目录

[Linux进程间通信 3](#_Toc40546271)

[1. 初识进程 3](#_Toc40546272)

[1.1进程的概念 3](#_Toc40546273)

[1.1.1程序 3](#_Toc40546274)

[1.1.2进程 3](#_Toc40546275)

[1.1.3进程和程序的联系 4](#_Toc40546276)

[1.1.4进程和程序的区别 4](#_Toc40546277)

[1.2进程的操作（创建、结束、回收） 5](#_Toc40546278)

[1.2.1创建进程 5](#_Toc40546279)

[1.2.2结束进程 8](#_Toc40546280)

[1.2.3回收进程 10](#_Toc40546281)

[2. 进程为什么需要通信 12](#_Toc40546282)

[3. 进程通信之管道通信 15](#_Toc40546283)

[3.1 无名管道 15](#_Toc40546284)

[3.1.1 特点 15](#_Toc40546285)

[3.1.2 创建无名管道 15](#_Toc40546286)

[3.1.3 读、写、关闭管道 15](#_Toc40546287)

[3.1.4 无名管道实现进程间通信 15](#_Toc40546288)

[3.2有名管道 20](#_Toc40546289)

[3.2.1 特点 20](#_Toc40546290)

[3.2.2 创建有名管道 20](#_Toc40546291)

[3.2.3 有名管道实现进程间通信 21](#_Toc40546292)

[4. 进程通信之IPC通信 25](#_Toc40546293)

[4.1 共享内存 26](#_Toc40546294)

[4.1.1特点 26](#_Toc40546295)

[4.1.2创建共享内存 27](#_Toc40546296)

[4.1.3应用程序如何访问共享内存 29](#_Toc40546297)

[4.1.5共享内存实现进程间通信 36](#_Toc40546298)

[4.2消息队列 43](#_Toc40546299)

[4.2.1什么是消息队列 43](#_Toc40546300)

[4.2.2特点 43](#_Toc40546301)

[4.2.3消息队列函数 43](#_Toc40546302)

[4.2.4消息队列实现进程间通信 45](#_Toc40546303)

[4.3信号量灯 48](#_Toc40546304)

[4.3.1什么是P、V操作 48](#_Toc40546305)

[4.3.2什么是信号量灯 48](#_Toc40546306)

[4.3.3信号量灯函数 49](#_Toc40546307)

[4.3.4信号量灯实现进程间同步/互斥 50](#_Toc40546308)

[5. 进程通信之信号通信 54](#_Toc40546309)

[5.1信号机制 54](#_Toc40546310)

[5.2常见信号类型 54](#_Toc40546311)

[5.3信号发送函数 55](#_Toc40546312)

[5.4进程捕捉信号 56](#_Toc40546313)

[6. 进程通信之socket通信 57](#_Toc40546314)

[6.1什么是socket 57](#_Toc40546315)

[6.2相关函数 58](#_Toc40546316)

[6.3 socket实现进程间通信 60](#_Toc40546317)

[6.4一个server和多个client之间的通信 64](#_Toc40546318)

Linux进程间通信

# 初识进程

在日常工作/学习中，读者可能会经常听到如下一些词：“作业”，“任务”，“开了几个线程”，“创建了几个进程”，“多线程”，“多进程”等等。如果系统学习过《操作系统》这门课程，相信大家对这些概念都十分了解。但对很多电子、电气工程专业（或是其他非计算机专业）的同学来说，由于这门课程不是必修课程，我们脑海中可能就不会有这些概念，听到这些概念的时候就会不知所云，不过没有关系，先让我们克服对这些概念的恐惧。比如小时候刚开始学习数学的时候，先从正整数/自然数开始学习，然后逐步接触到分数、小数、负数、有理数、无理数、实数，再到复数等等。这些操作系统中的概念也是这样，让我们从初级阶段开始学起，逐步攻克这些新概念背后的真正含义。

本篇主要讨论linux进程间通信方式，这个主题拆分开始来看，分为三个部分：linux(操作系统)、进程、进程间通信。Linux操作系统本篇暂且不谈，我们主要来关注后两个部分：进程，以及进程间通信。在探讨进程间通信之前，让我们先关注一个知识点概念----进程。

## 1.1进程的概念

### 1.1.1程序

在探讨进程之前，先思考一个问题：什么是程序？

嵌入式软件工程师每天的工作/学习内容就是看C/C++源代码、分析C/C++源代码、编写C/C++源代码（有人会说，应该还有最重要的调试程序，我每天的工作日常是三分写程序，七分调试程序，调试程序去哪里了，大家别着急，这里先卖一个关子）。这些独立的源代码就是一个个程序。它们有一个共同特点，在我们阅读、分析、编写的过程中，此刻都是静态的，它们存储在我们的硬盘上、公司的服务器上。

程序：存储在磁盘上的指令和数据的有序集合。如下就是一个程序，此刻它正安静地躺在硬盘上。

01 #include <stdio.h>

02

03 int main(int argc, char \*argv[])

04{

05 printf("hello world!\n");

06 return 0;

07}

### 1.1.2进程

有了上面程序的概念，先直接给出进程的定义。

进程：**具有一定独立功能的程序在一个数据集合上的一次动态执行过程**。它是动态的，包括创建、调度、执行和消亡（由操作系统完成的）。

定义中的每个词分开来我们都能理解，但是组合到一起成为一个句子时，我们又不知道什么意思了。图灵奖得主Pascal之父尼古拉斯·沃斯，提出过一个著名的公式：。所谓算法就是解决一个问题的方法，程序就是使用算法对特定数据进行处理，这些数据是一个广义上的概念，不单单指像1,2,3,…等等这样的数据。因此用更直白的语言来说，程序开始运行，对数据进行分析处理的过程就是一个进程。

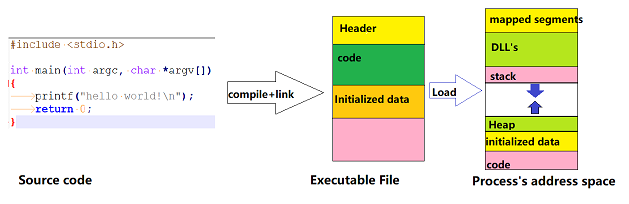
### 1.1.3进程和程序的联系

1. 程序是产生进程的基础。
2. 程序的每次执行构成不同的进程。
3. 进程是程序功能的体现（还记得之前提到的程序员日常工作中的一个重要事项----调试程序吗？调试的过程实际上就是程序的执行，就是本次程序功能的体现，因此这个时候它就是一个进程）。
4. 通过多次执行，一个程序可对应多个进程；通过调用关系，一个进程可包含多个程序。

### 1.1.4进程和程序的区别

|  |  |  |
| --- | --- | --- |
|  | 程序 | 进程 |
| 状态 | 静态的，是有序代码的集合 | 动态的，是程序功能的执行过程 |
| 生命期 | 永久的，长久保存在存储设备上 | 暂时的，一个程序执行结束，则它对应的进程结束 |

下图反应了从程序到进程的变化过程。



我们以一个生活中的例子来加深对进程和程序的理解：

1.有一位计算机科学家，他的女儿要过生日了，他准备给女儿做一个生日蛋糕，于是他去找了一本菜谱，跟着菜谱学习做蛋糕。

菜谱=程序 科学家=CPU 做蛋糕的原材料=数据 做蛋糕的过程=进程

2.科学家正在做蛋糕的时候，突然他的小儿子跑过来，说他的手被扎破了，于是科学家又去找了一本医疗手册，给小儿子处理伤口，处理完伤口之后，继续做生日蛋糕

医疗手册=新程序 给小儿子处理伤口=新进程

从做蛋糕切换到优先包扎伤口=进程切换 处理完伤口继续做生日蛋糕=进程恢复

介绍到这里，希望读者对进程已经建立起一些基础概念了，有关进程的深入部分，我们在这里暂且先不介绍，比如进程的组成包括哪些（代码段，用户数据段，系统数据段）？进程的类型有哪些？进程的状态有哪些等等？这些深入内容，在我们掌握了进程的基础知识之后，读者有兴趣的话，可以查阅相关书籍资料。

## 1.2进程的操作（创建、结束、回收）

### 1.2.1创建进程

使用fork函数来创建一个进程

头文件: #include <unistd.h>

函数原型: pid\_t fork(void);

返回值: 成功时，父进程返回子进程的进程号(>0的非零整数)，子进程中返回0;通过fork函数的返回值区分父子进程。

父进程: 执行fork函数的进程。

子进程: 父进程调用fork函数之后，生成的新进程。

请重点注意：这个函数的返回值和我们接触的绝大部分函数的返回值不一样。

一般地，一个函数的返回值只有一个值，但是该函数的返回值却有两个。实际上关于这个函数的返回值究竟有几个，可以换一种方式来理解，因为这个函数执行之后，系统中会存在两个进程----父进程和子进程，在每个进程中都返回了一个值，所以给用户的感觉就是返回了两个值。

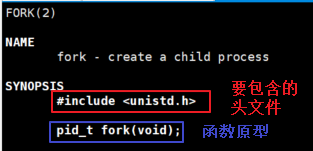
进程的特点：

1. 在linux中，一个进程必须是另外一个进程的子进程，或者说一个进程必须有父进程，但是可以没有子进程。
2. 子进程继承了父进程的内容，包括父进程的代码，变量，pcb，甚至包括当前PC值。在父进程中，PC值指向当前fork函数的下一条指令地址，因此子进程也是从fork函数的下一条指令开始执行。父子进程的执行顺序是不确定的，可能子进程先执行，也可能父进程先执行，取决于当前系统的调度。
3. 父子进程有独立的地址空间、独立的代码空间，互不影响，就算父子进程有同名的全局变量，但是由于它们处在不同的地址空间，因此不能共享。
4. 子进程结束之后，必须由它的父进程回收它的一切资源，否则就会成为僵尸进程。
5. 如果父进程先结束，子进程会成为孤儿进程，它会被INIT进程收养，INIT进程是内核启动之后，首先被创建的进程。

**Tips：**

在linux下，当我们不熟悉某个系统接口API函数时（比如不知道调用这个函数需要包含的头文件，不知道这个函数的每个参数的意义等等），我们可以在ubuntu下使用man命令来查看这个函数的说明。





示例程序（参考：jz2440\process\1th\_create\_process\create\_process.c）

01 /\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

02 \* 功能描述： 创建一个子进程

03 \* 输入参数： 无

04 \* 输出参数： 无

05 \* 返 回 值： 无

06 \* 修改日期 版本号 修改人 修改内容

07 \* -----------------------------------------------

08 \* 2020/05/16 V1.0 zh(ryan) 创建

09 \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/

10

11 #include <stdio.h>

12 #include <stdlib.h>

13 #include <unistd.h>

14 #include <sys/types.h>

15

16 int main(int argc, char \*argv[])

17 {

18 pid\_t pid;

19

20 pid = fork(); // 创建子进程

21

22 if (pid == 0) { // 子进程

23 int i = 0;

24 for (i = 0; i < 5; i++) {

25 usleep(100);

26 printf("this is child process i=%d\n", i);

27 }

28 }

29

30 if (pid > 0) { // 父进程

31 int i = 0;

32 for (i = 0; i < 5; i++) {

33 usleep(100);

34 printf("this is parent process i=%d\n", i);

35 }

36 }

37

38 while(1); //不让进程结束，以便我们查看进程的一些状态信息

39 return 0;

40 }

#### JZ2440实验

在jz2440开发板上实验，读者首先需要创建好NFS文件系统，jz2440开发板从网络文件系统启动，以便运行在ubuntu上编译好的可执行文件，关于如何搭建NFS文件系统请参考视频教程《u-boot\_内核\_根文件系统(ARM裸机1期加强版与2期驱动大全间的衔接)》。读者也可以在ubuntu上执行，将编译器从“arm-linux-gcc”换成“gcc”即可。

