Санкт-Петербургский политехнический университет Петра Великого Институт компьютерных наук и технологий

Высшая школа программной инженерии

К У Р С О В А Я Р А Б О Т А

«Разработка операционной системы реального времени»

По дисциплине «Архитектура программных систем»

Выполнили

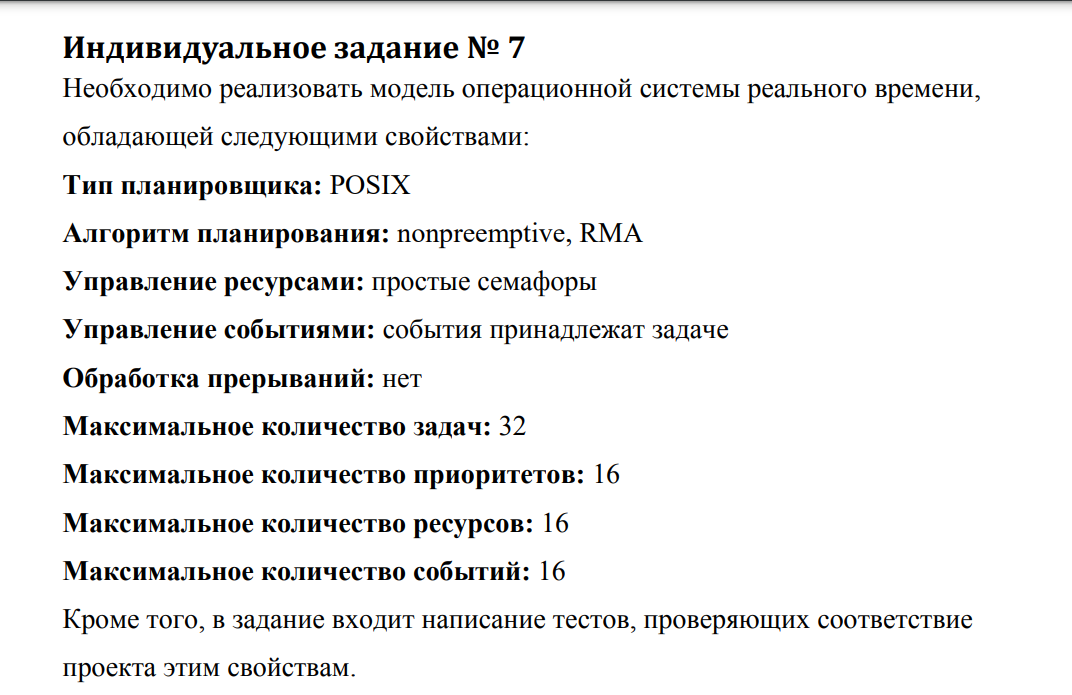
студенты гр. 3530904/00103 Бабушкин Е.А.

Ркуоводитель

Старший преподаватель Коликова Т. В.

Санкт-Петербург

## Задание



Проект выполнялся в команде с Молчановым И.Р., Легоньких А.П., Давлетшиной К.М.

## Теория

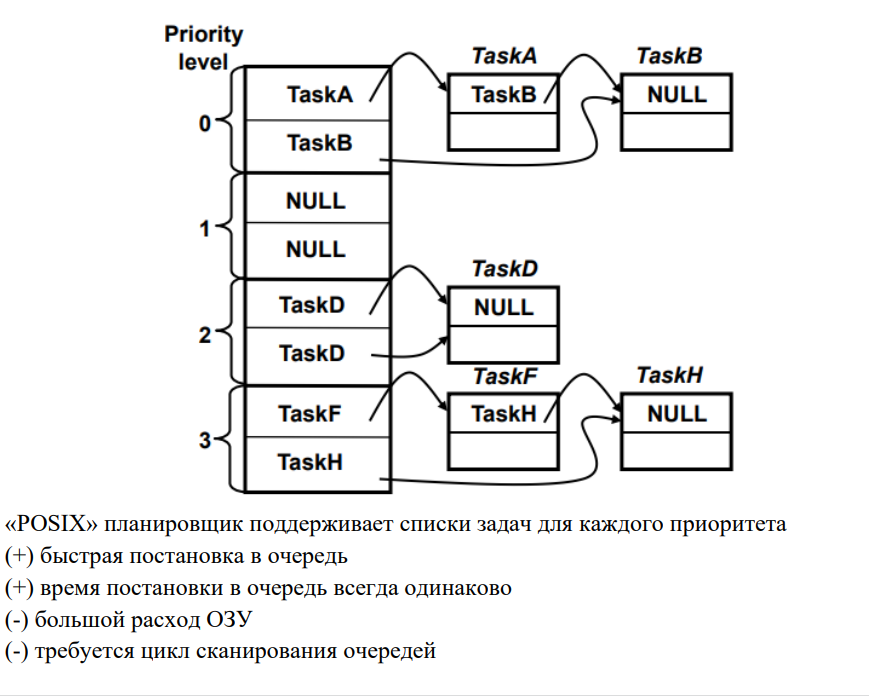
ОС общего назначения, особенно многопользовательские, ориентированы на оптимальное распределение ресурсов компьютера между пользователями и задачами (системы разделения времени), в операционных системах реального

