Sistemi Operativi

CPU Scheduler

Lo Scheduler della CPU viene invocato da una richiesta I/O da parte di un processo, quindi la CPU è inutile per esso che viene messo in uno stato Waiting, in modo tale da far schedulare un altro 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 vado 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)

Idea 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)

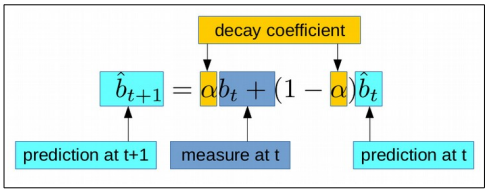
Ordinare dal processo più corto,cioè ha CPU burst più corto, al processo più lungo per avere un buon tempo di attesa e quindi massimizzare throughput. Lo svantaggio di SJF consiste nel fatto che bisogna conoscere il comportamento dei processi in anticipo (nella pratica non è possibile ciò).

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.
Predire burst successivo bisogna utilizzare un filtro passa-basso discreto.

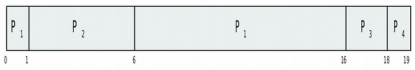
PRIORITY SCHEDULER

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 stesso vale per P4.

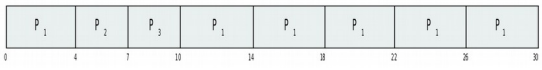
Con Prelazione (Preemptive)

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 il CPU, è preempted 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:



Notiamo che 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 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 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.

$$U = \sum_j \text{CPU}(p_j) = \sum_j \frac{t_j}{p_j} < 1$$

QUEUING MODELS

Descrive l'arrivo dei processi, e la CPU e I/O burst in maniera probabilistica.

FORMULA DI LITTLE

In uno stazionario, i processi che escono dalla coda devono uguali ai processi in arrivo.

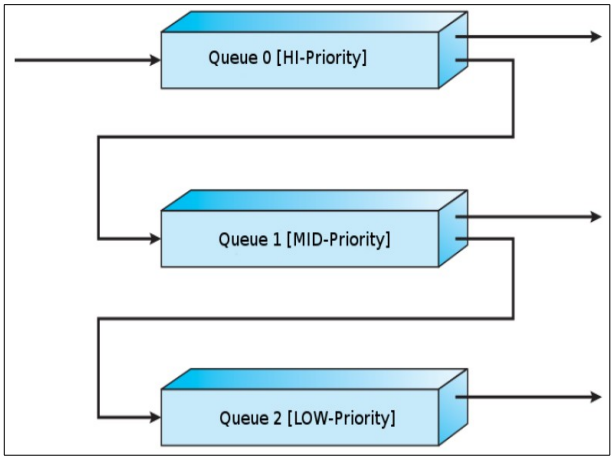
$n = \lambda \times W$

- Valid for any scheduling algorithm and arrival distribution
- n = average queue length
- W = average waiting time in queue
- λ = average arrival rate into queue

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.

$n = \lambda \cdot W$

MULTILEVEL-FEEDBACK QUEUE SCHEDULER



Per ogni coda dovrà essere definito un algoritmo di scheduling, tenendo conto che le code hanno priorità diverse.

Una possibile combinazione di algoritmi di scheduling per le 3 code è la seguente:
• queue 0: Round-Robin (RR) con un time quantum di 8
• queue 1: Round-Robin (RR) con un time quantum di 16
• queue 2: First Come - First Served (FCFS)

Inoltre, bisogna schedulare anche le code. Una implentazione sensata potrebbe usare uno scheduling basato su time-slice: in questo caso, ogni coda ha un certo periodo di tempo - proporzionale alla priorità della coda - in cui dispone della CPU - es. 50% queue 0, 35% queue 1 e 15% queue 2.

Un processo pu essere spostato in una coda a priorità più bassa se il suo CPU burst è maggiore del time quantum della sua coda (per non bloccare code ad elevata priorità). Contrariamente, può essere promosso ad una coda superiore se non è servito da troppo tempo nella coda attuale.
In questo modo, dovremmo poter evitare i fenomeni di starvation ed aging.

Le strutture dati fondamentali sono:
– Linked list per le code di processi
– Counter per ogni coda, che tenga conto del tempo in cui la CPU è impegnata dalla coda
– Variabile nel PCB che tenga conto del numero di quanti di CPU usati dal relativo processo

MAIN MEMORY

La memoria centrale e i registri sono l'unico storage a cui la CPU può accedere direttamente.
I file eseguibi sono sul disco binario file contenenti tutta informazione richiesta per :caricare un programma immagine in memoria: codice programma,variabili inizializzate, etc.

Programmi su disco, sono pronti per essere portati in memoria per l'esecuzione da una coda di input.
Senza supporto HW, deve essere caricato nell'indirizzo 0000, scomodo avere il primo indirizzo fisico del processo dell'utente sempre a 0000. Inoltre,gli indirizzi vengono rappresentati in modi diversi in diverse fasi della vita del programma:
1) Indirizzi di codice sorgente sono solitamente simbolici.
2) Indirizzi di codice compilati eseguono il binding a indirizzi rilocabili, cioè "14 byte dall'inizio di questo modulo".
3) Il linker o il loader collegheranno indirizzi rilocabili a indirizzi assoluti, cioè 74014.
4) Ogni associazione associa uno spazio di indirizzo a un altro.

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)

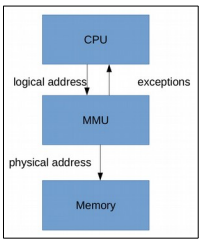
ALLOCAZIONE

Separare la memoria per dare ad ogni processo ciò di cui ha bisogno.
Vi possono essere numerosi problemi, primi tra tutto il fatto che la necessità di "memoria" può variare durante la vita del processo, per esempio che la dimensione dell'heap può variare. Oppure si può verificare il fenomeno della frammentazione, cioè potrebbe esserci abbastanza memoria ma nessun "buco" che soddisfi lo spazio richiesto da quel processo. Per ultimo, protezione,consiste nell'impedire a un processo utente di leggere/scrivere su memoria che non gli appartiene, oppure eseguire codice in aree non eseguibili.

INDIRIZZO LOGICO E FISICO

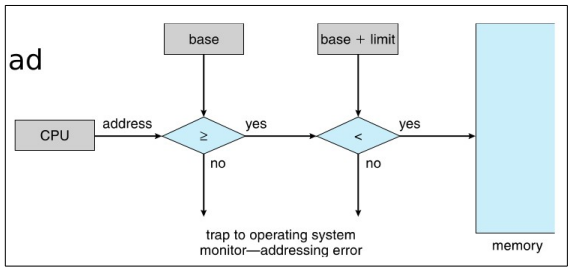
Indirizzo logico - generato dalla CPU; indicato anche come indirizzo virtuale.
Indirizzo fisico - indirizzo visto dai chip della RAM (sul bus degli indirizzi).

La MMU(Memory Management Unit) è dispositivo hardware che permette di tradurre gli inidirizzi logici in fidici, quindi fa esattamente la mappatura da indirizzo virtuale (logico) a quello fisico in fase di esecuzione.

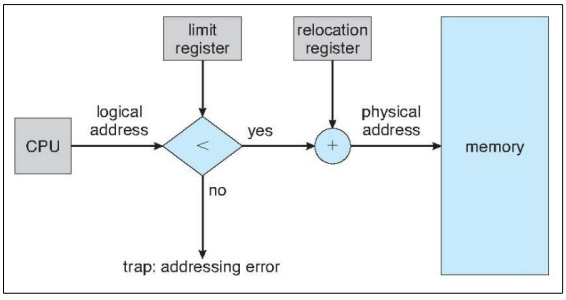


Notiamo che diverse configurazioni delle MMU(Unità gestione della memoria) producono diversi indirizzi fisici dagli stessi indirizzi logici. Come si evince dal disegno la CPU è ignara degli indirizzi fisici, può "configurare" la MMU.

Problema della Protezione (prima dell'uso del paging):



Prima una soluzione consisteva nel usare base e limitare I registri.
Ogni volta che la CPU genera un indirizzo, qualche logica controlla che il suo valore che sia entro i limiti di base e se non viene rispettato il limite viene lanciata un'eccezione.



