1、信号是 Linux 系统中进程间的通信或操作的一种机制.

2、进程管理由进程控制块 、进程调度 、中断处理 、任务队列 、定时器 ，bottom half 队列 、进程通信等部分组成。

3、Linux 中一个可执行文件有三部分组成：代码区、全局初始化数据区/静态数据区（数据段）以及未初始化数据区。

4、进程组是一个或多个进程的集合。它们与同一作业相关联，可以接收来自同一终端的各种信号。每个进程组都有唯一的进程组号，进程组号是可以在用户层修改的。

5、fork 函数调用成功后，其子进程会复制父进程的几乎所 有信息（除 PID 等信息），包括父亲进程的代码段、数 据段、BSS（未初始化数据区）、堆、栈、打开的文件描述符表（但共用同 一个文件表项）。

6、为什么要使用 wait()函数

由于父子进程执行顺序的不确定性，当子进程先于父进程退出时，子进程会留下一些资源来记录运行的信息，以提供给父进程进行访问。

如果父进程没有调用 wait 或 waitpid 函数的话，则子进程将会一直保留这些信息，成为僵尸进程。如果父进程调用了 wait 函数，子进程就不会成为僵尸进程。

除了僵尸进程外，还会有孤儿进程。孤儿进程是指因父亲进程先结束而导致一个子进程被 init 进程收养的进程。

7、C 语言关键字与函数 exit()在 main 函数退出时有相似 之处，但两者有本质的区别：

return 退出当前函数主体，exit()函数退出当前进程，因 此，在 main 函数里面 return(0)和 exit(0)完成一样的功 能。

return 仅仅从子函数中返回，而子进程用 exit()退出，调 用 exit()时要调用一段终止处理程序，然后关闭所有 I/O 流。

8、进程调度机制主要涉及到**调度方式、调度时机和调度策略**。

9、守护进程（Daemon）是运行在后台，并且一直在运行的的一种特殊进程,周期性地执行某种任务或等待处理某些发生的事件。独立于终端，避免进程被任何终端所产生的信息所打断，执行过程中的信息也不在任何终端上显示。

