

OS Lab

Lab3

191220008 陈南瞳

924690736@qq.com

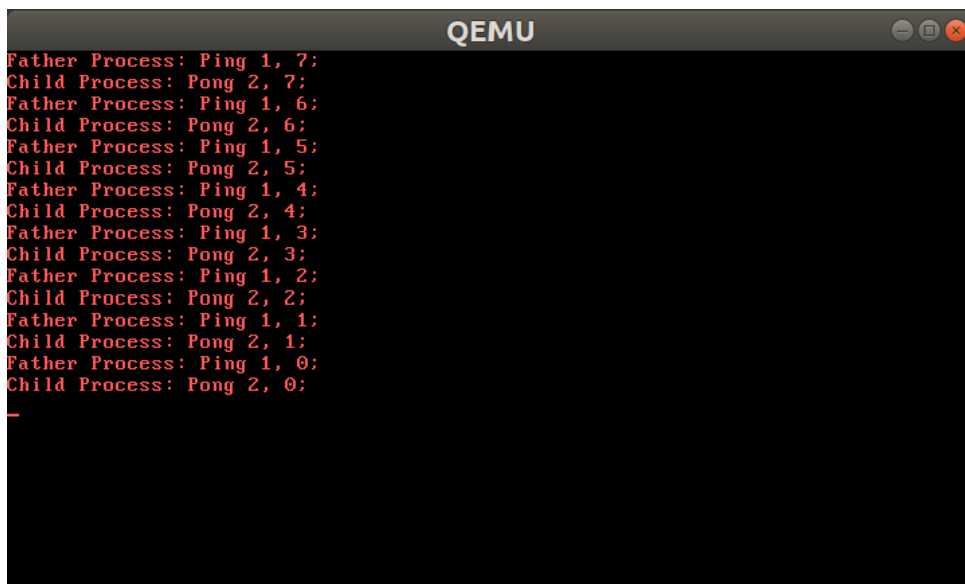
一、实验进度

完成了Lab3中的所有内容：

- 完成库函数
- 时钟中断处理
- 系统调用例程
- 中断嵌套
- 临界区

二、实验结果

(一) 开启/不开启嵌套中断



```
QEMU
Father Process: Ping 1, 7;
Child Process: Pong 2, 7;
Father Process: Ping 1, 6;
Child Process: Pong 2, 6;
Father Process: Ping 1, 5;
Child Process: Pong 2, 5;
Father Process: Ping 1, 4;
Child Process: Pong 2, 4;
Father Process: Ping 1, 3;
Child Process: Pong 2, 3;
Father Process: Ping 1, 2;
Child Process: Pong 2, 2;
Father Process: Ping 1, 1;
Child Process: Pong 2, 1;
Father Process: Ping 1, 0;
Child Process: Pong 2, 0;
```

(二) 临界区更新一致性

```
QEMU
Father PChild Process: Process: Ping 1, ong 2, 7;
7;
Child Father ProcesProcess: Pong 2,s: Ping 1, 6;
6;
FaChild Procesther Process: Pis: Pong 2, 5;
ng 1, 5;
ChFather ild Process: PonProcess: Ping 1,g 2, 4;
4;
Child PrFather Proceocess: Pong 2, 3ss: Ping 1, 3;
;
FChild Process:ather Process: P Pong 2, 2;
ing 1, 2;
ChilFatherd Process: Pong Process: Ping 12, 1;
, 1;
Child ProcFather Process: Pong 2, 0;
ess: Ping 1, 0;
```

三、实验修改的代码位置

(一) 完成库函数

在 `lib/syscall.c` 中，通过调用 `syscall` 函数，完成对库函数 `fork`，`sleep` 和 `exit` 的封装。

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

int sleep(uint32_t time) {
    return syscall(SYS_SLEEP, time, 0, 0, 0, 0);
}

int exit() {
    return syscall(SYS_EXIT, 0, 0, 0, 0, 0);
}
```

(二) 时钟中断处理

在 `kernel/kernel/irqHandle.c` 中，需要定义时钟中断处理函数 `timeHandle`。

在此之前，需要在函数 `irqHandle` 中添加部分保护与恢复的内容，并开启时钟中断的调用入口。

```
void irqHandle(struct StackFrame *sf) { // pointer sf = esp
    ...
    /*TODO 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 0x20:
```

```

        timerHandle(sf);
        break;
        ...
    }
    /*TODO Recover stackTop */
    pcb[current].stackTop = tmpStackTop;
}

```

进程切换有两种情况，一个是 sleep 相关（阻塞或恢复）；一个是进程时间片用完切换到下一个进程，其中 sleep 又可以转化为时间片用完，所以时钟中断处理 timerHandle 处是一个绝佳的进行进程切换的场所。

因此，考虑两个操作：

- 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。
- 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。

