

# 《操作系统实验》实验报告

实验名称：进线程切换

姓名：胡育玮

学号：171860574

邮箱：[yeevee@qq.com](mailto:yeevee@qq.com)

班级：17 级计算机科学与技术系 2 班

## 一、实验内容

1. 内核: 实现进程切换机制, 并提供系统调用 `fork`, `sleep`, `exit`
2. 库: 对上述系统调用进行封装; 实现一个用户态的线程库, 完成 `pthread_create` 、 `pthread_join` 、 `pthread_yield` 、 `pthread_exit` 等接口
3. 用户: 对上述库函数进行测试

## 二、实验过程

(1) 进程切换：使用实验网站提供的测试函数：

```
int uEntry(void)
{
    int ret = fork();
    int i = 8;

    if (ret == 0)
    {
        data = 2;
        while (i != 0)
        {
            i--;
            printf("Child Process: Pong %d, %d;\n", data, i);
            sleep(128);
        }
        exit();
    }

    else if (ret != -1)
    {
        data = 1;
        while (i != 0)
        {
            i--;
            printf("Father Process: Ping %d, %d;\n", data, i);
            sleep(128);
        }
        exit();
    }

    return 0;
}
```

系统调用的流程：

1) 测试函数调用 fork(): `int ret = fork();`

2) 进入位于 lib/syscall.c 的 fork()函数体:

```
pid_t fork() {
    return syscall(SYS_FORK, 0, 0, 0, 0, 0);
}
```

可见 fork()调用 syscall()来处理 fork 系统调用。

3) syscall()中:

```
asm volatile("movl %%eax, %0"::"m"(eax));
asm volatile("movl %%ecx, %0"::"m"(ecx));
asm volatile("movl %%edx, %0"::"m"(edx));
asm volatile("movl %%ebx, %0"::"m"(ebx));
asm volatile("movl %%esi, %0"::"m"(esi));
asm volatile("movl %%edi, %0"::"m"(edi));
asm volatile("movl %0, %%eax"::"m"(num));
asm volatile("movl %0, %%ecx"::"m"(a1));
asm volatile("movl %0, %%edx"::"m"(a2));
asm volatile("movl %0, %%ebx"::"m"(a3));
asm volatile("movl %0, %%esi"::"m"(a4));
asm volatile("movl %0, %%edi"::"m"(a5));
asm volatile("int $0x80");
asm volatile("movl %%eax, %0"::"m"(ret));
asm volatile("movl %0, %%eax"::"m"(eax));
asm volatile("movl %0, %%ecx"::"m"(ecx));
asm volatile("movl %0, %%edx"::"m"(edx));
asm volatile("movl %0, %%ebx"::"m"(ebx));
asm volatile("movl %0, %%esi"::"m"(esi));
asm volatile("movl %0, %%edi"::"m"(edi));
```

Syscall()中保存寄存器原来的值，把系统调用需要的参数放入相应寄存器中，然后 int 0x80

4) int 0x80 后最终来到 doIrq.S 中的 asmDoIrq 处:

```
.global asmDoIrq
asmDoIrq:
    pushal // push process state into kernel stack
    pushl %ds
    pushl %es
    pushl %fs
    pushl %gs
    pushl %esp //esp is treated as a parameter
    call irqHandle
    addl $4, %esp //esp is on top of kernel stack
    popl %gs
    popl %fs
    popl %es
    popl %ds
    popal
    addl $4, %esp //interrupt number is on top of kernel stack
    addl $4, %esp //error code is on top of kernel stack
    iret
```

