

lab5实验讲义

陈晨曦

2023 年 5 月 28 日

1 实验需要改动的文件

本次实验只需修改`src > myOS > kernel > task.c`文件，其他文件一律不需改动。

2 实验需要掌握的文件

`src > myOS > kernel > task.c`

`src > myOS > include > task.h`

`src > myOS > i386 > CTX_SW.s`

3 实验需要掌握的进程

本次实验实现的是上下文切换以及FCFS算法，本实验框架中一共有6进程，按进程的建立顺序分别为tskIdleBdy、initTskBody、myTsk0、myTsk1、myTsk2、startShell。为了完成本实验，你需要熟悉本实验中涉及的进程。

3.1 tskIdleBdy

进程死循环，并在循环中不断进行调度。

该进程定义在`src > myOS > kernel > task.c`。

3.2 initTskBody(即myMain)

在这个进程中进行初始化，并在该进程中创建myTsk0、myTsk1、myTsk2、startShell进程。

该进程定义在`src > userApp > main.c`。

3.3 myTsk0、myTsk1、myTsk2

测试进程，只具有myPrintf的功能。

该进程定义在`src > userApp > userTasks.c`。

3.4 startShell

startShell进程，具有之前实验的命令行功能。

该进程定义在`src > userApp > shell.c`。

4 涉及函数的讲解

4.1 myTCB的数据结构

stack: 为myTCB开辟栈空间（本次实验使用CTX_SW来进行上下文切换，而为了实现上下文切换，我们需要维护每个myTCB的stack空间）。

stkTop: 栈顶指针（本次实验使用CTX_SW来进行上下文切换，而为了实现上下文切换，我们需要维护每个myTCB的栈顶指针）。

TSK_State: 进程的状态(进程池中的TCB一共有四种状态：当前进程已经进入就绪队列中、当前进程还未进入就绪队列中、当前进程正在运行、进程池中的TCB为空未进行分配)。

TSK_ID: 进程的ID。

task_entrance: 函数入口（本次实验中，我们通过CTX_SW来进行上下文切换，而不是这个函数入口）。

nextTCB: 对于空闲的TCB，我们将空闲的TCB进行链表维护。对于处于就绪队列中的TCB我们也进行链表维护。

```
typedef struct myTCB {
    unsigned long *stkTop;
    unsigned long stack[STACK_SIZE];
    unsigned long TSK_State;
    unsigned long TSK_ID;
    void (*task_entrance)(void);
    struct myTCB * nextTCB;
} myTCB;
```

实验提供了全局变量currentTsk来表示当前正在运行的TCB，同时我们也提供了firstFreeTsk来表示进程池中第一个未被分配的进程。

```
myTCB * currentTsk;
myTCB * firstFreeTsk;
```

4.2 就绪队列的维护

为管理任务调度，还需实现一个就绪队列，它的元素是 myTCB 。对于 FCFS，你可以实现一个 FIFO 队列，将任务按照到达时间的顺序插入其中。

```
//就绪队列的结构体
typedef struct rdyQueueFCFS{
    myTCB * head;
    myTCB * tail;
    myTCB * idleTsk;
} rdyQueueFCFS;

rdyQueueFCFS rqFCFS;

//初始化就绪队列（需要填写）
void rqFCFSInit(myTCB* idleTsk) {

}
```

```

//如果就绪队列为空，返回True（需要填写）
int rqFCFSIsEmpty(void) {

}

//获取就绪队列的头结点信息，并返回（需要填写）
myTCB * nextFCFSTsk(void) {

}

//将一个未在就绪队列中的TCB加入到就绪队列中（需要填写）
void tskEnqueueFCFS(myTCB *tsk) {

}

//将就绪队列中的TCB移除（需要填写）
void tskDequeueFCFS(myTCB *tsk) {

}

```

4.3 任务池中任务的维护

需要实现任务的创建和销毁两种原语。我们通过静态的方式管理任务池：提前分配好一定数量（可自行配置）的 myTCB，存放在数组（任务池）中。创建任务时，直接从任务池中取出一个空闲的 myTCB；销毁时则将其重新设置为空闲，释放回任务池。

void tskStart(myTCB *tsk)：创建好任务后，需要启动任务时，调用此原语。传入参数是任务的 TCB，原语行为是启动任务，将任务状态设置为就绪，然后插入就绪队列。

void tskEnd()：此原语在某个任务执行结束后被调用。其行为是销毁当前任务，并通知操作系统可以进行调度、开始下一个任务。

```

//进程池中一个未在就绪队列中的TCB的开始（不需要填写）
void tskStart(myTCB *tsk){
    tsk->TSK_State = TSK_RDY;
    //将一个未在就绪队列中的TCB加入到就绪队列
    tskEnqueueFCFS(tsk);
}

//进程池中一个在就绪队列中的TCB的结束（不需要填写）
void tskEnd(void){
    //将一个在就绪队列中的TCB移除就绪队列
    tskDequeueFCFS(currentTsk);
    //由于TCB结束，我们将进程池中对应的TCB也删除
    destroyTsk(currentTsk->TSK_ID);
    //TCB结束后，我们需要进行一次调度
    schedule();
}

