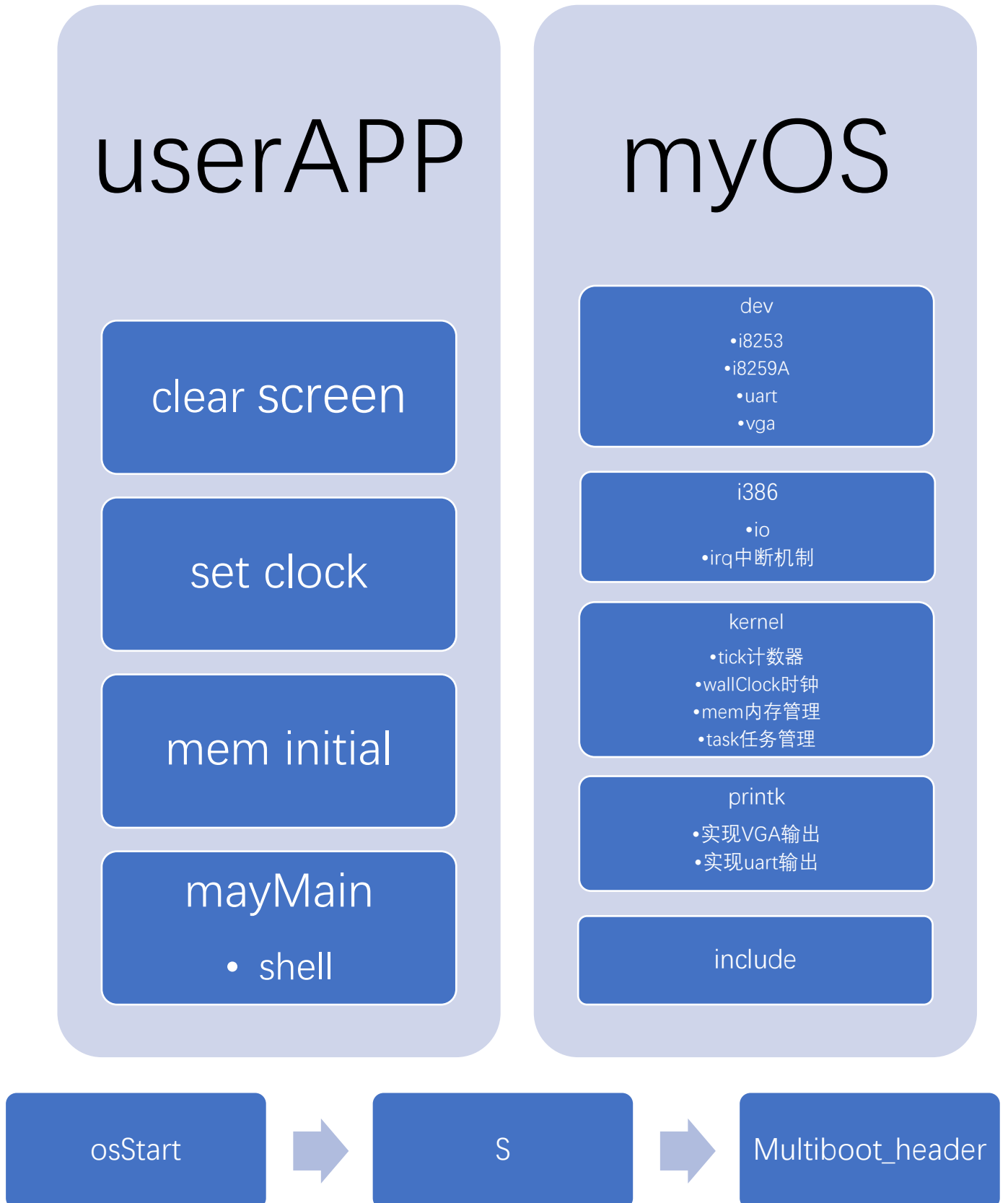


操作系统实验 5 报告

软件框图



主流程及其实现

在上次实验的基础上，增加下列内容。



主要功能模块及其实现

本次实验中要实现的功能模块分别按五个部分实现

TCB数据结构创建

- 栈
- ID
- 状态



任务池的建立

- 创建TCB数组作为任务池
- TCB初始化



上下文切换

- 封装CTX_SW
- 修改相关变量



task队列的建立

- 队列数据结构建立
- 创建与销毁
- 入队与出队
- 启动与结束
- 队空
- 求下一任务
- 初始化



其他

- OsStart修改
- main修改
- Shell实现end命令

源代码说明

在本次实验的根目录下，除了配置了 makefile 与 DS 文件，还有 4 个文件夹：multibootheader、myOS、output、userApp。multibootheader 中放置了有关 multibootheader 协议的 S 文件。在 output 中，原本为空文件夹，用来输出编译后生成的文件。在该文件夹中的目录结构与根目录相似，主要是为了对每个文件输出时不产生混淆。在 userApp 中存放的是 main 文件、memTestCase 文件、userTasks 文件与该文件夹下的 makefile 文件，本次实验中的 userTasks 是提供的测试 Task 相关功能文件。在 myOS 文件夹中存放了本次实验的主要源文件。直接存放在此该目录下的文件有 DS 文件、该文件夹下的 makefile 文件、链接器文件、一个配置地址的 S 文件和一个包含了 main 函数的与 multibootheader 中的 S 文件相关联的 C 文件。这个文件也是从汇编到 C 的转变。在该目录下则有 6 个实现相关功能的文件夹：dev、i386、printk、kernel、lib、include。每个文件夹下都有一个相应的 makefile 文件以及实现功能的 C 语言源文件。本次实验中在 lib 中添加了 task.h 文件，主要功能时任务池的空间声明与 TCB 数据结构的定义，特别的，我在 lib 中自行添加了颜色宏的头文件，方便用宏定义写入 myPrint 的颜色参数。kernel 文件夹中增加 task.c 源代码文件以实现 task 的相关功能。每个子目录下的 makefile 文件的输出目录都是在 output 中的同名目录。下面展示相关源代码。

```

-#ifndef __TASK_H__
#define __TASK_H__

-#ifndef USER_TASK_NUM
#include "../userApp/userApp.h"
-#endif

#define TASK_NUM (2 + USER_TASK_NUM) // at least: 0-idle, 1-init

#define STACK_SIZE 0x8000 // size of a task stack

#define initTskBody myMain // connect initTask with myMain

#define NEW 0 //新的
