

OS - PROGRAMMAZIONE DI SISTEMA

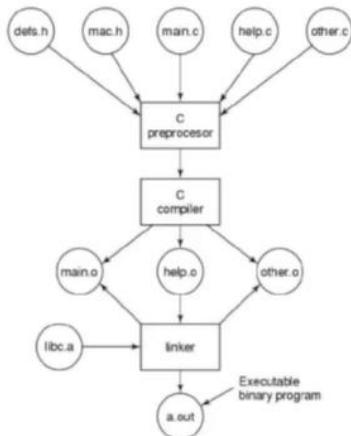
9) GESTIONE MEMORIA

(analizzare il ruolo dell'OS nel fornire ai processi utente una porzione di memoria logica [mappata sulla memoria fisica])

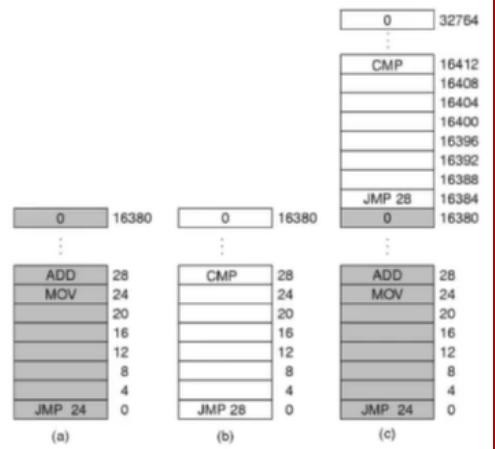
Un programma deve essere portato in memoria e messo in un processo per essere runnato (il sistema di elaborazione può avere più programmi attivi [Multiprogrammazione]).

Solo la **memoria principale (RAM)** e i **registri** sono **direttamente accessibili** dalla CPU (accesso a registro ≤ 1 clock; accesso a memoria principale = molti clock, causando anche **stalli** → per questo **tra memoria principale e registri c'è la CACHE**).

La memory unit vede solo uno stream di **indirizzi + read request** oppure **indirizzo + data + write request**. Viene richiesta **PROTEZIONE** della memoria per garantire operazioni corrette: bisogna garantire che **un processo possa accedere solo agli indirizzi del suo spazio di indirizzamento** ("address space"), definito da **base register** e **limit register** [indirizzi della CPU] (quindi lo spazio di un processo sarà **da base a base+limit**). È importante rimanere nei propri spazi di indirizzamento perché nella RAM oltre ai processi c'è anche il kernel.



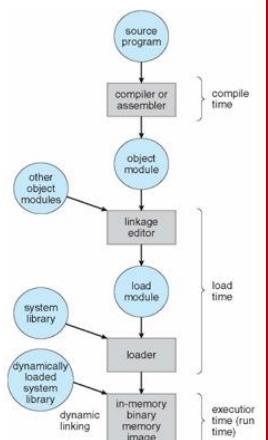
Vediamo il meccanismo di creazione di un programma eseguibile ("model of runtime"); come si vede dalla figura, è il compilatore che genera per ogni sorgente un file oggetto, poi uniti dal linker in un unico eseguibile:



Vediamo il problema di **allocare più processi a zone diverse della memoria (RELOCATION)**: i processi (a; grigio) e (b; bianco) in figura pensano di lavorare entrambi dall'indirizzo 0; se però vengono caricati in RAM come nel caso (c), vediamo che solo (a) parte da 0, mentre (b) ha come "zero" 16384. Quindi la jump 28 non porterebbe nella posizione voluta da (b); per questo si potrebbe **usare il base register** come offset, ovvero impostare base register = 16384 così che la jump porti a $28 + \text{base register} = 28 + 16384 = 16412$.

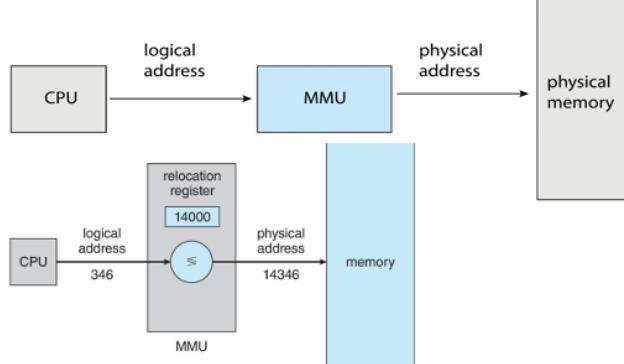
Quando un eseguibile viene portato da disco in memoria, non conosce a priori a quale indirizzo dovrà runnare: gli indirizzi nel codice sorgente sono quindi **"simbolici"** in quanto il **compilatore** traduce il programma in assembler scrivendo al posto di questi simboli degli **indirizzi rilocabili** (es. "14 Byte dall'inizio di questo modulo"): questo si chiama **ADDRESS BINDING** e può avvenire in 3 momenti/fasi:

1. **Compile Time** (fase di generazione dell'eseguibile) → bisogna conoscere a priori l'indirizzo a cui il processo andrà in memoria
2. **Load Time** (fase di caricamento dell'eseguibile in memoria) → il codice eseguibile deve essere rilocabile; sia il linker sia il loader devono collaborare per convertire gli indirizzi
3. **Execution Time** → binding è differito fino all'esecuzione in memoria di un determinato modulo/istruzione (es. se un programma chiama la funzione f1 e la funzione f2, si farà il binding degli indirizzi di queste funzioni solo quando vengono chiamate). La risoluzione



avviene tramite l'uso di **indirizzi logici** (usati dal processo) e **indirizzi fisici** (usati dalla CPU per accedere alla porzione desiderata della RAM).

Per convertire **indirizzi logici in fisici** serve un hardware veloce che faccia queste conversioni in tempi brevi. Nella **Memory Management Unit (MMU** → gestisce la memoria) è implementato il **relocation register** che, tramite un sommatore, viene sommato a tutti gli indirizzi logici generati da un processo per ottenere l'indirizzo fisico

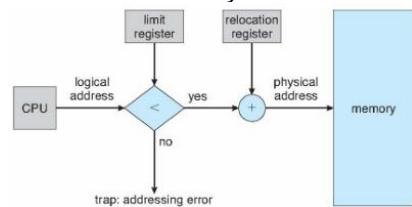


⚠ Il Binding può realizzare schemi “**dinamici**” di caricamento ed esecuzione del programma perché il sistema può evitare di risolvere gli indirizzi durante la scrittura del programma. Abbiamo infatti:

- **Dynamic Loading** = un programma può essere caricato in memoria scomposto in pezzi (es. una funzione non viene caricata fin quando non viene chiamata); ciò ottimizza l'uso della memoria, senza richiedere un supporto speciale dall'OS
- **Dynamic Linking** (linking = fase in cui il linker aggiunge al programma le funzioni dalle librerie, decidendone gli indirizzi) = pospone linking alla fase di esecuzione del programma, assegnando al posto della funzione effettiva uno **stub** (funzione fittizia) che verrà aggiornato ad Execution Time (usato molto in *shared libraries*)

Ora parliamo di **ALLOCATION** (come viene allocata la memoria ai processi e come la condividono):

- **CONTIGUOUS ALLOCATION** (più semplice) → ad ogni processo viene assegnato un range di indirizzi (detto **partizione**, di dimensione variabile). Si usa la somma del **relocation register** per la risoluzione dell'indirizzo logico-fisico e viene controllato che **indirizzo < limit register** (altrimenti si invoca una **trap** per gestire l'errore) [a dx immagine per tutto processo].

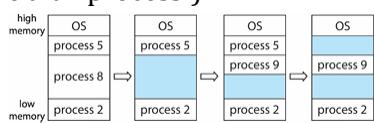


Il numero di processi collocabili in memoria dipende dal numero di partizioni allocabili. Quando un processo termina lascia uno “**spazio vuoto**” nel punto in cui era stato allocato; ci sono 3 politiche che permettono di decidere dove il processo viene allocato:

- **First-fit** = nel 1° buco abbastanza grande (devo quindi sapere a priori quanto occuperà il processo, ma è molto veloce)
- **Best-fit** = nel buco più piccolo possibile, abbastanza grande da contenere lo (< spazio vuoto lasciato)
- **Worst-fit** = nel buco più grande possibile (lasciando spazio residuo per anche altri processi)

Un altro problema nell'allocazione è la **FRAMMENTAZIONE** (cioè l'alternanza di spazi pieni e vuoti):

- **Frammentazione esterna** = alternanza fuori dai processi (es. nell'ultima parte dell'immagine)
- **Frammentazione interna** = memoria allocata ad un processo è più grande del necessario, lasciando una parte di memoria inutilizzata



Quando una memoria è troppo frammentata si fa **deframmentazione** (spostare i processi per minimizzare i buchi). Si può fare anche quando il processo è in esecuzione (“ready”), ma ciò può generare il “**problema di I/O**”: un processo potrebbe essere in “wait” (ovvero è sorgente/destinazione di un'operazione con delle periferiche); 2 soluzioni:

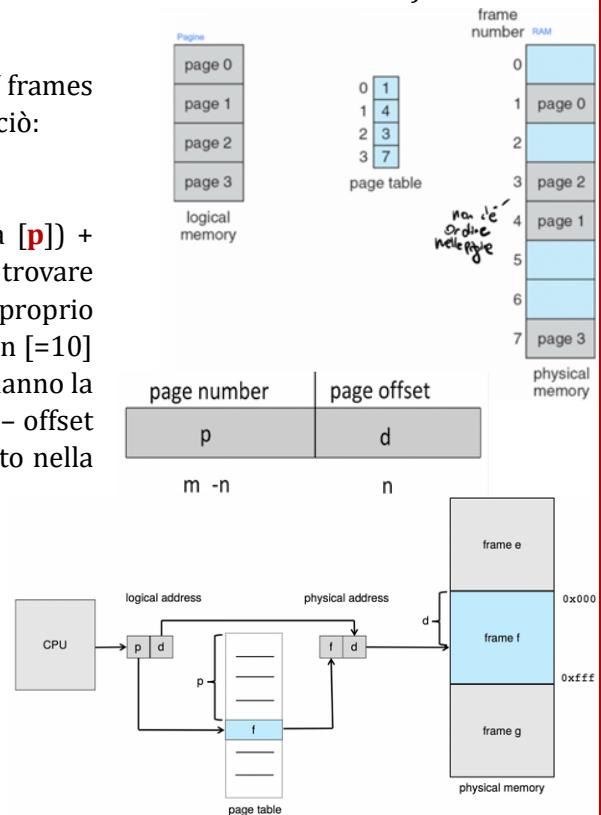
- vietare la rilocazione in memoria di un processo in wait di un I/O
- copiare i dati del processo necessari all'I/O in un buffer di kernel (che farà da intermediario)

La vera soluzione per ridurre la frammentazione è il **PAGING** (“**paginazione**”): non viene associato uno spazio variabile ai processi, ma si spezzettano in “**pagine**” di dimensione fissa (**frammentazione esterna risolta!**); anche la memoria fisica viene divisa in blocchi di dimensione fissa (2^i) detti **frames**. Quindi abbiamo che:

- **Memoria logica** = vettore di pagine contigue (ognuna allocata in un frame della memoria fisica)
- **Memoria fisica** = vettore di frames

Per allocare un programma di N pagine contigue, dobbiamo avere N frames liberi, ma questi molto difficilmente potranno essere contigui; perciò:

1. Viene creata una tabella della “**page table**”
2. Indirizzo logico diviso in **page number** (indice della pagina [p]) + **offset** (displacement = quanto devo scorrere nella pagina per trovare il dato [d]). La **dimensione del frame in potenza di 2 è comoda** proprio per questa divisione (es. indirizzo logico di m [= 32] bit di cui n [=10] dell'offset e m-n [=22] del page number). Il frame e le pagine hanno la stessa dimensione, quindi se indirizzo è ad esempio pagina 7 – offset 11, **non è necessaria traduzione** per l'offset (posizione del dato nella pagina sarà stessa posizione nel frame)
3. Come si può vedere in figura, la **CPU** emette l'indirizzo logico che viene diviso in p e d; l'offset (d) va diretto in memoria fisica, il page number (p) serve per trovare il numero del frame nella page table



Risolta la frammentazione esterna, **come si calcola l'interna?**

Esempio

Process size = 72.766 bytes. Page size = 2048 bytes. Quanti bytes di frammentazione interna?

$$\text{Numero di pagine necessarie: } \frac{\text{Process size}}{\text{Page size}} = \frac{72.766}{2048} = 35.53$$

Assegno 36 pagine, approssimando per eccesso all'intero più vicino. Ora posso calcolare quanto dell'ultima pagina viene usato e quanti bytes rimangono inutilizzati:

Numero di bytes utilizzati nell'ultima pagina: $72.766 - 35 \times 2048 = 1086 \text{ bytes}$

Numero di bytes inutilizzati nell'ultima pagina: $2048 - 1086 = 962 \text{ bytes}$

La frammentazione interna ammonta perciò a 962 bytes.

⚠ È meglio avere frame piccoli o grandi? Dal punto di vista della frammentazione la situazione migliora, ma avere più grame significa avere una page table più grande (nei sistemi reali le pagine vanno dagli 8 KB ai 4 MB)

⚠ Una riga della page table in realtà non contiene solo il frame number, ma anche se il frame è stato **modificato** o se è abilitato il **caching**

La page table viene **salvata in RAM** (troppo grande per i registri); per ottimizzarne l'uso vengono salvati 2 registri:

- **PTBR** (Page-Table Page Register) = contiene indirizzo della RAM dove la page-table inizia [come base]
- **PTLR** (Page-Table Length Register) = contiene la dimensione della page-table [come limit]

⚠ Quindi traduzione indirizzo logico-fisico con allocazione contigua richiede solo CPU (no spreco di clock), mentre con page-table richiede lettura in RAM: quindi per ogni accesso in memoria, devo farne 2 (1 per indirizzo del frame della page-table + 1 per l'effettivo frame; spreco di tempo).

Per ottimizzare ciò si usa **TLB** (Translation Lookaside Buffer), cioè una tabella salvata in CPU più piccola della page table; ogni riga ha associazione pagina logica – frame fisico (es. pagina 140 → frame 31; diversa dalla page-table che ha frame 31 scritto all'indice 140). Richiede una memoria associativa per la scansione della page-table prima di creare l'associazione.

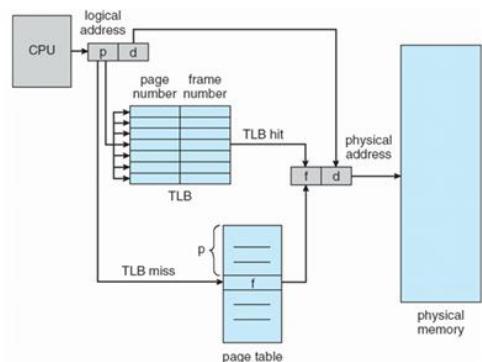
Valid	Virtual page	Modified	Protection	Page frame
1	140	1	RW	31
1	20	0	R X	38
1	130	1	RW	29
1	129	1	RW	62
1	19	0	R X	50
1	21	0	R X	45
1	860	1	RW	14
1	861	1	RW	75

Il **processo** è il seguente → vedo se ho dato nella TLB:

- se c'è, **TLB hit**
- se non c'è, **TLB miss** + copio il dato dalla RAM nella TLB + riparto cercando il dato nella TLB

Bisogna quindi **stimare la probabilità di hit**; facciamo un **esempio**: se accesso in memoria = 10ns con hit (dato in TLB) e 20ns con miss (2 accessi in RAM); 80% dei casi hit e 20% miss; allora il tempo effettivo di accesso in memoria sarà:

$$\text{EAT}(\text{Effective Access Time}) = 0.8 * 10 + 0.2 * 20 = 12\text{ns}$$

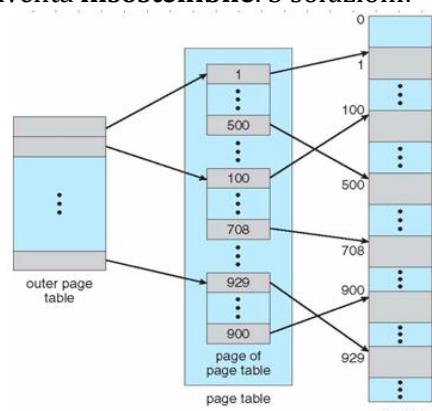
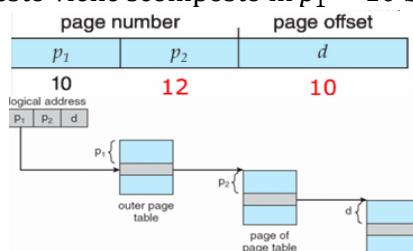


La page-table però ha alcuni **vantaggi**:

- ✓ **Memory Protection** → oltre al frame number, al "modified" e al caching, la riga della page table ha **1 bit di validità** (indica se la pagina è valida)
- ✓ **Condivisione pagine tra processi diversi** (solo in lettura) → permette di occupare meno RAM; ogni processo può vedere le pagine come se fossero sue, ma in realtà sono mappate tutte sullo stesso frame (queste pagine si dicono "**rientranti**" perché non hanno uno stato che salva la precedente esecuzione [no variabili globali]). **La condivisione deve essere solo sul codice rientrante** (e non estesa a tutti i dati del processo) [un esempio può essere mappare una libreria usata da 3 processi sullo stesso frame]

Com'è fatta la tabella delle pagine? Esempio: spazio di indirizzamento = 32 bit; dimensione pagine = 4KB (2^{12} Byte), quindi nell'indirizzo 12 bit bassi = offset, 20 bit alti = page number. Se abbiamo una RAM di Gbyte, riusciamo a rappresentare il numero di frame con 16-32 bit: se 16 bit non sono sufficienti ne usiamo 32, quindi 4 Byte per riga, ovvero una page-table di 4MB. In memoria la tabella **deve essere contigua**, quindi in questo caso (4MB) si può ancora fare, ma se n processi devono avere la loro page-table diventa **insostenibile**. 3 soluzioni:

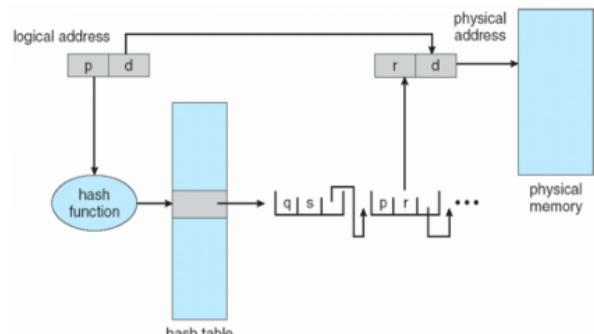
- **Hierarchical Paging (Paging Gerarchico)**: la page table (prima contigua) viene spezzata in sezioni e si mette davanti ad essa una tabella di "1° livello" che serve solo a trovare in quale tabella di "2° livello" si cerca il numero di frame. L'indirizzo logico di 32 bit viene diviso in 3 parti ora: nell'esempio qui sotto, abbiamo 10 bit = offset (d) e 22 bit = page number, ma questo viene scomposto in $p_1 = 10$ bit e $p_2 = 12$ bit:



La traduzione logico-fisico viene fatta usando p_1 = **seleziona una riga** nella page table più esterna contenente l'indirizzo di una page table di 2° livello, e poi p_2 = **individua la riga** della 2^ page table **che contiene il numero di frame**; dentro al frame si usa il displacement (offset) per trovare l'indirizzo fisico.

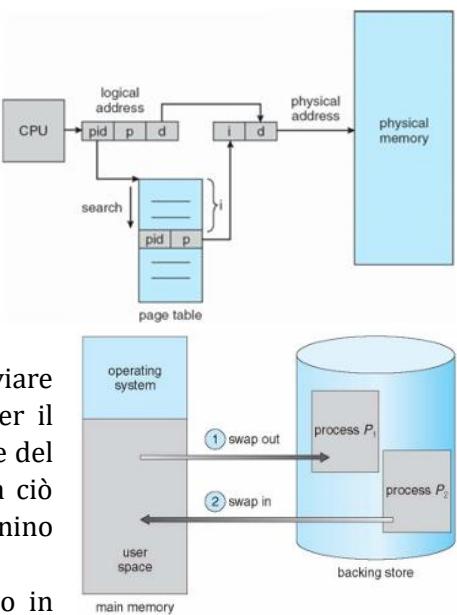
Esempio → pagine da 4 KB (2^{12}) → page-table ha 54 righe perché (come nell'esempio sopra) 12 bit bassi = offset, 20 bit alti = page number. Dividendo la tabella in 2 avrei comunque una tabella esterna da 2^{42} righe e una interna di 2^{10} righe → conviene fare un'ulteriore separazione (generando una page-table gerarchica a **3 livelli**). In questo modo però per accedere ad un dato dovrò accedere alla outer page, poi alla outer page, poi ancora alla inner page ed infine al frame: per ogni dato da prelevare dalla memoria avrò quindi 4 accessi, è importante perciò usare la **TLB** (TLB miss molto costoso)

- **Hashed Page Table**: > tempi di accesso grazie all'Hashing. Il vettore è generalmente più compatto poiché con le liste di collisione si possono mettere più entries nella stessa riga. La tabella di Hash può essere di 2 tipi:
 - 1^{\wedge} = ad ogni entry corrisponde 1 frame
 - 2^{\wedge} = ad ogni entry corrisponde 1 cluster



- **Inverted Page Table**: inverte la logica → tabella in cui **page number = contenuto, offset (frame) = indice** (es. se pagina 100 associata al frame 23, anziché $p[100] = 23$, cerco il valore 100 nella page table e per avere il numero di frame vedo qual è l'indice della riga contenente 100, cioè $100 = p[23]$). La dimensione della page-table è quindi proporzionale alla RAM, ma devo fare una ricerca lineare (della riga nella page-table); per risolvere posso usare una tabella di Hash, dove le entries sono poste in liste di adiacenza

- **Swapping**: supponiamo di avere troppi processi in RAM e di volerne avviare uno nuovo; si può usare lo **SWAPPING**, ovvero fare lo “**swap out**” per il processo da togliere e “**swap in**” per il nuovo processo che entra. I frame del processo uscente vengono spostati in una porzione del disco adibita a ciò (“**Backing store**”); se devo riportare i frame in RAM, non è detto che tornino dov'erano prima (dipende se gli indirizzi sono rilocabili).



