LINUX网络编程LINUX.note

进程间通信（IPC）

1. 内核空间里的一片缓存区（默认大小4096）
2. 进程间通信的方式

- 管道（父子进程，兄弟进程）
- 信号（开销）
- 共享映射区
- 本地套接字（最稳定）

3. 管道

- 管道实现原理：管道实为内核使用环形队列机制，借助内核缓冲区（4k）实现
- 队列：FIFO（先进先出）
- 环形队列：由软件实现，由尾部将数据压入队列中当队列填满之后，再压如数据就会将数据覆盖
- pipe（）函数就可以通过管道调用这个内核空间用来进程间通信的缓存区
- 本质是伪文件
- 有两个文件描述符，一个表示读，一个表示写
- 管道规定数据从管道的写端流入管道，从读端流出
- 管道的局限性：数据不能进程自己写，自己读

数据不能反复读，反复写

双向半双工

只能在拥有共同祖先的进程间使用管道

- 管道的用法：创建并打开管道：

```
int pipe(int pipefd[2]);成功 0 失败 -1 设置errno
参数fd[0]读端 fd[1]:写端
```

- 管道的例程

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <unistd.h>
#include <pthread.h>

void sys_err(const char *str)
{
        perror(str);
        exit(1);
}

int main(int argc,char *argv[])
```

```
{
        int ret;
        int fd[2];
        pid_t pid;
        char buf[4096];
        ret = pipe(fd);
        if(ret== -1)
                sys_err("pipi err");
        pid = fork();//fd[0] read
        if(pid > 0){
        close(fd[0]);
        write(fd[1],"hello ",strlen("hello"));
        close(fd[1]);

        }else if(pid == 0){
                close(fd[1]);
                ret = read(fd[0],buf,sizeof(buf));
        write(STDOUT_FILENO,buf,ret);
        close(fd[0]);
}
```

- 读管道：管道中有数据，read返回实际读到的字节数

  管道中无数据（1）管道写端被全部关闭，read返回0（好像读到文件结尾）

  （2）写端没有全部关闭，read阻塞等待（此时会让出CPU）

  - 写管道：管道读端全部被关闭，进程异常被终止

  管道读端没有全部关闭

  （1）管道已满，write阻塞

  （2）管道未满，write将数据写入，并返回实际写入的字节数

4. 实现 ls | wc -l

```
 void sys_err(const char *str)
{
        perror(str);
        exit(1);
}

int main(int argc,char *argv[])
{
        int fd[2];
```

```
        int ret;
        pid_t pid;
        ret = pipe(fd);
        if(ret < 0)
        {
        sys_err("pipe err");
        }
        pid = fork();
        if (pid < 0){
                sys_err("fork err:");
        }
        if(pid >0){
                close(fd[0]);
                dup2(fd[1],STDOUT_FILENO);
                execlp("ls","ls",NULL);
                sys_err("ls_execlp err:");


        }
        else if (pid == 0){
                close(fd[1]);
                        dup2(fd[0],STDIN_FILENO);
                execlp("wc","wc","-l",NULL);
                sys_err("wc_execlp err:");

        }
        return 0;
}
```

5. 有名管道

```
int mkfifo(const char *pathname, mode_t mode);//创建了管道文件
两个没有关系的进程间通信
这种方法就是通过文件来进行通讯，通讯的时候open 然后read write
```

6. 存储映射I/O：将磁盘上的文件映射到内存，这样就可以在不适用read和write函数的情况下，使用地址（指针）完成I/O操作

```
void *mmap(void *addr, size_t length, int prot, int flags,
                  int fd, off_t offset);
参数：addr:指定映射区的地址，通常传NULL表示让系统自动分配
```

```
    length:表示共享内存映射区的大小(<=文件的实际大小))
    prot:共享内存映射区的读写属性。PROT_READ、PROT_WRITE、
    flag:标注共享内存的共享属性：MAP_SHARED 、 MAP_SHARED_VALIDATE
    fd:用于创建共享内存映射区的那个文件的文件描述符
    offset:默认0，表示映射文件全部偏移位置须是4k的整数倍
返回值：
    成功：指向映射区的首地址
    失败：MAP_FAILED ERROR
释放映射    int munmap(void *addr, size_t length);

```

- mmap注意事项

