实验 6： Scheduler

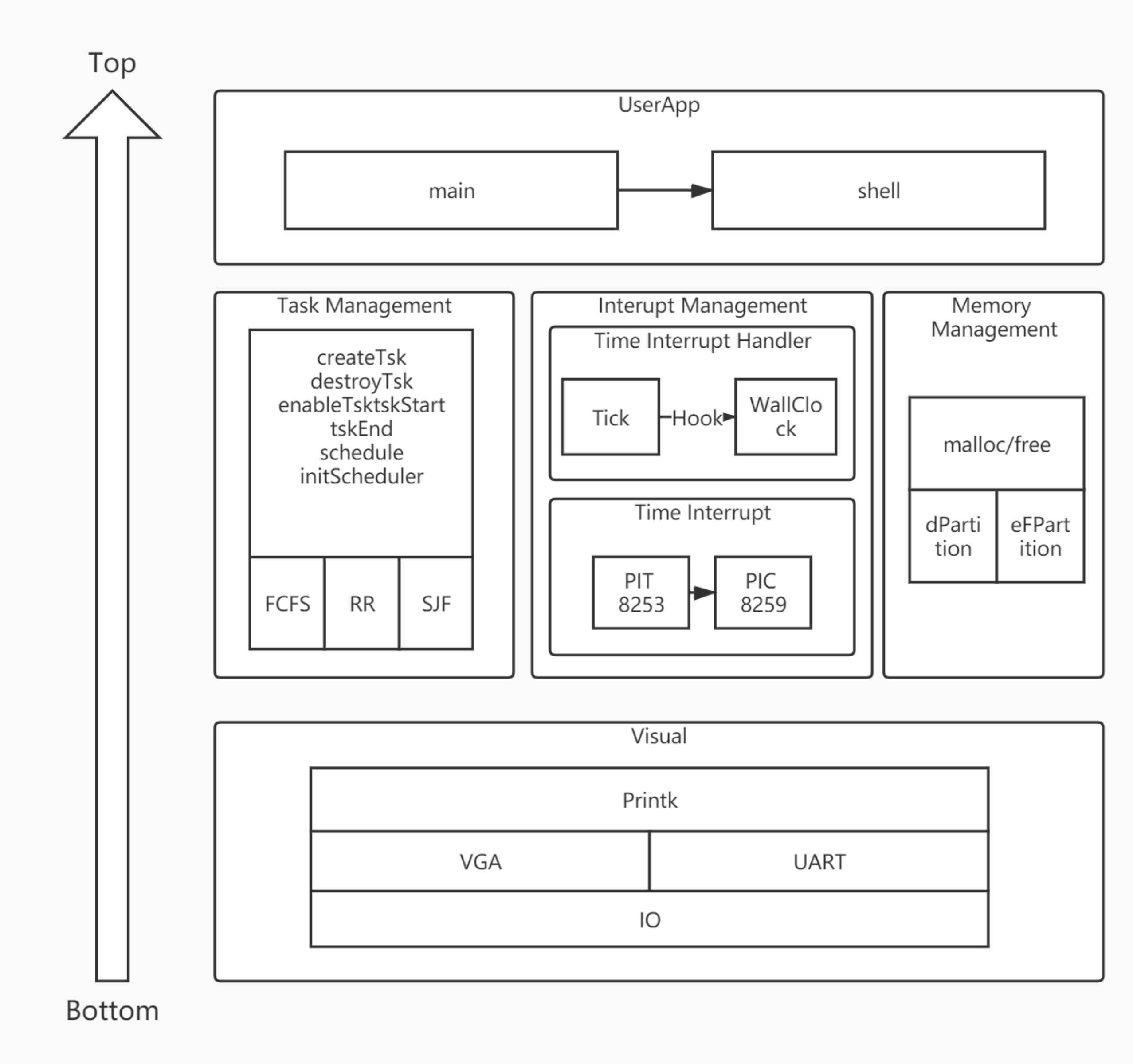
目录

软件框图

- 软件结构

  本次实验主要开发多样的任务调度功能，通过模块化方式实现FCFS、RR、SJF调度器，并提供统一的外部接口。

- 结构图（底层在下，顶层在上）：

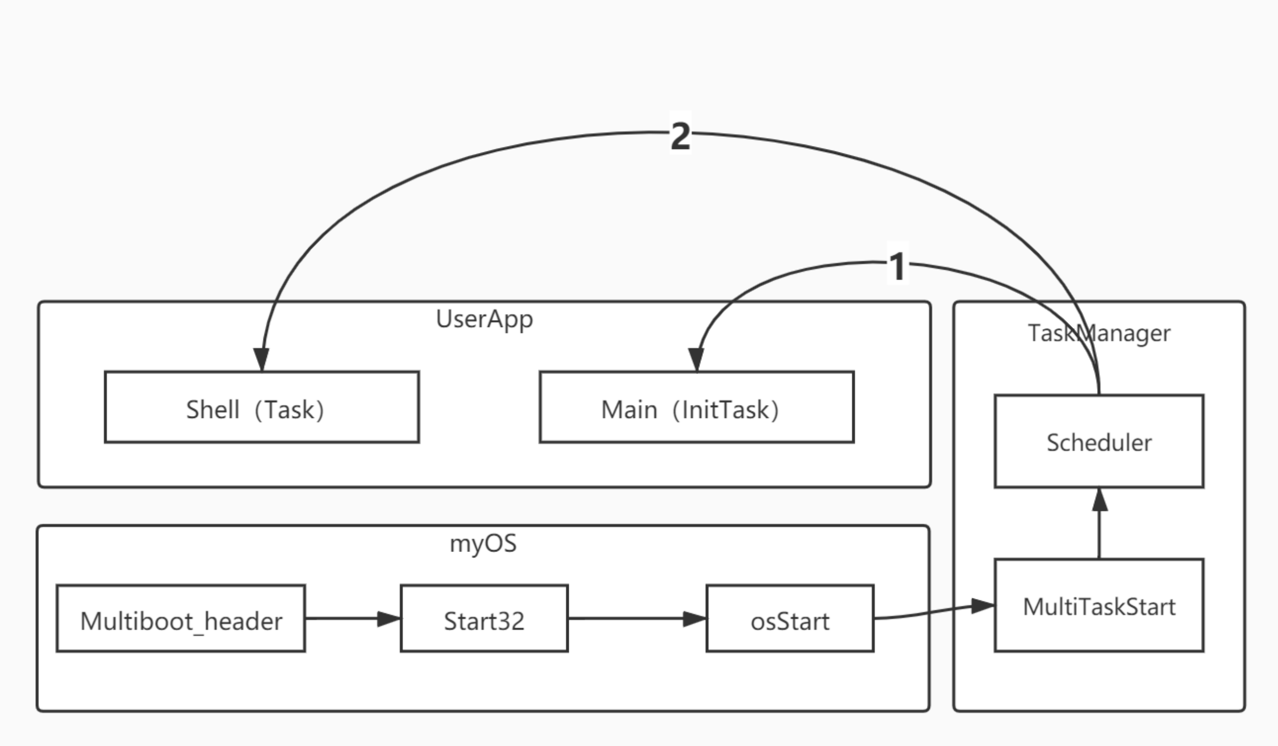


主流程

- 流程说明

主流程从Multiboot_header开始，首先进入Start32。在Start32中，程序进行了堆栈的初始化、IDT的初始化等必要的准备工作，然后将控制权移交到osStart。在osStart中，进行内存、时钟、PIT、PIC的初始化，并开启中断，之后创建任务IdleTask进行任务管理。InitTask（myMain）第一个进入就绪队列，在InitTask执行的时候，测试用例提供的任务也被加入就绪/等待队列，并被调度器管理起来。

- 流程图



功能模块

概述

软件的新增了RR、SJF调度器，相应的加入了taskPara、taskArr模块，分别对任务参数、任务到达进行管理。主要的困难是抢占调度的实现。

时间片轮转(RR)

- RR调度器模块

```
scheduler schedulerRR = {
    .nextTsk_func = nextRRTsk,
    .enqueueTsk_func = tskEnqueueRR,
    .dequeueTsk_func = tskDequeueRR,
    .schedulerInit_func = initRR,
    .schedule = scheduleRR,
    .createTsk_hook = 0,
    .tick_hook = 0
};
```

