

Министерство науки и высшего образования Российской Федерации  
Федеральное государственное бюджетное образовательное учреждение  
высшего образования  
«Ульяновский государственный технический университет»  
Кафедра «Информатика и вычислительная техника»

## **Операционные системы**

(название дисциплины)

### **Лабораторная работа №4**

## **«Проектирование пула потоков»**

(название (тема) работы)

Выполнил:  
студент группы ИВТАПбд-31  
Молчанов А. В.  
(Фамилия И. О.)

Проверил:  
преподаватель, аспирант кафедры «ИВТ»  
(должность)  
Беляев К. С.  
(Фамилия И. О.)

Ульяновск

2023

## Постановка задачи

Данная лабораторная работа включает в себя создание пула потоков с использованием Pthreads API и управление им, а также использование мьютексов POSIX и семафоров для синхронизации.

### Описание клиента:

- *void pool\_init()* – инициализирует пул потоков;
- *int pool\_submit(void (\*somefunction)(void \*p), void \*p)*, где *somefunction* – это указатель на функцию, которая будет выполняться потоком из пула, а *p* – это параметр, передаваемый функции;
- *void pool\_shutdown(void)* – выключает пул потоков после завершения всех задач.

### Описание пула потоков:

1. Функция *pool\_init()* создаст потоки при запуске, а также инициализирует блокировки взаимного исключения (mutex) и семафоры;
2. Функция *pool\_submit()* частично реализована и в настоящее время помещает выполняемую функцию, а также её данные в структуру задачи. Структура задачи представляет работу, которая будет выполнена потоком в пуле. *pool\_submit()* добавит эти задачи в очередь, вызывая функцию *enqueue()*, а рабочие потоки вызовут *dequeue()* для получения работы из очереди. Очередь может быть реализована статически (с использованием массивов) или динамически (с использованием связного списка). Функция *pool\_init()* имеет возвращаемое значение *int*, которое используется для указания того, была ли задача успешно отправлена в пул (0 указывает на успех, 1 указывает на неудачу). Если очередь реализована с использованием массивов, *pool\_init()* вернёт 1, если будет попытка отправить работу и очередь заполнена. Если очередь реализована как

связный список, *pool\_init()* пула всегда должен возвращать 0, если только не произойдёт ошибка выделения памяти;

3. Функция *worker()* выполняется каждым потоком в пуле, где каждый поток будет ожидать доступной работы. Как только работа станет доступной, поток удалит её из очереди и вызовет метод *execute()* для запуска указанной функции. Семафор можно использовать для уведомления ожидающего потока, когда работа передаётся в пул потоков. Могут использоваться как именованные, так и безымянные семафоры;
4. Блокировка мьютекса необходима во избежание состояния гонки при доступе или изменении очереди;
5. Функция *pool\_shutdown()* отменит каждый рабочий поток, а затем будет ждать завершения каждого потока, вызывая *pthread\_join()*. (Операция семафора *sem\_post()* – это точка отмены, которая позволяет отменить поток, ожидающий семафора).

## Детали реализации

Заголовочный файл `threadpool.h` содержит в себе такие необходимые для корректной работы программы строки кода как: объявление константных переменных длины очереди поступающих на выполнение пулом потоков задач (`QUEUE_SIZE`) и общее количество потоков в пуле (`NUMBER_OF_THREADS`).

Также, в данном файле объявлены заголовки основных функций, необходимых для реализации пула потоков, описание функционала которых представлено на Рисунке 1.

```
C threadpool.h x
C threadpool.h > ...
1  #ifndef _THREAD_POOL_
2  #define _THREAD_POOL_
3
4  #define QUEUE_SIZE 10
5  #define NUMBER_OF_THREADS 3
6
7  // Рабочая область пула потоков
8  void *worker(void *args);
9
10 // Выполняет (вызывает) задачу (функцию)
11 void execute(void (*function)(void *function_args), void *data);
12
13 // Инициализирует пул потоков
14 void pool_init(void);
15
16 // Отправляет задачу function с её данными data в пул (очередь) потоков
17 int pool_submit(void (*function)(void *function_args), void *data);
18
19 // Завершает работу пула потоков
20 void pool_shutdown(void);
21
22 #endif
23
```

Рисунок 1. Фрагмент кода №1

Файл `threadpool.c` является основным файлом данной лабораторной работы и содержит в себе реализацию функций из заголовочного файла `threadpool.h` описанного выше.

