1. 查找命令

```
ps -la
ps查看进程，a显示终端所有进程，l表示显示比较长的输出格式
```

```
ps -ef | grep -E 'bash|demo'
e表示显示所有进程，f表示全格式显示，grep为查找命令，-E表示多个关键字查找
执行结果如下
```

```
yuhufei@ubuntu:~$ ps -ef | grep -E 'bash|demo'
yuhufei   2579  2578  0 22:18 pts/2    00:00:01 -bash
yuhufei   2710  2709  0 22:22 pts/0    00:00:00 -bash
yuhufei   2752  2710  0 22:29 pts/0    00:00:00 ./demo_01
yuhufei   2787  2579  0 22:38 pts/2    00:00:00 grep --color=auto -E bash|demo
yuhufei@ubuntu:~$
```

```
ps -eo pid,ppid,sid,tty,pgrp,comm | grep -E 'bash|PID|demo'
o表示可以指定显示哪些列，pid为进程号，ppid表示父进程，sid表示所述会话，
tty表示终端，pgrp表示进程组，comm表示执行的命令。
执行后得到如下结果
```

```
yuhufei@ubuntu:~$ ps -eo pid,ppid,sid,tty,pgrp,comm | grep -E 'bash|PID|demo'
  PID  PPID   SID TT       PGRP COMMAND
 2579  2578  2579 pts/2    2579 bash
 2710  2709  2710 pts/0    2710 bash
 2752  2710  2710 pts/0    2752 demo_01
yuhufei@ubuntu:~$
```

```
sudo find / -name "signal.h" | xargs grep -in "SIGHUP"
查找名称为signal.h的文件，并在查找到文件中找到SIGHUP关键字；
-in中i表示不区分大小写，n表示显示关键字所在的行号；
xargs的作用是让grep在文件内容中查找。
```

```
sudo strace -e trace=signal -p 2820
追踪一个进程号为2820的进程
```

2. 信号

信号指的是一个通知，用来通知某一个进程发生了某一件事。通常是以SIG开头，如SIGHUP表示挂断。像SIGHUP，SIGINT等被定义成了宏，信号名称被定义为一些数字，且数字是从1开始的，如在使用的时候SIGHUP就被赋值为1，这样方便程序处理。

```
#include <stdio.h>
#include <unistd.h>
#include <signal.h>

int main(int argc, char * const * argv){
        printf("I am so happy to learn with you!\n");
        signal(SIGHUP, SIG_IGN);//SIGHUP表示挂断，SIG_IGN为忽略挂断，
                                //这时候断开进程所在的终端之后，进程不会退出
        for(;;){
                sleep(1);
                printf("休息1秒钟\n");
        }

        printf("The program is END!\n");
        return 0;
}
```


各种信号的宏定义

```
#define SIGHUP           1
#define SIGINT           2
#define SIGQUIT          3
#define SIGILL           4
#define SIGTRAP          5
#define SIGABRT          6
#define SIGIOT           6
#define SIGBUS           7
#define SIGFPE           8
#define SIGKILL          9
#define SIGUSR1         10
#define SIGSEGV         11
#define SIGUSR2         12
#define SIGPIPE         13
#define SIGALRM         14
#define SIGTERM         15
#define SIGSTKFLT       16
```

3.用kill发信号

```
kill 进程id表示杀死进程，其实是向进程发送了一个SIGTERM终止信号。
kill -1 进程号，向进程发送一个SIGHUP信号
kill -2 进程号，向进程发送一个SIGINT信号
kill -19 进程号，向进程发送一个SIGSTOP信号，用于停止进程
```

4.查看进程状态

```
ps -eo pid,ppid,sid,tty,pgrp,comm,stat | grep -E 'bash|PID|demo'
ps -aux | grep -E 'bash|PID|demo'
结果如下
```

```
yuhufei@ubuntu:~/WindowsCodes/chapter_03$ ps -eo pid,ppid,sid,tty,pgrp,comm,stat | grep -E 'bash|PID|demo'
  PID  PPID   SID TT        PGRP COMMAND         STAT
 2579  2578  2579 pts/2     2579 bash            Ss
 2710  2709  2710 pts/0     2710 bash            Ss+
 2853  2852  2853 pts/1     2853 bash            Ss+
yuhufei@ubuntu:~/WindowsCodes/chapter_03$ ps -aux | grep -E 'bash|PID|demo'
USER       PID %CPU %MEM    VSZ   RSS TTY      STAT START   TIME COMMAND
yuhufei   2579  0.0  0.2  22308  5080 pts/2    Ss   Feb28   0:01 -bash
yuhufei   2710  0.0  0.2  22300  5084 pts/0    Ss+  Feb28   0:00 -bash
yuhufei   2853  0.0  0.2  22252  4820 pts/1    Ss+  Feb28   0:00 -bash
yuhufei   3313  0.0  0.0  14220   956 pts/2    R+   10:01   0:00 grep --color=auto -E bash|PID|demo
yuhufei@ubuntu:~/WindowsCodes/chapter_03$
```