- 时间片轮转的调度算法，时间片长为 5 ticks。

```
void scheduleRR(void)
{
    while (1)
    {
        if (!rqRRIsEmpty())
```

```
        {
            // myPrintk(0xf, "RR Scheduling...\n");

            myTCB *nextTsk = nextRRTsk();

            idleTask->state = TSK_READY;
            nextTsk->state = TSK_RUNNING;
            //switch

            // myPrintk(0xf, "idle: %d...\n",idleTask);
            // myPrintk(0xf, "next: %d...\n",nextTsk);

            if(nextTsk->preempted)
            {
                context_switch_interrupt(idleTask, nextTsk);
            }
            else
            {
                context_switch(idleTask, nextTsk);
            }

            //return
            // myPrintk(0xf, "Return to IdleTask...\n");

            if (nextTsk->preempted)
            {
                nextTsk->state = TSK_READY;
                rqRR.current = rqRR.current->next;
            }
            else
            {
                nextTsk->state = TSK_DONE;
                tskDequeueRR();
                destroyTsk(nextTsk->id);
            }

            idleTask->state = TSK_RUNNING;
        }
    }
}
```

- 通过定期的时间中断重新激活调度器，将时间片分给就绪队列的下一个任务。

```
void preemptionParser(void)
{
    timer++;
    if(sysScheduler.tick_hook) sysScheduler.tick_hook();
    tick_hook_arr();
    if(noPreemption) return;
    if (currentTsk!=idleTask && !(timer % timeSlice))
    {
        // myPrintk(0xf,"preempt task: %d\n",currentTsk);
```

```
            currentTsk->preempted = 1;
            if(idleTask->preempted)
            {
                context_switch_interrupt(currentTsk, idleTask);
            }
            else
            {
                context_switch(currentTsk, idleTask);
            }

        }
}
```

短作业优先(SJF)

- SJF调度器模块

```
scheduler schedulerSJF = {
    .nextTsk_func = nextSJFTsk,
    .enqueueTsk_func = tskEnqueueSJF,
    .dequeueTsk_func = tskDequeueSJF,
    .schedulerInit_func = initSJF,
    .schedule = scheduleSJF,
    .createTsk_hook = 0,
    .tick_hook = 0
};
```

- SJF调度器，由于与RR一样是抢占式的调度，共用一个中断响应，不再赘述。此外，SJF的就绪队列用优先队列维护，以exetime为关键字，从小到大排序。优先队列的接口将在下面提到。

```
void scheduleSJF(void)
{
    while (1)
    {
        if (!rqSJFIsEmpty())
        {
            // myPrintk(0xf, "SJF Scheduling...\n");

            // priorityQueueDisplay(&rqSJF);
            myTCB *nextTsk = nextSJFTsk();
            tskDequeueSJF();


            idleTask->state = TSK_READY;
            nextTsk->state = TSK_RUNNING;

            //switch
```

```
            if(nextTsk->preempted)
            {
                context_switch_interrupt(idleTask, nextTsk);
            }
            else
            {
                context_switch(idleTask, nextTsk);
            }

            //return
            // myPrintk(0xf, "Return to IdleTask...\n");


            if (nextTsk->preempted)
            {
                if(nextTsk->exetime > timeSlice)
                {
                    nextTsk->state = TSK_READY;
                    nextTsk->exetime = nextTsk->exetime - timeSlice;
                    tskEnqueueSJF(nextTsk);
                }
                else
                {
                    myPrintk(0x4, "Time exceeded, terminating...\n");
                    nextTsk->state = TSK_DONE;
                    destroyTsk(nextTsk->id);
                }


            }
            else
            {
                nextTsk->state = TSK_DONE;
                destroyTsk(nextTsk->id);
            }


            idleTask->state = TSK_RUNNING;
        }
    }
}
```

