Commenti generali

- **Allocazione concatenata**: più adeguata a accessi sequenziali
- **Allocazione a indice**: più adeguata ad accessi diretti
- L'allocazione indicizzata richiede di mantenere in RAM una parte dei blocchi indice, possono occorre due o più accessi al disco se la RAM non è sufficiente:
  - uno (o più) per accedere al blocco indice giusto
  - uno per raggiungere il dato di interesse
- Alcuni sistemi combinano allocazione contigua e indicizzata: finché il file rimane di piccole dimensioni si usa l'allocazione contigua, oltre un certo limite si comincia ad usare un indice

Gestione dello spazio libero

- Occorre una struttura che consente di tener traccia dei blocchi liberi
- **Vettore di bit**: viene mantenuto un array di bit, ognuno dei quali corrisponde a un blocco. Se il blocco è libero il bit è impostato a 1, es:

  0 0 0 1 1 0 1 1 1 1 0 1 0 1 1 1 0 0 0 1 1 0

  il quarto e quinto blocco sono liberi, idem dal settimo al decimo e così via

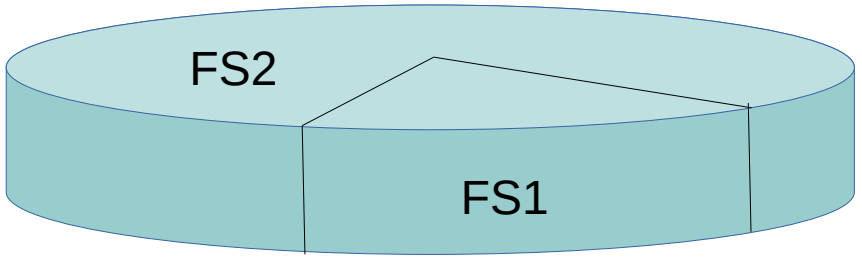
- NB: con questa soluzione è anche facile identificare sequenze di blocchi contigui liberi. Se so che mi servono tre blocchi liberi e li trovo contigui, meglio, perché i tempi di accesso saranno inferiori

Gestione dello spazio libero

- **Lista concatenata**: i blocchi liberi sono concatenati in una lista. Poiché i blocchi vengono allocati ai file uno per volta, basta pescare dalla testa della lista il primo blocco libero ed aggiornare il puntatore
- **Raggruppamento**: concatenazione di blocchi indice. Si usa un blocco libero (di dimensione N) per mantenere N-1 puntatori ad altrettanti blocchi liberi. Si usa l'ultimo puntatore per fare riferimento a un eventuale ulteriore blocco dello stesso tipo
- **Conteggio**: variante del precedente, in presenza di sequenze di blocchi liberi contigui si mantiene un riferimento al primo di tali blocchi e il numero di blocchi liberi ad esso consecutivi. Es. <K, 4> dove K è il riferimento a un blocco libero e 4 il numero di blocchi liberi consecutivi a partire da esso

Quantità di memoria massima gestibile

- La quantità massima di memoria gestibile da un file system dipende dalle strutture del file system
- **Esempio**: una FAT a 8bit consente di indicizzare al più 2 8 blocchi (256 blocchi), se ogni blocco è grande 1KB potrò al più gestire una memoria pari a 256KB
- Se devo gestire un disco di dimensioni superiori alla quantità di memoria gestibile da un FS dovrò partizionare il disco in FS diversi o una parte della memoria sarà inaccessibile!!

