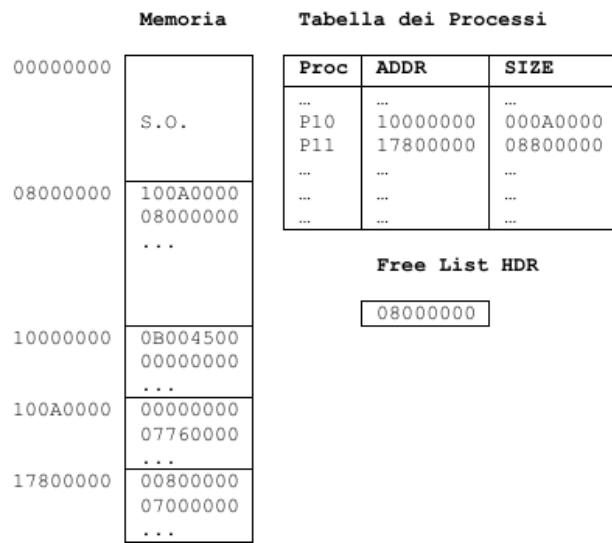


# MEMORIE 1 (esercizi)

## ESERCIZIO 1

SISTEMA con memoria di **512MByte**, in cui si utilizza uno schema di gestione a **partizioni (contigue) variabili** con unità **minima di allocazione** della memoria di **64 Byte** (ovvero lo spazio di memoria viene allocato in multipli di 64 byte). All'OS sono allocati in modo permanente i primi **128MByte** di memoria.

La **tabella dei processi** contiene, per ogni processo attivo, l'**indirizzo iniziale (ADDR)** e la **dimensione (SIZE)** della relativa partizione in memoria. La memoria viene allocata con strategia **Worst-Fit**. Le **partizioni libere** sono gestite mediante una lista linkata ordinata per dimensione decrescente, in cui ogni nodo rappresenta una partizione libera; i nodi della lista sono costituiti da due campi: puntatore alla **next** partizione libera, **size** della partizione → entrambi rappresentati su **4 byte** [dimensioni e indirizzi rappresentati in **Byte**] con **valore 0 = puntatore nullo** e sono memorizzati all'inizio della partizione che rappresentano. Situazione iniziale:



Si rappresentino le modifiche alle partizioni in memoria, alla tabella dei processi e alla Free List, in seguito all'attivazione di 2 nuovi processi, **P12** (25MB di memoria) e **P13** (150MB), con dopo la terminazione del processo **P11**

### Soluzione:

Cerco lo spazio più grande possibile (**Worst-Fit**) elencando gli spazi vuoti e prendo il più grosso ogni volta; **ELENCO TUTTE LE PARTIZIONI** nel formato (base, limit):

- $S_1 = (00000000, 08000000) = \text{OS}$
- $S_2 = (08000000, 08000000) = \text{LIBERA}$
- $S_3 = (10000000, 000A0000) = \text{P10}$
- $S_4 = (100A0000, X) = \text{LIBERA} \rightarrow X = 394264576 - 269090816 = 125173760 \rightarrow X = 07760000$
- $S_5 = (17800000, 07000000) = \text{P11}$

Quindi ho solo  $S_2$  e  $S_4$  LIBERE con **in ordine decrescente  $S_2$  e  $S_4$**

Quindi provo a mettere P12 in  $S_2$  (il più grande, WORST-FIT):

⚠ ATTENZIONE: **25MB** in esadecimale si scrivono come  $25 * 0x100000 \rightarrow 25$  in esadecimale è 19, quindi **0x01900000**; ANALOGAMENTE **150MB** in esadecimale si scrivono come  $150 * 0x100000 \rightarrow 150$  in esadecimale è 96, quindi **0x09600000** → **FARE I CALCOLI SEMPRE IN EXADECIMAL**

if ( $\text{size}_{S_2} = 08000000 > \text{size}_{P_2} = 25\text{MB} = 01900000$ ) POSSO ALLOCARE IN  $S_2$

Spazio libero rimasto (**size nuova di  $S_2$** ) =  $08000000 - 01900000 = 06700000$

**Base address nuovo di  $S_2$**  =  $08000000 + 01900000 = 09900000$

**Base address di P12** =  $08000000$ ; **Size di P12** =  $01900000$

→ **P12 = (08000000, 01900000);  $S_2$  = (09900000, 06700000)**

Riverifco il più grande e riordino le partizioni libere:  $size_{S_4} = 07760000$ ,  $size_{S_2} = 06700000$ , quindi  $S_4 > S_2$

Quindi per il processo P13 devo vedere se sta dentro  $S_4$ :

$size_{S_4} = 07760000 < size_{P_{13}} = 09600000 \rightarrow \text{NON POSSO ALLOCARE} \rightarrow \text{DEVO FARE SWAP OUT DI 1 PROCESSO SUBITO COLLEGATO A QUESTO SPAZIO} \rightarrow \text{SWAP OUT DI P10}$  (scelgo questo perché più semplice e crea una partizione contigua grossa):

$S_{dopoSwapOut} \rightarrow S_{unione(2,3,4)} = (09900000, [06700000+000A0000+07760000]) = (\textbf{09900000, ODF00000})$

$size_{S_{unione}} = 0DF00000 > size_{P_{13}} = 09600000 \rightarrow \text{POSSO ALLOCARE:}$

**Spazio libero rimasto (size nuova di  $S_{unione}$ )** =  $0DF00000 - 09600000 = 04900000$

**Base address nuovo di  $S_{unione}$**  =  $09900000 + 09600000 = 12F00000$

**Base address di P13** =  $09900000$ ; **Size di P13** =  $09600000$

$\rightarrow \text{P13} = (\textbf{09900000, 09600000}); S_{unione} = (\textbf{12F00000, 04900000})$

## ESERCIZIO 2

Sia dato un sistema di **memoria virtuale con paginazione** (indirizzo su Byte). Il sistema dispone di **TLB**, su cui si ha: **hit ratio = 99%**. La **page-table** viene realizzata con uno schema a **2 livelli**: indirizzo logico di 32 bit viene suddiviso (da MSB a LSB) in **3 parti**: **p1** = 10bit, **p2** = 11bit, **d** = 11bit [non si utilizzano ulteriori strutture dati (quali tabelle di hash o inverted page table) per velocizzare gli accessi].

- Cosa si intende per **HIT RATIO**: probabilità che la pagina cercata sia nel TLB (e non vada recuperata in memoria [miss])
- Si illustri lo **schema della page-table** e la sua **dimensione complessiva**, per un processo **P1** avente spazio di indirizzamento virtuale di **100 MByte**:

Prendo la potenza di 2 più piccola  $> 100 \rightarrow 128\text{MB} = 2^{27} \text{ Byte}$

**Dimensione pagine e frame** =  $2^d = 2^{11} \text{ Byte} = 2\text{KB}$  perché **d** = **offset** = 11 bit

Ipotizzando **entry della page table** = 32bit = 4Byte, avremo:

- **Tabella pagine 2° livello** →  $2^{p_2} = 2^{11}$  celle con dimensione quindi di  $2^{11} * 4 \text{ Byte}$
- **Tabella pagine 1° livello** →  $2^{32-27} = 2^5$  quindi **bastano solo 5 dei 10 bit** di **p1**

- Si calcolino **frammentazione esterna e interna** per il processo P1 (vedi punto precedente):
  - **Frammentazione esterna** → è 0 per definizione (in quanto è esterna al processo, quindi nel processo è 0)
  - **Frammentazione interna** → **spazio non usato dentro l'ultima pagina** assegnata a un processo; quindi, dato che le **pagine hanno una dimensione fissa di  $2^{11}$  Byte = 2KB**, dipende dallo spazio richiesto ed il **caso peggiore è 2KB** (ovvero spazio limite di 1 pagina) [es. **processo richiede 5.5KB di memoria virtuale** → **2KB + 2KB + 1.5KB** ovvero 3 pagine di cui però l'ultima ha 0.5KB inusati]

Nel nostro caso, essendo che il processo P1 ha **100 MB =  $100 * 2^{20}$  Byte**, e le pagine sono da **2 KB =  $2^{11}$  Byte**: 100 MB è **multiplo esatto di 2 KB** (perché  $100 * 2^{20} / 2^{11} = 100 * 2^9 = 51200$  pagine). **Nessuna pagina è parzialmente piena**, quindi **frammentazione interna = 0**

- Supponendo che la memoria RAM abbia **tempo di accesso** di 300 ns, si calcoli il tempo effettivo di accesso (**EAT**) per il caso proposto (**hit ratio = 99 %**):

**EAT = (hit ratio \* tempo accesso + miss ratio \* n° accessi miss \* tempo accesso)** → essendo che la page table è a 2 livelli, avrà **2 accessi per la page table (1° livello e 2° livello) + 1 accesso in RAM (miss)**, ovvero:  
 $EAT = (0.99 * 300\text{ns} + 0.01 * (2+1) * 300\text{ns}) = (0.99 * 300\text{ns} + 0.01 * 3 * 300\text{ns}) = \textbf{306ns}$

## ESERCIZIO 3

Si descrivano **vantaggi** e **svantaggi** di una **inverted page table (IPT)**, contro una **page table standard** [1 tabella per processo].