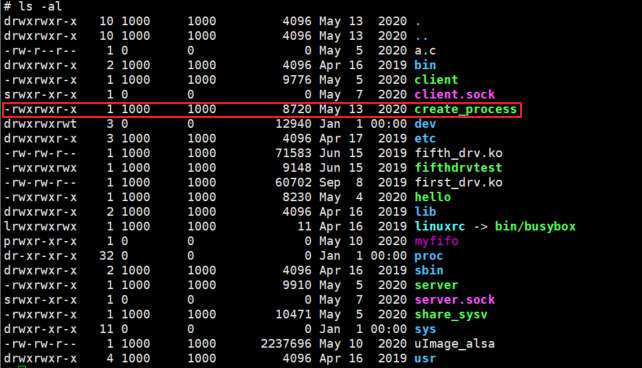
* 编译程序

arm-linux-gcc create\_process.c -o create\_process

* 将可执行文件test拷贝到NFS文件系统对应的目录下

cp create\_process /work/nfs\_root/first\_fs

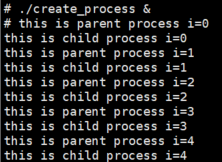
* 在jz2440开发板的串口下此时能看到该可执行文件



* 执行可执行文件

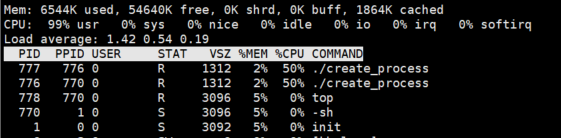
”&”表示在后台执行，这样我们可以继续在串口控制台下敲入命令，控制台能够接收到输入字符并作出响应；如果不加”&”，表示在前台执行，控制台不能对输入字符作出响应。

./create\_process &



* top命令查看进程状态

top



发现此时确实存在两个进程create\_process，其中一个进程PID是777(它的父进程PID是776)，另外一个进程PID是776(它的父进程PID是770)。

### 1.2.2结束进程

使用exit函数来结束一个进程

头文件: #include <stdlib.h>

函数原型: void exit (int status);

使用\_exit函数来结束一个进程

头文件: #include <unistd.h>

函数原型: void \_exit(int status);

两个函数的区别是：**exit结束进程时会刷新缓冲区，\_exit不会**；

这两个退出函数和return函数又有什么区别呢？exit和\_exit函数是返回给操作系统的，return函数是当前函数返回，返回到调用它的函数中，如果正好是在main函数中，return函数也返回给了操作系统，这个时候return和exit、\_exit起到了类似的作用。

程序实验：验证exit和\_exit的区别

示例1：使用exit退出（参考：jz2440\process\2th\_exit\_process\exit\_process.c）

01 /\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

02 \* 功能描述： 使用exit退出当前进程

03 \* 输入参数： 无

04 \* 输出参数： 无

05 \* 返 回 值： 无

06 \* 修改日期 版本号 修改人 修改内容

07 \* -----------------------------------------------

08 \* 2020/05/16 V1.0 zh(ryan) 创建

09 \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/

10 #include <stdio.h>

11 #include <stdlib.h>

12

13 int main(int argc, char \*argv[])

14 {

15 printf("hello world\n");

16 printf("will exit");

17 exit(0); //使用\_exit退出

18 }

示例2：使用\_exit退出（参考：jz2440\process\3th\_exit\_process\exit\_process.c）

01 /\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

02 \* 功能描述： 使用\_exit退出当前进程

03 \* 输入参数： 无

04 \* 输出参数： 无

05 \* 返 回 值： 无

06 \* 修改日期 版本号 修改人 修改内容

07 \* -----------------------------------------------

08 \* 2020/05/16 V1.0 zh(ryan) 创建

09 \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/

10 #include <stdio.h>

11 #include <stdlib.h>

12

13 int main(int argc, char \*argv[])

14 {

15 printf("hello world\n");

16 printf("will exit");

17 \_exit(0); //使用\_exit退出

18 }

在两个示例程序中，第15行比第16行的打印语句多了一个“\n”，它会强制将待打印的字符刷新到缓冲区，为了对比exit和\_exit的区别，在第16行中就没有加上“\n”，按照上面两个退出函数的区别，示例1应该会同时打印“hello world”和“will exit”，示例2程序只会打印“hello world”，不会打印“will exit”，那么到底是不是这样呢？我们在jz2440下验证一下。

#### JZ2440实验

##### 示例1

* 编译

arm-linux-gcc exit\_process.c -o exit\_process

* 拷贝到NFS

cp exit\_process /work/nfs\_root/first\_fs

* 运行

./exit\_process

运行结果，确实同时打印了“hello world”和“will exit”



##### 示例2

* 编译

arm-linux-gcc exit\_process.c -o exit\_process

* 拷贝到NFS

cp exit\_process /work/nfs\_root/first\_fs

* 运行

./exit\_process

运行结果，只打印了“hello world”，没有打印“will exit”



### 1.2.3回收进程

使用wait函数来回收一个进程

头文件: #include <sys/types.h>

#include <sys/wait.h>

函数原型: pid\_t wait(int \*status);

返回值: 成功返回子进程的进程号，失败返回-1

使用waitpid函数来回收一个进程

头文件: #include <sys/types.h>

#include <sys/wait.h>

函数原型: pid\_t waitpid(pid\_t pid, int \*status, int options);

返回值: 成功返回子进程的进程号，失败返回-1

程序示例：子进程退出，父进程回收子进程（参考：jz2440\process\4th\_exit\_wait\exit\_wait.c）

01 /\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

02 \* 功能描述： 使用exit退出子进程，父进程使用waitpid回收子进程的资源

03 \* 输入参数： 无

04 \* 输出参数： 无

05 \* 返 回 值： 无

06 \* 修改日期 版本号 修改人 修改内容

07 \* -----------------------------------------------

08 \* 2020/05/16 V1.0 zh(ryan) 创建

09 \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/

10 #include <unistd.h>

11 #include <stdio.h>

12 #include <stdlib.h>

13 #include <sys/types.h>

14 #include <sys/wait.h>

15

16 int main(int argc, char \*argv[])

17 {

18 int status = -1;

19 pid\_t pid;

20

21 pid = fork();

22 if (pid == 0){ //子进程

23 printf("fork\n");

24 exit(1);

25 } else if (pid > 0) { //父进程

26 pid = waitpid(pid, &status, 0);

27 printf("status=0x%x\n", status);

28 } else {

29 perror("fork\n");

30 }

31

32 return 0;

33 }

#### JZ2440实验

* 编译

arm-linux-gcc exit\_wait.c -o exit\_wait

* 拷贝到NFS

cp exit\_wait /work/nfs\_root/first\_fs

* 运行

./exit\_wait

运行结果



# 进程为什么需要通信

先让我们看如下两个简单的程序，这两个程序中都有一个同名全局变量“global”，唯一的区别是这个全局变量的初始值不同。**说明：以下两个示例程序是为了让我们理解进程的一个特点，因此实验环境是Ubuntu虚拟机**。

程序1：

01 #include <stdio.h>

02 int global = 1;

03

04 void delay(void)

05 {

06 unsigned int a = 1000000;

07 while(a--);

08 }

09

10 int main(int argc, char \*argv[])

11 {

12 while (1) {

13 printf("global=%d\n", global);

14 delay();

15 }

16 return 0;

17 }

程序2：

01 #include <stdio.h>

02 int global = 2;

03

04 void delay(void)

05 {

06 unsigned int a = 1000000;

07 while(a--);

08 }

09

10 int main(int argc, char \*argv[])

11 {

12 while (1) {

13 printf("global=%d\n", global);

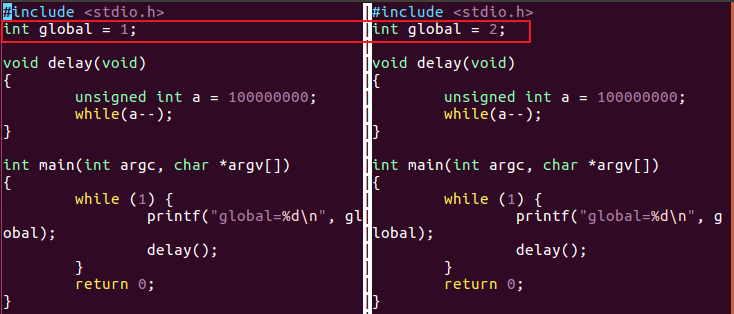
14 delay();

15 }

16 return 0;

17 }

两个程序的唯一区别如下红框所示：



* 编译程序

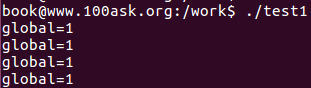
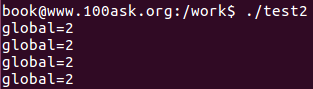
gcc test1.c -o test1

gcc test2.c -o test2

* 运行程序

./test1

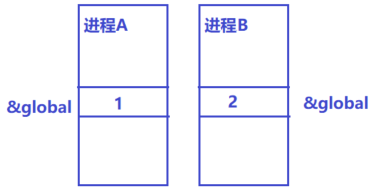
./test2

程序1运行结果 程序2运行结果

我们发现，两个程序运行之后，当前进程中的全局变量global的值并不会改变，它不会被改变成另外一个进程中的值，由此引出的进程的一个特点：**进程资源的唯一性，不共享性，它不能访问别的进程中的数据(地址空间)，也不能被别的进程访问本身的数据(地址空间)**。每个进程对其他进程而言，就是一个黑盒(后面读者学习到线程的时候，会发现在这个特性上，线程是有别于进程的)。

那么为什么会这样呢？这是因为操作系统为了保证系统的安全（进程A奔溃不会影响进程B，进程B仍然会继续运行），它会为每个进程分配特定的地址空间，每个进程只能在这个特定的地址空间执行指令、访问数据，如下图所示。程序需要访问某个变量时，都是通过变量地址去访问该变量的，在不同的进程中，同名变量对应不同的地址(处在当前进程地址空间范围内)，进程无法访问分配给它的地址范围之外的地址空间，自然就无法获得其他进程中的变量值。



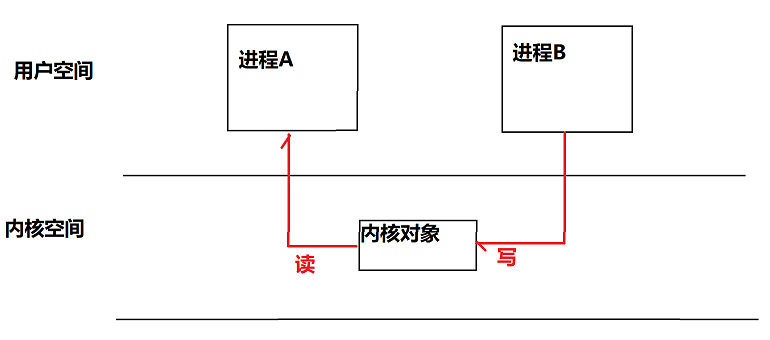
进程间为何需要通信呢？从上面的两个示例程序中，可以得知：**不同进程之间无法互相访问对方的地址空间**。但是在我们实际的项目开发中，为了实现各种各样的功能，不同进程之间一定需要数据交互，那么我们应该如何实现进程间数据交互呢？这就是进程间通信的目的：**实现不同进程之间的数据交互**。

在linux下，内存空间被划分为用户空间和内核空间，应用程序开发人员开发的应用程序都存在于用户空间，绝大部分进程都处在用户空间；驱动程序开发人员开发的驱动程序都存在于内核空间。

在用户空间，不同进程不能互相访问对方的资源，因此，在用户空间是无法实现进程间通信的。为了实现进程间通信，必须在内核空间，由内核提供相应的接口来实现，linux系统提供了如下四种进程通信方式。

|  |  |
| --- | --- |
| 进程间通信方式 | 分类 |
| 管道通信 | 无名管道、有名管道 |
| IPC通信 | 共享内存、消息队列、信号灯 |
| 信号通信 | 信号发送、接收、处理 |
| socket通信 | 本地socket通信，远程socket通信 |

linux有一个最基本的思想----“**一切皆文件**”，内核中实现进程间通信也是基于文件读写思想。不同进程通过操作内核里的同一个内核对象来实现进程间通信，如下图所示，这个内核对象可以是管道、共享内存、消息队列、信号灯、信号，以及socket。



# 进程通信之管道通信

管道分为无名管道和有名管道，其特点如下

|  |  |
| --- | --- |
| 类型 | 特点 |
| 无名管道 | 在文件系统中没有文件节点，**只能用于具有亲缘关系的进程间通信(比如父子进程)** |
| 有名管道 | 在文件系统中有文件节点，**适用于在同一系统中 的任意两个进程间通信** |

## 3.1 无名管道

### 3.1.1 特点

无名管道实际上就是一个单向队列，**在一端进行读操作，在另一端进行写操作**，所以需要两个文件描述符，描述符fd[0]指向读端，fd[1]指向写端。它是一个特殊的文件，所以无法使用简单open函数创建，我们需要pipe函数来创建。它只能用于具有亲缘关系的两个进程间通信。



### 3.1.2 创建无名管道

1.头文件#include <unistd.h>

2.函数原型: int pipe(int fd[2])