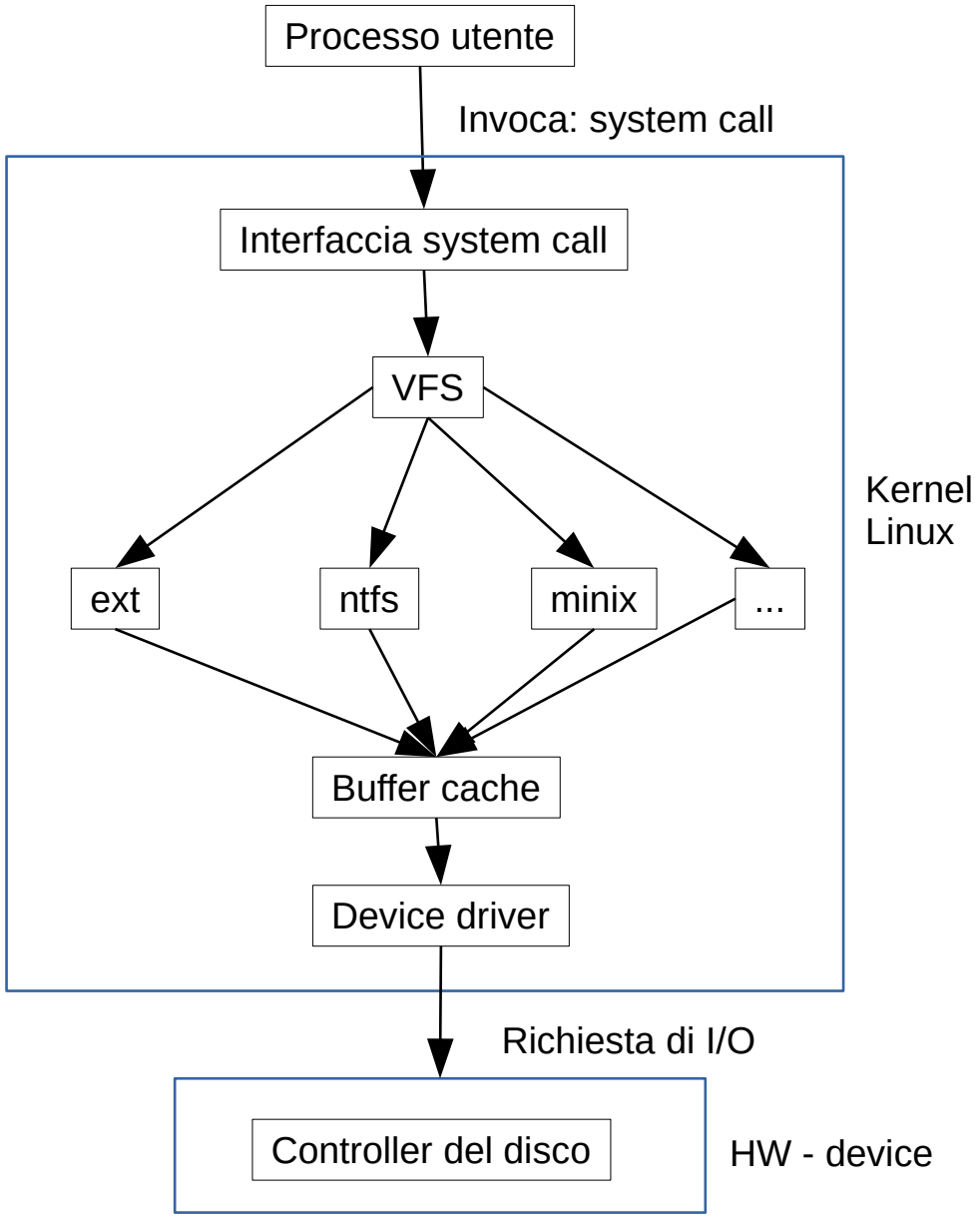


File System Virtuale

- Un disco può essere partizionato e ogni partizione può contenere FS di tipodifferente, eppure è possibile montare i diversi FS in un'unica struttura a cui l'utente ha accesso attraverso lo stesso insieme di comandi e system call, senza accorgersi delle diverse implementazioni
- Analogamente è possibile accedere a FS di rete senza rendersi conto che i file usati risiedono su una macchina diversa da quella in uso

- Ciò è possibile perché l'interazione fra i processi e il FS è mediata da un'**interfaccia astratta** che permette di prescindere dallo specifico tipo di FS usato
- Questa interfaccia è il **Virtual File System**

File system virtuale



I processi utente invocano delle system call offerte dal VFS

Il VFS sta a un livello di astrazione più alto dei FS veri e propri ed esegue quella parte del lavoro che è indipendente dalla realizzazione fisica

Il VFS è un livello di indirezione che gestisce le sys. call che lavorano su file, preoccupandosi di richiamare le necessarie routine dello specifico FS fisico per attuare l'I/O richiesto

I diversi FS fisici devono specificare l'implementazione di tutte le sys. call offerte dal VFS

Oggetti di un VFS

- Il VFS gestisce **vnode**:
  un vnode è un'**astrazione degli inode**, rappresenta un file, una directory, un link, un socket, un block o un character device, ecc.
- **NB**: mentre un inode ha un identificatore numerico unico per uno specifico FS, un vnode ha un id numerico unico per tutto l'insieme dei FS montati
- I vnode possono essere di diversi tipi a seconda di ciò che rappresentano, per esempio:
  - **superblock**: rappresenta un intero FS
  - **dentry**: rappresenta un generico elemento di una directory
  - **inode**: rappresenta un generico file
  - ...

