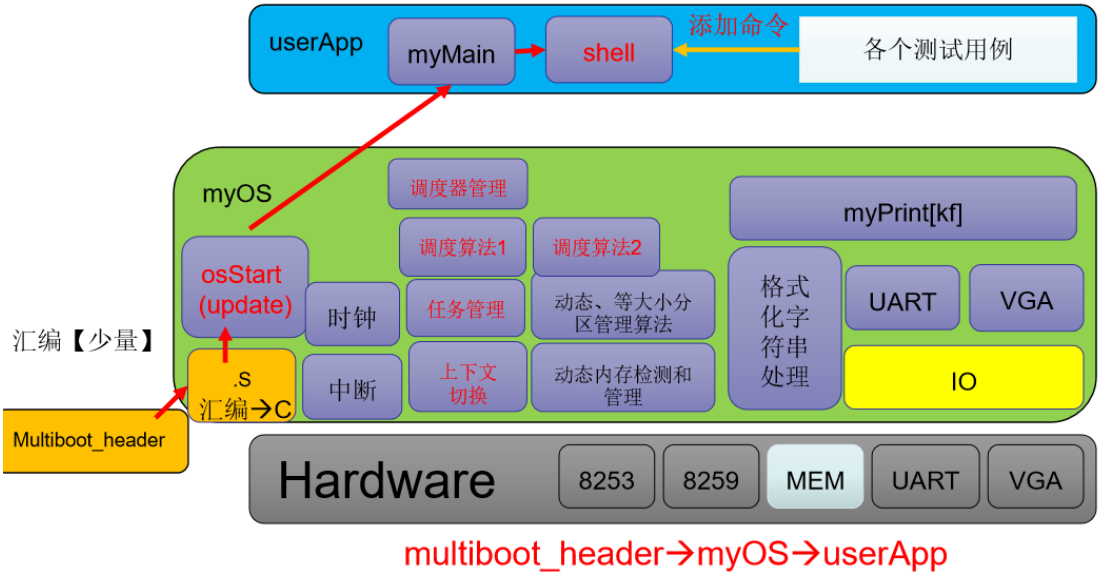
# Lab6 实验报告

PB18051098 徐碧涵

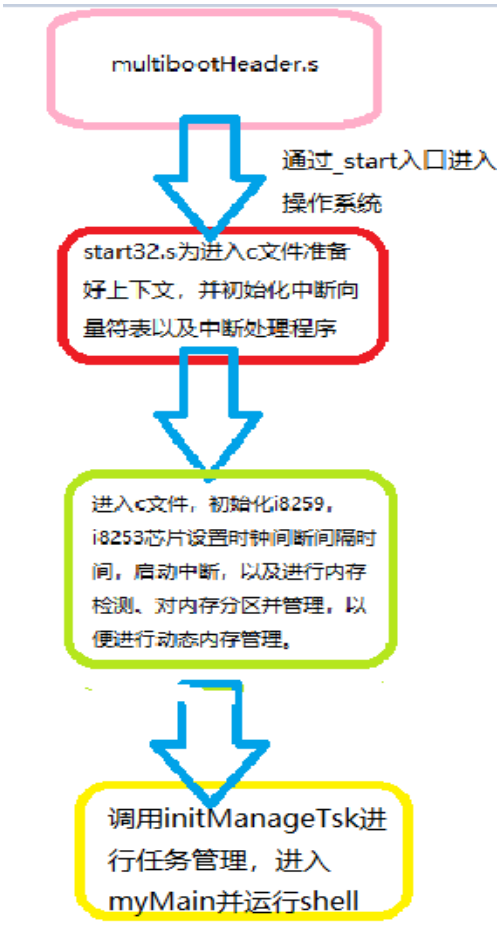
## 软件框图




本实验旨在实现提供多个调度算法用于任务管理，并提供统一调度接口，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。根据 myMain 的对调度算法的需求选择相应的调度算法并通过统一调度接口进行调度。

