Exercise 1

Concurrent programming using the Pthreads library

1.1 Introduction

In multitasking system a task which we have to resolve we can divide to subtasks. Subtasks can be executed by separate processes. Operating system reserves separate area of memory for every process. If we have multi processor computer processes are executed in parallel – each process is executed by separate processor. In single-processor computer, the operating system uses a hardware clock to allocate ,,time slices" for each currently running process. If the time slices are small and computer is not overloaded with too many processes trying to do something it appears to a user as if all the processes are running concurrently.

Threads are often called lightweight processes. They also permit concurrent realization of subtasks. Managing threads requires fewer system resources than managing processes. Threads have own program counter, a stack, and a set of registers but all the other data structures belong to the process which created them and threads share them. The operating system does not protect before parallel access to shared data structures. Each process may consists of single thread, like Process 1 or more threads like Processes 2 to 5 (see fig. 1.1 [1]). Comunnication between threads is more efficient and in many cases easier to use than communication between processes. Operating system does not control concurrent access to the same data structures by different threads. This is made by programmer. How to do it we will describe in next sections. In the next part of this work we are not going to describe parallel processes but we are going to describe a threads.

There are two kinds of threads: user-level threads and kernel-level threads (fig. 1.1). User-level threads avoid the kernel entirely. The kernel is not aware of the existence of threads,

[1] Fig. 1.1 is taken from Sun web page:
`http://docs.sun.com/app/docs/doc/801-6659/6i116aqmb?a=view`

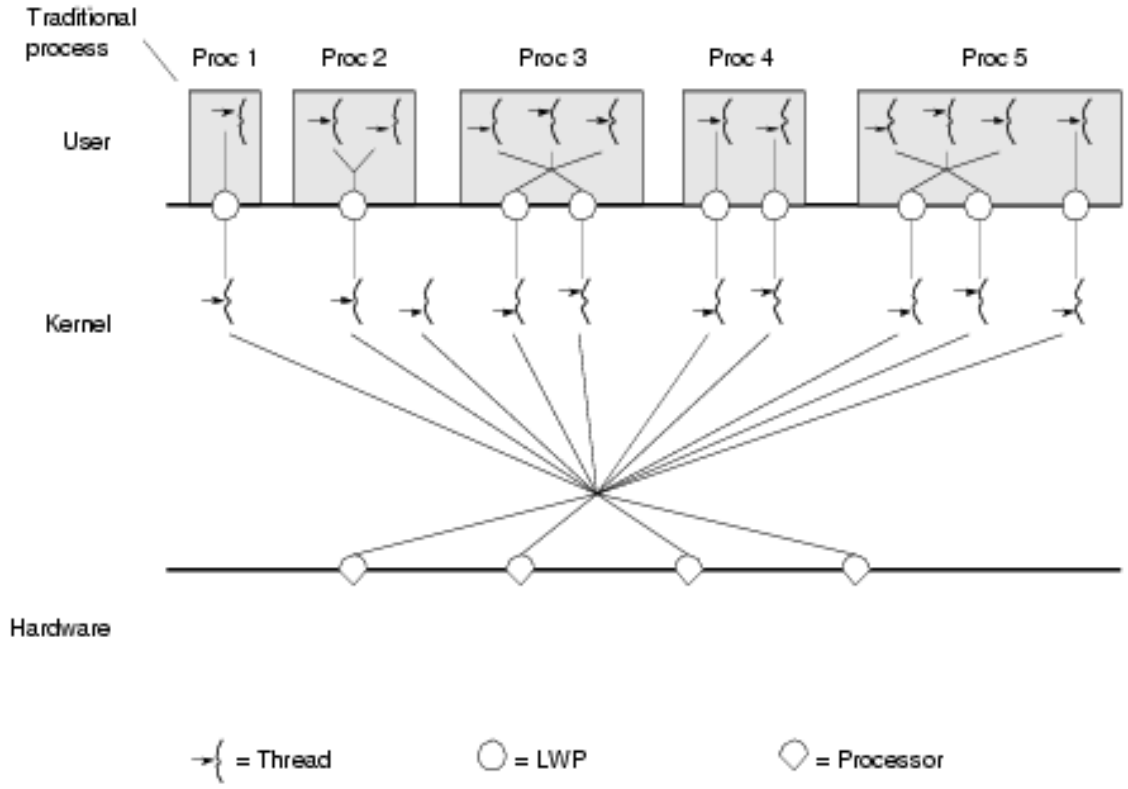

Figure 1.1: User-level and kernel-level threads

it only knows about existence of process containing threads. All thread management is done by the application by using a thread library. Thread switching can be done without kernel intervention. User-level thread can be run on any operating systems, it only needs a thread library. The threads library uses underlying threads of control called light-weight processes that are supported by the kernel. A light-weight process (LWP) is like as a virtual processor that executes code or system calls. It is a bridge between the user level and the kernel level. Each process consists of one or more light-weight processes, each of which runs single or more user-level threads. But the kernel does not know how many threads are executed by light-weight process. It treats whom like a single-threaded process. For example, Process 2 (fig. 1.1) contains only single light-weight process, which runs two threads but Process 4 contains two light-weight processes. Each of them executes only single thread.

