

30 | 应用间通信（二）：详解Linux进程IPC

2022-10-10 LMOS 来自北京



天下无鱼

<https://shikey.com/>

《计算机基础实战课》

[课程介绍 >](#)



讲述：陈晨

时长 11:19 大小 10.34M



你好，我是 LMOS。

上节课，我们学习了信号和管道这两种通信方法，这节课我们接着看看消息队列和共享内存这两种通信方式。在大型商业系统中，通常会把功能拆分成几大模块，模块以应用形式存在，就需要消息队列和内存共享来使模块之间进行通信和协作，这就是利用通信机制将应用解耦。

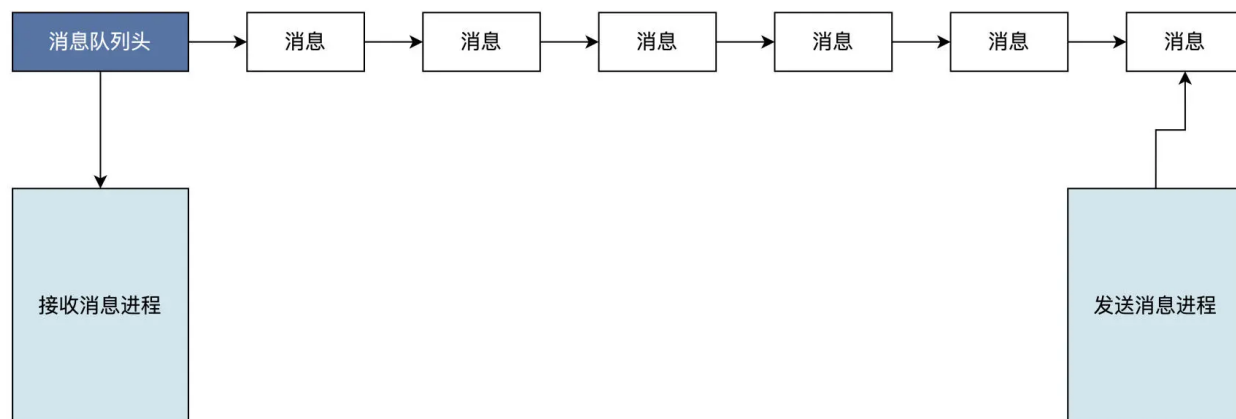
这节课的配套代码，你可以从 [这里](#) 下载。话不多说，我们正式开讲吧！

消息队列

消息队列是 Linux 提供了一种进程间通信方法，**它能让进程之间互相发送消息**。这些消息的形式由进程自己决定，可以是文本，也可以是二进制的，格式可以随意，只要另一个进程认识就行。

你可以把消息想象成一个数据记录，并且这个记录具有特定的格式以及特定的顺序。消息队列实质就是一个“收纳”消息的链表，消息会依次挂入这个叫做队列的链表中，一个链表节点就对应着一个消息。

接下来的问题就是，谁有权操作这个消息队列？答案是对这个消息队列有写权限的进程可以向其中插入新的消息；对消息队列有读权限的进程，则可以从其中读出消息。逻辑结构如下图所示：



极客时间

Linux 采用消息队列来实现消息传递，新的消息总是放在队列的末尾，但接收的时候通常是从队列头开始，也可以从中间抽取。发送消息的方式可以是同步的，也可以是异步的。在同步方式的情况下，发送方在消息队列为满时，要进入等待状态。接收方在消息队列为空时，也要进入等待状态；而异步方式中，发送方和接收方都不必等待，而是直接返回。

