

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）。

答：

会有以下结果，父子进程可能会每隔一段时间同时输出同一个 i 的值，并且逐步递减，直到为 0。

```
oslab@oslab-VirtualBox:~/桌面/lab3_all$ ./a.out
Father Process: Ping 1, 7;
Child Process: Pong 2, 7;
Child Process: Pong 2, 6;
Father Process: Ping 1, 6;
Father Process: Ping 1, 5;
Child Process: Pong 2, 5;
Child Process: Pong 2, 4;
Father Process: Ping 1, 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;
Child Process: Pong 2, 0;
Father Process: Ping 1, 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）？

答：

在处理器和存储器之间添加一个新的硬件模块 MMU 来实现虚存的管理。

分段式虚存管理：

在 MMU 中实现一个段基址寄存器，不同的进程在段基址中设置不同的值，这样物理地址=虚拟地址+基址。

分页式虚存管理：

需要的数据结构：

页目录(一个数组，每个元素的值为页表的起始地址)

页表：同样是一个数组，每个表项储存了虚拟页号、在不在主存、对应哪个物理页

维护过程

将一页约定为 4KB，将虚拟地址划分为虚拟页号(高 20 位)和页内地址(低 12 位)。

每次经过虚拟页号的两级页表查询，得到相应的物理页号。然后物理页号和页内地址拼接即可得到物理地址。若对应的物理页缺失，则发生缺页异常，需要再去磁盘中读取相应的页装载至主存中。

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？

答：

可以构建一个链表 list，初始化为所有空闲 PCB 相连。

每次分配给进程 PCB 时就将链表头分配给它。

每次释放进程时就将该进程的 PCB 加入链表头。

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？

答：

因为不同用户进程的现场信息都保存在了对应的内核堆栈之中。如果所有都对应相同内核堆栈，则会导致存储的现场信息被覆盖，不能够正确的切换回相应用户进程的用户态了。

exercise5: `stackTop` 和 `prevStackTop` 分别是什么值？

答：

`stackTop` 是栈顶指针，等于 `StackFrame` 的基地址；

`prevStackTop` 保存待恢复的栈顶信息，等于 `StackFrame` 被全部 `pop` 出去之后的 `esp` 大小。

exercise6: 请说说在中断嵌套发生时，系统是如何运行的？

答：

(1)

首先第一层中断发生了特权级切换，则先从用户进程对应的 TSS 里面取出对应内核栈的 `ss` 和 `esp`，把用户栈的 `ss` 和 `esp` 入栈，依次把 `eflags`, `cs`, `eip` 入栈，按照中断门描述符进入相应处理程序。

接下来软件 `push errorcode`，把中断号入栈，`pusha`，把 `ds`, `es`, `fs`, `gs` 入栈，`esp` 入栈，调用 `irqhandle`。

`Irqhandle` 中将当前的 `esp` 保存至了 `stackTop` 中，并将旧 `stackTop` 保存在 `prevStacktop` 中。

(2)

当发生中断嵌套时，首先需要保存当前中断处理程序的现场信息。

因为此时没有发生特权级切换，故硬件依次把 `eflags`, `cs`, `eip` 入栈，按照中断门描述符进入相应处理程序。

接下来软件 `push errorcode`，把中断号入栈，`pusha`，把 `ds`, `es`, `fs`, `gs` 入栈，`esp` 入栈，调用 `irqhandle`。

`Irqhandle` 中将 `esp` 保存至了 `stackTop` 中，并将旧 `stackTop` 保存在 `prevStacktop` 中。

(3)

当内层中断处理完成之后，就会将 `esp` 恢复成 `stackTop` 存储的值，然后 `pop` 掉一系列信息，将 `prevStacktop` 赋给 `stacktop`，这样就回到了上一层中断的现场。

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.....）

答：因为函数的调用和返回实际上就是栈帧的创建和释放，这和线程的特性——共享同一进程的全局变量和堆区——相符合。

更大的粒度的便是一个程序，这样和进程其实没什么区别。

更小的粒度便是代码段，很难划分清楚这个线程的部分，所以还是要用函数进行封装更佳。

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）

答：编写如下程序。

