Programma di Massima

Introduzione ai Sistemi Operativi

- storia
- servizi
- funzionalita'

Complementi di programmazione in C per Sistemi Operativi

- implementazione di concetti OOP
- strutture dati polimorfiche
- allocatori di memoria

Architetture hardware

- cpu/memoria/bus dati e indirizzi
- esempio di bare metal programming su microcontrollore

Interrupt

- sviluppo in interrupt handler su microcontrollore

Stack e context switch

- context switch su microcontrollore
- context switch in user space

Processi e Strutture del Kernel

- dual mode, meccanismo delle system call
- implementazione di spawn, exec, wait
- creazione di uno scheduler preemptive in user space

Thread

- concetti in pthread
- scheduling di thread

CPU Scheduling

- metriche
- FIFO, SJF, SRJF, Round Robin, Round Robin con Priorità
- controllare lo scheduler in PTHREAD

IPC

- motivazioni
- mutex
- semafori
- code
- memoria condivisa

Memoria

- protezione della memoria
- metriche
- segmentazione
- paginazione
- traduzione di indirizzi

Memoria Virtuale

- metriche
- analisi delle prestazioni
- algoritmi di rimpiazzamento delle pagine
- copy on write

File System

- interfaccia al file system
- controllo degli accessi
- operazioni sui file

Implementazione del File System

- virtual file system
- Esempi: FAT, Inodes

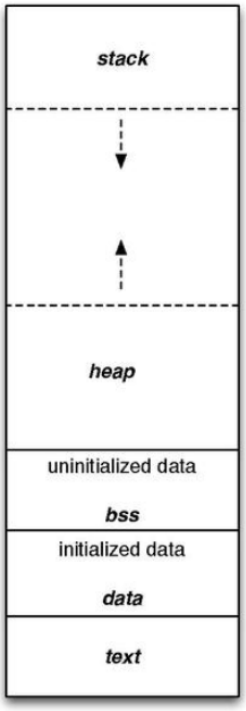
Context Switch:

Che cos'è il contesto di un processo?

Il contesto di un processo è formato da:

a. I registri della CPU

b. La memoria di processo:



i. Stack (.stack)
ii. Heap
iii. .bss: variabili non inizializzate
iv. .data: variabili inizializzate es. static
v. Program Code (.text), in read only durante l'esecuzione

salvando i valori di queste componenti in un Control Block posso fermare l'esecuzione di un processo e ripristinarla più avanti.

Come fare context switch a user level?

In C abbiamo una libreria apposita chiamata Ucontext

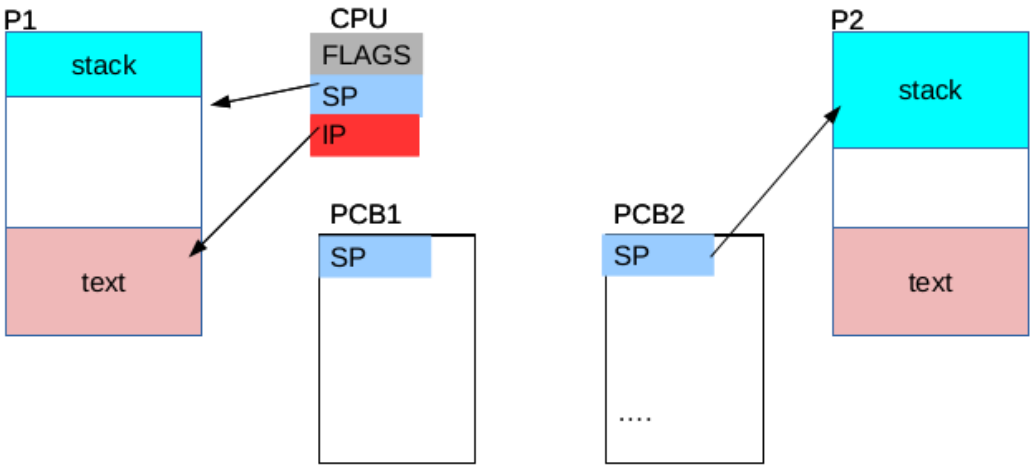
Come fare context switch su un microcontrollore AVR?

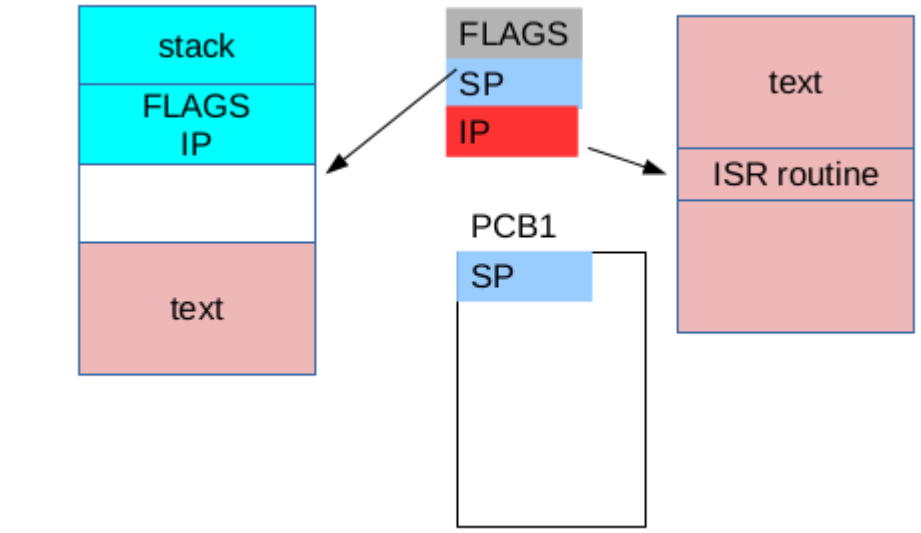
Implementiamo uno switcher di task che sia preemptive e controllato da un timer, usando una double linked list di TCB (task control block).

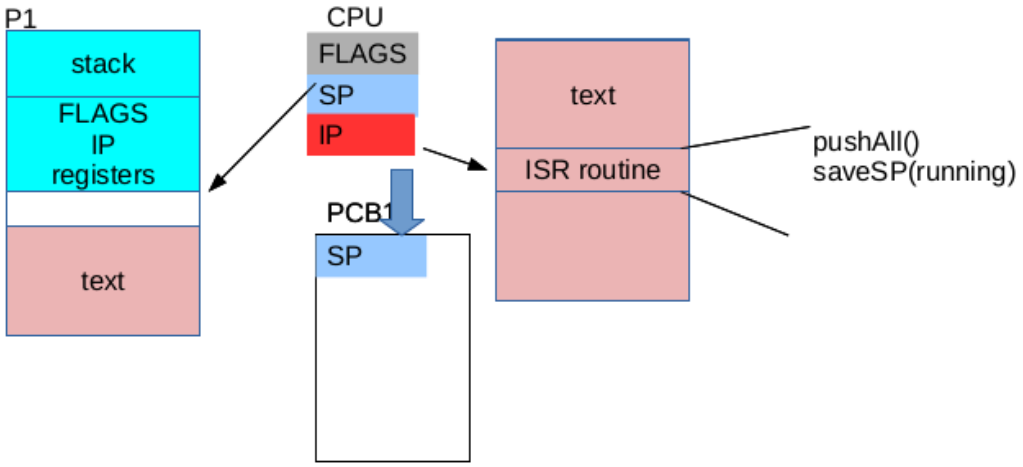
```
TCB {

    stack_pointer_salvato

    puntatore a funzione_che_deve_eseguire
    argomenti_per_funzione

    TCB_prev
    TCB_next
```

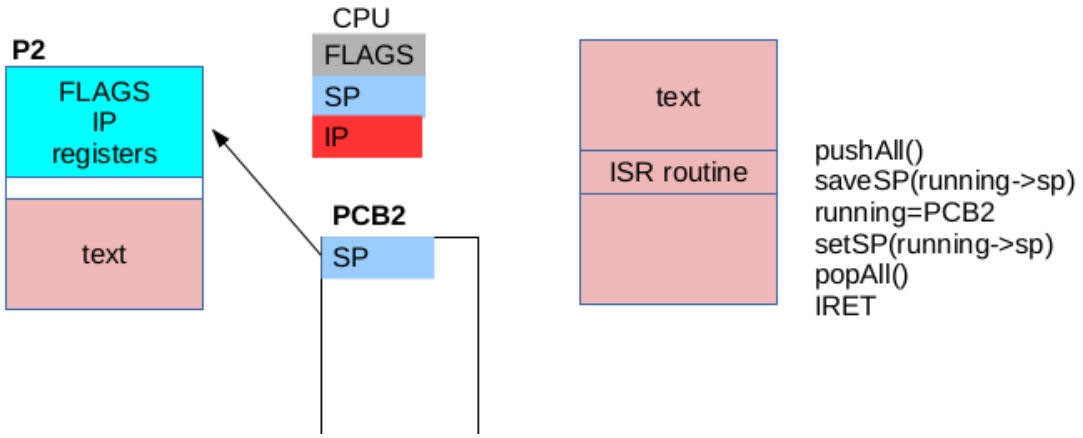
```
        puntatore_stack_bottom  (sembra non usarlo nel codice)
        stack_size

        status
}


TCBList {
        TCB* first
        TCB* last
        size
}
```

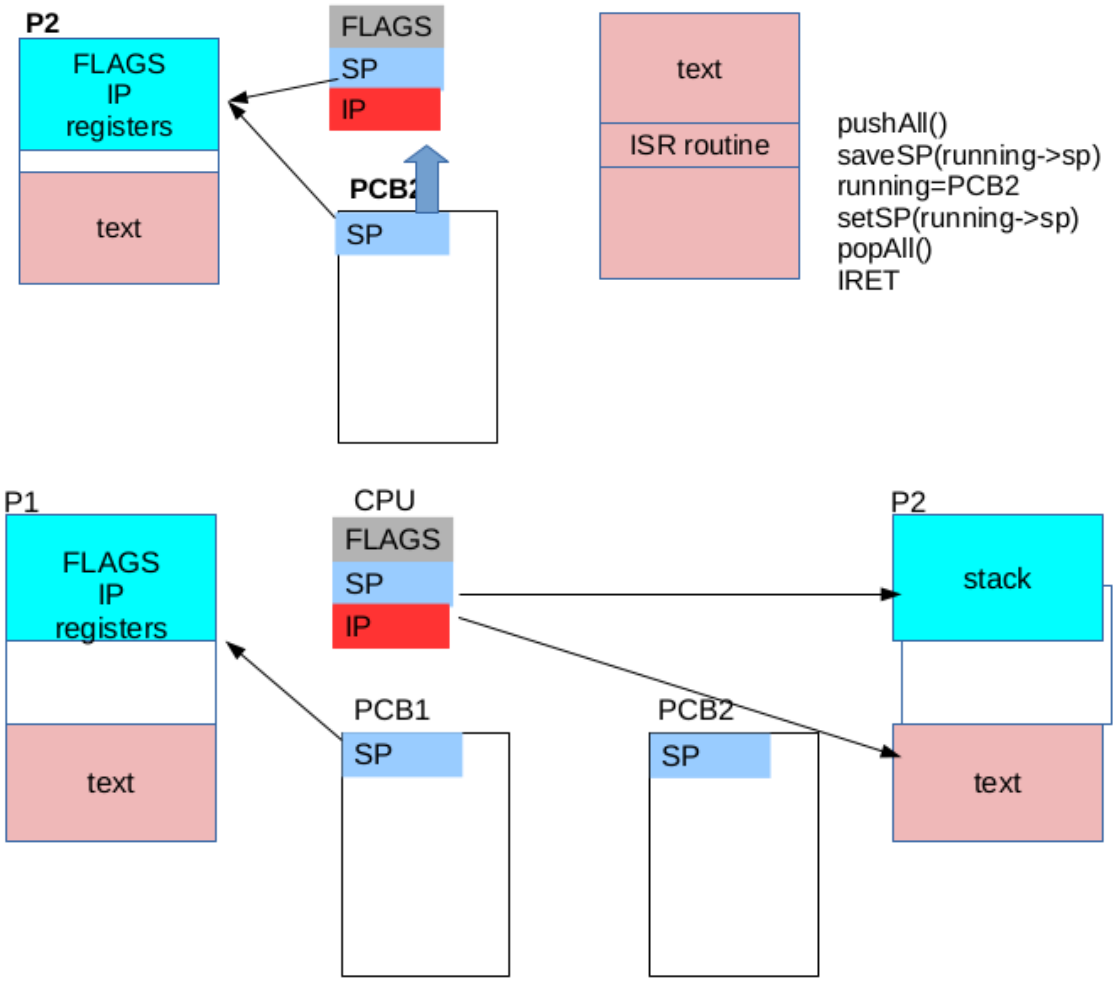
a. **Inizializzazione Processo** (TCB_create()): popoliamo i campi del TCB e inizializziamo la stack a 0, in cima alla stack deve esserci il valore di ritorno di _trampoline, una funzione senza argomenti che lancia i thread. Questa funzione è comoda perché così ogni volta che chiamo un thread non devo preoccuparmi delle calling convention ma chiamo lei e lei pensa a dare i giusti argomenti.
b. **Start (startSchedule()): pulliamo lo stack pointer dal TCB e mettiamo il processo a ready.**
c. **ContextSwitch** (schedule()) quando avviene un'interrupt
    - pushAll(): salva flag CPU, registri CPU e IP sulla stack di P1
    - saveSP(running -> SP): salva SP nel TCB di P1
    - running = PCB2
    - setSP(runningSP): dal TCB di P2 prendi l'SP e sostituiscilo
    - popAll(): pulliamo flag CPU, registri CPU e IP dalla stack di P2
    - IRET: Ritorna dall'interrupt




stack
FLAGS
IP
text
FLAGS
SP
IP
text
ISR routine
PCB1
SP


P1
stack
FLAGS
IP
registers
text
CPU
FLAGS
SP
IP
text
ISR routine
pushAll()
saveSP(running)
PCB1
SP


P2
FLAGS
IP
registers
text
CPU
FLAGS
SP
IP
PCB2
SP
text
ISR routine
pushAll()
saveSP(running->sp)
running=PCB2
setSP(running->sp)
popAll()
IRET



Memory Allocator:

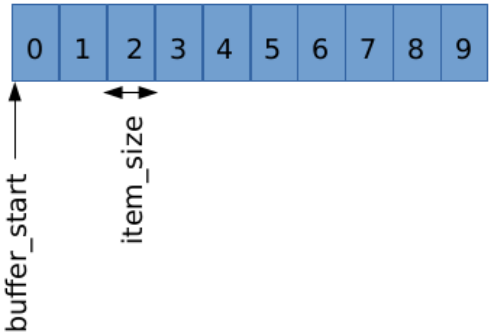
Parlami degli SLAB (pool) allocator:

Gli allocatori SLAB restituiscono blocchi della stessa dimensione *item_size.* Sappiamo inoltre il numero massimo *size_max* di item che si possono avere (così posso fare un Array List statico).

Ogni blocco è identificato da un indice *idx* e inizia all'indirizzo *ptr_block.*

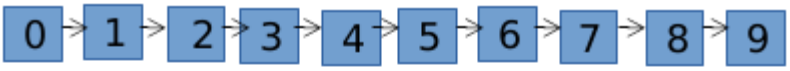
ptr_block = buffer_start + idx*item_size

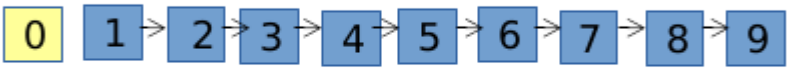
idx = (ptr_block - buffer_start) / item_size se idx non è intero ptr_block non è l'indirizzo di inizio di un blocco!



In questo modo posso restituire all'utente l'indirizzo del blocco in tempo O(1).

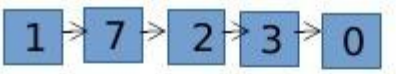
Ho bisogno però di mantenere traccia dei blocchi che sono attualmente liberi. Uso una struttura in cui ogni nodo contiene l'idx di un blocco libero. Quando l'utente richiede un blocco viene staccato il nodo in testa che contiene l'indice idx, quindi l'utente avrà accesso al blocco di memoria idx. Quando l'utente restituisce un blocco viene creato e inserito un nodo corrispondente in testa.

