Lab3实验报告

几个系统调用的编号

> ../lib/lib.h

```
#define SYS_WRITE 0
#define SYS_FORK 1
#define SYS_EXEC 2
#define SYS_SLEEP 3
#define SYS_EXIT 4
```

填好相应的系统函数的调用

> ../lib/syscall.c

```
pid_t fork()
{
    // TODO:call syscall
    return syscall(SYS_FORK, 0, 0, 0, 0, 0);
}

int sleep(uint32_t time)
{
    // TODO:call syscall
    return syscall(SYS_SLEEP, time, 0, 0, 0, 0);
}

int exit()
{
    // TODO:call syscall
    return syscall(SYS_EXIT, 0, 0, 0, 0, 0);
}
```

接下来是中断处理

> ../kernel/kernel/irqHandle.c

irqHandle函数对比lab2增加了部分保存与恢复的内容：

将当前内核栈顶用tmp保存，保存为上次栈顶，接着改为新的中断栈来处理中断，最后切换回来

```
void irqHandle(struct StackFrame *sf)
{ // pointer sf = esp
    /* Reassign segment register */
    asm volatile("movw %%ax, %%ds" ::"a"(KSEL(SEG_KDATA)));
```

```
    /*XXX Save esp to stackTop */
+   uint32_t tmpStackTop = pcb[current].stackTop;
+   pcb[current].prevStackTop = pcb[current].stackTop;
+   pcb[current].stackTop = (uint32_t)sf;

    switch (sf->irq)
    {
    case -1:
        break;
    case 0xd:
        GProtectFaultHandle(sf);
        break;
    case 0x20:
        timerHandle(sf);
        break;
    case 0x80:
        syscallHandle(sf);
        break;
    default:
        assert(0);
    }
    /*XXX Recover stackTop */
+   pcb[current].stackTop = tmpStackTop;
}
```

timerHandle

> 时间中断到来后，两个用户态进程 P1、 P2 进行进程切换的流程如下
>
> 1. 进程P1在用户态执行，8253可编程计时器产生时间中断
> 2. 依据TSS中记录的进程P1的 SS0:EPS0， 从P1的用户态堆栈切换至P1的内核堆栈，并将P1的现场 信息压入内核堆栈中，跳转执行时间中断处理程序
> 3. 进程P1的处理时间片耗尽，切换至就绪状态的进程P2，并从当前P1的内核堆栈切换至P2的内核堆 栈
> 4. 从进程P2的内核堆栈中弹出P2的现场信息，切换至P2的用户态堆栈，从时间中断处理程序返回执 行P2

时钟中断功能:

1. 遍历pcb，将状态为STATE_BLOCKED的进程的sleepTime减一，如果进程的sleepTime变为0，重新设为STATE_RUNNABLE
2. 将当前进程的timeCount加一，如果时间片用完（timeCount==MAX_TIME_COUNT）且有其它 状态为STATE_RUNNABLE的进程，切换，否则继续执行当前进程

```
void timerHandle(struct StackFrame *sf)
{
    // TODO
    for(int i = 0; i < MAX_PCB_NUM; i++){
        if(pcb[i].state == STATE_BLOCKED){
            pcb[i].sleepTime--;
            if(pcb[i].sleepTime == 0){
                pcb[i].state = STATE_RUNNABLE;
```

```
            }
        }
    }

    if(pcb[current].state == STATE_RUNNING){
        pcb[current].timeCount++;
        if (pcb[current].timeCount < MAX_TIME_COUNT){
            return;
        }
        else{
            pcb[current].timeCount = 0;
            pcb[current].state = STATE_RUNNABLE;
        }
    }

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

    pcb[current].state = STATE_RUNNING;
    switch_between_process();
}
```