3.参数: 管道文件描述符，有两个文件描述符，分别是fd[0]和fd[1]，管道有一个读端

4. fd[0]和一个写端fd[1]

5.返回值： 0表示成功；1表示失败

### 3.1.3 读、写、关闭管道

1.读管道 read，读管道对应的文件描述符是fd[0]

2.写管道 write，写管道对应的文件描述符是fd[1]

3.关闭管道 close，因为创建管道时，会同时创建两个管道文件描述符，分别是读管道文件描述符fd[0]和写管道文件描述符fd[1]，因此需要关闭两个文件描述符

### 3.1.4 无名管道实现进程间通信

#### 程序示例1

（参考：jz2440\process\_pipe\1th\_write\_pipe\my\_pipe\_write.c）

01 /\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

02 \* 功能描述： 创建一个管道，并向管道中写入字符串，然后从管道中读取，验证

03 能否读取之前写入的字符串

04 \* 输入参数： 无

05 \* 输出参数： 无

06 \* 返 回 值： 无

07 \* 修改日期 版本号 修改人 修改内容

08 \* -----------------------------------------------

09 \* 2020/05/16 V1.0 zh(ryan) 创建

10 \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/

11 #include <stdio.h>

12 #include <stdlib.h>

13 #include <unistd.h>

14

15 int main(int argc, char \*argv[])

16 {

17 int fd[2];

18 int ret = 0;

19 char write\_buf[] = "Hello linux";

20 char read\_buf[128] = {0};

21

22 ret = pipe(fd);

23 if (ret < 0) {

24 printf("create pipe fail\n");

25 return -1;

26 }

27 printf("create pipe sucess fd[0]=%d fd[1]=%d\n", fd[0], fd[1]);

28

29 //向文件描述符fd[1]写管道

30 write(fd[1], write\_buf, sizeof(write\_buf));

31

32 //从文件描述符fd[0]读管道

33 read(fd[0], read\_buf, sizeof(read\_buf));

34 printf("read\_buf=%s\n", read\_buf);

35

36 close(fd[0]);

37 close(fd[1]);

38 return 0;

39 }

#### JZ2440实验

* 编译

arm-linux-gcc my\_pipe\_write.c -o my\_pipe\_write

* 拷贝到NFS文件系统

cp my\_pipe\_write /work/nfs\_root/first\_fs

* 运行

./my\_pipe\_write

运行结果，发现能够正确读到管道中的字符串” Hello linux”。



#### 程序示例2

在利用无名管道实现进程间通信之前，先让我们看一下如下的程序：我们知道父子进程的执行顺序是不确定的，是受系统调度的。我们在父进程中创建一个子进程，我们想让父进程控制子进程的运行，父进程设置“process\_inter=1”，当“process\_inter=1”时，子进程才会执行打印操作，否则子进程不执行打印操作。我们看如下的程序能够实现我们的目的吗？

（参考：jz2440\process\_pipe\2th\_comm\test.c）

01 /\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

02 \* 功能描述： 1.在父进程中创建一个子进程，

03 2.父进程执行完后，将变量process\_inter赋值为1；

04 3.子进程判断process\_inter为1则执行后面的打印语句，否则不执行。

05 \* 输入参数： 无

06 \* 输出参数： 无

07 \* 返 回 值： 无

08 \* 修改日期 版本号 修改人 修改内容

09 \* -----------------------------------------------

10 \* 2020/05/16 V1.0 zh(ryan) 创建

11 \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/

12 #include <stdio.h>

13 #include <stdlib.h>

14 #include <unistd.h>

15 #include <sys/types.h>

16

17 int main(int argc, char \*argv[])

18 {

19 pid\_t pid;

20 int process\_inter = 0;

21

22 pid = fork(); // 创建子进程

23

24 if (pid == 0) { // 子进程

25 int i = 0;

26 while (process\_inter == 0); //

27 for (i = 0; i < 5; i++) {

28 usleep(100);

29 printf("this is child process i=%d\n", i);

30 }

31 }

32

33 if (pid > 0) { // 父进程

34 int i = 0;

35 for (i = 0; i < 5; i++) {

36 usleep(100);

37 printf("this is parent process i=%d\n", i);

38 }

39 process\_inter == 1;

40 }

41

42 while(1);

43 return 0;

44 }

#### JZ2440实验

* 编译

arm-linux-gcc test.c -o test

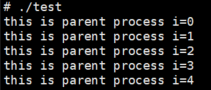
* 拷贝到NFS文件系统

cp test /work/nfs\_root/first\_fs

* 运行

./test

运行结果，发现第29行打印语句一直没有，子进程中process\_inter一直为0。



#### 程序示例3

（参考：jz2440\process\_pipe\3th\_pipe\_comm\comm\_fork.c）

01 /\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

02 \* 功能描述： 1.使用无名管道实现父子进程通信

03 2.父进程向管道中写入一个值

04 3.子进程从管道中读取该值，如果非零，则执行后面的打印，否则不执行

05 \* 输入参数： 无

06 \* 输出参数： 无

07 \* 返 回 值： 无

08 \* 修改日期 版本号 修改人 修改内容

09 \* -----------------------------------------------

10 \* 2020/05/16 V1.0 zh(ryan) 创建

11 \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/

12

13 #include <stdio.h>

14 #include <stdlib.h>

15 #include <unistd.h>

16 #include <sys/types.h>

17

18 int main(int argc, char \*argv[])

19 {

20 pid\_t pid;

21 char process\_inter = 0;

22 int fd[2], ret = 0;

23

24 ret = pipe(fd); //创建一个无名管道，必须在创建子进程之前

25 if (ret < 0) {

26 printf("create pipe fail\n");

27 return -1;

28 }

29 printf("create pipe sucess\n");

30

31 pid = fork(); //创建子进程

32

33 if (pid == 0) { // 子进程

34 int i = 0;

35 read(fd[0], &process\_inter, sizeof(process\_inter)); // 如果管道为空，则休眠等待

36 while (process\_inter == 0);

37 for (i = 0; i < 5; i++) {

38 usleep(100);

39 printf("this is child process i=%d\n", i);

40 }

41 } else if (pid > 0) { // 父进程

42 int i = 0;

43 for (i = 0; i < 5; i++) {

44 usleep(100);

45 printf("this is parent process i=%d\n", i);

46 }

47 process\_inter = 1;

48 sleep(2);

49 write(fd[1], &process\_inter, sizeof(process\_inter));

50 }

51

52 while(1);

53 return 0;

54 }

#### JZ2440实验

* 编译

arm-linux-gcc comm\_fork.c -o comm\_fork

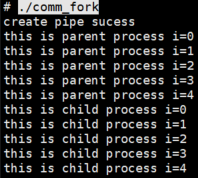
* 拷贝到NFS文件系统

cp comm\_fork /work/nfs\_root/first\_fs

* 运行

./comm\_fork

运行结果，因为第38行2s延时，父进程打印结束后大约2s左右的时间，子进程中的打印也正确输出了，如下所示。



## 3.2有名管道

### 3.2.1 特点

所谓有名管道，顾名思义，就是在内核中存在一个文件名，表明这是一个管道文件。Linux中存在7种文件类型，分别如下。

|  |  |
| --- | --- |
| 文件类型 | 文件特点 |
| 普通文件 | 标识符 ‘-’ ，用open方式创建 |
| 目录文件 | 标识符 ‘d’ ，用mkdir方式创建 |
| 链接文件 | 标识符 ‘l’, la -s， 又可以分为软链接，硬链接 |
| (有名)管道文件 | 标识 ‘p’， 用mkfifo创建 |
| socket文件 | 标识符 ‘s’，用socket创建 |
| 字符设备文件 | 标识符 ‘c’ |
| 块设备文件 | 标识符 ‘b’ |

有名管道既可以用于具有亲缘关系的进程间通信，又可以用于非亲缘关系的进程间通信，在我们的实际项目中，很多进程之间是不具有亲缘关系的，因此有名管道使用的情况会更普遍一些。

### 3.2.2 创建有名管道

函数原型 ： int mkfifo(const char \* filename, mode\_t mode)

参数 ：管道文件文件名，权限，创建的文件权限仍然和umask有关系

返回值 ： 成功返回0，失败返回-1

注意：mkfifo并没有在内核中生成一个管道，只是在用户空间生成了一个有名管道文件

### 3.2.3 有名管道实现进程间通信

#### 示例程序1

创建一个有名管道文件(参考：jz2440\process\_pipe\4th\_create\_myfifo\create\_myfifo.c)

01 /\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

02 \* 功能描述： 1.创建一个有名管道

03 \* 输入参数： 无

04 \* 输出参数： 无

05 \* 返 回 值： 无

06 \* 修改日期 版本号 修改人 修改内容

07 \* -----------------------------------------------

08 \* 2020/05/16 V1.0 zh(ryan) 创建

09 \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/

10

11 #include <stdio.h>

12 #include <stdlib.h>

13 #include <unistd.h>

14 #include <sys/types.h>

15

16 int main(int argc, char \*argv[])

17 {

18 int ret;

19

20 ret = mkfifo("./myfifo", 0777); //创建有名管道，文件权限为777

21 if (ret < 0) {

22 printf("create myfifo fail\n");

23 return -1;

24 }

25 printf("create myfifo sucess\n");

26

27 return 0;

28 }

#### JZ2440实验

* 编译

arm-linux-gcc create\_myfifo.c -o create\_myfifo

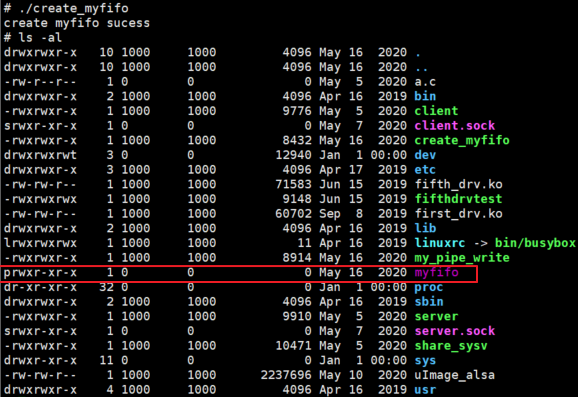
* 拷贝到NFS文件系统

cp create\_myfifo /work/nfs\_root/first\_fs

* 运行

./create\_myfifo

运行结果，发现在当前目录下生成一个有名管道文件myfifo（文件类型是“-p”）。



#### 示例程序2

进程1源码(参考：jz2440\process\_pipe\5th\_myfifo\_comm\5nd\_named\_pipe.c)

01 /\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

02 \* 功能描述： 1.进程1中创建一个有名管道3rd\_fifo，权限是0777

03 2.以写方式打开这个有名管道文件，并向其中写入一个值

04 \* 输入参数： 无

05 \* 输出参数： 无

06 \* 返 回 值： 无

07 \* 修改日期 版本号 修改人 修改内容

08 \* -----------------------------------------------

09 \* 2020/05/16 V1.0 zh(ryan) 创建

10 \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/

11

12 #include <stdio.h>

13 #include <stdlib.h>

14 #include <unistd.h>

15 #include <sys/types.h>

16 #include <fcntl.h>

17

18 int main(int argc, char \*argv[])

19 {

20 int i, ret, fd;

21 char p\_flag = 0;

22

23 /\* 创建有名管道 \*/

24 if (access("./3rd\_fifo", 0) < 0) { //先判断有名管道文件是否存在,不存在需要先创建

25 ret = mkfifo("./3rd\_fifo", 0777);

26 if (ret < 0) {

27 printf("create named pipe fail\n");

28 return -1;

29 }

30 printf("create named pipe sucess\n");

31 }

32

33 /\* 打开有名管道，以写方式打开 \*/

34 fd=open("./3rd\_fifo", O\_WRONLY);

35 if (fd < 0) {

36 printf("open 3rd\_fifo fail\n");

37 return -1;

38 }

39 printf("open 3rd\_fifo sucess\n");

40

41 for (i = 0; i < 5; i++) {

42 printf("this is first process i=%d\n", i);

43 usleep(100);

44 }

45 p\_flag = 1;

46 sleep(5);

47 write(fd, &p\_flag, sizeof(p\_flag));

48

49 while(1);

50 return 0;

51 }

进程2源码(参考：jz2440\process\_pipe\5th\_myfifo\_comm\5nd\_named\_pipe\_2.c)

