



Programmazione di Sistema e di Rete

Matteo Aprile
Professore: Franco Tommasi

INDICE

I	Libri di testo consigliati	2	VII	File I/O	12
II	Comandi utili	2	VII-A	Chiamata open	12
II-A	find: trovare tutti i file eseguibili	2	VII-B	Read e write flag	12
II-B	find: trovare file di intestazione del mac come stdio.h	2	VII-C	Chiamata openat()	12
II-C	lld: per capire che librerie usa il codice	2	VII-D	Open flags	12
II-D	gcc: per vedere tutta la gerarchia di file in una libreria	2	VII-E	Builtin umask	12
II-E	gcc: per vedere il codice con tutti i file importati	2	VII-F	Chiamata lseek()	12
II-F	gcc -g: debugging debole	2	VII-G	I/O efficiency	13
II-G	gcc -ggdb: debugging forte	2	VII-H	File sharing	13
II-H	xattr: usato per i file che entrano in quarantena su MacOS	3	VII-I	dup & dup2	14
II-I	apropos: per fare un a grep sulla riga name nelle pagine di manuale	3	VII-J	Ridirezione ad un file	14
III	Variabili di sistema definite in .bashrc	3	VII-K	Funzione fcntl()	15
III-A	INC	3	VIII	File and directory	16
IV	Introduzione	3	VIII-A	Chiamata stat()	16
IV-A	System call	3	VIII-B	User-ID e Group-ID	16
IV-B	Programma Make	3	VIII-C	Funzioni access() e faccessat()	17
IV-C	Direttive di preprocessore	4	VIII-D	Funzione umask()	17
IV-D	Librerie	4	VIII-E	Sticky bit	17
IV-E	Creazione librerie	5	VIII-F	File truncation	17
IV-F	Aggiornamento librerie	5	VIII-G	Hard link e link simbolici	17
V	System call	6	VIII-H	File times	17
V-A	Funzioni e system call	6	VIII-I	Reading directories	18
V-B	Capire se una funzione è una system call	7	VIII-J	Funzione chdir()	20
V-C	Numeri dei file descriptor	7	IX	System Data Files and Information	20
V-D	Meccanismi dei file	7	IX-A	Password file	20
V-E	Unbuffered I/O	7	IX-B	Time and Date Routines	20
V-F	Fork & Exec	8			
V-G	Thread	9			
V-H	Gestione degli errori	9			
V-I	Keyword in C ed accessi a variabili	9			
V-J	Segnal e interrupt	10			
V-K	Valori del tempo	10			
VI	Gli standard	11			
VI-A	Storia e basi	11			
VI-B	Limiti	11			
VI-C	Determinare l'allocazione	11			

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
- Gail Guida alla Programmazione in Linux, Simone Piccardi

II. COMANDI UTILI

A. *find: trovare tutti i file eseguibili*

```
1 $ find . -type f -perm -0100
2 ./standards/makeopt.awk
3 ./standards/makeconf.awk
4 ./proc/awkexample
5 ./systype.sh
6 ./advio/fixup.awk
```

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

```
1 $ find /Applications/Xcode.app/ -name stdio.h 2>/dev
   /null
```

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

```
1 $ ldd [nomecodice]
```

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

```
1 $ gcc -H lib.a
```

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

```
1 $ gcc -E file.c
```

F. *gcc -g: debugging debole*

```
1 $ gcc -g -ansi -I../include -Wall -DMACOS -
   D_DARWIN_C_SOURCE ls1.c -o ls1 -L../lib -lapue
```

G. *gcc -ggbd: debugging forte*

```
1 $ gcc -ggbd -ansi -I../include -Wall -DMACOS -
   D_DARWIN_C_SOURCE ls1.c -o ls1 -L../lib -lapue
```

con opzioni (ggbd per Linux e lldb per MacOS):

- s: per fare stepping
- n: per fare stepping over delle funzioni
- c: per continuare con il run
- b [# linea]: per impostare un breakpoint
- p [nome variabile]: per stampare le variabili possiamo avere la possibilità di vedere la memoria di una variabile:

```
1 (lldb) p &fd1
```

notare che andando a guardarle possiamo capire dove si trova lo stack.

Per capire fin dove arriva la "punta" dello stack possiamo usare:

```
1 (lldb) register read [nome registro]
```

- bt: per vedere lo stato dello stack
- l: (list) per vedere il codice sorgente
- r [args]: (run) per dare gli argomenti e runnare

per vedere il numero dell'indirizzo come decimale e quindi poi KB, MB, GB o TB...:

```
1 $ echo $((0x000000016fdffale))
2 6171916830
```

per vedere il contenuto in char (c), decimale (d), esadecimale (x) di un indirizzo di memoria:

```
1 (lldb) memory read --size 1 -l4 --format c --count 10 0x000000016fdffa1e
2 0x16fdffa1e: dd2.
3 0x16fdffa22: c\0ch
4 0x16fdffa26: un
```

Per poter leggere tutti i registri. Per leggere la "punta" usiamo come indirizzo quello dello stack pointer \$sp:

```
1 (lldb) memory read --size 4 --format x --count 160 $sp
```

Per i **registri** abbiamo:

- **rax**: registro dove c'è il numero della system call chiamata
- **rsp**: stack frame pointer. Salvo una volta che lo stack viene rilasciato
- **rbx, rcx, rdx, rdi, rsi**: parametri di funzioni e systemcall
- **rbp**: base dello stack frame. Salvo una volta che lo stack viene rilasciato
- **rip**: pointer all'istruzione successiva. Avrà un valore basso se punta nel text, viceversa se punta ad una libreria (lldb) disassemble: per far vedere il codice eseguibile in assembly delle funzione che sto eseguendo, dove indica con "->" la prossima istruzione che sta per essere eseguita.

