[Linux互斥与同步应用（四）：posix信号量的互斥与同步](http://blog.csdn.net/shallnet/article/details/42109093)

标签： [mutex](http://www.csdn.net/tag/mutex)[sync](http://www.csdn.net/tag/sync)

2014-12-23 21:03 883人阅读 [评论](http://blog.csdn.net/shallnet/article/details/42109093#comments)(0) [收藏](javascript:void(0);) [举报](http://blog.csdn.net/shallnet/article/details/42109093#report)

http://static.blog.csdn.net/images/category_icon.jpg 分类：

linux同步与互斥应用系列（6） http://static.blog.csdn.net/images/arrow_triangle%20_down.jpg

版权声明：本文为博主原创文章，未经博主允许不得转载。如果您觉得文章对您有用，请点击文章下面“顶”。

       【版权声明：尊重原创，转载请保留出处：blog.csdn.net/shallnet 或 .../gentleliu，文章仅供学习交流，请勿用于商业用途】

         在前面讲共享内存的IPC时曾说共享内存本身不具备同步机制，如果要实现同步需要使用信号量等手段来实现之，现在我们就来说说使用posix的信号量来实现posix多进程共享内存的同步。其实信号量也可以使用在同一进程的不同线程之间。

        信号量是一个包含非负整数的变量，有两个原子操作，wait和post，也可以说是P操作和V操作，P操作将信号量减一，V操作将信号量加一。如果信号量大于0，P操作直接减一，否则如果等于0，P操作将调用进程（线程）阻塞起来，直到另外进程（线程）执行V操作。

        信号量可以理解为一种资源的数量，通过这种资源的分配来实现同步或者互斥。当信号量的值为1时，可以实现互斥的功能，此时信号量就是二值信号量，如果信号量的值大于一时，可以实现进程（线程）并发执行。信号量和互斥锁条件变量之间的区别是：互斥锁必须由给它上锁的进程（线程）来解锁，而信号灯P操作不必由执行过它V操作的进程（线程）来执行；互斥锁类似于二值信号量，要么加锁，要么解锁；当向条件变量发信号时，如果此时没有等待在该条件变量上的线程，信号将丢失，而信号量不会。信号量主要用于之间同步，但也可以用在线程之间。互斥锁和条件变量主要用于线程同步，但也可以用于进程间的同步。

        POSIX信号量是一个sem\_t的变量类型，分有名信号量和无名信号量，无名信号量用在共享信号量内存的情况下，比如同一个进程的不同线程之间，或父子进程之间，有名信号量随内核持续的，所以我们可以跨多个程序操作它们。

       1. 函数sem\_open创建一个有名信号量或打开一个已存在信号量。函数原型如下：

**[cpp]** [view plain](http://blog.csdn.net/shallnet/article/details/42109093) [copy](http://blog.csdn.net/shallnet/article/details/42109093)

1. #include <fcntl.h> /\* For O\_\* constants \*/
2. #include <sys/stat.h> /\* For mode constants \*/
3. #include <semaphore.h>
5. sem\_t \*sem\_open(**const** **char** \*name, **int** oflag);
6. sem\_t \*sem\_open(**const** **char** \*name, **int** oflag, mode\_t mode, unsigned **int** value);
7. Link with -pthread

参数name为posix IPC 名称， 参数oflag可以为0， O\_CREAT 或O\_CREAT | O\_EXCL， 如果指定了O\_CREAT，第三、四参数需要指定。mode为指定权限位，value参数指定信号量初始值，二值信号量的值通常为1，计数信号量的值通常大于1。

函数返回执行sem\_t数据类型的指针，出错返回SEM\_FAILED。

       2. 函数sem\_close关闭有名信号量。

**[cpp]** [view plain](http://blog.csdn.net/shallnet/article/details/42109093) [copy](http://blog.csdn.net/shallnet/article/details/42109093)

1. #include <semaphore.h>
2. **int** sem\_close(sem\_t \*sem);
3. Link with -pthread.

        3.函数sem\_unlink删除有名信号量。

**[cpp]** [view plain](http://blog.csdn.net/shallnet/article/details/42109093) [copy](http://blog.csdn.net/shallnet/article/details/42109093)

1. #include <semaphore.h>
2. **int** sem\_unlink(**const** **char** \*name);
3. Link with -pthread.