其中STAT列表示进程状态列

T表示停止或被追踪

大S表示睡眠中的状态

小s表示session leader（会话首进程），其下有子进程

+表示位于前台的进程

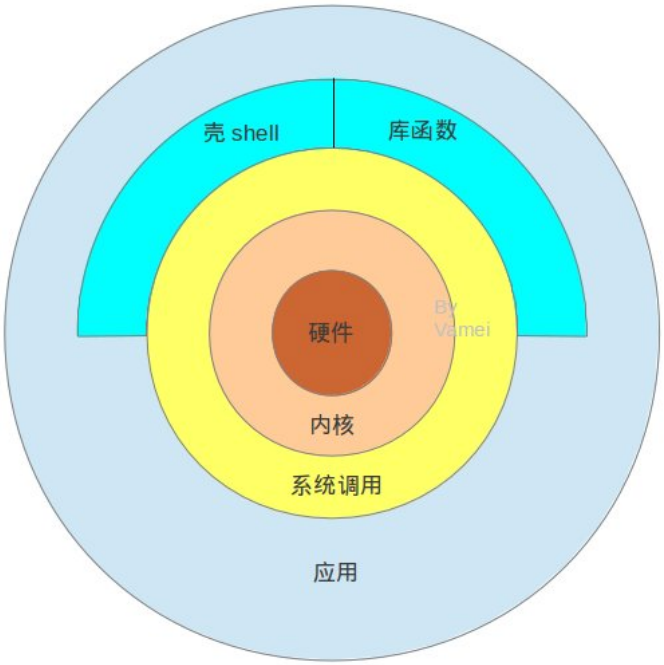
5.对信号的处理

当某个信号出现的时候，可以用以下三种方法进行处理

- 执行系统默认动作。代码中没有针对信号的处理代码，当该进程收到信号时，操作系统会对进程执行一个默认的动作
- 忽略信号。在代码中可以通过写代码的方式，使得进程忽略接收到的信号，则进程不会对该信号做任何动作。如**signal(SIGHUP, SIG_IGN),**SIG_IGN表示忽略该信号。**注：有两个信号是不能被忽略的特权信号。SIGKILL和SIGSTOP，其中kill -9命令向进程发送SIGKILL信号；kill -19表示向进程发送停止信号。**
- 捕捉该信号。程序员在代码中写一个信号处理函数，当系统收到该信号的时候自动调用该处理函数来执行相关代码。注：SIGKILL和SIGSTOP这两个信号是不能被捕捉和忽略的，因此不要写代码尝试捕捉这两个信号。

6. Linux操作系统体系结构

Linux操作系统大致分为用户态和内核态。



- 操作系统-内核：用于控制计算机的硬件资源，提供应用程序运行的环境。
- 程序员编写的程序，一般情况下运行在用户态（应用），当程序要执行一些特殊代码时，程序就可能切换到内核态，这种操作由操作系统控制，不需要人为介入。
- 系统调用其实就是调用一些库函数。
- shell，用shell终端连接虚拟机或服务器的时候就启动了一bash，用于输入各种命令。
- 用户态跳到内核态的几种情况：系统调用、异常事件、外围设备中断。

7. signal系统函数

当收到一个信号的时候，可以使用signal函数进行信号的捕捉和忽略。

```c
#include <stdio.h>
#include <unistd.h> // sleep()
#include <signal.h>

//信号处理函数，当收到某个信号的时候，用该函数处理相关业务
void sig_usr(int signo){
    if(signo == SIGUSR1){
            printf("收到了SIGUSR1信号! \n");
    }
    else if(signo == SIGUSR2){
            printf("收到了SIGUSR2信号! \n");
    }else{
            printf("收到了未捕捉到的信号! \n");
    }
}

int main(int argc, char * const * argv){
    //信号捕捉，signal是系统函数，第一个参数为接收到的信号，
    //第二个参数是针对该信号的业务处理函数
    if(signal(SIGUSR1, sig_usr) == SIG_ERR){
            printf("无法捕捉SIGUSR1信号! \n");
    }
    if(signal(SIGUSR2, sig_usr) == SIG_ERR){
```

```
            printf("无法捕捉SIGUSR2信号! \n");
    }
    for(;;){
            sleep(1);
            printf("休息1秒钟\n");
    }
    return 0;
}
```

