Seminary 6

Conditional variables

Considering two threads, one waits for an even to happen to be able to continue. We call it **rendezvous**.

B	A

`void* produceevent(void* nume) {`
`        ...`
`        printf("B producing event\n");`
`        event = 1;`
`        printf("Event done\n");`
`        ...`
`}`
	`void* waitevent(void* nume) {`
	`        ...`
	`        printf("A waiting event\n");`
	`        while (event == 0) {`
	`                ;`
	`        }`
	`        printf("A received event\n");`
	`        ...`
	`}`

The problem is that `while` performs an **active waiting (busy waiting)**, the processor is occupied by the tread with running the while instruction. To make use this wasted time, we use a conditional variable with an associated mutex:

```
pthread_cond_t var = PTHREAD_COND_INITIALIZER;

pthread_mutex_t varmtx = PTHREAD_MUTEX_INITIALIZER;
```

Code changes into:

`pthread_mutex_lock(&varmtx);`
`event = 1;`
`pthread_cond_signal(&var);`
`pthread_mutex_unlock(&varmtx);`
	`pthread_mutex_lock(&varmtx);`
	`while (event == 0) {`
	`    pthread_cond_wait(&var,&varmtx);`
	`}`
	`pthread_mutex_unlock(&varmtx);`

Pthread_cont_wait will:

- Unlock the mutex
- Wait for a signal
- When signal comes, lock mutex and continue execution

Pthred_cond_signal will:

- Trigger a signal that will wake wait

Semaphores

The POSIX system in Linux presents its own built-in semaphore library. To use it, we have to:

1. `#include <semaphore.h>`
2. Compile the code by linking with `-lpthread -lrt`

To lock a semaphore or wait we can use the **sem_wait** function:

```
int sem_wait(sem_t *sem);
```

To release or signal a semaphore, we use the **sem_post** function:

```
int sem_post(sem_t *sem);
```

A semaphore is initialised by using **sem_init**(for processes or threads) or **sem_open** (for IPC).

```
sem_init(sem_t *sem, int pshared, unsigned int value);
```

Where,

- **sem**: Specifies the semaphore to be initialized.
- **pshared**: This argument specifies whether or not the newly initialized semaphore is shared between processes or between threads. A non-zero value means the semaphore is shared between processes and a value of zero means it is shared between threads.
- **value**: Specifies the value to assign to the newly initialized semaphore.

To destroy a semaphore, we can use **sem_destroy**.

```
sem_destoy(sem_t *mutex);
```

Example: A binary semaphore has the effect of a mutex. The same is the case of a write lock.

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

sem_t mutex;
pthread_t t1[10];

void* thread(void* arg) {
    sem_wait(&mutex); //mutex lock
    //critical section
    sem_post(&mutex); //mutex unlock
}

int main()
{
    sem_init(&mutex, 0, 1); //initialize semaphore as binary

    for ... pthread_create(&t1[i],NULL,thread,NULL);
    for ... pthread_join(t1[i],NULL);

    sem_destroy(&mutex);
    return 0;
}
```

Barrier

```
#include <pthread.h>

int pthread_barrier_destroy(pthread_barrier_t *barrier);

int pthread_barrier_init(pthread_barrier_t *restrict barrier, const
pthread_barrierattr_t *restrict attr, unsigned count);

int pthread_barrier_wait(pthread_barrier_t *barrier);
```

The calling thread shall block until the required number of threads have called pthread_barrier_wait() specifying the barrier.

When the required number of threads have called pthread_barrier_wait() specifying the barrier, the constant PTHREAD_BARRIER_SERIAL_THREAD shall be returned to one unspecified thread and zero shall be returned to each of the remaining threads. At this point, the barrier shall be reset to the state it had as a result of the most recent pthread_barrier_init() function that referenced it.

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

#define THREAD_COUNT 4

pthread_barrier_t mybarrier;

void* threadFn(void *id_ptr) {
  int thread_id = *(int*)id_ptr;
  int wait_sec = 1 + rand() % 5;
  printf("thread %d: Wait for %d seconds.\n", thread_id, wait_sec);
  sleep(wait_sec);
  printf("thread %d: I'm ready...\n", thread_id);

  pthread_barrier_wait(&mybarrier);

  printf("thread %d: going!\n", thread_id);
  return NULL;
}

int main() {
  int i;
  pthread_t ids[THREAD_COUNT];
  int short_ids[THREAD_COUNT];

  srand(time(NULL));
  pthread_barrier_init(&mybarrier, NULL, THREAD_COUNT + 1);

  for (i=0; i < THREAD_COUNT; i++) {
    short_ids[i] = i;
    pthread_create(&ids[i], NULL, threadFn, &short_ids[i]);
  }

  printf("main() is ready.\n");
  pthread_barrier_wait(&mybarrier);
  printf("main() is going!\n");

  for (i=0; i < THREAD_COUNT; i++) {
    pthread_join(ids[i], NULL);
  }

  pthread_barrier_destroy(&mybarrier);

  return 0;
}
```

Problems & Exercises (To see solutions, Options -> see hidden text)

