a partire da 300kb \rightarrow 307200, gli aggiungo i 288 e ottengo l'indirizzo fisico di xyz che è 307488.

Il kernel mantiene le info sulle aree di memoria allocate in una struttura dati che è una tabella disponibile alla MMU. Quando la CPU manda gli indirizzi logici alla MMU, quest'ultima sfrutta le info dei vari blocchi allocati alle componenti del processo P e sulla base di queste info e sulla base dell'indirizzo logico, si va a calcolare l'indirizzo fisico di quel dato o istruzione.



A schematic of address translation in noncontiguous memory allocation.

Nell'area kernel abbiamo la tabella con le info dei processi, le parti tratteggiate rappresentano le porzioni di memoria allocate a un processo P.

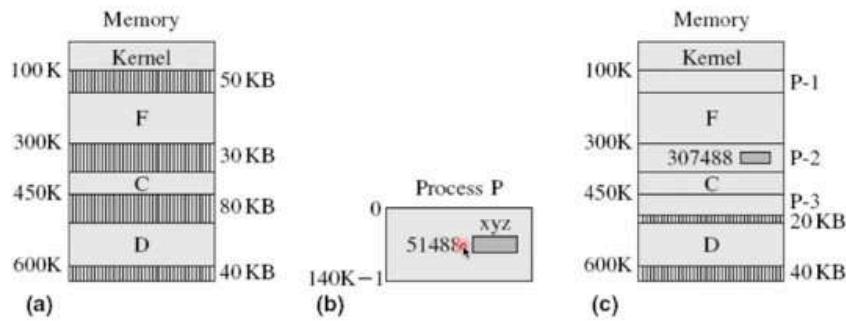
Vediamo come avviene questa traduzione.

Per capire la traduzione dobbiamo vedere com'è strutturato l'indirizzo logico. L'indirizzo logico è composto da due parti, la prima parte è l'ID componente di P e la seconda parte è lo scarto rispetto a quel blocco (numero di byte a partire da quel blocco). L'indirizzo logico è visto come una coppia ($comp_i, byte_i$) cioè componente dove si trova l'operando e istruzione e scarto del componente in cui si trova.

Facendo uso della tabella si calcola l'indirizzo fisico che è dato da: indirizzo di memoria fisico di partenza che contiene il componente del processo e il numero di byte di scarto.

Effective memory address of $(comp_i, byte_i)$
 = start address of memory area allocated to $comp_i$
 + byte number of $byte_i$ within $comp_i$

- Esempio: P fa riferimento all'area dati di xyz con l'indirizzo logico (P-2, 288)
 - La MMU calcola l'indirizzo fisico effettivo come $307.200 + 288 = 307.488$



APPROCCI ALL'ALLOCAZIONE NON CONTIGUA DELLA MEMORIA

Vediamo due approcci:

- **Paginazione:**

- I processi sono visti come costituiti da insieme di componenti di dimensioni fisse che prendono il nome di "pagine". Proprio per il fatto che ogni componente del processo è costituito da un numero fisso, significa che a ogni processo viene attribuito un numero di pagine per memorizzare le sue componenti. Questo significa che in questo modo non c'è frammentazione esterna perché ci sono porzioni della stessa dimensione da associare a un processo e sono tutte riempite completamente. Quello che si può avere è una frammentazione interna minimale: solo l'ultima pagina attribuita potrebbe essere più grande del necessario che non è completamente usata. È importante sapere che la dimensione delle pagine non è decisa a priori ma è dettata dalle caratteristiche HW (a seconda dell'architettura HW, avremo che la dimensione della

pagina è fissata. Tipicamente è una potenza di 2). Nel momento in cui un processo termina, le sue componenti liberano le pagine e possono essere riutilizzate per un uso successivo per altri processi.

- **Segmentazione**

- Si considera che il programmatore o il compilatore va a identificare tutte le componenti da un punto di vista logico di cui è composto un programma (composto da strutture dati, funzioni, procedure e sono viste come entità dette SEGMENTI). L'idea è quella di allocare porzioni di memoria di dimensioni tali da contenere ciascun segmento. In questo caso non c'è frammentazione interna perché attribuisco a ciascun segmento una porzione di memoria che lo contiene. Si può verificare però frammentazione esterna perché i segmenti hanno dimensioni variabili e per poter allocare memoria ai vari segmenti avrò bisogno di un blocco grande, poi piccolo ecc... e alla lunga rimangono blocchi che non contengono nessun segmento. Un vantaggio di questo approccio è che rende molto facile la condivisione di codice, dati, librerie tra vari processi.

Dato che entrambi i processi sono buoni, l'idea è quella di unirli per prendere il meglio da entrambi → **approccio ibrido**:

- **segmentazione con paginazione:** si mantiene la facilità di condivisione di codice e dati, evita la frammentazione esterna ma può rimanere un po' di paginazione interna.

ALLOCAZIONE CONTIGUA VS NON CONTIGUA

Function	Contiguous allocation	Noncontiguous allocation
Memory allocation	The kernel allocates a single memory area to a process.	The kernel allocates several memory areas to a process—each memory area holds one component of the process.
Address translation	Address translation is not required.	Address translation is performed by the MMU during program execution.
Memory fragmentation	External fragmentation arises if first-fit, best-fit, or next-fit allocation is used. Internal fragmentation arises if memory allocation is performed in blocks of a few standard sizes.	In paging, external fragmentation does not occur but internal fragmentation can occur. In segmentation, external fragmentation occurs, but internal fragmentation does not occur.
Swapping	Unless the computer system provides a relocation register, a swapped-in process must be placed in its originally allocated area.	Components of a swapped-in process can be placed anywhere in memory.

Per quanto concerne l'allocazione della memoria abbiamo detto che con l'allocazione contigua il kernel alloca per ogni processo un unico blocco di locazioni sequenziali ad un processo, nel caso della non contigua alloca blocchi di memoria di dimensioni variabili che si trovano in diverse zone della memoria.

Per quanto riguarda la traduzione degli indirizzi nell'allocazione contigua non è necessaria mentre è necessaria l'adozione dell'MMU per effettuare la traduzione degli indirizzi.

Esiste frammentazione esterna nella frammentazione contigua a seguito dell'adozione delle free list e si può avere frammentazione interna se i blocchi si allocano di dimensioni fisse. Nel caso dell'allocazione non contigua se si adotta la paginazione la frammentazione esterna non esiste e si può avere solo nell'ultima pagina frammentazione interna; nel caso della segmentazione possiamo avere frammentazione esterna e non

interna. Riguardo lo swapping, nel caso dell’allocazione contigua, tutto dipende se nell’architettura sottostante c’è un registro di rilocazione: se c’è lo swapping non è un problema, se non c’è lo swapping deve essere implementato in modo che quando si effettua lo swap out di un processo, al momento dello swap in devo allocare a quel processo lo stesso blocco di memoria che gli era stato allocato prima dello swap out (gestione poco flessibile e non ottimale). Nel caso dell’allocazione non contigua lo swapping si implementa facilmente perché si va a caricare diverse parti di un processo in qualsiasi parte della memoria.

PROTEZIONE DELLA MEMORIA

Abbiamo detto che l’indirizzo logico è costituito dalla coppia $(comp_i, byte_i)$ e che bisogna proteggere la memoria tra interferenze tra programmi (fare in modo che un processo non acceda a una porzione di memoria che non gli riguarda). Per fare questo ad ogni istruzione di tipo logico si fa un controllo dei limiti per essere certi che quella componente si trova nello spazio attribuito al processo. quindi l’MMU fa questo controllo sulla base dell’indirizzo logico e quando va a fare la traduzione va a controllare che $comp_i$ si trovi all’interno del processo in questione e che il byte si trovi in quel segmento, nel caso non fosse vero si genererebbe un interrupt. Il controllo dei limiti viene semplificato quando si adotta la paginazione perché con essa la componente dell’indirizzo logico che rappresenta l’offset in byte non può superare le dimensioni di una pagina e quindi di per sé non può andare oltre i propri limiti.

PAGINAZIONE

È uno schema di allocazione non contiguo della memoria e da un punto di vista logico nella paginazione lo spazio degli indirizzi da attribuire a un processo è visto come un’organizzazione lineare di

pagine (pagina 0,1,2...). Nella paginazione la dimensione di ciascuna pagina è di s byte, dove s è determinata dall'architettura della macchina. Questo s è una potenza di 2.

Gli indirizzi logici sono numerici e sono composti da una coppia (p_i, b_i) , dove p_i rappresenta il numero di pagina di quell'operando o istruzione e b_i rappresenta l'offset in byte. Quale caratteristiche hanno?

- $p_i \geq 0$ e $0 \leq b_i < s$

ESEMPIO

Esempio:

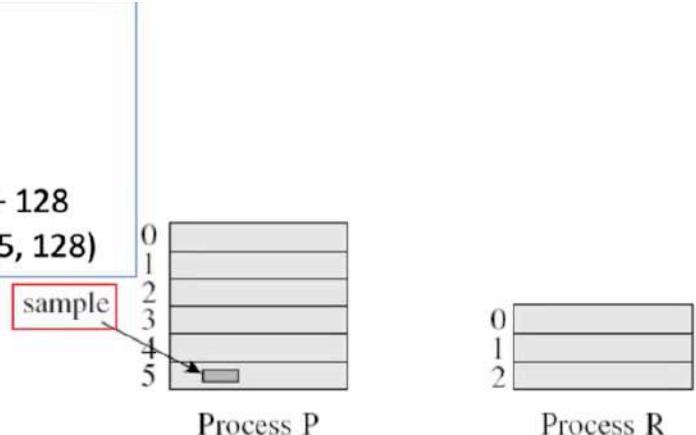
Dimensione pagine 1KB

Processo P dimensione 5500 byte

Processo R 2500 byte

sample ha indirizzo $5248 = 5 \times 1024 + 128$

MMU vede l'indirizzo come la coppia (5, 128)



Logical view of processes in paging.

Nell'esempio che vediamo ciascuna pagina ha una dimensione di 1KB (1024 byte).

Supponiamo di avere un processo P di 5500 byte e un processo R di 2500 byte. Se una pagina è di 1 KB, P ha bisogno di 6 pagine (con 5 non contiene il processo e allochiamo 6 pagine a P mentre a R vengono assegnate 3 pagine ($2500/1024=2,4\dots$ quindi 3 pagine) ecco perché l'ultima pagina porta a frammentazione interna: avrà più byte di quanto servono e non verrà riempita completamente).

Supponendo che in P ci sia la variabile sample con indirizzo 5248.

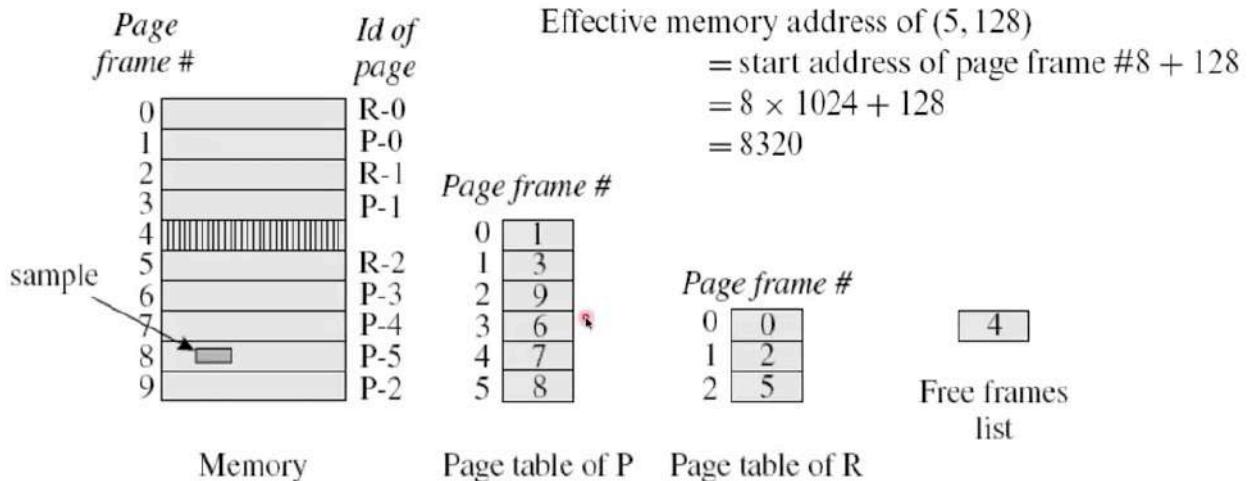
Siccome $5248 = 5 \times 1024 + 128$, l'indirizzo logico di sample sarà dato dalla coppia $(5, 128)$. 5 rappresenta il numero di pagina e 128 lo scostamento all'interno della pagina.

Le pagine logiche possono essere allocate a un insieme di pagine fisiche che nella memoria fisica non sono contigue.

PAGINAZIONE (cont.)

- La memoria fisica è suddivisa in aree chiamate **frame**, numerati a partire da zero.
- Un frame ha la stessa dimensione di un pagina
- Il kernel deve mantenere una lista dei frame liberi, per tenere traccia dei numeri di frame liberi.
- Per semplificare la traduzione degli indirizzi si usa una tabella nella quale il kernel tiene traccia di tutti i frame allocati a ogni pagina del processo.
- Per ogni processo il kernel costituisce una tabella delle pagine (PT)
 - o La PT ha un'entrata per ogni pagina del processo che indica il frame allocato alla pagina
 - o Mentre esegue la traduzione per un indirizzo logico (p_i, b_i) , la MMU usa il numero di pagina p_i per
 - Individuare la PT del processo
 - Ottenere il numero di frame allocato a p_i
 - Calcolare l'indirizzo di memoria effettivo

ESEMPIO



Ogni frame dimensione di **1KB**
Memoria 10KB -> frame 0-9
 sample ha indirizzo logico **(5, 128)**

Abbiamo una memoria di 10kb con ogni frame di 1kb, abbiamo quindi 10 frame numerati da 0 a 9. Supponiamo di avere P e R processi in memoria. P è costituito da 6 componenti da P0 a P5 e ogni componente è attribuito a una pagina. Lo stesso per R che va da P0 a P2 che ha 3 pagine.

Quando l'MMU va a consultare questa tabella sa che la pagina 0 è contenuta nella memoria fisica nel frame n°1, la pagina 1 nel frame 3, la pagina 2 nel frame 9 ecc... Lo stesso per R: la pagina 0 si trova nel frame 0, la pagina 1 nel frame 2 e la pagina 2 nel frame 5.

Il kernel mantiene una lista di frame liberi, in questo caso 4. Un indirizzo logico in questa organizzazione è costituito dalla solita coppia e l'MMU con il numero di pagina accede all'entrata della tabella di indice 5, scopre che la pagina 5 si trova nel frame 8, gli aggiunge lo scostamento e si ottiene l'indirizzo fisico 8320.

L'MMU non ha bisogno di fare questi calcoli che comporterebbe un overhead. L'MMU fa una **concatenazione di bit** tra indirizzi logici e fisici, grazie al fatto che le pagine sono di una dimensione che è una potenza di 2.

NOTAZIONE PER LA TRADUZIONE DEGLI INDIRIZZI

s dimensione di una pagina

l , lunghezza di un indirizzo logico (cioè, numero di bit)

l_p , lunghezza di un indirizzo fisico

n_b numero di bit usato per rappresentare il numero del byte in un indirizzo logico

n_p numero di bit usato per rappresentare il numero di pagina in un indirizzo logico

n_f numero di bit usato per rappresentare il numero di frame in un indirizzo fisico

- La dimensione di una pagina, s , è una potenza di 2

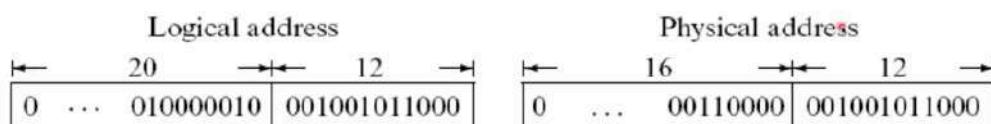
- n_b è scelto in modo che $s = 2^{n_b}$



L'MMU ottiene l'indirizzo logico concatenando gli n_p bit con gli n_b bit (nella rappresentazione complessiva dell'indirizzo logico sa che gli n_p bit rappresentano la macchina e n_b bit rappresentano il numero di byte).

Esempio: traduzione di indirizzo nella paginazione

- Indirizzi logici a 32 bit
- Dimensione pagina 4KB
 - 12 bit sono adeguati per indirizzare i byte in una pagina
 - $2^{12} = 4\text{KB}$
- Per una memoria di dimensioni di 256 MB, $l_p = 28$
- Se alla pagina 130 è allocato il frame 48,
 - $p_i = 130$ e $q_i = 48$
 - Se $b_i = 600$, gli indirizzi logici e fisici sono:



Vediamo un esempio con la figura in alto.

- Abbiamo una macchina ipotetica di 32 bit.
 - Le dimensioni di ogni pagina è 4KB
 - Di questi 32 bit dell'indirizzo logico posso 12bit per rappresentare i byte e il restante $32-12=20$ per i numeri di pagina.
 - Supponiamo di avere una memoria di 256MB, se li esprimiamo in byte sono 28bit.
 - L'indirizzo fisico è 28 bit.
 - Abbiamo detto che l'indirizzo fisico è costituito dal numero di bit coi quali rappresento i frame e siccome ciascun frame ha la stessa dimensione delle pagine, la porzione che rappresenta i byte è uguale a quella della pagina. Siccome è 12 e $l_p=28$ esce fuori che n_f è 16.
 - Supponiamo che alla pagina 130 è allocato il frame 48 nella tabella delle pagine
 - o B_i è 600
 - o Alla MMU arriva il numero 130 e 600 però è rappresentato in binario con un numero a 20 bit.
-

SEGMENTAZIONE

È un altro approccio per implementare l'allocazione non contigua della memoria. In pratica un programma è organizzato in un insieme di entità logiche. Il compilatore per ciascuna di questa entità gli fa corrispondere un segmento di memoria che avrà una dimensione sulla base della dimensione di queste entità e avrà un indirizzo di partenza. Il kernel mantiene una lista di segmenti (invece di avere una tabella delle pagine ha una tabella dei segmenti per un dato processo Q costituito da nome del segmento e indirizzo di partenza).

- Ogni indirizzo logico è formato da una coppia dove il primo elemento è il numero di segmento e il secondo elemento è lo scostamento in byte all'interno del segmento.
- Supponiamo che in update ci sia un'istruzione get_sample che si trova al byte 232, allora l'indirizzo è (update, get_sample). L'indirizzo di memoria effettivo è dato da indirizzo di partenza di update + scostamento in byte di get_sample e avremo 91608.
- Rispetto a prima c'è da dire che i segmenti sono di dimensioni variabili, ciò significa varie cose:
 1. La frammentazione esterna si può avere perché alloco memoria ai vari segmenti e possono rimanere piccoli segmenti che non si possono allocare ad alcun segmento.
 2. Se i segmenti hanno dimensione variabile, quando si va a fare la traduzione degli indirizzi, non si può applicare la concatenazione (era frutto che ciascun frame fossero potenze di 2 ed uguali) ma si va a fare una somma.
 3. Il kernel come gestisce queste aree libere da allocare ai vari segmenti? Lo può fare con una free list, è come se si avesse una strategia simile all'allocazione contigua solo che i segmenti possono trovarsi in qualsiasi parte della memoria. Quando un segmento viene rilasciato, quel blocco libero viene inserito nella lista dei blocchi liberi.

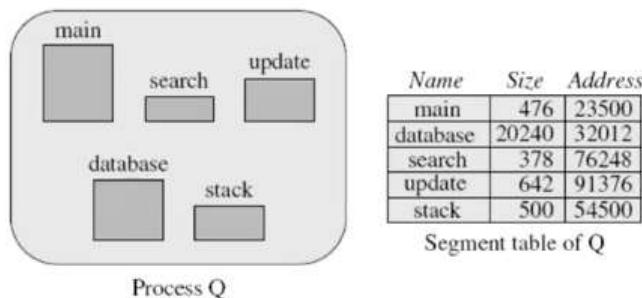


Figure 11.21 A process Q in segmentation.

- Il vantaggio principale è che posso condividere porzioni tra i vari processi, immaginiamo che alcuni segmenti siano relativi a librerie condivise: con la segmentazione è semplice effettuare questa condivisione. C'è un problema legato alla frammentazione esterna.
- Per ovviare a questo problema si adotta una segmentazione con paginazione.

SEGMENTAZIONE CON PAGINAZIONE

- A ogni porzione si associa un segmento ma all'interno di ciascun segmento si va a paginare. Le varie porzioni del segmento sono attribuite a un numero di pagine intere, posso avere frammentazione interna nell'ultima pagina che viene allocata a uno specifico segmento. Si evita la frammentazione esterna.
- Nella tabella dei segmenti abbiamo il segmento, la sua dimensione e un elemento che indica l'indirizzo della tabella delle pagine di quel segmento, quindi l'MMU è in grado di risalire alla tabella delle pagine di ogni segmento e la costruzione dell'indirizzo fisico viene fatta alla stessa maniera della paginazione.
- La traduzione di un indirizzo logico (s_i, b_i) è fatta in due passi:
 1. Dall'entrata di s_i nella *tabella dei segmenti* è determinato l'indirizzo della tabella delle pagine
 2. Il numero del byte b_i è diviso nella coppia (ps_i, bp_i) dove ps_i è il numero di pagina nel segmento s_i e bp_i è il numero del byte nella pagina p_i . Il calcolo dell'indirizzo effettivo è fatto come nella paginazione

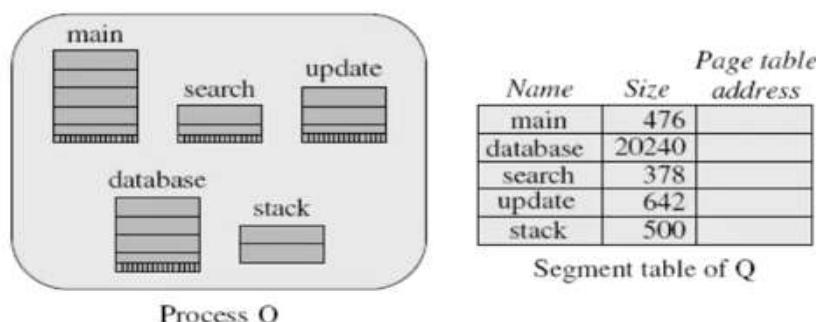


Figure 11.22 A process Q in segmentation with paging.

LEZ 21/05/2021

MEMORIA VIRTUALE

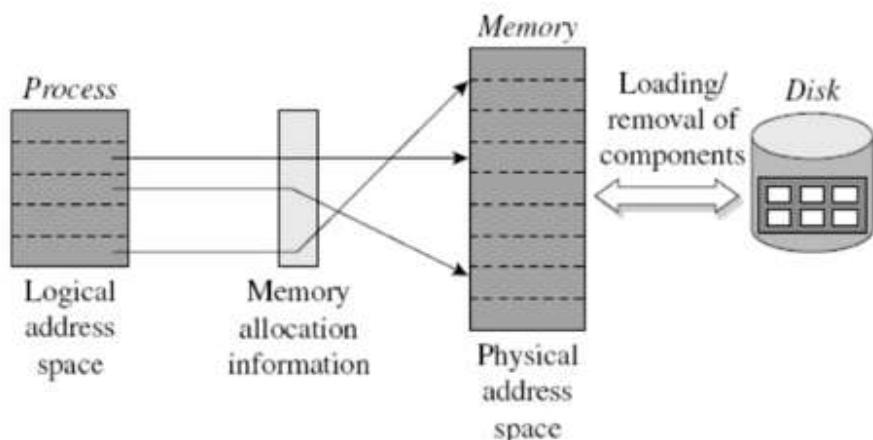
- Perché i moderni sistemi operativi utilizzano la memoria virtuale? La usano per due motivi: il primo è avere la possibilità di eseguire programmi la cui dimensione può anche eccedere quella della memoria fisica del calcolatore e il secondo è cercare di diminuire il carico di memoria per ciascun processo in modo da poter avere la possibilità di mantenere memoria a più processi ed eseguirne di più in maniera simultanea (aumenta grado di multi programmazione). C'è da dire che alla base dell'uso della memoria virtuale c'è uno schema di allocazione della memoria non contigua nel senso che ciascun processo è visto come un insieme di componenti e ciascuna componente può essere memorizzata ovunque nella porzione di memoria dell'area fisica. L'allocazione non contigua può dar luogo a frammentazione esterna, ma con la memoria virtuale è possibile allocare porzioni piccole di memoria a diverse componenti di un processo minimizzando la frammentazione esterna.

L'uso della memoria virtuale comporta l'uso di un supporto HW per rendere più efficienti le operazioni e per supportare il gestore della memoria virtuale (componente SW).

CONCETTI BASE

Formalmente possiamo definire la memoria virtuale come quella parte della gerarchia di memoria che è costituita dalla memoria principale e dal disco e consente di eseguire processi mantenendo nella memoria centrale solo una parte dello spazio di indirizzamento dei processi, è possibile avere solo alcune componenti, le altre potrebbero non essere in memoria. Questo

significa che le rimanenti parti non in memoria centrale risiedono sul disco. Nello schema della memoria virtuale entra in gioco la MMU che si preoccupa di tradurre gli indirizzi da logici a fisici. La cosa importante è che un processo può funzionare in maniera corretta anche se non tutto il suo spazio di indirizzamento è presente in memoria però la chiave perché tutto funzioni è che nel momento in cui all'interno del programma in esecuzione si fa riferimento a un indirizzo che non è in memoria, la porzione di memoria riferita da quella istruzione viene caricata on demand (al momento).



Nella figura abbiamo una visione della memoria virtuale: abbiamo il processo con il suo spazio di indirizzamento logico (5 componenti), ci sono 3 componenti in memoria fisica e quando si fa richiesta di essi si va a prelevare l'indirizzo fisico. Il gestore della memoria virtuale, nel momento in cui dovesse fare richiesta di una componente che non si trova in memoria fisica, a quel punto dovrebbe avviare una procedura per caricare quel componente dalla memoria secondaria alla memoria centrale.

Quando si avvia un processo, avviene il caricamento solo di un componente di un processo, in particolare quello che contiene l'indirizzo di avvio del processo stesso, a partire da quello ci sarà il

riferimento di altri componenti che saranno caricati laddove necessari. È importante che non solo si gestisca il fatto di dover caricare in memoria ma per garantire un buon grado di multiprogrammazione e garantire ai vari processi di poter essere schedulati, può capitare che a un dato processo a alcune componenti che già sono presenti in memoria siano rimosse e dando la possibilità ad altri processi di avere le componenti caricate per essere eseguiti. Queste componenti rimosse saranno ricaricate al momento opportuno.

Le prestazioni dipendono dal tasso con cui dobbiamo caricare le componenti dal disco alla memoria principale, sfruttiamo il principio di località dei riferimenti per mantenere bassi questi tassi (avere una probabilità elevata di trovare un componente già in memoria principale piuttosto che doverlo caricare dal disco).