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

```
1 $ xattr -d (delete) com.apple.quarantine [path sh]
```

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

```
1 $ apropos acl
```

III. VARIABILI DI SISTEMA DEFINITE IN .BASHRC

A. INC

```
1 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, però 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()**:

```
1 mov edx,4 ; lunghezza messaggio
2 mov ecx,msg ; puntatore al messaggio
3 mov ebx,1 ; file descriptor
4 mov eax,4 ; numero della sc
5 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 **referendosi ad esso tramite un numero, cioè il più 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 dà 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**:

```
1 DIRS = lib intro sockets advio daemons datafiles db
   environ \
2   fileio filedir ipc1 ipc2 proc pty relation
   signals standards \
3   stdio termios threadctl threads printer
   exercises
4
5 all:
6   for i in $(DIRS); do \
7     (cd $$i && echo "making $$i" && $(MAKE) ) ||
   exit 1; \
8   done
9
10 clean:
11   for i in $(DIRS); do \
12     (cd $$i && echo "cleaning $$i" && $(MAKE)
   clean) || exit 1; \
13   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:

```
1 target: prerequisiti
2 rule
```

dove:

- **target**: è la 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 può 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

```
1 file: file.o lib.o
2
3 clean:
4 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:

```
1 ROOT=..
2 PLATFORM=$(shell $(ROOT)/systype.sh)
3 include $(ROOT)/Make.defines.$(PLATFORM)
4
5 PROGS = getcputc hello ls1 mycat shell1 shell2
6         testerror uidgid
7
8 all:      $(PROGS)
9
10 %:  %.c $(LIBAPUE)
11     $(CC) $(CFLAGS) $@.c -o $@ $(LDFLAGS) $(LDLIBS)
12
13 clean:
14     rm -f $(PROGS) $(TEMPFILES) *.o
15
16 include $(ROOT)/Make.libapue.inc
```

dove:

- ROOT: cwd
- PLATFORM: assumerà 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 saprà come compilarlo grazie a tutte le definizioni di default presenti in:

```
1 $ make -p
```

notare che si può mettere un comando custom nelle rule del target

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

```
1 $ 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 BUFLLEN), 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

```
1 #ifdef VAR
2 print("hello");
3 #endif
```

Per evitare che più 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)

```
1 $ 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.

```
1 $ 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 corretto 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:

```
1 $ gcc -static
```

E. Creazione librerie

Per costruire una **libreria statica per MacOS**:

- 1) costruiamo il **file oggetto**:

```
1 $ gcc -c libprova.c
```

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

```
1 $ ar rcs libprova.a libprova.o
```

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

```
1 $ gcc -Wall -g -c useprova.c
```

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

```
1 $ gcc -g -o useprova useprova.o -L. -lprova
```

Per capire che libreria usa il codice si usa:

```
1 $ otool -L [nomecodice]
```

Per costruire una **libreria statica per Linux**:

- 1) costruiamo il **file oggetto**:

```
1 $ gcc -fPIC -Wall -g -c libprova.c
```

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

```
1 $ 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:

```
1 $ ln -sf libprova.so.0.0 libprova.so.0
```

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

```
1 $ ln -sf libprova.so.0 libprova.so
```

Per capire che libreria usa il codice si usa:

```
1 $ ldd [nomecodice]
```

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**:

```
1 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**:

```
1 $ 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:

```
1 $ 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:

```
1 #include "apue.h"
2 #include <dirent.h>
3
4 int
5 main(int argc, char *argv[])
6 {
7     DIR                *dp;
8     struct dirent      *dirp;
9
10    if (argc != 2)
11        err_quit("usage: ls1 directory_name");
12
13    if ((dp = opendir(argv[1])) == NULL)
14        err_sys("can't open %s", argv[1]);
15    while ((dirp = readdir(dp)) != NULL)
16        printf("%s\n", dirp->d_name);
17
18    closedir(dp);
19    exit(0);
20 }
```

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:

```
1 $ grep -rw "struct dirent" $INC
2
3 /Applications/Xcode.app/Contents/Developer/
  Platforms/MacOSX.platform/Developer/SDKs/
  MacOSX.sdk/usr/include//sys/dirent.h:struct
  dirent {
```

```
1 #ifndef _SYS_DIRENT_H
2 #define _SYS_DIRENT_H
3
4 #include <sys/_types.h>
5 #include <sys/cdefs.h>
6
7 #include <sys/_types/_ino_t.h>
8
9
10 #define __DARWIN_MAXNAMLEN      255
11
12 #pragma pack(4)
13
14 #if !__DARWIN_64_BIT_INO_T
15 struct dirent {
16     ino_t d_ino;                /* file
17     number of entry */
18     __uint16_t d_reclen;        /* length of
19     this record */
20     __uint8_t d_type;           /* file type
21     , see below */
22     __uint8_t d_namlen;        /* length of
23     string in d_name */
24     char d_name[__DARWIN_MAXNAMLEN + 1]; /*
25     name must be no longer than this */
26 };
27 #endif /* !__DARWIN_64_BIT_INO_T */
28
29 #pragma pack()
30
31 #define __DARWIN_MAXPATHLEN      1024
32
33 #define __DARWIN_STRUCT_DIRENTRY { \
```

