Utilizarea Sistemelor de Operare

Compilarea programelor în Linux

- Curs 6 -
10.11.2005

Universitatea POLITEHNICA Bucureşti

Compilarea programelor - Plan

- Program vs. Executabil
- Interpretare vs. Compilare
  - Interpretor
  - Compilator/Asamblor (+LinkEditor)
- Biblioteci
  - Introducere
  - Clasificare
- Compilarea cu GCC
  - Introducere
  - Optimizarea executabilelor
  - Utilizarea bibliotecilor cu GCC-ul
- GNU Make
- **Workshop**
  - Studiul unui executabil
  - Analiza timpilor de execuție
  - Studiul bibliotecilor
  - GNU Make – exemplu
  - Studiul apelurilor de sistrem – comanda **strace**
  - **Code proofing**
    - Memory leak detection
    - Call analysis

Programul

- Programul este o listă de instrucțiuni pas-cu-pas scrise într-un anumit limbaj de programare, instrucțiuni ce au ca scop îndeplinirea unui task
- În general, programele se prezintă într-un format uşor de înțeles
- Computerul poate executa programele doar dacă le "înțelege"
- Pentru "înțelegere", programele pot fi
  - Aduse într-o formă intermediară prin "traducere" (*compilare*) într-un limbaj uşor accesibil sistemului de operare şi procesorului
  - *Interpretate* în momentul în care se execută

Executabilul

- Executabilul (fişierul executabil) conține (de cele mai multe ori) o reprezentare binară de **instrucțiuni masină**
- Conținutul executabilelor "binare":
  - **Date (constante)**
  - **Codul efectiv ce urmează a fi rulat**
  - **Informații pentru debugging**
  - **Apeluri de sistem**
- Există cazuri cand executabilul conține o formă intermediară de cod ce necesită prezența unui **interpretor** pentru a putea fi rulat

Interpretare vs. Compilare

- Interpretare = traducerea (pe loc) şi execuţia unui program (în general numit **script**)
- Compilare = traducerea unui program (în general numit **cod sursă**) într-un *limbaj uşor de înteles de către maşina* (numit **cod obiect**), pentru a fi ulterior executat de către sistemul de operare

Interpretare vs. Compilare (pros & cons)

- Interpretare
  - Pros:
    - Cod uşor de inţeles de către progamator (*human-readable* - nivel de abstractizare ridicat)
    - *Debugging* facil
  - Cons:
    - Execuţia este lentă comparativ cu execuţia codului compilat
- Compilare
  - Pros:
    - Viteza mare de execuţie
  - Cons:
    - Procesul de *debugging* poate fi încet

Interpretarea vs. Compilarea Just-in-Time

- Interpretorul este un executabil (aplicație) care execută un program (**script**) care nu poate fi “înțeles” direct de catre sistemul de operare şi de procesor.
- Uneori se apelează la o etapă intermediară în procesul de interpretare - compilarea programului în **format byte-code** (o reprezentare intermediară optimizată). Acest cod este ulterior interpretat de catre un program pe maşina gazdă
- Compilarea **Just-in-Time** (sau compilarea dinamica) este o tehnică folosită pentru îmbunătățirea performanțelor la rularea de byte-code
- Într-un mediu JIT, programele sunt inițial compilate în format byte-code (format ce este în general **portabil**), dupa care acest cod este compilat pentru maşina pe care se doreşte rularea programului

Interpretarea vs. Compilarea Just-in-Time

- Interpretoare
  - **BASH**
  - PHP
  - JavaScript
  - Python
  - Ruby
  - Basic
  - etc.
- Compilatoare JIT
  - Java (Sun)
  - .Net (Microsoft)

Compilator vs. Asamblor

- Atât compilatorul cât şi asamblorul au ca scop producerea de **cod obiect** (**cod maşină**), ce urmează a fi rulat pe o anume maşină
- Spre deosebire de compilator, **asamblorul** utilizeaza un set restrâns de instrucțiuni, în mare parte fiind doar instrucțiunile procesorului în format literal (human readable), nu binar
- Programarea în *assembler* reprezintă cel mai scăzut nivel de programare (se programează direct resursele procesorului)

