

36 | 进程间通信：遇到大项目需要项目组之间的合作才行

2019-06-19 刘超

趣谈Linux操作系统

[进入课程 >](#)



讲述：刘超

时长 17:15 大小 15.81M

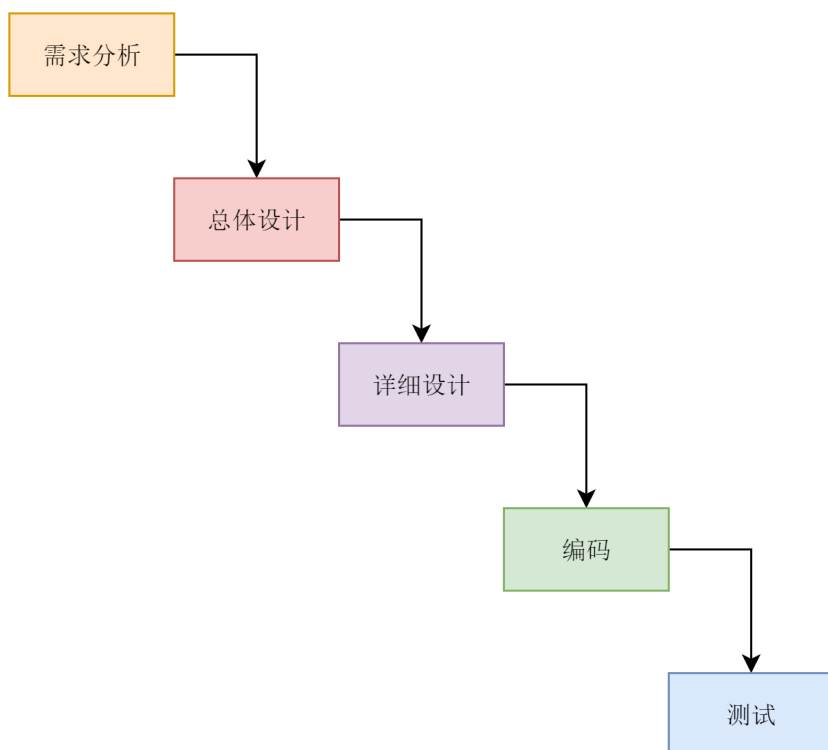


前面咱们接项目的时候，主要强调项目之间的隔离性。这是因为，我们刚开始接的都是小项目。随着我们接的项目越来越多，就不免遇到大项目，这就需要多个项目组进行合作才能完成。


两个项目组应该通过什么样的方式进行沟通与合作呢？作为老板，你应该如何设计整个流程呢？

管道模型

好在有这么多成熟的项目管理流程可以参考。最最传统的模型就是软件开发的**瀑布模型**（Waterfall Model）。所谓的瀑布模型，其实就是将整个软件开发过程分成多个阶段，往往是上一个阶段完全做完，才将输出结果交给下一个阶段。就像下面这张图展示的一样。



这种模型类似进程间通信的**管道模型**。还记得咱们最初学 Linux 命令的时候，有下面这样一行命令：


 复制代码

```
1 ps -ef | grep 关键字 | awk '{print $2}' | xargs kill -9
```

这里面的竖线 “|” 就是一个管道。它会将前一个命令的输出，作为后一个命令的输入。从管道的这个名称可以看出来，管道是一种单向传输数据的机制，它其实是一段缓存，里面的数据只能从一端写入，从另一端读出。如果想互相通信，我们需要创建两个管道才行。


管道分为两种类型，“|” 表示的管道称为**匿名管道**，意思就是这个类型的管道没有名字，用完了就销毁了。就像上面那个命令里面的一样，竖线代表的管道随着命令的执行自动创建、自动销毁。用户甚至都不知道自己在用管道这种技术，就已经解决了问题。所以这也是面试题里面经常会问的，到时候千万别说这是竖线，而要回答背后的机制，管道。

