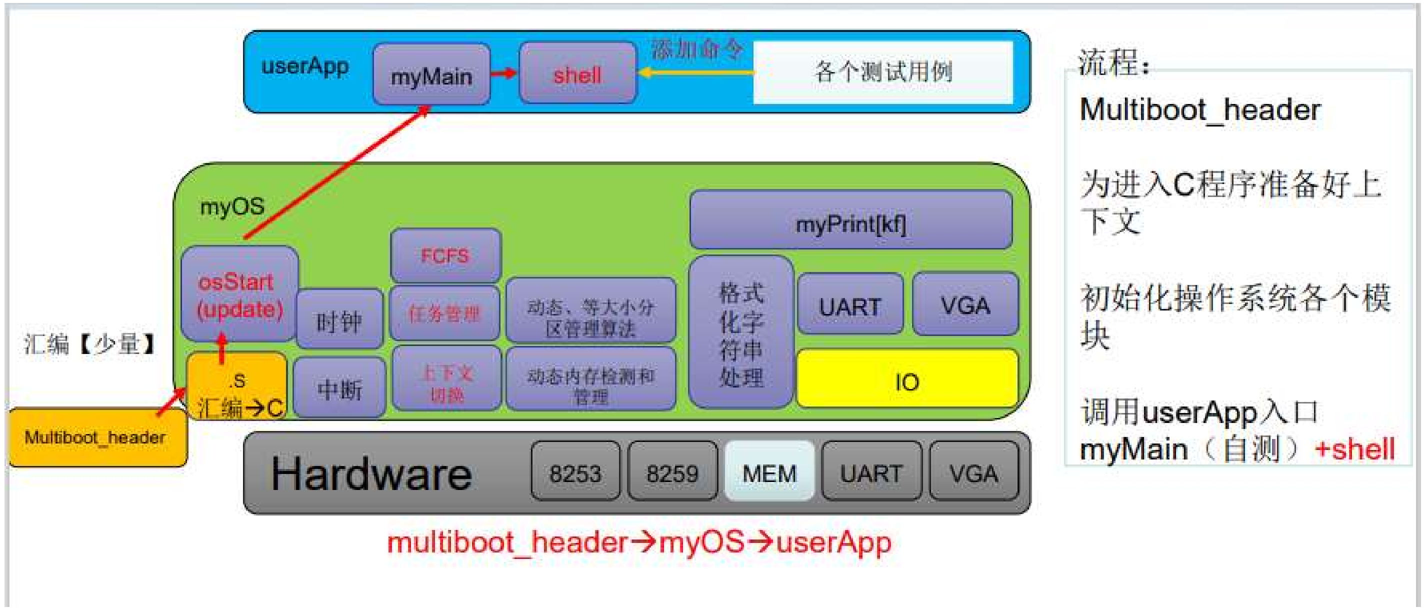
OS Lab5 & Lab6 report

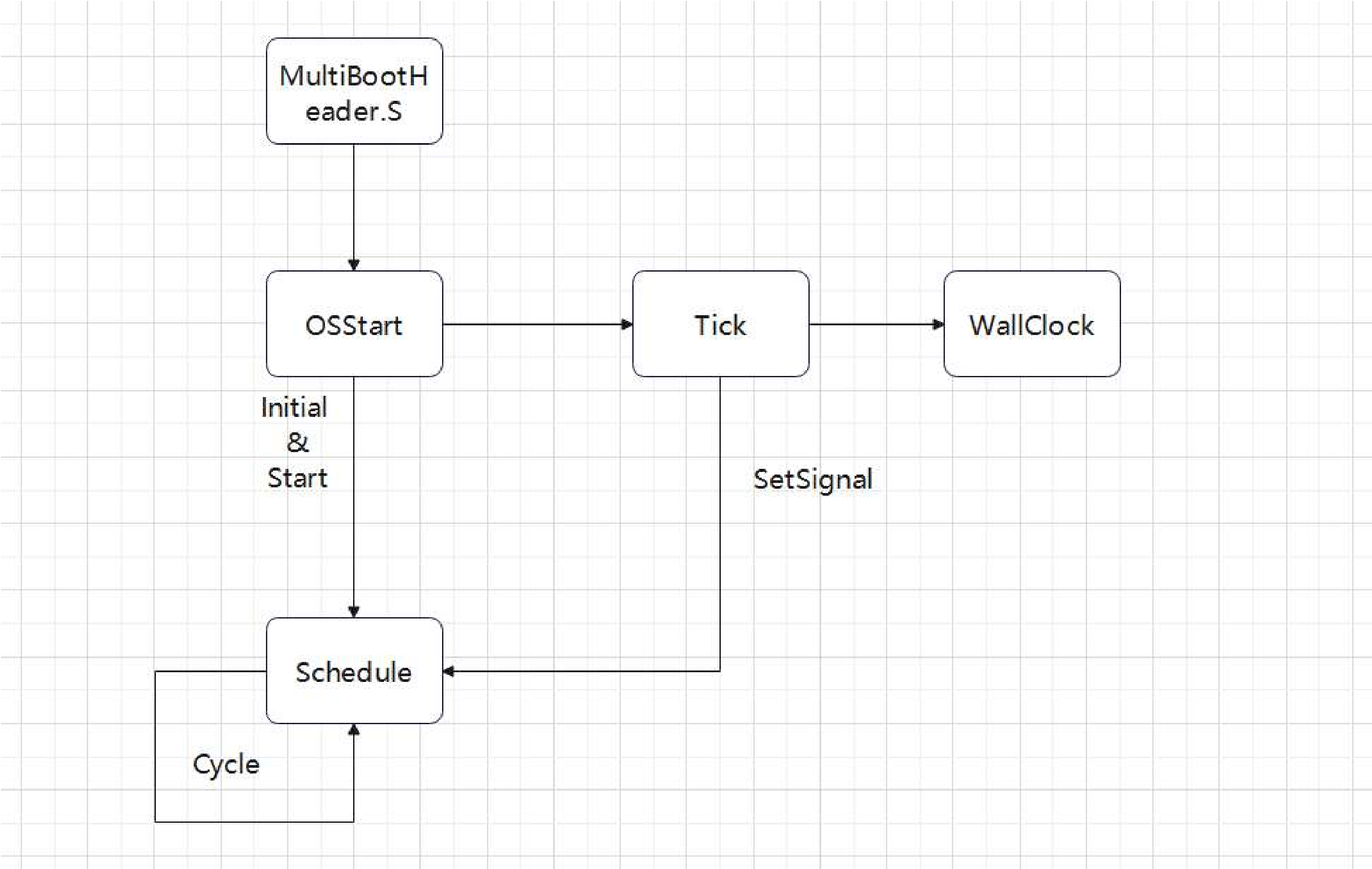
PB20000096 潘廷岳

一、实验框架

- 整体框架



- 执行流程



实现的内容

- SJF、FCFS、PRI 非抢占式调度
- 调度器与任务参数统一接口
- 采用tick动态检测ReadyQueue是否非空

二、实验代码介绍

- 关于启动牵引之类的的不再过多介绍

- 以下只介绍核心数据结构及功能函数

①核心数据结构

- Task相关
  - 采用五状态模型
  - TCB存储内容见注释

```c
typedef enum State {
    WAITING, //暂未使用
    READY,
    RUNNING,
    NEW,
    TERMINATED //暂未使用
} State;

// Task的属性
typedef struct tskPara {
    unsigned priority; //优先级（PRI）
    unsigned arrTime; //到达时间（未使用）
    unsigned exeTime; //执行时间(SJF)
} tskPara;

typedef struct TCB {
    int tid; // 进程ID
    State state;
    unsigned long* sp; // stack point
    unsigned long stk_top; // 分配空间首地址，也是栈底
    tskPara params; //parameter list
} TCB;

#define NULL_TCB ((TCB){.tid = -1, .state = READY, .sp = 0, .params = (tskPara){0, 0, 0}})

//空结构，便于曹祖



// NewQueue以链表形式组织管理
typedef struct NewTask_list {
    TCB data;                    // TCB_value
    struct NewTask_list* next;
} NewTask_list;                 //NewTask_Set
```

- Scheduler
    - 主要保存用何方法调度

```
typedef enum ScheduleWay {
    FCFS,
    SJF,
    PRI
}ScheduleWay;
```

- 各调度方法对应的队列

```
//SJF 队列结构（采用单向链表组织）
typedef struct ReadyQueueSJF {
    TCB data; //保存的TCB
    struct ReadyQueueSJF* next; // 下一队内元素
    int nowsize; //队列当前大小
    int maxsize; //可容纳的最大大小
} ReadyQueueSJF;


//PRI 队列结构（采用单向链表组织）
typedef struct ReadyQueuePRI {
    TCB data;
    struct ReadyQueuePRI* next;
    int nowsize;
    int maxsize;
} ReadyQueuePRI;


//FIFO 队列结构（采用循环队列）
typedef struct ReadyQueueFIFO {
    TCB* data;
    unsigned long head, tail;
    unsigned long maxsize;
    unsigned long nowsize;
} ReadyQueueFIFO;
```

②核心函数介绍

Ⅰ） void osStart()

