**КУРСОВАЯ РАБОТА**

Разработка автоматизированной системы «Модель операционной системы реального времени» по дисциплине «Архитектура вычислительных систем»

Выполнил студент гр. 33504/4 Череповский Д.К.

Руководитель доцент Коликова Т.В.

«\_\_\_» \_\_\_\_\_\_\_\_\_\_\_ 201\_ г.

**Оглавление**

[**1.** **Постановка задачи** 3](#_Toc509758072)

[**2.** **Описание структуры проекта** 4](#_Toc509758073)

[**3.** **Код программы** 6](#_Toc509758074)

# **Постановка задачи**

Необходимо реализовать модель операционной системы реального времени, которая

обладает следующими свойствами:

* **Тип планировщика:** плоский;
* **Алгоритм планирования:** preemptive, RMA;
* **Управление ресурсами:** PIP;
* **Управление событиями:** события принадлежат задаче;
* **Обработка прерываний:** нет;
* **Максимальное количество задач:** 32;
* **Максимальное количество ресурсов:** 16;
* **Максимальное количество событий:** 16;

Кроме того, в задание входит написание тестов, проверяющих соответствие проекта этим свойствам.

**Программные средства**

Данный проект был выполнен на языке программирования C++.

**Вступление**

ОС общего назначения, особенно многопользовательские, ориентированы на оптимальное распределение ресурсов компьютера между пользователями и задачами (системы разделения времени). В **операционных системах реального времени** (ОСРВ), подобная задача отходит на второй план - все отступает перед главной задачей - **успеть среагировать на события, происходящие на объекте**.

**Вытесняющие алгоритмы планирования (preemptive)**

Preemptive multitasking - вытесняющая многозадачность - это такой способ, при котором **решение о переключении процессора с выполнения одного процесса на выполнение другого процесса принимается планировщиком операционной системы**, а не самой активной задачей.

При вытесняющей многозадачности механизм планирования задач целиком сосредоточен в операционной системе, и программист пишет свое приложение, не заботясь о том, что оно будет выполняться параллельно с другими задачами. При этом операционная система выполняет следующие функции: определяет момент снятия с выполнения активной задачи, запоминает ее контекст, выбирает из очереди готовых задач следующую и запускает ее на выполнение, загружая ее контекст.

**Задача (task)** - единица обработки, выполняющаяся конкурентно с другими задачами. Задачи являются основным средством обработки внутренних событий. Задача имеет некоторое значение приоритета, определяющее ее относительные претензии на захват процессора. Эти претензии удовлетворяются ОС по определенному алгоритму. Вместо приоритета может использоваться значение срока исполнения. Задача имеет точку входа (entry point).

# **Описание структуры проекта**

1. **Управление ОС (файл os.cpp)**

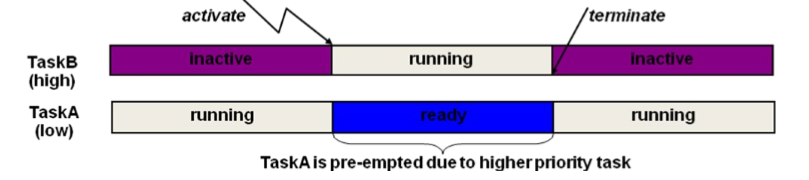
**StartOS(TTaskCall entry,int priority,char\* name)** - осуществляет необходимую инициализацию и запускает ОС на выполнение;

**ShutdownOS()** - используется для немедленного завершения работы системы; **InitializeStacks(int numStack)** - инициализация стеков задач;

