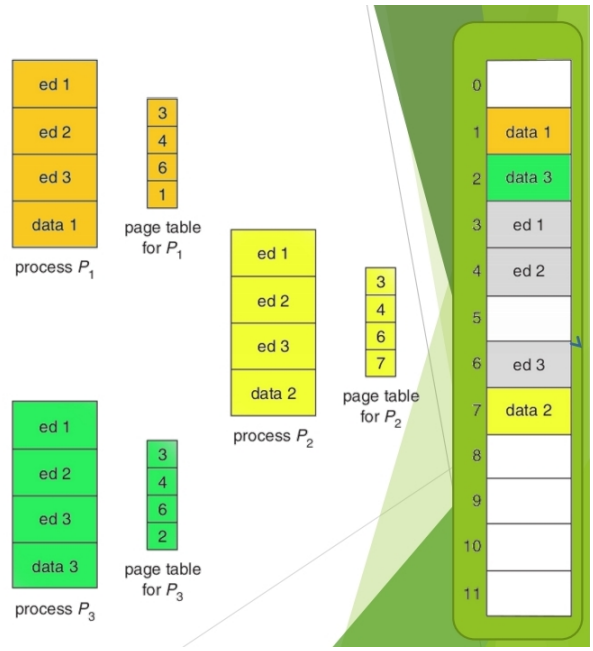
Pagine condivise

E' possibile condividere del codice che deve essere ***rientrante***, cioè non che non cambia durante l'esecuzione. Se non cambia mai allora si può condividere in ***modalità LETTURA*** con i processi che lo richiedono.



> `ed1` = editor 1, `data1` = dati 1

Si condivide tutto ciò che è condivisibile mentre i dati per ogni singolo utente saranno indipendenti gli uni dagli altri.

> Una sola copia di un editor (ed) sarà in memoria.

Gestione della cache

Se lavora su indirizzi logici, è direttamente lei che comunica con la CPU. ALtrimenti ci deve essere MMU che traduce gli indirizzi.

Quando si usa una cache con indirizzi virtuali, allora rende la ricerca al suo interno molto più veloce visto che non serve passare dalla MMU per determinare l'indirizzo fisico corrispondende a uno specifico indirizzo virtuale.

Problema degli ASID

Nella parte della convivisione, diversi indirizzi virtuali, fanno riferimento ad un unico indirizzo fisico. In memoria si ha un unico blocco al quale fanno riferimento diversi processi.

> *Quindi diversi indirizzi virtuali puntano allo stesso indirizzo fisico*

Potrebbe sorgere un **problema di coerenza** dovuto proprio a questo aspetto appena descritto. Il problema di incoerenza è detto ***ALIASING***

- Per risolvere questo problema si può usare una cache *Virtually Indexed Phisically Tagged* (*VIPT*), cache che **fa uso di indirizzi virtuali e tag fisici**. Indirizzo virtuale come indice di rircerca e indirizzo

fisico come tag.

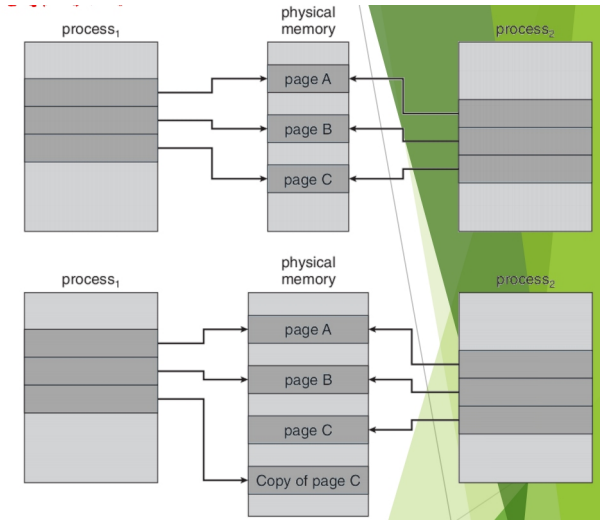
- ***VANTAGGI***: Rispetto a una cache con indirizzi fisici, la latenza è inferiore perchè ***si può eseguire in parallelo con la ricerca della TLB***
- *Il tag non pulò essere confrontato finchè non sarà trovato un indirizzo virtuale corrispondente*