```
29     __uint64_t d_ino;          /* file number of
30     entry */ \
31     __uint64_t d_seekoff;      /* seek offset (
32     optional, used by servers) */ \
33     __uint16_t d_reclen;       /* length of this
34     record */ \
35     __uint16_t d_namlen;       /* length of string
36     in d_name */ \
37     __uint8_t d_type;          /* file type, see
38     below */ \
39     char d_name[__DARWIN_MAXPATHLEN]; /*
40     entry name (up to MAXPATHLEN bytes) */ \
41 }
42
43 #if __DARWIN_64_BIT_INO_T
44 struct dirent __DARWIN_STRUCT_DIRENTRY;
45 #endif /* __DARWIN_64_BIT_INO_T */
46
47
48 #if !defined(_POSIX_C_SOURCE) || defined(
49     _DARWIN_C_SOURCE)
50 #define d_fileno      d_ino          /*
51     backward compatibility */
52 #define MAXNAMLEN     __DARWIN_MAXNAMLEN
53 /*
54  * File types
55  */
56 #define DT_UNKNOWN      0
57 #define DT_FIFO         1
58 #define DT_CHR          2
59 #define DT_DIR          4
60 #define DT_BLK          6
61 #define DT_REG           8
62 #define DT_LNK          10
63 #define DT_SOCK          12
64 #define DT_WHT          14
65
66 /*
67  * Convert between stat structure types and
68  * directory types.
69  */
70 #define IFTODT(mode)    (((mode) & 0170000) >>
71     12)
72 #define DTTOIF(dirtype) ((dirtype) << 12)
73 #endif
74
75 #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:

```
1 #define __DARWIN_STRUCT_DIRENTRY { \
2     __uint64_t d_ino;          /* file number of
3     entry */ \
4     __uint64_t d_seekoff;      /* seek offset (
5     optional, used by servers) */ \
6     __uint16_t d_reclen;       /* length of this
7     record */ \
8     __uint16_t d_namlen;       /* length of string
9     in d_name */ \
10    __uint8_t d_type;           /* file type, see
11    below */ \
12    char d_name[__DARWIN_MAXPATHLEN]; /*
13    entry name (up to MAXPATHLEN bytes) */ \
14 }
```

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


```
1 $ grep -rw "err_quit" $INC
```

infatti non restituisce nulla. Deve allora essere una funzione di libreria creata 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:

```
1 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:

```
1 $ 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":

```
1 #include "apue.h"
2
3 #define BUFSIZE 4096
4
5 int
6 main(void)
7 {
8     int    n;
9     char   buf[BUFSIZE];
10
11     while ((n = read(STDIN_FILENO, buf, BUFSIZE)) >
12            0)
13         if (write(STDOUT_FILENO, buf, n) != n)
14             err_sys("write error");
15
16     if (n < 0)
17         err_sys("read error");
18
19     exit(0);
20 }
```

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

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

– **buf**: indirizzo dell'inizio dell'array

– **BUFSIZE**: 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 ricominciare 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**:

```
1 $ grep -rw "STDIN_FILENO" $INC
2
3 /Applications/Xcode.app/Contents/Developer/Platforms
4 /MacOSX.platform/Developer/SDKs/MacOSX.sdk/usr/
5 include/unistd.h:#define STDIN_FILENO 0
6 /* standard input file descriptor */
7
8 /Applications/Xcode.app/Contents/Developer/Platforms
9 /MacOSX.platform/Developer/SDKs/MacOSX.sdk/usr/
10 include/asl.h: * asl_log_descriptor(c, m,
11 ASL_LEVEL_NOTICE, STDIN_FILENO,
12 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 basa UNIX 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 modalità 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.

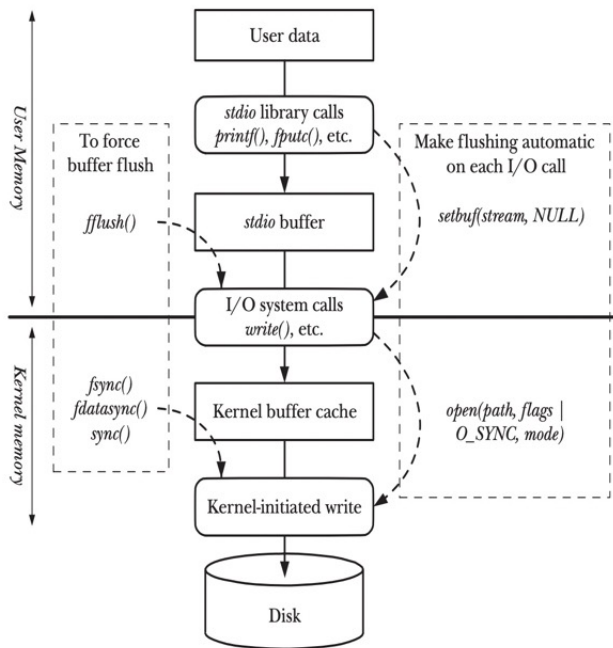


Figura 1. Schema unbuffered I/O

F. Fork & Exec

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

```
1 #include "apue.h"
2 #include <sys/wait.h>
3
4 int
5 main(void)
6 {
7     char    buf[MAXLINE];    /* from apue.h */
8     pid_t   pid;
9     int     status;
10
11     printf("%s "); /* print prompt (printf requires
12                    %s to print %) */
13     while (fgets(buf, MAXLINE, stdin) != NULL) {
14         if (buf[strlen(buf) - 1] == '\n')
15             buf[strlen(buf) - 1] = 0; /* replace
16             newline with null */
17
18         if ((pid = fork()) < 0) {
19             err_sys("fork error");
20         } else if (pid == 0) { /* child */
21             execlp(buf, buf, (char *)0);
22             err_ret("couldn't execute: %s", buf);
23             exit(127);
24         }
25
26         /* parent */
27         if ((pid = waitpid(pid, &status, 0)) < 0)
28             err_sys("waitpid error");
29         printf("%s ");
30     }
31     exit(0);
32 }
```

avremo allora:

- **fgets**: funzione dello **stdout** 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

