

VIRTUAL FILE SYSTEM

È l'insieme dei moduli di livello Kernel che permettono di effettuare operazioni di I/O, le quali avvengono tramite system call secondo uno schema omogeneo, per cui le chiamate di sistema sono le stesse indipendentemente da quale sia l'oggetto di I/O considerato. L'unica eccezione è data dall'operazione di instanziazione (=creazione) degli oggetti di I/O, la quale è data da system call ad hoc e differentiate.

PRINCIPALI OGGETTI DI I/O PRESENTI NEI SISTEMI OPERATIVI E GESTIBILI DAL VIRTUAL FILE SYSTEM:

- File
- Char device → permette alle applicazioni di emettere caratteri per esempio tramite una `printf()` o di acquisire sequenze di caratteri per esempio tramite una `scanf()`.
- Pipe
- FIFO
- Mailslot
- Socket → permette ai dati emessi in output di navigare in rete e ricevere dati dalla rete.
- Block device

Gli oggetti di I/O, essendo gestiti dal software, sono di fatto delle strutture dati.

Un'operazione di I/O coinvolge un'area di memoria in cui l'applicazione può scrivere alcuni dati destinati a un oggetto di I/O (output), oppure acquisire dati provenienti da un oggetto di I/O (input).

I servizi per eseguire operazioni di I/O sul virtual file system di un sistema operativo seguono due modelli: STREAM I/O e BLOCK I/O.

→ MODELLO STREAM I/O: nel momento in cui si effettua una lettura su un oggetto di I/O, è possibile che vengano consegnate da parte della struttura dati solo delle frazioni arbitrarie dei dati che precedentemente erano stati emessi in output verso l'oggetto. Di conseguenza, se si vuole riottenere in input tutti i dati che precedentemente sono stati emessi, potrebbero essere necessarie più letture.

→ MODELLO BLOCK I/O: nel momento in cui si effettua una lettura su un oggetto di

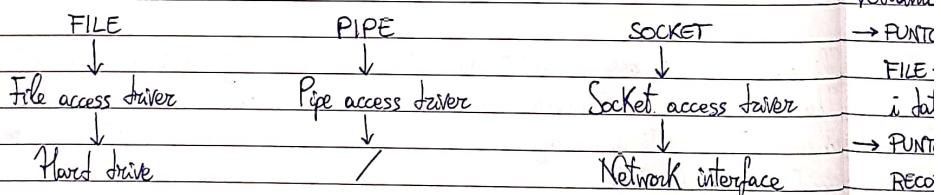
I/O, viene consegnata un'unità di dati, che sarebbe un blocco di dati precedentemente emesso in output verso l'oggetto tramite un'unica operazione, per cui non si ha alcun frammentamento delle informazioni.

Alcune strutture dati che implementano oggetti di I/O sono dette BACKEND: quando si esegue un'operazione di output verso di loro, il software del Kernel che le aggiorna chiamate anche un'interruzione con qualche dispositivo hardware (e.g. hard disk, inter facce di rete).

Driver:

Un'operazione di I/O a livello Kernel può essere differente e può richiedere l'esecuzione di moduli Kernel diversi a seconda del tipo di oggetto coinvolto, anche se viene chiamata la medesima system call per le varie strutture dati. Ciò è possibile grazie al DRIVER, di cui ne esiste uno differente per ogni tipologia di oggetto, implementa le relative operazioni di I/O e può includere o meno una rappresentazione di backend sui dispositivi hardware.

ESEMPIO:



Canale di I/O:

Tutte le operazioni di I/O vengono eseguite utilizzando un CANALE, che è un codice numerico (quindi un identificatore logico) o, in altri termini, una chiave di accesso a un'istanza di oggetto di I/O. In particolare, il Kernel è in grado di associare un canale (identificato tramite un descrittore/handle) all'istanza di oggetto di I/O corrispondente tramite la tabella di riferimento valida per processo, le cui entry, che sono appunto accessibili dai relativi descrittori/handle, contengono per esempio un puntoatore a uno specifico oggetto.

1 Sessione di I/O:

In realtà, il collegamento tra canale di I/O e oggetto di I/O non è diretto, bensì avviene mediante una struttura dati intermedia che si chiama SESSIONE. In particolare, quando avviene il setup di un canale di I/O, la entry corrispondente nella tabella di riportamenti porta a una sessione, all'interno della quale ci sono ulteriori informazioni utili per arrivare nella maniera corretta all'oggetto di I/O da coinvolgere.

Lo scopo della sessione è tener traccia di cosa è successo in uno specifico canale: di fatto, quando verranno inviate ulteriori chiamate di I/O su questo canale, esse avranno effetti diversi a seconda delle chiamate di I/O precedenti. Per esempio, la sessione tiene traccia di qual è il punto (byte) preciso dell'oggetto di I/O su cui effettuare la prossima operazione a valle del fatto che i byte precedenti contengono già informazioni e non devono essere sovrascritti.

È possibile che la chiusura di un canale comporti l'eliminazione della rispettiva sessione.

2 File system:

È un sottoinsieme dell'architettura del virtual file system che permette di gestire i file.

Vediamo che cos'è un file:

→ PUNTO DI VISTA DEL SISTEMA OPERATIVO:

FILE = unità minima informativa archiviabile; quando viene riaperto, ricompaiono tutti i dati scritti in precedenza.

→ PUNTO DI VISTA DELLE APPLICAZIONI:

RECORD = unità minima informativa utilizzabile quando vengono eseguite operazioni di I/O; è una frazione del contenuto del file e non può essere archiviato singolarmente.

Il file system mostra alle applicazioni un file come una SEQUENZA DI RECORD.

Il file system archivia e associa a ogni file un insieme di attributi (metadati), che sono informazioni che servono per gestire il contenuto del file e sono:

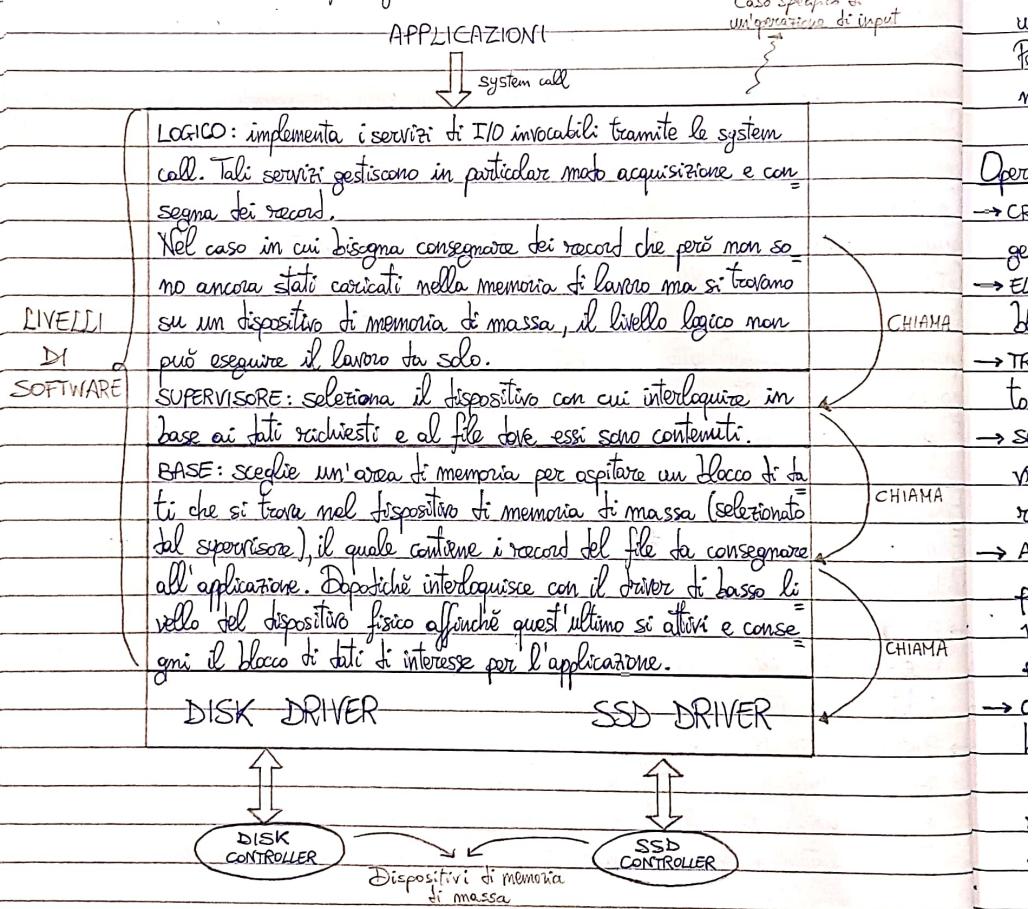
• NOME

• PROTEZIONE (si tratta di informazioni che indicano chi può far cosa sul contenuto in archivio)

• TIMESTAMP (è l'istante temporale dell'ultimo aggiornamento del file)

• ALTRO (in particolare, potrebbero esserci delle differenze tra sistemi operativi diversi)

Architettura di base di un file system:



→ Se l'operazione da effettuare è di output, il discorso è analogo: ciascun livello di software chiama quello immediatamente inferiore finché i dati emessi in output non vengono salvati su un dispositivo di memoria di massa.

In particolare, memorizzare le informazioni in tale dispositivo consente di mantenerle anche dopo che la macchina è andata in shut down (e quindi di mantenerle in maniera stabile).



→ BLOCCO DI DATI = unità minima informativa che può essere trasferita dalla memoria a un dispositivo fisico di storage (dispositivo di memoria di massa) e viceversa.

Perciò, le operazioni di I/O, che vengono invocate tramite i driver dei dispositivi, lavorano secondo uno schema a blocco (scrittura di un blocco / acquisizione di un blocco).

Operazioni base sui file:

→ CREAZIONE: allocazione di un RECORD DI SISTEMA (RS) che costituisce le informazioni per la gestione del file (e.g. gli attributi).

→ ELIMINAZIONE: deallocazione del record di sistema e cancellazione della memoria di tutti i blocchi di dati (che contenevano anche il RS stesso).

→ TRONCAMENTO: cancellazione della memoria dei soli blocchi di dati comprendenti il contenuto del file, senza quindi deallocare il record di sistema e gli attributi.

→ SCRITTURA/LETTURA DI RECORD: aggiornamento dell'indice (pointer al punto del file in cui dovrà essere eseguita la prossima operazione di input/output) valido per la sessione corrente.

→ APERTURA: setup di una nuova sessione e inizializzazione dell'indice di scrittura/lettura per questa sessione; inoltre, viene selezionata una entry libera della tabella di riferimenti valida per processo per arrivare, tramite un meccanismo di indirizzamento, alla sessione, e viene restituito all'applicazione il canale del canale.

→ CHIUSURA: rilascio della sessione e dell'indice di scrittura/lettura; avviene quando non si ha momentaneamente più interesse di lavorare sul file. Questa operazione NON deve comportare l'eliminazione del contenuto né tantomeno degli attributi del file: porta solo al rilascio del canale con conseguente liberazione della corrispondente entry della tabella valida per processo, che torna ricavabile per eventuali aperture successive di altri file.

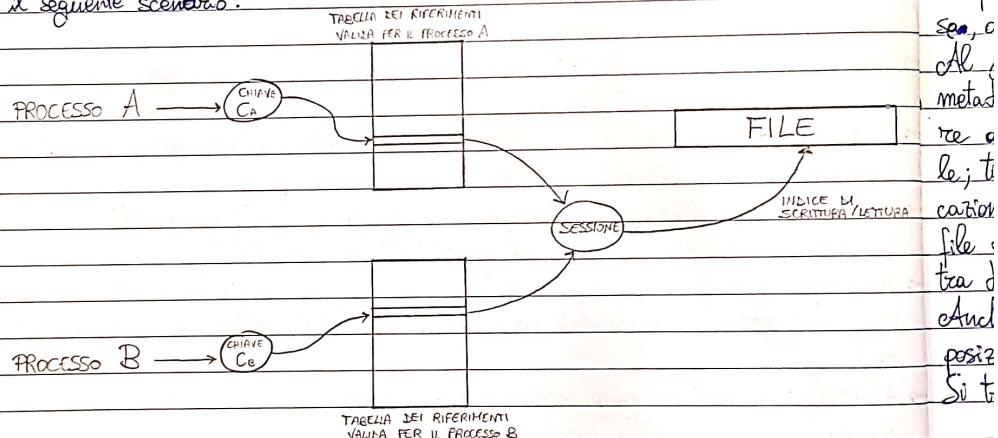
→ RIPOSIZIONAMENTO: aggiornamento dell'indice di scrittura/lettura senza cambiare il contenuto del file. Non è un'operazione di I/O, poiché non porta né alla scrittura né all'acquisizione di dati.

Indice di scrittura/lettura:

ie Non fa parte del record di sistema: infatti, non è un metadata associato al contenuto di

un file, bensì è un dato che indica solo la posizione precisa di questo contenuto su cui che de all'in uno specifico thread sta lavorando.

Inoltre, può essere condiviso da più processi. Di conseguenza, non fa parte della stessa immagine di processo, bensì dell'immagine di sessione. Si può quindi presentare il seguente scenario:



Le modalità di aggiornamento dell'indice di scrittura/lettura nell'operazione di riposizionamento dipendono dai METODI DI ACCESSO ai record di un file supportati da uno specifico file system. Esistono tre tipi di metodi di accesso fondamentali: SEQUENZIALE, SEQUENZIALE INDICIZZATO e DIRETTO.

Metodo di accesso sequenziale:

I record sono acceduti in sequenza: se si vuole leggere il record A, bisogna prima attraversare tutti i record compresi tra quello attualmente puntato dall'indice di scrittura/lettura e A. Inoltre, non è possibile spostare l'indice all'interno in modo arbitrario; il suo riposizionamento può avvenire solo all'inizio del file.

Si tratta dunque di un metodo non particolarmente flessibile ed è tipico di:

- FILE SEQUENZIALI, caratterizzati da record di taglia e struttura fisse.
- FILE A MUCCOLO, caratterizzati da record di taglia e struttura variabili; i metadati

ii che descrivono taglia e struttura di ogni record sono memorizzati in un HEADER posto all'inizio del record stesso.

Metodo di accesso sequenziale indirizzato:

È tipico di FILE SEQUENZIALI INDIRIZZATI, caratterizzati da record di taglia e struttura fisse, ordinati in base a un campo chiave.

Al file che contiene i record viene associato il FILE DI INDICI, che è caratterizzato da metadati (che, di fatto, sono degli indici) che descrivono le operazioni da eseguire per accedere ai record in modo diretto. Non è detto che esso indirizzi tutti i record del file principale; tuttavia, i suoi indici sono sicuramente disposti in ordine, il che permette alle applicazioni di leggere agevolmente un record anche se non è indirizzato, perché, tramite il file degli indici, possono determinare che questo record si trova in una zona del file compresa tra due specifiche chiavi.

Anche in questo metodo di accesso, l'unico modo per spostare l'indice all'indietro è ri-

posizionarlo all'inizio del file.

Si tratta di un metodo ancora oggi utilizzato per gestire informazioni a livello applicativo.

Metodo di accesso diretto:

Dopo un accesso all'i-esimo record, l'indice di scrittura/lettura assume il valore $i+1$.

Il riposizionamento (sia in avanti che all'indietro) dell'indice può avvenire in qualsiasi punto del file.

È il metodo attualmente adottato dal file system ed è tipico di:

- FILE DIRETTI, caratterizzati da record di taglia e struttura fisse e identificabili direttamente tramite l'indice della loro posizione all'interno del file.

Il contenuto dei file diretti non è d'interesse per il file system, bensì solo per le applicazioni.

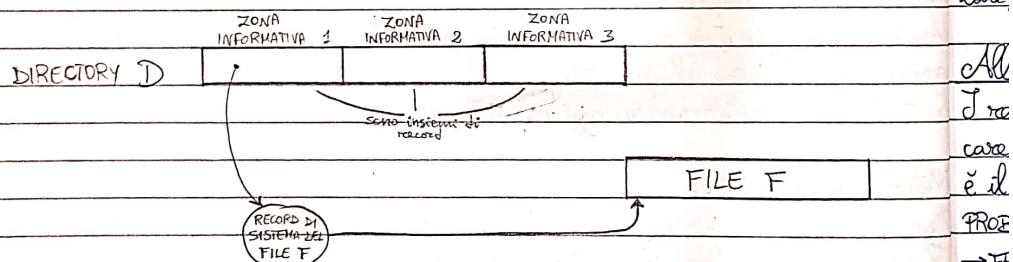
- FILE HASH, caratterizzati da record di taglia e struttura fisse con ordinamento per chiave: le chiavi dei singoli record vengono identificate con i corrispondenti indici mediante una funzione hash.

Directory:

I file all'interno del file system sono organizzati secondo una struttura gerarchica.

Per esempio, è possibile che i file F, F' si trovino nella directory /home, mentre il file F'' si trovi nella directory /home/francesco, che si trova all'interno ed è dipendente da /home.

Ma che cos'è una directory? È un file speciale D composto da una sequenza di record, i quali contengono le informazioni su quali sono i file che dipendono da D (può essere sia file classici che altre directory). In particolare, per identificare il file F all'interno di D, bisogna leggere il suo record di sistema che comprende i suoi attributi, tramite i quali è possibile risalire al suo contenuto. Un'altra informazione che in alcuni casi si può ricavare dai record che compongono D è il nome dei file che dipendono da D (vero nei sistemi UNIX).



Questa organizzazione è detta STRUTTURA DI DIRECTORY.

Collocazione di file nel dispositivo di memoria di massa:

Ciascun file è allocato in un dispositivo fisico di storage come un insieme di blocchi non necessariamente contigui che possono essere organizzati in tre modi differenti:

→ ORGANIZZAZIONE FISSA: si hanno file con record di taglia fissa. I record vengono collocati all'interno di un blocco B finché ce ne entriano, ed è possibile che alla fine di B avanzi uno spazio inutilizzato e insufficiente per accogliere un nuovo record. Questo fenomeno di spreco è detto FRAGMENTAZIONE INTERNA.

→ ORGANIZZAZIONE VARIABILE CON RIPORTO: si hanno record di taglia variabile che possono essere sovrapposti tra più blocchi nel caso in cui lo spazio che mette a dispo-

sizione il primo blocco non è sufficiente. In tal modo, la frammentazione interna viene eliminata.

→ ORGANIZZAZIONE VARIABILE SENZA RIPORTO: è uguale a quella variabile con riporto ma non offre la possibilità di suddividere un record tra più blocchi. Si rappresenta così la frammentazione interna.

Affinché il file system tenga traccia di quali blocchi sono liberi e quali invece sono occupati, si ricorre ad alcuni metadati di gestione, che sono:

→ LISTA LIBERA, in cui vengono elencati i blocchi liberi.

→ BIT MAP, che destina un bit a ogni blocco per indicare se esso è libero o meno.

Inoltre, il record di sistema di uno specifico file tiene traccia di quanti e quali blocchi sono allocati per quel file all'interno del dispositivo di memoria di massa. In particolare, esistono tre modalità principali di allocazione: CONTIGUA, A CATENA e INDICIZZATA.

Allocazione contigua:

I record di un file possono essere contenuti solo in blocchi contigui. È semplicissimo identificare tutti i blocchi utilizzati per il file: il record di sistema tiene traccia soltanto di qual è il blocco iniziale e di quanti blocchi sono coinvolti nell'allocazione.

PROBLEMATICA:

→ FRAMMENTAZIONE ESTERNA: possono esserci blocchi all'interno del dispositivo di memoria di massa che rimangono inutilizzati perché sono troppo sparsi e non possono essere accoppiati per ospitare file di certe dimensioni, data la necessità di allocare i record in blocchi non contigui.

→ SOLUZIONE: per ricompattare i blocchi sparsi, si attua la tecnica della DEFRAGIMENTAZIONE, che consiste nello slittamento di qualche file all'interno del dispositivo fisico di storage.

→ PROBLEMA DI SCELTA: se si hanno più buchi (= zone di blocchi contigui non ancora utilizzati) sufficientemente ampi per ospitare un nuovo file, quale conviene scegliere per effettuare l'allocazione? A tal proposito esistono due politiche principali:

• POLITICA BEST-FIT: tra i buchi in grado di ospitare il nuovo file, viene selezionato quello con meno blocchi.

• POLITICA WORST-FIT: tra i buchi in grado di ospitare il nuovo file, viene selezionato quello con più blocchi.

Allocazione a catena:

I record di un file possono essere contenuti anche in blocchi non contigui: ciascuno di questi blocchi è composto da una parte destinata a ospitare i record e un'altra parte che contiene dei metadati di posizionamento, tra cui per esempio un puntatore al blocco che comprende i record immediatamente successivi del file considerato. In tal modo, per ogni file si forma una vera e propria catena di blocchi all'interno del dispositivo di memoria di massa. Di conseguenza, per identificare tutti i blocchi utilizzati per un file, al record di sistema basta tenere traccia di qual è il blocco iniziale.

