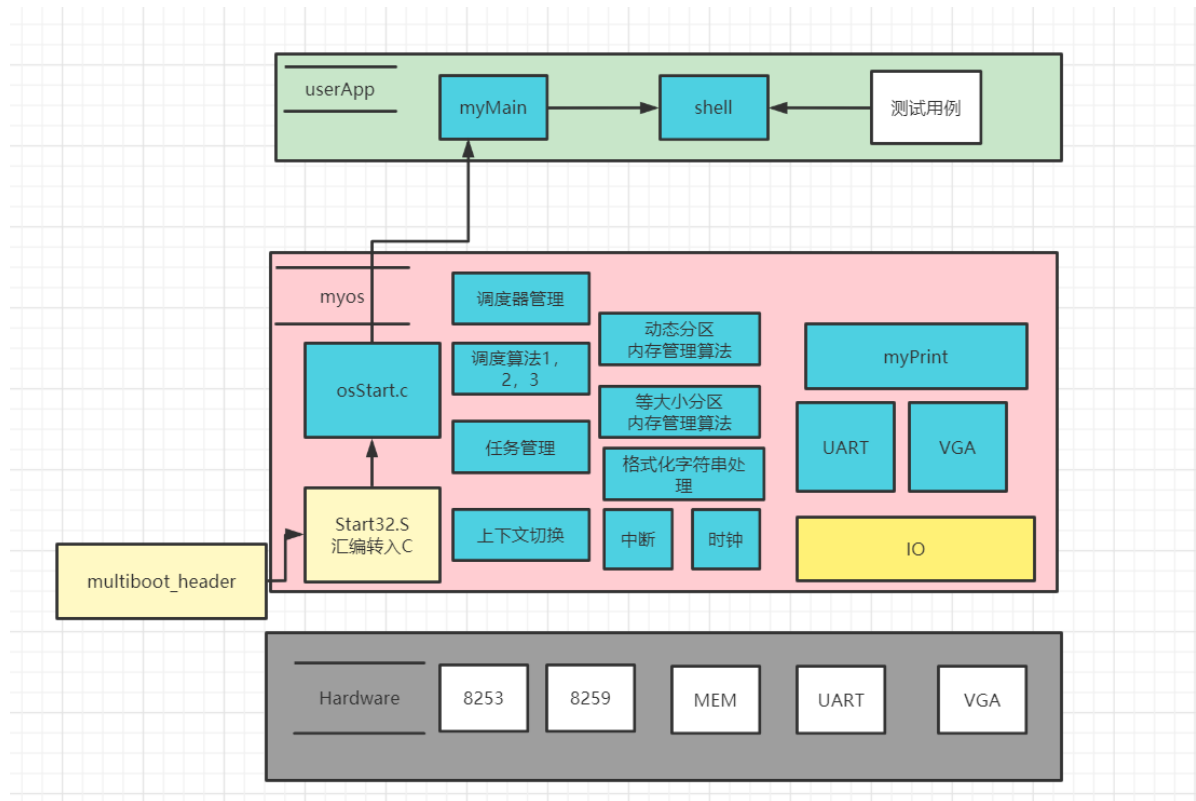
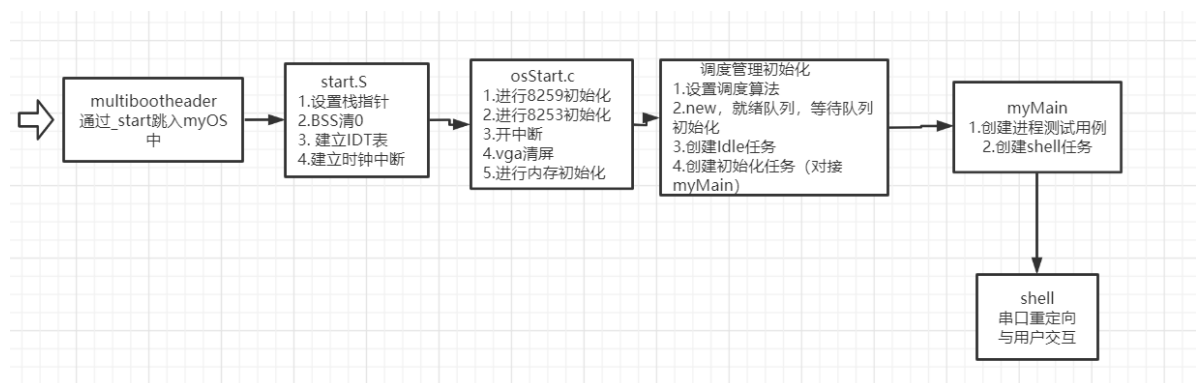


