Programmazione di Sistema – 2

Lucidi per il corso di Laboratorio di Sistemi Operativi tenuto da Paolo Baldan presso l'Università Ca' Foscari di Venezia, anno accademico 2004/2005. Parte di questo materiale è rielaborato dai lucidi del Corso di Laboratorio di Sistemi Operativi tenuto da Rosario Pugliese presso l'Università di Firenze, anno accademico 2001/02.

UNIX: Programmazione di Sistema

Per utilizzare i servizi offerti da UNIX, quali creazione di file, duplicazione di processi e comunicazione tra processi, i programmi applicativi devono interagire con il sistema operativo.

Per far ciò devono usare un insieme di routine dette system call, che costituiscono l'interfaccia funzionale del programmatore col nucleo di UNIX.

Le system call sono simili alle routine di libreria C ma eseguono una chiamata di subroutine direttamente nel nucleo di UNIX.

UNIX: System Call

Le system call sono delle "entry point" per il kernel.

Il programmatore chiama la funzione utilizzando la sintassi usuale delle funzioni C

```
int open(const char *path, int mode)
```

La funzione invoca, nel modo opportuno, il servizio del sistema operativo

- *"salva" gli argomenti della system call ed un numero identificativo della system call stessa (in registri)*
- *esegue una istruzione macchina trap, int, ...*

(sezione 2 di **man***:* **`man 2 syscall`***).*

UNIX: Funzioni di libreria

Le funzioni di libreria forniscono servizi di utilità generale al programmatore

Non sono entry point del kernel, anche se possono far uso di system call per realizzare il proprio servizio. Es.

- *printf può utilizzare la system call write per stampare*
- *strcpy (string copy) e atoi (convert ASCII to integer) non coinvolgono il sistema operativo.*

Possono essere sostituite con altre funzioni che realizzano lo stesso compito (in generale non possibile per le system call).

(sezione 3 di **man***:* **`man 3 libfun`***).*

UNIX: Programmazione di Sistema

Le chiamate di sistema possono essere raggruppate in tre categorie principali

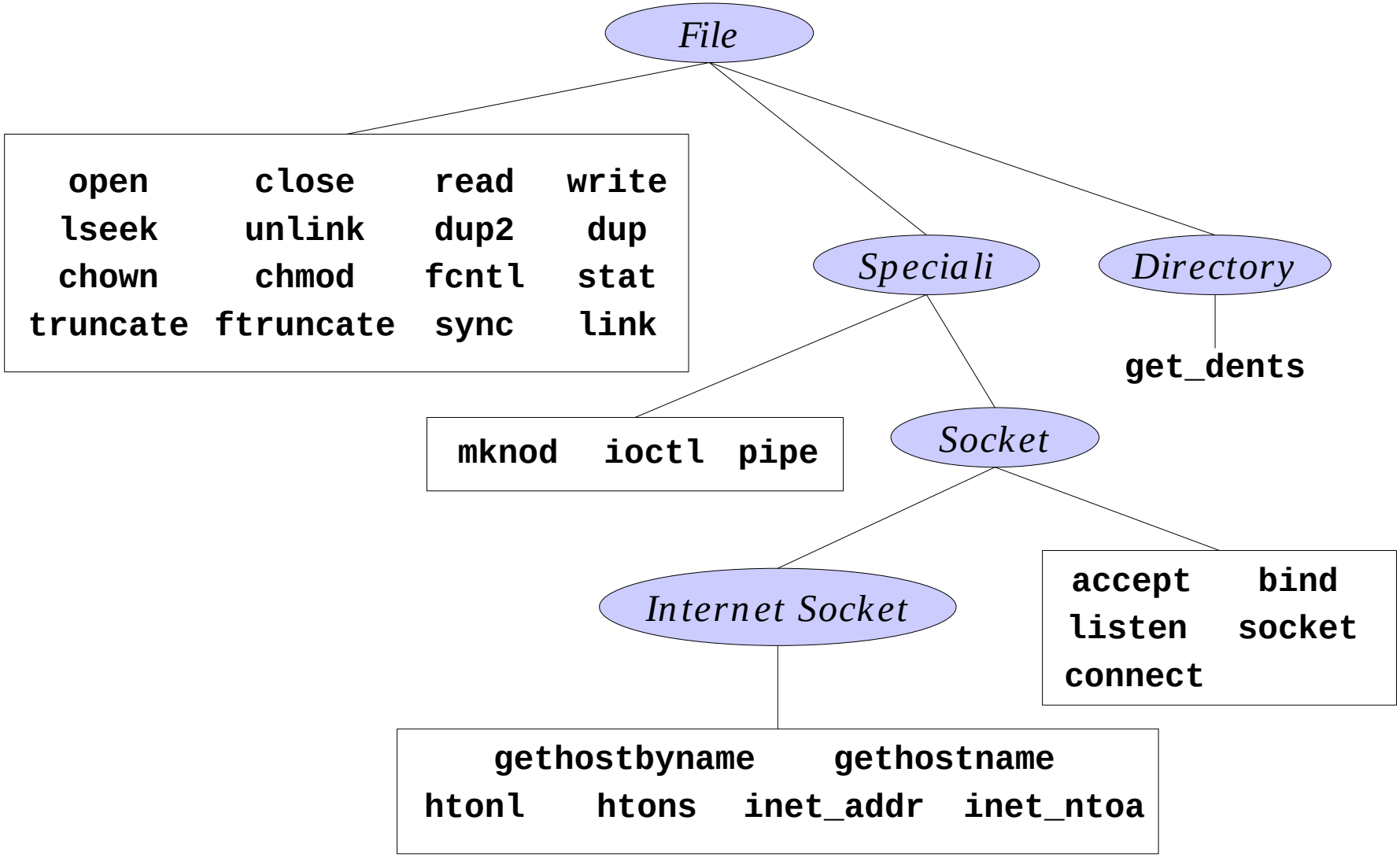
- *gestione dei file,*
- *gestione degli errori,*
- *gestione dei processi.*

La comunicazione tra processi (IPC - interprocess communication) rientra nella gestione dei file poiché UNIX tratta i meccanismi per IPC come file speciali.

