

Cosa sono?	8
A cosa serve?	8
Astrazioni	9
Programmi event-driven	9
Memoria centrale (RAM)	10
Memoria secondaria	10
Dispositivi di I/O	10
Architetture degli elaboratori	11
Cluster di elaboratori	11
Caratteristiche del SO	11
Dual mode	11
System call	12
Programmi e processi	12
Controllo dei processi	13
Gestione dei file	13
Gestione di device e informazioni	13
Comunicazione	13
Operazioni e chiamate di sistema	14
Gestione delle memoria principale e secondaria	14
Memoria e file system	15
SO come ambiente di lavoro	15
Interfacce utente	15
Implementazione di un SO	16
Stratificazione	16
Stratificazione e macchine virtuali	17
Microkernel	17
Tecnica a moduli	18
Processi	18
Parallelismo virtuale	19
Categorie di processi	19

Diagramma di transizione degli stati di un processo	20
PCB	20
Commutazione della CPU	21
Scheduling	21
Scheduling a breve termine	21
Algoritmi di scheduling	22
Criteri di scheduling	22
Diagrammi di Gantt	23
First-come-first-served	23
Prelazione	23
Shortest job first	24
Scheduling a Priorità	24
Starvation	24
Round Robin	24
Code a multilivello con feedback	25
Creazione	25
Terminazione	25
Processi cooperanti	26
Produttore / consumatore	26
Memoria condivisa	26
Inconsistenza dei dati	27
Scambio di messaggi	27
Socket	28
Remote procedure call	28
Pipe	28
Thread	29
Operazioni sui thread	30
Thread utente	30
Thread kernel	30
Associazione dei thread kernel	31

Lightweight process	31
Scheduling della CPU	32
Esecuzione concorrente asincrona	32
Sincronizzazione hardware	34
TestAndSet	34
Swap	35
Attesa limitata	36
Semafori	36
Inizializzazione e uso	37
Tipi di sincronizzazione	38
Operazioni atomiche	38
Problema: produttori-consumatori	38
Problema: lettori-scrittori	39
Problema: cinque filosofi	41
Chandy-Misra	42
Uso errato dei semafori	42
Monitor	42
Signal e M.E.	44
Transazione	44
Memoria volatile	44
Ripristino/Abort	45
Tutto il logfile?	45
Transazioni atomiche concorrenti	45
Serializzabilità	45
Protocollo di gestione dei lock	45
Gestione dei lock a due fasi	46
Protocolli basati su timestamp	46
Regole che definiscono il protocollo	46
Deadlock	46
Grafo di assegnazione delle risorse	47

Un caso reale: spooling	48
Che fare col deadlock?	48
Prevenzione del deadlock	48
1a strategia di Havender	48
2a strategia di Havender	49
3a strategia di Havender	49
Deadlock avoidance	49
Stato di allocazione delle risorse	50
Sequenza di esecuzione	51
Stati sicuri e sequenze sicure	51
Algoritmo di deadlock avoidance	52
Algoritmo del banchiere (avoidance)	52
Algoritmo di verifica della sicurezza	53
Algoritmo di gestione delle richieste	53
Rilevamento del deadlock per istanze singole di risorsa	54
Rilevamento del deadlock per istanze multiple di risorse	54
Rottura del deadlock	55
Soluzione 1: terminazione	55
Soluzione 2: prelazione	56
Far finta di niente	56
Indirizzi logici e fisici	56
Binding	56
Linking e loading	57
Linking e loading dinamici	57
Loading dinamico	57
Linking dinamico	57
Allocazione della RAM	58
Allocazione contigua	58
Allocazione a partizioni multiple	58
Criteri di scelta	59

Combattere la frammentazione	60
Swapping	60
Tempo di swapping	60
Paginazione della memoria	61
Pagine e strutture di supporto	62
Paginazione e indirizzi logici	62
Paginazione e rilocalizzazione	63
Paginazione e frammentazione	63
Architettura di paginazione	64
Tempi di accesso	64
Hit ratio	64
Aggiornamento del TLB	65
Protezione	65
Condivisione delle pagine	66
Paginazione multi-livello	67
Paginazione a due livelli	67
Vantaggi e svantaggi delle tabelle delle pagine	68
Segmentazione	68
Memoria virtuale	70
Spazio degli indirizzi di un processo	70
Lazy swapping	70
Bit di validità	71
Page fault	71
Paging on demand	71
Gestione del page fault	71
Area di swap	72
Processi padri e figli	72
Sostituzione di pagine	72
Implementazione della paginazione a richiesta	73
Sostituzione FIFO	73

Algoritmo ottimale	74
Least-recently used	74
Algoritmo con bit supplementare	74
Algoritmo di seconda chance	75
Algoritmo di seconda chance migliorato	75
Sostituzione su conteggio	76
Pool of free frames	76
Allocazione dei frame per i processi utente	77
Allocazione proporzionale	77
Dinamicità	77
Allocazione globale/locale	77
Thrashing	77
Effetti del thrashing	78
Thrashing e multiprogrammazione	78
Pagine attive e Working Set	79
Prepaginazione	79
Page fault frequency	80
Memoria virtuale e file	80
Mappatura dei file	80
Da RAM a disco	80
Mappatura I/O	81
Utente	81
Allocazione dei frame per il kernel	81
Sistema buddy	81
Allocazione a slab	82
File system	82
Tipi di file	83
Organizzazione del file system	83
Apertura e chiusura di un file	83
Lettura e scrittura su file	84

Protezione dei file	84
Directory	84
Directory con grafo aciclico	86
Link	86
Directory a grafo generale	87
Mount	87
File system distribuiti	88
In-core inode	88
Apertura di un file (inode)	89
Algoritmo NAMEI	89
Open di un file in Unix	90
Implementazione delle directory	90
Sequenza lineare	90
B-tree	90
Allocazione dello spazio disco ai file	91
Allocazione contigua	91
Allocazione concatenata	92
Allocazione concatenata FAT	93
Allocazione indicizzata	93
Gestione dello spazio libero	94
Quantità massima di memoria gestibile	94
Implementazione del file system	94
Livelli del file system	94
Struttura del disco	95
INODE / FCB	95
✨Glossario✨	95

Sistemi Operativi

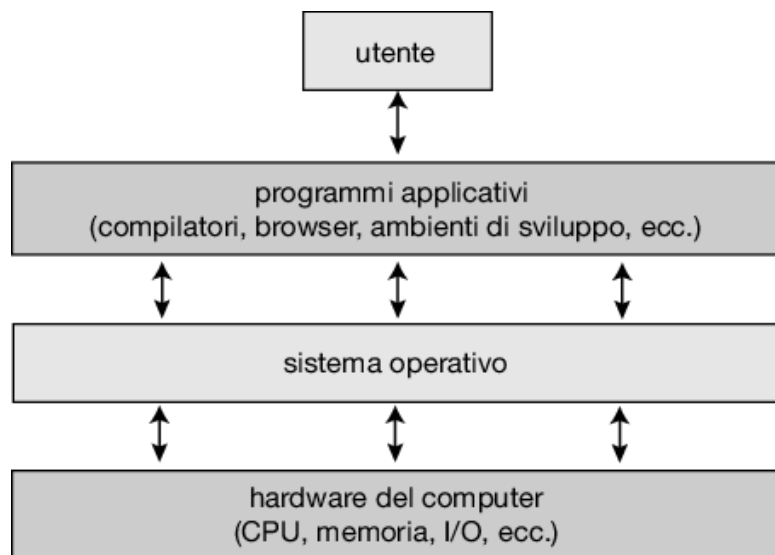
Cosa sono?

Permettono di usare l'hardware attraverso programmi.

In genere dividiamo un computer su 3 livelli:

- Livello 0: dispositivo fisico
- Livello 1: sistema operativo
- Livello 2: programmi

A cosa serve?



E' principalmente un **gestore di risorse**, ma è anche un **controllore** che gestisce l'esecuzione dei programmi applicativi.

Le risorse costituiscono il sistema e consentono la risoluzione di problemi. Possono essere di 2 tipi:

- **hardware:** CPU, RAM, I/O...
- **software:** code di messaggi, processi, thread...

Il SO gestisce molte cose:

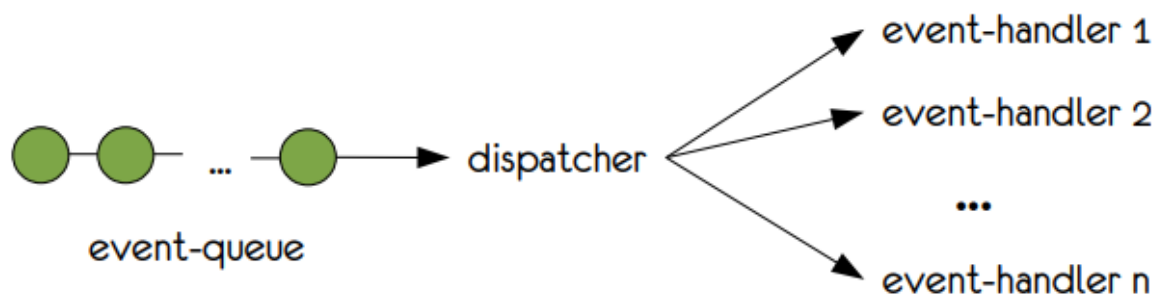
- esecuzione di programmi
- accesso alle risorse
- gestione di utenti
- sincronizzazione tra processi
- ecc..

Astrazioni

Il SO definisce delle "astrazioni", ovvero delle rappresentazioni dei vari elementi del SO stesso, in modo che siano di facile comprensione da parte dell'utente. In generale il SO

definisce delle **astrazioni software** di tutti i tipi di oggetti che occorre rappresentare e gestire per far funzionare un computer (compresi gli utenti), implementa opportuni algoritmi di controllo e contiene interfacce sia verso gli utenti sia verso i dispositivi fisici.

Programmi event-driven



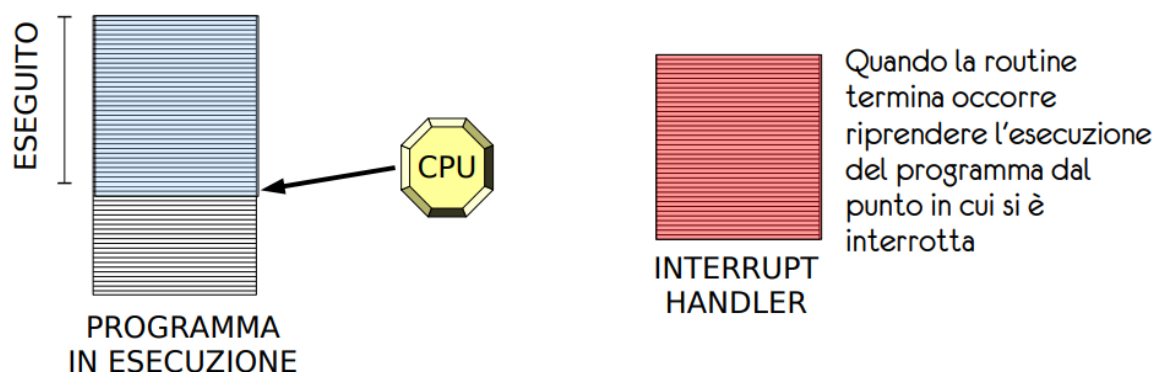
Sono dei programmi che eseguono diverse routine basandosi su determinate situazioni chiamate **eventi**. A ogni evento corrisponde una routine di gestione predefinita detta **“event-handler”**.

La gestione degli eventi deve essere molto efficiente in un SO.

Gli eventi possono essere accumulati in una coda, e quando vengono gestiti si esegue un **“dispatch”**, che consiste nell’individuare ed eseguire la routine associata all’evento. È possibile implementare il dispatch attraverso un vettore di puntatori agli event handler: basta associare a ogni evento un numero (id dell’evento e al contempo indice nel vettore). Il vettore è detto **vettore delle interruzioni** (interrupt vector).

La gestione degli interrupt causa un **context switch**:

1. SO sospende il processo e salva le sue informazioni in RAM
2. carica nei registri CPU le informazioni necessarie per l’esecuzione dell’handler



Memoria centrale (RAM)

E' l'unica memoria di grandi dimensioni direttamente accessibile dalla CPU. Queste interazioni avvengono tramite istruzioni load (RAM -> registro) e store (registro -> RAM).

E' molto più **veloce** di una memoria secondaria, ma è anche **volatile**, ovvero viene cancellata allo spegnimento della macchina, e ha una **capacità limitata** a causa dell'elevato costo di produzione.

Memoria secondaria

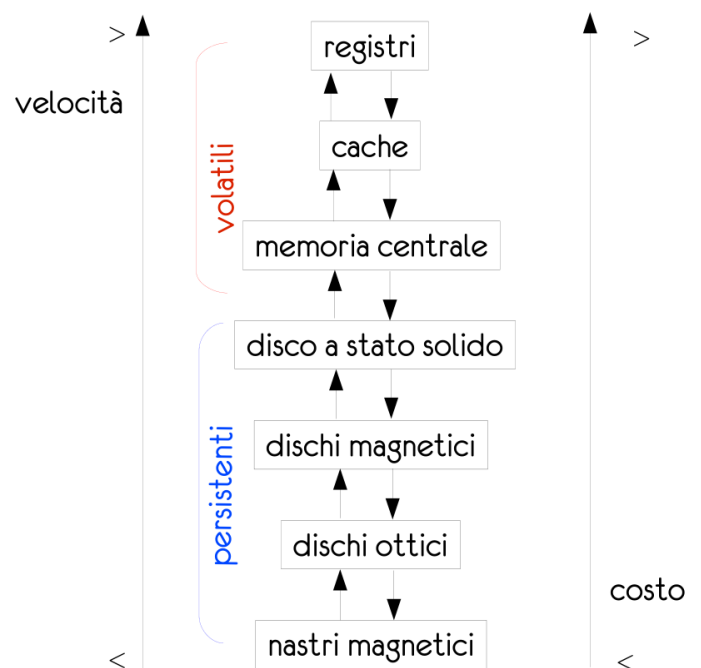
È una **memoria permanente**, con capacità elevata. In genere si tratta di un disco magnetico, ma gli SSD, nastri, CD/DVD/BD. Si differenziano in base al costo e la velocità di accesso.

Dispositivi di I/O

I dispositivi di input/output vengono gestiti da un certo numero di **controller**, tutti connessi da un bus.

Ogni controller:

- gestisce uno o più dispositivi;
- ha una memoria interna (**buffer** e **registri**);
- ha un software chiamato **driver** che si interfaccia col SO;
- usufruiscono della tecnica DMA (Direct Memory Access) per operazioni più rapide.



Architetture degli elaboratori

Gli elaboratori possono essere divisi in 2 categorie:

- **Monoprocessore (Single Core)**: possiedono una singola CPU, ma possono presentare processori ausiliari dedicati ad attività specifiche (es: controller)
- **Multiprocessore (Multicore)**: posseggono due o più CPU e godono di maggiore capacità elaborativa, maggiore affidabilità e gestione della memoria più efficiente, ma possiedono un costo di risorse HW più elevato legato alla gestione e sincronizzazione dei processori.

La scelta architetturale può dipendere sia dall'hardware che dal sistema operativo.

Cluster di elaboratori

Sono sistemi multiprocessore costituiti da insiemi di elaboratori completi (chiamati nodi). Ogni nodo può essere single o multi-core.

C'è bisogno di un software che gestisce il cluster, per controllare e sistemare i possibili crash e per dividere il carico sui nodi.

Caratteristiche del SO

1. **Multiprogrammazione**: il SO gestisce contemporaneamente un insieme di processi (job, task) distribuendo l'utilizzo della CPU fra i vari processi;
2. **Multitasking**: estensione della multiprogrammazione in cui si tiene conto dell'interazione con l'utente, facendo in modo che esso abbia la percezione che solo il suo job sia in esecuzione (**parallelismo virtuale**)

Entrambi avvengono tramite lo **scheduling della CPU**.

La gestione di quali processi spostare in memoria centrale e viceversa è detta **job scheduling**.

Dual mode

Politica di protezione del SO che permette ai processi di utilizzare un Instruction Set "sicuro", che non li lascia eseguire modifiche drastiche alla memoria (i processi possono solo accedere all'area di memoria a loro assegnata).

In Dual Mode, tramite un **bit di modalità**, è possibile decidere quale IS utilizzare:

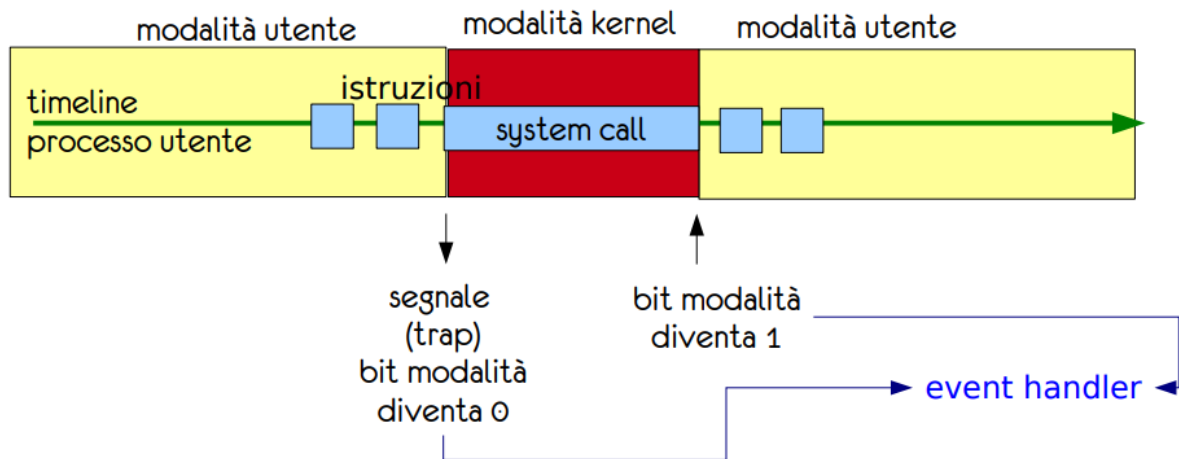
- 0 → modalità **kernel** (o supervisore): accesso all'intero IS
- 1 → modalità **utente**: non si ha accesso alle istruzioni di I/O

Al bootstrap, il bit di modalità è inizializzato a 0. Con la richiesta di esecuzione di processi utente, a seconda del processo, il bit di modalità cambierà valore.

Il modello a livelli è **possibile solo se la CPU fornisce a livello HW un supporto adeguato**.

System call

Comandi che possono essere utilizzati dai processi per eseguire indirettamente (le op. non le esegue il processo ma il SO) operazioni di I/O. Durante l'esecuzione di una system call, il bit di modalità vale 0.



Come controllare quante syscall eseguono i processi in uso:

- `ls`: elenca i file contenuti nella directory di lavoro
- `strace`: comando che consente di listare le system call eseguite da un processo e i segnali gestiti dal medesimo
- `strace -wc ls`: riassume in una tabella finale quanto intercettato

Programmi e processi

Un programma in sé è un file, un'entità passiva che non fa nulla. La sua esecuzione produce un'entità attiva detta **processo**.

Un programma può avere diversi processi che possono portare ad eseguire task diversi, ognuno con una propria area di memoria che può essere anche condivisa.

Di ogni processo viene mantenuta un'astrazione.

Il SO deve gestire i processi:

- identificazione;
- creazione e cancellazione;
- sincronizzazione e comunicazione;
- identificare e gestire il deadlock;
- evitare starvation.

NB: anche il SO è un processo.

Controllo dei processi

La terminazione dei processi è gestita da un system call, e la terminazione anormale può causare la copiatura dell'immagine del processo (dump o core). In questa maniera si può eseguire il codice passo per passo, per vedere cosa sia successo.

Un processo può anche fare delle azioni:

- caricare un programma diverso da quello originario
- generare un altro processo
- può essere sospeso per un lasso di tempo o in attesa di un evento (per esempio, un segnale)

Un processo, poi, ha delle caratteristiche, che possono essere reimpostate via system call

Particolarità:

- il SO “convive” con gli altri processi utente, perciò anche esso ha bisogno di utilizzare la CPU;
- il SO può riprendere la CPU assegnata ad un processo utente tramite l'utilizzo di **timer predefiniti**, che causano un interrupt del processo.

Gestione dei file

Le operazioni principali per la gestione dei file che il SO esegue sono:

- creazione/cancellazione
- apertura/chiusura
- lettura/scrittura
- riposizionamento/spostamento
- copiatura
- lettura/modifica delle proprietà

Gestione di device e informazioni

Ci sono varie operazioni che si possono fare sui dispositivi:

- request/release di uno
- lettura/scrittura/riposizionamento

Informazioni: sono system call che trasferiscono informazioni come la data e l'ora, informazioni sui processi o informazioni sui file all'utente o a suoi programmi.

Comunicazione

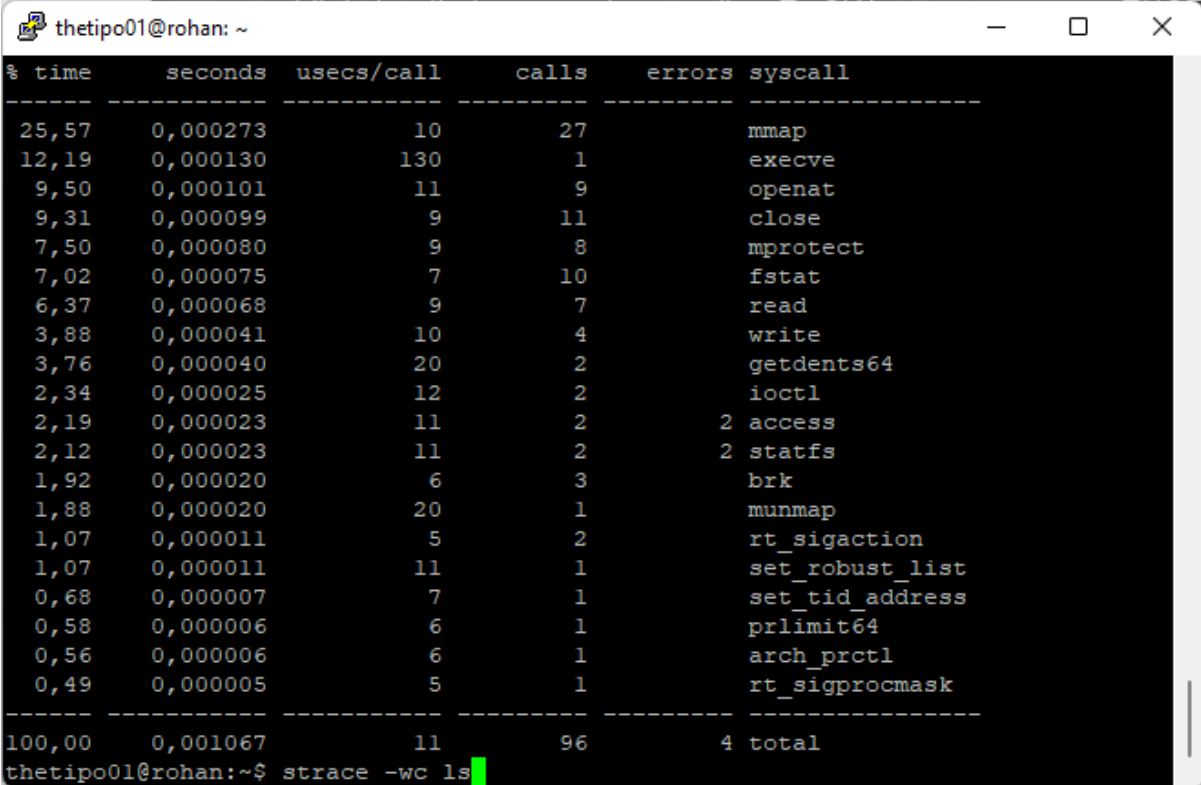
Ci sono 2 modelli principali per la comunicazione tra processi:

- **Scambio di messaggi:**
 - identificare macchina (get host id) e processo (get process id)
 - mittente: apertura connessione
 - destinatario: accettazione della connessione
 - scambio messaggio
 - chiusura connessione
- **Memoria condivisa**
 - allocazione di un'area di memoria condivisa
 - aggancio (attach) di un'area di memoria condivisa

Operazioni e chiamate di sistema

Anche una semplice operazioni di copia su una shell (`cp origine.odt destinazione.odt`) genera moltissime system call.

Con il comando `strace -wc <comando da eseguire>` si possono vedere tutte le system call che il comando specificato ha eseguito



% time	seconds	usecs/call	calls	errors	syscall
25,57	0,000273	10	27		mmap
12,19	0,000130	130	1		execve
9,50	0,000101	11	9		openat
9,31	0,000099	9	11		close
7,50	0,000080	9	8		mprotect
7,02	0,000075	7	10		fstat
6,37	0,000068	9	7		read
3,88	0,000041	10	4		write
3,76	0,000040	20	2		getdents64
2,34	0,000025	12	2		ioctl
2,19	0,000023	11	2	2	access
2,12	0,000023	11	2	2	statfs
1,92	0,000020	6	3		brk
1,88	0,000020	20	1		munmap
1,07	0,000011	5	2		rt_sigaction
1,07	0,000011	11	1		set_robust_list
0,68	0,000007	7	1		set_tid_address
0,58	0,000006	6	1		prlimit64
0,56	0,000006	6	1		arch_prctl
0,49	0,000005	5	1		rt_sigprocmask
100,00	0,001067	11	96	4	total

thetipo01@rohan:~\$ strace -wc ls

Gestione delle memoria principale e secondaria

La memoria principale (RAM) è limitata.

Il SO si deve occupare di:

- ricordarsi chi sta usando quale parte di memoria e quale parte è invece libera
- assegnare/revocare lo spazio a seconda delle necessità
- decidere quali processi vanno trasferiti in RAM e quali vanno rimossi

Il SO gestisce anche la memoria secondaria (HDD):

- assegnazione dello spazio
- scheduling del disco
- swapping

Memoria e file system

Gli utenti vedono la memoria organizzata in **file**, unità logiche di archiviazione di solito organizzati in **directory**.

Quando gli utenti utilizzano i nomi dei file e i loro “cammini” (path), il SO deve essere in grado di identificare i blocchi di memoria ad essi corrispondenti: perciò ogni file mantiene delle **proprietà** che aiutano al SO ad identificarli (es: tipo di file, permessi di accesso, data di creazione/modifica...).

SO come ambiente di lavoro

Il SO è in grado di offrire diversi servizi:

- **Interfaccia utente (UI, User Interface):**
 - linea di comando (CLI, Command Line Interface)
 - interfacce grafiche (GUI, Graphical User Interface)
- **Esecuzione di programmi:**
 - i programmi possono richiedere l'accesso a file e dispositivi I/O
 - deve essere possibile rilevare lo stato di terminazione di un programma
- **Comunicazione tra processi:**
 - memoria condivisa: processi lavorano in un'area comune che deve essere gestita in modo da evitare inconsistenze
 - scambio di messaggi (o pacchetti)
- **Protezione:**
 - gli utenti possono limitare l'accesso alle proprie informazioni
 - non tutti gli utenti possono eseguire tutti i comandi o accedere a ogni file

Interfacce utente

In molti SO l'interfaccia utente non è altro che un programma che viene avviato all'atto del login.

Esistono due tipi di interfacce utente

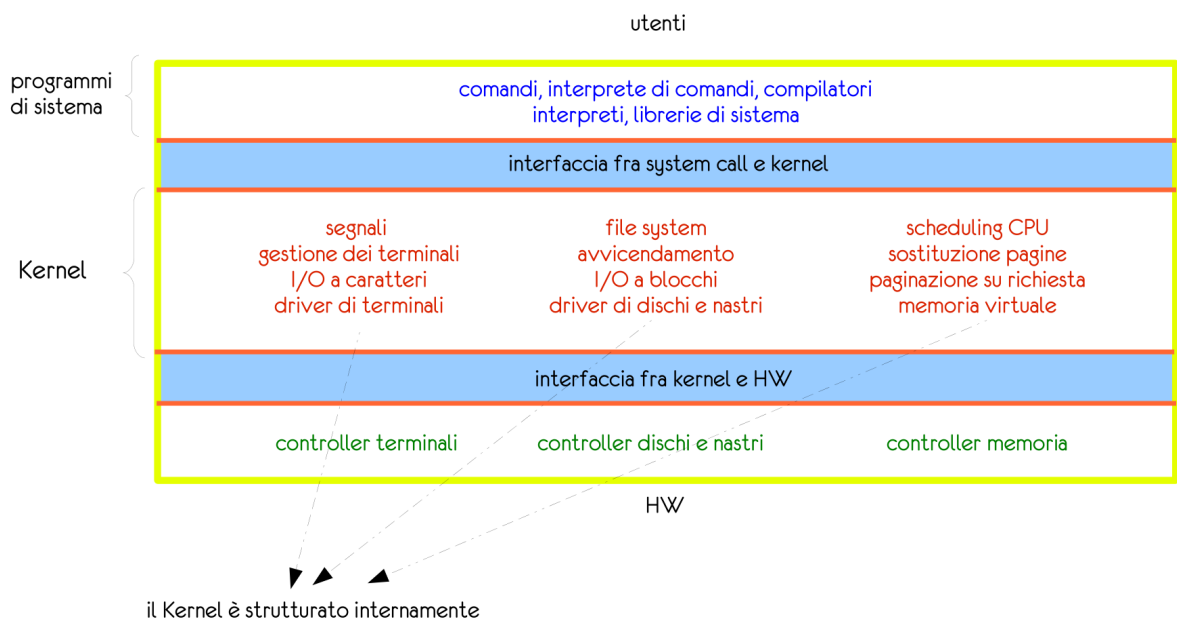
1. CLI (Command-line interface) o shell: esegue un ciclo infinito in cui attende un comando, lo esegue, e torna di nuovo in attesa.
2. GUI (Graphical User Interface): si basa su un desktop, su cui sono posti oggetti selezionabili e attivabili tramite puntatore

Implementazione di un SO

1. **Scelta di un linguaggio che supporti la portabilità:** mentre in passato i SO venivano scritti in linguaggio macchina (assembly), attualmente le CPU moderne sono troppo complesse per fare ciò, quindi vengono scritti in **linguaggi di alto livello** (es: Unix, Linux e Windows sono scritti in C);

2. **Definizione delle politiche di alto livello:** serve definire criteri e meccanismi per le funzioni che l'utente vorrà utilizzare
 - a. **criterio (o politica):** specifica un comportamento da seguire in una certa circostanza
 - b. **meccanismo:** strumenti (anche software) neutri rispetto ai criteri, che vengono quindi utilizzati per seguire i criteri
3. **Scelta di un'architettura per il SO:** per poter essere mantenuto e aggiornato, un SO deve essere ben strutturato, anche per evitare problemi dovuti alla crescita di complessità del programma e/o degli applicativi

Stratificazione



Per questioni di sicurezza e modularità, il sistema ha una struttura “a cipolla”: lo strato più interno corrisponde all’HW, quello più esterno all’interfaccia utente.