这个汇编函数把现场保存到用户栈中,调用 `irqHandle()`。  
`irqHandle()`最终调用 `syscallFork()`

5)在 `syscallFork()`中,根据注释内的指示来完成下一步工作:  
填充 `fork` 出的子进程的 PCB; 并尤其注意其 `regs` 字段。

```
if (i != MAX_PCB_NUM) {
    /*XXX copy userspace
    XXX enable interrupt
    */
    enableInterrupt();
    for (j = 0; j < 0x100000; j++) {
        *(uint8_t *) (j + (i+1)*0x100000) = *(uint8_t *) (j + (current-
        //asm volatile("int $0x20"); //XXX Testing irqTimer during sy
    }
    /*XXX disable interrupt
    */
    disableInterrupt();
    /*XXX set pcb
    XXX pcb[i]=pcb[current] doesn't work
    */
    /*XXX set regs */

    /*XXX set return value */
    pcb[i].regs.eax = 0;
    pcb[current].regs.eax = i;
}
else {
    pcb[current].regs.eax = -1;
}
return;
```

填写 PCB 的步骤:

1)设置核心栈栈顶: 设置新栈顶为 `pcb[i].regs` 的地址, 以便  
后续进行进程切换时, 能够恢复切换到的进程的在之前保存  
的现场信息:

```
pcb[i].stackTop = (uint32_t)(&pcb[i].regs); //设置栈顶:使得进程切换时能够读取保存在PCB中的现场信息
```

## 2) 其他一些字段的设置:

```
pcb[i].stackTop = (uint32_t)(&pcb[i].regs); //设置栈顶:核心栈每次使用后都要置空,
pcb[i].prevStackTop = pcb[current].prevStackTop;

pcb[i].state = STATE_RUNNABLE;           //进程状态为就绪
pcb[i].timeCount = 0;                    //时间片: 0
pcb[i].sleepTime = 0;                    //睡眠时间: 新fork出的进程没有睡眠
pcb[i].pid = pcb[current].pid + 4;       //这个可以随意, 不超出范围即可

for (j = 0; j < 32; j++)                 //name字段的拷贝
    pcb[i].name[j] = pcb[current].name[j];
```

3) regs 的设置: 把被 fork 的进程的运行现场保存在 fork 出的子进程的 PCB 里, 后面子进程被调度执行时, regs 里保存的值会被恢复到 CPU 的寄存器内:

```
/*XXX set regs */ //恢复之前保存到栈里的现场: 使用sf
pcb[i].regs.ss = USEL(2 * i + 2);
pcb[i].regs.esp = sf->esp;
pcb[i].regs.eflags = sf->eflags;
pcb[i].regs.cs = USEL(2 * i + 1);
pcb[i].regs.eip = sf->eip;

pcb[i].regs.error = sf->error;
pcb[i].regs.irq = sf->irq;

//pcb[i].regs.eax = sf->eax;
pcb[i].regs.ecx = sf->ecx;
pcb[i].regs.edx = sf->edx;
pcb[i].regs.ebx = sf->ebx;
pcb[i].regs.xxx = sf->xxx;
pcb[i].regs.ebp = sf->ebp;
pcb[i].regs.esi = sf->esi;
pcb[i].regs.edi = sf->edi;

pcb[i].regs.ds = USEL(2 * i + 2);
pcb[i].regs.es = USEL(2 * i + 2);
pcb[i].regs.fs = USEL(2 * i + 2);
pcb[i].regs.gs = USEL(2 * i + 2);
```

除段寄存器的值应该额外设置以外, 其它寄存器直接赋

为之前压入栈中的 sf 中相应变量的值。

段寄存器需要额外考虑如何赋值。

在 kvm.c 中，initProc()对主进程和一个用户进程进行了初始化，其中用户进程的初始化中，其对段寄存器的设置如下：

```
pcb[1].regs.ss = USEL(4);
pcb[1].regs.esp = 0x100000;
asm volatile("pushfl");
asm volatile("popl %0" : "=r"(pcb[1].regs.eflags));
pcb[1].regs.eflags = pcb[1].regs.eflags | 0x00000001;
pcb[1].regs.cs = USEL(3);
pcb[1].regs.eip = loadUMain();
pcb[1].regs.ds = USEL(4);
pcb[1].regs.es = USEL(4);
pcb[1].regs.fs = USEL(4);
pcb[1].regs.gs = USEL(4);
```