Poi: Una coppia di registri di riposizionamento e di limite definiscono lo spazio di indirizzamento logico
indirizzo fisico = trasferimento + indirizzo logico.

La CPU deve controllare ogni accesso alla memoria generato in modalità utente per assicurarsi che sia al di sotto del limite di registro.

Registri di rilocazione / limite vengono utilizzati per proteggere i processi utente l'uno dall'altro e dalla modifica del codice e dei dati da parte del sistema operativo. Il registro di base contiene il valore dell'indirizzo fisico più piccolo per il processo. Il registro dei limiti contiene un intervallo di indirizzi logici, ciascun indirizzo logico deve essere inferiore al registro dei limiti. MMU mappa l'indirizzo logico in modo dinamico.

Adesso bisogna dare quindi a ciascun processo un intervallo contiguo di indirizzi e quando un programma termina la sua memoria viene liberata.
Dimensione fissa: numero di programmi limitati dalla memoria.
Problemi:
→ limita numero dei processi, sprecato
→ frammentazione interna della memoria: memoria non utilizzata nella partizione

Dimensione variabile: la quantità di memoria del programma può variare
Problemi:
→ frammentazione esterna: potrebbe esserci sufficiente memoria libera ma non contigua per ospitare un nuovo processo

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

SEGMENTAZIONE
I primi processori avevano indirizzo spazi più grandi dei loro registri.
Come accedere ad una maggiore quantità di RAM?
Uso di registri di segmento, quello memorizzare i bit più alti di un indirizzo (ad es. 8086)
Successivamente è stato migliorato spostando i registri dei segmenti nella RAM, in una tabella di segmenti (Array) → Ciò consente un numero maggiore di segmenti, inoltre, può organizzare la memoria più vicino al layout logico della memoria del programma, e si avrà che diversi segmenti hanno diversi permessi.

(Prima del paging)
L'indirizzo logico è composto da <numero-segmento,offset>.
Tabella dei segmenti: mappa gli indirizzi fisici bidirezionali, ogni voce della tabella possiede
→ base: contiene indirizzo fisico iniziale in cui risiedono i segmenti di memoria.
→ limite: specifica la lunghezza del file segmento.
Il registro di base della tabella di segmenti (STBR) punta alla tabella del segmento in memoria.
Il registro della lunghezza della tabella di segmenti (STLR) indica il numero di segmenti utilizzati da un programma. il numero del segmento s è legale se s <STLR.
Per quanto riguarda la protezione, ad ogni voce della tabella dei segmenti è associato: un bit di validazione che se risulta essere uguale a 0 allora mi trovo un segmento "illegale" , e i privilegi di lettura/scrittura/esecuzione.

PAGING

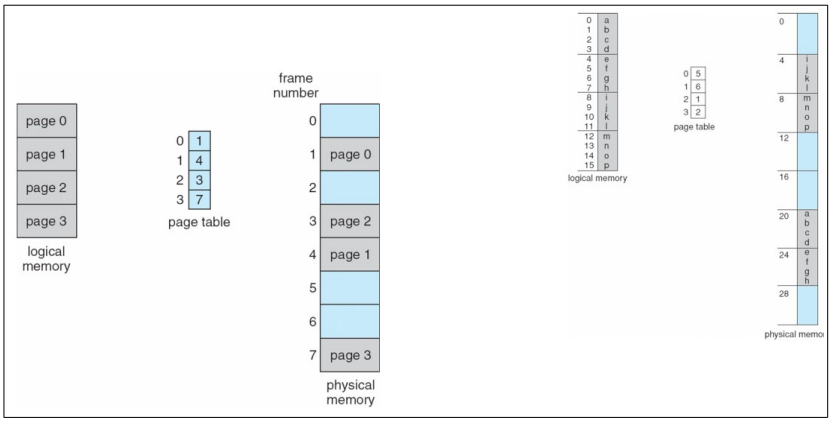
Oggi attualmente per virtualizzare la memoria si va a dividere la memoria fisica in blocchi dimensionati chiamati *frame*, la cui dimensione è una potenza di 2. Invece per quanto riguarda la memoria logica in blocchi della stessa dimensione chiamati *pagine(pages)*. Vado quindi ad impostare una tabella 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***.

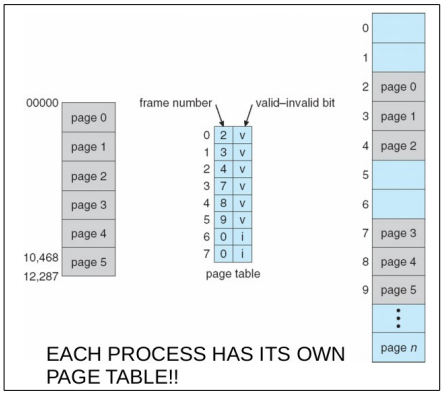
I due problemi di accesso alla memoria possono essere risolti mediante l'uso di una speciale cache hardware di ricerca rapida denominata memoria associativa o buffer di ricerca lato-coda (TLB). Protezione della memoria viene associando il bit di protezione a ciascun frame per indicare se è consentito l'accesso in sola lettura o in lettura-scrittura. È inoltre possibile aggiungere più bit per indicare la pagina solo per l'esecuzione e così via.

Bit "Valido" indica che la pagina associata si trova nello spazio degli indirizzi logici del processo ed è quindi una pagina legale. Bit "non valida" indica che la pagina non si trova nello spazio degli indirizzi logici del processo.

Pro: Nessuna frammentazione esterna
Contro: Frammentazione interna(diminuisce man mano che le pagine si riducono), accessi di memoria più lunghi (2 cicli di RAM), spazio per la tabella delle pagine.



Nel secondo caso (figura a sinistra) n (offset) =2, m=4, quindi 4 byte per pages. I numeri nella tabella delle pagine indicano quindi I "blocchi".



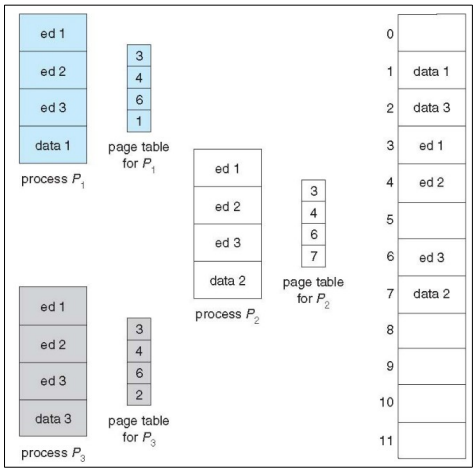
TLB

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+ε) α + (2+ ε)(1-α)
oppure
EAT= $p_{hint}(T_{TLB}+T_{RAM})+p_{fault}(2T_{TLB}+2T_{RAM})$
Calcolato sulla base statistica:
Ricerca TLB → ε, rapporto tra tempo di ricerca TLB e il tempo RAM, deve essere minore di 0.1
Rapporto di apertura (Hit Ratio) → α, percentuale di volte in cui numero di una pagina viene trovato nei registri associativi, rapporto relativo al numero dei registri associativi.

SHARED PAGES



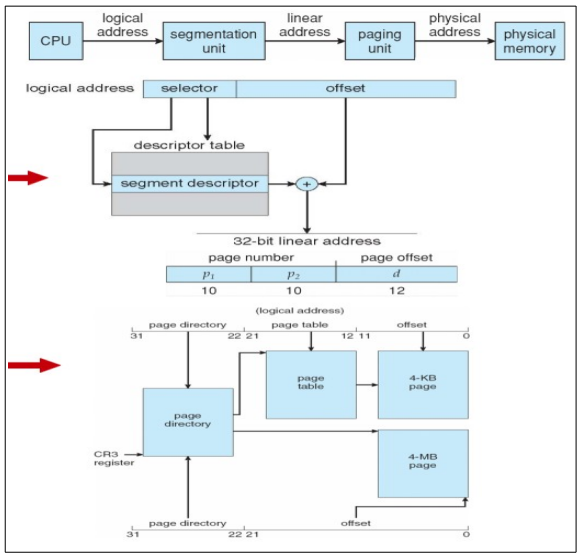
Codice condiviso (Shared Code):
→ Una copia del codice di sola lettura (rientranza) condivisa tra i processi (ad esempio, editor di testo, compilatori, sistemi di window).
→ Simile a più thread che condividono lo stesso spazio di processo.
→ Utile anche per la comunicazione tra processi se è consentita 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.

