4. Virtual File System

- **Virtual File System**: insieme di tutti i moduli software per operazioni di I/O (software di livello kernel)
- **Canale di I/O**: descriptor o Handle → identificatore logico con il quale, tramite TABELLA DEGLI OGGETTI, accedere all'istanza dell'oggetto di I/O.

File System

File := insieme di RECORD, tra cui un RECORD DI SISTEMA (RS) per i metadati del file

I file possono essere acceduti in maniere differente, a seconda dell'accesso ai record e della sua gestione:

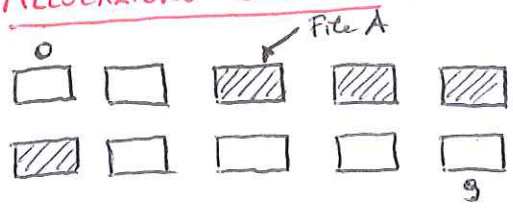
- **Accesso Sequenziale**:
  I records sono acceduti sequenzialmente e l'indice di lettura scrittura avanza di 1 ad ogni accesso.
  Riposizionamento solo ad INIZIO FILE → POCO FLESSIBILE!
  Tipico di file SEQUENZIALI (dim fissa) e file A NUCCHIO (variabili).
- **Accesso Sequenziale Indicizzato**:
  Al file viene associato un FILE DI INDICI, contenente uno SOTTOINSIEME delle chiavi, tramite i quali abbiamo accesso ad un range di record, dove ci muoviamo in maniera sequenziale.
  I records sono quindi ORDINATI PER CHIAVE, il riposizionamento puo' avvenire anche qui solo ad INIZIO FILE.
- **Accesso Diretto**
  Si puo' accedere a qualsiasi record tramite FUNZIONE HASH, quindi posso avere riposizionamento in QUALSIASI PUNTO del FILE!
  Dopo aver acceduto un record i, l'indice di lettura/scrittura puntera' al successivo record $i+1$.

• ALLOCAZIONE DI FILE:

Ciascun file è allocato in memorie di masse come un insieme di BLOCCHI LOGICI, non per forze contigui.

Il RS tiene traccia di quanti blocchi sono allocati per un file e dove sono. per capire quali blocchi sono liberi, il FILE SYSTEM usa strutture dati come Liste Libere o Bitmap.

• ALLOCAZIONE CONTIGUA:



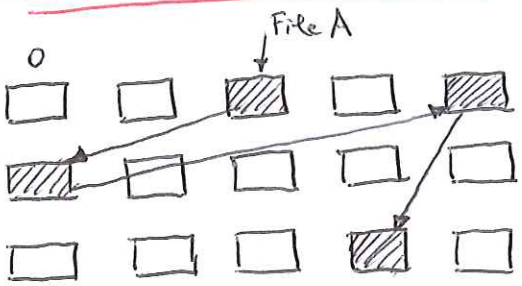
RS: file A - blocco inizio: 2, dim: 4

Il RS registra il blocco iniziale e il numero di blocchi allocati.

PROBLEMA: FRAMMENTAZIONE ESTERNA (pochi blocchi liberi contigui)

↳ SOLUZIONE: RICOMPATTAZIONE, ma è costosa!

• ALLOCAZIONE A CATENA:



RS: File A - blocco di inizio: 2

RS mantiene il riferimento al 1° blocco.

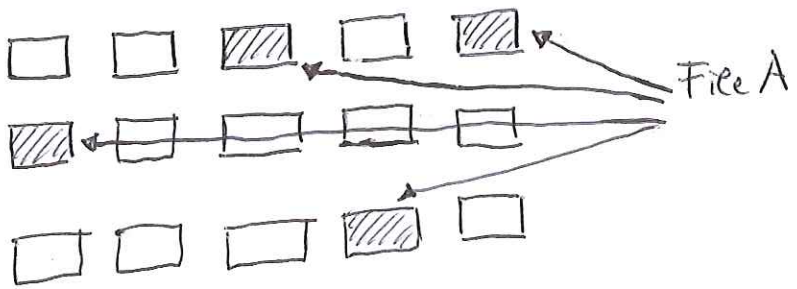
NON ho frammentazione!

Blocchi collegati tramite puntatori, come una LISTA.

