Лабораторная работа №2

Цель работы – изучение и применение механизмов синхронизации потоков с использованием семафоров и мьютексов для решения реальных задач многозадачных систем.

Задачи работы

- Реализовать многозадачные приложения, где потоки конкурируют за ограниченные ресурсы
- Разработать и внедрить семафоры для синхронизации потоков в различных сценариях многозадачности.
- Реализовать системы очередей с приоритетами, для днамического перераспределения ресурсов
- Осуществить адаптивную работу системы, где изменение внешних условий (например, перегрузка или выход из строя компонентов) приводит к автоматической перераспределению задач
- Научиться моделировать сложные логистические и производственные процессы с учетом приоритетов и аварийных ситуаций

Теоретическая часть

Механизмы синхронизации, такие как семафоры и мьютексы, являются ключевыми элементами для управления многозадачностью в многопоточных приложениях. Они позволяют координировать доступ нескольких потоков к общим ресурсам, предотвращая состояние гонки и обеспечивая корректность работы программы.

Семафоры — это объекты синхронизации, которые регулируют доступ потоков к ресурсу, поддерживая счетчик. Семафор с инициализацией 1 (бинарный семафор) позволяет только одному потоку получить доступ к ресурсу, а семафор с произвольным значением N позволяет одновременно работать N потокам.

Мьютексы (mutex — mutual exclusion) — это объекты синхронизации, которые позволяют гарантировать эксклюзивный доступ к ресурсу. Если один поток захватывает мьютекс, то другие потоки, пытающиеся захватить тот же мьютекс, будут заблокированы до тех пор, пока мьютекс не будет освобожден.

Семафоры бывают двух типов:

- **Бинарный семафор:** Это особый случай семафора, который может принимать только два значения: 0 и 1. Он используется для того, чтобы разрешить доступ только одному потоку в любой момент времени. Такой семафор часто используется для реализации взаимного исключения.
- **Счетный семафор:** В отличие от бинарного, подсчитывающий семафор может принимать значения больше единицы. Он используется, когда нужно разрешить доступ нескольким потокам, например, ограниченному количеству потоков, которые могут одновременно использовать ресурс. В случае с подсчитывающим семафором потоки могут "захватывать" и "освобождать" семафор, изменяя его значение, что позволяет эффективно распределять ресурсы между потоками.

В C++ стандартная библиотека предоставляет семафоры с помощью класса std::counting_semaphore, который можно инициализировать с максимальным значением для контроля доступа.

Пример использования семафора

```
#include <iostream>

#include <semaphore>

#include <thread>


std::counting_semaphore<3> sem(3); // Позволяет одновременно работать 3 потокам

void task(int id) {

   sem.acquire();  // Поток захватывает семафор

   std::cout << "Task " << id << " is working\n";

   std::this_thread::sleep_for(std::chrono::seconds(1));  // Эмуляция работы

   sem.release();  // Поток освобождает семафор

}

int main() {

   std::thread t1(task, 1);

   std::thread t2(task, 2);

   std::thread t3(task, 3);
```

```
    std::thread t4(task, 4);  // Этот поток будет ожидать, пока один из предыдущих завершит свою работу
    t1.join();
    t2.join();
    t3.join();
    t4.join();
    return 0;
}
```

Взаимодействие семафоров с мьютексами

Часто семафоры используются вместе с мьютексами для защиты данных:

Мьютексы предотвращают одновременную модификацию ресурса.

Семафоры ограничивают количество потоков, работающих с ресурсом одновременно.

Пример использования семафора с мьютексом

```
#include <iostream>
#include <thread>
#include <mutex>
#include <semaphore>

std::counting_semaphore<3> sem(3);
std::mutex mtx;
void task(int id) {
    sem.acquire();
    std::lock_guard<std::mutex> lock(mtx);  // Гарантирует защиту критической секции
    std::cout << "Thread " << id << " accessing shared resource\n";
    std::this_thread::sleep_for(std::chrono::seconds(1));
    sem.release();
}
```

```
int main() {
    std::thread threads[5];
    for (int i = 0; i < 5; ++i)
        threads[i] = std::thread(task, i + 1);
    for (auto &t : threads)
        t.join();
    return 0;
}
```

Семафор ограничивает количество одновременно работающих потоков, а мьютекс гарантирует безопасный доступ к std::cout.

