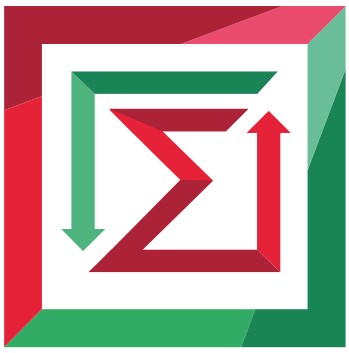
Министерство науки и высшего образования
Российской Федерации

Федеральное государственное бюджетное образовательное
учреждение высшего образования
«НОВОСИБИРСКИЙ ГОСУДАРСТВЕННЫЙ ТЕХНИЧЕСКИЙ УНИВЕРСИТЕТ»

НГТУ
НЭТИ

Кафедра теоретической и прикладной информатики

Лабораторная работа № 3
по дисциплине «Проектирование Систем Реального Времени»
Синхронизация потоков



ФАКУЛЬТЕТ:	ПМИ
ГРУППА:	ПМИМ-01
СТУДЕНТЫ:	Ершов П. К. Грициенко И. Г.
БРИГАДА:	7
ПРЕПОДАВАТЕЛЬ:	Кобылянский В. Г.

Новосибирск
2021

1. Цель работы

Целью работы является изучение механизмов синхронизации.

2. Задание на лабораторную

1. Скомпилировать и выполнить примеры программ. Проанализировать результаты выполнения.
2. Написать программу, реализующую задание в соответствии с вариантом:

№ варианта	Задание
7	На основе механизма условной переменной, разработать приложение подсчета контрольной суммы CRC16 файла. Один поток читает порциями данные из файла в общий буфер, второй поток подсчитывает контрольную сумму.

3. Ход работы.

3.1. Анализ работы программы-примера из пункта 2.4. работа с мутексом

Результаты выполнения кода-примера:

```
Thread<2> run jobs with parametr: 6
Thread<2>: 6!=720
Thread<2> run jobs with parametr: 3
Thread<2>: 3!=6
Thread<2> run jobs with parametr: 9
Thread<2>: 9!=362880
Thread<2> run jobs with parametr: 11
Thread<2>: 11!=39916800
Thread<2> run jobs with parametr: 5
Thread<2>: 5!=120
Thread<2> run jobs with parametr: 0
Thread<2>: 0!=1
Thread<2> run jobs with parametr: 0
Thread<2>: 0!=1
Thread<2> run jobs with parametr: 11
Thread<2>: 11!=39916800
Thread<2> run jobs with parametr: 1
Thread<2>: 1!=1
Thread<2> run jobs with parametr: 6
Thread<2>: 6!=720
Thread<2> run jobs with parametr: 0
Thread<2>: 0!=1
Thread<2> run jobs with parametr: 3
Thread<2>: 3!=6
Thread<2> run jobs with parametr: 6
Thread<2>: 6!=720
Thread<2> run jobs with parametr: 11
Thread<2>: 11!=39916800
Thread<2> run jobs with parametr: 7
Thread<2>: 7!=5040
Thread<2> run jobs with parametr: 7
Thread<2>: 7!=5040
Thread<2> run jobs with parametr: 9
Thread<2>: 9!=362880
Thread<2> run jobs with parametr: 10
Thread<2>: 10!=3628800
Thread<2> run jobs with parametr: 2
Thread<2>: 2!=2
Thread<3> run jobs with parametr: 9
Thread<3>: 9!=362880
```

Рисунок 1. Результат выполнения программы вычисляющей факториал с применением мутексов в работе

В главном потоке сначала выполняется функция jobs_producer(), которая формирует список задач (в данном случае, список из чисел, факториал которых нужно получить).

Затем создаются два потока с идентификаторами 2 и 3 соответственно. В этих потока выполняются функции process_job, которые вычисляют факториал числа, удаляя его из списка.

