Sisteme Încorporate
Îndrumar de Laborator

Cătălin Bogdan Ciobanu
George-Aurelian Feldioreanu
Alexandru Pușcașu
Robert-Nicolae Șolcă
Kertész Csaba Zoltán

TBD

Sisteme Încorporate
Îndrumar de Laborator

Cătălin Bogdan Ciobanu
George-Aurelian Feldioreanu
Alexandru Pușcașu
Robert-Nicolae Șolcă
Kertész Csaba Zoltán

TBD

Cuprins

1 Introducere **7**

2 Lucrarea 1 **13**

2.1 Exercițiul 1 16
2.2 Exercițiul 2 17
2.3 Exercițiul 3 17
2.4 Exercițiul 4 17
2.5 Exercițiul 5 18
2.6 Exercițiul 6 18

3 Lucrarea 2 **19**

3.1 Exercițiul 1 22
3.2 Exercițiul 2 22
3.3 Exercițiul 3 25

4 Asamblarea și dezasamblarea programelor **26**

4.1 Extragerea codului de asamblare 26
4.1.1 Exercițiul 1 29
4.2 Compilare și legare cod C și asamblare 30
4.2.1 Exercițiul 2 32
4.2.2 Exercițiul 3 33

4.3 Inline assembly 34
4.3.1 Exercițiul 4 36

5 Lucrarea 3 **37**
5.1 Exercițiul 1 40
5.2 Exercițiul 2 40

6 Lucrarea 4 **42**
6.1 Exercițiul 1 46
6.2 Exercițiul 2 47
6.3 Exercițiul 3 48

7 Lucrarea 5 **49**
7.1 Exercițiul 1 52
7.2 Exercițiul 2 53
7.3 Exercițiul 3 53
7.4 Exercițiul 4 53

8 Lucrarea 6 **54**
8.1 Exercițiul 1 60

9 Lucrarea 7 **62**
9.1 Exercițiul 1 69
9.2 Exercițiul 2 69

10 Lucrarea 8 **70**
10.1 Exercițiul 1 74
10.2 Exercițiul 2 76
10.3 Problema de 10 76

11 Criptografie **77**
11.1 Introducere 77
11.1.1 Criptografie 78
11.1.2 Recapitulare 79
11.1.3 XOR 79
11.1.4 Aplicații practice 80
11.2 Implementarea algoritmului 80
11.2.1 Exemplu 80
11.3 Criptarea unui fișiser 81
11.3.1 Indicații 81
11.4 Spargerea algoritmului - Metoda naivă 82
11.4.1 Indicații 82
11.5 Spargerea algoritmului - Sistem de ranking 82
11.5.1 Indicații 82

12 Hashing **84**
12.1 Introducere 84
12.1.1 Hashing 85
12.1.2 Secure Hash Algorithms 85
12.1.3 Aplicații practice 86
12.2 Utilizarea SHA256 86
12.2.1 Exemplu 87
12.3 Spargerea unei parole 87
12.3.1 Exemplu 87
12.3.2 Indicații 88
12.4 Spargerea parolelor în paralel 88

13 Pen-testing **89**
13.1 Introducere 90
13.1.1 Pen-testing 90
13.1.2 Raport 91
13.2 Spargerea unei aplicații web 92
13.2.1 Indicații 93
13.3 Scrierea unui raport 93
13.3.1 Indicații 93
13.4 Concluzii 94

Acronime **95**

Bibliografie **96**

Criterii de Evaluare

A. Compilare corectă a programelor din consolă fără warning-uri.
B. Execuția corectă a programului pe laboratorul virtual Raspberry Pi folosind multiple date de intrare care să acopere toate corner case-urile.
C. Management-ul corect al memoriei dinamice (alocare, eliberare), evitând memory leak-uri.
D. Evitarea overflow-urilor selectând tipul corect de date (număr de biți, întregi sau floating point, cu semn sau fără semn) și dimensiunea corectă a vectorilor.
E. Explicarea codului sursă pentru fiecare program realizat.
F. Indentarea corectă a codului sursă pentru programele realizate.
G. Adăugarea de comentarii explicative în codul sursă pentru programele realizate.
H. Explicarea algoritmilor utilizați în realizarea fiecărui program (funcționalitate, complexitate utilizând big-O notation).
I. Determinarea consumului de memorie pentru fiecare variabilă.
J. Cunoașterea tipurilor de date și a numărului de biți pentru fiecare variabilă tip de date folosit în programe.
K. Cunoașterea detaliilor tehnice de bază pentru plăcile Raspberry Pi 4 din laboratorul virtual (cantitate de RAM, model CPU, arhitectura CPU, număr de biți al CPU, număr de core-uri, sistem de operare).

L. Cunoașterea funcționalității compilatorului GNU gcc/g++ (sintaxă de compilare, flag-uri de debugging, flag-uri de optimizare).

1 Introducere

Introducere generala

- explicat la ce materia se foloseste, public tinta - cunostiinte prealabile - prezentare generala a modului de lucru si a lucrarilor - insistam pe ideea de laborator virtual

Raspberry Pi

Lucrul la acest laborator se va realiza pe dispozitive Raspberry Pi 4, Model B Rev. 1.4 (vezi fig. 1.1), cu următoarele specificații [1]:

- SoC Broadcom BCM2711, cu CPU Quad-core Cortex-A72 (ARM v8) 64-bit @ 1.5GHz
- 8GB RAM LPDDR4-3200
- Sistem de operare Raspbery PI OS, bazat pe Debian
- Conectivitate Wi-Fi, Ethernet, Bluetooth, USB,
- micro-HDMI, MIPI DSI, MIPI CSI, RCA

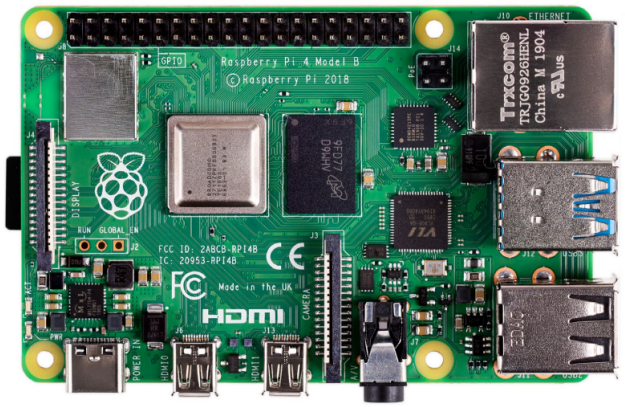

Figura 1.1: Placa Raspberry Pi 4, Model B [2]

Conectarea la laboratorul virtual Raspberry Pi

Conectarea la consola sistemului prin Secure Shell

Pentru accesul la dispozitiv prin această metodă este nevoie de un client SSH pe sistemul PC local de pe care va fi accesat laboratorul virtual, precum:

1. Pachetul de utilitare OpenSSH, disponibil pe o multitudine de sisteme de operare. De regulă, este preinstalat pe sisteme Unix-like (distribuții Linux, macOS) și ca utilitar opțional pe sisteme Windows.[3]

2. Utilitare third-party, precum *MobaXterm* [4], o suită completă pentru accesul la remote la sistemele de calcul.

Conexiunea se poate realiza folosind comanda:

```
ssh -p <port> <nume_utilizator>@<adresa_ip>
```

Aici, adresa IP a laboratorului virtual este reprezentată de domeniu, alături de numele de utilizator și portul asignat fiecărui student. Exemplu:

```
ssh -p 1234 popescui@exemplu.com
```

Conectarea la desktop environment-ul sistemului prin Remote Desktop Protocol

Se poate și este recomandat a se lucra și cu o interfață grafică, accesând desktop environment-ul sistemului print-o conexiune via RDP, proprietar Microsoft.

1. De pe sistemele cu Windows, se poate folosi aplicația Remote Desktop Connection (`mstsc.exe`), unde este specificat domeniul/adresa IP a sistemului accesat, alături de portul RDP asignat fiecărui student, ca în figura 1.2.

2. De pe alte sisteme de operare se pot folosi programe open-source. De exemplu, pe distribuții Linux, se pot folosi rdesktop, FreeRDP, etc.

După stabilirea conexiunii, se introduc credențialele de autentificare în dialogul afișat, având astfel acces la mediul desktop al sistemului. Se pot deschide programele instalate din taskbar-ul din partea superioară a ecranului (Terminal, editoare text, IDE-uri etc.)

Conectarea la sistem prin Visual Studio Code

Se mai poate lucra pe laboratorul virtual direct de pe sistemele locale folosind editorul de text Visual Studio Code, alături de extensia oficială

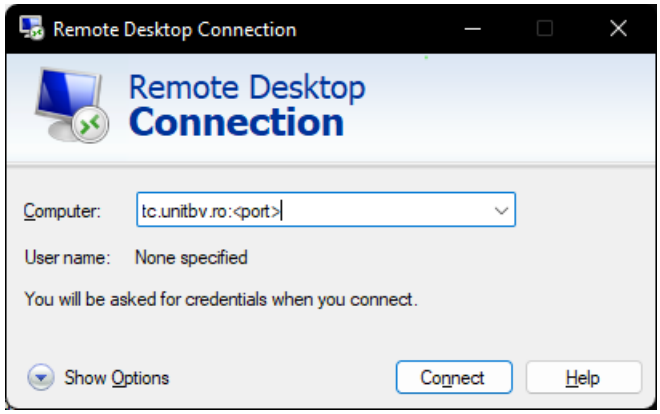

Figura 1.2: Căsuța dialog pentru accesul la Remote Desktop

Remote-SSH [5]. Aceasta permite accesul atât la sistemul de fișiere, permițând editarea codului sursă direct de pe sistemul local, cât și accesul la consolă.

Instalarea extensiei, exemplificare grafică în Figura 1.3:

1. se deschide tab-ul *Extensions*
2. se caută *remote ssh*
3. se deschide extensia *Remote - SSH*
4. se instalează această extensie

După instalarea extensiei, se configurează accesul astfel:

- O singura data:
    1. se apasă tasta *F1*
    2. se caută *connect to host*
    3. se va apăsa *Remote-SSH: Connect to host...*

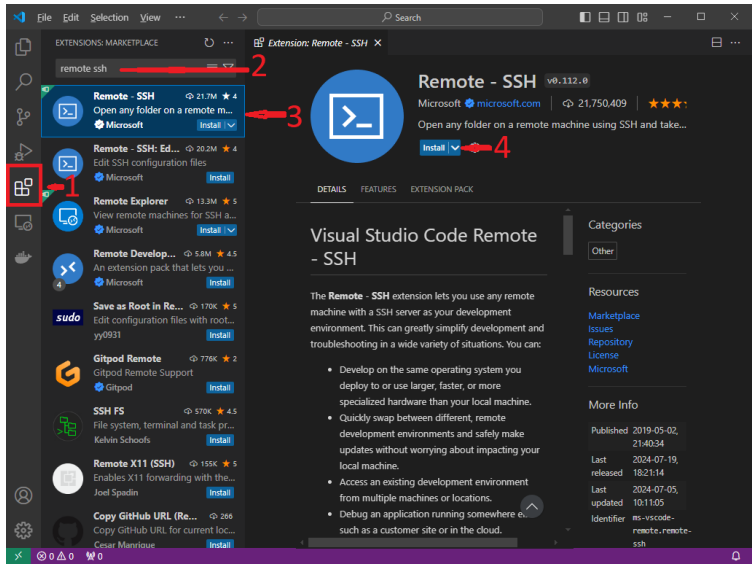

Figura 1.3: Pași instalare extensie SSH în VS Code

4. se selectează opțiunea + *Add new SSH host*
5. se introduce comanda de conectare cu SSH (*ssh -p <port> <username>@exemplu.com*) și se apasă tasta *Enter*
6. următoarea valoare se lasă valoarea default și se apăsă tasta *Enter*.

- La fiecare conectare:
  1. se apasă tasta *F1*
  2. se caută *connect to host*
  3. se va apăsa *Remote-SSH: Connect to host...*
  4. se selectează opțiunea + *Add new SSH host*
  5. se selectează opțiunea dorită

6. se introduce parola de autentificare și se selectează directorul de lucru dorit
7. **dacă se cere, alege Linux**

Daca nu se modifică workspace-ul în VS Code, la fiecare repornire configurația este realizată, doar trebuie reintrodusă parola.

Conectarea prin SSH, fără parolă

Pentru a nu reintroduce la fiecare conectare parola contului, *SSH* permite și conectarea cu cheie asimetrică, autentificarea realizându-se pe baza unei perechi de chei publică-privată. Pentru sistemele de operare Windows și Linux, pașii sunt descriși în cele ce urmează.

Pentru sistemele Windows

Folosim PowerShell:

```
    ssh-keygen
    type $env:USERPROFILE\.ssh\id_rsa.pub | ‘
ssh -p <port> <nume_utilizator>@<domeniu>
"cat >> .ssh/authorized_keys"
```

Pentru sistemele Linux

În terminal:

```
    ssh-keygen
    ssh-copy-id -i ~/.ssh/id_rsa.pub \
<nume_utilizator>@<domeniu> -p <port>
```

2 Lucrarea 1

Introducere

În acest laborator urmărim:

- compilare fișiere C/C++ cu utilitarul *GNU C Compiler - GCC*
- procesare argumente din linia de comandă în C/C++
- realizarea de fișiere Makefile

GNU C Compiler

Acest utilitar open-source poate fi folosit pentru compilarea programelor scrise în C/C++. Pentru codul sursă scris în C utilitarul este **gcc**, iar pentru codul sursă scris în C++ utilitarul este **g++**.

Pentru compilarea programelor este necesară o listă de fișiere de intrare, cu extensia *c/cpp*, și numele executabilului. Un exemplu de compilare:

```
gcc -o <nume_executabil> <nume_fișier>.c
g++ -o <nume_executabil> <nume_fișier>.cpp
```

După rularea comenzii, dacă există erori ele vor apărea explicate în consola, în caz contrar GCC va realiza un executabil, acesta poate fi rulat din consolă cu:

```
./<nume_executabil>
```

Argumente în linia de comandă

Argumentele de intrare sunt folosite pentru a transmite parametrii de configurație sau date de intrare pentru un program. Acest comportament este disponibil în orice limbaj de programare.

