Programmazione di Sistema e di Rete

Matteo Aprile
Professore: Franco Tommasi

INDICE

I. Libri di testo consigliati

- Advanced Programming in the Unix Environment, 3th ed, Stevens, Rago
- TCP/IP 1, Stevens (facoltativo)
- Unix Networking Programming the Socket Networking API, Stevens
- The Linux Programing Interface, Kerrisk
- manset
- Gapil Guida alla Programmazione in Linux, Simone Piccardi

II. Comandi utili

A. *find: trovare tutti i file eseguibili*

```
$ find . -type f -perm -0100
./standards/makeopt.awk
./standards/makeconf.awk
./proc/awkexample
./systype.sh
./advio/fixup.awk
```

B. *find: trovare file di intestazione del mac come stdio.h*

```
$ find /Applications/Xcode.app/ -name stdio.h 2>/dev
    /null
```

C. *lld: per capire che librerie usa il codice*

```
$ ldd [nomevodice]
```

D. *gcc: per vedere tutta la gerarchia di file in una libreria*

```
$ gcc -H lib.a
```

E. *gcc: per vedere il codice con tutti i file importati*

```
$ gcc -E file.c
```

F. *gcc -g: debugging debole*

```
$ gcc -g -ansi -I../include -Wall -DMACOS -
    D_DARWIN_C_SOURCE  ls1.c -o ls1  -L../lib -lapue
```

G. *gcc -ggbd: debugging forte*

```
$ gcc -ggbd -ansi -I../include -Wall -DMACOS -
    D_DARWIN_C_SOURCE  ls1.c -o ls1  -L../lib -lapue
```

H. *xattr: usato per i file che entrano in quarantena su MacOS*

```
$ xattr -d (delete) com.apple.quarantine [path sh]
```

I. *apropos: per fare un a grep sulla riga name nelle pagine di manuale*

```
$ apropos acl
```

III. Variabili di sistema definite in .bashrc

A. *INC*

```
INC="/Applications/Xcode.app/Contents/Developer/
    Platforms/MacOSX.platform/Developer/SDKs/MacOSX.
    sdk/usr/include/"
```

IV. Introduzione

A. System call

Sono uguali alle funzioni di libreria dal punto di vista sintattico, pero' cambia il modo di compilarle. Notare che non possono essere usati i nomi delle SC per delle function call.

Per poi poter "raccontare" tra umani le sequenze di bit che vengono mandate ai processori si usa assembly.

Sono effettivamente delle chiamate a funzioni ma poi dal codice assembly puoi capire che è una system call dato che ha dei meccanismi specifici.

Alcuni esempi di chiamate e registri:

- **eax** : registro dove metti il **numero della sc**
- **int 0x80**: **avvisa il kernel** che serve chiamare una sc
- **exit()**: chiudere un processo
- **write()**:

```
mov edx,4     ; lunghezza messaggio
mov ecx,msg   ; puntatore al messaggio
mov ebx,1     ; file descriptor
mov eax,4     ; numero della sc
int 0x80
```

dove nel file descriptor indichi a quale file devi mandare l'output. Questo viene usato dato che così non deve cercare il path ogni volta ma lo mantiene aperto riferendosi ad esso tramite un numero, cioe' il piu' piccolo disponibile.

B. Programma Make

Quando viene avviato verifica la presenza di un file chiamato "Makefile", oppure si usa 'make -f'. In questo file ci sono le regole di cosa fare per automatizzare delle azioni per un numero n di file. Se, durante la compilazione di massa, una di queste da un errore il programma make si interrompe, per evitare ciò si usa '-i' (ignore).

Il Makefile andrà ad aggiornare una libreria andando a guardare se una delle 3 date di ultima modifica si sono aggiornate.

Andiamo a guardare cosa contiene Makefile:

```
DIRS = lib intro sockets advio daemons datafiles db
    environ \
    fileio filedir ipc1 ipc2 proc pty relation
    signals standards \
    stdio termios threadctl threads printer
    exercises

all:
    for i in $(DIRS); do \
        (cd $$i && echo "making $$i" && $(MAKE) ) ||
     exit 1; \
    done

clean:
    for i in $(DIRS); do \
        (cd $$i && echo "cleaning $$i" && $(MAKE)
    clean) || exit 1; \
    done
```

dove:

- DIRS: lo si associa alle **stringhe singole** che gli sono state associate
- all: nel ciclo for:
  - manda un comando in subshell
  - $$i: riferimento alla variabile "i" del for + simbolo escape per il Makefile
  - $(MAKE): macro predefinita per i Makefile

La struttura è:

```
target: prerequisiti
    rule
```

dove:

- target: è l**a cosa che si vuole fare**, se essendo il primo target, sarà anche quello di default
- prerequisiti: **file e/o target** a loro volta
- rule: indica **cosa puo' fare il target**

Può capitare che prima di eseguire il Makefile ci sia uno script "configure".

In molti casi si ha un target "clean" che permette di pulire i file .o che sono inutili dopo la compilazione, o comunque qualsiasi tipo di file gli si voglia far eliminare. Questo tipo di target che non rappresentano un file, sono detti "phony" perchè fasulli, dato che non sono file ma sole parole

```
file: file.o lib.o

clean:
    rm file.o
```

Abbiamo delle variabili automatiche per rendere il lavoro più facile:

- **$@**: per riferirsi il target
- **$?**: tutti i prerequisiti più recenti del target
- **$^**: tutti i prerequisiti del target
- https://www.gnu.org/software/make/manual/make.html#Automatic-Variables

Un altro esempio di Makefile è:

```
ROOT=..
PLATFORM=$(shell $(ROOT)/systype.sh)
include $(ROOT)/Make.defines.$(PLATFORM)

PROGS = getcputc hello ls1 mycat shell1 shell2
    testerror uidgid

all:    $(PROGS)

%:  %.c $(LIBAPUE)
    $(CC) $(CFLAGS) $@.c -o $@ $(LDFLAGS) $(LDLIBS)

clean:
    rm -f $(PROGS) $(TEMPFILES) *.o

include $(ROOT)/Make.libapue.inc
```

dove:

- ROOT: cwd
- PLATFORM: assumera in valore del OS: macos/linux
- include: include un file
- PROGS: elenco dei programmi da usare
- %: target con nome variabile, indica un file
- %.c: target con nome variabile ma estensione .c
- $(CC): indica il compilatore dove cc è un link simbolico a clang
- $(CFLAGS) indica una macro predefinita dei default vuota che si può usare all'occorrenza

- all: target che prende in carico tutti i programmi che se saranno di tipo .c saranno presi in carico dal target successivo

Per la compilazione dei file, qualsiasi sia il linguaggio, make sapra' come compilarlo grazie a tutte le definizioni di default presenti in:

```
$ make -p
```

notare che **si puo' mettere un comando custom nelle rule** del target

Nell'eventualità di voler aggiornare un solo file della libreria senza far aggiornare il resto ci bastera' usare uno script che compila quel file passato a linea di comando.

```
$ gcc -ansi -I../include -Wall -DMACOS -
    D_DARWIN_C_SOURCE  ${1}.c -o ${1}  -L../lib -
    lapue
```

C. Direttive di preprocessore

Sono delle indicazioni date a gcc prima di iniziare la compilazione.