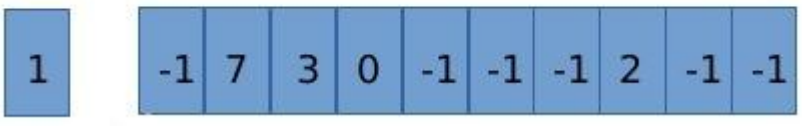




Non possiamo usare una linked list perché necessiteremo della malloc, allora usiamo un Array List e un intero che mantiene la *start_pos* dell'Array List (nella figura = 1)

A questo punto abbiamo tutto, riassumendo:






```
typedef struct PoolAllocator {

        // questa è la vera e propria memoria che l'allocatore gestisce
        char* buffer;
```

```
    // dimensione del buffer di memoria
    int buffer_size;

    // questo è l'Array List che mantiene la free_list, ha dimensione size_max
    int* free_list;

    // numero di blocchi attualmente disponibili
    int size;

    // numero massimo di blocchi che l'allocator può darti
    int size_max;

    // dimensione di un blocco
    int item_size;

    // puntatore al primo bucket dell'array list (sarebbe start_pos)
    int first_idx;


    // size of a bucket
    int bucket_size;  NON HO ANCORA CAPITO COSA SIA QUESTO

} PoolAllocator;
```

L'intefaccia dello SLAB è questa:

- PoolAllocator_init(allocator, item_size, num_items, memory_block, memory_size)

  memory_block è un buffer che l'allocator usa per mettere nella prima parte la memoria vera e propria, e nella seconda parte l'Array List per gestire la memoria vera e propria.

- PoolAllocator_getBlock(allocator)
- PoolAllocator_releaseBlock(allocator, block)

Parlami dei Buddy allocator:

Si partiziona ricorsivamente la memoria in due, ottenendo da un blocco "genitore" due "buddy". Seppur in questo modo si riescono ad allocare blocchi di dimensione arbitraria, si perde memoria quando si prova a allocare un blocco che è più piccolo della partizione minima.

Questo partizionamento ricorsivo porta a vedere la memoria gestita dal buddy strutturata in un albero binario.

Ricordiamo che, in un albero binario:

1. level(idx) = floor(log2(idx))
2. firstIdx(level) = 1 << level [sarebbe 2^level]
3. offset del nodo idx rispetto all'inizio del livello: idx - firstidx(level(idx))
4. indice del buddy di idx: buddyIdx(idx) = (idx % 2) ? idx-1 : idx+1
5. parentIdx(idx) = floor(idx / 2)

```
BuddyListItem {

        ListItem list;  // ogni buddy è un nodo della lista del livello

        int idx; // tree index

        int level; // level for the buddy

        char* start; // start of memory

        int size;  // dimensione della memoria

        struct BuddyListItem* buddy_ptr; // nato con lui quando si è splittato il parent

        struct BuddyListItem* parent_ptr;
}
```

Si mantiene una free list per ogni livello dell'albero, per allocare i BuddyListItem, che sono tutti della stessa dimensione, usiamo uno SLAB allocator. Saranno poi i campi "start" e "size" di BuddyListItem a variare in funzione del livello.

```
BuddyAllocator {

        ListHead free[MAX_LEVELS];

        ListHead occupied[MAX_LEVELS];

        int num_levels;
```

```
        PoolAllocator list_allocator;

        char* memory; // the memory area to be managed

        // the minimum page of RAM that can be returned
        int min_bucket_size; //NON HO ANCORA CAPITO COSA SIA QUESTO
}
```

L'intefaccia del Buddy Allocator è questa:

- init(BuddyAllocator, num_levels, buffer_internal_pool, buffer_size, memory, min_bucket_size)
- malloc(BuddyAllocator, size):
    - dalla dimensione richiesta calcola il livello
    - chiama getBuddy(level) che restituisce un buddy dalla lista level-esima.
        - Se a livello level la lista è vuota viene smezzato ricorsivamente un blocco di livello superiore
            - Trovato un blocco, si chiamano due CreateListItem(). CreateListItem() chiama il Pool allocator che restituisce un blocco di memoria. Questi due blocchi di memoria vengono attaccati come figli dell'attuale blocco trovato.
            - Il blocco viene inserito nella lista i-esima con List_pushBack()
            - Questa funzione viene chiamata ricorsivamente. Tornati al livello di partenza "level" può essere ora ritornato un buddy dalla lista level-esima
        - Se al livello corrente esiste un blocco questo viene detachato dalla lista e ritornato
    - Restituisce l'indirizzo 8 byte più avanti in modo da poter mettere l'indirizzo di inizio del blocco nei primi 8 byte
- free(BuddyAllocator, memory):
    - da memory (puntatore alla memoria utilizzata dall'utente) si va indietro di 8 byte per ottenere l'indirizzo *buddy_ptr* del buddy. Otteniamo quindi il buddy *item* dereferenziando *buddy_ptr*
    - chiama releaseBuddy(item) che rimette item nella sua lista
        - Se il buddy di item è free si fa il merge: si distruggono sia item che il buddy con destroyListItem e si chiama ricorsivamente releaseBuddy() sul parent dei buddies.

Problemi dei Buddy Allocator:

1. Spazio sprecato per memorizzare l'albero
2. Usiamo la ricorsione: buona per compilatore ma dovrebbe essere evitata negli OS. Soluzione: L'albero può essere immagazzinato in un bitmap

Interrupt:

Che cos'è un interrupt e come funziona:
nei sistemi moderni ogni interazione con l'OS è triggerata da un interrupt. Un interrupt può nascere da eventi esterni (timer, I/O), eccezioni interne (istruzioni illegali) o chiamate esplicite (syscall). Senza gli interrupt, sarebbe necessario fare Polling, cioè controllare continuamente lo stato di un certo registro.

Quando avviene un interrupt il contesto corrente viene salvato e la CPU cerca nell'Interrupt Vector (un array di puntatori a funzioni), qual è la ISR da eseguire. La ISR fa parte dell'OS e una volta entrati dentro si entra in Supervisor Mode e l'OS si occupa di gestire l'evento.

Interrupt ID	ISR pointer
0 (es. reset)	ADR 0
1 (es. serial)	ADR 1
2 (es. TRAP)	ADR 2
3 (...)	ADR 3
….....	……...

In genere un singolo interrupt fisico gestisce più devices (USB, …), quindi è compito dell'ISR trovare poi qual è il device che ha generato l'interrupt.

Le eccezioni sono interrupt triggerate a livello di software. Si dividono in:

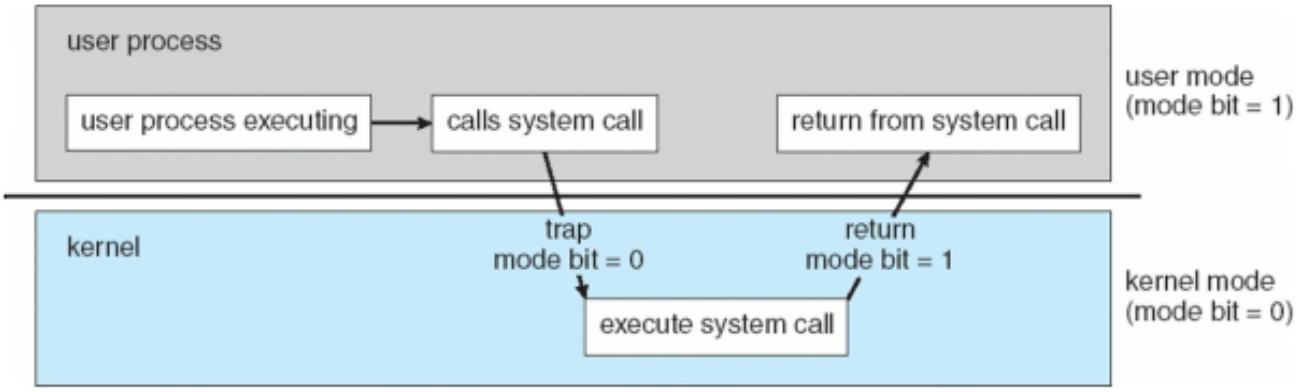
- Trap: ISR invocata dopo l'istruzione triggerante. Es. INT
  INT <XX> salta all'ISR il cui indirizzo è memorizzato in posizione <XX> dell'IV. C'è un numero limitato di entry dell'IV raggiungibili con una INT (altrimenti posso fare interrupt grevi tipo bloccare il disco con un'istruzione software. CALL <YY> invece è una call a una subroutine il cui indirizzo è <YY> ed è mappato nell'area di memoria del processo. Non richiede l'entrata in kernel mode se si è in user mode.
- Fault: ISR invocata prima dell'istruzione triggerante. Illegal instruction
- Abort: quando lo stato del processo per cui è scattata l'interrupt non può essere ristorato. Es. Double Fault Exception

Che cos'è una syscall e come si esegue:
Una syscall è una chiamata a sistema che il livello utente utilizza per richiedere un servizio a livello kernel. Ad esempio l'alterazione dell'IV può essere fatta solo in modalità kernel oppure la scrittura in memoria mappata su device (cioè le porte I/O dove per es. sta leggendo ethernet).

Dopo una chiamata a sistema si entra in kernel mode (mode bit in FLAGS della CPU viene settato di conseguenza). Per chiamare una syscall facciamo INT 0x80, dove 0x80 è l'indice dell'IV dove è memorizzato l'indirizzo della syscall. Si entra quindi in modalità privilegiata e si va a cercare nel vettore di puntatori a funzioni syscall mappate nello spazio kernel qual è la syscall invocata (l'indice dell'array alla quale questa è memorizzata è stato inserito in %eax dall'utente) e la si esegue. Ulteriori parametri, oltre all'indice della syscall nell'array, sono messi dall'utente nei registri della CPU e nella stack. In questo modo c'è una sola entry dell'IV dietro cui si nasconde l'OS con la sua modalità privilegiata. Questo meccanismo

permette di avere una sola “porta” da cui l’utente può passare, riducendo la vulnerabilità dell’OS.

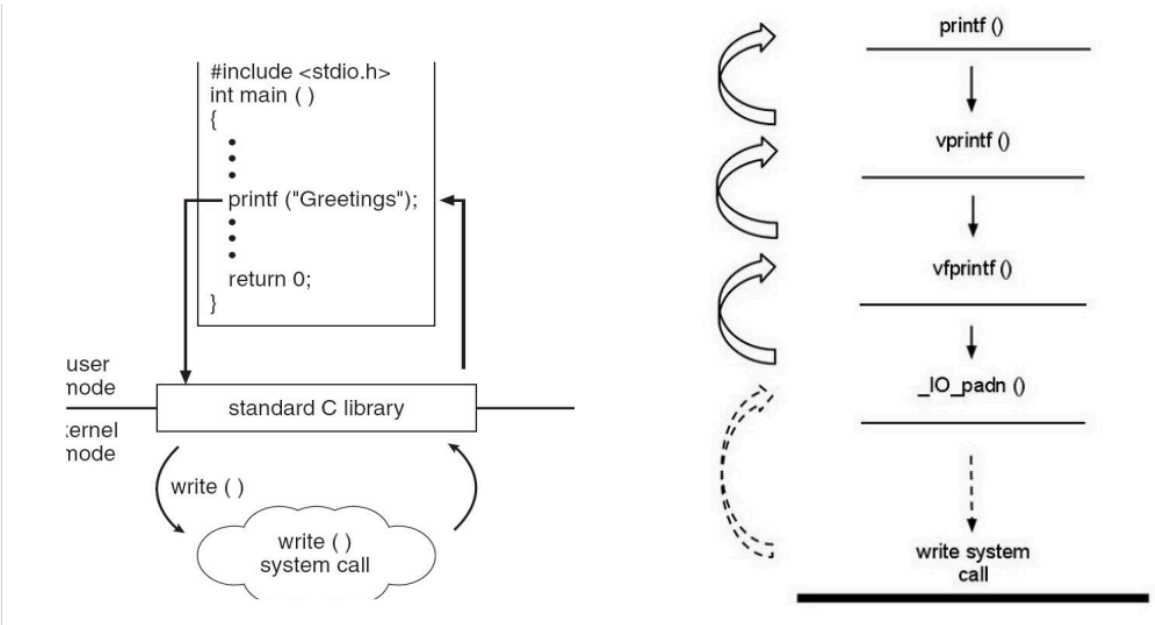
Come si ritorna in User Mode? Con IRET che ristora i flag.



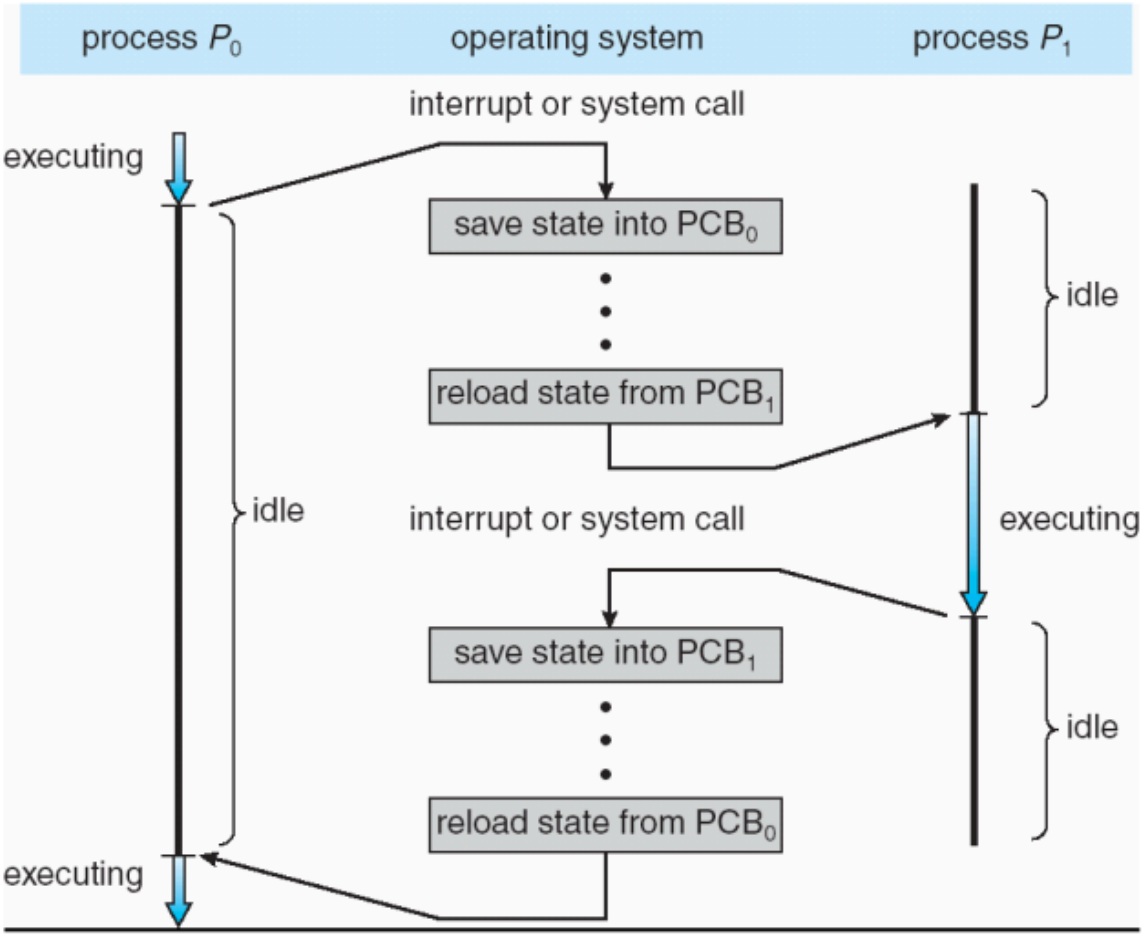
Le syscall servono per:

- gestire i processi
- gestire i file
- gestire i device
- comunicazione, connessioni e trasferimento messaggi

Il codice scritto a livello di syscall non è portabile in quanto è a basso livello e dipende fortemente dall’OS. Per garantire portabilità si possono utilizzare delle librerie standard (es. libc per C o POSIX per funzionalità più complesse come i thread) che hanno funzioni ad alto livello e che sotto chiamano le specifiche syscall del sistema.



Il context switch avviene in kernel mode. Infatti i PCB sono memorizzati in zona di memoria privilegiata. Un context switch avviene quando c'è un interrupt.