Pentru C/C++, argumentele sunt citite ca un tablou de șiruri de caractere accesate ca parametrii ai funcției `main`, de exemplu:

```
int main(int argc, char* argv[]) # pentru C++
void main(int argc, char **argv) # pentru C
```

Parametrul `argc` are numărul de argumente, iar `argv` un tablou de șiruri de caractere ce conțin aceste argumente (numele acestor parametrii este arbitrar, dare aceste două nume sunt uzuale).

Tipuri de date în C/C++

În cazul programelor realizate în C sau C++, se recomandă atenție la valorile maxime de reprezentare ale tipului de date folosit în calcularea și stocarea termenilor din șirul lui Fibonacci. În cazul tipurilor de date întregi, valoarea maximă este dependentă de dimensiunea de reprezentare în număr de biți. Dimensiunea în bytes al unui tip de date se poate afla cu

ușurință folosind operatorul `sizeof()`, iar valorile maxime se pot afla folosind constantele definite în header-ul `limits.h` [6].

Pentru a beneficia de un control sporit asupra lățimii în biți al tipurilor de date întregi și a asigura portabilitatea codului, se pot utiliza tipurile de date definite în header-ul C `stdint.h` [7], cu alias-ul `cstdint` în C++. Tipurile de date de bază sunt prezentate în tabelul 2.1, alături de constantele ce definesc valorile minime și maxime puse la dispoziție.

Tip	Descriere	Valoare minimă	Valoare maximă
int8_t	Număr întreg pe 8 biți, cu semn	$-(2^7)$ INT8_MIN	$2^7 - 1$ INT8_MAX
int16_t	Număr întreg pe 16 biți, cu semn	$-(2^{15})$ INT16_MIN	$2^{15} - 1$ IN16_MAX
int32_t	Număr întreg pe 32 de biți, cu semn	$-(2^{31})$ INT32_MIN	$2^{31} - 1$ INT32_MAX
int64_t	Număr întreg pe 64 de biți, cu semn (Dacă este suportat de sistem)	$-(2^{63})$ INT64_MIN	$2^{63} - 1$ INT64_MAX
uint8_t	Număr întreg pe 8 biți, fără semn	0	$2^8 - 1$ UINT8_MAX
uint16_t	Număr întreg pe 16 biți, fără semn	0	$2^{16} - 1$ UINT16_MA)
uint32_t	Număr întreg pe 32 de biți, fără semn	0	$2^{32} - 1$ UINT32_MAX
uint64_t	Număr întreg pe 64 de biți, fără semn (Dacă este suportat de sistem)	0	$2^{64} - 1$ UINT64_MAX

Tabelul 2.1: Tipurile de date de bază ale stdint

Pe lângă tipurile de date descrise în tabel, mai există tipurile de forma:

- `int_least`*N*`_t` și `uint_least`*N*`_t`, pentru numere întregi cu lățime egală cu cel puțin *N* biți, cu valorile posibile 8, 16, 32, 64.
- `int_fast`*N*`_t` și `uint_fast`*N*`_t`, pentru numere întregi cu lățimea în biți considerată cea mai rapidă pe platforma vizată, de valoare cel puțin *N*, cu valorile posibile 8, 16, 32, 64.

Fișiere Makefile

Fișierele Makefile sunt o metodă prin care se pot realiza automatizări complexe și utilizatorul să le apeleze printr-o singură comandă. Cel mai adesea ele sunt folosite pentru automatizarea compilării. Mai multe informații sunt disponibile pe siteul oficial [8]. Structura generală este:

```
<nume_etapă>: <dependință_1> <dependință_2>
    <comandă_1>
    <comandă_2>
```

Un exemplu de fișier Makefile pentru exercițiul 1:

```
ex1_c_compile:
    gcc ex1.c -o ex1_c

ex1: ex1_c_compile
    ./ex1_c

clean:
    rm -f ex1_c
```

Fișierul trebuie să se numească Makefile, trebuie să fim în același director cu acest fișier și etapele sunt rulate cu utilitarul make. Pentru a rula etapa *ex1* rulăm comanda: **make ex1**

2.1 Exercițiul 1

Conectați-vă la consola laboratorului virtual Raspberry Pi prin SSH și realizați un program în C și C++ care tipărește mesajul „Hello World” și rand nou în consolă.

2.2 Exercițiul 2

Conectați-vă la laboratorul virtual Raspberry Pi prin Remote Desktop și realizați un program în C sau C++ care afișează argumentele primite în linia de comanda în forma *Argumentul ... are valoarea: ....*

2.3 Exercițiul 3

Conectați-vă la laboratorul virtual Raspberry Pi prin Remote Desktop și realizați un program în C sau C++ care primește ca argument un număr întreg N (N < 90) și tipărește primii N termeni ai secvenței Fibonacci. Programul trebuie să verifice că se primește argumentul și că respectă constrângerea, în ambele cazuri se va afișa un mesaj corespunzător. Atenție la tipul de dată.

Pentru input-ul:

```
./<nume_executabil> 6
```

programul tipărește:

```
1 1 2 3 5 8
```

2.4 Exercițiul 4

Conectați-vă la laboratorul virtual Raspberry Pi și realizați un program în C/C++ care primește ca argument un număr întreg de N cifre (N < 100) și tipărește lista cifrelor numărului. Trebuie verificat că argumentul exista și să se afișeze un mesaj corespunzător.

Pentru comanda

```
./<nume_executabil> 2342
```

programul tipărește `[2,3,4,2]`.

2.5 Exercițiul 5

Conectați-vă la laboratorul virtual Raspberry Pi și realizați un program în C/C++ care citește de la tastatură un șir de caractere format din unul sau mai multe cuvinte separate prin unul sau mai multe spații și tipărește cuvintele, unul pe linie, într-un cadru dreptunghiular.

Pentru input-ul

```
 Ana   are   mere
```

programul tipărește:

```
********
* Ana  *
* are  *
* mere *
********
```

2.6 Exercițiul 6

Pentru exercițiile anterioare realizați un fișier Makefile care să compileze fiecare exercițiu în parte, să permită rularea fiecărui exercițiu individual și să permită ștergerea fișierelor generate.

3 Lucrarea 2

Introducere

În acest laborator urmărim:

- implementarea de aplicații cu mai multe surse
- compilarea aplicațiilor cu mai multe surse
- optimizarea programelor C/C++

Aplicații cu mai multe surse

Programele complexe au nevoie de un număr foarte mare de linii de cod, dacă sunt implementate într-un singur fișier, devin foarte greu de analizat și lucrat la ele. Folosirea unui singur fișier pentru tot codul sursă îngreunează și dezvoltarea unui program într-o echipă. Pentru aceasta, se alege împărțirea codului sursă în mai multe fișiere.

În C/C++ acestă împărțire se realizează realizând mai multe fișiere *header* (.h) și cod sursă (.c/.cpp). Fișierele header conțin semnăturile funcțiilor și se include în fiecare fișier care are nevoie de o funcție existentă în acel header. Fișierele cod sursă conțin implementările concrete ale funcțiilor din headere. Perechea de fișiere header și cod sursă au același nume, de exemplu: test.h și test.cpp.

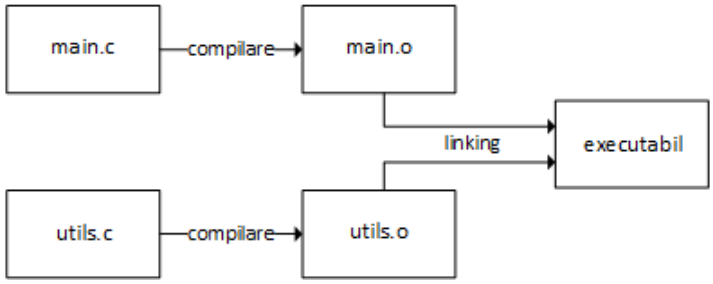

Figura 3.1: Exemplu simplu de compilare al unui program C format din două fișiere sursă

Deoarece fișierul header este inclus de mia multe ori, există riscul includerilor repetate, ceea ce duce le erori de compilare. Pentru rezolvarea aceste probleme se folosesc directive de precompilare, ca în exemplul următor:

```
#ifndef TEST_H
#define TEST_H
/* definiții de funcții */
#endif
```

O pereche fișier header și cod sursă se numește componentă. Fiecare componentă este compilată individual și combinarea lor se realizează în etapa de link-editare. Pentru aceasta componentele trebuiesc compilate cu argumentul *-c*, care semnalizează ca este o componentă, astfel nu primim erori ca lipsește funcția main. Pentru compilarea finală, se transmit la gcc, aceste binare intermediare și el le va link edita.

Bazându-ne pe exemplul din figura 3.1, compilarea unui astfel de program utilizând toolchain-ul GCC poate fi făcută cu comenzile:

```
# compilare sursa main, fara link-editare
```

```
gcc -o main.o -c main.c
# compilare utils, fara link-editare
gcc -o utils.o -c utils.c
```

generând fișierele obiect `main.o` și `utils.o`, care pot fi apoi legate într-un executabil cu comanda:

```
# link-editarea obiectelor anterioare
gcc -o <nume_executabil> main.o utils.o
```

Profiling

Pentru a optimiza un program trebuie să identificăm fiecare etapă a lui cat timp consumă, acest concept se numește *profiling*. GCC dispune de astfel de funcționalitate și analizează fiecare apel de funcție și la final realizează un raport. Pentru a avea acestă funcționalitate codul sursă trebuie compilat cu directiva *-pg*, această va insera instrucțiuni suplimentare pentru analiză.

Niveluri de optimimzare

În mod normal GCC va realiza o translație unu-la-unu a codului sursă în limbaj de asamblare. Dar compilatorul este capabil să realizeze optimizări de două tipuri: de reducere a dimensiuni executabilului sau de optimizarea a timpului de rulare. Pentru compilare optimizată se folosește directiva *-O* urmată de nivelul de optimizare, care este între 0 și 3. Nivel 0 - fără optimizări, nivel 1 - optimizare timp de rulare, nivel 2 - optimizare dimensiune executabil și nivel 3 optimizare timp de rulare și dimensiune executabil.

3.1 Exercițiul 1

Conectați-vă la laboratorul virtual Raspberry Pi și realizați un program în C sau C++ care:

1. Conține următoarele funcții într-un modul dedicat (fișier sursă, însoțit de un header cu declarațiile/prototipurile funcțiilor):
   - `generate`: primește un număr întreg `n` și returnează un șir de `n` valori întregi inițializate cu valori pseudoaleatoare.
   - `sort_<tip_sortare>`: primește un șir și returnează un șir nou sortat, cu un algoritm la alegere (exclus funcții de sortare din biblioteci). Exemple de algoritmi: `bubble sort, quick sort, merge sort, insertion sort, etc.`).
   - `merge`: primește două șiruri sortate, le contopește (interclasează) și returnează șirul nou obținut.
   - `print`: primește un șir și îl afișează în consolă.
2. Un program principal (în funcția `main`) care, utilizând funcțiile scrise, generează două șiruri de N elemente, sortează cele două șiruri și le contopește într-un singur șir de dimensiune 2N. Numărul de elemente N va fi citit ca o constantă dintr-un alt header `constants.h` și afișat la începutul programului.

3.2 Exercițiul 2

Pornind de la exercițiul precedent:

1. Automatizați procesul de compilare scriind un fișier Makefile cu opțiuni de compilare (build) și curățare fișiere temporare (de ex. fișiere obiect). De exemplu:

```
utils:
    gcc -o utils.o -c utils.c

build: utils
    gcc -o main.o -c main.c
    gcc -o <nume_executabil> main.o utils.o

clean:
    rm -f *.o  #ștergerea obiectelor din director
```

   Compilarea va fi realizată cu ajutorul utilizatului GNU make, folosind comanda:

```
make build
```

2. Eliminați mesajele ce tipăresc șirurile și ajustați numărul de elemente astfel încât timpul de rulare al programului să fie de minim 10 secunde.
   Timpul de rulare poate fi analizat folosind comanda:

```
time ./<executabil>
```

3. Realizați profilul de rulare al programului compilând programul cu flag-ul de compilare `-pg`. Pentru analiza programului este nevoie de un fișier binar `gmon.out`, acesta se obține prin rularea executabilului cel puțin odată. Analiza executabilului se realizează cu programul gprof, comoda de rulare este:

```
gprof <executabil> gmon.out
```

Acesta poate fi salvat într-un fișier text cu ajutorul operatorului *bash* de redirecționare a output-ului într-un fișier nou, urmând exemplul:

```
gprof <executabil> gmon.out > profil.txt
```

4. Actualizați Makefile-ul astfel încât să aibă o opțiune de compilare `profile` cu flag-ul generare a profilului inclus (de ex. `make profile`).
   Se poate realiza utilizând o variabilă ce reține lista de flag-uri, pentru a evita repetarea comenzilor, astfel:

```
FLAGS=   #inițializare flag-uri utilizate

utils:
    gcc $(FLAGS) -o utils.o -c utils.c

build:
    gcc $(FLAGS) -o main.o -c main.c
    gcc $(FLAGS) -o <executabil> main.o utils.o

profile: FLAGS+=-pg  #concatenare flag nou
profile: build
```

3.3 Exercițiul 3

Continuați exercițiul precedent:

1. Mutați funcția de sortare într-un modul sort (fișier sursă + header) separat și actualizați Makefile-ul. Adăugați, în modulul creat anterior, încă o nouă funcție care să implementeze un alt algoritm de sortare și sortați unul din șirurile din programul principal utilizând noua funcție.

2. Analizați noul timp de rulare al programului și refaceți profilul de rulare; dacă este cazul, ajustați numărul de elemente astfel încât timpul de rulare să fie de minim 20 secunde.

3. Recompilați programul, adăugând succesiv la opțiunile de compilare următoarele flag-uri de optimizare automată:

   ```
   -O0
   -O1
   -O2
   -O3
   ```

   Salvați pentru fiecare flag profilul de analiză. Analizați profilurile de optimizare, ce concluzii puteți trage? Adăugați la Makefile o opțiune de compilare pentru performanță maximă.

