Importanti Sistemi Operativi

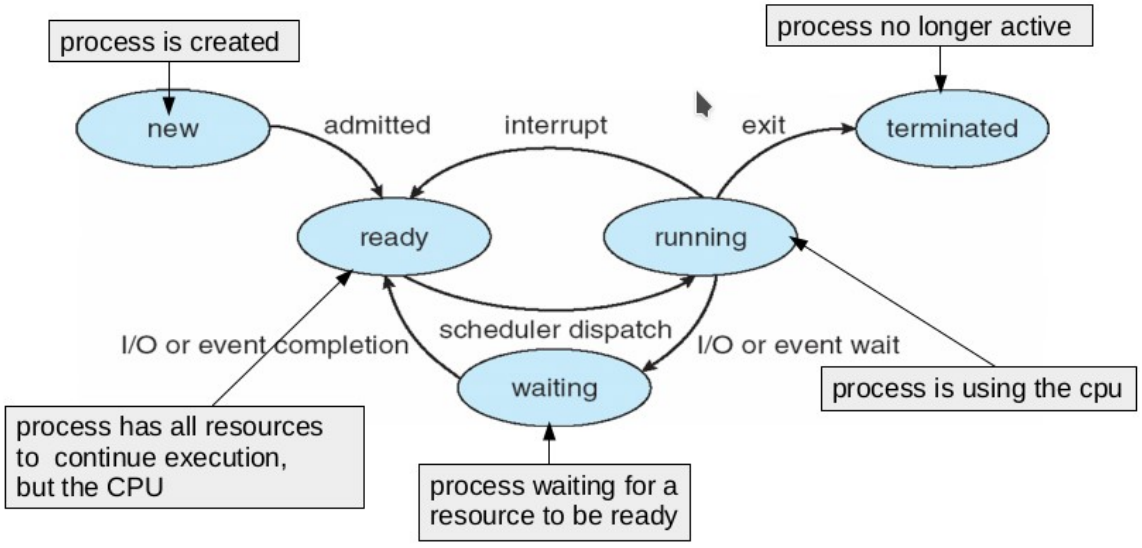
Slab Allocator : usato per allocare oggetti di dimensione fissa, può allocarne fino ad un numero massimo fissato. Il buffer viene quindi diviso in chunk di dimensioni item size. Per poter organizzare il buffer viene quindi usata una struttura ausiliaria che tenga l'indice dei blocchi ancora liberi. Una array list soddisfa tale richiesta.

Buddy Allocator : usato per allocare oggetti di dimensione variabile. Il buffer viene partizionato ricorsivamente in 2 - creando di fatto un albero binario. La foglia più piccola che soddisfa la richiesta di memoria sarà ritornata al processo. Il buddy associato ad una foglia sarà l'altra regione ottenuta dalla divisione del parent. Ovviamente, se un oggetto è più piccolo della minima foglia che lo contiene, il restante spazio verrà sprecato. Quando un blocco viene rilasciato, esso verraà ricompattato con il suo buddy (se libero), risalendo fino al livello più grande non occupato.

Una **Syscall** è una chiamata diretta al sistema operativo da parte di un processo user-level, esempio quando viene fatta una richiesta di IO.
L' **Iterrupt Vector** (**IV**) è un vettore di puntatori a funzioni; queste ultime saranno le **Interrupt Service Routine** (**ISR**) che gestiranno i vari interrupt. Analogamente, la **Syscall Table** (**ST**) conterrà in ogni locazione il puntatore a funzione che gestisce quella determinata syscall. Alla tabella verra associata anche un vettore contenente il numero e l'ordine di parametri che detta syscall richiede.
Per registrare una nuova syscall, essa va registrata nel sistema ed aggiunta al vettore delle syscall del sistema operativo, specificando il numero di argomenti (ed il loro ordine) nell'altro vettore di sistema apposito.

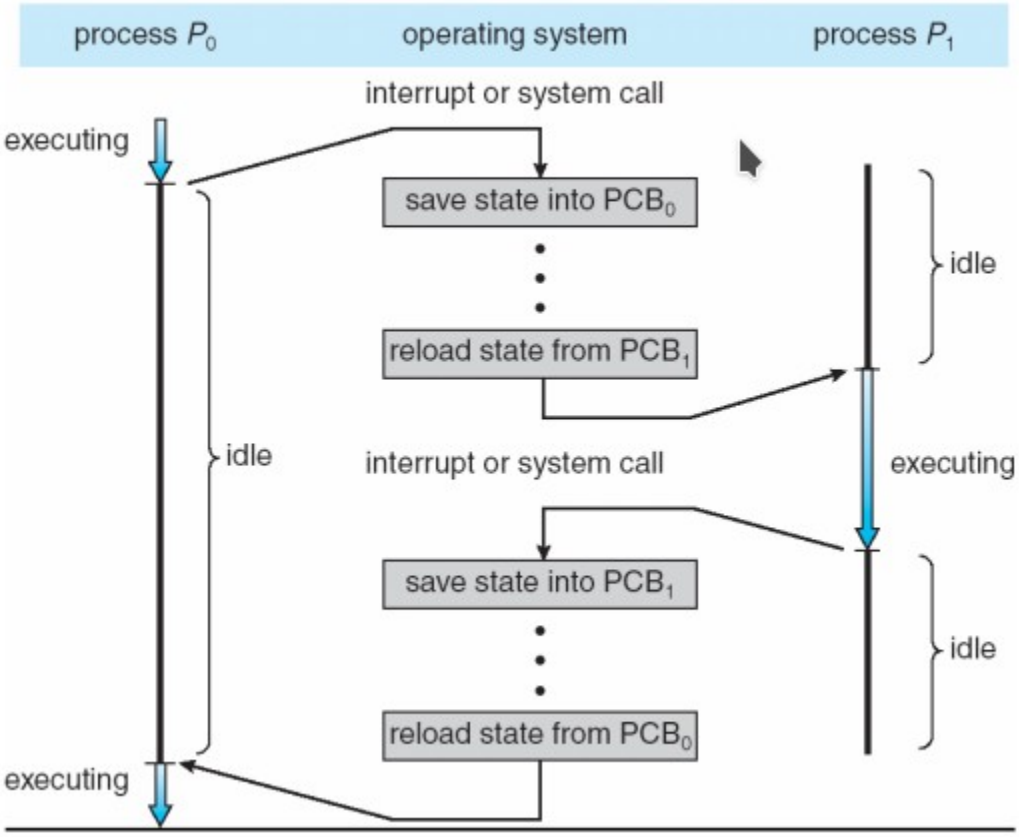
Il **Process Control Block** (**PCB**) di un processo contiene tutte le informazioni relative al processo a cui è associato. Esempi di informazioni contenute nel PCB sono:
– stato del processo (running, waiting, zombie ...)
– Program Counter (PC), ovvero il registro contente la prossima istruzione da eseguire
– registri della CPU
– informazioni sulla memoria allocata al processo
– informazioni sull'I/O relativo al processo.