Su ogni strato esiste un oggetto astratto, che incapsula dei dati. Le operazioni che poi elaborano questi dati possono scegliere se rendere visibili/accessibili all’esterno o no.

Vantaggi e svantaggi:

- semplicità di progettazione: per realizzare uno strato basta sapere quali funzionalità ha a disposizione, non importa sapere come sono realizzate
- semplicità di debug e verifica del sistema: ogni strato usa solo funzionalità messe a disposizione dallo strato immediatamente inferiore, possiamo verificare gli strati uno per volta
- difficoltà: definire in modo opportuno gli strati, e quali funzionalità mettere in ognuno di loro
- minore efficienza in fase di esecuzione: ogni passaggio di stato comporta infatti la chiamata di una nuova funzione, da caricare, che può richiedere parametri, da caricare a loro volta ...

Stratificazione e macchine virtuali

Le macchine virtuali sono un approccio alla stratificazione che implica la **duplicazione del comportamento di ogni componente hardware del computer**.

Invece di installare un SO direttamente, installandolo su una macchina virtuale può lavorare su un computer che ha già un suo SO.

I SO installati sulla macchina virtuale sono detti **ospiti**.

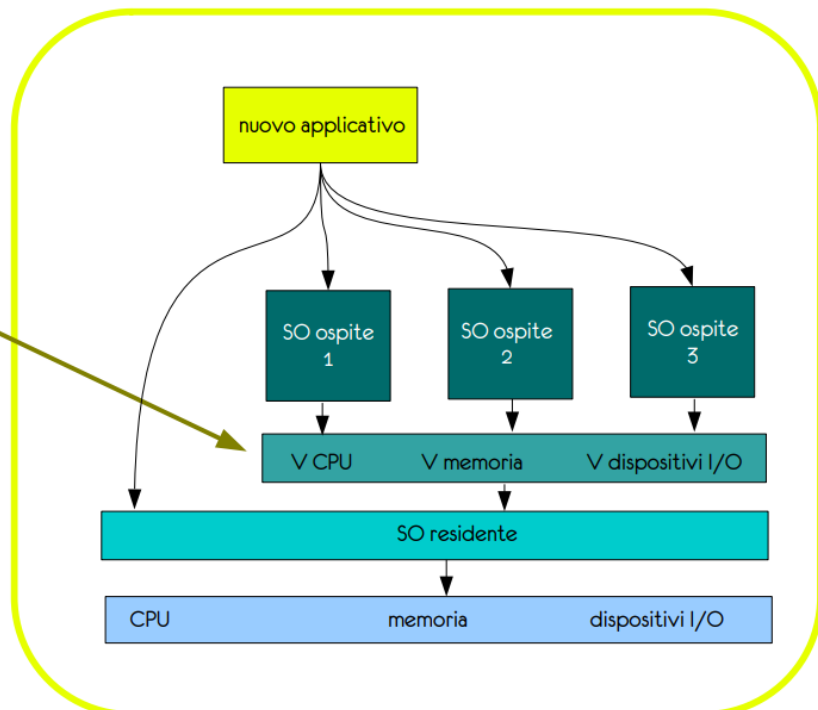
Può essere utile per usare SO diversi sullo stesso computer (anche da utenti diversi) o verificare/debuggare applicativi su SO diversi dal proprio.

Java Virtual Machine

CLR di .Net

Vmware

Hypervisor:
elemento di una
macchina virtuale che
fornisce un'astrazione
delle diverse risorse
hardware



Microkernel

Questo tipo di modularizzazione consiste nel rimuovere dal kernel tutto ciò che non è essenziale, spostandolo a livello di applicativo utente.

I **microkernel** contengono i servizi minimi per la **gestione di processi, comunicazione e memoria**.

Vantaggi:

- facilità di estensione: possibile aggiungere nuovi servizi senza modificare il kernel
- maggiore semplicità di adattamento a nuove architetture
- maggiore sicurezza: il kernel non viene direttamente affetto da cambiamenti

Svantaggi:

- possibile generare sovraccarichi quando processi utente vengono eseguiti con funzionalità di SO
- bottleneck: comunicazione indiretta

Tecnica a moduli

L'approccio considerato il migliore adottato, dai SO moderni. Ad un kernel minimale possono essere aggiunti moduli nuovi, in fase di avvio ma anche di esecuzione.

Aggiungere moduli significa aggiungere system call, ma anche capacità di gestire nuovo hardware.

Come nei sistemi a microkernel, il kernel qui gestisce un nucleo essenziale, che viene espanso poi con moduli.

Ogni modulo ha un'interfaccia ben definita.

Vantaggi:

- ogni modulo può usare qualsiasi altro modulo (flessibilità)
- la comunicazioni fra moduli è diretta (efficienza)

Processi

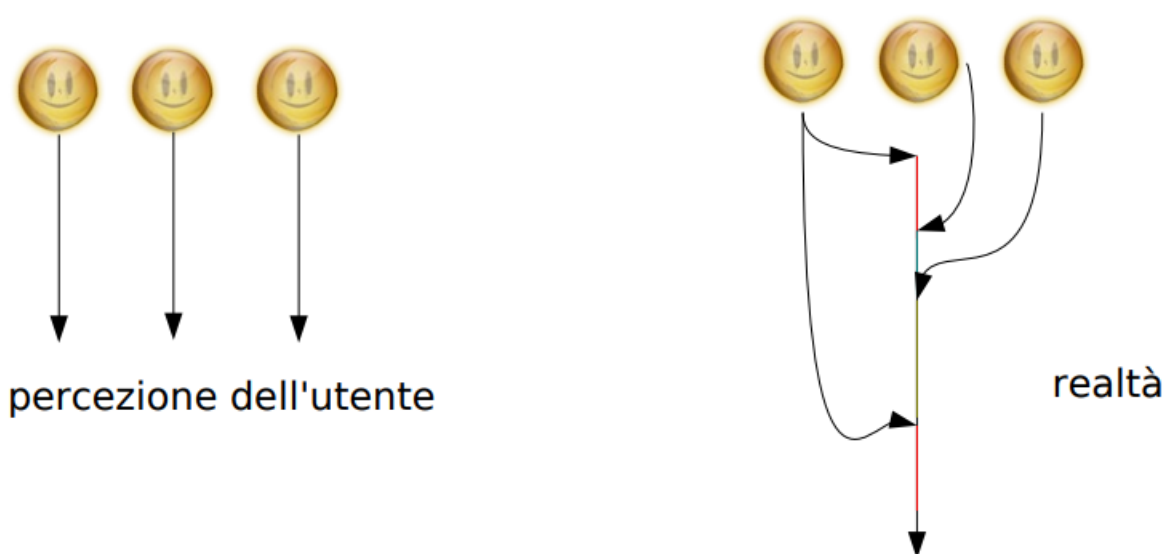
“processo” è un'astrazione, una rappresentazione interna al SO, che consente di pensare e realizzare meccanismi quali **multitasking**, **scheduling** della CPU, protezione.

Il SO deve mantenere informazioni riguardo i diversi task: ogni task esegue un programma, elabora dei dati, ha un utente “proprietario”, può avere una priorità.

Quando un task viene interrotto e ripreso occorre **mantenere tutte le informazioni necessarie**.

Ogni processo ha una propria **sezione dati** composta dallo stack di esecuzione e dallo heap, ma in RAM viene anche memorizzato (per ogni processo) il **programma** in sé e il **program counter**, che indica la prossima istruzione da eseguire.

Parallelismo virtuale

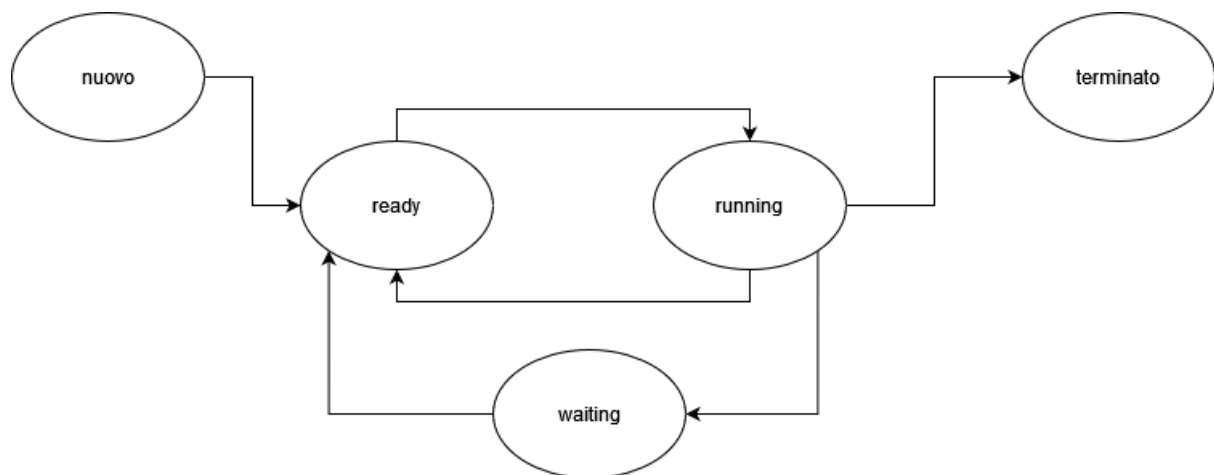


Categorie di processi

Ogni processo può avere diversi stati che aiutano il SO ad eseguire scheduling e parallelismo:

- **new**: appena creato
- **running**: in esecuzione
- **waiting**: in attesa di un evento (es: ricevere un dato, completamento di un'operazione I/O, sospensione volontaria)
- **ready**: aspetta l'assegnazione della CPU (può passare running → ready con un interrupt)
- **terminated**: cessata esecuzione (con exit, abort, kill)

Diagramma di transizione degli stati di un processo



PCB

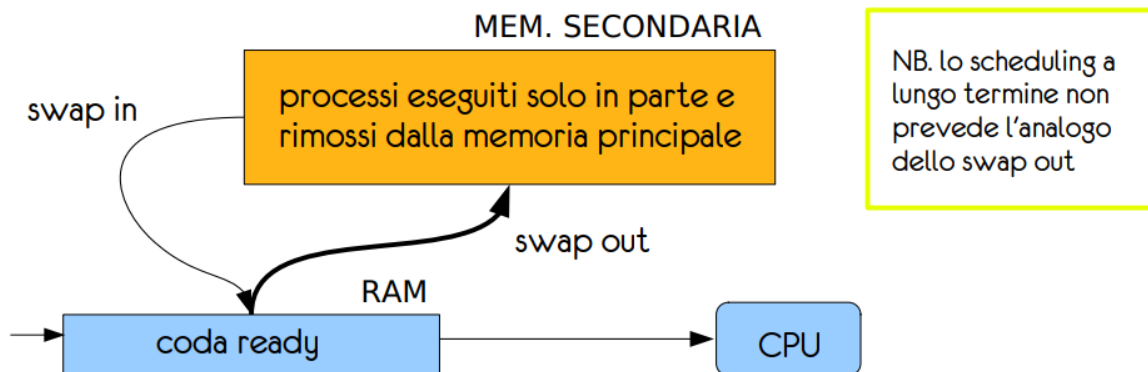
In un SO un processo è rappresentato dal **PCB**, Process Control Block, in cui sono rappresentate le sue informazioni:

- stato del processo
- PC (Program Counter)
- copia dei registri di CPU (copiati nel PCB quando il processo passa da running a ready)
- info sullo scheduling CPU (priorità e parametri di scheduling)
- info sulla gestione della memoria (tabella delle pagine)
- contabilizzazione risorse
- stato di I/O (es: dispositivi assegnati, file aperti...)

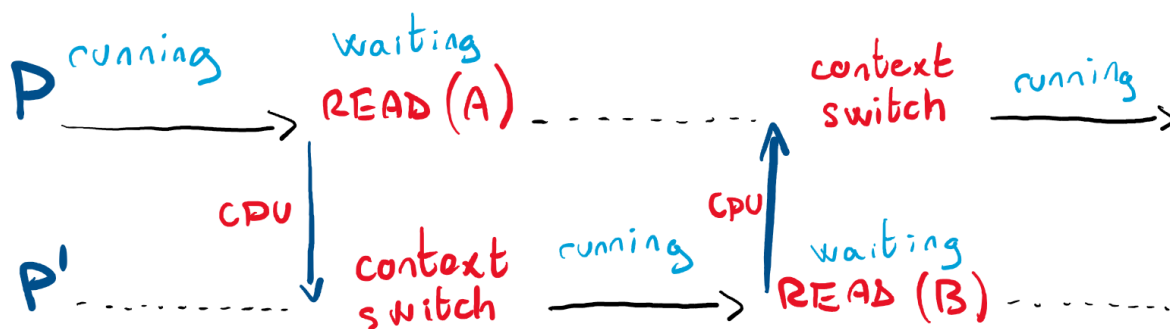
Vengono mantenuti in strutture dati dette **code**:

- coda **ready**: contiene tutti i PCB dei processi caricati in memoria in stato ready
- coda **di dispositivo**: contiene tutti i PCB dei processi in stato waiting

Quando un sistema ha in esecuzione più processi di quanto la RAM può sostenere, viene eseguito lo **swapping**: una parte dei processi, quindi i loro PCB, vengono trasferiti in memoria secondaria (swap out) fino a quando non sarà possibile eseguirli (swap in).



Commutazione della CPU



Il passaggio di un processo da running a waiting, o anche da ready a running richiede quello che si chiama **context switch**.

Questa operazione avviene esclusivamente in modalità kernel.

Il tempo per un context switch dipende dall'architettura, in quanto si possono avere diversi set di registri.

Scheduling

Abbiamo tre tipi di scheduling:

- Scheduling a lungo termine: presente in **sistemi batch**, in cui la coda dei processi era conservata in memoria secondaria. Si attiva quando un processo in esecuzione termina, scegliendone uno in memoria secondaria da caricare
 - criterio: mantenere un buon equilibrio fra processi CPU-bound e processi IO-bound
- Scheduling a medio termine: quando il **grado di multiprogrammazione** è troppo alto, e quindi non tutti i processi possono essere contenuti in RAM, a turno vengono spostati in memoria secondaria
- Scheduling a breve termine: politica di **avvicinamento alla CPU** dei processi

Scheduling a breve termine

Seleziona dalla coda ready il processo a cui assegnare la CPU.

Casi in cui occorre scegliere:

1. running → waiting
2. running → ready tramite interrupt
3. waiting → ready
4. running → terminated

Lo scheduling è **preemptive** se interviene in almeno uno dei casi 2 e 3, può occorrere anche in 1 o 4.

Lo scheduling è **non-preemptive** se interviene solo nei casi 1 o 4.

Il **dispatcher** effettua il cambio di contesto, effettua il posizionamento alla giusta istruzione, passa nella modalità di esecuzione giusta.

Algoritmi di scheduling

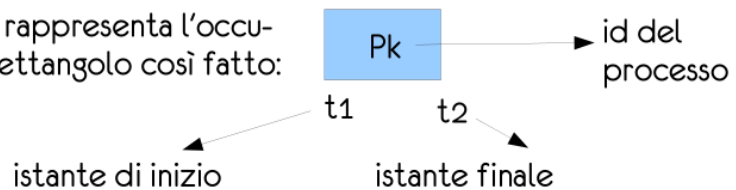
- **First-Come-First-Served** (FCFS)
- **Shortest Job First** (SJF)
- A priorità
- **Round Robin** (RR)
- A code multilivello
- A code multilivello **con feedback**

Criteri di scheduling

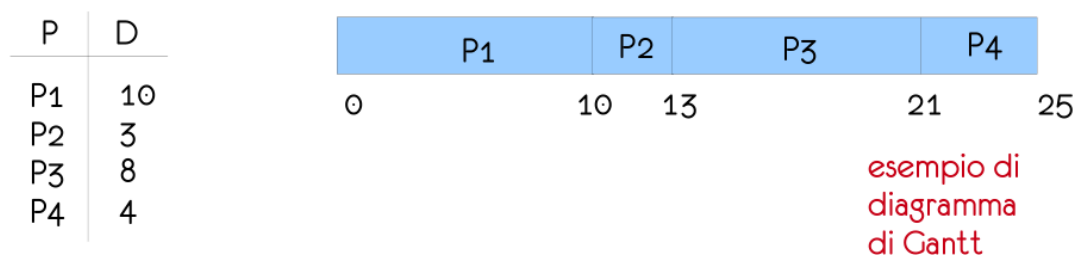
- mantenere la CPU il più attiva possibile
- **throughput**: numero di processi completati nell'unità di tempo
- **tempo di attesa**: tempo trascorso in coda ready
- **turnaround time**: tempo di completamento di un processo; somma del tempo di esecuzione e dei tempi di attesa
- **tempo di risposta**

Diagrammi di Gantt

In un diagramma di Gantt si rappresenta l'occupazione della CPU con un rettangolo così fatto:



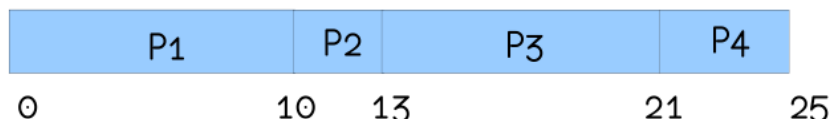
Es. dati i processi e relativi CPU burst in tabella, supponendo che la CPU venga allocata nell'ordine con cui sono dati i processi avremo il seguente diagramma



First-come-first-served

FCFS: PCB organizzati in una coda FIFO, non è preemptive, il primo processo arrivato è il primo processo ad avere la CPU, e la mantiene per un intero CPU burst.

Il tempo medio di attesa è abbastanza lungo, es:



$$tma = (ta_{P1} + ta_{P2} + \dots + ta_{Pn}) / num_proc$$

Prelazione

Meccanismo generale per il quale il **SO può togliere una risorsa riservata per un processo** (la CPU nello scheduling a breve termine) anche se il processo la sta utilizzando o la utilizzerà in futuro.

La prelazione richiede l'introduzione di meccanismi di **sincronizzazione** per evitare inconsistenze: se due processi condividono dati e uno dei due li sta aggiornando quando gli viene tolta la CPU, i dati possono risultare mancanti o corrotti.

La prelazione non è sempre possibile: occorre avere un'**architettura che supporti i timer** (in caso contrario lo scheduling è solo preemptive).

Shortest job first

SJF è ottimale nel minimizzare il tempo medio di attesa, seleziona sempre il processo avente CPU burst successivo di durata minima. Se la durata dei CPU burst non è nota, si prevede la durata sulla base dei CPU burst precedenti, combinati con una media esponenziale.

Questo scheduler può essere:

- preemptive (o shortest remaining time left): quando un nuovo processo diventa ready, lo scheduler controlla se il suo burst è inferiore a quanto rimane del burst del processo running
 - Sì: il nuovo processo diventa running, context switching
 - No: il nuovo processo va in coda ready
- non-preemptive: il processo viene messo in coda ready, eventualmente in prima posizione

Scheduling a Priorità

Ogni processo ha associata una priorità che decide l'ordine di assegnazione della CPU. Può essere con o senza prelazione.

La priorità può essere definita su due criteri:

- Internamente: sulla base di caratteristiche del processo. Per esempio, potremmo usare SJF e definire la priorità come l'inverso della durata stimata del CPU burst.
- Esternamente: non calcolabile, impostata in base a criteri esterni, dall'utente per esempio

Starvation

Tutti gli algoritmi a priorità sono soggetti a **starvation**: un processo potrebbe non ottenere mai la CPU perché continuano a passargli davanti nuovi processi a priorità maggiore.

Per risolvere, la priorità dei processi viene alzata con il trascorrere del tempo (**aging**).

Round Robin

Tipo di scheduling preemptive ideato per i sistemi time-sharing.

La coda ready è FIFO circolare. A ogni processo viene assegnata la CPU per un quanto di tempo, se non è sufficiente a concludere, il processo viene inserito nuovamente in coda ready e la CPU riassegnata al successivo.

P	D														
P1	10	<table><tr><td>P1</td><td>P2</td><td>P3</td><td>P4</td><td>P1</td><td>P3</td><td>P1</td></tr></table>							P1	P2	P3	P4	P1	P3	P1
P1	P2	P3	P4	P1	P3	P1									
P2	3	0	4	7	11	15	19	23	25						
P3	8														
P4	4	$tma = (0 + 4 + 7 + 11 + 11 + 8 + 4) / 4 = 45 / 4 = 11,25$													

Il tma è abbastanza alto ma **non si ha starvation**: ogni processo viene servito ogni $(n_proc - 1) * \text{quanto msec}$.

La velocità dell'algoritmo dipende anche dal quanto di tempo scelto: se è troppo lungo RR tenta di diventare un FCFS, mentre se è troppo corto i tempi del context switch possono rallentare e appesantire il processo.

Code a multilivello con feedback

Algoritmi utili quando si possono dividere i processi in categorie legate alla loro natura. La ready queue è divisa in tante code quante sono le categorie, che possono avere algoritmi di scheduling.

Tra le code, poi, esiste una priorità.

Se i processi possono cambiare code, si parla di multilivello con feedback

Creazione

Un processo viene generato dall'unica entità attiva gestita dal SO: un altro processo.

Con l'avvio del SO si genera un albero di processi che cresce e decresce dinamicamente, a seconda dell'evoluzione dell'elaborazione.

- Ogni processo ha un PID
- I processi figli, generati dal padre, in genere condividono delle risorse. In unix, per esempio, il figlio riceve una copia di tutte le variabili del padre

Terminazione

Un processo giunto alla sua ultima istruzione termina tramite la system call `exit`.

Il SO libera le risorse associate al processo.

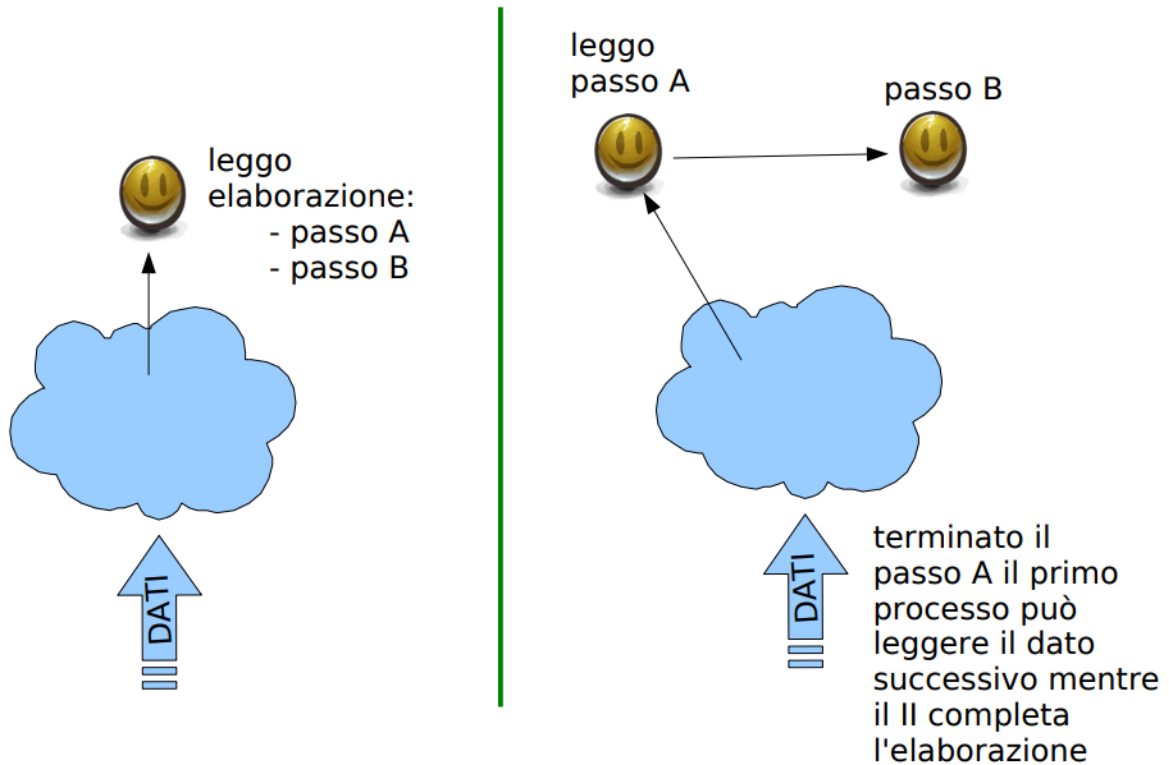
A terminazione completata, il padre del processo viene notificato della terminazione.

In certi SO, i dati sulla terminazione del figlio possono essere mantenuti finché non vengono ispezionati dal padre, che a sua volta si occupa di rimuoverli

Un processo può esplicitamente terminare un altro processo (conoscendo il suo PID) tramite system call se:

- il processo sta usando troppe risorse
- finisce di elaborare
- il padre è terminato (SO forza la terminazione dei figli)

Processi cooperanti



Produttore / consumatore

Un processo produce dei dati che vengono consumati da un altro processo.
Esempio: un web server produce pagine HTML consumate da un web browser.

Memoria condivisa

Area di memoria **fruibile da più processi**.

Richiesta tramite system call, i processi che la vogliono utilizzare devono agganciarla al proprio spazio di indirizzi.

Può essere modificata a piacimento secondo qualsiasi tipo di dati utile ai programmi che la utilizzano.

PRODUTTORE

...
alloca b di tipo buffer_cond come memoria condivisa

```
while (1) {  
    if (! pieno(b) ) {  
        nuovo = ... produci ...;  
        b.dato[b.inserisci] = nuovo;  
        b.inserisci = (b.inserisci+1) % D;  
    }  
}
```

CONSUMATORE

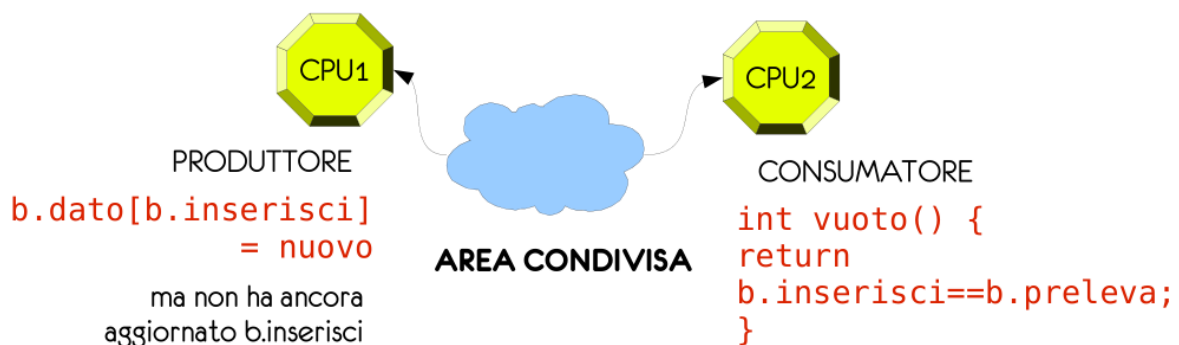
...
aggancia b al proprio spazio indirizzi

```
while (1) {  
    if (! vuoto(b) ) {  
        nuovo = b.dato[b.preleva];  
        b.preleva = (b.preleva+1) % D;  
    }  
}
```

Rimane comunque il bisogno di controllare gli accessi alla memoria per evitare inconsistenze.

Inconsistenza dei dati

Per colpa dello scheduler della CPU su sistemi a singolo core, o su sistemi con 2 o più core, potrebbero verificarsi problemi di inconsistenza dei dati.



Scambio di messaggi

Consente a due processi di comunicare senza condividere una stessa area di memoria.
Può essere:

- **diretto** o **indiretto**: se diretto, ricevente deve comunicare il PID al mittente
- **sincrono** (bloccante) o **asincrono** (non bloccante):
 - invio sincrono: mittente aspetta che ricevente riceve il messaggio

- ricezione sincrona: ricevente aspetta il messaggio
- rendez-vous: entrambi insieme
- a gestione automatica

Per comunicare, due processi utilizzano un **canale**, e le funzioni send e receive per scambiarsi messaggi.

- **comunicazione diretta**:
 - send(P, msg): invia msg al processo P;
 - receive(P, msg) / receive(id, msg): attendi un messaggio da P, memorizza il messaggio in msg / ricevi un messaggio, memorizza in id il PID del mittente e in msg il messaggio;
 - la reciproca conoscenza del PID finisce in un canale logico.
- **comunicazione indiretta**:
 - si utilizza una **mailbox** come canale intermedio di comunicazione, distinte dall'identità del ricevente;
 - send(M, msg): invia msg alla mailbox M;
 - receive(M, msg): attendi un messaggio alla mailbox M;
 - più mittenti e/o riceventi possono usare la stessa mailbox;
 - una stessa coppia di processi può utilizzare diverse mailbox per comunicare.
 - la mailbox ha una **capacità N definita da un buffer**:
 - 0: il canale non ha memoria (meccanismo no buffering); il mittente rimane sospeso se il ricevente non ha ancora consumato il messaggio ricevente (gestione esplicita del buffer);
 - N>0: il mittente rimane in attesa solo se il buffer è pieno (automatic buffering);
 - illimitata: il mittente non attende mai (automatic buffering).

Socket

Usato in sistemi client-server, **socket** è il nome dato ad un endpoint di un canale di comunicazione tra due processi.

Due processi che vogliono comunicare usano una coppia di socket, formati da l'IP della macchina concatenato ad una porta.