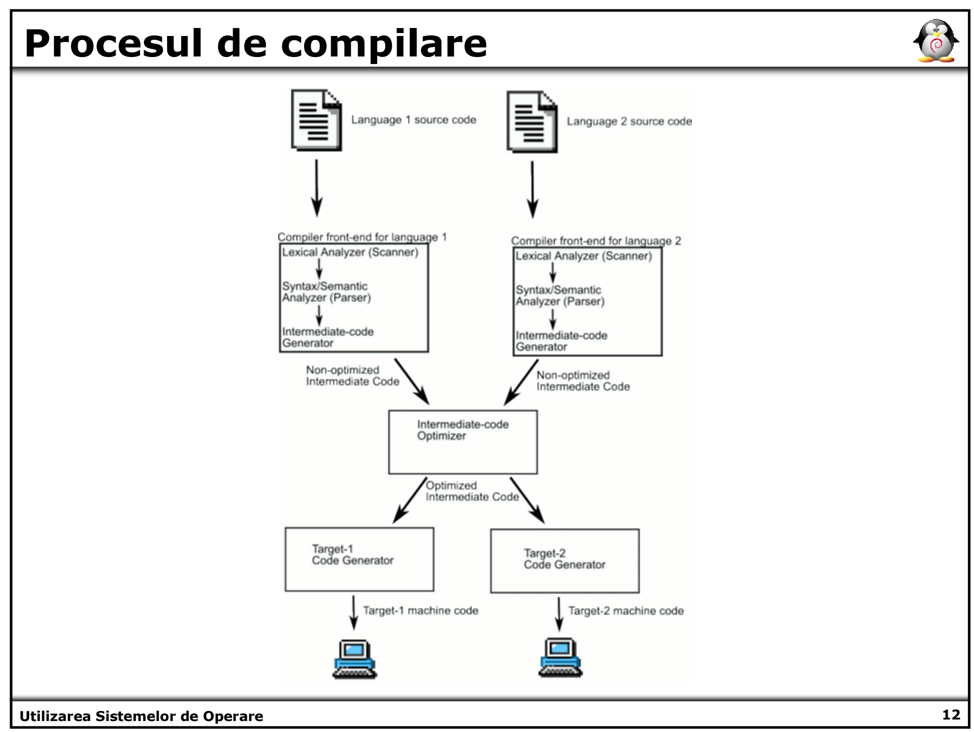
Compilatorul

- Compilatorul traduce **codul sursa** (de obicei scris intr-un limbaj evoluat de programare – high-level language) in **cod obiect**
- Aceasta traducere se poate realiza intr-unul sau mai multi pasi
- Rezultatul traducerii este pus intr-un fisier executabil, folosind un anumit **format**

Design-ul unui compilator

- De obicei un compilator are 2 componente principale:
- Front-end
  - Analiza lexicală – spargerea sursei în unitati atomice (cuvinte cheie, variabile etc.)
  - Analiza sintactică – identificarea structurilor sintactice (a.k.a. *parsing*)
  - Analiza semantică – sunt detectacte "ințelesurile" cuvintelor
  - Reprezentare intermediară (o structura de date, de obicei **arbore** sau **graf**)
- Back-end (de obicei multi-pas)
  - Analiza de compilare (analiza reprezentării intermediare pentru optimizări ulterioare, graful apelurilor etc.)
  - Optimizari
  - Generarea codului