另外一种类型是**命名管道**。这个类型的管道需要通过 `mkfifo` 命令显式地创建。

 复制代码


```
1 mkfifo hello
```

hello 就是这个管道的名称。管道以文件的形式存在，这也符合 Linux 里面一切皆文件的原则。这个时候，我们 ls 一下，可以看到，这个文件的类型是 p，就是 pipe 的意思。

 复制代码

```
1 # ls -l
2 prw-r--r--  1 root root          0 May 21 23:29 hello
```

接下来，我们可以往管道里面写入东西。例如，写入一个字符串。


 复制代码

```
1 # echo "hello world" > hello
```

这个时候，管道里面的内容没有被读出，这个命令就是停在这里的，这说明当一个项目组要把它的输出交接给另一个项

目组做输入，当没有交接完毕的时候，前一个项目组是不能撒手不管的。

这个时候，我们就需要重新连接一个终端。在终端中，用下面的命令读取管道里面的内容：

 复制代码

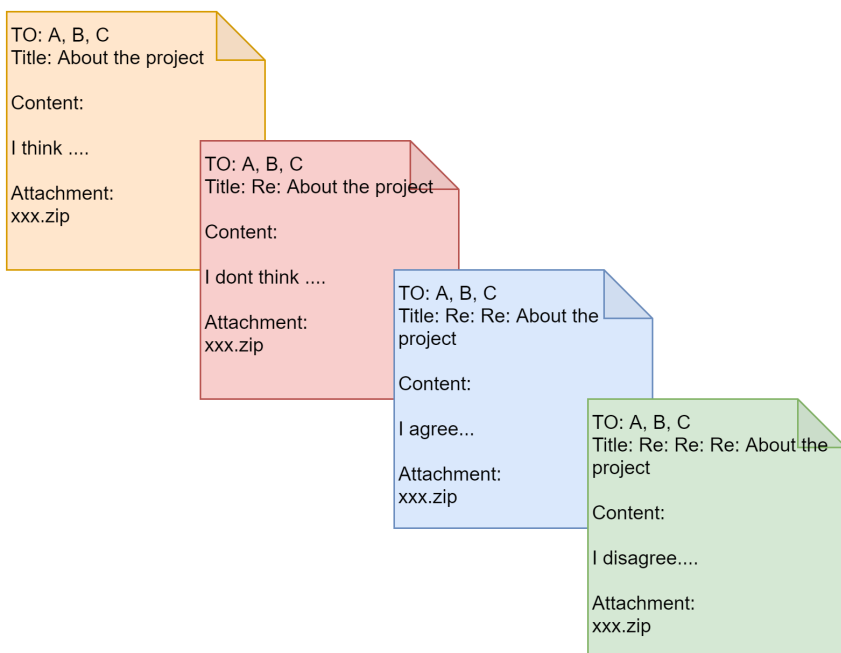
```
1 # cat < hello
2 hello world
```

一方面，我们能够看到，管道里面的内容被读取出来，打印到了终端上；另一方面，echo 那个命令正常退出了，也即交接完毕，前一个项目组就完成了使命，可以解散了。

我们可以看出，瀑布模型的开发流程效率比较低，因为团队之间无法频繁地沟通。而且，管道的使用模式，也不适合进程间频繁的交流数据。

于是，我们还得想其他的办法，例如我们是不是可以借鉴传统外企的沟通方式——邮件。邮件有一定的格式，例如抬头，正文，附件等，发送邮件可以建立收件人列表，所有在这个列表中的人，都可以反复的在此邮件基础上回复，达到频繁沟通的目的。

消息队列模型



这种模型类似进程间通信的消息队列模型。和管道将信息一股脑儿地从一个进程，倒给另一个进程不同，消息队列有点儿像邮件，发送数据时，会分成一个一个独立的数据单元，也就是消息体，每个消息体都是固定大小的存储块，在字节流上不连续。

这个消息结构的定义我写在下面了。这里面的类型 `type` 和正文 `text` 没有强制规定，只要消息的发送方和接收方约定好即可。

```
1 struct msg_buffer {
2     long mtype;
3     char mtext[1024];
4 };
```

接下来，我们需要创建一个消息队列，使用**msgget 函数**。这个函数需要有一个参数 **key**，这是消息队列的唯一标识，应该是唯一的。如何保持唯一性呢？这个还是和文件关联。