Hierarchical Page tables - Tabelle della pagina gerarchica

Spezza lo spazio logico degli indirizzi in più tabelle di pagine, consiste in una tecnica semplice, cioè è una tabella di pagina a due livelli.
Pro: Riduce le dimensioni della tabella delle pagine, perchè con le moderne capacità di memoria, la tabella delle pagine può essere ENORME
Contro: aumenta l'EAT (più penalità in caso di mancanze)

IA- 32

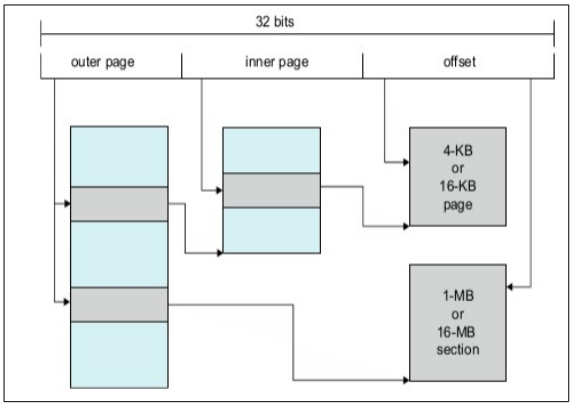


Combina la segmentazione e paging.
La CPU genera indirizzo logico,selettore assegnato all'unita di segmentazione che produce gli indirizzi lineari.

L'indirizzo lineare assegnato a un'unità di paging 2 livelli, la quale genera un indirizzo logico nella memoria centrale.

Le unità di paging sono equivalenti a MMU, le dimensioni delle pagine possono essere 4KB o 4MB.

ARM Architecture



Moderno, a basso consumo energetico, CPU a 32 bit. 4 KB e pagine da 16 KB. 1 MB e 16 MB di pagine (sezioni definite).
Un livello di paging per sezione, a due livelli per pagine più piccole. Due livelli TLB: il livello più esterno ha due TLB micro(un dato, un'istruzione), quello medio è singolo TLB principale, quello più interno è controllato,sui miss outers sono controllati, e sulle miss page della tabella eseguite dalla CPU.

Un processo può essere scambiato temporaneamente dalla memoria a un backing store e quindi riportato in memoria per l'esecuzione continua. Lo spazio di memoria fisico totale dei processi può superare la memoria fisica.

Backing store: disco veloce abbastanza grande da contenere copie di tutte le immagini di memoria per tutti gli utenti; deve fornire accesso diretto a queste immagini di memoria.

Roll out, roll-in - sostituzione della variante utilizzata per gli algoritmi di pianificazione basati su priorità; il processo con priorità più bassa viene scambiato in modo che il processo con priorità più alta possa essere caricato ed eseguito.

La maggior parte del tempo di swap è il tempo di trasferimento; il tempo di trasferimento totale è direttamente proporzionale alla quantità di memoria scambiata. Il sistema mantiene una coda pronta di processi pronti per l'uso che hanno immagini di memoria su disco.

Notiamo che Se i prossimi processi da mettere sulla CPU non sono in memoria, è necessario sostituire un processo e scambiare il processo di destinazione.

VIRTUAL MEMORY

Memoria virtuale: separazione della memoria logica dell'utente dalla memoria fisica. Un'architettura di sistema capace di simulare lo spazio di memoria centrale maggiore di quello fisicamente presente o disponibile, cioè si aggiunge spazio utilizzando uno spazio di memoria secondario su altri dispositivi, per esempio il disco.

- 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.
- Memoria virtuale implementata tramite: *Demand paging o demand segmentation.*

Spazio indirizzi virtuali: visualizzazione logica di come il processo viene archiviato in memoria

- Di solito inizia all'indirizzo 0, indirizzi contigui fino alla fine dello spazio.
- Nel frattempo, la memoria fisica è organizzata nei frame di pagina.
- MMU deve mappare logico a fisico.

Generalmente lo spazio degli indirizzi logici di progettazione per lo stack inizia con l'indirizzo logico Max e cresce "in basso" mentre l'heap cresce "su".
In questo modo massimizzo lo spazio degli indirizzi logici, nessuna memoria fisica è necessaria fin quando l'heap e lo stack non raggiungono una determinata nuova pagina. Memoria viene condivisa mappando le pagine in lettura e scrittura nello spazio degli indirizzi virtuale, ricordando che le pagine possono essere condivise durante la fork(), velocizzando la creazione del processo.

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.

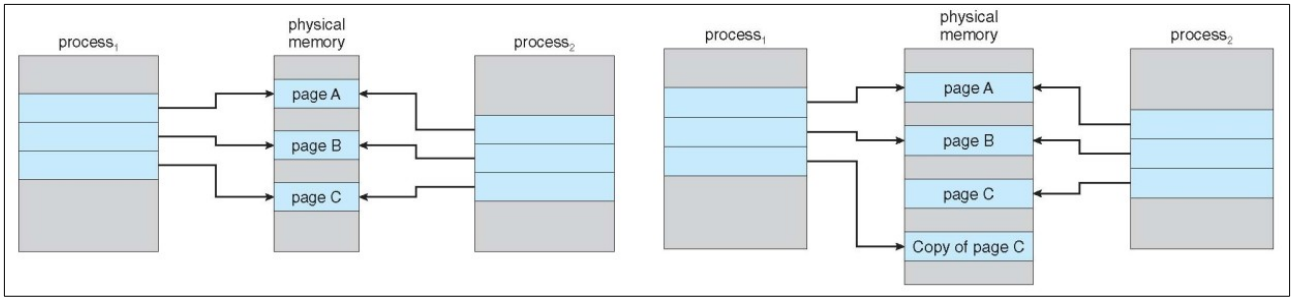
Lazy swapper: non scambia mai una pagina in memoria a meno che la pagina non sia necessaria.

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.

COPY ON WRITE(COW)

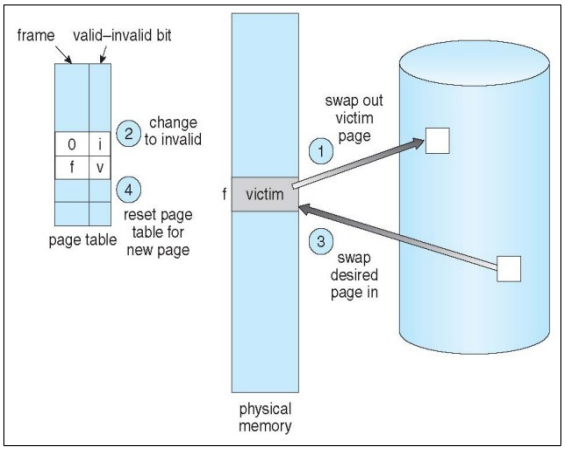
Durante la fork(), replico solo la tabella delle pagine, per puntare ai frame principali, ma vado a mettere un flag sulle pagine. Cioè setto flag "trap_on_write" uguale a 1. Se faccio "write" viene generata una trap e il frame viene copiato e il bit viene cancellato in modo tale che ulteriori accessi non vengano intercettati.



A sinistra abbiamo che 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 è copiato e il valore della tabella delle pagine del processo 1 è aggiornata.

Free Frames: Il sistema mantiene una lista di frame liberi (simili a uno SLAB), per ottenere rapidamente una pagina libera quando necessario. *Per sicurezza:* le pagine libere vengono azzerate (altrimenti un nuovo processo potrebbe leggere i dati di un processo morto).