4 Asamblarea și dezasamblarea programelor

Limbajul de asamblare este o reprezentare directă în mod text a instrucțiunilor din codul binar ce se execută pe procesor. Aceasta este bazat pe un set de mnemonice pentru instrucțiuni și folosește direct modurile de adresare oferite de procesor (registre, adresare directă, indirectă, operații adiționale pe adrese). Limbajul de asamblare permite o urmărire într-un format inteligibil pentru utilizatorul uman a execuției programului pe procesor și dacă e nevoie o utilizare mai eficientă a facilităților procesorului.

4.1 Extragerea codului de asamblare

Compilatorul `gcc` poate fi oprit în orice fază a compilării, pentru a putea obține fișierele din etapele intermediare. Acest lucru am observat în capitolul precedent, unde am folosit flagul `-c` pentru a opri compilarea înainte de legare și a obține codul obiect. Putem însă să oprim compilarea înainte de conversia în codul binar la etapa de asamblare și să obținem codul sub forma de mnemonice de limbaj de asamblare. Pentru a realiza acest lucru

folosim flag-ul `-S`.

Astfel pentru codul sursă (`hello.c`) cu conținutul:

```
#include <stdio.h>

int main()
{
   printf("Hello world\n");
   return 0;
}
```

executând comanda:

```
▶ gcc -S -o hello.S hello.c
```

vom obține fișierul `hello.S` (specificat cu flag-ul `-o`) care conține următorul cod:

```
   .arch armv8-a
   .file  "hello.c"
   .text
   .section .rodata
   .align 3
.LC0:
   .string "Hello world"
   .text
   .align 2
   .global main
   .type main, %function
main:
.LFB0:
   .cfi_startproc
   stp x29, x30, [sp, -16]!
   .cfi_def_cfa_offset 16
   .cfi_offset 29, -16
   .cfi_offset 30, -8
   mov x29, sp
   adrp  x0, .LC0
   add x0, x0, :lo12:.LC0
   bl puts
   mov w0, 0
   ldp x29, x30, [sp], 16
```

```
25    .cfi_restore 30
26    .cfi_restore 29
27    .cfi_def_cfa_offset 0
28    ret
29    .cfi_endproc
30 .LFE0:
31    .size main, .-main
32    .ident "GCC: (Debian 8.3.0-6) 8.3.0"
33    .section .note.GNU-stack,"",@progbits
```

Codul de asamblare în cazul nostru va fi alcătuit de instrucțiunile corespunzătoare funcției `main` și informații adiționale necesare pentru fazele următoare pentru a putea lega corect simbolurile externe. Acestea din urmă pot fi identificate prin cuvintele cheie care încep cu "`.`". Funcția în sine este alcătuit din doar 8 instrucțiuni:

- `stp` (linia 15) urmat de `mov x29, sp` de pe linia 19 salvează pe stivă registrele care urmează să fie modificate în funcție și modifică vârful stivei
- `adrp` urmat de `add` (liniile 20-21) o structură de stocare unei adrese de memorie independent de poziție, în cazul nostru adresa șirului de caractere *Hello world*. E nevoie de această operație pentru că în această fază compilatorul încă nu știe unde va fi poziționat șirul de caractere în codul final.
- `bl puts` face apel la funcția `puts` din biblioteca standard. Aici putem observa faptul că în codul C original am făcut apel la funcția `printf`, dar compilatorul văzând că parametrul la `printf` este un simplu șir de caractere fără nici o formatare va optimiza automat înlocuind cu funcția mai eficientă de tipărire pe ecran
- `mov w0, 0` (linia 23) setează valoarea returnată din funcție la `0`

- `ldp` (linia 24) restaurează registrele salvate la începutul funcției
- `ret` întoarce din funcție

Trebuie notat că opțiunea `-s` oprește compilarea înainte de asamblare, deci nu vom mai obține și fișierul executabil. Dacă vrem obținem amândouă într-un singur pas, putem folosi de flaguri dedicate asamblorului pentru a genera un listing:

```
▶ gcc -o hello hello.c -Wa,-adhlns=hello.lst
```

Fișierul de listing generat (`hello.lst`) va fi asemănător cu fișierul de asamblare, dar conține în plus și o reprezentare hexazecimală a instrucțiunilor binare, ceea ce înseamnă că aceasta poate fi folosit doar pentru a studia codul, dar nu poate fi folosit ca un program în limbaj de asamblare sine stătător.

La acest listing mai apare și posibilitatea a avea codul de asamblare și codul C întrețesut, pentru a identifica mai ușor anumite secțiuni de cod. Acest lucru se activează cu opțiunea includere a simbolurilor de depanare `-g`:

```
▶ gcc -o hello hello.c -Wa,-adhlns=hello.lst -g
```

4.1.1 Exercițiul 1

Studiați codul de asamblare pentru programele scrise la lucrările de laborator anterioare. Încercați să identificați elementele din ABI: modul de pasare a argumentelor, stocarea variabilelor, registre salvate de funcție și de apelant.

Descrierea completă a setului de instrucțiuni ARM64 puteți găsi la https://developer.arm.com/documentation/ddi0602/latest

4.2 Compilare și legare cod C și asamblare

Codul de asamblare (creat prin întreruperea procesului de compilare sau scris manual) poate fi legat în continuare într-un executabil folosind `gcc`. Aceasta detectează pe baza extensiei dacă este vorba de cod C (`.c`) sau de asamblare (`.S`) și execută pașii corespunzători (compilare/asamblare) pentru a crea executabilul.

Astfel după generarea fișierului de asamblare `hello.S` din exemplul precedent puteți obține fișierul executabil prin comanda:

```
▶ gcc -o hello hello.S
```

Se poate lega împreună și diferite surse (C și asamblare) la fel cum se leagă și codul C distribuit pe mai multe fișiere sursă.

De exemplu, având fișierul sursă `add.c`:

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

extern int add(int a, int b);

int main(int argc, char** argv)
{
   if (argc != 3)
   {
      printf("Two operands required\n");
      exit(1);
   }

   printf("%d\n", add(atoi(argv[1]), atoi(argv[2])));

   return 0;
}
```

respectiv fișierul în asamblare `add.S`

```
   .global add
   .type add, %function
```

```
3 add:
4     add w0, w0, w1
5     ret
```

se poate compila împreună cu comanda:

```
▶ gcc -o add add.c add.S
```

Acest program primește doi parametri din linia de comandă, convertește șirurile de caractere în numere și apelează funcția `add` pentru a le aduna. Funcția este declarată ca funcție externă și este definită într-un fișier separat scris în limbaj de asamblare.

Pentru a putea scrie un cod în asamblare care să fie apelat din cod C sau invers, trebuie avut grijă ca cele două să folosească aceleași convenții de lucru cu registrele interne ale procesorului. Această convenție este definită în Application Binary Interface (ABI).

În exemplul nostru am folosit de faptul că primele patru argumente a unei funcții sunt transmise în registrele x0–x7 (restul argumentelor fiind așezați pe stivă dacă e cazul) iar valoarea returnată din funcție este pusă în registrul x0.

Pe lângă plasarea argumentelor este important și comportamentul compilatorului în privința registrelor. Astfel sunt definite registrele care pot fi folosite liber de către funcții (*caller-saved*), compilatorul va avea grijă să le salveze înainte de apelarea funcției, și registrele care nu pot fi alterate de către funcții (*callee-saved*), acestea dacă sunt folosite în funcția apelată vor trebuie salvate și restaurate înainte de întoarcerea din funcție.

În cazul ARM64 folosind EABI (Embedded ABI), registrele x0–x7 sunt folosite pentru parametri, dar în același timp pot fi modificate și în funcția apelată (conform standardului C, parametri sunt tratați ca variabile locale în funcție). Pe lângă acestea x9–x15 sunt registre temporare ce pot fi modificate în funcții. Registrele x19–x28 sunt de tip callee-saved,

deci nu pot fi modificate (sau trebuie restaurate). Restul registrelor au semnificații speciale pentru compilator și nu ar trebui atinse de codul funcției. O excepție ar fi x30, care este tratat ca LR (Link Register), registru folosit pentru salvarea adresei de revenire din funcție, și evident nu trebuie modificat pentru a putea reveni din funcție, dar în cazul în care și funcția apelată vrea să apeleze o subrutină, atunci va trebui să folosească acest registru, asigurându-se că valoare lui este restaurată înainte de întoarcere din funcție.

4.2.1 Exercițiul 2

Implementați în asamblare o funcție care va umple un vector cu elementele din șirul lui Fibonacci, care poate fi apelat dintr-un program C:

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

extern void fib(int n, int v[]);

int main(int argc, char** argv)
{
   if (argc != 2)
   {
      printf("Usage: fibonacci N\n");
      exit(1);
   }

   int n = atoi(argv[1]);
   int v[n];
   fib(n, v);
   for (int i=0; i<n; i++)
   {
      printf("%d%c",v[i], i==n-1?'\n':' ');
   }
   return 0;
}
```

Provocare adițională: comparați vitezele de execuție a codului de asamblare cu un cod similar de C, și încercați elaborarea unui cod de asambla care să se execute mai rapid decât codul compilat din C.

4.2.2 Exercițiul 3

Pornind de la următorul program C:

```c
#include <stdio.h>
#include <stdlib.h>
#include <stdint.h>

void print_bits(uint32_t x)
{
   uint32_t mask = 1ul << 31;
   while (mask)
   {
      printf("%d", (mask & x) ? 1 : 0);
      mask >>= 1;
   }
}

int main(int argc, char** argv)
{
   if (argc != 2)
   {
      printf("Usage: bits N\n");
      exit(1);
   }

   print_bits(atoi(argv[1]));
   printf("\n");

   return 0;
}
```

implementați rutina de afișare a biților în asamblare.

4.3 Inline assembly

Deși implementarea în asamblare a programelor poate fi mai eficientă decât programele scrise în C, dezvoltarea în asamblare este mai anevoioasă și poate avea costuri mai mari de dezvoltare. Din această cauză și datorită optimizărilor foarte bune ale compilatoarelor moderne, scrierea programelor în întregime în asamblare nu mai este justificat.

Totuși există situații în care compilatorul C nu reușește să obțină codul cel mai bun pentru o bucățică mică din cod, care însă să prezinte o gâtuire de performanță în timpul executării sau să scadă din precizia dorită a operațiilor. Pentru a rezolva aceste situații compilatorul C permite ca programul C să conțină și linii de cod de asamblare intercalate (inline assembly).

Sintaxa pentru a scrie astfel de cod este prin folosirea cuvântului cheie `asm`:

```
asm("instrucțiuni asamblare" : intrări : ieșiri : efecte colaterale);
```

Instrucțiunile de asamblare se vor scrie în șirul de caractere (primul parametru), exact în forma în care sunt scrise și în programele scrise în asamblare. Trebuie avut grijă însă ca șirul este tratat ca un șir de caractere C, linia nouă de exemplu trebuie marcat cu `\n`. Intrările și ieșirile specifică ce registre vor fi folosite ca intrare respectiv ieșire de instrucțiunile în asamblare. Cu acestea pot fi specificate modul în care compilatorul să aranjeze variabilele C astfel încât să fie accesibile și în asamblare. Iar efectele colaterale precizează ce alte modificări vor apare (registre modificate, scriere în memorie etc.) în timpul execuției instrucțiunilor astfel încât compilatorul să țină cont de acestea în timpul optimizărilor pe care efectuează pe restul codului.

Instrucțiuni de asamblare în general vor fi folosite în cazul în care e nevoie de o operație care nu are corespondent în C sau în cazul în care

codul C scris (generând multiple instrucțiuni) poate fi înlocuit cu o altă instrucțiune în asamblare mai eficientă. Acest lucru se întâmplă de cele mai multe ori la folosirea unor extensii ale procesorului (cu e de exemplu coprocesorul de vectorizare SIMD, care oferă instrucțiuni accelerate care nu pot fi descrise în C în mod trivial).

Un exemplu pentru primul caz este instrucțiunea `NOP` – No Operation, care nu are corespondent în C. Dacă vrem să facem o temporizare precisă la nivel de tacte de ceas putem folosi linia:

```
asm volatile ("nop");
```

Cuvântul cheie `volatile` împiedică optimizatorul C să mute instrucțiunea după propria voie.

Un alt exemplu în care o singură instrucțiune ARM poate înlocui o serie mai lungă de instrucțiuni C este operație de inversare a ordinii biților. Aceasta este o operație frecvent întâlnită în prelucrări de semnale (pe care veți folosi într-o lucrare ulterioară). Pentru a implementa această operație avem nevoie de următorul cod C (având numărul în variabila `a` și rezultatul inversării va fi stocat în `r`):

```
uint32_t r = 0;
for (int i=0; i < 32; i++)
if (a & 1 << i)
   r |= 0x80000000u >> i;
```

Aceasta va rezulta într-un cod de asamblare asemănător cu următoarea secvență (presupunând că numărul este deja în registrul `w4` iar rezultatul va fi stocat în registrul `w0`):

```
   mov w1, 0
   mov w0, 0
   mov w2, 1
   mov w3, -2147483648
.L1:
   lsl w4, w2, w1
```

```
7     tst w4, w19
8     beq .L7
9     lsr w4, w3, w1
10    orr w0, w0, w4
11  .L2
12    add w1, w1, 1
13    cmp w1, 32
14    bne .L8
```

Pe de altă parte arhitectura ARM64 oferă instrucțiunea `RBIT` care realizează exact acest lucru. Astfel cele trei linii de cod C (care rezultă în 12 instrucțiuni în asamblare) putem înlocui cu o singură linie de inline assembly (rezultând într-o singură instrucțiune în asamblare):

```
1 asm ("rbit %w0, %w1" : "=r" (r) : "r" (a) : );
```

În corpul instrucțiunii s-a folosit sintaxa `%w0` respectiv `%w1` pentru a specifica registrele de 32 de biți (de la `w`) alocați pentru ieșire respectiv pentru intrare de către compilator. În câmpul pentru ieșire este specificat că este vorba de un registru în care se află variabila `r`, iar în câmpul pentru intrare este specificat registrul în care se află variabila `a`.

Compilatorul va înlocui automat cu registrele corespunzătoare, de exemplu putem ajunge la un cod

```
1     rbit w0, w4
```

presupunând aceleași registre ca în exemplul precedent, adică `w4` conține numărul de la intrare iar `w0` va conține numărul cu biți inversați în ordine.

4.3.1 Exercițiul 4

Adăugați exemplele pentru inversarea biților la programul din exercițiul precedent și observați rezultatul, respectiv codul de asamblare generat.

