

# 一、 线程概述

## 01 / 线程概述

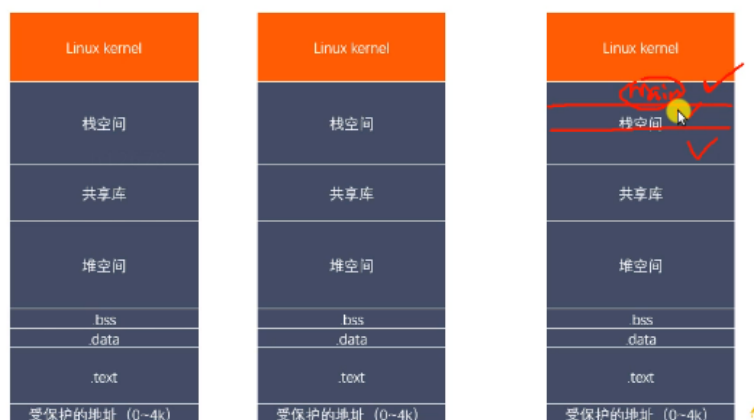
- 与进程 (process) 类似，线程 (thread) 是允许应用程序并发执行多个任务的一种机制。一个进程可以包含多个线程。同一个程序中的所有线程均会独立执行相同程序，且共享同一份全局内存区域，其中包括初始化数据段、未初始化数据段，以及堆内存段。（传统意义上的 UNIX 进程只是多线程程序的一个特例，该进程只包含一个线程）
- 进程是 CPU 分配资源的最小单位，线程是操作系统调度执行的最小单位。
- 线程是轻量级的进程 (LWP: Light Weight Process)，在 Linux 环境下载线程的本质仍是进程。
- 查看指定进程的 LWP 号: `ps -Lf pid`

`ps aux -> ps -LF pid`: 查看指定进程的线程。

## 02 / 线程和进程区别

- 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。

## 03 / 线程和进程虚拟地址空间



线程虚拟地址空间发生变化 (不共享): 栈空间, .text

## 04 / 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

## 05 / 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全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`



## 二、创建线程

Linux 环境下创建线程的方式：

- 1、使用 pthread 库：pthread 库是 POSIX 线程库的实现，是 Linux 下创建和管理线程的常用方式。
- 2、使用 C11 标准的 threads.h 库：C11 引入了一个新的线程库 threads.h，该库提供了一组函数来创建和管理线程。

### 3、 使用 C++11 标准的 thread 库: 和 C11 相同

06 / 线程操作

=

```
■ pthread_t pthread_self(void);  
■ int pthread_equal(pthread_t t1, pthread_t t2);  
■ int pthread_create(pthread_t *thread, const pthread_attr_t *attr,  
    void *(*start_routine) (void *), void *arg);  
■ 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);
```

```
// 由于pthread是第三方库，所以需要指定动态链接库的名称或者地址  
// Compile and link with -pthread.  
// gcc pthread_create.c -o create -l pthread  
// gcc pthread_create.c -o create -pthread
```

### 三、 终止线程

pthread 库:

1、线程函数返回: 线程函数可以通过返回来正常终止线程。当线程函数执行完毕并从函数中返回时, 线程会自动终止。这是最常见的线程终止方式。如 return NULL;

2、调用 pthread\_cancel 函数: 可以使用 pthread\_cancel 函数向指定的线程发送取消请求, 请求线程终止。被取消的线程需要在适当的位置检查取消请求并进行处理。

3、使用 pthread\_exit 函数: 可以在任意位置调用 pthread\_exit 函数来终止当前线程。这会立即终止线程的执行, 不会执行后续的代码。

thread 库:

在 C++11 中, 可以使用 `std::thread` 提供的方法来终止线程。然而, C++ 标准库没有提供直接的方式来终止线程, 因为线程的终止可能导致资源泄露或不一致的状态。相反, C++11 鼓励使用协作式的方式来终止线程。