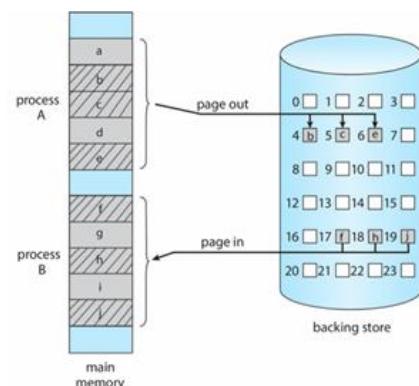
Quanto costa fare swapping? Un “**context switch**” (cambiare processo in esecuzione) costa (salvare i registri e verificare che i registri ripristinati siano validi per l'esecuzione), ma a ciò si aggiungono i **tempi di trasferimento dei frames** disco-RAM e viceversa (es. processo da 100MB; disco da 50MB/sec → 2 sec di swap out e 2 di swap in).

Un altro problema è come nella deframmentazione, ovvero **spostare un processo in wait di un I/O**; 2 soluzioni:

- non consentire swap out del processo
- fare in modo che un processo in stato di wait non abbia mai dati di memoria user coinvolti nell'I/O, ma ciò richiede un **double buffering**: copio i dati del processo dalla memoria user alla memoria kernel (+ veloce che trasferimento verso memoria esterna), poi faccio swap out del processo ed infine copio i dati dalla memoria kernel all'I/O (quando I/O è pronto)

Un altro swapping è lo **swapping with paging**: un processo potrebbe salvare sul backing store non tutte, ma solo alcune delle sue pagine (solo quelle non necessarie al proseguimento delle operazioni del processo).

Esempio → il processo A in figura lavora sulle pagine *a* e *d*, quindi *b*, *c* e *e* vengono swapped out sul backing store; il processo B ha bisogno di *f*, *h* e *j* che quindi vengono swapped in dal backing store

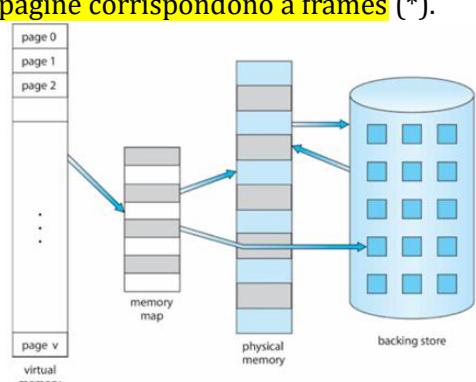


10) MEMORIA VIRTUALE

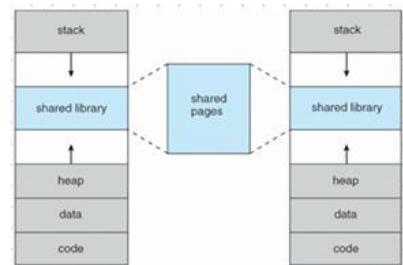
(analizzare la memoria virtuale + definire la paginazione a richiesta, utile per definire uno spazio di indirizzamento virtuale realizzato in modo dinamico)

MEMORIA VIRTUALE = netta separazione tra spazio di memoria di indirizzi logici e fisici (nasconde che una parte di indirizzi logici sono mappati su fisici ed una parte no) [quindi virtualmente spazio logico esiste, ma realmente no; quindi la porzione di memoria che vede un processo potrebbe essere più grande della RAM stessa, consentendo di condividere parti di memoria tra processi, di far runnare più processi e uno swap più efficiente]; **virtual address space** = insieme degli indirizzi logici che vede un processo (di solito parte da 0 ed è contiguo). **Gli indirizzi fisici corrispondono a frames, invece non necessariamente le pagine corrispondono a frames** (*)

La traduzione logico-fisica è fatta dalla **MMU** ed è implementata con la **“paginazione a richiesta”**: come si vede in figura, abbiamo lo spazio di indirizzamento virtuale (“virtual memory”) rappresentato più grande della RAM (“physical memory”). La **memory map** è l’insieme delle page-tables che traduce logico → fisico. Alcune pagine [blocchi blu] possono essere direttamente in un frame (*) (quindi in memoria fisica), mentre altre [blocchi in grigio] sono nel backing store

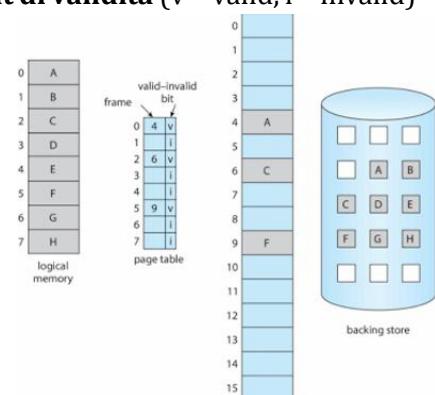


Qui invece vediamo com'è fatto lo spazio di indirizzamento virtuale ("virtual memory"): dati e codice sono messi in fondo essendo di dimensione fissa; lo stack parte dal 1° indirizzo, mentre l'heap dall'ultimo (essendo che cresceranno dinamicamente nel processo) [si possono mettere anche librerie condivise nel mezzo]



DEMANDING PAGING ("Paginazione a richiesta") = una pagina entra in un frame solo quando **necessario**; è simile allo **swapping with paging**, ma qui è centrale la singola pagina e non l'intero processo: se una pagina è "necessaria" ne facciamo riferimento e, se il riferimento non è valido, la pagina è ancora su disco, perciò la copiamo in RAM e la usiamo da lì. Possiamo quindi fare il **lazy swapper** (non porto la pagina di un processo in memoria fino a che quel processo non ne necessita); lo swapping avviene basandoci su predizioni di quali pagine avrò bisogno in futuro (serve una MMU dedicata a questo).

Può accadere che un processo tenti di **accedere ad una pagina che non è mappata su un suo frame, ma che è già in memoria** (es. pagina shared): per questo nelle page-tables e nelle TLB c'è il **bit di validità** (v = valid, i = invalid) affianco al frame (indica se pagina è in un frame ed è "memory resident" o no).

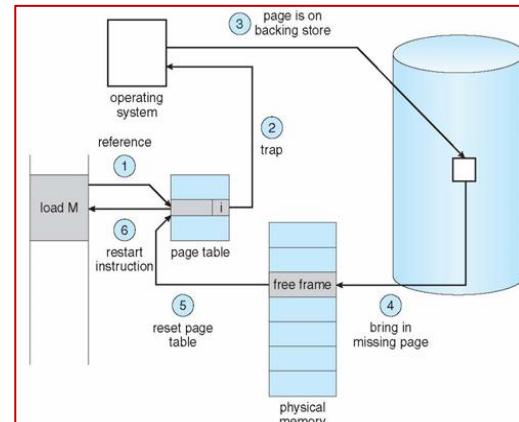


Esempio: accediamo alla page-table alle pagine 0, 2, 5: in figura, queste sono tutte in frame (quindi in RAM) e sono valide (v, v, v). Tutte le altre pagine sono

invalidi (i), creando quindi dei "buchi". Nel backing store ci sono tutte le pagine (sia in frame che no).

Come si gestisce un **PAGE FAULT**?

1. Si parte con un riferimento a pagina: il 1° riferimento invalid è gestito via hardware, ovvero la MMU fa una **trap**
2. Inviata la trap, il **kernel** deve capire se:
 - a. riferimento **errato** (es. puntatore null) → chiude programma
 - b. riferimento **corretto a pagina non in RAM**:
 - i. Cerca frame libero in RAM (free-frame list **(*)**)
 - ii. Copia pagina da disco a quel frame; il kernel si mette in wait e dà la CPU ad un altro processo
 - iii. Riceve **I/O completed** mediante interrupt da disco, salva i registri del processo partito mentre era in wait e capisce che l'interrupt veniva dal disco
 - iv. Sistema page-table
 - v. Setta a "v" il bit di validità
 - vi. Fa ripartire la load (ovvero il processo che aveva scatenato il page fault) mettendola in coda



⚠ La page fault impiega molto tempo; infatti un caso estremo è la "**pura paginazione a richiesta**" (parto da processo senza pagine in memoria ed ogni nuova richiesta genera page fault)

Abbiamo che il **page fault rate** (probabilità di page fault tra 0 e 1) modifica l'**EAT** (effective access time):

$$EAT = (1 - p) * t_{accesso\ memoria} + p * (\text{page fault overhead} + \text{swept}_{page\ out} + \text{swept}_{page\ in})$$

Esempio

Memory access time = 200 ns. Average page fault service time = 8 ms

$$EAT = (1 - p) * 200\text{ns} + p * (8\text{ ms}) = 200\text{ ns} + p * 7.999.800\text{ ns} \rightarrow \text{varia al variare di } p$$

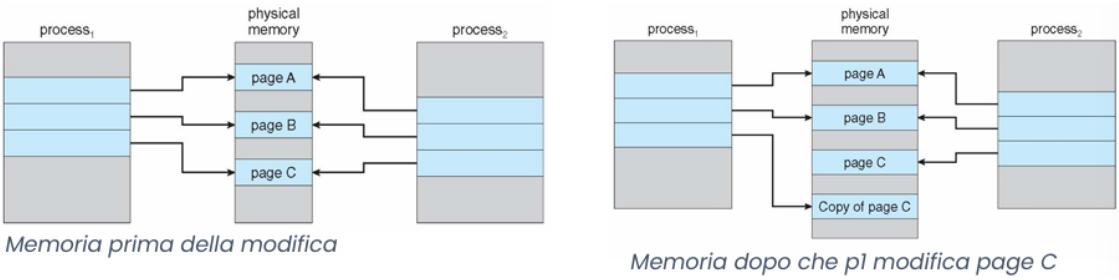
Ipotizzando un page fault ogni 1000 accessi: EAT = 8,2 μs .

Ho quindi rallentato l'accesso di un fattore pari a 40. Il page fault incide altamente sul tempo effettivo di accesso in memoria.

Un singolo page fault può portare più pagine in memoria (es. una somma tra 2 numeri in memoria che scrive risultato in memoria ha 4 accessi in RAM: lettura istruzione add, 2 operandi da leggere e variabile da scrivere); inoltre ci sono istruzioni che hanno indirizzi partenza-arrivo sovrapposti, o che sono più pagine.

Quindi il page fault può avere **più pagine da gestire** e necessita di **hardware prestante**. Per gestirlo:

- **FREE-FRAME LIST (*)** → lista di frames liberi (per trovare frame libero dopo page fault) [inizialmente la lista è tutta la memoria]. Ogni frame libero **punta** al frame libero successivo (il **kernel** punta al **1° frame libero** e può dare ai processi dei frame liberi pre-azzerati [pool di pagine azzerate da affiancare alla free-frame list]). Come si può vedere dagli step del page fault visti sopra, le 3 attività principali sono **interrupt, lettura da disco** (parte più lunga) e **riavvio del processo**
- **DEMANDING PAGE OPTIMIZATION** → per rendere più immediato il page fault un'opzione è migliorare l'accesso al disco, l'altra è rendere più veloce la **partizione di swap** e fare il demanding paging tra quella partizione e RAM (e non tra disco e RAM) [quindi > tempo di avvio, ma < problemi in esecuzione]

⚠ In **Solaris** si fa invece l'opposto: si fa paging su disco, ma **con lo swap out le pagine vengono eliminate** (e non salvate su disco [se servono, recuperate dall'eseguibile]). Per fare ciò: si usa anonymous memory, lo stack e l'heap partono puliti e vengono riempiti col tempo (es. scrivo nell'heap usando una malloc molto grande, ma poi ne uso solo una parte). Alcune delle pagine non vengono mai usate e non c'è mai uno swap in da disco (perciò non ci sarà nemmeno swap out) [alcune pagine vengono solo lette e non modificate, perciò le butto senza swap out]
- **COPY-ON-WRITE** → riguarda **processi che condividono le pagine** (es. dopo una fork, processi parent e child): se 1 dei processi modifica una pagina condivisa, questa viene copiata in memoria (**pagine duplicate solo se 1 dei processi ci scrive sopra**)
 

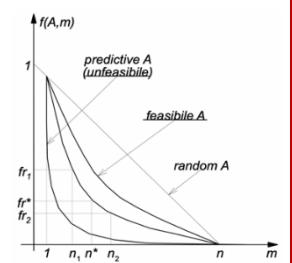
Se non ci sono più frame liberi? Dobbiamo scegliere una pagina (non quella a cui un processo accederà subito dopo) e toglierla; per sceglierla si usano **ALGORITMI DI PAGE REPLACEMENT** che usano il **dirty (modify)** bit, ovvero il bit associato ad ogni pagina che indica se il processo l'ha modificata o no (se non è stata modificata, dopo averla eliminata dalla RAM non serve copiarla su disco). Per **valutare gli algoritmi** uso una **reference string** (noi useremo **7 0 1 2 0 3 4 2 3 0 3 0 3 2 1 0 1 7 0 1**) e calcolo il n° di page faults, trovando la **Page Fault Frequency**:

$$\text{Page Fault Frequency} = f(A, m) = \sum_w p(w) * \frac{F(A, m, w)}{\text{len}(w)} \quad \text{con} \quad \left\{ \begin{array}{l} A = \text{algoritmo di page replacement} \\ m = \text{n° page frames disponibili} \\ w = \text{reference string} \\ p(w) = \text{probabilità di } w \\ \text{len}(w) = \text{lunghezza di } w \\ F(A, m, w) = \text{n° faults generati in base a } A, m, w \end{array} \right.$$

Nel grafico qui a dx si vedono **3 classificazioni di algoritmi** (il migliore è predittivo, poi feasible e poi random) in funzione di page fault frequency (ordinate) e n° di frames disponibili (ascisse) [n° page fault è quindi collegato con il n° frame assegnati al processo]. Un altro fattore è il **reference bit** (indica se processo ha mai fatto accesso alla pagina, anche in sola lettura).

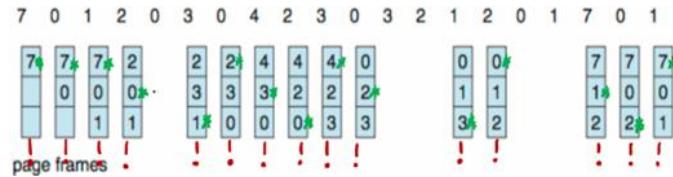
Vediamo questi algoritmi:

- **Basic Page Replacement** = consideriamo il caso in cui il page fault ha prelevato la pagina dal disco e non sa dove metterla (ricercando il frame, noto che la frame-list è vuota). Bisogna trovare un "**frame vittima**" da buttare fuori (e, se ha dirty bit attivo, copiare su disco; si preferisce comunque una pagina con dirty bit a 0); copio la pagina nel frame che ho liberato e riavvio il processo (quindi nulla di nuovo, come visto prima).



⚠️ Quanti frame assegno ad un processo all'avvio? > frame assegnati → < probabilità di page replacement, < processi ospitabili in RAM

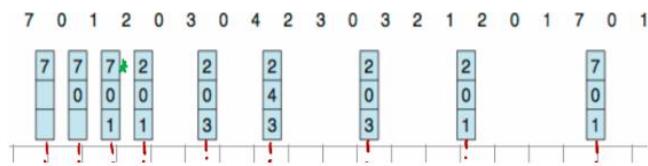
- **FIFO (First In First Out)** = 1^a pagina entrata in un frame sarà anche la 1^a ad uscire (semplice, non ottimale).
Esempio (nell'immagine * = next victim [head di coda FIFO]; ! = page fault): ho 3 frames; i primi 3 inserimenti [7,0,1] sono page fault non avendo frame disponibili. Quando devo aggiungere 2, ho page fault perché sono tutti pieni: lo inserisco al posto del 7 (**head = first in = next victim**) e la nostra head in diventa 0. Quando devo aggiungere 0 invece non avrò page fault perché 0 è già in frame. Quando devo inserire 3, tolgo lo 0 (head) e così via



⚠️ Il risultato varia in base alla stringa di riferimento: alcune stringhe portano alla "Anomalia di Belady" (> n° frame disponibili → > n° page faults) [es. 1 2 3 4 1 2 5 1 2 3 4 5]

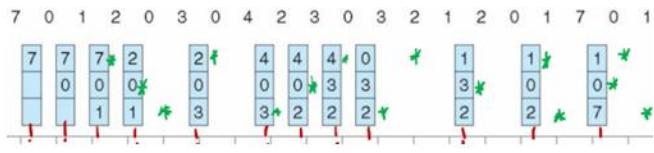
- **Optimal** = conosce il futuro (**legge tutta la reference string prima di eseguire l'algoritmo** e scelgo come next victim quella che per più tempo non sarà usata/non sarà più usata); **non realizzabile** (non so di quali pagine avrò bisogno), ma ottimale (il migliore).

Esempio: primi 3 page faults sono sempre inevitabili [***], ma quando devo inserire 2 vedo che: 0 usato nella iterazione dopo, 1 tra 10 iterazioni, 7 tra 14 → quindi elimino il 7 etc... (passiamo da 15 a 9 page faults)



- **LRU (Least Recently Used)** = si usa la storia passata per prevedere il futuro (**località temporale** = è probabile che la pagina a cui accederò più in là nel tempo, sia quella a cui ho fatto un accesso più lontano nel passato).

Esempio: i primi 3 page faults sono sempre inevitabili, ma quando devo inserire 2 vedo che il 7 è quello con accesso meno recente e lo sceglio come victim; etc... (passiamo a 12 page faults)



È molto costoso perché **ogni pagina ha contatore con ultimo accesso** (**al page fault si cerca min contatore**): si può implementare con una **lista doppio linkata sullo stack** (metto le pagine ordinate in modo che la più in alto dello stack sia la più recente, mentre l'ultima è la meno recente [vittima]) [uso lista e non vettore perché posso far slittare i valori]

Gli algoritmi LRU sono ancora troppo costosi, perciò si usano **LRU approssimati** (scelgo una pagina usata abbastanza meno recentemente e non la minore in assoluto) tramite il **reference bit** (pagina abbastanza recente ha reference bit = 1, altrimenti 0).