其中 USEL 宏的定义如下：

```
#define USEL(desc) (((desc) << 3) | DPL_USER)
```

根据实验讲解 PDF，GDT 共有 10 项。第 0 项为空；第 9 项指向 TSS 段，内核代码段和数据段占第 1、2 两项；剩下的 6 项可以作为 3 个用户进程的代码段和数据段。

由此可以认为，1 号用户进程的代码段和数据段为第 3、4 项，2 号用户进程的代码段和数据段为第 5、6 项，以此类推。因此对第 i 号用户进程，代码段为第  $2(i+1)-1$  项，数据段为第  $2(i+1)$  项，故在对 `pcb[i]` 的段寄存器进行设置时，有了上面语句。

#### 4) 返回值

```
/*XXX set return value */
pcb[i].regs.eax = 0;           //子进程:返回值为0
pcb[current].regs.eax = pcb[i].pid; //父进程:为子进程的pid
}

else {
    pcb[current].regs.eax = -1;    //失败则返回-1
}
return;
}
```

Fork 出的子进程的返回值为 0，父进程返回值为 fork 出的子进程的 pid 号码。

至此，fork()已实现完毕。

**exit()**: exit()是另一个系统调用，其实现方法如下：

进入 syscallExit()后：

- 1) 由于是退出进程，因此把进程状态设为 dead:
- 2) 调度另外的进程来执行。

调度过程：在进程表里选一个状态为 runnable 进程为当前进程，若全部用户进程都不是 runnable 状态，则选择 0 号进程（IDLE 进程）为当前进程：



```

void syscallExit(struct StackFrame *sf) {
    pcb[current].state = STATE_DEAD;

    int i;
    int find = 0;
    for (i = (current + 1) % MAX_PCB_NUM; i != current; i = (i + 1) % MAX_PCB_NUM)
    {
        if (pcb[i].state == STATE_RUNNABLE)
        {
            find = 1;
            break;
        }
    }

    if (find)
        current = i;
    else
        current = 0; //IDLE
}

```

将选中的进程置为当前进程，设置当前进程的状态为 running，再设置 tss 任务状态段的 esp0 项的值为当前进程的核心栈栈顶：

```

pcb[current].state = STATE_RUNNING; //切换，开始工作
tss.esp0 = (uint32_t)(&pcb[current].stackTop); //设置tss状态段

asm volatile("movl %0, %%esp" :: "m"(pcb[current].stackTop));
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");

```

然后将当前进程的内核栈顶置入 esp 寄存器中，**恢复保存在 pcb[current].regs 中的现场信息**，再返回到下一条指令，正式开始执行新进程。

**sleep()**: 最后一个系统调用是 sleep。实现过程:

1) 设置当前进程的状态为 blocked, 同时将 tf->ecx 中保存的 sleeptime 存入 PCB 中:

```
void syscallSleep(struct StackFrame *sf) {  
    pcb[current].state = STATE_BLOCKED;  
    pcb[current].sleepTime = sf->ecx;  
}
```

2) 执行进程切换。切换的步骤与 exit()中的一样, 不再赘述。

**还有一个要实现的函数是时钟中断**。实现细节:

1) 对所有其他进程, 检查其是否为 blocked 状态, 若是则说明其在睡眠, 应将其 sleeptime - 1. 若某个进程 sleeptime 减到了 0, 则将其唤醒为 runnable 状态:

```
int i;  
i = (current + 1) % MAX_PCB_NUM;  
// make blocked processes sleep time -1, sleep time to 0, re-run  
for (; i != current; i = (i + 1) % MAX_PCB_NUM)  
{  
    if (pcb[i].state == STATE_BLOCKED && pcb[i].sleepTime > 0)  
    {  
        pcb[i].sleepTime--;  
        if (pcb[i].sleepTime == 0)  
            pcb[i].state = STATE_RUNNABLE; //如果sleeptime为0, 则转为就绪态  
    }  
}
```

