

8. 进程间通信

本节重点

- 进程间通信介绍
- 掌握匿名命名管道原理操作
- 编写进程池
- 掌握共享内存
- 了解消息队列
- 了解信号量
- 理解内核管理IPC资源的方式，了解C实现多态

1. 进程间通信介绍

1-1 进程间通信目的

- 数据传输：一个进程需要将它的数据发送给另一个进程
- 资源共享：多个进程之间共享同样的资源。
- 通知事件：一个进程需要向另一个或一组进程发送消息，通知它（它们）发生了某种事件（如进程终止时要通知父进程）。
- 进程控制：有些进程希望完全控制另一个进程的执行（如Debug进程），此时控制进程希望能够拦截另一个进程的所有陷入和异常，并能够及时知道它的状态改变。

1-2 进程间通信发展

- 管道
- System V进程间通信
- POSIX进程间通信

1-3 进程间通信分类

管道

- 匿名管道pipe
- 命名管道

System V IPC

- System V 消息队列
- System V 共享内存
- System V 信号量

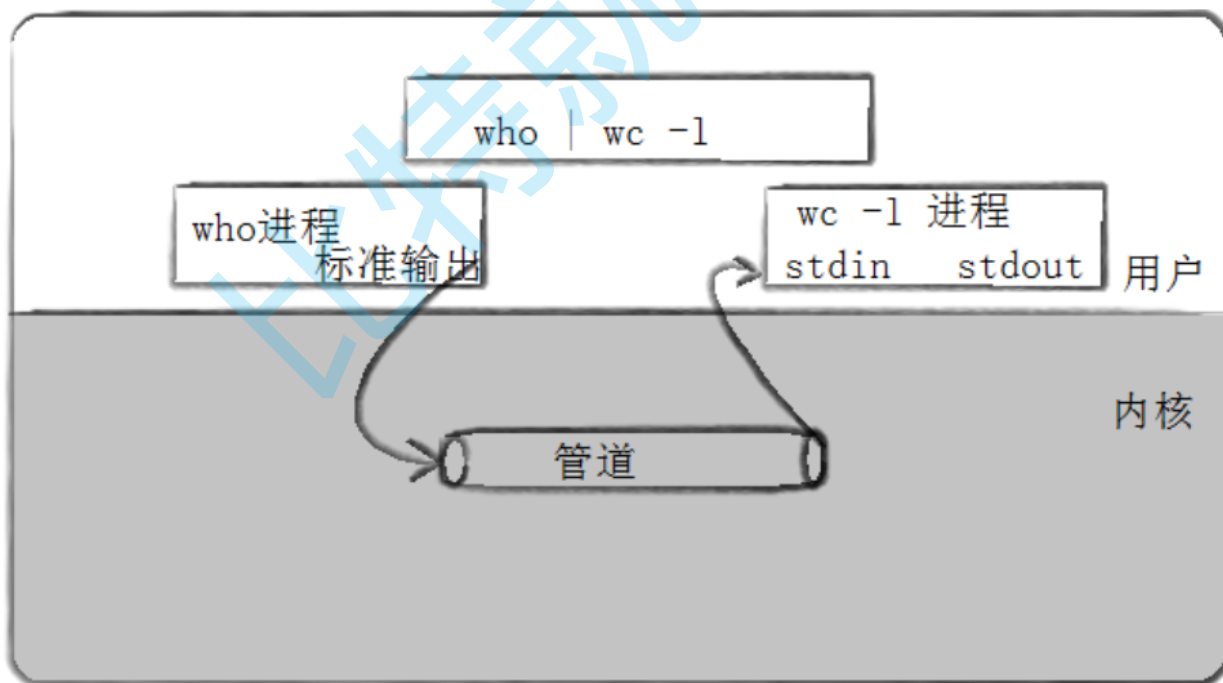
POSIX IPC

- 消息队列
- 共享内存
- 信号量
- 互斥量
- 条件变量
- 读写锁

2. 管道

什么是管道

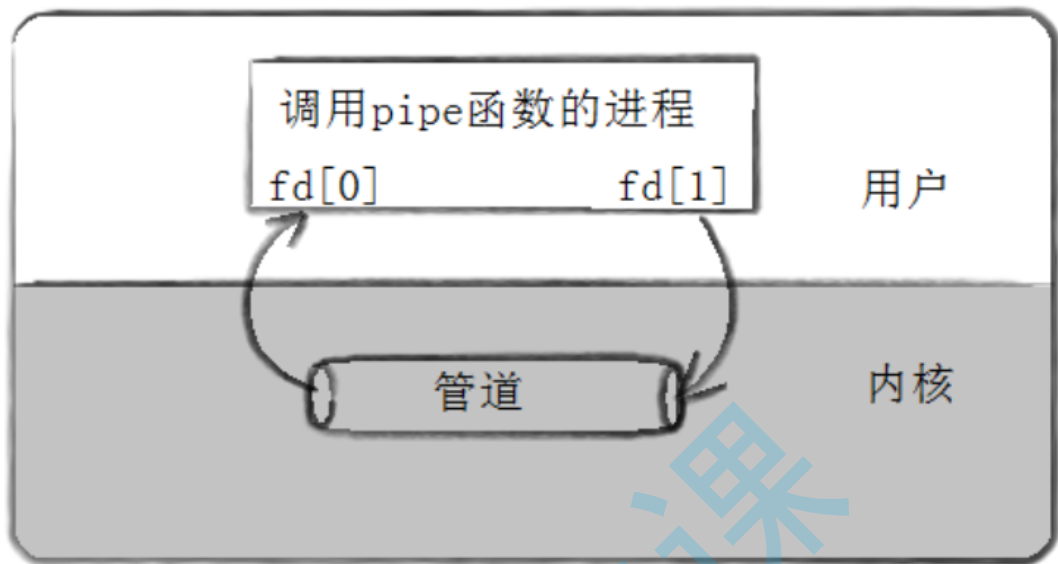
- 管道是Unix中最古老的进程间通信的形式。
- 我们把从一个进程连接到另一个进程的一个数据流称为一个“管道”



3. 匿名管道

- 1 `#include <unistd.h>`
- 2 功能: 创建一无名管道
- 3 原型

```
4 int pipe(int fd[2]);
5 参数
6 fd: 文件描述符数组,其中fd[0]表示读端, fd[1]表示写端
7 返回值:成功返回0, 失败返回错误代码
```



3-1 实例代码

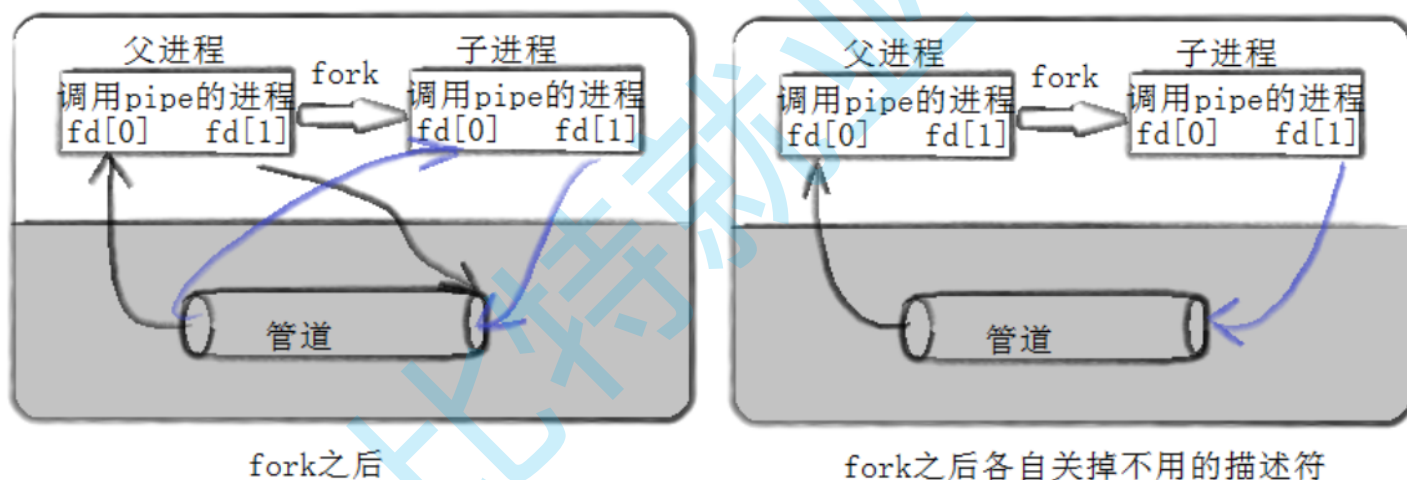
```
1 例子：从键盘读取数据，写入管道，读取管道，写到屏幕
2
3 #include <stdio.h>
4 #include <stdlib.h>
5 #include <string.h>
6 #include <unistd.h>
7
8 int main( void )
9 {
10     int fds[2];
11     char buf[100];
12     int len;
13
14     if ( pipe(fds) == -1 )
15         perror("make pipe"),exit(1);
16
17     // read from stdin
18     while ( fgets(buf, 100, stdin) ) {
19         len = strlen(buf);
20         // write into pipe
21         if ( write(fds[1], buf, len) != len ) {
22             perror("write to pipe");
23             break;
```

```

24     }
25     memset(buf, 0x00, sizeof(buf));
26
27     // read from pipe
28     if ( (len=read(fds[0], buf, 100)) == -1 ) {
29         perror("read from pipe");
30         break;
31     }
32
33     // write to stdout
34     if ( write(1, buf, len) != len ) {
35         perror("write to stdout");
36         break;
37     }
38 }
39 }

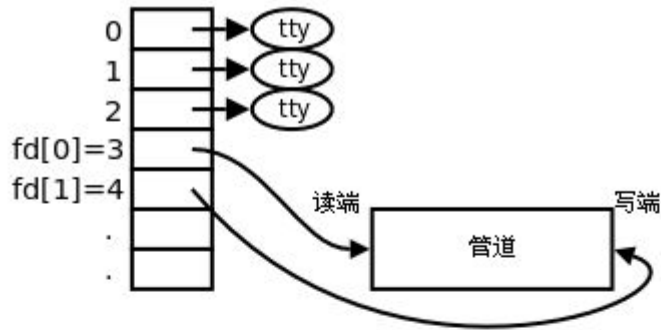
```

