

# 操作系统lab3【左右逢源-调度器】

---

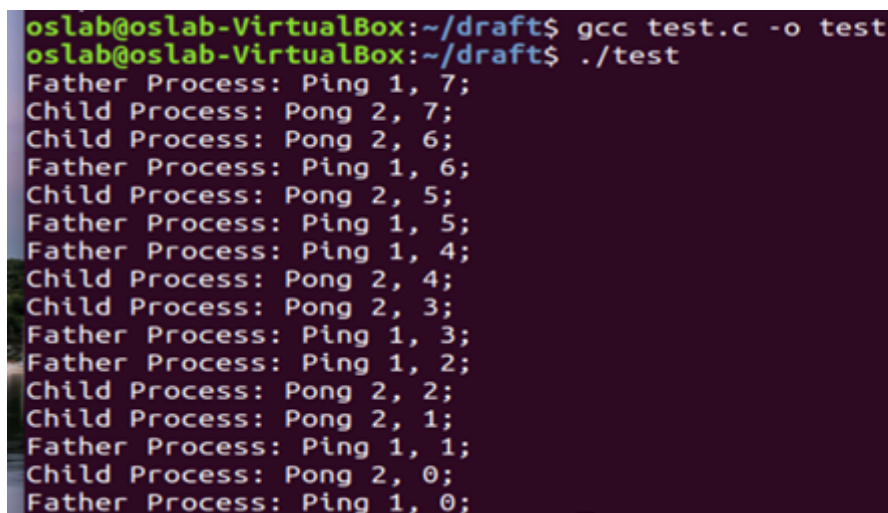
201220199 肖丹妮

- 9个exercise
- 3个task
- 1个challenge
  - challenge 2

## Exercises

---

### exercise1:



```
oslab@oslab-VirtualBox:~/draft$ gcc test.c -o test
oslab@oslab-VirtualBox:~/draft$ ./test
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;
Child Process: Pong 2, 1;
Father Process: Ping 1, 1;
Child Process: Pong 2, 0;
Father Process: Ping 1, 0;
```

### exercise2:

可以采用分页式虚存管理。

每个进程都有自己独立的虚拟地址空间。整个虚拟地址空间分为：内核空间和用户空间。内核空间包括内核代码和数据、物理存储区、以及和系统级上下文信息等；用户空间包括只读代码段、可读可写数据段、堆栈、共享库等用户级上下文信息。将这些区域划分成若干大小相等的虚拟页面，每个页面都属于某个区域。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，管理所有进程的页表。需要访问时，将虚拟地址转换为物理地址，若检测到页故障，可能是段故障、访问越权、访问越级等等，若是正常的缺页异常，就在主存中找到一个空闲的页框，从硬盘中将缺失的页面装入主存页框；否则将主存中的某个页框替换出去，从硬盘中将缺失的页面装入主存页框。并且更新页表项信息。

需要的数据结构：

1. 页表项PTE：存储每个页面的信息，是否在主存、在主存的物理基址，读写权限等
2. 页目录项PDE：存储每个页表的信息，下级页表是否在主存、在主存的物理基址、读写权限、是否可被用户进程访问、页大小等
3. 进程控制块pcb中增加进程页表的信息。

## exercise3:

使用空间大小为MAX\_PCB\_NUM的队列存储所有空闲PCB，初始时将所有PCB入队；后续，若某个进程的状态发生转换且变为STATE\_DEAD，就将它所占用的PCB回收、加在队尾；若需要给某进程分配空闲PCB，若队列不为空，在O(1)时间内直接取队首的PCB即可。

## exercise4:

1. 每个进程的内核堆栈都保存了自己的中断现场以及操作系统程序间相互调用的参数、返回值等。便于内核为不同的用户进程执行不同的处理。
2. 处于安全的考虑，如果用户栈和内核栈共享，用户就可以修改栈内容，突破内核的安全保护；如果用户进程崩溃，不会影响内核。

## exercise5:

(1) stackTop是进程的内核堆栈位置，也就是内核堆栈栈顶。进程切换时，令tss的esp0指针指向当前进程的stackTop，令esp寄存器的值=当前进程的stackTop，实现从上一个进程的内核堆栈切换到即将运行的进程的内核堆栈。

```
uint32_t tmpStackTop=pcb[current].stackTop;
tss.esp0=(uint32_t)&(pcb[current].stackTop);
asm volatile("movl %0,%%esp"::"m"(tmpStackTop));
```

(2) 取StackFrame结构体的首地址给stackTop，是为了保证stackTop一直指向进程内核堆栈中离栈顶最近的一个StackFrame结构体，当发生时钟中断时指向进程当前的上下文信息，进行进程切换，当进程切换回来时，就可以通过stackTop找到栈顶，恢复进程上下文。

## exercise6:

中断嵌套发生时，进程已经在内核态，无需进行特权级转换。

硬件：依次把eflags, cs, eip入栈，按照中断门描述符进入相应处理程序；

dolrq.S: 如果硬件没有push errorcode, push errorcode; 中断号入栈; pusha; ds, es, fs, gs入栈; esp入栈; 传递参数StackFrame \*sf=esp, 调用irqHandle

irqHandle: save esp: 更新stackTop, 令stackTop=sf=esp, 始终指向的是进程内核堆栈中离栈顶最近的一个StackFrame结构体，接着根据sf->irq执行中断处理程序

