

2022spring 《操作系统》lab3实验报告

姓名：武雅琛

学号：201220102

23个exercise,3个challenge, 5个task,3个conclusion

一、PART1: exercise

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）。

- sleep(128)太太太长了，选用了sleep(1)。

```
wumaomao@wumaomao-VirtualBox:~$ ./oslab
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;
Child Process: Pong 2, 1;
Father Process: Ping 1, 1;
Father Process: Ping 1, 0;
Child Process: Pong 2, 0;
```

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）？。
- 需要划分虚拟地址空间，设计页表项（当然页表项的结构要符合硬件的规范，但是可以修改页表项的大小和分级页表是否需要等等），要在使用指令初始化cr3寄存器保存页表在内存空间的首地址。
- 另外内核还要具有用户程序装载的过程中填写页表的能力，按照虚拟地址分配物理地址，处理缺页导致的异常。
- 为了提高效率，还可以保存页表的缓存项，设计TLB，考虑替换原则。

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 $O(n)$ 的空间快速找到这样一个空闲PCB呢？
- $O(n)$ 比较容易想，遍历数组查询是否PCB空闲即可。

- $O(1)$ 我想的办法就是**维护一个空闲PCB的队列**，程序结束运行归还PCB空间，那么就加入队列，根据FIFO原则取用。恰好每次取出一个队列元素的事件为 $O(1)$ 。实现可以利用可变长度的链表。

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？
 - 不同用户进程切换的过程中需要把当前进程的现场信息（比如断点，程序状态字，通用寄存器等等）压入内核堆栈。
 - 共用堆栈不利于对堆栈内容的管理，**多个进程都可以对同一个内核堆栈写入信息**，那么具体到某一个进程，内核栈是**不可控的，不可靠的，不稳定的**。那么内核栈就有可能溢出，乃至崩溃。
 - 另外，在依赖外部设备执行的进程中，如果一个进程请求外部设备服务进入内核态，运行驱动程序保存信息后会阻塞该进程，因此cpu会调度另一个进程，而恰好**被调度的进程请求了同一个外部设备**。那么对于驱动程序来说**内核堆栈可能出现的情形就会很复杂**，很容易混淆驱动程序对应的数据内容。如果选择不同的内核堆栈，是有利于保持驱动程序代码的一致性的。

exercise5

- stackTop有什么用？为什么一些地方要取地址赋值给stackTop？
 - 查找到stackTop regs的结构体的定义，是 `stackFrame`，此处将它的首地址赋给了 `stackTop`。而 `stackFrame` 正是对应了某进程内核栈的内容，`stackFrame` 的地址对应了该进程**内核栈的栈顶地址**。
 - 所以 `stackTop` 的作用是**索引了将要切换的进程内核栈的栈顶位置**，在切换到另一个进程的时候，用 `stackFrame` 赋值是为了提供**当前进程的内核栈索引**。
 - 而将 `stackTop` 赋值给 `esp`，是为了**切换到被调度的进程的内核栈**。（所以在内核栈 `push` 类似于函数调用前保存一些寄存器内容和返回地址，赋值 `stackTop` 类似于 `push ebp`？去索引原来的栈帧，当然也有区别，之前在一个栈中划分各个函数分配的栈帧，现在多进程切换就要区分多个内核栈分别执行了）。

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）
 - 中断嵌套发生的时候，首先硬件会**查询idt**查找中断描述符。但是由于是在内核态引发中断，所以 `cpl == 0`，**不存在用户态到内核态的特权级转换**。直接把**程序断点和运行状态压入内核栈**，转入中断处理程序运行。
 - 转入中断处理程序后，依然是先将现场、参数、段选择子等内容**压入内核栈得到 `StackFrame`** 然后做进一步的具体操作。
 - 在返回的时候，如果发生过中断嵌套，会通过**递归的释放各次中断发生时压入内核栈的内容来逐级返回用户态**。

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.....）
 - 更大的粒度：**对象**。对象上封装的是一组数据和在一组数据上可以执行的若干操作，而操作正是通过函数接口的方式实现的。
 - 更小的粒度：**指令**。它们是组成函数的基本单位。
 - 对象需要**共享一段在虚拟存储空间中的内存空间**，但是线程没有**独立的**资源保护和分派的权限，所以对象对于线程来说，颗粒度偏大。
 - 指令之间运行**联系紧密**，在软件上处理会产生较高的运行代价，**不适宜在软件上处理**。一般来说是通过**cpu流水线并行**来实现细颗粒度的指令级并行的。