Ho processo con virtual address space = 32 GB, dotato di 8GB di RAM, su una architettura a 64 bit (in cui si indirizza a Byte), con gestione della memoria paginata (pagine/frame da 1KB). Calcola la **size della page table (ad 1 solo livello) per il processo e della IPT** [ipotizzando **PID** processo su 16bit = 2B]. Si utilizzino 32 bit = 4B per **page/frame number** (indice di pagina/frame).

Si dica infine, usando la IPT proposta (32 bit per page/frame number), qual è la **max size possibile di virtual address space di un processo**.

### Soluzione:

#### 1. VANTAGGI:

- Dimensione** della tabella è quella della RAM fisica invece che quella del virtual address space
- 1 tabella per tutti i processi** (ogni frame associato ad 1 solo processo)

#### SVANTAGGI:

- Ricerca lineare** è lenta e costosa (devo cercare la pagina anziché accedervi direttamente) [risolvibile con tabelle di hash dove le entries sono poste in liste di adiacenza]

#### 2. Per i calcoli delle dimensioni (trascuro eventuali bit di validità/modifica):

##### **Page Table standard**

N pagine = address space/size pagina =  $32\text{GB}/1\text{KB} = 32\text{M} = \text{n}^{\circ}$  page numbers nella page table (entries)

Ogni page number (indice di pagina) è di 32 bit  $\rightarrow |\text{Page Table}| = \text{size page table} = 32\text{M}*4\text{B} = 128\text{MB}$

##### **IPT**

Tabella unica per i processi con 1 page number per ogni frame in RAM  $\rightarrow \text{n}^{\circ}$  pages (frames) = RAM/size pagina =  $8\text{GB}/1\text{KB} = 8\text{M}$

$|\text{IPT}| = \text{size IPT} = \text{n}^{\circ}$  pages \* (page number size + PID size) =  $8\text{M}*(4\text{B}+2\text{B}) = 48\text{MB}$

#### 3. **Max size di virtual address space** di processo:

n° max virtual pages di un processo =  $2^{\text{bit di page number}} = 2^{32} = 4\text{G}$

max size virtual address space = n° max virtual pages \* size pagina =  $4\text{G} * 1\text{KB} = 4\text{TB}$

# MEMORIE 2 (esercizi)

## ESERCIZIO 1

Si consideri la **sequenza di riferimenti in memoria** nel caso di un programma di 1000 parole: 261, 409, 985, 311, 584, 746, 632, 323, 470, 915, 858.

Trovare la **stringa dei riferimenti a pagine**, supponendo che la loro **dimensione sia di 200 parole**. Si usa un algoritmo di page replacement **second-chance set** (con limite di 3 frame disponibili) e si suppone che i riferimenti inizino a programma già avviato con i **3 frame già allocati alle pagine 4, 3 e 1**, e la **coda FIFO contenente** (in ordine) **4<sub>0</sub>, 3<sub>1</sub>, 1<sub>0</sub>** (dove pedice = reference bit).

Trovare **quali e quanti page fault** (ovvero accessi a pagine non dentro il resident set) [è richiesta la visualizzazione del resident set dopo ogni riferimento]. Si supponga che il bit di riferimento di una pagina venga inizializzato a 0 in corrispondenza di un page-fault.

### Soluzione:

Per trovare la stringa dei riferimenti a pagine si prende la sequenza di riferimenti in memoria e la si divide per la dimensione dei riferimenti a pagine (in questo caso 200):

$$Str = \frac{261}{200} = 1; \frac{409}{200} = 2; \frac{985}{200} = 4; \frac{311}{200} = 1; \frac{584}{200} = 2; \frac{746}{200} = 3; \frac{632}{200} = 3; \frac{323}{200} = 1; \frac{470}{200} = 2; \frac{915}{200} = 4; \frac{858}{200} = 4$$

Quindi la stringa di riferimento è = 1 2 4 1 2 3 3 1 2 4 4

Perciò, dato che nel testo ci viene detto che abbiamo **3 frame già allocati alle pagine 4, 3 e 1**, e la **coda FIFO contenente** (in ordine) **4<sub>0</sub>, 3<sub>1</sub>, 1<sub>0</sub>** (dove pedice = reference bit), allora avremo questa situazione:

	1	2	4	1	2	3	3	1	2	4	4
4 <sub>0</sub>	4 <sub>0</sub>	2 <sub>0</sub>	4 <sub>0</sub>	4 <sub>0</sub>	4 <sub>0</sub>	3 <sub>0</sub>	3 <sub>1</sub>	3 <sub>1</sub>	3 <sub>1</sub>	3 <sub>0</sub>	3 <sub>0</sub>
3 <sub>1</sub>	3 <sub>1</sub>	3 <sub>1</sub>	3 <sub>0</sub>	3 <sub>0</sub>	2 <sub>1</sub>	4 <sub>0</sub>	4 <sub>1</sub>				
1 <sub>0</sub>	1 <sub>1</sub>	1 <sub>1</sub>	1 <sub>0</sub>	1 <sub>1</sub>	1 <sub>1</sub>	1 <sub>0</sub>	1 <sub>0</sub>	1 <sub>1</sub>	1 <sub>1</sub>	1 <sub>0</sub>	1 <sub>0</sub>
	PF	PF		PF	PF				PF		

Ottenendo quindi **5 PF** (Page Faults).

## ESERCIZIO 2.a

Si consideri la **sequenza di riferimenti in memoria** nel caso in cui per ogni accesso (indirizzi in esadecimale [si indirizza il Byte]) si indica se si tratta di read (R) o write (W): R 3F5, R 364, W 4D3, W 47E, R 4C8, W 2D1, R 465, W 2A0, R 3BA, W 4E6, R 480, R 294, R 0B8, R 14E.

Supponiamo: **indirizzi fisici e logici su 12 bit; paginazione con pagine di 128 Byte; max indirizzo usabile C10**. Trovare **quante pagine** sono nello spazio di indirizzamento del programma e calcola **frammentazione interna**.

### Soluzione:

Nº tot pagine indirizzabili (incluse quelle fuori dai riferimenti proposti) = C10 = 1100 0 **001 0000**

Max indice pagina =  $2^C$  (esadecimale) =  $2^{12}$  (decimale) = 24 → quindi tot spazio indirizzabile ha 25 pagine.

[Metodo alternativo:  $C10_{hex} + 1 = 3089_{dec} \rightarrow NP = \frac{3089}{128} = 25$ ]

Invece la frammentazione interna: ultima pagina è occupata fino all'offset **0010000** (16) [guarda sopra C10] → quindi **FRAMMENTAZIONE INTERNA = 128-17 = 111 Byte**

[Metodo alternativo:  $128 - 3089\%128 = 128-17 = 111$  Byte]

## ESERCIZIO 2.b

Si determini la **stringa dei riferimenti a pagine** [si consiglia di passare da hex a binario per trovare il n° di pagina e (se necessario) il displacement/offset]. Si usa un algoritmo di page replacement **LRU**. Si assuma che siano disponibili **3 frame** agli indirizzi fisici (espressi in hex) **780, A00, B00**

[è richiesta la visualizzazione del resident set dopo ogni riferimento (i frame fisici contenenti pagine logiche)]

Determinare **quali e quanti page fault** (accessi a pagine non presenti nel resident set) si verificheranno. Si dica poi a quali indirizzi fisici vengono effettuati gli accessi **R 3F5, W 4D3, R 3BA** (indicati nell'esercizio 2.a).

### Soluzione:

Indirizzo logico ha **p,d** [pagina,displacement] dove una pagina contiene 128 Byte: per **d** servono 7 bit, per **p** 5 bit. Mettiamo i riferimenti in binario: **R (0011 1111 0101), W (0100 1101 0011), R (0011 0110 0011)**.

**Stringa dei riferimenti** [basta p non serve d]: 7 (perché  $2^3+1$ ), 6 (perché  $2^3+0$ ), 9, 8, 9, 5, 8, 5, 7, 9, 9, 5, 1, 2 [**p** è dato dai 5 bit più significativi dell'indirizzo logico e si calcola come il doppio della 1<sup>ª</sup> cifra esadecimale + il bit più significativo della 2<sup>ª</sup> cifra esadecimale].

Riferimenti		7	6	9	8	9	5	8	5	7	9	9	5	1	2
Resident Set	780	7	7	7	8		8			8	9			9	2
	A00		6	6	6		5			5	5			5	5
	B00			9	9		9			7	7			1	1
Page Fault		*	*	*	*		*			*	*			*	*

Indicare nella prima riga la stringa dei riferimenti a pagine (rappresentate a scelta in esadecimale o decimale), nelle tre successive (che rappresentano i 3 frame del resident set), le pagine allocate nei corrispondenti frame. La free frame list sia inizialmente (780, A00, B00). Nell'ultima riga si indichi la presenza o meno di un Page Fault

Numero totale di page fault: 9

Indirizzi logici: R 3F5: (0011 1111 0101) W 4D3: (0100 1101 0011) R 3BA: (0011 1011 1010)  
 Indirizzi fisici: R (0111 1110 0101) = 7F5, W (1011 0101 0011) = B53 → R (1011 0111 1010) = B3A

Attenzione: La pagina logica 7 viene posta in due frame diversi in due momenti diversi

## ESERCIZIO 3

Si consideri la **sequenza di riferimenti in memoria** nel caso di un programma di 4K parole in cui, per ogni accesso (indirizzi in hex), si indica se si tratta di read (**R**) o write (**W**): **W 3A1, R 3F5, R A64, W BD3, W 57E, R A08, R B85, W 3A0, R A1A, W A36, R B20, R 734, R AB8, R C4E, W B64**.