1. Threads

A. What will be printed by the code below?
R:

B. What needs to be changed in the code so that the threads with run concurrently?
R:

C. In case B, what will be printed at different executions?
R:

```
#include <pthread.h>
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <ctype.h>
#define MAXLINIE 1000

pthread_t tid[100];

void* partial(void* id) {
        int nr= *(int *)id;

        printf("Thread %ld - %d\n", pthread_self(), nr);
}

int main(int argc, char* argv[]) {
        int tnr[100];
        int i=0;

        for (i=0; i<100; i++) {
                 tnr[i]=i;
        }

        for (i=0; i<100; i++) {
             pthread_create(&tid[0], NULL, partial, (void*)&tnr[0]);
             pthread_join(tid[0],  NULL);
        }

        printf("Finished\n");
        return 0;
}
```

D. What will the following code print?

R:

```
#include <stdio.h>
#include <pthread.h>
int n=0;
void* f(void * a){
        n++;
        return NULL;
}
int main(){
        int i;
        pthread_t t[7];

        for (i=0; i<7; i++)
                pthread_create(&t[i], NULL, f, NULL);
        for (i=0; i<7; i++)
                pthread_join(t[i], NULL);
        printf("%d \n", n);
        return 0;
}
```

E. What will print the code below if we run the program with parameters 1 2 3 4?

R:

F. Modify the program to print 1 2 3 4 or 3 4 1 2.

R:

```
#include <pthread.h>
#include <stdio.h>
#include <stdlib.h>
typedef struct {char*n1; char*n2;} PERECHE;
pthread_t tid[100];
PERECHE pair;

void* computepairs(void* pair) {
    int n1 = atoi(((PERECHE*)pair)->n1);
    int n2 = atoi(((PERECHE*)pair)->n2);
    printf(”N1=%d N2=%d \n”, n1, n2);
}

int main(int argc, char* argv[]) {
    int i, p, n = (argc-1)/2;

    for (i = 1, p = 0; p < n; i += 2, p++) {
        pair.n1 = argv[i];
        pair.n2 = argv[i+1];
        pthread_create(&tid[p], NULL, computepairs, (void*)&pair);

    }
    for (i=0; i < n; i++)
        pthread_join(tid[i], NULL);

    return 0;
}
```

2. Mutex /RWLock

A. Replace the mutex with a different synchronisation mechanism that has the same effect:

R:

<table>
<tr><th>Mutex</th><th>RWLOCK</th><th>Binary Semaphore</th></tr>
<tr><td><pre>int n=0;
pthread_mutex_t m=
PTHREAD_MUTEX_INITIALIZER;

void* f(void * a){
      pthread_mutex_lock(&m);
      n++;
      pthread_mutex_unlock(&m);
      return NULL;
}

int main(){
      int i;
      pthread_t t[10];
      for (i=0; i<10; i++)
        pthread_create(&t[i],0,f,0);
      for (i=0; i<10; i++)
        pthread_join(t[i], NULL);

      pthread_mutex_destroy(&m);
      return 0;
}</pre></td><td></td><td></td></tr>
</table>

B. Consider the code below. Discuss the use of mutex and see if there could be a more efficient way to use it.

R:

```
#include <pthread.h>
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <ctype.h>
#define MAXLINIE 1000
pthread_t tid[100]; //we need to refer to each thread to join them
int countE=0;
pthread_mutex_t m= PTHREAD_MUTEX_INITIALIZER;

void* ucap(void* numei) {
    pthread_mutex_lock(&m);

    printf("Thread start: %ld ...> %s\n", pthread_self(), (char*)numei);
    char numeo[100];
    strcpy(numeo, (char*)numei);
    if ( numeo[0]>=`a` && numeo[0]<=`z`)
        numeo[0]+=’A’-‘a’;

    if (numeo[0]==’E’) countE++;

    printf("Thread finished: %ld > %s\n", pthread_self(), (char*)numeo);

    pthread_mutex_unlock(&m);

}
int main(int argc, char* argv[]) {
    int i;
    for (i=1; argv[i]; i++) {
        pthread_create(&tid[i], NULL, ucap, (void*)argv[i]);
        printf("Thread created: %ld ...> %s\n", tid[i], argv[i]);
    }
```

```
        for (i=1; argv[i]; i++) pthread_join(tid[i], NULL);
        printf("All threads finished\n");
        pthread_mutex_destroy(&m);

        return 0;
    }
```

3. Barriers

What will this code print?

A) When `pthread_barrier_init(&b1, 0, 11);` and we have 7 threads.

R:

B) If we change to `pthread_barrier_init(&b1, 0, 7);` we have 7 threads.

R:

C) If we change to `pthread_barrier_init(&b1, 0, 3);`.

R:

```
#include <stdio.h>
#include <pthread.h>
int n=0;
pthread_barrier_t b1,b2;

void* f(void * a){
        pthread_barrier_wait(&b1);
        n++;
        return NULL;
}

int main(){
        int i;
        pthread_t t[7];
        pthread_barrier_init(&b1, 0, 11);

        for (i=0; i<7; i++)
                pthread_create(&t[i], NULL, f, NULL);
        for (i=0; i<7; i++)
                pthread_join(t[i], NULL);

        pthread_barrier_destroy(&b1);
        printf("%d \n", n);
        return 0;
}
```

4. Conditional variables

What will be printed by the following program?