lab6实验报告

一、软件框图



二、主流程



2.文字说明

计算机开机之后，经过一些初始化步骤后，首先读取multibootheader文件，在校验对接完成之后，结尾处将借助myOS提供的_start 入口跳入myOS中。

进入myOS中首先执由汇编语言构成的start.S文件，这个文件会进行设置栈和将BSS 段清0两个操作，为下面的C语言执行提供必要的环境。接着进行IDT表的建立工作，为256个中断向量描述符开辟空间，初始化IDTR，并将每个向量均指向缺省函数。之后正式建立时间中断向量，跳入osStart.c中

进入osStart.c中调用库函数进行初始化工作（包括对8259，8253，mem的硬件初始化），初始化完成之后开中断，调用库函数进行VGA清屏，输出开始信息，之后进入调度器初始化，设置调度算法，初始化队列，创建Idle任务，创建init任务，对接main函数，之后便跳入main中。

main.c函数创建进程测试用例，创建shell任务，在测试用例结束后，shell进程开始运行，在终端与用户进行交互，读取用户命令，并进行处理。

三、功能模块

1. 相关的数据结构

```
// myTCB数据结构
typedef struct myTCB{
    //base_para
    unsigned long tid;
    unsigned long state;
    unsigned long *stackPtr;
    unsigned long *pc;

    struct myTCB *next;

    //expand_para
    unsigned long priority;
    unsigned long arrTime;
    unsigned long exeTime;
}myTCB;

//队列
typedef struct Queue{
    struct myTCB *head;
    struct myTCB *tail;
}Queue;

// 任务块参数
typedef struct tskPara {
    unsigned long priority;
    unsigned long arrTime;
    unsigned long exeTime;
} tskPara;
```

分别实现了任务管理块，队列（就绪与新创建队列均用此结构），任务参数块。

2. 任务原语

下面的函数是对进程的各个操作，包括创建任务，销毁任务，开始任务，结束任务四个任务原语。

此段代码为核心代码，其中的new,ready,running,terminal分别对应任务转换图中的四个模块。因为不涉及等待状态，故没有关于此状态的函数。

(1) 创建进程

```
// 创建新任务
unsigned long null2new(void (*tskBody)(void), tskPara* para){
    unsigned long start = kmalloc(sizeof(myTCB));

    myTCB newTsk;
    // 初始化基本参数
    newTsk.tid = getTid();
```

```

newTsk.state = NEW;
newTsk.pc = (unsigned long*)tskBody;
newTsk.next = (void*) 0;
newTsk.stackPtr = (unsigned long *) (kmalloc(STACK_SIZE) + STACK_SIZE - 4);
// 初始化拓展参数
newTsk.arrTime = para->arrTime;
newTsk.exeTime = para->exeTime;
newTsk.priority = para->priority;
// 初始化栈
stackInit(&(newTsk.stackPtr), tskBody);
//log: 栈低指针出现错误, 意外的修改了EMB

memcpy((void*)start, &newTsk, sizeof(myTCB));

// 插入队列, 若达到时间为0, 应立即启动
if(newTsk.arrTime == 0){
    ((myTCB*)start)->state = READY;
    tskEnqueue_sche(&rdyQueue, (myTCB*)start);
}
else{
    tskEnqueueSort(&newQueue, (myTCB*)start); //插入新建队列
}
return newTsk.tid;
}

