课 程 实 验 报 告

实验名称 进程调度

课程名称	操作系统
院　　系	计算机科学与技术系
学　　号	191220129
姓　　名	邢尚禹
邮　　箱	191220129@smail.nju.edu.cn
实验日期	2021 年 4 月

目录

1 实验进度和批改注意事项

已完成所有内容，包括选做。

批改时希望能关注这一点：框架代码的 bootloader **无法**在我的环境中正确运行（ubuntu20.04 + gcc 9.3），必须注释 bootMain 函数的以下两行（第 16，17 行）：

```
1  void bootMain(void)
2  {
3          int i = 0;
4          // int phoff = 0x34;
5          int offset = 0x1000;
6          unsigned int elf = 0x100000;
7          void (*kMainEntry)(void);
8          kMainEntry = (void (*)(void))0x100000;
9  
10         for (i = 0; i < 200; i++)
11         {
12                 readSect((void *)(elf + i * 512), 1 + i);
13         }
14 
15         kMainEntry = (void (*)(void))((struct ELFHeader *)elf)->entry;
16         // phoff = ((struct ELFHeader *)elf)->phoff;
17         // offset = ((struct ProgramHeader *)(elf + phoff))->off;
18 
19         for (i = 0; i < 200 * 512; i++)
20         {
21                 *(unsigned char *)(elf + i) = *(unsigned char *)(elf + i + offset);
22         }
23 
24         kMainEntry();
25 }
```

我猜想可能是由于 gcc 版本的问题，如果批改时不能正确运行，请将上述两行取消注释再尝试编译。

另外，在 kernel/irqHandle.c 中有几个宏，可以通过定义和取消定义实现对不同内容的测试。

- LOG：从 serial 输出一些调试信息，如执行系统调用，进程切换，阻塞和解除阻塞等；
- TEST_MULTI_IRQ：测试中断嵌套；
- TEST_COMPETE_DISPLAY：测试共享资源竞争相关内容。

2 实验思路和过程

2.1 完成库函数

直接调用 syscall 函数，通过宏选择系统调用类型，再传入参数即可。

```
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);
}
```

另外，为调试方便，需要自己补充一些库函数。由于框架代码已给出了 putChar 函数，它可以将指定的字符从串口输出。由此可以封装自己的 log 函数和 logint 函数，供调试使用。

```
void log(const char *str)
{
    for (int i = 0; i < 100 && str[i]; ++i)
        putChar(str[i]);
}
```

```c

void logint(uint32_t num)
{
    if (!num)
        putChar('0');
    char buf[12];
    int i = 0;
    for (; num; ++i)
    {
        buf[i] = (char)(num % 10 + '0');
        num /= 10;
    }
    while (--i >= 0)
        putChar(buf[i]);
}
```

2.2 时钟中断处理

在时钟中断到来时，要完成以下的处理：

- 更新被阻塞的进程的状态；
- 更新当前进程的时间片使用情况，确定是否需要切换进程；
- 进程切换。

状态更新、确定是否需要切换进程可以这样实现：

```c
for (int i = 0; i < MAX_PCB_NUM; ++i)
{
        if (pcb[i].state == STATE_BLOCKED && !--pcb[i].sleepTime)
        {
                pcb[i].state = STATE_RUNNABLE;
        }
}
if (pcb[current].state != STATE_RUNNING || ++pcb[current].timeCount >= MAX_TIME_COUNT)
{
```

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

进程切换可以直接采用指导文档给出的代码。

2.3 系统调用例程

2.3.1 fork

fork 需要实现以下内容：

- 拷贝 pcb 信息和用户栈；
- 设置 pcb 中只与子进程相关的信息，如 pid，内核栈指针，段寄存器等；
- 准备返回值，放入对应进程的 pcb 中的 eax 中。

```
void syscallFork(struct StackFrame *sf)
{
        int i = 0;
        for (; i < MAX_PCB_NUM; ++i)
                if (pcb[i].state == STATE_DEAD)
                        break;
        if (i != MAX_PCB_NUM)
        {
#ifdef TEST_MULTI_IRQ
                enableInterrupt();
                for (int j = 0; j < 0x100000; j++)
                {
                        *(uint8_t *)(j + (i + 1) * 0x100000) =
                            *(uint8_t *)(j + (current + 1) * 0
                            x100000);
```

```
			if (!(j % 0x1000))
				asm volatile("int $0x20");
		}
		disableInterrupt();
#else
		memcpy((void *)((i + 1) * 0x100000), (void *)((current + 1) * 0x100000), 0x100000);
#endif

		memcpy(&pcb[i], &pcb[current], sizeof(ProcessTable));
		// pcb[i] = pcb[current];
		pcb[i].stackTop = (uint32_t) & (pcb[i].regs);
		pcb[i].prevStackTop = (uint32_t) & (pcb[i].stackTop);
		pcb[i].state = STATE_RUNNABLE;
		pcb[i].timeCount = 0;
		pcb[i].sleepTime = 0;
		pcb[i].pid = i;

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

		pcb[i].regs.eax = 0;
		pcb[current].regs.eax = i;
	}
	else
		pcb[current].regs.eax = -1;
}
```