Задача представляет из себя структуру данных, состоящую из вызываемой функции и данных, которые будут передаваться в качестве её аргумента. Массив подобных задач и служит очередью.

Другие глобальные переменные (Рисунок 2) – это массив потоков из библиотеки `pthread`, мьютекс и две обёрточные функции над ним для выявления и отладки ошибок, которые могут возникнуть при его блокировке и разблокировке, а также семафор из библиотеки `semaphore`.

```
C threadpool.c x
C threadpool.c > ...
1  #include "assert.h"
2  #include "threadpool.h"
3  #include "pthread.h"
4  #include "semaphore.h"
5  #include "stdbool.h"
6
7  // Структура задачи (функции), которая должна быть выполнена потоком из пула
8  typedef struct
9  {
10     void (*function)(void *function_args); // Задача
11     void *data; // Её данные
12 } task;
13
14 task task_queue[QUEUE_SIZE];
15 int last_added_task_index = 0;
16
17 pthread_t threads[NUMBER_OF_THREADS];
18 pthread_mutex_t mutex; // = PTHREAD_MUTEX_INITIALIZER;
19 sem_t semaphore;
20
21 bool work_is_done = false;
22
23 void Pthread_mutex_lock(pthread_mutex_t *mutex)
24 {
25     int result_code = pthread_mutex_lock(mutex);
26     assert(result_code == 0);
27 }
28
29 void Pthread_mutex_unlock(pthread_mutex_t *mutex)
30 {
31     int result_code = pthread_mutex_unlock(mutex);
32     assert(result_code == 0);
33 }
34
```

Рисунок 2. Фрагмент кода №2

Работа пула потоков начинается с его инициализации в функции `pool_init`, где также происходит инициализация мьютекса и семафора.

Создание самих потоков происходит в цикле `for` библиотечной функцией `pthread_create`, в которую в качестве аргументов передаются объект потока `pthread_t` и функция `worker`, которую этот поток будет выполнять. Данная функция симулирует работу потока до тех пор, пока все задачи из очереди не будут выполнены и новых не будет поступать или, пока его работа не будет принудительно остановлена вызовом функции `pool_shutdown`.

Работая, потоки ожидают сигнала семафора о том, что появилась доступная работа, то есть задача была добавлена в очередь функцией `enqueue` вызовом обёрточной над ней функцией `pool_submit` с клиента (файл `client.c`).

Получив сигнал, один из доступных потоков извлекает только что поступившую задачу из очереди вызовом функции `dequeue`, проверяет, что она содержит в себе данные и выполняет её вызовом функции `execute`. Так как задача – это функция, то выполнение задачи заключается в вызове функции с аргументами в виде передаваемых в неё данных. Именно это и происходит в функции `execute`.

Вся описанная выше логика начала работы пула потоков приведена на Рисунке 3.