Light-weight process is referred to as a kernel-level thread. There is a one-to-one mapping of light-weight process to kernel-level thread. Each process within system is associated with single or more kernel-level threads. For example, Process 1 (fig. 1.1) is associated with only one kernel-level thread but Process 5 is associated with three kernel-level threads. In distinguishing from user-level threads, the kernel is aware of existence of kernel-level threads. All thread management is done by the kernel. There is no thread library but an API (Application Programming Interface) to kernel thread management. The kernel schedules many kernel-

level threads on many processors. It decides which processor executes kernel-level thread.

Threads can be executed concurrently if they are not run by the same light-weight process. If threads are run by the same light-weight process - like threads in Process 2 (fig. 1.1) - they cannot be executed concurrently, because they will be executed by the same processor. By that reason the first thread is blocked e.g. if the second is blocked during input-output operation. Other situation is in case of Process 5. It contains three light-weight processes, the last runs only single thread. So last thread can be run concurrently with other three threads. It will not be executed on the same processor that three other threads.

There are problems related to concurrent programing - synchronization, mutual exclusion and a deadlock. Often threads share the same resources like variables, files, printers, etc. Sometimes a thread must have exclusive access to a resource. For example, when a thread is updating variable, no other thread should have access to this variable at the same time. A sequence of instructions that access shared resource is named as a critical section. The necessity to restrict access to a resource is termed as a mutual exclusion. It involves the following:

- At most one thread has access to a shared resource.
- If there are multiple requests for a resource, it must be granted to one of the threads in finite time.
- When a thread has exclusive access to a shared resource it releases it in a finite time.
- When a thread requests a resource it must obtain the resource in finite time.
- A thread should not consume processor time while waiting for a resource.

There are several mechanisms that can be used to solve the mutual exclusion problem. We will describe semaphores, mutexes and monitors.

The second problem related to concurrent programing is a deadlock. It is a situation in which a set of threads are prevented from making any further progress by their mutually incompatible demands for additional resources. Deadlock can occur if, and only if, the following conditions all hold together:

- threads have exclusive access to the resources,
- threads continue to hold a resource while waiting for a new resource request to be granted,
- resources cannot be removed from a thread,
- there is a cycle of threads, each is awaiting a resource held by the next thread in the cycle.

A deadlock may be prevented by weakening one or more of the conditions. For example the second condition may be modified to require a thread to request all needed resources at one time. The circular-wait condition may be modified by imposing a total ordering on resources and insisting that they be requested in that order.

In next sections we are presented the Pthreads library for creating threads and synchronization. We will describe threads synchronization using semaphores, mutexes and condition variables. There are presented only part of library which knowledge is indispensable for realizing of laboratory exercise. It is not complete the Pthreads library manual. We assume that student has basic knowledge of theoretical problems which were presented in the lectures.

1.2 The Pthreads library

In the 1995 the POSIX (Portable Operating Systems Interface) organization defined a standard library for multithreaded programming. It is the IEEE POSIX 1003.1c standard. The library is called Pthreads and is availible in the UNIX and others operating systems. The Pthreads library consist of set of C language functions and types for thread managment and synchronization. The types and functions are described in the header file `pthread.h`. All applications using threads must to include it. Apart from that the library must be linked during compilation. For example for `gcc` compiler:

```
gcc -o example example.c -lpthread
```

where `example` is the name of executable (output) file and `example.c` is the name of a source file.

1.3 Threads management

1.3.1 Creating a thread

A thread is described by ,,thread id", commonly abbreviated as ,,tid". It is a kind of thread handle (a thread descriptor) and it is a variable of type `pthread_t`. The `pthread_t` type is defined in header file `pthread.h`.

A thread is represented by thread function which can have any name. It is passed a single parameter which has a type of pointer to `void`. A thread function must return a value, which becomes the thread's exit code. A returned value is a type of pointer to `void`.

A new thread is created by calling the function:

```
int pthread_create(pthread_t *tid, pthread_attr_t *attr,
                   void* (*thread_func)(void *), void *arg);
```

where:

- `tid` - a addres of a thread id,
- `attr` - specifies a thread attributes to be applied to a thread; if it is `NULL` then default attributes are used[2],
- `thread_func` - a addres of a thread function,
- `arg` - a addrees of a argument of a thread function.

If thread creation is not successful, the function returns a non-zero error code. Otherwise the function returns a zero and the thread id of the newly created thread is stored in the location pointed by first argument and `thread_func` function is called.

1.3.2 Termination of a thread

A thread can be terminated in several ways:

- when a thread function returns,
- when a thread ends itself by calling the `pthread_exit` function,
- a thread is canceled by another thread by call to `pthread_cancel` function,
- when ends process which created a thread.

A thread can terminate itself by calling function:

```
void pthread_exit(void *value).
```

It terminates the execution of the calling thread. It is called after a thread has completed its work and is no longer required to exist. The `value` argument is the return value of the thread. It can be consulted from another thread or process calling the `pthread_join` function (described in section 1.3.3). The `pthrtead_exit` terminates the thread but it does not close files. Any files opened inside the thread will remain open after the thread is terminated. It does not free the memory allocated inside the thread, either.

A thread can terminate another thread. It needs to call a function:

```
int pthread_cancel(pthread_t tid);
```

[2]A thread with default attributes suffices to realize tasks in the laboratory exercises.

where `tid` is a thread id, which is to be terminated. It sends a cancellation request to this thread. The target thread can ignore the request, honor it immediatly, or defer it till it reaches a cancellation point. The last option is default for all threads created with deffault attributes. A cancellation point is serve by functions that might suspend the execution of a thread for a long time. For example, there are following functions:

- `pthread_join`,
- `pthread_cond_wait`,
- `sem_wait`.