(由于调用到时钟中断时当前进程必然处于 running 状态, 因此遍历进程表时可以排除当前进程)

2) 对各进程的 `sleeptime` 处理完后，应对当前进程的时间片进行+1 处理。若+1 后时间片小于最大时间片长度，则该进程可以继续运行，于是直接从时间中断函数中 `return`，最后回到 `doIrq.S` 中，恢复当前进程的现场，然后回到用户态继续执行：

```
// if time count not max, process continue
pcb[current].timeCount++;
if (pcb[current].timeCount < MAX_TIME_COUNT)
    return; //直接回到doIrq中善后处理
```

若+1 后时间片等于最大时间片长度，**说明当前进程应被换下，执行进程切换**。此时先把当前进程的时间片计数器置为 0，状态置为 `runnable`:

```
else
{
    // else switch to another process
    /* echo pid of selected process */

    pcb[current].timeCount = 0; //将本进程的timecount置为0
    pcb[current].state = STATE_RUNNABLE; //状态从运行中改为就绪
```

再进行普通的进程调度：期间根据注释中的要求，输出切换后的进程号：

```

//然后切换进程
for (i = (current + 1) % MAX_PCB_NUM; i != current; i = (i + 1) % MAX_PCB_NUM)
{
    if (pcb[i].state == STATE_RUNNABLE)
    {
        //find = 1;
        break;
    }
}

current = i;
pcb[current].state = STATE_RUNNING;           //切换，开始工作
putChar(pcb[current].pid + '0');              //输出进程号

/*XXX recover stackTop of selected process */
// setting tss for user process
tss.esp0 = (uint32_t)(&pcb[current].stackTop); //设置tss状态段为核心栈栈顶

// switch kernel stack: 切换至核心栈
asm volatile("movl %0, %%esp" :: "m"(pcb[current].stackTop));

//这是原有的:弹出在进程控制块中保存的现场信息
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");

```

至此，进程切换已经实现完毕。执行测试用例：



1. 选择一个状态为 `dead` 的线程, 作为当前要创建的新线程:

```
int i = 0;
for (; i < MAX_TCB_NUM; i++)
{
    if (tcb[i].state == STATE_DEAD)
        break;
}

if (i == MAX_TCB_NUM)
    return -1;
```

2. 设置新进程的参数: 状态置为 `runnable`, 线程编号 `*thread` 为 `i`, 也即在线程表中的序号。将参数放到 `tcb` 的栈顶中, 并设置 `tcb` 中的 `ebp`, `esp` 为次栈顶的位置, 以方便调度到这个新线程时程序能够正确访问参数 `arg` (调度到这个线程后, 进入函数 `start_routine()` 内, 函数的头两个语句为 `push %ebp` 和 `mov %esp, %ebp`, 然后通过 `%ebp` 指向的栈位置来访问参数 `arg`)。设置 `eip` 为该线程要执行的函数的地址:

```
tcb[i].state = STATE_RUNNABLE;
*thread = i;
tcb[i].pthArg = (uint32_t)arg;
tcb[i].pthid = i;
//tcb[i].stackTop = (uint32_t>(&tcb[i].stack[1535]));
tcb[i].stack[1535] = (uint32_t)arg; //参数入栈
//tcb[i].cont.esp = (uint32_t>(&tcb[i].stack[1535]));
tcb[i].cont.ebp = (uint32_t>(&tcb[i].stack[1534]));
tcb[i].cont.esp = (uint32_t>(&tcb[i].stack[1534]));
//asm volatile("movl %0, %%eax" :: "m"(start_routine));
tcb[i].cont.eip = (uint32_t)start_routine;
//asm volatile("jmp *%eax");

return 0;
}
```

