# Limbajul C-ansi: pointeri, tablouri, IO

## Pointeri

### Aritmetica de pointeri, echivalenţa pointeri - tablouri

In C sunt permise o serie de operatii aritmetice cu pointeri, astfel:

Să considerăm **p** un pointer la un anumit tip de dată **T** (declarat **T - - - \*p - - -** ) şi **i** un întreg.

Expresiile **p + i** şi **p - i** (i poate fi pozitiv sau negativ) au ca rezultat tot un pointer cu valoarea mai mare sau mai mică cu **i \* sizeof(T)** decât **p**. De exemplu daca **p** este de tip **int** reprezentat pe **4** octeti, atunci **p + 3** indica o adresa cu **12** (3 locatii a cate 4 octeti) octeti mai mare decat adresa **p**.

O expresie **p1 - p2**, unde **p1** si **p2** sunt pointeri de un anumit tip **T** are ca rezultat un întreg **i** care indică câte locatii cu variabile de tip **T** pot fi plasate intre adresele **p1** si **p2**. De exemplu, daca **p1** si **p2** sunt pointeri de tip **double** ce se reprezinta pe **8** octeti, iar **p2 - p1** are valoarea **3**, atunci adresa **p2** este cu **24** octeti mai mare decat adresa **p1**.

Utilizatorul trebuie sa gestioneze pointerii fata de tipul lor, NU trebuie sa tina cont de lungimea de reprezentare a tipului. Fie **T** un tip de date. Să considerăm declaraţiile şi secvenţa de instrucţiuni:

T t[...], \*p; // Declararea tabloului t si a pointerului p

. . . Initializarea tabloului t . . .

p = t; // p are aceeasi valoare ca si t, inceputul de tablou.

Tabelul de mai jos indică câte patru specificări echivalente de elemente ale tabloului sau adrese ale acestora:

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| Elementele tabloului t: | t[0]  p[0]  \*p  \*t | t[1]  p[1]  \*(p+1)  \*(t+1) | t[2]  p[2]  \*(p+2)  \*(t+2) | t[3]  p[3]  \*(p+3)  \*(t+3) | . . . |
| Adresele elementelor tabloului t: | &t[0]  &p[0]  p  t | &t[1]  &p[1]  p+1  t+1 | &t[2]  &p[2]  p+2  t+2 | &t[3]  &p[3]  p+3  t+3 | . . . |

Această echivalenţă este cunoscută sub numele de **echivalenţa dintre pointeri şi tablouri.** Puteti testa folosind de exemplu programul:

#include <stdio.h>

main () {

long t[10], \*p;

int i;

for (i=0; i<10; t[i++]=i);

p = t;

for (i=0; i<10; i++)

printf("%d %d %d %d\n",t[i],p[i],\*(p+i),\*(t+i));

}

### Citirea unor linii si ordonarea lor alfabetică, varianta 1

Ne propunem să rezolvăm următoarea problemă: Se citeste de la intrarea standard un număr întreg **n**, urmat de citirea citirea a maximum **n** linii. Aceste linii sunt depuse într-un vector cu elemente (pointeri la) stringuri având **n** elemente. După terminarea citirilor, tabloul de linii se ordonează alfabetic, se tipăreşte tabloul ordonat şi se eliberează spaţiile alocate dinamic. Sursa programului este:

// Se citeste un intreg n urmat de citirea a maximum n linii.

// Se construie un vector alocat dinamic de n elemente cu aceste stringuri

// Se ordoneaza alfabetic liniile citite si se tiparesc.

// Se elibereaza toate spatiile alocate dinamic

#include <stdlib.h>

#include <stdio.h>

#include <string.h>

int main() {

char \*\*t, \*p, linie[1000];

int i, j, n, nef;

printf("n = ?"); fgets(linie, 1000, stdin);

n = atoi(linie);

t = (char\*\*)malloc(n \* sizeof(char\*));

for (i = 0; i < n; i++) {

printf("Linie %i = ?", i);

if (fgets(linie, 1000, stdin) == NULL) break;

p = (char \*)malloc(strlen(linie) + 1);

strcpy(p, linie);

t[i] = p;

}

nef = i;

for (i = 0; i < nef - 1; i++)

for (j = i + 1; j < nef; j++)

if (strcmp(t[i], t[j]) > 0) {

p = t[i];

t[i] = t[j];

t[j] = p;

}

printf("\n");

for (i = 0; i < nef; i++) printf("%s",t[i]);

for (i = 0; i < nef; i++) free(t[i]);

free(t); // Atentie la ordinea de eliberare!

return 0;

}

### Citirea unor linii si ordonarea lor alfabetică, varianta 2

Este vorba de (aproape) aceeaşi problemă ca cea de mai sus, cu deosebirea că nu se limitează numărul de linii ce se vor citi. Rezolvarea, de această dată, se face memorând liniile într-o listă simplu înlănţuită. După citirea fiecărei linii, se parcurge lista liniilor deja citite, iar linia curentă se inserează în locul în care liniile dinaintea ei sunt mai mici în ordine alfabetică, iar cele de după mai mari sau egale. In acest fel, la terminarea citirilor liniile sunt deja ordonate alfabetic. Sursa programului este:

// Se citesc de la intrarea standard un sir de linii.

// Dupa citirea fiecarei linii, aceasta se insereaza

// intr-o lista simplu inlantuita, in pozitia care sa

// respecte ordinea alfabetica (sortare prin insertie).

// Dupa terminarea citirilor lista se va tipari.

// Apoi se vor elibera toate spatiile ocupate dinamic.

#include <stdio.h>

#include <string.h>

#include <stdlib.h>

typedef struct node {char \*linie; struct node \*next;} NODE;

NODE \*cap;

void add(char \*linie) {

NODE \*nod, \*pn, \*qn;

nod = (NODE\*)malloc(sizeof(NODE));

nod->linie = (char\*)malloc(strlen(linie) + 1);

strcpy(nod->linie, linie);

for (pn = cap, qn = NULL; pn != NULL; qn = pn, pn = pn->next)

if (strcmp(nod->linie, pn->linie) <= 0) break;

if (qn == NULL) cap = nod; else qn->next = nod;

nod->next = pn;

}

int main () {

char linie[1000];

NODE \*pn;

cap = NULL;

for ( ; ; ) {

printf("Linie = ?");

if (fgets(linie, 1000, stdin) == NULL) break;

add (linie);

}

printf("\n");

for (pn = cap; pn != NULL; pn = pn->next) printf("%s",pn->linie);

for (pn = cap; pn != NULL; pn = pn->next) {

free(pn->linie);

free(pn);

}

return 0;

}

### Citirea unor linii si ordonarea lor alfabetică, varianta Go

Solutia Go nu cere limitari ale lungimilor liniilor, nici limitari ale numarului de linii.

// Se citesc de la stdin linii, care se depun in slice-ul linii.

// Dupa terminarea citirilor, se sorteaza linii si se scrie pe stdout.

package main

import (

"bufio"

"fmt"

"io"

"os"

"sort"

"strings"

)

func main() {

linii := []string{}

in := bufio.NewReader(os.Stdin)

for {

linie, err := in.ReadString('\n')

if err == io.EOF { break }

linie = strings.Replace(linie, "\n", "", -1) // Efectiv Unix

linie = strings.Replace(linie, "\r\n", "", -1) // Efectiv Windows

linii = append(linii, linie)

}

sort.Strings(linii)

for i := 0; i < len(linii); i++ { fmt.Println(linii[i]) }

}

### Citirea unor linii si ordonarea lor alfabetică, varianta Python

Solutia Python nu cere limitari ale lungimilor liniilor, nici limitari ale numarului de linii.

# Se citesc de la stdin linii, care se depun in lista linii.

# Dupa terminarea citirilor, se sorteaza linii si se scrie pe stdout.

import fileinput

def main():

linii = []

for linie in fileinput.input():

linie = linie.replace("\n", "") # Efectiv Unix

linie = linie.replace("\r\n", "") # Efectiv Windows

linii.append(linie)

linii.sort()

print (linii)

for linie in linii: print(linie)

main()

## Tablouri C alocare statică şi dinamică

### Tablouri bidimensionale statice

Tablourile cu elemente de tip **T** bidimensionale în c sunt declarate sub forma **T t[m][n] - - -**, unde **T** este tipul elementelor de tablou, **t** este numele variabilei tablou, iar **m** şi **n** sunt constante întregi. Constanta **m** indică numărul de linii, iar **n** este numărul de coloane. In memorie, începând cu adresa **t** sunt rezervate **m\*n** locaţii consecutive de tip **T**, în care elementele sunt reprezentate linie după linie.

Principalele dificultăţi ale acestui mod de reprezentare sunt:

* Obligaţia de a preciza dimensiunile **m** şi **n** cu valori maximale, deşi dimensiunile reale pot rezulta din calcule şi sunt mai mici decât dimensiunile **m** şi **n**.
* Transmiterea unui astfel de tablou ca şi parametru este o sursă puternică de erori dacă se dau dimensiunile reale în locul celor maximale alocate.

Exemplul următor defineşte o funcţie **list** care primeşte un tablou de întregi şi listează matricea. In program se alocă un tablou cu **3** linii şi **5** coloane. Programul apelează această funcţii pentru mai multe dimensiuni reale. In comentarii se văd efectele acestor apelări. Programul este:

#include <stdio.h>

void list(int m, int n, int a[m][n]) {

// Ok daca m <= cu cel alocat (3) si n == cu cel alocat (5)

// In caz contrar, sunt mari sanse sa interpreteze ca int o locatie aiurea.

printf("\n");

for (int i=0; i<m; i++) {

for (int j=0; j<n; j++) printf(" %02d", a[i][j]);

printf("\n");

}

}

int main() {

int a[3][5] = {{00,01,02,03,04},{10,11,12,13,14},{20,21,22,23,24}};

list(3, 5, a); // 00 01 02 03 04/10 11 12 13 14/20 21 22 23 24

list(2, 5, a); // 00 01 02 03 04/10 11 12 13 14

list(1, 5, a); // 00 01 02 03 04

list(3, 4, a); // 00 01 02 03/04 10 11 12/13 14 20 21

list(3, 3, a); // 00 01 02/03 04 10/11 12 13

list(3, 2, a); // 00 01/02 03/04 10

list(3, 1, a); // 00/01/02

list(2, 3, a); // 00 01 02/03 04 10

list(1, 4, a); // 00 01 02 03

list(4, 5, a); // 00 01 02 03 04/10 11 12 13 14/20 21 22 23 24/ ? ? ? ? ?

list(3, 6, a); // 00 01 02 03 04 10/11 12 13 14 20 21/22 23 24 ? ? ?

list(4, 4, a); // 00 01 02 03/04 10 11 12/13 14 20 21/22 23 24 ?

list(15, 1, a); // 00/01/02/03/04/10/11/12/13/14/20/21/22/23/24

list(1, 15, a); // 00 01 02 03 04 10 11 12 13 14 20 21 22 23 24

list(1, 20, a); // 00 01 02 03 04 10 11 12 13 14 20 21 22 23 24 ? ? ? ? ?

return 0;

}

### Tablouri bidimensionale dinamice

In baza echivalenţei dintre tablouri şi pointeri, se pot aloca tablouri dinamice! Sursa următoare alocă dinamic un tablou de 3X5 ca în exemplul precedent. Programul este:

#include <stdio.h>

#include <stdlib.h>

void list(int m, int n, int \*\*a) {

// Ok daca n <= cu cel alocat (5) si m <= cu cel alocat (3)

// In caz contrar, sunt mari sanse de Segmentation fault (core dumped),

// deoarece se fac adresari prin pointeri inexistenti.

// Uneori este posibil sa interpreteze ca int o locatie aiurea.

printf("\n");

for (int i=0; i<m; i++) {

for (int j=0; j<n; j++) printf(" %02d", a[i][j]);

printf("\n");

}

}

int main() {

int \*\*a;

a = (int\*\*)malloc(3\*sizeof(int\*));

printf("%d %d %d\n",a, sizeof(int\*\*), sizeof(int));

for (int i = 0; i < 3; i++) {

a[i] = (int\*)malloc(5\*sizeof(int));

printf("%d\n",a[i]);

for (int j = 0; j < 5; j++) a[i][j] = 10\*i + j;

}

//int a[3][5] = {{00,01,02,03,04},{10,11,12,13,14},{20,21,22,23,24}};

list(3, 5, a); // 00 01 02 03 04/10 11 12 13 14/20 21 22 23 24

list(2, 5, a); // 00 01 02 03 04/10 11 12 13 14

list(1, 5, a); // 00 01 02 03 04

list(3, 4, a); // 00 01 02 03/04 10 11 12/13 14 20 21

list(3, 3, a); // 00 01 02/10 11 12/20 21 22

list(3, 2, a); // 00 01/10 11/20 21

list(3, 1, a); // 00/10/20

list(2, 3, a); // 00 01 02/10 11 12

list(1, 4, a); // 00 01 02 03

list(4, 5, a); // Segmentation fault sau numere aiurea din zona nealocata

list(3, 6, a); // Segmentation fault sau numere aiurea din zona nealocata

list(4, 4, a); // Segmentation fault sau numere aiurea din zona nealocata

list(15, 1, a); // Segmentation fault sau numere aiurea din zona nealocata

list(1, 15, a); // Segmentation fault sau numere aiurea din zona nealocata

list(1, 20, a); // Segmentation fault sau numere aiurea din zona nealocata

return 0;

}

In primele 8 linii din funcţia main este ilustrat modul în care se poate aloca dinamic o matrice de 3 linii şi 5 coloane. Evident, în funcţiile **malloc**, în locul constantelor 3 şi 5 pot apărea orice variabile sau expresii întregi! Următoarele linii din main apelează funcţia **list** la fel ca la exemplul precedent. Comentariile arată efectele apelurilor. Se vede că în acest caz se pot folosi corect şi submatrice ale celei alocate, preluând mai puţine prime linii şi / sau prime coloane. Figurile următoare prezintă alocarea statică, respectiv dinamică pentru acelaşi tablou.

|  |  |
| --- | --- |
|  |  |

### Generalizări: tablouri dinamice nerectangulare şi / sau multidimensionale

Alocarea dinamică a tablourilor bidimensionale poate fi generalizată uşor. Iată câteva posibile generalizări:

* Tablouri bidimensionale nerectangulare (nu toate liniile au acelaşi număr de coloane).
* Tablouri tridimensionale, quatrodimensionale, multidimensionale etc.
* Tablouri multidimensionale nerectangulare.
* etc.

Pentru tablourile bidimensionale nerectangulare de dimensiune **mXn** există două rezolvări simple:

1. Se alocă în plus un tablou de dimensiune **m** care să reţină lungimea fiecărei linii.
2. **Folosirea ca terminatori a pointerilor NULL**. Această rezolvare este foarte generală şi se pretează perfect la limbajele C / C++: La pointerul **\*\*T (\*\*int** în cazul nostru) care reţine adresele începuturilor de linii i se alocă **m+1** locaţii de tip \*T în loc de **m** spaţii. Pe ultima poziţie se depune un pointer **NULL** şi astfel numărul de linii este reţinut automat - până la întâlnirea pointerului **NULL**. Pentru fiecare linie se alocă o locaţie în plus (**n+1** locuri sau lungimea efectivă a liniei plus 1) şi în ultima locaţie se pune **NULL** sau o valoare care nu poate să apară în cadrul matricei **MAXINT**, **MAXLONG**, **MAXDOUBLE** etc. De multe ori valorile elementelor de tablou pot fi structuri, stringuri etc. In acest caz ca element al matricei apare un pointer spre structura / stringul respectiv, iar ca terminator al liniei se poate folosi pointerul **NULL**.

Pentru tablourile tridimensionale indicatorul de tablou este de tip **\*\*\*T**. Acesta punctează la începuturile straturilor care sunt structuri de tip **\*\*T** (tablouri bidimensionale) şi fiecare urmează schema de reprezentare a tablourilor bidimensionale.

Pentru tablourile quatrodimensionale se pleacă de la tipul **\*\*\*\*T** şi se urmează schema de mai sus. De aici generalizarea este clară. Folosirea terminatorilor **NULL** la aceste scheme de alocare uşurează mult prelucrarea.

## Fişiere text şi fişiere binare

In limbajul CS se folosesc sintagmele de fişier text şi fişier binar. Fără pretenţia de a da definiţii exacte, încercăm să lămurim ce se înţelege, îndeobşte, prin aceste două tipuri de fişiere:

* Fişiere *text*, sunt cele al căror conţinut poate fi afişat pe un ecran sau poate fi tipărit pe o imprimantă. El este format dintr‑o succesiune de octeţi, fiecare conţinând codul unui caracter tipăribil: literă mare, literă mică, cifră, simbol special. Codificarea caracterelor se face folosind unul dintre sistemele de codificare standard: ASCII (pe 7 biţi), UNICODE (pe 8, 16, 32 biţi). La acest tip de fişiere articolul este format dintr-o *linie.* Două linii sunt separate fie prin ***'\n'*** – cazul Unix, ***\r*** în cazul MacOS, ***'\r\n'*** cazul Windows. Un fişier text se termină întotdeauna cu caracterul funcţional EOF (End Of File). Fiecare limbaj de programare are funcţii specifice de lucru cu fişiere text: **readln**, **writeln** etc.
* Fişiere *binare*, formate din şiruri de octeţi consecutivi fără nici o semnificaţie pentru afişare. Semnificaţia fiecărui octet este numai internă. Spre exemplu, fisiere avi, mp3, fişierele obiect rezultate in urma compilări şi fişierele executabile rezultate din editări de legături sunt fişiere binare. Evident că înşiruirea de biţi şi octeţi este "înţeleasă" de către CPU. Incercarea de a tipări direct un astfel de fişier nu are nici un sens!

In fapt, orice fişier "text" poate fi tratat ca şi un fişier binar dacă este accesat cu funcţii specifice fişierelor binare (care nu sunt din categoria celor text!).

Identificarea fişierelor text codate în altceva decât ASCII / 7 biţi este cunoscută sub numele de codificare UNICODE. De fapt acesta este un standard, iar codificările lui sunt UTF-8, UTF-16, UTF-32. Marcarea codificărilor UNICODE se face punând în primii octeţi ai fişierului o configuraţie de valori care să identifice tipul de codare. Această configuraţie (şir de octeţi) poartă numele de **BOM (Byte Order Mark)**. După caz, ea poate fi:

* EF BB BF, pentru UTF-8
* FE FF (maşini big-endian) respectiv FF FE (maşini little-endian), pentru UTF-16
* 00 00 FE FF (maşini big-endian) respectiv FF FE 00 00 (maşini little-endian) pentru UTF-32

## Operaţii IO în C

### Principalele funcţii

Exista doua posibilitati de efectuare a operatiilor I/O asupra unui fişier din programe C:

* Prin functiile standard C (**fopen, fclose, fgets, fprintf, fread, fwrite, fseek, sprints, sscanf** etc.) existente in bibliotecile standard C; prototipurile acestora se afla in fisierul header **<stdio.h>** (nivelul superior de prelucrare al fisierelor). Pentru orice detalii legate de aceste functii, ca si pentru alte functii inrudite cu acestea, se pot consulta manualele Unix **$ man numefunctie** sau **$ man 3 numefunctie**
* Prin functii standardizate POSIX (**open, close, read, write, lseek, dup, dup2, fcntl** etc.) care reprezinta puncte de intrare in nucleul Unix si ale caror prototipuri se afla de regula in fisierul header **<unistd.h>**, dar uneori se pot afla si in **<sys/types.h>**, **<sys/stat.h>** sau **<fcntl.h>** (nivelul inferior de prelucrare al fisierelor). Pentru orice detalii legate de aceste functii, ca si pentru alte functii inrudite cu acestea, se pot consulta manualele Unix: **$ man numefunctie** sau **$ man 2 numefunctie**

Prima categorie de functii o presupunem cunoscuta deoarece face parte din standardul C (ANSI). Functiile din aceasta categorie repereaza orice fisier printr-o structura **FILE \***, pe care o vom numi descriptor de fisier.

Functiile din a doua categorie constituie **apeluri sistem Unix pentru lucrul cu fisiere**. Ele (antetul lor) sunt cuprinse in standardul POSIX. Functiile din aceasta categorie repereaza orice fisier printr-un intreg nenegativ, numit **handle**, dar atunci cand confuzia nu este posibila il vom numi tot descriptor de fisier. Pentru a obtine detalii despre formatele de fisiere si despre functii sau comenzi specifice formatelor de fisiere se poate consulta **$ man 5 nume**

### Interclasarea a n fişiere text ordonate alfabetic

// Interclaseaza fisierele text, cu liniile ordonate alfabetic, ale caror nume

// sunt date la linia de comanda

#include <stdio.h>

#include <string.h>

#define MAXFILE 100

#define MAXNRFILES 20

#define MAXLINIE 1000

int main(int c, char \*\*argv) {

FILE \*fi[MAXNRFILES];

char lMax[MAXLINIE], lMin[MAXLINIE], liniaCurenta[MAXNRFILES][MAXLINIE];

int n, i;

lMax[0] = 0x7f; // Cea mai mare linie (nu e in fisiere)

lMax[1] = 0;

lMin[0] = 0;// Cea mai mica linie

for (i = 1, n = 0; argv[i]; i++) {

fputs(argv[i], stdout);

fi[n] = fopen(argv[i], "r");

if (fi[n] == NULL) continue; // Probabil nume eronat

liniaCurenta[n++][0] = 0;// La deschidere se pune linia vida

} // Terminat de deschis fisierele de intrare

for (;; ) { // Ciclul principal de interclasare

for (i = 0; i < n; i++) { // Citiri de linii din unele fisiere

if (strcmp(lMin, liniaCurenta[i]) != 0) continue;// Nu citeste

if (fgets(liniaCurenta[i], MAXLINIE, fi[i]) != NULL) continue;

strcpy(liniaCurenta[i], lMax);

fclose(fi[i]);// S-a terminat fisierul

}

strcpy(lMin, lMax); // Alege cea mai mica linie dintre curente

for (i = 0; i < n; i++)

if (strcmp(lMin, liniaCurenta[i]) > 0)

strcpy(lMin, liniaCurenta[i]);

if (strcmp(lMin, lMax) == 0) break;// Terminat interclasarile

// Scrierea in iesire:

// (1) iesire fara linii multiple, se scrie doar lMin

// (2) iesire cu linii multiple, se scriu cele egale cu lMin

for (i = 0; i < n; i++) { // (2)

if (strcmp(lMin, liniaCurenta[i]) != 0) continue;// Nu scrie

fputs(lMin, stdout);

}

} // Terminat interclasarile

return 0;

} // main

### Interclasarea a n fişiere text ordonate alfabetic - Go

// Interclaseaza fisierele text, cu liniile ordonate alfabetic, ale caror nume

// sunt date la linia de comanda

package main

import (

"bufio"

"fmt"

"io"

"os"

"strings"

)

func main() {

f := []\*os.File{}

fi := []\*bufio.Reader{}

liniaCurenta := []string{}

lMax, lMin := "~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~", ""

var n, i int

for i = 1; i < len(os.Args); i++ {

file, err := os.Open(os.Args[i])

if err != nil { continue }

fmt.Print(os.Args[i], " ")

f = append(f, file)

fi = append(fi, bufio.NewReader(file))

liniaCurenta = append(liniaCurenta, lMin)

}

n = len(fi)

fmt.Println(lMin)

fmt.Println(n)

fmt.Println(lMax)

for {

for i = 0; i < n; i++ {

if lMin != liniaCurenta[i] { continue }

linie, err := fi[i].ReadString('\n')

if err == io.EOF {

liniaCurenta[i] = lMax

f[i].Close()

continue

}

linie = strings.Replace(linie, "\n", "", -1) // Unix

liniaCurenta[i] = strings.Replace(linie, "\r\n", "", -1) // Windows

}

lMin = lMax

for i = 0; i < n; i++ {

if lMin > liniaCurenta[i] { lMin = liniaCurenta[i] }

}

if lMin == lMax { break }

// Scrierea in iesire:

// (1) iesire fara linii multiple, se scrie doar lMin

// (2) iesire cu linii multiple, se scriu cele egale cu lMin

for i = 0; i < n; i++ { // (2)

if lMin != liniaCurenta[i] { continue }

fmt.Println(lMin)

}

}

}

### Interclasarea a n fişiere text ordonate alfabetic - Python

# Interclaseaza fisierele text, cu liniile ordonate alfabetic, ale caror nume

# sunt date la linia de comanda

import os

import sys

def main():

f = []

liniaCurenta = []

lMax, lMin = "~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~", ""

n, i = 0, 0

for arg in sys.argv[1:]:

file = open(arg)

if file == None: continue

print(arg+" ", end='')

f.append(file)

liniaCurenta.append(lMin)

n = len(f)

print(lMin)

print(str(n))

print(lMax)

while True:

for i in range(n):

if lMin != liniaCurenta[i]: continue

linie = f[i].readline()

if linie == "":

liniaCurenta[i] = lMax

f[i].close()

continue

linie = linie.replace("\n", "") # Unix

liniaCurenta[i] = linie.replace("\r\n", "") # Windows

lMin = lMax

for i in range(n):

if lMin > liniaCurenta[i]: lMin = liniaCurenta[i]

if lMin == lMax: break

# Scrierea in iesire:

# (1) iesire fara linii multiple, se scrie doar lMin

# (2) iesire cu linii multiple, se scriu cele egale cu lMin

for i in range(n): # (2)

if lMin != liniaCurenta[i]: continue

print(lMin)

main()

### Oglindirea conţinutului unui fişier: soluţia 1

La linia de comanda se da un nume de fisier. Se cere sa se realizeze oglindirea acestui fisier - primul octet al fisierului se schimba cu ultimul, al doilea cu penultimul s.a.m.d pana se ajunge la jumatatea fisierului.

Prezentam doua variante de rezolvare si invitam studentii sa le testeze pe ambele si sa observe diferentele intre codurile C si intre vitezele de executie. In prima soluţie se transferă octet cu octet, iar în soluţia a doua (secţiunea următoare) se face transfer pe blocuri (în cazul nostru de câte 10000 octeţi). Codul (completat cu calculul duratei operatiei) este:

// Oglindeste continutul unui fisier binar dat la linia de comanda.

// Oglindirea se realizeaza citind caracter cu caracter.

// A se confrunta cu executia programului similar oglindan.c

#include <stdio.h>

#include <sys/types.h>

#include <sys/stat.h>

#include <fcntl.h>

#include <unistd.h>

#include <time.h>

int main(int argc, char \*argv[]) {

int f;

long s, d;

char bs, bd;

struct stat stare;

time\_t start;

start = time(NULL);

stat(argv[1], &stare);

f = open(argv[1], O\_RDWR);

for (s = 0, d = stare.st\_size-1; s < d; s++, d--) {

lseek(f, s, SEEK\_SET);

read(f, &bs, 1);

lseek(f, d, SEEK\_SET);

read(f, &bd, 1);

lseek(f, s, SEEK\_SET);

write(f, &bd, 1);

lseek(f, d, SEEK\_SET);

write(f, &bs, 1);

}

close(f);

printf("Durata: %d\n",(int)(time(NULL)-start));

return 0;

}

### Oglindirea conţinutului unui fişier: soluţia 1 Go

package main

import "os"

func main() {

var s, d int64

t, \_ := os.Stat(os.Args[1])

d = t.Size() - 1

bs := make([]byte, 1)

bd := make([]byte, 1)

f, \_ := os.OpenFile(os.Args[1], os.O\_RDWR, 0666)

for s=0; s < d; {

\_,\_ = f.Seek(s, 0)

\_,\_ = f.Read(bs)

\_,\_ = f.Seek(d, 0)

\_,\_ = f.Read(bd)

\_,\_ = f.Seek(s, 0)

\_,\_ = f.Write(bd)

\_,\_ = f.Seek(d, 0)

\_,\_ = f.Write(bs)

s++

d--

}

f.Close()

}

### Oglindirea conţinutului unui fişier: soluţia 1 Python

import sys

import os

s = 0

d = os.stat(sys.argv[1]).st\_size - 1

f = open(sys.argv[1], "rb+")

while s < d:

f.seek(s, 0)

bs = f.read(1)

f.seek(d, 0)

bd = f.read(1)

f.seek(s, 0)

f.write(bd)

f.seek(d, 0)

f.write(bs)

s = s + 1

d = d - 1

f.close()

### Oglindirea conţinutului unui fişier: soluţia buffer

Problema de mai sus are o soluţie mult mai eficientă folosind citirea pe blocuri:. Este util sa se retina din solutia a doua functiile construite de noi **Read** si **Write**. Ele citesc / scriu, eventual prin repetarea read / write, exact **n** octeţi. Utilizatorul trebuie să se asigure în prealabil că schimbul celor **n** octeţi este posibil. Aceste funcţii s-ar putea dovedi utile in multe situatii.

// Oglindeste continutul unui fisier binar dat la linia de comanda.

// Oglindirea se realizeaza citind blocuri de octeti consecutivi.

// A se confrunta cu executia programului similar oglinda1.c

#include <stdio.h>

#include <sys/types.h>

#include <sys/stat.h>

#include <fcntl.h>

#include <unistd.h>

#include <time.h>

#define MAX 10000

void oglinda(char sir[MAX], int n) {

int s, d;

char t;

for (s = 0, d = n - 1; s < d; s++, d--) {

t = sir[s];

sir[s] = sir[d];

sir[d] = t;

}

}

void Read(int f, char \*t, int n) {

char \*p;

int i, c;

for (p = t, c = n; ; ) {

i = read(f, p, c);

if (i == c) return;

c -= i;

p += i;

}

}

void Write(int f, char \*t, int n) {

char \*p;

int i, c;

for (p = t, c = n; ; ) {

i = write(f, p, c);

if (i == c) return;

c -= i;

p += i;

}

}

int main(int argc, char \*argv[]) {

int f, n;

long s, d, l, m;

char bs[MAX], bd[MAX];

struct stat stare;

time\_t start;

start = time(NULL);

stat(argv[1], &stare);

l = stare.st\_size;

m = l / 2;

n = MAX;

if (n > m) n = m;

f = open(argv[1], O\_RDWR);

for (s = 0, d = l - n; s + n <= m; s += n, d -= n) {

if (m - s < n) n = m - s;

lseek(f, s, SEEK\_SET);

Read(f, bs, n);

lseek(f, d, SEEK\_SET);

Read(f, bd, n);

oglinda(bs, n);

oglinda(bd, n);

lseek(f, s, SEEK\_SET);

Write(f, bd, n);

lseek(f, d, SEEK\_SET);

Write(f, bs, n);

}

close(f);

printf("Durata: %d\n",(int)(time(NULL)-start));

return 0;

}

### Oglindirea conţinutului unui fişier: soluţia buffer Go

package main

import "os"

func oglinda (sir [] byte) []byte {

s, d := 0, len(sir) - 1

for ; s < d; {

t := sir[s]

sir[s] = sir[d]

sir[d] = t

s++

d--

}

return sir

}

func Read(f \*os.File, n int) []byte {

sir := make([]byte, 0)

c := n

for {

t := make([]byte, c)

c,\_ = f.Read(t)

sir = append(sir, t...)

if len(sir) == n { return sir }

c = n - len(sir)

}

}

func Write(f \*os.File, sir []byte) {

t := sir

for {

c,\_ := f.Write(t)

if len(t) == c { return }

t = t[c:]

}

}

func main() {

const MAX = 10000

var s, d, m, l int64

t,\_ := os.Stat(os.Args[1])

l = t.Size()

m = l / 2

n := MAX

if int64(n) > m { n = int(m) }

s = 0

d = l - int64(n)

f,\_ := os.OpenFile(os.Args[1], os.O\_RDWR, 0666)

for s=0; s + int64(n) <= m; {

if m - s < int64(n) { n = int(m - s) }

f.Seek(s, 0)

bs := Read(f, n)

f.Seek(d, 0)

bd := Read(f, n)

f.Seek(s, 0)

Write(f, oglinda(bd))

f.Seek(d, 0)

Write(f, oglinda(bs))

s += int64(n)

d -= int64(n)

}

f.Close()

}

### Oglindirea conţinutului unui fişier: soluţia buffer Python

import sys

import os

def oglinda (sir):

s, d = 0, len(sir) - 1

res = bytearray(sir)

while s < d:

t = res[s]

res[s] = res[d]

res[d] = t

s = s + 1

d = d - 1

return bytes(res)

def Read(f, n):

sir = bytes()

c = n

while True:

t = f.read(c)

sir = sir + t

if len(sir) == n: return sir

c = n - len(sir)

def Write(f, t):

f.write(t)

def main():

MAX = 5000

l = os.stat(sys.argv[1]).st\_size

m = l / 2

n = MAX

if n > m: n = int(m)

s = 0

d = l - n

f = open(sys.argv[1], "rb+")

while s + n <= m:

if m - s < n: n = m - s

f.seek(s, 0)

bs = Read(f, n)

f.seek(d, 0)

bd = Read(f, n)

f.seek(s, 0)

f.write(oglinda(bd))

f.seek(d, 0)

f.write(oglinda(bs))

s = s + n

d = d - n

f.close()

main()

## Manipularea fişierelor în sisteme de fişiere

### Principalele prototipuri de funcţii

Iata prototipurile celor mai importante dintre aceste apeluri sistem:

* int chdir (const char \*nume);
* char \*getcwd(char \*mem, int dimensiune);
* int mkdir (const char \*nume, unsigned int drepturi);
* int rmdir (const char \*nume);
* int unlink(const char \*nume);
* int link(const char \*numevechi, const char \*numenou);
* int symlink(const char \*numevechi, const char \*numenou);
* int chmod (const char \*nume, unsigned int drepturi);
* int stat (const char \*nume, struct stat \*stare);
* int mknod(const char \*nume, unsigned int mod, dev\_t dev);
* int chown(const char \*nume, unsigned int proprietar, unsigned int grup);
* int access(const char \*nume, int permisiuni);
* int rename(const char \*numevechi, const char \*numenou);

### Parcurgerea recursivă a fişierelor dintr-un director şi descendenţi

// La linia de comanda se da un nume de director urmate de tipuri de fisiere

// Se face rezumatul fisierelor de tipurile specificate din directorul dat.

