[Linux进程间通信——使用信号量](http://blog.csdn.net/ljianhui/article/details/10243617)

一、什么是信号量

为了防止出现因多个程序同时访问一个共享资源而引发的一系列问题，我们需要一种方法，它可以通过生成并使用令牌来授权，在任一时刻只能有一个执行线程访问代码的临界区域。临界区域是指执行数据更新的代码需要独占式地执行。而信号量就可以提供这样的一种访问机制，让一个临界区同一时间只有一个线程在访问它，也就是说信号量是用来调协进程对共享资源的访问的。

信号量是一个特殊的变量，程序对其访问都是原子操作，且只允许对它进行等待（即P(信号变量))和发送（即V(信号变量))信息操作。最简单的信号量是只能取0和1的变量，这也是信号量最常见的一种形式，叫做二进制信号量。而可以取多个正整数的信号量被称为通用信号量。这里主要讨论二进制信号量。

二、信号量的工作原理

由于信号量只能进行两种操作等待和发送信号，即P(sv)和V(sv),他们的行为是这样的：

P(sv)：如果sv的值大于零，就给它减1；如果它的值为零，就挂起该进程的执行

V(sv)：如果有其他进程因等待sv而被挂起，就让它恢复运行，如果没有进程因等待sv而挂起，就给它加1.

举个例子，就是两个进程共享信号量sv，一旦其中一个进程执行了P(sv)操作，它将得到信号量，并可以进入临界区，使sv减1。而第二个进程将被阻止进入临界区，因为当它试图执行P(sv)时，sv为0，它会被挂起以等待第一个进程离开临界区域并执行V(sv)释放信号量，这时第二个进程就可以恢复执行。

三、Linux的信号量机制

Linux提供了一组精心设计的信号量接口来对信号进行操作，它们不只是针对二进制信号量，下面将会对这些函数进行介绍，但请注意，这些函数都是用来对成组的信号量值进行操作的。它们声明在头文件sys/sem.h中。

1、semget函数

它的作用是创建一个新信号量或取得一个已有信号量，原型为：

**[cpp]** [view plain](http://blog.csdn.net/ljianhui/article/details/10243617) [copy](http://blog.csdn.net/ljianhui/article/details/10243617)

[print?](http://blog.csdn.net/ljianhui/article/details/10243617)

1. **int** semget(key\_t key, **int** num\_sems, **int** sem\_flags);

第一个参数key是整数值（唯一非零），不相关的进程可以通过它访问一个信号量，它代表程序可能要使用的某个资源，程序对所有信号量的访问都是间接的，程序先通过调用semget函数并提供一个键，再由系统生成一个相应的信号标识符（semget函数的返回值），只有semget函数才直接使用信号量键，所有其他的信号量函数使用由semget函数返回的信号量标识符。如果多个程序使用相同的key值，key将负责协调工作。

第二个参数num\_sems指定需要的信号量数目，它的值几乎总是1。

第三个参数sem\_flags是一组标志，当想要当信号量不存在时创建一个新的信号量，可以和值IPC\_CREAT做按位或操作。设置了IPC\_CREAT标志后，即使给出的键是一个已有信号量的键，也不会产生错误。而IPC\_CREAT | IPC\_EXCL则可以创建一个新的，唯一的信号量，如果信号量已存在，返回一个错误。

semget函数成功返回一个相应信号标识符（非零），失败返回-1.

2、semop函数

它的作用是改变信号量的值，原型为：

**[cpp]** [view plain](http://blog.csdn.net/ljianhui/article/details/10243617) [copy](http://blog.csdn.net/ljianhui/article/details/10243617)

[print?](http://blog.csdn.net/ljianhui/article/details/10243617)

1. **int** semop(**int** sem\_id, **struct** sembuf \*sem\_opa, **size\_t** num\_sem\_ops);

sem\_id是由semget返回的信号量标识符，sembuf结构的定义如下：

**[cpp]** [view plain](http://blog.csdn.net/ljianhui/article/details/10243617) [copy](http://blog.csdn.net/ljianhui/article/details/10243617)

[print?](http://blog.csdn.net/ljianhui/article/details/10243617)

1. **struct** sembuf{
2. **short** sem\_num;//除非使用一组信号量，否则它为0
3. **short** sem\_op;//信号量在一次操作中需要改变的数据，通常是两个数，一个是-1，即P（等待）操作，
4. //一个是+1，即V（发送信号）操作。
5. **short** sem\_flg;//通常为SEM\_UNDO,使操作系统跟踪信号，
6. //并在进程没有释放该信号量而终止时，操作系统释放信号量
7. };

3、semctl函数

该函数用来直接控制信号量信息，它的原型为：

**[cpp]** [view plain](http://blog.csdn.net/ljianhui/article/details/10243617) [copy](http://blog.csdn.net/ljianhui/article/details/10243617)

[print?](http://blog.csdn.net/ljianhui/article/details/10243617)