我们可以指定一个文件，**ftok** 会根据这个文件的 **inode**，生成一个近乎唯一的 **key**。只要在这个消息队列的生命周期内，这个文件不要被删除就可以了。只要不删除，无论什么时刻，再调用 **ftok**，也会得到同样的 **key**。这种 **key** 的使用方式在这一章会经常遇到，这是因为它们都属于 **System V IPC 进程间通信机制体系**中。

```
1 #include <stdio.h>
2 #include <stdlib.h>
3 #include <sys/msg.h>
4
5
6 int main() {
7     int messagequeueid;
8     key_t key;
```

```
9
10
11  if((key = ftok("/root/messagequeue/messagequeuekey",
12  {
13      perror("ftok error");
14      exit(1);
15  }
16
17
18  printf("Message Queue key: %d.\n", key);
19
20
21  if ((messagequeueid = msgget(key, IPC_CREAT|0777)) ==
22  {
23      perror("msgget error");
24      exit(1);
25  }
26
27
28  printf("Message queue id: %d.\n", messagequeueid);
29 }
```




在运行上面这个程序之前，我们先使用命令 `touch messagequeuekey`，创建一个文件，然后多次执行的结果就会像下面这样：

 复制代码

```
1 # ./a.out
2 Message Queue key: 92536.
3 Message queue id: 32768.
```


System V IPC 体系有一个统一的命令行工具：ipcmk, ipcs 和 ipcrm 用于创建、查看和删除 IPC 对象。

例如，ipcs -q 就能看到上面我们创建的消息队列对象。


 复制代码

```
1 # ipcs -q
2
3
4 ----- Message Queues -----
5 key          msqid      owner          perms          used-bytes
6 0x00016978 32768      root          777            0
```

接下来，我们来看如何发送信息。发送消息主要调用 **msgsnd 函数**。第一个参数是 message queue 的 id，第二个参数是消息的结构体，第三个参数是消息的长度，最后一个参数是 flag。这里 IPC_NOWAIT 表示发送的时候不阻塞，直接返回。

下面的这段程序，getopt_long、do-while 循环以及 switch，是用来解析命令行参数的。命令行参数的格式定义在 long_options 里面。每一项的第一个成


员 “id” “type ” “message” 是参数选项的全称，第二个成员都为 1，表示参数选项后面要跟参数，最后一个成员 ‘i’ ‘t’ ‘m’ 是参数选项的简称。

 复制代码

```
1 #include <stdio.h>
2 #include <stdlib.h>
3 #include <sys/msg.h>
4 #include <getopt.h>
5 #include <string.h>
6
7
8 struct msg_buffer {
9     long mtype;
10    char mtext[1024];
11 };
12
13
14 int main(int argc, char *argv[]) {
15     int next_option;
16     const char* const short_options = "i:t:m:";
17     const struct option long_options[] = {
18         { "id", 1, NULL, 'i'},
19         { "type", 1, NULL, 't'},
20         { "message", 1, NULL, 'm'},
21         { NULL, 0, NULL, 0 }
22     };
23
24     int messagequeueid = -1;
25     struct msg_buffer buffer;
26     buffer.mtype = -1;
27     int len = -1;
28     char * message = NULL;
```


```
29  do {
30      next_option = getopt_long (argc, argv, short_option
31      switch (next_option)
32      {
33          case 'i':
34              messagequeueid = atoi(optarg);
35              break;
36          case 't':
37              buffer.mtype = atol(optarg);
38              break;
39          case 'm':
40              message = optarg;
41              len = strlen(message) + 1;
42              if (len > 1024) {
43                  perror("message too long.");
44                  exit(1);
45              }
46              memcpy(buffer.mtext, message, len);
47              break;
48          default:
49              break;
50      }
51  }while(next_option != -1);
52
53
54  if(messagequeueid != -1 && buffer.mtype != -1 && len
55      if(msgsnd(messagequeueid, &buffer, len, IPC_NOWAIT)
56          perror("fail to send message.");
57          exit(1);
58      }
59  } else {
60      perror("arguments error");
61  }
62
63  return 0;
```