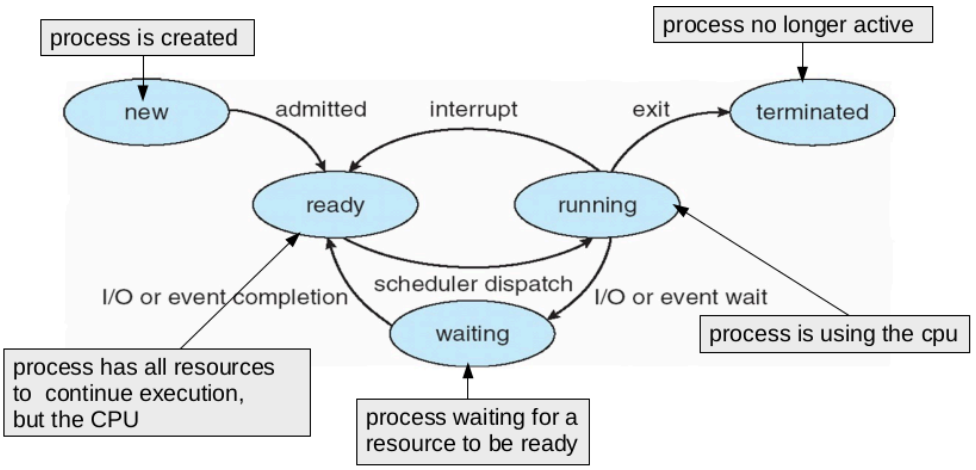


Processi:

Che cos'è un processo e come viene eseguito:

Un processo è un programma in esecuzione caratterizzato da:

- I registri della CPU
- la sua memoria
    a. il codice che esegue in .text
    b. stack
    c. heap
    d. .bss per variabili globali non inizializzate, riempite di zeri
    e. .data per variabili globali inizializzate
- Risorse: file/socket descriptor e costrutti di sincronizzazione (semafori, code…)
- il suo stato:



Esistono le seguenti syscall:

a. fork(): quando un processo chiama la fork() viene creato un processo “figlio” che ha una “copia” della memoria del padre. Anche le risorse (file descriptor ecc.) sono copiate. Precisamente, si tratta di COW: finché non avviene una modifica, la memoria del figlio è esattamente quella del padre. Il valore di ritorno della fork() è 0 per il processo figlio, mentre per il padre è il PID del figlio.

   Quando un figlio muore l’OS manda un segnale SIGCHLD al padre, mentre quando un padre muore tutti i figli ricevono SIGHUP

   Se il padre termina prima del figlio l’orfano diventa figlio diretto di init, il processo da cui parte l’OS

b. wait(): blocca l’esecuzione del processo finché un suo figlio non termina.
c. exit(retval): istruzione con cui il processo termina la sua esecuzione. Il processo rimane zombie (stato terminated) finchè il padre non legge retval con wait(). Dopodiché il processo smette di esistere. init fa wait() periodicamente per eliminare gli orfani.
d. exec(): rimpiazza la memoria di un processo con uno nuovo caricato da un program file. È possibile passare i parametri del main del processo e variabili d’ambiente con execv(argv) e execve(argv, char** environ)
e. vfork(): se vogliamo fare exec() subito dopo fork(). Infatti vfork() non copia la memoria del padre, che se vogliamo fare exec() subito dopo la fork() è uno spreco dato che questa memoria va buttata. Importante però è fare exec() subito dopo vfork().

Stack Frame:
Un processo, per ogni attivazione di funzione che esegue mantiene uno stack frame (o record di attivazione) che contiene gli argomenti passati, l’indirizzo di ritorno della funzione, variabili locali…

Come vengono schedulati i processi?

I processi vengono schedulati con uno scheduler. Uno scheduler sceglie il prossimo processo che riceverà tempo CPU. Uno scheduler usa delle linked list di PCB. Ci sono due tipi di scheduler di processi:

- Non preemptive (batch): a un processo in running non può essere tolto tempo CPU a meno che non faccia I/O request o termini.
- Preemptive: un processo in running può essere messo in coda running dopo aver consumato il suo quanto di tempo CPU

Quando il dispatcher esegue il context switch si invalida la cache.

I processi a user level sono caratterizzati da alternanza di

- CPU Burst: quanto di tempo in cui viene usata la CPU mentre I/O riposa.
- I/O Burst: tempo in cui viene usato l'I/O mentre la CPU riposa. Poiché il processo sta aspettando l'I/O e non usa la CPU, questa può essere data a un altro processo.

Ci sono alcune metriche utilizzabili per decidere come schedulare i processi:

- CPU utilization: quanto di tempo in cui la CPU è usata dai processi
- Turnaround time: tempo per completare l'esecuzione di un processo
- Throughput: numero di PROCESSI che completano la loro esecuzione nell'unità di tempo
- Waiting time: da quanto un processo è in ready queue
- Response time: quanto tempo serve a un processo che riceve un comando per dare la risposta

Prime due da massimizzare, ultime tre da minimizzare. In base all'applicazione alcune metriche sono più importanti di altre (esempio: per un editor il tempo di risposta è il più importante).

Per uno scheduler un processo non è nient'altro che:

- Tempo di Arrivo
- Lista di azioni con loro durata: CPU Burst e I/O burst.

In questo modo il comportamento di uno scheduler può essere illustrato su un diagramma con il tempo sull'asse X, una riga per CPU e una per I/O

Tipi di Scheduler:

- FCFS (First Come First Served): non preemptive, prende il primo processo che è in ready. Dopo I/O burst passa da waiting a ready, cioè finisce in fondo alla coda. Questo scheduler ha il problema del convoy effect. MA QUINDI NON PUÒ LEVARE IL TEMPO CPU A UN PROCESSO CHE STA IN WAITING? DEVE PER FORZA ASPETTARE CHE FINISCA L'IO??? SOPRA C'È SCRITTO CHE UN NON PREEMPTIVE PUÒ TOGLIERE CPU QUANDO FA I/O REQUEST, CONTROLLA

- SJF (Shortest Job First): non preemptive, evita il convoy effect, permettendo di abbassare i tempi di attesa e aumentare il th schedulando prima i processi il cui CPU burst è più basso. Il problema è che si devono conoscere i comportamenti dei processi (CPU e I/O burst) in anticipo. Poiché lo stesso processo ha tipicamente un comportamento ciclico, è comunque possibile STIMARE (non conoscere) CPU e I/O burst di un processo. Stimiamo il prox CPU burst facendo la media esponenziale delle lunghezze dei CPU burst precedenti: sia t_n la lunghezza dell'n-esimo burst di CPU e tau_n il valore predetto della lunghezza del burst n-esimo, allora la lunghezza predetta del CPU burst n+1-esimo è

  tau_{n+1} = alpha*t_n + (1 - alpha)tau_n

  con alpha coefficiente di decadimento tra 0 e 1

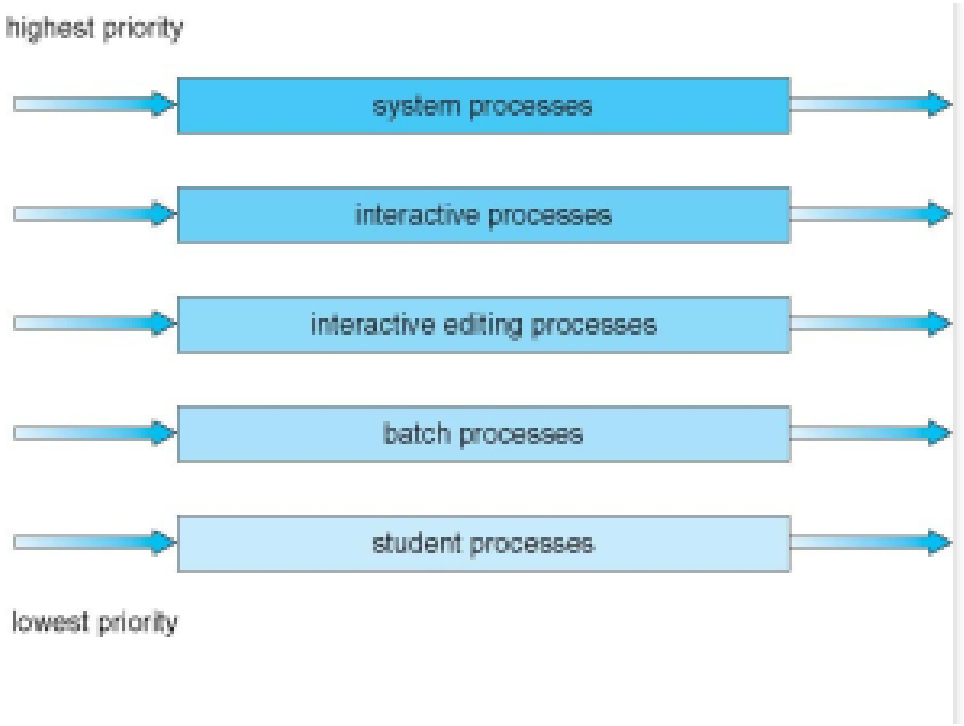
  in questo modo possiamo smoothare eventuali spike nei burst, che altrimenti degraderebbero la qualità della predizione per tutta la storia futura. Infatti se in generale ci si mantiene intorno a un certo valore costante ma a un tempo t c'è uno spike, se alpha è minore di 1 / 2 si fa contare di più la storia passata (tau_n) che lo spike attuale (t_n). Nel caso invece di alpha > 1 / 2 si fa pesare di più l'esperienza corrente che il passato, nel caso in cui ad esempio si consideri il passato come vetusto e non più utile.

- Priority Scheduler: non preemptive, ai processi è assegnato un intero. Se il loro intero è minore viene eseguito prima. SJF è un priority scheduler se la priorità è la lunghezza del prox CPU burst. Il problema della starvation può essere risolto con aging: più un processo passa tempo nella ready queue più aumenta la sua priorità.

- Preemptive Scheduler: tutti gli schemi precedenti si possono estendere a preemptive. Preemptive è essenzialmente uno scheduler se può mettere in ready un processo running che però non ha ancora richiesto un I/O. si assegna un quanto di tempo CPU ad ogni processo dopo di cui scatta un timer e viene messo in ready il processo. Con un RR (round robin) dopo essere stato messo in ready il processo va alla fine della coda. In questo modo nessun processo aspetta più di (N-1)*q, dove q è il quanto e N è il numero di processi. q deve essere più grande del context switch time, altrimenti si passa più tempo a switchare che ad eseguire processi.

  È possibile avere più code ready e un algoritmo di scheduling per coda. Ogni coda ha una priorità diversa. Ad esempio, si può avere una coda per processi in foreground che utilizza RR e una coda per processi in background che usa FCFS. , in questo modo è possibile anche massimizzare metriche diverse su code diverse grazie a diversi scheduler. Lo scheduler poi può scegliere processi con:
  a. Fixed Priority Scheduling: prima tutti i processi nella coda con più priorità (foreground), poi quelli con priorità minore (background). C'è il problema della starvation.
  b. Time Slice: ogni coda riceve una percentuale di tempo CPU. Es. 80% foreground, 20% background.

Si possono avere processi sempre nella stessa coda oppure Multilevel Feedback Queue in cui un processo può cambiare coda (implementazione dell'aging). Ad esempio un metodo per cui promuovere/declassare un processo può essere: se consuma costantemente tutto il suo quanto viene declassato, se consuma costantemente meno del suo quanto viene promosso. In questo modo, un processo che ha CPU burst più lunghi ha una priorità minore perché è in code più basse.



Multiprocessor Scheduling:

Asymmetric: solo un core entra in kernel mode, quindi non ho bisogno di sistemi di lock

Symmetric: ogni processore può entrare in kernel mode, quindi ho bisogno di lock.

Processor Affinity: un processo ha affinità con il processore su cui sta runnando. Legare un processo a un processore in modo che questo venga eseguito solo sulla CPU scelta può avere dei vantaggi. Soft affinity se l'OS cerca di mantenere il processo sullo stesso processore ma non lo garantisce (es. per load balancing), mentre hard affinity permette al processo di specificare un sottoinsieme di CPU sulle quali può runnare e su nessun'altra potrà runnare.

Real Time Scheduler:

soft se non c'è garanzia sullo scheduling dei task critici. Lo scheduler deve essere preemptive e priority-based.

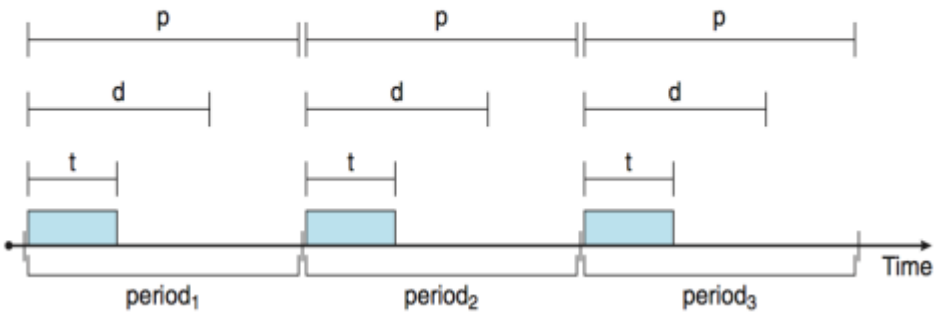
hard se sono i task critici vengono per forza eseguiti nella loro scadenza. Lo scheduler deve essere preemptive, priority-based e deve sempre rispettare le deadline. Precisamente:

t è il tempo di esecuzione del processo, d è la deadline e p è il periodo ogni quanto il processo parte.

- $0 \leq t \leq d \leq p$
- **Rate** of periodic task is $1/p$

$$U = \sum_i \frac{t_i}{d_i} < 1$$

this tells if I can schedule a set of processes in EDF



Se U (CPU utilizzata) non è < 1 non è possibile schedulare quei processi scelti in real time.

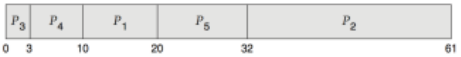
Come Valutare uno Scheduler:

- Deterministic modeling: scelto un determinato workload si calcolano a mano le performance di ogni algoritmo per quel workload

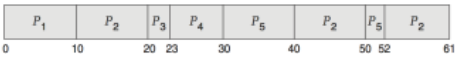
Process	Burst Time
P_1	10
P_2	29
P_3	3
P_4	7
P_5	12

- FCS is 28ms:

- Non-preemptive SFJ is 13ms:



- RR is 23ms:



- Queuing Models: **Legge di Little** n = lambda * W con n dimensione coda processi, lambda frequenza di arrivo media e W tempo di attesa medio.

- Simulazione: implementiamo un modello “fake” del sistema che mentre gira ci permette di raccogliere statistiche sulle performance dei vari algoritmi di scheduling

```
// un evento è o un CPU burst o un I/O burst, con la sua durata.
ProcessEvent {
        ListItem list;
        ResourceType type;  // enum di CPU e I/O burst
        int duration;
}
```

```
// un processo viene rappresentato come un PID, un tempo di arrivo e una lista di eventi (CPU/IO burst). Sarà poi sicuramente inserito in una lista di processi
FakeProcess {
        ListItem list;
        int pid; // assigned by us
        int arrival_time;
        ListHead events;
}
```

Corrispondentemente a ogni processo, abbiamo

<id processo>, <arrival_time>

CPU Burst, <duration>

I/0 Burst, <duration>

…

così da poter aggiungere i processi scrivendoli nel file e poi traducendoli in struct.

Poi implementiamo il vero e proprio FakeOS:

```
FakeOS {
        FakePCB* running
        ListHead ready
        ListHead waiting
        int timer
        scheduleFn schedule_fn //puntatore alla funzione che fa da scheduler
        void* schedule_args
        ListHead processes //tutti i processi sono qua all'inizio
}

FakePCB {
        ListItem list
        int pid
        ListHead events
}
```

FakeOS_simStep() fa un giro di giostra: aumenta di 1 il timer e se serve crea nuovi processi (os.timer == processo.arrival_time) o li schedula

Main Memory:

La CPU può accedere direttamente solo a registri e Memoria Centrale. I registri sono acceduti in al massimo un ciclo di clock, la cache ci mette diversi cicli di clock, mentre la main memory ci può mettere tanti cicli di clock.

