Министерство образования и науки РФ

**ТОМСКИЙ ГОСУДАРСТВЕННЫЙ УНИВЕРСИТЕТ СИСТЕМ**

**УПРАВЛЕНИЯ И РАДИОЭЛЕКТРОНИКИ (ТУСУР)**

**Факультет дистанционного образования (ФДО)**

Кафедра автоматизированных систем управления (АСУ)

**Реализация алгоритмов планирования использования**

**процессорного времени**

Отчет по лабораторной работе №1 по дисциплине

«Теория вычислительных процессов»

**Вариант № 15**

Выполнил: \_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

«\_\_\_\_» \_\_\_\_\_\_\_\_\_\_\_\_\_ 20\_\_ г.

Проверил: \_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

«\_\_\_\_» \_\_\_\_\_\_\_\_\_\_\_\_\_ 20\_\_ г.

Томск— 2016

ОГЛАВЛЕНИЕ

[1. Лабораторное задание 3](#_Toc454346454)

[2. Теоретические основы алгоритмов 5](#_Toc454346455)

[2.1 Многозадачность 5](#_Toc454346456)

[2.2 Семафоры 6](#_Toc454346457)

[2.3. Работа с цепочками 6](#_Toc454346458)

[2.4. Синхронизация цепочек 7](#_Toc454346459)

[3. Структуры данных в программе 9](#_Toc454346460)

[4. Описание формата входных данных 10](#_Toc454346461)

[5. Пример работы программы 11](#_Toc454346462)

[6. Вывод 12](#_Toc454346463)

[Список литературы 13](#_Toc454346464)

[Приложение А (обязательное) Исходный код 14](#_Toc454346465)

# 1. Лабораторное задание

Цель выполнения лабораторной работы № 1 — освоить реализацию алгоритмов планирования использования ресурсов с вытесняющей и невытесняющей многозадачностью, с абсолютным и относительным приоритетом. Освоить реализацию механизмов безопасности и синхронизации потоков, а также механизмов исключения тупиковых ситуаций.

В работе необходимо реализовать ряд алгоритмов распределения ресурсов между конкурирующими процессами. Каждый процесс характеризуется:

* уникальным идентификатором;
* приоритетом;
* временем CPU burst;
* списком требуемых ресурсов;
* дополнительными атрибутами (по вариантам).

Характеристики ресурса:

* уникальный идентификатор;
* наименование ресурса;
* дополнительные атрибуты (по вариантам).

Для имитации времени CPU burst (заданного в миллисекундах) процесс при получении кванта времени на доступ к ресурсу должен делать паузу на указанное количество миллисекунд. Входной файл должен иметь имя «input.txt»

Вариант № 15. Ресурс — преподаватель на экзамене. Атрибуты — ФИО, дисциплина, а также количество студентов N (N ≥ 1), у которых он может принимать экзамен одновременно.

Количество преподавателей — P (P ≥ 1). Атрибуты студента —

ФИО, номер группы и список дисциплин, по которым ему нужно

сдать экзамен. Алгоритмы планирования:

1. SJF, nonpreemptive.

2. SJF, preemptive, относительный приоритет.

Для блокировки доступа к преподавателю использовать семафор.

# 2. Теоретические основы алгоритмов

## 2.1 Многозадачность

В данной программе реализована многозадачность – для этого создается нужное количество цепочек, которые выполняют свою работу.

Также все цепочки синхронизируются с центральной. Это надо для того, чтобы можно было показать состояние цепочек в каждый квант времени.

Цепочки моделируют один из двух алгоритмов, указанных в задании.

Поток — наименьшая единица обработки, исполнение которой может быть назначено ядром операционной системы. Реализация потоков выполнения и процессов в разных операционных системах отличается друг от друга, но в большинстве случаев поток выполнения находится внутри процесса. Несколько потоков выполнения могут существовать в рамках одного и того же процесса и совместно использовать ресурсы, такие как память, тогда как процессы не разделяют этих ресурсов. В частности, потоки выполнения разделяют инструкции процесса (его код) и его контекст (значения переменных, которые они имеют в любой момент времени). В качестве аналогии потоки выполнения процесса можно уподобить нескольким вместе работающим поварам. Все они готовят одно блюдо, читают одну и ту же кулинарную книгу с одним и тем же рецептом и следуют его указаниям, причём не обязательно все они читают на одной и той же странице.