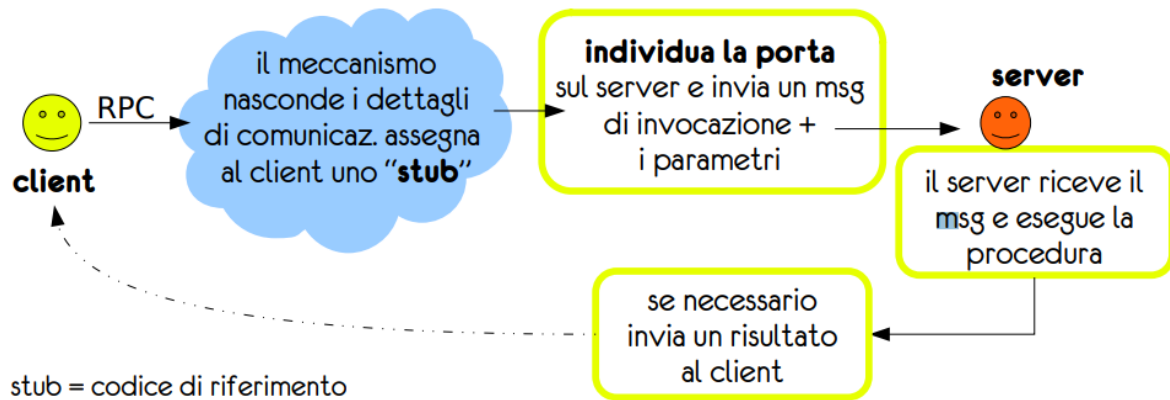
Le porte < 1024 sono riservate. Se un processo utente richiede una porta, riceverà un numero maggiore di 1024.

L'indirizzo di loopback (127.0.0.1) identifica la macchina stessa.

Remote procedure call

Meccanismo di scambio di messaggi utilizzato in sistemi client server per **invocare l'esecuzione di una procedura che risiede su un'altra macchina** connessa in rete.

I messaggi sono ben strutturati: hanno identificatore della procedura e parametri.



Pipe

Canale di comunicazione fra processi. Esistono due tipi di pipe:

- **Pipe anonima**
 - Canale simplex (unidirezionale), FIFO
 - Uno produce, l'altro consuma
- **Named Pipe**
 - FIFO
 - In unix è simplex, Windows full-duplex
 - Può far comunicare più di due processi
 - Spesso realizzata come file

Thread

Sono le **diverse parti di un processo** che cooperano per eseguire tutte le sue funzioni. Molti SO li considerano come l'unità di base d'uso della CPU al posto del processo.

Un thread è costituito da un identificatore, un program counter, un insieme di valori di registri.

Condivide con gli altri thread dello stesso processo il codice, la sezione dati, i file aperti, i segnali.

Un processo costituito da un solo thread è detto **heavyweight process**.

Vantaggi dell'utilizzo di thread:

- usare thread al posto di processi cooperanti è più efficiente
- molti programmi contengono sezioni di codice che è possibile eseguire indipendentemente
- context switch avviene più rapidamente
- si allocano meno risorse, in quanto molte vengono condivise
- la comunicazione è più veloce, perché avviene tramite variabili condivise
- è possibile assegnare thread diversi a core diversi in architetture multicore

Mentre i **thread** sono definiti dal programmatore e gestiti dal programma, i **processi** sono generati in modo invisibile all'utente, non permettendo ai programmi di gestirli direttamente ma solo tramite system call.

Alcuni linguaggi di programmazione includono specifiche istruzioni per la creazione e il controllo dei thread (es: Java, C#, Python), mentre per altri c'è bisogno di includere apposite librerie (es: C, C++; sono detti "a singolo flusso di controllo").

Operazioni sui thread

Sui thread è possibile eseguire diverse operazioni:

- **creazione** (nuova struttura dati);
- **terminazione**, più rapida di quella dei processi perché non richiede la gestione delle risorse;
- **sospensione**/blocco;
- **recupero**/risveglio;
- **join**: comporta l'attesa da parte di un thread della terminazione di un altro

Thread utente

Funzionamento a **singolo contesto**: ogni processo multithread deve gestire le info dei suoi thread, il loro scheduling, la comunicazione tra i thread.

Sono creati da funzioni di libreria che non possono eseguire istruzioni privilegiate.

Sono trasparenti al SO, che vede il processo come una sola entità.

Vantaggi:

- possibile utilizzarli su SO senza multi-threading in quanto sono gestiti internamente al processo
- criteri di scheduling adattabili alle esigenze del programma
- esecuzione più rapida perché non usa nè interrupt nè system call

Svantaggi:

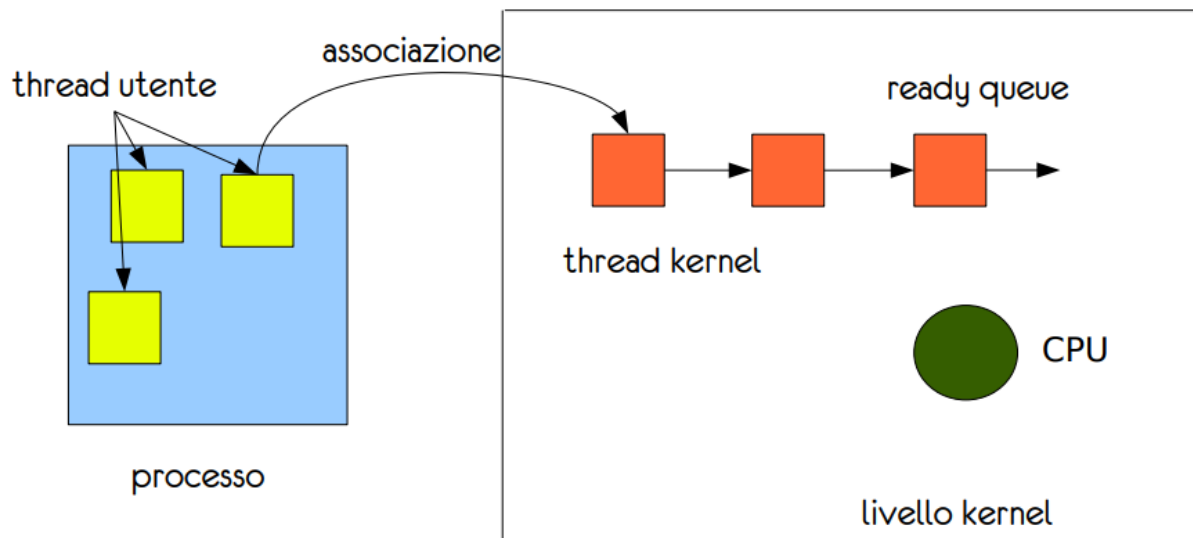
- non adattabili a sistemi multiprocessore
- operazioni I/O bloccano l'intero processo fino al termine dell'operazione

Thread kernel

Funzionamento a **molteplici contesti**, uno per ogni thread creato.

L'utente genera thread utente, e il SO li implementa a livello kernel associando ad essi delle strutture proprie (thread kernel sono strutture separate).

Sono gestiti dal SO, quindi esso gestisce anche il loro scheduling e sono più costosi perché per ogni thread deve mantenere gli appositi descrittori e le loro associazioni ad ogni processo.



Vantaggi:

- possibile distribuire i thread su più processori
- le operazioni di I/O non bloccano l'intero processo (eseguite solo sul thread)
- maggiore interattività con l'utente
- i singoli processi sono più veloci

Svantaggi:

- non portabili su SO non-multithread
- in caso di troppi thread, è possibile sovraccaricare il processore

Associazione dei thread kernel

- **uno a uno**: un thread utente assegnato a un solo thread kernel;
- **molti a uno**: una pool di thread utente assegnata a un solo thread kernel;
- **molti a molti**: una pool di thread utente assegnata a molteplici thread kernel (di solito di numero inferiore)

Lightweight process

Nei SO che usano il modello molti a molti o a due livelli, si ha la necessità di eseguire uno scheduling dei thread utente per l'accesso ai thread kernel.

Questo viene fatto introducendo un layer tra thread utente e kernel, chiamato Lightweight Process (LWP). Questi LWP sono visti come processori virtuali, e corrispondono ad un thread kernel.

Questa associazione viene fatta in modo esplicito dall'applicativo che deve usare delle "upcall" speciali.

L'utente quindi effettua il proprio scheduling dei thread su un insieme di LWP messi a disposizione dal kernel. Un thread utente è in esecuzione se ha un LWP assegnato.

Se un thread esegue una syscall bloccante, il SO informa l'applicativo con una upcall. L'applicativo quindi esegue un gestore della upcall, che salva lo stato del thread bloccante, e riassegna l'LWP ad un altro thread pronto per l'esecuzione. Quando poi si verifica l'evento che sveglia il thread sospeso, il SO con un'altra upcall farà segnare all'applicativo il thread come pronto, a cui verrà assegnato un LWP

Scheduling della CPU

In presenza di thread, lo scheduling della CPU è quindi a due livelli:

- process-contention scope(pcs): è lo scheduling effettuato all'interno di un processo per decidere a quali thread utente assegnare LWP. I thread kernel vengono gestiti come PCB
- system-contention scope (SCS): lo scheduling della CPU viene fatto tra tutti i thread kernel in queue, con una granularità più fine rispetto ai PCB, tutto in maniera indipendente dal processo di appartenenza

Esecuzione concorrente asincrona

Problema principale: interleaving delle istruzioni di diversi processi produttori eseguite in un'area di memoria condivisa. In assenza di controlli, diverse istruzioni sugli stessi dati possono non essere eseguite in modo atomico, producendo dati inconsistenti

(a) P1 esegue:	<code>buffer[free] = dato</code>	→	<code>buffer[4] = 2</code>
(b) P2 esegue:	<code>buffer[free] = dato</code>	→	<code>buffer[4] = 18</code>
(c) P1 esegue:	<code>free = (free+1) % D</code>	→	<code>free = 5</code>
(d) P2 esegue:	<code>free = (free+1) % D</code>	→	<code>free = 6</code>

Il problema è spesso presente in un SO con multitasking, in cui possono essere presenti allo stesso tempo diversi processi in modalità kernel.

Per risolvere il problema, viene implementata nel programma la **sezione critica**, ovvero una porzione di codice in cui un processo modifica variabili condivise **secondo un certo criterio sicuro**.

1) sezione non critica

2) sezione di ingresso

3) `buffer[free] = dato`

4) `free = (free+1) % D`

5) sezione di uscita

6) sezione non critica

corrisponde ad una richiesta, fatta al meccanismo di gestione, di accedere alla sezione critica

sezione critica

avvisa il meccanismo di gestione che è possibile consentire ad un altro processo l'accesso in sezione critica

Una sezione critica è determinata dalle variabili condivise utilizzate, e non deve essere eseguita con interleaving di istruzioni di altre sezioni critiche che usano le stesse variabili.

Criteri soddisfatti da una soluzione al problema della sezione critica:

1. **Mutua esclusione**: un solo processo per volta può eseguire la sua sezione critica
2. **Progresso**: nessun processo che non desideri utilizzare una variabile condivisa può impedirne l'accesso a processi che desiderano utilizzarla. Solo i processi che intendono entrare in sezione critica concorrono a determinare chi entrerà
3. **Attesa limitata**: esiste un limite superiore all'attesa di ingresso in sezione critica

Soluzione 1)

```
<sezione non critica>
while (turno != i) do no_op;
<sezione critica>
turno = j;
<sezione non critica>
```

sezione di ingresso

sezione di uscita

Problema: se un programma non ha più bisogno di usare la sezione critica, non ci entrerà; se è il turno per un programma di utilizzare la sezione critica ma non ci entra, l'altro processo rimane in waiting (violazione progresso)

Soluzione 2)

```
flag[i] = true;
while (flag[j]) do no_op;
<sezione critica>
flag[i] = false;
<sezione non critica>
```

sezione di ingresso

sezione di uscita

Problema: quando entrambi i processi hanno il flag attivo, entrambi vanno in loop infinito (violazione attesa limitata)

Soluzione 3) **algoritmo di Peterson**


```

1
flag[i] = true; turno = j;
while (flag[j] && turno == j) do no_op;

```

sezione di ingresso

<sezione critica>

flag[i] = false;

sezione di uscita

<sezione non critica>

garantisce tutti i criteri, ma funziona solo per 2 processi, quindi **non è generalizzabile**.

Soluzione 4) algoritmo del Fornaio

Codice di P_i :

```

choosing[i] = true;
ticket[i] = max_ticket + 1;
choosing[i] = false;

for (j = 0; j < N; j++) {
    while (choosing[j]) no_op;
    while
        (ticket[j] != 0 &&
         ticket[j] < ticket[i] ||
         ticket[j] == ticket[i] && j < i) no_op;
}

```

sto richiedendo un biglietto

mi viene dato un biglietto

non sto più richiedendo un biglietto

aspetto eventualmente che gli venga rilasciato il biglietto

se il processo j desidera entrare in SC

e ha un biglietto precedente il mio oppure ...

ha lo stesso numero ma il suo id di processo è precedente il mio, allora cedo il passo

per ogni processo concorrente eseguo un controllo

garantisce tutti i criteri, funziona per **N processi**, ma il codice è complesso e i processi fanno **busy waiting** (anziché sospendersi, attengono il turno tenendo occupata la CPU)

Sincronizzazione hardware

- 1) Potremmo disabilitare gli interrupt quando entriamo in una sezione critica, questo eviterebbe la prelazione. Ma causa interferenze pesanti con lo scheduling della CPU, e gli interrupt non possono essere mantenuti disabilitati a lungo.
- 2) Introdurre l'uso di lock: per entrare in una sezione critica, il processo deve avere ottenuto il giusto lock, che rilascerà al termine. Per questo motivo molte architetture forniscono operazioni di controllo e modifica del valore di una cella e istruzioni per lo scambio in modo atomico

TestAndSet

Implementazione:

```
boolean TestAndSet (boolean *variabile) {  
    boolean valore = *variabile;  
    *variabile = true;  
    return valore;  
}
```

Esecuzione:

```
while (TestAndSet(&lock));
```

```
<sezione critica>
```

```
lock = false;
```

L'esecuzione dell'intera routine è **atomica**.

TestAndSet restituisce il valore precedente di lock che sarà falso se nessun altro processo è in una sezione critica controllata tramite lock. Solo in questo caso si esce dal while.

Swap

Come alternativa a TestAndSet abbiamo lo swap, che scambia in modo atomico i valori dei suoi due parametri.

```
Swap(boolean *a, boolean *b) {  
    boolean temp = *a;  
    *a = *b;  
    *b = temp;  
}
```

La particolarità è nell'esecuzione della routine, che è atomica.

Usiamo lo swap per realizzare l'accesso in mutua esclusione ad una sezione. Oltre al lock, variabile condivisa, abbiamo anche una variabile booleana locale "chiuso"

```
chiuso = true;  
while(chiuso) Swap(&lock, &chiuso);
```

```
<sezione critica>
```

```
lock = false;
```

- Si esce dal ciclo while solo quando lock è false
- All'uscita da Swap lock risulta automatica impostato a true

Entrambi i metodi visti garantiscono la mutua esclusione, ma non l'attesa limitata. Non c'è garanzia che un processo che vuole eseguire una delle due non venga sempre prevaricato da altri.

Attesa limitata

Sia TestAndSet che Swap garantiscono la mutua esclusione, ma non l'attesa limitata: non garantiscono che un processo non venga sempre prevaricato dagli altri.

Perciò viene utilizzato l'algoritmo dell'**attesa limitata**

<u>Sezione d'ingresso:</u>	<u>Sezione d'uscita:</u>
<pre>attesa[i] = true; chiave = true; while (attesa[i] && chiave) chiave = TestAndSet(&lock); attesa[i] = false;</pre>	<pre>j = (i+1) % n; while ((j != i) && !attesa[i]) j = (j+1) % n; if (j == i) lock = false; else attesa[j] = false;</pre>

Con questo algoritmo persiste comunque il problema del busy waiting.

Semafori

Strumento di sincronizzazione introdotti da Dijkstra per minimizzare il busy-waiting (e anche per semplificare la vita ai programmatori).

Il semaforo non è altro che una variabile a cui si può accedere solo con due operazioni atomiche:

- P (proberen, verificare in olandese)
- V (verhogen, incrementare)

Definizione di P	Definizione di V
<pre>P (S) { while (S <= 0) no_op; S--; }</pre>	<pre>V(S) { S++; }</pre>

Per controllare una sezione critica, basterà quindi:

P(mutex);

<sezione critica>

V(mutex);

Questa attesa attiva dei semafori che abbiamo visto è detta **spinlock**. Ci sono altre implementazioni che lo evitano.

Ad esempio: ogni semaforo mantiene una lista di PCB dei processi sospesi su quel semaforo; quando un processo si sospende su quel semaforo, lo scheduler assegna la CPU

ad un altro processo, e quando il semaforo viene alzato, uno dei processi in attesa viene riattivato.

Vediamo una possibile implementazione del tipo di dato semaforo, nel caso questo includa una coda di attesa

```
typedef struct {
    int valore; /* Valore del semaforo */
    processo *lista; /* Lista di attesa relativa al semaforo */
} semaforo;

P (semaforo *s) {
    S->valore - -; /* Decremento il valore del semaforo */
    /* se il valore diventa negativo occorre
    sospendere il processo */
    if (S->valore < 0) {
        <aggiungi il PCB di questo processo a S->lista>
        /* block è una system call che richiama lo scheduler
        della CPU, che deve riassegnare la medesima a un altro
        processo, e il dispatcher, che deve effettuare il context
        switch */
        block();
    }
}

V (semaforo *S) {
    S->valore++; /* Incremento il valore del semaforo */
    /* Se il valore è negativo, allora ci sono processi da
    risvegliare */
    if (S->valore < 0) {
        <scegli un PCB P da S->lista>
        wakeup(P);
    }
}
```

NB: il valore del semaforo indica il numero di processi in attesa

Inizializzazione e uso

I valori dei semafori possono essere:

- **1** = valore di inizializzazione, risorsa disponibile;
- **0, -1, -2, ..., -n** = risorsa occupata

I semafori che possono assumere valori > 1 sono detti **semafori contatori**, il cui numero rappresenta una quantità di risorse disponibili.

Tipi di sincronizzazione

I semafori possono realizzare molti tipi di sincronizzazione:

- **Mutua esclusione:** tutti i processi coinvolti separano le loro sezioni critiche con $P(\text{mutex})$ e $V(\text{mutex})$, dove mutex è un semaforo di mutua esclusione;
- **Accesso limitato:** tutti i processi coinvolti separano le loro sezioni critiche con $P(\text{nr})$ e $V(\text{nr})$, dove nr è un semaforo contatore che permette a molteplici processi di eseguire in parallelo una certa sezione critica;
- **Ordinamento:** l'ordine di esecuzione dei processi viene esplicitamente controllato, ad esempio...

P1:	P(sem) codice	P2:	codice V(sem)
-----	------------------	-----	------------------

in quale ordine vengono eseguiti P1 e P2?

Operazioni atomiche

Le operazioni sui semafori **devono essere eseguite in modo atomico**, in quanto i semafori sono variabili condivise (le operazioni sono quindi dentro sezioni critiche).

Si può ottenere l'atomicità

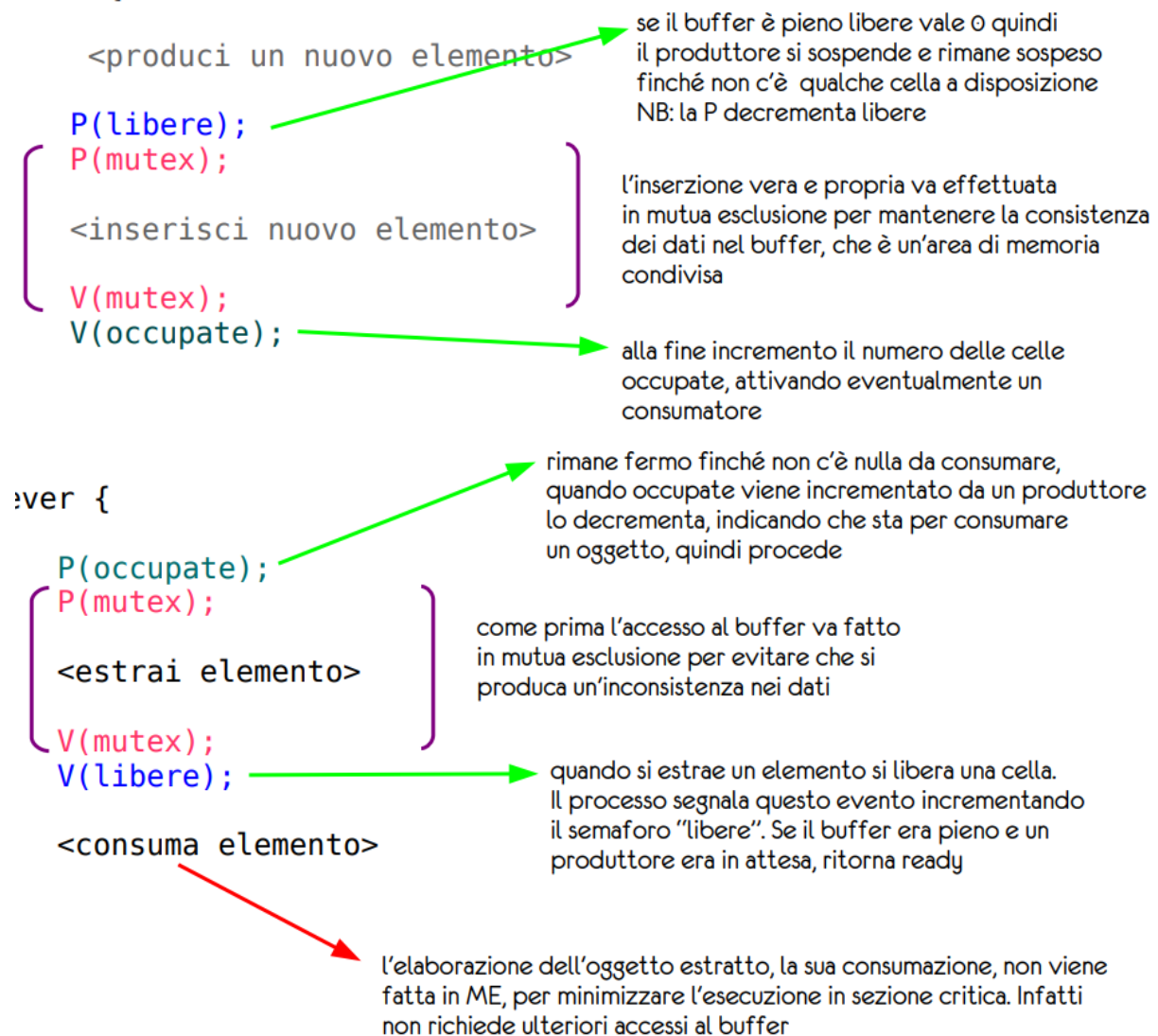
- sui sistemi monoprocesso disabilitando gli interrupt (P e V sono brevi quindi non rallentano il sistema);
- sui sistemi multiprocessore utilizzando gli spinlock (disabilitare gli interrupt cala le prestazioni).

Problema: produttori-consumatori

Si utilizzano 3 semafori: mutex (mutua esclusione) = 1, libere (n° caselle libere del buffer) = n, occupate (n° caselle occupate del buffer) = 0.

<u>Produttore</u>	<u>Consumatore</u>
<pre> forever{ <produci un nuovo elem.> P(libere); P(mutex); <inserisci un nuovo elem.> V(mutex); V(occupate); </pre>	<pre> forever{ P(occupate); P(mutex); <estrai elemento> V(mutex); V(libere); <consuma elemento> </pre>

}	}
---	---



Problema: lettori-scrittori

Dei processi usano la stessa area di memoria: gli scrittori modificano la risorsa (accedono in mut. ex.), i lettori la leggono soltanto (accedono in parallelo).

Si utilizza una variabile int condivisa `numlettori` (n° di processi che stanno leggendo) = 0, e si utilizzano i semafori `scrittura` (mutua esclusione) = 1 e `mutex` (mut. ex. per l'accesso a `numlettori`) = 1.

<u>Lettore</u>	<u>Scrittore</u>
<pre> forever { P(mutex); </pre>	<pre> forever{ P(scrittura); </pre>

<pre> numlettori++; if (numlettori == 1) P(scrittura); V(mutex); <legge> P(mutex); numlettori--; if (numlettori == 0) V(scrittura); V(mutex); } </pre>	<pre> <scrive> V(scrittura); } </pre>
--	--

P(scrittura);
<scrive>
V(scrittura);

uno scrittore accede all'area condivisa in ME con chiunque altro (lettori o scrittori) perché modifica la risorsa. Scrittura vale inizialmente 1

Lettores

forever {

P(mutex);
numlettori++;
[A] if (numlettori == 1)
P(scrittura);
V(mutex);

<legge>

P(mutex);
numlettori--;
[B] if (numlettori == 0)
V(scrittura);
V(mutex);

}

un lettore usa due classi di sezione critica: una relativa all'area di memoria da cui legge e una relativa alla variabile condivisa numlettori. Mutex controlla l'accesso a quest'ultima

[A] e [B] parentesizzano la sezione critica relativa all'area da cui si legge. Implementano un controllo per cui un lettore può accedervi a patto che nessuno scrittore stia operando.

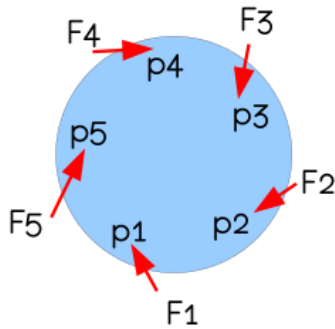
[A]:
se numlettori > 1 allora qualcun altro ha già fatto in modo tale che nessuno scrittore sia attivo. Il lettore può procedere nella lettura.

se numlettori == 1 sono l'unico lettore, tramite il semaforo scrittura controllo ed eventualmente attendo che non ci siano scrittori attivi. Poiché uso una P, quando riesco a passare blocco l'accesso ad eventuali scrittori

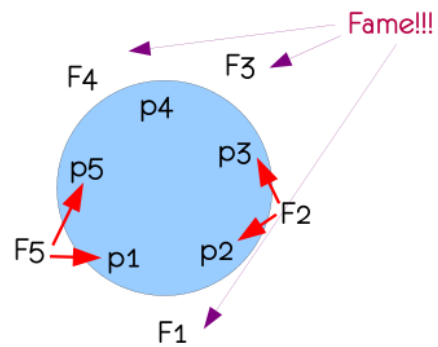
[B] decrementa numlettori ed eventualmente incrementa il semaforo scrittura

Problema: cinque filosofi

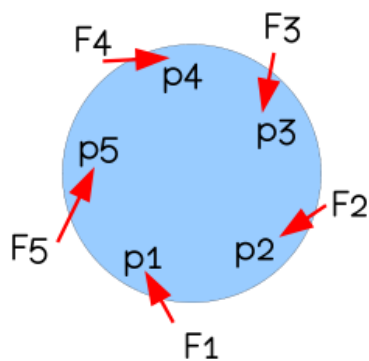
5 filosofi passano il tempo seduti intorno a un tavolo pensando e mangiando a fasi alterne. Per mangiare un filosofo ha bisogno di due posate ma ci sono solo cinque posate in tutto. Possibili problemi::



Deadlock: tutti hanno preso la posata di sinistra ma nessuno può prendere quella di destra



Starvation: due filosofi si accaparrano sempre le posate necessarie impedendo agli altri di mangiare



Livelock: tutti prendono la posata di sinistra ma nessuno può prendere quella di destra, quindi tutti rilasciano la posata di sinistra, poi ricominciano da capo. I processi non sono propriamente bloccati ma non c'è progresso.

Una possibile soluzione può essere la seguente:

```
forever {  
    P(posata[i]);  
    P(posata[i+1]%5);  
  
    <mangia>  
  
    V(posata[i]);  
    V(posata[i+1]%5);  
  
    <pensa>  
}
```

Questa soluzione usa 5 semafori, una per posata. I filosofi chiedono prima la posata di sinistra e poi quella di destra.

Questa soluzione però porta a deadlock e starvation.

Dijkstra propone una soluzione che consente di evitare il deadlock. Bisogna rompere la simmetria nell'accesso alle posate, introducendo la seguente regola: ogni filosofo deve prendere per prima la posata con indice minore.

Ad esempio, f1 prenderà quindi p1 prima di p2, ..., f4 prenderà p4 prima di p5

Chandy-Misra

Soluzione “equa”: quando due processi sono in competizione l’algoritmo non favorisce sempre lo stesso (causando starvation). Viene introdotto un concetto di precedenza dinamica tra processi (cambia nel tempo), che funziona associando un concetto di stato alla risorsa.

1. Ogni forchetta (risorsa) può essere “sporca” o “pulita”; inizialmente sono tutte “sporche” e ogni filosofo ne ha una;
2. Quando un filosofo vuole mangiare, invia ai suoi vicini dei messaggi per ottenere le forchette che gli mancano;
3. Quando un filosofo, che ha una forchetta, riceve una richiesta: se sta pensando, la cede altrimenti se la forchetta è pulita ne mantiene il possesso, se è sporca la cede. Quando passa una forchetta ne pone lo stato a “pulita”;
4. Dopo aver mangiato, tutte le forchette di un filosofo diventano “sporche”. Se risultano richieste pendenti per qualche forchetta, il filosofo la pulisce e la passa.

Uso errato dei semafori

Per evitare di implementare erroneamente i semafori, è consigliato incapsulare risorse di basso livello, come i semafori, in tipi di dati astratti, come i **monitor**, offerti poi come nuovi costrutti del linguaggio.

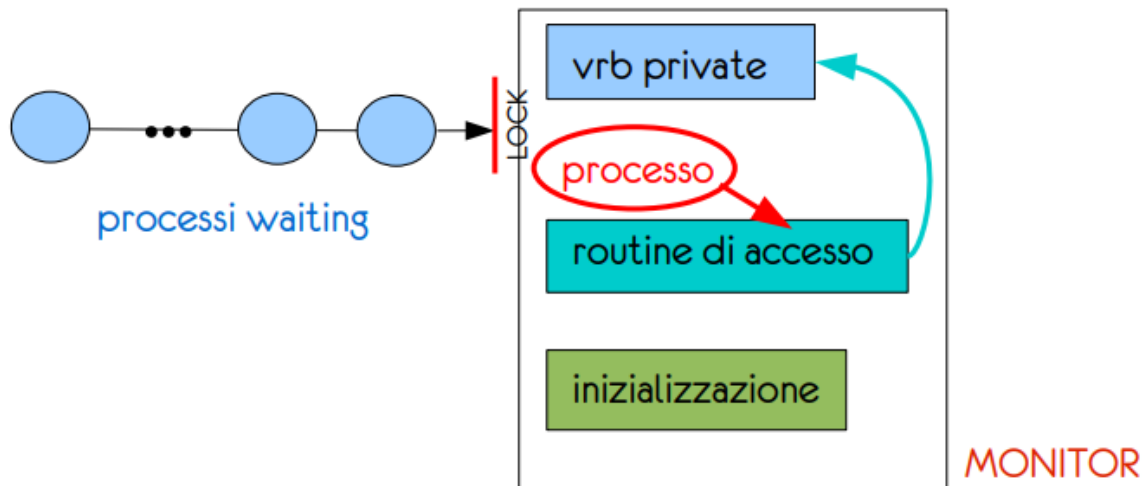
Monitor

Sono dei **costrutti di sincronizzazione** contenenti i dati e le operazioni necessarie per allocare una risorsa condivisa usabile in modo seriale.