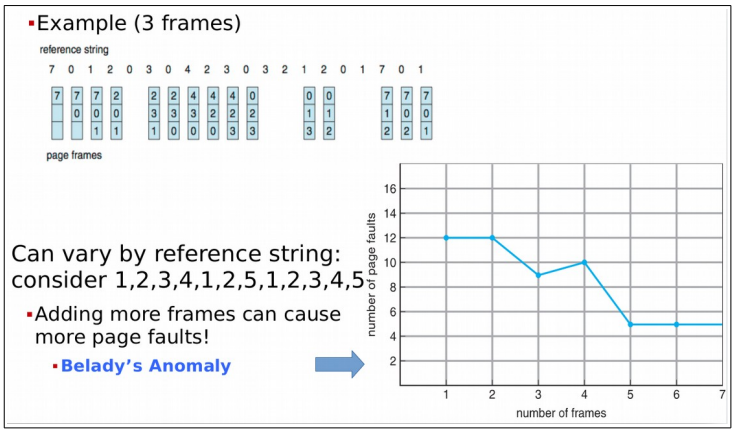
PAGE REPLACEMENT



Trovo la posizione della pagina desiderata su disco. Se trovo un frame libero lo uso, altrimenti utilizzo algoritmo di sostituzione delle pagine per selezionare un frame "vittima". Scrivi il frame vittima sul disco se sporco.

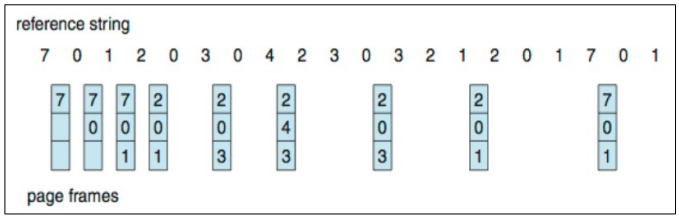
Porto la pagina desiderata nel (nuovo) frame libero; aggiorna la pagina e le tabelle dei frame. Continuo esecuzione processo che era stato bloccato dalla TRAP.

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

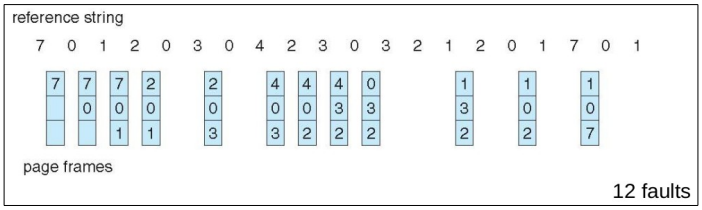


Idea: Sostituire la pagina che non verrà riutilizzata per un periodo di tempo piuttosto lungo.

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



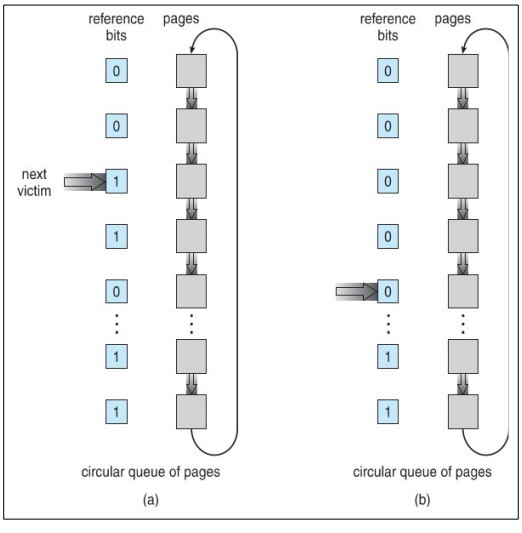
Idea: approssimativa ottimale prevedendo quale pagina verrà utilizzata per ultima.

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.

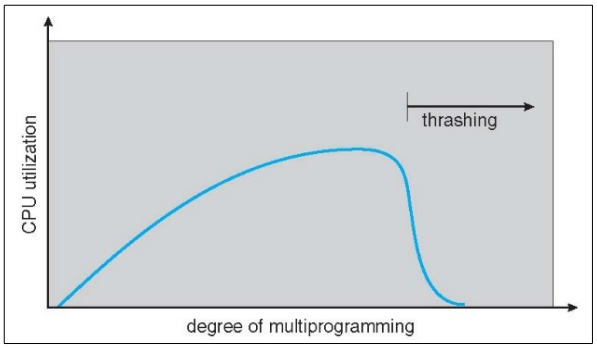
Vantaggi: L'implementazione è efficiente.
Svantaggi: Nel caso peggiore (tutti i bit impostati ad 1), viene scandita l'intera coda, dando ad ogni pagina una seconda chance. Second chance degenera in FIFO.

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.

TRASHING



Un processo è spende più tempo nella paginazione che nella propria esecuzione.
Succede quando:
Il processo non ha pagine "sufficienti", il tasso di errore della pagina è molto alto.
Conseguenze:
Efficienza CPU cala e SO che pensa di aver bisogno di aumentare il livello di multiprogrammazione. Nuovi processi hanno bisogno di acquisire frame, che però non sono liberi, entrano quindi nella coda del dispositivo di paginazione.

WORKING-SET MODEL

Osservare quanti frame stanno attualmente usando un processo,al fine di definire un modello di località.
Un programma è generalmente composto da varie località (insieme di pagine attive).
Quando un processo chiama una procedura, crea una nuova località che contiene istruzioni e variabili locali.
La struttura del programma definisce quindi le località, il trashing si verifica se: la somma delle dimensione della località > dimensione totale della memoria fisica allocata.

Il modello del working-set usa wss (p, t): insieme di pagine cui si accede nell'epoca t:

- t intervallo di tempo sotto analisi
  - se la durata è troppo piccola, non rappresentativa per la località, cioè non vai ad includere l'intera località
  - se la durata è troppo grande, cattura diverse frequenze di guasto delle località e il working set è correlato, cioè più località si sovrappongono
- p id del processo
- Il working-set è quindi un'approssimazione delle località del programma

Numero di frame richiesti all'istante t:
$D(t)=\sum_p wss(p,t)$

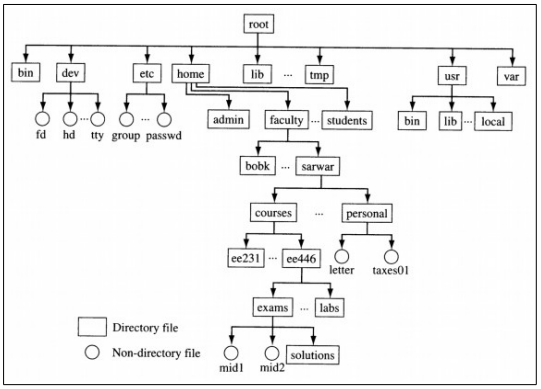
Se D(t)>numero max di frame → fenomeno Trashing

Algoritmi migliori – page fault rate	Anomalia Belady?
Optimal	No
LRU	No
Second-chance	Si
FIFO	Si

FILE SYSTEM

Il File system risiede nella memoria secondaria (dischi). Fornisce un accesso efficiente e conveniente al disco consentendo il salvataggio dei dati, che possono essere recuperati facilmente.

File: blocchi di dati memorizzati.
Directory: raccolta di file o altre directory.
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.

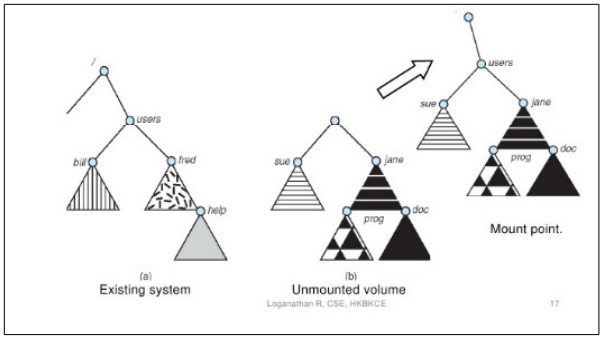


Il file system "root" (/) di solito ha diverse cartelle e alcuni di esse memorizzano file system aggiuntivi.

Per esempio: altri dischi come ad es. una pen drive oppure immagini di file come ad es. l'immagine di Ubuntu, o directory remote come ad es. i file system logici di case remote, popolati dal sistema operativo, non corrispondono necessariamente ai byte fisici sul disco.

/proc: informazioni sui processi in esecuzione
/sys: informazioni sul sistema (cpu / bus)
/dev: visualizzazione diretta di dispositivi fisici

MOUNT



Tramite operazione di mout il sistema operativo viene informato che un nuovo file system pronto per essere usato. L'operazione, quindi, provvederà ad associarlo 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).

