[Linux/Unix 多线程通信](https://www.cnblogs.com/wuyepeng/p/9749956.html)

线程间无需特别的手段进行通信，因为线程间可以共享数据结构，也就是一个全局变量可以被两个线程同时使用。 不过要注意的是线程间需要做好同步，一般用 mutex。 可以参考一些比较新的 UNIX/Linux 编程的书，都会提到 Posix 线程编程，比如《UNIX环境高级编程（第二版）》、《UNIX系统编程》等等。 Linux 的消息属于 IPC，也就是进程间通信，线程用不上。

* 使用多线程的理由之一是和进程相比，它是一种非常”节俭”的多任务操作方式。

我们知道，在 Linux 系统下，启动一个新的进程必须分配给它独立的地址空间，建立众多的数据表来维护它的代码段、堆栈段和数据段，这是一种”昂贵”的多任务工作方式。 而运行于一个进程中的多个线程，它们彼此之间使用相同的地址空间，共享大部分数据，启动一个线程所花费的空间远远小于启动一个进程所花费的空间，而且，线程间彼此切换所需的时间也远远小于进程间切换所需要的时间。

* 使用多线程的理由之二是线程间方便的通信机制。

对不同进程来说，它们具有独立的数据空间，要进行数据的传递只能通过通信的方式进行，这种方式不仅费时，而且很不方便。 线程则不然，由于同一进程下的线程之间共享数据空间，所以一个线程的数据可以直接为其它线程所用，这不仅快捷，而且方便。 当然，数据的共享也带来其他一些问题，有的变量不能同时被两个线程所修改，有的子程序中声明为 static 的数据更有可能给多线程程序带来灾难性的打击，这些正是编写多线程程序时最需要注意的地方。

**Table of Contents**

* Table of Contents
* 通信方式
  + 信号
  + 互斥锁
  + 条件变量
  + 信号量
  + 信号量与线程锁、条件变量
* 简单的多线程程序
  + 修改线程的属性
  + 线程的数据处理
  + 互斥锁
  + 信号量

**通信方式**

**信号**

Linux 用 pthread\_kill 对线程发信号。

Windows 用 PostThreadMessage 进行线程间通信，但实际上极少用这种方法。还是利用同步多一些 LINUX 下的同步和 Windows 原理都是一样的。不过 Linux 下的 singal 中断也很好用。

用好信号量，共享资源就可以了。

**互斥锁**

互斥锁，是一种信号量，常用来防止两个进程或线程在同一时刻访问相同的共享资源。

需要的头文件：pthread.h

互斥锁标识符：pthread\_mutex\_t

1. 互斥锁初始化：

函数原型： int pthread\_mutex\_init (pthread\_mutex\_t\* mutex,const pthread\_mutexattr\_t\* mutexattr);

函数传入值: mutex：互斥锁。

mutexattr：

* + PTHREAD\_MUTEX\_INITIALIZER 创建快速互斥锁。
  + PTHREAD\_RECURSIVE\_MUTEX\_INITIALIZER\_NP 创建递归互斥锁。
  + PTHREAD\_ERRORCHECK\_MUTEX\_INITIALIZER\_NP 创建检错互斥锁。

函数返回值：成功：0；出错：-1

1. 互斥操作函数
   * int pthread\_mutex\_lock(pthread\_mutex\_t\* mutex); //上锁
   * int pthread\_mutex\_trylock (pthread\_mutex\_t\* mutex); //只有在互斥被锁住的情况下才阻塞
   * int pthread\_mutex\_unlock (pthread\_mutex\_t\* mutex); //解锁
   * int pthread\_mutex\_destroy (pthread\_mutex\_t\* mutex); //清除互斥锁

函数传入值：mutex：互斥锁。

函数返回值：成功：0；出错：-1

使用形式：

|  |  |
| --- | --- |
| 1  2  3  4  5  6 | pthread\_mutex\_t mutex;  pthread\_mutex\_init (&mutex, NULL); /\*定义\*/  ...  pthread\_mutex\_lock(&mutex); /\*获取互斥锁\*/  ... /\*临界资源\*/  pthread\_mutex\_unlock(&mutex); /\*释放互斥锁\*/ |