Linux 系统下进程间传输消息要分三步走：建立消息队列、发送消息、接收消息。

我猜，聪明的你已经发现了，这三步正好就对应着三个接口函数，代码如下所示：

复制代码

```
1 //获取已经存在的消息队列，或者建立一个新的消息队列
2 // __key是一个整数，可以自己定义
3 // __msgflg是建立消息队列的标志和权限
4 //返回-1 表示失败，其他正整数为消息队列标识，像文件句柄一样
5 int msgget (key_t __key, int __msgflg);
6 //向__msqid表示的消息队列，发送一个新的消息
7 // __msqid表示消息队列
8 // __msgp表示消息结构
9 // __msgsz表示消息大小
10 // __msgflg同步、异步等标志
```

```

11 //返回-1 表示失败，其他表示发送成功
12 int msgsnd (int __msqid, const void *__msgp, size_t __msgsz, int __msgflg);
13 //在__msqid表示的消息队列，接收消息
14 // __msqid表示消息队列
15 // __msgp表示消息结构，用于接收消息内容
16 // __msgsz表示接收消息大小
17 // __msgtyp表示接收消息类型
18 // __msgflg同步、异步等标志
19 //返回-1 表示失败，其他表示成功接收消息的大小
20 ssize_t msgrcv (int __msqid, void *__msgp, size_t __msgsz, long int __msgtyp, i

```

Linux 内核运行过程中缓存了所有的消息队列，这也是为什么 `msgget` 函数能打开一个已经存在的消息队列。只有在 Linux 内核重启或者显示删除一个消息队列时，这个消息队列才会真正被删除。记录消息队列的数据结构（`struct ipc_ids`）位于 Linux 内核中，Linux 系统中的所有消息队列都能在该结构中访问。

在最新版本（2.6 以上的版本）的 Linux 中，`ipc_ids` 包含在 `ipc_namespace` 结构体中，而且 Linux 又定义了一个 `ipc_namespace` 结构的全局变量 `init_ipc_ns`，用来保存 `ipc_namespace` 结构的实例。这里我就不再往下展开了，你有兴趣可以自行研究。

现在我们结合实战练练手，试着用 Linux 消息队列机制，建立一个“自说自话”的聊天软件。这个聊天软件是这样设计的：首先在主进程中建立一个消息队列；然后建立一个子进程，在子进程中等待主进程发过来的消息，并显示出来；最后，主进程等待用户输入消息，并将消息发送给消息队列。

按照这个设计，看上去要分成这样三步去实现：首先我们需要建立消息队列。具体就是调用 `msgget` 函数，还要提供一个消息队列的键，这个键用于表示该消息队列的唯一名字。当这个键对应的消息队列存在的时候，`msgget` 函数将返回该消息队列的标识；如果这个队列不存在，就创建一个消息队列，然后返回这个消息队列的标识。

代码如下所示：

 复制代码

```

1 //消息类型
2 #define MSG_TYPE (041375)
3 //消息队列键
4 #define MSG_KEY (752364)
5 //消息大小
6 #define MSG_SIZE (256)

```

```

7 int main()
8 {
9     pid_t pid;
10    msgid = msgget(MSG_KEY, IPC_CREAT | 0666);
11    if (msgid < 0)
12    {
13        perror("建立消息队列出错\n");
14    }
15    // 建立子进程
16    pid = fork();
17    if (pid > 0)
18    {
19    }
20    else if (pid == 0)
21    {
22    }
23    return 0;
24 }

```

结合代码我们可以看到，msgget 函数的 __mflg 参数是 IPC_CREAT | 0666，其中的 IPC_CREAT 表示没有 MSG_KEY 对应的消息队列就新建一个，0666 则表示该消息队列对应的权限，即所有用户可读写。

接着是第二步实现，成功建立消息队列后，开始调用 fork 函数建立子进程。在子进程里什么也没干，我们这就来写子进程的代码，如下所示：

 复制代码

```

1 //消息体
2 typedef struct Msg
3 {
4     long type;
5     char body[MSG_SIZE];
6 } msg_t;
7 //子进程运行的函数 用于接收消息
8 int receive_main(int mid)
9 {
10    msg_t msg;
11    while (1)
12    {
13        ssize_t sz = msgrcv(mid, &msg, MSG_SIZE, MSG_TYPE, MSG_NOERROR);
14        if (sz < 0)
15        {
16            perror("获取消息失败");
17        }
18        printf("新消息:%s\n", msg.body);
19        //判断是exit就退出
20        if (strncmp("exit", msg.body, 4) == 0)

```

```

21     {
22         printf("结束聊天\n");
23         exit(0);
24     }
25 }
26 return 0;
27 }