PROBLEMA: accesso costoso; caricare tutti i blocchi precedenti per accedere uno

Si può fare RICOMPATTAZIONE, per ridurre i movimenti della testina

• ALLOCAZIONE INDICIZZATA:



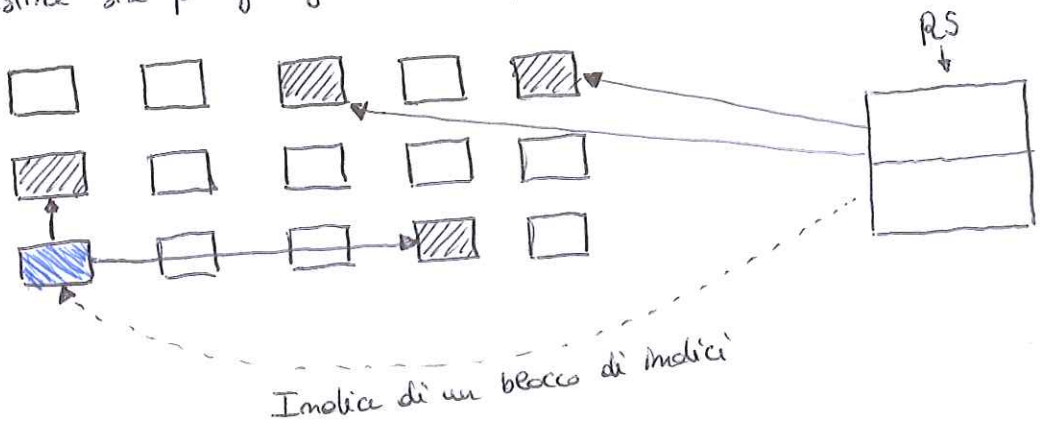
RS
↓
- blocco 2
- " 4
- " 5
- " 13

Il RS è più grande, contiene gli indici ad ogni blocco.

Può essere utile RICOMPATTARE per gli stessi motivi.

• ALLOCAZIONE INDICIZZATA A LIVELLI MULTIPLI :

L'RS mantiene degli indici diretti, più degli indici indiretti che puntano ad un blocco in memoria, il quale è un blocco di indici che puntano ad altri blocchi dati.
Utile per file di GRANDI DIMENSIONI. L'RS non cresce troppo e si riesce a gestire sia per file grandi che piccoli.



• BUFFER CACHE :

BUFFER CACHE := struttura dati SOFTWARE del Sistema Operativo che funge da buffer temporaneo per blocchi dati acceduti in I/O.

Utile per LETTURA ANTICIPATA (LOCALITÀ) e SCRITTURA RITARDATA;
Si può saturare! Politiche di sostituzione: 1) Least Recently Used
2) Least Frequently Used

• BUFFER CACHE A 2 SEZIONI :

C'è un contatore di riferimenti, che in realtà conta il numero di transizioni in sezione nuova. Si sostituisce il blocco col contatore più basso.

* In sezione nuova, i riferimenti NON aumentano il contatore;
* In sezione vecchia, un riferimento aumenta di 1 il contatore ed il blocco è portato in SEZIONE NUOVA.

• A 3 SEZIONI : c'è anche una SEZIONE INTERMEDIA, in cui avviene l'incremento dei contatori, che permette ai blocchi usciti dalla sezione nuova di avere più chances di restare in buffer cache.

FILE SYSTEM UNIX

- **i-node** := è il nome dato al RECORD DI SISTEMA (RS)

  [1 record = 1 byte]

C'è un ARRAY DI I-NODES di taglia fissa → si può SATURARE!

L'allocazione dei files è INDICIZZATA A LIVELLI MULTIPLI: l'i-node mantiene indici diretti, indiretti, doppiamente indiretti, triplicamente indiretti.

- **Esempio: TAGLIA MAX DI FILE**

Blocchi su disco = 512 byte
Indirizzo su disco = 4 byte

i-node (RS):
- 10 indici diretti
- 1 indiretto
- 1 dopp. indiretto
- 1 tripe. indiretto

$$\Rightarrow \#\text{indirizzi in un blocco} = \frac{512 \text{ byte}}{4 \text{ byte}} = 128 \text{ indirizzi/blocco}$$

$$\text{MAX BLOCCHI} = 10 + 128 + 128^2 + 128^3 = 2.113.674 \text{ blocchi}$$

$$\Rightarrow \text{MaxFile} = 2.113.674 \text{ blocchi} \cdot 512 \text{ byte} = 1.082.201.088 \text{ byte} \approx \boxed{1 \text{ Gb}}$$

- **ACL - Access Control List:**

Sono dei FILE SHADOW (con i-node shadow) associati ai files, con cui si possono specificare permessi d'accesso a grana fine per specifici utenti o gruppi.

* I DISPOSITIVI DI I/O sono gestiti come FILE! (descrittori)

Tutti i descrittori vengono ereditati sia tramite fork() che tramite execX()!

