

# 南京大学本科生实验报告

课程名称：操作系统

任课教师：叶保留

助教：尹熙喆/水兵

学院	计算机科学与技术	专业（方向）	计算机科学与技术
学号	201220062	姓名	黄子睿
Email	201220062@smail.nju.edu.cn	开始/完成日期	4/11-4/13

## 1. 实验名称：

OS LAB 3

## 2. 实验目的：

在 lab3 中，我们会实现进程调度，实现多进程同时运行。大家会与课堂上学习的理论知识相结合，实现一个自己设计的调度程序。具体内容是实现 fork, exit, sleep 等库函数和它们对应的系统调用！

## 3. 实验内容：

综述：

本次实验在基础功能之上，额外完成了分页机制的设计，内核级线程库的简略实现(create, exit, join)以及带参数的 exec 函数实现。

下面是内容索引：

[Exercise 1](#)

[Exercise 2](#)

[Exercise 3](#)

[Exercise 4](#)

[Exercise 5](#)

[Exercise 6](#)

[Exercise 7](#)

[Exercise 8](#)

[Exercise 9](#)

[Challenge 1](#)

[Challenge 2](#)

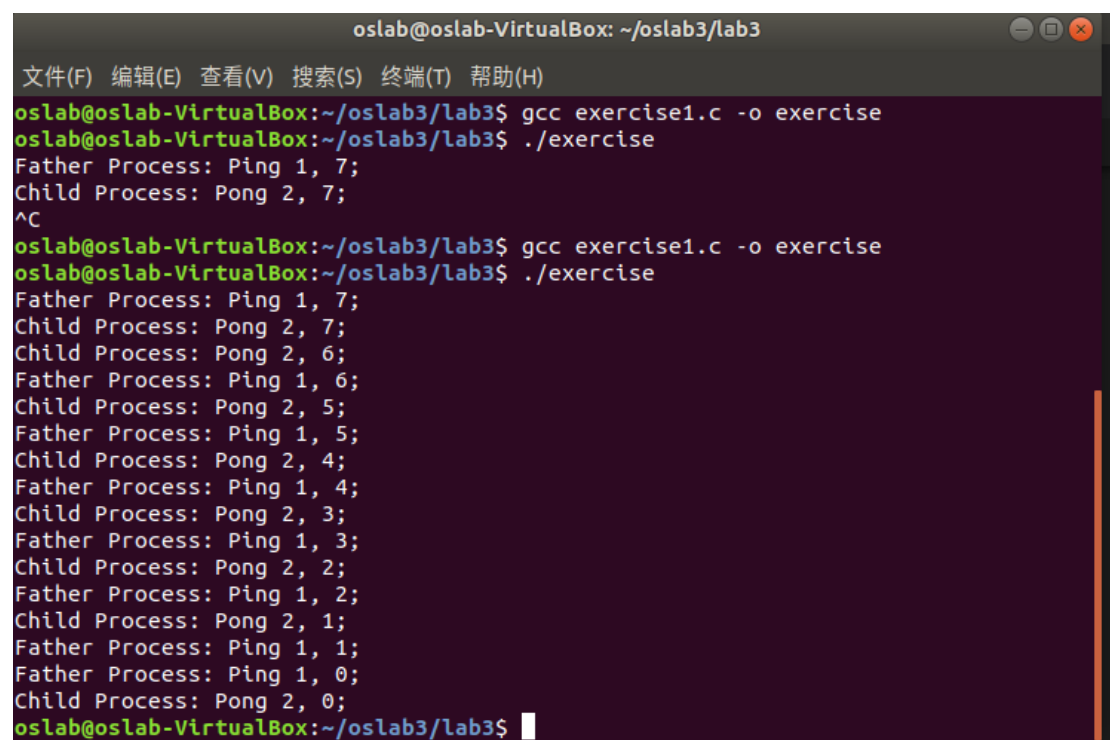
[Challenge 3](#)

[Challenge 4](#)

[Challenge 5](#)

