Laborator 01 - Introducere

Scop

- introducerea în tematica laboratorului
- familiarizarea cu mediul şi uneltele folosite în cadrul laboratorului

Cuvinte cheie

- programare de sistem, C, compilare, depanare, biblioteci
- gcc, make, gdb
- cl, nmake, Visual Studio

Materiale ajutătoare

- lab01-slides.pdf
- lab01-refcard.pdf
- Visual Studio Tutorials
- Video Introducere

Nice to read

- TLPI - Chapter 3, System Programming Concepts
- WSP4 - Chapter 1, Getting started with Windows

Desfăşurarea laboratorului

Laboratorul de Sisteme de Operare este unul de programare de sistem având drept scop aprofundarea conceptelor prezentate la curs şi prezentarea interfeţelor de programare oferite de sistemele de operare (system API). Un laborator va prezenta un anumit set de concepte şi va conţine următoarele activităţi:

- prezentare teoretică
- parcurgerea exerciţiilor rezolvate
- rezolvarea exerciţiilor propuse

Pentru o desfăşurare cât mai bună a laboratorului şi o **înţelegere deplină** a conceptelor vă recomandăm să parcurgeţi **conţinutul laboratorului** de acasă. De asemenea, pentru consolidarea cunoştinţelor folosiţi **suportul de laborator** prezentat în paragraful următor.

Suport de laborator

- adăugaţi ca bookmark secţiunea Resurse
- Linux
  - The Linux Programming Interface - TLPI
- Windows
  - Windows System Programming 4th Edition - WSP4
- General
  - lista de discuţii
  - canalul de IRC dedicat cursului #cs_so, de pe serverul freenode.

Prezentare

Pentru a oferi o arie de cuprindere cât mai largă, laboratoarele au ca suport familiile de sisteme de operare `Unix` şi `Windows`. Instanţele de sisteme de operare din familiile de mai sus alese pentru acest laborator sunt `GNU/Linux`, respectiv `Windows 7`.

În cadrul acestui laborator introductiv va fi prezentat mediul de lucru care va fi folosit în cadrul laboratorului de Sisteme de Operare cât şi în rezolvarea temelor de casă.

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. Pentru mai multe detalii despre proiectul GCC apăsaţi pe butonul `Show` (de acum înainte secţiunile suplimentare vor fi ascunse folosind astfel de butoane).

În cadrul laboratoarelor de Sisteme de Operare ne vom concentra asupra facilităţilor oferite de compilator pentru limbajele C şi C++. GCC are suport pentru 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 secţiunile ce urmează.

Utilizare GCC

Vom folosi pentru exemplificare un program simplu care tipăreşte la ieşirea standard un şir de caractere.

hello.c

```
#include <stdio.h>

int main(void)
{
        <a href="http://www.opengroup.org/onlinepubs/009695399/functions/printf.html">printf</a>("SO, ... hello world!\n");

      return 0;
}
```

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, în cazul nostru `hello.c`.

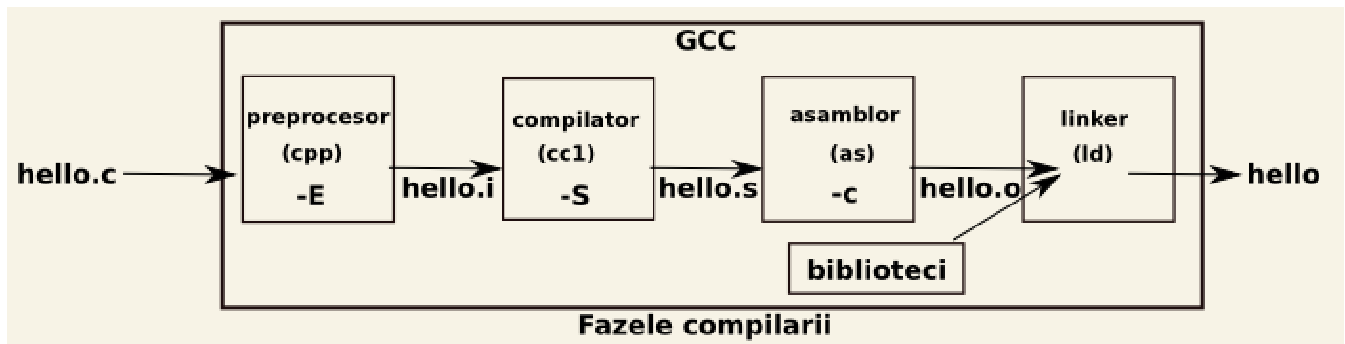
<table>
<tr>
<td>so@spook$ ls
hello.c
so@spook$ gcc hello.c
so@spook$ ls
a.out hello.c
so@spook$ ./a.out
SO, ... hello world!</td>
<td>so@spook$ ls
hello.c
so@spook$ gcc hello.c -o
hello
so@spook$ ls
hello hello.c
so@spook$ ./hello
SO, ... hello world!</td>
</tr>
</table>

Aşadar, comanda `gcc hello.c` a fost folosită pentru compilarea fişierului sursă `hello.c`. Rezultatul a fost obţinerea fişierului executabil `a.out` (nume implicit utilizat de `gcc`). Dacă se doreşte obţinerea unui executabil cu un alt nume se poate folosi opţiunea `-o`.

În mod similar se poate folosi `g++` pentru compilarea unui program sursă C++.

Fazele compilării

Compilarea se referă la obţinerea unui fişier executabil dintr-un fişier sursă. După cum am văzut în paragraful anterior comanda `gcc` a dus la obţinerea fişierului executabil `hello` din fişierul sursă `hello.c`. Intern, `gcc` trece prin mai multe faze de prelucrare a fişierului sursă până la obţinerea executabilului. Aceste faze sunt evidenţiate în diagrama de mai jos:



Opţiuni

Implicit, la o invocare a comenzii `gcc/g++` se obţine din fişierul sursă un executabil. Folosind diverse opţiuni, putem opri compilarea la una din fazele intermediare astfel:

- `-E` - se realizează doar preprocesarea fişierului sursă
  - `gcc -E hello.c` ? va genera fişierul preprocesat pe care, implicit, îl va afişa la ieşirea standard.
- `-S` - se realizează inclusiv faza de compilare
  - `gcc -S hello.c` ? va genera fişierul în limbaj de asamblare `hello.s`
- `-c` - se realizează inclusiv faza de asamblare
  - `gcc -c hello.c` ? va genera fişierul obiect `hello.o`

Opţiunile de mai sus pot fi combinate cu `-o` pentru a specifica fişierul de ieşire.