- **HARD LINK:**

```
int link ( const char* oldpath , const char* newpath)
```

Collega 2 file allo stesso contenuto e quindi allo STESSO I-NODE.
L'associazione nomefile - contenuto è registrata all'interno della directory.
Un contenuto e quindi un i-node viene RILASSO quando sia il reference counter dei file, sia quello delle sessioni sono a 0.
Anche la sessione viene chiuse solo quando tutti i CANALI aperti verso quella sessione vengono chiusi!

- **SYMBOLIC LINK:**

```
int symlink (const char* oldpath, const char* newpath)
```

Crea un SOFT LINK: nuovo file, collegato a oldpath. Permette di accedere agli stessi contenuti del file originale, ma NON incrementa il reference counter dell'i-node!

- **Residui di scrittura:**

do-while dentro un while!

```
res_r = read (0, buffer, MAXBUF);
while (res_r) {
    prev_w = 0;
    res_w = 0;
    do {
        prev_w += res_w;
        res_w = write (1, &buffer[prev_w], res_r - prev_w);
    } while (res_w + prev_w < res_r);
    res_r = read (0, buffer, MAXBUF);
}
```

FILE SYSTEM WINDOWS

Gli HARD-DRIVES sono divisi in volumi o partizioni e organizzati in cluster (unione di blocchi) da 512 a 64K bytes.

- Per ogni partizione, si ha una MFT - MASTER FILE TABLE, equivalente degli array di i-nodes.
  Ad ogni FILE corrisponde ALMENO 1 elemento della MFT (se il file è grande, più di 1).
  Ogni elemento contiene: nome file + info sicurezza + DATI o PUNTATORI a DATI (ad altri elementi della MFT)

* Lo SWAPPING è su FILE ("file swap") e gestito da demoni kernel.
  Non c'è un'area di swap dedicata come in UNIX!

I descrittori (in UNIX), qui sono (HANDLE) → CANALI

- **ACL**:

È formata da una lista di ACE (Access Control Entry); possono essere:

1) DACL: Discretionary ACL → specifica i permessi
2) SACL: System ACL → specifica azioni di log da eseguire, in base agli accessi

* L'ACL di default dipende dall'ACCESS TOKEN, quindi dai permessi, del processo chiamante.

Ogni processo o thread ha i suoi SID (Security IDentifier) come utente e come gruppo. Tali SID costituiscono l'access token.
Quando un processo tenta di accedere ad un oggetto, si controlla nell'access token se ne ha i permessi; altrimenti si scandiscono le ACE dell'ACL: la prima ad esplicitare se ne ha i permessi oppure no è decisiva.

HARD DISKS

Dispositivi elettro-meccanici basati su tracce e una testina che si muove.
Ogni blocco su una traccia corrisponde ad un BLOCCO LOGICO di un file:
CORRISPONDENZA 1 ad 1.

* Ogni blocco è leggibile e scrivibile in OGNI ISTANTE DI TEMPO

Usure legate alle parti meccaniche, non tanti accessi multipli di una stessa traccia o blocco.

PARAMETRI PER LO SCHEDULING: 1) SEEK TIME: tempo di ricerca della traccia
2) RITARDO DI ROTAZIONE: tempo affinché il blocco capiti sotto la testina

- **SEEK TIME**:

n = tracce da attraversare
m = tempo per attraversare 1 traccia
s = tempo di attivazione testina

$$\Rightarrow \boxed{T_{SEEK} = n * m + s}$$

- **RITARDO DI ROTAZIONE**:

b = bytes da trasferire
N = numero di bytes per traccia
r = velocità di rotazione (rev. per min)

$$\Rightarrow \boxed{T_{ROTAZIONE} = \frac{b}{r \times N}}$$

I/O SCHEDULING

- **FCFS :**

First Come First Served → Tipico dei TERMINALI

Non produce STARVATION, ma non minimizza il SEEK TIME

lunghezza media di ricerca : $$\frac{\sum_{i}^{|\text{insieme dist}|} dist_i}{|\text{insieme dist}|}$$

- **SSTF (Shortest Service Time First)**

Viene servita prima la richieste di I/O che provoca il minor movimento della testina → MINIMIZZA IL SEEK TIME

Tuttavia : 1) non minimizza il tempo d'attesa medio;
2) può provocare STARVATION.

- **SCAN (elevator algorithm) :**

Il SEEK avviene in una data direzione fino al termine della traccia o fin quando non ci sono più richieste in quella direzione.

PROBLEMI : 1) sfavorevole alle aree attraversate più di recente (sfrutta poco la LOCALITÀ)
2) può causare STARVATION → insieme delle richieste è dinamico, possono arrivarne altre.

