

## SISTEMI OPERATIVI – PER ESAME

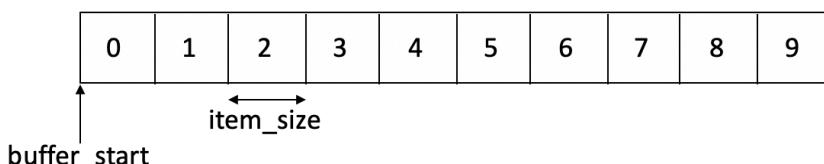
### Memory Allocators

La memoria è vista dal Kernel come un gigante array lineare e per far sì che esso sia efficiente, ogni operazione non può superare  $O(\log n)$ . Esistono due modi per allocare memoria e ognuno dei due per essere efficiente deve far fronte a due problemi:

- Frammentazione: Si presenta se l'array ha degli spazi vuoti all'interno di esso e quindi se la memoria non è contigua. Potrei addirittura avere abbastanza memoria ma magari non abbastanza spazio contiguo.
- Tempo: Quanto tempo occorre per ricevere/rilasciare memoria

#### Tipi di Allocazione

- **SLAB Allocator – Memoria Fissa**: È un allocatore di memoria che spesso viene utilizzato se ci sono tanti elementi di dimensione fissa (devo sapere quanti elementi ho). Consiste nell'“affettare” la memoria in tante fette quante sono gli elementi da allocare.



- Se la memoria inizia all'indirizzo `buffer_start`, l'indirizzo del blocco `idx` sarà:

$$\text{ptr\_block} = \text{buffer\_start} + \text{idx} \cdot \text{item\_size}$$

- Se conosco l'indirizzo, come trovo l'indice? Formula inversa:

$$\text{idx} = \frac{\text{ptr\_block} - \text{buffer\_start}}{\text{item\_size}}$$

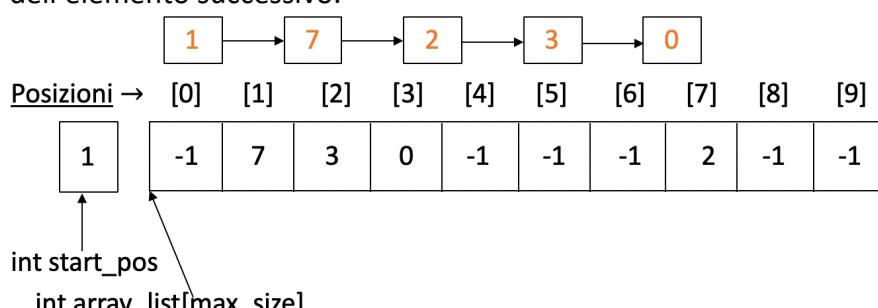
Per soddisfare richieste in  $O(1)$  potrei costruire uno SLAB Allocator formato da una lista di interi, che popoli la lista blocco per blocco. Purtroppo però non posso usare `malloc`, quindi a tale struttura preferisco l'ArrayList.

#### Slab Allocator → ArrayList

Se conosciamo la dimensione massima di una lista, possiamo mappare la lista in un array di `max_size` elementi. Per fare ciò, abbiamo bisogno di:

- Una posizione di partenza (`int start_pos`)
- Un array di interi (`int array_list[max_size]`)

Memorizzo virtualmente la lista nell'array. Nel blocco di posizione  $i$  metto l'indice dell'elemento successivo.

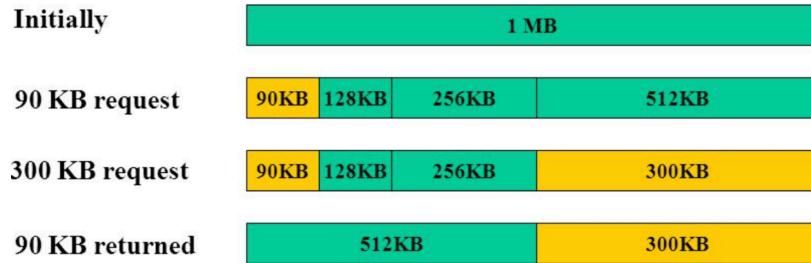


Spiegazione: L'array list sopra rappresenta gli elementi che devo allocare nello SLAB (array sotto), che per me rappresenta le posizioni.

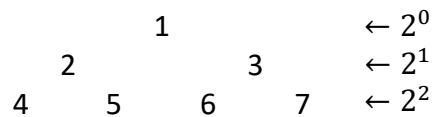
- In posizione 1, ci sarà il successore di 1 (7)
- In posizione 7, ci sarà il successore di 7 (2)
- In posizione 2, ci sarà il successore di 2 (3)
- In posizione 3, ci sarà il successore di 3 (0)



- Le restanti celle saranno riempite con -1
- **Buddy Allocator – Memoria Variabile:** Il buddy allocator ha un costo maggiore rispetto allo SLAB, ma posso utilizzarlo come una malloc. In questo caso, la regione di memoria viene divisa risorsivamente in due. Un “buddy” di un blocco di memoria è una regione ottenuta partizionando la regione “genitore”. Se richiedo memoria, il Buddy tenta di capire a quale livello si trova la regione interessata.



Possiamo vedere la memoria come un albero binario con nodi numerati con vista in ampiezza.



- Il livello di un nodo è la parte intera del  $\log_2(\text{indice})$
- L'indice del Primo nodo ( $\text{firstIdx}(i)$ ) di un livello  $i$  è  $2^i$  ( $= 1 \ll i$ )
- L'offset di un nodo nel suo livello è:  

$$\text{idx} - \text{firstIdx}(\text{level}(\text{idx}))$$
- L'indice del buddy del nodo  $i$ :  

$$\text{buddyIdx}(\text{idx}) = (i \% 2)? \text{idx} - 1: \text{idx} + 1$$
- Genitore del nodo  $\text{idx}$ :  

$$\text{parentIdx}(\text{idx}) = \text{floor}(\text{idx}/2)$$

Per gestire al meglio gli elementi dell'albero, creo una bit Map, ovvero popolo un array di 0 o 1 per far sì che sappia facilmente se il nodo è libero o no.

## Interrupts and Syscall

I moderni sistemi operativi sono basati sugli Interrupt, ovvero ogni interazione con l'OS è innescata da un interrupt. Un interrupt può sorgere in 3 modi diversi:

- Da eventi esterni (es: I/o, Timers)
- Eccezioni Interne (es: Istruzioni Illegali, ...)
- Chiamate esplicite (es: Syscall)

Due tipi di Interruzioni:

- Polling: Interrogo continuamente lo stato. Es: `while(1){if(key_pressed) ...}`
- Opzione di Interruzione: La maggior parte del tempo il sistema dorme, e viene svegliato solo al verificarsi di un evento.

Quando avviene una interruzione, il contesto corrente viene salvato a la CPU inizia ad eseguire la routine dell'interrupt. Di solito quando viene servito un interrupt, vengono disabilitate dall'OS interruzioni dello stesso tipo.

- **Interrupt Vector:** È un array di puntatori a funzione contenenti l'indirizzo delle ISR (*Interrupt Service Routine*) che il sistema operativo dovrà eseguire in risposta all'interrupt.
- **Eccezioni:** Le eccezioni sono interrupt software. In x86 sono suddivise in:
  - **Traps:** ISR viene invocata dopo aver attivato l'istruzione

- **Faults:** ISR viene invocata prima di aver attivato l'istruzione (evento che si verifica se si cerca di compiere una istruzione illegale (es: divisione per 0)).
  - **Aborts:** Lo stato del processo innescante non può essere recuperato (es: mentre gestisco un fault avviene un altro fault).
- **INT and CALL:** Le chiamate a sistema operativo sono interruzioni
  - **INT <xx>** (Interruzione): Salto all'ISR il cui indirizzo è salvato in posizione <xx> dell'interrupt vector. Salvo anche lo stato della CPU, salvando i flag in quanto verranno alterati.
  - **CALL <yy>** (Chiamata a Funzione): Chiamo una subroutine il cui indirizzo è <yy>, il quale deve essere uno dei validi indirizzi mappati nell'area di memoria eseguibile di un processo. I flag non vengono alterati
- **Dual Mode:** In dual mode il sistema viene suddiviso in due:
  - **User Mode:** In user mode solo un sottoinsieme di istruzioni può essere eseguito.
  - **Privileged Mode:** ISR sono eseguite in privileged mode.  
Come faccio da utente ad invocare il sistema operativo? 3 possibili soluzioni:
    - 1) Nascono l'intero OS dietro una voce dell'Interrupt Vector ← NO!
    - 2) Le specifiche funzioni dell'OS vengono invocate attraverso una istruzione INT<OS\_ISR>, ovvero nascondo tutto l'OS dietro ad un'interruzione software (INT 80)
    - 3) I parametri delle syscall possono essere:
      - a. Nei registri della CPU
      - b. Sulla Stack

Soluzione: Solo un sottoinsieme delle locazioni nell'Interrupt Vector può essere chiamato in User Mode senza generare eccezioni.
- **Syscall:** Il kernel offre molte funzionalità, ma abbiamo una singola ISR per gestirle tutte. Per eseguire una syscall enumeriamo le possibili funzioni e passo al registro EAX l'indice della funzione corrispondente all'indice del vettore delle syscall e per ripristinare lo stato precedente alla chiamata e ripristinare i flags utilizzo IRET (return from interrupt).  
Le Syscall offrono funzionalità di basso livello (read/write, open/close, wait,...) e sebbene queste funzionalità consentano lo sviluppo di applicazioni complesse, operando a livello di chiamata di sistema risulterebbe una stretta dipendenza dall'OS e il software dovrebbe essere riscritto per ogni tipo di OS.
- **Standards:** Programmazione a Strati
  - **Language Standards:** È un tipo di programmazione più ad “alto livello” e scrivere programmi usando solo le funzioni che offrono le loro librerie standard (libc) assicurano la portabilità.
  - **System Standards:** È un livello più basso, quindi meno portabile.

Esercizio: Cos'è la tabella delle syscall e cos'è l'interrupt vector?

L'interrupt vector è un vettore di puntatori a funzioni che rappresentano *ISR* (Interrupt Service Routine) che gestiscono i vari interrupt.

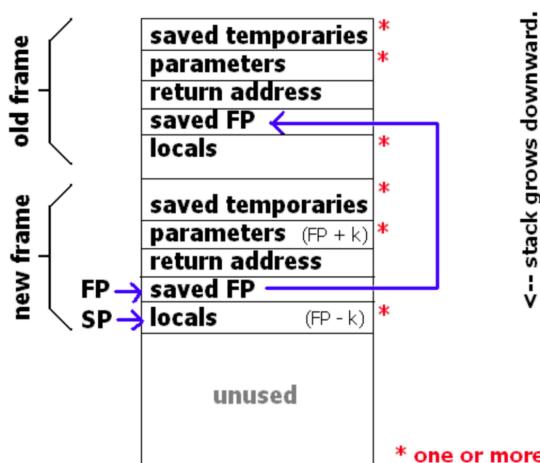
Analogamente, la tabella delle syscall conterrà in ogni locazione il puntatore a funzione che gestisce quella determinata syscall. Alla tabella verrà associata anche un vettore contenente il numero e l'ordine dei parametri che la syscall richiede.

## Processes

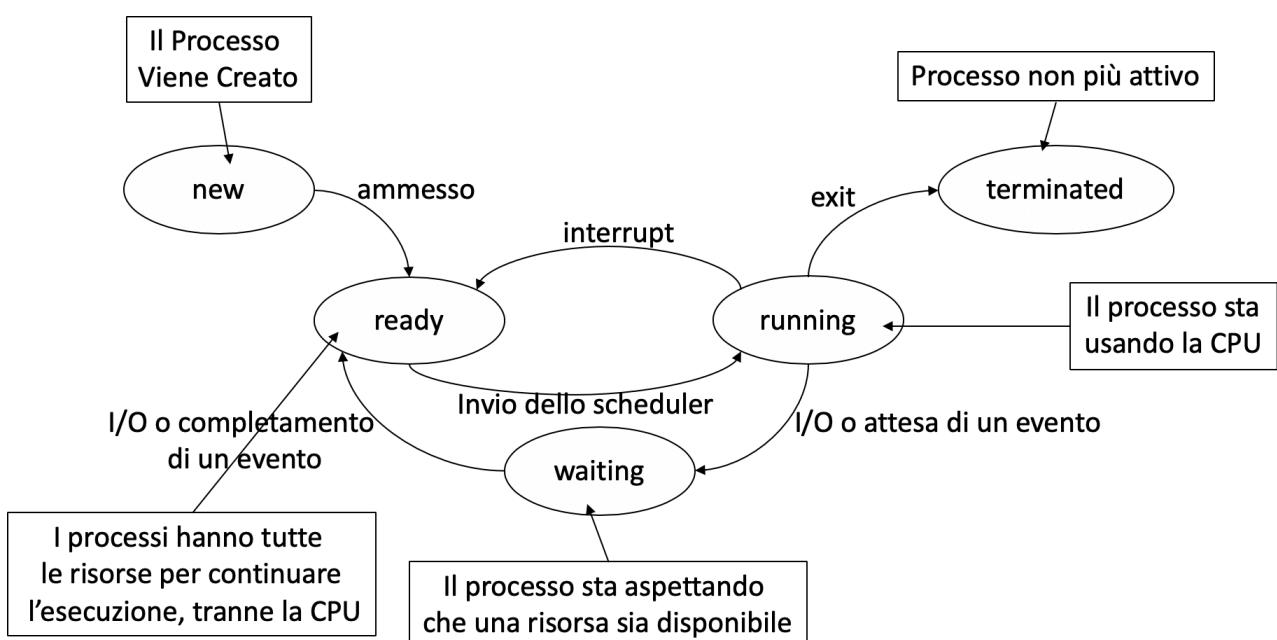
Un processo è un programma in esecuzione caratterizzato da:

- Registri della CPU
- Memoria
  - Code (.text)
  - Stack → int res; float b;
  - Heap (int\* p = malloc(...))
  - Variabili globali (.bss e .data)
  - Regioni mappate in memoria (tra stack e heap)
- Risorse
  - Descrittori File/Socket
  - Costrutti di Sincronizzazione (semafori, code)

**Stack Frame:** Uno stack frame è un'area di memoria di un programma organizzata a stack in cui vengono immagazzinate le informazioni (argomenti, indirizzi di ritorno, registri salvati, variabili globali) sulle subroutine attive in un dato momento.



**Stati di Un Processo:**



- New → Ready: Un processo entra nello stato ready dopo che le risorse richieste sono state allocate. In questo stato ci sono tutti i processi che si trovano in memoria centrale.
- Ready → Running: Un processo va nello stato running non appena viene passato il controllo della CPU. Dallo stato running può raggiungere tutti gli altri stati.
- Running → Terminated: Un processo passa dallo stato running allo stato terminated quando viene portata a termine l'esecuzione del programma e quando un processo viene terminato vengono rilasciate tutte le risorse che gli erano state assegnate.
- Running → Ready: Un processo passa dallo stato running allo stato ready quando il kernel decide di schedulare un altro processo. Es: un processo a priorità più alta va nello stato ready.
- Running → Waiting: Un processo passa dallo stato running allo stato waiting quando effettua una system call per richiedere l'uso di una risorsa o quando rimane in attesa di un evento. Un processo bloccato si trova in memoria ma non necessariamente in memoria centrale, es: in memoria swap.
- Waiting → Ready: Un processo passa dallo stato waiting allo stato ready quando la richiesta del processo viene soddisfatta o quando si verifica l'evento.