5 Lucrarea 3

Introducere

În acest laborator urmărim:

- alocarea dinamică în C și C++
- automatizări pentru analiza performanțelor programelor

Alocare dinamică

Aplicațiile reale nu cunosc memoria necesară în etapa de implementare, pentru aceasta folosesc alocarea dinamică, care permite alocarea de memorie atunci când aceasta este necesară. Alocarea dinamică difera în C și C++. Alocarea din C poate fi folosită și în C++, trebuie să se folosească același stil de alocare, deoarece fiecare bibliotecă are propriul ei sistem prin care ține evidența memoriei alocate.

Alocare dinamică în C

În C pentru alocarea dinamică avem nevoie de biblioteca *stdlib*. Ea pune la dispoziție două funcții: *malloc* și *free*. Funcția *malloc* alocă dinamic o zonă continuă de byți, primește ca parametru numărul de byți ce trebuiesc

alocați și returnează un pointer generic cu acea zona de memorie. Funcția *free* șterge memoria alocată dinamic, primește ca și parametru pointerul catre o zona de memorie care a fost alocată cu *malloc*. Ca și exemplu v-om aloca un vector cu 10 elemente de tip întreg:

```
#include <stdlib.h>
/* ... */
int main()
    int *p = (int *)malloc(10 * sizeof(int));
    /* ... */
    free(p);
    return 0;
}
```

Alocare dinamică în C++

În C++ există 2 perechi de operatori: pentru alocarea de matricii și pentru alocarea de date simple. Pentru alocarea de obiecte simple se folosește *new* și pentru ștergerea lor se folosește *delete*, la alocare se primește tipul de dată, iar la ștergere se trimite doar pointerul. La alocarea matricilor se folosește perechea *new[]* și *delete[]*, la alocare între parantezele drepte se specifică dimensiunea vectorului. Ca și exemplu v-om aloca un vector cu 10 elemente de tip întreg:

```
#include <new>
/* ... */
int main()
    int *p = new[10] int;
    /* ... */
```

```
    delete[] p;
    return 0;
}
```

Analiza timpului de rulare

Pentru o aplicație trebuie să alegem un algoritm care rulează rapid, pentru aceasta trebuie să rulăm pe date de mai multe dimensiuni și să monitorizăm timpul de rulare. Putem să pornim de la soluția descrisă în laboratorul anterior, gprof, salvăm rezultatele și după le extragem manual. Dar dacă vrem să simplificăm această sarcină, vom automatiza această sarcină în C++.

Pentru a calcula timpul de rulare a unei funcții, putem folosii biblioteca *chrono*, care ne permite să accesăm ceasul sistemului și astfel din diferența lor extragem timpul de rulare.

```
#include <algorithm>
#include <chrono>
#include <iostream>
#include<vector>
using namespace std;
using namespace std::chrono;

int main()
{
    cout << "Dimension,Time[ms]" << endl;
    vector <int> dims({10, 100, 100});
    for (const auto& dim: dims) {
        vector<int> values(dim);
```

```
        auto start = high_resolution_clock::now();
        sort(values.begin(), values.end());
        auto stop = high_resolution_clock::now();

        auto duration = duration_cast<microseconds>
            (stop - start);

        cout << dim << "," << duration << endl;
    }
    return 0;
}
```

În exemplul anterior am prezentat un program care rulează în mod automat sortarea mai multor șiruri de diferite dimensiuni. Rezultatele sunt afișate în format *CSV* (Comma Separated Values). Acest format poate fi salvat într-un fișier cu extensia .csv și deschis în orice editor de foi de calcul.

5.1 Exercițiul 1

Modificați codul din laboratorul anterior să folosească alocarea dinamică.

5.2 Exercițiul 2

Realizați o analiză comparativă între cinci algoritmi de sortare, patru aleși de voi și implementarea din STD[9]. Analiza trebuie realizată pe 10 dimensiuni și valori peste 1000 de elemente și cu diferențe mari între dimensiuni.

Pentru a se asigura calitatea comparației, toți algoritmii trebuie să folosească același șir de intrare. La final realizați un table, pornind de la capul de tabel 5.1 și un grafic cu rezultatele obținute.

Dimensiune	Sort1[ms]	Sort2[ms]	Sort3[ms]	Sort4[ms]	SortSTD[ms]

Tabelul 5.1: Cap tabel

6 Lucrarea 4

Introducere

Memoria Cache

În Figura 6.1 este prezentată ierarhia de memorie într-un calculator, dar începe să devină valabilă și în sistemele incorporate. Cea mai rapidă memorie este reprezentanta de regiștrii procesorului.

Ca și viteză de acces urmează memoria cache, care și ea este împărțită în mai multe niveluri. Memoria cache cea mai apropiată de procesor se numește L1, de la *Level 1*, la rândul lui este împărțita în 2 tipuri, o memorie de date (L1D) și o memorie de instrucțiuni (L1I), această împărțire există deoarece datorită pipeline-ului aceste două operații la memorie se pot întâmpla concomitent.

În memoria cache mai există 1-2 niveluri, L2 este comun, și Raspbeery Pi 4B dispune de acest nivel de cache. L3 este comun în procesoarele de server. Memoria cache este amplasată cel mai des foarte aproape de procesor. Spre deosebire de L1, cache L2 dispune de o capacitate ridicată de stocare, dar un timp de acces mai redus.

Regiștrii și memoria cache sunt de dimensiuni reduse, datorită costului ridicat de realizare a lor în siliciu. Memoria cache ajută la reducerea timpului de execuție a unui program, reducând timpul necesar operațiilor

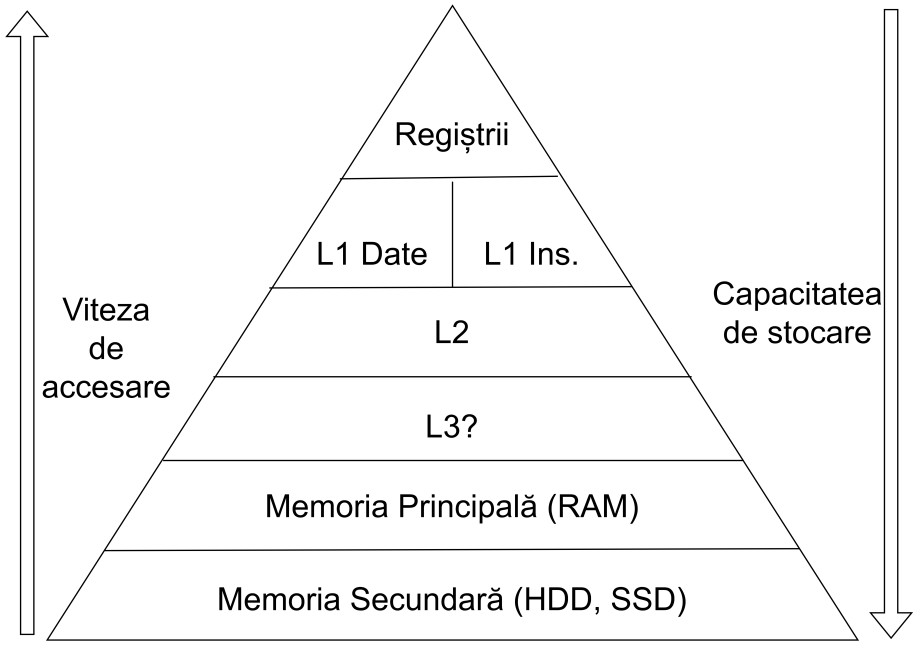

Figura 6.1: Ierarhia de memorie într-un calculator

cu memoria principală.

O memorie cache este organizată pe linii, o linie stochează mai mulți byți consecutivi. Deoarece memoria cache are dimensiuni mult mai mici decât memoria principală, este necesară o funcție matematică de corelație, pentru o linie sunt asignate mai multe adrese. Dacă se cer consecutiv adrese care corespund aceleiași linii, în memoria cache va apărea o înlocuire.

Cum în memoria cache o linie stochează mai mulți byți consecutivi, când se cere un byte, un interval, care conține și byteul cerut este încărcat în memorie. Dacă imediat vom cere un byte vecin, deoarece el se află în memoria cache, îl obținem repede. Acest comportament se numește locacitate, și se folosește ca optimizare a timpului de rulare.

Pentru a afla dimensiunea nivelurilor de cache putem folosii comanda *lscpu* în sistemele de operare Linux, care are o secțiune dedicată memoriei cache:

```
$ lscpu
Architecture:            x86_64
  CPU op-mode(s):        32-bit, 64-bit
  Byte Order:            Little Endian
CPU(s):                  4
...
Caches (sum of all):
  L1d:                   64 KiB (2 instances)
  L1i:                   64 KiB (2 instances)
  L2:                    512 KiB (2 instances)
  L3:                    3 MiB (1 instance)
...
```

În Figura 6.2 este prezentat un exemplu pentru locacitate. Este o memorie principala, cu timp de acces 400 de tacte de procesor, o memorie L2, cu o linie de 4 byți și un timp de acces 20 de tacte de procesor și memoria L1 are linii care stochează 2 byți. La început memoria cache este goală.

Dacă procesorul cere data de la **a[0][0]*, vor trece 421 de tacte de ceas până o obține acest byte (vezi calea cu linii albastre) deoarece trebuie să o aducă din memoria principală. Când răspunsul este returnat se umplu nivelele de cache. Dacă imediat cerem **a[0][1]* îl obținem într-un tact de ceas, deoarece el este prezent în L1. Dacă cerem ulterior **a[0][3]* îl obținem în 21 de tacte de ceas, deoarece aceste date trebuiesc aduse din L2 (vezi calea roșie). Dacă ulterior am cere **a[1][3]*, vor fi necesare dinou 421 de tacte de ceas, deoarece aceste date sunt în memoria principală.

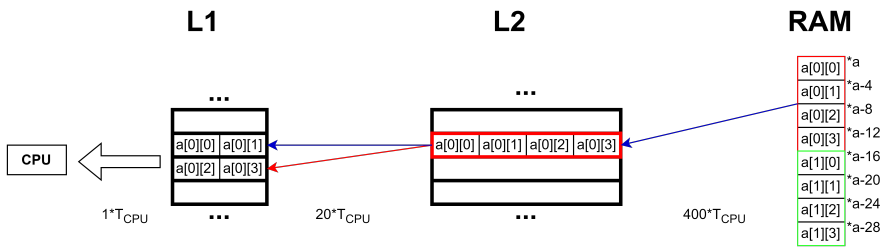

Figura 6.2: Exemplu pentru locacitatea memoriei cache

Unrolling

Procesoarele moderne folosesc tehnici de pipeline pentru optimizarea volumului de date procesate. În această tehnică fiecare etapă contribuie cu puțin la executarea unei instrucțiuni.

În astfel de procesoare exista 3 tipuri de hazard: funcțional, de date și de control. Hazardul funcțional este cauzat de dublul acces la memorie din doua etape a le procesorului, etapa de fetch și cea de memorie. Aceasta este rezolvată prin cele două tipuri de cache L1, de date și de instrucțiuni.

Hazardul de date apare atunci când o instrucțiune are nevoie de date calculate de instrucțiunea anterioară, trebuie să aștepte ca tot pipelineul să se termine. Acest tip de hazard este rezolvat de compilator în mare parte.

Hazardul de control apare la instrucțiunile de tip IF, deoarece adresa următoarei instrucțiuni diferă în funcție de valoarea de adevăr a expresiei. Acest tip de hazard nu poate fi eliminat complet.

O soluție pentru hazardul de control este reducerea numărului de instrucțiuni de control în program. O buclă de tip FOR execută de fiecare dată o instrucțiune de control de fiecare dată când execută corpul buclei. Aici soluția este să nu mai avem un pas egal cu unu și mult mai mare. Pentru că pasul este mult mai mare corpul buclei se repetă de mai multe ori și pentru

valorile sărite. De exemplu:

```
int a[12];
int sum = 0;
// fara unrolling
for(int  i = 0; i < 12; ++i)
    sum += a[i];

sum = 0;
// cu unrolling
for(int  i =0; i < 12; i+=4) {
    sum += a[i];
    sum += a[i + 1];
    sum += a[i + 2];
    sum += a[i + 3];
}
```

Pe baza exemplului, fără unrolling avem de executat 12 IF-uri, cu unrolling mai trebuie să executăm 3 IF-uri. La un procesor cu 5 etape, am salvat aproximativ 45 de tacte de ceas.

6.1 Exercițiul 1

Conectați-vă la laboratorul virtual Raspberry Pi și realizați un program în C sau C++ care calculează produsul matriceal al două matrici pătratice de valori reale reprezentată tipuri de date pe 32 de biți. Implantați înmulțirea într-o funcție și salvați-o pentru mai târziu.

Indicații

Se poate utiliza algoritmul clasic, folosind bucle imbricate.

Deoarece ulterior vom avea nevoie de matrici de dimensiuni mari, se recomanda alocare dinamică a matricilor, aceasta ajuta și la transmiterea lor către funcții. Exemplu:

```
#define n 10
int32_t* m = new int32_t*[n * n];
```

6.2 Exercițiul 2

Optimizați programul anterior folosind avantajele oferite de memoria cache. Continuați exercițiul precedent, modificând operațiile cu memoria, pentru a beneficia cat mai mult de locacitatea memoriei cache. Implantați înmulțirea într-o funcție și salvați-o pentru mai târziu. Adăugați argumentarea ca și comentariu în prima linie din corpul funcției. Folosind matrici de dimensiuni mari, de exemplu 8192×8192, și comparați implementarea naivă, față de cea optimizată cu memoria cache.

Indicații

Pentru a ne folosii de această proprietate, trebuie să schimbam modul de parcurgere a matricilor. Parcurgerea de dorit este parcurgerea linie cu linie. Cu toate că, complexitatea rămâne aceiași, $O(n^3)$, varianta optimizată este de aprox. $\times 2$ mai rapidă.

6.3 Exercițiul 3

Continuați exercițiul precedent adăugând și loop unrolling. Comparații timpii de rulare între diverse dimensiuni de desfacere a buclei. Adăugați argumentarea ca și comentariu în prima linie din corpul funcției.

