Министерство науки и высшего образования Российской Федерации

Федеральное государственное бюджетное образовательное учреждение

высшего образования

«Ульяновский государственный технический университет»

Кафедра «Вычислительная техника»

Дисциплина «Операционные системы»

**Лабораторная работа 4**

**Проектирование пула потоков**

Выполнил

Студент группы ИВТАСбд42

Зяббаров А.А.

Проверил:

преподаватель кафедры «ВТ»

Беляев К.С.

Ульяновск

2024

# Постановка задачи

Реализация пула потоков с использованием Pthreads и Posix синхронизации в языке программирования C. Необходимо создать функции для инициализации пула потоков, добавления задач в очередь, извлечения задач из очереди и выполнения этих задач потоками. Клиентский код должен иметь возможность инициировать завершение работы пула потоков после завершения всех задач. Решение включает в себя использование мьютексов и семафоров для обеспечения безопасности доступа и синхронизации работы с очередью задач.

# Реализация

Pthread\_mutex\_lock, Pthread\_mutex\_unlock и Pthread\_mutex\_init - это функции для работы с мьютексом, который используется для обеспечения безопасного доступа к общим данным. Pthread\_mutex\_lock блокирует мьютекс, предотвращая другие потоки от доступа к общим ресурсам. Pthread\_mutex\_unlock разблокирует мьютекс, разрешая другим потокам получить доступ. Pthread\_mutex\_init инициализирует мьютекс перед использованием.

Функция enqueue(task t) отвечает за добавление задачи в очередь пула потоков. При вызове она блокирует мьютекс для предотвращения возможных состояний гонки при изменении очереди. Если очередь не заполнена, задача добавляется в конец очереди, и семафор уведомляет о наличии новой задачи. После этого мьютекс разблокируется для других потоков.

Функция dequeue() отвечает за извлечение задачи из очереди. Семафор используется для блокировки, ожидая уведомления о наличии задачи в очереди. После получения уведомления, мьютекс блокируется для безопасного доступа к очереди. Задача извлекается, счетчик задач уменьшается, и мьютекс разблокируется. Извлеченная задача возвращается для выполнения.

Функция execute(void (\*somefunction)(void \*p), void \*p) просто запускает функцию somefunction с параметром p.

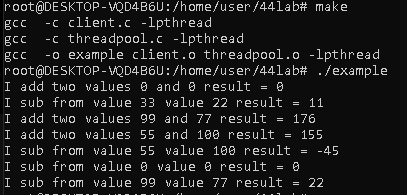
Функция worker(void \*param) представляет собой функцию, выполняемую каждым потоком пула. В бесконечном цикле поток ожидает доступной задачи. Как только задача доступна, она извлекается из очереди и выполняется с использованием execute.

Функция pool\_submit(void (\*somefunction)(void \*p), void \*p) добавляет новую задачу в пул потоков, создавая структуру task и передавая ее в enqueue для добавления в очередь.

Функция pool\_init(void) инициализирует мьютекс, семафор и создает указанное количество потоков для пула. Каждый поток начинает выполнение функции worker.

Функция pool\_shutdown(void) отправляет сигнал каждому потоку завершить выполнение, добавляя пустую задачу (NULL, NULL) в очередь для каждого потока. Затем она ожидает завершения каждого потока с использованием pthread\_join и освобождает ресурсы, такие как мьютекс и семафоры.

# Тестирование



# Вывод

Был реализован пул потоков с использованием Pthreads и Posix синхронизации на языке программирования C. Разработана функциональность для инициализации пула, добавления и извлечения задач в/из очереди, выполнения задач потоками, а также завершения работы пула. В работе использовались мьютексы и семафоры для обеспечения безопасности и синхронизации операций с очередью задач. Реализованное решение предоставляет эффективный механизм для параллельного выполнения задач в рамках пула потоков.

# Приложение. client.c

#include <stdio.h>

#include <unistd.h>

#include "threadpool.h"

struct data

{

int a;

int b;

};

void add(void \*param)

{

struct data \*temp;

temp = (struct data \*)param;

printf("I add two values %d and %d result = %d\n", temp->a, temp->b, temp->a + temp->b);

}

void sub(void \*param)

{

struct data \*temp;

temp = (struct data \*)param;

printf("I sub from value %d value %d result = %d\n", temp->a, temp->b, temp->a - temp->b);

}

void mul(void \*param)

{

struct data \*temp;

temp = (struct data \*)param;

printf("I mul two values %d and %d result = %d\n", temp->a, temp->b, temp->a \* temp->b);

}

void div(void \*param)