```

创建过程会申请两次空间, 一次申请分配控制块所需要的内存, 一次申请分配进程所需要的栈, 同时会初始化一些相应的参数。若任务到达时间小于系统时间, 将该任务直接放入就绪队列。否则, 插入new队列。

(2) 启动任务

```

// 启动任务
/* lab6 要求根据到达时间实现任务就绪动态化 */
void new2ready(unsigned long tid){
    //在new队列删除, 插入wait队列
    myTCB *tsk;
    tsk = tskRemove(&newQueue, tid);
    tsk->state = READY;
    tskEnqueue_sche(&rdyQueue, tsk); // 挂钩函数
}

```

启动任务即将任务从new队列转移到就绪队列, 同时设置该进程为就绪状态, 等待CPU调度。在lab6中该原语与时钟中断相关。

(3) 销毁任务

```

// 销毁任务, 释放栈空间
void ready2terminal(unsigned long tid){
    //在ready队列删除, 释放栈空间
    myTCB *tsk;
    tsk = tskRemove(&rdyQueue, tid);
    kfree((unsigned long)tsk->stackPtr);
    kfree((unsigned long)tsk);
}

```

从就绪队列移除该进程，释放栈空间和进程块本身空间。

(4) 结束任务

```
// 结束任务
void running2terminal(){
    currentTsk->state = TERMINAL;
    if(currentTsk->tid != 1){
        kfree((unsigned long)currentTsk->stackPtr);
    }
    else{//对于Idle特殊处理，将其进程栈空间保留，并重新初始化
        currentTsk->stackPtr = IdleTskStkPtr;
        stackInit(&(currentTsk->stackPtr), idleTsk);
    }
}
```

将当前运行的任务块状态设置为terminal，若该进程不为Idle任务则释放该进程的栈空间，释放进程块本身空间不在此原语中，由另一个函数完成。函数如下：

```
// 释放TCB空间
void terminal2null(myTCB *tsk){
    //释放进程块本身
    kfree((unsigned long) tsk);
}
```

将上面的函数进行封装，得到下面的任务原语

```
// 任务创建
unsigned long createTsk(void (*tskBody)(void), tskPara* para){
    null2new(tskBody, para);
}

// 任务销毁
void destroyTsk(unsigned long tid){
    ready2terminal(tid);
}

// 任务启动
void tskStart(unsigned long tid){
    new2ready(tid);
}

// 任务结束
void tskEnd(){
    running2terminal();
    schedule(); // 为接口，使用不同的调度策略
}
```

其中taskEnd()函数存在于每个进程的末尾。

3. 上下文切换

```
void CTX_SW(void);
// 上下文切换
void contextSwitch(unsigned long **prevTskStkAddr, unsigned long *nextTskStk) {
    prevTSK_StackPtrAddr = prevTskStkAddr;
    nextTSK_StackPtr = nextTskStk;
    CTX_SW();
}

void CTX_NEXT(void);
// 不进行保护现场的上下文切换
void setNextTsk(unsigned long *nextTskStk){
    nextTSK_StackPtr = nextTskStk;
    CTX_NEXT();
}
```

上面两个函数为调用上下文切换的两个C程序接口。第一个是需要调用的是要保存现场的上下文切换，适用于当前进程未执行完毕，需要进行另一个进程的情况。第二个是不保存现场的上下文切换，适用于当前任务以执行完毕，不需要再返回该进程。且切换进程时需要进行关中断。