## 主要功能模块及其实现

### 本实验实现了 FCFS 调度算法，SJF 调度算法，PRIORITY0 调度算法

#### 主要功能模块

**任务数据结构 myTCB**

**链表管理**

*双向链表*

```
typedef struct dLink_node{
    struct dLink_node * prev;
    struct dLink_node * next;
} dLink_node;
```

*链表初始化*

```
void dLinkNodeInit(dLink_node *node){
    node->prev = NULL;
    node->next = NULL;
}
```

*返回第一个节点*

```
dLink_node *dLinkListFirstNode(dLinkedList *list){
    return list->next;  //may be the head
}
```

*插入节点*

```
 * 在某节点之前插入新节点
 */
void dLinkInsertBefore(dLinkedList *list, dLink_node *old, dLink_node *toBeInserted){
    toBeInserted->next = old;
    toBeInserted->prev = old->prev;

    old->prev->next = toBeInserted;
    old->prev = toBeInserted;
}
```

*删除节点*

```
//删除某节点
void dLinkDelete(dLinkedList *list, dLink_node *toBeDeleted){
    dLink_node *before, *after;

    if(toBeDeleted == list) return;  //never delete head

    before = toBeDeleted->prev;
    after = toBeDeleted->next;

    before->next = after;
    after->prev =before;
}
```

对到达需要一定时间的任务进行管理

*等待到达任务队列管理*

```
typedef struct arrivingNode{
    /* node should be here*/
    dLink_node theNode;

    /* node body */
    unsigned int arrTime; //ralative to arrTimeBase
    myTCB* theTCB;
}arrNode;
```

*入队*

```
/* arrTime: small --> big */
void ArrListEnqueue(myTCB* tsk){
    //根据tsk新建一个节点 按照arrTime小到大的顺序插入到链表的对应位置
    //同时arrPool也要有tsk
    arrNode * allocate=tcb2Arr(tsk);
    arrNode *p;
    p=(arrNode*)dLinkListFirstNode(&arrList);
    allocate->arrTime=tsk->para.arrTime;
    allocate->theTCB=tsk;
    //按到达顺序先后插入arrList
    while(allocate->arrTime>p->arrTime&&(dLinkedList *)p!=&arrList){
        p=(arrNode *)p->theNode.next;
    }
    dLinkInsertBefore(&arrList,(dLink_node *)p,(dLink_node*)allocate);

}
```

***Tick-hook，每个时钟 tick 时判断是否到了到达时间，到了到达时间时插入就绪队列***

```
//每次tick调用判断是否达到到达时间，然后加入就绪队列
void tick_hook_arr(void){
    if (dLinkListIsEmpty(&arrList)) return;
    else {
        arrNode *tmp = (arrNode*)dLinkListFirstNode(&arrList);
        while(tmp->arrTime + arrTimeBase <= getTick() ){
            dLinkDeleteFifo(&arrList);

            tskStart(tmp->theTCB);

            if (dLinkListIsEmpty(&arrList)) break;
            tmp = (arrNode*)dLinkListFirstNode(&arrList);
        }
    }
}
```

**FCFS 调度算法**

***FCFS 队列管理***

```
myTCB *rqFIFO;  //ready queue

void rqFIFOInit(void) {
    rqFIFO = idleTsk;   // head <- idleTsk
    dLinkListInit((dLinkedList *)(&(rqFIFO->thisNode)));
}
```

*入队操作*

```
void dLinkInsertFifoBeforeTail(dLinkedList *list, dLink_node *toBeInserted){
    /* insert before tail */
    dLinkInsertBefore(list, list->prev, toBeInserted);
}
```

```
void tskEnqueueFIFO(myTCB *tsk){
    dLinkInsertFifo((dLinkedList*)rqFIFO,(dLink_node*)tsk);
}
```

*出队操作*

```
void dLinkInsertFifo(dLinkedList *list, dLink_node *toBeInserted){
    /* insert as tail */
    dLinkInsertBefore(list, (dLink_node*)list, toBeInserted);
}
```

```
void tskDequeueFIFO(myTCB *tsk){
    dLinkDelete((dLinkedList*)rqFIFO,(dLink_node*)tsk);
}
```