In taluni casi, potrebbe essere utile migrare dinamicamente il contenuto di un blocco in un'altra parte per effettuare una ricompattazione in modo tale che i record di uno specifico file si trovino tutti nella medesima zona del dispositivo di memoria di massa. Ad esempio, se il dispositivo è un hard disk e i blocchi relativi a un file si trovano sulla stessa traccia, il tempo per accedere a tutti i blocchi è sicuramente minore.

PROBLEMATICA:

→ COSTO DI ACCESSO AL RECORD POTENZIALMENTE ELEVATO: nel caso in cui si vuole lavorare con dei record che si trovano in fondo a un file, essi non sono direttamente intituiti all'interno del dispositivo di memoria di massa: inizialmente vengono trovati e trasferiti nella memoria di lavoro solo i record contenuti all'interno del primo blocco, il quale è l'unico a fornire informazioni sul posizionamento nella memoria fisica di storage del blocco immediatamente successivo, e così via, fino a rintracciare il blocco realmente d'interesse.

Questa sequenza di operazioni ha effettivamente un costo temporale elevato, soprattutto se viene utilizzato un metodo d'accesso flessibile come quello frettoloso.

Allocazione anticipata:

I record di un file possono essere contenuti anche in blocchi non contigui: il record di sistema tiene traccia dell'intera lista ordinata dei blocchi impiegati per quel determinato file. In tal modo, si può accedere a qualunque zona del file in modo diretto.

semplicemente consultando il relativo record di sistema.

Anche in questo caso potrebbe essere utile migrare dinamicamente il contenuto di un blocco da un'altra parte per effettuare una ricompattazione.

PROBLEMATICA:

→ SOTTOUTILIZZO DELLE INFORMAZIONI NEL RS: la tabella degli indici all'interno del record di sistema deve avere una taglia fissa, per cui deve disporre di tanto spazio per permettere l'allocazione di un eventuale file grande. Tuttavia, se il file ha dimensioni ridotte, necessita di pochi blocchi per essere ospitato, causando così uno spreco di risorse nel record di sistema.

→ SOLUZIONE: si ricorre all'INDICIZZAZIONE A LIVELLI MULTIPLI, per cui la tabella degli indici all'interno del record di sistema viene divisa in due sezioni:

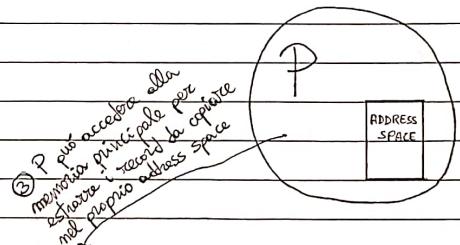
1) INDICI DIRETTI, che puntano ai blocchi che contengono i record del file.

2) INDICI INDIRETTI, che puntano ai blocchi che contengono a loro volta altri indici dei blocchi che ospitano i record del file.

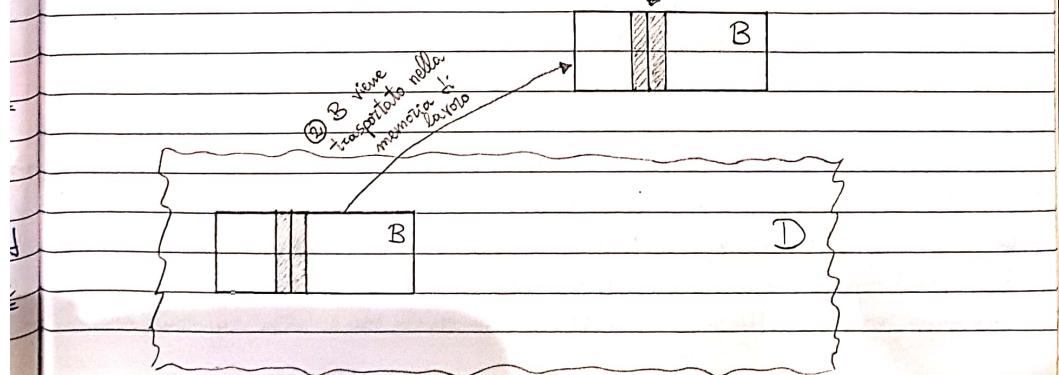
Buffer cache:

Consideriamo il seguente scenario:

① Il processo P chiama un servizio di I/O per poter leggere dei record di un file posto in un blocco B del dispositivo di memoria di massa D



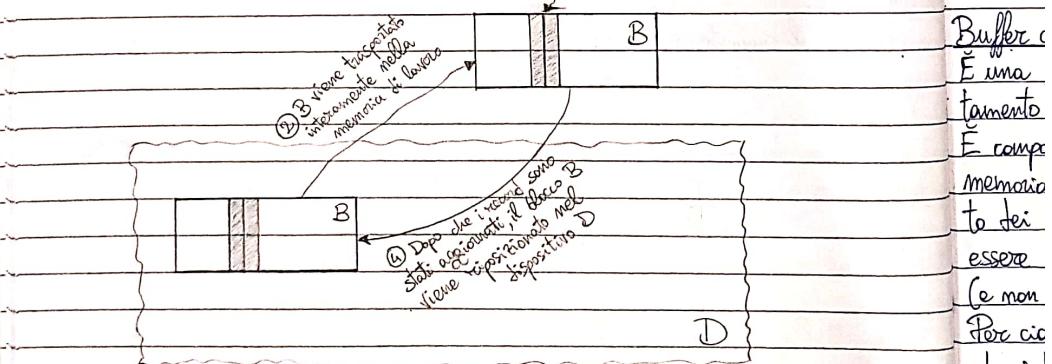
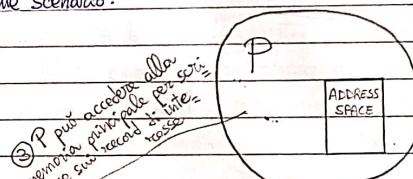
② B viene trasportato nella memoria di lavoro



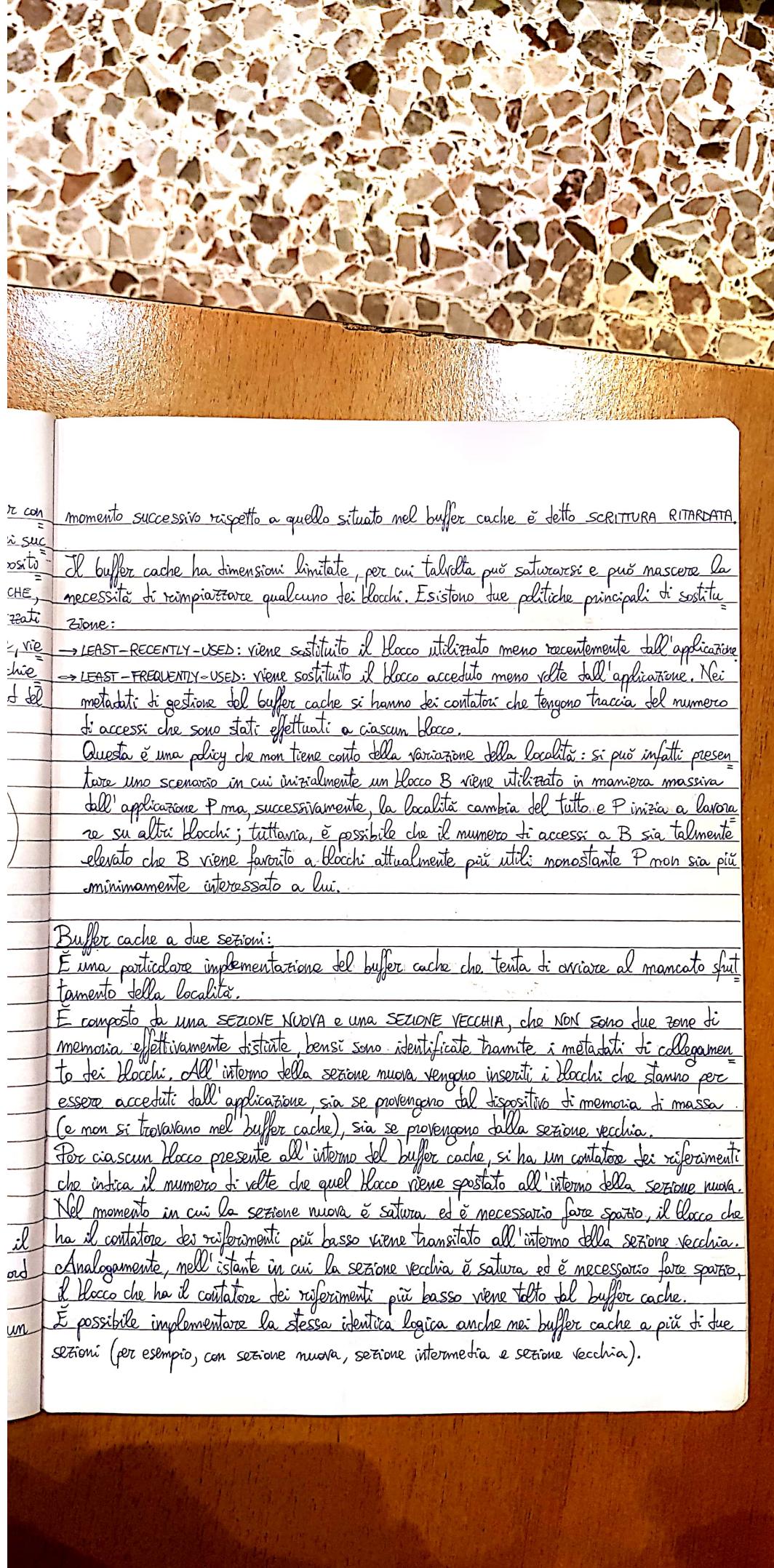
In un sistema del genere è utile mantenere il blocco B in memoria di lavoro per comunque sentire al processo P di accedere eventualmente ad altri record di B in operazioni successive (PRINCIPIO DI LOCALITÀ) senza necessità di estrarre nuovamente B nel dispositivo D. Per questo motivo, all'interno dei file system si ha il cosiddetto BUFFER CACHE, che è una zona di memoria utilizzata dal Kernel per mantenere temporaneamente bufferizzati i blocchi dei dispositivi. In particolare, nello schema mostrato nella pagina precedente, viene adottata una logica di LETTURA ANTICIPATA, in cui, oltre ai record effettivamente richiesti dal processo P, vengono anticipatamente messi a disposizione anche gli altri record del medesimo blocco.

D'altra parte, può verificarsi anche il seguente scenario:

- ① Il processo P chiama un servizio di I/O per poter scrivere su dei record di un file posti in un blocco B del dispositivo di memoria di massa D



Per gli stessi motivi di prima, anche in questo caso è utile mantenere temporaneamente il blocco B nel buffer cache, in modo tale che il processo P possa scrivere su diversi record di B in più tempi, e solo alla fine B viene riportato all'interno del dispositivo D.



Relazione fra operazioni di I/O e Swapping:

Sappiamo che lo swapping consiste nel trasferire l'address space di un dato processo dalla RAM all'area di swap o viceversa. Questa operazione viene gestita in modi diversi a seconda del sistema operativo:

→ UTILIZZO DEL FILE SYSTEM: gli address space, quando vengono portati fuori memoria, vengono scritti su un file (approccio Windows). La gestione dello swapping, tuttavia, potrebbe essere inefficiente: all'interno del file di swap, le varie sezioni di un address space possono trovarsi anche in punti molto lontani, per cui non si avrebbe località. Di conseguenza, si verifica un fenomeno detto di TRADEOFF SPAZIO-TEMPO che, in questo caso specifico, consiste nell'usare poco spazio di memoria penalizzando però le prestazioni temporali.

Per risolvere questa problematica, si può utilizzare l'approccio descritto qui in seguito.

→ UTILIZZO DELLA PARTIZIONE PRIVATA: gli address space, quando vengono portati fuori memoria, vengono trasferiti su un'apposita partizione di swap di un disco rigido o SSD (approccio UNIX). Tale partizione, tuttavia, è preservata, per cui non può essere adoperata per scopi diversi e, di conseguenza, si ha un'elevata frammentazione interna nel momento in cui viene sfruttata poco. D'altra parte, questo approccio garantisce una gestione ottimizzata dell'operazione di swapping in termini temporali: il posizionamento dei dati nei blocchi dell'area di swap non segue le regole di allocazione del file system, bensì delle regole di indirizzamento ottimizzate che permettono di identificare rapidamente tutti i blocchi relativi a uno specifico address space.

tate me
• Deduplic
cesso a

Vediamo
• Ogni f
• Il mo
sieme:
• Il met
• Il rec

fatto d
nel fi
• L'are
il num
muto

STRUZZI
Metadati (tipi
che indicano
vettore degli i
me dell'area
chi di dati s

File system UNIX:

Ne esistono diverse varianti con una base comune ma che differiscono per alcune peculiarità (proprie solo di alcune versioni), come:

- Unità di allocazione uguale a n blocchi, $n \geq 1$ (ciascun indice punta a n blocchi contigui).

- journaling, che è un meccanismo che permette al file system di ricostruire in maniera corretta il suo contenuto anche a seguito di anomalie (e.g. spegnimento nel momento in cui alcune informazioni aggiornate nel buffer cache non sono ancora state riportate).

tate nel dispositivo di memoria di massa).

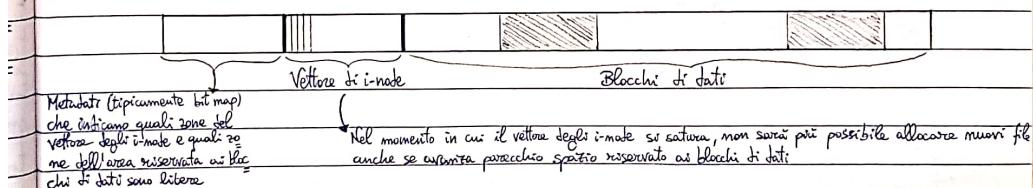
- Deduplicazione, che consiste nel salvare in un unico blocco il contenuto di più file identici.
- ACL (Access Control List), che comprende delle facility orientate alla sicurezza nell'accesso ai file.

Vediamo ora le caratteristiche di base comuni a tutti i file system UNIX:

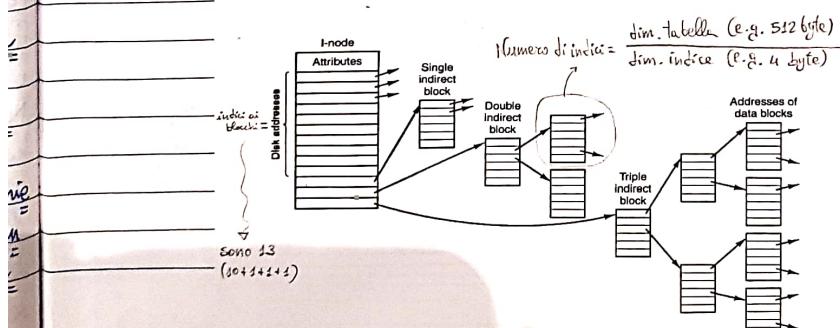
- Ogni file è trattato come una semplice sequenza di byte (un record = un byte).
- Il modello dei servizi sui file è lo stream I/O: è possibile lavorare solo su un sottoinsieme arbitrario di byte all'interno di un file.
- Il metodo di accesso ai file è diretto.
- Il record di sistema di un file viene denominato i-node (indexing-mode), a conferma del fatto che, tra le varie informazioni di cui tiene traccia, figurano gli indici dei blocchi nel dispositivo di memoria di massa che contengono quel file.
- L'archivio ha un'organizzazione gerarchica in cui le directory contengono il nome e il numero di i-node di ciascun file. In particolare, il nome di un file NON è contenuto nel suo i-node.

→ Può anche essere partitionato per contenere più file system

STRUTTURA DI UN DISPOSITIVO DI MEMORIA DI MASSA:



Struttura di un i-node



Attributi basici mantenuti all'interno di un i-node:

{-, d, b, c, p}	Indicano la tipologia di file (normale, directory, block-device, character-device, pipe).
UID (User ID - 2/4 byte)	Indicano i codici numerici del proprietario e del gruppo di utenti a cui appartiene.
GID (Group ID - 2/4 byte)	Sono tre triplettre di bit; in ciascuna tripletta vengono specificati rispettivamente i permessi di lettura, scrittura ed esecuzione del file.
Rwx rwx rwx	PRIMA TRIPPLETTA: si riferisce al proprietario. SECONDA TRIPPLETTA: si riferisce agli utenti dello stesso gruppo del proprietario. TERZA TRIPPLETTA: si riferisce a tutti gli altri utenti.
SUID (Set UID - 1 bit)	Sono bit che, se impostati a 1, qualunque utente che abbia il permesso di mandare il file in esecuzione potrà farlo per conto del proprietario / gruppo del proprietario (IDENTIFICAZIONE DINAMICA).
SGID (Set GID - 1 bit)	Se la tipologia di file è directory, rimuove la possibilità di cancellare i file in quella directory se non si è il proprietario.
Sticky (1 bit)	Se la tipologia di file è directory, rimuove la possibilità di cancellare i file in quella directory se non si è il proprietario.

Associazione user_id - username:

All'interno del Kernel, quanto un processo è attivo, viene associato a uno user_id e a un group_id, che sono codici numerici che indicano rispettivamente per quale utente e quale gruppo questa applicazione sta operando, e che quindi fanno parte dei metadati di gestione per il processo. L'associazione di uno username allo user_id è un fatto mercantile applicativo su come vengono organizzate le informazioni di gestione delle utenze. In particolare, in un sistema operativo UNIX, le utenze sono identificate tramite il file password che si trova nella directory /etc, e i gruppi sono identificati tramite il file group che è situato anch'esso nella directory /etc.

ACL (Access Control List):

È una lista dei permessi di accesso e gestione dei file molto più dettagliata delle triplette rwx e può avere taglia variabile. Viene implementata tramite un FILE SHADOW (file ombra) nel seguente modo:

Dato un qualunque file F e il suo i-node X, viene instantiato un altro file F' (detto file shadow) con il suo i-node X' (detto i-node shadow). All'interno di F' sono specificate tutte le informazioni che discriminano a grana fine ciò che gli utenti del sistema possono fare su F. Per relazionare i file F, F', all'interno dell'i-node X c'è un campo di indicizzazione che identifica X'.

COMANDI SHELL:

- `getfacl [nome_file]` → avvia un'applicazione che legge il contenuto del file shadow associato a uno specifico file.
- `setfacl -b [nome_file]` → aggiorna il file shadow rimuovendo le regole sulla sicurezza e sul controllo degli accessi.
- `setfacl -m [ug: id: permessi] [nome_file]` → aggiorna il file shadow aggiungendo nuove regole sulla sicurezza e sul controllo degli accessi.
- `chmod 000 [nome_file]` → modifica la tripletta di bit rwx, cancellando o ripristinando alcuni permessi senza andare a resettare la ACL.

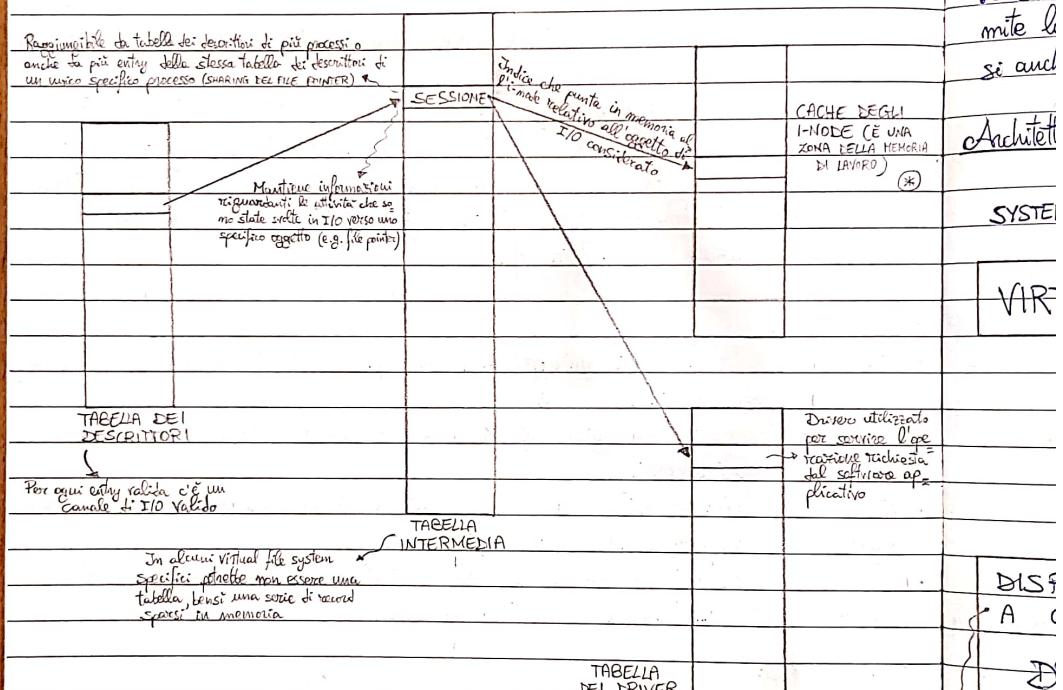
ESEMPIO DI ACL:

OWNER	<code>user :: rwx-</code>	
NAMED USER	<code>user: Jane: rwx-</code>	
OWNER'S GROUP	<code>group :: r--</code>	
MASK	<code>mask :: rwx-</code>	J permessi scritti nella maschera sovrappongono i permessi corrispondenti scritti in altre righe
OTHER	<code>other :: ---</code>	

Altri sulle SHELL:

- `touch [nome_file]` → crea il contenitore di un nuovo file ma senza alcun contenuto.
- `ls -i [nome_file]` → mostra l'indice dell'i-node relativo a uno specifico file.