Для реализации задачи используются следующие алгоритмы планирования процессов:

Если короткие задачи расположены в очереди ближе к ее началу, то общая производительность этих алгоритмов значительно возрастает. Если бы мы знали время следующих CPU burst для процессов, находящихся в состоянии ***готовность***, то могли бы выбрать для исполнения не процесс из начала очереди, а процесс с минимальной длительностью CPU burst. Если же таких процессов два или больше, то для выбора можно использовать любой алгоритм, к примеру FCFS. Квантование времени при этом не применяется. Описанный алгоритм получил название “кратчайшая работа первой” или *Shortest Job First* (*SJF*).

Алгоритм SJF может быть как вытесняющим, так и невытесняющим. В случае вытесняющего алгоритма новые могут потеснить уже исполняемые. В случае невытесняюшего задачи после начала выполнения продолжают выполнятся до конца, а только потом происходит выбор другой задачи.

## 2.2 Семафоры

Для синхронизации используется семафор.

Семафор — объект, ограничивающий количество одновременных блокировок, которые могут войти в семафор. Семафоры используются в многозадачных приложениях для синхронизации и защиты передачи данных через разделяемую память, а также для синхронизации работы процессов и потоков. Семафор — это объект, над которым можно выполнить три операции.

* Инициализация семафора (задать начальное значение счётчика) – используется функция CreateSemaphore
* Захват семафора - ждать пока счётчик станет больше 0, после этого уменьшить счётчик на единицу – WaitForSingleObject
* Освобождение семафора - увеличить счётчик на единицу. В зависимости от реализации семафор позволяет освободить счетчик не на 1, а на несколько единиц – Используется функция ReleaseSemaphore

## 2.3. Работа с цепочками

Цепочка это код, который выполняется в отдельном вычислительном процессе. Для программиста и пользователя это выглядит так, что кажется, что код разный код исполняется одновременно в разных программах (или подпрограммах). Для операционной системы (если она работает на одноядерном процессоре) это выглядит как частое переключение между работающими цепочками.

Если у процессора несколько ядер, то возможно исполнять несколько цепочек одновременно. При этом могут возникать проблемы доступа к ресурсам – к примеру если 2 цепочки одновременно читают одну общую переменную, изменяют ее и правят, то вероятен случай, когда результат одной из цепочек будет потерян и вычисления будут неверны. Для того, чтобы это избежать надо применять различные способы блокировки доступа к ресурсам. Это критические секции, семафоры и события.

Для создания цепочек используется функция, к примеру следующим образом:

CreateThread( NULL, 0, (LPTHREAD\_START\_ROUTINE) algo, (void\*) prep[index], 0, NULL );

Для проверки завершения можно использовать функцию WaitForSingleObject или использовать какие-то возможности самой программы. К примеру в текущей задаче все цепочки работают, пока есть студенты.

## 2.4. Синхронизация цепочек

Для синхронизации задач и вывода статистики используется семафор.

В начале такта основная программа ждет готовности всех цепочек. Для этого семафор имеет 0 свободных частей, которые могут быть забраны цепочками.

Это реализуется следующей командой:

semaphore = CreateSemaphore( NULL, 0, prep.size(), NULL );

Таким образом создается пустой семафор.

Каждая цепочка вначале рапортует о готовности и ждет входа в семафор. После готовности всех цепочек основная цепочка выводит статистику о исполнении и разблокирует семафор с количеством единиц, равных количеству цепочек. Это происходит с помощью следующей команды:

ReleaseSemaphore( semaphore, prep.size(), NULL );

После этого все начинается с начала – она ждет готовности всех цепочек.

# 3. Структуры данных в программе

В программе используется классы, которые содержат данные программы:

class TPrepodavatel

{

public:

char name[100], disc[100]; // Имя и дисциплина

int razmsd; // Количество одновременных сдач

vector<TOcenka\*> prinimaet; // Студенты, которые будут сдавать

int index;

bool gotov;

TPrepodavatel( ifstream &input, int \_index )

int not\_finish();

TOcenka \*find\_min\_t( bool pri = false );

};

// Класс оценки

class TOcenka

{

public:

TStudent \*stud;

TPrepodavatel \*prepod;

int time; // Времени до получения оценки

bool start, finish; // А мы начали/кончили сдавать?