If a thread executes any of these functions, it will check for deferred cancel requests. Funtions mentioned above are described in next sections.

1.3.3 Joining a thread

Joining thread is a one way for synchronization between threads, other ways will be described in next sections. A thread can wait for another thread to terminate by executing:

```
int pthread_join(pthread_t tid, void **value);
```

where:

- `tid` - thread id which we wait for terminating,
- `value` - a pointer to a value returned by terminated thread.

The joined thread must be in the joinable. All threads created with default attributes are joinable. On success, 0 is returned and the return value of terminated thread is stored in the location pointed to by `value` argument. On failure a non-zero value is returned.

When a joinable thread terminates, its memory resources - thread descriptor and stack - are not deallocated untill another thread call the `pthread_join` function on it. Therefore, it must be called once for each joinable thread to avoid memory leaks.

1.3.4 Sample code

```
 1:  #include <stdio.h>
 2:  #include <pthread.h>

     /* number of matrix columns and rows */
 3:  #define M 5
 4:  #define N 10
 5:  int matrix[N][M];
     /* thread function; it sums the values of the matrix in the row */
 6:  void *SumValues(void *i)
 7:  {
 8:    int n = (int)i;         /* number of row */
 9:    int total = 0;          /* the total of the values in the row */
10:    int j;
11:    for (j = 0; j < M; j++)      /* sum values in the "n" row */
12:      total += matrix[n][j];
13:    printf("The total in row %d is %d\n", n, total);
       /* terminate a thread and return a total in the row */
14:    pthread_exit((void*)total);
15:  }
16:  int main(int argc, char *argv[])
17:  {
18:    int i, j;
19:    pthread_t threads[N];             /* descriptors of threads */
20:    int total = 0;     /* the total of the values in the matrix */
21:    for (i = 0; i < N; i++)           /* initialize the matrix */
22:      for (j = 0; j < M; j++)
23:        matrix[i][j] = i * M + j;
24:    for(i = 0; i < N; i++)           /* create threads */
25:      if (pthread_create(&threads[i], NULL, SumValues, (void *)i))
26:      {
27:        printf("Can not create a thread\n");
28:        exit(1);
29:      }
30:    for (i = 0; i < N; i++)         /* wait for terminate a threads */
31:    {
32:      int value;                     /* value returned by thread */
33:      pthread_join(threads[i], (void **)&value);
34:      total += value;
35:    }
36:    printf("The total values in the matrix is %d\n", total);
37:    return 0;
38:  }
```

The total of the values of the matrix is computed by `N` threads. Each thread computes the total value in the row and returns it to the `main` function which sums it. The `main` function initializes the matrix (lines 21-23) and creates a `N` threads (lines 24-29). Each of `N` threads is created by the `pthread_create` function (line 25) and represented by descriptor. Thread descriptors are declared in line 19 as an `N` size array with the `pthread_t` type values. Threads share acces to the matrix which is declared as a global variable (line 5). The matrix is not protected in a critical section because threads only read it and do not update. Threads do not read values from the same matrix's cells because values in the same row are summed by only one thread.

Each thread executes the `SumValues` function which is defined in lines 6-15. The thread function has a single argument. Argument is passed to the thread function when thread is created - it is the last argument of the `pthread_create` function. It is an index of matrix's row which will be summed by thread. Each thread computes a total value in one row (lines 11-12) and returns it calling the `pthread_exit` function (line 14).

The `main` function waits for all threads to terminate using the `pthread_join` function (line 33) and computes the total of the values of the matrix. It sums values returned by all threads (line 34). When the total is computed it prints it to standard output using the `printf` function (line 36).

1.4 Mutexes

A mutex is a mutual exclusion is useful for protecting access to a shared resources from concurrent modifications. In fact, this is how the mutex got its name - MUTUal EXclusion. It has two possible states:

- locked - owned by a thread,
- unlocked - not owned by any thread.

Each mutex has a locking count (numer of locks operations performed on it by the calling threads). A mutex can never be owned by two different threads in same time. A thread attempting to lock a mutex that is already locked by another thread is suspended until the owning thread releases the mutex first. At this time, the first thread will wake up and continue execution, having the mutex locked by it. A suspended thread does not consume any CPU resources. Locking a mutex is an atomic operation.

1.4.1 Creating a mutex

Mutexes in the Pthread library are represented by variables of type `pthread_mutex_t`. A mutex must be initialized before it can be used. It is initialized by calling the function:

```
int pthread_mutex_init(pthread_mutex_t *mutex,
                       const pthread_mutexattr_t *attr);
```

where:

- `mutex` - a pointer to a mutex to be initialized,
- `attr` - specifies a mutex attributes to be applied to a mutex; if it is `NULL` then default attributes are used[3].

The `pthread_mutex_init` function always returns 0. The mutex is initially unlocked.

1.4.2 Using a mutex

Mutexes are used to protect a critical section. At entry to a critical section mutex is locked by using function:

```
int pthread_mutex_lock(pthread_mutex_t *mutex);
```

where `mutex` is a pointer to a mutex to be locked. The function returns 0 on success or a non-zero error code on failure. It increments a locking count of the mutex to be locked. If the mutex is currently unlocked, it becoms locked and owned by the calling thread, and the function returns immediately. Otherwise it suspends the calling thread until the mutex is unlocked. There is a non blocking function for locking mutex. It is a function:

```
int pthread_mutex_trylock(pthread_mutex_t *mutex).
```

It behaves identically to the `pthread_mutex_lock` function, excepts that it does not block the calling thread if the mutex is already locked by another thread. It returns immediately with the error code `EBUSY`.