Indicații

Rulați pe 10 dimensiuni de matrici, realizați un tabel cu timpii de rulare și realizați un grafic de tip linie. Alegeți dimensiunile de unrolling, să fie divizibile între ele, de exemplu: 2, 4, 8. Alegeți dimensiunile astfel încât să fie multipli de valorile de unrolling alese.

7 Lucrarea 5

Introducere

Auto-paralelizare

O soluție pentru optimizarea timpului de rulare îl reprezintă împărțirea programului pe mai multe threaduri care să lucreze in paralel. Acest lucru este realizabil dacă între threaduri nu există dependințe, un thread calculează date care sunt necesare altuia, sau dacă nu există scrierea unei variabile de mai multe threaduri.

Lucrul tu threaduri se poate realiza prin două metode: explicită și automată. Prin metoda implicită programatorul are de realizat programul pentru un thread și controlul threadurilor. Metoda automată presupune utilizarea de biblioteci special realiate pentru astfel de sarcini.

O astfel de biblioteca este *OpenMP* [10], ea include directive de compilare și funcții specifice. Este o bibliotecă ce rulează pe toate sistemele de operare și este din surse publice. Pentru folosirea acestei biblioteci trebuie folosit *#include <omp.h>* și la compilator trebuie adăugată opțiunea *-fopenmp*. Directivele de compilator încep cu *#pragma omp*.

De exemplu însumarea a doi vectori se poate paraleliza simplu prin:

```
#include <omp.h>
```

```
int main() {
    const int n = 100;
    int a[n], b[n], c[n];

    // init

    #pragma omp parallel for
    for(int i = 0; i < n; i++) {
        c[i] = a[i] + b[n];
    }
}
```

De exemplu paralelizarea însumării unui vector se poate realiza prin:

```
#include <omp.h>

int main() {
    const int n = 100;
    int a[n];
    int sum = 0;

    // init

    #pragma omp parallel for reduction(+:sum)
    for(int i = 0; i < n; i++) {
        sum += a[i];
    }
}
```

SIMD

SIMD însemna Single Instruction Multiple Data, este o metodă de paralelizare la nivel de date. Procesoarele care suportă SIMD au regiștrii care pot stoca mai multe date în același timp și mai multe unități aritmetice pentru a procesa datele în paralel.

Pentru procesoarele ARM această extensie se numește NEON. Ca și mod de lucru există 2 moduri: in-line assembly și bibliotecă. Metoda in-line assembly presupune folosirea instrucțiunilor direct în limbaj de asamblare și inserarea acelor fragmente de cod în programe C/C++. Varianta cu biblioteca dedicată aduce tipuri de date speciale și funcții pentru operarea cu acele noi tipuri de date. La final tot același comenzi de assembly ajung în executabil.

Un exemplu de însumare a doi vectori:

```
#include <arm_neon.h>

int main() {
    const int n = 100;
    int a[n], b[n], c[n];

    // init

    // tipuri de date care retin 4 numere int32
    int32x4_t aa, bb, cc;
    // lucram cuc ate 4 elemente deodata,
    // de aici si pasul
    for(int i = 0; i < n; i += 4) {
        // int32x4_t vld1q_s32(int32_t const *ptr)
```

```
        // incarcare de date in registrul vectorial
        // in acel registru se va incarca:
        // ptr[0], ptr[1], ptr[2], ptr[3]
        aa = vld1q_s32(*a[i]);
        bb = vld1q_s32(*b[i]);
        // functie speciala pentru insumare
        cc = vaddq_s32(aa, bb);
        // void vst1q_s32(int32_t *ptr,
        // int32x4_t val)
        // functie de copiere in memorie
         vst1q_s32(*c[i], cc);
    }
}
```

Documentația pentru comenzile din Neon este disponibilă pe site-ul GCC[11].

7.1 Exercițiul 1

Pornind de la exercițiile din laboratorul anterior, adăugați opțiunea de a paraleliza calculul matricial pe mai multe nuclee folosind biblioteca OpenMP. Măsurați timpul de rulare pe 1, 2, 3, respectiv 4 nuclee. Argumentați de ce nu există concurență și conflicte pe date. Implementați înmulțirea într-o funcție și salvați-o pentru mai târziu

Indicații

Rulați pe 10 dimensiuni de matrici, realizați un tabel cu timpii de rulare și realizați un grafic de tip bară.

7.2 Exercițiul 2

Studiați implementarea realizată de compania ARM, pentru înmulțirea vectorizată de matrici [12].

7.3 Exercițiul 3

Realizați o implementare personală de înmulțirii vectorzată de matrici, folosind extensia NEON. Implantați înmulțirea într-o funcție și salvați-o pentru mai târziu.

7.4 Exercițiul 4

Realizați o comparație a timpilor de rulare între înmulțirile de matrici implementate în ultimele două laboratoare.

Indicații

Rulați pe 10 dimensiuni de matrici, realizați un tabel cu timpii de rulare și realizați un grafic de tip linie.

8 Lucrarea 6

Introducere

În acest laborator urmărim:

- realizarea de aplicații folosind protocolul TCP/IP

Socket TCP

În sistemele de operare de tip LINUX cel mai de jos nivel oferit programatorilor este nivelul de transport. Exista două protocoale de transport, pentru protocolul Ethernet, UDP (User Datagram Protocol), orientat pe transferul de date, și protocolul TCP (Transmission Control Protocol), orientat pe transfer, transmițând un răspuns atunci când mesajul ajunge la destinatar.

Pentru server și client există proceduri diferite pentru operare. În cele ce urmează vom detalii aceste etape,

Etapele unui server:

1. **create** - cerere către sistemul de operare pentru creerea unui fișier special.

2. **bind** - cerere către sistemul de operare pentru conectarea fișierului special la o interfața de rețea și un port al interfeței, pentru a putea

accepta conexiuni.

3. **listen** - semnalizează începutul acceptări de conexiuni noi, și creează o coada de conexiuni.

4. **accept** - așteptarea unei conexiuni logice.

5. **read**
   write - comunicație folosind rețeaua.

6. **close** - închiderea fișierului special.

Etapele unui client:

1. **create** - cerere către sistemul de operare pentru creerea unui fișier special.

2. **connect** - realizarea unei conexiuni logice cu serverul.

3. **read**
   write - comunicație folosind rețeaua.

4. **close** - închiderea fișierului special.

Fiecare conexiune TCP este identificată unic prin patru parametri: adresă IP server, port server, adresă IP client și port client.

Bibilioteca *socket*

În sistemele de operare de tip UNIX procedurile pentru TCP sunt disponibile în biblioteca *sys/socket.h*. Pe aceste platforme un socket TCP este reprezentat ca un fișier special, la fiecare operație sistemul de opetarre

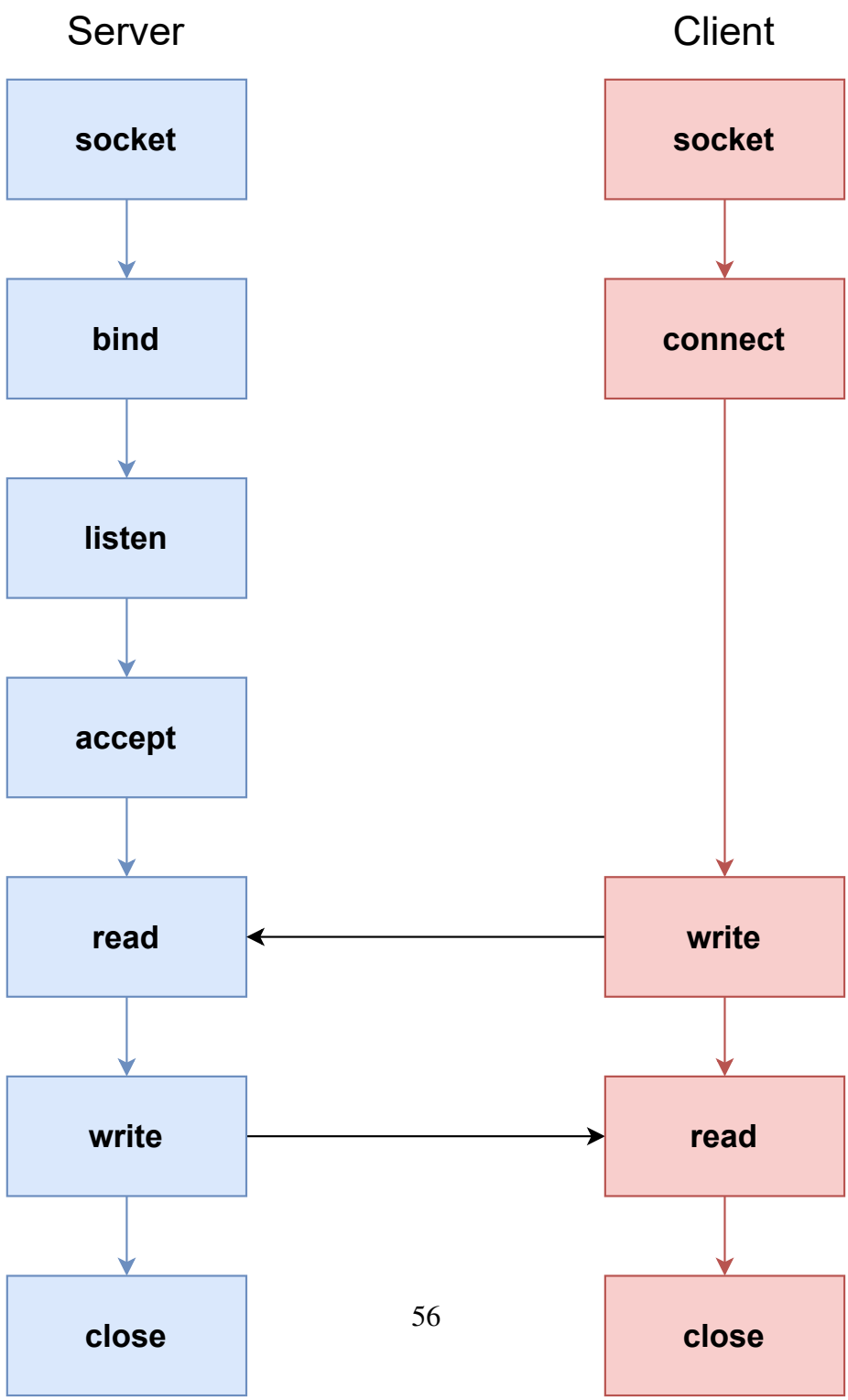

Figura 8.1: Reprezentare grafică etape TCP (bazat pe: [13])

asigură că datele ajung unde trebuie. Fiind la bază niște fișiere, operațiile de scriere și citire sunt specifice fișierelor. La realizarea unei conexiuni noi, la server se va realiza un nou fișier special pentru noua conexiune.

int socket(int domain, int type, int protocol)

Creează un nou fișier special și returnează identificatorul lui, numit descriptor.

Parametrii:

- **domain** - tipul de comunicație pentru care este creat socket-ul. În cazul nostru Ethernet IPv4 (**AF_INET**).
- **type** - protocolul de transfer: UDP (**SOCK_DGRAM**) sau TCP (**SOCK_STREAM**).
- **protocol** - tipul de protocol Ethernet dorit, 0 pentru IP.

int bind(int socket, const struct sockaddr *address, socklen_t address_len)

Conectează un socket la o interfață și un port. Returnează -1 în caz de eroare.

Parametrii:

- **socket** - descriptorul serverului.
- **address** - structura de date pentru configurare.
- **address_len** - dimensiunea în octeți.

int listen(int socket, int backlog)

Marchează pornirea serverului, şi specificarea dimensiuni, cozii de conexiuni. Returnează -1 în caz de eroare.

Parametrii:

- **socket** - descriptorul serverului.
- **backlog** - dimensiunea cozii de conexiuni.

int accept (int socket, struct sockaddr *address, socklen_t *address_len)

Aşteptarea unei noi conexiuni. Returnează decriptorul noi conexiuni şi datele de identificare a acesteia.

Parametrii:

- **socket** - descriptorul serverului.
- **address** - structura de date pentru identificarea conexiuni.
- **address_len** - dimensiunea în octeţi.

int connect(int socket, const struct sockaddr *address, socklen_t address_len)

Realizează o conexiune. Returnează -1 în caz de eroare.

Parametrii:

- **socket** - descriptorul unui socket.
- **address** - structura de date pentru configurare.

- **address_len** - dimensiunea în octeți.

Bibilioteca *unistd*

După ce există socket-ul, trimiterea și citirea de mesaje se realizează prin operații la un fișier. O biblioteca utilă este *unistd*, care pune la dispoziție operații de bază pentru fișiere.

ssize_t write(int fildes, void *buf, size_t nbyte)

Scrierea de date în fișier. Returnează numărul de octeți scriși.

Parametrii:

- **fildes** - descriptorul fișierului.
- **buf** - datele ce trebuiesc scrise, sub formă de vector.
- **nbyte** - numărul de octeți ce trebuiesc scriși.

ssize_t read(int fildes, void *buf, size_t nbyte)

Citirea de date din fișier. Returnează numărul de octeți citiți.

Parametrii:

- **fildes** - descriptorul fișierului.
- **buf** - vector pentru memoria tampon.
- **nbyte** - numărul maxim de octeți ce pot fi citiți.

int close(int fildes)

Închidere socket.

Parametrii:

- **fildes** - descriptorul fișierului.

8.1 Exercițiul 1

Realizați o aplicație de tip client-server în C++, folosind TCP/IP. Serverul primește un mesaj de la client și îi oferă un răspuns, după care se închide. Clientul primește un argument de transmis, ca și argument în linia de comandă, îl trimite la server și așteaptă răspunsul. Pentru verificare afișați mesajele transmise.

Indicații

Pentru bind avem nevoie de o structură de date pentru configurare, de exemplu:

```
#include <netinet/in.h>

struct sockaddr_in address;

address.sin_family = AF_INET;
address.sin_addr.s_addr = INADDR_ANY;
address.sin_port = htons(8080);
```

Iar pentru ase transmite această structură, se face o conversie de tip de date *(struct sockaddr *)*.

Exemplu pentru acceptarea unei noi conexiuni:

```
socklen_t clientSize;
struct sockaddr_in client;
int clientFd = accept(socketFd,
                      (struct sockaddr *)&client,
                      &clientSize);
```

Transmițând mesaje text, nu uitați ca în C șirurile sunt terminate cu NULL, acesta nu se transmite pe rețea din motive de redundantă, adaugați NULL la mesajul primit.

