Санкт-Петербургский политехнический университет Петра Великого
Институт компьютерных наук и технологий
Высшая школа программной инженерии

Курсовая работа

Модель операционной системы реального времени

по дисциплине

«Архитектура программных систем»

Выполнил: Савчук А.А.
Группа: гр. 3530904/00104

Проверил: Коликова Т. В.

Санкт-Петербург
2023

Содержание

1 Постановка задачи

Индвидуальное задание №4
Необходимо реализовать модель операционной системы реального времени обладающей следующими свойствами:

Тип планировщика: плоский

Алгоритм планирования: nonpreemptive, RMA

Управление ресурсами: PIP

Управление событиями: системные события

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.

2 Пояснения к варианту

2.1 Плоский планировщик

Плоский планировщик (*flat scheduler*) - это тип планировщика операционной системы, который использует простейший алгоритм планирования, такой как "первым пришел - первым обслужен"(*First Come, First Served, FCFS*). При таком подходе процессы выполняются в том порядке, в котором они поступили в очередь на выполнение, без учета их приоритетов или времени выполнения.

2.2 Невытесняющий алгоритм планирования

В данном варианте используется *nonpreemptive* или невытесняющий алгоритм планирования. Этот алгоритм основан на том, что активному потоку позволяется выполняться, пока он сам, по собственной инициативе, не отдаст управление операционной системе для того, чтобы та выбрала из очереди другой готовый к выполнению поток.

2.3 Протокол наследования приоритетов

Протокол наследования приоритетов (*PIP*) используется для совместного использования критических ресурсов между различными задачами без возникновения неограниченных инверсий приоритета [1].

Когда несколько задач ждут доступа к одному и тому же критическому ресурсу, которая в данный момент имеет доступ к этому ресурсу, присваивается самый высокий приоритет среди всех задач, ждущих этот ресурс. Если задача с более низким приоритетом, но имеющая доступ к критическому ресурсу, находится в ожидании, то ее приоритет увеличивается до приоритета задачи, которая в данный момент имеет доступ к этому ресурсу. Это позволяет нижестоящей задаче использовать критический ресурс без прерывания выполнения и избежать неограниченных инверсий приоритета.

[1] Инверсия приоритета - это ситуация, которая возникает при параллельном выполнении нескольких задач с разными приоритетами, когда задача с более высоким приоритетом блокируется задачей с более низким приоритетом из-за необходимости доступа к общему ресурсу

3 Описание структуры проекта

3.1 Управление ОС

StartOS(entry, priority, name) – Запуск ОС, инициализация основных элементов системы, активация начальной задачи.

ShutdownOS() – Завершение работы ОС.

3.2 Управление задачами

ActivateTask(entry, priority, name) – Инициализация задачи в системе.

TermitateTask() – Завершение задачи.

Schedule(task, mask) – Постановка задачи в очередь.

Dispatch() – Диспетчеризация задач, постановка на выполнение.

3.3 Управление ресурсами

InitRes(name) – Инициализация ресурса в системе.

PIP_GetRes(res) – Захват ресурса выполняющейся задачей.

PIP_ReleaseRes(res) – Освобождение ресурса.

3.4 Управление событиями

SetSysEvent(mask) – Установка системного события.

GetSysEvent(mask) – Возвращает текущее состояние системной маски установленных событий.

WaitSysEvent(mask) – Задача переводится в состояние ожидания.

4 Тестирование

4.1 Невытесняющий алгоритм планирования

Цель теста: Проверка работы невытесняющего алгоритма планирования.

Ожидаемый результат : Задача низкого приоритета завершает своё выполнение без прерываний, потом срабатывает задача высокого приоритета, после неё — задача среднего приоритета;

Результат:

```
StartOS!
[ActivateTask] Task1
End of Activate Task1
[Schedule] Task1
End of Schedule Task1
Task1 started

This is Task1!

[ActivateTask] Task2
End of Activate Task2
[Schedule] Task2
End of Schedule Task2
Task1 is running
[ActivateTask] Task3
End of Activate Task3
[Schedule] Task3
End of Schedule Task3
Task1 is running
[Terminate Task] Task1
End of Terminate Task Task1
Task3 started

This is Task3!

[Terminate Task] Task3
End of Terminate Task Task3
Task2 started

This is Task2!

[Terminate Task] Task2
End of Terminate Task Task2
ShutdownOS!
```

4.2 Системные события

Цель теста: Проверка работы системных событий.