#define READY 1 //就绪
#define WAIT 2 //等待
#define RUN 3 //运行
#define DEATH 4 //销毁

void initTskBody(void);

void CTX_SW(void*prev_stkTop, void*next_stkTop);

-#typedef struct myTCB {
    unsigned long* stkTop; //栈顶指针
    unsigned long stkBase[STACK_SIZE]; //栈底指针
    unsigned short int state; //状态
    unsigned short int pid; //任务ID
    struct myTCB* next; //下一个结点
    void (*task)(void); //任务
-} myTCB;

myTCB tcbPool[TASK_NUM];

myTCB * idleTsk; // idle 任务 */
myTCB * currentTsk; // 当前任务 */
myTCB * firstFreeTsk; // 下一个空 TCB */

void TaskManagerInit(void);

-#endif

```

task.h 代码，主要是自行定义了 TCB 数据结构的内容，同时还定义了若

干 task 状态宏

```

1  #include "../include/task.h"
2  #include "../include/myPrintk.h"
3
4  void schedule(void);
5  void destroyTsk(int takIndex);
6
7  /**
8   * 内部接口参考
9   */
10 typedef struct rdyQueueFCFS
11 {
12     myTCB* head;
13     myTCB* tail;
14     myTCB* idleTsk;
15 } rdyQueueFCFS;
16
17 rdyQueueFCFS rqFCFS;
18
19 void rqFCFSInit(myTCB* idleTsk)
20 {
21     rqFCFS.head = (myTCB*)0;
22     rqFCFS.tail = (myTCB*)0;
23     rqFCFS.idleTsk = idleTsk;
24 }
25
26 int rqFCFSIsEmpty(void)
27 {
28     if ((rqFCFS.head == (myTCB*)0) && (rqFCFS.tail == (myTCB*)0))
29         return 1;
30     else
31         return 0;
32 }
33
34 myTCB* nextFCFSTsk(void)
35 {
36     if (rqFCFSIsEmpty() == 1)
37         return rqFCFS.idleTsk;
38     else
39         return rqFCFS.head;
40 }
41