Per creare un nuovo processo viene utilizzata la system call fork. Con questa chiamata vengono create due istanze dello stesso processo:

- La memoria, i file descriptors e le risorse vengono copiati dal genitore al figlio
- Il valore di ritorno di una fork è 0 per il figlio, child\_pid per il genitore

Le syscall wait e waitpid sospendono l'esecuzione di un processo finché uno dei processi figli termina (wait) o uno specifico processo figlio termina (waitpid).

Quando il processo figlio muore, il genitore riceve un segnale (SIGCHLD), mentre quando il padre muore i figli ricevono un altro segnale (SIGHUP).

Un segnale non è una interruzione, ma un evento asincrono e per gestirli possiamo installare un gestore dei segnali (struct sigaction).

Un processo termina la sua esecuzione con exit. A questo punto rilascerà tutte le sue risorse e entra nello stato zombie. N.B.: Un processo muore definitivamente solo con la wait, ovvero solo quando qualcuno legge il suo valore di ritorno.

La syscall exec prende l'area di memoria di un processo e la rimpiazza con l'area di memoria di un program file. Quando un processo è in "running", esso inizierà la sua esecuzione da \_start, che è la routine in crt0.s che chiama il main.

#### Fork VS Vfork:

- Fork: non riesce se non si dispone di memoria sufficiente per duplicare un processo. Se ad esempio devo far eseguire una exec al processo figlio di una semplice "ls" e la memoria non è sufficiente, il programma non verrà eseguito.
- Vfork: Fa le stesse cose della fork ed ha gli stessi valori di ritorno ( $0 \rightarrow$  figlio,  $> 0 \rightarrow$  padre,  $< 0 \rightarrow$  Errore) ma ha migliori performance nel caso in cui utilizzo processi che compiono exec, poiché non copia inutilmente la memoria del padre nella creazione del figlio.

**Process Control Block:** È una semplice struct contenente:

- Process ID (PID)
- User ID (UID)
- Stato del programma (ready, running, ...)
- Stato della CPU del processo (registri)
- Informazioni di scheduling
- Informazioni sulla memoria (stack, tabella delle pagine)
- Informazioni di I/O (descrittori aperti)



### **Context Switch:**

La commutazione di contesto o context switch è un particolare stato del SO durante il quale avviene il cambiamento del processo correntemente in esecuzione sulla CPU. Questo permette a più processi di condividere una stessa CPU, permettendo di eseguire più programmi contemporaneamente.

Prima di tutto è necessario salvare lo stato del processo correntemente in esecuzione, tra cui il PC (Program Counter) ed il contenuto dei registri generali, in modo che l'esecuzione potrà essere ripresa in seguito. Queste informazioni sullo stato del processo vengono salvate nel PCB.

Successivamente lo scheduler sceglierà un processo tra quelli ready, in base alla propria politica di scheduling, e accederà al suo PCB per ripristinare il suo stato nel processore, in maniera inversa rispetto alla fase precedente.

### Quando arriva una interruzione:

- Salvo tutti i registri in una variabile (running) che conterrà il PCB correntemente in esecuzione
- Viene eseguito codice kernel e avviene il cambio di stack
- Carico nello stack pointer, lo stack pointer del kernel
- Scrivo in running il puntatore al PCB2

### Quando termina l'interruzione:

- Ri-popolo tutti i registri salvati prima
- Cambio lo stack pointer del processo corrente mettendo in running un qualsiasi valore di syscall
- IRET → Ritorno dall'interruzione ripristinando lo stato

## **CPU SCHEDULUNG**

Il CPU Scheduler è un modulo che ha lo scopo di decidere quale processo dovrà andare in running vedendo la coda di ready. Esso viene invocato quando un processo cambia stato da running a waiting o da running a ready o da waiting a ready, oppure quando termina, dunque quando vi è una richiesta di I/O da parte di un processo.

Esistono due **politiche di scheduling**:

- Preemptive: Da la possibilità di interrompere il processo correntemente in esecuzione a favore di un altro processo. Seleziona un processo e lo lascia in esecuzione per una certa quantità di tempo massima; se alla fine del tempo il processo è ancora in esecuzione, viene sospeso e lo scheduler seleziona un altro processo a cui attribuire l'uso della CPU. Una politica del genere richiede un interrupt timer alla fine dell'intervallo di tempo per restituire il controllo dalla CPU allo scheduler.
- Non Preemptive: Non da la possibilità di interrompere un processo in esecuzione. Essa seleziona un processo e lo lascia in esecuzione finché non si blocca, per attendere un evento o richiedere una risorsa, oppure finché non completa le sue operazioni rilasciando volontariamente la CPU.

**Dispatcher**: È un pezzo di funzione che ha lo scopo di impostare il controllo della CPU per il processo schedulato e caricare il suo stato salvato della CPU per iniziare o per proseguire l'esecuzione e modifica lo stato del processo in running. Compie due azioni:

- Passa in kernel mode e salva lo stato corrente del processo
- Passa in user mode mentre si ripristina il nuovo stato del processo.

Per quanto riguarda i costi, è molto oneroso per la cache in quanto ogni volta deve caricare in cache i dati di un processo.

### **Comportamenti dei processi – CPU/IO Bursts:**

Ogni processo può essere visto come un'alternanza di cicli da parte della CPU e cicli da parte dell'I/O, dunque l'esecuzione di un processo è costituita da due fasi:

- Esecuzione di istruzioni (CPU Burst = sequenza di operazioni svolte dalla CPU).

In questa fase vi è:

- Un intervallo di tempo in cui viene usata la CPU
- La CPU è nel collo di bottiglia
- L'I/O riposa

- Attesa di eventi o Operazioni di I/O (I/O Burst = sequenza di operazioni di I/O).

In questa fase vi è:

- Un intervallo di tempo in cui viene usato l'I/O
- L'I/O è nel collo di bottiglia
- La CPU riposa

In sistemi preemptive, i CPU burst sono molto corti.

### Metriche con le quali lo scheduler sceglie il nuovo processo in Running

Il comportamento dei processi viene monitorato in base ai seguenti indicatori:

- L'utilizzo della CPU: Frazione di tempo in cui la CPU viene usata dal processo
- Turnaround Time: Tempo che intercorre da quando viene lanciato il processo a quando esso termina
- Throughput: Numero di processi che terminano per unità di tempo (media di processi al secondo)
- Tempo di Attesa: Quanto tempo attende un processo nella coda di ready prima di entrare in running
- Tempo di Risposta: Tempo che ad esempi ci vuole per vedere un carattere sullo schermo dopo aver premuto un tasto

È logico che un buono scheduler vuole massimizzare l'utilizzo della CPU e il Throughput e minimizzare il Turnaround Time, il tempo di attesa e quello di risposta.

Per lo schedulamento dei processi, lo scheduler tiene conto del suo tempo di arrivo e di una lista di azioni che il processo svolge, tra cui la durata del CPU burst e dell'I/O burst.

### Politiche di Scheduling

Ricordiamo che:

- Il Tempo di Ammissione di un processo è il momento in cui esso viene sottomesso al sistema
- Il Tempo di Servizio di un processo è il tempo richiesto per completare la sua esecuzione
- Il Tempo di Turnaround ( $ta$ ) di un processo è il tempo impiegato per il completamento
- L' Average Waiting Time è la media complessiva tra i tempi di attesa dei processi

- **First Come First Served (FCFS)**

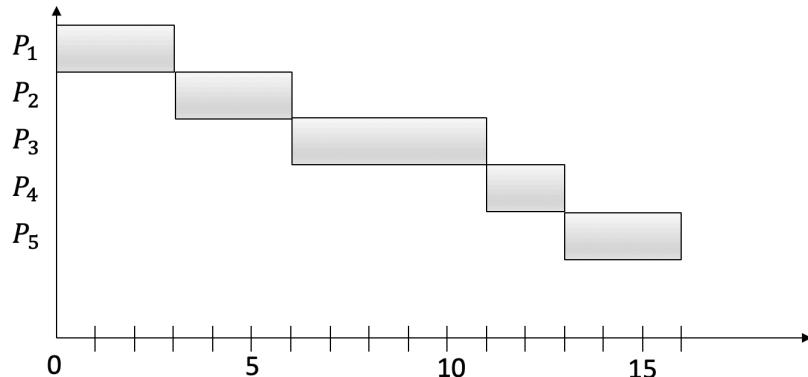
Questo è un algoritmo senza prelazione, senza priorità e statico. In questo caso i processi vengono schedulati nell'ordine in cui giungono al sistema, ovvero il primo ad essere eseguito è colui che per primo ha richiesto la CPU ed i processi successivi vengono schedulati con lo stesso criterio, non appena il processo completa le sue operazioni.

In parole povere, i processi ready sono organizzati come una coda di FIFO e i processi che richiedono la CPU vengono inseriti alla fine della coda.

Esempio:



Processo	$P_1$	$P_2$	$P_3$	$P_4$	$P_5$
Tempo di Ammissione (Arrival Time)	0	2	3	4	8
Tempo di Servizio (Burst Time)	3	3	5	2	3



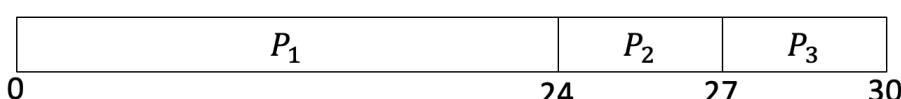
Si può notare dall'esempio che l'ordine di arrivo dei processi è  $P_1 \rightarrow P_2 \rightarrow P_3 \rightarrow P_4 \rightarrow P_5$  e che vengono eseguiti proprio nell'ordine con cui sono arrivati.

- $P_1$  arriva al tempo 0 e viene subito schedulato, in quanto primo e unico nella coda FIFO, rimane in esecuzione per 3 unità di tempo per poi rilasciare la CPU; nel frattempo sono arrivati anche  $P_2$  e  $P_3$ .
- $P_2$  arriva al tempo 2 ma va in esecuzione non appena termina  $P_1$  al tempo 3, quindi rimane in attesa per 1 unità di tempo. Al tempo 3 inizia la sua esecuzione che termina dopo 3 unità di tempo (al tempo 6). Il suo tempo di turnaround è pari a 4 unità di tempo ( $= 1$ (tempo attesa) + 3(sua durata)). Nel frattempo al già presente  $P_3$ , si è aggiunto alla coda il processo  $P_4$ .
- $P_3$  arriva al tempo 3, ma va in esecuzione non appena termina  $P_2$  al tempo 6, quindi rimane in attesa per 3 unità di tempo. Al tempo 6 inizia la sua esecuzione che termina dopo 5 unità di tempo, cioè al tempo 11. Turnaround Time = attesa + esecuzione =  $3 + 5 = 8$ . Nel frattempo al già presente  $P_4$  si è aggiunto nella coda anche  $P_5$ .
- $P_4$  arriva al tempo 4, ma va in esecuzione appena termina  $P_3$  al tempo 11, quindi rimane in attesa per 7 unità di tempo. Al tempo 11 inizia la sua esecuzione che termina dopo 2 unità di tempo al tempo 13. Turnaround Time =  $7 + 2 = 9$ . Nella coda è presente solo  $P_5$ .
- $P_5$  arriva al tempo 8, ma va in esecuzione appena termina  $P_4$  al tempo 13, quindi rimane in attesa per 5 unità di tempo. Inizia la sua esecuzione al tempo 13 e termina dopo 3 unità di tempo al tempo 16. Turnaround Time =  $5 + 3 = 8$

$$\text{Average Waiting Time} = \frac{(0 + 1 + 3 + 7 + 5)}{5} = 3.2$$

Esempio:

Processi	Burst Time	Arrival Time
$P_1$	24	0
$P_2$	3	1
$P_3$	3	2



- $P_1$  arriva al tempo 0 e viene subito schedulato e rimane in esecuzione fino al tempo 24. Nel frattempo nella coda sono già presenti  $P_2$  che arriva al tempo 1 e  $P_3$  che arriva al tempo 2. Waiting Time = 0
- $P_2$  arriva al tempo 1 ma deve attendere per 23 unità di tempo. Verrà schedulato al tempo 24 e terminerà al tempo 27. Nel frattempo nella coda abbiamo solo il processo  $P_3$ . Waiting Time = 23
- $P_3$  arriva al tempo 2 ma dovrà attendere per 25 unità di tempo. Verrà schedulato al tempo 27, subito dopo  $P_2$ . La coda ora è vuota. Waiting Time = 25

$$\text{Average Waiting Time} = \frac{(0 + 23 + 25)}{3} = 16$$

Supponiamo ad esempio che l'ordine di arrivo dei processi fosse  $P_2, P_3, P_1$ . In questo caso il waiting time di:

- $P_1 = 6$
- $P_2 = 0 \quad \Rightarrow \text{Average Waiting Time} = (6 + 0 + 3)/3 =$
- $P_3 = 3$

Il tempo medio di attesa in questo caso è molto più corto!

- **Shortest Job First (SJF)**

Questo algoritmo seleziona il processo in attesa che userà la CPU per minor tempo, in pratica è come se il CPU-BURST fosse la priorità.

Questo algoritmo eleva il throughput, ovvero i tempi di esecuzione dei vari processi, ma ha due problematiche:

- 1) È necessario conoscere in anticipo i tempi di esecuzione dei vari processi e visto che il SO non conosce a priori i tempi di servizio, occorre effettuare una stima dei CPU burst. Questa stima può essere effettuata conoscendo la "storia passata" in modo tale da poter fare una stima del "futuro".

Uso un filtro passa basso per effettuare la stima:

$$\hat{b}_{t+1} = \alpha b_t + (1 - \alpha)\hat{b}_t$$

Con:

$\hat{b}_{t+1}$  = Stima = predizione al tempo  $t + 1$

$\alpha$  = Coefficiente di decadimento (smussamento)

$b_t$  = Misurazione al tempo  $t$

$(1 - \alpha)\hat{b}_t$  = Predizione al momento precedente. Più è piccolo  $\alpha$ , più abbasso la frequenza del filtro passa basso

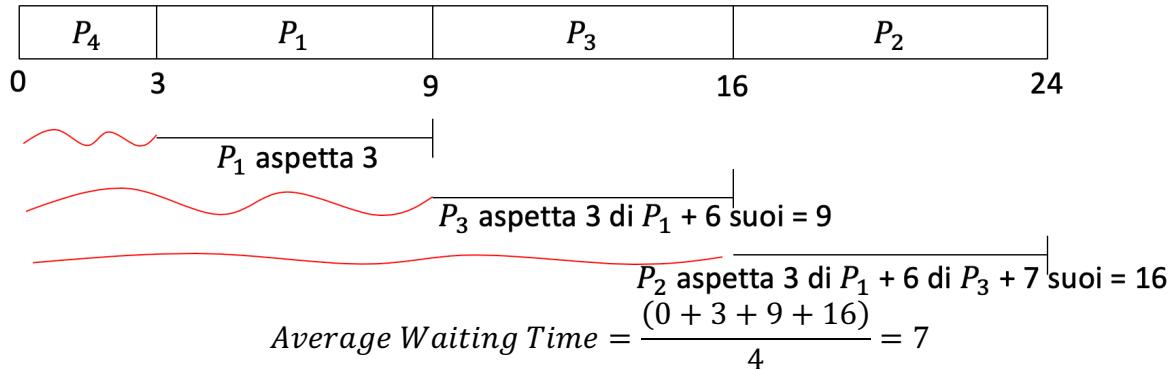
- 2) Possiede un potenziale problema di *starvation*, poiché è possibile che un processo rimanga in attesa troppo tempo prima di essere completato se vengono aggiunti continuamente piccoli processi alla coda dei processi pronti.