Architettura interna di un virtual file system UNIX:



④ Esistono due categorie di i-node all'interno della relativa cache:

- **STATICI:** vengono associati ai file (ma, in generale, a tutti gli oggetti di I/O che fanno parte dell'archivio) e vengono riportati su hard-drive allo spegnimento del sistema.
- **DINAMICI:** vengono associati ai dispositivi (che sono gestiti come i file) e possono essere rimossi allo spegnimento del sistema poiché non sono necessariamente i-node di oggetti in archivio.

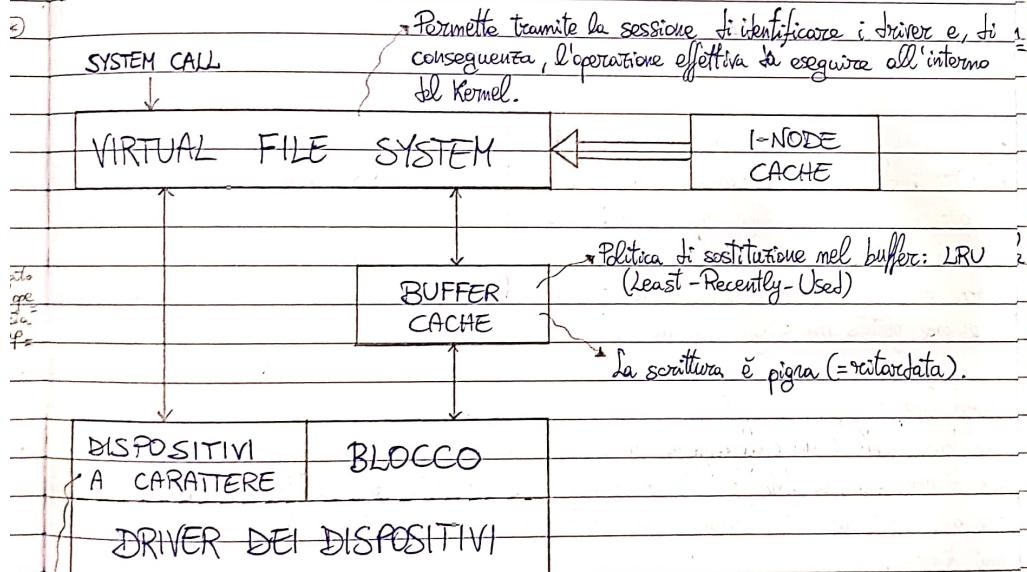
NOTE:

- Un file può essere gestito tramite più sessioni, ciascuna delle quali può lavorare su un

punto diverso del file.

- Le sessioni contengono anche le informazioni sul come può essere gestito il file tramite la relativa chiave di accesso (e.g. sola lettura / lettura+scrittura). Può trattarsi anche di impostazioni più stringenti rispetto ai permessi di accesso.

Architettura di I/O in sistemi UNIX:



Emettono/acquisiscono flussi di informazioni (non blocchi).

In particolare, quando emettono dati verso il sistema operativo, questi vengono prima eventualmente bufferizzati tramite il virtual file system e poi consegnati alle applicazioni.

Qui, in ogni caso, non c'è alcuna cache: poiché si tratta di un flusso di dati, una volta che ha raggiunto la destinazione, viene consumato e non è più disponibile.

Un discorso analogo vale per le operazioni di output verso i dispositivi a carattere.

Descrittori dei canali speciali:

- Descrittore 0 → associato al canale standard input → usato dalla funzione scanf()
- Descrittore 1 → associato al canale standard output → usato dalla funzione printf()
- Descrittore 2 → associato al canale standard error → è relativo all'output standard che vengono messi i dati riguardanti gli errori che si sono verificati durante l'esecuzione dell'applicazione. Questo canale di I/O può coincidere con quello dato dello standard output (che tipicamente è il terminale).

di lettura standard

Syn
Da
F e
sec
ver
Nel
in
sist

NOTE:

- Anche i canali 0, 1, 2 possono essere chiusi attraverso la system call close().
- Tutti i descrittori dei canali vengono ereditati dal processo figlio generato da una fork() (per cui si ha sharing del file pointer) e vengono conservati anche nel caso in cui viene chiamata una exec (a meno che non venga usato il flag O_CLOEXEC per l'apertura del file o viene chiamata la system call fcntl(), che dà indicazioni su come deve essere gestito un singolo canale).

com
• l
• l
• s
• o
• ro

Hard link e reference counter:

Un hard link è un collegamento tra un'istanza di una directory D (che contiene anche il nome di uno specifico file) e l'i-node di quel file dipendente da D. Per ogni file si possono avere più hard link (talla stessa directory e/o da più directory differenti) e, quindi, più nomi diversi.

Un file F viene dismesso col relativo i-node nel momento in cui non è più accessibile, ovvero quando si verificano le seguenti due condizioni:

- 1) Non esiste più alcun hard link per F.
- 2) Non esiste alcuna sessione attiva che porta a F.

Per verificare questa situazione, all'interno dell'i-node di ciascun file sono mantenuti due contatori (REFERENCE COUNTER) per tenere traccia di quanti hard link e quante sessioni confluono attualmente a quello specifico file.

Lo stesso meccanismo viene adottato per le sessioni, ciascuna delle quali contiene un reference counter per tenere traccia di quanti canali di I/O sono attivi per lei. Non appena il counter assume il valore zero, la relativa sessione viene dismessa.

Symbolic link (o soft link):

Dato un file F, un suo symbolic link è un altro file F' il cui contenuto è il nome di F e i cui metadati contengono i collegamenti che conducono a F. Si tratta quindi di un secondo modo per rendere possibile l'accesso a uno specifico file tramite più nomi diversi (in questo caso, quello di F e quello di F').

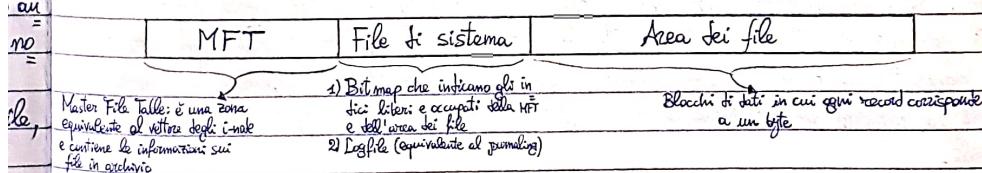
Nel momento in cui F viene dismesso, F' continua comunque a esistere ma non è più in grado di accedere al contenuto di F. Per questa ragione, i soft link non vengono considerati nei reference counter.

COMANDI SHELL:

- link [old_path] [new_path] → crea un hard link.
- ln -s [old_path] [new_path] → crea un soft link.
- stat [nome_file] → chiede al virtual file system informazioni riguardanti uno specifico file o uno specifico file system.
- rm [nome_file] → rimuove un hard link di uno specifico file.

File system Windows (NTFS):

STRUTTURA DI UN DISPOSITIVO DI MEMORIA DI MASSA:



Ogni hard drive può essere partizionato in volumi che contengono file system indipendenti l'uno dall'altro: a differenza di UNIX, non c'è alcuna relazione gerarchica tra i vari file system.

Inoltre, le informazioni all'interno degli hard drive sono organizzate in CLUSTER, che sono insiemi più o meno grandi di blocchi, ciascuno dei quali costituisce una vera e propria entità che contiene tutti i dati di un particolare file. La clusterizzazione serve per esempio ad alleggerire la rappresentazione delle bit map, in cui ogni bit non è più rela-

tivo a un singolo blocco, bensì a un intero cluster.

La dimensione di un cluster è customizzata e può variare tra 512 e 64.000 byte.

Arch

Attributi basici mantenuti all'interno di un elemento della MFT:

→ INFORMAZIONI STANDARD: timestamp, data e ora dell'ultimo accesso, ecc.

→ DESCRITTORE DI SICUREZZA: permessi di accesso

→ NOME

→ DATI / PUNTATORI AI BLOCCI DOVE SONO MANTENUTI I DATI

→ LISTA DI ATTRIBUTI: informazioni che indicano dove localizzare nella MFT eventuali attributi che non rientrano in questo stesso elemento (è praticamente un puntatore)

Permet
corret
su ogg

Permet
tivi f
di re
ria s
speci

Di fatto, a ogni file può corrispondere anche più di un elemento della MFT. In tal caso, questi elementi sono collegati tramite puntatori formando così una vera e propria lista.

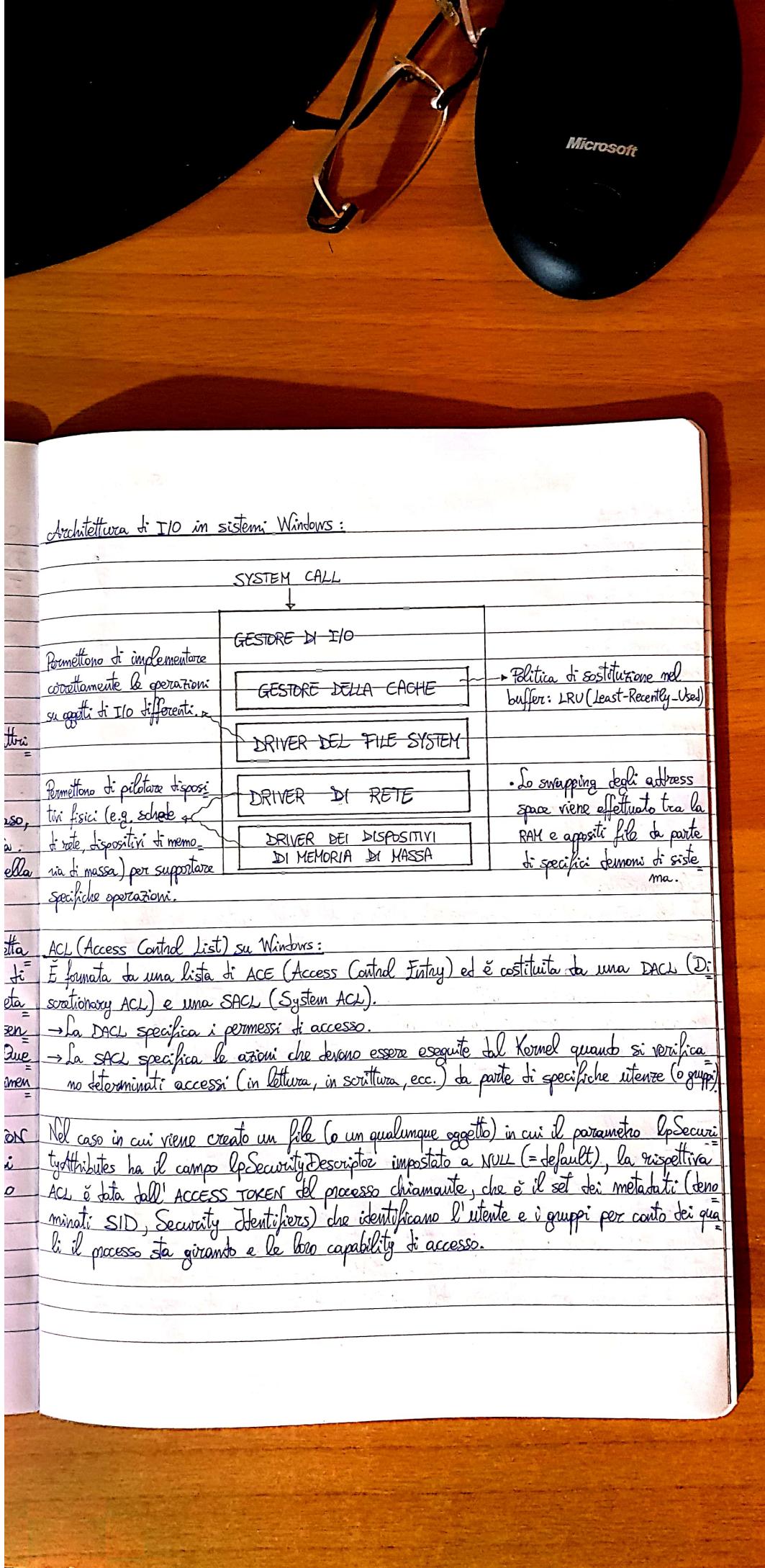
Se occupano posizioni contigue all'interno della MFT, allora possono essere caricati nella memoria cache di lavoro tutti simultaneamente.

Se un certo file ha dimensioni abbastanza ridotte, il suo contenuto può essere scritto direttamente in uno o più elementi della MFT. In tal caso, si tratta di un FILE IMMEDIATO: di fatto, i suoi dati possono essere portati subito in memoria di lavoro assieme ai suoi metadati, per cui risultano immediatamente disponibili per le operazioni delle applicazioni, senza necessità di dover caricare ulteriori blocchi del dispositivo di memoria di massa. Questa è la ragione per cui Windows è più adatto di UNIX a gestire tanti file di piccole dimensioni.

In ogni caso, è possibile mantenere anche file molto grandi che tipicamente sono NON IMMEDIATI: all'interno dei corrispondenti elementi della MFT, al posto del contenuto, si hanno dei puntatori ai blocchi all'interno del dispositivo di memoria di massa dove sono mantenuti i dati.

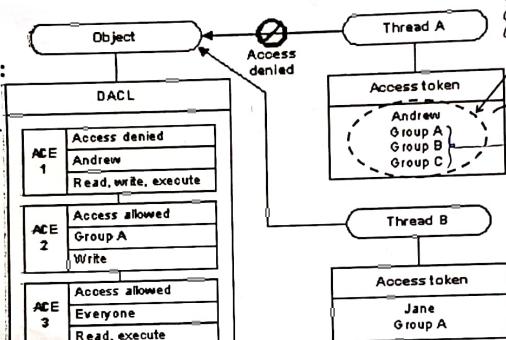
Nel
typ
AC
m
li





Un esempio

Di fatto, l'utente Andrew appartiene all'insieme EVERYONE: per stabilire in maniera univoca i suoi permessi di accesso, viene fatta solo la ACE che compare per prima all'interno della DACL tra tutte quelle che possono essere ricollegate ad Andrew.



Se l'access token offre dei permessi di accesso incondizionato all'oggetto, non c'è bisogno di scorrere la lista delle ACE.

Insieme dei gruppi per conto dei quali il thread A sta grida

Sono i gruppi di appartenenza di Andrew

Config
→ Bu

PR

→ E

F

File registry:

Il file system principale, in cui risiede l'istanza del Kernel, contiene dei file chiamati REGISTRY, che sono dei file di configurazione per l'operatività del sistema e contengono le utenze, i SID e la loro associazione agli username.

Non è possibile aprire il registry tramite una chiamata a CreateFile() perché non permette gli accessi da più sessioni. La loro apertura può dunque avvenire o attraverso una specifica interfaccia di sistema (e.g. l'editor di registro Regedit) o mediante il comando di shell wmic useraccount get name, sid .

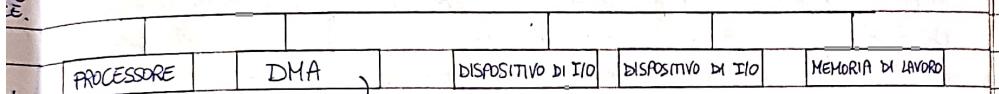
I/O scheduling:

Consiste nella pianificazione dell'uso dei dispositivi di I/O nel momento in cui più applicazioni ne richiedono concorrentemente il servizio.

In dispositivi come il terminale, l'I/O scheduling viene semplicemente attuato secondo una politica FCFS, perché gli utenti interattivi si aspettano di osservare in output (o fornire in input) dei dati in ordine. Tuttavia, esistono contesti in cui politiche di scheduling più articolate sono fondamentali per l'efficienza dell'architettura di I/O (e.g. disco rigido a rotazione).

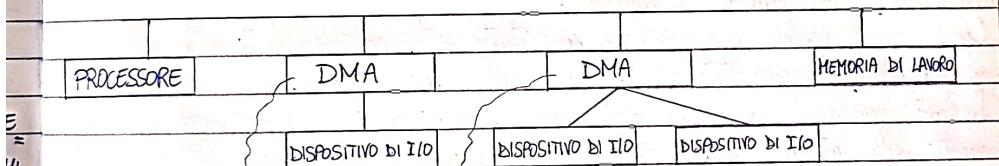
Configurazioni hardware:

→ BUS SINGOLO - DMA SEPARATO



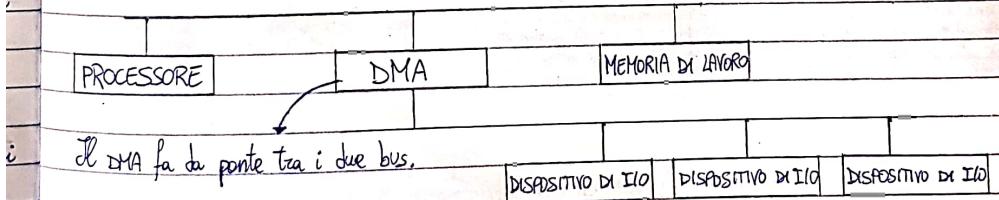
DIRECT MEMORY ACCESS: controller che permette al processore di accedere in memoria generando un solo interrupt per l'intero blocco di dati da leggere/scrivere. Affinché ciò sia possibile, il DMA deve pilotare tutti i dispositivi di I/O per i loro scambi di dati con la memoria.

→ BUS SINGOLO - DMA E I/O INTEGRATI



Ci sono più DMA che pilotano solo i dispositivi di I/O a loro connessi.

→ BUS DI I/O



Il DMA fa da ponte tra i due bus.

Con questa organizzazione, il bus del processore è molto più performante.

Dispositivi di memoria di massa:

→ **HARD DISK**: sono dispositivi elettrico-mecanici composti da tracce e da una testina che si muove sul raggio del disco per selezionare una specifica traccia. Le tracce ospitano blocchi di dati, ciascuno dei quali corrisponde a un BLOCCO LOGICO, che è l'unità minima all'interno della quale vengono inseriti i dati di un file (c'è una corrispondenza uno a uno tra blocco fisico e blocco logico). È comunque possibile che più blocchi logici costituiscano un cluster.

→ **SOLID STATE DRIVE (SSD)**: sono dispositivi meramente elettronici della classe NVM (Non Volatile Memory). Hanno una velocità più elevata nello scambio di dati con la memoria di lavoro rispetto agli hard disk. Ogni blocco fisico contiene un insieme di blocchi logici; la clusterizzazione avviene già nella costruzione dell' SSD.

Hard disk (fischi magnetici e rotazione):

Ogni blocco è leggibile e scrivibile a ogni istante di tempo indipendentemente dalle operazioni precedenti che sono state effettuate su quello specifico blocco.

Le parti meccaniche degli hard disk possono usurarsi col tempo e per mezzo degli spostamenti della testina, ma non direttamente a causa delle operazioni di lettura e scrittura sui blocchi.

RITARDO TOTALE PER IL SERVIZIO DI UN THREAD DA PARTE DELL'HARD DISK =

$$= \text{ATTESA DEL TURNO} + \text{ATTESA DEL CANALE} + \text{SEEK TIME} + \text{RITARDO ROTAZIONALE}$$

Tempo necessario affinché il thread sia scelto a discapito di altri per essere servito dall'hard disk

Tempo necessario affinché i bus e l'eventuale bridge siano liberi per permettere al hard disk di scambiarsi dati con la memoria di lavoro

Tempo di ricerca della traccia da parte della testina

Tempo necessario per accedere a uno specifico settore all'interno della traccia

TALI TEMPI DI ATTESA DIPENDONO DALLA POLITICA DI SCHEDULING

$$\rightarrow \text{SEEK TIME} = T_{\text{seek}} = m \cdot n + s$$

m = tempo necessario alla testina per spostarsi di una traccia (0,1 - 0,3 ms)

n = numero di tracce da attraversare

s = tempo di avvio della testina (3-20 ms)

$$\rightarrow \text{RITARDO DI ROTAZIONE} = T_{\text{rotaz}} = \frac{b}{r \cdot N}$$

b = numero di byte da trasferire

r = velocità di rotazione (300 - 7200 rpm)

↳ Giri al minuto

N = numero di byte per traccia

Analizziamo ora le principali politiche di scheduling per gli hard disk.