At exit from a critical section mutex is released by calling function:

```
int pthread_mutex_unlock(pthread_mutex_t *mutex).
```

On success the function returns 0, otherwise non-zero error code. A thread should always release a mutex that it has locked. If the mutex is created with default attributes, attempting to unlock the mutex if it was not locked by the calling thread results in undefined behavior. The function decrements a locking count of the mutex, and only when this count reaches 0 is the mutex actually unlocked.

[3]Mutexes with default attributes suffices to realize a tasks in the laboratory exercises.

1.4.3 Destroying a mutex

If a mutex is no longer needed it should be destroyed by calling function:

```
int pthread_mutex_destroy(pthread_mutex_t *mutex).
```

A mutex must be unclocked on entrance to this function. On success the function returns 0, otherwise it returns non-zero error code.

1.4.4 Sample code

This example presents how to use a mutex to protect a critical section. Program computes the total of the values of the matrix like application presented in section 1.3.4. The total value is computed by threads which execute thread function `SumValues` which is defined in lines 8-20. The total is stored in global variable `total` declared in line 6. All threads share this variable and update its value (line 17). It should be protected in a critical section. A critical section is protected by the mutex. It is represented by global variable with the `pthread_mutex_t` type declared in line 7. The mutex is created with default attributes in the `main` function (line 28).

The matrix is declared as a global variable (line 5). It is not protected in a critical section because threads do not update it. Each thread only reads the matrix's values (line 14).

Each thread computes a total at the one row (lines 13-14) and updates global variable storing a total values of the matrix (line 17). It locks the mutex (line 16) before it updates this variable and no other thread cannot change its value in the same time. When the variable is updated thread unlocks the mutex (line 18). All threads terminate and do not return a value (line 19).

The `main` function waits for all threads to terminate (lines 35-39), prints the total values of the matrix (line 40) and destroys the mutex (line 41).

```
 1:  #include <stdio.h>
 2:  #include <pthread.h>
     /* number of matrix rows and columns */
 3:  #define M 5
 4:  #define N 10
 5:  int matrix[N][M];
 6:  int total;                /* the total values in the matrix */
 7:  pthread_mutex_t mutex;    /* mutex protecting the "total" variable */
     /* thread function; it sums the values of the matrix */
 8:  void *SumValues(void *i)
 9:  {
10:    int n = (int)i;         /* number of row */
11:    int my_total = 0;       /* the total of the values in the row */
12:    int j;
13:    for (j = 0; j < M; j++)      /* sum values in the "n" row */
14:      my_total += matrix[n][j];
15:    printf("The total in row %d is %d\n", n, my_total);
16:    pthread_mutex_lock(&mutex);       /* lock a mutex */
17:    total += my_total;                /* update a global total */
18:    pthread_mutex_unlock(&mutex);     /* release a mutex */
19:    pthread_exit(NULL);               /* terminate a thread */
20:  }
21:  int main(int argc, char *argv[])
22:  {
23:    int i, j;
24:    pthread_t threads[N];           /* descriptors of threads */
25:    for (i = 0; i < N; i++)         /* initialize the matrix */
26:      for (j = 0; j < M; j++)
27:        matrix[i][j] = i * M + j;
28:    pthread_mutex_init(&mutex, NULL);  /* initialize a mutex */
29:    for(i = 0; i < N; i++)             /* create threads */
30:      if (pthread_create(&threads[i], NULL, SumValues, (void *)i))
31:      {
32:        printf("Can not create a thread\n");
33:        exit(1);
34:      }
35:    for (i = 0; i < N; i++)     /* wait for terminate a threads */
36:    {
37:      int value;      /* value returned by thread */
38:      pthread_join(threads[i], (void **)&value);
39:    }
40:    printf("The total values in the matrix is %d\n", total);
41:    pthread_mutex_destroy(&mutex);       /* destroy a mutex */
42:    return 0;
43:  }
```

1.5 Semaphores

Semaphores are used by the operating system to synchronize processes or threads. They are useful in synchronizing the access of different processes to shared resources. Semaphores are a programming construct designed by Edsger W. Dijkstra in the 1965 and published in his paper „*Co-operating Sequential Processes*". A semaphore `S` is an integer variable that takes only non-negative values. Before it can be used it should be initialized by non-negative value. A semaphore `S` can be accessed only through two atomic operations - `P(S)` and `V(S)` - which are defined as:

```
P(S):                              V(S):
  if S > 0 then                      if some processes are suspended then
    S := S - 1                         wake up one
  else                               else
    suspend calling process            S := S + 1
  end if                             end if
```

Letters `P` and `V` take from proper words in dutch - *proberen* (to try) and *verhogen* (to raise) or *passeren* (to pass) and *vrijgeven* (to release). In english `P` and `V` operations are named *wait* and *signal*.

If a semaphore takes any integer non-negative values it is named as the general semaphore. It may be used for a counter for resources shared between threads. There is the binary semaphore, when takes only two values: 0 and 1.

Using semaphores, you must be careful to avoid a deadlock condition. Deadlock occurs when a process locks a semaphore, then later tries to lock the same semaphore again when it has a 0 value. The process cannot resume until the semaphore is unlocked, and cannot unlock the semaphore until it can resume processing, so the process never resumes. At the same time, any other process that depends on the same semaphore will halt when it tries to lock the semaphore.