```
 用于创建映射区的大小为0.实际制定非0大小创建映射区出总线错误
  用于创建映射区的大小为0.实际制定0大小创建映射区出无效参数错误
  用于创建映射区域文件读写属性为只读，映射区属性为读、写。出无效错误
  offset必须是4096的整数倍（MMU映射的单位是4k）
  munmap释放的地址，必须是mmap申请的地址
  创建映射区的过程中隐含着一次对文件的打开操作
  mmap创建映射区出错率特别高，必须要检查mmap的返回值
```

- mmap test

```c
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <unistd.h>
#include <pthread.h>
#include <sys/mman.h>
#include <fcntl.h>
void sys_err(const char *str)
{
        perror(str);
        exit(1);
}

int main(int argc,char *argv[])
{
        char *p = NULL;
        int fd;
        int len;
```

```
        fd = open("test_mmap",O_RDWR|O_CREAT|O_TRUNC,777);
        if(fd == -1){
                sys_err("open err\n");
        }
        len = lseek(fd,10,SEEK_END);
        write(fd,"\n",1);
        //len = lseek(fd,0,SEEK_END);
        p = mmap(NULL,len,PROT_READ|PROT_WRITE,MAP_SHARED,fd,0);
        if(p == MAP_FAILED){
        sys_err("mmap_fail:");
        }
        strcpy(p,"hello mmap");
        printf("-----------%s------------\n",p);
        int ret = munmap(p,len);
        if(ret < 0){
                sys_err("mummap err:");
        }
return 0;
}
```

- mmap父子进程间通信：需要在创建映射区的时候指定对应的标志位参数flags

```
MAP_PRIVATE(私有映射)：父子进程各自独占映射区
MAP_SHARED(共享映射)：父子进程共享映射区
父子进程使用mmap进行进程间通信先映射再fork

```

- mmap无血缘关系进程间通信：

7. **信号**

- **简单**不能携带大量的信息 、满足条件才能发送。
- 信号的机制

信号是软件层面上的“中断”，无论程序执行到什么位置，必须立即停止运行，

所有信号的处理和产生都是由内核产生的

- 阻塞信号集：将某些信号加入集合，对他们设置屏蔽，当屏蔽x信号后，再收到该信号，该信号的处理将推后（解除屏蔽后）
- 未决信号集：信号产生，未决信号的位会立刻反转为1（屏蔽信号集）
- **信号四要素**：信号编号、信号名称、信号对应事件、默认处理时间
- man 7 signal

```
SIGHUP:当用户退出shell时，由该shell启动的所有进程将受到这个信号，默认动作为终止进程
SIGINT:ctrl+c
SIBQUIT:ctrl+\
SIGKILL:无条件终止进程
SIGTOP:无条件停止进程
SIGCHLD:子进程状态发生变化时，父进程会收到这个信号，默认动作时忽略这个信号（唯一一个默认忽略的信
SIGPIPE:向一个没有读端
```

- **需要注意，信号会被递送，但是不一定递达，不应该乱发信号**
-
-
-
-
-

8. kil函数：给指定进程发送指定信号（不一定杀死）

```
 int kill(pid_t pid, int sig);
```

9. 其他几个发信号的函数

```
int raise(void)
```

10. alarm函数：设置定时器。在指定seconds后，内核会给当前进程发送14）SIGAKLRM信号。进程收到该信号，默认动作终止进程

每个进程都有且只有唯一一个定时器

```
unsigned int alarm(unsigned int seconds);
返回的是上次信号剩余的秒数
取消定时alarm(0)
定时器与进程状态无关，无论进程处于何种状态，alarm都计时

```

- 练习

```
 编写程序，测试你的计算机一秒钟能数多少个数
 实际执行时间=系统时间（内核）+用户时间+等待时间
 优化程序：首选优化IO
```

11. setitimer函数：设置定时器。可替代alarm函数。精度微秒us 可以实现周期定时