{

struct data \*temp;

temp = (struct data \*)param;

printf("I div value %d by value %d result = %d\n", temp->a, temp->b, temp->a / temp->b);

}

void inc(void \*param)

{

struct data \*temp;

temp = (struct data \*)param;

for (int i = 0; temp->a < 1000000; i++)

{

printf("Inc value %d inc value %d\n", temp->a++, temp->b++);

}

}

int main(void)

{

// create some work to do

struct data work, work1, work2, work3;

work.a = 0;

work.b = 0;

work1.a = 55;

work1.b = 100;

work2.a = 33;

work2.b = 22;

work3.a = 99;

work3.b = 77;

// initialize the thread pool

pool\_init();

// submit the work to the queue

pool\_submit(&add, &work);

pool\_submit(&sub, &work);

// pool\_submit(&mul, &work);

// pool\_submit(&div, &work);

pool\_submit(&add, &work1);

pool\_submit(&sub, &work1);

// pool\_submit(&mul, &work1);

// pool\_submit(&div, &work1);

// pool\_submit(&add, &work2);

pool\_submit(&sub, &work2);

// pool\_submit(&mul, &work2);

// pool\_submit(&div, &work2);

pool\_submit(&add, &work3);

pool\_submit(&sub, &work3);

// pool\_submit(&mul, &work3);

// pool\_submit(&div, &work3);

// // may be helpful

// sleep(3);

// shutdown the thread pool

pool\_shutdown();

return 0;

}

# Приложение. threadpool.c

#include "threadpool.h"

#include <pthread.h>

#include <stdlib.h>

#include <stdio.h>

#include <semaphore.h>

#include <assert.h>

#define QUEUE\_SIZE 20

#define NUMBER\_OF\_THREADS 8

typedef struct {

void (\*function)(void \*p);

void \*data;

} task;

task work\_queue[QUEUE\_SIZE];

int queue\_front = 0;

int queue\_rear = -1;

int queue\_count = 0;

pthread\_t worker\_threads[NUMBER\_OF\_THREADS];

pthread\_mutex\_t queue\_mutex;

sem\_t task\_semaphore;

void Pthread\_mutex\_lock(pthread\_mutex\_t \*mutex) {

int rc = pthread\_mutex\_lock(mutex);

assert(rc == 0);

}

void Pthread\_mutex\_unlock(pthread\_mutex\_t \*mutex) {

int rc = pthread\_mutex\_unlock(mutex);

assert(rc == 0);

}

void Pthread\_mutex\_init(pthread\_mutex\_t \*mutex) {

int rc = pthread\_mutex\_init(mutex, NULL);

assert(rc == 0);

}

void enqueue(task t) {

Pthread\_mutex\_lock(&queue\_mutex);

if (queue\_count < QUEUE\_SIZE) {

queue\_rear = (++queue\_rear) % QUEUE\_SIZE;

work\_queue[queue\_rear] = t;

queue\_count++;

sem\_post(&task\_semaphore);

}

Pthread\_mutex\_unlock(&queue\_mutex);

}

task dequeue() {

task worktodo;

sem\_wait(&task\_semaphore);

Pthread\_mutex\_lock(&queue\_mutex);

worktodo = work\_queue[queue\_front];

queue\_front = (++queue\_front) % QUEUE\_SIZE;

queue\_count--;

Pthread\_mutex\_unlock(&queue\_mutex);

return worktodo;

}

void execute(void (\*somefunction)(void \*p), void \*p) {

(\*somefunction)(p);

}

void \*worker(void \*param) {

while (1) {

task worktodo = dequeue();

if (worktodo.function == NULL)

pthread\_exit(NULL);

execute(worktodo.function, worktodo.data);

}

}

int pool\_submit(void (\*somefunction)(void \*p), void \*p) {

task new\_task;

new\_task.function = somefunction;

new\_task.data = p;

enqueue(new\_task);

return 0;

}

void pool\_init(void) {

Pthread\_mutex\_init(&queue\_mutex);

sem\_init(&task\_semaphore, 0, 0);

for (int i = 0; i < NUMBER\_OF\_THREADS; i++) {

pthread\_create(&worker\_threads[i], NULL, worker, NULL);

}

}

void pool\_shutdown(void) {

for (int i = 0; i < NUMBER\_OF\_THREADS; i++) {

pool\_submit(NULL, NULL);

}

for (int i = 0; i < NUMBER\_OF\_THREADS; i++) {

pthread\_join(worker\_threads[i], NULL);

}

pthread\_mutex\_destroy(&queue\_mutex);

sem\_destroy(&task\_semaphore);

}