- 最关键的函数之一，为OS的启动做准备
- 与本次实验关联最大的各函数作用见注释

```
void osStart(void) {

    disable_interrupt();
    clear_screen();
    pressAnyKeyToStart();

    clear_screen();

    // 初始化并设置时钟中断，为抢占式调度做准备
    init8259A();
    init8253();
    init_tick();
    init_wall_clock(0, 0, 0);

    //初始化Mem及相关内存管理函数
    pMemInit();

    // 开时钟中断
    enable_interrupt();

    //设置调度模式
    Set_Model();

    //初始化任务调度器，并开始调度操作
    taskManager_init();


    // myPrintk(0x2, "Starting the OS...\n");
    // myPrintk(0x2, "Stop running... shutdown\n");

    while(1);
}
```

- 下面逐个介绍上述标有注释的部分重要函数
  - init_tick()
    - 该函数为部分需绑定tick的函数做准备

```
void init_tick(void) {
    tick_times = 0;
    for (int i = 0; i < MAX_TICK_HOOK_FUNC_NUM; i++)
        hook_list[i] = NULL;
}
```

- enable_interrupt()
    - 开时钟中断，以便进行ReadyQueue的轮询
- Set_Model()
    - 设置调度模式
    - 个人认为调度操作对程序员是透明的，故将其放于OSSart.c里了
    - 如需更改调度模式，只需修改以下函数即可

```
void Set_Model() {
    SW = SJF;
    //Support SJF、FCFS、PRI
}
```

- taskManager_init();
- 初始化任务调度器，并开始调度操作

Ⅱ）taskManager_init()

- 初始化任务调度器，并开始调度操作
- 依次执行三部分
    - 初始化NewQueue与ReadyQueue
    - 创建IDLE任务并放入NewQueue中
    - 创建myMain任务，并进一步放入就绪队列中
    - //启动多道程序调度器

```c
// 为tasklist分配空间
NewTask_list_head = (NewTask_list*) kmalloc(sizeof(NewTask_list));
NewTask_list_head->next = 0;
taskReadyQueue_init();

// Create idle task
idle_id = createTsk(idleTsk);
idle_sp = NewTask_list_head->next->data.sp;


// Create and Ready --> myMain
int myMain_id = createTsk(myMain);
tskStart_By_ID(myMain_id);

//启动多道程序调度器
startMultitask();
```

- 主要代码
    - taskReadyQueue_init()
        - ReadyQueue初始化统一接口

```c
void taskReadyQueue_init() {
    switch (SW)
    {
        case FCFS:
            QInit(&ReadyQueueFF, ReadyQueue_MaxLen);
            break;

        case SJF:
            QInitSJF(ReadyQueue_MaxLen);
            break;

        case PRI:
            QInitPRI(ReadyQueue_MaxLen);
            break;

        default:
            break;
    }
}
```

    - createTsk(TCB tsk)
        - 创建新进程并加入NewQueue
        - 函数返回新建进程的ID

```c
int createTsk(void (*tskBody)(void)) {
    TCB tcb ;
    tcb.tid = tid_count++;
    tcb.state = NEW;
    tcb.params = (tskPara) {.priority = 0, .arrTime = 0, .exeTime = 0};

    //根据预先设定的栈大小分配栈空间
    unsigned long stack_top = kmalloc(stack_size);
    tcb.stk_top = stack_top;
    tcb.sp = (unsigned long* )(stack_top + stack_size) - 1;

    stack_init(&tcb.sp, tskBody);

    // 将新建任务插入NewQueue中等待资源分配
    NewTask_list* tmp = (NewTask_list*) kmalloc(sizeof(NewTask_list));
    tmp->data = tcb;

    tmp->next = NewTask_list_head->next;
    NewTask_list_head->next = tmp;

    return tcb.tid;
}
```

- tskStart_By_ID(int id);
    - 通过进程ID进行索引，索引完成后加入就绪队列等待执行
    - 成功建立则返回真

```
Bool tskStart_By_ID(int tsk_id) {
    NewTask_list* tmp = NewTask_list_head->next;
    NewTask_list* prev = NewTask_list_head;

    //在NewQueuezhong进行索引
    for (; tmp; tmp = tmp->next) {
        if (tmp->data.tid == tsk_id) break;
        prev = tmp;
    }

    if (!tmp) return False;
    TCB tsk = tmp->data;

    // 传入待进入ReadyQueue的任务TCB
    if(tskStart(tsk) == False)
        return False; //The Ready Queue is full

    ReadyQueueEmpty = False;
    prev->next = tmp->next;

    return True;
}
```

- 其中出现的tskStart(TCB tsk)函数如下

  ```
  Bool tskStart(TCB tsk) {

      判断是否能够加入ReadyQueue中
      if(Check_ReadyQueue() == False)
          return False;

      tsk.state = READY;

      New_to_ReadyQueue(tsk);
      // 将指定任务加入TNewQueue

      return True;
  }
  ```

