

操作系统实验要求

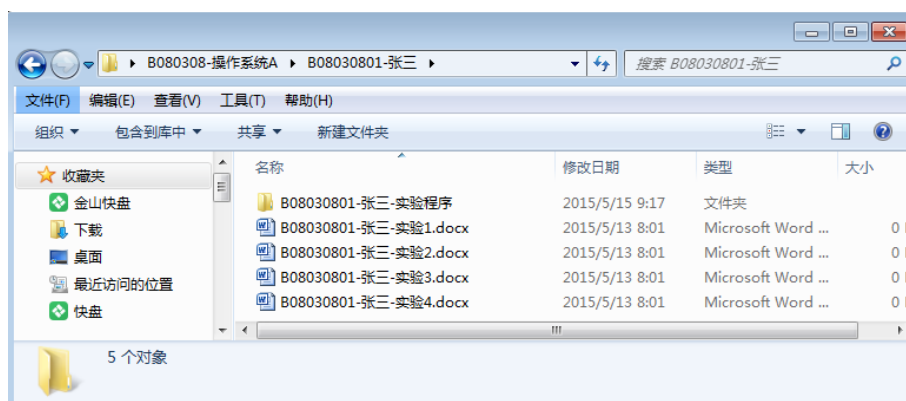
2015.4.23

2015.11.25

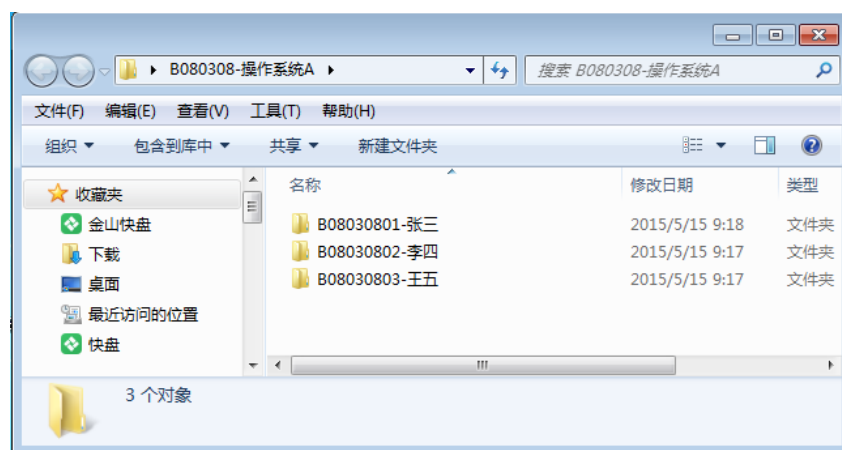
2018.04.27

实验报告提交方式

1. 只要提交电子档，班长负责整理。
2. 每位学生建立一个文件夹（起名如：B08030801-张三），文件夹中包括：
 - 1) 四份实验报告（起名如：B08030801-张三-实验 1.docx）
 - 2) 源程序文件夹（起名如：起名如 B08030801-张三-实验程序）
 - 3) 其它重要的电子档。



3. 每个班级建立一个文件夹（起名如：B080308-操作系统）



4. 班长将班级文件夹压缩后，发到邮箱 duanwh@126.com。（直接作为附件发送，不要使用 qq 文件中转站。）

实验 1 Linux 系统及进程创建

实验题目

Linux 系统及进程创建

实验目的

1. 掌握 Linux 操作系统的操作和使用;
2. 掌握 Linux 下 C 语言的编辑、编译、运行的全过程;
3. 掌握进程创建系统调用的使用。

实验内容

1. 熟悉 Linux 运行环境。
2. 学习 UNIX/LINUX 系统的 `pwd`, `ls`, `cd`, `ps`, `cp`, `kill` 等命令，查看运行结果。
3. 掌握 C 语言的编辑、编译、运行的全过程，掌握进程创建系统调用的使用。
 - a) 使用 `vi`（全屏幕编辑器）或者文本编辑器编写 C 语言文件。
 - b) 使用 `gcc`（C 语言编译器）编译 C 语言文件。
 - c) 运行编译生成的文件。
4. 设计一个程序，在掌握进程创建 `fork()` 系统调用的使用，在进程中创建若干子进程。在主进程、子进程中制定它们的运行先后关系，分析各个进程中运行情况。

实验 2 进程通信

实验题目

进程通信

实验目的

1. 熟练使用 Linux 的 C 语言开发环境
2. 掌握 Linux 操作系统下的并发进程间同步
3. 掌握 Linux 操作系统下的进程间通信

实验内容

1. 了解常见的消息通信方式：信号机制、消息队列机制、共享内存机制和管道机制。
2. 掌握消息队列机制中常用的系统调用有：建立一个消息队列 `msgget`；向消息队列发送消息 `msgsnd`；从消息队列接收消息 `msgrcv`；取或送消息队列控制信息 `msgctl`。
3. 掌握管道机制中常用的系统调用：建立管道文件 `pipe`；写操作 `write`，读操作 `read`。
4. 了解信号机制中常用的系统调用。
5. 了解共享内存机制中常用的系统调用。

实验 3 页面调度算法模拟

实验题目

页面调度算法模拟

实验目的

页面调度算法主要有：FIFO，最近最少使用调度算法（LRU），最佳算法（OPT）。设计并编写程序模拟以上算法。

实验内容

如有问题，请联系 duanwh@126.com

1. 掌握 FIFO, LRU 和 OPT 三种算法的原理。
2. 选择合适的存储结构，分别实现这三种算法。给定一个页面请求序列，给定系统所分配的物理块数后，给出分别采用这三种算法进行调度时产生的缺页次数以及缺页率。

实验 4 文件系统的模拟

实验题目

文件系统的模拟

实验目的

1. 掌握文件系统管理系统的基本原理。
2. 掌握常见的文件操作的系统调用。
3. 设计一个多用户文件系统，模拟文件管理的工作过程。

实验内容

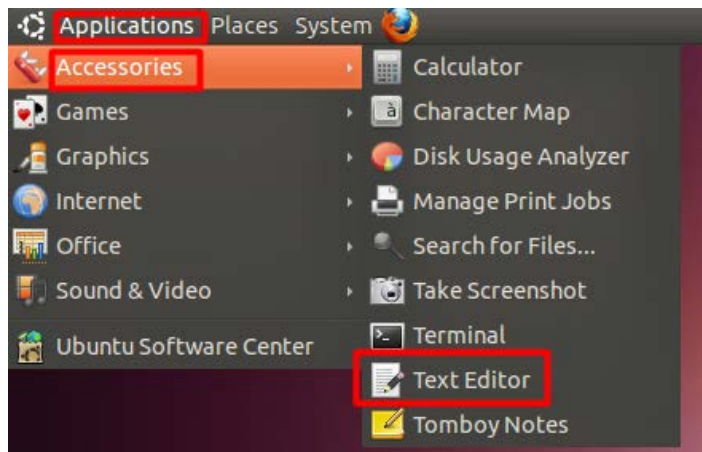
1. 实现一个文本文件的某信息的插入和删除。
2. 设计一个程序，实现文件复制命令。(cp)
3. 设计一个程序，实现文件目录的显示命令。(ls)
4. 设计一个多用户文件系统，模拟文件管理的工作过程。

