ГУАП

КАФЕДРА № 44

ОТЧЕТ
ЗАЩИЩЕН С ОЦЕНКОЙ

ПРЕПОДАВАТЕЛЬ

канд. техн. наук, доцент		Н.В. Кучин
должность, уч. степень, звание	подпись, дата	инициалы, фамилия

ОТЧЕТ О ЛАБОРАТОРНОЙ РАБОТЕ №1

МОДЕЛИРОВАНИЕ РАБОТЫ ДИСПЕТЧЕРА ЗАДАЧ

по курсу: ОПЕРАЦИОННЫЕ СИСТЕМЫ

РАБОТУ ВЫПОЛНИЛ

СТУДЕНТ ГР. № 4143		Е. Д. Тегай
	подпись, дата	инициалы, фамилия

Санкт-Петербург 2023

Цель работы

Написать и отладить программу, моделирующую работу диспетчера операционной системы в соответствии с заданной дисциплиной диспетчеризации.

Индивидуальное задание

Содержание индивидуального варианта № 13

Диспетчер на основе дисциплины LCFS (last come – first served – последним пришёл - первым обслужился). Данная дисциплина является вытесняющей и характерна для обработки прерываний.

Задаваемые исходные данные для тестирования и отладки по каждому варианту:

- число параллельно выполняемых задач, данное значение должно задаваться произвольно, т. е. не быть константой;

- данные по каждой задаче:

- имя задачи;

- момент активизации задачи (процесса), т. е. момент перехода задачи из состояния бездействия в состояние готовности.

- время выполнения её на процессоре.

Требования к программе

Разрабатываемая программа должна функционировать в мультизадачной среде, т.е. программа должна выполняться под управлением одной из версий ОС Windows, Unix и т.п. Результаты моделирования (временная диаграмма занятости процессора и диаграмма изменения во времени очереди готовых к выполнению задач) должны быть представлены в графическом виде или в виде таблицы состояний задач на каждом шаге(такте) моделирования.

Следует более подробно рассмотреть суть очереди LCFS. LCFS (Last-Come-First_Served) – стратегия планирования в операционных системах, где процесс, который приходит последним, обслуживается первым, даже если

другие процессы находятся в очереди дольше. Иными словами, задачи выполняются в порядке обратном времени их поступления в систему.

Рассмотрим простейший пример. Допустим, есть три задачи: A, B и C. Порядок их поступления таков:

1. Процесс A поступает в систему
2. Процесс B поступает в систему
3. Процесс C поступает в систему

Теперь идёт выполнение поступивших процессов с использованием LCFS. Это будет происходить в такой очередности:

1. Процесс C начинает выполняться первым, так как он пришёл последним
2. После завершения процесса C начинает выполняться процесс B
3. После завершения процесса B начинает выполняться процесс A

Графически это можно изобразить так, как показано на рисунке 1.

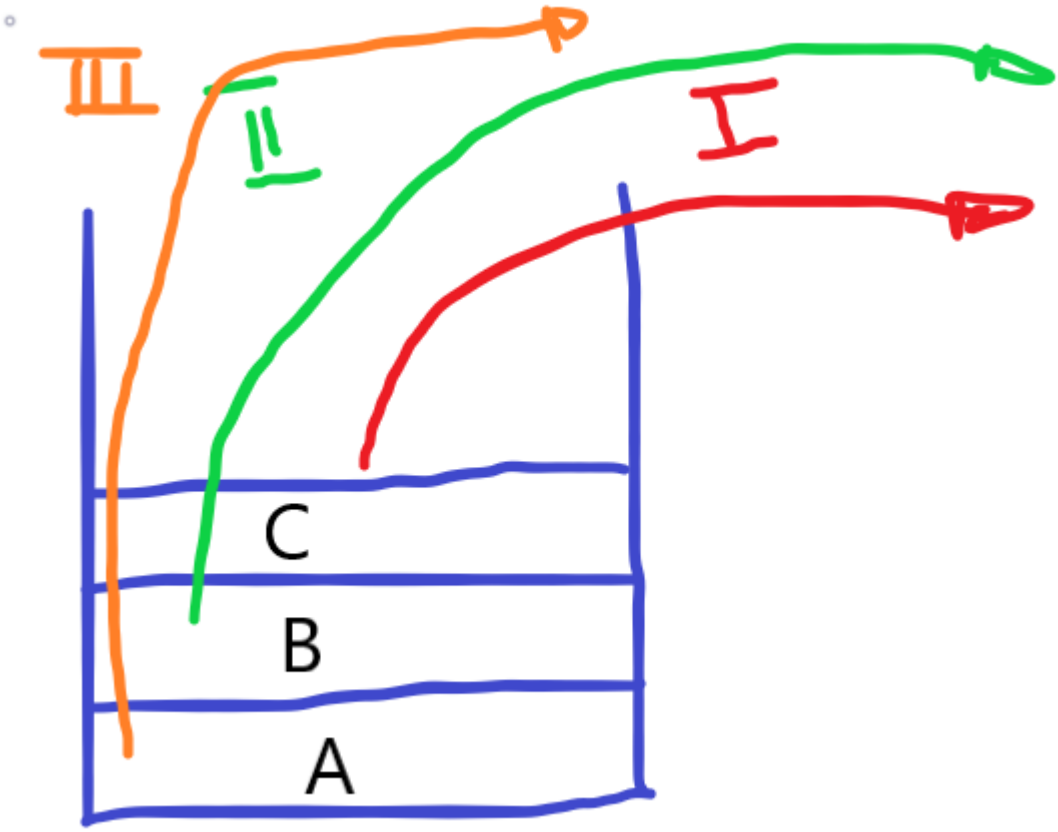

Рисунок 1 – Стек из процессов

Основными преимуществами являются:

1. В отличие от стратегий планирования, таких как Round Robin, где процессы переключаются с интервалами времени, LCFS выполняет процессы до их завершения без прерываний. Это может привести к эффекту "замедления" (convoy effect) — когда короткий процесс может ждать завершения более длинного процесса.