```
1 $ grep -rw "MAXLINE" include/
2 Binary file include//apue.h.gch matches
3 include//apue.h:#define MAXLINE 4096
   /* max line length */
```

- **stdin**: presente in stdio 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 getpid().

Se pid < 0 vorrà dire che la fork è fallita. Se pid = 0 vorrà dire che siamo nel child.

Il che è molto importante dato che **il codice verrà eseguito sia dal child che dal parent**, e sarà contenuto nella memoria 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 avanti 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

Processi

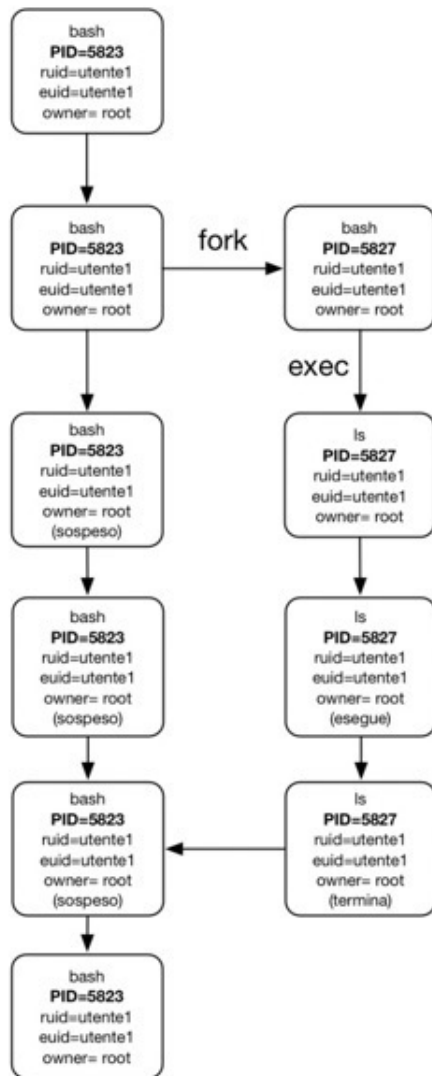


Figura 2. Esecuzione di fork ed exec

G. Thread

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

Tutto questo è **orchestrato dal kernel** che gestisce il **time sharing**.

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:

```

1 $ man 2 intro
2 ...
3 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.
4
5 2 ENOENT No such file or directory. A component of a
  specified pathname did not exist, or the
  pathname was an empty string.
6 ...
  
```

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

```
1 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

```

1 #include "apue.h"
2 #include <errno.h>
3
4 int
5 main(int argc, char *argv[])
6 {
7     fprintf(stderr, "EACCES: %s\n", strerror(EACCES)
8 );
9     errno = ENOENT;
10    perror(argv[0]);
11    exit(0);
  
```

dove:

- **fprintf**: 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**

```
1 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.

```
1 int *restrict x
```

J. Segnal e interrupt

Guardiamo il file shell2.c :

```
1 #include "apue.h"
2 #include <sys/wait.h>
3
4 static void sig_int(int);      /* our signal-
   catching function */
5
6 int
7 main(void)
8 {
9     char    buf[MAXLINE];      /* from apue.h */
10    pid_t    pid;
11    int      status;
12
13    if (signal(SIGINT, sig_int) == SIG_ERR)
14        err_sys("signal error");
15
16    printf("%% "); /* print prompt (printf requires
   %% to print %) */
17    while (fgets(buf, MAXLINE, stdin) != NULL) {
18        if (buf[strlen(buf) - 1] == '\n')
19            buf[strlen(buf) - 1] = 0; /* replace
   newline with null */
20
21        if ((pid = fork()) < 0) {
22            err_sys("fork error");
23        } else if (pid == 0) { /* child */
24            execlp(buf, buf, (char *)0);
25            err_ret("couldn't execute: %s", buf);
26            exit(127);
27        }
28
29        /* parent */
30        if ((pid = waitpid(pid, &status, 0)) < 0)
31            err_sys("waitpid error");
32        printf("%% ");
33    }
34    exit(0);
35 }
36
37 void
38 sig_int(int signo)
39 {
40     printf("interrupt\n%% ");
41 }
```

dove rispetto a shell1 abbiamo 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 processo
- 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 maggiore del clock time se abbiamo un processore multicore dato che somma il tempo dai diversi core (il tempo potrebbe essere diverso in base al core).

```
1 $ time [nome programma]
2
3 ...
4 real    0m0.006s
5 user    0m0.001s
6 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 interfacce 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.

```
1 -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**:

```
1 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 sì di avere più **portabilità**. 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:

```
1 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**:

```
1 $ grep -rw "pid_t" $INC | grep typedef
2
3 /sys/_types/_pid_t.h:typedef __darwin_pid_t pid_t;
4 /sys/_types.h:typedef __uint32_t __darwin_id_t;
5
6 $ grep -rw "__uint32_t" $INC | grep typedef
7
8 /i386/_types.h:typedef unsigned int __uint32_t;
```

tutti questi rimandi sono dati dalla **portabilità**.

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 modalità**. 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)**

```
1 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":

```
1 #define O_RDONLY    0x0000 /* reading only */  
2 #define O_WRONLY    0x0001 /* writing only */  
3 #define O_RDWR     0x0002 /* reading and writing */  
4 #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):

```
1 $ umask  
2 0022  
3  
4 $ ll file  
5 -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 **potrà 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 **bucio 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:

```
1 #include "apue.h"  
2 #include <fcntl.h>  
3  
4 char    buf1[] = "abcdefghg";  
5 char    buf2[] = "ABCDEFGHGH";  
6  
7 int  
8 main(void)  
9 {  
10     int    fd;  
11  
12     if ((fd = creat("file.hole", FILE_MODE)) < 0)  
13         err_sys("creat error");  
14  
15     if (write(fd, buf1, 10) != 10)  
16         err_sys("buf1 write error");  
17     /* offset now = 10 */  
18  
19     if (lseek(fd, 16384, SEEK_SET) == -1)  
20         err_sys("lseek error");
```

```

21  /* offset now = 16384 */
22
23  if (write(fd, buf2, 10) != 10)
24      err_sys("buf2 write error");
25  /* offset now = 16394 */
26
27  exit(0);
28 }

```

avremo:

```

1 $ xxd file.hole
2 00000000: 6162 6364 6566 0000  abcdefgh.....
3 00000010: 0000 0000 0000 0000  .....
4 00000020: 0000 0000 0000 0000  .....
5 00000030: 0000 0000 0000 0000  .....
6 00000040: 0000 0000 0000 0000  .....
7 00000050: 0000 0000 0000 0000  .....
8 00000060: 0000 0000 0000 0000  .....
9 00000070: 4142 4344 4546 4748  ABCDEFGH.....