01 /\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

02 \* 功能描述： 1.只读方式打开这个有名管道文件，并读取这个值

03 2.当这个值非零时，继续执行后面的打印输出语句

04 \* 输入参数： 无

05 \* 输出参数： 无

06 \* 返 回 值： 无

07 \* 修改日期 版本号 修改人 修改内容

08 \* -----------------------------------------------

09 \* 2020/05/16 V1.0 zh(ryan) 创建

10 \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/

11

12 #include <stdio.h>

13 #include <stdlib.h>

14 #include <unistd.h>

15 #include <sys/types.h>

16 #include <fcntl.h>

17

18 int main(int argc, char \*argv[])

19 {

20 int i;

21 int fd=open("./3rd\_fifo", O\_RDONLY);

22 char p\_flag = 0;

23

24 if (fd < 0) {

25 printf("open 3rd\_fifo fail\n");

26 return -1;

27 }

28

29 printf("open 3rd\_fifo sucess\n");

30 read(fd, &p\_flag, sizeof(p\_flag));

31 while(!p\_flag);

32 for (i = 0; i < 5; i++) {

33 printf("this is second process i=%d\n", i);

34 usleep(100);

35 }

36

37 while(1);

38 return 0;

39 }

#### JZ2440实验

* 编译

arm-linux-gcc 5nd\_named\_pipe.c -o 5nd\_named\_pipe

arm-linux-gcc 5nd\_named\_pipe\_2.c -o 5nd\_named\_pipe\_2

* 拷贝到NFS文件系统

cp 5nd\_named\_pipe /work/nfs\_root/first\_fs

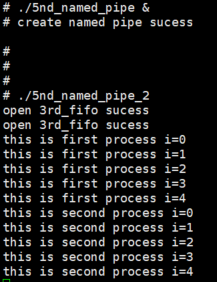
cp 5nd\_named\_pipe\_2 /work/nfs\_root/first\_fs

* 运行

注意我们这里都在后台运行可执行程序，方便我们在串口中断下多次输入。

./5nd\_named\_pipe &

./5nd\_named\_pipe\_2 &

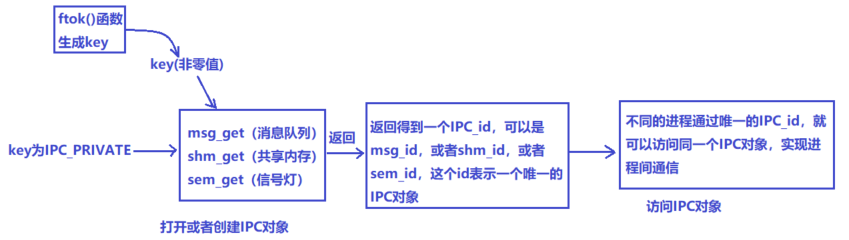


# 进程通信之IPC通信

IPC通信分为共享内存、消息队列以及信号灯。这些IPC对象（共享内存、消息队列、信号灯）都存在于内核空间中。

应用程序使用IPC通信的一般步骤如下：

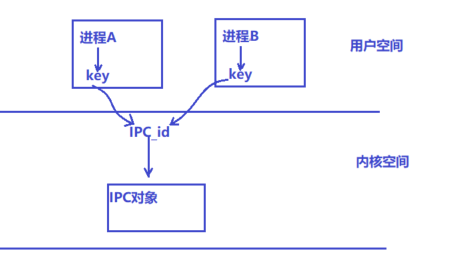
* 首先生成一个key值。有两种生成key的方式，一种是使用宏IPC\_PRIVATE表示一个key，它表示一个私有对象，只能用于当前进程或者具有亲缘关系的进程访问。另一种是使用ftok函数来生成一个key值，这种方式创建的IPC对象可以被不同的进程访问。
* 使用生成的key值，创建一个IPC对象（如果是已经创建好的IPC对象，则打开该IPC对象），这个时候每个IPC对象都有一个唯一的ID号（IPC\_id，可以是shm\_id，msg\_id，sem\_id，每个id代表一个IPC对象）。
* 进程通过IPC\_id，调用访问IPC通道的读写函数来操作IPC对象。调用shmctrl，shmat，shmdt来访问共享内存；调用msgctrl，msgsnd，msgrecv访问消息队列；调用semctrl，semop访问信号灯。



如何理解key和IPC\_id（shm\_id/msg\_id/sem\_id）

回答这个问题，请先思考一个问题，应用程序如何访问一个IPC对象（共享内存，消息队列、信号量灯）？

显然，我们需要一个唯一表示该IPC对象的身份ID（IPC\_id，该IPC\_id是由操作系统来管理的），但是由于这个ID只在当前创建该IPC对象的进程中可以获取到，在别的IPC进程中都没法获取，那么如何得到IPC对象的ID呢？这个时候就需要key值了，它相当于IPC\_id的一个别名，或者叫做外部名，因此key值必须也是唯一的，这样才能得到唯一的IPC对象id。不同进程通过同一个key值得到同一个IPC对象id，来访问同一个IPC对象。如下图所示



ftok函数

函数原型 : char ftok(const char \*path, char key)

参数 ： path，存在并且可以访问的文件路径

key，一个字符

返回值 ： 正确返回一个key值，出错返回-1

为何需要ftok函数先生成key，然后再创建IPC对象？

这就类似于无名管道和有名管道的区别，使用IPC\_PRIVATE宏创建的共享内存就类似于无名管道，只能实现有亲缘关系的进程间通信。

那么为什么又需要使用ftok生成一个key值呢？是否可以直接指定一个非零值呢？直接指定一个非零的key值做法是不建议的，因为读者自己指定的key值很有可能于系统中已经存在的key值一样。

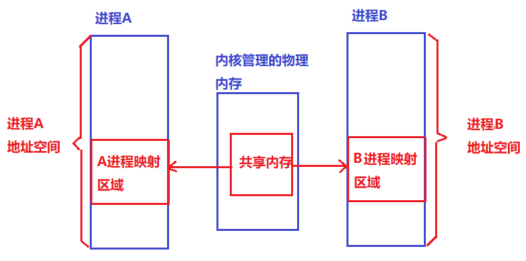
ftok函数创建了一个key值之后，就类似于有名管道，既可以实现具有亲缘关系的进程间通信，又能够实现非亲缘关系的进程间通信。

## 4.1 共享内存

### 4.1.1特点

所谓共享内存是指多个进程都可以访问的同一块地址空间，但是我们知道Linux操作系统为了保证系统执行的安全，为每个进程划分了各自独立的地址空间，每个进程不能访问别的进程的地址空间，那么共享内存实现的原理是什么呢？

内核开辟一块物理内存区域，进程本身将这片内存空间映射到自己的地址空间进行读写。



从图中可以看到，进程可以直接访问这片内存，数据不需要在两进程间复制，所以速度较快。共享内存没有任何的同步与互斥机制，所以要使用信号量来实现对共享内存的存取的同步。

当需要使用共享内存进行通信时，一般步骤如下：

* 先创建一片共享内存，该内存存在于内核空间中。
* 进程通过key值找到这片共享内存的唯一ID，然后将这片共享内存映射到自己的地址空间。
* 每个进程通过读写映射后的地址，来访问内核中的共享内存。

### 4.1.2创建共享内存

函数原型 : int shmget(key\_t key, int size, int shmflg)

头文件: #include <sys/shm.h>

函数参数 : key: IPC\_PRIVATE 或 ftok的返回值

IPC\_PRIVATE返回的key值都是一样的,都是0

size : 共享内存区大小

shmflg : 同open函数的权限位，也可以用八进制表示法

返回值 ： 成功，共享内存段标识符ID； -1 出错

#### 程序示例1（参考jz2440\process\_ipc\1st\_shm\1st\_shm.c）

01 /\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

02 \* 功能描述： 1.使用IPC\_PRIVATE创建共享内存

03 \* 输入参数： 无

04 \* 输出参数： 无

05 \* 返 回 值： 无

06 \* 修改日期 版本号 修改人 修改内容

07 \* -----------------------------------------------

08 \* 2020/05/16 V1.0 zh(ryan) 创建

09 \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/

10

11 #include <stdio.h>

12 #include <stdlib.h>

13 #include <unistd.h>

14 #include <sys/types.h>

15 #include <sys/shm.h>

16 #include <signal.h>

17

18 int main(int argc, char \*argv[])

19 {

20 int shmid;

21

22 shmid = shmget(IPC\_PRIVATE, 128, 0777);

23 if (shmid < 0) {

24 printf("create shared memory fail\n");

25 return -1;

26 }

27 printf("create shared memory sucess, shmid = %d\n", shmid);

28 system("ipcs -m");

29 return 0;

30 }

#### JZ2440实验

* 编译

arm-linux-gcc 1st\_shm.c -o 1st\_shm

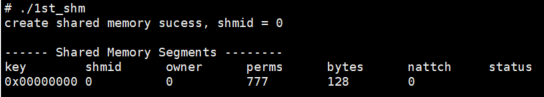
* 拷贝到NFS文件系统

cp 1st\_shm /work/nfs\_root/first\_fs

* 运行

执行第18行程序后，会在串口打印如下信息，这行语句的作用和我们直接在串口console下面输入“ipcs -m”是一样的。我们发现此时共享内存的key值为0。

./1st\_shm



#### 程序示例2（参考jz2440\process\_ipc\1st\_shm\2nd\_shm.c）

程序源码，使用fotk函数生成一个key值

01 /\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

02 \* 功能描述： 1.使用ftok函数生成的key创建共享内存

03 \* 输入参数： 无

04 \* 输出参数： 无

05 \* 返 回 值： 无

06 \* 修改日期 版本号 修改人 修改内容

07 \* -----------------------------------------------

08 \* 2020/05/16 V1.0 zh(ryan) 创建

09 \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/

10

11 #include <stdio.h>

12 #include <stdlib.h>

13 #include <unistd.h>

14 #include <sys/types.h>

15 #include <sys/shm.h>

16 #include <signal.h>

17

18 int main(int argc, char \*argv[])

19 {

20 int shmid;

21 int key;

22

23 key = ftok("./a.c", 'a'); //先创建一个key值

24 if (key < 0) {

25 printf("create key fail\n");

26 return -1;

27 }

28 printf("create key sucess key = 0x%X\n",key);

29

30 shmid = shmget(key, 128, IPC\_CREAT | 0777);

31 if (shmid < 0) {

32 printf("create shared memory fail\n");

33 return -1;

34 }

35 printf("create shared memory sucess, shmid = %d\n", shmid);

36 system("ipcs -m");

37 return 0;

38 }

#### JZ2440实验

* 编译

arm-linux-gcc 2nd\_shm.c -o 2nd\_shm

* 拷贝到NFS文件系统

cp 2nd\_shm /work/nfs\_root/first\_fs

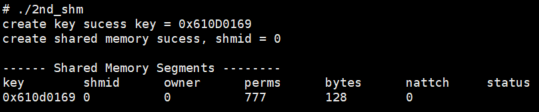
* 运行

我们需要在2nd\_shm所在的同级目录下创建一个文件a.c（在jz2440开发板上）

touch a.c

我们发现此时共享内存的key值为非零值0x610d0169.