Esempio: Assumiamo che i processi arrivino al tempo 0



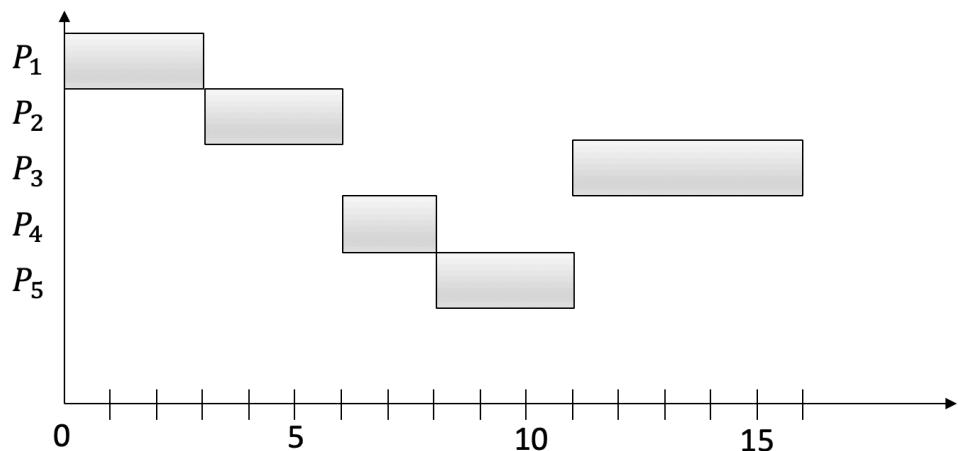
<u>Processi</u>	<u>Burst Time</u>
$P_1$	6
$P_2$	8
$P_3$	7
$P_4$	3

Ordine di Esecuzione:  $P_4 \rightarrow P_1 \rightarrow P_3 \rightarrow P_2$



Esempio:

<b>Processo</b>	$P_1$	$P_2$	$P_3$	$P_4$	$P_5$
<b>Tempo di Ammissione</b>	0	2	3	4	8
<b>Tempo di Servizio</b>	3	3	5	2	3



Come si nota dall'esempio, l'ordine di arrivo ed esecuzione possono non essere uguali.

- $P_1$  arriva al tempo 0 e viene subito schedulato, poiché è l'unico nella coda; rimane in esecuzione per 3 unità di tempo per poi rilasciare la CPU. Quando termina, nella coda sono presenti  $P_2$  e  $P_3$ . Il prossimo processo ad essere schedulato sarà  $P_2$  perché ha un tempo di servizio minore rispetto a  $P_3$ .
- $P_2$  arriva al tempo 2 ma va in esecuzione al tempo 3, rimane in attesa per 1 unità di tempo e al tempo 3 inizia la sua esecuzione che terminerà dopo 3 unità di tempo al tempo 6. Quando termina, nella coda saranno presenti  $P_3$  e  $P_4$  e dato che  $P_4$  è quello che il tempo di servizio minore, verrà schedulato per primo rispetto a  $P_3$ . Turnaround Time =  $1 + 3 = 4$
- $P_4$  arriva al tempo 4 e va in esecuzione al tempo 6, rimanendo in attesa per 2 unità temporali. Al tempo 6 inizia la sua esecuzione che terminerà dopo 2 unità di tempo al tempo 8. Quando termina, nella coda saranno presenti  $P_3$  e  $P_5$  e dato che quest'ultimo ha un tempo di servizio minore, verrà eseguito per primo.



Turnaround Time =  $2 + 2 = 4$

- $P_5$  arriva al tempo 8 e va subito in esecuzione, quindi la sua attesa è nulla. Inizia la sua esecuzione che terminerà dopo 3 unità temporali al tempo 11. Quando termina, nella coda sarà presente solo  $P_3$  che verrà schedulato.
- Turnaround Time =  $0 + 3 = 3$
- $P_3$  arriva al tempo 3 e va in esecuzione al tempo 11, rimanendo in attesa per 8 unità temporali. Al tempo 11 inizia la sua esecuzione che terminerà dopo 5 unità temporali al tempo 16. Turnaround Time =  $8 + 5 = 13$

$$Average Waiting Time = \frac{(0 + 1 + 2 + 0 + 8)}{5} = 2.2$$

- **Priority Scheduler (Non Preemptive)**

Uno scheduler con priorità assegna un numeretto ( $p$ ) ad un processo  $P$  che indica la priorità con cui schedulare il processo. Più la priorità è alta e più il numeretto sarà basso (massima priorità = 0), quindi tradizionalmente se  $p_1 < p_2$ ,  $p_1$  ha maggiore priorità di  $p_2$ . Questo tipo di scheduler risente del problema di *starvation* poiché processi a priorità bassa potrebbero non essere mai eseguiti.

Esempio:

Processi	Burst Time	Priority
$P_1$	10	3
$P_2$	1	1
$P_3$	2	4
$P_4$	1	5
$P_5$	5	2

$P_2$	$P_5$	$P_1$	$P_3$	$P_4$
0	1	6	16	18 19

*Waiting Time:*

- $P_2 = 0$
- $P_5 = 1$
- $P_1 = 6$        $\Rightarrow Average Waiting Time = (0 + 1 + 6 + 16 + 18)/5 = 8.2$  msec
- $P_3 = 16$
- $P_4 = 18$

- **Scheduler Preemptive**

Lo scheduler viene invocato o a seguito di interruzioni o quando viene eseguito il timer, ma comunque può decidere di non fare nulla.

Ogni processo ottiene una piccola unità di tempo CPU (un quanto cpu  $q$ , di solito 10/100ms) e se dopo questo tempo, sta continuando ad usare la CPU, viene sfrattato e posto nella coda di ready, altrimenti ne viene selezionato un altro a cui assegnare la CPU. Se il processo sfrattato viene posto alla fine → Round Robin (RR).



Lo scheduler viene invocato da:

- Richieste I/O
- Timer Interrupt

N.B.: Con  $N$  processi in ready e un quanto  $q$  di tempo, nessun processo potrà mai aspettare più di  $(N - 1) \cdot q$ .

- **Round Robin (RR)**

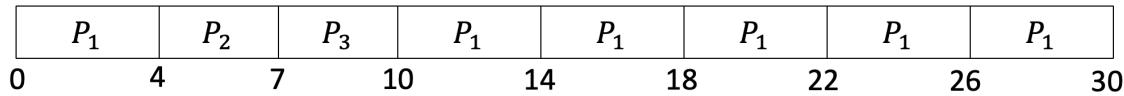
Ad ogni processo viene assegnato un quanto di tempo (o *time slice*), durante il quale al processo è assegnato l'uso della CPU. Per scandire i quanti, alla fine di ognuno di essi viene generato un timer interrupt.

Il Round Robin è facile da implementare: lo scheduler mantiene una coda di processi in stato ready e seleziona semplicemente il primo processo in coda e quando scade il quanto il processo viene messo in fondo alla lista. Assegnare un quanto troppo breve provoca troppi context-switch e peggiora l'efficienza della CPU (overhead troppo elevato), ma assegnarlo troppo alto può provocare tempi di risposta lunghi per richieste brevi.

Esempio:

Assumiamo che  $q = 4$

Processi	Burst Time
$P_1$	24
$P_2$	3
$P_3$	3



Considerazioni:  $q$  e il tempo di context-switch

- Più è piccolo  $q$  e più context switch posso ottenere e più scende il turnaround time
- Troppi context switch, poiché in quegli istanti invoco la cache
- Di solito conviene scegliere  $q$ , in modo tale che l'80% di CPU burst sia minore di  $q$ , perché nell'80% dei casi lo scheduler preemptive è come se non ci fosse nel sistema

- **Scheduling Multi-Livello**

Lo scheduling multilivello, o a code multiple, combina lo scheduling basato su priorità e quello round-robin per aumentare le prestazioni e i tempi di risposta. Esso utilizza varie code di processi ready, come ad esempio *foreground* (interattiva) e *background* (batch), e ad ogni classe viene associata una priorità. Alla coda con priorità più alta viene assegnato un quanto  $q$  più piccolo e alle code con priorità via via minore viene assegnato un quanto sempre più grande e ogni coda ha il suo algoritmo che la implementa (es: *foreground* - RR; *background* - FCFS). Insomma, la coda con priorità più bassa avrà il quanto più grande.

Abbiamo detto che vengono definite classi di priorità:

- I processi della classe più alta vengono eseguiti per un quanto;
- Quelli della classe più successiva per 2 quanti;
- Quelli della classe seguente per 4 e così via;
- Ogni qualvolta che un processo rimane in esecuzione per tutto il quanto, viene abbassato di una classe.

Se per esempio un processo  $A$  ha bisogno di 100 quanti, inizialmente gli viene assegnato un quanto, dopodiché viene schedulato un altro processo. Quando  $A$  riottiene la CPU, gli vengono assegnati due quanti, poi 4, poi 8, poi 16, poi 32, poi 64 quanti. In pratica ottiene la CPU per sette volte, anziché per 100 come sarebbe avvenuto con il round-robin.

- **Scheduling Multi Livello a code Feedback**



Ciò che lo contraddistingue dallo scheduling multilivello è l'uso delle priorità dinamiche, ciò permette allo scheduling di variare la priorità di un processo in modo tale da evitare starvation. Lo scheduler analizza il comportamento passato di un processo per modificare eventualmente la sua priorità. Quindi l'uso delle priorità dinamiche, consente ad un processo di muoversi tra le varie cose presenti.

- **Multiple Processor Scheduling**

Se ho più CPU, lo scheduling sarà più complesso.

- Homogeneous Processors: Tutti i core eseguono lo stesso set di istruzioni
- Asymmetric Multiprocessing: Solo una CPU può eseguire codice kernel, dunque il sistema è più semplice. Le interruzioni sono gestite da un solo core, quindi occorre parallelizzare per effettuare altre operazioni.
- Symmetric Multiprocessing (SMP): Ogni processore è self-scheduling. Tutti i processi sono in una coda di ready comune oppure ognuno potrebbe avere una sua coda privata.
- Processor Affinity: Forza l'esecuzione di un programma su un particolare core della macchina
  - Soft Affinity
  - Hard Affinity

- **Scheduling Real-Time**

In questo tipo di scheduling, il tempo gioca un ruolo fondamentale, poiché si devono rispettare delle scadenze, o altrimenti chiamate deadline. I sistemi real time sono classificati in:

- Hard Real-Time: Garantiscono che i compiti critici sono completati in un intervallo di tempo limitato, ovvero rispettano la deadline in maniera garantita. Lo scheduler deve avere la capacità di far rispettare le scadenze (deadlines).
- Soft Real-Time: Sono meno restrittivi e pur avendo scadenze da rispettare, se alcuni processi non lo fanno occasionalmente, sono tollerati. Rispettano la deadline in maniera probabilistica. In questo caso lo scheduler deve essere preemptive e basato su priorità.

Un sistema real-time deve poter reagire ad eventi che possono essere periodici, se si verificano ad intervalli regolari, o aperiodici, se si verificano in modo imprevedibile.

### Come Valutare uno Scheduler?

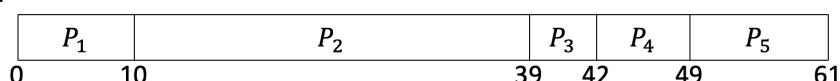
Scelto lo scheduler, lo stimo in modo deterministico, prendendo un particolare carico di lavoro predeterminato e definisco le prestazioni di ciascun algoritmo per quel carico di lavoro.

Consideriamo 5 processi che arrivano al tempo 0:

<u>Processi</u>	<u>Burst Time</u>
$P_1$	10
$P_2$	29
$P_3$	3
$P_4$	7
$P_5$	12

Per ogni algoritmo, calcolo il *minimum Average Waiting Time*:

- $FCS \rightarrow 28ms$ :



- Non Preemptive SJF → 13ms:

$P_3$	$P_4$	$P_1$	$P_5$		$P_2$	
0	3	10	20	32		61

- RR → 23ms:

$P_1$	$P_2$	$P_3$	$P_4$	$P_5$	$P_2$	$P_5$	$P_2$
0	10	20	23	30	40	50	52

Una stima del genere, non è efficace, infatti è preferibile usare:

### Queuing Models

Simulo i processi in base a una distribuzione di probabilità esponenziale, ragiono all'istante corrente, campiono e faccio simulazioni utilizzando la legge di Little.

La formula di Little è usata per stimare, tramite distribuzioni cumulative, quando succederà un evento dato un tempo corrente e ha validità per ogni algoritmo di scheduling. Il numero di processi che lasciano la coda, deve essere uguale al numero di processi che entrano in coda:

$$n = \lambda \cdot W$$

Con:

- $n$  = Lunghezza media della coda
- $W$  = Tempo medio di attesa in coda
- $\lambda$  = Tempo di arrivo medio in coda

Ad esempio, se in media arrivano 7 processi al secondo e 14 in coda, il tempo medio di attesa per un processo è di 2 secondi.

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  $\lambda$ . Tale legge si inserisce nello studio degli algoritmi mediante Queuing Models, assumendo che il workload sia conosciuto. In tal caso si effettuerà una stima probabilistica basata sulla distribuzione di CPU bursts e I/O bursts e sulla distribuzione dei tempi di arrivo dei processi.

## MAIN MEMORY

La CPU può accedere direttamente, solamente a memoria e a registri. L'accesso a registri avviene in 1 CPU clock o meno, mentre l'accesso a memoria può richiedere molti cicli, causando uno stall. La cache è posizionata tra memoria principale e registri della CPU e per garantire operazioni corrette, occorre proteggere la memoria. Lo scopo di una gerarchia di memoria, è quello di dare l'illusione di una memoria veloce e grande. La CPU fa riferimento alla memoria più veloce, la *cache*, quando deve accedere ad un'istruzione o ad un dato. Se il dato o l'istruzione non sono presenti in cache, viene prelevato dal livello successivo della gerarchia di memoria, che potrebbe essere una cache più lenta oppure la *memoria RAM*. Se l'istruzione o il dato non sono disponibili nemmeno al livello successivo della gerarchia, viene prelevato da un livello inferiore e così via.