```

// 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
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;
    }
}
// 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
if(pcb[current].state == STATE_RUNNING) {
    pcb[current].timeCount++;
    if (pcb[current].timeCount >= MAX_TIME_COUNT) {
        pcb[current].state = STATE_RUNNABLE;
        pcb[current].timeCount = 0;
    }
    else
        return;
}
int i;
for (i = (current + 1) % MAX_PCB_NUM; i != current; i = (i + 1) % MAX_PCB_NUM) {
    if (pcb[i].state == STATE_RUNNABLE)
        break;
}
current = i;
pcb[current].state = STATE_RUNNING;

```

最后再使用教程中的代码进行进程切换。

```
// 进程切换
uint32_t tmpStackTop;
tmpStackTop = pcb[current].stackTop;
pcb[current].stackTop = pcb[current].prevStackTop;
tss.esp0 = (uint32_t)&(pcb[current].stackTop);
asm volatile("movl %0, %%esp"::"m"(tmpStackTop)); // switch kernel stack
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");
```

(三) 系统调用例程

1、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，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。

```
// 在寻找一个空闲的pcb做为子进程的进程控制块
int i;
for (i = 0; i < MAX_PCB_NUM; i++) {
    if (pcb[i].state == STATE_DEAD)
        break;
}
// 如果没有空闲pcb，则fork失败，父进程返回-1
if (i == MAX_PCB_NUM) {
    pcb[current].regs.eax = -1;
    return;
}
...
// 成功则子进程返回0
pcb[i].regs.eax = 0;
...
// 成功则父进程返回子进程pid
pcb[current].regs.eax = i;
```

然后将父进程的资源复制给子进程。

```
// 将父进程的资源复制给子进程
enableInterrupt();
for (int j = 0; j < 0x100000; j++) {
    *(uint8_t*)(j + (i + 1) * 0x100000) = *(uint8_t*)(j + (current + 1) * 0x100000);
    if (j % 10000 == 0)
        asm volatile("int $0x20"); //XXX Testing irqTimer during syscall
}
disableInterrupt();
```

根据 kvm.c 中函数 initProc 里初始化 pcb[0] 和 pcb[1]的方法，对pcb中的内容逐一进行复制，返回值放在eax中。

```

struct ProcessTable {
    uint32_t stack[MAX_STACK_SIZE]; // 内核堆栈
    struct TrapFrame regs; // 陷阱帧，保存上下文
    uint32_t stackTop; // 保存内核栈顶信息
    uint32_t prevStackTop; // 中断嵌套时保存待恢复的栈顶信息
    int state; // 进程状态: STATE_RUNNABLE、STATE_RUNNING、STATE_BLOCKED、STATE_DEAD
    int timeCount; // 当前进程占用的时间片
    int sleepTime; // 当前进程需要阻塞的时间片
    uint32_t pid; // 进程的唯一标识
    char name[32]; // not used
};

```

```

struct TrapFrame {
    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;
};

```

2、syscallSleep

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`，然后利用模拟时钟中断，利用 `timerHandle` 进行进程切换，需要注意的是首先需要判断传入参数的合法性。

```

void syscallSleep(struct StackFrame *sf) {
    // 若传入的参数合法，则将当前的进程的sleepTime设置为传入的参数，将当前进程的状态设置为STATE_BLOCKED
    if (sf->ecx > 0) {
        pcb[current].sleepTime = sf->ecx;
        pcb[current].state = STATE_BLOCKED;
        // 模拟时钟中断
        asm volatile("int $0x20");
    }
    return;
}

```

3、syscallExit

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。

```

void syscallExit(struct StackFrame *sf) {
    // 将当前进程的状态设置为STATE_DEAD
    pcb[current].state = STATE_DEAD;
    // 模拟时钟中断
    asm volatile("int $0x20");
    return;
}

```

(四) 中断嵌套

在 `syscallFork` 中进行内存拷贝的原始代码为：

```
for (j = 0; j < 0x100000; j++) {
    *(uint8_t *) (j + (i + 1) * 0x100000) = *(uint8_t *) (j + (current + 1) *
0x100000);
}
```

代码的作用就是将 `current` 进程的内存空间拷贝到进程 `i` 的空间。

为了能在拷贝内存空间时就开启了嵌套中断，并防止由于 `cpu` 速度过快，可能来不及嵌套中断，代码就执行完了的情况，需要手动模拟时钟中断。

