

Laboratorul 1

Linia de comandă și execuția

1 Familiarizarea cu terminalul

În Tabelul 1 găsiți câteva comenzi utile navigării în terminal. Pentru fiecare citiți bine prima parte din manual. De exemplu pentru comanda `cp(1)` manualul începe cu

```
CP(1)                                General Commands Manual                                CP(1)
```

NAME

```
cp - copy files
```

SYNOPSIS

```
cp [-fipv] [-R [-H | -L | -P]] source target
cp [-fipv] [-R [-H | -L | -P]] source ... directory
```

DESCRIPTION

```
In the first synopsis form, the cp utility copies the
contents of the source file to the target file.
```

Prima linie ne spune că ne aflăm în secțiunea (1) numită **General Commands Manual**. Ce urmează în paranteză după numele comenzii specifică secțiunea. Mai departe avem rezumatul comenzii și tipul de parametri pe care-i acceptă. După introducere urmează descrierea pe larg. Pentru a vedea o comandă dintr-o secțiune anume folosiți:

```
$ man 1 write
```

```
$ man 2 write
```

unde prima comandă descrie `write(1)` iar a doua syscall-ul (funcția de sistem) `write(2)`.

Câteva operații și simboluri folosite în linia de comandă sunt descrise în Tabelul 2.

Comandă	Descriere
<code>man command</code>	manualul de utilizare
<code>pwd</code>	directorul curent
<code>ls</code>	conținutul directorului curent
<code>cp source target</code>	copiere fișiere
<code>mv source target</code>	mutare fișiere
<code>rm item</code>	ștergere fișiere
<code>mkdir dir</code>	creare director
<code>rmdir dir</code>	ștergere director gol
<code>echo str</code>	repetare string
<code>cd path</code>	schimbă directorul curent

Tabela 1: Comenzi uzuale

Simbol	Descriere
<code>.</code>	directorul curent
<code>..</code>	directorul părinte
	acasă (<code>/home/souser</code>)
<code>cmd > file</code>	redirecționare ieșire către fișier
<code>cmd1 cmd2</code>	pipe: legătură ieșire-intrare
<code>^</code>	tasta <code>ctrl</code>
<code>^w</code>	tastat concomitent <code>ctrl+w</code>

Tabela 2: Simboluri și operații în terminal

Urmează o sesiune-exemplu în terminal unde folosim câteva dintre comenzile și operațiile descrise mai sus.

```
$ pwd
/home/souser
$ touch foo
$ ls
foo
$ cp foo bar
$ ls
bar foo
$ mv bar baz
$ ls
baz foo
$ rm baz
$ ls
foo
$ mkdir test
$ cd test/
$ pwd
```

```

/home/souser/test
$ cd ..
$ rmdir test
$ echo hello
hello
$ echo hello > hello.txt
$ cat hello.txt
hello

```

2 Analizarea execuției unui binar

Pentru a observa cum încarcă sistemul de operare un executabil de pe disc, în cele ce urmează vom crea un executabil simplu de tip helloworld.

```

#include <stdio.h>

int main()
{
    printf("Hello ,\World!\n");
    return 0;
}

```

Acest program se poate scrie cu ajutorul oricărui editor. În linia de comandă se pot folosi `nano(1)`, `vi(1)`, `emacs(1)` sau oricare alt editor. Subiectul acesta nu face obiectul laboratorului.

2.1 Funcții sistem

Mai departe, analizăm funcțiile de sistem (syscalls) folosite pentru a executa binarul `hello`.

```

$ gcc helloworld.c -o hello
$ ./hello
Hello , World!
$ ktrace ./hello
Hello , World!
$ kdump

```

Comanda `ktrace(1)`, prescurtare de la kernel trace, ne ajută să vedem de ce funcții de sistem are nevoie `hello` pentru a fi executat. Echivalentul în Linux este `strace(1)`. Comanda `kdump(1)` ne ajută să vedem fișierul binar creat de `ktrace(1)`. Pornirea execuției are loc prin apelul funcției de sistem `execve(2)`.

```

46707 ktrace   CALL   execve(0x7f7ffffd5f08 ,0x7f7ffffd5da0 ,0x7f7ffffd5db0)
46707 ktrace   NAMI   "./hello"
46707 ktrace   ARGS

```

```

        [0] = "./hello"
41281 hello    NAMI  "/usr/libexec/ld.so"
41281 hello    RET   execve 0

```

Valorile hexazecimale sunt adrese din memorie corespunzătoare argumentelor apelului de sistem. Pentru a le descifra, folosiți manualul (`$ man 2 execve`).

```

EXECVE(2)          System Calls Manual
EXECVE(2)

```

```

NAME
    execve - execute a file

```

```

SYNOPSIS
    #include <unistd.h>

    int
    execve(const char *path, char *const argv[], char *const envp[]);

```

Părțile direct influențate de programul nostru sunt

```

41281 hello    CALL  write(1,0x7bd390d0000,0xe)
41281 hello    GIO   fd 1 wrote 14 bytes
        "Hello, World!
        "
41281 hello    RET   write 14/0xe

```

unde mesajul este afișat pe ecran cu ajutorul syscall-ului `write(2)`. Conform `$ man 2 execve`

```

WRITE(2)          System Calls Manual
WRITE(2)

```

```

NAME
    write, writev, pwrite, pwritev - write output

```

```

SYNOPSIS
    #include <unistd.h>

    ssize_t
    write(int d, const void *buf, size_t nbytes);
    [...]