执行结束，通过pop、iret返回上一层中断或者用户态。

## exercise7:

若以比函数更小的粒度来执行线程，那么包含同一函数的语句的线程之间需要共享栈区、pc和寄存器。

若以比函数更大的粒度来执行线程，线程与进程相近，失去了线程本身的意义。

## exercise8:

```
#include <stdio.h>
#include <unistd.h>
#include <stdlib.h>

int main() {
    int index = 0, fpid_1 = -1, fpid_2 = -1;
    fpid_1 = fork();
```

```

if(fpid_1 != -1 && fpid_1 != 0)
    fpid_2 = fork();
int i = 3;
if (fpid_1 == 0) {
    index = 1;
    while( i != 0) {
        i --;
        printf("Child1 Process: Pong %d, %d;\n", index, i);
        sleep(5);
    }
    exit(0);
}
else if (fpid_2 == 0) {
    index = 2;
    while( i != 0) {
        i --;
        printf("Child2 Process: Pong %d, %d;\n", index, i);
        sleep(5);
    }
    exit(0);
}
else if (fpid_1 != -1) {
    index = 0;
    while( i != 0) {
        i --;
        printf("Father Process: Ping %d, %d;\n", index, i);
        sleep(5);
    }
    exit(0);
}
return 0;
}

```

```

oslab@oslab-VirtualBox:~/draft$ gcc test1.c -o test1
oslab@oslab-VirtualBox:~/draft$ ./test1
Child1 Process: Pong 1, 2;
Father Process: Ping 0, 2;
Child2 Process: Pong 2, 2;
Father Process: Ping 0, 1;
Child1 Process: Pong 1, 1;
Child2 Process: Pong 2, 1;
Father Process: Ping 0, 0;
Child1 Process: Pong 1, 0;
Child2 Process: Pong 2, 0;

```

## exercise9:

调用exec的是子进程，不是内核进程，loadelf函数在内核中，因此调用loadelf将程序装载到子进程的地址空间不会覆盖loadelf函数的代码。

## Tasks

---

## task1:

调用 syscall

- fork: 设置参数SYS\_FORK=sf->eax
- exec: 设置参数SYS\_EXEC=sf->eax、sec\_start=sf->ecx、sec\_num=sf->edx
- sleep: 设置参数SYS\_SLEEP=sf->eax、time=sf->ecx
- exit: 设置参数SYS\_EXIT=sf->eax

## task2:

1. 在kvm.c中建立就绪PCB队列，将pcb[1]加入队列。

```
int pcb_ablequeue[MAX_PCB_NUM];  
int front = 0;  
int rear = 0;
```

2. timerHandle()函数

- 首先遍历pcb，将状态为STATE\_BLOCKED的进程的sleepTime减一，如果进程的sleepTime变为0，重新设为STATE\_RUNNABLE，并且将其加在pcb\_ablequeue队尾
- 判断时间片是否用完或者阻塞，寻找状态为STATE\_RUNNABLE的进程next：  
如果是，若pcb\_ablequeue不为空，则在pcb\_ablequeue中取队首；若pcb\_ablequeue为空，则令next=0  
如果是时间片用完，还需要将进程状态设为STATE\_RUNNABLE，timeCount为0，将其加在pcb\_ablequeue队尾
- 令current=next，state设为STATE\_RUNNING，timeCount和sleepTime都为0
- 使用手册里的代码切换进程

## task3:

- syscallFork:  
查找空闲pcb，找到父进程返回i，子进程返回0，并且将子进程加在pcb\_ablequeue队尾；找不到返回-1  
使用memcpy拷贝地址空间；拷贝pcb;
- syscallExec:  
将参数secstart、secnum传给loadelf，使用loadelf把新的程序加载到current的地址空间，通过eax返回0
- syscallSleep:  
若传入参数sf->ecx > 0，说明合法，将sleepTime设为sf->ecx，state设为STATE\_BLOCKED，利用asm volatile("int \$0x20")模拟时钟中断
- syscallExit:  
state设为STATE\_DEAD，利用asm volatile("int \$0x20")模拟时钟中断

## Challenges

---

## challenge2:

传给pthread\_create的参数包括：newthread：即将创建的新线程的pthread结构指针；attr：用户指定的线程创建属性；start\_routine：新线程执行的函数指针；arg：新线程函数的参数地址。

一，STACK\_VARIABLES宏定义了新的堆栈指针stackaddr，stackaddr指向即将分配的线程栈的有效栈顶（不含保护区）。如果用户未指定线程创建的属性attr，则使用默认的属性值default\_attr。

ALLOCATE\_STACK宏用来分配线程栈，在栈底创建pthread结构并初始化。然后设置执行线程函数的地址start\_routine和参数地址arg。二，通过THREAD\_SELF获取当前进程的pthread结构。根据pthread结构的ATTR\_FLAG\_SCHED\_SET和ATTR\_FLAG\_POLICY\_SET值设置flag。然后设置pthread的各个成员变量，包括线程的调度策略和线程调度的优先级。三，将前面创建的pthread结构赋值给传入的参数newthread，调用create\_thread继续创建线程。