#include <stdio.h>

#include <string.h>

#include <limits.h>

#include <stdlib.h>

#include <ctype.h>

#include <sys/types.h>

#include <sys/stat.h>

#include <dirent.h>

#include <time.h>

#define MAX 1024

#define TRUE 1

#define FALSE 0

char \*types[MAX];

time\_t start, lastTime = 0; // 31536000 secunde / an

int nr = 0;

int endWith(char \*nume, char \*tip) {

int ln, lt;

ln = strlen(nume);

lt = strlen(tip);

if (strcmp(nume + ln - lt, tip) == 0) return TRUE;

return FALSE;

}

int isType(char \*nume) {

int i;

if (types[0] == 0) return TRUE;

for (i = 0; types[i]; i++)

if (endWith(nume, types[i])) return TRUE;

return FALSE;

}

void bs2s(char nou[], char vechi[]) {

// Este efectiva doar la Windows. Pentru Unix este inefectiv

char \*p;

int i, j;

strcpy(nou, vechi);

for ( ; ; ) {

p = strchr(nou, '\\');

if (p == NULL) return;

for (i = 0; p[i] == '\\'; i++);

p[0] = '/';

if (i == 1) continue;

for (j = i; p[j]; p[j - i + 1] = p[j], j++);

}

}

void aFile(char numed[]) {

char numef[MAX];

DIR \*dir;

struct dirent \*d;

struct stat status;

dir = opendir(numed);

if (dir != NULL) {

while (d = readdir(dir)) {

if (d->d\_name[0] == '.') continue;

strcpy(numef, numed);

strcat(numef, "/");

strcat(numef, d->d\_name);

aFile(numef);

}

}

closedir(dir);

stat(numed, &status);

if (isType(numed) == FALSE) return;

if (!S\_ISREG(status.st\_mode)) return;

if (status.st\_mtime < lastTime) return;

bs2s(numef, numed);

printf("%d\t%s\n", ++nr, numef);

}

int main(int argc, char \*argv[]) {

int i, j;

char dir[MAX], absDir[MAX];

char \*help[] = {

"Se afiseaza numele fisierelor de anumite tipuri create dupa un moment dat.",

"Sunt trei feluri de argumente si se pot scrie in orice ordine:",

" - directorul de start, in absenta .; incepe cu / sau ./ sau ../",

" - momentul de start, in absenta 0; numarat in secunde de la 1970\_01\_01, 31536000 secunde / an",

" - tipurile fisierelor create dupa momentul de start; incep cu . urmat de litere si / sau cifre",

" In absenta tipurilor sunt date nuele tuturor fisierelor create dupa momentul de start.",

"De exemplu, fisierele c si java din directorul parinte dupa 2020 se obtin prin:",

"./lsRecursiv 1576800000 .java .c ../", NULL

};

dir[0] = '.';

dir[1] = 0;

start = time(NULL);

for (i=0; help[i]; printf("%s\n", help[i]), i++);

for (i=1, j=0; argv[i]; i++) {

if (argv[i][0] == '/' ||

(strlen(argv[i]) >= 2 && argv[i][0] == '.' && argv[i][1] == '/') ||

(strlen(argv[i]) >= 3 && argv[i][0] == '.' && argv[i][1] == '.' ||argv[i][2] == '/'))

strcpy(dir, argv[i]);

else if (argv[i][0] == '.') types[j++] = argv[i];

else if (isdigit(argv[i][0])) lastTime = atol(argv[i]);

else ; // toate celelalte stringuri se ignora

}

realpath(dir, absDir);

bs2s(dir, absDir);

types[j] = NULL;

printf("Director start: %s\nMoment de start: %ld\nExtensii: ",dir, lastTime);

for (i=0; types[i]; i++) printf("%s ",types[i]);

printf("\n");

aFile(absDir);

printf("Durata in secunde: %ld\n",time(NULL)-start);

fflush(stdout);

return 0;

}

### Parcurgerea recursivă a fişierelor dintr-un director şi descendenţi - Go

package main

import (

"os"

"time"

"path/filepath"

"strconv"

"strings"

"fmt"

)

var types = []string{}

var lastTime = int64(0)

func isType(nume string) bool {

if len(types) == 0 {return true }

for \_,typ := range types {

if strings.HasSuffix(nume, typ) { return true }

}

return false

}

func bs2s(vechi string) string {

nou := strings.ReplaceAll(vechi, "\\\\", "\\")

nou = strings.ReplaceAll(nou, "\\", "/")

return nou

}

func aFile(absDir string) {

nr := 0

filepath.Walk(absDir, func(path string, info os.FileInfo, err error) error {

if info.IsDir() { return nil }

if !isType(path) { return nil }

t,\_ := os.Stat(path)

ts := strconv.FormatInt(t.ModTime().UnixNano(), 10)

ti,\_ := strconv.ParseInt(ts, 10, 64)

if ti < lastTime { return nil }

nr++

fmt.Printf("%d\t%s\n", nr, bs2s(path))

return nil

})

}

func main() {

help := `Se afiseaza numele fisierelor de anumite tipuri create dupa un moment dat.

Sunt trei feluri de argumente si se pot scrie in orice ordine:

- directorul de start, in absenta .; incepe cu / sau ./ sau ../

- momentul de start, in absenta 0; numarat in secunde de la 1970\_01\_01, 31536000 secunde / an

- tipurile fisierelor create dupa momentul de start; incep cu . urmat de litere si / sau cifre

In absenta tipurilor sunt date nuele tuturor fisierelor create dupa momentul de start.

De exemplu, fisierele c si java din directorul parinte dupa 2020 se obtin prin:

./lsRecursiv 1576800000 .java .c ../`

dir := "."

start := time.Now()

fmt.Printf("%s\n", help)

for \_,arg := range os.Args[1:] {

if len(arg) >= 3 && (arg[:1] == "/" || arg[:2] == "./" || arg[:3] =="../") {

dir = arg

} else if arg[:1] == "." {

types = append(types, arg)

} else if '0' <= arg[0] && arg[0] <= 9 {

lastTime,\_ = strconv.ParseInt(arg, 10, 64)

}

}

absDir,\_ := filepath.Abs(dir)

fmt.Printf("Director start: %s Moment de start: %d Extensii: %v", bs2s(absDir), lastTime, types)

fmt.Printf("\n")

aFile(absDir)

t := time.Now()

d := t.Sub(start) / 1000000

fmt.Printf("Durata in secunde: %d\n", d)

}

### Parcurgerea recursivă a fişierelor dintr-un director şi descendenţi - Python

import sys

import os

import time

types = []

lastTime = 0

def isType(nume):

global types

if len(types) == 0: return True

for type in types:

if nume.endswith(type): return True

return False

def bs2s(vechi):

nou = vechi.replace("\\\\", "\\")

nou = nou.replace("\\", "/")

return nou

def aFile(absDir):

global lastTime

global types

nr = 0

for root, numed, numef in os.walk(absDir):

for nume in numef:

abs = os.path.realpath(os.path.join(root, nume))

status = os.stat(abs)

if not isType(abs): continue

if status.st\_mtime < lastTime: continue

nr += 1

print(str(nr)+"\t"+bs2s(abs))

def main():

help = """Se afiseaza numele fisierelor de anumite tipuri create dupa un moment dat.

Sunt trei feluri de argumente si se pot scrie in orice ordine:

- directorul de start, in absenta .; incepe cu / sau ./ sau ../

- momentul de start, in absenta 0; numarat in secunde de la 1970\_01\_01, 31536000 secunde / an

- tipurile fisierelor create dupa momentul de start; incep cu . urmat de litere si / sau cifre

In absenta tipurilor sunt date nuele tuturor fisierelor create dupa momentul de start.

De exemplu, fisierele c si java din directorul parinte dupa 2020 se obtin prin:

./lsRecursiv 1576800000 .java .c ../"""

global types

global lastTime

dir = "."

start = time.time()

print(help)

for arg in sys.argv[1:]:

if arg.startswith("/") or arg.startswith("./") or arg.startswith("../"): dir = arg

elif arg.startswith("."): types.append(arg)

elif arg.isdigit(): lastTime = int(arg)

else: pass #toate celelalte stringuri se ignora

absDir = os.path.realpath(dir)

print("Director start: "+bs2s(str(absDir))+" Moment de start: "+str(lastTime)+" Extensii: "+str(types))

aFile(absDir)

print("Durata in secunde: "+str(time.time()-start))

sys.stdout.flush()

main()

## Probleme propuse

# Comenzi, expresii regulare, filtrele grep, sed, awk

## Comenzi

O *comandă* (Unix sau Windowsx) este de forma:

**ncomandă opţiuni expresii fişiere**

"**ncomandă**" este numele propriu-zis al comenzii;

"**opţiuni**" o opţiune Unix este specificată de obicei printr-o singură literă. In unele cazuri, litera este urmată de un argument şir de caractere sau număr întreg. Un grup de opţiuni este de obicei precedat de - (minus).

"**expresii**" sunt şiruri de caractere, utilizate ca argumente pentru comanda respectivă. Un caz particular sunt *expresiile regulare*, (care indică machete sintactice ale unor şiruri), de care ne ocupăm în secţiunea următoare.

"**fişiere**" reprezintă unul sau mai multe fişiere specificate relativ (doar numele acestora) sau absolut (cu cale completă) sau specificări generice ale unor familii de fişiere.

a. Space is separator

b. First word is the command

c. Next words are arguments

i. Values: ls /etc

ii. Options

1. Short form: ls -l

2. Long form: ls --all

3. Short form with value: cut –d : –f 1,2,3 /etc/passwd

4. Long form with value: cut –delimiter=: –fields=1,2,3 /etc/passwd

In mod implicit, comenzilor Unix le sunt asociate trei fişiere: *fişierul standard de intrare*, *fişierul standard de ieşire, fişierul standard de erori*. Uneori, dacă se doreşte referirea la ele în linia de comandă, aceste fişiere pot fi specificate prin &0 &1 &2 sau 0, 1 2.

La terminarea execuţiei oricărei comenzi în sistem se returnează un număr întreg, numit *cod de retur* sau *exit status*. In general, codul de retur ‘0’ denotă faptul că execuţia comenzii s-a încheiat cu success.

Fişierul standard ce intrare este asociat implicit tastaturii, iar celelalte două sunt asociate monitorului. Aceste asocieri pot fi modificate (*redirectate, redirecţionate*) astfel:

**comanda <fin** datele de intrare (input-ul) pentru comandă se vor prelua din fişierul text **fin**, pregătit în prealabil.

**comanda >fout** sau **comanda >>fout** ieşirea standard va fi depusă în fişierul **fout**; dacă se foloseşte semnul "**>**" se va crea un fişier nou cu numele specificat în care se va scrie output-ul comenzii (în cazul în care fişierul există, conţinutul acestuia este suprascris); când se utilizează "**>>**" output-ul comenzii este adăugat la sfârşitul fişierului **fout** dacă fişierul există deja, în caz contrar creându-se fişierul respectiv.

**2>&1** specifică faptul că pentru comandă fişierul de erori standard va fi acelaşi cu fişierul de ieşire standard.

**comanda1 | comanda2** ieşirea standard pentru **comanda1** se constituie automat în intrare standard pentru **comanda2** (conectare în *pipe*). Cele două comenzi se execută în paralel, fără a se crea o ieşire standard pentru **comanda1**, nici intrare standard pentru **comanda2**. Cele două comenzi aşteaptă una după cealaltă livrarea / primirea de octeţi prin acest pipe.

## Deosebiri formale Unix Windows:

|  |  |  |  |
| --- | --- | --- | --- |
|  |  | Unix | Windows |
| 1 | Specificare absoluta fisier | /dir1/dir2/.../dirn/fisier | d:\dir1\dir2\...\dirn\fisier |
| 2 | Separator directoare PATH | dir1:dir2:...:dirn | dir1;dir2;...;dirn |
| 3 | Specificare optiune | com -opt | com /opt |
| 4 | Separtor linii in fisier text  (Mac OS linie\rlinie CR) | linie\nlinie (LF = 0A) | linie\r\nlinie (CR LF = 0D 0A) |
| 5 | Parametrii linie comanda:  com arg1 arg2 ... argn | $0 $1 ... $9 | %0 %1 ... %9 |
| 6 | Valoarea unei variabile shell | ${nume} | %nume% |

## Expresii regulare: definire şi exemple

***Expresiile******regulare*** sunt şabloane care indică anumite forme sintactice care trebuie să le aibă stringurile care **satisfac** aceste şabloane.

In cele ce urmează vom descrie, în cadrul cel mai general, notaţiile folosite **în expresiile regulare Unix.** Există multe tipuri de expresii regulare: Python, PERL, Java C++ etc. In esenţă există asemănări între acestea, dar din păcate sunt şi deosebiri. Mai mult, chiar expresiile regulare Unix sunt specifice unor anumite comenzi şi pot să difere, nesemnificativ, între ele.

In continuare, prin **c**, **c1** şi **c2** vom nota caractere, iar prin **r**, **r1** şi **r2** vom nota expresii regulare deja construite. Tabelul următor descrie machetele expresiilor regulare Unix:

|  |  |
| --- | --- |
| **Expresie regulară** | **Semnificaţie** |
| **.** | orice caracter |
| **\c** | caracterul **c** îşi pierde eventualul statut de caracter special |
| **[lista]** | un singur caracter, oricare din **lista** |
| **[c1-c2]** | orice caracter cuprins lexicografic între caracterele **c1** şi **c2** |
| **[^lista]** | negaţia lui **[lista]**, deci un singur caracter, care nu este în listă |
| **^** | următorul şablon se aplică numai la început de linie |
| **$** | următorul şablon se aplică numai la sfârşit de linie |
| **\<** | semnifică început de cuvânt (un cuvânt este format din litere, cifre sau -, orice alt caracter este considerat separator) |
| **\>** | semnifică sfârşit de cuvânt |
| **r\*** | şirul vid sau concatenarea repetată a expresiei regulare **r** ori de câte ori |
| **r+** | concatenarea repetată a expresiei regulare **r** cel puţin o dată |
| **r?** | şirul vid sau expresia regulară **r** |
| **(r)** | expresia **r** privită ca o singură entitate; în anumite situaţii este **\(r\)** |
| **r1 r2** | rezultatul concatenării expresiei regulare **r1** urmată de **r2** |
| **r1 | r2** | fie expresia regulară **r1**, fie expresia regulară **r2** |
| **r\{n,m\}** | repetă expresia regulară **r** de cel puţin **n** ori şi de cel mult **m** ori |
| **n,m** | partea de text dintre liniile **n** şi **m** |
| **. (caracterul punct)** | indică la editare linia curentă |
| **$ (caracterul dolar)** | indică la editare ultima linie |
| **/şir/** | prima dintre liniile următoare faţă de linia curentă care conţine **şir** |
| **?şir?** | prima.dintre liniile precedente faţă de linia curentă care conţine **şir** |

Iată câteva exemple:

[123] - oricare dintre cifrele 1, 2, sau 3

[123 ] - 1, 2, 3, sau spatiu

[a-z] - orice litera mica

[aeiou] - orice vocala

[A-Z] - orice litera mare

[0-9] - orice cifra

[^0-9] - orice care nu este cifra

[^,.:] - orice cu exceptia virgula, punct, doua puncte

O construcţie de forma:

[0-3][0-9][-/][0-1][0-9][-/][0-9]

indică scrierea unei date calendaristice. Printre formele admise amintim:

ZZ-LL-AA, ZZ/LL-AA, ZZ-LL/AA.

Numar real in diverse limbaje de programare:

[-+]?([0-9]+\.?|\.[0-9])[0-9]\*([eE][-+]?[0-9]+)?

De exemplu, numarul lui Avogadro se scrie: 6.023E+23

Sintaxa unei adrese email se poate defini, mai mult sau mai puţin exact:

+\@.+\..+

sau

[A-Z0-9.\_%+-]+@[A-Z0-9.-]+\.[A-Z]{2,4}

Sintaxa unui url:

^http:\/\/[a-zA-Z0-9\_\-]+\.[a-zA-Z0-9\_\-]+\.[a-zA-Z0-9\_\-]+$

sau

^(https?:\/\/)?[0-9a-zA-Z]+\.[-\_0-9a-zA-Z]+\.[0-9a-zA-Z]+$

Expresii regulare Go through the aspects above, giving the following examples for the rules

a. .\* – any sequence of characters

b. [a-zA-Z02468] – any letter, regardless of its case, and any even digit

c. [ \t] – space or tab

d. ^[^0-9]\+$ – non-empty lines containing any characters except digits

e. \([Nn][Oo] \)\+ - any refusal, no matter how insistent (eg No no no no no)

## Clasificarea comenzilor; comenzi filtru

### Clasificarea comenzilor Unix

Comenzile Unix pot fi clasificate în:

* Comenzi interne incluse in sh
* Comenzi externe, fie din setul standard Unix, fie elaborate de utilizatori:
  + Fişiere rezultate din compilări ale unor limbaje de programare (majoritar C)
  + Scripturi (fişiere de comenzi).

Nu avem în intentie prezentarea comenzilor Unix, ci recomandăm consultarea manualelor acestora: $ man [sectiune] numecomanda

Dintre cele mai populare comenzi Unix amintim:

* Filtre Unix - de regulă preiau intrarea standard, o transformă şi o dau la ieşirea standard: grep, sed, sort, uniq, cut, awk
* Comenzi de lucru cu fisiere (+directoare): ls, pwd, cat, find, locate, file, more, less, rm, mkdir, rmdir, cp, mv, cd, chmod, chown, ln, touch, du, cmp, diff, head, tail, split, wc
* Comenzi pentru aflarea de informatii despre useri: finger, w, who, ps, last, id, users
* Comenzi pentru informatii de retea: netstat, ping, hostname, host, ftp, who
* Alte comezi: clear, date, mail, uptime, df, fg, bg

Pentru aceste filtre prezentăm sumar sintexele şi un exemplu. Pentru detalii se pot consulta manualele acestor filtre.

### Filtrul grep

caută în unul sau mai multe fişiere (sau în intrarea standard) linii care satisfac o anumită expresie regulară. Distribuţiile Unix oferă mai multe variante: grep, egrep, fgrep, rgrep. In sintaxă fie se specifică direct expresia regulară prin pattern, fie acest pattern se depune într-un fişier:

grep [*optiuni*] [-e *pattern* | -f fis\_pattern] [fisier ...]

De exemplu, să se tipărească liniile din fişierul linii.c care conţin vocale scrise cu majuscule:

grep -e [AEIOU] linii.c

grep Let’s search for things in file /etc/passwd

a. Display all lines containing “dan”. The solution is below

i. grep “dan” /etc/passwd

b. Display the line of username “dan”. The username is the first field on the line, it is not empty, and it ends at the first :. We will rely on these aspects to insure that we only search the usernames, and not anything else.

i. grep –i “^dan:” /etc/passwd

c. Display the lines of all users who do not have digits in their username.

i. grep “^[^0-9]\+:” /etc/passwd

d. Display the lines of all users who have at least two vowels in their username. This is a little tricky, because the vowels do not need to be consecutive, so we need to allow for any characters between the vowels (including none), but we cannot allow : to appear between vowels, or else we would be searching outside the username.

i. grep -i "^[^:]\*[aeiou][^:]\*[aeiou][^:]\*:" /etc/passwd

ii. grep -i "^[^:]\*\([aeiou][^:]\*\)\{2,\}:" /etc/passwd

e. There will be lots of users displayed for the problem above, so let’s search for usernames with at least 5 vowels in their username. The first solution above will be really long for this case, but the second will be very easy to adapt, by changing 2 into 5.

i. grep -i "^[^:]\*\([aeiou][^:]\*\)\{5,\}:" /etc/passwd

f. Display the lines of all the users not having bash as their shell. The shell is the last value on the line, so we will use that when searching.

i. grep -v "/bash$" /etc/passwd

g. Display the lines of all users named Ion. We will have to search in the user-info field (the fifth field) of each line, ignore the upper/lower case of the letters, and insure that we do not display anybody containing the sequence “ion” in their names (eg Simion, Simionescu, or Ionescu).

i. grep -i "^\([^:]\*:\)\{4\}[^:]\*\<ion\>" /etc/passwd

Let’s consider a random text file a.txt, and search for things in it

a. Display all the non-empty lines

i. grep “.” a.txt

b. Display all the empty lines

i. grep “^$” a.txt

Display all lines containing an odd number of characters

i. grep “^\(..\)\*.$” a.txt

d. Display all lines containing an ocean name

i. grep –i “\<atlantic\>\|\<pacific\>\|\<indian\>\|\<arctic\>\|\<antarctic\>” a.txt

e. Display all lines containing an email address

i. What does an email address look like? It has the following structure.

username – let’s assume it can contain any character, except for @, \*, !, and ?

@ - separator between the username and the hostname

a. Sequence of at least two elements separated by .

b. Let’s assume an element can contain any letter, digit, dash, or underscore

ii. grep -i “\<[^@\*\!?]\+@[a-z0-9\_-]\+\(\.[a-z0-9\_-]\+\)\+\>” a.txt

### Filtrul sed (Stream EDitor)

este un editor de texte neconversaţional. El preia un fişier (sau mai multe, sau intrarea standard), aplică asupra lui comenzi de editare, după care rezultatul este tipărit la ieşirea standard (sau cu numele vechiului fişier daca se pune opţiunea -n).

sed [optiuni] [ -e comenziEdit | -f fis\_comenziEdit] [ fişier ... ]

De exemplu, să se elimine toate secvenţele **<cifră><literă><cifră>** dintr-un fişier:

$ sed "s/[0-9][a-z,A-Z][0-9]//g" fis1

Doua constructii de expresii regulare mai deosebite in sed. Constructia & (sau \& depinde de context) semnifica continutul patternului care s-a potrivit. De exemplu, comanda de editare

s/^[0-9][0-9]/(&)/g

pune în paranteză primele două cifre de la inceputul fiecarei linii: 4064123456 este înlocuit cu (40)64123456

Constructiile \1, \2, …, \9 semnifică prima, a doua, respectiv a 9-a potrivire din "expresie\_regulară")

$ cat phone.txt | sed 's/\(.\*)\)\(.\*-\)\(.\*$\)/Area code: \1 Second: \2 Third: \3/'

înlocuieşte fişierul iniţial phone.txt:

(555)555-1212

(555)555-1213

(555)555-1214

(666)555-1215

(666)555-1216

(777)555-1217

cu noul conţinut:

Area code: (555) Second: 555- Third: 1212

Area code: (555) Second: 555- Third: 1213

Area code: (555) Second: 555- Third: 1214

Area code: (666) Second: 555- Third: 1215

Area code: (666) Second: 555- Third: 1216

Area code: (777) Second: 555- Third: 1217

sed Let’s manipulate the content of /etc/passwd

a. Display all lines, replacing all vowels with spaces

i. sed “s/[aeiou]/ /gi” /etc/passwd

b. Display all lines, converting all vowels to upper case

i. sed “y/aeiou/AEIOU/” /etc/passwd

c. Display all lines, deleting those containing numbers of five or more digits:

i. sed “/[0-9]\{5,\}/d” /etc/passwd

d. Display all lines, swapping all pairs of letters

i. sed “s/\([a-z]\)\([a-z]\)/\2\1/gi” /etc/passwd

e. Display all lines, duplicating all vowels

i. sed “s/\([aeiou]\)/\1\1/gi” /etc/passwd

### Filtrul sort

Preia şi eventual concatenează fişierele specificate în intrare şi le dă la ieşirea standard cu liniile ordonate alfabetic. In absenţa specificării fişîerelor de intrare, se ia intrarea standard.

sort [optiuni] [ fisier . . . ]

Comanda are un mare număr de opţiuni şi acestea pot diferi de la un tip de Unix la altul. Oricum, prin opţiuni se pot specifica tipurile de comparaţii de sortare (text, numeric, date calendaristice etc. De asemenea, se pot indica porţiunile din linii şi ordinea acestora care se compară în vederea stabilirii ordinii. In fine, se poate cere eliminarea duplicatelor. In absenţa opţiunilor, la ieşire se furnizează liniile ordonate alfabetic.

### Filtrul uniq

tratează liniile adiacente care sunt identice (NU face sortare!).

uniq [optiuni] [ fisier ]

Fără opţiuni, din liniile adiacente identice se dă la ieşire doar prima.

### Filtrul cut

Preia linii de la fişierul (fişierele) de intrare sau de la intrarea standard şi la ieşire, din fiecare linie, dă la ieşire numai porţiunile indicate.

Indicarea porţiunilor se face prin liste cu construcţii de forma n n-m n- -m, unde n şi m sunt întregi, n < m, indicând numărul elementului (începând cu 1), de la n la m inclusiv, de la n pana la sfârşit, de la început până la m.

Numărarea se face la nivel de octet (-b), de caracter (-c) sau de câmp (-f). Diferenţa între caracter şi octet apare atunci când liniile conţin octeţi cu codul între 128 şi 255 (caractere din ASCII extins). Apelurile posibile sunt:

cut -b lista [ fisier . . . ]

cut -c lista [ fisier . . . ]

cut -f lista [-d delimitator] [ fisier . . . ]

Caracterul delimitator implicit este TAB, dar în locul lui se poate specifica altul, prin -d.

Ca exemplu, să considerăm un fişier FIS ce conţine:

foo:bar:baz:qux:quux

one:two:three:four:five:six:seven

alpha:beta:gamma:delta:epsilon:zeta:eta:theta:iota:kappa:lambda:mu

the quick brown fox jumps over the lazy dog

Efectul următoarelor trei rulări sunt:

$ cut -d ":" -f 5- FILE

quux

five:six:seven

epsilon:zeta:eta:theta:iota:kappa:lambda:mu

the quick brown fox jumps over the lazy dog

$ cut -c 4-10 FILE

:bar:ba

:two:th

ha:beta

quick

$ cut -c 4-10,14,16- FILE

:bar:bau:quux

:two:th:our:five:six:seven

ha:betama:delta:epsilon:zeta:eta:theta:iota:kappa:lambda:mu

quick w fox jumps over the lazy dog

## awk; programarea în awk

### Apelul şi definirea programelor awk

Acest utilitar prelucrează fişiere text, selectând acele linii din text care satisfac anumite conditii (şabloane, expresii regulare), carora li se aplica o serie de actiuni. Numele utilitarului vine de la cei trei proiectanţi şi implementatori ai lui: A. Aho, P. Wieinberger şi B. Kerninghan. In prezent există mai multe variante îmbunătăţite: **gawk, mawk, nawk** etc. Noi vom trata doar varianta standard. Sintaxa comenzii este:

awk [ -f fisier\_program | 'program' ] [ -Fc ] [ [ -v ] var=val ... ] [fisier ... ]

Explicăm parametrii începand din dreapta:

*Intrarea* în awk este constituită din lista de fişiere ale căror nume sunt date în linia de comandă: fisier . . . Aceasta lista poate lipsi, caz in care se prelucreaza intrarea standard.

*Rezultatul* filtrării prin awk este afişat la ieşirea standard.

Optiunea -v precede definirea unor variabile globale si a valorilor acestora: var=val .... Nu sunt necesare decăt specificările atribuirilor var=val, specificarea -v poate lipsi. Mai mult, prezenta lui -v in fisier de comenzi nu functioneaza! Este vorba de mecanismele de tratare a optiunilor atunci cand si awk si shell le trateaza.

Optiunea -F specifica caracterul c care va fi separator de cuvinte. In absenta optiunii, separatorul implicit este orice spatiu alb (BLANK, TAB \n \r).

**program** poate fi scris fie direct in comanda awk, fie pregatit in prealabil in **fisier\_program** si indicat prin optiunea -**f**. Liniile din **program** sunt de forma:

conditie { instructiuni}

awk tratează pe rând câte o linie din fişierele de intrare şi pentru fiecare execută instructiuni atunci când conditie ia valoarea true. de Dacă conditie lipseşte atunci se considera implicit adevarata. .

Sintaxa conditiilor si a instructiunilor sunt similare cu cele din limbajul C. Variabilele nu trebuie sa fie declarate, ele se initializeaza automat. Tipul lor se deduce din context. Initial, valorile variabilelor sunt 0 pentru numere si "" (sirul vid) pentru siruri. Operanzii pot fi expresii aritmetice, expresii relationale, constante şi variabile. Pentru variabilele de tip şir operatorul de concatenare este spatiul. Exista câteva funcţii de lucru cu şiruri. Se pot folosi variabile de tip tablou ale căror indici pot să fie numerici (cu numerotarea incepand de la 1) sau şiruri de caractere - acestea din urmă sunt **tablouri asociative.**.

*Expresiile* sunt sunt cele din C. Operatorii relaţionali se extind asupra stringurilor. Pentru stringuri există *operatorii de potrivire* cu expresiile regulare **expr ~ /expreg/** pentru potrivire şi **expr !~ /expreg/** pentru nepotrivire.

conditie - este o expresie logică construită cu operatorii din C: ||, &&, !, ().

*Condiţii predefinite:*

* BEGIN este adevarată înainte de prima linie din primul fişier
* END este adevarată după ultima linie din ultimul fişier

*Variabile predefinite:*

* NF - numărul de cuvinte (câmpuri) din linia curentă, cuvintele notate **$1, $2, . . . $NF**.
* NR - numărul de ordine al liniei curente (numărătoarea începe de la 1) ce include lungimile fisierelor deja prelucrate plus cea curenta a fisierului curent.
* FNR - numărul de ordine al liniei curente din fisierul curent; liniile cu nr. 1 sunt primele linii din fiecare fisier; numaratoare începe de la 1 la începutul fiecărui fişier
* FS - separatorul de câmpuri (spatiul alb sau optiunea -F)
* FILENAME - numele fişierului curent care este tratat
* OFS - separator de câmpuri la ieşire (implicit este spaţiu)
* ORS - separator de înregistrări la ieşire (implicit este linie nouă)
* ARGV – şirul parametrilor din linia de comandă. **specificarea program sau -f fisier\_program nu se ia in considerare ca argument.**
* ARGC – numărul parametrilor din linia de comandă. **Vezi mai sus.**
* variabilele globale definite prin optiunea -v.

*Accesarea câmpurilor:*

* se face cu $1, $2 ...$i, $(i+1), $NF, iar întreaga linie se referă cu $0
* sir1 sir2 este operatia de concatenare a sirurilor; se face scriind unul dupa altul sirurile de concatenat

*Funcţii predefinite:*

* length(sir) - lungime şir; length <=> length($0)
* substr(s,p,n) - subşirul lui s care începe la poziţia p şi are lungimea n
* index(s1,s2) - întoarce poziţia la care s2 apare în s1 sau 0 la absenţă
* sprintf(format, arg1,..) - întoarce ca rezultat şirul pe care printf l-ar tipări în C
* split(s,a,c) - unde s este şir, a este tablou şi c un caracter. Împarte şirul s în câmpuri considerând ca separator caracterul c; dacă c lipseşte atunci separatorul implicit este FS. Valorile împărţite sunt date ca valori elementelor tabloului a
* system(cmd) - executa comanda shell cmd si returneaza codul sau de retur

*Instrucţiuni*

* variabilă = exprsie
* instrucţiunile if, for, while ca şi în C
* for (i in numetablou) instrucţiune. i ia ca valori indicii lui numetablou şi se execută instrucţiune pentru fiecare valoare a lui i
* ; este separator de instrucţiuni
* } este separator de linie, continuarea unei linii se face cu caracterul \ pe ultima poziţie din linie
* print listă-expresii [ > nume-fiş] - afişează la ieşirea standard valoarea expresiilor separate prin OFS, iar la sfârşit de linie pune ORS. Dacă se specifica >nume-fis atunci scrierea se face în fişierul nume-fis.

:

### Ilustrare moduri de apel: numararea liniilor, cuvintelor si caracterelor

Pentru a ilustra modurile de apel awk, am ales un program de numărare linii, cuvinte şi caractere din fişierul numit deprelucrat.

Mai intai solutia cu programul **awk** scris intr-un fisier separat. Vom pregati in fisierul **fisp** programul:

{car += length($0)+1; cuv += NF;}

END {print "Fisier:" FILENAME, "Linii:" NR, "Cuvinte:" cuv, "Caractere:" car;}

(la length($0) se adauga 1 pentru a numara terminatorul de linie). Comanda de numarare va fi:

awk -f fisp deprelucrat

Dacă fişierul **deprelucrat** are conţinutul:

ggg oooiioi jxj

jjj

rezultatul executiei va fi:

Fisier:deprelucrat Linii:2 Cuvinte:4 Caractere:20

Si acum, varianta cu program scris direct in linia de comanda:

awk '{car += length($0)+1; cuv += NF;}\

END {print "Fisier:" FILENAME, "Linii:" NR, "Cuvinte:" cuv, "Caractere:" car;}'\

deprelucrat

Intr-o constructie de forma:

comanda | awk -f fisp

awk va prelucra iesirea standard data de comanda.

Prelucrarea a trei fisiere se face:

awk -f fisp fisier1 fisier2 fisier3

**In acest caz se va tipari numele ultimului fisier (fisier3), iar numarul de linii, cuvinte si caractere sunt valori cumulate din cele trei fisiere!**

### Câteva exemple simple

In exemplele care urmeaza vom include program direct in linia de comanda awk.

Manipulate the content of /etc/passwd, with the program provided on the command line

a. Display all the usernames, but only the usernames, and nothing else. We will use argument –F to tell AWK that the input file is separated by : , and then we will print the first field of each line, by not providing any selector for the block.

i. awk –F: ‘{print $1}’ /etc/passwd

b. Print the full name (the user info field) of the users on odd lines

i. awk –F: ‘NR % 2 == 1 {print $5}’ /etc/passwd

c. Print the home directory of users having their usernames start with a vowel

i. awk -F: '/^[aeiouAEIOU]/ {print $6}' /etc/passwd

d. Print the full name of users having even user ids

i. awk -F: '$3 % 2 == 0 {print $5}' /etc/passwd

e. Display the username of all users having their last field end with “nologin”

i. awk -F: '$NF ~ /nologin$/ {print $1}' /etc/passwd

f. Display the full names of all users having their username longer than 10 characters

i. awk -F: 'length($1) > 10 {print $5}' /etc/passwd

Keep using /etc/passwd as input file, but provide AWK programs in a file. The command will look like

a. awk -F: -f prog.awk /etc/passwd

b. Provide the content of file prog.awk so that the command above will print all user on even line having a group id less than 20: NR % 2 == 0 && $4 < 20 { print $5 }

c. Display the sum of all user ids BEGIN { sum=0 }\n{ sum += $3 }\nEND {print sum }

d. Display the product of the differences between the user id and the group id BEGIN { prod=1 }\n{ prod

\*= $3-$4 }\nEND { print prod }

### Afişarea primului cuvânt din fiecare linie

Vom lua ca fişier de intrare fişierul deprelucrat:

awk '{ print $1}' deprelucrat

Rezultatul execuţiei va fi:

ggg

jjj

### Afişarea liniilor care au un anumit ultim cuvânt.

Să se scrie un program care să afiseze liniile din fisierul f pentru care ultimul cuvant este CevaAiurea si numarul curent al fiecarei astfel de linii. Prezentam rezolvarea in trei variante: direct, cu variabila globala ultim si cu ARGV.

awk '$NF == "CevaAiurea" {print "direct:", NF, $0;}' f

awk '$NF == ultim {print "cu var:", NF, $0;}' \-v ultim=CevaAiurea f

awk '$NF == ARGV[2] {print "cu ARGV:", NF, $0;}' f CevaAiurea

# Dupa prelucrarea lui f va spune ca nu gaseste fisierul CevaAiurea.

awk '$NF == ultim {print "ca parametru in script shell:", NF, $0;}' ultim="$1" f

# CevaAiurea se va da la linia de comanda a fisierului shell

Daca fişierul **f** are conţinutul:

CevaAiurea

uguui iuuhih

hphph poihphp CevaAiurea

gggg

atunci rezultatul celor patru executii va fi:

SIR 1 CevaAiurea

SIR 3 hphph poihphp CevaAiurea

In loc de SIR apare direct: cu var: cu ARGV: ca parametru in script shell: cu mesaj de eroare (CevaAiurea no such file). Executia a patra trebuie inclusa intr-un script shell si CevaAiurea primul parametru al acestuia.

### Să se afişeze liniile mai lungi de 5 caractere

Am ales ca intrare fişierul nume\_fis. Afişarea acestor linii să se facă în ordinea inversă apariţiei lor, **pentru fiecare fisier in parte**. Mai intai prezentam varianta simpla, cand se da la intrare un singur fişier, apoi varianta generală, care trateaza mai multe fisiere de intrare:

# Solutia cu un singur fisier

awk 'length>5 {x[++n]=$0;}\

END {for ( ; n>=0; n--) print x[n];}' nume\_fis

# Solutia cu mai multe fisiere

awk 'END {print "Fisierul:"fisier; for ( ; n>=0; n--) print x[n];}\

NR>1 && FNR==1 {print "Fisierul:"fisier; for ( ; n>=0; n--) print x[n]; n=0;}\

FNR==1 {fisier=FILENAME;}\

length>5 {x[++n]=$0;}' nume\_fis nume\_fis f awk3

La varianta cu mai multe fisiere trebuie remarcată succesiunea celor 4 conditii pentru a "prinde" numele vechiului fisier atunci cand a aparut deja fisierul cel nou. Se vor tipări liniile mai lungi de 5 caractere din cele 4 fisiere (primele doua sunt de fapt acelasi fisier.