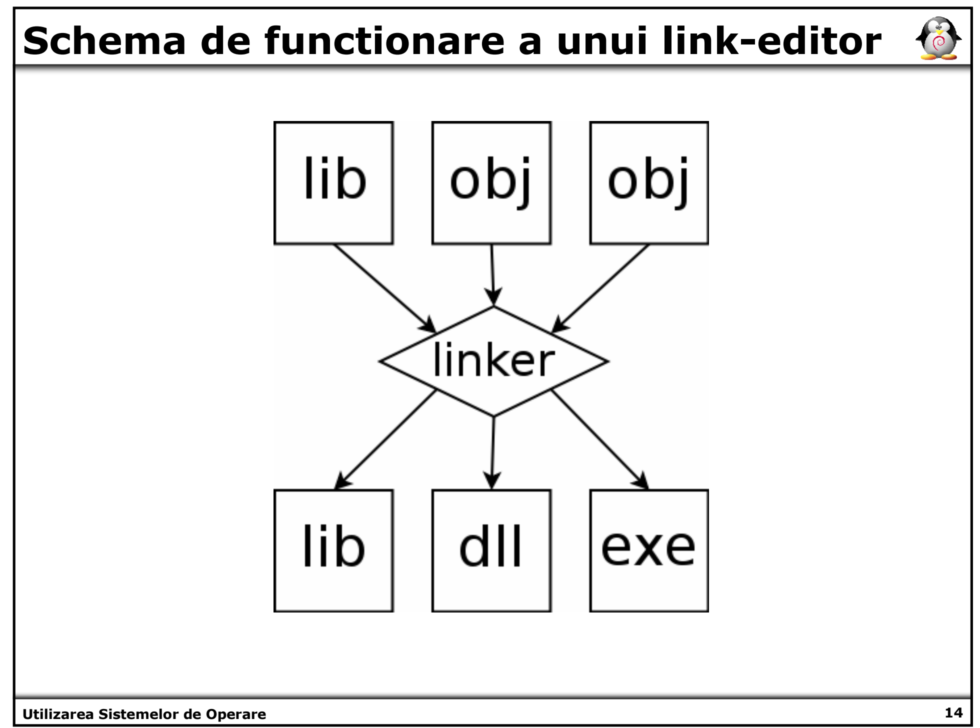
Procesul de compilare



Link-editare

- Un **linker** sau **link-editor** are ca sarcină să asambleze mai multe fişiere obiect într-un singur program executabil
- **Fisierele obiect** sunt module ce conțin cod maşina şi informații pentru linker. Aceste informații sunt reprezentate în mare parte prin definiții de **simboluri**, ce pot fi de doua feluri:
  - **Simboluri definite (exportate)** sunt funcții/variabile ce se găsesc în modul, pentru a fi utilizate de catre alte module
  - **Simboluri nedefinite (importate)** sunt funcții/variabile ce nu se găsesc în modul şi trebuie căutate în alte module
- Astfel, principalul task ce trebuie îndeplinit de către un linker este să "rezolve" simbolurile nedefinite, înlocuind referințele către acestea cu adresele simbolurilor din alte module

Schema de functionare a unui link-editor



Biblioteci

- Bibliotecă = library ≠ librarie
- O bibliotecă este o colecție de funcții des utilizate de către programe
- Bibliotecile se disting de executabile prin faptul ca ele nu sunt independente
- În prezent, o mare parte din codul programelor se regăseşte in biblioteci
- Exista 3 tipuri majore de biblioteci:
  - Biblioteci statice (static libraries)
  - Biblioteci partajate (shared libraries)
  - Biblioteci dinamice (dynamic libraires – a.k.a. DLLs)

Biblioteci de functii

- **Bibliotecile statice (static libraries)** sunt colecții de functii care la link-editare sunt incluse complet în executabilele obtinute
  - **Avantaje:** nu exista posibilitatea apariției de incompatibilități între library şi aplicație, din moment ce sunt compilate împreună; sunt utilizate doar la compilare, deci nu sunt critice pentru funcționarea sistemului
  - **Dezavantaj:** aplicațiile pot deveni mari, atât ca executabile cât şi ca sursă.

Biblioteci de functii

- **Bibliotecile partajate (shared libraries)** sunt colectii de funcții care în momentul link-editării sunt referite (reffered) în codul executabil rezultat şi incărcate de către sistem în momentul apelării programului.
  - **Avantaje:** aplicațiile sunt mici, simplu de întretinut
  - **Dezavantaje:** la modificări semnificative aplicațiile dependente trebuie recompilate; sunt necesare la rularea aplicațiilor, deci sunt critice pentru stabilitatea sistemului

Biblioteci de functii

Bibliotecile dinamice (dynamic libraries) sunt colectii de funcții care sunt încarcate de către sistemul de operare în momentul în care sunt căutate.

