Long-term Information Storage

0. Must store large amounts of data
1. Information stored must survive the termination of the process using it
2. Multiple processes must be able to access the information concurrently

L'unità *convenzionale* di conservazione dei dati è il *file*

- insieme di dati
- insieme di attributi

Il concetto di "file" varia da S.O. a S.O.

Alcuni S.O. non hanno il concetto di file (allo scopo di meglio condividere informazioni fra le applicazioni)

Es. prototipi di MacOS (mai realizzati)
Es. PalmOS

Nomi dei file

Extension	Meaning
file.bak	Backup file
file.c	C source program
file.gif	Compuserve Graphical Interchange Format image
file.hlp	Help file
file.html	World Wide Web HyperText Markup Language document
file.jpg	Still picture encoded with the JPEG standard
file.mp3	Music encoded in MPEG layer 3 audio format
file.mpg	Movie encoded with the MPEG standard
file.o	Object file (compiler output, not yet linked)
file.pdf	Portable Document Format file
file.ps	PostScript file
file.tex	Input for the TEX formatting program
file.txt	General text file
file.zip	Compressed archive

Il concetto di file per MS-DOS

dati:
una sequenza di byte

attributi:

- *nome*
- bit read-only?
- bit hidden?
- bit system?
- bit archived?
- timestamps (creazione, ultima modifica, ultimo accesso)
- lunghezza

Nota: non c'è il proprietario (MS-DOS non ha il concetto di "utente")

Il concetto di file per Unix

dati:
una sequenza di byte

attributi:
- UID, GID (utente e gruppo proprietari del file)
- mode (permessi, es. 644 `rw-r--r--`)
- timestamps (creazione, ultima modifica, ultimo accesso)
- lunghezza

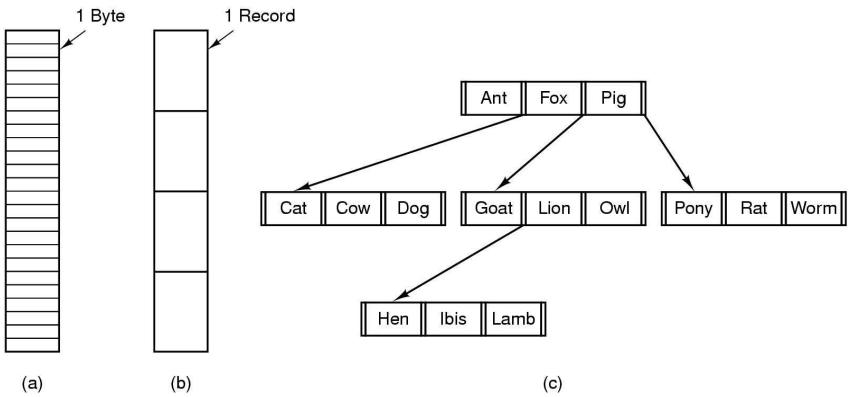
Nota: non c'è il nome; per Unix un file può avere tanti nomi diversi

Il concetto di file per NTFS

dati:
una o più sequenze di byte

attributi:
- proprietario
- ACL (Access Control List)
- timestamps
- lunghezza

Varie maniere di strutturare la parte "dati"



(a) Unix, MS-DOS, NTFS, MacOS
(b) VMS, AS/400, mainframe
(c) AS/400, mainframe

Seq. di record versus seq. di byte

Vantaggi del modello Unix (seq. di byte)
- più semplice
- più flessibile

Vantaggi del modello a record
- tutte le app usano lo stesso meccanismo per conservare i dati
- molte app che potrebbero richiedere un DBMS possono farne a meno

Il modello a record è tuttora popolare nei mainframe

Il modello di Unix:
- la semplicità è la proprietà più importante
- la struttura a record viene vista dall'applicazione, non dal SO
- le funzioni di DB vengono fornite da librerie (es. Berkeley DB)

Il continuo FS—DB

Compiti di un File System
- gestire l'insieme dei blocchi di una partizione
- mantenere informazioni persistenti
- gestire accesso concorrente

