

# Instrumente si Tehnici de Baza in Informatica

Semestrul I 2025-2026

Vlad Olaru

# Curs 10 - outline

- compilarea si link-editarea programelor
- depanarea programelor
- gestiunea proiectelor de programare

# De la sursa la executabil

- sursa – fisierul scris in limbaj de programare (ex. C, C++)
- compilator – traduce fisierul sursa in limbaj de nivel jos (limbaj masina)
- obiect – fisier binar rezultat din compilarea unui fisier sursa
- bibliotecă – fisier binar (deja compilat) ce ofera o anumita functionalitate (ex. functii de comunicare in retea)
- linkeditor (linker) – leaga obiecte si biblioteci pentru a produce executabilul
- executabil static – toate obiectele si bibliotecile sunt induse in fisierul executabil rezultat
- executabil dinamic – contine doar datele si instructiunile proprii + functii stub catre apelurile de biblioteca; bibliotecile sunt pastrate in fisiere separate

# Load time

Rezolvarea dependentelor executabilelor dinamice la momentul incarcarii in memorie:

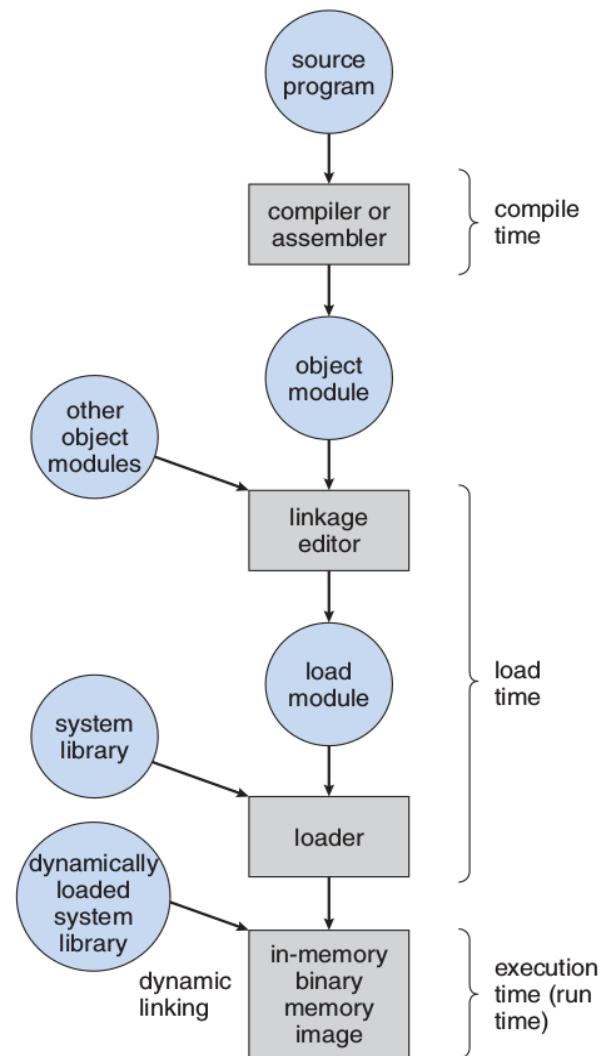
1. sistemul de operare incarca executabilul in memorie
2. inainte de a fi executat se cauta bibliotecile folosite
3. se incarca toate bibliotecile necesare
4. in caz ca nu se gasesc una sau mai multe biblioteci, executabilul este scos din memorie si executia este anulata

# Run time

Rezolvarea dependentelor executabilelor dinamice la nevoie:

1. sistemul de operare incarca executabilul in memorie
2. inainte de a fi executat se cauta bibliotecile strict necesare lansarii programului
3. pe parcursul executiei daca este nevoie de o functie de biblioteca
  1. executia programului este oprită
  2. se cauta in sistem biblioteca necesara
  3. se incarca in memorie
  4. se reporneste de unde a ramas executia programului
4. in orice moment daca o biblioteca nu este gasita si incarcata executabilul este scos din memorie si executia este anulata

# Compilare



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

# Compilatoare

Solutii UNIX:

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

Indiferent de compilator, in UNIX există comanda cc (C compiler) și, optional, comanda c++ (C++ compiler).

# Executabil simplu

În forma cea mai simplă compilatorul

- primește ca argumente fisierele sursa
- produce executabilul a.out
- denumire istorică: formatul de executabil standard pe prima versiune UNIX

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

Execuție prefixată cu ./ - shell-ul căuta executabilul în directorul curent (nu în \$PATH)

Cu nume de executabil:

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

# Obiecte

- generate cu optiunea -c
- compilatorul se opreste dupa producerea obiectului
- nu continua cu crearea unui executabil
- obiectul va fi folosit impreuna cu alte obiecte si biblioteci pentru a produce executabilul final

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

# Biblioteci predefinite

- Bibliotecile externe accesate cu optiunea - *lname* (**fara spatii!**)
  - Obs: numele complet al bibliotecii este *liblname.a* sau *liblname.so*
    - compilatorul cauta biblioteca intr-o lista de directoare
    - lista este de regula generata si intretinuta de comanda `ldconfig(1)`
    - variabila de mediu `LD_LIBRARY_PATH` contine directoare proprii ce contin biblioteci
    - biblioteca gasita este folosita impreuna cu alte obiecte si biblioteci pentru a produce executabilul final