实验参考程序

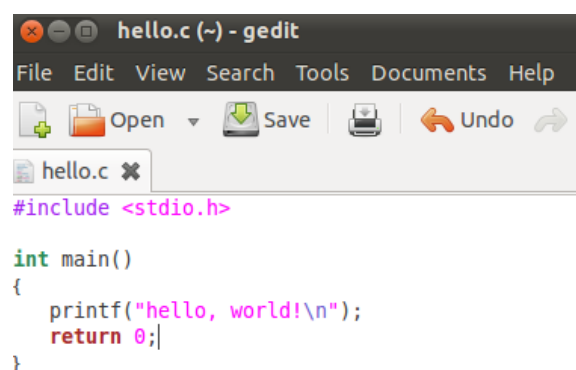
实验 1 Linux 系统及进程创建

1. C 语言编辑

(1) 使用附件中的文本编辑器编辑程序



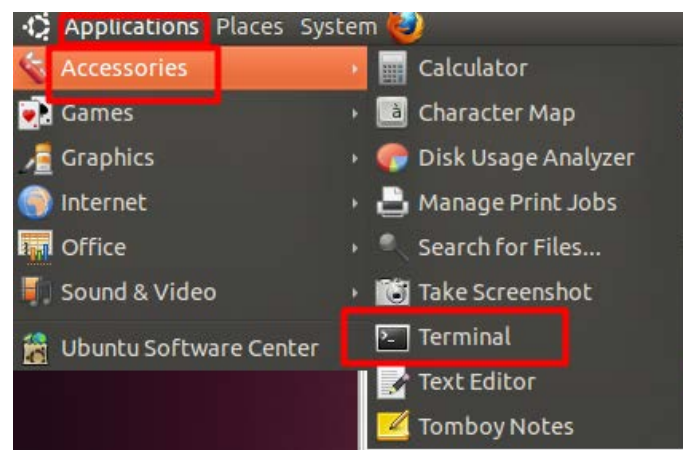
(2) 编辑最简单的 hello world 程序

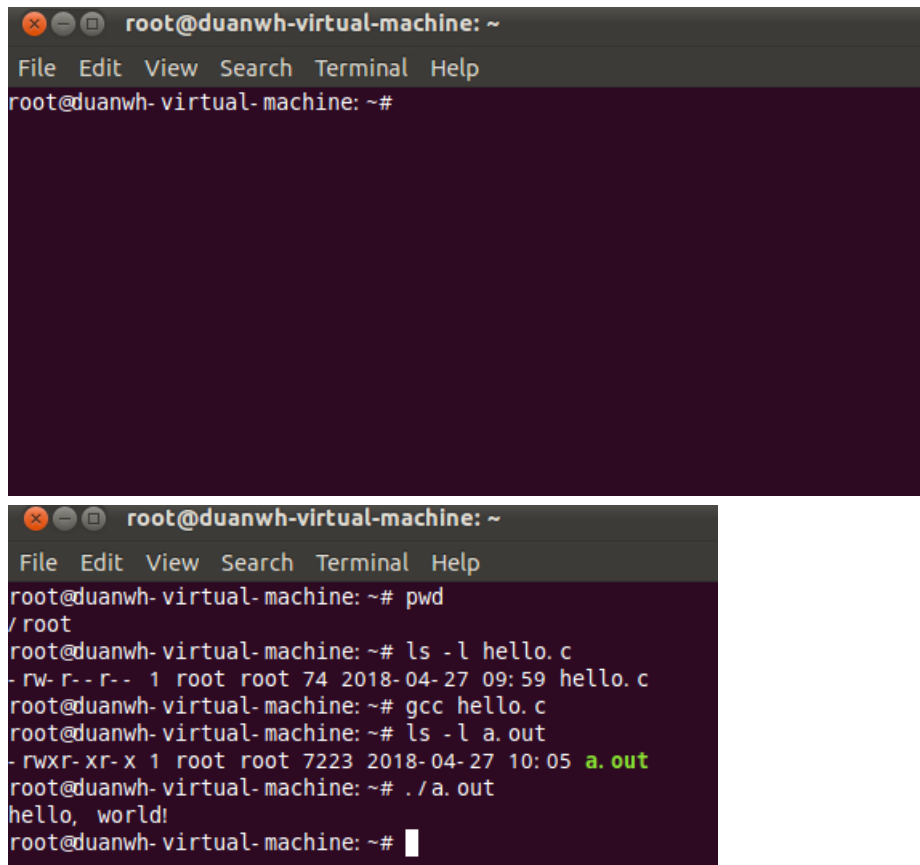


(3) 保存为 hello.c 源文件。

2. C 语言的编译和运行

(1) 进入终端





The image shows two screenshots of a terminal window. The top screenshot shows the terminal title 'root@duanwh-virtual-machine: ~' and a menu bar with 'File Edit View Search Terminal Help'. The bottom screenshot shows the same terminal with the following commands and output:

```
root@duanwh-virtual-machine: ~# pwd
/root
root@duanwh-virtual-machine: ~# ls -l hello.c
-rw-r--r-- 1 root root 74 2018-04-27 09:59 hello.c
root@duanwh-virtual-machine: ~# gcc hello.c
root@duanwh-virtual-machine: ~# ls -l a.out
-rwxr-xr-x 1 root root 7223 2018-04-27 10:05 a.out
root@duanwh-virtual-machine: ~# ./a.out
hello, world!
root@duanwh-virtual-machine: ~#
```