TOcenka( TStudent \*s, ifstream &input )

};

// Класс студента

class TStudent

{

public:

char name[100];

vector<TOcenka\*> sdacha; // Список дисциплин для сдачи

int t, pri, last\_sd;

bool used; // А мы использовали его в текущем такте?

TStudent( ifstream &input )

} ;

Эти классы содержать все необходимые данные

# 4. Описание формата входных данных

Входные данные описаны в следующим образом:

В файле input.txt содержа тся строки следующего формата:

* В первой строке тип алгоритма планирования (sjfpreemptive или sjfnonpreemptive)
* Во второй строке содержится количество преподавателей и студентов, а также параметр cpu burst, описывающий, сколько за каждый такт обробатывается количество частей сдачи студента.
* В следующих строках содержится инфомация о преподавателях – имя, предмет и количество одновременно сдаваемых студентов.
* В следующих строках содержится информация о студенте – имя, количество предметов для сдачи и время сдачи одного предмета (считается одинаковым для одного студента), а также приоритет. Также для каждого студента идет список предметов для сдачи.

Пример файла:

sjfnonpreemptive Алгоритм

3 8 1 Преподавателей, студентов, cpu burst

ivanov math 3 Имя преподавателя, предмет, размер очереди

petrov rusyaz 2

lebedev fiz 2

vasiliev 3 10 3 Имя студента, кол. предметов, время сдачи, приоритет

math fiz rusyaz Список предметов для сдачи (3 штуки)

karev 3 8 2

math fiz rusyaz

lavrentiev 2 7 2

math fiz

vasiliev 1 2 2

rusyaz

avdeev 2 9 1

rusyaz fiz

andreev 2 2 1

fiz rusyaz

golubev 1 4 0

fiz

zimin 3 6 4

rusyaz math fiz

В случае ошибки (предмет не найден) программа выводит сообщение про это и не запускается.

Также запуск не произойдет, если неправильно вписан алгоритм планирования.

# 5. Пример работы программы

В выходном отчете выводится следующая информация:

Каждая колонка – это сдачи одного студента.   
Сдачи объединены в группы по каждому студенту.

В первой строке информация о номере преподавателя, который должен принять экзамен у студента.

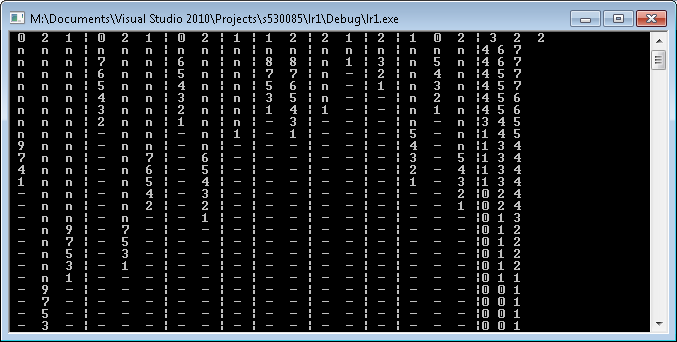
В конце указаны числа – размеры очередей у преподавателей.

Каждая из сдач находится в трех разных состояниях

* не начал сдавать (буква n)
* сдал (подчеркивание)
* число - время, оставшееся до окончания сдачи.

В конце показывается размер очереди преподавателя (в скобках) и количество студентов, которые сейчас сдают.

Последнее число – текущий студент, который встает в нужную очередь. Основная цепочка по очереди свободных в конце прошлого такта студентов, таким образом студент может сдавать только один экзамен одновременно.



К примеру здесь видно, что профессор 0 имеет очередь в 3 студента, поэтому после первого такта он обработал 1, 2 и 5 студента, а остальные ждут (или сдают других преподавателей)

# 6. Вывод

В данной работе мы научились писать многопоточные приложения и применять алгоритмы SJF в вытесняющем и невытесняющем режиме, а также использовать семафор для синхронизации цепочек.