Operazioni consentite da un VFS

- Un VFS definisce e mette a disposizione del livello utente un insieme di operazioni (**system call**) di base, fra cui troviamo per esempio:
  - open di un file
  - close di un file
  - read/write di un file
- uno specifico FS fisico, es. ext3 o winFS, per essere gestibile attraverso il VFS deve implementare tali funzioni, esattamente come accade per le interfacce in un linguaggio object-oriented.
- L'implementazione è talvolta indicata con il termine: **driver del file system**

Buffer-cache?

- Occorre fare una distinzione fra due tipi di memorie d'appoggio, definiti in base all'uso che se ne fa:
  - **buffer**: è una memoria in cui i dati vengono inseriti in attesa di essere consumati. I dati in un buffer vengono usati una volta sola.
  - **cache**: è una memoria in cui i dati vengono conservati per un po' di tempo in previsione di ulteriori utilizzi

- I dischi hanno associata una **memoria di appoggio in RAM** detta **buffer cache** ...

Buffer Cache

- **Passi di una scrittura su file**:
  - **Processo**: esegue una **write**
  - Il blocco da modificare, appartenente al file in questione, viene aggiornato **in RAM**
  - La modifica viene **propagata** al blocco del file conservato su disco
- Normalmente un'operazione di scrittura su di un file si conclude solo quando la modifica su disco è stata completata
- **Write-ahead**: meccanismo tramite il quale un'operazione di scrittura viene considerata conclusa al termine dell'aggiornamento della copia del blocco *contenuta in RAM*

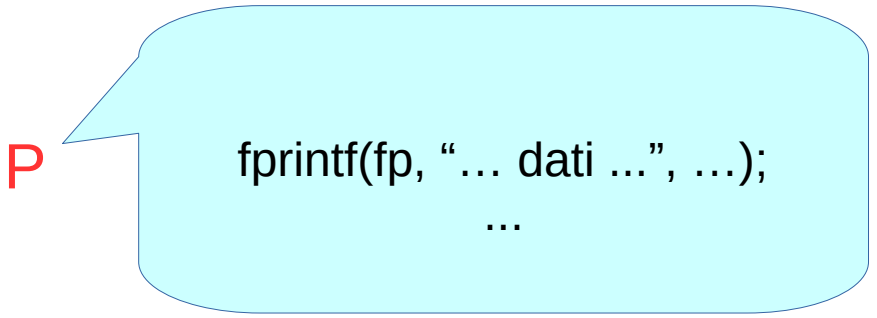
Read ahead – read behind

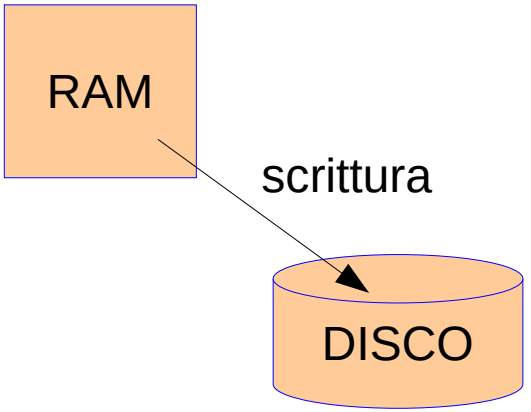
Quando il controller riceve una richiesta di lettura, individua un settore del disco e poi lo legge tutto (non solo il dato richiesto). La parte extra è inserita nella RAM e lasciata a disposizione per usi futuri.