2. Простота в реализации. Это делает его хорошим выбором для простых систем, где нет сложных алгоритмов планирования.

3. В отличие от других стратегий, таких как Priority Scheduling, LCFS не обрабатывает приоритеты процессов. Все процессы обрабатываются в порядке их поступления.

В качестве языка программирования был выбран C++.

Листинг программы

```
// ПОДКЛЮЧЕНИЕ ДИРЕКТИВ
/* Директива <iostream>, с помощью которой определяются объекты "cin", "cout", "cerr" и
т.п.
* элементов, необходимых для ввода и вывода данных в программе */
#include <iostream>
/*  Директива <stack>, которая предоставляет реализацию стека в виде шаблонного класса.
Также предоставляет функции для добавления элемента на вершину стеку (push), удаления
элемента с вершины стека (pop) и доступа к элементу на вершине стека (top)*/
#include <stack>
/* Директива <vector> предоставляет реализацию динамического массива в виде шаблонного
класса. Также предоставляет функции для добавления элементов в конец вектора (push_back),
удаления элемента с конца вектора (pop_back), доступа к элементам по индексу и др.*/
#include <vector>
/* Директива <algorithm> предоставляет реализации различных алгоритмов, таких как
* сортировка, поиск, удаление элементов и др. Эти алгоритмы широко используются для
обработки контейнеров, таких как векторы, списки, массивы и др. */
#include <algorithm>
/* Директива <queue> предоставляет реализацию структуры данных "очередь" (queue) в виде
* класса */
#include <queue>
/* Директива <iomanip> предоставляет манипуляторы форматирование, такие как setw */
#include <iomanip>

/* Директива, которая позволяет использовать все имена из пространства имён std без
явного указания этого пространства имён перед каждым именем */
using namespace std;

// СТРУКТУРЫ
/* Структура, описывающая состояние на каждом такте времени*/
struct Tick {
// Численная переменная, обозначающая номер тика
    int tick_number;
// Строковая переменная, обозначаюшая имя текущей задачи
    string execution_taskName;
/* Создание поля waiting_tasksQueue, которое представляет собой очередь строк.
*  Эта очередь используется для хранения и управления заачами, которые пребывают в
статусе
```

```
*  ожидания на текущем такте времени. Когда задача готова к выполнению, она будет
*  извлекаться из этой очереди */
    queue<string> waiting_tasksQueue;
};

/* Структура, описывающая информацию о задаче */
struct Task {
// Строковая переменная, обозначающая имя задачи
    string task_name;
// Численная переменная, обозначающая время (такт времени) начала работы задачи
    int task_startTime;
// Численная переменная, обозначающая полное время (такт/-ы времени) работы задачи
    int task_duration;
/* Численная переменная, обозначающая время (такт времени) выполнения задачи в контексте
 *  прерывания */
    int task_breakingTime = 0;
};

//ФУНКЦИИ
/* Функция, которая принимает вектор задач (vector<Task>)в качестве аргумента и
* возвращает вектор <Tick>. Является в каком-то смысле главной, потому что именно
* в ней происходят различные действия по принципу LCFS - last come - first served */
vector<Tick> TaskDispatcher(vector<Task> tasks) {
// Численная переменная-счётчик, которая обозначает число завершённых задач
    int finished_tasksCounter = 0;
// Строковая переменная, обозначающая имя текущей задачи
    string current_task;
/* Создание объекта типа Task с именем task_null, который обозначает задачу-пустышку.
*  Его использование уместно тогда, когда на текущем тике нет никаких изменений */
    Task task_null;
// Присвоение задаче-пустушке имени "-"
    task_null.task_name = "-";

/* Объявление переменной result типа  vector<Tick>. Проще говоря, Result представлет
* собой вектор (контейнер), элементами которого являются объекты типа Tick */
    vector<Tick> result;
/* Объявление итератора ptr, который предназначен для работы с контейнером (создан для
 * перебора элементов контейнера)*/
    vector<Task>::iterator ptr;
/* Объявление переменной tasks_stack, которая является контейнером типа vector,
* содержащая объекты типа Task. Грубо говоря, tasks_stack представляет собой стек задач,
* где каждая задача представлена объектом типа Task */
    vector<Task> tasks_stack;

/* Цикл, который будет выполняться до тех пор, пока все задачи не будут завершены.
*  Каждая итерация представляет собой один временной такт */
    for (int tick = 0; finished_tasksCounter < tasks.size(); tick++) {
// Создание объекта, который представляет текущий момент времени
        Tick current;
// Присвоение номера такта, равного текущей итерации цикла
        current.tick_number = tick;
// Цикл, который итерирует по стеку задач
        for (int i = 0; i < (int)tasks_stack.size() - 1; i++) {
// Получение имени задачи из текущего элемента стека
            current_task = tasks_stack[i].task_name;
// Поиск задачи с соответствующим именем в векторе tasks с использованием лямбда -
выражения
            ptr = find_if(tasks.begin(), tasks.end(), [&](Task& x) { return x.task_name
== current_task; });
// Проверка: если задача не найдена,
            if (ptr != tasks.end()) {