> *Nel caso di cache con indirizzi fisici, per capire se si tratta di un duplicato si deve aspettare che la TLB dia in output lindirizzo fisico*
> Nel caso di cache con indirizzi virtuali, si riescono a individuare i duplicato che puntano allo stesso frame in maniera molto più velocemente

Se uso la **tabella delle pagine invertite**, allora la memoria condivisa è implementata in maniera tale che non viene considerata l'ipotesi "Più indirizzi virtuali corrispondono a un unico frame" ma viene assunto che la **connessione *virtuale-frame* è 1 a 1 e basta.**

- Questo approccio funziona in maniera semplice quando si ha un **SINGOLO CORE**
- Diventa più complessi quando si ha un sistema **MULTICORE** (*MULTIPROCESSORE*)

Visto che **l'associazione indirizzo virtuale-frame è 1 a 1**, quando gli altri indirizzi virtuali fanno riferimento allo stesso frame e si tenterà di accedervi, ***avverrà un page fault***, ma in questo caso, il SO capisce che l'indirizzo è semplicemente **mappato da un altro indirizzo virtuale**



Se **accadono delle modifiche alla pagina condivisa** durante l'esecuzione dei processi (anche se un processo figlio modifica un file condiviso `fork()`), visto che le pagine vengono etichettate di *sola lettura*, si riceverà un errore e si ha una TRAP che passa al SO:

- Dopo la TRAP, si devono creare necessariamente 2 copie del file condiviso per i processi che lo devono modificare. In questo caso il processo padre/figlio avranno copie di file diverse, ma non conterranno l'intero blocco di memoria (perchè è stata modificata solo una parte del file condiviso e quindi non ha senso copiare di nuovo la parte non modificata) ma si usa la tecnica ***COPY-ON-WRITE***: le porzioni inalterate continueranno a essere condivise e verranno copiate fra i 2 processi solo le parti del file che sono state modificate. (Si ***condivide finchè è possibile***)

Da qui nasce un altro problema: la *parte da copiare* (che dovrà essere modificata) *dovrà copiare questi dati in un nuovo blocco di memoria* del nuovo processo, che dovrà essere cercato. ***Trovare blocchi liberi*** è proprio il problema che segue.

Molti SO, per migliorare le prestazioni complessive, tengono un ***POOL DI FRAME LIBERI*** in modo tale da usarli proprio in casi come questi e avere già a disposizione i frame disponibili per l'occupazione.

I *"blocchi liberi" non sono effettivamente liberi* perchè i dati sono comunque li ma non sono raggiungibili. Per svuotare un blocco di memoria si usa la tecnica ***ZERO-FILL-ON-DEMAND*** che azzera i blocchi prima che vengano assegnati nuovamente. Vengono cancellati, attraverso una sequenza di zeri, i contenuti precedenti del frame stesso.

VANTAGGI:

- questo azzeramento dei dati viene gestita dal kernel. Ne segue che vi è una **maggiore efficienza** nell'azzeramento dei blocchi
- incrementa la sicurezza perchè si garantisce che il blocco sia effettivamente libero
- *può avvenire in qualsiasi momento*, cioè quando **il sistema è inattivo**. Quindi non avviene proprio quando si ha necessità

Librerie condivise

Ci sono 2 modi di gestire i collegamenti alle librerie:

- ***LINKING STATICO***: Le librerie vengono incluse nel codice **in fase di generazione dell'eseguibile**, quindi nella sua immagine binaria di per sè
- ***LINKING DINAMICO***: Il collegamento viene posticipato **in fase di esecuzione**. All'interno dell'eseguibile viene messo uno stack che indica *come individuare la relativa libreria richiesta*, se è già in memoria, oppure indica *come deve essere caricata*
  - In questo caso si *risparmia sullo spazio in RAM e su disco*
  - L'approccio dinamico porta vantaggi anche quando si aggiornano le librerie. Si deve trovare un modo per distinguere le versioni delle librerie. Si usa un contatore: quando gli aggiornamento sono superiori rispetto a una certa soglia, questo contatore viene incrementato di 1. In tal caso ci si deve riferire alle nuovi versioni di libreria ma comunque i programmi potranno scegliere quale versione di libreria usare. Questo meccanismo prende il nome di ***LIBRERIE CONDIVISE***

File mappati in memoria

Le librerie condivise fanno parte di una struttura molto più generica detta "***FILE MAPPATI IN MEMORIA***"

Ogni volta che si fa una chiamata di sistema significa che si deve accedere al disco, quindi il sistema rallenta.