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）

- c语言代码：

```
1 //testsyscall.c
2 #include<unistd.h>
3 #include<stdio.h>
4 #include<stdlib.h>
5
6 int main()
7 {
8     int cnt = 0;
9     int ret = fork();
10
11     if (ret < 0)
12     {
13         printf("Error!\n");
14         exit(-1);
15     }
16     else if (ret == 0)
17     {
18
19         while(cnt < 6)
20         {
21             cnt += 1;
22             printf("Child Process: cnt = %d\n", cnt);
23             // sleep(3);
24         }
25         exit(0);
26     }
27     else
28     {
29         while(cnt < 6)
30         {
31             cnt += 1;
32             printf("Parent Process:cnt = %d\n", cnt);
33             sleep(1);
34         }
35         exit(0);
36     }
37
38 }
```

- 运行截图：

```
oslab@oslab-VirtualBox:~/testcode$ ./testfork
Parent Process:cnt = 1
Child Process: cnt = 1
Child Process: cnt = 2
Child Process: cnt = 3
Child Process: cnt = 4
Child Process: cnt = 5
Child Process: cnt = 6
Parent Process:cnt = 2
Parent Process:cnt = 3
Parent Process:cnt = 4
Parent Process:cnt = 5
Parent Process:cnt = 6
```

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）
 - 这个loadelf定义在irqHandle.c中，是内核中断处理程序的一个部分，位于从0x100000到0x200000的内核存储空间当中，而我们装载的程序至少在0x200000以上用户地址空间，关系不大。

二、PART2:challenge

challenge2

- 请说说内核级线程库中的pthread_create是如何实现即可。
 - 在Linux2.6之前内核不提供真正的线程服务，也就是说彼时线程是通过第三方用户库实现的用户级线程库来提供功能的，实质上对内核来说是透明的。
 - 从Linux2.6开始Native POSIX Thread Library (NPTL) 库被纳入Linux内核，新的线程在用户态中完成栈空间的分配和从父线程对栈空间的拷贝以及一些细节的处理，最后调用内核提供的系统调用clone来实现对于内核来说一对一的系统级线程（但对内核来说某种意义上也是一种进程，clone和fork的主要差异在于clone可以有选择的共享父进程的资源）。

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。

可以，我在oslab中实现了这项功能，并且在app_print.c起始的位置增加了相应的测试。

```
1 //app_print/main.c
2 int main(int argc, char* const argv[]) {
3     printf("argc: %d\n", argc); //argc = 2, argv[0] = "Hello", argv[1] =
   "world"
4     //printf("argc1:");
5     printf(argv[0]);
6     //printf("argc2:");
7     printf(argv[1]);
```

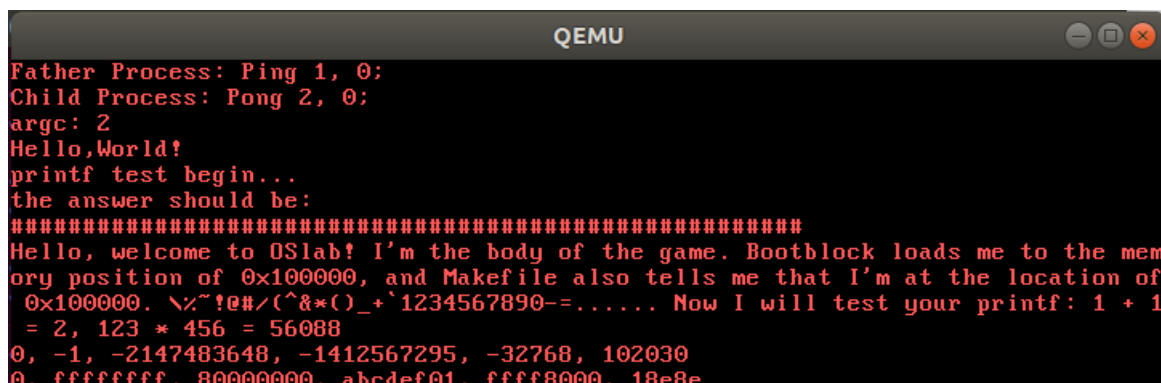
增加了对 argc 和 argv 的访问后发现在 main 函数中参数和函数的参数保存方式一致，按照从左到右的顺序从 esp + 4 往高地址存储。

简单来讲就是将字符指针参数在内核态解析并讲内容拷贝到从 0x100 开始的参数缓存区 (ARGVBUF) 中。

然后在 loadelf 之后找到重新装载的程序的最后一个段结束的位置，将参数拷贝在后面。

最后在 sf->esp 指向的用户栈中给出 argc 的值和指向 argv 字符指针数组的指针。

运行结果可见：

A screenshot of a QEMU terminal window. The title bar says 'QEMU'. The terminal output is as follows:

```
Father Process: Ping 1, 0;
Child Process: Pong 2, 0;
argc: 2
Hello,World!
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, 18c8e
```