времени (ОСРВ, RTOS), подобная задача отходит на второй план - все отступает перед главной задачей - успеть среагировать на события, происходящие на объекте.

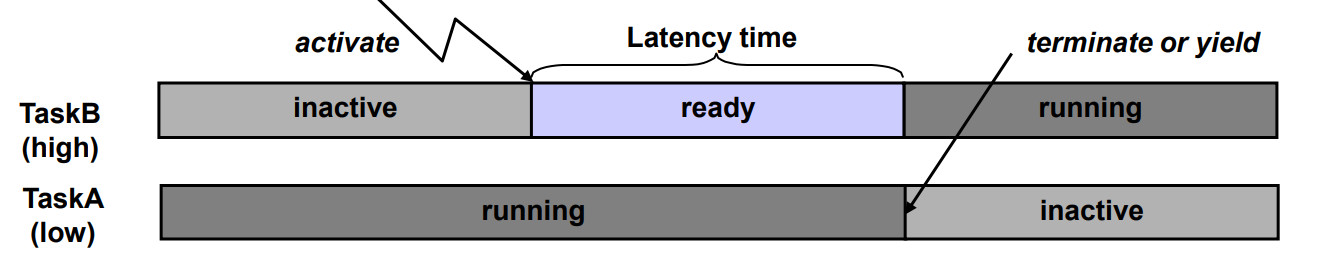
Задача (task) - единица обработки, выполняющаяся конкурентно с другими задачами. Задачи являются основным средством обработки внутренних событий. Задача имеет некоторое значение приоритета, определяющее ее относительные претензии на захват процессора. Эти претензии удовлетворяются ОС по определенному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

использоваться значение срока исполнения. Задача имеет точку входа (entry point).

Тип планировщика: POSIX



### Алгоритм планирования: nonpreemtive, RMA



Процесс планирования в RMA выглядит следующим образом:

* Определение периода исполнения каждой задачи. Период исполнения - это интервал времени между двумя последовательными запусками задачи.
* Назначение приоритетов задачам на основе их периодов исполнения. Задача с более коротким периодом получает более высокий приоритет.
* Запуск задачи с наивысшим приоритетом. Задача выполняется до тех пор, пока не завершится или пока не будет блокироваться (например, ожидать ввода/вывода).
* По мере появления новых задач, они добавляются в очередь задач, отсортированную по приоритетам. Если текущая выполняемая задача завершена или заблокирована, задача с наивысшим приоритетом из очереди становится новой выполняемой задачей.

### Изображение выглядит как диаграмма Автоматически созданное описаниеУправление ресурсами: простые семафоры

## Описание структуры проекта

## Мой вклад в проект заключался в реализации задач(tasks).

|  |  |
| --- | --- |
| Os.c | |
| int StartOS(TTask task) | Осуществляет необходимую инициализацию и запускает ОС на выполнение. |
| void ShutdownOS() | Используется для немедленного завершения работы системы. |