pthread\_create()结束。

## 2. pthread\_yield()

该函数让出当前线程，切换到新的线程来执行，因此要做的事情是：

(1) 保存当前线程的线场：

```
int pthread_yield(void)
{
    //切换：1. 保存现场：2. 选择一个阻塞的线程 3. 恢复这个线程的现场
    //1. 保存现场：把相应的值放到当前进程的context里
    /*
    struct Context
    {
        uint32_t edi, esi, ebp, esp, ebx, edx, ecx, eax;
        uint32_t eip;
    };
    */
    asm volatile("movl %%edi, %0" :: "m" (tcb[current].cont.edi));
    asm volatile("movl %%esi, %0" :: "m" (tcb[current].cont.esi));
    asm volatile("movl %%ebx, %0" :: "m" (tcb[current].cont.ebx));
    asm volatile("movl %%edx, %0" :: "m" (tcb[current].cont.edx));
    asm volatile("movl %%ecx, %0" :: "m" (tcb[current].cont.ecx));
    asm volatile("movl %%eax, %0" :: "m" (tcb[current].cont.eax));

    asm volatile("movl (%ebp), %0" : "=a" (tcb[current].cont.ebp));
    asm volatile("leal 8(%ebp), %0" : "=a" (tcb[current].cont.esp));
    asm volatile("movl 4(%ebp), %0" : "=a" (tcb[current].cont.eip));

    tcb[current].state = STATE_RUNNABLE;
```

将当前线程的寄存器值保存到 TCB 内，然后置 state 为 runnable. 需要注意的是，保存的线程的“现场”中 ebp、eip、

esp 的值不是执行到该条语句时 ebp、eip、esp 的值，因为当前线程执行的函数调用了 pthread\_yield(), 故要保存的现场的值应为调用 pthread\_yield() 之前那一刻各个寄存器的值，因此这三个寄存器的值应为当前函数 (pthread\_yield()) 的栈帧中的 old ebp 的值、保存的返回地址和 %ebp + 8。

(2) 执行线程的调度：

```
//2. 选择一个阻塞的线程
//3. 恢复这个线程的现场
int val = schedule(); //不包括cur
if (val == -1)
    return -1;
return 0;
}
```

pthread\_join:

(1) 若要等待的线程已经死亡，则直接进行线程调度：



```

18
19 int pthread_join(pthread_t thread, void **retval)
20 {
21     printf("In Join\n");
22     //if (tcb[thread].state == STATE_DEAD && tcb[current].sta
23     if (tcb[thread].state == STATE_DEAD)
24     { //要等待的进程的已经死亡：此时将自己的状态改为可以运行，然后调度
25         // tcb[current].state = STATE_RUNNABLE;
26         //调度
27         int val = schedule();
28         if (val == -1)
29             return -1;
30         //printf("\n");
31     }
32

```

(2) 若要等待的线程没有死亡，则**保存当前线程的现场**，再将 joinid 设置为要等的线程的 ID，置自己的状态为 blocked，然后，再执行线程调度。保存现场时针对 ebp 等三个寄存器同样不是简单地将 CPU 中寄存器的内容复制到 TCB 中：

```

else
{
    //要等待的进程的未死亡：此时将自己的joinid设为要等的进程的ID，然后保存现场，阻塞自己，调度别的进程上来做
    asm volatile("movl %%edi, %0" :: "m"(tcb[current].cont.edi));
    asm volatile("movl %%esi, %0" :: "m"(tcb[current].cont.esi));
    asm volatile("movl %%ebx, %0" :: "m"(tcb[current].cont.ebx));
    asm volatile("movl %%edx, %0" :: "m"(tcb[current].cont.edx));
    asm volatile("movl %%ecx, %0" :: "m"(tcb[current].cont.ecx));
    asm volatile("movl %%eax, %0" :: "m"(tcb[current].cont.eax));

    asm volatile("movl (%ebp), %0" : "=a"(tcb[current].cont.ebp));
    asm volatile("leal 8(%ebp), %0" : "=a"(tcb[current].cont.esp));
    asm volatile("movl 4(%ebp), %0" : "=a"(tcb[current].cont.eip));

    tcb[current].state = STATE_BLOCKED; //阻塞自己
    tcb[current].joinid = thread;

    int val = schedule(); //不包括cur
    if (val == -1)
        return -1;
}

return 0;
}