- New_to_ReadyQueue(TCB tsk)函数
  - ReadyQueue统一入口

```c
void New_to_ReadyQueue(TCB tsk) {
    switch (SW)
    {
        case FCFS:
            Qpush(&ReadyQueueFF, tsk);
            break;
        case SJF:
            QpushSJF(ReadyQueueSJF_head, tsk);
            break;
        case PRI:
            QpushPRI(ReadyQueuePri_head, tsk);
            break;

        default:
           break;
    }

    // manage according to FIFO

}
```

Ⅲ）startMultitask()

- 是多道调度的启动函数
  - 设置初始任务栈并进入Schedule（）

```c
void startMultitask(void) {
    BspContextBase = (unsigned long *)kmalloc(0x1000);
    BspContext = BspContextBase + 0x1000 - 1; ///

    //firstTsk = nextFCFSTsk();
    schedule();
}
```

- schedule()
  - 调度器工作函数

```c
void schedule(void){
    while (1){
        TCB NextTask;
        if (ReadyQueueEmpty == True) {
            tskStart_By_ID(idle_id);
        } else {
            ;
        }
        // 动态调度关键部分：ReadyQueueEmpty信号量
        // 将ReadyQueueEmpty信号量的更新与tick挂钩，能够实现动态检测ReadyQueue的状态（是否应当执行IDLE任务）


        // display_ReadyQueue();
        switch (SW)
        {
            case FCFS:
                NextTask = Qpop(&ReadyQueueFF);
                break;
            case SJF:
                NextTask = QpopSJF();
                break;
            case PRI:
                NextTask = QpopPRI();
                break;

            default:
                break;
        }

        NextTask.state = RUNNING;
        current_tsk = NextTask;

        // 上下文切换
        context_switch(&BspContext, current_tsk.sp);
    }
}
```

- 调度核心工作机制相关函数说明
    - HookInit()
        - 用于注册Hook函数

        ```c
        void HookInit() {
            append2hook(Check_and_Set_ReadyQueueState);
        }
        ```

- CheCheck_and_Set_ReadyQueueState()
  - 用于动态更新ReadyQueueEmpty信号量

```
void Check_and_Set_ReadyQueueState() {

    switch (SW)
    {
        case FCFS:
                if(Qempty(&ReadyQueueFF) == True) ReadyQueueEmpty = True;
                else ReadyQueueEmpty = False;
            break;
        case SJF:
                if(QemptySJF(ReadyQueueSJF_head) == True) ReadyQueueEmpty = True;
                else ReadyQueueEmpty = False;
            break;
        case PRI:
                if(QemptyPRI(ReadyQueuePri_head) == True) ReadyQueueEmpty = True;
                else ReadyQueueEmpty = False;
            break;
        default:
            break;
    }

}
```

- 进程切换、销毁相关原语
  - 进程切换：

```
void context_switch(unsigned long **prevTskStkAddr,
 unsigned long *nextTskStk) {
    prevTSK_StackPtrAddr = prevTskStkAddr;
    nextTSK_StackPtr = nextTskStk;
    CTX_SW();
}
```

  - 进程销毁：tskEnd() & destroyTsk()
    - 压入栈中执行

```c
void tskEnd() {
    destroyTsk(current_tsk.stk_top);
    context_switch(&current_tsk.sp,BspContext);
    schedule();
    return;
}

void destroyTsk(int tsk_stack_top) {
    kfree(tsk_stack_top);
}
```

IV）各调度算法相关函数

- 由于函数作用基本一致，这里只介绍SJF

```c
// SJF队列管理头节点
extern ReadyQueueSJF *ReadyQueueSJF_head;

// 初始化SJF
extern void QInitSJF(int ReadyQueue_MaxLen);

//打印SJF队列各元素相关信息（调试用）
extern void display_ReadyQueueSJF(ReadyQueueSJF* queue);

//入队
extern Bool QpushSJF(ReadyQueueSJF* queue, TCB tcb);

//判断是否为空队列
extern Bool QemptySJF(const ReadyQueueSJF* queue);

//出队
extern TCB QpopSJF();

//获取当前队列长度
extern unsigned long QlenSJF(const ReadyQueueSJF* queue);
```