C-SCAN : utilizza la scansione in una sola direzione.

- **FSCAN :**

Use 2 code distinte : CODA DI IMMAGAZZINAMENTO + CODA DI SCHEDULING

* CODA DI IMMAGAZZINAMENTO è (FCFS) → evita starvation ; permette di mantenere costante la coda di scheduling, per evitare problemi dovuti all'arrivo di nuove richieste.
* CODA DI SCHEDULING use SSTF, SCAN o C-SCAN.

`int fsync (int fd)` → per flushare sul dispositivo i dati in buffer cache, per non avere problemi di consistenza, ma ha un costo.

5. PIPES

Permettono comunicazione di tipo STREAM I/O. Fanno parte del Virtual FS.

* Una volta lette, le info spariscono dalle pipe!

A livello di SO, non sono altro che BUFFER.

* Possono essere usate solo da PROCESSI RELAZIONATI! (non hanno NOME; bisogna ereditare i descrittori!)

```
int pipe (int fd[2])
```

fd[0] canale di LETTURA dalla pipe
fd[1] canali di SCRITTURA sulla pipe

Lo stream di una pipe viene chiuso quando la pipe non ha più dati da consegnare e tutti i canali di scrittura verso la pipe (fd[1]) vengono chiusi!

- DEADLOCKS:

Le pipes possono causare STALLI: per esempio, un thread tenta di leggere su fd[0], ma non ci sono dati ed il canale fd[1] è ANCORA APERTO.
Ma se è lui l'unico thread, se fa la read() non può fare anche la write!

→ STALLO, DEADLOCK!

⇒ Ogni thread deve chiudere i canali che non use!

NAMED PIPE (FIFO)

```
int mkfifo (char* name, int mode)
```

Ritorna un UNICO FILE DESCRIPTOR per accedere alla FIFO (differenza con le PIPES)
Bisogna aprirla con open() come se fosse un file regolare.
HA UN NOME → Possono comunicare anche PROCESSI NON RELAZIONATI

* L'apertura di una FIFO è BLOCCANTE: il processo che tenta di accedervi in lettura è bloccato fin quando un altro non vi accede in scrittura, e viceversa.
* ogni FIFO deve avere sia un lettore che uno scrittore.

MESSAGGI

Sono UNITÀ DI DATI scambiabili tra processi tramite I/O

- Sia la spedizione che la ricezione di un messaggio avvengono in modo ATOMICO → BLOCK I/O

MESSAGE QUEUE IN UNIX

Send() e Receive () SINCRONE → si può riutilizzare subito il buffer

Ogni messaggio DEVE avere almeno il campo TIPO, gestito dal Sistema Operativo

→ Supporto al MULTIPLEXING (più tipi).

```
int msgget (key_t key, int flag)
```

1) key: chiave identificativa della CODA
2) flag: O_CREAT | O_EXCL | 0666 (in <sys/ipc.h>, <sys/msg.h>)

- RESTITUISCE: DESCRITTORE d'accesso alla coda, ma nella TABELLA GLOBALE IPCT - Inter Process Communication Table

* Il descrittore è ereditato anche dai figli!

```
int msgctl (int ds_coda, int cmd, struct msqid_ds * buff)
```

cmd: IPC_RMID, IPC_STAT, IPC_SET

↓

Rimuove

```
int msgsnd (int ds_code, const void* buff, size_t nbyte, int flag)
```

2) buff : è una struct messaggio con campo TIPO + contenuto

4) flag : 0 → bloccante

1 = IPC_NOWAIT → non bloccante

```
int msgrcv (int ds_code, const void* buff, size_t nbyte, long type, int flag)
```

4) type : tipo dei messaggi cui siamo interessati.

MAILSLOT IN WINDOWS

Send() e Receive() sincrone o asincrone, bloccanti e non.

NON ho i TIPI → NO MULTIPLEXING

In una client-server communication, il client invia messaggi al MAILSLOT del server, ma deve indicare un SUO MAILSLOT dove ricevere le risposte.

6. Memory Management

Con BINDING A TEMPO D'ESECUZIONE, il mapping run-time tra INDIRIZZI LOGICI e FISICI (frame) è risolto da un dispositivo hardware, detto MMU.

- **BASELINE MMU:**

MMU := Memory Management Unit contiene un REGISTRO DI RILOCAZIONE contenente la base per l'offset (indirizzo logico); quindi:

BASE + OFFSET = IND. FISICO

↑ Reg. di Rilocazione MMU — ↑ Ind. Logico — ↑ RAM

Con indirizzi a x bit, si possono generare 2^x offset; ma, l'address space potrebbe essere $< 2^x$ → Può esserci SEGFAULT!