```

abbiamo che la memoria sul disco è:

```

1 $ du -h file.hole
2 1.6M    file.hole

```

invece la size del file è:

```

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

```

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

```

1 #include "apue.h"
2
3 int
4 main(void)
5 {
6     if (lseek(STDIN_FILENO, 0, SEEK_CUR) == -1)
7         printf("cannot seek\n");
8     else
9         printf("seek OK\n");
10    exit(0);
11 }

```

G. I/O efficiency

Se devo **trasferire una grande quantità 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))

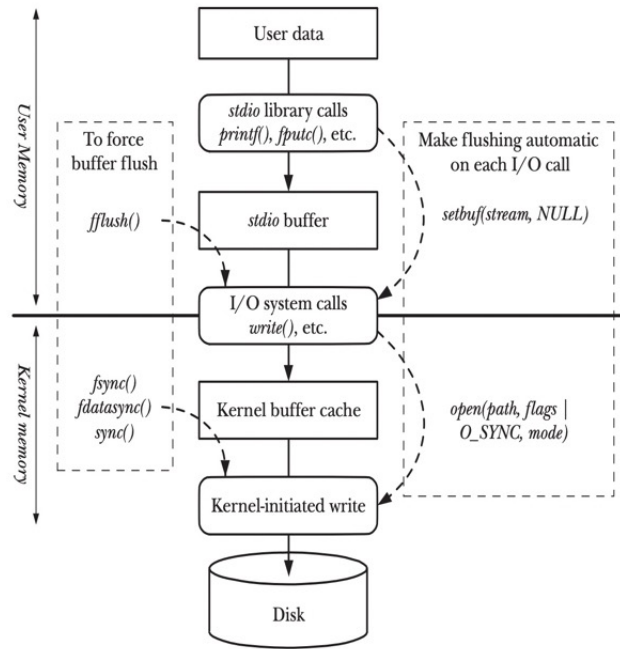


Figura 3. Schema unbuffered I/O

H. File sharing

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

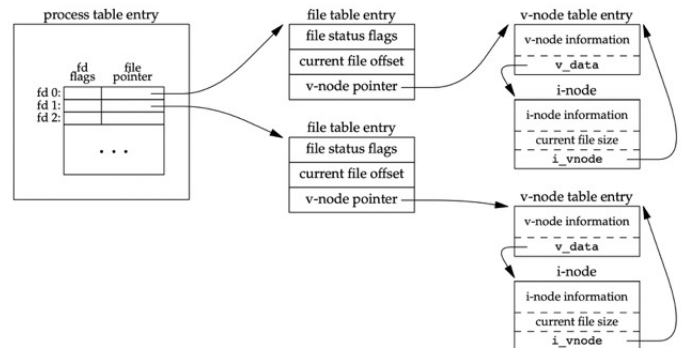


Figura 4. Unico processo accede a un file

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** avrà 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'i-node

Potrebbe capitare che **più processi si contengano un file** tra di loro per poterci accedere.

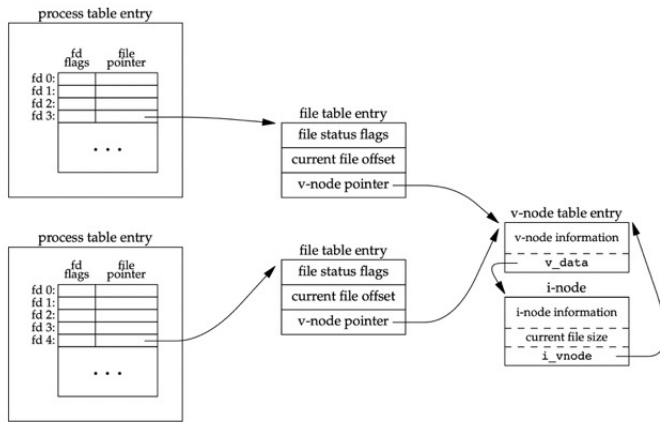


Figura 5. Più processi accedono a file

Ogni processo arriverà al file tramite 2 file table diverse ma con un "v-node pointer" 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 operazioni 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 esiste 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 andrà 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.

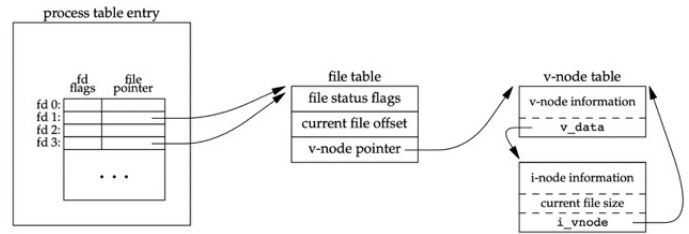


Figura 6. dup function

- `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:

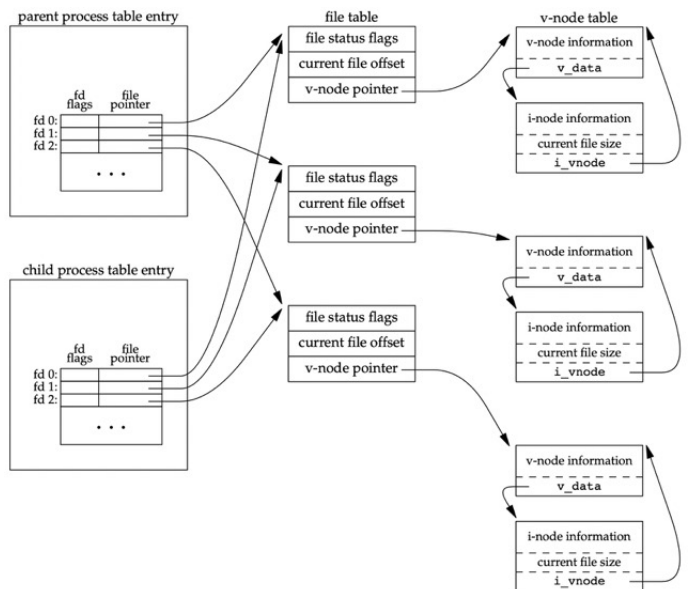


Figura 7. Fork con process table entry

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 gestione 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

```
41     printf(", synchronous writes");
42 #endif
43
44     putchar('\n');
45     exit(0);
46 }
```

infatti abbiamo:

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 ≥ 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:

```
1 #include "apue.h"
2 #include <fcntl.h>
3
4 int
5 main(int argc, char *argv[])
6 {
7     int    val;
8
9     if (argc != 2)
10         \
11
12     if ((val = fcntl(atoi(argv[1]), F_GETFL, 0)) <
13         0)
14         err_sys("fcntl error for fd %d", atoi(argv
15             [1]));
16
17     switch (val & O_ACCMODE) {
18     case O_RDONLY:
19         printf("read only");
20         break;
21     case O_WRONLY:
22         printf("write only");
23         break;
24     case O_RDWR:
25         printf("read write");
26         break;
27     default:
28         err_dump("unknown access mode");
29     }
30
31     if (val & O_APPEND)
32         printf(", append");
33     if (val & O_NONBLOCK)
34         printf(", nonblocking");
35     if (val & O_SYNC)
36         printf(", synchronous writes");
37
38 #if !defined(_POSIX_C_SOURCE) && defined(O_FSYNC) &&
39     (O_FSYNC != O_SYNC)
40     if (val & O_FSYNC)
```

```
1 $ ./fileflags 0 < /dev/tty000
2 read only
3
4 $ ./fileflags 1 > prova.txt
5 $ cat prova.txt
6 write only
7
8 $ ./fileflags 2 2>>prova.txt
9 write only, append
10
11 $ ./fileflags 5 5<>prova.txt
12 read write
```

per inserire o togliere i flag usiamo:

```
1 val |= flags;      /* turn on flags */
2 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**:

```
1 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 essere usata anche per file normali)
- **fstatat**: prende **sia un fd che un pathname**

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

```
1 struct stat {  
2     mode_t      st_mode;      /*file type & mode ( permissions)*/  
3     ino_t        st_ino;       /*i-node number (serial number)*/  
4     dev_t        st_dev;       /*device number (file system)*/  
5     dev_t        st_rdev;      /*device number for special files*/  
6     nlink_t      st_nlink;     /*number of links*/  
7     uid_t        st_uid;       /*user ID of owner*/  
8     gid_t        st_gid;       /*group ID of owner*/  
9     off_t        st_size;      /*size in bytes, for regular files*/  
10    struct timespec st_atim; /*time of last access*/  
11    struct timespec st_mtim; /*time of last modification*/  
12    struct timespec st_ctim; /*time of last file status change*/  
13    blksize_t      st_blksize; /*best I/O block size*/  
14    blkcnt_t       st_blocks; /*number of disk blocks allocated*/  
15 };
```

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 `/usr/include/sys/stat.h`

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

```
1 #include "apue.h"  
2  
3 int  
4 main(int argc, char *argv[])  
5 {  
6     int i;  
7     struct stat buf;  
8     char *ptr;  
9  
10    for (i = 1; i < argc; i++) {  
11        printf("%s: ", argv[i]);  
12        if (lstat(argv[i], &buf) < 0) {  
13            err_ret("lstat error");  
14            continue;  
15        }  
16        if (S_ISREG(buf.st_mode))  
17            ptr = "regular";  
18        else if (S_ISDIR(buf.st_mode))  
19            ptr = "directory";  
20        else if (S_ISCHR(buf.st_mode))  
21            ptr = "character special";  
22        else if (S_ISBLK(buf.st_mode))  
23            ptr = "block special";  
24        else if (S_ISFIFO(buf.st_mode))  
25            ptr = "fifo";  
26        else if (S_ISLNK(buf.st_mode))  
27            ptr = "symbolic link";  
28        else if (S_ISSOCK(buf.st_mode))  
29            ptr = "socket";  
30        else  
31            ptr = "** unknown mode **";  
32        printf("%s\n", ptr);  
33    }  
34    exit(0);  
35 }
```

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 sì che **se è stato root** e lascia i privilegi **potrà riconquistarli**

I permessi saranno verificabili tramite delle macro che hanno accesso a 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 `/usr/include/sys/stat.h`

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

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

```
1 ls -le file
```

Il test di accesso al file che il kernel esegue quando si lavora con un 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 può 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:

```
1 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 `/usr/include/unistd.h`

D. Funzione umask()

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

```
1 mode_t umask(mode_t mask);
```

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

Su linux la troviamo in:

```
1 $ cat /proc/$$/status
2 ...
3 Umask:      0022
4 ...
```

E. Sticky bit

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

```
1 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:

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

G. Hard link e link simbolici

La funzione per crearli in C è:

```
1 int link(const char *existingpath, const char *
    newpath);
2 int linkat(int efd, const char *existingpath, int
    nfd, const char *newpath, int flag);
```

Abbiamo delle funzioni che seguiranno i link simbolici ed altre che seguiranno direttamente il file:

Function	Does not follow symbolic link	Follows symbolic link
access		•
chdir		•
chmod		•
chown		•
creat		•
exec		•
lchown	•	
link		•
lstat	•	
open		•
opendir		•
pathconf		•
readlink	•	
remove	•	
rename	•	
stat		•
truncate		•
unlink	•	

Tabella IV

TABELLA DELLE FUNZIONI

Tramite i link simbolici è possibile creare dei loop nei quali l'OS si può bloccarsi.