Fasi di un processo



La syscall **fork** viene usata per creare un nuovo processo figlio a partire da un processo padre. Il processo creato, avrà una copia dell'adress space del padre, consentendo di comunicare facilmente tra loro. I processi, in questo caso, continueranno l'esecuzione concorrentemente. Il figlio, eredita anche i privilegi e gli attributi nel padre, nonchè alcune risorse (quali i file aperti). Una volta creato il processo, se non viene invocata l'istruzione exec() il child sarà una mera copia del parent (con una propria copia dei dati); in caso contrario sarà possibile eseguire un diveso comando.
Il padre aspetta che il figlio completi il suo task tramite l'istruzione **wait**(); quando il figlio finisce la sua esecuzione (o avviene una chiamata exit()), il padre riprende il suo flusso.
Nel caso il padre terminasse prima del figlio, quest'ultimo viene prelevato dal processo **init**.
Quando un figlio termina l'OS invia un segnale al padre (**SIGCHLD**); quando il padre termina, tutti I suoi figli ancora vivi vengono notificati della fine dello stesso attraverso un altro segnale (**SIGHUP**)

Context Switch
Supponendo di avere due processi P0 e P1 e che il primo sia in running. Per mettere in esecuzione P1, l'Operating System (OS) deve salvare lo stato corrente di P0 in modo da poter rirpistinare l'esecuzione dello stesso in un secondo momento. E' bene notare che il context switch è fonte di overhead a causa delle varie operazioni di preambolo e postambolo necessarie allo switch - e.g. salvare lo stato, blocco e riattivazione della pipeline di calcolo, svuotamento e ripopolamento della cache.



CPU Scheduler

Lo Scheduler della CPU viene invocato da una richiesta I/O da parte di un processo.

Scheduler senza prelazione (Non Preemptive): Un processo in esecuzione non viene mai tolto dalla CPU finché non viene ricevuta un richiesta I/O oppure il processo termina.
Scheduler con prelazione (Preemptive): Possibile switchare i vari processi dallo stato ready a running senza che vi sia stata per forza una richiesta I/O o che il processo sia terminato.
Dispatcher: Servono per cambiare da Kernel mode e salvare lo stato del processo corrente per poi ripassare a user mode mentro si va a restaurare lo stato del processo.

Ai fini dello scheduling l'importate è caratterizzare la fine di un processo. Alla CPU interessa solo sapere quanto tempo dovrà lavorare (**CPU burst**), **cioè fin quando non viene chiamata un I/O**, perché poi sarà I/O a dover lavorare (**I/O burst**).

La scelta di quale processo schedulare dipende dal comportamento dell'intero sistema:
• Utilizzo CPU (**massimizzare**): frazione di tempo in cui CPU utilizzata dai processi
• Tempo di consegna (**minimizzare**): time per completare processo
• Throughput (**massimizzare**): numero di processi che si completano in un'unità di tempo
• Tempo di attesa (**minimizzare**): quanto tempo processo deve rimanere nella coda ready
• Tempo risposta (**minimizzare**): quanto tempo deve aspettare processo per essere runnato dopo aver fatto una richiesta I/O

FIRST COME FIRST SERVER (FCFS)

Consiste nel prendere il primo processo che è in coda ready. Quando I/O termina sposto il processo dalla coda waiting alla fine di quella ready.

Example:

Process	Burst Time	Arrival Time
P_1	24	0
P_2	3	1
P_3	3	2

- The Gantt Chart for the schedule is:

- Waiting time for $P_1 = 0$; $P_2 = 23$; $P_3 = 25$
- Average waiting time: (0 + 23 + 25)/3 = 16

Suppose that the processes arrive in the order:

P_2 , P_3 , P_1

- **GANTT**

- Waiting time for $P_1 = 6$; $P_2 = 0$; $P_3 = 3$
- Average waiting time: (6 + 0 + 3)/3 = 3

Much better than previous case

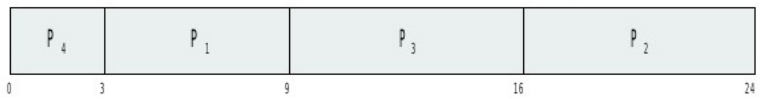
SHORTEST JOB FIRST (SJF)

Ordinare dal processo più breve,cioè ha CPU burst minore, al processo più lungo per avere un buon tempo di attesa e quindi massimizzare throughput. Lo svantaggio del SJF consiste nel fatto che bisogna conoscere il comportamento dei processi in anticipo (nella pratica non è possibile ciò).

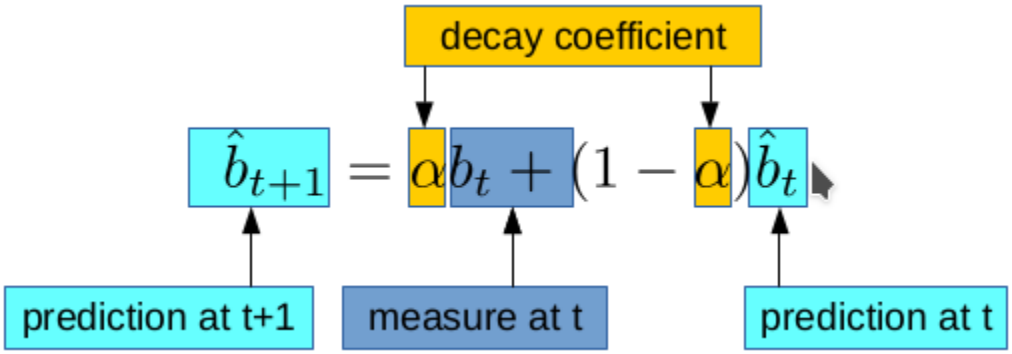
Assume all processes arrive at time 0

Process	Burst Time
P_1	6
P_2	8
P_3	7
P_4	3

- SJF scheduling chart



- Average waiting time = (3 + 16 + 9 + 0) / 4 = 7



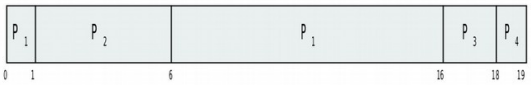
Se **α = 1** allora il burst successivo sarà uguale a quello precedente.
Per predire il burst successivo bisogna utilizzare un filtro passa-basso discreto.