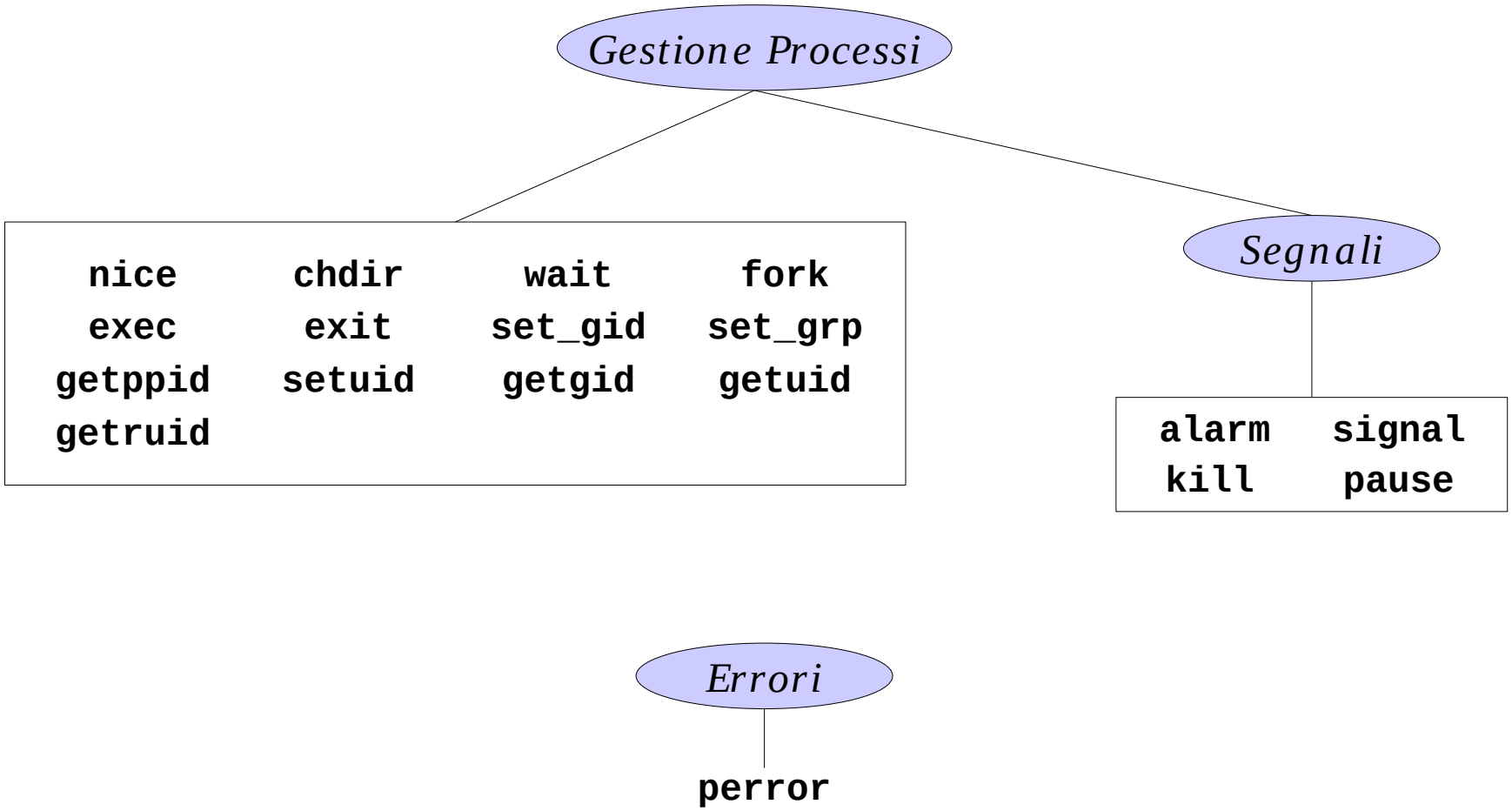
Per utilizzare le system call sono forniti opportuni file di intestazione. Ad es.

- **sys/file.h**
- **errno.h**

Gerarchia di System Call: File



Gerarchia di System Call: Processi



Gestione dei File

Gestione dei File

Le system call per la gestione dei file permettono di manipolare

- *file regolari,*
- *directory,*
- *file speciali.*

Tra i file speciali

- *link simbolici,*
- *dispositivi (terminali, stampanti),*
- *meccanismi di IPC (pipe e socket).*

Gestione dei File: Operazioni di Base

Le system call descritte nella prima parte realizzano le operazioni di base per la gestione dei file

`open()` - *apertura*

`read()` - *lettura*

`write()` - *scrittura*

`lseek()` - *posizionamento*

`close()` - *chiusura*

Unbuffered I/O (vs. buffered I/O library: tipo FILE e funzioni `fopen`, `fread`, `fwrite`, *...)*

Non sono veramente unbuffered ...