*返回即将调度的任务节点*

```
myTCB * nextFIFOTsk(void) {
    return (myTCB*)dLinkListFirstNode((dLinkedList*)rqFIFO);
}
```

***FCFS 调度管理器***

```
void schedulerInit_FCFS(void){
    rqFIFOInit();

    /* default for all task except idleTsk*/
    defaultTskPara.schedPolicy = SCHED_FCFS;

    /* special for idleTsk*/
    _setTskPara(idleTsk,&defaultTskPara);
    idleTsk->para.schedPolicy = SCHED_IDLE;
}
```

```
struct scheduler scheduler_FCFS = {
    .type = SCHEDULER_FCFS,
    .nextTsk_func = nextFIFOTsk,
    .enqueueTsk_func = tskEnqueueFIFO,
    .dequeueTsk_func = tskDequeueFIFO,
    .schedulerInit_func = schedulerInit_FCFS,
    .createTsk_hook = NULL,
    .tick_hook = NULL
};
```

**SJF 调度算法**

***SJF 队列管理***

```
myTCB *rqSJF;  //ready queue

void rqSJFInit(void) {
    rqSJF = idleTsk;    // head <- idleTsk
    dLinkListInit((dLinkedList *)(&(rqSJF->thisNode)));
}
```

*入队操作*

```
void tskEnqueueSJF(myTCB *tsk){
    myTCB *p = nextSJFTsk();
    //按运行时间大小插入队列，运行时间越短越早执行
    while(p->para.exeTime<tsk->para.exeTime&&p!=rqSJF){
        p=(myTCB *)p->thisNode.next;
    }
    dLinkInsertBefore((dLinkedList*)rqSJF,(dLink_node*)p,(dLink_node*)tsk);
}
```

*出队操作*

```
void tskDequeueSJF(myTCB *tsk){
    dLinkDelete((dLinkedList*)rqSJF,(dLink_node*)tsk);
}
```

*返回即将调度的任务节点*

```
myTCB * nextSJFTsk(void) {
    return (myTCB*)dLinkListFirstNode((dLinkedList*)rqSJF);
}
```

*根据任务的运行时间，在任务运行完相应的任务时间后进行调度*

```
//运行时间结束进行调度
void Timer_tick_hook(void){
    if(--currentTsk->para.exeTime==0){
        tskEnd();
    }
}
```

*SJF 调度管理器*

```
void schedulerInit_SJF(void){
    rqSJFInit();

    /* default for all task except idleTsk*/
    defaultTskPara.schedPolicy = SCHED_SJF;

    /* special for idleTsk*/
    _setTskPara(idleTsk,&defaultTskPara);
    idleTsk->para.schedPolicy = SCHED_IDLE;
}
```

```
struct scheduler scheduler_SJF = {
    .type = SCHEDULER_SJF,
    .nextTsk_func = nextSJFTsk,
    .enqueueTsk_func = tskEnqueueSJF,
    .dequeueTsk_func = tskDequeueSJF,
    .schedulerInit_func = schedulerInit_SJF,
    .createTsk_hook = NULL,
    .tick_hook = Timer_tick_hook       //
};
```

**PRIORITY0 调度算法**

*PRIORITY0 队列管理*

```
myTCB *rqPRIO;  //ready queue

void rqPRIO0Init(void) {
    rqPRIO = idleTsk;   // head <- idleTsk
    dLinkListInit((dLinkedList *)(&(rqPRIO->thisNode)));
}
```

*入队操作*

```
void tskEnqueuePRIO0(myTCB *tsk){
    myTCB *p = nextPRIO0Tsk();
    //按优先级顺序插入rqPRIO队列
    while(p->para.priority>tsk->para.priority&&p!=rqPRIO){   //优先级数越大优先级越高
        p=(myTCB *)p->thisNode.next;
    }
    dLinkInsertBefore((dLinkedList*)rqPRIO,(dLink_node*)p,(dLink_node*)tsk);


}
```

*出队操作*

```
void tskDequeuePRIO0(myTCB *tsk){
    dLinkDelete((dLinkedList*)rqPRIO,(dLink_node*)tsk);
}
```