当程序运行的时候，在命令行键入kill -USR1 进程号和 kill -USR2 进程号即可向该进程发送信号。

信号发送之后将会被signal（）函数捕捉，进入信号处理函数中来处理相关业务。

8.可重入函数和不可重入函数

可重入函数又称异步信号安全函数，指的是在信号处理的过程中调用是安全的函数。

反之称为不可重复函数。

在信号处理函数中不可以使用不可重入函数。如printf函数和malloc函数。

9.信号集

信号集能够把多个信号的状态保存下来。当进程收到一个信号的时候，会把该信号的状态位设置为1，在未处理完之前如果再次检测到该信号时会将该信号进行排队或丢弃。处理完成后又会将该信号的状态位设置为0。

信号集的数据类型为sigset_t。

10.信号集相关函数

sigemptyset: 将信号集中的所有信号清零，表示这些信号都没有到来

sigfillset：把所有的信号都设置为1，表示这些信号都已经到来，与sigemptyset正好相反。

sigaddset:将某个信号位设置为1。

sigdelset：将某个信号位设置为0。

sigprocmask:设置进程所对应的信号集。

sigismember:检测信号集的特定信号是否被置位。

信号集相关函数的代码演示

```
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <signal.h>

//信号处理函数，当收到某个信号的时候，用该函数处理相关业务
void sig_quit(int signo){
    printf("收到了SIGQUIT信号! \n");
    if(signal(SIGQUIT, SIG_DFL) == SIG_ERR){ // 将SIG_QUIT信号设置成系统默认，收到之后终止进程
            printf("无法为SIGQUIT信号设置默认处理! \n");
            exit(1);
    }
}
int main(int argc, char * const *argv)
{
    sigset_t newmask, oldmask, pendmask; //定义新的信号集和原有的信号集
    if(signal(SIGQUIT, sig_quit) == SIG_ERR){
            printf("无法捕捉SIGUSR1信号! \n");
            exit(1);
    }
    sigemptyset(&newmask); // newmask信号集中的所有信号都清零
    // 设置newmask信号集中的SIGQUIT信号位1，再来SIGQUIT信号时进程就收不到
    sigaddset(&newmask, SIGQUIT);

    //设置该进程所对应的信号集
    if(sigprocmask(SIG_BLOCK, &newmask, &oldmask) < 0){
            //第一个参数为SIG_BLOCK，表示设置当前进程的信号集为当前进程信号集和第二个参数的并集
```

```
28              //即把当前进程信号即设置为newmask，并将原始的信号集保存在第三个参数oldmask中，以备后用。
29              printf("sigprocmask(SIG_BLOCK)失败！\n");
30              exit(1);
31      }
32      printf("我要开始休息10s了，---begin---，此时我无法收到SIGQUIT信号！\n");
33      sleep(10);
34      printf("我已经休息了10s了，---end---\n");
35
36      //检测SIGQUIT是否被置位
37      if(sigismember(&newmask, SIGQUIT)){
38              printf("SIGQUIT信号被屏蔽了！\n");
39      }else{
40              printf("SIGQUIT信号没有被屏蔽了！\n");
41      }
42      //检测SIGHUP是否被置位
43      if(sigismember(&newmask, SIGHUP)){
44              printf("SIGHUP信号被屏蔽了！\n");
45      }else{
46              printf("SIGHUP信号没有被屏蔽了！\n");
47      }
48
49
50      // 取消对SIGQUIT的屏蔽
51      if(sigprocmask(SIG_SETMASK, &oldmask, NULL) < 0){
52              // 第一个参数为SIG_SETMASK表明设置进程新的信号屏蔽字为第二个参数执行的信号集
53              // 此时该进程的信号集为oldmask
54              printf("sigprocmask(SIG_SETMASK)失败！\n");
55              exit(1);
56      }
57
58      printf("sigprocmask(SIG_SETMASK)成功！\n");
59
60      if(sigismember(&oldmask, SIGQUIT)){
61              printf("SIGQUIT信号被屏蔽了！\n");
62      }else
63      {
64              printf("SIGQUIT信号没有被屏蔽了！您可以发送SIGQUIT信号了，我要sleep 10秒钟了！！\n");
65              int mysl = sleep(10);
66              if(mysl > 0){
67                      printf("sleep还没有睡够，剩余%d秒钟\n", mysl);
68              }
69      }
70
71      printf("再见了！\n");
72      return 0;
73  }
```

