

Laboratorio di Sistemi Operativi

Seconda Parte
Programmazione in C

La compilazione

- Le istruzioni contenute in uno script vengono lette ed eseguite da un apposito programma che le interpreta; il programma viene appunto indicato come **interprete**
- la shell è un interprete
- Il processo di interpretazione è quello che permette, mediante una analisi sintattica e lessicale, la *conversione* dei comandi in opportune sequenze di codici macchina.
- Alternativamente, è possibile *codificare* il comportamento di un programma mediante un *linguaggio di programmazione* che possa essere *compilato*.
- La compilazione è il procedimento che traduce la codifica di un programma (*sorgente*) in un formato *binario* direttamente eseguibile dalla macchina.

Compilazione - extra

- Esiste una terza possibilità che contempla la compilazione di un sorgente in un formato binario, non eseguibile direttamente dalla CPU ma da un ulteriore interprete.
- Il vantaggio di questa strategia risiede nell'inserimento di uno strato di *compatibilità*, individuabile nell'interprete stesso, che isola dalla particolare CPU, senza peraltro rallentare *troppo* l'esecuzione del programma.
- Un esempio è individuabile nel Java che ammette la compilazione nel *java byte code* che verrà poi interpretato dalla *java virtual machine*. Ogni dispositivo che contiene una JVM può eseguire allo stesso modo il programma, indipendentemente da sistema operativo e CPU.

Il compilatore

- Il programma che si occupa di convertire un sorgente nel corrispondente file binario è il compilatore.
- Il file binario prodotto dal compilatore può non essere direttamente eseguibile e richiedere un ulteriore processo di *organizzazione* detto *link*. Il file prodotto dal compilatore, infatti, può non essere completamente *risolto* in quanto i riferimenti ad oggetti non definiti nel sorgente (es. chiamate di funzioni di libreria) devono essere rintracciati in altri file.
- Solitamente si indica come *file oggetto* il risultato della fase di compilazione e come *file eseguibile* quello finale.
- E' anche possibile raggruppare più file oggetto in *librerie* che potranno essere *linkate* a più applicazioni.

Il compilatore C

- Il linguaggio di programmazione selezionato per questo corso è il C. Si farà affidamento sulla precedente conoscenza del linguaggio.
- L'invocazione del compilatore C si ottiene dalla linea di comando specificando il nome del file sorgente più altre eventuali opzioni ed argomenti.
- Alcuni compilatori, senza un'opportuna opzione della linea di comando, invocano automaticamente il linker per produrre direttamente l'eseguibile.
- La produzione del solo file oggetto si ottiene specificando l'opzione -c
- L'indicazione del nome di file in uscita si ottiene con lo switch -o

Il linker

- La risoluzione dei collegamenti fra più *moduli* oggetto e librerie è demandata al linker
- il linker accetta i nomi dei file oggetto e delle librerie da collegare più, eventualmente, il nome del file eseguibile da produrre.
- Le esercitazioni in laboratorio faranno uso del GCC sia come compilatore (se invocato con sorgenti e switch -c) che come linker (se invocato con file oggetto).
- Per ogni progetto è buona norma fornire anche un makefile
- Il make usato è quello GNU

L'interfaccia al Sistema Operativo

- L'equivalente di comandi e built-in della shell è da ricercare nelle funzioni della libreria standard del C o in una loro composizione
- Utilizzando opportunamente le funzioni di libreria è possibile realizzare la funzione di qualsiasi shell script ma con una velocità ed un controllo molto maggiore.
- L'accesso alle funzioni di libreria si ottiene includendo nei sorgenti la definizione delle interfacce (file *header .h*) ed inserendo nel processo di linking la libreria standard; utilizzando gcc anche come linker, l'inserimento delle librerie standard in fase di linking è automatico. Per i puristi è comunque possibile usare 'ld', avendo l'accortezza di elencare tutti i moduli necessari alla produzione dell'eseguibile -> *esercizio?*