Caricare un programma per eseguirlo:

quando facciamo exec prendiamo un program file ELF (file eseguibile) e lo trasformiamo in un'immagine di memoria. Il file ELF che contiene il programma è preso dalla memoria di massa.

dall'header dell'ELF estraiamo informazioni:

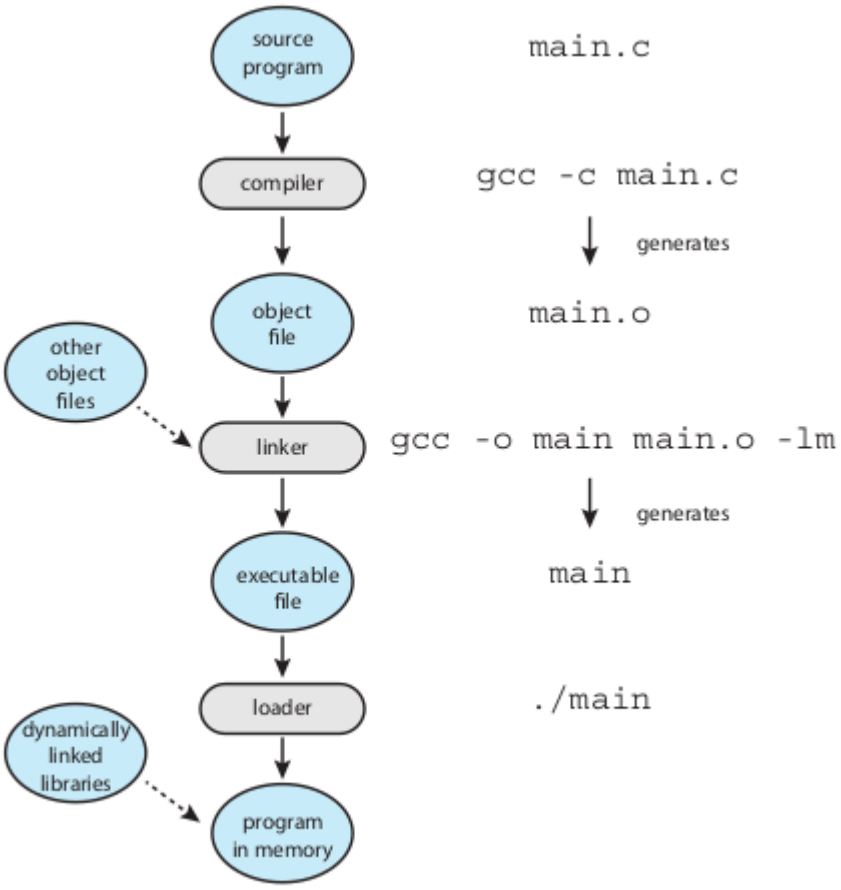
- segmenti (.text, .bss, .data) necessari per caratterizzare l'immagine di memoria
- tabella dei simboli: contiene i simboli (nomi di variabili, funzioni, costanti…) definiti o utilizzati nel codice sorgente del programma
- librerie dinamiche: caricata dinamicamente in fase di esecuzione (a differenza delle librerie statiche che sono integrate nell'eseguibile del programma), invece di essere collegata staticamente a un eseguibile in fase di compilazione. Inoltre se più programmi usano la stessa libreria dinamica questa in memoria c'è una sola volta e i programmi la linkano. Le librerie dinamiche vengono mappate nello spazio tra lo stack e l'heap. Il dynamic linker permette di linkare nel programma le librerie dinamiche:
    a. Carica le librerie dinamiche dai loro program files e mappale nel program space del processo
    b. Aggiusta i puntatori appesi ai simboli definiti nelle librerie in modo da poter essere usati nel programma

Ci sono tre modi di fare ***address binding***, cioè aggiustare gli indirizzi di istruzioni e dati dal file mettendoli in memoria in maniera tale che il codice possa essere eseguito correttamente:

- Tempo di Compilazione: il bootloader, il bios e il kernel sono codici assoluti, perché quando partono stanno là, quindi la loro locazione di memoria è nota a priori
- Tempo di Caricamento: codice rilocabile. C'è un monoblocco con tutti riferimenti relativi al suo interno, questo significa che il monoblocco può essere spostato tutto insieme dove ti pare, ma deve essere tutto insieme e non può né cambiare dimensione né essere spostato solo parzialmente.
- Tempo di Esecuzione: il binding è fatto a tempo di run time dato che il processo può cambiare locazione di memoria durante la sua esecuzione

Fasi di un programma:

Un file sorgente viene compilato in un file oggetto che può essere caricato su memoria fisica. Viene quindi linkato ad altri file oggetti anch'essi caricabili su memoria fisica (relocatable object file) ottenendo dal linking un eseguibile che poi viene caricato in memoria dal loader. La CPU adesso può eseguirlo e a tempo di esecuzione verranno linkate le librerie dinamiche dal dynamic linker



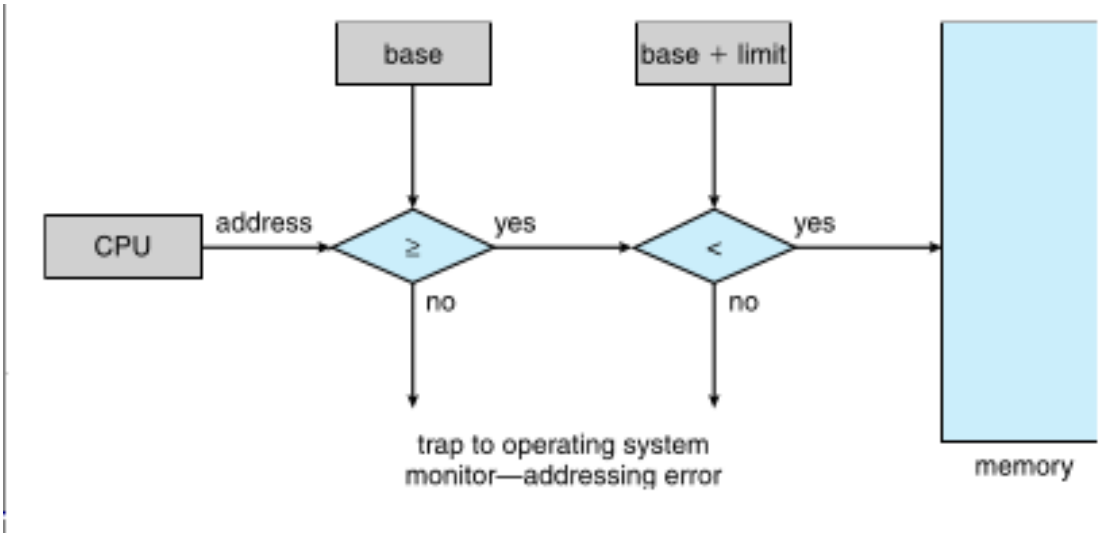
Nel file oggetto abbiamo delle label che vengono poi riempite con indirizzi veri della memoria a tempo di caricamento.

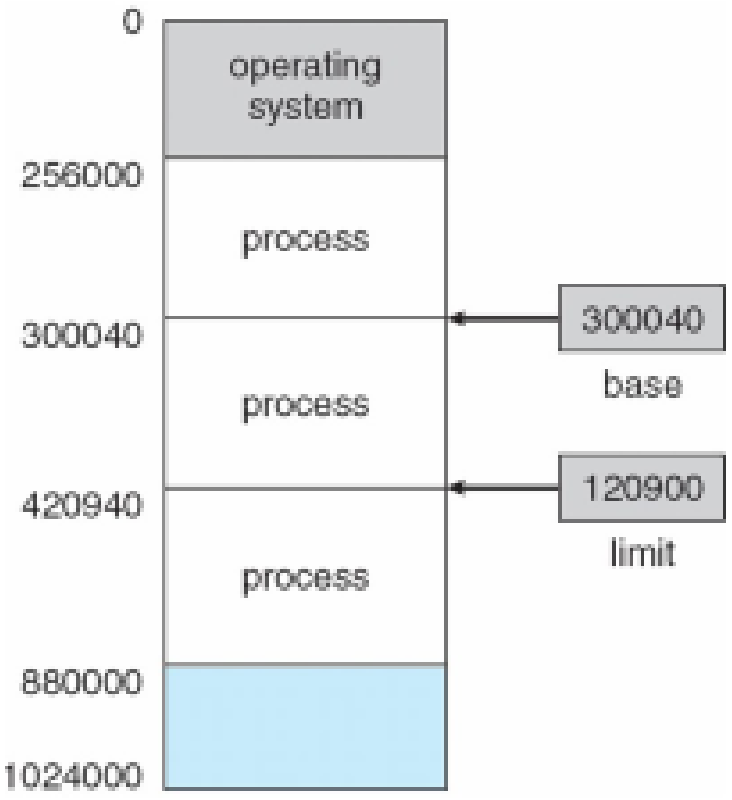
Allocazione Memoria:

Dobbiamo partizionare la memoria per darla ai processi. Dobbiamo stare attenti a:

- Il fatto che la memoria richiesta da un processo può cambiare durante la sua vita per le librerie dinamiche e per l'heap
- Frammentazione
- Protezione: in generale non vogliamo che un processo scriva/legga memoria che non è sua o esegua codice non eseguibile

Come fare protezione:
inizialmente il codice era solo assoluto e si faceva così: un registro *base* dava il limite inferiore per gli indirizzi della memoria del processo, mentre *base* + *limit* dava il limite superiore. Controllando se gli indirizzi erano contenuti in questo range, era possibile lanciare un interrupt in caso si facesse segmentation fault. Nel caso di context switch si cambiavano i valori dei due registri.

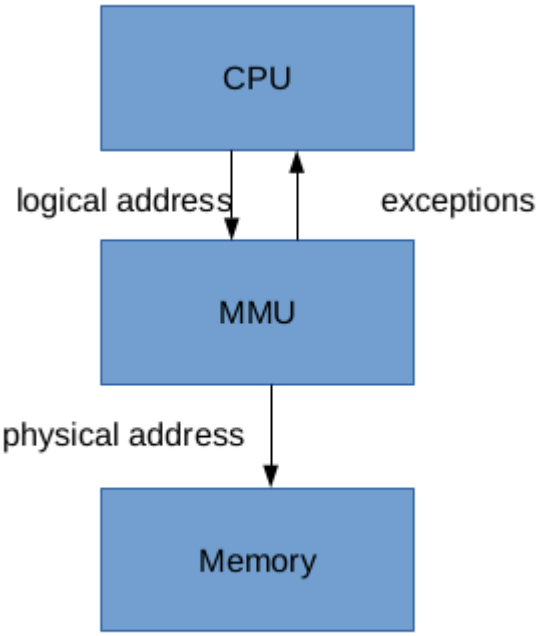




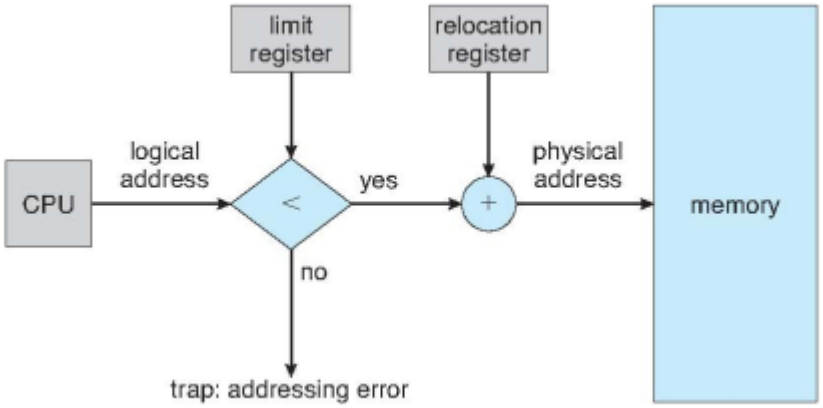
poi si è iniziato a parlare di indirizzi logici e fisici e quindi di MMU.

- Indirizzi logici: generati dalla CPU
- Indirizzi fisici: visti dalla RAM

La differenza tra indirizzi logici e fisici esiste solo a tempo di esecuzione. La Memory Management Unit è un pezzo di hardware che si occupa di mappare gli indirizzi logici in fisici a tempo di esecuzione. La CPU conosce solo gli indirizzi logici e può configurare la MMU, ma non conosce gli indirizzi fisici della RAM.



Le prime MMU erano così:



la CPU ancora non genera codice relativo ma solo assoluto. Quando la CPU genera un indirizzo logico, per ottenere quello fisico innanzitutto verifico come nel sistema descritto prima se esco dai bound, se non esco c'è un *relocation register* che contiene un offset e genero l'indirizzo fisico come offset del relocation + indirizzo logico

La MMU ha due gradi di libertà: dove comincia lo spazio di memoria del processo e dove finisce, non posso metterci un buco in mezzo né mettere un pezzo della memoria da un'altra parte, quindi supporta solo *l'allocazione contigua*. Il problema dell'allocazione contigua è la frammentazione. Vediamo più in dettaglio.

Multiple Partition Allocation: dividiamo la memoria in blocchi e diamo i blocchi ai processi.

- Dimensione blocchi fissa: (PAGINAZIONE)
    a. Frammentazione interna (all'interno di un processo sprechi la memoria): processi piccoli che hanno bisogno di poca memoria si prendono comunque tutto il blocco.

    b. Numero limitato di processi
- Dimensione blocchi variabile: (SEGMENTAZIONE)
    a. Frammentazione esterna (abbastanza memoria per allocare un processo ma non contigua)

Possiamo assegnare un blocco in tre modi:

- First-Fit: primo blocco che trovo lo do al processo
- Best-Fit: diamo il blocco più piccolo
- Worst-Fit: diamo il blocco più grande.

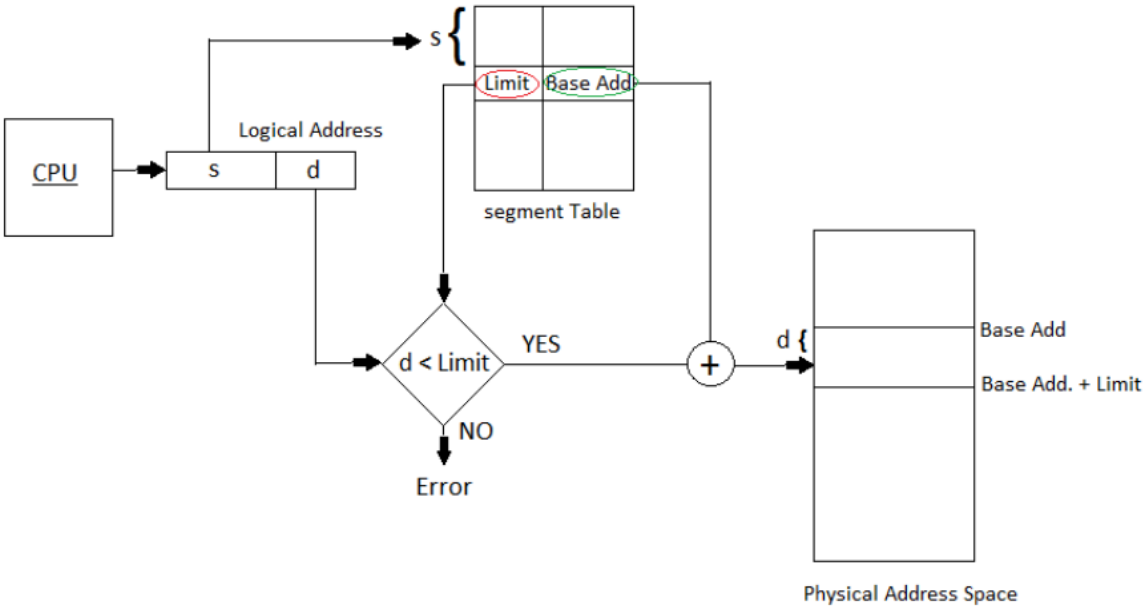
Per ridurre la frammentazione esterna si fa ***compaction***. Risistema la disposizione dei blocchi di memoria per mettere i free block tutti contigui. Si può fare solo se c'è relocazione degli indirizzi dinamici a tempo di esecuzione. Costa molto perché devi spostare tutta la memoria. Inoltre quando si fa I/O non possiamo spostare la memoria se si sta scrivendo/leggendo da un blocco. Come risolvere questo ultimo punto? Si può risolvere imponendo che si scriva soltanto nello spazio utente del kernel e non del processo (double buffering). Quando il processo è pronto riceve la copia dei suoi dati dal kernel.

Dopodiché le MMU si sono evolute: segmentazione. Prima si usavano 4 registri di segmentazione che permettevano di raggiungere il segmento dello stack, il segmento del .text ecc.

Non c'era alcun meccanismo di sicurezza, quindi si overlappavano i segmenti.