```

```

//以tskBody为参数在进程池中创建一个进程，并调用tskStart函数，将其加入就绪队列（需要填写）
int createTsk(void (*tskBody)(void)){//在进程池中创建一个进程，并把该进程加入到rqFCFS队列中

}

//以takIndex为关键字，在进程池中寻找并销毁takIndex对应的进程（需要填写）
void destroyTsk(int takIndex) { //在进程中寻找TSK_ID为takIndex的进程，并销毁该进程

}

```

4.4 stack_init和CTX_SW

为了更好的理解本实验，务必需要了解上下文切换的原理，特别是 stack_init 函数和 CTX_SW函数。值得思考的问题是，在现场的维护中，pushf和popf对应，pusha和popa对应，call和ret对应，但是为什么CTS_SW中只有ret而没有call呢？

```

CTX_SW:
pushf    #旧进程的标志寄存器入栈
pusha    #旧进程的通用寄存器入栈，此条指令和上一条指令一并，起到了保护现场的作用

movl prevTSK_StackPtr, %eax # prevTSK_StackPtrAddr是指针的指针，此行指将其存入 eax 寄存器
movl %esp, (%eax) # ( ) 是访存的标志，该语句的目的是存储任务的栈空间
movl nextTSK_StackPtr, %esp #该语句的目的是通过改变esp来切换栈

popa     #旧进程的通用寄存器出栈
popf     #旧进程的标志寄存器出栈
ret      #返回指令，从栈中取出返回地址，存入 eip 寄存器

