# Introducere

## Contents

# Introducere

Laboratoarele de Sisteme de Operare au drept scop aprofundarea conceptelor prezentate la curs şi prezentarea interfețelor de programare oferite de sistemele de operare (system API). Laboratorul este un laborator de system programming. Un laborator va aborda un set de concepte şi va conține un scurt breviar teoretic, o prezentare a API-ului asociat cu explicații şi exemple şi un set de exerciții pentru acomodarea cu acesta. Pentru a oferi o arie de cuprindere cât mai largă, laboratoarele au ca suport familiile de sisteme de operare Unix şi Windows. Instanțele de sisteme de operare din familiile de mai sus alese pentru acest laborator sunt **GNU/Linux**, respectiv **Windows XP Service Pack 2**.

În cadrul acestui laborator (laboratorul de introducere), va fi prezentat mediului de lucru care va fi folosit în cadrul laboratorului de Sisteme de Operare. Laboratorul foloseşte ca suport de programare limbajul C/C++. Pentru GNU/Linux se va folosi suita de compilatoare GCC, iar pentru Windows compilatorul Microsoft pentru C/C++ `cl`. De asemenea, pentru compilarea incrementală a surselor se vor folosi GNU make (Linux), respectiv nmake (Windows). Exceptând apelurile de bibliotecă standard, API-ul folosit va fi POSIX, respectiv Win32.

# Linux

## GCC

GCC este suita de compilatoare implicită pe majoritatea distribuţiilor Linux. GCC este unul din primele pachete software dezvoltate de organizația "Free Software Fundation" în cadrul proiectului GNU (Gnu's Not Unix). Proiectul GNU a fost inițiat ca un protest împotriva software-ului proprietar de Richard Stallman la începutul anilor '80.

La început, GCC se traducea prin "GNU C Compiler", pentru ca inițial scopul proiectului GCC era dezvoltarea unui compilator C portabil pe platforme UNIX. Ulterior, proiectul a evoluat astăzi fiind un compilator multi-frontend, multi-backend cu suport pentru limbajele C, C++, Objective-C, Fortran, Java, Ada. Drept urmare, acronimul GCC înseamnă, astăzi, "GNU Compiler Collection".

La numărul impresionant de limbaje de mai sus se adaugă şi numărul mare de platforme suportate atât din punctul de vedere al arhitecturii hardware (i386, alpha, vax, m68k, sparc, HPPA, arm, MIPS, PowerPC, etc.) cât şi al sistemelor de operare (GNU/Linux, DOS, Windows 9x/NT/2000, *BSD, Solaris, Tru64, VMS, etc.). La ora actuală, GCC-ul este cel mai portat compilator.

În cadrul laboratoarelor de Sisteme de Operare ne vom concentra asupra facilităţilor oferite de compilator pentru limbajul C/C++. GCC suportă stadardele ANSI, ISO C, ISO C99, POSIX dar şi multe extensii folositoare care nu sunt incluse în niciunul din standarde; unele din aceste extensii vor fi prezentate în secțiunile ce urmează.

### Utilizare GCC

O vedere de ansamblu asupra procesului de compilare este prezentata in imaginea de mai jos.

GCC foloseşte pentru compilarea de programe C/C++ comanda gcc, respectiv g++. O invocare tipică este pentru compilarea unui program dintr-un singur fişier sursă:

```
$ gcc hello.c
$ ./a.out
Hello, world!
```

Comanda `gcc hello.c` a fost folosită pentru compilarea fişierului sursă hello.c. Rezultatul a fost obținerea executabilului `a.out` care a fost rulat.

Executabilul `a.out` este executabilul implicit obținut de `gcc`. Dacă se doreşte obținerea unui executabil cu alt nume se poate folosi opțiunea `-o`:

```
$ gcc hello.c -o hello
$ ./hello
Hello, world!
```

Comanda de mai sus a produs executabilul `hello`.

La fel se poate folosi `g++` pentru compilarea unui fişier sursă C++:

```
$ g++ hello.cpp -o hello_cpp
$ ./hello_cpp
Hello, world!
```

## Opțiuni

După cum s-a observat, la o rulare a comenzii `gcc`/`g++` se obține din fişierul sursă un executabil. Folosind diverse opțiuni, putem opri compilarea la una din fazele intermediare astfel:

- -E - se realizează doar preprocesarea fişierului sursă
  - gcc -E hello.c , se va obtine fisierul preprocesat `hello.i`
- -S - se realizează inclusiv faza de compilare
  - gcc -S hello.c , se va obtine fisierul in limbaj de asamblare `hello.s`
- -c - se realizează inclusiv faza de asamblare
  - gcc -c hello.c , se va obtine fisierul obiect `hello.o`

La opțiunile de mai sus se poate folosi opțiunea -o pentru specificarea fişierului de ieşire:

```
$ gcc -c hello.c -o my_obj_hello.o
```

### Activarea avertismentelor (warnings)

În mod implicit, o rulare a gcc oferă puține avertismente utilizatorului. Pentru a activa afişarea de avertismente se foloseşte opțiunea -W cu sintaxa `-Woptiune-warning`. `optiune-warning` poate lua mai multe valori posibile printre care `return-type`, `switch`, `unused-variable`, `uninitialized`, `implicit`, `all`. Folosirea opțiunii `-Wall` înseamnă afişarea tuturor avertismentelor care pot cauza inconsistențe la rulare.

Considerăm ca fiind indispensabilă folosirea opțiunii `-Wall` pentru a putea detecta încă din momentul compilării posibilele erori. O cauză importantă a aparițiilor acestor erori o constituie sintaxa foarte permisivă a limbajului C. Sperăm ca exemplul de mai jos să justifice utilitatea folosirii opțiunii `-Wall`:

**Exemplu 1. intro-01.c**

```
#include <stdio.h>

 int main()
 {
printf("1 + 2 fac %d\n", suma(1, 2));
 }

 int suma(int a, int b, int c)
 {
return a + b + c;
 }
```

În exemplul mai sus, programatorul a uitat că funcția definită de el pentru adunare primeşte trei parametri şi nu doi. Dacă programul se compilează fără opțiunea -Wall, nu se vor genera erori sau avertismente, dar rezultatul nu va fi cel aşteptat:

```
$ gcc intro-01.c
$ ./a.out
1+2 fac -1073743413
```

Prgramul s-a compilat fără erori, pentru că funcția `suma` a fost declarată implicit de compilator (în C, în mod normal, funcțiile trebuie să fie declarate înainte de a fi folosite). O funcție declarat implicit are prototipul:

```
int function(...);
```

În prototipul de mai sus se poate recunoaşte operatorul `...` (se citeşte *elipses*) care precizează faptul că funcția are un număr variabil de parametri. Dacă se compilează acelaşi program folosind optiunea `-Wall`, programatorul va avea cel puțin ocazia să afle că funcția a fost declarată implicit (şi, în cazul de față, şi faptul că a uitat să întoarcă un rezultat din funcția `main`):

```
$ gcc intro-01.c -Wall
exemplul-1.c: In function `main':
exemplul-1.c:5: warning: implicit declaration of function `suma'
exemplul-1.c:6: warning: control reaches end of non-void function
```

Soluția este crearea unei declarații pentru funcția `suma` şi apelul corespunzător al acesteia:

**Exemplu 2. intro-02.c**

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

```
 int suma(int a, int b, int c);

 int main()
 {
printf("1 + 2 fac %d\n", suma(1, 2, 0));
return 0;
 }

 int suma(int a, int b, int c)
 {
return a + b + c;
 }
```

```
$ gcc -Wall intro-02.c
$ ./a.out
1 + 2 fac 3
```

Exemplul prezentat oferă doar una din erorile posibile pe care GCC le detectează atunci când se foloseşte optiunea `-Wall`. În concluzie, **folositi** opțiunea `-Wall`. În aceeaşi categorie, mai exista opțiunea `-Wextra` (echivalent cu opțiunea `-W`), mult mai agresivă.

### Alte opțiuni

Alte opțiuni utile sunt:

- -Lcale - aceast optiune instruieşte compilatorul s caute si în directorul cale bibliotecile pe care trebuie s le foloseasc programul; opțiunea se poate specifica de mai multe ori, pentru a aduga mai multe directoare
- -lbiblioteca - instruieşte compilatorul c programul are nevoie de biblioteca *biblioteca*. Fişierul ce conține biblioteca va fi denumit libbiblioteca.so sau libbiblioteca.a.
- -Icale - instruieşte compilatorul s caute fişierele antet (headere) şi în directorul *cale*; opțiunea se poate specifica de mai multe ori, pentru a aduga mai multe directoare
- -Onivel-optimizari - instuieşte compilatorul ce nivel de optimizare trebuie aplicat; *-O0* va determina compilatorul s nu optimizeze codul generat; *-O3* va determina compilatorul s optimizeze la maxim codul generat; *-O2* este pragul de unde compilatorul va începe s insereze direct în cod functiile inline în loc s le apeleze; *-Os* va optimiza programul pentru a reduce dimensiunea codului generat, si nu pentru viteză.
- -g - dac se foloseşte aceast opțiune compilatorul va genera în fişierele de ieşire informații care pot fi apoi folosite de un debugger (informații despre fişierele surs şi o mapare între codul maşin şi liniile de cod ale fişierelor surs)

Paginile de ajutor ale GCC (man gcc, info gcc) oferă o listă cu toate opțiunile posibile ale GCC.