接下来，我们可以编译并运行这个发送程序。

 复制代码

```
1 gcc -o send sendmessage.c
2 ./send -i 32768 -t 123 -m "hello world"
```

接下来，我们再来看如何收消息。收消息主要调用**msgrcv**函数，第一个参数是 message queue 的 id，第二个参数是消息的结构体，第三个参数是可接受的最大长度，第四个参数是消息类型，最后一个参数是 flag，这里 IPC_NOWAIT 表示接收的时候不阻塞，直接返回。


 复制代码

```
1 #include <stdio.h>
2 #include <stdlib.h>
3 #include <sys/msg.h>
4 #include <getopt.h>
5 #include <string.h>
6
7
8 struct msg_buffer {
9     long mtype;
10    char mtext[1024];
```

```
11 };
12
13
14 int main(int argc, char *argv[]) {
15     int next_option;
16     const char* const short_options = "i:t:";
17     const struct option long_options[] = {
18         { "id", 1, NULL, 'i'},
19         { "type", 1, NULL, 't'},
20         { NULL, 0, NULL, 0 }
21     };
22
23     int messagequeueid = -1;
24     struct msg_buffer buffer;
25     long type = -1;
26     do {
27         next_option = getopt_long (argc, argv, short_optior
28             switch (next_option)
29             {
30                 case 'i':
31                     messagequeueid = atoi(optarg);
32                     break;
33                 case 't':
34                     type = atol(optarg);
35                     break;
36                 default:
37                     break;
38             }
39     }while(next_option != -1);
40
41
42     if(messagequeueid != -1 && type != -1){
43         if(msgrcv(messagequeueid, &buffer, 1024, type, IPC_
44             perror("fail to recv message.");
45             exit(1);
```

```
46     }
47     printf("received message type : %d, text: %s.", buf
48 } else {
49     perror("arguments error");
50 }
51
52 return 0;
53 }
```

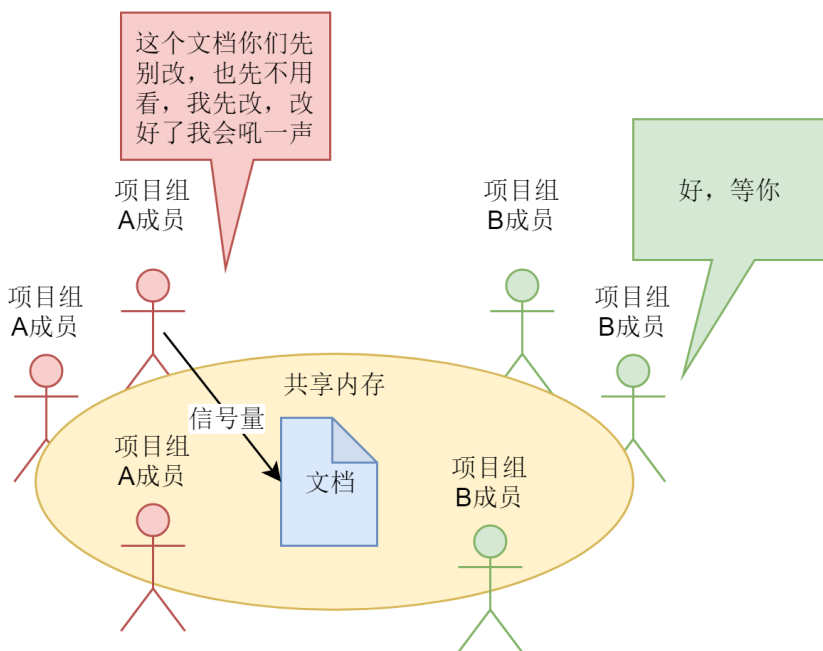
接下来，我们可以编译并运行这个发送程序。可以看到，如果有消息，可以正确地读到消息；如果没有，则返回没有消息。

 复制代码

```
1 # ./recv -i 32768 -t 123
2 received message type : 123, text: hello world.
3 # ./recv -i 32768 -t 123
4 fail to recv message.: No message of desired type
```