        4.函数sem\_wait和函数sem\_trywait测试信号量的值，如果信号量值大于0，函数将信号量的值减一并立即返回，如果信号量的值等于0，sem\_wait函数开始睡眠，直到信号量的值大于0，这是再减一并立即返回，而sem\_trywait函数不睡眠直接返回，并返回EAGAIN错误。函数原型为：

**[cpp]** [view plain](http://blog.csdn.net/shallnet/article/details/42109093) [copy](http://blog.csdn.net/shallnet/article/details/42109093)

1. #include <semaphore.h>
2. **int** sem\_wait(sem\_t \*sem);
3. **int** sem\_trywait(sem\_t \*sem);
4. **int** sem\_timedwait(sem\_t \*sem, **const** **struct** timespec \*abs\_timeout);
5. Link with -pthread.

        5. 函数sem\_post执行和sem\_wait相反的操作。当一进程（线程）执行完操作之后，应该调用sem\_post，将指定信号量加一，然后唤醒正在等待该信号量变为1的进程（线程）。函数原型：

**[cpp]** [view plain](http://blog.csdn.net/shallnet/article/details/42109093) [copy](http://blog.csdn.net/shallnet/article/details/42109093)

1. #include <semaphore.h>
2. **int** sem\_post(sem\_t \*sem);
3. Link with -pthread.

        6. 函数sem\_getvalue获取指定信号量的当前值，如果当前信号量已上锁，返回0，或者为负数，其绝对值为等待该信号量的解锁的线程数。

**[cpp]** [view plain](http://blog.csdn.net/shallnet/article/details/42109093) [copy](http://blog.csdn.net/shallnet/article/details/42109093)

1. #include <semaphore.h>
2. **int** sem\_getvalue(sem\_t \*sem, **int** \*sval);
3. Link with -pthread.

        7. 无名信号量使用sem\_init()初始化，函数原型为：

**[cpp]** [view plain](http://blog.csdn.net/shallnet/article/details/42109093) [copy](http://blog.csdn.net/shallnet/article/details/42109093)

1. #include <semaphore.h>
2. **int** sem\_init(sem\_t \*sem, **int** pshared, unsigned **int** value);
3. Link with -pthread.

    sem为无名信号量地址，value为信号量初始值，pshared指定信号量是在进程的各个线程之间共享还是在进程之间共享，如果该值为0，表示在进程的线程之间共享，如果非0则在进出之间共享，无名信号量主要用在有共同祖先的进程或线程之间。

          这一节使用posix共享内存来实现各个进程之间的通信，并使用posix信号量来达到互斥与同步的目的。首先我们来看看使用信号量实现对共享内存段的互斥访问。

          之前线程的访问使用如下方式来达到对公共的内存互斥访问：

**[cpp]** [view plain](http://blog.csdn.net/shallnet/article/details/42109093) [copy](http://blog.csdn.net/shallnet/article/details/42109093)

1. pthread\_mutex\_t     mutex = PTHREAD\_MUTEX\_INITIALIZER;
2. ......
3. pthread\_mutex\_lock(&mutex);//加锁
4. ......
5. /\*share memory handle\*/
6. ......
7. pthread\_mutex\_unlock(&mutex);//解锁
8. ......

现在我们也使用类似方式来实现：

**[cpp]** [view plain](http://blog.csdn.net/shallnet/article/details/42109093) [copy](http://blog.csdn.net/shallnet/article/details/42109093)

1. sem\_t \*sem\_mutex = NULL;
2. ......
3. SLN\_MUTEX\_SHM\_LOCK(SEM\_MUTEX\_FILE, sem\_mutex);//加锁
4. ......
5. /\*share memory handle\*/
6. ......
7. SLN\_MUTEX\_SHM\_UNLOCK(sem\_mutex);//解锁
8. ......

其中SEM\_MUTEX\_FILE为sem\_open函数需要的有名信号量名称。

其中两个加锁解锁的实现为：

**[cpp]** [view plain](http://blog.csdn.net/shallnet/article/details/42109093) [copy](http://blog.csdn.net/shallnet/article/details/42109093)

1. #define SLN\_MUTEX\_SHM\_LOCK(shmfile, sem\_mutex) do {\
2. sem\_mutex = sem\_open(shmfile, O\_RDWR | O\_CREAT, 0666, 1);\
3. **if** (SEM\_FAILED == sem\_mutex) {\
4. printf("sem\_open(%d): %s\n", \_\_LINE\_\_, strerror(errno));\
5. }\
6. sem\_wait(sem\_mutex);\
7. }**while**(0)
9. #define SLN\_MUTEX\_SHM\_UNLOCK(sem\_mutex) do {sem\_post(sem\_mutex);} while(0)

