1. Introducere

Acest laborator are ca scop familiarizarea studentilor cu

- moduri/optiuni de compilare a programelor in Linux
- executabilele si bibliotecile in Linux
- interactiunea dintre biblioteci si executabile in Linux

Pentru realizarea acestui laborator este necesar ca urmatoarele pachete Debian sa fie instalate: **`gcc g++ libc6-dev binutils file vim diff patch make strace`**

2. gcc - GNU C Compiller

2.1. gcc vs. g++

Scrieti urmatorul program C in editorul preferat:

Program name: m.c
[programul nu face nimic util :) - este utilizat in scop pur educativ]

```
int prod(int a, int b) {
        return a*b;
}

int main(int argc, char **argv){
        printf("\n%s",argv[0]);
        int i,j,p;
        p=1;
        for (j=0;j<10000000;j++)   // 7 zero-uri
        for (i=0;i<3;i++){
                p=prod(p,i+j);
        }
        printf("%d\n",p);
        return p;
}
```

Compilarea unui program C se face utilizand GNU C Compiler:

```
$ gcc <sursa> -Wall -o <fisier_executabil>
```

- Fara optiunea „`-o <fisier executabil>`” executabilul rezultat va fi denumit a.out
- optiunea `-Wall` activeaza toate warning-urile compilatorului (de multe ori sunt utile)

Compilati asfel programul C de mai sus utilizand:

```
$ gcc m.c -Wall -o m
m.c: In function `main':
m.c:12: warning: implicit declaration of function `printf'
```

Veti observa cum compilatorul indica faptul ca printf() nu este declarata si ca este utilizata declaratia implicita a acestuia. Pentru a corecta acest warning, este suficienta adaugarea urmatoarei linii la inceputul programului

Program name: m.c

```
# include <stdio.h>
// ...
```

Nota: fisierele executabilele in Linux sunt marcate cu un atribut special (chmod +x numefisier), extensia fiind ignorata

Modificati programul anterior asfel incat sa ajungeti la urmatoarea forma (modificarile sunt marcate cu **bold**) – salvati fisierul rezultat cu numele **n.cpp**.

Program name: n.cpp

```
#include <iostream>

using namespace std;

int prod(int a, int b) {
        return a*b;
}

int main(int argc, char **argv){
        cout << endl << argv[0];
        int i,j,p;
        p=1;
        for (j=0;j<10000000;j++)   // 7 zero-uri
        for (i=0;i<3;i++){
                p=prod(p,i+j);
        }
        return p;
}
```

Ati obtinut astfel un program C++, prin modificarea programului C anterior.

Nota: faptul ca are extensia .cpp nu il face un program C++. Compilatorul poate compila fisiere cu orice extensie, atat timp cat este instruit corect prin parametrii cum anume sa interpreteze continutul.

Compilarea programului C++ se realizeaza asemanator compilarii unui program C.

```
$ g++ n.cpp -Wall -o n
```

Observati ca s-a modificat numele compilatorului: am folosit de data aceasta compilatorul **g++**, nu **gcc** precum in cazul precedent. C++ vine cu GCC-ul ca biblioteca separata de functii. Mai multe despre biblioteci de functii putin mai tarziu.

Pentru moment, ca dovada a faptului ca C++-ul vine ca o biblioteca de functii, incercati comanda urmatoare:

```
$ gcc n.cpp -Wall -lstdc++ -o n
```

Observati ca:

- am folosit compilatorul gcc
- am utilizat optiunea de compilare „-lstdc++” (se scrie cu „L” mic) – aceasta instruieste compilatorul ca in faza de link-editare a legaturilor sa includa si biblioteca de functii standard C++

2.2. Optimizarea compilarii cu GCC

Pentru a studia impactul diferitelor optiuni de compilare, introduceti comanda:

```
$ for level in 0 1 2 3; do gcc m.c -O$level -o m$level; done
```

Aceasta comanda va compila fisierul m.c utilizand nivelurile de optimizare de la 0 la 3, obtinandu-se executabilele m0, m1, m2, m3, corespunzatoare fiecarui nivel de optimizare.