有了消息这种模型，两个进程之间的通信就像咱们平时发邮件一样，你来一封，我回一封，可以频繁沟通了。

共享内存模型



但是有时候，项目组之间的沟通需要特别紧密，而且要分享一些比较大的数据。如果使用邮件，就发现，一方面邮件的来去不及时；另外一方面，附件大小也有限制，所以，这个时候，我们经常采取的方式就是，把两个项目组在需要合作的期间，拉到一个会议室进行合作开发，这样大家可以直接交流文档呀，架构图呀，直接在白板上画或者直接扔给对方，就可以直接看到。


可以看出来，共享会议室这种模型，类似进程间通信的**共享内存模型**。前面咱们讲内存管理的时候，知道每个进程都有自己独立的虚拟内存空间，不同的进程的虚拟内存空间映射

到不同的物理内存中去。这个进程访问 A 地址和另一个进程访问 A 地址，其实访问的是不同的物理内存地址，对于数据的增删查改互不影响。

但是，咱们是不是可以变通一下，拿出一块虚拟地址空间来，映射到相同的物理内存中。这样这个进程写入的东西，另外一个进程马上就能看到了，都不需要拷贝来拷贝去，传来传去。

共享内存也是 System V IPC 进程间通信机制体系中的，所以从它使用流程可以看到熟悉的面孔。


我们可以创建一个共享内存，调用 `shmget`。在这个体系中，创建一个 IPC 对象都是 `xxxget`，这里面第一个参数是 `key`，和 `msgget` 里面的 `key` 一样，都是唯一定位一个共享内存对象，也可以通过关联文件的方式实现唯一性。第二个参数是共享内存的大小。第三个参数如果是 `IPC_CREAT`，同样表示创建一个新的。

 复制代码

```
1 int shmget(key_t key, size_t size, int flag);
```




创建完毕之后，我们可以通过 `ipcs` 命令查看这个共享内存。

 复制代码


```
1 #ipcs --shmems
2
3
4 ----- Shared Memory Segments -----
5 key          shmid    owner perms      bytes nattch status
6 0x00000000 19398656 marc  600      1048576 2        dest
```

接下来，如果一个进程想要访问这一段共享内存，需要将这个内存加载到自己的虚拟地址空间的某个位置，通过 `shmat` 函数，就是 `attach` 的意思。其中 `addr` 就是要指定 `attach` 到这个地方。但是这个地址的设定难度比较大，除非对于内存布局非常熟悉，否则可能会 `attach` 到一个非法地址。所以，通常的做法是将 `addr` 设为 `NULL`，让内核选一个合适的地址。返回值就是真正被 `attach` 的地方。

 复制代码

```
1 void *shmat(int shm_id, const void *addr, int flag);
```

如果共享内存使用完毕，可以通过 `shmdt` 解除绑定，然后通过 `shmctl`，将 `cmd` 设置为 `IPC_RMID`，从而删除这个共享内存对象。

 复制代码

```
1 int shmdt(void *addr);  
2  
3  
4 int shmctl(int shm_id, int cmd, struct shmid_ds *buf);
```

信号量

这里你是不是有一个疑问，如果两个进程 `attach` 同一个共享内存，大家都往里面写东西，很有可能就冲突了。例如两个进程都同时写一个地址，那先写的那个进程会发现内容被别人覆盖了。

所以，这里就需要一种保护机制，使得同一个共享的资源，同时只能被一个进程访问。在 System V IPC 进程间通信机制体系中，早就想好了应对办法，就是信号量

(Semaphore)。因此，信号量和共享内存往往要配合使用。


信号量其实是一个计数器，主要用于实现进程间的互斥与同步，而不是用于存储进程间通信数据。

我们可以将信号量初始化为一个数值，来代表某种资源的总体数量。对于信号量来讲，会定义两种原子操作，一个是**P 操作**，我们称为**申请资源操作**。这个操作会申请将信号量的数值减去 N ，表示这些数量被他申请使用了，其他人不能用了。另一个是**V 操作**，我们称为**归还资源操作**，这个操作会申请将信号量加上 M ，表示这些数量已经还给信号量了，其他人可以使用了。

