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]
```

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 - 23/27.09.22

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 - 30.09.22*

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 - 28.09.22*

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
```

seguiamo il percorso:

```
/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.

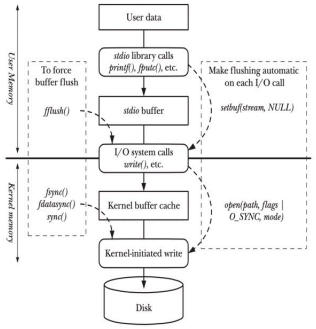

Figura 1. Schema unbuffered I/O

—- Prendiamo shell1.c che crea uno schell dal quale poter eseguire programmi:

```c
#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 della **stdoutput che legge la riga** che dai prima di dare invio e la mette in un buffer, con argomenti:
  - **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 di un "file"**

- if 1: permette di avere un null dove prima avevamo \n
-

if 2: abbiamo una fork che in genere viene invocata una volta e ritorna 2 volte dato che si creerà quella nel parent e quella nel child ritorna: 0 al child e pid del child al parent se il chld vuole sapere quelacosa sul parent si usa getppid(). se pid ¡ 0 allora la fork è fallita se è = 0 (caso del child)

questo if sarà ueseguito sia dal child che dal parent e poi

execlp: serve a far eseguire un altro codice che ha il suo codice sorgente e uno eseguibili. il programma è quello scritto in linea di comando (buf) se la roba va in porto e quinsi il programma eseguibile non esiste, continua con il nostro codice

il child chiude con. la exit 127 e rientra nel loop while

allora va avanti il parent con if 3: waitpid: se do un comando che impiega molto tempo non vedo la stampa del prompt. waitpid va ad aspettare il child quando avviene la fork tenendo appeso il prompt

pid: pid del child status: exit code del child qualdo finisce. indica la restituzione dei parametri di output al posto di quelli di input in modo da inserire in quell'indirizzo dai dati

i processi sono i programi in esecuzione . un programa quando esegue un porc ha un suo spazio di mempria virtuale grande quanto permette l'ndirizzamente dell amachhina su cui siamo: se a 32 bit allora $2^32 2bit$.

un porcesso pensa di avere tutto questo spazio a disposizione. in questo spazio c'è il codice da eseguire (detto testo in gergo UNIX). poi c'e dello paszio per le varibiali, uno stack. quidni ha bison di uno spazio dove fare le sue operazioni.

appena eseguita la fork si avranno 2 bash identici con lo spesso spazio di memoria (non che cindividono lo spesso spazio ma è solo un duplicato) ma pid diverso. appne fa la exec del programma, lo spazio di memroia viene azzerato e sostotuito dallo spazio di memoria del nuovo porgramma

CORREGGERE SOPRA CHE IL CHILD NON HA PID 0 MA È SOLAMENTE UN VARIABILE PER CONTENERE IL RISULTATO DELLA SC fork()

NEL WAITPID PID È IL PID DEL PROCESSO CREATO

PRIMA HO SBAGLIATO DOPO EXIT 127 IL CHILD MUORE

in pid dopo waitpid c'e il pid del child

importante capire che dopo la fork abbiamo 2 programmi .c con questo codice uguali in uno abbiamo l'esecuzoinone di un if ed in uno l'altro dato cheuno sarà un child e d uno il parent quidni uno avrà comre return della fork 0 ed nel parente la fork returna il pid del child

il child ceredita l'ambiente del parent che è nello spazio di memoria che si vienee a duplicare. questo fa si che la variabile di amm=biente PATH e1 vista dal child e può usare i programmi ai quali fa riferimnti

dopo le esxec questa può decidere se e quali vaiabili di ambiente gestire

img

THREAD

son ocome dei processi ma hanno la particolarità che ha la stesso spazio di memeroi del processo che lo ha generato.

quindi abbiamo un porbramma che in maniera asincorna al nostro e lavora nel nostro spazio di memoria.

eseguendo nello stesso spazio di memoria non si capisce nulla allora c'è una sincronizzazione tra le thread ma se non c'è ogniuno fa quello che vuole senza limiti. infatti il processo iniziale portà avere anche 2 trame di esecuzione contemporantea tutte nela stessa memoria. infatti si possoonocreare dei programmi multi thread in processori unicore.

i processi pensano di avere tutto il processore e la memoria per se il kernel da ai therad questa illuzione ma fa un time shering andando a far eseguire 10 millisec uno e poi un altro ecc ecc

GESTIONE DEGLI ERRORI

in genere se una funzione ritorna 0 se è andato tutto bene ma ci sono delle eccezionoic come la read che ritrona il numero di byte letti. ogni valore è specificato il suoi significato nel manuale dal quale si capisce cdove sono gli erriori.

le system call quando c'e un errore avvalorano la variabile "errno" che puo essere consultata in un progrmma con

```
extern int errno;
```

per usare questa variabile la si deve dichiarare come extern e definita nelle libreira di systema dove poi illinker gestisce tutto.

interessatne come variabile dato che viene avvalorata quando c'e un errore. il suo valore ha un isgnificato globale definito nel manuale:

```
$ man 2 intro
```

es: 2 ENOENT no sucj file or direcotory

il problema è che la variabile non viene svuotata quando passiamo l'errore . quindi per sapere quando e1 stato dato un erorre bisgna andare a consultarla quando la system call viene invocata

regole di errno:

1. valore amai resettata se non c'e un erroe e quidni bisgono cnsultarla solo quendo c'e1 stat l'errore

2. ha valore 0 se non e1 stata usata mai

delle funzioni per muoversi inquesto contesto sono:

strerror: restituisce la stringa del corrispettivo valore di errno

perror: legge errno e stampa un messagio che possiamo passare noi

```
#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 allo standard che vogliamo noi in questo caso allo starndard error, stampimao un errore

poi forziamo l'errore impostando errno ad un errore

perror: legge errno ed usa il nome del programma e poi lo stampa dicend cosa è successo

VI.

A.