```

task.c 代码部分 1，主要参考了教师展示的代码，实现了判断队空、求下一 task、初始化等功能。

```

42  /* tskEnqueueFCFS: insert into the tail node */
43  void tskEnqueueFCFS(myTCB* tsk)
44  {
45      if (rqFCFSIsEmpty() == 1)
46          rqFCFS.head = tsk;
47      else
48          rqFCFS.tail->next = tsk;
49      rqFCFS.tail = tsk;
50  }
51
52  /* tskDequeueFCFS: delete the first node */
53  void tskDequeueFCFS(myTCB* tsk)
54  {
55      rqFCFS.head = rqFCFS.head->next;
56      if (tsk == rqFCFS.tail)
57          rqFCFS.tail = (myTCB*)0;
58  }
59
60  // 用于初始化新创建的 task 的栈
61  // 这样切换到该任务时不会 stack underflow
62  void stack_init(unsigned long** stk, void (*task)(void))
63  {
64      *(*stk)-- = (unsigned long)0x08;      //CS高地址
65      *(*stk)-- = (unsigned long)task;      //eip
66      *(*stk)-- = (unsigned long)0x0202;    //init eflags: IF=1,BI
67      *(*stk)-- = (unsigned long)0xAAAAAAA; //EAX
68      *(*stk)-- = (unsigned long)0xCCCCCCC; //ECX
69      *(*stk)-- = (unsigned long)0xDDDDDDD; //EDX
70      *(*stk)-- = (unsigned long)0BBBBBBB; //EBX
71      *(*stk)-- = (unsigned long)0x4444444; //ESP
72      *(*stk)-- = (unsigned long)0x5555555; //EBP
73      *(*stk)-- = (unsigned long)0x6666666; //ESI
74      *(*stk) = (unsigned long)0x7777777;  //EDI低地址
75  }
76
77  void tskStart(myTCB* tsk)
78  {
79      tsk->state = READY;
80      tskEnqueueFCFS(tsk);
81  }
82

```

task.c 代码部分 2，实现出入队以及栈的初始化、任务的开始

```

83  - void tskEnd(void)
84      {
85          myPrintk(GREEN, "this task is end\n");
86          tskDequeueFCFS(currentTsk);
87          destroyTsk(currentTsk->pid);
88          schedule();
89      }
90
91  - /* createTsk
92     * tskBody():
93     * return value: taskIndex or, if failed, -1
94     */
95  - int createTsk(void (*tskBody)(void))
96      {
97          myPrintk(DARK_GREEN, "Creat a new task...\n");
98          - if (firstFreeTsk != (myTCB*)0)
99              {
100                  myTCB* newTCB = firstFreeTsk;
101                  firstFreeTsk = newTCB->next;
102                  newTCB->state = WAIT;
103                  newTCB->task = tskBody;
104                  newTCB->stkTop = newTCB->stkBase + STACK_SIZE - 1;
105                  newTCB->next = (myTCB*)0;
106                  stack_init(&(newTCB->stkTop), tskBody);
107                  myPrintk(GREEN, "Succcecss, now, Start\n");
108                  tskStart(newTCB);
109                  return newTCB->pid;
110              }
111          else
112              return -1;
113      }
...