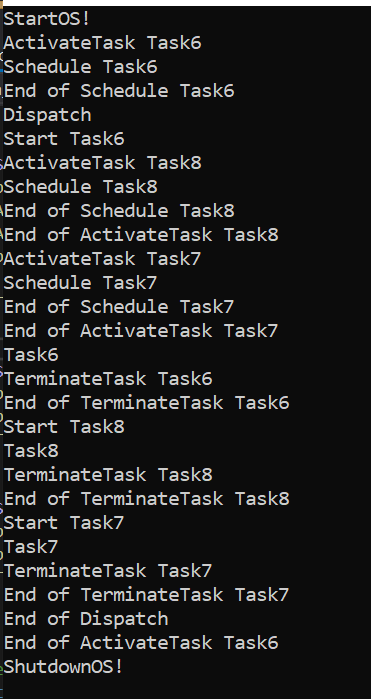
|  |  |
| --- | --- |
| Управление задачами (task.cpp) | |
| void ActivateTask(TTask task) | Переводит задачу в состояние ready |
| void TerminateTask(void) | Завершает выполнение текущей задачи, переводя её из состояния running в состояние inactive. При этом освобождаются все системные ресурсы, занятые задачей. |
| void Schedule(TaskPointer taskP, int mode) | Данная функция планирует задачу для выполнения. |
| void Dispatch(TaskPointer taskP) | Данная функция выполняет планирование и переключение контекста задачи. |

|  |  |
| --- | --- |
| event.c |  |
| TaskPointer getTaskById(TTask tTask) | Данный метод выполняет поиск задачи по ее идентификатору |
| void SetEvent(TTask tTask, const TEventMask mask) | Метод устанавливает события (mask) для указанной задачи (tTask). |
| void GetEvent(TTask tTask, TEventMask \*mask) | Данный метод получает события (mask) для указанной задачи |
| void ClearEvent(const TEventMask mask) | Метод очищает события, указанные в mask, для текущей выполняющейся задачи |
| void WaitEvent(const TEventMask mask) | Метод ждет возникновения событий, указанных в mask, для текущей выполняющейся задачи |

|  |  |
| --- | --- |
| Semaphore.c |  |
| void InitPVS(TSemaphore tSemaphore) | Данная функция инициализирует семафор |
| void P(TSemaphore tSemaphore) | Данная функция запрашивает доступ к семафору |
| void V(TSemaphore tSemaphore) | Данная функция освобождает ресурс, связанный с семафором |