FILE

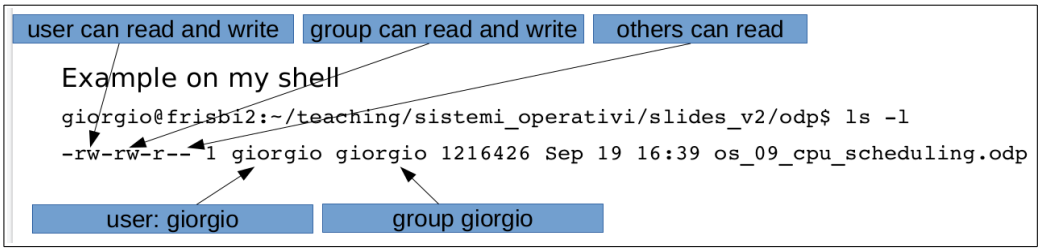
Un file "normale" è una memoria permanente con spazio di indirizzamento contiguo. Identificazione del tipo di file cone il comando della shell, file <filename> , che legge i primi byte di un file cercando di identificare il suo tipo.

un FILE è un puntatore opaco che offre un'interfaccia di alto livello ai file

- fopen
- fclose
- fprintf / fscanf
- fread / fwrite
- fseek

Queste operazioni sono mappate a sysc del sistema operativo specifico e sono implementate per "comportarsi" coerentemente tra i diversi sistemi operativi.

PERMESSI FILE



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.

Notiamo che la directory può ospitare un altro file system se esso è “montato” nella directory. In questo caso i file nella directory non sono accessibili finché il filesystem ospitato non è "smontato",la directory mostrerà la "root" del file system montato.

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.

Viene, quindi, creata una entry all’interno della tabella dei file aperti detenuta dal processo, la quale punterà alla relativa entry nella tabella globale, insieme ad altre informazione - e.g. permessi, locazione del cursore all’interno del file, ecc.

La syscall open() restituisce per l’appunto l’entry all’interno della tabella del processo - il File Descriptor. Piu’ open() su uno stesso file da parte di uno stesso processo generano descrittori diversi.

Tre descrittori sono "assegnati" per file
- 0: stdout
- 1: stdin
- 2: stderr

Tutte le azioni su un file sono fatte attraverso il suo descrittore. Ogni descrittore di file è in "puntatore" a una posizione di file all'interno del sistema operativo.

Le operazioni di lettura / scrittura si verificano nella posizione corrente del puntatore, cioè leggendo n byte dalla posizione x si sposta il puntatore su n + x.
“lseek” sposta il puntatore all’interno del file. Il processo può anche essere messo in pausa in attesa che un file venga modificato tramite select ().

Il filesystem /dev fornisce agganci per i dispositivi fisici nel sistema:
- /dev/sda : disk
- /dev/sda1: prima partizione del prima disco
- /dev/ttyACM0 : FDI
- /dev/video0 : webcam

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.

DISCHI

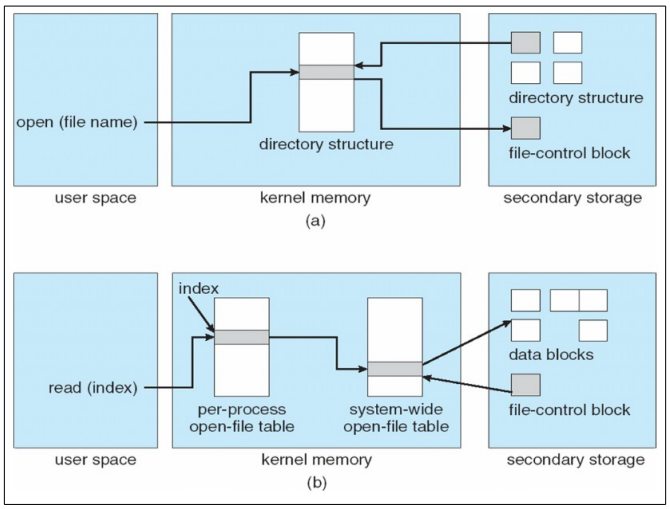
Archiviazione solitamente organizzata in partizioni.
Tabella della partizione: struttura dati che descrive layout del disco.
Si accede al disco tramite un controller del disco, che comprende operazioni come: leggi/scrivi.

La parte di basso livello del driver del disco offre alla parte di alto livello un'interfaccia unificata per queste operazioni. La schedulazione / memorizzazione nella cache sono utilizzati dalla parte superiore del driver, a seconda delle caratteristiche del supporto.

Al livello del SO:
1. leggo blocco xx
2. passo ad un altro processo, il disco gestisce le richieste e scrive in memoria.
3. Interrupt: identificare il disco, riprendere il processo e copiare i dati nello spazio del processo.

File System Driver: componente del sistema operativo che si occupa di un file system specifico; è sempre un interfaccia astratta, una classe virtuale. Per Esempio: un file system può vivere in un file (immagine)

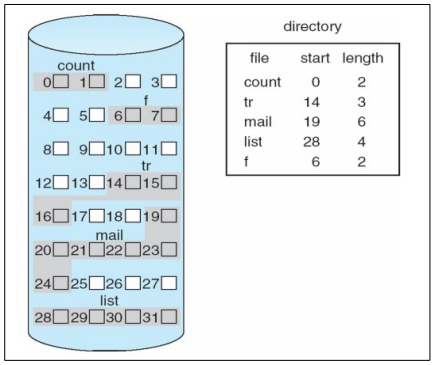
STRUTTURE DATI



Nel Kernel:
Per ogni file aperto, c'è un'istanza di una struttura: di tipo ***"OpenFileInfo"*** creata.
Per ogni descrittore aperto da un processo, le strutture appropriate ***"OpenFileRef"*** vengono inserite in un elenco accessibile sfogliando il PCB.
Ogni OpenFileRef punta al corrispondente OpenFileInfo unico e memorizza un puntatore di file indipendente per la gestione di seek / read / write.

Nel Disco
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.

ALLOCAZIONE FILE – CONTIGUOUS

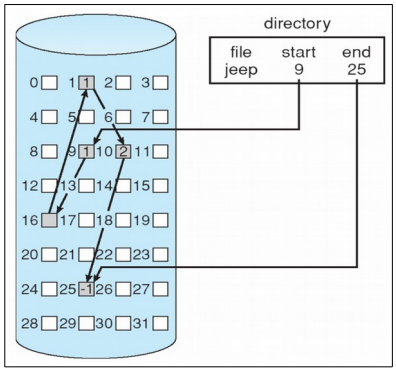


Allocazione contigua: ogni file occupa un insieme di blocchi contigui.

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

ALLOCAZIONE FILE – 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 puo' richiedere numerosi cicli di I/O.

ALLOCAZIONE FILE – FAT

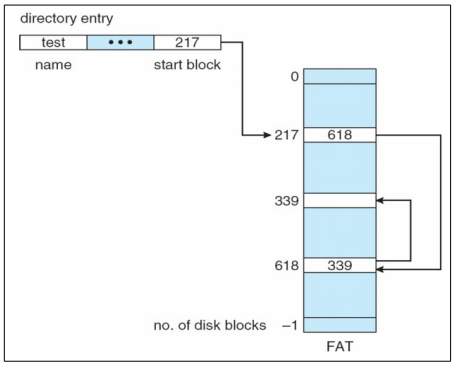
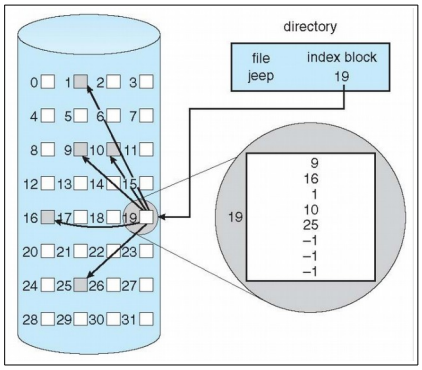


Tabella di allocazione file.
All'inizio del disco, metti un array (FAT)
L'array è un array_list che codifica la sequenza di blocchi.

