线程

1.概念

简介

与进程（process）类似，线程（thread）是允许应用程序并发执行多个任务的一种机制。一个进程可以包含多个线程。同一个程序中的所有线程均会独立执行相同程序，且共享同一份全局内存区域，其中包括初始化数据段、未初始化数据段，以及堆内存段。（传统意义上的 UNIX 进程只是多线程程序的一个特例，该进程只包含一个线程）

进程是 CPU 分配资源的最小单位，线程是操作系统调度执行的最小单位。

线程是轻量级的进程（LWP：Light Weight Process），在 Linux 环境下线程的本质仍是进程。

查看指定进程的 LWP 号：ps –Lf pid

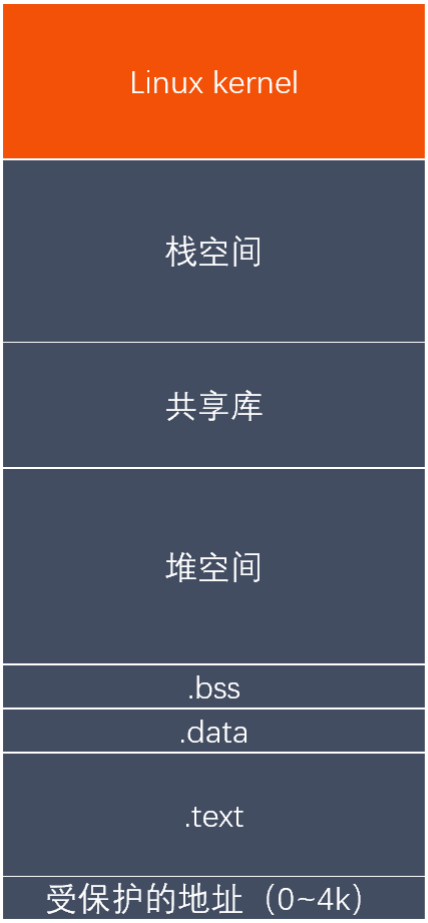
进程线程区别

进程间的信息难以共享。由于除去只读代码段外，父子进程并未共享内存，因此必须采用一些进程间通信方式，在进程间进行信息交换。

调用 fork() 来创建进程的代价相对较高，即便利用写时复制技术，仍然需要复制诸如内存页表和文件描述符表之类的多种进程属性，这意味着 fork() 调用在时间上的开销依然不菲。

线程之间能够方便、快速地共享信息。只需将数据复制到共享（全局或堆）变量中即可。

创建线程比创建进程通常要快 10 倍甚至更多。线程间是共享虚拟地址空间的，无需采用写时复制来复制内存，也无需复制页表。



线程之间共享和非共享资源

共享资源

- 进程 ID 和父进程 ID
- 进程组 ID 和会话 ID
- 用户 ID 和 用户组 ID
- 文件描述符表
- 信号处置
- 文件系统的相关信息：文件权限掩码（umask）、当前工作目录
- 虚拟地址空间（除栈、.text）

非共享资源

- 线程 ID
- 信号掩码
- 线程特有数据
- error 变量
- 实时调度策略和优先级
- 栈，本地变量和函数的调用链接信息

NPTL

当 Linux 最初开发时，在内核中并不能真正支持线程。但是它的确可以通过 clone() 系统调用将进程作为可调度的实体。这个调用创建了调用进程（calling process）的一个拷贝，这个拷贝与调用进程共享相同的地址空间。

LinuxThreads 项目使用这个调用来完成在用户空间模拟对线程的支持。不幸的是，这种方法有一些缺点，尤其是在信号处理、调度和进程间同步等方面都存在问题。另外，这个线程模型也不符合 POSIX 的要求。

要改进 LinuxThreads，需要内核的支持，并且重写线程库。有两个相互竞争的项目开始来满足这些要求。一个包括 IBM 的开发人员的团队开展了 NGPT（Next-Generation POSIX Threads）项目。同时，Red Hat 的一些开发人员开展了 NPTL 项目。NGPT 在 2003 年中期被放弃了，把这个领域完全留给了 NPTL。

NPTL，或称为 Native POSIX Thread Library，是 Linux 线程的一个新实现，它克服了 LinuxThreads 的缺点，同时也符合 POSIX 的需求。与 LinuxThreads 相比，它在性能和稳定性方面都提供了重大的改进。
查看当前 pthread 库版本：getconf GNU_LIBPTHREAD_VERSION

2. 线程操作函数

```
int pthread_create(pthread_t *thread, const pthread_attr_t *attr, void *
(*start_routine) (void *), void *arg);
pthread_t pthread_self(void);
int pthread_equal(pthread_t t1, pthread_t t2);
void pthread_exit(void *retval);
int pthread_join(pthread_t thread, void **retval);
int pthread_detach(pthread_t thread);
int pthread_cancel(pthread_t thread);
```

线程属性相关：

```
线程属性类型 pthread_attr_t
int pthread_attr_init(pthread_attr_t *attr);
int pthread_attr_destroy(pthread_attr_t *attr);
int pthread_attr_getdetachstate(const pthread_attr_t *attr, int *detachstate);
int pthread_attr_setdetachstate(pthread_attr_t *attr, int detachstate);
```