Preprocesarea

Preprocesarea presupune înlocuirea directivelor de preprocesare din fişierul sursă C. Directivele de preprocesare încep cu `#`. Printre cele mai folosite sunt:

- `#include` ? pentru includerea fişierelor header într-un alt fişier.
- `#define` şi `#undef` ? pentru definirea, respectiv anularea definirii de macrouri.
- `#if`, `#ifdef`, `#ifndef`, `#else`, `#elif`, `#endif`, pentru compilarea condiţionată.
  - utile pentru comentarea bucăţilor mari de cod. Pentru a comenta toată funcţia `do_evil_things` de mai jos nu putem folosi comentarii de tip C, ca în exemplul din dreapta, întrucat limbajul C nu permite comentariile imbricate. În astfel de cazuri se poate folosi directiva `#if` ca în exemplul din stânga.

```
#if 0
int do_evil_things(context_t *ctx)
{
        int go_drink;

        /* set student mode ON :) */

        ctx->go_drink = NO;
}
#endif
```

```
/*
int do_evil_things(context_t *ctx)
{
        int go_drink;

        /* set student mode ON :) */

        ctx->go_drink = NO;
}
*/
```

- utile pentru evitarea includerii de mai multe ori a unui fişier header, tehnică numită include guard. Astfel, în exemplul de mai jos, dacă fişierul este inclus, simbolul `_STRING_H` este deja definit de la prima includere, iar a doua operaţie de includere nu va avea nici un efect.

```
#ifndef _STRING_H
#define _STRING_H       1

__BEGIN_DECLS

/* Get size_t and NULL from <stddef.h>.  */
#define __need_size_t
#define __need_NULL

/*
 * string related defines
 */

#endif /* string.h  */
```

- `__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.

```
#include <stdio.h>

#define show_var(a) printf("Variable %s has
value %d\n", #a, a);

int main(void)
{
        int teh_var = 42;
        show_var(teh_var);
        return 0;
}
```

```
so@spook$ gcc -o show show.c
so@spook$ ls
show  show.c
so@spook$ ./show
Variable teh_var has value 42
```

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

Depanarea 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.

Compilarea

Compilarea este faza în care din fişierul preprocesat se obţine un fişier în limbaj de asamblare.

```
so@spook$ ls
hello.c
so@spook$ gcc -S hello.c
so@spook$ ls
hello.c  hello.s
```

În continuare sunt prezentate, în stânga, fişierul sursă C, `hello.c` iar în dreapta fişierul în limbaj de asamblare corespunzător `hello.s`.

<table>
<tr>
<td>

```
#include <stdio.h>

int main(void)
{
  <a
href="http://www.opengroup.org/onlinepubs/009695399/functions/printf.html">printf</a>("SO,
... hello world!\n");

  return 0;
}
```

</td>
<td>

```
    .file  "hello.c"
    .section        .rodata
.LC0:
    .string "SO, ... hello
world!"
    .text
.globl main
    .type  main, @function
main:
    pushl  %ebp
    movl   %esp, %ebp
    andl   $-16, %esp
    subl   , %esp
    movl   $.LC0, (%esp)
    call   puts
    movl
so@spook$ ls
hello.c
so@spook$ gcc -c hello.c
so@spook$ ls
hello.c  hello.o
, %eax
    leave
    ret
    .size  main, .-main
    .ident "GCC: (Ubuntu
4.4.1-4ubuntu9) 4.4.1"
    .section
.note.GNU-stack,"",@progbits
```

</td>
</tr>
</table>

Asamblarea

Asamblarea este faza în care codul scris în limbaj de asamblare este tradus în *cod maşină* reprezentând codificarea binară a instrucţiunilor programului iniţial. Fişierul obţinut poartă numele de fişier *cod obiect*, se obţine folosind opţiunea `-c` a compilatorului şi are extensia `.o`.

```
 void f(void);

/*
 * no definition for f here
 */

 int main(void)
 {
        f();
        return 0;
 }
```

Editarea de legături

Pentru obţinerea unui fişier executabil este necesară rezolvarea diverselor simboluri prezente în fişierul obiect. Această operaţie poartă denumirea de *editare de legături*, *link-editare*, *linking* sau *legare*.

<table>
<tr>
<td>

```
 void f(void);

 void f(void)
 {
 }


 int main(void)
 {
        f();
        return 0;
 }
```

</td>
<td>

```
so@spook$ ls
sample.c
so@spook$ gcc -c -o sample sample.c
so@spook$ ls
sample.c sample.o
so@spook$ gcc -o sample sample.c
/tmp/ccOVreJg.o: In function `main':
sample.c:(.text+0x7): undefined reference to `f'
collect2: ld returned 1 exit status
```

</td>
</tr>
</table>

```
so@spook$ ls
sample.c
so@spook$ gcc -c -o sample sample.c
so@spook$ ls
sample.c sample.o
so@spook$ gcc -o sample sample.c
so@spook$ ls
sample sample.c  sample.o
```

```
#include <stdio.h>

int main(void)
{
   int min = 10, max = 20, midpoint;

   /* midpoint = min+(max-min)/2; */
   midpoint = min + (max - min) >> 1;

   <a
href="http://www.opengroup.org/onlinepubs/009695399/functions/printf.html">printf</a>("The
middle of interval
           [%d, %d] is %d\n",
           min, max, midpoint);

   return 0;
}
```

Observăm că în partea stângă deşi am obţinut fişierul obiect `sample.o`, linkerul nu poate genera fişierul executabil întrucât nu găseşte definiţia funcţiei `f`. În partea dreaptă totul decurge normal, definiţia funcţiei `f` fiind inclusă în fişierul sursă.

