OS Lab 3 实验报告

赵思衡　201300024　zhaosh@smail.nju.edu.cn

一、实验环境

ubuntu 版本：20.04

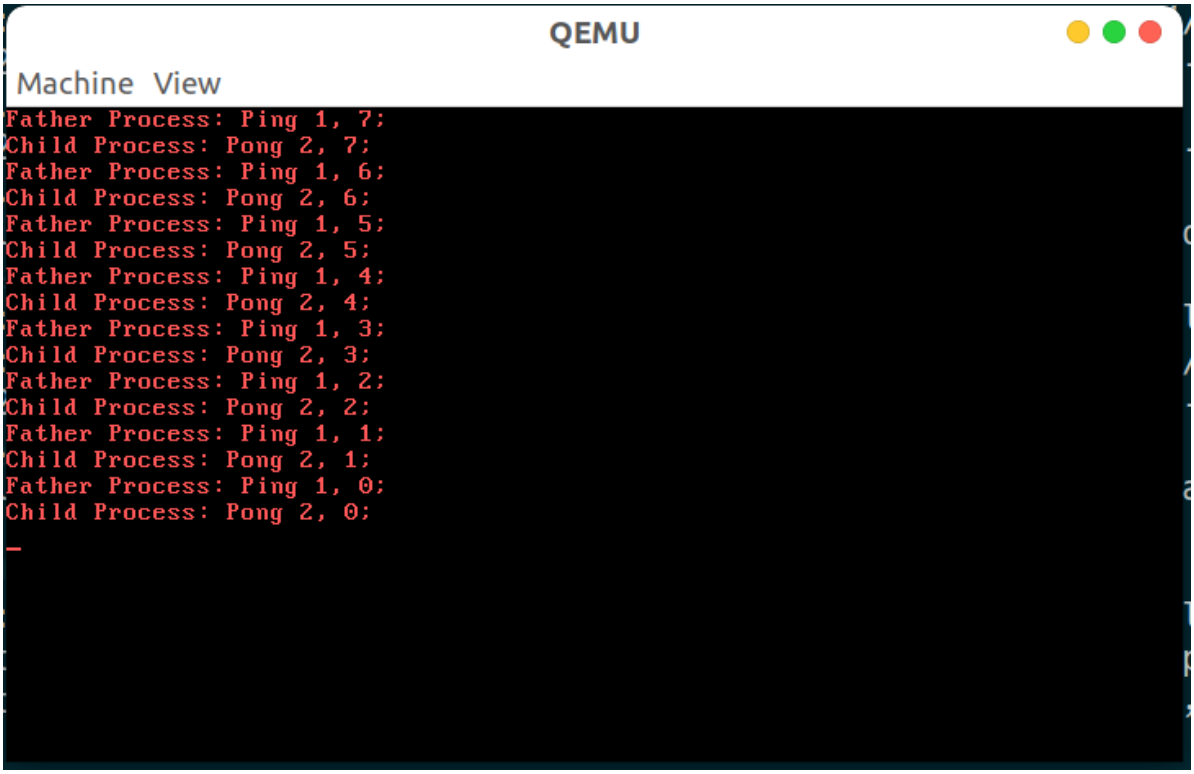
gcc 版本：9.4.0

二、实验进度

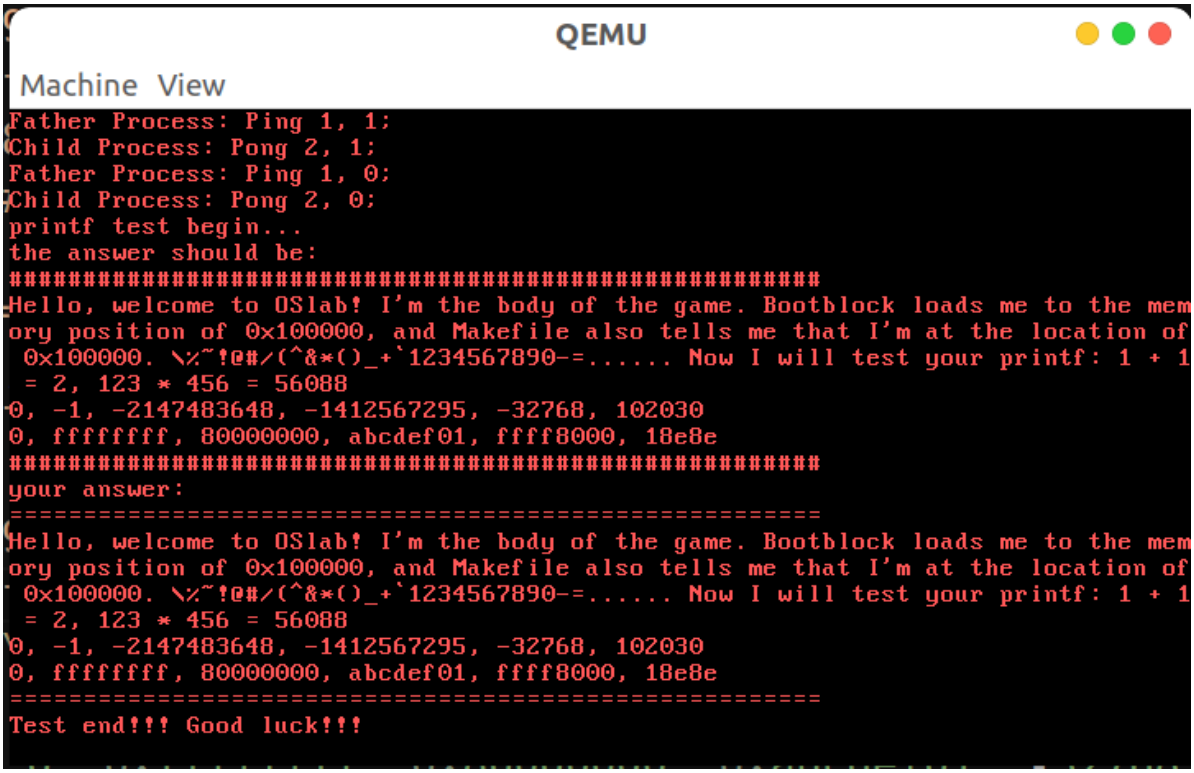
- 我完成了所有必做内容。
- 我完成了 exec 的选做内容
- 在二选一的内容中，我挑选了 **嵌套中断** 并将其完成。

三、实验结果

- 必做题结果：



- 选做题结果：



四、代码介绍

- /lab3/app/Makefile 和 /lab3/app_print/Makefile

修改了 Makefile中的 LDFLAGS 来缩减 app size

2 lab3-201300024/lab3/app/Makefile

```
@@ -4,7 +4,7 @@ LD = ld
CFLAGS = -m32 -march=i386 -static \
         -fno-builtin -fno-stack-protector -fno-omit-frame-pointer \
         -Wall -Werror -O2 -I../lib
- LDFLAGS = -m elf_i386
+ LDFLAGS = -m elf_i386 -z noseparate-code

UCFILES = $(shell find ./ -name "*.c")
LCFILES = $(shell find ../lib -name "*.c")
```

2 lab3-201300024/lab3/app_print/Makefile

```
@@ -4,7 +4,7 @@ LD = ld
CFLAGS = -m32 -march=i386 -static \
         -fno-builtin -fno-stack-protector -fno-omit-frame-pointer \
         -Wall -Werror -O2 -I../lib
- LDFLAGS = -m elf_i386
+ LDFLAGS = -m elf_i386 -z noseparate-code

UCFILES = $(shell find ./ -name "*.c")
LCFILES = $(shell find ../lib -name "*.c")
```

- /lab3/bootloader/boot.c

修正了框架代码的bug

```
@@ -37,7 +37,7 @@ void bootMain(void) {
        phnum = ((struct ELFHeader *)elf)->phnum;

        for(i = 0; i < phnum; i++){
-               thisph = ((struct ProgramHeader *)(elf + phoff));
+               thisph = ((struct ProgramHeader *)(elf + phoff) + i); // this place miss 1
                if(thisph->type == PH_LOAD){
                        load_memcpy((void *)thisph->vaddr, ((void *)elf+thisph->off), thisph->filesz);
                        load_memset((void *)thisph->vaddr+thisph->filesz, 0, thisph->memsz-thisph->filesz);
```

- /lab3/kernel/kernel/irqHandle.c
  - 在 timerHandle 函数中完成了时间中断以及进程切换任务
  - 在 syscallFork/syscallExec/syscallSleep/syscallExit 函数中完成了四个系统调相应的中断处理函数