```

我来描述一下这段代码的内容。子进程中，在一个循环里调用了 `msgrcv` 函数，接收 `mid` 标识的消息队列中的消息，存放在 `msg` 结构体中，消息的大小和类型都与发送消息一样，`MSG_NOERROR` 表示消息太大也不会出错。随后打印出消息内容，如果是 `exit` 的消息内容，则结束子进程。

最后，我们来完成第三步，有了接收消息的代码，还得有发送代码的程序，我们马上写好它，如下所示：

 复制代码

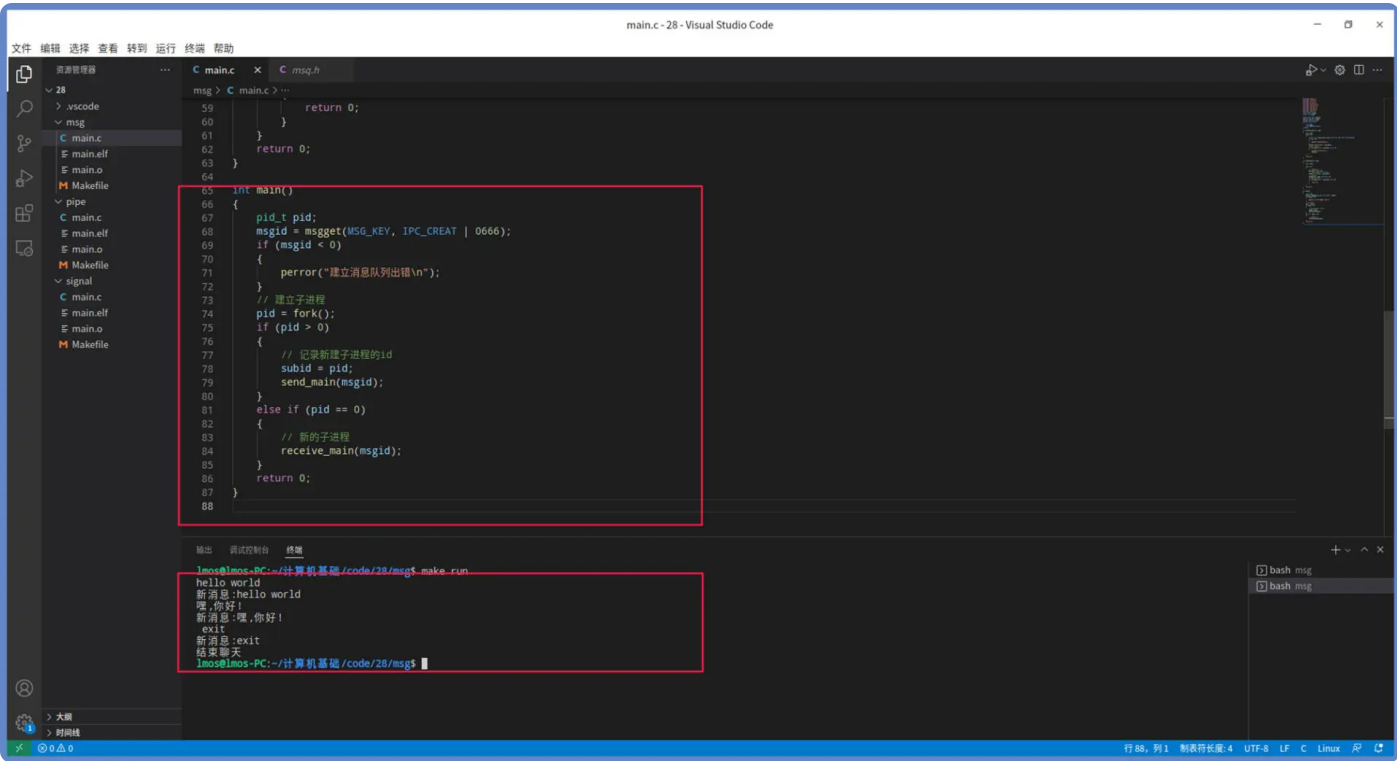
```

1 int send_main(int mid)
2 {
3     msg_t msg;
4     while (1)
5     {
6         // 设置消息类型
7         msg.type = MSG_TYPE;
8         // 获取用户输入并放在消息体中
9         scanf("%[^\n]*c", msg.body);
10        // 发送消息
11        msgsnd(mid, &msg, MSG_SIZE, 0);
12        //判断是exit就退出
13        if (strncmp("exit", msg.body, 4) == 0)
14        {
15            return 0;
16        }
17    }
18    return 0;
19 }