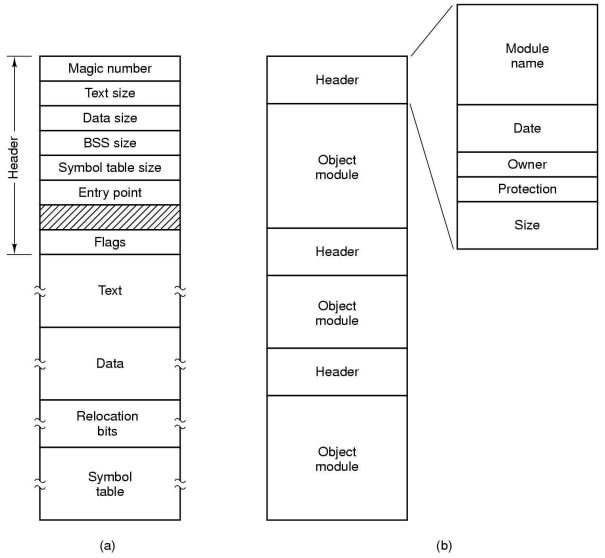
Compiti di un Database Management System
- mantenere informazioni persistenti
- gestire accesso concorrente
- indicizzare le informazioni

$\Rightarrow$ c'è una grande sovrapposizione di compiti

Alcuni DBMS incorporano funzioni di FS
- es. Oracle lavora su partizioni nude

Alcuni FS incorporano funzioni di DBMS
- es. s/390 ha i B-tree nativi nel FS

Tipi di file



Attributi dei file

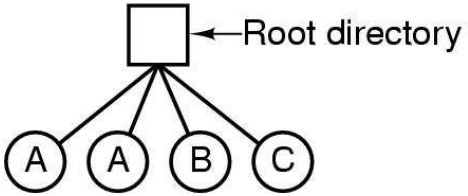
Attribute	Meaning
Protection	Who can access the file and in what way
Password	Password needed to access the file
Creator	ID of the person who created the file
Owner	Current owner
Read-only flag	0 for read/write; 1 for read only
Hidden flag	0 for normal; 1 for do not display in listings
System flag	0 for normal files; 1 for system file
Archive flag	0 for has been backed up; 1 for needs to be backed up
ASCII/binary flag	0 for ASCII file; 1 for binary file
Random access flag	0 for sequential access only; 1 for random access
Temporary flag	0 for normal; 1 for delete file on process exit
Lock flags	0 for unlocked; nonzero for locked
Record length	Number of bytes in a record
Key position	Offset of the key within each record
Key length	Number of bytes in the key field
Creation time	Date and time the file was created
Time of last access	Date and time the file was last accessed
Time of last change	Date and time the file has last changed
Current size	Number of bytes in the file
Maximum size	Number of bytes the file may grow to

Operazioni sui file

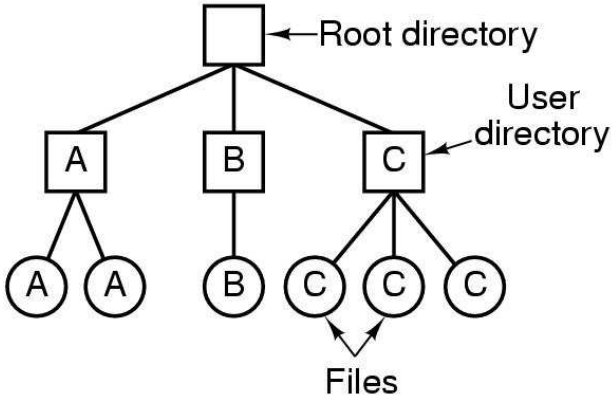
1. Create
2. Delete
3. Open
4. Close
5. Read
6. Write