2.3.2 sleep

函数 sleep 的实现并不困难，只需要将当前状态设置为 STATE_BLOCKED，并设置 sleepTime，再模拟时钟中断即可。注意要检查参数是否合法。

```
void syscallSleep(struct StackFrame *sf)
{
        if (sf->ecx <= 0)
                log("sleep time not positive!\n");
        else
        {
                pcb[current].state = STATE_BLOCKED;
                pcb[current].sleepTime = sf->ecx;
                asm volatile("int $0x20");
        }
```

2.3.3 exit

和 sleep 类似，只需要将当前状态设置为 STATE_DAED，并模拟时钟中断。

```
void syscallExit(struct StackFrame *sf)
{
        pcb[current].state = STATE_DEAD;
        asm volatile("int $0x20");
}
```

3 实验结果

3.1 进程切换及相关系统调用

kernel 可以加载用户程序运行，并进行正确的进程切换、系统调用。运行结果如下：

```
sleep blocked 2
process switch: 2 -> 0
unblocked 1
process switch: 0 -> 1
sleep blocked 1
unblocked 2
process switch: 1 -> 2
sleep blocked 2
process switch: 2 -> 0
unblocked 1
process switch: 0 -> 1
sleep blocked 1
unblocked 2
process switch: 1 -> 2
sleep blocked 2
process switch: 2 -> 0
unblocked 1
process switch: 0 -> 1
sleep blocked 1
unblocked 2
process switch: 1 -> 2
sleep blocked 2
process switch: 2 -> 0
unblocked 1
process switch: 0 -> 1
sleep blocked 1
unblocked 2
process switch: 1 -> 2
sleep blocked 2
process switch: 2 -> 0
unblocked 1
process switch: 0 -> 1
sleep blocked 1
unblocked 2
process switch: 1 -> 2
sleep blocked 2
process switch: 2 -> 0
unblocked 1
process switch: 0 -> 1
exit
unblocked 2
process switch: 1 -> 2
exit
process switch: 2 -> 0
```

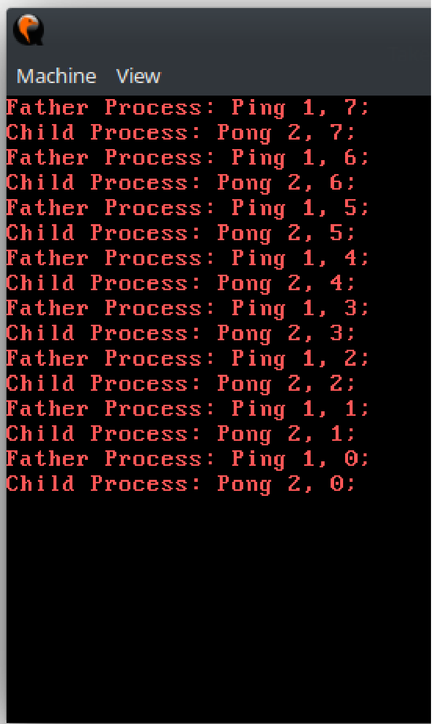

图 1: 进程切换及相关系统调用运行结果

3.2 中断嵌套

在 fork 中原始的用户栈复制代码中插入模拟时钟中断，实现中断的嵌套：

```
enableInterrupt();
for (int j = 0; j < 0x100000; j++)
{
        *(uint8_t *)(j + (i + 1) * 0x100000) = *(uint8_t *)(j +
            (current + 1) * 0x100000);
        if (!(j % 0x1000))
                asm volatile("int $0x20");
}
disableInterrupt();
```