3. 线程同步

简介

线程的主要优势在于，能够通过全局变量来共享信息。不过，这种便捷的共享是有代价的：必须确保多个线程不会同时修改同一变量，或者某一线程不会读取正在由其他线程修改的变量。

临界区是指访问某一共享资源的代码片段，并且这段代码的执行应为原子操作，也就是同时访问同一共享资源的其他线程不应终端该片段的执行。

线程同步：即当有一个线程在对内存进行操作时，其他线程都不可以对这个内存地址进行操作，直到该线程完成操作，其他线程才能对该内存地址进行操作，而其他线程则处于等待状态。

互斥量

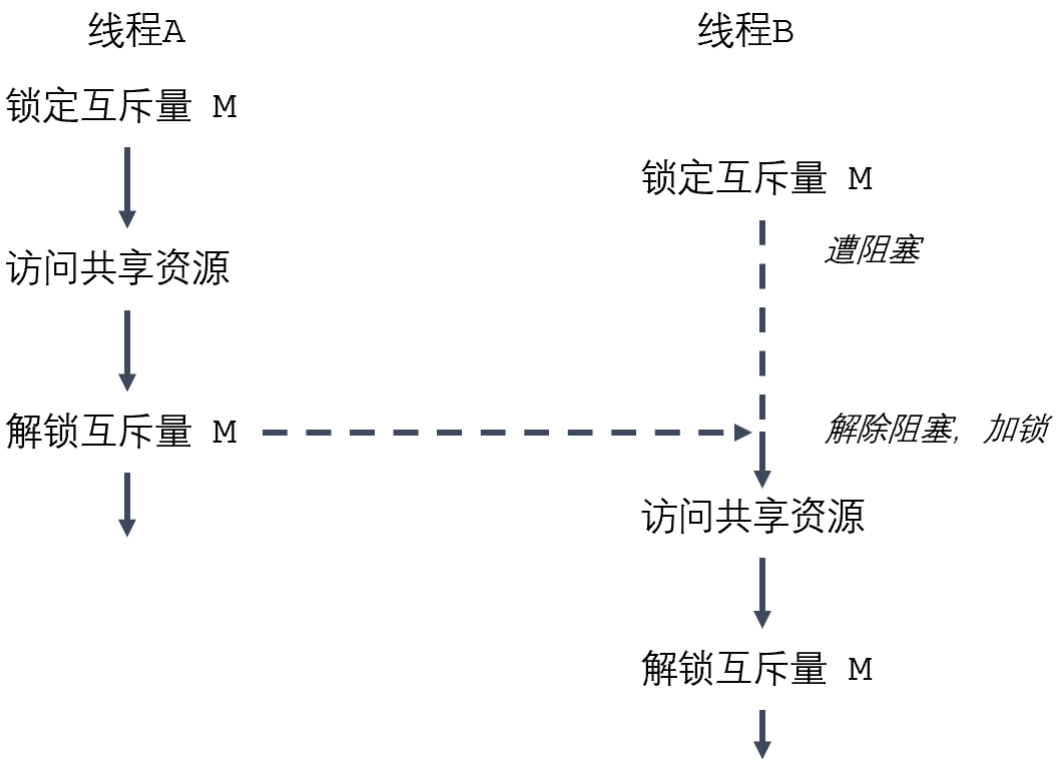
为避免线程更新共享变量时出现问题，可以使用互斥量（mutex 是 mutual exclusion的缩写）来确保同时仅有一个线程可以访问某项共享资源。可以使用互斥量来保证对任意共享资源的原子访问。

互斥量有两种状态：已锁定（locked）和未锁定（unlocked）。任何时候，至多只有一个线程可以锁定该互斥量。试图对已经锁定的某一互斥量再次加锁，将可能阻塞线程或者报错失败，具体取决于加锁时使用的方法。
一旦线程锁定互斥量，随即成为该互斥量的所有者，只有所有者才能给互斥量解锁。一般情况下，对每一共享资源（可能由多个相关变量组成）会使用不同的互斥量，每一线程在访问同一资源时将采用如下协议：

- 针对共享资源锁定互斥量
- 访问共享资源

- 对互斥量解锁

如果多个线程试图执行这一块代码（一个临界区），事实上只有一个线程能够持有该互斥量（其他线程将遭到阻塞），即同时只有一个线程能够进入这段代码区域，如下图所示：



互斥量相关操作函数：

```
互斥量的类型 pthread_mutex_t
int pthread_mutex_init(pthread_mutex_t *restrict mutex, const
pthread_mutexattr_t *restrict attr);
int pthread_mutex_destroy(pthread_mutex_t *mutex);
int pthread_mutex_lock(pthread_mutex_t *mutex);
int pthread_mutex_trylock(pthread_mutex_t *mutex);
int pthread_mutex_unlock(pthread_mutex_t *mutex);
```

