一、实验目的

实现一个简单的任务调度，了解基于时间中断进行进程切换完成任务调度的全过程。

二、设计思路

1. 在实验 2 的代码基础上，进行实验 3 的实现。

2. 初始化 IDT，

时间中断，时间中断的中断向量为 0x20，

在 idt.c 中的 initIdt 函数中参照 0x80 中断设置新的中断向量：

setIntr(idt+0x20, SEG_KCODE, (uint32_t)irqTimeHandle, DPL_USER);

在 doIrq 中：

```
.global irqTimeSyscall
irqTimeSyscall:
        pushl $0 // push dummy error code
        pushl $0x20 // push interrupt vector into kernel stack
        jmp asmDoIrq

.global irqSyscall
irqSyscall:
        pushl $0 // push dummy error code
        pushl $0x80 // push interrupt vector into kernel stack
        jmp asmDoIrq

.global asmDoIrq
asmDoIrq:
        pushal // push process state into kernel stack
        pushl %ds
        pushl %es
        pushl %fs
                                                        19,0-1        42%
```

修改 doIrq.S 文件中的相关代码，才能正确跳转到处理时间中断的函数。类似 irqSyscall，需要 push $0x20，然后跳转到 asmDoIrq，push %ds %es %fs %gs 后，调用 irqHandle 函数进行处理。这里已经完成了对 IDT 的初始化，以及相关跳转的函数的预处理。

3. 需要对 GDT 表项进行初始化，实验两个用户进程，初始化一个用户进程的 GDT 表项，设置一个新的 base：0x100000

```
        gdt[SEG_UCODE1] = SEG(STA_X | STA_R, 0x100000,       0xffffffff, DPL_USE
R);
        gdt[SEG_UDATA1] = SEG(STA_W,         0x100000,       0xffffffff, DPL_USE
R);
        gdt[SEG_VIDEO] = SEG(STA_W,         0xb8000,       0xffffffff, DPL_USER)
```

PCB（进程控制块），PCB 的结构如下：

struct TrapFrame {

uint32_t gs, fs, es, ds;

uint32_t edi, esi, ebp, xxx, ebx, edx, ecx, eax;

uint32_t irq; // 中断号

uint32_t error; // Error Code

uint32_t eip, cs, eflags, esp, ss;

};

struct ProcessTable {

uint32_t stack[MAX_STACK_SIZE]; // 内核堆栈
struct TrapFrame tf;
int state;//状态，虽然我不是通过这个来判断是否处于 RUNNING 或者
RUNNABLE 或者 BLOCKED
int timeCount;//时间片
int sleepTime;
uint32_t pid;//进程的 PID 号
};

对进程 0（idle）进行初始化，
用户进程 1（PID=1）进行初始化，
初始化 TSS，tss.ss 不需要改变，tss.esp0 指向用户进程 1（PID=1）的 tf 的头部。

```
void kernel_thread() {
    pcb[0].tf.cs = KSEL(SEG_KCODE);
    pcb[0].tf.ds = pcb[0].tf.es = pcb[0].tf.fs = pcb[0].tf.gs = pcb[0].tf.ss= KS
EL(SEG_KDATA);
    Log("%d, %d, %d\n", pcb[0].tf.cs, pcb[0].tf.ds, pcb[0].tf.ss);
    pcb[0].tf.eip = (uint32_t)cpu_idle;
    pcb[0].esp0 = (uint32_t)(&(pcb[0].state));
    pcb[0].state = RUNNABLE;
    pcb[0].timeCount = 0;
    pcb[0].sleepTime = 0;
    pcb[0].pid = 0;
    idleproc = &pcb[0];
}

void user_process(){
    pcb[1].tf.cs = USEL(SEG_UCODE);
    pcb[1].tf.ds = pcb[1].tf.es = pcb[1].tf.fs = pcb[1].tf.gs = pcb[1].tf.ss= US
EL(SEG_UDATA);
    Log("%d, %d, %d\n", pcb[1].tf.cs, pcb[1].tf.ds, pcb[1].tf.ss);
    pcb[1].state = RUNNABLE;
    pcb[1].esp0 = (uint32_t)(&(pcb[1].state));
    pcb[1].timeCount = 10;
                                                                  40,1          7%
```