PRIORITY SCHEDULER

Senza prelazione (Non-Preemptive)
Ai processi viene assegnata una priorità (se p1<p2, p1 ha priorità più alta), i processi che vi si trovano nella coda ready e hanno priorità più alta vengono schedulati per primi.
Da notare che SJF è un scheduler di priorità nel momento in cui la priortà è l'inverso del prossimo CPU burst.
L'unico problema che vi potrebbe essere è lo starvation dato che un processo che ha priorià bassa potrebbe non essere mai eseguito. La soluzione consiste nel aumentare la priorità mentre un processo trascorre del tempo nella coda ready.

Process	Burst Time	Priority	Arrival Time
P_1	11	3	0
P_2	5	1	1
P_3	2	4	2
P_4	1	5	3

- Priority scheduling Gantt Chart



- Average waiting time = 8.2 msec

Notiamo come quando arriva P2 che ha priorità maggiore rispetto a P1, P1 viene tolto dallo stato running e messo nella coda ready per far schedulare P2. Quando arriva P3, non h priorità maggiore di P2 quindi viene messo nella coda ready e lo stesso vale per P4.

Con Prelazione (Preemptive)

Ogni processo ottiene una piccola unità di tempo CPU (quanto q della CPU è di solito 10/100 ms).
Se dopo questo tempo il processo sta ancora utilizzando la CPU, è preempted e viene messo nella coda ready.
Il processo tolto viene messo alla fine della coda (Schema RR-Round Robin).
Notiamo che con N processi nella coda ready e con un quanto q, nessun processo aspetterà più di **N(N-1)*q**.

Assuming q=4

Process	Burst Time
P_1	24
P_2	3
P_3	3

The Gantt chart is:

Considerations:

- Typically, higher average turnaround than SJF, but better ***response***
- q should be large compared to context switch time
- q usually 10ms to 100ms, context switch < 10 usec

Notiamo che il tasso di risposta è migliore in quanto è limitato da q rispetto SJF, invece il tasso di cosegna (turnaround) è più alto di SJF.
N.B più piccolo è il quanto q, più context switches vengono fatti. Se vengono fatti molti context switch si ha uno spreco della CPU

REAL TIME CPU SCHEDULING

Soft real-time systems: nessuna garanzia su quando il processo critico in tempo reale sarà programmato. Lo scheduler deve supportare preventivamente, priority-base scheduling.
Hard real-time systems: hai dei task che devono essere eseguiti entro una deadline.
I processi hanno nuove caratteristiche: quelli periodici richiedono CPU a intervalli costanti.
Ha tempo di elaborazione t, deadline d, periodo p; tale che $0 \le t \le d \le p$.
Vale prima di tutto se la macchina ce la fa a soddisfare le richieste in base al fatto che ha dei task in determinati periodi e quindi viene fatta la verifica delle condizioni.

$$U = \sum_j \text{CPU}\,(p_j) = \sum_j \frac{t_j}{p_j}$$

FORMULA DI LITTLE

I processi che escono dalla coda devono essere uguali ai processi in arrivo.
La legge di Little è usata per valutare i vari algoritmi di scheduling, mettendo in relazione la **dimensione media della coda n** con **il tempo di attesa medio W** e **frequenza media di arrivo dei processi nella coda λ.**

$$n = \lambda \cdot W$$

Memoria

L'associazione di un indirizzo delle istruzioni e dei dati agli indirizzi di memoria può avvenire in tre fasi differenti:
→ **Tempo di compilazione**: se la posizione di memoria è nota a priori, può essere generato il codice assoluto; e si dovrà ricompilare il codice se si avvia la modifica della posizione. (assegnare indirizzo fisico nell'eseguibile)
→ **Tempo di caricamento**: deve generare codice rilocabile se la posizione della memoria non è nota al momento della compilazione. (codice si rialloca dato che non conosco dove sta nella memoria)
→ **Tempo di esecuzione**: Assegnazione viene ritardata fino al tempo di esecuzione se il processo può essere spostato durante la sua esecuzione da uno segmento di memoria a un altro. (i riferimenti vengono risolti solo dopo la prima esecuzione)

INDIRIZZO LOGICO E FISICO

Indirizzo logico - generato dalla CPU; indicato anche come indirizzo virtuale.
Indirizzo fisico - indirizzo visto dai chip della RAM (sul bus degli indirizzi).
La **MMU(Memory Management Unit)** è un dispositivo hardware che permette di tradurre gli inidirizzi logici in fisici, quindi fa esattamente la mappatura da indirizzo virtuale (logico) a quello fisico in fase di esecuzione.

Metodi assegnazione:
1) First-Fit: allochi primo.
2) Best-Fit: allochi il più piccolo.
3) Worst-fit: allochi il più grande.

Attualmente per virtualizzare la memoria si va a dividere la memoria fisica in blocchi dimensionati chiamati **frame**, la cui dimensione è una potenza di 2. Invece per quanto riguarda la memoria logica in blocchi della stessa dimensione chiamati **pagine**. Vado quindi ad impostare una tabella per la traduzione di indirizzi logici in fisici.
L'indirizzo generato dall CPU (quindi quello logico) è diviso in : **numero pagina**(page-number),p,usato per indicizzare nella tabella delle pagine che contiene l'indirizzo base di ciascuna pagina nella memoria fisica, e **page offset**,d, combinato con l'indirizzo base per definire l'indirizzo della memoria fisica che viene inviato alla MMU
La tabella delle pagine è mantenuta dalla memoria centrale.
Il **registro di base della tabella di pagine** (**PTBR**) punta alla tabella delle pagine.
Il **registro della lunghezza della pagina** (**PTLR**) indica la dimensione della tabella della pagina.
In questo schema ogni accesso di dati / istruzioni richiede due accessi alla memoria: uno per la tabella di pagina e uno per i dati / istruzioni.
I due problemi di accesso alla memoria possono essere risolti mediante l'uso di una speciale cache hardware di ricerca rapida denominata memoria associativa o buffer di ricerca lato-coda (**TLB**). La protezione della memoria avviene associando il bit di protezione a ciascun frame per indicare se è consentito l'accesso in sola lettura o in lettura-scrittura.

TLB

Memoria associativa alla CPU che memorizza le voci nella tabella delle pagine. Utilizzato per memorizzare nella cache le voci della tabella delle pagine. TLB è di ordini di grandezza più veloci della RAM. Quando la CPU vuole accedere a un indirizzo logico, guarda prima nel TLB per vedere se la pagina corrispondente è presente.
Se presente, utilizza il frame di TLB; altrimenti legge dalla tabella della pagina la voce corretta e sostituisce una vecchia voce nel TLB, alla fine genera il giusto indirizzo fisico.