```

对照代码可以看到，发送代码的就是 `send_main` 函数，这个函数由主进程调用，它会在一个循环中设置消息类型后，获取用户输入的消息内容并放入 `msg` 消息结构体中。然后，调用 `msgsnd` 函数向 `mid` 标识的消息队列发送消息，消息来自于 `msg` 结构体变量，指定 `MSG_SIZE` 为消息大小，并且以同步方式发送消息。

现在我们调试一下，如下图所示：



你也可以动手验证一下，如果出现跟我截图中相同的结果，就说明调试成功。

这就是 **Linux** 系统提供给消息队列机制，其作用就是方便进程之间通信，让我们轻松地实现一个简单的聊天软件。不过，聊天是一种特例，更多的时候是进程互相发送消息，通知对方记录数据或者要求对方完成某些工作。

现在，我们已经明白了消息队列机制是怎么回事，**Linux** 的进程间通信机制中还有共享内存这种机制，我们继续往下看。

共享内存

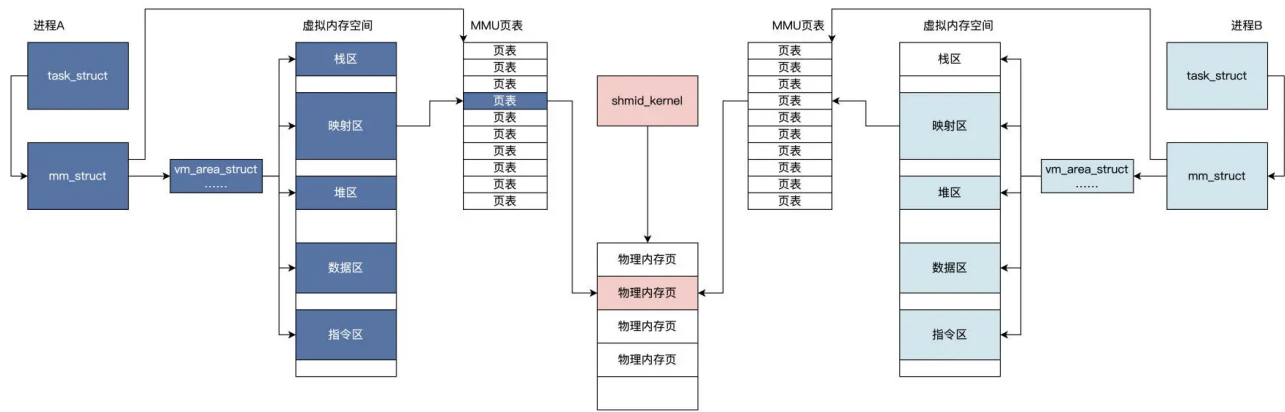
进程间通信实则是进程间传输数据，为了实现更高效率的通信，**Linux** 实现了共享内存这一机制。

共享内存其实是**把同一块物理内存映射到不同进程的虚拟地址空间当中**，不同的进程直接通过修改各自虚拟地址空间当中的内容，就可以完成通信。共享内存几乎不需要进行内存数据拷贝就能实现，即数据从进程 **A** 的虚拟内存空间中写入数据，立即就能被进程 **B** 感知。其它的进程间通信机制，需要经过 **Linux** 内核这种中转站进行多次的数据拷贝操作才可以。因此，使用共享内存通信比较高效。

Linux 内核提供了两种共享内存的实现，一种是基于物理内存映射，另一种是基于 `mmap` 文件映射，这个 `mmap` 函数我们在前面的课程中多次见过了，你可以回顾之前的课程。

