实验 5： Task Management

目录

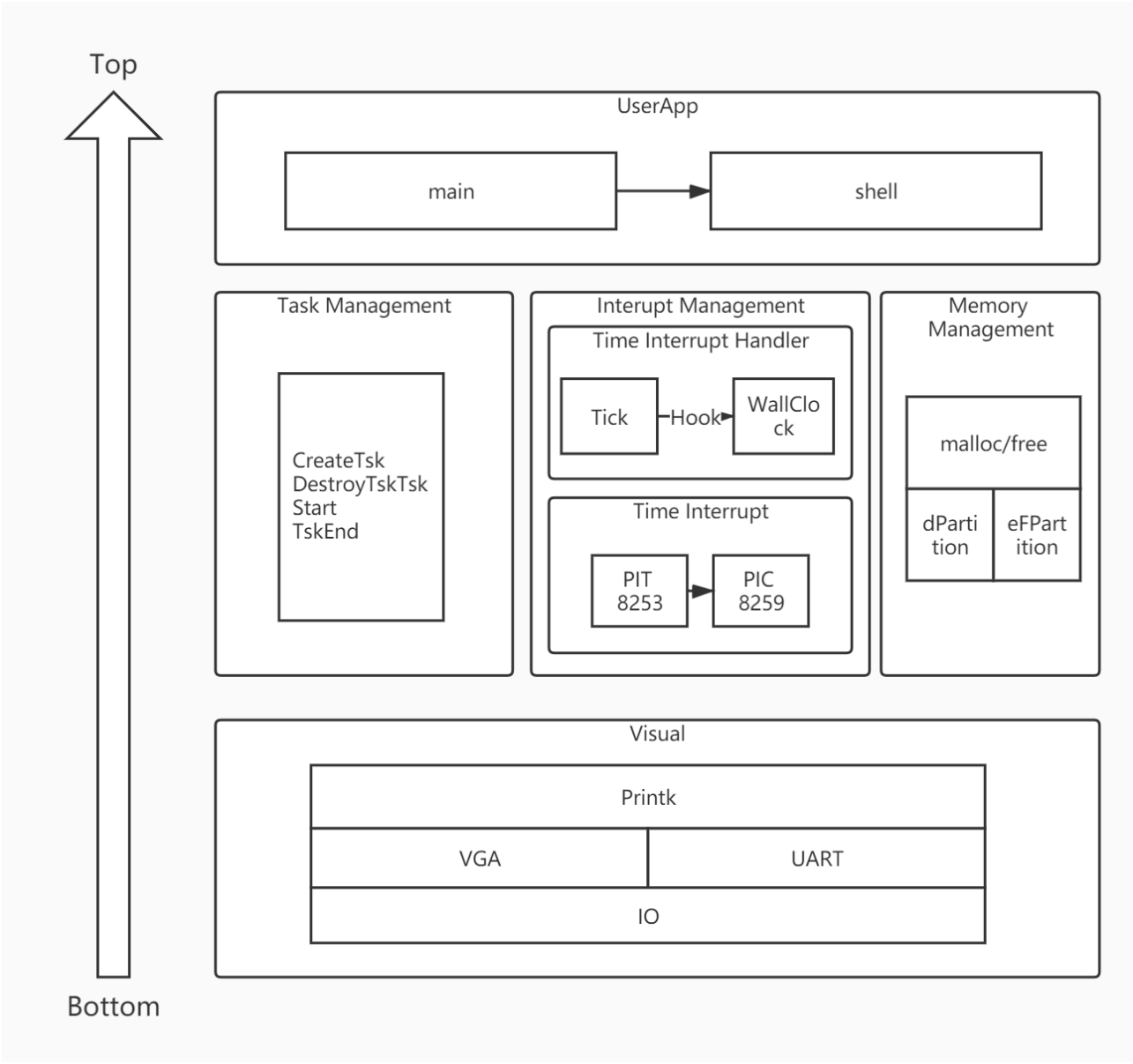
软件框图

- 软件结构

  本次实验主要开发任务管理功能，通过构建任务数据结构、上下文切换、实现FCFS调度算法来完成多任务的调度。

  此外，将Shell封装成了任务，可以实现批处理的操作系统。

- 结构图（底层在下，顶层在上）：

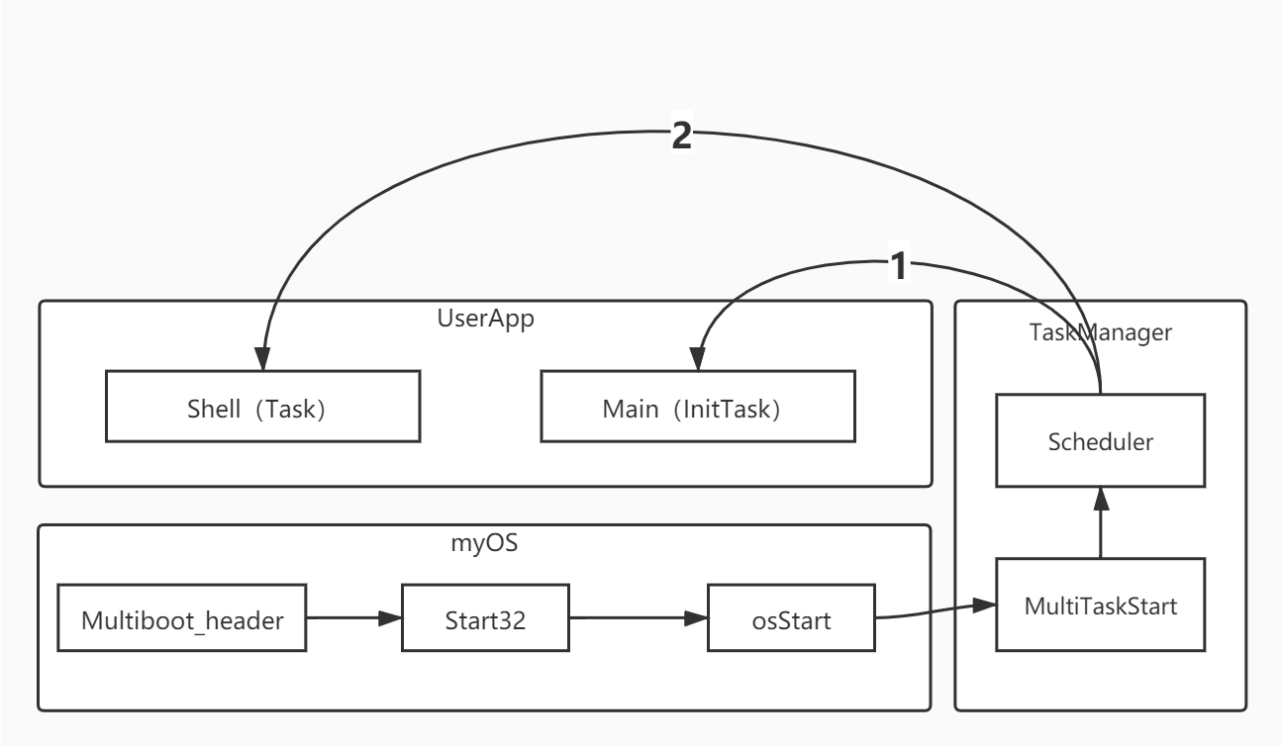


主流程

- 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PIT、PIT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，Shell也被加入就绪队列，从而进入命令行。如果在Shell中调用Exit，所有任务（除了IdleTask）都运行完毕，调度器陷入死循环。

- 流程图



功能模块

概述

软件的唯一新增的模块是进程（任务）管理模块，主要难点在于上下文切换。

多任务FCFS

- FCFS就绪队列出队入队操作。队头是下一个待运行的任务，运行完毕后出队，新任务从队尾加入。

```
/* tskEnqueueFCFS: insert into the tail node */
void tskEnqueueFCFS(myTCB *tsk)
{
    if (rqFCFS.head == 0)
    {
        rqFCFS.head = rqFCFS.tail = tsk;
        tsk->next = 0;
    }
    else
    {
        rqFCFS.tail->next = tsk;
        rqFCFS.tail = tsk;
        tsk->next = 0;
    }
}

/* tskDequeueFCFS: delete the first node */
void tskDequeueFCFS()
{
    if (rqFCFS.head == 0)
```

```
        myPrintk(0x7, "ohho, FCFS seems to have some mistake...\n");
    else if (rqFCFS.head->next == 0)
        rqFCFS.head = rqFCFS.tail = 0;
    else
        rqFCFS.head = rqFCFS.head->next;
}
```

- 任务的创建与销毁

```
int createTsk(void (*tskBody)(void))
{

    int flag = -1;
    int i;
    for (i = 0; i < TASK_NUM; i++)
    {
        if (TCBPtr[i] == 0)
        {
            TCBPtr[i] = (myTCB *)kmalloc(sizeof(myTCB));
            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]->stkBase - 7;
            TCBPtr[i]->next = 0;
            *(TCBPtr[i]->stkBase - 7) = tskBody;

            //init stack
            unsigned long esp;
            __asm__ __volatile__ ("movl %%esp,%0":"=a"(esp));
            __asm__ __volatile__ ("movl %0,%%esp"::"a"(TCBPtr[i]->stkTop));
            __asm__ __volatile__ ("pushf");
            __asm__ __volatile__ ("pusha");
            __asm__ __volatile__ ("movl %%esp,(%0)"::"a"(&(TCBPtr[i]->stkTop)));
            __asm__ __volatile__ ("movl %0,%%esp"::"a"(esp));

            flag = i;
            tskCnt++;
            break;
        }
    }
    return flag;
}

/* destroyTsk
 * takIndex:
 * return value: void
 */
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--;
    }
}
```