- 具体函数内容如下所示

```c
void QInitSJF(int ReadyQueue_MaxLen) {
    ReadyQueueSJF_head = (ReadyQueueSJF* )kmalloc(sizeof(ReadyQueueSJF));
    ReadyQueueSJF_head->nowsize = 0;
    ReadyQueueSJF_head->next = 0;
    ReadyQueueSJF_head->maxsize = ReadyQueue_MaxLen;
    myPrintf(0x7,"InitQueueMaxsize:%d\n",ReadyQueueSJF_head->maxsize);
} // 开辟空间并初始化链头管理节点

Bool QpushSJF(QUEUE* queue, TCB tcb) {

    ReadyQueueSJF* prev = ReadyQueueSJF_head;

    while(prev->next != 0 && prev->next->data.params.exeTime <= tcb.params.exeTime) {
        prev = prev->next;
    }

    ReadyQueueSJF* tmp = (ReadyQueueSJF* )kmalloc(sizeof(ReadyQueueSJF));

    tmp->data = tcb;
    tmp->next = prev->next;

    prev->next = tmp;
    ReadyQueueSJF_head->nowsize += 1;

    return True;
}  //入队函数


TCB QpopSJF() {
    if(!ReadyQueueSJF_head->nowsize)
        return NULL_TCB;

    ReadyQueueSJF* Next = ReadyQueueSJF_head->next;
    TCB tmp = Next->data;
    ReadyQueueSJF_head->next = Next->next;
    --(ReadyQueueSJF_head->nowsize);
    return tmp;
} // 出队函数

Bool QfullSJF(const QUEUE* queue) {
    if(queue->nowsize == queue->maxsize)
        return True;
    else
        return False;
}
```

```c
Bool QemptySJF(const QUEUE* queue) {
    if(queue->nowsize == 0)
        return True;
    else
        return False;
}

unsigned long QlenSJF(const QUEUE* queue) {
    return queue->nowsize;
}

void display_ReadyQueueSJF(ReadyQueueSJF* queue) {
    ReadyQueueSJF* tmp = queue;
    myPrintf(0x7, "\nReadyQueueSJF:\n");
    while(tmp != 0) {
        myPrintf(0x7, "tid: %d pri: %d \n", tmp->data.tid,tmp->data.params.exeTime);
        tmp = tmp->next;
    }
} //打印相关信息
```

Ⅴ）myMain()函数

- 用户函数，用于检测

```
void myMain(void) {

    myPrintf(0x7,"Into myMain\n");
    initShell();
    memTestCaseInit();

    enum {
        Pri,
        Arr,
        Exe
    };

    int shell_id = createTsk(startShell);
    int t1_id = createTsk(TaskTest_1);
    int t2_id = createTsk(TaskTest_2);
    int t3_id = createTsk(TaskTest_3);
    int t4_id = createTsk(TaskTest_4);
    int t5_id = createTsk(TaskTest_5);

    //设置任务优先级，用于优先级调度测试
    // 2 -> 3 -> 1 -> 4 -> 5
    setPara(Pri,3,t1_id);
    setPara(Pri,1,t2_id);
    setPara(Pri,2,t3_id);
    setPara(Pri,4,t4_id);
    setPara(Pri,6,t5_id);
    setPara(Pri,16,shell_id); // Lowest priority
    //display_NewQueue();

    ////设置任务所需时间，用于SJF调度测试
    // 1 -> 3 -> 2 -> 4 -> 5
    setPara(Exe,2,t1_id);
    setPara(Exe,5,t2_id);
    setPara(Exe,4,t3_id);
    setPara(Exe,7,t4_id);
    setPara(Exe,8,t5_id);
    setPara(Exe,16,shell_id); // Longest job

    tskStart_By_ID(t1_id);
    tskStart_By_ID(t2_id);
    tskStart_By_ID(t3_id);
    tskStart_By_ID(t4_id);
    tskStart_By_ID(t5_id);


    //最后启动shell
```