La MMU viene anche dotata di un REGISTRO LIMITE, che indica la fine dell'address space. Se si va oltre il limite, si genera una trap verso il SO.

- **PARTIZIONAMENTO STATICO:**

Memoria RAM suddivisa in partizioni FISSE. Ogni partizione → 1 PROCESSO!

PROBLEMI: 1) FRAMMENTAZIONE INTERNA

2) Non si possono ospitare processi con AS troppo grande per una partizione.

↓

Risolto parzialmente: molte piccole e poche grandi.

3) In assenza di swapping, GRADO DI MULTIPROGR. LIMITATO dal # di partizioni

OVERLAY: in caso di processi molto grandi, lo "swapping" di dati e istruzioni è compito del programmatore, tramite software (GESTORE DI OVERLAY)

ALLOCAZIONE DEI PROCESSI:

C'è sempre una partizione per il Software di SO.

- CODE MULTIPLE: Code per ogni partizione: riduce framm. interna
- CODA SINGOLA: riduce le prob. di avere partizioni inutilizzate → NO FRAMM. ESTERNA

- **PARTIZIONAMENTO DINAMICO**

Partizioni di NUMERO e TAGLIA VARIABILE. Partizione per 1 processo = taglia address space

* NON c'è FRAMMENTAZIONE INTERNA!

C'è FRAMMENTAZIONE ESTERNA ("buchi" di memoria libera)

↳ RICOMPATTAZIONE, ma: 1) richiede BINDING a tempo d'ESECUZIONE
2) Costosa!

- ALLOCAZIONE PROCESSI:
  - First Fit
  - Best Fit
  - Worst Fit

- SWAPPING: flessibile a tempo d'esecuzione, non a tempo di compil. /caricamento.

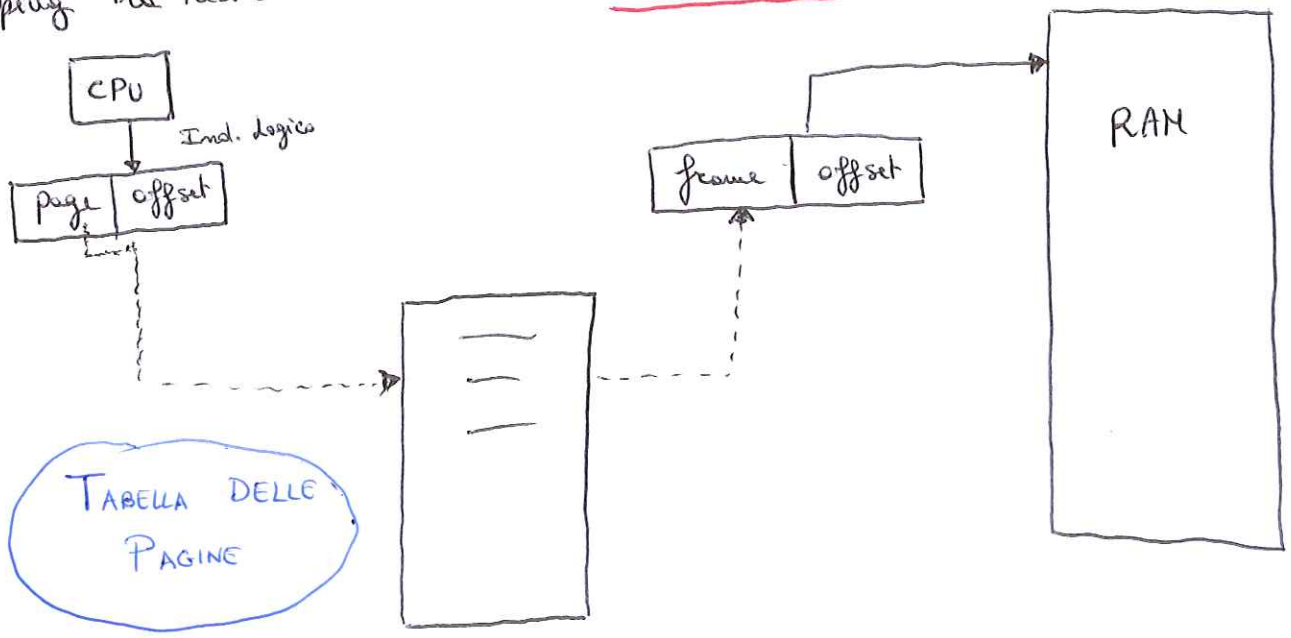
* VINCOLI: Richiede che non ci sia attività sull'address space
⇒ I/O ASINCRONO NON PERMESSO!
→ Buffer per l'I/O ASINCRONO allocati nella partizione del SO!

