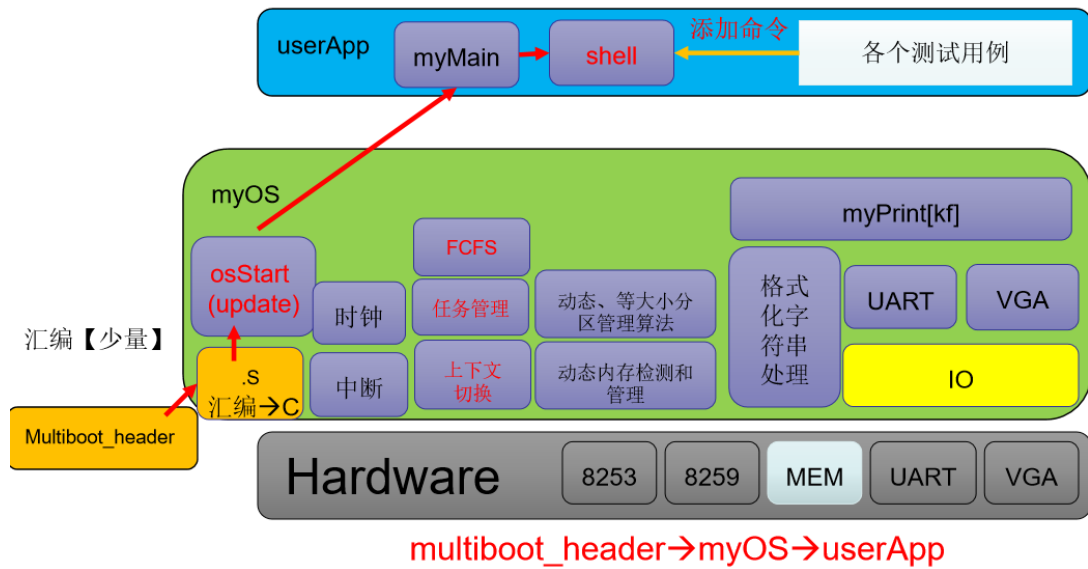


# Lab5 实验报告

PB18051098 徐碧涵

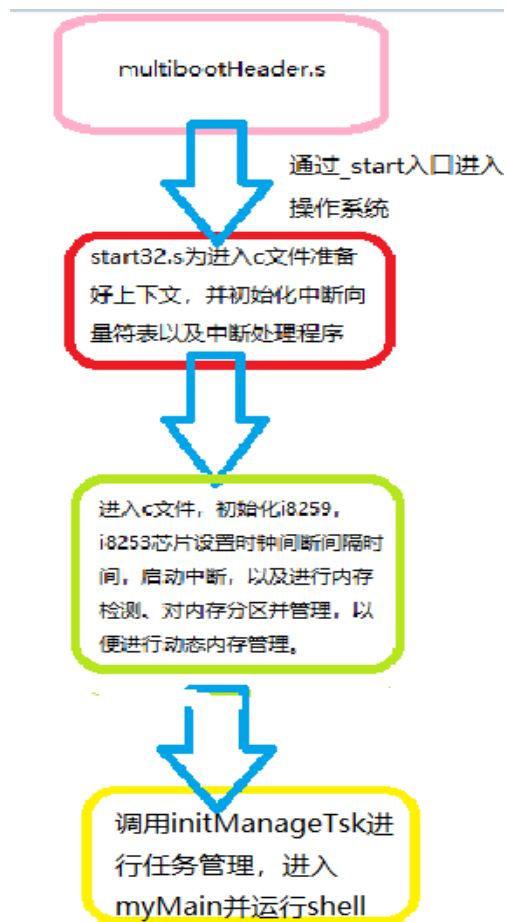
## 软件框图



本实验旨在实现任务管理，并利用 schedule 接口，FCFS 队列实现任务调度，最后通过 userApp 的 myMain 测试任务管理的功能。

## 主流程及其实现

主流程如下：



通过遵循 multiboot 启动协议的 multibootheader.s 文件中的 multiboot\_header 头部启动内核，利用\_start 入口进入操作系统内部，执行 start32.s 文件，在其中为进入 c 文件准备好上下文，初始化中断向量表 IDT。随后进入 c 文件对操作系统进行初始化，通过 init8259A(),init8253()初始化 i8259(可编程中断控制器)，i8253(可编程间隔定时器)芯片，然后启动时钟中断。之后调用 pMemInit()进行内存检测并利用动态分区算法管理内存，为之后动态内存分配和回收做好准备。随后调用 TaskManagerInit 入口进入任务管理，将 myMain 封装成任务进行调度从而实现功能。

## 主要功能模块及其实现

### 主要功能模块

任务数据结构 myTCB

```

struct TCB {
    unsigned long *stkTop;  /* 栈顶指针 */
    int tcbIndex;
    int state;
    unsigned long stack[STACK_SIZE];
    void* next;
};
typedef struct TCB myTCB;

```

## FCFS 队列管理

### 队列管理结构

```

typedef struct rdyQueueFCFS{
    myTCB * head;
    myTCB * tail;
    myTCB * idleTsk;
} rdyQueueFCFS;

rdyQueueFCFS rqFCFS;

```

### 队列初始化

```

void rqFCFSInit(myTCB* idleTsk) {
    rqFCFS.head=(myTCB *)0;
    rqFCFS.tail=(myTCB *)0;
    rqFCFS.idleTsk=idleTsk;
}

```

### 判断队列是否为空

```

int rqFCFSIsEmpty(void) {
    if(rqFCFS.head==(myTCB *)0&&rqFCFS.tail==(myTCB *)0)
        return 1;
    return 0;
}

```

### 返回即将调度的任务

```

myTCB * nextFCFSTsk(void) {
    if(rqFCFSIsEmpty())