Uno di questi è il **Second-chance Algorithm** (algoritmo FIFO: se la head della coda FIFO ha reference bit = 0 è la victim [in questo caso 0(1)], sennò scorro la coda fino a che non trovo una pagina con reference bit = 0). Di questo algoritmo ci sono **varianti che usano sia reference sia dirty** (modify) bit (cerco una pagina con ref=0 e mod=0 [non dovrò scriverla su disco], poi con ref=0 e mod=1 [dovrò scriverla su disco], poi con ref=1 e mod=0, infine con ref=1 e mod=1)

- **Counting** = guardano la **frequenza di accessi** in un certo Δt : una pagina mi serve se nell'ultimo tempo di riferimento (Δt) l'ho usata tante volte. 2 tipi (scelta dipende dal contesto):

- **LFU (Least Frequently Used)** → victim = pagina che ho usato poco
- **MFU (Most Frequently Used)** → victim = pagina che ho usato molto

- **Page Buffering** = evita problema di aver sbagliato la scelta della victim (scelto pagina che servirà tra poco): suppongo di aver bisogno di un frame libero, ma sono tutti pieni. Tengo un pool di frame liberi: quando parte page fault, seleziono una victim che viene aggiunta al **free-frame pool**; tutte le vittime aggiunte verranno copiate su disco in un secondo momento (quando e come avviene il salvataggio su disco è la parte difficile dell'algoritmo). Una volta inserita nel pool, se faccio riferimento alla pagina la recupero

⚠ Se nella scelta della politica di allocazione/sostituzione page prende parte anche il **programma utente** oltre che il kernel, si potrebbero avere vantaggi in quanto l'applicazione potrebbe essere più conscia di cosa succede, ma al tempo stesso ci potrebbero essere conflitti tra kernel e processo utente

Un altro aspetto per ottimizzare il paging è capire il **n° migliore (minimo) di FRAMES DA ALLOCARE**; parliamo quindi di **FIXED ALLOCATION**:

- **Equal Allocation** = alloco a tutti i processi un n° fisso di frames (es. se ho 100 frames a disposizione e 5 processi, alloco 20 frames ciascuno). So a priori quanta RAM dare ad un processo, ma potrei sprecare frames (un processo potrebbe non usarli tutti)
- **Proportional Allocation** = se conosco la dimensione di un processo (dimensione del suo "address space"), posso allocargli RAM proporzionalmente. **Esempio:**

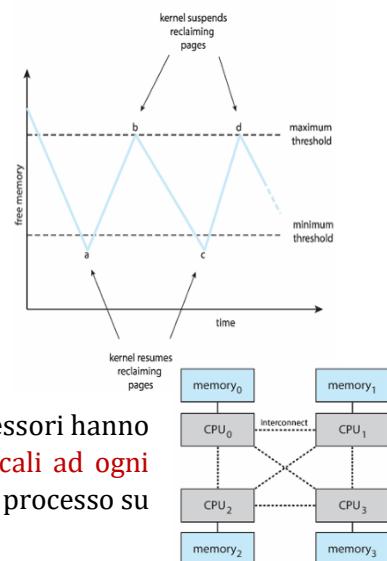
$$M = \text{n° frames totali} = 64 \quad S = \text{dimensione totale processi} = 137 \text{ pagine} \quad s_i = \text{dimensione processo } p_i$$

$$\rightarrow \begin{cases} S_1 = 10 \text{ pagine} \\ S_2 = 127 \text{ pagine} \end{cases} \quad a_i = \text{allocazione per } p_i = \frac{s_i}{S} * M \rightarrow \begin{cases} a_1 = \frac{10}{137} * 64 \approx 4 \\ a_2 = \frac{127}{137} * 64 \approx 57 \end{cases}$$

Un'ulteriore problema è **cercare la victim**:

- **Global replacement** = cerco la vittima **tra tutti i frames** (processo può rubare frames [RAM] ad un altro, cioè elimino indipendenza tra processi)
- **Local replacement** = processo cerca la vittima **solo tra le sue pagine** (uso subottimo della RAM)

RECLAIMING PAGES = come già detto, un processo può accedere ad una lista globale di free frames: il page replacement viene **attivato** non quando la lista è vuota, ma **quando il n° di free-frames scende sotto una soglia** e viene disattivato quando la free-list diventa sufficientemente piena

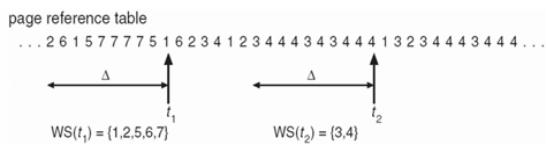


NUMA (Non-Uniform Memory Access) = nei sistemi moderni multicore più processori hanno una memoria condivisa: è come se la RAM fosse divisa in **partizioni di RAM locali ad ogni processore** (es. in figura, la CPU₀ accede più velocemente a mem₀ che a mem₁); un processo su una data CPU **conviene che usi i free-frames sulla memoria locale alla CPU**

THRASHING = quando un sistema va in crisi? Se il **processo non ha abbastanza frames**, genera molti page fault (costoso); innesca il **threading** (< prestazioni): stiamo salvando memoria per far girare quanti più processi, ma nessuno di questi processi viene effettivamente eseguito (stiamo cercando di non avere buchi nell'uso della CPU, ma dopo una certa soglia la CPU rimane inutilizzata perché **non riesco più ad assegnare le pagine**) → questo è il **THRASHING** (i processi lavorando con un **modello di località** → in thrashing ci vado quando la **sommatoria delle località dei processi > memoria disponibile**)

Un 1° modello che usa la località degli accessi è il **WORKING SET MODEL** ("località degli accessi" l'insieme delle pagine su cui un processo lavora in un certo Δt). Chiamiamo Δ = **working set window** (se troppo piccola, WSS non comprende località; se troppo grande, WSS comprende tutto programma → va presa adeguata) e **WSS_i** = **Working Set Size** del processo p_i (n° di pagine su cui ho lavorato negli ultimi istanti) ($\text{WSS} \in [1, \Delta]$). Chiediamo al sistema $D = \sum_i \text{WSS}_i$ prendendo i frames di tutto il processo.

Esempio: abbiamo $\Delta = 10$; all'istante t_1 ho Working Set = {1 2 5 6 7} [$\text{WSS} = 5$]; all'istante t_2 ho WS = {3 4} [$\text{WSS} = 2$]. Se riusciamo a fare in modo che le pagine che stanno nei frame coincidano con il WS (**D = WS**), abbiamo implementato bene il **Working Set Model**; se **memoria < D**, si ha **Thrashing**



⚠ Il WS Model ha un **difetto**: ad ogni istante il WS potrebbe aumentare o diminuire (es. da t_1 all'istante dopo in cui facciamo accesso alla pagina 6, non avrei page fault perché 6 era già in memoria, ma se voglio mantenere i frame allineati al WS devo **buttare la pagina 2 anche se non ho bisogno di fare spazio** [perché esce fuori dalla Δ] → il WS diventa {1 5 6 7} e dunque lo **swap out** avviene ad ogni accesso ad una nuova pagina e non ad ogni page fault come dovrebbe).

In realtà si **approssima** questa strategia con il **CAMPIONAMENTO FISSO** che usa il **reference bit** (es. $\Delta = 10000$, quindi non controllo la Δ ad ogni accesso ma aggiorno il WS dopo $t = 5000$ istanti usando un interrupt che per ogni pagina salva il reference bit e poi lo azzera; nei prossimi 5000 istanti ogni accesso mette il reference bit a 1 in modo da poterlo poi controllare e usare questa informazione per i page fault dei prossimi 5000 istanti)

Altra strategia è la **PAGE FAULT FREQUENCY** = se ho troppi page fault aumento il n° di frames dedicati al processo, mentre se sono pochi li diminuisco. Per **semplicità** si può usare anche solo 1 soglia delle 2 (solo troppi o solo pochi) e si può agire solo quando c'è un page fault (invece di intervalli fissi): misuriamo $\tau = \Delta t$ da ultimo page fault, e definiamo c sperimentalmente in modo che:

- se $\tau < c \rightarrow$ aggiungo un frame e ci mettiamo la nuova pagina
- se $\tau \geq c \rightarrow$ page replacement

La vittima viene scelta tra le pagine a cui non ho fatto accesso dall'ultimo page fault (sempre usando il **reference bit**); quello che cambia è che **sto usando i page fault come Δt** .

Esempio: scelgo $c = 3$; metto 6 nel 1° frame (page fault). Pagina 4 ho page fault → $\tau = 1 < 3$, quindi aggiungo un frame. All'istante 4 ho page fault per pagina 3 → $\tau = 2 < 3$, quindi aggiungo un frame. All'istante 5 ho page fault (pagina 2) → $\tau = 1 < 3$, quindi aggiungo un frame. All'istante 9 ho page fault (pagina 1) → $\tau = 4 > 3$, quindi aggiungo un frame e dentro ci metto pagina 1, poi azero i reference bit.

Tempo	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24
Stringa	6	4	4	3	2	4	4	4	1	3	3	2	4	4	5	1	2	2	2	6	1	2	5	4
Frame 0	6	6	6	6	6	6	6	6	6	6	6	6	6	6	5	5	5	5	5	6	6	6	6	6
Frame 1	-	4	4	4	4	4	4	4	4	4	4	4	4	4	4	4	4	4	4	2	2	2	2	2
Frame 2	-	-	3	3	3	3	3	3	3	3	3	3	3	3	3	3	3	3	3	1	1	1	1	1
Frame 3	-	-	-	2	2	2	2	2	2	2	2	2	2	2	2	2	2	2	2	-	-	-	5	5
Frame 4	-	-	-	-	-	1	1	1	1	1	1	1	1	1	1	1	1	1	1	-	-	-	-	4
Fault	*	*	*	*	*					*	*	*	*				*	*					*	*
Ref Bit						0				0									0					

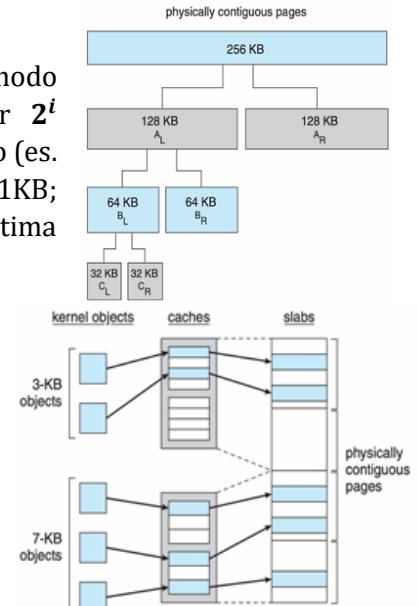
Diagramma manuale del logico di gestione dei page fault:

- Accesso a frame 2: vede ref bit=1
- Accesso a frame 3: vede ref bit=1
- Accesso a frame 4: vede ref bit=1
- Accesso a frame 5: vede ref bit=0
- Accesso a frame 6: vede ref bit=0
- Accesso a frame 7: vede ref bit=0
- Accesso a frame 8: vede ref bit=0
- Accesso a frame 9: vede ref bit=0
- Accesso a frame 10: vede ref bit=0
- Accesso a frame 11: vede ref bit=0
- Accesso a frame 12: vede ref bit=0
- Accesso a frame 13: vede ref bit=0
- Accesso a frame 14: vede ref bit=0
- Accesso a frame 15: vede ref bit=0
- Accesso a frame 16: vede ref bit=0
- Accesso a frame 17: vede ref bit=0
- Accesso a frame 18: vede ref bit=0
- Accesso a frame 19: vede ref bit=0
- Accesso a frame 20: vede ref bit=0
- Accesso a frame 21: vede ref bit=0
- Accesso a frame 22: vede ref bit=0
- Accesso a frame 23: vede ref bit=0
- Accesso a frame 24: vede ref bit=0

All'istante 15 ho page fault (pagina 5) → $\tau = 6 > 3$, vedo che 6 e 1 hanno reference bit a 0, quindi vengono rimpiazzate; metto pagina 5 etc...

Finora abbiamo parlato solo della memoria da allocare a processi utente, per massimizzare il numero di processi senza andare in thrashing. Il **KERNEL** però viene gestito in modo diverso perché ha **bisogno di alcune strutture dati allocate in modo contiguo** (es. le page table); esiste quindi un memory pool dedicato al kernel. Vediamo alcuni dei possibili allocator per il kernel:

- **BUDDY SYSTEM** = alloca dei blocchi di memoria di dimensione fissa in modo contiguo, cercando di ridurre la frammentazione esterna. Alloca per 2^i scegliendo i in modo da avere la minima potenza di $2 >$ dello spazio richiesto (es. in figura abbiamo una partizione libera di 256KB e il kernel ne chiede 21KB; quindi procediamo a dividere in partizioni più piccole, fermandoci a 32 [ultima potenza di 2 > 21]]
- **SLAB ALLOCATOR** = doppio livello di allocazione senza usare le potenze di 2. Abbiamo lo **slab** (1 o più pagine continue) e la **cache** (1 o più slabs): si cerca di allocare per 1 tipo di struttura dati del kernel 1 cache (fatta di slabs). Quando si crea una cache la si riempie di oggetti liberi, allocati poi quando vengono usati per una struttura dati del sistema (es. in figura il kernel ha bisogno di struct da 3KB e da 7KB; vengono allocate in 2 cache diverse)



⚠ [***] Per diminuire il numero di page fault dei primi accessi in memoria, possiamo prepaginare alcune delle pagine iniziali del processo (ovvero lo portiamo in frame prima che si faccia riferimento), cioè **PREPAGING**. Questo può essere sconveniente però perché non si sa che vengano effettivamente usate ($S = \text{n° pagine prepaginate}$, $a = \text{probabilità uso pagina} \rightarrow S \cdot a = \text{n° pagine usate effettivamente}$, $S \cdot (1-a) = \text{overhead}$)

⚠ Mentre la frammentazione esterna viene risolta con il paging, per l'interna è meglio avere **pagine piccole**, ma più sono piccole e più ne avrò ([dimensione page table >](#)); al contempo, dal punto di vista dell'**I/O overhead**, più le pagine sono **grandi** e **meno accessi al disco** per copiarle dovrò fare (e avrò anche [< page faults](#))

⚠ **TLB reach** (quanto copre la TLB) = TLB size * Page size

⚠ Come la struttura di un programma impatta sulla **località** ([Program Structure](#))? Ho 2 programmi, entrambi azzerano una matrice, ma uno procede per righe ed uno per colonne. Ogni riga è salvata su una pagina, quindi il programma 1 in totale dà $128 \times 128 = 16.384$ page faults, mentre il programma 2 dà 128 page faults:

```
for (i = 0; i < 128; i++)      for (j = 0; j < 128; j++)  
    for (j = 0; j < 128; j++)    for (i = 0; i < 128; i++)  
        data[i,j] = 0;           data[i,j] = 0;
```

11) **MEMORIA DI MASSA** (memorie di massa e di schedulazione, gestione del dispositivo e delle strutture RAID)

Per **MEMORIA DI MASSA** intendiamo:

- **HDD** (hard disk) = dispositivi con rotazione più lenti delle RAM; tra GB e TB; transfer rate lento (1Gbit/s). **t_{posizionamento}** (*random-access time*) = **t_{seek}** (tempo posizionamento testina su cilindro) + **t_{rotational latency}** (tempo ricerca settore). Se la testina tocca la superficie del disco ho un "**head crash**". Ogni settore è costituito da dipoli magnetici che, in base all'orientamento, fanno leggere 0 o 1; le letture sono fatte dalle testine (come abbiamo già detto cercano prima cilindro, poi settore)

<ul style="list-style-type: none">■ Access Latency = Average access time = average seek time + average latency<ul style="list-style-type: none">● For fastest disk 3ms + 2ms = 5ms● For slow disk 9ms + 5.56ms = 14.56ms■ Average I/O time = average access time + (amount to transfer / transfer rate) + controller overhead■ For example to transfer a 4KB block on a 7200 RPM disk with a 5ms average seek time, 1Gb/sec transfer rate with a .1ms controller overhead =<ul style="list-style-type: none">● 5ms + 4.17ms + 0.1ms + transfer time =● Transfer time = $4\text{KB} / 1\text{Gb/s} \times 8\text{Gb} / \text{GB} \times 1\text{GB} / 1024^2\text{KB} = 32 / (1024^2) = 0.031\text{ ms}$● Average I/O time for 4KB block = 9.27ms + .031ms = 9.301ms

- **NVM (Nonvolatile Memory Devices)**; SSD [solid state disk] se sono disk-drive) = molto più veloci degli HDD, costo >, affidabilità >, capacità <

Una memoria non volatile però assomiglia più ad una RAM che ad un HDD e viene **letta per pagine** (non per bit). Ha però **limite alle riscritture** (quindi ad oggi ci sono strategie in modo che le scritture siano su varie porzioni della memoria in modo equilibrato [evito il deperimento di solo alcune zone]). La vita di queste memorie viene misurata in drive writes per day (**DWPD**). [Come gestire queste memorie?](#)

- **Nand Flash Controller Algorithms** = se non si fanno riscritture, le pagine avranno alcuni dati validi ed alcuni non validi → il gestore del dispositivo deve avere una tabella per tracciare tali dati
- **Volatile Memory** = le memorie volatili possono essere usate per complementare la memoria di massa, ma alla fine del lavoro devo salvare i dati che mi servono (altrimenti tutto perso)

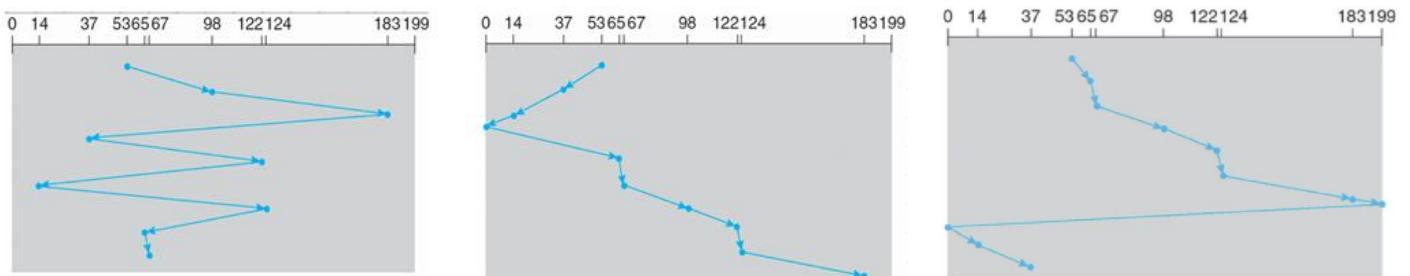
Vediamo la **DISK STRUCTURE** (oggi usata con alcune modifiche anche per gli SSD): un disco magnetico è come un **vettore di blocchi logici** (ogni blocco è un'unità di informazione da leggere in un colpo solo) da rimappare in **settori sequenziali** sul disco (settore 0 = 1° settore della traccia più esterna) [ci possono essere anche "**bad sectors**", ovvero settori guasti, che posso comunque ignorare se non sono troppi].

La **Disk Attachment** è l'interfaccia con cui il disco interagisce con il resto del sistema (es. SATA, USB...) che contiene i bus di I/O

Lo **Scheduling** è una componente dell'OS che dice dove scrivere e leggere sul disco per ridurre i tempi. Bisogna infatti considerare che sul disco arrivano **richieste** dal sistema operativo, dai processi di sistema e dai processi utente, perciò vanno **riordinate e ottimizzate**.

Esempio: supponiamo che i controller del disco possano tenere sospese delle operazioni in parallelo. Su un disco con tracce da 0 a 199 arrivano delle richieste (98, 183, 37, 122, 14, 124, 65, 67). La testina è sulla traccia 53:

usando strategia **FCFS** (First Come First Served) [1^a immagine] vedo che non ha senso andare da una parte all'altra del disco senza fermarmi su blocchi intermedi (es. ascensore che va da piano 53 a 183, ma che poi deve tornare al 67); perciò si usa la strategia **SCAN** (fa come un **ascensore**, ovvero recupera i "passeggeri" intermedi) [una variazione è il **C-SCAN** che fa 1 unica salita in alto e poi torna indietro (ultima immagine a dx)]



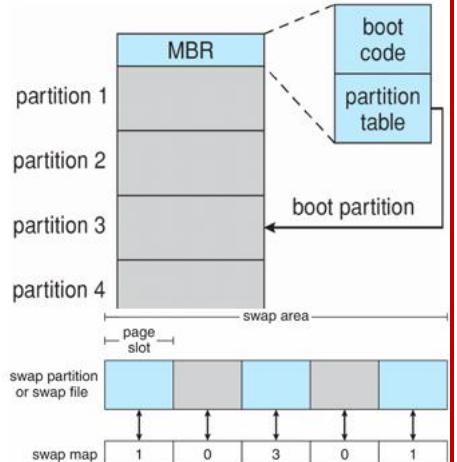
L'algoritmo migliore di disk scheduling è però **SSTF** (Shortest Seek Time First = cerco il dato più vicino alla testina). SCAN e C-SCAN sono buoni con dischi molto usati ma possono causare starvation (richiesta in attesa per troppo tempo). Linux implementa un deadline scheduler per evitarlo, ovvero le operazioni di I/O vengono ordinate in base alla loro scadenza

Ciò detto fino ad ora vale solo per le **memorie magnetiche** (dischi fisici) perché nelle memorie non volatili non c'è una testina mobile (sugli NVM c'è il **NOOP** [no scheduling]) e l'accesso più efficiente è quello **random**. Se sul disco ci sono informazioni errate, bisognerebbe individuare l'errore (checksum e CRC) e possibilmente correggerlo (ECC).

Dal punto di vista della gestione del disco, questo va **formattato**, cioè diviso in settori che il controller legge o scrive; per arrivare ad un **file system** (cioè una formattazione logica) serve partizionare il disco in uno o più gruppi di cilindri (ogni cilindro è un disco fisico, ma ognuno ha più dischi logici su cui si fa la formattazione logica [filesystem]; tra la visione logica di un disco e quella fisica ci sono dei livelli intermedi). Una delle partizioni è la **root partition** (che contiene l'OS) che viene **mounted** (cioè agganciata ad un OS in esecuzione) nella fase di **boot**. Quando si fa un mount si verifica che il filesystem è corretto. Le info sul boot sono nel "**boot block**". Se un sistema non ha questa partizione è **RAW**.

La **ROM** contiene il **BIOS** (Basic I/O System) che legge e carica in RAM l'**MBR** (Master Boot Record); questo ha le info su come il disco è partizionato (**partition table**) e su dove trovare il kernel per caricarlo in RAM (Bootstrap). Sul disco c'è anche la **SWAP PARTITION** (come un vettore di blocchi, mappato con una "swap map" [dice quali blocchi attivi e quali no]; è il backing store della RAM)

⚠ La disponibilità della rete permette di avere **dischi forniti da server** e non fisicamente presenti sul device in uso (distinguiamo storage in host-attached, network-attached e cloud). Uno **storage array** è un insieme di device di storage messi insieme per fornire un servizio, mentre una **storage area network** è l'insieme di 1 o più storage arrays.