1.5.1 Creating a semaphore

The types and functions for managment of semaphores are described in the header file `semaphore.h`. A semaphore is represented by variable, of type `sem_t`. It must be initialized before it can be used. It is initialized by following function:

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

where:

- `sem` - a pointer to a semafore to initialize,

- `shared` - if non-zero then a semaphore is shared between processes, otherwise it is local to the current process,
- `value` - an initial value of a semaphore.

The `sem_init` initialize a semaphore and return 0 on success, otherwise it returns -1.

1.5.2 Using a semaphore

In section 1.5 two atomic operations on semaphores (`P` and `V`) are defined. The first operation is implemented by following function:

```
int sem_wait(sem_t *semaphore).
```

It suspends the calling thread until a semaphore has non-zero count, and then decrements a semaphore count. It always return 0. In order to perform the `V(semaphore)` operation we should use function:

```
int sem_post(sem_t *semaphore).
```

It increments value of a semaphore. This function never blocks and returns 0 on success and -1 on error.

There is non blocking variant of the `sem_wait` function:

```
int sem_trywait(sem_t *semaphore).
```

If a semaphore count is 0, the calling thread is not blocked and function returns with error. Otherwise the action of the `sem_trywait` is the same as the action of the `sem_wait` function.

We can also read a current count of the semaphore. For the purposes of this operation is function:

```
int sem_getvalue(sem_t *semaphore, int *value).
```

It always returns 0 and stores the current count of the semaphore in the location pointed to by the second argument.

1.5.3 Destroying a semaphore

If a semaphore is no longer needed it should be destroyed by the call to function `sem_destroy`:

```
int sem_destroy(sem_t *semaphore).
```

A semaphore can be destroyed if there are no threads waiting on the semaphore at the time when the `sem_destroy` is called. It returns -1 on error and 0 otherwise.

1.5.4 Sample code

We present the use of semaphores, taking producer and consumer problem as an example. We have two processes - the producer and the consumer. The producer produces a products and stores them in a FIFO queue. The queue has limited capacity. Therefore the producer must wait for free places in the queue. Products are consumed by the consumer. When the queue is empty consumer waits for products in the queue.

Producer and consumer processes are implemented using separate threads. The consumer executes the `consumer` function (defined in lines 27-41) and the producer executes the `producer` function - defined in lines 11-26. Threads are created by the `main` function (lines 48-49).

The producer products integer values from range from 0 for 99. It uses a random number generator (line 16). Values are stored in the queue. The queue is represented by an array which is declared as global variable (line 8). The tail and the head of the queue is represented by two global variables - `tail` and `head` - declared in line 9. If produced value is equal 0 then the producer ends. The consumer consumes all values and ends work (line 30-39) after it reads 0 from the queue.

Number of products and number of free places in the queue are represented by semaphores - `products` and `empty`. Semaphores are declared as global variables (line 10). The `main` function initializes these semaphores. The `products` semaphore represents products in the queue and is initialized with zero (line 47) value because at the beginning the queue is empty. The `empty` semaphore represents free places in the queue and is initialized with value equal to the size of queue (line 46).

The producer can insert new value to the queueu (line 19) only if it is not full. It executes the `P` operation on the `empty` semaphore (line 18) before it inserts value to the queue. When new product is inserted to the queue the producer executes the `V` operation on the `products` semaphore (line 21) signaling, that number of product in the queue has accrued. After production the consumer executes the `sleep` function (line 22), that suspends execution of a thread for a given period time. The producer ends its work when the produced value is equal to 0. It does not return any value.

The consumer is analogous to the producer. It can take a product from the queue only if it is not empty. It executes the `P` operation on the `products` semaphore (line 32) before it takes a product from the queue (line 33). After that it signals that number of free places in the queue has accured, executing the `V` operation on the `empty` semaphore (line 36). The time of consumption is a random value from range from 0 for 4. The product's consumption is realized by calling the `sleep` function (line 37). The consumer ends his work when the consumed value is 0. It does not return any value.

The `main` function waits for the consumer and the producer threads termination (lines 50-51). After that it destroys a semaphores (lines 52-53) and terminates himself (line 54).

```
 1:  #include <stdio.h>
 2:  #include <stdlib.h>
 3:  #include <unistd.h>
 4:  #include <time.h>
 5:  #include <pthread.h>
 6:  #include <semaphore.h>

 7:  #define SIZE 10    /* size of the queue */
 8:  int queue[SIZE];   /* queue of products (values) */
 9:  int tail, head;    /* tail and head of the queue */
     /* semaphores representing number of free places and products */
10:  sem_t empty, products;

11:  void *producer(void *arg)      /* producer function */
12:  {
13:    int value;
14:    do      /* produce a values (products) until produced value is 0 */
15:    {
16:      int value = rand() % 100;  /* generate a value */
17:      printf("produced: %d\n", value);
18:      sem_wait(&empty);          /* wait for free place in the queue */
19:      queue[tail] = value;       /* insert new value to the queue */
20:      tail = (tail + 1) % SIZE;
21:      sem_post(&products);       /* increment number of products in the queue */
22:      sleep(rand() % 5);         /* rest after production */
23:    }
24:    while(value);
25:    pthread_exit(NULL);          /* terminate producer thread */
26:  }

27:  void *consumer(void *arg)      /* consumer function */
28:  {
29:    int value;
30:    do      /* consume a values (products) until consumed value is 0 */
31:    {
32:      sem_wait(&products);  /* wait for products in the queue */
33:      value = queue[head];  /* take a value from the queue */
34:      head = (head + 1) % SIZE;
35:      printf("consumed: %d\n", value);
36:      sem_post(&empty);         /* increment number of free places in the queue */
37:      sleep(rand() % 5);        /* consume taken product */
38:    }
39:    while(value);
40:    pthread_exit(NULL);         /* terminate producer thread */
41:  }
```

```
42:   int main(int argc, char *argv[])
43:   {
        /* descriptors of threads od producer and consumer */
44:     pthread_t prod_t, cons_t;

45:     srand(time(0));             /* initialize a random number generator */

        /* initialize a semaphores */
46:     sem_init(&empty, 0, SIZE);
47:     sem_init(&products, 0, 0);

        /* create producer and consumer threads */
48:     pthread_create(&prod_t, NULL, producer, NULL);
49:     pthread_create(&cons_t, NULL, consumer, NULL);

        /* wait for terminate producer and consumer threads */
50:     pthread_join(prod_t, NULL);
51:     pthread_join(cons_t, NULL);

        /* destroy a semaphores */
52:     sem_destroy(&empty);
53:     sem_destroy(&products);

54:     return 0;
55:   }
```