Типичные сценарии использования семафоров

Семафоры находят применение в широком спектре многопоточных задач.

Ограничение доступа к ресурсу

Семафоры используются для контроля количества потоков, имеющих доступ к ресурсу. Это особенно полезно в следующих случаях:

- Ограничение числа одновременно выполняемых задач (например, не более 5 пользователей могут одновременно загружать файл)
- Управление пулами ресурсов (например, работа с ограниченным числом подключений к базе данных)
- Контроль за использованием памяти или процессорного времени

Пример: разрешение доступа только двум потокам одновременно

```
std::counting_semaphore<2> resourceSemaphore(2);
```

Организация очереди потоков

Семафоры помогают управлять потоками так, чтобы они выполнялись в определенном порядке. Например, в задаче "Производитель - Потребитель" семафоры предотвращают ситуацию, когда потребитель пытается извлечь данные из пустого буфера.

Возможный механизм

- Производитель увеличивает семафор при добавлении элемента в буфер

- Потребитель уменьшает семафор перед извлечением элемента

Пример

```
#include <iostream>
#include <thread>
#include <queue>
#include <semaphore>
#include <mutex>

std::queue<int> buffer;
std::counting_semaphore<1> full(0);  // Количество заполненных ячеек
std::counting_semaphore<1> empty(1); // Количество пустых ячеек
std::mutex mtx;
void producer() {
   for (int i = 0; i < 5; ++i) {
      empty.acquire();
      std::lock_guard<std::mutex> lock(mtx);
      buffer.push(i);
      std::cout << "Produced: " << i << std::endl;
      full.release();
   }
}
void consumer() {
   for (int i = 0; i < 5; ++i) {
      full.acquire();
      std::lock_guard<std::mutex> lock(mtx);
      int item = buffer.front();
      buffer.pop();
      std::cout << "Consumed: " << item << std::endl;
```

```
        empty.release();
    }
}
int main() {
    std::thread prod(producer);
    std::thread cons(consumer);
    prod.join();
    cons.join();
    return 0;
}
```

Здесь семафоры full и empty обеспечивают синхронизацию потоков "Производитель - Потребитель".

Взаимодействие потоков через семафоры

Семафоры позволяют упорядочивать выполнение потоков. Например, можно сделать так, чтобы один поток ждал завершения работы другого перед началом своей работы.

Пример: поток B должен ждать, пока поток A выполнит свою задачу

```
#include <iostream>
#include <thread>
#include <semaphore>

std::counting_semaphore<1> sem(0);
void taskA() {
    std::cout << "Task A is running\n";
    std::this_thread::sleep_for(std::chrono::seconds(1));
    sem.release(); // Разрешаем потоку B продолжить выполнение
}
void taskB() {
```

```
    sem.acquire(); // Ждём, пока поток A завершится
    std::cout << "Task B is running after A\n";
}
int main() {
    std::thread t1(taskA);
    std::thread t2(taskB);
    t1.join();
    t2.join();
    return 0;
}
```

Прием называется управление порядком выполнения потоков.

Взаимные блокировки (Deadlocks) и их предотвращение

Взаимная блокировка (deadlock) – это ситуация, при которой два или более потока ожидают освобождения ресурсов, которые уже заблокированы друг другом.

Пример взаимной блокировки

```
#include <iostream>
#include <thread>
#include <mutex>

std::mutex m1, m2;
void task1() {
    std::lock_guard<std::mutex> lock1(m1);
    std::this_thread::sleep_for(std::chrono::milliseconds(100));
    std::lock_guard<std::mutex> lock2(m2);
    std::cout << "Task 1 completed\n";
}
void task2() {
```

```
    std::lock_guard<std::mutex> lock2(m2);
    std::this_thread::sleep_for(std::chrono::milliseconds(100));
    std::lock_guard<std::mutex> lock1(m1);
    std::cout << "Task 2 completed\n";
}
int main() {
    std::thread t1(task1);
    std::thread t2(task2);
    t1.join();
    t2.join();
    return 0;
}
```

Возникает deadlock, так как:

- task1() захватывает m1, затем пытается получить m2
- task2() захватывает m2, затем пытается получить m1

Методы предотвращения deadlock

1. Использование std::scoped_lock

```
std::scoped_lock lock(m1, m2);
```

Этот метод предотвращает deadlock, так как захватывает несколько мьютексов атомарно.