File: Tipica Sequenza di Operazioni

```
int fd; /* dichiara un descrittore di file */
...
fd = open(fileName, ...); /* apre un file ; fd è il descrittore */
if (fd == -1) {
  ... /* gestisce l'errore */     };
...
fcntl(fd, ...); /* modifica alcuni flag di I/O */
...
read(fd, ...); /* legge dal file */
...
write(fd, ...); /* scrive nel file */
...
lseek(fd, ...); /* si sposta all'interno del file */
...
close(fd); /* chiude il file, liberando il descrittore */
...
unlink(fileName); /* rimuove il file */
...
```

Gestione dei File

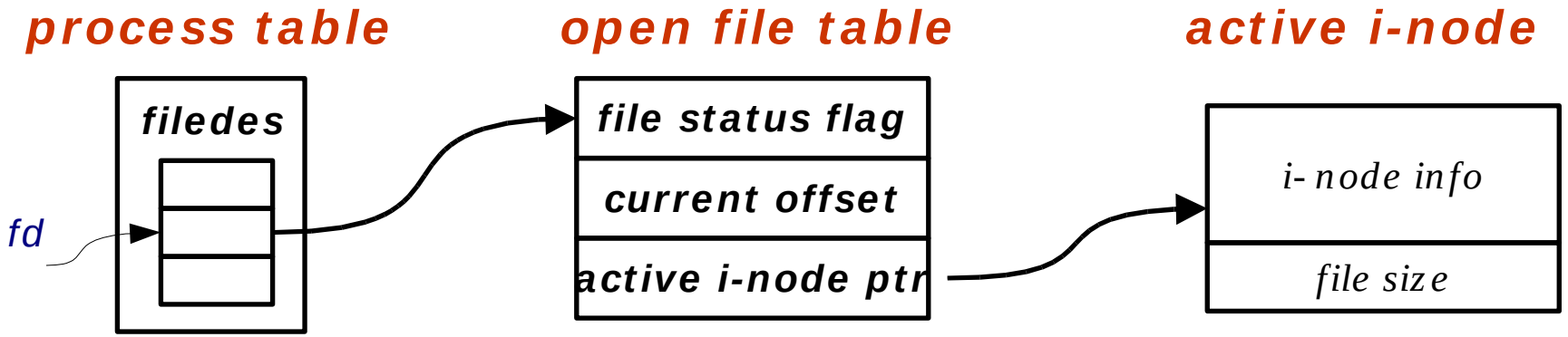
Per accedere a un file questo deve essere aperto tramite la system call **open( )**

- *localizza il file nel filesystem tramite il suo pathname;*
- *copia nella tabella degli i-node attivi l'i-node corrispondente al file;*
- *alloca una entry nella tabella dei file aperti globale;*
- *alloca una entry nella tabella dei descrittori di file aperti del processo;*

open( ) *ritorna*

- *un intero positivo fd, detto descrittore di file, utilizzato nelle successive operazioni di I/O sul file, se ha successo;*
- *-1 se fallisce.*

Gestione dei File: Strutture dati



Il process control block di ciascun processo contiene una tabella di puntatori a file descriptor presenti nella tabella dei file aperti.

La tabella dei file aperti contiene un file descriptor per ogni file aperto nel sistema, con un puntatore alla tabella degli i-node attivi.

La tabella degli i-node attivi contiene l'i-node per ogni file che sia aperto (una o più volte) nel sistema.

Descrittore di File

Molte system call di I/O richiedono come primo argomento un (puntatore a un) descrittore di file.

Ogni descrittore riferisce un insieme di proprietà (che sono indipendenti da quelle del file a cui punta)

- *Puntatore di lettura/scrittura*
  - *offset, rispetto all'inizio del file della posizione a cui sta leggendo o scrivendo;*
  - *inizializzato a 0 alla creazione del descrittore;*
  - *aggiornato ad ogni operazione di lettura/scrittura nel file.*
- *Flag close on exec*
  - *indica se il descrittore deve essere chiuso automaticam. all'invocazione di una system call della famiglia* **exec()**.
- *Flag append*
  - *indica se l'output sul file deve essere inserito in fondo.*