1. **Управление задачами (файл task.cpp)**

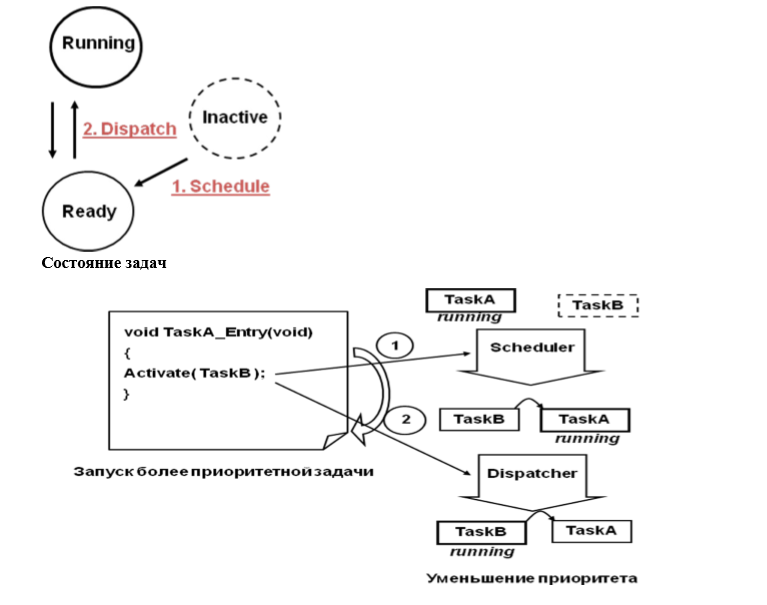
Планирование.

Наглядно вытесняющее планирование можно представить так:



**Планирование** состоит из двух шагов, выполняется друг за другом:

1. Собственно, планирование (scheduling), т.е. составления расписания.
2. Диспетчеризация (dispatching) – выборка задачи из списка и назначение ей процессора (переключение контекста)



**ActivateTask(TTaskCall entry,int priority,char\* name)** - активация задачи entry, а также задание приоритета и названия задачи.

**TerminateTask(void)** - завершение задачи.

**Schedule(int task, int dont\_show = 0)** - добавление задачи в очередь активных задач. Постановка осуществляется в хвост подсписка задач с равным приоритетом.

**Dispatch()** - выполняет переключение задач. Сохранение контекста, выбор следующей задачи и восстановление контекста выбранной задачи путем вызова функции TaskSwitch.

**InsertTaskBefore(int task, int item, int \*head)** - добавление задачи в список перед элементом item, в переменной head хранится указатель на голосу списка.

**InsertTaskAfter(int task, int item)** - добавление задачи после элемента item.

**RemoveTask(int task, int \*head)** - удаление задачи из списка, head указывает на начало списка.

**TaskSwitch(int nextTask)** - в данной функции выполняется восстановление контекста задачи nextTask.

1. **Управление ресурсами (файл resource.cpp)**

Для взаимоисключаемости доступа задачи (и обработчиков прерывай) к критическим секциям кода, т.е. к разделяемым ресурсам.

Задача

Задача

I/O

**InitRes(int ResNum, char\* name)** - инициализация ресурса. Осуществляет регистрацию ресурса в системе. Должен вызываться ДО использования ресурса в коде пользовательского приложения;

**PIP\_GetRes(int ResNum)** - захват ресурса. Вызов этого сервиса отмечает вход в критическую секцию. Выход из этой критической секции всегда должен осуществляться вызовом;

**PIP\_ReleaseRes(int ResNum)** - освобождение ресурса. Вызов этого сервиса отмечает выход из критической секции;

1. **Управление событиями (файл event.cpp)**

События (events) предназначены для обмена двоичными данными между задачами и обработчиками прерываний.

События также реализуются в виде флагов (masks, flags) или сигналов (signals).

Задача

Задача

Задача

Задача

Задача

**DeclareEvent(ID)** - осуществляет регистрацию события в системе (назначение подобно объявлению глобальных функций в языке С). Должен вызываться ДО использования идентификатора события в коде пользовательского приложения(test.cpp);

**SetEvent(int task, TEventMask mask)** - устанавливает для задачи task события, задаваемые маской mask. При вызове данного сервиса задача tesk будет переведена в готовое состояние, если она ждала хотя бы одного из событий, заданных маской mask;

**GetEvent(int task, TEventMask\* mask)** - возвращает текущее состояние маски установленных событий для задачи task;

**WaitEvent(TEventMask mask)** - если не установлено ни одного события, заданного маской, то задача переводится в состояние ожидания. При выходе из данного сервиса будут сброшены те события, которые вызвали этот выход;

**ClearEvent (TEventMask mask)** - сервис производит очистку признаков событий (в маске установленных событий задачи) в соответствии с параметром mask;

1. **Файл test.cpp** - пользовательское приложение.