Poi hanno deciso di dividere gli indirizzi logici in:

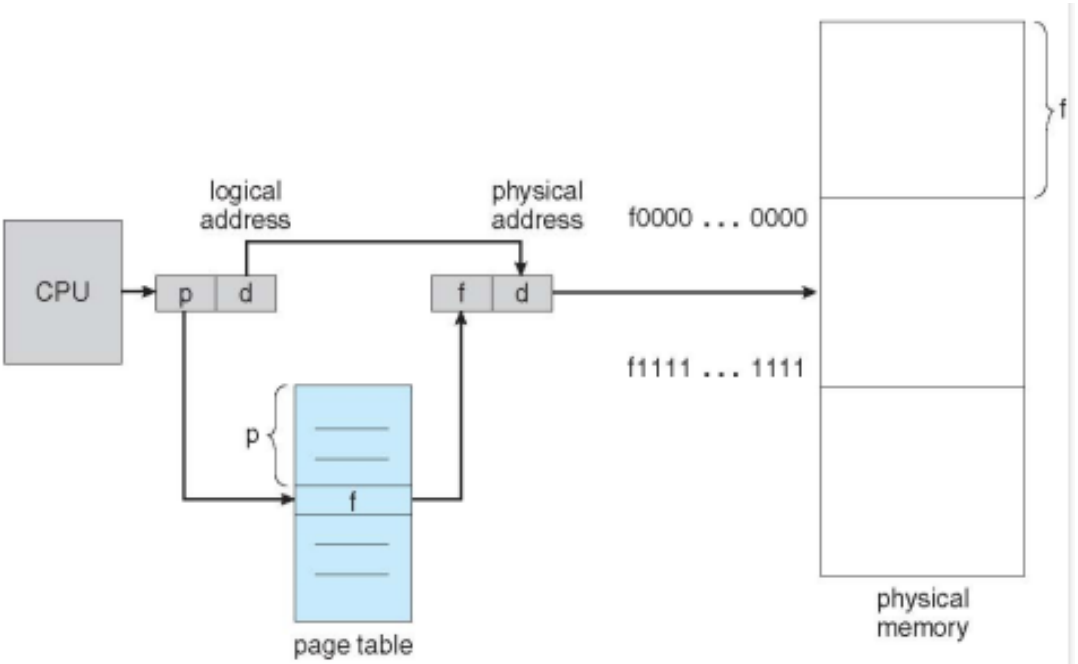
- Numero di segmento s: indice della segment table dove trovare limit e base address per quel segmento
- Offset d: offset dell'oggetto all'interno del segmento.



Abbiamo bisogno di due registri:

- Segment Table Base Register che punta alla segment table
- Segment Table Lenght Register che dà il numero di segmenti del programma

Quello che si fa oggi però è la *paginazione*: l'indirizzo logico si divide in page number e page offset. Quindi abbiamo 2^p pagine ognuna grossa 2^d bit. Dato un indirizzo di un oggetto p|d, vado nella tabella delle pagine a indice p, mi prendo l'indice f del frame di memoria fisica alla quale trovo l'oggetto e sostituisco f a p. Nel frame f al bit d-esimo troverò il mio oggetto. Ogni processo ha la sua tabella delle pagine.



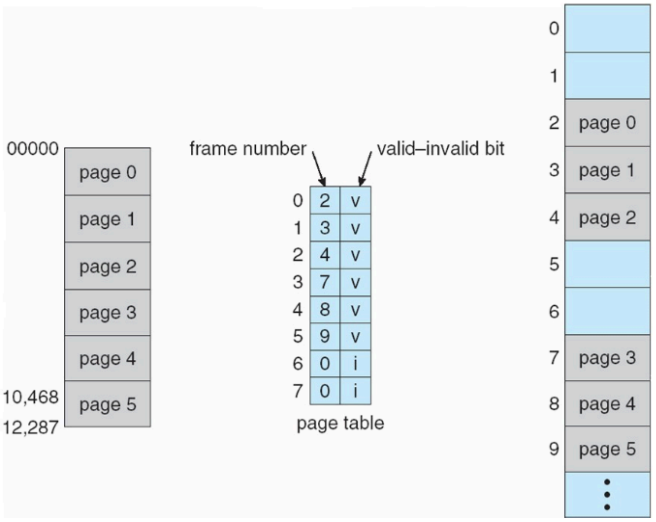
Abbiamo bisogno di due registri:

- Page Table Base Register che punta alla segment table
- Page Table Lenght Register che dà la dimensione della tabella delle pagine

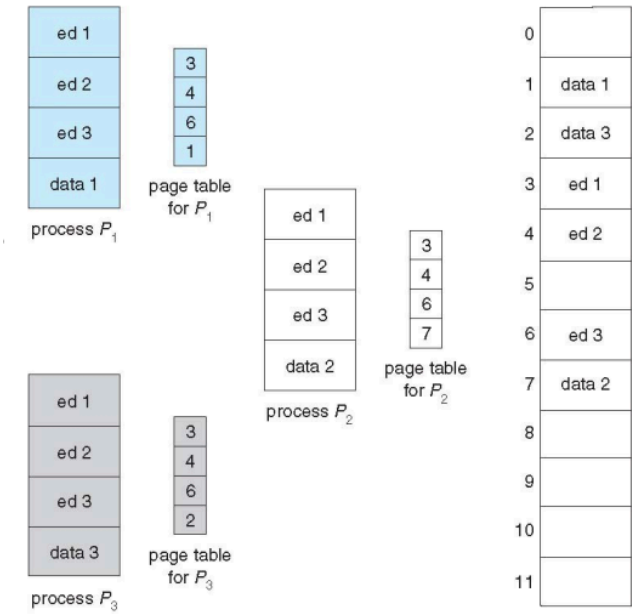
In questo modo ho risolto frammentazione esterna e non ho necessità di compaction: la memoria fisica ora può essere non contigua tanto i frame sono mappati sulle pagine che invece sono senza problemi contigui. L'importante è che le pagine siano contigue.

Devo però accedere due volte alla RAM, una volta alla tabella delle pagine e una volta alla memoria per prendere l'oggetto richiesto. Nella CPU allora mettiamo una cache in cui mettiamo le ultime entry della tabella delle pagine (TLB).

Per proteggere le pagine posso associare dei bit di protezione ad ogni entry nella tabella delle pagine. I bit indicano se il frame corrispondente è read-only, read-write, eseguibile, "valid" (cioè se la pagina corrispondente è mappata nella memoria logica del processo), invalid (la pagina non è mappata nella memoria logica del processo), …



Le shared pages, come le librerie, sono codice read-only di cui si fa una sola copia in memoria fisica (quindi memorizzati in un singolo frame) e vengono condivise tra tutti i processi (quindi da un singolo frame hai tante pagine quanti sono i processi che la usano). In figura ed 1, ed 2, ed 3 sono shared pages. I diversi processi che usano le pagine condivise si mappano dove vogliono loro nella loro memoria le pagine, quindi in generale il layout della tabella delle pagine per vari processi che usano shared pages sarà diverso.



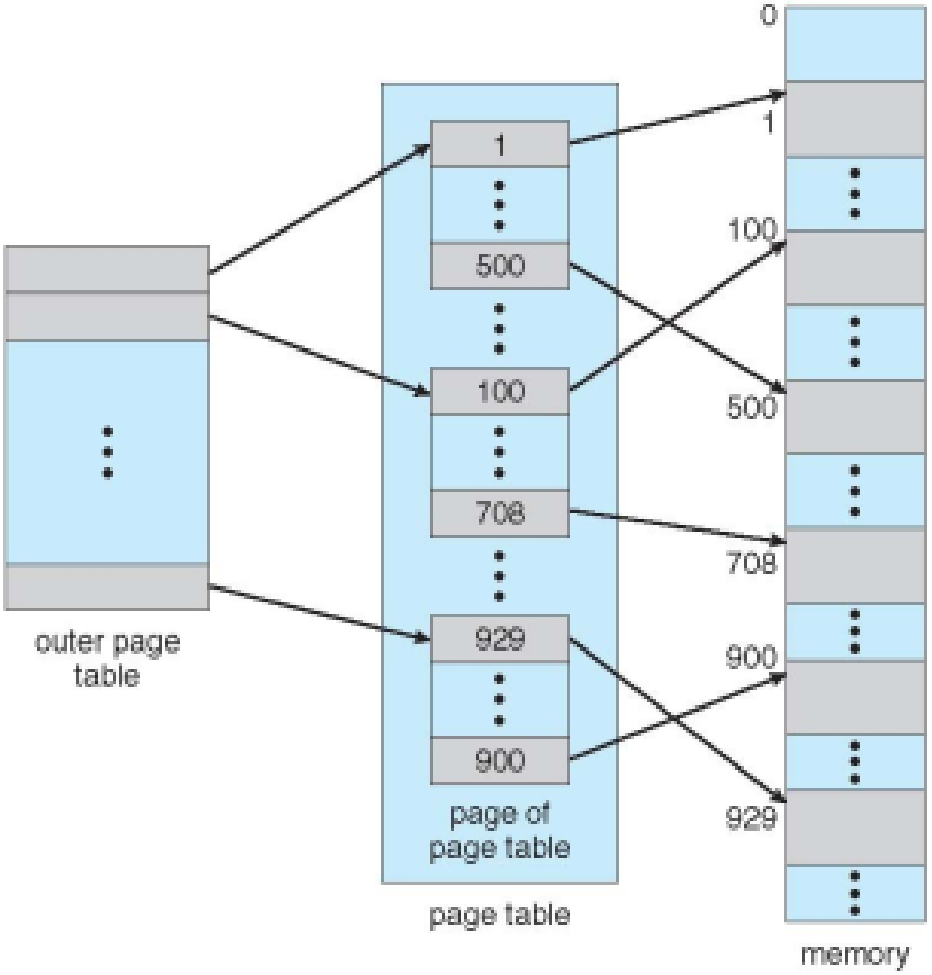
Effective Access Time (EAT): tempo atteso per accedere in memoria

EAT = alpha(T_ram + T_tlb) + (1 - alpha) 2 * (T_ram + T_tlb)

alpha è l'hit rate, cioè percentuale di volte in cui l'entry della table è in cache

Tabella delle Pagine Gerarchica:

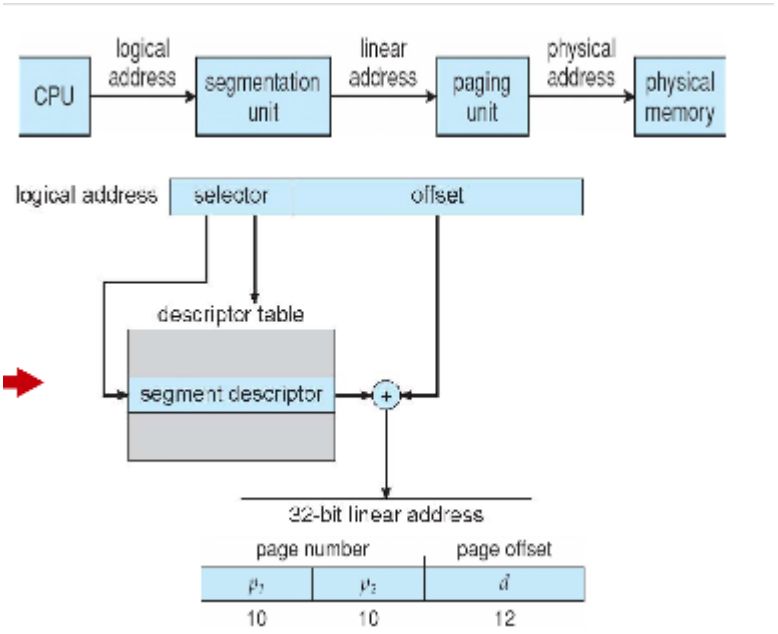
Ad esempio una tabella delle pagine a due livelli, in cui ogni entry della tabella delle pagine outer contiene l'indirizzo a una tabella delle pagine di secondo livello che contiene i soliti indirizzi f. Risparmi memoria perché invece di avere una mega tabellona da 1 a 900 hai delle sottotabelle di secondo livello sono più piccole, quindi se te serve soltanto un blocchetto piccolo di pagine setti la prima entry della outer page table così vai da 1 a 500 e tutte le altre le metti a invalid.



Se hai un miss della cache a outer page table devi fare due giri di memoria perché devi caricare nel TLB l'entry della outer table ma anche l'entry della table di secondo livello. Quindi aumenta l'EAT.

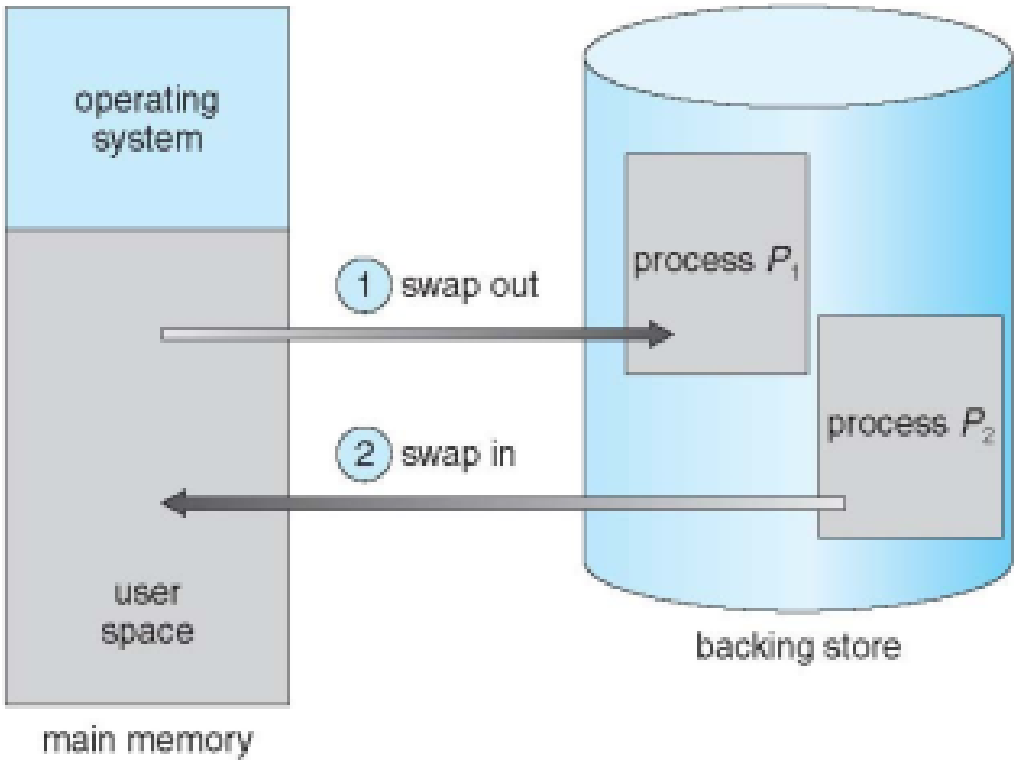
Approccio Ibrido: sia segmentazione che paginazione. L'indirizzo logico è diviso in tre parti: seg_num + offset_tabella_pagine + d.

Usiamo seg_num per trovare il segmento che contiene base e limit. Verificihiamo offset_tabella_pagine < limit e se è vero p = base + offset_tabella_pagine. Quindi abbiamo un indirizzo lineare. A pagina p troviamo f e quindi l'indirizzo fisico finale è f | d.



Swapping:

Spostare il processo con tutte le sue pagine di memoria dalla RAM sul disco.



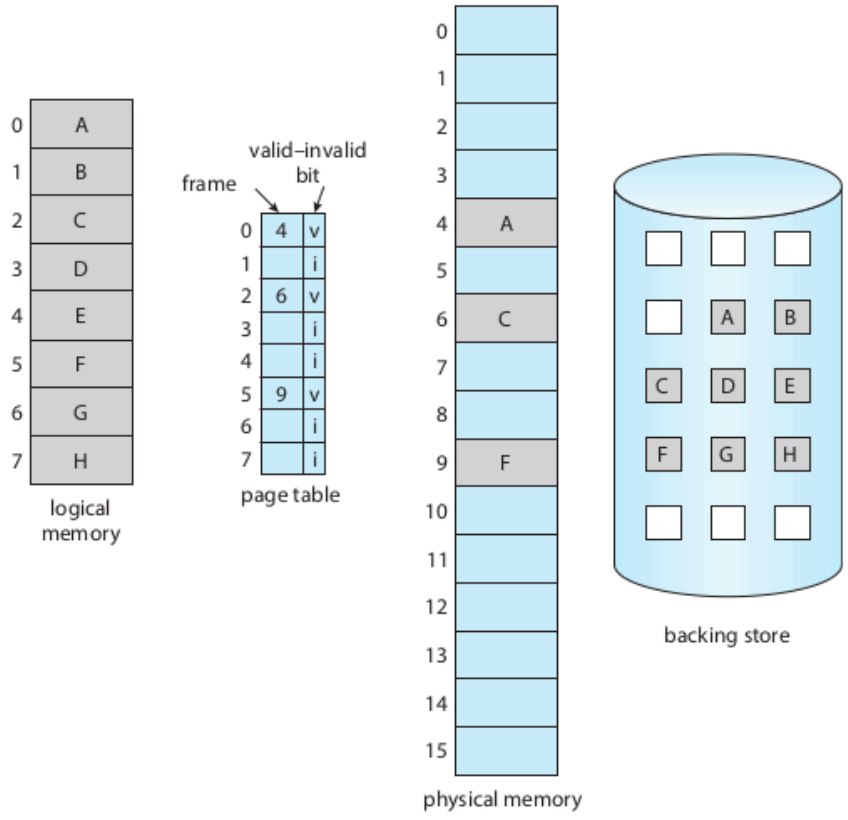
Quando c'è un I/O in corso su dei frame di memoria, se faccio swap out del processo che mappa questi frame succede un casino, quindi o impongo che non si può fare oppure anche qui faccio *double buffering* quindi l'I/O lo faccio nello spazio del kernel e poi quando il processo è pronto riceve la copia dei suoi dati dal kernel.