Ожидаемый результат: Событие перевелось в состояние ожидания по системному событию пока задача, которая поставит это событие не закончит свою работу.

Результат:

```
StartOS!
[ActivateTask] Task6
End of Activate Task6
[Schedule] Task6
End of Schedule Task6
Task6 started

This is Task6!

[ActivateTask] Task7
End of Activate Task7
[Schedule] Task7
End of Schedule Task7
Task6 is running
WaitSysEvent 1
Task7 started

This is Task7!

SetSysEvent 1
Task "Task6" is ready
Task7 is running
End of SetSysEvent 1
Task7
[Terminate Task] Task7
End of Terminate Task Task7
Task6 started
Task6 is running
End of WaitSysEvent 1
GetEvent=0
Task6
[Terminate Task] Task6
End of Terminate Task Task6
ShutdownOS!
```

4.3 PIP-алгоритм

Цель теста: Проверка работы PIP-алгоритма

Ожидаемый результат: Задача низкого приоритета должна успешно захватить и освободить ресурс, предоставив задаче высокого приоритета возможность успешно захватить его снова

Результат:

```
StartOS!
```

```
[ActivateTask] Task4
End of Activate Task4
[Schedule] Task4
End of Schedule Task4
Task4 started

This is Task4!

GetResource Res1
Resource Res1 captured by Task4
[ActivateTask] Task5
End of Activate Task5
[Schedule] Task5
End of Schedule Task5
Task4 is running
Release resource Res1
Resource Res1 released by Task4
Task4 is running
[Terminate Task] Task4
End of Terminate Task Task4
Task5 started

This is Task5!

GetResource Res1
Resource Res1 captured by Task5
Release resource Res1
Resource Res1 released by Task5
Task5 is running
[Terminate Task] Task5
End of Terminate Task Task5
ShutdownOS!
```

4.4 Полный функционал

Цель теста: Проверка работы всех функций вместе.

Результат:

```
StartOS!
[ActivateTask] Task8
End of Activate Task8
[Schedule] Task8
End of Schedule Task8
Task8 started

Start Task8

[ActivateTask] Task9
End of Activate Task9
```

```
[Schedule] Task9
End of Schedule Task9
Task8 is running
WaitSysEvent 1
Task9 started

Start Task9

GetResource Res1
Resource Res1 captured by Task9
[ActivateTask] Task10
End of Activate Task10
[Schedule] Task10
End of Schedule Task10
Task9 is running
SetSysEvent 1
Task "Task8" is ready
Task9 is running
End of SetSysEvent 1
WaitSysEvent 2
Task10 started

Start Task10

GetResource Res1
Resource Res1 captured by Task10
SetSysEvent 2
Task "Task9" is ready
Task10 is running
End of SetSysEvent 2
Release resource Res1
Resource Res1 released by Task10
Task10 is running

Task 10 finished

[Terminate Task] Task10
End of Terminate Task Task10
Task9 started
GetResource Res1
Resource Res1 captured by Task9
Task9 is running
End of WaitSysEvent 2
Release resource Res1
Resource Res1 released by Task9
Task9 is running

Task 9 finished

[Terminate Task] Task9
```

```
End of Terminate Task Task9
Task8 started
Task8 is running
End of WaitSysEvent 1

Task 8 finished

[Terminate Task] Task8
End of Terminate Task Task8
ShutdownOS!
```

Код Программы

global.cpp

```
#include "sys.h"

TTask TaskQueue[MAX_TASK];
TResource ResourceQueue[MAX_RES];
TResource EventQueue[MAX_EVENTS];

int RunningTask;
int FreeTask = 0;
int HeadTask = 0;
int FreeResource = 0;
TEventMask WorkingEvents;
```

defs.h

```

#ifndef APS_OS_REALTIME_DEFS_H
#define APS_OS_REALTIME_DEFS_H

#define MAX_TASK 32
#define MAX_RES 16
#define MAX_EVENTS 16

#endif //APS_OS_REALTIME_DEFS_H
```