Linux 的大多数服务器就是用守护进程实现的。比如，Internet 服务器 inetd，Web 服务器 httpd 等。

10、管道的实质是一个内核缓冲区，进程以先进先出的方式从缓冲区中存取数据：管道一端的进程顺序地将数据写入缓冲区，另一端的进程则顺序地读出数据。

管道实际上以类似文件的方式与进程交互，但它并不与磁盘打交道，所以效率要比文件操作高很多。

它有两个局限性：

（1）支持半双工；

（2）只有具有亲缘关系的进程之间才能使用这种无名管道；

使用管道的注意事项：

1.当读一个写端已经关闭的管道时，在所有数据被读取之后，read 函数返回值为 0，以指示到了文件结束处；

2.如果写一个读端关闭的管道，则产生 SIGPIPE 信号。如果忽略该信号或者捕捉该信号并处理程序返回，则 write 返回-1，errno 设置为 EPIPE

11、信号是 Linux 系统中用于进程之间异步通信的一种机制。信号机制除了基本通知功能外，还可以传递附加信息。

12、信号事件的发生有两个来源：

（1）硬件来源：用户按某些终端键时将产生信号，如 CTRL+C 将产生 SIGINT（中止信号）；硬件异常产生信号，如除数为 0 或无效的存储访问等。

（2）软件来源：终止进程信号，其他进程调用 kill 函数，将信号发送个另一个进程或进程组；软件异常产生信号。

13、**守护进程的实现：**

```
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <signal.h>
#include <fcntl.h>
#include <sys/syslog.h>
#include <sys/param.h>
#include <sys/types.h>
#include <sys/stat.h>
int init_daemon(const char *pname,int facility)  {
        int pid;        int i;
        signal(SIGTTOU,SIG_IGN);        signal(SIGTTIN,SIG_IGN);
        signal(SIGTSTP,SIG_IGN);        signal(SIGHUP,SIG_IGN);
        pid=fork();
        if(pid>0)
            exit(EXIT_SUCCESS);
        else if(pid<0)  {
            perror("fork");
            exit(EXIT_FAILURE);         }
        setsid();
        pid=fork();
        if(pid>0)               exit(EXIT_SUCCESS);
        else if(pid<0)  {
            perror("fork");             exit(EXIT_FAILURE); }
        for(i=0;i<NOFILE;i++)               close(i);
```

```
        open("/dev/null",O_RDONLY);
        open("/dev/null",O_RDWR);
        open("/dev/null",O_RDWR);
        chdir("/");
        umask(0);
        signal(SIGCHLD,SIG_IGN);        openlog(pname,LOG_PID,facility);
        return;
}
main(int argc,char * argv[])    {
        FILE *fp;
        time_t ticks;
        init_daemon(argv[0],LOG_KERN);
        while(1)            {
            sleep(1);
            ticks=time(NULL);
            syslog(LOG_INFO,"%s",asctime(localtime(&ticks)));
        }
}
```

14、**生成多个子进程：**

```
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
main()    {
   pid_t pid;
   int i=1;
   while(i<9)     {
        pid=fork();
        if(pid==0)              {
           printf("I'm no. %d process, my pid is %d.\n", i, getpid());
           if(i!=1)
              return 0;          }
        i++;
        }     }
```

15、**使用 exec 执行一段新代码：**

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

```
#include <sys/types.h>
int main(){
        pid_t result;
        result=fork();
        int newret;
        if(result==-1)    {
            perror("fork");
            exit(EXIT_FAILURE);          }

        else if(result==0)                    {
            printf("returen value is %d, this is child! pid=%d; ppid=%d.\n", result, getpid(),
getppid());
            execl("/bin/ls","ls","-l",NULL);  }
        else {
            sleep(3);
            printf("return value is %d, this is parent!pid=%d, ppid=%d.\n",result,
getpid(),getppid());
}}
```

16、**产生一个孤儿进程：**

```
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <sys/types.h>
int main ( )    {
   pid_t  pid;     int     k=3;
   pid = fork ( );
   switch ( pid )      {
       case -1: perror ( "fork error\n“ ); break;
       case 0 : while ( k > 0 )    {
                    printf ( "I'm child process, my PID is %d, my parent is
                                %d.\n",  getpid ( ), getppid ( ) );
                     sleep(1);          k--;}
                break;
       default : printf ( "I'm parent process, my PID is %d.\n", getpid ( ) );
                break;   }}
```

17、**父子进程通过无名管道通信：**

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

```
#include <stdlib.h>
#include <sys/types.h>
#include <sys/wait.h>
#include <unistd.h>
#include <string.h>
main()     {
                pid_t result;
                int r_num;
                int pipe_fd[2];
                char buf_r[100],buf_w[100];     memset(buf_r,0,sizeof(buf_r));
                if(pipe(pipe_fd)<0)                 {
                    perror("pipe");                 exit(EXIT_FAILURE); }
                result=fork();
                if(result<0)       {
                    perror("fork");                 exit(EXIT_FAILURE); }
                else if(result==0)     {
                    close(pipe_fd[1]);
                    if((r_num=read(pipe_fd[0],buf_r,100))>0)
                      printf("child process has read %d characters from the pipe,the string is:
%s\n",r_num,buf_r);
                    close(pipe_fd[0]);
                exit(0);            }
                if(pipe(pipe_fd)<0)  {
                    perror("pipe");                 exit(EXIT_FAILURE); }
                result=fork();
                if(result<0)       {
                    perror("fork");                 exit(EXIT_FAILURE); }
                else if(result==0)     {
                    close(pipe_fd[1]);
                    if((r_num=read(pipe_fd[0],buf_r,100))>0)
                      printf("child process has read %d characters from the pipe,the string is:
%s\n",r_num,buf_r);
                    close(pipe_fd[0]);
                exit(0);            }
```