如果一个线程已经给一个互斥量上锁了，后来在操作的过程中又再次调用了该上锁的操作，那么该线程将会无限阻塞在这个地方，从而导致死锁。这就需要互斥量的属性。

互斥量分为下面三种：

* + 快速型。这种类型也是默认的类型。该线程的行为正如上面所说的。
  + 递归型。如果遇到我们上面所提到的死锁情况，同一线程循环给互斥量上锁，那么系统将会知道该上锁行为来自同一线程，那么就会同意线程给该互斥量上锁。
  + 错误检测型。如果该互斥量已经被上锁，那么后续的上锁将会失败而不会阻塞，pthread\_mutex\_lock()操作将会返回EDEADLK。

互斥量的属性类型为pthread\_mutexattr\_t。 声明后调用pthread\_mutexattr\_init()来创建该互斥量。然后调用 pthread\_mutexattr\_settype来设置属性。 格式如下：int pthread\_mutexattr\_settype(pthread\_mutexattr\_t \*attr, int kind);

第一个参数attr，就是前面声明的属性变量；第二个参数kind，就是我们要设置的属性类型。他有下面几个选项：

* + PTHREAD\_MUTEX\_FAST\_NP
  + PTHREAD\_MUTEX\_RECURSIVE\_NP
  + PTHREAD\_MUTEX\_ERRORCHECK\_NP

下面给出一个使用属性的简单过程：

|  |  |
| --- | --- |
| 1  2  3  4  5  6 | pthread\_mutex\_t mutex;  pthread\_mutexattr\_t attr;  pthread\_mutexattr\_init(&attr);  pthread\_mutexattr\_settype(&attr,PTHREAD\_MUTEX\_RECURSIVE\_NP);  pthread\_mutex\_init(&mutex,&attr);  pthread\_mutex\_destroy(&attr); |

前面我们提到在调用pthread\_mutex\_lock()的时候，如果此时mutex已经被其他线程上锁，那么该操作将会一直阻塞在这个地方。如果我们此时不想一直阻塞在这个地方，那么可以调用下面函数：pthread\_mutex\_trylock。

如果此时互斥量没有被上锁，那么pthread\_mutex\_trylock将会返回0，并会对该互斥量上锁。如果互斥量已经被上锁，那么会立刻返回EBUSY。

**条件变量**

需要的头文件：pthread.h

条件变量标识符：pthread\_cond\_t

1. 互斥锁的存在问题：

互斥锁一个明显的缺点是它只有两种状态：锁定和非锁定。设想一种简单情景：多个线程访问同一个共享资源时，并不知道何时应该使用共享资源，如果在临界区里 加入判断语句，或者可以有效，但一来效率不高，二来复杂环境下就难以编写了，这是我们需要一个结构，能在条件成立时触发相应线程，进行变量修改和访问。

1. 条件变量:

条件变量通过允许线程阻塞和等待另一个线程发送信号的方法弥补了互斥锁的不足，它常和互斥锁一起使用。使用时，条件变量被用来阻塞一个线程，当条件不满足时，线程往往解开相应的互斥锁并等待条件发生变化。一旦其它的某个线程改变了条件变量，它将通知相应的条件变量唤醒一个或多个正被此条件变量阻塞的线程。 这些线程将重新锁定互斥锁并重新测试条件是否满足。

1. 条件变量的相关函数
   * pthread\_cond\_t cond = PTHREAD\_COND\_INITIALIZER; //条件变量结构
   * int pthread\_cond\_init(pthread\_cond\_t *cond, pthread\_condattr\_t*cond\_attr);
   * int pthread\_cond\_signal(pthread\_cond\_t \*cond);
   * int pthread\_cond\_broadcast(pthread\_cond\_t \*cond);
   * int pthread\_cond\_wait(pthread\_cond\_t \*cond, pthread\_mutex\_t \*mutex);
   * int pthread\_cond\_timedwait(pthread\_cond\_t \*cond, pthread\_mutex\_t \*mutex,
   * const struct timespec \*abstime);
   * int pthread\_cond\_destroy(pthread\_cond\_t \*cond);

详细说明

1. 创建和注销

条件变量和互斥锁一样，都有静态动态两种创建方式

* + 静态方式

静态方式使用PTHREAD\_COND\_INITIALIZER常量，如下：

pthread\_cond\_t cond=PTHREAD\_COND\_INITIALIZER

* + 动态方式

动态方式调用pthread\_cond\_init()函数，API定义如下：

int pthread\_cond\_init(pthread\_cond\_t \*cond, pthread\_condattr\_t \*cond\_attr)