1.6 Condition variables

A condition variable is similar to a semaphore but there are important differences in definition and the use of it. It can be used to suspend execution of threads and relinquish the CPU until some condition on shared data is satisfied. It is also used to resume a suspended thread when the condition becomes true. A condition variable is a queue of suspended threads but this queue is not visible to the programmer. There are two operations on a condition variable:

- signal the condition - when the condition becomes true, it awakens one of a suspended threads from a queue,
- wait for the condition - suspending a thread until another thread singnals a condition variable; suspended thread is inserted to a queue.

1.6.1 Creating a condition variable

A condition variable is represented by variable of type `pthread_cond_t`. It must always

be associated with a mutex, to avoid the race condition where a thread prepares to wait on a condition variable and another thread signals the condition just before the first thread actually waits on it. A thread should lock mutex before using condition variable.

A condition variable must be initialized before using. It is initialized by calling function:

```
int pthread_cond_init(pthread_cond_t *cond,
                      const pthread_condattr_t *attr);
```

where:

- `cond` - a pointer to a condition variable to be initialized,
- `attr` - the conditions variable attributes, if it is `NULL` then default attributes are used[4].

If success the function returns 0 and return non-zero value on error.

1.6.2 Using a condition variable

We may awake one of the threads that are waiting on the condition variable by calling function:

```
int pthread_cond_signal(pthread_cond_t *cond).
```

If several threads are waiting, only one is resumed, but it is not specified which one. Nothing happens if no threads are waiting on the condition variable. All threads that are waiting on the condition variable may be restarted in one function:

```
int pthread_cond_broadcast(pthread_cond_t *cond).
```

Nothing happens if no threads are waiting on the condition variable.

If we want to execute a wait operation we should use an additional mutex. A mutex should be locked before calling a wait operation. A wait operation on a condition variable is implemented by function:

```
int pthread_cond_wait(pthread_cond_t *cond, pthread_mutex_t *mutex).
```

The function suspends calling thread and release a mutex. When a thread resumes execution after a signal or broadcast operation, a thread will again own a mutex and it will be locked.

All functions for using condition variable return 0 if success. Any other returned value indicates that an error occurred.

[4] A condition variable with default attributes suffices to realize tasks in the laboratory exercises.

1.6.3 Destroying a condition variable

A condition variable is destroyed by calling following function:

```
int pthread_cond_destroy(pthread_cond_t *cond).
```

If success the function returns 0, otherwise it returns non-zero value.

1.6.4 Sample code

In section 1.5.4 we presented a solution of a producer and consumer problem using semaphores. In this section we present solution of this problem using condition variables. Additional, we present how to use condition variables to create an object which has monitor's properties.

1.6.4.1 Monitors

Semaphores described in section 1.5 are low-level mechanism for mutual exclusion. It is easy to make a deadlock using semaphores. For example if one thread executed a `P` operation and waits on the semaphore, another thread must execute a `V` operation on the same semaphore to unlock the first thread. Otherwise the first thread never resumes. A higher level solution would make implementing synchronization a little easier. It is implemented by monitors.

The monitor was introduced by C.A.R. Hoare in paper ,,*An Operating System Structuring Concept*" in the 1974. It encapsulates the representation of an abstract object and provides a set of operations on this object. A monitor is a program module that consists of variables that store the object's state and functions that implement operations on the object. A state of object can be changed only by the monitor's functions. It follows that monitor's variables are accessed only by calling monitor's functions. Functions in the same monitor cannot be executed cuncurrently. This is what allows monitors to enforce mutual exclusion. A monitor has following properties:

- only functions specified in the monitor are visible outside the monitor,
- the monitor's funtions may not access variables declared outside monitor,
- variables are initialized before any funtion is called by executing the initialization funtion when the monitor is created.

Synchronization in the monitor is performed by condition synchronization. Condition synchronization is the problem of delaying a thread until a given condition is true. It may be implemented by condition variables.

The Pthread library does not implement monitors, but we present how to create an object, which has a monitor properties, using condition variables.