3-2 用 fork 来共享管道原理

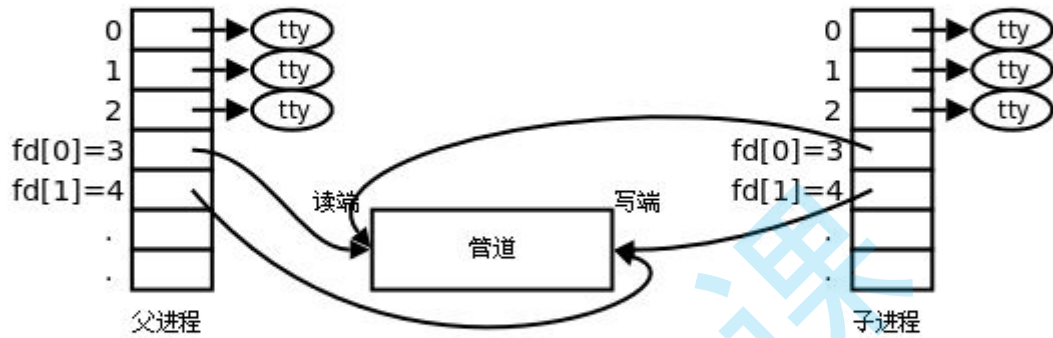


3-3 站在文件描述符角度-深度理解管道

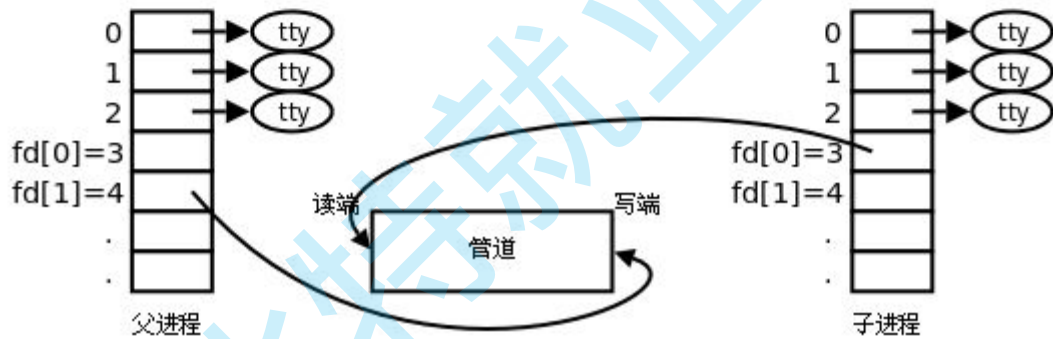
1. 父进程创建管道



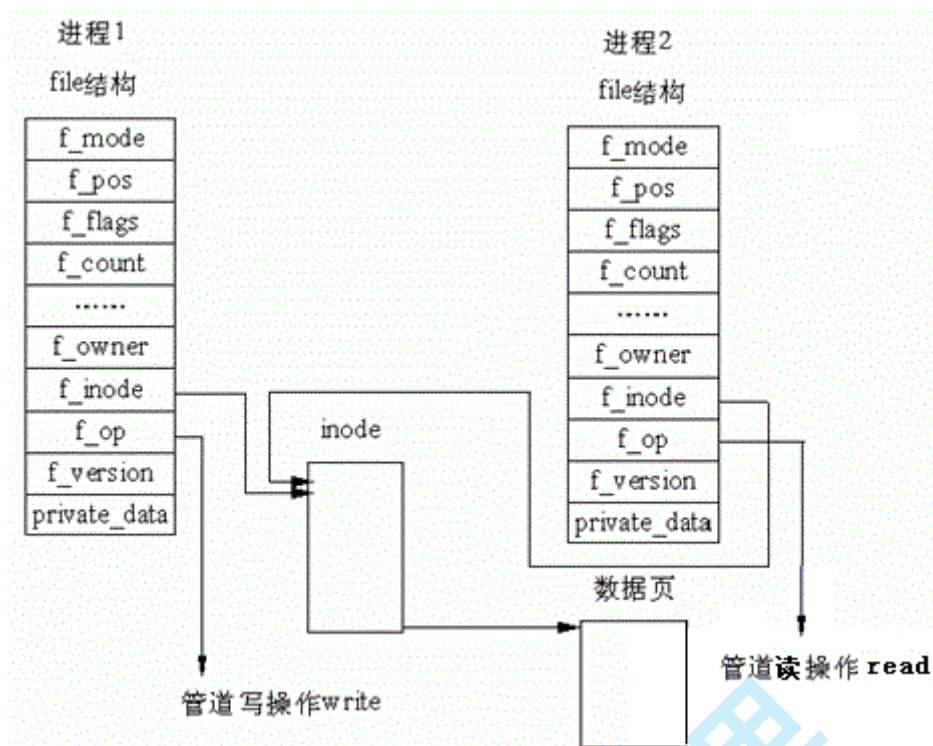
2. 父进程 fork 出子进程



3. 父进程关闭 fd[0]，子进程关闭 fd[1]



3-4 站在内核角度-管道本质



- 所以，看待管道，就如同看待文件一样！管道的使用和文件一致，迎合了“Linux一切皆文件思想”。

3-5 管道样例

3-5-1 测试管道读写

```

1  #include <unistd.h>
2  #include <stdlib.h>
3  #include <stdio.h>
4  #include <errno.h>
5  #include <string.h>
6  #define ERR_EXIT(m) \
7  do \
8  { \
9      perror(m); \
10     exit(EXIT_FAILURE); \
11 } while(0)
12
13
14 int main(int argc, char *argv[])
15 {
16     int pipefd[2];
17     if (pipe(pipefd) == -1)
18         ERR_EXIT("pipe error");
19
20     pid_t pid;
21     pid = fork();

```

```

22     if (pid == -1)
23         ERR_EXIT("fork error");
24     if (pid == 0) {
25         close(pipefd[0]);
26         write(pipefd[1], "hello", 5);
27         close(pipefd[1]);
28         exit(EXIT_SUCCESS);
29     }
30
31     close(pipefd[1]);
32     char buf[10] = {0};
33     read(pipefd[0], buf, 10);
34     printf("buf=%s\n", buf);
35
36     return 0;
37 }