```
 int getitimer(int which, struct itimerval *curr_value);
```

```
 int setitimer(int which, const struct itimerval *new_value,
```

```
                    struct itimerval *old_value);
 which ITIMER_REAL:自然时间计时法，计算自然时间（发送 14 SIGLARN 信号）
         ITIMER_VIRTUAL:虚拟时间计时法（用户空间）：进程在用户空间的时间，至今算进程占用cpu的时
         ITIMER_PROF：运行时计时，计算占用cpu及执行系统调用的时间（发送 27 SIGPROT信号）
new_value:定时秒数
old_value:传出参数，上次定时剩余时间
```

12. 信号集操作函数

进程地址空间里面包含进程控制块（pcb）（其实就是一个结构体），进程控制块里面有两个信号集

阻塞信号集是可以操作的，未决信号集时不能进行操作

- 阻塞信号集
- 未决信号集

```
 sigset_t set; //typedef ubsigned long sigset_t：自定义信号集
 int sigemptyset(sigset_t *set);将莫格信号集清0
 int sigfillset(sigset_t *set);将某个信号集置1
 int sigdelset(sigset_t *set,int signum);将某个信号请出信号集
 int sigismember(const sigset_t *set,int signum);判断某个信号是否在信号集中
 int sigaddset():增加一个信号至信号集
 int sigpending(sigset_t *set);读取当前未决信号集，通过set参数传出
```

- sigprocmask函数：用来屏蔽信号、接触屏蔽也是用该信号。其本质，读取或修改进程的信号屏蔽字（PCB中）

```
int sigprocmask(int how, const sigset_t *set, sigset_t *oldset);
how：SIG_BLOCK 阻塞
    SIG_UNBLOCK
    SIG_SETMASK:批量设置信号屏蔽集
set：操作的信号集
oldset：传出参数
```

- sigpending函数：参数为传出参数（读取当前进程的未决信号集）
- test：信号屏蔽

```
  void sys_err(const char *str)
{
        perror(str);
        exit(1);
}
void print_set(sigset_t *set)
{
```

```
        int i;
        for(i = 1; i < 32; i++)
        {
                if(sigismember(set,i))
                        putchar('1');
                else
                        putchar('0');
        }
        printf("\n");
}
int main(void)
{
        int ret;
        sigset_t set,oldset,pedset;
        sigemptyset(&set);
        sigaddset(&set,SIGINT);
        ret = sigprocmask(SIG_BLOCK,&set,&oldset);
        if(ret != 0)
        {
                sys_err("sigprocmask error");
        }
        while(1)
        {
        ret = sigpending(&pedset);
        if(ret == -1){
                sys_err("sigpeding error");
        }
        print_set(&pedset);
        sleep(1);
        }
        return 0;

}

```

-

13. 信号捕捉

- signal函数

```
注册一个信号捕捉函数
这个函数在不同版本的linux会有不同的行为，尽量比米娜使用它，取而代之使用sigaction函数
```

```
typedef void (*sighandler_t)(int);//函数指针：指向一个参数为int类型，返回值为void的指针。type
sighandler_t signal(int signum, sighandler_t handler);
signum:注册的信号的number
handler:捕捉到函数之后需要做的动作（函数）
```

```c
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <unistd.h>
#include <pthread.h>
#include <sys/mman.h>
#include <fcntl.h>
#include <signal.h>

void catch(int num)
{
printf("catch the signal\n");
}

int main()
{
signal(SIGINT,catch);//这里的这个处理函数只要是 返回值为void，参数为int就可以、
while(1);
return 0;
}

```

- sigaction函数：注册信号捕捉函数

```c
int sigaction(int signum, const struct sigaction *act,
                      struct sigaction *oldact);
 struct sigaction {
               void     (*sa_handler)(int);//函数的名字
               void     (*sa_sigaction)(int, siginfo_t *, void *);//一般不用
               sigset_t   sa_mask;//只工作在信号捕捉函数工作期间有用；
                                  //sigset_t 信号集 sa_mask 在函数运行期间会将阻塞信号集替
                                  //当 捕捉信号函数运行结束后再继续运行原来的mask；然后在这
                                  //上面一切操作信号的函数
```

```
            int        sa_flags;//需要设置的一些参数
                                //0 :屏蔽本信号
                                //1: 默认不屏蔽本信号
            void     (*sa_restorer)(void);//废弃
        };
 oldact:保存的是原来默认信号的处理状态
 act: 捕捉到信号的动作

```

- demo

