Sincronizarea threadurilor; soluții ale unor probleme celebre

Contents

1. Principalele tipuri de date şi funcţii de lucru cu threaduri

Prezentăm tabelul cu principalele fişiere header, tipuri de date şi funcţii care lucrează cu threaduri:

Tabelul următor, reluare din seminarul precedent, 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_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`

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 thread-ului	`void * work(void* a)`

2. 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.

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

```
#include <stdlib.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
```

trenuriSem.c

```
#include <semaphore.h>
#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;
```

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

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

```
    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]);
    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 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

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

#define N 5
pthread_mutex_t mutex;
```

SleepingBarberSem.c

```
#include <stdio.h>
#include <stdlib.h>
#include <pthread.h>
#include <unistd.h>
#include <semaphore.h>
#define N 5
sem_t mutex, somn;
```

```
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;
```

```
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;
```

}	}

5. 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. Când 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);
}
```

6. 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
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]);
```

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
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
```

```
    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);
```

```
        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);
}
```

```
    }
}

//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.

7. 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);
}
```

8. Utilizarea altor platforme de threaduri

Tabelul următor prezintăcomparativ 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.