```

3-5-2 创建进程池处理任务

Channel.hpp

```

1  #ifndef __CHANNEL_HPP__
2  #define __CHANNEL_HPP__
3
4  #include <iostream>
5  #include <string>
6  #include <unistd.h>
7
8  // 先描述
9  class Channel
10 {
11 public:
12     Channel(int wfd, pid_t who) : _wfd(wfd), _who(who)
13     {
14         // Channel-3-1234
15         _name = "Channel-" + std::to_string(wfd) + "-" + std::to_string(who);
16     }
17     std::string Name()
18     {
19         return _name;
20     }
21     void Send(int cmd)
22     {
23         ::write(_wfd, &cmd, sizeof(cmd));
24     }
25     void Close()

```

```

26     {
27         ::close(_wfd);
28     }
29     pid_t Id()
30     {
31         return _who;
32     }
33     int wFd()
34     {
35         return _wfd;
36     }
37     ~Channel()
38     {
39     }
40
41 private:
42     int _wfd;
43     std::string _name;
44     pid_t _who;
45 };
46
47 #endif

```

ProcessPool.hpp

```

1  #ifndef __PROCESS_POOL_HPP__
2  #define __PROCESS_POOL_HPP__
3
4  #include <iostream>
5  #include <string>
6  #include <vector>
7  #include <cstdlib>
8  #include <unistd.h>
9  #include <sys/types.h>
10 #include <sys/wait.h>
11 #include <functional>
12 #include "Task.hpp"
13 #include "Channel.hpp"
14
15 // typedef std::function<void()> work_t;
16 using work_t = std::function<void()>;
17
18 enum
19 {
20     OK = 0,

```



```

21     UsageError,
22     PipeError,
23     ForkError
24 };
25
26 class ProcessPool
27 {
28 public:
29     ProcessPool(int n, work_t w)
30         : processnum(n), work(w)
31     {
32     }
33     // channels : 输出型参数
34     // work_t work: 回调
35     int InitProcessPool()
36     {
37         // 2. 创建指定个数个进程
38         for (int i = 0; i < processnum; i++)
39         {
40             // 1. 先有管道
41             int pipefd[2] = {0};
42             int n = pipe(pipefd);
43             if (n < 0)
44                 return PipeError;
45             // 2. 创建进程
46             pid_t id = fork();
47             if (id < 0)
48                 return ForkError;
49
50             // 3. 建立通信信道
51             if (id == 0)
52             {
53                 // 关闭历史wfd
54                 std::cout << getpid() << ", child close history fd: ";
55                 for(auto &c : channels)
56                 {
57                     std::cout << c.wFd() << " ";
58                     c.Close();
59                 }
60                 std::cout << " over" << std::endl;
61
62                 ::close(pipefd[1]); // read
63                 // child
64                 std::cout << "debug: " << pipefd[0] << std::endl;
65                 dup2(pipefd[0], 0);
66                 work();
67                 ::exit(0);

```

```

68         }
69
70         // 父进程执行
71         ::close(pipefd[0]); // write
72         channels.emplace_back(pipefd[1], id);
73         // Channel ch(pipefd[1], id);
74         // channels.push_back(ch);
75     }
76
77     return OK;
78 }
79
80 void DispatchTask()
81 {
82     int who = 0;
83     // 2. 派发任务
84     int num = 20;
85     while (num-->0)
86     {
87         // a. 选择一个任务, 整数
88         int task = tm.SelectTask();
89         // b. 选择一个子进程channel
90         Channel &curr = channels[who++];
91         who %= channels.size();
92
93         std::cout << "#####" << std::endl;
94         std::cout << "send " << task << " to " << curr.Name() << ", 任务还
剩: " << num << std::endl;
95         std::cout << "#####" << std::endl;
96
97         // c. 派发任务
98         curr.Send(task);
99
100        sleep(1);
101    }
102 }
103
104 void CleanProcessPool()
105 {
106     // version 3
107     for (auto &c : channels)
108     {
109         c.Close();
110         pid_t rid = ::waitpid(c.Id(), nullptr, 0);
111         if (rid > 0)
112         {

```

```

113         std::cout << "child " << rid << " wait ... success" <<
std::endl;
114     }
115 }
116
117     // version 2
118     // for (auto &c : channels)
119     // for(int i = channels.size()-1; i >= 0; i--)
120     // {
121     //     channels[i].Close();
122     //     pid_t rid = ::waitpid(channels[i].Id(), nullptr, 0); // 阻塞了!
123     //     if (rid > 0)
124     //     {
125     //         std::cout << "child " << rid << " wait ... success" <<
std::endl;
126     //     }
127     // }
128
129     // version 1
130     // for (auto &c : channels)
131     // {
132     //     c.Close();
133     // }
134     //?
135     // for (auto &c : channels)
136     // {
137     //     pid_t rid = ::waitpid(c.Id(), nullptr, 0);
138     //     if (rid > 0)
139     //     {
140     //         std::cout << "child " << rid << " wait ... success" <<
std::endl;
141     //     }
142     // }
143 }
144
145 void DebugPrint()
146 {
147     for (auto &c : channels)
148     {
149         std::cout << c.Name() << std::endl;
150     }
151 }
152
153 private:
154     std::vector<Channel> channels;
155     int processnum;
156     work_t work;

```

```
157 };
158 #endif
```

Task.hpp

```
1  #pragma once
2
3  #include <iostream>
4  #include <unordered_map>
5  #include <functional>
6  #include <ctime>
7  #include <sys/types.h>
8  #include <unistd.h>
9
10 using task_t = std::function<void()>;
11
12 class TaskManger
13 {
14 public:
15     TaskManger()
16     {
17         srand(time(nullptr));
18         tasks.push_back([]()
19             { std::cout << "sub process[" << getpid() << " ] 执行访问数据库的任
19 务\n"
20             << std::endl; });
21         tasks.push_back([]()
22             { std::cout << "sub process[" << getpid() << " ] 执行url解析\n"
23             << std::endl; });
24         tasks.push_back([]()
25             { std::cout << "sub process[" << getpid() << " ] 执行加密任务\n"
26             << std::endl; });
27         tasks.push_back([]()
28             { std::cout << "sub process[" << getpid() << " ] 执行数据持久化任务
29             \n"
30             << std::endl; });
31     }
32     int SelectTask()
33     {
34         return rand() % tasks.size();
35     }
36     void Excute(unsigned long number)
37     {
38         if (number > tasks.size() || number < 0)
```

```

39     tasks[number]();
40 }
41 ~TaskManger()
42 {
43 }
44
45 private:
46     std::vector<task_t> tasks;
47 };
48
49 TaskManger tm;
50
51 void Worker()
52 {
53     while (true)
54     {
55         int cmd = 0;
56         int n = ::read(0, &cmd, sizeof(cmd));
57         if (n == sizeof(cmd))
58         {
59             tm.Excute(cmd);
60         }
61         else if (n == 0)
62         {
63             std::cout << "pid: " << getpid() << " quit..." << std::endl;
64             break;
65         }
66         else
67         {
68         }
69     }
70 }

```

Main.cc

```

1  #include "ProcessPool.hpp"
2  #include "Task.hpp"
3  void Usage(std::string proc)
4  {
5      std::cout << "Usage: " << proc << " process-num" << std::endl;
6  }
7
8  // 我们自己就是master
9  int main(int argc, char *argv[])
10 {

```

```

11     if (argc != 2)
12     {
13         Usage(argv[0]);
14         return UsageError;
15     }
16     int num = std::stoi(argv[1]);
17     ProcessPool *pp = new ProcessPool(num, Worker);
18     // 1. 初始化进程池
19     pp->InitProcessPool();
20     // 2. 派发任务
21     pp->DispatchTask();
22     // 3. 退出进程池
23     pp->CleanProcessPool();
24
25     // std::vector<Channel> channels;
26     // // 1. 初始化进程池
27     // InitProcessPool(num, channels, Worker);
28
29     // // 2. 派发任务
30     // DispatchTask(channels);
31
32     // // 3. 退出进程池
33     // CleanProcessPool(channels);
34
35     delete pp;
36     return 0;
37 }
38

```

Makefile

```

1  BIN=processpool
2  CC=g++
3  FLAGS=-c -Wall -std=c++11
4  LDFLAGS=-o
5  # SRC=$(shell ls *.cc)
6  SRC=$(wildcard *.cc)
7  OBJ=$(SRC:.cc=.o)
8
9  $(BIN):$(OBJ)
10     $(CC) $(LDFLAGS) $@ $^
11 %.o:%.cc
12     $(CC) $(FLAGS) $<
13
14 .PHONY:clean