E’ un **abstract data type**: in sé non è un tipo di dato, ma contiene dati e le istruzioni per utilizzarli.

Le **variabili** di un monitor sono **condivise dai processi che usano quel monitor**. Per accedere alle variabili condivise un processo deve eseguire una routine di accesso al monitor (possono essercene diverse).

Un solo processo per volta può essere attivo all’interno di un monitor.



un processo che riesce ad eseguire una routine di accesso al monitor acquisisce un **lock** sul monitor. Tutti gli altri processi che cercano di eseguire routine di accesso vengono sospesi in una coda di attesa esterna al monitor. Quando il lock viene rilasciato uno dei processi waiting viene riattivato

I monitor consentono anche di effettuare **sincronizzazione** tra processi **attraverso variabili** di tipo **condition** (boolean), su cui si possono eseguire solo le operazioni:

- `wait(x)`: esecutore sospeso se `x == false`;
- `signal(x)`: se un processo è sospeso sulla condition `x`, uno viene scelto e risvegliato;

monitor per un allocatore di risorse

```
boolean inUso = false;
```

```
condition disponibile;
```

```
monitorEntry void prendiRisorsa() {
    if (inUso) wait(disponibile);
    inUso = true;
}
```

```
monitorEntry void rilasciaRisorsa() {
    inUso = false;
    signal(disponibile);
}
```

se una qualche risorsa di interesse risulta usata da qualcun altro, allora mi sospendo sulla condition "disponibile". Quando mi risveglio continuo dalla linea di codice successiva: setto a true inUso.

Memento: il monitor è eseguito in ME quindi tutte le monitorEntry sono atomiche, a meno di sospensioni volontarie

quando rilascio una risorsa, faccio una notifica sulla condition "disponibile". Se qualcuno era in attesa si risveglia, altrimenti la notifica viene dimenticata

Signal e M.E.

Quando un processo (thread) esegue signal rischiamo di avere due processi attivi nel monitor.



Abbiamo due soluzioni:

- Segnalare e attendere
 - P attende
 - Q riprende ed esegue
- Segnalare e Proseguire
 - P continua
 - Q aspetta che P finisca

Transazione

Un insieme di istruzioni che esegue una singola funzione logica, ovvero una **sequenza di read e write che si conclude con un commit** (successo) **o con un abort** (fallimento).

La transazione deve essere atomica, e l'atomicità dipende dai dispositivi di memoria utilizzati per mantenere i dati elaborati dalla macchina:

- **memorie volatili** (RAM, cache, registri): cancellati a spegnimento;
- **memorie non volatili** (dischi, EEPROM): persistenti, ma non sempre "eterni";
- **memorie stabili** (memorie in RAID): supporti di memorizzazione che aggiungono politiche/strumenti di duplicazione, rendendo i dati "eterni"

Memoria volatile

Cosa succede se una transazione viene abortita e la memoria volatile viene cancellata?

Dobbiamo tenere traccia delle operazioni eseguite, un logfile mantenuto su memoria stabile.

Il log dovrà contenere quindi:

- l'inizio di una transazione T
 - <T, start>
- una sequenza di tuple, relativa ad un'operazione di write da fare
 - <ID transazione, ID dato modificato, valore precedente, nuovo valore>
- il successo della transazione
 - <T, commit>

Ripristino/Abort

A seguito di un crash di sistema, il SO controlla il log e per ogni transazione T registrata

- se a $\langle T, \text{start} \rangle$ non corrisponde un $\langle T, \text{commit} \rangle$, il SO esegue l'operazione $\text{undo}(T)$ che ripristina tutti i valori modificati dalla transazione eseguita in modo parziale;
- se a $\langle T, \text{start} \rangle$ corrisponde un $\langle T, \text{commit} \rangle$, il SO verifica che le modifiche registrate siano state effettivamente eseguite. In caso contrario, il SO esegue un $\text{redo}(T)$ attuando le modifiche.

Questa registrazione dei dati riduce leggermente l'efficienza dell'esecuzione, ma fornisce un'enorme affidabilità e stabilità al sistema.

Tutto il logfile?

Per evitare di scorrere tutto il log file ad ogni crash, introduciamo dei checkpoint.

Questi checkpoint ci fanno sapere che:

- tutte le transazioni prima di questo checkpoint sono state riportate in memoria stabile
- tutte le operazioni di scrittura registrate nel logfile sono state applicate con successo

In caso di crash, a noi basta andare a trovare il primo checkpoint, cosicché possiamo ignorare tutto quello che avviene prima, applicando le operazioni di undo/redo solo successive al checkpoint.

Transazioni atomiche concorrenti

Per combinare concorrenza e atomicità a livello logico, bisogna **garantire la proprietà di serializzabilità**.

La serializzabilità è la proprietà per cui la loro esecuzione concorrente è equivalente alla loro esecuzione in una sequenza arbitraria.

Serializzabilità

Dati N elementi, possiamo costruire N! sequenze diverse. Vogliamo però consentire un po' di concorrenza per aumentare l'efficienza complessiva dell'esecuzione, ma vogliamo anche le sequenze di esecuzione non seriali che sono equivalenti a quelle seriali, che producano risultati consistenti.

Introduciamo quindi le **operazioni conflittuali**:

- date due transazioni T1 e T2 e le due operazioni O1 e O2, se le operazioni appaiono in successione, accedono agli stessi dati e almeno una delle due operazioni è una write, allora O1 e O2 sono operazioni conflittuali

Protocollo di gestione dei lock

Per garantire la serializzabilità si può usare un meccanismo di lock in cui

- a ogni dato soggetto a transazione si associa un lock
- il lock di un dato può essere S (read only) o X (read & write)
- una transazione che intenda usare un certo dato deve richiederne il lock appropriato, ed eventualmente attendere che un'altra transazione lo rilasci

Gestione dei lock a due fasi

Inizialmente le transazioni sono in fase di “**crescita**”, ovvero possono ottenere nuovi lock per acquisire tutte le risorse necessarie, senza rilasciarne mai nessuno.

Successivamente, in fase di “**riduzione**”, le transazioni rilasciano via i lock per rimuovere le risorse inutilizzate, senza richiederne mai di nuovi.

In questo caso è possibile però avere deadlock.

Protocolli basati su timestamp

Il **timestamp** è una rappresentazione univoca di un istante temporale.

Il protocollo assegna ad ogni transazione TS un timestamp T prima della loro esecuzione.

Se due transazioni sono tali che $TS(T_1) < TS(T_2)$, allora il sistema deve garantire l'esecuzione sequenziale con T_1 prima di T_2 .

Regole che definiscono il protocollo

<1> se una transazione T desidera leggere D:

- <1.a> se $TS(T) < W(D)$: ho una transazione vecchia che cerca di leggere un valore sovrascritto da una più recente -> azione: si annulla la lettura richiesta e si esegue una sequenza di undo per annullare T
- <1.b> se $TS(T) > W(D)$: ok -> azione: si effettua la lettura e si aggiorna $R(D)$ assegnando $\max(R(D), TS(T))$

Osservazioni:

- L'algoritmo non impone un ordinamento corretto fra le transazioni, l'unico scopo è far sì che tutte le volte che una transazione legge un dato, abbia il dato originale oppure il dato modificato dalla transazione stessa;
- In questo caso viene **garantita l'atomicità funzionale**, diversa dall'atomicità di esecuzione per il fatto che viene consentito l'interleaving delle istruzioni;
- Si hanno quindi transazioni che non interferiscono tra di loro perché usano dati diversi oppure non vengono eseguite in modo concorrente.
- L'algoritmo **non è preventivo**: identifica situazioni problematiche e le aggiusta

Deadlock

Un deadlock è una situazione per cui un insieme di processi sono fermi in attesa di un evento che solo uno dei processi appartenenti all'insieme stesso potrebbe causare.

Se un processo vuole usare N istanze di una certa risorsa, dovrà chiederle al gestore delle risorse (SO)

Abbiamo quindi tre fasi:

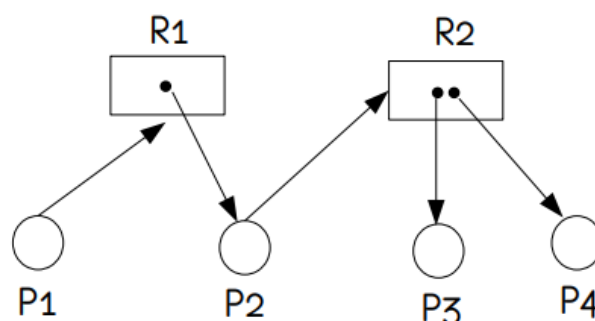
- richiesta
- uso
- rilascio

Esempio di deadlock fra due processi: P1 detiene l'unico lettore di fotografie e vuole la stampante per stampare delle foto, P2 ha la stampante e vuole il lettore perché deve, anch'esso, stampare delle foto.

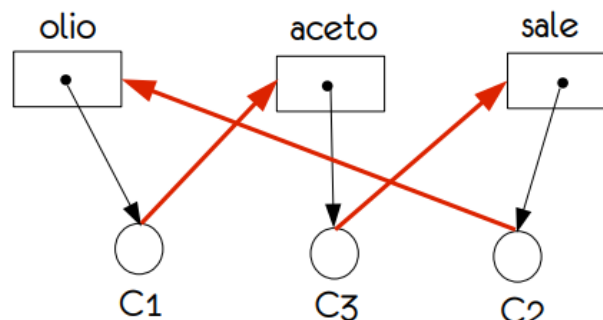
Grafo di assegnazione delle risorse

E' una rappresentazione delle assegnazioni che permette di rilevare situazioni di deadlock, tramite un grafo $G = \langle V, E \rangle$ tale per cui:

- V è l'insieme dei vertici ed è partizionato in due sottoinsiemi P ed R ($P \cap R = \emptyset$)
 - P è l'insieme di tutti i processi del sistema
 - R è l'insieme di tutte le classi di risorse del sistema
- E è l'insieme degli archi:
 - Un arco direzionato da R_i a P_j indica che una risorsa di classe R_i è stata assegnata al processo P_j ;
 - Un arco direzionato da P_j a R_i indica che il processo P_j ha richiesto ed è in attesa di una risorsa di tipo R_i .



Esempio per i 3 cuochi:



C1 aspetta C3, che aspetta C2, che aspetta C1: **deadlock**

NB: se il grafo non contiene ciclo, non c'è deadlock. **La presenza di un ciclo è condizione necessaria ma non sufficiente per avere deadlock.**

Un caso reale: spooling

Un processo deve stampare un documento, ma la stampante gestisce una pagina per volta. Quindi il processo dovrebbe attendere ogni volta la terminazione della pagina corrente prima dell'invio alla stampante della successiva. Ed ecco che quindi si introduce uno spooler.

Lo spooler è un programma che fa da intermediario tra i processi che vogliono stampare e la stampante. Il processo che stampa può terminare subito dopo aver inviato il documento.

L'invio può avvenire in due modi:

- 1) Il documento viene salvato in un file temporaneo e poi elaborato dallo spooler
 - a) Molti sistemi di spooling gestiscono solo documenti codificati in un certo formato
 - b) Bisogna quindi convertire il documento, salvarlo
 - c) Se termina la memoria prima che il processo di stampa termini il proprio lavoro, cosa succede? Un bel deadlock

Che fare col deadlock?

- **Rilevare** il deadlock è una pratica fondamentale;
- **Rompere** il deadlock richiede la capacità di monitorare richieste e assegnazioni di risorse
- Per **prevenire** il deadlock occorre definire opportuni protocolli di assegnazione delle risorse
- **Fare finta che il deadlock sia impossibile** è la tecnica più usata e poco costosa perché non richiede politiche né risorse aggiuntive

Prevenzione del deadlock

Per prevenire il deadlock è necessario rendere impossibile una delle 4 condizioni necessarie al deadlock:

- **Mutua Esclusione**: la richiesta di usare le risorse in ME può essere rilasciata solo per alcuni tipi di risorse, altre sono intrinsecamente ME;
- **Strategia di Havender 1** (possession and wait): se un processo ha bisogno di più risorse, le ottiene tutte insieme, oppure non ne ottiene nessuna;
- **Strategia di Havender 2** (prelazione): quando un processo con N risorse ne richiede un'altra, la ottiene subito oppure rilascia tutte le altre;
- **Strategia di Havender 3** (attesa circolare): imporre un ordinamento delle risorse e dei processi.

1^a strategia di Havender

Tutte le risorse necessarie ad un processo devono essere richieste insieme:

- se sono **tutte disponibili**, il sistema le **assegna** e il processo prosegue

- se anche solo **una non è disponibile**, il **processo** non ne acquisisce e si mette in **attesa**.

Vantaggio: previene il deadlock.

Svantaggio: spreco di risorse (non vengono utilizzate in modo ottimale in quanto un processo può tenersi per più tempo risorse che non usa più).

Questa strategia **non funziona per processi heavyweight** però se all'interno del processo riusciamo a distinguere più thread di esecuzione, ciascuno dei quali ha bisogno di un sottoinsieme delle risorse ed è generato solo quando occorre, la strategia può risultare efficace.

2^a strategia di Havender

Quando un processo richiede una **risorsa** che gli viene **negata**, **rilascia tutte le risorse** accumulate fino a quel momento. Eventualmente, il processo richiederà tutte le risorse che ha perso, più quella che gli serviva.

E' una tecnica costosa (perdere delle risorse può significare perdere un lavoro già compiuto in parte) e vale la pena solo se il sistema è tale per cui questa tecnica viene eseguita raramente.

Il suo uso in congiunzione a un criterio di priorità che predilige l'assegnazione di risorse a processi che ne richiedono poche (può causare starvation).

3^a strategia di Havender

Ogni risorsa ha assegnato un numero utilizzato per quella risorsa soltanto, che le rende ordinabili in ordine crescente ($R_1 < R_2 < \dots < R_n$).

Un processo che ha bisogno di M risorse deve **richiederle in ordine crescente**.

Non si può avere deadlock perché l'ordinamento delle richieste impedisce l'attesa circolare, ma non è molto flessibile.

Deadlock avoidance

Non si può sempre evitare deadlock a priori. I metodi che consentono di fare ciò richiedono alcune informazioni, come per esempio il numero di risorse di cui hanno bisogno.

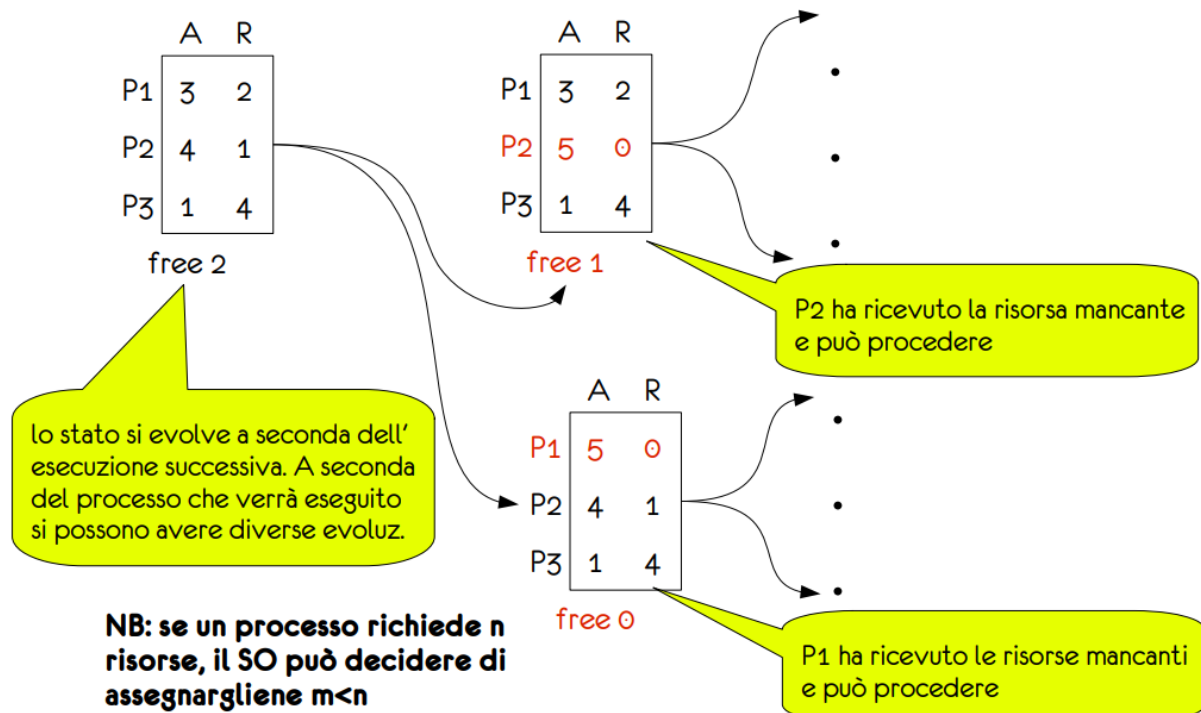
L'algoritmo di deadlock avoidance esamina lo stato di allocazione delle risorse e garantisce che in futuro non si formeranno attese circolari.

Dobbiamo quindi introdurre due nuove nozioni:

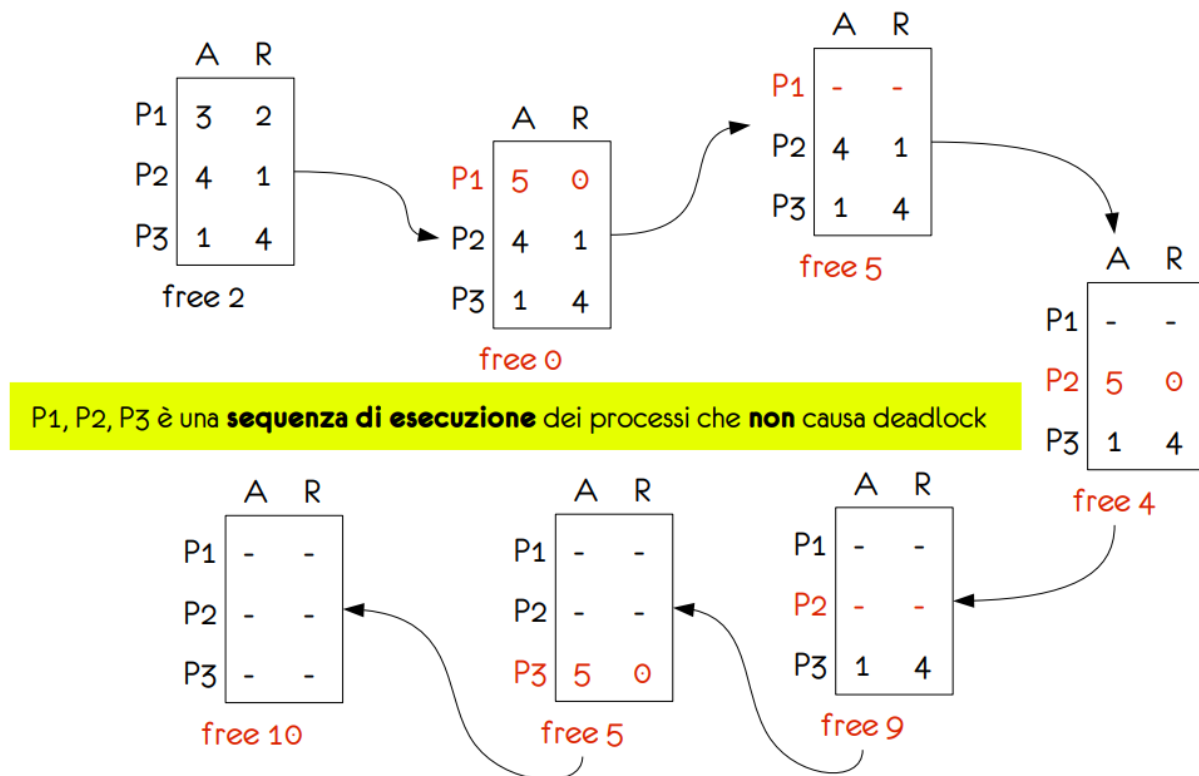
- 1) Stato (del sistema) sicuro: si dice che un sistema è in un stato sicuro se il SO può garantire che ciascun processo finisca la propria esecuzione in un tempo finito

- 2) Sequenza sicura: una sequenza di processi viene detta sicura se le richieste che ogni processo deve ancora fare sono soddisfacenti usando le risorse attualmente libere più le risorse usate e liberate da altri processi

Stato di allocazione delle risorse



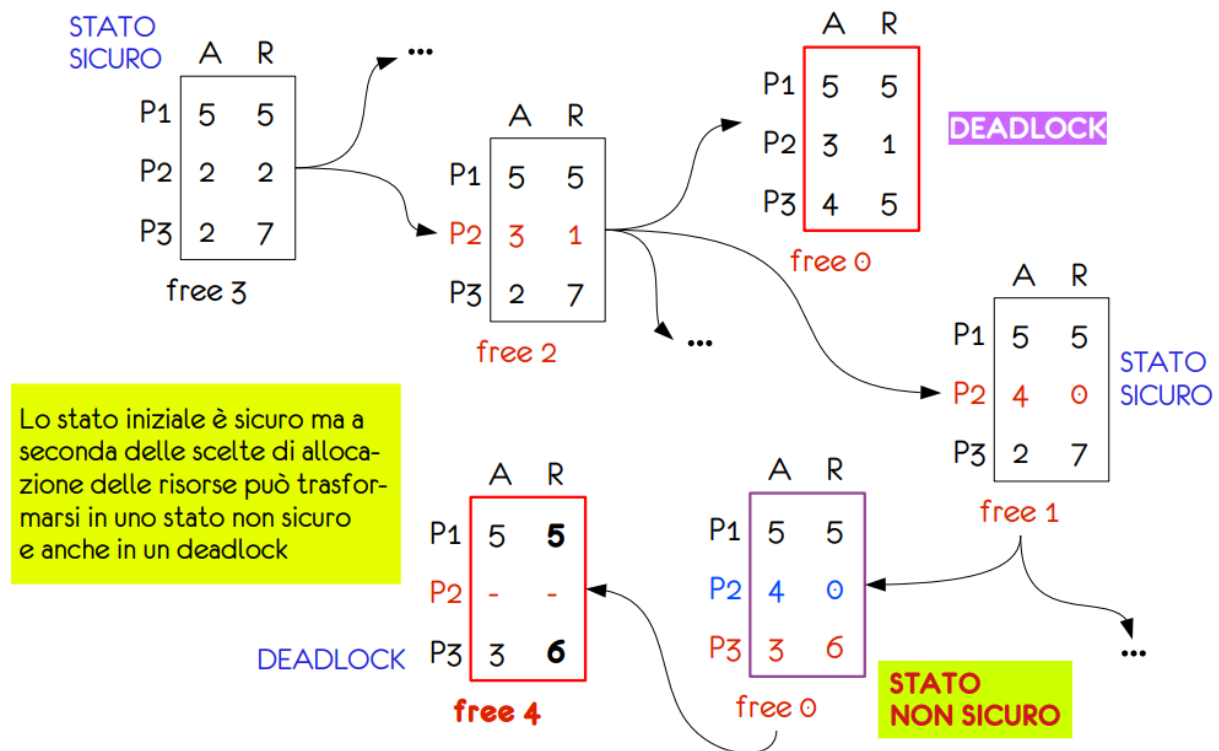
Sequenza di esecuzione



Stati sicuri e sequenze sicure

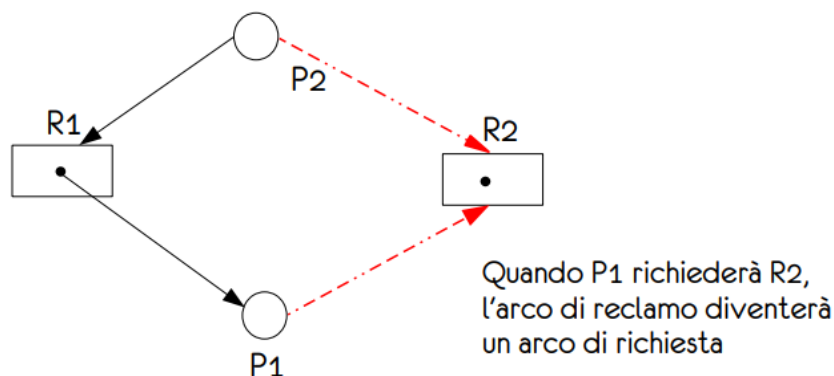
Uno stato è sicuro se da esso si dirama almeno una sequenza sicura, quindi **se esiste almeno un ordinamento dei processi che è una sequenza sicura**.

Uno stato non sicuro non è necessariamente di deadlock ma può portare ad esso.



Algoritmo di deadlock avoidance

Variante del grafo di assegnazione che utilizza un terzo tipo di arco: l'**arco di reclamo** (claim edge). Funziona solo se ogni classe di risorsa ha una sola istanza.



All'inizio tutti i processi inseriscono nel grafo di assegnazione un claim edge per ciascuna risorsa di cui avranno bisogno. Verranno **trasformati** in arco di richiesta **solo se non si genera un ciclo** (lo stato è quindi sicuro).

Algoritmo del banchiere (avoidance)

Si applica quando i processi richiedono $n > 1$ risorse di un certo tipo.

I processi sono visti come clienti di una banca a cui possono chiedere un prestito fino ad un certo limite.

Ogni nuovo processo deve dichiarare all'inizio il numero massimo di risorse di cui avrà bisogno. Ha complessità $O(N^2M)$, dove N è il numero di processi e M il numero delle classi di risorse gestite.

- `disponibili[M]`: indica la disponibilità per ogni classe di risorsa
- `massimo[N][M]`: per ciascun processo indica il numero massimo di risorse di ciascun tipo che saranno richieste
- `assegnate[N][M]`: indica quante risorse di ciascuna classe sono assegnate a ogni processo
- `necessarie[N][M]`: indica quante risorse di ciascun tipo ancora mancano ai vari processi ($\text{necessarie} = \text{massimo} - \text{assegnate}$)

L'algoritmo soddisfa una richiesta di un processo se l'assegnazione delle risorse richieste porta ad uno stato sicuro.

Viene diviso in due parti:

- 1) Un algoritmo per verificare che uno stato è sicuro
- 2) Un algoritmo per gestire una richiesta

Algoritmo di verifica della sicurezza

1. Siano `Lavoro[M]` e `Fine[N]` due array;
2. `Lavoro = Disponibili`;
3. `Fine[i] = false`, per ogni $i \in [1, N]$;
4. Cerca $i \in [1, N]$ tale che `Fine[i] = false && Necessarie[i] ≤ Lavoro`;
5. Se l'hai trovato:
 - a. `Lavoro = Lavoro + Assegnate[i]`;
 - b. `Fine[i] = true`;
 - c. `goto 4`;
6. Altrimenti:
 - a. `goto 7`;
7. Se per ogni $i \in [1, N]$ `Fine[i] = true`, lo stato è sicuro.

Algoritmo di gestione delle richieste

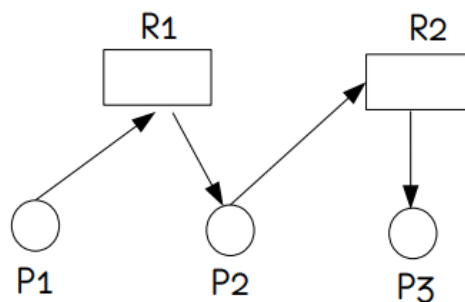
1. Consideriamo un processo j , sia `Richieste[M]` un vettore, `Richieste[i]` è il n° di risorse di classe i richieste da j in un certo istante;
2. Se `Richieste > Necessarie[j]` → ERRORE;
3. Se `Richieste > Disponibili`, j aspetta;
4. Altrimenti **simula l'esecuzione della richiesta**:
 - a. `Disponibili = Disponibili - Richieste`;
 - b. `Assegnate[j] = Assegnate[j] + Richieste`;
 - c. `Necessarie[j] = Necessarie[j] - Richieste`;
5. **Verifica se lo stato raggiunto è sicuro**:

- se sicuro, effettua l'assegnazione
- se non è sicuro, ripristino dei val. precedenti e sospensione del processo

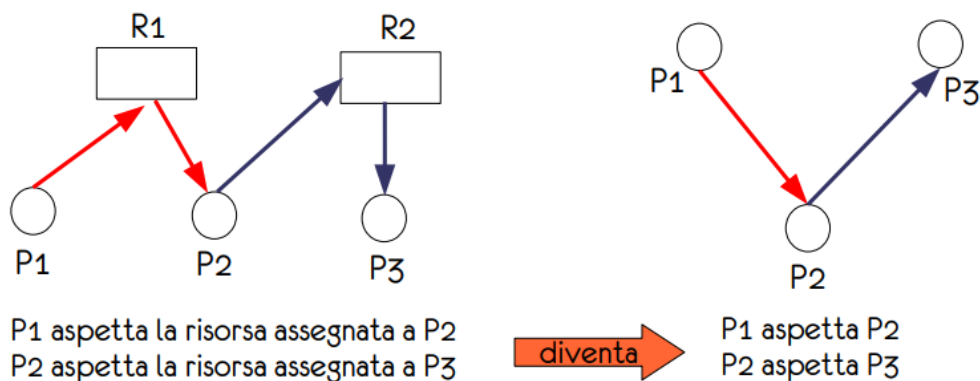
Rilevamento del deadlock per istanze singole di risorsa

Utilizzo del **grafo d'attesa**, una semplificazione del grafo di assegnazione che ha solo un tipo di vertici: i processi. Si può costruire un grafo d'attesa partendo da un grafo di allocazione.