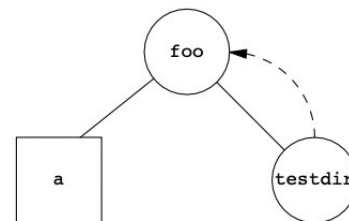


Figura 8. Loop generato da un link simbolico

Per i link simbolici usiamo le funzioni:

- **symlink()**: crea un link simbolico

```
1 int symlink(const char *actualpath, const char *
    sympath);
```

- **readlink()**: combina le azioni di apertura, lettura e chiusura

```
1 ssize_t readlink(const char* restrict pathname,
    char *restrict buf, size_t bufsiz);
```

H. File times

Abbiamo 3 tipi di file times:

Esistono delle funzioni per poter modificare i tempi:

- **futimens()**: dove "f" sta per fd, invece "u" per nanosecondi

```
1 int futimes(int fd, const struct timespec times
    [2]);
```

Field	Description	Example
st_atim	last-access time of file data	read
st_mtim	last-modification time of file data	write
st_ctim	last-change time of i-node status	chmod, chown

Tabella V

THE THREE TIME VALUES ASSOCIATED WITH EACH FILE

- **fd**: file descriptor, quindi cambi il tempo ad un file già aperto
- **times[2]**: array con tempo di accesso e di modifica che a loro volta sono composti da più campi dato che sono di tipo timespec

- **utimensat**: uguale a futimens ma usa i **microsecondi**, che sono meno affidabili detti da un compilatore dato che la granularità promessa è minore di quella possibile

```
1 int utimensat(int fd, const char *pathname,  
    const struct timespec times[2], int flag);
```

- **utimes**

o più semplicemente abbiamo "touch" che ci permette di modificare il tempo di accesso:

```
1 $ stat -x prova  
2 ...  
3 Access: Fri Oct 21 16:54:30 2022  
4 Modify: Fri Oct 21 16:54:30 2022  
5 Change: Fri Oct 21 16:54:30 2022  
6 Birth: Fri Oct 21 16:54:30 2022  
7  
8 $ touch prova  
9 $ stat -x prova  
10 ...  
11 Device: 1,18 Inode: 28513966 Links: 1  
12 Access: Fri Oct 21 16:55:06 2022  
13 Modify: Fri Oct 21 16:55:06 2022  
14 Change: Fri Oct 21 16:55:06 2022  
15 Birth: Fri Oct 21 16:54:30 2022
```

Per capire come funziona possiamo analizzare il file zap.c:

```
1 #include "apue.h"  
2 #include <fcntl.h>  
3  
4 int  
5 main(int argc, char *argv[])  
6 {  
7     int i, fd;  
8     struct stat statbuf;  
9     struct timespec times[2];  
10  
11     for (i = 1; i < argc; i++) {  
12         if (stat(argv[i], &statbuf) < 0) { /* fetch  
            current times */  
13             err_ret("%s: stat error", argv[i]);  
14             continue;  
15         }  
16         if ((fd = open(argv[i], O_RDWR | O_TRUNC)) <  
17             0) { /* truncate */  
18             err_ret("%s: open error", argv[i]);  
19             continue;  
20         }  
21         times[0] = statbuf.st_atim; /* per MacOS  
            usare st_atimeprec */  
22         times[1] = statbuf.st_mtim; /* per MacOS  
            usare st_mtimeprec */  
23         if (futimens(fd, times) < 0) /* reset  
            times */  
24             err_ret("%s: futimens error", argv[i]);  
25         close(fd);  
26     }  
27 }
```

```
26     exit(0);  
27 }
```

dove:

- **if 1**: leggo il file e lo metto nella **struttura nel buffer**
- **if 2**: effettuo una qualsiasi modifica, in questo caso un troncamento
- **times[...]**: carico i vecchi tempi
- **if 3**: resetta i tempi del file a quelli iniziali

notiamo però che la modifica non può intaccare anche il tempo di change non viene mai resettato a quello originario dato che **non può essere toccato**:

```
1 $ touch prova  
2 $ stat -x prova  
3 ...  
4 Access: Fri Oct 21 17:11:28 2022  
5 Modify: Fri Oct 21 17:11:28 2022  
6 Change: Fri Oct 21 17:11:29 2022  
7 Birth: Fri Oct 21 17:11:28 2022  
8  
9 $ ./zap prova  
10 $ stat -x prova  
11 ...  
12 Access: Fri Oct 21 17:11:28 2022  
13 Modify: Fri Oct 21 17:11:28 2022  
14 Change: Fri Oct 21 17:12:07 2022  
15 Birth: Fri Oct 21 17:11:28 2022
```

I. Reading directories

Abbiamo vari modi per leggere delle directory:

- **opendir**:

```
1 DIR *opendir(const char *pathname);
```

- **fdopendir**: per aprire una directory tramite fd

```
1 DIR *fdopendir(int fd);
```

- **readdir**: si posiziona nel leggere il file successivo della directory

```
1 struct dirent *readdir(DIR *dp);
```

- **rewinddir**: riposiziona il cursore all'inizio

```
1 void rewinddir(DIR *dp);
```

- **closedir**:

```
1 int closedir(DIR *dp);
```

- **telldir**: mi dice dove mi trovo nella directory, il long lo usa nella seekdir()

```
1 long telldir(DIR *dp);
```

- **seekdir**: ti rimette dove volevi tornare tramite il long della telldir

```
1 void seekdir(DIR *dp, long loc);
```

Non abbiamo una writedir() dato che ci sono **più modi per scrivere in una directory**:

- **link()**: creare un link
- **open()** tramite il flag **O_CREAT**
- **makedir()**

Un esempio di accesso a file è il programma ftw8.c al quale passiamo un path e ne **visita tutto l'albero**, è anche possibile

passare e fargli **eseguire un comando** ad ogni diramazione dell'albero.

Riguardo l'accesso a un indirizzo di memoria, sappiamo che un qualsiasi processo per poter allocare memoria deve **chiedere il permesso** al kernel. Sappiamo che 2 processi possono **condividere la stessa memoria** dato che gli indirizzi reali sono mappati sullo stesso byte ma nella memoria virtuale di un processo non è così.