1、通过退出线程函数: 线程函数可以通过正常返回来终止线程。当线程函数执行完毕并从函数中返回时, 线程会自动终止。如 `return;`

```
#include <iostream>
#include <thread>

void threadFunction()
{
    // 线程执行的代码

    // 返回线程的终止状态
    return;
}

int main()
{
    std::thread threadObj(threadFunction);
    threadObj.join(); // 等待线程终止
    std::cout << "Thread finished." << std::endl;
    return 0;
}
```

2、通过设置共享变量或标志位来控制线程终止: 可以使用共享的标志位或变量来控制线程的执行流程。线程函数可以在适当的位置检查这个标志位或变量, 并根据其值来决定是否终止线程。

```

std::atomic<bool> stopFlag(false);

void threadFunction()
{
    while (!stopFlag)
    {
        // 线程执行的代码
    }

    // 终止线程
    return;
}

int main()
{
    std::thread threadObj(threadFunction);

    // 设置标志位，请求线程终止
    stopFlag = true;

    threadObj.join(); // 等待线程终止
    std::cout << "Thread finished." << std::endl;
    return 0;
}

```

3、使用 `std::condition_variable` 来通知线程终止：可以使用条件变量来通知线程终止。线程函数在等待条件变量时，可以检查是否收到了终止的通知，如果收到通知，则终止线程。

```

std::mutex mtx;
std::condition_variable cv;
bool stopFlag = false;

void threadFunction()
{
    std::unique_lock<std::mutex> lock(mtx);
    cv.wait(lock, []{ return stopFlag; }); // 等待条件变量，并检查标志位

    // 终止线程
    return;
}

int main()
{
    std::thread threadObj(threadFunction);

    // 设置标志位，通知线程终止
    {
        std::lock_guard<std::mutex> lock(mtx);
        stopFlag = true;
    }
    cv.notify_one(); // 通知条件变量

    threadObj.join(); // 等待线程终止
    std::cout << "Thread finished." << std::endl;
    return 0;
}

```

#### 四、 线程回收

1、在 Linux 下，当主线程退出后，运行中的子线程的资源不会自动回收。这意味着子线程继续执行，但其资源（如栈空间、线程控制块等）将不会被自动释放。

—使用可分离线程（Detached Thread）：在创建子线程时，可以通过设置线程属性将其设置为可分离线程。可分离线程在终止后会自动回收其资源，不需要其他线程调用 `pthread_join` 来等待和回收子线程。

—其他子线程调用 `pthread_join` 函数去释放子线程的资源。(不常用)。

2、在 Linux 下，在主线程退出前，主线程不会自动回收已经终止的子线程的资源。主线程可以通过调用 `pthread_join` 来等待子线程的终止，并回收其资源。当子线程终止后，它的资源（如栈空间、线程控制块等）将保留在系统中，直到主线程调用 `pthread_join` 来回收这些资源。

—使用 `pthread_join` 等待子线程并回收资源：在主线程退出前，可以显式地调用 `pthread_join` 来等待子线程的终止，并回收其资源。需要注意的是，使用 `pthread_join` 等待子线程的终止可能会导致主线程阻塞，直到子线程终止。因此，确保在调用 `pthread_join` 前主线程已经完成了需要处理的任务，或者可以在主线程中设置适当的超时机制来避免无限阻塞。

#### 五、 连接已经终止的线程

Pthread\_join 函数。

## 六、 线程分离

当子线程先于 pthread\_detach()结束时，子线程的资源会等待到主线程结束时或整个进程结束时由操作系统自动回收，而不是在调用 pthread\_detach()之后立即回收。

## 七、 线程取消

调用 pthread\_cancel 函数：可以使用 pthread\_cancel 函数向指定的线程发送取消请求，请求线程终止。被取消的线程需要在适当的位置检查取消请求并进行处理。但是并不会立即终止，而是执行到取消点（系统调用）(man pthreads)的时候才终止。一般是在主线程中使用该函数去取消子线程。