- /lab3/lib/syscall.c

使用 syscall 完成了 fork/exec/sleep/exit 四个系统调用

五、遇到的问题与想法

1. Boot from hard disk

在最开始运行的时候，我遇到了程序卡在 boot from hard disk 进不去的问题。当然，经过了lab2的训练，我很快就意识到这是由于 bootMain 未能成功加载 kernel 内核。那我们就到 bootMain 中找问题。

由于我在 lab2 的实验报告中已经提出过，lab2 框架代码中的此部分的代码是有问题的，于是我就下意识地先将其替换为我在 lab2 中写的更框架无关的代码。最终发现是可以正确地从 bootloader 进入 kernel 的。

这也就意味着这一块代码确实是会影响运行的。

于是我就开始读该段代码，最后总发现了框架代码中少写了一个 `i` 的问题。

```
@@ -37,7 +37,7 @@ void bootMain(void) {
37  37          phnum = ((struct ELFHeader *)elf)->phnum;
38  38
39  39          for(i = 0; i < phnum; i++){
40      -               thisph = ((struct ProgramHeader *)(elf + phoff));
    40  +               thisph = ((struct ProgramHeader *)(elf + phoff) + i); // this place miss i
41  41                  if(thisph->type == PH_LOAD){
42  42                          load_memcpy((void *)thisph->vaddr, ((void *)elf+thisph->off), thisph->filesz);
43  43                          load_memset((void *)thisph->vaddr+thisph->filesz, 0, thisph->memsz-thisph->filesz);
```

那么为什么这段有问题的代码在ubuntu18.04上可以跑呢？这是由于默认工具链中链接器ld不同的缘故，导致其elf文件布局格式不同（具体可见app too large的问题）；而在ubuntu18.04中，ld的默认布局导致其elf文件中仅有一个可装载 LOAD 段，而在 ubuntu20.04 中有多个可加载段。

2. App too large

这是一个在ubunrtu 20.04 下会出现的问题。根据之前使用 objcopy 的经验，如果不想把每个 app 占用的合法地址由 20 section 扩大为 30 section 的话，那么只能从减小 elf 文件的大小着手。

我们先来看 uMain.elf 的程序头表：

```
Program Headers:
  Type           Offset   VirtAddr   PhysAddr   FileSiz MemSiz  Flg Align
  LOAD           0x001000 0x00000000 0x00000000 0x0080c 0x0080c R E 0x1000
  LOAD           0x002000 0x00001000 0x00001000 0x00204 0x00204 R   0x1000
  LOAD           0x003000 0x00003000 0x00003000 0x0000c 0x0000c RW  0x1000
  GNU_STACK      0x000000 0x00000000 0x00000000 0x00000 0x00000 RW  0x10
```

很明显地看出来，由于 elf 文件的布局过于松散，中间存在大量空隙，导致 elf 文件过大。

于是我们可以给我们的连接器ld设置控制参数 `-z noseparate-code`，其官方文档是这样说的：

> ## `-z separate-code`
>
> Introduced in binutils 2.31, it is default on Linux/x86. GNU ld has such a layout:
>
> - R `PT_LOAD`
> - RX `PT_LOAD`
> - R `PT_LOAD`
> - RW `PT_LOAD`
>
> The idea is that a byte (RX `PT_LOAD`) in the file that is mapped to the executable section will not be mapped to an R `PT_LOAD` at the same time. Note that the R after RX is not great. The better layout is to merge this R with the first R, but it seems to be difficult to implement in GNU ld.

> ## `-z noseparate-code`
>
> This is GNU ld's classic layout allowing the executable section to overlap with other `PT_LOAD`.
>
> - RX `PT_LOAD`
> - RW `PT_LOAD`
>
> The first `PT_LOAD` is often called text segment, but the term is inaccurate because the segment has read-only data as well.
>
> This layout is used by default in ld.lld 10, and there is no need to align any `PT_LOAD`.