        return rqFCFS.idleTsk;
    return rqFCFS.head;
}

```

### 入队操作

```

/* tskEnqueueFCFS: insert into the tail node */
void tskEnqueueFCFS(myTCB *tsk) {
    if(rqFCFSIsEmpty()){
        rqFCFS.head=tsk;
    }
    else
        rqFCFS.tail->next=tsk;
    rqFCFS.tail=tsk;
}

```

出队操作

```

/* tskDequeueFCFS: delete the first node */
void tskDequeueFCFS(myTCB *tsk) {
    rqFCFS.head=rqFCFS.head->next;
    if(tsk==rqFCFS.tail)
        rqFCFS.tail=(myTCB *)0;
}

```

Tskstart 将任务加入就绪队列

```

void tskStart(myTCB *tsk){
    tsk->state=TSK_RDY;
    tskEnqueueFCFS(tsk);
}

```

Tskend 当前任务从就绪队列出列，并销毁该任务，进行调度

```

void tskEnd(void){
    tskDequeueFCFS(currentTsk);
    destroyTsk(currentTsk->tcbIndex); //销毁数据结构
    schedule(); //任务结束进行调度
}

```

初始化任务栈

```
// 用于初始化新创建的 task 的栈
// 这样切换到该任务时不会 stack underflow
void stack_init(unsigned long **stk, void (*task)(void)){
    *(*stk)--=(unsigned long) 0x08;
    *(*stk)--=(unsigned long) task;

    *(*stk)--=(unsigned long) 0x0202;

    *(*stk)--=(unsigned long) 0xAAAAAAA;
    *(*stk)--=(unsigned long) 0xCCCCCCC;
    *(*stk)--=(unsigned long) 0xDDDDDDD;
    *(*stk)--=(unsigned long) 0BBBBBBBB;

    *(*stk)--=(unsigned long) 0x44444444;
    *(*stk)--=(unsigned long) 0x55555555;
    *(*stk)--=(unsigned long) 0x66666666;
    *(*stk)  =(unsigned long) 0x77777777;
}
```