В силу того, что функции могут обратиться к одному ресурсу одновременно, что вызовет ошибку исполнения. Для решения этой проблемы можно использовать мутекс, который позволяет потоку блокировать участок кода, защищая его от исполнения другими потоками. Это гарантирует последовательное обращения к ресурсам.

3.2. Анализ работы программы-примера из пункта 2.4. работа с семафорами

```
Thread<3> run jobs with parametr: 2
Thread<3>: 2!=2
Thread<4> run jobs with parametr: 10
Thread<4>: 10!=3628800
Thread<3> run jobs with parametr: 9
Thread<3>: 9!=362880
Thread<4> run jobs with parametr: 7
Thread<4>: 7!=5040
Thread<3> run jobs with parametr: 7
Thread<3>: 7!=5040
Thread<4> run jobs with parametr: 11
Thread<4>: 11!=39916800
Thread<3> run jobs with parametr: 6
Thread<3>: 6!=720
Thread<4> run jobs with parametr: 3
Thread<4>: 3!=6
Thread<3> run jobs with parametr: 0
Thread<3>: 0!=1
Thread<4> run jobs with parametr: 6
Thread<4>: 6!=720
```

Рисунок 2. Результат выполнения программы вычисляющей факториал с применением семафоров в работе

Программа в главном потоке создаёт семафор функцией **sem_init**(&sem, 0, 0) с изначальным значение 0. Это означает, что семафор заблокирован в начале работы программы, а с логической точки зрения, что задач нет.

Затем инициализируется поток, в котором запускается функция **jobs_producer**(**void** *arg), в которой создаётся число, факториал которого нужно получить. Семафор после этого увеличивается на 1 функцией

sem_post(&sem), что в свою очередь означает наличие задач для других потоков. При этом участок кода, связанный со списком задач блокируется с помощью блокировки мутекса. Это делается для того, чтобы сразу после разблокирования семафора не произошла критическая ошибка.

После этого инициализируются потоки, в которых выполняется функция **thread_function**(**void** *arg), которая с помощью функции **sem_wait**(&sem) ждёт, когда семафор будет разблокирован. Как только семафор увеличивается на 1 потоком с функцией **jobs_producer**, поток **thread_function**, сразу же блокирует семафор, уменьшая его значение на 1. После этого участок кода, в котором происходит обработка списка задач блокируется мутексом, задача из массива задач удаляется, а функция получает факториал числа из задачи.

Вся программа работает 10 секунд, после чего завершается.

3.3. Анализ работы программы-примера из пункта 2.4. работа с условными переменными

```
Starting consumer/producer example...
In producer thread...
In consumer thread...
producer:  got data from h/w
consumer:  got data from producer
producer:  got data from h/w
consumer:  got data from producer
producer:  got data from h/w
consumer:  got data from producer
producer:  got data from h/w
consumer:  got data from producer
producer:  got data from h/w
consumer:  got data from producer
producer:  got data from h/w
consumer:  got data from producer
producer:  got data from h/w
consumer:  got data from producer
producer:  got data from h/w
consumer:  got data from producer
producer:  got data from h/w
consumer:  got data from producer
producer:  got data from h/w
consumer:  got data from producer
```

Рисунок 3. Результат выполнения программы, один поток которой производит данные, а другой потребляет

Программа создаёт в главном потоке два потока:

1. Поток выполнения функции producer, которая производит данные.
2. Поток выполнения функции consumer, которая обрабатывает (потребляет) данные.

Для блокировки потока используются мутекс. Producer создаёт данные, блокирует мутекс после чего функция **pthread_cond_wait()** блокирует поток исполнения producer и разблокирует мутекс и начинает ждать когда другой поток разблокирует поток выполнения producer на случайной переменной с помощью функции **pthread_cond_signal()**.

