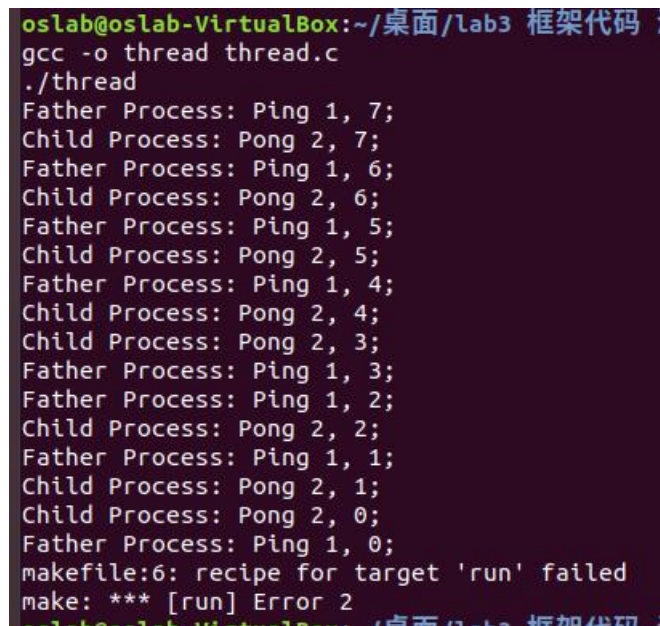


## OSLAB3

191830064 姜纪文

**exercisel:** 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

可以看到两个进程并发执行时，执行顺序是不确定。



```
oslab@oslab-VirtualBox:~/桌面/lab3 框架代码 :  
gcc -o thread thread.c  
./thread  
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;  
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;  
makefile:6: recipe for target 'run' failed  
make: *** [run] Error 2
```

**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）？

分页机制

**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？

我们用队列来管理各种状态的 PCB，例如空闲的 PCB 在空闲队列，就绪队列保存等待调度的 PCB，以及等待态的进程的 PCB 在等待队列中，正在运行的进程由运行队列保存。那么此时就绪队列有 50 个 PCB，某些结束了运行，则将它们的 PCB 插入空闲队列中；此时又有一个进程需要运行，我们可以从空闲队列中取出一个 PCB 并插入运行队列并将之前的进程从运行队列取出插入就绪队列。上述各种状态的改变都伴随着对 PCB 中相关参数的修改。

**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？

如果相同进程对应相同的内核堆栈，那么我们在调度时无法根据选择调度执行就绪态的程序，因为我们不知道该进程的执行线索在堆栈哪个位置；如果是每个进程有其相应的内核栈，则我们调度执行该进程时可以去其内核栈上找之前的执行状态。

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

stackTop 指向进程对应的内核栈的栈顶，取地址赋值给 stackTop 是为了找到被打断进程的  
执行状态，从而后续能够为调度到其他程序或者不发生调度，返回到原来的进程被打断前的  
状态。

**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）**

中断嵌套时，例如我们开始在 user space，然后发生时钟中断，硬件发生任务转换，进行  
堆栈的切换以及特权级的检查，然后保存 eip, cs, eflags 到该进程的内核栈上，就在刚刚保  
存完状态寄存器的值，又发生了中断，此时硬件发现特权级一致，那么不进行堆栈切换，但  
是还是要保存此时的各类的状态寄存器在这个进程的内核栈上，并且会修改 stackTop 的值，  
即此时内核栈上有两次运行的状态寄存器的值。那么第二次中断返回时，会根据 stackTop  
即内核栈 pop 出第一次中断的状态然后继续进行第一次中断，而后同理完成第一次中断处理  
后返回到 user space。

**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.....）**

比函数更小的粒度是语句，而更大的粒度则是进程，线程功能性处于二者之间，则以函数为  
粒度执行。

**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）**

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

int main()
{
    int pid = fork();
    if (pid > 0)
    {
        for (int i = 0; i < 4; i++)
        {
            printf("father process prints %d\n", i);
            sleep(1);
        }
        exit(1);
    }
    if (pid == 0)
    {
        for (int i = 0; i < 4; i++)
        {
            printf("child process prints %d\n", i);
            sleep(1);
        }
        exit(1);
    }
}
```

```
oslab@oslab-VirtualBox:~/桌面/lab3_test/test/test1$ make run
gcc -o mul_pro mul_pro.c
./mul_pro
father process prints 0
child process prints 0
father process prints 1
child process prints 1
father process prints 2
child process prints 2
child process prints 3
father process prints 3
makefile:4: recipe for target 'run' failed
make: *** [run] Error 1
```

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）

`loadelf` 函数的代码在内核区域，并且 `load` 过程是将程序则加载在 `current` 用户进程的内存区域（是远离操作系统的代码区的），所以是不会影响 `loadelf` 函数的代码的。

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

`glibc` 中的 `pthread_create` 涉及到了多个函数且任务有以下几点：

初始化线程属性

为线程分配栈空间 `ALLOCATE_STACK()`

启动线程 `create_thread()`