- Dato che l'istanza è singola, per evitare ridondanza manteniamo solo una notazione per la classe e una per le istanze



- Trasformiamo gli archi $P_i \rightarrow R_s, R_s \rightarrow P_j$ in $P_i \rightarrow P_j$:



Rilevamento del deadlock per istanze multiple di risorse

Il metodo precedente è applicabile solo se per ogni classe di risorsa esiste un'unica istanza. In generale per ogni classe di risorsa R^i si possono avere molte istanze: $|R^i| = N^i \geq 1$

Strutture utilizzate:

- $\text{Disponibili}[M] = \{n1, \dots, nk\}$ mantiene il numero di istanze disponibili di ogni risorsa

- $Assegnate[N][M]$: ogni riga della matrice indica quante istanze di ciascun tipo di risorsa sono state assegnate a un certo processo; $Assegnate[i]$ indica l'attuale assegnazione per processo P_i
- $Richieste[N][M]$: ogni riga della matrice indica la richiesta attuale di ogni processo, tale richiesta può comprendere istanze di risorse differenti (non si tratta di claim, cioè di richieste future)
- NB: Assegnate e Richieste catturano le situazioni di possesso e attesa correnti

Algoritmo utilizzato:

```

int Lavoro[M];
boolean Fine[N];
/* inizializzazione */
Lavoro = Disponibili;
for (i in [1, N]){
    if (Assegnate[i] = {0, 0, ..., 0}) Fine[i] = true;
    else Fine[i] = false;
}

/* calcolo */
while (esiste un indice i tale che Fine[i] = false && Richieste[i]
≤ Lavoro){
    Lavoro = Lavoro + Assegnate[i];
    Fine[i] = true;
}

/* test: c'è deadlock? */
for (i in [1, N]){
    if (Fine[i] = false) <<c'è deadlock>>
}

```

Rottura del deadlock

Tre possibili soluzioni:

1. **terminare** i processi coinvolti;
2. effettuare la **prelazione** delle risorse;
3. **riassegnare** le risorse.

Soluzione 1: terminazione

Terminare un processo è molto costoso in quanto il lavoro da esso svolto viene perduto. Occorre perciò adottare diverse politiche a seconda del bisogno:

- terminare **tutti i processi** coinvolti;

- terminare **un processo per volta** fino alla risoluzione del deadlock, applicando l'algoritmo dopo l'abort di ciascun processo.

(L'abort di un processo può far sorgere problemi di inconsistenza nelle transazioni atomiche, bisogna effettuare il rollback)

Soluzione 2: prelazione

Sottrarre risorse ad altri processi per assegnarle ad altri. Anche qua bisogna identificare una vittima, su criteri economici: la prelazione deve essere poco costosa. Bisogna anche evitare di yoinkare risorse sempre allo stesso processo (starvation).

Far finta di niente

Quando un utente si accorge che si è verificato un deadlock, lo risolve manualmente.

Indirizzi logici e fisici

Un **indirizzo logico** è un indirizzo prodotto dalla CPU.

Un **indirizzo fisico** è l'indirizzo di ogni parola di memoria.

Attraverso un processo chiamato **binding**, viene eseguito il collegamento tra lo spazio degli indirizzi logici (ind. prodotti dalla CPU) e lo spazio degli indirizzi fisici (ind. effettivi in ram).

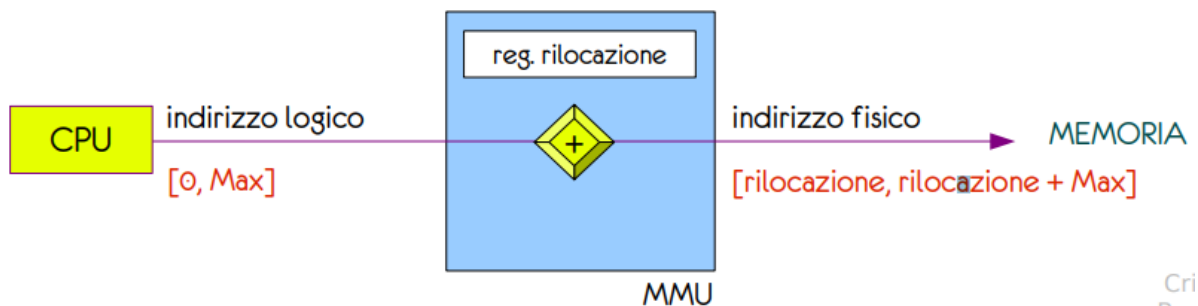
Binding

Mapping dallo spazio degli indirizzi logici di un processo allo spazio dei suoi indirizzi fisici.

Quando il binding viene fatto a compile time, indirizzo logico = fisico.

Quando il **binding viene fatto a exec time**, la corrispondenza deve essere calcolata (i due spazi di indirizzi non corrispondono).

Il binding è a carico dell'**MMU** (Memory Management Unit), che in genere funziona con un meccanismo basato sul registro base (registro di rilocazione).



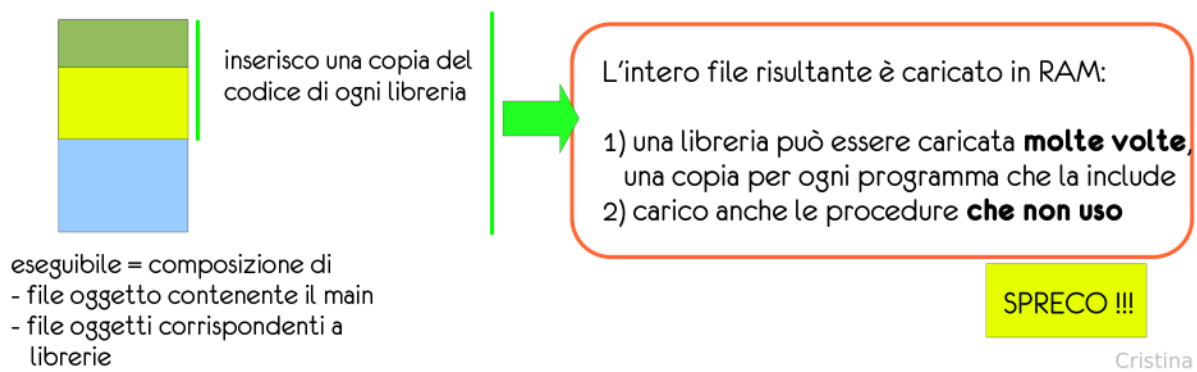
Linking e loading

Il linking è il processo di composizione dei moduli che servono ad un programma. Associa ai nomi di variabili e procedure usate da ciascun modulo (e non definiti in esso) le corrette definizioni.

Il loading è copiare un programma eseguibile (o parte) nella RAM

Questi due processi si dicono dinamici quando sono svolti a tempo di esecuzione, statici quando precedono l'esecuzione.

Approccio tradizionale



Linking e loading dinamici

Sono detti dinamici quando sono effettuati nella fase di esecuzione:

- **loading dinamico**: una procedura è caricata in RAM alla sua prima invocazione;
- **linking dinamico**: il collegamento del codice di una procedura al suo nome è effettuato alla sua prima invocazione (linker statico aggiunge solo uno stub della procedura).

Loading dinamico

Tutte le procedure risiedono in memoria secondaria sotto forma di codice rilocabile. Alla prima chiamata della procedura, viene caricata in RAM. Molto vantaggioso rispetto a dover caricare l'intera libreria in RAM.

Chiaramente, questo funziona solo se il SO ci fornisce gli strumenti per realizzare librerie a caricamento dinamico.

Linking dinamico

Rimanda il collegamento reale di una libreria alla fase di esecuzione. Dopo la compilazione, il linker statico arricchisce il programma aggiungendo gli **stub** relativi alle procedure appartenenti alle librerie dinamiche usate.

Durante l'esecuzione, lo stub verifica se il codice della procedura è già stato caricato in RAM:

- se sì, sostituisce se stesso con l'indirizzo della procedura in questione;
- se no, causa il caricamento del codice della procedura e poi si sostituisce con l'indirizzo del codice della procedura in questione.

Il vantaggio del linking dinamico risiede nella facilità di aggiornare le librerie condivise: se aggiorni una libreria dinamica, tutti i programmi che la usano faranno riferimento alla nuova versione senza dover ricompilare.

Allocazione della RAM

Esistono 3 approcci per la gestione della memoria principale:

1. Allocazione **contigua**;
2. **Paginazione**;
3. **Segmentazione**.

Ogni approccio fa riferimento a un modello di rappresentazione, che richiede apposite strutture dati e meccanismi di gestione.

Allocazione contigua

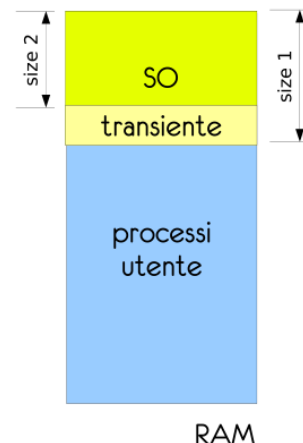
In questo modello si suddivide la RAM in due parti:

- una parte riservata al SO, in genere allocata nella parte bassa
- una parte riservata ai processi utente

Bisogna quindi proteggere la parte di memoria riservata al SO dalle letture/scritture dei processi utente, facilmente realizzabile usando un registro di rilocalizzazione.

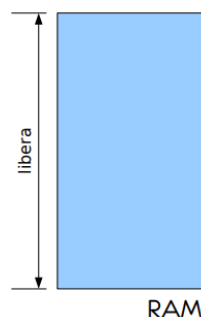
Il codice del SO può essere, a sua volta, suddiviso in una parte sempre necessaria, ed una parte che può essere utile o meno a seconda delle circostanze, chiamate **codice transiente**.

Questa parte può essere rimossa dalla RAM quando non serve, e quindi ci occorre modificare la partizione riservata al SO

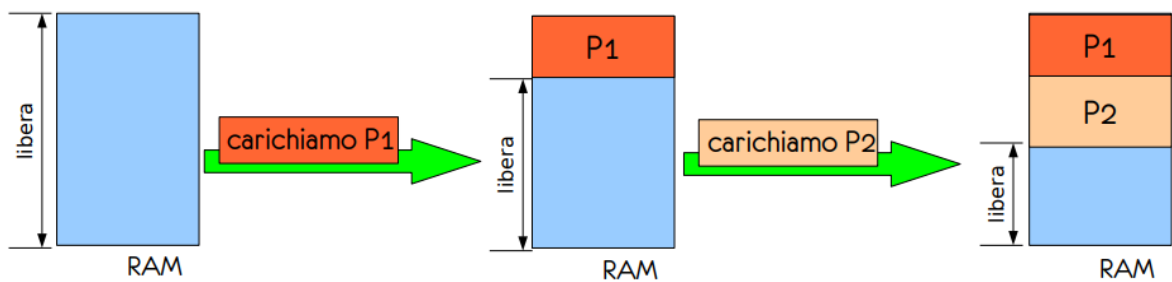


Allocazione a partizioni multiple

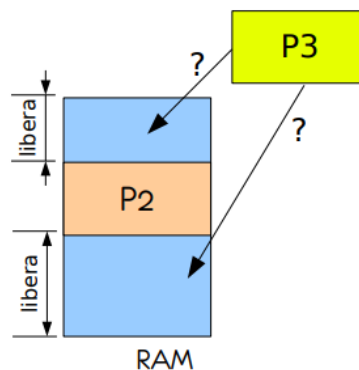
All'inizio, la RAM non utilizzata dal SO è libera.



Il SO deve mantenere una lista di porzioni libere dette buchi.



Un nuovo processo può essere caricato in RAM solo se esiste un buco abbastanza grande.



Criteri di scelta

Per caricare i processi in RAM, vengono utilizzati 3 criteri principali che aiutano a decidere quale porzione di memoria utilizzare:

- **Best-fit:** scelgo la porzione più piccola tra quelle adeguate a contenere l'immagine del processo;
- **First-fit:** scelgo la prima porzione abbastanza grande da contenere l'immagine del processo;
- **Worst-fit:** scelgo la porzione più grande tra quelle libere.

Per capire qual è la migliore, bisogna analizzare la **frammentazione** della memoria.

La frammentazione è lo spezzettamento della memoria in tante parti; ne esistono due tipi:

- **esterna**, se queste parti sono abbastanza grandi da poter essere utilizzabili;
- **interna**, se sono troppo piccole per essere utilizzate, e di solito vengono unite alle partizioni precedenti.

La **frammentazione è un problema**, in quanto è molto facile avere ampie quantità di memoria inutilizzabile: secondo la regola del 50% usando il first-fit, per ogni N blocchi di memoria si hanno $N/2$ blocchi di memoria inutilizzabile ($1/2$ della memoria totale).

In generale worst-fit è la strategia peggiore, first-fit e best-fit sono comparabili ma first-fit risulta computazionalmente meno costosa.

Combattere la frammentazione

Si può combattere la frammentazione attuando una politica di compattamento: spostare le immagini in memoria in modo che risultino contigue. Questo però, è applicabile solo se il binding tra indirizzi logici e fisici è effettuato a tempo di esecuzione

Swapping

In quanto la RAM ha dimensione limitata, è possibile che i processi in stato running/ready occupino più memoria di quanto ne sia disponibile.

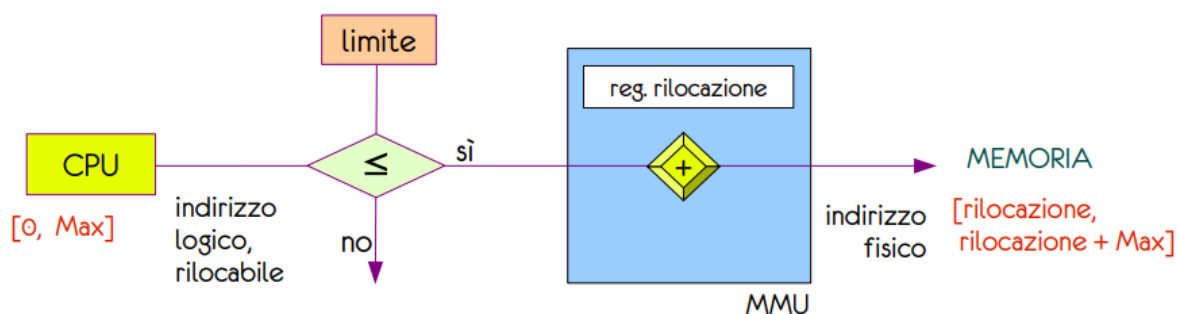
La soluzione consiste nel mantenere una parte dei processi ready in memoria secondaria e di tanto in tanto eseguire lo **swapping**, ovvero lo scambio dei processi tra le due memorie.

Con lo **swap in**, carico l'immagine di un processo ready dalla memoria secondaria (detta anche backing store) alla RAM.

Con lo **swap out**, scarico l'immagine di un processo che non è in esecuzione dalla RAM alla memoria secondaria.

La collocazione del processo in RAM dipende da quando viene effettuato il binding delle variabili:

- se il codice **non è rilocabile**, l'immagine del processo deve occupare la **stessa sezione** di RAM;
- se il codice **è rilocabile**, è possibile inserirlo in **qualsiasi** posizione.



registro limite e registro di rilocazione vengono caricati durante il context switch
il contenuto del registro di rilocazione può variare nel tempo

Tempo di swapping

Il tempo necessario al completamento dello swap è dato dal tempo di swap-out + tempo di swap in. Questo tempo è influenzato dalla dimensione della RAM usata dai singoli processi. Per esempio, se il tempo di trasferimento è pari a 1 MB/sec, quanto tempo impieghiamo a trasferire 100 KB di memoria usata dal processo?

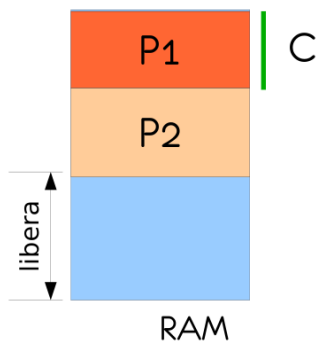
$$100KB/1MB/sec = 100KB/1000 KBsec = 0.1sec = 100 msec$$

Se anche lo swap-out è uguale allo swap-in, ci impiegheremo circa 200 msec.

Lo swapping non può essere effettuato se il processo sta effettuando operazione di I/O: queste operazioni non possono essere effettuato su variabili residenti in memoria secondaria.

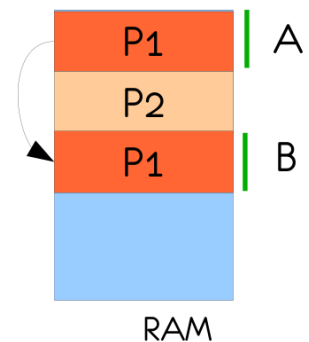
Paginazione della memoria

La paginazione è un meccanismo di gestione della RAM alternativa alla allocazione contigua, e sua caratteristica fondamentale è che consente allo spazio degli indirizzi fisici di un processo di non essere contiguo.



allocazione contigua

sp. indirizzi di P1 = C

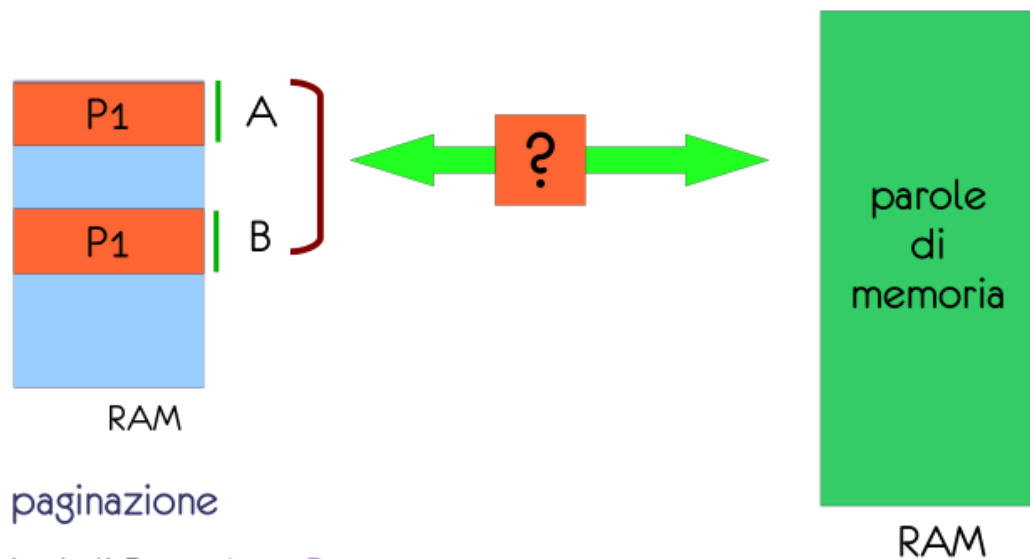


paginazione

sp. indirizzi di P1 = A \cup B

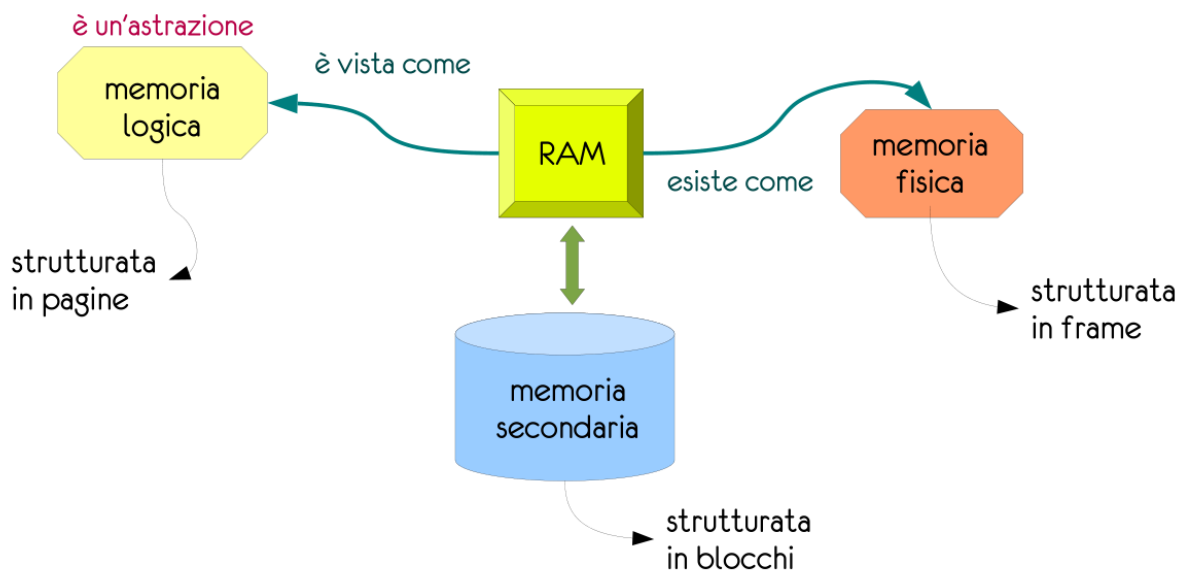
Questo consente di ridimensionare in modo dinamico lo spazio riservato ad un processo, aggiungendo o togliendo pagine su richiesta. Chiaramente, c'è bisogno di un opportuno supporto hardware.

Pagine e strutture di supporto



sp. indirizzi di P1 = $A \cup B$

Ci si presentano due problemi ora: come associare le porzioni di processo a RAM e come organizzare le diverse porzioni in un tutt'uno



Per fare il binding tra indirizzi logici e fisici, usiamo una tabella degli indirizzi

Paginazione e indirizzi logici

La dimensione è la stessa per tutte le pagine ed è definita dall'architettura (tra $2^9=512B$ e $2^{24}=16MB$)

La memoria logica viene suddivisa secondo questa formula:

- dimensione di una pagina = 2^n
- dimensione della mem. logica = 2^m

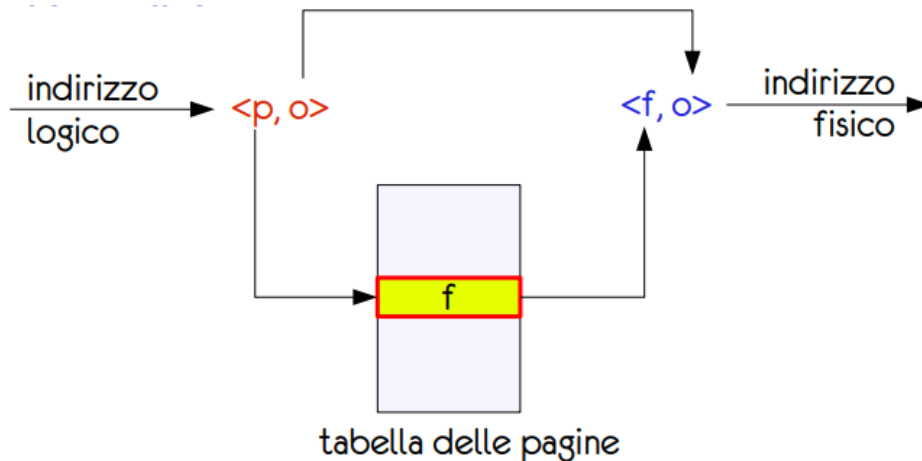
- $n^{\circ} \text{ pagine} = 2^{m-n}$
- bit per rappresentare n° di pagina = $m-n$
- bit per rappresentare scostamento all'interno di una pagina = n

L'indirizzo logico di una pagina è quindi una coppia $\langle p, o \rangle$ in cui p è il n° di pagina e o è l'offset.

Paginazione e rilocalizzazione

La rilocalizzazione nel contesto della paginazione significa **consentire l'accesso ad una pagina indipendentemente dal frame in cui è caricata**.

Per effettuare la rilocalizzazione, il registro di rilocalizzazione è sostituito dalla entry nella tabella delle pagine, corrispondente a p . Il valore f individua l'indirizzo di inizio del frame.



Paginazione e frammentazione

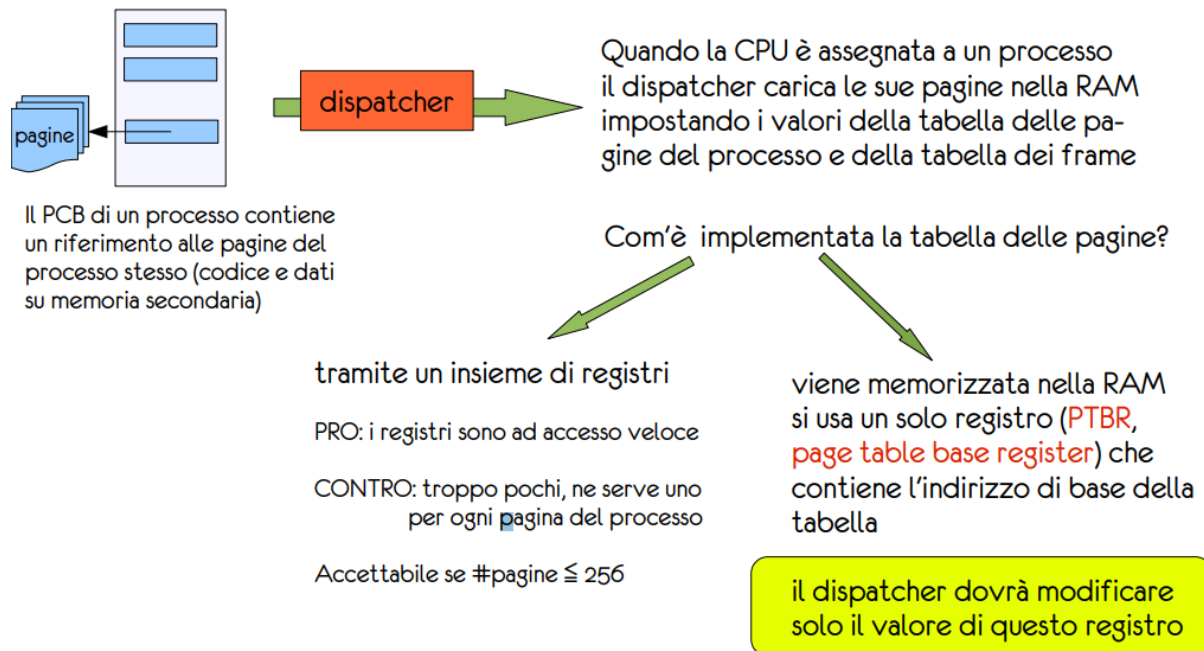
La paginazione elimina il problema della frammentazione esterna, ma **rimane frammentazione interna**.

Di ogni processo, solo l'ultima sua pagina allocata può presentare frammentazione interna, perché raramente la dimensione di un processo sarà un multiplo della grandezza di una pagina. Si può perciò dire che in media si ha mezza pagina inutilizzata per processo.

Occorre quindi **trovare una dimensione ottimale per le pagine** in modo da ridurre la frammentazione:

- utilizzando delle pagine di piccola dimensione, si può effettivamente ridurre la frammentazione, ma il numero delle pagine aumenta drasticamente;
- utilizzando delle pagine di grandi dimensioni, è più veloce trasferire grandi quantità di dati dalla mem. secondaria, ma ci sarà più memoria inutilizzata.

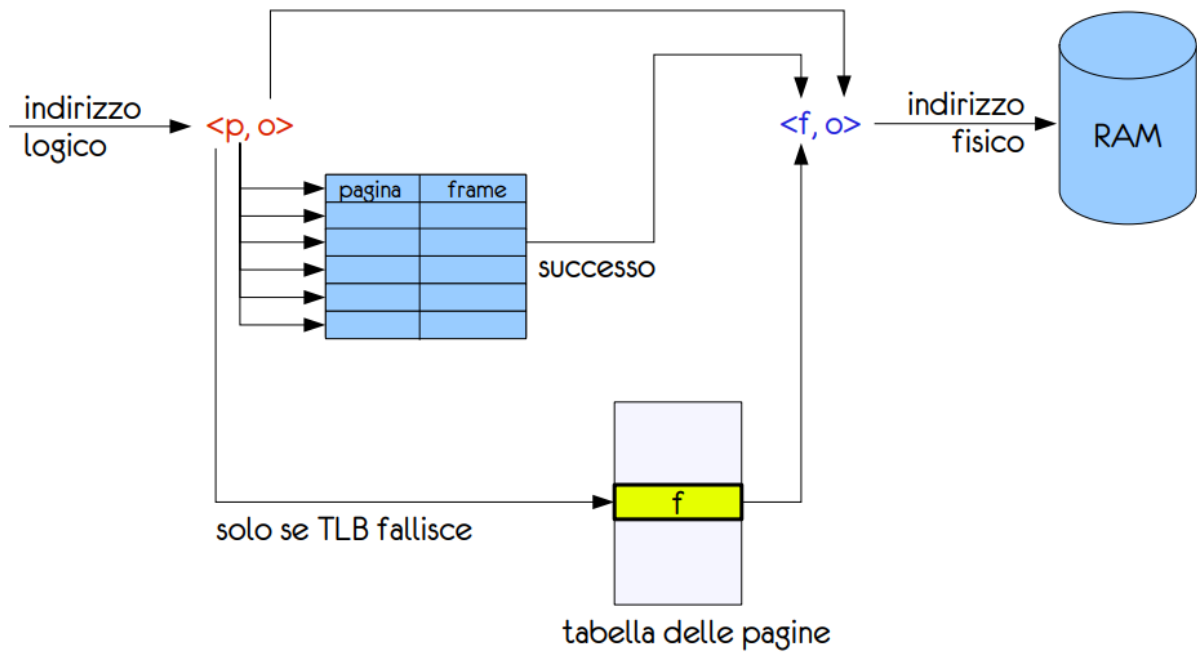
Architettura di paginazione



Tempi di accesso

Utilizzando il PTBR, viene fatto l'accesso alla ram 2 volte, una per individuare la page table tramite l'indirizzo contenuto in PTBR, un'altra poi per prelevare l'indirizzo fisico d'interesse.

Questo doppio accesso rallenta troppo l'esecuzione, perciò viene implementata la **TLB (Translation Look-aside Buffer)**, una cache molto veloce contenente delle coppie <chiave, valore>. Quando riceve un input lo confronta contemporaneamente con tutte le chiavi. Se trova una chiave corrispondente restituisce il valore associato.



Hit ratio

E' la percentuale di successo di reperimento di una pagina tramite TLB (TLB hit), e consente di calcolare il tempo medio effettivo di accesso ad una pagina.