(1) 使用 `pwd` 查看当前路径

`pwd`

(2) 使用 `ls` 查看文件 `hello.c`。

`ls -l hello.c`

(3) 使用 `gcc` 对源文件进行编译。

`gcc hello.c`

(4) 使用 `ls` 查看生成的可执行文件 `a.out`

`ls -l a.out`

(5) 执行 `a.out`，查看执行结果。

`./a.out`

注意：

(1) 编译时没有使用 `-o` 选项指定可执行文件的文件名，确实生成的可执行文件是 `a.out`

(2) 可以使用 `-o` 选项指定可执行文件的文件名

`gcc hello.c -o hello.out`

指定的可执行文件的文件名是 `hello.out`

(3) 执行文件时必须显示的指明是当前文件夹下的可执行文件，也就是在 `a.out` 前必须加上 `./`。这里 `./` 表示是当前文件夹下的。

3. 进程创建 1

`/*fork1.c*/`

`#include <stdio.h>`

```
int main()
{
    int p1, p2, i;

    while ((p1 = fork()) == -1); /*创建进程 p1*/

    if(p1 == 0)
    {
        for (i = 0; i < 8; i++)
        {
            printf("daughter %d \n", i);
        }
    }
    else
    {
        while ((p2 = fork()) == -1); /*创建进程 p2*/
        if (p2 == 0)
        {
            for (i = 0; i < 8; i++)
            {
                printf("son %d \n", i);
            }
        }
        else
        {
            for (i = 0; i < 8; i++)
            {
                printf("parent %d \n", i);
            }
        }
    }
}
```

运行结果

```
root@duanwh-virtual-machine: ~# ./fork1.out
parent 0
parent 1
parent 2
parent 3
parent 4
parent 5
parent 6
parent 7
root@duanwh-virtual-machine: ~# son 0
son 1
son 2
son 3
son 4
son 5
son 6
son 7
daughter 0
daughter 1
daughter 2
daughter 3
daughter 4
daughter 5
daughter 6
daughter 7
```

执行结果中打印出来的语句没有交叉，好像三个进程是顺序执行，而不是并发执行。

4. 进程创建 2，增加 `sleep` 语句使得打印结果有交叉。

`/*fork2.c*/`

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

```
int main()
```

```
{
```

```
    int p1, p2, i;
```

```
    while ((p1 = fork()) == -1); /*创建进程 p1*/
```

```
    if(p1 == 0)
```

```
    {
```

```
        for (i = 0; i < 8; i++)
```

```
        {
```

```
            printf("daughter %d \n", i);
```

```
            sleep(4);
```

```
        }
```

```
    }
```

```
    else
```

```
    {
```

```
        while ((p2 = fork()) == -1); /*创建进程 p2*/
```

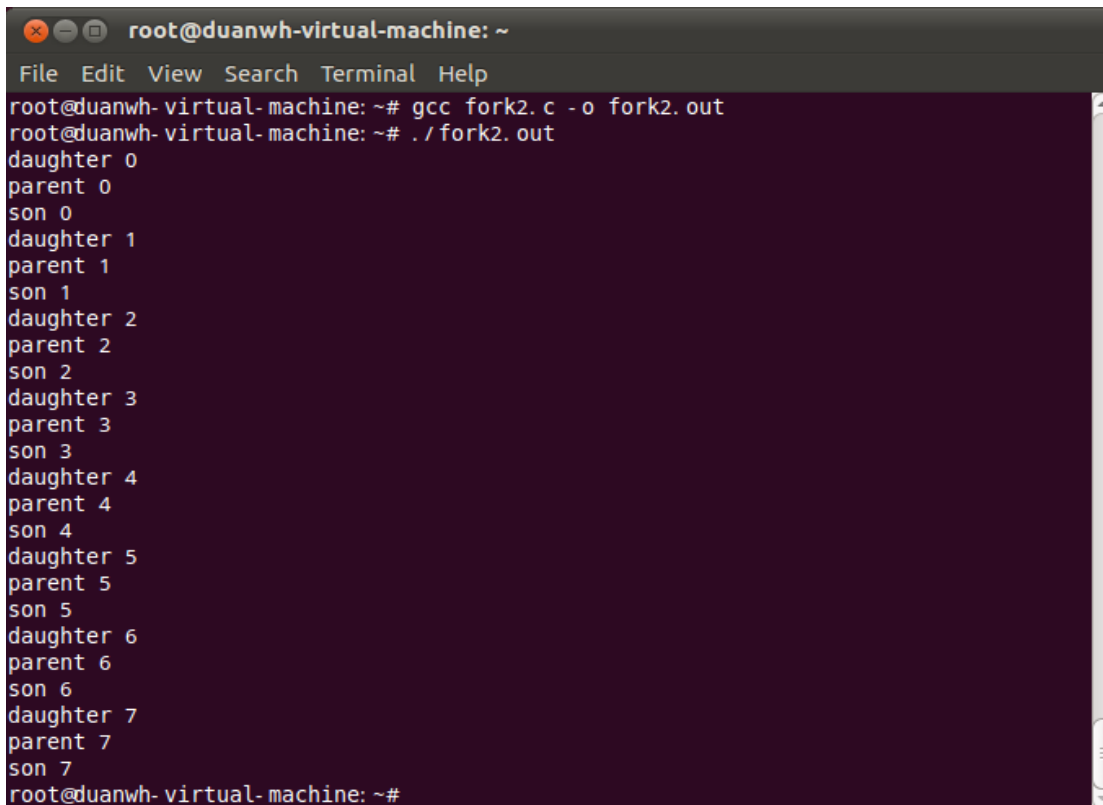
```
        if (p2 == 0)
```

```
        {
```

```
        for (i = 0; i < 8; i++)
        {
            printf("son %d \n", i);
            sleep(4);
        }
    }
    else
    {
        for (i = 0; i < 8; i++)
        {
            printf("parent %d \n", i);
            sleep(4);
        }
    }

    return 0;
}
```

程序运行结果



```
root@duanwh-virtual-machine: ~
File Edit View Search Terminal Help
root@duanwh-virtual-machine: ~# gcc fork2.c -o fork2.out
root@duanwh-virtual-machine: ~# ./fork2.out
daughter 0
parent 0
son 0
daughter 1
parent 1
son 1
daughter 2
parent 2
son 2
daughter 3
parent 3
son 3
daughter 4
parent 4
son 4
daughter 5
parent 5
son 5
daughter 6
parent 6
son 6
daughter 7
parent 7
son 7
root@duanwh-virtual-machine: ~#
```

5. 增加代码，判断以下程序的执行产生多少个进程

```
#include <stdio.h>
int main()
```



```
{  
    fork();  
    fork();  
    fork();  
    return 0;  
}
```