### Caricamento di un Programma

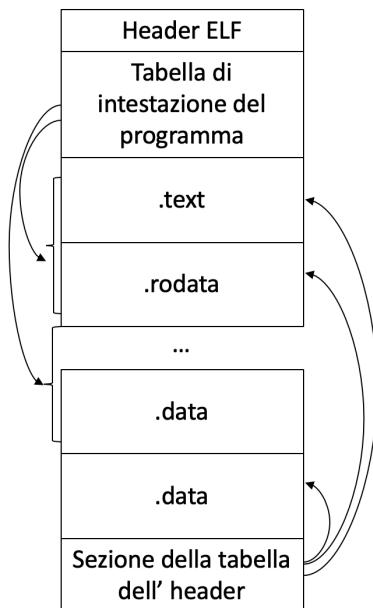
Cosa succede se faccio una exec? Un file binario (program file) viene aperto dall'OS e contiene tutte tutte le informazioni per creare un processo. L'header di un programma viene decodificato per estrarre informazioni riguardo:

- Segmenti di Memoria (.text, .bss, .data)
- Librerie dinamiche richieste (link a librerie esterne richieste)

Il dynamic Linker, guarda nel file .helf (nell'header del file binario) e vede quali sono le funzioni che vengono usate per collegarle al mio programma, mappa le librerie nello spazio del programma, regola i riferimenti a simboli definiti nelle librerie e ripete tale procedimento ricorsivamente.



## Struttura di un Header Elf



Gli eseguibili, sono file binari sul disco che contengono informazioni necessarie per caricare l'immagine del programma in memoria (codice del programma, inizializzazioni di variabili, ...).

## **Allocazione Statica e Dinamica della Memoria (Address Binding)**

Il binding è il processo tramite cui si associano gli indirizzi di memoria alle entità di un programma. Un'entità di un programma, come ad esempio una funzione o una variabile, ha un insieme di attributi e ognuno di essi ha un valore. Il binding consiste nello specificare il valore di un attributo. Per esempio, una variabile di un programma ha attributi come il nome, il tipo, la dimensione e un binding del nome specifica il nome della variabile. Se ho ad esempio delle istruzioni di jump, il dinamic linker nell'istruzione di salto rimpiazzerà gli indirizzi simbolici che do alla jump (jump L) con dei valori fisici.

Nota bene:

- Gli indirizzi nel file sorgente si chiamano simboli
- Gli indirizzi nel file ".o" non sono quelli veri, ma fungono da wild card, celle del vettore che saranno rimpiazzate con l'indirizzo vero e proprio in fase di caricamento.

Esistono 3 tipi di Binding:

- Durante la compilazione: Assegnazione di indirizzi fisici direttamente nell'eseguibile.
- Durante il caricamento: Utilizza label. Quando avviene il caricamento del programma, avverrà anche la traduzione da label (indirizzi logici) a indirizzi fisici.
- Durante l'esecuzione: è quello che fa il linker dinamico.

In generale possiamo distinguere tra binding statico (*early binding*), che avviene prima dell'esecuzione di un programma, e binding dinamico (*late binding*), eseguito durante l'esecuzione del programma.

Esempio: Come funziona in partita un Linker? Come creo una libreria dinamica?

#include <stdio.h>

```
typedef(void* MyFunctionPointer)(void);
void printfLoader(void);
```

```
MyFunctionPointer fakePrintf = printfLoader();
```

```
void printfLoader(){
    // fai cose
```



Luca's Mac



```

fakePrintf = <Function pointer load>;
(*fakePrintf)();

}

//All' inizio parte il loader, dalla volta dopo non partirà più

//Come creare una libreria dinamica:
void CoolFunction(int i){           // in file libCose.c
    printf("%d", i);
}

// compilazione ---> gcc -o cose.so -fPIC -shared libCose.c ---> Crea uno shared object
// In questo modo posso usarla come libreria esterna e farla usare a un mio programma
//     gcc -o prog prog.c -L. -lcose

```

### **Dynamic Linking**

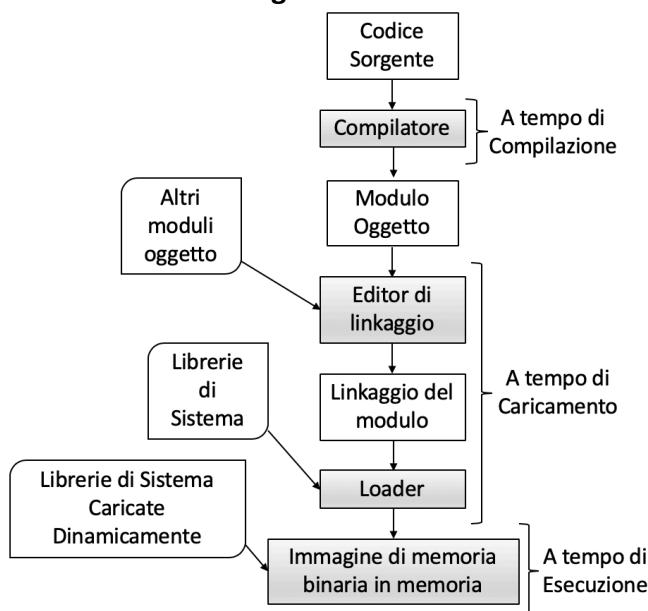
Il linker, LD su linux, ha il compito di collegare tutti gli indirizzi. Per vedere i simboli definiti in un file “.o”, basta fare nm file.o. Otterrò un file contenente *T* ed *U*, in cui *T* sono i simboli definiti nel file e gli *U* sono simboli non definiti ma usati.

Il linking statico e dinamico potrebbe non esserci all'esame!

Occorre tenere bene in mente la differenza tra *linker* e *loader*: il primo si occupa di collegare insieme i moduli per formare un programma eseguibile e il secondo invece carica un programma o una parte di esso in memoria per l'esecuzione.

- Nel Linking Statico, il linker collega tutti i moduli di un programma prima che cominci la sua esecuzione. Se più programmi usano lo stesso modulo di una libreria, ogni programma riceverà una propria copia del modulo. Ciò significa che diverse copie dello stesso modulo potranno essere presenti in memoria allo stesso tempo se i programmi che usano il modulo vengono eseguiti simultaneamente.
- Il Linking Dinamico viene invece eseguito durante l'esecuzione di un programma binario. Il linker viene invocato quando, durante l'esecuzione, si incontra un riferimento esterno non assegnato. Il linker collega il riferimento esterno e riprende successivamente l'esecuzione del programma. I vantaggi sono molti:
  - i moduli non invocati durante l'esecuzione non vengono linkati;
  - se un modulo è usato da più programmi è presente una sola volta in memoria;
  - se si aggiorna una libreria di moduli, il programma utilizzerà automaticamente la nuova versione del modulo.

## Fasi di Vita di un Programma

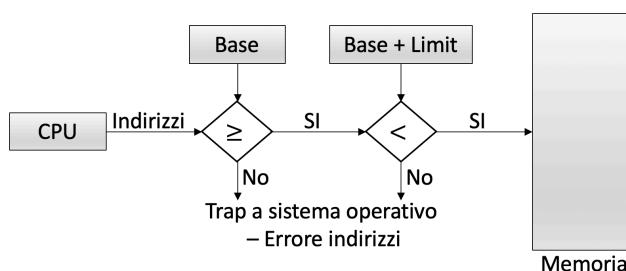


- 1) Quando nasce, nel codice sorgente gli indirizzi sono simboli
- 2) In fase di Compilazione, gli indirizzi divengono file oggetto, trasformandosi in Label
- 3) In fase di Caricamento, le Label vengono trasformate in indirizzi locali
- 4) In fase di Esecuzione, vengono risolte altre label

## Allocazione

Dopo aver tradotto gli indirizzi, il processo viene caricato in memoria, che viene vista dal processo come un grande array con molto spazio. I processi hanno bisogno di spazio contiguo, ma la memoria potrebbe aver bisogno di cambiare durante la vita di un processo. Vi sono anche altri due problemi importanti: la frammentazione, che avviene quando non ho abbastanza spazio contiguo per allocare un processo, e la protezione, ovvero la memoria tra due processi non deve sovrapporsi.

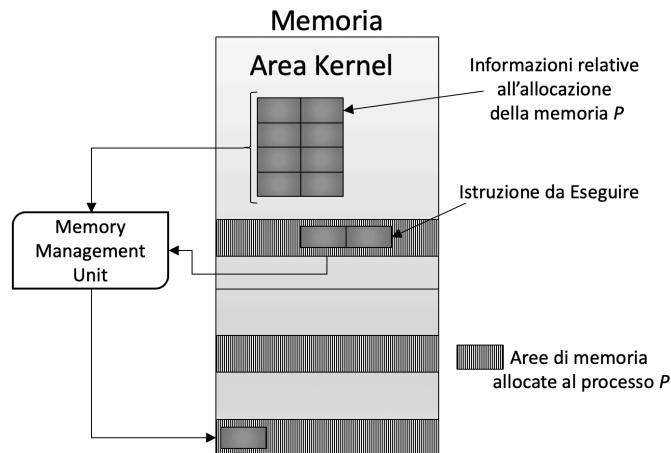
- **Protezione della memoria:** La memoria principale disponibile è di solito condivisa tra un certo numero di processi e, visto le numerose operazioni di swap-in e swap-out tra memoria e dischi, esiste una certa difficoltà nel tenere separati i vari processi in memoria. Occorre quindi assicurarsi che ogni processo abbia uno spazio di memoria separato che non interferisca con gli spazi di memoria degli altri processi. A tal fine occorre definire l'intervallo di indirizzi a cui un processo può accedere legalmente e garantire che possa accedere soltanto a questi indirizzi. La protezione della memoria è garantita da due registri di controllo della CPU, il *registro base*, che contiene l'indirizzo di partenza della memoria allocata, e il *registro limite*, che contiene la dimensione della memoria allocata al programma.



- **Indirizzo logico, fisico e traduzione di indirizzo**

Ricordo che:

- Indirizzo Logico = Indirizzo di istruzione o di un dato utilizzato dalla CPU
- Indirizzo Fisico = Indirizzo in memoria in cui sono presenti un'istruzione o un dato



L'insieme di indirizzi logici di un processo costituisce lo spazio di indirizzamento logico del processo stesso, mentre l'insieme di indirizzi fisici nel sistema costituisce lo spazio di indirizzamento fisico del sistema stesso.

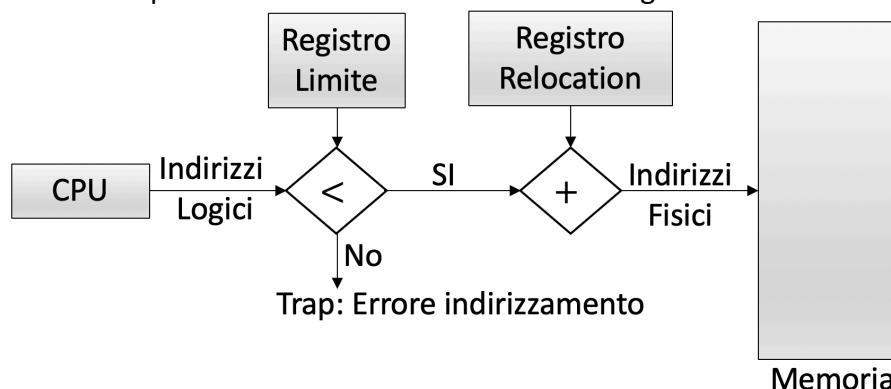
Il kernel memorizza le informazioni relative alle aree di memoria allocate al processo  $P$  in una tabella e le rende disponibili alla *memory management unit (MMU)*, circuito situato tra CPU e memoria. La CPU invia gli indirizzi logici di ogni dato o istruzione utilizzati nel processo alla MMU, la quale utilizzerà le informazioni relative all'allocazione della memoria, memorizzate nella tabella, per calcolare i corrispondenti indirizzi fisici. Questo indirizzo è chiamato indirizzo di memoria effettivo del dato o dell'istruzione e la procedura per calcolare tale indirizzo a partire da un logico è chiamata traduzione dell'indirizzo.

#### Metodi per fare MMU:

- Reallocation and Limit (HW): Ci occorrono solamente due registri e un adder.  

$$\text{Indirizzi Fisici} = \text{Reallocation} + \text{Indirizzi Logici}$$

In questa tipologia, la CPU deve verificare ogni accesso a memoria generato in user mode per essere sicura che sia al di sotto del registro limite e ogni programma partirà dall'indirizzo 0. Per ottenere l'indirizzo fisico, basta quindi prendere l'indirizzo logico, sommarlo al mio indirizzo fisico e se va oltre un comparatore ( $<$ ) farà avvenire la trap. N.B.: La memoria deve essere contigua.



- Contiguous Allocation: Con questa tipologia ciascun processo è contenuto in una singola sezione contigua della memoria e potrei anche ricompattare la memoria, copiando la memoria di ogni processo nella parte alta.  
L'indirizzo base, conterrà il valore del più piccolo indirizzo fisico dei processi e l'indirizzo limite conterrà un range di indirizzi logici, i quali devono essere minori dell'indirizzo limite.
- Multiple Partition Allocation: Divido la memoria per ogni processo e ne do un pezzo di egual dimensione ad ognuno. Posso implementarla in due modi:

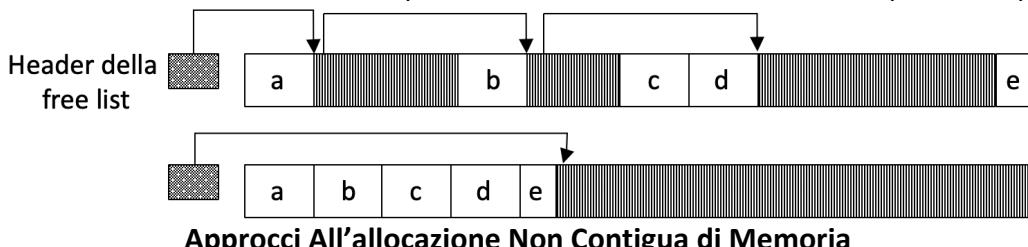
- 1) A dimensione Fissa: Il numero di programmi è limitato dalla memoria.  
Problematiche: Frammentazione interna e spreco di memoria.
- 2) A dimensione Variabile: L'ammontare dei programmi in memoria può variare.  
Problematiche: Frammentazione esterna poiché potrebbe esserci abbastanza memoria libera ma non contigua per allocare un nuovo processo.

Come assegnare lo spazio ad un processo?

- First-Fit: Alloco il primo spazio libero che sia abbastanza grande per contenerlo
- Best-Fit: Alloco il più piccolo spazio libero che sia abbastanza grande. Per fare ciò però dovrei scandire prima tutta la memoria.
- Worst-Fit: Alloco lo spazio più largo che trovo.

### Compattazione di Memoria

Per evitare o limitare la frammentazione esterna di memoria, occorre utilizzare la compattazione. In questo approccio, tutte le aree di memoria libere vengono unite per formare un'unica area di memoria libera. Questo risultato può essere ottenuto impacchettando tutte le aree di memoria allocata verso un'estremità della memoria stessa. La compattazione è possibile solamente se la riallocazione di memoria è dinamica, e il procedimento viene effettuato a tempo di compilazione.



### Approcci All'allocatione Non Contigua di Memoria

- **Segmentazione**

Nella segmentazione, un programma è diviso in vari segmenti di diversa dimensione ed ogni segmento rappresenta un'entità logica del programma come una funzione o una struttura dati. La segmentazione permette una facile condivisione del codice, dei dati e delle funzioni di un programma proprio perché questi sono organizzati in segmenti.

