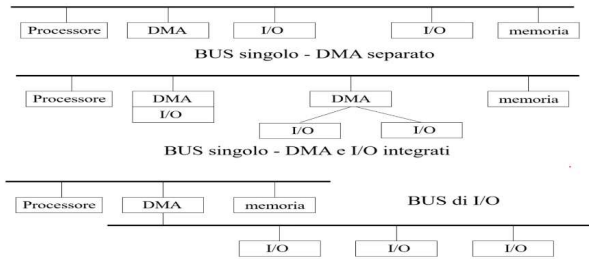
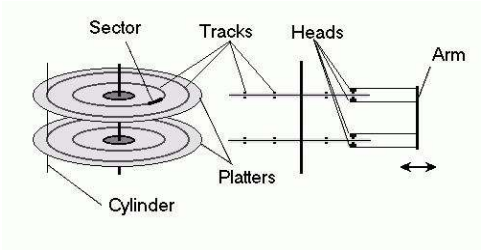
Virtual file system 7 I/O scheduling

martedì 11 novembre 2025 12:49

Andiamo ora a vedere la pianificazione tramite la quale i dispositivi i/o vengono attivati per le loro operazioni. Di solito per queste operazioni si usa uno scheduling **fcfs** (come per esempio i terminali -> non effettuo lettura anticipate scrittura ritardata e non ho buffer cache) . Ma attenzione che questi scheduling non sono sempre usate. Vediamo ora la "rete" che permette questa interazione tra dispositivi e memoria:



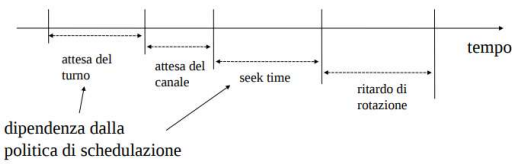
Dove per i/o si intende un reale dispositivo , mentre il DMA è il dispositivo che ci permette di trasferire dati da dispositivo a memoria. Dato l'unico bus nella prima soluzione si potrebbe avere lo stallo . Nella prima variante (DMA separato) per ogni classe di dispositivo si ha un opportuno DMA , mentre per l'ultima (Bus di i/o) soluzione attuale si ha che si hanno due bus diretti uno per interazione tra processore DMA e memoria ed un altro che serve solo per il passaggio di dati con i dispositivi. Vediamo ora i principali dispositivi di i/o : **i dispositivi di massa** ovevro dischi magnetici o SSD. I primi sono dispositivi elettro-magnetici ; buon compromesso tra velocità e costo ; composto da tracce che ospitano blocchi ; esiste una testina che scorre sulle tracce ; per blocchi si intendono quelli logici.



In dettaglio : ogni blocco fisico mappa un blocco logico (identificato da metadati : i-node/mft). Ognuno di questi blocchi è leggibile e scrivibile in un qualunque istante di tempo. Ovviamente ha usura basata sulle parti meccaniche. In dettaglio :

Parametri
- tempo di ricerca della traccia (seek time)
- ritardo di rotazione per l'accesso al settore sulla traccia

Punto di vista del processo



Vediamo un esempio :

Seek time
- n = tracce da attraversare
- m = costante dipendente dal dispositivo
- s = tempo di avvio

$$T_{seek} = m \times n + s$$

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

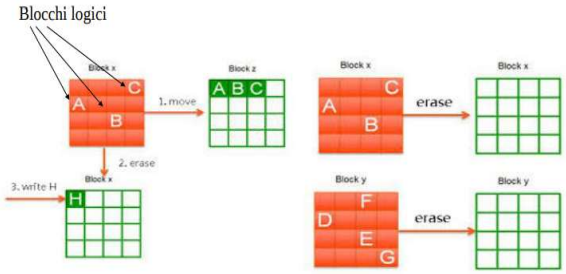
$$T_{rotazione} = \frac{b}{r \times N}$$