/* то увеличивается переменная task_breakingTime, обозначающая время выполнения задачи
* в контексте ожидания */
                ptr->task_breakingTime++;
            }
```

```
        }
// Цикл, который проходится по всем элементам вектора tasks
        for (auto& k : tasks) {
// Проверка: еслии время начала задачи равно текущему тику,
            if (k.task_startTime == tick) {
// то текущая задача добавляется в конец вектора
                tasks_stack.push_back(k);
            }
        }
 //Проверка: есть ли задача, которая не закончилась?
        if (!tasks_stack.empty() && tasks_stack.back().task_duration <= 0) {
// Если задача завершилась, то она удаляется из вектора tasks_stack
            tasks_stack.pop_back();
// Также идёт увеличение счётчика завершённых задач
            finished_tasksCounter++;
        }
// Проверка: если в стеке ещё остались задачи,
        if (!tasks_stack.empty()) {
// то идёт уменьшение времени выполнения последней задачи в векторе на 1 такт
            tasks_stack.back().task_duration--;
        }
// Проверка: если вектор tasks_stack непустой,
        if (!tasks_stack.empty()) {
/* то полю execution_taskName в объекте current присваивается имя последней задачи в
векторе
* (текущая задача)*/
            current.execution_taskName = tasks_stack.back().task_name;
// Прохождение по всем элементам стека
            for (auto s : tasks_stack) {
// Заполнение очереди именами всех задач, которые пребывают в статусе ожидания
                current.waiting_tasksQueue.push(s.task_name);
            }
        }
// Иначе (нет выполняющихся задач в текущем такте)
        else {
// На текущем такте производится "пустая" итерация
            current.execution_taskName = task_null.task_name;
        }
// Добавление в вектор result объект current
        result.push_back(current);
    }
// Возвращение результата
    return result;
}

// Функция, которая демонстрирует с помощью дополнительного табличного вывода результат
void TaskDispatcherTable(const vector<Tick>& result) {
/* Прохождение по циклу с целью заполнения символом "-" строки, равной 43 элементам
*  (число выбрано из тех соображений, чтобы наименования столбцов были в рамках линий) */
    for (int i = 1; i <= 43; i++) {
            cout << "-";
    }
    cout << endl;

 // Вывод наименований столбцов и ограничительной линии
    printf("|%4s|%10s|%10s|\n", "Тик", "Загрузка процессора", "Очередь ожидания");
    for (int i = 1; i <= 43; i++) {
        cout << "-";
    }
    cout << endl;
// Вывод информации о каждом элементе вектора result
// Прохождение по каждому элементу вектора result
    for (const auto& w : result) {
// Вывод информации о номере текущего такта и имени задачи, которая выполняется в данном
такте
```

```
        cout << setw(3) << w.tick_number << "  | " << setw(6) << w.execution_taskName <<
setw(14) << "     |         ";
 // Создание копии очереди  waiting_tasksQueue
        queue<string> wait = w.waiting_tasksQueue;
// Пока эта копия непуста,
        while (!wait.empty()) {
// Если имя задачи в начале очереди-копии не совпадает с именем выполняющейся задачи,
            if (wait.front() != w.execution_taskName) {
// то это имя выводится в таблице
                cout << wait.front() << "  ";
            }
// Удаление элемента из начала очереди
            wait.pop();
        }
        cout << endl;
    }
// Цикл заполнения символами "-" (ограничительная линия таблицы)
    for (int i = 1; i <= 43; i++) {
       cout << "-";
    }
}

// ГЛАВНАЯ ФУНКЦИЯ MAIN
int main() {
// Установка локали в русском языке
    setlocale(LC_ALL, "Rus");

// Входные данные
// Численная переменная tasks_number, которая обозначает общее количество задач
    int tasks_number;
    cout << "Пожалуйста, введите количество задач: ";
// Ввод пользователем значения и дальнейшее присвоение этого ввода переменной
tasks_number
    cin >> tasks_number;

/* Создание вектора processes типа Task размерностью tasks_number (для хранения задач)*/
    vector<Task> processes(tasks_number);
/* Цикл, который повторяется столько раз, сколько всего задач.
 * Многоповторный ввод пользователем данных о каждой задаче*/
    for (int i = 0; i < tasks_number; ++i) {
        cout << "Пожалуйста, введите имя задачи: ";
// Ввод пользователем имени задачи в векторе processes по индексу i
        cin >> processes[i].task_name;
        cout << "Пожалуйста, введите время активации задачи: ";
// Ввод пользователем такта начала работы задачи в векторе processes по индексу i
        cin >> processes[i].task_startTime;
        cout << "Пожалуйста, введите время выполнения задачи: ";
// Ввод пользователем времени выполнения задачи в векторе processes по индексу i
        cin >> processes[i].task_duration;
    }
/* Создание результирующего вектора result, который инициализируется результатами
 * вызова функции TaskDispatcher */
    vector<Tick> result = TaskDispatcher(processes);
/* Вызов функции TaskDispatcherTable которая демонстрирует с помощью дополнительного
 * табличного вывода результат */
    TaskDispatcherTable(result);
// Завершение выполнения программ
    return 0;
}
```