В файле записываются декларация Task, Resource, Event

В ней также находиться программа main(), запускающая программу

# **Код программы**

**Заголовочные файлы**

**Файл sys.h**

#ifndef SYS\_H

#define SYS\_H

/\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/

/\* sys.h \*/

/\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/

#include "defs.h"

#include <csetjmp>

enum TTaskState { *TASK\_RUNNING* = 0, *TASK\_READY* = 1, *TASK\_SUSPENDED* = 2,

*TASK\_WAITING* = 3 };

typedef unsigned int TEventMask;

typedef struct Type\_Task

{

int next, prev; //след и пред элем в стеке

int priority; //чем меньше число, тем меньше приоритет

int ceiling\_priority; //переменная для сравнив с больш приорит

void (\*entry)(void);

char\* name;

TTaskState task\_state;

int switch\_count;

jmp\_buf context;

TEventMask waiting\_events;

TEventMask working\_events; //сработавшие события

} TTask;

typedef struct Type\_resource

{

int task;

char\* name;

} TResource;

extern jmp\_buf InitStacks[MAX\_TASK];

extern TTask TaskQueue[MAX\_TASK];

extern TResource ResourceQueue[MAX\_RES];

extern int RunningTask; //текущая выполняющаяся задача, изменять только в функции TaskSwitch

extern int TaskHead; //первая задача в списке, голова

extern int TaskCount;

extern int FreeTask;

extern jmp\_buf MainContext; //главный контекст

void **Schedule**(int task, int dont\_show = 0);

void **Dispatch**();

#endif // SYS\_H

**Файл rtos\_api.h**

#ifndef RTOS\_API\_H

#define RTOS\_API\_H

/\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/

/\* rtos\_api.h \*/

/\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/

#define DeclareTask(TaskID,priority)\

TASK(TaskID); \

enum {TaskID##prior=priority}

#define TASK(TaskID) void TaskID(void)

typedef void **TTaskCall**(void);

int **ActivateTask**(TTaskCall entry,int priority,char\* name);

void **TerminateTask**(void);

int **StartOS**(TTaskCall entry,int priority,char\* name);

void **ShutdownOS**();

void **InitRes**(int ResNum, char\* name);

void **PIP\_GetRes**(int ResNum);

void **PIP\_ReleaseRes**(int ResNum);

#define DeclareEvent(ID) \

const int Event\_##ID = (ID)\*(ID);

void **SetEvent**(int task, TEventMask mask);

void **GetEvent**(int task, TEventMask\* mask);

void **WaitEvent**(TEventMask mask);

void **ClearEvent** (TEventMask mask);

#endif // RTOS\_API\_H

**Файл defs.h**

#ifndef DEFS\_H

#define DEFS\_H

/\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/

/\* defs.h \*/

/\*\*\*\* \*\*\*\*\*\*\*\*\*\*\*\*\*/

#define MAX\_TASK 32

#define MAX\_RES 16

#define MAX\_EVENT 16

#define MAX\_PRIORITY 16

#endif // DEFS\_H

**Исходные файлы**

**Файл event.cpp**

/\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/

/\* event.c \*/

/\*\* \*\*\*\*\*\*\*\*\*\*\*\*\*\*/

#include <stdio.h>

#include "sys.h"

#include "rtos\_api.h"

//Устанавливает для задачи события, задаваемые маской

void **SetEvent**(int task, TEventMask mask)

{

printf("SetEvent %i for task %s\n", mask, TaskQueue[task].name);

TaskQueue[task].working\_events |= mask;

if(TaskQueue[task].task\_state == *TASK\_WAITING* &&

(TaskQueue[task].working\_events & TaskQueue[task].waiting\_events))

{

TaskQueue[task].waiting\_events &= !mask;

TaskQueue[task].working\_events &= !mask;

TaskQueue[task].task\_state = *TASK\_READY*;

Dispatch();

}

printf("End of SetEvent %i for task %s\n", mask, TaskQueue[task].name);

}

//Возвращает текущее состояние маски установленных событий для задачи

void **GetEvent**(int task, TEventMask\* mask)

{

\*mask = TaskQueue[task].working\_events;

}

//Задача переводится в состояние ожидания

void **WaitEvent**(TEventMask mask)

{

printf("WaitEvent %i to task %s\n", mask, TaskQueue[RunningTask].name);

TaskQueue[RunningTask].waiting\_events = mask;

if((TaskQueue[RunningTask].working\_events & mask) == 0)

{//ждемс

TaskQueue[RunningTask].task\_state = *TASK\_WAITING*;