Parlando ancora di DISK STRUCTURE, la **struttura RAID** (Redundant Array of Inexpensive Disks) aumenta l'affidabilità di un disco aggiungendo **ridondanza**. Le strategie usate sono molte con l'obiettivo di **aumentare** il tempo medio tra guasti (mean time between failures [MTBF]): ogni dato è **copiato 2 volte** (ovvero per perderlo dovrei perdere entrambe le copie). **Esempio:** ho un disco con 100.000 ore di mean time to failure (MTTF) che richiede 10 ore di riparazione (va ricoppiato manualmente). Il tempo medio per perdere un dato è $100000^2 / 2 * 10 = 500 * 10^6$ ore, cioè 57.000 anni. Ciò significa che se un solo disco fallisce ogni 100.000 ore, statisticamente lo stesso dato fallisce su entrambi i dischi in contemporanea ogni 57.000 anni

⚠ Per valutare la RAID si usano indicatori come **MTTF** (mean time to failure), **MTTR** (mean time to repair [copia su altro disco]), **MTTDL** (mean time to data loss → se mirroring, $MTTDL = \frac{MTTF^2}{2 \cdot MTTR}$), **MIRRORING** (o SHADOWING, tecnica di ridondanza per cui ogni dato è copiato 2 volte su 2 dischi diversi)

12) I/O SYSTEMS

(come il kernel gestisce l'I/O, interfaccia verso le applicazioni, implementazioni hardware di ciò che richiede il software)

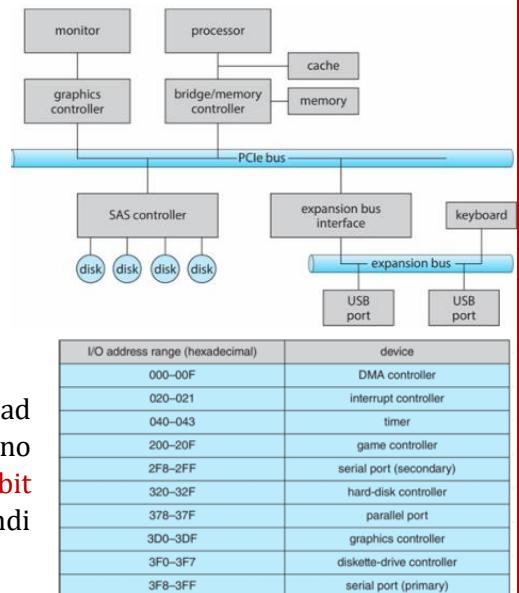
L'I/O è importante in un OS in quanto è usato da tutti i programmi ("il pc non fa il caffè, ma deve poter controllare la macchinetta"). Dal punto di vista hardware, il livello più basso è il **device drivers** (componente dell'OS che pilota direttamente l'hardware).

Distinguiamo **I/O HARDWARE** in hardware di **storage** (memoria), **trasmissione** (schede di rete) e **human interface** (mouse, tastiere...). Riguardo a come questi device si interfacciano con il computer distinguiamo:

- **Porta** = punto fisico in cui il dispositivo si connette alla CPU
- **Bus** = mezzo di trasporto tra periferica e sistema di elaborazione; ce ne sono di vari tipi in base a velocità e interfaccia. I **Daisy Chain** sono meccanismi a catena in cui, se il dispositivo A parla a B, avviene una specie di "passaparola". Fra i bus ci sono **PCI**, **expansion bus**, **SCSI**
- **Controller (host adapter)** = entità attiva, cioè un microcontrollore che è sul dispositivo I/O e si interfaccia con i bus per parlare con la CPU

Come interagiscono i dispositivi I/O con la CPU? I dispositivi sono visti dalla CPU come se avessero delle locazioni di memoria, ma in realtà i device driver mettono comandi/indirizzi/dati su dei registri posti nella periferica. Un dispositivo I/O avrà uno **status register** (letto dalla CPU per capire lo stato del dispositivo), un **control register** (su cui viene scritto il comando da eseguire), un **data-in register** e un **data-out register**. Di solito sono **registri piccoli** (1-4 Byte), mappati in modo che la CPU li veda come **normali indirizzi in memoria**: infatti le CPU possono avere istruzioni assembler specifiche per i dispositivi I/O oppure possono agire normalmente senza sapere di essere di fronte a dispositivi I/O.

Un **esempio** di possibile configurazione dei dispositivi che corrispondono ad indirizzi in I/O è quella nell'immagine qui a lato (proprio perché sono registri piccoli, normalmente questi indirizzi hanno bisogno di **meno bit** rispetto alla RAM [es. nell'esempio qui bastano 3 cifre esadecimali, quindi da 0 a 4 KB]).



Come interagire con un dispositivo I/O? Con il **POLLING** (meccanismo basato su busy-waiting già accennato anche in altri corsi): per ogni dato da leggere e scrivere su I/O dobbiamo:

- per la **scrittura** → aspettare che il dispositivo sia pronto a ricevere un dato
- per la **lettura** → aspettare che il dato sia pronto

Per fare ciò:

1. Leggo il busy bit dallo status register, fin quando non va a 0
2. L'host mette a 1 il read o write bit (ovvero il control register) e scriviamo sul data-in/data-out register
3. L'host (cioè il microcontrollore) imposta l'operazione (es. per una write dice "ho scritto il dato")
4. Il controller dice "sono occupato" (busy bit = 1) e fa il trasferimento
5. Il controller pulisce il busy bit

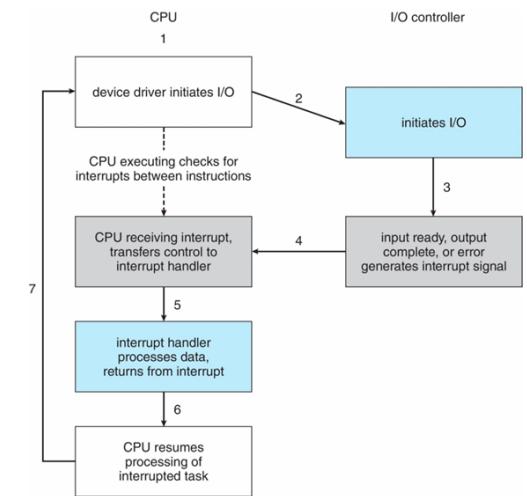
⚠ Il punto 1 però è critico: dato che l'attesa potrebbe essere lunga, se la CPU decide di fare altro mentre è in busy wait, potrebbe perdere l'esatto momento in cui il busy bit va a 0 (senza quindi riuscire ad operare sull'I/O)

L'alternativa al polling è usare **INTERRUPT** (es. se mi devo svegliare alle 8 di mattina: polling = mi sveglio ogni minuto e controllo l'ora, interrupt = metto sveglia alle 8 e mi sveglio solo in quel momento). La CPU ad un certo punto riceve sulla "**Interrupt-request line**" una richiesta di interruzione (attivando un protocollo specifico). L'**interrupt handler** riceve tali richieste di interrupt (le può ritardare o disattivare), mentre l'**interrupt vector** associa dei numeri interi a delle azioni precise per dire cosa fare quando arriva un interrupt, in base a chi lo ha mandato.

La **sequenza di operazioni** è:

1. **device driver** riceve una richiesta di operazione
2. viene passato il **comando** all'hardware di I/O
3. dispositivo fa l'operazione e **manda interrupt**
4. CPU **riceve interrupt** e **attiva interrupt handler**
5. **handler** fa le istruzioni della **ISR** (interrupt service routine) [deve essere molto veloce]
6. CPU termina il processo della ISR e **riprende** da dove era stata interrotta (il 7° step è ritornare al 1°)

⚠ Gli interrupt sono quindi una forma di **eccezione** (non interna, ma esterna) usata per gestire gli errori software/hardware



Non tutto può essere svolto dalla CPU, infatti alcuni trasferimenti sono demandati **direttamente al dispositivo I/O**. Per fare ciò, si usa il **DMA** (Direct Memory Access). Questo meccanismo richiede un **DMA controller** sul device di I/O, ovvero un modulo hardware che pilota i bus per trasferire dalla memoria dei dati. Per "rubare" il controllo del bus, l'OS deve fornire gli indirizzi, il tipo di operazione e la size: esso opera un "**cycle stealing**" (durante cui la CPU non potrà usare il bus) [è un meccanismo molto efficace]

Parlando di **APPLICATION I/O INTERFACE**, vediamo che le applicazioni usano le system call per richiamare le operazioni fatte dal device driver. I **dispositivi** possono essere:

- a flusso di caratteri singoli (come la tastiera)
- a blocchi (come i dischi)
- sequenziale o ad accesso diretto
- sincrono o asincrono
- condivisibile o dedicato
- veloce o lento
- scrittura+lettura, solo lettura o solo scrittura

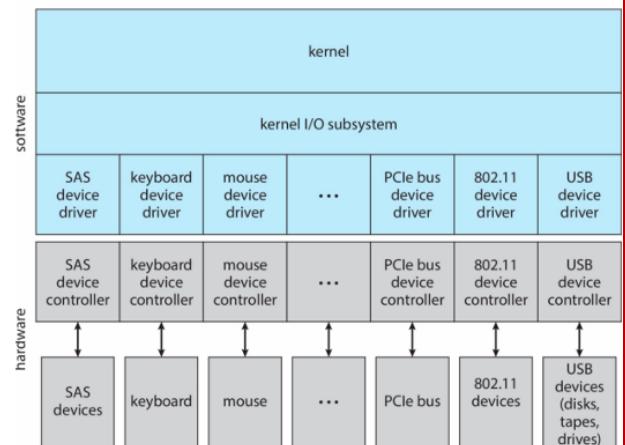
Questa classificazione si vede nell'**architettura del sottosistema di I/O del kernel**: in grigio c'è l'hardware (es. dispositivo I/O e device controller); **per ogni dispositivo** c'è un **device driver** (software) per gestirlo.

I **dispositivi I/O** vengono raggruppati in:

- **Block I/O device** ("a **blocchi**") = esempi sono i **dischi**, che supportano read, write o seek (cioè spostarsi al settore senza fare nulla). Ci sono 2 modalità: **Raw I/O** (senza usare il filesystem, vado direttamente ai settori sul disco) o **Direct I/O** (uso filesystem). È possibile **Memory-mapped file access** (accesso ai file "memory-mapped", ovvero i file sul dispositivo I/O sono mappati virtualmente come se fossero delle struct in memoria). Usano il **DMA**
- **Character I/O device** ("a **caratteri**") = esempi sono **tastiera** e **stampa a video**; normalmente sono disponibili delle **get()** o delle **put()** (tipicamente sequenziali). Non usano il **DMA**
- **Network device** ("dispositivi di rete") = in Linux, Unix e Windows l'interfaccia usata per le reti è il **socket** ed il comando più usato è il **select()** [non li vedremo in questo corso]
- **Clocks & Timers** = non tutti gli I/O sono uguali per scambiare i dati, ma i "programmable internal timer" servono per generare **segnali temporizzati**

Ci sono diverse **tipologie di I/O**:

- **BLOCCANTE (SINCRONO)** → **processo si blocca fin quando l'I/O viene completato**. È facile da implementare, ma spesso non sufficiente



- **NON BLOCCANTE** → processo fa partire l'I/O senza fermarsi ad aspettare; l'I/O torna subito un risultato parziale. Può essere implementato: in automatico (usando dei **buffer** per “parcheggiare” i dati in memoria) o manualmente (con **multi-thread**: 1 thread per all'I/O mentre gli altri fanno altre operazioni). Questo tipo di I/O può ritornare il conteggio dei byte scritti/letti, mentre la `select()` dice se il dato è stato scritto/letto
- **ASINCRONO** → è un **non bloccante**, ma con un **meccanismo per far capire al programma se l'I/O è terminato**; le strategie usate per fare ciò sono 2:
 1. Faccio partire l'I/O, faccio altro e quando ho finito mi metto in **wait del dato dell'I/O**
 2. Specifico al sistema una **callback da eseguire al termine dell'I/O**

⚠ Se devo gestire un vettore di richieste di I/O anziché una singola richiesta, devo usare il **VECTORED I/O**

Dal punto di vista del kernel c'è un problema di **schedulazione**, cioè avere **più richieste di operazioni su dispositivi di I/O**: per gestirle si usa una coda **FIFO**; altri os usano il **'fairness'** (cioè avere un criterio di suddivisione dei servizi tra i vari richiedenti). Un altro aspetto è il **buffering**, ovvero quando devo trasferire dati in output da memoria (**A**) ad un I/O (**B**), li salvo in memoria kernel (*buffer intermedio*) prima di trasferirli all'I/O; questo per vari motivi:

- **Differenza di velocità** tra A e B → trasferisco da A al buffer (liberando A) e poi dal buffer a B (**A così non deve aspettare B**)
- **Differenza tra le dimensioni dei dati trasferiti** da A a B → supponiamo che A scriva a blocchi di 1KB, mentre B legge a blocchi di 4KB: possiamo fare 4 trasferimenti da A al buffer e poi 1 dal buffer a B
- È possibile che un processo voglia scrivere dati su disco e, dopo aver fatto la richiesta di scrittura, voglia **modificare i dati senza aspettare la fine della scrittura** → se ho un duplicato, il buffer intermedio permette di capire chi lo ha fatto e quando (bufferizzare = creare una copia del dato da qualche parte)

⚠ Per fare un'altra copia sul buffer, A deve aspettare che il buffer si sia **svuotato**: per risolvere ciò si può usare il **double buffering**, cioè uso 2 buffer → 1 su cui lo user scrive, 1 su cui il kernel processa e poi si scambiano

Altri tipi di buffering sono:

- ❖ **caching** = una copia ridondante di un'informazione su un dispositivo più veloce, sempre allineata rispetto all'originale (nella pratica è la solita cache)
- ❖ **spooling** = forma di buffering usata solo per i dispositivi di uscita (es. stampanti): è una semplice coda per gestire le richieste per questi dispositivi; le richieste verranno poi eseguite 1 alla volta
- ❖ **device reservation** = gestione del dispositivo in mutua esclusione (attenzione a deadlock)

Vediamo **altre cose sulla gestione dell'I/O da parte del kernel**:

- **Gestione degli errori** → l'OS può cercare più volte per vedere se le cose si sistemano da sole (approccio spesso usato nelle reti), ma in generale c'è la possibilità di tornare codici di errore se l'I/O fallisce
- **I/O protection** → un processo user potrebbe involontariamente/intenzionalmente svolgere azioni “illegali” (errate) e tenendo conto che le operazioni di I/O sono “privilegiate” (in quanto passano dal kernel) c'è bisogno di fare queste operazioni mediante system calls
- **Kernel data structure** → il kernel salva per tutti i dispositivi delle informazioni di stato in delle tabelle; per gestire i buffer, le allocazioni e le azioni dei devices ci sono strutture complicate generalmente dinamiche (o in alcuni OS “object-oriented”)
- **Power management** → anche la gestione dell'alimentazione viene delegata al kernel (così come la gestione delle temperature, etc...) [es. Android cerca di capire come sono connessi i componenti tra loro e questi sono in grado di dire “non sono utilizzato, spegnimi” per risparmio energetico]

Riassumendo:

- Management of the name space for files and devices
- Access control to files and devices
- Operation control (for example, a modem cannot seek())
- File-system space allocation
- Device allocation
- Buffering, caching, and spooling
- I/O scheduling
- Device-status monitoring, error handling, and failure recovery
- Device-driver configuration and initialization
- Power management of I/O devices

Ma nella pratica come eseguo un'operazione a partire dalla richiesta di I/O? Supponiamo una lettura di un file da disco per un dato processo: dobbiamo determinare qual è il nome del dispositivo che contiene il file, tradurre dal nome al dispositivo, mettere i dati dal disco in un buffer e poi dal buffer al processo e infine faccio return.

Gli STREAMS sono un esempio di comunicazione "full-duplex" (bidirezionali); consistono di una head (ovvero il processo) e di un driver (dal lato del dispositivo) e prevedono delle code di lettura e scrittura; sono asincroni all'interno, sono basati su messaggi e flow control

⚠ L'I/O impatta molto sulle performance di un sistema. Per migliorare le prestazioni, dovremo ridurre il numero di context switch, il data copying e gli interrupt

13) FILE SYSTEM INTERFACE

(vedremo come vogliamo che il file system sia fatto [nel capitolo 14 invece vedremo com'è fatto realmente], cosa sono i file, i direttori e che tipo di accessi vogliamo ai dischi, la protezione etc...)

FILE = sequenza di byte che può essere vista come uno spazio di indirizzamento contiguo, a cui possiamo assegnare un "tipo di dato", cioè specificare se il file contiene dati o codice. I file di dati possono essere numerici, caratteri o binari (dipende dal tipo di utilizzo); ci sono poi i file di testo, sorgente ed eseguibili. Gli attributi di un file sono:

- Nome (unica informazione scritta in linguaggio umano)
- Identifier (unico nel file system)
- Type
- Location (puntatore alla posizione del file sul dispositivo)
- Size
- Protection (permessi r-w-x)
- Time, date e user identification

Le operazioni che si possono fare su un file sono: creazione, scrittura, lettura, riposizionamento, troncamento, apertura e chiusura. Per agire su un file, bisogna prima aprirlo e per farlo serve la **Open File Table** (tiene traccia dei file aperti). All'interno dei file c'è il **file pointer** (puntatore all'ultima posizione scritta/letta); dei file va anche salvata la posizione del file sul disco e chi può accedervi.

Dato che un file può essere acceduto da più processi in parallelo c'è un contatore che li conta (**file-open count**), ma ci vuole anche una politica di accessi che gestisca le sincronizzazioni: il **file locking** è implementato fornendo dei **lock** associati automaticamente ad un file (possono essere obbligatori o advisory [avvertono gli altri di cosa stanno facendo]; possono essere anche fatti solo su una porzione e non su tutto il file). Ci sono 2 tipi di lock:

- **Shared** lock → condiviso, ma chi accede al file può farlo solo in lettura
- **Exclusive** lock → esclusivo in quanto voglio modificare il file

Come sono organizzati i dati in un file? [FILE STRUCTURE] 3 opzioni (non rigide tra loro):

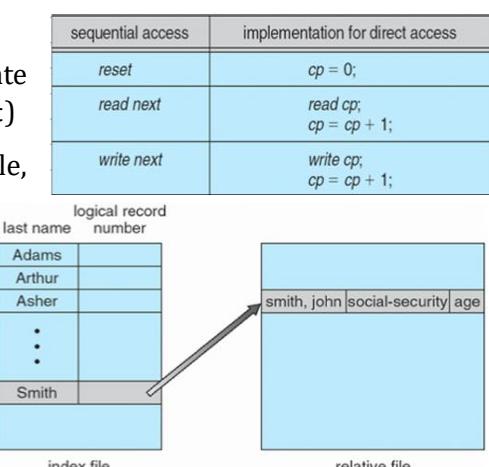
- ❖ **no struttura** = semplice vettore di byte che va interpretato da chi scrive/legge
- ❖ **in record** = file diviso in righe di testo (lunghezza fissa o variabile)
- ❖ **in strutture complesse** = formattate e rilocabili (ogni sezione va interpretata correttamente)

Tipologie di ACCESSO AL FILE:

- **SEQUENZIALE** = file ha posizione iniziale (per il **rewind**), finale e corrente (le uniche operazioni possibili saranno **read next**, **write next** e **reset**)
- **DIRETTO** = file è fixed in "logical records" (che possono essere parole, blocchi...) ed, essendo n = numero di blocco, le operazioni consentite sono **read n**, **write n**, **position to n**; una volta posizionati, si può poi accedere sequenzialmente tramite **read next** e **write next**.

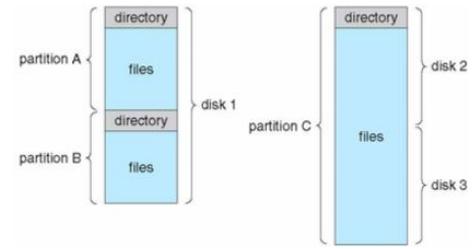
A sinistra è mostrato come si può simulare l'accesso sequenziale nei file ad accesso diretto tramite **current pointer** (cp).