Результаты работы программы

Результаты продемонстрированы на рисунках 2 - 14.

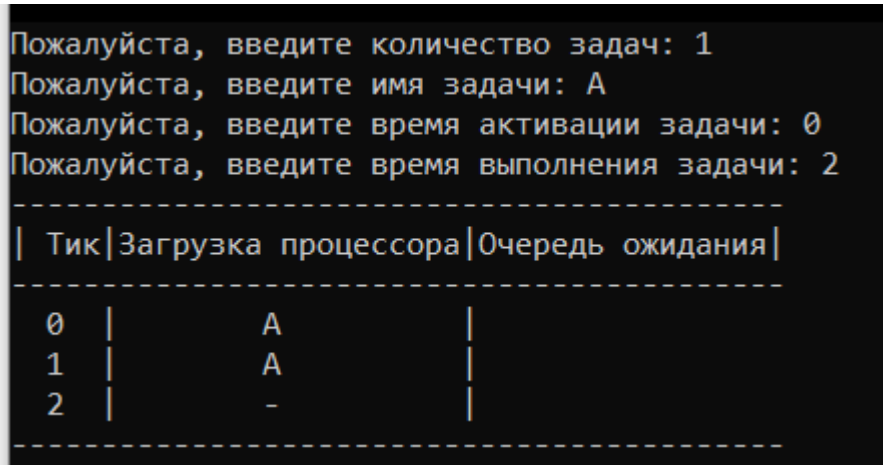

Рисунок 2 – Результат работы программы

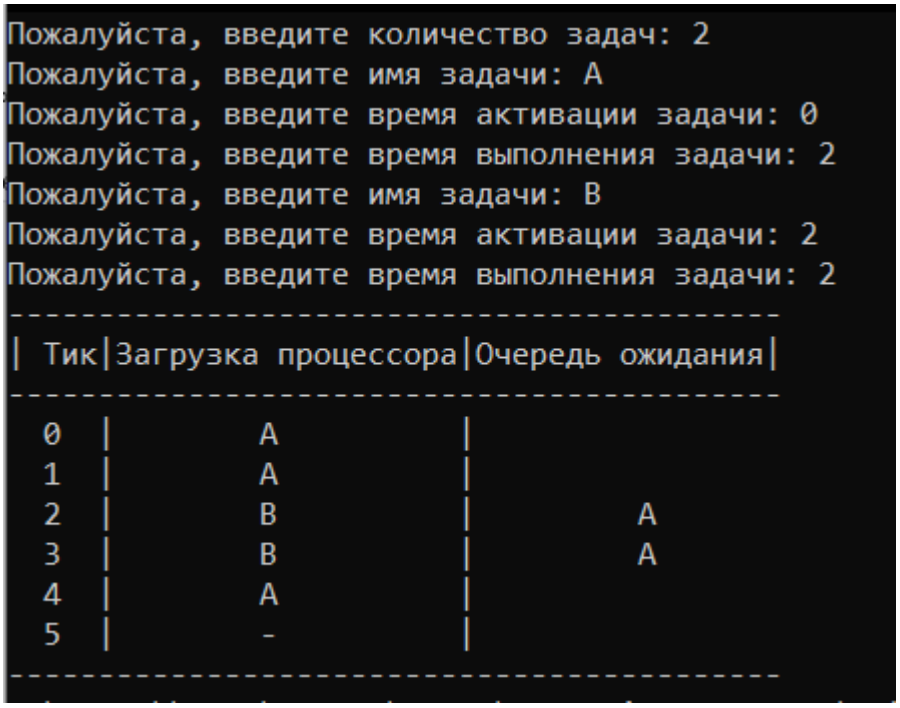

Рисунок 3 – Результат работы программы

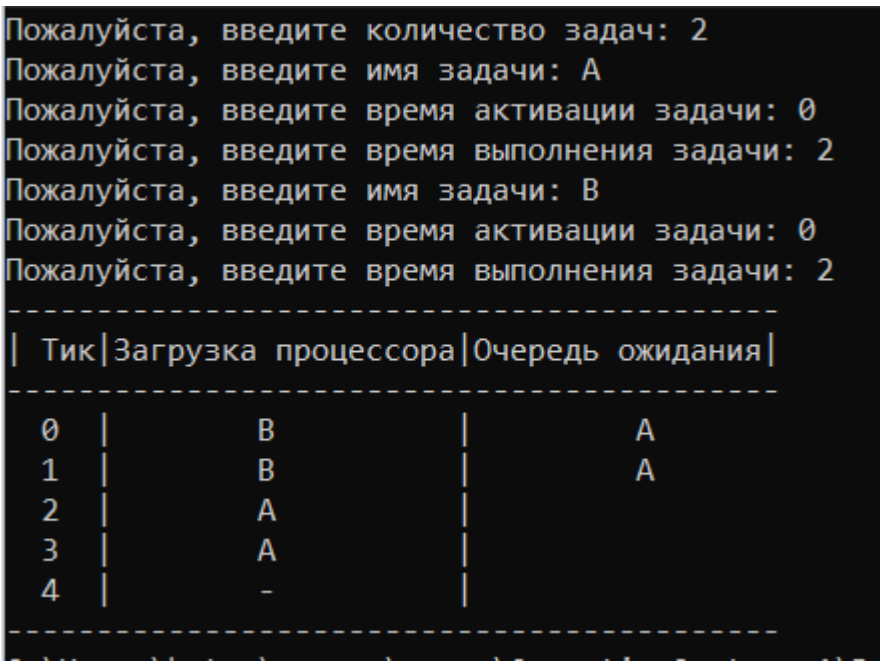