EAT

Tempo medio ad eseguire un operazione in memoria.

$$EAT = (1+\varepsilon)\,\alpha + (2+\varepsilon)(1-\alpha)$$

oppure

$$EAT = p_{hint}\,(T_{TLB} + T_{RAM}) + p_{fault}\,(2T_{TLB} + 2T_{RAM})$$

SHARED PAGES

Codice condiviso (Shared Code):

→ Una copia del codice di sola lettura (rientranza) condivisa tra I processi (ad esempio, editor di testo, compilatori).
→ Simile a più thread che condividono lo stesso spazio di processo.
→ Utile anche per la comunicazione tra processi se è consentita la condivisione di pagine di lettura-scrittura.

Codice privato e dati (Private code and data):

→ Ogni processo conserva una copia separata del codice e dei dati
→ Le pagine per il codice e i dati privati possono apparire ovunque nello spazio degli indirizzi logici.

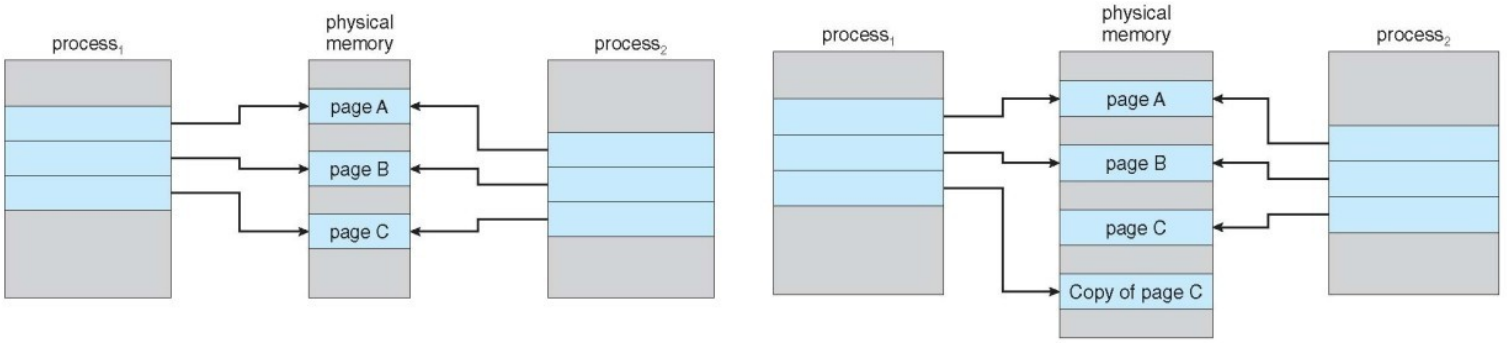
VIRTUAL MEMORY

• Solo una parte del programma deve essere in memoria per l'esecuzione.
• Lo spazio degli indirizzi logici può quindi essere molto più grande dello spazio degli indirizzi fisico.
• Consente di condividere gli spazi degli indirizzi con diversi processi.
• Più programmi in esecuzione contemporaneamente.
• Meno I / O necessari per caricare o scambiare processi.

DEMAND PAGING

Invece di portare l'intero processo nella memoria al momento del caricamento, porto una pagina nella memoria solo quando è necessaria. Se la pagina è necessaria ci deve essere un riferimento ad essa in memoria; se il riferimento non è valido, interruzione; se la pagina non è in memoria viene aggiunta/portata in memoria.
Lazy swapper: non scambia mai una pagina in memoria a meno che la pagina non sia necessaria.
Caso peggiore: accesso a una pagina che non è nella RAM.
Context switch al SO (page fault trap, salva lo stato precedente l'esecuzione dell'istruzione) che controlla che il riferimento alla pagina sia legale e determina la posizione della pagina su disco. I/O emette lettura da disco a un frame libero: attendere richiesta venga servita, attendere tempo di ricerca e/o della latenza del dispositivo, infine trasferimento della pagina su frame; mentre si aspetta, allocare CPU ad un altro utente.
Ricevo interrupt da I/O del disco → Context switch al SO.
Correggere tabella delle pagine per mostrare la pagina che ora è in memoria, ritornare al processo che aveva causato un page fault.

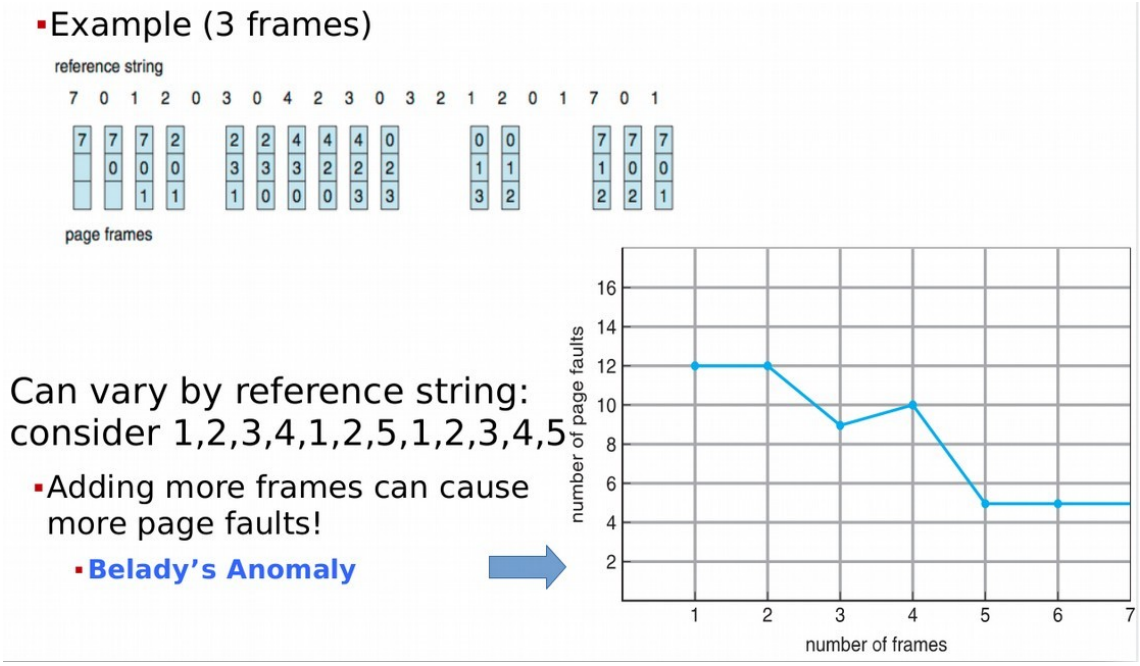
EAT= Effective Access Time = (1 - p) x memory access (RAM) + p (page fault overhead + swap page in)
Se p=0 non avviene page fault, se p=1 ogni "riferimento" è un fault.