Per migliorare le prestazioni in geenrale è quello di **trattare i file I/O in modo alternativo**, cioè come se fossero dei classici accessi di routine alla memoria. Vuol dire ***associare una parte dello spazio degli indirizzi virtuali al file.***

- Questo approccio si chiama ***MAPPATURA IN MEMORIA DI UN FILE***: si mappa il blocco del disco a una o a più pagine mediante *demand-paging* (***paginazione su richiesta***: cioè che all'inizio del programma si hanno tutti i page fault quindi verranno creati i link fra indirizzi virtuali e frame)

> Usando questo approccio si ha ***un accesso più veloce al file***

- Queste operazioni (lettura, e in particolar modo la scrittura) non sono sincrone ma possono avvenire in base a determinate condizioni

Mappatura

Essa avviene una specifica chiamata di sistema (`mmap`).
In generale, diversi SO, a prescindere se è richiesta o meno, per migliorare le prestazioni, operano per default ed effettuano la mappatura del file in memoria senza considerare se è stata eseguita una `mmap` o meno.

> *E' possibile consentire a più processi di mappare lo stesso file allo stesso momento. In questo caso, un'eventuale modifica è visualizzabile da tutti gli altri processi che mappano la stessa copia del file*

La condivisione dei file mappati in memoria è simile e analogo alla memoria condivisa. Molti SO usano diverse chiamate di sistema per diverse operazioni (`mmap()`)
La memoria condivisa si avrà usando le chiamate di sistema `shmget()` (*share memory get) e `shmat()` per agganciare la memorie condivisa.

Allocazione memoria per il kernel

- In generale, la memoria che è unica, serve per le operazioni di sistema gestite dal kernel.
- La suddivisione dei blocchi deve essere in parte per l'utente (paginata ma con frammentazione interna) e in parte per le attività di sistema.
- Un processo eseguito dal kernel è diverso da un processo utente.
- Per questo motivo si ha che le due porzioni sono separate

In generale:

- Tenendo conto che metà di una pagina è sprecata, viene mantenuta una porzione di memoria per le attività del kernel per alcuni motivi:
- Il kernel richiede l'uso di strutture dati di vario tipo che sono spesso molto più piccoli (in dimensioni) di una pagina. Quindi gestirle attraverso una pagina porterebbe ad avere grossa frammentazione interna
- Il kernel cerca di occupare la memoria riducendo il più possibile la frammentazione. Per questo motivo il kernel si ***AUTOGESTIONA***
  - Le pagina sono assegnate a frame disposti anche in maniera non contigua
  - In memoria, invece, le informazioni sono disposti in maniera contigua per accelerare le operazioni
  - Visto che si devono cercare parti di memoria non contigui, allora il kernel cerca blocchi contigui per sbrigarsi. Allora, in alcuni casi, ***si ha un'impossibilità di paginare l'allocazione*** e per questo motivo il kernel gestisce in maniera diversa queste situazioni rispetto all'utente

Per allocare memoria al kernel si usa il metod ***SLAB ALLOCATION***: si alloca in maniera efficiente ed è i ngrado di ridurre la frammentazione interna

E' formato da 2 parti:

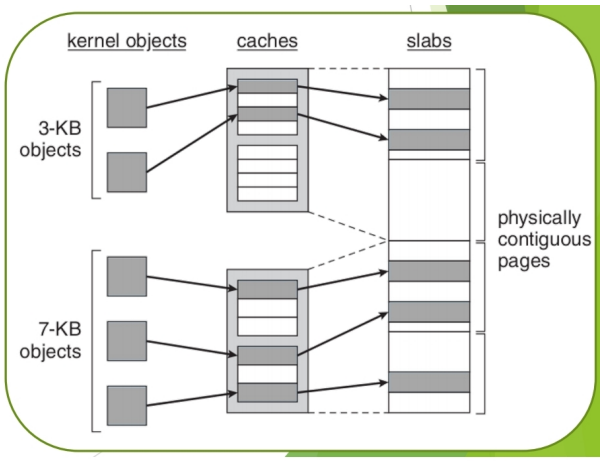
- ***slab***: sequenzza di una o più pagine **fisicamente contigue**
- ***cache***: contiene *una o più slab*
- Ogni cache esiste per ciascuna struttura dati usata dal kernel (*cache per il descrittore dei processi, cache per i semafori ecc...*) ed è popolata da slab che sono istanze delle struttre dati dello **stesso tipo**.