三、PART3:task

task1

- 这一部分算是对实验2的复习，我们在 lab3/lib/syscall.c 中留下了三个库函数待完成，你需要调用 syscall 完善库函数。

☒ 完成fork

```
1 | int pid_t = syscall(SYS_FORK,0,0,0,0,0);
2 |     //printf("%d\n", pid_t);
3 |     return pid_t;
```

☒ 完成exec

返回值表明是否成功执行exec。

并且传入exec变长参数。

```
1 | return syscall(SYS_EXEC,sec_start, sec_num, (uint32_t)argv,0,0 );
```

☒ 完成sleep

同样如果传入了time < 0,返回-1。

```
1 | return syscall(SYS_SLEEP, time, 0,0,0,0);
```

☒ 完成exit

```
1 | syscall(SYS_EXIT, 0, 0,0,0,0);
```

task2

- 完成时钟中断处理函数。

- 时钟中断在本实验中承担了两项职务：**进程时间片计数和进程调度**。我利用**当前进程的状态**来区分对时钟中断的不同需求。

- 如果当前为运行态(RUNNING)，遍历PCB，对所有阻塞进程的sleeptime减一，如果减至0，从等待态释放出来转化为就绪态 (RUNNABLE)。
- 条件判断进入进程切换：

时间片用尽、进程结束、进程阻塞，或者位于idle进程检查是否存在就绪进程。

```
1  if (pcb[current].timeCount >= MAX_TIME_COUNT ||  
    pcb[current].state == STATE_DEAD || pcb[current].state ==  
    STATE_BLOCKED || current == 0)
```

- 条件判断通过后遍历PCB，查找就绪态进程，查找成功后切换进程即可。
- 如果不成功，如果当前进程位于**运行态**，也就是时间片用尽引发的进程调度，则 `pcb[current].timeCount = 0;`，**继续运行**。如果是被**阻塞**或者**进程结束**，切换到**idle进程**等待新的就绪进程到来。

```
1      if (idx == MAX_PCB_NUM)  
2      {  
3          if (pcb[current].state == STATE_RUNNING)  
4              pcb[current].timeCount = 0;  
5              //在当前进程新的时间片中继续执行  
6          else  
7          {  
8              putchar('i'); putchar('d');  
9              putchar('l');putchar('e');putchar('\n');  
10                 current = 0; //切换到idle进程  
11                 ....//进程切换  
12             }  
        }
```

- 如果未发生进程调度，即时间片尚未用尽的运行态进程，只需要timeCount加一即可。

task3

- 完成系统调用处理函数。

☒ syscallFork

☒ 拷贝地址空间

只需要注意在内核态**段基址为0x0**，所以要将**虚拟地址转换为物理地址**拷贝进程的地址空间。转换依照 `kvm.c` 设定的gdt即可。

```
1  //syscallFork()  
2  void* src_addr = (void*)((current+1)*0x100000);  
3  void* dst_addr = (void*)((i+1)*0x100000);  
4  memcpy(dst_addr, src_addr, 0x100000);
```

- ☒ 在父子进程返回正确的返回值。需要理解内核态改变内核栈对应eax的区域来实现返回值的**效果**。

```
1 | pcb[i].regs.eax = 0;  
2 | sf->eax = pid;
```

✓ syscallExec

从内核栈中取到需要的所需的参数传入 `loadelf` 函数即可。

- 将 `sf->esp` 清空放在栈顶。
- 将 `sf->eip` 设置到新的程序的 `entry` 的位置。
- 关于变长参数的处理见 `challenge 5` 的部分。

✓ syscallSleep

- 判断参数的合法性，如果 `sleepTime <= 0`，返回-1
- 将当前进程转换为阻塞态，并修改 `sleepTime`，借助时钟中断处理进程切换。

```
1 | if (sf->ecx <= 0)  
2 | {  
3 |     sf->eax = -1;  
4 |     return;  
5 | }  
6 | pcb[current].timeCount = MAX_TIME_COUNT;  
7 | pcb[current].sleepTime = sf->ecx;  
8 | pcb[current].state = STATE_BLOCKED;  
9 | asm volatile("int $0x20");
```

✓ syscallExit

和 `syscallSleep` 的实现相近。

四、PART4: feeling

- challenge, 不愧是你。QAQ。