- 任务的启动与结束。用户代码需要遵守在任务结尾调用tskEnd的规范。

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

void tskEnd(void)
{
    //context switch to idle task
    context_switch_task(currentTsk,idleTask);
}
```

- 调度算法FCFS

```
void scheduleFCFS(void)
{
    while (1)
    {
        if (!rqFCFSIsEmpty())
        {
            myPrintk(0xf,"IdleTask Scheduling...\n");

            myTCB *nextTsk = nextFCFSTsk();
            idleTask->state = TSK_READY;
            nextTsk->state = TSK_RUNNING;
            currentTsk = nextTsk;
            //switch
            context_switch_task(idleTask, nextTsk);
            //return
            myPrintk(0xf,"Return to IdleTask...\n");
            tskDequeueFCFS();
            destroyTsk(currentTsk->id);

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

        }
```

```
    }
}
```

- 上下文切换

```
.text
.code32
  .global CTX_SW
CTX_SW:
    pushf
    pusha
    movl prevTSK_StackPtr, %eax
    movl %esp, (%eax)
    movl nextTSK_StackPtr, %esp
    popa
    popf
    ret
```

```
void context_switch_stkptr(myTCB *prevTskStkPtr, myTCB *nextTskStkPtr)
{
    prevTSK_StackPtr = (unsigned long **)&prevTskStkPtr;
    nextTSK_StackPtr = (unsigned long *)nextTskStkPtr;
    CTX_SW();
}