### Prelucrari asupra unui fisier cu campuri fixe.

Sa presupunem ca avem fisierul text log cu linii avand fiecare dintre ele cinci campuri care inventariaza activitatile unor useri conectati la niste servere:

popescu www.scs.ubbcluj.ro azi 60 130

ionescu www.scs.ubbcluj.ro maine 3 20

dan linux.scs.ubbcluj.ro ieri 7 400

popescu www.scs.ubbcluj.ro azi 20 130

dan www.scs.ubbcluj.ro ieri 35 20

dan linux.scs.ubbcluj.ro alaltaieri 400 10

Cele 5 campuri inseamna:

User AdresaServer DataConectarii DurataConectarii SiteuriAccesate

Se cer:

1. Pentru fiecare data, numarul total de useri conectati, durata totala a conectarilor.
2. Pentru fiecare user numarul total de conectari, durata totala a conectarilor, totalul siteurilor accesate, cea mai lunga conectare.
3. Pentru fiecare server, numarul total de conectari, durata totala a acestora, cea mai scurta conectare.

Programele celor trei cerinte sunt:

#awk5 cerinta a.

awk 'NF >= 4 {tuc[$3]++; dtc[$3]+=$4;}\

END {print "Solutie a:";\

print "Total useri conectati:"; for (u in tuc) print "\t", u, tuc[u];\

print "Durate totale conectari:"; for (u in dtc) print "\t", u, dtc[u];}' log

#awk5 cerinta b.

awk 'NF >= 5 {tc[$1]++; dtc[$1]+=$4; tsa[$1]+=$5; if ($4>clc) clc = $4;}\

END {print "Solutie b:";\

print "Total conectari:"; for (u in tc) print "\t", u, tc[u];\

print "Durata totala conectari:"; for (u in dtc) print "\t", u, dtc[u];\

print "Total site-uri accesate:"; for (u in tsa) print "\t", u, tsa[u];\

print "Cea mai lunga conectare:", clc;}' log

#awk5 cerinta c.

awk 'BEGIN {csc = 999999999;}\

END {print "Solutie c:";\

print "Total conectari:"; for (u in tc) print "\t", u, tc[u];\

print "Durate conectari:"; for (u in dtc) print "\t", u, dtc[u];\

print "Cea mai scurta conectare:" csc;}\

NF >= 4 {tc[$2]++; dtc[$2]+=$4; if ($4 < csc) csc = $4;}' log

Solutiile celor trei rulari sunt:

Solutie a:

Total useri conectati:

alaltaieri 1

ieri 2

azi 2

maine 1

Durate totale conectari:

alaltaieri 400

ieri 42

azi 80

maine 3

Solutie b:

Total conectari:

popescu 2

ionescu 1

dan 3

Durata totala conectari:

popescu 80

ionescu 3

dan 442

Total site-uri accesate:

popescu 260

ionescu 20

dan 430

Cea mai lunga conectare: 400

Solutie c:

Total conectari:

www.scs.ubbcluj.ro 4

linux.scs.ubbcluj.ro 2

Durate conectari:

www.scs.ubbcluj.ro 118

linux.scs.ubbcluj.ro 407

Cea mai scurta conectare:3

### Rearanjarea cuvintelor din liniile unui fisier

Se dă un string numit **dictionar de prefixe**, fiecare prefix urmat de ':', si un fisier text. Se cere crearea unui alt fisier, ale caror linii au acelasi continut ca si cele din fisierul initial, dar puse intr-o alta ordine. Noua ordine este urmatoarea: mai intai cuvintele care incep cu unul din prefixele din dictionar, in ordinea acestor prefixe; apoi cuvintele ramase, in ordinea din linia initiala. In noul fisier, cuvintele vor avea cate un sufix de forma: ";n", unde n este numarul de pozitie in linia initiala.

awk 'NR==1{split(dict, d, ":"); print dict;}\

$0 != "" {print $0;\

for (i=1; i<=NF; i++) if ($i != "") $i = $i ":" i;\

i=0;

for (k=1; d[k]!=""; k++)\

for (j=i+1; j<=NF; j++)\

if (index($j,d[k])==1) {\

i++; t = $j;\

for (l=j; l>=i; l--) $l = $(l-1);\

$i = t;\

}\

print $0;}' dict={:c:FI: awk1

Ca fisier de prelucrat am folosit programul de rezolvare a exemplului 1 de mai sus. Ca dictionar am folosit trei cuvinte (intamplator primele doua au cate un caracter).

Ca rezultat se tipareste dictionarul si fiecare linie inainte si dupa prelucrare:

{:c:FI:

awk '{car += length($0)+1; cuv += NF;}\

cuv:5 awk:1 '{car:2 +=:3 length($0)+1;:4 +=:6 NF;}\:7

END {print "Fisier:" FILENAME, "Linii:" NR, "Cuvinte:" cuv, "Caractere:" car;}'\

{print:2 cuv,:8 car;}'\:10 FILENAME,:4 END:1 "Fisier:":3 "Linii:":5 NR,:6 "Cuvinte:":7 "Caractere:":9

deprelucrat

deprelucrat:1

## Probleme propuse

# Programare shell

## Ce este un script (fişier de comenzi) shell

Un **script** (fişier de comenzi) este un fişier text care conţine în el:

* comenzi Unix;
* directive (ale interpretorului de comenzi shell) de control al fluxului execuţiei acestor comenzi.

Unscript se comporta, la randul lui, ca si o comanda shell. **Numele unui fişier script nu trebuie să respecte nici o cerinţă sintactică!**

Noi vom adăuga la numele de scripturi sufixul **.sh** ca o convenţie proprie de a ilustra conţinutul. In general, un script este folosit pentru a manevra fişiere din sistemul de fişiere. Dacă se doreşte intervenţia în interiorul fişierelor, de regulă a celor text, se folosesc comenzi filtru auxiliare: grep, sed, cut, awk etc.

Pentru testarea unor condiţii in structurile de control if sau while **se exploatează codul de retur al terminării comenzii precedente. Valoarea 0 a codului de retur reprezintă valoare de adevăr (TRUE), iar o valoare nenulă înseamnă FALSE**. De multe ori în condiţiile din if sau while se foloseşte, comanda test expresie ( notat uneori mai elegant [ expresie ] ) care întoarce codul de retur în funcţie de valoarea de adevăr a condiţiei din test.

Orice comanda shell se poate rula:

1. Direct la prompter în linia de comanda, aşa cum am vazut in seminarul precedent
2. Comanda se scrie intr-un script urmând a fi rulată odată cu execuţia scriptului, de care ne ocupăm în cele ce urmează.

Dacă script este numele unui fişier de comenzi din directorul curent, rularea acestuia se poate face:

1. ./script ... sau calebsoluta/script ... dacă fişierul script are drepturi de execuţie. Pentru fixarea drepturilor, în particular şi a celor de execuţie, se foloseşte comanda chmod.
2. script ... daca script are drepturi de executie şi dacă directorul curent este inscris în PATH
3. sh script ... sau sh caleabsoluta/script ... indiferent dacă are sau nu drepturi de execuţie.

Prin ... am notat: argumente ale comenzii, optiuni, fisiere, expresii, redirectari: < > >> <& >&

Daca o comanda trebuie reprezentata pe doua (sau mai multe) linii consecutive, toate liniile componente, cu exceptia ultimei, se incheie cu \<ENTER> (succesiunea de caractere '\\' '\n').

O succesiune de doua comenzi se separă prin: ; && || \n (linie noua) | (conectarea pipe a doua comenzi) & (executie in background). Mai multe comenzi pot fi grupate prin ( comenzi ) sau { comenzi }

Intr-o linie, dacă apare caracterul # (diez), atunci tot restul liniei este interpretat ca şi **comentariu**. Este indicat (nu obligatoriu) ca scriptul shell sa inceapa cu un comentariu special

#!/bin/sh sau #!/bin/bash sau #!/bin/ksh . . .

Prin aceasta se indica sub controlul cărui Shell se va interpreta scriptul: sh, bash, ksh etc. In cele ce urmează vom folosi numai scriptul sh.

## Structuri de control si comenzi folosite în programarea shell

**Structuri de control ale interpretorului sh** (directivele) sunt: if, case, for, while, until. Pentru specificarea acestor structuri se folosesc cuvintele rezervate if, then, elif, else, fi, case, in, esac, for, do, done, while, until. Sintaxele acestor construcţii sunt:

**if** *lc* **then** *lc* [**elif** *lc* **then** *lc* ]\* [**else** *lc* ]?**fi**

**case** *cuvant* **in** [ *cuvant* [ | *cuvant* ]\* **)** *lc* **;;** ]+ **esac**

**for** *nume* **do** *lc* **done**

**for** *nume* **in** [ *cuvant* ]+ **do** *lc* **done**

**while** *lc* **do** *lc* **done**

**until** *lc* **do** *lc* **done**

Prin *lc* am notat, generic, orice succesiune de comenzi, legate între ele prin ; | || &&

In strânsă legătură cu structurile de control acest context se pot folosi comenzile speciale true, false, break, continue. De asemenea, o serie de comenzi standard Unix sunt utile în principal în scripturi shell: shift, read, readonly, sleep, exit, echo, test (echivalent cu "[ ..]"), export, expr, basename, (apostroafele inverse)

Semantica acestor construcţii este similară cu cea întâlnită în limbajele de programare. Exemplele care urmează vor ilustra funcţionarea lor.

## Exemple de scripturi shell

### Fişiere generice

i. \* - Matches any sequence of characters, including a void sequence, but not the first dot in a filename

ii. ? - Matches any single character, but not the first dot in a filename

iii. [abc] – List of optional characters, support ranges like the regular expressions

iv. [!abc]- Negated list of optional characters (similar to [^abc] from regular expressions)

### Numărul total de linii de cod C din directorul dat ca parametru care nu sunt goale:

#!/bin/bash

S=0

for f in $1/\*.c; do

N=`grep "[^ \t]" $f | wc -l`

S=`expr $S + $N`

1. done

echo $S

Aceeaşi problemă, dar din toate subdirectoarele directorului dat ca parametru.

#!/bin/bash

S=0

for f in `find $1 –type f –name “\*.c”; do

N=`grep "[^ \t]" $f | wc -l`

S=`expr $S + $N`

done

echo $S

### Un exemplu de if

#!/bin/bash

for A in $@; do

if [ -f $A ]; then

echo $A is a file

elif [ -d $A ] then

echo $A is a dir

elif echo $A | grep -q "^[0-9]\+$"; then

echo $A is a number

else

echo We do not know what $A is

fi

done

### Citiri de la consolă şi test de existenţă / citire a fişierelor

#!/bin/bash

F=””

while [ -z "$F" ] || [ ! -f "$F" ] || [ ! -r "$F" ]; do

read -p "Provide an existing and readable file path:"$F

done

or

#!/bin/bash

F=””

while test -z "$F" || ! test -f "$F" || ! test -r "$F"; do

read -p "Provide an existing and readable file path:" $F

done

### Monitorizarea stării unui director şi anunţul modificării lui

#!/bin/bash

D=$1

if [ -z "$D" ]; then

echo "ERROR: No directory provided for monitoring" &gt;&2

exit 1

fi

if [ ! -d "$D" ]; then

echo "ERROR: Directory $D does not exist" &gt;&2

exit 1

fi

STATE=""

while true; do

S=""

for P in `find $D`; do

if [ -f $P ]; then

LS=`ls -l $P | sha1sum`

CONTENT=`sha1sum $P`

else

LS=`ls -l –d $P | sha1sum`

CONTENT =`ls -l $P | sha1sum`

fi

S="$S\n$LS $CONTENT"

done

if [ -n "$STATE" ] && [ "$S" != "$STATE" ]; then

echo "Directory state changed"

fi

STATE=$S

sleep 1

done

a. We use sha1sum to get a checksum that is statistically impossible to be identical for different contents

b. We checksum the details of the file (ls –l) as well as its content

c. For directories, we use the –d flag of ls to list the directory details and not its content, and we use the output

of ls –l for the directory content

### Anunţarea userilor cu directoare prea mari

#!/bin/bash

# check-home-dir-size.sh parent maxim

# parent este parintele directoarelor care trebuie verificate, iar directoarele

# au numele utilizatorilor.

# maxim este dimensiunea maxima in Ko

# Trebuie sa fie instalat mail, de exemplu sudo apt install mailutils

parent=$1

maxim=$2

cd $parent

for f in \* ; do

if [ ! -d $f ] ; then continue; fi

lung=`du -c $f | awk '$2 == "total" { print $1 }'`

if [ $lung -lt $maxim ] ; then continue; fi

echo $f

mail -s "Avertisment" $f@scs.ubbcluj.ro <<MESAJ

Directorul este prea mare

MESAJ

done

### Anunţarea userilor cu prea multe procese

#!/bin/bash

# check-process-count.sh maxim

# maxim este numarul maxim de procese admis

maxim=$1

for user in `who | awk '{ print$1 }'` ; do

if [ `ps -u $user | wc -l` -ge $maxim ] ; then

echo $user

write $user <<MESAJ

Aveti prea multe procese

MESAJ

fi

done

### Distrugerea proceselor prea vechi

#!/bin/bash

# check-process-age.sh ore

# ore este numarul maxim de procese admis

# ps -eo pid,etime da pidul si durata de viata

for pid in `ps -eo pid,etime | awk 'length($2) == 8 && ore <= substr($2,1,2)+0 {print $1}' ore=$1` ; do

echo $pid

#kill -9 $pid

done

### Anunţul încărcării exagerate a procesorului şi / sau a memoriei

#!/bin/bash

# check-server-load.sh addr procMax memMax

# addr adresa unde se trimite avertismentul

# procMax incarcarea maxima (procente) a serverului dupa 5 minute

# memMax incarcarea maxima (procente) a memoriei

procMax=$2

memMax=$3

while true ; do

#sleep 300

proc=`uptime | awk '{s=$(NF-1); s = substr(s,1,index(s,".")-1); print s}'`

mem=`free | awk '$1=="Mem:" {print $2\*100/$3}'`

echo $proc

echo $mem

if [ $proc > $procMax -o $mem > $memMax ] ; then

mail -s "Avertisment" $1 <<MESAJ

Server / memorie prea incarcat(a)

MESAJ

fi

done

### Verificare perechi fisier - lungime

Sa se scrie un fisier de comenzi care primeste ca parametri perechi formate din nume de fisier si un numar. Pentru fiecare astfel de pereche se va verifica daca dimensiunea fisierului coincide cu numarul respectiv si se va afisa un mesaj corespunzator.

#!/bin/sh

# Sa se scrie un fisier de comenzi care primeste ca parametri perechi formate

# din nume de fisier si un numar. Pentru fiecare astfel de pereche se va

# verifica daca dimensiunea fisierului coincide cu numarul respectiv, si se va

# afisa un mesaj corespunzator.

while true; do

if [ -z $1 ] || [ -z $2 ]; then break; fi

fisier=$1

numar=$2

if [ ! -f $fisier ]; then continue; fi # nu e fisier

if [ `echo $numar | egrep -c "[0-9]+"` -eq 0 ]; then continue; fi # NaN

#lung=`ls -l $fisier | cut -d" " -f5`

lung=`ls -l $fisier | awk '{print $5;}'`

if [ `echo $lung | egrep -c "[0-9]+"` -eq 0 ]; then continue; fi # NaN

if [ $numar -eq $lung ]; then echo "$fisier are lungimea $numar"; fi

shift 2

done

### Numărul mediu de linii:

Sa se scrie un script shell care primeste ca parametru un nume de director si va determina numarul mediu de linii din toate fisierele text din acest director si din toate subdirectoarele acestuia.

#!/bin/sh

# Sa se scrie un script shell care primeste ca parametru un nume de director si

# va determina numarul mediu de linii din toate fisierele text din acest

# director si din toate subdirectoarele acestuia.

if [ $# -ne 1 ]; then echo "trebuie dat un director" >&2; exit 1; fi

if [ ! \( -d $1 \) ]; then echo "$1 nu exista sau nu este director" >&2; exit 2; fi

TotalLinii=0

TotalFisiere=0

find $1 -type f -print | sort | while read Fisier t; do

if [ `file $Fisier | grep -ci "ASCII text"` -eq 0 ] ; then continue ; fi

linii=`wc -l <$Fisier` # cu $Fisier, da la iesire si numele fisierului

TotalLinii=`expr $linii + $TotalLinii`

TotalFisiere=`expr $TotalFisiere + 1`

echo $TotalFisiere >/tmp/${LOGNAME}TotalFisiere

echo $TotalLinii >/tmp/${LOGNAME}TotalLinii

done

TotalFisiere=`cat /tmp/${LOGNAME}TotalFisiere`

TotalLinii=`cat /tmp/${LOGNAME}TotalLinii`

rm /tmp/${LOGNAME}TotalFisiere /tmp/${LOGNAME}TotalLinii

medie=`expr $TotalLinii / $TotalFisiere`

echo "Linii" $TotalLinii "Fisiere" $TotalFisiere "Medie" $medie

### Supravegherea continutului unui grup de directoare

Sa se scrie un script shell care monitorizeaza mai mute directoare. Monitorizarea se refera la aparitia unui fisier in aceste directoare. Numele directoarelor se dau la linia de comanda, iar numele fisierului urmarit se da de la tastatura. De asemenea, se da de la tastatura timpul, in secunde, intre doua cautari.

#!/bin/sh

# Sa se scrie un script shell care monitorizeaza mai mute directoare.

# Monitorizarea se refera la aparitia unui fisier in aceste directoare.

# Numele directoarelor se dau la linia de comanda, iar

# numele fisierului urmarit se da de la tastatura.

# De asemenea, se da de la tastatura timpul, in secunde, intre doua cautari.

echo -n "Introduceti numele fisierului: "; read nume

echo -n "Introduceti timpul (sec) intre cautari: "; read timp

while true; do

find $\* -type f -print | while read af; do

nf=`echo $af | awk -F/ '$NF>1{print $NF;}'`

if [ "$nf" = "$nume" ]; then echo $nume "in" $af; fi

done

sleep $timp

done

Remarcam utilizarea in find a variabilei **$\*** care reuneste toate argumentele liniei de comanda, privite ca o lista de nume (**$@** vede aceeasi lista ca un singur string).

### Selectare numai cuvinte din litere mici

Sa se scrie un script care primeste la linia de comanda doua nume de fisiere: intrare iesire. Scriptul preia intrare si pune in iesire liniile din intrare din care se retin numai cuvintele formate din litere mici. In iesire se vor ordona alfabetic liniile si se vor elimina dublurile.

Evident, pot fi mai multe soluţii. Noi vom da două soluţii, ambele folosind expresii regulare. O soluţie foloseşte awk, cealaltă sed.

#!/bin/sh

# Sa se scrie un script care primeste la linia de comanda doua nume

# de fisiere: intrare iesire

# Scriptul preia intrare si pune in iesire liniile din intrare

# din care se retin numai cuvintele formate din litere mici.

# iesire se vor ordona alfabetic liniile si se vor elimina dublurile.

# varianta awk sort

awk <$1 '{for (i=1; i<=NF; i++)\

if ($i !~ /^[a-z]+$/) $i="";\

linie = ""; for (i=1; i<=NF; i++) if($i != "") linie = linie $i " ";\

if (linie != "") print substr(linie,1,length(linie)-1);}' |\

sort -u >$2.awk

# varianta sed sort

sed <$1 -e 's/\t/ /g'| # InlocuiesteTAB cu un spatiu \

sed -e 's/$/ /g'| # Adauga spatiu la sfarsit \

sed -e 's/^[^ ]\*[^a-z ][^ ]\* / /g'| # Primul cuvant din linie \

sed -e 's/ [^ ]\*[^a-z ][^ ]\*$/ /g'| # Ultimul cuvant din linie \

sed -e 's/[^ ]\*[^a-z ][^ ]\* / /g' | # Cuvintele terminate de spatiu \

sed -e 's/ / /g'| # Reduce numarul de spatii. De ce trebuie repetata? \

sed -e 's/ / /g'| # Reduce numarul de spatii \

sed -e 's/ / /g'| # Reduce numarul de spatii \

sed -e 's/ / /g'| # Reduce numarul de spatii \

sed -e 's/ / /g'| # Reduce numarul de spatii \

sed -e 's/^ //g' | # Sterge spatiul de la inceput \

sed -e 's/ $//g'| # Sterge spatiul de la sfarsit \

sed -e '/^$/d' | # Sterge liniile goale \

sort -u >$2.sed

### Primul cuvant de cel putin 5 caractere

Din directorul curent, să se determine primul fişier text care conţine o linie al cărei prim cuvant are cel puţin 5 caractere. Exemplul este dat pentru a testa break 2 (ieşirea din două cilcuri interioare), care se pare că nu merge. Este suplinit prin break - uri succesive.

Selectarea primului cuvânt din linie se poate face în două moduri: folosind **cut** sau fara el.

#!/bin/sh

# Sa se caute in directorul curent primul fisier text care contine o linie

# in care primul cuvant este mai scurt de 5 caractere.

for x in \* ; do

if [ `file $x | grep -ci "ASCII text"` -eq 0 ] ; then continue ; fi

#Variabila cuv1 retine primul cuvant de pe o linie, delimitator spatiu

cat $x | while read cuv1 t ; do

#cat $x | cut -d" " -f1 | while read cuv1 ; do

#verificam daca linia nu e vida, respectiv lungimea primului cuvant

if [ ! -z $cuv1 ] && [ `expr length $cuv1` -ge 5 ] ; then

echo In $x s-a gasit $cuv1 cu lungimea `expr length $cuv1`

#break 2 # Se iese din doua cicluri

break # Se iese din ciclul while

fi

done

break # Se iese din ciclul for

done

### Lista fişierelor cu anumite drepturi:

Sa se construiasca un fisier de comenzi care primeste ca parametru un nume de director (sa-l numim D) si un numar intreg (sa-l numim N). Pentru fiecare fisier din directorul D sau din subdirectoarele acestuia, pentru care userul are drepturi de citire si de executie, sa se afiseze (maximum) primele N linii.

#!/bin/sh

# Sa se construiasca un fisier de comenzi care primeste ca

# parametru un nume de director (sa-l numim D) si un numar intreg (sa-l numim N).

# Pentru fiecare fisier din directorul D sau din subdirectoarele acestuia,

# pentru care userul are drepturi de citire si de executie,

# sa se afiseze (maximum) primele N linii.

# verificam daca exista 2 parametrii in linia de comanda;

if [ ! $# -eq 2 ]; then echo "usage: shell1.sh director numar"; exit 1;fi

# verificam daca primul parametru este nume de director

if [ ! -d $1 ];then echo "$1 nu este director\!"; exit 1;fi

D=$1 # Numai pentru a fi in ton cu enuntul. Putem folosi si $1

N=$2 # Numai pentru a fi in ton cu enuntul. Putem folosi si $2

# find $D -perm -u=rx -type f

# va afisa toate fisierele din directorul $D si din subdirectoare

# pentru care user-ul (u) are drept de citire (r) si de executie (x).

# O alternativa pentru comanda find ar fi test cu optiunile -f,-r, -x

for fis in `find $D -perm -u=rx -type f`; do

echo $fis # afisam numele fisierului

head -$N $fis # afisam primele $N linii din fisier

done

### Primele linii din fişiere cu cuvinte lungi:

Sa se creeze un fisier care contine numele tuturor fisierelor text, dintr-un director dat ca parametru si din subdirectoarele lui, care au cuvinte mai lungi de **n** caractere, unde **n** se citeste de la tastatura. Lista rezultata va fi ordonată alfabetic.

#!/bin/sh

# Sa se creeze un fisier care contine numele tuturor fisierelor text dintr-un

# director dat ca parametru si din subdirectoarele lui, care au cuvinte mai

# lungi de n caractere, unde n se citeste de la tastatura.

# Fisierul rezultat va fi ordonat alfabetic.

if [ $# -lt 1 ] ; then echo "introduceti cel putin un argument" ; exit 1; fi

if [ ! -d $1 ] ; then echo "$1 nu e director" ; exit 2; fi

rm /tmp/${LOGNAME}numefisiere /tmp/${LOGNAME}rezultat >/dev/null 2>&1

echo -n "Introduceti n: "

read n

for fis in `find $1 -type f -print`; do

if [ `file $fis | grep -c text` -ne 1 ] ; then continue; fi

# variabilele n si fis din interiorul lui awk sunt altele decat cele din sh

# La fel si $1 $2 . . .

awk -v n=$n -v fis=$fis '\

{ for (i=1; i<=NF; i++) if (length($i) > n) print fis; }'\

<$fis >>/tmp/${LOGNAME}numefisiere # De ce am folosit /tmp ??

done

sort -u </tmp/${LOGNAME}numefisiere >/tmp/${LOGNAME}rezultat

cat /tmp/${LOGNAME}rezultat

### Redenumirea fişierelor de un anumit tip si numararea aparitiilor unui cuvant

Sa se scrie un script shell care primeste 4 parametri: director, extensie1, extensie2, cuvant. Scriptul va redenumi toate fisierele cu extensia extensie1 (dupa .) din director si subdirectoarele acestuia, dandu-le extensia extensie2. Va numara de cate ori apare cuvant in fiecare fisier.

#!/bin/sh

# Sa se scrie un script shell care primeste 4 parametri:

# director, extensie1,  extensie2, cuvant

# Scriptul va redenumi toate fisierele cu extensia extensie1 (dupa .)

# din director si subdirectoarele acestuia, dandu-le extensia extensie2.

# Va numara de cate ori apare cuvant in fiecare fisier.

if [ $# -ne 4 ]; then echo "director, extensie1,  extensie2, cuvant" >&2; exit 1; fi

if [ ! \( -d $1 \) ]; then echo "$1 nu exista sau nu este director" >&2; exit 2; fi

Total=0

find $1 -type f -print | sort | while read Fisier; do

if [ `file $Fisier | grep -ci "ASCII text"` -eq 0 ]; then continue; fi

ext=`echo $Fisier | awk -F. '$NF>1{print $NF;}'`

if [ -z $ext ] || [ $ext != $2 ]; then continue; fi

nume=`echo $Fisier | awk -F. '$NF>1{print substr($0,1,length($0)-length($NF));}'`

apare=`grep -ci $4 $Fisier`

Total=`expr $Total + $apare`

echo $Total>/tmp/${LOGNAME}Total

echo "In" $Fisier $4 "apare de" $apare "ori"

echo $Fisier "se va redenumi in" $nume$3

#mv $Fisier $nume$3 # Aici re face redenumirea

done

echo $4 "apare in total de" `cat /tmp/${LOGNAME}Total` "ori"

rm /tmp/${LOGNAME}Total

Remarcam separarea partilor din numele absolut al unui fisier folosind awk.

### Aparitia unor cuvinte in linii din fisiere

Sa se scrie un script care primeste la linia de comanda triplete: fisier, cuvant numar

Pentru fiecare astfel de triplet, se vor afisa toate liniile din fisier care contin cuvant exact de numar ori.

#!/bin/sh

# Sa se scrie un script care primeste la linia de comanda triplete:

# fisier, cuvant numar

# Pentru fiecare astfel de triplet, se vor afisa toate liniile

# din fisier care contin cuvant exact de numar ori.

while true; do

if [ -z $1 ] || [ -z $2 ] || [ -z $3 ]; then break; fi

fisier=$1

cuvant=$2

numar=$3

if [ ! -f $fisier ]; then continue; fi # nu e fisier

if [ `file $fisier | grep -c "text"` -eq 0 ]; then continue; fi

if [ `echo $numar | egrep -c "[0-9]+"` -eq 0 ]; then continue; fi # NaN

awk '{k=0; for (i=1; i<=NF; i++) if ($i == cuv) k++;\

if (k==num) print FILENAME ":" $0;}' num=$numar cuv=$cuvant $fisier

shift 3

done

### Reunirea fişierelor text

Se cere un script **sh** care primeste la linia de comandă un nume de director. Se cere ca toate fişierele cu conţinut text din acest director şi din descendenţii lui să fie concatenate într-unul singur, într-o formă tipăribilă. Fişierul în care se reunesc să înceapă cu un cuprins, în ordine alfabetică, care reperează fişierele componente făcând repertorizarea la nivel de linie sursă. Iată cum apare o porţiune din fişierul reuniune.

0 linii pana la ./b: ASCII text, with CRLF, LF line terminators

6 linii pana la ./c: ASCII text

9 linii pana la ./cl: POSIX shell script, ASCII text executable

38 linii pana la ./f: ASCII text

42 linii pana la ./f5: POSIX shell script, ASCII text executable

68 linii pana la ./lr: POSIX shell script, ASCII text executable

98 linii pana la ./pall: POSIX shell script, ASCII text executable

135 linii pana la ./pall.sh: POSIX shell script, ASCII text executable

Total general: 172 linii in 8 fisiere.

========================================

|| 0 linii pana la ./b: ASCII text, with CRLF, LF line terminators

========================================[

Sa se scrie un fisier de comenzi care preia un fisier de intrare dat ca

parametru si creeaza din el un alt fisier (al carui nume este dat ca

parametru) in care pastreaza doar cuvintele care contin litere mici. Fisierul

se va ordona alfabetic. Daca in rezultat exista linii consecutive identice, se

va pastra doar una dintre ele.

]========================================

========================================

|| 6 linii pana la ./c: ASCII text

========================================[

Sa se scrie un script shell care monitorizeaza aparitia in toate directoarele

date ca si parametru in linia de comanda a unui nume de fisier citit de la

tastatura.

]========================================

========================================

|| 9 linii pana la ./cl: POSIX shell script, ASCII text executable

========================================[

- - - - -

========================================

|| 135 linii pana la ./pall.sh: POSIX shell script, ASCII text executable

========================================[

- - - - -

]========================================

Total general: 172 linii in 8 fisiere.

========================================

Sursa scriptului este:

#!/bin/sh

# Sa se reuneasca intr-un fisier text, intr-o forma tiparibila,

# toate fisierele text din directorul $1.

# Fisierul text va incepe cu un cuprins al fisierelor continute.

if [ $# -ne 1 ]; then echo "folosire: pall director" >&2; exit 1; fi

if [ ! \( -d $1 \) ]; then echo "$1 nu exista sau nu este director" >&2; exit 2; fi

rm /tmp/${LOGNAME}Afisari /tmp/${LOGNAME}DeListat>/dev/null 2>&1

rm /tmp/${LOGNAME}/Total >/dev/null 2>&1

TotalLinii=0

find $1 -type f -print | sort | while read FISIER; do

if [ `file $FISIER | grep -ci "ASCII text"` -eq 0 ] >/dev/null 2>&1

#if file $FISIER | egrep "exec|data|empty|reloc|cannot open" >/dev/null 2>&1

then continue

fi

NrLinii=`wc -l <"$FISIER"` # De ce este necesara delimitarea cu " ?

Linie=${TotalLinii}" linii pana la "`file $FISIER`

echo $Linie >/dev/tty

echo $Linie >> /tmp/${LOGNAME}DeListat

echo "\n\n========================================" >> /tmp/${LOGNAME}Afisari

echo "|| "$Linie >> /tmp/${LOGNAME}Afisari

echo "========================================[" >> /tmp/${LOGNAME}Afisari

cat $FISIER >> /tmp/${LOGNAME}Afisari

#pr -f $FISIER >> /tmp/${LOGNAME}Afisari

echo "]========================================" >> /tmp/${LOGNAME}Afisari

TotalLinii=`expr $TotalLinii + $NrLinii`

echo $TotalLinii >/tmp/${LOGNAME}Total # De ce este necesar ?

done

TotalLinii=`cat </tmp/${LOGNAME}Total`

NrFisiere=`wc -l </tmp/${LOGNAME}DeListat`

Linie="Total general: $TotalLinii linii in $NrFisiere fisiere."

echo $Linie >/dev/tty

echo $Linie >>/tmp/${LOGNAME}DeListat

echo $Linie >>/tmp/${LOGNAME}Afisari

echo "========================================" >> /tmp/${LOGNAME}Afisari

cat /tmp/${LOGNAME}Afisari >>/tmp/${LOGNAME}DeListat

rm /tmp/${LOGNAME}Afisari /tmp/${LOGNAME}Total

Pentru a nu perturba culegerea de fişiere, fişierele auxiliare necesare sunt memorate în directorul /tmp, prefixate cu numele userului (/tmp este directorul tuturor userilor). Mai întâi se verifică corectitudinea parametrului, apoi se sterg eventualele fişiere temporare vechi. Construcţia >/dev/null 2>&1 ascunde fişierele de ieşire şi de erori standard. Lista tuturor fişierelor din director este dată de find, iar în ciclul while se lipesc fişierele de tip text. Decizia că un fişier conţine text este obţinută prin file şi grep. Câteva variabile de mediu împreună cu efectul comenzii wc realizează calculele de numere de linii aferente. Derularea comenzii se vede pe tty, indiferent de ieşirea standard a scriptului. Fişierul /tmp/${LOGNAME}DeListat conţine reuniuneea fişierelor cu conţinut text din $1.

### Analizaţi textul sursă al unui script

Considerând că directorul **DIR** conține o ierarhie de subdirectoare și fișiere text răspundeți la următoarele întrebări despre scriptul Shell UNIX de mai jos:

1. Ce va conține fișierul 1.txt după rularea scriptului?
2. Ce va conține fișierul 2.txt după rularea scriptului?
3. Explicați în detaliu expresiile regulare de pe liniile 2 și 5.

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8 | for f in `find DIR -type f`; do  if grep -q "^[^0-9]" $f; then  echo $f >> 1.txt  fi  if ! grep -q "[a-z]$" $f; then  echo $f > 2.txt  fi  done |

Răspuns:

1. Căile (relative la DIR) ale tuturor fișierelor care conțin linii care nu încep cu cifră.
2. Calea (relativă la DIR) a ultimului fișier găsit de comanda find care nu conține linii terminându-se cu literă mică.

Linia 2: început de linie, interval de cifre negat. Linia 5: interval de litere mici, sfârșit de linie

## Probleme propuse

1. Răspundeţi la următoarele întrebări, considerând o rulare a scriptului shell de mai jos:

1. De câte ori se afişează "OK"? Justificaţi răspunsul.
2. Care e valoarea variabilei f?
3. Care e valoarea variabilei d?
4. Care e valoarea variabilei x?
5. Care e valoarea variabilei y?

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9 | f=`find . -type f`  d=`find . -type d`  for x in $f; do  for y in $d; do  if [ $x = $y ]; then  echo "OK"  fi  done  done |

2. Se d fişierul **abc.sh** conţinând scriptul Shell UNIX de mai jos. Răspundeţi la următoarele întrebări:

b) Ce se întâmplă dacă scriptul este rulat fără argumente?

c) Ce va tipări pe ecran rularea ./abc.sh f3 şi ce fişiere (nume şi conţinut) va crea, dacă f3 conţine “abc 74 2-8 aa 3a =c b2” şi rularea se face într-un director conţinând doar fişierele abc.sh 􀚍ş f3?

d) Daţi un exemplu de fişier f3 astfel încât rularea de la punctul precedent să creeze 4 fişiere noi astfel încât numele niciunuia să nu aibă prefixul f3.