In pratica, un processo viene visto come un insieme di segmenti che però sono sparsi in memoria, per questo motivo gli indirizzi logici e indirizzi fisici non corrispondono. Ogni indirizzo logico è rappresentato nella forma  $(si, bi)$ , il primo rappresenta l'ID di un segmento e il secondo è lo scostamento in byte all'interno del segmento.

La tabella dei segmenti è situata in memoria e contiene:

- Base: Contiene l'indirizzo fisico in cui il segmento risiede in memoria
- Limit: Specifica la lunghezza del segmento

Un numero di segmento  $s$  è legale se  $s$  è minore della STLR (Segment table length register).

Nella tabella dei segmenti vengono aggiunti anche bit di protezione e ogni entry della tabella associa:

- Bit di validità = 0 → Segmento illegale
- Privilegi di lettura/scrittura/esecuzione

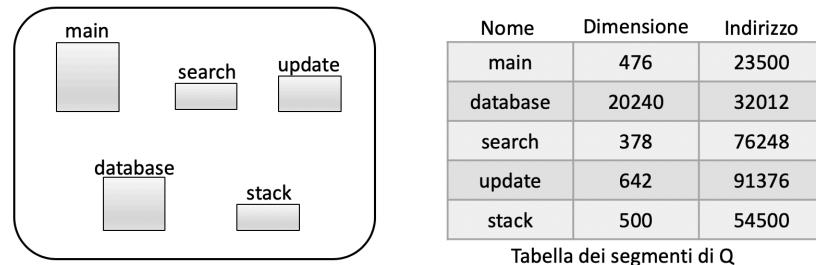
Offre numerosi vantaggi:

- 1) Semplifica la gestione di strutture dati che crescono, poiché l'OS può anche espandere il segmento
- 2) Permette la modifica e la ricompilazione indipendente dei programmi, senza richiedere che si rifaccia il link
- 3) Si presta alla condivisione dei processi, poiché è possibile allocare un programma di utilità in un segmento e renderlo accessibile ad altri processi



- 4) Si presta alla protezione, poiché un segmento può contenere un insieme ben definito di programmi o dati a cui posso assegnargli privilegi.

Esempio: Consideriamo la procedura `get_sample` del segmento `update`. Supponiamo che questa procedura abbia numero di byte 232.



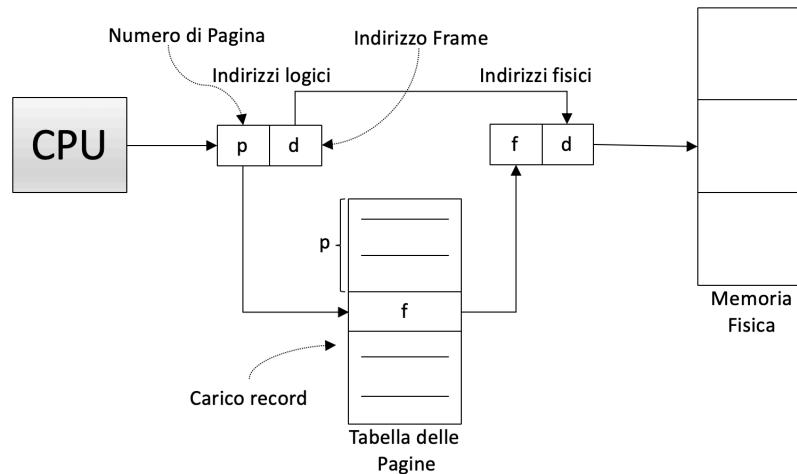
Per ottenere il suo indirizzo di memoria effettivo, occorre sommare  $91376 + 232 = 91608$ .

#### • Paginazione

Nella paginazione, ogni processo viene diviso in parti di dimensioni fissata chiamate pagine, la cui dimensione viene definita dall'architettura del sistema ed è una potenza di 2. La memoria può memorizzare un numero intero di pagine e viene partizionata in aree o blocchi di memoria detti frame, che hanno la stessa dimensione di una pagina.

Gli indirizzi generati dalla CPU, vengono divisi in:

- Numero di pagina ( $p$ ), utilizzato come un indice all'interno della tabella delle pagine, che contiene indirizzo di base per ogni pagina nella memoria fisica
- Offset di pagina ( $d$ ), che combinato con indirizzo base per definire l'indirizzo fisico in memoria che verrà inviato alla memory unit.



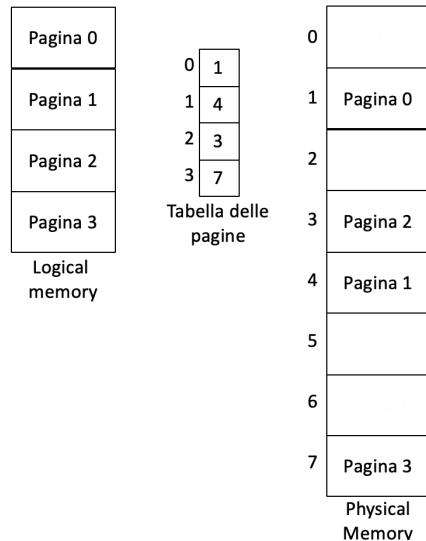
Per implementare il paging, mi occorrono due strutture hardware: 2 registri nella CPU.

- Page-table base register (**PTBR**) = Dove inizia la tabella delle pagine
- Page-table length register (**PTLR**) = Size della tabella delle pagine

Ogni area di memoria è esattamente della stessa dimensione della pagina, per cui non si crea frammentazione esterna nel sistema. La frammentazione interna può crearsi poiché all'ultima pagina di un processo viene allocato un frame della dimensione di una pagina, anche se è più piccolo della dimensione di una pagina.

Un ulteriore problema è che ogni accesso in memoria mi costerà il doppio (Doppia fase di fetch) e la soluzione a tale problema sta nell'utilizzare il **TLB**: piccola cache associativa in cui registro l'array di memoria in cui accedo più spesso. Tale TLB è una sorta di mappa (numero di pagina:valore).

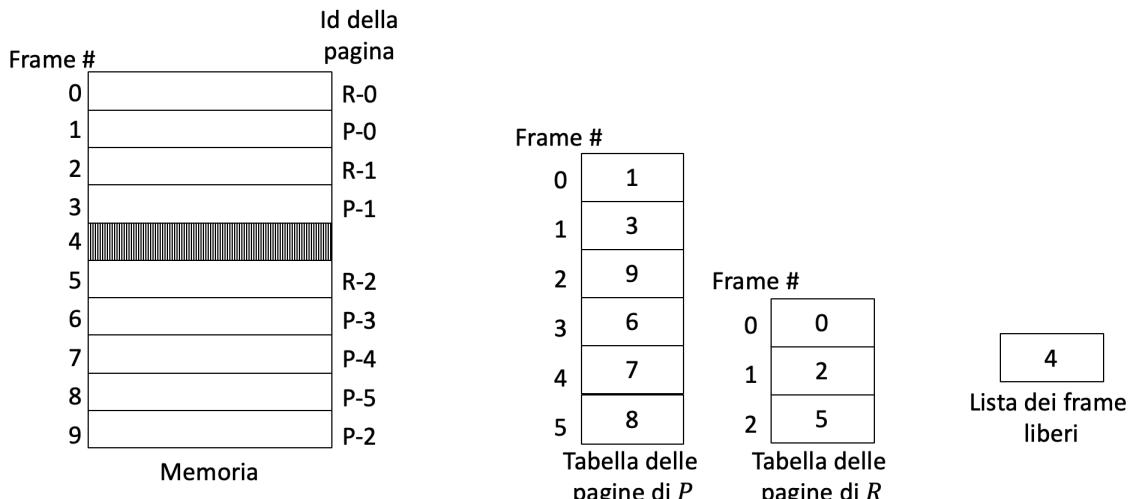
Esempio: Un processo vede 4 pagine di memoria



Esempio: Si considerino due processi  $P$  ed  $R$  in un sistema che usa pagina di dimensione 1KB. I byte in una pagina sono, quindi numerati da 0 a 1023. Il processo  $P$  ha indirizzo di avvio 0 e dimensione 5500 byte. Dunque avrà 6 pagine numerate da 0 a 5. Le pagine 0, 1, 2, 3 e 4 avranno 1024 byte, per un totale di 5120, mentre la pagina 5 avrà solo 380 byte e i restanti saranno sprecati, causando frammentazione interna. Se un dato ha indirizzo 5248, ovvero  $5 \cdot 1024 + 128$ , vuol dire che si trova nella pagina 5 e la MMU vedrà il suo indirizzo come la coppia (5, 128).

Il processo  $R$  ha dimensione 2500 byte, dunque avrà 3 pagine, numerate da 0 a 2. Le pagine 0 e 1 avranno 1024 byte e la pagina 2 avrà 452 byte.

Visto che i frame hanno la stessa dimensione delle pagine, allora avranno dimensioni pari a 1KB. Il computer ha una memoria di 10KB, per cui i frame sono numerati da 0 a 9. 6 frame saranno occupati dal processo  $P$  e 3 dal processo  $R$ .



Le pagine contenute nei frame sono mostrate come  $P-0, \dots, P-5$  per  $P$  e come  $R-0, \dots, R-2$  per  $R$ .

La lista dei frame liberi contiene un solo elemento, perché solo il frame 4 risulta libero.

La tabella delle pagine di  $P$  riporta il frame allocato a ogni pagina di  $P$  e analogo discorso vale per la tabella delle pagine di  $R$ .



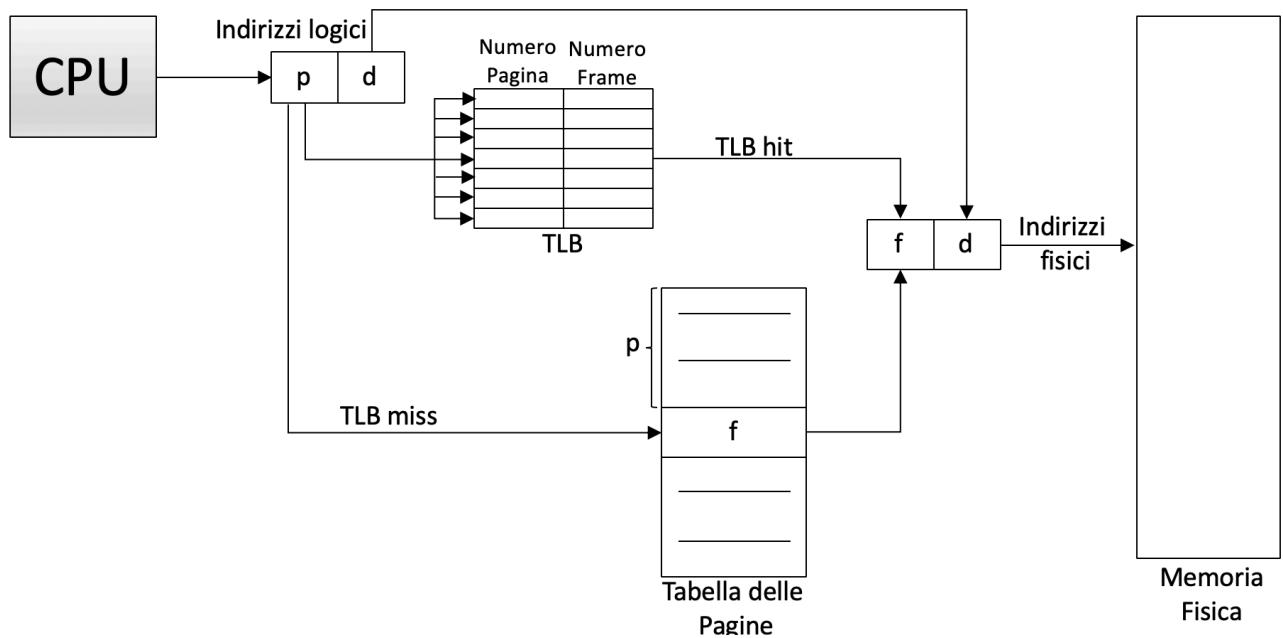
Supponiamo che ad esempio il processo  $P$  usi l'indirizzo logico (5,128) durante la sua esecuzione, questo sarà tradotto nell'effettivo indirizzo di memoria usando la seguente equazione:

$$\begin{aligned}
 & \text{indirizzo di memoria effettivo di (5, 128)} \\
 & = \text{indirizzo di avvio del frame allocato a 5} \\
 & + \text{numero dei byte, cioè } \#8 + 128 = 8 \cdot 1024 + 128 = 8320
 \end{aligned}$$

**TLB**

Il TLB (*Translation Look-aside Buffer*) è una memoria cache associativa molto veloce, anche più della RAM, contenuta nella MMU che permette di velocizzare la traduzione di indirizzi. Esso contiene porzioni della tabella delle pagine, cioè solo pochi elementi di quest'ultima, in particolare contiene gli elementi che sono stati usati più di recente. Di ogni elemento memorizza alcune informazioni principali, come il numero di pagina, il numero di frame, il bit di modifica e il campo per la protezione.

Funzionamento: Quando alla MMU arriva un indirizzo virtuale da tradurre dalla CPU, l'hardware controlla prima se il relativo numero di pagina è presente nel TLB, confrontandolo in parallelo (== simultaneamente) con tutti gli elementi. Se presente, utilizza il frame nel TLB, altrimenti legge dalla tabella delle pagine l'entry corretta e la inserisce nel TLB, rimuovendo un vecchio dato. Successivamente genera l'indirizzo fisico corretto.



#### Tempo di Accesso Effettivo a Memoria (Per Esame) - EAT

Il calcolo del tempo effettivo a memoria si trova attraverso una formula che rappresenta nient'altro che una media pesata.

$$EAT = \underbrace{(1 + \varepsilon)\alpha}_{\substack{\text{prob. che sia} \\ \text{in memoria}}} + \underbrace{(2 + \varepsilon)(1 - \alpha)}_{\substack{\text{prob. che non sia} \\ \text{in memoria. Pago 2 volte.}}}$$

$\varepsilon$  = TLB lookup = rapporto tra il tempo di ricerca nel TLB e il tempo di RAM (< 1.0)

$\alpha$  = Percentuale di Successo = Percentuale di volte in cui un numero di pagina viene trovato nei registri associativi.