```

task.c 代码部分 3， 主要实现任务的结束与创建


```

119 void destroyTsk(int takIndex)
120 {
121     //查找
122     for (int i = 0; i < TASK_NUM; i++)
123     {
124         if (takIndex == tcbPool[i].pid)
125         {
126             tcbPool[i].state = DEATH;
127             break;
128         }
129     }
130     //空闲Tsk查找
131     for(int i=0;i<TASK_NUM;i++)
132     {
133         if (tcbPool[i].state == DEATH || tcbPool[i].state == NEW)
134         {
135             firstFreeTsk = tcbPool + i;
136             return;
137         }
138     }
139 }
140
141 unsigned long** prevTSK_StackPtr;
142 unsigned long* nextTSK_StackPtr;
143 //对CTX_SW的封装
144 void context_switch(myTCB* prevTsk, myTCB* nextTsk)
145 {
146     prevTSK_StackPtr = &(prevTsk->stkTop);
147     nextTSK_StackPtr = nextTsk->stkTop;
148     currentTsk = nextFCFSTsk();
149     myPrintk(GREEN, "CTX_SW...\n");
150     CTX_SW(prevTSK_StackPtr, nextTSK_StackPtr);
151 }
152
153 void scheduleFCFS(void)
154 {
155     context_switch(currentTsk, nextFCFSTsk());
156 }
157
158 void schedule(void)
159 {
160     scheduleFCFS();
161 }

```

task.c 代码部分 4，主要实现任务的销毁、封装实现的上下文切换与

调度

```

160     * idle 任务
161     */
162     void tskIdleBdy(void)
163     {
164         while (1)
165         {
166             schedule();
167         }
168     }
169
170     unsigned long BspContextBase[STACK_SIZE];
171     unsigned long* BspContext;
172
173     //start multitasking
174     void startMultitask(void)
175     {
176         BspContext = BspContextBase + STACK_SIZE - 1;
177         prevTSK_StackPtr = &BspContext;
178         currentTsk = nextFCFSTsk();
179         nextTSK_StackPtr = currentTsk->stkTop;
180         myPrintk(GREEN, "CTX_SW runing start...\n");
181         CTX_SW(prevTSK_StackPtr, nextTSK_StackPtr);
182         myPrintk(GREEN, "SuccessInit!\n");
183     }

```

task.c 代码部分 5，主要实现了 idle 任务和多任务模式

```

185     void TaskManagerInit(void)
186     {
187         //初始化 TCB 数组
188         for (int i = 0; i < TASK_NUM; i++)
189         {
190             tcbPool[i].pid = i;
191             if (i < TASK_NUM - 1)
192                 tcbPool[i].next = tcbPool + i + 1;
193             else
194                 tcbPool[i].next = (myTCB*)0;
195             tcbPool[i].stkTop = tcbPool[i].stkBase + STACK_SIZE - 1;
196         }
197         //创建 idle 任务
198         idleTsk = tcbPool;
199         stack_init(&(idleTsk->stkTop), tskIdleBdy);
200         rqFCFSInit(idleTsk);
201         firstFreeTsk = tcbPool + 1;
202         //创建 init 任务 (使用 initTskBody)
203         createTsk(initTskBody);
204         //切入多任务状态
205         myPrintk(DARK_GREEN, "START MULTITASKING.....\n");
206         startMultitask();
207         myPrintk(DARK_GREEN, "STOP MULTITASKING.....ShutDown\n");
208     }

```

task.c 代码部分 6，主要实现任务管理的初始化

```

#include "include/i8253.h"
#include "include/i8259.h"
#include "include/irq.h"
#include "include/uart.h"
#include "include/vga.h"
#include "include/mem.h"
#include "include/task.h"
#include "include/myPrintk.h"

void osStart(void)
{
    //pressAnyKeyToStart(); // prepare for uart device
    init8259A();
    init8253();
    enable_interrupt();

    clear_screen();

    pMemInit();
    unsigned long tmp = dPartitionAlloc(pMemHandler, 100);
    dPartitionWalkByAddr(pMemHandler);
    dPartitionFree(pMemHandler, tmp);
    dPartitionWalkByAddr(pMemHandler);

    TaskManagerInit();
    while (1);
}