Descrittore di File

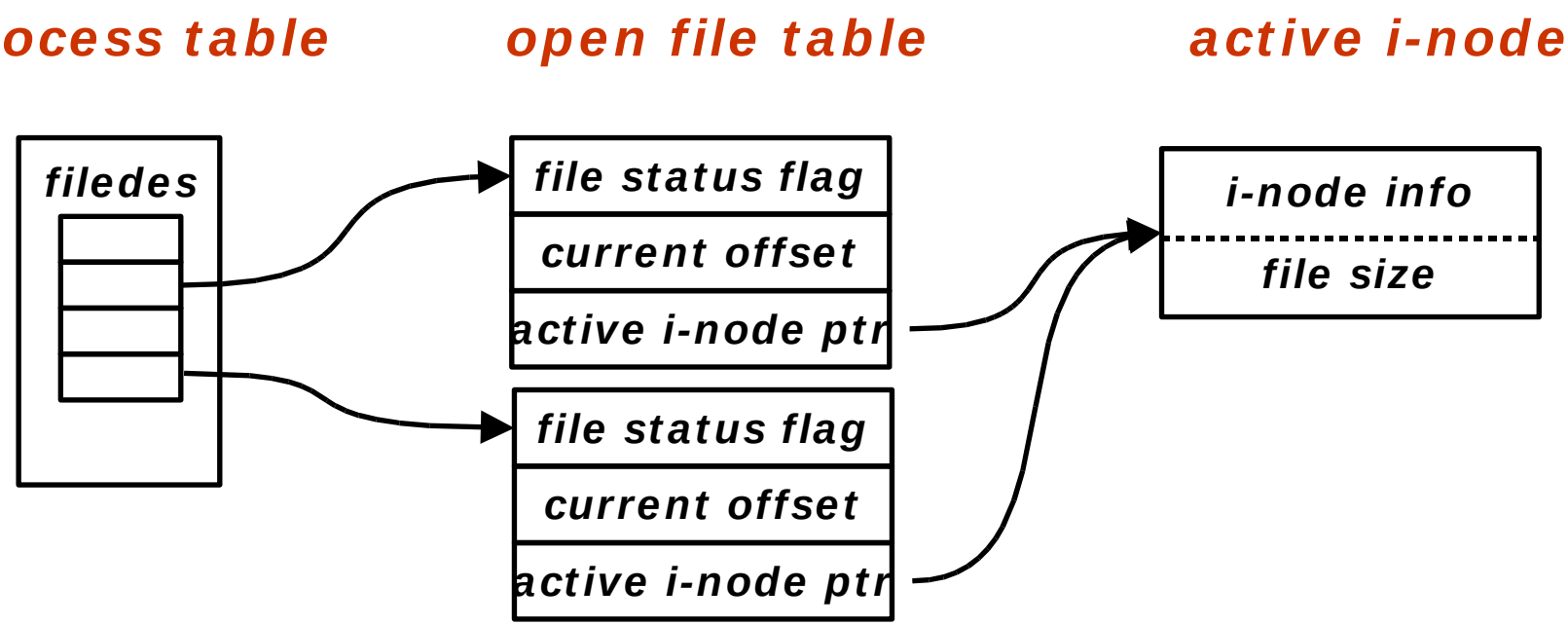
Se il file è speciale (es. pipe o socket), ci sono altre proprietà associate al descrittore

- *Flag blocking*
  - *indica se un processo si deve bloccare quando tenta di leggere dal file e questo è vuoto.*
- *SIGIO pid/gid*
  - *identificatore di processo o di gruppo a cui va spedito un segnale SIGIO quando l'input sul file diventa disponibile.*

Le proprietà associate ad un descrittore possono esser manipolate tramite le system call **`open()`** *e* **`fcntl()`**.

Descrittore di File

Un file può essere aperto più volte con descrittori diversi (anche dallo stesso processo)

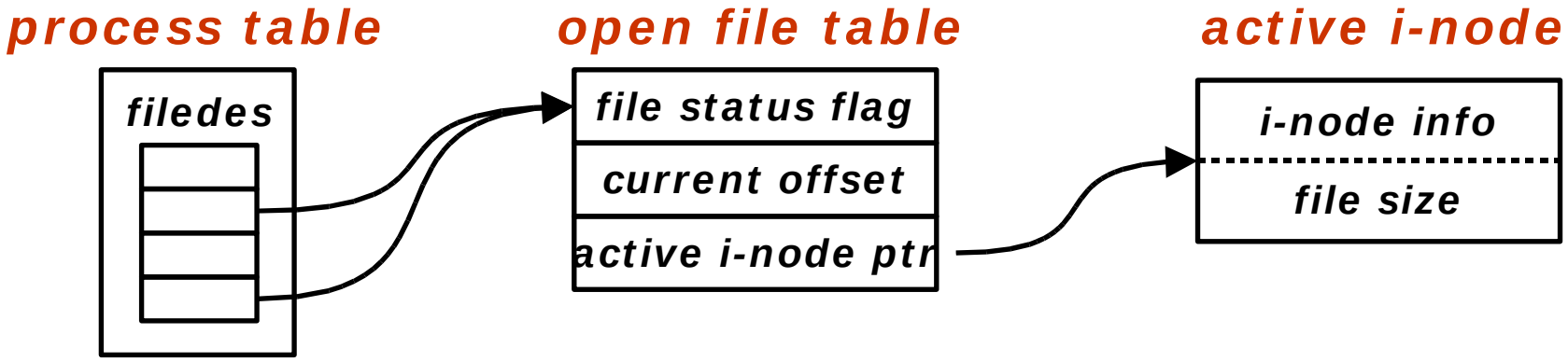



```
fd1=open("myfile", ...)
fd2=open("myfile", ...)
```

Descrittore di File

Uno stesso descrittore può essere riferito da più entry delle tabelle dei processi