由于中断处理开销较大，如果在 0x100000 次循环中每次均插入一个时钟中断，运行非常缓慢。因此设置为每 0x1000 次插入一个时钟中断。运行结果如下：

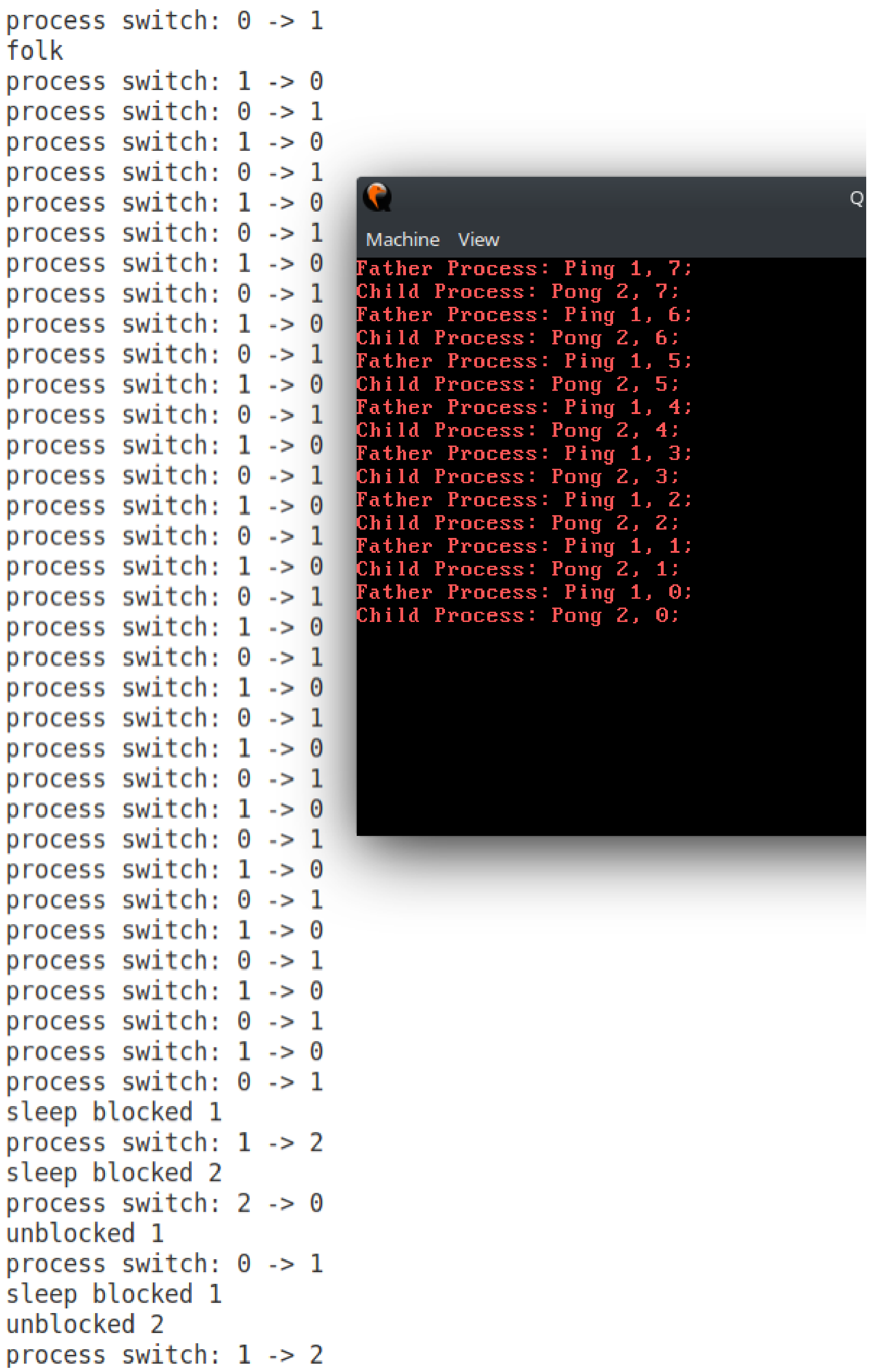
process switch: 0 -> 1
folk
process switch: 1 -> 0
process switch: 0 -> 1
process switch: 1 -> 0
process switch: 0 -> 1
process switch: 1 -> 0
process switch: 0 -> 1
process switch: 1 -> 0
process switch: 0 -> 1
process switch: 1 -> 0
process switch: 0 -> 1
process switch: 1 -> 0
process switch: 0 -> 1
process switch: 1 -> 0
process switch: 0 -> 1
process switch: 1 -> 0
process switch: 0 -> 1
process switch: 1 -> 0
process switch: 0 -> 1
process switch: 1 -> 0
process switch: 0 -> 1
process switch: 1 -> 0
process switch: 0 -> 1
process switch: 1 -> 0
process switch: 0 -> 1
process switch: 1 -> 0
process switch: 0 -> 1
process switch: 1 -> 0
process switch: 0 -> 1
process switch: 1 -> 0
process switch: 0 -> 1
process switch: 1 -> 0
process switch: 0 -> 1
process switch: 1 -> 0
process switch: 0 -> 1
sleep blocked 1
process switch: 1 -> 2
sleep blocked 2
process switch: 2 -> 0
unblocked 1
process switch: 0 -> 1
sleep blocked 1
unblocked 2
process switch: 1 -> 2
Machine View
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;