Рисунок 4 – Результат работы программы

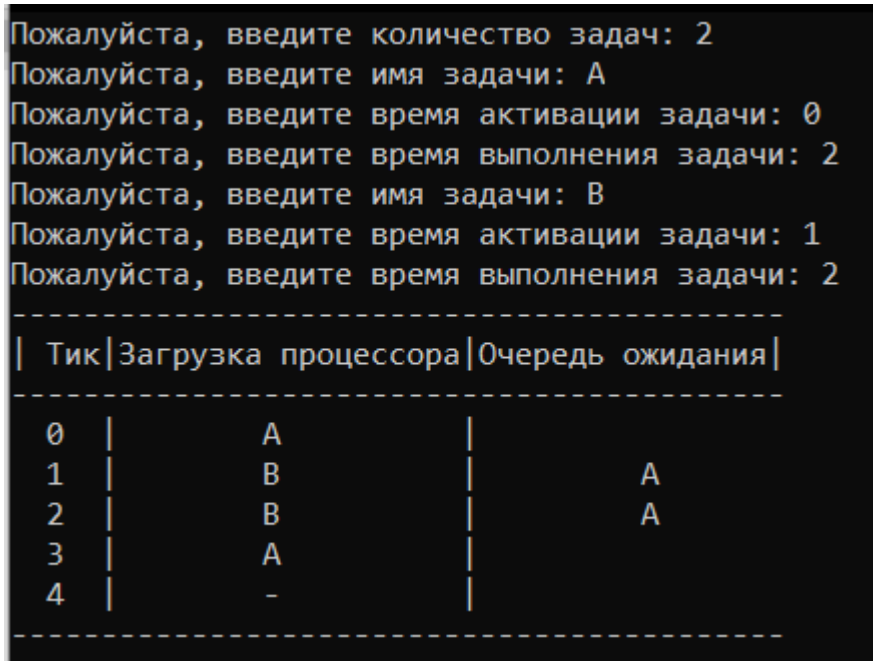

Рисунок 5 – Результат работы программы

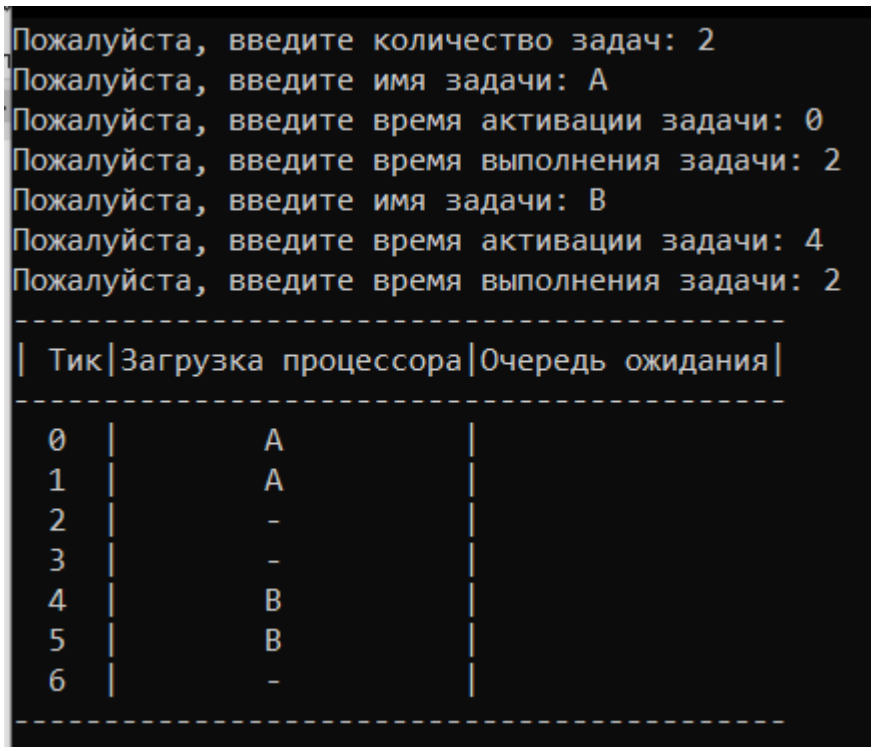

Рисунок 6 – Результат работы программы

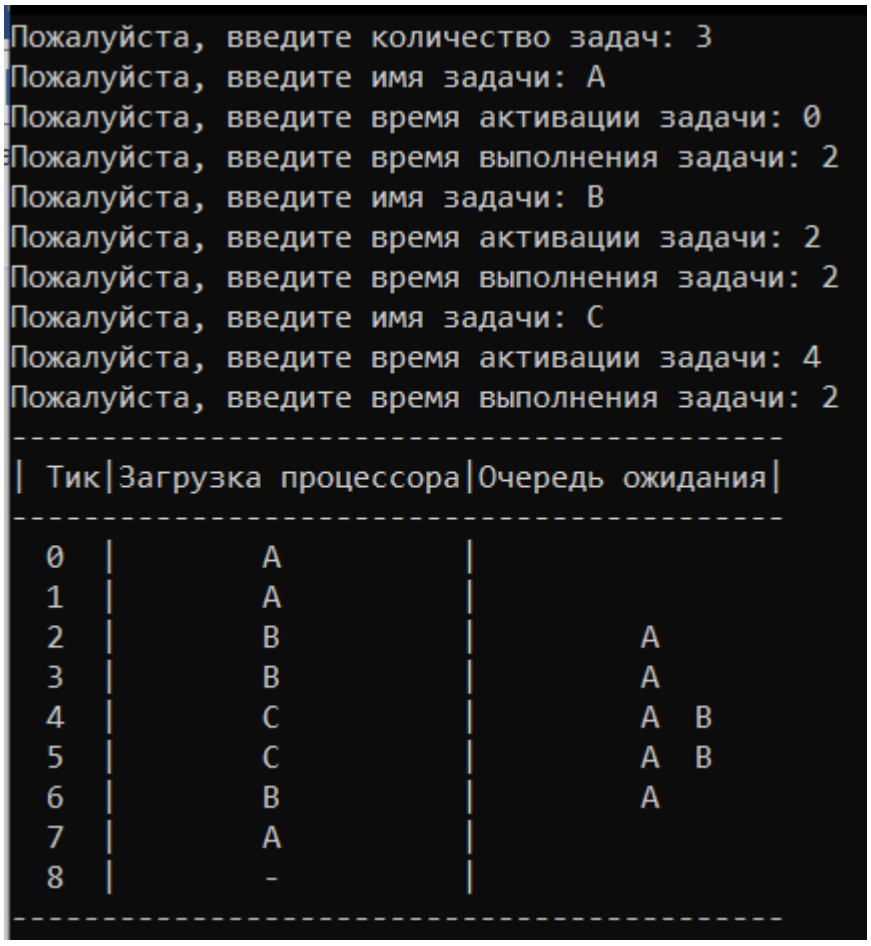