Lo slab allocation usa le cache per memorizzare oggetti del kernel, appunto:

- quando viene creata una cache omogenea, inizialmente molti oggetti allocati vengono contrassegnati come "liberi". Il numero di oggetti nella cache dipende dalla sua dimensione
- Una volta assegnato uno slab, esso diventa "*utilizzato*"

Esempio:

- slab da 12K
- Composto da 3 pagine e ognuno formato da 4K
- Quanti oggetti si possono memorizzare? 6 oggetti da 2K



Gli slab si possono trovare in **3 stati diversi**:

- ***PIENO***: tutti gli oggetti dello slab sono contrassegnati come "occupati" (o "usati")
- ***VUOTO***: tutti gli oggetti dello slab sono contrassegnati come "liberi"
- ***PARZIALE***: alcuni oggetti dello slab sono contrassegnati come "liberi" e altri come "occupati"

In base allo stato dello slab:

- quando si riceve la richiesta di trovare un oggetto vuoto, viene ricercato all'interno dell'insieme degli slab che sono nello stato **PARZIALE**
- se non ci sono slab nello stato parziale si passa allo stato **VUOTO**
- se non ci sono nemmeno questa volta, allora si deve ***ALLOCARE UN NUOVO SLAB*** e verrà assegnato ad una determinata cache allocando *pagine fisicamente contigue*

Vantaggi dell'algoritmo dello Slab Allocation

- ***NESSUNA MEMORIA VIENE SPRECATA*** per frammentazione: il kernel richiede ***la quantità esatta di memoria*** per quell'oggetto
- le richieste di memoria possono essere eseguite molto rapidamente (**efficienza**). L'efficienza è più elevata quando si hanno **frequenti richieste di allocazione e deallocazione**

File System

La parte del SO che si occupa dei dati memorizzati viene detto ***FILE SYSTEM***. Essa gestisce la creazione, modifica, lettura, modo di accesso ecc...

I file rappresentano sia **programmi** (codice ecc..) e sia **dati**. E' una sequenza di byte o di record il cui significato lo da il creatore.
Ogni file ha deti dettagli:

- tipo di file
- nomenclatura
- posizione nel file system
- dimensione del file (data in byte o *in blocchi*)
- i tipi di accessi permessi a tale file
- data di creazione/modifica con il relativo proprietario

La **struttura di directory** risiede **nella memoria secondaria** e tiene tutte le informazioni relative ai file. Dall'**identificatore del file** si riesce a risalire a tutti gli **attributi del file**.

> *La directory è un file speciale che può contenere altre directory o file*

Le operazioni possibili su un file sono: lettura, scrittura, esecuzione, creare, spostamento.

Problema: (*da approfondire* da altri appunti)

- Tutte le operazioni possono essere fatte **contemporaneamente da diversi processi**
- In questo caso, quando più processi operano all'interno di un file, si usa una **tabella a 2 livelli**:
  - **un livello** viene usata per ogni processo (tiene conto del numero di processi che usano quel file) e punta a una tabella di file a un'altra livello di sistema
  - **nel secondo livello** usa una *tabella a livello di sistema* (contiene le informazioni indipendenti del processo).
- Ogni volta che viene eseguita una open() viene incrementato un contatore all'interno del file che tiene conto del numero di aperture. Di conseguenza tiene traccia di quanti processi stanno usando quel file. (in caso di close() il contatore viene decrementato)

Alcuni SO usano dei meccanismi per bloccare un file quando è aperto, quindi evitare il suo uso da altri processi. Si usano i **LOCK**. Possono essere:

- ***shared***: possono essere acquisiti allo stesso momento da più processi
- ***exclusive***: quando si ottiene lock di scrittura (se il file è libero da qualsiasi altro lock scrittura) allora nessun altro può acquisire un lock di scrittura finchè il primo processo non termina
- ***mandatory***: il SO impedisce a qualsiasi altro processo di poter accedere al file bloccato mentre un altro ha il lock i scrittura
- ***advisory***: è un blocco di avviso. SO non blocca l'acquisiione di lock da parte di un processo

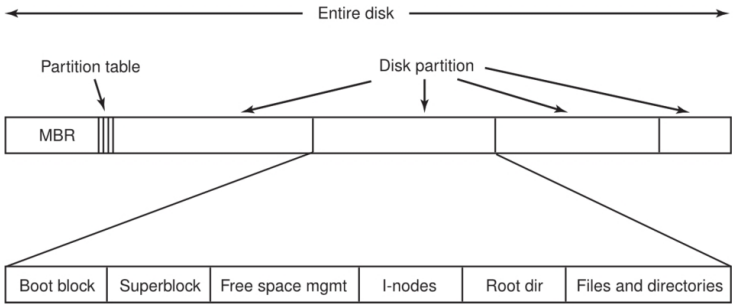
> *Molti SO usano i lock esclusivi sui file*

Possono esserci anche file **REGOLARI** binari o ASCII e **SPECIALE** a carattere o a blocchi

Struttura file system

Un disco è diviso da una o più ***partizioni*** e in ognuna di essa si installano SO diversi e ne segue che i file system sono diversi.

La parte iniziale è sempre il settore 0, chiamato ***MBR*** (*Master Boot Record*). Contiene una serie di tabella relative alle partizioni (vuote o contenenti altri file system) e indicano gli indirizzi iniziali/finali delle partizioni. Fra tutte le partizioni, una è segnata come ***ATTIVA*** ed è quella che verrà presa in considerazione all'avvio della macchina



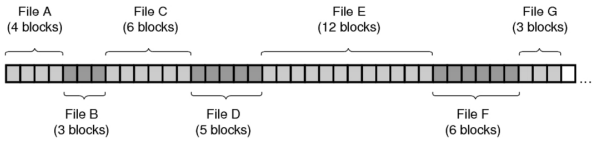
Una partizione del disco contiene:

- BLOCCO DI AVVIO
- ***SUPERBLOCCO***: contiene tutti i parametri necessari al file system per l'avvio: il tipo di file system e il numero di blocchi che ha e altre informazioni amministrative
- ***BLOCCHI LIBERI DI FILE SYSTEM***: sono blocchi liberi soottoforma di **bitmap** o di **puntatori**
- ***I-NODE***: è un array, una per ciascun file, che contiene tutte le info relative al file stesso, tranne il suo contenuto
- ***ROOT DIRECTORY***: è la radice dell'albero del file system
- ***FILES E DIRECTORY*** del file system

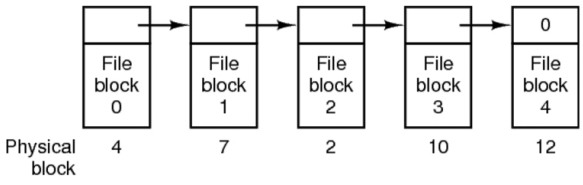
Implementazione file

l'implementazione può avvenire mediante allocazione:

- **contigua**:
  - Ha un limite: l'**accesso** al blocco è **diretto**, quindi **molto veloce**. Siccome si va a blocchi contigui, questo è ***soggetto a frammentazione interna/esterna***



- **con le liste concatenate**: contiene un puntatore al successivo nodo e una parte dedicata ai file. In questo caso si ha che:
  - Il problema della frammentazione interna rimane ma si evita la frammentazione esterna. La ricerca **non è diretta** e quest'operazione è molto **lenta**



La tabella di allocazione dei file *FAT* viene caricata per andare interamente in memoria. Quindi:

- non richiede accessi al disco visto che già si trova in memoria
- occupa spazio in memoria, chiaramente (ma non è di grandi dimensioni, quindi è *trascurabile*)
- il blocco di dati è sempre disponibile mediante puntatori
- E' più usato nei sistemi Windows mentre in Linux si usa I-Node che contiene le info inerenti a un file e ha il seguente vantaggio: tiene traccia mediante un indice dei vari blocchi di un file. Rispetto al FAT risulta **più piccola in dimensioni**