Come è realizzata la memoria virtuale? Prima di questo riepiloghiamo i metodi su cui è basata la memoria virtuale.

- La memoria virtuale usa la paginazione maggiormente.

MEMORIA VIRTUALE CON PAGINAZIONE

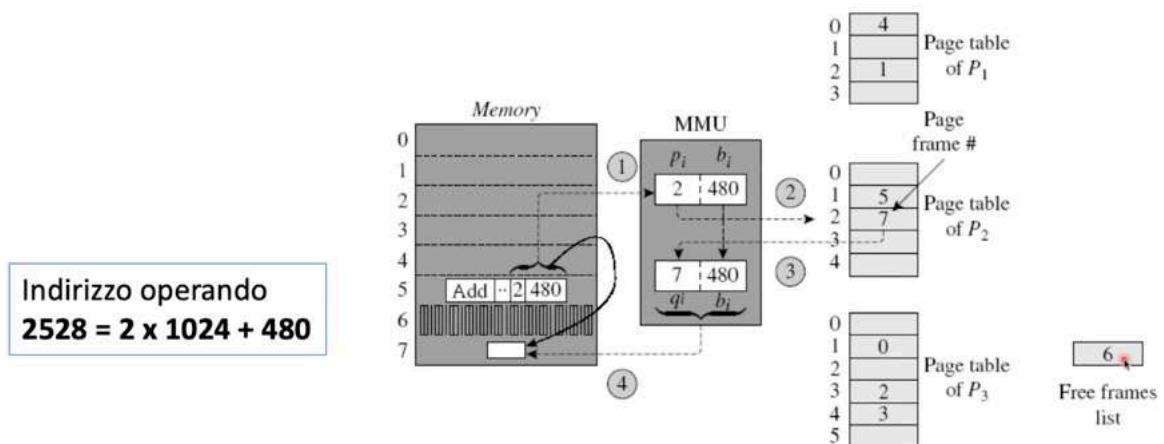


Figure 12.2 Address translation in virtual memory using paging.

Ricordiamo come funziona la paginazione: lo spazio di indirizzamento di un processo è organizzato in un insieme di blocchi di dimensione fissa chiamato pagine e ogni pagina ha una dimensione fissa che è una potenza di 2. Ciascuna pagina che fa parte della memoria virtuale ha una corrispondenza con un blocco fisico di memoria, in particolare la memoria fisica nello schema di paginazione è organizzata anch'essa in blocchi di dimensione fissa che hanno la stessa dimensione della pagina e che prendono il nome di frame. In un sistema di questo tipo un indirizzo logico è costituito da una coppia (p_i, b_i) in cui p_i fa riferimento al numero di pagina e b_i è lo scostamento di quella pagina. Il processo di traduzione da parte della MMU consiste nell'individuare l'indirizzo di partenza del frame che contiene la pagina e aggiungere lo scostamento in byte. Ciò è realizzato in maniera molto efficiente: essendo le pagine e i frame di dimensioni pari a potenze di 2, le operazioni di somma si ottengono con una concatenazione di bit e viene mantenuta una tabella della pagine in modo che quando alla MMU arriva un indirizzo logico prende l'indirizzo, il numero di pagina, accede all'entrata della tabella dell'indice in questione dove vi è il frame che memorizza la pagina, prende il numero del frame, lo concatena con lo scostamento in byte e insieme questa concatenazione rappresenta l'indirizzo fisico in memoria.

- Nel caso della memoria virtuale può succedere che il nostro processo P2 faccia riferimento a un indirizzo logico che coinvolge la pagina 3. Così facendo arriverebbe all'MMU ma si accorgerebbe che se avessi un'istruzione in cui ho “3”, vede che non è nella memoria fisica e questo comporterebbe un'interrupt da parte dell'MMU che indica che la pagina non è in memoria e questo attiva il gestore della memoria virtuale che una volta che ha ricevuto la

notifica dell'interrupt parte per avviare il processo di caricamento della pagina 3 del processo P2 dalla memoria secondaria a quella principale. A quel punto supponendo che quel frame sia in numero 6, una volta fatto il caricamento, si ripristina l'esecuzione del processo dall'istruzione che ha causato l'interrupt (page fault).

PAGINAZIONE SU RICHIESTA

Come si gestisce il caricamento di una pagina che non è in memoria centrale?

- Perché tutti funzioni è necessario che l'intero spazio di indirizzamento di un processo sia mantenuto sul disco: abbiamo sul disco una copia dell'intero spazio di indirizzamento, quest'area è chiamata *area di swap*.
- Nel momento in cui viene avviato il processo, il gestore della memoria virtuale va a allocare lo spazio di swap e va a copiare codice, dati. Viene caricata però la pagina che contiene l'indirizzo d'avvio del processo quindi da quel momento in poi tutti i riferimenti alle altre pagine saranno gestiti andando a caricare le pagine di volta in volta dalla memoria centrale. Questo tipo di operazione prende il nome di **page-in**.
- **Page-in:** caricamento in memoria da disco della pagina del processo che riferisce un'istruzione o dato in una pagina non in memoria.
- Per dare la possibilità ad altri processi di vedersi caricate pagine in memoria, può avvenire un'operazione di rimozione di alcune pagine ad alcuni processi → **page-out:** rimozione dalla memoria di una pagina e sua memorizzazione su disco se modificata dall'ultimo caricamento.
- **Sostituzione della pagina:** strategia attraverso la quale posso decidere quale delle pagine di un processo possono

essere rimosse dalla memoria principale. La sostituzione viene effettuata da un algoritmo e consiste nel caricare la pagina in un frame che conteneva un'altra pagina, questo comporta il page out se la pagina che si sta rimuovendo era stata modificata dall'ultimo caricamento. Dopo il page out c'è il page in (rimuovo una pagina per caricarne un'altra).

PAGINA SU RICHIESTA: TABELLA DELLE PAGINE

Misc info						
Valid bit	Page frame #	Prot info	Ref info	Modified	Other info	

Field	Description
Valid bit	Indicates whether the page described by the entry currently exists in memory. This bit is also called the <i>presence</i> bit.
Page frame #	Indicates which page frame of memory is occupied by the page.
Prot info	Indicates how the process may use contents of the page—whether read, write, or execute.
Ref info	Information concerning references made to the page while it is in memory.
Modified	Indicates whether the page has been modified while in memory, i.e., whether it is dirty. This field is a single bit called the <i>dirty</i> bit.
Other info	Other useful information concerning the page, e.g., its position in the swap space.

Il gestore ha bisogno di info per sapere quel pagina caricare, quale rimuovere o sostituire. Per fare questo, ogni entrata della tabella delle pagine contiene info aggiuntive su ciascuna pagina del processo e queste info le vediamo nella figura in alto.

1. Valid bit: indica se la pagina che si sta riferendo è in memoria (1) oppure no (0).
2. Page frame: frame che contiene la pagina.

3. Prot info: info di protezione che indicano il processo che tipo di riferimento è autorizzato a fare (se può scrivere, leggere solamente, ...).
 4. Ref inf: riferimenti alla pagina fatti quando la pagina era in memoria. È utile all'algoritmo di sostituzione: ref info ci dà la possibilità di sapere se quella pagina è stata usata di recente oppure no
 5. Bit modified: indica se la pagina è stata modificata in memoria rispetto all'ultimo caricamento. È anche detto *dirty bit*.
 6. Altre info: possono essere utili tra le quali la posizione della pagina sul disco fisico.
-

PAGINAZIONE SU RICHIESTA: FAULT DI PAGINA

Table 12.2 Steps in Address Translation by the MMU

Step	Description
1. Obtain page number and byte number in page	A logical address is viewed as a pair (p_i, b_i) , where b_i consists of the lower order n_b bits of the address, and p_i consists of the higher order n_p bits (see Section 11.8).
2. Look up page table	p_i is used to index the page table. A page fault is raised if the <i>valid bit</i> of the page table entry contains a 0, i.e., if the page is not present in memory.
3. Form effective memory address	The <i>page frame #</i> field of the page table entry contains a frame number represented as an n_f -bit number. It is concatenated with b_i to obtain the effective memory address of the byte.

Mentre la MMU va a fare la traduzione di un indirizzo logico, va a controllare se il bit di validità della pagina è 1 o 0: se 1 sa che la pagina è in memoria centrale oppure non c'è e si deve verificare

un page fault. In quel caso la MMU consultando il valid bit genera questo interrupt.

La routine dell'interrupt sa che è stato causato da un page fault, va a invocare il gestore della memoria virtuale al quale passa il numero di pagina che lo ha causato e a quel punto il gestore va a effettuare un'operazione di page-in in modo tale da portare la pagina dalla memoria secondaria a quella centrale.

Nello schema in alto vediamo i passi durante la traduzione dell'indirizzo logico da parte dell'MMU.

1. Abbiamo la coppia (p_i, b_i) , va a considerare se la pagina è memoria o meno, se non c'è attiva il gestore della memoria virtuale, se valid è 1 usa p_i come indice nella tabella delle pagine e va a calcolarsi l'indirizzo fisico del frame che contiene quella pagina.
 2. Sia il gestore che la MMU interagiscono per verificare quando è necessario effettuare un page-in. La MMU fa il controllo e se manca la pagina della memoria, genera il page fault che attiva il gestore.
-

In questo schema le linee tratteggiate sono operazioni svolte dalla MMU mentre quelle continue sono gli accessi alla struttura dati che fa il gestore della memoria virtuale.

Nell'esempio vediamo i passi che si verificano quando avviene la traduzione degli indirizzi che comporta un page fault e il conseguente page-in.

Facciamo riferimento a un indirizzo logico (3,682). L'MMU riceve questo indirizzo, accede alla tabella delle pagine, va all'entrata 3 e scopre che il bit di validità è zero perché nell'esempio la pagina 3 non è in memoria (ci sono solo pagina 1 e 2). Siccome la pagina 3 non è presente, la MMU genera il page fault che viene analizzato dal vettore del tipo di interrupt, verifica e attiva il

gestore della memoria virtuale che riceve il numero di pagina 3 come argomento. Il gestore della memoria virtuale va a prelevare un frame dalla memoria libera che dovrà contenere la pagina che deve essere caricata in memoria centrale, avvia poi un'operazione di I/O che comporta il caricamento della pagina 3 al frame numero 6.

Fa poi il page-in: all'interno del frame 6 viene caricata la pagina numero 3 e infine aggiorna la tabella delle pagine che sarà aggiornata modificando il bit di validità a 1 e all'entrata corrispondente sarà inserito il numero di frame 6.

Paginazione su richiesta (cont.)

E' generato un page fault interrupt perché il Valid bit della pagina 3 è 0

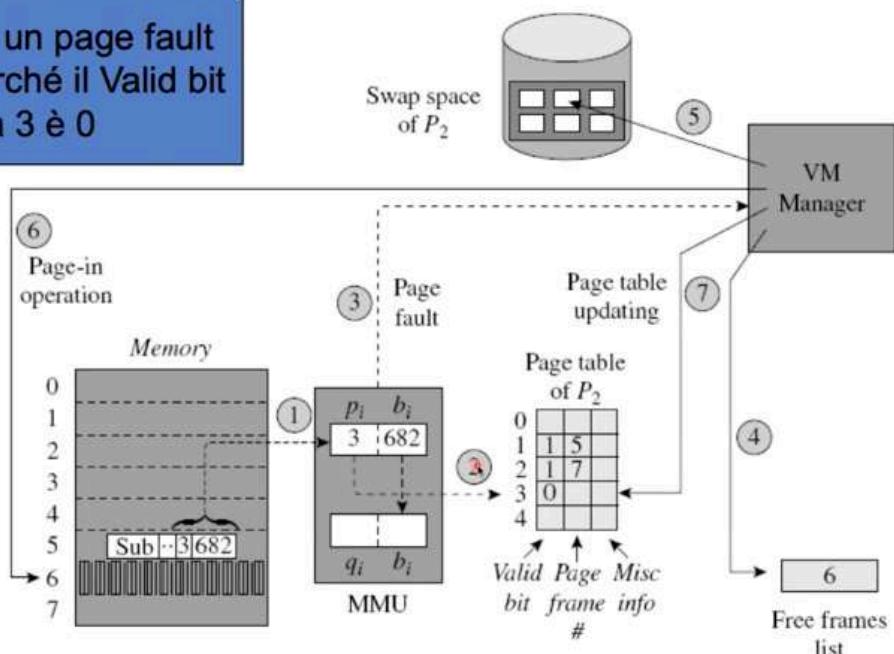


Figure 12.4 Demand loading of a page.

Passo 4: avvia operazione di I/O per caricare la pagina nel frame 6

- Completato I/O il gestore aggiorna l'entrata della tabella delle pagine

PAGINAZIONE SU RICHIESTA: SOSTITUZIONE DI PAGINA

Se non ci fosse alcun frame libero cosa succede?

È necessario attivare un algoritmo di sostituzione delle pagine che libera un frame occupato.

Quando l'algoritmo trova un rame da liberare, sulla base delle info contenute nella pagina, se la pagina è dirty deve fare un page-out (deve sovrascrivere la pagina in questione e deve memorizzare sul disco la pagina che sta rimuovendo).

Page-in e page-out sono operazioni di I/O per cui creano un traffico di pagine che prende il nome di “traffico di pagine” (movimento di pagine da e verso la memoria). Un elevato I/O di pagine può degradare le prestazioni del sistema.

TEMPO DI ACCESSO IN MEMORIA EFFETTIVO (EAT)

Nel momento in cui si usa la memoria virtuale con paginazione, qual è il tempo effettivo di accesso alla memoria?

Questo tempo di accesso è il tempo medio di accesso atteso dal processo e dipende da 2 fattori:

- La pagina a cui si fa riferimento in memoria
- La pagina a cui si fa riferimento non è in memoria

Possiamo considerare la formula in basso per calcolare il tempo effettivo di accesso alla memoria:

$$\begin{aligned} \text{Effective memory access time} = & pr_1 \times 2 \times t_{\text{mem}} \\ & + (1 - pr_1) \times (t_{\text{mem}} + t_{\text{pfh}} + 2 \times t_{\text{mem}}) \end{aligned}$$

hit ratio memoria → pr_1 probability that a page exists in memory
 t_{mem} memory access time
 t_{pfh} time overhead of page fault handling

Entrano in gioco il tempo che impiega l'MMU per fare la traduzione dell'indirizzo e quello del gestore se si dovesse verificare un page fault.

Se pr_1 rappresenta la probabilità che la pagina esista in memoria (hit ratio della memoria), allora il tempo è suddiviso in due componenti:

1. Pagina in memoria: $probabilità * 2 * tempo necessario per accedere in memoria$. Due volte perché si accede 2 volte quando ci troviamo in memoria: accediamo alla tabella delle pagine e poi accediamo quando andiamo al frame effettivo.
 2. Pagina non in memoria: si aggiunge un altro componente, la probabilità che la pagina non sia in memoria.
 $1 - probabilità che ci sia * (tempo di accesso della memoria + overhead + 2 * tempo di accesso alla memoria)$.
- L'EAT si può abbassare riducendo i page fault
 - Un modo consiste nel caricare le pagine prima che siano necessarie ad un processo
 - Windows: carica una pagina all'occorrenza di un page fault ed anche alcune pagine adiacenti
 - Linux: consente ad un processo di specificare quali pagine dovrebbero essere precaricate
 - Il programmatore può usare questa possibilità per migliorare il tempo di accesso effettivo

SOSTITUZIONE DELLE PAGINE

Se c'è stato un page fault e si deve sostituire una pagina perché non ci sono frame liberi, si attiva l'algoritmo di sostituzione delle pagine. L'algoritmo deve scegliere un frame da liberare sulla base di un criterio in modo da evitare che dopo aver rimosso una pagina dalla memoria fisica alla prossima istruzione ci sia un richiamo a quella pagina rimossa.

Gli algoritmi di sostituzione delle pagine si basano sulla **località dei riferimenti**: nel breve tempo gli indirizzi logici generati all'interno di un programma si raggruppano in specifiche porzioni del loro spazio di indirizzamento logico, ciò avviene per due ragioni:

1. Quando si esegue un codice di un programma l'esecuzione è sequenziale perché solo il 20% delle istruzioni sono di salto
2. I processi su alcune strutture dati tendono a fare operazioni simili.

Da questo ragionamento si evince che le istruzioni e i riferimenti ai dati in esse tendono a essere vicine a istruzioni eseguite recentemente o a un dato referenziato in precedenza. Un algoritmo di sostituzione non deve sostituire pagine che fanno riferimento a istruzioni o dati riferiti nell'ultimo momento.

- Possiamo definire la **località corrente di un processo**: fa riferimento all'insieme delle pagine referenziate in poche istruzioni precedenti. Questo parametro consente di abbassare il numero di page fault.

Ciò nonostante, anche se evito di selezionare una pagina nella località corrente, ci sarà comunque un page fault, infatti se definiamo:

- **Regione di prossimità di un indirizzo logico** a_i : tutti gli indirizzi logici in prossimità dell'istruzione a_i .
- Può succedere che questa regione sia più grande di una pagina e servirebbero più pagine (page fault) o un'istruzione potrebbe non essere vicino a istruzioni precedenti (**shift della località di un processo**).

Questa idea la rappresentiamo con la seguente immagine:

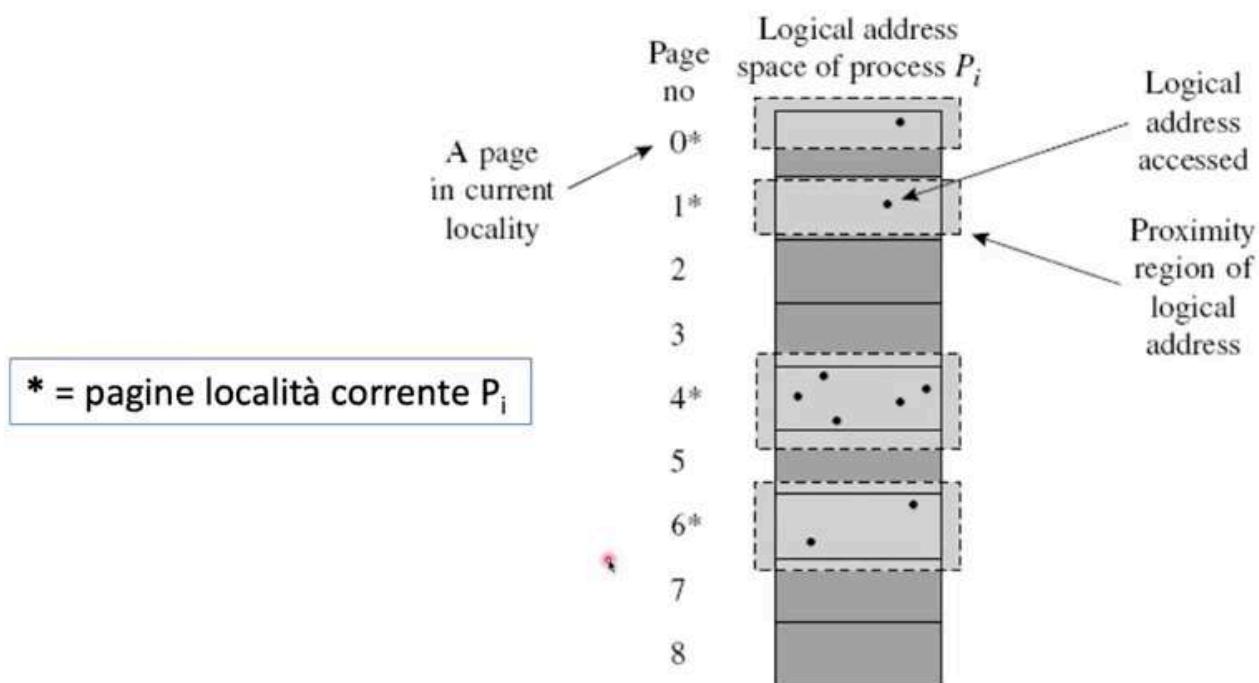


Figure 12.5 Proximity regions of previous references and current locality of a process.

- Questo è lo spazio di indirizzamento logico del processo P_i , i puntini neri rappresentano gli indirizzi logici usati dal processo e abbiamo le regioni di prossimità (quelle tratteggiate).
- Quelle in grigio scuro sono le pagine con i relativi numeri. Gli asterischi rappresentano le pagine nella località corrente del processo P_i , vediamo che gli indirizzi referenziati nella pagina 4 hanno una regione di prossimità che coinvolge la pagina 4 ma anche la pagina 3 e 5 che non sono parte della località corrente del processo. Se rimuovessi la pagina 3 o 5,

che non fanno parte della località, un’istruzione successiva nella pagina 4 potrebbe usare un’istruzione della pagina 3 (in parte contenuta nella località della pagina 4) che è stata però rimossa (page fault).

- Questo dimostra come effettivamente anche se si usa il principio di località può accadere che si possa referenziare una pagina che non fa parte della località corrente del processo.
 - Sfruttando queste info, il gestore decide quale pagina sostituire.
-

ALLOCAZIONE DELLA MEMORIA AD UN PROCESSO

È importante osservare come varia il numero di page fault al variare delle allocazioni della memoria che si alloca a un processo. Se si volesse abbassare il numero di page fault si potrebbe allocare più memoria a un processo (avremo più pagine). Se rappresentiamo con un grafico il numero di page fault rispetto al numero di frame allocati a un processo, il page fault si abbatte. Non possiamo però allocare memoria in maniera indiscriminata: nonostante il page fault diminuisca, se si alloca più memoria a un processo significa che si sta togliendo memoria ad altri processi riducendo il grado di multiprogrammazione del sistema per cui la quantità di memoria da allocare deve rispettare il giusto compromesso tra avere una memoria sufficiente per un processo in maniera tale da contenere i page fault e avere buone prestazioni che garantiscono un buon grado di multiprogrammazione (area grigio chiaro in figura). Ci sono meccanismi in cui si valutano dinamicamente le condizioni di esecuzione di ciascun processo e quale può essere l’allocazione ottimale.

Quando ci troviamo a sinistra del grafico, c'è un elevato tasso di page fault perché ci sono pochi frame allocati e se si fa riferimento alle pagine è probabile che queste non siano in memoria, ciò comporta che si ha un'elevata successione di operazioni di I/O e la CPU è in stato idle (non viene sfruttata) → questa operazione si chiama **TRASHING**

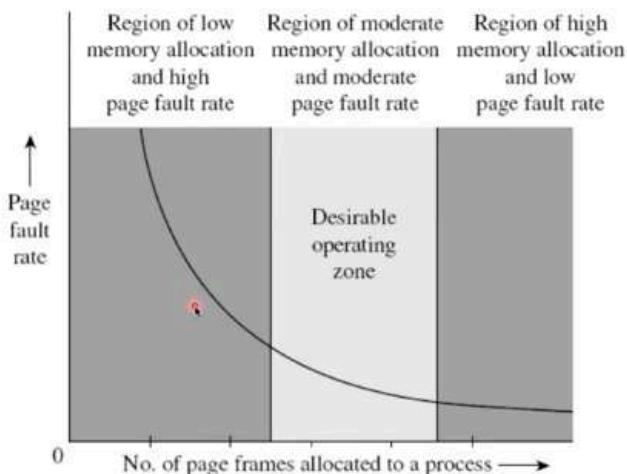


Figure 12.6 Desirable variation of page fault rate with memory allocation.

DIMENSIONE DELLA PAGINA OTTIMALE

Quando si va a mettere a punto la memoria virtuale si deve considerare qual è la dimensione ottimale di pagina.

Abbiamo detto che le dimensioni di una pagina dipende dall'HW sottostante ma è importante la dimensione che si sceglie.

La dimensione della pagina impatta su diversi aspetti:

- Numero di bit richiesti per rappresentare il byte.
- Se uso pagine più grandi, quando alloco l'ultima pagina a un processo potrei avere maggiore frammentazione interna e avere spreco di memoria.
- Impatta sulla dimensione della tabella delle pagine
- I tassi di page fault quando una quantità fissa di memoria è allocata ad un processo

Ci sono leggi empiriche sull'ottimale dimensione della pagina. Si cerca di individuare una convenzione tra costi HW e operazioni efficienti.

HARDWARE DI PAGINAZIONE

Quando si utilizza la memoria virtuale si ha necessità di un supporto HW, c'è un componente SW anche cioè il gestore della memoria virtuale che deve essere supportato da HW.

In un sistema multiprogrammato possiamo avere tanti processi in memoria: per ogni processo abbiamo una tabella delle pagine quindi dobbiamo avere un supporto per identificare le varie tabelle delle pagine. Per fare ciò la MMU contiene un registro che si chiama **“Page Table Access Register → PTAR”**, il registro di indirizzo della tabella delle pagine che punta all'inizio della tabella. La MMU usa questo registro per individuare la tabella delle pagine corretta relativa a un indirizzo logico.

Dato un indirizzo logico (p_i, b_i) la MMU va a determinare la tabella delle pagine corretta perché calcola il contenuto di PTAR a cui aggiunge poi il numero di pagine moltiplicato per la lunghezza dell'entrata delle pagine. Quando arriva un indirizzo logico si utilizza il numero di pagina che viene sommato a PTAR e questo consente di individuare la tabella delle pagine corrispondente in memoria e di fare le operazioni successive.

- Dove viene preso PTAR?
 - Il kernel può memorizzare questo indirizzo all'interno del PCB di un processo. Quando schedulo un processo dal PCB prendo l'indirizzo iniziale della tabella delle pagine e lo metto in PTAR, quando c'è l'indirizzo logico la MMU consulta PTAR e svolge le operazioni corrispondenti.

- Si pone un problema di protezione: nella memoria ho tante tabelle delle pagine e devo evitare interferenze tra pagine di processi diversi e avviene un'operazione di protezione della memoria.

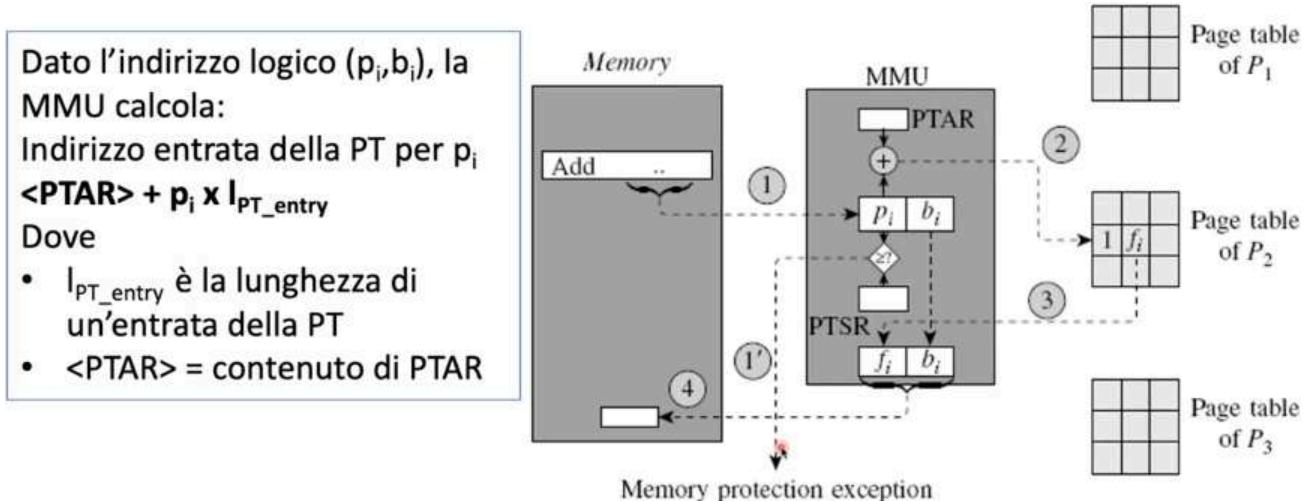


Figure 12.7 Address translation in a multiprogrammed system.