11. fork()函数

fork()函数用于创建进程，一个可执行程序执行1次就是1个进程，再执行一次就又是一个进程。

在一个进程中可以使用fork()函数创建一个子进程，当该子进程创建的时候，它从fork函数的下一条语句开始执行与父进程相同的代码。

即fork函数能产生一个和当前进程完全一样的新进程，程序就有了两个执行通路。

fork函数使用案例

```
1  #include <stdio.h>
```

```c
#include <stdlib.h>
#include <unistd.h>
#include <signal.h>

void sig_usr(int signo){
    //getpid()函数用来获取进程的id
    printf("收到了SIGUSR1信号，进程id=%d\n", getpid());
}

int main(int argc, char const *argv[])
{
    pid_t pid;
    printf("进程开始执行！\n");

    if(signal(SIGUSR1, sig_usr) == SIG_ERR){
            printf("无法捕捉SIGUSR1信号！\n");
            exit(1);
    }
    //创建一个新进程
    pid = fork();

    //判断子进程是否创建成功
    if(pid < 0){
            printf("子进程创建失败！\n");
            exit(1);
    }

    //现在父进程和子进程同时开始执行
    for(;;){
            sleep(1);
            printf("休息1s，进程id=%d\n", getpid());
    }
    printf("再见了！\n");

    return 0;
}
```

有fork行为的进程，程序员 应该拦截并处理SIGCHLD信号，否则杀死子进程的时候容易产生僵尸进程。

```c
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <signal.h>
#include <sys/wait.h>

void sig_usr(int signo){
    int status;
    switch(signo){
            case SIGUSR1:
                    printf("收到了SIGUSR1信号，进程id=%d\n", getpid());
                    break;
            case SIGCHLD:
                    printf("收到了SIGCHLD信号，进程id=%d\n", getpid());
                    //用waitpid获取子进程的终止状态，这样子进程就不会变成僵尸进程了
                    pid_t pid = waitpid(-1, &status, WNOHANG);
                    // -1表示等待任何子进程
```

```
                    // &status表示保存子进程的状态信息
                    // 提供额外选项，WNOHANG表示不要阻塞，让waitpid立即返回
                    if(pid == 0 || pid == 1){
                            // 0 表示子进程未结束
                            // 1 表示waitpid调用错误
                            return;
                    }
                    // 走到这里表示程序成功，程序返回
                    return;
                    break;
    }
    //getpid()函数用来获取进程的id
    printf("收到了SIGUSR1信号，进程id=%d\n", getpid());
}

int main(int argc, char const *argv[])
{
    pid_t pid;
    printf("进程开始执行！\n");

    if(signal(SIGUSR1, sig_usr) == SIG_ERR){
            printf("无法捕捉SIGUSR1信号！\n");
            exit(1);
    }
    if(signal(SIGCHLD, sig_usr) == SIG_ERR){
            printf("无法捕捉SIGCHLD信号！\n");
            exit(1);
    }
    //创建一个新进程
    pid = fork();

    //判断子进程是否创建成功
    if(pid < 0){
            printf("子进程创建失败！\n");
            exit(1);
    }

    //现在父进程和子进程同时开始执行
    for(;;){
            sleep(1);
            printf("休息1s，进程id=%d\n", getpid());
    }
    printf("再见了！\n");

    return 0;
}
```