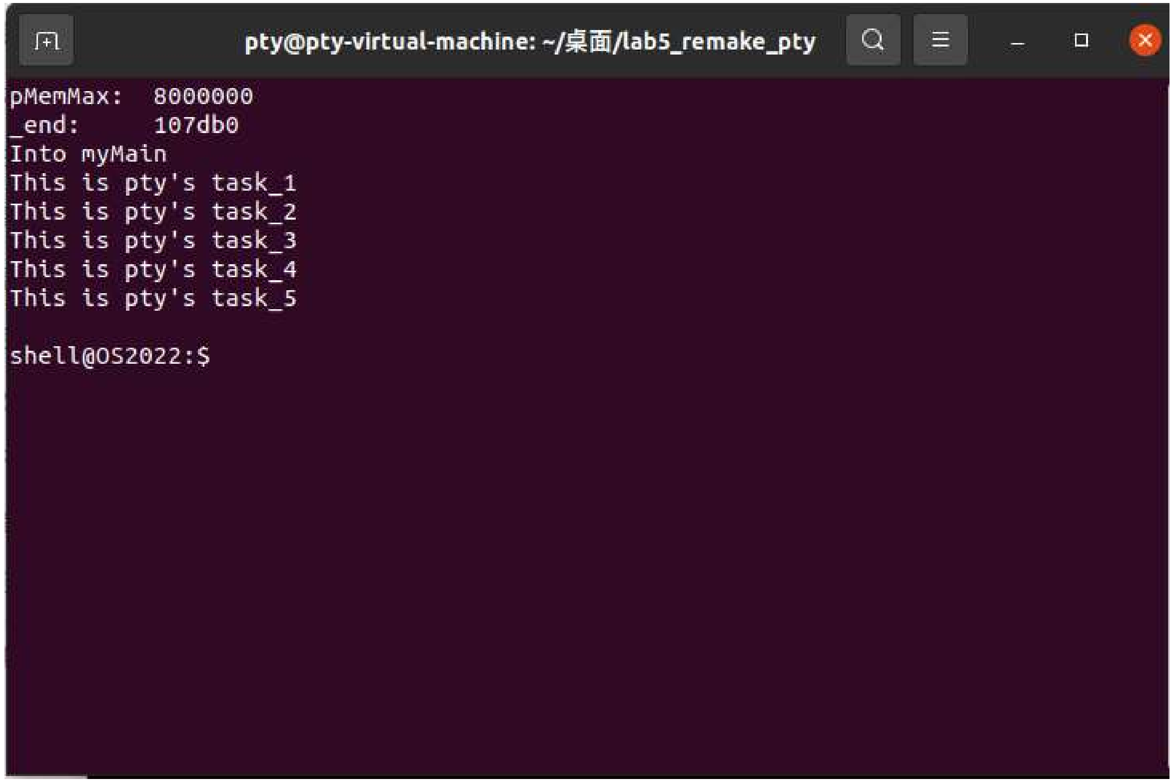
```
    tskStart_By_ID(shell_id);
}
```

三、实验结果展示

```c
// 测试用例

void TaskTest_1() {
    myPrintf(0x7, "This is pty's task_1\n");
}

void TaskTest_2() {
    myPrintf(0x7, "This is pty's task_2\n");
}

void TaskTest_3() {
    myPrintf(0x7, "This is pty's task_3\n");
}

void TaskTest_4() {
    myPrintf(0x7, "This is pty's task_4\n");
}

void TaskTest_5() {
    myPrintf(0x7, "This is pty's task_5\n");
}


int shell_id = createTsk(startShell);
int t1_id = createTsk(TaskTest_1);
int t2_id = createTsk(TaskTest_2);
int t3_id = createTsk(TaskTest_3);
int t4_id = createTsk(TaskTest_4);
int t5_id = createTsk(TaskTest_5);

//设置任务优先级，用于优先级调度测试
// 2 -> 3 -> 1 -> 4 -> 5
setPara(Pri,3,t1_id);
setPara(Pri,1,t2_id);
setPara(Pri,2,t3_id);
setPara(Pri,4,t4_id);
setPara(Pri,6,t5_id);
setPara(Pri,16,shell_id); // Lowest priority
//display_NewQueue();

////设置任务所需时间，用于SJF调度测试
// 1 -> 3 -> 2 -> 4 -> 5
setPara(Exe,2,t1_id);
setPara(Exe,5,t2_id);
setPara(Exe,4,t3_id);
setPara(Exe,7,t4_id);
```