- Nella paginazione intervengono diverse componenti HW per poter semplificare il compito di gestione di tutta la memoria virtuale:
 - Controllo della protezione della memoria
 - Traduzione efficiente della traduzione degli indirizzi
 - Supporto alla sostituzione delle pagine andando a collezionare i riferimenti fatti in precedenza alle pagine

Table 12.3 Functions of the Paging Hardware

Function	Description
Memory protection	Ensure that a process can access only those memory areas that are allocated to it.
Efficient address translation	Provide an arrangement to perform address translation efficiently.
Page replacement support	Collect information concerning references made to pages. The virtual memory manager uses this information to decide which page to replace when a page fault occurs.

HW DI PAGINAZIONE: PROTEZIONE DELLA MEMORIA

Per poter implementare la protezione della memoria in questo schema di memoria virtuale con paginazione si usa un ulteriore registro detto **Page table size Register → PTSR**, è un'informazione che usa l'MMU per verificare che un processo acceda a una pagina sua e non a una che non esiste.

PTSR dà la dimensione della tabella delle pagine: nel momento in cui ho il momento della pagina lo vado a confrontare con la dimensione del PTSR, se è maggiore non esiste altrimenti si procede normalmente. PTSR viene usato come il registro dimensione delle info contenute nel PSW.

Devo poi verificare se una data pagina può essere acceduta con i relativi diritti da parte di quel processo: se il processo vuole modificare la pagina ma non ha i privilegi sta violando la protezione. Per verificare i permessi si controlla il campo Prot Info dell'entrata della pagina nella tabella delle pagine che contiene i privilegi di accesso di un processo a una pagina (codificati come bit dove ogni bit è un permesso).

HW DI PAGINAZIONE: TRADUZIONE INDIRIZZO E GENERAZIONE PAGE FAULT

Un'altra cosa importante è che nel momento in cui si fa riferimento a una tabella delle pagine durante il processo di traduzione di un indirizzo ci sono dei tempi che riguardano l'accesso alla memoria centrale.

Si può andare a risparmiare su questi tempi introducendo un ulteriore supporto HW che è costituito dal **TLB** (una sorta di memoria cache).

- **TLB:** memoria associativa piccola e veloce usata per accelerare la traduzione dell'indirizzo. Si cerca di fare in modo che le info sulle pagine per fare la traduzione,

piuttosto che reperirle nella tabella delle pagine, siano presenti nel TLB velocizzando i processi.

Il TLB è costituito da un insieme di entrate dove ciascuna di essa contiene il numero di pagina a cui si fa riferimento, il frame che contiene quella pagina e info sulla protezione relativa alla pagina. Con tutte queste info si riesce a tradurre l'indirizzo e si evita di accedere alla tabella delle pagine in memoria centrale. Ovviamente è una piccola memoria e queste info si possono avere per un numero di pagine limitato.

- Come funziona la traduzione dell'indirizzo con il TLB? (figura)

Abbiamo un indirizzo logico (2,480): con il TLB si vede se la pagina 2 è contenuta nel TLB e si prende il numero del frame che contiene la pagina 2 e le info sulla protezione e a questo punto si costruisce l'indirizzo fisico consultando solo il TLB e accedo in memoria. Può succedere che si fa riferimento a una pagina non presente in TLB e si verifica un “TLB MISS” (se c’è TLB HIT) e si accede normalmente alla memoria centrale e si reperiscono le relative info nella tabella delle pagine

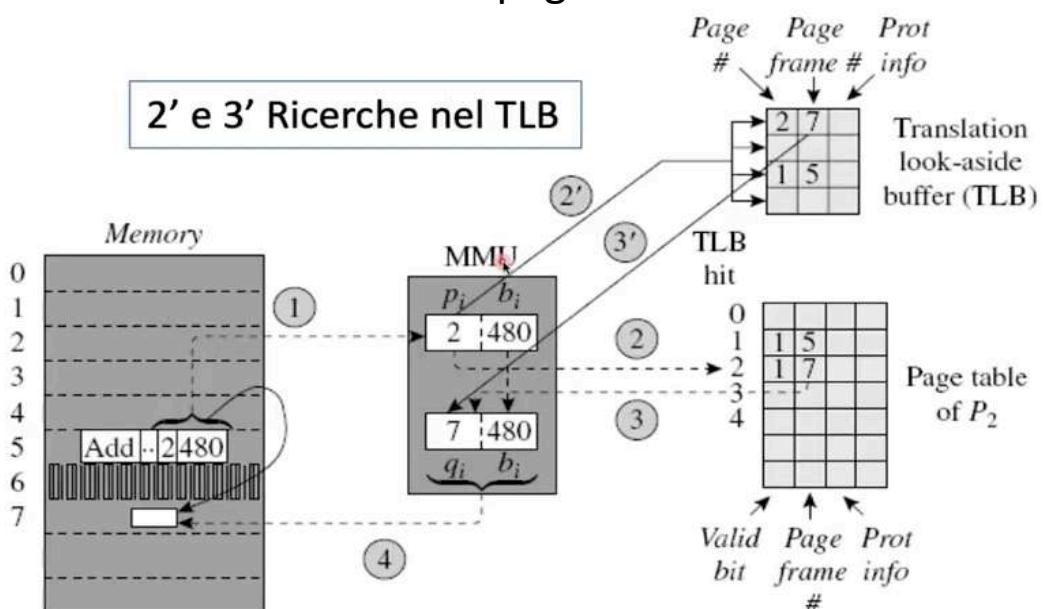
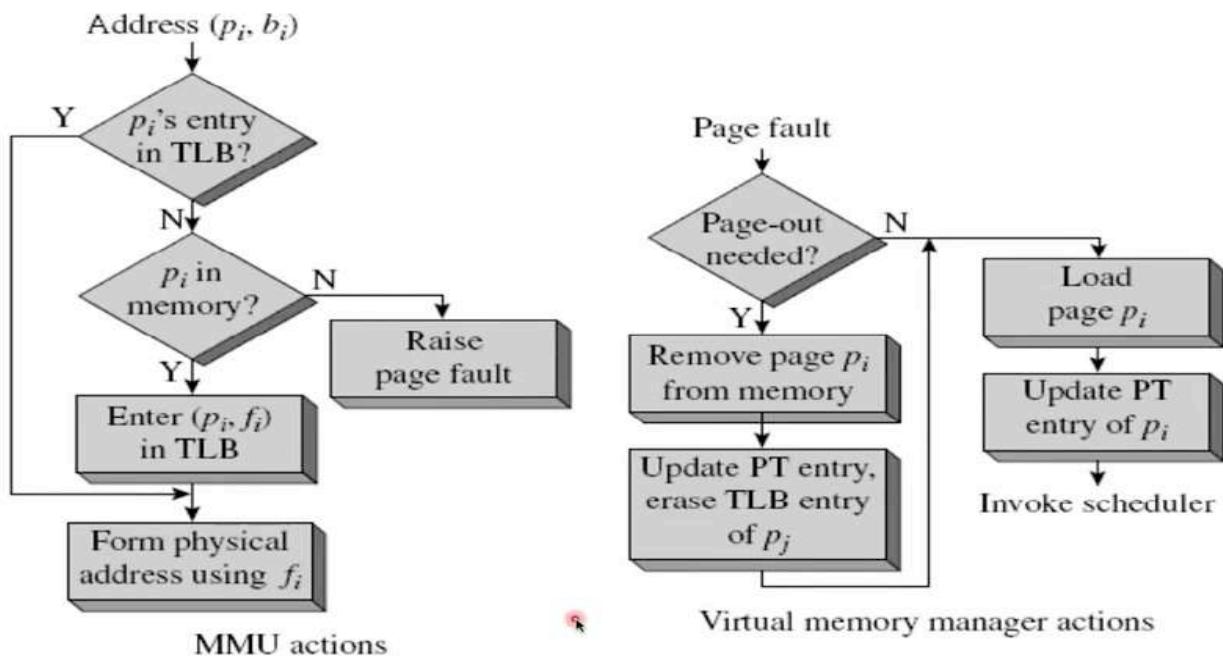


Figure 12.8 Address translation using the translation look-aside buffer and the page table.

TRADUZIONE INDIRIZZO E GENERAZIONE PAGE FAULT

Dobbiamo stabilire i passi che si verificano durante la traduzione degli indirizzi e page fault che è eseguito con la memoria virtuale con paginazione e TLB. Vediamo passo passo:

1. La pagina p_i si trova nel TLB?
 - a. Se non si trova nel TLB verifico se si trova all'interno della memoria accedendo alla tabella delle pagine: se non è in memoria l'MMU genera page fault che attiva il gestore della memoria virtuale.
 - b. Se non era in TLB ma era in memoria principale la pagina va messa nel TLB, fatto ciò, costruisco l'indirizzo logico e lo uso per le operazioni
2. Se si verifica page fault:
 - a. È necessario un page out? Ci sono frame liberi in memoria? Se non ci sono frame liberi si carica la pagina che interessa e vado ad aggiornare la tabella delle pagine e lo scheduler riprende le esecuzioni. Se è necessario rimuovo la pagina della memoria, aggiorno la tabella delle pagine e se la pagina era presente nel TLB la vado a rimuovere



I TLB possono essere gestiti in HW o SW, in SW c'è il vantaggio che il gestore possa dedicare diverse strategie per memorizzare e tabella delle pagine.

Se l'MMU fa traduzione degli indirizzi accedendo alla tabella delle pagine, può succedere che si può andare a minare la protezione della memoria se la MMU fa una traduzione degli indirizzi con entrate della tabella delle pagine riempite con processi precedenti. Per evitare questi problemi ci sono due approcci:

1. Ogni entrata del TLB può contenere l'id del processo in esecuzione nel momento della creazione dell'entrate.

Avendo l'id del processo non si può accedere al TLB per fare una traduzione relativa a un altro processo relativa a quello corrente.

2. Il kernel deve scaricare (flush) il TLB mentre esegue la commutazione tra processi

➤ **Dobbiamo prendere in considerazione il tempo di accesso medio per ogni processo quando abbiamo il TLB.**

Ricordiamo la formula vista precedentemente senza TLB:

$$\begin{aligned}\text{Effective memory access time} = & pr_1 \times 2 \times t_{\text{mem}} \\ & + (1 - pr_1) \times (t_{\text{mem}} + t_{\text{pfh}} + 2 \times t_{\text{mem}})\end{aligned}$$

Mentre se consideriamo anche il TLB avremo la seguente formula

Effective memory access time =

$$\begin{aligned}& pr_2 \times (t_{\text{TLB}} + t_{\text{mem}}) + (pr_1 - pr_2) \times (t_{\text{TLB}} + 2 \times t_{\text{mem}}) \quad (12.3) \\ & + (1 - pr_1) \times (t_{\text{TLB}} + t_{\text{mem}} + t_{\text{pfh}} + t_{\text{TLB}} + 2 \times t_{\text{mem}})\end{aligned}$$

pr_1 probability that a page exists in memory

pr_2 probability that a page entry exists in TLB

t_{mem} memory access time

t_{TLB} access time of TLB

t_{pfh} time overhead of page fault handling

TLB hit ratio

- Sono usati alcuni meccanismi per migliorare le prestazioni:
 - Le entrate wired TLB per le pagine del kernel: mai sostituite
 - Le superpagine
-

SUPPORTO PER LA SOSTITUZIONE DELLE PAGINE

Dobbiamo fare in modo di contenere il numero di page fault e il gestore della memoria quando applica l'algoritmo di sostituzione delle pagine deve sfruttare la località di riferimento. La località per poter basarsi su quei concetti ha bisogni di info che bisogna memorizzare per far lavorare l'algoritmo.

Le info da memorizzare sono:

- a. L'istante in cui una pagina è stata usata l'ultima volta in questo modo ci sono alcuni algoritmi (LRU) che decidono quale pagina rimuovere. Se voglio eliminare l'istante temporale si necessita di un elevato numero di bit ed è quindi costoso e si preferisce usare un singolo bit per indicare se quella pagina è stata riferita o meno.
 - b. Se una pagina è dirty. L'info è il bit modified presente all'entrata della tabella delle pagine.
-

ORGANIZZAZIONE DELLA PT IN PRATICA

Nel momento in cui si usa il TLB e il gestore si occupa di implementare le istruzioni per aggiornare il TLB, l'MMU quando traduce ha a che fare solo col TLB. Il gestore della memoria virtuale può preoccuparsi di cercare di ottimizzare l'uso della memoria per memorizzare la tabella delle pagine perché in un dato momento abbiamo un elevato numero di processi e significa tante tabelle delle pagine per processo. se lo spazio di indirizzamento logico è elevato significa che le relative tabelle possono occupare memoria per ordine megabyte o giga per ciascun processo e se dobbiamo memorizzare ogni tabella per processo non è una soluzione fattibile, ci sono strategie di

implementazione delle tabelle delle pagine che mirano a contenere la dimensione delle tabelle in memoria.

Una prima soluzione è quella di adottare piuttosto che la tabella delle pagine classica, una **tabella delle pagine invertita** e **tabella delle pagine multilivello**.

- **Tabella delle pagine invertita:** memorizza info inverse rispetto alla tabella delle pagine ordinaria. Invece di descrivere ciascun blocco di memoria virtuale, mantiene info sui frame (pagine fisiche) e per ognuno di esso contiene l'info su quale pagina quel frame memorizza. La dimensione della tabella invertita non dipende più dal numero di processi e dalla dimensione della pagina ma dalla dimensione della memoria della macchina (non interessa quanti processi ci sono perché teniamo traccia dei frame della memoria principale). La tabella contiene la coppia (id programma, #pagina): per ciascun processo abbiamo quale numero di pagina contenuto da quel frame. Il limite è che le info su una pagina devono essere cercate al contrario della tabella classica
- **Tabella delle pagine multilivello:** nel momento in cui si ha una tabella delle pagine molto grande, si va a paginare anche la tabella stessa. Nel sistema ho due tipi di pagine: pagine della tabella delle pagine e pagine del processo. Facendo in questo modo si limita la quantità di memoria necessaria per gestire la tabella delle pagine perché mantengo, ad esempio in un'organizzazione a due livelli, una tabella di primo livello di dimensioni ridotte che contiene le info sulla pagina della tabella delle pagine che contiene le info sulla pagina del processo e le altre possono essere tenute su disco riducendo la quantità di memoria.

- In ambo gli approcci è usato il TLB per ridurre i riferimenti alla memoria durante la traduzione degli indirizzi

TABELLA DELLE PAGINE INVERTITA

Ogni entrata della tabella delle pagine invertita è costituito dalla coppia (processo, pagina a cui si fa riferimento). Quando c'è un indirizzo logico si costituisce l'indirizzo (p_i, b_i) poi si preleva l'id del processo e insieme al numero di pagina lo si va a cercare all'interno della tabella delle pagine invertita. Se non c'è la pagina in memoria si genera un page fault.

La chiave per un funzionamento efficiente è quella di ottimizzare la ricerca: si può usare un hash table: prendo id e numero di pagina, li concateno considerandoli un numero intero, applico una funzione di hashing che restituisce un indice che fa riferimento alla tabella delle pagine. Nella tabella di hash si può avere la collisione e per gestirla si utilizza l'hashing con concatenazione: si aggiunge un campo puntatore in modo che quando si inserisce nella tabella le info con i relativi processi, se si dovesse avere un'altra coppia con lo stesso valore di hashing, si crea un link nel campo puntatore che punta all'entrata successiva

- Ogni entrata della IPT è una coppia ordinata (id processo, numero pagina)
- La coppia (R, p_i) nell'entrata f_i -esima indica che il frame f_i è occupato dalla pagina p_i del processo R
- Lo scheduler, selezionato un processo, ne preleva l'id dal PCB e lo invia in un registro della MMU

L'uso di tabelle hash
Velocizza la ricerca

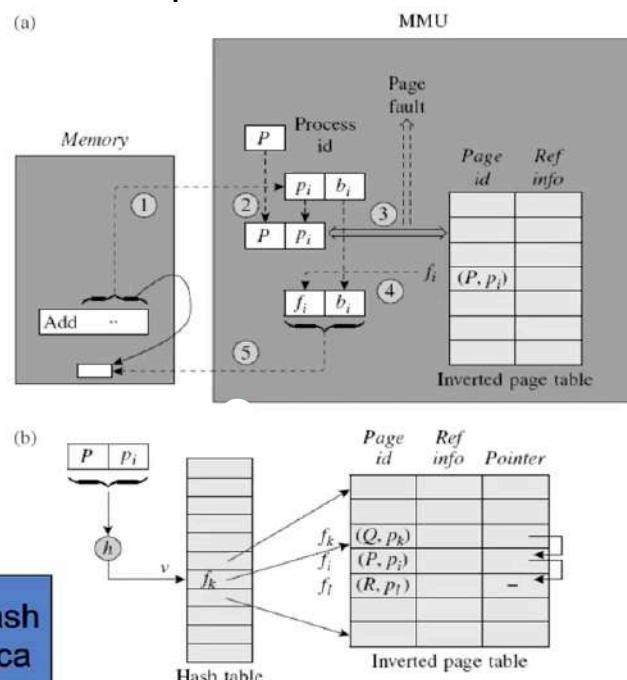


Figure 12.10 Inverted page table: (a) concept; (b) implementation using a hash table.

POLITICHE DI SOSTITUZIONE DELLE PAGINE

Un algoritmo di sostituzione delle pagine deve individuare un frame della memoria che contiene qualche pagina che con probabilità elevata non sarà referenziata nel breve termine, per evitare di incrementare il tasso di page I/O.

Esistono diverse strategie per fare questa scelta.

- Partiamo dalla “strategia di sostituzione delle pagine ottimale”: andiamo a stabilire un criterio che dà con certezza la possibilità di minimizzare il numero di page fault causati dalle pagine scelte dall’algoritmo. Questo tipo di strategia non è implementabile ma è importante usarlo perché funge da riferimento teorico in base al quale possiamo valutare gli algoritmi implementabili.
- Uno di questi è la “strategia di sostituzione FIFO”: individua la pagina da sostituire sulla base di quanto tempo fa è stata caricata, l’algoritmo sostituisce le pagine caricate in memoria per prima.
- Un’altra è la “LRU”: sceglie la pagina da un insieme di pagine non referenziate da molto tempo. Questa strategia è efficace perché sfrutta la località dei riferimenti: quando abbiamo parlato di come si deve individuare una pagina da sostituire si deve sfruttare la località dei riferimenti, sappiamo che gli indirizzi logici relativi all’istruzione corrente, i successivi saranno indirizzi vicini a quest’ultima. Se uso una strategia LRU vado a sostituire una pagina che non si trova probabilmente in una località corrente e soddisfa la proprietà della località dei riferimenti.

- Analizziamo un metodo chiamato “stringhe di riferimento pagina”
 - È una lista di pagine riferite da un processo durante le sue operazioni. La lista è costruita monitorando le operazioni di un processo e forma una sequenza di pagine che le istruzioni e dati utilizzeranno. Per comodità, oltre alla stringa di riferimento, è utile supporre che questa stringa abbia un ordinamento cronologico隐式的: si va a considerare per ogni stringa di riferimento una stringa dei riferimenti temporale t_1, t_2, t_3 dove ciascuno sta a indicare l'istante temporale in cui è avvenuto il riferimento a una specifica pagina all'interno della stringa dei riferimenti.

ESEMPIO: STRINGA DI RIFERIMENTO PAGINA

Supponiamo di avere un computer che supporta istruzioni a 4 byte. Ogni pagina abbia 1kb di dimensione e all'interno di un programma i simboli A e B siano riferiti alla pagina 2 e 5.

Date queste assunzioni vediamo qual è la stringa dei riferimenti di pagina associata a questo programma.

La prima istruzione ci dice che il programma parte all'indirizzo 2040. Se ciascuna istruzione richiede 4byte allora la successiva è 2044, 2048 ecc...

Le pagine sono numerate a partire da 0 e sono di 1024 byte (1kb). La pagina zero arriva a 1024, la pagina uno a 2048 ecc... questo significa che la seconda istruzione appartiene alla pagina numero 1 poiché si trova a 2040. L'istruzione due contiene B che è la pagina 5. L'istruzione successiva è 2044 (ancora pagina 1) ma fa

riferimento al dato A che fa riferimento alla pagina 2. L'istruzione successiva inizia a 2048 (inizio della pagina 2) ma fa riferimento all'operando B (pagina 5) la successiva istruzione è ancora nella pagina 2 e ha loop che si trova nella pagina 1 e così via...

```
        START  2040
        READ   B
LOOP    MOVER  AREG, A
        SUB    AREG, B
        BC    LT, LOOP
        ...
        STOP
A      DS     2500
B      DS     1
END
```

Stringa riferimento pagina 1, 5, 1, 2, 2, 5, 2, 1

Stringa riferimento temporale $t_1, t_2, t_3, t_4, t_5, t_6, t_7, t_8, \dots$

SOSTITUZIONE OTTIMALE

Una volta visto come è definita una stringa dei riferimenti, vediamo gli algoritmi di sostituzione di pagina.

- Il primo algoritmo che discutiamo è l'algoritmo di sostituzione ottimale.

Dobbiamo scegliere una pagina da sostituire in maniera tale che alla lunga questa ci minimizzi il numero di page fault. Significa che valuteremo tutte le alternative e sceglieremo la sequenza tale per cui qualsiasi altra sequenza non può portare a un numero inferiore di page fault. Per fare questa scelta ottimale, la strategia deve considerare tutte le possibili alternative e analizzare le implicazioni della scelta sui page fault futuri che si possono verificare → ecco perché questa strategia non è implementabile: il gestore della memoria virtuale non è in grado di sapere i

comportamenti futuri dei vari processi e che tipo di referenziazione faranno.

È utile però perché ci dà un riferimento rispetto al quale si possono fare confronti con altre strategie di sostituzione.

Nel 1966 è stato dimostrato che la sostituzione ottimale è equivalente a una scelta di questo tipo: data la stringa dei riferimenti si tratta di andare a scegliere la pagina da sostituire tra quella che nel riferimento successivo all'interno della stringa dei riferimenti è quella che viene riferita più tardi. Quando vedremo come potrebbe funzionare applichiamo questa regola (mi trovo nella stringa dei riferimenti, quale sarà la pagina da andare a sostituire? Quella che compare più tardi all'interno della stringa).

SOSTITUZIONE FIFO

La strategia FIFO considera la pagina caricata in memoria da più tempo. Per fare questo il gestore della memoria deve marcare temporalmente ciascuna pagina quando viene caricata in memoria e si utilizza il campo *ref info* della tabella delle pagine e ogni volta che carica una pagina la prima volta la marca temporalmente.

SOSTITUZIONE LRU

Questa strategia sfrutta la località dei riferimenti. Quando si verifica un page fault devo sostituire la pagina usata meno recentemente e questo soddisfa il principio della località dei riferimenti. L'entrata della tabella delle pagine deve essere anche qui marcata temporalmente con l'istante dell'ultimo riferimento temporale. Questo istante è inizializzato quando la pagina è carica per la prima volta in memoria e aggiornata a ogni riferimento fatto alla pagina.

ESEMPIO: SOSTITUZIONE OTTIMALE, FIFO , LRU

- Consideriamo le seguenti stringhe di riferimento pagina e stringhe di riferimento temporale per un processo P

Stringa riferimento pagina 0, 1, 0, 2, 0, 1, 2, ...
 Stringa riferimento temporale $t_1, t_2, t_3, t_4, t_5, t_6, t_7, \dots$

- Vediamo il funzionamento delle diverse strategie di sostituzione pagine con $alloc = 2$

- alloc rappresenta il numero di frame di pagina allocati al processo P

Abbiamo una stringa di riferimento pagina e la stringa di riferimento temporale per un processo P.

Vediamo come si comportano questi algoritmi supponendo che in ogni istante il gestore può allocare $alloc=2$ (a ogni istante ai processi vengono allocati 2 frame).

		Optimal			FIFO			LRU			
Time instant	Page ref	Valid bit	Ref info	Replace-	Valid bit	Ref info	Replace-	Valid bit	Ref info	Replace-	
t_1	0	0	1	-	0	1	t_1	-	0	1	t_1
		1	0		1	0		1	0		
		2	0		2	0		2	0		
t_2	1	0	1	-	0	1	t_1	-	0	1	t_1
		1	1		1	1	t_2	-	1	1	t_2
		2	0		2	0		2	0		
t_3	0	0	1	-	0	1	t_1	-	0	1	t_1
		1	1		1	1	t_2	-	1	1	t_2
		2	0		2	0		2	0		
t_4	2	0	1	Replace 1 by 2	0	0		Replace 0 by 2	0	1	t_3
		1	0		1	1	t_2		1	0	
		2	1		2	1	t_4		2	1	t_4
t_5	0	0	1	-	0	1	t_5	Replace 1 by 0	0	1	t_5
		1	0		1	0		1	0		
		2	1		2	1	t_4		2	1	t_4
t_6	1	0	0	Replace 0 by 1	0	1	t_5	Replace 2 by 1	0	1	t_5
		1	1		1	1	t_6		1	1	t_6
		2	1		2	0			2	0	
t_7	2	0	0	-	0	0		Replace 0 by 2	0	0	
		1	1		1	1	t_6		1	1	t_6
		2	1		2	1	t_7		2	1	t_7

In alto vediamo i risultati dei tre algoritmi. Discutiamone.

- **Algoritmo ottimale**

Al tempo t1 viene referenziata la pagina zero. Si carica la pagina zero in memoria e nella tabella delle pagine avremo che il bit di validità è impostato a 1 ad indicare che la pagina è in memoria e il campo ref non ha nessuna info associata.