./2nd\_shm



### 4.1.3应用程序如何访问共享内存

我们知道创建的共享内存还是处于内核空间中，用户程序不能直接访问内核地址空间，那么用户程序如何访问这个共享内存呢？

#### shmat函数

将共享内存映射到用户空间，这样应用程序就可以直接访问共享内存了

函数原型 ： void \*shmat(int shmid, const void \*shmaddr, int shmflg)

参数 ： shmid ID号

shmaddr 映射地址， NULL为系统自动完成的映射

shmflg SHM\_RDONLY共享内存只读

默认是0，可读可写

返回值：成功，映射后的地址；失败，返回NULL

程序示例（参考jz2440\process\_ipc\1st\_shm\3nd\_shm.c）

01 /\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

02 \* 功能描述： 1.创建共享内存，将该共享内存地址通过shmat映射到用户地址空间

03 2.用户通过标准输入向这个共享内存中输入一行字符串

04 3.然后从该共享内存中读取内容，验证是否能够读取到

05 \* 输入参数： 无

06 \* 输出参数： 无

07 \* 返 回 值： 无

08 \* 修改日期 版本号 修改人 修改内容

09 \* -----------------------------------------------

10 \* 2020/05/16 V1.0 zh(ryan) 创建

11 \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/

12 #include <stdio.h>

13 #include <stdlib.h>

14 #include <unistd.h>

15 #include <sys/types.h>

16 #include <sys/shm.h>

17 #include <signal.h>

18

19 int main(int argc, char \*argv[])

20 {

21 int shmid;

22 int key;

23 char \*p;

24

25 key = ftok("./a.c", 'b');

26 if (key < 0) {

27 printf("create key fail\n");

28 return -1;

29 }

30 printf("create key sucess key = 0x%X\n",key);

31

32 shmid = shmget(key, 128, IPC\_CREAT | 0777);

33 if (shmid < 0) {

34 printf("create shared memory fail\n");

35 return -1;

36 }

37 printf("create shared memory sucess, shmid = %d\n", shmid);

38 system("ipcs -m");

39

40 p = (char \*)shmat(shmid, NULL, 0);

41 if (p == NULL) {

42 printf("shmat fail\n");

43 return -1;

44 }

45 printf("shmat sucess\n");

46

47 //等待console输入，然后向共享内存写入数据

48 fgets(p, 128, stdin);

49

50 //读共享内存

51 printf("share memory data:%s\n", p);

52

53 //再次读共享内存

54 printf("share memory data:%s\n", p);

55 return 0;

56 }

##### JZ2440实验

* 编译

arm-linux-gcc 3nd\_shm.c -o 3nd\_shm

* 拷贝到NFS文件系统

cp 3nd\_shm /work/nfs\_root/first\_fs

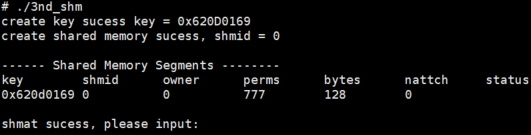
* 运行

我们需要在3nd\_shm所在的同级目录下创建一个文件a.c（在jz2440开发板上）

touch a.c

此时会提示用户输入信息

./3nd\_shm



我们在console下输入任意字符，比如“hello linux”，然后按下回车，发现打印如下



问题：代码中第51行读了一遍共享内存，然后第54行又读了一遍共享内存，发现两次都能读到共享内存的内容，说明共享内存被读了之后，内容仍然存在。而在管道中，读了一遍管道内容之后，如果紧接着读取第二遍（在没有新写入的前提下），我们是不能读到管道中的内容的，说明管道只要读取一次之后，内容就消失了，读者可以通过实验自行验证一下。

#### shmdt函数

函数原型：int shmdt(const void \*shmaddr)

参数 ； shmat的返回值

返回值 : 成功0，出错-1

程序示例（参考jz2440\process\_ipc\1st\_shm\4th\_shm.c）

01 /\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

02 \* 功能描述： 1.创建共享内存，将该共享内存地址通过shmat映射到用户地址空间

03 2.用户通过标准输入向这个共享内存中输入一行字符串

04 3.然后从该共享内存中读取内容

05 4.调用shmdt解除地址映射，此时应用程序继续访问会出错

06 \* 输入参数： 无

07 \* 输出参数： 无

08 \* 返 回 值： 无

09 \* 修改日期 版本号 修改人 修改内容

10 \* -----------------------------------------------

11 \* 2020/05/16 V1.0 zh(ryan) 创建

12 \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/

13 #include <stdio.h>

14 #include <stdlib.h>

15 #include <unistd.h>

16 #include <sys/types.h>

17 #include <sys/shm.h>

18 #include <signal.h>

19 #include <string.h>

20

21 int main(int argc, char \*argv[])

22 {

23 int shmid;

24 int key;

25 char \*p;

26

27 key = ftok("./a.c", 'b');

28 if (key < 0) {

29 printf("create key fail\n");

30 return -1;

31 }

32 printf("create key sucess key = 0x%X\n",key);

33

34 shmid = shmget(key, 128, IPC\_CREAT | 0777);

35 if (shmid < 0) {

36 printf("create shared memory fail\n");

37 return -1;

38 }

39 printf("create shared memory sucess, shmid = %d\n", shmid);

40 system("ipcs -m");

41

42 p = (char \*)shmat(shmid, NULL, 0);

43 if (p == NULL) {

44 printf("shmat fail\n");

45 return -1;

46 }

47 printf("shmat sucess\n");

48

49 //write share memory

50 fgets(p, 128, stdin);

51

52 //start read share memory

53 printf("share memory data:%s\n", p);

54

55 //start read share memory again

56 printf("share memory data:%s\n", p);

57

58 //在用户空间删除共享内存的地址

59 shmdt(p);

60

61 memcpy(p, "abcd", 4); //执行这个语句会出现segment fault，因为解除了共享内存地址映射

62 return 0;

63 }

##### JZ2440实验

* 编译

arm-linux-gcc 4th\_shm.c -o 4th\_shm

* 拷贝到NFS文件系统

cp 4th\_shm /work/nfs\_root/first\_fs

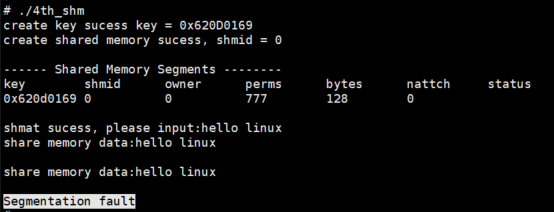
* 运行

我们需要在4th\_shm.c所在的同级目录下创建一个文件a.c（在jz2440开发板上）

touch a.c

运行，此时会提示用户输入信息,输入完之后，执行第61行语句会出现Segmentation fault，这是程序期待的现象。

./4th\_shm



#### shmctl函数

函数原型：int shmctl(int shmid, int cmd, struct shmid\_ds \*buf)

参数 ； shmid : 共享内存标识符

cmd : IPC\_START (获取对象属性) --- 实现了命令 ipcs -m

IPC\_SET (设置对象属性)

IPC\_RMID (删除对象属性) --- 实现了命令 ipcrm -m

buf : 指定IPC\_START/IPC\_SET时用以保存/设置属性

返回值 : 成功0，出错-1

程序示例（参考jz2440\process\_ipc\1st\_shm\5th\_shm.c）

01 /\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

02 \* 功能描述： 1.创建共享内存，将该共享内存地址通过shmat映射到用户地址空间

03 2.用户通过标准输入向这个共享内存中输入一行字符串

04 3.然后从该共享内存中读取内容

05 4.调用shmdt解除地址映射，此时应用程序继续访问会出错

06 5.最后调用shmctl函数删除内核中的共享内存

07 \* 输入参数： 无

08 \* 输出参数： 无

09 \* 返 回 值： 无

10 \* 修改日期 版本号 修改人 修改内容

11 \* -----------------------------------------------

12 \* 2020/05/16 V1.0 zh(ryan) 创建

13 \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/

14

15 #include <stdio.h>

16 #include <stdlib.h>

17 #include <unistd.h>

18 #include <sys/types.h>

19 #include <sys/shm.h>

20 #include <signal.h>

21 #include <string.h>

22

23 int main(int argc, char \*argv[])

24 {

25 int shmid;

26 int key;

27 char \*p;

28

29 key = ftok("./a.c", 'b');

30 if (key < 0) {

31 printf("create key fail\n");

32 return -1;

33 }

34 printf("create key sucess key = 0x%X\n",key);

35

36 shmid = shmget(key, 128, IPC\_CREAT | 0777);

37 if (shmid < 0) {

38 printf("create shared memory fail\n");

39 return -1;

40 }

41 printf("create shared memory sucess, shmid = %d\n", shmid);

42 system("ipcs -m");

43

44 p = (char \*)shmat(shmid, NULL, 0);

45 if (p == NULL) {

46 printf("shmat fail\n");

47 return -1;

48 }

49 printf("shmat sucess\n");

50

51 //write share memory

52 fgets(p, 128, stdin);

53

54 //start read share memory

55 printf("share memory data:%s\n", p);

56

57 //start read share memory again

58 printf("share memory data:%s\n", p);

59

60 //在用户空间删除共享内存的地址

61 shmdt(p);

62

63 //memcpy(p, "abcd", 4); //执行这个语句会出现segment fault

64

65 shmctl(shmid, IPC\_RMID, NULL);

66 system("ipcs -m");

67 return 0;

68 }

##### JZ2440实验

* 编译

arm-linux-gcc 5th\_shm.c -o 5th\_shm

* 拷贝到NFS文件系统

cp 5th\_shm /work/nfs\_root/first\_fs

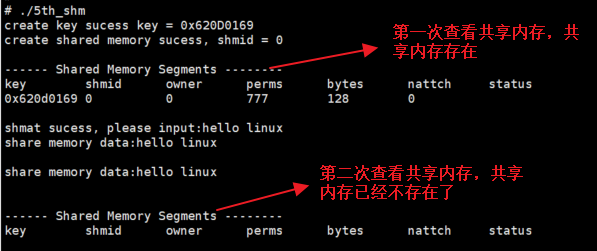
* 运行

我们需要在5th\_shm.c所在的同级目录下创建一个文件a.c（在jz2440开发板上）

touch a.c

运行。此时会提示用户输入信息，第一次执行第42行语句时，读者可以看到共享内存，第二次执行第66行语句时，读者就看不到共享内存了，因为此时共享内存已经被删除了。

./5th\_shm



### 4.1.5共享内存实现进程间通信

步骤：

1. 创建/打开共享内存

2. 映射共享内存，即把指定的共享内存映射到进程的地址空间用于访问

3. 读写共享内存

4. 撤销共享内存映射

5. 删除共享内存对象

使用共享内存时的一些注意点或是限制条件

1. 共享内存的数量是有限制的，通过ipcs -l命令查看，当然如果我们具有管理员权限，可以通过 cat /proc/sys/kernel/shmmax来查看

2. 共享内存删除的时间点，shmctl添加删除标记，只有当所有进程都取消共享内存映射时(即所有进程调用shmdt之后)，才会删除共享内存。

示例源码（参考jz2440\process\_ipc\1st\_shm\6th\_shm.c）

01 /\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

02 \* 功能描述： 1.在父进程中创建使用key值为IPC\_PRIVATE创建一个共享内存

03 2.然后在父进程中创建一个子进程

04 3.通过标准输入，父进程向共享内存中写入字符串

05 4.父进程调用发送信号函数通知子进程可以读取共享内存了

06 5.子进程收到父进程发送过来的信号，开始读取共享内存

07 6.子进程读完共享内存后，发送信号通知父进程读取完成

08 \* 输入参数： 无

09 \* 输出参数： 无

10 \* 返 回 值： 无

11 \* 修改日期 版本号 修改人 修改内容

12 \* -----------------------------------------------

13 \* 2020/05/16 V1.0 zh(ryan) 创建

14 \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/

15

16 #include <stdio.h>

17 #include <stdlib.h>

18 #include <unistd.h>

19 #include <sys/types.h>

20 #include <sys/shm.h>

21 #include <signal.h>

22 #include <string.h>

23

24 void myfun(int signum)

25 {

26 return;

27 }

28

29 int main(int argc, char \*argv[])

30 {

31 int shmid;

32 int key;

33 char \*p;

34 int pid;

35

36

37 shmid = shmget(IPC\_PRIVATE, 128, IPC\_CREAT | 0777);

38 if (shmid < 0) {

39 printf("create shared memory fail\n");

40 return -1;

41 }

42 printf("create shared memory sucess, shmid = %d\n", shmid);

43

44 pid = fork();

45 if (pid > 0) { // 父进程

46 signal(SIGUSR2, myfun);

47 p = (char \*)shmat(shmid, NULL, 0);

48 if (p == NULL) {

49 printf("shmat fail\n");

50 return -1;

51 }

52 printf("parent process shmat sucess\n");

53 while (1) {

54 //从标准输入获取字符串，将其写入到共享内存

55 printf("parent process begin to write memory data:");

56 fgets(p, 128, stdin);

57 kill(pid, SIGUSR1); // 发信号通知子进程读共享内存

58 pause(); // 等待子进程读完共享内存的信号

59 }

60 }

61 if (pid == 0) { // 子进程

62 signal(SIGUSR1, myfun);

63 p = (char \*)shmat(shmid, NULL, 0);

64 if (p == NULL) {

65 printf("shmat fail\n");

66 return -1;

67 }

68 printf("child process shmat sucess\n");

69 while (1) {

70 pause(); // 等待父进程发信号，准备读取共享内存

71 //子进程开始读共享内存，并发信号给父进程告知读完成

72 printf("child process read share memory data:%s\n", p);

73 kill(getppid(), SIGUSR2);

74 }

75 }

76

77 //在用户空间删除共享内存的地址

78 shmdt(p);

79

80 //memcpy(p, "abcd", 4); //执行这个语句会出现segment fault

81

82 shmctl(shmid, IPC\_RMID, NULL);

83 system("ipcs -m");

84 return 0;