Lo stesso FAT è relativamente piccolo e rimane nella RAM.
Codifica implicitamente la struttura del blocco (blocchi non validi nell'elenco)

ALLOCAZIONE FILE – 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 entreà nell'index block.

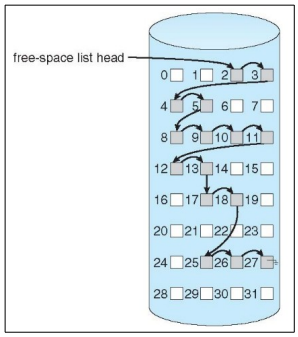
Tale tipo di allocazione permette di guardagnare velocità rispetto ad una implementazione tramite linked list nel caso in cui si effettuino molti accessi scattered ai blocchi (non bisogna scorrersi tutta la lista, ma basta fare una ricerca).
Il costo da pagare è lo spazio necessario a contenere l'index block stesso.

IMPLEMENTAZIONE DIRECTORY

Lista lineare di nomi di file con un puntatore ai blocchi dati: semplice ma ha un elevato tempo di ricerca (tempo lineare)
Hash Table: lista lineare con struttura dati hash: riduce tempo di ricerca nella directory ma vi possono causare delle *collisioni,* situazione in cui due nomi di file hash vogliono accedere alla stessa posizione.

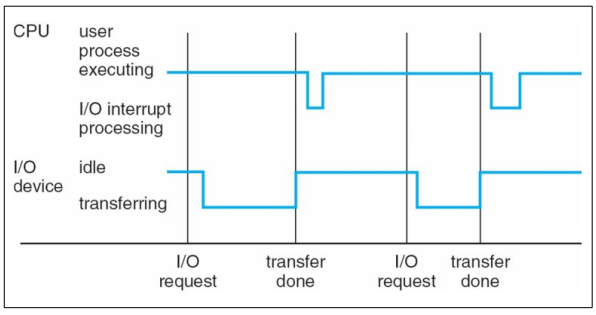
GESTIRE SPAZIO LIBERO

Bitmap (formato dati per la rappresentazionei di immagini) : Mantieni una bitmap all'inizio del disco, dove ogni bit corrisponde a un blocco. Quando cerchi un blocco libero, usa la bitmap per trovare quello più vicino.



Lista collegata: SLAB Allocator che collega tutti I blocchi liberi su disco.

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.

Cosa succede se un programma utente altera il vettore di interrupt?
Cosa succede quando un programma utente scrive cose casuali su un dispositivo mappato in memoria (ad esempio il controller del disco)?

Impedire al programma di farlo definendo due modalità operative per la CPU: modalità privilegiata e utente.
In modalità utente può essere eseguito solo un sottoinsieme delle istruzioni
Cambiare bit flag è un operazione per chi possiede la modalità privilegiata
ISR vengono sempre eseguiti in modalità privilegiata

ECCEZIONI

Eccezioni sono interruzioni attivate dal Software
3 categorie:

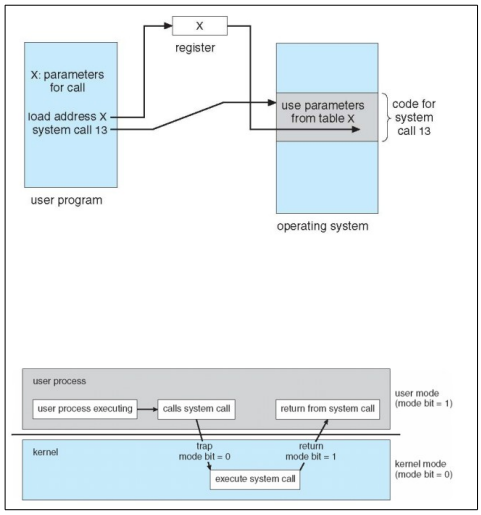
- TRAP: ISR viene chiamata dopo attivazione di un'interruzione
- FAULT : ISR chiamato prima attivazione dell'interruzione
- ABORT: Stato del processo che ha subito interrupt non può essere recuperato

INT VS CALL

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.

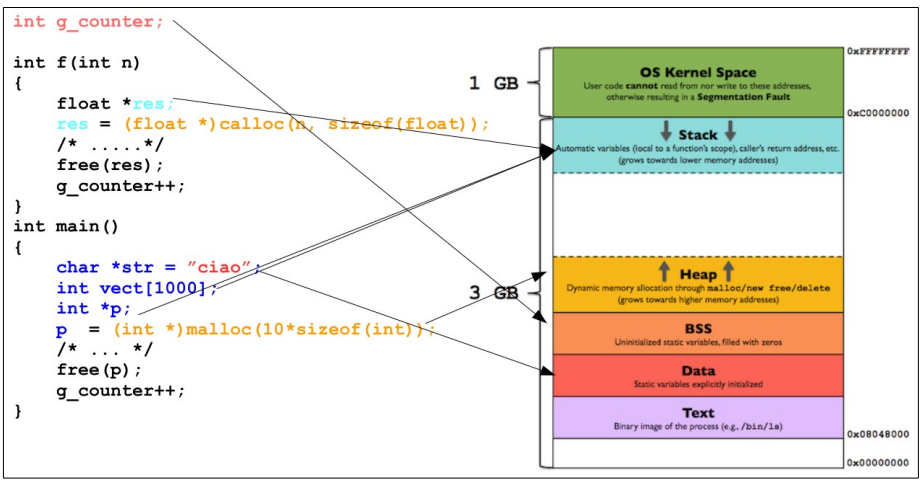
Nel caso si volesse aggiungere una nuova syscall, essa, una volta definita, va registrata nel sistema e
aggiunta al vettore delle syscall del sistema operativo, specificandone il numero di argomenti.

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.

PROCESSO



Un processo è un programma in esecuzione.
In un sistema operativo multitasking, possono essere contemporaneamente eseguite più istanze dello stesso programma.
Il processo è caratterizzato dai *Registri della CPU, Memoria* (in particolare heap e stack), *Risorse* (file descriptor/socket, semafori, code).

Processo può aver diversi stati: *ready* (processo pronto per essere eseguito), *waiting* (in attesa di risultato magari dovuto ad una chiamata I/O), *running* (processo in esecuzione).

Stack frame è un puntatore ad un registro speciale che memorizza i record di attivazione per ciascuna funzione.

FORK()

```
// typical fork example

int main(int argc, char** argv) {
     int v=fork();
     if (v){
          // we are in the parent;
          doParentStuff();

          // see in 2 slides
          // waits for child termination;
          int retval;
          int pid=wait(&retval)
     } else {
          doChildStuff();
          return 0
     }
}
```

System call utilizzata per creare un nuovo processo. Dopo la fork viene quindi generato un processo figlio il quale copia la memoria dal processo padre (colui che ha fatto la fork()), i vari file descriptor e le risorse.
Se il valore di ritorno della fork = 0 allora mi trovo nel processo figlio, altrimenti nel padre.

WAIT() : sospende esecuzione del processo padre finché uno dei figli non termina; utilizzando la funzione waitpid, aspetto che termini un figlio specifico (inserendo pid specifico).

Cosa succede quando un processo genitore termina prima di suo figlio?
Il processo "orfano" diventa figlio della madre di tutti i processi (init/systemd), ovvero primo processo avviato dal sistema.

```
const char parent_prefix[]="parent";
const char child_prefix[]="child";
const char* prefix=parent_prefix;
pid_t pid;
const int num_rounds_parent=10;
const int num_rounds_child=5;

void childFunction() {
  for (int r=0; r<num_rounds_child; ++r) {
    printf("%s looping, pid: %d, round: %d \n",
          prefix, pid, r);  sleep(1); }
}
void parentFunction() {
  for (int r=0; r<num_rounds_parent; ++r) {
    printf("%s looping, pid: %d, round: %d \n",
         prefix, pid, r); sleep(1);  }
}
int main(int argc, char** argv) {
  pid=getpid(); // here we store the process id
  printf("%s started, pid: %d\n", prefix, pid);
  printf("%s now forking\n", prefix);
  pid_t fork_result=fork();
  if(fork_result==0){
    prefix=child_prefix;
    pid=getpid();
    printf("%s started, pid: %d\n", prefix, pid);
    childFunction();
  } else {
    printf("%s continuing, pid: %d\n", prefix, pid);
    parentFunction();
  }
  printf("%s terminating, pid: %d\n", prefix, pid);
}
```

Genero due processi: padre e figlio.
Entrambi eseguono un ciclo inutile per un certo numero di round.
(funzioni void parentFunction() e void childFunction())
Uno dei due morirà.