All'istante temporale t2 abbiamo un altro riferimento alla pagina 1. Anche questa pagina verrà caricata in memoria, il bit di validità è impostato a 1 a indicare che la pagina 1 è in memoria. All'istante t3 viene riferita la pagina 0 che già si trova in memoria e non succede nulla. Al tempo t4 viene referenziata la pagina 2: nella pagina 2 accade che abbiamo 2 frame già allocati e uno dei due deve essere sostituito e dobbiamo rimuovere una delle pagine dal frame corrispondente in memoria: quale pagina andiamo a sostituire? Andiamo a sostituire la pagina 1 perché se vediamo la stringa dei riferimenti c'è la pagina 1 che viene referenziata più tardi. A t6 viene referenziata la pagina 1 e dobbiamo fare spazio: la pagina che viene referenziata più tardi è zero, per cui si sostituisce 0 con 2.

- **Algoritmo FIFO**

Nel caso fifo e nell'LRU inizialmente è uguale, tranne per l'uso di ref info. Al tempo 1 viene caricata la pagina 0 e viene impostato anche il tempo t1 nel campo ref info. Al tempo t2 viene riferita la pagina 1, viene caricata e viene marcato il campo. Al tempo t3 viene riferita la pagina zero e non succede nulla. Al tempo t4 viene riferita la pagina 2: deve avvenire una sostituzione di pagina e la strategia FIFO va a sostituire la pagina con quella caricata per prima, quindi la pagina zero per cui si sostituisce 0 con 2 e si aggiorna il campo ref info. Al tempo t5 viene riferito 0 che non è in

memoria, scambio 0 con 1 perché è stata caricata prima tra 1 e 2 e aggiorno il campo ref info. Al tempo t6 si fa riferimento alla pagina 1 che non è in memoria e va sostituita con la pagina 2. Al tempo t7 viene riferita la pagina 2 e viene sostituita con la pagina 0.

- **Algoritmo LRU**

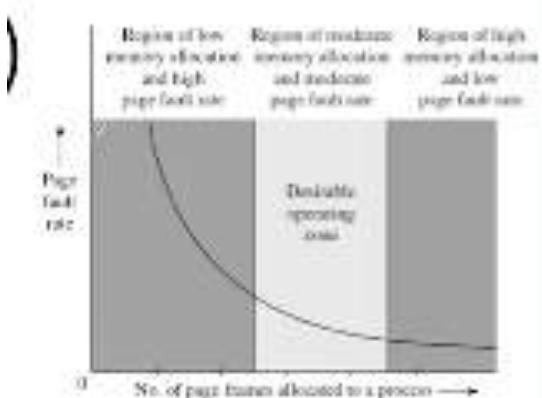
I primi passi sono gli stessi: a t1 viene caricato 0 e si marca temporalmente, lo stesso al tempo t2 viene caricato 1 e marcato temporalmente, al tempo t3 viene riferita 0 ma già c'è, viene marcata temporalmente con t3 e non succede nulla. Al tempo t4 viene riferita la pagina 2: devo effettuare la sostituzione e l'algoritmo LRU sceglie la pagina riferita da più tempo cioè la pagina 1. Si sostituisce 1 con 2 e marchiamo temporalmente con t4. Al tempo t5 c'è riferimento alla pagina 0 che già è in memoria e dobbiamo solamente aggiornare il campo ref info a t5. Al tempo t6 è riferita la pagina 1 che non è in memoria e andrà sostituita con la pagina 2 che è stata riferita meno recentemente e aggiorniamo il campo ref info. Al tempo t7 viene fatto riferimento alla pagina 2, si scambia con la pagina 0 e concludiamo.

- Quale funziona meglio?

- Se assumiamo che ogni primo caricamento avvenga generando un page fault, bisogna contare il page fault ogni volta che viene caricato 0 e 1 e i relativi page fault.
 - Per la strategia ottimale abbiamo 4 page fault.
 - Per la strategia FIFO abbiamo 6 page fault.
 - Per la strategia LRU abbiamo 5 page fault.

- In definitiva conviene ovviamente la strategia ottimale, tra FIFO e LRU risulta migliore la LRU.
 - La strategia FIFO seleziona la pagina da sostituire sulla base del caricamento in memoria più vecchio. Questa strategia sicuramente non è desiderabile perché l'ultimo caricamento potrebbe essere anche caricato da più tempo ma ciò non significa che nel breve non possa essere referenziata, soprattutto se si trova nella località corrente dell'indirizzo e non tenendo conto di ciò è una strategia che porta a lungo tempo più page fault rispetto all'LRU.
 - LRU tiene conto degli ultimi riferimenti delle pagine e tenendo conto di ciò risulta migliore rispetto alla strategia FIFO.
-

POLITICHE DI SOSTITUZIONE PAGINA



In generale, quando abbiamo parlato del fenomeno del **trashing** che ha che fare del numero di page fault che si hanno quando si fanno riferimento alle pagine di un processo. Abbiamo detto che il numero di page fault dipende da quanti frame andiamo ad allocare a ciascun processo, se si allocano pochi frame, il processo fa riferimento a diverse pagine e avrà poche pagine in memoria causando diversi page fault degradando le prestazioni del sistema. Se allochiamo più frame, il numero di page fault

diminuisce ma allocare tanti frame a un processo significa che riduciamo il sistema ad avere più processi simultanei riducendo le sue prestazioni. C'è una confort zone che è il giusto compromesso tra frame allocati e page fault che si creano. Detto ciò, purché una strategia della pagina possa ricadere nella zona desiderabile, deve godere della **proprietà dello stack**. Per introdurre questa proprietà si introduce la seguente notazione

Una politica di sostituzione di pagina possiede la **proprietà dello stack** se $\{pi\}_n^k \subseteq \{pi\}_m^k$ per tutti gli n, m tali che $n < m$

dove $\{pi\}_n^k$ indica l'insieme delle pagine in memoria al tempo t_k^+ se $\text{alloc}_i = n$ durante l'intera attività del processo P_i (t_k^+ implica l'istante dopo t_k ma prima di t_{k+1})

L'insieme delle pagine memoria al tempo t_k^+ considerando che al processo alloco n frame durante la sua attività.

Diciamo che una politica di sostituzione delle pagine soddisfa la proprietà dello stack se il numero di pagine all'istante $k +$ per il processo pi quando gli si alloca n frame devono essere incluse nelle pagine che gli sono allocate nello stesso istante temporale se allo stesso processo gli si allocano m frame dove $m > n$.

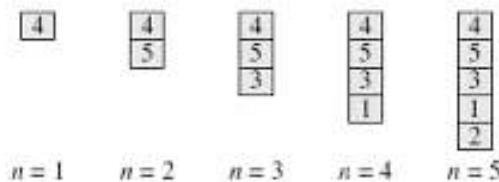
Significa che se considero l'esecuzione del processo pi nei vari istanti e ogni istante t_k^+ , le pagine presenti in memoria in n frame devono essere contenute nello stesso istante nell'insieme delle pagine se gli allocassi m frame, se è vero quella politica di sostituzione delle pagine gode della proprietà dello stack.

Consideriamo l'esempio:

abbiamo il processo nel quale ci sono le pagine allocate per diversi valori di alloc. Supponiamo di essere all'istante k , la pagina che è allocata in $n = 1$ è inclusa nella pagina quando è $n = 2$, così queste sono allocate quando $n = 3$, e così via...

Una politica di sostituzione di pagina possiede la **proprietà dello stack** se $\{pi\}_n^k \subseteq \{pi\}_m^k$ per tutti gli n, m tali che $n < m$

dove $\{pi\}_n^k$ indica l'insieme delle pagine in memoria al tempo t_k^+ se $\text{alloc}_i = n$ durante l'intera attività del processo P_i (t_k^+ implica l'istante dopo t_k ma prima di t_{k+1})



PROPRIETA' DELLO STACK

Perché la proprietà dello stack fa sì che se l'algoritmo di sostituzione la soddisfa non incrementa i page fault.

Supponiamo di avere due esecuzioni del processo P_i dove consideriamo due esecuzioni diverse: una con il numero di frame allocati pari a n e un altro pari a m con $m > n$. Se la politica soddisfa la proprietà dello stack significa che fissato un istante temporale tutte le pagine presenti quando il numero di frame era n sono contenute quando il numero di frame allocati è m .

Quando eseguo m frame significa che ci saranno $m - n$ pagine che possono essere riferite o meno durante l'esecuzione. Se sono riferite quando c'è l'esecuzione con m trovo la pagina in memoria ma se siamo in n un riferimento a esse genera un page fault.

- All'aumentare del numero di frame allocati non possono aumentare i page fault, possono al più essere costanti.

PROBLEMI CON POLITICA FIFO

Vediamo il comportamento rispetto alla proprietà dello stack della strategia FIFO.

Supponiamo di avere una stringa di riferimento e di avere 2 diverse esecuzioni per la stessa esecuzione quando abbiamo alloc=3 e alloc=4.

Abbiamo il processo Pi per il quale viene usato una strategia FIFO.

Vediamo lo schema:

- **FIFO**

la prima volta viene riferita la pagina 5 e viene caricata in memoria. Carichiamo poi la pagina 4 e la 3. Abbiamo poi la pagina 2, non c'è spazio e sostituiamo 2 con 5 (è stata caricata per prima) poi abbiamo la pagina 1 che viene sostituita con 4 (caricata da più tempo), abbiamo poi la pagina 4 sostituita, la pagina 2 e così via... I numeri in basso con l'asterisco sono le pagine con page fault e sostituzione di pagina.

Stesso algoritmo con 4 frame allocati.

- **LRU**

Carico la prima pagina, poi la seconda e la terza in memoria. Quando referenzio la 2 bisogna farle spazio andando a sostituire la pagina referenziata da più tempo cioè la pagina 5 e sostituisco 2 con 5. Abbiamo la pagina 1 che viene sostituita con la pagina 4. Si prosegue con alloc=4.

Osserviamo che



la strategia FIFO non possiede la proprietà dello stack

Page reference string 5, 4, 3, 2, 1, 4, 3, 5, 4, 3, 2, 1, 5, ...

Reference time string $t_1, t_2, t_3, t_4, t_5, t_6, t_7, t_8, t_9, t_{10}, t_{11}, t_{12}, t_{13}$,

FIFO	$alloc_i = 3$	<table border="1"> <tbody> <tr><td></td><td></td><td>3</td><td>3</td><td>3</td><td>4</td><td>4</td><td>4</td><td>4</td><td>4</td><td>2</td><td>2</td><td>2</td></tr> <tr><td></td><td></td><td>4</td><td>4</td><td>4</td><td>1</td><td>1</td><td>1</td><td>5</td><td>5</td><td>5</td><td>5</td><td>5</td></tr> <tr><td>5</td><td>5</td><td>5</td><td>2</td><td>2</td><td>2</td><td>3</td><td>3</td><td>3</td><td>3</td><td>1</td><td>1</td><td>1</td></tr> <tr><td>5</td><td>4*</td><td>3*</td><td>2*</td><td>1*</td><td>4*</td><td>3*</td><td>5*</td><td>4</td><td>3</td><td>2*</td><td>1*</td><td>5</td></tr> </tbody> </table>			3	3	3	4	4	4	4	4	2	2	2			4	4	4	1	1	1	5	5	5	5	5	5	5	5	2	2	2	3	3	3	3	1	1	1	5	4*	3*	2*	1*	4*	3*	5*	4	3	2*	1*	5
		3	3	3	4	4	4	4	4	2	2	2																																										
		4	4	4	1	1	1	5	5	5	5	5																																										
5	5	5	2	2	2	3	3	3	3	1	1	1																																										
5	4*	3*	2*	1*	4*	3*	5*	4	3	2*	1*	5																																										
	$alloc_i = 4$	<table border="1"> <tbody> <tr><td></td><td></td><td>2</td><td>2</td><td>2</td><td>2</td><td>2</td><td>2</td><td>3</td><td>3</td><td>3</td><td>3</td><td>3</td></tr> <tr><td></td><td></td><td>3</td><td>3</td><td>3</td><td>3</td><td>3</td><td>4</td><td>4</td><td>4</td><td>4</td><td>4</td><td>5</td></tr> <tr><td>5</td><td>5</td><td>5</td><td>5</td><td>1</td><td>1</td><td>1</td><td>5</td><td>5</td><td>5</td><td>1</td><td>1</td><td>1</td></tr> <tr><td>5</td><td>4*</td><td>3*</td><td>2*</td><td>1*</td><td>4</td><td>3</td><td>5*</td><td>4*</td><td>3*</td><td>2*</td><td>1*</td><td>5*</td></tr> </tbody> </table>			2	2	2	2	2	2	3	3	3	3	3			3	3	3	3	3	4	4	4	4	4	5	5	5	5	5	1	1	1	5	5	5	1	1	1	5	4*	3*	2*	1*	4	3	5*	4*	3*	2*	1*	5*
		2	2	2	2	2	2	3	3	3	3	3																																										
		3	3	3	3	3	4	4	4	4	4	5																																										
5	5	5	5	1	1	1	5	5	5	1	1	1																																										
5	4*	3*	2*	1*	4	3	5*	4*	3*	2*	1*	5*																																										
	Time line																																																					
LRU	$alloc_i = 3$	<table border="1"> <tbody> <tr><td></td><td></td><td>3</td><td>3</td><td>3</td><td>4</td><td>4</td><td>4</td><td>4</td><td>4</td><td>1</td><td>1</td><td>1</td></tr> <tr><td></td><td></td><td>4</td><td>4</td><td>4</td><td>1</td><td>1</td><td>1</td><td>5</td><td>5</td><td>2</td><td>2</td><td>2</td></tr> <tr><td>5</td><td>5</td><td>5</td><td>2</td><td>2</td><td>2</td><td>3</td><td>3</td><td>3</td><td>3</td><td>3</td><td>3</td><td>5</td></tr> <tr><td>5</td><td>4*</td><td>3*</td><td>2*</td><td>1*</td><td>4*</td><td>3*</td><td>5*</td><td>4</td><td>3</td><td>2*</td><td>1*</td><td>5*</td></tr> </tbody> </table>			3	3	3	4	4	4	4	4	1	1	1			4	4	4	1	1	1	5	5	2	2	2	5	5	5	2	2	2	3	3	3	3	3	3	5	5	4*	3*	2*	1*	4*	3*	5*	4	3	2*	1*	5*
		3	3	3	4	4	4	4	4	1	1	1																																										
		4	4	4	1	1	1	5	5	2	2	2																																										
5	5	5	2	2	2	3	3	3	3	3	3	5																																										
5	4*	3*	2*	1*	4*	3*	5*	4	3	2*	1*	5*																																										
	$alloc_i = 4$	<table border="1"> <tbody> <tr><td></td><td></td><td>2</td><td>2</td><td>2</td><td>2</td><td>2</td><td>5</td><td>5</td><td>5</td><td>1</td><td>1</td><td>1</td></tr> <tr><td></td><td></td><td>3</td><td>3</td><td>3</td><td>3</td><td>3</td><td>3</td><td>3</td><td>3</td><td>3</td><td>3</td><td>3</td></tr> <tr><td>5</td><td>5</td><td>5</td><td>5</td><td>1</td><td>1</td><td>1</td><td>1</td><td>1</td><td>1</td><td>2</td><td>2</td><td>2</td></tr> <tr><td>5</td><td>4*</td><td>3*</td><td>2*</td><td>1*</td><td>4</td><td>3</td><td>5*</td><td>4</td><td>3</td><td>2*</td><td>1*</td><td>5*</td></tr> </tbody> </table>			2	2	2	2	2	5	5	5	1	1	1			3	3	3	3	3	3	3	3	3	3	3	5	5	5	5	1	1	1	1	1	1	2	2	2	5	4*	3*	2*	1*	4	3	5*	4	3	2*	1*	5*
		2	2	2	2	2	5	5	5	1	1	1																																										
		3	3	3	3	3	3	3	3	3	3	3																																										
5	5	5	5	1	1	1	1	1	1	2	2	2																																										
5	4*	3*	2*	1*	4	3	5*	4	3	2*	1*	5*																																										
	Time line																																																					

POLITICHE DI SOSTITUZIONE DELLE PAGINE

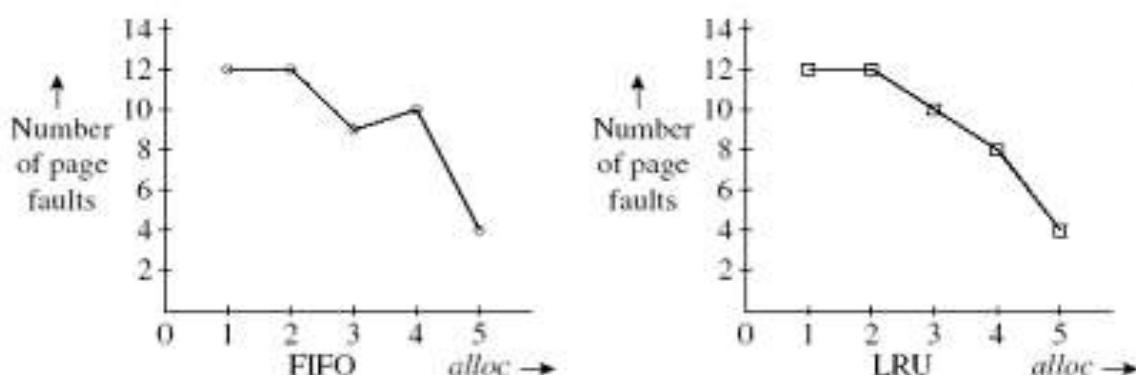


Figure 12.18 (a) Belady's anomaly in FIFO page replacement; (b) page fault characteristic for LRU page replacement.

Se visualizziamo il numero di page fault in funzione del numero di frame allocati per le due strategie ci accorgiamo graficamente che la strategia FIFO ha problemi: se aumento il numero di frame

aumenta comunque il numero di page fault, al contrario della strategia LRU che all'aumentare del numero di frame calano i page fault.

Questo fenomeno della strategia FIFO va sotto il nome di **“anomalia di belady”**.

POLITICHE DI SOSTITUZIONE PAGINA IN PRATICA

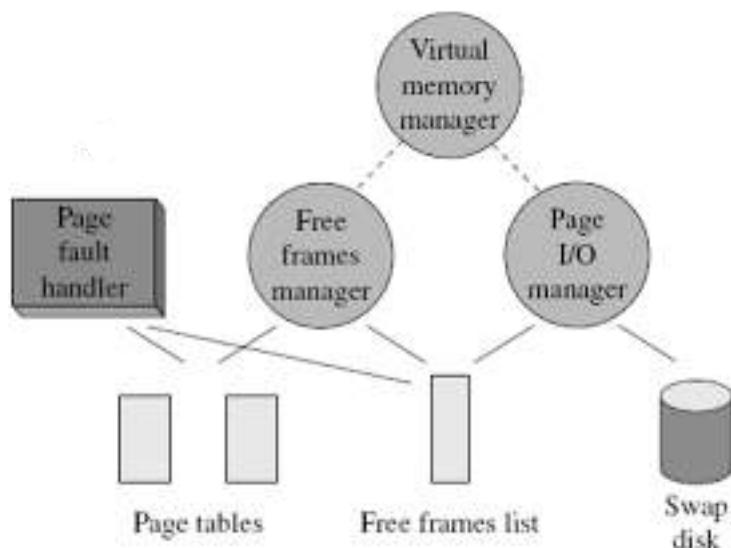


Figure 12.19 Page replacement in practice.

Vediamo come è organizzato praticamente il gestore della memoria virtuale.

Il gestore della memoria gestisce una lista di frame liberi e a ogni istante cerca di mantenere questa lista un insieme ridotto di frame.

Il gestore è costituito da 2 thread demoni(sono sempre in esecuzione in background) : un thread è il gestore dei frame liberi e l'altro è il gestore di I/O di pagine.

Il gestore dei frame liberi viene attivato ogni volta che la lista dei frame liberi contiene un numero di frame che va sotto una soglia definita dal gestore in fase di booting del sistema. Quando il gestore nota che la lista contiene un numero di frame liberi molto piccolo, attiva il gestore dei frame liberi che va a scandire le

pagine dei processi presenti in memoria alla ricerca di qualche pagina che possa essere rimossa per riempire la lista di frame liberi.

La cosa interessante è che si può evitare di fare il page-in: il gestore dei frame liberi sceglie una pagina da sostituire e in più segna all'interno della lista di frame se il frame contiene una pagina modificata di recente aggiornando il dirty bit.

A questo punto il gestore dei frame liberi si può mettere in uno stato di sleep e attivarsi successivamente.

Quando è necessario fare una sostituzione di pagina, il gestore dell'I/O di pagina va ad eseguire le operazioni di page-out: esamina la lista dei frame liberi, va a vedere i frame con dirty bit pari a 1 e per questi fa il page-out memorizzando la pagina contenuta in quel frame sul disco.

Il gestore dei page fault funziona come un qualsiasi componente del kernel che è guidato da eventi: si attiva quando c'è l'interrupt generato dall'MMU quando c'è un page fault e va a verificare se la pagina che deve essere caricata in memoria si trova ancora all'interno dei frame liberi liberati dalle operazioni precedenti, fa questo perché quando si rimuove una pagina dai frame la si sovrascrive quindi il gestore dei frame liberi seleziona alcune pagine e mette i frame nella lista dei frame liberi, l'I/O di pagina esegue il page-out dove necessario. Il gestore dei page fault non fa altro che modificare il bit di validità della pagina da 0 a 1 e rimuove il frame dalla lista dei frame liberi.

➤ Vediamo nella partica quali problemi sorgono

La strategia FIFO va scartata a prescindere perché non gode della proprietà dello stack e non si implementa.

Si può implementare la strategia LRU ma in ogni caso anche questa non è praticabile perché la LRU prevede che a ogni riferimento di pagina il campo *ref info* della tabella delle pagine deve essere aggiornato con l'istante temporale. Per memorizzare ogni riferimento d'istante temporale occorre un numero elevato di bit e non c'è la possibilità di memorizzare nel campo *ref info* gli istanti temporali, non essendo possibile ciò non è possibile implementare la LRU così come è stata descritta fino ad ora. La maggior parte degli HW disponibili per il campo *ref info* mettono a disposizione addirittura 1 solo bit e con questo dobbiamo realizzare le strategie di sostituzione di pagina che devono approssimare la strategia LRU perché è la migliore tra le fattibili.

Si implementa quindi un'approssimazione dell'LRU nella quale il campo *ref info* non contiene l'istante temporale del riferimento ma contiene un solo bit che sta a indicare se quella pagina è stata riferita (1) o meno (0). Le strategie che usano questo singolo bit prendono il nome di “**NRU**” (Not Recently Used).

La strategia più semplice prevede che nel momento in cui una pagina viene caricata per la prima volta in memoria, questo bit viene inizializzato a zero e nel momento in cui c'è da sostituire una pagina va alla ricerca di pagine il cui bit *ref info* è zero. Se in un dato momento, il bit è 1 per tutte le pagine va a resettare tutti i bit e ne sceglie una da sostituire. Questa strategia è semplice da implementare ma non discrimina bene le pagine referenziate da più o meno tempo. Il campo *ref info* ci dice che la pagina è stata referenziata però quando tutte sono referenziate ne sceglie una random e potrebbe scegliere una pagina che non è detto sia stata referenziata nel tempo più lontano rispetto a qualche altra pagina, si può fare quindi meglio.

Esistono infatti delle varianti NRU che prendono il nome di “**algoritmi di clock**”, che consentono una migliore

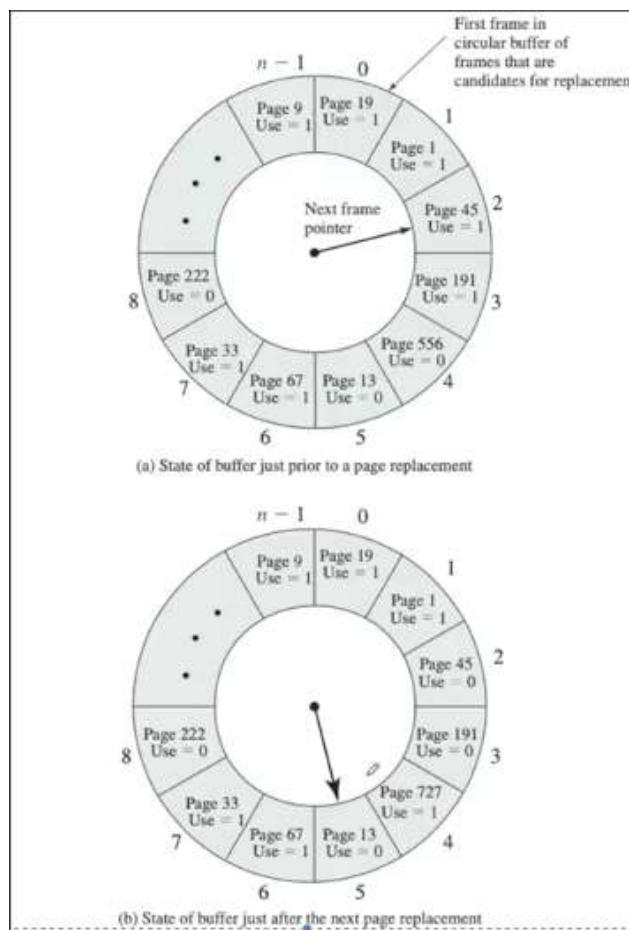
discriminazione tra le pagine perché resettano il bit di ref info ma lo fanno periodicamente e la periodicità ha a che fare col tempo rispetto al quale sono state riferite dando la possibilità di sostituire pagine che probabilisticamente riferite da più tempo. Si chiama clock perché la lista dei frame liberi è una lista circolare (orologio) e la lancetta è un puntatore.

Ci sono due varianti:

1- Algoritmo di clock a una lancetta.

2- Algoritmo di clock a due lancette: ha due puntatori (Reset e Controllo).

CLOCK A UNA LANCETTA

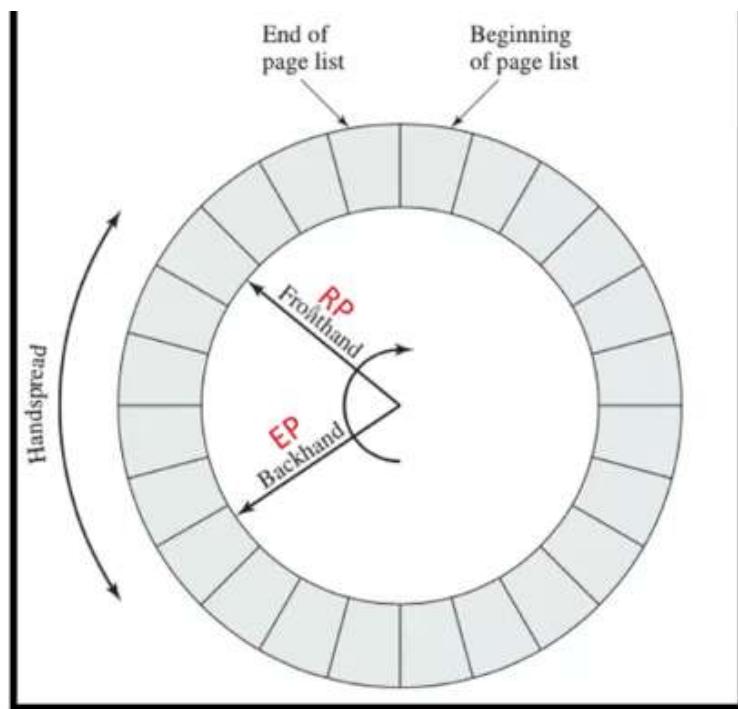


Nello schema dove leggiamo “use” significa ref.