Iniziano tutte con '#':

- **#include**: serve ad **includere delle librerie** di sistema (<lib.h>) oppure di librerie fatte da noi e non in directory standard ("lib.h")
- **#define**:
  - permette di **creare delle "macro"**, che vanno a sostituire una stringa con un'altra (es: #define BUFLEN), può capitare che debbano essere definite delle macro prima che si compili il programma, in questi casi si usa scrivere es: '-DMACOS'
  - permette di **creare delle "function like macro"** (es: #define ABSOLUTE_VALUE(x) (((x<0)?-(x):(x))
- **#ifdef, #ifndef, #endif**: usata per far accadere qualcosa nel caso un macro sia stata definita

```
#ifdef VAR
print("hello");
#endif
```

Per evitare che piu' file includano lo stesso si usano degli #ifndef in tutto il codice, in modo da evitare doppie definizioni.

D. Librerie

Durante la fase di compilazione creiamo dei file oggetto (.o) per ogni file in cui è scritta la descrizione delle funzioni di libreria (.c)

```
$ gcc -c bill.c
```

Si andrà poi a creare il prototipo della funzione (.h).

In fine tramite il linker si andranno ad unire tutti i file per crearne uno unico con tutte le definizioni delle funzioni incluse nelle librerie, di sistema e non, importate. Si vanno quindi a sciogliere tutti i riferimenti incrociati.

```
$ gcc -o program program.o bill.o
```

Per quanto riguarda le funzioni di sistema NON abbiamo il file sorgente ma abbiamo direttamente l'eseguibile. In compenso abbiamo un file di libreria, cioè un insieme di file oggetto linkati in un unico file, dove c'è il codice oggetto di tutte le funzioni.

Abbiamo **2 tipi di librerie**:

- statiche: è una **collezione di file oggetto** che hanno il codice compilato delle funzioni e che verranno **linkati al momento della compilazione**. Il programma che si crea sarà possibile essere eseguito solo sullo stesso OS.
  Il **problema si ha nell'aggiornamento delle librerie al momento della scoperta di un bug**. Una volta coretto servirà ricevere la versione corretta per poter aggiornare il programma.
- dynamic: ricordano il concetto di plug-in, quindi **viene invocato a runtime e caricato in memoria** (es: aggiornamenti dei OS). **L'eseguibile non viene toccato la correzione avviene solo nella libreria**.
  Il requisito maggiore è che chi si passa il codice debba avere lo stesso OS dell'altro utente. Notare che **non cambia il prototipo** dato che sennò bisognerà ricompilare l'intero programma.

In generale le librerie statiche sono molto pericolose infatti alcuni OS le aboliscono per le questioni di sistema. Su linux si ha come libreria statica 'lib.c' che è la libreria con le funzioni più usate in c. Per macos è stata abolita.

Per compilare con la versione dinamica non servono opzioni, per la statica si usa:

```
$ gcc -static
```

E. Creazione librerie

Per costruire una libreria statica per MacOS:

1) costruiamo il **file oggetto**:

```
$ gcc -c libprova.c
```

2) costruiamo la **libreria** (con ar=archive, c=create se lib.a non esiste):

```
$ ar rcs libprova.a libprova.o
```

3) costruire il **codice** che usa la libreria (con -Wall=verbose warning, -g=debugging, -c=create del file):

```
$ gcc -Wall -g -c useprova.c
```

4) **linker** per risolve le chiamate incrociate (con -L.=dove prendere la libreira, -l[nomelib]=usare la libreria):

```
$ gcc -g -o useprova useprova.o -L. -lprova
```

Per capire che librerie usa il codice si usa:

```
$ otool -L [nomecodice]
```

Per costruire una libreria statica per Linux:

1) costruiamo il **file oggetto**:

```
$ gcc -fPIC -Wall -g -c libprova.c
```

2) costruiamo la **libreria** (con 0.0=versione della libreira):

```
$ gcc -g -shared -Wl,-soname,libprova.so.0 -o
    libprova.so.0.0 libprova.o -lc
```

3) costruire il **link simbolico per aggiornare le librerie** senza aggiornare gli eseguibili e senza cambiare il nome del programma:

```
$ ln -sf libprova.so.0.0 libprova.so.0
```

4) **linker** per risolve le chiamate:

```
$ ln -sf libprova.so.0 libprova.so
```

Per capire che librerie usa il codice si usa:

```
$ ldd [nomevodice]
```

F. Aggiornamento librerie

Su **Linux** il sistema andra' a prendere direttamente una libreria dinamica, per evitare ciò e far trovare la nostra, basterà impostare una variabile di ambiente:

```
LD_LIBRARY_PATH=`pwd` ldd useprova
```

Tipicamente la libreria viene distribuita nelle directory di sistema andandola ad "installare".

Su **MacOS** la libreria dinamica è un .dylib:

```
$ gcc -dynamiclib libprova.c -o libprova.dylib
```

Quindi eseguendo il programma trovera' la libreria controllando nella directory corrente e quindi non serve creare la variabile di ambiente come su Linux.

i file di intestazione del mac come stdio.h per cercarla uso:

```
$ find /Applications/Xcode.app/ -name stdio.h 2>/dev
    /null
```

V. System call

A. Funzioni e system call

Se prendiamo un funzionamento più semplice del comando "ls" potrebbe essere:

```
#include "apue.h"
#include <dirent.h>

int
main(int argc, char *argv[])
{
    DIR             *dp;
    struct dirent   *dirp;

    if (argc != 2)
        err_quit("usage: ls1 directory_name");

    if ((dp = opendir(argv[1])) == NULL)
        err_sys("can't open %s", argv[1]);
    while ((dirp = readdir(dp)) != NULL)
        printf("%s\n", dirp->d_name);

    closedir(dp);
    exit(0);
}
```

dove abbiamo che:

- DIR: struttura dati
- struct dirent: **tipo struttura** che contiene al suo interno diversi tipi di variabili.

  Per capire se è una funzione di sistema lanciamo:

```
$ grep -rw "struct dirent" $INC

/Applications/Xcode.app/Contents/Developer/
    Platforms/MacOSX.platform/Developer/SDKs/
    MacOSX.sdk/usr/include//sys/dirent.h:struct
    dirent {
```

```
#ifndef _SYS_DIRENT_H
#define _SYS_DIRENT_H

#include <sys/_types.h>
#include <sys/cdefs.h>

#include <sys/_types/_ino_t.h>


#define __DARWIN_MAXNAMLEN      255

#pragma pack(4)

#if !__DARWIN_64_BIT_INO_T
struct dirent {
    ino_t d_ino;                    /* file
    number of entry */
    __uint16_t d_reclen;            /* length of
     this record */
    __uint8_t  d_type;              /* file type
    , see below */
    __uint8_t  d_namlen;            /* length of
     string in d_name */
    char d_name[__DARWIN_MAXNAMLEN + 1];    /*
    name must be no longer than this */
};
#endif /* !__DARWIN_64_BIT_INO_T */

#pragma pack()

#define __DARWIN_MAXPATHLEN     1024

#define __DARWIN_STRUCT_DIRENTRY { \
```

```
    __uint64_t  d_ino;      /* file number of
    entry */ \
    __uint64_t  d_seekoff;  /* seek offset (
    optional, used by servers) */ \
    __uint16_t  d_reclen;   /* length of this
    record */ \
    __uint16_t  d_namlen;   /* length of string
    in d_name */ \
    __uint8_t   d_type;     /* file type, see
    below */ \
    char      d_name[__DARWIN_MAXPATHLEN]; /*
    entry name (up to MAXPATHLEN bytes) */ \
}

#if __DARWIN_64_BIT_INO_T
struct dirent __DARWIN_STRUCT_DIRENTRY;
#endif /* __DARWIN_64_BIT_INO_T */



#if !defined(_POSIX_C_SOURCE) || defined(
    _DARWIN_C_SOURCE)
#define d_fileno        d_ino           /*
    backward compatibility */
#define MAXNAMLEN       __DARWIN_MAXNAMLEN
/*
 * File types
 */
#define DT_UNKNOWN       0
#define DT_FIFO          1
#define DT_CHR           2
#define DT_DIR           4
#define DT_BLK           6
#define DT_REG           8
#define DT_LNK          10
#define DT_SOCK         12
#define DT_WHT          14

/*
 * Convert between stat structure types and
    directory types.
 */
#define IFTODT(mode)    (((mode) & 0170000) >>
    12)
#define DTTOIF(dirtype) ((dirtype) << 12)
#endif


#endif /* _SYS_DIRENT_H  */
```