rtos_api.h

```

#ifndef APS_OS_REALTIME_RTOS_API_H
#define APS_OS_REALTIME_RTOS_API_H

#include "sys.h"

#define DeclareTask(TaskID, priority)\
TASK(TaskID); \
enum {TaskID##prior=priority}

#define TASK(TaskID) void TaskID(void)

#define DeclareEvent(task, EventID)\
TEvent Event##EventID = TEvent(task, 1 << (EventID - 1));

typedef void TTaskCall(void);

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

void TerminateTask(void);

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

void ShutdownOS();
```

```

int InitRes(const char *name);

void PIP_GetRes(int res);

void PIP_ReleaseRes(int res);

// наш вариант

#define DeclareSysEvent(ID) \
const int Event_##ID = (ID)*(ID);

void SetSysEvent(TEventMask mask);

void GetSysEvent(TEventMask* event);

void WaitSysEvent(TEventMask mask);

#endif
```

sys.h

```

#ifndef APS_OS_REALTIME_SYS_H
#define APS_OS_REALTIME_SYS_H

#include "defs.h"
#include <csetjmp>
#include "string.h"

#define INSERT_TO_TAIL 1
#define INSERT_TO_HEAD 0

#define _NULL -1

enum TTaskState {
    TASK_RUNNING,
    TASK_READY,
    TASK_SUSPENDED,
    TASK_WAITING,
    TASK_DONE
};

typedef int TEventMask;
extern TEventMask WorkingEvents;

typedef struct Type_Task {
    int ref;
    int priority;
    TTaskState state;
    TEventMask waiting_events;
```

```
30
31      int res;
32      unsigned int switchNumber = 0;
33
34      void (*entry)(void);
35
36      jmp_buf context;
37      char *name;
38  } TTask;
39
40  typedef struct Type_resource {
41      int task;
42      int priority;
43      const char *name;
44  } TResource;
45
46  extern int HeadTask;
47
48  extern TTask TaskQueue[MAX_TASK];
49  typedef struct Event {
50      Event(char *name, TEventMask mask) {
51          this->task = name;
52          this->mask = mask;
53      }
54
55      char *task;
56      TEventMask mask;
57  } TEvent;
58
59  extern TResource ResourceQueue[MAX_RES];
60  extern TResource EventQueue[MAX_EVENTS];
61  extern int RunningTask;
62  extern int FreeTask;
63  extern int FreeResource;
64
65  void Schedule(int task, int mode = INSERT_TO_TAIL);
66
67  void Dispatch();
68
69
70  #endif
```

events.cpp

```
1  #include "sys.h"
2  #include "rtos_api.h"
3  #include <stdio.h>
4  #include "string.h"
5  #include <stdexcept>
6
7  void SetSysEvent(TEventMask mask) {
```

```
8       printf("SetSysEvent %i\n", mask);
9       WorkingEvents |= mask;
10      for (int i = 0; i < MAX_TASK; i++) {
11          if (TaskQueue[i].state == TASK_WAITING &&
12              (WorkingEvents & TaskQueue[i].waiting_events)) {
13              TaskQueue[i].waiting_events &= !mask;
14              TaskQueue[i].state = TASK_READY;
15              printf("Task \"%s\" is ready\n", TaskQueue[i].name);
16          }
17      }
18      WorkingEvents &= !mask;
19      Dispatch();
20      printf("End of SetSysEvent %i\n", mask);
21  }
22
23  void GetSysEvent(TEventMask *mask) {
24      *mask = WorkingEvents;
25  }
26
27  void WaitSysEvent(TEventMask mask) {
28      printf("WaitSysEvent %i\n", mask);
29      TaskQueue[RunningTask].waiting_events = mask;
30      if ((WorkingEvents & mask) == 0) {
31          TaskQueue[RunningTask].state = TASK_WAITING;
32
33          int task = RunningTask;
34          RunningTask = _NULL;
35          setjmp(TaskQueue[task].context);
36
37          Dispatch();
38      }
39      printf("End of WaitSysEvent %i\n", mask);
40  }
41
```

