perror: individuare l'errore quando una system call restituisce "-1"

- Quando una system call (o una funzione di libreria) non va a buon fine, restituisce come valore "-1"
- Come si fa a sapere più precisamente quale è l'errore che si è verificato?
- Si può usare la funzione C **perror**, insieme con la variabile esterna **errno**

1

```
/* USO DELLA FUNZIONE PERROR */
#include<stdio.h>
#include<signal.h>
#include<errno.h>

main()
{
int i;
printf("\nvalore iniziale di errno = %d\n",errno);

i = execl("ls"); //darà sicuramente un errore...

printf("\nrisultato exec = %d, errno = %d\n",i,errno);
perror("errore nella exec = ");

/* l'eventuale stringa passata in input a perror viene stampata insieme
con l'errore: serve per identificare meglio in quale punto del programma
si è verificato... */
```

2

```
/* MA, ATTENZIONE: ERRNO VIENE MODIFICATA SOLO IN
CASO DI ERRORE MA NON RESETTATA A ZERO SE NON SI
SONO VERIFICATI ERRORI */


i = signal(SIGUSR1,SIG_DFL); //questa funziona...
printf("\nrisultato signal = %d, errno = %d\n",i,errno);
perror("risultato = ");

/* ERRNO VA RESETTATA PRIMA DI CHIAMARE UN'ALTRA
SYSTEM CALL, ALTRIMENTI VIENE SEGNALATO UN ERRORE
ANCHE SE LA SYSTEM CALL E' ANDATA A BUON FINE...*/

errno = 0;
perror("risultato = ");
}
```

3

Per saperne di più

- man perror
- man –s 2 intro
  - qui potete leggere l'elenco di tutti i valori che può assumere la variabile errno, insieme con una descrizione più dettagliata dell'errore stesso
  - quando perror viene chiamata va a leggere il valore scritto dentro errno, e stampa sullo **standard error (stderr)** una breve descrizione dell'errore stesso.

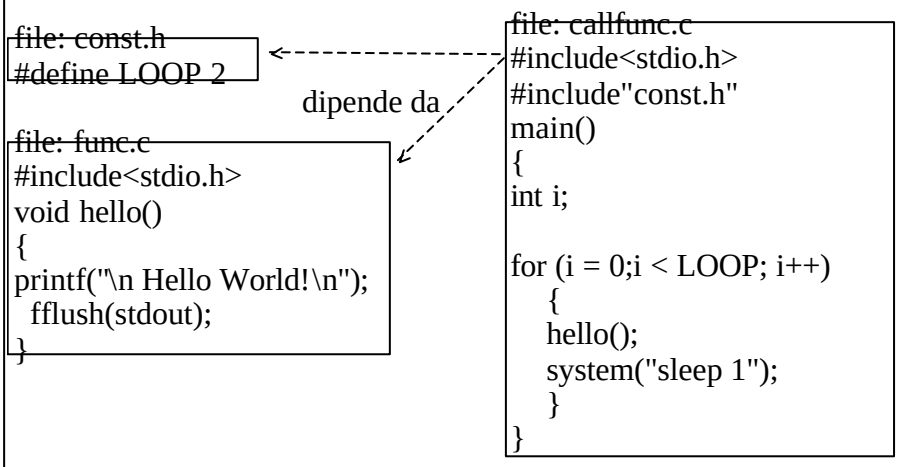
4

Make: mantenere, aggiornare e rigenerare programmi e file “collegati”

- grandi sistemi software sono fatti di vari moduli oggetto e file di definizioni, che vengono compilati e linkati insieme per dare origine a uno o più eseguibili.
- Quando vengono modificati solo alcuni dei moduli/file, può non essere agevole capire esattamente cosa sia necessario ricompilare per mantenere aggiornato tutto il software.
- Naturalmente si può semplicemente ricompilare tutto, ma questa non è una soluzione molto efficiente...
- Si può allora usare la utility **make** (che è comoda anche per piccoli sistemi software...)

5

esempio semplice semplice...



$> gcc -o pippo callfunc.c func.c

6

il makefile

- un makefile è un file di testo che contiene una sequenza di specifiche che rispettano la seguente sintassi:

TARGET: DIPENDENZE

tab

- per default un makefile prende il nome di "**makefile**" o di "**Makefile**" (ma altri nomi sono permessi)

7

il makefile

- **TARGET**: è l'oggetto che vogliamo ottenere
- **DIPENDENZE**: tutti i file da cui dipende TARGET
- **COMANDO**: che comando usare per ottenere TARGET a partire dai file da cui TARGET dipende

TARGET: DIPENDENZE

tab

8

Un semplice makefile

makefile

```
pippo: callfunc.c func.c const.h
        gcc -o pippo callfunc.c func.c
```

ossia:

- il target pippo dipende da: callfunc.c, func.c, const.h
- il target viene “raggiunto” eseguendo il comando specificato