Activarea avertismentelor

În mod implicit, o rulare a `gcc` oferă puţine avertismente utilizatorului. Pentru a activa afişarea de avertismente se folosesc opţiunile de tip `-W` cu sintaxa `-W`*`optiune-avertisment`*. *`optiune-avertisment`* poate lua mai multe valori posibile printre care `return-type`, `switch`, `unused-variable`, `uninitialized`, `implicit`, `all`. Folosirea opţiunii `-Wall` înseamnă afişarea tuturor avertismentelor care pot cauza inconsistenţe la rulare.

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

middle.c

```
so@spook$ ls
middle.c
so@spook$ gcc -o middle middle.c
so@spook$ ./middle
Middle of interval [10, 20] is 10
```

```
so@spook$ gcc -Wall -o middle middle.c
middle.c: In function ?main?:
middle.c:8: warning: suggest parentheses around ?+? inside
?>>?

#include <stdio.h>
#include "util.h"

int main(void)
{
        f1();
        f2();
        return 0;
}
```

La prima rulare, rezultatul nu e nici pe departe cel aşteptat. Eroarea poate fi detectată uşor dacă includem şi opţiunea `-Wall` la compilare. (operatorul `+` are prioritate în faţa operatorului `>>`)

Opţiuni utile

Compilarea din mai multe fişiere

Exemplele de până acum tratează programe scrise într-un singur fişier sursă. În realitate, aplicaţiile sunt complexe şi scrierea întregului cod într-un singur fişier îl face greu de menţinut şi greu de extins. În acest sens aplicaţia este scrisă în mai multe fişiere sursă denumite module. Un modul conţine, în mod obişnuit, funcţii 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:

main.c

```
#ifndef _UTIL_H
#define _UTIL_H   1

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


#endif
```

util.h

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

void f1(void)
{
   <a
href="http://www.opengroup.org/onlinepubs/009695399/functions/printf.html">printf</a>("Current
file name is %s\n", __FILE__);

}
```

f1.c

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

void f2(void)
{
   <a
href="http://www.opengroup.org/onlinepubs/009695399/functions/printf.html">printf</a>("Current
line %d in file %s\n",
           __LINE__, __FILE__);
}
```

f2.c

```
so@spook$ ls
f1.c  f2.c  main.c
util.h
so@spook$ gcc -Wall
main.c f1.c f2.c -o main
so@spook$ ls
f1.c  f2.c  main  main.c
util.h
so@spook$ ./main
Current file name f1.c
Current line 8 in file
f2.c
```

În programul de mai sus se apelează, respectiv, funcţiile `f1` şi `f2` în funcţia `main` pentru a afişa diverse informaţii. Pentru compilarea acestora se transmit toate fişierele C ca argumente către `gcc`:

```
so@spook$ ls
f1.c  f2.c  main.c  util.h
so@spook$ gcc -Wall -c  f1.c
so@spook$ gcc -Wall -c  f2.c
so@spook$ gcc -Wall -c  main.c
so@spook$ ls
f1.c  f1.o  f2.c  f2.o  main.c  main.o  util.h
so@spook$ gcc -o main main.o f1.o f2.o
so@spook$ ls
f1.c  f1.o  f2.c  f2.o  main  main.c  main.o  util.h
so@spook$ ./main
Current file name f1.c
Current line 8 in file f2.c
```

Executabilul a fost denumit `main`; pentru acest lucru s-a folosit opţiunea `-o`.

Se observă folosirea fişierului header `util.h` pentru declararea funcţiilor `f1` şi `f2`. Declararea unei funcţii se realizează prin precizarea antetului. Fişierul header este inclus în fişierul `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 `f1.c` şi `f2.c`. Codul acestora este integrat în executabil în momentul link-editării.

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

```
so@spook$ ls -l /usr/lib/libm.*
-rw-r--r-- 1 root root 496218 2010-01-03 15:19 /usr/lib/libm.a
lrwxrwxrwx 1 root root     14 2010-01-14 12:17 /usr/lib/libm.so -> /lib/libm.so.6
```

Se observă obţinerea executabilului `main` prin legarea modulelor obiect. Această abordare are avantajul eficienţei. Dacă se modifică fişierul sursă `f2.c` 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`.

Biblioteci în Linux

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

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

int main(void)
{
   double x = 1000.0;
   <a href="http://www.opengroup.org/onlinepubs/009695399/functions/printf.html">printf</a>("Cubic root for %g is %g\n", x, cbrt(x));
   return 0;
}
```

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

- **statice** - au, de obicei, extensia .a
- **partajate** - au extensia .so

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

<table>
<tr>
<td>cbrt.c
<pre>so@spook$ ls
cbrt.c
so@spook$ gcc -Wall -o cbrt cbrt.c
/tmp/ccwvm1zq.o: In function `main':
cbrt.c:(.text+0x1b): undefined reference to `cbrt'
collect2: ld returned 1 exit status
so@spook$ gcc -Wall -o cbrt -lm cbrt.c
so@spook$ ./cbrt
Cubic root for 1000 is 10</pre></td>
<td><pre>so@spook$ gcc -Wall -c f1.c
so@spook$ gcc -Wall -c f2.c</pre></td>
</tr>
</table>

Se observă că, în primă fază, nu s-a rezolvat simbolul `cbrt`. După legarea bibliotecii matematice, programul s-a compilat şi a rulat fără probleme.

Crearea unei biblioteci statice

Pentru crearea de biblioteci vom folosi fişierele din secţiunea Compilarea din mai multe fişiere. Vom include modulele obiect rezultate din fişierele sursă f1.c şi f2.c într-o bibliotecă pe care o vom folosi ulterior pentru obţinerea executabilului final.

Primul pas constă în obţinerea modulelor obiect asociate:

```
so@spook$ ar rc libintro.a f1.o f2.o
so@spook$ gcc -Wall main.c -o main -lintro
```