resource.cpp

```
1   #include "sys.h"
2   #include "rtos_api.h"
3   #include <stdio.h>
4   #include <stdexcept>
5
6   // Инициализация ресурса в системе.
7   int InitRes(const char *name) {
8
9       int resource = FreeResource;
10      FreeResource++;
11      ResourceQueue[resource].name = name;
12      ResourceQueue[resource].priority = 0;
13      ResourceQueue[resource].task = _NULL;
14      return resource;
15  }
```

```

void PIP_GetRes(int res) {
    printf("GetResource %s\n", ResourceQueue[res].name);

    if (TaskQueue[RunningTask].priority < ResourceQueue[res].priority) {
        printf("%s is trying to capture the res of higher priority",
               TaskQueue[RunningTask].name);
        throw std::exception();
    }
    if (ResourceQueue[res].task != _NULL) {
        TaskQueue[ResourceQueue[res].task].state = TASK_WAITING;
    }

    TaskQueue[RunningTask].res = res;
    ResourceQueue[res].priority = TaskQueue[RunningTask].priority;
    ResourceQueue[res].task = RunningTask;

    printf("Resource %s captured by %s\n", ResourceQueue[res].name,
           TaskQueue[RunningTask].name);
}

void PIP_ReleaseRes(int res) {
    printf("Release resource %s\n", ResourceQueue[res].name);

    if (RunningTask != ResourceQueue[res].task) {
        printf("%s is trying to release the resource %s, which "
               "is not captured by it", TaskQueue[RunningTask].name,
               ResourceQueue[res].name);
        throw std::exception();
    }

    TaskQueue[ResourceQueue[res].task].res = _NULL;
    ResourceQueue[res].task = _NULL;
    ResourceQueue[res].priority = 0;

    printf("Resource %s released by %s\n", ResourceQueue[res].name,
           TaskQueue[RunningTask].name);
    Dispatch();
}
```

task.cpp

```
#include <stdio.h>
#include "sys.h"
#include "rtos_api.h"

void ActivateTask(TTaskCall entry, int priority, char *name) {
    printf("[ActivateTask] %s\n", name);

    int occupy = FreeTask;
```

```
9       FreeTask = TaskQueue[occupy].ref;
10
11      // Инициализируем поля задачи.
12      TaskQueue[occupy].priority = priority;
13      TaskQueue[occupy].name = name;
14      TaskQueue[occupy].waiting_events = 0;
15      TaskQueue[occupy].entry = entry;
16      TaskQueue[occupy].switchNumber = 0;
17      TaskQueue[occupy].res = _NULL;
18      TaskQueue[occupy].state = TASK_READY;
19
20      printf("End of Activate %s\n", name);
21
22      Schedule(occupy);
23  }
24
25  void TerminateTask(void) {
26      int task = RunningTask;
27
28      printf("[Terminate Task] %s\n", TaskQueue[task].name);
29      TaskQueue[task].state = TASK_DONE;
30      RunningTask = _NULL;
31
32      printf("End of Terminate Task %s\n", TaskQueue[task].name);
33
34      Dispatch();
35  }
36
37  void Schedule(int task, int mode) {
38      printf("[Schedule] %s\n", TaskQueue[task].name);
39
40      int cur = HeadTask;
41      int prev = _NULL;
42      int priority = TaskQueue[task].priority;
43
44
45      while (cur != _NULL && TaskQueue[cur].priority > priority) {
46          prev = cur;
47          cur = TaskQueue[cur].ref;
48      }
49
50      if (mode == INSERT_TO_TAIL) {
51          while (cur != _NULL && TaskQueue[cur].priority == priority) {
52              prev = cur;
53              cur = TaskQueue[cur].ref;
54          }
55      }
56
57      TaskQueue[task].ref = cur;
58      if (prev == _NULL) { HeadTask = task; }
```

```
59      else TaskQueue[prev].ref = task;
60      TaskQueue[task].state = TASK_READY;
61
62      printf("End of Schedule %s\n", TaskQueue[task].name);
63
64      Dispatch();
65  }
66
67  void Dispatch() {
68      int task = HeadTask;
69      while (task != _NULL) {
70          switch (TaskQueue[task].state) {
71              case TASK_DONE:
72                  task = TaskQueue[task].ref;
73                  continue;
74              case TASK_RUNNING:
75                  printf("%s is running\n", TaskQueue[task].name);
76                  return;
77              case TASK_READY:
78                  if (RunningTask != _NULL) {
79                      task = TaskQueue[task].ref;
80                      continue;
81                  }
82                  printf("%s started\n", TaskQueue[task].name);
83                  RunningTask = task;
84                  TaskQueue[task].state = TASK_RUNNING;
85
86                  if (TaskQueue[task].res != _NULL) {
87                      PIP_GetRes(TaskQueue[task].res);
88                  }
89                  TaskQueue[task].switchNumber++;
90                  if (TaskQueue[task].switchNumber == 1) {
91                      TaskQueue[task].entry();
92                  } else {
93                      longjmp(TaskQueue[task].context, 1);
94                  }
95                  return;
96              case TASK_WAITING:
97                  task = TaskQueue[task].ref;
98                  continue;
99          }
100     }
101 }
102
```