Рисунок 7 – Результат работы программы

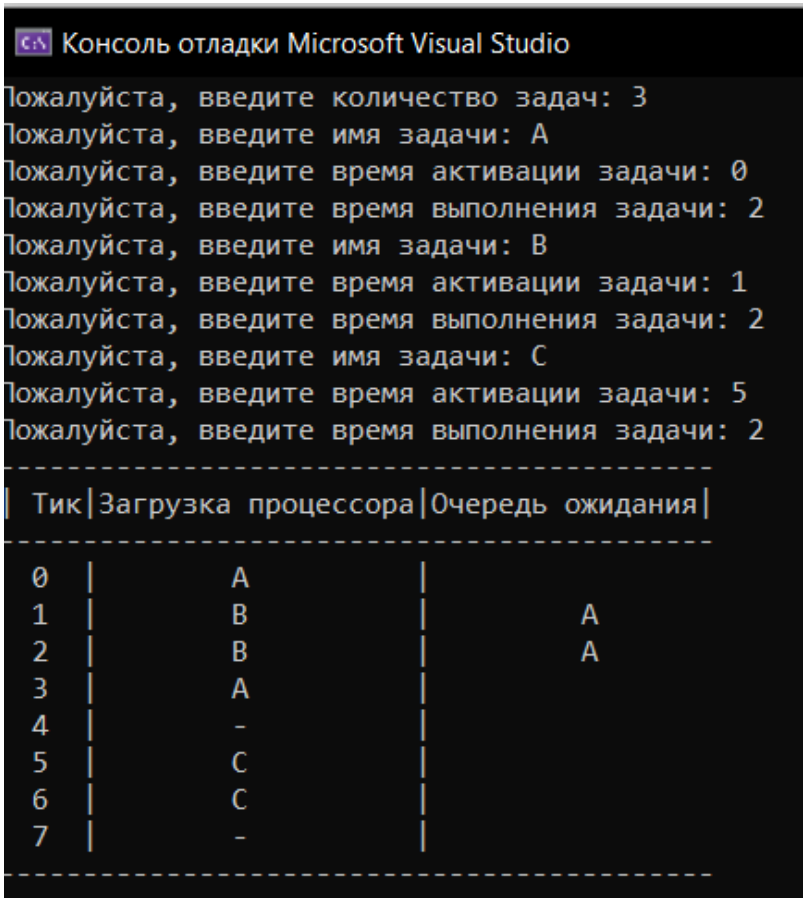

Рисунок 8 – Результат работы программы

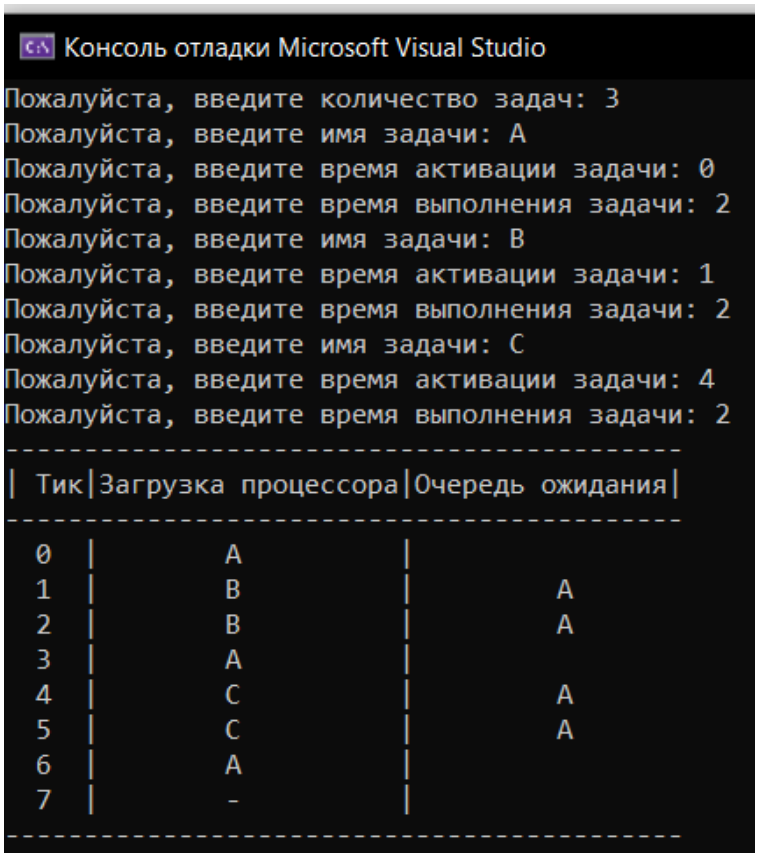

Рисунок 9 – Результат работы программы

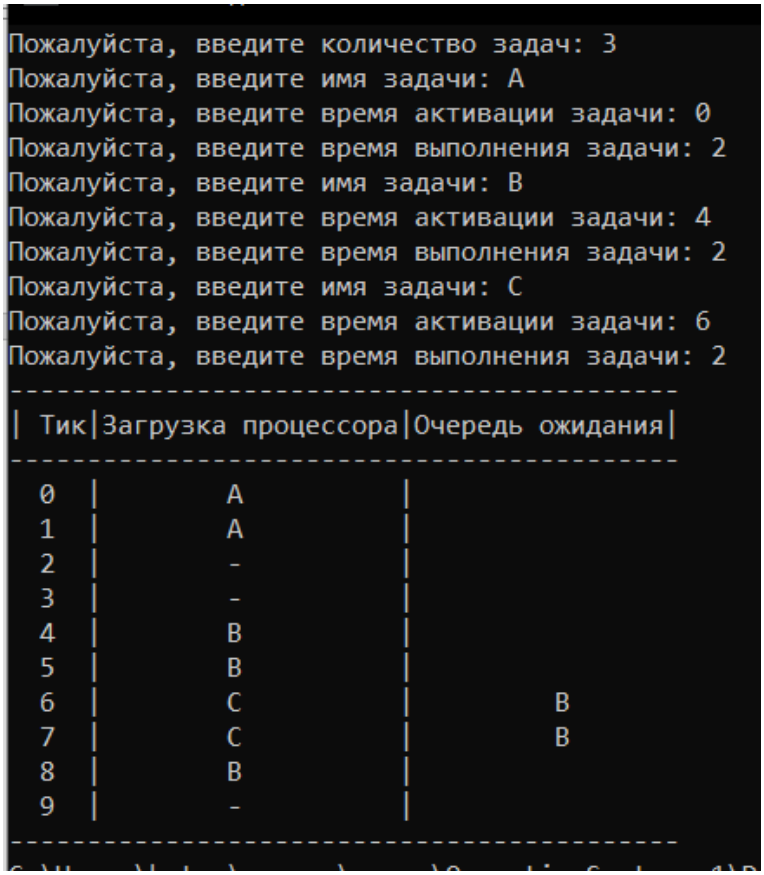

Рисунок 10 – Результат работы программы