⚠ A partire da questi metodi di accessi si possono realizzare tipologie di accesso più complesse, come creazione di un indice per



il file (nel esempio in figura c'è 1 file che fa da indice e 1 che contiene i dati; il file indice contiene cognome ed indirizzo logico che consente di raggiungere per accesso diretto il 2° file)

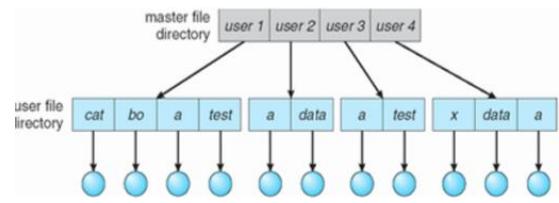
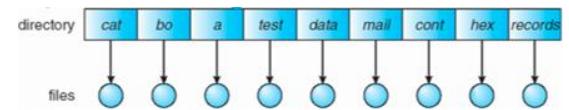
DISK STRUCTURE = un disco può essere diviso in partizioni e può includere una politica RAID. Una **partizione** può essere usata in modalità **raw** (cioè sistemandolo manualmente i blocchi in memoria) o può essere **formattata con un file system**. Ogni **volume** può contenere più partizioni ed 1 solo file system (caratterizzato da directories, tabelle e info varie). Nell'**esempio** qui a lato vediamo che il disco 1 è stato diviso in 2 partizioni (ognuna con il suo file system), mentre i dischi 2 e 3 sono invece uniti come un'unica partizione



DIRECTORY STRUCTURE (direttori) = modalità per poter **collezionare più file in una struttura gerarchica organizzabile su più livelli**; in un direttore ci sono dei nodi e ciascuno corrisponde ad un file. Le **operazioni** fatte su una directory sono cercare, creare, eliminare o elencare files

DIRECTORY ORGANIZATION: i direttori sono organizzati in modo logico per trovare un file in modo efficiente. Si usano quindi strategie per nominare i files (più comodo per l'utente), tenendo conto però che 2 file possono avere lo stesso nome e che un file può avere più nomi diversi [può anche essere utile raggruppare i file per categoria, tipo etc...]:

- **SINGLE-LEVEL DIRECTORY** → i nodi sono i file, i rettangoli sono i direttori; è difficile sia dare nomi (univoci) [**naming**], sia fare raggruppamenti [**grouping**]
 - **TWO-LEVEL DIRECTORY** → a più livelli è molto più semplice raggruppare file e posso anche nominarli allo stesso modo, ma per **utenti diversi** (**risolvo i problemi di naming e grouping**) [devo però riferimi al **path-name**, ovvero il nome composto da nome e percorso] [si possono usare directory a **N livelli** ma ciò complica la ricerca del file]
- Il **current directory (cd)** il direttorio in cui ci troviamo ed il path-name può essere quindi **relativo** (parte dal cd) o **assoluto** (parte dalla radice)



⚠ I direttori non sono necessariamente un albero, perché i filesystem permettono di condividere i nodi (files e directories possono appartenere a più directories). Perciò la struttura utilizzata è un **Grafo-Aciclico** (dunque i file possono avere path-name diversi → **"aliasing"**)

Eliminando un file condiviso da un direttorio, devo eliminarlo automaticamente dagli altri, generando il **dangling pointer** (puntatore a **null**). Ci sono 2 soluzioni:

- **Back pointers** (puntatori all'indietro) = puntano anche ai direttori che contengono il file; quando il file viene cancellato anche gli altri puntatori al file possono vedere la modifica. Si può implementare come vettore o come lista daisychain
- **Tenere il file vivo** finchè non viene cancellato da tutti i direttori

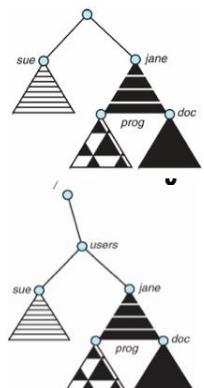
Un altro problema è che se non possiamo vedere globalmente il grafo dei direttori possiamo generare dei **CICLI**. Per evitarlo:

- ✓ Non condivisione di direttori (solo file) [evito cicli per costruzione]
- ✓ Implementare una garbage collection che se trova cicli li rimuove
- ✓ Algoritmo di individuazione dei cicli ogni volta che creo un nuovo link tra i direttori

MOUNTING FILE SYSTEM: fare **mounting** significa agganciare le info che abbiamo descritto al OS con l'idea di **vedere in un unico grafo più dischi**.

C'è **file sharing tra utenti** (servono quindi meccanismi di protezione per decidere chi può scrivere, leggere...): c'è una gerarchia in cui un utente è identificato da **user-id + group-id**.

C'è **file sharing a livello di rete**: ci sono protocolli come FTP o file system distribuiti o quasi in automatico nel world wide web (non li vedremo). Sulla rete bisogna saper gestire gli errori: mentre se non trovo un dato sul disco ho un errore, sulla rete il più delle volte è semplicemente caduta la connessione; occorre anche specificare come sincronizzare i file, dati gli accessi multipli.



Se **più processi user accedono ad un file** il puntatore è condiviso o ognuno ha il suo? I tipi di accesso sono lettura, scrittura, esecuzione, append (in unix ad esempio ci sono l'owner, group e public che hanno ognuno privilegi diversi)

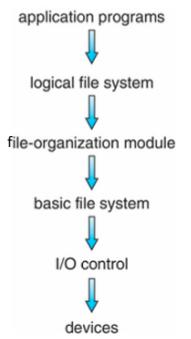
14) FILE SYSTEM IMPLEMENTATION

(struttura effettiva del filesystem, come da un punto di vista logico si organizza un file che dal punto di vista fisico è un insieme di blocchi su un disco)

Il **FILE SYSTEM** ha le informazioni in memoria secondaria (ovvero il **DISCO**, non-volatile [memoria di massa o "secondaria"]], fornisce un'**interfaccia** verso queste informazioni e fornisce una **traduzione logico-fisico**, oltre che un accesso efficiente al disco.

Come abbiamo già detto, il disco lavora per settori/blocchi. Un **file** viene rappresentato come un **oggetto/struct** dal **File Control Block (FCB** → contiene le info per accedere ai dettagli del file). A fine capitolo vedremo il **device driver** (componente che controlla l'I/O).

Nello schema, I/O control e devices hanno direttamente a che fare con l'hw, mentre il file system è organizzato a **livelli (LAYERS)**: il **basic file system**, il **modulo di organizzazione dei file** ed il **file system logico**. Più si va verso l'alto, più il livello di **astrazione** aumenta (sopra ho gli applicativi, ovvero i programmi utente). Vediamo questi layers nei dettagli:



- **DEVICE DRIVERS (I/O control)** → componenti **software** che gestiscono l'**accesso ai dispositivi di I/O**. Non riceve comandi del tipo "read file.txt", ma "read drive 1, cylinder 72, track 2, sector 10" + "porta ciò che leggi in memoria all'indirizzo 1060", ovvero siamo **più vicini all'hardware**
- **BASIC FILE SYSTEM** → informazioni già indipendenti dall'hardware (connesse al **dispositivo logico**). Riceve comandi come "prendi il blocco 123" vedendo il **file come vettore di blocchi**. A questo livello si velocizza tramite dei **buffer** che mantengono i dati in transito (RAM), evitando letture/scritture multiple dal disco (livello astrazione = **singolo blocco**)
- **FILE ORGANIZATION MODULE** → gestisce la **collocazione dei file su blocchi logici su disco**, traduce blocco logico in blocco fisico, gestisce lo **spazio libero** e l'**allocazione** sul disco (astrazione = **gruppo di blocchi**)
- **LOGICAL FILE SYSTEM** → capisce i **metadati** e traduce il nome del file nel **numero** del file (UNIX-like [dopo vedremo anche gli *i-nodes*]) o il **puntatore** al file (Windows [usa gli "handle"]). È il livello più alto a cui si gestiscono le **informazioni** che sono più vicine a ciò che vuole l'**utente**

⚠ La **gestione a strati** riduce la complessità delle operazioni e fornisce interfacce chiare. Di file system inoltre ce ne possono essere vari che differiscono per come codificano il file e per i dispositivi supportati: noi vedremo strategie generiche, qualcosa di UNIX vedendo gli *inodes* e qualche dettaglio su OS161 nella parte internals

Parliamo ora di **FILE SYSTEM OPERATIONS** (ciò che deve fornire l'API di un file system [funzioni da supportare]): per effettuarle servono delle strutture dati gestite dal Kernel, sia su disco che in memoria.

Per prima cosa vanno gestite le informazioni che permettono di fare **bootstrap** (init dell'OS), mediante 2 strutture: Boot Control Block [BCB] (sul **disco**; contiene le **info da leggere subito** per far partire il sistema) e **Volume Control Block [VCB]** (contiene il **superblock** e la **master file table**, che in sostanza contengono info sulle **partizioni** del disco) [*non approfondiamo*]. Sul disco ci sono poi le **strutture a direttori** (collezionare i file e organizzarli per agevolare le ricerche). L'informazione sul **singolo file** è il **File Control Block [FCB]** (contiene **dettagli sul file** relativi ai permessi, alle date di creazione/modifica, il numero di inodes e altre info a seconda del tipo di file).

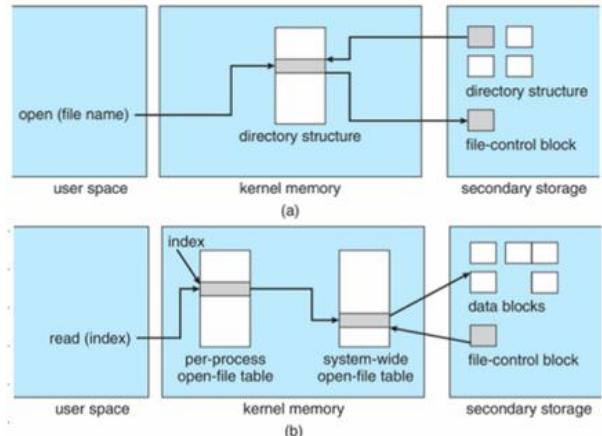
Tutto ciò visto finora è salvato sul disco; quando l'OS fa il **mount** di un filesystem (lo aggancia cioè al disco), in **memoria** ci saranno strutture dati che replicano/completano le info su disco (es. ogni volta che abbiamo gestito un file in C, abbiamo replicato in memoria le info del file → prendi i dati, copiali in una struttura dati in RAM, lavoraci e ricopia tutto sul file [**lavorare in RAM è più comodo**]).

Questo succede anche nella gestione del file system: le strutture dati in memoria che il Kernel usa per lavorare sui file sono una copia di quelle su disco (**ridondanza**). La **mount-table** invece esiste solo in memoria memorizza

le info sui file-system agganciati. Poi ci sono le **system-wide open-file table** (considera tutti i file aperti nell'OS) e **per-process open-file table** (considera i file aperti da ogni singolo processo).

In figura vediamo la **open** di un file (a) e la **read** (b) [analogamente funzionano la **close** e la **write**]. A sx abbiamo lo **user space** (processo utente), al centro la **kernel memory** (strutture dati del kernel), a dx il **disco**. La **open** viene fatta una volta sul file (come una specie di inizializzazione), mentre la **read** viene ripetuta più volte (su ciò che ha generato la **open**). Vediamo la **open** nel dettaglio:

- a) **open** → parte dal nome del file e localizza su disco il FCB: per farlo, serve una **copia della directory structure** (tabella di ricerca nella memoria kernel). Si ottiene quindi il **puntatore** al file (es. `fp = fopen('a.txt0')` in C): nella pratica il **FCB** viene copiato nella **system wide open file table** ed il processo della **open** troverà una entry nella sua **per-process open-file table** → se un altro processo fa **open** dello stesso file, esso non viene copiato nuovamente in memoria, ma la sua **per-process table** avrà una riga che punterà alla stessa copia del file già fatta nella **system-wide table**



⚠ Queste 2 tabelle devono permettere di poter avere puntatori al file diversi per processi diversi (uno legge in testa, l'altro in coda etc...) oppure condividere i puntatori

Parliamo ora di **DIRECTORY IMPLEMENTATION**: i direttori sono una tabella di simboli speciali perché non si cerca in una sola lista/settore, ma si cerca il **path** del file (+ il **nome**) in un **grafo aciclico** (ch13). Si possono vedere i direttori come:

- **liste lineari** di nomi di file (dove ognuno punta al FCB) → **semplici**, ma **ricerche lineari onerose** come tempo
- **hash table** → permettono **accesso diretto** (evitando ricerca lineare), ma hanno il problema dei sottodirettori: se devo cercare un file che ha percorso `/root/sottocartella1/sottocartella2/.../`, devo cercare i vari direttori in liste/tabelle diverse tra loro, oppure bisogna gestire il caso di tabelle multiple

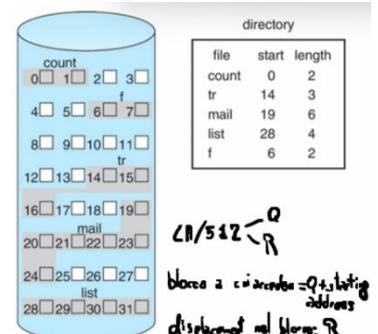
Ora vediamo i vari metodi di **ALLOCAZIONE DEI FILE**:

- **CONTIGUOUS ALLOCATION** → file occupa **blocchi contigui** il cui numero dipende dalla dimensione del file.

In figura si vede la trasformazione indirizzo logico-fisico: il disco è una sequenza numerata di blocchi logici; preso un indirizzo logico ("LA" → logical address), la directory è una tabella che dice che il file **count** parte dal blocco 0 e ne occupa 2, il file **tr** pare da 14 ed è lungo 3 etc...

Facciamo la **read** di un LA: **Q** = **numero blocco**, **R** = **posizione nel blocco** (displacement) → avendo blocchi di 512 bytes, divido LA per la dimensione del blocco e ottengo come quoziente Q e come resto R

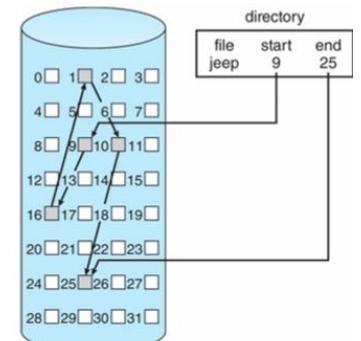
- ✓ **PRO:** semplice
- ✓ **CONTRO:** causa di frammentazione esterna



⚠ **Extended Based System** = schema di allocazione contigua, usato nei sistemi moderni. Un'**extent** è un insieme di blocchi contigui ed un file occupa più extent. Riduce i problemi dell'allocazione contigua allocando non un unico grande intervallo di blocchi contigui, ma tanti intervalli contigui più piccoli

- **LINKED ALLOCATION (linked list)** → file diventa una **lista di blocchi** (no contiguità = **no frammentazione esterna**), favorendo la sequenzialità di lettura dei file [con i processi non potevamo farlo perché l'accesso ai dati di un processo non è sempre sequenziale a differenza dei file]. Ogni blocco contiene il **puntatore al next block** (file termina se puntatore == NULL)

⚠ Problema di affidabilità: se **perdo un blocco**, perdo il suo puntatore al next e quindi tutti i successivi (in figura in questa lista JEEP se perdo il blocco 1, perdo anche 10 e 25)

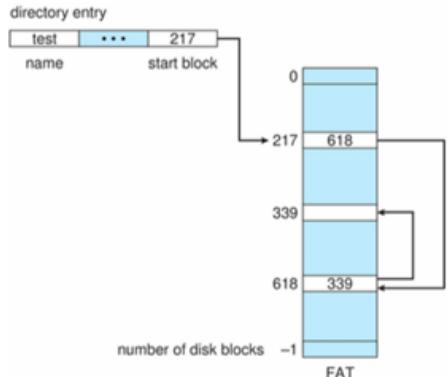


⚠ Le liste sono strutture lineari quindi, se dovessi fare un **accesso sequenziale**, dovrei accedere al blocco n-esimo scandendo tutti i blocchi precedenti

⚠ Una parte della memoria del blocco viene “**mangiata**” dal **puntatore**: nella traduzione da LA, il displacement sarà **R+1**, ovvero conto la dimensione del puntatore (a patto che il puntatore sia all'inizio di ogni blocco) [i dati non partono cioè dall'inizio del blocco ma sono shiftati della dimensione del puntatore, devo considerarlo nel tradurre gli indirizzi]

È stata inventata poi la **File Allocation Table (FAT)** con l'obiettivo di scorporare i puntatori dai blocchi di dato (in modo che, se perdo il blocco, perdo i dati ma non i blocchi successivi): rappresento i puntatori in un vettore/matrice “**FAT**” parallelo ai dati

⚠ La FAT avrà bisogno di essere allocata e porterà via dei blocchi: **ESEMPIO** → ho 1mln blocchi da 1 KB, quindi **1 GB di blocchi di dato**; ogni blocco avrà il suo puntatore nella FAT, che avrà quindi 1mln puntatori. Se un puntatore è da 4 byte, la **FAT peserà 4 MB, molto meno del peso dei dati**

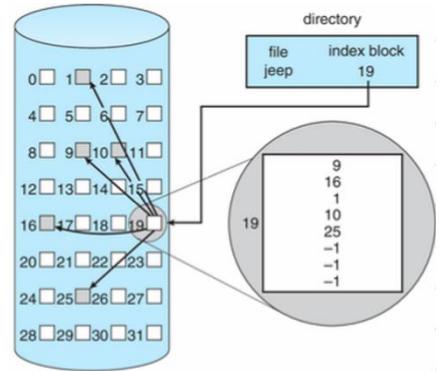


Nella directory (**in figura**) dico che il file **test** inizia al blocco 217: tale blocco non contiene il puntatore al next, ma il puntatore è nella FAT nella riga 217, che punta al blocco 618. Quindi nel vettore dei blocchi di dato (che non c'è in figura) prendo il blocco 618 e poi nella riga 618 della FAT vedo qual è il blocco successivo

⚠ Essendo la FAT molto leggera, posso farne una copia e risolvere eventuali errori

- **INDEXED ALLOCATION** → **accesso diretto** per risolvere l'alto costo di ricerca delle linked lists (come visto prima, se voglio il 100° blocco dovrò scandire tutti i 99 elementi precedenti). La indexed fornisce **per ogni file** il suo “**blocco di indici**”

ESEMPIO: abbiamo il file **Jeep** sempre contenuto nei blocchi 1-9-10-16-26 che però, anziché linkati internamente, hanno dei puntatori in un altro blocco (in questo caso il 19) che funziona come una tabella: se voglio il valore 0 allora sarà nel blocco 9, il valore 1 nel 16 etc... In questo modo ho accesso diretto



La **traduzione logico-fisico** è simile a quella dell'allocazione contigua: **LA / 512 = Q e R**, però una volta trovato l'indice Q, ci posso arrivare in modo diretto.

Ma se il **numero di indici** che possono stare in un blocco **non bastano**? 2 soluzioni:

- **Linked list di blocchi di indici**: la ricerca sequenziale costa, ma solo sui blocchi di indici (che sono meno dei blocchi di dati).

ESEMPIO: ogni blocco della tabella indici contiene 511 indici + 1 puntatore alla next tabella di indici; ogni indice contiene 1 puntatore ad un blocco di 512 dati → **per tradurre logico-fisico: LA / (512 * 512) = Q₁ e R₁** (Q_1 = blocco di indici in cui cercare l'indice); **per ottenere la posizione dell'indice nel blocco: R₁/512 = Q₂ e R₂** (Q_2 = posizione dell'indice nel blocco; R_2 = posizione dell'indice nel blocco))
- **Blocchi di indici a 2 livelli** (sempre accesso diretto): 1 blocco contiene gli indici per arrivare ad 1 dei blocchi di indici da cui prenderò il puntatore al blocco di dato (simili alle tabelle gerarchiche nella page table).

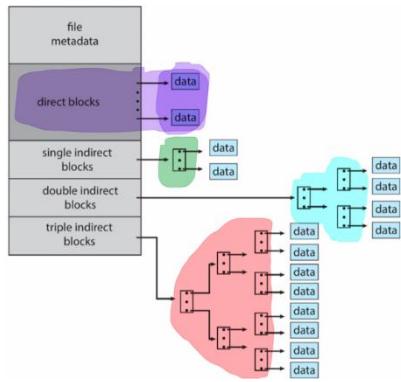
ESEMPIO: traduzione: **LA / (512 * 512) = Q₁ e R₁** (Q_1 = riga dell'indice interno), poi **R₁/512 = Q₂ e R₂** (Q_2 = riga dell'indice del blocco; R_2 = displacement)

⚠ In **UNIX/Linux** si usa uno schema basato su **I-node**, cioè una **tabella di indici gerarchica ma sbilanciata**. La tabella è come un **albero ruotato di 90°**, la cui **root** è il **FCB** (chiamato **i-node**). Anziché avere un File System dove tutti i file avranno n-livelli di indici, uso **meno indici se il file è piccolo mentre di più se è grande**. L'i-node, contiene i metadati del file e i puntatori ad alcuni blocchi diretti.

Esempio: i primi 10 blocchi di dato sono direttamente nel FCB; se il file è piccolo non serve nient'altro, altrimenti anche i blocchi dall'11° in poi sono puntati da un blocco di indici (contenuto sempre nell'i-node).

Se il file consuma questo 1° blocco di indici, avrà un altro blocco di indici a 2 livelli; se anche questo è pieno si usa un ulteriore blocco a 3 livelli... Quindi la maggior parte della capienza dei blocchi di dato è dal triplo livello. Se 1 blocco può contenere 512 indici, triple indirect block potrebbe indirizzare 512^3 blocchi di dato.

Globalmente ho **numero di blocchi di dato totale** = $5123 + 5122 + 512 + 10$



⚠ **PERFORMANCE:** **CONTIGUA** va bene per accesso **sequenziale** e **diretto**; **LINKED** per accesso **sequenziale** (già detto prima); **INDEXED** per accesso **diretto** e senza frammentazione esterna, ma più **difficile** da implementare e accedere ad un blocco di dato può richiedere **molti accessi in lettura** perché devo leggere sia indice sia dato (come nella page table gerarchiche) [una soluzione sarebbe **allocare i blocchi di dato vicini** in modo da entrare in uno e spostarsi agli altri senza passare di nuovo per il blocco degli indici; oppure esecuzioni parallele]

Dopo aver visto come si allocano i dati, vediamo la **GESTIONE DELLO SPAZIO LIBERO**. Il file system ha una **free-space list** per i blocchi disponibili oppure una **bit vector** o **bit map** parallela ai blocchi che mette un 1 se il blocco è libero, 0 se è occupato.

Servono però **algoritmi** per gestirla (vedremo esempio bitmap OS161), ma **ad ogni blocco serve 1 bit aggiuntivo**

Esempio: blocchi di 4KB (2^{12} byte), disco da 1TB (2^{40} byte). Il **n° blocchi** sarà $2^{40}/2^{12} = 2^{28}$ blocchi, ovvero una bitmap di 32MB. **Ottimizzando compattando 4 blocchi contigui in un cluster**, otterò bitmap da 8MB.