Se devo scegliere la pagina da sostituire seguo il puntatore e vedo ad esempio che un bit ref è 1, ciò significa che è stata riferita

recentemente e quindi salto e vado avanti. Supponiamo che la pagina riferita sia la 727, vede che il bit è 0 per cui questa pagina viene rimossa e all'interno viene caricata la pagina 727. Il bit viene impostato a 1 e il puntatore continua ad avanzare. Quando tutti i bit ref sono uguali a 1 l'algoritmo clock degenera in una strategia FIFO perché scorre tutta la lista e ritorna la primo frame che sarà quello da rimuovere.

ALGORITMO DI CLOCK A DUE LANCETTE



In questo caso abbiamo due lancette, un puntatore di reset e un puntatore di controllo che esamina i vari frame all'interno della lista e questi puntatori hanno una distanza predeterminata dall'algoritmo. Ogni volta che si esamina un frame, si sposta il puntatore e se sposto uno anche l'altro si sposta. Succede che il puntatore di controllo valuta il frame: se il bit ref è zero prende la pagina e la sostituisce, se è 1 va avanti. Il bit di reset quando si sposta finisce su di un frame: se il frame aveva un bit ref 1 lo resetta, se è 0 lo lascia inalterato.

ESEMPIO: ALGORITMO DI CLOCK A DUE LANCETTE

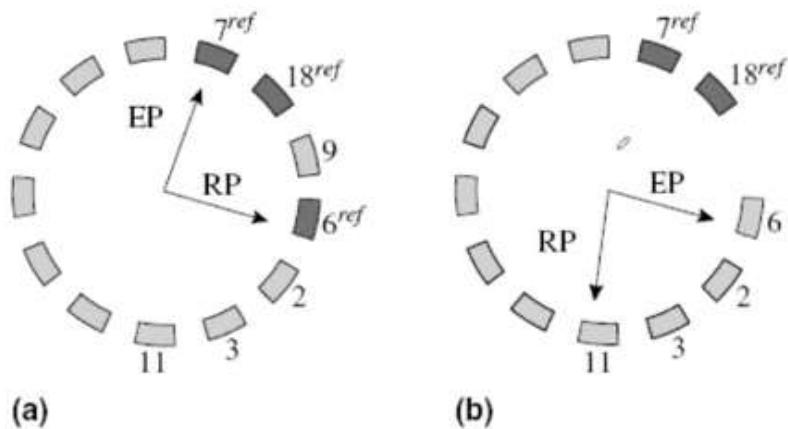


Figure 12.20 Operation of the two-handed clock algorithm.

Abbiamo i frame grigio chiaro che sono frame con bit ref 0 mentre quelli scuri hanno ref 1. Supponiamo che dobbiamo liberare un frame: EP esamina il frame a cui punta, vede che è 1 e quindi avanza. Simultaneamente il bit di reset fa diventare il bit che punta a 0. Vanno avanti insieme. Il bit ref di EP è ancora 1 e non lo tocca e lo stesso per il bit di reset che va avanti. Ora EP punta a 9 che ha bit 0 e la pagina viene sostituita e i due puntatori avanzano. Arriviamo ad EP su 6 e RP su 11. EP sostituisce 6 e RP non fa nulla e si spostano in avanti. Il senso è che nel momento in cui il puntatore di reset resetta il bit ref, la distanza tra i due puntatori determina anche dopo quanto tempo EP va a consultare quel frame per verificare se sostituirlo o meno. Questo significa che più la distanza tra i puntatori è breve, più si fa in modo che in memoria ci siano quei frame che sono stati riferiti da più di recente perché anche se ho resettato quel frame potrebbe essere nuovamente riferito. Se la distanza aumenta esamino quel frame dopo più tempo e rimuovo le pagine rimosse da più tempo. Il parametro essenziale è la distanza tra i puntatori.

CONTROLLARE L'ALLOCAZIONE DI MEMORIA AD UN PROCESSO

Sulla base di queste strategie so quale frame liberare ma ancora non si sa quanti frame devo allocare per processo. In che modo impostiamo il valore di $alloc_i$. Certamente l'algoritmo deve avere la proprietà dello stack. Ci sono due diverse possibilità.

- Allocazione di memoria statica: per ogni processo si alloca un certo numero di frame. Il criterio è alcune caratteristiche del processo. Deciso quanti frame da allocare, questa è irreversibile e la strategia di sostituzione delle pagine è locale: scelgo le pagine da sostituire tra le pagine del processo. Questo tipo di allocazione soffre di tutti i problemi delle strategie statiche: se sbaglio potrei allocare un numero di frame insufficiente e viceversa.
- Allocazione di memoria variabile: imposto il valore di alloc in maniera dinamica sulla base dei comportamenti dei processi. Le sostituzione di pagina possono essere fatte in modo globale e locale.
 - Globale: si sceglie tra i frame da sostituire tutti i frame di tutte le pagine che contengono i processi
 - Locale: si scelgono le pagine tra quelle del processo in questione solamente. Il gestore della memoria deve determinare periodicamente il valore corretto di alloc per un processo. La locale ha prestazioni migliori.

Lo schema locale usa il “**working set**”: è l’insieme di pagine di memoria di un processo che sono state referenziate nelle precedenti Δ istruzioni dove Δ è un parametro. Nello schema

working set l'allocatore alloca un numero di frame pari alla dimensione del working set.

WORKING SET → NOTAZIONE

- Le precedenti Δ istruzioni costituiscono la finestra del working set
- $WS_i(t, \Delta)$ working set P_i al tempo t per la finestra Δ
- $WSS_i(t, \Delta)$ dimensione del working set $WS_i(t, \Delta)$
 - Numero di pagine in $WS_i(t, \Delta)$
- $WSS_i(t, \Delta) \leq \Delta$
 - Una pagina può essere referenziata più di una volta in una finestra di WS
 - Un allocatore di memoria basato sul working set o mantiene l'intero working set in memoria oppure se non può sospendere il processo. Ad ogni istante per un dato processo alloc o viene impostato alla dimensione del suo working set o a zero. Questa strategia assicura buone hit ratio in memoria perché Δ preserva la località dei riferimenti. Il numero di page fault aumenteranno quando Δ passa nella finestra temporale dove c'è stato un passaggio da una località di riferimento a un'altra località. Contenendo il numero di page fault si contiene il fenomeno del thrashing.

WORKING SET: GRADI DI MULTIPROGRAMMAZIONE

Attraverso l'uso del working set, il gestore può aumentare o diminuire il grado di multiprogrammazione del sistema basandosi sulle info derivanti dal working set. In che modo?

- Se ci sono un certo insieme di processi in esecuzione $\{P_k\}$, potrei decidere di abbassare il grado di multiprogrammazione, ciò significa che uno di questi processi in esecuzione lo devo sospendere perché si verifica che la somma della dimensione del working set di tutti i k

processi sia maggiore del numero di frame disponibili in memoria fisica. Se ho più pagine riferite rispetto al numero di frame devo ridurre il grado di multiprogrammazione perché si dovrebbero fare sostituzioni di pagine per liberare frame.

$$\sum_k WSS_k > \#frame$$

- Posso andare a incrementare il grado di multiprogrammazione quando la somma della dimensione dei working set di tutti i k processi è minore di tutti i frame nella memoria fisica

$$\sum_k WSS_k < \#frame$$

Se questo è vero cerco un processo P_g tale per cui il suo working set è minore del numero di frame disponibili – la somma delle dimensioni del working set di k

$$WSS_g \leq (\#frame - \sum_k WSS_k)$$

Il gestore della memoria virtuale tiene traccia per ogni processo sia del valore di $alloc_i$ che la dimensione del working set per gestire l'aumento e la diminuzione del grado di multiprogrammazione.

- Per abbassare il grado di multiprogrammazione il gestore sceglie un processo, P_i , da sospendere
 - Esegue un page-out per ogni pagina modificata di P_i e cambia lo stato dei frame a libero
 - $alloc_i$ è impostato a 0, mentre WSS_i rimane inalterato
- Per incrementare il grado di multiprogrammazione, si ripristina P_i e si pone $alloc_i = WSS_i$
 - Carica la pagina di P_i che contiene la prossima istruzione da eseguire, le altre pagine sono caricate in corrispondenza dei page fault
 - Alternativamente, si caricano tutte le pagine di WS_i , ma ridondanza dei caricamenti possibile

IMPLEMENTAZIONE DEL WORKING SET

L'implementazione del working set non è esattamente quella appena descritta poiché ci si scontra contro difficoltà implementative. Il principio resta lo stesso.

Per ogni processo a ogni istante temporale è difficile valutare esattamente qual è il working set fissato Δ . Piuttosto che farlo ogni istante temporale, lo si fa periodicamente e alla fine di ogni intervallo si valuta il working set di ogni processo e si usano quelle info per determinare i working set dell'intervallo successivo.

Vediamo l'esempio.

Ci sono 60 frame liberi e valutiamo i working set con i frame allocati per ciascun processo per ogni 100 istanti temporali.

La dimensione del WS del processo p1 è 14 e alloc è esattamente 14 frame.

P2 ha 20 WS e gli si allocano 20 frame.

P3 ne ha 18 e gli si allocano.

P4 ne ha 10 e non gli si alloca nulla poiché i frame liberi sono 8.

Dopo un certo intervallo temporale vado a rivalutare le dimensioni del WS.

Vediamo che il WS di p1 è 12.

P2 è 24.

P3 è 19.

P4 è 10 ma anche questa volta non riceverà nulla.

Dopo un altro intervallo la situazione cambia.

Il WS di P1 è 14.

P2 è 11.

P3 è 20

P₄ è 10 e finalmente gli si allocano le pagine necessarie

Process	<i>t</i> ₁₀₀		<i>t</i> ₂₀₀		<i>t</i> ₃₀₀		<i>t</i> ₄₀₀	
	WSS	alloc	WSS	alloc	WSS	alloc	WSS	alloc
<i>P</i> ₁	14	14	12	12	14	14	13	13
<i>P</i> ₂	20	20	24	24	11	11	25	25
<i>P</i> ₃	18	18	19	19	20	20	18	18
<i>P</i> ₄	10	0	10	0	10	10	12	0

Figure 12.21 Operation of a working set memory allocator.

COPY ON WRITE

Abbiamo studiato la fork(): quando un processo invoca una fork si genera un processo figlio che è una copia identica del padre e bisogna fare un exec() per renderlo diverso. Nelle prime versioni, quando la memoria virtuale non era stata inventata, la fork era gestita così: si creava un nuovo processo e il processo figlio era una copia esatta del padre, si copiavano tutte le pagine del processo padre nel processo figlio. Questa era una situazione che poteva portare uno spreco di pagine.

Abbiamo anche detto che la vfork() è obsoleta perché nei sistemi moderni si adotta la tecnica del copy on write: quando genero un processo figlio, le pagine del processo padre non vengono copiate. Inizialmente padre e figlio condividono la stessa copia in memoria delle pagine coinvolte però viene usato un flag indicato con c. quando vado a creare una fork() tutte le pagine del processo padre sono marcate con c che sta a indicare copy on write cioè finchè le pagine non sono toccate, sono condivise da processo padre e processo figlio ma nel momento in cui il processo figlio referenzia una variabile che era del processo padre, succede che la pagina in quel momento viene effettivamente copiata e a quel punto i due processi avranno una copia privata di quella pagina. Il copy on write fa funzionare tutto come prima e fa risparmiare copie non necessarie.

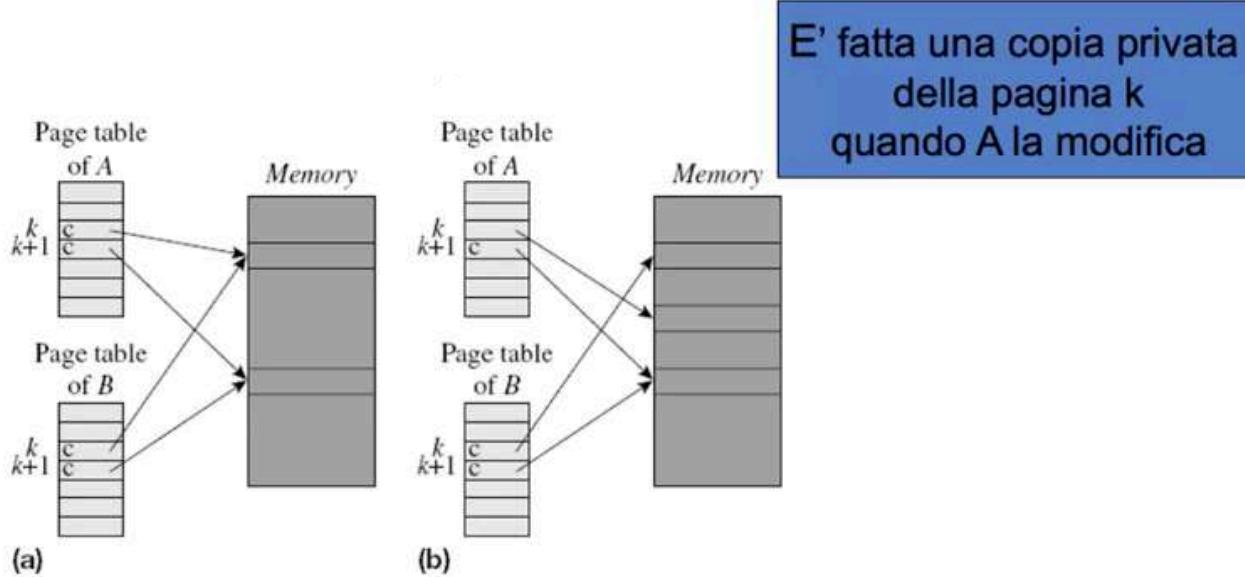


Figure 12.24 Implementing copy-on-write: (a) before and (b) after process A modifies page k .

FILE SYSTEM

La componente del SO con cui gli utenti interagiscono maggiormente: l'insieme dei file con i quali abbiamo modo di lavorare quotidianamente, file di diverso tipo. All'interno di un qualsiasi computer possiamo maneggiare file audio, video, di testo con proprietà diverse ma dal punto di vista dell'utente sono acceduti alla stessa maniera. Il compito del SO è quello di consentire un accesso alle varie operazioni sui file quanto più immediate e fornire una convenienza. Ad esempio una caratteristica che deve fornire il SO quando lavoriamo coi file è la possibilità di associare all'atto della creazione di un nuovo file, un qualsiasi nome senza far sì che l'utente si debba preoccupare che quel nome è stato utilizzato da un altro utente per un proprio file. Questo perché il SO deve fornire la possibilità ai file di uno specifico utente di poter essere condivisi con altri utenti purché abbiano determinati permessi. Compito del SO è di garantire anche una forma di protezione contro le interferenze di utenti non autorizzati.

E' fatta una copia privata
della pagina k
quando A la modifica

Quando lavoriamo coi file, le info sono memorizzate su disco. L'altro aspetto che quindi deve tener conto il SO è quello di effettuare operazioni di I/O e uso del disco quanto più efficiente. Per poter gestire questi due aspetti, uno legato agli utenti e l'altro l'ottimizzazione delle operazioni sui file in termini di scrittura e lettura su disco, il SO contempla una gestione del file system organizzata in due parti:

- La prima parte è proprio il file system che consiste nell'organizzazione dei file, il modo di accesso, i meccanismi di protezione
 - Dall'altro c'è l'input / output control system che è la componente che si preoccupa di ottimizzare le operazioni sui file.
-

PANORAMICA ELABORAZIONE DEI FILE

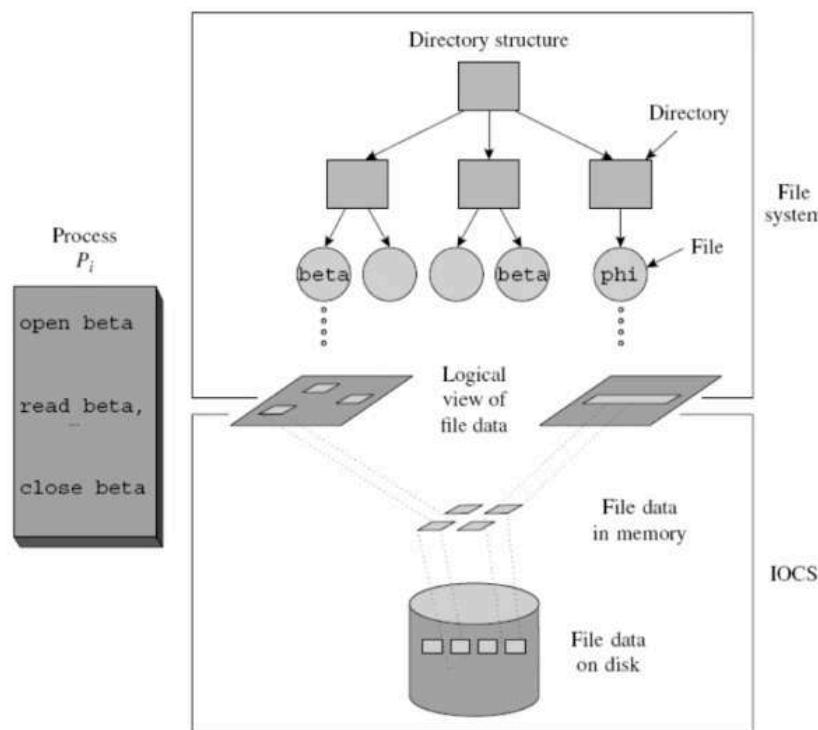


Figure 13.1 File system and the IOCS.

Facciamo una panoramica delle operazioni che si fanno dal lato utente sui file.

Tipicamente si fanno operazioni di lettura, creazione, scrittura, chiusura di un file.

Quello che vediamo in figura è un'organizzazione per operazioni di questo tipo e in particolare tutta la componente del SO che gestisce i file è suddivisa in due parti: una logica che consente di organizzare i file in un certo modo e l'altro l'ottimizzazione delle operazioni di I/O.

Nella figura vediamo che per garantire un uso semplice e garantire le operazioni di cui prima (il fatto che possa associare un nome qualsiasi di file senza preoccuparsi di altri utenti e condividerli...), è organizzato in una sorta di albero in cui abbiamo una directory radice che a seconda del tipo del SO ha un nome diverso, che contiene info sulle directory associate agli utenti che possono usare il sistema. Vediamo che abbiamo due tipi di file chiamati "beta" e quando ho un processo Pi, che fa operazioni tipiche sui file come lettura, apertura e chiusura, il nome del file determina il file specifico che andiamo ad aprire. Quando il processo apre il file fa riferimento a uno specifico file del file system e può fare le sue operazioni.

La parte logica si conclude con l'organizzazione del file: i file sono di diverso tipo all'interno di un sistema ma possiamo considerarli in due categorie:

- File strutturati
- File non strutturati, visti come una sequenza di byte.

Abbiamo poi la componente che ottimizza le operazioni di I/O su disco, la IOCS (input output control system). La IOCS va a prelevare il contenuto di un file dal punto di vista logico e va a realizzare le operazioni su disco facendo un mapping dalla vista logica a quella fisica. In questo tipo di operazioni la IOCS può far uso di un supporto HW per ottimizzare le operazioni di I/O.

FILE SYSTEM E IOCS

Da un punto di vista logico, il file system vede un file come una raccolta di dati che sono di proprietà di un certo utente e che può essere condiviso con altri utenti se hanno permessi. Il file system deve far sì che questi dati siano memorizzati per lungo tempo in maniera affidabile. Inoltre compito del file system è usare un nome random senza problemi. File system e IOCS è come se costituissero una gerarchia che adotta delle politiche per seguire l'obiettivo del file system e dell'IOCS (ottimizzazione operazioni I/O). I due componenti interagiscono nel senso che durante l'operazione di lettura e scrittura del file system, questo invia una richiesta all'IOCS che richiede sotto forma di syscall l'operazione su disco. In generale vediamo i servizi che un file system e IOCS forniscono:

FILE SYSTEM:

- Organizzazione dei file in una struttura di directory per gestire utenti
- Protezione contro accessi non consentiti
- Semantica per condividere l'accesso ai file a utenti
- Memorizzazione affidabile.

IOCS:

- Efficienza delle operazioni I/O
- Efficienza dell'accesso ai dati.

Un file system consiste di due tipologie di dati:

- Dati effettivi contenuti nel file
- Insieme di info di controllo dette ***metadati***.

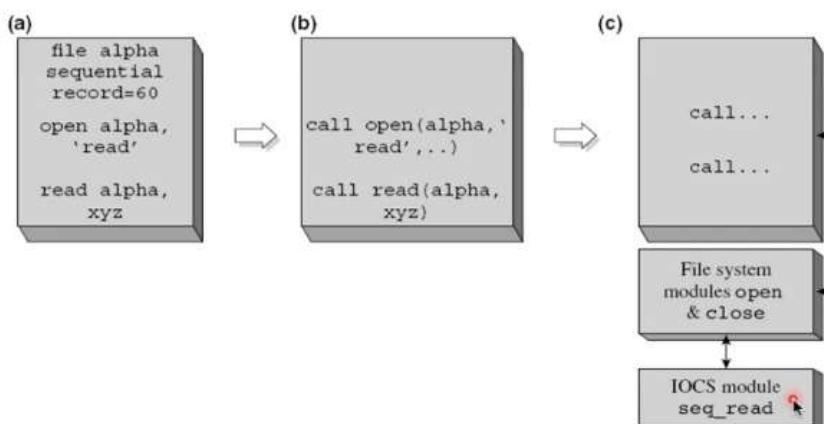
ELABORAZIONE DEI FILE IN UN PROGRAMMA

Quando lavoriamo coi file molto spesso lo si fa con un linguaggio di programmazione. Si usano funzioni che dichiarano un file con le sue caratteristiche e fare le tipiche operazioni di questo.

Da un punto di vista del linguaggio di programmazione, un file è un oggetto con insieme di attributi che descrivono come è organizzato il file (se i dati sono strutturati, se sono un flusso di byte sequenziale) e questi attributi descrivono i metodi per accedere al file stesso. Durante l'esecuzione di un programma che opera su un file, l'elaborazione viene implementata dai moduli di una libreria del file system e anche dai moduli dell'IOCS.

In figura abbiamo nella prima parte un codice nel quale c'è una dichiarazione di un file di tipo sequenziale (Si accede sequenzialmente ai record). Lo stesso file viene aperto e si effettua una lettura. In questo modo il compilatore sa qual è il tipo di file, come accedervi e tutte le istruzioni per accedere a ciascun record, il compilatore poi va a sostituire le chiamate come open, read... con i moduli della libreria del file system e passa gli attributi del file come chiamate a queste funzioni.

L'istruzione del linguaggio ad alto livello viene sostituita con la libreria. In fase di esecuzione l'operazione è realizzata con una cooperazione tra le funzioni della libreria del file system e dal modulo dell'IOCS.



FILE E OPERAZIONI SU FILE

Vediamo come sono organizzati i dati all'interno di un file. I file sono di vario tipo e ciascun file è caratterizzato da un insieme di attributi che ne determinano le sue caratteristiche (tipo di file, dimensione, posizione su disco, info su controllo di accesso).

Ciascun file è mantenuto nelle directory, ad ogni entrata della directory corrisponde un file e ci sono gli annessi attributi.

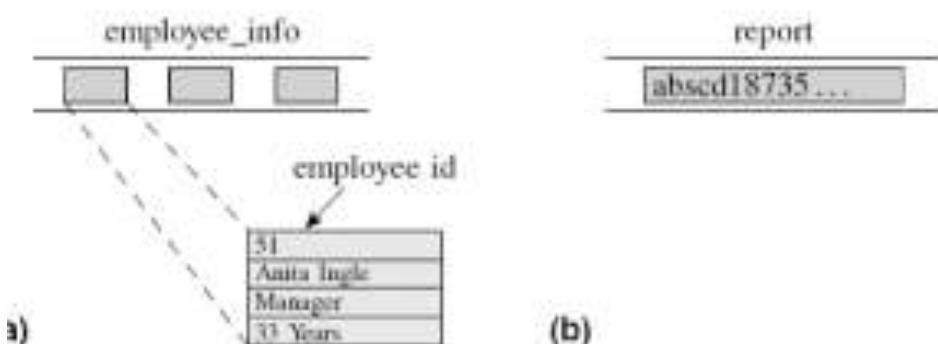
Il file system usa le info sugli attributi per localizzare i file e assicura che tutte le operazioni siano consistente con gli attributi del file stesso: se devo fare una scrittura il file system si accerta che posso fare un'operazione del genere.

I file possono essere organizzati in due categorie:

- Strutturati: collezioni di record, dove ogni campo contiene un dato di un certo tipo. Le info memorizzati in un file strutturato sono organizzati in record dove ognuno di esso deve essere identificabile e avviene grazie a un valore "campo chiave" che è univoco.
- Stream di byte: il file è una sequenza di byte, senza struttura preposte.

La figura illustra le due differenze:

- Nel primo caso abbiamo un file strutturato, si suppone che i record abbiano info su impiegati e ogni record ha una serie di info, il campo chiave, il nome, il cognome, il ruolo, l'età
- Nel secondo caso abbiamo uno stream di byte che possono essere acceduti in modo sequenziale



Il file system si preoccupa di operazioni tipiche su file come apertura, scrittura, chiusura e altre operazioni:

- Copia di un file
- Cancellazione
- Rinominare
- Attribuire dei permessi specifici su quel file per utenti

Queste sono tutte operazioni messe a disposizione del file system. Queste info possono richiedere info associate al file che sono contenute all'interno all'entrata della directory.

L'accesso effettivo al contenuto del file è gestito dal IOCS.

ORGANIZZAZIONE DEI FILE E METODI DI ACCESSO

A prescindere da un file record o una sequenza di byte, un file può essere acceduto in maniera diversa, anche in base alla natura del dispositivo di I/O su cui il file è memorizzato.

Definiamo il modello di accesso a un record come la modalità di accesso a ciascun record o un singolo byte all'interno di un file. I due possibili modelli di accesso sono:

- **Sequenziale:** data la posizione in un certo istante, possiamo accedere al record successivo da leggere solo se questo segue o precede il record corrente.
- **Casuale:** non ci sono limitazioni e possiamo accedere a un qualsiasi record senza tener conto della sua posizione.

L'organizzazione del file è una combinazione del metodo con cui i record sono memorizzati in un file e di una procedura per potervi accedere opportunamente.

La loro efficacia dipende anche dal tipo di periferica sottostante, se avessi un'unità a nastro, questa si presta per un accesso

sequenziale mentre un disco efficiente implementa l'accesso sequenziale e casuale.