图 2: 中断嵌套运行结果

可见程序可以很好地支持中断的嵌套。

3.3 共享资源竞争

在 syscallPrint 函数中循环的最后添加一个模拟时钟中断，可以使多个 syscallPrint 函数并发执行。

```
for (i = 0; i < size; i++)
{
        /* print and update curser */
#ifdef TEST_COMPETE_DISPLAY
        asm volatile("int $0x20"); //XXX Testing irqTimer during syscall
#endif
```

运行结果如下：

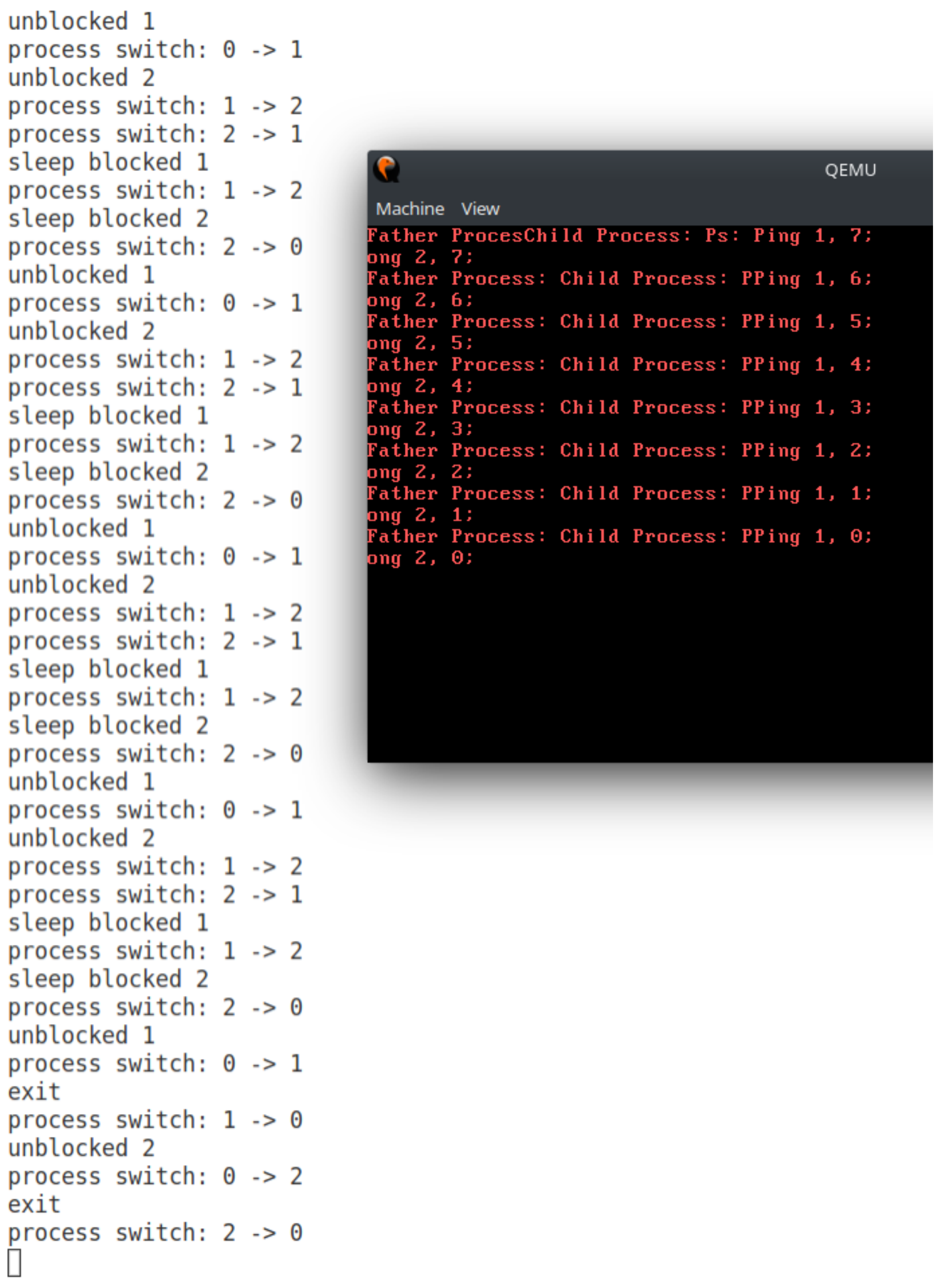

图 3: 共享资源竞争运行结果

可以看到，由于共享变量 displayRow 和 displayCol 被多个并发进程同时访问和修改，出现了打印上的问题。

将相关代码反汇编，可以看到原本的执行流被 int 打断，再返回时，displayRow 和 displayCol 被修改，就会出现错误。

```
100918: 66 0f be d2             movsbw %dl,%dx
10091c: 80 ce 0c                or     $0xc,%dh
10091f: 8d 04 9b                lea    (%ebx,%ebx,4),%eax
100922: c1 e0 04                shl    $0x4,%eax
100925: 03 07                   add    (%edi),%eax
100927: 01 c0                   add    %eax,%eax
100929: 05 00 80 0b 00          add    $0xb8000,%eax
10092e: 66 89 10                mov    %dx,(%eax)
100931: 8b 07                   mov    (%edi),%eax
100933: 40                      inc    %eax
100934: 89 07                   mov    %eax,(%edi)
100936: 83 f8 50                cmp    $0x50,%eax
100939: 74 12                   je     10094d <syscallPrint+0x7d>
10093b: cd 20                   int    $0x20
10093d: 46                      inc    %esi
10093e: 3b 75 d4                cmp    -0x2c(%ebp),%esi
100941: 74 40                   je     100983 <syscallPrint+0xb3>
100943: 26 8a 16                mov    %es:(%esi),%dl
100946: 8b 19                   mov    (%ecx),%ebx
100948: 80 fa 0a                cmp    $0xa,%dl
10094b: 75 cb                   jne    100918 <syscallPrint+0x48>
10094d: 43                      inc    %ebx