Altrimenti si può usare la **free-list**, implementata come una linked list di cui salvo solo in testa il 1° blocco libero e poi ogni blocco punta al next; questo ha:

- **pro:** non occupo memoria aggiuntiva
- **contro:** se voglio trovare un blocco libero in una posizione specifica (es. per allocare dati in modo contiguo), dovrò fare la solita ricerca lineare che scandisce tutta la lista

Per **ottimizzare la free-list** posso fare **Grouping**: prendo blocco libero e ci metto puntatore a next + n puntatori ai blocchi successivi della lista; si può anche fare **Counting** (tengo il puntatore al prossimo blocco libero e conto i blocchi liberi contigui successivi)

⚠ **TRIMMING UNUSUED BLOCKS** = mentre gli **HDD** consentono di sovrascrivere un blocco, gli **SSD** hanno bisogno di cancellare il blocco e poi riscriverlo (questo per capire che alle volte free-list e bitmap non bastano)

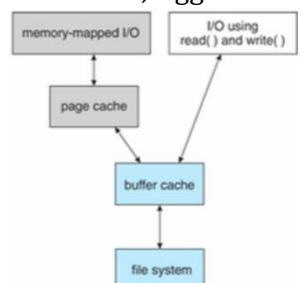
⚠ **EFFICIENZA del file system:** dipende da strategie di allocazione scelte, dalle ricerche sui direttori, dalle tabelle usate dai direttori e se i dati sono a dimensione fissa o variabile

⚠ **PERFORMANCE (pt.2):** mantenere i dati vicini influenza le prestazioni (più grossa lettura in unico accesso). Incide sulle prestazioni anche il **buffer cache** (mettere i blocchi più usati in RAM).

Bisogna anche decidere se usare **I/O SINCRONO** (attendo che finisca) o **ASINCRONO** (mando la scrittura e nel frattempo faccio altro; > veloce, ma > complesso). Ci sono poi **free behind** e **read-ahead** (letto un blocco, leggo anche i successivi).

Ultimo aspetto prestazionale è la **page cache** (usata per il **memory-mapped I/O** → si virtualizzano i blocchi del disco sulle pagine di RAM → si crea una corrispondenza tra pagine in RAM e blocchi sul disco e si usa la page cache per tracciarla).

Essendo però che anche il file system ha il suo buffer cache (per l'I/O, vedi sopra), rischio di avere **2 livelli di caching (inefficiente)** → in UNIX si fa una cache unificata [questo per dire: "Attenti che in un'ottica architetturale **duplicando le strategie di ottimizzazione si sputtana tutto**".



Ultimo aspetto è il **RECOVERY**: per evitare errori, bisogna fare il check di dischi, ma l'unico modo è avere un backup e delle politiche di restore. Per tracciare gli errori si usano dei **log structured** (salvo ad ogni iterazione lo stato attuale e ciò che sto per fare → se c'è un errore non perdo nulla)

OS161

0) INTRODUZIONE OS161

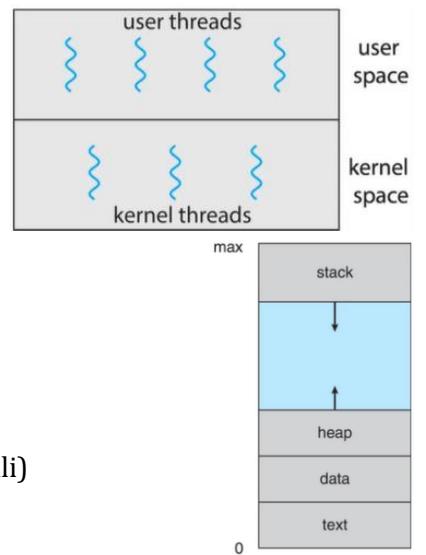
PROCESSO/THREAD = programma in esecuzione con i suoi dati (programma = entità passiva; processo = entità attiva); lo **stato/contesto** è rappresentato da stato della CPU + stack. La memoria associata ad un processo è composta da **codice, dati e stack** (solo questo fa parte del **contesto** di un programma in esecuzione).

Le librerie viste finora (es. POSIX threads) possono solo generare “**user threads**” (che possono eventualmente essere **mappati** su kernel threads); i “**kernel threads**” sono invece generati dal kernel e verranno schedulati come unità separate (vengono proprio **creati** come nuove entità). Ci sono diverse modalità di **MAPPING** “user threads – kernel threads”:

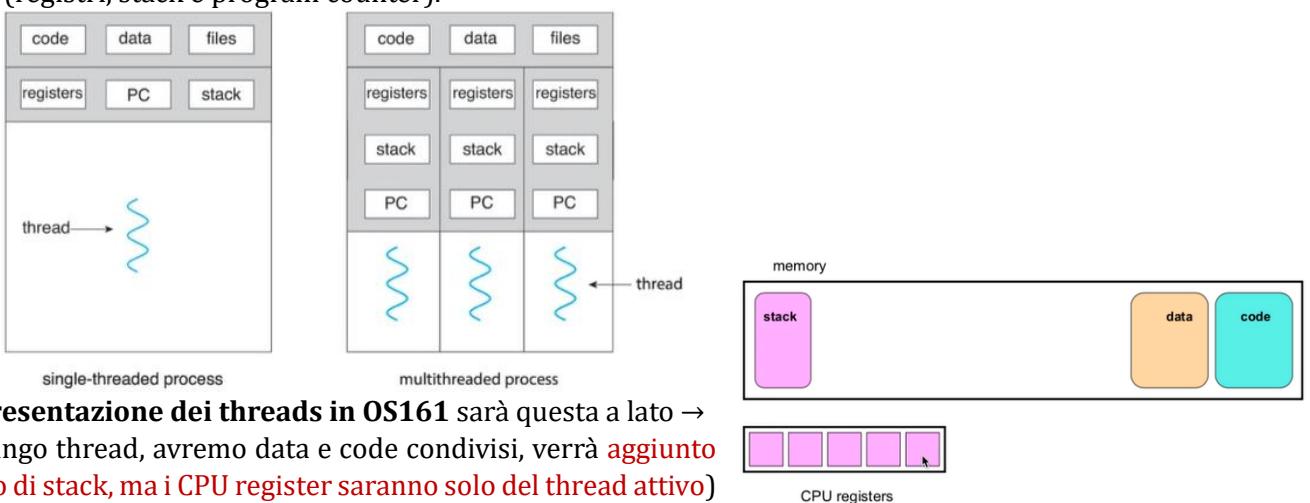
- *Many-to-One* (più user threads su 1 kernel thread)
- *One-to-One* (1-1)
- *Many-to-Many* (mappature multiple)

Un processo ha un suo address space ed è composto da **più parti**:

- **codice** (text section)
- **attività** corrente (program counter + processor registers)
- **stack** (dati temporanei: parametri funzioni, indirizzi di ritorno, variabili locali)
- **data section** (variabili globali)
- **heap** (memoria allocata dinamicamente run-time)



Un processo può avere **1 o + threads**; come si vede nell'img, i singoli threads di 1 processo hanno solo il contesto separato (registri, stack e program counter).



La **rappresentazione dei threads in OS161** sarà questa a lato → (se aggiungo thread, avremo data e code condivisi, verrà **aggiunto un blocco di stack**, ma i **CPU register saranno solo del thread attivo**)

In **OS161** (come abbiamo leggermente visto nel 1° laib) c'è una **libreria di threads**, che si basa su una **struct C** chiamata “**thread**” con struttura:

```
/* see kern/include/thread.h */

struct thread {
    char *t_name;
    const char *t_wchan_name;
    threadstate_t t_state;

    /* Thread subsystem internal fields. */
    struct thread_machdep t_machdep;
    struct threadlistnode t_listnode;
    void *t_stack; /* Kernel-level stack */
    struct switchframe *t_context; /* Saved Register context (on stack) */
    struct cpu *t_cpu; /* CPU thread runs on */
    struct proc *t_proc; /* Process thread belongs to */
    ...
};
```

FUNZIONI di `thread.h`: ne useremo molte, cerca di vedere la [documentazione](#). Qua ne vedremo qualcune:

- `int thread_fork (const char* name, struct proc* proc, void (*entrypoint)(void *, unsigned long), void *data1, unsigned long data2)` → **crea un thread** con nome name, appartenente al processo proc, che starterà l'esecuzione della **funzione puntata da entrypoint**, che riceverà **2 parametri** (void *, unsigned long) [questi 2 parametri sono quelli indicati dopo nei parametri della fork, ovvero (void *data1, unsigned long data2)]
⚠️ Di fatto, la `thread_fork` fa una `thread_create` (alloca la struct thread), fa una `switchframe_init` (salva il contesto sullo stack [viene creato un switchframe nello stack]) e poi fa una `thread_make_runnable` (imposta stato del thread a **READY** e lo aggiunge alla **coda della CPU** [non viene fatto partire immediatamente]). Ad un certo punto ci sarà una chiamata a `thread_switch` (riceve un nuovo stato per il thread in esecuzione; se lo stato è **READY**, questo viene tolto dalla coda e diventa **RUNNABLE**; si toglie quindi il thread dall'esecuzione corrente e si fa partire l'esecuzione del prossimo [preso dalla testa della coda della CPU; viene anche fatta una `switchframe_switch` per mettere il contesto del thread come corrente (ovvero fa **context switching**)])
- `_DEAD void thread_exit (void)` → **fa la exit del thread corrente** (gli interrupt non devono essere disattivati)
- `void thread_yield (void)` → **fa uscire dall'esecuzione il thread corrente** (volontariamente il thread si ferma e lascia la CPU a qualcun altro)
- `thread_switch (threadstate_t newstate,...)` → **chiama al suo interno in assembler** la `switchframe_switch` (salva i regisitri/contesto nello stack e lo mette nel prossimo thread da eseguire)

⚠️ Essendo che OS161 gira su un processore MIPS, rivediamo l'uso dei registri (visto a Calcolatori Elettronici):

<code>R0, zero = ## zero (always returns 0)'</code>	<code>R08-R15, t0-t7 = ## temps (not preserved by subroutines)'</code>
<code>R1, at = ## reserved for use by assembler</code>	<code>R24-R25, t8-t9 = ## temps (not preserved by subroutines)'</code>
<code>R2, v0 = ## return value / system call number</code>	<code>## can be used without saving</code>
<code>R3, v1 = ## return value</code>	<code>R16-R23, s0-s7 = ## preserved by subroutines</code>
<code>R4, a0 = ## 1st argument (to subroutine)'</code>	<code>## save before using,</code>
<code>R5, a1 = ## 2nd argument</code>	<code>## restore before return</code>
<code>R6, a2 = ## 3rd argument</code>	<code>R26-27, k0-k1 = ## reserved for interrupt handler</code>
<code>R7, a3 = ## 4th argument</code>	<code>R28, gp = ## global pointer</code>
	<code>## (for easy access to some variables)'</code>
	<code>R29, sp = ## stack pointer</code>
	<code>R30, s8/fp ## 9th subroutine reg / frame pointer</code>
	<code>R31, ra = ## return addr (used by jal)'</code>

Vediamo 3 Kernel thread tests:

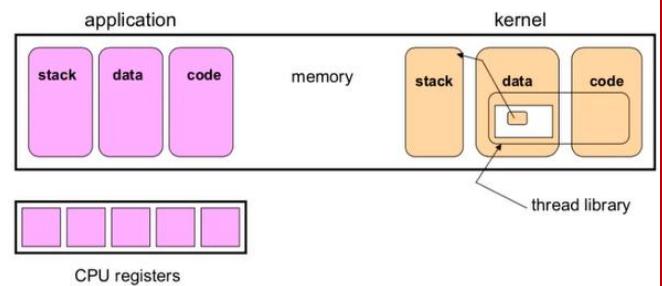
- **tt1** = chiama `threadtest→runthreads(1/*loud*)` per creare NTHREADS (8) threads che runnano **loudthread**; quindi 8 threads che mischiano l'output di chars (120 chars ognuno)
- **tt2** = chiama `threadtest2→runthreads(0/*quiet*)` per creare NTHREADS threads che fanno **quietthread**; quindi 8 threads che fanno busy wait (200000 per iterazione) seguita da output di 1 char (0...7)
- **tt3** = chiama `threadtest3→runtest3` per creare un certo di numero di threads che fanno sleep o sync

Cosa succede internamente quando invece **creiamo un processo utente**? In memoria ci sarà sempre il **kernel**, ma avremo in memoria da qualche parte anche il codice, i dati e lo stack dell'**applicazione utente**.

La struttura del processo utente in OS161 è il PCB (Process Control Block) ed è definita dentro `kern/include/proc.h` così:

```
struct proc {
    char *p_name;                                // Nome del processo
    struct spinlock p_lock;                      // Lock sul processo
    unsigned p_numthreads;                        // N° threads del processo
    struct addrspace *p_addrspace;                // Virtual address space (Virtual memory)
    struct vnode *p_cwd;                          // Working directory corrente
}
```

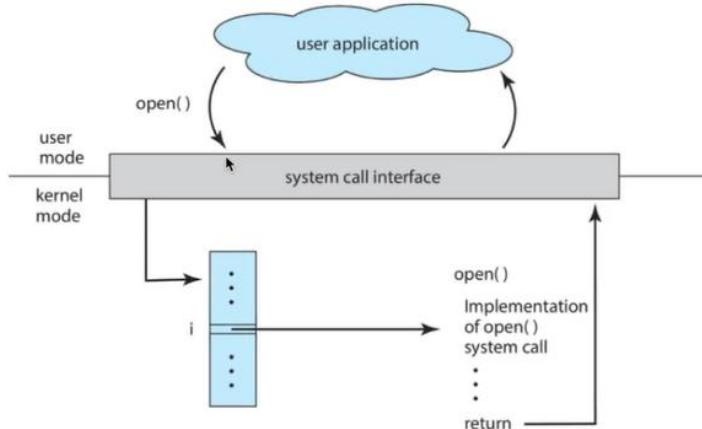
⚠️ Abbiamo che i **thread** contengono il puntatore al loro **processo**, ma il **processo** contiene solo il **nº di thread**; come si vede dall'img, in OS161 ogni thread ha **2 stack**: 1 per quando il thread è in "**user mode**" (esegue **applicazione utente** non privilegiata), 1 per la "**kernel mode**" (esegue il **kernel**)



Quindi, quando un processo è in esecuzione avremo questa configurazione (img dx); per eseguire un processo utente, scrivo da terminale `p nome_programma {<args>}` (quando chiamo questo comando, avvengono una serie di comandi concatenati: `cmd_prog` → `common_prog` → `proc_create_runprogram` → `thread_fork` → `cmd_progthread` → `runprogram`).

⚠ int `runprogram(char *progname)` → apre il file (`vfs_open`), crea un nuovo address space (`as_create`) e lo attiva (`proc_setas + as_activate`). Poi viene caricato il programma dal file (`load_elf`), si chiude il file (`vfs_close`), si definisce lo stack user (`as_define_stack`) e poi in assembler `enter_new_process` (questa funzione trucca un avvio di un processo utente facendo finta che ci sia un processo utente che ritorna da un thread o da un interrupt/trap [infatti qui non si passa dallo `switchframe`, ma da una `trapframe`])

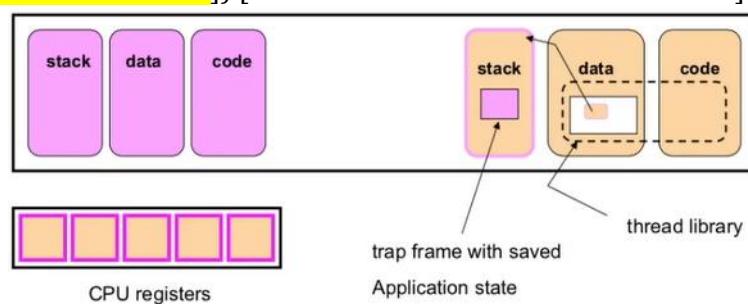
In un OS, le **SYSTEM CALLS** sono il livello intermedio che permette all'utente di ricevere servizi dalle varie componenti dell'OS; sono solitamente scritte in C/C++. Una syscall è `ssize_t read(int fd, void *buf, size_t count)` (dove `fd` è il file descriptor, `buf` è il buffer da dove i dati verranno letti, `count` è il n° max di byte). Questa che vediamo però non è il tratto di kernel eseguito dalla syscall, ma solo l'**API** (interfaccia per l'utente); l'interfaccia mantiene una **tabella indicizzata con i numeri delle syscall** e questi numeri indicano al kernel quale procedura va eseguita (passando quindi da user mode a kernel mode tramite un'interruzione/trap). Qui si vede un **esempio** con la `open`:



Ma se c'è questa associazione numerica tra syscall API e kernel, come posso passare i parametri della syscall al kernel? Potrei mettere i parametri in dei **registri** (ma non so se ci stanno tutti), potrei metterli in un **blocco** (Linux) o potrei metterli nello **stack** (push).

⚠ Quindi quando chiamo delle **funzioni di libreria** (come la `printf`), queste sono un **layer** sotto cui ci sono le **syscall** effettive (con le loro API e con le procedure nel kernel)

In OS161 c'è un **UNICO HANDLER** che viene chiamato per le **eccezioni** e per gli **interrupts**; per ogni eccezione/interrupt avremo quindi una **funzione handler** all'interno di questo handler. Come dicevamo prima, quando faccio una **trap**, avrò un **trapframe** (che funziona come lo `switchframe`, ma avviene durante le interruzioni [**eccezioni, interrupt o passaggio user-kernel mode**]) [una sorta di cambio di contesto forzato]:



Come dicevamo sopra, l'**unico handler** (che conterrà varie sottofunzioni/switch-case in base all'interruzione) viene chiamato con la **void syscall(struct trapframe *tf)**, che ha codice 8:

```

EX_IRQ      0    /* Interrupt */
EX_MOD      1    /* TLB Modify (write to read-only page) */
EX_TLBL     2    /* TLB miss on load */
EX_TLBS     3    /* TLB miss on store */
EX_ADEL     4    /* Address error on load */
EX_ADES     5    /* Address error on store */
EX_IBE      6    /* Bus error on instruction fetch */
EX_DBE      7    /* Bus error on data load *or* store */
EX_SYS     8    /* Syscall */
EX_BP       9    /* Breakpoint */
EX_RI      10   /* Reserved (illegal) instruction */
EX_CPU     11   /* Coprocessor unusable */
EX_OVF     12   /* Arithmetic overflow */

```

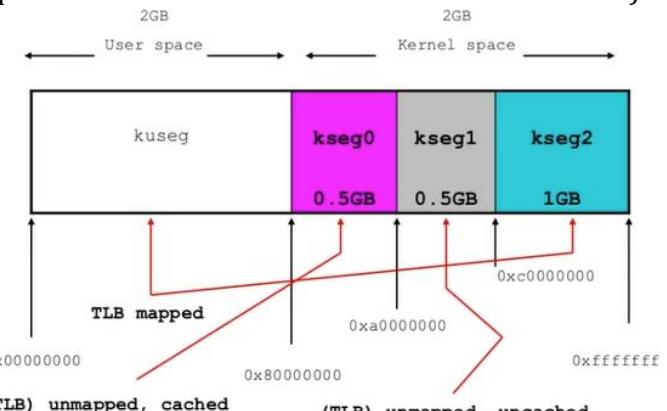
OS161 ha come **syscalls** per la gestione di processi **fork, execv, _exit, waitpid e getpid**.

MEMORY MANAGEMENT OS161:

- Ogni processo ha il suo **address space**, ma il Kernel? 2 possibilità:
 - o **Kernel su spazio fisico** → no traduzione indirizzi logici-fisici, ma usa **solo indirizzi fisici**: in kernel mode (o "privileged" mode) non usare MMU, ma usare gli indirizzi nelle istruzioni come indirizzi fisici
 - o **Kernel in un address space virtuale separato** → **usa traduzione logico-fisico**, ma **separa user address space e kernel address space**: quando cambio modalità però, devo cambiare indirizzi (es. page tables)

OS161 usa una via di mezzo: **indirizzi tradotti logico-fisico, ma address space user-kernel unificato**; il **kernel è mappato in una porzione degli address space virtuali di ogni processo** (in questo modo posso vedere solo la giusta porzione degli address space dei processi a seconda della modalità in cui mi trovo) [questo permette al kernel di vedere il processo utente in modo più semplice]

⚠ Dalla foto vediamo che dei 2GB di Kernel Space: **kseg0** non è mappato in TLB (non si prevede paginazione [traduzione logico-fisico mediante tabella], ma solo una traduzione veloce basata su relocation) ma usa la cache [**contiene kernel code e dati**]; **kseg1** non è mappato in TLB e non usa la cache (qui ci andranno **dispositivi di I/O**, quindi non ha senso velocizzare con cache perché sono lenti di base); **kseg2** è mappato in TLB [**non usato**]



- **TLB di OS161** è *software-controlled* e può avere max 64 entries; ogni riga contiene n° pagina, n° frame fisico, id address space e vari flag. Ci sono funzioni di basso livello per gestire la TLB (**tlb_write, tlb_random, tlb_read, tlb_probe**).

Se la MMU non riesce a tradurre logico-fisico usando la TLB, viene lanciata un'**eccezione** ("**page fault**"). Il **gestore di traduzione logico-fisico di OS161** è **DUMBVM** (ovvero gestisce traduzione logico-fisico degli indirizzi per kernel e user programs, usando la TLB); lo spazio di indirizzamento virtuale è nella struct **addrspace** (sopra citata; **definisce spazio di codice, di data e stack [contigui]**):

```

struct addrspace {
#if OPT_DUMBVM
    vaddr_t as_vbase1; /* base virtual address of code segment */
    paddr_t as_pbase1; /* base physical address of code segment */
    size_t as_npages1; /* size (in pages) of code segment */
    vaddr_t as_vbase2; /* base virtual address of data segment */
    paddr_t as_pbase2; /* base physical address of data segment */
    size_t as_npages2; /* size (in pages) of data segment */
    paddr_t as_stackpbase; /* base physical address of stack */
#else
    /* Put stuff here for your VM system */
#endif
};

```

Come avviene l'**effettiva traduzione logico-fisico**? Abbiamo detto che avviene tramite hardware (quindi MMU + TLB), ma se abbiamo un **page fault**? In questo caso, anziché la tabella delle pagine, viene usato

direttamente l'address space per riuscire a cercare se l'indirizzo logico appartenga a code, data o stack: una volta trovato a quale segmento appartiene, si tenta una traduzione logico-fisico mediante **“relocation”**.