Valori classici
- s = 20/3/.. msec.
- m = 0.3/0.1/.. msec.
- r = 300/3600/.../7200 rpm (floppy vs hard disk)

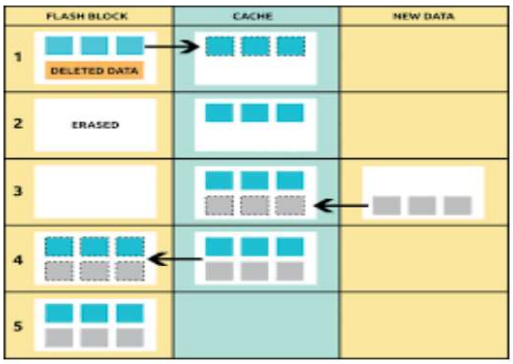
fattore critico

Per quanto riguarda invece gli ssd : dispositivi solo elettronici ; appartengono alla classe NVM (non volatile memory) ; hanno velocità più alta ; **ogni blocco fisico corrisponde ad insieme di blocchi logici** . Ogni blocco è leggibile e scrivibile ma ci deve essere il passaggio per lo stato intermedio di

cancellazione. Ogni passaggio su cancellazione si ha usura del blocco (circa 100000). Si ha quindi una gestione trasparente del "derouting" delle scritture per ammortizzare costi ed effetti negativi della cancellazione : all'interno del dispositivo non usando il software (garbage collector). Vediamo un esempio:



Mentre per la cancellazione:



Vediamo ora in dettaglio tutti i vari scheduling usati:

1. **FCFC (first came first served)**
   a. Adottato al livello del kernel
   b. Le richieste di i/o in base all'ordine di arrivo
   c. Non si produce starvation, ma non minimizza il seek time
   d.

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 – 146

$$\text{lunghezza media di ricerca} \quad \frac{\sum_{i=1}^{|insieme\ dist|} dist_i}{|insieme\ dist|} = 55.3$$

2. **SSTF(shortest service time first)**
   a. Si dà priorità a richiesta di i/o a minor movimento della testina
   b. Non minimizza il tempo di attesa medio e potrebbe produrre starvation
   c.

Un esempio
Traccia iniziale = 100
Insieme delle tracce coinvolte = 55 – 58 – 39 –18 – 90 – 160 – 150 – 38 – 184
Riordino in base al seek time = 90 – 58 – 55 – 39 – 38 – 18 – 150 – 160 – 184

distanze 10 – 32 – 3 – 16 – 1 – 20 – 132 – 10 – 24

$$\text{lunghezza media di ricerca} \quad \frac{\sum_{i=1}^{|insiemedist|} dist_i}{|insiemedist|} = 27.5$$

   d. Una volta eseguita una traccia essa viene cancellata dalla coda
   e. Riduce la metà il seek time
3. **Scan (elevator algothitm)**
   a. Il movimento della testina avviene solo in una data direzione (serviamo tutte le richieste).
   b. Quindi le tracce attraversate recentemente sono sfavorite
   c. Variante : **c-scan** richieste solo in una sola direzione

Un esempio

Un esempio

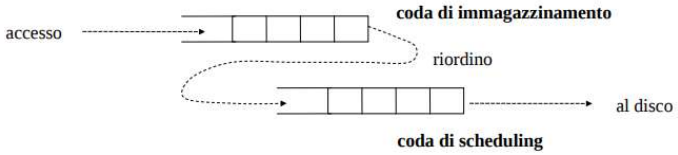
Traccia iniziale = 100
Direzione iniziale = crescente
Insieme delle tracce coinvolte = 55 – 58 – 39 –18 – 90 – 160 – 150 – 38 – 184
Riordino in base alla direzione di seek = 150 – 160 – 184 – 90 – 58 – 55 – 39 – 38 – 18

d. **distanze** 50 – 10 – 24 – 94 – 32 – 3 – 16 – 1 – 20

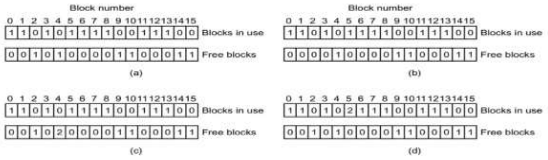
lunghezza media di ricerca $\frac{\sum_{i=1}^{|insiemedist|} dist_i}{|insiemedist|} = 27.8$

