Mediul de programare

Utilizarea Sistemelor de Operare

Paul Irofti

Universitatea din București
Facultatea de Matematică și Informatică
Department de Informatică
Email: paul.irofti@fmi.unibuc.ro

De la sursă la executabil

- sursa – fișierul scris în limbaj de programare (ex. C, C++)
- compilator – traduce fișierul sursă în limbaj mașină (de asamblare)
- obiect – fișier rezultat din compilarea unui fișier sursă
- bibliotecă – fișier binar (deja compilat) ce oferă o anumită funcționalitate (ex. funcții de comunicare în rețea)
- linker – leagă obiecte și biblioteci pentru a produce executabilul
- executabil static – toate obiectele și bibliotecile se regăsesc în fișierul executabil rezultat
- executabil dinamic – conține doar informațiile și instrucțiunile proprii; bibliotecile sunt păstrate în fișiere separate

Load time

Rezolvarea dependențelor executabilelor dinamice la momentul încărcării în memorie:

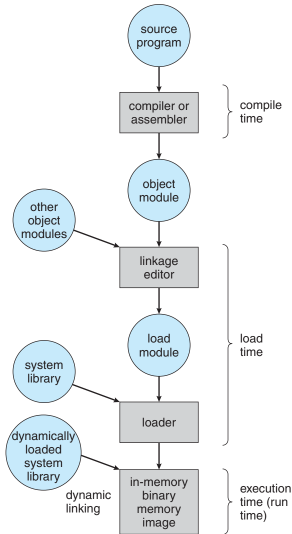
1. sistemul de operare încarcă executabilul în memorie
2. înainte de a fi executat se caută bibliotecile folosite
3. se încarcă toate bibliotecile necesare
4. în caz că nu se găsesc una sau mai multe biblioteci executabilul este scos din memorie și execuția este anulată

Run time

Rezolvarea dependențelor executabilelor dinamice la nevoie:

1. sistemul de operare încarcă executabilul în memorie
2. înainte de a fi executat se caută bibliotecile strict necesare lansării programului
3. pe parcursul execuției dacă este nevoie de o bibliotecă
   3.1 execuția programului este oprită
   3.2 se caută în sistem biblioteca necesară
   3.3 se încarcă în memorie
   3.4 se repornește de unde a rămas execuția programului
4. în orice moment dacă o bibliotecă nu este găsită și încărcată executabilul este scos din memorie și execuția este anulată

Compilare



http://codex.cs.yale.edu/avi/os-book/

Compilatoare

În UNIX există două soluții principale:

- gcc – GNU Compiler Collection (C, C++, Java etc)
- clang – LLVM front-end pentru C, C++, Objective C, OpenCL etc.

Indiferent de compilator, în UNIX există comanda `cc` (C compiler) și, opțional, comanda `c++` (C++ compiler).

Executabil simplu

În forma cea mai simplă compilatorul

- primește ca argumente fișierele sursă
- produce executabilul `a.out`
- denumit din motive istorice: formatul de executabil standard pe prima versiune UNIX

Example

```
$ ls
hello.c
$ cc hello.c
$ ls
a.out    hello.c
$ ./a.out
Hello, World!
```

Pentru execuție prefixăm `a.out` cu `./` pentru a spune shell-ului să nu caute executabilul în `$PATH` ci în directorul curent.

Obiecte

- se folosește opțiunea -c
- compilatorul se oprește după producerea obicetului
- nu continuă cu crearea unui executabil
- obiectul va fi folosit împrenuă cu alte obiecte și biblioteci pentru a produce executabilul final

Example

```
$ ls
hello.c
$ cc -c hello.c
$ ls
hello.o   hello.c
```

Biblioteci

Pentru a folosi biblioteci externe

- se folosește opțiunea `-l`*nume* (**fără spații!**)
- compilatorul caută biblioteca într-o listă de directoare
- lista este de regulă generată și întreținută de comanda `ldconfig(1)`
- poate fi folosită variabila de mediu `$LD_LIBRARY_PATH` pentru a adăuga directoare proprii ce conțin biblioteci
- biblioteca găsită este folosită împrenună cu alte obiecte și biblioteci pentru a produce executabilul final

Example

Folosirea bibliotecii de sistem *crypto*:

```
$ cc hello.c -lcrypto
```

Opțiuni utile

Prezentăm aici câteva opțiuni utile folosite des la compilare

- `-o` *nume* – numele executabilului sau obiectului rezultat (ex. `hello` înloc de `a.out`)
- `-Wall` – afișează toate mesajele de *warning*
- `-g` – pregătește executabilul pentru o sesiune de depanare (debug)
- `-O`*număr* – nivelul de optimizare (0 $\rightarrow$ fără optimizări)
- `-O2` este folosit implicit, `-O0` este folosit adesea împreună cu `-g` pentru depanare