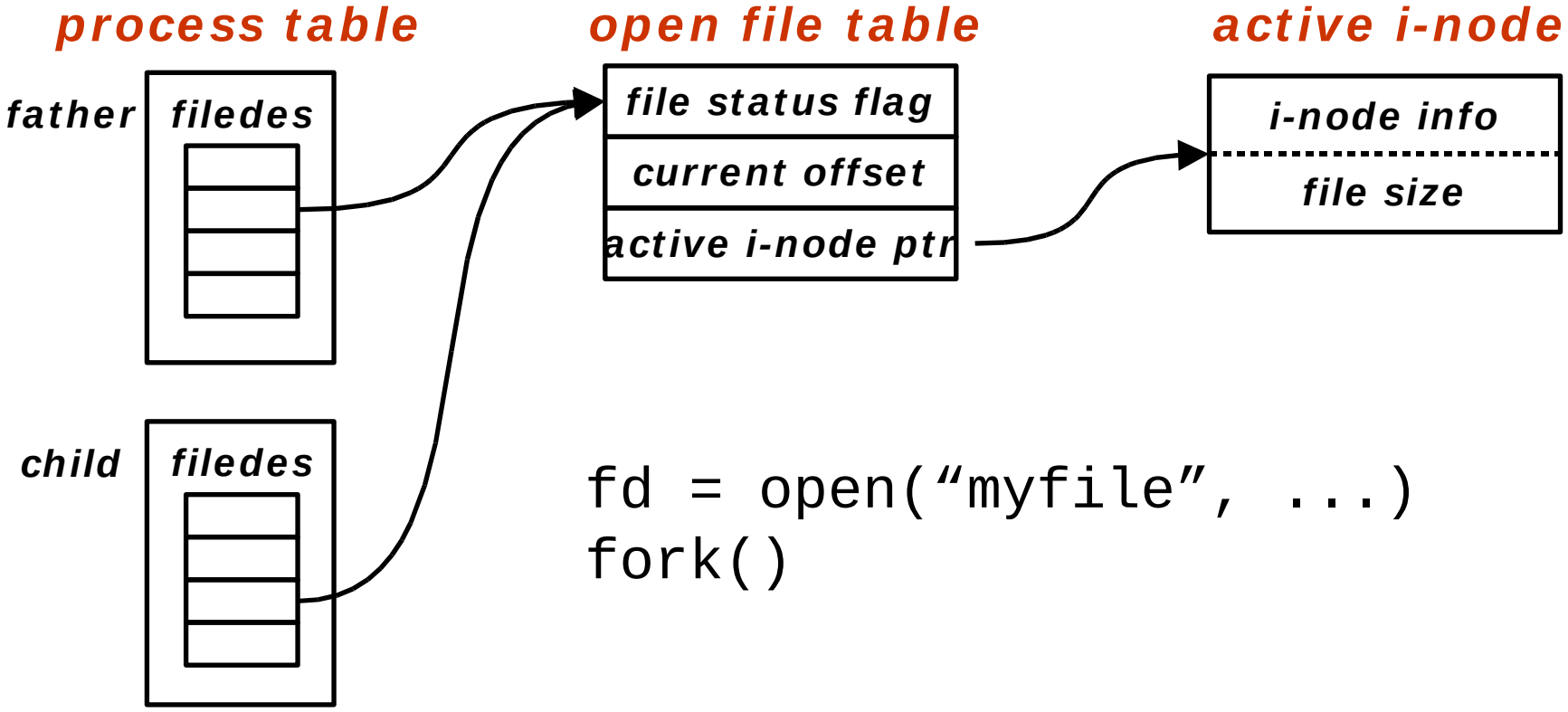
- *all'interno dello stesso processo (tramite* **dup( )***)*
- *...*



Descrittore di File

Uno stesso descrittore può essere riferito da più entry delle tabelle dei processi

in processi diversi (tramite **`fork()`***)*



Descrittore di File

I descrittori sono numerati a partire da 0.

I primi tre hanno un significato particolare

- 0: corrisponde allo standard input;
- 1: corrisponde allo standard output;
- 2: corrisponde allo standard error.

Gestione dei File: Chiusura

Quando un processo non ha più bisogno di un file aperto, lo chiude invocando **`close()`**.

Tutti i file aperti da un processo sono chiusi automatic. quando il processo termina (ma è buona pratica di programmazione chiudere esplicitamente i file aperti).

Quando il riferimento ad un file viene chiuso

- *la corrispondente entry nella process table viene liberata e può essere riassegnata da una successiva* **open()**;
- *se non ci sono altri puntatori alla entry nella open file table, anche questa viene liberata;*
- *se il file non è aperto in nessun processo anche l'i-node nella tabella degli active i-node viene deallocato.*

Esempio: reverse.c

`reverse -c [fileName]`

Inverte le linee di input lette da **`fileName`** *e le mostra sullo standard output.*

Se non si specifica alcun file di input, reverse inverte lo standard input.

Con l'opzione **`-c`**, *reverse inverte anche i caratteri su ogni linea.*

[reverseLine.c]

reverse.c

```
#include <fcntl.h> /* For file mode definitions */
#include <stdio.h>
#include <stdlib.h>

/* Enumerator */
enum { FALSE, TRUE }; /* Standard false and true values */
enum { STDIN, STDOUT, STDERR }; /* Standard I/O channel */

/* #define Statements */
#define BUFFER_SIZE    4096   /* Copy buffer size */
#define NAME_SIZE      12
#define MAX_LINES      100000 /* Max lines in file */

/* Globals */
char *fileName = NULL; /* Points to file name */
char tmpName [NAME_SIZE];
int charOption = FALSE; /* Set to true if -c option is used */
int standardInput = FALSE; /* Set to true if reading stdin */
int lineCount = 0; /* Total number of lines in input */
int lineStart [MAX_LINES]; /* Store offsets of each line */
int fileOffset = 0; /* Current position in input */
int fd; /* File descriptor of input */
```

reverse.c

```
int main(int argc, char *argv[]) {
  parseCommandLine (argc,argv); /* Parse command line */
  pass1 (); /* Perform first pass through input */
  pass2 (); /* Perform second pass through input */
  return (/* EXIT SUCCESS */ 0); /* Done */
}

/*********************************************************/
int parseCommandLine (int argc, char *argv[]) {
  int i;
  for (i = 1; i < argc; i++) {
     if(argv[i][0] == '-')
       processOptions (argv[i]);
     else if (fileName == NULL)
       fileName = argv[i];
     else
       usageError (); /* An error occurred */
  }
  standardInput = (fileName == NULL);
}
```