A) As it is, with CONDITION=9, **R:**
B) If we #define CONDITION 3, **R:**
C) If we #define CONDITION 10? **R:**

```
#include <stdio.h>
#include <pthread.h>
#include <unistd.h>
#define CONDITION 9

int n=0;
pthread_mutex_t m = PTHREAD_MUTEX_INITIALIZER;
pthread_cond_t c = PTHREAD_COND_INITIALIZER;

void* f(void* a){

        pthread_mutex_lock(&m);

        n++;
        if (n>CONDITION)
                pthread_cond_signal(&c);

        pthread_mutex_unlock(&m);

        return NULL;
}


int main(){
        int i;
        pthread_t t[10];

        for (i=0; i<10; i++)
                pthread_create(&t[i], NULL, f, NULL);


        pthread_mutex_lock(&m);

        while (n<=CONDITION)
                pthread_cond_wait(&c, &m);
        printf("%d \n", n);

        pthread_mutex_unlock(&m);


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


        return 0;
}
```

5. Semaphores

Given the code below:

A) What will this program print when the semaphore is binary? **R:**

B) What will it print when `sem_init(&sem, 0, 100);` ? **R:**

C) How about when `sem_init(&sem, 0, 20);` ? **R:**

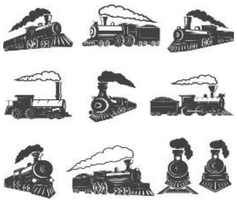
```
#include <stdlib.h>
#include <stdio.h>
#include <pthread.h>
#include <semaphore.h>
#include <unistd.h>
#define T 1000

sem_t sem;
pthread_t t[T];
long n=0;

void* f(void* v){
        int i;
        for (i=0; i<1000; i++){
                sem_wait(&sem);
                n++;
            // to test, you need to find ways to mix up thread operations. For example:
           //     int a=n;   a++;   if (i<5 && rand()<50000)  sleep(rand()%2);    n=a;
                sem_post(&sem);
        }
        return NULL;
}

int main(){
        sem_init(&sem, 0, 1);
        int i=0;

        for (i=0; i<T; i++)
                pthread_create(&t[i], NULL, f, NULL);
        for (i=0; i<T; i++)
                pthread_join(t[i], NULL);

        sem_destroy(&sem);
        printf("%lu\n",n);
        return 0;
}
```

6. Trains (Conditional variable and Semaphores)

Between two train stations A and B (say Cluj and Vienna) there are m trains than need to pass on n lines. M trains enter Cluj train station, and they want to get to Vienna. Between Cluj and Vienna there are n lines, m>n, but in the train station we can have at most n trains. Trains enter Cluj at a random interval, if there is a free line they continue towards Vienna, and this takes a certain (random) amount of time. Simulate these trains passing by.

M trains

N lines →



A (Cluj)

B (Vienna)

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;

void* trece(void* tren) {
    int t, l;
    t = *(int*)tren;

    sleep(1 + rand()%SLEEP); //Before=> 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);
}
```

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;

void* trece(void* tren) {
    int t, l;
    t = *(int*)tren;

    slep(1 + rand()%SLEEP); //Before=>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
}
```

```
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);
    // what about &i instead &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);

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

```
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);

    // what about &i instead &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);
}
```

Home Training Problems

1. Solve problem 6 from Seminary 5 (BEST COMPUTER SCIENCE MEMES) using conditional variables: students check the winners list then, if not all the winners were listed, they wait until new winners are announced. When a sponsor announces a winner, they notify students to check the list again.

2. After implementing the conditional variable solution for the BEST COMPUTER SCIENCE MEMES contest, the announcements website server (which was a quick set-up for the contest on a server already overcrowded with loads of other student stuff…) started to crash because all 100 participants and their 1500 colleagues, family and friends were trying to check the list at the same time when receiving a notification. Prevent this from happening by adding a semaphore that allows a maximum of 30 persons (readers) to check the list at the same time.

Read more

Deadlock

Conditions for Deadlock

Coffman et al. (1971) showed that four conditions must hold for there to be a deadlock:
1. Mutual exclusion condition. Each resource is either currently assigned to exactly one process or is available.
2. Hold and wait condition. Processes currently holding resources that were granted earlier can request new resources.
3. No preemption condition. Resources previously granted cannot be forcibly taken away from a process. They must be explicitly released by the process holding them.
4. Circular wait condition. There must be a circular chain of two or more processes, each of which is waiting for a resource held by the next member of the chain.

Deadlock solutions

Read: http://nicku.org/ossi/lab/processes/posix-threads.pdf
Program 2 and 3 (with main in Program 6) present a deadlock situation.
Program 4 is the back off solution to avoid deadlock.
Program 5 is the reordering of resources solution to avoid deadlock.