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12 | n=0  for i in `cat $1`; do  c=`echo $i|cut -c1`  if echo $i|grep -q "^[0-9][0-9]\*$"; then  echo $i >> $1.nr  elif echo $c|grep -q "[A-Za-z]"; then  echo $i >> $c  else  n=`expr $n + 1`  fi  done  echo $n |

3. Sa se creeze doua fisiere, unul care va contine lista tuturor fisierelor dintr-un director si din subdirectoarele acestuia, iar al doilea lista tuturor subdirectoarelor. Fisierele vor fi ordonate dupa dimensiune, iar directoarele alfabetic.

4. Sa se calculeze suma fiecarei coloane de numere din oricate fisiere. Pentru fiecare fisier se vor afisa aceste sume, numarul maxim de coloane si numarul de linii.

5. Sa se scrie un fisier de comenzi care preia un fisier de intrare dat ca parametru si creeaza din el un alt fisier (al carui nume este dat ca parametru) in care pastreaza doar vocalele. Daca in rezultat exista linii consecutive identice, se va pastra doar una dintre ele.

6. Sa se scrie un fisier de comenzi care are ca parametri triplete formate dintr-un nume de fisier si doua cuvinte. Pentru fiecare astfel de triplet, se va inlocui in fisier ultima aparitie din fiecare linie a primului cuvant cu cel de-al doilea cuvant.

7. Sa se scrie un fisier de comenzi care preia un fisier de intrare dat ca parametru si creeaza din el un alt fisier (al carui nume este dat ca parametru) in care pastreaza doar cuvintele care contin litere mici. Fisierul se va ordona alfabetic, si se vor semnala liniile consecutive identice.

8. Sa se afiseze, pentru fiecare fisier din linia de comanda, cuvantul care apare de cele mai multe ori. Afisarea se va face in ordine descrescatoare a numarului de aparitii.

9. Sa se scrie un fisier de comenzi care va afisa toate fisierele dintr-un director si din subdirectoarele acestuia asupra carora au drepturi de scriere toate cele trei categorii de utilizatori. Aceste fisiere vor fi apoi redenumite, adaugandu-se sufixul .all.

10.Sa se afiseze, pentru fiecare fisier din linia de comanda, numarul liniei care apare de cele mai multe ori, afisarea facandu-se in ordinea decsrescatoare a numarului de aparitii.

# Procese Unix (în C): fork, exec, exit, wait system, signals

## Standardul POSIX de gestiune a erorilor în apeluri sistem: errno

Marea majoritate a funcţiilor C şi practic toate apelurile sistem Unix întorc un rezultat care "spune" dacă funcţia/apelul s-a derulat normal sau dacă a apărut o situaţie deosebită. In caz de eşec funcţia / apelul sistem întoarce fie un întreg nenul (valoarea 0 este rezervată pentru succes), fie un pointer NULL etc. Metodologic, **se recomandă** **SA SE APELEZE:**

**if (functie( - - - ) == SUCCES) { tratare normala }**

**else { tratare situatie de derulare anormala }**

**NU (aşa cum din comoditate apelează mulţi programatori) :**

**functie( - - -); tratare normala**

Evident forma a doua este mai scurtă, dar nu sunt tratate situaţiile de excepţie.

Pentru o abordare unitară a acestor tratamente, standardul POSIX oferă prin **#include <errno.h>** o variabilă întreagă **errno**, (care nu trebuie declarată) a carei valoare este setata de sistem *nunai în caz de derulare anormală a apelului***! La o situaţie de derulare anormală sistemul fixează o valoare nenulă ce indică cauza erorii**. Pentru detalii vezi man errno, precum şi lista completă a cazurilor de erori, aflată de exemplu la **http://www.virtsync.com/c-error-codes-include-errno**

La apelul cu succes al unei funcţii sistem *errno nu se seteaza la 0!* Pentru a se vedea şi în clar eroarea depistată se pot folosi funcţiile strerror şi perror dau detalii pentru fiecare valoare a lui errno:

**#include <errno.h>**

**- - -**

**if (functie( - - - ) == SUCCES) { tratare normala }**

**else { perror("Eroarea depistata este:");**

**// sau printf("Eroarea depistata este:%s", strerror(errno); }**

## Principalele apeluri sistem Unix care operează cu procese

Tabelul următor prezintă sintaxele principalelor apeluri sistem Unix care operează cu procese:

|  |
| --- |
| Funcţii specifice proceselor |
| fork()  exit(n)  wait(p)  exec\*(c, lc)  system(c) |

Tipurile de exec:



Prototipurile lor sunt descrise, de regula, in <unistd.h> Parametrii sunt:

* n este intreg – codul de retur cu care se termină procesul;
* p este un pointer la un întreg unde fiul întoarce codul de retur (extras cu funcţia WEXITSTATUS);
* c este o comandă Unix;
* lc este linia de comandă (comanda c urmată de argumentele liniei de comandă);
* f este un tablou de doi intregi – descriptori de citire / scriere din / in pipe;
* nume este numele (de pe disc) al fişierului FIFO, iar drepturi sunt drepturile de acces la acesta;
* fo si fn descriptori de fisiere: fo deschis in program cu open, fn poziţia în care e duplicat fo.

In caz de eşec, functiile întorc -1 (NULL la popen) si poziţionează errno se depisteaza ce eroare a aparut.

Funcţiile system si popen au comanda completă (un string), interpretabilă de shell: aceste funcţii lansează mai întâi un shell, apoi în acesta lansează comanda c. Această lansare se face simplu folosind un apel sistem execl: **execl("/bin/sh", "sh", "-c", c, NULL);**

Comanda c din apelurile exec\* ***NU permite specificări folosite uzual în liniile shell***. Astfel, în c NU trebuie să apară specificări generice de fişiere, redirectări de intrări / ieşiri standard, variabile shell, captări de ieşiri prin construcţii ` - - comanda - - ` etc. Dacă totuşi se doreşte acest lucru, trebuie să se lanseze, ca mai sus, interpretorul sh cu opţiunea -c şi apoi să se specifice comanda.

## Exemple de lucrul cu procese

### Utilizări simple fork exit, wait

Vom prezenta şi discuta două exemple de programe care utilizează apelurile sistem fork, exit, wait. Să considerăm **programul f1.c** căruia i-am numerotat liniile sursă:

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19 | #include <stdio.h>  #include <stdlib.h>  #include <unistd.h>  #include <sys/wait.h>  int main() {  int p, i;  p=fork();  if (p == -1) {perror("fork imposibil!"); exit(1);}  if (p == 0) {  for (i = 0; i < 10; i++)  printf("Fiu: i=%d pid=%d, ppid=%d\n", i, getpid(), getppid());  exit(0);  } else {  for (i = 0; i < 10; i++)  printf("Parinte: i=%d pid=%d ppid=%d\n", i, getpid(), getppid());  wait(0);  }  printf("Terminat; pid=%d ppid=%d\n", getpid(), getppid());  } |

Vom analiza comportamentul acestui program în diverse situaţii, făcând o serie de modificări în această sursă.

**Rularea în forma iniţială**: Sunt afişate 21 linii: 10 ale fiului de la linia 11 cu pidul lui şi al părintelui. 11 ale fiului, 10 de la linia 15 şi ultima de la linia 18. Părintele părintelui este pidul shell. Este posibil ca ordinea primelor 20 de linii să apară amesteecate, linii ale fiului şi liniile ale părintelui. Dacă la linia 10 şi la linia 14 se înlocuieşte 10 cu 1000, se vor afişa 2001 linii iar amestecarea între liniile fiului şi ale părintelui va fi mai evidentă.

**Comentarea liniei 12**: Procesul fiu se termină la linia 18, ca şi părintele. Se vor tipări 22 linii, linia 18 se va tipări de două ori: odată de părinte şi odată de fiu.

**Comentarea liniei 16**: Părintele nu mai aşteaptă terminarea fiului şi acesta din urmă rămâne în starea zombie. Se tipăresc cele 21 de linii ca în primul caz. O observaţie intereesantă: dacă ieşirea programului se redirectează într-un fişier pe disc, apar cele 21 linii. In schimb, dacă ieşirea se face direct pe terminal, apar doar liniile fiului. De ce oare? Rămâne un TO DO pentru studenţi.

**Comentarea liniilor 12 şi 16**: Se tipăresc 22 linii, cu aceeaşi observaţie de mai sus, de la comentarea liniei 16. Aici recomandăm modificări ale numărului liniilor tipărite de fiu (linia 10) şi a celor tipărite de părinte (linia 14). Se vor vedea efecte interesante.

Să considerăm **programul f2.c**:

main() { fork(); if (fork()) {fork();} printf("Salut\n");}

Care este efectul execuţiei acestui program? (Acoladele nu sunt necesare, dar le-am pus pentru a evidenţia mai bine corpul lui if). Să facem o primă analiză:

* Primul fork naşte un proces fiu. Ambele procese au de executat secvenţa: if (fork()) fork(); printf("Salut\n");
* Condiţia fork din if mai naşte câte un proces fiu cărora le rămâne de făcut doar printf("Salut\n"); In acelaşi timp, cele două procese care evaluează if mai au de făcut {fork();} printf("Salut\n"); Până aici avem patru procese.
* Fiecare fork dintre acolade mai naşte câte un proces fiu căruia îi mai rămâne de făcut printf("Salut\n"); Avem încă două procese în plus.
* In concluzie, avem şase (6) procese care au de executat printf("Salut\n"); In consecinţă, se va tipări de 6 ori Salut.

Merită să studiem mai atent acest exemplu. Principala carenţă a lui este aceea că nici un părinte care naşte un fiu nu aşteaptă terminarea lui prin wait. Consecinţa, vor rămâne câteva procese în starea zombie.

Pentru a aprofunda analiza, să rescriem puţin programul f2.c:

#include <stdio.h>

#include <unistd.h>

#include <sys/wait.h>

int main() {

printf("START: pid=%d ppid=%d\n", getpid(), getppid());

int i=-2, j=-2, k=-2;

i=fork();

if (j=fork())

{k=fork();}

printf("Salut pid=%d ppid=%d i=%d j=%d k=%d\n",getpid(),getppid(),i,j,k);

}

In fapt, am reţinut în variabilele i, j, k valorile PID-urilor create pe parcursul execuţiei. Rezultatul execuţiei este:

START: pid=3998 ppid=3810

Salut pid=3998 ppid=3810 i=3999 j=4000 k=4001

florin@ubuntu:~/c$ Salut pid=4001 ppid=1700 i=3999 j=4000 k=0

Salut pid=4000 ppid=1700 i=3999 j=0 k=-2

Salut pid=3999 ppid=1700 i=0 j=4002 k=4003

Salut pid=4003 ppid=1700 i=0 j=4002 k=0

Salut pid=4002 ppid=1700 i=0 j=0 k=-2

Să analizăm ordinea în care se execută aceste instrucţiuni:

* 3810 este PID-ul shell care afişează prompterul, iar 3998 este PID-ul programului iniţial.
* Procesul 3998 crează fiul i cu PID-ul 3999, fiul j cu PID-ul 4000 şi fiul k cu PID-ul 4001. Apoi îşi face tipărirea şi se termină - se vede tipărirea prompterului.
* Cele trei procese 3999, 4000 şi 4001 rămân active dar sunt în starea zombie (PPID-ul lor este 1700).
* Procesul 4001 preia controlul procesorului, valorile i şi j sunt moştenite de la 3998, iar k = 0 fiind vorba de fork în fiu, face tipărirea şi se termină.
* Procesul 4000 preia controlul procesorului, valorile i şi k sunt moştenite de la 3998 - i creat, k încă necreat, iar j = 0 fiind vorba de fork în fiu, face tipărirea şi se termină.
* Procesul 3999 preia controlul procesorului, i = 0 fiind vorba de fork în fiu, crează fiul j cu PID-ul 4002 şi fiul k cu PID-ul 4003. Apoi îşi face tipărirea şi se termină.
* Procesul 4003 preia controlul procesorului, valorile i şi j sunt moştenite de la 3999, iar k = 0 fiind vorba de fork în fiu. Apoi îşi face tipărirea şi se termină.
* Procesul 4002 preia controlul procesorului, valorile i şi k sunt moştenite de la 3999, iar j = 0 fiind vorba de fork în fiu. Apoi îşi face tipărirea şi se termină.

Tabelul următor prezintă cele 6 procese: ce valori moştenesc de la părinte, ce cod mai au de executat şi ce valori finale au (ce tipăresc).

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| PID 3998  PPID 3810  i=-2 j=-2  k=-2  i=fork();  if (j=fork())  {k=fork();}  printf - - -  i=3999 j=4000 k=4001 |  |  |  |  |  |
|  | PID 4001  PPID 3998  i=3999 j=4000 k=0  printf - - -  i=3999 j=4000 k=0 | PID 4000  PPID 3998  i=3999  j=0 k=-2  printf - - -  i=3999 j=0 k=-2 | PID 3999  PPID 3998  i=0 j=-2 k=-2  if (j=fork())  {k=fork();}  printf - - -  i=0 j=4002 k=4003 |  |  |
|  |  |  |  | PID 4003  PPID 3999  i=0  j=4002 k=0  printf - - -  i=0 j=4002 k=0 | PID 4002  PPID 3999  i=0 j=0 k=-2  printf - - -  i=0 j=0 k=-2 |

In acest tabel, valoarea PPID este cea reală a părintelui creator, deşi la momentul terminării fiului părintele nu mai există, aşa că procesul intră în starea zombie.

Pentru a evita starea acumularea de procese în starea zombie, se poate folosi, spre exemplu, secvenţa:

- - -

# include <signal.h>

int main() {

signal(SIGCHLD, SIG\_IGN);

- - -

In acest fel se cere ignorarea trimiterii de către fiu a semnalului SIGCHLD, pe care părintele ar trebui să îl primească (să fie în viaţă), să îl trateze cu un wait. Prin această ignorare, procesul fiu fiu este sters din sistem imediat după terminarea lui.

### Utilizări simple execl, execlp. execv, system

Urmatoarele două programe, desi diferite, au acelasi efect. Toate trei folosesc o comanda de tip exec, spre a lansa din ea comanda shell:

ls -l

**Programul 1:**

#include <stdio.h>

#include <unistd.h>

int main() {

char\* argv[3];

argv[0] = "/bin/ls";

argv[1] = "-l";

argv[2] = NULL;

execv("/bin/ls", argv);

}

Aici se pregateste linia de comandă în vectorul argv spre a o lansa cu execv.

**Programul 2:**

#include <stdio.h>

#include <unistd.h>

#include <stdlib.h> // trebuie pentru system

int main() {

//execl("/bin/ls", "/bin/ls", "-l", NULL);

// execlp("ls", "ls", "-l", NULL);

// execl("/bin/ls","/bin/ls","-l","p1.c","execl.c", "fork1.c", "xx", NULL);

// execl("/bin/ls","/bin/ls","-l","\*.c", NULL);

system("ls -l \*.c");

}

Aici se executa, pe rand, numai una dintre cele 5 linii, comentând pe celelalte 4. Ce se va intampla?

* Primul execl lanseaza ls prin cale absolută şi are acelaşi efect ca şi programul 1.
* Al doilea lansează ls prin directoarele din PATH, efectul este acelaşi.
* Al treilea cere ls pentru o listă de fişiere. Pentru cele care nu există, se dă mesajul: /bin/ls: cannot access 'xx': No such file or directory (in loc de xx apar numele fişierelor inexistente);
* Al patrulea exec va da mesajul: /bin/ls: cannot access \*.c: No such file or directory **Nu este interpretat asa cum ne-am astepta!** De ce? Din cauza faptului ca specificarea \*.c reprezinta o specificare generica de fisier, dar numai shell "stie" acest lucru si el (shell) inlocuieste aceasta specificare, in cadrul uneia dintre etapele de tratare a liniei de comanda. La fel stau lucrurile cu evaluarea variabilelor de mediu, ${---}, inlocuirea dintre apostroafele inverse ` --- `, redirectarea I/O standard etc.
* Apelul system are efectul aşteptat, făcând rezumatul tuturor fişierelor de tip c din directorul curent.

Să ne oprim puţin asupra funcţiei system. Ea crează prin fork un proces fiu, în care lansează comanda execl("/bin/sh", "sh", "-c", c, NULL) aşa cum am arătat mai sus şi întoarce codul de retur cu care s-a terminat execuţia comenzii. Doritorii pot să vadă sursa system.c, care este o funcţie simplă, de maximum 100 linii în care se includ comentariile, tratările cu errno ale posibilelor erori şi manevrarea unor semnale specifice. Sursa poate fi găsită la: http://man7.org/tlpi/code/online/dist/procexec/system.c

### Un program care compileaza şi rulează alt program

Exemplul care urmeaza are acelasi efect ca si scriptul sh:

#!/bin/sh

if gcc -o ceva $1

then ./ceva $\*

else echo "Erori de compilare"

fi

Noi nu il vom implementa în sh, ci vom folosi programul compilerun.c.

// Similar cu scriptul shell:

// #!/bin/sh

// if gcc -o ceva $1; then ./ceva $\*

// else echo "Erori de compilare"

// fi

#include<stdio.h>

#include<unistd.h>

#include <stdlib.h>

#include<string.h>

#include <sys/wait.h>

int main(int argc, char\* argv[]) {

char comp[200];

char\* run[100];

int i;

strcpy(comp, "gcc -o ceva ");

strcat(comp, argv[1]); // Fabricat comanda de compilare

if (WEXITSTATUS(system(comp)) == 0) {

run[0] = "./ceva";

for (i = 1; argv[i]; i++) run[i] = argv[i];

run[i] = NULL; // Fabricat comanda pentru execv

execv("./ceva", run);

}

printf("Erori de compilare\n");

}

Compilarea lui se face

gcc -o compilerun compilerun.c

Executia se face, de exemplu, prin

./compilerun argvenvp.c a b c

Cefect, daca compilarea sursei argument (argvenvp.c) este corecta, atunci compilatorul gcc creeaza fisierul ceva si intoarce cod de retur 0, dupa ceva este lansat prin execv. Daca esueaza compilarea, se va tipari doar mesajul "Erori de compilare".

Am ales ca şi exemplu de program argvenvp.c:

#include <stdio.h>

#include <string.h>

int main(int argc, char \*argv[], char \*envp[]) {

int i;

printf("Argumentele:\n");

for (i = 1; argv[i]; i++) printf("%s\n", argv[i]);

printf("Cateva variabile de mediu:\n");

for (i = 0; envp[i]; i++)

if (strncmp("HOME", envp[i], 4)==0 || strncmp("LOGNAME", envp[i], 7)==0)

printf("%s\n", envp[i]);

}

Secvenţa de execuţie este:

florin@ubuntu:~/c$ gcc -o compilerun compilerun.c

florin@ubuntu:~/c$ ./compilerun argvenvp.c a b c

Argumentele:

argvenvp.c

a

b

c

Cateva variabile de mediu:

HOME=/home/florin

LOGNAME=florin

florin@ubuntu:~/c$

### Capitalizarea cuvintelor dintr-o listă de fişiere text

Se cere un program care primeşte la linia de comandă o listă de fişiere text. Se cere ca toate aceste fişiere să fie transformate în altele, cu acelaşi conţinut, dar în care fiecare cuvânt să înceapă cu literă mare. Se vor lansa procese paralele pentru prelucrarea simultană a tuturor fişierelor.

Pentru aceasta, vom crea mai întâi un program cu numele cap din sursa cap.c. Acesta primeşte la linia de comandă numele a două fişiere text, primul de intrare, al doilea de ieşire cu cuvintele capitalizate:

#include <stdio.h>

#include <string.h>

#define MAXLINIE 100

main(int argc, char\* argv[]) {

printf("Fiu: %d ...> %s %s\n", getpid(), argv[1], argv[2]);

FILE \*fi, \*fo;

char linie[MAXLINIE], \*p;

fi = fopen(argv[1], "r");

fo = fopen(argv[2], "w");

for ( ; ; ) {

p = fgets(linie, MAXLINIE, fi);

linie[MAXLINIE-1] = '\0';

if (p == NULL) break;

if (strlen(linie) == 0) continue;

linie[0] = toupper(linie[0]); // Pentru cuvantul care incepe in coloana 0

for (p = linie; ; ) {

p = strstr(p, " ");

if (p == NULL) break;

p++;

if (\*p == '\n') break;

\*p = toupper(\*p); // Caracterul de dupa spatiu este facut litera mare

}

fprintf(fo, "%s", linie);

}

fclose(fo);

fclose(fi);

}

Al doilea program, numit master.c va crea câte un proces pentru fiecare nume de fişier primit la linia de comandă şi în acel proces va lansa cap fi fi.CAPIT

#include <stdio.h>

#include <string.h>

#include <unistd.h>

main(int argc, char\* argv[]) {

int i, pid;

char argvFiu[200];

for (i=1; argv[i]; i++) {

pid = fork();

if (pid == 0) {

strcpy(argvFiu, argv[i]);

strcat(argvFiu, ".CAPIT");

execl("./cap", "./cap", argv[i], argvFiu, NULL);

} else

printf("Parinte, lansat fiul: %d ...> %s %s \n", pid, argv[i], argvFiu);

}

for (i=1; argv[i]; i++) wait(NULL);

printf("Lansat simultan %d procese de capitalizare\n", argc - 1);

}

Compilari:

>gcc -o cap cap.c

>gcc -o master master.c

Lansare master f1 f2 ... fi ... fn

### Câte perechi de argumente au suma un număr par?

La linia de comandă se dau n perechi de argumente despre care se presupune ca sunt numere întregi si positive. Se cere numărul de perechi care au suma un număr par, numărul de perechi ce au suma număr impar si numărul de perechi în care cel putin unul dintre argumente nu este număr strict pozitiv.

Rezolvarea: In procesul părinte se va crea câte un process fiu pentru fiecare pereche. Oricare dintre fii întoarce codul de retur:

* 0 daca perechea are suma pară,
* 1 daca suma este impară,
* 2 daca unul dintre argumente este nul sau nenumeric.

Parintele aşteaptă terminarea fiilor şi din codurile de retur întoarse de aceştia va afisa rezultatul cerut.

Vom da doua solutii:

1. Solutia 1 cu textul complet intr-un singur fisier sursa
2. Solutia 2 cu doua texte sursa si unul să îl apeleze pe celalalt prin exec.

**Solutia 1:**

#include <stdio.h>

#include <stdlib.h>

#include <string.h>

#include <sys/wait.h>

main(int argc, char\* argv[]) {

int pare = 0, impare = 0, nenum = 0, i, n1, n2;

for (i = 1; i < argc-1; i += 2) {

if (fork() == 0) {

n1 = atoi(argv[i]); // atoi intoarce 0

n2 = atoi(argv[i+1]); // si la nenumeric

if (n1 == 0 || n2 == 0) exit(2);

exit ((n1 + n2) % 2);

}

}

for (i = 1; i < argc-1; i += 2) {

wait(&n1);

switch (WEXITSTATUS(n1)) {

case 0: pare++; break;

case 1: impare++; break;

default: nenum++;

}

}

printf("Pare %d, impare %d, nenumerice %d\n",pare, impare, nenum);

}

**Solutia 2:**

Se creaza programul par.c care primeşte la linia de comandă o pereche de argumente. Din această sursă se va consttiui prin gcc –o par par.c executabilul par:

main(int argc, char \*argv[]) {

int n1, n2;

n1 = atoi(argv[1]); // atoi intoarce 0

n2 = atoi(argv[2]); // si la nenumeric

if (n1 == 0 || n2 == 0) exit(2);

exit ((n1 + n2) % 2);

}

Se creaza programul master.c care primeşte la linia de comandă n perechi de argumente. El va crea n procese fii şi în fiecare va lansa prin exec programul par. Din aceasta sursa se va consttiui prin gcc –o master master.c executabilul master:

main(int argc, char\* argv[]) {

int pare = 0, impare = 0, nenum = 0, i, n1, n2;

for (i = 1; i < argc-1; i += 2) {

if (fork() == 0) {

execl(“./par”, “./par”, argv[i], argv[i+1], NULL);

}

}

for (i = 1; i < argc-1; i += 2) {

wait(&n1);

switch (WEXITSTATUS(n1)) {

case 0: pare++; break;

case 1: impare++; break;

default: nenum++;

}

}

printf("Pare %d, impare %d, nenumerice %d\n",pare, impare, nenum);

}

Intrebare la ambele solutii: Ce se întamplă daca wait si switch nu sunt plasate în cicluri for succesive ci în acelaşi for care crează procesele fii?

## Semnale Unix; exemple de utilizare

### Evitarea proceselor zombie

Versiunea Unix System V Release 4 şi Linux: este suficient ca în partea de iniţializare, înainte de crearea proceselor ce pot deveni zombie:

# include <signal.h>

- - -

signal(SIGCHLD, SIG\_IGN);

Versiunea Unix BSD, efectul unui apel sistem signal este valabil o singură dată:

#include <signal.h>

- - -

void waiter(){ // Functie de manipulare a apelurilor signal

wait(0); // Sterge fiul recent terminat

signal(SIGCHLD, waiter); // Reinstalare handler signal

}// waiter

- - -

signal(SIGCHLD, waiter); // Plasat în partea de iniţializare

### Schema client / server: adormire si deşteptare

Programul *server* este pus în adormire şi va fi trezit de fiecare *client* ca să-i satisfacă o cerere, după care intră din nou în adormire. prezentăm două variante de server: *iterativ* şi *concurent*.

#include <signal.h>

#include <stdio.h>

#include <unistd.h>

main() {

for ( ; ; ) {

printf("%d doarme . . .\n", getpid());

kill(getpid(), SIGSTOP); // Doarme, asteptand cereri

|  |  |
| --- | --- |
| **Server iterativ** | **Server concurent** |
| printf("Servesc cererea . . .");  serveşteCerereClient( . . . ); | if (fork() == 0) {  printf("Servesc cererea. . .");  serveşteCerereClient( . . . );  } |

}

}

Trezirea se poate face fie printr-o comandă:

$ kill -SIGCONT pidserver

fie printr-un program client de forma:

#include <signal.h>

#include <stdio.h>

#include <stdlib.h>

main(int argc, char \*\* argv) {

int pidserver = atoi(argv[1]);

// - - - Clientul afla pid-ul serverului si prepara o cerere

kill(pidserver, SIGCONT); //Desteptarea, server!

// - - - Clientul trateaza raspunsul

}

### Aflarea unor informaţii de stare

Pentru un program care dureaza mult, se vrea din când în când să se afle stadiul calculelor:

#include <stdio.h>

#include <signal.h>

// informatii globale de stare

int numar;

void tipareste\_stare(int semnal) {

// Tipareste informatiile de stare solicitate

printf("Numar= %d\n", numar);

}//handlerul de semnal

main() {

signal(SIGUSR1,tipareste\_stare);

//- - -

for(numar=0; ; numar++) {

// - - -

}//for

}//main

Pentru tipărirea stadiului curent, cunoscănd (ps) pidul programului, se dă comanda:

$ kill -SIGUSR1 pid

### Tastarea unei linii în timp limitat

SE cere ca tastarea unei linii de la terminal să se facă în timp limitat (în cazul nostru 5 secunde), altfel se anulează citirea şi programul se aduce în starea dinaintea lansării citirii:

#include <stdio.h>

#include <setjmp.h>

#include <sys/signal.h>

#include <unistd.h>

#include <string.h>

jmp\_buf tampon;

void handler\_timeout (int semnal) {

longjmp (tampon, 1);

}//handler\_timeout

int t\_gets (char \*s, int t) {

char \*ret;

signal (SIGALRM, handler\_timeout);

if (setjmp (tampon) != 0)

return -2;

alarm (t);

ret = fgets (s, 100, stdin);

alarm (0);

if (ret == NULL)

return -1;

return strlen (s);

}//t\_gets

main () {

char s[100];

int v;

while (1) {

printf ("Introduceti un string: ");

v = t\_gets (s, 5);

switch (v) {

case -1:

printf("\nSfarsit de fisier\n");

return(1);

case -2:

printf ("timeout!\n");

break;

default:

printf("Sirul dat: %s a.Are %d caractere\n", s, v-1);

}//switch

}//while

}//t\_gets.c

### Blocarea tastaturii

Se blochează tastatura până când se tastează parola cu care s-a făcut login:

#define \_XOPEN\_SOURCE

#include <stdio.h>

//#include <sys/types.h>

#include <unistd.h>

#include<string.h>

#include <pwd.h>

#include <shadow.h>

#include <signal.h>

main() {

char \*cpass, pass[15];

struct passwd \*pwd;

struct spwd \*shd;

signal(SIGHUP, SIG\_IGN);

signal(SIGINT, SIG\_IGN);

signal(SIGQUIT, SIG\_IGN);

setpwent();

pwd = getpwuid(getuid());

endpwent();

setspent();

shd = getspnam(pwd->pw\_name);

endspent();

setuid(getuid()); // Redevin userul real

for ( ;; ) {

strcpy(pass, getpass("...tty LOCKED!!"));

cpass = crypt(pass, shd->sp\_pwdp);

if (!strcmp(cpass, shd->sp\_pwdp))

break;

}//for

}//main

//lockTTY.c

// Compilare: gcc -lcrypt -o lockTTY lockTTY.c

// User root: chown root.root lockTTY

// User root: chmod u+s lockTTY

// Executie: ./lockTTY

## Probleme propuse

1. Programul apelat compara doua sau mai multe numere primite ca argumente si returneaza cod 0 daca toate sunt egale, 1 altfel. Programul apelant citeste niste numere si spune daca sunt egale.

2. Programul apelat primeste ca argumente un nume de fisier si une sir de caractere si scrie in fisier sirul oglindit. Programul apelat citeste niste siruri de caractere si concateneaza oglindirile lor.

3. Programul apelat primeste ca argumente niste numere si returneaza cod 0 daca produsul lor este pozitiv, 1 daca e negativ si 2 daca e nul. Programul apelant citeste un sir de numere si afiseaza daca produsul lor este pozitiv, negativ sau zero.

4. Programul apelat primeste ca argumente doua numere naturale si un nume de fisier si scrie in fisier cel mai mic multiplu comun al numerelor. Programul apelant citeste un sir de numere naturale si afisieaza cel mai mic multiplu comun al lor.

5. Programul apelat primeste ca argumente doua numere naturale si un nume de fisier si scrie in fisier cel mai mare divizor comun al numerelor. Programul apelant citeste un sir de numere naturale si afisieaza cel mai mare divizor comun al lor.

6. Programul apelat primeste ca argumente doua numere si un nume de fisier si scrie in fisier produsul numerelor. Programul apelant citeste un sir de numere si afiseaza produsul lor.

7. Programul apelat primeste ca argumente trei nume de fisiere, primele doua continand cate un sir crescator de numere intregi, si scrie in al treilea fisier rezultatul interclasarii sirurilor din primele doua fisiere. Programul apelant citeste un sir de numere intregi, le sorteaza si scrie rezultatul sortarii.

# Comunicarea între procese Unix: pipe, FIFO, popen, dup2

## Principalele apeluri sistem de comunicare între procese

Tabelul următor prezintă sintaxele principalelor apeluri sistem Unix pentru comunicare între procese:

|  |
| --- |
| Funcţii specifice comunicării între procese |
| pipe(f)  mkfifo(nume, drepturi)  FILE \*popen(c, “r|w”)  pclose(FILE \*)  dup2(fo,fn) |

Prototipurile lor sunt descrise, de regula, in <unistd.h> Parametrii sunt:

* c este o comandă Unix;
* f este un tablou de doi intregi – descriptori de citire / scriere din / in pipe;
* nume este numele (de pe disc) al fişierului FIFO, iar drepturi sunt drepturile de acces la acesta;
* fo si fn descriptori de fisiere: fo deschis in program cu open, fn poziţia în care e duplicat fo.

In caz de eşec, functiile întorc -1 (NULL la popen) si poziţionează errno se depisteaza ce eroare a aparut.

Funcţia popen are comanda completă (un string), interpretabilă de shell şi apoi executată

## Analizaţi textul sursă

Considerând că toate instrucțiunile din fragmentul de cod de mai jos se execută cu succes, răspundeți la următoarele întrebări:

1. Ce va tipări rularea codului așa cum este?
2. Câte procese se creează, incluzând procesul inițial, dacă lipsește linia 8? Specificați relația părinte fiu dintre aceste procese.
3. Câte procese se creează, incluzând procesul inițial, dacă mutăm instrucțiunea de pe linia 8 pe linia 11 (pornind de la codul dat)? Specificați relația părinte fiu dintre aceste procese.
4. Ce va tipări rularea codului, dacă liniile 16 și 17 se mută în interiorul ramurii else, începând cu linia 11 a codului inițial? Justificați răspunsul.

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21 | int main() {  int pfd[2], i, n;  pipe(pfd);  for(i=0; i<3; i++) {  if(fork() == 0) {  write(pfd[1], &i, sizeof(int));  close(pfd[0]); close(pfd[1]);  exit(0);  }  else {  // a se vedea punctele c) si d)  }  }  for(i=0; i<3; i++) {  wait(0);  read(pfd[0], &n, sizeof(int));  printf("%d\n", n);  }  close(pfd[0]); close(pfd[1]);  return 0;  } |

Răspuns:

1. 0, 1, 2 pe linii separate în orice ordine.
2. 8 procese, arbore cu 8 procese.
3. 4 procese, arbore cu 4 procese.
4. 0, 1, 2 pe linii separate întodeauna în această ordine.

## Utilizări simple pipe şi FIFO

Pentru a ilustra modul de lucru cu pipe şi cu FIFO, vom pleca de la exemplul cunoscut de **adunare paralelă rea a patru numere**, exemplu care reclamă necesitatea comunicării între procese. Sursa programului **add4Rau.c** este:

#include <stdio.h>

#include <unistd.h>

#include <stdlib.h>

#include <sys/wait.h>

int main () {

int a[] = {1,2,3,4};

if (fork()==0) { // Procesul fiu

a[0]+=a[1];

exit(0);

}

a[2]+=a[3]; // Procesul parinte

wait(NULL);

a[0]+=a[2];

printf("Suma este %d\n", a[0]);

}

Se ştie, suma tipărită va fi 8, nu 10, deoarece informaţia din procesul fiu nu ajunge în părinte. Vom da trei soluţii corecte pentru această problemă, toate vor tipări "**Suma este 10**".