Pentru depanarea codului, se poate folosii Makefile și VS Code, pornind de la indicațiile din [14]. În Linux unealta de depanare este **gdb**.

9 Lucrarea 7

Introducere

În acest laborator urmărim:

- implementarea de protocoale dedicate realizate peste TCP/IP

Atunci când transmitem mesaj dedicate, trebuie să avem o abordare diferită. Linux ne pune la dispoziția să transmitem octeți pe rețea, orice mesaje am vrea să transmitem trebuie să le codificăm.

Abordarea recomandată, mesajul codificat să aibă 3 câmpuri:

- **id** - un număr pe 32 de biți, care codifica în mod unic fiecare tip de mesaj, acestă ne va ajuta la decodificarea datelor.
- **lungime** - un număr pe 32 de biți, care specifică lungimea câmpului de date. Avem nevoie de acest câmp deoarece există posibilitatea ca un mesaj să nu ajungă tot odată, să fie împărțit în mia multe segmente, deci trebuie să știm că trebuie sa mai efectuăm citiri, sau clientul poate transmite mai multe mesaje, trebuie să știm unde se termină mesajul actual.
- **date** - datele mesajului codificate binar.

Astfel, dacă un mesaj nu are date, are cel puțin 8 octeți, id și lungime. Lungimea în acel caz are valoarea 0.

O atenție sporită trebuie acordată tipului de date text, deoarece ele sunt terminate cu NULL. Există două moduri de a se transmite:

1. modul C - stringul terminat cu NULL, lungimea textului mai primește un octet, valoarea NULL, și așa este adăugat la date. Când se extrage, se știe că la întâlnirea lui NULL am terminat un text.

2. modul Java - se reține lungimea textului și caracterele textului. La transmitere mai întâi se transmite lungimea, după caracterele. La recepție se identifică câmpul cu lungimea și se extrag acel număr de caractere. Această procedură trebuie respectată pentru fiecare text.

Recomandare: Se recomandă creerea unui tip de date enum, pentru a reține tipul mesajului.

Recomandare: Pentru a fi scalabilă soluția, este mai ușor realizarea unei interfețe, care are 3 funcții: pentru serializarea mesajului (convertirea obiectului într-un șir de octeți), pentru deserializarea mesajului (pentru popularea obiectului, cu date dintr-un șir de octeți) și returnarea tipului de mesaj.

Exemplu:

messages.h

```
#ifndef MESSAGES
#define MESSAGES

void intToBuf(int, char*);
int bufToInt(char*);
void strToBuf(char*, char*);
```

```
char* bufToStr(char*);

enum MsgId {ERROR, MSG};

// strcura de date pentru stocarea mesajelor
// are un vector de octeti si lungimea lor
struct Message {
    int len;
    char* data;

    Message(int);
    ~Message();
};

class IMessage {
    public:
        virtual Message serialize() = 0;
        virtual void desearilize(const Message&) = 0;
        virtual MsgId getType() = 0;
};

class ErrorMessage: public IMessage {
    public:
        Message serialize();
        void desearilize(const Message&);
        MsgId getType();
};
```

```
class MsgMessage: public IMessage {
    public:
        MsgMessage();
        MsgMessage(const char*);
        Message serialize();
        void desearilize(const Message&);
        char* getMsg();
        void setMsg(const char*);
        MsgId getType();
        ~MsgMessage();
    protected:
        char* msg = nullptr;
};

#endif
```

messages.cpp

```
#include "messages.h"
#include <cstring>

void intToBuf(int n, char* buf) {
    buf[0] = n >> 24;
    buf[1] = n >> 16;
    buf[2] = n >> 8;
    buf[3] = n;
}

int bufToInt(char* buf) {
```

```
    return int((unsigned char)(buf[0]) << 24 |
            (unsigned char)(buf[1]) << 16 |
            (unsigned char)(buf[2]) << 8 |
            (unsigned char)(buf[3]));
}

void strToBuf(char* s, char* buf) {
    strcpy(buf, s);
}

char* bufToStr(char* buf) {
    char* s = new char[strlen(buf) + 1];
    strcpy(s, buf);
    return s;
}

Message ErrorMessage::serialize() {
    Message msg(8);
    intToBuf(MsgId::ERROR, msg.data);
    intToBuf(0, msg.data + 4);

    return msg;
}

void ErrorMessage::desearilize(const Message& msg) { }

MsgId ErrorMessage::getType()
{
```

```
    return MsgId::ERROR;
}

MsgMessage::MsgMessage(){ }

MsgMessage::MsgMessage(const char *p)
{
    MsgMessage();
    setMsg(p);
}

Message MsgMessage::serialize()
{
    Message m(9 + strlen(msg)); // + null terminator
    intToBuf(MsgId::MSG, m.data);
    intToBuf(strlen(msg) + 1, m.data + 4);
    strToBuf(msg, m.data + 8);

    return m;
}

char* MsgMessage::getMsg() {
    return msg;
}

void MsgMessage::setMsg(const char* p) {
    if (msg != nullptr) {
        delete[] msg;
```

```
    }

    msg = new char[strlen(p) + 1];
    strcpy(msg, p);
}

MsgId MsgMessage::getType()
{
    return MsgId::MSG;
}

MsgMessage::~MsgMessage()
{
    if (msg != nullptr)
    {
        delete[] msg;
    }

}

void MsgMessage::desearilize(const Message& m) {
    msg = bufToStr(m.data + 8);
}

Message::Message(int _len) {
    len = _len;
    data = new char[_len];
}
```

```
Message::~Message() {
    delete[] data;
}
```

Pentru a avea o implementare scalabilă, recomandăm implementarea a doua funcții care folosesc polimorfismul în timpul rulări. Aceste doua funcții sunt pentru transmiterea și primirea de mesaje dedicate. Funcție care transmite acest tip de mesaje primește un descriptor pentru conexiune și un pointer către *IMessage*. Pentru decodificare, funcția primește descriptorul conexiuni, pe baza primilor 4 octeți se poate identifica tipul de mesaj și astfel știi de ce clasă avem nevoie.

9.1 Exercițiul 1

Pornind de la implementarea din laboratorul anterior și codul sursă atașat, modificații exercițiul anterior să folosească mesaje dedicate. Pentru simplificare realizați două funcții, pentru transmiterea și recepționarea generică de mesaje, care au următoarele semnături:

```
void customSend(IMessage* message, int fd);
IMessage* CustomRcv(int fd);
```

9.2 Exercițiul 2

Actualizați aplicația anterioară prin adăugarea unui nou timp de mesaj dedicat.

10 Lucrarea 8

Probleme în reprezentarea în virgulă mobilă

Reprezentarea comună a numerelor reale este folosind numerele în virgulă mobilă, reprezentare asemănătoare cu cea științifica. Reprezentarea științifica are trei componente: partea întreagă, partea fracțională și exponentul. În acest format de reprezentare trebuie ca partea întreagă să fie reprezentată de o cifră nenulă, iar exponentul are valoarea necesară ca numărul să fie cel inițial. De exemplu: 2023.11 vă fi scris sub formă științifica ca $2.02311 * 10^3$.

Dificultățile apar atunci când trebuie să efectuăm operații algebrice. Mai întâi trebuie să egalăm exponenții celor două numere, după să efectuăm operația, și în cele din urmă să rescriem rezultatul în format științific. De exemplu $2.202311 * 10^3 + 1.811 * 10^1 = 220.2311 * 10^1 + 1.811 * 10^1 = 222.0421 * 10^1 = 2.220421 * 10^3$. Fiind implicații mai mulți pași neregulați astfel de unități aritmetice sunt dificil de implementat în hardware. O soluție o este algebra în virgulă fixă.

Reprezentarea în virgulă fixă

Rereprezentarea în virgulă fixă costa prin scrierea numerelor cu un număr finit de zecimale, exemple pentru atunci când partea zecimală este formată din 4 cifre: $3.4 = 3.4000$ sau $3 = 3.0000$. Avantajele apar pe partea de calcule algebrice.

Algebra în virgulă fixă

Operațiile algebrice în virgulă fixă sunt mai simple deoarece nu mai trebuie să egalăm exponenți, iar orice depășire a părții zecimale este ignorată.

Adunarea

Operația de adunare este foarte simplă, și se poate realiza în mod direct, fără alte modificări ale numerelor. De exemplu: $3.4000 + 3.0000 = 6.4000$.

Înmulțirea

Și înmulțirea este o operație simplă în algebră cu virgulă fixă. La un singur lucru trebuie avut grijă, și anume depășirea părții fracționare, deoarece, atunci când înmulțim două numere cu virgulă, numărul de cifre de după virgulă este egală cu suma cifrelor fracționale a celor doua numere. Astfel la înmulțire apare o depășire a părții fracționare, dar aceste cifre suplimentare sunt ignorate. De exemplu: $3.4000 * 3.0000 = 10.20000000$, după cum putem observa apare o depășire, dar ignorăm cifrele în plus, și astfel numărul devine 10.2000.

Reprezentarea în virgulă fixă în binar

La fel ca și în baza zece, trebuie să decidem câți biți alegem pentru partea întreagă și câți biți alegem pentru partea fracționară. Astfel un număr în virgulă fixă poate fi stocat în memorie ca un număr întreg și toate operațiile folosite vor fi pentru numere reale.

Un număr în virgulă fixă poate fi stocat ca și un număr întreg, de exemplu: $3.4000 = 34000 * 10^{-4}$, putem să reținem doar 34000 și să ignorăm exponentul. La fel este și cu reprezentarea în binar, doar că baza este 2.

Transformata Fourier Rapidă

Transformata Fourier este folosită pentru analiza spectrală a unui semnal, prin descompunerea semnalului într-o sumă infinită de sinusoide și reprezentarea amplitudinii și fazei acestor componente în funcție de frecvența lor. Pentru a calcula reprezentarea în domeniul spectral pe un calculator se folosește Transformata Fourier Discretă, definit prin ec. 10.1.

$$X(k) = \sum_{n=0}^{N-1} x(n) e^{-j2\pi \frac{kn}{N}}, \quad k = \overline{0, N-1} \tag{10.1}$$

O implementare directă pentru acest calcul presupune un algoritm de complexitate $O(N^2)$ (datorită celor două bucle după n și k), ceea ce prezintă costuri de calcul prea mari pentru a fi fezabil în majoritatea cazurilor. În 1965 însă, doi ingineri, James Cooley și John Tuckey, au descoperit o metodă de calcul mai eficient pentru Transformata Fourier [15]. Algoritmul lor, ulterior denumit FFT (Fast Fourier Transform), se bazează pe faptul că sinusoidele complexe care stau la baza calculului transformatei prezintă unele simetrii care permit ca o parte din coeficienții calculelor să nu fie

calculate direct ci să fie refolosite din calcule anterioare. Pentru a ajunge la coeficienți ce pot fi refolosiți există două posibilități: decimarea în timp și decimarea în frecvență. Decimarea în timp împarte vectorul din domeniul timp în elemente pare și elemente impare și reconstruiește vectorul din domeniul frecvență prin calcularea jumătății inferioare și refolosirea valorilor pentru jumătatea superioară conform ecuației 10.2. Decimarea în frecvență abordează invers, împarte vectorul din domeniul timp în jumătate inferioară și jumătate superioară și calculează elementele pare din vectorul din domeniul frecvență și refolosește valorile pentru elementele impare.

$$\begin{cases} X(k) & = X_{par}(k) + w_N^k \cdot X_{impar}(k) \\ X(k+\frac{N}{2}) & = X_{par}(k) - w_N^k \cdot X_{impar}(k) \end{cases}, \quad k = \overline{0, \frac{N}{2} - 1} \tag{10.2}$$

Coeficientul w_N^k notează rădăcina de pe cercul complex calculat prin:

$$w_N^k = e^{-j2\pi \frac{k}{N}} \tag{10.3}$$

Reprezentând vizual ecuația cu decimare în timp obținem celula de tip fluture (fig. 10.1), care stă la baza algoritmului FFT. Aplicând această celulă recursiv pe toate nivelele de decimare a vectorului în timp putem obținem un algoritm de complexitate $O(N \log_2 N)$.

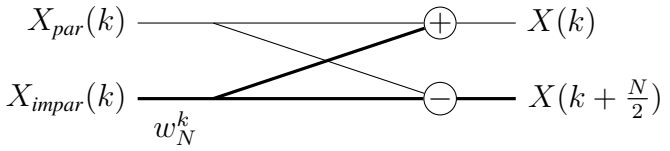

Figura 10.1: Celulă de tip fluture

O detaliere mai cuprinzătoare a acestei operații găsiți și în îndrumarul de prelucrări digitale de semnale [16]. Algoritmul transformatei Fourier

rapide prin decimare în timp este prezentat în următorul pseudocod:

$X \leftarrow x[\text{bitrev}])$ ▷ adresare prin inversarea biților
$N \leftarrow 2$
while $\frac{N}{2} < len_x$ **do**
  for $k \leftarrow 0$ **to** $\frac{N}{2} - 1$ **do**
    $w \leftarrow w_N^k$
    for $n \leftarrow k$ **to** $len_x - 1$ **step** N **do**
      $a \leftarrow X(n)$
      $b \leftarrow w \cdot X(n + \frac{N}{2})$ ▷ înmulțire complexă
      $X(n) \leftarrow a + b$ ▷ adunare complexă
      $X(n + \frac{N}{2}) \leftarrow a - b$
    end for
  end for
  $N \leftarrow 2 \cdot N$
end while

Descrierea mai detaliată a algoritmului puteți găsi în [17] sau [18].