После того как функция **pthread_cond_wait()** разблокировала мутекс, функция consumer блокирует мутекс и ждёт когда появятся данные. Когда данные получены и обработаны, consumer функцией **pthread_cond_signal** разблокирует поток producer и разблокирует мутекс.

3.4. Программа успешно реализована

```
Starting consumer/producer example...
In producer thread file used TEST2
In consumer thread...
producer:  got TEST1

consumer:  checksum 45341 received TEST1

producer:  got TEST2

consumer:  checksum 58446 received TEST2

producer:  got TEST3

consumer:  checksum 55167 received TEST3

producer:  got TEST4
consumer:  checksum 60627 received TEST4
```

Рисунок 4. Результат выполнения программы по получению контрольной суммы CRC16

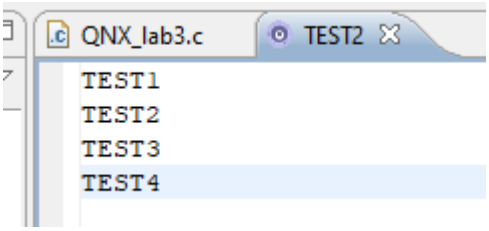


Рисунок 5. Содержимое используемого файла TEST2

4. Код программы

Код программы-примера использования мутекса

```
#include <stdlib.h>
#include <stdio.h>
#include <malloc.h>
#include <math.h>
#include <pthread.h>

#define N 20 // число заданий

// структура-элемент списка
struct job
{
	int n;                      // параметр задания
	struct job *next;  // указатель на следующее задание
};

struct job *job_queue;      // список заданий

// функция выполнения задания
void process_job(struct job *the_job)
{
	int i, M=0;
	long factorial=1;
	printf("Thread<%d> run jobs with parametr: %d\n", pthread_self(), the_job->n);
// здесь проводятся вычисления
// в данном случае вычисление факториала из числа
		// указанного в задании (число n).
	M=the_job->n;
	for(i=2; i<=M; i++)
	{
		factorial*=i;
	}
	printf("Thread<%d>: %d!=%ld\n", pthread_self(), M, factorial);
}

// функция потока, выполняющего задания
void *thread_function(void *arg)
{
	while(1)
	{
		struct job *next_job;
		// захватываем мутекс
		pthread_mutex_lock(&mutex);
		// в критической секции монопольно работаем со списком заданий
		if (job_queue==NULL)
			next_job=NULL;
		else
		{
			next_job=job_queue;
			// переводим указатель на следующее задание
			job_queue=job_queue->next;
		}
		pthread_mutex_unlock(&mutex);
		// на выходе из критической секции
		// получаем указатель на текущее задание (next_job)
		if (next_job==NULL)
			break;
		process_job(next_job);
		free(next_job);
	}
	return NULL;
}


// функция-генератор заданий
void *jobs_producer(void *arg)
{
	int i;
	struct job *point;
	for(i=N; i>0; i--)
	{
		// создаем список из N заданий
		point=(struct job*)malloc(sizeof(struct job));
		point->n=rand()%12;
		point->next=job_queue;
		job_queue=point;
	}
	return NULL;
}

int main(int argc, char *argv[])
```

```
{
        pthread_t first_tid, second_tid;
        // генерируем задания
        jobs_producer(NULL);
        // и создаем два потока для их выполнения
        pthread_create(&first_tid, NULL, thread_function, NULL);
        pthread_create(&second_tid, NULL, thread_function, NULL);

        // ждем завершения потоков, которые выполняют задания
        pthread_join(first_tid, NULL);
        pthread_join(second_tid, NULL);
        return EXIT_SUCCESS;
}
```