其实就是初始化一个二值信号量，其初始值为1，并执行wait操作，使信号量的值变为0，此时其它进程想要操作共享内存时也需要执行wait操作，但此时信号量的值为0，所以开始等待信号量的值变为1。当当前进程操作完共享内存后，开始解锁，执行post操作将信号量的值加一，此时其它进程的wait可以返回了。

下面为一个互斥访问共享内存的示例，posix共享内存实现请查看前面IPC的系列文章。

ser process：

**[cpp]** [view plain](http://blog.csdn.net/shallnet/article/details/42109093) [copy](http://blog.csdn.net/shallnet/article/details/42109093)

1. **int** nms\_shm\_get(**char** \*shm\_file, **void** \*\*shm, **int** mem\_len)
2. {
3. **int** fd;
5. fd = shm\_open(shm\_file, O\_RDWR | O\_CREAT, 0666);
6. **if** (fd < 0) {
7. printf("shm\_pen <%s> failed: %s\n", shm\_file, strerror(errno));
8. **return** -1;
9. }
11. ftruncate(fd, mem\_len);
13. \*shm = mmap(NULL, mem\_len, PROT\_READ | PROT\_WRITE, MAP\_SHARED, fd, 0);
14. **if** (MAP\_FAILED == \*shm) {
15. printf("mmap: %s\n", strerror(errno));
16. **return** -1;
17. }
19. **return** 0;
20. }
21. **int** main(**int** argc, **const** **char** \*argv[])
22. {
23. sem\_t \*sem\_mutex = NULL;
24. **char** \*str = NULL;
26. SLN\_MUTEX\_SHM\_LOCK(SEM\_MUTEX\_FILE, sem\_mutex); //加锁
28. nms\_shm\_get(SHM\_FILE, (**void** \*\*)&str, SHM\_MAX\_LEN); //下面三行互斥访问共享内存
29. sleep(6);
30. snprintf(str, SHM\_MAX\_LEN, "posix semphore server!");
32. SLN\_MUTEX\_SHM\_UNLOCK(sem\_mutex); //解锁
34. sleep(6);
36. shm\_unlink(SHM\_FILE);
38. **return** 0;
39. }

client process:

**[cpp]** [view plain](http://blog.csdn.net/shallnet/article/details/42109093) [copy](http://blog.csdn.net/shallnet/article/details/42109093)

1. **int** main(**int** argc, **const** **char** \*argv[])
2. {
3. sem\_t \*sem\_mutex;
4. **char** \*str = NULL;
5. SLN\_MUTEX\_SHM\_LOCK(SEM\_MUTEX\_FILE, sem\_mutex);
6. nms\_shm\_get(SHM\_FILE, (**void** \*\*)&str, SHM\_MAX\_LEN);
7. printf("client get: %s\n", str);
8. SLN\_MUTEX\_SHM\_UNLOCK(sem\_mutex);
9. **return** 0;
10. }

先启动服务进程首先加锁，创建共享内存并操作它，加锁中sleep 6秒，以便测试客户进程是否在服务进程未释放锁时处于等待状态。客户进程在服务进程启动之后马上启动，此时处于等待状态，当服务进程6秒之后解锁，客户进程获得共享内存信息。再过6秒之后，服务进程删除共享内存，客户进程再此获取共享内存失败。

**[cpp]** [view plain](http://blog.csdn.net/shallnet/article/details/42109093) [copy](http://blog.csdn.net/shallnet/article/details/42109093)

1. # ./server &
2. [1] 21690
3. # ./client
4. client get: posix semphore server!
5. # ./client
6. shm\_open <share\_memory\_file> failed: No such file or directory
7. client get: (null)
8. [1]+ Done ./server

posix有名信号量创建的信号量文件和共享内存文件在/dev/shm/目录下：

**[cpp]** [view plain](http://blog.csdn.net/shallnet/article/details/42109093) [copy](http://blog.csdn.net/shallnet/article/details/42109093)

1. # ls /dev/shm/
2. sem.sem\_mutex share\_memory\_file
3. #

在两个进程共享数据时，当一个进程向共享内存写入了数据后需要通知另外的进程，这就需要两个进程之间实现同步，这里我们给上面的程序在互斥的基础上加上同步操作。同步也是使用posix信号量来实现。

server process：

**[cpp]** [view plain](http://blog.csdn.net/shallnet/article/details/42109093) [copy](http://blog.csdn.net/shallnet/article/details/42109093)