尽管POSIX标准中为条件变量定义了属性，但在LinuxThreads中没有实现，因此cond\_attr值通常为NULL，且被忽略。

注销一个条件变量需要调用pthread\_cond\_destroy()，只有在没有线程在该条件变量上等待的时候才能注销这个条件变量，否则返回 EBUSY。因为Linux实现的条件变量没有分配什么资源，所以注销动作只包括检查是否有等待线程。API定义如下：int pthread\_cond\_destroy(pthread\_cond\_t \*cond)

1. 等待和激发
   * 等待
     + int pthread\_cond\_wait(pthread\_cond\_t \*cond, pthread\_mutex\_t \*mutex) //等待
     + int pthread\_cond\_timedwait(pthread\_cond\_t \*cond, pthread\_mutex\_t \*mutex,
     + const struct timespec \*abstime) //有时等待

等待条件有两种方式：无条件等待pthread\_cond\_wait()和计时等待pthread\_cond\_timedwait()，其中计时等待方式 如果在给定时刻前条件没有满足，则返回ETIMEOUT，结束等待，其中abstime以与time()系统调用相同意义的绝对时间形式出现，0表示格林 尼治时间1970年1月1日0时0分0秒。

无论哪种等待方式，都必须和一个互斥锁配合，以防止多个线程同时请求pthread\_cond\_wait()（或 pthread\_cond\_timedwait()，下同）的竞争条件（Race Condition）。mutex互斥锁必须是普通锁（PTHREAD\_MUTEX\_TIMED\_NP）或者适应锁 （PTHREAD\_MUTEX\_ADAPTIVE\_NP），且在调用pthread\_cond\_wait()前必须由本线程加锁 （pthread\_mutex\_lock()），而在更新条件等待队列以前，mutex保持锁定状态，并在线程挂起进入等待前解锁。在条件满足从而离开 pthread\_cond\_wait()之前，mutex将被重新加锁，以与进入pthread\_cond\_wait()前的加锁动作对应。

* + 激发

激发条件有两种形式，pthread\_cond\_signal()激活一个等待该条件的线程，存在多个等待线程时按入队顺序激活其中一个；而pthread\_cond\_broadcast()则激活所有等待线程。</p>

1. 其他操作

pthread\_cond\_wait ()和pthread\_cond\_timedwait()都被实现为取消点，因此，在该处等待的线程将立即重新运行，在重新锁定mutex后离开 pthread\_cond\_wait()，然后执行取消动作。也就是说如果pthread\_cond\_wait()被取消，mutex是保持锁定状态的， 因而需要定义退出回调函数来为其解锁。

pthread\_cond\_wait实际上可以看作是以下几个动作的合体:

解锁线程锁；

等待条件为true；

加锁线程锁；

使用形式：

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11 | // 线程一代码  pthread\_mutex\_lock(&mutex);  if (条件满足)  pthread\_cond\_signal(&cond);  pthread\_mutex\_unlock(&mutex);  <p>// 线程二代码  pthread\_mutex\_lock(&mutex);  while (条件不满足)  pthread\_cond\_wait(&cond, &mutex);  pthread\_mutex\_unlock(&mutex);  /\*线程二中为什么使用while呢？因为在pthread\_cond\_signal和pthread\_cond\_wait返回之间，有时间差，假设在这 个时间差内，条件改变了，显然需要重新检查条件。也就是说在pthread\_cond\_wait被唤醒的时候可能该条件已经不成立。\*/ |

**信号量**

信号量其实就是一个计数器，也是一个整数。 每一次调用 wait 操作将会使 semaphore 值减一，而如果 semaphore 值已经为 0，则 wait 操作将会阻塞。 每一次调用post操作将会使semaphore值加一。

需要的头文件：semaphore.h

信号量标识符：sem\_t

主要函数：

sem\_init

功能： 用于创建一个信号量，并初始化信号量的值。

函数原型： int sem\_init (sem\_t\* sem, int pshared, unsigned int value);

