# sem\_post

#include <semaphore.h>

函数原型

int sem\_post(sem\_t \*sem);

说明

sem\_post函数的作用是给[信号量](https://baike.baidu.com/item/%E4%BF%A1%E5%8F%B7%E9%87%8F" \t "_blank)的值加上一个“1”，它是一个“[原子操作](https://baike.baidu.com/item/%E5%8E%9F%E5%AD%90%E6%93%8D%E4%BD%9C)”－－－即同时对同一个信号量做加“1”操作的两个[线程](https://baike.baidu.com/item/%E7%BA%BF%E7%A8%8B" \t "_blank)是不会冲突的；而同 时对同一个文件进行读、加和写操作的两个程序就有可能会引起冲突。信号量的值永远会正确地加一个“2”－－因为有两个线程试图改变它。 当有线程阻塞在这个信号量上时，调用这个函数会使其中一个线程不在阻塞，选择机制是有线程的调度策略决定的。

返回值

sem\_post() 成功时返回 0；错误时，[信号量](https://baike.baidu.com/item/%E4%BF%A1%E5%8F%B7%E9%87%8F" \t "_blank)的值没有更改，-1 被返回，并设置 *errno* 来指明错误。

错误

EINVAL

sem 不是一个有效的[信号量](https://baike.baidu.com/item/%E4%BF%A1%E5%8F%B7%E9%87%8F" \t "_blank)。

EOVERFLOW

信号量允许的最大值将要被超过。

# 睡眠函数sleep、usleep

#include <unistd.h>

unsigned int sleep(unsigned int seconds);//睡眠秒

返回值：成功返回0,或者返回剩余的要睡眠的秒数（被signal中断后）.

int usleep(useconds\_t usec);//睡眠微秒

返回值：成功返回0,出错返回-1.

useconds\_t == unsigned int

秒的进制：

1秒 (s)= 1000毫秒(ms)

1毫秒 = 1000微秒(us)

1微秒 = 1000纳秒(ns)

# sem\_t

初始化信号量

它的原型为：　extern int sem\_init \_\_P ((sem\_t \*\_\_sem, int \_\_pshared, unsigned int \_\_[value](https://baike.baidu.com/item/value/2285610)));

头文件为： #include <semaphore.h>

sem为指向[信号量](https://baike.baidu.com/item/%E4%BF%A1%E5%8F%B7%E9%87%8F/9807501" \t "_blank)结构的一个指针；

pshared不为0时此信号量在进程间共享，否则只能为当前进程的所有[线程](https://baike.baidu.com/item/%E7%BA%BF%E7%A8%8B/103101" \t "_blank)共享；

value给出了信号量的初始值。

函数sem\_post( sem\_t \*sem )用来增加信号量的值当有[线程阻塞](https://baike.baidu.com/item/%E7%BA%BF%E7%A8%8B%E9%98%BB%E5%A1%9E/2233470" \t "_blank)在这个信号量上时，调用这个函数会使其中的一个线程不再阻塞，选择机制同样是由线程的调度策略决定的。

函数sem\_wait( sem\_t \*sem )被用来阻塞当前线程直到信号量sem的值大于0，解除阻塞后将sem的值减一，表明公共资源经使用后减少。

函数[sem\_trywait](https://baike.baidu.com/item/sem_trywait/3250669) ( sem\_t \*sem )是函数sem\_wait（）的非阻塞版本，它直接将[信号量](https://baike.baidu.com/item/%E4%BF%A1%E5%8F%B7%E9%87%8F/9807501" \t "_blank)sem的值减一。

函数sem\_destroy(sem\_t \*sem)用来释放信号量sem。[1]

相关应用

（1）[信号量](https://baike.baidu.com/item/%E4%BF%A1%E5%8F%B7%E9%87%8F/9807501)用sem\_init函数创建的，下面是它的说明：

#include<semaphore.h>

int sem\_init (sem\_t \*sem, int pshared, unsigned int value);

这个函数的作用是对由sem指定的信号量进行初始化，设置好它的共享选项，并指定一个整数类型的初始值。pshared参数控制着信号量的类型。如果 pshared的值是0，就表示它是当前进程的局部信号量；否则，其它进程就能够共享这个信号量。只对不让进程共享的信号量感兴趣。（这个参数受版本影响），　[Linux](https://baike.baidu.com/item/Linux/27050" \t "_blank)[线程](https://baike.baidu.com/item/%E7%BA%BF%E7%A8%8B/103101)一般不支持进程间共享[信号量](https://baike.baidu.com/item/%E4%BF%A1%E5%8F%B7%E9%87%8F/9807501)，pshared传递一个非零将会使函数返回ENOSYS错误。