**Soluţia 1: comunicarea prin pipe** este dată în programul **add4p.c**:

#include <stdio.h>

#include <unistd.h>

#include <stdlib.h>

#include <sys/wait.h>

int main () {

int a[] = {1,2,3,4}, f[2];

pipe(f);

if (fork()==0) { // Procesul fiu

close(f[0]); // Nu trebuie

a[0]+=a[1];

write(f[1], &a[0], sizeof(int)); // Scrie suma partiala

close(f[1]);

exit(0);

}

close(f[1]); // Nu trebuie in procesul parinte

a[2]+=a[3]; // Procesul parinte

read(f[0], &a[0], sizeof(int));

close(f[0]);

wait(NULL);

a[0]+=a[2];

printf("Suma este %d\n", a[0]);

}

**Soluţia 2: comunicarea prin FIFO cu procesele în aceeaşi sursă**. FIFO permite comunicarea între două procese care nu sunt, neapărat, înrudite. Din această cauză, se obişnuieşte ca fişierul FIFO să se creeze în directorul /tmp. Această creare se poate face, de exemplu, prin comanda:

$ mkfifo /tmp/fifo1

Evident, crearea se face înainte de a lansa procesele care o utilizează. Natural, atunci când nu mai avem nevoie de acest FIFO, el se şterge cu comanda:

$ rm /tmp/fifo1

Sursa pentru această soluţie este add4f.c:

#include <stdio.h>

#include <unistd.h>

#include <stdlib.h>

#include <fcntl.h>

#include <sys/wait.h>

#include <sys/types.h>

int main () {

int a[] = {1,2,3,4}, f;

if (fork()==0) { // Procesul fiu

f = open("/tmp/fifo1", O\_WRONLY);

a[0]+=a[1];

write(f, &a[0], sizeof(int)); // Scrie suma partiala

close(f);

exit(0);

}

f = open("/tmp/fifo1", O\_RDONLY);

a[2]+=a[3]; // Procesul parinte

read(f, &a[0], sizeof(int));

close(f);

wait(NULL);

a[0]+=a[2];

printf("Suma este %d\n", a[0]);

}

**Soluţia 3: comunicarea prin FIFO între două procese create din surse diferite.** Tabelul următor prezintă fişierele add4fTata.c şi add4fFiu.c care vor comunica între ele:

|  |  |
| --- | --- |
| **add4fTata.c** | **add4fFiu.c** |
| #include <stdio.h>  #include <unistd.h>  #include <stdlib.h>  #include <fcntl.h>  #include <sys/types.h>  int main () {  int a[] = {1,2,3,4}, f;  f=open("/tmp/fifo1",O\_RDONLY);  a[2]+=a[3];  read(f, &a[0], sizeof(int));  close(f);  a[0]+=a[2];  printf("Suma este %d\n", a[0]);  } | #include <stdio.h>  #include <unistd.h>  #include <stdlib.h>  #include <fcntl.h>  #include <sys/wait.h>  #include <sys/types.h>  int main () {  int a[] = {1,2,3,4}, f;  f = open("/tmp/fifo1", O\_WRONLY);  a[0]+=a[1];  write(f, &a[0], sizeof(int));  close(f);  } |

Inainte de a lansa procesele, trebuie creat FIFO. procesele se pot lansa în orice ordine, deoarece se aşteaptă unul după celălalt. Eventual fiul poate fi lansat în background.

## Simulare sh pentru who | sort şi who | sort | cat (dup2)

**Problema 4:** Simularea unui shell care executa comanda: $ who | sort.

Pentru simulare, programul principal va crea doua procese fii in care va lansa, prin exec, comenzile who si sort. Inainte de crearea acestor fii, va crea un pipe pe care il va da celor doi fii ca sa comunice intre ei: who isi va redirecta iesirea standard in acest pipe cu ajutorul apelului dup2, iar sort va avea ca intrare standard acest pipe, redirectat de asemenea cu dup2.

O extindere naturală este conectarea în pipe a trei programe, de exemplu who | sort | cat. (De aici, generalizarea la un pipeline între n comenzi este uşor de făcut). Sursele celor două programe sunt date în tabelul următor:

|  |  |
| --- | --- |
| **who | sort** | **who | sort | cat** |
| #include <stdio.h>  #include <unistd.h>  #include <sys/wait.h>  int main () {  int p[2];  pipe (p);  if (fork() == 0) {  close (p[0]);  dup2 (p[1], 1);  execlp ("who", "who", NULL);  } else if (fork() == 0) {  close (p[1]);  dup2 (p[0], 0);  execlp ("sort", "sort", NULL);  } else {  close (p[0]);  close (p[1]);  wait (NULL);  wait (NULL);  }  } | #include <stdio.h>  #include <unistd.h>  #include <sys/wait.h>  int main () {  int p[2], q[2];  pipe (p);  pipe (q);  if (fork() == 0) {  close (p[0]);  close (q[0]);  close (q[1]);  dup2 (p[1], 1);  execlp ("who", "who", NULL);  } else if (fork() == 0) {  close (p[1]);  close (q[0]);  dup2 (p[0], 0);  dup2 (q[1], 1);  execlp ("sort", "sort", NULL);  } else if (fork() == 0) {  close (p[0]);  close (p[1]);  close (q[1]);  dup2 (q[0], 0);  execlp ("cat", "cat", NULL);  } else {  close (p[0]);  close (p[1]);  close (q[0]);  close (q[1]);  wait (NULL);  wait (NULL);  wait (NULL);  }  } |

Principiul este simplu: dacă avem n comenzi în pipeline, atunci trebuie construite n-1 pipe-uri. Procesul ce execută prima comandă îşi va redirecta ieşirea standard în primul pipe. Procesul ce execută ultima comandă îşi va redirecta intrarea standard in ultimul pipe. Procesele ce execută comenzile intermediare, să zicem procesul i cu i>1 şi i < n-1, va avea ca intrare standard pipe-ul i-1 şi ca ieşire standard pipe-ul i.

Evident, în locul comenzilor who, sort, cat pot să apară orice comenzi, cu orice argumente.

## Paradigma client / server; exemple

Problema pe care o vom rezolva folosind paradigma client / server este următoarea:

Să se scrie un program **server** care primeşte în FIFO-ul /tmp/CERERE un string de 20 caractere:

|  |  |
| --- | --- |
| **numar** întreg pozitiv fără semn > 1,  exact 10 caractere, cu completare  de zerouri la stânga | **nume** din exact 10 caractere, fără  spaţii albe (blank, tab, \n, \r etc.). |

Serverul descompune **numar** în factori primi şi scrie această descompunere în FIFO-ul **/tmp/nume**

Să se scrie un program **client** care construie un string de 10 caractere **nume** care să reprezinte unic procesul client (nume putând să fie, de exemplu, PID-ul procesului completat la stânga cu zerouri). Clientul primeşte la linia de comandă un număr întreg de maximum 10 cifre, formează cererea ca mai sus şi o scrie într-un FIFO **/tmp/CERERE**. Apoi citeşte răspunsul de la server pe FIFO-ul /tmp/nume, mai întâi lungimea stringului, apoi stringul răspuns. In final tipăreşte răspunsul primit pe ieşirea lui standard, după care şterge FIFO-ul **/tmp/nume**.

Vom da două soluţii pentru **server**:

* Un **server iterativ**, care citeşte o cerere, o execută, trimite răspunsul, apoi iarăşi revine la citirea unei noi cereri ş.a.m.d.
* Un **server concurent**, care citeşte o cerere, crează un proces fiu căruia îi trimite cererea, după care revine la citirea unei noi cereri ş.a.m.d. Toată sarcina de tratare a cererii revine procesului fiu, care evoluează în acelaşi timp cu preluarea de către server a unor noi cereri.

**Intrebare**: de ce este nevoie ca cererea să aibă această structură rigidă?

Vom începe cu **prezentarea client.c**. Sursa acestuia este:

#include <stdio.h>

#include <unistd.h>

#include <stdlib.h>

#include <fcntl.h>

#include <string.h>

int main (int argc, char\* argv[]) {

char cerere[21] = "01234567890123456789",

fiforasp[16] = "/tmp/0123456789", rasp[500];

int lrasp, fc, fr;

sprintf(rasp, "%010lu", atol(argv[1]));

memcpy(cerere, rasp, 10);

sprintf(rasp, "%010d", getpid());

memcpy(&cerere[10], rasp, 10);

memcpy(&fiforasp[5], rasp, 10);

while ((fc = open("/tmp/CERERE", O\_WRONLY)) < 0) sleep(1); // !!

printf("Cerere: %s\n", cerere);

write(fc, cerere, 20);

close(fc);

while ((fr = open(fiforasp, O\_RDONLY)) < 0) sleep(1); // !!

read(fr, &lrasp, sizeof(int));

read(fr, rasp, lrasp);

printf("Raspuns: %s\n", rasp);

close(fr);

unlink(fiforasp);

}

Acţiunea clientului este cea descrisă mai sus. Trebuie să atragem atenţia asupra celor două apeluri sistem open care deschid FIFO de cerere şi de răspuns. Sarcina creării FIFO-urilor revine serverului. Ordinea celor două open trebuie să fie aceeaşi şi la server, altfel se ajunge la deadlock.

Se observă că open se reia până când se reuşeşte deschiderea. Aici reluarea se face din secundă în secundă (se poate şi mai des sau chiar reluare fără pauză). Motivul acestei reluări este acela că dacă serverul ocupă procesorul după client, este posibil ca unul din FIFO-uri să nu fie creat la momentul open al clientului. Dacă se întâmplă asta, se va trece de open, în ciuda regulilor FIFO, şi se semnalează "No such file or directory".

**Cele două servere**, serveriter,c şi serverconc.c sunt foarte asemănătoare. Le prezentăm simultan în tabelul de mai jos.

|  |  |
| --- | --- |
| **serveriter.c** | **serverconc.c** |
| #include <stdio.h>  #include <unistd.h>  #include <stdlib.h>  #include <fcntl.h>  #include <sys/stat.h>  #include <string.h>  int main (int argc, char\* argv[]) {  char cerere[21] = "01234567890123456789";  char fiforasp[16]="/tmp/0123456789";  char rasp[500], factor[50];  int lrasp, fc, fr;  unsigned long n, p, d;  unlink("/tmp/CERERE");  mkfifo("/tmp/CERERE", 0777);  fc = open("/tmp/CERERE", O\_RDONLY);  for ( ; ; ) {  if (read(fc, cerere, 20) != 20)  continue;  memcpy(&fiforasp[5],&cerere[10],10);  cerere[10] = '\0';  n = atol(cerere);  sprintf(rasp, "%lu = ", n);  d = 2;  while(n > 1) {  p = 0;  while(n % d == 0) {  p = p + 1;  n = n / d;  }  if (p > 0) {  sprintf(factor,  "%lu^%lu \* ",d,p);  strcat(rasp, factor);  }  d = d + 1;  }  rasp[strlen(rasp) - 3] = '\0';  lrasp = strlen(rasp) + 1;  unlink(fiforasp);  fr = mkfifo(fiforasp, 0777);  fr = open(fiforasp, O\_WRONLY);  write(fr, &lrasp, sizeof(int));  write(fr, rasp, lrasp);  close(fr);  }  } | #include <stdio.h>  #include <unistd.h>  #include <stdlib.h>  #include <fcntl.h>  #include <sys/stat.h>  #include <string.h>  # include <signal.h>  int main (int argc, char\* argv[]) {  signal(SIGCHLD, SIG\_IGN);  char cerere[21]= "01234567890123456789";  char fiforasp[16]= "/tmp/0123456789";  char rasp[500], factor[50];  int lrasp, fc, fr;  unsigned long n, p, d;  unlink("/tmp/CERERE");  mkfifo("/tmp/CERERE", 0777);  fc = open("/tmp/CERERE", O\_RDONLY);  for ( ; ; ) {  if (read(fc, cerere, 20) != 20)  continue;  if (fork() == 0) {  memcpy(&fiforasp[5],&cerere[10],10);  cerere[10] = '\0';  n = atol(cerere);  sprintf(rasp, "%lu = ", n);  d = 2;  while(n > 1) {  p = 0;  while(n % d == 0) {  p = p + 1;  n = n / d;  }  if (p > 0) {  sprintf(factor,  "%lu^%lu \* ",d,p);  strcat(rasp, factor);  }  d = d + 1;  }  rasp[strlen(rasp) - 3] = '\0';  lrasp = strlen(rasp) + 1;  unlink(fiforasp);  fr = mkfifo(fiforasp, 0777);  fr = open(fiforasp, O\_WRONLY);  write(fr, &lrasp, sizeof(int));  write(fr, rasp, lrasp);  close(fr);  }  }  } |

Serverul concurent, după ce primeşte cererea, crează un proces fiu, care implicit primeşte această cerere. Pentru ca fii să nu rămână în starea zombie, prima instrucţiune din server este apelul sistem signal.

**Client / server ce verifică dreprurile unui fişier**. Clientul primeşte la linia de comandă trei cifre octale şi un nume de fişier. Ii trimite serverului aceste informaţii împreună cu PID-ul clientului. Pentru uniformitate, se trimite la server de exact 109 caractere: PID-ul pe 6, drepturile pe 3 şi numele de fişier pe 100. De asemenea, clientul crează un FIFO cu numele PID-ului, unde să primească răspunsul. Serverul verifică drepturile şi trimite pe FIFO de răspuns un string "Ok" sau "Nu" dacă fişierul are sau nu drepturile.

|  |  |
| --- | --- |
| **Server.c** | **Client.c** |
| #include <unistd.h>  #include <sys/stat.h>  #include <fcntl.h>  #include <stdio.h>  #include <string.h>  #include <sys/types.h>  int main() {  char cerere[109], mode[6], pid[7];  unlink("/tmp/cerere"); // Se sterge preventiv  mkfifo("/tmp/cerere", 0777); // Se creaza FIFO de cereri cu drepturi rw  int f = open("/tmp/cerere", O\_RDONLY);  for ( ; ; ) {  if (read(f, cerere, 109) == 0) { // Nu este solicitat de client  sleep(1); // Dupa o scurta pauza verifica  continue; // sosirea unei noi cereri.  }  pid[6] = 0; struct stat b;  stat(cerere+9, &b);  sprintf(mode,"%d", b.st\_mode);  strncpy(pid, cerere, 6);  printf("Server: cerere=%s pid=%s mode=%s\n",cerere, pid, mode);  int g = open(pid, O\_WRONLY);  if (strncmp(cerere+6, mode, 3) == 0) write(g, "Ok", 3);  else write(g, "Nu", 3);  close(g);  } } | #include <unistd.h>  #include <sys/stat.h>  #include <fcntl.h>  #include <stdio.h>  #include <string.h>  #include <sys/types.h>  int main (int argc, char \*\*argv) {  char cerere[109], pid[7], raspuns[3] ;  sprintf(pid,"%06d",getpid());  pid[6] = 0;  strncpy( cerere, pid, 6);  strncpy( cerere+6, argv[1],3);  strcpy( cerere+9, argv[2]);  printf("Client: cerere=%s pid=%s ",cerere, pid);  unlink(pid); // Se sterge preventiv  mkfifo(pid, 0777); // Se creaza FIFO cu raspunsul  int f = open("/tmp/cerere", O\_WRONLY);  write(f, cerere, 109);  close(f);  int g = open(pid, O\_RDONLY);  read(g, raspuns, 3);  printf("raspuns=%s\n", raspuns);  if (strcmp(raspuns, "Ok") == 0) printf("Totul este Ok!\n");  else printf("Drepturile au fost modificate\n");  close(g);  unlink(pid); // Se sterge FIFO de raspuns  return 0;  } |

## Exemple de utilizare popen

**Utilizare popen cu scriere in intrarea standard** pentru comenda lansată: de exemplu, scrierea in ordine alfabetică a argumentelor şi a variabilelor de mediu:

#include <stdio.h>

main (int argc, char \*argv[], char \*envp[]) {

FILE \*f; int i;

f = popen("sort", "w");

for (i=0; argv[i]; i++ )

fprintf(f, "%s\n", envp[i]);

for (i=0; envp[i]; i++ )

fprintf(f, "%s\n", envp[i]);

pclose (f);

}// Rezultatul, pe iesirea standard

**Utilizare popen cu preluarea iesirii standard** a comenzii lansate, de exemplu să se verificare că userul florin este logat:

#include <stdio.h>

#include <string.h>

main () {

FILE \*f; char l[1000], \*p;

f = popen("who", "r");

for ( ; ; ) {

p = fgets(l, 1000, f);

if (p == NULL) break;

if (strstr(l, "florin")) {

printf("DA\n");

pclose (f);

return;

}

}

printf("NU\n");

pclose (f);

}

**Lansarea în paralel a mai multor programe filtru,** folosind mai multe popen. Presupunem ca avem un program lansabil: $ filtru intrare iesire care transforma fisierul intrare in fisierul iesire dupa reguli stabilite de user. Se cere un program care primeste la linia de comandă mai multe nume de fisiere de intrare, care să fie filtrate în procese paralele în fisiere de ieşire. Programul este:

#include <stdio.h>

#include <string.h>

#include <unistd.h>

main(int argc, char\* argv[]) {

int i;

char c[50][200];

FILE \*f[50];

for (i=1; argv[i]; i++) {

strcpy(c[i], "./filtru ");

strcat(c[i], argv[i]);

strcat(c[i], " ");

strcat(c[i], argv[i]);

strcat(c[i], ".FILTRU");

popen(c[i], "r");

f[i] = popen(c[i], "r");

pclose(f[i]);

}

}

## Probleme propuse

1. Se dă fişierul **grep.c** care conţine fragmentul de cod de mai jos şi care se compilează în directorul personal al utilizatorului sub numele **grep**. Răspundeţi la următoarele întrebări, considerând că toate instrucţiunile se

execut cu succes.

1. Enumeraţi şi explicaţi valorile posibile ale variabilei **n**.
2. Ce vor afişa pe ecran următoarele rulări, considerând că directorul personal al utilizatorului nu se află în variabila de mediu PATH
   1. grep grep grep.c
   2. ./grep grep grep.c
   3. ./grep grep

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21 | int main(int c, char\*\* v) {  int p[2], n;  char s[10] = "ceva";  pipe(p);  n = fork();  if(n == 0) {  close(p[0]);  printf("înainte\n");  if(c > 2)  execlp("grep","grep",v[1],v[2],NULL);  strcpy(s, "dup ");  write(p[1], s, 6);  close(p[1]);  exit(0);  }  close(p[1]);  read(p[0], s, 6);  close(p[0]);  printf("%s\n", s);  return 0;  } |

2. Considerând că toate instrucțiunile din fragmentul de cod de mai jos se execută cu succes, răspundeți la următoarele întrebări:

1. Desenaţi ierarhia proceselor create, incluzând şi procesul părinte.
2. Daţi fiecare linie afişată de program, împreună cu procesul care o tipăreşte.
3. Câte caractere sunt citite din pipe?
4. Cum este afectată terminarea proceselor dacă lipseşte linia 20?
5. Cum este afectată terminarea proceselor dacă lipsesc liniile 20 şi 21?

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22  23 | int main() {  int p[2], i=0;  char c, s[20];  pipe(p);  if(fork() == 0) {  close(p[1]);  while(read(p[0], &c, sizeof(char))) {  if (i<5 || i > 8) {  printf("%c", c);  }  i++;  }  printf("\n"); close(p[0]);  exit(0);  }  printf("Result: \n");  strcpy(s, "exam not passed");  close(p[0]);  write(p[1], s, strlen(s)\*sizeof(char));  close(p[1]);  wait(NULL);  return 0;  } |

3. Clientul transmite serverului un nume de fisier iar serverul intoarce clientului continutul fisierului indicat sau un mesaj de eroare in cazul ca fisierul dorit nu exista.

4. Clientul ii transmite serverului un nume de utilizator, iar serverul ii returneaza clientului datele la care utilizatorul respectiv s-a conectat.

5. Clientul ii transmite serverului un nume de server Unix, si primeste lista tuturor utilizatorilor care lucreaza in acel moment la serverul respectiv.

# Pthreads; sincronizări cu mutex, cond, barrier

## Principalele tipuri de date şi funcţii de lucru cu threaduri

Tabelul următor prezintă principalele fişiere header, tipuri de date şi funcţii care lucrează cu threaduri:

|  |  |
| --- | --- |
| **Fişere header** | <pthread.h> |
| **Specificare biblioteci** | -pthread |
| **Tipuri de date** | pthread\_t  pthread\_mutex\_t  pthread\_cond\_t  pthread\_barrier\_t  pthread\_rwlock\_t  sem\_t |
| **Funcţii de creare thread**  **şi aşteptare terminare** | pthread\_create  pthread\_join  pthread\_exit |
| **Variabile mutex** | pthread\_mutex\_init  pthread\_mutex\_lock  pthread\_mutex\_unlock  pthread\_mutex\_destroy |
| **Variabile condiţionale** | pthread\_cond\_init  pthread\_cond\_wait  pthread\_cond\_signal  pthread\_cond\_broadcast  pthread\_cond\_destroy |
| **Bariere** | pthread\_barrier\_init  pthread\_barrier\_destroy  pthread\_barrier\_wait |
| **Variabile reader/writer** | pthread\_rwlock\_init  pthread\_rwlock\_wrlock  pthread\_rwlock\_rdlock  pthread\_rwlock\_unlock  pthread\_rwlock\_destroy |
| **Semafoare** | sem\_init  sem\_wait  sem\_post  sem\_destroy |
| **Funcţia de descriere a acţiunii threadului** | void \* work(void\* a) |

## Capitalizarea cuvintelor dintr-o listă de fişiere text

Dorim sa transformam un fişier text într-un alt fişier text, cu acelaşi conţinut, dar în care toate cuvintele din el sa înceapă cu literă mare. Un astfel de program / procedura o vom numi **capitalizare** si se apeleaza furnizand doi parametri: **fisierintrare** **fisieriesire**

Ne propunem să prelucrăm simultan mai multe astfel de fisiere. Programul primeste numele a **n** fisiere de intrare, iar fisierele de iesire vor primi acelasi nume la care i se adauga terminatia .CAPIT. Programul va crea câte un thread pentru fiecare pereche de fişiere. Sursa capitalizari.c este:

#include <pthread.h>

#include <stdio.h>

#include <stdlib.h>

#include <string.h>

#include <ctype.h>

#define MAXLINIE 1000

pthread\_t tid[100];

// pthread\_t tid; // Vezi comentariul de la sfârşitul sursei

void\* ucap(void\* numei) {

printf("Threadul: %ld ...> %s\n", pthread\_self(), (char\*)numei);

FILE \*fi, \*fo;

char linie[MAXLINIE], numeo[100], \*p;

strcpy(numeo, (char\*)numei);

strcat(numeo, ".CAPIT");

fi = fopen((char\*)numei, "r");

fo = fopen(numeo, "w");

for ( ; ; ) {

p = fgets(linie, MAXLINIE, fi);

linie[MAXLINIE-1] = '\0';

if (p == NULL) break;

if (strlen(linie) == 0) continue;

linie[0] = toupper(linie[0]); // Cuvant incepe in coloana 0

for (p = linie; ; ) {

p = strstr(p, " ");

if (p == NULL) break;

p++;

if (\*p == '\n') break;

\*p = toupper(\*p);

}

fprintf(fo, "%s", linie);

}

fclose(fo);

fclose(fi);

printf("Terminat threadul: %ld ...> %s\n", pthread\_self(), (char\*)numei);

}

int main(int argc, char\* argv[]) {

int i;

for (i=1; argv[i]; i++) {

pthread\_create(&tid[i], NULL, ucap, (void\*)argv[i]);

// pthread\_create(&tid, NULL, ucap, (void\*)argv[i]); // Vezi comentariul de la sfârşitul sursei

printf("Creat threadul: %ld ...> %s\n", tid[i], argv[i]);

}

for (i=1; argv[i]; i++) pthread\_join(tid[i], NULL);

// for (i=1; argv[i]; i++) pthread\_join(tid, NULL); // Vezi comentariul de la sfârşitul sursei

printf("Terminat toate threadurile\n");

}

Compilarea se face: gcc -pthread capitalizari.c, iar rularea ./a.out f1 f2 . . .

In sursa de mai sus sunt trei linii comentariu, în care, în loc de tid[i] se foloseşte tid. Este vorba de a folosi o singură variabilă în loc de a folosi un tablou în care să se folosească câte o intrare pentru fiecare thread în parte. Din punct de vedere sintactic, folosirea unei singure variabile este perfect valabilă. Insă, din punct de vedere al funcţionării, dacă se foloseşte o singură variabilă, atunci funcţiile pthread\_join vor aştepta numai după ultimul thread!

## Capitalizarea cuvintelor dintr-o listă de fişiere text - soluţia go

//Se capitalizeaza multithreading continutul unor fisiere:./capitalizeWord f1 f2 ...

package main

import (

"bufio"

"fmt"

"io"

"os"

"strings"

"bytes"

)

func ucap(numei string, finish chan<- struct{}) {

print(" Capitalizare: "+numei+"\n")

f,\_ := os.Open(numei)

fi := bufio.NewReader(f)

fo,\_ := os.Create(numei+".CAPIT")

for {

linie,err := fi.ReadString('\n')

if err == io.EOF { break }

r := 0

b := byte(linie[0])

u := string(bytes.ToUpper([]byte{b}))

linie = u+linie[1:]

for {

i := strings.IndexByte(linie[r:], ' ')

if i == -1 { break }

i = r + i + 1

r = i + 1

if r >= len(linie) { break }

b = byte(linie[i])

u = string(bytes.ToUpper([]byte{b}))

linie = linie[:i]+u+linie[(i+1):]

}

fmt.Fprint(fo, linie)

}

f.Close()

fo.Close()

print(" Terminat capitalizare "+numei+"\n")

finish <- struct{}{}

}

func main() {

finish := make(chan struct{})

for i :=1; i < len(os.Args); i++ { go ucap(os.Args[i], finish) }

for i :=1; i < len(os.Args); i++ { <- finish }

print("Terminat toate capitalizarile\n");

}

## Capitalizarea cuvintelor dintr-o listă de fişiere text - soluţia python

# Se capitalizeaza multithreading continutul unor fisiere:./capitalizeWord f1 f2 ...

import sys

import os

import threading

def ucap(numei):

print(" Capitalizare: "+numei+"\n")

fi = open(numei, "r")

fo = open(numei+".CAPIT", "w")

while True:

linie = fi.readline()

if linie == "": break

r = 0

linie = linie[0].upper()+linie[1:]

while True:

i = linie[r:].find(" ")

if i == -1: break

i = r + i + 1

r = i + 1

if r >= len(linie): break

linie = linie[:i]+linie[i].upper()+linie[(i+1):]

fo.write(linie)

fi.close()

fo.close()

print(" Terminat capitalizare "+numei+"\n")

def main():

t = []

for i in range(1, len(sys.argv)):

t.append(threading.Thread(target=ucap, args=(sys.argv[i],)))

t[i-1].start()

for i in range(1, len(sys.argv)):

t[i-1].join()

print("Terminat toate capitalizarile\n")

main()

## De ce sunt necesare variabilele mutex?

Să construim un program care are o variabilă globală **count** şi 1000 de threaduri. Fiecare thread incrementează de câte 1000 de ori variabila count. In acest scop să considerăm programul următor:

#include <pthread.h>

#include <stdio.h>

int count = 0;

pthread\_t tid[1000];

void\* inc(void\* nume) {

for (int i = 0; i < 1000; i++) {

int temp = count; temp++; count = temp;

}

}

int main(int argc, char\* argv[]) {

int i;

for (i=0; i < 1000; i++)

pthread\_create(&tid[i], NULL, inc, NULL);

for (i=0; i < 1000; i++) pthread\_join(tid[i], NULL);

printf("count=%d\n", count);

}

Evident, funcţia inc a threadului se poate scrie: **count += 1000**. In mod intenţionat am "prelungit" execuţia în funcţia threadului. Rezultatele unor execuţii repetate ale programului de mai sus sunt:

florin@ubuntu:~/pthreads$ gcc -pthread inc.c

florin@ubuntu:~/pthreads$ ./a.out

count=997000

florin@ubuntu:~/pthreads$ ./a.out

count=997000

florin@ubuntu:~/pthreads$ ./a.out

count=980000

florin@ubuntu:~/pthreads$ ./a.out

count=1000000

florin@ubuntu:~/pthreads$ ./a.out

count=991000

florin@ubuntu:~/pthreads$ ./a.out

count=996000

florin@ubuntu:~/pthreads$ ./a.out

count=980000

florin@ubuntu:~/pthreads$ ./a.out

count=1000000

florin@ubuntu:~/pthreads$ ./a.out

count=997000

florin@ubuntu:~/pthreads$ ./a.out

count=995000

florin@ubuntu:~/pthreads$

De ce acest comportament ciudat? Pentru că execuţia threadurilor se face în paralel, deci este posibil ca cele trei instrucţiuni ale corpului for din funcţia **inc** să se înterpătrundă, aşa încât două threaduri să memoreze succesiv valoarea lui **count** independent de reţinerea ei în variabila locală temp. Astfel, se pierde efectul unuia dintre threaduri.

Pentru a asigura derularea corectă, se impune utilizarea unui mutex:

Se declară variabila globală

pthread\_mutex\_lock(&exclusiv);

Corpul ciclului se va înlocui cu:

pthread\_mutex\_lock(&exclusiv);

int temp = count; temp++; count = temp;

pthread\_mutex\_unlock(&exclusiv);

## Câte perechi de argumente au suma număr par?

La linia de comandă se dau **n** perechi de argumente despre care se presupune ca sunt numere întregi si pozitive. Se cere numărul de perechi care au suma un număr par, numărul de perechi ce au suma număr impar si numărul de perechi în care cel putin unul dintre argumente nu este număr strict pozitiv.

Rezolvarea: Se va crea câte un thread pentru fiecare pereche. Trei variabile globale, cu rol de contoare, vor număra fiecare categorie de pereche.

**Important:** deoarece threadurile pot incrementa simultan unul dintre contoare, este necesar să se asigure accesul exclusiv la aceste contoare. De aceea, vom folosi o variabilă mutex care va proteja acest acces.

Intrebare: este corect sau nu să fie protejat fiecare contor printr-o variabilă mutex proprie? In exemplul nostru le protejăm simultan pe toate trei cu acelaşi mutex.

Sursa programului este:

#include <pthread.h>

#include <stdio.h>

#include <stdlib.h>

#define MAXLINIE 1000

typedef struct {char\*n1; char\*n2;} PERECHE;

pthread\_t tid[100];

PERECHE pereche[100];

// PERECHE pereche; // Vezi comentariul de la sfârşitul sursei

pthread\_mutex\_t mut = PTHREAD\_MUTEX\_INITIALIZER;

int pare = 0, impare = 0, nenum = 0;

void\* tippereche(void\* pereche) {

int n1 = atoi(((PERECHE\*)pereche)->n1);

int n2 = atoi(((PERECHE\*)pereche)->n2);

if (n1 == 0 || n2 == 0) {

pthread\_mutex\_lock(&mut);

nenum++;

pthread\_mutex\_unlock(&mut);

} else if ((n1 + n2) % 2 == 0) {

pthread\_mutex\_lock(&mut);

pare++;

pthread\_mutex\_unlock(&mut);

} else {

pthread\_mutex\_lock(&mut);

impare++;

pthread\_mutex\_unlock(&mut);

}

}

int main(int argc, char\* argv[]) {

int i, p, n = (argc-1)/2;

for (i = 1, p = 0; p < n; i += 2, p++) {

pereche[p].n1 = argv[i];

pereche[p].n2 = argv[i+1];

pthread\_create(&tid[p], NULL, tippereche, (void\*)&pereche[p]);

// !! Apelul urmator nu este corect, Vezi explicatia la sfârşitul sursei

// pthread\_create(&tid[p], NULL, tippereche, (void\*)&pereche);

}

for (i=0; i < n; i++) pthread\_join(tid[i], NULL);

printf("perechi=%d pare=%d impare=%d nenum=%d\n",n,pare,impare,nenum);

}

In sursa de mai sus există un comentariu unde în loc de pereche[p] se foloseşte pereche. Este vorba de a folosi o aceaşi variabilă simplă în loc de a folosi un tablou în care să se folosească câte o intrare pentru fiecare thread în parte. Din punct de vedere sintactic, folosirea unei singure variabile este perfect valabilă. Insă, din punct de vedere al funcţionării, dacă se foloseşte o singură variabilă, se comite una dintre cele mai dese erori de logică în lucrul cu threaduri: Dacă se foloseşte o singură variabilă care transmite parametrul de intrare pentru toate apelurile ucap ale tuturor threadurilor, este posibil ca (PERECHE\*)pereche)->n1 sau (PERECHE\*)pereche)->n2 să nu preia intrările pregătite, ci să preia valori pregătite pentru threadul următor!

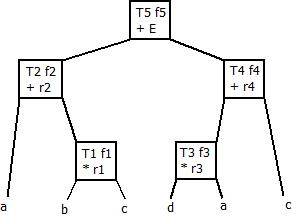
## Evaluarea expresie aritmetică operator / thread şi paralelizare maximă

Se dă expresia aritmetică

E = ( a + b \* c ) + ( d \* a + c )

unde a, b c, d sunt numere întregi. Se cere evaluarea acestei expresii executând fiecare operaţie într-un thread separat şi cu lansarea a câtor mai multe threaduri în acelaşi timp.

Pentru rezolvare, vom folosi cinci threaduri T1, T2, T3, T4, T5, variabilele intermediare r1, r3, r3, r4, E şi funcţiile de thread f1, f2, f3, f4, f5. Arborele din figura următoare ilustrează evaluarea acestei expresii:



In cele cinci pătrate sunt ilustrate cele cinci threaduri care contribuie la rezolvarea problemei. Pentru fiecare thread se indică numele threadului, funcţia care îi determină funcţionarea, operaţia executată şi variabila unde se depune rezultatul operaţiei.

Sursa programului este dată mai jos:

# include <stdio.h>

# include <pthread.h>

pthread\_t t[6];

int a=1, b=2, c=3, d=4, r1, r2, r3, r4, E;

void \* f1 (void \* x) {r1 = b \* c;}

void \* f2 (void \* x) {r2 = a + r1;}

void \* f3 (void \* x) {r3 = d \* a;}

void \* f4 (void \* x) {r4 = r3 + c;}

void \* f5 (void \* x) {E = r2 + r4;}

int main() {

/\* O prima varianta de creare / join

pthread\_create(&t[1], NULL, f1, NULL);

pthread\_create(&t[2], NULL, f2, NULL);

pthread\_create(&t[3], NULL, f3, NULL);

pthread\_create(&t[4], NULL, f4, NULL);

pthread\_create(&t[5], NULL, f5, NULL);

pthread\_join(t[1], NULL);

pthread\_join(t[2], NULL);

pthread\_join(t[3], NULL);

pthread\_join(t[4], NULL);

pthread\_join(t[5], NULL);

Sfarsit varianta 1 - rezultat E=0! \*/

// Varianta corecta:

// T2 lansat dupa ce se termina T1 si T4 lansat dupa ce se termina T3

// T5 lansat numai dupa ce se termina T2 si T4

pthread\_create(&t[1], NULL, f1, NULL);

pthread\_create(&t[3], NULL, f3, NULL);

pthread\_join(t[1], NULL);

pthread\_join(t[3], NULL);

pthread\_create(&t[2], NULL, f2, NULL);

pthread\_create(&t[4], NULL, f4, NULL);

pthread\_join(t[2], NULL);

pthread\_join(t[4], NULL);

pthread\_create(&t[5], NULL, f5, NULL);

pthread\_join(t[5], NULL);

printf("E = %d\n", E);

}

Dacă se rulează prima variantă, în care nici un thread nu aşteaptă după altul, rezultatul este 0! Asta pentru faptul că threadurile care încarcă variabilele r1 - r4 nu apucă să o facă şi threadul T5 preia din ele valorile 0!

Varianta corectă este aceea în care T2 şi T4 aşteaptă să se termine mai întâi T1 şi T2, apoi T5 aşteaptă să se termine mai întâi T2 şi T4.