函数传入值： sem：信号量。

pshared：决定信号量能否在几个进程间共享。 由于目前 Linux 还没有实现进程间共享信息量，所以这个值只能取0。 value：初始计算器

函数返回值： 0：成功；-1：失败。

其他函数

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9 | //等待信号量  int sem\_wait (sem\_t\* sem);  int sem\_trywait (sem\_t\* sem);  //发送信号量  int sem\_post (sem\_t\* sem);  //得到信号量值  int sem\_getvalue (sem\_t\* sem);  //删除信号量  int sem\_destroy (sem\_t\* sem); |

功能：sem\_wait和sem\_trywait相当于 P 操作，它们都能将信号量的值减一， 两者的区别在于若信号量的值小于零时，sem\_wait 将会阻塞进程，而 sem\_trywait 则会立即返回。

sem\_post 相当于 V 操作，它将信号量的值加一，同时发出唤醒的信号给等待的进程（或线程）。

sem\_getvalue 得到信号量的值。

sem\_destroy 摧毁信号量。

使用形式：

|  |  |
| --- | --- |
| 1  2  3  4  5  6 | sem\_t sem;  sem\_init(&sem, 0, 1); /\*信号量初始化\*/  ...  sem\_wait(&sem);   /\*等待信号量\*/  ... /\*临界资源\*/  sem\_post(&sem);   /\*释放信号量\*/ |

**信号量与线程锁、条件变量**

信号量与线程锁、条件变量相比还有以下几点不同：

1. 锁必须是同一个线程获取以及释放，否则会死锁。而条件变量和信号量则不必。
2. 信号的递增与减少会被系统自动记住，系统内部有一个计数器实现信号量，不必担心会丢失，而唤醒一个条件变量时，如果没有相应的线程在等待该条件变量，这次唤醒将被丢失。

**简单的多线程程序**

首先在主函数中，我们使用到了两个函数，pthread\_create 和 pthread\_join，并声明了一个 pthread\_t型的变量。

pthread\_t 在头文件 pthread.h 中已经声明，是线程的标示符

函数 pthread\_create 用来创建一个线程，函数原型：

|  |  |
| --- | --- |
| 1  2  3  4 | #include <pthread.h>    int pthread\_create(pthread\_t \*thread, const pthread\_attr\_t \*attr,    void \*(\*start\_routine) (void \*), void \*arg); |

* 第一个参数为指向线程标识符的指针，
* 第二个参数用来设置线程属性，
* 第三个参数是线程运行函数的起始地址，
* 最后一个参数是运行函数的参数。

若我们的函数thread不需要参数，所以最后一个参数设为空指针。第二个参数我们也设为空指针，这样将生成默认属性的线程。

返回值:

* 0: 当创建线程成功时
* non-zero: 说明创建线程失败，常见的错误返回代码为
  + EAGAIN 表示系统限制创建新的线程，例如线程数目过多了；
  + EINVAL 表示第二个参数代表的线程属性值非法。

创建线程成功后，新创建的线程则运行参数三和参数四确定的函数， 原来的线程则继续运行下一行代码。

函数pthread\_join用来等待一个线程的结束。函数原型为：

|  |  |
| --- | --- |
| 1  2  3 | #include <pthread.h>    int pthread\_join(pthread\_t thread, void \*\*retval); |

第一个参数为被等待的线程标识符，第二个参数为一个用户定义的指针，它可以用来存储被等待线程的返回值。这个函数是一个线程阻塞的函数，调用它的函数将一直等待到被等待的线程结束为止，当函数返回时，被等待线程的资源被收回。

一个线程的结束有两种途径，

* 一种是象我们上面的例子一样，函数结束了，调用它的线程也就结束了；
* 另一种方式是通过函数 pthread\_exit 来实现。它的函数原型为：

|  |  |
| --- | --- |
| 1  2  3 | #include <pthread.h>    void pthread\_exit(void \*retval); |

唯一的参数是函数的返回代码，只要 pthread\_join 中的第二个参数 retval 不是NULL，这个值将被传递给 retval。

最后要说明的是，一个线程不能被多个线程等待，否则第一个接收到信号的线程成功返回，其余调用 pthread\_join 的线程则返回错误代码 ESRCH。

**修改线程的属性**

设置线程绑定状态的函数为 pthread\_attr\_setscope，它有两个参数，第一个是指向属性结构的指针，第二个是绑定类型，它有两个取值：

* PTHREAD\_SCOPE\_SYSTEM（绑定的）
* PTHREAD\_SCOPE\_PROCESS（非绑定的）

下面的代码即创建了一个绑定的线程。

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10 | #include <pthread.h>    pthread\_attr\_t attr;  pthread\_t tid;    /\*初始化属性值，均设为默认值\*/  pthread\_attr\_init(&attr);  pthread\_attr\_setscope(&attr, PTHREAD\_SCOPE\_SYSTEM);    pthread\_create(&tid, &attr, (void \*)task\_func, NULL); |