Scheduling FCFS (First Come First Served):

Le richieste di I/O vengono servite nell'ordine di arrivo, per cui non si ha starvation.

Tuttavia, il seek time non viene minimizzato, poiché non viene considerato lo stato attuale
nella ricerca del dispositivo (in particolare la posizione della testina) quando deve essere scelta la prossima richiesta da accogliere: può capitare spesso che la testina vada da un'estremità all'altra del disco tra un'operazione e l'altra.

ESEMPIO:

Traccia iniziale: 100

Sequenza delle tracce accedute: 55 - 58 - 39 - 18 - 90 - 160 - 150 - 38 - 184

Distanze:

45 - 3 - 19 - 21 - 72 - 70 - 10 - 112 - 16

$$\text{Lunghezza media di ricerca} = \frac{\sum_{i=1}^n \text{DIST}_i}{|\text{INSIEME DIST}|} = 55,3$$

Scheduling SSTF (Shortest Service Time First):

Si dà priorità alla richiesta di I/O che produce il minor movimento della testina. Perciò, il tempo medio di attesa del turno non viene minimizzato e si può avere starvation a discapito delle tracce lontane dalla testina nel momento in cui arrivano dinamicamente nuove richieste di accesso a delle tracce più vicine.

ESEMPIO:

Traccia iniziale: 100

Sequenza delle tracce coinvolte: 55 - 58 - 39 - 18 - 90 - 160 - 150 - 38 - 184

Ricordino in base al seek time: 90 - 58 - 55 - 39 - 38 - 18 - 150 - 160 - 184

Distanze:

10 - 32 - 3 - 16 - 1 - 20 - 132 - 10 - 24

$$\text{Lunghezza media di ricerca} = \frac{\sum_{i=1}^n \text{DIST}_i}{|\text{INSIEME DIST}|} = 27,5$$

Scheduling SCAN (algoritmo dell'ascensore):

Il movimento della testina avviene in una data direzione, fino al termine delle tracce o finché non ci sono più richieste in quella direzione; dopodiché la testina torna indietro.

Questa policy è sfavorevole all'area attraversata più di recente: se una richiesta su una data traccia arriva leggermente in ritardo, viene penalizzata. Si ha quindi un ridotto shuttleamento della località e la possibilità di avere starvation a discapito delle tracce che non si trovano lungo la direzione di spostamento della testina.

Esiste una variante che si chiama C-SCAN, in cui la testina si muove sempre nella stessa direzione: al termine della scansione, non si sposta gradualmente all'indietro, bensì viene riportata direttamente all'estremo opposto.

ESEMPIO:

Traccia iniziale: 100

Direzione iniziale: CRESCENTE

Sequenza delle tracce coinvolte:

55 - 58 - 39 - 18 - 90 - 160 - 150 - 38 - 18

Rioridino in base alla direzione di seek:

150 - 160 - 180 - 90 - 58 - 55 - 39 - 38 - 18

Distanze:

50 - 10 - 24 - 94 - 32 - 3 - 16 - 1 - 20

| INSIEME DIST |

$$\text{Lunghezza media di ricerca} = \frac{\sum_{i=1}^n \text{DIST}_i}{|\text{INSIEME DIST}|} = 27,8$$

Scheduling FSCAN:

Vengono utilizzate due code distinte per la gestione delle richieste: la coda di IMMAGAZINAMENTO e la coda di SCHEDULING. Inizialmente le richieste vengono inserite all'interno della coda di immagazzinamento e, in un secondo momento, vengono spostate a gruppi in quella di scheduling in base all'ordine di arrivo. In questa seconda coda, le richieste vengono ricontestate seguendo l'algoritmo SSTF, SCAN oppure C-SCAN.

Questa politica ha in sé un giusto compromesso tra il problema della starvation e quello del seek time, ed è tuttora utilizzata in alcuni sistemi operativi moderni (per esempio in LINUX).

Dispositivi SSD:

• Politica di Scheduling: FCFS

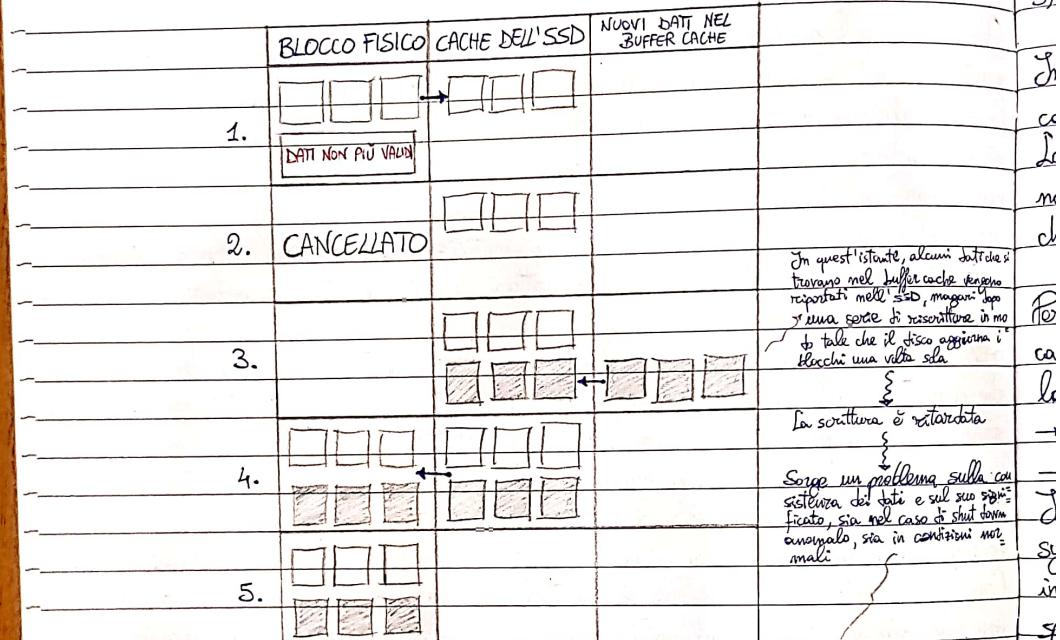
Ogni blocco è leggibile e scrivibile, ma può essere scritto solo dopo essere stato cancellato; questo implica che i blocchi hanno una macchina a stati integrata che deve trovarsi nello stato di cancellazione per consentire un'operazione di scrittura. A ogni cancellazione, il blocco si deteriora per effetto della transizione che la macchina a stati integrata deve compiere. Di conseguenza, esiste un limite di numero di cancellazioni e riscrittura (orientativamente 100 000) prima che il blocco sia considerato inutilizzabile. Inoltre, c'è uno sbilanciamento tra la velocità di lettura e quella di scrittura perché, appunto, un'operazione di scrittura richiede in più la fase di cancellazione.

All'interno di un SSD si ha un'implementazione degli algoritmi di gestione del DEROUTING, che consistono nell'effettuare il più possibile le scritture sui blocchi inutilizzati del dispositivo, con lo scopo di ammortizzare i costi e gli effetti negativi delle cancellazioni. Perciò, può capitare spesso che gli aggiornamenti di determinati blocchi logici comportino delle scritte su blocchi fisici dell'SSD di volta in volta diversi. Per questo motivo, l'SSD è dotato di una tabella che tiene traccia di quali blocchi fisici contengono determinati blocchi logici.

Ciascun elemento di questa tabella è una tupla <Blocco Logico, Blocco Fisico, PAGINA>, dove una pagina corrisponde all'unità minima su cui è possibile effettuare le operazioni di scrittura/lettura, e rappresenta la posizione all'interno del blocco fisico nella quale è situato il blocco logico. Tuttavia, l'unità minima su cui può essere attuata una cancellazione rimane l'intero blocco fisico.

Le scritture in punti sempre diversi dei medesimi blocchi logici portano col tempo ad avere, per ogni blocco fisico, molte pagine non cancellate che contengono informazioni non più valide e, di fatto, rappresentano uno spreco di spazio. Per affrontare tale situazione senza uscire dall'SSD, la cancellazione avviene tramite particolari algoritmi chiamati GARBAGE COLLECTION, che implementano la tecnica dell'OVER PROVISIONING: nel momento in cui la percentuale delle pagine disponibili per le scritture successive all'interno di uno specifico blocco B scende al di sotto di una certa soglia, B viene interamente cancellato, riportando da un'altra parte il contenuto delle sole pagine di B che sono ancora valide. Queste ultime vengono talvolta ricompattate con dei blocchi logici nuovi, come mostrato nello schema raffigurato nella pagina seguente.

(*) e.g. nella cache interna dell'SSD



Semantica della consistenza sul file system:

- SEMANTICA UNIX: ogni scrittura di dati è direttamente visibile a ogni lettura eseguita successivamente per riportare i blocchi sul dispositivo di memoria di massa, poiché sia le scritture che le letture avvengono sul buffer cache. Tale semantica è supportata anche da Windows.
- SEMANTICA DELLA SESSIONE: ogni scrittura di dati avviene in un'istanza privata per la sessione di lavoro e sarà visibile solo dopo che il file sarà stato chiuso. Tale semantica è supportata dai file system distribuiti, i quali permettono la memorizzazione dei file in dispositivi di archiviazione distribuiti in una rete informatica.

Problematiche di consistenza nei file system:

Supponiamo di avere il seguente scenario:

- 1) Avviene una scrittura sul buffer cache.
- 2) Si verifica un'anomalia (per esempio uno spegnimento improvviso del sistema).

3) Dopo che eventualmente il sistema è stato ripristinato, avviene una lettura sul buffer cache per riportare i dati sul dispositivo di memoria di massa.

In questo caso, i dati che vengono letti non sono quelli aggiornati, per cui si ha un'inconsistenza.

Lo stesso discorso vale per l'aggiornamento dei metadati di un file, che su UNIX porta a un'operazione di scrittura su un i-node all'interno della cache degli i-node prima che esso venga riconsegnato al dispositivo di memoria di massa.

Per ricostruire una consistenza all'interno del file system nel momento in cui si verificano delle anomalie, si possono sfruttare delle facility che permettono di fare un'analisi del lo stato del file system per cercare di identificare situazioni di irregolarità:

→ SU UNIX: fsck

→ SU WINDOWS: scandisk

In particolare, queste applicazioni verificano tramite dei metadati se la struttura del file system è ancora corretta: se sì, allora sarà possibile perlomeno ricominciare a lavorare in maniera corretta sul file system stesso, per cui i suoi contenuti costituiscono un assetto secondario. Più nello specifico, fsck e scandisk effettuano i seguenti controlli:

- Quali sono i blocchi in uso e in quali file si trovano; salvo scenari di deduplicazione, appartengono sperabilmente a un solo file.
- Quante volte (auspicabilmente una o nessuna) i blocchi liberi compaiono nella lista libera, che è un metadato che indica quali blocchi possono essere utilizzati in futuro per registrare informazioni relative ai file.
- Se effettivamente tutti i blocchi o sono in uso (e quanti sono impiegati per un file) o compaiono nella lista libera.

ESEMPIO:

Block number	Blocks in use	Free blocks
0 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15	1 1 0 1 0 1 1 1 0 0 1 1 1 0 0	0 0 1 0 0 0 0 1 1 0 0 0 1 1
(a)		
Block number	Blocks in use	Free blocks
0 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15	1 1 0 1 0 1 1 1 0 0 1 1 1 0 0	0 0 0 1 0 0 0 0 1 1 0 0 0 1 1
(b)		
Block number	Blocks in use	Free blocks
0 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15	1 1 0 1 0 1 1 1 0 0 1 1 1 0 0	0 0 1 0 2 0 0 0 0 1 1 0 0 0 1 1
(c)		
Block number	Blocks in use	Free blocks
0 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15	1 1 0 1 0 1 1 1 0 0 1 1 1 0 0	0 0 1 0 1 0 0 0 0 1 1 0 0 0 1 1
(d)		

a) Situazione consistente

b) Il blocco 2 non è in nessun file né nella lista libera: aggiungilo alla lista libera

c) Il blocco 4 compare due volte nella lista libera: togli un'occorrenza

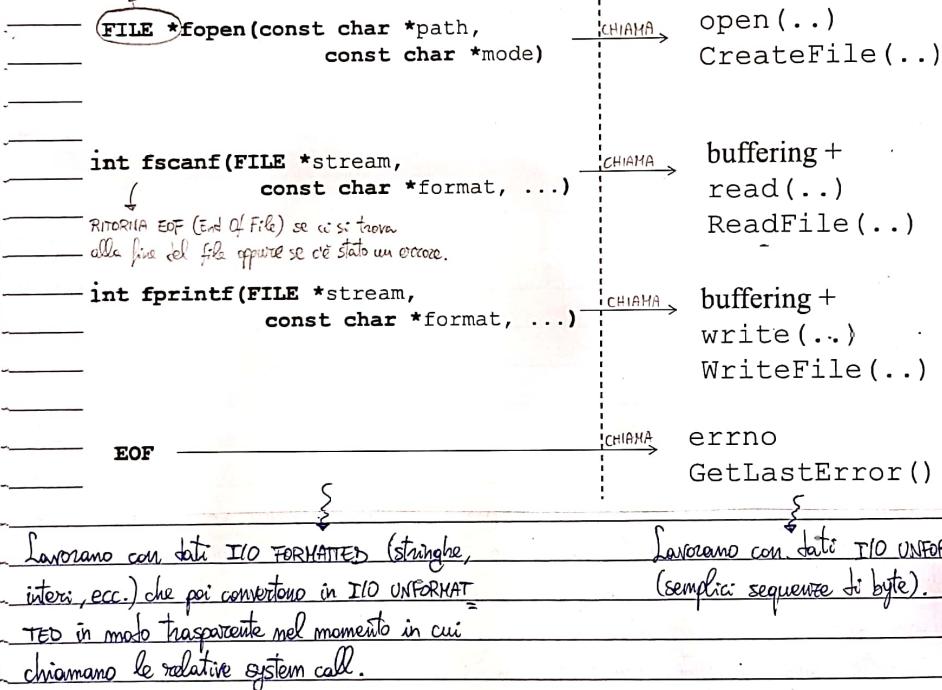
d) Il blocco 5 compare in due file: duplica il blocco e sostituisco in uno dei file (rimedio parziale!!!)

perché è probabile che i contenuti dei due file non siano consistenti rispetto alle operazioni che erano state effettuate in precedenza

Puntatore alla struttura dati che racorda l'operatività della libreria <stdio.h> rispetto alle varie API di sistema.

ESEMPIO: `stdout`, che punta alla struttura dati per la gestione dell'output verso il canale standard d'output (con descrivente = 1).

stdio.h vs file system API



COMUNICAZIONE TRA THREAD/PROCESSI

Può avvenire tramite particolari oggetti di I/O (PIPE, named PIPE) oppure attraverso un meccanismo di scambio di messaggi non implementato direttamente col virtual file system.

PIPE:

È un oggetto di I/O con la finalità di comunicare qualcosa secondo un modello stream, per cui è possibile leggere e scrivere flussi di byte come se scorressero all'interno di un tubo. La comunicazione è MONODIREZIONALE: è possibile immettere dati solo a un estremo del tubo ed estrarli dall'altro estremo.

Una volta lette, le informazioni spariscano dalla PIPE e non possono più rappresentarsi, a meno che non vengano scritte nuovamente.

Dal punto di vista del sistema operativo, la PIPE è una struttura dati implementata a livello Kernel, che consiste in un buffer e dei metadati di gestione. Dunque, quello che realmente accade è: un processo scrive dei dati sul buffer e un altro processo (o anche se stesso) va a leggerli, rendendoli non più disponibili (per cui potranno essere sovrascritti).

Gli unici processi che possono comunicare tramite una PIPE sono quelli RELAZIONATI tra loro (l'uno deve essere il parent o comunque un capostipite dell'altro). Per bypassare questo vincolo, si utilizzano le NAMED PIPE (denominate anche FIFO).

PIPE in sistemi UNIX:

Vengono utilizzate tramite due descrittori distinti, di cui uno permette di leggere dati dal buffer e l'altro consente di inserire dati sul buffer.

La situazione per cui non è più possibile leggere nuovi dati dal flusso è nota come FINE DELLO STREAM e si verifica se sono soddisfatte due condizioni:

- 1) Nel buffer non ci sono più dati disponibili per lettura successive.
 - 2) Tutte le sessioni in scrittura su quel buffer sono chiuse.
- In questo caso, una chiamata alla funzione `read()` restituisce zero.

Nell'ipotesi in cui invece tutti i canali di lettura sono chiusi (per cui non c'è alcun lettore che possa acquisire dati dalla PIPE), una chiamata alla funzione `write()` genera una segnalazione denominata SIGPIPE o Broken-pipe con conseguente chiusura dell'ap-

plificazione: si tratta di una condizione di ANOMALIA SEVERA dovuta a un utilizzo di risorse del tutto inutile.

Stalli (o deadlock - punti morti):

Supponiamo che un processo P abbia i canali aperti sia in lettura sia in scrittura su una PIPE, che sia l'unico processo attivo, e che lavori solamente con un thread T.

Supponiamo inoltre che il buffer non contenga dati validi. La PIPE comunque rimane attiva perché esiste un canale di scrittura aperto. Tuttavia, nel caso in cui T chiama una `read()` sulla PIPE, il thread entra nello stato di blocco in attesa che qualcun altro immetta nuovi dati nel buffer, ma ciò non accadrà mai perché non esistono altri thread attivi: T rimarrà nello stato `Hocked` per sempre, generando così uno stallone.

Questo scenario di deadlock si verifica anche se si hanno più processi (o thread) con i canali aperti sia in lettura che in scrittura su una PIPE attualmente senza dati validi, e tutti loro entrano nello stato di blocco in attesa che vengano scritti dei nuovi dati sul buffer.

Onde evitare queste situazioni, bisogna essere particolarmente attenti a dismettere i canali che non servono: se un thread è interessato ad acquisire dati da una PIPE, è buona norma mantenere aperto per lui solo il canale in lettura.

Anche un'operazione di scrittura su una PIPE può essere bloccante: il thread chiamante viene messo in attesa nel momento in cui cerca di inserire nella PIPE dei dati ma il buffer è saturo. La traccia esce dallo stato di blocco quando qualcun altro invoca una lettura sulla PIPE, consumandone il contenuto. Per questo motivo, uno stallone si verifica anche se si hanno esclusivamente thread "scrittori" con qualche canale di lettura aperto.

Utilizzo delle PIPE da shell:

Quando sulla shell viene lanciato il comando `A | B`, viene di fatto instantiata una PIPE attraverso cui B prende in input il valore di ritorno di A. Questa operazione avviene precisamente nel seguente modo:

1) La shell crea una PIPE ottenendo i canali `fd[0], fd[1]` rispettivamente per le letture e le scritture.

- 2) Poiché A, B sono comandi esterni, la shell esegue due fork() per creare due processi figli (P_A, P_B), i quali a loro volta effettuano una exec per poter implementare i corrispondenti comandi. A questo punto, per le caratteristiche delle funzioni fork() ed exec, i child hanno entrambi ereditato fd[0], fd[1].
- 3) P_A dismette il canale standard di output, mentre P_B chiude il canale standard di input.
 - 4) P_A , tramite una dup(), ridirige il canale standard di output su fd[1], mentre P_B , sempre mediante una dup(), ridirige il canale standard di input su fd[0].
 - 5) P_A esegue il comando A e immette il suo output nel buffer.
 - 6) P_B estrae i dati dalla PIPE, li acquisisce in input ed esegue il comando B.

Named PIPE (o FIFO) in sistemi UNIX:

Due processi P, P' non relativati non possono comunicare tra loro tramite una PIPE, a meno che non sia coinvolto un loro capostipite in comune, per due motivi:

- P non è autorizzato a generare P' mediante una o più fork(), per cui P' non ha potuto ereditare da P i descrittori fd[0], fd[1] della PIPE (e viceversa).
- La PIPE è un oggetto di I/O senza nome; per cui P, P' non possono richiederne i descrittori nel corso della loro esecuzione.

Per risolvere questa problematica, si ricorre alla named PIPE (o FIFO), che è una PIPE con nome a cui è possibile accedere sia in lettura che in scrittura tramite un unico descrittore. All'interno del driver viene quindi discriminato se i dati devono essere aggiunti al flusso o estrarri: dal buffer.

Inoltre, per operare su una FIFO, basta chiamare una open() come se fosse un file. Tuttavia, l'apertura di una FIFO è bloccante: se un processo tenta di aprirla in lettura, viene messo in attesa fino a quando un altro non la apre in scrittura (e viceversa).

Esistono due modi per inhibire questo comportamento:

- Aggiungere il flag O_NONBLOCK al valore del parametro mode della system call open().
- Aprire la FIFO sia in scrittura che in lettura tramite un'unica chiamata a open().

Anche nel caso delle FIFO, se tutti i canali di lettura sono chiusi, un'invocazione di write() porta a una condizione di anomalia severa e genera il segnale SIGPIPE.

