

《操作系统》实验三报告

201220069 周心同

一、实验目的

在lab3中，我们会实现进程调度，实现多进程同时运行。大家会与课堂上学习的理论知识相结合，实现一个自己设计的调度程序。具体内容是实现fork, exit, sleep等库函数和它们对应的系统调用！

二、实验内容

- 本次实验共有如下任务点需要完成（编号都是从1开始计数）：
 - 9个exercise
 - 3个task
 - 5个challenge
- 完成内容：
 - 9个exercise（完成）
 - 3个task（完成）
 - 5个challenge（完成2, 4和5）

三、实验过程和结果（按照手册顺序）

exercise1: 请把上面的程序，用gcc编译，在你的Linux上面运行，看看真实结果是啥（有一些细节需要改，请根据实际情况进行更改）。

输出如下：

```
Father Process: Ping 1, 7;  
Child Process: Pong 2, 7;  
Child Process: Pong 2, 6;  
Father Process: Ping 1, 6;
```

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

猜想是正确的。

exercise2: 请简单说说，如果我们想做虚拟内存管理，可以如何进行设计（比如哪些数据结构，如何管理内存）？ 用分页机制

- 1.每个进程有自己独占的虚拟地址空间
- 2.虚拟地址空间和物理地址之间以“页”为单位对应
- 3.主存中放不下的“页”放到磁盘上去

每个进程维护一个页表，操作系统维护各个进程的虚拟页和物理页之间的映射关系。

challenge1: 请发挥你的编码能力，在实验框架代码上设计一款虚拟内存管理机制。并说明你的设计（可以和上面的exercise结合，此题是bonus，不写不扣分~）。

exercise3: 我们考虑这样一个问题：假如我们的系统中预留了100个进程，系统中运行了50个进程，其中某些结束了运行。这时我们又有一个进程想要开始运行（即需要分配给它一个空闲的PCB），那么如何能够以 $O(1)$ 的时间和 $O(n)$ 的空间快速地找到这样一个空闲PCB呢？

创建一个栈，每有一个PCB空闲就将PID入栈，这样找到一个空闲PCB只需将PID出栈即可，用了 $O(1)$ 的时间和 $O(n)$ 的空间。

exercise4: 请你说说, 为什么不同用户进程需要对应不同的内核堆栈?

内核堆栈存储进程的现场信息, 多个进程则需要存储多个现场信息, 可以使进程之间相互独立, 防止互相影响。

例如, 如果有进程以p1, p2, p3的顺序运行一定时间, p1运行完后将自己的现场信息保存进内核堆栈, 随后p2运行, p2运行完后也需要将自己的现场信息保存进内核堆栈, 在之后p1再次运行时, 如果和p2保存于同一块内核堆栈, 则p1可以访问到p2的现场信息, 会造成进程之间相互影响, 甚至可以恶意利用。

exercise5: stackTop有什么用? 为什么一些地方要取地址赋值给stackTop?

stacktop是指向进程现场信息的指针, 在进程切换的时候将栈顶指针切换到另一个进程的栈 (即stackTop) 上, 此时栈上保存的信息就是该进程的现场信息了, 所以只需pop (好像和陷入中断是反过来的) 就能切换进程的上下文, pop出来的值就是结构体参数stackframe的值。

exercise6: 请说说在中断嵌套发生时, 系统是如何运行的? (把关键的地方说一下即可, 简答)

中断嵌套发生时, 保存被打断的任务的上下文信息入栈 (同时将esp保存进stackTop), 再根据异常和中断号去查表得到处理程序的入口地址, 执行中断处理程序后返回到stackTop (保存的是上一个中断或用户进程的esp), 从栈里弹出先前保存的被暂停程序的现场信息再继续执行。

exercise7: 那么, 线程为什么要以函数为粒度来执行? (想一想, 更大的粒度是....., 更小的粒度是.....)