简单来说，使用 noseparate-code 选项可以将 rodata 段与 .text 段合并到一起。

可见，在ubuntu18.04 中，其使用的默认ld 版本是 ld.lld 10，默认使用 noseparate-code 选项；而在ubuntu20.04中使用的 separate-code 布局。

这样我们得到的新的程序头表就如下所示，仅包含两个可装载段，满足大小需求。

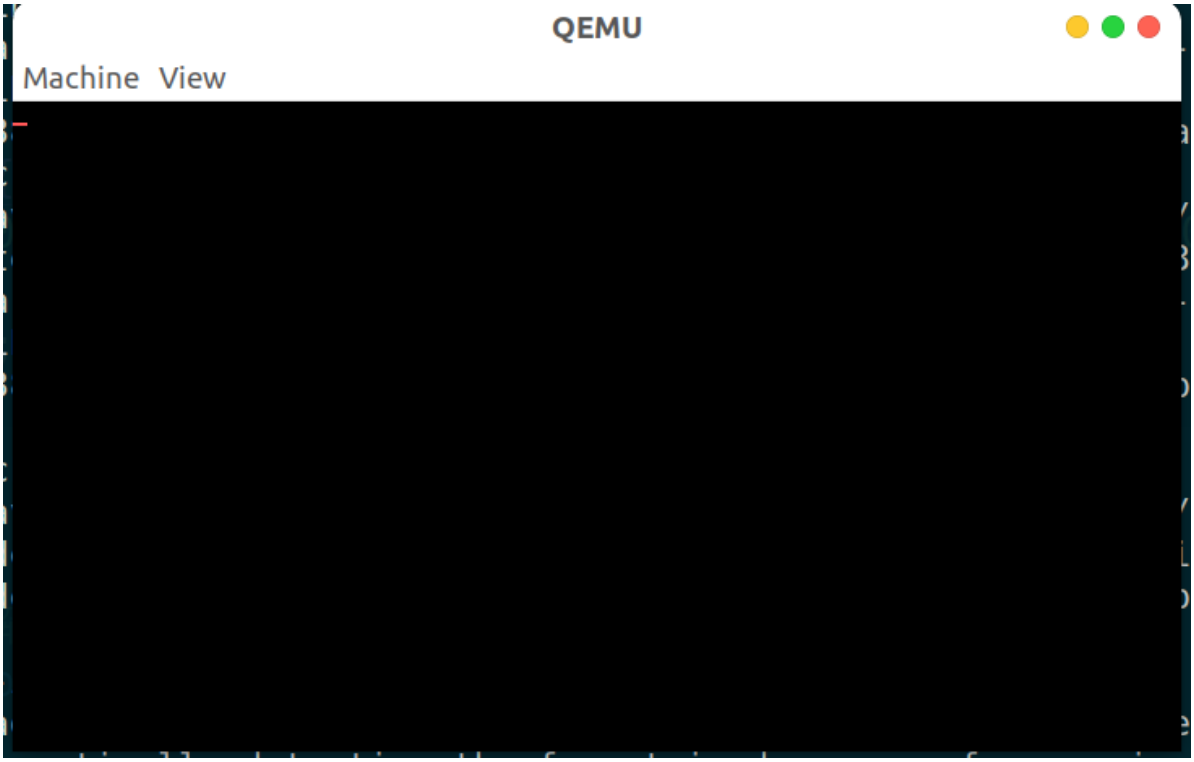
```
Program Headers:
  Type           Offset   VirtAddr   PhysAddr   FileSiz MemSiz  Flg Align
  LOAD           0x001000 0x00000000 0x00000000 0x00a10 0x00a10 R E 0x1000
  LOAD           0x002000 0x00002000 0x00002000 0x0000c 0x0000c RW  0x1000
  GNU_STACK      0x000000 0x00000000 0x00000000 0x00000 0x00000 RW  0x10
```

3. 嵌套中断问题

我选做的是嵌套中断这个问题，框架代码给出：

```
enableInterrupt();
for (j = 0; j < 0x100000; j++) {
    *(uint8_t *)(j + (i + 1) * 0x100000) = *(uint8_t
*)(j + (current +1) * 0x100000);
    asm volatile("int $0x20"); //XXX Testing
irqTimer during syscall
}
disableInterrupt();
```