- **PAGINAZIONE**:

Address Space suddiviso in pagine di X bytes.
RAM suddivise in FRAMES di X bytes.
→ ALLOCAZIONE NON CONTIGUA → NON HO FRAMMENTAZIONE ESTERNA!

- Framm. Interna limitata in media a $\frac{X}{2}$ bytes.

Mapping tra indirizzi tramite una TABELLA DELLE PAGINE!



Ha una entry per ogni pagina, dove è mappato il frame corrispondente.

→ SUPPORTI HARDWARE per efficienza: CACHE TLB!

IND. LOGICO: | k bit | r bit | → 2^k pagine; 2^r dati per pagina

⇒ OFFSET ∈ $[0; 2^r - 1]$ → Non si può uscire fuori dal frame

↓

PROTEZIONE DI MEMORIA GARANTITA.

- **TLB:**

TLB := Translation Lookaside Buffer → è una CACHE HARDWARE

Mantiene associazioni tra pagine e frames del processo attualmente in esecuzione.

Si può avere un HIT o un MISS; in quest'ultimo caso è il FIRMWARE a consultare le tabelle delle pagine e ad aggiornare il TLB.

La TABELLA DELLE PAGINE è PER PROCESSO!

Se ne tiene traccia tramite reg. di processore CR3.

- **SEGMENTAZIONE:**

Address Space suddiviso in SEGMENTI: dati, testo, stack, ...

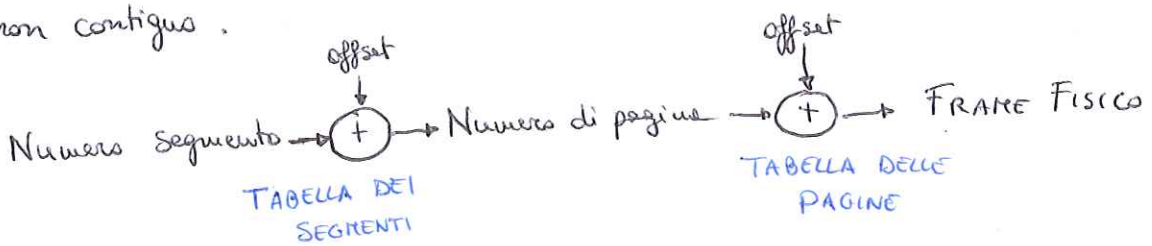
Sono aree di memoria VARIABILI → Si possono allocare in modo non contiguo.

C'è una TABELLA DEI SEGMENTI per risolvere il mapping

Ha il problema della FRAMMENTAZIONE ESTERNA!

SEGMENTAZIONE PAGINATA

Ogni SEGMENTO è visto come SUDDIVISO IN PAGINE, collocabili in RAM in modo non contiguo.



CONDIVISIONE

2 processi che hanno pagine con contenuto identico, vedono mappate le pagine allo STESSO FRAME in RAM.

Per esempio, con la (fork()) → TESTO = uguale
DATI = uguali, ma con protezione COPY ON WRITE

⇒ La prima volta che vengono scritti, si dedica un altro frame al processo.

La Tabella delle Pagine mantiene bit di controllo per le modalità d'accesso e protezione (come "copy on write") alle pagine.

- **PAGINE KERNEL:**

C'è un' UNICA ISTANZA di CODICE/DATI KERNEL.

Tutti i processi hanno pagine logiche mappate sugli indirizzi fisici del Kernel
→ SONO IN CONDIVISIONE TRA TUTTI I PROCESSI.

- PAGINE possono essere MAPPATE o NON e MATERIALIZZATE o non, una volta mappate.
  Struttura dati Kernel: VM AREA LIST → mantiene riferimenti sulle pagine mappate

MEMORIA VIRTUALE

Pagine MAPPATE, ma NON MATERIALIZZATE.
Dovuto a: 1) swapping (anche parziale)
2) pagine mai materializzate (ANONYMOUS → EMPTY ZERO MEMORY)

Gestione dei page fault tramite il PRESENCE BIT nella Tabella delle Pagine:
MAJOR FAULT: su pagine swapped-out → costoso (richiede I/O).

DIRTY BIT:
Bit di controllo nella Tabella delle Pagine che indica se una pagina è stata accedute in scrittura. In tal caso, al momento della sostituzione, la "VITTIMA" deve essere aggiornata sull' hard-drive.

ALGORITMI SELEZIONE VITTIMA

- **ALGORITMO OTTIMO:**
  Sostituire la pagina alla quale ci si riferirà DOPO PIÙ LUNGO TEMPO!
  Non implementabile! È un termine di paragone.