Un file system supporta diversi tipi di organizzazioni:

- Organizzazione sequenziale
 - Organizzazione diretta
 - Organizzazione indicizzata
-

ORGANIZZAZIONE SEQUENZIALE DEI FILE

In un file sequenziale ciascun record è memorizzato come una sequenza, sulla base del campo chiave. È un tipo di organizzazione utile quando abbiamo la possibilità di preordinare i dati e poi accedervi in modo sequenziale.

Sono supportate due tipi di operazioni:

- Lettura/scrittura del prossimo record
- Saltare il prossimo record

Questa organizzazione è anche adatta per lo stream di byte: nell'organizzazione non strutturata, dove un file è visto come una sequenza di byte, questa organizzazione si presta molto perché possiamo accedere a un byte nell'ordine in cui è scritto nel file system.

ORGANIZZAZIONE DIRETTA DEI FILE

Con questa organizzazione è immediato fare operazioni di elaborazione dei record quando possono essere acceduti anche in maniera casuale.

In questo caso il file viene denominato ad **accesso diretto** e l'idea è che quando dobbiamo fare un'operazione di lettura o scrittura, andiamo a specificare il valore del campo chiave che è quello

associato al record e a quel punto il campo viene usato per determinare l'indirizzo del record sul dispositivo e poi realizzare l'operazione di lettura o scrittura.

L'organizzazione si presta bene per un accesso diretto, specifico un campo chiave e a prescindere dalla sua posizione, vi posso accedere direttamente perché uso la funzione che associa a ogni record un indirizzo sulla periferica.

Gli svantaggi di questa organizzazione sono differenti:

- Si deve accedere specificando il record e poi calcolare l'indirizzo fisico: questo comporta overhead.
- Quando si tratta di gestire file ad accesso diretto su disco, questa organizzazione determina uno spreco di memoria perché un disco, per sua conformazione fisica, è costituito da un insieme di tracce.
Le tracce più esterne contengono più info ma col file ad accesso diretto, su ciascuna traccia si pongono la stessa quantità di record e dello spazio sulle tracce più esterne viene inutilizzato.
- Presenza di record fintizi per valori chiave che non si usano.

ESEMPIO: FILE AD ACCESSO SEQUENZIALE E DIRETTO

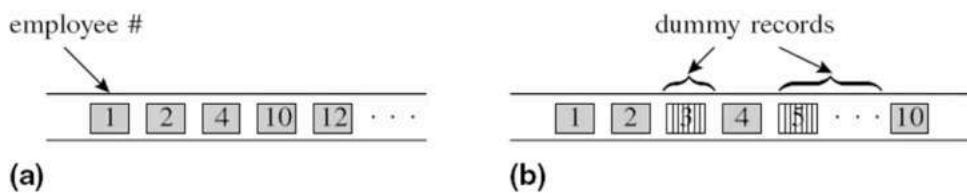
Nell'esempio vediamo l'organizzazione sequenziale di un file e ad accesso diretto.

Entrambi i file contengono info su un certo numero di impiegati e in un dato momento, gli impiegati 3, 5-9, 11 sono stati rimossi.

- a. Nel caso di organizzazione sequenziale abbiamo i record effettivamente presenti e vi accedo sequenzialmente. Se ci troviamo sull'impiegato 1 posso accedere al 2 e così via.
- b. Nel caso di un accesso diretto, specifichiamo il numero del record e tramite l'associazione record-indirizzo fisico, accediamo all'interno del file.

Vediamo come l'accesso diretto fa sì che per poter essere usato sia necessario avere questi record fintizi che contengono il valore chiave e sono necessari per far funzionare il sistema.

- Gli impiegati con numeri 3, 5-9, 11 hanno lasciato l'azienda
 - I file ad accesso diretto usano record fintizi per tali record



ORGANIZZAZIONE INDICIZZATA DEI FILE

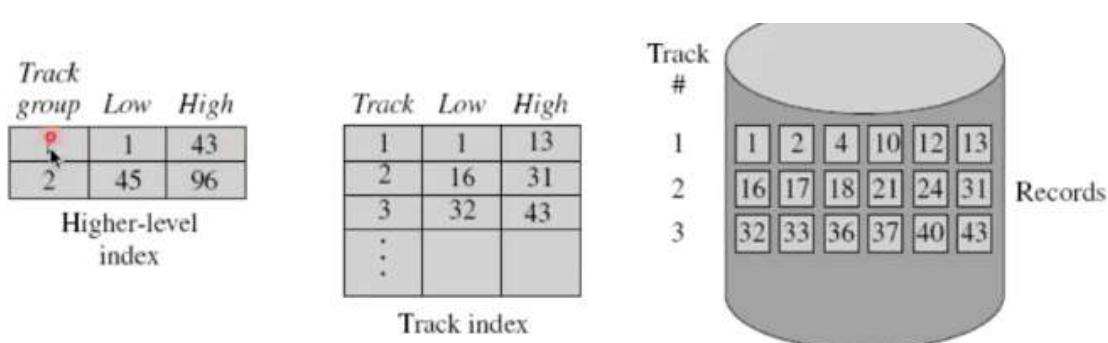
Per accedere ai vari record di un file si utilizza un indice che aiuta a determinare l'indirizzo fisico di un record con una chiave. Possiamo avere un'organizzazione indicizzata *pura* dove l'indice è una coppia (chiave, indirizzo su disco) e a ogni chiave ho l'indirizzo fisico sul disco che contiene quel record.

Ciò comporta che nel momento in cui devo fare una ricerca, la faccio sull'indice e individuo la chiave ricercata con la sua posizione fisica sul disco.

Il vantaggio è che se il file che contiene l'indice è più piccolo del file di dati, la ricerca avviene molto più veloce rispetto a farla sul file di dati.

È anche vero che ci possono essere situazioni dove l'indice assume dimensioni elevati e ciò comporterebbe operazioni non efficienti e si possono usare strategie ibride: uso un indice a più livelli:

- Organizzazione indicizzata sequenziale: si usano due strutture. Ottengo un indice di livello più alto in cui vado a raggruppare un certo numero di indici che contengono i dati che dobbiamo ricercare. L'altra info associata al record più elevato è il minimo e il massimo memorizzato all'interno del gruppo di tracce. Si parte dall'indice elevato per individuare qual è il gruppo di tracce che può contenere il record che ricerchiamo (grazie ai valori min e max). A quel punto si procede a fare una ricerca per vedere in quale traccia si trova il record, una volta individuata andiamo sul dispositivo fisico dove viene fatta una ricerca sequenziale.



Il vantaggio è che le operazioni di ricerca le faccio su file indice che sono più piccoli rispetto alla dimensione del file e riesco avere un compromesso all'efficienza d'accesso rispetto all'organizzazione sequenziale ed è migliore dell'organizzazione indicizzata pura nel caso in cui la dimensione dell'indice dovesse essere troppo grande.

METODO DI ACCESSO

Il modulo dell'IOCS che implementa gli accessi ad uno specifico file, si basa sull'organizzazione del file.

La procedura di accesso che viene implementata nell'IOCS dipende da com'è organizzato il file. Siccome il modulo di IOCS passa dalla visione logica dei file all'organizzazione fisica, utilizza delle tecniche di supporto HW per ottimizzare le operazioni di I/O.

- Una tecnica consiste nel fare un buffering dei record: prima di un'operazione di lettura o scrittura, i dati sono messi preliminarymente messi in un buffer in memoria prima che avvenga l'operazione sul file. Per cercare di limitare l'attesa da parte di un processo.

 - Altra tecnica è quella di fare blocchi di dati: viene letto o scritto sul dispositivo di I/O un grande blocco di dati, di dimensione maggiore di un record nel file. È come se si scrivesse più record, ciò viene fatto per ridurre il numero di operazioni di I/O necessarie per elaborare operazioni su un file. Piuttosto che fare tante operazioni di I/O, si fa un blocco di operazioni e si riduce il numero di accesso al dispositivo fisico.
-

DIRECTORY

Abbiamo detto che i file all'interno del file system sono organizzati in directory che possono essere viste come tabelle dove a ogni entrata ci sono info su ciascun file. Un'entrata di una directory può rappresentare una directory (cartella nella cartella). Ogni entrata della directory contiene tutte le info necessarie per svolgere le operazioni sul file. Queste info sono usate per inviarle al modulo di IOCS e per le operazioni di protezione per evitare utenti non autorizzati.

Vediamo una serie di attributi associati a un file:

- Nome del file
- Tipo e dimensione.
- Informazioni associate a dove sono localizzati i blocchi che contengono i dati dei file.
- Informazioni sulla protezione.
- Numero di processi che hanno quel file aperto.
- Lock che indica se un processo che usa il file lo sta facendo in maniera esclusiva o meno.
- Flag che indicano altre info sulla natura del file.
- Varie info che indicano ID del proprietario, istante in cui è creato, ultimo utilizzo, ultima modifica ecc...

<i>File name</i>	<i>Type and size</i>	<i>Location info</i>	<i>Protection info</i>	<i>Open count</i>	<i>Lock</i>	<i>Flags</i>	<i>Misc info</i>

Relativamente alle directory, sappiamo che un file system ospita file posseduti da più utenti, ciò significa che il file system deve concedere agli utenti due prerogative:

- Libertà di assegnare i nomi ai file.
- Condivisione dei file.

Per consentire queste operazioni, il file system è organizzato in un albero di directory dove la prima directory è chiamata “*master directory*”, che contiene tutte le info sulle directory associate ai vari utenti.

In figura a,b,c rappresentano 3 ipotetici utenti e la master directory ha tutte le info su di essi.

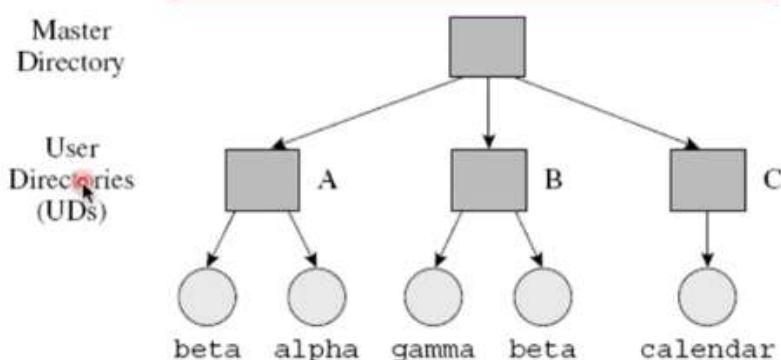
Andando a separare le directory dei vari utenti, si consegne il primo obiettivo (ciascun utente può usare un nome qualsiasi già usato da un altro utente). L’utente a può chiamare il suo file “beta”, come può farlo anche l’utente b poiché sono due file diversi a cui si accede in modo diverso. Si accede a partire dalla directory che deve aprire il file.

Utente A apre beta: `open(beta, ...)`
Utente B apre beta: `open(beta, ...)`

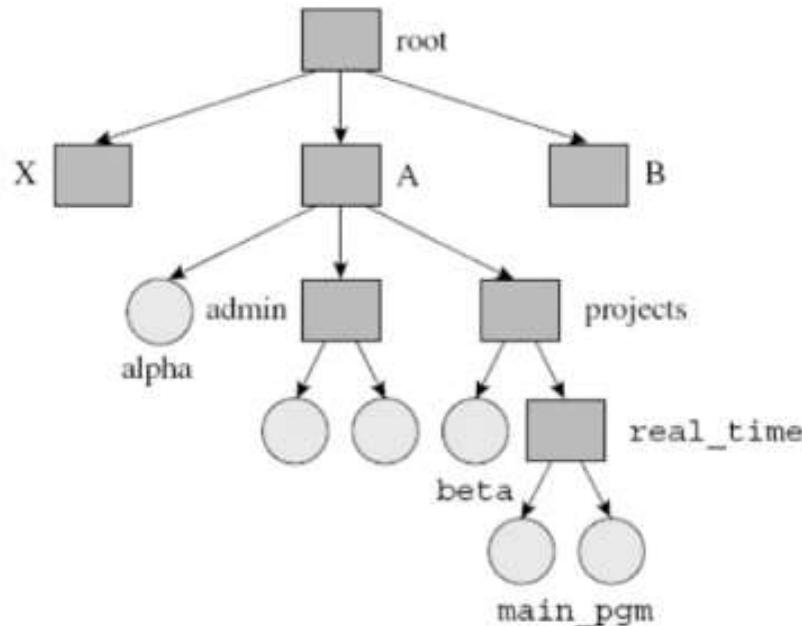
In questo modo però non è chiaro com’è possibile condividere file tra utenti. Bisognerebbe introdurre una sintassi specifica per la condivisione: se utente C vuole accedere al file di A

- utente C può eseguire
`open(A->beta, ...)`

Per verificare se ciò è possibile, si va a verificare il campo di info di protezione del file beta per vedere se l’utente C ha i permessi necessari per aprire il file di A.



ALBERI DELLE DIRECTORY



questa è la struttura adottata tipicamente da Multix (pro genitore di UNIX).

La master directory prende il nome di ROOT e all'interno ci sono le info associate agli utenti del sistema (x, a, b in figura). Ciascun utente a partire dalla propria directory può costruirsi il proprio file system (creare cartelle, file, cartelle in cartelle ...).

In questa organizzazione si incontrano concetti moderni:

- Home directory → per ogni utente il proprio file system si sviluppa a partire da questa
- Directory correnti: specifica directory in cui ci troviamo in un momento.

Quando si fanno operazioni per identificare un file si usano dei percorsi per identificare il file, i percorsi sono specificati attraverso i “path name” che si dividono in:

- Path name relativi → path name seguiti a partire dalla directory corrente. Se devo specificare un nome di un file da aprire, lo cerco nella directory corrente.

- Path name assoluti → nel caso in cui voglio identificare un file nel file system si usa l'assoluto che specifica il percorso completo a partire dalla directory root e si usa lo / per dividere le varie cartelle fino ad arrivare al file che si cerca.

GRAFI DELLE DIRECTORY

In questo modo abbiamo difficoltà nella condivisione: dobbiamo specificare i path name relativi che possono essere difficili da specificare, per cui piuttosto che usare una struttura ad albero pura, si usa un grafo aciclico (albero con collegamenti su di uno stesso file da parte di più directory).

La condivisione diventa molto più semplice, e la si implementa con il concetto di link.

- Forma generale di un link: (`<from_file_name>, <to_file_name>, <link_name>`)

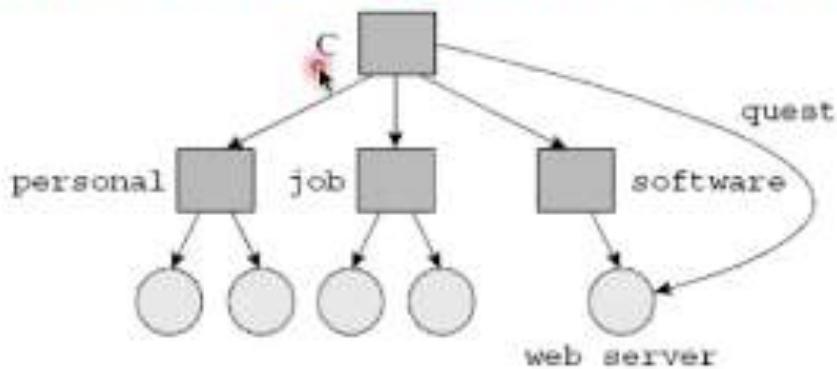
Definisco il file su cui voglio costruire il collegamento, do il target e assegno a questo collegamento un nome.

In figura abbiamo il grafo, un file web server che si trova nella directory software dell'utente C, possiamo andare a creare un link che è situato all'interno dell'origine della cartella C, il target è il file web server e il nome del link è *quest*. È stato creato un link di nome *quest* all'interno della cartella C.

Possiamo andare a collegare il file web server tramite due percorsi (grazie al grafo). Se avessi un altro utente potrei definire un altro link da un file a un altro utente, supportando la condivisione.

Devo garantire comunque l'accesso se quell'utente ha i permessi necessari.

(~C, ~C/software/web_server, quest)



PROTEZIONE DEI FILE

I permessi possono essere rappresentati in vari modi.

Ipotizziamo che le info di protezione siano sotto forma di lista di controllo accessi nella quale per ciascun utente vado a considerare quali sono i suoi privilegi di accesso per uno specifico file. Avremo quindi una tabella per ogni file con tutti gli utenti e i loro permessi di accesso.

In questo caso avrei una tabella troppo grande perché si considera ogni utente, si considerano quindi i gruppi utente per diminuire la grandezza della tabella. La lista di privilegi di accesso, caso linux ecc, sono operazioni di lettura, scrittura ed esecuzione. Tutte queste info sono contenute nel campo ***protection info*** che è contenuto nell'entrata di ciascun file all'interno di una directory.

ALLOCAZIONE DI SPAZIO SUL DISCO

All'interno di un disco possono risiedere più file system, ogni FS è creato su un disco logico, ovvero su una partizione di un disco. Proprio perché un FS è creato da un punto di vista logico, la componente HW ignora l'organizzazione logica, ciò significa che nel momento in cui si deve allocare spazio per un file,

l'allocazione dello spazio ad un file è compito del FS e non dell'IOCS perché siccome viene creato su un disco logico, l'organizzazione logica non è nota all'IOCS.

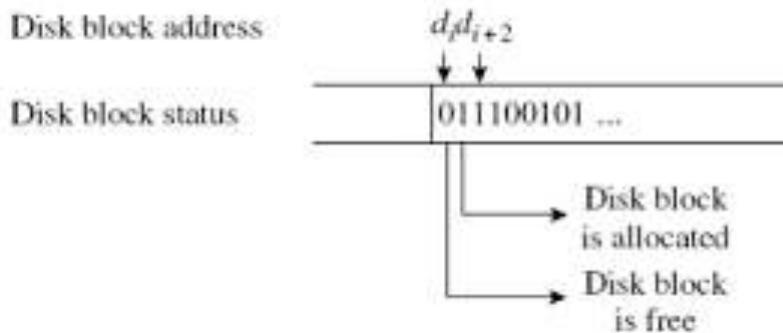
Si possono fare diversi tipi di allocazioni di memoria su disco:

- In passato si faceva un'allocazione contigua, si allocavano blocchi contigui su disco di una certa dimensione. Se il file è piccolo si ha un blocco piccolo, se grande un blocco grande. In questo tipo di modello si avevano problemi di frammentazione esterna perché allocando blocchi contigui di diverse dimensioni rimangono poi piccole zone contigue non sufficienti per memorizzare file. Si ha anche frammentazione interna perché quando si alloggia la memoria al file, si allocava una memoria superiore a quanto richiesto per far crescere il file successivamente e quella memoria poteva non essere utilizzata. Inoltre c'è un altro inconveniente in questa allocazione: richiedeva complesse organizzazioni per evitare l'uso di blocchi danneggiati, con l'allocazione contigua in fase di formattazione del disco, si identificavano i blocchi danneggiati e si annotavano i loro indirizzi, si allocavano poi blocchi alternativi e si costruiva una tabella con blocco danneggiato e blocco sostituito in modo che l'IOCS andasse a verificare se il blocco era danneggiato, in modo da consultare la tabella e prendere il blocco sostitutivo. Tutti i sistemi moderni non usano questa allocazione.
- I sistemi moderni adottano la strategia di allocazione non contigua (come nella memoria centrale). All'atto della creazione di un file, viene attribuita una certa quantità di memoria on demand o nel momento in cui il file cresce di dimensione e tutto ciò che è necessario successivamente

avviene on demand. Per realizzare questa allocazione bisogna affrontare 3 problematiche:

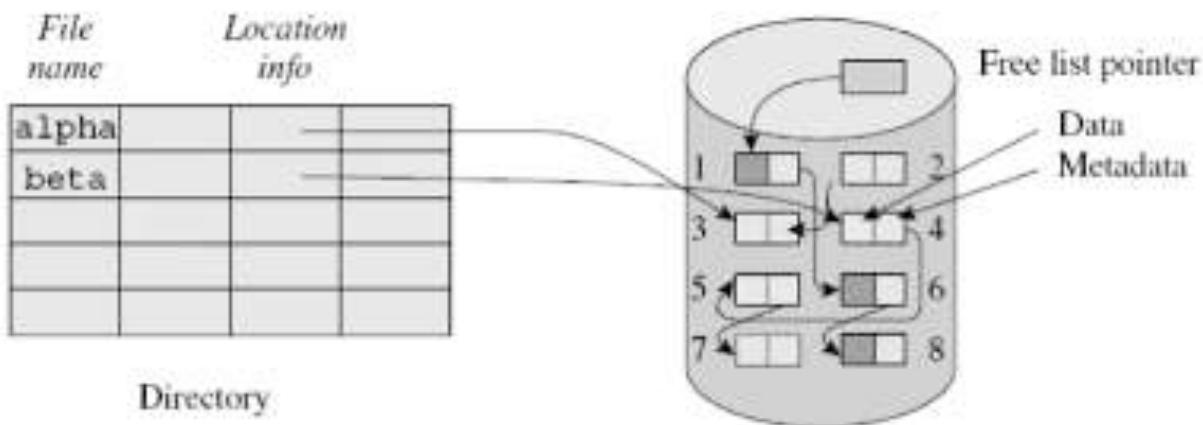
- Come si gestisce lo spazio su disco? Mediante l'uso di una free list (lista concatenata di blocchi liberi all'interno del disco e quando è necessario allocare un nuovo blocco si preleva da questa lista i blocchi necessari per le operazioni). Alternativamente si può usare una struttura "DISK STATUS MAP" che è una tabella in cui per ciascun blocco su disco c'è un bit associato che indica se il blocco è libero o meno.
- Il secondo problema è quello di evitare troppi movimenti di una testina del disco: significa che devo allocare blocchi non contigui ma devo evitare blocchi che sono estremamente sparsi su disco perché spostare il disco da blocco all'altro comporta spostamenti ampi della testina che hanno tempi molto elevati rispetto a operazioni tra CPU e memoria centrale. Per evitare ciò si usano le estensioni: quando vado ad allocare spazio in maniera non contigua, non alloco i singoli blocchi ma cerco di allocare blocchi contigui detti "cluster di blocchi" o "gruppi di cilindri", insieme di blocchi vicini in un disco.
- Accedere ai dati nel file: bisogna gestire le info sullo spazio allocato a ciascun file per poter accedere ai suoi dati. Il tipo di accesso dipende dall'approccio che si implementa che è un accesso di tipo concatenato o indicizzato.

Per quanto riguarda la DSM (disk status map), abbiamo una tabella dove a ogni blocco è associato un bit che rappresenta se un blocco è libero o allocato (prende il nome anche di bit map). Ogni volta che si allocano blocchi si consulta questa tabella. Abbiamo una sequenza di bit 0 (libero) - 1(allocato).



Un vantaggio della DSM: per ridurre il movimento delle testine la DMS sceglie subito i blocchi cluster perché basta vedere nella sequenza quali sono i blocchi liberi consecutivi.

ALLOCAZIONE CONCATENATA



L'idea è quella di partire dalla directory e i blocchi allocati a un disco sono contenuti nella lista concatenata che è costruita in questo modo:

partendo dalla directory ho il file in una cartella, nell'entrata della directory per quel file, nel campo location info ho l'indirizzo del primo blocco allocato fisicamente sul disco. Ciascun blocco è

costituito da due parti: il dato e un indirizzo che è il blocco successivo allocato a quel file.

Ad esempio in figura il blocco 3 è seguito dal blocco 2.

- C'è anche una free list, un puntatore che punta al primo blocco libero che a sua volta punta ad altri blocchi liberi.

Questa allocazione è facile da implementare e sia allocazione e deallocazione introduce un basso overhead.

È adeguata per file di tipo sequenziale perché si segue l'ordine dei blocchi allocati per accedere ai vari record o byte.

Il campo puntatore di ciascun blocco è chiamato metadato, è la info di controllo per accedere ai dati effettivi.

Nei file non sequenziale non si ha efficienza.

- Un problema che sorge è la scarsa affidabilità perché se dovesse danneggiare un puntatore al prossimo blocco, perderei tutto il percorso.

Questa allocazione era usata da **MS-DOS** che usa una variante dell'allocazione concatenata che memorizza i metadati separatamente dai dati nel file. MS-DOS usa una struttura chiamata “**FAT**” (*file allocation table*).

In questa gestione partiamo dalla directory con le entrate dei file. Location info contiene il primo blocco che è allocato al file e si trova nella tabella FAT.

Nel file alfa, accediamo all'elemento 3 della tabella e contiene il blocco successivo allocato al file, il numero 2 che viene marcato con “**end**” per indicare che non c'è altro di allocato in seguito. I blocchi grigi sono quelli della tabella dei blocchi liberi.

Lo svantaggio di questa variante è che se si deve tenere l'indirizzo su disco, prima di accedere a esso devo accedere prima alla FAT per poi ottenere l'indirizzo dei blocchi e ciò causa overhead (la FAT è comunque nel disco).

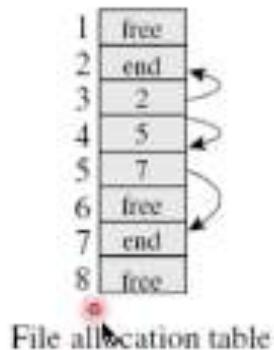
- Questo problema si risolve tenendo la FAT in memoria principale e poi si va al disco.

C'è poi il problema dell'affidabilità che permane:

- Rispetto all'organizzazione base, è maggiormente affidabile perché se si danneggia un blocco del disco perdo solamente quello.
- Si può verificare un altro problema: la FAT se si danneggia su disco prima di portarla in memoria, perdo TUTTI i dati sul disco e ho situazioni disastrose.

<i>File name</i>	<i>Location info</i>
alpha	3
beta	4

Directory



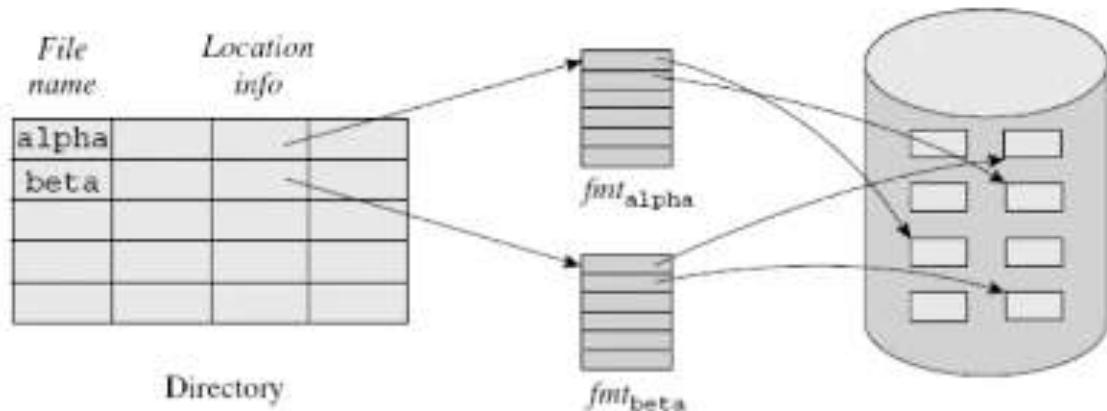
ALLOCAZIONE INDICIZZATA

Per allocare blocchi si va a gestire un indice.