Тестирование

* Тестирование task manager

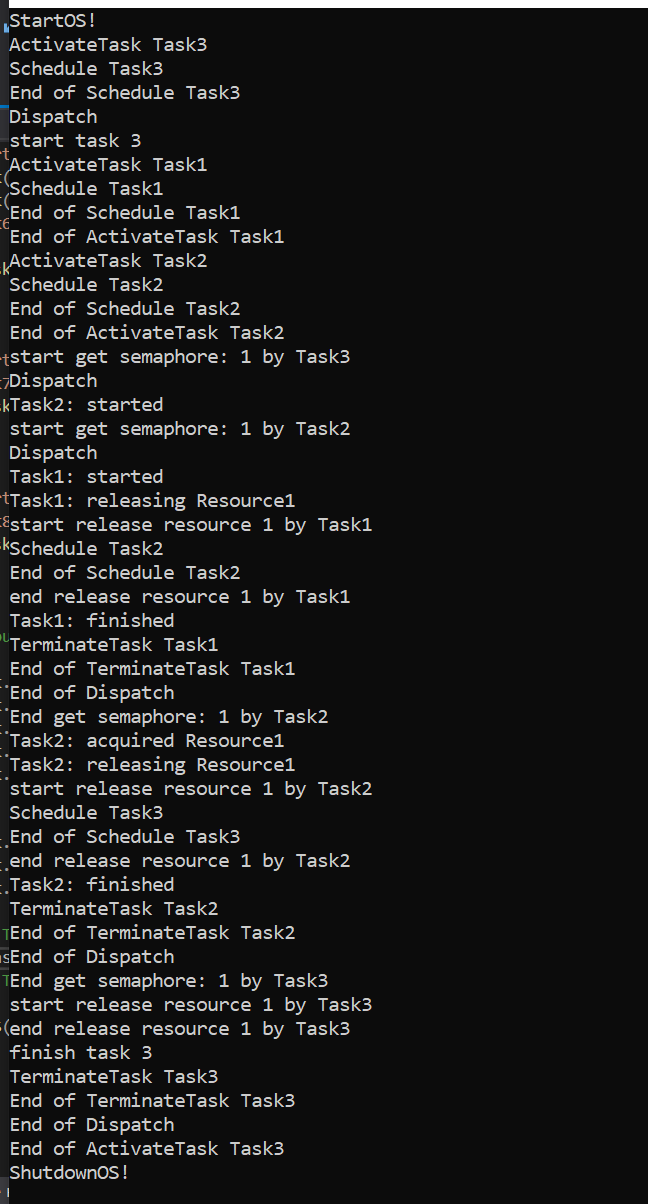


* Тестирование событий

Изображение выглядит как текст, снимок экрана, Шрифт

Автоматически созданное описание

* Тестирование ресурсов



## Вывод

По результатам работы над курсовом проектом была написана система управления ОС реального времени с соблюдением всех необходимых требований. Применены на практике теоретические знания, полученные во время лекций и самостоятельного изучения предоставленного материала.

Были получены навыки переключения контекстов и реализации алгоритмов планирования (nonpreemptive, RMA), управления ресурсами (простые семафоры) и управления событиями. Составлен отчёт по проделанной работе, в котором

описывается структура проекта, а также предоставлен код составных частей проекта.

## Приложение 1.

#### defs.h

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

/\* defs.h \*/

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

#define MAX\_TASK 32

#define MAX\_PRIORITY 16

#define MAX\_SEM 16

#define MAX\_EVENTS 16

#### sys.h

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

/\* sys.h /

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

#ifndef SYS\_H

#define SYS\_H

#include <stdbool.h>

#include "defs.h"

#define INSERT\_TO\_TAIL 1

#define INSERT\_TO\_HEAD 0

#define READY\_STATUS\_TASK 0

#define WAIT\_SEM\_STATUS\_TASK 1

#define WAIT\_EVENT\_STATUS\_TASK 2

#define INACTIVE\_STATUS\_TASK 3

#define NOT\_INIT\_SEM 2

typedef struct Type\_Task {

int id;

int next;

int status;

void (\*entry)(void);

const char \*name;

bool waitEvents[MAX\_EVENTS];

bool setEvents[MAX\_EVENTS];

} Type\_Task;

typedef struct {

Type\_Task taskQueue[MAX\_TASK];

int head;

int free;

} BeginEndTTaskQueue;

typedef struct {

int priority;

int index;

} TaskPointer;

typedef struct {

TaskPointer blockedTask;

int status;

int headStack;

TaskPointer taskWaitStack[MAX\_TASK];

} Type\_Semaphore;

extern BeginEndTTaskQueue TaskQueue[MAX\_PRIORITY];

extern Type\_Semaphore SemaphoreQueue[MAX\_SEM];

extern TaskPointer RunningTask;

extern TaskPointer nullptrTask;

void Schedule(TaskPointer task, int mode);

void Dispatch(TaskPointer task);

#endif

#### global.c

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

/\* global.c \*/

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

#include "sys.h"

BeginEndTTaskQueue TaskQueue[MAX\_PRIORITY];

Type\_Semaphore SemaphoreQueue[MAX\_SEM];

TaskPointer RunningTask;

TaskPointer nullptrTask = {.priority = -1, .index = -1};

#### rtos\_api.h

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

/\* rtos\_api.h \*/

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

#include <stdbool.h>

#include "defs.h"

#define BLOCKED\_SEM 1

#define READY\_SEM 0

#define DeclareTask(TaskName, UniqueId)\

TASK(TaskName); \

const char TaskName##Name[]=#TaskName; \

TTask TaskName##TTask={.id=UniqueId, .entry=TaskName, .name=TaskName##Name}

#define TASK(TaskName) void TaskName(void)

#define DeclareSemaphore(SemName, SemId, StartStatus) \

TSemaphore SemName = {.id = SemId, .startStatus = StartStatus}

#define DeclareEvent(EventId) \

enum { Event##EventId=EventId}

typedef bool TEventMask[MAX\_EVENTS];

typedef void TTaskCall(void);

typedef struct {

int plannedTimeWork;

const char \*const name;

TTaskCall \*const entry;

const int id;

} TTask;

typedef struct {

int id;

int startStatus;

} TSemaphore;