Dispatch();

}

printf("End of WaitEvent %i to task %s\n", mask, TaskQueue[RunningTask].name);

}

//Очистка признаков событий (в маске установленных событий задачи) в соответствии с параметром маски

void **ClearEvent** (TEventMask mask)

{

TaskQueue[RunningTask].working\_events &= !mask;

}

**Файл global.cpp**

/\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/

/\* global.c \*/

/\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/

#include "sys.h"

TTask TaskQueue[MAX\_TASK];

TResource ResourceQueue[MAX\_RES];

int RunningTask;

int FreeTask;

int FreeResource;

**Файл os.cpp**

/\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/

/\* os.c \*/

/\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/

#include <stdio.h>

#include "sys.h"

#include "rtos\_api.h"

//Инициализация стеков

void **InitializeStacks**(int numStack)

{

char cushionSpace[100000];

cushionSpace[99999] = 1; // отключаем оптимизацию массивов

if(!setjmp(InitStacks[numStack]))

{//инициализация

if(numStack <= MAX\_TASK)

InitializeStacks(numStack+1);

}

else

{ //выполнение

TaskQueue[RunningTask].entry();

}

}

//Программа стартует с этой функции

int **StartOS**(TTaskCall entry,int priority,char\* name)

{

int i;

RunningTask = TaskHead = -1; //текущая выполняющаяся задача нет списка

TaskCount = 0; //подсчет задач

FreeTask=0; //ожидающие задачи

printf("StartOS!\n");

InitializeStacks(0); //инициализация стека

for(i=0;i<MAX\_TASK;i++)

{

TaskQueue[i].next=i+1; //номер массива на след элемент

TaskQueue[i].prev = i - 1; //пред элем

TaskQueue[i].task\_state = *TASK\_SUSPENDED*; //пометка ожид(неактив задач)

TaskQueue[i].switch\_count = 0; // ключ

TaskQueue[i].waiting\_events = 0; //ожидание задачи

TaskQueue[i].working\_events = 0; //сработавшие задачи

}

//создание массива задач

TaskQueue[MAX\_TASK-1].next=0; //присваиваем последнему элементу 0

TaskQueue[0].prev=MAX\_TASK-1; //присваиваем послед предпоследний

if(!setjmp(MainContext)) //если не 0

ActivateTask(entry,priority,name); //запускаем функцию активации задачи

return 0;

} //Завершает работу системы(задача завершена)

void **ShutdownOS**()

{

printf("ShutdownOS!\n");

}

**Файл resource.cpp**

/\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/

/\* resource.c \*/

/\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/

#include "sys.h"

#include "rtos\_api.h"

#include <stdio.h>

void **InitRes**(int ResNum, char\* name) //инициализ ресурса

{

printf("InitRes %s\n",name);

ResourceQueue[ResNum].name = name;

ResourceQueue[ResNum].task = -1;

}

void **PIP\_GetRes**(int ResNum) //ресурс запрашиваемый задачей

{

printf("PIP\_GetRes %s\n",ResourceQueue[ResNum].name);

while(ResourceQueue[ResNum].task != -1)

{// ресурс заблокирован

int rtask = ResourceQueue[ResNum].task;

printf("Resource is blocked by %s\n", TaskQueue[rtask].name);

if(TaskQueue[rtask].ceiling\_priority <= TaskQueue[RunningTask].ceiling\_priority)

{ //увеличив времен приоритет на 1

printf("Inverse prioritet from %i to %i\n",

TaskQueue[rtask].ceiling\_priority,

TaskQueue[RunningTask].ceiling\_priority+1);

TaskQueue[rtask].ceiling\_priority =

TaskQueue[RunningTask].ceiling\_priority+1;

Schedule(rtask); //добавляем в очередь

}

Dispatch(); //переключение

}

ResourceQueue[ResNum].task = RunningTask; //ресурс занят

printf("End of PIP\_GetRes %s\n",ResourceQueue[ResNum].name);

}

void **PIP\_ReleaseRes**(int ResNum) //ресурс подлежащий освобождению

{

printf("PIP\_ReleaseRes %s\n",ResourceQueue[ResNum].name);

int rtask = ResourceQueue[ResNum].task;

ResourceQueue[ResNum].task = -1; //сбрасываем временный приоритет до базового

if(TaskQueue[rtask].ceiling\_priority != TaskQueue[rtask].priority)