reverse.c

```
int processOptions (char *str) {
  int j;
  for (j = 1; str[j] != '\0'; j++) {
     switch(str[j]) { /* Switch on command line flag */
        case'c':
          charOption = TRUE;
          break;
        default:
          usageError ();
          break;
     }
  }
}
/*****************************************************************/
int usageError (void) {
  fprintf (stderr, "Usage: reverse -c [filename]\n");
  exit (/* EXITFAILURE */ 1);
}
```

reverse.c

```
int pass1 (void) {   /* Perform first scan through file */

   int tmpfd, charsRead, charsWritten;
   char buffer [BUFFER_SIZE];

   if (standardInput) { /* Read from standard input */
     fd = STDIN;
     sprintf (tmpName, ".rev.%d", getpid ()); /* Random name */
     /* Create temporary file to store copy of input */
     tmpfd = open (tmpName, O_CREAT | O_RDWR, 0600);
     if (tmpfd == -1) fatalError ();
   }
   else { /* Open named file for reading */
      fd = open (fileName, O_RDONLY);
      if (fd == -1) fatalError ();
   }
```

reverse.c

```
    lineStart[0] = 0; /* Offset of first line */
    while (TRUE) { /* Read all input */
       /* Fill buffer */
       charsRead = read (fd, buffer, BUFFER_SIZE);
       if (charsRead == 0) break; /* EOF */
       if (charsRead == -1) fatalError (); /* Error */
       trackLines (buffer, charsRead); /* Process line */
       /* Copy line to temporary file if reading from stdin */
       if (standardInput) {
          charsWritten = write (tmpfd, buffer, charsRead);
          if(charsWritten != charsRead) fatalError ();
       }
    }

    /* Store offset of trailing line */
    lineStart[lineCount] = fileOffset;
    /* If reading from standard input, prepare fd for pass2 */
    if (standardInput) fd = tmpfd;
}
```

reverse.c

```
int trackLines (char *buffer, int charsRead) {
  /* Store offsets of each line start in buffer */
  int i;
  for (i = 0; i < charsRead; i++) {
     ++fileOffset; /* Update current file position */
     if (buffer[i] == '\n') lineStart[++lineCount] = fileOffset;
  }
}

/*****************************************************/
int pass2 (void) {
  /* Scan input file again, displaying lines in reverse order */
  int i;
  for (i = lineCount - 1; i >= 0; i--)
    processLine (i);
  close (fd); /* Close input file */
  if (standardInput) unlink (tmpName); /* Remove temp file */
}
```

reverse.c

```
int processLine (int i) {
  /* Read a line and display it */
  int charsRead;
  char buffer [BUFFER_SIZE];

  /* Find the line and read it */
  lseek (fd, lineStart[i], SEEK_SET);
  charsRead = read (fd, buffer, lineStart[i+1] - lineStart[i]);

  /* Reverse line if -c option was selected */
  if (charOption) reverseLine (buffer, charsRead);
  /* Write it to standard output */
  write (1, buffer, charsRead);
}
```

reverse.c

```
int reverseLine (char *buffer, int size) {
  /* Reverse all the characters in the buffer */
  int start = 0, end = size - 1;
  char tmp;
  if (buffer[end] == '\n') --end; /* Leave trailing newline */
  /* Swap characters in a pairwise fashion */
  while (start < end) {
     tmp = buffer[start];
     buffer[start] = buffer[end];
     buffer[end] = tmp;
     ++start; /* Increment start index */
     --end; /* Decrement end index */
  }
}
/*****************************************************************/
int fatalError (void) {
  perror ("reverse: "); /* Describe error */
  exit (1);
}
```

open()

`int open(char *path, int mode [,mode_t perms])`

apre (o crea) un file in lettura e/o scrittura

`path:` *pathname assoluto o relativo del file.*

`mode:` *indica la modalità di apertura*

si ottiene combinando, tramite l'operatore `|', flag di lettura/scrittura ed eventualmente altri flag.

`perms:`*permessi di accesso al file (in forma ottale o con costanti)*

è specificato solo quando il file è creato;

sui permessi influisce anche il valore di umask.

open() *restituisce un intero non negativo fd, se ha successo; altrimenti, restituisce il valore -1.*

open(): Flag di lettura/scrittura

I valori predefiniti dei flag sono nel file **`fcntl.h`**.

Flag di lettura/scrittura sono

- **`O_RDONLY`**: *apertura in sola lettura (0);*
- **`O_WRONLY`**: *apertura in sola scrittura (1);*
- **`O_RDWR`** : *apertura in lettura e scrittura (2).*

Solo una di queste costanti può comparire in mode.

Altre costanti (aggiunte in or "|" a una di queste tre) definiscono altri comportamenti.

open(): Altri flag

O_APPEND: *Il puntatore al file è posizionato alla fine del file prima di ogni* **write()**.

O_CREAT: *Se il file non esiste viene creato*

l'UID "effettivo" del processo è usato come ID del possessore;

il parametro **perm** *(se presente) determina, insieme con umask, i permessi iniziali sul file.*

O_EXCL: *Con* **O_CREAT**, *se il file esiste* **open()** *fallisce.*

O_NONBLOCK: *Per pipe e socket*

se il flag è attivo

open() in sola lettura termina immediatamente, a prescindere dal fatto che il lato di scrittura sia aperto o no;

open() in sola scrittura fallisce se il lato di lettura non è aperto;

se il flag non è attivo

una open() su un file in sola lettura o in sola scrittura si blocca fino a che l'altro lato non è aperto.