A sinistra abbiamo che il processo 1 ha fatto la fork generando il processo 2, il quale copia cosi la tabella delle pagine e andrà settato bit trap=1. IMPORTANTE: I frame non vengono copiati.
A destra invece abbiamo che il processo 1 ha scritto sulla pagina C, generando cosi una trap, in questo modo il frame C viene copiato e il valore della tabella delle pagine del processo 1 viene aggiornata.

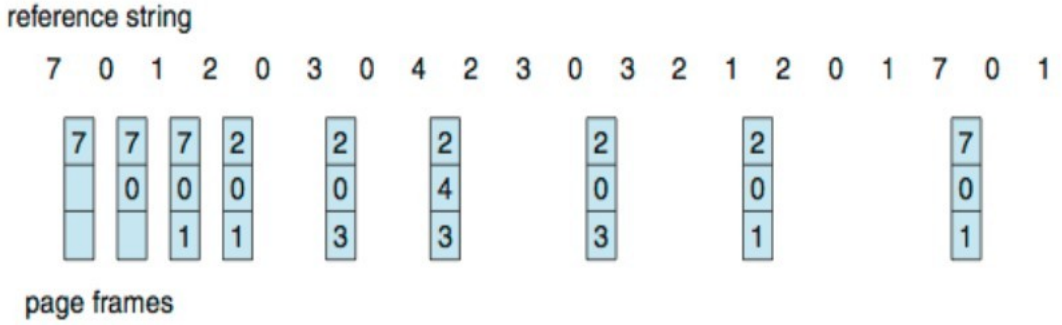
PAGE REPLACEMENT

FIFO ALGORITHM



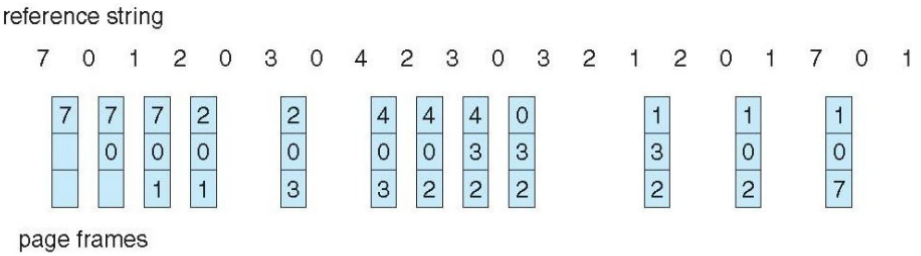
Idea: Scegliere come "vittima" la pagina che è stata scambiata per ultima, cioè quella che tra quelle che ci sono che è stata "modificata" per ultima.

OPTIMAL ALGORITHM



Idea: Sostituire la pagina che non verrà riutilizzata per un periodo di tempo piuttosto lungo. Vedo i numeri alla dx del numero che devo inserire, e sostituisco a quello che risulta essere il più lontano. Optimal value per questo algoritmo è 9 (in questo esempio), numero di page fault fatte.
Fornisce un upper bound: tutti gli algoritmi saranno peggiori di quelli ottimali.

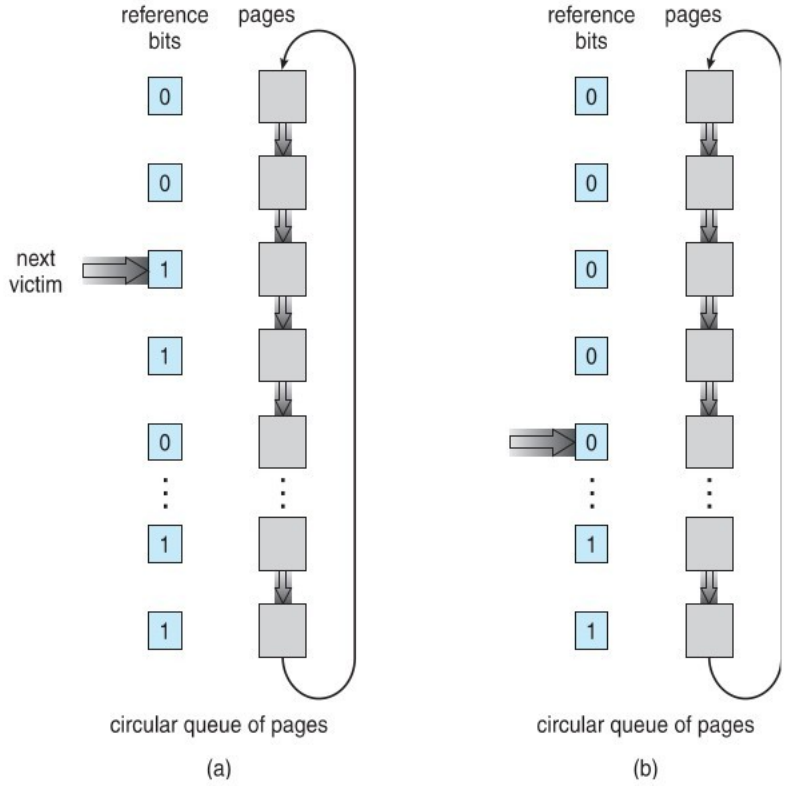
LRU ALGORITHM



12 faults

Idea: approssimativa ottimale prevedendo quale pagina verrà utilizzata per ultima.
Vedo I numeri alla sx del numero che devo inserire e sostituisco a quello “meno recente”.
Usa la conoscenza precedente per ottenere la previsione. LRU non soggetto all'anomalia di Belady.
Limitazioni: LRU richiede hardware speciale, ma è ancora lento.

LRU APPROSSIMAZIONI – SECOND-CHANCE ALGORITHM



Algoritmo della seconda possibilità è un algoritmo FIFO per la sostituzione delle pagine che usa un reference bit (implementato in hardware) per capire quali pagine bisogna rimpiazzare.
In particolare:
Se la pagina da sostituire ha bit di riferimento = 0 -> sostituisco la pagina (vittima)
Se bit di riferimento = 1 imposto il bit di riferimento 0, lascia la pagina in memoria, vado avanti con quella successiva (senso orario).
Un modo per implementare tale algoritmo è attraverso una coda circolare con un puntatore alla pagina da rimpiazzare - che scorre nella coda finché non
trova una pagina con reference bit pari a 0.