## Compilarea din mai multe fişiere

Exemplele de până acum tratează programe scrise într-un singur fişier sursă. În realitate, aplicațiile sunt complexe şi scrierea întregului cod într-un singur fişier îl face greu de menținut şi greu de extins. În acest sens aplicația este scrisă în mai multe fişiere sursă denumite module. Un modul conține, în mod obişnuit, funcții care îndeplinesc un rol comun.

Următoarele fişiere sunt folosite ca suport pentru a exemplifica modul de compilare a unui program provenind din mai multe fişiere sursă:

**Exemplu 3. intro-03-main.c**

```
#include <stdio.h>
#include "intro-03-util.h"

 int main(void)
 {
  ();   f1
  ();   f2

return 0;
 }
```

**Exemplu 3. intro-03-util.h**

```
#ifndef _INTRO_03_UTIL_H
#define _INTRO_03_UTIL_H      1

 void f1(void);
 void f2(void);

 #endif
```

**Exemplu 3. intro-03-f1.c**

```
#include "intro-03-util.h"
#include <stdio.h>

 void f1(void)
 {
printf("Fisierul curent este %s\n", __FILE__);
 }
```

**Exemplu 3. intro-03-f2.c**

```
#include "intro-03-util.h"
#include <stdio.h>

 void f2(void)
 {
printf("Va aflati la linia %d din fisierul %s\n", __LINE__, __FILE__);
 }
```

În programul de mai sus se apelează, respectiv, funcțiile `f1` şi `f2` în funcția `main` pentru a afişa diverse informații. Pentru compilarea acestora se transmit toate fişierele C ca argumente comenzii `gcc`:

```
$ gcc -Wall intro-03-main.c intro-03-f1.c intro-03-f2.c -o intro-03
$ ./intro-03
Fisierul curent este intro-03-f1.c
Va aflati la linia 5 din fisierul intro-03-f2.c
```

Executabilul de ieşire a fost denumit intro-03; pentru acest lucru s-a folosit opțiunea `-o`.

Se observă folosirea fişierului header intro-03-util.h pentru declararea funcţiilor f1 şi f2. Declararea unei funcţii se realizează prin precizarea antetului. Fişierul header este inclus în fişierul intro-03-main.c pentru ca acesta să aibă cunoştinţă de formatul de apel al funcţiilor f1 şi f2. Funcţiile f1 şi f2 sunt definite, respectiv, în fişierele intro-03-f1 şi intro-03-f2. Codul acestora este integrat în executabil în momentul link-editării.

În general în obţinerea unui executabil din surse multiple se obişnuieşte compilarea fiecărei surse până la modulul obiect şi apoi link-editarea acestora:

```
$ gcc -Wall -c intro-03-f1.c
$ gcc -Wall -c intro-03-f2.c
$ gcc -Wall -c intro-03-main.c
$ gcc intro-03-f1.o intro-03-f2.o intro-03-main.o -o intro-03-m
$ ./intro-03-m
Fisierul curent este intro-03-f1.c
Va aflati la linia 5 din fisierul intro-03-f2.c
```

Se observă obţinerea executabilului intro-03-m prin legarea modulelor obiect. Această abordare are avantajul eficienţei. Dacă se modifică fişierul sursă intro-03-f2 atunci doar acesta va trebui compilat şi refăcută link-editarea. Dacă s-ar fi obţinut un executabil direct din surse atunci s-ar fi compilat toate cele trei fişiere şi apoi refăcută link-editarea. Timpul consumat ar fi mult mai mare, în special în perioada de dezvoltare când fazele de compilare sunt dese şi se doreşte compilarea doar a fişierelor sursă modificate.

Scăderea timpului de dezvoltare prin compilarea numai a surselor care au fost modificate este motivaţia de bază pentru existenţa utilitarelor de automatizare precum make sau nmake.

Un lucru important în utilizarea header-elor pentru aplicaţii cu mai multe fişiere este folosirea directivelor de procesare #ifndef, #define, #endif prezentate în secţiunea următoare. Un fişier header tipic va avea structura:

```
#ifndef _NUME_HEADER_H    /* numele fisierului header scris cu majuscule */
#define _NUME_HEADER_H  1

 /* includere de alte header-e */
 /* macrodefiniţii */
 /* declaraţii de structuri */
 /* declaraţii de funcţii */

#endif
```

Aceste directive de preprocesare au rolul de a proteja declaraţiile din header în cazul în care acesta este inclus de mai multe ori. Astfel, la prima includere nu va fi definit _NUME_HEADER_H (#ifndef), drept pentru care se defineşte _NUME_HEADER_H (#define) şi se prelucrează diversele declaraţii. La următoarea includere _NUME_HEADER_H va fi deja definit (#ifndef) şi nu va mai fi prelucrată partea de declaraţii, evitându-se astfel generarea unor erori de genul "multiple declaration". De remarcat că, pentru fişiere antet diferite este necesar ca simbolurile declarate la început, după modelul de mai sus, să fie diferite; altfel este posibil ca declaraţiile din al doilea antet protejat să fie în mod eronat omise după preprocesare.

Directivele de preprocesare __FILE__ şi __LINE__ sunt expandate de preprocesor la numele fişierului, respectiv numărul liniei.

# Preprocesorul. Opţiuni de preprocesare

Preprocesorul este prima componentă apelată în momentul folosirii comenzii gcc. Preprocesorul pe distribuţiile Linux este GNU CPP. După CPP se apelează compilatorul efectiv (GCC), apoi asamblorul (GAS) şi apoi linker-ul (GNU LD). Rolul CPP este acela de prelucrare a directivelor şi a operatorilor de preprocesare.

Directivele de preprocesare cele mai întâlnite sunt:

- #include pentru includerea de fişiere (de obicei header) într-un alt fişier
- #define, #undef folosite pentru definirea, respectiv anularea definirii de macrouri

```
#define MY_MACRO       1                    /* macro simplu */
#undef MY_MACRO

 /* macro cu parametri; parantezele sunt importante! */
#define my_macro(a,b)  ((a) + (b))

#define my_func_substit(a,b,c)             \  /* macro substituent de funcţie */
      do {                                 \
              int i;                       \
                                           \
              c = 0;                       \
              for (i = a; i < b; i++)      \
                      c += i;              \
      } while (0)                          \
```

- #if, #ifdef, #ifndef, #else, #elif, #endif folosite pentru compilare condiţionată

```
#define ON        1
#define OFF       0
#define DEBUG     ON

#if DEBUG == ON
 /* C code ... do stuff */
#else
 /* C code ... do some other stuff */
#endif
```

- __FILE__, __LINE__, __func__ sunt înlocuite cu numele fişierului, linia curentă în fişier şi numele funcţiei
- operatorul # este folosit pentru a înlocui o variabilă transmisă unui macro cu numele acesteia

**Exemplu 4. intro-04.c**

```
#include <stdio.h>

#define expand_macro(a) printf ("variabila %s are valoarea %d\n", #a, a)

 int main (void)
 {
         int my_uber_var = 12345;

         expand_macro (my_uber_var);

         return 0;
 }
```

```
$ gcc -Wall intro-04.c
$ ./a.out
variabila my_uber_var are valoarea 12345
```

- operatorul ## (token paste) este folosit pentru concatenarea între un argument al macrodefiniției şi un alt şir de caractere sau între două argumente ale macrodefiniției.

## Opțiuni pentru preprocesor la apelul gcc

Preprocesorului îi pot fi transmise opțiuni prin parametri transmişi comenzii gcc. Pentru aceasta se pot utiliza opțiunile -I sau -D.

Opțiunea -I este utilă pentru a preciza locul în care se află fişierele incluse. Astfel, daca fişierul header `utils.h` se află în directorul `includes/`, utilizatorul poate include fişierul în forma

```
#include "utils.h"
```

dar va trebui să precizeze calea către fişier folosind opțiunea -I:

```
$ gcc -Iincludes [...]
```

Opțiunea -D este utilă pentru a defini macrouri în linia de comandă:

```
$ gcc -D __DEBUG__ [...]
$ gcc -D SIMPLE_MACRO=10 [...]           ; echivalent cu #define SIMPLE_MACRO    10
```

Opțiunea -U este utilă pentru a anula definirea unui macro.