**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：加上头文件之后编译运行，结果如下图所示。可以看到实际上确实如手册所说，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，因为 linux 中的 fork 函数采取 copy\_on\_write 计数，当一方执行写操作时数据被修改，子进程将拷贝得到独立的数据副本，此时父子进程的用户栈是独立的。



```
oslab@oslab-VirtualBox: ~/oslab3/lab3
文件(F) 编辑(E) 查看(V) 搜索(S) 终端(T) 帮助(H)
oslab@oslab-VirtualBox:~/oslab3/lab3$ gcc exercise1.c -o exercise
oslab@oslab-VirtualBox:~/oslab3/lab3$ ./exercise
Father Process: Ping 1, 7;
Child Process: Pong 2, 7;
^C
oslab@oslab-VirtualBox:~/oslab3/lab3$ gcc exercise1.c -o exercise
oslab@oslab-VirtualBox:~/oslab3/lab3$ ./exercise
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;
Child Process: Pong 2, 4;
Father Process: Ping 1, 4;
Child Process: Pong 2, 3;
Father Process: Ping 1, 3;
Child Process: Pong 2, 2;
Father Process: Ping 1, 2;
Child Process: Pong 2, 1;
Father Process: Ping 1, 1;
Father Process: Ping 1, 0;
Child Process: Pong 2, 0;
oslab@oslab-VirtualBox:~/oslab3/lab3$
```

## Exercise 2 与 Challenge 1:

### 分页式内存的设计:

我在这次试验里面实现了一个分页式虚存管理以及在分页基础上的 Copy-On-Write 技术。构思的时候感觉还行，但最后实现起来发现是真的复杂，比我一开始预计的要难整的多，主要是因为引入到不同的内存等级（PA 全程在 ring0 跑）与进程切换（PA 不用考虑，也就意味着 PA 的页表初始化完成后始终是静态

的，而这里涉及到进程切换，用户态的页表也要跟着变）之后，程序的行为比较难掌控。

最后我的分页实现还是尽力模仿了 Linux，但根据框架代码的设计简化了一些部分。比如考虑到框架代码中的所有进程不会使用超过 0x100000bit 以上的内存，所以 Linux 中的进程虚存管理数据结构就省略了。同时页表数量也不需要很多，因为程序运行的范围有限。

首先讲一下我的分页实现。为了实现分页，先定义页表项，即结构体 PageDescriptor 以及 1024 个页表项组成的页框 PageFrame：

```
1. union PageDescriptor{
2.     struct{
3.         uint32_t val;
4.     };
5.     struct{
6.         uint32_t p      : 1;
7.         uint32_t rw     : 1;
8.         uint32_t us     : 1;
9.         uint32_t pwt    : 1;
10.        uint32_t pcd     : 1;
11.        uint32_t access  : 1;
12.        uint32_t dirty   : 1;
13.        // real_rw and dontcare is actually zero:0 in real linux desc
            riptor
14.        uint32_t real_rw  :1;
15.        uint32_t dontcare :1;
16.        uint32_t avl     : 3;
17.        uint32_t base    : 20;
18.    };
19. };
20. typedef union PageDescriptor PageDescriptor;
21. struct PageFrame{
22.     PageDescriptor content[1024];
23. };

```

注意到内核态指令的虚拟地址都是 0x100000 开始的，不会超过 0x200000；而用户态的程序，无论其物理地址在何处，其虚拟地址总归是 0 到 0x100000 的范围。

而没有指令会访问到 0x00400000 以上的地址。因此，对于两级页表的结构，仅仅需要使用页目录 PageDir[0]，即只需要一张对应的页表 pageFrame 就可以了。而实际在运行时，只需要用到 pageFrame[0]到 pageFrame[0x100]的用户态以及 pageFrame[0x100]到 pageFrame[0x200]的内核态即可。为了内核态同时访问多个进程（实际上跑起来最多两个，需要 COW 技术实现的时候需要复制一下页框内容），在额外用 pageFrame[0x200]到 pageFrame[0x300]的空间用作备用空间就可以了。

每次进程切换的时候，需要将对应进程的页表项调入实际页表中。因此，修改 ProcessTable，增加该进程对应的进程页表拷贝：PageDescriptor pageTb[NR\_PAGES\_PER\_PROC]，NR\_PAGES\_PER\_PROC 是一个宏，值为 0x100，因为每个用户进程最多只占 0x100000 的内存空间。每次进程切换时，将这一页表拷贝到 pageFrame[0]到 pageFrame[0x100]的位置上，并且注意每次更新页表都需要重新赋值 cr3，以刷新 TLB。