Si determini la **stringa dei riferimenti a pagine**, supponendo che la loro dimensione sia di 512 parole. Si usa un algoritmo di page replacement **Enhanced Second-Chance** [al reference bit (inizializzato a 0 in corrispondenza del 1º accesso a una nuova pagina dopo il relativo page fault), si unisce il modify bit].

Si assume che: pagina venga sempre modificata in corrispondenza di una write; siano disponibili 3 frame; l'algoritmo operi così: dato il puntatore alla pagina corrente (strategia FIFO) si fa un **1º giro** (senza modificare il reference bit) sulle pagine **per localizzare la vittima** [l'ordine di priorità è (reference,modify): (0,0), (0,1), (1,0), (1,1) [già visto a teoria]]; trovata la vittima, si fa un **2º giro per azzerare i reference bit delle pagine "salvate"** (comprese tra la posizione di partenza e la vittima)].

Determinare **quali e quanti page fault** (accessi a pagine non presenti nel resident set) [è richiesta la visualizzazione del resident set dopo ogni riferimento, indicando per ogni frame i bit di riferimento e modifica]. Pagine numerate a partire da 0.

Utilizzare lo schema seguente per svolgere l'esercizio: indica nella **1^ riga** la stringa dei riferimenti a pagine (rappresentate in hex o dec); nella **2^ R o W**; nelle **3 successive** (che rappresentano i 3 frame del resident set), le pagine allocate nei corrispondenti frame, indicando per ognuna i reference e modify bit.

Indicare (evidenziandola) quale pagina si trova in testa al FIFO; nell'ultima riga si indichi la presenza o meno di un **Page Fault**.

⚠ Ogni cifra esadecimale rappresenta 4 bit; 2 cifre sono 8 bit → quindi la divisione per 512 ( $2^9$ ) si ha eliminando le 2 cifre hex meno significative e dividendo ulteriormente per 2 [in pratica si divide per 2 la prima delle 3 cifre esadecimali]

Riferimenti	1	1	5	5	2	5	5	1	5	5	5	3	5	6	5
Read/write	W	R	R	W	W	R	R	W	R	W	R	R	R	R	W
Resident Set	1 <sub>01</sub>	1 <sub>11</sub>	1 <sub>01</sub>	1 <sub>01</sub>	1 <sub>01</sub>	1 <sub>01</sub>									
			5 <sub>00</sub>	5 <sub>11</sub>	5 <sub>01</sub>	5 <sub>11</sub>	5 <sub>01</sub>	5 <sub>11</sub>							
				2 <sub>01</sub>	3 <sub>00</sub>	3 <sub>00</sub>	6 <sub>00</sub>	6 <sub>00</sub>							
Page Fault	X		X		X							X		X	

N° page fault = 5

# MEMORIE 3 (esercizi)

## ESERCIZIO 1

Si consideri la **sequenza di riferimenti in memoria**: 577517111434123. Si utilizzi un algoritmo di page replacement **working-set (versione esatta)** con finestra  $\Delta = 3$ , assumendo che siano **disponibili max 3 frame**. Determinare **quali e quanti page fault** (accessi a pagine non nel resident set) e **page out** (rimozioni di pagine dal resident set). Si richiede la visualizzazione (dopo ogni accesso) del resident set [è richiesta la visualizzazione del resident set dopo ogni accesso].

Si vuole poi definire una misura di località del programma svolto, basato sulla “*Reuse Distance*” → la **Reuse Distance al tempo  $t_i$  in cui si accede alla pagina  $P_i$**  è il **n° di pagine (diverse da  $P_i$ ) a cui si è fatto accesso a partire dal precedente accesso a  $P_i$**  [se sono al 1° accesso a una pagina, è il numero totale di pagine cui si è fatto accesso fino a  $t_i$ ] [es. a  $t = 3$  faccio il 2° accesso a  $P = 5$ , avendo quindi RD3 = 1 (in quanto tra i due accessi alla pagina 5 si è fatto accesso (2 volte) alla sola pagina 7)].

Dati i vari  $RD_i$ , se ne calcoli il valor medio  $RD_{avg}$ ; inoltre la **località del programma** svolto viene definita come  $L = 1 / (1 + RD_{avg})$ . Si calcolino  $RD_i$ ,  $RD_{avg}$  e  $L$ .

Utilizzare lo schema seguente per svolgere l'esercizio (sono già indicati i riferimenti e la RD ai tempi 0 e 3) [essendo disponibili max 3 frame, la finestra  $\Delta = 3$  comprende l'accesso corrente al tempo  $t_i$ ; la scelta del frame nel resident set è arbitraria (sono quindi possibili altre soluzioni, con frame permutati)].

Riferimenti	5	7	7	5	1	7	1	1	1	4	3	4	1	2	3
Resident Set	5	5	5	5	5	5				4	4	4	4	4	3
	7	7	7	7	7	7	7	7		3	3	3	3	2	2
					1	1	1	1	1	1	1	1	1	1	1
	x	x			x					x	x		x	x	x
Page Fault							x		x			x		x	x
Page Out															
RD	0	1	0	1	2	2	1	0	0	3	4	1	2	5	3

Quindi:

n° tot **page fault** = 8;

n° tot **page out** = 5;

$RD_{avg} = 25/15 = 1.67 \rightarrow L = 1/(1+1.67) = 0.375$

## ESERCIZIO 2

Sia data la **stringa di riferimenti a pagine** 3, 4, 1, (3, 1, 4, 4, 3, 1, 1)\*10 [\*10 indica che la stringa tra parentesi viene ripetuta/iterata 10 volte]. Si usa algoritmo di page replacement **working-set (versione esatta)** con finestra  $\Delta = 3$ . Si visualizzino nello schema che segue i riferimenti e il resident set dopo ogni riferimento, indicando i **page-fault** e i **page-out**. Si richiede la visualizzazione solo fino alle prime 2 iterazioni della sotto-stringa ripetuta 10 volte.

⚠ Ogni riga del resident set è un frame, quindi una pagina in un frame non può cambiare riga quando rimane nel resident set. 3 casi:

- page-fault senza page-out → si usi il 1° frame libero dall'alto
- page-out e (contemporaneo) page-fault → si riusa il frame appena liberato (da page-out)
- page-out e page-fault relativi alla stessa pagina → page replacement evita sia page-out sia page-fault

Tempo	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16
Riferimenti	3	4	1	3	1	4	4	3	1	1	3	1	4	4	3	1	1
Resident Set	3	3	3	3	3	3			1	1	1	1	1	1	3	3	3
	4	4	4		4	4	4	4					4	4	4	4	
		1	1	1	1	1	3	3	3	3	3	3	3		1	1	
	x	x	x		x	x	x	x					x	x	x	x	
Page Fault					x		x	x					x		x	x	
Page Out					x		x	x	x				x	x	x	x	x

**Domande:** il comportamento del resident set nelle 8 iterazioni non riportate sarà:

- identico a quello dell'intervallo 10-16 (ripetuto 8 volte)? (SI/NO motivare)
  - **Risposta:** **NO** perché, seppur il periodo è di 7 tempi, le pagine 1 e 3 non si trovano nello stesso frame all'inizio e alla fine del periodo (**basta guardare la differenza tra 3-9 e 10-16**)
- ripeterà 4 volte l'intervallo 3-16? (SI/NO motivare)
  - **Risposta:** **NO** perché, seppur i 14 istanti (3-16) siano l'unione dei primi 2 periodi di 7 istanti, il 1° periodo (3-9) ha un comportamento iniziale diverso (**presenza della pagina 4 al t = 3**)
- avrà una diversa configurazione?
  - **Risposta:** si ripete 4 volte l'intervallo 3-16 (condizione della domanda precedente) se si elimina la pagina 4 a  $t = 3$  e il page-out a  $t = 4$ ; oppure si ripete 8 volte l'intervallo 10-16, alternando ad ogni iterazione i contenuti del 1° e 3° frame (condizione 1° domanda)

Quanti **page fault e page-out** ci saranno in TOT (comprese le 8 iterazioni mancanti)?

9 PF e 7 PO nella parte visualizzata +  $8 \times (3 \text{ PF e } 3 \text{ PO})$  nella parte mancante → **TOT = 33PF e 31PO**

## ESERCIZIO 3

Sia dato il frammento di codice di programma rappresentato che fa un calcolo matriciale (img a dx). Il codice macchina generato è eseguito in un sistema con **memoria virtuale gestita con paginazione** (pagine di 2KB) con page replacement **SECOND CHANCE**.

Un **float** è su **32 bit** e le **istruzioni** in codice macchina sono contenute in **1 sola pagina**. Si suppone che **M** e **V** siano allocati ad **indirizzi logici contigui** (prima **M**, poi **V**) a partire dall'indirizzo logico **0x5524AE00** [la matrice **M** è allocata in "row major", cioè per righe].