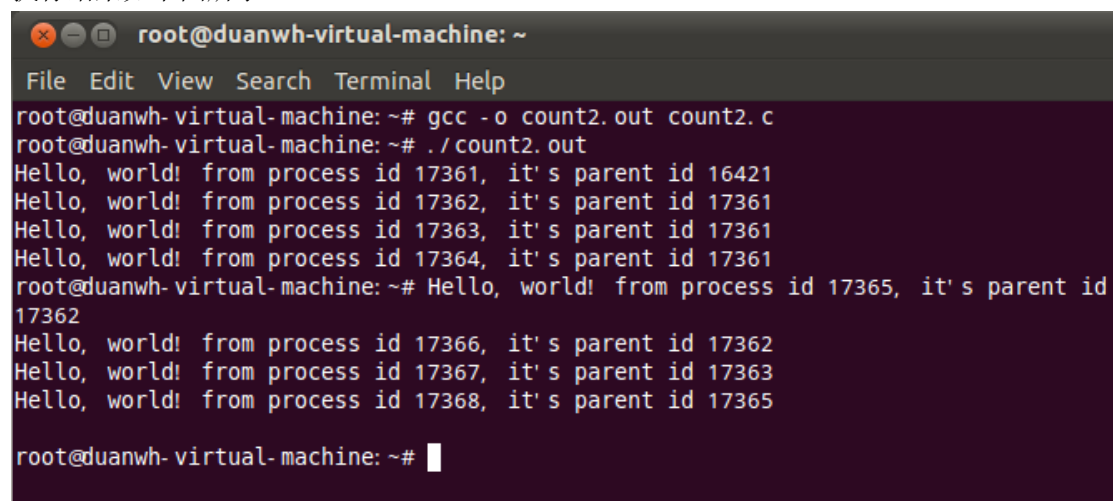
试着使用两个系统调用

- (1) 获得当前的进程号 `getpid()`;
- (2) 获得父进程的进程号 `getppid()`;
- (3) 等待子进程结束 `wait()`;
- (4) 在创建多个进程后，执行的过程中可能会产生孤儿进程，即父进程先于子进程之前结束。在这种情况下，系统将子进程的父进程设定为 `init` 进程（进程号为 1）。
- (5) 要解决孤儿进程问题，可以在父进程中增加 `wait` 系统调用。

参考程序：

```
/*count2.c*/  
#include <stdlib.h>  
#include <stdio.h>  
  
int main()  
{  
    fork();  
    fork();  
    fork();  
    printf("Hello, world! from process id %d, it's parent id %d\n", getpid(), getppid());  
    wait(0);  
    wait(0);  
    wait(0);  
    return 0;  
}
```

执行结果如下图所示。



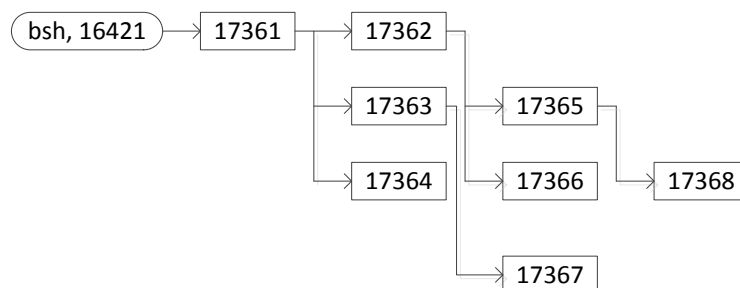
```
root@duanwh-virtual-machine: ~  
File Edit View Search Terminal Help  
root@duanwh-virtual-machine: ~# gcc -o count2.out count2.c  
root@duanwh-virtual-machine: ~# ./count2.out  
Hello, world! from process id 17361, it's parent id 16421  
Hello, world! from process id 17362, it's parent id 17361  
Hello, world! from process id 17363, it's parent id 17361  
Hello, world! from process id 17364, it's parent id 17361  
root@duanwh-virtual-machine: ~# Hello, world! from process id 17365, it's parent id  
17362  
Hello, world! from process id 17366, it's parent id 17362  
Hello, world! from process id 17367, it's parent id 17363  
Hello, world! from process id 17368, it's parent id 17365  
root@duanwh-virtual-machine: ~#
```

可以画出这 8 个进程的进程关系树。

```
root@duanwh-virtual-machine:~# ./count2.out
Hello, world! from process id 17361, it's parent id 16421
Hello, world! from process id 17362, it's parent id 17361
Hello, world! from process id 17363, it's parent id 17361
Hello, world! from process id 17364, it's parent id 17361
root@duanwh-virtual-machine:~# Hello, world! from process id 17365, it's parent id 17362
Hello, world! from process id 17366, it's parent id 17362
Hello, world! from process id 17367, it's parent id 17363
Hello, world! from process id 17368, it's parent id 17365
```

进程号	父进程号
17361	16421
17362	17361
17363	17361
17364	17361
17365	17362
17366	17362
17367	17363
17368	17365

16421 是 bash 的进程号, 16421 创建了第一个进程 17361; 17361 创建了 17362, 17363, 17364 三个进程; 17362 进程创建了 17365 和 17366 两个进程; 17363 创建了 17367 一个进程; 17365 创建了 17368 一个进程。



6. 在父进程中使用 `fork` 创建一个子进程，子进程拷贝了父进程的代码段和数据段。系统还提供一个 `vfork` 也是创建一个子进程，子进程共享父进程的代码段和数据段。`fork` 后父子进程的执行次序不确定。`vfork` 保证子进程先运行，在调用 `exec` 或 `exit` 之前与父进程数据是共享的，在它调用 `exec` 或 `exit` 之后父进程才可能被调度运行。

7. 实验总结

- a) 常用的命令行
- b) 熟悉 Linux 下的 C 语言编辑、编译和运行。
- c) 特殊进程
 - i. 父进程
 - ii. 子进程
 - iii. `init` 进程（1 号进程）