1. Append
2. Seek
3. Get attributes
4. Set attributes
5. Rename
6. Lock

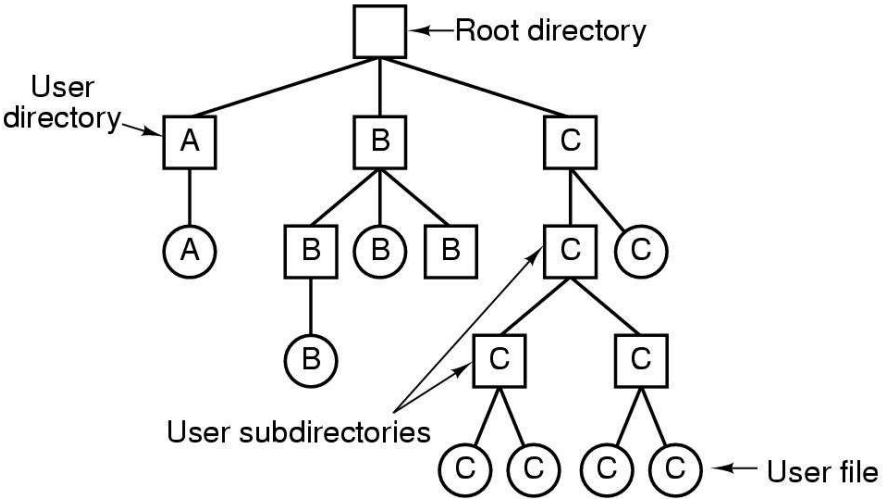
Directories



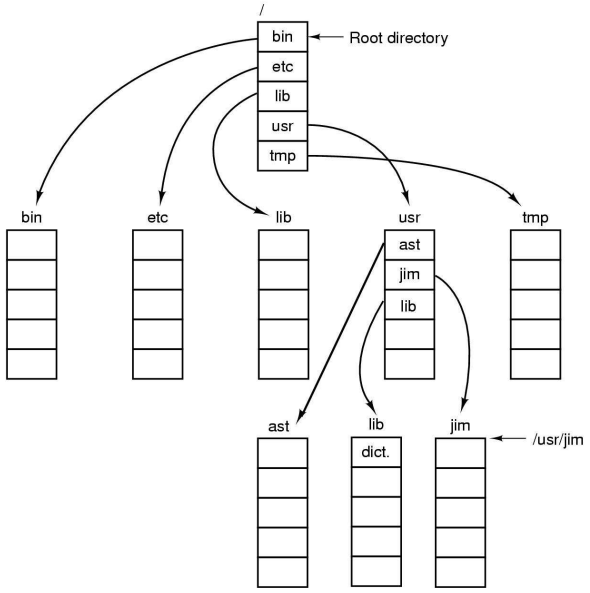
Directories



Directories



Pathnames



Operazioni su directory

1. Create
2. Delete
3. Opendir
4. Closedir
5. Readdir
6. Rename
7. Link
8. Unlink

Manipolazione di file in Unix

I file aperti vengono acceduti tramite file descriptor

file descriptor: intero non negativo (piccolo)

La shell di Unix mantiene la convenzione:
- fd 0 è lo *standard input*
- fd 1 è lo *standard output*
- fd 2 è lo *standard error*

il fd è usato come indice nella *file descriptor table* del processo

File Operations: aprire un file

Prima di usare un file occorre *aprirlo*
- input: file name
- input: flags
- output: file descriptor

int open(const char *pathname, int flags);

open(2) restituisce il più piccolo fd non usato

flags:
- O_RDONLY, O_WRONLY, O_RDWR
- O_APPEND
- O_CREAT
- O_TRUNC
- O_NONBLOCK
- ...

File Operations: creazione

esempio:

```
fd = open("pippo", O_CREAT | O_TRUNC | O_WRONLY, 0644);
```

Note:

- se O_CREAT è presente, si aggiunge un terzo argomento *mode*
- il mode in C è espresso come una costante ottale (inizia con 0!)

File operations: chiudere un file

int close(int *fd*);

Quando un processo termina, tutti i file aperti vengono chiusi dal kernel

Esempio: testiamo standard input per vedere se supporta seek

```
#include <sys/types.h>

int main() {
  if (-1 == lseek(0, 0, SEEK_CUR)) {
    printf("stdin non supporta seek\n");
  } else {
    printf("seek OK\n");
  }
  exit(0);
}
```

File operations: accesso non sequenziale

Ogni file aperto ha un "current file offset"

Indica la posizione per la prossima operazione di lettura o scrittura

Può essere manipolato con la syscall lseek(2)

off_t lseek(int *fd*, off_t *offset*, int *whence*);

I valori possibili per *whence* cambiano l'interpretazione di *offset*
- SEEK_SET: dall'inizio del file
- SEEK_CUR: dalla posizione corrente
- SEEK_END: dalla fine del file