**Domande:**

- Quante **pagine (e frame)** sono necessarie per contenere la matrice e il vettore?

**Risposta:**

**dim(V) = 512 \* sizeof(float) = 2KB** →  $2\text{KB}/2\text{KB} = 1 \text{ pagina}$

**dim(M) = 512 \* 512 \* sizeof(float) = 1MB** →  $1\text{MB}/2\text{KB} = 500 = 512 \text{ pagine}$

L'indirizzo di partenza **0x5524AE00** **NON è multiplo di pagina** (dovrebbe terminare con 11 bit a 0 [pagine di 2KB]), ma inizia a  $\frac{3}{4}$  di pagina → quindi servono correzioni: **V+1 → 2 pagine; M+1 → 513 pagine**

- Ipotizzando che le variabili **i, j** siano allocate in registri della CPU, quanti **accessi in memoria** (in lettura e scrittura) fa il programma proposto, per accedere a dati? [non vanno conteggiati gli accessi a istruzioni]

**Risposta:**

**Notazione:**  $N_i$  numero iterazioni del for esterno  
 $N_j$  numero iterazioni del for interno

**Soluzione**

```
for (i=0;i<512;i++){ //  $N_i = 512$  iterazioni
    V[i]=0; // 1 Write per iterazione, in totale  $N_i(512)$  write
    for (j=0;j<=i;j++){ //  $N_j = i+1$  iterazioni, ripetute  $N_i(512)$  volte (i cambia)
        V[i] += M[i][j]; //  $N_j = \sum_{i=0..511}(i+1) = 512*(512+1)/2 = 128K+256 = 131328$ 
        oppure // per ogni iterazione 2 Read e 1 Write, in totale
        V[i] += M[i][511-j]; // 2*N_j read, N_j write
```

Quindi **semplicemente** si avranno:

- ❖ **1 Write nel ciclo esterno** ( **$V[i] = 0$** ) ( $i$ ) che avviene **512 volte** ( $0 \leq i \leq 512$ )
- ❖ **2 Read** (lettura del valore su  $V[i]$  e su  $M[i][j]$ ) **seguite da 1 Write** (modifica del valore di  $V[i]$ ) nel ciclo interno ( $j$ ) →  $512*(512+1)/2$

Quindi avrò  **$N_i$  WRITE +  $2*N_j$  READ +  $N_j$  WRITE**

```
float M[512][512],V[512];
...
for (i=0; i<512; i++) {
    V[i]=0;
    for (j=0; j<=i; j++) {
        if (i%2==0) {
            V[i] += M[i][j];
        }
        else {
            V[i] -= M[i][511-j];
        }
    }
}
```

- $N_T$  = tot accessi a dati in memoria;  $N_L$  = tot accessi a dati nella stessa pagina di 1 dei precedenti 10 accessi → si definisce **località del programma** (per i dati)  $L = N_L/N_T$ . Calcolala per il programma sopra proposto.

**SI SVOLGE L'ESERCIZIO SUPPONENDO ALLINEAMENTO A PAGINA. LA SOLUZIONE ESATTA RICHIEDEREbbe MINIME CORREZIONI**

Siccome ogni riga di M occupa esattamente una pagina, il fatto di percorrere la riga in avanti o all'indietro non ha alcun impatto su località e page fault. Località e page fault sono quindi gli stessi che si avrebbero con l'algoritmo semplificato:

```
for (i=0; i<512; i++) {
    V[i]=0;
    for (j=0; j<=i; j++) {
        V[i] += M[i][j];
    }
}
```

**Accessi a M (in lettura)**

Per ogni valore di  $i$  (per ogni riga di M), l'unico accesso non locale è il primo ( $j=0$ ) perché accede a una nuova pagina (nuova riga), a cui non si era ancora fatto accesso. Gli accessi non locali sono quindi 512

**Accessi a V (in lettura o scrittura)**

V sta tutto in una sola pagina. L'unico accesso non locale è il primo. Quindi 1 accesso non locale

**In totale** gli accessi NON locali sono  $1+512 = 513$

$N_T = N_i$  (azzeramenti del vettore) +  $3 \cdot N_j$  (iterazioni interne: 2 Read e 1 Write)

$N_L = N_T - 513$  (accessi NON locali)

$$\begin{aligned} L &= L = N_L/N_T = (N_T - 513)/N_T = (512 + 3 \cdot (128K + 256) - 513) / (512 + 3 \cdot (128K + 256)) = \\ &= 1 - 513 / (512 + 3 \cdot (128K + 256)) \approx 1 - 512/3 \cdot 128K = 1 - 1/3 \cdot 256 = \\ &= 1 - 0,0013 = 9,9987 \end{aligned}$$

- Calcolare i **page fault** generati dal programma proposto, supponendo che siano allocati 10 frame, di cui 1 utilizzato (già all'inizio) per le istruzioni

1 solo page fault per V (2 tenendo conto del non allineamento)

Su M 1 page fault per ogni riga/pagina. Poi si continua a farvi accesso.

In totale  $1+512 = 513$  page fault (2 + 512 tenendo conto del non allineamento)

# FILE SYSTEM

## ESERCIZIO 1

Ho 2 file system (**F1** e **F2**) su 2 volumi, basati rispettivamente su **FAT** e su **Inode**. **Puntatori** 32 bit (4B), **blocchi** 4KB. Entrambi i volumi hanno una parte riservata ai metadati (direttori, FAT e FCB oppure Inode, blocchi indice...) e una parte ai blocchi di dato (lo spazio riservato ai blocchi di dato è lo **stesso** per F1 e F2). **FAT** 150 MB.

**Domande:**

- Quanti **blocchi di dato** possono contenere (al massimo) i file presenti nei 2 file system?

**Risposta:**

**N blocchi F1:** ad ogni entry nella FAT corrisponde un blocco dato  $\rightarrow N_{F1} = 150\text{MB}/4\text{B} = 37,5 \text{ M blocchi}$

**N blocchi F2:**  $N_{F2} = N_{F1} = 37,5 \text{ M blocchi}$  (detto nel testo che sono uguali)

- Quale è il **max n° di blocchi dato liberi**?

**Risposta:**

**F1:** stesso valore di quelli totali  $\rightarrow 37,5 \text{ M blocchi}$

**F2:** come F1  $\rightarrow 37,5 \text{ M blocchi}$  (detto nel testo che sono uguali)

Si supponga che i **blocchi dato liberi di F1** siano gestiti mediante **free list**:

- La free list di F1 può essere rappresentata **direttamente nella FAT**?

**Risposta:**

**SI.** La FAT contiene sia le liste dei riferimenti a blocchi occupati dai file che la free-list: non ha senso fare altrimenti, in quanto **gli indici (nella FAT) dei blocchi liberi sono disponibili (non usati per file), quindi utilizzabili per la free list.**

Si vorrebbero organizzare **blocchi liberi di F2** mediante **lista concatenata di blocchi indice** (ognuno contiene un puntatore al next blocco indice e  $N_{Free}$  puntatori a blocchi di dato liberi):

- E' possibile adottare tale rappresentazione (mentre per i file si usa una rappresentazione basata su **Inode**)?

**Risposta:**

**SI.** La rappresentazione ad Inode non rende necessario che anche i blocchi liberi siano come file aggiuntivi accessibili con Inode; volendo, si potrebbe usare una bitmap o una lista semplice di blocchi: perciò "lista di blocchi indice, ognuno contenente puntatori a blocchi liberi" è lecita

- Quale è il valore di  $N_{Free}$ ?

**Risposta:**

$N_{Free} = 4\text{KB}/4\text{B}-1 = 1023$  [n° totale di puntatori in un blocco – puntatore al next in lista concatenata]

- Quanti **blocchi indice** servono (al massimo) per la **free list di F2**?

**Risposta:**

Ogni blocco indice punta a  $N_{Free} = 1023$  blocchi liberi

Max n° di blocchi liberi è  $N_{F2} = 37,5 \text{ M blocchi}$

Max n° di blocchi indice per la free list è  $N_{IndFree} = 37,5 \text{ M} / 1023 = 37,5 \text{ K} * 1024 / 1023 \approx 37,5 \text{ K}$

Siano dati un file "a.mp4" (in **F1**) di dimensione 317 MB e un file "b.mp4" (in **F2**) di dimensione 751 MB:

- Quanti **blocchi di dato** occupa *a.mp4*?

**Risposta:**  $317 \text{ MB} / 4 \text{ KB} = 79,25 \text{ K blocchi}$

- Quanti **indici nella FAT** sono utilizzati per *a.mp4*?

**Risposta:**  $79,25 \text{ K}$  (uguale al numero di blocchi dato)

- Quanti **blocchi di dato e di indice (escluso l'Inode)** occupa *b.mp4*?