函数的调用和返回实际上就是栈帧的创建和释放, 所以, 函数所需的最小资源就是线程所需的最小资源。

更大的粒度是程序，对应了进程，更小的粒度是代码语句，无法承载一个线程。

challenge2: 请说说内核级线程库中的 pthread_create 是如何实现即可。

定义一个指针stackaddr指向新分配的栈的栈顶，创建一个pthread结构并初始化（根据传进来的参数），根据attr分配线程属性，根据要求创建一个合法大小的线程栈，设置执行线程函数的地址start_routine和参数地址args(赋值给pd)，设置一些相关的标志属性（如线程数+1）。

随后将函数参数和函数指针放入栈中，调用syscall进入内核。在内核中调用_do_fork,然后返回用户态。

返回后会先执行start_thread，以遍后续的处理，在这里准备好参数并调用用户的函数（先前存放在了栈中），在函数执行完后，释放线程相关数据并修改一些标志属性。

challenge3: 你是否能够在框架代码中实现一个简易内核级线程库，只需要实现create, destroy函数即可，并仍然通过时钟中断进行调度，并编写简易程序测试。不写不扣分，写出来本次实验直接满分。

challenge4: （不写不扣分，写了加分）

- 1.在create_thread函数里面，我们要用线性时间搜索空闲tcb，你是否有更好的办法让它更快进行线程创建？
- 2.我们的这个线程库，父子线程之间关联度太大，有没有可能进行修改，并实现一个调度器，进行线程间自由切换？（这时tcb的结构可能需要更改，一些函数功能也可以变化，你可以说说思路，也可以编码实现）
- 3.修改这个线程库或者自由设计一个线程库。

1.创建一个栈，每有一个TCB空闲就将TID入栈，这样找到一个空闲TCB只需将TID出栈即可，用了 $O(1)$ 的时间和 $O(n)$ 的空间。

2.经过观察可以发现该程序一个线程的运行过程是从launch(到子进程代码)到yield(到父进程代码)来回切换的过程，(而且第一次launch时父进程esp还是0，后面就有了一个值，推测是跟父进程给有关，esp的这个值是随机的，但是子进程和父进程eip的相对偏移量是定值，同时两个子进程的eip变得一样)。

tcb结构先修改，把父线程的结构删去，即改为

```
typedef struct tcb{
    __uint32_t ESP, EIP, ResValue;
    __uint8_t stack[maxStackSize];           //栈
    enum threadStatus status;                //线程状态
} tcb;
```

对于一个线程调度器，存放一个线程序列，原父线程也在内。

通过while循环不断检测正在运行的线程是否结束（设一个标志位），然后主动切换，在线程的运行中，每过一定时间片检查要不要切换（同样设一个标志位）。

当要切换线程时，先调用yield（正在运行的线程），只不过原先结构中的父线程内容改为序列中父线程的esp和eip，然后再调用launch（要切换的线程），同原先结构中的父线程改为序列中父线程的esp和eip。（这里有可能不需要，不确定，而且原本main中的调用顺序也需要修改，不怎么好实现）

3.自由切换如果意味着test函数中不能有yield和destroy，我发现在线程的运行中如何每过一定时间片检查标志位在用户级线程不好实现。同时为了最大程度复用助教代码（偷懒），就把主线程放到list[0]，每次切换线程都先切回0，这样就解决了关联度太大的问题。见代码report/thread1。

report/thread2中写了一个如果有两个子线程会相互切换，只有一个会切换回父线程查看的调度器，貌似有一点bug，同时因为一开始理解错了，只能一次运行两个子线程。

经过thread2的错误我发现如果要解决自由切换的问题，切回父线程也是不能少的，因为不切回父线程并不能start新的线程，所以依然要切回父线程再切到别的线程，所以thread1其实已经够了。