Esempio:

tempo di accesso al TLB = 20 nsec

tempo di accesso alla RAM = 100 nsec

hit ratio = 82% = 0.82

percentuale di accessi tramite page table = 18% = 0.18

$T_{TLB} = \text{accesso TLB} + \text{accesso RAM} = 20 + 100 = 120 \text{ nsec}$

$T_{PT} = 2 \cdot \text{accesso RAM} = 100 + 100 = 200 \text{ nsec}$

$T_M = (0.82 \cdot T_{TLB}) + (0.18 \cdot T_{PT}) = 0.82 \cdot 120 + 0.18 \cdot 200 = 134.4 \text{ nsec}$

Aggiornamento del TLB

Quando non trovo una pagina di interesse (TLB miss)

- se c'è spazio nel TLB si inserisce la nuova coppia $\langle \text{pagina}, \text{frame} \rangle$;
- se non c'è spazio nel TLB, si sostituisce una coppia (in genere quella meno recente) con quella nuova;

Protezione

Nella tabella delle pagine del processo viene anche memorizzata la **modalità di accesso alle pagine**, che può essere

- read only
- r/w
- exec
- invalid (non appartiene allo spazio degli indirizzi del processo)

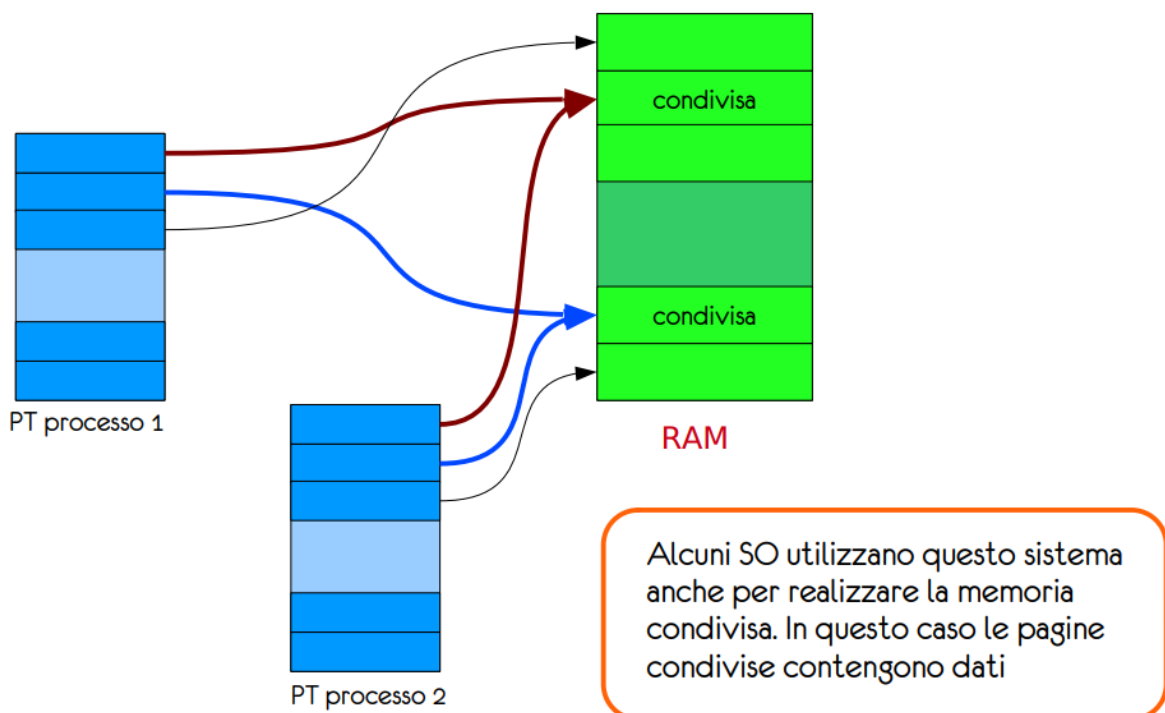
Si possono avere pagine non valide quando la tabella delle pagine di un processo è più piccola dello spazio riservato ad essa nella RAM. Se un processo tenta di accedere ad una pagina non valida, viene generato un interrupt.

Condivisione delle pagine

Spesso nei sistemi time-sharing diversi utenti usano lo stesso programma contemporaneamente, o in un contesto di multiprogrammazione più processi usano la stessa libreria contemporaneamente.

Se fosse possibile condividere il codice, sarebbe possibile diminuire lo spazio occupato in memoria in quanto non ci sarebbe il bisogno di allocare le pagine per ogni processo/utente che utilizza il programma ma una volta sola.

NB: solo le pagine di codice accessibili in sola lettura (**codice rientrante**) possono essere condivise.

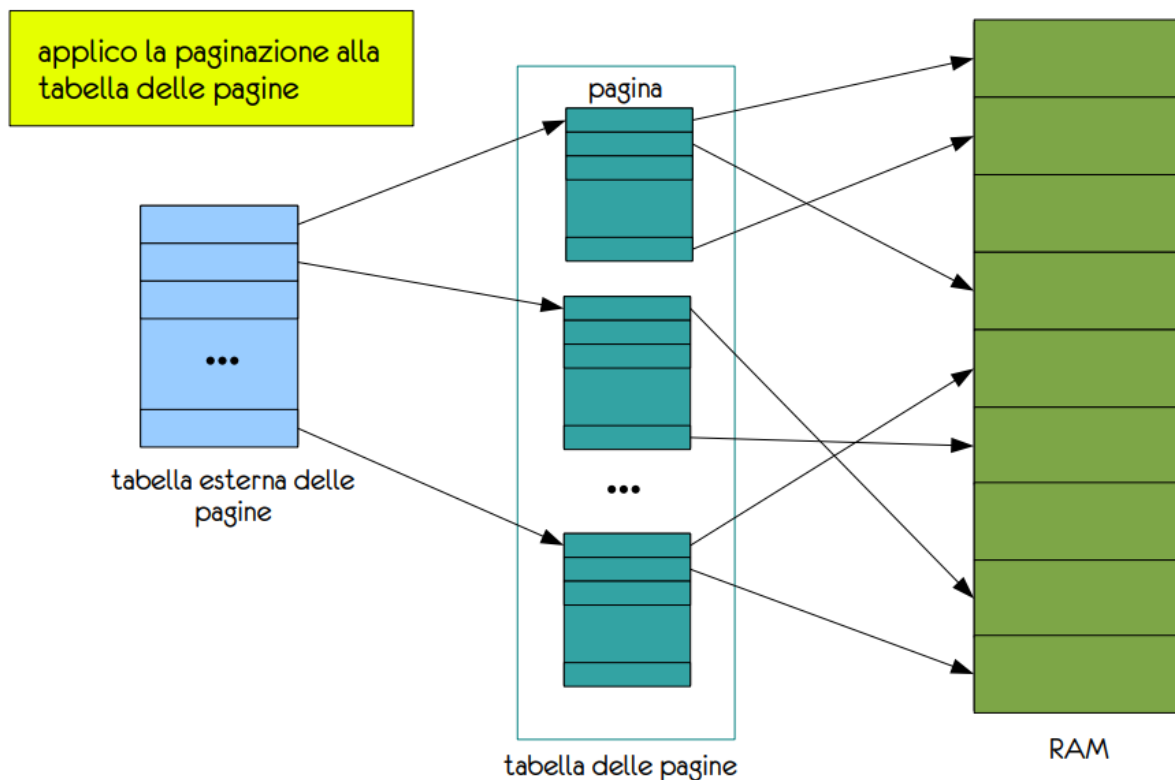


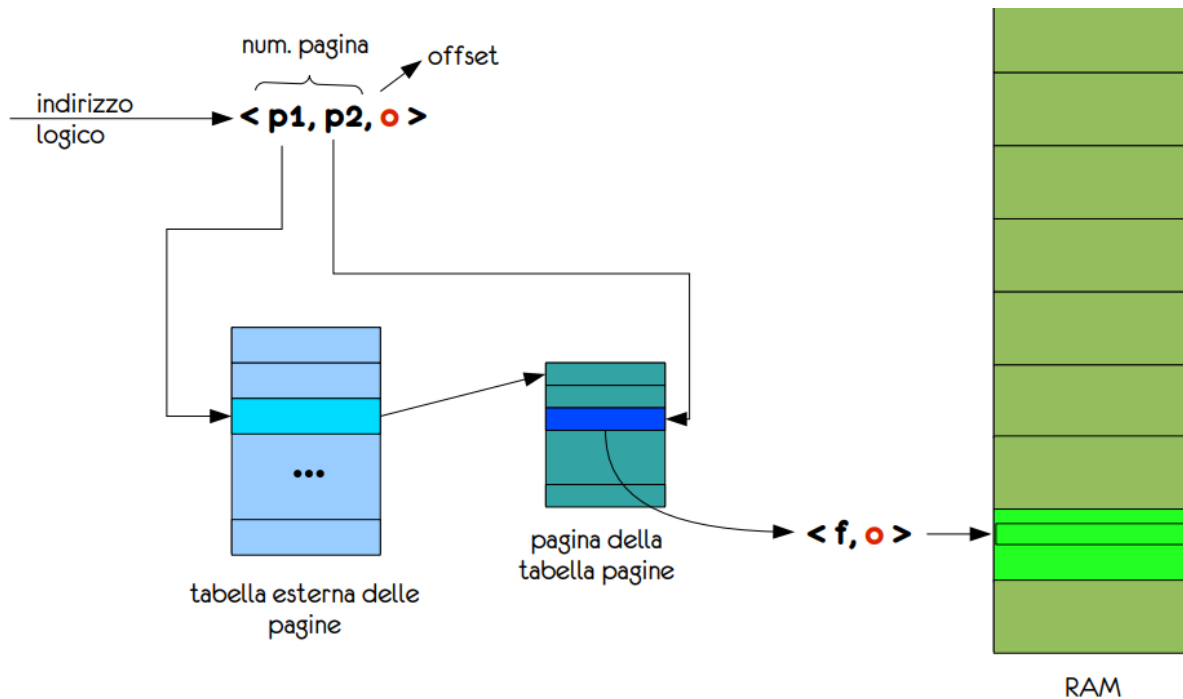
Paginazione multi-livello

La dimensione della tabella delle pagine dipende dalla dimensione dello spazio degli indirizzi logici. Per esempio, con indirizzi a 32 bit, potremmo avere massimo 2^{20} entry. E se ogni entry occupa 4 byte, la tabella occuperà 4 MB.

In realtà, nessun processo usa l'intero spazio degli indirizzi. La maggior parte della tabella andrebbe sprecata. Per risolvere questo problema, possiamo suddividere la tabella in tante parti organizzate.

Paginazione a due livelli





Vantaggi e svantaggi delle tabelle delle pagine

Vantaggi:

- rappresentazione naturale se si adotta un approccio processo-centrico;
- ordinata in modo tale che sia semplice per il SO identificare il punto in cui si trova l'indirizzo del frame di interesse.

Svantaggi:

- può diventare troppo grande e invece di aiutare a gestire la RAM diventa essa stessa un problema da gestire.

Segmentazione

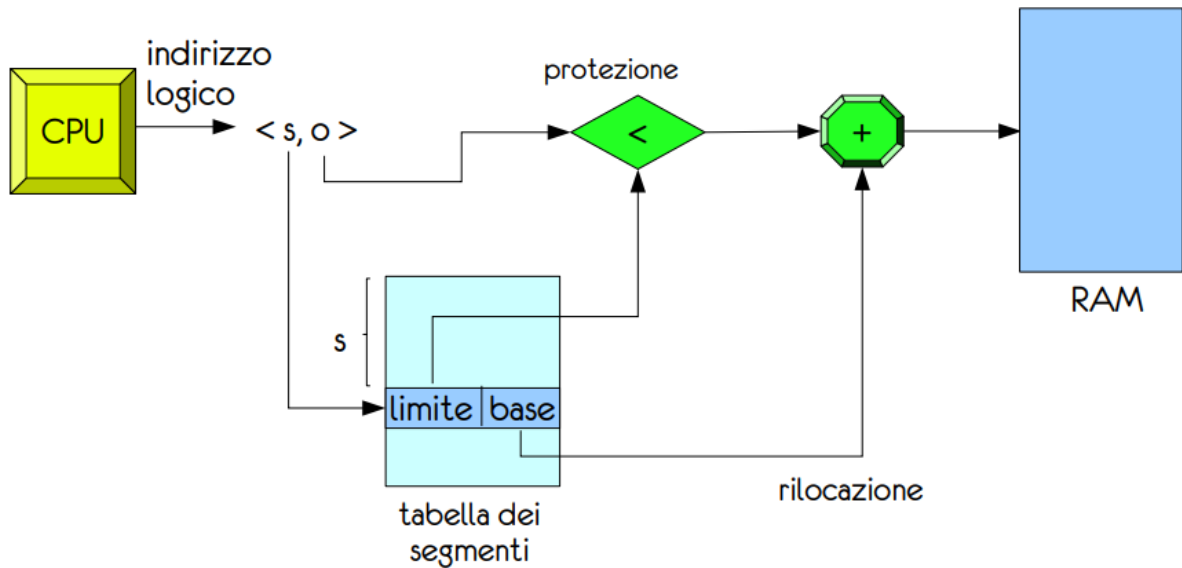
Mentre la paginazione struttura la memoria in un insieme di elementi, il cui contenuto può essere indifferentemente costituito da codice e/o dati, la segmentazione modella la memoria secondo uno schema più vicino al modo in cui il programmatore vede il programma: **ogni processo ha una porzione di RAM organizzata come un insieme di segmenti di memoria**, di dimensione (molto) variabile.

Per organizzare la memoria in questo modo, il compilatore del programma ha il compito di organizzare la memoria in tali segmenti. Ad esempio, per un programma scritto in C serviranno i segmenti per:

- codice
- var. globali
- stack (per ogni thread)
- heap
- libreria standard del C

- librerie esterne

L'indirizzo logico di un segmento è $\langle \text{id_segmento}, \text{offset} \rangle$.



s = indice della entry relativa al segmento

ogni segmento è caratterizzato da due valori: **indirizzo di base** e **indirizzo limite** perché la loro dimensione è varia

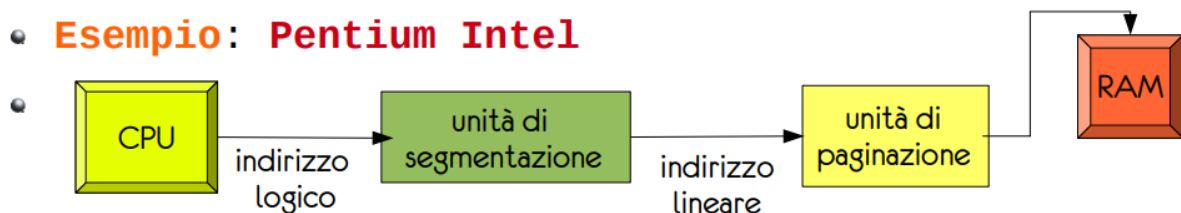
La segmentazione è spesso più efficiente della paginazione.

I problemi della segmentazione sono simili a quelli dello schema a partizioni multiple, anche se la segmentazione è molto più flessibile:

- occorre gestire dinamicamente la memoria libera
- si ha frammentazione esterna

Alcune architetture sfruttano una tecnica mista tra paginazione e segmentazione

• **Esempio: Pentium Intel**



Ogni segmento è strutturato in pagine

dimensione max segmento 4GB
numero max di segmenti per processo 16K

Il processore ha 6 registri di segmento che permettono a un processo di fare riferimento a 6 segmenti contemporaneamente

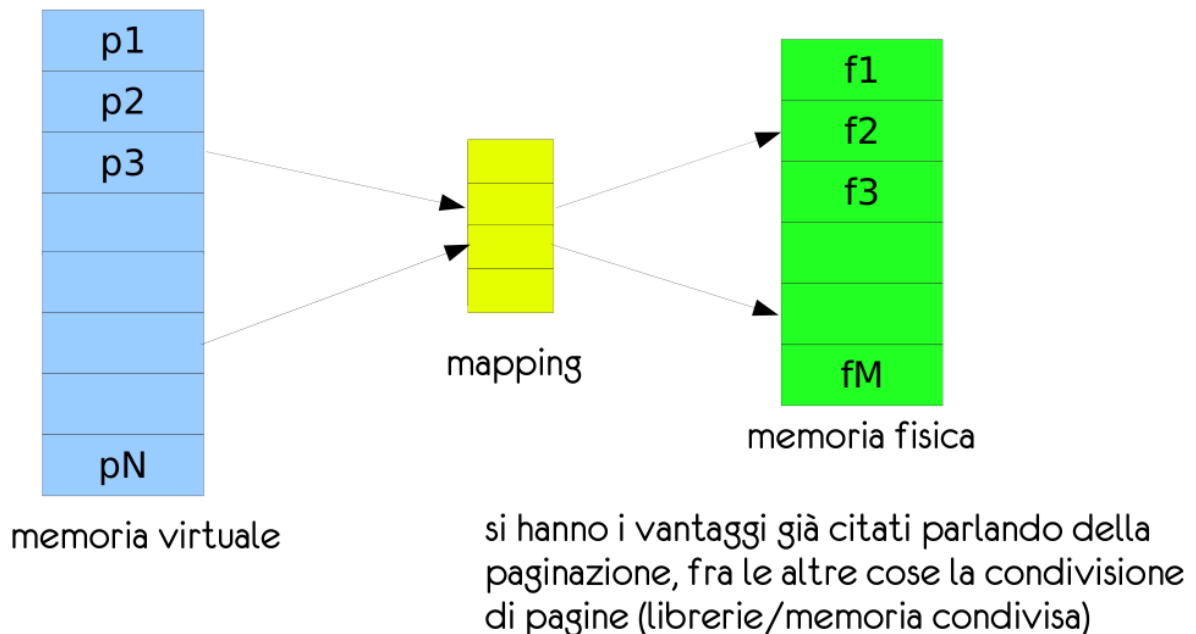
8K riservati \in **tabella locale dei descrittori**

8K condivisi \in **tabella globale dei descrittori**

Memoria virtuale

In precedenza, abbiamo sempre lasciato intendere che i processi fossero sempre caricati completamente in RAM. Ma sarebbe conveniente rilasciare questo vincolo, così ogni processo può occupare meno RAM.

Separando la memoria logica dalla memoria fisica, nasce proprio il concetto di **memoria virtuale**



Spazio degli indirizzi di un processo

E' predefinito e in genere molto più grande dello spazio occupato dal processo.

Con l'esecuzione, stack e heap cresceranno uno verso l'altro, Per contenere nuovi dati potranno essere allocati nuovi frame.

Questo non cambia lo spazio degli indirizzi virtuali del processo, semplicemente una parte di indirizzi prima non corrispondenti ad alcun indirizzo assoluto saranno ora mappati su parole di memoria effettive.

Lazy swapping

I processi vengono caricati in modo parziale: quando una pagina diventa utile, la si carica in RAM al posto di una pagina già presente, che verrà scaricata in memoria secondaria.

Occorre definire un meccanismo per mantenere traccia della locazione delle pagine (RAM o mem. secondaria).

La parte del SO che gestisce il caricamento delle pagine è detto **pager**.

Bit di validità

Per ogni pagina nella page table di un processo è presente un bit che indica la validità della pagina:

- una pagina è **valida** se è presente in RAM e appartiene allo spazio degli indirizzi del processo;
- una pagina è **invalida** se è in memoria secondaria o non appartiene allo spazio degli indirizzi del processo.

Page fault

Quando un processo tenta di accedere ad una pagina non valida, viene generato un interrupt (**page fault exception**). Se questa pagina è conservata in memoria secondaria (ed è quindi valida), viene copiata da disco in un frame libero, si aggiorna la tabella delle pagine e il processo continua come se non fosse successo niente.

Altrimenti, se la pagina è invalida, si termina il processo.

Paging on demand

Anche detta paginazione a richiesta, è un meccanismo per cui una pagina viene caricata in RAM solo se è stata richiesta (i processi vengono caricati senza pagine)

Non servono ulteriori strutture per applicare questo meccanismo e vale il principio di località dei riferimenti, ma potrebbe essere più pesante di un meccanismo normale.

tempo di accesso effettivo = $(1 - p) * ma + p * t_{gpf}$

dove: p = prob. che si verifichi PF ($p \in [0, 1]$)

ma = tempo medio di accesso RAM ($ma \in [10, 200]$ nsec)

t_{gpf} = tempo di gestione di un page fault ($t_{gpf} \in 8$ msec)

Gestione del page fault

Procedimento per la gestione del page fault:

- si genera un'interruzione
- salvataggio dei registri e dello stato del processo
- verifica dell'interruzione: in questo caso si determina che si tratta di page fault
- controllo della correttezza del riferimento e individuazione della locazione occupata dalla pagina mancante su disco
- lettura e copiatura della pagina da memoria secondaria in RAM (operazione di I/O)
- durante l'attesa per il completamento di questa operazione, allocazione della CPU a un altro processo
- interrupt che segnala il completamento dell'operazione di caricamento della pagina
- aggiornamento della tabella delle pagine
- quando lo scheduling riavvia il processo sospeso, ripristino dello stato
- riesecuzione dell'istruzione interrotta

Per riassumere:

- 1) servizio di interruzione per page fault
- 2) lettura della pagina
- 3) riavvio del processo

Area di swap

Come già detto, le pagine di un processo che non sono in RAM sono contenute in memoria secondaria, che viene vista proprio come un'estensione della RAM (nonostante i tempi di accesso maggiori).

Ogni SO gestisce l'area di swap in maniera molto diversa: sia per il dimensionamento, sia per i suoi contenuti. Per quanto riguarda la dimensione, linux suggerisce di allocare il doppio della quantità di RAM, mentre solaris suggerisce di allocare una quantità pari alla differenza fra la dimensione dello spazio degli indirizzi logici e la dimensione della RAM.

Per l'implementazione, ci sono due approcci:

- come file: l'area di swap diventa un file speciale del file system
 - pro: si evita il problema di dimensionamento, perchè un file si può ridimensionare secondo le necessità
 - contro: la gestione del file system rallenta ulteriormente l'accesso
- come partizione a sé, non formattata: si usa un gestore speciale per usare algoritmi ottimizzati per ridurre i tempi di accesso
 - pro: maggiore velocità
 - contro: per ridimensionarla bisogna ripartizionare il disco

Terminiamo con il dire che ora si preferisce mantenere solamente i dati nello swap: il codice tanto si può prelevare dal file system.

Processi padri e figli

Quando un processo P esegue una **fork**, viene creato un processo "figlio", che esegue lo **stesso codice** del "padre" (processo che lo ha generato) e ha una **copia delle var. e dello stack** del padre.

NB: tramite codice è possibile (e comunemente usato) cambiare il codice che il figlio deve eseguire.

Con l'esecuzione della fork sorgono 2 problemi relativi alla gestione della memoria tra i processi:

- **uplicazione delle pagine**, risolta condividendo le pagine tra processi padre e figlio
- **sovrascrittura delle pagine**, risolta usando la tecnica di "copiatura su scrittura": quando uno dei due processi esegue una exec, si allocano delle nuove pagine per il processo chiamante e si carica il nuovo codice in esse.

Sostituzione di pagine

Avviene quando, all'avvenimento di un page fault, non si trova spazio in RAM per la pagina richiesta. Il meccanismo richiede la **copiatura di due pagine**: vittima copiata in mem. secondaria, nuova pagina copiata in RAM.

Per evitare molteplici casi di **duplice scrittura**, si cerca di limitarla solo ai casi in cui serve:

- serve copiare una volta per mettere la pagina nuova in RAM;
- serve copiare un'altra volta per copiare la vittima in mem. secondaria, SOLO SE essa è stata modificata rispetto ad una sua copia già presente.

Per controllare i casi di modifica viene mantenuto un bit per ogni pagina (chiamato **dirty bit**) per indicare se la pagina è stata modificata o meno.

Osservazioni:

- **memoria logica e fisica sono completamente separate;**
- spazio ind. logici dei processi > spazio ind. fisici offerto dalla RAM;
- serve realizzare un meccanismo dinamico tramite il quale caricare/sostituire pagine a seconda dell'esigenza;
- serve introdurre meccanismi per aumentare l'efficienza, in quanto lettura e copiatura sono operazioni pesanti;
- incremento del livello di multiprogrammazione

Implementazione della paginazione a richiesta

Per implementarla, serve sviluppare 2 algoritmi:

- **allocazione dei frame:** spartisce M frame liberi fra N processi;
- **sostituzione delle pagine:** secondo un criterio predefinito e informazioni prese dalle pagine, sceglie le pagine da sostituire.

Esistono quindi diverse implementazioni:

- FIFO (First In First Out)
- ottimale
- LRU (Least Recently Used)
- approssimazione LRU:
 - seconda chance
 - bit supplementare
- conteggio:
 - LFU (Least Frequently Used)
 - MFU (Most Frequently Used)

Sostituzione FIFO

Ad ogni pagina **si associa un marcatore temporale**: quando è stata caricata in memoria. Si sceglie poi di **sostituire la pagina più vecchia in memoria**.

Questo è possibile perché le pagine sono organizzate in una coda FIFO: andremo infatti a prendere sempre la prima pagina in coda.

Questa tecnica è molto semplice da realizzare, ma non si comporta sempre bene: il fatto che una pagina sia stata caricata da tempo non significa per forza che non sia più in uso.

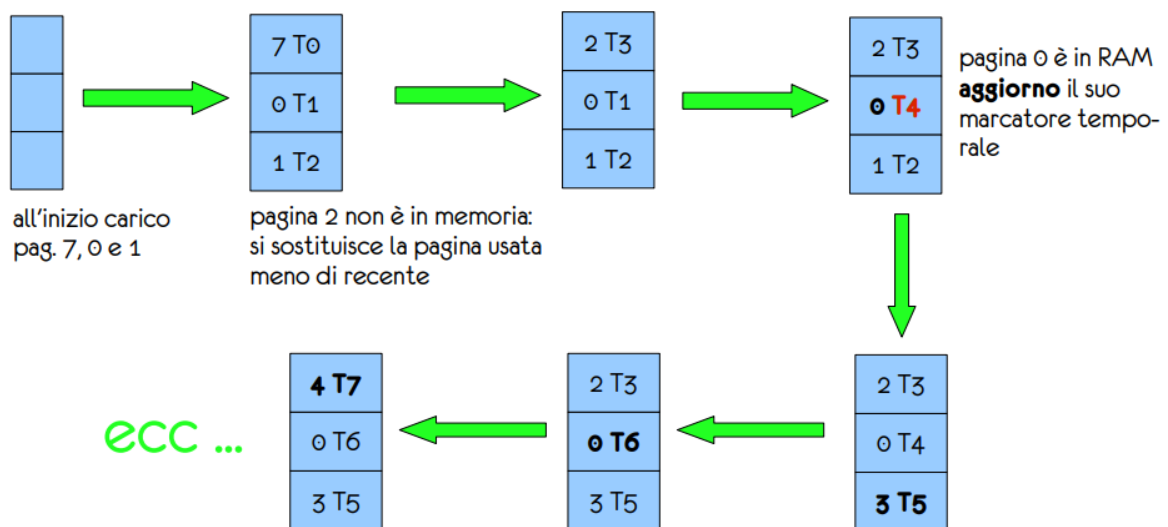
Questo algoritmo presenta anche l'**anomalia di Belady**: la frequenza delle assenze può aumentare con l'aumentare del numero di frame in RAM.

Algoritmo ottimale

È l'algoritmo migliore: non presenta l'anomalia di Belady, e ha la frequenza di page fault più bassa. Si **sostituisce la pagina che non verrà usata per il periodo di tempo più lungo**. Purtroppo non è implementabile, perché non sappiamo quando una pagina verrà richiesta.

Least-recently used

Simile all'algoritmo FIFO, basiamo la scelta sul tempo di utilizzo. Questo algoritmo è un'approssimazione dell'OPT, perché ci basiamo su quanto sia stata usata attualmente una pagina. Scegliamo sempre la pagina usata meno di recente.

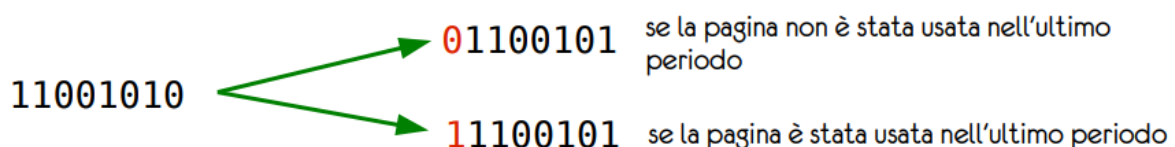


Algoritmo con bit supplementare

Si associa a ogni elemento nella tabella delle pagine una serie di bit che realizzano registri a scorrimento.

A intervalli regolari un timer passa il controllo al SO, che:

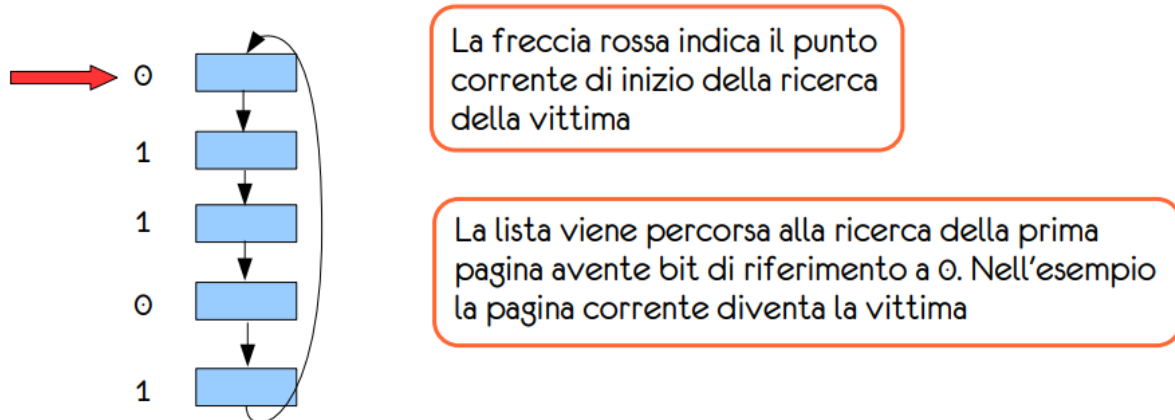
- sposta i bit nella sequenza traslandoli a destra di 1;
- copia il bit di riferimento nel bit più significativo;
- scarta il bit meno significativo.



Quindi i bit di riferimento delle pagine vengono azzerati: una pagina che ha il suo registro a 00000000 non è stata usata negli ultimi 8 periodi di tempo.

Algoritmo di seconda chance

Unisce il metodo LRU a FIFO: le pagine sono mantenute in una coda circolare FIFO e mantengono un bit di riferimento.