dove vediamo che se la variabile "__DARWIN_64_BIT_INO_T" è stata definita avremo che la struttura di struct dirent è:

```
#define __DARWIN_STRUCT_DIRENTRY { \
    __uint64_t  d_ino;      /* file number of
    entry */ \
    __uint64_t  d_seekoff;  /* seek offset (
    optional, used by servers) */ \
    __uint16_t  d_reclen;   /* length of this
    record */ \
    __uint16_t  d_namlen;   /* length of string
    in d_name */ \
    __uint8_t   d_type;     /* file type, see
    below */ \
    char      d_name[__DARWIN_MAXPATHLEN]; /*
    entry name (up to MAXPATHLEN bytes) */ \
}

#if __DARWIN_64_BIT_INO_T
struct dirent __DARWIN_STRUCT_DIRENTRY;
#endif /* __DARWIN_64_BIT_INO_T */
```

- if: esegue un controllo sugli args. Notiamo che "err_quit" non è una funzione di sistema da:

```
$ grep -rw "err_quit" $INC
```

  infatti non restituisce nulla. Deve allora essere una funzione di libreria create da noi quindi non presente nella directory standard.
  La funzione andrà a dare un messaggio di errore e poi esce dal programma.
- opendir: serve ad aprire una directory andandola a caricare nella RAM.
- while: leggiamo la directory e la inseriamo nella struttura che poi sarà richiamata tramite:

```
dirp->d_name
```

  dove "d_name" è il nome dello slot in cui è contenuto il nome del file.
- exit: restituisce l'exit code del programma

B. Capire se una funzione è una system call

Andiamo a vedere se e' una funzione o una system call tramite "man", lo si capisce tramite la dicitura in alto alla pagina del manuale:

- **Library Functions** Manual
- **System Calls** Manual

Abbiamo anche esempi piu' particolari, come fork, dove è indicata come system call ma in realtà le richiama ma in prima persona.

Potremo trovare i simboli di una libreria tramite:

```
$ nm lib.a
```

che ci fa vedere, per ogni file oggetto, i simboli associati per ogni funzione.

Le system call le troveremo in "$INC/sys/syscall.h"

C. Numeri dei file descriptor

Prendiamo un esempio semplificato del comando "cat":

```
#include "apue.h"

#define BUFFSIZE    4096

int
main(void)
{
    int     n;
    char    buf[BUFFSIZE];

    while ((n = read(STDIN_FILENO, buf, BUFFSIZE)) >
     0)
        if (write(STDOUT_FILENO, buf, n) != n)
            err_sys("write error");

    if (n < 0)
        err_sys("read error");

    exit(0);
}
```

ogni processo ha 3 file descriptor usati 0, 1, 2.

- BUFFSIZE: macro di preprocessore
- read: **system call** con parametri:
  - **STDIN_FILENO**: file descriptor per dire da quale "numero di deve leggere" si vuole leggere. Cioè per leggere dal file indicato nello standard input

  - **buf**: indirizzo dell'**inizio dell'array**
  - **BUFFSIZE**: quando deve leggere

  Restituisce il numero di char che ha letto, dato che potrebbe leggere meno byte di quelli richiesti nel caso in cui il file ne contenga di meno. Ad ogni sua iterazione si ricorda la "posizione nel file" che gli permette di non leggere sempre i primi n byte ma di rincominciare da dove ha lasciato.
- write: richiede gli stessi valori di read tranne per **STDOUT_FILENO** e **ritorna il numero byte effettivamente letti**
-

per capire quanto vale STDIN_FILENO:

```
$ grep -rw "STDIN_FILENO" $INC

/Applications/Xcode.app/Contents/Developer/Platforms
    /MacOSX.platform/Developer/SDKs/MacOSX.sdk/usr/
    include//unistd.h:#define     STDIN_FILENO    0
       /* standard input file descriptor */
/Applications/Xcode.app/Contents/Developer/Platforms
    /MacOSX.platform/Developer/SDKs/MacOSX.sdk/usr/
    include//asl.h: * asl_log_descriptor(c, m,
    ASL_LEVEL_NOTICE, STDIN_FILENO,
    ASL_LOG_DESCRIPTOR_READ);
```

Sappiamo che un processo per eseguire un programma, esegue prima una fork e poi con exec esegue il programma. Prima di eseguire la fork il child chiude il file 1 e quando fa una open, la system call prenderà il file nel quale reindirizzare lo STDOUT e restituirà il numero 1.

Su questo sistema si base UINX infatti avviene anche con le pipe "—". Permette di creare programmi complessi unendo tanti piccoli programmi specializzati in un'unica funzione.

È molto importante capire che i child ereditano i file descriptor dei parent quindi non è necessario che il programma corrente faccia una open dei file descriptor.

D. Meccanismi dei file

Un file è in insieme di meccanismi:

- apri
- leggi
- scrivi
- chiudi

Questi meccanismi sono applicabili a file, cartelle, stampanti ecc..., solo che per ogni "tipo" i 4 meccanismi si adeguano a ciò che il caso particolare deve fare.

E. Unbuffered I/O

Le system call rappresentano una barriera tra kernel e programmi, dove avremo rispettivamente due diverse modalita' di utilizzo:

- **kernel mode**: ha tutti i privilegi
- **user mode**: non può accedere a tutte le celle di memoria

Per ottimizzare la scrittura sulla memoria da parte del kernel si utilizza la libreria STDIOLIB che incrementa le prestazioni dato che gestisce il passaggio di pacchetti con il kernel in modo da inviare dei pacchetti consistenti ogni tot e non piccoli pacchetti soni secondo. Per fare ciò usa un buffered i/o che, una volta riempiti dei buffer, gli manda al kernel.

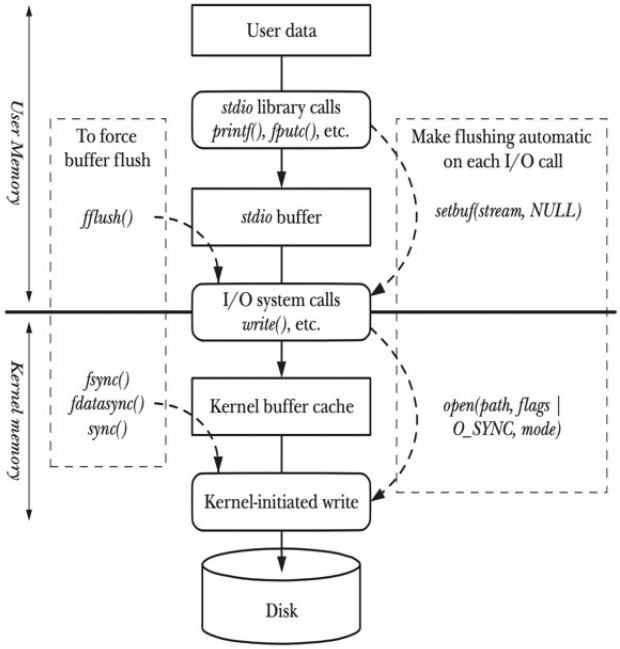

Figura 1. Schema unbuffered I/O

F. Fork & Exec

Prendiamo il codice di shell1.c che crea uno schell dal quale poter eseguire programmi:

```
#include "apue.h"
#include <sys/wait.h>

int
main(void)
{
    char    buf[MAXLINE];   /* from apue.h */
    pid_t   pid;
    int     status;

    printf("%% ");  /* print prompt (printf requires
      %% to print %) */
    while (fgets(buf, MAXLINE, stdin) != NULL) {
        if (buf[strlen(buf) - 1] == '\n')
            buf[strlen(buf) - 1] = 0; /* replace
    newline with null */

        if ((pid = fork()) < 0) {
            err_sys("fork error");
        } else if (pid == 0) {      /* child */
            execlp(buf, buf, (char *)0);
            err_ret("couldn't execute: %s", buf);
            exit(127);
        }

        /* parent */
        if ((pid = waitpid(pid, &status, 0)) < 0)
            err_sys("waitpid error");
        printf("%% ");
    }
    exit(0);
}
```

avremo allora:

- fgets: funzione dello **stdoutput che legge la stringa** che dai prima di dare invio, quando una system call viene interrotta la fgets restituisce null facendo fermare il loop. Ha come argomenti:
  - **buf**: buffer nel quale mettere la stringa
  - **MAXLINE**: proviene da una nostra libreria

```
$ grep -rw "MAXLINE" include/
Binary file include//apue.h.gch matches
include//apue.h:#define MAXLINE 4096
        /* max line length */
```

  - **stdin**: presente in stdiolib ed è una **struttura file che definisce uno standard input** tramite un puntatore ad un "file"

- if 1: permette di avere un null dove prima avevamo \n
- if 2: abbiamo una fork() che dopo che viene invocata ritorna 2 volte, questo perché andrà a creare 2 bash identici con memorie uguali nei contenuti ma indipendenti, l'unico cambiamento è il pid. fork() andrà quindi a restituire 0 nel child e il pid del child al parent tramite getppid().
  Se pid < 0 vorrà dire che la fork e fallita. Se pid $= 0$ vorrà dire che siamo nel child.
  Il che è molto importante dato che il codice verra' eseguito sia dal child che dal parent, e sarà contenuto nella memora virtuale che hanno i programmi grande 2^{32} o 2^{64} in base all'OS.
  Appena viene eseguita l'exec, lo spazio di memoria viene azzerato ma a discrezione del programma, vengono salvate alcune variabili di ambiente.
  execlp: serve a far **eseguire un codice** (buf) del quale abbiamo il sorgente e l'eseguibile
  if 3: serve a far andare aventi il parent.
  waitpid: aspetta il child nel caso impieghi troppo tempo ad eseguire la sua azione, **tenendo appeso il prompt**. Con argomenti:
  - **pid**: pid del child
  - **&status**: reindirizza l'exit code del child, quando finisce, nella variabile status

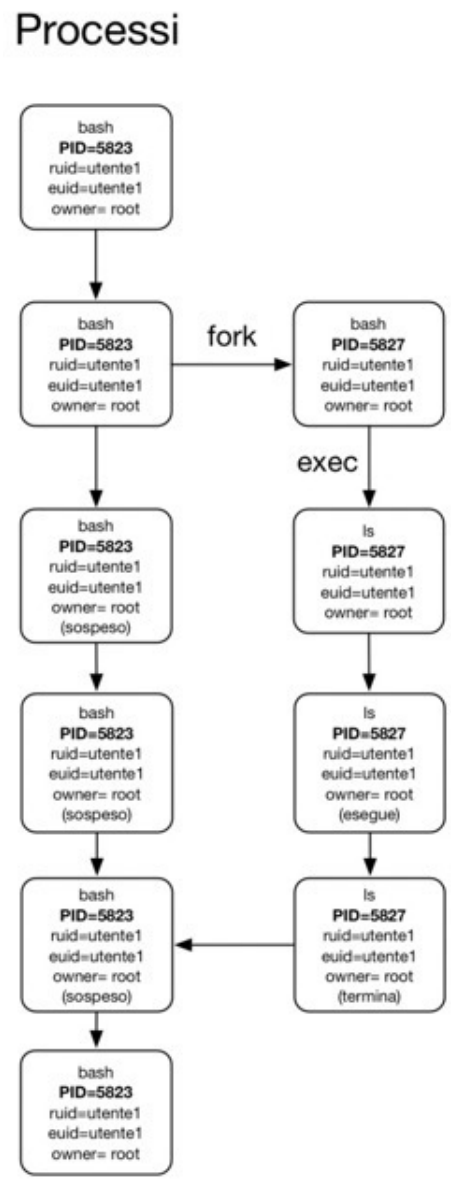

Figura 2. Esecuzione di fork ed exec

G. Thread

Sono dei processi con lo stesso spazio di memoria del parent. Per evitare che ogniuno scriva dove vuole, avviene una sincronizzazione tra le thread. Questo metodo viene usato nelle macchine unicore per poter svolgere più operazione "contemporaneamente".

Tutto questo è orchestrato dal kernel che gestisce il time shearing.

H. Gestione degli errori

Per convenzione una funzione ritorna 0 se è andato tutto bene. Ci sono delle eccezioni, come la read, che ritorna il numero di byte letti.

Ogni valore possibile ritornato è specificato nel manuale:

```
$ man 2 intro
...
1 EPERM Operation not permitted. An attempt was made
     to perform an operation limited to processes
    with appropriate privileges or to the owner of a
     file or other resources.

2 ENOENT No such file or directory. A component of a
     specified pathname did not exist, or the
    pathname was an empty string.
...
```

Per le system call, quando avviene un errore, si avvalora la variabile "errno" che può essere consultata in un programma con

```
extern int errno;
```

Con "errno" bisogna tenere in conto che:

1) non viene svuotata quando passiamo l'errore. Quindi per sapere quando è stato dato un errore bisogna andare a consultarla quando la system call viene invocata
2) vale 0 se non usata

Per la gestione degli errori useremo:

- strerror: restituisce la **stringa del valore** di errno
- perror: legge errno e stampa un messaggio a piacere

```
#include "apue.h"
#include <errno.h>

int
main(int argc, char *argv[])
{
    fprintf(stderr, "EACCES: %s\n", strerror(EACCES)
    );
    errno = ENOENT;
    perror(argv[0]);
    exit(0);
}
```

dove:

- fprinf: stampa un errore allo standard specificato

I. Keyword in C ed accessi a variabili

In C le variabili con la keyword "const" serve a dare l'accesso ad una variabile ma in sola lettura. Ciò che la frena è il compilatore.