//task api

void ActivateTask(TTask task);

void TerminateTask(void);

//os api

int StartOS(TTask task);

void ShutdownOS();

//semaphore api

void InitPVS(TSemaphore tSemaphore);

void P(TSemaphore tSemaphore);

void V(TSemaphore tSemaphore);

//events api

void SetEvent(TTask tTask, const TEventMask mask);

void GetEvent(TTask tTask, TEventMask \*mask);

void ClearEvent(const TEventMask mask);

void WaitEvent(const TEventMask mask);

#### event.c

#include <stdio.h>

#include "sys.h"

#include "rtos\_api.h"

#include "addings.h"

TaskPointer getTaskById(TTask tTask) {

int prior = getPriorityByTime(tTask.plannedTimeWork);

TaskPointer res = {.priority = prior, .index = 0};

while (res.index < MAX\_TASK && getTask(res)->id != tTask.id) {

res.index++;

}

if (res.index == MAX\_TASK || getTask(res)->status == INACTIVE\_STATUS\_TASK) {

return nullptrTask;

}

return res;

}

void SetEvent(TTask tTask, const TEventMask mask) {

TaskPointer taskP = getTaskById(tTask);

if (equalTaskPointers(taskP, nullptrTask)) {

printf("Error: task with id %d not found or this task have INACTIVE\_STATUS\n", tTask.id);

return;

}

Type\_Task \*task = getTask(taskP);

printf("start set mask event for task %s\n", task->name);

bool flag = false;

for (int i = 0; i < MAX\_EVENTS; ++i) {

if (task->status == WAIT\_EVENT\_STATUS\_TASK && task->waitEvents[i] && mask[i]) {

task->status = READY\_STATUS\_TASK;

flag = true;

}

task->setEvents[i] |= mask[i];

}

if (flag) {

Schedule(taskP, INSERT\_TO\_HEAD);

}

printf("end set mask event for task %s\n", task->name);

}

void GetEvent(TTask tTask, TEventMask \*mask) {

TaskPointer taskP = getTaskById(tTask);

if (equalTaskPointers(taskP, nullptrTask)) {

printf("GetEvent: Error: task with id %d not found or this task have INACTIVE\_STATUS\n", tTask.id);

return;

}

Type\_Task \*task = getTask(taskP);

for (int i = 0; i < MAX\_EVENTS; ++i) {

(\*mask)[i] = task->setEvents[i];

}

}

void ClearEvent(const TEventMask mask) {

Type\_Task \*task = getTask(RunningTask);

for (int i = 0; i < MAX\_EVENTS; ++i) {

if (mask[i]) {

task->setEvents[i] = false;

}

}

}

void WaitEvent(const TEventMask mask) {

bool needWait = true;

Type\_Task \*task = getTask(RunningTask);

for (int i = 0; i < MAX\_EVENTS; ++i) {

if (mask[i] && task->setEvents[i]) {

needWait = false;

break;

}

task->waitEvents[i] = mask[i];

}

if (needWait) {

printf("start wait event by %s\n", task->name);

TaskPointer waitTask = RunningTask;

task->status = WAIT\_EVENT\_STATUS\_TASK;

TaskPointer cur = {.priority = waitTask.priority, .index = TaskQueue[waitTask.priority].head};

TaskPointer prev = {.priority = waitTask.priority, .index = -1};

while (cur.index != waitTask.index) {

prev.index = cur.index;

cur.index = getTask(cur)->next;

}

if (prev.index == -1) {

TaskQueue[waitTask.priority].head = getTask(waitTask)->next;

} else {

getTask(prev)->next = getTask(waitTask)->next;

}

RunningTask = getNextTask();

Dispatch(waitTask);

printf("free wait event by %s\n", task->name);

}

ClearEvent(mask);

}

#### os.c

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

/\* os.c \*/

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

#include <stdio.h>

#include "sys.h"

#include "rtos\_api.h"

int StartOS(TTask task) {

RunningTask = nullptrTask;

printf("StartOS!\n");

for (int i = 0; i < MAX\_PRIORITY; i++) {

TaskQueue[i].head = -1;

TaskQueue[i].free = 0;

for (int j = 0; j < MAX\_TASK; ++j) {

TaskQueue[i].taskQueue[j].next = j + 1;