## Debugging folosind directive de preprocesare

De multe ori, un dezvoltator va dori să poată activa sau dezactiva foarte facil afişarea de mesaje suplimentare (de informare sau de debug) în sursele sale. Metoda cea mai simplă pentru a realiza acest lucru este prin intermediul unui macro:

```
#define DEBUG      1

#ifdef DEBUG
 /* afisare mesaje debug */
#endif
```

Dacă se foloseşte opțiunea -D în linia de comandă, atunci definiția macroului DEBUG poate fi eliminată:

```
$ gcc -DDEBUG [...]
```

Folosirea perechii de directive #ifdef, #endif prezintă dezavantajul încărcării codului. Se poate încerca modularizarea afişării mesajelor de debug printr-o construcție de forma:

```
#ifdef DEBUG
#define Dprintf(msg)  printf(msg)
#else
#define Dprintf(msg)                /* do nothing */
#endif
```

În momentul de față problema este folosirea mai multor argumente la printf. Acest lucru poate fi rezolvat prin intermediul macrourilor cu număr variabil de parametri sau variadic macros, apărute în standardul ISO C99:

```
#ifdef DEBUG
#define Dprintf(msg,...)  printf(msg, __VA_ARGS__)
#else
#define Dprintf(msg,...)                /* do nothing */
#endif
```

Singura problema care mai poate apărea este folosirea Dprintf cu un singur argument. În acest caz macroul se expandează la printf (msg,), expresie invalidă în C. Pentru a elimina acest incovenient se foloseşte operatorul ##. Dacă acesta este folosit peste un argument care nu există, atunci virgula se elimină şi expresia devine corectă. Acest lucru nu se întâmplă în cazul în care argumentul există (altfel spus operatorul ## nu schimbă sensul de până atunci):

```
#ifdef DEBUG
#define Dprintf(msg,...)  printf(msg, ##__VA_ARGS__)
#else
#define Dprintf(msg,...)               /* do nothing */
#endif
```

Un ultim retuş este afişarea, dacă se doreşte, a fişierului şi liniei unde s-a apelat macroul:

```
#ifdef DEBUG
#define Dprintf(msg,...)  printf("[%s]:%d", msg, __FILE__, __LINE__, ##__VA_ARGS__)
#else
#define Dprintf(msg,...)                /* do nothing */
#endif
```

# Linker-ul. Opțiuni de link-editare. Biblioteci

Linker-ul este folosit pentru a "unifica" mai multe module obiect şi biblioteci şi a obține un executabil sau o bibliotecă. Linker-ul are rolul de a rezolva simbolurile nedefinite dintr-un modul obiect prin inspectarea celor existente într-un altul. Erorile de linker apar ca urmare a lipsei unui simbol, ca în exemplul de mai jos:

**Exemplu 5. intro-05-main.c**

```
void f(void);

 int main (void)
 {
         f();

         return 0;
 }
```

**Exemplu 5. intro-05-f.c**

```
#include <stdio.h>

 void f(void)
 {
         printf ("Hello, World!\n");
 }
```

```
$ gcc -Wall intro-05-main.c
/tmp/ccVBU35X.o: In function `main':
intro-05-main.c:(.text+0x12): undefined reference to `f'
collect2: ld returned 1 exit status
$ gcc -Wall intro-05-main.c intro-05-f.c
$ ./a.out
Hello, World!
```

La o primă rulare a apărut eroare pentru că linker-ul nu a găsit funcția f. Includerea intro-05-f.c în lista de fişiere compilate a rezolvat această problemă.

Linker-ul pe distribuțiile Linux este GNU LD. Executabilul asociat este ld. De obicei comanda gcc apelează în spate ld pentru a efectua link-editarea modulelor obiect. Opțiunile de linking sunt de obicei transmise comenzii gcc. Opțiunile cele mai utile sunt cele care sunt legate de biblioteci.

## Biblioteci

O bibliotecă este o colecție de funcții precompilate. În momentul în care un program are nevoie de o funcție, linker-ul va apela respectiva funcție din bibliotecă. Numele fişierului reprezentând biblioteca trebuie să aibă prefixul lib:

```
$ ls -l /usr/lib/libm.*
-rw-r--r-- 1 root root 481574 Jul 30 23:41 /usr/lib/libm.a
lrwxrwxrwx 1 root root     14 Aug 25 20:20 /usr/lib/libm.so -> /lib/libm.so.6
```

Biblioteca matematică este denumită libm.a sau libm.so. În Linux bibliotecile sunt de două tipuri:

- statice, au de obicei, extensia .a
- dinamice, au extensia .so

Detalii despre crearea bibliotecilor se găsesc în secțiunea următoare.

Legarea se face folosind opțiunea -l transmisă comenzii gcc. Astfel, dacă se doreşte folosirea unor funcții din math.h, trebuie legată biblioteca matematică:

**Exemplu 6. intro-06.c**

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

 #define RAD     (M_PI / 4)

 int main (void)
 {
         printf ("sin = %g, cos = %g\n", sin (RAD), cos (RAD));

         return 0;
 }
```

```
$ gcc -Wall intro-06.c
/tmp/ccRqG57V.o: In function `main':
intro-06.c:(.text+0x1b): undefined reference to `cos'
intro-06.c:(.text+0x2c): undefined reference to `sin'
collect2: ld returned 1 exit status
```

```
$ gcc -Wall intro-06.c -lm
$ ./a.out
sin = 0.707107, cos = 0.707107
```

Se observă că, în primă fază, nu s-au rezolvat simbolurile cos şi sin. După legarea bibliotecii matematice, programul s-a compilat şi a rulat fără probleme.

## Crearea de biblioteci

Pentru crearea de biblioteci vom folosi exemplul 3. Vom include modulele obiect rezultate din fişierele sursă intro-03-f1.c şi intro-03-f2.c într-o bibliotecă pe care o vom folosi ulterior pentru obținerea executabilului final.

Primul pas constă în obținerea modulelor obiect asociate:

```
$ gcc -Wall -c intro-03-f1.c
$ gcc -Wall -c intro-03-f2.c
```

### Crearea unei biblioteci statice

O biblioteca statica este o arhiva ce contine fisiere obiect creata cu ajutorul utilitarului ar.

```
$ ar rc libintro.a intro-03-f1.o intro-03-f2.o
$ gcc -Wall intro-03-main.c -o intro-lib -lintro
/usr/bin/ld: cannot find -lintro
collect2: ld returned 1 exit status
```

Aruncati o privire in pagina de manual a utilitarului ar si interpretati parametrii rc de mai sus.

Linker-ul returnează eroare precizând că nu găseşte biblioteca libintro. Aceasta deoarece linker-ul nu a fost configurat să caute şi în directorul curent. Pentru aceasta se foloseşte opțiunea -L, urmată de directorul în care trebuie căutată biblioteca (în cazul nostru este vorba de directorul curent):

```
$ gcc -Wall intro-03-main.c -o intro-lib -lintro -L.
$ ./intro-lib
Fisierul curent este intro-03-f1.c
Va aflati la linia 5 din fisierul intro-03-f2.c
```

### Crearea unei biblioteci partajate

Spre deosebire de o biblioteca statica despre care am vazut ca nu este nimic altceva decat o arhiva de fisiere obiect, o biblioteca partajata este ea insasi un fisier obiect. Crearea unei biblioteci partajate se realizează prin intermediul linker-ului. Optiunea -shared indica compilatorului sa creeze un obiect partajat si nu un fisier executabil. Este, de asemenea, indicată folosirea opțiunii -fPIC:

```
$ gcc -shared -fPIC intro-03-f1.o intro-03-f2.o -o libintro_shared.so
$ gcc -Wall intro-03-main.c -o intro-lib -lintro_shared -L.
$ ./intro-lib
./intro-lib: error while loading shared libraries: libintro_shared.so:
     cannot open shared object  file: No such file or directory
```

La rularea executabilului se poate observa că nu se poate încărca biblioteca partajată. Cauza este deosebirea dintre bibliotecile statice şi bibliotecile partajate. În cazul bibliotecilor statice codul funcției de bibliotecă este copiat în codul executabil la link-editare. De partea cealaltă, în cazul bibliotecilor partajate, codul este încărcat în memorie în momentul rulării.

Astfel, în momentul rulării unui program, loader-ul (programul responsabil cu încărcarea programului în memorie), trebuie să ştie unde să caute biblioteca partajată pentru a o încărca în memorie în cazul în care aceasta nu a fost încărcată deja. Loader-ul foloseşte câteva căi predefinite (**/lib**, **/usr/lib**, etc) şi de asemenea locații definite în variabila de mediu LD_LIBRARY_PATH:

```
$ export LD_LIBRARY_PATH=$LD_LIBRARY_PATH:.
$ ./intro-lib
Fisierul curent este intro-03-f1.c
Va aflati la linia 5 din fisierul intro-03-f2.c
```

În exemplul de mai sus variabilei de mediu LD_LIBRARY_PATH i-a fost adăugată calea către directorul curent rezultând în posibilitatea rulării programului. LD_LIBRARY_PATH va rămâne modificată cât timp va rula consola curentă. Pentru a face o modificare a unei variabile de mediu doar pentru o instană a unui program se face atribuirea noii valori înaintea comenzii de execuie:

```
$ LD_LIBRARY_PATH=$LD_LIBRARY_PATH:.  ./intro-lib
Fisierul curent este intro-03-f1.c
Va aflati la linia 5 din fisierul intro-03-f2.c
$ ./intro-lib
./intro-lib: error while loading shared libraries: libintro_shared.so:
     cannot open shared object  file: No such file or directory
```

# GNU Make

Make este un utilitar care permite automatizarea si eficientizarea sarcinilor. In mod particular este folosit pentru automatizarea compilarii programelor. Dupa cum s-a precizat, pentru obtinerea unui executabil provenind din mai multe surse este ineficienta compilarea de fiecara data a fiecarui fisier si apoi link-editarea. Se compileaza fiecare fisier separat, iar la o modificare se va recompila doar fisierul modificat.

## Exemplu simplu Makefile

Utilitarul Make foloseşte un fişier de configurare denumit Makefile. Un astfel de fişier conține reguli şi comenzi de automatizare. În continuare este prezentat un exemplu foarte simplu de Makefile cu ajutorul căruia se va specifica sintaxa Make.

**Exemplu 7. Makefile**

```
all:
	gcc -Wall intro-04.c -o intro-04

clean:
	rm -f intro-04
```

Pentru rularea exemplului de mai sus se folosesc comenzile:

```
$ make
gcc -Wall intro-04.c -o intro-04
$ ./intro-04
variabila my_uber_var are valoarea 12345
```

Exemplul prezentat mai sus conține două reguli: all şi clean. La rularea comenzii `make` se execută prima regulă din Makefile (în cazul de față all, nu contează în mod special denumirea). Comanda executată este `gcc -Wall intro-04.c -o intro-04`. Se poate preciza explicit ce regulă să se execute prin transmiterea ca argument comenzii `make`:

```
$ make clean
rm -f intro-04
$ make all
gcc -Wall intro-04.c -o intro-04
```

În exemplul de mai sus se foloseşte regula clean pentru a şterge executabilul intro-04 şi comanda make all pentru a obține din nou acel executabil.

Se observă că nu se transmite niciun argument comenzii make pentru a preciza fisierul Makefile care va trebui analizat. În mod implicit, GNU Make caută, în ordine, fişierele GNUmakefile, Makefile, makefile şi le analizează. Pentru a preciza ce fişier Makefile trebuie analizat, se foloseşte opțiunea -f. Astfel, în exemplul de mai jos, folosim fişierul Makefile.ex1:

```
$ mv Makefile Makefile.ex1
$ make
make: *** No targets specified and no makefile found.  Stop.
$ make -f Makefile.ex1
gcc -Wall intro-04.c -o intro-04
$ make -f Makefile.ex1 clean
rm -f intro-04
$ make -f Makefile.ex1 all
gcc -Wall intro-04.c -o intro-04
```

În primă fază se incearcă rularea simplă a comenzii `make`. Întrucât make nu găseşte niciunul din fişierele GNUmakefile, Makefile sau makefile, returnează eroare. Prin precizarea opțiunii `-f Makefile.ex1` se specifică fişierul Makefile de analizat. De asemenea, se poate preciza şi regula care să fie executată.

## Sintaxa unei reguli

În continuare este prezentată sintaxa unei reguli dintr-un fişier Makefile:

```
target: prerequisites
<tab> command
```

`target` este, de obicei, fişierul care se va obține prin rularea comenzii `command`. După cum s-a observat şi din exemplul anterior, poate să fie o țintă virtuală care nu are asociat un fişier. `prerequisites` reprezintă **dependințele** necesare pentru a urmări regula; de obicei sunt fişiere necesare pentru obținerea țintei. <tab> reprezintă caracterul tab şi trebuie neaparat folosit înaintea precizării comenzii. `command` o listă de comenzi (niciuna*, una, oricâte) rulate în momentul în care se trece la obținerea țintei.

Un exemplu indicat pentru un fişier Makefile este:

**Exemplu 8. Makefile.ex2**

```
all: intro-04

intro-04: intro-04.o
        gcc intro-04.o -o intro-04

intro-04.o: intro-04.c
        gcc -Wall -c intro-04.c

clean:
        rm -f *.o *~ intro-04
```

Se observă prezența regulii all care va fi executată implicit. all are ca dependință intro-04 şi nu execută nicio comandă; intro-04 are ca dependință intro-04.o şi realizează link-editarea fişierului intro-04.o; intro-04.o are ca dependință intro-04.c şi realizează compilarea şi asamblarea fişierului intro-04.c. Pentru obținerea executabilului se foloseşte comanda:

```
$ make -f Makefile.ex2
gcc -Wall -c intro-04.c
gcc intro-04.o -o intro-04
```

## Funcționarea unui fişier Makefile

Pentru explicarea funcționării unui fişier Makefile, vom folosi exemplul de mai sus. În momentul rulării comenzii make se poate preciza target-ul care se doreşte a fi obținut. Dacă acesta nu este precizat, este considerat implicit primul target întâlnit în fişierul Makefile folosit; de obicei, acesta se va numi all.

Pentru obținerea unui target trebuie satisfăcute dependințele (prerequisites) acestuia. Astfel,

- pentru obținerea target-ului all trebuie obținut target-ul intro-04, care este un nume de executabil
- pentru obținerea target-ului intro-04 trebuie obținut target-ul intro-04.o
- pentru obținerea target-ului intro-04.o trebuie obținut intro-04.c; acest fişier există deja, şi cum acesta nu apare la randul lui ca target în Makefile, nu mai trebuie obținut
- drept urmare se rulează comanda asociată obținerii intro-04.o; aceasta este `gcc -Wall -c intro-04.c`
- rularea comenzii duce la obținerea target-ului intro-04.o, care este folosit ca dependință pentru intro-04
- se rulează comanda `gcc intro-04.o -o intro-04` pentru obținerea intro-04
- intro-04 este folosit ca dependință pentru all; acesta nu are asociată nicio comandă deci este automat obținut.

De remarcat este faptul că un target nu trebuie să aibă neapărat numele fişierului care se obține. Se recomandă, însă, acest lucru pentru înțelegerea mai uşoară a fişierului Makefile, şi pentru a beneficia de faptul că make utilizează timpul de modificare al fişierelor pentru a decide când nu trebuie să facă nimic.

Acest format al fişierului Makefile are avantajul eficientizării procesului de compilare. Astfel, după ce s-a obținut executabilul intro-04 conform fişierului Makefile anterior, o nouă rulare a make nu va genera nimic:

```
$ make -f Makefile.ex2
make: Nothing to be done for `all'.
```

Mesajul "Nothing to be done for 'all'" înseamnă că ținta all are toate dependințele satisfăcute. Dacă, însă, folosim comanda touch pe fişierul obiect, se va considera că a fost modificat şi vor trebui refăcute target-urile

care depindeau de el:

```
$ touch intro-04.o
$ make -f Makefile.ex2
gcc intro-04.o -o intro-04
$ make -f Makefile.ex2
make: Nothing to be done for `all'.
```

La fel, dacă ştergem fişierul obiect, acesta va trebui regenerat, ca şi toate target-urile care depindeau, direct sau indirect, de el:

```
$ rm intro-04.o
$ make -f Makefile.ex2
gcc -Wall -c intro-04.c
gcc intro-04.o -o intro-04
```

## Folosirea variabilelor

Un fişier Makefile permite folosirea de variabile. Astfel, un exemplu uzual de fişier Makefile este:

**Exemplu 9. Makefile.ex3**

```
CC = gcc
CFLAGS = -Wall -g

all: intro-04

intro-04: intro-04.o
        $(CC) $^ -o $@

intro-04.o: intro-04.c
        $(CC) $(CFLAGS) -c $<

.PHONY: clean
clean:
        rm -f *.o *~ intro-04
```

În exemplul de mai sus au fost definite variabilele CC şi CFLAGS. Variabila CC reprezintă compilatorul folosit, iar variabila CFLAGS reprezintă opţiunile (flag-urile) de compilare utilizate; în cazul de faţă sunt afişarea avertismentelor şi compilarea cu suport de depanare. Referirea unei variabile se realizează prin intermediul construcţiei $(VAR_NAME). Astfel, $(CC) se înlocuieşte cu gcc, iar $(CFLAGS) se înlocuieşte cu -Wall -g.

Nişte variabile predefinite sunt $@, $^ şi $<. $@ se expandează la numele target-ului. $^ se expandează la lista de cerinţe, iar $< se expandează la prima cerinţă. În acest fel, comanda