Se una funzione vuole lavorare anche al suo esterno usiamo:

1) attraverso il **passaggio di variabili dai parametri**
2) vengono passati gli **indirizzi delle variabili** abilitandone la scrittura
3) tramite variabili globali

```
const int *x
```

Un'altra keyword è "restrict" che serve a non far creare copie di una variabile, questo per fare in modo che il compilatore non si possa confondere con le copie di quella variabile.

```
int *restrict x
```

J. Segnal e interupt

Guardiamo il file shell2.c :

```
#include "apue.h"
#include <sys/wait.h>

static void sig_int(int);       /* our signal-
    catching function */

int
main(void)
{
    char    buf[MAXLINE];   /* from apue.h */
    pid_t   pid;
    int     status;

    if (signal(SIGINT, sig_int) == SIG_ERR)
        err_sys("signal error");

    printf("%% ");  /* print prompt (printf requires
     %% to print %) */
    while (fgets(buf, MAXLINE, stdin) != NULL) {
        if (buf[strlen(buf) - 1] == '\n')
            buf[strlen(buf) - 1] = 0; /* replace
    newline with null */

        if ((pid = fork()) < 0) {
            err_sys("fork error");
        } else if (pid == 0) {      /* child */
            execlp(buf, buf, (char *)0);
            err_ret("couldn't execute: %s", buf);
            exit(127);
        }

        /* parent */
        if ((pid = waitpid(pid, &status, 0)) < 0)
            err_sys("waitpid error");
        printf("%% ");
    }
    exit(0);
}

void
sig_int(int signo)
{
    printf("interrupt\n%% ");
}
```

dove rispetto a shell1 abbbiamo come differenze:

- dichiarazione di funzione per gestire un segnale
- if 1: gestisce un segnale di errore
  - signal: gli diciamo che se arriva il segnale SIGINT allora eseguire la nostra funzione sig_int. In caso contrario, viene restituito SIG_ERR e la variabile globale errno viene impostata per indicare l'errore.
  - sig_int: avrà in ingresso un numero intero, che rappresenta il segnale, dato dal kernel.
- definizione della funzione dove viene gestito tramite una stampa

I processi si interfacciano con il modo esterno tramite delle funzionalità dette Inter Process Communication (IPC) alcune di queste sono i segnali.

I segnali fanno parte dei software interrupt dove il kernel:

1) interrompe l'esecuzione di un porcesso
2) esegue il codice definito per quell'interrupt
3) se non ci sono danni al processo questo riprende da dove era stato interrotto

Per quanto riguarda le hardware interrupt intendiamo i segnali di interrupt, dati dalle periferiche, che arrivano al processore. L'interrupt e' un numero identificativo che fa capire la natura di quell'interrupt tramite una tabella degli interrupt dove ad ogni numero corrisponde l'indirizzo ad una routine.

Il kernel invia tutti i segnali che saranno causati da condizioni particolari o accessi a memoria non autorizzati ma alle quali ha accesso (es: malloc che richiede l'accesso a memoria).

Come programmatori bisognerà occuparsi di gestire i segnali tramite:

1) azione di **default**
2) **ignorare** il segnale
3) gestione del segnale **scrivendo il signal endler**

Tramite il manuale di "signal" possiamo vedere tutti i tipi di segnale che esistono. Come si può notare alcuni hanno le diciture:

- **terminate process**: termina il processo
- **create core image**: effettua una fotocopia del core prima di terminare il processo in modo da poter effettuare un debug

Potremo inviare i segnali tramite il comando "kill". Per esempio:

- SIGINT: ctrl C
- SIGQUIT: ctrl \
- SIGSTOP: Ctrl Z

K. Valori del tempo

Il sistema gestisce il tempo in secondi a partire dalla Epoch 01/01/1970 e per ogni processo dà:

- clock time: **tempo effettivo di esecuzione** del processo da quando è nato a quando è terminato
- user clock time: tempo che **non richiede l'intervento del kernel** (speso dalla CPU)
- system clock time: tempo che **richiede l'intervento del kernel**

dove **il clock time non è la somma di user e system** dato che non si contano i processi che intervengono nel mezzo. La loro somma può essere magiore del clock time se abbiamo un porcessore multicore dato che somma il tempo dai diversi core (il tempo potrebbe essere diverso in base al core).

```
$ time [nome programma]

...
real    0m0.006s
user    0m0.001s
sys     0m0.005s
```

VI. Gli standard

A. Storia e basi

La standardizzazione di UNIX e' iniziata nel 1988 facendo affidamento ad alcuni standard di C dato che fa usi di interfaccie e prototipi.

In definitiva abbiamo gli standard:

- Posix.1-2001 / SUSv3: (http://pubs.opengroup.org/onlinepubs/009604599/)
- Posix.1-2008 / SUSv4: più usato in ambiti di automazioni aziendali, infatti sono specializzate sullo scambio di informazione in segnali realtime. Per questo la sua certificazione non è stata presa da nessuno se non fa un IBM. (http://pubs.opengroup.org/onlinepubs/9699919799/)

(PS: le versioni sono back compatibili quindi se settiamo -D_XOPEN_SOURCE=700 non precludiamo la SUSv3)

Nonostante gli standard ogni OS fa delle sue modifiche su alcune cose esterne alle SUS.

Per verificare il tipo di standard su un applicativo (_XOPEN_SOURCE) o un sistema (_XOPEN_VERSION), si fa affidamento alle "feature test macros" consultabili dai **codici di intestazione .h**. Per esempio _XOPEN_SOURCE impostata a 600 o 700 indica SUSv3 o SUSv4.

```
-D_XOPEN_SOURCE=600
```

in questo modo potremo allora andare a compilare tutti i programmi conformi su qualsiasi OS.

B. Limiti

Abbiamo dei limiti di compilazione che possono essere visti nei file di intestazione.

Possiamo visualizzare i runtime limit che tramite le funzioni: sysconf (es: lunghezza massima del nome dei file che dipende dal filesystem può capirlo tramite pathconf su un file qualunque di quel filesystem)

- sysconf: usato per **determinare il valore corrente di un limite**
- pathconf: da **informazioni sul file system** e per fare ciò gli serve poter arrivare ad un qualunque file del filesystem
- fpathconf: come pathconf ma **prende anche il file descriptor**

Tutti e 3 prendono come parametro:

```
int name
```

che restituisce una chiave in base a cosa si vuole indagare. In pratica fanno riferimento ad un nome simbolico che si riferisce ad un valore. Tutte queste chiamate fanno si di avere più **portabilita'**. Se queste chiamate sono fatte da file include vincono sempre quelli di sysconf. Saranno precedute da "_SC_" per i sysconf e da "_PC_" per i pathconf.

C. Determinare l'allocazione

Supponendo di avere bisogno di uno spazio dove mettere un nome di file (path) per poterlo gestire. Tramite la funzione "path_alloc" ci faremo restituire un puntatore ad una memoria capace di contenere il massimo dei caratteri gestibili dal sistema (es: $2^32 \rightarrow 32$ bit, $2^64 \rightarrow 64$ bit).

Per vedere qual è la lunghezza usiamo la variabile limite:

```
pathconf(_PC_NAME_MAX)
```

in genere avremo NAME_MAX = 255.

Lo stesso lavoro di riferimenti simbolici uno dopo l'altro è "pid_t", per poter lasciare piu' liberta' al programmatore:

```
$ grep -rw "pid_t" $INC | grep typedef

/sys/_types/_pid_t.h:typedef __darwin_pid_t pid_t;
/sys/_types.h:typedef __uint32_t __darwin_id_t;

$ grep -rw "__uint32_t" $INC | grep typedef

/i386/_types.h:typedef unsigned int __uint32_t;
```

tutti questi rimandi sono dati dalla portabilita'.

VII. File I/O

A. Chiamata open

I file I/O sono le funzioni che gestisce buffered I/O ed in contrapposizione da quelle della libreria "stdlib". La chiamata open() fa parte di queste funzioni, i suoi argomenti sono:

- arg: path assoluto o relativo
- flag: bit che indica l'**attivazione di alcune modalita'**. Si avrà allora a **settare il bit della flag a 1**:
- mode: serve a dare i **privilegi con cui i file deve essere creato** (**da usare solo nella creazione del file**)

```
open(file, O_RDWR | O_APPEND | O_CREAT | O_TRUNC
    , file_mode)
```

avremo allora 11000001010 con:

O_RDWR = 2
O_APPEND = 8
O_CREAT = 512
O_TRUNC = 1024

B. Read e write flag

I bit delle flag abbiamo un modo "scomodo" per rappresentarle dato che non seguono la normale "accensione dei singoli bit":

```
#define O_RDONLY   0x0000 /* reading only */
#define O_WRONLY   0x0001 /* writing only */
#define O_RDWR     0x0002 /* reading and writing */
#define O_ACCMODE  0x0003 /* above modes */
```

sono dette maschere per leggere o scrivere.