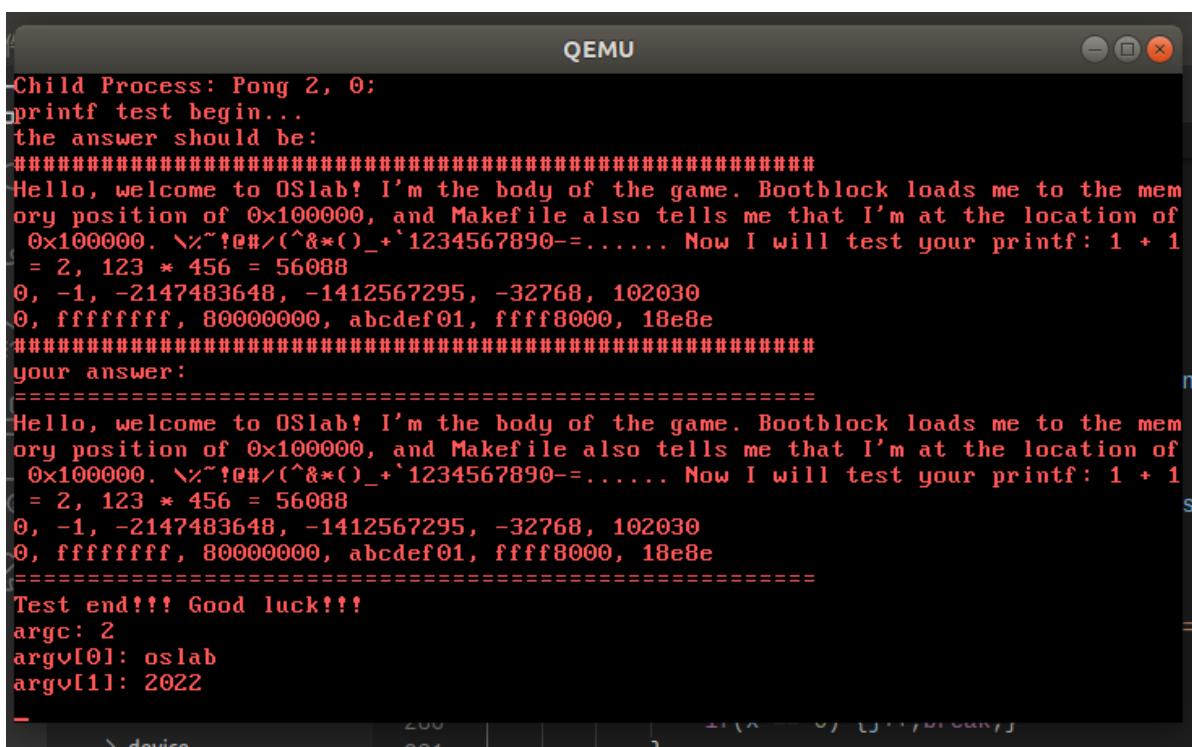
而且这样的话我们的框架代码貌似也不能支持多线程。。。。

challenge5: 你是否可以完善你的exec, 第三个参数接受变长参数, 用来模拟带有参数程序执行。

举个例子, 在shell里面输入cat a.txt b.txt, 实际上shell会fork出一个新的shell (假设称为shell0), 并执行exec("cat", "a.txt", "b.txt")运行cat程序, 覆盖shell0的进程。

不必完全参照Unix, 可以有自己的思考与设计。

```
}
int exec2(uint32_t sec_start, uint32_t sec_num, uint32_t argc, char * argv[]){
    /*TODO:call syscall*/
    return syscall(SYS_EXEC, sec_start, sec_num, argc, (uint32_t)argv, 0);
}
```



```
QEMU
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!!!
argc: 2
argv[0]: oslab
argv[1]: 2022
```

详细见代码

exercise8: 请用fork, sleep, exit自行编写一些并发程序, 运行一下, 贴上你的截图。(自行理解, 不用解释含义, 帮助理解这三个函数)

```
int main ()
{
    int pid = fork();
    if (pid == -1)
        printf("fork failed");
```


exercise9: 请问，我使用loadelf把程序装载到当前用户程序的地址空间，那不会把我loadelf函数的代码覆盖掉吗？（很傻的问题，但是容易产生疑惑）

实验2中，我们默认操作系统有一个内核进程，一个用户进程。

- 内核在链接时通过 `-Ttext 0x100000` 指定程序的起始地址，同时 `bootloader` 在加载时将内核进程加载到内存的 `0x100000` 处，设置GDT中内核代码段和数据段的基址为 `0x0`，值得一提的是内核 `ss` 也指向数据段，设置 `esp` 为 `0x200000`。
- 用户程序则不同，链接时指定程序起始地址为 `0x0`，却加载到内存的 `0x200000`，这时虚拟地址到物理地址就只能通过段选择子和 `GDT` 进行转换，所以设置代码段和数据段的段基址为 `0x200000`。这样在我们执行用户程序代码的时候，就是按照 `0x200000+eip` 作为指令地址来访问。

因为loadelf函数的代码在操作系统里面，应该在0x100000开头处，而用户程序的地址空间肯定在0x200000往后。

四、 实验的启示/意见和建议

虽然代码量比较小，但是问题挺多的，比如exec里面entry的作用，一开始直接无视了，还有timerHandle里面一开始在遍历的时候顺便标记，忘了考虑本身是STATE_RUNNABLE的情况，还有fork等函数返回值一开始全都写了return 0没改等等，bug比较多，手册里提醒的很详细就得意忘形了。