{

printf("Reset prioritet for %s from %i to %i\n",

TaskQueue[rtask].name,

TaskQueue[rtask].ceiling\_priority,

TaskQueue[rtask].priority);

TaskQueue[rtask].ceiling\_priority = TaskQueue[rtask].priority;

Schedule(rtask);

}

Dispatch();

//переход к след задаче

printf("End of PIP\_ReleaseRes %s\n",ResourceQueue[ResNum].name);

}

**Файл task.cpp**

/\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/

/\* task.c \*/

/\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/

#include <stdio.h>

#include <string.h>

#include "sys.h"

#include "rtos\_api.h"

int **ActivateTask**(TTaskCall entry,int priority,char\* name) //функция активации задачи

{

int task,occupy;

printf("ActivateTask %s\n",name);

task=TaskHead; //указатель на голову

occupy=FreeTask; //указатель на свободную

//изменяем список свободных задач

RemoveTask(occupy, &FreeTask);

//удаление задачи из списка

TaskQueue[occupy].priority=priority;

TaskQueue[occupy].ceiling\_priority=priority;

TaskQueue[occupy].name=name;

TaskQueue[occupy].entry=entry;

TaskQueue[occupy].switch\_count = 0;

TaskQueue[occupy].task\_state = *TASK\_READY*; //пометка

TaskCount++;

printf("End of ActivateTask %s\n",name);

Schedule(occupy); //добавление задачи в очередь активных задач

if(task!=TaskHead)

{

Dispatch(); //обеспечивает переключение задач

}

return occupy;

}

void **TerminateTask**(void) //завершить задачу

{

int task;

TaskCount--; //вычитаем из количества задач

task=RunningTask; //текущая выполняющаяся задача

printf("TerminateTask %s\n",TaskQueue[task].name);

TaskQueue[task].task\_state = *TASK\_SUSPENDED*; //вносим в ожидающие

RemoveTask(task, &TaskHead); //удаляем задачу из списка

//добавляем задачу в список свободных

InsertTaskBefore(task, FreeTask, &FreeTask);

printf("End of TerminateTask %s\n",TaskQueue[task].name);

if(TaskCount == 0)

longjmp(MainContext, 1); //то все закончили

Dispatch(); //перескакиваем на другую

}

void **RemoveTask**(int task, int \*head) //удаление задачи из списка

{

if(TaskQueue[task].next == -1) //есть ли список

return;

if(\*head == task)

{

if(TaskQueue[task].next == task)

\*head = -1; //если это был послед невыполнен задача то списка больше нет

else

\*head = TaskQueue[task].next; //тогда перходим к след задачи из списка

}

TaskQueue[TaskQueue[task].prev].next = TaskQueue[task].next; //заменяем задачи на оставшиеся

TaskQueue[TaskQueue[task].next].prev = TaskQueue[task].prev;

TaskQueue[task].next = -1;

TaskQueue[task].prev = -1;

}

void **Schedule**(int task, int dont\_show) //добавление в список активных задач

{

int cur; //времен перемен содерж раб номер головы

int priority; //времен, запом приоритет для сравнения

if(TaskQueue[task].task\_state == *TASK\_SUSPENDED*) //?если равно ожидает

return;

if(!dont\_show)

printf("Schedule %s\n",TaskQueue[task].name);

if(TaskHead == -1) //если это первая задача

{

TaskHead = task; //то это и голова и след и пред

TaskQueue[task].next = task;

TaskQueue[task].prev = task;

}

else if(TaskCount > 1) //если список уже составляется

{

cur = TaskHead; //указатель на голову

priority=TaskQueue[task].ceiling\_priority; //запоминаем времен приоритет

RemoveTask(task, &TaskHead); //удаление задачи из списка

while (TaskQueue[cur].ceiling\_priority >= priority)

{

cur=TaskQueue[cur].next;

if(cur == TaskHead) //нет задачи с таким низким приоритетом - поставим ее в конец списка

break;

}

if(priority <= TaskQueue[TaskHead].ceiling\_priority && cur == TaskHead) //вставляем в конец списка

InsertTaskAfter(task, TaskQueue[TaskHead].prev);

else

InsertTaskBefore(task, cur, &TaskHead); //вставляем перед элементом

}

if(!dont\_show)

printf("End of Schedule %s\n",TaskQueue[task].name);