例如，你有 100 元钱，就可以将信号量设置为 100。其中 A 向你借 80 元，就会调用 P 操作，申请减去 80。如果同时 B 向你借 50 元，但是 B 的 P 操作比 A 晚，那就没有办法，只好等待 A 归还钱的时候，B 的 P 操作才能成功。之后，A 调用 V 操作，申请加上 80 元，也就是还给你 80 元，这个时候信号量有 20 元了，这时候 B 的 P 操作才能成功，才能借走这 50 元。


所谓**原子操作** (Atomic Operation)，就是任何一块钱，都只能通过 P 操作借给一个人，不能同时借给两个人。也就是说，当 A 的 P 操作（借 80）和 B 的 P 操作（借 50），几乎同时到达的时候，不能因为大家都看到账户里有 100 就都成功，必须分个先来后到。

如果想创建一个信号量，我们可以通过 `semget` 函数。看，又是 `xxxget`，第一个参数 `key` 也是类似的，第二个参数 `num_sems` 不是指资源的数量，而是表示可以创建多少个信号量，形成一组信号量，也就是说，如果你有多种资源需要管理，可以创建一个信号量组。

 复制代码


```
1 int semget(key_t key, int num_sems, int sem_flags);
```

接下来，我们需要初始化信号量的总的资源数量。通过 `semctl` 函数，第一个参数 `semid` 是这个信号量组的 `id`，第二个参数 `semnum` 才是在这个信号量组中某个信号量的 `id`，第三个参数是命令，如果是初始化，则用 `SETVAL`，第四个参数是一个 `union`。如果初始化，应该用里面的 `val` 设置资源总量。

 复制代码

```
1 int semctl(int semid, int semnum, int cmd, union semun
2
3
4 union semun
5 {
6     int val;
7     struct semid_ds *buf;
8     unsigned short int *array;
```

```
9     struct seminfo *__buf;  
10 };
```



无论是 P 操作还是 V 操作，我们统一用 semop 函数。第一个参数还是信号量组的 id，一次可以操作多个信号量。第三个参数 numops 就是有多少个操作，第二个参数将这些操作放在一个数组中。

数组的每一项是一个 struct sembuf，里面的第一个成员是这个操作的对象是哪个信号量。

第二个成员就是要对这个信号量做多少改变。如果 sem_op < 0，就请求 sem_op 的绝对值的资源。如果相应的资源数可以满足请求，则将该信号量的值减去 sem_op 的绝对值，函数成功返回。

当相应的资源数不能满足请求时，就要看 sem_flg 了。如果把 sem_flg 设置为 IPC_NOWAIT，也就是没有资源也不等待，则 semop 函数出错返回 EAGAIN。如果 sem_flg 没有指定 IPC_NOWAIT，则进程挂起，直到当相应的资源数可以满足请求。若 sem_op > 0，表示进程归还相应的资源数，将 sem_op 的值加到信号量的值上。如果有进程正在休眠等待此信号量，则唤醒它们。

```
1 int semop(int semid, struct sembuf semoparray[], size_t
2
3
4 struct sembuf
5 {
6     short sem_num; // 信号量组中对应的序号, 0~sem_nums-1
7     short sem_op;  // 信号量值在一次操作中的改变量
8     short sem_flg; // IPC_NOWAIT, SEM_UNDO
9 }
```

信号量和共享内存都比较复杂，两者还要结合起来用，就更加复杂，它们内核的机制就更加复杂。这一节我们先不讲，放到本章的最后一节重点讲解。

信号

上面讲的进程间通信的方式，都是常规状态下的工作模式，对应到咱们平时的工作交接，收发邮件、联合开发等，其实还有一种异常情况下的工作模式。

例如出现线上系统故障，这个时候，什么流程都来不及了，不可能发邮件，也来不及开会，所有的架构师、开发、运维都要被通知紧急出动。所以，7 乘 24 小时不间断执行的系统都需要有告警系统，一旦出事情，就要通知到人，哪怕是半夜，也要电话叫起来，处理故障。