这时的current已经改成新值了，但是还需要进行一些进程间切换的操作：

> 每个用户进程的内核堆栈也是不一样的，所以每次切换进程时需要将tss的esp0设置对应用户进程的内核堆栈位置

```
void switch_between_process(){
    //从当前P1的内核堆栈切换至P2的内核堆栈
    uint32_t tmpStackTop = pcb[current].stackTop;
    pcb[current].stackTop = pcb[current].prevStackTop;
    tss.esp0 = (uint32_t)&(pcb[current].stackTop);
    asm volatile("movl %0, %%esp"::"m"(tmpStackTop));
    //从进程P2的内核堆栈中弹出P2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");
    //切换至P2的用户态堆栈
    asm volatile("iret");
    //返回执行P2
}
```

根据前面系统调用的编号，在syscallHandle处填入fork sleep和exit的分支

```
void syscallHandle(struct StackFrame *sf)
{
    switch (sf->eax)
    { // syscall number
    case 0:
        syscallWrite(sf);
        break; // for SYS_WRITE
    /*TODO Add Fork,Sleep... */
    case 1:
        syscallFork(sf);
        break;
    case 3:
        syscallSleep(sf);
        break;
    case 4:
        syscallExit(sf);
        break;
    default:
        break;
    }
}
```

记得在最前面添加timerHandle, syscallFork, syscallSleep, syscallExit的声明

syscallFork

syscallFork要做的是在寻找一个空闲的pcb做为子进程的进程控制块，将父进程的资源复制给子进程。如果没有空闲pcb，则fork失败，父进程返回-1，成功则子进程返回0，

父进程返回子进程pid 在处理fork时有以下几点注意事项：

1. 代码段和数据段可以按照2.4.1.节最后的说明进行完全拷贝

    > 在实验3中默认用户进程起始地址为 0x200000 ，每个进程占用 0x100000 大小的内存。通过设置不同的段基址来隔离用户进程在内存中的位置，这样理论上可以通过切换段选择子的值进行用户进程的切换
    >
    > 在实验中，采用线性表的方式组织pcb，也就是将pcb以数组形式连续存放，为了简单起见，可以将pcb 的pid设为其索引。内核进程会占据0号pcb，剩下的分配给用户进程。同样为了简单，我们默认每个 pcb对应进程的内存空间固定， pcb[i] 对应的内存起始地址为 (i + 1) * 0x100000 ，大小为 0x100000

    因此1号用户进程的内存空间为0x200000 - 0x300000，2号进程的内存空间为0x300000 - 0x400000，以此类推

2. pcb的复制时，需要考虑哪些内容可以直接复制，哪些内容通过计算得到，哪些内容和父进程无关：

    直接复制：regs中的大多数（di, esi, ebp, xxx, ebx, edx, ecx, eax, irq, error, eip, eflags, esp）

    计算得到：

    stackTop、prevStackTop：当前栈顶 - 当前地址 + 目标地址 （即偏移量不变，基准地址平移）；

regs中的 cs, ss, ds, es, fs, gs：根据 ../kernel/kerne/kvm.c 1号用户进程的 cs 为USEL(3)，ss, ds, es, fs, gs 为USEL(4)，以此类推， `i` 号用户进程的 cs 为 `USEL(2*i+1)` ，ss, ds, es, fs, gs 为 `USEL(2*i+2)`