Nel caso in cui i dati dell'esercizio presentino anche la probabilità di page-fault ( $p_{fault}$ ) o page-hit ( $p_{hit}$ ), la formula diventa:

$$\begin{aligned}
 EAT &= p_{hit}(T_{TLB} + T_{RAM}) + p_{fault}(2T_{TLB} + 2T_{RAM}) \\
 \text{N.B.: } p_{hit} + p_{fault} &= 1
 \end{aligned}$$

### Esempio:

Considera  $\alpha = 80\%$ ,  $\varepsilon = 20\text{ns}$  per la ricerca nel TLB, 100ns per l'accesso a memoria.

$$EAT = 0.80 \cdot 120 + 0.20 \cdot 220 = 140\text{ns}$$

### Esempio:

Considera  $\alpha = 99\%$ ,  $\varepsilon = 20\text{ns}$  per la ricerca nel TLB, 100ns per accesso a memoria.

$$EAT = 0.99 \cdot 120 + 0.01 \cdot 220 = 121\text{ns}$$

### Esempio:

Considera che il tempo di accesso medio ad una pagina e quello di accesso al TLB siano definiti rispettivamente come  $EAT = 120\text{ns}$  e  $T_{TLB} = 1\text{ns}$ , mentre la probabilità di page fault sia pari a  $p_{fault} = 1e - 3$ . Calcola il tempo di un ciclo di lettura/scrittura  $T_{RAM}$ .

$$EAT = p_{hit}(T_{TLB} + T_{RAM}) + p_{faul}(2T_{TLB} + 2T_{RAM})$$

Notiamo che vale la relazione  $p_{hit} + p_{fault} = 1$  e descrivendo una probabilità, la loro somma deve essere unitaria. Sostituisco i valori numerici e calcolo  $T_{RAM}$ , ottenendo:

$$T_{RAM} = \frac{119001}{1001} \approx 118.88\text{ns}$$

### Esempio:

Calcola la probabilità di page fault considerando che un accesso al  $TLB$  impieghi 1ns, un ciclo di lettura scrittura impieghi 100ns ed il tempo di accesso medio alla memoria sia pari a 110ns.

Pongo l'hit ratio ( $p_{hit}$ ) =  $p$ , quindi avrei il seguente tempo di accesso effettivo in memoria:

$$EAT = [p \cdot (1 + 100) + (1 - p) \cdot (2 + 200)]\text{ns}$$

Sapendo che il tempo di accesso medio è di 110ns, derivo che:

$$[p \cdot (1 + 100) + (1 - p) \cdot (2 + 200)]\text{ns} = 110\text{ns}$$

Basta quindi risolvere l'equazione in  $p$  per trovare la probabilità desiderata come  $q = 1 - p$ .

■

## Paging Protection

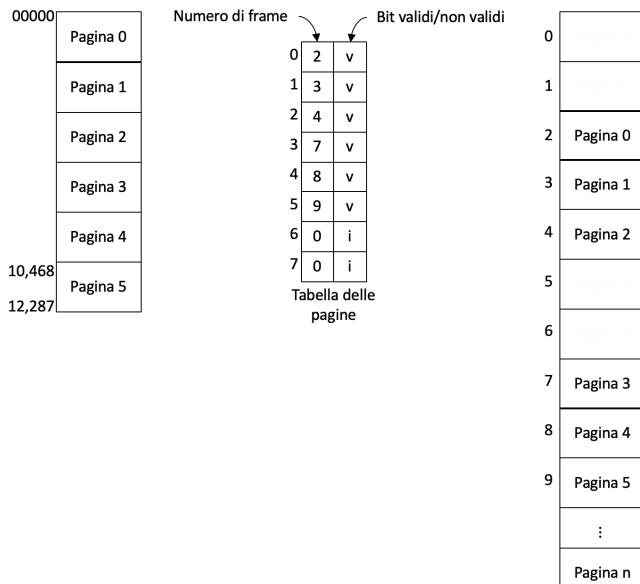
La protezione della memoria è implementata attraverso un bit che indica se in quella zona è possibile solamente la lettura o anche la scrittura. È possibile anche aggiungere più bit per indicare se è possibile anche l'esecuzione. Viene anche associato un bit di validità per ogni entry della tabella:

- Valid → indica che la pagina associata è nello spazio di indirizzamento logico del processo e questa è una pagina legale
- Invalid → indica che la pagina non è nello spazio di indirizzamento logico del processo
- È possibile utilizzare  $PTLR$  → Page table length register

Ogni violazione viene vista come una trap per il kernel.

I bit in avanzo vengono usati come permessi per le pagine (flag) e vengono gestiti dall'MMU. Ogni processo ha una sua individuale copia della tabella delle pagine e vi saranno quindi pagine valide e pagine non valide.





Attraverso la paginazione, posso fare in modo che due processi condividano memoria, mappando lo stesso frame su due tabelle diverse.

### Shared Pages

Esistono due tipi di shared pages:

- Shared Code  
Una copia in sola lettura viene condivisa tra processi, come fanno ad esempio gli editor di testo, compilatori, .... Questa tipologia è molto utile per la comunicazione interprocesso se è abilitata la condivisione di pagine in lettura-scrittura.
- Private Code and Data  
In questo caso, ogni processo mantiene una copia separata di codice e dati e le pagine per il codice privato e dati potrebbero apparire in qualunque posizione nello spazio di indirizzamento logico.

### Gerarchia delle tabelle delle pagine

La tabella delle pagine potrebbe essere troppo grande e la soluzione a tale problema è creare una gerarchia, in modo tale che invece di gestire un solo indice, suddivido le pagine in strati.

Occorre suddividere lo spazio di indirizzi logici in pagine multiple e una semplice tecnica per implementarlo è fare una paginazione a due livelli.

Pro: Ridulo la dimensione della tabella delle pagine; Contro: incremento il tempo di accesso effettivo in memoria, che ora mi costa 3 accessi.

### Swapping

Nei sistemi multiutente e multiprogrammati normalmente non vi è abbastanza memoria principale per mantenere tutti i processi attivi. Occorre quindi trasferire alcuni dei processi dalla memoria sul disco per poi successivamente introdurli in memoria. Lo swapping consiste nel caricare interamente in memoria ogni processo, eseguirlo per un certo tempo e spostarlo nuovamente sul disco in un'area riservata chiamata "area di swap". Le operazioni svolte generalmente sono:

- Swap-in quando si porta un processo dal disco in memoria
- Swap-out quando si porta un processo dalla memoria sul disco

Lo swapping potrebbe creare a lungo andare molti buchi all'interno della memoria e impatta molto sul tempo del context switch e una delle soluzioni è la compattazione, spostando tutti i processi il più indietro possibile.

Se a un processo serve altra memoria, posso decidere di prendere una pagina, buttarla e darla al processo in questione.



Esempio: 100MB di processi swappati su hard disk con velocità di trasferimento di 50MB/sec

- Tempo di Swap-Out = 2000ms
- Tempo totale di context switch = 4000ms = 4 secondi

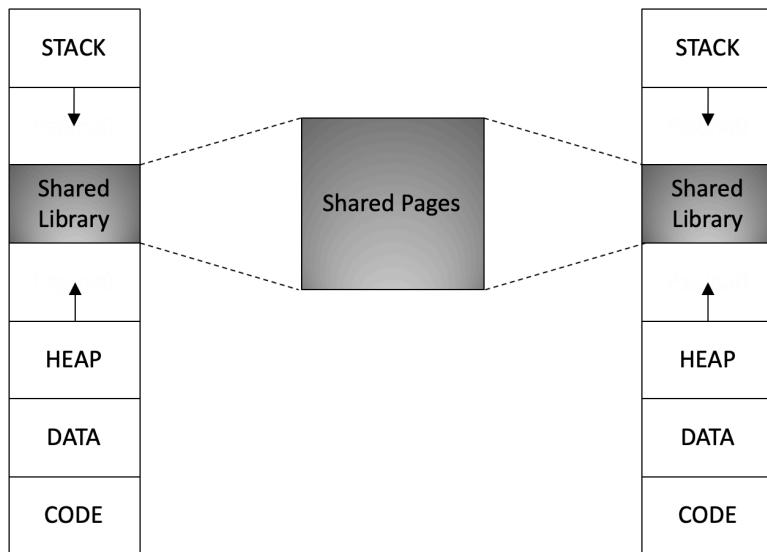
## Virtual Memory

La memoria virtuale è una parte della gerarchia della memoria composta da una memoria e da un disco. Durante l'esecuzione di un processo, alcune componenti del suo spazio di indirizzamento (codice e dati) si trovano in memoria, mentre altre risiedono su un disco e vengono caricate in memoria solo quando necessario durante l'esecuzione del processo. Questa soluzione fa sì che la richiesta totale di memoria di un processo possa superare la dimensione della memoria del sistema e permette anche che un maggior numero di processi risiedano in memoria contemporaneamente, perché ognuno di loro occupa meno memoria della propria dimensione. Lo spazio di indirizzamento virtuale di solito inizia all'indirizzo 0 e viene allocato fino alla fine dello spazio necessario. La memoria fisica viene organizzata in frames e l'MMU mappa indirizzi logici in fisici.

Il kernel implementa l'illusione di una memoria più grande di quella reale tramite una combinazione di mezzi hardware e software, attraverso l'MMU (HW) e il gestore della memoria virtuale (SW).

Essa si basa sul modello di allocazione di memoria non contigua, dunque:

- Le parti di un processo (librerie dinamiche, funzioni, ...) possono essere caricate in aree di memoria non adiacenti
- Le librerie a sistema vengono condivise mappandole in spazi di indirizzamento virtuale
- L'indirizzo di ciascun operando o istruzione di un processo è un indirizzo logico e la MMU lo traduce nell'indirizzo di memoria effettivo in cui si trova.



Quando un processo viene mandato in esecuzione, il gestore della memoria virtuale carica solo quella porzione che contiene l'indirizzo di start del processo, cioè l'indirizzo dell'istruzione con cui la sua esecuzione comincia.

Virtual memory può essere implementato tramite:

- **Paginazione su Richiesta (*Demand Paging*)**: Ciascuna porzione di indirizzamento è chiamata pagina e tutte le pagine hanno egual dimensione, potenza di due. L'idea di base è la seguente: portare una pagina in memoria solo al momento del primo riferimento a una locazione appartenente alla pagina stessa. Quando la CPU fa riferimento a una locazione in un'altra pagina, l'OS deve sospendere il processo, il quale viene tolto dalla CPU e messo in stato di *waiting for page*, portare la pagina in memoria e quando possibile far proseguire l'esecuzione del processo.
  - Se occorre la pagina → La restituisco
    - Se la referenza non è valida → Abort
    - Se non è in memoria → La carico in memoria

Pure Demand Paging = Caso estremo: lancio il processo senza nessuna pagina in memoria

#### Performance in Demand Paging

Caso peggiore, accesso a pagina non in RAM:

- Context switch a OS
- Verifica se la referenza è legale
- Attesa finché non viene servita la richiesta di read
- Mentre in attesa, alloca CPU per altri utenti
- Ricezione di interrupt dal I/O subsystem del disco → Context switch to OS

Misurazione Performance-EAT:

$$EAT = (1 - p) \cdot \text{accesso a memoria} + p(\text{page fault overhead} + \text{swap page in})$$

Il tasso di errore di paginazione →  $0 \leq p \leq 1$

- Se  $p = 0$  → No page fault
- Se  $p = 1$  → Ogni referenza è un fault

#### Esempio:

Tempo di accesso a memoria = 200ns; Tempo medio servizio page-fault = 8ms

$$EAT = (1 - p) \cdot 200 + p(8ms) = (1 - p \cdot 200 + p \cdot 8000000) = 200 + p \cdot 7999800$$

Se un accesso su 1000 provoca page-fault, allora  $EAT = 8.2$  microsecondi.

Se desideriamo un peggioramento delle performance minore del 10%:

$$220 > 200 + 7999800 \cdot p$$

$$p < 0.000025 \rightarrow (1/400000)$$



## Ottimizzazione Demand Paging

- **Swap Space:** Consiste in aree disco senza un file system (raw mode).
    - All'avvio occorre copiare l'intera immagine del processo nello spazio swap, durante il caricamento del processo
    - In esecuzione viene effettuato swap-in e swap-out nello/dallo spazio swap
    - Quando avviene swap-out la memoria sarà di sola lettura
  - **Copy on Write (COW):** Durante la *fork*, viene replicata solo la tabella delle pagine del processo e viene settato ad 1 un flag “*trap\_on\_write*” sulle pagine. Durante la scrittura, viene generato trap: il frame viene copiato e viene utilizzato un contatore sui frames per abilitare fork multiple.
- Esempio:
- Pre-Fork:
- Il processo 1 fa fork e genera il processo 2
  - La tabella delle pagine viene copiata e viene settato il bit *trap\_on\_write*
  - I flags non vengono copiati
- Dopo-Fork:
- Quando il processo 1 scrive sulla pagina *c*, viene generata trap
  - Il frame “C” viene copiato e viene aggiornato il valore nella tabella delle pagine del processo 1
- **Free Frames:** Il sistema detiene la lista dei frames liberi e le pagine libere vengono inizializzate a 0, altrimenti un altro processo potrebbe leggere ogni dato di un altro processo già “morto”.

- **Segmentazione su Richiesta (Demand Segmentation)**

È molto simile alla paginazione su richiesta, ma viene implementato da sistemi operativi in cui l'hardware disponibile non è sufficiente per implementare la paginazione. La tabella dei segmenti contiene un bit di validità atto a verificare se il segmento si trova già nella memoria fisica oppure no. Se il segmento non è presente nella memoria fisica, avviene un fault e tale segmento viene portato in memoria.

## Politiche di Sostituzione delle Pagine

Il gestore della memoria, si occupa anche di conservare le informazioni utili per la protezione della memoria. Queste informazioni vengono memorizzate in un campo della tabella delle pagine, insieme ai privilegi di accesso, dunque se avviene un acceso con modalità non consentita verrà generato un interrupt.

Il gestore dovrà decidere indipendentemente:

- Quale pagina deve essere sostituita quando si verifica page fault e non ci sono frame liberi in memoria
- Periodicamente quanta memoria, cioè quanti frame allocare per ciascun processo.

Quando si verifica page fault, il gestore della memoria dovrà procedere sostituendola attraverso un algoritmo di sostituzione delle pagine.

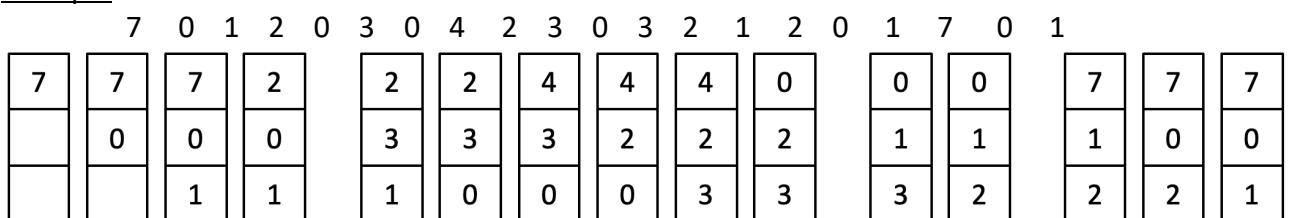
L'obiettivo di una politica di sostituzione di pagine è quello di sostituire le pagine che non si useranno nell'immediato. Per fare ciò ci basiamo sul concetto di *stringa dei riferimenti delle pagine*, ovvero una sequenza di pagine costruita monitorando l'esecuzione di un processo formando una sequenza di numeri di pagine a cui esso ha fatto accesso durante la sua esecuzione. Per convenienza associamo una stringa di riferimento dei tempi  $t_1, t_2, t_3, \dots$  a ogni stringa dei riferimenti alle pagine, in modo tale che al  $k$ -esimo processo venga associato il tempo  $t_k$ .