**线程的数据处理**

和进程相比，线程的最大优点之一是数据的共享性，各个进程共享父进程处沿袭的数据段，可以方便的获得、修改数据。

但这也给多线程编程带来了许多问题。我们必须当心有多个不同的进程访问相同的变量。 许多函数是不可重入的，即同时不能运行一个函数的多个拷贝（除非使用不同的数据段）。 在函数中声明的静态变量常常带来问题，函数的返回值也会有问题。 因为如果返回的是函数内部静态声明的空间的地址，则在一个线程调用该函数得到地址后使用该地址指向的数据时，别的线程可能调用此函数并修改了这一段数据。 在进程中共享的变量必须用关键字 volatile 来定义，这是为了防止编译器在优化时（如 gcc 中使用 -OX 参数）改变它们的使用方式。 为了保护变量，我们必须使用信号量、互斥等方法来保证我们对变量的正确使用。

**互斥锁**

互斥锁用来保证一段时间内只有一个线程在执行一段代码。必要性显而易见：假设各个线程向同一个文件顺序写入数据，最后得到的结果一定是灾难性的

**信号量**

原来总是用互斥锁（MUTEX）和环境变量（cond）去控制线程的通信，用起来挺麻烦的，用信号量（SEM）来通信控制就方便多了！

用到信号量就要包含semaphore.h头文件。  
可以用sem\_t类型来声明一个型号量。

#include <semaphore.h>

sem\_t \* sem\_open(const char \*name, int oflag, ...);

用 int sem\_init(sem\_t \*sem, int pshared, unsigned int value) 函数来初始化型号量， 第一个参数就是用 sem\_t 声明的信号量， 第二变量如果为 ０，表示这个信号量只是当前进程中的型号量，如果不为 ０，这个信号量可能可以在两个进程中共享。 第三个参数就是初始化信号量的多少值。

* sem\_wait(sem\_t \*sem) 函数用于接受信号，当 sem > 0 时就能接受到信号，然后将 sem–;
* sem\_post(sem\_t \*sem) 函数可以增加信号量。
* sem\_destroy(sem\_t \*sem) 函数用于解除信号量。

以下是一个用信号控制的一个简单的例子:

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22  23  24  25  26  27  28  29  30  31  32  33  34  35 | #include <stdio.h>  #include <semaphore.h>  #include <pthread.h>    sem\_t sem1, sem2;    void \*thread1(void \*arg) {      sem\_wait(&sem1);      setbuf(stdout,NULL);//这里必须注意，由于下面输出"hello"中没有‘n’符，所以可能由于输出缓存已满，造成输不出东西来，所以用这个函数把输出缓存清空      printf("hello ");      sem\_post(&sem2);  }    void \*thread2(void \*arg) {      sem\_wait(&sem2);      printf("world!n");  }    int main() {      pthread\_t t1, t2;        sem\_init(&sem1,0,1);//初始化化信号量为1，所以会先打印线程1      sem\_init(&sem2,0,0);//初始化信号量为0        pthread\_create(&t1,NULL,thread1,NULL);      pthread\_create(&t2,NULL,thread2,NULL);        pthread\_join(t1,NULL);      pthread\_join(t2,NULL);        sem\_destroy(&sem1);      sem\_destroy(&sem2);        return 0;  } |

//程序的实现是控制先让thread1线程打印"hello "再让thread2线程打印"world!"

mutex互斥体只用于保护临界区的代码(访问共享资源),而不用于锁之间的同步，即一个线程释放mutex锁后，马上又可能获取同一个锁，而不管其它正在等待该mutex锁的其它线程。

semaphore信号量除了起到保护临界区的作用外，还用于锁同步的功能，即一个线程释放semaphore后，会保证正在等待该semaphore的线程优先执行，而不会马上在获取同一个semaphore。

如果两个线程想通过一个锁达到输出1,2,1,2,1,2这样的序列，应使用semaphore, 而使用 mutex 的结果可能为1,1,1,1,1,2,2,2,111…..。