O_TRUNC: *Se il file esiste, è troncato a lunghezza 0.*

open(): esempi

```
sprintf(tmpName, ".rev.%d", getpid()); /* Random name */
    /* Create temporary file to store a copy of input */
tmpfd = open (tmpName, O_CREAT | O_RDWR, 0600);
if (tmpfd == -1) fatalError ();
```

La system call **`getpid()`** *restituisce il PID del processo chiamante. Questo è unico in tutto il sistema, per cui si usa spesso per generare nomi unici per file temporanei.*

Per creare il file si usa il flag **`O_CREAT`** *e si impostano i permessi iniziali tramite cifre ottali.*

```
fd = open (fileName, O_RDONLY);
if (fd == -1) fatalError ();
```

Per aprire il file **`fileName`** *esistente si specifica solo il flag di lettura/scrittura.*

read()

`ssize_t read(int fd, void *buf, size_t count)`

Copia nel buffer **`buf`** *una sequenza di* **`count`** *byte letti a partire dalla posizione corrente del file riferito dal descrittore* **`fd`***.*

Aggiorna il puntatore di lettura/scrittura.

Legge quanti più byte possibile, fino ad un massimo di count, e ritorna il numero dei byte effettivamente letti (senza contare l'eventuale carattere **EOF***).*

In particolare, se invocata dopo che si è già letto l'ultimo byte, restituisce il valore 0, che indica la fine del file.

Se fallisce restituisce il valore -1.

read()

Nota: **read()** *esegue input di basso livello*

non ha le capacità di formattazione di **scanf()**, *ma...*

salta lo strato intermedio delle funzioni di libreria C e quindi risulta molto più veloce.

Es.

```
while (TRUE) {
 charsRead = read (fd, buffer, BUFFER_SIZE);
 if (charsRead == 0) break; /* EOF */
 if (charsRead == -1) fatalError (); /* Error */
}
```

read è usata per legger **BUFFER_SIZE** *caratteri per volta.*

write()

ssize_t write(int fd, void *buf, size_t count)

Scrive nel file riferito dal descrittore **fd** *una sequenza di* **count** *byte letti dal buffer* **buf**.

Se il flag **O_APPEND** *è stato attivato, prima di ogni scrittura il puntatore è spostato alla fine del file. Altrimenti scrive a partire dalla posizione corrente.*

Aggiorna il puntatore di lettura/scrittura.

Scrive quanti più byte possibile, fino ad un massimo di **count**, *e ritorna il numero dei byte effettivamente scritti.*

Se **write()** *ritorna un valore inferiore a* **count** *probabilm. il disco è pieno.*

Se fallisce restituisce il valore -1.

write(): esempi

Nota: Valgono per **`write()`**, *rispetto a* **`printf()`**, *le considerazioni fatte per* **`read()`**, *rispetto a* **`scanf()`**.

Es.

```
/* Copy line to temporary file if reading from stdin */
if (standardInput) {
   charsWritten = write (tmpfd, buffer, charsRead);
   if (charsWritten != charsRead) fatalError ();
}
```

lseek()

off_t lseek(int fd, off_t offset, int mode)

Dato il descrittore di file **fd**, *sposta il puntatore alla posizione corrente di* **offset** *byte.*

- **mode** *specifica come interpretare* **offset**. *I tre possibili valori sono definiti in* **stdio.h**
  - **SEEK_SET**: *offset relativo all'inizio del file;*
  - **SEEK_CUR**: *offset relativo alla posizione corrente;*
  - **SEEK_END**: *offset relativo alla fine del file;*
- *fallisce se si tenta di muovere il puntatore ad un punto che precede l'inizio del file;*
- *estende il file se si sposta il puntatore oltre la fine (senza allocare spazio, nel file si crea un "buco").*

Ritorno la posizione corrente, se ha successo; ritorna il valore -1, altrimenti.

lseek(): esempi

```
lseek (fd, lineStart[i], SEEK_SET); /* Find the line and
                                       read it */
charsRead = read (fd, buffer, lineStart[i+1]-lineStart[i]);
```

lineStart[i] *contiene la posizione, relativa all'inizio del file, del primo carattere della riga i-ma. Quindi* **lseek()** *sposta il puntatore all'inizio di tale riga.*

La lunghezza della riga è calcolata come **lineStart[i+1]-lineStart[i]** *e viene letta l'intera riga.*

```
currentOffset = lseek (fd, 0, SEEK_CUR);
```

memorizza nella variabile **currentOffset** *la posizione corrente, senza spostare il puntatore.*

lseek(): Oltre la fine

Se si esegue una **lseek()** *spostando il puntatore oltre la fine del file ...*

Se si esegue una **write()**, *il nucleo del SO estende automaticamente la dimensione del file.*

Nota: viene modificato il campo file size nella tabella degli i-node attivi, e quindi la modifica è visibile anche a chi operi sul file tramite un altro descrittore.

L'area intermedia del file è trattata come se fosse riempita da caratteri **NULL** *(codifica ASCII 0).*

Tuttavia, non viene allocato spazio disco per l'area intermedia e quindi il file occupa meno spazio disco di uno analogo in cui i caratteri **NULL** *siano stati inseriti esplicitamente dall'utente.*

close()

`int close(int fd)`

Libera il descrittore di file **`fd`**.

Se **`fd`** *è l'ultimo descrittore associato al file, le risorse del nucleo per quel file vengono deallocate.*

Quando un processo termina, tutti i suoi descrittori di file sono chiusi automaticamente, ma è preferibile chiuderli esplicitamente quando non servono più.