Esistono 3 algoritmi di sostituzione:

- Politica Ottimale: Consiste nel sostituire le pagine in modo tale che il numero totale di fault durante l'esecuzione di un processo sia il minimo possibile. In pratica vengono sostituite solo le pagine che non si useranno per il periodo di tempo più lungo. Questa politica naturalmente è impossibile da implementare, in quanto il sistema non ha nessun modo di sapere quando farà riferimento a ciascuna delle pagine, non potendo conoscere il comportamento futuro di un processo.
- Politica First-In-First-Out (FIFO): Ad ogni page fault, sostituisce la pagina che è stata caricata in memoria prima di ogni altra pagina del processo, cioè quella che risiede in memoria da più tempo. In pratica, il sistema operativo mantiene una lista di tutte le pagine correntemente in memoria, dove la pagina di testa è la più vecchia e la pagina in coda è quella arrivata più recente. Durante un page fault, la pagina in testa viene rimossa, anche se è la più utilizzata, ed è proprio per questo motivo che viene raramente utilizzato questo algoritmo nella sua forma più pura.

Esempio: 3 Frames



→ 12 Fault

Esempio: 3 Frames

2, 3, 2, 1, 5, 2, 4, 5, 3, 2, 5, 2



⇒ 6 Page Fault

- Politica Least Recently Used (LRU): Ad ogni page fault, viene sostituita la pagina utilizzata meno di recente con la pagina richiesta. In pratica, viene scaricata la pagina usata meno di recente. Questa politica è realizzabile ma non conveniente poiché per implementarla completamente è necessario mantenere delle liste concatenate di tutte le pagine in memoria, con la pagina usata più di recente in testa alla lista e la difficoltà sta nel fatto che la lista va aggiornata ad ogni riferimento alla memoria.

Esempio: Tutte e 3 le politiche insieme

Stringa dei riferimenti delle pagine:

0, 1, 0, 2, 0, 1, 2, ...

Stringa dei tempi dei riferimenti:

$t_1, t_2, t_3, t_4, t_5, t_6, t_7, \dots$

		OTTIMALE			FIFO			LRU			
Istante	Rif. Pagina	Bit Validità	Ref Info	Sostituzione	Bit Validità	Ref Info	Sostituzione	Bit Validità	Ref Info		
$t_1$	0	0	1	—	0	1	$t_1$	—	0	1	$t_1$
		1	0		1	0			1	0	
		2	0		2	0			2	0	
$t_2$	1	0	1	—	0	1	$t_1$	—	0	1	$t_1$
		1	1		1	1	$t_2$		1	0	$t_2$
		2	0		2	0			2	0	
$t_3$	0	0	1	—	0	1	$t_1$	—	0	1	$t_1$
		1	1		1	1	$t_2$		1	0	$t_2$
		2	0		2	0			2	0	
$t_4$	2	0	1	Sostituisco 1 con 2	0	0		Sostituisco 0 con 2	0	1	$t_3$
		1	0		1	1	$t_2$		1	0	
		2	1		2	1	$t_4$		2	0	$t_4$
$t_5$	0	0	1	—	0	1	$t_5$	Sostituisco 1 con 0	0	1	$t_5$
		1	0		1	0			1	0	
		2	1		2	1	$t_4$		2	0	$t_4$
$t_6$	1	0	0	Sostituisco 0 con 1	0	1	$t_5$	Sostituisco 2 con 1	0	1	$t_5$
		1	1		1	1	$t_6$		1	0	$t_6$
		2	1		2	0			2	0	
$t_7$	2	0	0	—	0	0		Sostituisco 0 con 2	0	1	
		1	1		1	1	$t_6$		1	0	$t_6$
		2	1		2	1	$t_7$		2	0	$t_7$

### Approssimazione LRU

Come appena visto, nell'algoritmo LRU, ad ogni pagina è associato un bit di referenza nella tabella delle pagine:

- Ogni pagina associa un bit, inizialmente è 0
- Quando si fa riferimento ad una pagina, il bit viene settato ad 1

Una seconda implementazione dell'algoritmo fa uso di Clock:

- Se si fa riferimento ad una pagina con reference bit = 0 → viene sostituito
- Altrimenti il bit = 1 → {
  - Viene settato il reference bit a 0 e viene lasciata la pagina in memoria
  - Per la sostituzione della prossima pagina si segue la solita regola

}

In questo caso vengono modificate le pagine in base all'accesso e viene modificato il bit:

- 0, 0: Miglior candidato (no write)
- 0, 1: write, ma usato molto tempo fa
- 1, 0: Usato di recente, ma no write
- 1, 1: Caso peggiore



## Working Set

Il concetto di working set fornisce una base per decidere quanti e quali pagine di un processo dovrebbero essere in memoria per ottenere una buona prestazione di processo. L'insieme di tutte le pagine di un processo viene detto working set (insieme di lavoro) ed è stato introdotto il modello working set per ridurre sensibilmente il tasso di page fault. Ciò consiste nell'assicurarsi che l'insieme di lavoro sia caricato totalmente in memoria prima di consentire ad un altro processo di andare in esecuzione.

Quando la memoria fisica libera è insufficiente a contenere il working set corrente di un processo, quest'ultimo comincerà a generare parecchi page fault, rallentando la propria velocità di esecuzione. Quando parecchi processi cominciano ad andare in **trashing**, ovvero a spendere più tempo per la paginazione che per l'esecuzione, il sistema operativo potrebbe erroneamente indotto ad aumentare il grado di multiprogrammazione, dato che la CPU rimane inattiva per la maggior parte del tempo a causa dell'intensa attività di I/O. In questo modo verranno avviati nuovi processi che però, a causa della mancanza di frame liberi, cominceranno a loro volta ad andare in trashing. In breve, le operazioni del sistema inizieranno a collassare e sarei costretto a "killare" brutalmente i processi!

## File System

La parte del sistema operativo che si occupa dei file si chiama *File System* e risiede su uno storage secondario (dischi).

- Fornisce un'interfaccia utente per la memorizzazione, mappatura logica a fisica
- Fornisce un efficiente e conveniente accesso al disco abilitando la conservazione di dati.

### Concetti Base:

- File: Blocco di dati memorizzato
- Directory: Collezione di file o altre directory

- Mount Point: Directory che detiene il file system di un device
- Link: Puntatore ad un file nel file system
  - Logico: La cancellazione del link non inficia sul file originale
  - Fisico: Quando il numero di link è 0, il file viene cancellato

“Root” di un file system (/) di solito contiene numerose cartelle ed alcune di loro memorizzano un file system aggiuntivo:

- Altri dichi
- Immagini di File
- Directory remote
- File System logici, popolati dall’OS che potrebbero non necessariamente corrispondere a byte fisici sul disco.
  - /proc → Informazioni riguardo ai processi in running
  - /sys → Informazioni sul sistema
  - /dev → Vista diretta ai dispositivi fisici

## Mount

L’operazione di montaggio, “connette” il file system alla struttura delle directory del sistema. Questa operazione è utile quando ci sono più file system nel sistema e dura finché il file system non viene smontato o finché il sistema non viene riavviato.

## File

Un file regolare è:

- Una memoria permanente con spazi degli indirizzi contigui
- Il suo contenuto viene definito dal creatore

## Attributi di un file:

- Per l’utente
  - Nome
  - Identificatore
  - Directory
  - Size
  - Permessi
  - Proprietario
  - Tempi di creazione, ultima modifica
  - ...
- Per il Sistema
  - Identificatore → Unico nel sistema
  - Posizione → Dove sul device
  - ...

## Directory

È nient’altro che un file di voci di file, che può contenere:

- Hard link: puntatori a file. Cancellare tutti gli hard link di un file equivale a cancellare il file!
- Soft link: Puntatori “soft”, cancellare tutti i soft link non equivale a cancellare il file.
- Altre directory

Una directory può ospitare un altro file system, se quest’ultimo viene montato nella directory. In questo caso i files nella directory non saranno accessibili affinché il file system ospitato verrà “smontato”.

## File Descriptor



Il descrittore è un intero che caratterizza un file all'interno di un processo e viene ottenuto come valore di ritorno dall'apertura/creazione di un file. Anche se lo stesso file potrà essere aperto più volte, ogni open ritornerà un differente file descriptor.

Un file descriptor in altre parole identifica la struttura nel kernel che identificherà a sua volta un'istanza di file descriptor nel PCB di un processo.

In automatico vengono allocati 3 descrittori:

- 0 → Stdout
- 1 → Stdin
- 2 → Stderr

`int open(const char* pathname, int flags, mode_t mode);` → Conterrà il file descriptor del file

`int creat(const char* pathname, mode_t mode);` → Crea il file

`int close(int fd)` → Chiude il file

Funzioni per copiare un descrittore:

```
int dup(int oldfd);
int dup2(int oldfd, int newfd);
int dup3(int oldfd, int newfd, int flags);
```

### Reading/Writing

Dato che un file è una sequenza di bytes, esso può essere visto come un array che può essere scritto a blocchi. Ogni file descriptor è associato a un puntatore che punta alla posizione che assumerà il file all'interno dell'OS e le operazioni read e write si verificano nella posizione corrente del puntatore e fanno side effect su di esso, dato che leggere  $n$  byte dalla posizione  $x$  equivale a spostare il puntatore di  $n + x$  (`lseek()`). Analogamente alle socket, i processi possono essere messi in attesa di un cambiamento su un file tramite la funzione `select()`, la quale resta in attesa per  $n$  secondi e si sbloccherà o allo scadere del tempo o se si verifica un evento, ad esempio la pressione di un tasto.

### Devices

Il filesystem /dev fornisce un aggancio per dispositivi fisici nel sistema:

- /dev/sda: disco
- /dev/sda1: Prima partizione del primo disco
- /dev/ttyACM0: FTDI = Seriale
- /dev/video0: webcam

In Unix, ogni device è accessibile come un file e questi vengono categorizzati in 2 tipi: "a blocchi" o "a caratteri". I devices a blocchi supportano la ricerca e la lettura, mentre quelli a carattere sono streams, dunque posso leggere e scrivere sequenzialmente (quello che succede con le telecamere) e possono essere configurati attraverso l'interfaccia termios.

### ioctl e Termios

`int ioctl(int fd, unsigned long request, ...);`

L'interfaccia del file è un modo unificato che ha Unix per accedere ai dispositivi, giostrando tutto con un file. Funzioni speciali su un device possono essere accedute tramite la system call ioctl, che prende in input un interno che fungerà da richiesta dell'azione che dovrà essere intrapresa. Ogni driver installa la propria "request" e il proprio gestore delle richieste quando viene caricato.

### Mmap

```
void *mmap(void *addr, size_t length, int prot, int flags, int fd, off_t offset);
int munmap(void *addr, size_t length);
```

Mmap mappa un sottoinsieme delle pagine del processo in una routine completamente diversa e ritorna un puntatore all'area di memoria. Quando un file viene "mmappato" le caches vengono gestite trasparentemente e la load/write verranno gestite dal gestore della memoria.

## Physical Devices



Luca's Mac



Ogni dispositivo nel sistema è differente da un altro e la sua implementazione potrebbe massimizzare la coerenza dell'interfaccia e forzare il driver a seguire un'altra specifica interfaccia.

- Un driver è una sorta di “classe virtuale” che dovrebbe scavalcare metodi standard definiti dall'interfaccia del SO

I dispositivi di memoria obbediscono all'interfaccia “a blocchi” ma potrebbe succedere che un driver di uno specifico dispositivo utilizzi altri devices del sistema, come ad esempio i driver USB.

I dispositivi “a blocchi” in /dev, dovrebbero implementare l'interfaccia del dispositivo a blocchi, come:

- Query
- Seek
- Scrivi blocco
- Leggi blocco
- ...

### Dischi

Un disco è un dispositivo primitivo che effettua solamente due operazioni basilari:

- Leggi blocco
- Scrivi blocco

Per ogni dispositivo disco, esiste un driver che offre funzionalità di lettura e scrittura e comunica con il disco fisico. Sopra avrà un disk scheduler, il quale si occuperà di far fare alla testina del disco movimenti più regolari possibili, e sopra ancora l'interfaccia del sistema. Un disco fisico ammette comunque un'interfaccia simil-file.

Un disco ha la lettura a blocchi, e non carattere per carattere ed è proprio per questo che conviene che tutte le strutture dati siano un multiplo della dimensione del disco.

### Implementazione di un File System

L'idea di base è quella che un disco viene visto come un grande file binario che viene acceduto a blocchi (512/4096). Sul file binario occorre “mappare” un filesystem con le sue cartelle sottocartelle, ..., come se fosse una sorta di struttura dati. Il sistema dovrebbe essere consapevole del tipo di filesystem all'interno di una partizione, per dialogare correttamente con esso.

Esempio di filesystem:

- VFAT
- Ext2/3/4
- Ntfs
- ...

### Driver del FileSystem

Un driver è quella componente del sistema che si occuperà dell'effettivo dialogo con il filesystem.

Questo utilizza un'uniforme interfaccia a blocchi ed implementa funzioni quali:

- Open
- Close
- Read
- ...

### Strutture Dati per la manipolazione di FileSystem

- Nel Kernel (Memorizzazione Temporanea)
  - Per ogni file aperto, viene creata un'istanza della struttura e viene memorizzato il descrittore del file aperto dal processo invocante, tramite un'apposita struttura “OpenFileInfo”

- Per ogni descrittore aperto da un processo, un'appropriata struttura “OpenFileRef” viene memorizzata in una lista accessibile dal PCB
- Ogni “OpenFileRef” punta alla corrispondente “OpenFileInfo” e memorizza un indipendente puntatore a file per la gestione di seek/read/write.
- Sul Disco (Memorizzazione Permanente)
 

Un file è grande almeno un blocco del disco.

  - Ogni file è caratterizzato da una struttura, chiamata FileControlBlock (FCB), che risiede all'inizio del blocco e serve a gestire informazioni specifiche del file
  - Ogni directory ha una struct di intestazione che estende l'FCB, in parole poche una directory è un file che possiede un record che contiene il file della directory.

### **Allocazione di un File**

I primi file system usavano il modello di **allocazione contigua** della memoria, allocando una singola area di memoria a ogni file al momento della creazione. In questo tipo di allocazione, ogni file occupa un insieme di blocchi contigui sul disco, quindi per poter effettuare un'allocazione del genere basta sapere solo il blocco iniziale e la lunghezza del file. Un'allocazione di questo tipo però porta alla frammentazione esterna, generando aree di memoria troppo piccole per potere essere riutilizzate e non solo, potrebbe generare anche frammentazione interna poiché il file system era progettato per allocare spazio extra sul disco per consentire ai file di crescere.

L'**allocazione concatenata** (linked) risolve il problema della frammentazione esterna e quello della dichiarazione delle dimensioni del file, entrambi presenti nell'allocazione contigua della memoria. In questa allocazione, ogni file è rappresentato da una lista concatenata di blocchi del disco, che possono essere sparpagliati ovunque sul disco. Ogni blocco sul disco avrà due campi: dati, che contiene i dati, e metadati, che è un campo di tipo link che punta al prossimo blocco. I contro di questo tipo di allocazione sono che se cambio record, la testina del disco si muoverà da una parte all'altra, ma al contrario l'accesso sequenziale è relativamente efficiente.