SECOND-CHANCE MIGLIORATO

Si considera la coppia ordinata di bit seguente: (bit di riferimento, bit di modifica)

- (0,0): pagina né recentemente usata né recentemente modificata; la migliore candidata alla sostituzione.
- (0,1): pagina non usata recentemente, ma modificata; non è una buona candidata perché prima andrebbe sincronizzata su disco.
- (1,0): pagina usata recentemente ma non modificata; non è una buona candidata perché sarà usata presto, probabilmente.
- (1,1): pagina usata e modificata recentemente; pessima candidata, perché sarà utilizzata nel prossimo futuro e dovrà essere scritta su swap.

FYLE SYSTEM

Mount Point: directory che contiene il file system di un dispositivo.
Link: puntatore a un file nel filesystem, che può essere logico: l'eliminazione del collegamento non ha effetto sul file originale, oppure fisico: quando il numero di collegamenti è 0, il file viene eliminato.
Mount: Tramite operazione di mout il sistema operativo viene informato che un nuovo file system pronto per essere usato. L'operazione, quindi, provvederà adassociarlo con un dato mount-point, ovvero la posizione all'interno della gerarchia del file system del sistema dove il nuovo file system verrà caricato.
Prima di effettuare questa operazione di attach, ovviamente bisognerà controllare la tipologia e l'integrità del file system. Una volta fatto ciò, il nuovo file system sarà a disposizione del sistema (e dell'utente).
Directory: File che contiene "voci" di file e può essere:
• **Hard link**: puntatori a un file. Sono equivalenti l'uno all'altro. cancellando tutti i collegamenti fisici su un file, cancella il file.
• **Soft link**: sono puntatori morbidi, l'eliminazione di tutti i collegamenti simbolici non elimina un file.

FILE DESCRIPTOR

Un File Descriptor consiste in un file handler che viene restituito ad un processo in seguito ad una chiamata alla syscall open(). In seguito a tale chiamata, il sistema scandisce il file system in cerca del file e, una volta trovato, il FCB è copiato nella tabella globale dei file aperti.
Per ogni singolo file aperto, anche se da più processi esiste una sola entry nella tabella globale dei file aperti.

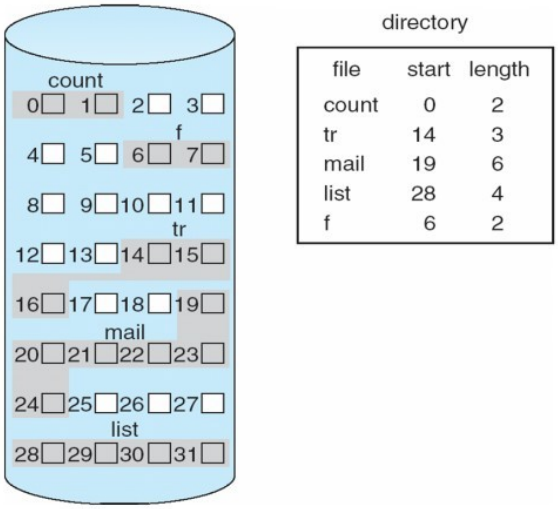
Un **driver** è una "classe virtuale" che dovrebbe sovrascrivere i metodi standard definiti dai dispositivi di archiviazione dell'interfaccia e obbedire alla "interfaccia di blocco". A volte il driver di un dispositivo di archiviazione specifico utilizza altri dispositivi su un sistema.

Ogni file è caratterizzato da una struttura, denominata "**FileControlBlock** (**FCB**)" che contiene tutte le informazioni relative al file a cui è associato. Esempi di informazioni possono essere: permessi, dimensione, data di creazione, numero di inode (se esiste), ecc. .
Inoltre il FCB contiene informazione su la locazione sul disco dei dati del file
- ad esempio in un FS con allocazione concatenata il puntatore al primo blocco del file.
Ogni directory ha un'intestazione (struttura) che estende FCB. Entrambe le directory e i file possono estendersi su più blocchi. Questo viene fatto usando i puntatori ai blocchi.

Con **VFS** (**Virtual File System**) viene indicato tutto il livello di astrazione costruito su un File System concreto.
Il VFS rende possibile ad un client di accedere a diversi tipi di File System in maniera uniforme e consistente.
Un esempio di VFS e' la directory /proc in Linux. I file contenuti in detta directory sono in realtà runtime information del sistema - e.g. memoria di sistema, configurazione hardware - le quali vengono rese accessibili come un file tramite l'astrazione del VFS.

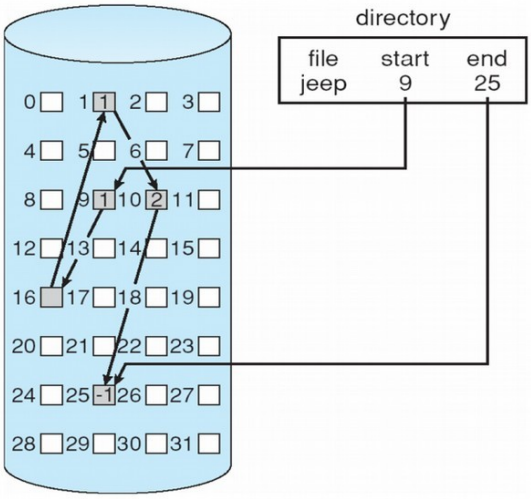
ALLOCAZIONE FILE

CONTIGUOUS



Allocazione contigua: ogni file occupa un insieme di blocchi contigui.
Semplice: sono necessari solo la posizione iniziale (numero del blocco) e la lunghezza (numero di blocchi).
I problemi includono:
1) trovare spazio per il file,
2) frammentazione esterna,
3) necessità di compattazione off-line (inattività) o on-line

LINKED



Linked Allocation, ogni file sarà una lista concatenata di blocchi, dove ogni blocco contiene il puntatore al successivo. Ciò permette quindi di allocare i blocchi in maniera scattered sul disco (non devono essere per forza contigui). Il file finisce quindi con un puntatore a NULL.
Ovviamente, in questo caso, localizzare un blocco può richiedere numerosi cicli di I/O.