与父进程无关：state、timeCount、sleepTime、pid。

3. 返回值放在哪

   放在eax寄存器中

提示： initProc 中有初始化 pcb[0] 和 pcb[1] 的经验可供参考

参考：PCB和StackFrame的结构

> ../kernel/include/x86/memory.h

```
struct StackFrame {
    uint32_t gs, fs, es, ds;
    uint32_t edi, esi, ebp, xxx, ebx, edx, ecx, eax;
    uint32_t irq, error;
    uint32_t eip, cs, eflags, esp, ss;
};

struct ProcessTable {
    uint32_t stack[MAX_STACK_SIZE];
    struct StackFrame regs;
    uint32_t stackTop;
    uint32_t prevStackTop;
    int state;
    int timeCount;
    int sleepTime;
    uint32_t pid;
    char name[32];
};
```

```
void syscallFork(struct StackFrame* sf){
    putStr("syscallFork\n");

    int new_index = 0;
    while(new_index < MAX_PCB_NUM && pcb[new_index].state != STATE_DEAD){
        new_index++;
    }
    if(new_index == MAX_PCB_NUM){
        pcb[current].regs.eax = -1;
        return;
    }

    for(int i = 0; i < 0x100000; i++){
        *(unsigned char *)(i + (new_index + 1) * 0x100000) = *(unsigned char *)(i
+ (current + 1) * 0x100000);
    }

    pcb[new_index].pid = new_index;
    pcb[new_index].state = STATE_RUNNABLE;
```

```
    pcb[new_index].timeCount = 0;
    pcb[new_index].sleepTime = 0;

    pcb[new_index].stackTop = pcb[current].stackTop - (uint32_t)&(pcb[current]) +
(uint32_t)&(pcb[new_index]);
    pcb[new_index].prevStackTop = pcb[current].prevStackTop - (uint32_t)&
(pcb[current]) + (uint32_t)&(pcb[new_index]);

    pcb[new_index].regs.edi = pcb[current].regs.edi;
    pcb[new_index].regs.esi = pcb[current].regs.esi;
    pcb[new_index].regs.ebp = pcb[current].regs.ebp;
    pcb[new_index].regs.xxx = pcb[current].regs.xxx;
    pcb[new_index].regs.ebx = pcb[current].regs.ebx;
    pcb[new_index].regs.edx = pcb[current].regs.edx;
    pcb[new_index].regs.ecx = pcb[current].regs.ecx;
    pcb[new_index].regs.eax = pcb[current].regs.eax;
    pcb[new_index].regs.irq = pcb[current].regs.irq;
    pcb[new_index].regs.error = pcb[current].regs.error;
    pcb[new_index].regs.eip = pcb[current].regs.eip;
    pcb[new_index].regs.eflags = pcb[current].regs.eflags;
    pcb[new_index].regs.esp = pcb[current].regs.esp;

    pcb[new_index].regs.cs = USEL(2*new_index + 1);
    pcb[new_index].regs.ss = USEL(2*new_index + 2);
    pcb[new_index].regs.ds = USEL(2*new_index + 2);
    pcb[new_index].regs.es = USEL(2*new_index + 2);
    pcb[new_index].regs.fs = USEL(2*new_index + 2);
    pcb[new_index].regs.gs = USEL(2*new_index + 2);

    pcb[current].regs.eax = new_index;
    pcb[new_index].regs.eax = 0;
    return;
}
```

syscallSleep

sleep系统调用用于进程主动阻塞自身，内核需要将该进程由 RUNNING 状态转换为 BLOCKED 状态，设置 该进程的 SLEEP 时间片，并切换运行其他 RUNNABLE 状态的进程

将当前的进程的sleepTime设置为传入的参数，将当前进程的状态设置为STATE_BLOCKED，然后利用

```
asm volatile("int $0x20");
```

模拟时钟中断，利用 timerHandle 进行进程切换

需要注意的是判断传入参数的合法性

```c
void syscallSleep(struct StackFrame *sf){
    putStr("syscallSleep\n");

    pcb[current].state = STATE_BLOCKED;
    assert(sf->ecx > 0);
    pcb[current].sleepTime = sf->ecx;
    asm volatile("int $0x20");
}
```

syscallExit

exit系统调用用于进程主动销毁自身，内核需要将该进程由 RUNNING 状态转换为 DEAD 状态，回收分配 给该进程的内存、进程控制块等资源，并切换运行其他 RUNNABLE 状态的进程

将当前进程的状态设置为STATE_DEAD，然后模拟时钟中断进行进程切换

```c
void syscallExit(struct StackFrame *sf){
    putStr("syscallExit\n");

    pcb[current].state = STATE_DEAD;
    asm volatile("int $0x20");
}
```