Код программы-примера использования семафоров

```
#include <stdlib.h>
#include <stdio.h>
#include <malloc.h>
#include <math.h>
#include <pthread.h>
#include <semaphore.h>

struct job
{
        int n;
        struct job *next;
};

struct job *job_queue;

pthread_mutex_t mutex=PTHREAD_MUTEX_INITIALIZER;

// объявляем семафор для блокирования потоков-исполнителей,
// если нет заданий в списке
sem_t sem;

// функция выполнения задания
void process_job(struct job *the_job)
{
        int i, M=0;
        long factorial=1;
        printf("Thread<%d> run jobs with parametr: %d\n", pthread_self(), the_job->n);
// здесь проводятся вычисления
// в данном случае вычисление факториала из числа, указанного в задании (число n).
        M=the_job->n;
        for(i=2; i<=M; i++)
        {
                factorial*=i;
        }
        printf("Thread<%d>: %d!=%ld\n", pthread_self(), M, factorial);
}

void *thread_function(void *arg)
{
        while(1)
        {
                struct job *next_job;
                // уменьшаем на единицу счетчик семафора.
                // если счетчик равен нулю, то поток
                // будет ждать увелечения значения счетчика
                sem_wait(&sem);
                pthread_mutex_lock(&mutex);
                // входим в критическую секцию
                // и монопольно действуем со списком заданий -
                // удаляем одно задание из списка
                next_job=job_queue;
                job_queue=job_queue->next;
                pthread_mutex_unlock(&mutex);
                process_job(next_job);
                free(next_job);
        }
        return NULL;
}
// функция-генератор заданий
// задания генерируется во время работы
void *jobs_producer(void *arg)
{
        struct job *point;
        while(1)
        {
                // создаем элемент-задание
                point=(struct job*)malloc(sizeof(struct job));
                point->n=rand()%12;
                // входим в критическую секцию
```

```
                // для записи задания в список
                pthread_mutex_lock(&mutex);
                point->next=job_queue;
                job_queue=point;
                // увеличиваем на единицу счетчик семафора,
                // тем самым сообщая, что появилось задание
                sem_post(&sem);
                pthread_mutex_unlock(&mutex);
                sleep(1);// задания генерируется с задержкой
        }
        return NULL;
}

int main(int argc, char *argv[])
{
        // в начале список заданий пуст
        job_queue=NULL;
        // инициализируем счетчик семафора значением ноль
        sem_init(&sem, 0, 0);
        pthread_create(NULL, NULL, jobs_producer, NULL);
        pthread_create(NULL, NULL, thread_function, NULL);
        pthread_create(NULL, NULL, thread_function, NULL);

        // подождем 10 секунд и завершим приложение
        sleep(10);
        return EXIT_SUCCESS;
}
```

Код программы-примера использования условных переменных

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

// флаг готовности данных
int data_ready=0;
//статически инициализируем мутекс и условную переменную
pthread_mutex_t mutex=PTHREAD_MUTEX_INITIALIZER;
pthread_cond_t condvar=PTHREAD_COND_INITIALIZER;

// этот поток обрабатывает данные - потребитель
void *consumer(void *notused)
{
        printf("In consumer thread...\n");
        while(1)
        {
                pthread_mutex_lock(&mutex);
                while(!data_ready)
                {
                        // если нет данных, то ждем, пока они появятся
                        pthread_cond_wait(&condvar, &mutex);
                }
                printf("consumer:  got data from producer\n");
                // здесь может быть обработка данных
                // это имитируется функцией sleep()
                sleep(1);
                // устанавливаем флаг, что нет необработанных данных
                data_ready=0;
                // разблокируем производителя данных
                pthread_cond_signal(&condvar);
                pthread_mutex_unlock(&mutex);
        }
}

// этот поток получает данные - производитель
void *producer(void *notused)
{
        printf("In producer thread...\n");
        while(1)
        {
                // получаем данные из устройства
                // это имитируется функцией sleep()
                sleep(1);
                printf("producer:  got data from h/w\n");
                pthread_mutex_lock(&mutex);
                while(data_ready)
                {
                        // если есть данные, то ждем, пока они обработаются
                        pthread_cond_wait(&condvar, &mutex);
                }
                // устанавливаем флаг, что есть необработанные данные
                data_ready=1;
                // разблокируем потребителя данных
                pthread_cond_signal(&condvar);
```