```

`pthread_exit()`:

(1) 置当前线程状态为死亡；再逐个检查线程表，若有线程正在等待当前线程且状态为 `blocked`，则唤醒之为 `runnable`：

```
void pthread_exit(void *retval)
{
    /*
     1. 将自己的状态置为死亡
     2. 逐个查找线程表，看有那些进程join了自己：将他们置为就绪态
    */
    tcb[current].state = STATE_DEAD;

    int i;
    for (i = 0; i < MAX_TCB_NUM; i++)
    {
        if (tcb[i].joinid == tcb[current].pthid && tcb[i].state == STATE_BLOCKED)
            tcb[i].state = STATE_RUNNABLE;
    }

    schedule();
    // int val = schedule(0);
    // if (val != -1)
    //     //recover();

    return;
}
```

然后执行线程的调度。

调度函数 `schedule()`:

(1) 从 `current` 线程的在线程表中后一个位置的线程开始寻找，寻找第一个不是 `current` 线程的 `runnable` 的线程。若是整个线程表只有 `current` 线程时 `runnable` 的，则调度到当前线程，否则调度到另一个线程：

```

int schedule()
{
    int i;
    i = (current + 1) % MAX_TCB_NUM;
    for (; i != current; i = (i + 1) % MAX_TCB_NUM)
    {
        if (tcb[i].state == STATE_RUNNABLE)
            break;
    }

    if (i == current && tcb[i].state != STATE_RUNNABLE)
        return -1;

    current = i;
    tcb[current].state = STATE_RUNNING;
}

```

(2) 然后恢复线程的现场信息：使用 `jmp` 指令，直接跳转到相应的指令的位置去执行新的线程：

```

asm volatile("movl %0, %%edi" :: "m"(tcb[current].cont.edi));
asm volatile("movl %0, %%esi" :: "m"(tcb[current].cont.esi));
asm volatile("movl %0, %%ebx" :: "m"(tcb[current].cont.ebx));
asm volatile("movl %0, %%edx" :: "m"(tcb[current].cont.edx));
asm volatile("movl %0, %%ecx" :: "m"(tcb[current].cont.ecx));
asm volatile("movl %0, %%eax" :: "m"(tcb[current].cont.eax));

asm volatile("movl %0, %%ebp" :: "m"(tcb[current].cont.ebp));
asm volatile("movl %0, %%esp" :: "m"(tcb[current].cont.esp));
//asm volatile("movl %0, %%eax" :: "m"(tcb[current].cont.eip));
//asm volatile("jmp *%eax");
asm volatile("jmp *%0" :: "m"(tcb[current].cont.eip));

return 5;
}

```

至此，线程的调度实现完毕。实现效果：

```
QEMU
In Join
Ping@1-1
child Process: 2, 7;
Ping@2-2
Pong@3
Ping@4-1
Ping@5-2
Pong@6
Ping@7-1
Ping@8-2
Pong@9
child Process: 2, 6;
Ping@10-1
Ping@11-2
Pong@12
Ping@13-1
Ping@14-2
Pong@15
Ping@16-1
```

```
QEMU
Ping@10-1
Ping@11-2
Pong@12
Ping@13-1
Ping@14-2
Pong@15
Ping@16-1
Ping@17-2
child Process: 2, 5;
Ping@18-1
Ping@19-2
Ping@20-1
Ping@21-2
Ping@22-1
Ping@23-2
Ping@24-1
Ping@25-2
child Process: 2, 4;
In Join
In Join
child Process: 2, 3;
child Process: 2, 2;
child Process: 2, 1;
child Process: 2, 0;
_
```