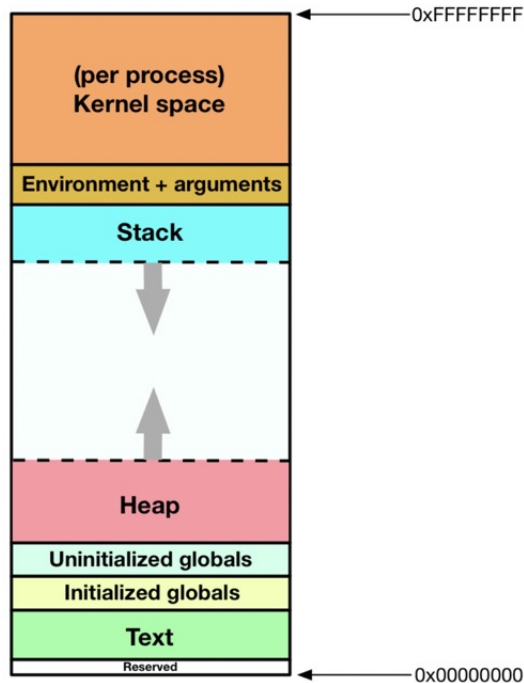


Figura 9. Memoria virtuale di un processo

Questa memoria (in questo per sistemi a 32bit) è suddivisa in:

- **Reserved**: parte riservata da 0 a n GB
- **Text**: **codice eseguibile** del programma, utile al processore
- **Initialized globals**: variabili dichiarate globali ed iniziate
- **Uninitialized globals**:
- **Heap**: parte alla quale ci si accede tramite delle richieste, per esempio con la malloc. Notare che la maggior parte di queste zone di memoria sono preallocate. Nell'heap ci finiscono anche le librerie dinamiche, dato che è codice di sola lettura. Queste sono allocate in modo randomico per rendere complicata la loro ricerca
- **...**: spazio virtuale super mega ampio
- **Kernel space**: zona vietata al programmatore dove il kernel fa le sue cose
- **Environment + arguments**:
 - **arguments**: stringhe passate al programma che le interpreterà
 - **environment**: trovo i caratteri ASCII delle variabili di ambiente
- **Stack**: usato per **lavorare con le funzioni**, infatti quando una **funzione viene invocata mette in uno stack-**

frame le variabili locali (o automatiche) allocate al momento della chiamata della funzione. Questa funzione potrebbe chiamarne un'altra, ecc ecc, allora lo **stack cresce verso il basso**. Ogni funzione può **accedere alle proprie variabili e a quelle globali**. È stata scelta questa organizzazione perché **non si può prevedere a priori quanto spazio servirà** perché potrebbe essere richiamata una funzione in modo ricorsivo, allora si usa lo **swapping** e si fa di **non allocare nella RAM tutta la memoria del processo** ma solo quella necessaria. Il resto dei dati verranno pescati dalla memoria di massa tramite uno swap-in.

Il massimo di dimensioni dello stack le vediamo da:

```
1 $ ulimit -a
2
3 core file size          (blocks, -c) 0
4 data seg size           (kbytes, -d) unlimited
5 file size               (blocks, -f) unlimited
6 max locked memory       (kbytes, -l) unlimited
7 max memory size         (kbytes, -m) unlimited
8 open files              (-n) 256
9 pipe size               (512 bytes, -p) 1
10 stack size              (kbytes, -s) 8192
11 cpu time                (seconds, -t) unlimited
12 max user processes      (-u) 709
13 virtual memory          (kbytes, -v) unlimited
```

di default sono a 8MB.

Per quanto riguarda il codice di ftw8.c (File Three Walk) sarà importante da padroneggiare per poter **visitare tutti i nodi** a partire da una root decidendo, per ogni nodo, quale programma eseguire tramite una variabile **Myfunc** creata da un typedef:

```
1 typedef int    Myfunc(const char *, const struct
                    stat *, int);
```

dove il tipo può assumere più valori da usare come **coltellino svizzero**:

- **FTW_F**: ha uno switch annidato dove prende, da st_mode, un campo della struttura attraverso un puntatore "->". Viene poi messo in "and" con una maschera per poter prendere il tipo di file sul quale fare i case con questi case si potrà effettuare una statistica dei tipi di file nell'albero.
- avremo anche:
 - **path_alloc**: restituisce la massima dimensione disponibile per un path in un determinato sistema operativo: PATH_MAX + 1
 - **doaph**: funzione ricorsiva per poter entrare nei nodi dell'albero
 - **strcopy**: non sicura perché soggetta a overflow dello stack (strncpy più sicura)
 - **fullpath**: conterrà tutti i path delle foglie, dell'albero, da dare poi alla funzione lstat()
 - **lstat()**: cioè una "stat" ma che mantiene i link simbolici come tali
 - **dopath if 1**: controlla l'andamento di lstat
 - **dopath if 2**: se il file non è una directory fa un return perché è arrivato alla foglia e quindi risale la directory
 - **dopath if 3**: se il file è una directory allora ret = 0

Notare che per far fermare la lettura si usa il carattere 0 dato che null = 0 (ma != "0").

J. Funzione chdir()

È l'equivalente al comando "cd", al quale passiamo un path e lui **cambia al cwd corrente in quella indicata dal nuovo path**. Una sua variante è fchdir() che prende come parametro un file descriptor passato dalla funzione open con flag O_DIRECTORY.

IX. SYSTEM DATA FILES AND INFORMATION

A. Password file

Nei sistemi Linux è presente come:

```
| /etc/passwd
```

Nei sistemi MacOS, invece, non può essere vista in modalità "normale", infatti si può vedere solo se in modalità single user.

B. Time and Date Routines

Sono delle **routine per effettuare il conto del tempo utile** per il profiling di un programma dove si va a capire quando il programma si è tardato con l'esecuzione:

- **time()**: che ritorna il tempo in sec facendo riferimento alla epoch
- **gettimeofday()**: aggiunge i microsec quindi abbiamo una struttura di 2 interi
- **clock_gettime()**: aggiunge i nanosec quindi abbiamo una struttura di 2 interi

Questo tempo può essere **trasformato in una data**, infatti possiamo andare dai sec ad una struttura data tramite:

- **gmtime()**: tempo del meridiano 0
- **localtime()**: info sulla timezone raggruppate nella struttura "tm"
- **mktime()**: per effettuare il processo opposto

Per vedere quando compiremo 1.000.000.000 di sec:

```
| date -r $(( $(date -j -f %Y%m%d-%H%M%S  
20010101-190000 +%s) + 1000000000 )) '+%m/%d/%Y  
:%H:%M:%S'
```