- **LRU - LEAST RECENTLY - USED:**
  Sostituire la pagina alla quale NON ci si riferisce DA PIÙ TEMPO → sfrutta la località temporale
  * difficile da implementare → troppo lavoro da dover mantenere

- **FIFO:**
  Viene sostituita la pagina che È IN RAM da PIÙ TEMPO
  * NON sfrutta la località! Ma è di facile implementazione

ANOMALIA DI BELADY

Data una sequenza di riferimenti casuali, aumentando il numero di frames in RAM, il numero di page fault AUMENTA (!) anziché diminuire.

* FIFO soffre dell'ANOMALIA DI BELADY.
* ALGORITMI A STACK, come (LRU), non soffrono delle Anomalie!

ALGORITMO DELL'OROLOGIO

Anche detto NOT RECENTLY USED.
Per ogni FRAME ho un REFERENCE BIT, messo a 1 ogni volta che il frame è acceduto.
C'è una LANCETTA DI SELEZIONE delle vittime, che sceglie la prima pagina quando passa, i bit a 1 vengono riportati a 0.
Logica con reference bit a zero. Se fa un giro completo, riparte. In quanto, Sfrutta bene la LOCALITÀ.
* Reference bit mantenuto nella Tabella delle Pagine (STICKY FLAG)
* La vittima non sempre è identificabile → con MULTITHREADING l'esecuzione è NON ATOMICA e ci può essere conflitto sul Reference Bit

→ PIÙ BIT: Si usa il ref. bit combinato con il DIRTY BIT:
(rb=0, db=0) → ideale!

7. SINCRONIZZAZIONE

- **SEZIONE CRITICA** : porzione di traccia dove:
  -) un processo / thread può leggere / scrivere dati condivisi con altri processi / thread
  -) la correttezza dipende dall' INTERLEAVING tra le tracce d'esecuzione

ALGORITMO DI DEKKER

```
Var turno : int;

PROCESSO X
While turno ≠ X do no-op;
  < sez. critica > ;
turno := Y;

PROCESSO Y
While turno ≠ Y do no-op;
< sez. critica > ;
turno := X;
```

C'è un flag "turno" che permette di accedere in ALTERNANZA STRETTA ;
tuttavia non c'è garanzia di PROGRESSO e la velocità è limitata dal processo più lento.
Si lavora A TURNI → NON VA BENE PER PROCESSI CONCORRENTI !

2° TENTATIVO

```
Var flag : array [1,n] of boolean;

PROCESSO X
While flag[Y] do no-op;
  flag[X] := TRUE;
< sez. critica >
  flag[X] := FALSE;
```

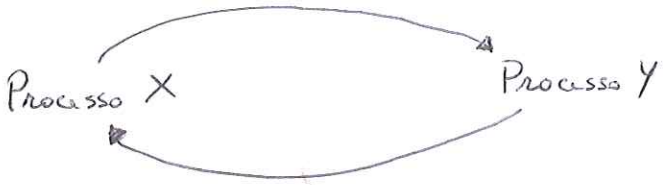
C'è un ARRAY di FLAG per ogni processo.
C'è garanzia di PROGRESSO, ma NON di MUTUA ESCLUSIONE dovuto all'interleaving nel mettere i flag a TRUE o FALSE e possibilità di essere descheduleti ritrovandosi poi in 2 in sezione critica.

3° TENTATIVO

Anticipa il proprio flag a TRUE.

```
var flag : array [1,n] of boolean;

PROCESSO X
flag [X] := TRUE;
While flag [Y] do no-op;
< sezione critica >
flag [X] := FALSE;
```



* C'è possibilità di DEADLOCK (entrambi i flag a TRUE) e quindi non c'è garanzia di ATTESA LIMITATA.

4° TENTATIVO

```
var flag : array [1,n] of boolean
  PROCESSO X
  flag [X] := TRUE;
  While flag [Y] do {
        flag [X] := FALSE;
        < pausa >
        flag [X] := TRUE;
  }
  < sez. critica >
  flag [X] := FALSE;
```

} Evita il DEADLOCK, lasciando l'accesso all'altro processo

* Può produrre STARVATION, quindi non dà garanzia di attesa limitata.

Algoritmo del Fornaio (Lamport)

L'idea è di assegnare dei numeri per prenotare un turno.

```
var choosing: array [1,n] of boolean;
    number : array [1,n] of int;

repeat {
    choosing [i] := TRUE;
    number [i] := < max of array number [] + 1 >; // mi metto in coda
    choosing [i] := FALSE;
    for j = 1 to n do {
        while choosing [j] do no-op;
        while number [j] ≠ 0 and (number [j], j) < (number [i], i) do no-op;
    }
    < sezione critica >
    number [i] = 0;
} until FALSE;
```

Se più thread hanno lo stesso numero di turno (buffer circolare), si dà precedenza al thread con indice più basso ($j < i$).

