《操作系统》实验报告三

姓名：胡增杰

院系：计算机科学与技术系

学号：201220131

教师：叶保留

Exercise

exercise1

运行结果如下：

```
oslab@oslab-VirtualBox:~/oslab/exercise$ vim exercise1.c
oslab@oslab-VirtualBox:~/oslab/exercise$ gcc -o main exercise1.c
oslab@oslab-VirtualBox:~/oslab/exercise$ ./main
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;
oslab@oslab-VirtualBox:~/oslab/exercise$
```

exerise2

可以采用分页机制，将内存和磁盘的空间分成固定大小的块，每次将一个页的内容从磁盘加载到内存，在访问地址时如果对应的页没有被加载到内存，就使用中断机制产生页缺失，然后进行处理，将对应的页重新加载到内存，如果内存已满，就使用合理的替换算法进行替换，比如 LRU（least recently used）算法，中断处理之后就返回原来的指令继续执行。

使用的数据结构是页表，进行虚拟地址到物理地址的转化（简易版本），每个表项包含了的信息大致有：对应页的起始物理地址、该表项是否被加载到内存、使用次数（如果替换算法为 LRU 算法）等等。

exercise3

可以设置一个数组用来存储之前已经结束的进程的下标，然后要分配新的进程时直接取这个数组末尾的那个下标表示的 pcb，同时为了维护这个数组，分配之后将其删除，这样分配新的进程的代价为 O(1)，维护这个数组的代价也是 O(1),故总的代价也是 O(1).

exercise4

程序运行的时候有两种状态，内核态和用户态，执行时会在这两种状态之间进行切换，当从用户态进入内核态时，需要保存用户态的返回地址等有用信息，以便可以从内核态正确返回到用户态，而不同的进程执行的代码个功能时不同的，包含着不同的用户态信息，而且不同进程内核态执行的内容也各不相同，如果不为每个进程分配一个内核栈，就不能实现不同进程执行不同代码的功能。

exercise5

从 PCB 的定义可以看出，在每个 PCB 中为该进程分配了对应的内核栈，而 stackTop 就是在指示这个栈的栈顶位置。由于栈是从高地址向低地址增长的，所以 stackTop 赋值为&pcb[i].regs,这样就可以指到 regs 对应的起始处，这个地址也就是栈 stack 的最高处。

```
struct ProcessTable {
    uint32_t stack[MAX_STACK_SIZE];
    struct StackFrame regs;
    uint32_t stackTop;
    uint32_t prevStackTop;
    int state;
    int timeCount;
    int sleepTime;
    uint32_t pid;
    char name[32];
};
typedef struct ProcessTable ProcessTable;
```

exercise6

当发生中断嵌套时，在内核栈中保存原来中断处理例程的指令地址，寄存器值等信息，然后再将指令跳到新的中断的处理例程的位置机型执行，如果在新的处理过程中再发生中断，就重复上面的过程，当处理结束后，从栈中取出寄存器内容、返回地址，重新恢复原中断处理程序的现场进行执行。

exercise7

比函数更小的粒度是指令，指令的执行会涉及寄存器，但是单个的指令不足以执行一个设计好的功能，只有指令序列中具有一定的逻辑关系，才能配合执行某一个特定的功能，而函数就是具有逻辑关系的指令序列，因此执行粒度不能比函数小。

比函数更大的粒度是程序，程序是静态的代码和数据，而进程则是程序的一次动态执行，如果使用比函数更大的粒度，那就是进程本身了，

而我们引入线程的概念就是为了节省进程的无谓消耗，提高效率。所以执行粒度也不能比函数大。

exercise8

exercise9

不会把 loadelf 的代码覆盖掉，因为 syscall 的 loadelf 实在内核态执行的，实在内核空间，而装载到的是当前进程的用户空间，因此不会覆盖 loadelf 的代码。

Challenge

只写了

challenge2

pthread_create 的实现和 fork()有很多相似之处：

首先找到空闲的 TCB，为新的线程分配 TCB。

然后队新线程的 TCB 进行初始化，设置状态，初始化栈帧，设置 EIP。

这样就可以让新的线程获得自己的栈区、PC 和寄存器使用了。

challenge4

(1) 快速找到空闲 TCB 的方法：

可以记录一个全局队列 ready,用来记录所有可用的 TCB 的序号，这个队列只存储可用的 tcb 的序号，并不存储 tcb 本身。

这个队列 ready 通过数组来实现，大小为 SIZE = maxThread +1= 256+1，同时有首尾指针，首部指针 head 只进行出队列的指标，尾指针 tail 只进行入队列的指标。队列的维护操作与用数组实现队列的操作是一样的，不再赘述。

队列的初始化为 ready[i] = i,因为初始情况所有 tab 都可用，这是 head=0， tail=maxThread-1.

当分配 tcb 时，将队列头部的序号分配给新线程，并将 head = (head+1)%SIZE。

当有线程被销毁时，其 tcb 序号变成可用，将其加入队列尾部，并将 tail=(tail+1)%SIZE

当队列为空时，也就是 head = tail 时，没有 tcb 可用，如果此时想创建线程，产生错误。

当队列满时，也就是 head=(tail+1)%SIZE 时，（其实对于这个实现这种情况不会发生，因为队列的大小比 tcb 总数还要大 1）如果想在入队列，则出现错误。这种情况也不应该让其发生，因为如果一旦发生，那么这个没有被加入队列的序号在之后就永远没有机会被重新使用，这样就是内存泄漏，故应该让队列大小比 tcb 表数量要至少大 1（这里的至少大 1 为了维护队列判定满的操作)。

(2)和（3）没写