Vom utiliza pentru analiza timpilor de executie comanda „time" care primeste ca parametru numele executabilului ce se doreste a fi testat („man time" pentru mai multe informatii).

Programul l-am facut special ca la pornire sa tipareasca numele executabilului. Se vor observa astfel timpii de rulare ai fiecarei versiuni a executabilului, in functie de nivelul optimizarii. Se observa ca timpii nu scad permanent. In functie de codul ce se compileaza, exista cazuri cand timpii pentru un nivel de optimizare mai mare sunt mai mari decat pt un nivel mai mic de optimizare.

```
$ for level in 0 1 2 3; do time ./m$level; done
```

Pentru a se verifica diferentele de cod ce se produc in cazul diferitelor niveluri de optimizare, vom utiliza parametrul -S al GCC-ului pentru a obtine codul assambler.

```
$ for level in 0 1 2 3; do gcc m.c -O$level -S -o m$level.s; done
```

Vom obtine astfel fisiere de forma mx.s, cu x de la 0 la 3. Pentru a verifica diferentele dintre doua astfel de fisiere, vom folosi programul „vimdiff".

```
$ vimdiff m1.s m2.s
```

Fisierele *.s contin codul in *assembler* al programelor compilate. Codul in assembler este o reprezentare „user-friendly" a codului masina (rezultat in urma compilarii). Se observa ca, pentru cod masina diferit, codul assembler este diferit. In cazul nostru, codul masina este diferit datorita optimizarilor introduse.

Nota: pentru a intelege ceva din codul afisat, trebuie intai sa faceti cursul de Programare in Limbaj de Asamblare :)

Programul „vimdiff" evidentiaza diferentele dintre 2 fisiere. Poate fi considerat un fel de viewer asociat programului „diff" pe care il vom studia mai tarziu. Iesirea se face cu comanda „:q" apelata de doua ori.

3. Biblioteci de functii in Linux

Bibliotecile de functii (en: libraries) au aparut pentru a oferi o mai mare flexibilitate dezvoltatorilor de software. Acestea reunesc functii des utilizate de catre mai programe, fara a fi necesar astfel rescrierea acestora pentru fiecare program in parte.

3.1. GNU Linker - ld

Trebuie sa amintim intai cum sunt utilizate aceste biblioteci de functii. Dupa cum se stie, programele sursa trec prin faza de compilare pentru a ajunge programe executabile. De cele mai multe ori, prin „compilare" se inteleg implicit de fapt doua procese:

1. compilare = trecerea din limbaj de programare in cod obiect
2. link-editare = editarea legaturilor cu bibliotecile de functii

La apelul unei comenzi precum

```
$ gcc <sursa> -Wall -o <fisier_executabil>
```

GCC-ul apeleaza implicit si linker-ul – ld. Pentru a inhiba acest comportament, se poate utiliza parametrul **-c**, in acest fel obtinandu-se doar un simplu fisier obiect. Putem folosi aceasta facilitate pentru a compila functiile unei aplicatii in fisiere obiect separate. Ulterior, putem compila aplicatia mare, incluzand fisierele obiect compilate anterior.

Nota: fisiere obiect obtinute prin compilarea cu parametrul **-c** nu sunt implicit biblioteci de functii.

Exemplu de apelare:

```
$ gcc -c fctp.c
$ gcc -c fctafis.c
$ gcc -c myprog.c
$ gcc myprog.o fctp.o fctafis.o -o hello
```

(aceste comenzi pot fi introduse intr-un *Makefile* – vezi restul laboratorului)

Continutul fisierelor se gaseste pe pagina urmatoare – „evident", programul rezultat nu face nimic :).

