操作系统实验五报告

PB21020651 武宇星

思考题

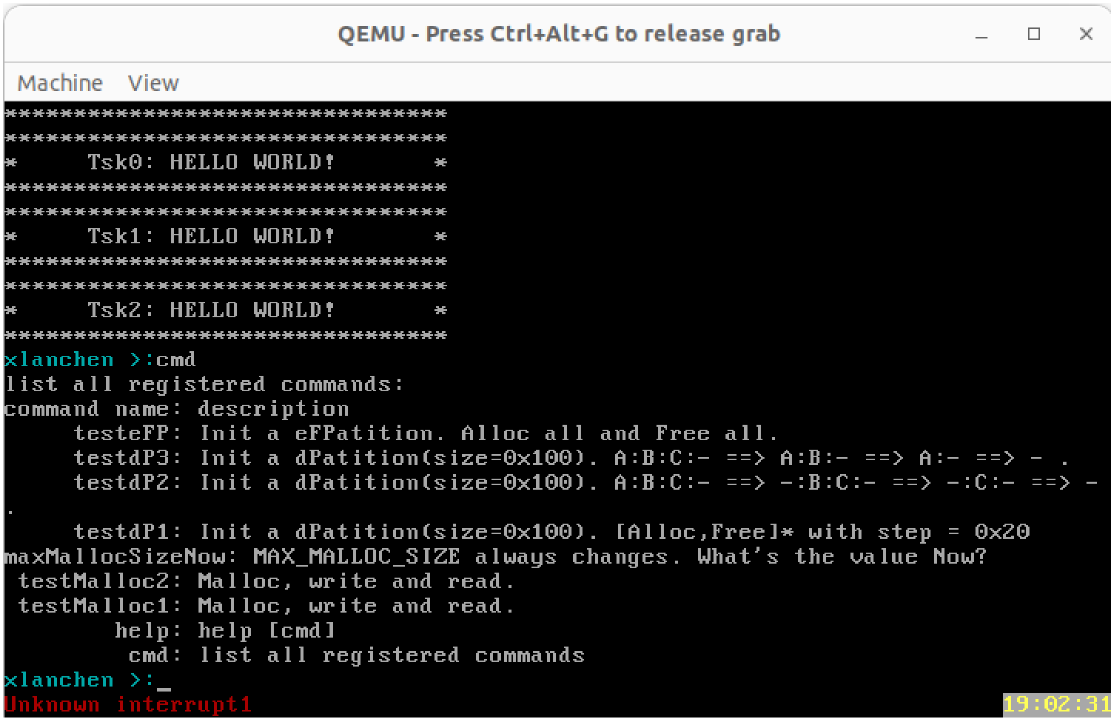
1.因为 CTS_SW 函数不是通过内部 call 调用的，而是在外部 C 语言调用时会自动将函数的返回地址压入栈中 ret 就相当于 return 了

2.Stack_init 函数完成了首次上下文切换的准备工作，进行任务栈的初始化，入栈 tskend 使得每个任务完成时自动跳转到该函数，入栈 task 则是首次上下文切换的任务入口，然后入栈 0x0202 允许中断，最后再入栈八个通用寄存器。

3.stack[STACK SIZE]数组的作用是存储任务的栈空间，每个任务都需要有自己独立的栈空间，用于保存任务执行过程中的局部变量、函数调用信息,而 BspContextBase[STACK_SIZE]数组的作用是保存上下文切换时的栈空间。可以看到他的地址被赋给 prevTSK_StackPtr，保护了前一个任务的上下文。