PIPE in sistemi Windows:

- Hanno perfettamente le stesse caratteristiche delle PIPE nei sistemi UNIX. C'è solo un dettaglio da tenere in considerazione: su Windows, la creazione di nuovi processi non avviene tramite fork(), bensì mediante CreateProcess(), per cui i child ereditano gli handle verso una PIPE non direttamente, ma con alcuni passaggi supplementari.

Named PIPE in sistemi Windows:

- Anche se hanno a grandi linee le stesse caratteristiche delle named PIPE nei sistemi UNIX, ma con qualche differenza in più:
 - La rimozione di una named PIPE dal sistema avviene automaticamente nel momento in cui viene chiuso l'ultimo handle (su UNIX, per dismettere una FIFO, bisogna chiamare il servizio unlink()).

→ Una named PIPE su Windows ha caratteristiche più flessibili:

- 1) Può essere sia unidirezionale con un unico buffer, sia bidirezionale con due buffer (uno di input e uno di output).
- 2) Può trasmettere dati sia secondo il modello stream, sia a blocchi.
- 3) Può avere più di un'istanza: se possibile, ne viene creata una nuova quando tutte le altre sono già impegnate. In particolare, una named PIPE può essere messa nello stato "impegnato" tramite la funzione ConnectNamedPipe(), che mette il thread chiamante nello stato di blocco in attesa che qualche altro thread apra la stessa named PIPE per mettersi in comunicazione con lui.

Messaggi nei sistemi operativi:

- Sono unità di dati che possono essere scambiate tra due o più processi tramite servizi di I/O. In quanto unità di dati, sono indivisibili: sia nella spedizione, sia nella ricezione, un messaggio è intero o non viene scambiato proprio. Al massimo, il destinatario può solo fornire una frazione mentre lo legge.

Primitive di spedizione:

Questi due parametri sono il minimo indispensabile

MODELLO GENERALE DI API:

Send(destinatione, messaggio)

→ PUNTATORE AL BUFFER IN CUI SI TROVA IL CONTENUTO DEL MESSAGGIO

SPEDIZIONE	BLOCCANTE	NON BLOCCANTE
SINCRONA	Il chiamante non riprende il controllo finché il buffer non può essere sovrascritto senza sovraccaricare dati al messaggio, ovvero finché il sottosistema di spedizione non acquisisce correttamente i dati per conservarne una copia a livello Kernel.	Il buffer utilizzato per spedire il messaggio deve essere rimesso immediatamente a disposizione per scrittura successive, senza mandare il chiamante nello stato di blocco. Affinché ciò sia possibile, il sistema di gestione dei messaggi deve essere in grado di sfruttare determinate risorse. Se e.g. l'utente fa uso del Kernel
SINCRONA RANDEZ-VOUS	Il chiamante non riprende il controllo finché il messaggio non viene effettivamente ricevuto (uso di acknowledgement).	questa possibilità viene meno, la spedizione fallisce.
ASINCRONA		È possibile riutilizzare immediatamente il buffer contenente il messaggio, per cui i dati in corso di spedizione possono essere sovrascritti.

Primitive di ricezione:

MODELLO GENERALE DI API:

Receive (sorgente, messaggio)

Questi due parametri sono il minimo indispensabile

→ PUNTATORE AL BUFFER IN CUI DOVRÀ ESSERE CONSEGNATO IL MESSAGGIO

Il servizio di ricezione di messaggi può essere BLOCCANTE o NON BLOCCANTE:

→ SERVIZIO BLOCCANTE: il chiamante entra in attesa finché non arriva almeno un messaggio

da una specifica sorgente,

→ SERVIZIO NON BLOCCANTE: il chiamante non entra mai in attesa: in particolare, nel caso in cui non sia disponibile alcun messaggio da parte di una specifica sorgente, il controllo viene restituito subito al chiamante con un avviso di mancata ricezione di dati.

Sia il servizio bloccante che quello non bloccante possono essere o sincroni o asincroni:

• In una ricezione sincrona, se il messaggio è disponibile, viene immediatamente conse-

gnato all'interno del buffer.

- In una ricezione asincrona, quando il chiamante riprende il controllo, è possibile che gli venga comunicato che la consegna è avvenuta correttamente ma il messaggio verrà scritto all'interno del buffer successivamente.

Identificazione di sorgenti e destinazioni:

Una tecnica di indirizzamento dei messaggi può essere DIRETTA o INDIRETTA:

- INDIRIZZAMENTO DIRETTO: le sorgenti e le destinazioni coincidono con identificatori di processi (possono essere pid/handle oppure identificativi all'interno del subsistema di messaggistica).

ESEMPIO:

Send (P, message)

↓
PROCESSO CHE DOVRÀ RICEVERE IL MESSAGGIO

Receive (Q, message)

↓
PROCESSO DA CUI DOVRÀ PROVENIRE IL MESSAGGIO

- INDIRIZZAMENTO INDIRETTO: le sorgenti e le destinazioni non coincidono con identificatori di processi, bensì con identificatori di MAILBOX (o core di MESSAGGI / DEPOSITI DI MESSAGGI).

Una mailbox è un'entità rappresentata tramite una struttura dati che può essere associata o meno a uno specifico processo; tale associazione esiste se la mailbox viene disposta nel momento in cui il processo corrispondente termina l'esecuzione.

Alternativamente, un deposito di messaggi può essere associato a riferimenti che i processi possono avere verso di lui. È il caso di Windows, in cui i processi possono avere un handle per immettere/estraere dati da una determinata mailbox, la quale viene rimossa nel momento in cui tutte le maniglie valide sono dismesse.

Esiste una terza possibilità, in cui un deposito di messaggi può esistere indipendentemente dai processi e dai loro riferimenti verso di lui. Questo è uno scenario tipico di UNIX.

ESEMPIO:

Send (A, message)

↓
MAILBOX IN CUI IMMETTERE IL MESSAGGIO

Receive (A, message)

↓
MAILBOX DA CUI ESTRARRE IL MESSAGGIO

L'indirizzamento indiretto, rispetto a quello diretto, permette, grazie alla mailbox, di instaurare in più le seguenti relazioni:

- UNO-A-MOLTI: un processo può inviare messaggi che possono essere letti da chiunque appartenga a un particolare insieme di processi.

MOLTI-A-MOLTI: tanti processi possono inviare messaggi che possono essere letti da chiunque appartenga a un particolare insieme di processi.

Buffering di messaggi:

Vediamo più nel dettaglio dove e come il sottosistema di messaggistica può mantenere i messaggi:

- IN MEMORIA KERNEL → nel momento in cui un thread T spedisce un messaggio M, non è necessario impostare una Receive() affinché T esca dallo stato di blocco, perché il sottosistema di messaggistica copia M dal buffer di livello user al buffer di livello Kernel.

La capacità di bufferizzazione può essere:

→ NULLA: può essere in transito al massimo un messaggio per volta.

→ LIMITATA

→ IDEALMENTE ILLIMITATA

E' possibile definire un tempo massimo per la bufferizzazione di un messaggio, il quale viene buttato quando scatta il timeout.

- so
mes
os
im
ser
ste
- IN MEMORIA UTENTE → talvolta possono essere utilizzati dei buffer di livello user per mantenere messaggi anche dopo la spedizione; questo accade tipicamente quando non si ha più spazio a livello Kernel. In tal caso, quando un thread T spedisce un messaggio M, è necessario impostare una Receive() affinché il buffer utilizzato per memorizzare M venga liberato e sia pronto per spedizioni successive. Se non si ricorre a una Receive(), M può andare perso a seguito di una sovrascrittura (caso tipico delle spedizioni non bloccanti e asincrone).

Formato dei messaggi:

I messaggi possono essere strutturati in due modi:

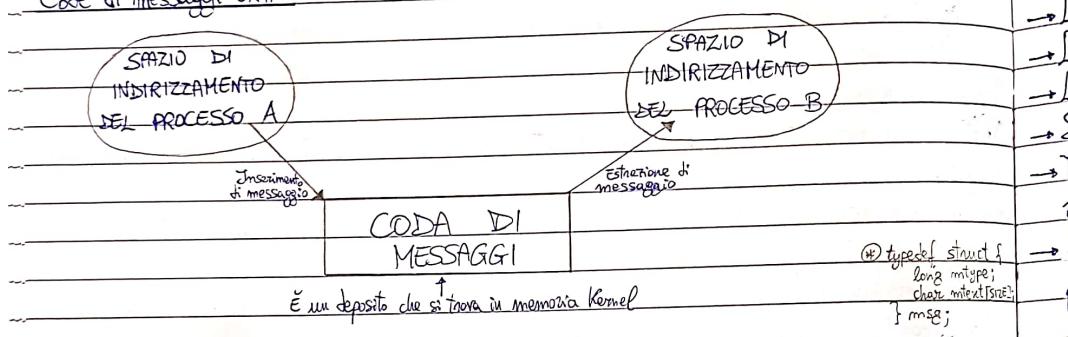
→ SOLO CONTENUTO (PAYLOAD)

stau
part

→ HEADER INIZIALE (tipo di messaggio; destinazione; sorgente; lunghezza; informazioni di controllo) + CONTENUTO

Il tipo e le informazioni di controllo sono di interesse non solo per le applicazioni in sé ma anche per il sottosistema di messaggistica, perché possono essere utilizzati per stabilire in che ordine i messaggi vanno effettivamente consegnati (uso di politiche più avanzate rispetto a First In First Out).

Coda di messaggi UNIX:



- La tecnica di indirizzamento è indiretta.
- La spedizione può essere o sincrona bloccante o sincrona non bloccante.
- La ricezione può essere o sincrona bloccante o sincrona non bloccante.
- Si ha un buffer in memoria Kernel con capacità limitata.
- I messaggi hanno un header che ne specifica il tipo e, all'interno delle code, possono essere suffivisi in tanti flussi differenti, uno per ogni tipologia. Ciò permette ai thread dei tori di estrarre dal deposito un messaggio appartenente a uno specifico flusso e, quindi, di una tipologia a scelta. Questa peculiarità è nota come MULTIPLEXING.

IPCT (Tabella degli oggetti di comunicazione tra processi):

Nel momento in cui un processo chiama `msgget()` per creare o aprire una coda di messaggi, gli viene restituito un descrittore di coda, che però non è relativo a una entry della tabella dei riferimenti valida per processo, bensì a una entry di una tabella unica in tutto il sistema che si chiama IPCT (Inter Process Communication Table). Questo consente a qualsiasi processo di disporre dello stesso descrittore di coda; infatti, dati due processi P, Q:

- P, Q possono accedere alla medesima coda di messaggi utilizzando lo stesso codice numero, ottenendo così il medesimo descrittore.
- P può scrivere su un file o una PIPE un descrittore di coda che potrà poi essere letto e acquisito da Q.
- Se P genera un processo figlio P' tramite una `fork()`, P' eredita gli eventuali descrittori di coda di P.

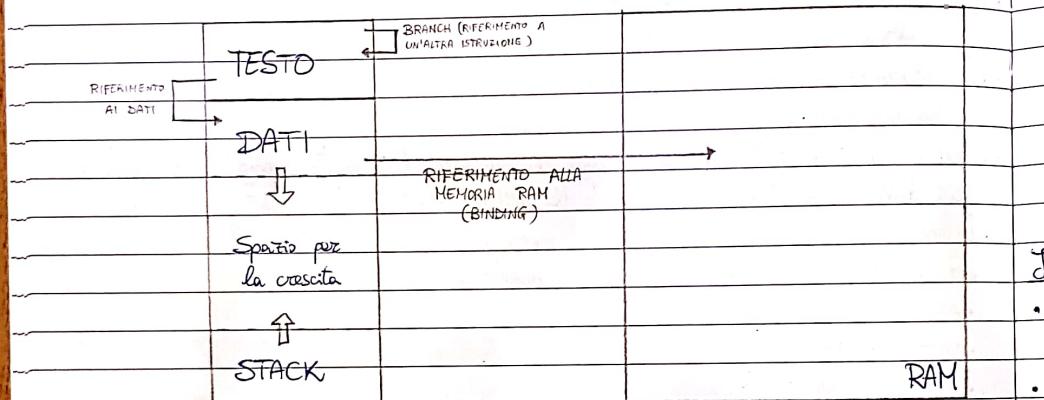
Mailslot Windows:

- La tecnica di intrattenimento è indiretta.
- La spedizione può essere sia sincrona che asincrona, sia bloccante che non bloccante.
- La ricezione può essere sia bloccante che non bloccante, sia sincrona che asincrona.
- Si ha un buffer in memoria Kernel con capacità limitata.
- Non c'è alcuna gestione della tipologia dei messaggi (no multiplexing), per cui le estrazioni dal mailslot avvengono secondo la politica First In First Out.
- I mailslot, a differenza delle code di messaggi UNIX, sono gestiti dal virtual file system, per cui molti loro servizi sono uguali a quelli dei file:
 - APERTURA: CreateFile()
 - LETTURA: ReadFile() → A differenza di UNIX, il campo lpNumberOfBytesRead deve essere posto almeno uguale alla dimensione del messaggio letto, altrimenti la Win-API darà errore.
 - SCRITTURA: WriteFile()
 - CHIUSURA DI UNA MANIGLIA: CloseHandle() → permette di eliminare il mailslot se non ci sono altri handle validi per lui.

25
26
27
28
29
30

ua
ea
ri

GESTIONE DELLA MEMORIA



Binding dei riferimenti:

È l'associazione di istruzioni e dati a indirizzi della memoria RAM e può essere effettuato in tre modi: a tempo di compilazione, a tempo di caricamento oppure a tempo di esecuzione.

→ BINDING A TEMPO DI COMPIAZIONE: il codice del programma è ASSOLUTO, per cui la posizione dell'applicazione e tutti i riferimenti di istruzioni e dati vengono stabiliti all'interno della memoria RAM già a tempo di compilazione e rimangono sempre fissi. Per modificare la locazione in cui il processo è caricato in memoria, è necessaria la ricompilazione.

Questa modalità di binding è dunque adatta solo a contesti di esecuzione seriale e a sistemi batch monoprogrammati.

→ BINDING A TEMPO DI CARICAMENTO: il codice del programma è RIVOCABILE, per cui, ogni volta che l'applicazione viene lanciata, viene definito un indirizzo base I nella RAM da cui iniziare a caricare il processo e, a ogni riferimento che nell'address space si trova a un certo offset Δ , corrisponde un riferimento nella RAM che si trova nella posizione $I + \Delta$. Tuttavia, la locazione in memoria del processo rimane fissa durante tutta l'esecuzione. Anche questa modalità di binding è adatta solo a contesti di esecuzione seriale e a sistemi batch monoprogrammati.

ATTUALMENTE IN VIGORE

→ BINDING A TEMPO DI ESECUZIONE: la posizione in memoria del processo può variare durante la sua attività a seguito di swap-out e swap-in. Perciò, il codice del programma è RILOCABILE DINAMICAMENTE e un riferimento viene riportato in memoria RAM solo nel momento in cui viene utilizzato durante l'esecuzione. Affinché ciò sia implementabile, sono necessari determinati supporti a livello hardware.
Questa modalità di binding è dunque adatta a sistemi batch multiprogrammati, multi-thread e time-sharing.

Instruzioni logici e fisici:

- INDIRIZZO LOGICO = modo con cui un'istruzione riferisce a un altro oggetto all'interno dell'address space (che può essere un'altra istruzione o un dato).
- INDIRIZZO FISICO = posizione effettiva in memoria RAM di un'istruzione o un dato.

La CPU lavora con instruzioni logici: per esempio, il Program Counter, che identifica qual è la prossima istruzione da eseguire, è un riferimento logico, non un indirizzo RAM.

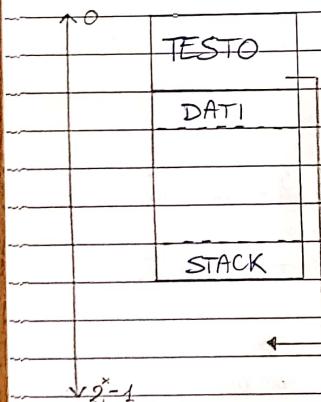
- Per i binding a tempo di compilazione e a tempo di caricamento, gli indirizzi logici e fisici coincidono, per cui ogni indirizzo generato dalla CPU viene direttamente caricato nel memory-address-register.
- Per il binding a tempo di esecuzione, invece, questa corrispondenza non c'è e il mapping degli indirizzi logici sugli indirizzi fisici avviene tramite un dispositivo hardware denominato Memory Management Unit (MMU).

MMU:

È un dispositivo che contiene il cosiddetto REGISTRO DI RILOCAZIONE, il cui valore è pari all'indirizzo base della RAM nel quale inizia l'address space del processo attualmente assegnato alla CPU. Perciò, l'indirizzo fisico in RAM di un determinato oggetto all'interno del contenitore di memoria è dato dalla somma tra il suo indirizzo logico e il valore del registro di rilocazione. Tale registro viene aggiornato dal software del sistema ogni volta che la CPU viene riassegnata.

In particolare, il valore di rilocazione rientra tra i metadati di gestione di ogni processo.

Supponiamo ora che all'interno dell'address space di una specifica applicazione gli indirizzi sono espressi a X bit. Poiché un numero a X bit può assumere 2^X valori diversi, gli spostamenti all'interno del contenitore di memoria possono potentialmente andare da 0 a $2^X - 1$. Tuttavia, l'address space potrebbe anche essere più piccolo e cominciare meno indirizzi.



In questo caso, per mettere di un bug o di un attacco, può succedere che un riferimento vada a finire al di fuori del contenitore, rischiando così di effettuare un'operazione di lettura o scrittura in una zona della RAM ad esempio relativa a un'applicazione diversa.

Per evitare questo scenario, all'interno della MMU viene inserito un secondo registro, denominato REGISTRO di MITE, il cui valore è pari all'indirizzo della RAM nel quale termina l'address space del processo attualmente assegnato alla CPU. In particolare, se a causa di un errore un indirizzo logico viene associato a un

indirizzo fisico al di fuori del contenitore, viene generata una trap e il controllo viene passato a un modulo del sistema operativo.

Tra l'altro, nei sistemi operativi più datati, gli address space avevano una taglia fissa decisa a tempo di compilazione, per cui non esisteva l'idea di poter espandere alcune zone del contenitore (e.g. heap, stack) durante l'esecuzione. Tenendo conto anche di questo, vediamo come può essere gestita la memoria RAM in contesti di multiprogrammazione, in cui si ha a che fare con più address space.

Suddivisione statica a partizioni multiple:

La memoria RAM è suddivisa in partizioni fisse, ciascuna delle quali è dedicata a ospitare un singolo processo. Una di esse in particolare contiene il codice e i dati del sistema operativo, che servono a gestire per esempio il CPU-scheduling e l'assegnazione della memoria. Comunque sia, in assenza di swapping, il grado di multiprogrammazione è limitato dal nu-

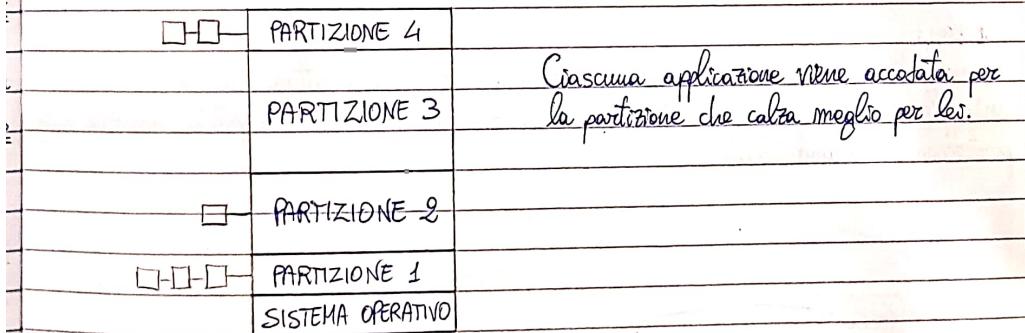
mero di partizioni.

PROBLEMI:

- Ogni processo occupa un'intera partizione intipentemente dalla sua dimensione. Può dunque presentarsi il fenomeno della FRAMMENTAZIONE INTERNA, in cui la taglia di un processo è minore di quella della partizione che lo ospita e, quindi, si ha uno spreco di spazio nella memoria RAM.
- Al contrario, un programma potrebbe essere troppo grande per essere contenuto in una qualsiasi partizione.
- Tali problemi possono essere alleviati (ma non del tutto risolti) usando molte partizioni di taglia piccola e poche partizioni di taglia grande.

