困难一：内核线程的理解以及内核线程与用户进程之间的区别（不仔细看实验指导书，理解不到位）。

在仔细看实验指导书之前，我以为内核线程与用户进程间除了运行权限之间的区别外，没有什么区别，以为内核线程切换时也需要内存空间进行切换，但这个理解是有问题的。

内核线程只运行在内核态而用户进程会在在用户态和内核态交替运行；所有内核线程直接使用共同的 ucore 内核内存空间，不需为每个内核线程维护单独的内存空间而用户进程需要维护各自的用户内存空间。从内存空间占用情况这个角度上看，我们可以把线程看作是一种共享内存空间的轻量级进程。

困难二：ucore 是否做到给每个新 fork 的线程一个唯一的 id?

我首先对这是不太理解的，我认为只要 get_pid 函数每次返回的 pid 不一样不就可以给每一个线程一个唯一的 id 了，但是问题有可能不是出现在唯一的 id 上，有可能是在分配 id 是出现中断，使分配的 id 产生错误。这里的具体做法实现一个锁机制，分配 id 前禁止中断，分配完后可以允许中断，这中间实现的就相当于一个原子操作。

困难三：switch_to，进程间通信

汇编代码的 switch_to 不太懂（没有学过汇编），但这个又比较重要，所以找相关博客了解了解。

```
switch_to:                      # switch_to(from, to)
    # save from's registers
```

```
    movl 4(%esp), %eax          #保存 from 的首地址
    popl 0(%eax)                #将返回值保存到 context 的 eip
    movl %esp, 4(%eax)          #保存 esp 的值到 context 的 esp
    movl %ebx, 8(%eax)          #保存 ebx 的值到 context 的 ebx
    movl %ecx, 12(%eax)         #保存 ecx 的值到 context 的 ecx
    movl %edx, 16(%eax)         #保存 edx 的值到 context 的 edx
    movl %esi, 20(%eax)         #保存 esi 的值到 context 的 esi
    movl %edi, 24(%eax)         #保存 edi 的值到 context 的 edi
    movl %ebp, 28(%eax)         #保存 ebp 的值到 context 的 ebp

    # restore to's registers
    movl 4(%esp), %eax          #保存 to 的首地址到 eax
    movl 28(%eax), %ebp         #保存 context 的 ebp 到 ebp 寄存器
    movl 24(%eax), %edi         #保存 context 的 ebp 到 ebp 寄存器
    movl 20(%eax), %esi         #保存 context 的 esi 到 esi 寄存器
    movl 16(%eax), %edx         #保存 context 的 edx 到 edx 寄存器
    movl 12(%eax), %ecx         #保存 context 的 ecx 到 ecx 寄存器
    movl 8(%eax), %ebx          #保存 context 的 ebx 到 ebx 寄存器
    movl 4(%eax), %esp          #保存 context 的 esp 到 esp 寄存器
    pushl 0(%eax)               #将 context 的 eip 压入栈中
    ret
```

switch_to 函数主要完成的是进程的上下文切换，先保存当前寄存器的值，然后再将下一进程的上下文信息保存到对于寄存器中。

吐槽一：汇编代码还是不太好理解，进程间切换的代码也比较绕。

吐槽二：相关的只是内容得认真看，不然会影响实验的理解。