1. **int** semctl(**int** sem\_id, **int** sem\_num, **int** command, ...);

如果有第四个参数，它通常是一个union semum结构，定义如下：

**[cpp]** [view plain](http://blog.csdn.net/ljianhui/article/details/10243617) [copy](http://blog.csdn.net/ljianhui/article/details/10243617)

[print?](http://blog.csdn.net/ljianhui/article/details/10243617)

1. **union** semun{
2. **int** val;
3. **struct** semid\_ds \*buf;
4. unsigned **short** \*arry;
5. };

前两个参数与前面一个函数中的一样，command通常是下面两个值中的其中一个

SETVAL：用来把信号量初始化为一个已知的值。p 这个值通过union semun中的val成员设置，其作用是在信号量第一次使用前对它进行设置。

IPC\_RMID：用于删除一个已经无需继续使用的信号量标识符。

四、进程使用信号量通信

下面使用一个例子来说明进程间如何使用信号量来进行通信，这个例子是两个相同的程序同时向屏幕输出数据，我们可以看到如何使用信号量来使两个进程协调工作，使同一时间只有一个进程可以向屏幕输出数据。注意，如果程序是第一次被调用（为了区分，第一次调用程序时带一个要输出到屏幕中的字符作为一个参数），则需要调用set\_semvalue函数初始化信号并将message字符设置为传递给程序的参数的第一个字符，同时第一个启动的进程还负责信号量的删除工作。如果不删除信号量，它将继续在系统中存在，即使程序已经退出，它可能在你下次运行此程序时引发问题，而且信号量是一种有限的资源。

在main函数中调用semget来创建一个信号量，该函数将返回一个信号量标识符，保存于全局变量sem\_id中，然后以后的函数就使用这个标识符来访问信号量。

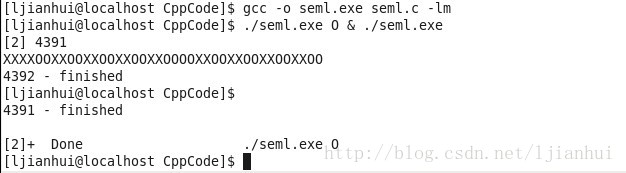
源文件为seml.c，代码如下：

**[cpp]** [view plain](http://blog.csdn.net/ljianhui/article/details/10243617) [copy](http://blog.csdn.net/ljianhui/article/details/10243617)

[print?](http://blog.csdn.net/ljianhui/article/details/10243617)

1. #include <unistd.h>
2. #include <sys/types.h>
3. #include <sys/stat.h>
4. #include <fcntl.h>
5. #include <stdlib.h>
6. #include <stdio.h>
7. #include <string.h>
8. #include <sys/sem.h>
10. **union** semun
11. {
12. **int** val;
13. **struct** semid\_ds \*buf;
14. unsigned **short** \*arry;
15. };
17. **static** **int** sem\_id = 0;
19. **static** **int** set\_semvalue();
20. **static** **void** del\_semvalue();
21. **static** **int** semaphore\_p();
22. **static** **int** semaphore\_v();
24. **int** main(**int** argc, **char** \*argv[])
25. {
26. **char** message = 'X';
27. **int** i = 0;
29. //创建信号量
30. sem\_id = semget((key\_t)1234, 1, 0666 | IPC\_CREAT);
32. **if**(argc > 1)
33. {
34. //程序第一次被调用，初始化信号量
35. **if**(!set\_semvalue())
36. {
37. fprintf(stderr, "Failed to initialize semaphore\n");
38. exit(EXIT\_FAILURE);
39. }
40. //设置要输出到屏幕中的信息，即其参数的第一个字符
41. message = argv[1][0];
42. sleep(2);
43. }
44. **for**(i = 0; i < 10; ++i)
45. {
46. //进入临界区
47. **if**(!semaphore\_p())
48. exit(EXIT\_FAILURE);
49. //向屏幕中输出数据
50. printf("%c", message);
51. //清理缓冲区，然后休眠随机时间
52. fflush(stdout);
53. sleep(rand() % 3);
54. //离开临界区前再一次向屏幕输出数据
55. printf("%c", message);
56. fflush(stdout);
57. //离开临界区，休眠随机时间后继续循环
58. **if**(!semaphore\_v())
59. exit(EXIT\_FAILURE);
60. sleep(rand() % 2);
61. }
63. sleep(10);
64. printf("\n%d - finished\n", getpid());
66. **if**(argc > 1)
67. {
68. //如果程序是第一次被调用，则在退出前删除信号量
69. sleep(3);
70. del\_semvalue();
71. }
72. exit(EXIT\_SUCCESS);
73. }
75. **static** **int** set\_semvalue()
76. {
77. //用于初始化信号量，在使用信号量前必须这样做
78. **union** semun sem\_union;
80. sem\_union.val = 1;
81. **if**(semctl(sem\_id, 0, SETVAL, sem\_union) == -1)
82. **return** 0;
83. **return** 1;
84. }
86. **static** **void** del\_semvalue()
87. {
88. //删除信号量
89. **union** semun sem\_union;
91. **if**(semctl(sem\_id, 0, IPC\_RMID, sem\_union) == -1)
92. fprintf(stderr, "Failed to delete semaphore\n");
93. }
95. **static** **int** semaphore\_p()
96. {
97. //对信号量做减1操作，即等待P（sv）
98. **struct** sembuf sem\_b;
99. sem\_b.sem\_num = 0;
100. sem\_b.sem\_op = -1;//P()
101. sem\_b.sem\_flg = SEM\_UNDO;
102. **if**(semop(sem\_id, &sem\_b, 1) == -1)
103. {
104. fprintf(stderr, "semaphore\_p failed\n");
105. **return** 0;
106. }
107. **return** 1;
108. }
110. **static** **int** semaphore\_v()
111. {
112. //这是一个释放操作，它使信号量变为可用，即发送信号V（sv）
113. **struct** sembuf sem\_b;
114. sem\_b.sem\_num = 0;
115. sem\_b.sem\_op = 1;//V()
116. sem\_b.sem\_flg = SEM\_UNDO;
117. **if**(semop(sem\_id, &sem\_b, 1) == -1)
118. {
119. fprintf(stderr, "semaphore\_v failed\n");
120. **return** 0;
121. }
122. **return** 1;
123. }