死锁

有时，一个线程需要同时访问两个或更多不同的共享资源，而每个资源又都由不同的互斥量管理。当超过一个线程加锁同一组互斥量时，就有可能发生死锁。

两个或两个以上的进程在执行过程中，因争夺共享资源而造成的一种互相等待的现象，若无外力作用，它们都将无法推进下去。此时称系统处于死锁状态或系统产生了死锁。

死锁的几种场景：

- 忘记释放锁
- 重复加锁
- 多线程多锁，抢占锁资源

读写锁

当有一个线程已经持有互斥锁时，互斥锁将所有试图进入临界区的线程都阻塞住。但是考虑一种情形，当前持有互斥锁的线程只是要读访问共享资源，而同时有其它几个线程也想读取这个共享资源，但是由于互斥锁的排它性，所有其它线程都无法获取锁，也就无法读访问共享资源了，但是实际上多个线程同时读访问共享资源并不会导致问题。

在对数据的读写操作中，更多的是读操作，写操作较少，例如对数据库数据的读写应用。为了满足当前能够允许多个读出，但只允许一个写入的需求，线程提供了读写锁来实现。

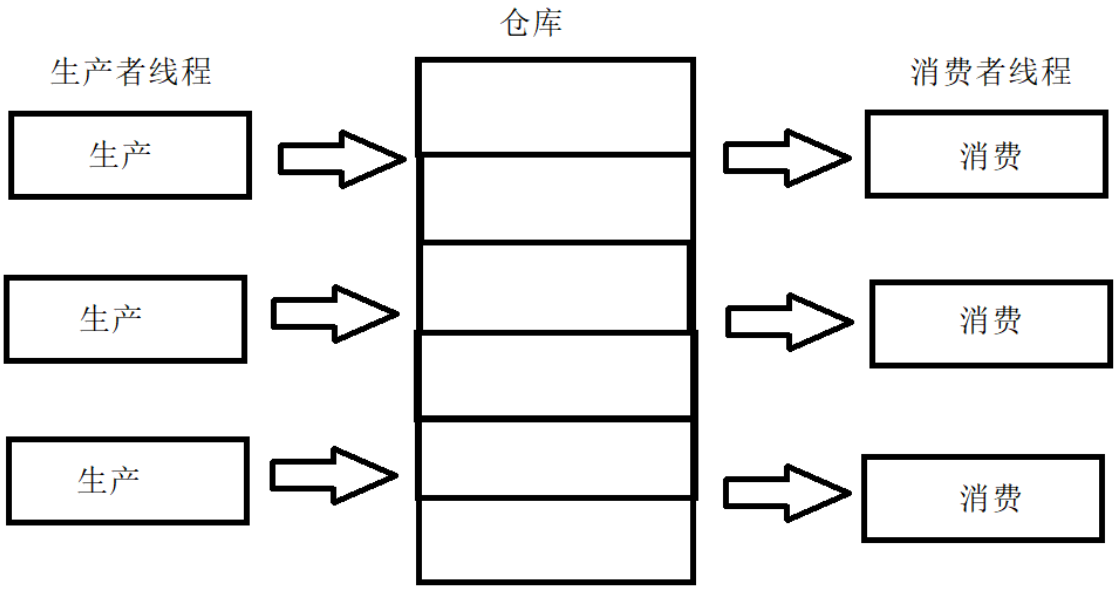
读写锁的特点：

- 如果有其它线程读数据，则允许其它线程执行读操作，但不允许写操作。
- 如果有其它线程写数据，则其它线程都不允许读、写操作。
- 写是独占的，写的优先级高。

读写锁相关函数：

```
读写锁的类型 pthread_rwlock_t
int pthread_rwlock_init(pthread_rwlock_t *restrict rwlock, const
pthread_rwlockattr_t *restrict attr);
int pthread_rwlock_destroy(pthread_rwlock_t *rwlock);
int pthread_rwlock_rdlock(pthread_rwlock_t *rwlock);
int pthread_rwlock_tryrdlock(pthread_rwlock_t *rwlock);
int pthread_rwlock_wrlock(pthread_rwlock_t *rwlock);
int pthread_rwlock_trywrlock(pthread_rwlock_t *rwlock);
int pthread_rwlock_unlock(pthread_rwlock_t *rwlock);
```

生产者消费者模型



条件变量

```
条件变量的类型 pthread_cond_t
int pthread_cond_init(pthread_cond_t *restrict cond, const pthread_condattr_t
*restrict attr);
int pthread_cond_destroy(pthread_cond_t *cond);
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);
int pthread_cond_signal(pthread_cond_t *cond);
int pthread_cond_broadcast(pthread_cond_t *cond);
```

信号量

```
信号量的类型 sem_t
int sem_init(sem_t *sem, int pshared, unsigned int value);
int sem_destroy(sem_t *sem);
int sem_wait(sem_t *sem);
int sem_trywait(sem_t *sem);
int sem_timedwait(sem_t *sem, const struct timespec *abs_timeout);
int sem_post(sem_t *sem);
int sem_getvalue(sem_t *sem, int *sval);
```