```

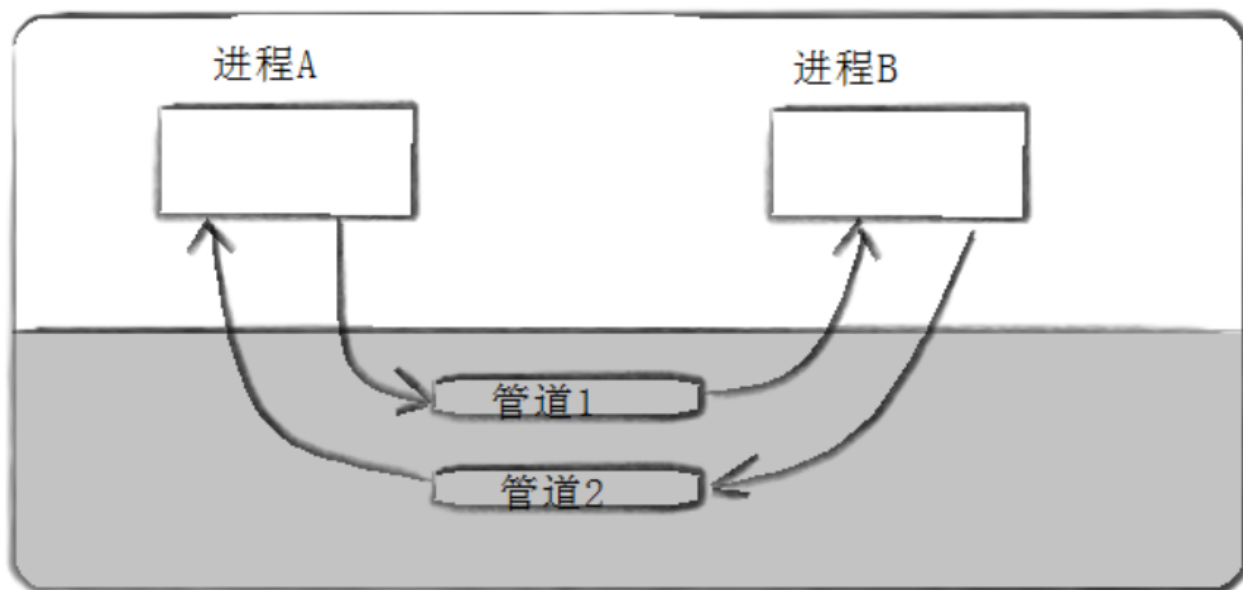
```
15 clean:
16         rm -f $(BIN) $(OBJ)
17
18 .PHONY:test
19 test:
20         @echo $(SRC)
21         @echo $(OBJ)
```

3-6 管道读写规则

- 当没有数据可读时
 - O_NONBLOCK disable: read调用阻塞, 即进程暂停执行, 一直等到有数据来到为止。
 - O_NONBLOCK enable: read调用返回-1, errno值为EAGAIN。
- 当管道满的时候
 - O_NONBLOCK disable: write调用阻塞, 直到有进程读走数据
 - O_NONBLOCK enable: 调用返回-1, errno值为EAGAIN
- 如果所有管道写端对应的文件描述符被关闭, 则read返回0
- 如果所有管道读端对应的文件描述符被关闭, 则write操作会产生信号SIGPIPE, 进而可能导致write进程退出
- 当要写入的数据量不大于PIPE_BUF时, linux将保证写入的原子性。
- 当要写入的数据量大于PIPE_BUF时, linux将不再保证写入的原子性。

3-7 管道特点

- 只能用于具有共同祖先的进程 (具有亲缘关系的进程) 之间进行通信; 通常, 一个管道由一个进程创建, 然后该进程调用fork, 此后父、子进程之间就可应用该管道。
- 管道提供流式服务
- 一般而言, 进程退出, 管道释放, 所以管道的生命周期随进程
- 一般而言, 内核会对管道操作进行同步与互斥
- 管道是半双工的, 数据只能向一个方向流动; 需要双方通信时, 需要建立起两个管道



利用两个管道实现双向通信

3-8 验证管道通信的4种情况

- 读正常&&写满
- 写正常&&读空
- 写关闭&&读正常
- 读关闭&&写正常

4. 命名管道

- 管道应用的一个限制就是只能在具有共同祖先（具有亲缘关系）的进程间通信。
- 如果我们想在不相关的进程之间交换数据，可以使用FIFO文件来做这项工作，它经常被称为命名管道。
- 命名管道是一种特殊类型的文件

4-1 创建一个命名管道

- 命名管道可以从命令行上创建，命令行方法是使用下面这个命令：

```
1 $ mkfifo filename
```

- 命名管道也可以从程序里创建，相关函数有：

```
1 int mkfifo(const char *filename, mode_t mode);
```


创建命名管道:

```
1 int main(int argc, char *argv[])
2 {
3     mkfifo("p2", 0644);
4     return 0;
5 }
```

4-2 匿名管道与命名管道的区别

- 匿名管道由pipe函数创建并打开。
- 命名管道由mkfifo函数创建，打开用open
- FIFO（命名管道）与pipe（匿名管道）之间唯一的区别在它们创建与打开的方式不同，一但这些工作完成之后，它们具有相同的语义。

4-3 命名管道的打开规则

- 如果当前打开操作是为读而打开FIFO时
 - O_NONBLOCK disable：阻塞直到有相应进程为写而打开该FIFO
 - O_NONBLOCK enable：立刻返回成功
- 如果当前打开操作是为写而打开FIFO时
 - O_NONBLOCK disable：阻塞直到有相应进程为读而打开该FIFO
 - O_NONBLOCK enable：立刻返回失败，错误码为ENXIO

实例1. 用命名管道实现文件拷贝

读取文件，写入命名管道:

```
1 #include <unistd.h>
2 #include <stdlib.h>
3 #include <stdio.h>
4 #include <errno.h>
5 #include <string.h>
6
7 #define ERR_EXIT(m) \
8 do \
9 { \
10     perror(m); \
11     exit(EXIT_FAILURE); \
12 } while(0)
13
```

```

14 int main(int argc, char *argv[])
15 {
16     mkfifo("tp", 0644);
17     int infd;
18     infd = open("abc", O_RDONLY);
19     if (infd == -1) ERR_EXIT("open");
20
21     int outfd;
22     outfd = open("tp", O_WRONLY);
23     if (outfd == -1) ERR_EXIT("open");
24     char buf[1024];
25     int n;
26     while ((n=read(infd, buf, 1024))>0)
27     {
28         write(outfd, buf, n);
29     }
30     close(infd);
31     close(outfd);
32     return 0;
33 }

```

读取管道，写入目标文件:

```

1 #include <unistd.h>
2 #include <stdlib.h>
3 #include <stdio.h>
4 #include <errno.h>
5 #include <string.h>
6
7 #define ERR_EXIT(m) \
8     do \
9     { \
10         perror(m); \
11         exit(EXIT_FAILURE); \
12     } while(0)
13
14 int main(int argc, char *argv[])
15 {
16     int outfd;
17     outfd = open("abc.bak", O_WRONLY | O_CREAT | O_TRUNC, 0644);
18     if (outfd == -1) ERR_EXIT("open");
19
20     int infd;
21     infd = open("tp", O_RDONLY);
22     if (outfd == -1)

```

```

23     ERR_EXIT("open");
24     char buf[1024];
25     int n;
26     while ((n=read(infd, buf, 1024))>0)
27     {
28         write(outfd, buf, n);
29     }
30     close(infd);
31     close(outfd);
32     unlink("tp");
33     return 0;
34 }

```

实例2. 用命名管道实现server&client通信

```

1 # ll
2 total 12
3 -rw-r--r--. 1 root root 46 Sep 18 22:37 clientPipe.c
4 -rw-r--r--. 1 root root 164 Sep 18 22:37 Makefile
5 -rw-r--r--. 1 root root 46 Sep 18 22:38 serverPipe.c
6
7 # cat Makefile
8 .PHONY:all
9 all:clientPipe serverPipe
10
11 clientPipe:clientPipe.c
12     gcc -o $@ $^
13 serverPipe:serverPipe.c
14     gcc -o $@ $^
15
16 .PHONY:clean
17 clean:
18     rm -f clientPipe serverPipe