```
/usr/bin/ld: cannot find -lintro
collect2: ld returned 1 exit status
```

O bibliotecă statică este o arhivă ce conţine fişiere obiect creată cu ajutorul utilitarului ar (interpretaţi parametrii `rc`).

so@spook$ gcc -Wall main.c -o main -lintro -L. so@spook$ ./main Current file name is f1.c Current line 5 in file f2.c	so@spook$ gcc -fPIC -c f1.c so@spook$ gcc -fPIC -c f2.c so@spook$ gcc -shared f1.o f2.o -o libintro_shared.so so@spook$ gcc -Wall main.c -o main -lintro_shared -L. so@spook$ ./main ./main: error while loading shared libraries: libintro_shared.so: cannot open shared object file: No such file or directory

Atenţie: -lintro trebuie sa fie dupa specificarea sursei si a fişierului exectuabil

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 opţiunea **`-L`**, urmată de directorul în care trebuie căutată biblioteca (în cazul nostru este vorba de directorul curent).

Dacă biblioteca se numeşte `libnume.a`, atunci ea va fi referită cu `-lnume`

Crearea unei biblioteci partajate

Spre deosebire de o bibliotecă statică despre care am văzut că nu este nimic altceva decât o arhivă de fişiere obiect, o bibliotecă partajată este ea însăşi un fişier obiect. Crearea unei biblioteci partajate se realizează prin intermediul linker-ului. Optiunea **`-shared`** indică compilatorului să creeze un obiect partajat şi nu un fişier executabil. Este, de asemenea, indicată folosirea opţiunii **`-fPIC`** la crearea fişierelor obiect.

```
so@spook$ export LD_LIBRARY_PATH=$LD_LIBRARY_PATH:.
so@spook$ ./main
Current file name is f1.c
Current line 5 in file f2.c
```

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 funcţiei 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 locaţii definite în variabila de mediu **`LD_LIBRARY_PATH`**:

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

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 instanţă a unui program se face atribuirea noii valori înaintea comenzii de execuţie:

```
all:
       gcc -Wall hello.c -o hello
clean:
       rm -f hello
```

GNU Make

Make este un utilitar care permite automatizarea şi eficientizarea sarcinilor. În mod particular este folosit pentru automatizarea compilării programelor. După cum s-a precizat, pentru obţinerea unui executabil provenind din mai multe surse este ineficientă compilarea de fiecare dată a fiecărui fişier şi apoi link-editarea. Se compilează fiecare fişier separat, iar la o modificare se va recompila doar fişierul modificat.

Exemplu simplu de Makefile

Utilitarul make foloseşte un fişier de configurare denumit `Makefile`. Un astfel de fişier conţine reguli şi comenzi de automatizare.

<table>
<tr>
<td>Makefile
<pre>so@spook$ make
gcc -Wall hello.c -o hello
so@spook$ ./hello
SO, ... hello world!</pre></td>
<td><pre>so@spook$ make clean
rm -f hello
so@spook$ make all
gcc -Wall hello.c -o hello</pre></td>
<td><pre>so@spook$ mv
Makefile
Makefile.ex1
so@spook$
make
make: *** No
targets
specified
and no
makefile
found. Stop.
so@spook$
make -f
Makefile.ex1
gcc -Wall
hello.c -o
hello
so@spook$
make -f
Makefile.ex1
clean
rm -f hello</pre></td>
</tr>
</table>

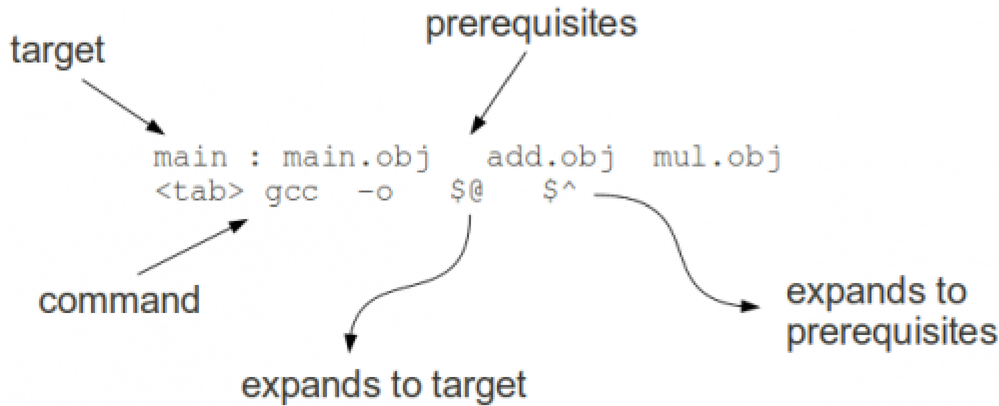
Exemplul prezentat mai sus conţine două reguli: `all` şi `clean`. La rularea comenzii **`make`** se execută prima regulă din Makefile (în cazul de faţă `all`, nu contează în mod special denumirea). Comanda executată este `gcc -Wall hello.c -o hello`. Se poate preciza explicit ce regulă să se execute prin transmiterea ca argument comenzii `make`. (comanda **`make clean`** pentru a şterge executabilul `hello` şi comanda **`make all`** pentru a obţine din nou acel executabil).

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 opţiunea **`-f`**. Astfel, în exemplul de mai jos, folosim fişierul Makefile.ex1:

```
all: hello

hello: hello.o
        gcc hello.o -o hello

hello.o: hello.c
        gcc -Wall -c hello.c

clean:
        rm -f *.o *~ hello
```

Sintaxa unei reguli

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



- **target** - este, de obicei, fişierul care se va obţine prin rularea comenzii command. După cum s-a observat şi din exemplul anterior, poate să fie o ţintă virtuală care nu are asociat un fişier.
- **prerequisites** - reprezintă dependenţele necesare pentru a urmări regula; de obicei sunt fişiere necesare pentru obţinerea ţintei.
- - 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 obţinerea ţintei.

Un exemplu indicat pentru un fişier `Makefile` este:

Makefile.ex2