对应到操作系统中，就是信号。信号没有特别复杂的数据结构，就是用一个代号一样的数字。Linux 提供了几十种信号，分别代表不同的意义。信号之间依靠它们的值来区分。这就像咱们看警匪片，对于紧急的行动，都是说，“1 号作战任务”开始执行，警察就开始行动了。情况紧急，不能啰里啰嗦了。

信号可以在任何时候发送给某一进程，进程需要为这个信号配置信号处理函数。当某个信号发生的时候，就默认执行这个函数就可以了。这就相当于咱们运维一个系统应急手册，当遇到什么情况，做什么事情，都事先准备好，出了事情照着做就可以了。

总结时刻

这一节，我们整体讲解了一下进程间通信的各种模式。你现在还能记住多少？

类似瀑布开发模式的管道

类似邮件模式的消息队列

类似会议室联合开发的共享内存加信号量

类似应急预案的信号

当你自己使用的时候，可以根据不同的通信需要，选择不同的模式。

管道，请你记住这是命令行中常用的模式，面试问到的话，不要忘了。

消息队列其实很少使用，因为有太多的用户级别的消息队列，功能更强大。

共享内存加信号量是常用的模式。这个需要牢记，常见到一些知名的以 C 语言开发的开源软件都会用到它。

信号更加常用，机制也比较复杂。我们后面会有单独的一节来解析。

课堂练习

这节课的程序，请你务必自己编译通过，搞清楚参数解析是怎么做的，这个以后你自己写程序的时候，很有用，另外消息队列模型的 API 调用流程，也要搞清楚，要知道他们都属于 System V 系列，后面我们学共享内存和信号量，能看到完全类似的 API 调用流程。

欢迎留言和我分享你的疑惑和见解，也欢迎可以收藏本节内容，反复研读。你也可以把今天的内容分享给你的朋友，和他一起学习和进步。

趣谈 Linux 操作系统

像故事一样的操作系统入门课

刘超

网易杭州研究院

云计算技术部首席架构师



新版升级：点击「👤 请朋友读」，10位好友免费读，邀请订阅更有**现金**奖励。

© 版权归极客邦科技所有，未经许可不得传播售卖。页面已增加防盗追踪，如有侵权极客邦将依法追究其法律责任。

上一篇 35 | 块设备（下）：如何建立代理商销售模式？

下一篇 37 | 信号（上）：项目组A完成了，如何及时通知项目...

精选留言 (11)

写留言



W.jyao

2019-06-21

老师有没有考虑过出网络编程专栏。期待



1



恩言

2019-06-19

正在刷第二遍的趣谈网络协议，也不知道为什么有一种冲动要表达一下自己的想法，虽然趣谈网络协议已经过去了很长时间，但在刷第二遍的时候从整个结构上看真的是太清晰了，层层递进，感觉非常的棒。这次的操作系统这个系列我觉得也值得刷N遍，每次应该都会有不同的认识。



WB

2019-06-19

进程间还可以利用socket通信



安排

2019-06-19

socket估计要单独开一节



大王叫我来巡山

2019-06-24

我最想看的阻塞非阻塞，同步异步一直还没有看到





一笔一画

2019-06-23

请教下老师, system v和posix两套区别是什么? 为什么要搞两套?



WL

2019-06-21

请问老师信号量仅仅是一个对资源数量的标识, 那怎么知道具体是哪个资源被申请和释放了呢, 如果不对具体的资源做标记, 不是还是可能引起冲突吗?



安排

2019-06-19

system v这套api和文件操作那些api在形式上差别较大, 所以现在用的多的是posix进程间通信那套东西。



Sharry

2019-06-19

共享内存的确常用, Android 的 Ashmen 共享内存就是基于 Linux 的共享内存操作



W.jyao

2019-06-19

命令行参数也可以算一种，哈哈。



刘強

2019-06-19

xxxget () 函数，从这个“get”字面意思上来说，是不是也有个资源池或者缓冲什么的机制？