```
                pthread_mutex_unlock(&mutex);
        }
}
int main()
{
        printf("Starting consumer/producer example...\n");

        // создаем потоки производителя и потребителя данных
        pthread_create(NULL, NULL, producer, NULL);
        pthread_create(NULL, NULL, consumer, NULL);

        // подождем немного и завершим приложение
        sleep(10);
        return 0;
}
```

Код разработанной программы

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <pthread.h>
#include <sys/neutrino.h>
#include <errno.h>

#define FILENAME_MAX 255
#define N 1024
// флаг готовности данных
int data_ready=0;
//статически инициализируем мутекс и условную переменную
pthread_mutex_t mutex=PTHREAD_MUTEX_INITIALIZER;
pthread_cond_t condvar=PTHREAD_COND_INITIALIZER;

char buff[FILENAME_MAX]; // главный массив

unsigned short crc16(const unsigned char* data_p, unsigned short length) // функция получения контрольной суммы
{
    unsigned char x;
    unsigned short crc = 0xFFFF;

    while (length--){
        x = crc >> 8 ^ *data_p++;
        x ^= x>>4;
        crc = (crc << 8) ^ ((unsigned short)(x << 12)) ^ ((unsigned short)(x <<5)) ^ ((unsigned short)x);
    }
    return crc;
}


// этот поток обрабатывает данные - потребитель
void *consumer(void *notused)
{
        printf("In consumer thread...\n");
        while(1)
        {
                pthread_mutex_lock(&mutex);
                while(!data_ready)
                {
                        // если нет данных, то ждем, пока они появятся
                        pthread_cond_wait(&condvar, &mutex);
                }
                unsigned char *conbuf = (unsigned char *)buff;
                unsigned short lenght = (unsigned short)strlen(conbuf);
                unsigned short crc16Summ = crc16(conbuf, lenght);
                printf("consumer:  checksum %i received %s\n", (int)crc16Summ, buff);
                // здесь может быть обработка данных
                sleep(1);
                // устанавливаем флаг, что нет необработанных данных
                data_ready=0;
                // разблокируем производителя данных
                pthread_cond_signal(&condvar);
                pthread_mutex_unlock(&mutex);
        }
}

// этот поток получает данные - производитель
void *producer(void *notused)
{
        int i = 0;
        char b[N];
        char *ch = (char *) malloc(sizeof (char) * N);
        char fileName[FILENAME_MAX];
```

```
        FILE *fl;

        strcpy(fileName, notused);

        printf("In producer thread file used %s\n", fileName);

        if ((fl = fopen(fileName, "r")) == NULL)
        {
                printf("Невозможно открыть файл для чтения.\n");
                exit(1);
        }

        while(!feof(fl))
        {
                sleep(1);

                fgets(b, N, fl);
                if (b[0] != '\n' )
                        strcpy(ch, b);

                strcpy(buff, ch);
                printf("producer:  got %s\n", buff);
                pthread_mutex_lock(&mutex);

                while(data_ready)
                {
                        // если есть данные, то ждем, пока они обработаются
                        pthread_cond_wait(&condvar, &mutex);
                }
                sleep(1);
                // устанавливаем флаг, что есть необработанные данные
                data_ready = 1;
                // разблокируем потребителя данных
                pthread_cond_signal(&condvar);
                pthread_mutex_unlock(&mutex);
                i++;
        }
}

int main()
{
        printf("Starting consumer/producer example...\n");

        // создаем потоки производителя и потребителя данных
        char *file_name = "TEST2";
        pthread_create(NULL, NULL, producer, (void*)file_name);
        pthread_create(NULL, NULL, consumer, NULL);

        // подождем немного и завершим приложение
        sleep(10);
        return 0;
}
```