FAT

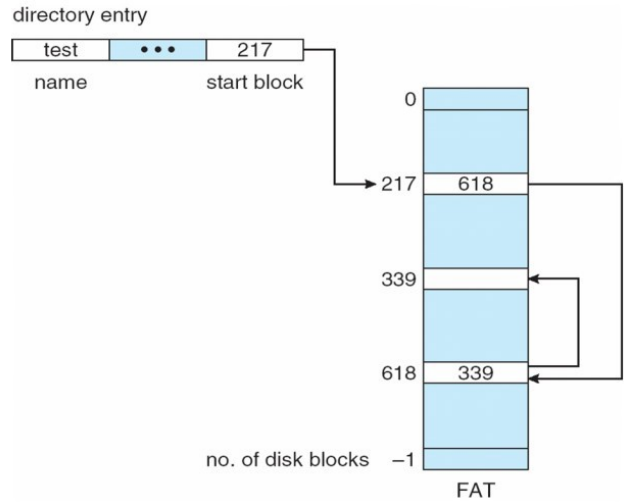
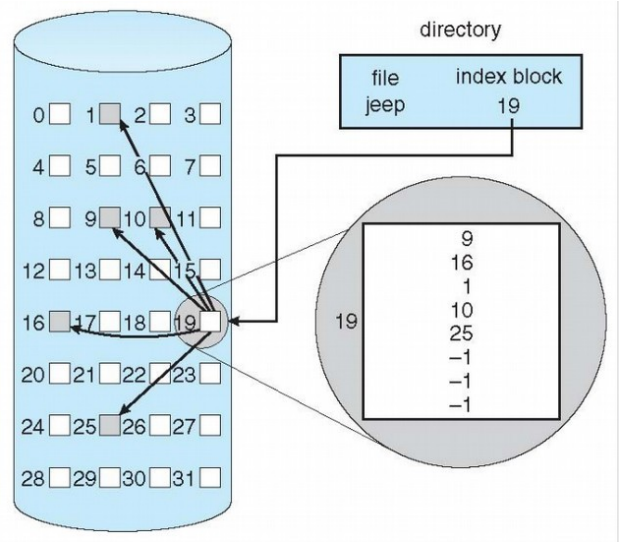


Tabella di allocazione file.
All'inizio del disco, metti un array (FAT)
L'array è un array_list che codifica la sequenza di blocchi.
Lo stesso FAT è relativamente piccolo e rimane nella RAM.
Codifica implicitamente la struttura del blocco (blocchi non validi nell'elenco)

INDEXED



Nel caso di Indexed Allocation i blocchi di un file saranno posti in maniera scattered sul disco (e non contigua), analogamente a quanto avviene nella allocazione a lista (Linked Allocation).
In questo caso però ogni file conterrà un index block, ovvero un blocco contenente i puntatori a tutti gli altri blocchi componenti il file.
Quando un file viene creato, tutti i puntatori dell'index block sono settati a null; quando un nuovo blocco viene richiesto e scritto, il puntatore a tale blocco entrerà nell'index block.
Tale tipo di allocazione permette di guardagnare velocità rispetto ad una implementazione tramite linked list nel caso in cui si effettuino molti accessi scattered ai blocchi (non bisogna scorrersi tutta la lista, ma basta fare una ricerca).
Il costo da pagare è lo spazio necessario a contenere l'index block stesso.

INTERRUPT

Un'interruzione è un segnale inviato da un controller alla CPU per informare il sistema del verificarsi di un evento. Quando si verifica un interrupt, il contesto corrente viene salvato e la CPU avvia l'esecuzione della routine di identificazione degli interrupt.

INTERRUPT VECTOR

Interrupt vector è un vettore di puntatore a funzioni che contiene gli indirizzi dell' ISR (Interrupt Service Rountine, le quali gestiranno i vari interrupt). Ogni posizione del vettore è associato un particolare evento.
Per esempio interrupt id =0 (reset) verrà associato a ADR 0 del puntatore ISR.

ECCEZIONI

Eccezioni sono interruzioni attivate dal Software
3 categorie:

- **TRAP**: ISR viene chiamata dopo attivazione di un'interruzione
- **FAULT** : ISR chiamato prima attivazione dell'interruzione
- **ABORT**: Stato del processo che ha subito interrupt non può essere recuperato

INT <XX>
→ passa a ISR il cui indirizzo è memorizzato nella posizione <XX> del vettore di interrupt, <XX> è l'indice del vettore ISR; inoltre, vi è un numero limitato di punti di ingresso controllati per l'istruzione INT e nel momento in cui si passa all'ISR vengono modificati i bit di flag della CPU.
CALL <YY>
→ chiamo una "funzione" che si trova all'indirizzo <YY>. <YY> può essere qualsiasi indirizzo valido mappato sulla memoria eseguibile di un processo. La diffenza con INT sta anche nel fatto che i bit di flag della CPU non vengono modificati.

SYSCALL

Una syscall è una chiamata diretta al sistema operativo da parte di un processo user-level (modalità utente), esempio quando invoco la funzione printf.

Nel caso si volesse aggiungere una nuova syscall, essa, una volta definita, va registrata nel sistema e aggiunta al vettore delle syscall del sistema operativo, specificandone il numero di argomenti.
Come ripristinare il bit di modalità dopo aver eseguito un syscall?
Viene fatto da IRET (Return from interrupt), istruzione che ripristina i flag.
La tabella delle syscall conterrà in ogni locazione il puntatore a funzione che gestisce quella determinata syscall.
Alla tabella verrà associato anche un vettore contenente il numero e l'ordine di parametri che detta syscall richiede.

La syscall **ioctl** permette di interagire con il driver di un device generico - e.g. una webcam.
Tramite essa sarà possibile ricavare e settare i parametri di tale device - e.g. ricavare la risoluzione della webcam o settarne la tipologia di acquisizione dati.
Per configurare devices seriali a caratteri - e.g. terminali - è possibile usare le API racchiuse nell'interfaccia **termios**.
Tramite di essa, avremo accesso a tutte le informazioni relative al device – e.g. baudrate, echo, ...

IPC

I processi spesso hanno bisogno di comunicare con gli altri processi, quindi condividono informazioni.
IPC di base possiede memoria condivisa e passaggio di messaggi.
Passaggio di messaggi (**message passing**): A differenza della memoria condivisa, è necessario ogni volta l'intervento dell'SO, quindi aumenta overhead. I riscontri positivi consistono nel fatto che l'implementazione è più semplice. Ci sono 2 operazioni **send**(destinatario/mailbox, messaggio) e **receive**(mittentemailbox, messaggio).
La comunicazione può essere invece può essere sia sincrona che asincrona, diretta o indiretta, limitata o illimitata.
Una coda di messaggi è un oggetto gestito dall'SO, che implementa una mailbox. I processi devono però conoscere identificatore della coda per potervi operare.

Memoria condivisa (**Shared Memory**): è un meccanismo del sistema che permette di condividere un'area di memoria tra più processi. Analogamente ai semafori, più shared memory possono essere presenti nel sistema.
Oltre alle operazioni offerte pure dai messaggi, quali creazione e gestione, devono essere previste le operazioni che permettono di mappare l'aria di memoria condivisa nello spazio del processo (attach e detach).