统一的任务接口

- 尽管有多种调度方式，仍可以将任务管理的接口统一起来。

```
int createTsk(void (*tskBody)(void), tskPara *para)
{

    int flag = -1;
    int i;
```

```
    for (i = 0; i < TASK_NUM; i++)
    {
        if (TCBPtr[i] == 0)
        {
            TCBPtr[i] = (myTCB *)kmalloc(sizeof(myTCB));
            TCBPtr[i]->preempted = 0;
            TCBPtr[i]->id = i;
            TCBPtr[i]->run = tskBody;
            TCBPtr[i]->state = TSK_WAIT;
            TCBPtr[i]->stkLimit = (unsigned long *)kmalloc(STACK_SIZE);
            TCBPtr[i]->stkBase = TCBPtr[i]->stkLimit + STACK_SIZE - 1;
            TCBPtr[i]->stkTop = TCBPtr[i]->stkLimit + STACK_SIZE;

            if(!para)
            {
                _setTskPara(TCBPtr[i], &defaultTskPara);
            }
            else
            {
                _setTskPara(TCBPtr[i], para);
            }

            //init stack
            *--TCBPtr[i]->stkTop = (unsigned long)0x0202;   //eflags
            *--TCBPtr[i]->stkTop = (unsigned long)0x08;     //cs
            *--TCBPtr[i]->stkTop = (unsigned long)tskBody;  //eip
            *--TCBPtr[i]->stkTop = (unsigned long)0x0202;   //eflags

            *--TCBPtr[i]->stkTop = (unsigned long)0xAAAAAAAA; //eax
            *--TCBPtr[i]->stkTop = (unsigned long)0xCCCCCCCC; //ecx
            *--TCBPtr[i]->stkTop = (unsigned long)0xDDDDDDDD; //edx
            *--TCBPtr[i]->stkTop = (unsigned long)0xBBBBBBBB; //ebx

            *--TCBPtr[i]->stkTop = (unsigned long)0x44444444; //esp
            *--TCBPtr[i]->stkTop = (unsigned long)0x55555555; //ebp
            *--TCBPtr[i]->stkTop = (unsigned long)0x66666666; //esi
            *--TCBPtr[i]->stkTop = (unsigned long)0x77777777; //edi

            flag = i;
            tskCnt++;
            break;
        }
    }
    if(sysScheduler.createTsk_hook) sysScheduler.createTsk_hook(TCBPtr[i]);
    return flag;
}

int enableTsk(int tskIndex)
{
    __asm__ __volatile__("call disable_interrupt");

    if(TCBPtr[tskIndex]->arrtime==0) tskStart(TCBPtr[tskIndex]);
    else tskStartDelayed(TCBPtr[tskIndex]);
```

```
    __asm__ __volatile__("call enable_interrupt");

}
void destroyTsk(int tskIndex)
{
    if (tskCnt > 0 && tskIndex < TASK_NUM && TCBPtr[tskIndex] != 0)
    {
        kfree(TCBPtr[tskIndex]->stkLimit); //free stack
        kfree(TCBPtr[tskIndex]);           //free TCB
        TCBPtr[tskIndex] = 0;              //reset TCB pointer
        tskCnt--;
    }
}

void schedule(void)
{
    sysScheduler.schedule();
}

void tskStart(myTCB *tsk){
    tsk->state = TSK_READY;
    sysScheduler.enqueueTsk_func(tsk);
}
void tskEnd(void){
    //context switch to idle task
    context_switch(currentTsk, idleTask);
}
void initScheduler(void){
    initArrList();
    sysScheduler.schedulerInit_func();
}
```

- 此外，还有管理系统当前调度器的接口。

```
unsigned int getSysScheduler(void);
void setSysScheduler(unsigned int what);
void getSysSchedulerPara(unsigned int who, unsigned int *para);
void setSysSchedulerPara(unsigned int who, unsigned int para);
```

到达队列

- 优先队列的接口，源文件位于/myOS/lib目录下。

```
typedef struct
{
    myTCB *task;
}priorityQueueNode;
```

```
typedef struct
{
    priorityQueueNode *heap;
    int (*compareLT)(unsigned long *priorityQueueNode1, unsigned long
*priorityQueueNode2);
    int total;
}priorityQueue;

void priorityQueueInit(priorityQueue *PQ, int (*cmpLT)(unsigned long
*priorityQueueNode1, unsigned long *priorityQueueNode2));
int priorityQueuePush(priorityQueue *PQ, myTCB *task);
int priorityQueuePop(priorityQueue *PQ);
int priorityQueueIsEmpty(priorityQueue *PQ);
priorityQueueNode* priorityQueueTop(priorityQueue *PQ);
void priorityQueueDisplay(priorityQueue *PQ);
```