页框的分配与释放都采用数组链表的结构维护，具体的操作过程如下 exercise3 所示，这里先略过。Kernel 需要维护空闲的页框组成的数组链表，每个进程都在对应的 pcb 中维护分配给自己的页框。

在分页的基础上，进一步实现 Copy-On-Write。当调用 fork()函数创建子进程的时候，不需要为子进程分配独立的物理页框，让它直接访问父进程的物理页框，当需要对这些物理内存进行写操作的时候，再为子进程分配独立的物理页框拷贝。因此需要将父进程所有页表项的 rw 位（表示是否可写）清零，表示不能写，同时设计 PageFaultHandler 函数以处理缺页或也读写故障，在页故障处理函数中进行页故障分析与拷贝恢复的工作。为了恢复 rw 位被清零的可写页面，需

要记录 rw 被清零的页面的实际 rw 位。这里观察到，现实中的 pageDescriptor 中有两位空闲位，取其中一位，记为 real\_rw，用来记录实际上的 rw。这样在页故障处理中，只需要检查 real\_rw 位就可以判断是否需要执行写时拷贝（COW）操作，抑或报页访问错误。

在实现 COW 时，总拷贝一张页框对应的内容，这需要内核态同时访问两个用户进程的物理空间。首先内核需要为子进程分配独立的物理页框，改写其进程页表，在对物理页框中的内容进行逐字节拷贝。此时，current 进程的页表已经被复制在 pageFrame[0]到 pageFrame[0x100]之间，只需要将另一个用户进程的对应页表项赋值到 pageFrame[0x200]往上的部分就可以实现内核同时访问两个用户进程的物理空间。这部分内容的代码在 irqHandle.c 文件中，相关内容用宏 PAGE\_ENABLED 标志出来了。

COW 的实现不止于此，还包括对 ProcessTable 结构体的修改，对 syscallExit, syscallExec, syscallFork 的修改，增加进程状态 STATE\_ZOMBIE，修改 vga 逻辑（因为 vga 从 0xb800 开始，这恰好是用户进程运行时的逻辑地址），以及 initPage 等预处理等等相关操作，因为涉及的代码太多，无论代码还是逻辑上都很复杂，实验报告里不可能面面俱到地展示出来，如果想看助教哥哥还是去看代码吧，主要是 irqHandle.c 与 kvm.c 两个文件以及 memory.h 头，相关代码都用宏 PAGE\_ENABLED 标志出来了，这里就不再赘述了。

**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：通过一个 next 数组与 first 指针，利用数组指针的形式，可以在  $O(1)$  的时间复杂度与  $O(n)$  的空间复杂度下完成空闲 PCB 的查找。

```
1. unsigned int first = 1;
2. /*pcb[0]预留给 idle（即 init）进程，第一个空的 pcb 块从下标 1 开始*/
3. unsigned int next[MAX_PCB_NUM] = {0};
4. /*将 next 数组初始化为全零*/
5.
6. /*当有一个进程开始运行，正在申请空闲的 PCB 空间*/
7. if (first!=0){
8.     把 pcb[first]分配给该进程
9.     first = next[first]; //更新 first 指针
10. }
11. else{
12.     报错，无空闲的 pcb 块可用。
13. }
14.
15. /*当一个进程结束运行返回占用的 pcb 块，设其占用的编号为 i*/
16. next[i] = first;
17. first = i;
```

**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：不同的用户进程在系统调用的过程中也是并发的，有可能出现中断嵌套，因此它们的栈区（这里指内核栈）需要区分开来。

**Exercise5：**stackTop 有什么用？为什么一些地方要取地址赋值给 stackTop？

答：stackTop 在进程切换的时候用，总是记录对应进程最后一次面对中断压入栈的栈帧地址。由于进程切换都是从中断开始的，因此通过 stackTop 可以快速定位进程在内核态中最近一次的栈帧位置，并通过一系列返回操作从中断中返回进程继续执行。为了维护 pcb[i].stackTop 的值，每一次中断压入栈帧后都需要更新 stackTop 的值（即在 irqHandle 函数中的 pcb[current].stackTop=(uint32\_t)sf;），