在我将其复制上去并运行之后， QEMU 进入了 app 之后却不能运行，只是卡在了最开始的红色下划线处。



在我检查了我的代码却没发现问题之后，我决定将它仍在这并去吃饭。

等我吃完饭回来却发现 QEMU 成功运行完了。这时我大概猜到，这是由于嵌套中断的次数太多而导致运行时间过长的原因。于是我挑选了一个适中的频率：

```
enableInterrupt();
for(int j = 0; j < 0x100000; j++){
    *(unsigned char *)((i+1)*0x100000 + j) = *
(unsigned char *)((current+1)*0x100000 + j);
    if(j % 0x100 == 0){
        asm volatile("int $0x20");
    }
}
disableInterrupt();
```

这样，进入 QEMU 有一个明显的卡顿可以说明我们确实进入了嵌套中断，而又不至于等待太长时间。

六、思考题解答

EX1:

> linux下进程的创建及运行有两个命令fork和exec， 请说明他们的区别？ps鼓励自己实现相关代
> 码，并附上关键部分代码的区别

- 影响进程上的不同：
  - fork是在一个父进程中新建一个子进程，会增加进程的数目，改变进程的pid
  - exec是在一个已经存在的进程中加载可执行文件，其不会创建新的进程，也不会改变进程的pid
- 影响程序运行上的不同：
  - fork在创建一个新的子进程之后，除了改变进程的pid之外，会将父进程的地址空间、用户态堆栈以及进程控制快pcb完全拷贝至子进程的内存中，而不做任何改变，其实质就是复制一份当前进程并与其独立执行。

- exec在一个已经存在的进程中对程序进行重新装在，会覆盖原进程，将进程的代码、数据、堆栈用新程序替换。

fork:

```
// 拷贝地址空间
for (j = 0; j < 0x100000; j++) {
    *(uint8_t *)(j + (i + 1) * 0x100000) = *(uint8_t
*)(j + (current +1) * 0x100000);
}

// 拷贝pcb
memcpy(&pcb[i],&pcb[current],sizeof(ProcessTable));
```

exec:

```
loadelf(secstart, secnum, (current + 1) * 0x100000,
&entry);
pcb[current].regs.eip = entry;
```

由上述实现的关键代码可以看出，fork做的是将current进程的地址空间、pcb、内核堆栈拷贝到新的进程中，而exec做的是将新的指定程序加载到当前进程的地址空间当中并跳转执行。

EX2:

> 请在实验报告中简要说明你对 fork/exec/wait/exit函数的分析，并分析fork/exec/wait/exit在
> 实现中是如何影响进程的执行状态的?

fork 函数会在当前父进程中创建一个子进程，并将父进程的地址空间、内核堆栈、进程控制快等内容全部拷贝到新创建的子进程中去。在fork的实现中，它并不会直接影响一个进程的执行，只是在pcb数组中寻找了一个 `STATE_DEAD` 状态的进程，并将其重新设置为 `RUNNABLE` 的状态，以供后续切换。

exec 函数会通过 `loadelf` 函数加载一个新的可执行文件，并通过设置当前 pcb 的 eip 寄存器来实现执行流的跳转。

wait 函数通过将当前进程的执行状态设置为 `STATE_BLOCKED` 并设置睡眠时间来阻塞当前进程交出cpu，然后通过 `int $0x20` 时间中断来实现进程的切换。

exit 函数通过将当前进程的执行状态设置为 `STATE_DEAD` 来结束当前进程，然后通过 `int $0x20` 时间中断来实现进程的切换。

EX3:

> 请在实验报告中描述当创建一个用户态进程并加载了应用程序后，CPU是如何让这个应用程序最
> 终在用户态执行起来的。即这个用户态进程被OS选择占用CPU执行（RUNNING态）到具体执行应用程序第
> 一条指令的整个经过。

在本次实验中，当我们使用 fork 函数创建一个用户态进程并使用 exec 函数加载一个应用程序之后，cpu 会在pcb中设置当前eip的值为加载的用户程序的entry地址，接着陷入时间中断。在时间中断处理函数中，我们会将之前在调用exec函数时压入栈中的一系列寄存器以及栈的位置的值返回，然后再使用 iret 指令从内核中返回用户态。这之后，在用户态中我们就会根据之前的系统寄存器以及设置好的eip的值运行新的应用程序。

EX4：

> 请给出一个用户态进程的执行状态生命周期图（包执行状态，执行状态之间的变换关系，以及产
> 生变换的事件或函数调用）。