```

osStart 代码部分

```

14 // init task 入口
15 void myMain(void)
16 {
17     clear_screen();
18
19     createTsk(myTsk0);
20     createTsk(myTsk1);
21     createTsk(myTsk2);
22
23     initShell();
24     memTestCaseInit();
25     createTsk(startShell);
26
27     tskEnd();
28 }
29

```

main 代码部分，将 Shell 作为一个

Task

```
int end(int argc, unsigned char** argv)
{
    tskEnd();
    return 0;
}
```

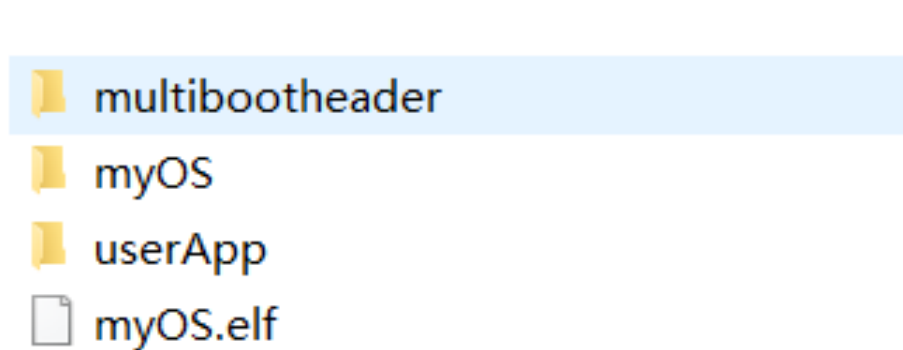
除了讲义要求的必须部分，我还实现了一个 end 指令，其作用即在 Shell 中执行 tskEnd()

代码布局说明

所有的引导模块将按页（4KB）边界对齐，物理内存地址从 1M 处开始。本次实验中在封装上下文切换的函数时修改了 current_task 和 prev/next_task 的信息。

编译过程说明：

在 Ubuntu 中先搜索到 lab4 的目录，然后通过指令 make 完成编译，可以看到在 output 目录中的对应目录中分别输出了与根目录下对应文件相同文件。



运行和运行结果说明：

在 Ubuntu 中通过 QEMU 启动已经编译生成的 bin 文件，得到 Linux 的图形化界面运行结果，然后再通过 Ubuntu 启动一个交互界面，用于输入与输出。

```
QEMU
Creat a new task...
Succecss,now,Start
Creat a new task...
Succecss,now,Start
Creat a new task...
Succecss,now,Start
this task is end
CTX_SW...
*****
*      Tsk0: HELLO WORLD!      *
*****
this task is end
CTX_SW...
*****
*      Tsk1: HELLO WORLD!      *
*****
this task is end
CTX_SW...
*****
*      Tsk2: HELLO WORLD!      *
*****
this task is end
CTX_SW...
please input a cmd >:_
```

刚运行内核时的 QEMU 界面。

```
ws@LAPTOP-J9NGC786: /mnt/d/360MoveData/Users/asus/Desktop/Computer_Program/os2020-labs/lab5/src
cmd
cmd
list all registered commands:
command name: description
  testeFP: Init a eFPatition. Alloc all and Free all.
  testdP3: Init a dPatition(size=0x100). A:B:C:- ==> A:B:- ==> A:- ==> - .
  testdP2: Init a dPatition(size=0x100). A:B:C:- ==> -:B:C:- ==> -:C:- ==> - .
  testdP1: Init a dPatition(size=0x100). [Alloc,Free]* with step = 0x20
maxMallocSizeNow: MAX_MALLOC_SIZE always changes. What's the value Now?
testMalloc2: Malloc, write and read.
testMalloc1: Malloc, write and read.
  end: end the shell
  help: help [cmd]
  cmd: list all registered commands
please input a cmd >:end
```

在运行了 end 命令后，QEMU 界面退出。

遇到的问题和解决方案：

1. 不能理解任务管理的具体实现方式

通过网络工具查询和了解不，然后通过借鉴与咨询助教、参考教师所给示例实现。

2. 无法正常对上栈

咨询助教后了解到是没有给栈指针赋给空间，遂将指针成员变量改为数组形式实体空间的自动分配。