*返回即将调度的任务节点*

```
myTCB * nextPRIO0Tsk(void) {
    return (myTCB*)dLinkListFirstNode((dLinkedList*)rqPRIO);
}
```

***PRIORITY0 调度管理器***

```
void schedulerInit_PRIO0(void){
    rqPRIO0Init();

    /* default for all task except idleTsk*/
    defaultTskPara.schedPolicy = SCHED_PRIO;

    /* special for idleTsk*/
    _setTskPara(idleTsk,&defaultTskPara);
    idleTsk->para.schedPolicy = SCHED_IDLE;
}
```

```
struct scheduler scheduler_PRIO0 = {
    .type = SCHEDULER_PRIORITY0,
    .nextTsk_func = nextPRIO0Tsk,
    .enqueueTsk_func = tskEnqueuePRIO0,
    .dequeueTsk_func = tskDequeuePRIO0,
    .schedulerInit_func = schedulerInit_PRIO0,
    .createTsk_hook = NULL,
    .tick_hook = NULL
};
```

提供统一调度接口

*设置当前使用的调度算法*

```
void setSysScheduler(unsigned int what){

    //设置*sysScheduler使用最上面列举的哪种调度器
    if(what==SCHEDULER_PRIORITY0)
        sysScheduler=&scheduler_PRIO0;
    else if(what==SCHEDULER_SJF)
        sysScheduler=&scheduler_SJF;
    else sysScheduler = &scheduler_FCFS;
}
```

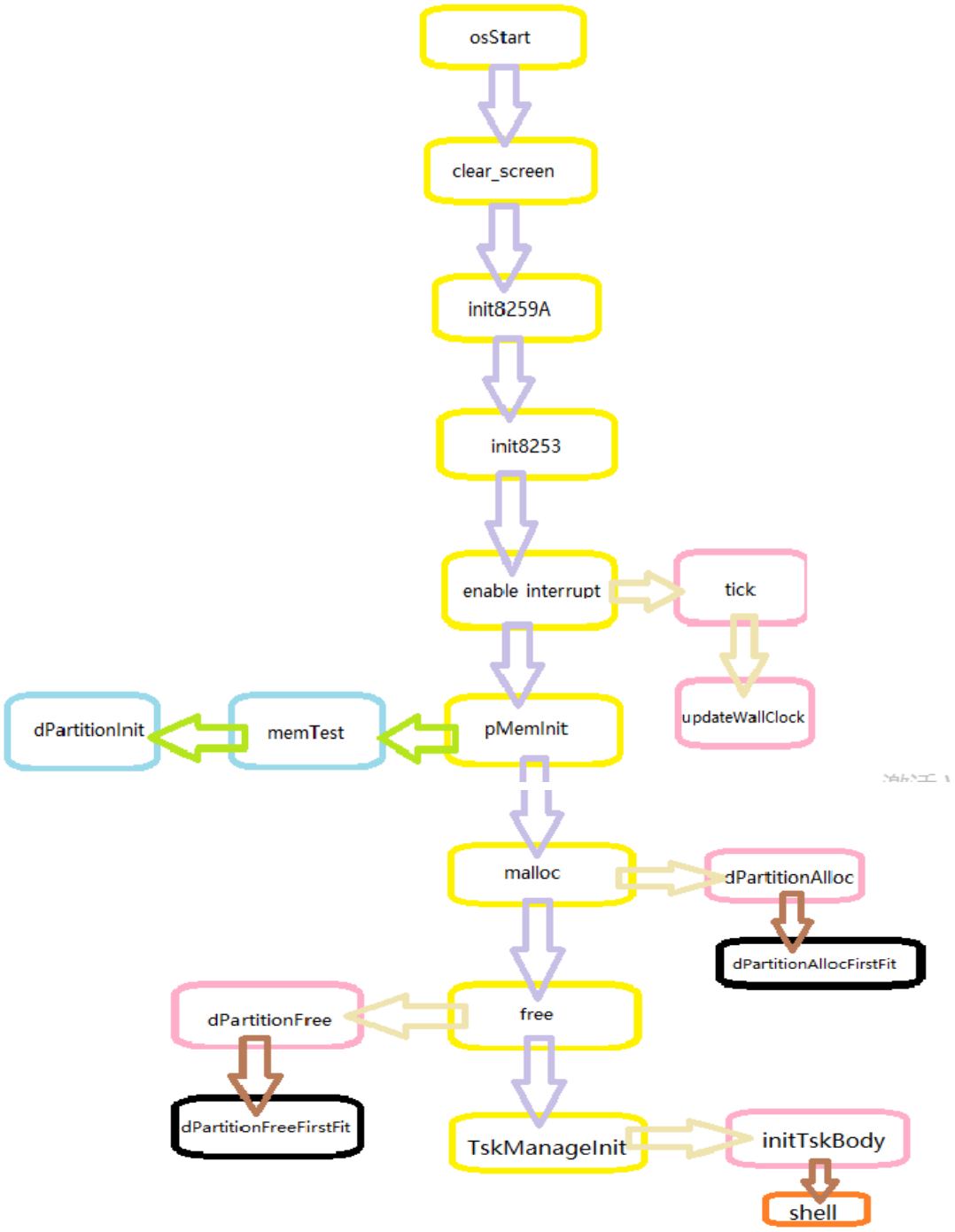
*提供接口*

```
myTCB *nextTsk(void){
    if(sysScheduler->nextTsk_func)
        sysScheduler->nextTsk_func();
}

void enqueueTsk(myTCB *tsk){
    if(sysScheduler->enqueueTsk_func)
        sysScheduler->enqueueTsk_func(tsk);
}

void dequeueTsk(myTCB *tsk){
    if(sysScheduler->dequeueTsk_func)
        sysScheduler->dequeueTsk_func(tsk);
}

void createTsk_hook(myTCB *created){
    if(sysScheduler->createTsk_hook)
        sysScheduler->createTsk_hook(created);
}

extern void scheduler_hook_main(void);

void schedulerInit(void){
   scheduler_hook_main();    //设置SysScheduler的调度算法
   if(sysScheduler->schedulerInit_func)
        sysScheduler->schedulerInit_func();
}
```