Questo approccio applica una sorta di criterio di località negli accessi al disco è un'attività nota come **read ahead** o **read behind**

Buffer Cache

- **Perché il write ahead incrementa l'efficienza?**
- Consideriamo un processo che esegue la scrittura senza write ahead:

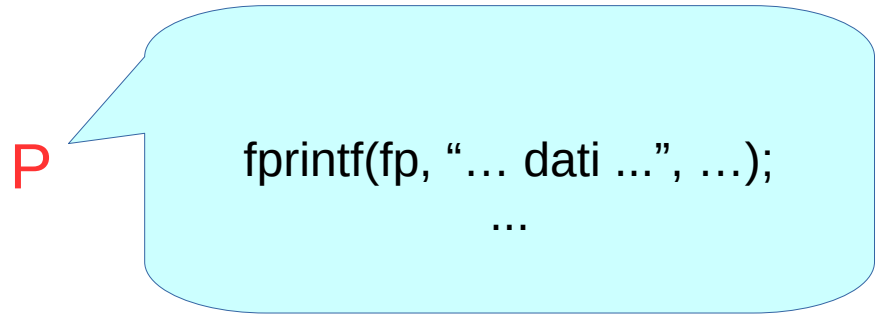


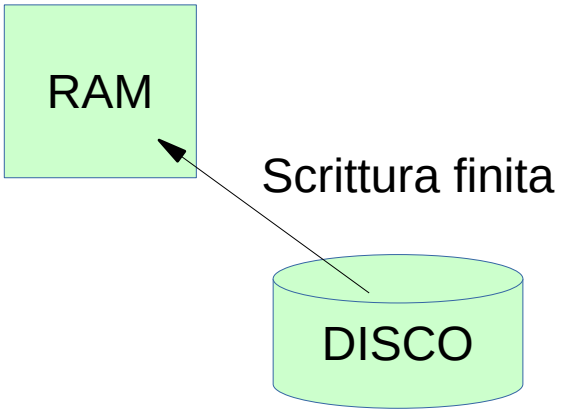


- **Per eseguire questo codice il processo è nello stato RUNNING**
- **La scrittura è un'operazione di output, come tutte le operazioni di I/O fa cambiare lo stato del processo in WAITING**

Buffer Cache

- **Perché il write ahead incrementa l'efficienza?**
- Consideriamo un processo che esegue la scrittura senza write ahead:

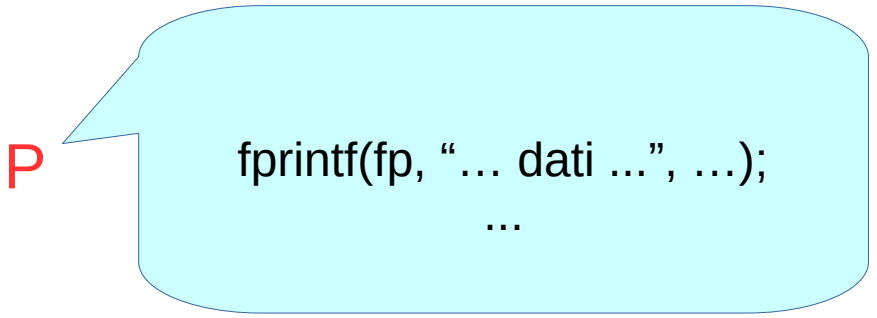




- **P rimane WAITING finché non termina la scrittura su disco**
- **Dopo P diventa READY**
- **Infine dopo un tempo che dipende dallo scheduling della CPU, P torna RUNNING e prosegue la sua esecuzione**

Buffer Cache

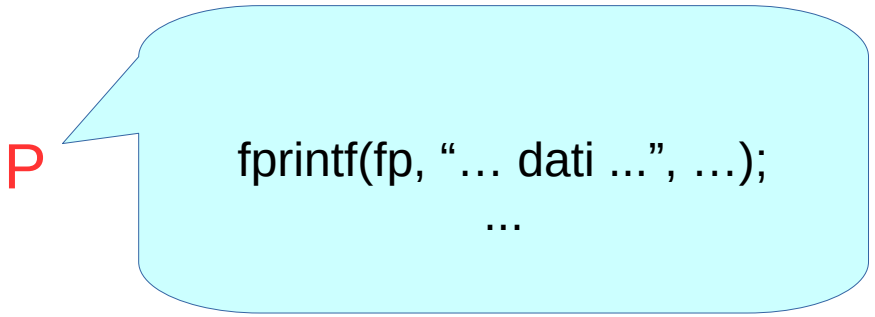
- **Perché il write ahead incrementa l'efficienza?**
- Consideriamo lo stesso processo che esegue la scrittura con il write ahead:

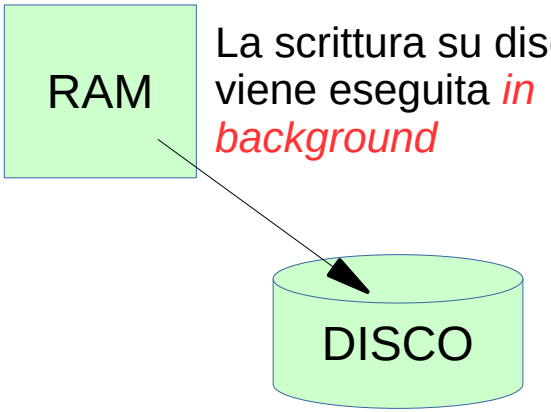


  - **Per eseguire questo codice il processo è nello stato RUNNING**
  - **Poiché si usa il write-ahead, La scrittura è considerata terminata quando è completata in RAM**
  - **Il processo continua la sua esecuzione senza cambiare stato**

Buffer Cache

- **Perché il write ahead incrementa l'efficienza?**
- Consideriamo lo stesso processo che esegue la scrittura con il write ahead:



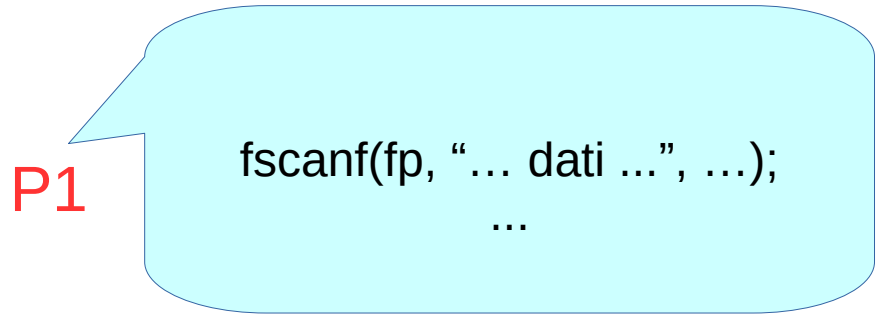


- **Per eseguire questo codice il processo è nello stato RUNNING**
- **Poiché si usa il write-ahead, La scrittura è considerata terminata quando è completata in RAM**
- **Il processo continua la sua esecuzione senza cambiare stato**

È necessario che l'interazione di scrittura con il device non richieda l'intervento della CPU (es. DMA)

Buffer Cache

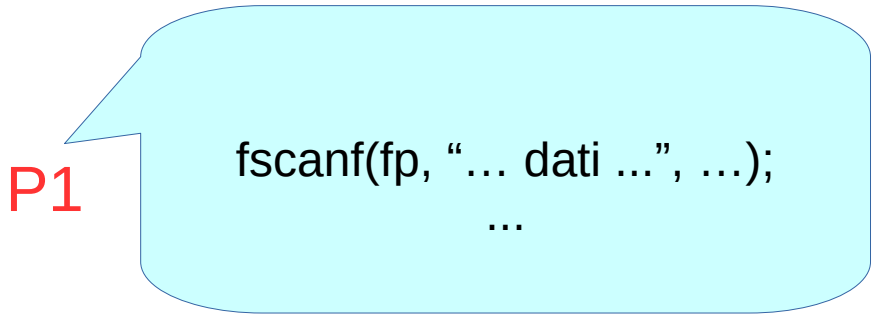
- **Perché il buffer cache incrementa l'efficienza?**
- Consideriamo un processo che voglia leggere il file prima scritto da P:



  - **Per eseguire questo codice il processo è nello stato RUNNING**
  - **La lettura è un'operazione di input, dovrebbe far cambiare lo stato del processo a WAITING**

Buffer Cache

- **Perché il buffer cache incrementa l'efficienza?**
- Consideriamo un processo che voglia leggere il file prima scritto da P:



  - **Poiché si usa il buffer cache, i blocchi dati modificati sono in RAM**
  - **Il processo vi accede e continua la sua esecuzione senza cambiare stato**