```c
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <unistd.h>
#include <pthread.h>
#include <sys/mman.h>
#include <fcntl.h>
#include <signal.h>

void sys_err(const char *str)
{
        perror(str);
        exit(1);
}
void catch(int num)
{
printf("catch the signal\n");
}

int main()
{
        int ret;
        struct sigaction act,oldact;
        act.sa_handler = catch;                 //set callback fanction name
        sigemptyset(&act.sa_mask);              //set mask
        act.sa_flags = 0;
        ret = sigaction(SIGINT,&act,&oldact);
                if(ret == -1)
                {
```

```
30                      sys_err("sigaction err:");
31              }
32          while(1);
33  return 0;
34  }
35
```

- 信号捕捉特性

1. 进程正在运行时，默认PCB中有一个信号屏蔽字，它决定了进程自动屏蔽那些信号。当注册了某个信号捕捉函数，捕捉到该函数后，要调用该函数。而该函数有可能执行很长时间，在这期间锁屏蔽的信号不由原来的信号屏蔽字决定。调用完信号处理函数，再恢复

2. 信号捕捉函数执行期间，是否屏蔽xxx信号是由sa_flag决定的

3. 阻塞的常规信号不支持排队，

4. 捕捉函数执行期间，被屏蔽信号多次发送，解除屏蔽后只处理一次（产生多次只记录一次）

- 借助信号捕捉回收子进程

· 只要子进程的状态发生变化，就会产生SIGCHLD信号。（子进程终止时、子进程接收到SIGSTOP信号停止时、子进程处在停止态，接收到SIGCONT后唤醒时

借助SIGCHLD信号回收子进程：子进程结束运行，其父进程会收到SIGCHLD 信号，这个信号时默认忽略的，可以设置捕捉这个信号，来回收子进程。

```
1  void catch_child(int sign)
2  {
3          pid_t waitpid;
4          while(waitpid = wait(NULL) != -1)
5          {
6          printf("-------------catch child is %d \n",(int)waitpid);
7          }
8          return ;
9          }
10 int main(void)
11 {
12         pid_t pid;
13         int i;
14         for(i = 0; i<15; i++){
15                 if((pid = fork()) == 0)
16                 break;
17         }
18         if(15 == i ){
19                 struct sigaction act;
20                 sigemptyset(&act.sa_mask);
21                 act.sa_flags = 0;
```

```
22                  act.sa_handler = catch_child;
23                  sigaction(SIGCHLD,&act,NULL);
24                  printf("I am parent, pid = %d\n",getpid());
25                  while(1);
26          }
27          else {
28
29                  printf("I am child, pid = %d\n" ,getpid());
30
31          }
32
33          return 0;
34  }
35
```

- 中断系统调用

系统调用可以分为两类，慢速系统调用和其他系统调用

1. 慢速系统调用：可能会使进程永远阻塞的系统调用。如 wait（）：它会一直等待子进程结束，如果子进程一直没有结束，则就会永远阻塞在这里

2. 通过设置sa_flags来设置系统调用后被终端后是否重启

-
-
-

14. 守护进程：

- 创建子进程，父进程退出
- 在子进程中创建新会话

setsid（）函数

使子进程完全独立起来，脱离控制

- 改变当前目录为根目录

chdir()函数

防止占用可卸载的文件系统

也可以换成其他路径

- 重设文件权限掩码

umask（）函数

防止继承的文件创建屏蔽字拒绝某些权限

增加守护进程灵活性

- 关闭文件描述符

继承的打开文件不会用到，浪费系统资源，无法卸载

- 业务逻辑

```c
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <unistd.h>
#include <pthread.h>
#include <sys/mman.h>
#include <fcntl.h>
#include <signal.h>

int sys_err(const char *str)
{
        perror(str);
        exit(1);
}

int main()
{

        pid_t pid;
        int ret,fd;
        pid = fork();
        if(pid > 0)
        exit(0);
        setsid();
        if(pid == -1)
        {
        sys_err("fork err");
        }
        ret = chdir("/home/");
        umask(0022);
        close(STDIN_FILENO);
        open("/dev/null",O_RDWR);
        dup2(fd,STDOUT_FILENO);
        dup2(fd,STDERR_FILENO);

        while(1);
        return 0;
}

```

15.
16.
17.
18.
19.
20.

线程同步

1. 概念
2. 互斥量
3. 信号
4. 条件变量