```

serverPipe.c

```

1 #include <stdio.h>
2 #include <sys/types.h>
3 #include <sys/stat.h>
4 #include <fcntl.h>
5 #include <unistd.h>
6 #include <stdlib.h>
7

```

```

8 #define ERR_EXIT(m) \
9 do{\
10     perror(m);\
11     exit(EXIT_FAILURE);\
12 }while(0)
13
14 int main()
15 {
16     umask(0);
17     if(mkfifo("mypipe", 0644) < 0){
18         ERR_EXIT("mkfifo");
19     }
20     int rfd = open("mypipe", O_RDONLY);
21     if(rfd < 0){
22         ERR_EXIT("open");
23     }
24
25     char buf[1024];
26     while(1){
27         buf[0] = 0;
28         printf("Please wait...\n");
29         ssize_t s = read(rfd, buf, sizeof(buf)-1);
30         if(s > 0 ){
31             buf[s-1] = 0;
32             printf("client say# %s\n", buf);
33         }else if(s == 0){
34             printf("client quit, exit now!\n");
35             exit(EXIT_SUCCESS);
36         }else{
37             ERR_EXIT("read");
38         }
39     }
40
41     close(rfd);
42     return 0;
43 }

```

clientPipe.c

```

1 #include <stdio.h>
2 #include <sys/types.h>
3 #include <sys/stat.h>
4 #include <fcntl.h>
5 #include <unistd.h>
6 #include <stdlib.h>

```

```

7  #include <string.h>
8
9  #define ERR_EXIT(m) \
10 do{\
11     perror(m);\
12     exit(EXIT_FAILURE);\
13 }while(0)
14
15 int main()
16 {
17     int wfd = open("mypipe", O_WRONLY);
18     if(wfd < 0){
19         ERR_EXIT("open");
20     }
21
22     char buf[1024];
23     while(1){
24         buf[0] = 0;
25         printf("Please Enter# ");
26         fflush(stdout);
27         ssize_t s = read(0, buf, sizeof(buf)-1);
28         if(s > 0 ){
29             buf[s] = 0;
30             write(wfd, buf, strlen(buf));
31         }else if(s <= 0){
32             ERR_EXIT("read");
33         }
34     }
35
36     close(wfd);
37     return 0;
38 }

```

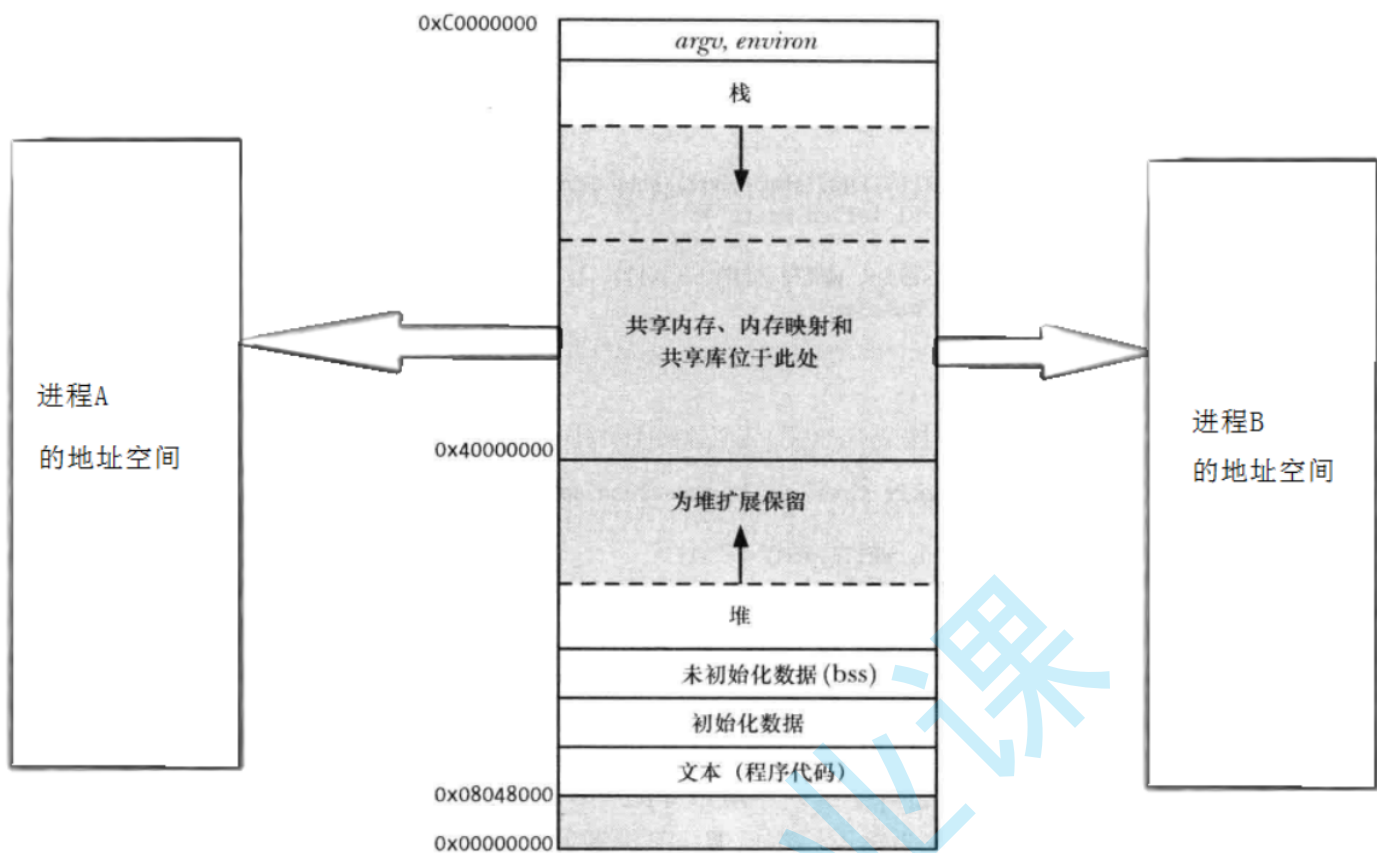
结果:

The image shows two terminal windows side-by-side. The left window is titled 'hb@MiWiFi-R1CL-srv:/home/hb/bit-code/listen_class/pipe' and shows the output of running './serverPipe'. It displays 'Please wait...' followed by receiving input from a client: 'client say# hello world', 'client say# nihao', 'client say# good', and finally 'client quit, exit now!'. The right window is titled 'hb@MiWiFi-R1CL-srv:/home/hb/bit-code/listen_cl' and shows the output of running './clientPipe'. It displays 'Please Enter# hello world', 'Please Enter# nihao', 'Please Enter# good', 'Please Enter# ^C', and then returns to the prompt '[root@MiWiFi-R1CL-srv pipe]# '.

5. system V共享内存

共享内存区是最快的IPC形式。一旦这样的内存映射到共享它的进程的地址空间，这些进程间数据传递不再涉及到内核，换句话说就是进程不再通过执行进入内核的系统调用来传递彼此的数据

5-1 共享内存示意图



5-2 共享内存数据结构

```
1 struct shmid_ds {
2     struct ipc_perm    shm_perm;           /* operation perms */
3     int                shm_segsz;          /* size of segment
    (bytes) */
4     __kernel_time_t    shm_atime;          /* last attach time */
5     __kernel_time_t    shm_dtime;          /* last detach time */
6     __kernel_time_t    shm_ctime;          /* last change time */
7     __kernel_ipc_pid_t shm_cpid;           /* pid of creator */
8     __kernel_ipc_pid_t shm_lpid;           /* pid of last operator */
9     unsigned short      shm_nattch;        /* no. of current
    attaches */
10    unsigned short      shm_unused;         /* compatibility */
11    void                shm_unused2;        /* ditto - used by
    DIPC */
12    void                shm_unused3;        /* unused */
13 };
```

5-3 共享内存函数

shmget函数

- 1 功能：用来创建共享内存
- 2 原型
- 3 `int shmget(key_t key, size_t size, int shmflg);`
- 4 参数
- 5 key:这个共享内存段名字
- 6 size:共享内存大小
- 7 shmflg:由九个权限标志构成，它们的用法和创建文件时使用的mode模式标志是一样的
- 8 取值为IPC_CREAT：共享内存不存在，创建并返回；共享内存已存在，获取并返回。
- 9 取值为IPC_CREAT | IPC_EXCL：共享内存不存在，创建并返回；共享内存已存在，出
错返回。
- 10 返回值：成功返回一个非负整数，即该共享内存段的标识码；失败返回-1

shmat函数

- 1 功能：将共享内存段连接到进程地址空间
- 2 原型
- 3 `void *shmat(int shmid, const void *shmaddr, int shmflg);`
- 4 参数
- 5 shmid：共享内存标识
- 6 shmaddr:指定连接的地址
- 7 shmflg:它的两个可能取值是SHM_RND和SHM_RDONLY
- 8 返回值：成功返回一个指针，指向共享内存第一个节；失败返回-1