（2）这两个函数控制着信号量的值，它们的定义如下所示：

#include <semaphore.h>

int sem\_wait(sem\_t \* sem);

int sem\_post(sem\_t \* sem);

这两个函数都要用一个由sem\_init调用初始化的信号量对象的指针做参数。

sem\_post函数的作用是给信号量的值加上一个“1”，它是一个“[原子操作](https://baike.baidu.com/item/%E5%8E%9F%E5%AD%90%E6%93%8D%E4%BD%9C/1880992" \t "_blank)"即同时对同一个[信号量](https://baike.baidu.com/item/%E4%BF%A1%E5%8F%B7%E9%87%8F/9807501" \t "_blank)做加“1”操作的两个线程是不会冲突的；而同时对同一个文件进行读、加和写操作的两个程序就有可能会引起冲突。信号量的值永远会正确地加一个“2”－－因为有两个线程试图改变它。

sem\_wait函数也是一个原子操作，它的作用是从信号量的值减去一个“1”，但它永远会先等待该信号量为一个非零值才开始做减法。也就是说，如果你对一个值为2的信号量调用sem\_wait(),线程将会继续执行，信号量的值将减到1。如果对一个值为0的信号量调用sem\_wait()，这个函数就会地等待直到有其它线程增加了这个值使它不再是0为止。如果有两个线程都在sem\_wait()中等待同一个信号量变成非零值，那么当它被第三个线程增加一个“1”时，等待线程中只有一个能够对[信号量](https://baike.baidu.com/item/%E4%BF%A1%E5%8F%B7%E9%87%8F/9807501" \t "_blank)做减法并继续执行，另一个还将处于等待状态。

信号量这种“只用一个函数就能原子化地测试和设置”的能力下正是它的价值所在。还有另外一个信号量函数[sem\_trywait](https://baike.baidu.com/item/sem_trywait/3250669" \t "_blank)，它是sem\_wait的非阻塞搭档。sem\_trywait是一个立即返回函数，不会因为任何事情阻塞。根据其[返回值](https://baike.baidu.com/item/%E8%BF%94%E5%9B%9E%E5%80%BC/9629649)得到不同的信息。如果返回值为0，说明信号量在该[函数调用](https://baike.baidu.com/item/%E5%87%BD%E6%95%B0%E8%B0%83%E7%94%A8/4127405" \t "_blank)之前大于0，但是调用之后会被该函数自动减1，至于调用之后是否为零则不得而知了。如果返回值为EAGAIN说明信号量计数为0。

（3） 获得信号量sem的值，并保存到valp中。下面的定义：

#include<semaphore.h>

int sem\_getvalue(sem\_t \*sem, int \*valp);

（4） 最后一个[信号量](https://baike.baidu.com/item/%E4%BF%A1%E5%8F%B7%E9%87%8F" \t "_blank)函数是sem\_destroy。这个函数的作用是在我们用完信号量对它进行清理。下面的定义：

#include<semaphore.h>

int sem\_destroy (sem\_t \*sem);

这个函数也使用一个信号量指针做参数，归还自己占据的一切资源。在清理信号量的时候如果还有线程在等待它，用户就会收到一个错误。

然而在linux的线程中，其实是没有任何资源关联到[信号量](https://baike.baidu.com/item/%E4%BF%A1%E5%8F%B7%E9%87%8F/9807501)对象需要释放的，因此在linux中，销毁信号量对象的作用仅仅是测试是否有线程因为该信号量在等待。如果函数返回0说明没有，正常注销信号量，如果返回EBUSY，说明还有线程正在等待该信号量的信号。

与其它的函数一样，这些函数在成功时都返回“0”。[1]

使用步骤

1.声明[信号量](https://baike.baidu.com/item/%E4%BF%A1%E5%8F%B7%E9%87%8F/9807501" \t "_blank)sem\_t sem1;

2.初始化信号量sem\_init(&sem1,0,1);

3.sem\_post和sem\_wait函数配合使用来达到[线程同步](https://baike.baidu.com/item/%E7%BA%BF%E7%A8%8B%E5%90%8C%E6%AD%A5/4855164" \t "_blank)

4.释放信号量int sem\_destroy (&sem1);[1]