Se ha successo restituisce il nuovo offset

Nota: *non tutti i file* supportano questa operazione

Se fallisce, restituisce -1

More seek fun

L'esecuzione di lseek(2) non causa mai I/O

Posso usare lseek(2) per spostare l'offset al di là della fine del file

Se a questo punto scrivo sul file, ottengo un file con un "buco" (hole)

I buchi non consumano blocchi sul disco

```
# ls -l file.hole
-rw-r-r-    1 matteo   users     1234774 Apr 14 21:55 file.hole
# du file.hole
4       file.hole
```

ls(1) riporta la lunghezza

du(1) riporta i blocchi occupati

File operations: leggere da un file

ssizet_t read(int *fd*, void **buf*, size_t *nbytes*);

Se ha successo, restituisce il numero di byte letti

Può essere meno di *nbytes*:
- in un file regolare, se raggiungo la fine del file
  - alla chiamata successiva, restituisce 0
- leggendo dal terminale, ottengo una riga alla volta
- leggendo dalla rete, ottengo solo i byte immediatamente disponibili

Se fallisce, restituisce -1

File operations: scrivere su un file

ssizet_t write(int *fd*, void **buf*, size_t *nbytes*);

Se ha successo, restituisce il numero di byte scritti

Di solito è uguale a *nbytes* a meno che
- esaurisco lo spazio su disco (errore)
- esaurisco la dimensione massima per file per processo (errore)
- scrivendo sulla rete, il processo ricevente è lento a leggere (non è un errore)

Se fallisce, restituisce -1

Esempio: copiare stdin su stdout

```
#define BUF_SIZE 8192
int main() {
  int n;
  char buf[BUF_SIZE];

  while ((n = read(0, buf, BUF_SIZE)) > 0) {
    if (n != write(1, buf, n)) {
      perror("write error");
      exit(EXIT_FAILURE);
    }
  }
  if (n < 0) {
    perror("read error");
    exit(EXIT_FAILURE);
  }
  exit(0);
}
```

Operazioni atomiche (i)

Per appendere dati in fondo a un file potrei usare

```
lseek(fd, 0, SEEK_END);
write(fd, buf, nbyter);
```

Ma questo codice ha una race condition

Per appendere dati in modo garantito atomico apro il file con O_APPEND

Operazioni atomiche (ii)

Due processi potrebbero sincronizzarsi in questo modo: entrambi cercano di creare un file; solo uno avrà successo. Il file funge da mutex.

```
open("/tmp/mymutex", O_CREAT | O_EXCL, 0600);
```

Testo di riferimento

Per imparare a usare *bene* Unix da *programmatore*:

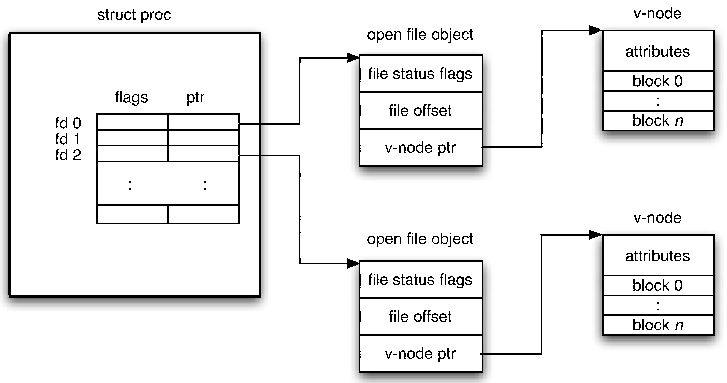
W. Richard Stevens, W. T. Rago
Advanced Programming in the Unix Environment, 2nd edition
Addison-Wesley 2005

Strutture dati nel kernel: i processi e i file

La `struct proc` per ciascun processo
- contiene una file descriptor table
- ogni riga della fd table contiene
  - file descriptor flags
  - puntatore a un file object

Il file object contiene
- file status flags (`O_RDONLY`, `O_WRONLY`, `O_NOBLOCK`,...)
- current file offset
- puntatore a un v-node

Il v-node è associato univocamente a un file (regolare o meno)
- attributi del file
- device
- lista dei blocchi