⚠ Abbiamo più volte citato **ELF** (Executable and Linking Format), ovvero un formato di file eseguibile per il quale troviamo la corrispondente chiamata alla creazione del processo nella chiamata alla syscall **execv** (crea il processo da associare ad un eseguibile) [a differenza della **runprogram** (dove è il **kernel** che crea un processo utente da assegnare ad un exe; chiamata con **p programma**), la **execv** viene comandata da un **processo utente**]. Un **file ELF** contiene un **header** e **2 segmenti** (**code** e **data**) che andranno rispettivamente caricati nell'address space virtuale del processo (dall'elf descritto); per ogni segmento, l'**ELF** include un **image** e un **header** (che dice **dove inizia il segmento, la length del segmento, dove inizia l'image nell'elf e la length dell'image**). Quindi per fare **load** di un file **ELF** dobbiamo avere i 2 segmenti (code e data) e caricare in entrambi una porzione del file elf.

1) MEMORIA OS161

Il gestore di memoria che vedremo è (come già detto) **DUMBVM**: si basa su **allocazione contigua**, allocando **multipli di pagine** (4096 byte frame). 2 funzioni utili sono:

- **getppages** → chiama la **ram_stealmem** (in mutua esclusione) [in **dumbvm.c**]
- **ram_stealmem** → **alloca** memoria RAM contigua partendo da **firstpaddr** (che viene incrementato) [in **ram.c**]

```
static paddr_t
getppages(unsigned long npages) {
    paddr_t addr;

    spinlock_acquire(&stealmem_lock);

    addr = ram_stealmem(npages);

    spinlock_release(&stealmem_lock);

    return addr;
}
```

```
paddr_t ram_stealmem(unsigned long npages)
{
    paddr_t paddr;
    size_t size = npages * PAGE_SIZE;

    if (firstpaddr + size > lastpaddr) {
        return 0;
    }

    paddr = firstpaddr;
    firstpaddr += size;

    return paddr;
}
```

⚠ DUMBVM: si chiama **“DUMB”** (stupido) perché non viene tenuta traccia della memoria presa da **ram_stealmem**

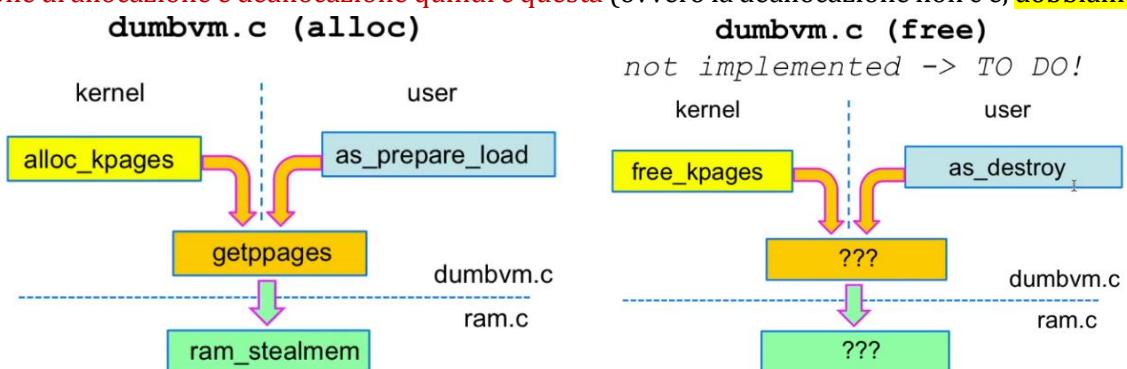
L'allocatore che vedremo è **usabile sia da user sia da kernel memory** (questo lo abbiamo già visto nell'ultima img sopra [quella con kuseg, kseg0...]).

Si passa da user a kernel con addizione e sottrazione di **2GB** di spazio agli indirizzi (semplice aritmetica di indirizzi) → ovvero avrà indirizzo **logico 0x80000000** corrispondente all' indirizzo **fisico 0x0**, avrà indirizzo **logico 0x80000200** corrispondente all'indirizzo **fisico 0x200**, avrà indirizzo logico **0x80100000** corrispondente all'indirizzo **0x100000** (max indirizzo della ram di mips [os161]) [*]

All'indirizzo **0x8003b000** (logico = **firstfree**; fisico = **0x3b000 = firstpaddr**) ho il 1° blocco di memoria libera; quindi **DUMBVM** (con le sue funzioni viste sopra) **opera/alloca solo da questo indirizzo in poi** (ignorando i precedenti; questo **lo si vede dal bootstrap di os161**)

⚠ Quindi: **firstpaddr = firstfree - MIPS_KSEG0** (in quanto **MIPS_KSEG0 = 0x80000000 = 2GB**) [*]

La situazione di **allocazione e deallocazione** quindi è questa (ovvero la deallocazione non c'è, dobbiamo farla noi):



Come possiamo implementare la deallocazione? Soluzione proposta di DEALLOCATORE in DUMBVM:

- Allocazione **contigua** a pagine **comune a kernel e a user**
- Uso di **bitmap** per tenere traccia delle pagine/frames allocati (qui useremo il termine pagina sia per la pagina fisica sia per la logica); quando bisogna allocare, si **cercano frame disponibili tra quelli precedentemente allocati e ora liberati ("riciclatore di pagine")**, se non ce ne sono si chiama la **ram_stealmem**
 - ⚠ Questa bitmap è implementata però come un semplice **vettore di char** (e non di bit); 1 = liberato, 0 = allocato (ma non liberato ancora)
- Visto l'allocatore, quindi vediamo il **deallocator**: per poter deallocare (**free**) dobbiamo avere il **puntatore alla 1^ pagina dell'intervallo** e la **size** (**n° di pagine contigue da fare free**); per tenere traccia di ciò, dobbiamo anche avere una **tavella** che associa **puntatore-size**. Avremo infatti 2 funzioni:
 - **void free_kpages(vaddr_t addr)** → qui serve la tabella perché passiamo solo il puntatore
 - **void as_destroy(struct addrspace *as)** → qui non serve perché size è salvata in address space

Come si può realizzare questo programma (lo vedremo poi bene nel lab [DEALLOCATORE in dumbvm.c]):

Global variables (and test function)

```
static struct spinlock freemem_lock = SPINLOCK_INITIALIZER;

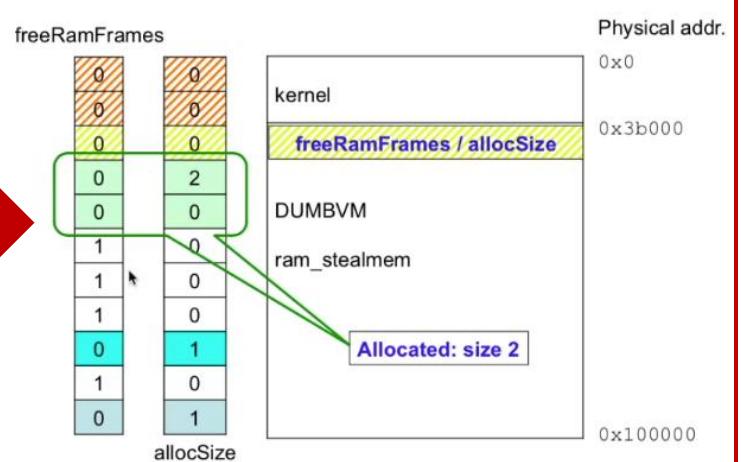
static unsigned char *freeRamFrames = NULL;
static unsigned long *allocSize = NULL;
static int nRamFrames = 0;

static int allocTableActive = 0;

static int isTableActive () {
    int active;
    spinlock_acquire(&freemem_lock);
    active = allocTableActive;
    spinlock_release(&freemem_lock);
    return active;
}
```



Dumbvm



Vediamo che questi vettori hanno una porzione sprecata a causa della parte di RAM del kernel non allocabile. Nell'esempio qui in figura, vediamo (nella parte verde) che sono state **allocate 2 pagine/frames** (**allocSize** ha elemento 2) e che queste sono ancora occupate (non libere; **freeRamFrames** ha elemento 0); poi invece c'è un intervallo con **3 pagine libere contigue** (i tre 1 di fila nel **freeRamFrames**).

Vediamo quindi le **2 funzioni prima citate (free_kpages e as_destroy)**:

```
void free_kpages(vaddr_t addr) {
    if (isTableActive()) {
        paddr_t paddr = addr - MIPS_KSEG0;
        long first = paddr/PAGE_SIZE;
        KASSERT(nRamFrames>first);
        freeppages(paddr, allocSize[first]);
    }
}

void as_destroy(struct addrspace *as) {
    dumbvm_can_sleep();
    freeppages(as->as_pbase1, as->as_npages1);
    freeppages(as->as_pbase2, as->as_npages2);
    freeppages(as->as_stackpbase, DUMBVM_STACKPAGES);
    kfree(as);
}
```

⚠ Notiamo che **vaddr** = indirizzo **logico** nello spazio virtuale, **paddr** = indirizzo **fisico** (nella **free_kpages**)

Per quanto riguarda l'**inizializzazione** della RAM (**vm_bootstrap**) →

```
void vm_bootstrap(void) {
    int i;
    nRamFrames = ((int)ram_getsize())/PAGE_SIZE;
    /* alloc freeRamFrame and allocSize */
    freeRamFrames = kmalloc(sizeof(unsigned char)*nRamFrames);
    allocSize = kmalloc(sizeof(unsigned long)*nRamFrames);
    if (freeRamFrames==NULL || allocSize==NULL) {
        /* reset to disable this vm management */
        freeRamFrames = allocSize = NULL; return;
    }
    for (i=0; i<nRamFrames; i++) {
        freeRamFrames[i] = (unsigned char)0; allocSize[i] = 0;
    }
    spinlock_acquire(&freemem_lock);
    allocTableActive = 1;
    spinlock_release(&freemem_lock);
}
```

Ora vediamo le 2 funzioni sopra richiamate dalle altre funzioni, ovvero:

getppages

```
static paddr_t getppages(unsigned long npages) {
    paddr_t addr;

    /* try freed pages first */
    addr = getfreepages(npages);
    if (addr == 0) /* call stealmem */
        spinlock_acquire(&stealmem_lock);
        addr = ram_stealmem(npages);
        spinlock_release(&stealmem_lock);
    }
    if (addr != 0 && isTableActive()) {
        spinlock_acquire(&freemem_lock);
        allocSize[addr/PAGE_SIZE] = npages;
        spinlock_release(&freemem_lock);
    }
    return addr;
}
```

getfreepages

```
static paddr_t getfreepages(unsigned long npages) {
    paddr_t addr;
    long i, first, found, np = (long)npages;

    if (!isTableActive()) return 0;
    spinlock_acquire(&freemem_lock);
    // Linear search of free interval
    for (i=0,first=found=-1; i<nRamFrames; i++) {
        if (freeRamFrames[i]) {
            if (i==0 || !freeRamFrames[i-1])
                first = i; /* set first free in an interval */
            if (i-first+1 >= np)
                found = first;
        }
    }
    if (found>=0) {
        for (i=found; i<found+np; i++) {
            freeRamFrames[i] = (unsigned char)0;
        }
        allocSize[found] = np;
        addr = (paddr_t) found*PAGE_SIZE;
    }
    else {
        addr = 0;
    }

    spinlock_release(&freemem_lock);
    return addr;
}
```

freepages

```
static int freepages(paddr_t addr, unsigned long npages) {
    long i, first, np=(long)npages;
    if (!isTableActive()) return 0;
    first = addr/PAGE_SIZE;
    KASSERT(allocSize!=NULL);
    KASSERT(nRamFrames>first);

    spinlock_acquire(&freemem_lock);
    for (i=first; i<first+np; i++) {
        freeRamFrames[i] = (unsigned char)1;
    }
    spinlock_release(&freemem_lock);

    return 1;
}
```

⚠ QUESTE FUNZIONI SONO TUTTE VISIBILI NEL WORD DEI LAB IN LAB2

2) FILE SYSTEM & I/O OS161

Il **FILE SYSTEM** di OS161 ha 2 livelli:

- **Virtual file system** (`kern/vfs/`) = indipendente dal file system usato; qui si possono aggiungere file systems
- **Actual file system** (`kern/fs/`) [la sottocartella `sfs` contiene l'implementazione del “*simple*” file system] [noi però finora abbiamo lavorato su un file system emulato (`emufs`)]

Per quanto riguarda i **dispositivi I/O** (a sx img con devices e loro *Device ID*), è importante imparare come funziona il **LAMEbus** di MIPS: ha **32 slots** (ognuno con address space di dimensione fissa [**64K**; l'intero bus = $64K \times 32 = 2\text{MB}$], mappata su memoria fisica del dispositivo); **no DMA** (direttamente trasferimenti dalla CPU). Il 1° indirizzo fisico (da cui parte l'address space) è il **LAMEBASE**. Il **31° slot** contiene il **bus controller**, il cui spazio di indirizzamento di 64K è diviso in **2 parti**: lower-half con 32 config regions da 1K (**1 per slot**) e upper-half con 32 config regions da 1K (**1 per CPU**).

```
Kern/dev/lamebus/lamebus.h
/* CS161 devices */
#define LBCS161_UPBUSCTL      1
#define LBCS161_TIMER         2
#define LBCS161_DISK          3
#define LBCS161_SERIAL        4
#define LBCS161_SCREEN        5
#define LBCS161_NET           6
#define LBCS161_EMUFS         7
#define LBCS161_TRACE         8
#define LBCS161_RANDOM        9
#define LBCS161_MPBUSCTL     10
```

⚠ Tutte queste informazioni su LAMEbus e FS di OS161/MIPS sono in documentazione online (slides per url)

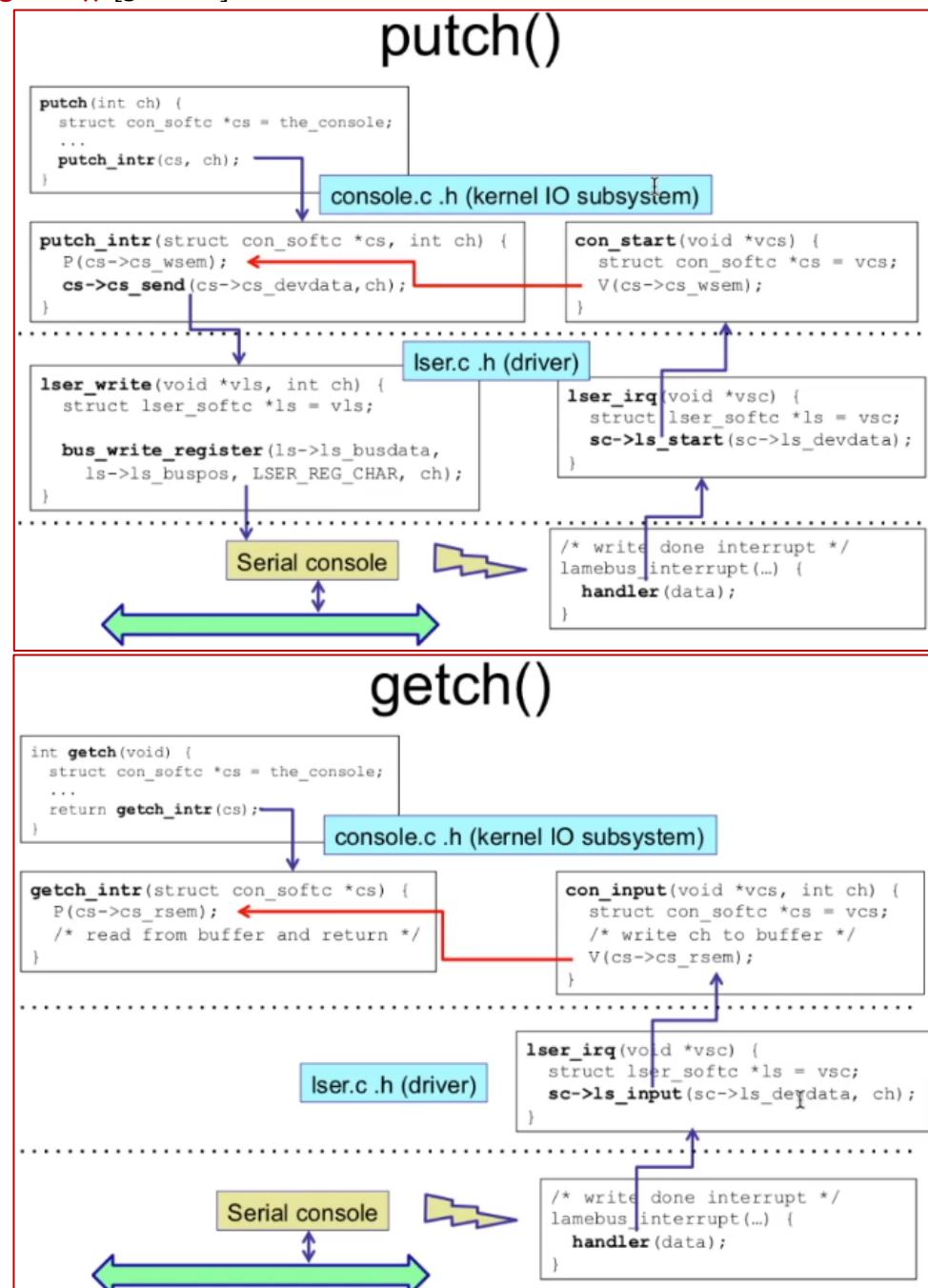
Vediamo alcune informazioni sui **dispositivi di I/O** sopra citati:

- **SERIAL CONSOLE** (DID = 4) → **tastiera + modalità testuale** (semplice interfaccia di console). Bisogna aspettare che una write sia finita prima di iniziare un'altra. **Write** sul 1° registro printa il carattere, **Read** da esso ritorna l'ultimo carattere scritto.

Registers	
Offset	Description
0-3	Character buffer
4-7	Write IRQ register
8-11	Read IRQ register
12-15	Reserved

Informazioni su di essa (costanti e funzioni) si trovano nel file **lser.c** (che sarebbe il **driver** della console seriale) e **lser.h** (qui c'è una struct **lser_softc** con i puntatori per far funzionare questa serial console) nella cartella **kern/dev/lamebus**.

2 operazioni (che abbiamo usato durante il lab per implementare le syscalls **write** e **read**) sono **putch()** [**putchar**] e **getch()** [**getchar**]:



⚠ Mentre **putch** ha **azione diretta lser_write**, la **getch** invece fa tutto in **interrupt**, ovvero appena premo un tasto da tastiera la **console scatena un interrupt** che chiama **lser_irq** che passa il char alla **con_input** (che finito il suo uso libera il semaforo per azionare la **getch_intr**)

3) PRIMITIVE DI SINCRONIZZAZIONE OS161

Partiamo da un ripasso sulla **SINCRONIZZAZIONE**; problemi sono:

- **Race conditions** ("corse critiche") = il risultato di un elaborazione concorrente dipende dall'ordine in cui sono effettuate le singole operazioni
- **Deadlock** ("stallo") = ogni entità aspetta qualcosa da altre entità e nessuno procede
- **Starvation** = un entità rimane sempre in attesa, in quanto la CPU è sempre occupata da altre entità

Definiamo **SEZIONE CRITICA** ("critical section"; SC) una sezione di programma che deve essere eseguita in mutua esclusione (solo 1 thread/processo per volta), ma bisogna comunque garantire un progresso, cioè i **REQUISITI** sono:

- **MUTUA ESCLUSIONE** (P_i e P_j non possono eseguire le SC insieme in modo concorrente)
- **PROGRESSO** (se P_i vuole eseguire la sua SC e nessun altro è all'interno di essa, P_i non aspetta indefinitamente)
- **ATTESA LIMITATA** (se P_i è in wait della sua SC, ci deve essere un limite al n° di ingressi degli altri alla SC)

```
do {
    entry section
    critical section
    exit section
    remainder section
} while (true);
```

Le soluzioni di ciò **dipendono dal kernel e dai parallel cores**:

- Single core con **preemption** (ovvero permette di fermare un processo/thread in kernel mode)
- Single core senza preemption
- Multicore (sempre possibile avere programmi/thread paralleli)

Soluzione di Peterson: abbiamo 2 processi che hanno una SC in comune; abbiamo **load** e **store atomiche**. I 2 processi condividono 2 variabili: **int turn** (indica a chi tocca) e **boolean flag[2]** ($\text{flag}[i] = \text{true}$ se P_i è pronto ad entrare nella SC). I 2 processi avranno questo **codice qui a dx** (vediamo solo P_i , in quanto P_j sarà speculare)

```
while (true){
    flag[i] = true;
    turn = j;
    while (flag[j] && turn == j)
        ;
    /* critical section */
    flag[i] = false;
    /* remainder section */
}
```

⚠ Seppur garantisca i 3 requisiti della SC, non garantisce il funzionamento su architetture moderne (multicore) [questo perché è sempre consentita concorrenza nei multicore, quindi non è detto che P_i escluda P_j]

Soluzione che permette processi concorrenti (cioè architetture moderne) è **LOCK**.