```
$(CC) $^ -o $@
```

se expandează la

```
gcc intro-04.o -o intro-04
```

iar comanda

```
$(CC) $(CFLAGS) -c $<
```

se expandează la

```
gcc -Wall -g -c intro-04.c
```

Pentru mai multe detalii despre variabile consultaţi pagina info

[1]
sau manualul online
[2]
.

## Folosirea regulilor implicite

De foarte multe ori nu este nevoie să se precizeze comanda care trebuie rulată; aceasta poate fi detectată implicit.

Astfel, în cazul în care se precizează regula:

```
main.o: main.c
```

se foloseşte implicit comanda

```
$(CC) $(CFLAGS) -c -o $@ $<
```

Astfel, fişierul Makefile.ex2 de mai sus poate fi simplificat, folosind reguli implicite, ca mai jos:

**Exemplu 10. Makefile.ex4**

```
CC = gcc
CFLAGS = -Wall -g

all: intro-04

intro-04: intro-04.o

intro-04.o: intro-04.c

.PHONY: clean

clean:
	rm -f *.o *~ intro-04
```

Pentru rulare, se foloseşte comanda:

```
$ make -f Makefile.ex4
gcc -Wall -g   -c -o intro-04.o intro-04.c
gcc   intro-04.o   -o intro-04
```

Se observă că se folosesc reguli implicite. Makefile-ul poate fi simplificat şi mai mult, ca în exemplul de mai jos:

**Exemplul 11. Makefile.ex5**

```
CC = gcc
CFLAGS = -Wall -g

all: intro-04

intro-04: intro-04.o

.PHONY: clean

clean:
        rm -f *.o *~ intro-04
```

În exemplul de mai sus s-a eliminat regula `intro-04.o: intro-04.c`. Make "vede" că nu există fişierul intro-04.o şi caută fişierul C din care poate să-l obţină. Pentru aceasta creează o regulă implicită şi compilează fişierul intro-04.c:

```
$ make -f Makefile.ex5
gcc -Wall -g   -c -o intro-04.o intro-04.c
gcc   intro-04.o   -o intro-04
```

De remarcat este faptul ca daca avem un singur fisier sursa nici nu trebuie sa existe un fisier Makefile pentru a obtine executabilul dorit.

```
$ls
intro-04.c
$ make intro-04
cc intro-04.c -o intro-04
```

Pentru mai multe detalii despre reguli implicite consultaţi pagina info

[3]
sau manualul online
[4]
.

## Exemplu complet de Makefile

Folosind toate facilitaţile de până acum, ne propunem compilarea unui executabil client şi a unui executabil server.

Fişierele folosite sunt:

- executabilul server depinde de fişierele C server.c, sock.c, cli_handler.c, log.c, sock.h, cli_handler.h, log.h;
- executabilul client depinde de fişierele C client.c, sock.c, user.c, log.c, sock.h, user.h, log.h;

Dorim, aşadar, obţinerea executabilelor `client` şi `server` pentru rularea celor două entităţi. Structura fişierului Makefile este prezentată mai jos:

**Exemplu 12. Makefile.ex6**

```
CC = gcc                        # compilatorul folosit
CFLAGS = -Wall -g               # optiunile pentru compilare
LDLIBS = -lefence               # optiunile pentru linking
```

```
# creeaza executabilele client si server
all: client server

# leaga modulele client.o user.o sock.o in executabilul client
client: client.o user.o sock.o log.o

# leaga modulele server.o cli_handler.o sock.o in executabilul server
server: server.o cli_handler.o sock.o log.o

# compileaza fisierul client.c in modulul obiect client.o
client.o: client.c sock.h user.h log.h

# compileaza fisierul user.c in modulul obiect user.o
user.o: user.c user.h

# compileaza fisierul sock.c in modulul obiect sock.o
sock.o: sock.c sock.h

# compileaza fisierul server.c in modulul obiect server.o
server.o: server.c cli_handler.h sock.h log.h

# compileaza fisierul cli_handler.c in modulul obiect cli_handler.o
cli_handler.o: cli_handler.c cli_handler.h

# compileaza fisierul log.c in modulul obiect log.o
log.o: log.c log.h

.PHONY: clean

clean:
        rm -fr *~ *.o server client
```

Pentru obținerea executabilelor `server` şi `client` se foloseşte:

```
$ make -f Makefile.ex6
gcc -Wall -g -c -o client.o client.c
gcc -Wall -g -c -o user.o user.c
gcc -Wall -g -c -o sock.o sock.c
gcc -Wall -g -c -o log.o log.c
gcc client.o user.o sock.o log.o -lefence -o client
gcc -Wall -g -c -o server.o server.c
gcc -Wall -g -c -o cli_handler.o cli_handler.c
gcc server.o cli_handler.o sock.o log.o -lefence -o server
```

Regulile implicite intră în vigoare şi se obțin, pe rând, fişierele obiect şi fişierele executabile. Variabila LDLIBS este folosită pentru a preciza bibliotecile cu care se face link-editarea pentru obținerea executabilului.

# Depanarea programelor

Exist câteva unelte GNU care pot fi folosite atunci când nu reuşim s facem un program s ne asculte. *gdb*, acronimul de la "Gnu DeBugger" este probabil cel mai util dintre ele, dar exist si altele, cum ar fi ElectricFence, gprof sau mtrace. gdb va fi prezentat pe scurt în secțiunile ce urmeaz.

# GDB

Dacă doriți să depanați un program cu GDB nu uitați să compilați programul cu optiunea -g. Folosirea acestei opțiuni duce la includerea în executabil a informațiilor de debug.

## Rularea GDB

GDB poate fi folosit în două moduri pentru a depana programul:

- rulându-l folosind comanda gdb
- folosind fisierul core generat în urma unei erori grave (de obicei *segmentation fault*)

Cea de a doua modalitate este utilă în cazul în care bug-ul nu a fost corectat înainte de lansarea programului. În acest caz, dacă utilizatorul întâlneşte o eroare gravă, poate trimite programatorului fişierul core cu care acesta poate depana programul şi corecta bug-ul.

Cea mai simplă formă de depanare cu ajutorul GDB este cea în care dorim să determinăm linia programului la care s-a produs eroarea. Pentru exemplificare considerăm următorul program:

**Exemplu 13. exemplul-6.c**

```
#include <stdio.h>

int f(int a, int b)
{
        int c;

        c = a + b;

        return c;
}

int main()
{
        char *bug = 0;

        *bug = f(1, 2);

        return 0;
}
```

Dup compilarea programului acesta poate fi depanat folosind GDB. Dup încrcarea programului de depanat, GDB într în mod interactiv. Utilizatorul poate folosi apoi comenzi pentru a depana programul:

```
$ gcc -Wall -g exemplul-6.c
$ gdb a.out
[...]
(gdb) run
Starting program: /home/tavi/cursuri/so/lab/draft/intro/a.out

Program received signal SIGSEGV, Segmentation fault.
0x08048411 in main () at exemplul-6.c:16
16              *bug=f(1, 2);
(gdb)
```

Prima comandă folosită este `run`. Această comandă va porni execuția programului. Dacă această comandă primeşte argumente de la utilizator, acestea vor fi transmise programului. Înainte de a trece la prezentarea unor comenzi de bază din gdb, să demonstrăm cum se poate depana un program cu ajutorul fişierului core:

```
# ulimit -c 4
# ./a.out
Segmentation fault (core dumped)
# gdb a.out core
Core was generated by `./a.out'.
Program terminated with signal 11, Segmentation fault.
#0  0x08048411 in main () at exemplul-6.c:16
16              *bug=f(1, 2);
(gdb)
```

### Comenzi de bază GDB

Câteva din comenzile de bază în gdb sunt `breakpoint`, `next` şi `step`. Prima dintre ele primeşte ca argument un nume de funcție (ex: main), un număr de linie şi, eventual, un fişier (ex: `break sursa.c:50`) sau o adresă (ex: `break *0x80483d3`). Comanda next va continua executia programului până ce se va ajunge la următoarea linie din codul surs. Dacă linia de executat conține un apel de funcție, funcția se va executa complet. Dacă se doreşte şi inspectarea funcțiilor trebuie să se folosească `step`. Folosirea acestor comenzi este exemplificată mai jos:

```
$ gdb a.out
(gdb) break main
Breakpoint 1 at 0x80483f6: file exemplul-6.c, line 14.
(gdb) run
Starting program: /home/tavi/cursuri/so/lab/draft/intro/a.out

Breakpoint 1, main () at exemplul-6.c:14
14              char *bug=0;
(gdb) next
16              *bug=f(1, 2);
(gdb) next

Program received signal SIGSEGV, Segmentation fault.
0x08048411 in main () at exemplul-6.c:16
16              *bug=f(1, 2);
(gdb) run
The program being debugged has been started already.
Start it from the beginning? (y or n) y
Starting program: /home/tavi/cursuri/so/lab/draft/intro/a.out

Breakpoint 1, main () at exemplul-6.c:14
14              char *bug=0;
(gdb) next
16              *bug=f(1, 2);
(gdb) step
f (a=1, b=2) at exemplul-6.c:8
8               c=a+b;
(gdb) next
9               return c;
(gdb) next
10      }
(gdb) next

Program received signal SIGSEGV, Segmentation fault.
0x08048411 in main () at exemplul-6.c:16
```

```
16              *bug=f(1, 2);
(gdb)
```

O altă comandă utilă este list. Aceasta va lista fişierul sursă al programului depanat. Comanda primeşte ca argument un număr de linie (eventual nume fişier), o funcţie sau o adresă de la care să listeze. Al doilea argument este opţional şi precizează câte linii vor fi afişate. În cazul în care comanda nu are niciun parametru, ea va lista de unde s-a oprit ultima afişare.