Observatii:

- Pentru a vedea ce comenzi apeleaza gcc-ul cand compileaza un program, utilizati parametrul**-v**
- Puteti vedea simbolurile fiecarui fisier obiect utilizand comanda **nm** – spre exemplu (pentru a intelege ce reprezinta U/T/etc., cititi **man nm**) :

```
$ nm fctafis.o
00000000 T afis
0000001b T endl
         U printf
```

Nume fisier: fctp.h

```
int mprod(int a, int b);
int mdiv(int a, int b);
```

Nume fisier: fctp.c

```
#include "fctp.h"

int mprod(int a, int b) { return a*b; }
int mdiv(int a, int b){ return ((b!=0)?a/b:a); }
```

Nume fisier: fafis.h

```
void afis(char *);
void endl();
```

Nume fisier: fafis.c

```
#include "fctafis.h"
#include <stdio.h>

void afis(char *s){ printf("Text afisat: %s\n",s);  }
void endl(){ printf("\n"); }
```

Nume fisier: myprog.c

```
#include "fctafis.h"
#include "fctp.h"

int main(){
        afis("hello there");
        return (mprod(10,2));
}
```

(pentru a intelege fisierul de mai jos, cititi intai partea a 2-a a laboratorului)

Nume fisier: Makefile

```
hello: fctp.h fctp.c fctafis.h fctafis.c myprog.c
        gcc -c fctp.c
        gcc -c fctafis.c
        gcc -c myprog.c
        gcc myprog.o fctp.o fctafis.o -o hello
clean:
        rm hello
```

3.2. Tipuri de biblioteci de functii

Exista 3 tipuri de biblioteci de functii:

- static libraries
- shared libraries
- dynamic libraries

Toate bibliotecile de functii exista sub forma de fisiere independente in system.

3.3. Localizarea bibliotecilor de functii

Bibliotecile statice, dat fiind utilitatea lor doar in procesul de compilare, se gasesc in general in **/usr/lib** sau in **/usr/local/lib** si au extensia **.a** (archive – vom vedea ulterior)

Bibliotecile partajate, avand in vedere ca sunt utile permanent in sistem, se gasesc in general in directorul **/lib** si au extensia **.so** (shared object).

3.4. Interactiunea dintre SO si bibliotecile de functii

- La compilare, **ld** cauta bibliotecile partajate in directoarele standard si in directoarele adaugate in linia de comanda prin intermediul unor parametrii speciali (**`-rpath dir`** si **`-Ldir`**).
- La rularea manuala (dupa update-ul bibliotecilor), **programul ldconfig** cauta in locatiile standard si in directoarele specificate in **/etc/ld.so.conf**, seteaza link-urile simbolice corecte in directoarele cu link-uri dinamice (pentru a urma standard-ul) si salveaza referintele catre biblioteci intr-un cache. Rularea programului ldconfig la fiecare bootare este ineficienta – acesta este un motiv in plus pentru utilizarea unui cache.
- Intr-un sistem bazat pe biblitecile standard Gnu C, inclusiv toate sistemele Linux, pornirea unui executabil ELF (formatul standard al executabilelor) determina incarcarea si rularea unui program loader. Pe sistemele linux, acesta este numit **/lib/ld-linux.so.X** (unde X este numarul versiuni). Acest loader gaseste si incarca bibliotecile partajate utilizate de catre program.
- Daca se doreste supraincarcarea unor functii din niste biblioteci dar cu pastrarea restului bibliotecii, se pot introduce bibliotecile supraincarcate in **/etc/ld.so.preload** – aceste biblioteci „preincarcate" vor avea intaietate relativ la setul standard. Acest fisier pentru preincarcari este utilizat de obicei pentru patch-uri de urgenta – o distrbutie nu include de obicei un astfel de fisier.

3.5. Informatii despre libraries

Informatii primare despre bibliotecile de functii le putem afla prin intermediul comenzii file. Mergeti intr-un director [cd, ls] cu biblioteci statice si intr-un director cu biblioteci partajate. Din fiecare director, alegeti cate un fisier si executati comanda file pentru acel fisier.

Biblioteci statice:

```
$ file /usr/lib/libm.a
/usr/lib/libm.a: current ar archive
```

Bibliotecile statice sunt de fapt arhive de fisiere obiect. Acest laborator nu trateaza crearea bibliotecilor statice.