TaskQueue[i].taskQueue[j].status = INACTIVE\_STATUS\_TASK;

}

TaskQueue[i].taskQueue[MAX\_TASK - 1].next = -1;

}

for (int i = 0; i < MAX\_SEM; i++) {

SemaphoreQueue[i].headStack = -1;

SemaphoreQueue[i].status = NOT\_INIT\_SEM;

SemaphoreQueue[i].blockedTask = nullptrTask;

for (int j = 0; j < MAX\_TASK; ++j) {

SemaphoreQueue[i].taskWaitStack[j] = nullptrTask;

}

}

ActivateTask(task);

return 0;

}

void ShutdownOS() {

printf("ShutdownOS!\n");

}

#### semaphore.c

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

/\* resource.c \*/

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

#include "rtos\_api.h"

#include "addings.h"

#include <stdio.h>

void InitPVS(TSemaphore tSemaphore) {

if (tSemaphore.id > MAX\_SEM || tSemaphore.id < 0) {

printf("Error: Requested semaphore with tSemaphore %d cannot initialize.\n", tSemaphore.id);

return;

}

if (SemaphoreQueue[tSemaphore.id].status != NOT\_INIT\_SEM) {

printf("Error: Requested semaphore with tSemaphore %d cannot initialize. Semaphore already initialize\n",

tSemaphore.id);

return;

}

SemaphoreQueue[tSemaphore.id].status = tSemaphore.startStatus;

}

void P(TSemaphore tSemaphore) {

if (tSemaphore.id >= MAX\_SEM || tSemaphore.id < 0) {

printf("Error: Requested semaphore with tSemaphore %d not found.\n", tSemaphore.id);

return;

}

if (SemaphoreQueue[tSemaphore.id].status == NOT\_INIT\_SEM) {

printf("Error: Semaphore with tSemaphore %d not initialized.\n", tSemaphore.id);

return;

}

if (equalTaskPointers(RunningTask, SemaphoreQueue[tSemaphore.id].blockedTask)) {

printf("Error: Semaphore %d already blocked task %s.\n", tSemaphore.id, getTask(RunningTask)->name);

return;

}

if (SemaphoreQueue[tSemaphore.id].headStack + 1 == MAX\_TASK) {

printf("Error: Semaphore %d wait list of tasks full.\n", tSemaphore.id);

return;

}

printf("start get semaphore: %d by %s\n", tSemaphore.id, getTask(RunningTask)->name);

if (SemaphoreQueue[tSemaphore.id].status == READY\_SEM) {

SemaphoreQueue[tSemaphore.id].status = BLOCKED\_SEM;

SemaphoreQueue[tSemaphore.id].blockedTask = RunningTask;

printf("End get semaphore: %d by %s\n", tSemaphore.id, getTask(RunningTask)->name);

} else {

SemaphoreQueue[tSemaphore.id].headStack++;

TaskPointer waitTask = RunningTask;

SemaphoreQueue[tSemaphore.id].taskWaitStack[SemaphoreQueue[tSemaphore.id].headStack] = waitTask;

TaskPointer cur = {.priority = waitTask.priority, .index = TaskQueue[waitTask.priority].head};

TaskPointer prev = {.priority = waitTask.priority, .index = -1};

while (cur.index != waitTask.index) {

prev.index = cur.index;

cur.index = getTask(cur)->next;

}

if (prev.index == -1) {

TaskQueue[waitTask.priority].head = getTask(waitTask)->next;

} else {

getTask(prev)->next = getTask(waitTask)->next;

}

getTask(waitTask)->status = WAIT\_SEM\_STATUS\_TASK;

RunningTask = getNextTask();

Dispatch(waitTask);

printf("End get semaphore: %d by %s\n", tSemaphore.id, getTask(waitTask)->name);

}

}

void V(TSemaphore tSemaphore) {

if (tSemaphore.id >= MAX\_SEM || tSemaphore.id < 0) {

printf("Error: Requested semaphore with tSemaphore %d not found.\n", tSemaphore.id);

return;

}

if (SemaphoreQueue[tSemaphore.id].status == NOT\_INIT\_SEM) {

printf("Error: Semaphore %d not initialized.\n", tSemaphore.id);

return;