Example

Compilează `hello.c` cu simboluri de debug, fără optimizare, și numește executabilul `hello`

```
$ cc -g -O0 hello.c -o hello
```

Depanare

Noțiuni folosite într-o sesiune de depanare

- break, breakpoint
  - punct în care să se oprească execuția
  - poate fi o funcție, o linie dintr-un fișier, chiar și o instrucțiune de asamblare
  - se poate investiga starea variabilelor la momentul respectiv
- backtrace, trace, stack trace
  - apelul de funcții care a dus în punctul curent
  - exemplu: `main()` $\rightarrow$ `decompress()` $\rightarrow$ `zip_decompress()`
- watch, watchpoint
  - punct în care să se oprească execuția **dacă este îndeplinită o condiție cerută de utilizator**
  - poate fi modificarea unei zone de memorie sau a unei variabile

Programul cel mai comun folosit pentru depanarea executabilelor este `gdb(1)`.

Comenzi `gdb(1)`

Comenzi uzuale

- `break` *simbol* – setează breakpoint la simbolul respectiv (ex. `main()` sau `hello.c:10`)
- `run` – începe execuția (de regulă după ce au fost setate breakpoint-urile)
- `next` – o dată ajuns într-un punct de oprire, mergi la următoarea instrucțiune
- `continue` – continuă până la următorul punct de oprire sau până la terminarea execuției programului
- `print` *variabilă* – afișează valoarea unei variabile
- `quit` – ieșire din `gdb(1)`

Sesiune de depanare

Example

```
$ cc -g -O0 hello.c -o hello
$ gdb hello
(gdb) break main
Breakpoint 1 at 0x546: file hello.c, line 5.
(gdb) run
Starting program: /home/paul/wrk/ub/uso/curs/3/hello
Breakpoint 1 at 0x101a49c00546: file hello.c, line 5.

Breakpoint 1, main () at hello.c:5
5               printf("Hello, World!");
(gdb) next
Hello, World!
6               return 0;
(gdb) continue
Continuing.
Program exited normally.
(gdb) quit
```

Proiecte

Produsele software sunt alcătuite din mai multe componente:

- diferite module (ex. comunicație, logging)
- diferite biblioteci (ex. compresie, criptografie, http)
- mai multe fișiere sursă (numite și unități de compilare)
- între toate acestea apar diferite dependențe (ex. modulul de comunicație depinde de biblioteca http)
- inter-dependențele dictează ordinea de compilare
- dependențele și ordinea de compilare este scrisă formal în fișiere de tip `Makefile`
- instrucțiunile dintr-un `Makefile` sunt executate cu ajutorul comenzi `make(1)`

Makefile

Format fix

```
target: dependency1 dependency2 [...]
        commands
```

- `target` – ce va fi produs
- `dependency` – ingredientele necesare (deja existente)
- `commands` – comenzile pentru a produce `target`-ul
- **Atenție!** – este un `TAB` înaintea `commands`
- primul `target` din fișier sau, dacă există, `target`-ul `all` vor fi executate implicit

Avantaj: recompilează doar fișierele modificate!

Makefile simplu

Example

```
$ cat Makefile
all: hello
hello: hello.c
        cc -o hello hello.c
clean:
        rm hello
$ make
cc -o hello hello.c
$ make clean
rm hello
$ make hello
cc -o hello hello.c
```

Variabile `Makefile`

Câteva variabile utile

- `$@` – numele `target`-ului curent (partea stângă)
- `$^` – toate dependențele (partea dreaptă)
- `$<` – numele primei dependențe (ex. `dependency1`)
- `.type1.type2:` – `target`-ul transformă fișere de tip 1 în fișiere de tip 2 (ex. `.c.o:`)
- `$(VAR:old=new)` – înlocuiește `old` cu `new` în variabilă

Makefile complex

Example

```
PROG=hello
SRCS=hello.c
OBJS=$(SRCS:.c=.o)
CC=cc
CFLAGS=-g -O0
INSTALL=install

all: $(PROG)

$(PROG): $(OBJS)
	$(CC) -o $@ $^ $(CFLAGS)
.c.o:
	$(CC) -c -o $@ $< $(CFLAGS)
install: $(PROG)
	$(INSTALL) $< ${HOME}/bin
clean:
	-rm $(PROG) $(OBJS)
```