L'indice viene memorizzato come tabella e prende il nome di "file map table" e memorizza gli indirizzi dei blocchi del disco allocati ad un file.

In questa organizzazione per ciascun file viene mantenuta l'indirizzo del FMT (file map table) e all'interno della tabella ci sono i blocchi fisici allocati sul disco.

Nella forma base la FMT è un array di indirizzi di blocchi di disco.



È più affidabile rispetto all'allocazione concatenata perché se si danneggia un elemento, perdo solo quello ma posso accedere agli altri.

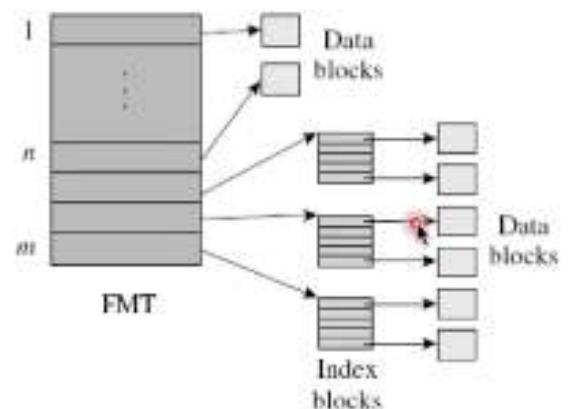
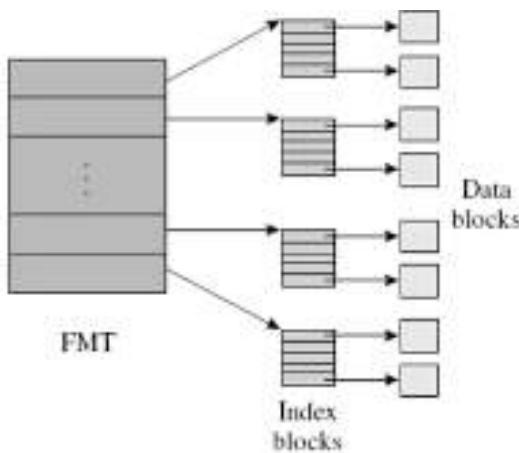
È un'organizzazione che non si adatta all'organizzazione sequenziale dei file perché si dovrebbe comunque consultare la tabella per accedere al record successivo.

Per i file ad accesso diretto è performante.

Se ho file piccoli l'intera FMT posso tenerla nell'entrata della directory corrispondente al file in questione, se il file è di grosse dimensioni non può avvenire per cui sono state create varianti.

Le varianti usano organizzazioni indicizzate a più livelli:

- FMT a due livelli: ho una prima tabella che punta a sotto tabelle di blocchi indice da cui poi ho gli indirizzi effettivi dei blocchi sul disco. L'accesso ai blocchi di dati è lento perché deve fare due passaggi ma la cosa importante è avere la prima dalla quale si ricavano gli accessi. È compatta ma lenta.
- Organizzazione FMT ibrida: una parte della map table indica direttamente i blocchi e un'altra parte usa un'indicizzazione a due livelli. Se ho piccoli file uso solo la parte di accesso diretto e velocizzo l'accesso



INTERFACCIA TRA FILESYSTEM E IOCS

L'interfaccia tra FS e l'IOCS devono interagire.

Il FS utilizza l'IOCS per eseguire operazioni di I/O e per fare questo c'è bisogno di un'interfaccia tra loro che è rappresentata da tre strutture dati:

1. File Map Table
2. File Control Block: contiene info riguardo l'elaborazione del file, c'è nome, tipo, organizzazione del file, se è un record,

metodo di accesso, dimensione di un record, blocco... contiene poi info sulla directory e lo stato del processo corrente.

3. Active File Table: tabella dei file attivi o aperti.

Per far sì che un FS invochi l'IOCS per la realizzazione di read o write ci si basa su queste tabelle che hanno le info necessarie per veicolare una richiesta di read del FS al modulo IOCS corrispondente. Questa organizzazione è usata anche da UNIX.

AFFIDABILITÀ DEL FILE SYSTEM

L'affidabilità dipende dalla possibilità di memorizzare i nostri dati, certi di non perderli e di recuperarli in caso di guasto.

L'affidabilità minimizza e non annulla le perdite.

L'affidabilità indica quanto il FS funziona correttamente, anche in caso di guasto.

I FS forniscono due strategie:

- Assicurare la correttezza della creazione, cancellazione e aggiornamenti
- Prevenire la perdita dei dati.

Guasto: difetto del sistema che crea un malfunzionamento nel FS, un guasto può essere mancanza di corrente.

PERDITA DI CONSISTENZA DEL FILE SYSTEM

Ci sono diverse situazioni per cui si possono perdere info memorizzate nei nostri file. A che fare con i dati e i metadati. Negli esempi supponiamo di avere un'allocazione concatenata.

Perdere consistenza significa che la correttezza dei metadati e info nei blocchi vanno perse e non ci consente di reperire i dati o averli non corretti.

Un qualsiasi guasto può comportare diversi problemi:

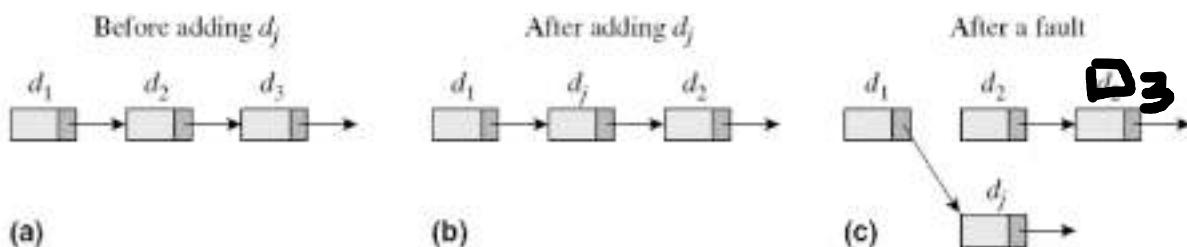
- Qualche dato di un file aperto viene perso.
- Parte di un file aperto può diventare inaccessibile.
- Si possono scambiare contenuti di due file.

➤ Ad esempio nel primo caso, consideriamo l'aggiunta di un blocco del disco ad un file e un guasto al passo 3.

1. $d_j.next := d_1.next;$
2. $d_1.next := address(d_j);$
3. Write d_1 to disk.
4. Write d_j to disk.

Il campo next del blocco da allocare deve puntare al campo next di d_1 . Il campo d_1 nuovo deve puntare a d_j e una volta aggiornati i puntatori devo aggiornare le info di d_1 e d_j su disco.

Supponiamo che si verifichi un errore dopo aver eseguito l'istruzione 3. Abbiamo aggiornato i campi puntatori, d_1 è scritto in disco ma d_j non è stato scritto su disco. Ciò significa che ci troviamo dopo un fault in una situazione dove parte del contenuto del file di partenza non è più accessibile, d_1 punta a d_j ma d_j non punta a nulla.



Un altro esempio perdita di consistenza: scambio di contenuto tra due file.

Supponiamo di avere due processi:

- P1 cancella il blocco dk da beta.
- P2 aggiunge un nuovo record ad alfa, il FS alloca un nuovo blocco dj e lo mette prima del blocco d2 in alfa

Supponiamo che dj=dk: dopo che P1 ha cancellato dk e lo ha reso disponibile, dk viene allocato al processo P2 aggiunge dj tra d1 e d2. Supponiamo che il file alfa venga chiuso, gli aggiornamenti su disco avvengono alla chiusura del file, il FS aggiorna alfa su disco. A questo punto c'è un guasto e non si è aggiornato beta. In questa situazione si è verificato che dk è stato deallocated, è stato allocato a dj che è posto tra d1 e d2 ma non abbiamo memorizzato beta e dh punta ancora a dk, in questo caso il guasto causa come malfunzionamento il fatto che si miscela il contenuto di due file.

APPROCCI PER L'AFFIDABILITA' DEL SO

Come gestire l'affidabilità del SO.

Si usano due strategie:

- Recupero → nel momento in cui c'è malfunzionamento si ripristinano i dati a prima che avvenisse il malfunzionamento dove il FS era in uno stato consistente.
- Tolleranza ai guasti → anche se si verifica un malfunzionamento, il sistema è in grado di operare come se non fosse accaduto nulla.

Se a un certo punto un blocco diventa illeggibile per un guasto, nel caso del recupero si recupera il contenuto di quel blocco a una situazione precedente, nel caso della tolleranza ai guasti l'idea è che ogni dato è memorizzato in due blocchi, se quel

blocco non è più raggiungibile riesco ad accedere alla sua copia, ovviamente se si corrompe anche la copia perdo il dato.

INPUT OUTPUT CONTROL SYSTEM (IOCS)

Abbiamo parlato di FS e di come l'interfaccia con la quale l'utente interagisce è in grado di gestire le operazioni tipiche sui file.

LIVELLI DI IOCS

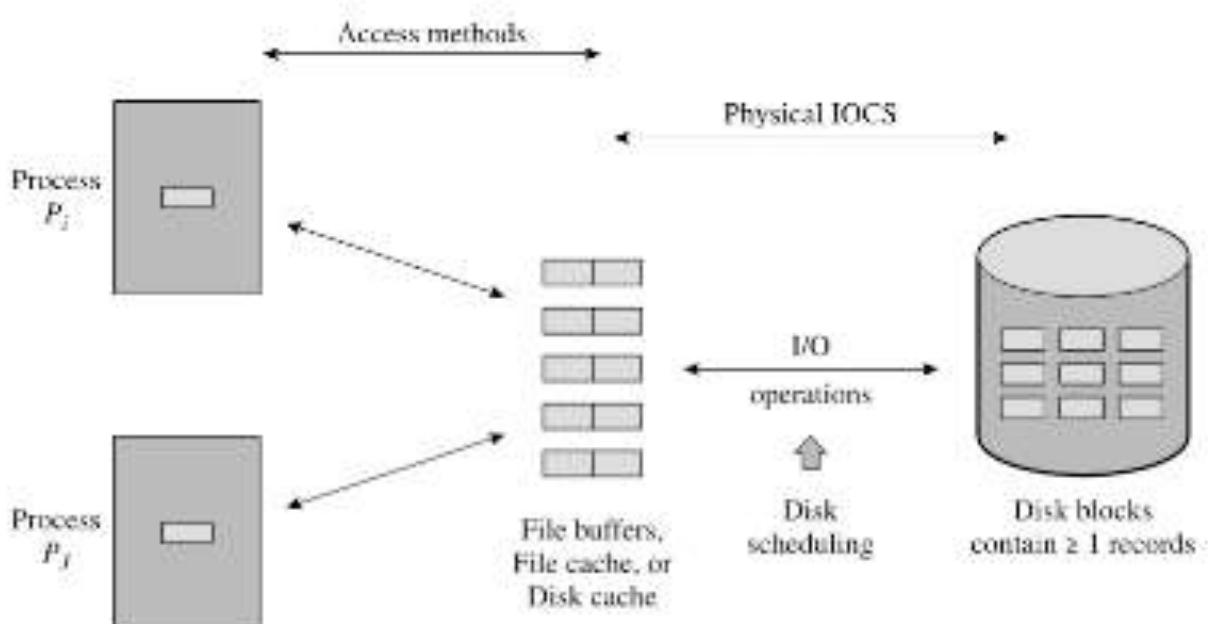


Figure 14.1 Implementation of file operations by the IOCS.

L'IOCS ha due obiettivi:

- Andare a consentire un'implementazione efficiente di tutte le attività di elaborazione dei file che un processo può invocare (lettura, scrittura, esecuzione ...).
- Avere un throughput elevato delle periferiche di I/O.

Per perseguire questi obiettivi l'IOCS è suddiviso a sua volta in due livelli:

- Il primo livello si occupa dei metodi di accesso: si riferiscono alle operazioni di lettura e scrittura dei dati al fine di implementare in modo efficiente le operazioni che un file può invocare.
- Il secondo livello è l'IOCS fisico che si occupa della realizzazione effettiva delle operazioni di scrittura e lettura su un file. Per implementare ciò, l'IOCS fisico esegue l'input output sulla periferica e usa le politiche di scheduling per migliorare il throughput dei dispositivi di I/O che sono coinvolti.

Quando abbiamo dei processi che eseguono richiesta di operazioni di lettura o scrittura su un file, il FS passa la richiesta all'IOCS il quale mantiene alcuni dati in aree di memoria (buffer, cache di file) in modo da velocizzare le operazioni.

Ci si basa sull'uso dei buffer per risolvere alcuni problemi di accesso condiviso al bus che si trova tra la memoria e la CPU e per velocizzare le operazioni sulla periferica stessa.

Questa separazione tra metodi di accesso e IOCS fisico consente di separare le problematiche relative all'implementazione dei file a livello di processo da quello a livello di dispositivo.

Inoltre nei vecchi SO l'IOCS fisico faceva parte del kernel mentre nei SO moderni lo si tiene fuori dal kernel per una questione di estendibilità del SO, tenendolo fuori dal kernel è più facile apportargli delle modifiche.

Assumiamo che l'IOCS fisico è fuori dal kernel, viene richiamato da syscall e l'IOCS fisico stesso è in grado di invocare altre funzioni del kernel attraverso ulteriori syscall.

ORGANIZZAZIONW DELL'I/O

Vediamo come è organizzato l'I/O considerando un I/O basato sull'uso del DMA. DMA è di supporto alle operazioni di I/O e le velocizza.

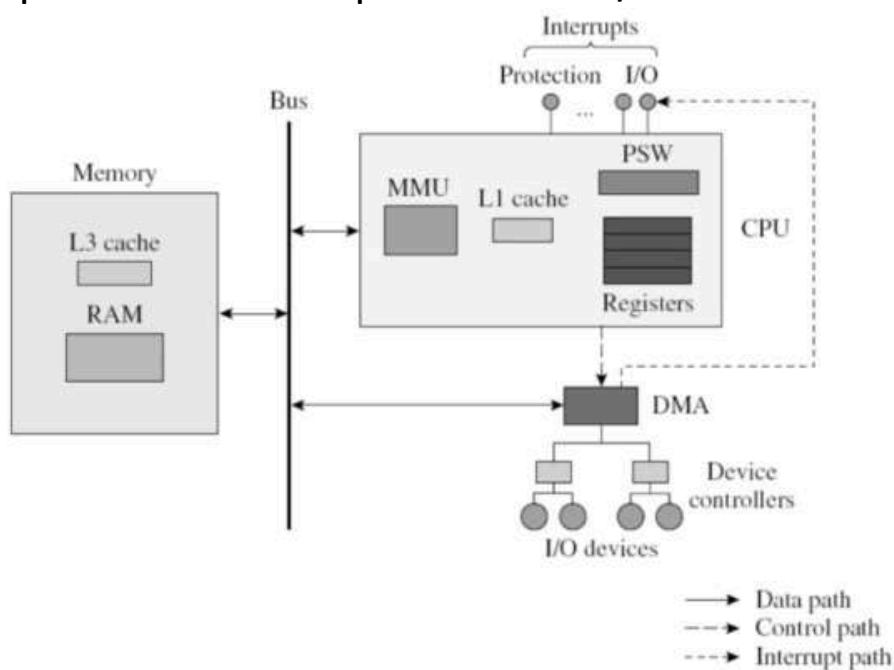
In questa organizzazione, osserviamo in figura che i dispositivi di I/O sono connessi a dei controller delle periferiche, ogni classe di periferica ha un controller. ES i dischi fissi hanno il relativo controller come i CD-ROM.

I controller sono a loro volta connessi ai controller del DMA.

Ogni controller di una periferica ha un ID numerico univoco e identifico all'interno del SO un controller di una periferica con l'ID e anche ogni classe di periferica ha un ID univoco.

Quindi il dispositivo è identificato dalla coppia (*ID controller, ID classe del controller*).

Quando si eseguono operazioni di I/O dove c'è il DMA, la CPU da via all'istruzione di I/O e durante il trasferimento dei dati dalla periferica alla memoria centrale, la CPU non interviene ed è libera di fare altro, infatti avviene che l'operazione di I/O viene eseguita eseguendo istruzioni di I/O e il DMA, un controller di dispositivo e il dispositivo implementano un'operazione di I/O.



OPERAZIONI DI I/O

I comandi di I/O sono tipicamente memorizzati in memoria e l'indirizzo dell'area di memoria che contiene i comandi viene usato come operando di istruzione di I/O che la CPU invia al controller del DMA quando deve avviare l'operazione di I/O. Per esempio se è necessario effettuare una *read* si specifica il disco e la CPU avvia un'operazione mediante un'istruzione di I/O

- Operazione di I/O *read* da un blocco del disco (*track_id, block_id*) eseguita mediante l'istruzione

I/O-init(controller_id, device_id), I/O_command_addr

Dove *I/O_command_addr* è l'indirizzo di partenza dell'area di memoria che contiene i seguenti comandi:

1. Posiziona la testina del disco sulla traccia *track_id*
 2. Leggi il record *record_id* nell'area di memoria con indirizzo di partenza *memory_add*
-

THIRD PARTY DMA

Nel DMA di terze parti quando viene eseguita un'istruzione di I/O:

- Il controller del DMA va a inviare i dettagli dei comandi di I/O al controller dello specifico dispositivo coinvolto nell'operazione.
- Il dispositivo una volta svolto le sue operazioni, consegna i dati al controller del dispositivo.
- Ad esempio nell'operazione di lettura, il controller si preoccupa di interagire col DMA per trasferire i dati in memoria.
- Il trasferimento dei dati in memoria avviene nel seguente modo:

- Il controller invia un segnale **DMA request** quando è pronto per trasferire i dati.
- Il DMA, dopo aver ricevuto il segnale, ottiene il controller del bus e vi pone l'indirizzo di memoria che partecipa al trasferimento e invia un segnale per dire che ha ricevuto il tutto al controller del dispositivo (segnale di **knowlagement**).
- Il controller dopo aver ricevuto il segnale trasferisce i dati verso la memoria.
- Quando il trasferimento è terminato, il DMA genera un interrupt di completamento dell'I/O a cui associa un codice che è l'indirizzo della periferica che ha completato l'operazione di I/O, parte l'interrupt e la sua routine analizza e individua il dispositivo che ha completato la sua operazione di I/O e agisce di conseguenza.

Durante il trasferimento dati attraverso il DMA, la CPU non interviene e in effetti essa continua il suo proseguimento.

Ciò potrebbe causare problemi: il bus è condiviso tra DMA, CPU e memoria e nel momento in cui la CPU esegue altro mentre è in atto il trasferimento dei dati in memoria, può succedere che il bus può essere conteso. Per evitare contese e ritardi, una tecnica usata è la “**cycle stealing**” che garantisce che la CPU e il DMA possono usare il bus senza causare troppi ritardi: la CPU va a favorire il controller del DMA per usare il bus in punti specifici del ciclo di istruzione, in particolare quando la CPU è in prossimità di leggere dati dalla memoria dà la possibilità al DMA di usare il bus. Il DMA quando deve trasferire i dati, attende fino a quando la CPU arriva a uno di questi punti del ciclo: raggiunto quel punto la CPU garantisce l'accesso al bus al DMA.

DISPOSITIVI DI I/O

All'interno di un sistema ci sono diverse periferiche di I/O e ognuna è diversa anche dal punto di vista fisico, alcune periferiche si basano sulla generazione di segnali elettromeccanici, altri della memorizzazione dati ottica o elettromagnetica.

La natura del dispositivo consente di classificare ciascuna periferica anche sulla base di alcuni criteri:

- Scopo: dispositivo di stampa, input e memorizzazione
- Natura dell'accesso: come il dispositivo accede al mezzo:
 - Sequentiale: tastiera, mouse, rete, nastro
 - Casuale: dischi.
- Modalità trasferimento dati: caratteri o blocchi.

L'info letta o scritta in un comando di I/O costituisce un record.

DISPOSITIVI DI I/O: MODALITA' TRASFERIMENTO DATI

Per quanto riguarda la modalità di trasferimento delle informazioni sui dispositivi di I/O, dipende dalla velocità di trasferimento che non è uguale per tutte le periferiche di I/O:

- Dispositivo di I/O lento: Tastiera, mouse e stampanti sono dispositivi a carattere e viene trasferito un carattere alla volta tra memoria e periferica. Questi dispositivi contengono un buffer che memorizza il carattere e il controller genera un

interrupt di conseguenza a una lettura o scrittura nel buffer. I controlli possono essere connessi direttamente al bus.

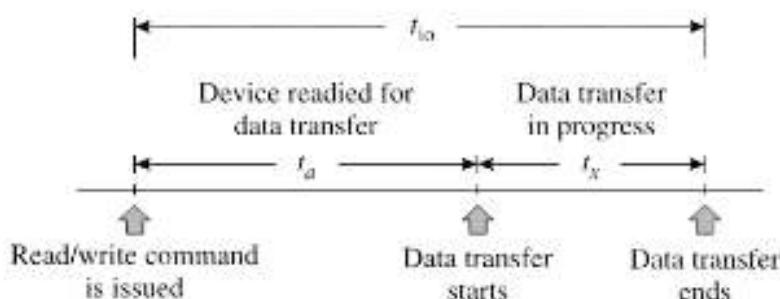
- Abbiamo poi dispositivi I/O veloci: nastri, dischi. Lavorano in modalità a blocco, sono connessi ad un controller di DMA e devono trasferire i dati a specifiche velocità. È necessario rispettare velocità specifiche perché si potrebbero avere problemi: i dati potrebbero essere persi durante una read se il bus non può accettare una periferica, lo stesso per una write se non riesce a consegnarla nel tempo stabilito. I dati sono trasferiti tra le periferica di I/O e un buffer nel DMA.
-

DISPOSITIVI DI I/O: TEMPO DI ACCESSO E DI TRASFERIMENTO

Le operazioni di I/O richiedono un tempo di esecuzione che può essere caratterizzato come la somma di due componenti.

- Tempo di I/O t_{i_o} : tempo che intercorre tra l'istruzione di I/O e completamento operazione.
- Tempo di accesso t_a : intervallo tra un comando read o write e l'inizio del trasferimento.
- Tempo di trasferimento t_x : tempo necessario per trasferire dati da/verso una periferica durante un'operazione di read o write.

$$\circ t_{i_o} = t_a + t_x$$



DISPOSITIVI DI I/O: INDIVIDUAZIONE E CORREZIONE ERRORI

Un altro aspetto che riguarda le periferiche di I/O sono la possibilità di individuare occorrenza di errore durante la lettura o il trasferimento di dati.

I dati da trasferire sono visti come un flusso di bit e l'idea è di usare dei codici per rappresentare queste info.

Ci sono tecniche per individuare e quindi correggere tali errori.

- Per individuare gli errori, si memorizzano le info ridondanti con i dati e lo si fa con tecniche standard esistenti. L'idea è che quando si leggono dati da una periferica, si leggono anche le info di individuazione dell'errore. Tali info sono calcolate nuovamente dai dati letti, usando la stessa tecnica:
 - Si confrontano le info lette dal mezzo di I/O e quelle determinate dai dati letti.
 - Un **mismatch** indica l'occorrenza di errore in fase di memorizzazione.
- La correzione dell'errore è fatta in modo analogo: si memorizzano info aggiuntive che algoritmi di correzione sfruttano per individuare l'errore e correggerlo.

Se voglio individuare e correggere errori devo memorizzare info ridondanti che causano overhead, se correggo l'errore è anche maggiore.

Per l'individuazione dell'errore ci sono diversi approcci:

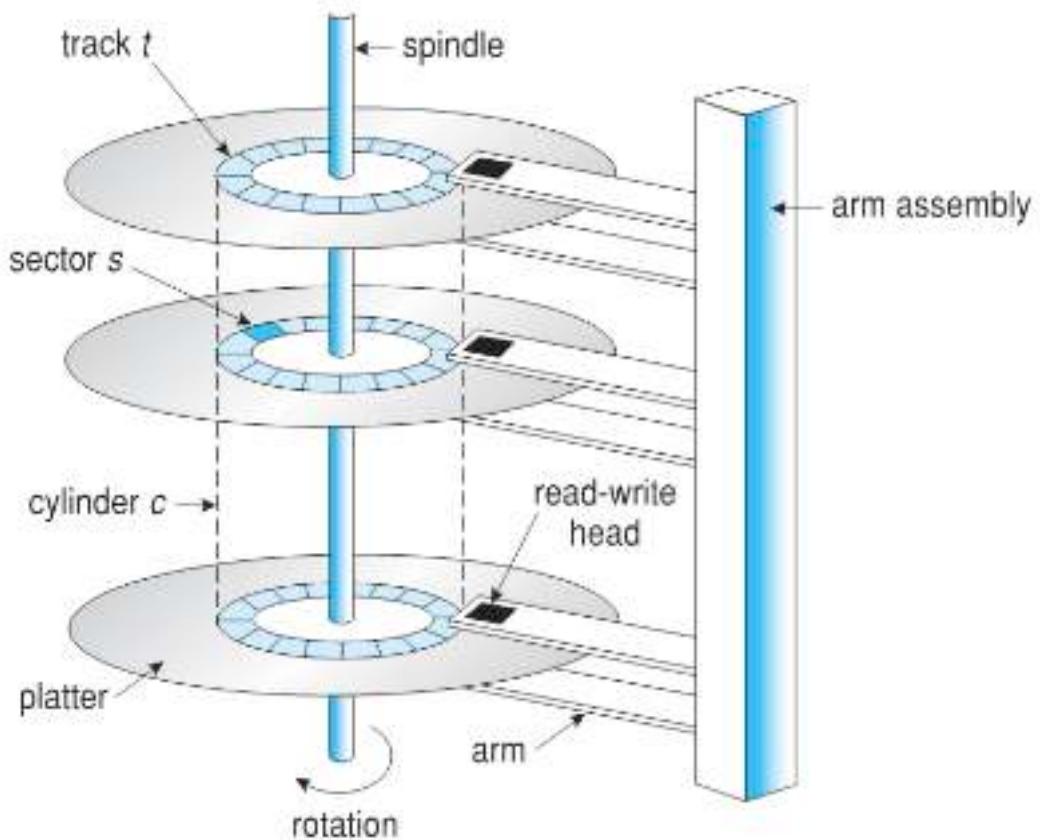
- Per molto tempi i sistemi vecchi usavano **i bit di parità**: quando si memorizza un byte di info, gli si associa una info aggiuntiva (bit parità) che registra se il numero di bit impostato a 1 è pari o dispari. Sapendo ciò posso sapere se si è verificato un errore: supponiamo che nel byte (8 bit) si è

corrotto (da 1 a 0 o viceversa) e la parità del byte è cambiata e non corrisponde più al byte stesso e mi accorgo dell'errore. Se invece si corrompe il bit di parità esso non corrisponde alla parità calcolata e ci si accorge dell'errore.