Risposta:

DATO:  $751 \text{ MB} / 4 \text{ KB} = 187,75 \text{ K}$

INDICE: 10 blocchi dato sono puntati direttamente all'Inode. Un blocco indice punta a  $1024 = 1\text{K}$  blocchi dato  
Indice indiretto singolo: 1 blocco indice

Indici indiretti doppi: a 1° livello 1, a 2° livello  $[(187,75 \text{ K}-10) / 1\text{K}] - 1 = 187$  (quindi basta indiretto doppio)

Totale: 1 (singolo) + 1 (doppio primo livello) + 187 (doppio secondo livello) = 189

## ESERCIZIO 2

Sia dato il modulo di un **KERNEL** che deve gestire la **schedulazione delle richieste di accesso a un disco**:

- Perché le politiche **SCAN e C-SCAN** non servono con dischi SSD?

Risposta:

Perchè cercano di ridurre la distanza totale percorsa: questo ha senso per i dischi **magnetici** (si riduce il tempo di posizionamento delle testine), ma per i dischi **SSD** non ci sono costi legati alla differenza tra indici di blocco.

- E' possibile che il sistema abbia un disco **magnetico** (A) di 500GB e uno **SSD** (B) di 200GB, organizzati **in un unico volume**? Se sì, **dimensione del volume**?

Risposta:

SI. La tecnologia del disco viene vista a livello driver (basso). Partizionamento e formattazione permettono sia di ricavare più volumi da 1 disco che di unire più dischi in 1 volume ( $\text{dim volume} = 500 + 200 = 700\text{GB}$ )

Nel sistema (**ATTENZIONE**: si tratta di un'altra sotto-domanda, indipendente dalla precedente) si vogliono supportare **paginazione e swapping**

- E' più efficiente una partizione di **swap raw** oppure uno **swapfile in un filesystem esistente**?

Risposta:

**Più efficiente** (in  $t_{accesso}$ ) la partizione **raw** perché si gestiscono/utilizzano **direttamente blocchi fisici su disco**, senza passare dai vari livelli del file system

Si vuole realizzare un **nuovo tipo di filesystem**, nel quale è possibile avere **sia file sia direttori condivisi** (**albero di direttori è un DAG!**)

- Perché eventuali **cicli** nel grafo dei direttori rappresentano un **problema**?

Risposta:

Perché **con i cicli il grafo non è un DAG** (cioè un direttorio avrebbe come discendente/successore nell'albero un suo antenato/predecessore) → **va evitato!!!**

- Quali **strategie** è possibile adottare per evitare il problema dei cicli?

Risposta:

- Condividere solo file (foglie) e non i direttori
- **Garbage collection**: si accettano i cicli, ma periodicamente si esegue un controllo che individua e rimuove eventuali cicli nel file system (**più efficiente**)
- Ad ogni generazione di un nuovo link si controlla presenza di eventuali cicli (come garbage, ma ad ogni generazione di link)

## ESERCIZIO 3

File system Unix basato su **i-node** con **13 puntatori** (10 diretti, 1 indiretto singolo, 1 doppio e 1 triplo). Puntatori su **32 bit (4B)** e blocchi su **2KB**. Il file system contiene **1000 file** di **dimensione media 15MB** e che la **frammentazione interna totale** è di **1MB**, si calcoli il **max n° possibile di file con indice indiretto triplo e con indice indiretto doppio** che possono essere presenti nel file system.

**Risposta:**

Blocco indice contiene  $2\text{KB}/4\text{B} = 512$  **puntatori**

**Dimensione totale file** =  $15\text{MB} * 1000 = 15000\text{MB}$

**Occupazione blocchi** = **dim tot file + frammentazione interna** =  $15001\text{MB} = 15001 * 512$  blocchi

Max n° file con **indice indiretto doppio/triplo (N<sub>2</sub>/N<sub>3</sub>)** → per calcolare n° max bisogna considerare l'occupazione minima (**MIN<sub>2</sub>/MIN<sub>3</sub>** → **occupazioni minime** dei 2 tipi di file):

$$MIN_2 = (10 + 512 + 1) \text{ blocchi}$$

$$MIN_3 = (10 + 512 + 512^2 + 1) \text{ blocchi}$$

$$N_2 = \text{occupazione blocchi} * \text{puntatori}/MIN_2 = 15001 * 512 / MIN_2 = 14685$$

$$N_3 = \text{occupazione blocchi} * \text{puntatori}/MIN_3 = 15001 * 512 / MIN_3 = 29$$

$N_2 > n^o$  file presenti (che è 1000) → max n° file con indice indiretto doppio ( $N_2$ ) è **limitato a 1000** (e non 14685)

Max n° possibile di file **privi di frammentazione interna?**

**Risposta:**

La **frammentazione interna complessiva** è scritta nel testo (1MB)

Max n° di file privi di frammentazione interna (**N<sub>0MAX</sub>**) si ottiene ripartendo la frammentazione sul min n° di file ( $N_{min}$ ) aventi frammentazione max di 1 blocco (**1B** ciascuno), cioè:

$$N_{min} = 1\text{MB}/(2\text{KB} - 1\text{B}) = 513 \quad \rightarrow \quad N_{0MAX} = 1000 - 513 = 487$$

Si sa che per un file **a.dat** si utilizzano **2000 blocchi di indice**. Quanti **blocchi di dato** occupa e quale dimensione ha il file **a.dat** [qualora non siano possibili valori univoci, si determinino valori min e max]?

**Risposta:**

**N° blocchi di indice** per 1 file che riempie il 2° livello = 1 (singolo) + 1 (doppio 1° liv) + 512 (doppio 2° liv) = **514**

Dunque per **2000** blocchi indice occorre indice indiretto triplo → **2000 - 514 = 1486 blocchi indice**

Calcolo **n° blocchi indice triplo a 2° e 3°** livello ci sono [**a 1° livello ce n'è 1 solo**]:

$$1486/512 = 3 \rightarrow 1 \text{ (triplo, 1° liv.)} + 3 \text{ (triplo, 2° liv.)} + 1482 \text{ (triplo, 3° liv.)}$$

**Occupazione: N° blocchi dato (NBL)** [min e max dipendono da n° indici nell'ultimo blocco (**min 1; max 512**)]:

$$NBL_{min} = 10 \text{ (diretti)} + 512 \text{ (singolo)} + 512 * 512 \text{ (doppio)} + 1481 * 512 + 1 \text{ (triplo)} = 1020939$$

$$NBL_{MAX} = NBL_{min} + 511 = 1021450$$

**Dimensione:** la **min** si ottiene assumendo **frammentazione interna max** (2KB = 2047B). La **max** si ottiene assumendo **frammentazione 0**:

$$DIM_{min} = NBL_{min} * 2\text{KB} - 2047\text{B}$$

$$DIM_{MAX} = NBL_{MAX} * 2\text{KB}$$

## ESERCIZIO 4

Disco **SSD** (solid state drive). Quale è l'operazione che ne limita la vita e perché?

**Risposta:**

La durata di un disco SSD è limitata dal **n° max di cancellazioni** (effettuate per blocchi/pagine; hanno impatto anche sul n° max di write [in quanto una scrittura (ri-scrittura) deve essere preceduta da cancellazione])

Supponiamo che disco SSD da **1TB** sia garantito per **3 anni** e supporti max **5TB di dati scritti al giorno**. Qual è **n° max totale di byte leggibili e scrivibili durante l'intera vita del disco**?

**Risposta:**

**Lettura:** non c'è limite → le letture non sono un problema

**Scrittura:** limite è il **limite giornaliero \* 3 anni** →  $N_{write,max} = 5\text{TB} * 3 * 365 \approx 5.5 * 10^3\text{TB}$

Disco con struttura **RAID**. Che cosa si intende con:

- **MTTR** (Mean Time To Repair)? Tempo medio per riparare il disco quando si guasta
- **MTTDL** (Mean Time To Data Loss)? Tempo medio tra 2 perdite di dato (1 dato viene perso perché interviene un 2° guasto dopo un 1°, prima che la riparazione sia terminata)

Disco con struttura **RAID** con 2 dischi in configurazione "**mirrored**". Se un disco può guastarsi in modo indipendente dall'altro, con **MTTF** = 50000 ore e **MTTR** = 20 ore, quanto sarà il **MTTDL**?

**Risposta:**

$$\text{MTTDL} = \text{MTTF}^2 / (2 * \text{MTTR}) = 6.25 * 10^7 \text{ ore}$$

# I/O

## ESERCIZIO 1

File system dov'è **possibile accesso concorrente** (più processi) a 1 stesso file:

- Quali operazioni deve fare l'OS per fare **open()** e **close()**?

Risposta:

- **open** → parametri = **nome\_file** e **modalità** di apertura; return **fd** (file descriptor [o puntatore a file]):
  - 1) **nome\_file** viene cercato; se trovato, return **FCB** [copia nella **system-wide open-file table**]
  - 2) viene creata una nuova entry nella **per-process open-file table** [riferimento a FCB nella system wide], di cui si ritorna il puntatore o indice (**fd**)
- **close** → chiude il file, eliminando le relative entry nelle open-file tables (se non sono più utili):
  - 1) entry nella **per-process open-file table** viene cancellata
  - 2) **counter\_riferimenti--** nella **system-wide table**; se **counter==0**, cancello entry anche da questa

- A quali **strutture** (tabelle di gestione file) si deve fare accesso per realizzare una **read()** e/o una **write()**?