```

```

DESCRIPTION
    write() attempts to write nbytes of data to the object
    referenced by the descriptor d from the buffer pointed
    to by buf.

```

Descriptorii sunt folosiți pentru a indica spre anumite fișiere din sistem. În loc de a folosi `/dir/subdir/file.txt` putem folosi un număr despre care știm

Descriptor	Fișier	Folosit de
0	stdin	scanf(3)
1	stdout	printf(3)
2	stderr	perror(3)

Tabela 3: Descriptori rezervați

că atunci când apare se referă la acest fișier (este vorba de o structură de tip cheie-valoare). Primii trei descriptori sunt rezervați. Detalii în Tabelul 3.

2.2 Biblioteci

Funcția `printf(3)` folosită în `helloworld.c` este implementată în biblioteca standard de C, numită `libc`. Cu ajutorul utilitarului `ldd(1)` putem vedea de ce biblioteci are nevoie `hello` pentru a fi executat.

```
$ ldd ./hello
./hello:
Start      End          Type Open Ref GrpRef Name
9e7c6e0000 9e7c7002000 exe  1   0   0   ./hello
9ea8816b000 9ea8844a000 rlib 0   1   0   /usr/lib/libc.so.90.0
9ea60800000 9ea60800000 rtld 0   1   0   /usr/libexec/ld.so
```

Primele coloane indică unde în memorie poate fi găsită fiecare bibliotecă. Putem vedea că pe lângă `libc` mai este necesară și `ld.so`. Aceasta din urmă este o bibliotecă specifică cu care sistemul de operare caută, găsește și încarcă în memorie bibliotecile utilizate de executabil (în cazul nostru doar `libc`).

2.3 Simboluri

Pentru a vedea de ce simboluri are nevoie `hello`, putem folosi `nm(1)`.

```
$ nm ./hello
00200e50 a  _DYNAMIC
00200fb0 a  _GLOBAL_OFFSET_TABLE_
          W  _Jv_RegisterClasses
00201000 A  __bss_start
00201000 A  __data_start
00201000 B  __dso_handle
00000550 T  __fini
00200e40 D  __guard_local
00000350 T  __init
00000420 W  __register_frame_info
000003c0 T  __start
          U  _csu_finish
00201000 A  _edata
00201058 A  _end
```

Comandă	Descriere
b symbol	oprirea execuției la simbol
p var	tipărește valoarea variabilei
n	următoarea instrucțiune
c	continuarea execuției
q	ieșire

Tabela 4: Comenzi gdb(1)

```
000003c0 T _start
          U atexit
          U exit
00000000 F helloworld.c
00000528 T main
          U puts
```

Prima coloană arată adresa la care se găsește fiecare simbol. A doua indică tipul simbolului (**U** este prescurtarea de la **Undefined**) iar ultima numele. Executabilul nostru folosește explicit doar `printf(3)` din biblioteca standard care în exemplul de sus este implementat cu ajutorul lui `puts(3)` iar adresa la care se va găsi această funcție va fi stabilită când se va încărca în memorie biblioteca C, în timpul execuției (vezi secțiunea precedentă). Un alt simbol cunoscut este `main` și `helloworld.c` (fișierul sursă).

2.4 Instrumentare

Un prim utilitar folosit la investigarea și instrumentarea executabilelor este `gdb(1)`. Întâi vom recompila cu simboluri de debug:

```
$ gcc -g -O0 helloworld.c -o hello
```

Opțiunile în plus sunt `-g` care adaugă efectiv simbolurile (numărul liniei în fișierul sursă, instrucțiunea C etc.) și `-O0` pentru a elimina optimizările compilatorului ce ar putea rezulta în eliminarea anumitor variabile sau instrucțiuni C, pierzându-se astfel corespondența cu fișierul sursă.

În Tabelul 4 sunt puse câteva instrucțiuni uzuale folosite pentru a depana cu `gdb`. Încheiem cu o sesiune de depanare unde se observă faptul că breakpoint-ul se pune de fapt la o adresă în memorie, deși argumentul este un simbol și liniile din fișierul sursă sunt afișate corespunzător.

```
$ gdb ./hello
(gdb) b main
Breakpoint 1 at 0x530: file helloworld.c, line 5.
(gdb) r
Starting program: /home/souser/hello
Breakpoint 1 at 0xbac61500530: file helloworld.c, line 5.

Breakpoint 1, main () at helloworld.c:5
```

```
5             printf("Hello, World!\n");
(gdb) n
Hello, World!
6             return 0;
(gdb) n
7         }
(gdb) n
0x00000bac61500414 in _start () from /home/souser/hello
(gdb) n
Single stepping until exit from function _start,
which has no line number information.

Program exited normally.
(gdb) q
```