

《操作系统》lab 3实验报告

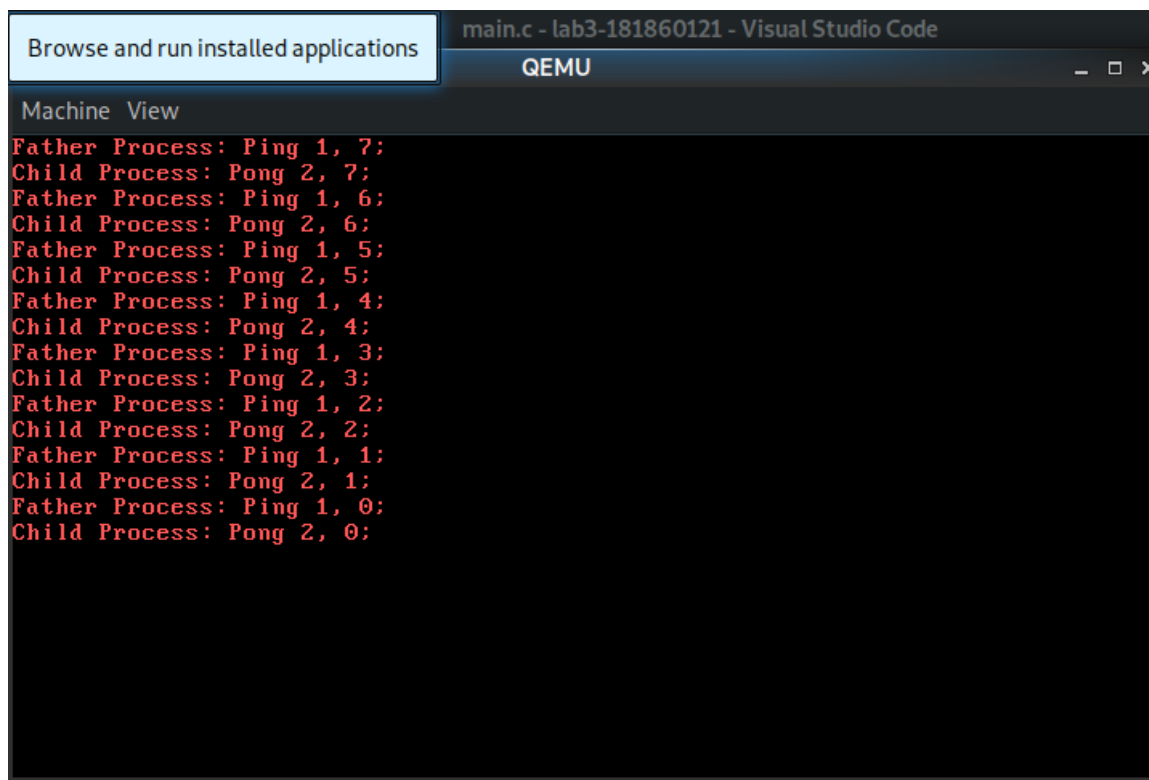
姓名	学号	邮箱	日期
薛飞阳	181860121	502126785@qq.com	2020/4/23

1. 实验进度

我完成了全部必做内容，以及选做中的中断嵌套

2. 实验结果

//截图为开启了中断嵌套的运行结果



```
main.c - lab3-181860121 - Visual Studio Code
QEMU
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;
```

```
main.c - lab3-181860121 - Visual Studio Code
QEMU
Machine View
Father Process: Ping 1, 1:
Child Process: Pong 2, 1:
Father Process: Ping 1, 0:
Child Process: Pong 2, 0:
printf test begin...
the answer should be:
#####
Hello, welcome to OSlab! I'm the body of the game. Bootblock loads me to the mem
ory position of 0x100000, and Makefile also tells me that I'm at the location of
0x100000. \x~!@#/(^&*(*)_+'1234567890-=..... Now I will test your printf: 1 + 1
= 2, 123 * 456 = 56088
0, -1, -2147483648, -1412567295, -32768, 102030
0, ffffffff, 80000000, abcdef01, ffff8000, 18e8e
#####
your answer:
=====
Hello, welcome to OSlab! I'm the body of the game. Bootblock loads me to the mem
ory position of 0x100000, and Makefile also tells me that I'm at the location of
0x100000. \x~!@#/(^&*(*)_+'1234567890-=..... Now I will test your printf: 1 + 1
= 2, 123 * 456 = 56088
0, -1, -2147483648, -1412567295, -32768, 102030
0, ffffffff, 80000000, abcdef01, ffff8000, 18e8e
=====
Test end!!! Good luck!!!
```

在lab3目录下运行 `make; make play` 后，控制台上的输出与lab2基本一致，为系统创建os.img的部分，这里不再截图展示，在QEMU的显示界面可以看到，与 `lab3/app/main.c` 中的 `uEntry(void)` 函数所期望实现的一样，父子进程分别从 `i = 7` 开始输出不同的字符串，然后子进程 `exit`，父进程通过 `exec()` 函数加载 `app_print` 并执行，输出与lab2一致的字符串。