## De ce sunt necesare variabilele condiţionale?

In exemplul de mai sus unele threaduri au fost nevoite să aştepte terminarea altora pentru a îşi termina activitatea lor. Ce este de făcut dacă NU poate trece de un punct (să zicem **A**) al execuţiei până când un alt thread nu ajunge cu execuţia într-un punct (să zicem **B**). (In literatură fenomenul se numeşte **rendezvouz**). Să schematizăm acest fenomen prin următorul program cu două threaduri, în care evenimentul de rendezvouz este variabila considerăm un exemplu:

#include <pthread.h>

#include <stdio.h>

int eveniment = 0;

// - - -

pthread\_t tid[2];

void\* produceeveniment(void\* nume) {

// - - - Calcule - - -

// B

printf("B va produce evenimentul\n");

eveniment = 1;

printf("B a produs evenimentul\n");

// - - - alte calcule - - -

}

void\* asteaptaeveniment(void\* nume) {

// - - - Calcule - - -

// A

printf("A asteapta evenimentul\n");

while (eveniment == 0) {

;

}

printf("A a primit evenimentul\n");

// - - - alte calcule - - -

}

int main(int argc, char\* argv[]) {

pthread\_create(&tid[1], NULL, asteaptaeveniment, NULL);

pthread\_create(&tid[0], NULL, produceeveniment, NULL);

pthread\_join(tid[0], NULL);

pthread\_join(tid[1], NULL);

}

In funcţie de ordinea de creare a celor două threaduri, efectul execuţiei va fi:

|  |  |
| --- | --- |
| A asteapta evenimentul  B va produce evenimentul  B a produs evenimentul  A a primit evenimentul | B va produce evenimentul  B a produs evenimentul  A asteapta evenimentul  A a primit evenimentul |

**Marea problemă** este aşteptarea evenimentului: **while (eveniment == 0){ ; };**

In modul de mai sus avem de-a face cu o **aşteptare activă**, threadul care aşteaptă ocupă procesorul cu execuţia lui **while**! Este de dorit ca până la apariţia evenimentului, threadul care aşteaptă **să nu ocupe procesorul**! (Din păcate, soluţia cu semnale, pe care am descris-o la procese NU MERGE, deoarece threadurile producător şi de aşteptare sunt în acelaşi proces.).

**Rezolvarea se face printr-o variabilă condiţională**. Pentru aceasta se declară o variabilă condiţională şi o variabilă mutex asociată ei:

pthread\_cond\_t variabila = PTHREAD\_COND\_INITIALIZER;

pthread\_mutex\_t mutexasociat = PTHREAD\_MUTEX\_INITIALIZER;

Secvenţa de provocare a evenimentului se înlocuieşte cu una din variantele (a doua variantă sincronizează şi tipărirea celor două mesaje):

|  |  |
| --- | --- |
| pthread\_mutex\_lock(&mutexasociat);  eveniment = 1;  pthread\_cond\_signal(&variabila);  pthread\_mutex\_unlock(&mutexasociat); | pthread\_mutex\_lock(&mutexasociat);  printf("B va produce evenimentul\n");  eveniment = 1;  pthread\_cond\_signal(&variabila);  printf("B a produs evenimentul\n");  pthread\_mutex\_unlock(&mutexasociat); |

Secvenţa de cedare a procesorului pe durata aşteptării evenimentului se înlocuieşte cu una din variantele (a doua variantă sincronizează şi tipărirea celor două mesaje):

|  |  |
| --- | --- |
| pthread\_mutex\_lock(&mutexasociat);  while (eveniment == 0) {  pthread\_cond\_wait(&variabila,  &mutexasociat);  }  pthread\_cond\_signal(&variabila);  pthread\_mutex\_unlock(&mutexasociat); | pthread\_mutex\_lock(&mutexasociat);  printf("A asteapta evenimentul\n");  while (eveniment == 0) {  pthread\_cond\_wait(&variabila,  &mutexasociat);  }  pthread\_cond\_signal(&variabila);  printf("A a primit evenimentul\n");  pthread\_mutex\_unlock(&mutexasociat); |

In funcţie de ordinea de creare a celor două threaduri, efectul execuţiei va fi:

|  |  |
| --- | --- |
| B va produce evenimentul  B a produs evenimentul  A asteapta evenimentul  A a primit evenimentul | A asteapta evenimentul  B va produce evenimentul  B a produs evenimentul  A a primit evenimentul |

## Bariera - exemplu

Pthread\_barrier este o construcție care permite ca mai multe threaduri independente să „aștepte” în spatele unei „bariere” pana când toate cele care aşteaptă ajung la barieră. Bariera este opțională în standardul POSIX, aşa că unele sisteme de operare nu o includ. Iată un exemplu simplu de utilizare a unei bariere:

#include <stdio.h>

#include <stdlib.h>

#include <unistd.h>

#include <pthread.h>

#include <time.h>

#define THREAD\_COUNT 4

pthread\_barrier\_t mybarrier;

void\* threadFn(void \*id\_ptr) {

int thread\_id = \*(int\*)id\_ptr;

int wait\_sec = 1 + rand() % 5;

printf("thread %d: Wait for %d seconds.\n", thread\_id, wait\_sec);

sleep(wait\_sec);

printf("thread %d: I'm ready...\n", thread\_id);

pthread\_barrier\_wait(&mybarrier);

printf("thread %d: going!\n", thread\_id);

return NULL;

}

int main() {

int i;

pthread\_t ids[THREAD\_COUNT];

int short\_ids[THREAD\_COUNT];

srand(time(NULL));

pthread\_barrier\_init(&mybarrier, NULL, THREAD\_COUNT + 1);

for (i=0; i < THREAD\_COUNT; i++) {

short\_ids[i] = i;

pthread\_create(&ids[i], NULL, threadFn, &short\_ids[i]);

}

printf("main() is ready.\n");

pthread\_barrier\_wait(&mybarrier);

printf("main() is going!\n");

for (i=0; i < THREAD\_COUNT; i++) pthread\_join(ids[i], NULL);

pthread\_barrier\_destroy(&mybarrier);

return 0;

}

## Adunarea în paralel a n numere

Vom da, ca exemplu de utilizare a thread-urilor, evaluarea în paralel a sumei mai multor numere întregi. Evident, operaţia de adunare a *n* numere, chiar dacă *n* este relativ mare, nu impune cu necesitate însumarea lor în paralel. O facem totuşi pentru că reprezintă un exemplu elocvent de calcul paralel, în care esenţa este reprezentată de organizarea prelucrării paralele, aceeaşi şi pentru calcule mult mai complicate.

Presupunem că se dă un număr natural *n*  şi un vector *a* având componentele întregi *a[0], a[1], . . . a[n-1]*. Ne propunem să calculăm, folosind cât mai multe thread-uri, deci un paralelism cât mai consistent, suma acestor numere. Modelul de paralelism pe care ni-l propunem este ilustrat mai jos, pentru *m* = 8:

Mai întâi sunt calculate, în paralel, următoarele patru adunări:

a[0] = a[0] + a[1]; a[2] = a[2] + a[3]; a[4] = a[4] + a[5]; a[6] = a[6] + a[7];

După ce primele două adunări, respectiv ultimele două adunări s-au terminat, se mai execută în paralel încă două adunări:

a[0] = a[0] + a[2]; a[4] = a[4] + a[6];

In sfârşit, la terminarea acestora, se va executa:

a[0] = a[0] + a[4];

Operaţiile de adunare se desfăşoară în paralel, având grijă ca fiecare adunare să se efectueze numai după ce operanzii au primit deja valori în adunările care trebuie să se desfăşoare înaintea celei curente. Este deci necesară o operaţie de *sincronizare* între iteraţii. Considerând că fiecare operaţie de adunare se execută într-o unitate de timp, din cauza paralelismului s-au consumat doar 3 unităţi de timp în loc de 7 unităţi de timp care s-ar fi consumat în abordarea secvenţială. In calcule s-au folosit 7 thread-uri, din care maximum 4 s-au executat în paralel.

Să considerăm acum problema pentru *n* numere şi să implementăm soluţia, cu intenţia de a folosi un număr maxim de thread-uri. Mai întâi extindem setul de numere până la *m* elemente, unde *m* este cea mai mică putere *l* a lui 2 mai mare sau egală cu *n*, adică: *2l-1 < n <= 2l = m,* unde *l = partea întreagă superioară a lui log2n.* Elementele *a[n], . . ., a[m-1]* vor primi valoarea 0. Determinarea valorii lui **m** (cea mai mică putere a lui 2 mai mare sau egală cu **n**) se face uşor prin:

for (m = 1; n > m; m \*= 2);

Pentru organizarea calculelor în regim multithreading, este convenabil să adoptăm o schemă arborescentă de numerotare a thread-urilor, ilustrată în figura de mai jos pentru 32 de numere.

arbore

Frunzele acestui arbore, reprezentate în pătrăţele, reprezintă **operanzii** de adunat. Nodurile interioare, reprezentate în cerculeţe, reprezintă **threadurile** care efectuează adunările.

Un thread oarecare **i** are doi fii. Threadurile de pe ultimul nivel interior au ca fii câte doi operanzi din tabloul de însumat. Celelalte le vom numi **threaduri interioare** şi au câte două **threaduri fii**, numerotate **2\*i** şi **2\*i+1**. Fiecare thread interior îşi va face propria operaţie de adunare numai după ce cei doi fii ai săi îşi vor termina adunările lor.

Fiecare thread **i** face o adunare de forma *a[s] = a[s] + a[d]*. In cele ce urmează vom determina indicii **s** şi **d** în funcţie de **i**. Vom numi **threaduri fraţi** threadurile ce se află pe acelaşi nivel, numerotaţi în ordinea crescătoare a vârstei lor. In cazul nostru, 2 şi 3 sunt fraţi cu 2 cel mai mic, 4, 5, 6 şi 7 sunt fraţi cu 4 cel mai mic, 8, 9, . . ., 15 sunt fraţi cu 8 cel mai mic, 16, 17, . . ., 31 sunt fraţi cu 16 cel mai mic ş.a.m.d. Fraţii cei mici de pe fiecare nivel au ca număr o putere a lui doi.

Este uşor de observat că fraţii de pe acelaşi nivel au acelaşi număr de operanzi: dacă **m** este numărul total de operanzi (putere a lui 2), **i** este numărul unui thread de pe un anumit nivel, iar **j** este numărul fratelui cel mic al acestuia, atunci fraţii de pe acest nivel au fiecare câte **m / j** operanzi. Indicele **s** al primului operand al threadului **i** este egal cu suma numărului de operanzi ai fraţilor lui mai mici, iar pentru **d** se mai adaugă jumătate din numărul de operanzi ai threadului, adică **d = s + m / j / 2**.

Determinarea numărului **j** al fratelui cel mic înseamnă găsirea celei mai mari puteri a lui 2 care este mai mică sau egală cu **i** şi ea se determină uşor prin secvenţa:

for (j = m; j > i; j /= 2);

Cu aceste precizări, sursa programului de adunare multithreading este:

#include <pthread.h>

#include <stdio.h>

#include <stdlib.h>

int n, m; // n = numarul de operanzi; m = min {2^k >= n}

int\* a; // valoarea 1 pentru pana la n-1, 0 de la n la m-1

pthread\_t \*tid; // id-urile threadurilor; -1 thread nepornit

pthread\_mutex\_t print = PTHREAD\_MUTEX\_INITIALIZER; // Printare exclusiva

// Rutina thread-ului nr i de adunare

void\* aduna(void\* pi) {

int i, j, sa, da, st = 0, dr = 0, k;

i = \*(int\*)pi; // Retine numarul threadului

if (i < m / 2) {

st = 2 \* i; // Retine fiul stang

dr = st + 1; // Retine fiul drept

while (tid[st] == -1); // Asteapta sa inceapa fiul stang

// while (tid[st] == -1) sleep(1); // poate asa!

// Cel mai sanatos este să se utilizeze un set de variabile coditionale

// care sa semnaleze pornirile threadurilor.

while (tid[dr] == -1); // Asteapta sa inceapa fiul drept

// while (tid[dr] == -1) sleep(1); // poate asa!

pthread\_join(tid[st], NULL); // Asteapta sa se termine fiul stang

pthread\_join(tid[dr], NULL); // Asteapta sa se termine fiul drept

}

for (j = m; j > i; j /= 2); // Determina fratele cel mic

for (k = j, sa = 0; k < i; k++) sa += m / j; // operand stang

da = sa + m / j / 2; // operand drept

a[sa] += a[da]; // Face adunarea proppriu-zisa

pthread\_mutex\_lock(&print);// Asigura printare exclusiva

printf("Thread %d: a[%d] += a[%d]", i, sa, da);

if (st > 0) printf(" (dupa fii %d %d)\n", st, dr); else printf("\n");

pthread\_mutex\_unlock(&print);

}

// Functia main, in care se creeaza si lanseaza thread-urile

int main(int argc, char\* argv[]) {

n = atoi(argv[1]); // Numarul de numere de adunat

for (m = 1; n > m; m \*= 2); // m = min {2^k >= n}

int\* pi;

int i;

a = (int\*) malloc(m\*sizeof(int)); // Spatiu pentru intregii de adunat

pi = (int\*) malloc(m\*sizeof(int)); // Spatiu pentru indicii threadurilor

tid = (pthread\_t\*) malloc(m\*sizeof(pthread\_t)); // id-threads

for (i = 0; i < n; i++) a[i] = 1; // Aduna numarul 1 de n ori

for (i = n; i < m; i++) a[i] = 0; // Completeaza cu 0 pana la m

for (i = 1; i < m; i++) tid[i] =-1; // Threadurile sunt inca nepornite

for (i = 1; i < m; i++) pi[i] = i; // Threadurile sunt inca nepornite

for (i = 1; i < m; i++)

// De ce folosim mai jos &pi[i] in loc de &i? vezi un exemplu precedent!

pthread\_create(&tid[i], NULL, aduna, (void\*)(&pi[i])); // Threadul i

pthread\_join(tid[1], NULL); // Asteapta dupa primul thread

printf("Terminat adunarile pentru n = %d. Total: %d\n", n, a[0]);

free(a); // Eliberaza tabloul de numere

free(pi); // Elibereaza tabloul de indici de threaduri

free(tid); // Elibereaza tabloul de id-uri de threaduri

}

In legătură cu aşteptările active **while (tid[st] == -1); while (tid[dr] == -1);**

Am folosit acest mod de aşteptare pentru simplitatea codului; mai sunt şi altele posibile:

* Să se pornească threadurile în maniera "bottom up" în arborele asociat.
* variantă "mai puţin ortodoxă" este de a ceda controlat procesorul pentru un interval de timp: **while (tid[st] == -1) sleep(1);**
* Fiecare thread să îşî pornească singur sei doi fii pe care (eventual) îi are.
* Să se folosească un tablou de variabile condiţionale care să semnaleze / să aştepte pornirile threadurilor

# Sincronizarea threadurilor; soluţii ale unor probleme celebre

## Propunere de rezolvare a două probleme simple

1. Sa se scrie un program care creeaza doua thread-uri si are doua variabile globale numite **numere\_pare** si **numere\_impare**. Fiecare thread va genera numere aleatoare si in functie de paritatea lor va incrementa variabila globala respectiva. Thread-urile se opresc cand ambele variabile depasesc 100. Programul principal afiseaza cele doua variabile globale si apoi se termina.

2. Sa se scrie un program care primeste fisiere ca si argumente in linia de comanda. Pentru fiecare argument, programul lanseaza un thread care va calcula dimensiunea fisierului si o va aduna la o variabila globala comuna. Programul principal afiseaza dimensiunea totala a fisierelor primite ca si argumente si se termina.

## Intre A şi B sunt n linii prin care trec m trenuri, m > n

In gara A intră simultan maximum **m** trenuri care vor să ajungă în gara B. De la A spre B există simultan **n** linii, **m > n**. Fiecare tren intră în A la un interval aleator. Dacă are linie liberă între A şi B, o ocupă şi pleacă către B, durata de timp a trecerii este una aleatoare. Să se simuleze aceste treceri. Soluţiile, una folosind variabile condiţionale, cealaltă folosind semafoare, sunt prezentate în tabelul următor.

|  |  |
| --- | --- |
| **trenuriMutexCond.c** | **trenuriSem.c** |
| #include <stdlib.h>  #include <pthread.h>  #include <stdio.h>  #include <unistd.h>  #include <string.h>  #include <time.h>  #define N 5  #define M 13  #define SLEEP 4  pthread\_mutex\_t mutcond;  pthread\_cond\_t cond;  int linie[N], tren[M], inA[M+1], dinB[M+1];  pthread\_t tid[M];  int liniilibere;  time\_t start;  void t2s(int \*t, int l, char \*r) {  int i;  char n[10];  sprintf(r, "[");  for ( i = 0; i < l; i++) {  sprintf(n,"%d, ",t[i]);  strcat(r, n);  }  i = strlen(r) - 1;  if ( r[i] == ' ') r[i - 1] = 0;  strcat(r, "]");  }  void prinT(char \*s, int t) {  int i;  char a[200],l[200],b[200];  for (i = 0; inA[i] != -1; i++);  t2s(inA, i, a);  t2s(linie, N, l);  for (i = 0; dinB[i] != -1; i++);  t2s(dinB, i, b);  printf("%s %d\tA:%s\tLines:%s\tB:%s\ttime: %ld\n",s,t,a,l,b,time(NULL)-start);  }  //rutina unui thread  void\* trece(void\* tren) {  int i, t, l;  t = \*(int\*)tren;  sleep(1 + rand() % SLEEP); // Modificati timpii de stationare    pthread\_mutex\_lock(&mutcond);  for ( i = 0; inA[i] != -1; i++);  inA[i] = t;  prinT("EnterA", t);  for ( ; liniilibere == 0; ) pthread\_cond\_wait(&cond, &mutcond);  for (l = 0; linie[l] != -1; l++);  linie[l] = t;  liniilibere--;  for ( i = 0; inA[i] != t; i++);  for ( ; i < M; inA[i] = inA[i + 1], i++);  prinT(" A => B", t);  pthread\_mutex\_unlock(&mutcond);  sleep(1 + rand() % SLEEP);  pthread\_mutex\_lock(&mutcond);  linie[l] = -1;  liniilibere++;  for ( i = 0; dinB[i] != -1; i++);  dinB[i] = t;  prinT(" OutB", t);  pthread\_cond\_signal(&cond);  pthread\_mutex\_unlock(&mutcond);  }  //main  int main(int argc, char\* argv[]) {  int i;  start = time(NULL);  pthread\_mutex\_init(&mutcond, NULL);  pthread\_cond\_init(&cond, NULL);  liniilibere = N;  for (i = 0; i < N; linie[i] = -1, i++);  for (i = 0; i < M; tren[i] = i, i++);  for (i = 0; i < M + 1; inA[i] = -1, dinB[i] = -1, i++);    // ce credeti despre ultimul parametru &i?  for (i=0; i < M; i++) pthread\_create(&tid[i], NULL, trece, &tren[i]);  for (i=0; i < M; i++) pthread\_join(tid[i], NULL);    pthread\_mutex\_destroy(&mutcond);  pthread\_cond\_destroy(&cond);  return 0;  } | #include <semaphore.h>  #include <pthread.h>  #include <stdlib.h>  #include <stdio.h>  #include <unistd.h>  #include <string.h>  #include <time.h>  #define N 5  #define M 13  #define SLEEP 4  sem\_t sem; // Asteapta / semnaleaza eliberarea uneia din cele N linii  sem\_t sem, mut; // Asigura acces exclusiv la tabelele globale  int linie[N], tren[M], inA[M+1], dinB[M+1];  pthread\_t tid[M];  time\_t start;  void t2s(int \*t, int l, char \*r) {  int i;  char n[10];  sprintf(r, "[");  for ( i = 0; i < l; i++) {  sprintf(n,"%d, ",t[i]);  strcat(r, n);  }  i = strlen(r) - 1;  if ( r[i] == ' ') r[i - 1] = 0;  strcat(r, "]");  }  void prinT(char \*s, int t) {  int i;  char a[200],l[200],b[200];  for (i = 0; inA[i] != -1; i++);  t2s(inA, i, a);  t2s(linie, N, l);  for (i = 0; dinB[i] != -1; i++);  t2s(dinB, i, b);  printf("%s %d\tA:%s\tLines:%s\tB:%s\ttime: %ld\n",s,t,a,l,b,time(NULL)-start);  }  //rutina unui thread  void\* trece(void\* tren) {  int i, t, l;  t = \*(int\*)tren;  sleep(1 + rand()%SLEEP); // Inainte de ==> A    sem\_wait(&mut);  for ( i = 0; inA[i] != -1; i++);  inA[i] = t;  prinT("EnterA", t);  sem\_post(&mut);  sem\_wait(&sem); // In A ocupa linia  sem\_wait(&mut);  for (l = 0; linie[l] != -1; l++);  linie[l] = t;  for ( i = 0; inA[i] != t; i++);  for ( ; i < M; inA[i] = inA[i + 1], i++);  prinT(" A => B", t);  sem\_post(&mut);  sleep(1 + rand()%SLEEP); // Trece trenul A ==> B    sem\_wait(&mut);  linie[l] = -1;  for ( i = 0; dinB[i] != -1; i++);  dinB[i] = t;  prinT(" OutB", t);  sem\_post(&mut);  sem\_post(&sem); // In B elibereaza linia  }  // main  int main(int argc, char\* argv[]) {  int i;  start = time(NULL);  sem\_init(&sem, 0, N);  sem\_init(&mut, 0, 1);  for (i = 0; i < N; linie[i] = -1, i++);  for (i=0; i < M; tren[i] = i, i++);  for (i = 0; i < M + 1; inA[i] = -1, dinB[i] = -1, i++);  // ce credeti despre ultimul parametru &i in loc de &tren[i]?  for (i=0; i < M; i++) pthread\_create(&tid[i], NULL, trece, &tren[i]);  for (i=0; i < M; i++) pthread\_join(tid[i], NULL);  sem\_destroy(&sem);  sem\_destroy(&mut);  return 0;  } |

In varianta cu variabile condiţionale, toate acţiunile critice de gestiune a liniilor şi tipăriri se execută sub protecţia variabilei mutcond. In varianta cu semafoare, pentru protecţie se foloseşte semaforul binar mut; nu este necesară întretinerea unei variabile liniilibere, sarcina aceasta fiind preluata de semaforul sem.

O posibilă execuţie ar fi:

EnterA 8 A:[8] Lines:[-1, -1, -1, -1, -1] B:[] time: 1

A => B 8 A:[] Lines:[8, -1, -1, -1, -1] B:[] time: 1

EnterA 5 A:[5] Lines:[8, -1, -1, -1, -1] B:[] time: 2

A => B 5 A:[] Lines:[8, 5, -1, -1, -1] B:[] time: 2

EnterA 6 A:[6] Lines:[8, 5, -1, -1, -1] B:[] time: 2

A => B 6 A:[] Lines:[8, 5, 6, -1, -1] B:[] time: 2

EnterA 10 A:[10] Lines:[8, 5, 6, -1, -1] B:[] time: 2

A => B 10 A:[] Lines:[8, 5, 6, 10, -1] B:[] time: 2

EnterA 9 A:[9] Lines:[8, 5, 6, 10, -1] B:[] time: 2

A => B 9 A:[] Lines:[8, 5, 6, 10, 9] B:[] time: 2

EnterA 4 A:[4] Lines:[8, 5, 6, 10, 9] B:[] time: 3

EnterA 7 A:[4, 7] Lines:[8, 5, 6, 10, 9] B:[] time: 3

EnterA 11 A:[4, 7, 11] Lines:[8, 5, 6, 10, 9] B:[] time: 3

EnterA 0 A:[4, 7, 11, 0] Lines:[8, 5, 6, 10, 9] B:[] time: 3

OutB 10 A:[4, 7, 11, 0] Lines:[8, 5, 6, -1, 9] B:[10] time: 3

A => B 4 A:[7, 11, 0] Lines:[8, 5, 6, 4, 9] B:[10] time: 3

EnterA 2 A:[7, 11, 0, 2] Lines:[8, 5, 6, 4, 9] B:[10] time: 4

EnterA 3 A:[7, 11, 0, 2, 3] Lines:[8, 5, 6, 4, 9] B:[10] time: 4

EnterA 1 A:[7, 11, 0, 2, 3, 1] Lines:[8, 5, 6, 4, 9] B:[10] time: 4

EnterA 12 A:[7, 11, 0, 2, 3, 1, 12] Lines:[8, 5, 6, 4, 9] B:[10] time: 4

OutB 4 A:[7, 11, 0, 2, 3, 1, 12] Lines:[8, 5, 6, -1, 9] B:[10, 4] time: 4

A => B 7 A:[11, 0, 2, 3, 1, 12] Lines:[8, 5, 6, 7, 9] B:[10, 4] time: 4

OutB 6 A:[11, 0, 2, 3, 1, 12] Lines:[8, 5, -1, 7, 9] B:[10, 4, 6] time: 5

A => B 11 A:[0, 2, 3, 1, 12] Lines:[8, 5, 11, 7, 9] B:[10, 4, 6] time: 5

OutB 9 A:[0, 2, 3, 1, 12] Lines:[8, 5, 11, 7, -1] B:[10, 4, 6, 9] time: 5

A => B 0 A:[2, 3, 1, 12] Lines:[8, 5, 11, 7, 0] B:[10, 4, 6, 9] time: 5

OutB 8 A:[2, 3, 1, 12] Lines:[-1, 5, 11, 7, 0] B:[10, 4, 6, 9, 8] time: 5

A => B 2 A:[3, 1, 12] Lines:[2, 5, 11, 7, 0] B:[10, 4, 6, 9, 8] time: 5

OutB 7 A:[3, 1, 12] Lines:[2, 5, 11, -1, 0] B:[10, 4, 6, 9, 8, 7] time: 5

A => B 3 A:[1, 12] Lines:[2, 5, 11, 3, 0] B:[10, 4, 6, 9, 8, 7] time: 5

OutB 5 A:[1, 12] Lines:[2, -1, 11, 3, 0] B:[10, 4, 6, 9, 8, 7, 5] time: 6

A => B 1 A:[12] Lines:[2, 1, 11, 3, 0] B:[10, 4, 6, 9, 8, 7, 5] time: 6

OutB 0 A:[12] Lines:[2, 1, 11, 3, -1] B:[10, 4, 6, 9, 8, 7, 5, 0] time: 6

A => B 12 A:[] Lines:[2, 1, 11, 3, 12] B:[10, 4, 6, 9, 8, 7, 5, 0] time: 6

OutB 3 A:[] Lines:[2, 1, 11, -1, 12] B:[10, 4, 6, 9, 8, 7, 5, 0, 3] time: 7

OutB 2 A:[] Lines:[-1, 1, 11, -1, 12] B:[10, 4, 6, 9, 8, 7, 5, 0, 3, 2] time: 9

OutB 1 A:[] Lines:[-1, -1, 11, -1, 12] B:[10, 4, 6, 9, 8, 7, 5, 0, 3, 2, 1] time: 9

OutB 12 A:[] Lines:[-1, -1, 11, -1, -1] B:[10, 4, 6, 9, 8, 7, 5, 0, 3, 2, 1, 12] time: 9

OutB 11 A:[] Lines:[-1, -1, -1, -1, -1] B:[10, 4, 6, 9, 8, 7, 5, 0, 3, 2, 1, 12, 11] time: 9

## Intre A şi B sunt n linii prin care trec m trenuri, m > n; soluţia go

package main

import (

"sync"

"time"

"fmt"

"math/rand"

"strings"

)

const N = 5

const M = 13

const SLEEP = 4

var liniilibere int

var linie, tren, tid, inA, dinB []int

var cond sync.Cond

var start time.Time

var finish chan struct{}

func t2s(t []int) string {

r := "["

for i := 0; i < len(t); i++ { r += fmt.Sprintf("%d, ", t[i]) }

if strings.HasSuffix(r, ", ") { r = r[:len(r)-2] }

r += "]"

return r

}

func prinT(s string, t int) {

n := time.Now()

d := n.Sub(start)

r := fmt.Sprintf("%s %d\tA:%s\tLines:%s\tB:%s\ttime: %f",s,t,t2s(inA),t2s(linie),t2s(dinB),d.Seconds())

fmt.Println(r)

}

func trece(t int) {

var l int

time.Sleep(time.Duration(rand.Intn(SLEEP)))

cond.L.Lock()

inA = append(inA, t)

prinT("EnterA", t)

for ; liniilibere == 0; { cond.Wait() }

for l = 0; inA[l] != t; l++ { }

copy(inA[l:], inA[(l+1):])

inA = inA[:(len(inA)-1)]

linie[l] = t

liniilibere -= 1

prinT(" A => B", t)

cond.L.Unlock()

time.Sleep(time.Duration(rand.Intn(SLEEP)))

cond.L.Lock()

linie[l] = -1

liniilibere += 1

dinB = append(dinB, t)

prinT(" OutB=>", t)

cond.Signal()

cond.L.Unlock()

finish <- struct{}{}

}

func main() {

start = time.Now()

finish = make(chan struct{})

liniilibere = N

mutcond := sync.Mutex{}

cond = \*sync.NewCond(&mutcond)

for i := 0; i < N; i++ { linie = append(linie, -1) }

for i := 0; i < M; i++ { tren = append(tren, i) }

for i := 0; i < M; i++ { go trece(tren[i]) }

for i := 0; i < M; i++ { <-finish }

}

## Intre A şi B sunt n linii prin care trec m trenuri, m > n; soluţia python

import threading

import random

import time

import datetime

N = 5

M = 13

SLEEP = 4

cond = threading.Condition()

liniilibere = N

linie, tren, tid, inA, dinB = [], [], [], [], []

start = time.time()

def prinT(s, t):

global start, linie, inA, dinB

print(s+" "+str(t)+"\tA:"+str(inA)+"\tLines:"+str(linie)+"\tB:"+str(dinB)+"\ttime: "+str(time.time()-start))

def trece(t):

global N, M, SLEEP, mut, cond, liniilibere, tren, inA, dinB

time.sleep(random.randint(0, SLEEP))

cond.acquire()

inA.append(t)

prinT("EnterA", t)

while liniilibere == 0: cond.wait()

inA.remove(t)

for l in range(N):

if linie[l] == -1: break

linie[l] = t

liniilibere -= 1

prinT(" A => B", t)

cond.release()

time.sleep(random.randint(0, SLEEP))

cond.acquire()

linie[l] = -1

liniilibere += 1

dinB.append(t)

prinT(" OutB=>", t)

cond.notify()

cond.release()

def main():

global N, M, SLEEP, mut, cond, liniilibere, tren

for i in range(N): linie.append(-1)

for i in range(M): tren.append(i)

for i in range(M):

tid.append(threading.Thread(target=trece, args=(tren[i],)))

tid[i].start()

for i in range(M): tid[i].join()

main()

## Problema frizerului somnoros

Intr-o frizerie există un frizer, un scaun pentru frizer și n scaune pentru clienți care așteaptă. Când nu sunt clienți care așteaptă frizerul stă pe scaunul lui și doarme. Când doarme şi apare primul client, frizerul este trezit. Dacă apare un client si are loc pe scaun atunci așteaptă, altfel pleacă de la frizerie netuns.

|  |  |
| --- | --- |
| **SleepingBarberMutCond.c** | **SleepingBarberSem.c** |
| #include <stdio.h>  #include <stdlib.h>  #include <pthread.h>  #include <unistd.h>  #define N 5  pthread\_mutex\_t mutex;  pthread\_cond\_t somn;  int scauneLibere = N, locTuns = 0, locNou = 0, clientNou = 0, clientTuns = 0;  int scaun[N];  void p(char\* s) {  printf("clientNou: %d, clientTuns: %d, locNou: %d, locTuns: %d, scauneLibere: %d, scaune: [ ", clientNou, clientTuns, locNou, locTuns, scauneLibere);  for (int i = 0; i < N; i++) printf("%d ", scaun[i]);  printf(" ]. %s\n", s);  }  void\* client(void\* a) {  pthread\_mutex\_lock(&mutex);  if (scauneLibere == 0) {  p("Clientul pleaca netuns!");  pthread\_mutex\_unlock(&mutex);  pthread\_exit(NULL);  }  scaun[locNou] = clientNou;  locNou = (locNou + 1) % N;  scauneLibere--;  p("Clientul a ocupat loc");  if (scauneLibere == N - 1) pthread\_cond\_signal(&somn);  pthread\_mutex\_unlock(&mutex);  }  void\* frizer(void \*a) {  for ( ; ; ) {  pthread\_mutex\_lock(&mutex);  while(scauneLibere == N){  p("Frizerul doarme");  pthread\_cond\_wait(&somn, &mutex);  }  clientTuns = scaun[locTuns];  scaun[locTuns] = 0;  locTuns = (locTuns + 1) % N;  scauneLibere++;  p("Frizerul tunde");  pthread\_mutex\_unlock(&mutex);  sleep(2); // Atat dureaza "tunsul"  }  }  int main() {  pthread\_mutex\_init(&mutex, NULL);  pthread\_cond\_init(&somn, NULL);  for (int i = 0; i < N; scaun[i] = 0, i++);  pthread\_t barber;  pthread\_create(&barber, NULL, frizer, NULL);  for ( ; ; ){  pthread\_t customer;  sleep(rand() % 3);  clientNou++;  pthread\_create(&customer, NULL, client, NULL);  }  return 0;  } | #include <stdio.h>  #include <stdlib.h>  #include <pthread.h>  #include <unistd.h>  #include <semaphore.h>  #define N 5  sem\_t mutex, somn;  int locTuns = 0, locNou = 0, clientNou = 0, clientTuns = 0;  int scaun[N];  void p(char\* s) {  printf("clientNou: %d, clientTuns: %d, locNou: %d, locTuns: %d, scaune: [ ", clientNou, clientTuns, locNou, locTuns);  for (int i = 0; i < N; i++) printf("%d ", scaun[i]);  printf(" ]. %s\n", s);  }  void\* client(void\* a) {  sem\_wait(&mutex);  int so;  sem\_getvalue(&somn, &so);  if (so == N) {  p("Clientul pleaca netuns!");  sem\_post(&mutex);  pthread\_exit(NULL);  }  scaun[locNou] = clientNou;  locNou = (locNou + 1) % N;  p("Clientul a ocupat loc");  sem\_post(&somn);  sem\_post(&mutex);  }  void\* frizer(void \*a) {  for ( ; ; ) {  sem\_wait(&mutex);  int so;  sem\_getvalue(&somn, &so);  if (so == 0)  p("Frizerul doarme");  sem\_post(&mutex);  sem\_wait(&somn);  sem\_wait(&mutex);  clientTuns = scaun[locTuns];  scaun[locTuns] = 0;  locTuns = (locTuns + 1) % N;  p("Frizerul tunde");  sem\_post(&mutex);  sleep(2); // Atat dureaza "tunsul"  }  }  int main() {  sem\_init(&mutex, 0, 1);  sem\_init(&somn, 0, 0);  for (int i = 0; i < N; scaun[i] = 0, i++);  pthread\_t barber;  pthread\_create(&barber, NULL, frizer, NULL);  for ( ; ; ){  pthread\_t customer;  sleep(abs(rand() % 3));  clientNou++;  pthread\_create(&customer, NULL, client, NULL);  }  return 0;  } |

## Problema cinei filosofilor

Cinci (n) filosofi sunt așezați la o masă rotundă. Fiecare filosof are în față o farfurie cu spagheti. Pentru a mânca spagheti un filosof are nevoie de două furculițe. Intre două farfurii există o furculiță (5 sau n în total). Viața unui filosof constă din perioade în care gândește și perioade când mănâncă. Când un filosof devine flămând, el încearcă să ia furculițele din stânga și din dreapta. Cănd reușește va mânca un anumit timp după care pune furculițele jos.