方便在后续进程切换中返回。

考虑到 push 与 pop 的区别，stackTop 在复制上也有不同。如果是在进程调度中切换到一个新的进程，初始化 stackTop 应当是 &(pcb[num].stackTop)，方便进入内核态之后的一系列 push 操作；如果是需要从内核态中返回，则是 &(pcb[i].regs)，方便中断返回时的一系列 pop 操作。

**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）**

答：中断嵌套流程如下：

1、由于中断嵌套总是发生在内核态下，因此不涉及特权级转换。因此硬件只需要依次把 eflags, cs, eip 入栈，按照中断门描述符进入相应处理程序。

2、根据 irqHandle.S 中的内容，软件则需要：

如果硬件没有 push errorcode 的话，就 push errorcode

(1) 把中断号入栈

(2) pusha

(3) 把 ds, es, fs, gs 入栈

(4) esp 入栈（当做 StackFrame 结构体的指针！）

(5) 然后调用 irqHandle

**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：调度的角度来讲，比线程更大的粒度是进程，而比它小的粒度是单条指令的

执行；从程序架构的角度讲，函数（子程序）组成了整个程序，而函数又由语句（指令）组合而成。

程序对应了进程，进程是程序的一次执行，需要为它分配对应的内存空间，包括数据区（代码、全局变量、用户栈等）、核心栈区、控制块（pcb）等内存空间，一次调度消耗较大；而单条指令的执行只需要 CPU 就可以了，是运算的最小单位，只能够串行，不存在调度的必要。

因此，综合程序的性能考虑，基于栈操作的函数进行线程的调度是最合适的。

**Exercise 8** 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。

答：结合我在本次实验中实现的分页机制、内核级线程以及带参数的 `exec()` 函数，我编写了一个建议的测试 `main` 函数，在 `oslab/lab3/app_exp/main.c` 中可以看到源代码，这里不再额外复制，仅仅展示运行后的截图，具体逻辑与行为的对应最好还是看看 `app_exp/main.c` 里的源代码。

具体的程序流程是首先执行 `app/main.c` 中的 `ping`, `pong`, 再用带参数的 `exec()` 函数装载 `app_exp/main.c` 中的程序，打印传入的字符串参数，具体结果如下图：

```
QEMU
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:

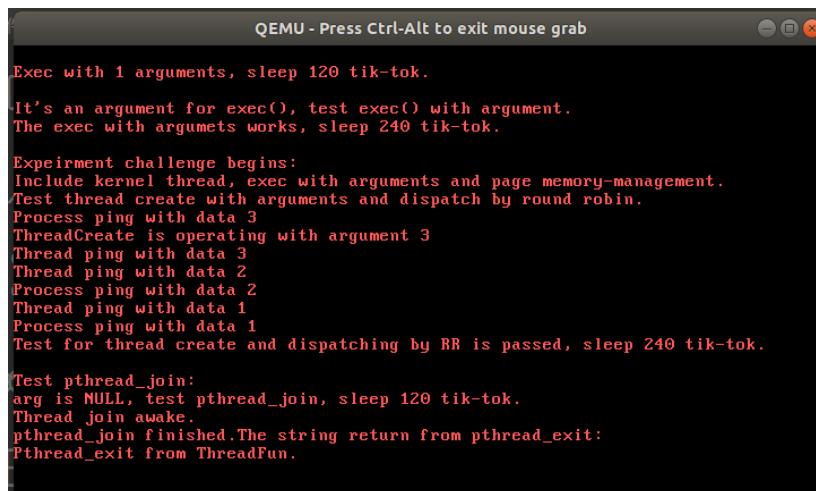
Exec with 1 arguments, sleep 120 tik-tok.

It's an argument for exec(), test exec() with argument.
The exec with arguments works, sleep 240 tik-tok.

Expeirment challenge begins:
Include kernel thread, exec with arguments and page memory-management.
Test thread create with arguments and dispatch by round robin.
Process ping with data 3
ThreadCreate is operating with argument 3
Thread ping with data 3
-
```