Virtual Memory:

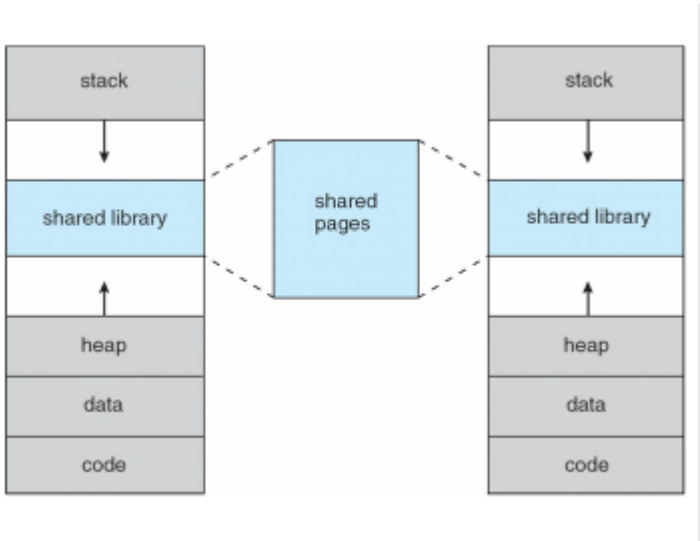
Raramente un programma viene eseguito interamente (esempio non uso tutte le funzionalità di Word quando lo uso), quindi posso caricare in RAM solo una parte del programma.

Per farlo usiamo la memoria virtuale: separa la memoria logica che l'utente vede da quella fisica della RAM. La virtual memory quindi permette di avere programmi che sono più grandi della memoria fisica, perché carico in memoria solo la parte che sto eseguendo e il resto lo metto su disco.

La memoria virtuale di un processo parte tipicamente dall'indirizzo 0 ed è contigua fino alla fine, quindi ogni processo vede alla sua memoria come fosse tutta e l'unica memoria del computer.



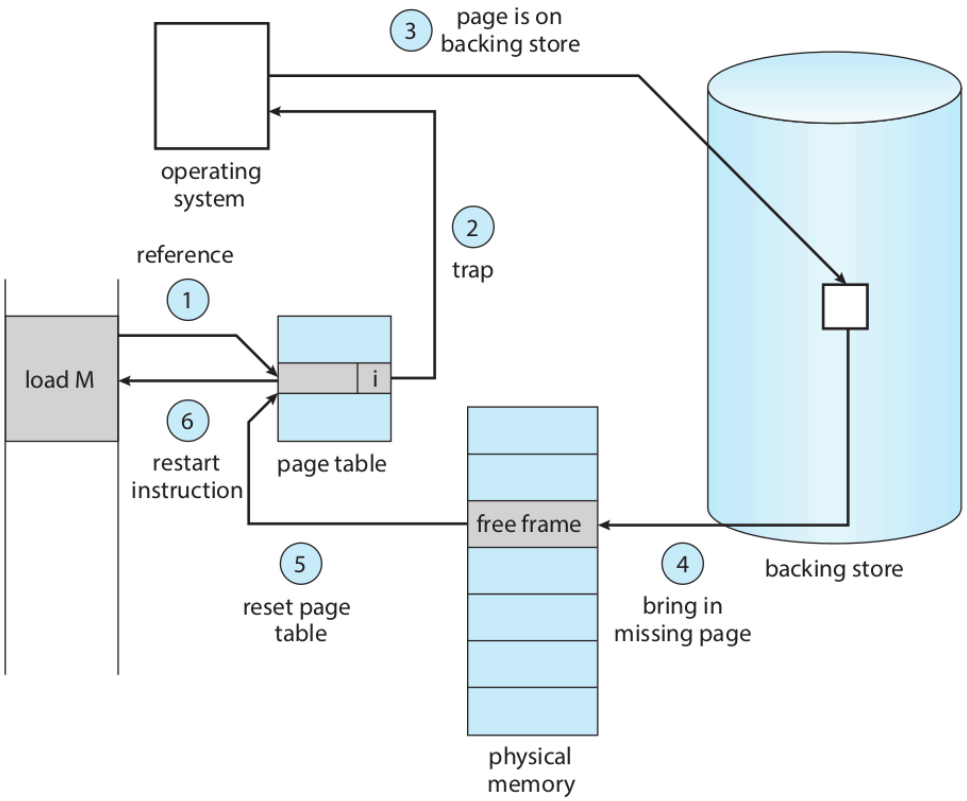
Le shared pages vengono mappate nel mare di mordor tra stack e heap



Ma come faccio swap in / swap out dei frame da RAM a disco? Possiamo caricare su RAM le pagine solo quando servono (tecnica ***demand paging***). Usiamo bit valid/invalid per capire se una pagina è attualmente caricata su RAM oppure no. Se il bit è invalid vuol dire o che la pagina non è mappata nello spazio logico del processo oppure che è mappata ma è su disco. Accedere a una pagina invalid causa un ***page fault***, Quando avviene un page fault:

1. Controlliamo in una internal table del processo se la pagina è invalid perché non è mappata, e in caso terminiamo il processo con segmentation fault
2. Altrimenti la pagina sta nel disco e la andiamo a caricare in RAM cercando un frame libero (se non c'è si swappa con quello meno utilizzato).
3. Caricata la pagina andiamo a modificare internal table e page table per indicare che la pagina è ora su RAM
4. Poiché si tratta di un fault, si riparte subito prima dell'istruzione che l'ha generato, ma stavolta la pagina è in memoria quindi il processo accede alla memoria correttamente.

Il caso estremo è quello in cui iniziamo un processo con 0 pagine in RAM (***pure demand paging***)



Data p la probabilità di fare un page fault, L'EAT per il demand paging è
EAT = (1 - p) * memory access + p * (page fault overhead + swap page in + memory access) = (1 - p) * memory access + p * (page fault overhead + swap page in)

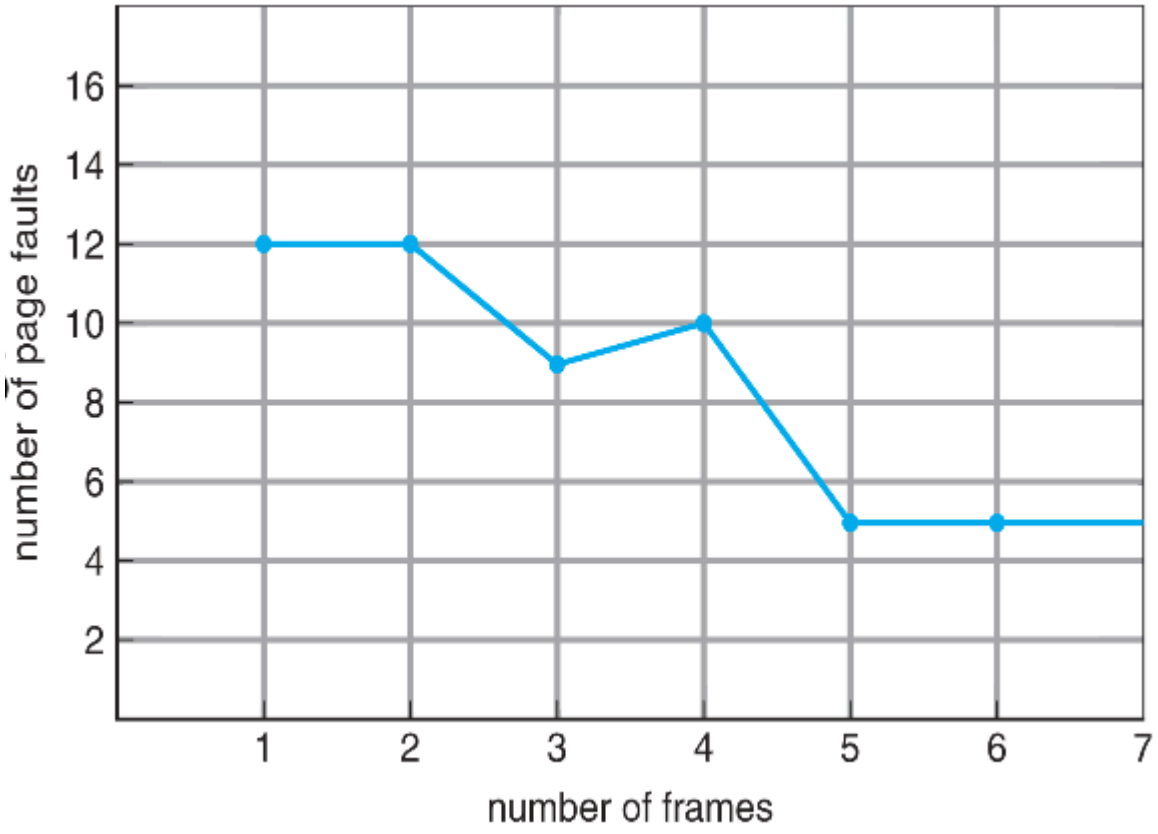
Ottimizzazioni del Demand Paging:

- Spazio di Swap su disco. Lo spazio di swap è in raw mode (senza file system), quindi non ha l'overhead del file system e l'I/O è quindi più veloce. All'inizio si può caricare tutta l'immagine del processo nello spazio di swap e all'occorrenza spostare le pagine su RAM.
- Copy on Write (COW): durante una fork() il processo figlio copia soltanto la tabella delle pagine del padre, mentre il resto della memoria punta alla memoria del padre. Nel momento in cui o il padre o il figlio modificano la loro memoria, avviene una copia della pagina di memoria del padre nello spazio del figlio, poiché ora le due pagine non sono più uguali.
- Free Frames: il sistema mantiene una lista di free frames per ottenere facilmente una pagina free

Nel caso in cui non ci siano frame liberi:
quando non ci sono frame liberi in RAM, bisogna scegliere un frame vittima da swappare su disco. Se ci accorgiamo che la vittima non è stata alterata in RAM, è possibile evitare la riscrittura su disco dei dati (invece di 2 page transfer ne faccio uno solo), poiché su disco c'è la sua esatta copia. Possiamo implementare questa meccanica con un modify bit che vale 1 se viene modificata la pagina.

Se sappiamo la sequenza delle pagine a cui il processo accede (<7, 0, 2, 1, …>) possiamo fare un algoritmo di pace replacement:

- FIFO: butto via la pagina che è stata swappata più tempo fa. Soffre dell'anomalia di Belady: offrire più frame a un processo può portare a più page fault



- Optimal Page Replacement: la scelta migliore è swapparlo con quello che verrà acceduto più tardi nel futuro. Implementabile perché deve prevedere il futuro ma è massimo teorico.
- LRU (Least Recently Used): swappare con la pagina che è stata acceduta più tempo fa. Potresti usare un contatore per ogni pagina, e quando accedi una pagina aggiorni il contatore col tempo corrente, poi quando devi fare lo swap vedi il tempo più vecchio. Oppure potresti usare una lista di pagine e ogni volta che una pagina è acceduta la muovi in cima. Tutti e due hanno troppo overhead quindi quello che si fa è che si fa è il second-chance algorithm.
- Approssimazione LRU: clock algorithm. Aggiungiamo un bit di riferimento nella tabella delle pagine ad ogni pagina. Inizialmente tutti i bit sono a 0, quando su una pagina si scrive o legge il suo bit viene messo a 1. In un certo intervallo di tempo, possiamo conoscere quindi quali pagine sono state accedute, seppur non sappiamo l'ordine. Abbiamo bisogno di supporto hw per il bit di riferimento e per poterlo modificare. Consideriamo la tabella delle pagine come un array circolare e manteniamo un puntatore alla prossima pagina da rimpiazzare.
  Second-Chance Algorithm (o Clock Algorithm): quando abbiamo bisogno di una pagina iniziamo a scorrere
  a. Quando una pagina viene caricata su RAM mettiamo il bit a 1 (era già a 1 se ci abbiamo letto o scritto)

b. Se il bit di riferimento della pagina corrente è 0 swappiamola
c. Se il bit è 1 diamo alla pagina una seconda chance: settiamo il bit a 0 e andiamo avanti

Quando tutti i bit sono settati a 1, Second-Chance degenera a FIFO replacement.

Enhanced Second-Chance Algorithm: utilizziamo bit di riferimento e il già citato modify bit. Abbiamo quindi quattro classi (riferimento, modify), in quest'ordine:

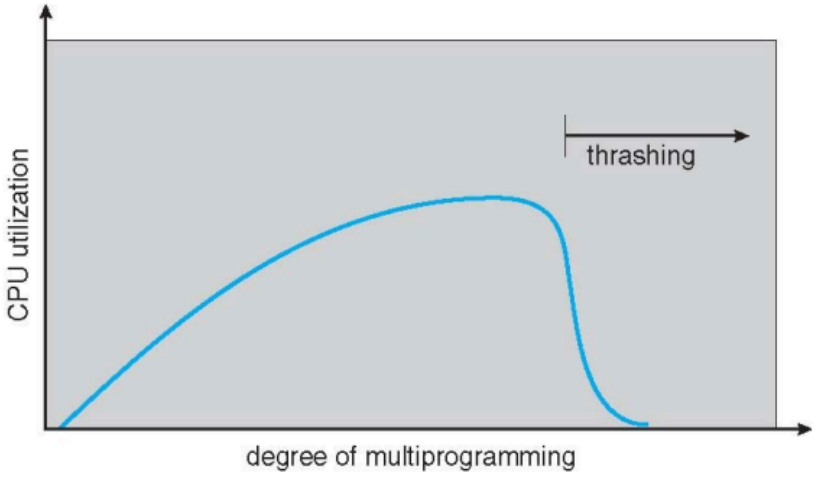
a. (0,0) non usata recentemente né modificata rispetto al disco
b. (0,1) non usata recentemente ma modificata rispetto al disco
c. (1,0) usata recentemente ma uguale alla copia su disco
d. (1,1) usata recentemente e modificata rispetto al disco

Quando cerchiamo una pagina, swappiamo con la pagina che ha classe più piccola

Se swappo una pagina con secondo bit a 0 posso buttarla senza ricopiarla su disco perché tanto è uguale alla versione su disco.

Trashing: quando un processo è occupato a swappare in e out le pagine e quindi non lavora sul suo core task. Questo accade quando l'insieme di pagine attualmente in RAM sono tutte necessarie per l'esecuzione del task ma non sono abbastanza, e alcuni frame necessari al processo sono su disco. In questo caso infatti un frame richiesto verrà swappata dal disco alla RAM, togliendo però un altro frame anch'esso attualmente in uso, dovendo presto riswappare per ottenerla.