对应的汇编代码如下：

```
.text
.code32

.global prevTSK_StackPtrAddr
.global nextTSK_StackPtr

.global CTX_SW
.global CTX_NEXT

CTX_SW:
    cli
    # 保护现场
    pushf                    # 旧进程的标志寄存器入栈
    pusha                    # 旧进程的通用寄存器入栈
    movl prevTSK_StackPtrAddr,%eax    # eax是指针
    movl %esp, (%eax)           # ( ) 是访存的标志
    movl nextTSK_StackPtr, %esp    # 新进程的栈顶指针值存入 esp 寄存器
    # 恢复现场
    popa                      # 新进程的通用寄存器出栈
    popf                      # 新进程的标志寄存器出栈
    sti
    ret

CTX_NEXT:
    cli
    movl nextTSK_StackPtr, %esp    # 新进程的栈顶指针值存入 esp 寄存器
    # 恢复现场
    popa                      # 新进程的通用寄存器出栈
    popf                      # 新进程的标志寄存器出栈
    sti
    ret
```

4. 任务管理器初始化

```
void tskManagerInit(void){
    // 设置调度算法
    setSysScheduler(RR);

    // 三个队列的初始化
    initQueue(&newQueue);
    initQueue(&rdyQueue);
    initQueue(&waitQueue);
    currentTsk = (void*)0;

    // 创建两个任务
    myPrintf(0x7, "create idleTsk\n");
    struct tskPara tmp;
    setTskPara(PRIORITY, 0, &tmp);
    setTskPara(EXETIME, INFINITE, &tmp);
    setTskPara(ARRTIME, INFINITE, &tmp);
    createTsk(idleTsk, &tmp); //id 为 1
    IdleTskStkPtr = newQueue.head->stackPtr + 10; // 要栈底指针

    myPrintf(0x7, "create initTsk\n");
    setTskPara(PRIORITY, 0, &tmp);
    setTskPara(EXETIME, 0, &tmp);
    setTskPara(ARRTIME, 0, &tmp);
    createTsk(initTskBdy, &tmp); // id 为 2
    myPrintf(0x7, "\n");

    //进入多任务模式
    myPrintf(0x2, "START MULTITASK...\n");
    startMultitask();
    myPrintf(0x2, "STOP MULTITASK.....SHUTDOWN\n");
}
```

先设置任务调度算法，然后初始化new队列，就绪队列，等待队列（本实验未用到该队列），接着创建Idle任务和Init任务，最后启动多任务运行模式，多任务运行函数代码如下：

```
//启动多任务运行
void startMultitask(void) {
    myTCB *tsk;
    tsk = tskDequeue(&rdyQueue); // 手动使init函数出队
    currentTsk = tsk;
    setNextTsk(currentTsk->stackPtr); // 开始第一个任务
}
```

因为是第一次切换到进程运行，前面没有进程，则使用不保护现场的上下文切换。切换的任务为init任务，而init任务本质就是调用mymain函数。

5. 任务随时钟动态到达

首先更新tick函数

```
void tick(void){
    int temp1, temp2;
    system_ticks++;
    //对HH,MM,SS进行处理
    if(system_ticks == 24 * H_TICKNUM)
        system_ticks = 0;
    HH = system_ticks / H_TICKNUM;
    temp1 = system_ticks % H_TICKNUM;
    MM = temp1 / M_TICKNUM;
    temp2 = temp1 % M_TICKNUM;
    SS = temp2 / S_TICKNUM;
    oneTickUpdateallClock(HH,MM,SS);
    /* lab6 注册新的挂钩函数，作用为使任务动态到达 */
    tskDynArr(system_ticks/S_TICKNUM);
    /* lab6 注册新的挂钩函数，若调度算法与时间片有关，则调用此函数*/
    scheduler_tick();
    return;
}
```

在每次时钟中断时，均调用tskDynArr()函数，代码如下：

```
// 任务动态到达
void tskDynArr(unsigned long sysTime){
    if(newQueue.head->arrTime <= sysTime && newQueue.head->tid != 1){
        tskStart(newQueue.head->tid);
    }
}
```