```

```

//初始化栈空间（不需要填写）
void stack_init(unsigned long **stk, void (*task)(void)){
    *(*stk)-- = (unsigned long) 0x08;    //高地址
    *(*stk)-- = (unsigned long) task;    //EIP
    *(*stk)-- = (unsigned long) 0x0202; //FLAG寄存器

    *(*stk)-- = (unsigned long) 0xAAAAAAAA; //EAX
    *(*stk)-- = (unsigned long) 0xCCCCCCCC; //ECX
    *(*stk)-- = (unsigned long) 0xDDDDDDDD; //EDX
    *(*stk)-- = (unsigned long) 0BBBBBBBB; //EBX

    *(*stk)-- = (unsigned long) 0x44444444; //ESP
    *(*stk)-- = (unsigned long) 0x55555555; //EBP
    *(*stk)-- = (unsigned long) 0x66666666; //ESI
    *(*stk)  = (unsigned long) 0x77777777; //EDI
}

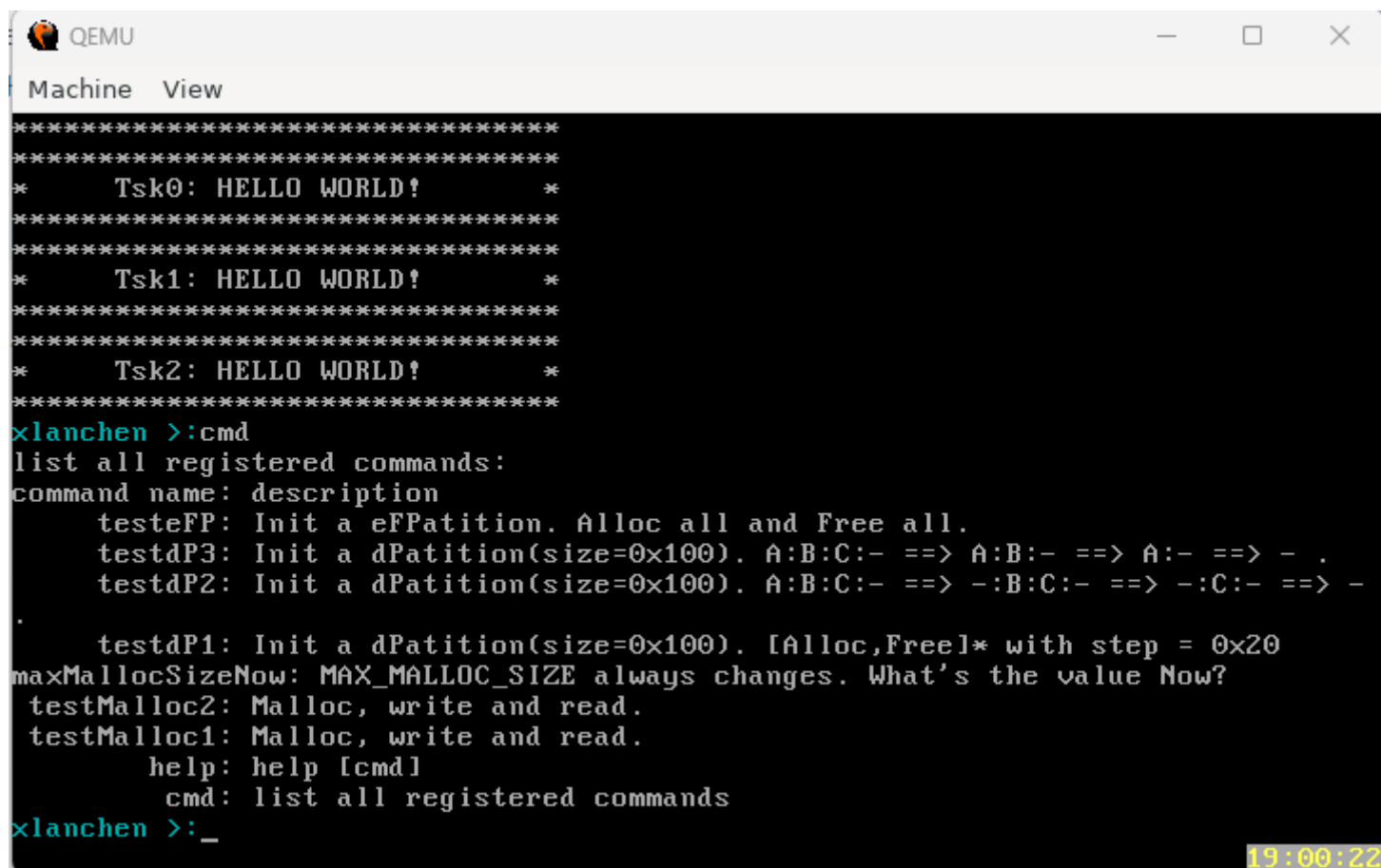
```

4.5 TaskManagerInit和startMultitask

TaskManagerInit：在这个函数中我们实现三件事。1.初始化进程池（所有的进程状态都是TSK_NONE）。2.创建tskIdleBdy和initTskBody任务。3.调用startMultitask，进入多任务调度模式。

startMultitask：进入多任务调度模式。

5 实验结果展示



```
Machine View
*****
*****
*      Tsk0: HELLO WORLD!      *
*****
*****
*      Tsk1: HELLO WORLD!      *
*****
*****
*      Tsk2: HELLO WORLD!      *
*****
xlanchen >:cmd
list all registered commands:
command name: description
  testeFP: Init a eFPatition. Alloc all and Free all.
  testdP3: Init a dPatition(size=0x100). A:B:C:- ==> A:B:- ==> A:- ==> - .
  testdP2: Init a dPatition(size=0x100). A:B:C:- ==> -:B:C:- ==> -:C:- ==> -
  .
  testdP1: Init a dPatition(size=0x100). [Alloc,Free]* with step = 0x20
maxMallocSizeNow: MAX_MALLOC_SIZE always changes. What's the value Now?
testMalloc2: Malloc, write and read.
testMalloc1: Malloc, write and read.
  help: help [cmd]
  cmd: list all registered commands
xlanchen >:_
19:00:22
```

图 1: 实验结果展示

6 实验提交

6.1 思考题

- 1) 在上下文切换的现场维护中，pushf和popf对应，pusha和popa对应，call和ret对应，但是为什么 CTS_SW 函数中只有ret而没有call呢？
- 2) 谈一谈你对 stack_init 函数的理解。
- 3) myTCB结构体定义中的stack[STACK_SIZE]的作用是什么？BspContextBase[STACK_SIZE]的作用又是什么？
- 4) prevTSK_StackPtr是一级指针还是二级指针？为什么？

6.2 实验报告要求

- 1) 回答上述思考题。

2) 截图实验运行结果。

3) 上传源代码。