说明：

- 1 shmaddr为NULL，核心自动选择一个地址
- 2 shmaddr不为NULL且shmflg无SHM_RND标记，则以shmaddr为连接地址。
- 3 shmaddr不为NULL且shmflg设置了SHM_RND标记，则连接的地址会自动向下调整为SHMLBA的整数倍。
公式： $shmaddr - (shmaddr \% SHMLBA)$
- 4 shmflg=SHM_RDONLY，表示连接操作用来只读共享内存

shmdt函数

- 1 功能：将共享内存段与当前进程脱离
- 2 原型
- 3 `int shmdt(const void *shmaddr);`
- 4 参数
- 5 shmaddr：由shmat所返回的指针
- 6 返回值：成功返回0；失败返回-1
- 7 注意：将共享内存段与当前进程脱离不等于删除共享内存段

shmctl函数

1 功能：用于控制共享内存

2 原型

3 int shmctl(int shmid, int cmd, struct shmid_ds *buf);

4 参数

5 shmid:由shmget返回的共享内存标识码

6 cmd:将要采取的动作（有三个可取值）

7 buf:指向一个保存着共享内存的模式状态和访问权限的数据结构

8 返回值：成功返回0；失败返回-1

命令	说明
IPC_STAT	把shmid_ds结构中的数据设置为共享内存的当前关联值
IPC_SET	在进程有足够权限的前提下，把共享内存的当前关联值设置为shmid_ds数据结构中给出的值
IPC_RMID	删除共享内存段

实例1. 共享内存实现通信

测试代码结构

```
1 # ls
2 client.c  comm.c  comm.h  Makefile  server.c
3 # cat Makefile
4
5 .PHONY:all
6 all:server client
7
8 client:client.c comm.c
9     gcc -o $@ $^
10 server:server.c comm.c
11     gcc -o $@ $^
12 .PHONY:clean
13 clean:
14     rm -f client server
```

comm.h


```

1 #ifndef _COMM_H_
2 #define _COMM_H_
3 # include <stdio.h>
4 # include <sys/types.h>
5 # include <sys/ipc.h>
6 # include <sys/shm.h>
7 # define PATHNAME "."
8 # define PROJ_ID 0x6666
9
10 int createShm(int size);
11 int destroyShm(int shmid);
12 int getShm(int size);
13
14 # endif

```

comm.c

```

1 #include "comm.h"
2
3 static int commShm(int size, int flags)
4 {
5     key_t key = ftok(PATHNAME, PROJ_ID);
6     if(key < 0){
7         perror("ftok");
8         return -1;
9     }
10    int shmid = 0;
11    if( (shmid = shmget(key, size, flags)) < 0){
12        perror("shmget");
13        return -2;
14    }
15    return shmid;
16 }
17 int destroyShm(int shmid)
18 {
19     if(shmctl(shmid, IPC_RMID, NULL) < 0){
20         perror("shmctl");
21         return -1;
22     }
23     return 0;
24 }
25
26 int createShm(int size)
27 {
28     return commShm(size, IPC_CREAT|IPC_EXCL|0666);

```

```

29 }
30
31 int getShm(int size)
32 {
33     return commShm(size, IPC_CREAT);
34 }

```

server.c

```

1 #include "comm.h"
2
3 int main()
4 {
5     int shmId = createShm(4096);
6
7     char *addr = shmat(shmId, NULL, 0);
8     sleep(2);
9     int i = 0;
10    while(i++<26){
11        printf("client# %s\n", addr);
12        sleep(1);
13    }
14
15    shmdt(addr);
16    sleep(2);
17    destroyShm(shmId);
18    return 0;
19 }

```

client.c

```

1 #include "comm.h"
2
3 int main()
4 {
5     int shmId = getShm(4096);
6     sleep(1);
7     char *addr = shmat(shmId, NULL, 0);
8     sleep(2);
9     int i = 0;
10    while(i<26){
11        addr[i] = 'A'+i;
12        i++;

```

```

13         addr[i] = 0;
14         sleep(1);
15     }
16
17     shmdt(addr);
18     sleep(2);
19     return 0;
20 }

```

结果演示

The image shows two terminal windows side-by-side. The left window is titled 'hb@MiWiFi-R1CL-srv:/home/hb/bit-code/L2_class/11_class/shm' and shows the output of running './server'. It prints a series of strings from 'client#' to 'client# ABCDEFGHIJKLMNOP'. The right window is titled 'hb@MiWiFi-R1CL-srv:/home/hb/bit-code/L2_class/11_class/shm' and shows the output of running './client'. It prints '^C' and then returns to the prompt.

ctrl+c终止进程,再次重启

```

1 # ./server
2 shmget: File exists
3 # ipcs -m
4 ----- Shared Memory Segments -----
5 key          shmid    owner      perms      bytes      nattch     status
6 0x66026a25  688145    root       666        4096       0
7
8 # ipcrm -m 688145 #删除shm ipc资源，注意，不是必须通过手动来删除，这里只为演示相关指令，删除IPC资源是进程该做的事情

```

bytes 和 nattch 有时间可以研究一下

注意：共享内存没有进行同步与互斥！共享内存缺乏访问控制！会带来并发问题。

实例2. 借助管道实现访问控制版的共享内存

Comm.hpp

```

1 #pragma once
2
3 #include <fcntl.h>

```

```
4 #include <sys/ipc.h>
5 #include <sys/shm.h>
6 #include <sys/stat.h>
7 #include <sys/types.h>
8 #include <unistd.h>
9 #include <cassert>
10 #include <cstdio>
11 #include <ctime>
12 #include <cstring>
13 #include <iostream>
14
15 using namespace std;
16
17 #define Debug 0
18 #define Notice 1
19 #define Warning 2
20 #define Error 3
21 const std::string msg[] = {
22     "Debug",
23     "Notice",
24     "Warning",
25     "Error"
26 };
27 std::ostream &Log(std::string message, int level)
28 {
29     std::cout << " | " << (unsigned)time(nullptr) << " | " << msg[level] << "
30     | " << message;
31     return std::cout;
32 }
33 #define PATH_NAME "/home/hyb"
34 #define PROJ_ID 0x66
35 #define SHM_SIZE 4096 // 共享内存的大小, 最好是页(PAGE: 4096)的整数倍
36 #define FIFO_NAME "./fifo"
37 class Init {
38 public:
39     Init() {
40         umask(0);
41         int n = mkfifo(FIFO_NAME, 0666);
42         assert(n == 0);
43         (void)n;
44         Log("create fifo success", Notice) << "\n";
45     }
46     ~Init() {
47         unlink(FIFO_NAME);
48         Log("remove fifo success", Notice) << "\n";
49     }
}
```

```

50 };
51
52 #define READ O_RDONLY
53 #define WRITE O_WRONLY
54 int OpenFIFO(std::string pathname, int flags) {
55     int fd = open(pathname.c_str(), flags);
56     assert(fd >= 0);
57     return fd;
58 }
59
60 void CloseFifo(int fd) {
61     close(fd);
62 }
63
64 void Wait(int fd) {
65     Log("等待中....", Notice) << "\n";
66     uint32_t temp = 0;
67     ssize_t s = read(fd, &temp, sizeof(uint32_t));
68     assert(s == sizeof(uint32_t));
69     (void)s;
70 }
71
72 void Signal(int fd) {
73     uint32_t temp = 1;
74     ssize_t s = write(fd, &temp, sizeof(uint32_t));
75     assert(s == sizeof(uint32_t));
76     (void)s;
77     Log("唤醒中....", Notice) << "\n";
78 }
79
80 string TransToHex(key_t k) {
81     char buffer[32];
82     snprintf(buffer, sizeof buffer, "0x%x", k);
83     return buffer;
84 }
85

```

ShmServer.cc

```

1 #include "Comm.hpp"
2
3 Init init;
4
5 int main() {
6     // 1. 创建公共的Key值