Cosa succede però **se non si riesce ad acquisire il lock?** 2 vie:

- Continuare a provare → **"spin"** o **"busy-wait"**; va bene con attese corte
- Rilasciare il processore (vado in wait) → va bene con attese lunghe

```
do {
    acquire lock
    critical section
    release lock
    remainder section
} while (true);
```

Come implementare il LOCK? C'è bisogno di **supporto hardware** (su unicore posso disabilitare interrupt e devo eseguire senza preemption; su multicore ci sono **istruzioni hardware atomiche**). Abbiamo **3 tipi di hw support**:

- **Memory barriers** = le memorie condivise nei multicore possono essere:
 - **strongly ordered** → modifica di 1 processore subito visibile a tutti gli altri processori
 - **weakly ordered** → modifica di 1 processore non subito visibile a tutti gli altri processori (+ veloce)
 Quindi una **"memory barrier"** (**BARRIERA**) è un'istruzione che forza attesa fino a che le modifiche in memoria non siano state propagate ovunque (limita l'*out-of-order execution*)
- **Istruzioni hardware:**
 - **Test & Set** → eseguita in modo **atomico**: ritorna il valore originale del parametro e setta il nuovo valore del parametro a true. A dx come si usa con la SC (lock = false all'inizio)

```
boolean test_and_set (boolean *target)
{
    boolean rv = *target;
    *target = true;
    return rv;
}
```

```
do {
    while (test_and_set(&lock))
        ; /* do nothing */

    /* critical section */

    lock = false;
    /* remainder section */
} while (true);
```

- **Compare & Swap** → eseguita in modo **atomico**: ritorna il valore originale del parametro e setta *value = new_value, ma solo se $*\text{value} == \text{expected}$ (**only under this condition we swap**)

```
int compare_and_swap(int *value, int expected, int new_value) {
    int temp = *value;

    if (*value == expected)
        *value = new_value;
    return temp;
}
```

```
while (true){
    while (compare_and_swap(&lock, 0, 1) != 0)
        ; /* do nothing */

    /* critical section */

    lock = 0;
    /* remainder section */
}
```

- **Atomic variables** = le funzioni come la **compare_and_swap** sono solitamente usati come blocchi di altri tool di sincronizzazione; 1 di questi tool è una “**variabile atomica**” (garantisce atomicità su read/write su tipi come int e boolean) [es. usare **increment(&sequence)** dobbiamo avere sequence atomica; sotto il codice esempio]

```
void increment(atomic_int *v)
{
    int temp;

    do {
        temp = *v;
    } while (temp != (compare_and_swap(v, temp, temp+1)));
}
```

Tutte queste soluzioni però sono complicate; noi useremo i **MUTEX LOCKS** (variabili booleane che assomigliano ai semafori, che proteggono le SC con **acquire()** lock e **release()** lock [operazioni atomiche che permettono di acquisire il lock se il mutex è libero e poi rilasciarlo]). Questa soluzione però richiede “**busy waiting**” (per questo è detta **SPINLOCK**). Un esempio di pseudoimplementazione è il seguente:

```
acquire() {
    while (!available)
        ; /* busy wait */
    available = false;;
}

release() {
    available = true;
}
while (true) {
    acquire lock
    critical section
    release lock
    remainder section
}
```

Lo **SPINLOCK** (o **BUSY WAIT**) è la 1^ primitiva di sincronizzazione disponibile in OS161 (utile se attesa è corta):

OS/161 Spinlocks

(kern/thread/spinlock.c)

```
spinlock_acquire(struct spinlock *splk) {
    ...
    while (1) {
        /* Do test-test-and-set, that is, read first before
           doing test-and-set, to reduce bus contention.
           Test-and-set is a machine-level atomic operation
        */
        if (spinlock_data_get(&splk->splk_lock) != 0) {
            continue;
        }
        if (spinlock_data_testandset(&splk->splk_lock) != 0) {
            continue;
        }
        break;
    }
    ...
}
```

Però ci sono anche i **SEMAFORI** in OS161:

```
struct semaphore *s;
s = sem_create("MySem1", 1); /* initial value is 1 */

P(s); /* do this before entering critical section */

critical section /* e.g., call to list remove front */

V(s); /* do this after leaving critical section */
```

Un **SEMAFORO** è una variabile intera su cui sono possibili le operazioni di **wait** (cioè **P**) e di **signal** (cioè **V**):

```
wait(S) {
    while (S <= 0)
        ; // busy wait
    S--;
}

signal(S) {
    S++;
}
```

Ci sono **semafori con conteggio** e **semafori binari (mutex)** [già visti ad OS] [i mutex potrebbero sembrare come i mutex lock di prima, ma i semafori permettono la modifica anche non all'owner del semaforo stesso]. Ogni semaforo ha una waiting queue con entry del tipo [**valore, puntatore a next entry**]; ci sono 2 operazioni:

- **block** = mette il processo chiamante nella waiting queue (**in OS161 thread_sleep()** [simile a **thread_yield**] dove il thread si mette volontariamente in attesa del wakeup)

```
typedef struct {
    int value;
    struct process *list;
} semaphore;
```

➤ **wakeup** = toglie 1 dei processi dalla coda e lo mette nella ready queue (in OS161 `thread_wakeup()` sveglia i thread in attesa su un certo semaforo)

⚠ In OS161, dato che siamo su MIPS, funziona il concetto visto prima negli unicore dove basta *disabilitare gli interrupts* per garantire mutua esclusione; quindi in precedenza (**nelle versioni vecchie**) i semafori erano realizzati proprio basandoli sugli **interrupts**!

Qui invece rivediamo un'altra implementazione di **wait** e **signal**, ma **senza busy waiting**:

```
wait(semaphore *S) {
    S->value--;
    if (S->value < 0) {
        add this process to S->list;
        block();
    }
}

signal(semaphore *S) {
    S->value++;
    if (S->value <= 0) {
        remove a process P from S->list;
        wakeup(P);
    }
}
```

⚠ In OS161, abbiamo anche un'implementazione della primitiva di **lock** (come un semaforo binario inizializzato ad 1, **ma** che come abbiamo precedentemente, la release deve essere fatta per forza dall'owner)

```
struct lock *mylock = lock_create("LockName");

lock_acquire(mylock);
critical section /* e.g., call to list remove head */
lock_release(mylock);
```

Ma perché quindi non basta usare solo i semafori e bom? Perché sono **complessi da usare** e potrebbero **facilmente generare deadlock** con un uso errato. Per questo negli anni è stato sviluppato un costrutto di alto livello chiamato **monitor**, ovvero un **ADT** con le sue variabili interne **accessibili solo dal codice della procedura corrente** e che permette **solo 1 processo attivo per volta**).

Vediamo un **ESEMPIO** di come gestire la concorrenza su 3 variabili:

```
/* shared state vars with some initial value */
int x,y,z;
/* mutual exclusion for shared vars */
struct lock *mylock = lock_create("Mutex");
/* semaphore to wait if necessary */
struct semaphore *no_go = sem_create("MySem", 0);

compute_a_thing {
    lock_acquire(mylock); /* lock out others */
    /* compute new x, y, z */
    x = f1(x); y = f2(y); z = f3(z);
    if (x != 0 || (y <= 0 && z <= 0)) V(no_go);
    lock_release(mylock); /* enable others */
}

use_a_thing {
    lock_acquire(mylock); /* lock out others */
    if(x == 0 && (y > 0 || z > 0))
        P(no_go);
    /* Now either x is non-zero or y and z are
       non-positive. In this state, it is safe to run
       "work" on x,y,z, which may also change them */
    work(x,y,z);
    lock_release(mylock); /* enable others */
}
```

When waiting, mylock owned !
Other threads cannot modify x,y,z
and possibly unblock

Per risolvere il problema evidenziato dal banner giallo, dobbiamo fare in modo che, mentre il “**consumer**” **rilasci il lock quando va in attesa; poi attendere e riottenere il lock al wakeup**. C’è però quindi una **finestra temporale in cui non c’è mutua esclusione quando il consumer esce dall’attesa** (al wakeup non sappiamo se esattamente nello stesso istante altri processi escono dall’attesa, quindi per questo viene inserito il **while**):

```
use_a_thing {
    lock_acquire(mylock); /* lock out others */
    while (x == 0 && (y > 0 || z > 0)) {
        lock_release(mylock); /* no deadlock */
        P(no_go);
        lock_acquire(mylock); /* lock for next test */
    }
    /* Now either x is non-zero or y and z are
       non-positive. In this state, it is safe to run
       "work" on x,y,z, which may also change them */
    work(x,y,z);
    lock_release(mylock); /* enable others */
}
```

Ci sono poi le **Condition Variables** (cv): ricevono il lock come parametro e funzionano solo **in sinergia con un lock**, ma nascondono il rilascio e la ripresa esplicita del lock (all’interno sono delle **code di attesa**); se abbiamo una **condition x** (ovvero una cv), questa permette operazioni **atomiche** come:

- `cv_wait(struct cv *cv, struct lock *lock)` → **rilascia** il lock, **aspetta**, riacquisisce il lock prima del return
- `cv_signal(struct cv *cv, struct lock *lock)` → **sveglia 1** dei thread incodati (se non ce ne sono, ignorata)
- `cv_broadcast(struct cv *cv, struct lock *lock)` → **sveglia tutti** i thread incodati (///, ignorata)

⚠ Il **lock** come 2° parametro non è necessario nella **signal** e nel **broadcast**, ma viene messo in quanto entrambe devono essere **chiamate solo quando si è in mutua esclusione su questo lock** (quindi viene fatto solo un controllo su di esso per la mutua esclusione)

⚠ Le CV permettono di implementare i **monitor** (in OS161 però dovremo usare le CV esplicitamente senza monitor, in quanto questi sono implementati solo in linguaggi di alto livello e noi invece useremo il C in OS161)

Con le condition variables abbiamo 2 scelte (ricorda che P_1 e P_2 non possono andare insieme):

1. **signal & wait** → processo P **svegliato** subito e **parte** (quindi è già pronto per partire)
2. **signal & continue** → processo P **svegliato ma deve aspettare** per la sua esecuzione

Vediamo un **ESEMPIO** di schema di sincronizzazione che usa le CV in OS161 (anche qui al wakeup non sappiamo se esattamente nello stesso istante altri processi escono dall'attesa, quindi per questo viene inserito il **while**):

P_i	P_j
<pre>lock_acquire(lock); while (condition not true) { cv_wait(cond, lock); } ... // do stuff lock_release(lock);</pre>	<pre>lock_acquire(lock); ... // modify condition cv_signal(cond); lock_release(lock);</pre>

Rivediamo l'**ESEMPIO** di come gestire la concorrenza su 3 variabili, ma questa volta usando CV:

```
/* shared state vars with some initial value */
int x,y,z;
/* mutual exclusion for shared vars */
struct lock *mylock = lock_create("Mutex");
/* condition variable to wait if necessary */
struct cv *no_go = cv_create("CondV");

compute_a_thing {
    lock_acquire(mylock); /* lock out others */
    /* compute new x, y, z */
    x = f1(x); y = f2(y); z = f3(z);
    if (x != 0 || (y <= 0 && z <= 0)) cv_signal(no_go); }
    lock_release(mylock); /* enable others */
}

use_a_thing {
    lock_acquire(mylock); /* lock out others */
    while (x == 0 && (y > 0 || z > 0)) {
        cv_wait(no_go, mylock);
    }
    /* Now either x is non-zero or y and z are
       non-positive. In this state, it is safe to run
       "work" on x,y,z, which may also change them */
    work(x,y,z);
    lock_release(mylock); /* enable others */
}
```

In OS161, oltre agli spinlock (ovvero alle CV) c'è una 2^ primitiva di sincronizzazione detta **WAIT CHANNEL** (**mini condition variables usabili solo a livello di kernel**; il lock che usano però è uno **spinlock con busy wait**) [in OS161 non si possono annidare troppi spinlock, abbiamo errore]. Anche qui abbiamo le solite funzioni:

```
wchan_sleep(struct wchan *wc, struct spinlock *lk) {
    /* may not sleep in an interrupt handler */
    KASSERT(!curthread->t_in_interrupt);

    /* must hold the spinlock */
    KASSERT(spinlock_do_i_hold(lk));

    /* must not hold other spinlocks */
    KASSERT(curcpu->c_spinlocks == 1);

    thread_switch(S_SLEEP, wc, lk);
    spinlock_acquire(lk);
}

wchan_wakeone(struct wchan *wc, struct spinlock *lk) {
    struct thread *target;
    KASSERT(spinlock_do_i_hold(lk));

    /* Grab a thread from the channel */
    target = threadlist_remhead(&wc->wc_threads);

    /* Note that thread_make_runnable acquires a runqueue
       lock while we're holding LK. This is ok; all
       spinlocks associated with wchans must come before the
       runqueue locks, as we also bridge from the wchan lock
       to the runqueue lock in thread_switch. */
    thread_make_runnable(target, false);
}
```

Inoltre, come abbiamo già accennato, in OS161 sono implementati anche i **SEMAFORI**, con le loro funzioni **P e V**:

```
struct semaphore {
    char *name;
    struct wchan *sem_wchan;
    struct spinlock sem_lock;
    volatile int count;
};
```

```

void P(struct semaphore *sem) {
    /* May not block in an interrupt handler. For robustness,
     always check, even if we can actually complete the
     without blocking. */
    KASSERT(curthread->t_in_interrupt == false);

    /* Use the semaphore spinlock to protect the wchan as well */
    spinlock_acquire(&sem->sem_lock);
    while (sem->sem_count == 0) {
        /* Note that we don't maintain strict FIFO ordering of
         threads going through the semaphore; */
        wchan_sleep(sem->sem_wchan, &sem->sem_lock);
    }
    KASSERT(sem->sem_count > 0);
    sem->sem_count--;
    spinlock_release(&sem->sem_lock);
}

```

```

void V(struct semaphore *sem) {
    KASSERT(sem != NULL);

    spinlock_acquire(&sem->sem_lock);

    sem->sem_count++;
    KASSERT(sem->sem_count > 0);
    wchan_wakeone(sem->sem_wchan, &sem->sem_lock);

    spinlock_release(&sem->sem_lock);
}

```

⚠ Cosa dobbiamo implementare quindi in OS161 nei laib? **Locks e Condition Variables**

4) FILE (open/close)

Questa parte termina ciò che abbiamo trattato nella parte **2) File System e I/O**. Nella libreria **unistd.h** (UNIX):

```

int open(const char *filename, int flags, ...);
ssize_t read(int filehandle, void *buf, size_t size);
ssize_t write(int filehandle, const void *buf, size_t size);
int close(int filehandle);
off_t lseek(int filehandle, off_t pos, int code);
int dup2(int filehandle, int newhandle);

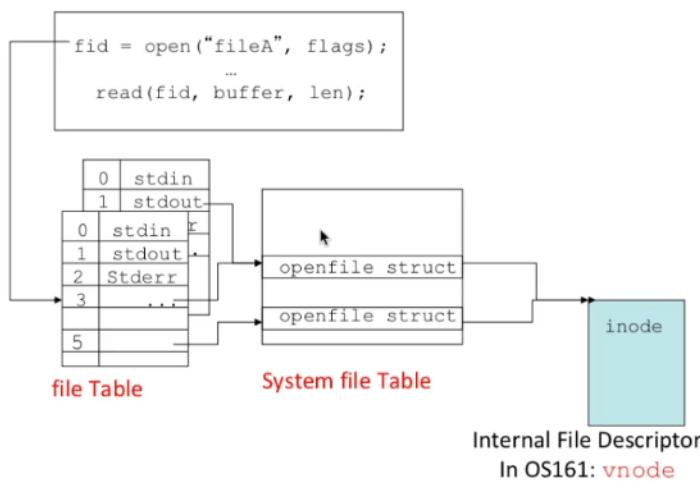
```

Abbiamo trattato bene **in precedenza come implementare la `read` e la `write` in OS161**; ora dobbiamo farlo per **open e close**. Ci soffermiamo sui **metadati** (non il contenuto) dei file. I **processi figli** (nati dalla fork) condividono i file con il padre (hanno anche lo stesso offset/posizione nel file aperto) [questo capitolo tratta il LAB5]

Come si rappresentano i file aperti per ogni processo? Soluzione proposta è usare una **struct openfile** per ogni file aperto + un vettore di oggetti ***openfile** (ovvero 1 **fileTable** per processo) riferito al File Descriptor (FD). La struct **openfile** conterrà:

- **file pointer** (un puntatore al **vnode** [che sarebbe l'inode (internal FD) in OS161] ottenuto da `vfs_open`)
- **mode** (read-only, write-only, read-write...)
- **offset**
- **lock**
- **reference count**

In questo modo un file può essere **aperto più volte** e quindi **avere più openfile structs**. Inoltre, una struct **openfile** può essere **condivisa da thread concorrenti** ma **serves un lock per farlo**:



Dove va salvata la `fileTable`? Si può **salvare nella struct del processo** (**proc**; vista in precedenza); questa sarà definita come una **struct opaca** (non accedo ai campi da `proc`, ma posso fare riferimento ad essa con puntatore).

Dove va salvata la `SystemFileTable`? Il professore ha riusato il **file_syscalls.c** per definire sia la struct **openfile** sia questo vettore di **openfile** (la **SystemFileTable**).

La **sys_open**(filename, flag, retfd) dovrà fare le seguenti cose:

- aprire un file (creando un oggetto openfile)
- ottenere vnode dalla vfs_open
- inizializzare l'offset nella openfile
- mettere la openfile in systemFileTable
- ritornare il FD della openfile

```
int
sys_open(userptr_t path, int openflags, mode_t mode, int *errp)
{
    int fd, i;
    struct vnode *v;
    struct openfile *of=NULL;
    int result;

    result = vfs_open((char *)path, openflags, mode, &v);
    if (result) {
        *errp = ENOENT;
        return -1;
    }
    /* search system open file table */
    for (i=0; i<SYSTEM_OPEN_MAX; i++) {
        if (systemFileTable[i].vn==NULL) {
            of = &systemFileTable[i];
            of->vn = v;
            of->offset = 0; // TODO: handle offset with append
            of->countRef = 1;
            break;
        }
    }
    if (of==NULL) {
        // no free slot in system open file table
        *errp = ENFILE;
    }
    else {
        for (fd=STDERR_FILENO+1; fd<OPEN_MAX; fd++) {
            if (curproc->fileTable[fd] == NULL) {
                curproc->fileTable[fd] = of;
                return fd;
            }
        }
        // no free slot in process open file table
        *errp = EMFILE;
    }
    vfs_close(v);
    return -1;
}
```

La int **sys_close**(fd) dovrà fare le seguenti cose:

- usare fd per trovare la openfile nella fileTable
- cancellare la openfile se ultima apertura (se supportiamo aperture multiple)

```
int
sys_close(int fd)
{
    struct openfile *of=NULL;
    struct vnode *vn;

    if (fd<0 || fd>OPEN_MAX) return -1;
    of = curproc->fileTable[fd];
    if (of==NULL) return -1;
    curproc->fileTable[fd] = NULL;
    if (--of->countRef > 0) return 0; // just decrement ref cnt
    vn = of->vn;
    of->vn = NULL;
    if (vn==NULL) return -1;

    vfs_close(vn);
    return 0;
}
```

La int **sys_read** dovrà fare le seguenti cose:

- usare **fd** per trovare la **openfile** nella **fileTable**
- accedere a **offset** della **openfile**
- settare un **userio record** (**uios**)
- chiamare la **VOP_READ(openfile->vnode, userio)**
- settare **openfile->offset = userio.offset;**
- settare ***retval = quantità letta**

```
static int
file_read(int fd, userptr_t buf_ptr, size_t size) {
    struct iovec iov;
    struct uio ku;
    int result, nread;
    struct vnode *vn;
    struct openfile *of;
    void *kbuf;

    if (fd<0 || fd>OPEN_MAX) return -1;
    of = curproc->fileTable[fd];
    if (of==NULL) return -1;
    vn = of->vn;
    if (vn==NULL) return -1;

    kbuf = kmalloc(size);
    uio_kinit(&iov, &ku, kbuf, size, of->offset, UIO_READ);
    result = VOP_READ(vn, &ku);
    if (result) {
        return result;
    }
    of->offset = ku.uio_offset;
    nread = size - ku.uio_resid;
    copyout(kbuf, buf_ptr, nread);
    kfree(kbuf);
    return (nread);
}
```

Analogamente la **sys_write** (entrambe queste implementazioni sono con il buffer):

```
static int
file_write(int fd, userptr_t buf_ptr, size_t size) {
    struct iovec iov;
    struct uio ku;
    int result, nwrite;
    struct vnode *vn;
    struct openfile *of;
    void *kbuf;

    if (fd<0 || fd>OPEN_MAX) return -1;
    of = curproc->fileTable[fd];
    if (of==NULL) return -1;
    vn = of->vn;
    if (vn==NULL) return -1;

    kbuf = kmalloc(size);
    copyin(buf_ptr, kbuf, size);
    uio_kinit(&iov, &ku, kbuf, size, of->offset, UIO_WRITE);
    result = VOP_WRITE(vn, &ku);
    if (result) {
        return result;
    }
    kfree(kbuf);
    of->offset = ku.uio_offset;      ↴
    nwrite = size - ku.uio_resid;
    return (nwrite);
}
```

FINE