## 八、 线程属性

### 07 / 线程属性

```
■ 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);
```

查看线程属性：man pthread\_attr\_+tab+tab

```
nana@nana-virtual-machine:~/linux/linux_thread$ man pthread_attr_
pthread_attr_destroy      pthread_attr_getsigmask_np  pthread_attr_setinheritsched
pthread_attr_getaffinity_np  pthread_attr_getstack      pthread_attr_setschedparam
pthread_attr_getdetachstate  pthread_attr_getstackaddr  pthread_attr_setschedpolicy
pthread_attr_getguardsize    pthread_attr_getstacksize  pthread_attr_setscope
pthread_attr_getinheritsched pthread_attr_init           pthread_attr_setsigmask_np
pthread_attr_getschedparam   pthread_attr_setaffinity_np pthread_attr_setstack
pthread_attr_getschedpolicy  pthread_attr_setdetachstate pthread_attr_setstackaddr
pthread_attr_getscope        pthread_attr_setguardsize  pthread_attr_setstacksize
nana@nana-virtual-machine:~/linux/linux_thread$ man pthread attr
```

显示当前用户的资源限制情况：ulimit -a

```
nana@nana-virtual-machine:~/linux/linux_thread$ ulimit -a
real-time non-blocking time (microseconds, -R) unlimited
core file size              (blocks, -c) 0
data seg size               (kbytes, -d) unlimited
scheduling priority         (-e) 0
file size                   (blocks, -f) unlimited
pending signals              (-i) 15188
max locked memory           (kbytes, -l) 495552
max memory size              (kbytes, -m) unlimited
open files                   (-n) 1024
pipe size                   (512 bytes, -p) 8
POSIX message queues         (bytes, -q) 819200
real-time priority          (-r) 0
stack size                   (kbytes, -s) 8192
cpu time                     (seconds, -t) unlimited
max user processes           (-u) 15188
virtual memory               (kbytes, -v) unlimited
file locks                   (-x) unlimited
nana@nana-virtual-machine:~/linux/linux_thread$
```

## 九、 线程同步

```
140701811603008 正在卖第 4 张票
140701811603008 正在卖第 1 张票
140701803210304 正在卖第 1 张票
140701794817600 正在卖第 -1 张票
```

为什么会出现以上问题？最终有可能会卖-2张票。(三个子线程售票)。

没有设置线程同步。解决方法：互斥锁；读写锁；条件变量；信号量；



- 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。

临界区是访问共享资源的代码段!!!!!!

## 十、 互斥锁

- 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

sudo apt-get install manpages-posix-dev。

### 03 / 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);`

访问互斥量的时候可能出现死锁。


### 04 / 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



## 十一、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。

 牛客网

## 06 / 读写锁相关操作函数

- 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);`

## 十二、生产者和消费者模型

生产者与生产者之间是互斥的，生产者与消费者之间是同步的，消费者与消费者之间是互斥的。

需要通过条件变量、信号量去建立生产者和消费者的通信。没有条件

变量或信号量的模型中，无法实现生产者与消费者之间的同步。

一对一模型：

容器满的时候，生产者阻塞；容器空的时候，消费者阻塞。

多对多模型：

互斥锁：多个生产者不能同时访问容器，多个消费者也不能同时访问容器。（生产者之间互斥，消费者之间互斥）

读写锁：多个生产者不能同时访问容器，多个消费者可以同时访问容器；或者：多个生产者可以同时访问容器，多个消费者不能同时访问容器。（生产者之间互斥，消费者之间同步；生产者之间同步，消费者之间互斥）。

### 十三、生成 core 文件的问题

```
/*
1、ulimit -a
|    // 查看系统受限资源
2、ulimit -c unlimited
|    // 设置core不受限
3、cat /proc/sys/kernel/core_pattern
|    // 查看系统对core文件的处理。
|    // 若显示 %p ... %E，说明系统对core进行了默认处理
4、sudo service apport stop
|    // 关闭系统支持服务。start 打开 restart 重启
5、gcc a.c -o a -g |//
6、gdb core
7、core-file core
8、set breakpoint
*/
```

## 十四、条件变量

### 08 / 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);`



## 十五、信号量

### 09 / 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);`

只是用信号量不能保证线程安全问题。需要和互斥锁联合使用。