Per quanto riguarda la chiamata read ci sono più casi in cui il numero di byte restituito e' minore di quello chiesto:

1) possiamo avere un valore di ritorno di 50 se chiedo di leggere 100 perché **ci sono solo 50 byte**
2) quando legge da un **terminale**: la read ritorna quando dai invio
3) quando legge da una **rete**: puoi leggere 100 byte ma se ne leggi 10 interrompe la lettura e ritorna, invece se non arriva nulla rimane in attesa
4) quando legge da una **pipe**: se non arriva nulla rimane appesa, se arriva qualcosa interrompe e restituisce quello che ha letto
5) quando **interrotta da un segnale** e alcuni dati sono stati letti gli restituisce

C. Chiamata openat()

Prende un file descriptor (passato dalla open() sulla directory) di una directory per passare il path relativo a quella directory. La sua falla nel sistema sta nella possibilità di continuare ad accedere a file anche dopo che sono cambiati i privilegi, dato che lasciando aperta la sessione del file non saremo soggetti ai nuovi privilegi.

Questa chiamata è interessante per le Thread che vogliono lavorare in un loro ambiente.

D. Open flags

Abbiamo un certo numero di flag standard dichiarate dall'SUSv3, il resto possono essere a discrezione dei sistemi UNIX.

Le flags più usate sono:

- O_DIRECTORY: **limita la chiamata** open ad una directory specifica
- O_CREAT: flag per dire che si vuole **creare un file**
- O_TRUNC: se vuoi creare un file nuovo ed ne esiste già uno, il **vecchio viene azzerato**
- O_EXCL: se il file già esiste fa **fallire la chiamata**

E. Builtin umask

È un valore presente in ogni processo ed ereditato dal parent ma il child può comunque modificarla. Il valore restituito è un numero ottale (inizia con 0):

```
$ umask
0022

$ ll file
-rw-r--r-- 1 docente staff 0 14 Ott 09:02 file
```

quindi la umask taglia i permessi dei file creati da quel processo. Quindi, in questo caso, un 666 diventa 644.

F. Chiamata lseek()

Per un file appena creato la sua "current position" si trova all'inizio del file, ci si potra' muovere nel file tramite lseek(). Gli argomenti sono:

1) file descriptor
2) offset: per dire **dove ci si vuole spostare**
3) whence: indica **da quale punto** si deve applicare l'offset:
   - SEEK_SET: valore preciso da dove partire
   - SEEK_CUR: presa la current position inserire un **gap e poi scrivere** (può essere negativo)
   - SEEK_END: gap dal quale inserire **rispetto alla fine** del file. Se negativo scrivo prima, se positivo posso lasciare un **buco di byte** e poi scrivere. Nei nuovi sistemi i blocchi vuoti vengono allocati.

Un esempio di buco in un file dato da un numero positivo con SEEK_END:

```
#include "apue.h"
#include <fcntl.h>

char    buf1[] = "abcdefgh";
char    buf2[] = "ABCDEFGH";

int
main(void)
{
    int     fd;

    if ((fd = creat("file.hole", FILE_MODE)) < 0)
        err_sys("creat error");

    if (write(fd, buf1, 10) != 10)
        err_sys("buf1 write error");
    /* offset now = 10 */

    if (lseek(fd, 16384, SEEK_SET) == -1)
        err_sys("lseek error");
    /* offset now = 16384 */

    if (write(fd, buf2, 10) != 10)
        err_sys("buf2 write error");
    /* offset now = 16394 */

    exit(0);
}
```

avremo:

```
$ xxd file.hole
00000000: 6162 6364 6566 0000    abcdefgh........
00000010: 0000 0000 0000 0000    ................
00000020: 0000 0000 0000 0000    ................
00000030: 0000 0000 0000 0000    ................
00000040: 0000 0000 0000 0000    ................
00000050: 0000 0000 0000 0000    ................
00000060: 0000 0000 0000 0000    ................
00000070: 4142 4344 4546 4748    ABCDEFGH........
```

abbiamo che la memoria sul disco è:

```
$ du -h file.hole
1.6M    file.hole
```

invece la size del file è:

```
$ stat -x file.hole
  File: "file.hole"
  Size: 1638410      FileType: Regular File
  Mode: (0644/-rw-r--r--)         Uid: (  501/
    matt)  Gid: (   20/   staff)
Device: 1,16   Inode: 27657406    Links: 1
Access: Fri Oct 14 09:59:17 2022
Modify: Fri Oct 14 09:57:59 2022
Change: Fri Oct 14 09:57:59 2022
 Birth: Fri Oct 14 09:47:24 2022
```

Possiamo avere dei file descriptor seekble se il file è regolare o meno:

```
#include "apue.h"

int
main(void)
{
    if (lseek(STDIN_FILENO, 0, SEEK_CUR) == -1)
        printf("cannot seek\n");
    else
        printf("seek OK\n");
    exit(0);
}
```

G. I/O efficiency

Se devo trasferire una grande quantita' di dati avrò una dimensione ottimale con la quale posso ottimizzare il passaggio dei blocchi di dati. I tempi possono essere misurati con "time" per capire in che modo il nostro GB debba essere sezionato (in K, M, ...).

(provare a fare ciò tramite intro/mycat e tramite time capire le tempistiche (A CASA))

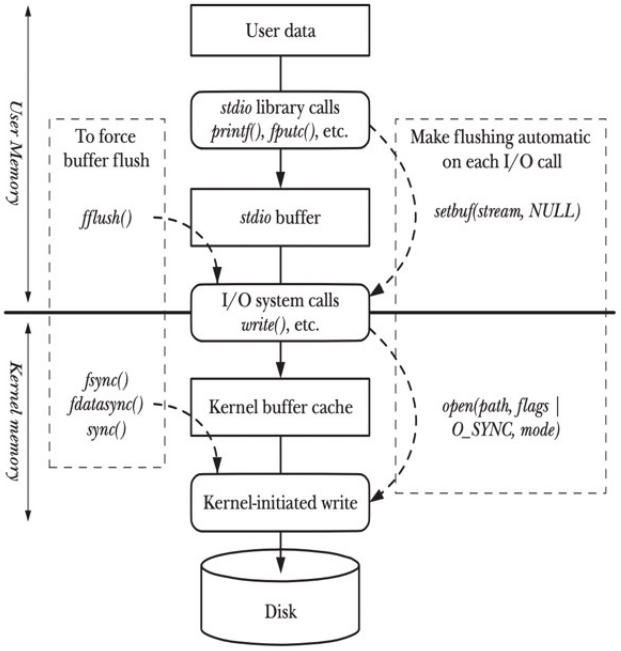

Figura 3. Schema unbuffered I/O

H. File sharing

Quando un processo accede ad un file per utilizzarlo accade che:

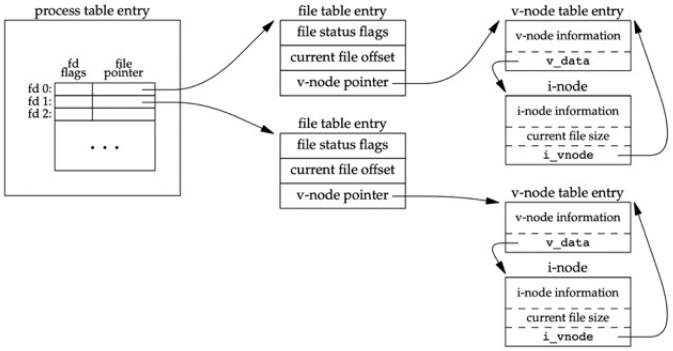

Figura 4. Unico processo accede a un file

Il processo è rappresentato dal "process table entry" con i vari file descriptor con la loro flag e puntatore alla memoria delle file table entry. Ogni file descriptor avra' una file table entry, quindi ci saranno tanti file table quanti fd ci sono, con i seguenti dati:

- **file status flags**:
- **current file offset**:
- **v-node pointer**: (con v=virtual) puntatore ad una v-node table entry che contiene i dati, cioè:
  - **v-node information**
  - **v_data**: puntatore all'inode

Potrebbe capitare che piu' processi si contengano un file tra di loro per poterci accedere.