# Список литературы

1. Рихтер Дж. Windows для профессионалов / Дж. Рихтер. — СПб. : Питер, 2000. — 752 с.
2. Калайда В.Т. Теория вычислительных процессов: учеб. пособие / В. Т. Калайда. — Томск : Изд-во ТМЦДО, 2013.
3. Вебер Дж. Технология Win32 в примерах / Дж. Вебер. — СПб. : БХВ-Петербург, 1998. — 1070 с.
4. Таненбаум Э. Современные операционные системы / Э. Таненбаум. — 3-е изд. — СПб. : Питер, 2010. — 1120 с.
5. Воеводин В.В. Параллельные вычисления. — СПб.: БХВ- Петербург, 2002. — 608 с.

# Приложение А (обязательное) Исходный код

//

#include "stdafx.h"

#include <vector>

#include <Windows.h>

#include <process.h>

#include <conio.h>

#include <iostream>

#include <fstream>

using namespace std;

class TPrepodavatel;

class TStudent;

class TOcenka;

vector<TPrepodavatel\*> prep;

vector<TStudent\*> stud;

// КЛасс преподавателя

class TPrepodavatel

{

public:

char name[100], disc[100]; // Имя и дисциплина

int razmsd; // Количество одновременных сдач

vector<TOcenka\*> prinimaet; // Студенты, которые будут сдавать

int index;

bool gotov;

TPrepodavatel( ifstream &input, int \_index )

{

gotov = false;

index = \_index;

input >> name >> disc >> razmsd;

}

int not\_finish();

TOcenka \*find\_min\_t( bool pri = false );

};

// Класс оценки

class TOcenka

{

public:

TStudent \*stud;

TPrepodavatel \*prepod;

int time; // Времени до получения оценки

bool start, finish; // А мы начали/кончили сдавать?

TOcenka( TStudent \*s, ifstream &input )

{

char n[100];

input >> n;

stud = s;

prepod = NULL; // \*/

for( int i = 0; i < prep.size(); i++ )

if (strcmp( prep[i]->disc, n ) == 0)

prepod = prep[i];

if (prepod == NULL)

{

printf( "Error in input.txt\n" );

return;

}

start = false;

finish = false;

}

};

// Класс студента

class TStudent

{

public:

char name[100];

vector<TOcenka\*> sdacha; // Список дисциплин для сдачи

int t, pri, last\_sd;

bool used; // А мы использовали его в текущем такте?

TStudent( ifstream &input )

{

int c;

input >> name >> c >> t >> pri;

for( int index = 0; index < c; index++ )

sdacha.push\_back( new TOcenka( this, input ) );

last\_sd = 0;

used = false;

}

} ;

int qt; // Количество миллисекунд

HANDLE semaphore;

int TPrepodavatel::not\_finish() {

int r = 0, index;

for( index = 0; index < prinimaet.size(); index++ )

if (!prinimaet[index]->finish)

r++;

return r;

}

// Функция ищет среди неиспользованных (!used) минимум

TOcenka\* TPrepodavatel::find\_min\_t( bool pri ) {

int index, min\_i;

for( int cpri = 0; cpri < 4; cpri++ )

{

min\_i = -1;

for( int index = 0; index < prinimaet.size(); index++ )

if (!prinimaet[index]->stud->used & !prinimaet[index]->finish)

{

// Если не текущий приоритет - пропустить

if (pri & (prinimaet[index]->stud->pri != cpri)) continue;

if ( (min\_i < 0) || (prinimaet[index]->time < prinimaet[min\_i]->time) )

min\_i = index;

}

// Если с приоритетами и ничего не нашли в текущем

if (pri & (min\_i < 0)) continue;

// Что-то нашли

if (min\_i >= 0)

{

prinimaet[min\_i]->stud->used = true;

return prinimaet[min\_i];

}

}

return NULL;

}

// Завершение

bool konec = false;

// Вытесняющий алгоритм или нет

bool preemptive = false;

// Ищем минимальных и только при завершении ищем другого студента

void algo( void \*data )

{

TPrepodavatel \*prep = (TPrepodavatel\*) data;

int index, si, index2, pri;

TOcenka\* ocenki[10];

memset( &ocenki, 0, sizeof(ocenki) );

while (!konec)

{

prep->gotov = true;