85 }

#### JZ2440实验

* 编译

arm-linux-gcc 6th\_shm.c -o 6th\_shm

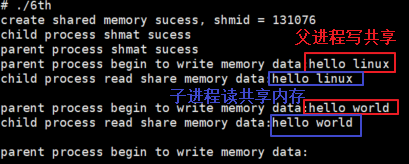
* 拷贝到NFS文件系统

cp 6th\_shm /work/nfs\_root/first\_fs

* 运行

父进程从标准输入获取用户输入的字符串，然后子进程会打印出该字符串。

./6th\_shm



server进程源码（参考jz2440\process\_ipc\1st\_shm\7th\_shm\_1.c）

01 /\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

02 \* 功能描述： 1.server进程使用ftok生成一个key值，利用这个key值创建一个共享内存

03 2.通过标准输入，向共享内存中写入字符串

04 3.server进程调用发送信号函数通知client进程

05 \* 输入参数： 无

06 \* 输出参数： 无

07 \* 返 回 值： 无

08 \* 修改日期 版本号 修改人 修改内容

09 \* -----------------------------------------------

10 \* 2020/05/16 V1.0 zh(ryan) 创建

11 \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/

12

13 #include <stdio.h>

14 #include <stdlib.h>

15 #include <unistd.h>

16 #include <sys/types.h>

17 #include <sys/shm.h>

18 #include <signal.h>

19 #include <string.h>

20

21 struct mybuf

22 {

23 int pid;

24 char buf[124];

25 };

26

27 void myfun(int signum)

28 {

29 return;

30 }

31

32 int main(int argc, char \*argv[])

33 {

34 int shmid;

35 int key;

36 struct mybuf \*p;

37 int pid;

38

39 key = ftok("./a.c", 'a');

40 if (key < 0) {

41 printf("create key fail\n");

42 return -1;

43 }

44 printf("create key sucess\n");

45

46 shmid = shmget(key, 128, IPC\_CREAT | 0777);

47 if (shmid < 0) {

48 printf("create shared memory fail\n");

49 return -1;

50 }

51 printf("create shared memory sucess, shmid = %d\n", shmid);

52

53 signal(SIGUSR2, myfun);

54 p = (struct mybuf \*)shmat(shmid, NULL, 0);

55 if (p == NULL) {

56 printf("shmat fail\n");

57 return -1;

58 }

59 printf("parent process shmat sucess\n");

60

61 p->pid = getpid(); // 将server进程的pid号写入到共享内存

62 pause(); // 等待client读取到server pid号

63 pid=p->pid; // 获取client的进程号

64

65 while (1) {

66 //write share memory

67 printf("parent process begin to write memory data\n");

68 fgets(p->buf, 124, stdin);

69 kill(pid, SIGUSR1); // 向client发送信号通知client读取共享内存数据

70 pause(); // 等待client读取完共享内存数据

71 }

72

73 //在用户空间删除共享内存的地址

74 shmdt(p);

75

76 shmctl(shmid, IPC\_RMID, NULL);

77 system("ipcs -m");

78 return 0;

79 }

client进程源码（参考jz2440\process\_ipc\1st\_shm\7th\_shm\_2.c）

01 /\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

02 \* 功能描述： 1.client进程使用ftok生成一个key值，利用这个key值打开一个共享内存

03 2.client进程收到server进程发送过来的信号之后，开始读取共享内存

04 3.子进程读完共享内存后，发送信号通知父进程读取完成

05 \* 输入参数： 无

06 \* 输出参数： 无

07 \* 返 回 值： 无

08 \* 修改日期 版本号 修改人 修改内容

09 \* -----------------------------------------------

10 \* 2020/05/16 V1.0 zh(ryan) 创建

11 \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/

12

13 #include <stdio.h>

14 #include <stdlib.h>

15 #include <unistd.h>

16 #include <sys/types.h>

17 #include <sys/shm.h>

18 #include <signal.h>

19 #include <string.h>

20

21 struct mybuf

22 {

23 int pid;

24 char buf[124];

25 };

26

27 void myfun(int signum)

28 {

29 return;

30 }

31

32 int main(int argc, char \*argv[])

33 {

34 int shmid;

35 int key;

36 struct mybuf \*p;

37 int pid;

38

39 key = ftok("./a.c", 'a');

40 if (key < 0) {

41 printf("create key fail\n");

42 return -1;

43 }

44 printf("create key sucess\n");

45

46 shmid = shmget(key, 128, IPC\_CREAT | 0777);

47 if (shmid < 0) {

48 printf("create shared memory fail\n");

49 return -1;

50 }

51 printf("create shared memory sucess, shmid = %d\n", shmid);

52

53 signal(SIGUSR1, myfun);

54 p = (struct mybuf \*)shmat(shmid, NULL, 0);

55 if (p == NULL) {

56 printf("shmat fail\n");

57 return -1;

58 }

59 printf("client process shmat sucess\n");

60

61 // get server pid

62 //read share memory

63 pid = p->pid;

64 // write client pid to share memory

65 p->pid = getpid();

66 kill(pid, SIGUSR2); // tell server process to read data

67

68 //client start to read share memory

69

70 while (1) {

71 pause(); // wait server process write share memory

72 printf("client process read data:%s\n", p->buf); // read data

73 kill(pid, SIGUSR2); // server can write share memory

74 }

75

76 //在用户空间删除共享内存的地址

77 shmdt(p);

78

79 shmctl(shmid, IPC\_RMID, NULL);

80 system("ipcs -m");

81 return 0;

82 }

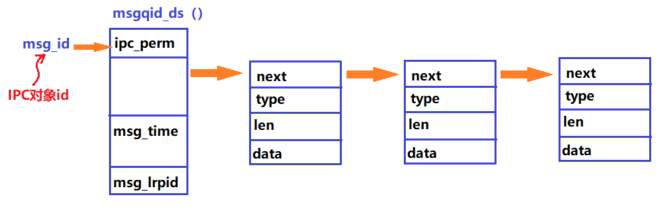
该源码留给读者自行实验，需要注意的是，因为这个时候需要运行两个console，分别运行server进程和client进程，可以采用一个串口console，一个telnet console。我们也可以在ubuntu下开启两个terminal验证。

## 4.2消息队列

### 4.2.1什么是消息队列

消息队列是消息的链表，它是一个链式队列，和管道类似，每个消息多有最大长度限制，可用cat/proc/sys/kernel/msgmax查看。

内核为每个消息队列对象维护了一个数据结构msgqid\_ds，用于标识消息队列，以便让进程知道当前操作的是哪一个消息队列，每一个msqid\_ds表示一个消息队列，并通过msqid\_ds.msg\_first、msg\_last维护一个先进先出的msg链表队列，当发送一个消息到该消息队列时，把发送的消息构造成一个msg的结构对象，并添加到msqid\_ds.msg\_first、msg\_last维护的链表队列。在内核中的表示如下：



### 4.2.2特点

1. 生命周期跟随内核，消息队列一直存在，需要用户显示调用接口删除或者使用命令删除。
2. 消息队列可以实现双向通信
3. 克服了管道只能承载无格式字节流的缺点

### 4.2.3消息队列函数

#### msgget函数

创建或者打开消息队列的函数

头文件：#include <sys/types.h>

#include <sys/ipc.h>

#include <sys/msg.h>

原型： int msgget(key\_t key, int flag)

参数： key 和消息队列关联的key值

flag 消息队列的访问权限

返回值： 成功，消息队列ID，出错 -1

#### msgctl函数

消息队列控制函数

原型： int msgctl(int msgqid, int cmd, struct msqid\_ds \*buf)

参数： msgqid 消息队列ID

cmd IPC\_STAT 读取消息队列的属性，并将其保存在buf指向的缓冲区中

IPC\_SET 设置消息队列的属性，这个值取自buf参数

IPC\_RMID 从系统中删除消息队列

buf 消息缓冲区

返回值： 成功 0，出错 -1

#### msgsnd函数

把一条消息添加到消息队列中

头文件#include <sys/types.h>

#include <sys/ipc.h>

#include <sys/msg.h>

原型： int msgsnd(int msgqid, const void \*msgp, size\_t size, int flag)

参数： msgqid 消息队列ID

msgp 指向消息的指针，常用消息结构msgbuf如下

struct msgbuf {

long mtype; //消息类型

char mtext[N]; //消息正文

};

size 消息正文的字节数

flag IPC\_NOWAIT 消息没有发送完成也会立即返回

0： 直到发送完成函数才会返回

返回值： 成功 0，出错 -1

#### msgrcv函数

从一个消息队列接受消息

原型： int msgrcv(int msgqid, void \*msgp, size\_t size, long msgtype, int flag)

参数： msgqid 消息队列ID

msgp 接收消息的缓冲区

size 要接收消息的字节数

msgtype 0 接收消息队列中第一个消息

大于0 接收消息队列中第一个类型为msgtype的消息

小于0 接收消息队列中类型值不大于msgtype的绝对值且类型值又最小的消息

flag IPC\_NOWAIT 没有消息，会立即返回

0： 若无消息则会一直阻塞

返回值： 成功 接收消息的长度，出错 -1

### 4.2.4消息队列实现进程间通信

server源码（参考jz2440\process\_ipc\2nd\_shm\write\_msg.c）

01 /\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

02 \* 功能描述： 1.server进程向消息队列中写数据

03 \* 输入参数： 无

04 \* 输出参数： 无

05 \* 返 回 值： 无

06 \* 修改日期 版本号 修改人 修改内容

07 \* -----------------------------------------------

08 \* 2020/05/16 V1.0 zh(ryan) 创建

09 \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/

10

11 #include <stdio.h>

12 #include <stdlib.h>

13 #include <unistd.h>

14 #include <sys/types.h>

15 #include <sys/msg.h>

16 #include <signal.h>

17 #include <string.h>

18

19 struct msgbuf {

20 long type; //消息类型

21 char voltage[124]; //消息正文

22 char ID[4];

23 };

24

25 int main(int argc, char \*argv[])

26 {

27 int msgid, readret, key;

28 struct msgbuf sendbuf;

29

30 key = ftok("./a.c", 'a');

31 if (key < 0){

32 printf("create key fail\n");

33 return -1;

34 }

35 msgid = msgget(key, IPC\_CREAT|0777);

36 if (msgid < 0) {

37 printf("create msg queue fail\n");

38 return -1;

39 }

40 printf("create msg queue sucess, msgid = %d\n", msgid);

41 system("ipcs -q");

42

43 // write message queue

44 sendbuf.type = 100;

45 while(1) {

46 memset(sendbuf.voltage, 0, 124); //clear send buffer

47 printf("please input message:");

48 fgets(sendbuf.voltage, 124, stdin);

49 //start write msg to msg queue

50 msgsnd(msgid, (void \*)&sendbuf, strlen(sendbuf.voltage), 0);

51 }

52

53 return 0;

54 }

client源码（参考jz2440\process\_ipc\2nd\_shm\read\_msg.c）

01 /\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

02 \* 功能描述： 1.client进程从消息队列中读数据

03 \* 输入参数： 无

04 \* 输出参数： 无

05 \* 返 回 值： 无

06 \* 修改日期 版本号 修改人 修改内容

07 \* -----------------------------------------------

08 \* 2020/05/16 V1.0 zh(ryan) 创建

09 \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/

10

11 #include <stdio.h>

12 #include <stdlib.h>

13 #include <unistd.h>

14 #include <sys/types.h>

15 #include <sys/msg.h>

16 #include <signal.h>

17 #include <string.h>

18

19 struct msgbuf {

20 long type; //消息类型

21 char voltage[124]; //消息正文

22 char ID[4];

23 };

24

25 int main(int argc, char \*argv[])

26 {

27 int msgid, key;

28 struct msgbuf readbuf;

29

30 key = ftok("./a.c", 'a');

31 if (key < 0){

32 printf("create key fail\n");

33 return -1;

34 }

35 msgid = msgget(key, IPC\_CREAT|0777);

36 if (msgid < 0) {

37 printf("create msg queue fail\n");

38 return -1;

39 }

40 printf("create msg queue sucess, msgid = %d\n", msgid);

41 system("ipcs -q");

42

43 // read message queue

44 while(1) {

45 memset(readbuf.voltage, 0, 124); //clear recv buffer

46 //start read msg to msg queue

47 msgrcv(msgid, (void \*)&readbuf, 124, 100, 0);

48 printf("recv data from message queue:%s", readbuf.voltage);

49 }

50

51 return 0;