3. 实验修改的代码

3.1. 完成库函数

修改位置： `lab3/lib/syscall.c`

在 `syscall.c` 中，为四个库函数补充内容，基本格式为添加上 `return syscall(SYS_XXX, 其他参数...)` 其中，`exec()` 调用 `syscall` 时，在参数1传入 `filename` 字符串的地址，参数2传入 `filename` 字符串的长度，以便之后使用；`sleep()` 函数在参数1传入参数 `time`；`fork()` 和 `exit()` 函数并未传入指定参数，即参数均设为缺省值0。

3.2. 时钟中断处理

修改位置： `lab3/kernel/kernel/irqHandle.c`

在 `irqHandle.c` 中的 `timeHandle()` 函数中需要实现时钟中断的处理，包括以下两个内容：

1. 处理pcb中状态为 `STATE_BLOCK` 的进程
2. 处理当前进程

第一个内容的实现较为简单，使用for循环从0开始遍历所有进程即可，代码部分如下所示：

```

for (int i = 0; i < MAX_TIME_COUNT; i++)
{
    if (pcb[i].state == STATE_BLOCKED)
    {
        pcb[i].sleepTime--;
        if (pcb[i].sleepTime == 0) //wake
        {
            pcb[i].state = STATE_RUNNABLE;
        }
    }
}
}

```

第二部分的逻辑如下，首先需要判断当前进程是否满足条件 `pcb[current].state == STATE_RUNNING && pcb[current].timeCount < MAX_TIME_COUNT`，如果满足这个条件，就不需要切换进程，只需要将 `timeCount++` 即可，然后注意需要判断 `timeCount++` 后是否等于 `MAX_TIME_COUNT`，如果等于，需要将其状态设为 `STATE_RUNNABLE` 并切换进程；如果不满足，就意味着当前进程为 `DEAD` 或 `BLOCKED`，同样需要切换，因此考虑把切换部分代码独立成函数。该函数的切换功能在手册中已经给出实现，这里仅介绍函数找到切换下标过程的实现。由于使用轮询调度，找下标的循环应该使用从 `current + 1` 开始到 `current - 1` 结束，即每次做 `i = (i + 1) % MAX_PCB_NUM`，同时，按照要求，`IDLE` 进程在最后考虑，所以，`ChangeProcess()` 找下标的过程如下所示：

```

int i = (current + 1) % MAX_PCB_NUM;
for (; i != current; i = (i + 1) % MAX_PCB_NUM)
{
    if (i == 0) //IDLE
        continue;
    if (pcb[i].state == STATE_RUNNABLE)
    {
        break;
    }
}
if (i == current) //No Runnable Program, goto IDLE
    i = 0;
current = i;

```

之后就可以进行进程的切换了。进程切换通过切换 `tss` 寄存器的 `esp0` 切换内核堆栈，然后再通过一系列指令切换上下文后，通过 `iret` 指令退出内核态。

`timeHandle` 的第二部分的流程如下：

```

if (pcb[current].state == STATE_RUNNING && pcb[current].timeCount <
MAX_TIME_COUNT)
{
    pcb[current].timeCount++;
    if (pcb[current].timeCount == MAX_TIME_COUNT)
    {
        /* time all used, maybe need to change state? */
        pcb[current].state = STATE_RUNNABLE;
        ChangeProcess();
    }
}
else
{
    ChangeProcess();
}

```

3.3. 系统调用例程

3.3.1. syscallFork

修改位置： lab3/kernel/kernel/irqHandle.c

分析 `syscallFork()` 函数，该函数需要完成的事情有

1. 找到一个DEAD进程
2. 如果没有，则fork失败，否则，就需要将父进程的一些资源复制给子进程

完全复制的内容有：

- 内存中的部分，即从 `(i + 1) * 0x100000` 开始的0x100000个字节，包括代码段、数据段、堆栈段等；
- 除了cs, ss, ds段寄存器和eax寄存器以外的寄存器值；
- 其他一些变量，如timeCount、sleepTime、state

与父进程不同的有：

- stackTop和prevStackTop，其中stackTop需要赋值为当前进程pcb的regs的首地址，这样才能在执行完之后重新回到用户态；
- cs、ds、ss寄存器，需要设置指向当前进程的段地址；
- eax值，对于子进程，需要设置为0；对于父进程，需要设置为子进程PID；
- pid值，为当前进程自己的PID

按照这个思路，我们很容易就可以写出syscallFork部分的代码了。由于这部分代码基本就是按照上述思路进行变量的赋值和内存的拷贝，在理清需要复制哪些东西之后实现起来几乎没有难度，在这里就不再赘述。