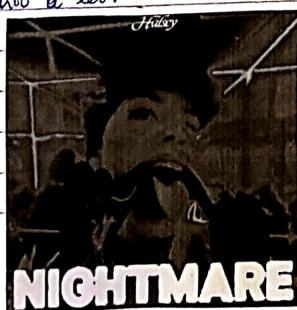
Inoltre, i processi possono essere allocati in RAM secondo due possibili logiche:

→ CON CODE MULTIPLE

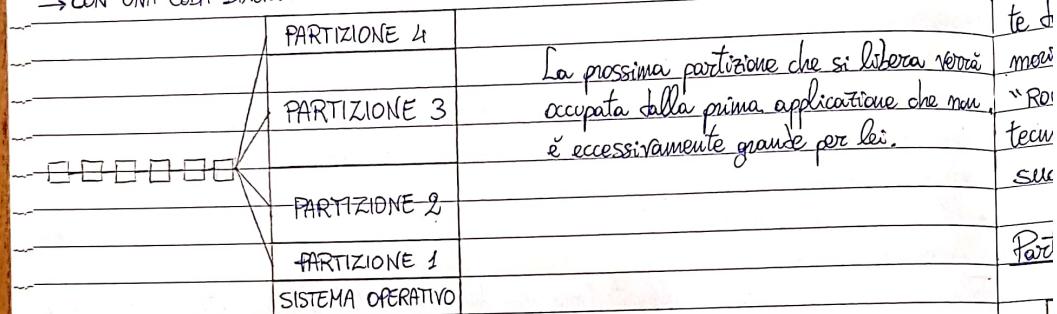


• Riduzione della frammentazione interna

• Rischio di frammentazione esterna: possono esserci partizioni di una grandezza tale che nessun processo viene assegnato a lei.



→ CON UNA CODA SINGOLA

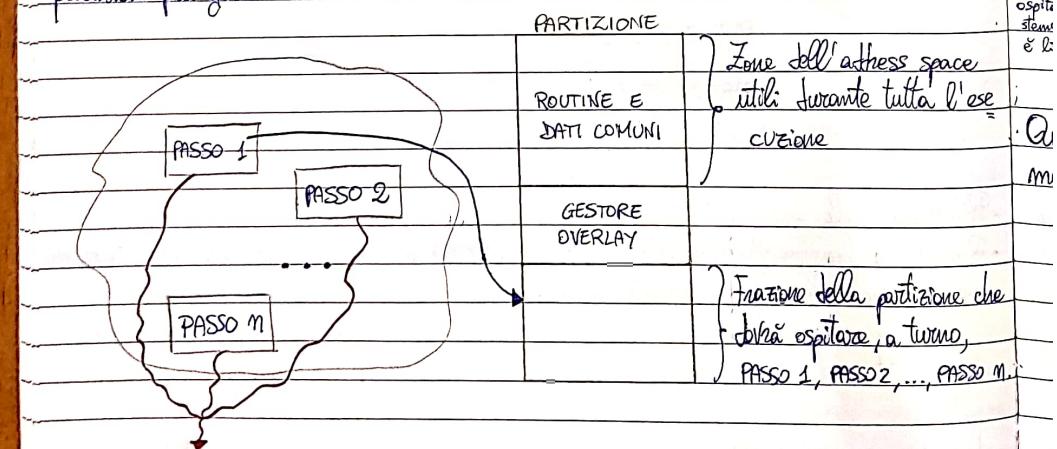


- Eliminazione della frammentazione esterna
- Rischio di un'elevata frammentazione interna dovuta a un'assegnazione più casuale dei processi alle varie partizioni

Overlay:

È una particolare tecnica della gestione della memoria che si attua nel momento in cui un'applicazione ha dimensioni talmente elevate che non può essere ospitata neanche nella partizione più grande della RAM.

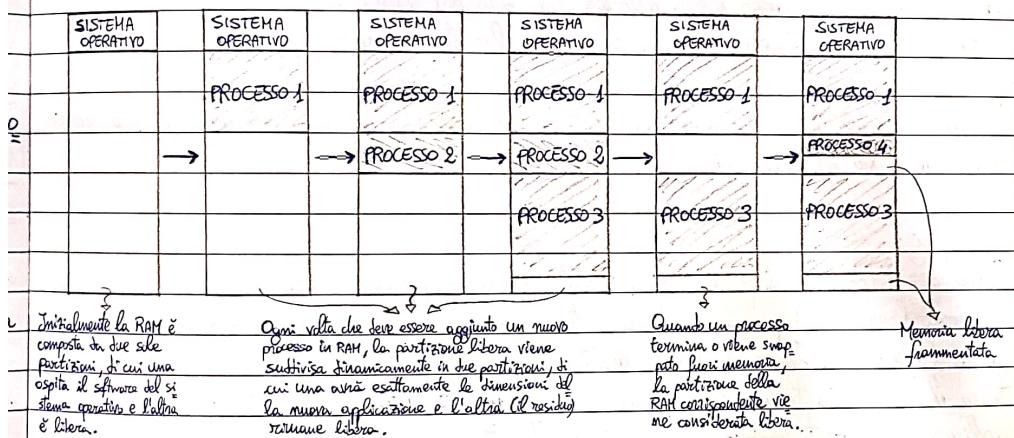
Jm1:
comp
part
ospiti
stew
è l:
Q
m



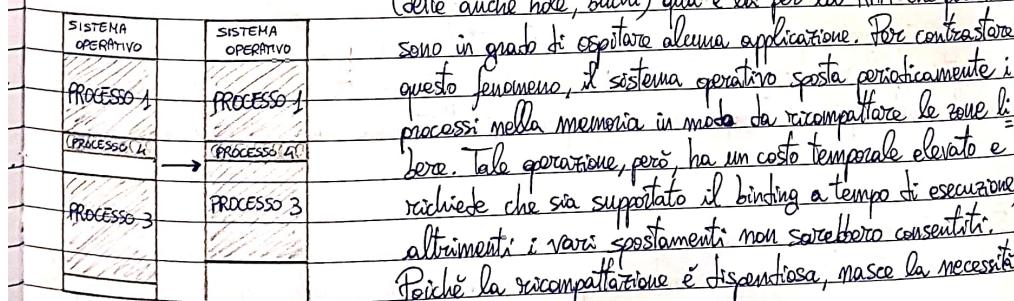
Zone dell'address space utili solo durante una specifica fase dell'esecuzione: si alterneranno per l'accesso in memoria RAM.

Nei sistemi operativi più ancestrali, questa tecnica doveva essere implementata manualmente dal programmatore, il quale si preoccupava, tra le varie cose, di gestire gli indirizzi di memoria (inserendo i giusti riferimenti verso PASSO 1, PASSO 2, ..., PASSO n) a partire dalla zona "ROUTINE E DATI COMUNI" e di implementare il gestore dell'overlay. Attualmente questa tecnica è automatizzata grazie alla memoria virtuale e ad altre features che analizzeremo successivamente.

Partizioni dinamiche:



Questa tecnica di partizionamento della memoria prevede del tutto la frammentazione interna, ma può presentare quella esterna nel momento in cui ci sono tante piccole partizioni libere (dette anche hole, buchi) qua e là per la RAM che però non



di trovare un criterio per l'allocazione dei processi nella RAM che faccia in modo che gli spostamenti debbano essere fatti il meno frequentemente possibile. In particolare, esistono tre politiche principali:

→ FIRST FIT: il processo viene allocato nel primo blocco in grado di ospitarlo, per cui non viene considerata la sua taglia esatta.

→ BEST FIT: il processo viene allocato nel più piccolo buco in grado di ospitarlo: così lo spazio che avanza è sperabilmente il più piccolo possibile in modo da non impattare troppo sulla frammentazione esterna, ma difficilmente potrà accogliere nuove applicazioni.

→ WORST FIT: il processo viene allocato nel buco più grande: così viene generato a sua volta un hole abbastanza esteso da essere facilmente in grado di ospitare nuove applicazioni.

Le politiche best fit e worst fit sono le più interessanti perché sono strategiche, ma nessuna delle due è né ottimale, né migliore dell'altra.

Swapping:

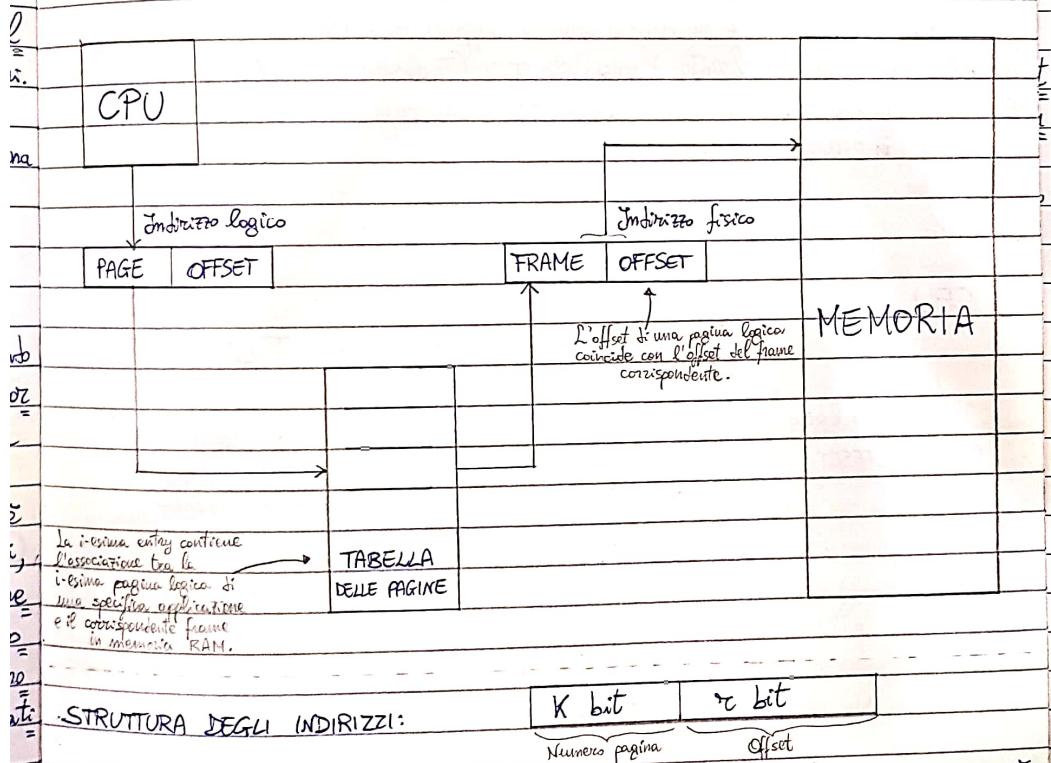
Come già sappiamo, i processi nello stato di blocco possono essere riportati fuori dalla memoria (swap-out) per far posto ad altre applicazioni, per poi essere riportati dentro quando servono (swap-in). In particolare, questa operazione è interessante nei sistemi che supportano il binding a tempo di esecuzione, che sono gli unici che consentono di riportare le applicazioni in punti arbitrari della RAM all'atto dello swap-in.

Affinché un processo possa essere portato fuori memoria senza alcuna inconsistenza, è necessario che nel suo address space, e quindi nella zona della RAM riservata a lui, ci sia completa inattività: la CPU infatti non è l'unico componente a lavorare in memoria, ma c'è anche il DMA che può eseguire operazioni di I/O asincrone sulle applicazioni, anche quando loro non sono nello stato running. Per affrontare questa problematica, il DMA registra le attività all'interno di appositi buffer nel sistema operativo, il quale riporterà le operazioni sui processi d'interesse nel momento opportuno.

Paginazione:

È una tecnica di allocazione dei processi in RAM che supera completamente il vincolo dato dalla necessità di inserire gli address space in zone contigue della memoria. Infatti, le

spazio di indirizzamento di un'applicazione viene visto come un insieme di PAGINE che hanno una taglia fissa pari a X byte, mentre la RAM è allo stesso modo partitionata in FRAME, anch'essi tutti da X byte. Ogni pagina dell'address space viene caricata in uno specifico frame in memoria, e l'allocazione, appunto, può anche avvenire in modo non contiguo. Così, viene risolta del tutto la frammentazione esterna, ma può presentarsi quella interna (in maniera comunque non troppo incisiva) nel momento in cui la taglia di un contenitore di memoria non è un multiplo di X , per cui una porzione di un frame in RAM rimane inutilizzata.



La riga entry contiene l'associazione tra la riga pagina logica di uno specifico applicazione e il corrispondente frame in memoria RAM.

STRUTTURA DEGLI INDIRIZZI:

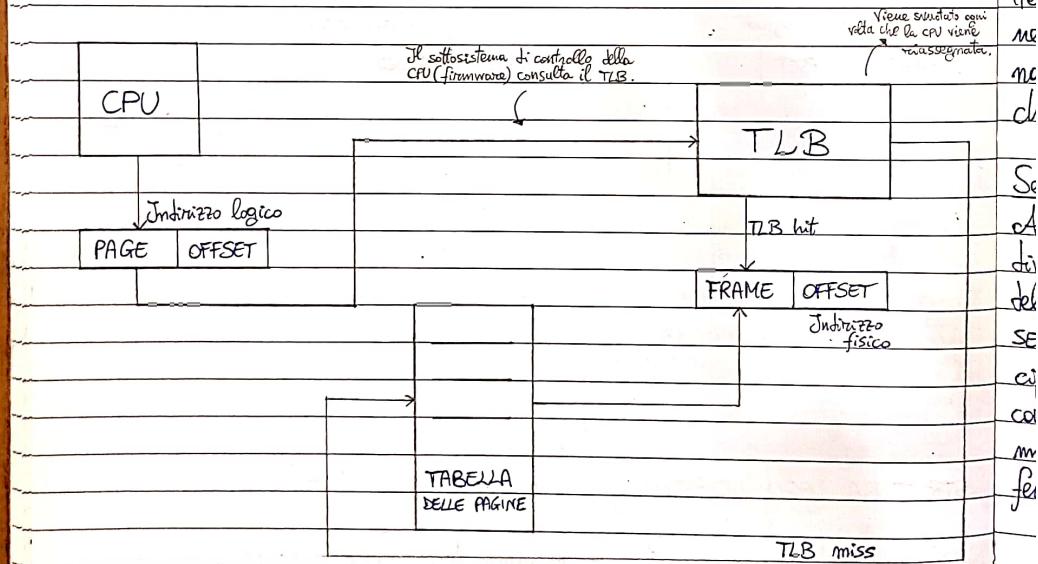
K bit	r bit
Numero pagina	Offset

Se il numero di bit per l'offset è r , le pagine devono avere una dimensione pari a 2^r : in questo modo, fissati i K bit più significativi, e data la pagina P corrispondente, qualunque indirizzo ricade all'interno di P, senza possibilità di interferire con altre pagine collocate in altri frame in memoria.

Di conseguenza, a parità di lunghezza totale degli indirizzi, è necessario trovare il giusto compromesso tra il numero di bit che identificano la pagina (K) e il numero di bit per l'offset (r): se r è troppo grande, le pagine hanno dimensioni elevate, per cui il numero della frammentazione interna diventa più consistente; al contrario, se K è troppo grande, la tabella delle pagine esplode.

Ma chi è che consulta la tabella delle pagine? Se fosse il software del sistema operativo, a ogni istruzione deve essere generata una trap per passare il controllo a lui, il che è infatti. Perciò, è necessario qualche supporto hardware.

In primo luogo, viene utilizzato il cosiddetto TLB (Translation-Lookaside-Buffer), che è una cache che mantiene alcune associazioni tra pagine e frame per il processo correntemente in esecuzione sulla CPU.



In caso di miss da parte del TLB, il sottosistema di controllo della CPU consulta anche la tabella delle pagine per poi caricare nel TLB l'associazione PAGINA-FRAME utile per la

istruzioni successive: di fatto, viene sfruttato il principio della località, poiché è molto probabile che l'applicazione debba lavorare sulla stessa porzione del testo (o anche dei dati, ad esempio quando si hanno degli array), e quindi della stessa pagina, per un po' di tempo.

Dunque, il ruolo del software del sistema operativo è semplicemente quello di decidere quali frame fisici assegnare a determinate pagine logiche e popolare la tabella delle pagine di conseguenza. Infatti, è il sistema operativo stesso a tener traccia di quali frame sono liberi, il loro e quali no istante per istante.

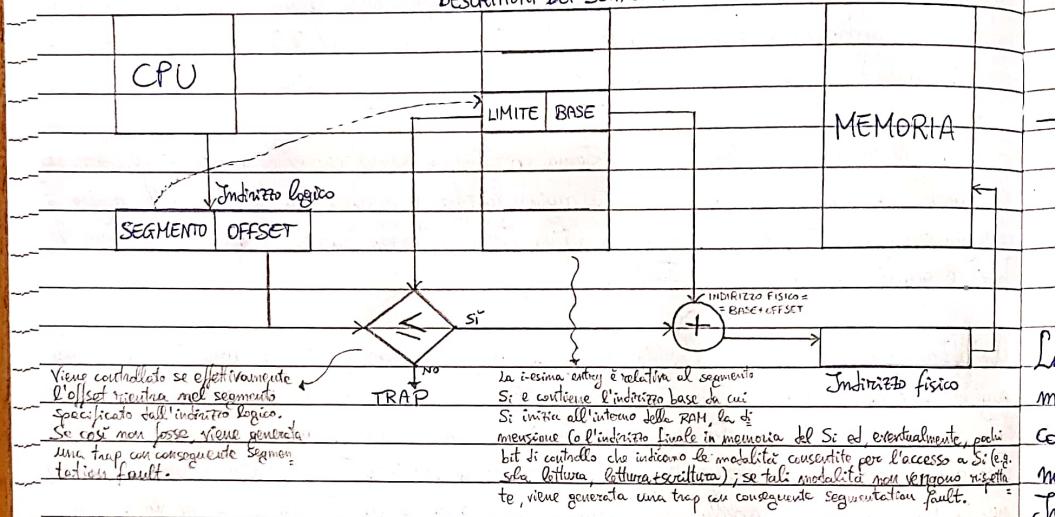
Un'altra cosa importante è che in memoria, a livello Kernel, esiste una tabella delle pagine per ogni processo attivo, per cui la CPU deve essere tenuta in moto tale da tener traccia della tabella giusta nel momento in cui sta girando una specifica applicazione. Per questo motivo, all'interno del processore si trova un registro che identifica la posizione in memoria della tabella delle pagine correntemente da utilizzare, e che viene aggiornato ogni volta che la CPU viene riassegnata. Nei processori x86, questo registro si chiama CR3 (Control Register 3).

Segmentazione:

Anch'essa è una tecnica di allocazione dei processi in RAM in cui è possibile che frazioni diverse del medesimo spazio di indirizzamento siano posizionate in locazioni non contigue della memoria. La differenza della paginazione è che ciascun address space è suddiviso in SEGMENTI, che non sono necessariamente tutti della stessa taglia. Di conseguenza, il principio abbattito della segmentazione è quello della partizione dinamica, in cui la RAM non è composta da frazioni predefinite, bensì viene suddivisa di volta in volta in base ai segmenti che deve ospitare. Si tratta perciò di una tecnica che evita completamente il fenomeno della frammentazione interna, ma potrebbe essere soggetta a quella esterna.

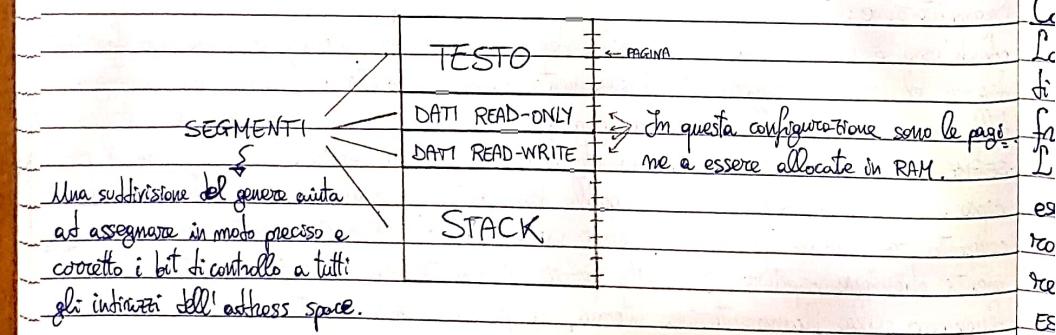


TABELLA DEI
DESCRITTORI DEI SEGMENTI



Segmentazione paginata:

È possibile combinare le tecniche di paginazione e segmentazione nel seguente modo:



Una locazione di memoria viene identificata in due step:

→ PRIMO INDIRIZZO: [NUMERO DI SEGMENTO] [OFFSET NEL SEGMENTO]

Il numero di segmento, tramite la tabella dei descrittori, non identifica la base all'interno della RAM, bensì l'indirizzo logico in cui quel segmento inizia nell'address space.

Di conseguenza, per individuare la posizione effettiva nel contenitore (ovvero l'INDIRIZZO LINEARE), basta sommare la base del segmento con l'offset.

→ SECONDO INDIRIZZO:

NUMERO DI PAGINA | OFFSET NELLA PAGINA

L'indirizzo lineare viene espresso secondo uno schema paginato, con alcuni bit afferenti a una specifica pagina nell'address space, ed altri bit che indicano l'offset in quella stessa pagina. Dopo ciò, attraverso la tabella delle pagine, è possibile risalire al frame corretto, all'interno della memoria RAM.