Biblioteci dinamice:

Veti observa ca in directorul **/lib** spre exemplu sunt foarte multe fisiere cu extensia **.so** urmata de o serie de numere. Acesta este modul in care se realizeaza diferentierea versiunilor.

```
$ file /lib/libm.so.6
/lib/libm.so.6: symbolic link to `libm-2.3.3.so'

$ file /lib/libm-2.3.3.so
/lib/libm-2.3.3.so: ELF 32-bit LSB shared object, Intel 80386, version 1
(SYSV), not stripped
```

Utilizati programul **nm** pentru a vizualiza lista simbolurilor dintr-o biblioteca (statica sau dinamica). Prin simboluri se intelege denumirile date functiilor publice (functii ce pot fi utilizate de programe) din aceste biblioteci.

Exemplu:

```
$ nm /usr/lib/libm.a
...
```

3.6. Utilizarea bibliotecilor de functii

(Acest laborator nu trateaza utilizarea bibliotecilor de functii statice)

Dupa cum ati vazut anterior, am fortat link-uirea bibliotecii de functii standard C++ in programul nostru la apelarea compilatorului GCC cu:

```
$ gcc n.cpp -Wall -lstdc++ -o n
```

Similar, GCC-ul poate instrui ld-ul (pe care il apeleaza automat la detectia unui program cu functia main()) sa includa in procesul de link-editare si alte biblioteci partajate.

Scrieti urmatorul program in editorul preferat:

Program name: math.c

```
#include <math.h>

int main(){
        return sin(3);
}
```

Incercati sa il compilati in mod uzual, fara a forta link-uirea cu nici o biblioteca de functii.

```
$ gcc math.c -Wall -o myapp
/tmp/cczVPRTj.o(.text+0x26): In function `main':
: undefined reference to `sin'
collect2: ld returned 1 exit status
```

Daca va veti uita in math.h (**/usr/include/math.h**), veti vedea ca functia sin() este definita. Acest fisier header specifica faptul ca functiile sunt definite in biblioteca math (functii *externe*).

Pentru ca aceasta compilare sa se realizeze cu success, trebuie utilizata optiunea -lm („LM”):

```
$ gcc math.c -Wall -lm -o myapp
```

-lm ii spune compilatorului sa link-editeze si biblioteca matematica.

Un program/biblioteca partajata poate fi analizat(a) pentru a vedea care sunt dependentele din punct de vedere al bibliotecilor partajate utilizand comanda „ldd”

```
$ ldd /usr/lib/libm.so
[...]
```

Biblioteca de functii matematice depinde de biblioteca standard C.

```
$ ldd myapp
[...]
```

Se observa astfel cum aplicatia noastra depinde de libm.so, biblioteca de functii matematice.

```
$ ldd n
[...]
```

Programul nostru C++ depinde implicit de biblioteca de functii matematice, biblioteca standard C++ si biblioteca standard C.

4. Patch-uri

4.1. Diff

Utilitarul diff compara fisiere linie cu line si este astfel capabil sa detecteze schimbarile de continut intre doua fisiere.

Pentru exemplificare, vom folosi fisierele C si C++ utilizate anterior. Modul in care se apeleaza comanda diff este urmatorul:

```
$ diff <fisier1> <fisier2>
```

```
$ diff m.c n.cpp
```

Se observa la iesire cum anume trebuie modifica *fisieru1* (m.c) pentru ca in el sa se regaseasca ceea ce este in *fisierul2* (n.cpp).

Un parametru util in crearea diferentelor este parametrul **-u**. Acesta instruieste diff-ul sa afiseze diferentele in mod „unified context”. Acest lucru presupune si afisarea a X linii (X=3 by default) aflate inainte si X linii aflate dupa sectiunea modificata.

```
$ diff -u m.c n.cpp
```

Acest format permite patch-uirea cu mici decalaje (existenta unui *fuzz*), programul care se ocupa cu acest lucru detectand noua pozitie dupa liniile vecine care raman neschimbate.