- iv. 守护进程（幽灵进程，daemon）
- v. 孤儿进程
- vi. 僵尸进程（zombie）

8. 以下的程序执行会产生孤儿进程。

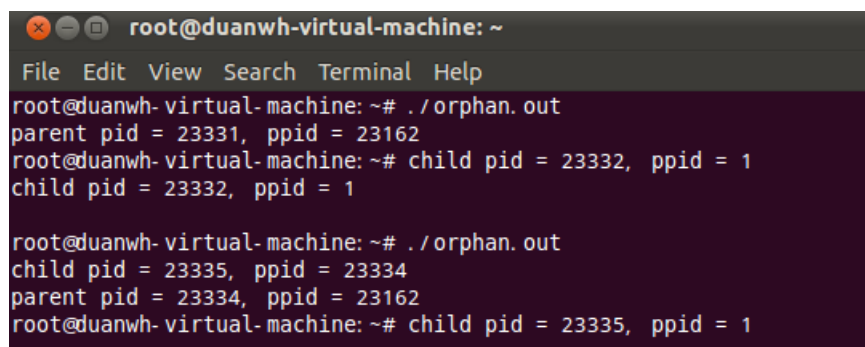
```
#include <stdio.h>
```

```
int main()
{
    int pid;

    pid = fork();

    if (pid == 0)
    {
        printf("child pid = %d, ppid = %d\n", getpid(), getppid());
        sleep(2);
        printf("child pid = %d, ppid = %d\n", getpid(), getppid());
    }
    else
    {
        printf("parent pid = %d, ppid = %d\n", getpid(), getppid());
    }
    return 0;
}
```

执行结果



```
root@duanwh-virtual-machine: ~
File Edit View Search Terminal Help
root@duanwh-virtual-machine: ~# ./orphan.out
parent pid = 23331, ppid = 23162
root@duanwh-virtual-machine: ~# child pid = 23332, ppid = 1
child pid = 23332, ppid = 1

root@duanwh-virtual-machine: ~# ./orphan.out
child pid = 23335, ppid = 23334
parent pid = 23334, ppid = 23162
root@duanwh-virtual-machine: ~# child pid = 23335, ppid = 1
```

执行结果分析：

（1）执行了两次，第一次是 **parent** 先打印，第二次是 **child** 进程先打印。两次的执行结果不同，这个就体现了操作系统基本特性中的异步性。

（2）分析第一次执行结果。（父进程的进程号是 **23331**，子进程的进程号是 **23332**）

1）父进程先打印，打印结束后，父进程结束。

2）子进程打印第一次的时候已经是孤儿进程了。子进程的两次打印结果是一样的。

（3）分析第二次执行结果。（父进程的进程号是 **23334**，子进程的进程号是 **23335**）

1）子进程先打印第一次，此时父进程没有结束，子进程不是孤儿进程，打印出了子进程的进程号和父进程的进程号。子进程执行 **sleep** 语句进入阻塞状态。

2) 父进程执行，打印自身的进程号和自己的父进程的进程号。父进程结束。

3) 子进程等待时间到了后继续执行，打印第二句话，此时父进程已经结束，子进程变成了孤儿进程。

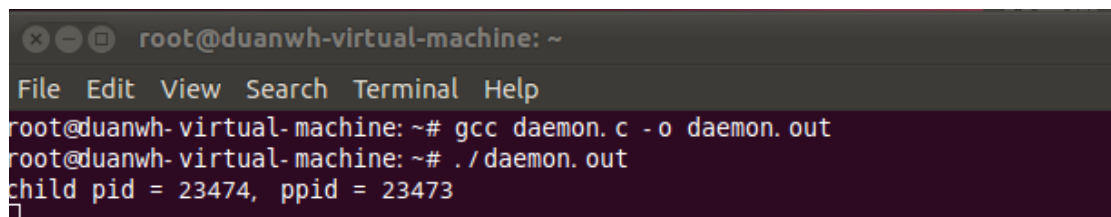
9. 以下的程序执行会出现僵尸进程。

```
/*daemon.c*/
#include <stdio.h>

int main()
{
    int pid;

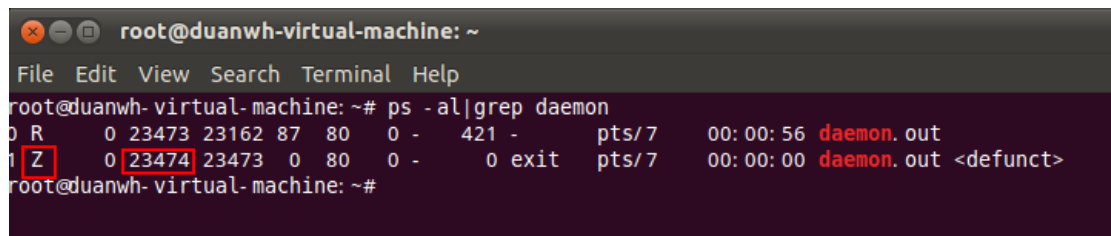
    pid = fork();

    if (pid > 0)
    {
        while(1)
        {
            //nothing
        }
    }
    else
    {
        printf("child pid = %d, ppid = %d\n", getpid(), getppid());
    }
    return 0;
}
```



```
root@duanwh-virtual-machine: ~
File Edit View Search Terminal Help
root@duanwh-virtual-machine: ~# gcc daemon.c -o daemon.out
root@duanwh-virtual-machine: ~# ./daemon.out
child pid = 23474, ppid = 23473
```

另打开一个终端，使用 `ps -allgrep daemo` 可以看到子进程是僵尸状态。



```
root@duanwh-virtual-machine: ~
File Edit View Search Terminal Help
root@duanwh-virtual-machine: ~# ps -allgrep daemon
0 R   0 23473 23162 87  80   0 -    421 -      pts/7    00:00:56 daemon.out
1 Z   0 23474 23473  0  80   0 -     0 exit   pts/7    00:00:00 daemon.out <defunct>
root@duanwh-virtual-machine: ~#
```

执行结果分析：

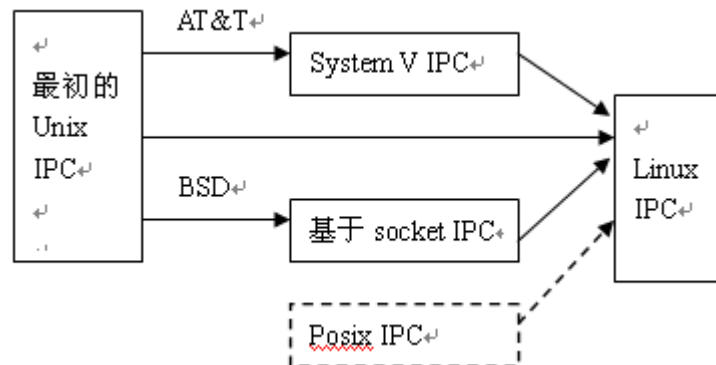
(1) 涉及到两个进程，父进程的进程号 23473，子进程的进程号 23474。