```
$ gdb a.out
(gdb) list exemplul-6.c:1
1       /* exemplul-6.c */
2       #include <stdio.h>>
3
4       int f(int a, int b)
5       {
6               int c;
7
8               c=a+b;
9               return c;
10      }
(gdb) break exemplul-6.c:8
Breakpoint 1 at 0x80483d6: file exemplul-6.c, line 8.
(gdb) run
Starting program: /home/tavi/cursuri/so/lab/draft/intro/a.out

Breakpoint 1, f (a=1, b=2) at exemplul-6.c:8
8               c=a+b;
(gdb) next
9               return c;
(gdb) continue
Continuing.

Program received signal SIGSEGV, Segmentation fault.
0x08048411 in main () at exemplul-6.c:16
16              *bug=f(1, 2);
```

Comanda continue se foloseşte atunci când se doreşte continuarea execuţiei programului. Ultima comandă de bază este print. Cu ajutorul acesteia se pot afişa valorile variabilelor din funcţia curentă sau a variabilelor globale. `print` poate primi ca argument şi expresii complicate (dereferenţieri de pointeri, referenţieri ale variabilelor, expresii aritmetice, aproape orice expresie C valid). În plus, `print` poate afişa structuri de date precum `struct` şi `union`.

```
$ gdb a.out
(gdb) break f
Breakpoint 1 at 0x80483d6: file exemplul-6.c, line 8.
(gdb) run
Starting program: /home/tavi/cursuri/so/lab/draft/intro/a.out

Breakpoint 1, f (a=1, b=2) at exemplul-6.c:8
8               c=a+b;
(gdb) print a
$1 = 1
(gdb) print b
$2 = 2
(gdb) print c
$3 = 1073792080
(gdb) next
9               return c;
(gdb) print c
```

```
$4 = 3
(gdb) finish
Run till exit from #0  f (a=1, b=2) at exemplul-6.c:9
0x08048409 in main () at exemplul-6.c:16
16              *bug=f(1, 2);
Value returned is $5 = 3
(gdb) print bug
$6 = 0x0
(gdb) print (struct sigaction)bug
$13 = {__sigaction_handler =
        {
                sa_handler = 0x8049590 <object.2>,
                sa_sigaction = 0x8049590 <object.2>
        },
        sa_mask =
        {
                __val =

                {
                        3221223384, 1073992320, 1, 3221223428,
                        3221223436, 134513290, 134513760, 0, 3221223384,
                        1073992298, 0, 3221223436, 1075157952,
                        1073827112, 1, 134513360, 0, 134513393, 134513648, 1,
                        3221223428, 134513268, 134513760, 1073794080,
                        3221223420, 1073828556, 1, 3221223760, 0,
                        3221223804, 3221223846, 3221223866
                }
        },
        sa_flags = -1073743402,
        sa_restorer = 0xbffff9f2}
(gdb)
```

# Windows

## Compilatorul Microsoft cl.exe

Soluția folosită pentru platforma Windows în cadrul acestui laborator este cl.exe, compilatorul Microsoft pentru C/C++. Recomandăm instalarea Microsoft Visual C++ Express 2008 (9.0)

[5]
(versiunea Professional a Visual C++ este disponibilă gratuit în cadrul MSDNAA
[7]
). Programele C/C++ pot fi compilate prin intermediul interfeței grafice sau în linie de comandă.

În Windows fişierele cod obiect au extensia *.obj*. Pentru a compila exemplul clasic "hello world" prezentat mai jos:

**Exemplu 17. hello.c**

```
#include <stdio.h>

int main(void)
{
        printf("Hello, world!\r\n");
}
```

se poate rula:

```
cl hello.c
```

Pentru lista de opțiuni ce pot fi pasate compilatorului, rulați în linia de comanda:

```
cl /?
```

Pentru lista de opțiuni ce pot fi transmise linker-ului, rulați în linia de comandă:

```
link /?
```

Aceste opțiuni pot apărea şi într-o invocare a lui cl, după opțiunea /link. După cum se poate vedea mai sus numele linker-ului este link (programul link.exe).

Conținutul variabilei de sistem CL este adăugat automat de cl.exe în continuarea parametrilor luați din linia de comandă.

Se vor prezenta mai jos o serie de opțiuni uzuale:

- /Wall - enable all warnings
- /LIBPATH:<dir> - această opțiune indică linker-ului să caute şi în directorul dir bibliotecile pe care trebuie să le folosească programul; opțiunea se foloseşte după /link
- /I<dir> - caută şi în acest director fişierele incluse prin directiva include
- /c - se va face numai compilarea, adică se va omite etapa de link-editare.

Opțiunile cel mai des întâlnite privind optimizarea codului sunt afişate mai jos :

```
/O1 minimize space
/O2 maximize speed
/Os favor code space
/Ot favor code speed
/Od disable optimizations (default)
/Og enable global optimization
```

Fişierele de ieşire pentru o sursă *.c* sunt :

```
/Fo<file> name object file
/Fa[file] name assembly listing file
/Fp<file> name precompiled header file
/Fe<file> name executable file
```

# Biblioteci

La fel ca şi pe Linux, în Windows se pot crea biblioteci statice sau biblioteci partajate.

## Crearea unei biblioteci statice

Vom considera exemplul folosit pentru crearea de biblioteci în Linux (intro-03-main.c, intro-03-util.h, intro-03-f1.c, intro-03-f2.c):

```
>cl /c intro-03-f1.c
Microsoft (R) 32-bit C/C++ Optimizing Compiler Version 14.00.50727.42 for 80x86
Copyright (C) Microsoft Corporation.  All rights reserved.

intro-03-f1.c

>cl /c intro-03-f2.c
Microsoft (R) 32-bit C/C++ Optimizing Compiler Version 14.00.50727.42 for 80x86
Copyright (C) Microsoft Corporation.  All rights reserved.

intro-03-f2.c

>cl /c intro-03-main.c
Microsoft (R) 32-bit C/C++ Optimizing Compiler Version 14.00.50727.42 for 80x86
Copyright (C) Microsoft Corporation.  All rights reserved.

intro-03-main.c

>lib /out:intro.lib intro-03-f1.obj intro-03-f2.obj
Microsoft (R) Library Manager Version 8.00.50727.42
Copyright (C) Microsoft Corporation.  All rights reserved.

>cl intro-03-main.obj intro.lib
Microsoft (R) 32-bit C/C++ Optimizing Compiler Version 14.00.50727.42 for 80x86
Copyright (C) Microsoft Corporation.  All rights reserved.

Microsoft (R) Incremental Linker Version 8.00.50727.42
Copyright (C) Microsoft Corporation.  All rights reserved.