- 用优先队列实现到达队列的维护，以arrtime为关键字，从小到大排序。

```
/* for task arriving */
extern void tskStart(myTCB *tsk);
extern unsigned long getTick(void);

/* time unit: tick */
/* zero arriving time: x ticks*/
unsigned int arrTimeBase = 0x0;

priorityQueue PQArr;

int cmpLTArr(priorityQueueNode *node1, priorityQueueNode *node2){
    if(node1->task->arrtime <= node2->task->arrtime) return 1;
    else return 0;
}
void initArrList(void){
    priorityQueueInit(&PQArr, cmpLTArr);
}

/* arrTime: small --> big */
void ArrListEnqueue(myTCB* tsk){
    tsk->state = TSK_WAIT;
    priorityQueuePush(&PQArr, tsk);
}

void tskStartDelayed(myTCB* tsk){
    ArrListEnqueue(tsk);
}

void tick_hook_arr(void){
    while(  !priorityQueueIsEmpty(&PQArr) &&
            priorityQueueTop(&PQArr)->task->arrtime + arrTimeBase <= getTick())
    {
        // myPrintk(0x7,"tick: %d\n", getTick());
        // priorityQueueDisplay(&PQArr);
```

```
        tskStart(priorityQueueTop(&PQArr)->task);
        priorityQueuePop(&PQArr);
    }
}
```

源代码说明

- 代码组织(* 表示更改或新增)

```
|---- lab5/
    |---- src/
        |---- source2img.sh     生成elf脚本
        |---- myOS/
            |---- userInterface.h
            |---- start32.S
            |---- osStart.c
            |---- dev/
                |---- i8253.c
                |---- i8259A.c
                |---- uart.c
                |---- vga.c
            |---- i386/
                |---- io.c
                |---- io.h
                |---- irqs.c
                |---- CTX_SW.S
            |---- kernel/
                |---- mem/
                    |---- eFPartition.c
                    |---- dPartition.c
                    |---- malloc.c
                    |---- pMemInit.c
                |---- tick.c
                |---- wallClock.c
                |---- task.c
                |---- taskPara.c
                |---- scheduler/
                    |---- schedulerFCFS.c
                    |---- schedulerRR.c
                    |---- schedulerSJF.c
            |---- printk/
                |---- myPrintk.c
                |---- vsprintf.c
            |---- include/
                |---- i8295.h
                |---- i8259A.h
                |---- io.h
                |---- irqs.h
                |---- kmem.h
                |---- mem.h
                |---- myPrintf.h
```

```
                |---- myPrintk.h
                |---- priorityQueue.h    *
                |---- scheduler.h        *
                |---- schedulerFCFS.h    *
                |---- schedulerRR.h      *
                |---- schedulerSJF.h     *
                |---- tick.h             *
                |---- task.h             *
                |---- taskPara.h         *
                |---- taskArr.h          *
                |---- uart.h
                |---- vga.h
                |---- vsprintf.h
                |---- wallClock.h
        |---- userApp/
            |---- main.c
            |---- shell.c
            |---- shell.h
            |---- memTestCase.c
            |---- memTestCase.h
            |---- userTasks.c
            |---- userApp.h
        |---- multibootHeader/
            |---- multibootHeader.S
```

- Makefile 组织

```
include $(SRC_RT)/myOS/Makefile
include $(SRC_RT)/userApp/Makefile
```

```
|---- lab6/
    |---- src/
        |---- myOS/
            |---- dev/
            |---- i386/
            |---- kernel/
                |---- mem/
                |---- scheduler/    *
            |---- printk/
        |---- userApp/
```

地址空间说明

- ld文件

```
SECTIONS {
    . = 1M;
```

```
    .text : {
        *(.multiboot_header)
        . = ALIGN(8);
        *(.text)
    }

    . = ALIGN(16);
    .data       : { *(.data*) }

    . = ALIGN(16);
    .bss        :
    {
        __bss_start = .;
        _bss_start = .;
        *(.bss)
        __bss_end = .;
    }
    . = ALIGN(16);
    _end = .;
    . = ALIGN(512);
}
```