Dynamic load libraries = biblioteci dinamice încarcate la cerere de către programe

Exemplu de utilizare: programarea plugin-urilor pentru diferite aplicații

Bibliotecile dinamice sunt cunoscute în Windows sub numele de DLL-uri.

Compilarea cu GCC

GNU Compiler Collection

- Ada (GCC for Ada a.k.a. GNAT)
- C
- C++ (GCC for C++ a.k.a. G++)
- Fortran (GCC for Fortran a.k.a. GFortran)
- Java (GCC for Java a.k.a. GCJ)
- Objective-C

= o **colecție de compilatoare** produse în **proiectul GNU**, colecție distribuita de **Free Software Foundation** (FSF) sub licență GNU GPL and LGPL

Compilarea cu GCC

Sintaxa:

```
$ gcc <sursa> -g -Wall -o <fexecutabil>
```

- `-g` = include simboluri pentru debugging în fisierul rezultat
- `-Wall` = afişeaza toate warning-urile
- `-o <fexecutabil>` = rezultatul compilării este pus în fişierul `<fexecutabil>`
- În absenţa lui `-o <fexecutabil>`, în mod implicit fişierul rezultat este `a.out`

- Pentru rularea programului rezultat în directorul curent, în Linux se utilizează comanda

```
$ ./fexecutabil
```

Optimizarea programelor la compilare

Se poate modifica gradul de optimizare al unui executabil prin utilizarea parametrului –Ox

- **-O0**: nici o optimizare
- **-O1** (acelaşi rezultat este obtinut şi cu „-O” fără număr după):
- **-O2**: timp de compilare mai mare decat in cazul anterior, performanță îmbunătățita; aproape toate optimizările sunt activate, mai puțin „loop unrolling” si „function inlining”
- **-O3**: toate optimizările activate

Exemplu:

```
$ gcc <sursa> -O2 -o <fexecutabil>
```


Utilizarea bibliotecilor cu GCC (exemple)

- Link-editarea implicită în procesul de compilare a bibliotecii de funcții standard C++

```
$ gcc <prog.cpp> -Wall -lstdc++ -o <executabil>
$ g++ <prog.cpp> -Wall -o <executabil>
```

- Link-editarea implicită a bibliotecii de funcții matematice

```
$ gcc <mat.c> -Wall -lm -o <executabil_math>
```

- În mod asemănător, alte biblioteci pot fi link-uite în procesul de compilare folosind parametrii ”–l”

Automatizarea compilarii – GNU Make

- Make este un utilitar care utilizează un script, numit *Makefile*, pentru a determina automat secvența de paşi ce trebuie repetați deoarece nişte fisiere s-au modificat.
- În principal este utilizat pentru:
  - (re)compilarea programelor compuse din mai multe fişiere
  - testarea programelor.
- Script-ul *Makefile* este de obicei generat de către un alt script în funcție de configuratia masinii curente
- O compilare standard, pe o maşină Linux, are următoarele etape:

```
$ ./configure
$ make
$ make install
```


Un Makefile

- Un *Makefile* simplu

```
gccopt=-Wall
optimizare=-O2
ccompiler=gcc
cppcompiler=g++
progc:
	$(ccompiler) m.c $(gccopt) $(optimizare) -o progc
progcpp:
	$(cppcompiler) n.cpp $(gccopt) $(optimizare) -o progcpp
all: progc progcpp
clean:
	rm progc
	rm progcpp
```

Workshop

- Studiul unui executabil, studiul unei biblioteci
  - `file, ldd, nm`
- Analiza timpilor de execuție
  - `time`
- Studiul system-call-urilor
  - `strace`
- **Code proofing (valgrind)**
  - Memory leak detection
    - `tool=memcheck, tool=addrcheck`
  - Call analysis
    - `tool=callgrind, kcallgrind (application)`