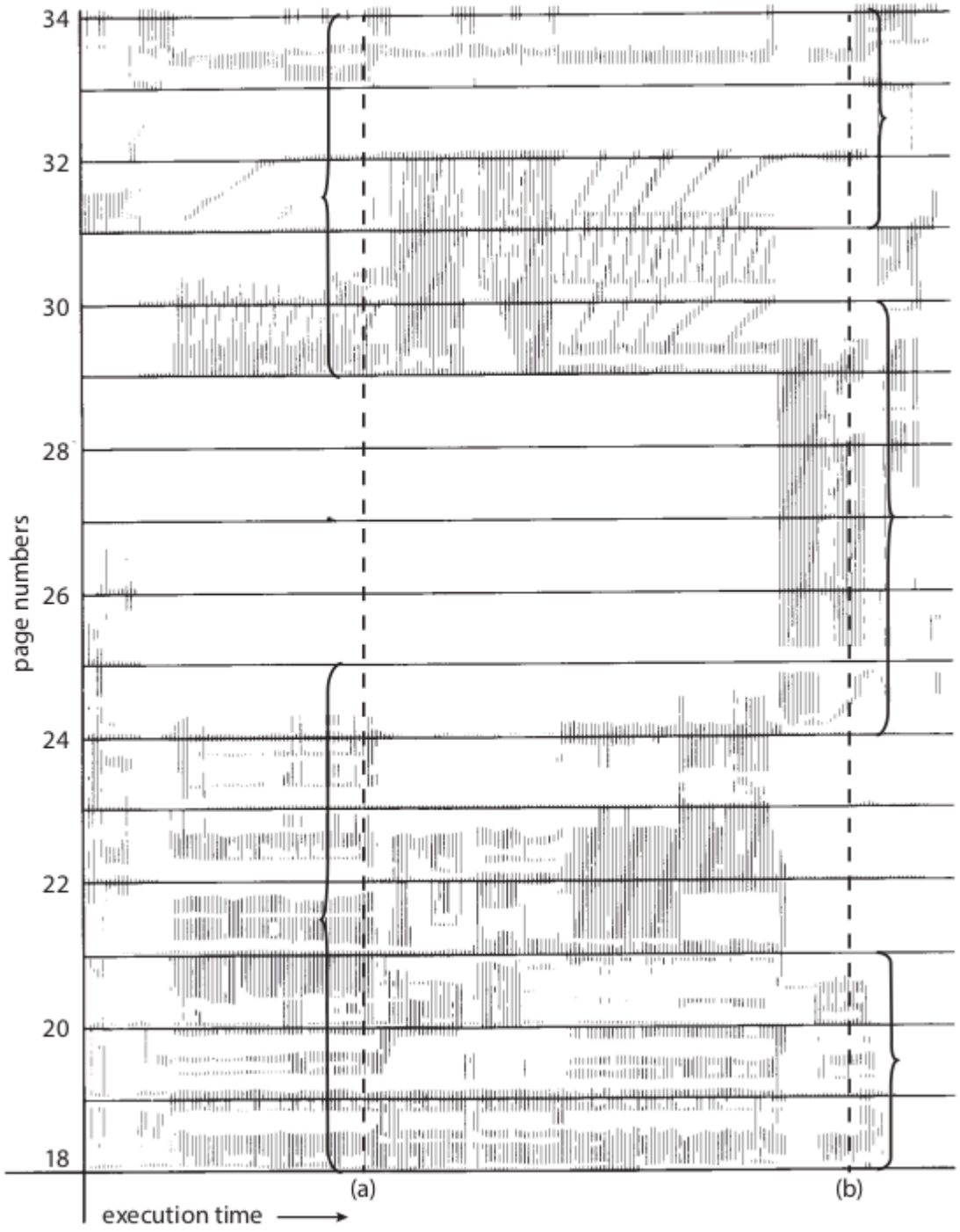
Una causa di thrashing può essere la seguente: dato un algoritmo di page replacement globale, che swappa frame senza sapere a che processo le sta togliendo, due processi potrebbero "ostacolarsi" l'uno con l'altro nel momento in cui, chiedendo una pagina, vanno a swappare frame attualmente usate dall'altro. In questo modo i processi iniziano a passare più tempo nel page swapping che nel loro core, riducendo l'utilizzo della CPU. L'OS, che si accorge che la CPU utilization è bassa, decide di aumentare il numero di processi per aumentare la CPU utilization, ma i nuovi processi generano nuovi page fault e diminuiscono ancora di più la CPU utilization, quindi la CPU aumenta il grado di multiprogramming e così via in un circolo vizioso.



Possiamo limitare l'effetto del thrashing con un *local replacement algorithm* (priority replacement algorithm). In questo modo un processo che vuole swappare una frame può farlo sono con frame che sono allocate a lui. Questo però non risolve il problema del tutto, perché il tempo di attesa per ricevere una pagina aumenterà anche se il processo non è in thrashing.

Memory Access Pattern:
L'unico modo di prevenire il thrashing è dare al processo il giusto numero di frame di cui ha bisogno. Come facciamo a capire qual è questo numero di frame che il processo necessita?

Usiamo il modello di località: la località è un insieme di pagine che vengono usate insieme. Le località vengono definite dalla struttura del programma e dei dati. Quando un processo esegue, si muove da località a località. Quindi il principio è: se ho acceduto a qualcosa poco fa è probabile lo riprenderò perché fa parte della attuale località. È grazie a questo modello se le cache funzionano. Qua sotto un pattern di accesso a pagine:



Quindi, durante l'esecuzione di un processo, una volta terminati i page fault per allocare la località, non faulterà più finché non cambierà località. Questo significa che se non permettiamo di allocare un numero di frame almeno grande quanto la località, il processo andrà in thrashing sicuro, perché non ci saranno mai in RAM tutte le pagine della località che ci servono contemporaneamente.

Working-Set Model:

Basandoci sull'assunzione di località, possiamo definire una ***finestra di working-set***. Dato un parametro Δ, l'insieme di pagine dato dagli ultimi Δ accessi a pagine è il working-set. Quindi dopo Δ accessi, se una pagina non è stata acceduta viene droppata dal working-set. Se Δ troppo piccolo non prende tutta la località, se troppo grande prende più località. Data WSS(p) la dimensione del working-set per il processo con PID p, possiamo dire che:

$D = \sum_{p} \text{WSS(p, Δ)}$

è il numero totale di frame richiesti. Se D > max_frames si verifica thrashing.

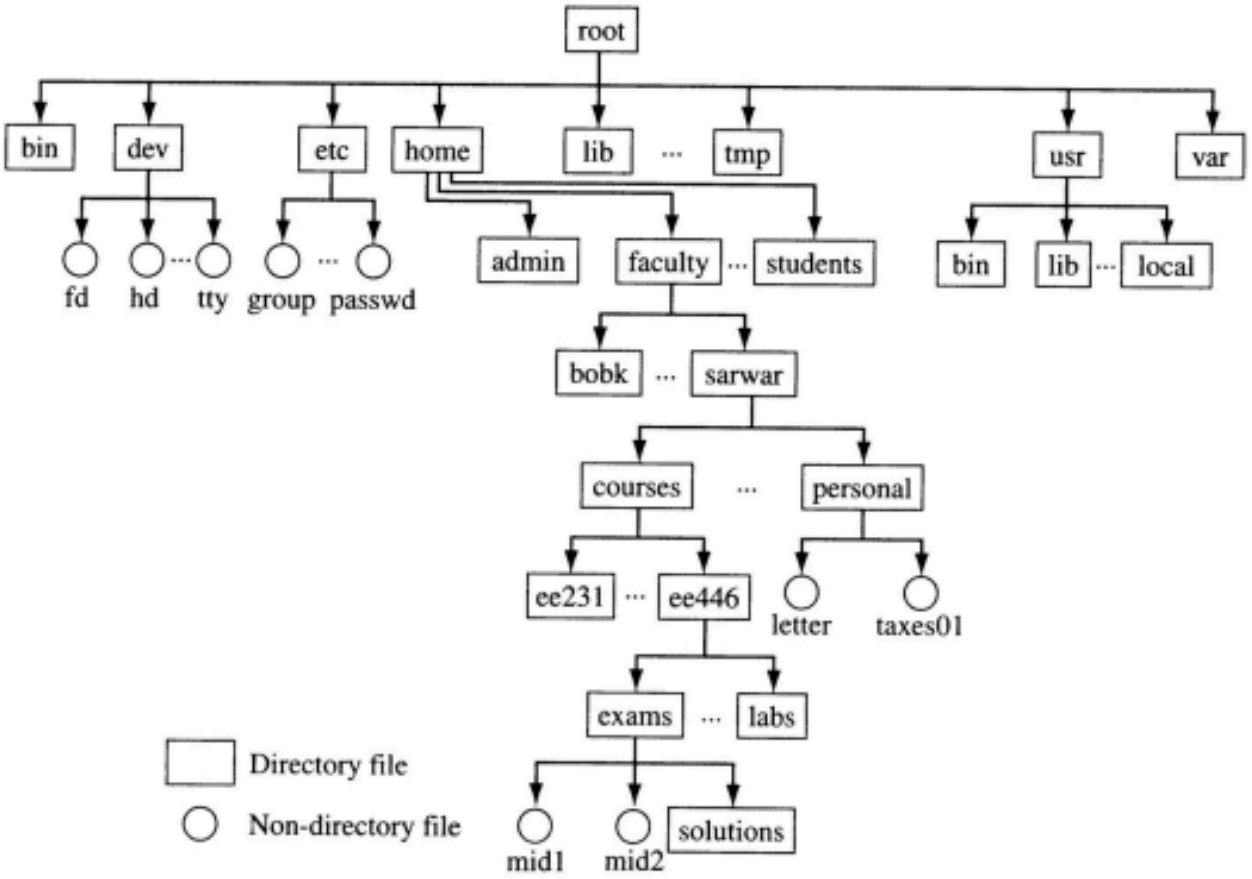
Istruzione mmap:
per fare shared memory tra processi ma anche per mappare file da disco su memoria. Quando mappo un file dal disco a memoria nel mare di mordor del processo posso leggermelo/scriverci come fosse un array mentre sotto avvengono i meccanismi di caching e swapping delle sue pagine. Mentre scorro il file, ciò che ho già visto viene buttato in modo da fare spazio in memoria.

File System

Il file system risiede su memoria secondaria. Crea un'interfaccia (API) utente per lo storage.

Concetti importanti sono:

- Mount/Umount:
    a. Mount: prendere device (che è un file) e mapparlo nell'albero delle directory
    b. mount point: directory su cui si monta il file system di un device, quindi fa da bridge da un device e un altro.
- Link:
    a. soft (symbolic) link. Quando viene eliminato un soft link non succede nulla al file originale.
    b. hard link. A tutti gli effetti un mirror del file originale. Esiste per ogni file su disco almeno un link fisico: quello della directory da cui lo raggiungiamo. Se il numero di link fisici per un file è 0 il file viene eliminato.



La root (/) contiene importanti directory come:

- /sys: file system con cui il kernel comunica all'utente lo stato dell'hardware. Non corrisponde a nessun disco fisico in memoria secondaria (ma comunque deve risiedere in RAM per poterlo leggere/scrivere) anche perché non avrebbe senso andare ogni volta a leggere/scrivere su disco informazioni come queste. Quando linux parte oppure si carica un modulo il sistema operativo popola delle strutture dati qua dentro.
- /proc: file system (finto come sys, non vado in memoria secondaria a leggere aggiornamenti sui processi ogni volta ma ce li ho in RAM) che l'OS popola aggiornando le informazioni sui processi. È una vista sugli attuali PCB esistenti.
- /dev: vista sui device fisici.

File: un file è uno storage persistente virtualmente considerabile come un array di caratteri. In Unix tutto è un file o comunque ha un API file-like. Attributi di un file:

- Nome
- Identificatore
- Directory in cui risiede
- Dimensione
- Permessi: <rwx> per <ugo> (per i file directory esecuzione è fare listing)
- Ownership: posseduto da un utente e posseduto da un gruppo
- Timestamps: creazione, ultima modifica…

Oltre a questi il sistema operativo vede:

- Un identificatore unico per il sistema
- La location (il device) dove è fisicamente memorizzato

File Descriptor: ogni processo che apre un file mantiene un intero chiamato file descriptor. Tutte le azioni sul file sono fatte attraverso il file descriptor. I file descriptor 1 e 2 sono esclusivi per stdout (stampa su terminale) e stderr (stampa errore su terminale) rispettivamente.

Device e Driver: i device sono dispositivi fisici che si mettono nel sistema e sono accessibili come file. Si dividono in device a blocchi (b davanti ai permessi) e a caratteri (c davanti ai permessi). Il sistema operativo on mount va a inserire in /dev il device.

Un driver fa override dei metodi standard per operare sulla memoria fisica che sono definiti dall'interfaccia.

ioctl() (input/output control) è una system call che permette di comunicare con il driver di un device a user level. ioctl() ha tre argomenti: un identificatore del dispositivo, un intero che seleziona uno dei comandi implementati dal driver (è il modo in cui specifichi quale request vuoi fare, ogni device driver installa le sue richieste e gli handler per le richieste), un puntatore a una struttura dati per scambiarsi informazioni tra applicazione e driver.

Ad esempio se voglio dire alla telecamera di cambiare risoluzione mi serve l'identificatore della telecamera, l'intero relativo alla funzione (request) che voglio evocare (cambia_risoluzione) e nella struttura dati gli passi le informazioni sulla risoluzione.

Disco: di solito organizzato in partizioni. C'è bisogno quindi di una tavola delle partizioni per descrivere il layout. Il disco per il sistema operativo è un file: posso scrivere sul disco accedendo come fosse un array, a prescindere dal suo file system. Perciò per scriverci come fosse un file ho bisogno di implementare operazioni elementari come read(), write() e seek() (truncate no). Il kernel offre al vendor del driver l'interfaccia più semplice possibile per interagire con l'hardware. Una volta che il vendor del driver ha implementato queste funzionalità il device si può mettere nella macchina. Il driver quindi per il kernel è una classe astratta che viene implementata dal vendor.

Quindi abbiamo disaccoppiato l'esistenza del file system e le sue operazioni dal disco fisico e le operazioni che facciamo sul disco come fosse un file (read(), write() e seek()).

DMA: Direct Memory Access. Un device accede direttamente alla memoria interna della macchina. Alcuni device necessitano di trasferire grandi quantità di dati a frequenze elevate. Effettuare tali trasferimenti tramite il protocollo genericamente usato per altri tipi di periferiche richiederebbe l'intervento della CPU per trasferire un byte alla volta i dati. Ciò risulterebbe in un overhead ingestibile per l'intera macchina, consumando inutilmente la CPU. Per consentire il corretto funzionamento di tali device, tali periferiche possono avvalersi di controllori dedicati che effettuano DMA, cioe' andando a scrivere direttamente sul bus di memoria. La CPU sara' incaricata soltanto di "validare" tale trasferimento e poi sara' di nuovo libera di eseguire altri task. Questo tipo di periferiche sono molto comuni ai

giorni nostri e sono usate nella maggior parte dei dispositivi elettronici - pc, smartphones, servers, ... Esempi di periferica che si avvalgono di controller DMA sono videocamere, dischi, schede video, schede audio, ecc.

Quando devi aprire una seriale c'è l'equivalente dell'ioctl che è il termios() (l'hanno nobilitata facendo un'interfaccia a parte perché erano molto usate).

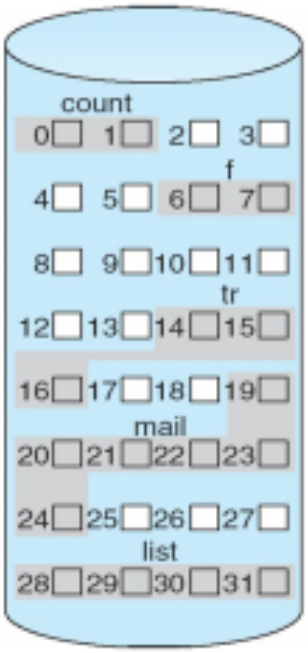
Implementare un file system: voglio directory e file su una zona di memoria. Un file system è un pezzo di software che implementa open(), write(), read() lato "esterno" e dall'altra parte aderire all'interfaccia del disco, cioè utilizzarla per creare sul dispositivo delle strutture dati che saranno directory, file ecc.

Abbiamo bisogno:

- Lato kernel di un array di OpenFileInfo. Ogni elemento di questo array è associato a un file aperto. Il fd restituito dalla open() è l'intero che identifica l'indice dell'array dove trovare le informazioni sul file.
- Lato disco: ogni file ha una struct FileControlBlock su disco. Dentro al FCB ho la posizione del file sul disco, il nome del file, i permessi, i gruppi ecc. FCB è su disco ed è mirrorata su RAM quando dobbiamo fare le modifiche ovviamente.
  Una directory, che è un file speciale, ha un FCB e poi la sequenza di record per i file che contiene.

Tutti i file (anche le directory) posso spannare su più blocchi, come gestire questa cosa?

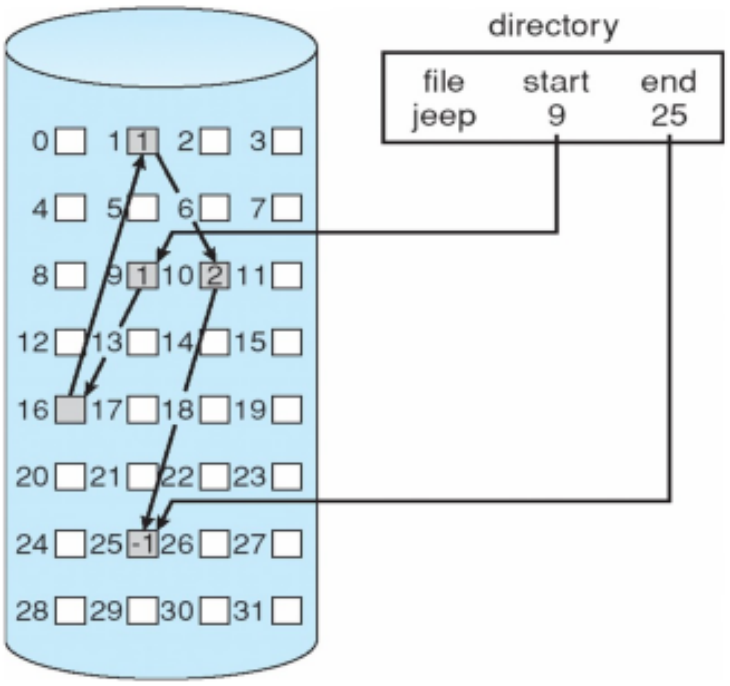
- Contiguous Allocation: mi serve solo la starting location e la sua lunghezza, ma ho problemi di frammentazione esterna. Inoltre se un file cresce mi può andare a raggiungere altri file, poi che faccio?