- 地址空间表

Offset	Field	Macro
0	.code	
1M	.text	
ALIGN(16)	.data	
ALIGN(16)	.bss	__bss_start, _bss_start
		_bss_end
ALIGN(16)		_end

编译过程说明

- 主Makefile

```
OS_OBJS       = ${MYOS_OBJS} ${USER_APP_OBJS}

output/myOS.elf: ${OS_OBJS} ${MULTI_BOOT_HEADER}
    ${CROSS_COMPILE}ld -n -T myOS/myOS.ld ${MULTI_BOOT_HEADER} ${OS_OBJS} -o
output/myOS.elf

output/%.o : %.S
    @mkdir -p $(dir $@)
    @${CROSS_COMPILE}gcc ${ASM_FLAGS} -c -o $@ $<

output/%.o : %.c
```

```
	@mkdir -p $(dir $@)
	@${CROSS_COMPILE}gcc ${C_FLAGS} -c -o $@ $<
```

- 说明

  根据Makefile分为两步：编译和链接。

  第一步，编译汇编代码(*.S)和c代码(*.c)并输出对象文件(*.o)。

  第二步，将这些对象文件链接并输出可执行可链接文件(myOS.elf)。

运行和运行结果说明

测试用例(SJF)

- 调度器配置

```
void __main(void)
{
    setSysScheduler(SJF);
    TaskManagerInit();
}
```

测试任务

```
void myTsk0(void){
    int j=1;
    while(j<=10){
        myPrintf(0x7,"myTSK0::%d    \n",j);
        busy_n_ms(120);
        j++;
    }
    tskEnd();   //the task is end
}

void myTsk1(void){
    int j=1;
    while(j<=10){
        myPrintf(0x7,"myTSK1::%d    \n",j);
        busy_n_ms(120);
        j++;
    }
    tskEnd();   //the task is end
}

void myTsk2(void){
    int j=1;
    while(j<=10){
        myPrintf(0x7,"myTSK2::%d    \n",j);
```

```
        busy_n_ms(120);
        j++;
    }
    tskEnd();   //the task is end
}
```

加载任务

```
void myMain(void){
    myPrintf(0x4,"myMain()\n");

    int shell = createTsk(shellTask, 0);
    int task0 = createTsk(myTsk0, 0);
    int task1 = createTsk(myTsk1, 0);
    int task2 = createTsk(myTsk2, 0);

    tskPara para[4];
    for(int i=0;i<4;i++) initTskPara(&para[i]);
    setTskPara(ARRTIME, 300, &para[0]);
    setTskPara(EXETIME, 1000, &para[0]);
    _setTskPara(TCBPtr[shell], &para[0]);

    setTskPara(ARRTIME, 500, &para[1]);
    setTskPara(EXETIME, 30, &para[1]);
    _setTskPara(TCBPtr[task0], &para[1]);

    setTskPara(ARRTIME, 600, &para[2]);
    setTskPara(EXETIME, 20, &para[2]);
    _setTskPara(TCBPtr[task1], &para[2]);

    setTskPara(ARRTIME, 700, &para[3]);
    setTskPara(EXETIME, 10, &para[3]);
    _setTskPara(TCBPtr[task2], &para[3]);

    enableTsk(shell);
    enableTsk(task0);
    enableTsk(task1);
    enableTsk(task2);

    tskEnd();
}
```