```
so@spook$ make -f Makefile.ex2
gcc -Wall -c hello.c
gcc hello.o -o hello
```

Se observă prezenţa regulii `all` care va fi executată implicit.

- **all** are ca dependenţă `hello` şi nu execută nicio comandă;
- **hello** are ca dependenţă `hello.o` şi realizează link-editarea fişierului `hello.o`;
- **hello.o** are ca dependenţă `hello.c` şi realizează compilarea şi asamblarea fişierului `hello.c`.

Pentru obţinerea executabilului se foloseşte comanda:

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

Funcţionarea unui fişier Makefile

Pentru obţinerea unui target trebuie satisfăcute dependenţele (prerequisites) acestuia. Astfel, pentru obţinerea targetului implicit (primul target), în cazul nostru `all`:

- pentru obţinerea target-ului `all` trebuie obţinut target-ul `hello`, care este un nume de executabil
- pentru obţinerea target-ului `hello` trebuie obţinut target-ul `hello.o`
- pentru obţinerea target-ului `hello.o` trebuie obţinut `hello.c`; acest fişier există deja, şi cum acesta nu apare la rândul lui ca target în `Makefile`, nu mai trebuie obţinut
- drept urmare se rulează comanda asociată obţinerii `hello.o`; aceasta este `gcc -Wall -c hello.c`
- rularea comenzii duce la obţinerea target-ului `hello.o`, care este folosit ca dependenţă pentru `hello`
- se rulează comanda `gcc hello.o -o hello` pentru obţinerea executabilului `hello`
- `hello` este folosit ca dependenţă pentru `all`; acesta nu are asociată nici o comandă deci este automat obţinut.

De remarcat este faptul că un target nu trebuie să aibă neapărat numele fişierului care se obţine. Se recomandă, însă, acest lucru pentru înţelegerea 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 obţinut executabilul hello conform fişierului `Makefile` anterior, o nouă rulare a make nu va genera nimic:

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

```
all: hello

hello: hello.o
        $(CC) $^ -o $@

hello.o: hello.c
        $(CC) $(CFLAGS) -c $<

.PHONY: clean
clean:
        rm -f *.o *~ hello
```

Folosirea variabilelor

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

Makefile.ex3

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

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`.

Variabile predefinite folositoare sunt:

- **`$@`** se expandează la numele target-ului.
- **`$^`** se expandează la lista de cerinţe.
- **`$<`** se expandează la prima cerinţă.

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.

<table><tr><td><pre>so@spook$ $(CC) $(CFLAGS) -c -o $@ $<</pre></td><td><pre>CC = gcc
CFLAGS = -Wall -g

all: hello

hello: hello.o

hello.o: hello.c

.PHONY: clean

clean:
        rm -f *.o *~ hello</pre></td></tr></table>

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

<table>
<tr>
<td>

Makefile.ex4

<pre>
so@spook$ make -f Makefile.ex4
gcc -Wall -g   -c -o hello.o hello.c
gcc hello.o   -o hello

CC = gcc
CFLAGS = -Wall -g

all: hello

hello: hello.o

.PHONY: clean

clean:
        rm -f *.o *~ hello
</pre>

</td>
<td>

Makefile.ex5

<pre>
so@spook$ make -f Makefile.ex5
gcc -Wall -g   -c -o hello.o hello.c
gcc   hello.o   -o hello

so@spook$ls
hello.c
so@spook$ make hello
cc hello.c -o hello
</pre>

</td>
</tr>
</table>

De remarcat faptul că dacă avem un singur fişier sursă nici nu trebuie să existe un fisier `Makefile` pentru a obţine executabilul dorit.

```
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 mai multe detalii despre reguli implicite consultaţi pagina info [3] sau manualul online [4].

Exemplu complet de ''Makefile''

Depanarea programelor

Există câteva unelte GNU care pot fi folosite atunci când nu reuşim să facem un program să ne asculte. **gdb**, acronimul de la "Gnu DeBugger" este probabil cel mai util dintre ele, dar există şi altele, cum ar fi `ElectricFence`, `gprof sau mtrace`. gdb este prezentat pe scurt aici.

Windows

Compilatorul Microsoft cl.exe

Soluţia folosită pentru platforma Windows în cadrul acestui laborator este `cl.exe`, compilatorul Microsoft pentru C/C++. Recomandăm instalarea Microsoft Visual C++ Express 2010 (10.0) (versiunea Professional a Visual C++ este disponibilă gratuit în cadrul MSDNAA). Programele C/C++ pot fi compilate prin intermediul interfeţei grafice sau în linie de comandă. În cele ce urmează vom prezenta compilarea folosind linia de comandă. În Windows fişierele cod obiect au extensia `*.obj`.

hello.c
`cl hello.c`
	`$ cl /?    /* list of options for compiler */`
	`$ link /?  /* list of options for linker */`
	`cl /Fomyobj.obj /c mysrc.c`

Se vor prezenta mai jos o serie de opţiuni uzuale:

- **/Wall** - activează toate warning-urile
- **/LIBPATH:** - această opţiune indică linker-ului să caute şi în directorul dir bibliotecile pe care trebuie să le folosească programul; opţiunea se foloseşte după /link
- **/I** - caută şi în acest director fişierele incluse prin directiva include
- **/c** - se va face numai compilarea, adică se va omite etapa de link-editare.
- **/D** - definirea unui macro de la compilare

Opţiuni privind optimizarea codului: - **/O1** minimizează spaţiul ocupat - **/O2** maximizează viteza - **/Os** favorizează spaţiul ocupat - **/Ot** favorizează viteza - **/Od** fără optimizări (implicit) - **/Og** activează optimizările globale	Setarea numelui pentru diferite fişiere de ieşire: - **/Fo** nume fişier obiect - **/Fa** nume fişier în cod de asamblare - **/Fp** nume fişier header precompilat - **/Fe** nume fişier executabil

Exemple:

- Creare fişier obiect `myobj.obj` din sursa `mysrc.c`: cl /Famyasm.asm /FA /c mysrc.c
- Creare fişier `myasm.asm` în cod de asamblare din sursa `mysrc.c`: >lib /out:<nume.lib> <lista fişiere obiecte>