(2) 父进程中执行空的 `while` 语句，子进程打印完一句话就结束，进入僵尸状态等待父进程处理。而父进程一直处于无限循环状态，没有去处理。

如有问题，请联系 duanwh@126.com

实验 2 进程通信

1. Linux 进程通信



- (1) 最初的 UNIX 最初 Unix IPC 包括：管道、FIFO、信号；
- (2) System V IPC 主要局限在单个计算机中，包括：System V 消息队列、System V 信号灯、System V 共享内存区；
- (3) BSD Socket 通信主要侧重于不同计算机之间的网络通信。
- (4) Posix IPC 是电子电气工程协会（IEEE）开发了一个独立的 Unix 标准现有大部分 Unix 和流行版本都是遵循 POSIX 标准的，而 Linux 从一开始就遵循 POSIX 标准；Posix IPC 包括：Posix 消息队列、Posix 信号灯、Posix 共享内存区。

2. 共享内存的通信方式

- (1) 有两个程序，一个是发送，一个是接收。

发送进程的代码如下：

/*共享内存的发送程序 sndshm.c，先运行发送程序，再运行接收程序*/

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <unistd.h>
#include <sys/types.h>
#include <sys/shm.h>

int main()
{
    int shmid; /*共享内存的内部标识*/
    char *viraddr; /*定义附接到共享内存的虚拟地址*/
    char buffer[BUFSIZ];

    /*创建共享内存*/
    shmid = shmget(1234, BUFSIZ, 0666|IPC_CREAT);
    /*附接到进程的虚拟地址空间*/
    viraddr = (char *)shmat(shmid, 0, 0);
```

```
/*循环输入信息，直到输入 end 结束*/
while(1)
{
    puts("Enter some text:");
    fgets(buffer, BUFSIZ, stdin);
    strcat(viraddr, buffer);    /*追加到共享内存*/

    if(strncmp(buffer, "end", 3) == 0)
        break;
}
shmdt(viraddr);                /*断开链接*/

return 0;
}
接收进程的代码如下：
/*共享内存的接收进程程序 rcvshm.c*/
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <unistd.h>
#include <sys/types.h>
#include <sys/shm.h>

int main()
{
    int shmid;
    char *viraddr;

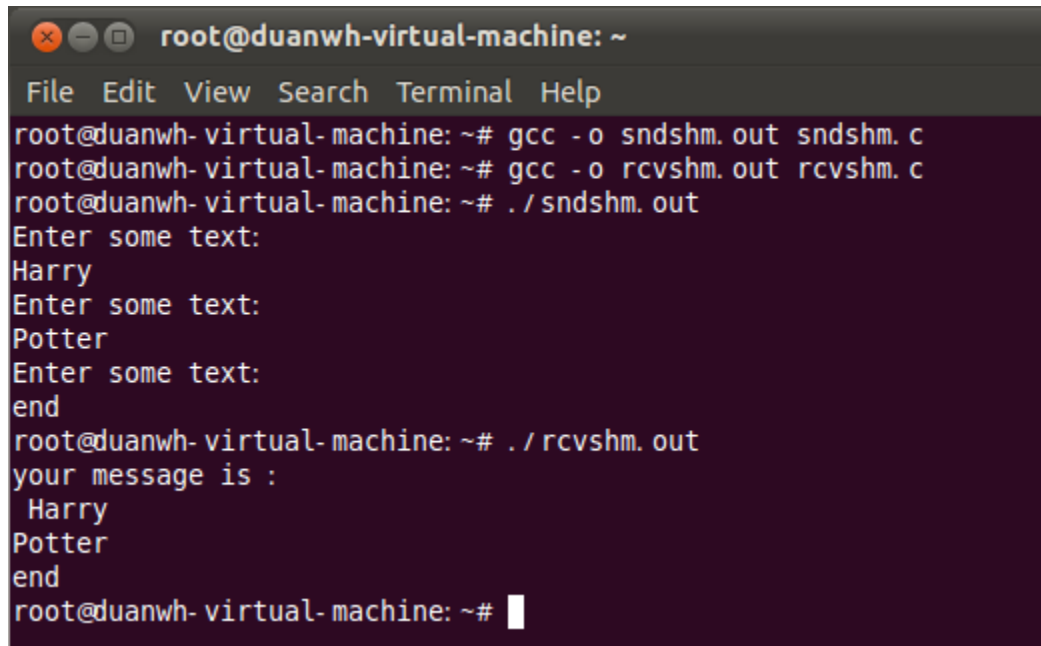
    /*获取共享内存*/
    shmid = shmget(1234, BUFSIZ, 0666|IPC_CREAT);
    /*附接到进程的虚拟地址空间*/
    viraddr = (char *)shmat(shmid, 0, 0);

    /*打印信息内容*/
    printf("your message is :\n %s", viraddr);

    /*断开链接*/
    shmdt(viraddr);
    /*撤销共享内存*/
    shmctl(shmid, IPC_RMID, 0);

    return 0;
}
```

(2) 编译运行两个程序，查看运行结果。务必先执行 `sndshm.out`，再执行 `rcvshm.out`。



```
root@duanwh-virtual-machine: ~  
File Edit View Search Terminal Help  
root@duanwh-virtual-machine: ~# gcc -o sndshm.out sndshm.c  
root@duanwh-virtual-machine: ~# gcc -o rcvshm.out rcvshm.c  
root@duanwh-virtual-machine: ~# ./sndshm.out  
Enter some text:  
Harry  
Enter some text:  
Potter  
Enter some text:  
end  
root@duanwh-virtual-machine: ~# ./rcvshm.out  
your message is :  
Harry  
Potter  
end  
root@duanwh-virtual-machine: ~#
```

(3) 共享存储区通信方式涉及到的系统调用

shmget
shmat
shmdt
shmctl

3. 消息队列的通信方式

(1) 消息队列通信方式有两个程序，一个负责发送，另一个负责接收。

发送进程代码：

```
/*发送消息进程 sndfile.c*/  
#include <stdio.h>  
#include <stdlib.h>  
#include <string.h>  
#include <unistd.h>  
#include <sys/types.h>  
#include <sys/msg.h>  
  
#define MAXMSG 512 /*定义消息长度*/  
  
/*定义消息缓冲区队列中的数据结构*/  
struct my_msg  
{  
    long int my_msg_type;  
    char some_text[MAXMSG];  
}msg;
```



```
int main()
{
    int msgid;          /*定义消息缓冲区内部标识*/
    char buffer[BUFSIZ]; /*用户缓冲区*/

    /*创建消息队列*/
    msgid = msgget(1234, 0666|IPC_CREAT);