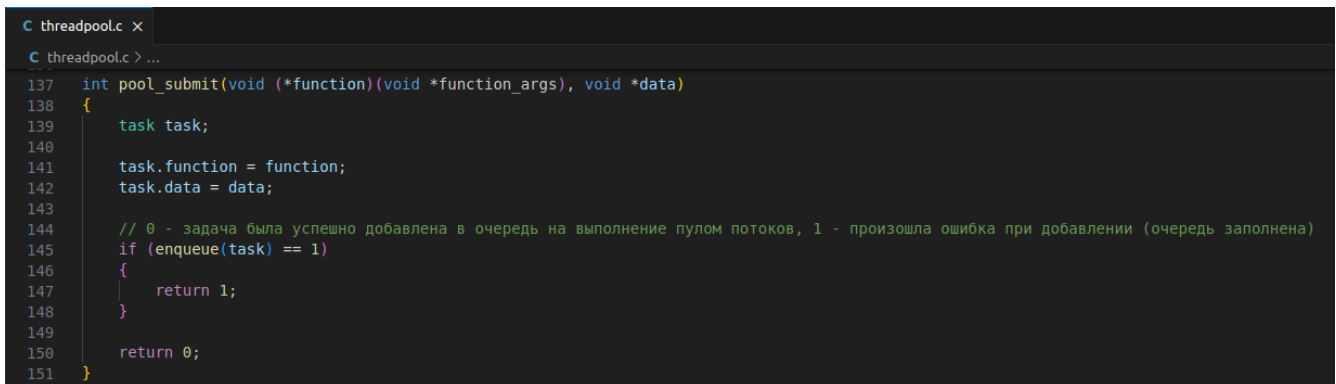
```

C threadpool.c x
C threadpool.c > ...
95 void *worker(void *args)
96 {
97     while (!work_is_done)
98     {
99         // Поток ожидает доступную работу
100         sem_wait(&semaphore);
101
102         task execute_task = dequeue();
103
104         if (execute_task.data == NULL)
105         {
106             work_is_done = true;
107             break;
108         }
109
110         execute(execute_task.function, execute_task.data);
111     }
112
113     // Завершаем работу вызвавшего данную функцию потока
114     pthread_exit(0);
115 }
116
117 void execute(void (*function)(void *function_args), void *data)
118 {
119     (*function)(data);
120 }
121
122 void pool_init(void)
123 {
124     int return_code = pthread_mutex_init(&mutex, NULL);
125     assert(return_code == 0); // 0 - success, 1 - error
126
127     // 0 - семафор будет разделяться между потоками внутри одного процесса, 0 - его начальное значение
128     sem_init(&semaphore, 0, 0);
129
130     for (int thread = 0; thread < NUMBER_OF_THREADS; ++thread)
131     {
132         // threads[thread] - один из трёх потоков, worker - функция, которую они все будут выполнять
133         pthread_create(&threads[thread], NULL, worker, NULL);
134     }
135 }

```

Рисунок 3. Фрагмент кода №3

Функция `pool_submit` (Рисунок 4) представляет из себя API-функцию, которую можно вызвать с клиента, передав ей в качестве аргументов задачу (функцию и данные), из которых будет сформирован объект структуры `task`, который впоследствии будет передан на добавление в очередь функцией `enqueue` (Рисунок 5).



```
C threadpool.c x
C threadpool.c > ...
137 int pool_submit(void (*function)(void *function_args), void *data)
138 {
139     task task;
140
141     task.function = function;
142     task.data = data;
143
144     // 0 - задача была успешно добавлена в очередь на выполнение пулом потоков, 1 - произошла ошибка при добавлении (очередь заполнена)
145     if (enqueue(task) == 1)
146     {
147         return 1;
148     }
149
150     return 0;
151 }
```

Рисунок 4. Фрагмент кода №4

Данная функция позволяет добавить задачу в очередь на выполнение пулом потоков и содержит в себе критическую секцию, так как запросы с клиента могут приходить в любое время. В связи с этим, добавление поступившей задачи происходит с захватом блокировки одним из доступных потоков во время которой он может последовательно заполнить массив `task_queue` новыми данными, а не перезаписать уже имеющиеся из-за условий гонки (*race conditions*), которая могла бы возникнуть между потоками при отсутствии мьютекса.

После добавления новой задачи в очередь она становится доступной и ожидающие работы потоки должны быть уведомлены об этом с использованием механизма семафора и его функции `sem_post` для увеличения его значения на 1. Один из ожидающих потоков как бы забирает эту единицу себе вызовом функции `sem_wait` и выходит из состояния ожидания, продолжая свою работу.



```

C threadpool.c x
C threadpool.c > ...
35 // Добавляет задачу в очередь на выполнение пулом потоков
36 int enqueue(task enqueueing_task)
37 {
38     if (last_added_task_index < QUEUE_SIZE)
39     {
40         // Критическая секция
41
42         Pthread_mutex_lock(&mutex);
43
44         while (task_queue[last_added_task_index].function != NULL && task_queue[last_added_task_index].data != NULL && last_added_task_index < QUEUE_SIZE)
45         {
46             ++last_added_task_index;
47         }
48
49         task_queue[last_added_task_index] = enqueueing_task;
50         sem_post(&semaphore); // Уведомляем один из ожидающих потоков, что работа была передана в пул на выполнение (стала доступной)
51
52         Pthread_mutex_unlock(&mutex);
53
54         //
55
56         return 0;
57     }
58
59     return 1;
60 }
61