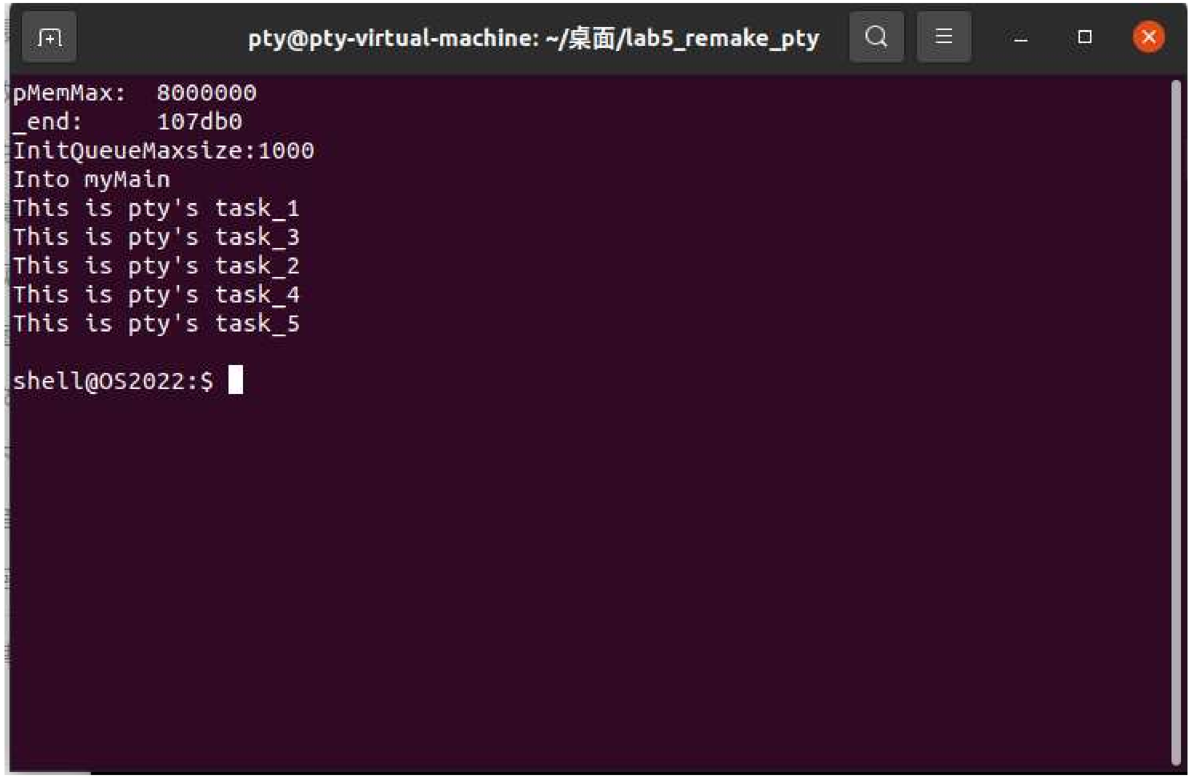
```
setPara(Exe,8,t5_id);
setPara(Exe,16,shell_id); // Longest job

tskStart_By_ID(t1_id);
tskStart_By_ID(t2_id);
tskStart_By_ID(t3_id);
tskStart_By_ID(t4_id);
tskStart_By_ID(t5_id);

//FCFS应为 1 -> 2 -> 3 -> 4 -> 5
```