这里，我们仅仅讨论基于物理内存映射的实现，它与消息队列很像。Linux 内核会建立一个 `shmid_kernel` 结构，通过 `ipc_namespace` 结构的全局变量 `init_ipc_ns` 结构，就能找到系统中所有的 `shmid_kernel` 结构。该 `shmid_kernel` 结构会关联到一组物理内存页面，最后这组物理内存页面，会映射到各自进程虚拟内存空间中的相关区域。

基于物理内存映射的实现方式，大致逻辑如下图所示：



极客时间

Linux 系统下进程间共享内存也分两步：分别是建立共享内存区和绑定进程内存区，然后就可以读写共享内存了。

这两步对应两个接口函数，代码如下所示：

复制代码

```
1 // 获取已经存在的共享内存，或者建立一个新的共享内存
2 // __key是一个整数可以自己定义
3 // __size是建立共享内存的大小
4 // __shmflg是建立共享内存的标志和权限
5 // 返回-1 表示失败，其他正整数为共享内存标识，像文件句柄一样
6 int shmget (key_t __key, size_t __size, int __shmflg);
7 // 绑定进程内存地址到__shmid的共享内存
8 // __shmid表示建立的共享内存
9 // __shmaddr绑定的地址，传NULL则系统自动分配
10 // __shmflg是绑定地址区间的读写权限
11 // 返回-1,表示失败，其它是成功绑定的地址
12 void *shmat (int __shmid, const void *__shmaddr, int __shmflg);
13 // 解除绑定内存地址
14 // __shmaddr为之前绑定的地址
```

```
15 // 返回-1,表示失败
16 int shmdt (const void *__shmaddr);
```

有了几个接口，我们就来写代码测试一下。我们依然采用建立两个进程的方式，在主进程中写入共享内存，在子进程中读取共享内存，但是我们首先要在主进程中建立共享内存。


我们马上写代码实现它们，如下所示：

 复制代码

```
1 #define SHM_KEY (752364)
2 #define SHM_BODY_SIZE (4096-8)
3 #define SHM_STATUS (SHM_BODY_SIZE)
4 typedef struct SHM
5 {
6     long status;
7     char body[SHM_BODY_SIZE];
8 } shm_t;
9 int main()
10 {
11     pid_t pid;
12     // 建立共享内存
13     shmid = shmget(SHM_KEY, sizeof(shm_t), IPC_CREAT | 0666);
14     if (shmid < 0)
15     {
16         perror("建立共享内存出错\n");
17     }
18     // 建立子进程
19     pid = fork();
20     if (pid > 0)
21     {
22         // 主进程
23         send_main(shmid);
24     }
25     else if (pid == 0)
26     {
27         // 新的子进程
28         receive_main(shmid);
29     }
30     return 0;
31 }
```

上述代码中调用了 `shmget` 函数传入了 `IPC_CREAT`，表示没有 `SHM_KEY` 对应的共享内存，就建立一块共享内存，大小为 `shm` 结构体的大小。

建立好共享内存就可以开始创建子进程了，创建成功后主进程开始执行 `send_main` 函数，子进程运行 `receive_main` 函数。下面我们开始编写这两个函数：