Esempio

- Un utente scrive il programma **codice.c** usando **editor di testo** e **compilatore**. Supponiamo che l'utente modifichi il file, lo salvi e chiuda l'editor e solo dopo compili il file modificato:
  - L'**editor** è un esempio di processo che scrive un file
  - senza write ahead, ad ogni *save* l'editor rimarrebbe WAITING fino al termine della copiatura su disco
  - Con il write ahead a ogni save l'editor consente all'utente di continuare a lavorare non appena la richiesta di scrittura è stata notificata al controller del disco
  - La scrittura su disco vera e propria avverrà in background

Esempio

- Un utente scrive il programma **codice.c** usando **editor di testo** e **compilatore**. Supponiamo che l'utente modifichi il file, lo salvi e chiuda l'editor e solo dopo compili il file modificato:
- Il **compilatore** è un esempio di programma che legge un file per elaborarlo. Il file risiede su disco ma la presenza del buffer cache rende più efficiente l'accesso ai blocchi dati:
  - se **codice.c** è appena stato salvato, anche se è stato chiuso dal processo editor, i suoi blocchi dati sono (probabilmente) ancora nel buffer cache
  - non serve aspettare il caricamento del file da disco per effettuare la compilazione

Ripristino del file system

- Quando si avvia un elaboratore, il SO compie una verifica della **consistenza del file system.** **FSCK** (**file system check**): è una routine che controlla se all'ultimo spegnimento il FS è stato smontato (operazione di unmount) correttamente.
- L'**unmount** forza l'esecuzione delle scritture pendenti

- Il modo in cui viene effettuato il controllo dipende dal FS. Es. in ext2 si verifica un campo del superblocco, che può assumere i valori **clean** e **unclean**

1) quando si fa il mount del FS il campo è settato ad unclean
2) quando si fa l'unmount viene settato a clean
3) se c'è un crash, il campo rimane unclean

Ripristino del file system

- Se fsck scopre che il FS non è stato smontato correttamente, avvia un controllo minuzioso (che non studiamo) del disco intero, individuando le inconsistenze e risolvendole per quanto possibile
- **La durata di questo controllo dipende dalla dimensione del disco**
- **PROBLEMI**:
  - Se il problema si verifica su un server che mantiene i dati di qualche servizio critico (es. banca, compagnia di volo, ...) il tempo trascorso nella verifica è un costo perché durante tutto il tempo il FS è off-line e non può essere usato
  - altro problema: transazioni. Se il crash ha interrotto una transazione, occorrerà avviare delle operazioni di ripristino di uno stato consistente

- **Soluzione**: mantenere qualche informazione in più che consenta di andare ad effettuare verifiche mirate; perché controllare porzioni di file modificate un'ora fa? Soltanto le modifiche effettuate poco prima del crash possono non essere state applicate ...

Journal

- Il **journal** è un **logfile** mantenuto su disco, in cui si tiene nota di tutte le operazioni di scrittura su file
- viene gestito in modo simile ai logfile per le transazioni quando un processo effettua una scrittura, prima il SO registra l'operazione che si va ad effettuare sul logfile, poi esegue la scrittura vera e propria del dato su file

Journaling del file system

- Le annotazioni presenti nel **journal** consentono di focalizzare il recovery sui soli file su cui sono state effettuate scritture **negli ultimi istanti prima del crash**
- Prima viene scritta l'annotazione nel journal, poi viene modificato il file oggetto della scrittura
- Consente di distinguere fra i seguenti casi, a seconda del momento in cui è avvenuto il disastro da recuperare:
  - **Crash successivo alla scrittura su disco del file**: l'annotazione nel journal corrisponde a quanto presente nel file, nessuna azione viene intrapresa
  - **Crash occorso fra la scrittura del journal e la scrittura su disco del file**: l'annotazione nel journal non corrisponde al contenuto del file, il contenuto del file viene reso consistente rispetto all'annotazione
  - **Crash occorso prima dell'annotazione nel journal**: la scrittura non ha mai avuto luogo

Journaling

Vantaggio

- La durata del ripristino non dipende dalle dimensioni del FS, bastano pochi secondi per riportare la consistenza dei dati

Svantaggio

- La gestione del journal appesantisce la gestione delle operazioni sui file. Nonostante ciò oggi come oggi i journal sono considerati essenziali