```

```

7     key_t k = ftok(PATH_NAME, PROJ_ID);
8     assert(k != -1);
9     Log("create key done", Debug) << " server key : " << TransToHex(k) << endl;
10
11     // 2. 创建共享内存 -- 建议要创建一个全新的共享内存 -- 通信的发起者
12     int shmid = shmget(k, SHM_SIZE, IPC_CREAT | IPC_EXCL | 0666);
13     if (shmid == -1) {
14         perror("shmget");
15         exit(1);
16     }
17     Log("create shm done", Debug) << " shmid : " << shmid << endl;
18
19     // 3. 将指定的共享内存, 挂接到自己的地址空间
20     char* shmaddr = (char*)shmat(shmid, nullptr, 0);
21     Log("attach shm done", Debug) << " shmid : " << shmid << endl;
22
23     // 4. 访问控制
24     int fd = OpenFIFO(FIFO_NAME, O_RDONLY);
25     while (true) {
26         // 阻塞
27         Wait(fd);
28         // 临界区
29         printf("%s\n", shmaddr);
30         if (strcmp(shmaddr, "quit") == 0)
31             break;
32     }
33     CloseFifo(fd);
34
35     // 5. 将指定的共享内存, 从自己的地址空间中去关联
36     int n = shmdt(shmaddr);
37     assert(n != -1);
38     (void)n;
39     Log("detach shm done", Debug) << " shmid : " << shmid << endl;
40
41     // 6. 删除共享内存, IPC_RMID即便是有进程和当下的shm挂接, 依旧删除共享内存
42     n = shmctl(shmid, IPC_RMID, nullptr);
43     assert(n != -1);
44     (void)n;
45     Log("delete shm done", Debug) << " shmid : " << shmid << endl;
46     return 0;
47 }

```

ShmClient.cc

```
1 #include "Comm.hpp"
```

```
2
3 int main() {
4     // 1. 创建公共的Key值
5     key_t k = ftok(PATH_NAME, PROJ_ID);
6     if (k < 0) {
7         Log("create key failed", Error) << " client key : " << TransToHex(k)
8         << endl;
9         exit(1);
10    }
11    Log("create key done", Debug) << " client key : " << TransToHex(k) << endl;
12    // 2. 获取共享内存
13    int shmid = shmget(k, SHM_SIZE, 0);
14    if (shmid < 0) {
15        Log("create shm failed", Error) << " client key : " << TransToHex(k)
16        << endl;
17        exit(2);
18    }
19    Log("create shm success", Error) << " client key : " << TransToHex(k) <<
20    endl;
21    // 3. 挂接共享内存
22    char* shmaddr = (char*)shmat(shmid, nullptr, 0);
23    if (shmaddr == nullptr) {
24        Log("attach shm failed", Error) << " client key : " << TransToHex(k)
25        << endl;
26        exit(3);
27    }
28    Log("attach shm success", Error) << " client key : " << TransToHex(k) <<
29    endl;
30    // 4. 写
31    int fd = OpenFIFO(FIFO_NAME, O_WRONLY);
32    while (true) {
33        ssize_t s = read(0, shmaddr, SHM_SIZE - 1);
34        if (s > 0) {
35            shmaddr[s - 1] = 0;
36            Signal(fd);
37            if (strcmp(shmaddr, "quit") == 0)
38                break;
39        }
40    }
41    CloseFifo(fd);
42    // 5. 去关联
43    int n = shmdt(shmaddr);
44    assert(n != -1);
```

```
44     Log("detach shm success", Error) << " client key : " << TransToHex(k) <<
    endl;
45     return 0;
46 }
```

6. system V消息队列 — 选学了解即可

- 消息队列提供了一个从一个进程向另外一个进程发送一块数据的方法
- 每个数据块都被认为是有一个类型，接收者进程接收的数据块可以有不同的类型值
- 特性方面
 - IPC资源必须删除，否则不会自动清除，除非重启，所以system V IPC资源的生命周期随内核

7. system V信号量 — 选学了解即可

信号量主要用于同步和互斥的，下面先来看看什么是同步和互斥。

7-1 并发编程，概念铺垫

- 多个执行流(进程), 能看到的同一份公共资源：共享资源
- 被保护起来的资源叫做临界资源
- 保护的方式常见：互斥与同步
- 任何时刻，只允许一个执行流访问资源，叫做互斥
- 多个执行流，访问临界资源的时候，具有一定的顺序性，叫做同步
- 系统中某些资源一次只允许一个进程使用，称这样的资源为临界资源或互斥资源。
- 在进程中涉及到互斥资源的程序段叫临界区。你写的代码=访问临界资源的代码(临界区)+不访问临界资源的代码(非临界区)
- 所谓的对共享资源进行保护，本质是对访问共享资源的代码进行保护