 复制代码

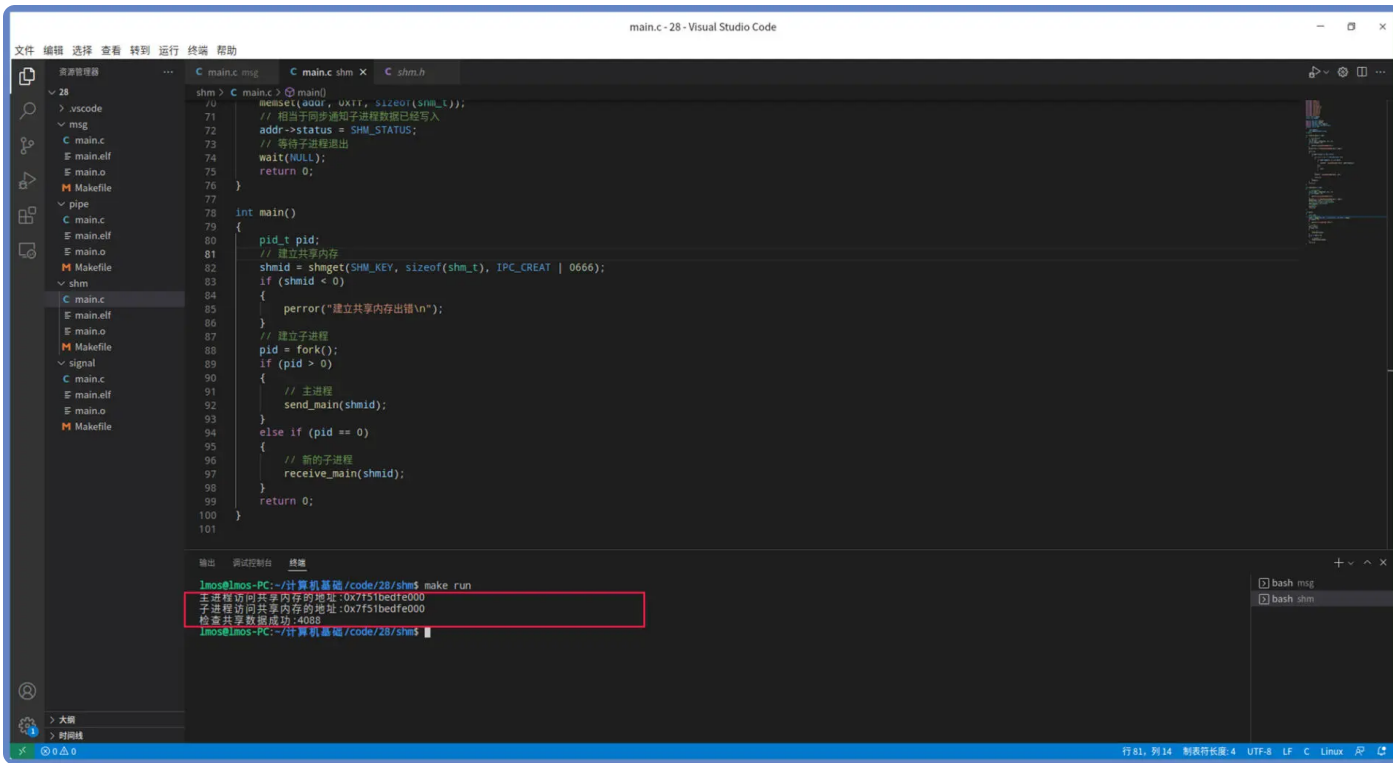
```
1 int receive_main(int mid)
2 {
3     // 绑定共享内存
4     int ok = 0;
5     shm_t* addr = shmat(mid, NULL, 0);
6     if ((long)addr < 0)
7     {
8         perror("绑定共享内存失败\n");
9     }
10    printf("子进程访问共享内存的地址:%p\n", addr);
11    while (1)
12    {
13        if(addr->status == SHM_STATUS)
14        {
15            for (int i = 0; i < SHM_BODY_SIZE; i++)
16            {
17                if (addr->body[i] != (char)0xff)
18                {
19                    printf("检查共享数据失败:%x\n", addr->body[i]);
20                }
21                else
22                {
23                    ok++;
24                }
25            }
26            printf("检查共享数据成功:%d\n", ok);
27            return 0;
28        }
29        sleep(2);
30    }
31    return 0;
32 }
33
34 int send_main(int mid)
35 {
36     // 绑定共享内存
37     shm_t* addr = shmat(mid, NULL, 0);
38     if ((long)addr < 0)
39     {
40         perror("绑定共享内存失败\n");
41     }
42     printf("主进程访问共享内存的地址:%p\n", addr);
43     memset(addr, 0xff, sizeof(shm_t));
44     // 相当于同步通知子进程数据已经写入
45     addr->status = SHM_STATUS;
46     // 等待子进程退出
```

```
47     wait(NULL);
48     return 0;
49 }
```