Dacă toți ridică simultan furculița din stânga rezultă: **deadlock**. Altfel, după preluarea furculiței din stânga, fiecare verifică să fie disponibilă şi cea din dreapta şi în caz negativ o pune înapoi pe cea din stânga. Dacă toți ridică furculița din stânga simultan, vor vedea furculița din dreapta indisponibilă, vor pune înapoi furculița din stânga și se reia din început: **starvation**

O soluţie simplă, dar cu un paralelism nu prea mare, se obţine dacă se asociază fiecărei furculiţe câte un mutex şi câte un thread fiecărui filosof. Sursa este:

#include <stdio.h>

#include <stdlib.h>

#include <pthread.h>

#include <unistd.h>

#define N 5

int nt[N];

pthread\_t t[N];

pthread\_mutex\_t mutex[N];

void\* filosof(void \*n) {

int i = \*((int\*)n);

for ( ; ; ) {

pthread\_mutex\_lock(&mutex[i]);

pthread\_mutex\_lock(&mutex[(i + 1) % N]);

printf("%d mananca\n", i);

pthread\_mutex\_unlock(&mutex[(i + 1) % N]);

pthread\_mutex\_unlock(&mutex[i]);

sleep(rand()%2); // Cam atat dureaza mancatul

printf("%d cugeta\n", i);

sleep(rand()%3); // Cam atat dureaza cugetatul

}

}

int main() {

int i;

for (i = 0; i < N; i++) {

nt[i] = i;

pthread\_mutex\_init(&mutex[i], NULL);

}

for (i = 0; i < N; i++)

pthread\_create(&t[i], NULL, filosof, &nt[i]);

for (i = 0; i < N; i++)

pthread\_join(t[i], NULL);

}

O soluție care să asigure un maximum de paralelism este ca fiecare filosof să aibă câte două threaduri, unul de mâncare şi unul de cugetare. Pentru a mânca, se asociază fiecărui filosof o variabilă condiţională ce îi dă dreptul să mănânce. Apare un mic inconvenient: este posibil să apară la acelaşi filosof două cugetări consecutive, sau două mâncări consecutive . . .

#include <stdio.h>

#include <stdlib.h>

#include <pthread.h>

#include <unistd.h>

#define FILOSOFI 5

#define MANANCA 1

#define CUGETA 2

#define FLAMAND 3

#define TRUE 1

#define FALSE 0

int stare[FILOSOFI];

int nt[FILOSOFI];

pthread\_t t[2\*FILOSOFI];

pthread\_cond\_t cond[FILOSOFI];

pthread\_mutex\_t mutex[FILOSOFI];

int poateManca(int i) {

int stanga = (i - 1 + FILOSOFI) % FILOSOFI;

int dreapta = (i + 1) % FILOSOFI;

if(stare[i] == FLAMAND && stare[stanga] != MANANCA && stare[dreapta] != MANANCA) {

stare[i] = MANANCA;

pthread\_cond\_signal(&cond[i]);

return TRUE;

} else

return FALSE;

}

void\* mananca(void \*n) {

int i = \*((int\*)n);

while (TRUE) {

pthread\_mutex\_lock(&mutex[i]);

stare[i] = FLAMAND;

while (poateManca(i) == FALSE)

pthread\_cond\_wait(&cond[i], &mutex[i]);

printf("%d mananca\n", i);

pthread\_mutex\_unlock(&mutex[i]);

sleep(abs(rand()%2));

}

}

void\* cugeta(void \*n) {

int i = \*((int\*)n);

while (TRUE) {

pthread\_mutex\_lock(&mutex[i]);

stare[i] = CUGETA;

printf("%d cugeta\n", i);

pthread\_mutex\_unlock(&mutex[i]);

sleep(abs(rand()%5));

}

}

int main() {

int i;

for (i = 0; i < FILOSOFI; i++) {

nt[i] = i;

stare[i] = CUGETA;

pthread\_cond\_init(&cond[i], NULL);

pthread\_mutex\_init(&mutex[i], NULL);

}

for (i = 0; i < FILOSOFI; i++) {

pthread\_create(&t[i], NULL, mananca, &nt[i]);

pthread\_create(&t[i+FILOSOFI], NULL, cugeta, &nt[i]);

}

for (i = 0; i < 2\*FILOSOFI; i++)

pthread\_join(t[i], NULL);

}

## Problema producătorilor şi a consumatorilor

Se dă un *recipient* care poate să memoreze un număr limitat de **n** obiecte în el. Se presupune că sunt active două categorii de procese care accesează acest recipient: *producători* şi *consumatori*. Producătorii introduc obiecte în recipient iar consumatorii extrag obiecte din recipient.

Pentru ca acest mecanism să funcţioneze corect, producătorii şi consumatorii trebuie să aibă acces exclusiv la recipient. In plus, dacă un producător încearcă să acceseze un recipient plin, el trebuie să aştepte consumarea cel puţin a unui obiect. Pe de altă parte, dacă un consumator încearcă să acceseze un recipient gol, el trebuie să aştepte până când un producător introduce obiecte în el.

Pentru implementari, vom crea un **Recipient** având o capacitate limitată MAX. Există un număr oarecare de procese numite **Producător**, care depun, în ordine şi ritm aleator, numere întregi consecutive în acest recipient. Mai există un număr oarecare de procese **Consumator**, care extrag pe rând câte un număr dintre cele existente în recipient.

In textele sursă, tablourile **p**, **v** şi metoda / funcţia **scrie**, sunt folosite pentru afişarea stării recipientului la fiecare solicitare a uneia dintre get sau put. Numărul de producători şi de consumatori sunt fixaţi cu ajutorul constantelor **P** şi **C**.

In sursa unui thread **producător**, variabila **art** dă numărul elementului produs, iar **i** este numărul threadului. După efectuarea unei operaţii **put**, threadul face **sleep** un interval aleator de timp.

In sursa unui thread **consumator**, după o operaţie **get**, acesta intră în **sleep** un interval aleator de timp.

|  |  |
| --- | --- |
| **prodConsMutexCond.c** | **prodConsSem.c** |
| #include <pthread.h>  #include <stdlib.h>  #include <unistd.h>  #include <stdio.h>  #define N 10  #define P 12  #define C 1  #define PSLEEP 5  #define CSLEEP 4  int buf[N], p[P], c[C], nt[P + C];  pthread\_t tid[P + C];  int indPut, indGet, val, bufgol;  pthread\_mutex\_t exclusbuf, exclusval, mutgol, mutplin;  pthread\_cond\_t gol, plin;  //afiseaza starea curenta a producatorilor si a consumatorilor  void afiseaza() {  int i;  for (i=0; i < P; i++) printf("P%d\_%d\t", i, p[i]);  for (i=0; i < C; i++) printf("C%d\_%d\t", i, c[i]);  printf("B: ");  for (i=0; i < N; i++) if (buf[i] != 0) printf("%d ", buf[i]);  printf("\n");  fflush(stdout);  }  //rutina unui thread producator  void\* producator(void\* nrp) {  int indp = \*(int\*)nrp;  for ( ; ; ) {  pthread\_mutex\_lock(&exclusval);  val++;  p[indp] = -val; // Asteapta sa depuna val in buf  pthread\_mutex\_unlock(&exclusval);    pthread\_mutex\_lock(&mutgol);  for ( ; bufgol == 0; ) {  pthread\_cond\_wait(&gol, &mutgol);  }  pthread\_mutex\_unlock(&mutgol);  pthread\_mutex\_lock(&exclusbuf);  buf[indPut] = -p[indp];  bufgol--;  p[indp] = -p[indp]; // A depus val in buf  afiseaza();  p[indp] = 0; // Elibereaza buf si doarme  indPut = (indPut + 1) % N;  pthread\_mutex\_unlock(&exclusbuf);  pthread\_mutex\_lock(&mutplin);  pthread\_cond\_signal(&plin);  pthread\_mutex\_unlock(&mutplin);  sleep(1 + rand() % PSLEEP);  }  }  //rutina unui thread consumator  void\* consumator(void\* nrc) {  int indc = \*(int\*)nrc;  for ( ; ; ) {  c[indc] = -1; // Asteapta sa scoata din buf    pthread\_mutex\_lock(&mutplin);  for ( ; bufgol == N; ) {  pthread\_cond\_wait(&plin, &mutplin);  }  pthread\_mutex\_unlock(&mutplin);  pthread\_mutex\_lock(&exclusbuf);  c[indc] = buf[indGet]; // Scoate o valoare din buf  buf[indGet] = 0; // Elibereaza locul din buf  bufgol++;  afiseaza();  c[indc] = 0; // Elibereaza buf si doarme  indGet = (indGet + 1) % N;  pthread\_mutex\_unlock(&exclusbuf);  pthread\_mutex\_lock(&mutgol);  pthread\_cond\_signal(&gol);  pthread\_mutex\_unlock(&mutgol);  sleep(1 + rand() % CSLEEP);  }  }  //functia principala  int main() {  pthread\_mutex\_init(&exclusbuf, NULL);  pthread\_mutex\_init(&exclusval, NULL);  pthread\_mutex\_init(&mutgol, NULL);  pthread\_mutex\_init(&mutplin, NULL);  pthread\_cond\_init(&gol, NULL);  pthread\_cond\_init(&plin, NULL);  int i;  val = 0;  indPut = 0;  indGet = 0;  bufgol = N;  for (i=0; i < N; buf[i] = 0, i++);  for (i=0; i < P; p[i] = 0, nt[i] = i, i++);  for (i=0; i < C; c[i] = 0, nt[i + P] = i, i++);    for (i = 0; i < P; i++) pthread\_create(&tid[i], NULL, producator, &nt[i]);  for (i = P; i < P + C; i++) pthread\_create(&tid[i], NULL, consumator, &nt[i]);  for (i = 0; i < P + C; i++) pthread\_join(tid[i], NULL);  pthread\_mutex\_destroy(&exclusbuf);  pthread\_mutex\_destroy(&exclusval);  pthread\_mutex\_destroy(&mutgol);  pthread\_mutex\_destroy(&mutplin);  pthread\_cond\_destroy(&gol);  pthread\_cond\_destroy(&plin);  } | #include <semaphore.h>  #include <pthread.h>  #include <stdlib.h>  #include <unistd.h>  #include <stdio.h>  #define N 10  #define P 12  #define C 1  #define PSLEEP 5  #define CSLEEP 4  int buf[N], p[P], c[C], nt[P + C];  pthread\_t tid[P + C];  int indPut, indGet, val;  sem\_t exclusbuf, exclusval, gol, plin;  //afiseaza starea curenta a producatorilor si a consumatorilor  void afiseaza() {  int i;  for (i=0; i < P; i++) printf("P%d\_%d\t", i, p[i]);  for (i=0; i < C; i++) printf("C%d\_%d\t", i, c[i]);  printf("B: ");  for (i=0; i < N; i++) if (buf[i] != 0) printf("%d ", buf[i]);  printf("\n");  fflush(stdout);  }  //rutina unui thread producator  void\* producator(void\* nrp) {  int indp = \*(int\*)nrp;  for ( ; ; ) {  sem\_wait(&exclusval);  val++;  p[indp] = -val; // Asteapta sa depuna val in buf  sem\_post(&exclusval);    sem\_wait(&gol);    sem\_wait(&exclusbuf);  buf[indPut] = -p[indp]; // A depus val in buf  p[indp] = -p[indp];  afiseaza();  p[indp] = 0; // Elibereaza buf si doarme  indPut = (indPut + 1) % N;  sem\_post(&exclusbuf);    sem\_post(&plin);    sleep(1 + rand() % PSLEEP);  }  }  //rutina unui thread consumator  void\* consumator(void\* nrc) {  int indc = \*(int\*)nrc;  for ( ; ; ) {  c[indc] = -1; // Asteapta sa scoata din buf    sem\_wait(&plin);    sem\_wait(&exclusbuf);  c[indc] = buf[indGet]; // Scoate o valoare din buf  buf[indGet] = 0; // Elibereaza locul din buf  afiseaza();  c[indc] = 0; // Elibereaza buf si doarme  indGet = (indGet + 1) % N;  sem\_post(&exclusbuf);    sem\_post(&gol);    sleep(1 + rand() % CSLEEP);  }  }  //functia principala  int main() {  sem\_init(&exclusbuf, 0, 1);  sem\_init(&exclusval, 0, 1);  sem\_init(&gol, 0, N);  sem\_init(&plin, 0, 0);  int i;  val = 0;  indPut = 0;  indGet = 0;  for (i = 0; i < N; buf[i] = 0, i++);  for (i = 0; i < P; p[i] = 0, nt[i] = i, i++);  for (i=0; i < C; c[i] = 0, nt[i + P] = i, i++);    for (i = 0; i < P; i++) pthread\_create(&tid[i], NULL, producator, &nt[i]);  for (i = P; i < P + C; i++) pthread\_create(&tid[i], NULL, consumator, &nt[i]);  for (i = 0; i < P + C; i++) pthread\_join(tid[i], NULL);  sem\_destroy(&exclusbuf);  sem\_destroy(&exclusval);  sem\_destroy(&gol);  sem\_destroy(&plin);  } |

Situaţia la un moment dat este dată prin stările producătorilor, stările consumatorilor şi conţinutul bufferului după efectuarea operaţiei.

Stările fiecărui producător (**P**) sunt afişate prin câte un întreg:

* <0 indică aşteptare la tampon plin pentru depunerea elementului pozitiv corespunzător,
* >0 dă valoarea elementului depus,
* 0 indică producător inactiv pe moment.

Stările fiecărui consumator(**C**) sunt afişate prin câte un întreg:

* -1 indică aşteptare la tampon gol,
* >0 dă valoarea elementului consumat,
* 0 indică consumator inactiv pe moment.

## Problema cititorilor şi a scriitorilor

Se dă o *resursă* la care au acces două categorii de procese: *cititori* şi *scriitori.* Regulile de acces sunt: la un moment dat resursa poate fi accesată simultan de **oricâţi scriitori** sau **exact de un singur scriitor**.

Problema este inspirată din accesul la baze de date (resursa). Procesele cititori accesează resursa numai în citire, iar scriitorii numai în scriere. Se permite ca mai mulţi cititori să citească simultan baza de date. In schimb fiecare proces scriitor trebuie să acceseze exclusiv la baza de date.

Simularea noastră se face astfel.

Pentru implementari, consideram un obiect pe care Il vom numi “bază de date” (**Bd**), . Există un număr oarecare de procese numite **Scriitor**, care efectuează, în ordine şi ritm aleator, scrieri în bază. Mai există un număr oarecare de procese **Cititor**, care efectuează citiri din **Bd**.

O operaţie de scriere este efectuată asupra **Bd** în mod individual, fără ca alţi scriitori sau cititori să acceseze **Bd** în acest timp. Dacă **Bd** este utilizată de către alte procese, scriitorul aşteaptă până când se eliberează, după care execută scrierea. In schimb, citirea poate fi efectuată simultan de către oricâţi cititori, dacă nu se execută nici o scriere în acel timp. In cazul că asupra **Bd** se execută o scriere, cititorii aşteaptă până când se eliberează **Bd**.

Variabila **cititori** reţine de fiecare dată câţi cititori sunt activi la un moment dat. După cum se poate observa, instanţa curentă a lui **Bd** este blocată (pusă în regim de monitor) pe parcursul acţiunilor asupra variabilei **cititori**. Aceste acţiuni sunt efectuate numai în interiorul metodelor **scrie** şi **citeste**.

Metoda **citeste** incrementează (în regim monitor) numărul de cititori. Apoi, posibil concurent cu alţi cititori, îşi efectuează activitatea, care aici constă doar în afişarea stării curente. La terminarea acestei activităţi, în regim monitor decrementează şi anunţă thread-urile de aşteptare. Acestea din urmă sunt cu siguranţă numai scriitori. Metoda **scrie** este atomică (regim monitor), deoarece întreaga ei activitate se desfăşoară fără ca celelalte procese să acţioneze asupra **Bd**.

Metoda **afisare** are rolul de a afişa pe ieşirea standard starea de fapt la un moment dat. Situaţia la un moment dat este dată prin stările cititorilor şi ale scriitorilor. Stările fiecărui scriitor (**S**) sunt afişate prin câte un întreg: **-3** indica scriitor nepornit, **-2** indica faptul ca scriitorul a scris si urmeaza sa doarma, **-1** indică aşteptare ca cititorii să-şi termine operaţiile, **0** indică scriere efectivă. In mod analog, stările fiecărui cititor (**C**) sunt afişate prin câte un întreg: **-3** cititor nepornit, **-2** a citit si urmeaza sa doarma, **-1** indică aşteptarea terminării scrierilor, **0** indică citire efectivă.

Vom prezenta trei implementări:

* **citScrMutexCond.c** care folosesc variabile mutex şi variabile condiţionale.
* **citScrSem.c** care folosesc semafoare.
* **cirScrRWlock.v** care folosesc in instrument de sincronizare specific: blocare reader / writer.

Sursele acestor implementări sunt:

**citScrMutexCond.c**

#include <pthread.h>

#include <stdlib.h>

#include <unistd.h>

#include <stdio.h>

#define S 5

#define C 5

#define CSLEEP 2

#define SSLEEP 3

pthread\_t tid[C + S];

int c[C], s[S], nt[C + S];

pthread\_mutex\_t mutcond, exclusafis;

pthread\_cond\_t cond;

int cititori;

//afiseaza starea curenta a cititorilor si scriitorilor

void afiseaza() {

int i;

pthread\_mutex\_lock(&exclusafis);

for (i = 0; i < C; i++) printf("C%d\_%d\t",i, c[i]);

for (i = 0; i < S; i++) printf("S%d\_%d\t",i, s[i]);

printf("\n");

fflush(stdout);

pthread\_mutex\_unlock(&exclusafis);

}

//rutina thread cititor

void\* cititor(void\* nrc) {

int indc = \*(int\*)nrc;

for ( ; ; ) {

c[indc] = -1; // Asteapta sa citeasca

pthread\_mutex\_lock(&mutcond);

cititori++;

c[indc] = 0; // Citeste

afiseaza();

pthread\_mutex\_unlock(&mutcond);

sleep(1 + rand() % CSLEEP);

c[indc] = -2; // A citit si doarme

pthread\_mutex\_lock(&mutcond);

cititori--;

pthread\_cond\_signal(&cond);

pthread\_mutex\_unlock(&mutcond);

sleep(1 + rand() % CSLEEP);

}

}

void\* scriitor (void\* nrs) {

int inds = \*(int\*)nrs;

for ( ; ; ) {

s[inds] = -1; // Asteapta sa scrie

pthread\_mutex\_lock(&mutcond);

for ( ; cititori > 0; ) {

pthread\_cond\_wait(&cond, &mutcond);

}

s[inds] = 0; // Scrie

afiseaza();

sleep(1 + rand() % SSLEEP);

s[inds] = -2; // A scris si doarme

pthread\_mutex\_unlock(&mutcond);

sleep(1 + rand() % SSLEEP);

}

}

//functia principala "main"

int main() {

pthread\_mutex\_init(&exclusafis, NULL);

pthread\_mutex\_init(&mutcond, NULL);

pthread\_cond\_init(&cond, NULL);

int i;

for (i = 0; i < C; c[i] = -3, nt[i] = i, i++); // -3 : Nu a pornit

for (i = 0; i < S; s[i] = -3, nt[i + C] = i, i++);

for (i = 0; i < C; i++) pthread\_create(&tid[i], NULL, cititor, &nt[i]);

for (i = C; i < C + S; i++) pthread\_create(&tid[i], NULL, scriitor, &nt[i]);

for (i = 0; i < C + S; i++) pthread\_join(tid[i], NULL);

pthread\_cond\_destroy(&cond);

pthread\_mutex\_destroy(&mutcond);

pthread\_mutex\_destroy(&exclusafis);

}

**citScrSem.c**

#include <semaphore.h>

#include <pthread.h>

#include <stdlib.h>

#include <unistd.h>

#include <stdio.h>

#define C 2

#define S 5

#define CSLEEP 3

#define SSLEEP 1

pthread\_t tid[C + S];

int c[C], s[S], nt[C + S];

sem\_t semcititor, exclusscriitor, exclusafis;

int cititori;

//afiseaza starea curenta a cititorilor si scriitorilor

void afiseaza() {

int i;

sem\_wait(&exclusafis);

for (i = 0; i < C; i++) printf("C%d\_%d\t",i, c[i]);

for (i = 0; i < S; i++) printf("S%d\_%d\t",i, s[i]);

printf("\n");

fflush(stdout);

sem\_post(&exclusafis);

}

//rutina thread cititor

void\* cititor(void\* nrc) {

int indc = \*(int\*)nrc;

for ( ; ; ) {

c[indc] = -1; // Asteapta sa citeasca

sem\_wait(&semcititor);

cititori++;

if (cititori == 1) sem\_wait(&exclusscriitor);

sem\_post(&semcititor);

c[indc] = 0; // Citeste

afiseaza();

sleep(1 + rand() % CSLEEP);

c[indc] = -2; // A citit si doarme

sem\_wait(&semcititor);

cititori--;

if (cititori == 0) sem\_post(&exclusscriitor);

sem\_post(&semcititor);

sleep(1 + rand() % CSLEEP);

}

}

//rutina thread scriitor

void\* scriitor (void\* nrs) {

int inds = \*(int\*)nrs;

for ( ; ; ) {

s[inds] = -1; // Asteapta sa scrie

sem\_wait(&exclusscriitor);

s[inds] = 0; // Scrie

afiseaza();

sleep(1 + rand() % SSLEEP);

s[inds] = -2; // A scris si doarme

sem\_post(&exclusscriitor);

sleep(1 + rand() % SSLEEP);

}

}

//functia principala "main"

int main() {

sem\_init(&semcititor, 0, 1);

sem\_init(&exclusscriitor, 0, 1);

sem\_init(&exclusafis, 0, 1);

int i;

for (i = 0; i < C; c[i] = -3, nt[i] = i, i++); // -3 : Nu a pornit

for (i = 0; i < S; s[i] = -3, nt[i + C] = i, i++);

for (i = 0; i < C; i++) pthread\_create(&tid[i], NULL, cititor, &nt[i]);

for (i = C; i < C + S; i++) pthread\_create(&tid[i], NULL, scriitor, &nt[i]);

for (i = 0; i < C + S; i++) pthread\_join(tid[i], NULL);

sem\_destroy(&semcititor);

sem\_destroy(&exclusscriitor);

sem\_destroy(&exclusafis);

}

**citScrRWlock.c**

#include <pthread.h>

#include <stdlib.h>

#include <unistd.h>

#include <stdio.h>

#define C 7

#define S 5

#define CSLEEP 2

#define SSLEEP 3

pthread\_t tid[C + S];

int c[C], s[S], nt[C + S];

pthread\_rwlock\_t rwlock;

pthread\_mutex\_t exclusafis;

//afiseaza starea curenta a cititorilor si scriitorilor

void afiseaza() {

int i;

pthread\_mutex\_lock(&exclusafis);

for (i = 0; i < C; i++) printf("C%d\_%d\t",i, c[i]);

for (i = 0; i < S; i++) printf("S%d\_%d\t",i, s[i]);

printf("\n");

fflush(stdout);

pthread\_mutex\_unlock(&exclusafis);

}

//rutina thread cititor

void\* cititor(void\* nrc) {

int indc = \*(int\*)nrc;

for ( ; ; ) {

c[indc] = -1; // Asteapta sa citeasca

pthread\_rwlock\_rdlock(&rwlock);

c[indc] = 0; // Citeste

afiseaza();

sleep(1 + rand() % CSLEEP);

c[indc] = -2; // A citit si doarme

pthread\_rwlock\_unlock(&rwlock);

sleep(1 + rand() % CSLEEP);

}

}

//rutina thread scriitor

void\* scriitor (void\* nrs) {

int inds = \*(int\*)nrs;

for ( ; ; ) {

s[inds] = -1; // Asteapta sa scrie

pthread\_rwlock\_wrlock(&rwlock);

s[inds] = 0; // Scrie

afiseaza();

sleep(1 + rand() % SSLEEP);

s[inds] = -2; // A scris si doarme

pthread\_rwlock\_unlock(&rwlock);

sleep(1 + rand() % SSLEEP);

}

}

//functia principala "main"

int main() {

pthread\_rwlock\_init(&rwlock, NULL);

pthread\_mutex\_init(&exclusafis, NULL);

int i;

for (i = 0; i < C; c[i] = -3, nt[i] = i, i++); // -3 : Nu a pornit

for (i = 0; i < S; s[i] = -3, nt[i + C] = i, i++);

for (i = 0; i < C; i++) pthread\_create(&tid[i], NULL, cititor, &nt[i]);

for (i = C; i < C + S; i++) pthread\_create(&tid[i], NULL, scriitor, &nt[i]);

for (i = 0; i < C + S; i++) pthread\_join(tid[i], NULL);

pthread\_rwlock\_destroy(&rwlock);

pthread\_mutex\_destroy(&exclusafis);

}

## Utilizarea altor platforme de threaduri

Tabelul următor prezintă comparativ trei platforme de lucru cu threaduri în C.

|  |  |  |  |
| --- | --- | --- | --- |
| **API elems. \OS** | **Linux** | **Solaris** | **MS Windows** |
| **Headers** | #include<stdio.h>  #include<pthread.h>  #include<stdlib.h>  #include <semaphore.h> | #include<stdio.h>  #include<thread.h>  #include<synch.h>  #include <semaphore.h>  #include<stdlib.h>  #include<math.h> | #include <windows.h>  #include <stdlib.h>  #include <stdio.h>  #include <math.h> |
| **Libraries** | -lpthread -lm | -lrt -lm |  |
| **Data Types** | pthread\_t  pthread\_mutex\_t  pthread\_cond\_t  pthread\_rwlock\_t  sem\_t | thread\_t  mutex\_t  cond\_t  rwlock\_t  sema\_t | HANDLE  CRITICAL\_SECTION  CONDITION\_VARIABLE  SRWLOCK  HANDLE |
| **Threads** | pthread\_create  pthread\_join | thr\_create  thr\_join | CreateThread  WaitForSingleObject |
| **Function Decl** | void\* worker(void\* a) | void\* worker(void\* a) | DWORD WINAPI worker(LPVOID a) |
| **Mutexes** | pthread\_mutex\_init  pthread\_mutex\_lock  pthread\_mutex\_unlock  pthread\_mutex\_destroy | mutex\_init  mutex\_lock  mutex\_unlock  mutex\_destroy | InitializeCriticalSection  EnterCriticalSection  LeaveCriticalSection  DeleteCriticalSection |
| **Conditional**  **Variables** | pthread\_cond\_init  pthread\_cond\_wait  pthread\_cond\_signal  pthread\_cond\_destroy | cond\_init  cond\_wait  cond\_signal  cond\_destroy | InitializeConditionVariable  SleepConditionVariableCS  WakeConditionVariable  !Trebuie compilate cu Visual  Studio incepand cu Vista,  Windows 7 si mai recente! |
| **Read/Write**  **Locks** | pthread\_rwlock\_init  pthread\_rwlock\_wrlock  pthread\_rwlock\_rdlock  pthread\_rwlock\_unlock  pthread\_rwlock\_destroy | rwlock\_init  rw\_wrlock  rw\_rdlock  rw\_unlock    rwlock\_destroy | InitializeSRWLock  AcquireSRWLockExclusive  AcquireSRWLockShared  ReleaseSRWLockExclusive  AcquireSRWLockShared    !Trebuie compilate cu Visual  Studio incepand cu Vista,  Windows 7 si mai recente! |
| **Semaphores** | sem\_init  sem\_wait  sem\_post  sem\_destroy | sema\_init  sema\_wait  sema\_post  sema\_destroy | CreateSemaphore  WaitForSingleObject  ReleaseSemaphore  CloseHandle |

In fişierul **threads.zip** sunt implementate prezentate mai sus pe diverse platforme şi folosind diverse instrumente de sincronizare.

## Probleme propuse

# API Unix vs API Windows

## Deosebiri formale Unix - Windows

Principalele deosebiri formale între SO Unix şi SO Windows sunt prezentate în tabelul următor:

|  |  |  |  |
| --- | --- | --- | --- |
|  |  | Unix | Windows |
| 1 | Specificare absoluta fisier | /dir1/dir2/.../dirn/fisier | d:\dir1\dir2\...\dirn\fisier |
| 2 | Separator directoare PATH | dir1:dir2:...:dirn | dir1;dir2;...;dirn |
| 3 | Specificare optiune | com -opt | com /opt |
| 4 | Separtor linii in fisier text  (Mac OS linie\rlinie CR) | linie\nlinie (LF = 0A) | linie\r\nlinie (CR LF = 0D 0A) |
| 5 | Parametrii linie comanda:  com arg1 arg2 ... argn | $0 $1 ... $9 | %0 %1 ... %9 |
| 6 | Valoarea unei variabile shell | ${nume} | %nume% |

## Fişiere de comenzi de tip bat (MS-Windows)

Detalii in cap. 6, Boian Fl. s.a. Sisteme de operare, Risoprint, 2006

Fisierele de comenzi DOS / Windows sunt, intr-o oarecare masura, similare fisierelor de comenzi Shell de sub Unix. Ele sunt mult mai sarace in facilitati si directive decat cele oferite de sistemele Shell. Un fisier de comenzi DOS / Windows contine in interiorul lui comenzi DOS si un numar limitat de directive. Din punct de vedere formal, numele unui fisier de comenzi trebuie sa se termine (sa fie de tipul / extensia) .bat

Continutul unui fisier de comenzi bat:

* comenzi DOS;
* etichete (nume la inceput de linie precedat de :);
* caracterele speciale | > < @ % ;
* parametri formali (%n);
* variabile globale (%nume%);
* variabile locale (numai in FOR);
* directive.

Directivele principale:

FOR %%variabilalocala IN ( multime ) DO comanda **variabilalocala** parcurge **multime** si pentru fiecare valoare executa **comanda**

CALL fisiercomenzi [ parametri ]fişierul de comenzi cheama **fisierdecomenzi** şi după execuţie revine.

IF [NOT] ERRORLEVEL n comanda daca codul de retur al comenzii precedente este mai mare sau egal decât **n** (sau strict mai mic în cazul NOT), atunci se executa **comanda**.

IF [NOT] sir1 == sir2 comanda dacă cele două şiruri sunt egale (sau diferite in cazul NOT), atunci se executa **comanda**.

IF [NOT] EXIST fisier comanda dacă **fisier** există (sau nu există în cazul NOT), atunci se executa **comanda**.

GOTO eticheta urmatoarea linie de executat va fi cea marcata cu **eticheta**.

SHIFT mută spre stânga cu o poziţie argumentele liniei de comandă: %0 se pierde, %1 devine %0, %2 devine %1 s.a.m.d.

SET nume=valoare defineste variabila de mediu **nume** careia îi atribuie **valoare**; utilizarea ei (obtinerea valorii) se face prin %**nume**%

ECHO [ ON | OFF | mesaj ] permite sau interzice afisarea la executie a liniilor fisierului de comenzi, sau afiseaza **mesaj** pe iesirea standard.

PAUSE [ mesaj ] afişează **mesaj** pe iesirea standard şi asteapta apăsarea unei taste.

REM comentariu definirea unei linii comentariu

Comenzi DOS mai des folosite:

* de lucru cu discul: diskcopy, sys, format, fdisk, chkdsk
* de lucru cu directoare: mkdir(md), chdir(cd), rmdir(rd), dir, path, subst
* de lucru cu fisiere: more, attrib, del, erase, deltree, fc, find, move, rename, sort, xcopy, copy, type, print
* eticheta de volum: label, vol
* alte comenzi: choice, edit, keyb, mode, cls, date, time, ver, echo, rem

In unele situatii, directivele bat si comenzile DOS nu sunt suficiente pentru rezolvarea unor probleme cu fisiere bat. Din aceasta cauză, utilizatorul se vede nevoit să mai scrie o serie de mici programe (de exemplu în C sau C++), care să se termine cu diverse coduri de retur şi să fie integrate în fisierele de comenzi.

## Exemplu de fişier bat: concatenare fişiere

Sa se scrie un fisier de comenzi care primeste cel puţin doi parametrii: primul este numele, eventual specificat absolut, al unui fişier text în care se concatenează fişierele ale căror nume urmează în lista de parametrii. Pentru rezolvare vom reţine in variabila **dest** primul parametru. Apoi facem un SHIFT pentru a trece la următorul parametru. In continuare, într-o structură repetitivă vom parcurgem restul parametrilor. Pentru fiecare testam existenta fişierului şi în caz afirmativ il vom adăuga continutul la **dest**.

Sursa, in fisierul **concat.bat** este:

@echo off

REM verificam numarul de parametrii

if "%2"=="" goto err1

set dest=%1

shift

REM parcurgem ceilalti parametrii din linia de comanda

:loop

if "%1"=="" goto end

REM verificam daca exista %1 si in caz afirmativ il concatenam

if EXIST %1 type %1 >>%dest%

shift

goto loop

:err1

echo Trebuie minim doi parametrii!

goto end

:end

REM afisam continutul fisierului toate.txt

type %dest%

Apelul se face:

- - ->concat.bat dir1\toate.txt a.txt b.txt c ex1.pas tpc.cpp

## Particularităţi C / C++ pentru MS-Windows

Limbajul nativ de dezvoltare a aplicaţiilor Windows este C++. Din această cauză, în cele ce urmează vom descrie principiile programării folosind construcţii C şi C++. Headerul <windows.h> conţine principalele construcţii de limbaj folosite în interfaţa Windows.

**Constante.** In <windows.h> se definesc o serie de constante. Numele acestora este compus din două părţi: o primă parte indică grupul din care face parte constanta, apoi caracterul “\_” şi în final numele specific, de regulă suficient de lung încât să sugereze ce reprezintă. Pentru detalii se poate consulta MSDN.

**Tipuri de date.** Windows foloseşte o serie de tipuri de date prin care s-a urmărit creşterea portabilităţii aplicaţiilor în cazul unor noi arhitecturi de calculatoare. Astfel, avem tipurile BOOL, BYTE, DWORD (32 biţi), FARPROC (pointer spre funcţie), LPSTR (pointer către string), LPMSG (pointer către o structură MSG) etc.

Pentru desemnarea obiectelor sunt definite nişte tipuri de date speciale: *descriptor* sau *handle*. Acestea sunt întregi pe 16 biţi prin intermediul cărora se pot referi obiecte: fişiere, procese, threaduri, evenimente, timere etc.

**Nume de variabile.** Atribuirea de nume pentru variabile se face respectându-se anumite convenţii, provenite din experienţa programatorilor. Este vorba de notaţia ungară de denumire a variabilelor. Ele nu sunt restricţii impuse de sistem, dar este de preferat să fie respectate. De regulă numele atribuite sunt lungi, încep cu literă mică, iar în cadrul numelor apar litere mari la începuturile cuvintelor care le compun. De multe ori, când este vorba de o singură variabilă de un anumit tip, numele ei este numele tipului, scris cu literă mică.

Tot ca şi convenţii, începuturile (prefixele) numelor de variabile au semnificaţie: b pentru BOOL, by pentru BYTE, c pentru char, dw pentru DWORD, fn pentru funcţie h pentru handle, i pentru int, lp pentru pointer lung, w pentru WORD etc.

## Aplicaţii consolă; un filtru

Cele mai simple aplicaţii care se pot scrie sub Windows sunt aplicaţiile consolă. Acestea sunt, în fapt, aplicaţii cu intrare şi ieşire standard în mod text, la fel ca şi la programele simple sub Unix. Spre exemplu, un program extrem de simplu este un filtru. Acesta citeşte linie cu linie de la intrarea standard şi dă la ieşire aceleaşi linii, scurtate la primele 10 caractere. Sursa Filtru.cpp este este prezentată în continuare.