52 }

#### JZ2440实验

* 编译

arm-linux-gcc write\_msg.c -o write\_msg

arm-linux-gcc read\_msg.c -o read\_msg

* 拷贝到NFS文件系统

cp write\_msg /work/nfs\_root/first\_fs

cp read\_msg /work/nfs\_root/first\_fs

* 运行

先在后台执行read\_msg，然后在前台运行write\_msg，此时在console下输入字符串，可以看到client进程能读到消息队列中的字符串

./read\_msg &

./ write\_msg



## 4.3信号量灯

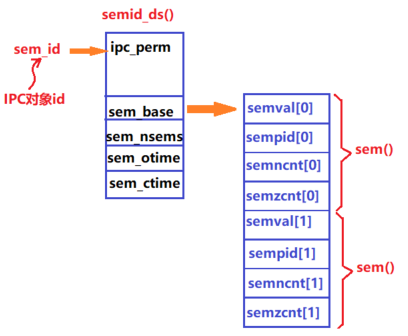
### 4.3.1什么是P、V操作

当不同进程需要访问同一个资源时，由于不同进程的执行次序是未知的，有可能某个进程正在写该资源，而另一个进程正在读该资源，这样会造成进程执行的不确定性。这样的同一个资源，我们称为共享资源，共享资源一次只允许一个进程访问。因此进程在访问共享资源时，需要加上同步、互斥操作。

一般地，P操作表示申请该共享资源，V操作表示释放该共享资源。

### 4.3.2什么是信号量灯

它是信号量的集合，包含多个信号量，可对多个信号灯同时进行P/V操作，主要用来实现进程、线程间同步/互斥。内核为每个信号量灯维护了一个数据结构semid\_ds，用于标识信号量灯，以便进程知道当前操作的是哪个信号量灯，在内核中的表示如下所示。



它和POSIX规范中的信号量有什么区别呢？POSIX规范中的信号量只作用于一个信号量，而IPC对象中的信号量灯会作用于一组信号量。

|  |  |  |
| --- | --- | --- |
| 功能 | 信号量(POSIX) | 信号量灯(IPC对象) |
| 定义信号变量 | sem\_t sem1 | semget |
| 初始化信号量 | sem\_init | semctl |
| P操作 | sem\_wait | semop |
| V操作 | sem\_post | semop |

为什么需要IPC对象中的信号量灯呢？有POSIX规范中的信号量不够吗？

考虑如下场景：

1. 线程A和线程B都需要访问共享资源1和共享资源2，在线程A中会需要先申请共享资源1，然后再申请共享资源2。
2. 但是在线程B中，会先申请贡献资源2，然后再申请共享资源1。
3. 当线程A中开始申请共享资源1时，紧接着会申请共享资源2；而此时线程B中开始申请共享资源2时，紧接着会申请共享资源1。
4. 线程B正在占用着共享资源2，线程A正在占着共享资源1，导致线程B申请不到共享资源1，它就不会释放共享资源2；线程A申请不到共享资源2，它就不会释放共享资源1；这样就造成了死锁。

### 4.3.3信号量灯函数

#### semget函数

创建或者打开函数

头文件#includde <sys/types.h>

#includde <sys/ipc.h>

#includde <sys/sem.h>

原型: int semget(key\_t key, int nsems, int semflag)

参数： key 和信号灯集关联的key值

nsems 信号灯集包含的信号灯数目

semflag 信号灯集的访问权限

返回值： 成功，信号灯ID，出错 -1

#### semctl函数

信号量灯控制函数

头文件#includde <sys/types.h>

#includde <sys/ipc.h>

#includde <sys/sem.h>

原型: int semctl(int semid, int semnum, int cmd, ...union semun arg)

注意最后一个参数不是地址，可以有，可以没有

参数： semid 信号灯集id

semnum 要修改的信号灯集编号,删除操作时，这个值可以设置为任意值

cmd GETVAL 获取信号灯的值

SETVAL 设置信号灯的值

IPC\_RMID 删除信号灯

union semun arg: union semun {

int val; /\* Value for SETVAL \*/

struct semid\_ds \*buf; /\* Buffer for IPC\_STAT, IPC\_SET \*/

unsigned short \*array; /\* Array for GETALL, SETALL \*/

struct seminfo \*\_\_buf; /\* Buffer for IPC\_INFO (Linux-specific) \*/

};

返回值： 成功，消息队列ID，出错 -1

#### semop函数

p/v操作函数

头文件#includde <sys/types.h>

#includde <sys/ipc.h>

#includde <sys/sem.h>

原型: int semop(int semid, struct sembuf \*opsptr, size\_t nops)

参数： semid 信号灯集id

opsptr struct sembuf{

short sem\_num; //要操作信号灯的编号

short sem\_op; //0: 等待，直到信号灯的值变为0，1:资源释放，V操作，-1:分配资源，P操作

short sem\_flg; //0: IPC\_NOWAIT, SEM\_UNDO

}

nops 要操作信号灯个数

返回值： 成功，消息队列ID，出错 -1

### 4.3.4信号量灯实现进程间同步/互斥

程序源码（参考jz2440\process\_ipc\3rd\_shm\share\_sysv.c）

01 /\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

02 \* 功能描述： 1.父进程从键盘输入字符串到共享内存.

03 2.子进程删除字符串中的空格并打印.

04 3.父进程输入quit后删除共享内存和信号灯集，程序结束.

05 \* 输入参数： 无

06 \* 输出参数： 无

07 \* 返 回 值： 无

08 \* 修改日期 版本号 修改人 修改内容

09 \* -----------------------------------------------

10 \* 2020/05/16 V1.0 zh(ryan) 创建

11 \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/

12

13 #include <stdio.h>

14 #include <stdlib.h>

15 #include <string.h>

16 #include <sys/ipc.h>

17 #include <sys/sem.h>

18 #include <sys/types.h>

19 #include <sys/shm.h>

20 #include <signal.h>

21 #include <unistd.h>

22

23 #define N 64

24 #define READ 0

25 #define WRITE 1

26

27 union semun {

28 int val;

29 struct semid\_ds \*buf;

30 unsigned short \*array;

31 struct seminfo \*\_\_buf;

32 };

33

34 void init\_sem(int semid, int s[], int n)

35 {

36 int i;

37 union semun myun;

38

39 for (i = 0; i < n; i++){

40 myun.val = s[i];

41 semctl(semid, i, SETVAL, myun);

42 }

43 }

44

45 void pv(int semid, int num, int op)

46 {

47 struct sembuf buf;

48

49 buf.sem\_num = num;

50 buf.sem\_op = op;

51 buf.sem\_flg = 0;

52 semop(semid, &buf, 1);

53 }

54

55 int main(int argc, char \*argv[])

56 {

57 int shmid, semid, s[] = {0, 1};

58 pid\_t pid;

59 key\_t key;

60 char \*shmaddr;

61

62 key = ftok(".", 's');

63 if (key == -1){

64 perror("ftok");

65 exit(-1);

66 }

67

68 shmid = shmget(key, N, IPC\_CREAT|0666);

69 if (shmid < 0) {

70 perror("shmid");

71 exit(-1);

72 }

73

74 semid = semget(key, 2, IPC\_CREAT|0666);

75 if (semid < 0) {

76 perror("semget");

77 goto \_\_ERROR1;

78 }

79 init\_sem(semid, s, 2);

80

81 shmaddr = shmat(shmid, NULL, 0);

82 if (shmaddr == NULL) {

83 perror("shmaddr");

84 goto \_\_ERROR2;

85 }

86

87 pid = fork();

88 if(pid < 0) {

89 perror("fork");

90 goto \_\_ERROR2;

91 } else if (pid == 0) {

92 char \*p, \*q;

93 while(1) {

94 pv(semid, READ, -1);

95 p = q = shmaddr;

96 while (\*q) {

97 if (\*q != ' ') {

98 \*p++ = \*q;

99 }

100 q++;

101 }

102 \*p = '\0';

103 printf("%s", shmaddr);

104 pv(semid, WRITE, 1);

105 }

106 } else {

107 while (1) {

108 pv(semid, WRITE, -1);

109 printf("input > ");

110 fgets(shmaddr, N, stdin);

111 if (strcmp(shmaddr, "quit\n") == 0) break;

112 pv(semid, READ, 1);

113 }

114 kill(pid, SIGUSR1);

115 }

116

117 \_\_ERROR2:

118 semctl(semid, 0, IPC\_RMID);

119 \_\_ERROR1:

120 shmctl(shmid, IPC\_RMID, NULL);

121 return 0;

122 }

#### JZ2440实验

* 编译

arm-linux-gcc share\_sysv.c -o share\_sysv

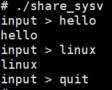
* 拷贝到NFS文件系统

cp share\_sysv /work/nfs\_root/first\_fs

* 运行

在父进程的console下输入字符串，此时子进程会读取到这个字符串。

./share\_sysv

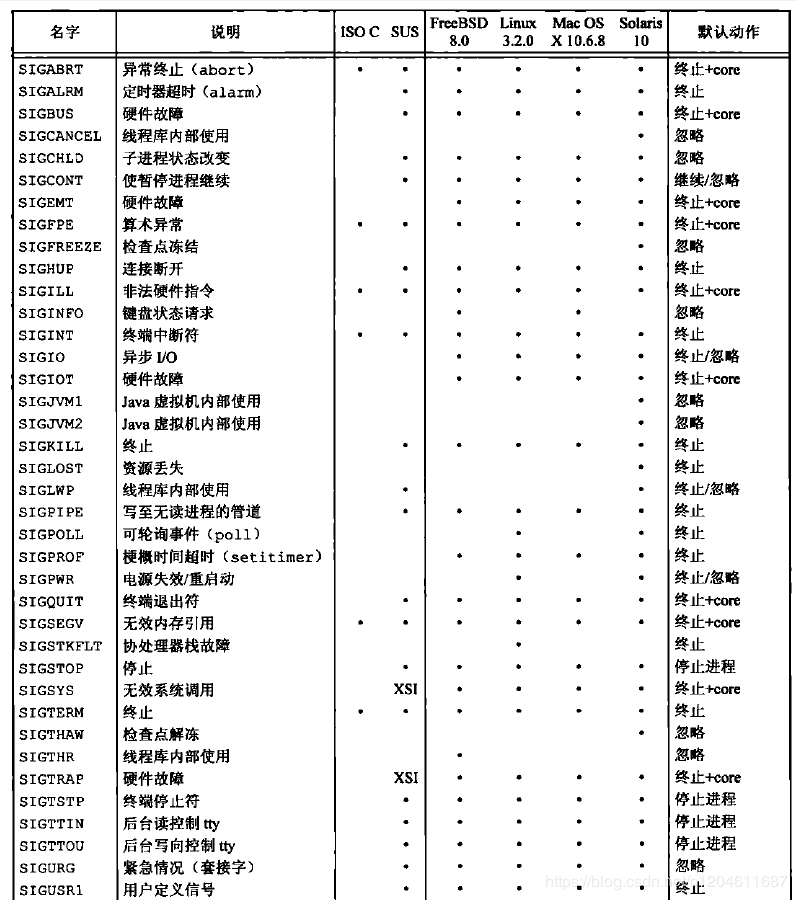


# 进程通信之信号通信

## 5.1信号机制

1. 一般地，中断主要是指硬件中断，比如GPIO中断、定时器中断，这些硬件中断时外设模块工作时，发送给CPU的，也是一种异步方式。
2. 信号是软件层次上对中断机制的一种模拟，是一种异步通信方式。
3. Linux内核通过信号通知用户进程，不同的信号类型代表不同的事件。
4. Linux对早期的unix信号机制进行了扩展。

## 5.2常见信号类型



## 5.3信号发送函数

#### kill函数

头文件 #include <unistd.h>

#include <signal.h>

函数原型int kill(pid\_t pid, int sig);

参数 pid ： 指定接收进程的进程号

0代表同组进程；-1代表所有除了INIT进程和当前进程之外的进程

sig ： 信号类型

返回值 成功返回0，失败返回EOF

#### raise函数

头文件 #include <unistd.h>

#include <signal.h>

函数原型int raise(int sig);

参数 sig ： 信号类型

返回值 成功返回0，失败返回EOF

#### alarm函数

头文件 #include <unistd.h>

#include <signal.h>

函数原型 int alarm(unsigned int seconds);

参数 seconds 定时器的时间

返回值 成功返回上个定时器的剩余时间，失败返回EOF

#### pause函数

进程调用这个函数后悔一直阻塞，直到而被信号中断，功能和sleep类似。

头文件 #include <unistd.h>

#include <signal.h>

函数原型 int pause(void);