Se si chiude un descrittore che era già chiuso, si verifica un errore.

Restituisce il valore 0, se ha successo; restituisce il valore -1, altrimenti.

Nota: Quando un file viene chiuso, non è garantito che i buffer associati siano immediatamente scaricati su disco (cfr. **`sync()`**).

Esempio: mycopy.c

```
/* mycopy src trg: crea una copia del file src e la chiama trg */

#include <stdio.h>
#include <sys/file.h>
#define  BUFFSIZE   8192

int main(int argc, char *argv[]) {
  int fdSource; /* file descriptor per il file origine */
  int fdTarget; /* file descriptor per il file copia*/
  int n;
  char buf[BUFFSIZE]; /* buffer di transizione */

  fdSource = open(argv[1], O_RDONLY);
  fdTarget = open(argv[2], O_WRONLY | O_CREAT, 0600);
  /* copia il file sorgente sul target a blocchi di BUFFSIZE byte */
  while ( (n = read(fdSource, buf, BUFFSIZE)) > 0 )
    if (write(fdTarget, buf, n) != n) {
      perror("write error");
      exit(1);
  }
}
```

Esempio: invert.c

invert <file>: *stampa il file di testo in input invertito.*

```
#include <stdio.h>
#include <sys/file.h>

int main(int argc, char *argv[]) {
  int fd;
  char buff;
  const int charSize = sizeof(char);

  if (argc != 2) {
    printf("invert: Usage invert <txtFile>\n");
    exit(-1); }

  fd = open(argv[1], O_RDONLY);

  lseek(fd, 0, SEEK_END); /* Si posiziona alla fine del file */

  /* legge il file al contrario: ad ogni passo sposta il puntatore di
     due passi indietro, perche' la lettura lo aggiorna */
  while (lseek(fd, -2*charSize, SEEK_CUR) != -1) {
    read(fd, &buff, charSize);
    printf("%c", buff); }
}
```

Atomicità delle operazioni

Se un file (descriptor) è condiviso tra processi possono verificarsi interazioni indesiderate.

Esempio: Append realizzata con

```
lseek(fd, 0, SEEK_END); write(fd, buff,...);
```

Problema: Un altro processo potrebbe [estendere il file / spostare il puntatore] tra la **lseek** *e la scrittura.*

Soluzione: Flag **O_APPEND** *di open.*

Esempio: Creazione di un file se non esiste

```
if open(nome, O_RDONLY)< 0)open(nome,O_CREAT)
```

Problema: Un processo potrebbe creare il file tra le open!

Soluzione: Flag **O_EXECL** *di open con* **O_CREAT**.

Gestione degli errori: perror()

Una system call ritorna -1 se fallisce.

Per gestire gli errori originati dalle system call, i due principali ingredienti da utilizzare sono

`errno`
variabile globale che contiene il codice numerico dell'ultimo errore generato da una system call;

`perror()`
subroutine che mostra una descrizione "testuale" dell'ultimo errore generato dall'invocazione di una system call.

Gestione degli errori: errno

Ogni processo ha una variabile globale **errno**

inizializzata a 0 quando il processo è creato;

se si verifica un errore dovuto ad una system call, ad **errno** *è assegnato un codice numerico corrispondente;*

errno.h *contiene codici di errore predefiniti. Es.*

```
#define  EPERM   1 /* Not owner */
#define  ENOENT  2 /* No such file or directory */
#define  ESRCH   3 /* No such process */
#define  EINTR   4 /* Interrupted system call */
#define  EIO     5 /* I/O error */
```

Una system call che fallisce sovrascrive il valore di **errno.** *Una eseguita con successo ... dipende (meglio salvare l'errore, se serve).*

Gestione degli errori: perror()

`void perror (char *str)`

Mostra la stringa **`str`**, *seguita da ":" e da una stringa che descrive il valore corrente di* **`errno`** *(chiusa da newline).*

Se non ci sono errori da riportare, viene mostrata la stringa **`Error 0`** *(o, in alcuni sistemi,* **`Success`***).*

Non è una system call, ma una routine di libreria.

Per accedere alla variabile **`errno`** *ed invocare* **`perror()`** *occorre includere il file* **`errno.h`**.

I programmi dovrebbero controllare se il valore ritornato da una system call è -1 e, in questo caso, invocare **`perror()`** *per una descrizione dell'errore.*

Esempio: showErrno.c

```
#include <stdio.h>
#include <fcntl.h>
#include <errno.h>

int main(void) {
  int fd;
     /* Open a non-existent file to cause an error */
  fd = open ("nonexist.txt", O_RDONLY);
  if (fd == -1) {    /* fd == -1 => an error occurred */
     printf ("errno = %d\n", errno);
     perror ("main");
  };
  fd = open ("/", O_WRONLY);    /* Force a different error */
  if (fd == -1) {
    printf ("errno = %d\n", errno);
    perror ("main");
  };

  /* continua ... */
```

Gestione degli errori: perror()

```
 /* Execute a successful system call */
  fd = open ("nonexist.txt", O_RDONLY | O_CREAT, 0644);
  printf ("errno = %d\n", errno); /* Display after successful call */
  perror ("main");
  errno = 0;   /* Manually reset error variable */
  perror ("main");
  return 0;
}
```

Output del programma

```
$ showErrno
errno = 2
main: No such file or directory
errno = 21
main: Is a directory
errno = 21
main: Is a directory (?) / Illegal seek!
main: Error 0 / Success
$
```