Una variante importante del metodo di assegnazione concatenata consiste nell'uso della *tabella di assegnazione del file*, tenuta in memoria e chiamata **File Allocation Table** (FAT). Per contenere tale tabella si riserva una sezione del disco all'inizio di ciascuna partizione e la FAT ha un elemento per ogni blocco del disco. Per un blocco allocato a un file, il corrispondente elemento della FAT contiene l'indirizzo del blocco successivo. In questo modo il blocco e il suo elemento nella FAT insieme formano una coppia che contiene la stessa informazione contenuta nel blocco nel classico schema dell'allocazione concatenata. Il vantaggio è che per trovare blocchi liberi è molto facile in quanto nella prima parte ho la vista di tutti i blocchi liberi.

L'**Allocazione Indicizzata** risolve il problema di accesso diretto, presente nell'allocazione concatenata, raggruppando tutti i puntatori in una sola locazione: *il blocco indice*. Nell'allocazione indicizzata si mantengono tutti i puntatori ai blocchi di un file in una *tabella indice* chiamata file map table (FMT), nella quale sono riportati gli indirizzi dei blocchi del disco allocati a un file. Ogni file ha il proprio blocco indice e nella forma più semplice un FMT è un array di indirizzi di blocchi del disco. Ogni blocco ha un solo campo, il campo dati e l'i-esimo elemento del blocco indice punterà all'i-esimo blocco del file. Se il file si espande e non può essere più contenuto, l'ultimo puntatore punterà a un altro indice di blocchi che conterrà puntatori che a loro volta punteranno ad un'altra struttura. Questo fa sì che si venga a creare una sorta di gerarchi rappresentabile con una struttura ad albero.

### **Come Implementare le Directory?**

Una directory può essere implementata tramite:

- **Lista lineare** di nomi di file con puntatori ai blocchi di dati
  - Semplice da programmare
  - Finito il blocco la directory continuerà nel blocco successivo



- Contro: Se ho troppi file in una directory, fare la ricerca è molto onerosa poiché dovrò riordinare tutta la lista
- **Hash Table** – Lista lineare con struttura dati hash. Dato un nome di un file riceverò dalla tabella hash tutti quelli che corrispondono al nome del file.
  - Decrementa il tempo di ricerca
  - Potrebbero generarsi collisioni dal momento che due nomi uguali di file si trovano nella stessa posizione

### Gestione dello Spazio Libero

- **Bitmap**
  - Mantengo un bitmap all'inizio del disco, nel quale ogni bit corrisponderà ad un blocco
  - Durante la ricerca di un blocco libero, viene utilizzato il bitmap per cercare il più vicino
  - Leggermente più lento, ma considera il layout del disco
- **Linked List**
  - Uguale allo SLAB allocator
  - Punta a tutti i blocchi liberi sul disco

## IPC

I processi si scambiano informazioni attraverso la *comunicazione interprocesso* (IPC). Lo scambio dei messaggi (*message passing*) si adatta a diverse situazioni in cui lo scambio di informazioni tra processi gioca un ruolo fondamentale. Uno dei suoi usi più importanti è nel modello *client-server*

in cui un processo server fornisce un servizio e altri processi client gli inviano messaggi per utilizzare il servizio.

### Message Passing

Quando i processi interagiscono fra loro devono essere sincronizzati per garantire mutua esclusione ed hanno bisogno di scambiarsi informazioni per cooperare tra loro, in questo caso tramite lo scambio di messaggi. In questo caso, la memoria non è condivisa e la cooperazione avviene in questo modo:

- Il processo  $P_1$  consegna un'area di memoria al kernel
- Il kernel la consegna a  $P_2$
- Non avviene sincronizzazione ma il messaggio deve essere copiato

Le primitive che lo implementano sono:

- Send(sender/mailbox, messaggio)
- Il messaggio dunque può essere inviato direttamente ad un processo o encapsulato in una mailbox; nell'ultimo caso ogni processo che conosce la mailbox potrà accedere al messaggio.
- Receive(sender/mailbox, messaggio)

La comunicazione può avvenire in modo:

- Sincrono (bloccante) o asincrono (non bloccante)
- Diretta (tra processi) o indiretta (tramite mailbox)
- Limitata o illimitata

### Implementazione Message Passing

Una coda di messaggi è un oggetto gestito dal sistema che implementa una mailbox. I processi devono conoscere un identificativo della coda per operarci.

Funzionalità

- Open: Apertura coda di messaggi → *mq\_open()*
- Close: Chiusura della coda → *mq\_close()*
- Unlink: Distruzione coda → *mq\_unlink()*
- Get/set attr: setto se bloccante o meno → *mq\_getattr()*/*mq\_setattr()*
- Put a Message
- Wait for a message / notify when is ready → *mq\_notify()*

### Shared Memory

Si tratta di un'area di memoria condivisa tra i processi, molto simile alle code di messaggi o ai semafori. Una memoria condivisa, una volta aperta, ha bisogno di essere "mappata" nell'area di memoria di un processo tramite la funzione *mmap*. Senza di questa un altro processo non potrebbe accedere nella stessa area di memoria.

- Scrittura su memoria condivisa

```
int main(int argc, char const *argv[]) {
    char* resource_name = argv[1];

    int fd = shm_open(resource_name, O_RDWR | O_CREAT, 0666);
    if(fd < 0) {
        handle_error_en("Errore nella creazione della memoria condivisa.
Errore: %s", sys_errlist[errno]);
    }
    int ftruncate_result = ftruncate(fd, SHMEM_SIZE);      // passo le
dimensioni del file per poterlo aumentare
    if(ftruncate_result < 0) {
```



```

        handle_error_en("Non riesco a troncare l'oggetto memoria
condiviso. Errore: %s", sys_errlist[errno]);
    }

    // Setto l'area di memoria e la imposto in scrittura
    void* my_mem_area = mmap(NULL, SHMEM_SIZE, PROT_WRITE, MAP_SHARED, fd,
0);

    int num_rounds = 100;
    for(int i = 0; i < num_rounds; i++) {
        char* buf = (char*)my_mem_area;
        snprintf(buf, "Messaggio %d", i);           // scrivo in memoria
        printf("Scrittura [%s]\n",buf);
        sleep(1);
    }

    int unlink_result = shm_unlink(resource_name);           // cancello il file
    if(unlink_result < 0){
        handle_error_en("Non riesco a distruggere l'oggetto memoria.
Errore %s", sys_errlist[errno]);
    }
    return 0;
}

```

- **Lettura da memoria condivisa**

```

int main(int argc, char const *argv[]){
    char* resource_name = argv[1];

    int fd = shm_open(resource_name, O_RDONLY, 0666);
    if(fd < 0){
        handle_error_en("Errore nella creazione della memoria condivisa.
Errore: %s", sys_errlist[errno]);
    }

    int SHMEM_SIZE = 0;

    struct stat shm_status;

    void* my_mem_area = mmap(NULL, SHMEM_SIZE, PROT_READ, MAP_SHARED, fd,
0);           // blocco in lettura

    while(1){
        char* buf = (char*) my_mem_area;
        printf("%s\n", buf);
        sleep(1);
    }

    int unlink_result = shm_unlink(resource_name);
    if(unlink_result < 0){
        handle_error_en("Non riesco a distruggere l'oggetto memoria.
Errore %s", sys_errlist[er])
    }
}

```



```

    }

    return 0;
}

```

## Stack e Context Switch

Un contesto di un processo, ovvero ciò che mi serve per farlo eseguire o per far continuare la sua esecuzione, richiede:

- Registri della CPU
- Memoria del processo
  - Stack
  - Codice
  - Variabili inizializzate
  - Variabili globali

Se ogni processo utilizza solo la sua memoria, l'esecuzione potrebbe essere interrotta e poi fatta ripartire salvando e ripristinando i registri della CPU.

### Coroutines

È un pezzo di programma che serve ad effettuare salti da una funzione all'altra, cambiando il contesto. Ciò è implementato in C dalla libreria ucontext.

Creerò dunque un contesto tramite la struct ucontext\_t e le funzioni:

- ucontext\_t\* uc\_link → Creano il contesto prendendo in input stack e il puntatore ad un altro contesto a cui passerò l'esecuzione
- sigset\_t uc\_sigmask → Per vedere se il contesto reagisce o meno ad interruzioni
- stack\_t us\_stack → Puntatore alla stack che conterrà tutti i registry della CPU
- mcontext\_t uc\_mcontext

**getContext** → Salva lo stato del contesto in ucp

```
int getcontext(ucontext_t *ucp);
```

```

struct ucontext_t ctx; // salvo il nostro punto di salto
// get scrive <-> set effettua il salto
int f2() {
    setContext(&ctx);           // setto il contenuto della funzione
}                                // appena invocata su ctx

int f1() {
    ...
    getContext(&ctx);         // setto ctx per saltare alla prossima
    ...                         // istruzione dopo getcontext
    f2();
}

```

**setContext** → Salva il contesto corrente in ucp, contesto che precedentemente era stato salvato

```
int setcontext(const ucontext_t *ucp);
```

**makeContext** → Crea contesto di esecuzione, un trampolino, lanciando una funzione come prima istruzione. Scrive su ucp un nuovo contesto che se attivato porta all'esecuzione di una funzione.

```
void makecontext(ucontext_t *ucp, void (*func)(), int argc, ...);
```



**swapContext** → Scrive il contesto di esecuzione in cui voglio andare e legge dalla CPU dove voglio saltare

```
int swapcontext(ucontext_t *oucp, const ucontext_t *ucp);
```

Esempio:

```
#include <stdio.h>
#include <ucontext.h>
#include <stdlib.h>
#include <unistd.h>

#define STACK_SIZE      16384
ucontext_t main_context, f1_context, f2_context;

int num_iterations = 100;

void f1(){      // ad ogni ciclo fa swap, salva in f1 e legge da f2
    printf("Parte F1!\n");
    for (int i = 0; i < num_iterations; i++) {
        printf("F1: %d\n", i);
        sleep(1);
        swapcontext(&f1_context, &f2_context);
    }
}

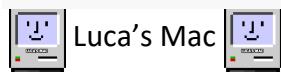
void f2(){ // simmetrica ad f1
    printf("Parte F2!\n");
    for (int i = 0; i < num_iterations; i++) {
        printf("F2: %d\n", i);
        sleep(1);
        swapcontext(&f2_context, &f1_context);
    }
}

// gli passo la stack esplicitamente
char f1_stack[STACK_SIZE];
char f2_stack[STACK_SIZE];

int main(int argc, char *argv[]) {
    // riempio i campi dei getcontext
    getcontext(&f1_context);           // salvo il contesto di f1 solo per
                                       // riempire i campi di ucontext

    f1_context.uc_stack.ss_sp = f1_stack; // gli passo la stack
    f1_context.uc_stack.ss_size = STACK_SIZE;
    f1_context.uc_stack.ss_flags = 0;
    f1_context.uc_link = &main_context;   // quando terminerà continua in
                                         // maincontext

    // Creo un trampolino per la prima funzione
    makecontext(&f1_context, f1, 0, 0);   // quando eseguito lancia f1
```



```

f2_context = f1_context;           // setto gli stessi flags per non ripetere
                                   // due volte l'assegnazione
// gli cambio i valori
f2_context.uc_stack.ss_sp = f2_stack;
f2_context.uc_stack.ss_size = STACK_SIZE;
f2_context.uc_stack.ss_flags = 0;
f2_context.uc_link = &main_context;

// creo il trampolino per la seconda funzione
makecontext(&f2_context, f2, 0, 0);

// passo il controllo ad f1 e salvo il contesto in main_context
swapcontext(&main_context, &f1_context);

printf("Escooo!\n");
}

```

Esempio:

```

#include <ucontext.h>
#include <stdio.h>
#include <stdlib.h>

static ucontext_t uctx_main, uctx_func1, uctx_func2;

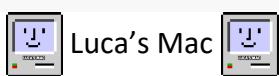
#define handle_error(msg) \
    do { \
        perror(msg); \
        exit(EXIT_FAILURE); \
    } while (0)

static void func1(void){
    printf("func1: Sono Partita\n");
    printf("func1: swapcontext(&uctx_func1, &uctx_func2) => Salto a func2\n");
    if (swapcontext(&uctx_func1, &uctx_func2) == -1)
        handle_error("swapcontext");
    printf("func1: Sono Terminata\n");
}

static void func2(void){
    printf("func2: Sono Partita\n");
    printf("func2: swapcontext(&uctx_func2, &uctx_func1) => Salto a func1\n");
    if (swapcontext(&uctx_func2, &uctx_func1) == -1)
        handle_error("swapcontext");
    printf("func2: Sono Terminata\n");
}

int main(int argc, char *argv[]){
    char func1_stack[16384];
    char func2_stack[16384];

```



```

    if (getcontext(&uctx_func1) == -1) handle_error("getcontext");
    uctx_func1.uc_stack.ss_sp = func1_stack; // gli do la stack
    uctx_func1.uc_stack.ss_size = sizeof(func1_stack); // gli do la size
    uctx_func1.uc_link = &uctx_main; // gli passo cosa dovrà eseguire
    makecontext(&uctx_func1, func1, 0);

    if (getcontext(&uctx_func2) == -1) handle_error("getcontext");
    uctx_func2.uc_stack.ss_sp = func2_stack; // gli do la stack
    uctx_func2.uc_stack.ss_size = sizeof(func2_stack); // gli do la size
                                                // della stack
    /* Successor context is f1(), unless argc > 1 */
    uctx_func2.uc_link = (argc > 1) ? NULL : &uctx_func1;
    makecontext(&uctx_func2, func2, 0);

    printf("main: swapcontext(&uctx_main, &uctx_func2)\n");
    if (swapcontext(&uctx_main, &uctx_func2) == -1)
        handle_error("swapcontext");

    printf("main: exiting\n");
    return 0;
}

```

### Esempio:

```

#include <stdio.h>
#include <stdlib.h>
#include <ucontext.h>
#include <unistd.h>

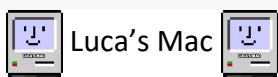
static ucontext_t ctx[3];

static void f1 (void){
    printf("Start f1\n");
    sleep(1);
    swapcontext(&ctx[1], &ctx[2]); // da qui salto alla funzione 2
    sleep(1);
    printf("Fine f1\n");
}

static void f2 (void){
    printf("Start f2\n");
    sleep(1);
    swapcontext(&ctx[2], &ctx[1]);
    sleep(1);
    printf("Fine f2\n");
}

int main(int argc, char *argv[]){
    char st1[8192];
    char st2[8192];

```



```

getcontext(&ctx[1]);           // setto il contesto di ctx[1]
ctx[1].uc_stack.ss_sp = st1; // gli do la stack
ctx[1].uc_stack.ss_size = sizeof(st1); // gli do la size della stack
ctx[1].uc_link = &ctx[0]; // link a dove deve saltare
makecontext(&ctx[1], f1, 0); // gli dico cosa deve lanciare (f1)

/* FACCIO LA STESSA COSA CON IL SECONDO: Setto il ctx[2], ... */

getcontext(&ctx[2]); // <-----+ A QUI
ctx[2].uc_stack.ss_sp = st2;// |
| |
ctx[2].uc_stack.ss_size = sizeof(st2); // |
| |
ctx[2].uc_link = &ctx[1];// |
| |
makecontext(&ctx[2], f2, 0); // |
| |
// |
| |
puts("Inizio il salto, dopo aver già settato tutta la merda!"); // |
swapcontext(&ctx[0], &ctx[2]); // ---+ SALTO DA QUI-----+
| |

return 0;
}

```



Luca's Mac