1.6.4.2 Monitor for the producer and the consumer problem

In this section we present the monitor for the producer and the consumer problem. Certainly it is only a program module but not a process. It is described in a pseudocode similar to C language. The monitor contains the queue of values and makes possible to insert a new value to this queue and take a value from this queue. We do not specify a type of values of the queue. Is is not necessary for understanding this problem. The queue can include values of any types.

```
 1:   monitor producer_consumer
 2:   {
 3:     private:
 4:       const int SIZE = ...;
 5:       type_of_value queue[SIZE];
 6:       int tail, head;
 7:       condition not_full, not_empty;
 8:     public:
 9:       void producer_consumer()         /* initialize a monitor */
10:       {
11:         tail = head = 0;
12:       }
13:       void put(type_of_value value)    /* insert a new value to the queue */
14:       {
15:         if (queue is full)
16:           not_full.wait();
17:         queue[tail] = value;
18:         tail = (tail + 1) mod SIZE;
19:         not_empty.signal();
20:       }
21:       type_of_value get()             /* take a value from the queue */
22:       {
23:         type_of_value value;
24:         if (queue is empty)
25:           not_empty.wait();
26:         value = queue[head];
27:         head = (head + 1) mod SIZE;
28:         not_full.signal();
29:         return value;
30:       }
31:   }
```

Presented monitor contains the following variables: the queue for values (line 5) and variables represent the tail and the head of the queue (line 6). These variables are accessible only inside the monitor. Additonally the monitor contains two condition variables describes

situation when the queue is not empty and not full (line 7). These variables make possible to delay a thread using this monitor when the queue is empty or full. The monitor contains a constant which represents size of the queue, too.

In the producer - consumer problem we have only two operations: the producer inserts a new value to the queue, the consumer takes a value from the queue. These operations are implemented by monitor's functions - `put` (lines 13-20) and `get` (lines 21-30). These functions are visible outside the monitor.

The `put` function (lines 13-20) inserts a new value to the queue if there is a free place. When the queue is full a thread waits for place (line 16). For this purpose a `not_full` condition variable is used. Instruction in line 16 should be understand as ,,wait until the queue is not full". It is equivalent with ,,wait for free place in the queue". After inserting a new value to the queue it is signaled that ,,the queue is not empty" (line 19). There is used a `not_empty` condition variable.

The consumer can take a value from the queue using the `get` function (lines 21-30). It is analogous to the `put` function. When the queue is empty it waits for a value in the queue (line 25). There is used a `not_empty` condition variable. Line 16 should be understand as ,,wait until the queue is not empty" or ,,wait for a value in the queue". After takinig a value from the queue it is signaled using a `not_full` condition variable (line 28), that ,,the queue is not full".

We presented a monitor and now we can present processes of the producer and the consumer. Processes will be described by functions.

```
 1:   producer_consumer m;    /* object represents the monitor */
 2:   void producer()         /* the producer's process */
 3:   {
 4:     type_of_value value;
 5:     while (true)
 6:     {
 7:       value = produce_new_value();
 8:       m.put(value);       /* insert produced value to the queue */
 9:     }
10:   }
11:   void consumer()         /* the consumer's process */
12:   {
13:     type_of_value value;
14:     while (true)
15:     {
16:       value = m.get();    /* take a value from the queue */
17:       consume_value(value);
18:     }
19:   }
```

Processes of the producer and the consumer never end. They share a monitor defined in line 1. The producer products a new value (line 7) and inserts it to the queue (line 8) using monitor's function `put`. The consumer consumes values produced by the producer. It takes a value from the queue using monitor's function `get`.

In next section we will present implementation of described monitor using the Pthread library and condition variables.