Lista completă de opţiuni o puteţi găsi aici

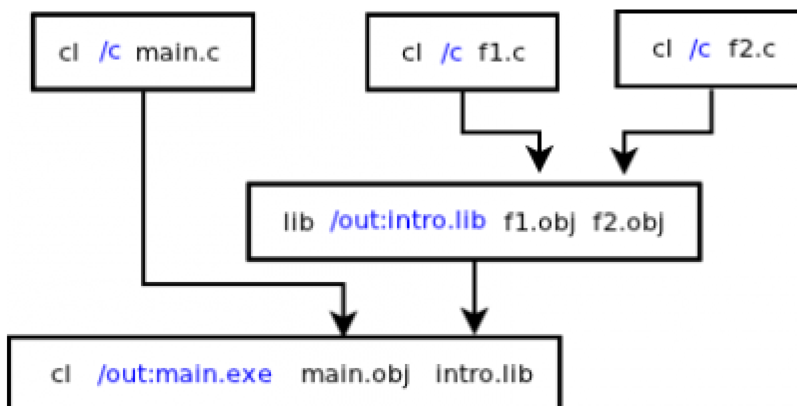
Biblioteci în Windows

Crearea unor biblioteci statice

Pentru a crea biblioteci statice se foloseşte comanda `lib`

```
# obţinem fişierul obiect f1.obj din sursa f1.c
>cl /c f1.c
Microsoft (R) 32-bit C/C++ Optimizing Compiler Version 14.00.50727.42 for 80x86
Copyright (C) Microsoft Corporation.  All rights reserved.

f1.c

#obţinem fişierul f2.obj din sursa f2.c
>cl /c f2.c
Microsoft (R) 32-bit C/C++ Optimizing Compiler Version 14.00.50727.42 for 80x86
Copyright (C) Microsoft Corporation.  All rights reserved.

f2.c

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

main.c

#obţinem biblioteca statică intro.lib din f1.obj şi f2.obj
>lib /out:intro.lib f1.obj f2.obj
Microsoft (R) Library Manager Version 8.00.50727.42
Copyright (C) Microsoft Corporation.  All rights reserved.

#intro.lib este compilat împreună cu main.obj pentru a obţine main.exe
>cl 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:main.exe
main.obj
intro.lib
```

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




```
#include <stdio.h>

#include "funs.h"

#define DLL_IMPORTS

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

        return 0;
}
```

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

Crearea unor biblioteci partajate

Bibliotecile partajate din Linux au ca echivalent bibliotecile `DLL` (Dynamic Link Library) în Windows. Crearea unei biblioteci 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 (funcţii, variabile) care vor fi folosite.

Pentru precizarea simbolurilor care vor fi exportate de bibliotecă se folosesc identificatori predefiniţi:

- **__declspec(dllimport)**, este folosit pentru a importa o funcţie dintr-o bibliotecă.
- **__declspec(dllexport)**, este folosit pentru a exporta o funcţie dintr-o bibliotecă.

Exemplul de mai jos prezintă trei programe: două dintre ele vor fi legate într-o bibliotecă partajată, iar celălalt conţine codul de utilizare a funcţiilor exportate.

main.c 1

```
#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
```

funs.h

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

void f1(void)
{
    <a
href="http://www.opengroup.org/onlinepubs/009695399/functions/printf.html">printf</a>("Current
file name is %s\n", __FILE__);

}
```

<table>
<tr>
<td>

f1.c

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

void f2(void)
{
   <a
href="http://www.opengroup.org/onlinepubs/009695399/functions/printf.html">printf</a>("Current
line %d in file %s\n",
           __LINE__, __FILE__);
}
```

</td>
<td>

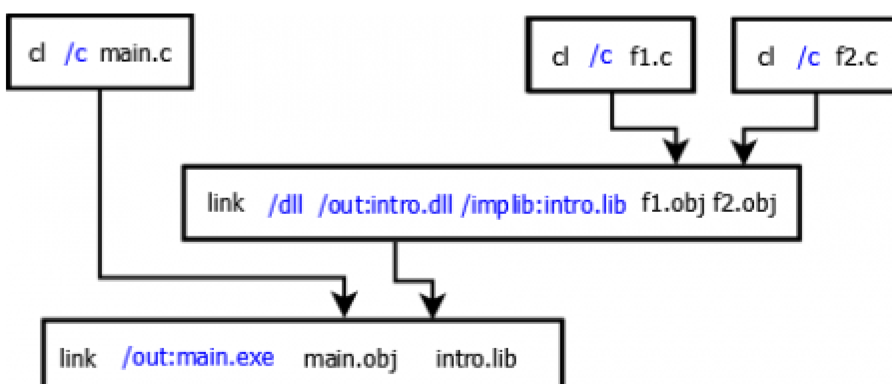
f2.c

```
>cl /LD f1.obj 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:f1.dll
/dll
/implib:f1.lib
f1.obj
f2.obj
   Creating library
f1.lib and object f1.exp

>cl main.obj 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
f1.lib
```

</td>
</tr>
</table>

Aşadar, pentru crearea bibliotecii partajate şi utlizarea acesteia de către programul main parcurgem următorii paşi:

- `f1.c` va exporta funcţia `f1()` folosind **__declspec(dllexport)**
- `f2.c` va exporta funcţia `f2()` folosind **__declspec(dllexport)**
- `main.c` va importa funcţiile `f1()` şi `f2()` folosind **__declspec(dllimport)**
- după obţinerea fişierelor obiect `f1.obj` şi `f2.obj` acestea vor fi folosite la crearea bibliotecii partajate folosind opţiunea `/LD` a comenzii `cl`.
- în final legăm `main.obj` cu biblioteca partajată şi obţinem `main.exe`



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

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

>main.exe
Current file name is f1.c
Current line 6 in file f2.c
```

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

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

EXE_NAMES = CUseParser.exe UseParser.exe DisplayStructure.exe

all : $(EXE_NAMES)

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

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

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

clean : exe_clean obj_clean

obj_clean :
  del *.obj

exe_clean :
  del $(EXE_NAMES)
```

Nmake

Nmake este utilitarul folosit pentru compilare incrementală pe Windows. Nmake are o sintaxă foarte asemănătoare cu Make. Un exemplu simplu de makefile este cel ataşat parser-ului de la tema 1:

Makefile

```
win\VS Tutorial\Debug>"Hello World.exe"
```

Nmake oferă următoarele variabile speciale:

Macro	Semnificaţie
$@	numele ţintei curente
$*	numele ţintei curente mai puţin extensia
$**	toate dependenţele unei ţinte
$?	toate dependenţele mai vechi decât ţinta

Exerciţii

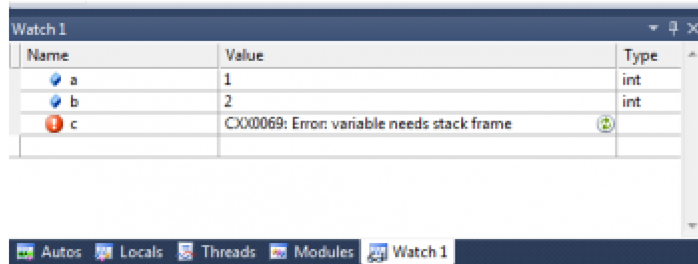
În rezolvarea laboratorului folosiţi arhiva de sarcini lab01-tasks.zip

Tastaţi CTRL+ALT+Insert pentru a vă loga pe Windows

Windows

1. (**3 puncte**) Utilizare Visual Studio
   1. (**1 punct**) Compilare şi rulare
      - Intraţi în directorul `win/VS Tutorial` şi daţi dublu-click pe fişierul `*.sln` pentru a deschide proiectul Visual Studio.
      - Pentru a compila proiectul:
        - Build Build Solution sau **F7**
      - Pentru a rula proiectul:
        - Debug Start Without Debugging sau **Ctrl + F5**

- Rulaţi executabilul din linia de comandă:
  - Porniţi command prompt-ul de Visual Studio: ToolsVisual Studio Command Prompt (sau folosiţi linkul de pe Desktop)
  - Navigaţi până ajungeţi în folderul **Debug** din directorul rădăcină al proiectului
  - Acum puteţi rula executabilul:> cl hello.c

2. (**1 punct**) Creare proiect nou:
   - File New Project
     - Project types Win32
     - Visual Studio installed templates Win32 Console Application
     - Dati un nume proiectului
     - Application Settings Console Application, **Empty Project**
   - Adăugaţi fişier-ul `VS Tutorial/debug.c` la proiect
     - Source file Right Click Add Existing Item
   - Compilaţi
   - Pentru a vedea prima eroare: F8
   - Modificaţi antetul funcţiei `f` astfel încât să întoarcă `int`
3. (**1 punct**)Debugging
   - Compilaţi şi rulaţi proiectul anterior. Programul ar trebui să afişeze valoarea salvată în variabila `bug`.
   - Adăugaţi un breakpoint la funcţia `f`:
     - Click la linia cu definiţia funcţiei `f` (linia 6), apoi F9
   - Rulaţi în mediul de debug: F5
   - Rulaţi step-by-step programul:
     - Debug Step Over (F10)
   - Setaţi watch-uri pentru variabilele a, b, c, bug:
     - Debug Windows Watch Watch1
     - Adaugati pe rand numele variabilelor
     - 

   - Observaţi valoarea variabile `bug` în momentul în care apare problema, cât şi mesajele afişate în fereastra Output
   - Remediaţi problema şi rulaţi din nou programul
4. Mai multe informaţii utile despre Visual Studio găsiţi aici.

2. (**4 puncte**) Makefiles
   - **Atenţie** Acest set de exerciţii se rulează din command-shell-ul de Visual Studio (**nu** `cmd.exe`).
     - Găsiţi link la acesta pe Desktop sau accesând ToolsVisual Studio Command Prompt
   1. (**1 punct**) Compilarea unui singur fişier
      - Intraţi în directorul win/1-hello/.
      - Folosind cl obţineţi şi rulaţi executabilul `hello`.
        - > hello.exe
        - > cl /Fehello_win hello.c
      - Folosind cl obţineţi şi rulaţi executabilul `hello_win`.
        - > hello_win.exe
        - > nmake
      - Rămâneţi în directorul win/1-hello/.
      - Analizaţi fişierul Makefile.
      - Folosind nmake obţineţi şi rulaţi executabilul `hello`.
        - > hello.exe
        - $ gcc hello.c
   2. (**1 punct**) Compilarea din mai multe surse
      - Intraţi în directorul win/2-debug/.
        - Analizaţi fişierele `main.c` şi `add.c`. (Hint: `type main.c`)
        - Completaţi fişierul **Makefile.ndbg** astfel încât:
          - să obţineţi obiecte din sursele `main.c` şi `add.c`.
          - să obţineţi executabilul `main.exe` din obiectele creat.
        - Completaţi fişierul **Makefile.dbg** astfel încât:
          - să compilaţi cu simbolul DEBUG__ definit.
          - să obţineţi obiecte din sursele `main.c` şi `add.c` şi executabilul `main.exe` (ca la subpunctul precedent)
        - **Hints**
          - Înlocuiţi "# TODO ?" cu regulile necesare
          - Folositi /f pentru preciza fişierul Makefile
          - Folosiţi cl /? pentru a determina cum se defineşte un macro în linia de comandă.
            - Revedeţi secţiunea cl.