返回值 成功返回0，失败返回EOF

#### signal函数

设置信号响应方式，请注意这个函数和kill、killall的区别，我们中文使用者会理解为发信号，实际上它并不是发信号。

头文件 #include <unistd.h>

#include <signal.h>

函数原型 void (\*signal(int signo, void(\*handler)(int)))(int)

参数 signo 要设置的信号类型

handler 指定的信号处理函数；

返回值 成功返回0，失败返回EOF

## 5.4进程捕捉信号

程序源码（参考jz2440\process\_single\send\_single.c）

01 /\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

02 \* 功能描述： 1.捕捉终端发送过来的信号

03 \* 输入参数： 无

04 \* 输出参数： 无

05 \* 返 回 值： 无

06 \* 修改日期 版本号 修改人 修改内容

07 \* -----------------------------------------------

08 \* 2020/05/16 V1.0 zh(ryan) 创建

09 \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/

10

11 #include <stdio.h>

12 #include <stdlib.h>

13 #include <unistd.h>

14 #include <sys/types.h>

15 #include <signal.h>

16

17 void handler(int signo)

18 {

19 switch (signo) {

20 case SIGINT:

21 printf("I have got SIGINT\n");

22 break;

23

24 case SIGQUIT:

25 printf("I have got SIGQUIT\n");

26 break;

27

28 default:

29 printf("don't respond to this signal[%d]\n", signo);

30 exit(0);

31 }

32 }

33

34 int main(int argc, char \*argv[])

35 {

36 signal(SIGINT, handler);

37 signal(SIGQUIT, handler);

38 while (1)

39 pause();

40 return 0;

41 }

#### JZ2440实验

* 编译

arm-linux-gcc send\_single.c -o send\_single

* 拷贝到NFS文件系统

cp send\_single /work/nfs\_root/first\_fs

* 运行

./send\_single

实际上在利用共享内存实现进程间通信时，我们已经使用到了信号通信，父进程写完共享内存后发送信号通知子进程，子进程收到信号后开始读共享内存，这里就不在给出两个进程之间使用信号通信的例子了，请读者参考共享内存中实现两个进程通信的代码。

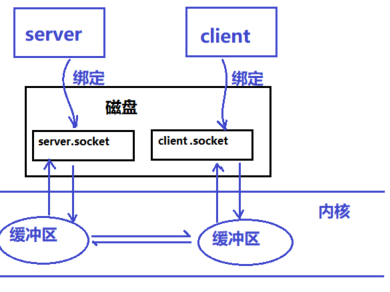
# 进程通信之socket通信

## 6.1什么是socket

先思考一个问题：网络环境中的进程如何实现通信？比如当我们使用QQ和好友聊天的时候，QQ进程是如何与服务器以及你好友所在的QQ进程之间通信的？这些靠的就是socket来实现的。

Socket起源于UNIX，Unix/Linux基本哲学之一就是“一切皆文件”，都可以用“打开open –> 读写write/read –> 关闭close”模式来操作。在《有名管道》那一节中，我们知道socket也是一种文件类型，只不过socket是一种伪文件，存在于内核缓冲区中，大小不变，一直是0。

socket文件一定是成对出现的，server端有一个套接字文件，client端也有一个套接字文件，每个进程需要和对应的套接字文件绑定，每个进程通过读写它的套接字文件，交由内核实现，如下所示。



一般地，socket用来实现网络环境中，不同主机上的进程通信，但是也可以用来在同一个主机上的不同进程之间通信，本小节主要探讨socket用在同一个主机上的不同进程间通信。

## 6.2相关函数

#### socket函数

创建socket文件描述符函数

头文件#include <sys/types.h>

#include <sys/socket.h>

原型: int socket(int domain, int type, int protocol)

返回值： 成功，消息队列ID，出错 -1

#### bind函数

将socket文件描述符和某个socket文件绑定

头文件#include <sys/types.h>

#include <sys/socket.h>

原型: int bind(int sockfd, const struct sockaddr \*addr, socklen\_t addrlen);

参数： sockfd：利用系统调用socket()建立的套接字描述符

addr：代表需要绑定的本地地址信息

addrlen: 本地地址信息长度

返回值： 成功，消息队列ID，出错 -1

#### listen函数

设置监听某个socket文件描述符，设置能够同时和服务端连接的客户端数量，一般只有server会调用

头文件#include <sys/types.h>

#include <sys/socket.h>

原型: int listen(int sockfd, int backlog);

参数： sockfd：利用系统调用socket()建立的套接字描述符

backlog：server可以接受连接的最大client数量

返回值： 成功，消息队列ID，出错 -1

#### accept函数

等待client建立连接的函数，一般只有server会调用

头文件#include <sys/types.h>

#include <sys/socket.h>

原型: int accept(int sockfd,struct sockaddr \*addr,socklen\_t \*addrlen);

参数： sockfd：利用系统调用socket()建立的套接字描述符

addr：指向已经建立连接的对端client地址信息的指针

addrlen: 对端client地址信息长度

返回值： 成功，消息队列ID，出错 -1

#### connet函数

client主动连接server函数，一般只有client才会调用

头文件#include <sys/types.h>

#include <sys/socket.h>

原型: int connect(int sockfd, const struct sockaddr \*addr, socklen\_t addrlen);

参数： sockfd：利用系统调用socket()建立的套接字描述符

addr：指向已经建立连接的对端server地址信息的指针

addrlen: 对端server地址信息长度

返回值： 成功，消息队列ID，出错 -1

#### send函数

发送数据函数

头文件#include <sys/types.h>

#include <sys/socket.h>

原型: ssize\_t send(int sockfd, const void \*buf, size\_t len, int flags);

参数： sockfd：指向要发送数据的socket文件描述符，已经建立连接的文件描述符

buf: 存放要发送数据的缓冲区

len: 实际要发送数据的字节数

flags: 一般为0或者如下的宏

MSG\_DONTROUTE 绕过路由表查找

MSG\_DONTWAIT 仅本操作非阻塞

MSG\_OOB　　　　 发送或接收带外数据

MSG\_PEEK　　 窥看外来消息

MSG\_WAITALL　　 等待所有数据

返回值： 成功，消息队列ID，出错 -1

#### recv函数

接收数据函数

头文件#include <sys/types.h>

#include <sys/socket.h>

原型: ssize\_t recv(int sockfd, void \*buf, size\_t len, int flags);

参数： sockfd：已经建立连接的文件描述符

buf: 存放要接收数据的缓冲区

len: 实际要接收数据的字节数

flags:一般为0或者如下的宏

MSG\_DONTROUTE 绕过路由表查找

MSG\_DONTWAIT 仅本操作非阻塞

MSG\_OOB　　　　 发送或接收带外数据

MSG\_PEEK　　 窥看外来消息

MSG\_WAITALL　　 等待所有数据

返回值： 成功，消息队列ID，出错 -1

## 6.3 socket实现进程间通信

程序实现一般步骤

Server端

1.创建socket

2.绑定socket

3.设置监听

4.等待客户端连接

5.发送/接收数据

Client端

1.创建socket

2.绑定socket

3.连接

4.发送/接收数据

server源码（参考jz2440\process\_socket\server.c）

01 /\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

02 \* 功能描述： 1.server打印client发送过来的字符串，并将该字符串回发给client

03 \* 输入参数： 无

04 \* 输出参数： 无

05 \* 返 回 值： 无

06 \* 修改日期 版本号 修改人 修改内容

07 \* -----------------------------------------------

08 \* 2020/05/16 V1.0 zh(ryan) 创建

09 \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/

10

11 #include <stdio.h>

12 #include <stdlib.h>

13 #include <unistd.h>

14 #include <sys/types.h>

15 #include <sys/stat.h>

16 #include <string.h>

17 #include <arpa/inet.h>

18 #include <sys/un.h>

19

20 int main(int argc, char \*argv[])

21 {

22 int lfd ,ret, cfd;

23 struct sockaddr\_un serv, client;

24 socklen\_t len = sizeof(client);

25 char buf[1024] = {0};

26 int recvlen;

27

28 //创建socket

29 lfd = socket(AF\_LOCAL, SOCK\_STREAM, 0);

30 if (lfd == -1) {

31 perror("socket error");

32 return -1;

33 }

34

35 //如果套接字文件存在，删除套接字文件

36 unlink("server.sock");

37

38 //初始化server信息

39 serv.sun\_family = AF\_LOCAL;

40 strcpy(serv.sun\_path, "server.sock");

41

42 //绑定

43 ret = bind(lfd, (struct sockaddr \*)&serv, sizeof(serv));

44 if (ret == -1) {

45 perror("bind error");

46 return -1;

47 }

48

49 //设置监听，设置能够同时和服务端连接的客户端数量

50 ret = listen(lfd, 36);

51 if (ret == -1) {

52 perror("listen error");

53 return -1;

54 }

55

56 //等待客户端连接

57 cfd = accept(lfd, (struct sockaddr \*)&client, &len);

58 if (cfd == -1) {

59 perror("accept error");

60 return -1;

61 }

62 printf("=====client bind file:%s\n", client.sun\_path);

63

64 while (1) {

65 recvlen = recv(cfd, buf, sizeof(buf), 0);

66 if (recvlen == -1) {

67 perror("recv error");

68 return -1;

69 } else if (recvlen == 0) {

70 printf("client disconnet...\n");

71 close(cfd);

72 break;

73 } else {

74 printf("server recv buf: %s\n", buf);

75 send(cfd, buf, recvlen, 0);

76 }

77 }

78

79 close(cfd);

80 close(lfd);

81 return 0;

82 }

client源码（参考jz2440\process\_socket\client.c）

01 /\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

02 \* 功能描述： 1.client从标准输入获取到一个字符串，然后将这个字符串发送给server

03 \* 输入参数： 无

04 \* 输出参数： 无

05 \* 返 回 值： 无

06 \* 修改日期 版本号 修改人 修改内容

07 \* -----------------------------------------------

08 \* 2020/05/16 V1.0 zh(ryan) 创建

09 \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/

10

11 #include <stdio.h>

12 #include <stdlib.h>

13 #include <unistd.h>

14 #include <sys/types.h>

15 #include <sys/stat.h>

16 #include <string.h>

17 #include <arpa/inet.h>

18 #include <sys/un.h>

19

20 int main(int argc, char \*argv[])

21 {

22 int lfd ,ret;

23 struct sockaddr\_un serv, client;

24 socklen\_t len = sizeof(client);

25 char buf[1024] = {0};

26 int recvlen;

27

28 //创建socket

29 lfd = socket(AF\_LOCAL, SOCK\_STREAM, 0);

30 if (lfd == -1) {

31 perror("socket error");

32 return -1;

33 }

34

35 //如果套接字文件存在，删除套接字文件

36 unlink("client.sock");

37

38 //给客户端绑定一个套接字文件

39 client.sun\_family = AF\_LOCAL;

40 strcpy(client.sun\_path, "client.sock");

41 ret = bind(lfd, (struct sockaddr \*)&client, sizeof(client));

42 if (ret == -1) {

43 perror("bind error");

44 return -1;

45 }

46

47 //初始化server信息

48 serv.sun\_family = AF\_LOCAL;

49 strcpy(serv.sun\_path, "server.sock");

50 //连接

51 connect(lfd, (struct sockaddr \*)&serv, sizeof(serv));

52

53 while (1) {

54 fgets(buf, sizeof(buf), stdin);

55 send(lfd, buf, strlen(buf)+1, 0);

56

57 recv(lfd, buf, sizeof(buf), 0);

58 printf("client recv buf: %s\n", buf);

59 }

60

61 close(lfd);

62 return 0;

63 }

#### JZ2440实验

* 编译

arm-linux-gcc server.c -o server

arm-linux-gcc client.c -o client

* 拷贝到NFS文件系统

cp server /work/nfs\_root/first\_fs

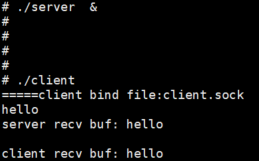
cp client /work/nfs\_root/first\_fs

* 运行

为方便看程序运行结果，server在后台执行；client在前台运行，client能够接收来自终端的输入。

./server &

./client



## 6.4一个server和多个client之间的通信

在实际项目开发中， 更常见的一种场景是：一个server要和多个client之间通信，这部分实现方式交给读者自行实现，实现的方式有很多，比如如下两种方式，当然还要其他方法。

1. 多进程实现，一个主进程用来实现检测client的连接，每检测一次client连接，则为这个client创建一个专门的进程，用于实现两者间通信。
2. 也可以使用多线程实现，一个主线程用来检测client的连接，每检测一次client连接，则为这个client创建一个专门的线程，用于实现两者间通信。