Risposta:

- **per-process open-file table** → nella entry c'è offset (posizione di read/write nel file) + modalità di accesso
- **system-wide open-file table** (coinvolta anche nella conversione indirizzo logico – fisico nel file [**p** e **d**])

- Cosa sono **system-wide open-file table** e **per-process open-file table**? Perché in un OS possono essere necessarie entrambe?

Risposta:

Le tabelle contengono tutte le info per gestire correttamente i file; sono necessarie entrambe perché un file può essere **aperto contemporaneamente più volte sia da 1 singolo processo** (anche con più thread) **sia da più processi**.

Nella **system-wide** troviamo: copia del FCB, eventuali primitive di sincronizzazione per accessi condivisi ecc...

Nella **per-process** troviamo: puntatore al file, modalità di accesso (read-only, read-write ...)

## ESERCIZIO 2

Ho un **buffer di kernel** usato come passaggio dai **blocchi di un file in transito tra disco e memoria user**. I dati da trasferire (es. mediante una **read(fd, addr, size)**, con **addr** e **size** che determinano la destinazione in memoria user) fanno un **passaggio in più: da disco a buffer kernel** (per una dimensione **size**), poi **da buffer kernel alla vera destinazione addr**.

Perché può essere vantaggioso il buffer kernel, pur costringendo a un passaggio in più in RAM?

Risposta:

Vantaggio principale è il **disaccoppiare memoria user da accesso a disco**: in sistemi con paginazione e/o swapping, si può fare **swap out di un processo utente** in attesa di I/O **oppure di una pagina** coinvolta in tale I/O **perchè il buffer user non rimane bloccato in attesa di I/O**.

Altro vantaggio del doppio buffer è poter fare **concorrenza “pipelining”** (trasferire memoria kernel – user e contemporaneamente disco – memoria kernel).

Altro vantaggio è la funzione di **cache** (buffer kernel già riempito all'inizio, per evitare processo in attesa di I/O).

⚠ Il vantaggio **NON** è quello di **evitare al processo user di fare l'I/O!!!** Il processo USER NON NE HA I PRIVILEGI: sia con buffer che senza, l'I/O viene effettuato da una syscall, mediante un opportuno driver (di KERNEL): in un caso il driver lavora su memoria user (e la blocca) nell'altro caso su buffer kernel

In un sistema con paginazione, il parametro **size** può essere arbitrario oppure deve essere un multiplo della dimensione di blocco o di una pagina?

Risposta:

Il parametro **size** è arbitrario perché la **read** è a livello **user** (non ha dipendenza con le strategie di paginazione).

Si supponga di usare un “**doppio**” buffer (**double buffer**) [ricorda teoria: mentre 1 buffer (detto “kernel”) fa trasferimento da disco, 1 buffer (detto “user”) può essere usato per trasferire dati alla destinazione in mem user; dopo ogni operazione si scambiano i ruoli dei 2 buffer].

Supponendo di voler leggere sequenzialmente un file da **200KB**, quanti Byte passano in totale sul bus nei 2 casi (**singolo e doppio buffer**)? Per le lettura da disco si usa **DMA**. Nel caso di doppio buffer si dimezza?

Risposta:

Il doppio buffer velocizza le operazioni, ma il numero di byte gestiti è lo stesso [cambia la collocazione dei dati]. I 200KB passano 1 volta sul bus durante il trasferimento in DMA **disco→buffer**. Il trasferimento **buffer→memoria user** è una copia da RAM a RAM.

Nel caso di trasferimento gestito dalla **CPU**, i dati passano **2 volte sul bus dati** (RAM→CPU; CPU→BUS); in questo caso in totale transitano **200KB + 2\*200KB = 600KB**.

## ESERCIZIO 3

Si considerino i 2 tipi di sincronizzazione I/O **sincrono e asincrono**. Differenze? Si definisca poi I/O **bloccante e non bloccante**: si tratta di sinonimi di asincrono e sincrono?

Risposta (sintetica):

**bloccante** ≡ **sincrono** (il processo che effettua I/O attende il completamento)

**non bloccante** permette al processo di proseguire; **asincrono** è **non bloccante** + tecniche per gestire il completamento dell’I/O (funzioni di *wait* oppure *callback* richiamate dall’OS al completamento dell’I/O)

Un I/O **sincrono** può essere fatto in **polling** oppure è necessario farlo in **interrupt**? E **asincrono**?

Risposta:

Fare una syscall **read/write** in modo sincrono o asincrono può esser fatto con driver **sia in polling sia in interrupt**

In che modo un processo beneficia di un I/O asincrono? Si può scrivere per un I/O **asincrono** un programma con istruzioni dopo l’I/O che dipendano dai **dati coinvolti nell’I/O**?

Risposta:

Il processo beneficia perchè fa altro mentre l’I/O è in corso (sorta di concorrenza).

Il processo **non può usare i dati coinvolti nell’I/O**; se vuole, deve sincronizzarsi-mettersi in *wait* sull’I/O asincrono.

## ESERCIZIO 4

Gestione di una **richiesta di I/O a blocchi** realizzato con **DMA**:

- Cosa si intende con **cycle stealing**?

R: **Cycle stealing** è la **sottrazione di cicli di BUS alla CPU**, mentre il **DMA** controlla i BUS di accesso alla RAM

- Perché il trasferimento in **DMA** è vantaggioso rispetto all’IO programmato?

Risposta: 2 motivi:

- 1) I trasferimenti con DMA avvengono “direttamente” tra IO e RAM (no CPU, x2 operazioni)
- 2) Mentre si trasferiscono dati in DMA la CPU può fare altro

- Devo trasferire **40KB** di dati **disco→RAM**, quanti Byte passano sul bus nei 2 casi (**DMA e IO programmato**)?

R: 40KB in DMA e 80KB nel caso di IO programmato (**40KB \* 2**, perché passano dalla CPU [vedi sopra])

- Se si usa per l’I/O un **buffer kernel**, in che cosa il doppio buffer si differenzia dal singolo buffer e per quale motivo può risultare vantaggioso? R: [Vedi sopra: mentre un buffer viene scritto, l’altro (riempito in precedenza) può essere letto in parallelo; con il buffer singolo 1 deve attendere il completamento dell’altro]

# OS161

## ESERCIZIO 1 (AS\_DEFINE\_REGION)

Ecco una parte della funzione `as_define_region` (file `dumbvm.c`). Si ricevono come parametri `as = 0x80048720`, `vaddr = 0x412370` e `sz = 4128` [ricorda che `PAGE_SIZE = 4096` e `PAGE_FRAME = 0xfffff000`]. Si simulino le istruzioni proposte, indicando in esadecimale i valori di operandi e risultato.

```
as_define_region(struct addrspace *as, vaddr_t vaddr, size_t sz,
                 int readable, int writeable, int executable) {
    size_t npages;
    /* Align the region. First, the base... */
    sz += vaddr & ~(vaddr_t)PAGE_FRAME;
    vaddr &= PAGE_FRAME;
    /* ...and now the length. */
    sz = (sz + PAGE_SIZE - 1) & PAGE_FRAME;
    npages = sz / PAGE_SIZE;
    ...
}
```

Risposta:

<pre>sz += vaddr &amp; ~(vaddr_t)PAGE_FRAME; vaddr &amp;= PAGE_FRAME; sz = (sz + PAGE_SIZE - 1) &amp; PAGE_FRAME; npages = sz / PAGE_SIZE;</pre>	<pre>4096 = 0x1000 4128 = 0x1020  0x1020 += 0x412370 &amp; 0x000FFF sz &lt;- 0x1020+0x370 = <b>0x1390</b> 0x412370 &amp;= 0xFFFF000 vaddr &lt;- <b>0x412000</b> sz &lt;- (0x1390+0xFFFF) &amp; 0xFFFF000 = <b>0x2000</b>  npages &lt;- 0x2000 / 0x1000 = 2</pre>
--	--

Supponiamo che `sz` sia inferiore alla dimensione di una pagina `PAGE_SIZE` (es. **4090**). Si può avere `npages = 2?`

Risposta:

SI perchè il segmento viene allineato a un multiplo di pagina sia all'inizio che alla fine: in questo caso c'è una forma di frammentazione interna sia sulla 1^ sia sull'ultima pagina.

Con `4090 = 0xFFA`, `sz` avrebbe prima assunto il valore `0xFFA+0x370 = 0x126A`, quindi `0x2000`

## ESERCIZIO 2 (GETFREEPPAGES + FIRST & BEST FIT)

Ecco una realizzazione di `getfreeppages`, che alloca un intervallo di `npages` contigue di memoria fisica libera →

```
static paddr_t
getfreeppages(unsigned long npages) {
    paddr_t addr = 0;
    long i, first, found, np = (long)npages;

    if (!isTableActive()) return 0;
    spinlock_acquire(&freemem_lock);
    for (i=0,first=found=-1;i<nRamFrames;i++) {
        if (freeRamFrames[i]) {
            if (i==0 || !freeRamFrames[i-1])
                /* set first free in an interval */
                first = i;
            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;
    }
    spinlock_release(&freemem_lock);
    return addr;
}

/* first-fit */

/* best-fit */
```

La funzione realizza una politica di **allocazione best-fit, worst-fit, first-fit o altro?**

Risposta:

Nessuna delle tre.