* while choosing [j] do no-op; → se qualcuno sta aspettando di scegliere il numero, aspetto che il suo numero sia riportato sull'array number [].

* Problemi: c'è BUSY WAITING → Spreco di CPU

Istruzioni RMW:

RMW := Read - Modify - Write. Fanno le 3 operazioni in modo ATOMICO.

```
function test_and_set ( var z: int ) : boolean;
{
  if (z == 0) {
      z := 1;
      test_and_set := TRUE;
  }
else test_and_set := FALSE;
}
```

```
var serratura : int;

PROCESSO X
while ! test_and_set (serratura) do no-op;
< sez. critica > ;
serratura := 0;
```

Fin quando serratura non diventa 0 non riusciamo ad accedere in sezione critica

- L'istruzione macchina più comunemente supportata è la CAS: Compare And Swap.

Pthread Spinlocks

- spinlock_t lock;
- int pthread_spin_init (pthread_spinlock_t * lock, int pshared);

  pshared può essere: 1) PTHREAD_PROCESS_SHARED → per thread di più processi
  2) PTHREAD_PROCESS_PRIVATE → per thread dello stesso processo

- pthread_spin_lock (&lock);
- pthread_spin_unlock (&lock);

* Causano ancora BUSY WAITING !

SEMAFORI

SEMAFORO := Struttura dati che include un intero ≥ 0, con associate 3 operazioni eseguite in MODO ATOMICO dal Kernel:

1) INIZIALIZZAZIONE;

2) **wait**: tenta di decrementare di 1 l'intero; nel caso in cui non sia disponibile, il processo va in stato di wait → NO busy waiting

3) **signal**: rilascio di un lock acquisito, incrementando di 1 il "distributore"; libera un processo che ha eseguito una wait bloccante.

PSEUDO-CODICE Produce - Consumer

SHARED: item buffer[N]; semaphore S = 1; counter;

PRODUCER

```
PRIVATE int in=0; item X;

Repeat {
    < produce X >

retry:
    wait(S)
    if counter = N {        # buffer pieno, rilascia il lock.
        signal(S);
        goto retry;
    }
    else {
        buffer[in] := X;
        in := (in+1) mod N;
        counter := counter + 1;
        signal(S);
    }

} until false
```

CONSUMER

```
PRIVATE int out=0; item Y;

Repeat {
    wait(S);
    if counter = 0 {        # Se non c'è nulla da leggere, rilascia il lock
        signal(S);
    }
    else {
        Y := buffer[out];
        out := (out+1) mod N;
        counter := counter - 1;
        signal(S);
        < consume Y >
    }
} until false
```

```
# include <sys/ipc.h>
# include <sys/shm.h>
```

```
int semget (key_t key, int size, int flag)
```

- Restituisce il descrittore nella IPCT GLOBALE! In realtà è un ARRAY SEMAFORICO

2) size: numero di "distributori" del semaforo

3) flag: IPC_CREAT | IPC_EXCL ...

```
int semctl (int ds_sem, int sem_num, int cmd, union semun arg)
```

2) sem_num: su quale "distributore" operare

3) cmd: IPC_RMID, IPC_STAT, IPC_SET, IPC_GETALL, IPC_SETALL, IPC_GETVAL, IPC_SETVAL

4)
```
union semun {
    int val;                    // se cmd = SETVAL
    struct semid_ds* buff;      // se cmd = IPC_STAT o IPC_SET
    ushort* array;              // se cmd = SETALL o GETALL
}
```

```
int semop (int ds_sem, struct sembuf oper[], int number)
```

2) array di struct sembuf (puntatore)

3) numero di elementi validi nell'array

```
struct sembuf {
    ushort sem_num;     // elemento nell'array semaforico
    short  sem_op;      // incremento o decremento
    short  sem_flg;     // IPC_NOWAIT - SEM_UNDO
}
```

pthread MUTEX

Può avere solo 2 stati: 1 = disponibile, 0 = occupato → È un SEMAFORO BINARIO

- pthread_mutex_t mutex;
- int pthread_mutex_init (pthread_mutex_t *mutex, const pthread_mutexattr_t* attr);
- pthread_mutex_lock (&mutex);
- pthread_mutex_unlock (&mutex);
- pthread_mutex_trylock (&mutex); → prova il lock, ma NON va in wait.