- Controllo di ridondanza ciclico CRC: si usa una funzione di hash per rilevare errori su più bit.

In ambo gli approcci si usa l'aritmetica modulo-2: si fanno addizioni senza resto e sono implementate come un OR_ESCLUSIVO.

STRUTTURA DI UN HARD DISK



Con un HD l'elemento di memorizzazione è un oggetto circolare detto “**piatto**”. Ogni piatto ruota intorno al proprio asse.

La superficie di ogni piatto è ricoperta da un materiale magnetico. Per poter leggere o scrivere su piatto ci sono delle testine sospese su ciascuna superficie. Queste testine sono montate su un braccio chiamato “**attuatore**”.

Le testine sono mosse dal braccio in blocco, si muovono tutte allo stesso modo. La superficie di ciascun piatto ha una struttura logica che vede il piatto diviso in tracce circolari: sono delle tracce concentriche che variano dalla parte più interna alla più esterna del disco e le info sono scritte lungo queste tracce.

Fissata una traccia, tutte le tracce su tutti i piatti costituiscono il “**CILINDRO**”. Ciascuna traccia per una questione di ottimizzazione viene divisa in un insieme di porzioni logiche chiamate settori.

In un unità disco ci possono essere migliaia di cilindri concentrici e ogni traccia a centinaia di settori.

La dimensione di settori è fissa ma varia da sistema a sistema.

Quando un disco è in funzione, tutti i piatti ruotano perché c'è un motore che li fa ruotare ad elevata velocità (Dai 60 ai 250 giri al secondo) e la velocità è espressa in giri al minuto.

DISCO MAGNETICO

La testina di lettura e scrittura di un piatto questa registra e legge sulla superficie. Un byte è memorizzato in modo seriale lungo una traccia circolare sulla superficie del disco.

Sui dischi non viene usata l'info di parità per l'individuazione degli errori ma il CRC.

Su ogni traccia è marcata la posizione di inizio traccia e ai record di una traccia sono attribuiti numeri seriali rispetto a tale posizione, il disco può accedere ad ogni record con un indirizzo (*numero traccia, numero record*).

- Il tempo di accesso è:

- $t_a = t_s + t_r$
- t_s **tempo di ricerca**, tempo per posizionare la testina sulla traccia richiesta
- t_r **latenza rotazionale**, tempo per accedere il record desiderato sulla traccia

La latenza rotazionale media è il tempo richiesto per una rivoluzione di metà del disco (3-4 ms).

È possibile aumentare la capacità dei dischi andando ad aggiungere dei piatti, ma se si usano più piatti servono più testine, l'attuatore è più pesante e sottopone il disco a maggiore stress meccanico.

Il disco è caratterizzato da un insieme di **cilindri**.

Un cilindro è costituito dall'insieme delle tracce con lo stesso indice sul piatto.

Tutte le tracce di un cilindro sono accedute dalla stessa posizione della testine poiché sono fisse.

L'uso del cilindro consente di ridurre il numero di movimenti della testina e i dati adiacenti di un file sono messi sulle tracce di uno stesso cilindro: se ho dati adiacenti li memorizzo sulle tracce dello stesso cilindro che possono essere lette con un solo movimento del braccio.

L'indirizzo di un record è caratterizzato dal numero di cilindro, numero di superficie, numero di record.

Per ottimizzare l'uso della superficie del disco le tracce sono organizzate in settori:

- Slot di dimensione fissa
- Contengono un record di info

La suddivisione in settori può essere fatta da parte dell'HW (**hard sectoring**) o SW (**soft sectoring**).

ORGANIZZAZIONE DEI DATI SU DISCO

È chiaro che se si vuole ottimizzare le operazioni di accesso del disco ecc... è fondamentale che i dati siano organizzati per garantire un accesso efficiente. I dati letti da un dispositivo di I/O sono memorizzati nel buffer del DMA e il DMA li trasferisce in memoria come un blocco singolo. Ciò comporta che mentre è in atto questo trasferimento il disco continua a girare, quindi significa che uno o più settori seguenti possono passare sotto la testina fino al momento in cui il trasferimento è completato. Ciò vuol dire che dobbiamo aspettare la prossima rivoluzione del disco per riposizionarci sul settore e continuare il trasferimento. Se viene eseguita un'operazione di lettura sul successivo settore abbiamo una perdita di tempo nell'attesa del prossimo giro e il tempo di trasferimento subisce un ritardo. Ecco perché bisogna organizzare bene i dati.

Lo stesso si ha con la commutazione della testina. Quando è necessario accedere alle testine posizionate su piatti differenti, questo tempo di commutazione perde dei settori e deve aspettare il prossimo giro. Lo stesso tempo di ricerca può generare un problema simile.

È necessario adottare tecniche di organizzazione dati su disco per non causare eccessivi ritardi nei tempi di trasferimento.

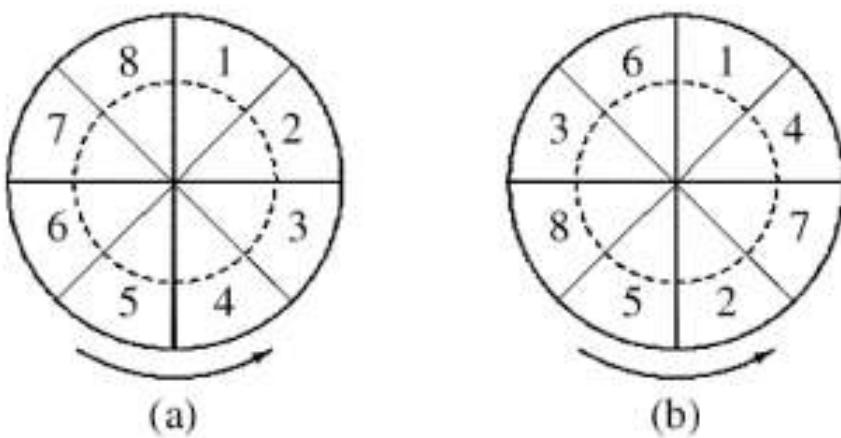
Per fare ciò si usano diverse tecniche:

TECNICA 1. Alternanza dei settori

Se l'operazione di lettura coinvolge la lettura di più settori, durante il trasferimento delle info contenute in un settore, per esempio dalla periferica alla memoria, il disco si muove e il successivo settore quando deve essere portato in memoria lo perdo e devo aspettare il prossimo giro.

L'idea è quindi di alternare i settori, i settori consecutivi non sono posti consecutivamente ma si alternano in modo che se il disco ruota ho la possibilità di avere quel settore sotto la testina.

In figura abbiamo i settori non alternati mentre a destra abbiamo alternanza a 2 quindi se sto leggendo 1 dopo un po' leggerò di nuovo il settore di 1 e si evita il problema di aspettare 1 giro completo.



Il numero di settore che si saltano prende il nome di *"fattore di alternanza"*.

TECNICA 2. Testina asimmetrica

Il disco richiede del tempo per commutare la lettura dei dati da una traccia a un'altra traccia (tempo di commutazione) e durante questo tempo il disco potrebbe perdere dei settori e devo aspettare il prossimo giro.

Con l'asimmetria della testina si organizzano le posizioni di inizio traccia su piatti diversi del cilindro in modo che lo sfasamento possa contrastare il movimento della testina e non perdere il settore considerato.

Per quanto riguarda l'asimmetria di cilindro, si organizzano le posizioni di inizio traccia su cilindri consecutivi, i dati sono resi asimmetrici come per la testina. Ciò minimizza il numero di giri per posizionarsi su un'operazione di lettura e scrittura.

DRIVER DI DISPOSITIVO

L'IOCS FISICO gestisce varie operazioni di I/O: avvio, completamento e gestione degli errori per tutti i dispositivi di I/O. Se fosse necessaria l'aggiunta di una nuova classe di dispositivi, **si dovrebbe modificare l'IOCS**. Se l'IOCS fa parte del kernel, modificare l'IOCS sarebbe complesso.

I moderni SO fanno uso dei **driver di dispositivo**. L'IOCS fisico fornisce solo un supporto generico per le operazioni di I/O rimandando le operazioni al driver del dispositivo per una specifica classe. Ciò significa che il driver di dispositivo non fa parte dell'IOCS fisico perché in questo modo è sempre possibile aggiungere nuovi dispositivi senza modificare l'IOCS fisico. I driver bisogna averli in memoria e sono caricati o in fase di boot o all'occorrenza del SO.

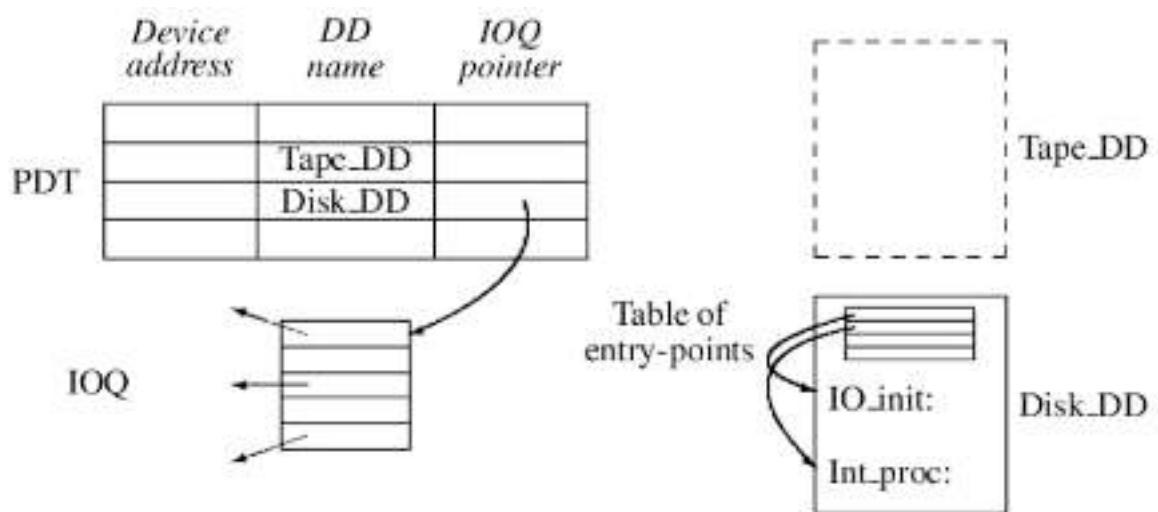
Esempio: i dischi sono caricati in fase di boot mentre la penna USB in fase del funzionamento del SO. Ciò è possibile perché i driver sono esterni all'IOCS fisico.

Perché l'IOCS fisico possa basarsi su driver di dispositivo è necessario usare strutture dati coinvolte nella interazione tra IOCS fisico e driver di dispositivo.

- In figura abbiamo una tabella di dispositivi fisici **PDT** dove ogni entrata contiene il nome del driver **DD NAME** e un puntatore a un'altra struttura chiamata **CODA DI I/O** dove per ciascuna classe del dispositivo l'IOCS fisico inserisce la specifica operazione da effettuare.

I driver sono caricati in memoria o in fase di boot o all'occorrenza. Il driver del disco è in memoria mentre il driver del nastro è caricato all'occorrenza. Il driver di dispositivo contiene delle funzioni che svolgono le operazioni di avvio, completamento ecc di operazioni di I/O. queste funzioni sono all'inizio della tabella che contiene gli

indirizzi per ciascuna funzione per quel driver di dispositivo. Quando l'IOCS è richiamato dalla CPU, l'IOCS fisico localizza la specifica periferica nella PDT ed esegue un'operazione generica di inserimento di dettagli nella **IOQ (Coda di I/O)**, consulta il campo nome del driver di dispositivo, ottiene l'identità del dispositivo e lo carica in memoria se non è presente. Recupera l'indirizzo del punto di ingresso relativo all'operazione di I/O e passa il controllo al driver del dispositivo che intercede col **DMA** per completare le operazioni.



SCHEDULING DEL DISCO

L'IOCS fisico e i driver di dispositivo adottano una politica di scheduling per ottimizzare il throughput, bisogna massimizzare il lavoro da fare sul disco e minimizzare i movimenti delle testine lungo il disco.

Esistono diversi algoritmi di scheduling del disco:

- **FIFO**: la prima richiesta è la prima servita.
- **Shortest seek time first**: serve la richiesta di accesso che minimizza il tempo della testina.

- **SCAN**: muove le testine da un estremo all'altro di un piatto servendo le richieste di I/O man mano che le incontra, quando si **arriva alla fine del piatto si riparte**.
 - **LOOK**: *variante di SCAN*: piuttosto che arrivare alla fine del piatto si arriva all'ultima della direzione e si ritorna indietro.
 - **CSCAN** (*C sta per circolare*): nel momento in cui servo le richieste passando da un estremo all'altro del disco, la direzione del movimento arrivata alla fine del piatto non viene invertita **ma** si ricomincia da capo.
 - **C-LOOK**: arrivo all'ultima richiesta e ricomincio da capo.
-

RICHIESTE DI I/O PER LO SCHEDULING DEL DISCO

$$t_{\text{hm}} = t_{\text{const}} + | \text{track}_1 - \text{track}_2 | \times t_{\text{pt}}$$

t_{const} and t_{pt}	= 0 ms and 1 ms, respectively
Current head position	= Track 65
Direction of last movement	= Toward higher numbered tracks
Current clock time	= 160 ms

Requested I/O operations:

Serial number	1	2	3	4	5
Track number	12	85	40	100	75
Time of arrival	65	80	110	120	175

Supponiamo di avere 5 operazioni di I/O su un disco composto da 200 tracce (le tracce sono concentriche che partono dall'interno all'esterno di ogni piatto). Le richieste sono rappresentate nella tabella, con un ID e per ciascuna richiesta abbiamo la traccia da accedere e l'istante temporale in cui avviene la richiesta.

Consideriamo poi la formula: t_{hm} rappresenta il tempo totale che impiega una testina per spostarsi dalla traccia 1 alla traccia 2 che è dato dal tempo t_{Const} e dal prodotto della differenza della traccia 1 e 2 per t_{pt} che è il tempo per spostare la testina da una traccia all'altra. Supponiamo che t_{pt} è 1ms e che la testina si trovi alla traccia 65 e che l'istante in cui è stata completata la precedente richiesta di I/O è 160ms.

La cosa importante è che le testine si muovono verso le tracce con numeri maggiori.

Vediamo come funzionano gli algoritmi di scheduling del disco.

DETTAGLI SCHEDULING DEL DISCO.

Policy	Details	Scheduling decisions					Σ Seek time
		1	2	3	4	5	
FCFS	Time of decision	160	213	286	331	391	256
	Pending requests	1, 2, 3, 4	2, 3, 4, 5	3, 4, 5	4, 5	5	
	Head position	65	12	85	40	100	
	Selected request	1	2	3	4	5	
	Seek time	53	73	45	60	25	
SSTF	Time of decision	160	180	190	215	275	143
	Pending requests	1, 2, 3, 4	1, 3, 4, 5	1, 3, 4	1, 3	1	
	Head position	65	85	75	100	40	
	Selected request	2	5	4	3	1	
	Seek time	20	10	25	60	28	
Look	Time of decision	160	180	195	220	255	123
	Pending requests	1, 2, 3, 4	1, 3, 4, 5	1, 3, 5	1, 3	1	
	Head position	65	85	100	75	40	
	Selected request	2	4	5	3	1	
	Seek time	20	15	25	35	28	
C-Look	Time of decision	160	180	195	283	311	186
	Pending requests	1, 2, 3, 4	1, 3, 4, 5	1, 3, 5	3, 5	5	
	Head position	65	85	100	12	40	
	Selected request	2	4	1	3	5	
	Seek time	20	15	88	28	35	

Req	Track	Time
1	12	65
2	85	80
3	40	110
4	100	120
5	75	175

$$t_{hm} = t_{const} + | \text{track}_1 - \text{track}_2 | \times t_{pt}$$

- **FCFS** (first cum first served) → all'istante 160 ci sono 4 richieste pendenti. (1,2,3,4). La posizione iniziale della testina è 65. FCFS serve chi è arrivato prima quindi la richiesta 1 (a 65ms) che richiede di accedere alla traccia 12. Ci dobbiamo spostare dalla traccia 65 alla traccia 12 e abbiamo detto che spostarsi da una traccia all'altra costa 1 ms quindi $65-12=53$ ms tempo di ricerca. Dopo aver servito alla richiesta 1 passiamo al secondo passo di scheduling. Siamo partiti dall'istante 160 e siamo arrivati a 53 quindi al secondo passo il tempo di decisione sarà $160+53=213$. A 213 abbiamo 4 richieste pendenti (2,3,4,5) e FCFS va a servire la richiesta che è arrivata prima, la seconda. La testina si trova su 12. Due ci chiede la traccia 85 quindi $85-12=73$ tracce → tempo di ricerca. Al passo 3 abbiamo 286 ($213+73$) serviamo la richiesta 3. La testina si trova sulla traccia 85 e dobbiamo arrivare alla traccia 40: tempo di ricerca è 45. Al passo 4 abbiamo $286+45=331$ e andiamo a servire la 4 richiesta. La testina si trova su 40 e dobbiamo arrivare a 100 e il tempo di ricerca sarà 60. All'ultimo passo abbiamo $331+60=391$ con la richiesta 5 che viene servita. Si accede alla traccia 75, da 100 il tempo di ricerca è 25. Il tempo di ricerca totale è la somma dei tempi di ricerca (256 ms).
- **SSTF** (Shortest time first) → minimizza il tempo di ricerca. Tempo 160 e 4 richieste pendenti. Andiamo a servire quella col tempo di ricerca minore, trovandoci nella testina 65 andiamo a servire la richiesta numero 2 che vuole accedere a 85, avendo uno scostamento di 20 tracce e viene servita la richiesta numero due. Al secondo passo abbiamo $160+20=180$ con 4 richieste pendenti. Ci troviamo nella

traccia 85 e andiamo a servire la traccia 5 che richiede la traccia 75, il tempo di ricerca sarà 10. Al terzo passo abbiamo 190 con 3 richieste pendenti e andiamo a servire la richiesta 4 che accede alla traccia 100 e il tempo di ricerca sarà 25 e così via...

Il tempo di ricerca totale sarà 143 ms.

- **Look** (come lo scan ma arrivato all'ultima traccia del piatto torno indietro). Siamo a 160 con la traccia 65. Se mi muovo con la traccia di numero crescente incontro la richiesta 2. Se servo la seconda avrò un tempo di ricerca di 20 secondi. Al secondo passo siamo a 180 con 4 richieste pendenti, dopo 85 incontro la traccia 100 (richiesta 4) e il tempo di ricerca è 15. A questo punto ci troviamo alla traccia 100 e l'algoritmo tiene conto delle richieste e numero di traccia e sa che deve tornare indietro. Al passo 3 torna indietro e con le richieste pendenti 1,3,5 va a incontrare la traccia 75 che avrà un tempo di ricerca di 25 e si continua a ritroso...

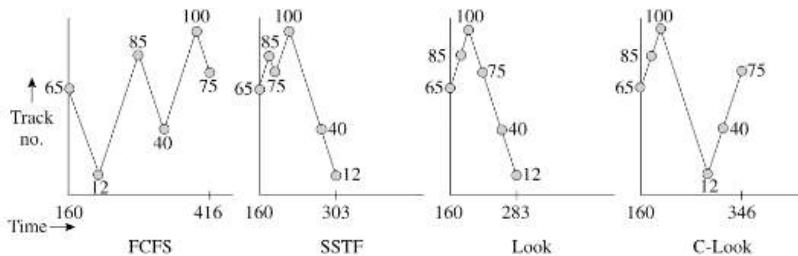
Il tempo di ricerca totale è 123 ms.

- **C-Look:** funziona come look e quando siamo arrivati alla decisione numero 3 (ultima traccia lungo la direzione), invece di ritornare indietro ricomincia da capo, da 100 ci spostiamo alla prima traccia che incontriamo ricominciando da capo, cioè la 1, poi la 3 e poi la 5.

Il tempo di ricerca totale è 186 ms.

Deduciamo che per questo esempio specifico l'algoritmo migliore è il LOOK. Vediamo il grafico.

Prestazioni degli algoritmi di scheduling



In generale non sappiamo il miglior metodo ma empiricamente il look e il c-look sono i migliori.

TEMPI DI ACCESSO NELLO SCHEDULING DEL DISCO

- Supponiamo che nella coda delle richieste di un'unità disco composta da 200 tracce si trovano le richieste di dati nei blocchi
 - 39700 – 304 – 115 – 2600 – 2120 – 270 – 321 – 0 – 760 – 20000
 - il blocco *i-esimo* è memorizzato nella traccia *i mod 200*
- La testina ha eseguito l'ultimo movimento portandosi dalla traccia 85 alla traccia **97**
- Si ipotizzi che lo spostamento da una traccia ad un'altra richieda tempo medio pari a 40 μ s per traccia, che l'inversione della direzione di movimento richieda in media 80 μ s e la velocità di rotazione sia di 7200 giri
- Si vuole determinare il **tempo richiesto, complessivamente, per accedere alle tracce** indicate per le politiche SSTF, C-SCAN e LOOK.

Soluzione

- Latenza rotazionale: $60/(2 \times 7200) = 4.17 \text{ ms}$
- Dobbiamo determinare la traccia alla quale si trova il blocco (**i mod 200**)
- Blocchi: 39700 – 304 – 115 – 2600 – 2120 – 270 – 321 – 0 – 760 – 20000
- Tracce: 100 – 104 – 115 – 0 – 120 – 70 – 121 – 200 – 160 – 0

- SSTF.** La sequenza di scheduling è:
 - 97 100 104 115 120 121 160 200 70 0
 - Le distanze tra tracce della sequenza sono: 3 4 11 5 1 39 40 130 70
 - Il tempo di accesso è $t_a = (303 \times 40 \mu\text{s}) + 80 \mu\text{s} + (9 \times 4.17 \text{ ms}) = 12.12 \text{ ms} + 0.08 \text{ ms} + 37.53 \text{ ms} = 49.73 \text{ ms}$

- 2. C-SCAN
 - 97 100 104 115 120 121 160 200 0 70
 - Le distanze tra le tracce
 - 3 4 11 5 1 39 40 200 70
 - $t_a = (373 \times 40 \mu s) + 80 \mu s + (9 \times 4.17 ms) = 14.92 ms + 0.08 ms + 37.53 ms = 52.53 ms$
 - 3. LOOK
 - 97 100 104 115 120 121 160 200 70 0
 - 3 4 11 5 1 39 40 130 70
 - $t_a = (303 \times 40 \mu s) + 80 \mu s + (9 \times 4.17 ms) = 12.12 ms + 0.08 ms + 37.53 ms = 49.73 ms$
-

Codice filosofi a cena

Algoritmo:

```
var success:bool
repeat
    success = false
    while(not success)
    if entrambe le forchette sono disponibile then
        prendi le forchette uno alla volta
        success = true
    if success = false then
        block(Pi)
    {mangia}
    Posa le fork [in mutua esclusione]
    If il vicino di sinistra è in attesa della forchetta destra
    Then activate (vicino di sinistra)
    If vicino di destra in attesa della forchetta sinistra
        Then activate(vicino di destra)
    {pensa}
forever
```

Come vengono gestiti gli eventi tramite lo scambio di messaggi con ECB (message passing)?

Come vengono gestiti gli eventi tramite lo scambio di messaggi con ECB:
Quando si invia o si riceve un messaggio viene creato un evento.

Es: Sono un processo Pi che deve inviare un msg ad un processo Pj e lo faccio tramite una send non bloccante (così lo invio e non mi blocco nell'aspettare che Pj lo riceva).

Il Kernel costruisce in questo caso una struttura IMCB (interprocess message control block) per memorizzare le informazioni necessarie per la consegna del msg sul buffer del kernel associato a IMCB (vengono memorizzate temporaneamente).

Quando Pj invocherà receive allora il kernel non fa altro che prendere il messaggio memorizzato nel buffer che sta allocato all'IMCB e lo mette nell'area di memoria specificata come secondo parametro di receive(...,msg).

Siccome i msg possono essere più di uno nella struttura dati IMCB c'è un puntatore che serve per costruire una lista di messaggi di IMCB che serve per lo scambio di messaggi.

Le liste di IMCB per send bloccanti possono essere fatte con indirizzamento diretto e indirizzamento indiretto.

- Con l'indirizzamento diretto ogni coppia di Processi che comunicano crea una lista di msg tramite gli IMCB. Con la *send* il processo indica esplicitamente l'id del destinatario. Per quanto riguarda la *receive* non può specificare in anticipo da chi riceverà il messaggio, la soluzione è l'indirizzamento indiretto.
- Con l'indirizzamento indiretto per ogni Processo ho la relativa lista di msg IMCB da cui devo attingere. E quando invoco receive un elemento da IMCB viene rilevato. I messaggi sono inviati in una struttura dati condivisa, mailbox.
- MAILBOX: ha un nome univoco e il proprietario è il processo che l'ha creata. Tutti i processi che intendono inviare messaggi usano l'id della box.

ECB:

per fare la consegna dei messaggi tra processi, visto che i tempi nei quali i processi mittente e destinatario invocano send e receive sono asincroni, il kernel deve gestire la consegna in caso il msg sia recapitabile oppure no. Entrano in gioco gli ECB.

esempio: utilizzo degli ECB per implementare la tecnica a indirizzamento diretto con chiamate bloccanti send e receive. Come si implementa?

- a) sono il Pi mittente che invia un msg con una send bloccante e rimango bloccato in attesa che il msg venga consegnato. Il processo Pj invoca receive, che succede? Il kernel va a cercare nella classe di eventi legata agli scambi dei msg, lo trova, lo estrae, prende l'identificativo del processo in attesa dell'evento (ci sarà l'id di Pi bloccato sull'evento) e cambia il suo stato da blocked a ready e poi avviene la consegna del msg.
- b) succede lo stesso per Pj con receive che rimane bloccato se non è stato mandato nessun msg da Pi. In questo caso l'evento è che Pi deve inviare il msg a Pj per far controllare al kernel l'evento nell'ECB.

Contenuto ECB: 1) descrizione dell'evento anticipato, 2) identificativo del processo in attesa dell'evento, 3) un puntatore a un ECB per la creazione delle liste di ECB (posso avere più eventi dello stesso tipo che si verificano).

N.B: il kernel contiene una lista separata degli ECB per ogni classe di evento (msg interprocesso, operazioni di I/O, etc) in modo tale che il campo puntatore ECB venga utilizzato per inserire l'ECB appena creato nella lista appropriata degli ECB. L'ECB viene creato al blocco del processo e viene inserito nella lista apposita. Quando si verifica l'evento che il P bloccato stava aspettando, il kernel esamina la lista appropriata per trovare un ECB con una descrizione dell'evento corrispondente e farà le operazioni indicate in (a).