10.1 Exercițiul 1

Pornind de la codul de mai jos, implementați o clasă pentru numerele în virgulă fixă. Verificați corectitudinea operațiilor.

```
template <size_t fraction_bites = 15> class FixedPoint32 {
protected:
    int32_t value;

public:
    FixedPoint32(const int32_t _value = 0): value(_value) {
```

```
    FixedPoint32(const double _value) { }

    FixedPoint32<fraction_bites>& operator=
        (double _value) {}

    FixedPoint32<fraction_bites>& operator=
        (const FixedPoint32<fraction_bites>& b) {}

    operator float() const {}

    FixedPoint32<fraction_bites> operator +
        (const FixedPoint32<fraction_bites>& b) const {}

    FixedPoint32<fraction_bites> operator +=
        (const FixedPoint32<fraction_bites>& b) {}

    FixedPoint32<fraction_bites> operator -
        (const FixedPoint32<fraction_bites>& b) const {}

    FixedPoint32<fraction_bites> operator *
        (const FixedPoint32<fraction_bites>& b) const {}
};

template <size_t fn = 15>
std::ostream& operator <<
    (std::ostream& m_outstream, const FixedPoint32<fn> nr)
    return m_outstream;
}
```

10.2 Exercițiul 2

Măsurații eroarea medie și eroarea medie pătratică, între numerele reprezentate în virgulă mobilă și numerele în virgulă fixă. Ca și aplicație, considerați numere în intervalul $[0, 1)$ și variați partea mobilă între 1 și 30 de biți. Reprezentați pe un grafic tip linie.

Indicații

Pentru o comparație mai bună, folosiți numerele în reale cu precizie dublă, asta implică și modificarea clasei folosite anterior.

10.3 Problema de 10

Implementați algoritmul FFT (Fast Fourier Transform) folosind algebra în virgulă fixă, SIMD și procesarea în paralel. Comparați timpul de rulare cu o implementare în virgulă fixă.

11 Criptografie

În următoarele trei laboratoare vei explora partea de securitate a aplicațiilor.

În acest prim laborator de securitate vei învăța aspectele fundamentale ale criptografiei. Laboratorul începe cu o parte introductivă în care sunt prezentate aspectele teoretice ale criptografiei și ale algoritmului de criptare XOR. Acestea sunt urmate de 4 exerciții care au ca scop familiarizarea cu algoritmul de criptare XOR, atât din perspectiva unui utilizator cât și din perspectiva unei persoane care vrea să recupereze informația criptată.

Așadar, în acest laborator vei învăța:

- Scopurile criptografiei
- Cum să implementezi și cum să folosești algoritmul XOR
- Cum să spargi acest algoritm pentru a găsi informația ascunsă

11.1 Introducere

Pentru a ascunde informația de un posibil actor malițios, aplicațiile folosesc criptografia, Astfel dacă informația este furată, aceasta o sa fie într-un format mai greu de accesat, care va încetini atacatorul. O metodă de a ascunde informația se poate implementa folosind XOR.

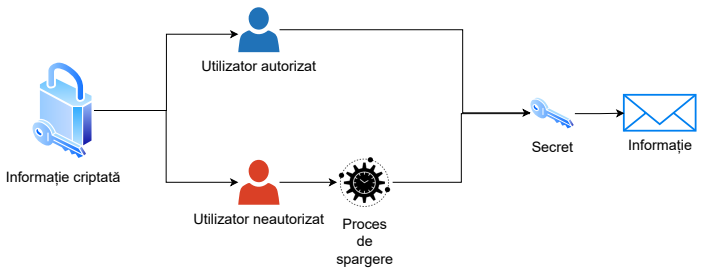

Figura 11.1: Accesul la informație în cazul criptografiei

11.1.1 Criptografie

Criptografia este disciplina care se ocupă de securizarea informației, cu scopul de a preveni accesul neautorizat. Aceasta este puternic legată de alte discipline precum informatică, teoria numerelor și algebră.

Deși scopul este acela de a preveni, în realitate, scopul principal este acela de a îngreuna procesul de obține informația, deoarece cu suficiente resurse și cu diverse metode, informația se poate obține și de către un utilizator neautorizat. În Figura 11.1 se poate observa cum în cazul unui utilizator autorizat, accesul la informație este unul rapid, deoarece acesta are acces la un secret care îl ajută să citească informația. În cazul unui utilizator neautorizat, acesta este nevoit să realizeze un proces de spargere al algoritmului pentru a obține secretul care îi permite să citească informația. Acest proces poate să fie foarte costisitor din punctul de vedere al timpului și al resurselor folosite. Un algoritm de criptografie este considerat bun dacă acest cost tinde către infinit, sau altfel spus dificultatea de a citi informația este una foarte mare.

În cazul în care doriți să învățați mai multe despre criptografie, vă invităm să citiți *Serious Cryptography: A Practical Introduction to Modern Encryption* [19].

A	B	A $\oplus$B
0	1	0
0	1	1
1	0	1
1	1	0

Tabelul 11.1: Tabelul operației XOR

11.1.2 Recapitulare

XOR (Exclusive OR - Sau Exclusiv) este operația logică al cărei rezultat este adevărat doar dacă unul dintre cei doi operanzi are valoarea de adevărat. Simbolul operației XOR este: $\oplus$.

Spre exemplu, valoarea operației 101 $\oplus$111 este 010.

11.1.3 XOR

XOR reprezintă un algoritm simplu de implementat care ascunde informația folosind operația XOR.

Având un mesaj M de mârime n și o cheie K de mârime m, putem lua fiecare caracter al mesajului și să folosim operația XOR pentru a ascunde datele. Pesudocodul pentru un astfel de algoritm se găsește la 0.

input $secret$, key, $secretSize$, $KeySize$
output $xoredSecret$
for $i \leftarrow 0$ **to** $secretSize - 1$ **do**
    $xoredSecret_i \leftarrow secret_i \oplus K \mod KeySize$
end for

11.1.4 Aplicații practice

Chiar dacă acest algoritm este unul simplu, acesta este folosit in cazul sistemelor care doresc să facă ingineria inversă dificilă. Acestea pot să fie fie aplicații legitime care vor sa prevină furtul de proprietate intelectuală, fie aplicații malițioase care vor să prevină detecția și prevenția. Zeus reprezintă un troian care avea ca scop furtul de informații financiare și care se folosea de o variație a algoritmului XOR ca să evite detecția [20].

11.2 Implementarea algoritmului

Implementați o aplicație care primește un string și o cheie ca argumente și returnează rezultatul criptării folosind XOR.

11.2.1 Exemplu

Spre exemplu dacă dorim să criptăm litera A cu cheia B, mai întâi se va salva în memorie, în format binar, aceste două valori. Pentru a se reprezenta caractere o metodă este folosirea codurilor ASCII. Pentru litera A valoarea este 65, în timp ce pentru litera B, codul ASCII este 66. În format binar acestea sunt:

01000001

01000010

În urma aplicării operației XOR, va rezulta următorul rezultat:

00000010

Daca se foloseste cheia **XOR**, pentru string-ul **Sisteme Embedded** va rezulta următoarea secvență de biți:

00001011 00100110 00100001 00101100 00101010 00111111 00111101

01101111 00010111 00110101 00101101 00110111 00111100 00101011 00110111 00111100

11.3 Criptarea unui fișiser

Extindeți aplicația prin implementarea funcționalității criptării unui fișier.

11.3.1 Indicații

Încercați să rezolvați exercițiul folosind o abordare cu flag-uri. Astfel dacă aplicația primește, spre exemplu **-string** aplicația va folosi implementarea de la exercițiul 1, și dacă primește **-file** va folosi funcționalitatea criptării unui fișier.

Fișierul criptat poate să fie și unul binar!

Nu doar fișierele text pot să fie criptate. Fișierele binare pot să fie și ele criptate cu scopul de a preveni furtul de proprietate intelectuală. Dacă vrei să vezi diferența dintre un fișier binar și unul text rulează următoarea comandă:

```
cat /bin/ls
```

Aceasta va printa pe ecran conținutul în format text al binarului **ls**. Fișierele binare reprezintă fișiere a căror informație este codata în format binar. Spre exemplu câteva dintre cele mai cunoscute tipuri de fișiere binare sunt: imagini, executabile și documente.

11.4 Spargerea algoritmului - Metoda naivă

Extindeți aplicația prin implementarea unei metode de decriptare a unui mesaj text fără a ști cheia inițială.

11.4.1 Indicații

Plecați de la premiza că mesajul înainte de criptare a fost text. O metodă de a încerca găsirea cheii este prin încercarea mai multor variante și ignorarea celor care nu oferă un output care să respecte cerința. În cazul nostru, dacă la final output-ul nu reprezintă un string valid, cheia din start nu e bună. Salvați toate output-urile valide într-un fișier, împreună cu cheia.

11.5 Spargerea algoritmului - Sistem de ranking

Acest exercițiu are un grad de dificultate mai ridicat!

Pentru a vedea mai ușor care este cheia adevărată, se poate folosi un sistem de ranking pentru a ordona output-urile. Implementați o metodă de ranking pentru a ordona output-urile și după salvațiile într-un fișier. Explicați de ce ați ales acea metodă.

11.5.1 Indicații

O sugestie pentru o astfel de metodă este verificarea dacă mesajul conține sau nu o anumită secvență. O metoda mai avansată ar fi verificarea limbii mesajului.
O varianta simplistă de ranking este cea in care generam toate soluțiile

pana la o anumita mărime și scoatem soluțiile care conțin caractere care nu se regăsesc în alfabet. Pseudo-codul pentru o astfel de soluție este descrisă în 0.

input $xoredSecret$, $maximumSize$, $alphabet$
output $solutionList$
for $size \leftarrow 1$ **to** $maximumSize$ **do**
    $keyList \leftarrow generateKeys(size)$
    for $keyIndex \leftarrow 0$ **to** $size! - 1$ **do**
        $key \leftarrow keyList_{keyIndex}$
        for $c \in key$ **do**
            if $c \notin alphabet$ **then**
                check next key
            end if
        end for
        $solutionList \leftarrow solutionList \cup key$
    end for
end for

Acestea sunt doar sugestii. Puteți implementa orice metodă.

12 Hashing

În capitolul anterior au fost prezentate elementele de bază ale criptografiei. În acest laborator vă invităm să explorați un domeniu apropiat și anume algoritmii de hashing.

Laboratorul începe cu o introducere teoretică în ceea ce înseamnă un algoritm de hashing și prezintă câteva aspecte teoretice ale algoritmilor din familia Secure Hash Algorithms (SHA). După această secțiune introductivă urmează trei exerciții care presupun utilizarea unui algoritm SHA256 și spargerea unei informații care a fost ascunsă folosind acest algoritm.

Așadar, scopul acestui laborator este să te învețe:

- Ce este un algoritm de hashing?
- Cum se poate folosi acest algoritm?
- Cum se pot afla informațiile care aufost ascunse folosind un algoritm de hashing?
- Cum se poate optimiza acest proces?

12.1 Introducere

Un element esențial al unui algoritm de criptare este posibilitatea de a fi decriptat. Există totuși scenarii în care nu dorim ca o anumită informație

să poată fi decriptată. Un astfel de scenariu este stocarea parolelor. Dacă folosim un algoritm de criptare pentru a stoca parolele, un actor malițios poate pur și simplu să folosească funcția de decriptare pentru a găsi parola.

12.1.1 Hashing

În aceste situații este mai indicat să se folosească algoritmi de hashing. Aceștia reprezintă funcții matematice care nu au o operație inversă. Altfel spus, odată ce o informație fost ascunsă cu un algoritm de hashing, nu există o metodă de a recupera acea informație.

Matematic, o funcție de hash $f : X \to Y$ transformă un element x din mulțimea X într-o valoare hash y din mulțimea Y, exprimat în Ecuația 12.1.

$$f(x) = y, \quad \text{unde } x \in X \text{ și } y \in Y \tag{12.1}$$

În Ecuația 12.2 este exprimat, în mod matematic, faptul că nu există o funcție inversă care să permită recuperarea informației inițiale.

$$\forall x \in X, \nexists\, g(f(x)) = x \tag{12.2}$$

În sistemele informatice, X poate să fie văzută ca mulțimea informațiilor în format ASCII și Y poate să fie văzută ca mulțimea informațiilor în format hex. Desigur aceste mulțimi pot să fie diferite în funcție de algoritmii utilizați.

12.1.2 Secure Hash Algorithms

SHA reprezintă o familie de algoritmi de hashing publicați de către National Institute of Standards and Technology (NIST) [21].

Această familie este împărțită în alte 3 sub-categorii: SHA-1, SHA-2 și SHA-3. Fiecare categorie conține mai mulți algoritmi care utilizează mărimi diferite pentru cuvintele folosite. Spre exemplu, SHA-2-256 utilizează cuvinte de 32 de biți în timp ce SHA-2-512 utilizează cuvinte de 64 de biți. Pe lângă aceste versiuni există și versiuni care folosesc cuvinte trunchiate, precum SHA-2-384.

O comparație între algoritmii SHA poate să fie găsită pe pagina celor de la Code Signing [22].

12.1.3 Aplicații practice

Cel mai întâlnit scenariu de aplicare ale algoritmilor de hashing este stocarea parolelor. În cazul în care baza de date este furată, procesul ca un atacator să descopere parolele este unul mai greu. În acest caz procesul de autentificare presupune compararea parolei introduse de utilizator și hash-ul stocat în baza de date. Dacă cele doua sunt identice, înseamnă ca parola introdusă este corectă.

Amazon Redshit [23] oferă la dispoziție mai mulți algoritmi deja implementați, un exemplu fiind SHA2-512.

Un alt exemplu de utilizare al algoritmilor de hasing este tehnologia blockchain. Spre exemplu Ethereum utilizează algoritmul Keccak-256 [24].

12.2 Utilizarea SHA256

Realizați o aplicație care citește un fișier text, care conține o listă de posibile parole și generează un alt fișier care conține rezultatul funcției SHA256 pentru fiecare parolă citită.

12.2.1 Exemplu

Pentru un fișier care conține următoarele cuvinte:
Sistem
Embedded

Se va genera un alt fișier care să conține:
1c466db7924386fbe638b95eba421a55cff01b29717e156942b68499289998da
a877d89c2afbb8182f7687f1a957e9d3491fe94d93764072538f22a9f45c589a

Indicații

Pentru a găsi hash-ul unui string se pot folosi diferite biblioteci precum haslib [25] pentru Python sau hash-library [26] pentru C++.

12.3 Spargerea unei parole

Funcțiile de hash sunt funții care nu se pot inversa (față de algoritmii de criptografie), așadar pentru spargerea de hash-uri soluția se bazează pe brute-force. O metodă de a face acest lucru este prin utililizarea unei liste de posibile parole, care este convertită într-un rainbow table prin generarea de hash-urilor.