/out:intro-03-main.exe
intro-03-main.obj
intro.lib
```

Pentru obţinerea unei biblioteci statice folosim comanda lib. Argumentul /out: precizează numele bibliotecii statice de ieşire. Biblioteca are de obicei extensia .lib. Pentru obținerea executabilului se foloseşte cl care primeşte ca argumente fişierele obiect şi bibliotecile care conțin funcțiile dorite.

## Crearea unei biblioteti partajate

Bibliotecile partajate din Linux au ca echivalent bibliotecile DLL (Dynamic Link Library) în Windows. Crearea unei bibilioteci partajate pe Windows este mai complicată decât pe Linux. Pe de o parte pentru că în afara bibliotecii partajate (`dll`), mai trebuie creată o bibliotecă de import (`lib`). Pe de altă parte, legarea bibliotecii partajate presupune exportarea explicită a simbolurilor (funcii, variabile) care vor fi folosite.

Pentru precizarea simobolurilor care vor fi exportate de bibliotecă se folosesc identificatori predefinii:`__declspec(dllimport)` i `__declspec(dllexport)`. `__declspec(dllimport)` este folosit pentru a importa o funcție dintr-o bibliotecă. La fel, `__declspec(dllexport)` este folosit pentru a exporta o funcie dintr-o bibliotecă. Exemplul de mai jos prezintă trei programe: două dintre ele vor fi legate într-o bibliotecă partajată, iar celălalt conine codul de utilizare a funciilor exportate.

Exemplu 3. intro-03-main.c

```
#include <stdio.h>
#define DLL_IMPORTS
#include "intro-03-funs.h"

int main(void)
{
         f1();
         f2();

         return 0;
}
```

Exemplu 3. intro-03-funs.h

```
#ifndef FUNS_H
#define FUNS_H   1

#ifdef DLL_IMPORTS
#define DLL_DECLSPEC __declspec(dllimport)
#else
#define DLL_DECLSPEC __declspec(dllexport)
#endif

DLL_DECLSPEC void f1 (void);
DLL_DECLSPEC void f2 (void);

#endif
```

Exemplu 3. intro-03-f1.c

```
#include <stdio.h>
#include "intro-03-funs.h"

void f1(void)
{
         printf("Fisierul curent este %s\n", __FILE__);
}
```

Exemplu 3. intro-03-f2.c

```
#include <stdio.h>
#include "intro-03-funs.h"

void f2(void)
{
        printf("Va aflati la linia %d din fisierul %s\n", __LINE__, __FILE__);
}
```

Pentru crearea unei biblioteci partajate (`dll`) se foloseşte opțiunea `/LD` la comanda `cl`:

```
>cl /LD intro-03-f1.obj intro-03-f2.obj
Microsoft (R) 32-bit C/C++ Optimizing Compiler Version 14.00.50727.42 for 80x86
Copyright (C) Microsoft Corporation.  All rights reserved.

Microsoft (R) Incremental Linker Version 8.00.50727.42
Copyright (C) Microsoft Corporation.  All rights reserved.
```

```
/out:intro-03-f1.dll
/dll
/implib:intro-03-f1.lib
intro-03-f1.obj
intro-03-f2.obj
   Creating library intro-03-f1.lib and object intro-03-f1.exp

>cl main.obj intro-03-f1.lib
Microsoft (R) 32-bit C/C++ Optimizing Compiler Version 14.00.50727.42 for 80x86
Copyright (C) Microsoft Corporation.  All rights reserved.

Microsoft (R) Incremental Linker Version 8.00.50727.42
Copyright (C) Microsoft Corporation.  All rights reserved.

/out:main.exe
main.obj
intro-03-f1.lib
```

Alternativ, biblioteca poate fi obinută cu ajutorul comenzii `link`:

```
>link /nologo /dll /out:intro-03.dll /implib:intro-03.lib intro-03-f1.obj intro-03-f2.obj
  Creating library intro-03.lib and object intro-03.exp

>link /nologo /out:main.exe intro-03-main.obj intro-03.lib

>main.exe
Fisierul curent este intro-03-f1.c
Va aflati la linia 6 din fisierul intro-03-f2.c
```

# Nmake

*Nmake* este utilitarul folosit pentru compilare incrementală pe Windows

[6]
. Nmake are o sintaxă foarte asemănătoare cu Make. Un exemplu simplu de makefile este cel ataşat parser-ului de la tema 1:

```
OBJ_LIST = parser.tab.obj parser.yy.obj
CPPFLAGS = /nologo /W4 /Wp64 /EHsc
CFLAGS   = /nologo /W4 /Wp64 /EHsc /Za

EXE_NAMES = CUseParser.exe UseParser.exe DisplayStructure.exe

all : $(EXE_NAMES)

CUseParser.exe : CUseParser.obj $(OBJ_LIST)
  $(CPP) $(CPPFLAGS) /Fe$@ $**

UseParser.exe : UseParser.obj $(OBJ_LIST)
  $(CPP) $(CPPFLAGS) /Fe$@ $**

DisplayStructure.exe : DisplayStructure.obj $(OBJ_LIST)
  $(CPP) $(CPPFLAGS) /Fe$@ $**

