Probleme de sincronizare a threadurilor: cond, rwlock, sem

1. **Contents**

2. Principalele tipuri de date şi funcţii de lucru cu threaduri (reluare)

Prezentăm din nou tabelul din seminarul precedent care prezintă principalele fişiere header, tipuri de date şi funcţii care lucrează cu threaduri:

Elemente fundamentale pthreads		**Instrumente de sincronizare şi coordonare**	

Fişere header	`<pthread.h>`
	`<stdlib.h>`
	`<semaphore.h>`
		Variabile mutex	`pthread_mutex_init`
			`pthread_mutex_lock`
			`pthread_mutex_unlock`
			`pthread_mutex_destroy`

Specificare biblioteci	`-pthread`	**Variabile condiţionale**	`pthread_cond_init`
			`pthread_cond_wait`
			`pthread_cond_signal`
			`pthread_cond_destroy`

Tipuri de date	`pthread_t`
	`pthread_mutex_t`
	`pthread_cond_t`
	`pthread_rwlock_t`
	`sem_t`
		Variabile reader/writer	`pthread_rwlock_init`
			`pthread_rwlock_wrlock`
			`pthread_rwlock_rdlock`
			`pthread_rwlock_unlock`

Funcţii de creare thread şi aşteptare terminare	`pthread_create`
	`pthread_join`
		Semafoare	`sem_init`
			`sem_wait`
			`sem_post`
			`sem_destroy`

Funcţia de descriere a acţiunii threadului		`void* worker(void* a)`	

3. Intre două gări A şi B, m trenuri trec simultan pe n linii, cu m > n

In gara A intră simultan maximum **m** trenuri care vor să ajungă în gara B. Intre A şi B există simultan **n** linii, **m > n**, dar în gară se pot afla simultan doar **n** trenuri. 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. Soluiile, una folosind variabile condiţionale, cealaltă folosind semafoare, sunt prezentate în tabelul următor.

trenuriMutCond.c	**trenuriSem.c**
`#include <stdlib.h>`	`#include <semaphore.h>`

```
#include <pthread.h>
#include <stdio.h>
#include <unistd.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];
pthread_t tid[M];
int liniilibere;
time_t start;

//rutina unui thread
void* trece(void* tren) {
    int t, l;
    t = *(int*)tren;

    sleep(1 + rand()%SLEEP); // Inainte de
==> A

    pthread_mutex_lock(&mutcond);
    printf("Moment %lu tren %d: ==> A\n",
time(NULL)-start, t);
    for ( ; liniilibere == 0; )
pthread_cond_wait(&cond, &mutcond);
    for (l = 0; l < N; l++) if (linie[l]
== -1) break;
    linie[l] = t; // In A ocupa linia
    liniilibere--;
    printf("\tMoment %lu tren %d: A ==> B
linia %d\n",time(NULL)-start, t, l);
    pthread_mutex_unlock(&mutcond);

    sleep(1 + rand()%SLEEP); // Trece
trenul  A ==> B

    pthread_mutex_lock(&mutcond);
    printf("\t \tMoment %lu tren %d: B
==>, liber linia %d\n", time(NULL)-start,
t, l);
    linie[l] = -1;
    liniilibere++;
    pthread_cond_signal(&cond); // In B
elibereaza linia
    pthread_mutex_unlock(&mutcond);
}



//main
int main(int argc, char* argv[]) {
    int i;
    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++);
    start = time(NULL);
    // ce credeti despre ultimul parametru
&i?
    for (i=0; i < M; i++)
pthread_create(&tid[i], NULL, trece,
```

```
#include <pthread.h>
#include <stdlib.h>
#include <stdio.h>
#include <unistd.h>
#include <time.h>
#define N 5
#define M 13
#define SLEEP 4
sem_t sem, mut;
int linie[N], tren[M];
pthread_t tid[M];
time_t start;

//rutina unui thread
void* trece(void* tren) {
    int t, l;
    t = *(int*)tren;

    slep(1 + rand()%SLEEP); // Inainte de
==> A

    sem_wait(&mut);
    printf("Moment %lu tren %d: ==> A\n",
time(NULL)-start, t);
    sem_post(&mut);

    sem_wait(&sem); // In A ocupa linia

    sem_wait(&mut);
    for (l = 0; l < N; l++) if (linie[l]
== -1) break;
    linie[l] = t;
    printf("\tMoment %lu tren %d: A ==> B
linia %d\n",time(NULL)-start, t, l);
    sem_post(&mut);

    sleep(1 + rand()%SLEEP); // Trece
trenul  A ==> B

    sem_wait(&mut);
    printf("\t \tMoment %lu tren %d: B
==>, liber linia %d\n", time(NULL)-start,
t, l);
    linie[l] = -1;
    sem_post(&mut);

    sem_post(&sem); // In B elibereaza
linia
}

// main
int main(int argc, char* argv[]) {
    int i;
    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++);
    start = time(NULL);

    // 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]);
```

```
&tren[i]);
    for (i=0; i < M; i++)
pthread_join(tid[i], NULL);

    pthread_mutex_destroy(&mutcond);
    pthread_cond_destroy(&cond);
}
```

```
    for (i=0; i < M; i++)
pthread_join(tid[i], NULL);

    sem_destroy(&sem);
    sem_destroy(&mut);
}
```

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`.

4. 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

```
#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
```

prodConsSem.c

```
#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
```

```
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);
```

```
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
```

```
        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);
}
```

```
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.

5. 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);
}
```

6. 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 imlementeze problema filosofilor folosind threaduri. Fiecare filosof va fi modelat cu ajutorul unui thread. Un numar de n (cel putin 2) filosofi stau la o masa circulara. Acestia fie mananca, fie gandesc. Intre orice doi filosofi exista un singur betigas (pe masa exista n betigase). Pentru a manca un filosof trebuie sa foloseasca cele doua betigase aflate unul in stanga si unul in dreapta lui. Programul va afisa cand un filosof vrea sa manance, daca poate sau nu, cand inceteaza sa manance etc. Perioadele in care un filosof manca sau gandeste vor fi de ordinul secundelor. Numarul de filosofi va fi variabil fiind dat ca parametru. Se vor folosi daca e 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 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.

8. 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.