4. **Fscan (variante di scan)**
   a. Usa 2 code distinte per la gestione delle richieste
   b. Le code sono FIFO
   c. La coda di scheduling viene gestito con algortirmo ottimizzato (SSTF,SCAN,CSCAN)
   d. Non esiste starvation

e.



Andiamo a vedere ora la consistenza : ogni scrittura è visibile ad ogni lettura , ma attenzione ai dati presenti su dispositivo di massa vs contro quelli su buffer cache (inconsistenza dati) . Un esempio :



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 sostituiscilo in uno dei file (rimedio parziale!!!)

in UNIX fsck mentre scandisk in WINDOWS per vedere se ci sono problemi . Quindi andiamo a vedere come sincronizzare le operazioni :

```
NAME
       fsync,  fdatasync - synchronize a file's complete in-core state with
       that on disk

SYNOPSIS
       #include <unistd.h>

       int fsync(int fd);

       int fdatasync(int fd);

DESCRIPTION
       fsync copies all in-core parts of a file to disk,  and  waits  until
       the  device  reports  that all parts are on stable storage.  It also
       updates metadata stat information. It does  not  necessarily  ensure
       that the entry in the directory containing the file has also reached
       disk.  For that an explicit fsync on  the  file  descriptor  of  the
       directory is also needed.

       fdatasync does the same as fsync but only flushes user data, not the
       meta data like the mtime or atime.
```

Mentre per windows :

The **FlushFileBuffers** function flushes the buffers of a specified file and causes all buffered data to be written to a file.

```
BOOL FlushFileBuffers(
  HANDLE hFile
);
```

Parameters

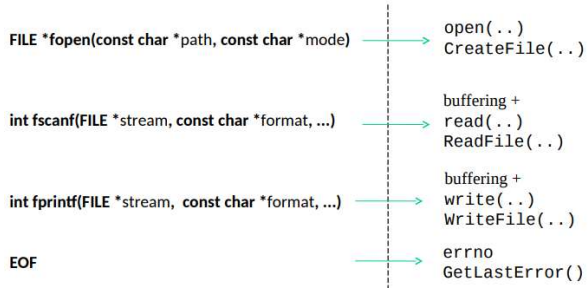
hFile
[in] A handle to an open file.

The file handle must have the GENERIC_WRITE access right. For more information, see File Security and Access Rights.

If *hFile* is a handle to a communications device, the function only flushes the transmit buffer.

If *hFile* is a handle to the server end of a named pipe, the function does not return until the client has read all buffered data from the pipe.

Quindi vediamo la corrispondenza tra le API e le funzioni della libreria :



Vediamo ultimo esempio :

```c
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <unistd.h>
#include <sys/types.h>
#include <sys/stat.h>
#include <fcntl.h>

#define CYCLES 1000

char *string = "francesco";

int main (int argc, char *argv[]) {

        int fd;
        int i;

        if (argc!= 2) {
        printf("Syntax: prog <file_name>\n");
        exit(-1);
        }

        fd=open(argv[1], O_CREAT| O_TRUNC|O_WRONLY,
                        S_IRUSR|S_IWUSR);
        if (fd== -1) {
                perror("Open error: ");
                exit(-2);
        }

        for(i=0; i<CYCLES; i++){
                write(fd,string,strlen(string));
#ifdef FLUSH //try running with this macro or not and then check execution times
                fsync(fd);
#endif
        }
}
```

Il quale mostra quanto ci mette a scrivere. Se includo macro il tempo aumenta (time ./a.out pippo)