2. Фиксированный порядок захвата ресурсов

Гарантировать, что все потоки блокируют мьютексы в одном и том же порядке.

3. Использование std::try_lock()

Этот метод позволяет избежать блокировки, если один из мьютексов занят.

4. Приоритетное освобождение ресурсов

Если поток не может получить доступ ко всем ресурсам, он должен освободить уже захваченные и попытаться снова.

Очереди с приоритетами

Очередь с приоритетами — это структура данных, которая используется для хранения элементов с приоритетом, где каждый элемент имеет свой приоритет, и элементы с более высоким приоритетом извлекаются первыми. Очереди с приоритетами широко используются в системах, где нужно обработать задачи в зависимости от их важности.

В языке C++ очередь с приоритетами реализована с помощью контейнера std::priority_queue. Этот контейнер поддерживает порядок элементов на основе их приоритета, где по умолчанию элементы с большим значением имеют более высокий приоритет.

Чтобы создать очередь с приоритетами, необходимо определить структуру данных, которая будет храниться в очереди. Затем через перегрузку оператора < можно установить правило сортировки элементов. Например, для того чтобы очередь сортировалась по убыванию приоритета, нужно определить структуру задачи с приоритетом.

Пример использования очереди с приоритетами

```
#include <iostream>
#include <queue>
#include <vector>
struct Task {
    int id;
    int priority;
    bool operator<(const Task& other) const {
        return priority < other.priority;  // Задачи с более высоким приоритетом идут первыми
    }
};
int main() {
    std::priority_queue<Task> pq;
    pq.push({1, 2});  // Задача с приоритетом 2
    pq.push({2, 1});  // Задача с приоритетом 1
    pq.push({3, 3});  // Задача с приоритетом 3
```

```
    while (!pq.empty()) {
        Task t = pq.top();
        pq.pop();
        std::cout << "Task " << t.id << " with priority " << t.priority << std::endl;
    }
    return 0;
}
```

Вариант 1: Производственная логистика и интеллектуальная обработка данных

Задача 1: Интеллектуальная система распределения грузовиков

- В порту 5 погрузочных кранов и 10 грузовиков
- Каждый грузовик ожидает, пока будет доступен кран, и загружается за 3-6 секунд
- Если в очереди ожидания более 5 грузовиков, запускается резервный кран
- Если в порту остается менее 3 загруженных грузовиков, запускается аварийная загрузка (ускоренный режим)
- Грузовики конкурируют за краны, используя семафоры и очереди

(std::uniform_int_distribution<> dis(3, 6); пример загрузки (от 3 до 6 секунд)

uniform_int_distribution<> — это шаблон класса из библиотеки C++, который используется для генерации случайных целых чисел с равномерным распределением. Это означает, что каждое число из заданного диапазона имеет одинаковую вероятность быть выбранным.

(3, 6) — это диапазон значений, которые могут быть сгенерированы. То есть генератор случайных чисел будет выбирать случайные числа от 3 до 6 включительно. В данном случае, возможные значения: 3, 4, 5 и 6.)

Задача 2: Адаптивная многопоточная обработка видеопотоков

- 6 потоков получают видеоданные с камер наблюдения
- Потоки используют 3 ускорителя для обработки данных
- Если один ускоритель выходит из строя, его задачи перераспределяются
- Важные события (движение, подозрительная активность) получают приоритет перед обычными кадрами
- Глобальный мониторинг контролирует нагрузку и перераспределяет задачи

(к примеру - четные кадры – важные)

Вариант 2: Гибридные системы контроля доступа и интеллектуальное производство

Задача 1: Система интеллектуального контроля доступа

- Офисное здание с 5 входами и сенсорами доступа
- Каждое утро потоки сотрудников конкурируют за вход
- Система адаптируется: если поток людей возрастает, открываются дополнительные турникеты
- Высокоприоритетные сотрудники (директора, инженеры) получают приоритет
- Если турникет зависает или перестает работать, аварийная система автоматически перенаправляет людей

(threads.push_back(std::thread(employee, i, i % 5 == 0)); пример - Высокоприоритетные сотрудники с id 5, 10, 15, 20)