①FCFS调度



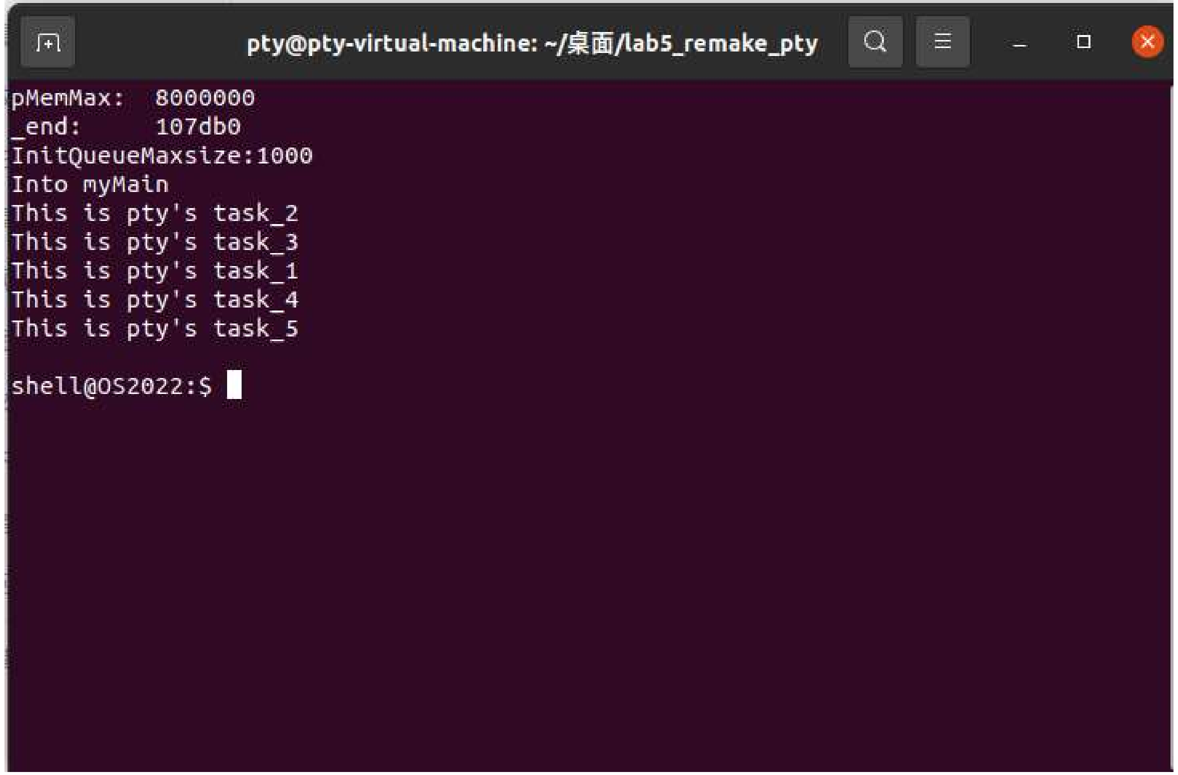
- 按顺序执行，最后启动shell
- 显然正确

②SJF调度



- 根据上述代码内参数设置，应当为 1 -> 3 -> 2 -> 4 -> 5
- 最后启动shell
- 显然正确

①PRI调度



- 根据上述代码内参数设置，应当为 2 -> 3 -> 1 -> 4 -> 5
- 最后启动shell
- 显然正确

四、文件组织

- 文件组织

```
C:.
│  Makefile
│  source2img.sh
│
├─.vscode
│      configurationCache.log
│      dryrun.log
│      settings.json
│      targets.log
│
├─multibootheader
│      multibootHeader.S
│
├─myOS
│  │  Makefile
│  │  myOS.ld
│  │  osStart.c
│  │  start32.S
│  │  userInterface.h
│  │
│  ├─dev
│  │      i8253.c
│  │      i8259A.c
│  │      Makefile
│  │      uart.c
│  │      vga.c
│  │
│  ├─i386
│  │      CTX_SW.S
│  │      io.c
│  │      irq.S
│  │      irqs.c
│  │      Makefile
│  │
│  ├─include
│  │  │  HookInit.h
│  │  │  interrupt.h
│  │  │  io.h
│  │  │  kmalloc.h
│  │  │  malloc.h
│  │  │  mem.h
│  │  │  myPrintk.h
│  │  │  string.h
│  │  │  tick.h
│  │  │  types.h
│  │  │  uart.h
```

```
│  │  │  vga.h
│  │  │  vsprintf.h
│  │  │  wallClock.h
│  │  │
│  │  └─schedule
│  │          QueueFIFO.h
│  │          QueuePRI.h
│  │          QueueSJF.h
│  │          scheduler.h
│  │          task.h
│  │
│  ├─kernel
│  │  │  Makefile
│  │  │
│  │  ├─mem
│  │  │      dPartition.c
│  │  │      eFPartition.c
│  │  │      Makefile
│  │  │      malloc.c
│  │  │      pMemInit.c
│  │  │
│  │  ├─schedule
│  │  │      Makefile
│  │  │      QueueFIFO.c
│  │  │      QueuePRI.c
│  │  │      QueueSJF.c
│  │  │      scheduler.c
│  │  │      task.c
│  │  │
│  │  └─timer
│  │          HookInit.c
│  │          Makefile
│  │          tick.c
│  │          wallClock.c
│  │
│  ├─lib
│  │      Makefile
│  │      string.c
│  │      vsprintf.c
│  │
│  └─printk
│          Makefile
│          myPrintk.c
│
├─output
│  │  myOS.elf
│  │
```

```
│   ├─multibootheader
│   │       multibootHeader.o
│   │
│   ├─myOS
│   │   │  osStart.o
│   │   │  start32.o
│   │   │
│   │   ├─dev
│   │   │       i8253.o
│   │   │       i8259A.o
│   │   │       uart.o
│   │   │       vga.o
│   │   │
│   │   ├─i386
│   │   │       CTX_SW.o
│   │   │       io.o
│   │   │       irq.o
│   │   │       irqs.o
│   │   │
│   │   ├─kernel
│   │   │   ├─mem
│   │   │   │       dPartition.o
│   │   │   │       eFPartition.o
│   │   │   │       malloc.o
│   │   │   │       pMemInit.o
│   │   │   │
│   │   │   ├─schedule
│   │   │   │       QueueFIFO.o
│   │   │   │       QueuePRI.o
│   │   │   │       QueueSJF.o
│   │   │   │       scheduler.o
│   │   │   │       task.o
│   │   │   │
│   │   │   └─timer
│   │   │            HookInit.o
│   │   │            tick.o
│   │   │            wallClock.o
│   │   │
│   │   ├─lib
│   │   │       string.o
│   │   │       vsprintf.o
│   │   │
│   │   └─printk
│   │            myPrintk.o
│   │
│   └─userApp
│           main.o
```

```
│          memTestCase.o
│          shell.o
│          taskTestCase.o
│
└─userApp
        main.c
        Makefile
        memTestCase.c
        memTestCase.h
        shell.c
        shell.h
        taskTestCase.c
        taskTestCase.h
```

- Makefile组织