ALTRE COSE PRESENTI NEI VECCHI ESAMI

La legge di **Amdahl** permette di valutare il guadagno di performance derivante dal rendere disponibili più core computazionali ad una applicazione che ha componenti sia seriali che parallele.
Indicando con **S** la **porzione seriale** dell'applicazione e con **N** il **numero di core a disposizione**, si avrà quindi la seguente relazione:

$$\text{GAIN} \leq \frac{1}{S + \frac{1-S}{N}}$$

Differenza tra segmenti e pagine

Gli indirizzi virtuali sono resi necessari per indicizzare un address space più grande del numero di registri disponibili.
La memoria quindi viene organizzata in segmenti o pagine o una combinazione di entrambe. Di conseguenza, i primi bit dall'indirizzo virtuale faranno riferimento ad uno di essi, mentre la restante parte indicherà lo spiazzamento all'interno del segmento o pagina selezionata.
I segmenti sono stati il primo approccio a tale problema. Ogni segmento ha dimensione variabile e perciò è identificato da indirizzo di base e limite. Tali informazioni sono contenute in una tabella apposita (**Segment Table**).
Ovviamente il sistema può proteggere alcuni segmenti non rendendoli accessibili tramite un bit di controllo. I contro della segmentazione sono principalmente frammentazione esterna e la necessità di essere ricompattati.
Per far fronte alle pecche della segmentazione, architetture moderne usano la paginazione per virtualizzare l'address space. La memoria viene quindi divisa in frames (pagine) di dimensione uguale (prestabilita).

La parte alta dell'indirizzo virtuale verrà usata come key per la tabella delle pagine (Page Table), andando ad evidenziare la pagina corrente. La rimanente parte dell'indirizzo verra' usata come offset all'interno della pagina. La paginazione permette anche di proteggere zone di memoria (tramite bit di protezione) e può essere implementata anche in maniera gerarchica (con n livelli di indirezione).
E' bene notare che entrambi gli approcci necessitano di strutture hardware dedicate per funzionare efficientemente.

- Si consideri l'implementazione di un file system che gestisce i files mediante lista concatenata di blocchi (vedi progetto), ed un file system che invece utilizza un albero di indici memorizzati negli inode.
Domanda
Illustrare brevemente i vantaggi dell'uno e dell'altro nell'eseguire le seguenti operazioni:
• accesso sequenziale
• accesso indicizzato
• operazioni su file di testo

SOLUZIONE
1. Accesso Sequenziale: in questo caso, il file system che usa la lista concatenata sarà favorito, garantendo una maggiore velocità dell'operazione. Ciò poiché non bisogna effettuare alcuna ricerca per trovare il blocco successivo, poiché esso sarà semplicemente il blocco next nella lista.
2. Accesso Indicizzato: questa operazione - contrariamente alla precedente - risulta essere molto onerosa per il file system che usa la linked list. Infatti, per ogni accesso, bisognerà scorrere tutta la lista finchè non viene trovato il blocco desiderato. La ricerca tramite inode risulterà molto più efficiente.
3. Accesso su file di testo: per la natura del tipo di file (testo), la linked list risulterà più efficiente ancora una volta. Questo poiché i file di testo sono memorizzati in maniera sequenziale sul disco, riportandoci al caso 1.

fopen(...) una funzione di alto livello che ritorna una stream, mentre **open**(...) una syscall di basso livello che ritorna un file descriptor. fopen(...), infatti, nasconde una chiamata alla syscall open(...).

- Cosa significa Copy On Write (COW) nella gestione della memoria? Illustrare con un semplice esempio il suo funzionamento
Creando un nuovo processo figlio a partire da un padre implica una copia della memoria usata da quest'ultimo.
E' bene notare che molto probabilmente il figlio chiamera' una exec(), perciò tale copia può risultare inutile. A tal porposito è possibile usare la tecnica di Copy-On-Write. In questo caso, inizialmente entrambi i processi - padre e figlio - condividono le stesse pagine di memoria – marcate come copy-on-write pages - e quando uno dei due vuole modificare una delle pagine questa verrà copiata. Il processo è illustrato visivamente nella Figura 2.

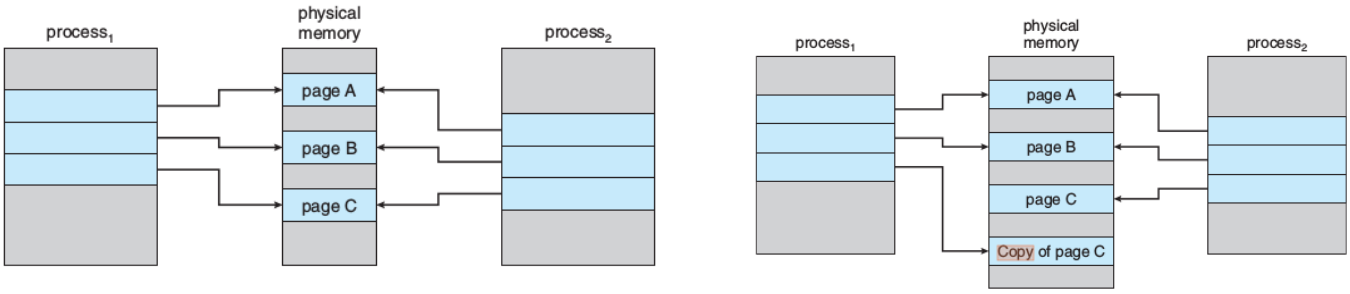


Figure 2: Illustrazione del funzionamento del processo Copy-On-Write: i due processi condividono le stesse pagine di memoria finchè una delle due non necessita di modificarne qualcuna. In quel caso, la pagina interessata verrà copiata.

La directory **proc** è un virtual filesystem, poichè non contiene file reali, bensì runtime system information - e.g. memoria di sistema, configurazione hardware, etc. Molte delle utilities di sistema, infatti, si riferiscono ai file contenuti in questa cartella - e.g. lsmod ≡ cat /proc/modules.

La **vfork**() è progettata come variante più efficiente della **fork**(), specializzata nel caso l'istruzione immediatamente successiva alla fork(), nel processo figlio sia una exec(). A differenza della fork(), la vfork() non replica la memoria del processo padre. Se la prima istruzione eseguita è una **exec**(), l'operazione di copia non è necessaria in quanto l'immagine del processo figlio verrà sovrascritta dalla exec(). Non chiamare la exec() dopo una vfork() genera comportamenti non specificati, in quanto la memoria del padre non viene replicata.