First-fit NO perché le iterazioni **non si fermano alla prima soluzione trovata**

Best-fit e Worst fit NO in quanto **non c'è una ricerca/gestione di minimo o massimo**

La soluzione ritornata è l'**ultima tra quelle trovate** → una sorta di "**last-fit**"

Modifica la funzione (scrivendo nel riquadro libero le parti modificate, che dovrebbero limitarsi alle istruzioni evidenziate in grigio) **in modo tale da realizzare una politica first-fit e una best-fit**

Risposta:

<pre>static paddr_t getfreepages(unsigned long npages) {     paddr_t addr = 0;     long i, first, found, np = (long)npages;      if (!isTableActive()) return 0;     spinlock_acquire(&amp;freemem_lock);     for (i=0,first=found=-1;i&lt;nRamFrames;i++) {         if (freeRamFrames[i]) {             /* controlla ogni frame libero (anche                interno all'intervallo.                Se è il primo di un intervallo,                "ricorda" la posizione. */             if (i==0    !freeRamFrames[i-1])                 /* set first free in an interval */                 first = i;             /* ogni intervallo compatibile con np                viene assegnato. Quindi si ritorna                l'ultimo */             if (i-first+1 &gt;= np)                 found = first;         }     }     if (found&gt;=0) {         for (i=found; i&lt;found+np; i++) {             freeRamFrames[i] = (unsigned char)0;         }         allocSize[found] = np;         addr = (paddr_t) found*PAGE_SIZE;     }     spinlock_release(&amp;freemem_lock);     return addr; }</pre>	<pre>/* ATTENZIONE: la soluzione proposta non è l'unica: ne sono possibili altre (simili) */  /* first-fit: appena trovato esce */ /* variante con uscita strutturata */ ...     for (i=0,first=found=-1;          i&lt;nRamFrames &amp;&amp; found&lt;0; i++) { ...     /* variante con uscita non strutturata */     ...         if (i-first+1 &gt;= np) {             found = first;             break;         }     ...  /* best fit: ricerca minimo (solo alla fine di un intervallo) */ int min; ...     /* si possono in alternativa mettere tutte        le condizioni in AND (unico if) */     /* se è l'ultimo frame di un intervallo */     if (i==nRamFrames-1    !freeRamFrames[i+1])         /* se la dimensione va bene */         if (i-first+1 &gt;= np)             /* se batte il minimo provvisorio */             if (found&lt;0    i-first+1 &lt; min) {                 found = first; min = i-first+1;             }     ...</pre>
---	---

## ESERCIZIO 3 (SYSCALL)

Si spieghi nella funzione **syscall()** che cos'è la variabile **callno** (a cui si assegna il valore **tf->tf\_v0**)?

Risposta:

È il **selettor della system call da effettuare** usato in **syscall** per selezionare il relativo **case** → questo valore viene assegnato al campo **tf->tf\_v0** dalla routine che gestisce la **trap** e chiama **mips\_trap->syscall**.

Cosa significano le seguenti istruzioni alla fine di **syscall()**? Cosa sono i campi **tf\_v0** e **tf\_a3** del **trapframe**?

Risposta:

```
if (err) {  
    tf->tf_v0 = err;  
    tf->tf_a3 = 1;  
}  
else {  
    tf->tf_v0 = retval;  
    tf->tf_a3 = 0;  
}
```

Le istruzioni, poste alla fine della funzione **syscall**, gestiscono il lo stato e il valore di ritorno:

- In **v0** viene posto il valore di ritorno della system call (che coincide con un codice di errore in caso, appunto di errore)
- In **a3** viene ritornato lo stato successo(0)/errore(1)

## ESERCIZIO 4 (LOCK)

Realizzazione dei LOCK in OS161. Quale **thread** deve essere considerato **owner** (proprietario) **di un lock**?

- thread che ha **creato il lock**?  
R: **NO** → creatore del lock ≠ owner
- ultimo **thread che ha chiamato lock\_acquire**?  
R: **Non necessariamente** → solo se lock\_acquire viene fatta sul lock in questione (non su un altro lock) e il thread abbia effettivamente **acquisito il lock** (non sia ancora in attesa)  
⚠ **Owner di un lock** = thread che ha effettuato **lock\_acquire** sul lock ed ha **superato l'attesa**

Date le funzioni **lock\_release** e **lock\_do\_i\_hold** in figura, sono presenti **errori**: trova e correggi?

<pre>void loc_release(struct lock *lock) {     KASSERT(lock != NULL);     spinlock_acquire(&amp;lock-&gt;lk_lock);     KASSERT(lock_do_i_hold(lock));     lock-&gt;lk_owner=NULL;     wchan_wakeone(lock-&gt;lk_wchan, &amp;lock-&gt;lk_lock);     spinlock_release(&amp;lock-&gt;lk_lock); }</pre>	<pre>bool lock_do_i_hold(struct lock *lock) {     spinlock_acquire(&amp;lock-&gt;lk_lock);     if (lock-&gt;lk_owner==curthread)         return true;     spinlock_release(&amp;lock-&gt;lk_lock);     return false; }</pre>
---	--

Risposta:

Errore nella **lock\_do\_i\_hold** → il **return true** se owner == current\_thread, ma **senza spinlock\_release** (questa viene fatta solo nel caso false) [soluzione potrebbe essere una variabile booleana al posto di ritornare subito]:

```
bool lock_do_i_hold(struct lock *lock) {
    bool ret;
    spinlock_acquire(&lock->lk_lock);
    ret = lock->lk_owner==curthread;
    spinlock_release(&lock->lk_lock);
    return ret;
}
```

Errore nella **lock\_release** → **spinlock\_acquire** prima della **lock\_do\_i\_hold**, che tenterà di acquisire lo stesso **spinlock** (**DEADLOCK**) [soluzione = spostare la spinlock\_acquire dopo la chiamata a lock\_do\_i\_hold]

## ESERCIZIO 5 (SINCRONIZZAZIONE)

Perché in contesto **multi-core** (più CPU) non si può realizzare la **mutua esclusione** semplicemente disabilitando e riabilitando l'**interrupt**?

Risposta:

Perché l'interrupt verrebbe **disabilitato solo sulla CPU corrente** → non precluderebbe quindi l'esecuzione di altri thread/processi sulle altre CPU

Dato il codice (ridotto all'essenziale) delle **funzioni semaforiche P e V** qui riportato:

```
void P(struct semaphore *sem) {
    spinlock_acquire(&sem->sem_lock);
    while (sem->sem_count == 0) {
        wchan_sleep(sem->sem_wchan,
                    &sem->sem_lock);
    }
    sem->sem_count--;
    spinlock_release(&sem->sem_lock);
}
```

```
void V(struct semaphore *sem) {
    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);
}
```

Rispondi alle seguenti domande:

- A cosa serve lo **spinlock**? (facendo riferimento al suo uso in entrambe le funzioni)

Risposta:

Lo **spinlock** serve per poter usare un **wait-channel**, ovvero per chiamare **wchan\_sleep** e **wchan\_wakeone** in quanto serve a garantire la **gestione in mutua esclusione della condizione sem->sem\_count**

- Perché la **P** contiene un ciclo **while** al posto di **if(sem->sem\_count == 0)**, mentre la **V** no?

Risposta:

Perché la sincronizzazione tra **wchan\_wakeone** e **wchan\_sleep** (ovvero il risveglio) **non garantisce che la condizione, che è vera alla chiamata di wchan\_wakeone, lo sia ancora al ritorno da wchan\_sleep → altri thread potrebbero modificarla nel frattempo** [⚠ questi altri thread non sono svegli per un risveglio errato/spurio, ma semplicemente si svegliano anch'essi correttamente per un'altra chiamata di V e generano questa condizione problematica]

- Perché la **wchan\_sleep** riceve come **parametro lo spinlock**? Vale lo stesso motivo per la **wchan\_wakeone**?

Risposta:

La **wchan\_sleep** lo usa davvero: deve **rilasciare lo spinlock** (prima di mettere thread in stato “**wait**”) **per poi riprenderlo al risveglio** (prima di return al chiamante).

La **wchan\_wakeone** **non deve fare nulla sullo spinlock** (le versioni Unix/Linux **NON** hanno questo parametro [visto a teoria]) → in OS161 il parametro viene solo **verificato** (in una KASSERT) **perché al momento delle chiamata il thread deve essere owner dello spinlock**

- È possibile che la chiamata alla **wchan\_wakeone** svegli più di 1 thread in attesa su **wchan\_sleep**? Se **NO**, perché? Se **SI**, come si fa per rilasciare 1 solo thread in attesa su **P**?

Risposta:

**NO.** La **wchan\_wakeone** svegli **1 solo thread** in attesa su **wchan\_sleep**. La funzione che sveglia più thread in attesa (tutti) è la **wchan\_wakeall**

## ESERCIZIO 6 (my\_sys\_exit)

Siamo in OS161. Si supponga che la funzione **syscall()**, se ha il valore **SYS\_exit** in **callno**, chiama la funzione **my\_sys\_exit()**. Definirne il **prototipo** e scriverne la **chiamata** (con parametri attuali) in **syscall()**, dato questo frammento di codice incompleto.