Задача 2: Гибкая производственная линия

- 4 станка выполняют заказы с разными уровнями приоритета
- Если один станок выходит из строя, заказы автоматически перераспределяются
- Потоки срочных заказов могут прерывать обычные процессы
- Используется очередь с динамическими приоритетами для оптимальной загрузки
- Производственная линия автоматически адаптируется к нагрузке

(приоритет заказа, чем меньше число - тем выше приоритет)

Вариант 3: Кластерные вычисления и динамическая координация транспортных потоков

Задача 1: Кластерная система обработки задач

- Кластер состоит из 5 серверов, обрабатывающих задачи
- Каждая задача размещается в очереди, распределяется по серверам
- Если один сервер перегружен, его задачи переносятся на другие
- Важные задачи получают доступ к ресурсам первыми
- Если нагрузка на кластер >80%, запускается дополнительный сервер

(приоритет задачи, чем меньше число - тем выше приоритет)

(если нагрузка > 80%, добавляем новый сервер запускаем новый поток)

Задача 2: Оптимизация дорожного трафика

- В городе есть 10 перекрестков с умными светофорами
- Если поток машин возрастает >70% на одном перекрестке, светофор адаптирует время сигналов
- Экстренные службы (пожарные, скорая помощь) получают приоритет и могут переключать сигналы
- Если на перекрестке застряли машины, активируется аварийная система управления
- Используются очереди, мьютексы и семафоры для управления движением

(экстренная служба – реализация через флаг)

Вариант 4: Квантовые вычисления и адаптивные системы мониторинга

Задача 1: Квантовый симулятор распределенных вычислений

- Кластер квантового симулятора состоит из 4 квантовых процессоров и 10 потоков вычислений
- Каждая задача требует эксклюзивного доступа к процессору
- Если процессор перегружен, задачи разделяются на более мелкие
- Приоритетные задачи (критически важные расчёты) имеют преимущество
- Если один процессор выходит из строя, его задачи переносятся на соседние
- Используются семафоры для контроля количества активных задач

(приоритет задачи, чем меньше число - тем выше приоритет)

(критически важная задача определяется через флаг)

Задача 2: Адаптивная система мониторинга энергосети

- В городе 10 станций мониторинга следят за энергопотреблением
- Каждая станция отправляет пакет данных в центральный сервер
- Если поток данных превышает 80% загрузки, сервер включает дополнительные обработчики
- В случае аварии низкоприоритетные данные отбрасываются
- Система должна автоматически адаптироваться к резким скачкам нагрузки
- Потоки конкурируют за ресурсы сервера, используя семафоры и мьютексы

(приоритет данных, чем меньше число - тем выше приоритет)

(критически важная информация определяется через флаг)

Пример

Задача: Управление доступом к серверам с использованием семафоров и очереди с приоритетами

Условия задачи

- Есть серверный кластер из 3 серверов, каждый из которых обрабатывает задачи
- Задачи поступают в очередь. Задачи могут быть разных типов:

1. Важные задачи, которые должны быть выполнены в первую очередь.
2. Обычные задачи, которые могут быть выполнены позже, если ресурсы серверов будут свободны.

- Каждая задача требует эксклюзивного доступа к серверу
- Время обработки задач варьируется от 1 до 5 секунд
- Если на сервере недостаточно мощности, задача должна быть перераспределена на другие серверы, если они свободны
- При недостатке серверных мощностей для выполнения задачи система должна создавать резервные серверы

Для решения задачи будет использована очередь с приоритетами для сортировки задач и семафоры для управления доступом к серверным ресурсам.

Семафоры

- Семафоры будут использоваться для синхронизации доступа к серверам и предотвращения одновременной работы на одном сервере несколькими задачами.
- Для каждого сервера будет выделен семафор, который разрешает доступ только одному потоку (задаче) на сервер.

Очередь с приоритетами

- Для управления задачами с разным приоритетом будет использоваться очередь с приоритетами, где задачи с более высоким приоритетом (важные) будут обрабатываться первыми.