WaitForSingleObject( semaphore, INFINITE );

for( index = 0; index < prep->razmsd; index++ )

{

// Если в текущем месте ничего нет или алгоритм вытесняющий

if ( (ocenki[index] == NULL) | preemptive )

// Алгоритм вытесняющий с приоритетом (и наоборот)

ocenki[index] = prep->find\_min\_t( !preemptive );

if (ocenki[index] != NULL)

{

// Обработает

ocenki[index]->start = true;

ocenki[index]->time -= qt;

// Завершаем, на следующем такте найдем нового (возможно)

if (ocenki[index]->time <= 0)

{

ocenki[index]->finish = true;

ocenki[index] = NULL;

}

}

}

Sleep(1);

}

}

int alg;

// Найти студента, который не сдает

bool naiti\_studenta( int &studi, int &sdachai )

{

int index, index2;

bool fl;

TOcenka \*sd;

for( index = 0; index < stud.size(); index++ )

{

// В текущем такте занят

if (stud[index]->used) continue;

fl = true;

for( index2 = 0; (index2 < stud[index]->sdacha.size()) & fl; index2++ )

{

sd = stud[index]->sdacha[index2];

// Студент чем-то занят

if (sd->start & !sd->finish)

fl = false;

}

// Студент ничем не занят

if (fl)

{

if (stud[index]->last\_sd < stud[index]->sdacha.size())

{

studi = index;

sdachai = stud[index]->last\_sd++;

return true;

}

}

}

return false;

}

int \_tmain(int argc, \_TCHAR\* argv[])

{

// Читаем вход

ifstream input;

input.open( "input.txt" );

char buf[100];

int index, index2, count1, count2;

TStudent \*s;

TOcenka \*ocenka;

// input.getline( buf,100 );

input >> buf;

int stud\_c, prep\_c, c;

input >> count1 >> count2 >> qt;

for( index = 0; index < count1; index++ )

prep.push\_back( new TPrepodavatel( input, index ) );

for( index = 0; index < count2; index++ )

{

stud.push\_back( s = new TStudent( input ) );

for( index2 = 0; index2 < s->sdacha.size(); index2++ )

printf( "%2d ", s->sdacha[index2]->prepod->index );

printf( "|" );

}

printf( "\n" );

input.close ();

// Семафор для синхронизации

semaphore = CreateSemaphore( NULL, 0, prep.size(), NULL );

if (strcmp( buf, "sjfnonpreemptive" ) == 0)

{

for( index = 0; index < prep.size(); index++ )

CreateThread( NULL, 0, (LPTHREAD\_START\_ROUTINE) algo, (void\*) prep[index], 0, NULL );

}

else if (strcmp( buf, "sjfpreemptive" ) == 0)

{

preemptive = true;

}

else

printf( "Neizvestii algoritm\n" );

while (!konec)

{

// Дожидаемся пока все будут готовы

while(true)

{

bool gotov = true;

for( index = 0; index < prep.size(); index++ )

if (!prep[index]->gotov)

{

gotov = false;

break;

}

if (gotov)

break;

Sleep(1);

}

konec = true;

for( index = 0; index < prep.size(); index++ )

prep[index]->gotov = false;

int studi, sdachai;

if (naiti\_studenta( studi, sdachai ))

{

s = stud[ studi ];

ocenka = s->sdacha[ sdachai ];

ocenka->time = s->t;

ocenka->prepod->prinimaet.push\_back( ocenka );

}

// Выводим статистику

int t = 0;

for( index = 0; index < stud.size(); index++ )

{

s = stud[index];

for( index2 = 0; index2 < s->sdacha.size(); index2++ )

{

ocenka = s->sdacha[index2];

// t++;

if (ocenka->finish)

printf( " - " );

else

{ // Кто-то еще не завершился

konec = false;

if (!ocenka->start)

printf( " n " );

else

printf( "%2d ", ocenka->time );

}

} // \*/

s->used = false;

printf( "|" );

}

// Сколько человек еще не сдало

for( index = 0; index < prep.size(); index++ )

printf( "%d ", prep[index]->not\_finish () );

printf( "\n" );

ReleaseSemaphore( semaphore, prep.size(), NULL );

Sleep(10);

}

printf( "end\n" );

\_getch();

return 0;

}