```

Рисунок 5. Фрагмент кода №5

Для извлечения задачи из очереди потоки используют функцию `dequeue` (Рисунок 6), которая по той же причине, что и `enqueue` содержит в себе критическую секцию с блокировкой мьютекса. После извлечения задачи в отдельную переменную под названием `dequeuing_task`, её объект зануляется NULL-значениями, происходит проверка на извлечение последней задачи (в этом случае работа считается выполненной и программу можно завершить вызовом функции `pool_shutdown`) и разблокировка мьютекса захватившим его потоком.

Циклы `while` в обеих функциях (Рисунки 5-6) необходимы для избежания ситуации перезатирания уже имеющихся в очереди задач и их последовательного добавления.

```

C threadpool.c x
C threadpool.c > ...
62 // Извлекает (удаляет) задачу из очереди на выполнение пулом потоков и передаёт её на выполнение одному из них
63 task dequeue()
64 {
65     // Критическая секция
66
67     Pthread_mutex_lock(&mutex);
68
69     while (task_queue[last_added_task_index].function == NULL && task_queue[last_added_task_index].data == NULL && last_added_task_index > 0)
70     {
71         --last_added_task_index;
72     }
73
74     task dequeuing_task = task_queue[last_added_task_index];
75
76     task_queue[last_added_task_index].function = NULL;
77     task_queue[last_added_task_index].data = NULL;
78
79     if (last_added_task_index > 0)
80     {
81         --last_added_task_index;
82     }
83     else
84     {
85         work_is_done = true;
86     }
87
88     Pthread_mutex_unlock(&mutex);
89
90     //
91
92     return dequeuing_task;
93 }
94

```

Рисунок 6. Фрагмент кода №6

Функция `pool_shutdown` (Рисунок 7) позволяет завершить работу пула потоков, отменив все ожидающие из них вызовом `sem_post`. Правильно будет дать работающим потокам закончить начатое, поэтому вызов библиотечной функции `pthread_join` для каждого из потоков во втором цикле `for` позволяет нам это сделать.