Strumenti messi a disposizione dalla libreria C

- Gestione errori: `errno`, `strerror`
- Allocazione memoria: `malloc`, `obstacks`, `alloca`
- Manipolazione caratteri: `isalpha`, `isdigit`, ...
- Manipolazione stringhe: `strcpy`, `strcat`, `strchr`, `strtok`
- Ricerca, ordinamento e regexp: `bsearch`, `fnmatch`, `regcomp`
- I/O
 - stream I/O: `fopen`, `fprintf`, `fputc`, `fgetc`
 - low level I/O: `open`, `read`, `write`, `ioctl`, file locks
 - interfaccia al file system: `opendir`, `readdir`, `stat`, `unlink`
 - pipe, fifo e socket: `pipe`, `popen`, `mkfifo`, `socket`, `bind`, `socketpair`, `connect`, `listen`, `send`, `recv`
 - interfaccia al terminale: `tcsetattr`, `tcgetattr`

... continua

- Funzioni matematiche: sin, cos, atoi, div
- Data, ora e timer: clock, difftime, ctime, setitimer, sleep
- Segnali: signal, sigaction, raise, kill, pause
- Avvio e terminazione di processi: getopt, getenv, putenv, exit, atexit, on_exit, fork, exec, wait

Strumenti analizzati nel corso

- Questo corso esaminerà, in prevalenza, gli strumenti della LIBC che interagiscono con il kernel per mettere a disposizione le facilitazioni del sistema operativo.
- Gli strumenti *algoritmici* come i vari sort ed analizzatori sintattici sono presenti indipendentemente dal sistema operativo (si possono trovare anche in ambienti di sviluppo embedded os-less).
- Analizzeremo, in pratica, gli strumenti per il process-handling, l'inter-process-communication (IPC), e l'interazione con il filesystem.
- Questi strumenti, pur con svariate eccezioni e particolarità, sono disponibili anche in altri sistemi operativi sia per PC che embedded, sia standard che real-time.

Esecuzione di un programma

- Prendiamo in esame l'operazione di avvio o lancio di un programma eseguibile dal prompt della shell:
 - la shell individua il file corrispondente al nome del programma se esiste ed è eseguibile
 - la shell crea un nuovo processo che eseguirà il programma mediante la chiamata *fork* della LIBC
 - la libreria esegue l'invocazione del sistema operativo mediante *syscall 2* che crea un nuovo thread clonando l'attuale
 - il processo esistente di shell attenderà o meno il completamento del programma (figlio) e ripresenterà il prompt
 - il nuovo processo esegue una chiamata alla funzione di libreria C *exec...()*
 - la libreria esegue l'invocazione del sistema operativo mediante *syscall 11* che legge il binario del programma (codice e dati) sovrascrivendo l'immagine (core) di quello attuale.
 - Il nuovo processo esegue il codice contenuto nell'eseguibile.

Parametrizzazione di un programma

- Un programma può accedere a diverse informazioni relative alla propria invocazione:
 - accedendo all'elenco di parametri eventualmente passati sulla linea di comando della shell all'atto dell'invocazione
 - accedendo al database di variabili *ambiente* in essere per il processo padre al momento dell'invocazione
 - accedendo alla copia locale di tutte le variabili in essere per il processo padre all'atto dell'invocazione
 - accedendo al file system
 - comunicando con altri processi tramite il sistema operativo
 - accedendo all'hardware mediante (meglio) l'ausilio del sistema operativo
- L'uso di queste informazioni permette l'interazione con il programma che, diversamente, compierebbe sempre la stessa funzione.

I parametri della linea di comando

- L'entry point di un programma C è la funzione *main* che viene invocata (questo è il compito della routine di *startup* fornita con la libreria standard) con due parametri:
 - int argc : numero di argomenti della linea di comando
 - char *argv[] : array di stringhe che rappresenta il contenuto dell'intera linea di comando suddiviso da spazi (*)
- E' possibile analizzare queste informazioni manualmente o mediante alcune funzioni di libreria:
 - getopt
 - getopt_long (estensione GNU)
- L'operazione con la quale si interpreta la linea di comando è solitamente indicata come *command line parse*.