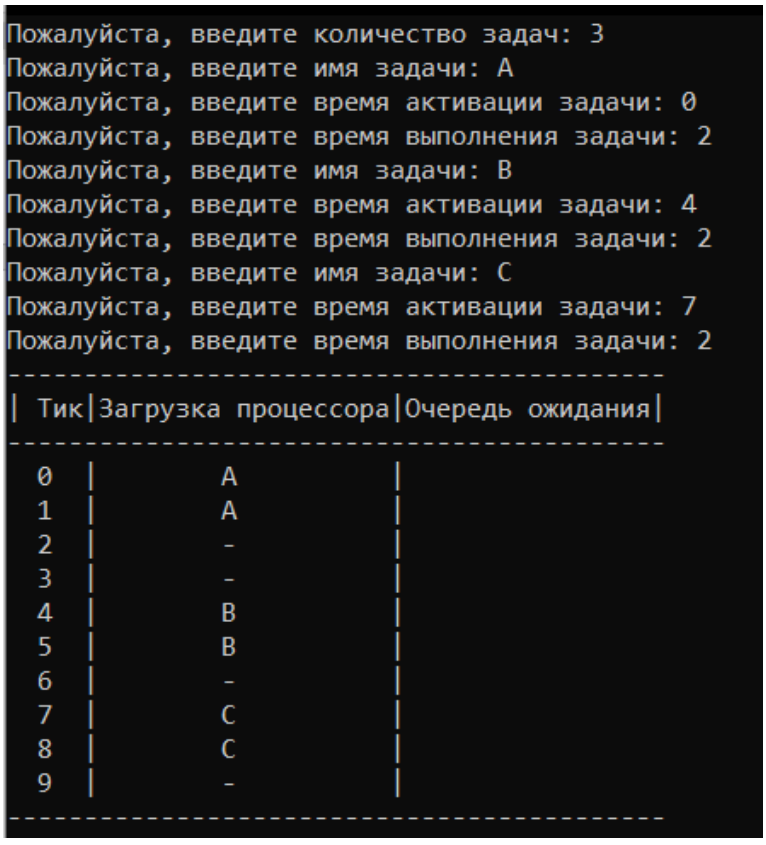

Рисунок 11 – Результат работы программы

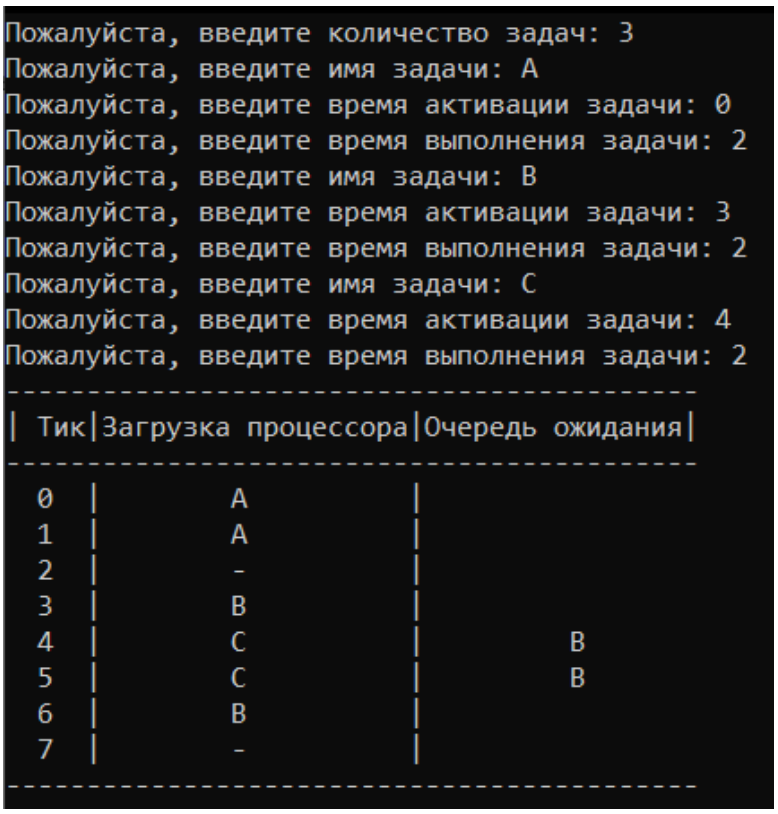

Рисунок 12 – Результат работы программы

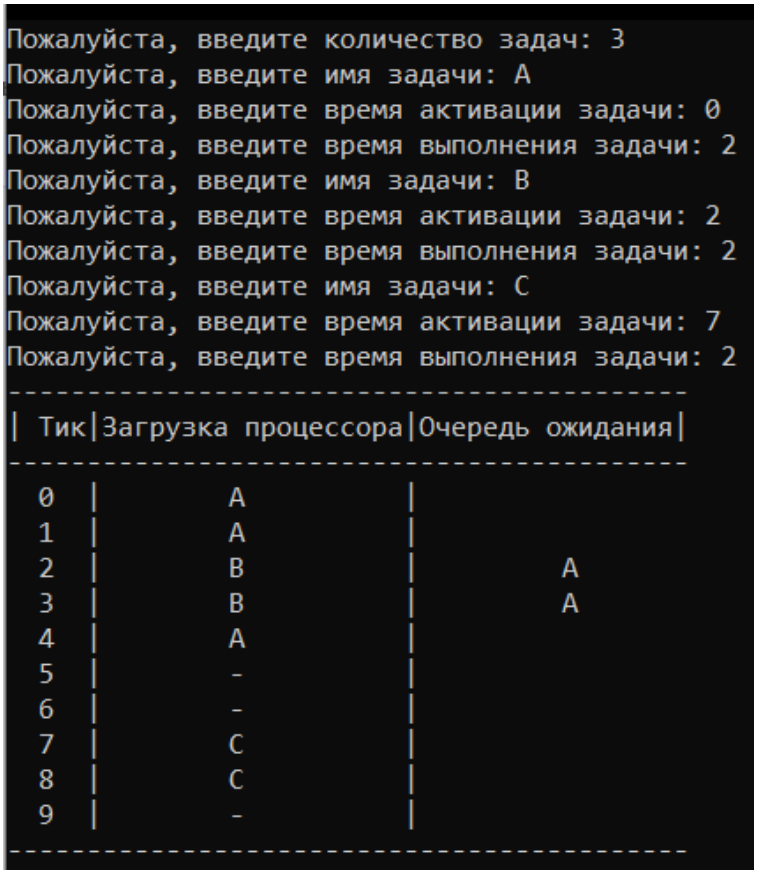

Рисунок 13 – Результат работы программы