3.3.2. syscallSleep和syscallExit

修改位置： lab3/kernel/kernel/irqHandle.c

只需要修改当前进程状态为STATE_BLOCKED/STATE_DEAD后通过内联汇编执行指令 `int $0x20` 即可。

3.3.3. syscallExec

本函数需要完成3项任务：

1. 读取文件名
2. 通过文件名，使用 `loadelf()` 函数加载文件内容到内存
3. 若成功，设置eip，否则设置eax为-1

下面介绍前两项任务的实现。

Step 1 读取文件名：

修改位置： lab3/kernel/kernel/irqHandle.c

由于之前的 `exec()` 函数将文件名首地址作为第2个参数，文件长度作为第3个参数，所以直接使用 `tf->acx`

和 `tf->edx` 就可以得到这两个变量，然而，对于文件名首地址而言，我们其存储的文件名字符串位于数据段，我们不能直接通过引用得到文件名，而必须通过 `ds` 段寄存器来间接逐字符读取字符串。参考 `syscallPrint` 中的相关实现，读取文件名的操作如下：

```
/*get filename in tmp */
int sel = tf->ds;
char *str = (char *)tf->ecx;
int size = tf->edx;
int i = 0;
char tmp[256]; //filename no more than 256 B
char character = 0;
asm volatile("movw %0, %%es" :: "m"(sel));
for (i = 0; i < size; i++)
{
    asm volatile("movb %%es:(%1), %0"
                  : "=r"(character)
                  : "r"(str + i));
    tmp[i] = character;
}
tmp[i] = '\0';
```

这样，我们就将文件名读到了 `tmp` 字符串内，然后通过 `int ret = loadElf(tmp, (current + 1) * 0x100000, &entry);` 就可以得到返回值和入口地址，接下来需要做的就是实现 `loadElf()` 函数。

Step 2 `loadElf()` 的实现

修改位置：`lab3/kernel/kernel/kvm.c`

`loadElf()` 函数需要将 `elf` 文件读取，并将其 `LOAD` 段加载到内存中，然而，由于没有读文件的相关库函数，就需要借用 `inode` 把整个 `elf` 读到某个空间内，这里采用的做法在该函数内开一个大小为 `0x100000` 的字符串数组用于临时读取 `elf` 文件，这个做法可能与实际操作系统的使用方法不同。另一个可行的做法是找到一个 `DEAD` 进程，并利用该进程占用的内存空间临时存储 `elf` 文件，然而如果没有 `DEAD` 进程就会出错。

借鉴 `loadUmain()` 函数，我们可以使用 `inode` 将文件读取到临时数组中，并得到 `elf` 头。通过这个 `elf` 头，我们可以获得两个重要的参数——程序的入口地址 `entry` 和程序头偏移量 `phoff`，如下所示：

```
*entry = ((struct ELFHeader *)elf)->entry;
int phoff = ((struct ELFHeader *)elf)->phoff;
```

需要注意的是，如果在读取 `elf` 的任何过程中出现了错误就需要返回 -1。

然后，利用 `phoff`，我们就可以通过强制类型转换得到第一个程序头 `ph`，再从 `elf` 头中读取程序头数量 `phnum`，构造出最后一个程序头的下一个位置 `eph`。接下来就可以以 `ph` 为起点、`eph` 为终点遍历访问所有程序头了，如下所示：

```
struct ProgramHeader *ph = (struct ProgramHeader *) (elf + phoff);
struct ProgramHeader *eph = ph + ((struct ELFHeader *)elf)->phnum;
```

访问时判断类型是否为 `LOAD (0X1)`，如果是，则需要做：

- 将 `filesize` 个字节读到相应的物理地址中
- 将从 `filesize` 到 `memsize` 的空间清零

由于 `Program Header` 含有 `vaddr`，所以只需要将这个值加上 `physAddr`，就得到了相应的物理位置的起始点，这两个任务的实现如下：

```

for (i = 0; i < ph->filesz; i++)
{
    *(uint8_t *) (physAddr + ph->vaddr + i) = *(uint8_t *) (elf + i + ph->off);
}
for (; i < ph->memsz; i++)
{
    *(uint8_t *) (physAddr + ph->vaddr + i) = 0;
}

```

到这里loadelf就已经成功了，返回0即可。

第三项任务较为简单，如果成功返回，eip设为在loadelf里得到的entry即可。

至此我们已经实现了全部的必做任务。

3.4. 选做：中断嵌套

关于中断嵌套的处理涉及到以下3个位置：

lab3/kernel/kernel/irqHandle.c irqHandle()

在irqHandle()里，在处理时钟中断之前，将prevStackTop存为当前的Stacktop，用于保存待恢复的栈顶信息；

lab3/kernel/kernel/irqHandle.c timerHandle()->ChangeProcess()

在切换进程后，将tss寄存器的esp0和Stacktop设为prevStackTop，将内核堆栈栈顶切换到原先的Stacktop；

lab3/kernel/kernel/irqHandle.c syscallFork()

在fork时检测中断嵌套，如下所示：

```

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

```

这样，在执行syscallFork()时遇到指令int \$0x20，就会先进入irqHandle()，然后保存下当前（syscallFork()）的堆栈信息，这样在执行到timerHandle()之后，即使发生了进程的切换，原先待恢复的栈顶也可以被恢复，就保证了中断嵌套的顺利执行。