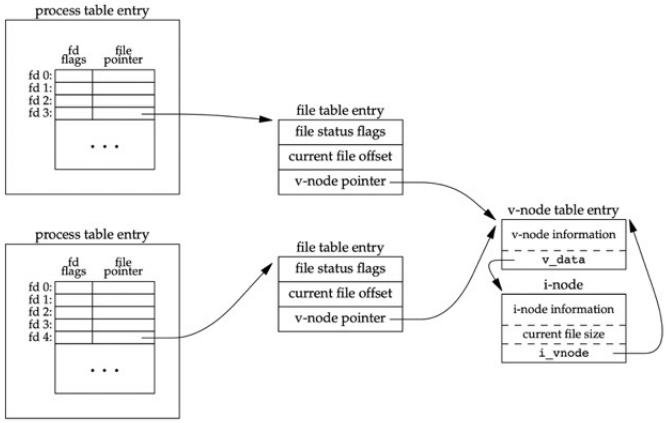

Figura 5. Più processi accedono a file

Ogni processo arriverà al file tramite 2 file table diverse ma con un "v-node prointer" allo stesso file. Dato che 2 processi stanno contemporaneamente scrivendo servirà chi gestisce il tutto altrimenti ci sarà una sovrascrittura dell'ultimo processo sugli altri.

In queste situazioni il kernel si intromette per evitare di far "intromettere" altri processi durante l'esecuzione delle system call. Questo è possibile grazie alle operazioni atomiche (tutto gestito dal programmatore del kernel). Un'esempio di operazioni atomiche è l'inserimento della flag O_APPEND.

Le chiamate per creare operazioni atomiche con più system call sono:

- pread(): è come chiamare lseek() e dopo read()
- pwrite(): come write() ma con un parametro offset che permette di effettuare un'operazione atomica scrivendo in un punto del file **senza rischio che le operazione di lseek e write si diano fastidio**. Vanno allora a fare più operazioni in una unica ma atomica

Un'altro flag da tenere sott'occhio è O_EXCL, in questo caso se il file da creare esite non viene creato. Potrebbe succedere che se il file non esiste e viene dato l'OK per crearlo ma nel frattempo viene creato da un altro processo, allora il primo andra' a sovrascrivere.

Quest'operazione viene usata per poter usare in modo esclusivo il nome di un file da un processo in modo da non farlo utilizzare da un altro mentre lui esegue le sue operazioni su altri file (il file provvisorio verrà poi eliminato).

I. dup & dup2

Vediamo le seguenti funzioni:

- dup: **prende un fd e restituisce un duplicato del fd**. Quindi **abbiamo 2 fd che puntano alla stessa file table**.

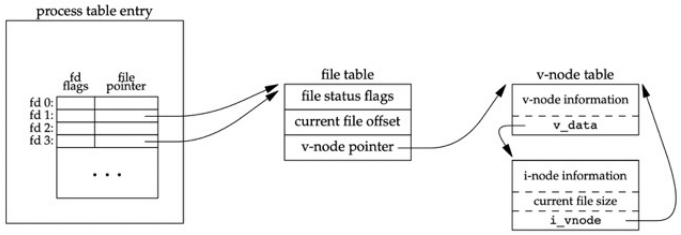

Figura 6. dup function

- dup2: prende un fd da duplicare **dicendogli anche il numero del fd**, se il numero che gli passiamo è già preso allora si **forza la chiusura** del fd e lo si assegna a ciò che vogliamo

Con le fork avremo:

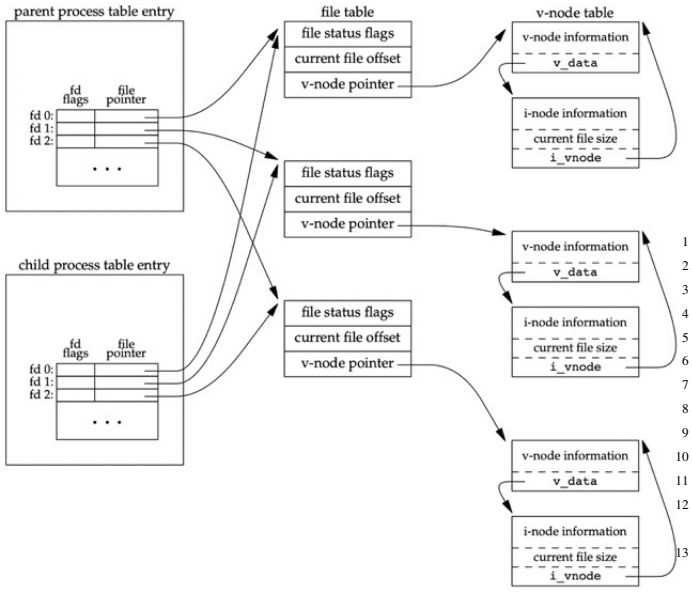

Figura 7. Fork con process table entry

dopo una fork i processi child e parent sono uno la copia dell'altro quindi avremo che i fd del child punteranno alle stesse file table e quindi agli stessi spazi di memoria alla quale accedono entrambi. Abbiamo quindi il problema che se il child modifica qualcosa lo vedrà anche il parent.

J. Ridirezione ad un file

Quindi apriamo un file ci viene dato un fd con numero minore (in genere 3), se faccio una dup2(3, 1) abbiamo la ridirezione a file. A fare tutte le operazioni e' lo shell e grazie alla eredità dei fd i processi child avranno già tutti i dati.

La gestire di queste operazioni dipende dal flag CLOEXEC. Infatti quando lo shell vede la ridirezione (>) allora:

- se sei il child esegui l'apertura del file
- ottiene fd 3
- esegui dup(3, 1)
- esegui una exec
- se il flag e' 0 (cioè non chiudere quando fai un exec) allora il child avrà in input l'output del parent
- se il flag e' 1 vogliamo che il file venga chiuso ed il fd riassegnato

K. Funzione fcntl()

È un coltellino svizzero per controlli sul fd e ha come parametri:

- fd
- cmd: usato per **chiamare delle funzionalità** specifiche tramite flags:
  - **F_DUPFD**: si **duplica il fd** e tramite il terzo argomento avremo all'**assegnazione di un fd $\geq$ del valore di quell'argomento**
  - **F_GETFD, F_SETFD**: get/set fd flag
  - **F_GETFL, F_SETFL**: get/set file status flag (per cambiare il flag **mentre il file è ancora aperto**)
  - **F_GETLK, F_SETLK**: get/set record locks: servono a **bloccare parti di file**
- "args"