    /*循环向消息队列中发送消息，直到输入 end 结束*/
    while(1)
    {
        puts("Enter some text:");
        fgets(buffer, BUFSIZ, stdin);
        msg.my_msg_type = 1;
        strcpy(msg.some_text, buffer);
        msgsnd(msgid, &msg, MAXMSG, 0); /*发送消息到缓冲队列中*/

        if (strncmp(msg.some_text, "end", 3) == 0)
            break;
    }

    return 0;
}
```

接收进程代码:

/*消息队列机制的接收程序 rcvfile.c*/

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

```
#include <sys/types.h>
#include <sys/msg.h>
```

```
#define MAXMSG 512
```

```
struct my_msg
{
    long int my_msg_type;
    char some_text[MAXMSG];
}msg;
```

```
int main()
{
```

如有问题，请联系 duanwh@126.com

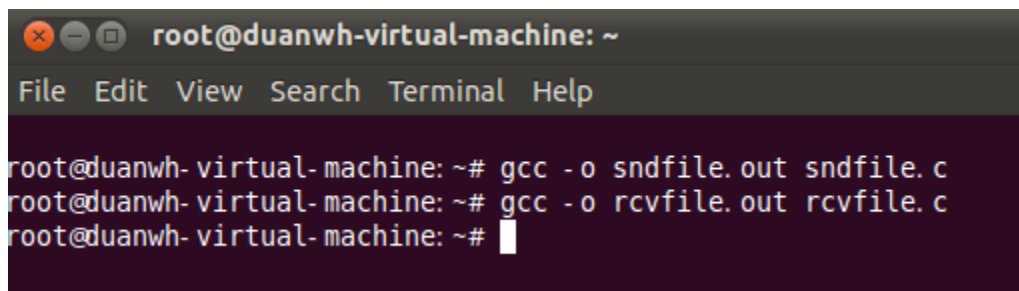
```
int msgid;
long int msg_to_receive = 0;

msgid = msgget(1234, 0666|IPC_CREAT);
/*循环从消息队列中接收消息，读入 end 结束接收*/
while (1)
{
    msgrcv(msgid, &msg, BUFSIZ, msg_to_receive, 0);
    printf("You wrote:%s", msg.some_text);

    if (strncmp(msg.some_text, "end", 3) == 0)
        break;
}

msgctl(msgid, IPC_RMID, 0);
return 0;
}
```

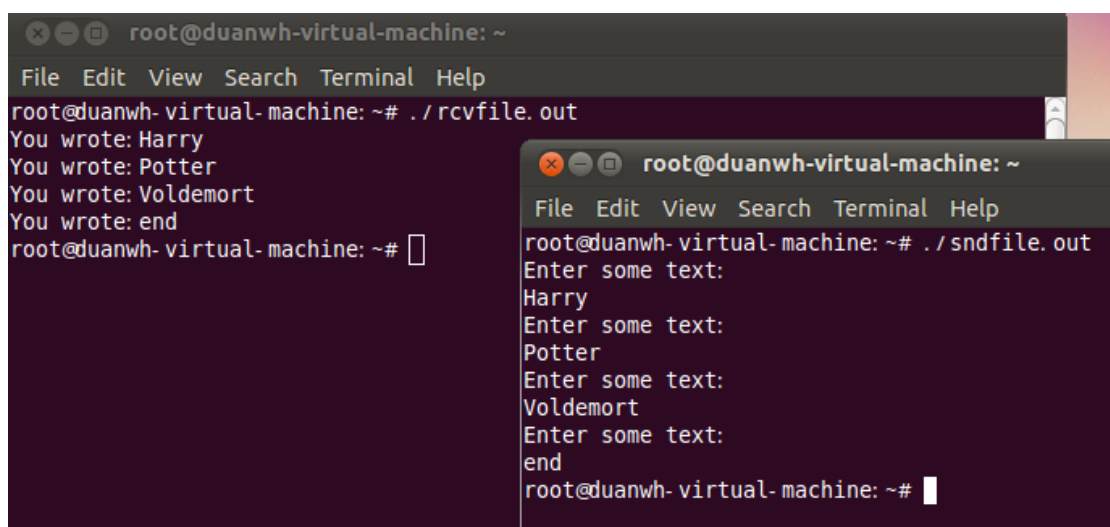
(2) 编译两个程序。



```
root@duanwh-virtual-machine: ~
File Edit View Search Terminal Help

root@duanwh-virtual-machine: ~# gcc -o sndfile.out sndfile.c
root@duanwh-virtual-machine: ~# gcc -o rcvfile.out rcvfile.c
root@duanwh-virtual-machine: ~#
```

运行两个程序查看运行结果。开启两个终端，一个终端先运行 `sndfile.out`，再在另一个终端中运行 `rcvfile.out`。



```
root@duanwh-virtual-machine: ~
File Edit View Search Terminal Help

root@duanwh-virtual-machine: ~# ./rcvfile.out
You wrote: Harry
You wrote: Potter
You wrote: Voldemort
You wrote: end
root@duanwh-virtual-machine: ~#

root@duanwh-virtual-machine: ~
File Edit View Search Terminal Help

root@duanwh-virtual-machine: ~# ./sndfile.out
Enter some text:
Harry
Enter some text:
Potter
Enter some text:
Voldemort
Enter some text:
end
root@duanwh-virtual-machine: ~#
```

在 `sndfile.out` 所在的终端不停的输入消息，可以看到 `rcvfile.out` 所在终端接收到消息。

(3) 消息队列涉及到的系统调用。

```
msgget
msgsnd
msgrcv
msgctl
```

4. 管道通信

(1) 程序

/*管道文件 pipe.c*/

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

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