随后程序会调用内核级线程库的 `pthread_create` 函数生成一个线程，这个线程与父进程一起以 `round-robin` 的方式调度，分别打印 `Process ping with` 与 `Thread ping with`。之后调用 `pthread_join` 函数，实现 `vfork` 的功能；最后调用 `pthread_exit` 函数实现线程返回值得功能。具体结果如下图。内核级线程的函数实现将在下文 `challenge2` 中阐述。



```
QEMU - Press Ctrl-Alt to exit mouse grab

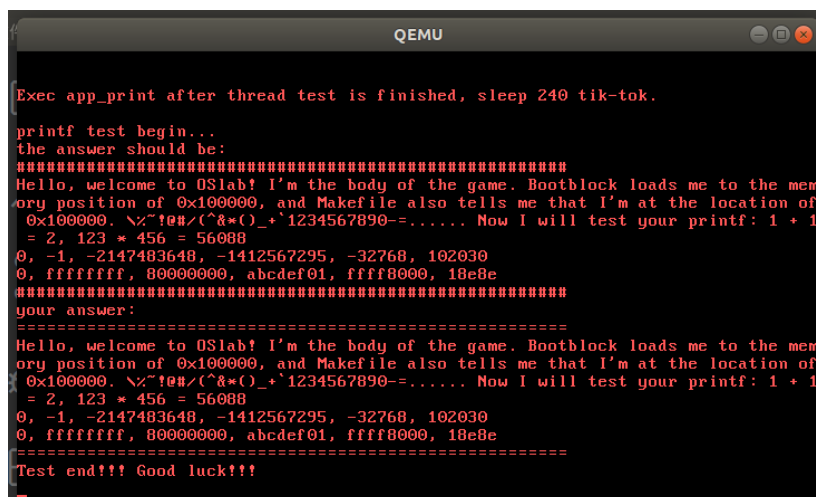
Exec with 1 arguments, sleep 120 tik-tok.

It's an argument for exec(), test exec() with argument.
The exec with argumets works, sleep 240 tik-tok.

Expeirment challenge begins:
Include kernel thread, exec with arguments and page memory-management.
Test thread create with arguments and dispatch by round robin.
Process ping with data 3
ThreadCreate is operating with argument 3
Thread ping with data 3
Thread ping with data 2
Process ping with data 2
Thread ping with data 1
Process ping with data 1
Test for thread create and dispatching by RR is passed, sleep 240 tik-tok.

Test pthread_join:
arg is NULL, test pthread_join, sleep 120 tik-tok.
Thread join awake.
pthread_join finished.The string return from pthread_exit:
Pthread_exit from ThreadFun.
```

最后再调用 `exec`，切换到 `app_print/main.c` 中运行。



```
QEMU

Exec app_print after thread test is finished, sleep 240 tik-tok.

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?!\0#/"^&*()_+'1234567890-=..... Now I will test your printf: 1 + 1
= 2, 123 * 456 = 56088
0, -1, -2147483648, -1412567295, -32768, 102030
0, ffffffff, 80000000, abcdef01, ffff8000, 18e8e
#####
your answer:
=====
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?!\0#/"^&*()_+'1234567890-=..... Now I will test your printf: 1 + 1
= 2, 123 * 456 = 56088
0, -1, -2147483648, -1412567295, -32768, 102030
0, ffffffff, 80000000, abcdef01, ffff8000, 18e8e
=====
Test end!!! Good luck!!!
```

**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：不会，`loadelf` 是内核段里面的函数，不会被对用户段的操作覆盖。

**Challenge2:** 请说说内核级线程库中的 `pthread_create` 是如何实现即可。

答: Linux 线程库的 `pthread_create` 的函数申明为:

```
int pthread_create(pthread_t *tidp, const pthread_attr_t *attr, (void*)(*start_rtn)(void*), void *arg);
```

其中第一个参数为指向线程标识符的指针, 第二个参数用来设置线程属性, 第三个参数是线程运行函数的起始地址, 第四个参数是运行函数的参数。函数在调用成功完成之后返回零。其他任何返回值都表示出现了错误。