clean : exe_clean obj_clean

obj_clean :
  del *.obj
```

```
exe_clean :
  del $(EXE_NAMES)
```

# Exerciții

## Quiz

Pentru auto-evaluare răspundei la întrebările din acest quiz.

## Exerciții pre-laborator

Folosiți arhiva de pre-sarcini a laboratorului.

### Linux

Folosiți directorul `lin/` din arhiva de pre-sarcini a laboratorului.

1. Intrați în subdirectorul `ex1`.
   - Folositi comanda make pentru a compila programul `ex1.c`
   - Cum se cheamă executabilul obținut?
   - Rulați-l.
2. Rămâneți în subdirectorul `ex1`.
   - Curățați directorul curent folosind `make` (trebuie să rămână doar fişierul Makefile şi `ex1.c`)
   - Parcurgeți codul programului `ex1.c`. Ce observați?
   - Folosiți o comandă simplă (`gcc ...`) pentru compilarea programului, dar care să includă opțiunea -Wall. Ce warning-uri apar?
   - Corectați greşelile din programul `ex1.c`?
3. Modificati fisierul Makefile din directorul `ex1`.
   - În urma rulării comenzii `make` programul `ex1.c` trebuie să fie compilat cu opțiunea `-Wall` (la fel ca la punctul anterior).
   - Rulați executabilul astfel obținut.
4. Intrați în directorul `ex2` şi modificați programul `ex2.c`.
   - Inlocuiti _doar_ comentariul `/* TODO */` astfel incat dupa compilare si rulare programul sa afiseze mesajul *Hello, World!*.
   - Pentru compilare folosiți fişierul Makefile existent.

### Windows

Folosiți directorul `win/` din arhiva de pre-sarcini a laboratorului.

1. Intrați în subdirectorul `ex1`.
   - Folositi comanda nmake pentru a compila programul `ex1.c`
   - Cum se cheamă executabilul obținut?

- Rulați-l.

2. Curățați directorul curent folosind `make` (trebuie să rămână doar fişierul Makefile şi `ex1.c`)
   - Modificați programul `ex1.c` astfel încât să nu mai apară mesajele de avertizare de la punctul anterior.
   - Recompilați programul `ex1.c` şi rulați executabilul obținut.
3. Modificati fisierul Makefile din directorul `ex1`.
   - În urma rulării comenzii `make` programul `ex1.c` trebuie să fie compilat cu o opțiune de optimizare.
   - Rulați executabilul astfel obținut.
4. Intrați în directorul `ex2` şi modificați programul `ex2.c`.
   - Inlocuiti _doar_ comentariul `/* TODO */` astfel incat dupa compilare si rulare programul sa afiseze mesajul *Hello, World!*.
   - Pentru compilare folosiți fişierul Makefile existent.

# Exerciții de laborator

Folosiți arhiva de sarcini a laboratorului (alternativ: aici).

## Linux

Folosiți directorul `lin/` din arhiva de sarcini a laboratorului.

1. Fazele compilării , creare makefile
   - (**0.5 puncte**) Intrați în directorul `comp/`.
     - Verificați conținutul fişierului `comp.c` din directorul curent.
     - Definiți macrourile `FIRST_NAME` şi `LAST_NAME` cu prenumele respectiv numele vostru.
     - Compilați programul manual şi obțineți executabilul `a.out`. Rulați executabilul şi analizați rezultatul.

       **Hints**

       · Citiți secțiunea Utilizare GCC din laborator.
   - (**1 punct**) Rămâneți în directorul `comp`.
     - Din fişierul sursă `comp.c` obțineți fişierul preprocesat `comp.i` şi observați cum se modifică apelul funcției de afişare.De ce dimensiunea fişierului preprocesat este semnificativ mai mare decât a fişierului sursă inițial?
     - Din fişierul preprocesat `comp.i` obțineți fişierul în limbaj de asamblare `comp.s`. Analizați conținutul acestui fişier şi modificați-l astfel încât programul executabil rezultat să afişeze prima literă a numelui în loc de numele întreg.
     - Din fişierul în limbaj de asamblare `comp.s` obțineti fişierul obiect `comp.o` iar din acesta generați fişierul executabil `comp`.

       **Hints**

       · Dacă inițial programul afişa `Harry Potter` , în urma modificării va trebui sa afişeze `Harry P` .

       · Folosiți imaginea din secțiunea Utilizare GCC şi citiți secțiunea Opțiuni.
   - (**0.5 puncte**) Curățați directorul `comp/` de fişierele intermediare, păstrați doar fişierul comp.c.
     - Completați fişierul `Makefile` pentru automatizarea procesului de construire a tuturor fişierelor specificate mai sus (`comp.i`, `comp.s`, `comp.o` , `comp`).

◊ Prima țintă din fişierul `Makefile` va trebui să construiască fişierul executabil `comp` trecând prin toate fazele compilării.
◊ Trebuie să existe ținte pentru crearea fişierelor `comp.i`, `comp.s`, `comp.o`, astfel `make comp.i` va trebui să creeze fişierul preprocesat.
◊ Modificati fisierul `comp.s` astfel incat sa afiseze doar prima litera din nume.
◊ Rulati make si executabilul obtinut si analizati rezultatul.

**Hints**

· Folosiți opțiunea -o pentru a specifica numele fişierului de ieşire din orice fază de compilare.
· Citiți sectiunea GNU Make din laborator.

2. Utilizare macrouri in linia de comandă, rulare make

♦ (**0.5 puncte** ) Intrați în directorul `dbg/`.
◊ Folosiți `make` pentru compilare.
◊ Rulați executabilul obținut.
◊ Modificați _doar_ fişierul `Makefile` pentru ca după compilare şi rulare să se afişeze mesajele transmise ca argument macroului `Dprintf`.

**Hints**:

· Trebuie definit macroul `DEBUG__`.
· Cum definiți un macro din linia de comandă, cautați `-D` in gcc(1).
· Folosiți variabila standard `CPPFLAGS`.

♦ ( **0.5 puncte**) Rămâneți în directorul `dbg/`.
◊ Creați două fişiere Makefile astfel:
◊ Fişierul `Makefile.ndbg` este folosit pentru compilarea fişierului `debug_test.c` fără suport de macrouri de debug.
◊ Fişierul `Makefile.dbg` este folosit pentru compilarea fişierului `debug_test.c` cu suport de macrouri de debug.
◊ Fişierul `Makefile.ndbg` duce la obținerea executabilului `ndbg`.
◊ Fişierul `Makefile.dbg` duce la obținerea executabilului `dbg`.

♦ Folosiți cele două fişiere Makefile pentru compilarea fişierului `debug_test.c`.

**Hints**:

◊ Folosiți makefile-ul deja existent şi adaptați-l acolo unde este nevoie.
◊ Cum puteți folosi un fişier Makefile specific folosind `make`.

♦ În fiecare caz rulați executabilul obținut.

3. Utilizare biblioteci, flag-uri de compilare

♦ (**1 punct**) Intrați în directorul `exp/`.
◊ Verificați conținutul fişierului `exp.c` din directorul curent.
◊ Rulați comanda `make`.
◊ Ce program generează eroarea respectivă?
◊ Modificați fişierul `Makefile` pentru a corecta eroarea.
◊ Rulați fişierul executabil şi analizați rezultatul. Este corect?

**Hints**

· Folosiți variabila standard `LDLIBS`.
· Compilați programul cu toate avertismentele activate. Folosiți variabila standard `CFLAGS` .
· Citiți secțiunea Biblioteci din laborator.

4. Crearea biblioteci, utilizare gdb

♦ (**2 puncte**) Intrați în directorul `mean/`.
◊ Completați fişierul `Makefile` astfel încât:
· La rularea comenzii `make` să creeze executabilele `mean_a` şi `mean_so`.
· Executabilul `mean_a` presupune obținerea bibliotecii statice `libmean_a.a` şi legarea acesteia cu modulul obiect `mean.o`.

- Executabilul `mean_so` presupune obținerea bibliotecii partajate `libmean_so.so` şi legarea acesteia cu modulul obiect `mean.o`.
- Bibliotecile sunt obținute din modulele obiect `am.o` şi `hm.o`.

◊ Rulați executabilele astfel obținute. Unde este problema?

◊ Folositi utilitarul `gdb` pentru a identifica eroarea si corectati-o.

**Hints**:

- Folositi optiunea -g pentru a obtine fisiere obiect ce contin simbolurile de debug , altfel gdb nu va e de folos.
- 1/2 = 0 , 1.0/2 = 0.5
- Nu uitați să configurați variabila `LD_LIBRARY_PATH` pentru rularea executabilului `mean_so`.
- Pentru obținerea bibliotecilor şi a executabilelor nu veți putea folosi comenzi implicite; va trebui să scrieți explicit comenzile.
- Folosiți variabila standard `LDFLAGS`.
- Citiți secțiunile Biblioteci şi Depanarea programelor din laborator.

## Windows

Folosiți directorul `win/` din arhiva de sarcini a laboratorului.

1. Compilare din mai multe fişiere
   - ♦ (**1 punct**)Intrați în directorul `comp/`.
     - ◊ In fişierul `main.c` definiți macrourile `FIRST_NAME` şi `LAST_NAME` cu prenumele respectiv numele vostru.
     - ◊ Obțineți executabilul `main.exe` prin compilarea şi legarea fişierelor C (`main.c` şi `welcome.c`).
     - ◊ Rulați executabilul `main.exe`.
     - ◊ Completați fişierul `Makefile` astfel încât la rularea comenzii `nmake` să se compileze fişierele C şi să se obțină executabilul `main.exe`

       **Hints**:

       - Citiți secțiunea referitoare la Compilatorul Microsoft cl.exe din laborator.
2. Utilizare macrouri in linia de comanda, rulare `nmake`
   - ♦ (**2 puncte**) Intrați în directorul `dbg/`.
     - ◊ Folosiți `nmake` pentru compilare.
     - ◊ Rulați executabilul obținut.
     - ◊ Modificați _doar_ fişierul `Makefile` pentru ca după compilare şi rulare să se afişeze mesajele transmise ca argument macroului `Dprintf`.

       **Hints**:

       - Trebuie definit macroul `DEBUG__`.
       - Cum definiți un macro din linia de comandă? (`cl /?`)
       - Folosiți variabila standard `CFLAGS`.
       - Citiți secțiunea Nmake din laborator.
   - ♦ Rămâneți în directorul `dbg/`.
     - ◊ Creați două fişiere Makefile astfel:
       - Fişierul `Makefile.ndbg` este folosit pentru compilarea fişierului `debug_test.c` fără suport de macrouri de debug.
       - Fişierul `Makefile.dbg` este folosit pentru compilarea fişierului `debug_test.c` cu suport de macrouri de debug.
       - Fişierul `Makefile.ndbg` duce la obținerea executabilului `ndbg.exe`.

· Fişierul `Makefile.dbg` duce la obținerea executabilului `dbg.exe`.

◊ Folosiți cele două fişiere Makefile pentru compilarea fişierului `debug_test.c`.

◊ În fiecare caz rulați executabilul obținut.

**Hints**

· Folosiți makefile-ul deja existent şi adaptați-l acolo unde este nevoie.

3. Creare biblioteci

♦ (**2 puncte**) Intrați în directorul `mean/`.

◊ Completați fişierul Makefile astfel încât:

· La rularea comenzii `make` să creeze executabilele `mean_static.exe`, respectiv `mean_dynamic.exe`.

· Executabilul `mean_static.exe` presupune obținerea bibliotecii statice `mean_static.lib` şi legarea acesteia cu modulul obiect `mean.obj`.

· Executabilul `mean_dynamic.exe` presupune obținerea bibliotecii partajate `mean_dynamic.dll` şi legarea acesteia cu modulul obiect `mean.obj`.

· Bibliotecile sunt obținute din modulele obiect `am.obj` şi `hm.obj`.

◊ Rulați executabilele astfel obținute.

**Hints**:

· Pentru legare (atât pentru executabile cât şi pentru biblioteci) folosiți comanda `link`.

· Definiti la compilare ( folosind /D ) macro-ul CONTEXT, astfel:

- CONTEXT=1, atunci cand se compileaza fisierele C pentru obtinerea bibliotecii statice.
- CONTEXT=2 sau 3 atunci cand se compileaza fisierele C pentru obtinerea bibliotecii dinamice in functie de rolul pe care il are fisierul in crearea bibliotecii ( 2-importa functii din biblioteca sau 3-exporta functii din bibilioteca).

· Atentie la numele fisierelor obiect.

· Citiți secțiunea Biblioteci din laborator.

# Soluții

- Soluții exerciții laborator 1

# Resurse utile

1. GCC Online Documentation
2. The C Preprocessor
3. GNU C Library
4. Program Library HOWTO
5. Using LD, the GNU Linker
6. GNU Make Manual
7. GDB Documentation
8. Visual C++ Express
9. Nmake Tool
10. Building and Linking with Libraries
11. Dynamic Link Library

12. Creating and Using DLLs
13. Dynamic Libraries

# Note

1. ^ info make "Using Variables"
2. ^ http://www.gnu.org/software/make/manual/make.html#Using-Variables
3. ^ info make "Implicit Rules"
4. ^ http://www.gnu.org/software/make/manual/make.html#Implicit-Rules
5. ^ http://www.microsoft.com/express/vc/
6. ^ Nmake Reference
7. ^ Programul MSDNAA