10094e: 89 19                   mov    %ebx,(%ecx)
100950: c7 07 00 00 00 00       movl   $0x0,(%edi)
100956: 83 fb 19                cmp    $0x19,%ebx
100959: 75 e0                   jne    10093b <syscallPrint+0x6b>
10095b: c7 01 18 00 00 00       movl   $0x18,(%ecx)
100961: 89 4d cc                mov    %ecx,-0x34(%ebp)
100964: 8b 5d d0                mov    -0x30(%ebp),%ebx
```

```
100967: e8 cc fc ff ff          call   100638 <scrollScreen>
10096c: 8b 4d cc                mov    -0x34(%ebp),%ecx
10096f: eb ca                   jmp    10093b <syscallPrint+0x6b>
```

4 问题与思考

1. 一开始直接使用框架代码给出的 makefile 进行编译并执行，但发现报错"boot block too large"，于是根据实验 0 的相关指导将命令改为 objcopy -S -j .text -O binary bootloader.elf bootloader.bin，即可正确完成编译。

2. 在命令行中运行 qemu-system-i386 os.img，窗口显示"no bootable device"。通过询问助教，得知原因可能是 gcc 版本问题。安装 gcc6 并编译可成功运行。另外，还有一个解决方案是加一个编译参数-fno-asynchronous-unwind-tables，经过尝试也可以成功。查询相关资料知，"This option determines whether unwind information is precise at an instruction boundary or at a call boundary. If -fno-asynchronous-unwind-tables is specified, the unwind table is precise at call boundaries only." 新版本的 gcc 默认行为是"-fasynchronous-unwind-tables"，而旧版的 gcc 默认"-fno-asynchronous-unwind-tables"。猜想 unwind information 的位置不同将影响加载过程，而 qemu 模拟的是老式的硬件，可能会产生不匹配。

3. 在 fork 函数中拷贝 pcb 时，如果采用直接按位赋值的形式，即 pcb[i] = pcb[current]，则无法正常完成功能。这是由于 pcb 中有一个内核栈的数组，在按位赋值时只会复制指针（即浅拷贝），会使复制前后的指针指向同一个内存区域，造成错误。在使用 C 语言时，必须时刻注意类似的指针相关的问题，否则会出现严重的问题。

5 建议

1. 建议将官方的实验环境升级到 20.04LTS 版本，这样不容易产生 gcc 版本问题，可以节省很多时间;
2. 既然一定会产生 boot block too large 的问题，建议直接将 Makefile 改成 objcopy -S -j .text -O binary bootloader.elf bootloader.bin。