Grazie al file fileflags.c possiamo vedere come è stato aperto il file:

```
#include "apue.h"
#include <fcntl.h>

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

    if (argc != 2)
        \

    if ((val = fcntl(atoi(argv[1]), F_GETFL, 0)) <
    0)
        err_sys("fcntl error for fd %d", atoi(argv
    [1]));

    switch (val & O_ACCMODE) {
    case O_RDONLY:
        printf("read only");
        break;

    case O_WRONLY:
        printf("write only");
        break;

    case O_RDWR:
        printf("read write");
        break;

    default:
        err_dump("unknown access mode");
    }

    if (val & O_APPEND)
        printf(", append");
    if (val & O_NONBLOCK)
        printf(", nonblocking");
    if (val & O_SYNC)
        printf(", synchronous writes");

#if !defined(_POSIX_C_SOURCE) && defined(O_FSYNC) &&
     (O_FSYNC != O_SYNC)
    if (val & O_FSYNC)
```

```
        printf(", synchronous writes");
#endif

    putchar('\n');
    exit(0);
}
```

infatti abbiamo:

```
$ ./fileflags 0 < /dev/ttys000
read only

$ ./fileflags 1 > prova.txt
$ cat prova.txt
write only

$ ./fileflags 2 2>>prova.txt
write only, append

$ ./fileflags 5 5<>prova.txt
read write
```

per inserire o togliere i flag usiamo:

```
val |= flags;       /* turn on flags */
val &= ~flag;       /* turn off flags */
```

VIII. File and directory

A. Chiamata stat()

Legge l'inode e ti fornisce i dati che puoi sapere riferiti a quell'inode. Tramite il comando dumpe2fs che fa il dump di tutta la struttura dati nella partizione designata.

Se vogliamo trovare i puntatori ed i blocchi di un file dato il suo nome andiamo a guardare l'inode, troviamo i blocchi che gli appartengono e poi tramite il comando dd possiamo andare tagliare i blocchi nel file che rappresenta la nostra partizione dove ci saranno da rispettare alcune regole sul quanti byte rappresentano un blocco ecc (regole del file system).

La funzione stat prende come parametri:

```
int stat(const char *restrict pathname, struct stat
    *restrict buf);
```

con:

- path: del quale dire i dati
- struct: parametri di uscita dato che è un puntatore

Le sue varianti sono:

- fstat: usa un **fd al posto del path**
- lstat: prende dei **link simbolici** in pathname (può esser usata anche per file normali)
- fstatat: prende **sia un fd che un pathname**

Nella struttura stat abbiamo tutti i dati come output dalla funzione:

```
struct stat {
    mode_t      st_mode;    /*file type & mode (
    permissions)*/
    ino_t       st_ino;     /*i-node number (serial
    number)*/
    dev_t       st_dev;     /*device number (file
    system)*/
    dev_t       st_rdev;    /*device number for
    special files*/
    nlink_t     st_nlink;   /*number of links*/
    uid_t       st_uid;     /*user ID of owner*/
    gid_t       st_gid;     /*group ID of owner*/
    off_t       st_size;    /*size in bytes, for
    regular files*/
    struct timespec st_atim;/*time of last access*/
    struct timespec st_mtim;/*time of last
    modification*/
    struct timespec st_ctim;/*time of last file
    status change*/
    blksize_t   st_blksize; /*best I/O block size*/
    blkcnt_t    st_blocks;  /*number of disk blocks
    allocated*/
};
```

dove st_mode contiene sia il tipo di file che i privilegi. Per vedere che file è ci sono delle function like macro:

Macro	Type of file
S_ISREG()	regular file
S_ISDIR()	directory file
S_ISCHR()	character special file
S_ISBLK()	block special file
S_ISFIFO()	pipe or FIFO
S_ISLNK()	symbolic link
S_ISSOCK()	socket

Tabella I
FILE TYPE MACROS IN ¡SYS/STAT.H¿

è un'esempio di check del tipo di file il file filetype.c:

```c
#include "apue.h"

int
main(int argc, char *argv[])
{
    int         i;
    struct stat buf;
    char        *ptr;

    for (i = 1; i < argc; i++) {
        printf("%s: ", argv[i]);
        if (lstat(argv[i], &buf) < 0) {
            err_ret("lstat error");
            continue;
        }
        if (S_ISREG(buf.st_mode))
            ptr = "regular";
        else if (S_ISDIR(buf.st_mode))
            ptr = "directory";
        else if (S_ISCHR(buf.st_mode))
            ptr = "character special";
        else if (S_ISBLK(buf.st_mode))
            ptr = "block special";
        else if (S_ISFIFO(buf.st_mode))
            ptr = "fifo";
        else if (S_ISLNK(buf.st_mode))
            ptr = "symbolic link";
        else if (S_ISSOCK(buf.st_mode))
            ptr = "socket";
        else
            ptr = "** unknown mode **";
        printf("%s\n", ptr);
    }
    exit(0);
}
```

B. User-ID e Group-ID

Ogni processo ha piu' ID associati a lui:

- real user/group ID: indica chi siamo
- effective user/group ID: usato quando c'è il set user id attivo e si **presenta come proprietario del file** e potrà quindi leggerlo, scriverlo ecc
- supplementary group ID: quando un utente è stato aggiunto ad un gruppo
- saved set-user/group-ID: sono delle identità associate ad un processo e fa si che **se è stato root** e lascia i privilegi **potrà riconquistarli**

I permessi saranno verificabili tramite delle macro che hanno accesso un solo bit:

st_mode mask	Meaning
S_IRUSR	user-read
S_IWUSR	user-write
S_IXUSR	user-execute
S_IRGRP	group-read
S_IWGRP	group-write
S_IXGRP	group-execute
S_IROTH	other-read
S_IWOTH	other-write
S_IXOTH	other-execute

Tabella II
THE NINE FILE ACCESS PERMISSION BITS, FROM ¡SYS/STAT.H¿

Notare che per eliminare un file esistente non ci vuole il permesso di scrittura sul file ma il permesso di srittura sulla directory dato che stai scrivendo la directory.

Grazie alle ACL (sono delle estenzioni) possiamo dare il permesso al file e dire che non può essere cancellato da un utente specifico o meno. Usiamo l'opzione:

```
ls -le file
```

Il test di accesso al file che il kernel esegue quando si lavroa conun file dipende dal proprietario del file. Il test sono:

- effective user ID = 0: ...
- effective user ID = owner ID: ... se non c'è il permesso ma lo ha il gruppo allora non potrò accederci dato che il test guarda user e poi group (in pratica non viene proprio guardato)

C. Funzioni access() e faccessat()

Dice in anticipo se si puo' o no fare una cosa. Il processo ha i suoi ID ed in base ai permessi potrà, o no, fare delle operazioni. Usiamo questa funzione:

```c
int access(const char *pathname, int mode);
```

con mode:

mode	Description
R_OK	test for read permission
W_OK	test for write permission
X_OK	test for execute permission

Tabella III
THE MODE FLAGS FOR ACCESS FUNCTION, FROM ¡UNISTD.H¿

D. Funzione umask()

Mostra in ottale i privilegi che verrano tolti al file in merito a lettura e scrittura:

```c
mode_t umask(mode_t mask);
```

viene ereditata dai processi data la loro possiblità di creare file.

Su linux la troviamo in:

```
$ cat /proc/$$/status
...
Umask:      0022
...
```

E. Sticky bit

Permesso che ha senso per le cartelle quando piu' persone ci lavorano, ma senza che un utente possa eliminare file che non gli appartengono.

```
drwxrwxrwt  15 root  wheel   480 Oct 14 22:00 Shared
```

la stessa cosa esiste per le cartelle temporanee.

F. File truncation

È possibile da file tagliare una parte di file esprimendo la lunghezza:

```c
int truncate(const char *pathname, off_t length);
int ftruncate(int fd, off_t length);
```