该函数就是不断检测new队列的第一个进程是否需要启动（new队列按启动时间排序），若任务需要启动，则调用相关原语。

6. 调度器采用统一接口

不同的调度算法对应的某些函数实现不同，但是对外要统一接口，因此涉及hook机制，首先将不同的实现函数组合为一个调度器，数据结构为一个结构体。代码如下：

```
struct scheduler {
    unsigned int type;

    void (*nextTsk_func)(void);
    void (*enqueueTsk_func)(Queue *queue, myTCB *tsk);
    void (*tick_hook)(void); //if set, tick_hook will be called every tick
};
```

由于本人实现的三个调度算法只有这三个函数实现不同，因此调度器只包含这三个函数。

然后根据不同情况挂不同的调度器。

```
void setSysScheduler(unsigned int what){
    switch (what)
```

```

{
    case FCFS:
        setScheFunc(scheduler_FCFS);
        break;
    case SJF:
        setScheFunc(scheduler_SJF);
        break;
    case RR:
        setScheFunc(scheduler_RR);
        break;

    default:
        break;
}
}

```

```

void setScheFunc(struct scheduler sche) {
    nextTsk_hook = sche.nextTsk_func;
    enqueueTsk_hook = sche.enqueueTsk_func;
    tick_hook = sche.tick_hook;
}

```

此外任务参数也需要统一接口，相应函数在taskPara.c中，较为简单，不再展示。

7. FCFS实现

FCFS实现较为简单，因为new队列是根据达到时间排序，且任务是随时间动态达到，则在任务到达后查到就绪队列尾部即可。

```

void tskEnqueue_FCFS(Queue* queue, myTCB* tsk){
    if(queueEmpty(queue)){
        queue->head = tsk;
    }
    else{
        queue->tail->next = tsk;
    }
    queue->tail = tsk;
    tsk->next = (void *)0;
}

```

8. SJF实现

SJF入队函数依靠执行时间从低到高排序，在CPU调度时直接选择就绪队列的头节点即可，无需遍历整个队列。

```

void tskEnqueue_SJF(Queue* queue, myTCB* tsk){
    // 按到达时间排序
    myTCB* p = queue->head;
    myTCB* pre = p;
    if(queueEmpty(queue)){
        queue->head = tsk;
        queue->tail = tsk;
    } // 插头尾节点不同
    else{

```



```

        while(p != (void*)0 && tsk->exeTime >= p->exeTime){
            pre = p;
            p = p->next;
        }
        // 维护尾指针
        if(pre == queue->tail && p == (void*)0){
            queue->tail = tsk;
        }
        // 维护头指针
        if(p == queue->head){
            queue->head = tsk;
        }
        else{
            pre->next = tsk;
        }
        tsk->next = p;
    }
}

```

9. RR实现

RR实现较为重要的是和tick相关的函数，设置时间片为40ms，每次时钟中断计数器加1，达到40后进行判断，此时就绪队列不空，则需要进行任务轮转，将现在运行的任务结束再次插入就绪队列，选择下一个任务运行。

```

void tickHook_RR(){
    //当开始多任务运行，且就绪队列不为空，且现在不为Idle任务运行时才轮转
    if(i++ >= 40 && currentTsk && !queueEmpty(&rdyQueue) && currentTsk->tid !=
1){ //40ms 为一个时间片
        i = 0;
        running2ready(); // 将现在运行的任务放回就绪队列
        nextTsk_RR();
    }
}

```

```

void nextTsk_RR(void){
    i = 0;
    ready2running(); // 从就绪队列选取下一个任务运行
}

```

四、源代码说明

1.目录组织

```

$ tree
.
|-- doc
|   `-- report.pdf
`-- src
    |-- Makefile
    |-- multibootheader