test.cpp

```
1  #include <stdio.h>
2  #include "rtos_api.h"
3
4  // Объявление задач.
5  DeclareTask(Task1, 1);
```

```
6  DeclareTask(Task2, 2);
7  DeclareTask(Task3, 3);
8  DeclareTask(Task4, 1);
9  DeclareTask(Task5, 2);
10 DeclareTask(Task6, 2);
11 DeclareTask(Task7, 0);
12 DeclareTask(Task8, 1);
13 DeclareTask(Task9, 2);
14 DeclareTask(Task10, 3);
15
16
17 char nameTask6[] = "Task6";
18 char nameTask8[] = "Task8";
19 char nameTask9[] = "Task9";
20
21 // Объявления событий.
22
23 DeclareSysEvent(1);
24 DeclareSysEvent(2);
25 DeclareSysEvent(3);
26
27 int Res1;
28
29 int main(void) {
30     Res1 = InitRes("Res1");
31     char name1[] = "Task1";
32     char name2[] = "Task4";
33     char name3[] = "Task6";
34     char name4[] = "Task8";
35     //StartOS(Task1, Task1prior, name1);
36     //StartOS(Task4, Task4prior, name2);
37     //StartOS(Task6, Task6prior, name3);
38     StartOS(Task8, Task8prior, name4);
39
40     ShutdownOS();
41     return 0;
42 }
43
44 // Test 1. nonpreemptive algorithm
45 TASK(Task1) {
46     printf("\nThis is Task1!\n\n");
47     char name[] = "Task2";
48     char name1[] = "Task3";
49     ActivateTask(Task2, Task2prior, name);
50     ActivateTask(Task3, Task3prior, name1);
51     TerminateTask();
52 }
53
54 TASK(Task2) {
55     printf("\nThis is Task2!\n\n");
```

```
56      TerminateTask();
57  }
58
59  TASK(Task3) {
60      printf("\nThis is Task3!\n\n");
61      TerminateTask();
62  }
63  // Test 2. PIP resource managment algorithm
64  TASK(Task4) {
65      printf("\nThis is Task4!\n\n");
66      PIP_GetRes(Res1);
67      char name[] = "Task5";
68      ActivateTask(Task5, Task5prior, name);
69      PIP_ReleaseRes(Res1);
70      TerminateTask();
71  }
72
73  TASK(Task5) {
74      printf("\nThis is Task5!\n\n");
75      PIP_GetRes(Res1);
76      PIP_ReleaseRes(Res1);
77      TerminateTask();
78  }
79
80  // Test 3. Tasks events
81  TASK(Task6) {
82      printf("\nThis is Task6!\n\n");
83      char name[] = "Task7";
84      ActivateTask(Task7, Task7prior, name);
85      WaitSysEvent(Event_1);
86      TEventMask event;
87      GetSysEvent(&event);
88      printf("GetEvent=%i\n", event);
89      printf("Task6\n");
90      TerminateTask();
91  }
92
93  TASK(Task7) {
94      printf("\nThis is Task7!\n\n");
95      SetSysEvent(Event_1);
96      printf("Task7\n");
97      TerminateTask();
98  }
99
100 // Test 4. Full Functionality
101
102 TASK(Task8) {
103     printf("\nStart Task8\n\n");
104     char name[] = "Task9";
105     ActivateTask(Task9, Task9prior, name);
```

```
106     WaitSysEvent(1);
107     printf("\nTask 8 finished\n\n");
108     TerminateTask();
109 }
110
111 TASK(Task9) {
112     printf("\nStart Task9\n\n");
113     PIP_GetRes(Res1);
114     char name1[] = "Task10";
115     ActivateTask(Task10, Task10prior, name1);
116     char name2[] = "Task8";
117     SetSysEvent(1);
118     WaitSysEvent(2);
119     PIP_ReleaseRes(Res1);
120     printf("\nTask 9 finished\n\n");
121     TerminateTask();
122 }
123
124 TASK(Task10) {
125     printf("\nStart Task10\n\n");
126     PIP_GetRes(Res1);
127     char name[] = "Task9";
128     SetSysEvent(2);
129     PIP_ReleaseRes(Res1);
130     printf("\nTask 10 finished\n\n");
131     TerminateTask();
132 }
```