Cosa succede se il padre muore prima del figlio?
Quando un genitore termina tutti i suoi figli vivi sono avvisati della conclusione tramite un altro segnale (SIGHUP).

Cosa succede se il processo figlio muore prima del padre?
Processo padre viene informato dal SO, tramite segnale (SIGCHILD), della terminazione di una dei suoi figli.

EXIT()

Un processo termina la sua esecuzione con exit (retval).
Un processo terminato entra nello stato terminato (zombie) e tutte le sue risorse vengono rilasciate.
Un processo di zombie rimane attivo fino a quando il genitore legge il suo valore di uscita tramite un wait.
Quando ciò accade, il processo cessa di esistere. Init / systemd attende periodicamente i propri figli. Ciò garantisce che i processi orfani che terminano non siano zombie per sempre.

EXEC()

```
int main(int argc, char** argv) {
  if (argc<3) {
    printf("usage %s <int> <path> <args>\n",
    argv[0]);
  }

  char* prog_path=argv[2];
  int num_instances=atoi(argv[1]);
  int active_instances=0;
  printf("starting program %s in %d instances\n",
         prog_path, num_instances);

  for (int i=0; i<num_instances; ++i){
    pid_t child_pid=fork();
    if(! child_pid){
      int result=execv(prog_path, argv+2);
      if (result) {
     printf("something wrong with exec errno=%s\n",
          strerror(errno));
      }
    } else
      active_instances++;
  }
  int status;
  while(active_instances) {
    pid_t child_pid = wait(&status);
    printf("son %d  died, mourning\n", child_pid);
    active_instances--;
  }
  printf(" launcher terminating\n");
  return 0;
```

Carica in memoria del processo chiamante un nuovo programma da eseguire, il contenuto precedente viene distrutto (memoria e risorse comprese), iniziando cosi ad eseguire il nuovo programma.

L'execvp(const char * path, const char* argv[]) dove primo argomento è il programma da eseguire,il secondo argomento è un array di stringhe con terminazione null.

Un processo da eseguire potrebbe richiedere informazioni aggiuntive: *variabili di ambiente e parametri principali (argc, argv).*

Le variabili d'ambiente sono accessibili attraverso un array di stringhe terminate NULL globale: ***char ** environ***

ogni voce ha il modulo "Nome = valore" e l'ultima voce è 0.

VFORK()

```
int main(int argc, char** argv) {
  if (argc<3) {
    printf("usage %s <int> <path> <args>\n",
    argv[0]);
  }

  char* prog_path=argv[2];
  int num_instances=atoi(argv[1]);
  int active_instances=0;
  printf("starting program %s in %d instances\n",
         prog_path, num_instances);

  for (int i=0; i<num_instances; ++i){
    pid_t child_pid=vfork();
    if(! child_pid){
      int result=execv(prog_path, argv+2);
      if (result) {
     printf("something wrong with exec errno=%s\n",
         strerror(errno));
      }
    } else
      active_instances++;
  }
  int status;
  while(active_instances) {
    pid_t child_pid = wait(&status);
    printf("son %d  died, mourning\n", child_pid);
    active_instances--;
  }
  printf(" launcher terminating\n");
  return 0;
}
```

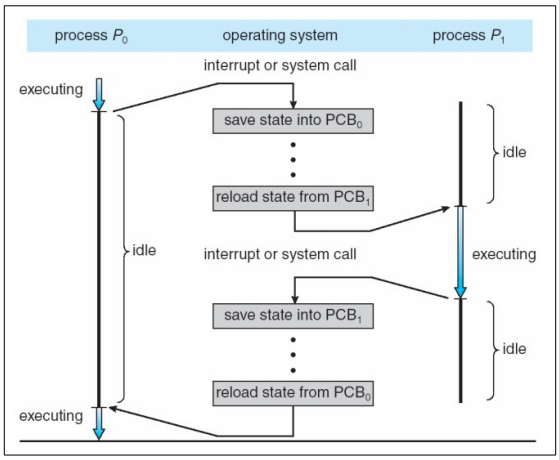
Stesso comportamento della fork, ma viene usato per creare nuovi processi senza però far si che si vengano a copiare le tabelle della pagina del processo principale.

vfork () differisce da fork per il fatto che è specializzata nel caso l'istruzione immediatamente successiva alla fork(), nel processo figlio sia una exec(); inoltre non replica la memoria del processo padre, questo perchè 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(). Se non viene fatta l'exec dopo la vfork(), genero comportamenti non specificati.

PCB

Struttura dati che memorizza le informazioni del processo, in particolare mantiene le informazioni sullo stato del processo; inoltre si trova un'area privilegiata della memoria. PCB di solito contiene ID del processo, ID dell'utente, stato del programma(ready,waiting,running),Stato della CPU del processo(registri), informazioni sulla memoria(tabella delle pagine), informaizioni I/O.

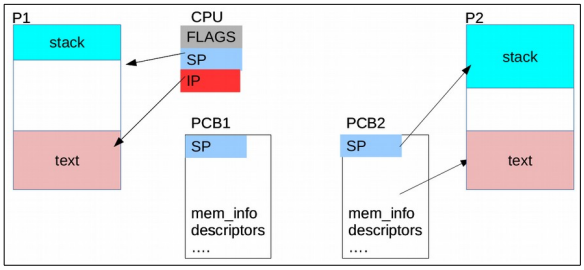
CONTEXT SWITCH



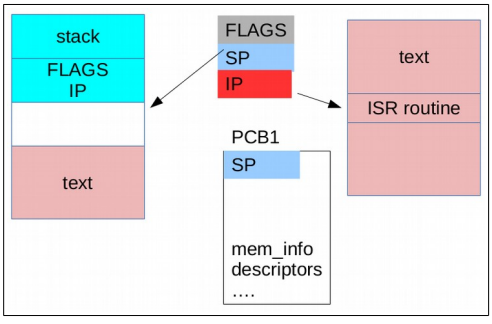
Il context switch si verifica quando un processo in esecuzione viene interrotto. Questo può accadere solo a causa di un interrupt. Questi eventi causano l'esecuzione del codice del kernel.
Quando viene restituito il codice del kernel, il successivo processo che accederà alla CPU potrebbe o potrebbe non essere quello interrotto, a seconda della decisione del kernel.

Per esempio (figura sx) per poter rimuovere dall'esecuzione un processo P0 e quindi eseguire un nuovo processo P1, il SO deve salvare lo stato corrente del processo P0 in modo da poter rirpistinare l'esecuzione dello stesso in un secondo momento. Le informazioni di un processo sono contenute nel suo Process Control Block (PCB).

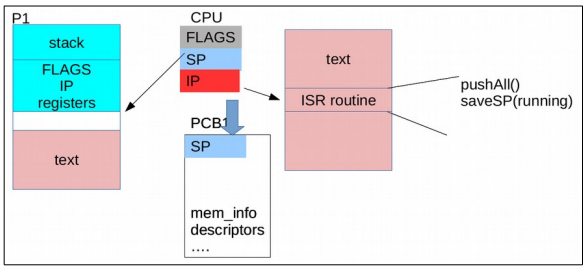
E' bene notare che il context switch è fonte di overhead a causa delle varie operazioni di preambolo e
postambolo necessarie allo switch - e.g. salvare lo stato, blocco e riattivazione della pipeline di calcolo, svuotamento e ripopolamento della cache.



Assumiamo di avere due processi : P1 e P2 nello stato ready.
P2 è stato prelazionato, e il suo stato è stato salvato in PCB2.
P1 è in esecuzione
Cosa dovrebbe succedere affinchè a seguito di un'interruzione la CPU venga assegnata a P2?



