

# lab5 实验文档

## 1 实验说明

### 1.1 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。

### 1.2 实验需要阅读的文件

- `src \ myOS \ kernel \ task.c`
- `src \ myOS \ kernel \ task.h`
- `src \ myOS \ i386 \ CTX_SW.S`

## 2 实验基础

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。

### 2.1 `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`。

### 2.2 `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`。

### 2.3 `myTsk0`、`myTsk1`、`myTsk2`

- 测试进程，只具有 `myPrintf` 的功能。
- 该进程定义在 `src\userApp\userTasks.c`。

### 2.4 `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`。

## 3 实验内容

### 3.1 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;      /* 栈顶指针 */
    /* 开辟了一个大小为 STACK_SIZE 的栈空间 */
    unsigned long stack[STACK_SIZE];
    unsigned long TSK_State;    /* 进程状态 */
    unsigned long TSK_ID;       /* 进程 ID */
    void (*task_entrance)(void); /* 进程的入口地址 */
    struct myTCB * nextTCB;      /* 下一个 TCB */
} 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;          /* 当前任务 */
myTCB* firstFreeTsk;        /* 下一个空闲的 TCB */
```

### 3.2 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) {
    //对 rqFCFS 进行初始化处理
```

```

}

//如果就绪队列为空，返回 True（需要填写）
int rqFCFSIsEmpty(void) {
    //当 head 和 tail 均为 NULL 时，rqFCFS 为空

}

//获取就绪队列的头结点信息，并返回（需要填写）
myTCB * nextFCFSTsk(void) {
    //获取下一个 Tsk

}

//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 tsk 入队 rqFCFS

}

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

}

```

### 3.3 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

}

```

### 3.4 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 *stackinit* 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？

### 3.5 TaskManagerInit 和 startMultitask

- TaskManagerInit：在这个函数中我们实现三件事。
  - 初始化进程池（所有的进程状态都是 TSK NONE）。
  - 创建 *tskIdleBdy* 和 *initTskBody* 任务。
  - 调用 *startMultitask*，进入多任务调度模式。
- startMultitask：进入多任务调度模式。

```

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) 0xAAAAAAA;    //EAX
    *(*stk)-- = (unsigned long) 0xCCCCCCC;    //ECX
    *(*stk)-- = (unsigned long) 0xDDDDDDD;    //EDX
    *(*stk)-- = (unsigned long) 0BBBBBBB;    //EBX

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

```

```

    *(*stk) = (unsigned long) 0x77777777; //EDI
}

```

## 4 实验示例

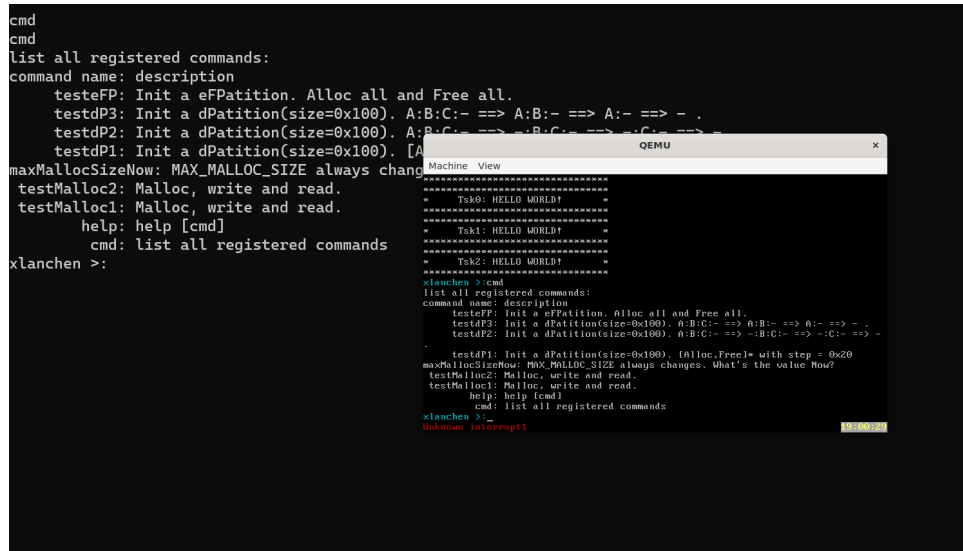


图 1: 实验示例

## 5 提交要求

### 5.1 思考题

- 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?
- 谈一谈你对 stack init 函数的理解。
- 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?
- prevTSK\_ StackPtr 是一级指针还是二级指针? 为什么?

### 5.2 实验报告要求

实验报告中回答上述思考题, 截图实验运行结果。请将代码打包压缩 (包含报告) 提交, 并且将报告再单独提交一份。两份文件一次性提交。命名为学号 \_ 姓名 \_lab5

提交日期: 2025 年 6 月 1 日 23 点 59 分前