```

```

|  `-- multibootHeader.S
|-- myOS
|  |-- Makefile
|  |-- dev
|  |   |-- Makefile
|  |   |-- i8253.c
|  |   |-- i8259A.c
|  |   |-- uart.c
|  |   `-- vga.c
|  |-- i386
|  |   |-- CTX_SW.S
|  |   |-- Makefile
|  |   |-- io.c
|  |   |-- irq.S
|  |   `-- irqs.c
|  |-- include
|  |   |-- i8253.h
|  |   |-- i8259A.h
|  |   |-- io.h
|  |   |-- irqs.h
|  |   |-- kmalloc.h
|  |   |-- malloc.h
|  |   |-- mem.h
|  |   |-- myPrintk.h
|  |   |-- mymath.h
|  |   |-- mystring.h
|  |   |-- task.h
|  |   |-- taskQueue.h
|  |   |-- tick.h
|  |   |-- tskPara.h
|  |   |-- uart.h
|  |   |-- vga.h
|  |   |-- vsprintf.h
|  |   `-- wallClock.h
|  |-- kernel
|  |   |-- Makefile
|  |   |-- mem
|  |   |   |-- Makefile
|  |   |   |-- dPartition.c
|  |   |   |-- eFPartition.c
|  |   |   |-- kmalloc.c
|  |   |   |-- malloc.c
|  |   |   `-- pMemInit.c
|  |   |-- task
|  |   |   |-- Makefile
|  |   |   |-- idleTsk.c
|  |   |   |-- initTsk.c
|  |   |   |-- scheduler
|  |   |   |   |-- FCFS.c
|  |   |   |   |-- Makefile
|  |   |   |   |-- RR.c
|  |   |   |   |-- SJF.c
|  |   |   |   |-- scheduler.c
|  |   |   |   `-- scheduler.h
|  |   |   |-- task.c

```

```
| | | |-- taskQueue.c
| | | |-- tskManagerInit.c
| | | `-- tskPara.c
| | |-- tick.c
| | `-- wallClock.c
| |-- lib
| | |-- Makefile
| | |-- myPrintk.c
| | |-- mymath.c
| | |-- mystring.c
| | `-- vsprintf.c
| |-- myOS.ld
| |-- osStart.c
| |-- start32.s
| `-- userInterface.h
|-- source2img.sh
|-- tests
| |-- testFCFS
| | |-- FCFSTestCase.c
| | |-- Makefile
| | |-- main.c
| | |-- scheTestCase.c
| | |-- scheTestCase.h
| | |-- shell.c
| | `-- shell.h
| |-- testRR
| | |-- Makefile
| | |-- RRTTestCase.c
| | |-- main.c
| | |-- scheTestCase.c
| | |-- scheTestCase.h
| | |-- shell.c
| | `-- shell.h
| `-- testSJF
|   |-- Makefile
|   |-- SJFTestCase.c
|   |-- main.c
|   |-- scheTestCase.c
|   |-- scheTestCase.h
|   |-- shell.c
|   `-- shell.h
`-- userApp
    |-- FCFSTestCase.c
    |-- Makefile
    |-- main.c
    |-- scheTestCase.c
    |-- scheTestCase.h
    |-- shell.c
    `-- shell.h
```