hello world

```
/* ****  
 * $Id: hello.c,v 1.1 2002/02/11 08:50:46 valealex Exp $  
 * Classico 'Hello World'. Scrive sullo standard output  
 * una stringa ed esce.  
 * Compilare con gcc -o hello hello.c  
 * *****/  
  
/* inclusione dell'interfaccia (prototipo) per printf */  
#include <stdio.h>  
  
/* prototipo della funzione 'main' */  
int main(int argc, char **argv);  
  
/* definizione della funzione 'main' */  
int main(int argc, char **argv) {  
    /* argc ed argv non sono usati, il compilatore  
     * potrebbe dare un messaggio di warning */  
    printf("Hello world\n");  
    /* uscita dalla funzione main -> uscita dal programma */  
}
```

makefile per hello world

```
# #####  
# Makefile di esempio per hello.c  
# $Id: Makefile,v 1.1 2002/02/11 08:56:47 valealex Exp $  
# #####  
  
# Il primo 'target' e' quello che viene 'costruito' per  
# default con l'invocazione 'make' nella directory in  
# cui si trova Makefile  
  
hello: hello.c  
    gcc -o hello hello.c  
  
# Indica che:  
# hello dipende da hello.c (hello: hello.c)  
# Se hello non esiste o e' piu vecchio di hello.c allora make  
# esegue le istruzioni che seguono. Attenzione, ogni linea di  
# istruzione deve iniziare con un TAB e non con spazi (alcuni  
# editor 'grafici' sostituiscono i tab con spazi e, in tal  
# caso make dara' un errore.
```

lettura argomenti

```
/* **** */
* $Id: clargs.c,v 1.1 2002/02/11 09:06:22 valealex Exp $
* Compilare con gcc -o clargs clargs.c
* **** */
/* inclusione dell'interfaccia (prototipo) per printf */
#include <stdio.h>
/* prototipo della funzione 'main' */
int main(int argc, char **argv);
/* definizione della funzione 'main' */
int main(int argc, char **argv) {
    /* argc e' il contatore degli argomenti (come $# nella shell */
    int contatore; /* variabile di appoggio */
    for(contatore=0; contatore<argc; contatore++) {
        /* ciclo for per tutti gli argomenti della linea di comando.
        * L'argomento i-esimo e' la stringa con indirizzo argv[i].
        * Ricordate che l'indirizzo di una stringa (array di caratteri)
        * e' l'indirizzo al quale si trova il primo carattere e la
        * stringa termina con il primo carattere nullo. */
        printf("Argomento n.%d=%s\n", contatore, argv[contatore]);
    }
}
```

makefile per i due target

```
# #####  
# Makefile di esempio  
# $Id: Makefile,v 1.2 2002/02/11 09:10:21 valealex Exp $  
# #####  
  
# Creiamo un target fittizio 'all' che provochi la  
# compilazione di tutti i nostri eseguibili, senza  
# alcuna istruzione (linea vuota che segue)  
all: hello clargs  
  
# il programma precedente  
hello: hello.c  
    gcc -o hello hello.c  
  
# il nuovo  
clargs: clargs.c  
    gcc -o clargs clargs.c
```

target 'clean'

```
# #####  
# Makefile di esempio  
# $Id: Makefile,v 1.3 2002/02/11 09:13:06 valealex Exp $  
# #####  
  
all: hello clargs  
  
hello: hello.c  
    gcc -o hello hello.c  
  
clargs: clargs.c  
    gcc -o clargs clargs.c  
  
# creiamo un altro target per richiedere la 'pulizia'  
# di tutti i file costruiti. Notare che questo target non  
# ha dipendenze, quindi viene eseguito sempre.  
# Per richiedere la pulizia e' sufficiente digitare:  
# make clean  
clean:  
    rm hello  
    rm clargs
```

comandi complessi

```
# #####  
# Makefile di esempio per hello.c  
# $Id: Makefile,v 1.4 2002/02/11 09:14:35 valealex Exp $  
# #####  
  
all: hello clargs  
  
hello: hello.c  
    gcc -o hello hello.c  
  
clargs: clargs.c  
    gcc -o clargs clargs.c  
  
# Inseriamo un controllo per evitare che si cerchi di  
# cancellare un file inesistente:  
clean:  
    if [ -f hello ]; then rm hello; fi  
    if [ -f clargs ]; then rm clargs; fi
```