fork函数在父进程和子进程中返回值是不同的，因此可以根据返回值识别出当前是父进程还是子进程。

```
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <signal.h>
#include <sys/wait.h>

int g_mygbltest = 0;
```

```
8
9  //防止出现僵尸进程
10 void sig_usr(int signo){
11     int status;
12     switch(signo){
13             case SIGUSR1:
14                     printf("收到了SIGUSR1信号，进程id=%d\n", getpid());
15                     break;
16             case SIGCHLD:
17                     printf("收到了SIGCHLD信号，进程id=%d\n", getpid());
18                     //用waitpid获取子进程的终止状态，这样子进程就不会变成僵尸进程了
19                     pid_t pid = waitpid(-1, &status, WNOHANG);
20                     // -1表示等待任何子进程
21                     // &status表示保存子进程的状态信息
22                     // 提供额外选项，WNOHANG表示不要阻塞，让waitpid立即返回
23                     if(pid == 0 || pid == 1){
24                             // 0 表示子进程未结束
25                             // 1 表示waitpid调用错误
26                             return;
27                     }
28                     // 走到这里表示程序成功，程序返回
29                     return;
30                     break;
31     }
32     //getpid()函数用来获取进程的id
33     printf("收到了SIGUSR1信号，进程id=%d\n", getpid());
34 }
35
36
37 int main(int argc, char const *argv[])
38 {
39     pid_t pid;
40     printf("进程开始执行！\n");
41
42     if(signal(SIGCHLD, sig_usr) == SIG_ERR){
43             printf("无法捕捉SIGCHLD信号！\n");
44     }
45     // 创建一个新进程，pid = 0则表示子进程创建成功，
46     // 也可以根据pid来划分子进程和父进程处理的代码
47     pid = fork();
48
49     //判断子进程是否创建成功
50     if(pid < 0){
51             printf("子进程创建失败！\n");
52             exit(1);
53     }
54
55     // 执行子进程
56     if(pid == 0)
57     {
58             for(int i = 0; i < 500; i++)
59             {
60                     g_mygbltest++;
61                     sleep(1);
```

```
                printf("我是子进程, id = %d, g_mygbltest = %d\n", getpid(), g_mygbltest);
        }
    }
    //执行父进程
    else
    {
        for(int i = 0; i < 500; i++)
        {
                g_mygbltest++;
                sleep(1);
                printf("我是父进程, id = %d, g_mygbltest = %d\n", getpid(), g_mygbltest);
        }
    }
    printf("g_mygbltest = %d\n", g_mygbltest); // 500

    return 0;
}
```

对于上述代码中的全局变量g_mygbltest，每个进程是单独计数的

12. fork失败的原因

- 系统中的进程太多
- 创建的进程数量超过了当前用户允许创建的最大数量

13. 守护进程

守护进程是一种长期在后台运行的进程，不与任何终端关联。

运行一个进程的时候在后面加&就是后台进程了。后台进程不同于守护进程

14. 守护进程的编写规则

- 调用umask(0),不限制文件权限
- fork一个子进程，然后父进程退出
- 文件描述符：文件描述符是非负整数，用于标识一个文件，打开或者创建一个文件的时候都会返回一个文件描述符。0表示标准输入STDIN_FILENO，1表示标准输出STDOUT_FILENO，2标准错误STDERR_FILENO
- 输入输出重定向：标准输出文件描述符不指向屏幕，而是重定向到其他文件中。ls -la > file 即是把ls -la的输出结果重定向到file文件中。cat < file，为输入重定向。
- 空设备：“/dev/null”称为空设备，可以吞噬一切写入数据。避免守护进程有输入和输出。
- dup2（）函数，像复制指针一样，将参数1指向的内容赋给参数2，使参数2也指向参数1所指向的内容。

守护进程代码（来源于nginx源码）

```
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <signal.h>
#include <sys/stat.h>
#include <fcntl.h>

//创建守护进程
int ngx_daemon(){
    int fd;
    switch(fork()){
        case -1:
            //进程创建失败
            return -1;
        case 0:
            // 子进程，走到这里直接break
            break;
        default:
            // 父进程直接退出
```

```
                exit(0);
    }

    // 只有子进程的流程才能走到这里
    if(setsid() == -1){
            return -1;
    }
    umask(0); // 设置为0，不要让它来限制文件权限

    fd = open("/dev/null", O_RDWR); // 以读写方式打开黑洞设备
    if(fd == -1){
            return -1;
    }
    if(dup2(fd, STDIN_FILENO) == -1){ // 关闭标准输入
            return -1;
    }
    if(dup2(fd, STDOUT_FILENO) == -1){ // 关闭标准输出
            return -1;
    }
    if(fd > STDERR_FILENO){
            if(close(fd) == -1){ // 释放资源
                    return -1;
            }
    }
    return 1;
}

int main(int argc, char const *argv[])
{
    /* code */
    if(ngx_daemon() != 1){
            //创建守护进程失败
            return 1;
    }
    else{
            for(;;){
                    sleep(1);
                    printf("休息1s，进程id = %d!\n", getpid());
            }
    }
    return 0;
}
```