对照代码可以看到，两个函数都是调用 `shmat` 函数，它们为各自进程绑定了一个虚拟内存地址，并基于该地址访问共享内存。

在 `send_main` 函数中，先把共享的内存写入 `0xff`。最后，设置 `status` 字段用来同步，因为 Linux 对共享不提供任何同步机制，所以需要我们自己处理。`receive_main` 函数中会循环检查 `status` 字段，如果是 `SHM_STATUS`，就对 `addr->body` 中的数据进行一个个检查，并且记录检查结果。

我们来看看执行结果，如下图所示：



上图中的结果证明了我们的设计和预期相符，我们只要往共享内存中写入数据，其它进程立马就感知到了，并且得到了数据。

这就是共享内存的妙处，通过把物理内存页面映射到多个进程的虚拟内存中，使其访问相同的物理内存，数据不需要在各进程之间复制，这是一种性能非常高的进程间通信机制。

重点回顾

课程告一段落，我们做个总结。

进程之间要协作，就需要进程之间可以进行通信。为此 **Linux** 实现了多种通信机制，这节课我们主要探讨了消息队列和共享内存。

消息队列能使进程之间互相发送消息，这些消息的形式格式可以随意设定。从数据结构的角度看，消息队列其实是一个挂载消息的链表。发送消息的进程把消息插入链表，接收消息的进程则从链表上获取消息。同步手段由内核提供，即消息链表空了则接收进程休眠，消息链表满了发送进程就会休眠。

共享内存的实现是把同一块物理内存页面，映射到不同进程的虚拟地址空间当中，进程之间直接通过修改各自虚拟地址空间当中的内容，就能完成数据的瞬间传送。一个进程写入修改了共享内存，另一个进程马上就可以感知到。

不知道你是不是已经发现了一个问题，这些进程通信方式，只能用于本机进程间通信，不能用于远程通信，如果需要对计算机之间的进程远程通信，就需要使用套接字。套接字是一种网络通信编程接口，有兴趣的同学可以自己了解一下。

这节课的导图如下，供你参考：

进程通信（下）

- 管道** 能把一个进程产生的数据输送到另一个进程
- 信号** Linux操作系统为进程设计的一种软件中断机制，具有异步特性

- 消息队列**
 - 如何理解** 让进程之间互相发送消息，消息形式随意
实质就是一个“容纳”消息的链表，消息会依次挂入这个叫做队列的链表中，一个链表节点就对应着一个消息
 - 工作机制**
 - 同步方式
 - 异步方式
 - Linux系统下进程间如何传输消息？**
 - ① 建立消息队列
 - ② 发送消息
 - ③ 接收消息这三步正好对应三个接口函数
 - 实例研究** 用Linux消息队列机制，建立一个极简聊天软件

- 共享内存**
 - 如何理解** 把同一块物理内存映射到不同进程的虚拟地址空间当中
不同的进程之间通信，直接通过修改各自虚拟地址空间当中的内容即可完成通信
一个进程写入修改了共享内存，另一个进程马上就可以感知到
 - 优点** 该方式通信比较高效，性能也非常高
 - 两种实现**
 - 基于物理内存映射
 - 基于mmap文件映射
 - 工作机制** 建立共享内存区和绑定进程内存区 这两步同样对应两个接口函数

计算机历史

硬件芯片
(手写CPU)RISC-V
环境搭建

语言与指令

应用与内存

IO/与文件

综合应用

技术雷达

win

好，今天的课程讲完了，我们下一次再见。

思考题

进程间通信哪些是同步的，哪些是异步的？

期待你在留言区跟我交流互动，也推荐你把这节课分享给更多朋友，共同进步。

分享给需要的人，Ta购买本课程，你将得 20 元

生成海报并分享

上一篇 29 | 应用间通信（一）：详解Linux进程IPC

下一篇 国庆策划01 | 知识挑战赛：检验一下学习成果吧！

精选留言 (1)

写留言



peter

2022-10-11 来自北京

请教老师一个问题：

Q1：消息类型和键的数值有什么含义？

文中消息队列的代码中，“消息类型`#define MSG_TYPE (041375)`；消息队列键`#define MSG_KEY (752364)`”，这两个数字有特别含义吗？还是随机定义的？