#include <stdio.h>

#include <string.h>

int main(int c, char\* a[]) {

char l[128];

for (;;) {

if (gets(l)==NULL)

break;

if (strlen(l) > 10)

l[10]= 0;

printf("%s\r\n", l);

}

return 0;

}

Lansarea unui astfel de filtru se face, de asemenea, dintr-o fereastră Cmd, putându-se, la fel ca în Unix sau Dos, să se redirecteze intrarea şi ieşirea lui standard, astfel:

Filtru.exe <FisierIntrare >FisierIesire

## Acces la fişiere şi aşteptarea unor evenimente

Tabelul care urmează prezintă numele unor funcţii de lucru cu fişiere sub Windows. Pentru mai multe informatii legate de prototipurile acestor funcţii, se recomandă consultarea documentaţiei MSDN.

|  |  |
| --- | --- |
| **Nume funcţie** | **Rol** |
| CreateFile | Crearea unui fişier cu anumite atribute |
| OpenFile | Deschide un fişier deja creat |
| WriteFile | Scrie binar într-un fişier |
| ReadFile | Citeşte binar dintr-un fişier |
| CloseHandle | Inchide fişier, eveniment, timer etc. |

Win32 API oferă un set de funcţii de aşteptare pentru a permite unui program să îşi suspende temporar execuţia în aşteptarea unui eveniment. Funcţiile de aşteptare blochează execuţia programului până când criteriul specificat a fost îndeplinit. Tipul funcţiei de aşteptare determină criteriul utilizat. În timpul aşteptării procesul consumă foarte puţine resurse sistem, fiind vorba de o aşteptare pasivă – intrare în sleep. Tabelul de mai jos prezintă pe scurt rolurile principalelor funcţii de aşteptare.

|  |  |
| --- | --- |
| Funcţie wait | Descriere |
| WaitForSingleObject() | Aşteaptă după un anumit obiect ca acesta să ajungă în starea setat (de exemplu terminarea unui proces sau valoarea pozitiva a unui semafor). |
| WaitForSingleObjectEx() | Ca şi precedentul, plus aşteptarea a altor două evenimente: terminarea unei operaţii de intrare ieşire, sau sosirea unui apel asincron în threadul curent. |
| WaitForMultipleObjects() | Aşteaptă după o mulţime de obiecte. Ieşirea din aşteptare se poate face fie când unul dintre obiecte este setat, fie când toate obiectele ajung în starea setat. |
| WaitForMultipleObjectsEx() | Ca şi precedentul, plus aşteptarea celor două evenimente specificate în cazul funcţiei WaitForSingleObjectEx. |

Cea mai simplă dintre aceste funcţii şi cea mai des utilizată este WaitForSingleObject.

## Procese Windows

**Crearea unui proces** în Windows se face prin apelul funcţiei CreateProcess dintr-un alt proces. Funcţia are următorul prototip:

BOOL CreateProcess (LPCTSTR lpszImageName,

LPCTSTR lpszCommandLine,

LPSECURITY\_ATTRIBUTES lpsaProcess,

LPSECURITY\_ATTRIBUTES lpsaThread,

BOOL fInheritHandles,

DWORD fdwCreate,

LPVOID lpvEnvironment,

LPTSTR lpszCurDir,

LPSTARTUPINFO lpsiStartInfo,

LPPROCESS\_INFORMATION lppiProcInfo);

Atunci când se apelează funcţia CreateProcess, sistemul creează un spaţiu de adresare şi încarcă noul proces în acest spaţiu. După această operaţie, sistemul creează threadul primar pentru noul proces şi-l lansează în execuţie. Să vedem semnificaţia parametrilor funcţiei CreateProcess:

Pentru semnificaţia parametrilor se poate consulta MSDN.

**Terminarea unui proces.** Un proces poate fi terminat pe două căi: apelând din interior funcţia ExitProcess sau apelând din exterior funcţia TerminateProcess. Este preferabilă prima cale, cea de-a doua trebuie folosită doar pentru situaţii extreme. Prototipul celor două funcţii sunt:

VOID ExitProcess (UINT fuExitCode);

BOOL TerminateProcess (HANDLE hProcess, UINT fuExitCode);

Pentru detalii se poate consulta MSDN.

## Exemplu: rezumatul directorului curent

Pentru utilizare sub Windows, vom folosi ca intermediar un fisier de comenzi ls.bat, care contine o singura linie:

dir %1

Sursa execWin.cpp a programului este:

#include <windows.h>

#include <stdio.h>

int main() {

STARTUPINFO si = { sizeof(si) };

PROCESS\_INFORMATION pi;

// ls.bat contine linia: dir %1

printf("Procesul parinte %d va creea un fiu\n", GetCurrentProcessId());

// Rulati alternativ cu una dintre urmatoarele doua linii comentata:

BOOL b = CreateProcess("ls.bat", NULL, NULL, NULL,

// BOOL b = CreateProcess("ls.bat", "ls.bat \*.cpp", NULL, NULL,

FALSE, 0, NULL, NULL, &si, &pi);

WaitForSingleObject(pi.hProcess, INFINITE);

printf("Terminat procesul fiu %d creeat de parintele %d\n",

pi.dwProcessId, GetCurrentProcessId());

return 0;

}

## Exemplu: câte perechi de numere nenule au suma un număr par?

Problema, rezolvată şi sub Unix, este trivial de simplă, dar potrivită pentru a exemplifica utilizarea CreateProcess, WaitForSingleObject si ExitProcess.

Enuntul problemei: Se dau la linia de comanda n perechi de numere intregi. Programul va crea **n** procese fii, fiecare primind doua argumente consecutive din linia de comanda. Oricare dintre fii intoarce codul de retur:

* 0 daca perechea are suma para,
* 1 daca suma este impara,
* 2 daca unul dintre argumente este nul sau nenumeric.

Părintele aşteaptă terminarea fiilor şi va afişa rezultatul. In continuare vom implementa un program separat pentru procesul fiu. Sursa lui, paritateFiu.cpp este:

#include <stdio.h>

#include <windows.h>

main(int argc, char\* argv[]) {

int n1, n2;

n1 = atoi(argv[1]); // atoi intoarce 0

n2 = atoi(argv[2]); // si la nenumeric

if (n1 == 0 || n2 == 0) ExitProcess(2);

if ((n1 + n2) % 2 == 0) ExitProcess(0);

else ExitProcess(1);

}

Acesta va fi compilat cu:

gcc -o paritateFiu paritateFiu.cpp

Sursa paritate.cpp este:

#include <stdio.h>

#include <windows.h>

main(int argc, char\* argv[]) {

STARTUPINFO si = { sizeof(si) };

PROCESS\_INFORMATION pi[100];

char linieCom[1000];

int pare = 0, impare = 0, nenum = 0, i;

DWORD n1;

for (i = 1; i < argc-1; i += 2) {

strcpy(linieCom, "paritateFiu.exe ");

strcat(linieCom, argv[i]);

strcat(linieCom, " ");

strcat(linieCom, argv[i+1]);

BOOL b = CreateProcess("paritateFiu.exe", linieCom, NULL, NULL,

FALSE, 0, NULL, NULL, &si, &pi[i]);

}

// Parintele asteapta terminarile fiilor

for (i = 1; i < argc-1; i += 2) {

WaitForSingleObject(pi[i].hProcess, INFINITE);

GetExitCodeThread(pi[i].hThread, &n1);

switch (n1) {

case 0: pare++;break;

case 1: impare++;break;

default: nenum++;

}

}

printf("Pare %d, impare %d, nenumerice %d\n",pare, impare, nenum);

}

## Exemplu: capitalizarea mai multor fişiere text

Dorim să transformăm un fişier text într-un alt fisier text, cu acelaşi conţinut, dar în care toate cuvintele din el sa înceapa cu literă mare. Un astfel de program va fi apelat:

capitalizare fisierintrare fisieriesire

Ne propunem sa prelucram simultan mai multe astfel de fisiere. De aceea vom creea un proces master, care primeste la linia de comanda numele fisierelor al caror continut va fi capitalizat:

master fisier1 fisier2 - - - fisiern

Rezultatul va consta din fisierele:

fisier1.CAPIT, fisier2.CAPIT, - - - fisiern.CAPIT

Procesul master va crea **n** procese fii, iar fiecare fiu i va lansa prin CreateProcess programul:

capitalizare fisi fisi.CAPIT

Sursa capitalizare.cpp este:

#include <stdio.h>

#include <windows.h>

#include<ctype.h>

#define MAXLINIE 100

main(int argc, char\* argv[]) {

FILE \*fi, \*fo;

char linie[MAXLINIE], \*p;

fi = fopen(argv[1], "r");

fo = fopen(argv[2], "w");

if (fi == NULL && fo == NULL) ExitProcess(1);

for ( ; ; ) {

p = fgets(linie, MAXLINIE, fi);

linie[MAXLINIE-1] = '\0';

if (p == NULL) break;

if (strlen(linie) == 0) continue;

linie[0] = toupper(linie[0]);

for (p = linie; ; ) {

p = strstr(p, " ");

if (p == NULL) break;

p++;

if (\*p == '\n') break;

\*p = toupper(\*p);

}

fprintf(fo, "%s", linie);

}

fclose(fo);

fclose(fi);

}

Programul primeste la linia de comanda numele celor doua fisiere. Se deschid aceste fisiere si se citeste fisierul de intrare linie cu linie. Cu ajutorul pointerului p, se parcurge linia curenta si se cauta pe rand cate un spatiu, dar care sa nu fie ultimul caracter din linie. Urmatorul caracter este apoi transformat in litera mare (toupper face aceasta transformare numai daca caracterul este efectiv o litera mica).

Sursa master.cpp este:

#include <stdio.h>

#include <windows.h>

main(int argc, char\* argv[]) {

STARTUPINFO si = { sizeof(si) };

PROCESS\_INFORMATION pi[100];

int i;

char nume[200];

for (i=1; argv[i]; i++) {

strcpy(nume, "capitalizare ");

strcat(nume, argv[i]);

strcat(nume, " ");

strcat(nume, argv[i]);

strcat(nume, ".CAPIT"); // fabricat numele iesirii

// incarcat programul de capitalizare

BOOL b = CreateProcess("capitalizare.exe", nume, NULL, NULL,

FALSE, 0, NULL, NULL, &si, &pi[i]);

}

printf("Lansat simultan %d procese de capitalizare\n",i-1);

}

Se parcurg argumentele liniei de comanda si pentru fiecare dintre ele se creeaza un proces fiu. In tabloul nume se construieste numele fisierului de iesire. Apoi se incarca programul capitalizare cu cele doua nume de fisiere date "la linia de comanda".

Cele doua programe se compileaza:

gcc -o capitalizare capitalizare.c

gcc -o master master.c

Lansarea se face:

master fis1 fis2 - - - fisn

## Pipe sub Windows

În Windows, ca şi în Unix, există două posibilităţi de a folosi pipe în IPC. O primă variantă este *pipe* *anonime,* care se pot folosi numai pentru comunicarea între procese de pe aceeaşi maşină. A doua variantă este *pipe cu nume,* folosite pentru comunicarea între procese ce operează nu neapărat pe aceeaşi maşină Windows.

Un **pipe anonim** poate fi folosit, ca şi pipe-ul de sub Unix, pentru comunicarea între procese descendente din creatorul pipe-ului. In urma creării, procesul creator obţine doi descriptori - handle - unul de citire şi altul de scriere. Procesul creator poate trimite fiilor (nepoţilor etc.) handle-urile pipe-ului, în momentul creării proceselor fii prin apeluri ale funcţiei CreateProces. Pentru ca fiul să moştenească handle-ul la pipe, părintele trebuie să seteze parametrul fInheritedHandle din apelul CreateProces, la valoarea TRUE.

Un pipe fără nume se creează:

BOOL CreatePipe (PHANDLE phRead,PHANDLE phWrite,

LPSECURITY\_ATTRIBUTES lpsa, DWORD cbPipe);

Funcţia întoarce TRUE în caz de succes sau FALSE la eşec.

Pipe cu nume este un mecanism de comunicare între două sisteme diferite, ambele fiind operaţionale pe platforme Windows

In figura următoare sunt prezentate succesiunile apelurilor sistem, atât pentru server, cât şi pentru client. Cititorul poate uşor observa particularizările necesare pentru comunicarea prin pipe anonim.

Crearea unui pipe cu nume se face prin apelul sistem CreateNamedPipe, cu prototipul:



HANDLE CreateNamedPipe(LPSTR numePipe,

DWORD optiuniModOpen,

DWORD optiuniModPipe,

DWORD nMaxInstances,

DWORD lungBufOut,

DWORD lungBufIn,

DWORD timeOut,

LPSECURITY\_ATTRIBUTES lpsa)

numePipe este un string prin care se indică numele pipe-ului. Convenţiile Microsoft de specificare a acestor nume impun două sintaxe, una pentru pipe local şi alta pentru pipe de pe o altă maşină. Aceste specificări sunt:

\\.\**PIPE**\numePipePeMAsina

\\adresaMasina\**PIPE**\numePipePeMasina

Pentru alte detalii, vezi MSDN.

După crearea unui pipe cu nume, serverul apelează:

ConnectNamedPipe(HANDLE hNamedPipe, LPOVERLAPPED lpo)

Primul handle este cel întors de crearea pipe. Al doilea parametru, de regulă NULL, indică faptul că se aşteaptă la conectare până când un client se conectează efectiv la pipe. (A se compara această regulă cu cea similară de la FIFO de sub Unix).

La fel ca şi la pipe anonime, se folosesc apelurile ReadFile şi WriteFile pentru schimbul cu pipe.

Serverul îşi încheie activitatea apelând:

DisconnectNamedPipe(HANDLE hNamedPipe);

CloseHandle (HANDLE hNamedPipe);

Pentru client, conectarea la un pipe cu nume presupune un apel sistem CreateFile:

In cazul creării unui pipe cu nume, numeFisier reprezintă numele pipe-ului, cu sintaxa specificată mai sus, la apelul CreateNamedPipe.

## Threaduri MS-Windows; generalităţi

Tabelul de mai jos prezintă comparativ, tipurile de date, variabilele şi principalele funcţii care operează cu threaduri:

|  |  |  |
| --- | --- | --- |
| **API elems. \OS** | **Linux** | **MS Windows** |
| **Headers** | #include<stdio.h>  #include<pthread.h>  #include<stdlib.h>  #include <semaphore.h> | #include <windows.h>  #include <stdlib.h>  #include <stdio.h>  #include <math.h> |
| **Libraries** | -pthread -lm |  |
| **Data Types** | pthread\_t  pthread\_mutex\_t  pthread\_cond\_t  pthread\_rwlock\_t  sem\_t | HANDLE  CRITICAL\_SECTION  CONDITION\_VARIABLE  SRWLOCK  HANDLE |
| **Threads** | pthread\_create  pthread\_join | CreateThread  WaitForSingleObject |
| **Function Decl** | void\* worker(void\* a) | DWORD WINAPI worker(LPVOID a) |
| **Mutexes** | pthread\_mutex\_init  pthread\_mutex\_lock  pthread\_mutex\_unlock  pthread\_mutex\_destroy | InitializeCriticalSection  EnterCriticalSection  LeaveCriticalSection  DeleteCriticalSection |
| **Conditional**  **Variables** | pthread\_cond\_init  pthread\_cond\_wait  pthread\_cond\_signal  pthread\_cond\_destroy | InitializeConditionVariable  SleepConditionVariableCS  WakeConditionVariable  !Trebuie compilate cu Visual Studio incepand cu Vista, Windows 7 si mai recente! |
| **Read/Write**  **Locks** | pthread\_rwlock\_init  pthread\_rwlock\_wrlock  pthread\_rwlock\_rdlock  pthread\_rwlock\_unlock  pthread\_rwlock\_destroy | InitializeSRWLock  AcquireSRWLockExclusive  AcquireSRWLockShared  ReleaseSRWLockExclusive  AcquireSRWLockShared    !Trebuie compilate cu Visual Studio incepand cu Vista, Windows 7 si mai recente! |
| **Semaphores** | sem\_init  sem\_wait  sem\_post  sem\_destroy | CreateSemaphore  WaitForSingleObject  ReleaseSemaphore  CloseHandle |

Sub Windows NT, *threadul* este cea mai mică entitate executabilă la nivel nucleu. Fiecare proces conţine unul sau mai multe thread-uri. In momentul creării procesului, odată cu el se crează *threadul primar* al acestuia. Threadul primar poate crea la rândul său alte thread-uri cu care va partaja spaţiul de adrese al procesului comun. De asemenea, ele mai partajează şi alte resurse sistem: descriptori de fişiere, etc.

## Operaţii asupra threadurilor: creare, terminare

Prototipul funcţiei de **creare** este:

HANDLE CreateThread(

LPSECURITY\_ATTRIBUTES lpThreadAttributes,

DWORD dwStackSize,

LPTHREAD\_START\_ROUTINE lpStartAddress,

LPVOID lpParameter,

DWORD dwCreationFlags,

LPDWORD lpThreadId

);

lpStartAddress - pointer la funcţia ce dirijează threadul

lpParameter - argumentul funcţiei

lpThreadId - pointer la identificatorul threadului

La crearea threadului, este generat un descriptor care identifică în mod unic threadul în sistem. După creare, se lansează în execuţie funcţia specificată prin parametrul lpStartAddress. Această funcţie are parametrii specificaţi prin lpParameter şi întoarce o valoare de tip DWORD. Pentru a determina valoarea întoarsă de această funcţie, se poate folosi funcţia GetExistCodeThread().

**Terminarea unui thread.** Un thread îşi încheie execuţia în următoarele condiţii:

• la ieşirea din procedura asociată threadului.

• la apelul funcţiilor ExitProcess(), ExitThread() apelate din threadul curent.

• dacă se apelează ExitProcess() sau TerminateThread()din alte procese, cu argument handler-ul threadului care urmează a fi distrus sau din alte thread-uri, folosind, de asemenea, funcţia TerminateThread().

Prototipurile unora dintre funcţiile de terminare a unui threa sunt:

void ExitThread(UINT exitcode);

BOOL TerminateThread(HANDLE hThread, DWORD exitcode);

BOOL GetExitCodeThread(HANDLE hThread, LPDWORD exitcode);

Parametrul hThread identifică threadul care se va termina.

Apelul ExitThread provoacă terminarea threadului curent cu întoarcerea codului de ieşire exitcode. După apelul funcţiei, stiva asociată threadului este eliberată, iar starea obiectului thread, privit ca eveniment, devine semnalată.

## Instrumente standard de sincronizare

Mecanismele de comunicare şi sincronizare între thread-uri sunt furnizate de interfaţa Win32API, care furnizează primitive de lucru cu *evenimente, semafoare, variabile mutex, secţiuni critice*. Aceste obiecte de sincronizare au: *semnalat* şi  *nesemnalat*. Starea semnalat presupune de obicei îndeplinirea unei condiţii şi semnalarea acestui fapt unor thread-uri interesate. Pentru aşteptarea semnalării (starea semnalat) se poate folosi funcţia WaitForSingleObject

**Variabilele** **mutexError! Bookmark not defined.** permit implementarea accesului exclusiv la o resursă partajată între mai multe thread-uri. Semantica obiectelor de sincronizare mutex este similară cu cea întâlnită la implementarea thread-urilor de pe platformele Unix. ***Crearea*** unei astfel de variabile se face cu funcţia:

HANDLE CreateMutex(

LPSECURITY\_ATTRIBUTES lpMutexAttributes,

BOOL binitialOwner,

LPCTSTR lpName

);

***Ocuparea*** unei variabile mutex se face prin funcţiile de aşteptare, de exemplu apelul WaitForSingleObject(g\_hMutex, INFINITE) se va termina doar când starea variabilei mutex indentificată prin handler-ul g\_hMutex devine semnalată.

***Eliberarea*** unei variabile mutex se realizeaza cu funcţia:

BOOL ReleaseMutex(HANDLE hMutex);

***Distrugerea*** unei variabile mutex se face fie prin invocând CloseHandle(). Dacă acest apel lipseşte, variabilele mutex sunt eliminate de sistem.

**Semafoare fără nume.** Pentru sincronizarea thread**Error! Bookmark not defined.**-urilor din cadrul aceluiaşi proces, este de preferată utilizarea semafoarelor fără nume, deoarece nucleul sistemului este scutit de gestiunea lor. Crearea unui astfel de semafor trebuie făcută folosind apelul CreateSemaphore, (vezi MSDN). Pentru a fi semafor anonim trebuie ca ultimul parametru, pointer la numele semaforului, să aibă valoarea NULL.

Valoarea semaforului poate fi mărită cu o cantitate pozitivă apelând funcţia ReleaseSemaphore()Ea are ca prim argument handle-ul semaforului şi cantitatea cu care se măreşte valoarea.

Aşteptarea la semafor se face folosind funcţiile de aşteptare descrise mai sus.

**Secţiuni critice**. O variabilă de tip secţiune critică se declară astfel:

CRITICAL\_SECTION numeSectiuneCritica;

Utilizarea secţiunii critice se face astfel:

EnterCriticalSection(&numeSectiuneCritica);

- - - Corpul sectiunii critice - - -

LeaveCriticalSection(&numeSectiuneCritica);

## Exemplu: m trenuri trec pe n linii între A şi B

|  |
| --- |
| **TrenuriMutexCond Windows** |
| #include <windows.h>  #include <winbase.h>  #include <stdlib.h>  #include <stdio.h>  #define N 5  #define M 13  #define SLEEP 4  CRITICAL\_SECTION mut, mutcond;  CONDITION\_VARIABLE cond; // A fost adaugat la Vista si da eroare de compilare la windows mai vechi!  int linie[N], tren[M];  DWORD tid[M];  HANDLE doneEv[M];  int liniilibere;  //rutina unui thread  DWORD WINAPI trece(LPVOID tren) {  int i, s, t, l;  t = \*(int\*)tren;  s = 1 + rand() % SLEEP; // Modificati timpii de stationare    EnterCriticalSection(&mutcond);  for ( ; liniilibere == 0; ) {  SleepConditionVariableCS(&cond, &mutcond, INFINITE);  }  LeaveCriticalSection(&mutcond);    EnterCriticalSection(&mut);  for (l = 0; l < N; l++)  if (linie[l] == -1) break;  linie[l] = t;  liniilibere--;  printf("Trenul %d pe linia %d pentru %d secunde. Trenuri in gara:", t, l, s);  for (i=0; i< N; i++)  if (linie[i] != -1)  printf(" %d",linie[i]);  printf(".\n");  fflush(stdout);  LeaveCriticalSection(&mut);  Sleep(s); // Modificati timpii de sleep  EnterCriticalSection(&mut);  linie[l] = -1;  liniilibere++;  LeaveCriticalSection(&mut);    EnterCriticalSection(&mutcond);  WakeConditionVariable(&cond);  LeaveCriticalSection(&mutcond);  SetEvent(doneEv[t]);  }  //main  main(int argc, char\* argv[]) {  int i;  InitializeCriticalSection(&mut);  InitializeCriticalSection(&mutcond);  InitializeConditionVariable(&cond);  liniilibere = N;  for (i = 0; i < N; linie[i] = -1, i++);  for (i=0; i < M; tren[i] = i, i++);  for (i=0; i < M; i++) doneEv[i] = CreateEvent(0, FALSE, FALSE, 0);    // ce credeti despre ultimul parametru &i?  for (i=0; i < M; i++) CreateThread(NULL, 16384, (LPTHREAD\_START\_ROUTINE)trece,  (LPVOID)&tren[i], 0, &tid[i]);  for (i=0; i < M; i++) WaitForSingleObject(doneEv[i],INFINITE);    DeleteCriticalSection(&mut);  DeleteCriticalSection(&mutcond);  DeleteConditionVariable(&cond);  } |

Contents

[0. Limbajul C-ansi: pointeri, tablouri, IO 1](#_Toc33428043)

[0.1. Pointeri 1](#_Toc33428044)

[0.1.1. Aritmetica de pointeri, echivalenţa pointeri - tablouri 1](#_Toc33428045)

[0.1.2. Citirea unor linii si ordonarea lor alfabetică, varianta 1 1](#_Toc33428046)

[0.1.3. Citirea unor linii si ordonarea lor alfabetică, varianta 2 2](#_Toc33428047)

[0.1.4. Citirea unor linii si ordonarea lor alfabetică, varianta Go 3](#_Toc33428048)

[0.1.5. Citirea unor linii si ordonarea lor alfabetică, varianta Python 3](#_Toc33428049)

[0.2. Tablouri C alocare statică şi dinamică 4](#_Toc33428050)

[0.2.1. Tablouri bidimensionale statice 4](#_Toc33428051)

[0.2.2. Tablouri bidimensionale dinamice 5](#_Toc33428052)

[0.2.3. Generalizări: tablouri dinamice nerectangulare şi / sau multidimensionale 6](#_Toc33428053)

[0.3. Fişiere text şi fişiere binare 6](#_Toc33428054)

[0.4. Operaţii IO în C 7](#_Toc33428055)

[0.4.1. Principalele funcţii 7](#_Toc33428056)

[0.4.2. Interclasarea a n fişiere text ordonate alfabetic 7](#_Toc33428057)

[0.4.3. Interclasarea a n fişiere text ordonate alfabetic - Go 8](#_Toc33428058)

[0.4.4. Interclasarea a n fişiere text ordonate alfabetic - Python 9](#_Toc33428059)

[0.4.5. Oglindirea conţinutului unui fişier: soluţia 1 10](#_Toc33428060)

[0.4.6. Oglindirea conţinutului unui fişier: soluţia 1 Go 10](#_Toc33428061)

[0.4.7. Oglindirea conţinutului unui fişier: soluţia 1 Python 11](#_Toc33428062)

[0.4.8. Oglindirea conţinutului unui fişier: soluţia buffer 11](#_Toc33428063)

[0.4.9. Oglindirea conţinutului unui fişier: soluţia buffer Go 12](#_Toc33428064)

[0.4.10. Oglindirea conţinutului unui fişier: soluţia buffer Python 13](#_Toc33428065)

[0.5. Manipularea fişierelor în sisteme de fişiere 14](#_Toc33428066)

[0.5.1. Principalele prototipuri de funcţii 14](#_Toc33428067)

[0.5.2. Parcurgerea recursivă a fişierelor dintr-un director şi descendenţi 15](#_Toc33428068)

[0.5.3. Parcurgerea recursivă a fişierelor dintr-un director şi descendenţi - Go 16](#_Toc33428069)

[0.5.4. Parcurgerea recursivă a fişierelor dintr-un director şi descendenţi - Python 17](#_Toc33428070)

[0.6. Probleme propuse 18](#_Toc33428071)

[1. Comenzi, expresii regulare, filtrele grep, sed, awk 19](#_Toc33428072)

[1.1. Comenzi 19](#_Toc33428073)

[1.2. Deosebiri formale Unix Windows: 20](#_Toc33428074)

[1.3. Expresii regulare: definire şi exemple 20](#_Toc33428075)

[1.4. Clasificarea comenzilor; comenzi filtru 21](#_Toc33428076)

[1.4.1. Clasificarea comenzilor Unix 22](#_Toc33428077)

[1.4.2. Filtrul grep 22](#_Toc33428078)

[1.4.3. Filtrul sed (Stream EDitor) 23](#_Toc33428079)

[1.4.4. Filtrul sort 25](#_Toc33428080)

[1.4.5. Filtrul uniq 25](#_Toc33428081)

[1.4.6. Filtrul cut 25](#_Toc33428082)

[1.5. awk; programarea în awk 26](#_Toc33428083)

[1.5.1. Apelul şi definirea programelor awk 26](#_Toc33428084)

[1.5.2. Ilustrare moduri de apel: numararea liniilor, cuvintelor si caracterelor 28](#_Toc33428085)

[1.5.3. Câteva exemple simple 28](#_Toc33428086)

[1.5.4. Afişarea primului cuvânt din fiecare linie 29](#_Toc33428087)

[1.5.5. Afişarea liniilor care au un anumit ultim cuvânt. 29](#_Toc33428088)

[1.5.6. Să se afişeze liniile mai lungi de 5 caractere 30](#_Toc33428089)

[1.5.7. Prelucrari asupra unui fisier cu campuri fixe. 30](#_Toc33428090)

[1.5.8. Rearanjarea cuvintelor din liniile unui fisier 32](#_Toc33428091)

[1.6. Probleme propuse 32](#_Toc33428092)

[2. Programare shell 33](#_Toc33428093)

[2.1. Ce este un script (fişier de comenzi) shell 33](#_Toc33428094)

[2.2. Structuri de control si comenzi folosite în programarea shell 34](#_Toc33428095)

[2.3. Exemple de scripturi shell 34](#_Toc33428096)

[2.3.1. Fişiere generice 34](#_Toc33428097)

[2.3.2. Numărul total de linii de cod C din directorul dat ca parametru care nu sunt goale: 34](#_Toc33428098)

[2.3.3. Un exemplu de if 35](#_Toc33428099)

[2.3.4. Citiri de la consolă şi test de existenţă / citire a fişierelor 35](#_Toc33428100)

[2.3.5. Monitorizarea stării unui director şi anunţul modificării lui 35](#_Toc33428101)

[2.3.6. Anunţarea userilor cu directoare prea mari 36](#_Toc33428102)

[2.3.7. Anunţarea userilor cu prea multe procese 36](#_Toc33428103)

[2.3.8. Distrugerea proceselor prea vechi 36](#_Toc33428104)

[2.3.9. Anunţul încărcării exagerate a procesorului şi / sau a memoriei 37](#_Toc33428105)

[2.3.10. Verificare perechi fisier - lungime 37](#_Toc33428106)

[2.3.11. Numărul mediu de linii: 37](#_Toc33428107)

[2.3.12. Supravegherea continutului unui grup de directoare 38](#_Toc33428108)

[2.3.13. Selectare numai cuvinte din litere mici 38](#_Toc33428109)

[2.3.14. Primul cuvant de cel putin 5 caractere 39](#_Toc33428110)

[2.3.15. Lista fişierelor cu anumite drepturi: 39](#_Toc33428111)

[2.3.16. Primele linii din fişiere cu cuvinte lungi: 40](#_Toc33428112)

[2.3.17. Redenumirea fişierelor de un anumit tip si numararea aparitiilor unui cuvant 40](#_Toc33428113)

[2.3.18. Aparitia unor cuvinte in linii din fisiere 41](#_Toc33428114)

[2.3.19. Reunirea fişierelor text 41](#_Toc33428115)

[2.3.20. Analizaţi textul sursă al unui script 43](#_Toc33428116)

[2.4. Probleme propuse 43](#_Toc33428117)

[3. Procese Unix (în C): fork, exec, exit, wait system, signals 45](#_Toc33428118)

[3.1. Standardul POSIX de gestiune a erorilor în apeluri sistem: errno 45](#_Toc33428119)

[3.2. Principalele apeluri sistem Unix care operează cu procese 45](#_Toc33428120)

[3.3. Exemple de lucrul cu procese 46](#_Toc33428121)

[3.3.1. Utilizări simple fork exit, wait 46](#_Toc33428122)

[3.3.2. Utilizări simple execl, execlp. execv, system 49](#_Toc33428123)

[3.3.3. Un program care compileaza şi rulează alt program 49](#_Toc33428124)

[3.3.4. Capitalizarea cuvintelor dintr-o listă de fişiere text 51](#_Toc33428125)

[3.3.5. Câte perechi de argumente au suma un număr par? 52](#_Toc33428126)

[3.4. Semnale Unix; exemple de utilizare 53](#_Toc33428127)

[3.4.1. Evitarea proceselor zombie 53](#_Toc33428128)

[3.4.2. Schema client / server: adormire si deşteptare 54](#_Toc33428129)

[3.4.3. Aflarea unor informaţii de stare 54](#_Toc33428130)

[3.4.4. Tastarea unei linii în timp limitat 55](#_Toc33428131)

[3.4.5. Blocarea tastaturii 55](#_Toc33428132)

[3.5. Probleme propuse 56](#_Toc33428133)

[4. Comunicarea între procese Unix: pipe, FIFO, popen, dup2 57](#_Toc33428134)

[4.1. Principalele apeluri sistem de comunicare între procese 57](#_Toc33428135)

[4.2. Analizaţi textul sursă 57](#_Toc33428136)

[4.3. Utilizări simple pipe şi FIFO 58](#_Toc33428137)

[4.4. Simulare sh pentru who | sort şi who | sort | cat (dup2) 60](#_Toc33428138)

[4.5. Paradigma client / server; exemple 61](#_Toc33428139)

[4.6. Exemple de utilizare popen 64](#_Toc33428140)

[4.7. Probleme propuse 65](#_Toc33428141)

[5. Pthreads; sincronizări cu mutex, cond, barrier 67](#_Toc33428142)

[5.1. Principalele tipuri de date şi funcţii de lucru cu threaduri 67](#_Toc33428143)

[5.2. Capitalizarea cuvintelor dintr-o listă de fişiere text 67](#_Toc33428144)

[5.3. Capitalizarea cuvintelor dintr-o listă de fişiere text - soluţia go 68](#_Toc33428145)

[5.4. Capitalizarea cuvintelor dintr-o listă de fişiere text - soluţia python 69](#_Toc33428146)

[5.5. De ce sunt necesare variabilele mutex? 70](#_Toc33428147)

[5.6. Câte perechi de argumente au suma număr par? 71](#_Toc33428148)

[5.7. Evaluarea expresie aritmetică operator / thread şi paralelizare maximă 72](#_Toc33428149)

[5.8. De ce sunt necesare variabilele condiţionale? 73](#_Toc33428150)

[5.9. Bariera - exemplu 75](#_Toc33428151)

[5.10. Adunarea în paralel a n numere 76](#_Toc33428152)

[6. Sincronizarea threadurilor; soluţii ale unor probleme celebre 79](#_Toc33428153)

[6.1. Propunere de rezolvare a două probleme simple 79](#_Toc33428154)

[6.2. Intre A şi B sunt n linii prin care trec m trenuri, m > n 79](#_Toc33428155)

[6.3. Intre A şi B sunt n linii prin care trec m trenuri, m > n; soluţia go 82](#_Toc33428156)

[6.4. Intre A şi B sunt n linii prin care trec m trenuri, m > n; soluţia python 83](#_Toc33428157)

[6.5. Problema frizerului somnoros 84](#_Toc33428158)

[6.6. Problema cinei filosofilor 85](#_Toc33428159)

[6.7. Problema producătorilor şi a consumatorilor 87](#_Toc33428160)

[6.8. Problema cititorilor şi a scriitorilor 91](#_Toc33428161)

[6.9. Utilizarea altor platforme de threaduri 96](#_Toc33428162)

[6.10. Probleme propuse 96](#_Toc33428163)

[7. API Unix vs API Windows 97](#_Toc33428164)

[7.1. Deosebiri formale Unix - Windows 97](#_Toc33428165)

[7.2. Fişiere de comenzi de tip bat (MS-Windows) 97](#_Toc33428166)

[7.3. Exemplu de fişier bat: concatenare fişiere 98](#_Toc33428167)

[7.4. Particularităţi C / C++ pentru MS-Windows 99](#_Toc33428168)

[7.5. Aplicaţii consolă; un filtru 100](#_Toc33428169)

[7.6. Acces la fişiere şi aşteptarea unor evenimente 100](#_Toc33428170)

[7.7. Procese Windows 101](#_Toc33428171)

[7.8. Exemplu: rezumatul directorului curent 101](#_Toc33428172)

[7.9. Exemplu: câte perechi de numere nenule au suma un număr par? 102](#_Toc33428173)

[7.10. Exemplu: capitalizarea mai multor fişiere text 103](#_Toc33428174)

[7.11. Pipe sub Windows 104](#_Toc33428175)

[7.12. Threaduri MS-Windows; generalităţi 106](#_Toc33428176)

[7.13. Operaţii asupra threadurilor: creare, terminare 107](#_Toc33428177)

[7.14. Instrumente standard de sincronizare 108](#_Toc33428178)

[7.15. Exemplu: m trenuri trec pe n linii între A şi B 109](#_Toc33428179)