Funzionamento:

- Carico la pagina e le associo un bit di riferimento impostato ad 1;
- Quando diventerà necessario caricare una nuova pagina, la ricerca partirà dalla posizione successiva, in questo caso il bit di rif. ora vale 1;
- Quando il bit di riferimento è impostato a 1: la pagina viene saltata ma il suo bit di riferimento viene azzerato. Idem per la pagina successiva;
- A questo punto si incontra una pagina con bit di riferimento a 0 e la si sovrascrive mettendo il bit di riferimento a 1.

Osservazioni:

- La struttura circolare della coda è un modo per concedere un po' di tempo in RAM a ciascuna pagina: infatti una pagina con bit che passa da 1 a 0 non viene rimossa subito ma solo quando si tornerà alla sua locazione dopo aver percorso tutta la lista;
- Se tutti i bit di rif. vengono impostati a 1, diventa FIFO.

Algoritmo di seconda chance migliorato

Ogni pagina ha associata una coppia di bit $\langle r, m \rangle$:

- riferimento: indica se la pagina è stata usata di recente;
- modifica: indica se la pagina è stata modificata di recente;

Si individuano 4 classi di pagine:

- $\langle 0, 0 \rangle$: non usata, non modificata;
- $\langle 0, 1 \rangle$: non usata, modificata;
- $\langle 1, 0 \rangle$: usata, non modificata;
- $\langle 1, 1 \rangle$: usata, modificata;

Il criterio di scelta è basato sulla priorità delle 4 classi: $00 < 01 < 10 < 11$.

Sostituzione su conteggio

Basati sul conteggio del numero di riferimenti fatti a ciascuna pagina:

- **LFU (Least Frequently Used)**: sostituisce la pagina col minor numero di riferimenti. Si basa sull'idea che una pagina molto usata avrà un conteggio alto. Diventa però difficile distinguere fra una pagina che è stata molto usata in passato e una che è molto usata di recente;
- **MFU (Most Frequently Used)**: sostituisce la pagina col maggior numero di riferimenti. Si basa sull'idea che se una pagina ha un contatore basso è probabilmente stata usata di recente.

Pool of free frames

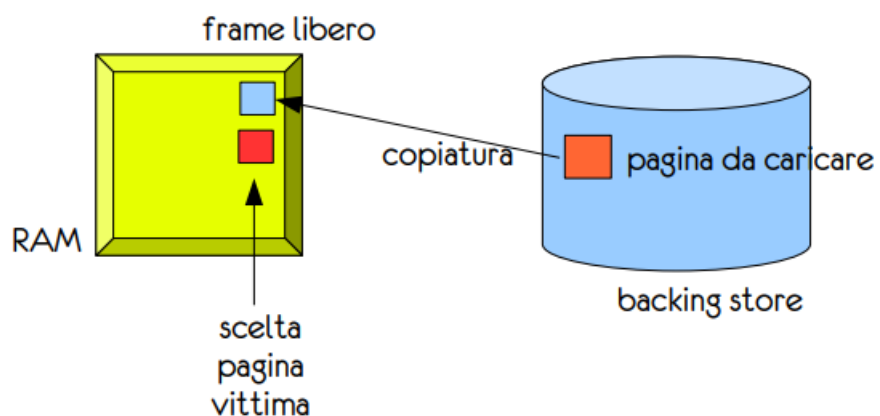
Spesso gli algoritmi di sostituzione delle pagine sono affiancati da altre procedure finalizzate a incrementare le prestazioni del sistema.

Fra queste una tecnica che consente di iniziare la copiatura in RAM della pagina necessaria per proseguire prima che la pagina vittima sia stata copiata in memoria secondaria.

L'idea è associare un piccolo pool di frame liberi. Quando diventa necessario caricare una pagina nuova:

- la si copia in un frame libero associato al processo
- si sceglie una vittima e la si copia in mem. secondaria
- si libera quindi un frame che viene aggiunto al pool di frame liberi

In questo modo, non viene cancellata la vittima, che non dovrà essere ricopiata se nasce il bisogno di accedervi nuovamente.



La pagina vittima va ad arricchire il pool dei frame liberi assegnati al processo ma non viene cancellata o sovrascritta, al contrario rimane accessibile attraverso la tabella delle pagine

Allocazione dei frame per i processi utente

L'algoritmo di allocazione dei frame si applica quando si hanno a disposizione N frame liberi, occorre caricare 1 o + processi e bisogna decidere come spartire i frame.

In generale, il n° di page fault è inversamente proporzionale al n° di frame allocati per ciascun processo. L'idea è quindi di mantenere in memoria un **numero minimo di frame** per processo, per ridurre la probabilità che si generi page fault.

Il numero minimo di frame dipende dall'istruzione e dall'architettura:

- se ha un solo operando, occorrono ≥ 2 pagine (codice e dati);
- se è un riferimento a memoria, occorrono ≥ 3 pagine (codice, dati, riferimento);
- se è grande, può stare a cavallo tra due pagine;
- può avere più livelli di indirizzamento indiretto, quindi tutta la mem. virtuale potrebbe dover essere caricata...

Allocazione proporzionale

Indichiamo con VM^i la dimensione della memoria logica occupata dal nostro processo. Indichiamo poi con M il numero di frame disponibili; il numero di frame allocati al processo

sarà quindi: $m = \frac{VM^i}{V} \times M$

Se abbiamo poi definito un numero minimo di frame necessari per caricare un'istruzione, potrebbe essere necessario incrementare tale valore per i processi più leggeri, o diminuirlo per quelli più pesanti.

Dinamicità

Se il numero di processi in RAM aumenta bisognerà redistribuire il numero di frame ancora liberi ai nuovi processi, se diminuisce si può assegnare un numero maggiore di frame.

Allocazione globale/locale

Una questione che non abbiamo ancora considerato è la priorità dei processi in connessione all'allocazione dei frame. Ci si aspetta che processi a priorità maggiore abbiano a disposizione un maggiore numero di frame. Ma tutte le strategie che abbiamo visto fino ad oggi sono eque: ogni processo ha la stessa priorità.

Cosa facciamo allora, se un processo ad alta priorità finisce i frame.

Thrashing

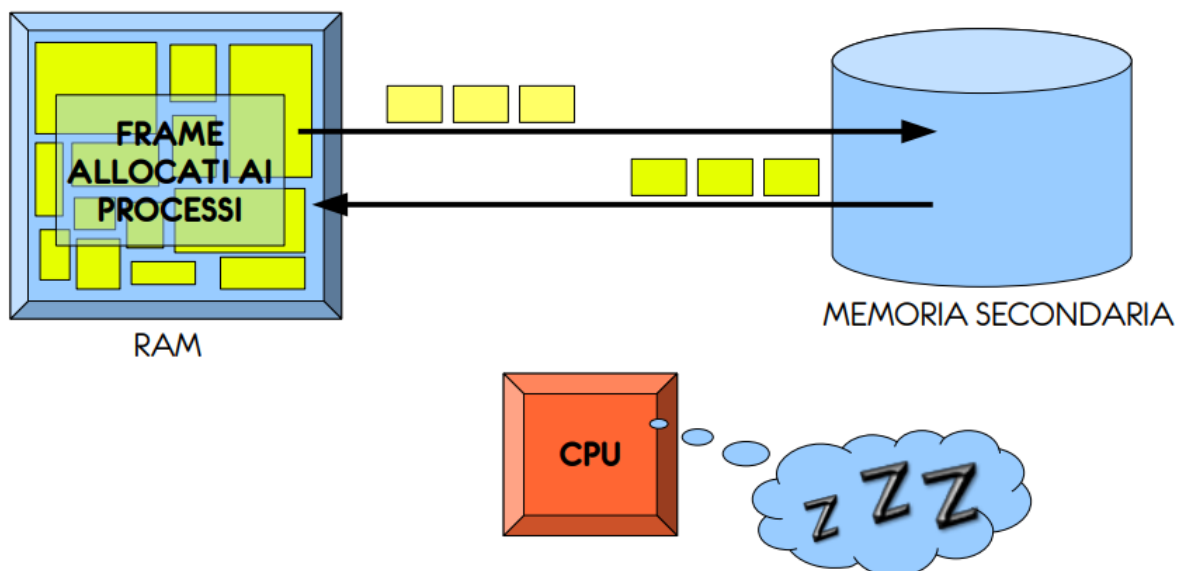
Il thrashing è un fenomeno che si verifica nella gestione della memoria virtuale: quando una pagina in realtà in uso da un processo viene sostituita, e subito dopo viene scatenato di nuovo un page fault, avviene questo fenomeno.

Questo loop può continuare per molto tempo, e si spenderà più tempo nel sostituire le pagine che nell'eseguire il processo.

Come possibile soluzione, potremmo scegliere il frame di un altro processo

Effetti del thrashing

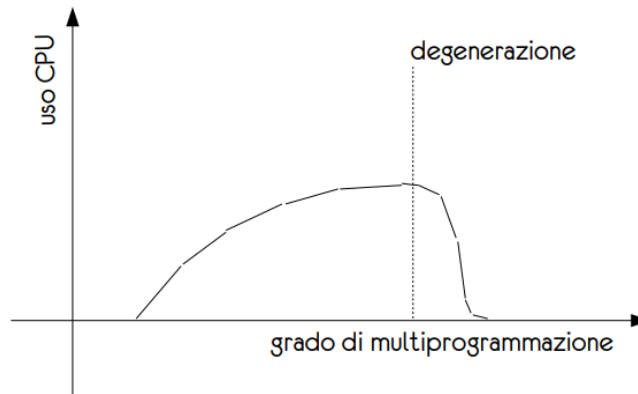
Quando si ha thrashing perché i processi hanno a disposizione meno frame del numero di pagine attive, l'attività della CPU tende a diminuire: i processi tendono a rimanere in attesa del completamento di operazioni di I/O.



Thrashing e multiprogrammazione

Molti SO monitorano l'uso della CPU: quando questo cala, se ci sono processi conservati in mem. secondaria, portano in RAM qualche processo in più.

Ai nuovi processi vengono allocati frame sottratti ad altri processi in RAM se la strategia di allocazione è globale, quindi aumenta la frequenza di page fault, diminuisce l'uso della CPU e si ripete fino al blocco totale.



In conclusione, **si ha thrashing perché la politica di allocazione dei frame è globale**: il SO può sottrarre frame ad un processo per assegnarli ad un altro, rendendo degeneri più processi alla volta.

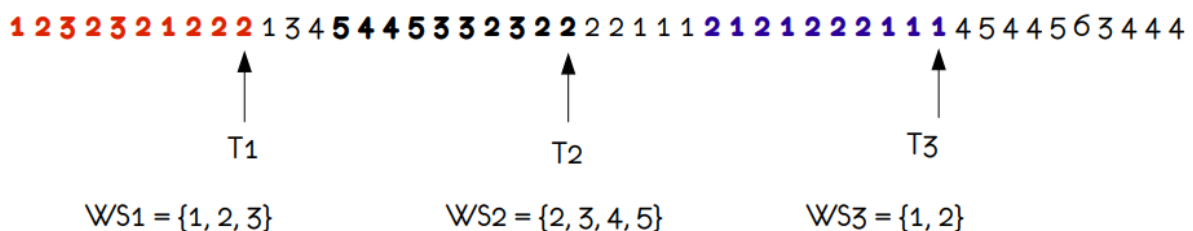
Pagine attive e Working Set

L'ideale è cercare di **prevedere di quante pagine avrà bisogno un processo e assegnargli un numero sufficiente di frame**.

L'allocazione dipenderà dal numero di pagine attive per processo, e si distribuiranno frame liberi solo ai processi che ne hanno bisogno. Poiché il numero di pagine attive varia nel tempo, se cresce si allocano nuovi frame e se decresce si liberano frame.

Questo approccio prende il nome di **Working Set**. Esso si basa sul principio di località: per periodi di durata consistente, il programma avrà accesso solo ad un sottoinsieme del suo codice, variabili globali e locali (quindi un sottoinsieme di pagine, le pagine attive). Il WS è quindi **l'insieme delle pagine attive di un processo** in un certo periodo di tempo.

Ad esempio, per $\delta = 10$:



Definita la finestra δ è possibile calcolare il WS di ogni processo, e quindi calcolare la quantità di frame necessari ai processi in esecuzione:

$$D = \sum_{i=1}^n |WS^i|$$

Quando $D > n^\circ$ frame liberi, si genera thrashing: significa che qualche processo non ha a disposizione frame a sufficienza.

Se $D < n^\circ$ frame liberi, è possibile avviare nuovi processi (possibilmente limitandosi ai frame disponibili).

Quando il WS di un processo cresce, se non ci sono frame liberi, il SO sceglie uno dei processi, lo copia in memoria secondaria, lo sospende e assegna (parte dei) suoi frame al processo richiedente: così si evita il thrashing.

Prepaginazione

Per via del modo in cui sono definiti i working set, ad ogni cambiamento di località si ha un certo numero di page fault.

Ma un problema a cui possiamo porre rimedio, riguarda l'eliminazione dei page fault ad ogni swap in. Se memorizziamo insieme al PCB anche il WS, possiamo caricare in memoria tutte le pagine utili subito. Questa tecnica prende il nome di **prepaginazione**.

Page fault frequency

I working set funzionano molto bene: ma per evitare il fenomeno del thrashing, possiamo usare altre tecniche.

La frequenza dei page fault aumenta in presenza di thrashing. Basta porre un limite alla frequenza massima di page fault accettata: se si supera, si alloca un nuovo frame

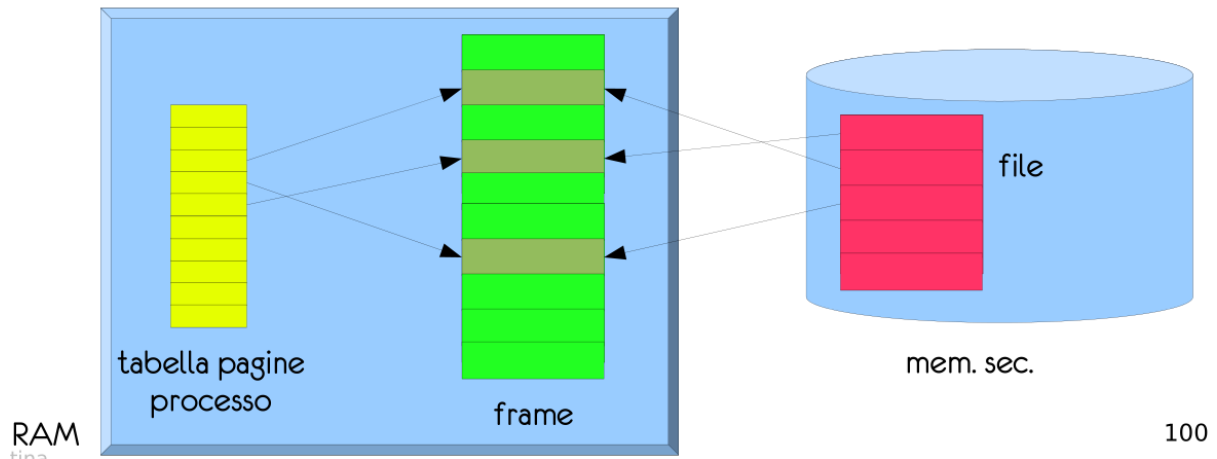
Memoria virtuale e file

Quando i processi agiscono sui file tramite system call `read()` e `write()`, richiedono accessi sequenziali alla memoria secondaria, in quanto il file stesso non fa parte del loro codice.

Eseguire operazioni direttamente sul disco comporta bassa efficienza: sarebbe meglio eseguire le operazioni sulla RAM (memoria più veloce) e riportare ogni tot le modifiche in mem. secondaria. → mappatura dei file

Mappatura dei file

Procedimento nel quale si associano degli indirizzi virtuali di un processo ad una parte di un file a cui ha accesso



Scrivere su RAM renderà le operazioni di I/O più veloci, e se un file è usato da più processi, lo si carica in RAM una singola volta e si condividono le pagine

Da RAM a disco

E' possibile utilizzare diverse strategie per riportare le modifiche dalle copie dei file in RAM ai file effettivi presenti in mem. secondaria:

- **controllo periodico**: se durante il controllo la pagina risulta modificata, la si copia in mem. secondaria;
- **chiusura del file**;
- operazioni di **flush** esplicite nel codice del programma;

Mappatura I/O

Oltre ai file, alcuni SO mappano anche i registri dei controller dei dispositivi di I/O. Così l'interazione con questi device avviene solamente scrivendo/leggendo le porzioni della RAM associate ai device.

Utente

I meccanismi per gestire la memoria sono trasparenti all'utente. Ma lo stile di programmazione può influenzare il numero di page fault.

Per esempio, supponiamo di dover inizializzare una matrice con il seguente codice:

```
for (j=0; j<128; j++) // per ogni colonna
    for (i=0; i<128; i++) // per ogni riga
        mat[i][j] = 0;
```

Il codice è corretto, ma le matrici sono immagazzinate per riga: in questa maniera generiamo più page fault del dovuto.

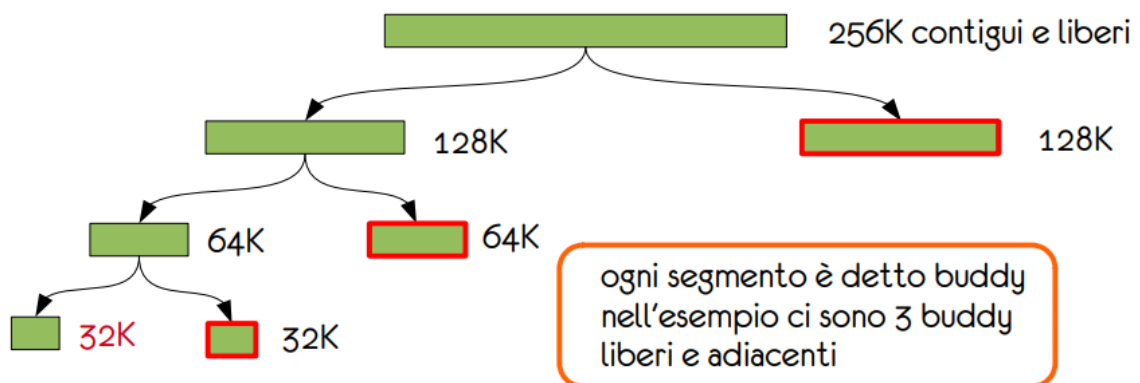
Allocazione dei frame per il kernel

Differente dall'allocazione per i processi utente a causa di caratteristiche differenti dei processi kernel:

- necessità di strutturare dati di dimensioni variabili, spesso molto più piccole di una pagina;
- alcuni dispositivi interagiscono direttamente con la RAM: c'è bisogno di avere aree di memoria contigua;
- **codice e dati del kernel non sono sottoposti a paginazione.**

Sistema buddy

Si utilizza quindi il **sistema buddy**, anche noto come sistema gemellare, che consiste nell'utilizzo di **pagine fisicamente contigue** allocate in memoria in unità di **dimensioni pari a potenze di 2** (4KB, 8KB, 16KB...), e arrotondando poi per eccesso le richieste (es: se servono 6KB, si allocano 8KB).



Lo svantaggio principale di questo sistema è l'alta frammentazione interna: per allocare 33KB uso 64KB, per allocare 65KB ne uso 128 ecc.

Allocazione a slab

Una slab è una sequenza di pagine fisicamente contigue, e una cache un insieme di slab. Con questo sistema viene mantenuta una cache per ogni tipo di strutture dati usate dal SO, per esempio: i semafori, PCB... Ogni cache contiene delle istanze, e queste istanze possono essere libere o occupate.

Quando viene richiesta una nuova istanza, il SO cerca la cache in questione, e se c'è un'istanza libera usa quella. Se non la trova, alloca una nuova slab.

Questa tecnica è molto efficiente, perchè elimina il problema della frammentazione interna. Introdotta in Solaris, ora è usata anche da Linux (in precedenza usava il buddy system).

File system

Programmi e dati sono memorizzati in memoria secondaria, organizzata in file e directory che insieme formano il **file system**.

Il file è un'astrazione: è un **insieme di informazioni correlate** (a discrezione dell'autore) a **cui è associato un nome**. Per l'utente, i file sono gli elementi base in cui è organizzata la memoria, ma non sono uguali tra loro.

Ogni file può essere diviso in due parti:

- contenuto del file;
- i suoi metadati, un insieme di caratteristiche che descrivono il file stesso:
 - nome;
 - (The)tipo(01);
 - dimensione;
 - proprietario;
 - protezione;
 - data/ora di modifica;
 - ecc.

In alcuni sistemi, per associare al file un suo tipo viene utilizzato un codice all'interno del file stesso, chiamato **magic number** (es: CAFEBABE (hex) → java class file, %PDF → file pdf, %! → file postscript ecc.).

Le **estensioni dei file** (.pdf, .java, .xml ecc.) servono per **definire un particolare formato con cui i dati sono organizzati**: i file immagine sono quindi organizzati diversamente dai file di testo, ma anche un file .docx (documento di Word) è organizzato diversamente da un file .html ecc...

Tipi di file

I file possono essere caratterizzati anche da tipologie più generali rispetto alla specifica applicazione che li gestirà, in particolare:

- **file alfanumerici**, contengono una sequenza di caratteri. Es:
 - file di testo;
 - file sorgenti;
- **file binari**, contengono byte organizzati secondo una struttura precisa ben diversa da un file di testo. Es:
 - file oggetto;
 - file eseguibili;

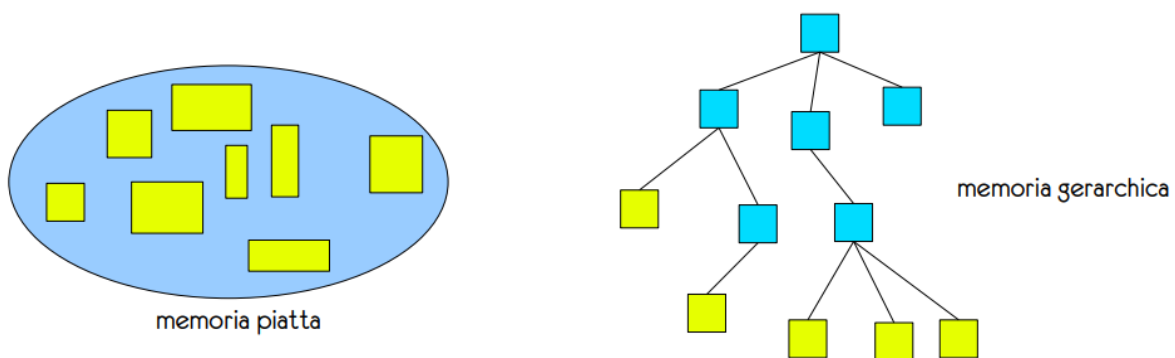
Organizzazione del file system

Il **file system** in sé è la struttura logica secondo la quale sono organizzati i file.

Un file system è mantenuto attraverso l'utilizzo di strutture dati interne, preposte al mantenimento dei metadati (dati inerenti i file e dati inerenti le directory).

Esistono due visioni classiche:

- **organizzazione piatta**: insieme di file tutti allo stesso livello (appunto "piatto"), non possono esserci file con lo stesso nome;
- **organizzazione gerarchica**: memoria organizzata ad albero, i cui nodi interni fungono da contenitori e sono chiamati **directory**.



Apertura e chiusura di un file

Un **file aperto** da un processo è inteso come una **risorsa in uso da parte del processo**.

L'**apertura** di un file comporta l'allocazione di un insieme di risorse di file system che consentono l'accesso al file stesso.

Chiudere un file significa rilasciare le risorse di file system allocate. Se il processo era l'unico utilizzatore del file, le pagine di RAM usate per consentire un'elaborazione efficiente dei dati possono essere liberate.

In genere, sia apertura che chiusura restituiscono/richiedono un handle come argomento, che definisce il file aperto/il file da chiudere. Ad esempio in C, la funzione `fopen` restituisce come handle un `FILE *` (puntatore al file), mentre la `fclose` richiede come parametro il `FILE *` ottenuto con `fopen`.

Lettura e scrittura su file

Anche lettura e scrittura richiedono un handle, ma occorre anche definire il punto del file da cui leggere o scrivere:

- **accesso sequenziale:** non si specifica esplicitamente la posizione, ma il SO mantiene un puntatore alla posizione corrente (mantenuta nella tabella dei file aperti del processo);
 - quando si legge, si fa avanzare un puntatore che indica la posizione raggiunta all'interno del file;
 - quando si scrive, il contenuto viene aggiunto al fondo del file;
- **accesso diretto:** si può leggere/scrivere in punti specifici del file, che vanno indicati espressamente;
 - il file è visto come una sequenza di record di uguale dimensione;
 - conoscendo dimensione e posizione dei record è possibile accedervi direttamente;
- **accesso a indice:** un file indicizzato è costituito da due file:
 - il file dei contenuti, memorizzati in un formato specifico;
 - il file indice, con i riferimenti ai record;

Protezione dei file

Per gestire gli accessi ai file, alcuni SO richiedono di indicare la modalità di apertura di un file: solo **lettura**, **lettura/scrittura**, **esecuzione**. Certi SO consentono di associare ai file dei diritti di accesso che indicano le modalità di apertura del file consentite agli utenti (es. un utente non può sovrascrivere il codice del SO).

Certi SO consentono di associare dei **lock** ai file:

- **lock condiviso** (lettura): consente a n processi di effettuare determinate operazioni sullo stesso file, anche in parallelo;
- **lock esclusivo** (scrittura): solo il processo che detiene il lock può usare il file;

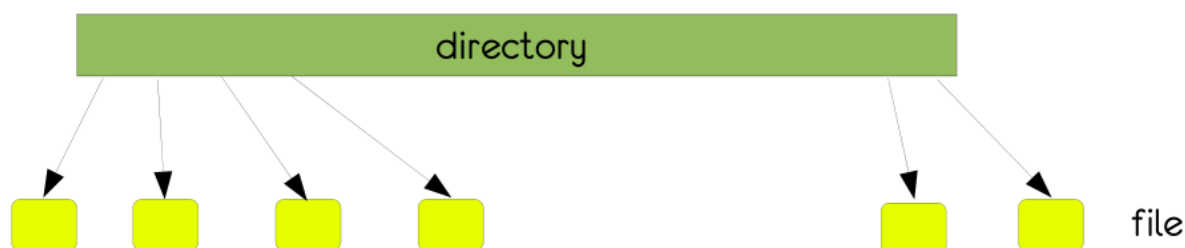
Questi lock possono essere consigliati o obbligatori.

Directory

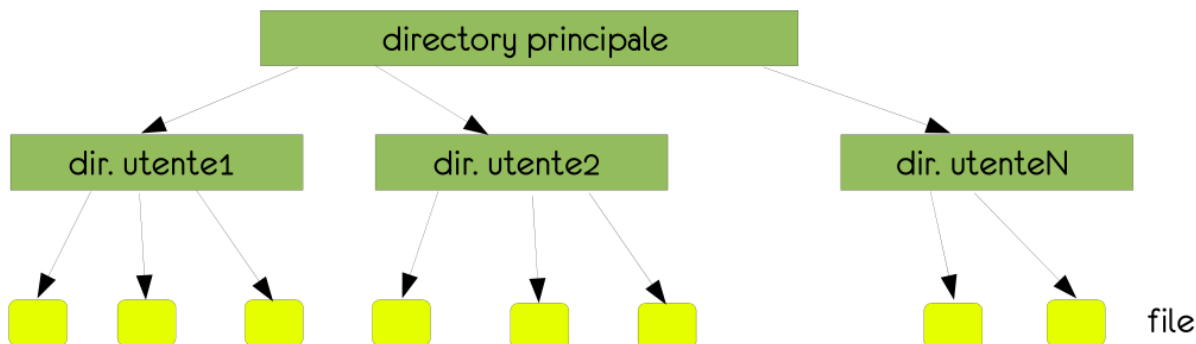
È un entità che può contenere file o altre directory. Prima, molti file system erano piatti: cioè, i file sono tutti sullo stesso livello, senza alcuna organizzazione. Altri consentivano l'uso di un solo livello di directory. Il primo FS gerarchico è nato con Unix.

Dal punto di vista astratto, la directory non è altro che una tabella che consente di accedere ai contenuti di un file a partire dal suo nome. Le operazioni possibili sono le solite: scrittura, lettura, ricerca, attraversamento.

Directory a un livello: contengono i file tutti nella stessa directory. Non si può quindi avere due file con lo stesso nome, la multiutenza diventa difficile.



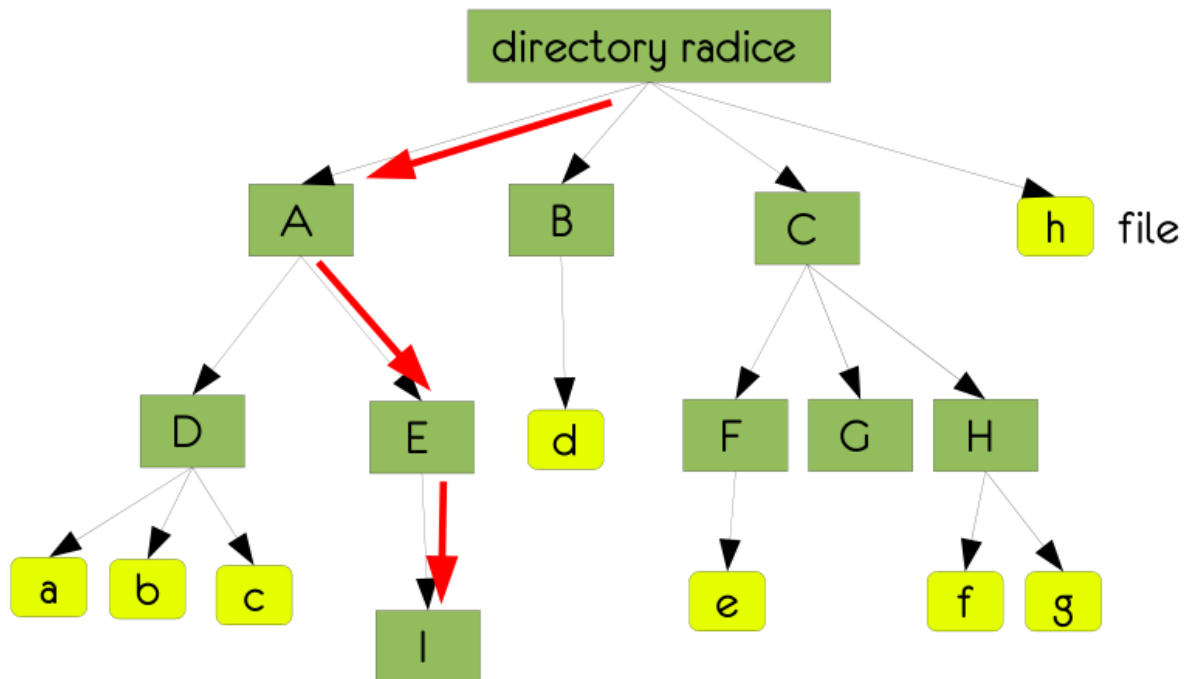
Directory a due livelli: ogni utente ha una propria cartella. Sistema il problema precedente, ma con molti file diventa di difficile uso.