Объекты мьютекса и семафора уничтожаются соответствующими функциями в завершении работы программы.

```

C threadpool.c x
C threadpool.c > ...
153 void pool_shutdown(void)
154 {
155     for (int thread = 0; thread < NUMBER_OF_THREADS; ++thread)
156     {
157         // Отменяем все ожидающие потоки
158         sem_post(&semaphore);
159     }
160
161     for (int thread = 0; thread < NUMBER_OF_THREADS; ++thread)
162     {
163         // Ждём завершения работы каждого потока thread
164         pthread_join(threads[thread], NULL);
165     }
166
167     pthread_mutex_destroy(&mutex);
168     sem_destroy(&semaphore);
169 }
170

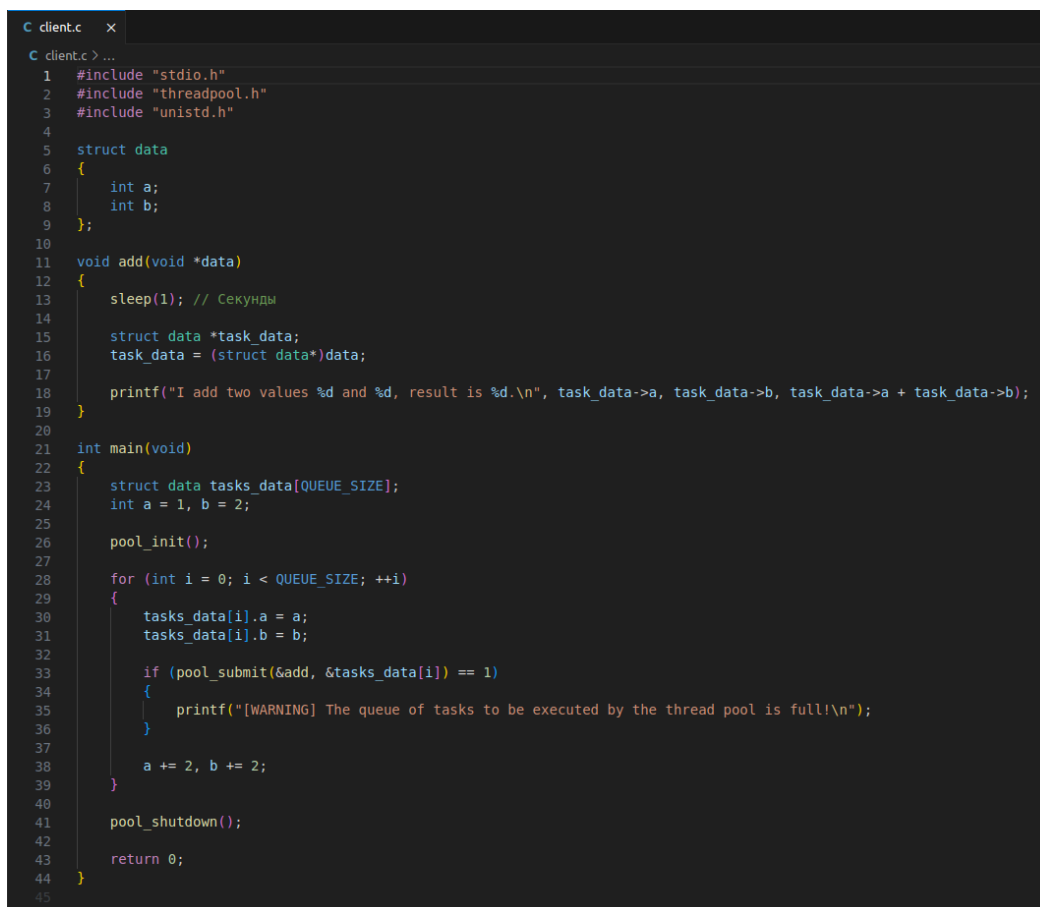
```

Рисунок 7. Фрагмент кода №7

Файл `client.c` (Рисунок 8) неоднократно упоминавшийся выше представляет из себя своеобразный клиент для взаимодействия с пулом потоков – вызовом всех тех функций, которые были подробно описаны ранее.

Именно в этом файле происходит формирование задач в массиве `tasks_data` с одинаковой для каждой из них вызываемой функцией `add` для сложения двух целых чисел `a` и `b`, которые являются данными (аргумент функции `data`), разными для каждой из задач.

Вызов `sleep` на 1 секунду позволяет получить более недетерминированный вывод в консоль, так как без этой функции он получается более однообразным и трудноразличимым из-за лёгковесности взятой операции (сложение чисел) и, как следствие, очень быстрого выполнения программы.



```
C client.c x
C client.c > ...
1  #include "stdio.h"
2  #include "threadpool.h"
3  #include "unistd.h"
4
5  struct data
6  {
7      int a;
8      int b;
9  };
10
11 void add(void *data)
12 {
13     sleep(1); // Секунды
14
15     struct data *task_data;
16     task_data = (struct data*)data;
17
18     printf("I add two values %d and %d, result is %d.\n", task_data->a, task_data->b, task_data->a + task_data->b);
19 }
20
21 int main(void)
22 {
23     struct data tasks_data[QUEUE_SIZE];
24     int a = 1, b = 2;
25
26     pool_init();
27
28     for (int i = 0; i < QUEUE_SIZE; ++i)
29     {
30         tasks_data[i].a = a;
31         tasks_data[i].b = b;
32
33         if (pool_submit(&add, &tasks_data[i]) == 1)
34         {
35             printf("[WARNING] The queue of tasks to be executed by the thread pool is full!\n");
36         }
37
38         a += 2, b += 2;
39     }
40
41     pool_shutdown();
42
43     return 0;
44 }
45
```

Рисунок 8. Фрагмент кода №8