2.Makefile组织

```
.
├─ MULTI_BOOT_HEADER
│ └─ output/multibootheader/multibootHeader.o
├─ MYOS_OBJS
│ └─ DEV_OBJS
│   │ └─ output/myOS/dev/uart.o
│   │ └─ output/myOS/dev/vga.o
│   │ └─ output/myOS/dev/i8259A.o
│   │ └─ output/myOS/dev/i8253.o
│   └─ I386_OBJS
│     │ └─ output/myOS/i386/io.o
│     │ └─ output/myOS/i386/irqs.o
│     │ └─ output/myOS/i386/irq.o
│     └─ KERNEL_OBJS
│       │ └─ output/myOS/kernel/tick.o
│       │ └─ output/myOS/kernel/wallClock.o
│       │ └─ MEM_OBJS
│       │   │ └─ output/myOS/kernel/mem/pMemInit.o
│       │   │ └─ output/myOS/kernel/mem/dPartition.o
│       │   │ └─ output/myOS/kernel/mem/eFPartition.o
│       │   │ └─ output/myOS/kernel/mem/malloc.o
│       │   │ └─ output/myOS/kernel/mem/kmalloc.o
│       │   └─ TASK_OBJS
│       │     │ └─ output/myOS/kernel/task/idleTsk.o
│       │     │ └─ output/myOS/kernel/task/initTsk.o
│       │     │ └─ output/myOS/kernel/task/task.o
│       │     │ └─ output/myOS/kernel/task/taskQueue.o
│       │     │ └─ output/myOS/kernel/task/tskManagerInit.o
│       │     │ └─ output/myOS/kernel/task/tskPara.o
│       │     └─ SCHEDULER_OBJS
│       │       │ └─ output/myOS/kernel/task/scheduler/FCFS.o
│       │       │ └─ output/myOS/kernel/task/scheduler/RR.o
│       │       │ └─ output/myOS/kernel/task/scheduler/SJF.o
│       │       └─ output/myOS/kernel/task/scheduler/scheduler.o
│       └─ LIB_OBJS
│         │ └─ output/myOS/lib/vsprintf.o
│         │ └─ output/myOS/lib/myPrintk.o
│         │ └─ output/myOS/lib/mymath.o
│         │ └─ output/myOS/lib/mystring.o
│         └─ output/myOS/osStart.o
│ └─ output/myOS/start32.o
└─ USER_APP_OBJS
  └─ output/userApp/main.o
  └─ output/userApp/shell.o
  └─ output/userApp/scheTestCase.o
  └─ output/userApp/XXXTestCase.o
```

最后一项视具体编译而定

五、测试用例与运行结果说明

1. 测试用例源代码

```
void sleep(int time){
    //沉睡几毫秒
    int start = system_ticks;
    while(system_ticks - start < time);
}

void test1(){
    for(int i = 0; i < 10; i++){
        myPrintf(0x1, "test1----->loop:%d\n", i);
        sleep(20);
    }
    tskEnd();
}

void test2(){
    for(int i = 0; i < 10; i++){
        myPrintf(0x2, "test2----->loop:%d\n", i);
        sleep(20);
    }
    tskEnd();
}

void test3(){
    for(int i = 0; i < 10; i++){
        myPrintf(0x3, "test3----->loop:%d\n", i);
        sleep(20);
    }
    tskEnd();
}

void test4(){
    for(int i = 0; i < 10; i++){
        myPrintf(0x4, "test4----->loop:%d\n", i);
        sleep(20);
    }
    tskEnd();
}
```

四个任务，每个的任务都是循环输出相应字符串。相应的参数设定在不同的算法不同。

如SJF的参数设定如下：

```
void SJFTestCase(void){
    tskPara para[4];

    setTskPara(PRIORITY, 0, &para[0]);
    setTskPara(EXETIME, 100, &para[0]);
    setTskPara(ARRTIME, 0, &para[0]);

    setTskPara(PRIORITY, 0, &para[1]);
    setTskPara(EXETIME, 300, &para[1]);
```

```

    setTskPara(ARRTIME, 0, &para[1]);

    setTskPara(PRIORITY, 0, &para[2]);
    setTskPara(EXETIME, 200, &para[2]);
    setTskPara(ARRTIME, 0, &para[2]);

    setTskPara(PRIORITY, 0, &para[3]);
    setTskPara(EXETIME, 100, &para[3]);
    setTskPara(ARRTIME, 0, &para[3]);

    createTsk(test1, &para[0]);
    createTsk(test2, &para[1]);
    createTsk(test3, &para[2]);
    createTsk(test4, &para[3]);
}

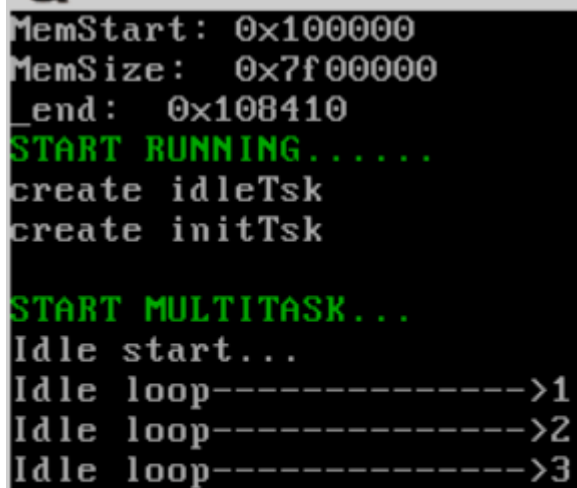
```