De cele mai multe ori, rezultatul comenzii diff este pus intr-un fisier ce este referit ca „diff file"/„patch file" - extensia acestuia este de obicei *diff* sau *patch* (fara a avea vreo importanta pentru programe). Modul in care se salveaza rezultatul comenzii este:

```
$ diff -u m.c n.cpp > my.diff
```

Fisierele obtinute cu **-u** pot fi precedate de oricate linii de text, fara a interfera cu diff-ul propriu-zis, deoarece aceasta are un header specific. Acest lucru permite, spre exemplu, aplicarea diff-ului (=patch) cu tot cu e-mail-ul cu care este trimis (luand in considerare ca avem continutul e-mail-ului la inceput, atasamentul la sfarsit). Intreg e-mail-ul poate fi directionat catre programul patch pentru aplicarea modificarilor.

4.2. Patch

Patch este programul cu functia inversa programului diff. Aceasta aplica un diff asupra unui fisier.

Pentru exemplificare vom folosi initial un mod de apelare mai putin utilizat. Realizam intai o copie a programului C scris anterior.

```
$ cp m.c m2.c
$ patch m2.c my.diff
```

Verificand continutul lui m2.c vom descoperi ca acesta a fost patch-uit pentru a ajunge la versiunea programului in C++.

Similitudinea intre m2.c si n.cpp poate fi observata si folosind comanda diff

```
$ diff -u m2.c n.cpp

```

Nu va fi afisat nimic deoarece nu exista nici o diferenta.

In general, patch se utilizeaza sub forma:

```
$ patch -pX <patchfile
```

unde X este numarul de slash-uri ce trebuie indepartate din caile patch-urilor fisier (patchfile). Utilitatea lui -p se va vedea putin mai tarziu, cand vom analiza diff-ul dintre doua directoare.

Se observa faptul ca nu este indicat ce fisier trebuie patch-uit. Patch foloseste un algoritm de detectare a fisierului ce se doreste a fi patch-uit – algoritm descris in **man patch**. Daca nu se determina automat fisierul/fisierele, utilizatorul va fi chestionat in legatura cu fiecare patch.

Cea mai mica unitate a unui patch poarta numele de **hunk**. Programul patch va incerca aplicarea fiecarui hunk. Datorita schimbarilor majore in fisierul destinatie, anumite hunk-uri pot sa nu fie aplicate. In asemenea conditii se spune ca are loc un conflict. Conflictele se rezolva pot rezolva:

- manual (patch-uire manuala)
- prin resincronizarea sursei (de unde a provenit patch-ul) cu modificarile de la destinatie, dupa care sursa face un nou patch

4.3. Diferente intre directoare

Presupunem ca avem sursele unui program in directorul /src. Dorim sa efectuam modificari in copia curenta, dupa care aceste modificari dorim sa le trimitem altor programatori.

Pasii pe care ii vom urma sunt urmatorii:

```
$ cp -a src/ src-old/
# modificari in src/...
$ diff -uNr src-old/ src/ > src.patch
```

Am facut o copie iar in src/ am facut modificarile.
Folosind comanda diff, am realizat un patch cu toate modificarile fisierelor din directorul src/, relativ la versiunea veche salvata in src-old/

Un alt programator, care primeste fisierul .patch si doreste sa-si modifice sursele din src/, va utiliza comenzile:

```
$ cd src/
$ patch -up1 ../src.patch
```

Astfel, toate modificarile se vor reflecta in directorul celui de-al doilea programator.

Pentru mai multe informatii despre parametrii utilizati, consultati **man patch** / **man diff**.

5. GNU Make

Make este un utilitar care utilizeaza un script, numit *Makefile*, pentru a determina automat secventa de pasi ce trebuie repetati deoarece niste fisiere s-au modificat. In principal este utilizat pentru:

- recompilarea programelor compuse din mai multe fisiere
- testarea programelor.