```
// 将父进程的资源复制给子进程
enableInterrupt();
for (int j = 0; j < 0x100000; j++) {
    *(uint8_t *) (j + (i + 1) * 0x100000) = *(uint8_t *) (j + (current + 1) *
0x100000);
    if (j % 0x10000 == 0)
        asm volatile("int $0x20"); //XXX Testing irqTimer during syscall
}
disableInterrupt();
```

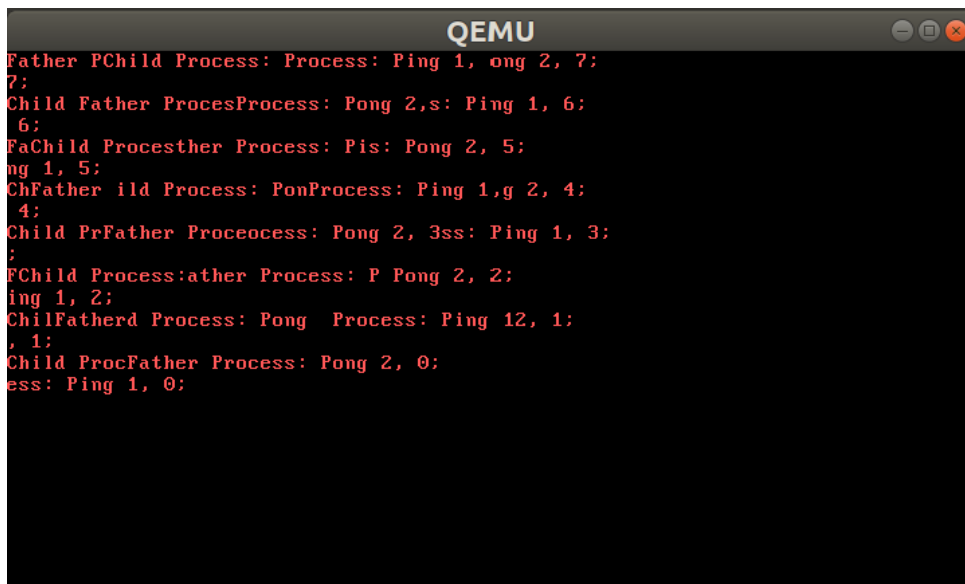
在实际操作过程中发现在每次循环都调用时钟中断会导致长时间的等待，因此改为每 `0x10000` 次循环调用一次时钟中断处理，便能正常快速的运行。

(五) 临界区

在函数 `syscallPrint` 中利用 `int $0x20` 指令主动陷入时间中断来模拟多个进程并发地处理系统调用，对共享资源（例如内核的数据结构，视频显存等等）进行竞争的场景。

```
void syscallPrint(struct StackFrame *sf) {
    ...
    asm volatile("int $0x20"); //XXX Testing irqTimer during syscall
    //asm volatile("int $0x20:::"memory"); //XXX Testing irqTimer during
syscall
    ...
}
```

验证结果：



```
QEMU
Father PChild Process: Process: Ping 1, ong 2, 7;
7;
Child Father ProcesProcess: Pong 2,s: Ping 1, 6;
6;
FaChild Procesther Process: Pis: Pong 2, 5;
ong 1, 5;
ChFather ild Process: PonProcess: Ping 1,g 2, 4;
4;
Child PrFather Proceocess: Pong 2, 3ss: Ping 1, 3;
;
FChild Process:ather Process: P Pong 2, 2;
ing 1, 2;
ChilFatherd Process: Pong Process: Ping 12, 1;
, 1;
Child ProcFather Process: Pong 2, 0;
ess: Ping 1, 0;
```

可知，多个进程并发地进行系统调用，对共享资源进行竞争可能会产生一致性问题，带来未知的BUG；因此，在系统调用过程中，对于临界区的代码不宜开启外部硬件中断，而对于非临界区的代码，则可以开启外部硬件中断，允许中断嵌套。

四、遇到的问题和对这些问题的思考

1、timeHandle的处理

起初跑测试用例时，一直没有输出，后来在检查 timerHandle 时发现，函数的处理逻辑有一些问题。

按照要求应该是只有时间片用完时才会进程切换，但我一开始并没有对不是时间片用完的情况做处理，在不是时间片用完的情况时，直接return后，问题得以解决。

```
// 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
if(pcb[current].state == STATE_RUNNING) {
    pcb[current].timeCount++;
    if (pcb[current].timeCount >= MAX_TIME_COUNT) {
        pcb[current].state = STATE_RUNNABLE;
        pcb[current].timeCount = 0;
    }
    else
        return; // *****修改位置*****
}
```

2、嵌套中断的处理

起初，也是一直没有输出，后来，观察到循环的次数为 0x100000 次，在每次循环都调用时钟中断会长时间的等待，所以修改为了每 0x10000 次循环调用一次时钟中断，程序便可正常运行。

```
// 将父进程的资源复制给子进程
enableInterrupt();
for (int j = 0; j < 0x100000; j++) {
    *(uint8_t*)(j + (i + 1) * 0x100000) = *(uint8_t*)(j + (current + 1) *
0x100000);
    if (j % 0x10000 == 0) // *****修改位置*****
        asm volatile("int $0x20"); //XXX Testing irqTimer during syscall
}
disableInterrupt();
```

五、实验心得或对提供帮助的同学的感谢

本次实验相较于lab2来说变得清晰了许多，难度也有所降低，实验教程比较有条理，并且要求比较明确，比lab2更容易让人理解，所以做实验的效率也提高了不少。

整个实验做下来，对时钟中断与进程切换的关系，以及三个系统调用（syscallFork，syscallSleep，syscallExit）有了更加深刻的理解，收获颇多。