其余两个可在对应的测试文件中查看。

2. 运行结果分析

!!!改用不同调度器的方法为在终端输入 `./source2img.sh testxxx` (xxx可为FCFS,RR,SJF)

(1) RR结果



```

MemStart: 0x100000
MemSize: 0x7f00000
_end: 0x108410
START RUNNING.....
create idleTsk
create initTsk

START MULTITASK...
Idle start...
Idle loop----->1
Idle loop----->2
Idle loop----->3

```

```

test1----->loop:1
test1----->loop:2
test2----->loop:0
test2----->loop:1
test2----->loop:2
test3----->loop:0
test3----->loop:1
test4----->loop:0
test4----->loop:1
test4----->loop:2
test1----->loop:3
test1----->loop:4
test1----->loop:5
test2----->loop:3
test2----->loop:4
test2----->loop:5
test3----->loop:2
test3----->loop:3
test3----->loop:4
test4----->loop:3
test4----->loop:4
test1----->loop:6
test1----->loop:7

```

```

Idle loop----->16
Idle loop----->17
Idle loop----->18
Idle loop----->19
Idle loop----->20
Idle loop----->21
Idle loop----->22
Idle loop----->23
Idle loop----->24
Idle loop----->25
Idle loop----->26
Idle loop----->27
Idle loop----->28
Idle loop----->29
Idle loop----->30
Idle loop----->31
Idle loop----->32
Idle loop----->33
Idle loop----->34
Idle loop----->35
Idle loop----->36
Idle loop----->37
Idle loop----->38

```

Student >:

Unknown interrupt1

00:00:35

如图前面输出一些调试信息，表示系统正常启动，因为第一个任务达到时间为5秒，则在初始任务完成后会运行idle任务。到第5秒时，开始运行第一个任务，每个任务不能在一个时间片运行完毕，则会切换下一个任务运行，所以出现了图中任务1，2，3，4交替运行的情况。

设置shell为30s时启动，则在第30s开始后进入shell交互程序。

(2) SJF结果

```

test1----->loop:2
test1----->loop:3
test1----->loop:4
test1----->loop:5
test1----->loop:6
test1----->loop:7
test1----->loop:8
test1----->loop:9
test4----->loop:0
test4----->loop:1
test4----->loop:2
test4----->loop:3
test4----->loop:4
test4----->loop:5
test4----->loop:6
test4----->loop:7
test4----->loop:8
test4----->loop:9
test3----->loop:0
test3----->loop:1
test3----->loop:2
test3----->loop:3
test3----->loop:4
Unknown interrupt1 00:00:04

```

由于四个任务的执行时间分别为100, 300, 200, 100。则任务的运行次序为1, 4, 3, 2。符合预期。

(3) FCFS结果

```

Idle loop----->10
test1----->loop:0
test1----->loop:1
test1----->loop:2
test1----->loop:3
test1----->loop:4
test1----->loop:5
test1----->loop:6
test1----->loop:7
test1----->loop:8
test1----->loop:9
Idle start...
Idle loop----->1
Idle loop----->2
Idle loop----->3
Idle loop----->4
Idle loop----->5
Idle loop----->6
test2----->loop:0
test2----->loop:1
test2----->loop:2
test2----->loop:3
test2----->loop:4
Unknown interrupt1 00:00:11

```

四个任务的到达时间为5, 10, 15, 20, 则运行顺序为1, 2, 3, 4。符合预期。