7-2 信号量

特性方面

- IPC资源必须删除，否则不会自动清除，除非重启，所以system V IPC资源的生命周期随内核

理解方面

- 信号量是一个计数器

作用方面

- 保护临界区

本质方面

- 信号量本质是对资源的预订机制

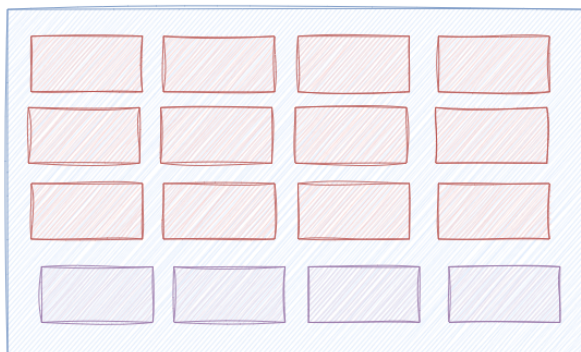
操作方面


- 申请资源，计数器--，P操作
- 释放资源，计数器++，V操作

资源整体使用



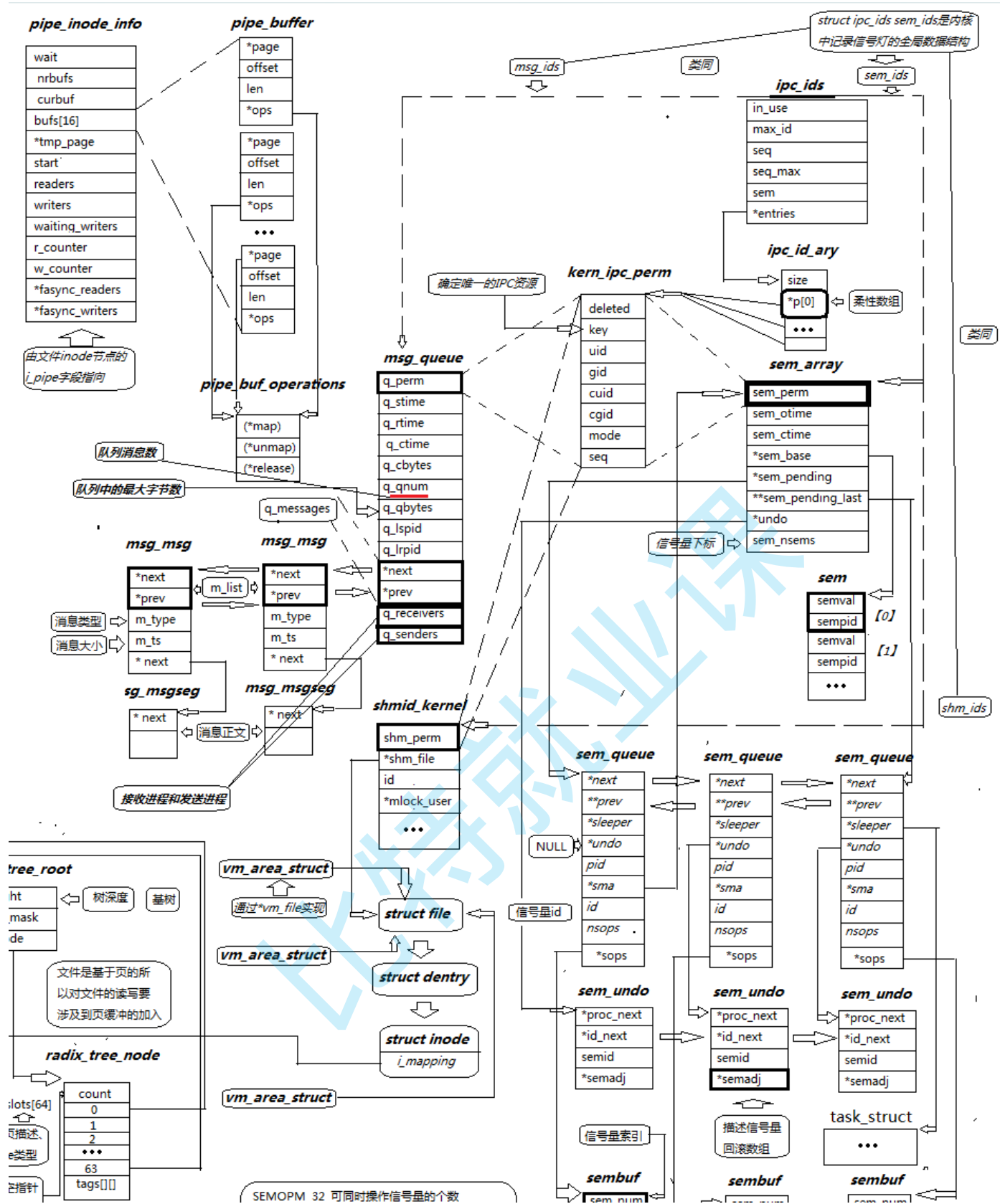
资源不是整体使用



 电影院的例子

8. 内核是如何组织管理IPC资源的

比特就业课



- 参考Linux内核 2.6.11 源码，其他源码实现可能有差别
- 课堂看源码可以看 2.6.18 ，这块从11到18变化不大

```
struct ipc_ids {  
    ...  
};  
替换  
1 个结果, 包含于 1 个文件中 - 在编辑器中打开  
h util.h linux-2.6.18\ipc 9 1  
struct ipc_ids {
```

linux-2.6.18 > ipc > h util.h > ipc_ids > mutex

```
14 void sem_init (void);  
15 void msg_init (void);  
16 void shm_init (void);  
17  
18 struct ipc_id_ary {  
19     int size;  
20     struct kern_ipc_perm *p[0];  
21 };  
22  
23 struct ipc_ids {  
24     int in_use;  
25     int max_id;  
26     unsigned short seq;  
27     unsigned short seq_max;  
28     struct mutex mutex;  
29     struct ipc_id_ary nullentry;  
30     struct ipc_id_ary* entries;  
31 };  
22
```

```
struct ipc_id_ary {  
    int size;  
    struct kern_ipc_perm *p[0];  
};
```

```
struct kern_ipc_perm  
{  
    spinlock_t lock;  
    int deleted;  
    key_t key;  
    uid_t uid;  
    gid_t gid;  
    uid_t cuid;  
    gid_t cgid;  
    mode_t mode;  
    unsigned long seq;  
    void *security;  
};
```

```

/* one msq_queue structure for each present queue on the system */
struct msg_queue {
    struct kern_ipc_perm q_perm;
    int q_id;
    time_t q_stime;           /* last msgsnd time */
    time_t q_rtime;          /* last msgrcv time */
    time_t q_ctime;          /* last change time */
    unsigned long q_cbytes;   /* current number of bytes on queue */
    unsigned long q_qnum;     /* number of messages in queue */
    unsigned long q_qbytes;   /* max number of bytes on queue */
    pid_t q_lspid;           /* pid of last msgsnd */
    pid_t q_lrpid;           /* last receive pid */

    struct list_head q_messages;
    struct list_head q_receivers;
    struct list_head q_senders;
};

/* One sem_array data structure for each set of semaphores in the system. */
struct sem_array {
    struct kern_ipc_perm sem_perm; /* permissions .. see ipc.h */
    int sem_id;
    time_t sem_otime; /* last semop time */
    time_t sem_ctime; /* last change time */
    struct sem *sem_base; /* ptr to first semaphore in array */
    struct sem_queue *sem_pending; /* pending operations to be processed */
    struct sem_queue **sem_pending_last; /* last pending operation */
    struct sem_undo *undo; /* undo requests on this array */
    unsigned long sem_nsems; /* no. of semaphores in array */
};

```

```

struct shmid_kernel /* private to the kernel */
{
    struct kern_ipc_perm    shm_perm;
    struct file *          shm_file;
    int                    id;
    unsigned long          shm_nattch;
    unsigned long          shm_segsz;
    time_t                 shm_atim;
    time_t                 shm_dtim;
    time_t                 shm_ctim;
    pid_t                  shm_cprid;
    pid_t                  shm_lprid;
    struct user_struct *mlock_user;
};

```

附录：

- 在minishell中添加管道的实现，下面代码供有兴趣的同学参考，课堂不做说明

```

1 # include <stdio.h>
2 # include <stdlib.h>
3 # include <unistd.h>
4 # include <string.h>
5 # include <fcntl.h>
6
7 # define MAX_CMD 1024
8 char command[MAX_CMD];
9 int do_face()
10 {
11     memset(command, 0x00, MAX_CMD);
12     printf("minishell$ ");
13     fflush(stdout);
14     if (scanf("%[^\n]%"c", command) == 0) {
15         getchar();
16         return -1;
17     }
18     return 0;
19 }

```

```

20 char **do_parse(char *buff)
21 {
22     int argc = 0;
23     static char *argv[32];
24     char *ptr = buff;
25
26     while(*ptr != '\0') {
27         if (!isspace(*ptr)) {
28             argv[argc++] = ptr;
29             while((!isspace(*ptr)) && (*ptr) != '\0') {
30                 ptr++;
31             }
32             continue;
33         }
34         *ptr = '\0';
35         ptr++;
36     }
37     argv[argc] = NULL;
38     return argv;
39 }
40 int do_redirect(char *buff)
41 {
42     char *ptr = buff, *file = NULL;
43     int type = 0, fd, redirect_type = -1;
44     while(*ptr != '\0') {
45         if (*ptr == '>') {
46             *ptr++ = '\0';
47             redirect_type++;
48             if (*ptr == '>') {
49                 *ptr++ = '\0';
50                 redirect_type++;
51             }
52             while(isspace(*ptr)) {
53                 ptr++;
54             }
55             file = ptr;
56             while((!isspace(*ptr)) && *ptr != '\0') {
57                 ptr++;
58             }
59             *ptr = '\0';
60             if (redirect_type == 0) {
61                 fd = open(file, O_CREAT|O_TRUNC|O_WRONLY, 0664);
62             }else {
63                 fd = open(file, O_CREAT|O_APPEND|O_WRONLY, 0664);
64             }
65             dup2(fd, 1);
66         }

```

```
67     ptr++;
68 }
69 return 0;
70 }
71 int do_command(char *buff)
72 {
73     int pipe_num = 0, i;
74     char *ptr = buff;
75     int pipefd[32][2] = {{-1}};
76     int pid = -1;
77
78     pipe_command[pipe_num] = ptr;
79     while(*ptr != '\0') {
80         if (*ptr == '|') {
81             pipe_num++;
82             *ptr++ = '\0';
83             pipe_command[pipe_num] = ptr;
84             continue;
85         }
86         ptr++;
87     }
88     pipe_command[pipe_num + 1] = NULL;
89     return pipe_num;
90 }
91 int do_pipe(int pipe_num)
92 {
93     int pid = 0, i;
94     int pipefd[10][2] = {{0}};
95     char **argv = {NULL};
96
97     for (i = 0; i <= pipe_num; i++) {
98         pipe(pipefd[i]);
99     }
100     for (i = 0; i <= pipe_num; i++) {
101         pid = fork();
102         if (pid == 0) {
103             do_redirect(pipe_command[i]);
104             argv = do_parse(pipe_command[i]);
105             if (i != 0) {
106                 close(pipefd[i][1]);
107                 dup2(pipefd[i][0], 0);
108             }
109             if (i != pipe_num) {
110                 close(pipefd[i + 1][0]);
111                 dup2(pipefd[i + 1][1], 1);
112             }
113             execvp(argv[0], argv);
```



```
114         }else {
115             close(pipefd[i][0]);
116             close(pipefd[i][1]);
117             waitpid(pid, NULL, 0);
118         }
119     }
120     return 0;
121 }
122
123 int main(int argc, char *argv[])
124 {
125     int num = 0;
126     while(1) {
127         if (do_face() < 0)
128             continue;
129         num = do_command(command);
130         do_pipe(num);
131     }
132     return 0;
133 }
```