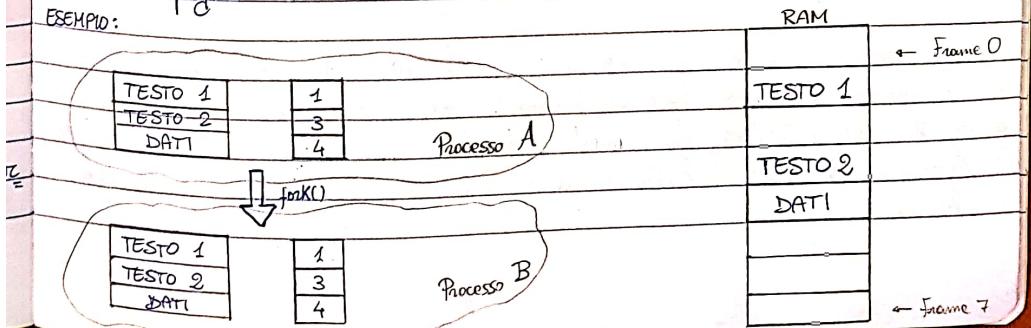
La segmentazione paginata è tuttora implementata nell'architettura x86, dove alcuni segmenti di default sono CS (CODE SEGMENT) e DS (DATA SEGMENT). Infatti, tra le varie cose, permette la corretta gestione delle variabili TLS (Thread Local Storage), che vengono associate a dei segmenti appositi (uno per ogni thread attivo all'interno del processo). In questo modo, ciascuna traccia accede in modo sicuro e senza rischio di interferenze alla zona di address space contenente le sue variabili TLS, semplicemente usando il numero del segmento adeguato.

Condivisione della memoria:

La paginazione supporta uno schema di condivisione della memoria: se gli address space di più processi differenti hanno delle pagine in comune, queste vengono mappate nello stesso frame all'interno della RAM, risparmiando così memoria fisica.

L'efficienza di questa tecnica viene espressa nel momento in cui, su UNIX, un processo A esegue una fork(), generando un clone B. Almeno inizialmente, B eredita da A l'intero address space, e tutte le sue pagine logiche vengono mappate sugli stessi frame in RAM relativi alle pagine di A.

ESEMPIO:



È tipico che i due processi lavorino in modo indipendente, per cui potrebbe essere necessario evitare che le operazioni di scrittura di B siano osservabili anche da A e viceversa. Per questo motivo, nelle tabella delle pagine viene aggiunto per ogni entry un ulteriore bit di protezione denominato COPY ON WRITE: se è settato a 1 per una specifica pagina P, nel momento in cui uno dei due processi (supponiamo B) deve sovrascrivere il contenuto di P, la pagina viene prima duplicata in memoria RAM e la tabella di B viene aggiornata per associare P al frame fisico appena allocato.

Controllazione delle pagine Kernel:

Mentre un thread è in esecuzione, se esso chiama una system call, oppure se si raggiunge un interrupt, un handler (ovvero il gestore di accesso al Kernel) passa il controllo al Kernel, che inizia a operare con le sue istruzioni macchina e i suoi dati, i quali sono organizzati in memoria secondo uno schema basato su paginazione.

Di conseguenza, dato l'address space di un processo, esso è costituito da alcune pagine logiche a livello user e altre pagine logiche a livello Kernel. Sono necessari funzionali ulteriori bit di controllo per ogni entry della tabella che discriminino se la relativa pagina è accessibile in modalità privilegiata o non privilegiata.

In particolare, tutti i frame in RAM relativi a istruzioni o dati di livello Kernel sono a istanza singola, per cui lo schema della controllazione della memoria è ampiamente sfruttato: in effetti, qualunque thread entri in modalità Kernel, deve lavorare con delle istruzioni machine e dei dati che sono predefiniti, a seconda che sia soggetto a un interrupt, a una chiamata di system call o alla terminazione. A questo punto subentra il problema della sincronizzazione tra più thread concorrenti che devono entrare in modalità Kernel, ma lo analizzeremo successivamente.

NB: Le pagine logiche di livello Kernel non sono governate dallo schema copy on write.

Nel caso specifico dei processori x86 a 32 bit, l'address space di un determinato processo può contenere fino a 2^{32} (= 4 GIGA) indirizzi differenti, ciascuno dei quali corrisponde a un byte. Ogni contenitore, che a questo punto può occupare fino a 4 GB di memoria, dedica l'ultimo GB alle pagine di livello Kernel e ha a disposizione ben 3 GB per le pagine di livello user.

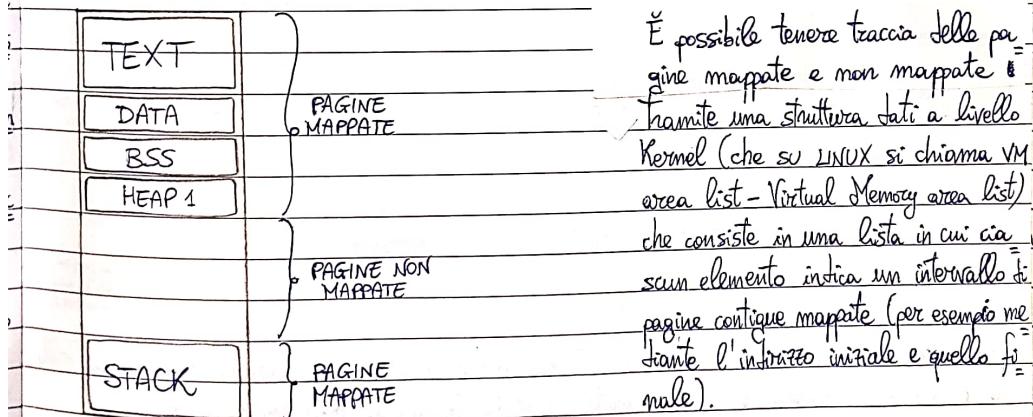
Pagine mappate e non mappate:

Consideriamo l'address space di uno specifico processo attivo. Una sua qualsiasi pagina P_i può essere allocata in RAM come può anche non esserlo, per esempio perché il contenitore di memoria è stato interamente o in parte swapped fuori. In questo scenario il concetto di memoria è virtualizzato: determinate pagine dell'address space logicamente esistono ma non sono momentaneamente reperibili nello storage fisico.

Più in generale, le uniche pagine correntemente utilizzabili (e quindi effettivamente valide) sono quelle **MAPPATE NELL'ADDRESS SPACE**. È comunque possibile chiedere al memory manager del sistema operativo di mappare in corso d'opera ulteriori pagine, e sta proprio in questo la capacità di espansione dinamica della quantità di informazioni che possono essere mantenute all'interno del contenitore di memoria.

ESEMPIO:

Consideriamo solo la zona a livello utente di un address space: le pagine di livello Kernel sono valide solo durante le esecuzioni in modalità privilegiata.



È possibile tenere traccia delle pagine mappate e non mappate tramite una struttura dati a livello Kernel (che su LINUX si chiama VM area list - Virtual Memory area list) che consiste in una lista in cui ciascun elemento indica un intervallo di pagine contigue mappate (per esempio mediante l'indirizzo iniziale e quello finale).

N.B.: Attenzione a non confondere i concetti di "pagina mappata" e "pagina allocata in RAM"!

A partire da questa configurazione è possibile apportare la seguente modifica:

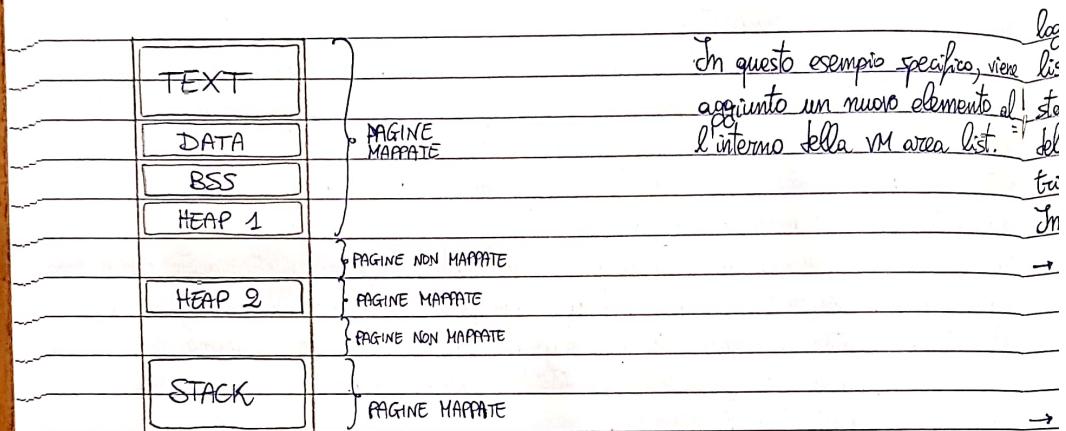
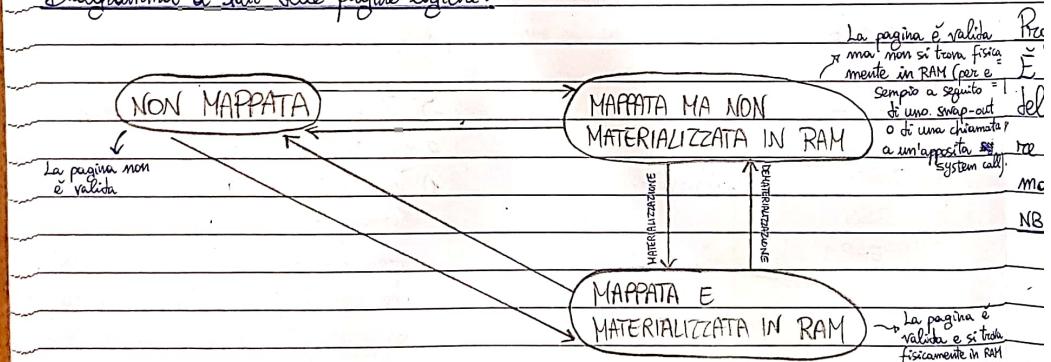


Diagramma a stati delle pagine logiche:



NB: È necessario un ulteriore bit di controllo per ogni entry all'interno della tabella delle pagine per discriminare le pagine materializzate da quelle non materializzate.

Materializzazione di pagine anonime:

Nel momento in cui una pagina P viene mappata tramite la system call `mmap()` e il flag `MAP_ANONYMOUS`, essa non viene materializzata subito. Di conseguenza, all'istante del primo accesso, non si trova ancora in alcun frame in memoria RAM, per cui viene generato un PAGE FAULT che passa il controllo all'architettura di traduzione tra instruzio-

logico e indirizzo fisico. In particolare, questa architettura va a consultare la VM area list per verificare se effettivamente P è mappata e se l'accesso è conforme alle regole prese stabiliti per P (e.g. read only, read+write): se sì, P viene materializzata, la tabella delle pagine viene aggiornata e il controllo della CPU torna all'applicazione in esecuzione; altrimenti viene generata una trap con conseguente SEGMENTATION FAULT.

In ogni caso, c'è qualche differenza tra un accesso in lettura e uno in scrittura:

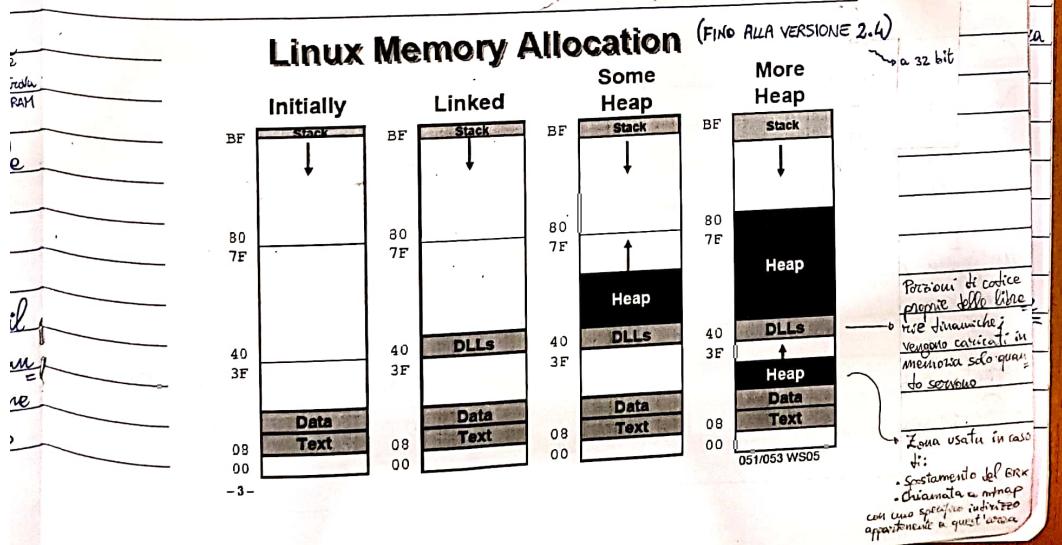
→ Se l'accesso è in lettura, per mezzo del flag MAP_ANONYMOUS, viene caricato in RAM un frame con tutti i byte impostati a zero (MEMORIA EMPTY-ZERO); tuttavia, se un frame fatto così era già da prima fisicamente presente, si attua il principio della costruzione.

→ Se l'accesso è in scrittura, onde evitare interferenze, viene allocato a priori un nuovo frame all'interno della memoria RAM.

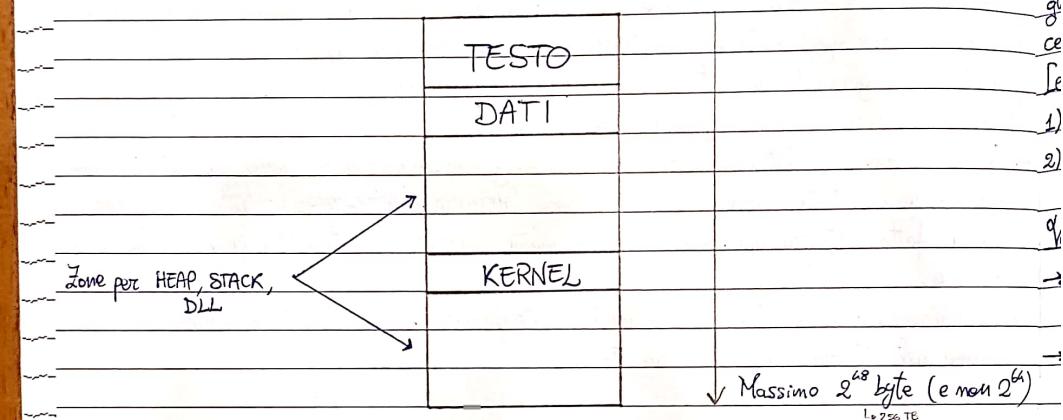
Program-break (BRK) in sistemi UNIX:

È l'indirizzo lineare prima del quale tutte le pagine sono sicuramente mappate ma dopo del quale potrebbero anche non esserlo. Perciò, spostare il BRK in avanti potrebbe comportare la mappatura di alcune pagine, mentre riportarlo all'interno potrebbe causare delle demappature.

N.B.: Il program-break non è necessariamente allineato alla fine di una pagina logica.



Nelle versioni successive (a 64 bit), le pagine accessibili a livello Kernel si trovano nel bel mezzo del contenitore e non in fondo:



Struttura degli address space in Windows:

→ VERSIONE A 32 BIT DI DEFAULT: 2GB di pagine di livello user + 2GB di pagine di livello Kernel

→ VERSIONE A 32 BIT CON 4GT (4 GB tuning): 3G di pagine di livello user + 1GB di pagine di livello Kernel

→ VERSIONI A 64 BIT: altamente variabile

Comando malloc():

Non è una funzione dello standard di sistema né in Windows né in UNIX, bensì appartiene allo standard di linguaggio. Quando viene invocata, chiama mmap() o VirtualAlloc() a seconda del sistema operativo, e mappa le pagine necessarie.

Tale comando, inoltre, adotta uno schema di PRRESERVING DI INDIRIZZI LOGICI, in cui sfrutta un'intera pagina per:

- Allocare dinamicamente le aree di memoria richieste dall'utente.
- Allocare dei metadati per tener traccia delle locazioni di memoria già utilizzate dall'utente e di quelle libere.
- Preservare tutto lo spazio rimanente per chiamate di malloc() successive.

UN ARRAY DICHIARATO così: char v[] = {0}; // align=4; -- (aligned(4096));
VERRE COLLOCATO ESATTAMENTE ALL'INIZIO DI UNA PAGINA

2) Memoria virtuale:

È un'architettura basata sulla possibilità di avere all'interno di un address space alcune pagine mappate che non sono correntemente presenti in RAM. In particolare, permette di avere processi parzialmente presenti in memoria e parzialmente swapped fuori.

Le possibili cause della non presenza in RAM di pagine mappate sono due:

- 1) Le pagine non sono mai state materializzate.
- 2) Le pagine sono state materializzate ma poi sono state portate fuori RAM.

vantaggi della memoria virtuale:

- Permette l'esecuzione di processi il cui spazio di indirizzamento eccede le dimensioni della memoria fisica.
- Permette di aumentare il numero di processi che possono essere mantenuti contemporaneamente in RAM: infatti, ciascuno di essi ha solo una porzione dell'address space in memoria, per cui impiega meno risorse fisiche.
- Permette la riduzione del tempo necessario alle operazioni di swapping, poiché, se eseguite in modo parziale, richiedono un numero di operazioni di I/O minore.
- Sfrutta i principi di località spaziale e temporale dato che un processo lavora sulle stesse pagine per un po' di tempo quando si imbatté in cicli all'interno della zona testo o in array all'interno della zona dati.
- Gestisce con efficienza il sovraccarico delle strutture dati: se per esempio viene allocata tanta memoria per un array ma, di fatto, in esecuzione ne serve solo una frazione, allora verranno materializzate solo le pagine strettamente necessarie, evitando così uno spreco di risorse.
- Le informazioni che servono di rado (come per esempio il blocco di istruzioni macchina da eseguire in caso di errore severo) vengono tenute fuori memoria finché effettivamente non ce n'è bisogno.

C) gestione dei page fault:

Come già sappiamo, la tabella delle pagine di uno specifico processo ha per ogni entry un bit di controllo (denominato bit di presenza) che indica se la pagina corrispondente è allocata o meno in un frame all'interno della RAM. Se si tenta l'accesso a una pagina P

col bit di presenza settato a zero (e quindi non presente in memoria fisica), viene generato un PAGE FAULT che, se possibile, porta il software del Kernel a materializzare P .

In particolare, i page fault possono essere classificati in MINOR-FAULT e MAJOR-FAULT.

→ MINOR-FAULT: si ha quando la pagina P non è presente in alcun punto dell'architettura e, quindi, non è swapped-out. In tal caso, il sistema operativo deve semplicemente reificare quali sono i frame liberi e sceglierne uno per assegnarlo a P : non c'è necessità di interagire con un dispositivo di I/O per recuperare il contenuto di P .

→ MAJOR-FAULT: si ha quando la pagina P è swapped fuori memoria. In tal caso, il software del Kernel deve interagire con un hard-drive che ricaricherà P in un frame all'interno della RAM, e poi ratagnanciare quel frame a P nella tabella delle pagine.

Di conseguenza, per servire un major-fault, è necessario più tempo.

T
Pe

Trascorso il ritardo provocato dai minor-fault, il tempo medio effettivo di accesso alla memoria è uguale a:

$$ma + pft \cdot f$$

DII

→ ma = tempo di accesso alla memoria RAM

di

→ pft = tempo medio di caricamento della pagina da hard-drive (tempo di page fault)

po

→ f = frequenza di page fault

→

Poiché "pft" è più grande di "ma" per almeno tre ordini di grandezza, è necessario adottare una politica di gestione della memoria RAM che riduca il più possibile la frequenza dei page fault f . In particolare, quando la RAM è saturata e bisogna materializzare un'altra pagina, è bene effettuare una sostituzione in modo furbo, cercando di non dematerializzare una pagina che servirà nuovamente a breve.

→

Aspetti coinvolti nella sostituzione delle pagine:

1) Resident set management, che consiste in due questioni fondamentali:

• Se l'insieme delle pagine potentialmente sacrificabili per la sostituzione deve essere limitato al processo che causa il page fault oppure può essere esteso a tutte le applicazioni.

In
lo

• Qual è il numero minimo di frame da allocare per ciascun processo.

te

2) Qual è la vittima da scegliere all'interno dell'insieme delle pagine potentialmente sacrificabili. In ogni caso, a questo insieme non possono appartenere le pagine soggette

a blocco, le quali devono ritrovarsi assegnate sempre allo stesso frame fisico, per cui non potrebbero essere scappate fuori memoria. Un esempio tipico sono le pagine di livello kernel. In particolare, il sistema operativo mantiene un lock bit per ciascun frame, il cui valore indica se la pagina corrispondente può essere considerata per una sostituzione oppure è bloccata.