测试

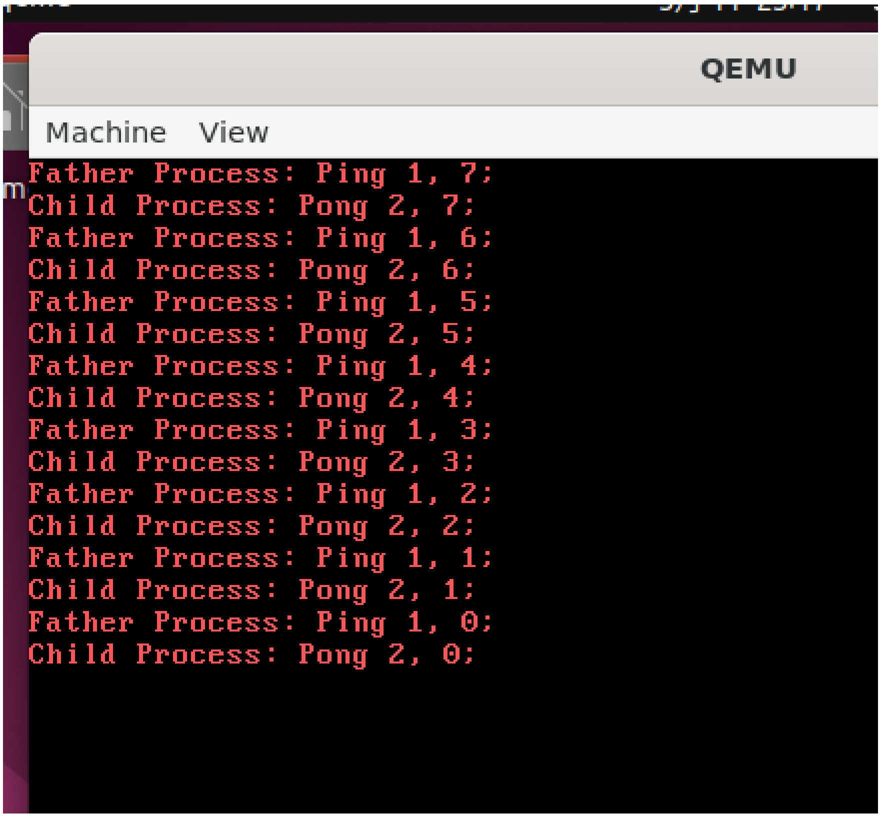
> ../app/main.c

```c
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();
    }
    while (1)
        ;
    return 0;
}
```

主进程先复制一份子进程，和它具有相同且独立的 i，然后主进程运行 `else if (ret != -1)` 分支，把 i 减为7，输出 `Ping1, 7` 并 sleep 128个时钟中断；接着子进程运行 `if (ret == 0)` 分支，把 i 减为7，输出 `Pang  2, 7` ，并sleep128个时钟中断；接着主进程醒来，在while循环中继续进行 i 的递减，输出 `Ping1, 6` 然后sleep，然后子进程醒来 .....

（128个时钟中断大概是1s左右？）



附录：一些额外的修改

> ../bootloader/boot.c

```
    // offset = ((struct ProgramHeader *)(elf + phoff))->off;
```

> ../bootloader/Makefile

```
    objcopy  -S -j .text  -O binary bootloader.elf bootloader.bin
    chmod +x ../utils/genBoot.pl
```

> ../kernel/Makefile

```
    chmod +x ../utils/genKernel.pl
```

> ../kernel/device/serial.h

```
void putStr(char *);
```

> ../kernel/kernel/serial.c

```
void putStr(char *ch){
    while(ch && (*ch) && (*ch)!='\0'){
        putChar(*ch);
        ch++;
    }
}
```

> ../Makefile

```
run:
    make clean
    make os.img
    make play
```