}

if (SemaphoreQueue[tSemaphore.id].status == READY\_SEM) {

printf("Error: Semaphore %d not blocked.\n", tSemaphore.id);

return;

}

printf("start release resource %d by %s\n", tSemaphore.id, getTask(RunningTask)->name);

if (SemaphoreQueue[tSemaphore.id].headStack == -1) {

SemaphoreQueue[tSemaphore.id].status = READY\_SEM;

SemaphoreQueue[tSemaphore.id].blockedTask = nullptrTask;

} else {

int indWait = SemaphoreQueue[tSemaphore.id].headStack--;

SemaphoreQueue[tSemaphore.id].blockedTask = SemaphoreQueue[tSemaphore.id].taskWaitStack[indWait];

SemaphoreQueue[tSemaphore.id].taskWaitStack[indWait] = nullptrTask;

TaskPointer nonWaitTask = SemaphoreQueue[tSemaphore.id].blockedTask;

getTask(nonWaitTask)->status = READY\_STATUS\_TASK;

Schedule(nonWaitTask, INSERT\_TO\_HEAD);

}

printf("end release resource %d by %s\n", tSemaphore.id, getTask(RunningTask)->name);

}

#### task.c

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

/\* task.c \*/

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

#include <stdio.h>

#include <assert.h>

#include "sys.h"

#include "rtos\_api.h"

#include "addings.h"

void ActivateTask(TTask task) {

TaskPointer occupy;

int priorityByTime = getPriorityByTime(task.plannedTimeWork);

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

occupy.priority = priorityByTime;

occupy.index = TaskQueue[priorityByTime].free;

if (occupy.index == -1) {

printf("Error: too much tasks\n");

return;

}

Type\_Task \*occupyTask = getTask(occupy);

TaskQueue[priorityByTime].free = occupyTask->next;

occupyTask->id = task.id;

occupyTask->status = READY\_STATUS\_TASK;

occupyTask->name = task.name;

occupyTask->entry = task.entry;

for (int i = 0; i < MAX\_EVENTS; ++i) {

occupyTask->setEvents[i] = false;

occupyTask->waitEvents[i] = false;

}

Schedule(occupy, INSERT\_TO\_TAIL);

if (RunningTask.priority == -1) {

RunningTask = occupy;

Dispatch(nullptrTask);

}

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

}

void TerminateTask(void) {

for (int i = 0; i < MAX\_SEM; ++i) {

if (equalTaskPointers(RunningTask, SemaphoreQueue[i].blockedTask)) {

printf("Error: task %s blocked semaphore %d, it cannot terminate itself.\n",

getTask(RunningTask)->name, i);

return;

}

}

Type\_Task \*task = getTask(RunningTask);

task->status = INACTIVE\_STATUS\_TASK;

printf("TerminateTask %s\n", task->name);

TaskPointer prev = {.priority = RunningTask.priority, .index = -1};

TaskPointer cur = {.priority = RunningTask.priority, .index = TaskQueue[RunningTask.priority].head};

while (cur.index != -1 && cur.index != RunningTask.index) {

prev.index = cur.index;

cur.index = getTask(cur)->next;

}

if (prev.index == -1) {

TaskQueue[cur.priority].head = task->next;

} else {

getTask(prev)->next = task->next;

}

task->next = TaskQueue[RunningTask.priority].free;

TaskQueue[RunningTask.priority].free = RunningTask.index;

RunningTask = getNextTask();

printf("End of TerminateTask %s\n", task->name);

}

void Schedule(TaskPointer taskP, int mode) {

printf("Schedule %s\n", getTask(taskP)->name);

int priority = taskP.priority;

if (mode == INSERT\_TO\_TAIL) {

getTask(taskP)->next = -1;

TaskPointer curr = {.priority = priority, .index = TaskQueue[taskP.priority].head};

TaskPointer prev = {.priority = priority, .index = nullptrTask.index};

while (curr.index != -1) {

prev.index = curr.index;

curr.index = getTask(curr)->next;

}

if (prev.index == -1) {

TaskQueue[priority].head = taskP.index;

} else {

getTask(prev)->next = taskP.index;

}

} else {

getTask(taskP)->next = TaskQueue[taskP.priority].head;

TaskQueue[priority].head = taskP.index;

