# 内核级线程

难度系数：★★★★★

## 实验目的

- 深入理解线程和进程的区别；
- 全面实践进程、地址空间、程序执行体、PCB、调度、用户接口等操作系统概念；
- 获得在实际操作系统上设计与实现一个完整子系统的经验。

## 实验内容

1. 在Ubuntu下编写多线程的应用程序memtest.c，解决一个可并行运算的问题。
2. 在Linux 0.11的进程管理基础上，按照POSIX Threads(https://computing.llnl.gov/tutorials/pthreads/)标准实现内核级线程，使其能支持memtest.c的运行

### 多线程应用

编写一个多线程的应用程序（memtest.c），采用多线程并发的方式进行“并行”运算。该程序要解决的问题是“内存测试”，即测试一段内存是否可靠。这段内存在进程空间内分配，大小不应小于1MB。

#### 内存测试算法

对任一被测内存单元，先写入一个数，再从该单元读出，比较写入和读出的数是否相等。如果相等，说明该单元功能正常，否则就不正常。对一个单元，要用如下5种数据测试：

1. 全0，即0
2. 全1，即0xFF
3. 01010101，即0x55
4. 10101010，即0xAA
5. 随机数

#### 主线程功能

主线程就是进程建立后直接拥有的线程。它用命令行方式和用户交互，接受用户的命令，进行相应的工作，打印状态信息。命令包括：

1. times number：设置对每个内存单元进行测试的次数。number即次数。每次测试都要测完全部的5种数据。例如，命令“times 5”表示对每个内存单元，每种数据都要测5次。
2. thread number：设置进行测试工作的线程个数。number即个数。主线程并不算在number内。例如，命令“thread 2”表示同时建立2个工作线程进行并发测试。
3. go：按照设定的参数建立工作线程，开始测试。主线程仍然在命令行等待用户命令。
4. status：打印所有工作线程的工作状态，包括它们的当前进度和测试结果等。
5. abort：停止测试。让所有工作线程都退出。
6. exit：停止测试并退出程序。

#### 工作线程功能

该程序始终只有主线程和用户交互，工作线程即不打印信息，也不读用户的输入。主线程和工作线程之间通过全局变量进行信息传递。工作线程唯一的功能是对主线程分配给它的内存区间进行测试。

### 内核级线程

要做到，同一个进程下的各个线程之间要能共享除指令执行序列、栈、寄存器以外的一切资源。相关功能通过系统调用和函数库共同完成。系统调用负责内核内的相关工作，其接口可自定义，直接实现在Linux 0.11已有的源程序文件中。函数库（命名为pthread.c和pthread.h）和应用程序链接到一起，对系统调用进行更高级别的封装，供应用程序直接调用。函数库应至少包含如下函数：

**pthread_attr_init**

```
int pthread_attr_init(pthread_attr_t *attr);
```

用默认值初始化attr指向的pthread_attr_t结构。该数据是调用pthread_create()的第二个参数。pthread_attr_t主要定义了创建线程时需要用户提供的属性信息，pthread_create()根据这些信息创建线程。属性的具体内容可完全自定义。

函数成功时返回0，出错时返回错误号。

**pthread_create**

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

该函数用来创建一个线程。attr是创建线程时使用的各种属性，由pthread_attr_init()设定。当该线程被调度时会从函数start_routine（一段用户态代码）开始执行。arg做为参数被传递给start_routine。start_routine的原型为：

```
void * start_routine(void *arg);
```

如果线程创建成功，返回值0，并且把线程的ID值存放在thread中；当创建不成功时会返回一个错误号：EAGAIN表示系统缺乏足够的资源来创建线程，EINVAL表示attr结构中的属性值非法。

**pthread_exit**

```
void pthread_exit(void *value_ptr);
```

将调用该函数的线程销毁。它没有返回值，因为调用它的线程已经销毁，所以返回值没有任何地方可以“返回”。value_ptr是传给父线程的返回值，父线程调用pthread_join()可得到这个值。这是线程主动终止的唯一方式。

**pthread_join**

```
int pthread_join(pthread_t thread, void **value_ptr);
```

将调用它的线程阻塞，一直等到thread结束为止。其中thread为被等待的线程ID，value_ptr会接收到被等待线程通过pthread_exit()设置的返回值。

## 实验报告

如实现了内核级线程，请在实验报告中说明你的实现方法和编译步骤。如只完成了用户态应用，则在实验报告中说明你是如何实现abort功能的。

## 评分标准

- 用户态应用，60%
- 内核级线程，30%
- 实验报告，10%

## 实验提示

Pointers to Linux 0.11 code, Internet and your brain

### 内核级线程流派

内核级线程的实现主要有三个流派。

一个流派以Windows和Solaris为代表，线程是系统调度和管理执行体的基本单位，而进程的功能弱化为单纯的资源管理。每个进程至少有一个线程，同一个进程之内的线程通过PCB共享资源。

另一个流派以Linux和FreeBSD（移植的Linux线程库）为代表，仍然以进程为调度和资源管理的基本单位，但允许不同的进程之间共享全部虚拟地址空间。这样，共享地址空间的进程们只要再拥有自己独立的栈，就像线程一样了。这种实现方法叫做轻量级进程（Light Weight Process）。相对第一个流派而言，它的效率比较低。

最后一个流派其实是将进程与线程完全揉合在一起，多见于嵌入式系统中。在这种方式下，所有的进程都共享同一个地址空间，这样它们每一个都相当于一个线程。

### 如何使用函数库

如果将pthread.h和pthread.c看做是一个库，那么在memtest.c中包含pthread.h，然后用如下命令编译：

```
gcc -Wall -o memtest memtest.c pthread.c
```

这样，memtest.c和pthread.c就一起被编译、链接到了memtest。

理论上，memtest.c应该可以在任何内核版本大于2.6的系统上编译。在这样的系统上使用系统自带pthread库的方法是：

```
gcc -Wall -o memtest memtest.c -lpthread
```

memtest是一个可在Linux下运行的示例，供参考。