3) Come tener traccia di eventuali modifiche apportate alla pagina P da scappare fuori: in effetti, se P è stata accedita in scrittura, è necessario effettuare una copia esplicita del suo contenuto all'interno dell'area di swap nel momento in cui viene dematerializzata per non perdere le informazioni aggiornate.

Tracciamento delle modifiche:

Per sapere se bisogna copiare nell'area di swap la pagina P da dematerializzare, all'interno della tabella delle pagine si ricorre a un ulteriore bit per ogni entry denominato DIRTY BIT, che è settato a 1 se la copia deve essere effettuata e viceversa. Se supponiamo, di sì, data la pagina P da scappare fuori e data una pagina Q da inserire in memoria al posto di P, le operazioni che vengono eseguite sono le seguenti:

→ tramite un'operazione di I/O, il contenuto aggiornato di P viene caricato nell'area di swap.

→ Sia il bit di presenza che il dirty bit relativi a P, che finora erano impostati a 1, vengono resettati a 0.

→ Tramite un'altra operazione di I/O, Q viene allocata nel frame dove si trovava P. → Il bit di presenza relativo a Q, che valeva 0, viene settato a 1 dal software del Kernel; anche il corrispondente dirty bit era uguale a 0, ma il firmware del processore non ne cambia il valore finché non viene eventualmente effettuata un'operazione di scrittura su Q.

In particolare, poiché il dirty bit viene settato a 1 da parte del sottosistema di controllo in memoria (il firmware) e viene consultato (ed eventualmente resettato a 0) da parte del software del sistema operativo, esso fa parte della classe degli STICKY FLAG.

i. Algoritmi di selezione della vittima:

→ METRICA DI VALUTAZIONE: frequenza di page fault (in particolare major fault)

ES

Paghi

→ METODO DI VALUTAZIONE: si utilizzano determinate sequenze di riferimenti a pagine logiche, generate in modo casuale oppure in base a tracce reali di esecuzione.

→ ASPETTATIVA PER UN BUON ALGORITMO: all'aumentare del numero di frame della memoria centrale, per qualunque sequenza di riferimenti a pagine logiche, la frequenza di page fault dovrebbe decrescere monotonicamente.

Page

Algoritmo ottimo:

• Seleziona come vittima la pagina alla quale ci si riferisce dopo più tempo.

• E

• È impossibile da implementare perché richiede che il sistema operativo abbia una conoscenza esatta degli eventi futuri.

no

• Si può comunque usare come termine di paragone per valutare gli altri algoritmi.

Tu

ESEMPIO:

Page reference:	2	3	2	1	5	2	4	5	3	2	5	2	Page
	2	2	2	2	2	2	4	4	4	2	2	2	
		3	3	3	3	3	3	3	3	3	3	3	
:				1	5	5	5	5	5	5	5	5	

Page fault:	P	P	P	P	P	P	P	Page

Algoritmo Least-Recently-Used (LRU):

• Seleziona come vittima la pagina alla quale non ci si riferisce da più tempo.

Arr

• Non è utopico come l'algoritmo ottimo poiché il sistema operativo deve basarsi sugli eventi passati.

Gli

• Rimane comunque molto difficile da implementare: affinché il Kernel tenga traccia di ogni singolo accesso che viene effettuato, deve ricevere una trap per ciascuna iterazione, cosa che si basa solo in caso di page fault. Un'alternativa potrebbe consistere nell'utilizzo di architetture hardware supplementari che memorizzino la sequenza degli accessi che si fanno in memoria ma, di fatto, non è stata ancora trovata una soluzione efficace.

FIF

ESEMPIO:

Page riferite:	2	3	2	1	5	2	4	5	3	2	5	2
Page fault:	P	P	P	P	P	P	P	P	P	P	P	P

Algoritmo First-In-First-Out (FIFO):

• Seleziona come vittima la pagina presente in memoria da più tempo.

• È semplice da implementare: basta tenere traccia delle materializzazioni delle pagine, il che è immediatamente possibile, poiché l'inserimento di una pagina all'interno di un frame fisico deriva sempre da un page fault, il quale passa il controllo al Kernel.
• Tuttavia, a differenza dei due algoritmi precedenti, non sfrutta il principio della località.

ESEMPIO:

Page riferite:	2	3	2	1	5	2	4	5	3	2	5	2
Page fault:	P	P	P	P	P	P	P	P	P	P	P	P

Anomalia di Belady:

Gli algoritmi di selezione della vittima possono essere suddivisi in due categorie:

→ A: algoritmi che soffrono l'anomalia di Belady

→ B: algoritmi che non soffrono l'anomalia di Belady

In particolare, un algoritmo è caratterizzato da questa anomalia se, a parità di sequenza dei riferimenti e così l'aumentare dei frame in memoria RAM, il numero dei page fault che esso produce può aumentare.

FIFO è un algoritmo di questo tipo.

ESEMPIO:													
Pagine riferite:	0	1	2	3	0	1	4	0	1	2	3	4	tiva vittima
	0	1	2	3	0	1	4	4	4	2	3	3	sare
	0	1	2	3	0	1	1	1	1	4	2	2	un
		0	1	2	3	0	0	0	0	1	4	4	delle
3 page fault:	P	P	P	P	P	P	P	P	P	P	P	P	ugli le,
Pagine riferite:	0	1	2	3	0	1	4	0	1	2	3	4	l'ar
	0	1	2	3	3	3	4	0	1	2	3	4	Qua
	0	1	2	2	2	2	3	4	0	1	2	3	sibi
		0	1	1	1	1	2	3	4	0	1	2	par
		0	0	0	0	1	2	3	4	0	1	Nel	ter tan
10 page fault:	P	P	P	P	P	P	P	P	P	P	P	P	pro line

Dall'altra parte, gli algoritmi che non soffrono l'anomalia di Belady sono detti a STACK che e hanno la seguente caratteristica: per qualsiasi sequenza di riferimenti e a ogni istante all tempo, l'insieme delle pagine in una RAM composta da n frame è un sottoinsieme della collezione delle pagine in una RAM composta da $n+1$ frame. LRU è un esempio di algoritmo a stack.

Algoritmo dell'ordiglio (Not Recently Used):

Riprende la logica dell'LRU ma è più facile da implementare: seleziona come vittima una qualsiasi tra le pagine a cui non ci si è riferiti recentemente.

A ciascun frame in memoria (e quindi a ogni pagina materializzata) è associato un RE= bit che si trova nella tabella delle pagine e che viene settato a 1 dal firmware ogni volta che la pagina corrispondente viene referenziata con la necessità del sottosistema del processore di accedere alla tabella (per esempio, a seguito di un TLB miss). D'altra parte, il bit viene periodicamente impostato a 0 da parte di un demone di sistema per esem= bit pio ogni volta che il TLB viene scambiato a seguito di un cambio di contesto.

Nel momento in cui è necessario riempire una pagina all'interno della RAM, i vari re= ne ference bit vengono consultati finché non se ne trova uno impostato a 0: la pagina rela= no

In questo bit sembra non essere stata usata la recente, per cui viene scelta la come vittima. Tuttavia, potrebbe succedere che tutti i bit siano impostati a 1, per cui il Kernel non sarà in grado di selezionare una pagina da dematerializzare: per questo motivo, ogni volta che un reference bit viene consultato, viene automaticamente resettato a 0, per cui, nella peggiore delle ipotesi, è necessario effettuare un giro completo dei frame prima di incontrare un bit uguale a 0; in questo caso specifico, viene di fatto scelta come vittima una pagina casuale, il che è un comportamento patologico dell'algoritmo il quale, talvolta, può presentare l'anomalia di Belady.

Questa problematica viene accentuata nelle architetture multi-core, in cui non è sempre possibile identificare la vittima: supponiamo di avere due thread X, Y che vengono eseguiti in parallelo su due CPU-core diversi. Allora può presentarsi il seguente scenario:

nel momento in cui la RAM è saturata, X sta per accedere a una pagina logica non mappata, per cui deve prima scegliere una vittima da swapare fuori memoria consultando i reference bit. Supponiamo che il bit relativo al frame A sia impostato a 1 e che, quindi, venga resettato a 0. Può succedere che nel frattempo il thread Y acceda alla pagina logica contenuta in A e che, magari, il suo reference bit venga riportato a 1. Se per caso X compie un giro completo per la selezione della vittima, si ritroverà poi il reference bit associato ad A di nuovo uguale a 1; chiaramente, questa situazione potrebbe verificarsi per tutti i frame della memoria RAM, per cui non si è più in grado di selezionare una buona vittima nel momento in cui c'è bisogno di liberare spazio per accogliere la nuova pagina logica. In definitiva, ritrovarsi con la RAM saturata non è mai una buona situazione: perciò, è bene scegliere le vittime da dematerializzare in anticipo, in modo tale che ci siano sempre dei frame disponibili quando deve essere allocata una nuova pagina.

a) Algoritmo dell'ondaggio con bit addizionali:

È una variante del Not Recently Used in cui, nella tabella delle pagine, oltre al reference bit, viene consultato anche il dirty bit. Infatti, dal punto di vista dell'efficienza, è meglio dematerializzare una pagina che non è stata sovrascritta, in modo tale da non dover effettuare una copia esplicita all'interno dell'area di swap. Perciò, quando deve essere scelta una vittima da rimuovere dalla RAM, hanno precedenza le pagine che hanno sia il reference

bit che il dirty bit pari a 0; solo nel caso in cui non ce ne sono, vengono considerate queste quelle col reference bit uguale a 0 e il dirty bit uguale a 1.

con f
Si t:

Resident set:

È il numero di frame fisici che sono destinati a ospitare pagine di uno specifico processo. Più diminuisce, più applicazioni possono essere mantenute almeno parzialmente in memoria; tuttavia, se è troppo piccolo, la frequenza dei page fault diventa inaccettabilmente alta, poiché le poche pagine logiche materializzate non sono sufficienti per catturare l'intera località dei processi.

Esistono tre approcci per gestire il resident set: l'ALLOCAZIONE FISSA, l'ALLOCAZIONE VARIABILE e l'ALLOCAZIONE MISTA.

Allocazione fissa:

Il numero di frame fisici da riservare a uno specifico processo è fisso: quando è necessario effettuare uno swap-out, l'insieme delle pagine potentialmente sacrificabili per la sostituzione in RAM è limitato all'applicazione che causa il page fault.

Allocazione variabile:

Il numero di frame fisici da riservare a uno specifico processo può variare durante l'esecuzione: quando è necessario effettuare uno swap-out, l'insieme delle pagine sacrificabili per la sostituzione in RAM è esteso a tutte le applicazioni. In tal modo si ha un migliore utilizzo della memoria ma anche una maggiore complessità della sua gestione.

Allocazione mista:

Quando un processo viene attivato, si alloca per lui un certo numero di frame fisici e, all'atto di un page fault, la vittima da sostituire in RAM viene selezionata tra le pagine di questa stessa applicazione. Tuttavia, la taglia del resident set dei processi attivi viene ricalcolata periodicamente e viene eventualmente aumentata quella delle applicazioni con meno località. In particolare, per misurare la località, possono essere adottati due metodi:

- 1) OSSERVAZIONE DELLA FREQUENZA DEI PAGE FAULT: il resident set di un processo con la fre-

te: quando il page fault sotto una soglia minima viene restituito a favore di un'altra applicazione con l'alto sopra di una soglia massima.

Si tratta di un approccio limitativo poiché vengono considerati esclusivamente i page fault e non altri fenomeni come gli accessi effettivi alle varie pagine logiche.

2) WORKING SET: si definisce working set di un processo al tempo t l'insieme $W(t, \Delta)$ costituito dalle pagine accedute almeno una volta negli ultimi Δ riferimenti.

Un'applicazione ha correntemente una località massima se, nelle ultime Δ iterazioni, è stata referenzialmente sempre la stessa pagina, per cui $W(t, \Delta) = 1$.

All'opposto, un'applicazione ha correntemente una località minima se, nelle ultime Δ iterazioni, sono state referenziate Δ pagine diverse, per cui $W(t, \Delta) = \Delta$.

L'idea è quella di riportare periodicamente la taglia del resident set al valore del working set.

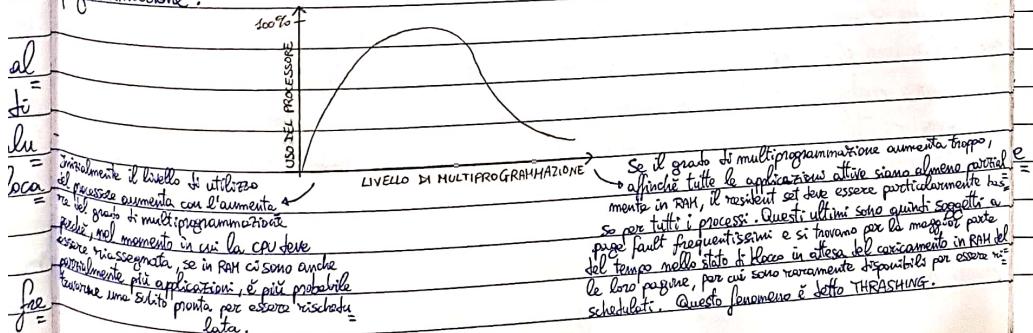
PROBLEMI:

• Scegliere un valore di Δ opportuno è difficile: se è troppo elevato, possono essere catturate anche delle pagine che sono uscite dalla località; se invece è troppo basso, la località potrebbe non essere catturata del tutto.

• È necessario tenere traccia di tutti gli ultimi Δ riferimenti ma, come già accennato a proposito dell'algoritmo LRU per la selezione della vittima, è un'attività complicata se non si ha un page fault.

3) Thrashing:

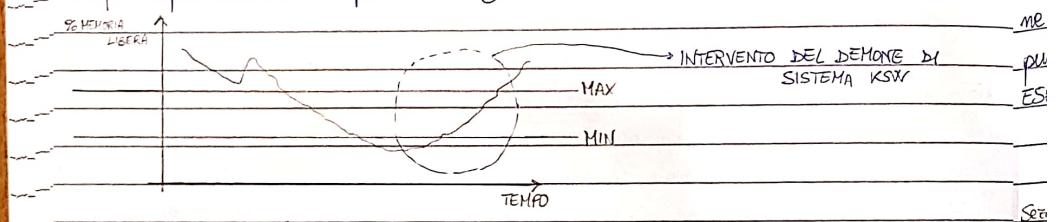
Analizziamo il comportamento del livello di utilizzo del processore al variare del grado di multiprogrammazione:



Sostituzione delle pagine in sistemi UNIX:

- Algoritmo dell'orologio per la selezione della vittima da swappare fuori memoria
- Allocazione variabile

→ tramite il demone di sistema KSW, il software libera dei frame fisici ~~o~~ ogni volta che ~~sto~~ la percentuale di memoria libera scende sotto una soglia minima per cercare di riportarla ~~Si~~ questa percentuale al di sopra di una soglia massima.



Sostituzione delle pagine in sistemi Windows:

- Algoritmo di tipo Not Recently Used per la selezione della vittima da swappare fuori memoria
- Allocazione fissa

→ Se la memoria libera scende sotto una soglia minima, il gestore libera alcuni frame.

→ Se la memoria libera supera una soglia massima, il resident set di qualche applicazione viene aumentato.

Altri aspetti della memoria condivisa:

Sia su UNIX sia su Windows (rispettivamente tramite le funzioni `shmget()` e `CreateFileMapping()`) è possibile allocare dei frame fisici in modo tale che siano condivisibili da più processi anche se non sono relativi tra loro. In questo modo, in particolare su UNIX, viene implementata una entry nella IPICT (Inter Process Communication Table) con un descrittore per la memoria condivisa, esattamente come avviene per le code di messaggi; l'altra parte, su Windows, si utilizzano ancora una volta gli handle, come per tutte le strutture dati e gli oggetti visti finora.

Paginazione a livelli multipli:

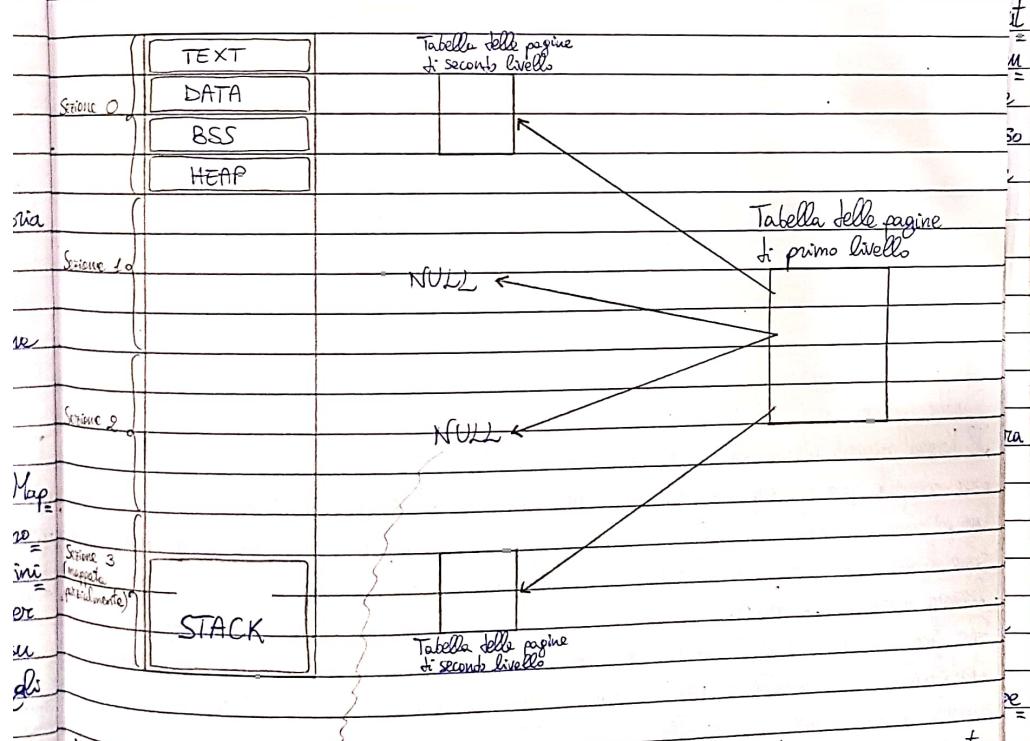
Finita abbiamo visto una tabella delle pagine come un array composto da tante entry quanti

te sono le pagine all'interno di un address space, indipendentemente dal fatto che siano mappate o meno. Questo comporta un utilizzo inefficiente della memoria a livello Kernel.

È possibile far di meglio introducendo la paginazione a livelli multipli: l'address space viene suddiviso in sezioni, ciascuna delle quali è composta da un certo numero di pagine.

Si ha una tabella delle pagine di primo livello, in cui la i -esima entry corrisponde alla i -esima sezione. Solo se quest'ultima ha almeno una pagina logica materializzata in RAM, allora viene allocata a livello Kernel una tabella delle pagine di secondo livello raggiungibile tramite un puntatore da quella di primo livello.

ESEMPIO:



L'istruzione di una tabella di secondo livello non avviene nel momento in cui viene mappata la prima pagina della sezione corrispondente, bensì quando viene materializzata a seguito di un fault.

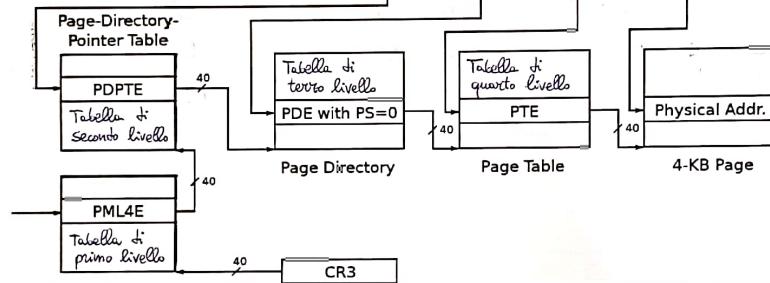
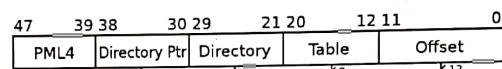
Gli indirizzi logici del contenitore di memoria sono espressi nel seguente modo:

SEZIONE PAGINA OFFSET.

Schema di paginazione su x86-64 (long mode)

(A 4 LIVELLI)

Linear Address



Buddy system:

Il Kernel riserva alcuni frame per se stesso e identifica in loro quali sono le aree libere e le aree occupate per scrivere eventualmente nuove strutture dati e nuovi contenuti di livello Kernel (un po' come fa malloc()). Tuttavia, oltre a questo, deve anche tenere traccia di quali frame sono liberi e quali no per allocare sia le informazioni di livello Kernel ma anche per materializzare nuove pagine logiche.

Per fare ciò, viene adottato il cosiddetto BUDDY SYSTEM, in cui ogni porzione della RAM viene identificata dinamicamente o come un frame singolo o come un insieme contiguo composto da un numero di frame pari a una potenza di due. Tale rappresentazione può essere implementata con un semplice albero, anche se nella realtà si ricorre a strutture molto più complesse.

ESEMPIO:

