第一次上机总结

```
int main()
{
  pid_t  pid;
   /* fork another process */
   pid = fork();
   if (pid < 0) { /* error occurred */
          fprintf(stderr, "Fork Failed");
          exit(-1);
   }
   else if (pid == 0) { /* child process */
          execlp(“/bin/ls", "ls", NULL);
   }
   else { /* parent process */
          /* parent will wait for the child to complete */
          wait (NULL);
          printf ("Child Complete");
          exit(0);
   }
}
```

fork函数（1）

- 一个进程，包括代码、数据和分配给进程的资源。fork（）函数通过系统调用创建一个与原来进程几乎完全相同的进程，也就是两个进程可以做完全相同的事，当然如果初始参数或者传入的变量不同，两个进程也可以做不同的事。

- 一个进程调用fork（）函数后，系统先给新的进程分配资源，例如存储数据和代码的空间。然后把原来的进程的所有值都复制到新的进程中，只有少数值与原来的进程的值不同。相当于克隆了一个自己。

fork函数（2）

- **fork的一个奇妙之处就是它仅仅被调用一次，却能够返回两次，它可能有三种不同的返回值：**
  1）在父进程中，fork返回新创建子进程的进程ID；
  2）在子进程中，fork返回0；
  3）如果出现错误，fork返回一个负值；

- **fork出错可能有两种原因**
  ① 当前的进程数已经达到了系统规定的上限，这时errno的值被设置为EAGAIN。
  ② 系统内存不足，这时errno的值被设置为ENOMEM。

三种资源拷贝方式

✓共享

共享同一资源，如虚存空间、文件等。仅增加有关描述符的用户计数器。

✓直接拷贝

相同的结构，原样复制。

✓COW

在需要的时候才复制。

创建进程的系统调用

提供三个创建进程的系统调用：

✓fork

用于普通进程的创建，采用COW方式。

✓vfork

完全共享的创建，共享同一资源。

✓clone

由用户指定创建的方式。

exec函数族（1）

- 实际上在**Linux**中，并不存在一个exec()的函数形式，exec指的是一组函数，一共有6个，分别是：

```
#include <unistd.h>

int execl(const char *path, const char *arg, ...);
int execlp(const char *file, const char *arg, ...);
int execle(const char *path, const char *arg, ..., char *const envp[]);
int execv(const char *path, char *const argv[]);
int execvp(const char *file, char *const argv[]);
int execve(const char *path, char *const argv[], char *const envp[]);
```

其中只有execve是真正意义上的系统调用，其它都是在此基础上经过包装的库函数

exec函数族（2）

- exec函数族的作用是**根据指定的文件名找到可执行文件，并用它来取代调用进程的内容**，换句话说，就是在调用进程内部执行一个可执行文件。这里的可执行文件既可以是二进制文件，也可以是任何Linux下可执行的脚本文件。

- 与一般情况不同，**exec函数族的函数执行成功后不会返回，因为调用进程的实体，包括代码段，数据段和堆栈等都已经被新的内容取代，**只留下进程ID等一些表面上的信息仍保持原样。

wait函数

- #include <sys/types.h>
  #include <sys/wait.h>
  pid_t wait(int *status)

- 进程一旦调用了wait，就立即阻塞自己，由wait自动分析是否当前进程的某个子进程已经退出，如果让它找到了这样一个已经变成僵尸的子进程，wait就会收集这个子进程的信息，并把它彻底销毁后返回；如果没有找到这样一个子进程，wait就会一直阻塞在这里，直到有一个出现为止。
- wait()要与fork()配套出现,如果在使用fork()之前调用wait(),wait()的返回值则为-1,正常情况下wait()的返回值为子进程的PID
- 当父进程没有使用wait()函数等待已终止的子进程时,子进程就会进入一种无父进程清理自己尸体的状态，此时的子进程就是僵尸进程，不能在内核中清理尸体的情况

- 相关函数waitpid()

LINUX 进程通信机制IPC简介

Linux IPC（1）

- **管道**（Pipe）及有名管道（named pipe）：管道可用于具有亲缘关系进程间的通信，有名管道克服了管道没有名字的限制，因此，除具有管道所具有的功能外，它还允许无亲缘关系进程间的通信

- **信号**（Signal）：信号是比较复杂的通信方式，用于通知接受进程有某种事件发生，除了用于进程间通信外，进程还可以发送信号给进程本身；linux除了支持Unix早期信号语义函数sigal外，还支持语义符合Posix.1标准的信号函数sigaction

- **消息队列**（报文队列）：消息队列是消息的链接表，包括Posix消息队列system V消息队列。有足够权限的进程可以向队列中添加消息，被赋予读权限的进程则可以读走队列中的消息。消息队列克服了信号承载信息量少，管道只能承载无格式字节流以及缓冲区大小受限等缺点

Linux IPC（2）

- **共享内存**：使得多个进程可以访问同一块内存空间，是最快的可用IPC形式。是针对其他通信机制运行效率较低而设计的。往往与其它通信机制，如信号量结合使用，来达到进程间的同步及互斥

- **信号量**（semaphore）：主要作为进程间以及同一进程不同线程之间的同步手段

- **套接字**（Socket）：更为一般的进程间通信机制，可用于不同机器之间的进程间通信。起初是由Unix系统的BSD分支开发出来的，但现在一般可以移植到其它类Unix系统上：Linux和System V的变种都支持套接字。

共享内存（1）

- 顾名思义，共享内存就是允许两个不相关的进程访问同一个逻辑内存。不同进程之间共享的内存通常安排为同一段物理内存。进程可以将同一段共享内存连接到它们自己的地址空间中，所有进程都可以访问共享内存中的地址，就好像它们是由用C语言函数malloc分配的内存一样。而如果某个进程向共享内存写入数据，所做的改动将立即影响到可以访问同一段共享内存的任何其他进程。
- **优点**：使用方便，函数的接口简单；数据的共享是直接访问内存，加快了程序的效率。同时，它也不像匿名管道那样要求通信的进程有一定的父子关系。
- **缺点**：共享内存没有提供同步的机制，这使得我们在使用共享内存进行进程间通信时，往往要借助其他的手段来进行进程间的同步工作。

共享内存（2）

- #include <sys/shm.h>
- 创建共享内存
  - **int** shmget(key_t key, **size_t** size, **int** shmflg);
- 把共享内存连接到当前进程的地址空间
  - **void *shmat(int shm_id, const void *shm_addr, int shmflg);**
- 将共享内存从当前进程中分离
  - **int shmdt(const void *shmaddr);**
- 控制共享内存
  - **int shmctl(int shm_id, int command, struct shmid_ds *buf);**

```
int main()
{
    int running = 1;//程序是否继续运行的标志
    void *shm = NULL;//分配的共享内存的原始首地址
    struct shared_use_st *shared;//指向shm
    int shmid;//共享内存标识符
    //创建共享内存
    shmid = shmget((key_t)1234, sizeof(struct shared_use_st), 0666|IPC_CREAT);
    if(shmid == -1)
    {
        fprintf(stderr, "shmget failed\n");
        exit(EXIT_FAILURE);
    }
    //将共享内存连接到当前进程的地址空间
    shm = shmat(shmid, 0, 0);
    if(shm == (void*)-1)
    {
        fprintf(stderr, "shmat failed\n");
        exit(EXIT_FAILURE);
    }
    printf("\nMemory attached at %X\n", (int)shm);
```

```
//设置共享内存
  shared = (struct shared_use_st*)shm;
  shared->written = 0;
  while(running)//读取共享内存中的数据
  {
    //没有进程向共享内存定数据有数据可读取
    if(shared->written != 0)
    {
      printf("You wrote: %s", shared->text);
      sleep(rand() % 3);
      //读取完数据，设置written使共享内存段可写
      shared->written = 0;
      //输入了end，退出循环（程序）
      if(strncmp(shared->text, "end", 3) == 0)
        running = 0;
    }
    else//有其他进程在写数据,不能读取数据
      sleep(1);
  }
  //把共享内存从当前进程中分离
  if(shmdt(shm) == -1)
  {
    fprintf(stderr, "shmdt failed\n");
    exit(EXIT_FAILURE);
  }
  //删除共享内存
  if(shmctl(shmid, IPC_RMID, 0) == -1)
  {
    fprintf(stderr, "shmctl(IPC_RMID) failed\n");
    exit(EXIT_FAILURE);
  }
  exit(EXIT_SUCCESS);
}
```

System V消息队列（1）

- 消息队列是由一条由消息连接而成的链表，是消息的链式队列，它保存在内核中，通过消息队列的引用标识符来访问。
- 信息被放置在一个预定义的消息结构中，进程生成的消息指明了该消息的类型，并把它放入一个由系统负责维护的消息队列中去。
- 访问消息队列的进程可以根据消息的类型，有选择地从队列中遵照FIFO原则读取特定类型的消息。

System V消息队列（2）

- 根据关键字生成标识符。
  - key_t ftok(const char *pathname, int proj_id);
- 打开或创建消息队列
  - int msgget(ket_t key, int msgflg);
- 向队列传递消息
  - int msgsnd(int msqid, struct msgbuf * msgp, size_t msgsz, int msgflg);
- 从队列中获得消息
  - int msgrcv(int msqid, struct msgbuf * msgq, size_t msgsz, long msgtype, int msgflg);
- 消息队列的属性控制
  - int msgctl (int msgqid, int cmd, struct msqid_ds *buf)

第二次上机实验

- 编写程序，完成如下功能：
  - 首先，使用System V消息队列机制，构建一个消息队列；
  - 分别生成两个进程，一个进程向生成的队列发消息，另一个进程从队列中取消息；
  - 运行期间，使用shell命令ipcs观测消息队列中的变化情况；
  - 最后，从内核中删除进程。