5.1. Formatul unui Makefile

Instructiunile pentru make se gasesc intr-un Makefile/makefile si determina ce actiuni trebuie facute pentru a satisface anumite cerinte. Sintaxa unui makefile este de forma:

Makefile

```
Target... : dependencies ...
<tab>command
<tab>command
<tab>...
```

unde:

- target = numele unui fisier ce trebuie generat sau numele unei actiuni
- dependencies = lista de actiuni (fisiere) ce trebuie indeplinite (ce trebuie sa existe) pentru a se realiza target-ul – make determina daca trebuie re-executat target-ul daca una dintre dependete s-a modificat (unul din fisierele din lista de dependente s-a modificat)
- command = comanda ce duce la realizarea target-ului (de obicei la comenzi sunt trecute comenzile de compilare care duc la realizarea target-ului)
- <tab> = caracterul tab

Regulile formeaza un lant de dependente. Pentru ca un target sa fie facut, se verifica daca toate dependentele lui sunt facute. Daca nu este indeplinita aceasta conditie, se executa dependentele intr-un fel asemanator.

Pentru a exemplifica functionarea make-ului, introduceti urmatorul Makefile (atentie la TAB-uri):

Makefile

```
pregatire: mancare curatenie
        echo "Putem primi musafiri"
        touch pregatire
mancare: cumparaturi
        echo "Gata MANCARE"
        touch mancare
curatenie:
        echo "Gata CURATENIE"
        touch curatenie
cumparaturi:
        echo "Gata CUMPARATURI"
        touch cumparaturi
pierde_cumparaturi:
        rm cumparaturi
        echo "pierdut CUMPARATURI"
clean:
        rm cumparaturi
        rm curatenie
        rm mancare
        rm pregateste_musafiri
        echo "Gata ANULARE"
```

In acealasi director in care ati salvat fisierul Makefile, executati comenzile:

```
$ make mancare
[...]
```

Observatii:

- este executat si target-ul „cumparaturi” din lista de dependente pentru „mancare”.
- fiecare target se termina cu „touch numetarget” pentru ca `make` sa isi dea seama care target-uri au fost facute.
- make afiseaza atat comenzile din target-uri cat si rezultatul lor.

Daca incercam sa rulam target-ul cumparaturi...

```
$ make cumparaturi
make: `cumparaturi' is up to date.
```

... vom observa ca make detecteaza ca acesta a fost executat (detectia se face prin intermediul fisierului „cumparaturi” pe care il cream in target-ul „cumparaturi” la sfarsit, prin comanda „touch cumparaturi”.

Putem simula ca am pierdut cumparaturile...

```
$ make pierdut_cumparaturi
[...]
```

Putem face din nou cumparaturile...

```
$ make cumparaturi
[...]
```

Iar daca facem incercam sa facem mancare...

```
$ make mancare
[...]
```

... target-ul mancare va fi rulat din nou, cu toate ca il rulasem cu succes inainte. De ce s-a intamplat asta? Make si-a dat seama ca intre timp, am facut cumparaturi noi :)

La rularea target-ului „pregatire”...

```
$ make pregatire
[...]
```

... numai task-urile nefacute vor fi rulate.

De obicei, Makefile-urile au si target-uri care anuleaza toate celealte target-uri (in cazul compilarii programelor, spre exemplu, se sterg fisierele obiect) – in cazul nostru target-ul pentru anularea completa a modificarilor este „clean”.

Se poate testa si intregul pachet de target-uri legate de pregatire cu:

```
$ make clean
[...]

$ make pregatire
[...]
```

Vom crea acum un Makefile pentru compilarea programelor de mai sus (intr-un singur director numai un fisier **Makefile** poate exista):

Makefile

```
Progc: m.c
        gcc m.c -Wall -o progc
progcpp: n.cpp
        g++ n.cpp -Wall -o progcpp
all: progc progcpp
clean:
        rm progc
        rm progcpp
```

Observati ca am dat ca nume de target-uri exact numele executabilelor obtinute in urma compilarii. Acest lucru permite make-ului sa detecteze cand un target a fost facut deja si sa il sara in eventualitatea in care este prezent ca o dependenta.