12.3.1 Exemplu

Pentru un fișier care conține următorul wordlist:
Sistem
Embedded
CPU

Si un fisier care conține următoarele hash-uri:
1c466db7924386fbe638b95eba421a55cff01b29717e156942b68499289998da
e745d36c29e6e1843427e3ea90b6844d81807f2f2e403099881e08351d08dd7e
Se va genera un alt fișier care să conțină:

1c466db7924386fbe638b95eba421a55cff01b29717e156942b68499289998da
Sisteme

12.3.2 Indicații

Un astfel de algoritm este prezentat la 0. Acesta primește ca input un Rainbow Table (RT) și o listă de hash-uri (H) de dimensiune n. Daca hash-ul se găsește în rainbow table, atunci se memorează cuvântul inițial într-o structură R. Pseudocodul unei astfel de abordâri se găsește la 0

input $rainbowTable$, $hashList$
output $crackedHashes$
for $i \leftarrow 0$ **to** $n - 1$ **do**
  if $hashList_i \in rainbowTable$ **then**
    $crackedHashes \leftarrow crackedHashes \cup rainbowTable_{hashList_i}$
  end if
end for

12.4 Spargerea parolelor în paralel

Extindeți aplicația implementând o versiune a algoritmului de cracking folosind procesarea în paralel. Realizați un grafic sugestiv în care să comparați cele două implementări.

13 Pen-testing

În laboratoarele anterioare am explorat atât partea de apărare, prin utilizarea unor algoritmi care să protejeze informația, cât și partea de atac, în cadrul exercițiilor în care ne-am propus să spargem o informație criptată sau să găsim o parolă asupra căreia s-a utilizat un algoritm de hashing. În acest laborator o să facem un pas înainte în această direcție și ne vom pune în rolul unui pen-tester care are de testat securitatea unei aplicații web.

Acest laborator începe cu o introducere în penetration testing (pen-testing), prezentând atât partea de tehnică realizare a unui astfel de test cât și partea de raport care trebuie realizat ulterior. După această introducere vă propunem două exerciții care au ca scop familiarizarea într-un cadru practic cu un astfel de test, prin spargerea unei aplicații web găzduite pe Raspberry Pi și realizarea unui raport privind acest test.

Așadar acest laborator își propune să te învețe:

- Conceptele de bază din pen-testing
- Cum să realizezi un pen-test al unei aplicații web
- Cum să scrii un raport pentru un pen-test

13.1 Introducere

O parte importantă pentru a asigura securitatea unei aplicații este realizarea unor teste din perspectiva atacatorului. Astfel se pot găsi și rezolva problemele de securitate înainte ca un actor malițios să le abuzeze.

13.1.1 Pen-testing

Pen-testing se referă la procesul de testare a securității unei anumite aplicații din perspectiva unui atacator. Acesta poate lua trei forme:

1. White-box pen-testing - Persoana care realizează testarea are acces inițial complet la aplicație, codul sursă și la rețea
2. Black-box pen-testing - Persoana care realizează testarea pornește din ipostaza unui atacator extern și este nevoit să își formeze propriul drum către rețea
3. Gray-box pen-testing - Persoana care realizează testarea are acces inițial limitat. Asta poate să însemne să aibă acces la aplicație dar nu la cod, sau să aibâ acces la un punct izolat din rețea

Indiferent de tip, pen-testing-ul urmează în general următorii 5 pași:

1. Reconnaissance - Căutarea de informații despre țintă. Asta poate să însemne identificarea porturilor deschise pe sistem sau identificarea sistemelor din rețea folosind **nmap** [27], identificarea paginilor prezente pe un website folosind **gobuster** [28] sau căutarea de informații pe motoarele de căutare, spre exemplu utilizând **Google Dorking** [29].

2. Vulnerability Identification - Identificarea vulnerabilităţiilor unui anumit sistem. Acest pas poate să fie făcut manual folosind unelte precum **Burp Suite** [30] pentru identificarea vulnerabilităţiilor din aplicaţiile web sau folosind plugin-ul **PEDA**[31] pentru a testa aplicaţiile binare cu **GNU Debugger (GDB)** [32]. Există şi unelte care pot automatiza anumite părţi din acest proces precum **OWSAP Zed Attack Proxy (ZAP)** [33] sau **Nessus**[34].

3. Exploitation - Exploatarea vulnerabilităţii. Acest pas presupune declanşarea unei vulnerabilităţi folosind o încărcătura maliţioasă (payload). Acest lucru se poate realiza fie manual fie cu unelte precum **pwntools**[35] sau **Metasploit**[36]

4. Post Exploitation - Acţiuni care au loc după ce vulnerabilitatea a fost exploatată. Acest pas presupune orice activităţi care au loc după exploatare, precum mişcarea laterală în reţea sau revenirea la un pas anterior pentru acces privilegiat.

5. Raportarea - Pasul final reprezintă scrierea unui raport care să fie utilizat pentru a îmbunătăţii sistemul.

Pentru aplicaţiile web, cele mai întâlnite vulnerabilităţi sunt sumarizate în **OWSAP Top 10** [37]. În cazul în care doriţi să aflaţi mai multe despre acestea, vă învităm să accesaţi acest site pentru a învăţa mai multe despre aceste atacuri.

13.1.2 Raport

La finalul testării se realizează un raport care are ca scop să înştiinţeze atât managementul cât şi personalul tehnic despre vulnerabilităţile găsite.

Acesta începe cu un sumar executiv care are ca public țintă managementul. În acest capitol se prezintă în mare vulnerabilitățile fără a se oferi detalii tehnice.

În aceast capitol trebuie evitat cât de mult posibil jargonul tehnic, pentru a vă asigura că o persoană care nu are cunoștiințe tehnice poate înțelege impactul acelor vulnerabilități. După urmează o parte tehnică în care trebuie descris, cu foarte multe detalii, vulnerabilitățile găsite, impactul lor, cum se pot abuza și , opțional, cum se pot rezolva. Acest lucru trebuie să includă capturi de ecran care să demonstreze acele vulnerabilități. Pe lângă acestea trebuie să fie incluse și ce alte teste s-au realizat, chiar dacă acestea nu s-au finalizat cu găsirea unei vulnerabilități.

Un sfat important în scrierea unui raport este să se evite căutarea unui țap ispășitor. Încercați să menționați inclusiv aspectele pozitive găsite, nu doar cele negative.

Șabloane pentru astfel de rapoarte pot să fie găsite pe site-ul celor de la **Offensive Security (OffSec)** [38] una dintre cele mai importante entități în materie de certificări în zona de pen-testing.

Acest exercițiu are un grad de dificultate mai ridicat. Un motto celebru al celor de la Offensive Security este **Try Harder!** [39]. Chiar dacă pare dificil, încercați din nou și nu renunțați!

13.2 Spargerea unei aplicații web

Accesând placa video pe portul 80 puteți să găsiți aplicație web. Scopul vostru este să încercați să citiți conținutul fișierului **/etc/passwd**.

13.2.1 Indicații

Gândiți-vă mai întâi din perspectiva dezolvatorului. Voi cum ați implementa acest website?

Folosiți-va cunoștințele legate de execuția comenzilor.

13.3 Scrierea unui raport

Realizați un raport în care să prezentați testul desfășurat anterior Pentru a realiza raportul folosiți șablonul celor de la OffSec [38].

13.3.1 Indicații

Sunt necesare următoarele secțiuni:

1. Introducere - Realizați o scurtă introducere referitoare la testul pe care l-ați realizat. Menționați obiectivele principale și cerințele necesare pentru a îi permite unei alte persoane să realizeze acest test (software utilizat, sistem de operare, etc.)

2. Rezumat - Punctele principale ale testului. Aici trebuie să evitați limbajul tehnic.

3. Metodologii - Menționați metodele pe care le-ați încercat, precum activitățile de colectare a informației despre țintă

4. Detalii tehnice - Menționați ceea ce ați realizat. Aici trebuie să specificați detalii tehnice și să includeți capturi de ecran.

13.4 Concluzii

TODO: sumar ce s-a facut, ce s-a urmarit, ce lucrari au fost

Acronime

RDP Remote Desktop Protocol. 9

SSH Secure Shell. 8, 10, 11, 16

Bibliografie

[1] *Raspberry Pi 4 Model B specifications*, URL: https://www.raspberrypi.com/products/raspberry-pi-4-model-b/specifications.

[2] *Raspberry Pi 4 Model B product brief*, URL: https://datasheets.raspberrypi.com/rpi4/raspberry-pi-4-product-brief.pdf.

[3] *OpenSSH for Windows*, URL: https://learn.microsoft.com/en-us/windows-server/administration/openssh/openssh_install_firstuse.

[4] *MobaXterm*, URL: https://mobaxterm.mobatek.net/.

[5] *Microsoft Remote-SSH*, URL: https://marketplace.visualstudio.com/items?itemName=ms-vscode-remote.remote-ssh.

[6] *C limits.h*, URL: https://pubs.opengroup.org/onlinepubs/009695399/basedefs/limits.h.html.

[7] *C stdint.h*, URL: https://pubs.opengroup.org/onlinepubs/009695399/basedefs/stdint.h.html.

[8] *GNU Make Docs*, URL: https://www.gnu.org/software/make/manual/make.html.

[9] *cpusplus.com - sort*, URL: https://cplusplus.com/reference/algorithm/sort/.

[10] *ECE 563: OpenMP*, URL: https://engineering.purdue.edu/~smidkiff/ece563/files/ECE563OpenMPTutorial.pdf.

[11] *ARM NEON Intrinsics*, URL: https://gcc.gnu.org/onlinedocs/gcc-4.4.7/gcc/ARM-NEON-Intrinsics.html.

[12] *Example - matrix multiplication*, URL: https://developer.arm.com/documentation/102467/0200/Example---matrix-multiplication.

[13] *Socket Programming in C/C++*, URL: https://www.geeksforgeeks.org/socket-programming-cc/.

[14] *How to Set up C++ Debugging in VSCode Using a Makefile*, URL: https://hackernoon.com/how-to-set-up-c-debugging-in-vscode-using-a-makefile.

[15] James W. Cooley și John W. Tukey, "An Algorithm for the Machine Computation of Complex Fourier Series, vol. 19," in *Mathematics of Computation* 19.90 (1965), pp. 297–301.

[16] Csaba Zoltán Kertész, *Prelucrarea Digitală a Semnalelor*, Editura Universității Transilvania.

[17] J. Arndt, *Matters Computational: Ideas, Algorithms, Source Code*, Springer Berlin Heidelberg, 2010, ISBN: 9783642147647, URL: https://www.jjj.de/fxt/fxtbook.pdf.

[18] W. H. Press et al., *Numerical Recipes in C - The Art of Scientific Computing*, 2nd ed., Cambridge University Press, 1992.

[19] Jean-Philippe Aumasson, *Serious Cryptography: A Practical Introduction to Modern Encryption*, USA: No Starch Press, 2017, ISBN: 1593278268.

[20] Singh Gurjeet și Porcar Antonio, *Zeus and SpyEye: A Banking Trojan Analysis*, tech. rep., IOActive Labs, Dec. 2011, URL: https://ioactive.com/pdfs/ZeusSpyEyeBankingTrojanAnalysis.pdf.

[21] National Institute of Standards and Technology, *National Institute of Standards and Technology*, https://www.nist.gov/, Accessed: 2024-08-14, 2024.

[22] Codesigningstore, *Hash Algorithm Comparison*, Accessed: 2024-09-10, 2024, URL: https://codesigningstore.com/hash-algorithm-comparison.

[23] Amazon Web Services, Inc., *Amazon Redshift Documentation*, Accessed: 2024-08-12, 2024, URL: https://docs.aws.amazon.com/redshift/.

[24] A.M. Antonopoulos și G. Wood, *Mastering Ethereum: Building Smart Contracts and DApps*, O'Reilly, 2018, ISBN: 9781491971949, URL: https://books.google.ro/books?id=SedSMQAACAAJ.

[25] *Portable C++ Hashing Library*, URL: https://github.com/stbrumme/hash-library.

[26] Stephan Brumme, *Hash Library*, https://github.com/stbrumme/hash-library, Accessed: 2024-09-10, 2024.

[27] Gordon Lyon, *Nmap: Network Mapper*, https://nmap.org/, Accessed: 2024-08-14, 2024.

[28] OJ Reeves, *Gobuster*, https://github.com/OJ/gobuster, Accessed: 2024-08-14, 2024.

[29] Offensive Security, *Google Hacking Database (GHDB)*, https://www.exploit-db.com/google-hacking-database, Accessed: 2024-08-14, 2024.

[30] PortSwigger Ltd., *Burp Suite: Web Vulnerability Scanner*, https://portswigger.net/burp, Accessed: 2024-08-14, 2024.

[31] Long Le, *PEDA: Python Exploit Development Assistance for GDB*, https://github.com/longld/peda, Accessed: 2024-08-14, 2024.

[32] Free Software Foundation, *GDB: The GNU Project Debugger*, https://www.gnu.org/software/gdb/, Accessed: 2024-08-14, 2024.

[33] OWASP, *OWASP ZAP: The Zed Attack Proxy*, https://www.zaproxy.org/, Accessed: 2024-08-14, 2024.

[34] Inc. Tenable, *Nessus: The World's Most Comprehensive Vulnerability Scanner*, https://www.tenable.com/products/nessus, Accessed: 2024-08-14, 2024.

[35] Gallopsled, *pwntools Documentation*, https://docs.pwntools.com/en/stable/, Accessed: 2024-08-14, 2024.

[36] Rapid7, *Metasploit: The World's Most Used Penetration Testing Framework*, https://www.metasploit.com/, Accessed: 2024-08-14, 2024.

[37] OWASP Foundation, *OWASP Top Ten*, https://owasp.org/www-project-top-ten/, Accessed: 2024-08-14, 2024.

[38] Offensive Security, *PEN-200 Reporting Requirements*, https://help.offsec.com/hc/en-us/articles/360046787731-PEN-200-Reporting-Requirements, Accessed: 2024-08-14, 2024.

[39] Offensive Security, *What It Means to Try Harder*, https://www.offsec.com/blog/what-it-means-to-try-harder/, Accessed: 2024-08-14, 2024.