APUE - 10

阅读目的：了解线程的概念、线程的创建、及同步机制。
阅读时间：4 小时
阅读概况：第 11、12 章

第 11 章　线程

1. 线程与进程

程序或可执行文件是一个静态的实体，它只是一组指令的集合，没有执行的含义。进程是一个动态的实体。

一个进程可能包含多个线程，传统意义上的进程是多线程的一种特例，即该进程只包含一个线程。

- 进程是系统进行资源分配和调度的一个独立单位。
- 线程是 CPU 调度和分派的基本单位。

每个线程都包含有表示执行环境所必需的信息。其中包括进程中标识线程的线程 ID、一组寄存器、栈、调度优先级和策略、信号屏蔽字、error 变量以及线程私有数据。

进程之间彼此的地址空间是独立的，但线程会共享内存地址空间。同一个进程的多个线程共享一份**全局内存区域**，包括 **文件描述符、代码段、初始化数据段、未初始化数据段和动态分配的堆内存段。**

2. 线程的优点

- 创建线程花费的时间要少于创建进程花费的时间。
- 终止线程花费的时间要少于终止进程花费的时间。
- 线程之间上下文切换的开销，要小于进程之间的上下文切换。
- 线程之间数据的共享比进程之间的共享要简单。
- 发挥多核优势，充分利用CPU资源。

3. pthread 库

函数	功能描述
pthread_create	创建线程
pthread_self	获取线程 ID
pthread_equal	检查两个线程 ID 是否相等
pthread_exit	退出线程
pthread_join	等待线程退出
pthread_detach	设置线程状态为分离状态
pthraed_cancel	取消线程
pthread_cleanup_push pthread_cleanup_pop	线程退出，清理函数注册和执行

pthread_create

```
#include <pthread.h>

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

pthread_create 函数的第一个参数是 pthread_t 类型的指针，线程创建成功的话，会将分配的线程 ID 填入该指针指向的地址。线程的后续操作将使用该值作为线程的唯一标识。

pthread_selt

```c
#include <pthread.h>

pthread_t pthread_self(void);
```

pthread_t 类型的线程 ID，本质就是一个进程地址空间上的一个地址。

pthread_equal

```c
#include <pthread.h>

int pthread_equal(pthread_t t1, pthread_t t2);
```

返回值是 0 的时候，表示两个线程是同一个线程，非零值则表示不是同一个线程。

pthread_exit

```c
#include <pthread.h>

void pthread_exit(void *value_ptr);
```

value_ptr 是一个指针，存放线程的“临终遗言”。线程组内的其他线程可以通过调用 pthread_join 函数接收这个地址，从而获取到退出线程的临终遗言。如果线程退出时没有什么遗言，则可以直接传递 NULL 指针，如下所示:

```c
pthread_exit(NULL);
```

但是这里有一个问题，就是不能将遗言存放到线程的局部变量里，因为如果用户写的线程函数退出了，线程函数栈上的局部变量可能就不复存在了，线程的临终遗言也就无法被接收者读到。

pthread_join

```c
#include <pthread.h>

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

线程库提供了 pthread_join 函数，用来等待某线程的退出并接收它的返回值。这种操作被称为连接(joining)。

该函数第一个参数为要等待的线程的线程 ID，第二个参数用来接收返回值。
根据等待的线程是否退出，可得到如下两种情况:

- 等待的线程尚未退出，那么 pthread_join 的调用线程就会陷入阻塞。
- 等待的线程已经退出，那么 pthread_join 函数会将线程的退出值(void* 类型)存放到 retval 指针指向的位置。

pthread_detach

```c
#include <pthread.h>

int pthread_detach(pthread_t thread);
```

默认情况下，新创建的线程处于可连接(Joinable)的状态，可连接状态的线程退出后，需要对其执行连接操作，否则线程资源无法释放，从而造成资源泄漏。

pthread_detach 函数来将线程设置成已分离(detached)的状态，如果线程处于已分离的状态，那么线程退出时，系统将负责回收线程的资源。

> 所谓已分离，并不是指线程失去控制，不归线程组管理，而是指线程退出后，系统会自动释 放线程资源。若线程组内的任意线程执行了 exit 函数，即使是已分离的线程，也仍然会受到影响，一并退出。

pthread_cancel

```
int pthread_cancel(pthread_t thread);
```

一个线程可以通过调用该函数向另一个线程发送取消请求。这不是个阻塞型接口，发出请求后函数就立刻返回了，而不会等待目标线程退出之后才返回。

如果成功该函数返回 0，否则将错误码返回。对于 glibc 实现而言，调用 pthread_cancel 时，会向目标线程发送一个 SIGCANCEL 的信号。

4. 线程的属性

线程创建的第二个参数是 pthread_attr_t 类型的指针，pthread_attr_init 函数会将线程的属性重置成默认值。

```
pthread_attr_t attr;
pthread_attr_init(&attr);
```