La struttura a tre livelli

Esiste fin dalle primissime versioni di Unix

Serve a consentire la condivisione dei file

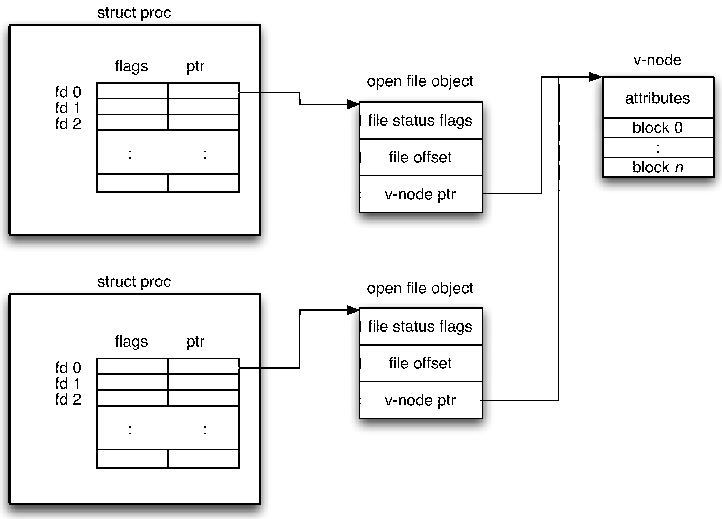
0. fd table (una per processo)
1. file object (uno per ogni esecuzione di open(2))
2. v-node (uno per file)

Operazioni compiute dalla open(2):

- percorre il pathname per trovare il file
- crea il v-node (se non è già aperto)
- crea il file object
- aggiorna la fd table

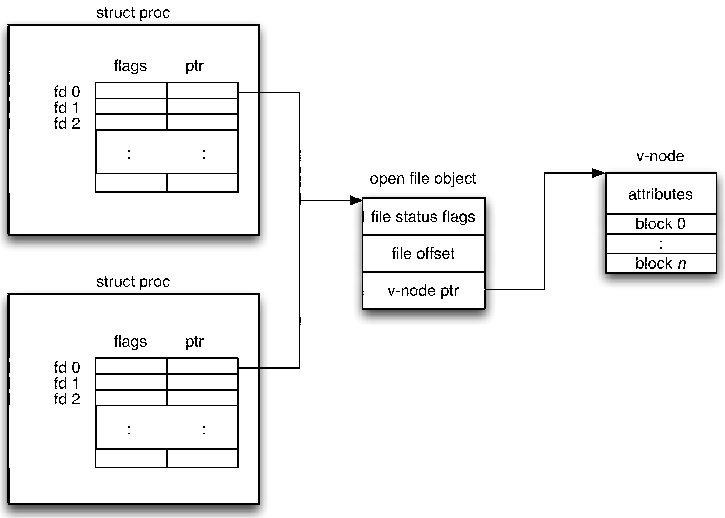
Condivisione, primo caso

- due processi indipendenti aprono lo stesso file
- devono avere current file offset separati



Condivisione, secondo caso

Dopo una fork(2), i due processi hanno fd table identiche



Esempio: la redirezione della shell (i)

$ *command* > *file*

Ridirigi fd 1 sul *file*

```
if (fork() == 0) {
  // siamo nel figlio
  close(1);

  // ora la open(2) restituirà 1
  open("file", O_CREAT | O_TRUNC | O_WRONLY, 0644);

  execve("command", ...);
} else {
...
```

Ora si spiega...

... perché in Unix usiamo fork + exec

Fra la fork e la exec, la shell può manipolare la fd table

... il file descriptor 1 non viene mai chiuso durante l'esecuzione di questo script ... quindi l'offset cresce sempre!