时间中断处理。思路如下：
每当产生时间中断，遍历数组有效部分，用其中的 PID 号去查找 PCB 数组，将所有 BOLCK 的 sleepTime 减 1，判断此时 sleepTime 是否小于等于 0，如果满足，则将 blocked 中的这个进程号移入 runnable,判断 running 正在进行的进程是否为 0 号进程（PID=0，idle），如果是 idle，则去寻找 runnable 队列中是否有其他进程准备就绪，若存在 RUNNABLE 的进程，将其替换，并给其一定的时间片（timeCount=10），若不存在，则不进行操作，继续让 idle 运行。如果正在 running 的这个进程不是 idle，则将其时间片减 1，判断其时间片是否耗尽，若耗尽，则去 runnable 中寻找是否有其他进程准备就绪，若无，则继续给 IDLE，若有.

```
void schedule()
{
        Log("schedule\n");
        int i = 1;
        for(i = 1; i < processnum; i++)
        {
                if(pcb[i].state == RUNNABLE && pcb[i].pid != curproc->pid)
                {
                        struct ProcessTable * proc = &pcb[i];
                        Log("%d, %d, yonghu\n", i, curproc->pid);
                        proc_run(proc);
                        break;
                }
        }
        if(i == processnum)
        {
                proc_run(idleproc);
        }
}
```

```
void proc_run(struct ProcessTable *proc) {
        curproc = proc;
        Log("%x\n", curproc - pcb);
        Log("%x\n", curproc->tf.eip);
        curproc->sleepTime = 0;
        curproc->timeCount = 10;
        curproc->state = RUNNING;
        tss.ss0 = KSEL(SEG_KDATA);
        tss.esp0 = (int)(&curproc->state);
        asm volatile ("movl %0, %%esp" :: "r" (&curproc->tf));
        asm volatile ("popl %gs");
        asm volatile ("popl %fs");
        asm volatile ("popl %es");
        asm volatile ("popl %ds");
        asm volatile ("popal");
        asm volatile ("addl $8, %esp");
        asm volatile ("iret");
        Log("yyyyyyy\n");
}
```

FORK:

一个空的，可用的 PCB 块，对新的 PCB 块进行初始化，部分值与父进程不同，但大多数的相同的，ss、ds、cs 通过最开始设置的用户进程 2 的 GDT 表项来设置值。同时要将父进程在内存中的内容复制到子进程中去，完成后，将子进程放入 runnable。

如图：

```
        Log("kkkkk\n");
        int i = 0;
        int childnum = 0;
        for(; i < processnum; i++)
        {
                if(pcb[i].state == DEAD)
                {
                        childnum = i;
                        break;
                }
        }
        if(i == processnum)
                childnum = i;
        processnum++;
        pcb[childnum] = *curproc;
        pcb[childnum].pid = childnum;
        pcb[childnum].state = RUNNABLE;
        pcb[childnum].timeCount = 10;
        pcb[childnum].tf.esp = (int)(&pcb[childnum].state);
        pcb[childnum].tf.eax = 0;
        pcb[childnum].tf.cs = USEL(SEG_UCODE1);
        pcb[childnum].tf.ds = USEL(SEG_UDATA1);
        pcb[childnum].tf.ss = USEL(SEG_UDATA1);
```

SLEEP:

将正在 running 的进程放入 blocked 队列中去，并且将其 sleepTime 修改，将 runnable 中的可用进程放入 running 中，若 runnable 队列为空，则将 idle 放入 running 中。

如图：

```
void sys_sleep(struct TrapFrame *tf)
{
        Log("%d\n", curproc->pid);
        int sleep_time = tf->eax;
        Log("%d\n", sleep_time);
        if(curproc->pid != 0)
        {
                curproc->sleepTime = sleep_time;
                curproc->state = BLOCKED;
                curproc->timeCount = 10;
                schedule();
                Log("vvvvvvvv\n");
        }
}
```

EXIT

exit 将取下来的 running 进程直接杀死，设为 DEAD

```
void sys_exit(struct TrapFrame *tf)
{
        if(curproc->pid != 0)
        {
                Log("hhhhhhh\n");
                curproc->state = DEAD;
                curproc->timeCount = 10;
                curproc->sleepTime = 0;
                schedule();
                Log("lllllll`\n");
        }
}
```

很遗憾，最终实验结果中碰到 popGP，最终只实现如下部分，即从子进程切换到 idle 和，idle 切换到到子进程，再次切换到 idle 转换父进程时，则失败。
回去继续调整