Curs 7
Probleme clasice de sincronizare

Readers-Writers Problem

Context: readers => doar citesc datele

Writers => citesc si scriu datele

Problema: permite mai multor readers sa citeasca in acelasi timp si UN SINGUR Writer poate accesa data la un anumit moment.

Algoritm de rezolvare a problemei (utlizand mutexi):

```
while (true){
        down(mutex);
        read_count++;
        if (read_count == 1) /* first reader */
             down(rw_mutex);
        up(mutex);
        ...
        /* reading is performed */
        ...
        down(mutex);
        read_count--;
        if (read_count == 0) /* last reader */
               up(rw_mutex);
        up(mutex);
}
```

Dining-Philosophers Problem

Context: n filosofi se afla la o masa rotunda cu un bol de orez in mijloc

Vecinii nu interactioneaza intre ei. Uneori se incearca sa se ridice 2 bete de mancat (in cazul in care sunt 5 de exemplu)

Algoritm de rezolvare a problemei (utilizand semafoare):

```
#define THINKING        0
#define HUNGRY          1
#define EATING          2
int state[N];
semaphore_t   mutex = {1};
semaphore_t   ph[N];   // toate initializate cu 0
void philosopher(int i) {
        while(true) {
                think();
                take_forks(i);
                eat();
                put_forks(i);
        }
}
```

```
void take_forks(int i) {
        down(mutex);
        state[i] = HUNGRY;
        test(i);
        up(mutex);
        down(ph[i]);
}
void put_forks(int i) {
        down(mutex);
        state[i] = THINKING;
        test((i – 1) % N));
        test((i + 1) % N));
        up(mutex);
}
```

```
void test(int i) {
        if(state[i] == HUNGRY &&
          state[(i-1)%N] != EATING) &&
          state[(i+1)%N] != EATING)
        {
                state[i] = EATING;
                up(ph[i]);
        }
}
```

Bounded-Buffer (producer-consumer problem)

Se da un sir de caractere si doua tipuri de persoane autorizate sa il modifice. Un tip are voie sa scrie, iar celalalt tip are voie sa steaerga cate un caracter. Este interzis sa se stearga daca sirul este vid si este interzis sa se adauge un caracter daca lungimea sirului ajunge la o constanta N.

Algoritm care sa rezolve problema utilizand mutex:

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

#define ITEMS           8
#define BUFFER_SIZE     4
int buffer[BUFFER_SIZE];

void *producer(void*);
void *consumer(void*);

int spaces, items, tail;
pthread_cond_t space, item;
pthread_mutex_t buffer_mutex;

int main()
{
        pthread_t producer_thread, consumer_thread;
        void *thread_return;
        int result;

        spaces = BUFFER_SIZE;
        items = 0;
        tail = 0;

        pthread_mutex_init(&buffer_mutex, NULL);
        pthread_cond_init(&space, NULL);
        pthread_cond_init(&item, NULL);

        if(pthread_create(&producer_thread, NULL, producer, NULL) ||
           pthread_create(&consumer_thread, NULL, consumer, NULL))
                exit(1);

        if(pthread_join(producer_thread, &thread_return))
                exit(1);
        else
                printf("producer returns with %d\n", (int)thread_return);

        if(pthread_join(consumer_thread, &thread_return))
                exit(1);
        else
                printf("consumer returns with %d\n", (int)thread_return);
        exit(0);
}
```

```c
void *producer(void *arg)
{
        int i;

        for(i = 0; i < ITEMS; i++)
        {
                pthread_mutex_lock(&buffer_mutex);
                while(spaces == 0)
                        pthread_cond_wait(&space, &buffer_mutex);
                buffer[(tail + items) % BUFFER_SIZE] = i;
                items += 1;
                spaces -= 1;
                printf("producer puts %d\n", i);
                pthread_mutex_unlock(&buffer_mutex);
                pthread_cond_signal(&item);
        }
        pthread_exit(0);
        return (void*)0;
}

void *consumer(void *arg)
{
        int i, b;

        for(i = 0; i < ITEMS; i++)
        {
                pthread_mutex_lock(&buffer_mutex);
                while(items == 0)
                        pthread_cond_wait(&item, &buffer_mutex);
                b = buffer[tail % BUFFER_SIZE];
                tail += 1;
                items -= 1;
                spaces += 1;
                printf("consumer gets %d\n", b);
                pthread_mutex_unlock(&buffer_mutex);
                pthread_cond_signal(&space);
        }
        pthread_exit(0);
        return (void*)0;
}
```

Sincronizari kernel

Windows -> utilizeaza interrupt masks pt a proteja accesul la resurse globale pe sisteme cu uniprocesor

->utilizeaza spinlocks pt cele cu multiprocesor

Linux pune la dispozitie:

1. Semafoare
2. Atomic integer (countere)
3. Spinlocks
4. Reader-writer versions of both

Solaris pune la dispozitie:

1. Lockuri adaptive
2. Variabile de condiite
3. Semafoare
4. Reader-writer locks
5. Turnstiles

Posix pune la dispozitie:

1. Mutex locks
2. Semafoare (named=pot fi folosite de procese unrelated / unnamed)
3. Variabile de conditie

Memorie tranzactionala

Se bazeaza pe ideea de tranzactii din baze de date. Astfel, un thread termina modificarile operate pe o zona de memorie partajata fara a se coordona cu alte threaduri, iar modificarile sunt trecute intr-un log.

La finalul tranzactiei se incearca un COMMIT. Daca reuseste (in acest caz nicio alta tranzactie nu a apucat sa faca vreun commit reusit intre timp) tranzactia e validata si modificarile devin permanente, altfel tranzactia e abandonata, se da roll-back si se incearca iar pana cand se reuseste.

BENEFICIU: nu mai e nevoie de sincronizare prin lock-uri la memorie partajata.