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

源代码说明

- 代码组织

```
|---- 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
            |---- printk/
                |---- myPrintk.c
                |---- vsprintf.c
            |---- include/
                |---- i8295.h
                |---- i8259A.h
                |---- io.h
                |---- irqs.h
                |---- kmem.h
                |---- mem.h
                |---- myPrintf.h
                |---- myPrintk.h
                |---- tick.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
```

```
|---- lab5/
    |---- src/
        |---- myOS/
            |---- dev/
            |---- i386/
            |---- kernel/
                |---- mem
            |---- 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

Offset	Field	Macro
		_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

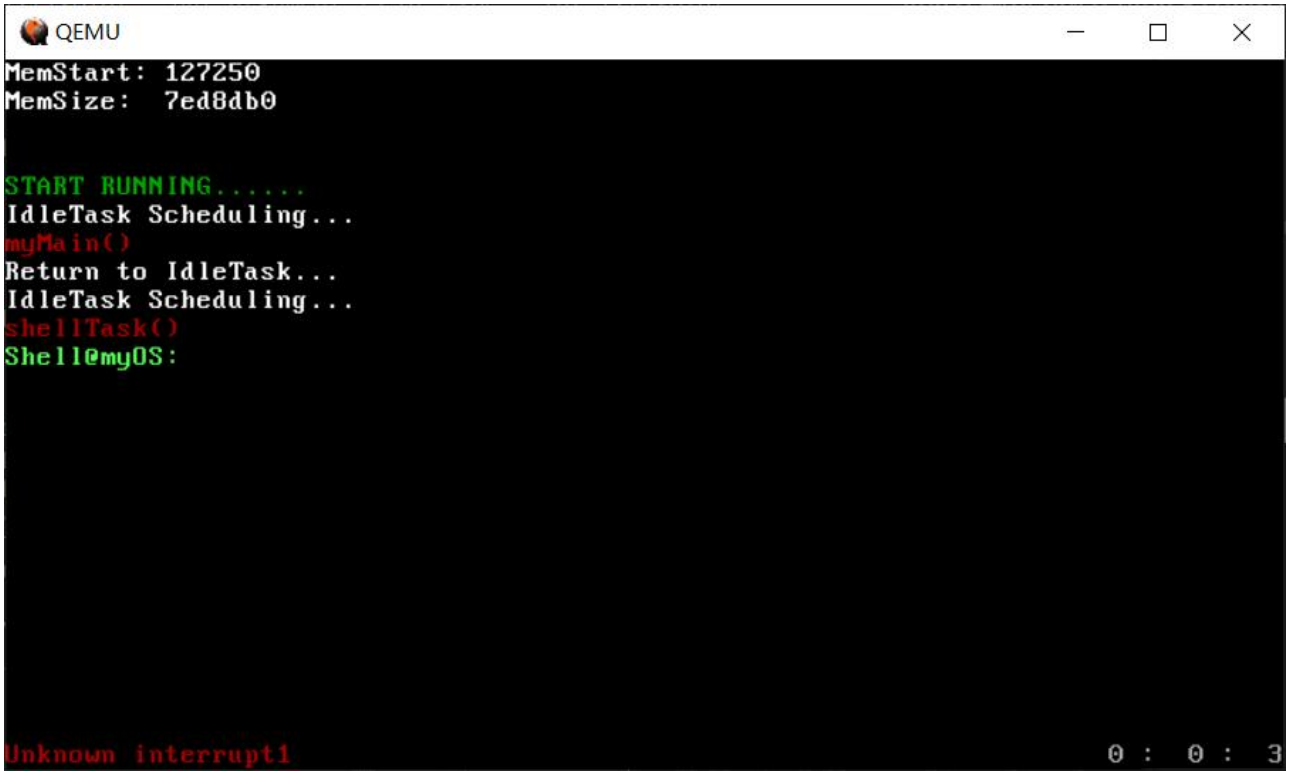
- 运行

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

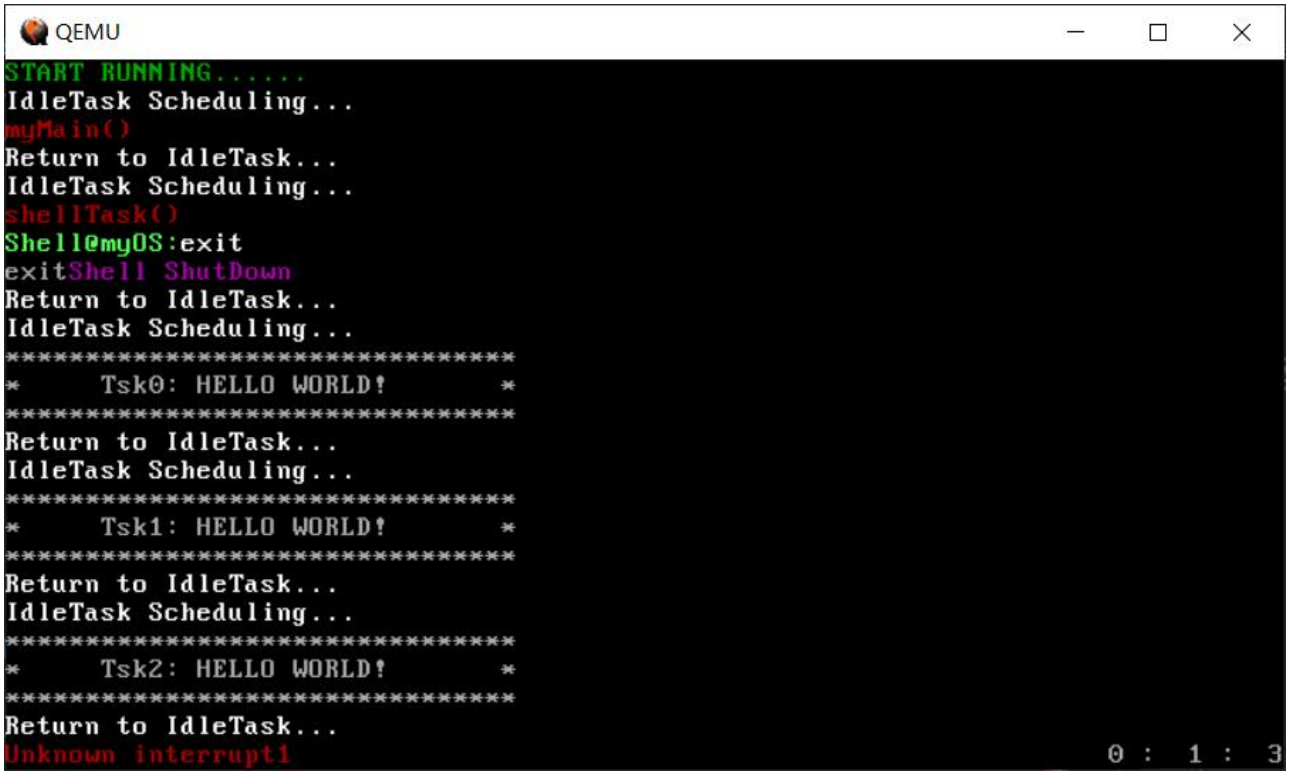
  将之前编译链接生成的elf文件，加载到qemu中运行。

- 运行结果

成功创建了Shell任务，



输入exit命令退出Shell，随即开始其他测试任务。



各测试用例测试结果正常。

遇到的问题及解决

问题：上下文切换后程序跑飞了。

解决方案：创建任务时初始化任务私有栈，先压入taskBody用于ret，再pushf， pusha（这个操作仅仅是为了防止underflow，压入的内容是没有意义的）。如此一来调用上下文切换的时候，对于一个新的任务，popa,

popf就不会underflow，随后的ret直接跳转到taskBody，从而任务得以开始执行。