1. **int** main(**int** argc, **const** **char** \*argv[])
2. {
3. sem\_t \*sem\_mutex = NULL;
4. sem\_t \*sem\_consumer = NULL, \*sem\_productor = NULL;
5. **int** semval;
6. **char** \*sharememory = NULL;
8. sem\_consumer = sem\_open(SEM\_CONSUMER\_FILE, O\_CREAT, 0666, 0); //初始化信号量sem\_consumer ，并设置初始值为0
9. **if** (SEM\_FAILED == sem\_consumer) {
10. printf("sem\_open <%s>: %s\n", SEM\_CONSUMER\_FILE, strerror(errno));
11. **return** -1;
12. }
14. sem\_productor = sem\_open(SEM\_PRODUCTOR\_FILE, O\_CREAT, 0666, 0);//初始化信号量sem\_productor ，并设置初始值为0
15. **if** (SEM\_FAILED == sem\_productor) {
16. printf("sem\_open <%s>: %s\n", SEM\_PRODUCTOR\_FILE, strerror(errno));
17. **return** -1;
18. }
20. **for** (;;) {//服务进程一直循环处理客户进程请求
21. sem\_getvalue(sem\_consumer, &semval);
22. printf("%d waiting...\n", semval);
23. **if** (sem\_wait(sem\_consumer) < 0) {//如果sem\_consumer为0，则阻塞在此，等待客户进程post操作使sem\_consumer大于0，此处和客户进程同步
24. printf("sem\_wait: %s\n", strerror(errno));
25. **return** -1;
26. }
27. printf("Get request...\n");
29. SLN\_MUTEX\_SHM\_LOCK(SEM\_MUTEX, sem\_mutex);//此处开始互斥访问共享内存
30. nms\_shm\_get(SHM\_FILE, (**void** \*\*)&sharememory, SHM\_MAX\_LEN);
31. sleep(6);
32. snprintf(sharememory, SHM\_MAX\_LEN, "Hello, this is server's message!");
33. SLN\_MUTEX\_SHM\_UNLOCK(sem\_mutex);
35. sem\_post(sem\_productor);//使信号量sem\_productor加一，使阻塞的客户进程继续执行
36. printf("Response request...\n");
37. }
39. sem\_close(sem\_consumer);
40. sem\_close(sem\_productor);
41. **return** 0;
42. }

client process：

**[cpp]** [view plain](http://blog.csdn.net/shallnet/article/details/42109093) [copy](http://blog.csdn.net/shallnet/article/details/42109093)

1. **int** main(**int** argc, **const** **char** \*argv[])
2. {
3. sem\_t \*sem\_consumer = NULL, \*sem\_productor = NULL;
4. **struct** timespec timeout;
5. **int** ret;
6. **char** \*sharememory = NULL;
7. sem\_t \*sem\_mutex;
8. sem\_consumer = sem\_open(SEM\_CONSUMER\_FILE, O\_RDWR);//获取信号量sem\_consumer的值
9. **if** (SEM\_FAILED == sem\_consumer) {
10. printf("sem\_open <%s>: %s\n", SEM\_CONSUMER\_FILE, strerror(errno));
11. **return** -1;
12. }
13. sem\_productor = sem\_open(SEM\_PRODUCTOR\_FILE, O\_RDWR);//获取信号量sem\_productor 的值
14. **if** (SEM\_FAILED == sem\_productor) {
15. printf("sem\_open <%s>: %s\n", SEM\_PRODUCTOR\_FILE, strerror(errno));
16. **return** -1;
17. }
18. //clear\_exist\_sem(sem\_productor);
20. SLN\_MUTEX\_SHM\_LOCK(SEM\_MUTEX, sem\_mutex);//互斥访问共享内存
21. nms\_shm\_get(SHM\_FILE, (**void** \*\*)&sharememory, SHM\_MAX\_LEN);
22. printf("sharememory: %s\n", sharememory);
23. SLN\_MUTEX\_SHM\_UNLOCK(sem\_mutex);
25. sem\_post(sem\_consumer);//信号量sem\_consumer加一，唤醒是阻塞在该信号量上的服务进程
26. printf("Post...\n");
27. sem\_wait(sem\_productor);//等待服务进程回应
28. /\*
29. timeout.tv\_sec = time(NULL) + SEM\_TIMEOUT\_SEC;
30. timeout.tv\_nsec = 0;
31. ret = sem\_timedwait(sem\_productor, &timeout);
32. if (ret < 0) {
33. printf("sem\_timedwait: %s\n", strerror(errno));
34. }
35. \*/
37. printf("Get response...\n");
38. sem\_close(sem\_consumer);
39. sem\_close(sem\_productor);
41. **return** 0;
42. }

本节示例源码下载：

[点击打开链接](http://download.csdn.net/detail/gentleliu/8292189)