1.6.4.3 Implementation of the producer and the consumer monitor using the Pthread library and condition variables

```
 1:  #include <stdio.h>
 2:  #include <stdlib.h>
 3:  #include <unistd.h>
 4:  #include <time.h>
 5:  #include <pthread.h>

     /* class representing monitor for producer and consumer problem */
 6:  class monitor
 7:  {
 8:    static const int SIZE = 10;   /* size of queue */
 9:    int queue[SIZE];              /* queue of products (values) */
10:    int tail, head;               /* tail and head of the queue */
11:    int nr_of_prod;               /* number of products in the queue */
12:    pthread_cond_t not_full;      /* used to suspend thread when queue is full */
13:    pthread_cond_t not_empty;     /* used to suspend thread when queue is empty */
14:    pthread_mutex_t mutex;        /* associated with condition variables */
15:  public:
16:    monitor();
17:    ~monitor();
18:    void put(const int);
19:    int get();
20:  };

     /* constructor of monitor's object */
21:  monitor::monitor()
22:  {
23:    nr_of_prod = tail = head = 0;      /* the queue is empty */

       /* initialize a mutex and condition variables with default attributes */
24:    pthread_mutex_init(&mutex, NULL);
25:    pthread_cond_init(&not_full, NULL);
26:    pthread_cond_init(&not_empty, NULL);
27:  }
```

```
      /* destructor of monitor's object */
28:   monitor::~monitor()
29:   {
        /* destroy a mutex and conditional variables */
30:     pthread_mutex_destroy(&mutex);
31:     pthread_cond_destroy(&not_full);
32:     pthread_cond_destroy(&not_empty);
33:   }
      /* insert a new value to the queue */
34:   void monitor::put(const int value)
35:   {
36:     pthread_mutex_lock(&mutex);        /* lock a mutex */

        /* the queue is full - wait for free place in the queue */
37:     if (nr_of_prod == SIZE)
38:       pthread_cond_wait(&not_full, &mutex);

39:     queue[tail] = value;               /* insert new value to the queue */
40:     nr_of_prod++;
41:     tail = (tail + 1) % SIZE;

42:     pthread_cond_signal(&not_empty);   /* signal products in the queue */
43:     pthread_mutex_unlock(&mutex);      /* release a mutex */
44:   }

      /* return a value from the queue */
45:   int monitor::get()
46:   {
47:     pthread_mutex_lock(&mutex);         /* lock a mutex */

        /* the queue is empty - wait for products in the queue */
48:     if (nr_of_prod == 0)
49:       pthread_cond_wait(&not_empty, &mutex);

50:     int value = queue[head];            /* take a value from the queue */
51:     nr_of_prod--;
52:     head = (head + 1) % SIZE;

53:     pthread_cond_signal(&not_full);     /* signal free place in the queue */
54:     pthread_mutex_unlock(&mutex);       /* release a mutex */
55:     return value;                       /* return a value taken from the queue */
56:   }

57:   monitor prod_cons_mon;         /* monitor used for producer and consumer problem */
```

```
58:  void *producer(void *arg)      /* producer function */
59:  {
60:    int value;
61:    do      /* produce values (products) until produced value is non-zero */
62:    {
63:      value = rand() % 100;      /* produce a value */
64:      printf("produced: %d\n", value);
65:      prod_cons_mon.put(value);  /* insert produced value to the queue */
66:      sleep(rand() % 5);         /* rest after production */
67:    }
68:    while(value);
69:   pthread_exit(NULL);           /* terminate producer thread */
70:  }

71:  void *consumer(void *arg)          /* consumer function */
72:  {
73:    int value;
74:    do    /* consume values (products) until consumed value is non-zero */
75:    {
76:      value = prod_cons_mon.get();   /* get a first value from the queue */
77:      printf("consumed: %d\n", value);
78:      sleep(rand() % 5);             /* consume taken product */
79:    }
80:    while(value);
81:    pthread_exit(NULL);              /* terminate producer thread */
82:  }

83:  int main(int argc, char *argv[])
84:  {
       /* descriptors of threads of producer and consumer */
85:    pthread_t prod_t, cons_t;

86:    srand(time(0));             /* initialize a random number generator */

       /* create producer and consumer threads */
87:    pthread_create(&prod_t, NULL, producer, NULL);
88:    pthread_create(&cons_t, NULL, consumer, NULL);

       /* wait for terminate producer and consumer threads */
89:    pthread_join(prod_t, NULL);
90:    pthread_join(cons_t, NULL);
91:    return 0;
92:  }
```

Producer (lines 58-70) and consumer (lines 71-82) are implemented using separate threads like in section 1.5.4. The monitor is declared as global variable (line 57). Producer and consumer do not contain any synchronization instructions like in a sample presented, they only call a monitor's functions.

We defined a class representing a monitor (lines 6-20). The class contains the queue for products (line 9) and two functions - `put` (lines 34-44) and `get` (lines 45-56) - implementing suitable operations: inserting a new value to the queue and getting a first value from the queue. A value returned by the `get` function is removed from the queue. The queue can be accessed outside a monitor only using these functions.

A producer must wait if the queue is full and a consumer must wait if the queue is empty. Threads are suspended by means of condition variables. Variable `not_full` is used to suspend a producer (line 38) trying to insert a value to a full queue. Conditional variable `not_empty` is used to suspend a consumer (line 49) when the queue is empty. Additional a mutex is used (declared in line 14) associated with a conditional variables. The mutex is locked in the begining of the `put` (line 36) and `get` (line 47) funtions end released in the end of these functions (lines 43 and 54). It guarantees that functions `put` and `get` will not be execute concurrently.

The `not_empty` condition variable is signalled after inserting a new product to the queue (line 43). It is signallled because the queue contains at least a one value and consumer waiting for it should be resumed (line 49). And vice versa - after removed of a value from the queue, the `not_full` condition variable is signalled (line 53). There is at least one free place in the queue and we should resume suspended producer waiting for it (line 38).

A conditional variable and a mutex are initialized in the constructor of the object (lines 21-27). They are destroyed in the destructor (lines 28-33). The `main` function only creates producer and consumer threads (lines 87-88) and waits till are terminated (lines 89-90). A source code is presented below.

Bibliography

[1] Gregory R. Andrews *Foundations of Multithreaded, Parallel, and Distributed Programming*, Addison Wesley Longman 2000

[2] M. Ben-Ari *Principles of Concurrent and Distributed Programming*, Prentince Hall International Ltd 1990

[3] Brian W. Kernighan, Dennis M. Ritchie *The C Programming Language, Second Edition*, Prentice-Hall 1988

[4] `http://yolinux.com/TUTORIALS/LinuxTutorialPosixThreads.html`

[5] `http://www.humanfactor.com/pthreads/`

[6] `http://www.llnl.gov/computing/tutorials/pthreads/`

[7] `http://users.actcom.co.il/~choo/lupg/tutorials/multi-thread/multi-thread.html`

[8] `http://www-106.ibm.com/developerworks/linux/library/l-posix1.html`

[9] `http://www-106.ibm.com/developerworks/library/l-posix2/`

[10] `http://www-106.ibm.com/developerworks/library/l-posix3/`

[11] `http://www.opengroup.org/onlinepubs/7908799/xsh/threads.html`

[12] `http://docs.sun.com/app/docs/doc/801-6659`