Реализация

```
#include <iostream>

#include <thread>
```

```
#include <queue>
#include <random>
#include <semaphore>
#include <chrono>
#include <vector>
#include <mutex>

// Структура для задачи
struct Task {
    int id;
    int priority;  // 1 - важная, 2 - обычная
    int duration;  // Время на выполнение задачи в секундах
};
// Оператор сравнения для очереди с приоритетами
struct ComparePriority {
    bool operator()(const Task& t1, const Task& t2) {
        // Задачи с меньшим числом (более важные) идут раньше
        return t1.priority > t2.priority;
    }
};
std::priority_queue<Task, std::vector<Task>, ComparePriority> task_queue;  // Очередь задач
std::counting_semaphore<3> servers(3);  // 3 сервера
std::mutex output_mutex;  // Мьютекс для синхронного вывода в консоль
// Функция обработки задачи на сервере
void process_task(Task task) {
    servers.acquire();  // Ожидание освобождения сервера
    std::this_thread::sleep_for(std::chrono::seconds(task.duration));  // Симуляция времени обработки задачи
```

```
    // Выводим информацию о выполнении задачи
    {
        std::lock_guard<std::mutex> guard(output_mutex);
        std::cout << "Task " << task.id << " (priority " << task.priority << ") completed after "
                  << task.duration << " seconds.\n";
    }
    servers.release();  // Освобождение сервера
}
// Функция для добавления задач в очередь
void add_task(int id, int priority) {
    std::random_device rd;
    std::mt19937 gen(rd());
    std::uniform_int_distribution<> dis(1, 5);  // Генерация случайного времени на выполнение (от 1 до 5 секунд)
    Task task = {id, priority, dis(gen)};
    task_queue.push(task);
    std::lock_guard<std::mutex> guard(output_mutex);
    std::cout << "Task " << id << " with priority " << priority << " added to the queue.\n";
}
// Функция обработки задач из очереди
void process_tasks() {
    while (!task_queue.empty()) {
        Task task = task_queue.top();
        task_queue.pop();
        process_task(task);
    }
}
int main() {
    // Добавляем задачи в очередь
```

```
    add_task(1, 1);  // Важная задача
    add_task(2, 2);  // Обычная задача
    add_task(3, 1);  // Важная задача
    add_task(4, 2);  // Обычная задача
    add_task(5, 2);  // Обычная задача

    // Создаем потоки для обработки задач
    std::vector<std::thread> threads;
    for (int i = 0; i < 3; ++i) {
        threads.push_back(std::thread(process_tasks));
    }
    // Ожидаем завершения всех потоков
    for (auto& t : threads) {
        t.join();
    }
    return 0;
}
```

1. Структура Task

Содержит информацию о задаче: идентификатор, приоритет (1 — важная задача, 2 — обычная) и продолжительность выполнения задачи.

2. Очередь с приоритетами

Используется стандартная очередь с приоритетами для обработки задач в порядке их важности. Важно, что задачи с более высоким приоритетом (приоритет 1) обрабатываются первыми.

3. Семафор servers

Инициализируем семафор с значением 3, что означает, что одновременно может работать только 3 задачи (по числу серверов).

4. Мьютекс output_mutex

Используется для синхронизации вывода в консоль, чтобы избежать одновременного вывода нескольких потоков.

5. Функции

add_task: Добавляет задачу в очередь с случайным временем обработки.

process_task: Обрабатывает задачу, захватывая сервер и освобождая его после выполнения.

process_tasks: Обрабатывает задачи из очереди по очереди в порядке их приоритета.

6. Основной поток

Добавляет задачи в очередь с различными приоритетами и запускает 3 потока для обработки этих задач.

Пример вывода программы, где задачи обрабатываются в зависимости от их приоритета

Task 1 with priority 1 added to the queue.

Task 2 with priority 2 added to the queue.

Task 3 with priority 1 added to the queue.

Task 4 with priority 2 added to the queue.

Task 5 with priority 2 added to the queue.

Task 1 (priority 1) completed after 4 seconds.

Task 3 (priority 1) completed after 3 seconds.

Task 2 (priority 2) completed after 5 seconds.

Task 4 (priority 2) completed after 2 seconds.

Task 5 (priority 2) completed after 3 seconds.

- Задачи с приоритетом 1 (важные) обрабатываются раньше, чем задачи с приоритетом 2 (обычные).
- Задачи выполняются параллельно с учетом ограничения на количество серверов (3 потока).
- Каждая задача блокирует сервер, пока не завершится, и после завершения освобождает сервер для следующей задачи.