Executati (pentru a vedea cum se comporta):

```
$ make progcpp
...
$ make all
...
$ make clean
...
$ make all
...
```

Pentru o flexibilitate mai mare in realizarea target-urilor, vom modifica acum Makefile-ul de mai sus prin introducerea unor variabile. Makefile-ul rezultat va arata in felul urmator:

Makefile

```
gccopt=-Wall
optimizare=-O2
ccompiler=gcc
cppcompiler=g++

progc:
        $(ccompiler) m.c $(gccopt) $(optimizare) -o progc
progcpp:
        $(cppcompiler) n.cpp $(gccopt) $(optimizare) -o progcpp
all: progc progcpp
clean:
        rm progc
        rm progcpp
```

Observati ca:

- am scos optiunea „-Wall” intr-o variabila separata
- am adaugat optiunea „optimizare” tot ca o variabila
- compilatoarele le-am trecut in variabile separate

Principalul avantaj pe care il aduce utilizarea variabilelor este flexibilitatea crescuta in realizarea si modificarea comenzilor de compilare pentru diferite conditii.

Programele care vin ca surse se distrbuie in general cu un script de configurare, numit „configure”. Acesta are ca scop stabilirea unor variabile ce sunt introduse intr-un *Makefile*. Aceste variabile pot seta diferite lucruri, precum compilatorul ce va fi folosit, bibliotecile ce vor fi incluse etc.

5.2. Note informative

Make functioneaza si fara existenta unui *Makefile*:

- **`make myprog`** # va compila myprog.c
- **`make CFLAGS=-Wall myprog`** # va compila myprog.c cu compile flags: -Wall

Pentru o intretinere mai usoara a *Makefile*-urilor, puteti studia cum se utilizeaza regulile sablon (pattern rules) cu make.

6. Analiza system-call-urilor si a semnalelor

6.1. Utilizarea lui strace

Programul strace este util in urmarirea system-call-urilor efectuate la executia unui program. Aceasta analiza reprezinta o metoda destul de eficienta in debugging-ul programelor al caror cod sursa nu-l avem.

Sa analizam output-ul programului pentru o aplicatie simpla:

```
$ strace echo "USO"
execve("/bin/echo", ["echo", "USO"], [/* 51 vars */]) = 0
uname({sys="Linux", node="aquarium", ...}) = 0
[...]
access("/etc/ld.so.preload", R_OK)      = -1 ENOENT (No such file or directory)
open("/etc/ld.so.cache", O_RDONLY)      = 3
fstat64(3, {st_mode=S_IFREG|0644, st_size=141679, ...}) = 0
[...]
close(3)                                = 0
open("/lib/tls/libc.so.6", O_RDONLY)    = 3
read(3, "\177ELF\1\1\1\0\0\0\0\0\0\0\0\0\3\0\3\0\1\0\0\0\220R\1"..., 512) = 512
fstat64(3, {st_mode=S_IFREG|0755, st_size=1363203, ...}) = 0
[...]
close(3)                                = 0
[...]
fstat64(1, {st_mode=S_IFCHR|0600, st_rdev=makedev(136, 1), ...}) = 0
[...]
write(1, "USO\n", 4USO
)                       = 4
[...]
exit_group(0)                           = ?
```

Ce observam?

- primul syscall (execve) ne indica ce parametrii a primit programul nostru: „echo” si „USO”
- programul nostru a incercat sa deschida fisierul /etc/ld.so.preload dar nu a fost gasit (vezi descrierea de la biblioteci de fisiere)
- syscall-ul „open” a returnat valoarea 3 – aceasta reprezinta un file descriptor (fd) – file descriptor-ul este folosit in procesele de citire si in procesele de scriere
- syscall-ul „close” are ca singur parametru file descriptor-ul ce trebuie inchis – o data inchis, un file descriptor poate fi refolosit
- syscall-ul „read” prezent mai jos indica faptul ca s-a citit din file descriptor-ul 3
- syscall-ul „write” are 3 parametrii:
  - 1 = file descriptor-ul unde se scrie: in cazul nostru, dupa cum stim, 1 este file descriptor-ul pentru stdout (iesirea standard)
  - „USO\n” = acesta este textul ce trebuie afisat de catre comanda „echo”
  - 4 = lungimea textului ce trebuie afisat
- Nu este nimic in neregula cu acest program. A functionat corect si am putut identifica cateva system call-uri frecvente. Sa incercam acum programul strace pe o aplicatie mai putin „cooperanta”. Sa construim chiar noi aceasta aplicatie, cu toate ca este posibil sa intalnim chiar practic o asemenea situatie... cand totul se intrerupe brusc.