```
int main(void) {
    int i = 0;
    int ret;

    for(i<3;++i)
    {
        ret = fork();
        if(ret==0)
            break;
        else if(ret<0)
        {
            printf("fork error\n");
            exit(1);
        }
        else
        {
            printf("fork success , ret = %d , i = %d\n",ret,i);
        }
    }

    if(i == 3)
        printf("Parent process:i = %d\n",i);
    else
        printf("Child process:i = %d\n",i);

    sleep(1);
    return 0;
}
```

```
oslab@oslab-VirtualBox:~/桌面/lab3_all$ ./a.out
fork success , ret = 5786 , i = 0
fork success , ret = 5787 , i = 1
fork success , ret = 5788 , i = 2
Parent process:i = 3
Child process:i = 1
Child process:i = 2
Child process:i = 0
```

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）

答：

因为 `loadelf` 存储在内核空间中，并不会被用户程序的装载覆盖掉。

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

答：

可以看到 `pthread_create` 的语法格式如下：

```
int pthread_create(pthread_t *thread,
                  const pthread_attr_t *attr,
                  void *(*start_routine) (void *),
                  void *arg);
```

在 `pthread_create()` 中，首先进行栈空间的分配、各种结构的拷贝、必要标志的设置，然后调用函数

```
int create_thread (struct pthread *pd, const struct pthread_attr *attr,
                  STACK_VARIABLES_PARMS)
```

而在 `create_thread()` 函数中，又调用了函数

```
do_clone (struct pthread *pd, const struct pthread_attr *attr,
          int clone_flags, int (*fct) (void *), STACK_VARIABLES_PARMS,
          int stopped)
```

所以实际上 Linux 中的线程创建最终是调用了 `clone` 函数实现的。

而 `clone` 函数中，则是通过 `syscall` 系统调用，切换至内核态，然后执行 `sys_fork()` 函数，进入相应的服务例程，进入 `do_fork()` 函数：

```
do_fork(unsigned long clone_flag, unsigned long stack_start, struct
pt_regs *regs, unsigned long stack_size, int _user *parent_tidptr, int
_user *child_tidptr);
```

`do_fork()` 函数则根据传入的标识符 `clone_flag` 设置，进行线程的创建，其中最主要是通过 `copy_process()` 完成大部分的创建工作，例如申请内存空间、创建 PCB、创建内核栈和用户栈、创建新的 `task_struct` 结构等。

进行完 `do_fork()` 函数，如此线程创建完毕，并可在可运行队列中等待调度执行。

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。

答：

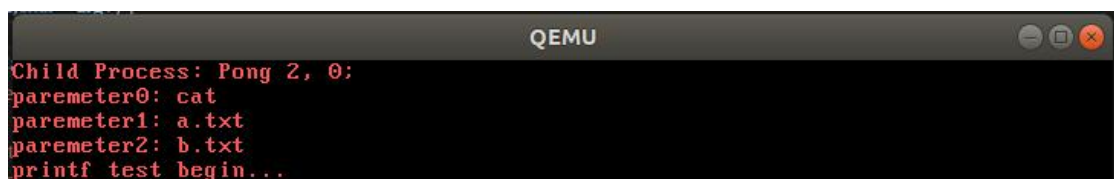
- 将 `exec` 定义如下：

```
int exec(uint32_t sec_start, uint32_t sec_num, int argc, char **argv)
```

- 调用 `syscall` 时也同时将 `argc` 和 `argv` 传递进去 (`argc` 和 `argv` 的意义和 C++ `main` 函数相同)
- 在 `syscallExec` 中, 首先取出 `argc` 和 `argv`, 因为 `argv` 指向的参数字符串数组存放在用户栈中, 所以我们首先将他们拷贝在核心栈中。
- 照常加载新的用户进程, 加载完成后, 将参数字符串从核心栈拷贝到新的用户栈中。
- 获取加载的用户进程对应的 `esp` 指针, 将 `esp+4` 设置为 `argc`, 将 `esp+8` 设置为一个指向参数字符串数组的指针即可
- 接着就可以在用户进程中修改 `exec` 进行相应的调用, 并在新加载的进程中测试参数是否传递正确。验证结果如下:

```
char argv[3][10]={"cat","a.txt","b.txt"};
char *p1[3];
for(int i=0;i<3;++i)
    p1[i]=argv[i];
exec(221, 20,3,p1);
```

```
int main(int argc,char**argv){
    for(int i=0;i<argc;++i)
        printf("parameter%d: %s\n",i,argv[i]);
}
```



```
QEMU
Child Process: Pong 2, 0:
parameter0: cat
parameter1: a.txt
parameter2: b.txt
printf test begin...
```

可以看到成功打印出了三个参数的内容。