Folosirea bibliotecii de functii matematice *libm.a* sau *libm.so*:

```
$ cc hello.c -lm
```

# Crearea bibliotecilor statice

- se creeaza o arhiva cu fisierele obiect

```
$ ar -r libhello.a hello.o
```

- eventual se verifica arhiva

```
$ ar -t libhello.a
```

```
$ objdump -p libhello.a
```

```
$ nm libhello.a
```

- se linkeditaaza programul

```
$ gcc -o main main.c -L. -lhello
```

- se ruleaza programul

```
$ ./main
```

# Crearea bibliotecilor dinamice

- se creeaza fisierele obiect in format fpic/fPIC  
`$ gcc -c hello.c -fpic`
- se creeaza biblioteca dinamica (shared object)  
`$ gcc -shared -o libhello.so hello.o`
- eventual se pune biblioteca in directoarele predefinite ale sistemului (/lib,/usr/lib,/usr/local/lib, samd) si se ruleaza ldconfig
- se linkedita programul (cu -L daca nu s-a efectuat pasul anterior)  
`$ gcc -o main main.c -L. -lhello`
- se verifica dependentele programului  
`$ ldd main`

# Rularea programelor linkeditate dinamic

- daca biblioteca e in rpath (s-a rulat ldconfig, ldd o gaseste)

```
$ ./main
```

- altfel, trebuie specificata calea catre biblioteca dinamica

```
$ LD_LIBRARY_PATH=. ./main
```

sau

```
$ export LD_LIBRARY_PATH=.
```

```
$ ./main
```

# Optiuni de compilare utile

Optiuni utile folosite des la compilare

- `-o nume` – numele executabilului sau obiectului rezultat (ex. `hello` in loc de `a.out`)
- `-Wall` – afiseaza toate mesajele de *warning*
- `-g` – pregeste executabilul pentru o sesiune de depanare (debug)
- `-O $n$ umar` – nivelul de optimizare ( $0 \rightarrow$  fara optimizari)
- `-O2` este folosit implicit, `-O0` este folosit adesea impreuna cu `-g` pentru depanare

Ex: compilare `hello.c` cu simboluri de debug, fara optimizare, in executabilul cu numele `hello`

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

# Depanare

Notiuni folosite intr-o sesiune de depanare

- break, breakpoint
  - punct in care sa se opreasca executia
  - poate fi o functie, o linie dintr-un fisier sursa, chiar si o instructiune de asamblare
  - se poate investiga starea variabilelor la momentul respectiv
- backtrace, trace, stack trace
  - apelul de functii care a dus in punctul curent
  - exemplu: main() → decompress() → zip decompress()
- watch, watchpoint
  - punct in care să se opreasca executia *daca este indeplinita o conditie ceruta de utilizator*
  - poate fi modificarea unei zone de memorie sau a unei variabile

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

# Comenzi gdb(1)

## Comenzi uzuale

- `break simbol` – seteaza breakpoint la simbolul respectiv (ex. `main()` sau `hello.c:10`)
- `run` – incepe executia (de regula dupa ce au fost setate breakpointurile)
- `next` – o data ajuns intr-un punct de oprire, mergi la urmatoarea instructiune
- `continue` – continua pana la urmatorul punct de oprire sau pana la terminarea executiei programului
- `print variabila` – afiseaza valoarea unei variabile
- `quit` – iesire din gdb(1)

# Sesiune de depanare

```
$ cc -g -O0 hello.c -o hello
$ gdb hello
(gdb) break main
Breakpoint 1 at 0x546: file hello.c, line 5.
(gdb) run
Starting program: /home/user/src/hello
Breakpoint 1 at 0x101a49c00546: file hello.c, line 5.

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

# Proiecte

Produsele software sunt alcătuite din mai multe componente:

- diferite module (ex. comunicatie, logging)
- diferite biblioteci (ex. compresie, criptografie, http)
- mai multe fisiere sursa (numite si unitati de compilare)
- intre ele apar diferite dependente (ex. modulul de comunicatie depinde de biblioteca http)
- inter-dependentele dicteaza ordinea de compilare
- dependentele si ordinea de compilare descrise formal in fisiere de tip Makefile
- instructiunile dintr-un Makefile sunt executate cu ajutorul comenzi make(1)

Obs: Makefile e nume implicit, se poate insa folosi orice alt nume

```
$ make -f MyOwnMakefile
```

# Makefile

Format fix

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

- target – rezultatul regulii
- dependency – resursele necesare (deja existente)
- commands – comenziile pentru a produce target-ul
- **Atentie!** – este un TAB inaintea commands
- primul target din fisier sau, daca exista, target-ul all vor fi executate implicit (i.e., apel make fara parametru)

**Avantaj:** recompileaza doar fisierele modificate!

# Makefile simplu

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

# Variabile Makefile

## Variabile utile

- `$@` – numele target-ului curent (partea stanga)
- `$^` – toate dependentele (partea dreapta)
- `$<` – numele primei dependente (ex. dependency1)
- `.type1.type2:` – target-ul transforma fisiere de tip 1 in fisiere de tip 2 (ex. `.c.o:`)
- `$(VAR:old=new)` – inlocuieste `old` cu `new` in variabila `VAR`

# Makefile complex

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

all: $(PROG)

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