Отчет по лабораторной работе №10

«Планировщик процессов»

Выполнил: студент группы РИС-22-1б

Поважный Виталий Евгеньевич

Вариант 11

17.03.2025

**Задание**

Цель работы: научиться организовывать планировщик процессов.

Необходимо разработать программу, запускающую от 10 процессов (потоков выполнения/нитей/корутин) и планировщик этих процессов. В планировщике реализовать три очереди процессов. Планирование процессов реализовать по вариантам. При реализации использовать библиотеку libpth (https://www.gnu.org/software/pth/pthmanual.html) или С++20 coroutines.

Вариант:





**Ход работы**

В данной работе реализована система планирования процессов с использованием корутин C++20. Каждый процесс представлен в виде объекта Process, содержащего идентификатор, приоритет и корутину Task, представляющую выполнение задач процесса. Управление процессами осуществляется с помощью класса Scheduler, который поддерживает три стратегии планирования:

**Архитектура решения**:

1. **Корутины (Task)**:  
   Каждый процесс представлен корутиной, которая приостанавливается после выполнения итерации (co\_await std::suspend\_always{}). Это позволяет переключаться между процессами без создания потоков ОС.
2. **Процессы (Process)**:  
   Содержат:
   * id — идентификатор процесса.
   * dynamic\_priority — динамический приоритет (уменьшается на 10 после каждого кванта).
   * task — корутина, связанная с процессом.
3. **Очереди планировщика (Scheduler)**:
   * **Очередь 1 (FIFO)**: Для процессов с высоким приоритетом (≥50). Выполняется строго по порядку.
   * **Очередь 2 (Round Robin)**: Для среднего приоритета (30–49). Каждый процесс выполняется по одному кванту, затем перемещается в конец очереди.
   * **Очередь 3 (FIFO)**: Для низкого приоритета (<30). Активна, только когда пусты первые две очереди.
4. **Динамический приоритет**:  
   После каждого выполнения кванта приоритет процесса снижается. При достижении пороговых значений (50, 30) процесс перемещается в соответствующую очередь, что обеспечивает вытеснение более приоритетными задачами.

Пример работы программы ниже:



Рисунок 1 – Пример работы планировщика

Код программы:

#include <iostream>

#include <coroutine>

#include <vector>

#include <deque>

#include <optional>

#include <string>

#include <memory>

#include <algorithm>

#include <cstdlib>

#include <ctime>

struct Task {

// Внутренняя структура, необходимая для работы корутин

struct promise\_type {

Task get\_return\_object() {

return Task(std::coroutine\_handle<promise\_type>::from\_promise(\*this));

}

static std::suspend\_always initial\_suspend() noexcept { return {}; } // Указывает, что корутина приостанавливается сразу после создания.

static std::suspend\_always final\_suspend() noexcept { return {}; } // Указывает, что корутина приостанавливается после завершения.

void unhandled\_exception() { std::terminate(); } // Завершает программу при необработанном исключении.

void return\_void() {} // Корутина ничего не возвращает.

};

std::coroutine\_handle<promise\_type> coro; // Хранит дескриптор корутины

Task(std::coroutine\_handle<promise\_type> h) : coro(h) {}

Task(const Task&) = delete; // Копирование запрещено

Task(Task&& other) noexcept : coro(other.coro) { other.coro = nullptr; } // Перемещение разрешено

~Task() { if (coro) coro.destroy(); }

bool resume() { // Возобновляет выполнение корутины и возвращает true, если она еще не завершена.

if (!coro.done()) {

coro.resume();

return !coro.done();

}

return false;

}

};

struct Process {

int id;

int dynamic\_priority; // Динамический приоритет

Task task;

Process(int \_id, int \_priority, Task&& \_task) // Конструктор инициализирует поля, принимая корутину через перемещение

: id(\_id), dynamic\_priority(\_priority), task(std::move(\_task)) {

}

};

// Это корутина, которая выполняет заданное количество итераций

Task processFunction(int id, int totalIterations) {

for (int i = 1; i <= totalIterations; i++) {

std::cout << "[Процесс " << id << "] Итерация " << i << std::endl;

co\_await std::suspend\_always{};

}

}

class Scheduler {

public:

Scheduler() : high\_priority(50), medium\_priority(30) {}

void addProcess(Process&& proc) {

auto p = std::make\_unique<Process>(std::move(proc)); // Принимает процесс и создает умный указатель

if (p->dynamic\_priority >= high\_priority) {

queue1.push\_back(p.get());

}

else if (p->dynamic\_priority >= medium\_priority) {

queue2.push\_back(p.get());

}

else {

queue3.push\_back(p.get());

}

processes.push\_back(std::move(p));

}

// Запускает планировщик, пока есть процессы в любой из очередей.

void run() {

while (!queue1.empty() || !queue2.empty() || !queue3.empty()) {

if (executeQueue1()) continue;

if (executeQueue2()) continue;

executeQueue3();

}

}

private:

std::deque<Process\*> queue1; // FIFO, high priority

std::deque<Process\*> queue2; // RR, medium priority

std::deque<Process\*> queue3; // FIFO, low priority

std::vector<std::unique\_ptr<Process>> processes;

const int high\_priority;

const int medium\_priority;

bool executeQueue1() {

if (queue1.empty()) return false;

auto\* p = queue1.front();

queue1.pop\_front();

bool active = p->task.resume();

updatePriority(p);

if (active) {

requeueProcess(p);

}

return true;

}

// Выполняет процесс один квант, обновляет приоритет, и если процесс активен

bool executeQueue2() {

if (queue2.empty()) return false;

auto\* p = queue2.front();

queue2.pop\_front();

bool active = p->task.resume();

updatePriority(p);

if (active) {

requeueProcess(p);

}

else {

return true;

}

if (p->dynamic\_priority >= medium\_priority && p->dynamic\_priority < high\_priority) {

queue2.push\_back(p);

}

else {

requeueProcess(p);

}

return true;

}

void executeQueue3() {

if (queue3.empty()) return;

auto\* p = queue3.front();

queue3.pop\_front();

bool active = p->task.resume();

updatePriority(p);

if (active) {

requeueProcess(p);

}

}

// Уменьшает приоритет на 10 после каждого кванта, не позволяя ему стать меньше 0.

void updatePriority(Process\* p) {

p->dynamic\_priority -= 10; // Пример динамического изменения приоритета

if (p->dynamic\_priority < 0) p->dynamic\_priority = 0;

}

// Перераспределяет процесс в очередь в зависимости от нового значения

void requeueProcess(Process\* p) {

if (p->dynamic\_priority >= high\_priority) {

queue1.push\_back(p);

}

else if (p->dynamic\_priority >= medium\_priority) {

queue2.push\_back(p);

}

else {

queue3.push\_back(p);

}

}

};

int main() {

setlocale(LC\_ALL, "");

std::srand(std::time(nullptr));

Scheduler scheduler;

const int NUM\_PROCESSES = 10;

const int ITERATIONS\_PER\_PROCESS = 3;

std::cout << "=== Процессы и их приоритеты ===" << std::endl;

for (int i = 0; i < NUM\_PROCESSES; ++i) {

int priority = std::rand() % 100;

auto task = processFunction(i, ITERATIONS\_PER\_PROCESS);

scheduler.addProcess(Process(i, priority, std::move(task)));

std::cout << "[Процесс " << i << "] Итерация " << priority << std::endl;

}

std::cout << "=== Запуск планировщика ===" << std::endl;

scheduler.run();

return 0;

}