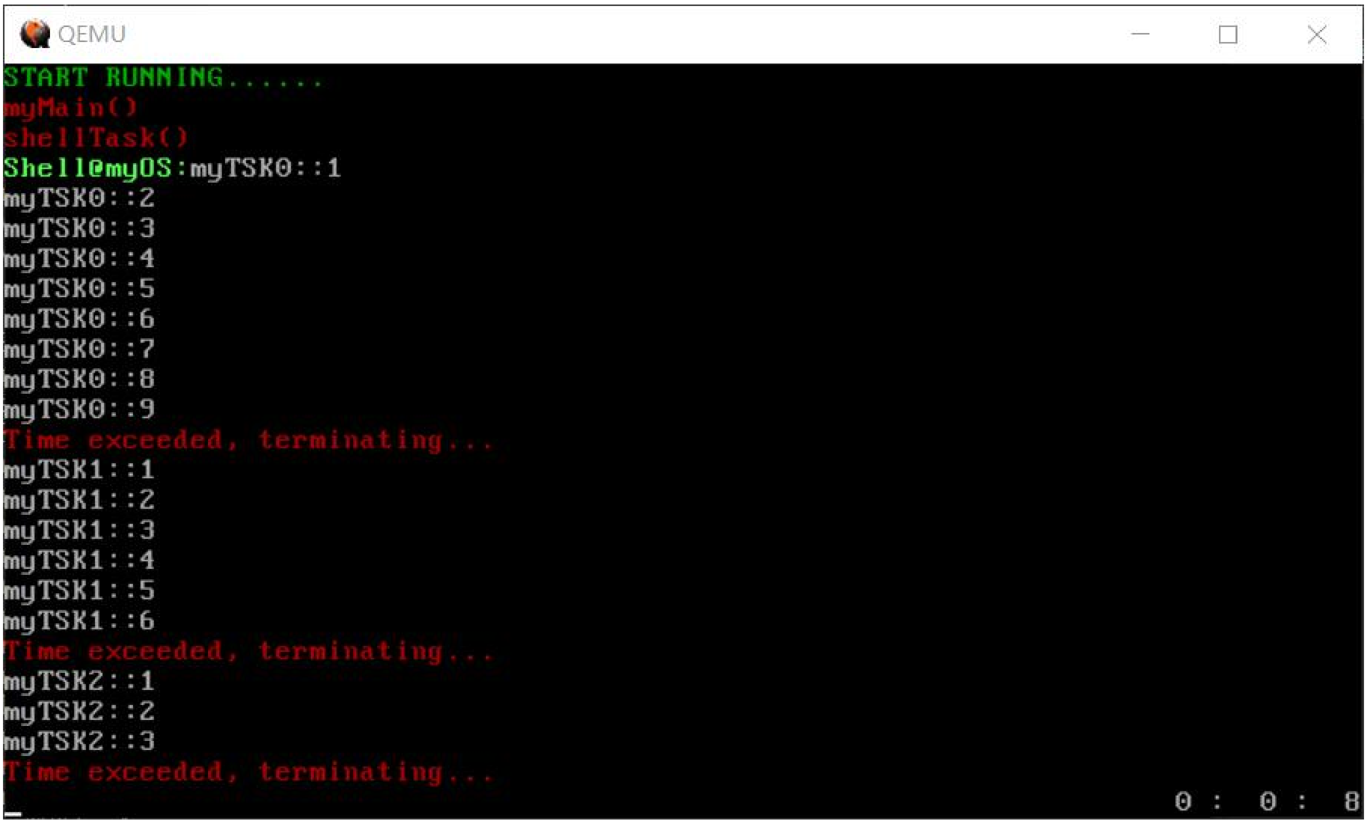
运行

```
执行命令：
`qemu-system-i386 -kernel output/myOS.elf -serial pty &

将之前编译链接生成的elf文件，加载到qemu中运行。
```

运行结果

- 最先加载的是shell任务，其exetime很长，为1000 ticks，因此被随后到达的短任务myTSK0抢占。这说明抢占调度、等待队列功能正常。
- 之后由于exetime不足以运行完整个任务，接下来的三个任务都被终止了。实际上，shell在占有 CPU 1000 ticks 后也会被强制终止。当然，这个强制终止是可定制的。



遇到的问题及解决

问题：调度器出错，任务队列炸了。

解决方案：对于SJF就绪队列，入队和出队都会触发队列的维护。因此，当调度明确下一个该执行的任务时，就必须让它出队。否则，当下一个任务创建子任务后，队头可能发生改变。由于只能让队头出队，因此下次调用出队就可能让新创建的子任务出队，而创建子任务的进程错误地留在队列中，尽管它已经运行完毕。

还有一些很愚蠢的错误，就不提了。