Directory ad albero: i file diventano foglie dell'albero, abbiamo infiniti livelli. Usare cammini assoluti, es. [/home/thetipo01/YADMB/audio_cache/dQw4w9WgXcQ-youtube.dca](#)

diventa scomodo. Per questo l'utente viene posizionato virtualmente in una directory di lavoro.

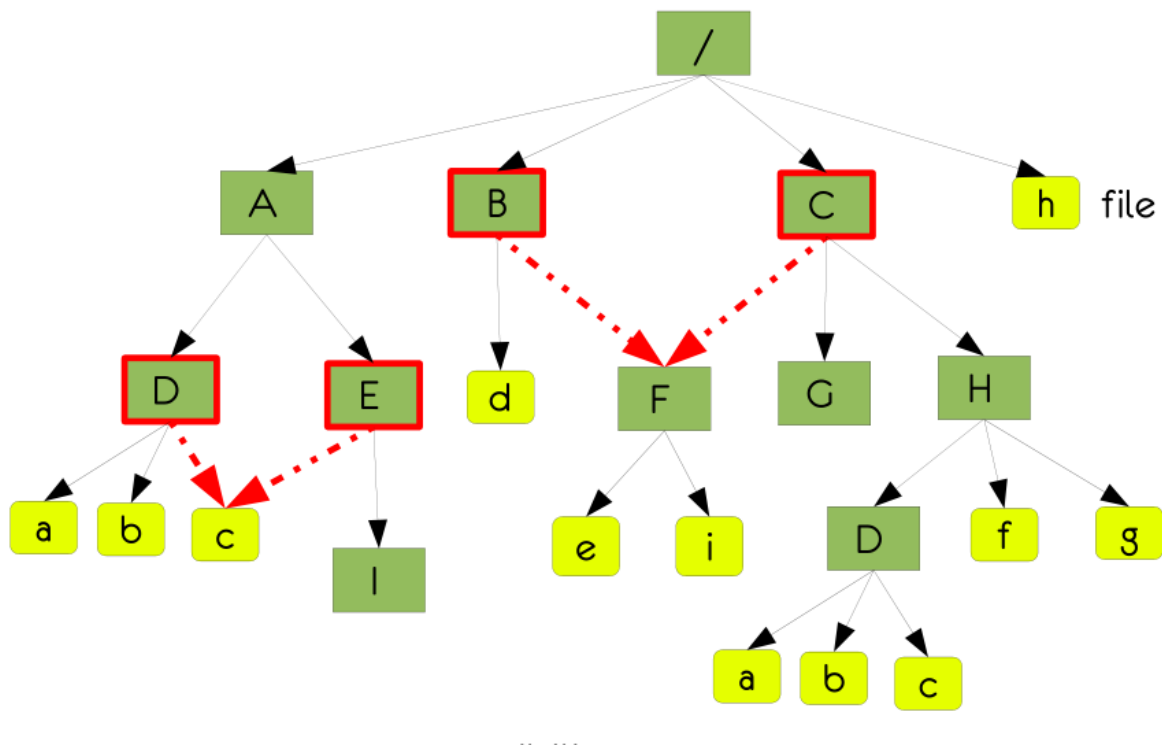
Ogni utente sarà proprietario di un sottoalbero, e un utente sarà proprietario della parte di sistema (amministratore).



Directory con grafo aciclico

Un file/directory può essere accessibile da cammini diversi. Se si vuole eliminare un file che è puntato da due sottoalberi diversi, significa due cose:

1. il file non serve a nessuno dei due utenti: viene rimosso
2. solo uno dei due utenti rinuncia ad usarlo: solo uno dei collegamenti viene rimosso

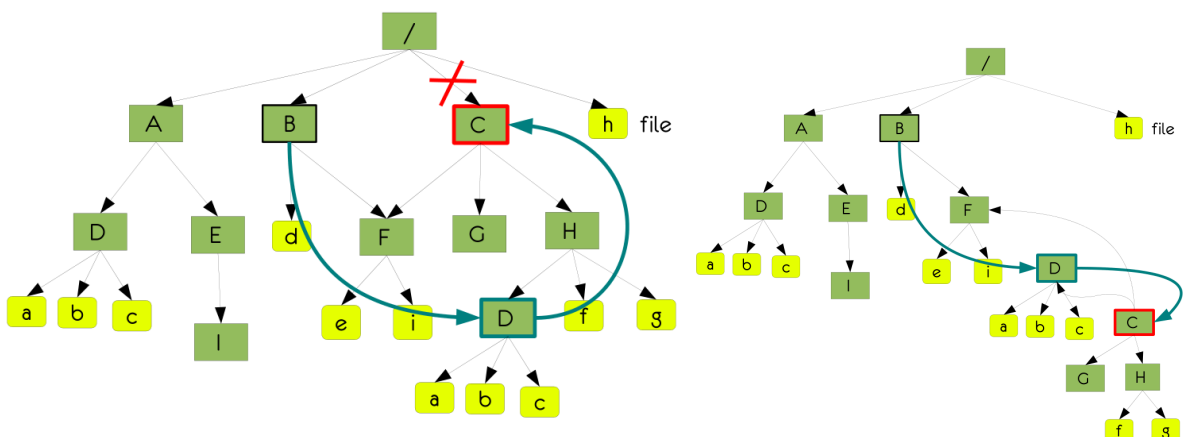


Link

I link sono riferimenti a file/directory. Ci sono due possibili soluzioni:

- Soluzione demandata all'utente: l'utente ha due scelte
 - link simbolico: se il link creato viene eliminato, il file/directory a cui punta rimane intatto
 - link fisico: eliminare questo link vuol dire anche cancellare ciò a cui punta
- Soluzione globale: mantenere il numero di riferimenti a quello che puntiamo
 - creare un link aumenta il valore
 - cancellare un link lo decrementa
 - quando il valore arriva a zero, il file/directory viene rimossa

Directory a grafo generale

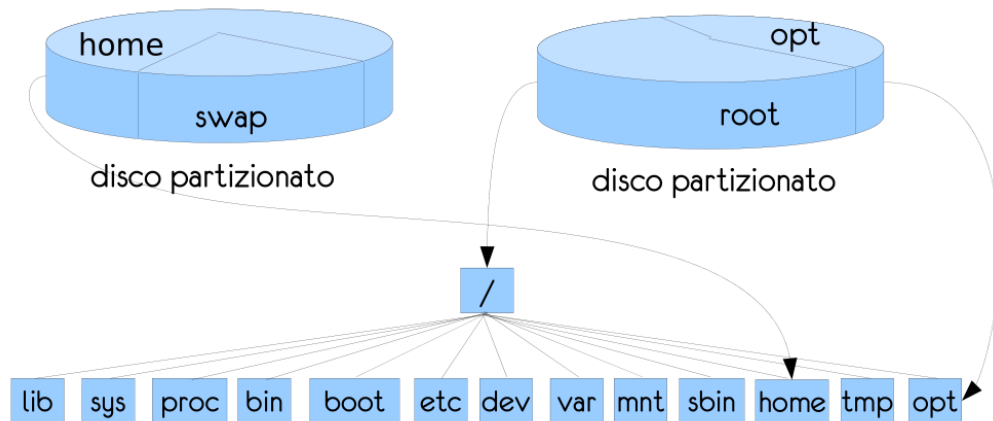


Possiamo avere collegamenti all'indietro.

In questo caso, eliminare il collegamento dalla radice a C, fa cambiare la forma al grafo

Mount

Device possono essere aggiunti/rimossi; possiamo poi partizionare gli hdd, e montare suddette partizioni in punti diversi



Il mount su linux viene fatto nel file `/etc/fstab`.

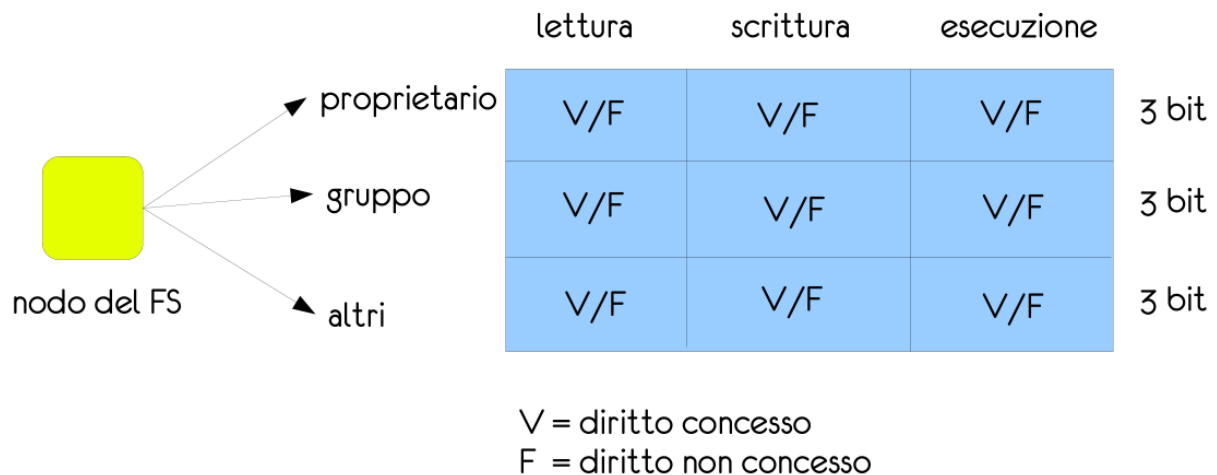
File system distribuiti

I file a cui si accede attraverso un FS possono anche risiedere su macchine diverse, collegate tra di loro in rete. Il modello client-server si basa sul concetto di una macchina chiamata server che contiene fisicamente i file condivisi. Le macchine che vogliono accedere ai file sono dette client.

Questi FS distribuiti si possono montare esattamente come le partizioni locali.

Per proteggere i file da accessi impropri, si possono usare due soluzioni:

1. ACL (Access Control List)
 - a. ad ogni nodo del FS viene associata una lista che specifica gli utenti che possono accedere al FILE
 - b. questa soluzione però complica l'implementazione delle directory, perchè dovrebbe mantenere una ACL, che possono essere lunghe
2. Proprietario, gruppo, altri
 - a. ogni nodo ha un proprietario
 - b. tutti gli utenti sono divisi in gruppi di lavoro



Questo viene poi implementato sfruttando 3 bit per ogni tipo.

In-core inode

Quando un file viene aperto all'inizio, viene arricchito da alcune informazioni aggiuntive. In linux, per esempio, può contenere queste informazioni:

- inode copiato da disco
- Stato dell'in-core inode: dice se
 - l'inode è locked (il file non è disponibile)
 - la copia dell'inode in RAM è diversa da quella su disco,
 - il file è un punto di mount
- device number: identificatore del FS a cui appartiene il file
- inode number: identificatore dell'inode nella struttura dati su disco
- contatore del numero di riferimenti all'inode (numero di utilizzi del file attuali)

Apertura di un file (inode)

Supponiamo di volere aprire un file. Dovremo fare le seguenti operazioni:

- Il kernel mantiene una inode table di dimensione finita
- Nell'eseguire la open: controlla se l'inode corrispondente al file è già stato caricato, se sì userà l'incore inode trovato altrimenti:
- Se c'è spazio, crea un nuovo incore inode, lo "locka" e va a copiarvi l'inode su disco corrispondente al file da usare
- Se non c'è spazio l'operazione fallisce e viene restituito un errore (così il processo richiedente non rischia di rimanere sospeso per tempi lunghi)
- se esisteva già lo vedremo fra poco

Gli inode, però, sono immagazzinati in una sequenza di blocchi. Come facciamo a trovare un file, a partire da un lungo percorso assoluto?

Algoritmo NAMEI

Per identificare l'inode relativo ad un file bisogna usare un algoritmo che è in grado di eseguire i seguenti passaggi:

1. accedo alla directory *mieiFile*
2. cerco fra i suoi contenuti *Documenti*
3. accedo alla directory *Documenti*
4. cerco fra i suoi contenuti *anno2022*
5. accedo alla directory *anno2022*
6. cerco fra i suoi contenuti *slideSisOp.odp*
7. trovo il numero di un inode

Questo algoritmo prende il nome di **algoritmo NAMEI**.

```
/* algoritmo */
inode-number NAMEI(string cammino)

if (la prima directory del cammino è "/")
    current = root inode;
else
    current = inode della working directory;

repeat
    el = leggi prossimo elemento da input;
    if (el == null)
        return current;
    else
        if (el è contenuto in current)
            current = inode associato a el;
        else
            return (no inode);
until (el == null)

return current;
```

Open di un file in Unix

Quindi, per aprire un file in unix, un processo dovrà:

- eseguire `open(pathname, flags)`
- il SO verifica se il file è in uso da qualche processo
- Se no:
 - 1) Utilizza l'algoritmo namei per trovare il numero di inode del file
 - 2) Calcola l'indirizzo su disco che permette di accedere fisicamente all'inode
 - 3) Invia al controller del disco il comando di lettura, che risulterà nella copiatura dell'inode in una entry della tabella in RAM (in-core inode)
 - 4) Aggiorna la tabella di sistema
- Se si:
 - 1) Identifica l'in-core inode e lo rende accessibile al processo modificando la tabella di sistema

0	stdin
1	stdout
2	stderr

Flussi standard che permettono l'interazione del processo con il suo ambiente (default: lettura da tastiera, scrittura su terminale, scrittura su terminale)

La system call open restituisce come handle del file un **file descriptor**, un numero intero. Ogni volta che viene chiamata la open, viene aggiunto il file nella tabella dei file, una tabella globale con i riferimenti a tutti i file aperti

Implementazione delle directory

Punto critico nella progettazione di un file system.

Esistono diversi tipi:

- lista lineare (Unix): una directory è una sequenza di coppie <nomeFile, inodeNumber>;
- B-tree;
- tabella hash;

Sequenza lineare

Limiti:

- per verificare se un file è in una directory, bisogna scorrerla interamente;
- ricerca di tipo lineare è lenta;
- difficile mantenere la lista ordinata senza appesantire la gestione;
- necessarie strutture di appoggio e algoritmi per gestire i buchi creati dalla cancellazione dei file;

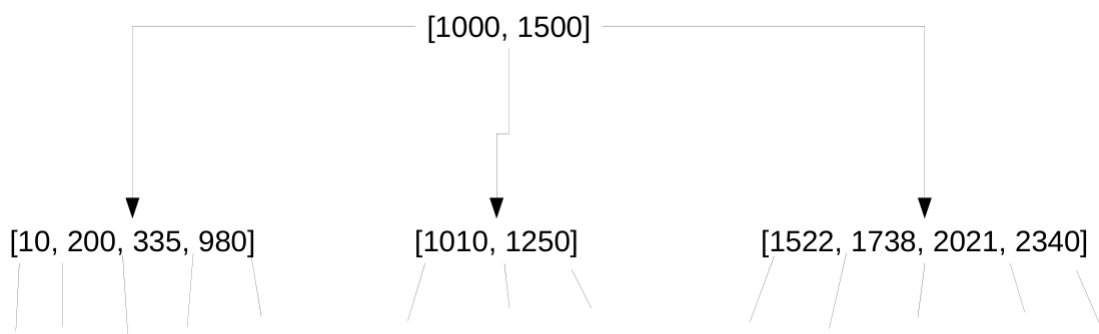
Possibili migliorie

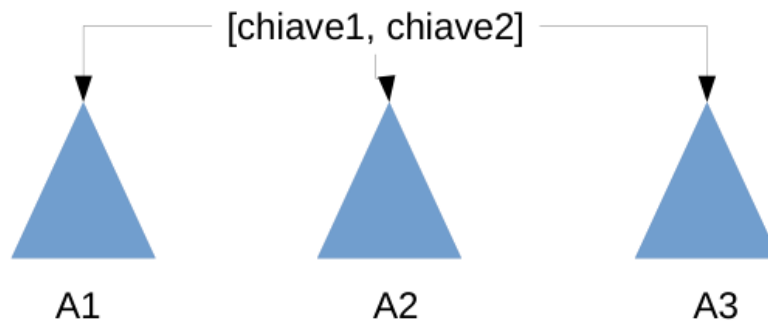
- usare una struttura non lineare (es: ad albero)

B-tree

Un B-tree è un albero ordinato: ogni nodo può contenere da N a $2N$ elementi detti chiavi. Un nodo avrà poi $K+1$ puntatori a nodi del livello successivo.

Il numero N è anche detto ordine dell'albero. Questo, è un albero di ordine 2





- Tutte le chiavi del sottoalbero A1 sono minori di chiave1
- Tutte le chiavi del sottoalbero A2 sono comprese fra chiave1 e chiave2
- Tutte le chiavi di A3 sono maggiori di chiave2
- In un FS le chiavi possono essere identificatori di inode; ad ogni identificatore è associato anche l'inode relativo

Per ricercare in un B-tree si parte sempre dalla radice. Il suo ordinamento consente tempi rapidi di ricerca. Il suo bilanciamento e fan-out (apertura dell'albero) sono caratteristiche sfruttate per velocizzare la ricerca.

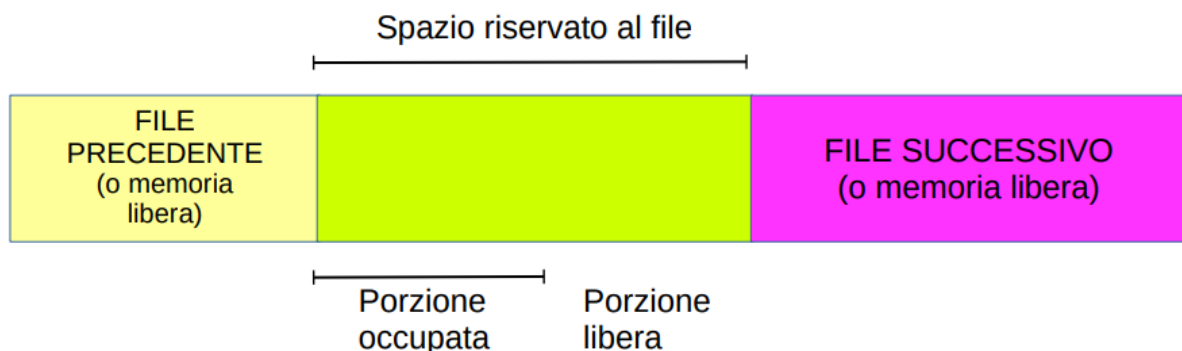
Allocazione dello spazio disco ai file

Ci sono tre metodi per organizzare la memorizzazione dei dati su disco:

- **contigua**;
- **concatenata** (e la variante FAT);
- **indicizzata**;

Allocazione contigua

Ogni file è allocato in una sequenza contigua di blocchi. In genere utilizzata negli HDD a disco magnetico.



Vantaggi:

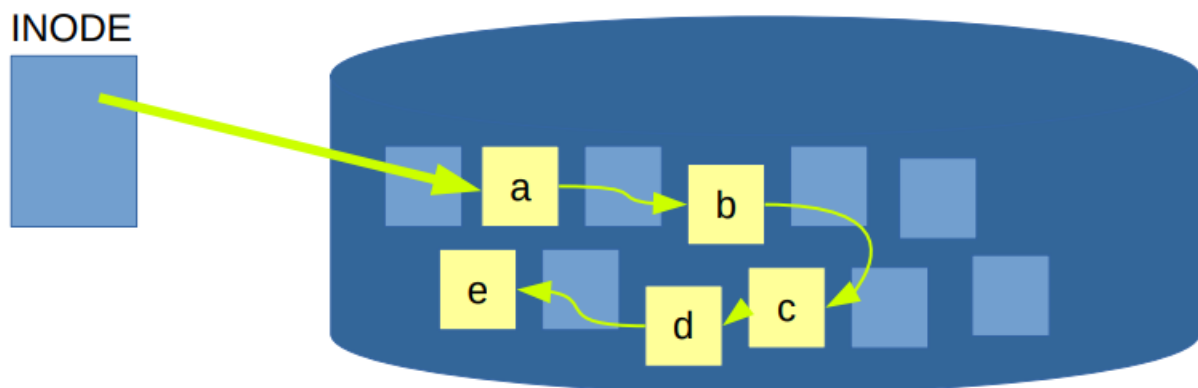
- **rapidità di accesso ai file**: tempo di seek (posizionamento della testina sulla traccia giusta) trascurabile;
- quando si accede a un file **si tiene traccia dell'ultimo blocco letto**, quindi se abbiamo appena letto il blocco B, sia l'accesso sequenziale (al blocco B+1) sia l'accesso diretto (al blocco B+k) sono immediati.

Svantaggi:

- difficile gestire la quantità di memoria da riservare per i file (può essere troppa o troppo poca, il file può crescere);
- frammentazione esterna, c'è bisogno di deframmentazione ogni tanto;
- bisogna gestire i buchi di memoria.

Allocazione concatenata

Consiste nello spezzare il file in parti che possono essere allocate in modo non contiguo (blocchi di dati). Ogni blocco contiene un puntatore a quello successivo.



Vantaggi:

- non è necessario preallocare memoria per i file;
- la lista concatenata è dinamica;
- non è necessario deframmentare;

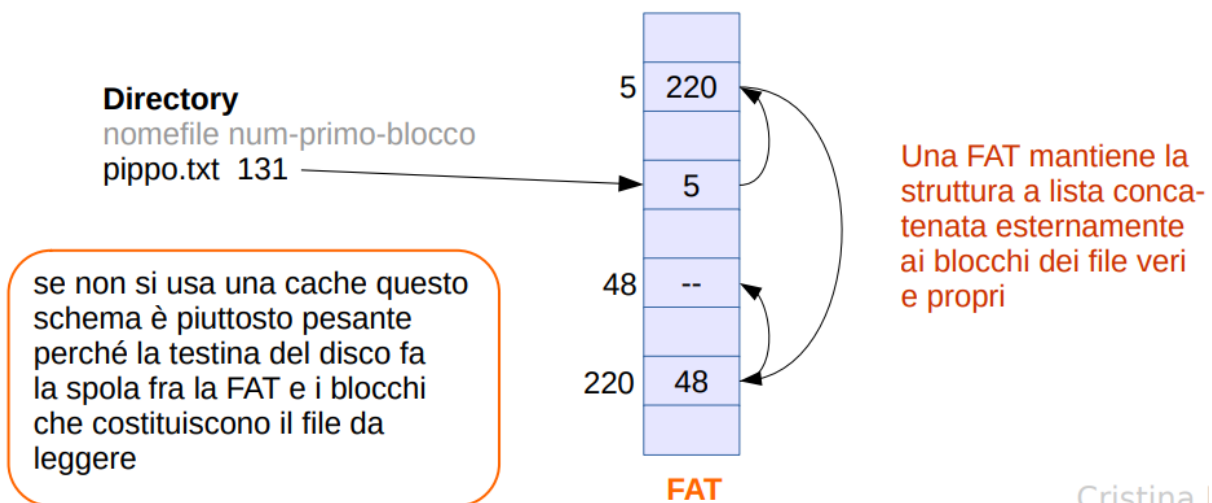
Svantaggi:

- **solo l'accesso sequenziale è efficiente**, accesso diretto e indicizzato devono comunque scorrere la lista dei blocchi;
- i puntatori ai blocchi sono sparsi per il disco, e ogni salto ha latenza (spreco di tempo);
- occorre spazio per mantenere i puntatori ai blocchi successivi;
- **se si corrompe un puntatore, il resto del file va perso.**

Allocazione concatenata FAT

Usata nei sistemi MS-DOS e successori.

Si riserva una sezione della partizione per mantenere una tabella che ha tanti elementi quanti blocchi. Se un blocco fa parte di un file, il contenuto della entry corrispondente in tabella è un riferimento al blocco successivo:



Allocazione indicizzata

Questo tipo di allocazione risolve i problemi delle precedenti soluzioni introducendo un blocco indice. Posseduto da ogni file, il blocco indice è un array degli indirizzi dei blocchi che costituiscono il file. I riferimenti ai blocchi indice dei file sono mantenuti nelle directory.

Per l'accesso:

- tramite la directory recupero il blocco indice
- tramite i riferimenti contenuti nel blocco indice posso accedere ai vari blocchi dati

Commenti: più adeguata ad accessi diretti; l'allocazione indicizzata richiede di mantenere in RAM una parte dei blocchi indice, possono occorrere due o più accessi al disco se la RAM non è sufficiente:

- uno (o più) per accedere al blocco indice giusto
- uno per raggiungere il dato di interesse

Alcuni sistemi combinano allocazione contigua e indicizzata: finché il file rimane di piccole dimensioni si usa l'allocazione contigua, oltre un certo limite si comincia ad usare un indice.

Gestione dello spazio libero

Per tenere traccia dei blocchi liberi, si utilizza una struttura basata su un array di bit, ognuno dei quali corrisponde ad un blocco. Se il blocco è libero, il bit è impostato ad 1.

Es: 0 0 0 1 1 0 1 1 1 1 0 1 0 1 1 1 0 0 0 1 1 0

blocchi liberi = {4, 5, 7, 8, 9, 10, 12, 14, 15, 16, 20, 21}

Diverse tecniche di utilizzo:

- **lista concatenata:** i blocchi liberi sono concatenati in una lista. Poiché i blocchi vengono allocati ai file uno per volta, basta pescare dalla testa della lista il primo blocco libero ed aggiornare il puntatore.
- **raggruppamento:** concatenazione di blocchi indice. Si utilizza un blocco libero per mantenere N-1 puntatori ad altrettanti blocchi liberi, e l'ultimo puntatore per un eventuale ulteriore blocco simile.

- **conteggio:** variante del precedente, in presenza di sequenze di blocchi liberi contigui si mantiene un riferimento al primo di tali blocchi e il numero di blocchi liberi ad esso consecutivi.

Quantità massima di memoria gestibile

Dipende dalle strutture del file system. Se devo gestire un disco di dimensioni superiori alla quantità di mem. massima gestibile, bisogna partizionare il disco in file system diversi.

Es: una FAT a 8 bit consente di indicizzare al più $2^8 = 256$ blocchi, se ogni blocco è grande 1KB potrò gestire al più una memoria pari a 256KB.

Implementazione del file system

Programmi e dati sono conservati in memoria secondaria; la memoria secondaria ha le seguenti caratteristiche:

- è organizzata in blocchi (un blocco può comprendere più settori)
- è possibile accedere direttamente a qualsiasi blocco
- è possibile leggere un blocco, modificarlo e
- riscriverlo esattamente nella stessa posizione di memoria
- si accede alla memoria secondaria attraverso un FILE SYSTEM

Livelli del file system

Il FS è strutturato in una gerarchia di livelli:

- driver di dispositivo: si occupa del trasferimento dei dati da dispositivo di memoria secondaria a RAM e viceversa. È il gestore del dispositivo
- file system di base: è proposto a passare comandi al driver di dispositivo
- modulo di organizzazione dei file: è a conoscenza di come i file sono memorizzati su disco, è in grado di tradurre indirizzi logici in indirizzi fisici. Mantiene e gestisce anche l'informazione relativa ai blocchi liberi
- file system logico: gestisce il FS a livello di metadati. Ogni file è rappresentato da un File Control Block (FCB)

Struttura del disco

Come già detto in precedenza, un disco può essere diviso in più partizioni. Ogni partizione può avere il suo FS, ed i diversi FS sono composti in una sola struttura.

Ogni FS non è altro che una sequenza di blocchi di memoria secondaria. Esistono poi cosiddetti blocchi speciali:

- boot control block: blocco che in presenza di un SO contiene informazioni necessarie per la fase di bootstrap
- volume control block (o superblocco): descrive lo stato del FS, grandezza e altre informazioni (il numero di blocchi liberi, usati e la loro dimensione)

- struttura delle directory: organizza i file, implementata in modo diverso, e contiene le coppie <nome file, FCB>

INODE / FCB

Un **FCB** (o **inode**) viene conservato in memoria secondaria, e mantiene le informazioni relative ai file:

- identità del proprietario (User ID)
- tipo del file (regolare, directory, link, device...)
- diritti di accesso (r w x per ogni tipo di utente)
- tempi di accesso e modifica (data/ora ultimo accesso/modifica, n° di link al file)
- dimensione del file (numero di byte)
- tabella per l'accesso ai dati (indirizzi dei blocchi di mem. secondaria contenenti i dati)

Gli inode hanno **dimensione fissa** per semplificare l'accesso.

✨Glossario✨

- **Interleaving**: nei tempi di attesa di job 1, viene eseguito job 2
- **Parallelismo virtuale**: utenti diversi con programmi diversi vengono convinti dal SO che i loro programmi vengono eseguiti in modo contiguo
- **Time sharing**: distribuzione del tempo di calcolo tra i job in maniera equa, per non creare tempi di attesa eccessivi.
- **File**: dati che hanno un formato consistente agli occhi degli utenti
- **Directory**: insieme di file
- **Evento**: notifica che è avvenuto qualcosa
 - Interrupt: evento hw
 - Trap: evento SW
- **Handler**: gestore di eventi, diversi per ogni tipo
- **Context switching**: salvare le informazioni relative ad un processo in RAM (sospingendolo), per lasciare spazio ad un altro processo
- **RAM**: è una risorsa a cui può accedere direttamente la CPU
- **Risorse**: device/dispositivi
- **Architettura multicore**: più processori, 1 CPU
- **Strutture dati**: rappresentano dell'informazione, e sono delle astrazione che il SW deve manipolare
- **Instruction set CPU**: ci accedono solamente i processi di sistema
 - La dual mode consente ai processi user mode di eseguire istruzioni kernel mode con una system call
 - Controllo dei processi
 - Gestione dei file
 - Gestione dei device
 - Gestione delle informazioni
 - Comunicazioni fra processi
- **Swapping**: salvare la memoria usata da un processo su memoria permanente
- **Albero dei processi**:

- `ps axjf`
- `ps -ejh`
- **Code e memorie condivise:**
 - `ipcs`
 - `ipcs -cu`
- **Thread**
 - `ps m -L | more`

- `top`