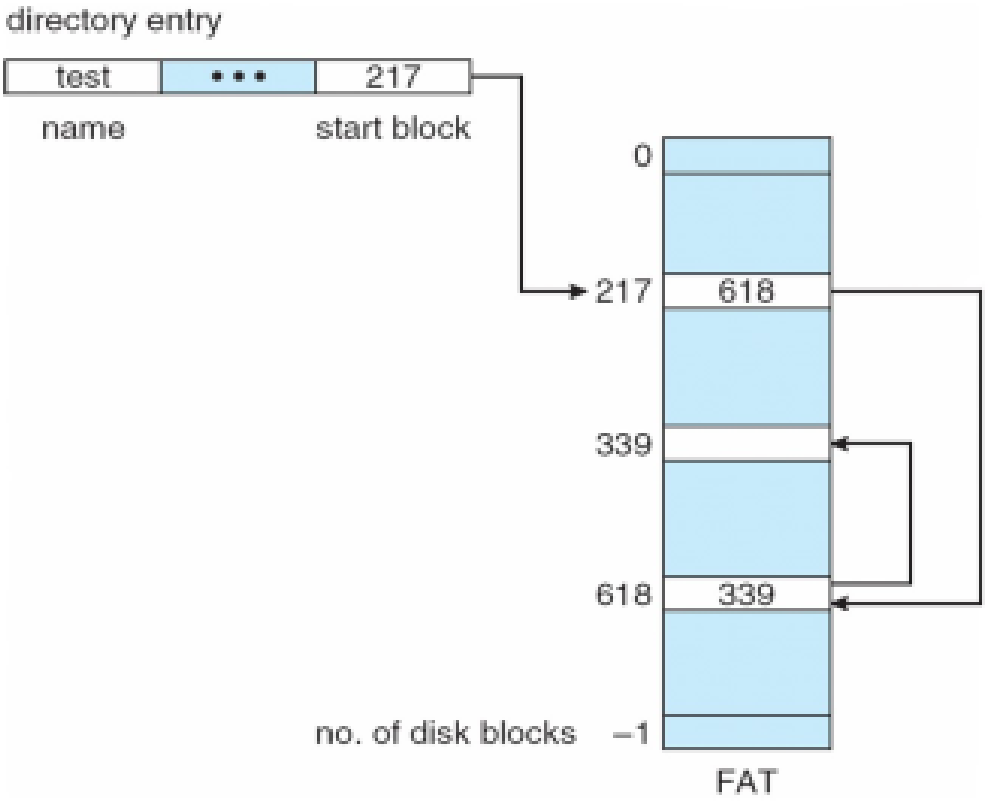
directory

file	start	length
count	0	2
tr	14	3
mail	19	6
list	28	4
f	6	2

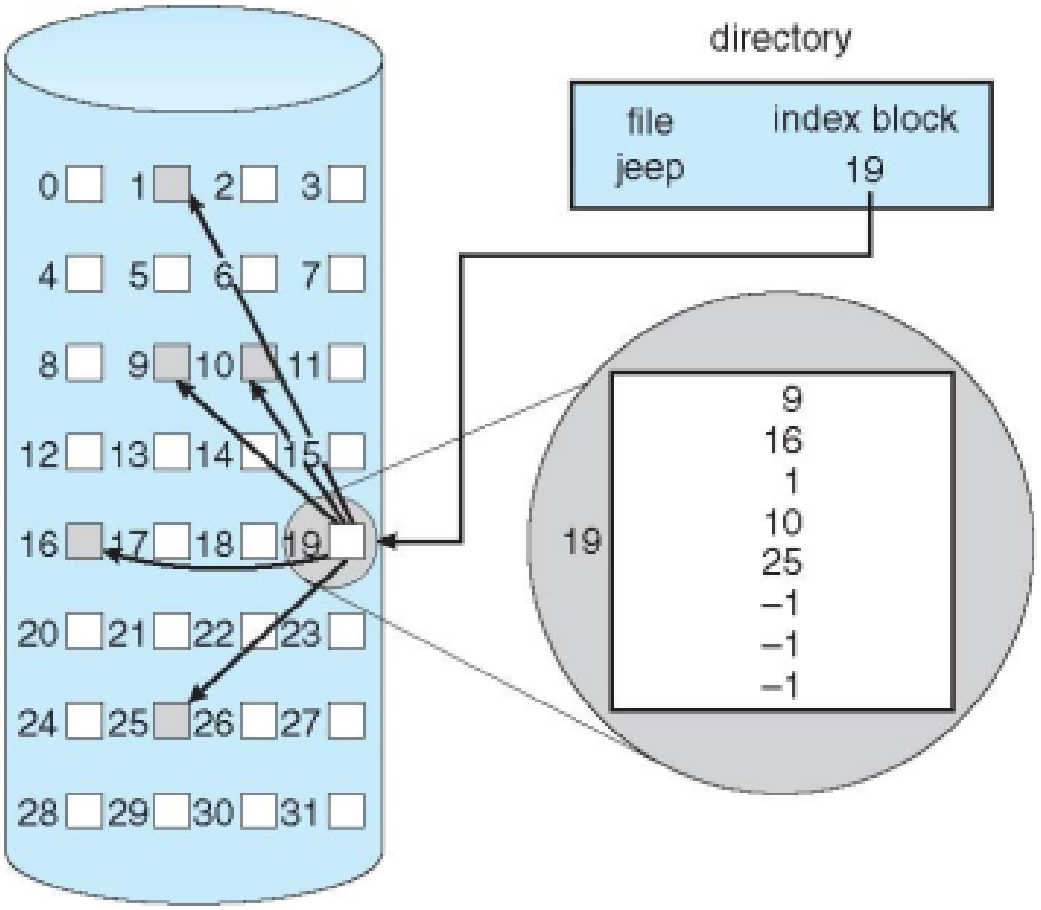
- Linked Allocation: in un blocco oltre alle info sul file ci metto un intero che indica l'indice del blocco successivo. In questo modo evito la frammentazione esterna ma aumento il costo d'accesso: contiguous allocation O(1) mentre in questo caso O(n). Linked Allocation è comunque meglio della Indexed per accessi sequenziali, soprattutto se l'FCB non sta in RAM ma su disco quindi nella Indexed ogni volta mi devo fare un accesso a disco per prendermi il prossimo indice (se invece sta su RAM mi devo fare un accesso a RAM che costa molto meno però comunque è di più rispetto a Linked Allocation).



- FAT: all'inizio del disco ho una File Allocation Table. Ogni directory entry contiene l'indice del blocco di inizio del file. Questo indice serve anche per andare a recuperare nella FAT l'informazione sulla posizione dove si trova il prossimo blocco del file. La FAT è piccola quindi si può caricare tutta su RAM quando serve per navigare nel disco.

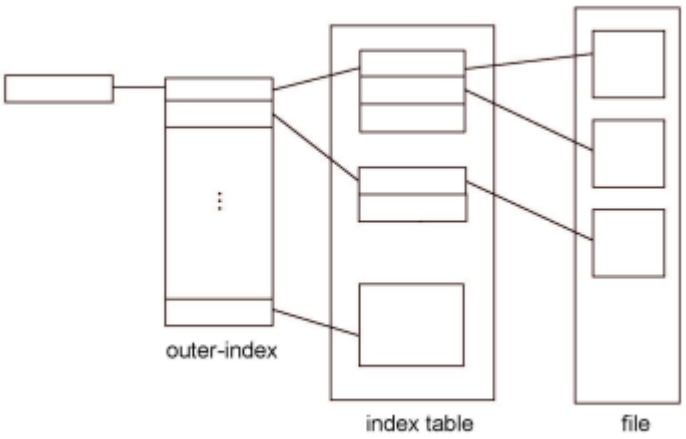


- Indexed: (Inode): modo standard di Unix di allocare i file. Tenere traccia della struttura del file come fosse un albero. Quindi io nel FCB tengo un array dove dico dove si trova il blocco 1, 2, 3, …



C’è un problema: nei sistemi esistono file che sono più piccoli di un blocco del disco. Quindi con questa struttura qua spreco due blocchi: uno per l’FCB e l’altro dove metto il file. Posso risolvere con un FCB che uso sia per memorizzare gli indici sia per memorizzare dei dati così se il file è molto piccolo sta tutto dentro il FCB. Trade off: sto togliendo dello spazio alla linked list che ci memorizza quanti blocchi abbiamo e quali sono, sto limitando la dimensione massima dei blocchi su cui un file può spannare.

Posso fare ancora di più: nel FCB ci metto solo i dati e eventualmente il primo puntatore a un blocco che contiene dei blocchi di alto livello (outer-index). Se i dati esondano prendi un blocco di indici di alto livello in cui ci metti gli indici delle tabelle di blocchi (index table) che a loro volta sono puntatori a blocchi.



In questo modo puoi aumentare quanto vuoi la dimensione del file basta aumentare la profondità dell'albero. Implicitamente la FAT ha una bitmap dentro.

Come implementare una directory:

- Linear List di entry che rappresentano i FCB ai file. Se ho 2 milioni di file? È Inefficiente.
- Hash Table in cui metto le varie entry per avere più efficienza sulla ricerca. Bisogna gestire le collisioni.

Come gestisco lo spazio libero? Con una free list come in RAM con gli allocatori.
Un'alternativa è una bitmap.

Inter Process Communication

Sono due: o dai una memoria condivisa ai processi oppure dai la possibilità di inviare messaggi da un processo a un altro.

Message Passing:

Primitive:

- send() può inviare a esattamente un processo (comunicazione diretta) oppure a una mailbox comune (indiretta)
- receive()

Può essere bloccante o non: se non ci sono messaggi metti in pausa il processo che legge, quando si riceve un messaggio lo sblocchi.

Idem quando si invia ed è piena.

code limitate (upper bound molto grande oppure proprio senza) / illimitate

Altre funzionalità: considera la mailbox come un file

- open: crea una nuova mailbox
- close: non cancella la mailbox ma comunque ti ci stacchi.
- unlink: distrugge la coda dalla RAM (se vai in /dev/mqueue non la trovi più CONTROLLA, NON VIENE ELIMINATA QUANDO I LINK SONO 0 ?). In realtà la coda è realmente rimossa nel momento in cui tutti i processi che hanno un link (file descriptor) alla coda l'hanno chiuso, prima di ciò gli altri possono ancora usarla.
- get/set attr: dimensione massima dei messaggi, numero massimo di messaggi che la coda può contenere, …
- wait for a message/notify when a message is ready

Le code posix create dai programmi sono in /dev/mqueue finché non vengono distrutte o si fa il boot (perché non sono fisicamente su disco)

Shared Memory:

Sono mappate su dei file anche loro. Tre funzionalità:

- open
- truncate: ridimensionare la memoria in base a quanto ci interessa
- mappa: mappiamo la shmem nello spazio di memoria del processo
- close
- destroy

Le shared memory sono in dev/shm finché non vengono unlinkate o non si fa il boot.

Segnali

Il processo non sa da chi ha ricevuto il segnale a differenza di una callback. Una callback è quando su una finestra clicchi un pulsante e la finestra si accorge che tu hai cliccato (quindi sanno che sei tu ad aver cliccato). Le callback sono esplicitamente chiamate dal codice e in C si implementano con un puntatore a funzione.

Se arriva un segnale che non viene ignorato e neanche gestito con la semplice terminazione del programma, ma bensì si vuole gestirlo con delle operazioni specifiche, succede la seguente cosa:

- Ricevuto il segnale il flusso di esecuzione del processo viene interrotto
- Viene invocata la callback con cui si vuole gestire il segnale (non è come quando clicchi il pulsante sulla finestra perchè appunto non è che la callback fa parte del normale flusso di esecuzione del processo ma invece viene interrotto)

Per catturare un segnale si usa sighandler_t signal(int signum, sighandler_t handler)

le specifiche degli handler per i segnali sono ereditati se si usa fork() mentre con exec() vengono ereditati solo se handler è stato settato a SIG_IGN oppure SIG_DFL, per tutti gli altri si rimette a SIG_DFL

Come fa un kernel a gestire i segnali?

Innanzitutto facciamo questa analogia: i segnali sono interrupt ad alto livello. Le interrupt sono controllate alla fine di ogni ciclo di clock, l'equivalente del ciclo macchina per i processi è un'invocazione dello scheduler o di una syscall, insomma quando invochi il kernel. Posso fare così quindi:

- Un segnale ha un suo contesto, quando viene spostato da ready a running controllo se nel frattempo quel processo ha ricevuto dei segnali (perché ho settato a 1 da qualche parte una variabile corrispondente)
- In tale caso invece di riesumare il processo riesumo il contesto del signal handler
- Quando l'handler ha finito il flusso di esecuzione ritorna al processo

Una volta ricevuto un segnale devo mascherarlo perché non posso riceverlo due volte altrimenti faccio casino: mentre sto gestendo un segnale dovrei rigestirlo di nuovo

Se durante il normale flusso di esecuzione di un processo c'è una syscall che ci mette molto tempo e intanto ricevi un segnale che fai? Interrompi la syscall? Finisci la syscall e gestisci il segnale? Ma i segnali sono installati a livello di processo, quindi più thread hanno tutti lo stesso signal handler, quindi se un thread becca un SIGFAULT mentre l'altro sta in syscall che fai?

Le syscall sono progettate per abortire se un processo riceve un segnale. Ritornando errore e dentro la var globale errno mettono EINTR

Con pause() blocchi il processo in attesa di un segnale ma non ti dice quale segnale ha ricevuto. Per sapere quale segnale hai ricevuto devi scrivere il signal handler per ogni segnale possibile e ad ogni signal handler fai manifestare di essere il gestore per il segnale x per esempio tramite la scrittura su una variabile x

La gestione del segnale comporta automaticamente che il signal_handler viene annullato e prossima volta viene ignorato, quindi se lo vuoi sempre ogni volta che esegui il signal_handler devi reinstallarlo proprio dentro il codice del signal_handler

Signal Sets: interfaccia per gestire insieme tutti i segnali

sigset_t è un insieme di segnali con operazioni per aggiungere, eliminare e fare test di appartenenza

in questo modo posso settare tutti i segnali insieme ma sopratutto *gli handler non vanno reinstallati ogni volta che viene eseguito l'handler*. Questo evita problemi come il seguente:

L'utente preme ^C per un processo

Viene inviato un segnale SIGINT al processo che ha un handler installato

Prima che venga reinstallato l'handler l'utente preme un'altra volta ^C

Il processo, che non ha un handler installato, termina

In questo modo invece i segnali rimangono e non vengono automaticamente disarmati, così l'utente può premere ^C quante volte vuole e per ogni volta verrà attivato l'handler

Masking Signals: sigprocmask syscall per alterare la maschera dei segnali.

Possiamo quindi bloccare ^C quando stiamo gestendo ^C

sigaction: modo moderno di mettere segnali

riempi una struttura dati dove metti l'handler, la maschera dei segnali, …

sa_handler è l'handler vecchio

sa_sigaction è il nuovo, prende un siginfo_t

siginfo_t popolata dal kernel e contiene un sacco di informazioni sul segnale

Device

In linux tutto è un file, anche i device vengono visti come file. Esistono char device (ci interagisci con stream device) e block device (gestiscono lo spostamento dati, tipo l'SSD). Anche i driver dei device, i pezzi di codice che interagiscono con i device, sono file.

Come interagire da user con il joystick:
Il device del joystick sta in /dev/input/ apriamo il file e leggiamo dal file degli eventi js_event
Il driver interagisce con lo stack usb, legge i dati dal js li impacchetta e li mette nel file (questo però è lato kernel).

Come interagire da user con la porta seriale:
Sta in /dev/ttyUSB0. Bisogna configurare la seriale (settare baud rate, dimensione dati, …) tramite la struttura *termios*. ioctl() viene reimplementata dai device driver sulla base del loro utilizzo specifico. Per la seriale abbiamo una wrapper di ioctl() che è termios. Cutecom è un front end per la configurazione e la visualizzazione della seriale, quindi fa queste cose che abbiamo detto però da GUI.

Come interagire da user con la telecamera:

Apriamo la camera come un file (camera_open), configuriamo camera e DMA controller (camera_init) e chiediamo alla camera di iniziare a riempire i buffer (camera_start), dopo catturiamo i frame dal buffer e li scriviamo su disco.