运行结果如下：



注：这个程序的临界区为main函数for循环不的semaphore\_p和semaphore\_v函数中间的代码。

例子分析 ：同时运行一个程序的两个实例，注意第一次运行时，要加上一个字符作为参数，例如本例中的字符‘O’，它用于区分是否为第一次调用，同时这个字符输出到屏幕中。因为每个程序都在其进入临界区后和离开临界区前打印一个字符，所以每个字符都应该成对出现，正如你看到的上图的输出那样。在main函数中循环中我们可以看到，每次进程要访问stdout（标准输出），即要输出字符时，每次都要检查信号量是否可用（即stdout有没有正在被其他进程使用）。所以，当一个进程A在调用函数semaphore\_p进入了临界区，输出字符后，调用sleep时，另一个进程B可能想访问stdout，但是信号量的P请求操作失败，只能挂起自己的执行，当进程A调用函数semaphore\_v离开了临界区，进程B马上被恢复执行。然后进程A和进程B就这样一直循环了10次。

五、对比例子——进程间的资源竞争

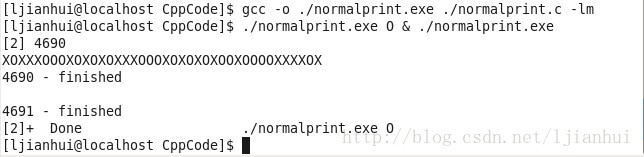
看了上面的例子，你可能还不是很明白，不过没关系，下面我就以另一个例子来说明一下，它实现的功能与前面的例子一样，运行方式也一样，都是两个相同的进程，同时向stdout中输出字符，只是没有使用信号量，两个进程在互相竞争stdout。它的代码非常简单，文件名为normalprint.c，代码如下：

**[cpp]** [view plain](http://blog.csdn.net/ljianhui/article/details/10243617) [copy](http://blog.csdn.net/ljianhui/article/details/10243617)

[print?](http://blog.csdn.net/ljianhui/article/details/10243617)

1. #include <stdio.h>
2. #include <stdlib.h>
4. **int** main(**int** argc, **char** \*argv[])
5. {
6. **char** message = 'X';
7. **int** i = 0;
8. **if**(argc > 1)
9. message = argv[1][0];
10. **for**(i = 0; i < 10; ++i)
11. {
12. printf("%c", message);
13. fflush(stdout);
14. sleep(rand() % 3);
15. printf("%c", message);
16. fflush(stdout);
17. sleep(rand() % 2);
18. }
19. sleep(10);
20. printf("\n%d - finished\n", getpid());
21. exit(EXIT\_SUCCESS);
22. }

运行结果如下：



例子分析：

从上面的输出结果，我们可以看到字符‘X’和‘O’并不像前面的例子那样，总是成对出现，因为当第一个进程A输出了字符后，调用sleep休眠时，另一个进程B立即输出并休眠，而进程A醒来时，再继续执行输出，同样的进程B也是如此。所以输出的字符就是不成对的出现。这两个进程在竞争stdout这一共同的资源。通过两个例子的对比，我想信号量的意义和使用应该比较清楚了。

六、信号量的总结

信号量是一个特殊的变量，程序对其访问都是原子操作，且只允许对它进行等待（即P(信号变量))和发送（即V(信号变量))信息操作。我们通常通过信号来解决多个进程对同一资源的访问竞争的问题，使在任一时刻只能有一个执行线程访问代码的临界区域，也可以说它是协调进程间的对同一资源的访问权，也就是用于同步进程的。