4.根据定义 unsigned long **prevTSK_StackPtr，它是一个二级指针。

实验运行结果



源代码

```
#include "../include/task.h"
#include "../include/myPrintk.h"
```

```
void schedule(void);
void destroyTsk(int takIndex);

#define TSK_RDY 0             //表示当前进程已经进入就绪队列中
#define TSK_WAIT -1           //表示当前进程还未进入就绪队列中
#define TSK_RUNING 1          //表示当前进程正在运行
#define TSK_NONE 2            //表示进程池中的 TCB 为空未进行分配

//tskIdleBdy 进程（无需填写）
void tskIdleBdy(void) {
	while(1){
		schedule();
	}
}

myTCB tcbPool[TASK_NUM];//进程池的大小设置

myTCB * idleTsk;                    /* idle 任务 */
myTCB * currentTsk;                 /* 当前任务 */
myTCB * firstFreeTsk;               /* 下一个空闲的 TCB */

//tskEmpty 进程（无需填写）
void tskEmpty(void){
}

//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
	rqFCFS.head = (void *)0;
	rqFCFS.tail = (void *)0;
	rqFCFS.idleTsk = idleTsk;
}

//如果就绪队列为空，返回 True（需要填写）
int rqFCFSIsEmpty(void) {//当 head 和 tail 均为(void *)0 时，rqFCFS 为空
```

```
	return (rqFCFS.head == (void *)0 && rqFCFS.tail == (void *)0);
}

//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
	if (rqFCFSIsEmpty()) {
		return rqFCFS.idleTsk;
	} else {
		return rqFCFS.head;
	}
}

//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
	if (rqFCFSIsEmpty()) {
		rqFCFS.head = tsk;
		rqFCFS.tail = tsk;
		tsk->nextTCB = (void *)0;
	} else {
		rqFCFS.tail->nextTCB = tsk;
		rqFCFS.tail = tsk;
		tsk->nextTCB = (void *)0;
	}
}

//将就绪队列中的 TCB 移除（需要填写）
void tskDequeueFCFS(myTCB *tsk) {//rqFCFS 出队
	if (rqFCFSIsEmpty()) {
		tsk = (void *)0;
		return;
	} else {
		tsk = rqFCFS.head;
		rqFCFS.head = rqFCFS.head->nextTCB;
		if (rqFCFS.head == (void *)0) {
			rqFCFS.tail = (void *)0;
		}
	}
}

//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) 0xBBBBBBBB; //EBX

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

}

//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
    if (firstFreeTsk == (void *)0) {
        return -1;
    }

    myTCB *newTsk = firstFreeTsk;
    firstFreeTsk = firstFreeTsk->nextTCB;
    newTsk->TSK_State = TSK_RDY;
    newTsk->task_entrance = tskBody;
    stack_init(&(newTsk->stkTop), tskBody);
    tskEnqueueFCFS(newTsk);
```

```
	return newTsk->TSK_ID;
}

//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 tSkIndex) {//在进程中寻找 TSK_ID 为 takIndex 的进程，并销毁该进程
	if (currentTsk == (void *)0) {
		return;
	}

	tcbPool[tSkIndex].nextTCB = firstFreeTsk;
	tcbPool[tSkIndex].stkTop = tcbPool[tSkIndex].stack+STACK_SIZE-1;;
	tcbPool[tSkIndex].task_entrance = tskEmpty;
	tcbPool[tSkIndex].TSK_State = TSK_NONE;

	firstFreeTsk = &tcbPool[tSkIndex];
}

unsigned long **prevTSK_StackPtr;
unsigned long *nextTSK_StackPtr;

//切换上下文（无需填写）
void context_switch(myTCB *prevTsk, myTCB *nextTsk) {
	prevTSK_StackPtr = &(prevTsk->stkTop);
	currentTsk = nextTsk;
	nextTSK_StackPtr = nextTsk->stkTop;
	CTX_SW(prevTSK_StackPtr,nextTSK_StackPtr);
}

//FCFS 调度算法（无需填写）
void scheduleFCFS(void) {
	myTCB *nextTsk;
	nextTsk = nextFCFSTsk();
	context_switch(currentTsk,nextTsk);
}

//调度算法（无需填写）
void schedule(void) {
	scheduleFCFS();
}

//进入多任务调度模式(无需填写)
unsigned long BspContextBase[STACK_SIZE];
unsigned long *BspContext;
```

```
void startMultitask(void) {
	BspContext = BspContextBase + STACK_SIZE -1;
	prevTSK_StackPtr = &BspContext;
	currentTsk = nextFCFSTsk();
	nextTSK_StackPtr = currentTsk->stkTop;
	CTX_SW(prevTSK_StackPtr,nextTSK_StackPtr);
}

//准备进入多任务调度模式(无需填写)
void TaskManagerInit(void) {
	// 初始化进程池（所有的进程状态都是 TSK_NONE）
	int i;
	myTCB * thisTCB;
	for(i=0;i<TASK_NUM;i++){//对进程池 tcbPool 中的进程进行初始化处理
		thisTCB = &tcbPool[i];
		thisTCB->TSK_ID = i;
		thisTCB->stkTop = thisTCB->stack+STACK_SIZE-1;//将栈顶指针复位
		thisTCB->TSK_State = TSK_NONE;//表示该进程池未分配，可用
		thisTCB->task_entrance = tskEmpty;
		if(i==TASK_NUM-1){
			thisTCB->nextTCB = (void *)0;
		}
		else{
			thisTCB->nextTCB = &tcbPool[i+1];
		}
	}
	//创建 idle 任务
	idleTsk = &tcbPool[0];
	stack_init(&(idleTsk->stkTop),tskIdleBdy);
	idleTsk->task_entrance = tskIdleBdy;
	idleTsk->nextTCB = (void *)0;
	idleTsk->TSK_State = TSK_RDY;
	rqFCFSInit(idleTsk);

	firstFreeTsk = &tcbPool[1];

	//创建 init 任务
	createTsk(initTskBody);

	//进入多任务状态
	myPrintk(0x2,"START MULTITASKING......\n");
	startMultitask();
	myPrintk(0x2,"STOP MULTITASKING......SHUT DOWN\n");
```

```
}
```