语法：void srand (unsigned seed)；  
　　参数说明：unsigned seed:随机数产生器的初始值(种子值)。  
　　功能说明：srand设置产生一系列伪随机数发生器的起始点，要想把发生器重新初始化，可用1作seed值。任何共它的值都把发生器匿成一个随机的起始点。rand检索生成的伪随机数。在任何调用srand之前调用rand与以1作为seed调用srand产生相同的序列。[1]

此函数可以设定rand函数所用的随机数产生演算法的种子值。任何大于一的种子值都会将rand随机数所产生的虚拟随机数序列重新设定一个起始点。[2]

包含文件：stdlib. h  
　　返回值：无  
　　相关函数：rand，random ，randomize

# rand 和srand

srand函数是[随机数](https://baike.baidu.com/item/%E9%9A%8F%E6%9C%BA%E6%95%B0" \t "_blank)发生器的[初始化](https://baike.baidu.com/item/%E5%88%9D%E5%A7%8B%E5%8C%96)函数。原型：void srand(unsigned int seed);srand和[rand()](https://baike.baidu.com/item/rand%28%29" \t "_blank)配合使用产生伪[随机数](https://baike.baidu.com/item/%E9%9A%8F%E6%9C%BA%E6%95%B0" \t "_blank)序列。

**函数一：int rand(void)；**

功能：产生随机值，从srand (seed)中指定的seed开始，返回一个[seed, RAND\_MAX（0x7fff）)间的随机[整数](https://baike.baidu.com/item/%E6%95%B4%E6%95%B0/1293937" \t "_blank)。

**函数二：void srand(unsigned seed)；**

参数[seed](https://baike.baidu.com/item/seed/27784)是rand()的种子，用来初始化rand()的起始值。

可以认为rand()在每次被调用的时候，它会查看：

1） 如果用户在此之前调用过srand(seed)，给seed指定了一个值，那么它会自动调用srand(seed)一次来初始化它的起始值。

2） 如果用户在此之前没有调用过srand(seed)，它会自动调用srand(1)一次。

函数用法

[rand函数](https://baike.baidu.com/item/rand%E5%87%BD%E6%95%B0)在产生随机数前，需要系统提供的生成伪随机数序列的种子，rand根据这个种子的值产生一系列随机数。如果系统提供的种子没有变化，每次调用rand函数生成的伪随机数序列都是一样的。srand(unsigned seed)通过参数seed改变系统提供的种子值，从而可以使得每次调用rand函数生成的伪随机数序列不同，从而实现真正意义上的“随机”。通常可以利用系统时间来改变系统的种子值，即srand(time(NULL))，可以为rand函数提供不同的种子值，进而产生不同的随机数序列

参考链接：<https://baike.baidu.com/item/srand/796881?fr=aladdin>

# gcc中-pthread和-lpthread的区别

一般情况下，我们在链接一个（文件名为libxxx.so或libxxx.a等的）库时，会使用-lxxx的方式；在Linux中要用到多线程时，需要链接pthread库，按照惯例，我们应该使用-lpthread的方式来进行链接；但是，通过日常的观察，我发现很多开源代码都是使用了-pthread参数，而非使用-lpthread，这是为什么呢？

一通Google后，总算找到一些线索：

1. 为了可移植性：在Linux中，pthread是作为一个单独的库存在的（libpthread.so），但是在其他Unix变种中却不一定，比如在FreeBSD中是没有单独的pthread库的，因此在FreeBSD中不能使用-lpthread来链接pthread，而使用-pthread则不会存在这个问题，因为FreeBSD的编译器能正确将-pthread展开为该系统下的依赖参数。同样道理，其他不同的变种也会有这样那样的区别，如果使用-lpthread，则可能在移植到其他Unix变种中时会出现问题，为了保持较高的可移植性，我们最好还是使用-pthread（尽管这种做法未被接纳成为C标准，但已基本是事实标准）。
2. 添加额外的标志：在多数系统中，-pthread会被展开为“-D\_REENTRANT -lpthread”，即是除了链接pthread库外，还先定义了宏\_REENTRANT。定义这个宏的目的，是为了打开系统头文件中的各种多线程支持分支。比如，我们常常使用的错误码标志errno，如果没有定义\_REENTRANT，则实现为一个全局变量；若是定义了\_REENTRANT，则会实现为每线程独有，从而避免线程竞争错误。

详细链接：

<http://blog.chinaunix.net/uid-25909722-id-3026989.html>

<https://blog.csdn.net/skylinethj/article/details/38569243>

<https://blog.csdn.net/weixin_30617695/article/details/99408878>

<https://blog.csdn.net/jctian000/article/details/77740468>