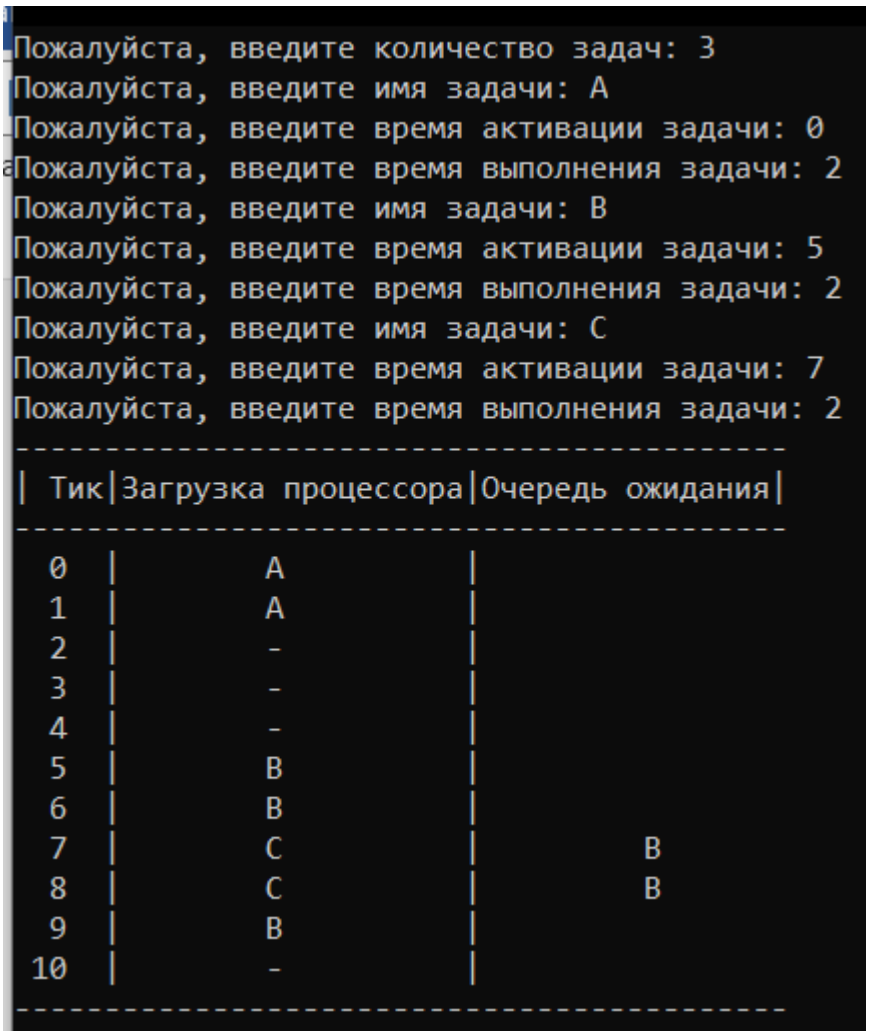

Рисунок 14 – Результат работы программы

Вывод

В результате выполнения данной лабораторной работы была написана и отлажена программа, моделирующая работу диспетчера операционной системы в соответствии с заданной дисциплиной диспетчеризации.