Esempio: gli script di shell

Un altro caso in cui due processi condividono un file è dopo una fork(2)
Es. uno script di shell "prova.sh"

```
/bin/echo ciao
/bin/echo a tutti
```

eseguo

```
$ sh prova.sh > foo
```

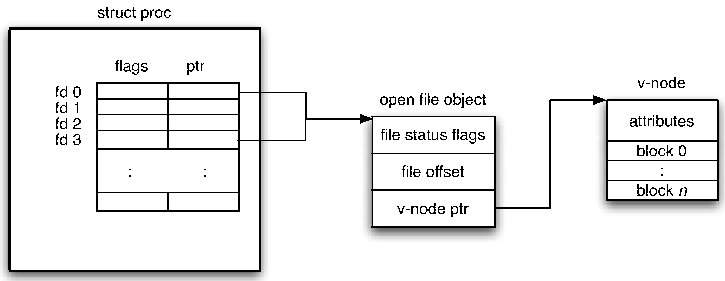
Cosa contiene "foo" alla fine?
a. ciao\na tutti
b. a tutti
c. (niente)

Condivisione di file, terzo caso

int dup(int *fd*);
"Duplica" il file descriptor *fd*

Se ha successo restituisce un nuovo file descriptor che è il più piccolo fra quelli non aperti

Il nuovo fd punta a una copia della riga del *fd* originale



Esempio: la redirezione della shell (ii)

$ *command* > file 2>&1

Ridirigi fd 2 sullo stesso file di fd 1

```
if (fork() == 0) {
  // siamo nel figlio
  close(1);

  // ora la open(2) restituirà 1
  open("file", O_CREAT | O_TRUNC | O_WRONLY, 0644);

  close(2);

  // la dup restituirà 2
  dup(1);

  execve("command", ...);
}
```

Esercizio...

Perché questi due frammenti di shell si comportano in modo diverso?

```
$ a.out > outfile 2>&1
```

```
$ a.out 2>&1 > outfile
```

Suggerimento: la shell interpreta i suoi argomenti da sinistra a destra

Da dove vengono standard input, output ed error?

Minix: servers/init/init.c

```
int main(void)
{
  struct stat stb;

  if (fstat(0, &stb) < 0) {
        /* Open standard input, output & error. */
        (void) open("/dev/null", O_RDONLY);
        (void) open("/dev/log", O_WRONLY);
        dup(1);
  }
          :
```

Ancora

Linux 0.01, init/main.c

```
void init(void)
{
        int i,j;

        setup();
        if (!fork())
                _exit(execve("/bin/update",NULL,NULL));
        (void) open("/dev/tty0",O_RDWR,0);
        (void) dup(0);
        (void) dup(0);
              :
```

Strutture dati del kernel associate a un processo

- file descriptor table
- (ormai la conosciamo!)

- current working directory
  - serve a interpretare i pathname parziali

- current root
  - serve a interpretare i pathname assoluti
  - si può cambiare con chroot(2)

La parte di process table che compete al FS

Minix servers/fs/fproc.h

```
EXTERN struct fproc {
  mode_t fp_umask;              /* mask set by umask system call */
  struct inode *fp_workdir;     /* pointer to working directory's inode */
  struct inode *fp_rootdir;     /* pointer to current root dir (see chroot) */
  struct filp *fp_filp[OPEN_MAX];/* the file descriptor table */
  uid_t fp_realuid;             /* real user id */
  uid_t fp_effuid;              /* effective user id */
  gid_t fp_realgid;             /* real group id */
  gid_t fp_effgid;              /* effective group id */
  dev_t fp_tty;                 /* major/minor of controlling tty */
    :      :                          :
  pid_t fp_pid;                 /* process id */
  long fp_cloexec;              /* bit map for POSIX Table 6-2 FD_CLOEXEC */
} fproc[NR_PROCS];
```

Tabella degli *open file objects*

```
/* This is the filp table.  It is an intermediary between file descriptors
 *  and inodes.  A slot is free if filp_count == 0.
 */

EXTERN struct filp {
  mode_t filp_mode;           /* RW bits, telling how file is opened */
  int filp_flags;             /* flags from open and fcntl */
  int filp_count;             /* how many file descriptors share this slot?*/
  struct inode *filp_ino;     /* pointer to the inode */
  off_t filp_pos;             /* file position */

  /* the following fields are for select() and are owned by the generic
   * select() code (i.e., fd-type-specific select() code can't touch these).
   */
  int filp_selectors;         /* select()ing processes blocking on this fd */
  int filp_select_ops;        /* interested in these SEL_* operations */

  /* following are for fd-type-specific select() */
  int filp_pipe_select_ops;
} filp[NR_FILPS];
```