Programul de test pe care il vom folosi este urmatorul:

testfile.c

```
#include <stdio.h>

int main()
{
        FILE *f=fopen("fisier_nou.txt","r");
        fclose(f);
        return 0;
}
```

Pare a fi ceva in neregula? Ne prefacem ca nu stim :)

Compilam fisierul cu make, asa cum am invatat anterior, fara a utiliza un Makefile:

```
$ make testfile
cc     testfile.c   -o testfile
```

... sau... mai bine...

```
$ rm testfile
$ make CFLAGS=-Wall testfile
cc -Wall    testfile.c   -o testfile
```

Am reusit astfel sa introducem in procesul de compilare un parametru ce ne poate ajuta sa eliminam o parte din probeme. Din fericire, nu au aparut warning-uri.

Incercam sa rulam programul tocmai compilat...

```
$ ./testfile
Segmentation fault (core dumped)
```

(mesajul afisat poate sa difere: poate sa lipseasca „core dumped”)

Se pare ca rezultatul nu este cel dorit. (Wikipedia:) Aceasta este o eroare ce apare cand programul incearca sa apeleze o locatie de memorie pentru care nu are drepturi de access, sau acceseaza o locatie de memorie intr-un fel incorect (ex: incearca sa scrie o zona care este read-only).

Ignorand faptul ca noi avem sursa, exact aceasta este situatia in care un program mai putin grijuliu va poate pune. Nu stiti despre ce este vorba, nu aveti sursa si nu merge. La prima vedere poate parea ingrijorator, dar strace poate fi de ajutor.

Iata cum:

```
$ strace ./testfile
execve("./testfile", ["./testfile"], [/* 51 vars */]) = 0
uname({sys="Linux", node="aquarium", ...}) = 0
[...]
open("fisier_nou.txt", O_RDONLY)        = -1 ENOENT (No such file or directory)
--- SIGSEGV (Segmentation fault) @ 0 (0) ---
+++ killed by SIGSEGV (core dumped) +++
```

Ce sa vezi? A incercat sa deschida un fisier si s-a blocat. Intr-adevar, programul poate fi imbunatatit :)

Sa incercam sa-l executam dupa ce cream manual fisierul pe care il cauta:

```
$ touch fisier_nou.txt
$ ./testfile
```

Nici o problema. Programul s-a executat fara a se bloca. Strace confirma acest lucru:

```
$ strace ./testfile
execve("./testfile", ["./testfile"], [/* 51 vars */]) = 0
uname({sys="Linux", node="aquarium", ...}) = 0
[...]
open("fisier_nou.txt", O_RDONLY)        = 3
close(3)                                = 0
exit_group(0)                           = ?
```

Pentru ca avem sursa, putem chiar incerca sa reparam problema prin adaugarea unei linii de test a existentei fisierului (de fapt, de verificare a unui pointer):

testfile.c

```
#include <stdio.h>

int main()
{
        FILE *f=fopen("fisier_nou.txt","r");
        if (!f) return 1;
        fclose(f);
        return 0;
}
```

Sa si rulam programul nou:

```
$ rm testfile
$ make CFLAGS=-Wall testfile
cc -Wall    testfile.c   -o testfile
$ rm fisier_nou.txt
$ ./testfile
```

Nici o problema.

Sa vedem cum interpreteaza strace aceasta rezolvare:

```
$ strace ./testfile
execve("./testfile", ["./testfile"], [/* 51 vars */]) = 0
uname({sys="Linux", node="aquarium", ...}) = 0
[...]
open("fisier_nou.txt", O_RDONLY)        = -1 ENOENT (No such file or directory)
exit_group(1)                           = ?
```

Exact cum ne-am asteptat. Programul nu gaseste fisierul „fisier_nou.txt” si iese returnand 1 (dupa cum indica si system call-ul „exit_group”).