9

Un semplice makefile

$>make

```
        gcc -o pippo callfunc.c func.c
```

- make cerca nella dir. corrente un makefile e **se anche uno solo dei file nella lista di dipendenze risulta più recente del target (pippo nel nostro caso) esegue il comando associato al target**
- per cui se make viene eseguito due volte di seguito, (e nessun file è stato modificato) ecco il risultato:

$>make

```
        make: ‘pippo’ is up to date
```

10

Il makefile

- Usato così, make è comodo perchè non occorre scrivere esplicitamente l'intero comando di compilazione, ma non permette di risparmiare tempo, poichè tutti i sorgenti vengono comunque ricompilati, anche quelli per cui non è necessario...
- Invece noi vogliamo che vengano ricompilati solo i programmi cambiati, e quelli che dipendono ai programmi cambiati
- per ottenere questo, dobbiamo dare ad un makefile più informazioni sulle dipendenze tra i vari programmi coinvolti nella compilazione

11

Il makefile

- il comando di compilazione:

  $> gcc -o pippo callfunc.c func.c

- ha l'effetto di compilare callfunc.c e func.c nei rispettivi file oggetto, e poi di linkare questi ultimi nell'eseguibile pippo
- Il comando è quindi equivalente a:

  $> gcc -c callfunc.c //genera callfunc.o

  $> gcc -c func.c //genera func.o

  $> gcc -o pippo callfunc.o func.o

- con la differenza che alla fine i due file oggetto callfunc.o func.o generati saranno lasciati nella cartella.

12

Il makefile

- Nel semplice esempio che abbiamo usato, supponiamo di modificare la costante LOOP (ossia il file const.h). Che cosa dobbiamo fare per produrre il nuovo eseguibile pippo?

  $> gcc -c callfunc.c

  $> gcc –o pippo callfunc.o func.o

- perchè callfunc.c dipende da const.h (infatti lo include), e pippo dipende da callfunc. Non c'è bisogno di ricompilare func.c, ma è sufficiente linkare func.o al nuovo callfunc.o
- Tutto questo possiamo farlo fare (cioè possiamo far decidere cosa è necessario fare) alla utility make.

13

Un makefile più articolato

makefile

```
func.o: func.c
        gcc -c func.c

callfunc.o: callfunc.c const.h
        gcc -c callfunc.c

pippo: callfunc.o func.o
        gcc -o pippo callfunc.o func.o
```

notate: il target pippo dipende dai target callfunc.o e func.o

- occhio! ora ci sono tre target nel makefile. In generale un makefile può contenere più target, ognuno con i propri comandi, e i vari target possono naturalmente essere collegati fra loro: E' IL NOSTRO CASO!

14

Un makefile più articolato

- Ora possiamo usare il makefile in diversi modi:

$>make — viene processato solo il primo target del makefile (func.o nel nostro caso)

$> make target1 — viene processato il target specificato (ad esempio: make callfunc.o)

$> make target1 target2 — possiamo anche specificare più target (ad esempio: make func.o callfunc.o)

$> make –f myfile target — possiamo usare un makefile con nome diverso da quello di default (in questo caso “myfile”)

15

Un makefile più articolato

- Ad esempio, provate a dare il comando: “make pippo” Il risultato sarà:

$>make pippo
gcc -c func.c
gcc -c callfunc.c
gcc -o pippo func.o callfunc.o

- ma se ora modificate const.h e rilanciate “make pippo”:

$>make pippo
gcc -c callfunc.c
gcc -o pippo func.o callfunc.o

- func.o non viene ricompilato, perchè non dipende da const.h

16

Le clausole all e clear

- un makefile contiene di solito le clausole all e clear:

makefile

```
func.o: func.c
        gcc -c func.c

callfunc.o: callfunc.c const.h
        gcc -c callfunc.c

pippo: callfunc.o func.o
        gcc -o pippo callfunc.o func.o

all: pippo func.o callfunc.o

clear:
        rm pippo func.o callfunc.o
```

17

Un makefile più articolato

- e possiamo usare il makefile in modo ancora più articolato:

$>make all

causa l'eventuale processazione di tutti i target specificati (pippo, callfunc.o e func.o nel nostro caso), ovviamente solo se è necessario

$> make clear

causa la rimozione di tutti i target specificati

18

macro e commenti

makefile

```
CC = gcc

func.o: func.c
        $(CC) -c func.c

callfunc.o: callfunc.c const.c
        $(CC) -c callfunc.c

# i commenti iniziano con il cancelletto

pippo: callfunc.o func.o
        $(CC) -c pippo callfunc.o func.o
```

19

alcune osservazioni

- make può essere usato per compilare programmi C, e in generale programmi in qualsiasi linguaggio il cui compilatore possa essere lanciato da linea di comando
- make non è in realtà limitato alla compilazione di programmi, ma può descrivere qualsiasi situazione in cui alcuni file devono essere aggiornati automaticamente quando alcuni altri sono cambiati (ad esempio file latex)
- In altre parole, il COMANDO associato ad un target non deve necessariamente essere una compilazione: make esegue qualsiasi cosa associata al target specificato
- per saperne di più: **man make**

20