CreateTsk 创建任务

```
int createTsk(void (*tskBody)(void)) {
    //作业池中分配数据结构；初始化栈；加入调度队列
    if(firstFreeTsk!=(myTCB *)0)
    {
        myTCB *alloc=firstFreeTsk;
        firstFreeTsk=firstFreeTsk->next;
        alloc->next=(myTCB *)0;
        stack_init(&(alloc->stkTop),tskBody);
        tskStart(alloc);
        return alloc->tcbIndex;
    }
    return -1;
}
```

DestroyTsk 销毁任务

```
//将数据结构归还tcbPool
void destroyTsk(int takIndex) {
    tcbPool[takIndex].state=1;
    tcbPool[takIndex].next=firstFreeTsk;
    firstFreeTsk=&tcbPool[takIndex];
}
```

切换上下文

```

void context_switch(myTCB *prevTsk, myTCB *nextTsk) {
    prevTSK_StackPtr=&prevTsk->stkTop;
    nextTSK_StackPtr = nextTsk->stkTop;
    CTX_SW(prevTSK_StackPtr,nextTSK_StackPtr);
}

```

调度

```

void scheduleFCFS(void) {
    myTCB *nextTsk,*preTsk;
    preTsk=currentTsk;
    currentTsk=nextTsk=nextFCFSTsk();
    context_switch(preTsk,nextTsk);
}

```

Idle 任务

```

void tskIdleBdy(void) {
    while(1)
        schedule();
}

```

Startmultitask 进入多任务状态

```

//start multitasking, 切换到init任务
void startMultitask(void) {
    BspContext = BspContextBase + STACK_SIZE -1;
    prevTSK_StackPtr = &BspContext;
    currentTsk = nextFCFSTsk();
    nextTSK_StackPtr = currentTsk->stkTop;
    CTX_SW(prevTSK_StackPtr,nextTSK_StackPtr);
}

```

TaskManagerInit 任务管理初始化

```

void TaskManagerInit(void) {

    // 初始化 TCB 数组
    // 创建 idle 任务
    // 创建 init 任务（使用 initTskBody）
    // 切入多任务状态

    int i;
    myTCB *thisTCB;
    for(i=0;i<TASK_NUM;i++){
        thisTCB=&tcbPool[i];
        thisTCB->tcbIndex=i;
        if(i==TASK_NUM-1)
            thisTCB->next=(myTCB *)0;
        else
            thisTCB->next=&tcbPool[i+1];
        thisTCB->stkTop=thisTCB->stack+STACK_SIZE-1;
    }
    idleTsk=&tcbPool[0];
    stack_init(&(idleTsk->stkTop),tskIdleBdy);
    rqFCFSInit(idleTsk);
    firstFreeTsk=&tcbPool[1];
    createTsk(initTskBody);
    startMultitask();
}

```

流程图如下：



## Makefile 组织

基本上每一个文件夹下都有一个 **Makefile** 文件将目标文件生成中间文件, 便于后期整体组织

*目录组织如下:*

```

multibootheader
    multibootheader.s
myOS
    i386

```



- io.c
- irq.s
- irqs.c
- CTX\_SW.s
- Makefile
- dev
  - i8253.c
  - i8259A.c
  - uart.c
  - vga.c
  - Makefile
- printk
  - vsprintf.c
  - myprintk.c
  - Makefile
- kernel
  - mem
    - dPartition.c
    - eFPartition.c
    - Makefile
    - kmalloc.c
    - malloc.c
    - pMemInit.c
    - task.c
  - tick.c
  - WallClock.c
  - Makefile
- lib
  - string.c
  - Makefile
- Makefile
- myOS.ld
- osStart.c
- start32.s
- userApp
  - main.c
  - shell.c
  - Makefile
- Makefile
- source2img.sh