*调度*

```
void schedule(void){
    static int idle_times=0;
    myTCB * prevTsk;
    disable_interrupt();

    prevTsk=currentTsk;
    currentTsk=nextTsk();
    if(prevTsk==idleTsk&&currentTsk==idleTsk);
    else context_switch(prevTsk,currentTsk);

    enable_interrupt();
}
```

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
  - task_arr.c
  - task_sched.c
  - taskPara.c
- tick.c
- WallClock.c
- Makefile
- task_sched
  - task_fifo.c
  - task_sjf.c
  - task_prio0.c
  - Makefile

lib
- string.c
- dLinkList.c
- Makefile

Makefile

myOS.ld
osStart.c
start32.s
userApp
main.c
shell.c
Makefile
tests
test0_fcfs
test1_prio0
test2_sjf
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

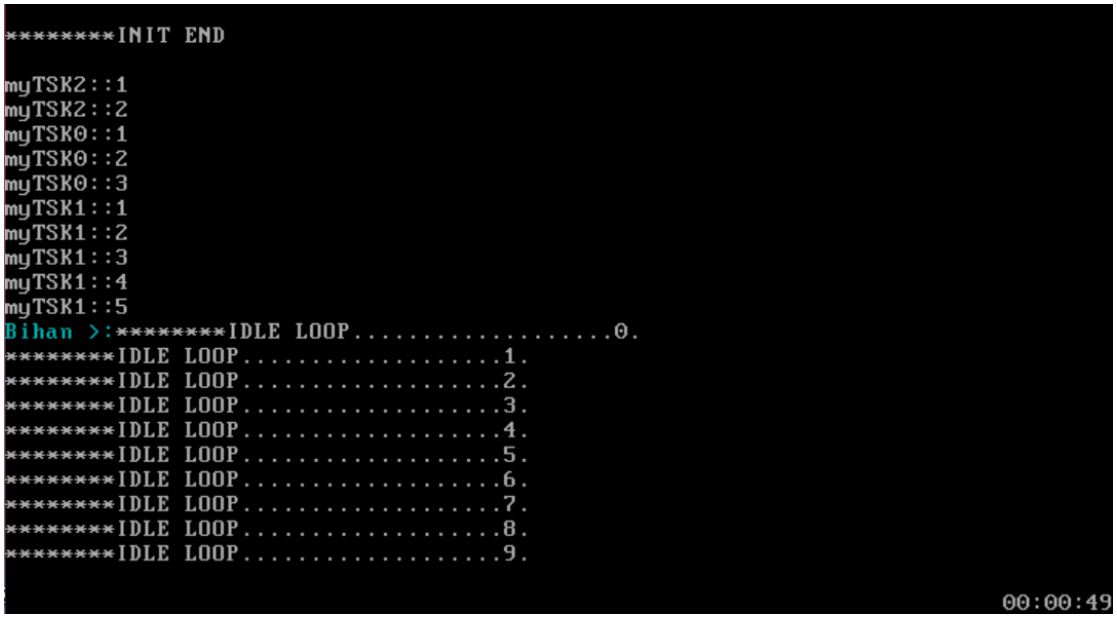
输入./source2img.sh test0_fcfs 命令运行脚本文件，编译后进而执行.elf 文件，运行结果如下：


测试给出参数如下：

```
        setTskPara(ARRTIME,50,&tskParas[0]);
        createTsk(myTSK0,&tskParas[0]);

        setTskPara(ARRTIME,100,&tskParas[1]);
        createTsk(myTSK1,&tskParas[1]);

        setTskPara(ARRTIME,0,&tskParas[2]);
        createTsk(myTSK2,&tskParas[2]);

        initShell();
memTestCaseInit();
        setTskPara(ARRTIME,120,&tskParas[3]);//
        createTsk(startShell,&tskParas[3]);  //    startShell();
```

任务按照 2,0,1,3 的顺序到达，执行顺序与之相符。

输入./source2img.sh  test1_prio0 命令运行脚本文件，编译后进而执行.elf 文件，运行结果如下：




设置参数如下：

```
        setTskPara(ARRTIME,0,&tskParas[0]);
        setTskPara(EXETIME,20,&tskParas[0]);
        createTsk(myTSK0,&tskParas[0]);

        setTskPara(ARRTIME,0,&tskParas[1]);
        setTskPara(EXETIME,30,&tskParas[1]);
        createTsk(myTSK1,&tskParas[1]);

        setTskPara(ARRTIME,0,&tskParas[2]);
        setTskPara(EXETIME,10,&tskParas[2]);
        createTsk(myTSK2,&tskParas[2]);

        initShell();
memTestCaseInit();
        setTskPara(ARRTIME,0,&tskParas[3]);//xiugai120
        setTskPara(EXETIME,1200,&tskParas[3]);
        createTsk(startShell,&tskParas[3]);  //    startShell();
```

任务执行时间按照从小到大的顺序为：2,0,1,3，且运行相应的运行时间后强行中断任务，运行下一个任务。执行结果符合。

输入./source2img.sh test2_sjf 命令运行脚本文件，编译后进而执行.elf 文件，运行结果如下：


设置参数如下：

```
        setTskPara(ARRTIME,0,&tskParas[0]);
        setTskPara(PRIORITY,4,&tskParas[0]);
        createTsk(myTSK0,&tskParas[0]);

        setTskPara(ARRTIME,0,&tskParas[1]);
        setTskPara(PRIORITY,2,&tskParas[1]);
        createTsk(myTSK1,&tskParas[1]);

        setTskPara(ARRTIME,0,&tskParas[2]);
        setTskPara(PRIORITY,3,&tskParas[2]);
        createTsk(myTSK2,&tskParas[2]);

        initShell();
memTestCaseInit();
        setTskPara(ARRTIME,0,&tskParas[3]);//xiugai120
        setTskPara(PRIORITY,1,&tskParas[3]);
        createTsk(startShell,&tskParas[3]);  //    startShell();
```

优先级数越大，越优先执行，优先级从大到小顺序为：0,2,1,3；执行结果符合。