}

// добавить задачу перед item, head указывает на голову списка

void **InsertTaskBefore**(int task, int item, int \*head) //вставить элемент перед текущ, head, указат на голову

{

if(TaskQueue[task].next != -1) //если списка нет

return;

if(item == \*head) //если указатель на перв элемент(0)

\*head = task; //то меняем голову

TaskQueue[task].next = item;

TaskQueue[task].prev = TaskQueue[item].prev;

TaskQueue[TaskQueue[item].prev].next = task;

TaskQueue[item].prev = task;

}

void **InsertTaskAfter**(int task, int item) //вставить элемент после тек элем

{

if(TaskQueue[task].next != -1) //если списка нет

return;

TaskQueue[task].next = TaskQueue[item].next;

TaskQueue[task].prev = item;

TaskQueue[TaskQueue[item].next].prev = task;

TaskQueue[item].next = task;

}

void **Dispatch**() //переключение задач

{

if(RunningTask != -1) //если есть список, изменяется только в функ TaskSwitch

printf("Dispatch - %s\n", TaskQueue[RunningTask].name);

else printf("Dispatch\n");

if(RunningTask != -1 && TaskQueue[RunningTask].task\_state == *TASK\_RUNNING*)

TaskQueue[RunningTask].task\_state = *TASK\_READY*; //меняем пометку

int cur = TaskHead; //указатель на голову

while(TaskCount)

{

if(TaskQueue[cur].task\_state == *TASK\_READY*)

{ //переключаемся на следующую задачу

if(RunningTask == -1 || TaskQueue[RunningTask].task\_state == *TASK\_SUSPENDED*)

{//запоминать контекст не надо

TaskSwitch(cur);

}

else if(!setjmp(TaskQueue[RunningTask].context))

TaskSwitch(cur);

break;

}

cur = TaskQueue[cur].next;

if(cur == TaskHead)

{

printf("Dont have ready tasks\n");

longjmp(MainContext,1);

}

}

printf("End of Dispatch - %s\n", TaskQueue[RunningTask].name);

Schedule(cur, 1); //добавляем в список активных задач

}

void **TaskSwitch**(int nextTask) //востоновление задачи nexttext переключатель задач

{

if(nextTask == -1) //если нет массива возврат nextTask –элем с котом мы сейчас работ cur

return;

TaskQueue[nextTask].task\_state = *TASK\_RUNNING*;

RunningTask = nextTask; //текущая выполняющаяся задача

TaskQueue[nextTask].switch\_count++; //ставим номер счетчика задачи

if(TaskQueue[nextTask].switch\_count == 1)

{

printf("Dispatch - run task %s\n", TaskQueue[nextTask].name);

longjmp(InitStacks[nextTask], 1); //востоновление предыдущей задачи

}

else

longjmp(TaskQueue[nextTask].context, 1);

}

**Файл test.cpp**

/\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/

/\* test.c \*/

/\*\*\*\*\* \*\*\*\*\*\*\*\*\*\*\*\*/

#include <stdio.h>

#include <stdlib.h>

#include "rtos\_api.h"

DeclareTask(Task1,1); //декларация, объявление задач

DeclareTask(Task2,2);

DeclareTask(Task3,3);

DeclareResource(Res1,5); //объявление рисурсов

DeclareResource(Res2,5);

DeclareResource(Res3,5);

DeclareEvent(1);

DeclareEvent (2);

DeclareEvent (3);

int **main**(void) //активкация первой задачи

{

printf("Hello!\n");

// StartOS(Task1,Task1prior,"Task1");

StartOS(Task6,Task6prior,"Task6");

ShutdownOS();

return 0;

}

TASK(Task1)

{

printf("Start Task1\n");

ActivateTask(Task2,Task2prior,"Task2");

printf("Task1\n");

**TerminateTask**();

}

//Работа с событиями

int Task7\_ID;

TASK(Task7)

{

printf("Start Task7\n");

Task7\_ID = RunningTask;

ActivateTask(Task8,Task8prior,"Task8");

ClearEvent(Event\_1);

WaitEvent(Event\_1);

TEventMask evnt;

GetEvent(Task7\_ID, &evnt);

printf("Task7 GetEvent=%i\n", evnt);

printf("Task7\n");

**TerminateTask**();

}