底层文件夹下的 Makefile 文件使当前文件夹下的.c 文件生成中间文件，处于上层文件夹的 Makefile 将下层文件夹中生成的中间文件进行组织最后由最顶层文件夹下的 Makefile 汇总生成目标文件。

## 代码布局说明(地址空间)

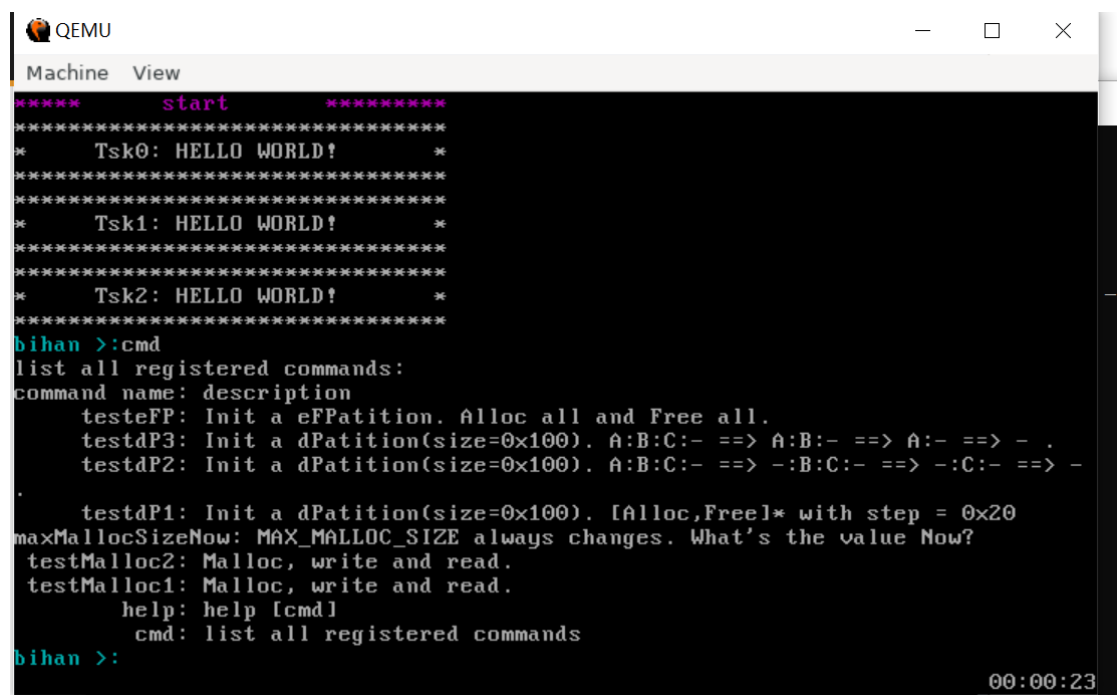
代码在内存空间中的地址以及布局由 myOS.ld 文件所确定，代码将从物理内存 1M 的位置开始存放，代码段 8 个字节对齐，之后存放程序中已经初始化的全局变量并按 16 字节对齐，在之后存放未初始化的全局变量按 16 字节对齐，下一个存放的代码按 512 个字节对齐

## 编译过程说明

通过 ./source2img.sh 命令执行 make 命令在 Makefile 的组织下使所有 .c.s 文件生成中间文件最终按照 myOS.ld 文件代码布局以及地址空间的要求在内存空间生成并存放 .elf 文件。

## 运行和运行结果说明

输入 ./source2img.sh 命令运行脚本文件，编译后进而执行 .elf 文件，运行结果如下：



```
QEMU
Machine View
***** start *****
*****
Tsk0: HELLO WORLD!
*****
Tsk1: HELLO WORLD!
*****
Tsk2: HELLO WORLD!
*****
bihan >: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.
help: help [cmd]
cmd: list all registered commands
bihan >:
00:00:23
```

## 遇到的问题和解决方案说明

问题：任务切换一直失败

解决方案：stack\_init 栈顶指针多减减了一次。