3. (**2 puncte**)
   1. Creare bibliotecă statică
      - Intraţi în directorul `win/3-bounds/`
      - Analizaţi fişierele `bounds.c`, `min.c` şi `max.c`
      - Completaţi fişierul **`Makefile.static`** astfel încât:
        - La rularea `nmake bounds_static.lib` să se creeze biblioteca statică **`bounds_static.lib`**. Biblioteca va conţine fişierele obiect asociate fişierelor `min.c` şi `max.c`
        - La rularea comenzii nmake să se creeze fişierul executabil `bounds_static.exe` obţinut din legarea fişierului obiect corespunzător fişierului `bounds.c` cu biblioteca `bounds_static.lib`
      - **Hints**
        - Revedeti sectiunea crearea unei biblioteci statice
        - Chemaţi asistentul înainte de a trece la subpunctul următor.
   2. Creare bibliotecă dinamică
      - Rămâneţi în directorul `win/3-bounds/`
      - Completaţi fişierul **`Makefile.dynamic`** astfel încât:
        - La rularea `nmake bounds_dynamic.dll` să se creeze biblioteca dinamică **`bounds_dynamic.dll`**. Biblioteca va conţine fişierele obiect asociate fişierelor `min.c` şi `max.c`
        - La rularea comenzii `nmake` să se creeze fişierul executabil `bounds_dynamic.exe` obţinut prin legarea fişierului obiect corespunzător fişierului `bounds.c` cu biblioteca `bounds_dynamic.dll`
      - **Hints**
        - Revedeti secţiunea crearea unei biblioteci dinamice
        - Nu uitaţi să adăugaţi `__declspec(dllimport)` şi `__declspec(dllexport)` la **antetele** funcţiilor
        - La crearea executabilului final aveţi grijă să vă legaţi cu fişierul ".lib", nu ".dll"
        - Dacă folosiţi `cl` pentru a crea biblioteca dinamică, puteţi folosi '/Fe' pentru a seta numele bibliotecii

Linux

1. (**3 puncte**) Fişiere 'make'
   1. (**1 punct**) Compilarea unui singur fişier
      - Intraţi în directorul `lin/1-hello/`.
      - Analizaţi fişierul `hello.c`.
      - Folosind gcc obţineţi şi rulaţi executabilul `a.out`.
        - $ ./a.out
        - $ gcc -o hello hello.c
      - Folosind gcc obţineţi executabilul hello.
        - $ ./hello
        - $ ./hello
   2. (**2 puncte**) Creare biblioteci
      1. Creare bibliotecă statică
         - Intraţi în directorul `lin/2-lib/`
         - Completaţi fişierul `Makefile_static` astfel încât:
           - La rularea comenzii `make libhexdump_static` să creeze biblioteca statică `libhexdump_static.a` Biblioteca va conţine fişierele obiect asociate fişierelor `hexdump.c` şi `sample.c`
           - La rularea comenzii `make` să creeze executabilul `main_static` obţinut din legarea fişierului obiect corespunzător lui `main.c` cu biblioteca `libhexdump_static.a`.
         - **Hints**
           - Folosiţi ar pentru a crea biblioteca statică
           - Revedeţi secţiunea crearea unei biblioteci statice.
      2. Creare bibliotecă dinamică
         - Rămâneţi în directorul `lin/2-lib/`
         - Completaţi fişierul `Makefile_dynamic` reguli astfel încât:
           - La rularea comenzii `make libhexdump_dynamic` să creeze biblioteca dinamică `libhexdump_dynamic.so`. Biblioteca va conţine fişierele obiect asociate fişierelor `hexdump.c` şi `sample.c`
           - La rularea comenzii `make` pe lângă biblioteca dinamică `libhexdump_dynamic.so` obţinută anterior să se creeze şi executabilul `main_dynamic` obţinut din legarea fişierului obiect corespunzător lui `main.c` cu biblioteca partajată `libhexdump_dynamic.so`.
         - **Hints**
           - Revedeţi secţiunea despre crearea unei biblioteci dinamice.

BONUS

1. **1 so karma** Compilare din mai multe surse, opţiuni la compilare.
   - Intraţi în directorul `lin/3-ops/`.
     - Analizaţi fişierele `ops.c`, `mul.c` şi `add.c`
     - Fişierul `ops.c`, se foloseşte de funcţiile definite în `mul.c` şi `add.c` pentru a realiza operaţii de adunare şi înmulţire simple.
     - Creaţi fişierul `Makefile`, astfel încât:
       - obţineţi din surse fişierele obiect `mul.o`, `add.o` şi `ops.o`.
       - obţineţi executabilul `ops` din obiectele create.
     - **Observaţi** rezultatul obţinut pentru sumă şi înmulţire. Este corect? Rezolvaţi.
   - Rămâneţi în directorul `lin/3-ops/`
     - La compilarea fişierului `ops.c` definiţi simbolul `HAVE_MATH`.
     - Obţineţi şi rulaţi executabilul `ops`
       - **Hints**
         - Revedeţi secţiunea despre compilarea mai multor fişiere.
         - Pentru definirea simbolurilor în linia de comandă se foloseşte opţiunea `-D`.
         - Pentru a folosi funcţia pow trebuie să includeţi fişierul `math.h` şi să legaţi biblioteca `libm`.
         - Legarea unei biblioteci se face folosind opţiunea `-l`.
2. **1 so karma** Utilizare gdb
   - Intraţi în directorul `lin/4-gdb/`
     - Analizaţi fişierul `fault.c`
     - Completaţi fişierul `Makefile` astfel încât la rularea comenzii `make` să se obţină fişierul executabil fault.
     - Compilaţi
     - Folosiţi gdb pentru a determina cauza erorilor din fişierul `fault.c`
       - **Hints**
         - Citiţi secţiunea GDB
         - Folosiţi opţiunea `-g` pentru a compila sursa cu simbolurile de debug incluse.
         - Folosiţi comanda `print` pentru a printa valorile variabilelor când faceţi depanarea.
3. **1 so karma** Editare de legături
   - Intraţi în directorul `lin/5-linker/`
     - Analizaţi fişierele `main.c` şi `str.c`.
     - De ce nu obţinem o eroare de compilare?
     - Rulaţi programul `main` şi explicaţi rezultatele.

EXTRA

- JNI

Soluţii

lab01-sol.zip

Resurse utile

- Linux
  1. GCC online documentation
  2. Tech Talk: Preprocesorul C
  3. Linking, Loading and Library Management under Linux
  4. The GNU C Library
  5. Program Library HOWTO
  6. GNU make manual
  7. GDB documentation
- Windows
  1. Visual C++ Express
  2. Nmake tool
  3. Nmake Macros
  4. Dynamic link library

5. Creating and using DDL's
6. Dynamic libraries

From:
http://elf.cs.pub.ro/so/wiki/ - **Sisteme de Operare**

Permanent link:
http://elf.cs.pub.ro/so/wiki/laboratoare/laborator-01

Last update: **2012/03/16 18:33**