如果忽略 `pthread_create` 函数的 `attr` 参数, 对其进行简要分析, 则其运行过程可以分为两步, 首先程序为线程分配用户栈空间, 在调用 `do_clone` 函数进行进程创建 (Linux 不额外区分线程与进程)。 `do_clone` 的过程实际上就是 `fork` 的过程, 这里不额外阐述了。

**Challenge3:** 你是否能够在框架代码中实现一个简易内核级线程库, 只需要实现 `create`, `destroy` 函数即可, 并仍然通过时钟中断进行调度, 并编写简易程序测试。

答: 我所实现的内核级线程库是在分页与 COW 的基础上实现的。具体实现了 `pthread_create`, `pthread_exit` 与 `pthread_join` 三个函数。三者的函数原型为:

```
1. int pthread_create(pthread_t tid, void* func, uint32_t argNum, ...);
2. int pthread_exit(void *retval);
3. int pthread_join(pthread_t tid, void** value_ptr);
```

具体声明信息请看 `lab3/lib/pthread.h` 头文件。

创建线程的逻辑与创建进程大同小异。创建线程 `systemPthreadCreate` 函数的实现基本与 `systemFok` 大体相同, 但根据线程共享数据区与代码段的原理, 在写时拷贝的背景下, 只需要将用户栈对应页表项的 `rw` 位指令, 方便后续执行 COW 功能即可。同时构造线程对应函数的参数, 需要将传入的参数按正确的顺

序压入线程对应用户栈中，构造函数运行时的栈帧就可以了。这种传递参数的思路在 exec 传参中也有体现。

线程的 pthread\_join 需要实现 vfork 标志对应的功能，即将调用该函数的父进程（或线程）的状态设置为 STATE\_BLOCKED，等待该线程运行结束才将父进程挂起，等待继续运行。同时 pthread\_join 需要将线程的返回值（由 pthread\_exit 返回）传递给父进程。这些功能需要在结构体 ProcessTable 中添加一些表项，具体如下所示：

```
1. int join_pid;  
2. int join_retval;  
3. int waiter_pid;
```

其中 join\_pid 用来对父进程标志等待结束的线程对应的 pcb 号，默认值为-1；join\_retval 对应了线程的返回值；waiter\_pid 用来对线程标志对应的父进程，默认值为-1。

调用了 pthread\_join 的线程需要由 pthread\_exit 释放，它的一个参数就是返回值，需要在释放前传递给父进程。

三个函数正在内核中对应 syscallThreadCreate, syscallThreadExit, syscallThreadJoin 三个函数，具体实现不在这里阐述，需要的话可以参考 pthread.h, irqHandle.c 与 syscall.c 三个文件，对应的代码用 PTHREAD\_ENABLED 宏标志出来了。

内核级线程库的测试请看上文 Exercise8。

**Challenge4:** 在 create\_thread 函数里面，我们要用线性时间搜索空闲 tcb，你是否有更好的办法让它更快进行线程创建？

答：challenge4 简要回答以下第一个问题，可以用上文提到的数组链表的方式实

现时间复杂度  $O(1)$ ，空间复杂度为  $O(n)$  的算法。

**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。

答：在栈中模拟函数栈帧以传递参数的思路与 challenge2 中 `pthread_create` 传参的思路一致。但是 `exec` 特殊的地方在于作为参数的字符串（`main` 函数的两个函数参数是 `int argc, char** argv`）原本储存在进程对应的数据段中，但是 `exec` 将重置进程的所有内存信息，会导致字符串被新进程的数据覆盖。为了解决这个问题，我认为规定将字符串参数保存在 `0x50000` 以上的一段连续空间中。在具体执行时，首先将需要的参数字符串逐字拷贝到 `0x50000` 以上的空间，一般情况下这一段空间不会被新进程的数据段覆盖。再将拷贝后的字符串地址作为参数传递到新进程的用户栈中以构造函数栈帧。具体的代码实现请看 `irqHandle` 函数的 `syscallExec` 函数，里面给出了针对原来分段式框架的实现以及针对我模拟的分页模式的实现。