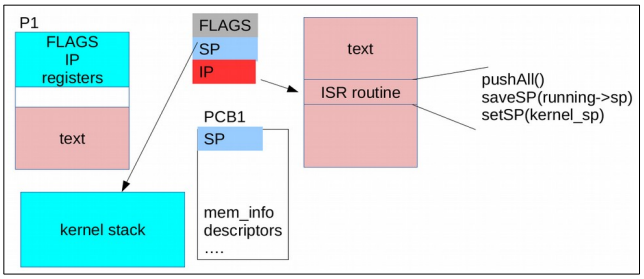
Arriva l'interrupt, quindi la CPU salva i flag e il contatore delle istruzioni sullo stack e passa alla modalità privilegiata e gestisce l'ISR appropriato.



Per poter ripristinare P1 nel futuro, occorre salvare lo stato della CPU nel PCB.

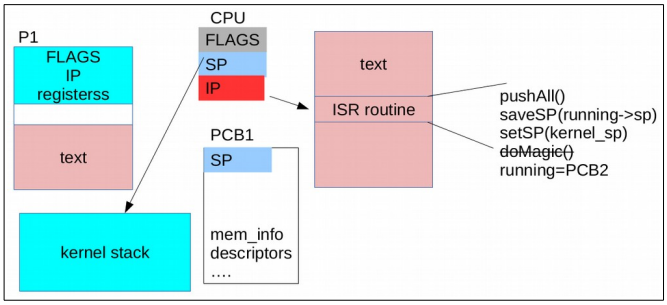
É buona norma salvare salvare sullo stack registri che verranno alterati in questo processo e copiare il loro valore dallo stack al PCB.

Nel salvare lo stack pointer sul PCB occorre alterarlo (SP+size_of_saved_register), in modo da ottenere il suo valore all'entrata nella ISR.



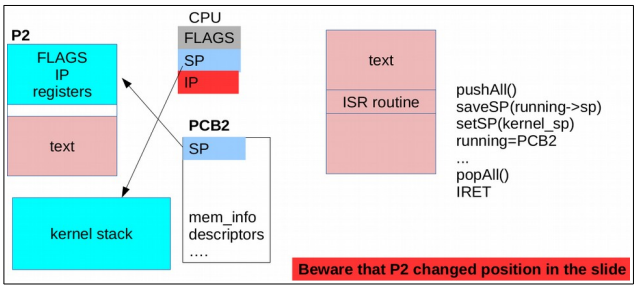
A questo punto si esegue il cambio di stack,commutandolo ad uno stack del kernel.

Cambiare stack garantisce che eventuali alterazioni da parte del processo nel suo stack non abbiano effetto sul comportamento del kernel.



Se la trap è stata attivata da un syscall, potremmo voler cercare i parametri sul PCB o sullo stack di P1.

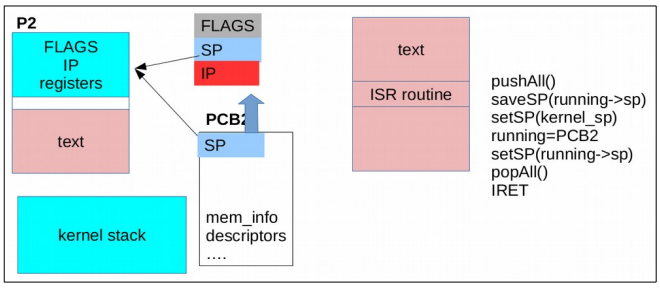
Se il nostro compito è solo quello di eseguire un cambio di contesto(context switch) su P2, dobbiamo selezionare P2 come prossimo



Supponiamo che P2 sia il nostro prossimo processo da eseguire e dato che P2 è prelazionato (preempted), quindi salvato lo stato nel PCB2.

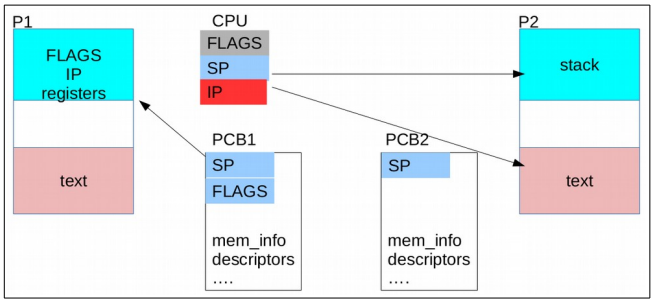
Sappiamo inoltre, che l'ultima istruzione che viene eseguita in modalità kernel sarà un ritorno da interrupt (IRET), in cui verranno recuperati I flag.

L'istruzione IRET ripristina i flag dello stack, commuta in modalità utente e prosegue l'esecuzione dal valore dell'IP nello stack. Il suo risultato dipende dalla consistenza dello stack.
Lo stack di P2 era stato assunto consistente, pertanto l'operazione da eseguire prima della IRET è cambio di stack.



Al fine di garantire la corretta esecuzione di P2 dobbiamo ripristinare lo stato della CPU a prima che l'interruzione che ha prelazionato P2.

Si copiano I valori dal PCB2 alla CPU.



Alla fine P2 continua la sua esecuzione come se nulla fosse accaduto.
Stato di P1 è al sicuro nel PC.

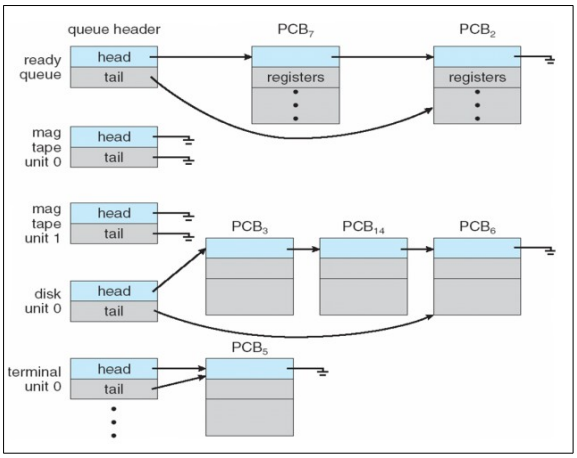
```
pushAll()
saveSP(running->sp)
setSP(kernel_sp)

doMagic()

setSP(running->sp)
popAll()
IRET
```

Un ISR generico non segue le consuete convenzioni di chiamata C.
Osserviamo che le istruzioni di entrata (preambolo) salvano lo stato della CPU nel PCB e le istruzioni in uscita (postambolo) lo ripristinano.

Utilizzo lo stack pointer (SP) ,push, pop per manipolare I registri.
Se per esempio un syscall vuole leggere alcuni argomenti, li recupera dallo stack del processo corrente (o dai registri), accessibile/i tramite SP salvato nel PCB corrente.

Questo vuol dire che la doMagic() può implementare una generica system call del kernel, senza doversi preoccupare di ripristinare lo stato del processo.

SCHEDULER PROCESSI



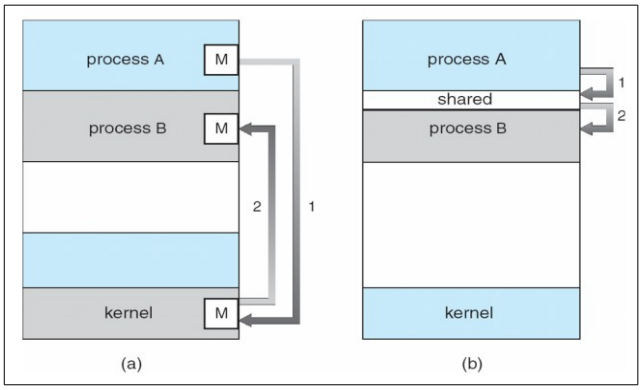
Lo scheduler sceglie prossimo processo da eseguire e lo sceglie in base al contesto di utilizzo:
Mainframe → massimizzare le risorse
Desktop → minimizzare tempi di reazione

Inoltre lo scheduler usa le code (queue) per gestire I processi in esecuzione, di solito implementate come linked list di PCB.

SO mantiene aggiornate le varie code dello scheduler:
Job queue: insieme di tutti I processi in esecuzione
Ready queue: insieme processi residenti nella main memory in attesa, ma pronti per essere eseguiti
Device queue: insieme processi in attesa di qualche operazione I/O

Dipendentemente dallo stato del sistema I processi vengono spostati da una coda ad un'altra.

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).