线程的属性及默认值：

属性	默认值	说明
detachstare	PTHREAD_CREATE_JOINABLE	可分离状态
stackaddr	NULL	不指定线程栈的基址，由系统决定栈基址
stacksize	8M	默认线程栈大小为 8MB
guardsize	PAGESIZE	警戒缓冲区
priority	0	进程调度相关，优先级为 0
policy	SCHED_OTHER	进程调度相关，调度策略为 SCHED_OTHER
inheritsched	PTHREAD_INHERIT_SCHED	进程调度相关，继承启动进程的调度策略

5. 线程的终止

下面的三种方法中，线程会终止，但是进程不会终止(如果线程不是进程组里的最后一个线程的话):

- 创建线程时的 start_routine 函数执行了 return，并且返回指定值。
- 线程调用 pthread_exit。
- 其他线程调用了 pthread_cancel 函数取消了该线程

如果线程组中的任何一个线程调用了 exit 函数，或者主线程在 main 函数中执行了 return 语句，那么整个线程组内的所有线程都会终止。

6. 互斥量

大部分情况下，线程使用的数据都是局部变量，变量的地址在线程栈空间内，这种情况下，变量归属于单个线程，其他线程无法获取到这种变量。

但实际的情况是，很多变量都是多个线程共享的，这样的变量称为共享变量(shared variable)。可以通过数据的共享，完成多个线程之间的交互。

互斥量初始化

互斥量采用的是英文 mutual exclusive (互相排斥之意)的缩写，即 mutex。
正确地使用互斥量来保护共享数据，首先要定义和初始化互斥量。POSIX 提供了两种初始化互斥量的方法。

```
#include <pthread.h>
//静态
```

```
pthread_mutex_t mutex = PTHREAD_MUTEX_INITIALIZER;
//动态
int pthread_mutex_init(pthread_mutex_t *restrict mutex,
                       const pthread_mutexattr_t *restrict attr);
```

互斥量的销毁

```
int pthread_mutex_destroy(pthread_mutex_t *mutex);
```

- 使用 PTHREAD_MUTEX_INITIALIZER 初始化的互斥量无须销毁。
- 不要销毁一个已加锁的互斥量，或者是真正配合条件变量使用的互斥量。
- 已经销毁的互斥量，要确保后面不会有线程再尝试加锁。

互斥量的加锁与解锁

```
int pthread_mutex_lock(pthread_mutex_t *mutex);

int pthread_mutex_trylock(pthread_mutex_t *mutex);

int pthread_mutex_unlock(pthread_mutex_t *mutex);
```

7. 读写锁

很多时候，对共享变量的访问有以下特点:大多数情况下线程只是读取共享变量的值，并不修改，只有极少数情况下，线程才会真正地修改共享变量的值。

对于这种情况，读请求之间是无需同步的，它们之间的并发访问是安全的。然而写请求必须锁住读请求和其他写请求。

读写锁的行为：

当前锁状态	读锁请求	写锁请求
无锁	OK	OK
读锁	OK	阻塞
写锁	阻塞	阻塞

如果临界区比较大，读写锁高并发的优势就会显现出来，但是如果临界区非常小，读写锁的性能短板就会暴露出来。由于读写锁无论是加锁还是解锁，首先都会执行互斥操作，加上读写锁还 需要维护当前读者线程的个数、写锁等待线程的个数、读锁等待线程的个数，因此这就决定了读写锁的开销不会小于互斥量。

比较适合读写锁的场景是:**临界区的大小比较可观，绝大多数情况下是读，只有非常少的写。**

8. 条件等待

线程在条件不满足的情况下，主动让出互斥量给其他线程，线程在此处等待，等待条件的满足;一旦条件满足，线程就可以立刻被唤醒。

线程之所以可以安心等待，依赖的是其他线程的协作，它确信会有一个线程在发现条件满足以后，将向它发送信号，并且让出互斥量。如果其他线程不配合(不发信号，不让出互斥量)，这个主动让出互斥量并等待事件发生的线程就会一直等待下去。

通常，线程会对一个条件进行测试，如果条 件不满足，就等待(pthread_cond_wait)，或者等待一段有限的时间(pthread_cond_timedwait)。相关函数的定义如下:

```
int pthread_cond_wait(pthread_cond_t *restrict cond,
                      pthread_mutex_t *restrict mutex);

int pthread_cond_timedwait(pthread_cond_t *restrict cond,
                           pthread_mutex_t *restrict mutex,
```

```
                                const struct timespec *restrict abstime);
```

条件等待是线程间同步的一种手段，如果只有一个线程，条件不满足，那么只会一直等待下去， 所以必须要有一个线程通过某些操作，改变共享数据，使原先不满足的条件变得满足了，并且友好地 通知等待在条件变量上的线程。

条件不会无缘无故地突然变得满足了，必然会牵扯到共享数据的变化。所以一定要有互斥锁来保护。没有互斥锁，就无法安全地获取和修改共享数据。