Da dove può o deve provenire il valore **status**? A che variabile o campo di struct va assegnato?

```
my_sys_exit(...)

    ... = status;
    ...

    cv_signal(...);

    ...
    ...

    thread_exit();
}
```

```
my_sys_exit(int status) { // lo stato viene ricevuto come parametro
    struct proc *p = curproc; // serve per poter accedere al processo dopo averlo
                             // staccato da curthread (curproc non più valido)

    p->p_status = status; // salva lo stato di ritorno per la waitpid
    proc_remthread(curthread); // stacca il processo dal thread

    // segnala al processo che fa la waitpid (per usare cv_signal è
    // necessario possedere il relativo lock)
    lock_acquire(p->p_lock);
    cv_signal(p->p_cv, p->p_lock);
    lock_release(p->p_lock);

    // meglio NON fare as_destroy qui (lo farà la proc_destroy)
    // il thread finisce qui (diventa zombie)
    thread_exit();
}
```

La **chiamata** sarà del tipo (dato che **tf\_a0** contiene lo **stato**):

```
my_sys_exit((int)tf->tf_a0);
```

## ESERCIZIO 7 (copia di argv e argc)

Perché in OS161, per gestire gli argomenti al main, è necessario creare una **copia di argv e argc**?

Risposta:

Perchè gli **argomenti al main** devono essere in memoria **user** affinchè il programma utente possa accedervi; dato che gli argomenti al main **sono in origine in memoria kernel**, è necessario farne una **copia**

Dove va creata la copia?

Risposta:

Va fatta ovviamente in **memoria user**, ma in particolare in una **parte accessibile dell'address space** → quindi la soluzione più semplice è l'**inizio** (indirizzi alti) **dello (user) stack**

Perché non bastano i valori originali **nargs** e **args**, passate alla **cmd\_prog** (prototipo `static int cmd_prog(int nargs, char **args)`)?

Risposta:

Già detto prima: i valori originali sono in **memoria kernel**, quindi **non accessibili al processo user** [si potrebbe aggiungere che, anche se il processo potesse accedervi, sarebbero nello stack di un altro kernel thread, ovvero quello del **menu** (anch'esso da duplicare per garantire consistenza/accessibilità da altro thread)]

## ESERCIZIO 8 (sys\_waitpid, sys\_open, sys\_close + copyin, copyout)

Si può fare la **sys\_waitpid** utilizzando per l'attesa un lock (su cui fare **lock\_acquire**), mentre la segnalazione da parte della **sys\_exit** viene realizzata con **lock\_release** dello stesso lock?

Risposta:

NO. Come già detto in precedenza: un lock non può essere usato per sincronizzazioni di tipo wait-signal, proprio per il problema dell'**ownership** → il lock serve solo per **mutua esclusione**

[per wait-signal si usano **semafori o CV** (dove il lock associato serve per mutua esclusione, non per wait-signal)]

Si vogliono fare le syscalls **sys\_open** e **sys\_close**. Bisogna associare a un file descriptor il concetto di **ownership** da parte di un thread, in modo che **solo il thread che ha fatto open** di un file può fare **close** del file?

Risposta:

NO. Un file può essere chiuso da un thread diverso da quello che lo ha aperto. Un file ha però un concetto di **ownership legato al processo** (in quanto un file descriptor è associato a **per-process open-file table**)

A cosa servono le funzioni OS161 **copyin e copyout**?

Risposta:

Fanno copie di dati tra memoria user e kernel → **copyin** verso memoria kernel, duale la **copyout**. La differenza rispetto ad altre copie consiste nella **gestione consistente di eccezioni legate a indirizzi/puntatori user non validi**, impedendo al kernel di terminare in modo anomalo

Sostituiamo una chiamata **copyin(src,dst,size)** con **memmove(dst,src,size)**. È lecito? Si perde/guadagna qualcosa oppure sono equivalenti?

Risposta:

SI, è lecito, ma si perde la protezione da eccezioni/erri

## ESERCIZIO 9 (load\_elf + VOP\_READ)

Sia data la porzione della funzione **load\_elf** in figura (dove si vuole leggere dal file ELF l'**header** del file, con destinazione la struct **eh**):

```

load_elf(struct vnode *v, vaddr_t *entrypoint)
{
    Elf_Ehdr eh; /* Executable header */
    int result;
    struct iovec iov;
    struct uio ku;
    /*
     * Read the executable header from offset 0 in the file.
     */
    result = VOP_READ(v, &eh, sizeof(eh));
    ...
}

```

Questa porzione è **errata**. Spiega perché e correggi!

**Risposta:**

Chiamata a **VOP\_READ** è **errata** → la lettura va fatta con una strategia diversa: **prima si definisce l'operazione** da effettuare mediante **uio\_kinit** [per operazione in memoria kernel] usando le variabili **ku** e **iov**; poi si chiama **VOP\_READ**, usando **ku**.

La versione corretta è:

```

uio_kinit(&iov, &ku, &eh, sizeof(eh), 0, UIO_READ);
result = VOP_READ(v, &ku);

```

Si dica in breve che cosa sono (e a cosa servono nel programma proposto):

- parametro **v**  
R: puntatore a vnode (ovvero il FCB del file ELF da cui si legge)
- variabile **ku**  
R: struct dove si imposta l'operazione di I/O (con **uio\_kinit**) prima di farla (mediante **VOP\_READ**). Punta a **iov** e contiene campi per definire lettura/scrittura kernel/user
- variabile **iov**  
R: struct dove vengono caricati indirizzo in memoria e dimensione della destinazione di una **VOP\_READ**

## ESERCIZIO 10 (uio\_kinit, load\_elf, load\_segment)

Ecco parti delle funzioni **uio\_kinit**, **load\_elf** e **load\_segment**:

```

void uio_kinit (struct iovec *iov, struct uio *u,
                void *kbuf, size_t len,
                off_t pos, enum uio_rw rw) {
    iov->iov_kbase = kbuf;
    iov->iov_len = len;
    u->uio_iov = iov;
    u->uio_iovcnt = 1;
    u->uio_offset = pos;
    u->uio_resid = len;
    u->uio_segflg = UIO_SYSSPACE;
    u->uio_rw = rw;
    u->uio_space = NULL;
}
int load_elf (struct vnode *v, vaddr_t *entrypoint) {
    Elf_Ehdr eh; /* Executable header */
    Elf_Phdr ph; /* "Program header" = segment header */
    int result;
    struct iovec iov;
    struct uio ku;
    ...
    uio_kinit(&iov, &ku, &eh, sizeof(eh), 0, UIO_READ);
    result = VOP_READ(v, &ku);
    ...
}

```

```

load_segment (struct addrspace *as, struct vnode *v,
              off_t offset, vaddr_t vaddr,
              size_t memsize, size_t filesize,
              int is_executable) {

    struct iovec iov;
    struct uio u;
    int result;

    iov.iov_ubase = (userptr_t)vaddr;
    iov.iov_len = memsize; // length of the memory space
    u.uio_iov = &iov;
    u.uio_iovcnt = 1;
    u.uio_resid = filesize; // amount to read from the file
    u.uio_offset = offset;
    u.uio_segflg = is_executable ? UIO_USERISPACE : UIO_USERSPACE;
    u.uio_rw = UIO_READ;
    u.uio_space = as;

    result = VOP_READ(v, &u);
    ...
}

```

Si spieghi brevemente il ruolo della **struct iovec** e della **struct uio**, in relazione alla successiva **VOP\_READ**.

**Risposta:**

**iovec** → contiene puntatore alla destinazione della read + dimensione (ovvero **&eh** e **sizeof(eh)** nella **load\_elf**, **vaddr** e **memsize** nella **load\_segment**)

**uio** → contiene tutte le info necessarie per l'IO:

- puntatore a **iovec**
- offset + n° byte da leggere nel file
- info sul virtual space (kernel/user) + tipo di I/O (R/W) da fare

⚠ Prima di fare I/O in spazio **kernel** → basta chiamare **uio\_kinit** (per collegare le 2 struct)

Prima di fare I/O in spazio **user** → le 2 struct vanno caricate in forma esplicita (non c'è **uio\_kinit** in **user**)

Perché **load\_segment** usa **UIO\_USERISPACE/UIO\_USERSPACE**, mentre all'inizio della **load\_elf** si usa (tramite **uio\_kinit**) **UIO\_SYSSPACE**?

**Risposta:**

Perché la 1^ parte di **load\_elf** acquisisce dal file elf l'**header** e lo mette in una variabile locale in memoria **kernel** (I/O di tipo **UIO\_SYSSPACE**).

**load\_segment** deve acquisire i **segmenti veri e propri** dal file elf alle **partizioni di memoria user appena allocate per il processo** (I/O di tipo **UIO\_USERISPACE** per la partizione **code** e **UIO\_USERSPACE** per la partizione **data**)

Perché al campo **u->uio\_space** in un caso viene assegnato **NULL**, mentre nell'altro si assegna **as**?

**Risposta:**

L'assegnazione fornisce le info necessarie alla traduzione indirizzi logici-fisici.

Per lo spazio **kernel** non serve (quindi puntatore **NULL**) perchè la traduzione è sommare/sottrarre **MIPS\_KSEG0**.

Per lo spazio **user** serve il puntatore alla struct **addrspace** del processo (dove sono definite le mappature logico-fisico dei 2 segmenti e dello stack)