# 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

1. Sa se scrie un program care va inmulti doua matrici de dimensiuni mari folosind un numar de n threaduri, n fiind dat ca parametru. Fiecare element al matricei rezultat va fi calculat de un anumit thread. Spre exemplu, daca matricea rezultat are 3 linii si 5 coloane iar n=4, elementul (1,1) al matricei rezultat va fi calculat de threadul 1, (1,2) de threadul 2, (1,3) de threadul 3, (1,4) de threadul 4, (1,5) de threadul 1, (2,1) de threadul 2, (2, 2) de threadul 3 etc. Programul va afisa timpul in care se calculeaza matricea rezultat. Se vor compara rezultatele obtinute ruland programul utilizand un numar diferite de threaduri (1, 2, 4, 8). Problema se va rula pentru matricii de dimensiuni mari (spre exemplu de 1000x1000) cu elemente generate aleator.

2. Sa se scrie un program care folosind threaduri simuleaza decolarea si aterizarea avioanelor pe un aeroport. "Din senin" apar threaduri (avioane) create de un thread daemon, avioane care trebuie sa aterizeze pe o pista unica. La crearea fiecarui thread care reprezinta un avion, se stabilieste aleator pentru acesta o cantitate de combustibil ramasa si o ora la care trebuie sa decoleze. Un thread daemon va coordona aterizarile si decolarile pe pista unica, astfel incat nici un avion aflat in aer sa nu ramana fara combustibil, iar intarzierea decolarii avioanelor de la sol sa fie minima.Se vor folosi, daca este cazul, variabile mutex si / sau variabile conditionale.

3. Sa se scrie un program care numara, folosind threaduri, numarul de cuvinte 'the' din mai multe fisiere date ca parametri. Programul va afisa la final timpul total de executie, timpul de executie per fisier si topul celor mai harnice trei threaduri (timp de executie / dimensiune fisier analizat). Problema va fi implementata si fara threaduri, afisandu-se de asemenea timpul de executie.

Observatii: Fisierele text date ca parametrii trebuie sa aiba o dimensiune relativ mare. Pentru o rezolvare 'cat mai placuta' a problemei se recomanda utilizarea ca versiunilor text in limba engleza a diferitor romane clasice din literatura universala disponibile la adresa: www.gutenberg.org.

4. Sa se scrie un program care sorteaza un sir folosind threaduri. Programul principal creeaza un thread T1 a carui sarcina este sortarea intregului sir. Acest thread, creaza la randul sau doua threaduri T2 si T3 a caror sarcina este sortarea celor doua jumatati ale sirului. Dupa ce threadurile T2 si T3 termina de sortat cele doua jumatati, threadul T1 interclaseaza jumatatile sirului pentru a obtine varianta sortata. Pentru sortarea celor doua jumatati ale sirului threadurile T2 si T3 vor aplica un mecanism similar. Programul va fi rulat pentru un sir cu cateva zeci de mii de elemente. La sfarsit va fi afisat timpul in care a fost sortat intregul sir.Se vor folosi, daca este cazul, variabile mutex si / sau variabile conditionale.

5. Sa se scrie un program care genereaza un labirint sub forma unei matrici de mari dimensiuni ce contine numai 0 si 1 (0 liber, 1 zid). Folosind threaduri sa se incerce rezolvarea labirintului. Pornind din centrul labirintului, un numar de unul, doua, trei sau patru threaduri (dupa caz) vor porni in fiecare directie incercand sa iasa din labirint. Cand ajunge la o intersectie, threadul curent va crea alte threaduri care vor porni pe caile accesibile din intersectie, threadul curent poate continua si el pe o cale accesibila. Se va tipari frecvent matricea labirintului, fiecare thread lasand "o urma" pe unde a trecut (spre exemplu id-ul sau). Se vor folosi, daca este cazul, variabile mutex si / sau variabile conditionale.

6. Sa se scrie un program care cauta, folosind n threaduri, fisierele cu o anumita extensie dintr-un anumit director si din toate subdirectoarele sale. Programul primeste ca parametru numarul n de threaduri, directorul si extensia. Primul thread "cauta" doar la primul nivel in directorul respectiv, afiseaza eventualele fisiere gasite cu extensia respectiva si pune intr-o lista FIFO toate subdirectoarele intalnite. Celelalte threaduri (ca si primul thread dupa ce termina cu directorul dat ca parametru) extrag pe rand cat un subdirector din lista si il proceseaza mai departe in aceeasi maniera. Programul se termina cand lista de directoare este vida. Se vor folosi, daca este cazul, variabile mutex si / sau variabile conditionale.

7. Sa se scrie un program care simuleaza o agentie de pariuri. Programul va opera cu trei tipuri de threaduri. Primul tip reprezinta thredul deamon ce reprezinta agentia de pariuri. Al doilea tip reprezinta threadurile care reprezinta meciurile dintre doua echipe pe care le ofera spre pariere agentia. Toate threadurile de al doilea tip vor rula aceeasi perioada de timp, spre exemplu 90 de secunde). Ultimul tip o reprezinta pariorii, care vor paria "live" pe rezultatele finale ale meciurilor. Threadul daemon va oferi pariorilor cote "live" de castig. Spre exemplu, daca in secunda 80 scorul este 3-0 pentru prima echipa, agentia va oferi o cota de 1.05 pentru acest rezultat final. Pariorii pot paria oricand pe acest rezultat final, insa nu isi pot modifca pariul. Rezultatele meciurilor se modifica aleator pana la final, pariorul putand castiga de 1.05 ori suma pariata sau pierde toata suma daca rezultatul se schimba, de exemplu devine 3-4. Threadurile ce reprezinta pariorii pleaca initial cu o suma pe care o detin, fiind scoase din joc daca raman fara bani. Dupa mai multe etape, se va fisa topul pariorilor in functie de castig. Se vor folosi, daca este cazul, variabile mutex si / sau variabile conditionale.