}

printf("End of Schedule %s\n", getTask(taskP)->name);

}

void Dispatch(TaskPointer taskP) {

printf("Dispatch\n");

do {

getTask(RunningTask)->entry();

} while (!equalTaskPointers(taskP, RunningTask));

printf("End of Dispatch\n");

#### } test.c

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

/\* test.c \*/

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

#include <stdio.h>

#include "rtos\_api.h"

DeclareSemaphore(Sem\_1, 1, BLOCKED\_SEM);

DeclareTask(Task1, 1);

DeclareTask(Task2, 2);

DeclareTask(Task3, 3);

DeclareTask(Task4, 4);

DeclareTask(Task5, 5);

DeclareTask(Task6, 6);

DeclareTask(Task7, 7);

DeclareTask(Task8, 8);

DeclareEvent(1);

TASK(Task1) {

printf("Task1: started\n");

//printf("Task1: acquired Resource1\n");

// Simulate work with the resource

printf("Task1: releasing Resource1\n");

V(Sem\_1);

printf("Task1: finished\n");

TerminateTask();

}

TASK(Task2) {

printf("Task2: started\n");

P(Sem\_1);

printf("Task2: acquired Resource1\n");

// Simulate work with the resource

printf("Task2: releasing Resource1\n");

V(Sem\_1);

printf("Task2: finished\n");

TerminateTask();

}

TASK(Task3) {

InitPVS(Sem\_1);

printf("start task 3\n");

ActivateTask(Task1TTask);

ActivateTask(Task2TTask);

P(Sem\_1);

V(Sem\_1);

printf("finish task 3\n");

TerminateTask();

}

TASK(Task4) {

printf("start %s\n", Task4Name);

ActivateTask(Task5TTask);

TEventMask mask;

GetEvent(Task4TTask, &mask);

mask[Event1] = true;

WaitEvent(mask);

printf("free %s\n", Task4Name);

TerminateTask();

}

TASK(Task5) {

printf("start %s\n", Task5Name);

TEventMask mask;

GetEvent(Task4TTask, &mask);

mask[Event1] = true;

SetEvent(Task4TTask, mask);

printf("free %s\n", Task5Name);

TerminateTask();

}

TASK(Task6) {

printf("Start Task6\n");

ActivateTask(Task8TTask);

ActivateTask(Task7TTask);

printf("Task6\n");

TerminateTask();

}

TASK(Task7) {

printf("Start Task7\n");

printf("Task7\n");

TerminateTask();

}

TASK(Task8) {

printf("Start Task8\n");

printf("Task8\n");

TerminateTask();

}

//event.c resource.c

int main() {

Task1TTask.plannedTimeWork = 300;

Task2TTask.plannedTimeWork = 100;

Task3TTask.plannedTimeWork = 300;

Task4TTask.plannedTimeWork = 80;

Task5TTask.plannedTimeWork = 200;

Task6TTask.plannedTimeWork = 300;

Task7TTask.plannedTimeWork = 200;

Task8TTask.plannedTimeWork = 100;

StartOS(Task6TTask); // test task manager

//StartOS(Task3TTask); // tests semaphore

//StartOS(Task4TTask); // tests events

ShutdownOS();

return 0;

}

case 2:

GetResource(*Task6prior*, *Res4*, "Task6"); StartOS(Task6, *Task6prior*, "Task6"); break;

case 3:

StartOS(Task4, *Task4prior*, "Task4"); TerminateTask();

default:

break;

}

ShutdownOS();

return 0;

}

**TASK**(Task1) {

printf("Task1\n");

ActivateTask(Task2, *Task2prior*, "Task2"); TerminateTask();

}

**TASK**(Task2) {

ActivateTask(Task3, *Task3prior*, "Task3"); printf("Task2\n");

TerminateTask();

}

**TASK**(Task3) {

printf("Task3\n"); TerminateTask();

}

**TASK**(Task4) {

printf("Task4\n"); TerminateTask();

}

**TASK**(Task6) {

printf("Task6\n"); GetResource(*Task7prior*, *Res4*, "Res4"); ActivateTask(Task7, *Task7prior*, "Task7"); TerminateTask();

}

**TASK**(Task7) {

printf("Task7\n"); TerminateTask();

}