```
int main()
```

```
{
```

```
    int p1, fd[2];
```

```
    char outpipe[50];          /*定义读缓冲区*/
```

```
    char inpipe[50] = "This is a message from child!";    /*定义写缓冲区*/
```

```
    pipe(fd);
```

```
    while ((p1 = fork()) == -1);
```

```
    if (p1 == 0)                /*子进程中写*/
```

```
    {
```

```
        write(fd[1], inpipe, 50);
```

```
    }
```

```
    else                        /*父进程中读*/
```

```
    {
```

```
        wait(0);
```

```
        read(fd[0], outpipe, 50);
```

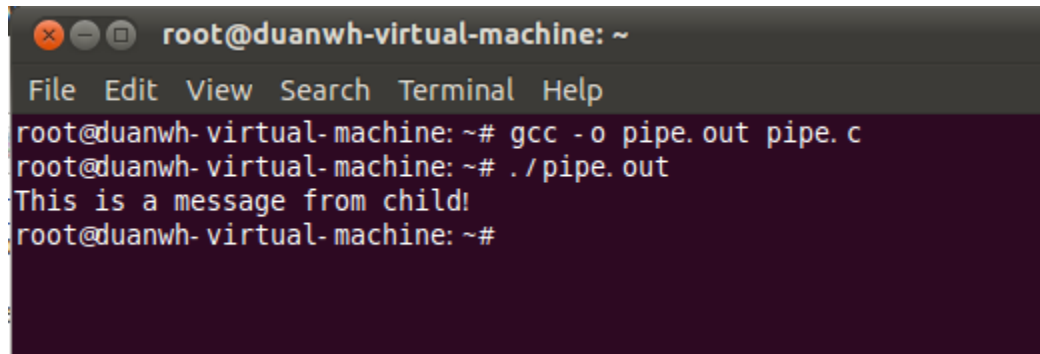
```
        printf("%s \n", outpipe);
```

```
    }
```

```
    return 0;
```

```
}
```

(2) 编译运行。



```
root@duanwh-virtual-machine: ~  
File Edit View Search Terminal Help  
root@duanwh-virtual-machine: ~# gcc -o pipe.out pipe.c  
root@duanwh-virtual-machine: ~# ./pipe.out  
This is a message from child!  
root@duanwh-virtual-machine: ~#
```

5. 信号机制

(1) 程序

第一个程序有信号处理机制

/*signal.c*/

```
#include <stdio.h>  
#include <unistd.h>  
#include <signal.h>  
  
void int_func(int sig);  
int k;          /*定义循环变量*/  
  
void int_func(int sig)  
{  
    k = 0;  
}  
  
int main()  
{  
    signal(SIGINT, int_func);  
    k = 1;  
  
    while (k == 1)  
    {  
        printf("Hello, world!\n");  
    }  
  
    printf("OK!\n");  
    printf("pid:  %d,  ppid: %d \n", getpid(), getppid());  
}
```

(2) 编译运行:

如有问题，请联系 duanwh@126.com

```
root@duanwh-virtual-machine: ~  
File Edit View Search Terminal Help  
root@duanwh-virtual-machine: ~# gcc signal.c -o signal.out  
root@duanwh-virtual-machine: ~# ./signal.out
```

(3) 查看运行结果:

```
root@duanwh-virtual-machine: ~  
File Edit View Search Terminal Help  
Hello, world!  
Hello, world!  
Hello, world!  
Hello, world!  
Hello, world!  
Hello, world!  
Hello, world!  
Hello, world!  
Hello, world!  
Hello, world!  
Hello, world!  
Hello, world!  
Hello, world!  
Hello, world!  
Hello, world!  
Hello, world!  
Hello, world!  
Hello, world!  
Hello, world!  
Hello, world! ^C  
OK!  
pid: 26223, ppid: 26084  
root@duanwh-virtual-machine: ~#
```

分析运行结果:

- 1) 进程执行 `while` 循环，不停的打印 “Hello world!” 字样。
- 2) 按下 `ctrl + c` 键发送 `SIGINT` 信号。
- 3) 进程接到 `SIGINT` 信号，中断（不是终止）`while` 循环，执行 `int_func` 函数，在 `int_func` 函数中修改了 `k` 的值为 0。
- 4) 进程继续执行 `while` 循环，发现 `k = 0` 了，循环的条件不成立，从而终止循环。
- 5) 继续执行程序中 `while` 后的代码，打印 “ok” 字样，打印进程的进程号和父进程的进程号。

(4) 修改程序, 去掉信号处理语句。`//signal(SIGINT, int func);`

无信号处理机制

```
/*signal.c*/
```

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

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

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

```
void int_func(int sig);
```

```
int k;                /*定义循环变量*/
```

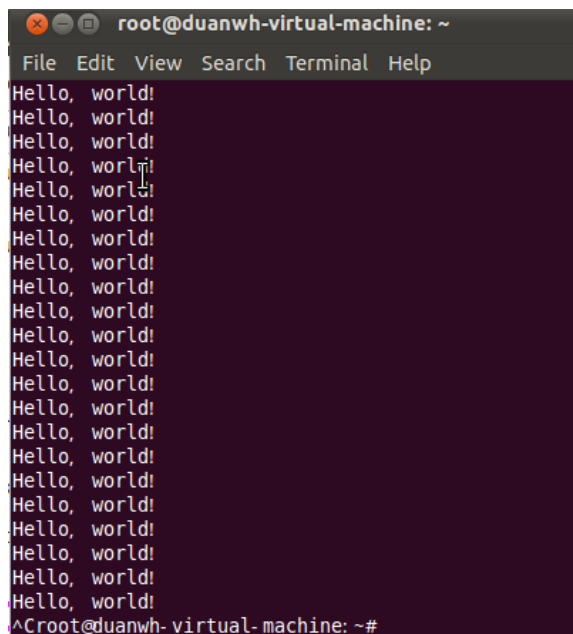
```
void int_func(int sig)
{
    k = 0;
}

int main()
{
    //signal(SIGINT, int_func);
    k = 1;

    while (k == 1)
    {
        printf("Hello, world!\n");
    }

    printf("OK!\n");
    printf("pid:  %d,  ppid: %d \n", getpid(), getppid());
}
```

无信号处理机制，程序运行后，不断显示“Hello, world!”字样，按下 `ctrl + c`，进程直接终止（是在 `while` 语句处终止进程），没有执行 `while` 语句后的语句打印“Ok”，打印进程号等等。

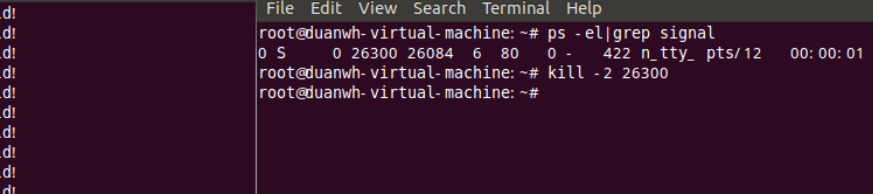
A screenshot of a terminal window titled 'root@duanwh-virtual-machine: ~'. The terminal shows the output of the program, which is a continuous stream of 'Hello, world!' messages. The window has a menu bar with 'File', 'Edit', 'View', 'Search', 'Terminal', and 'Help'. The prompt at the bottom is '^Croot@duanwh-virtual-machine: ~#', indicating that the program was terminated by pressing Ctrl+C.

（5）使用 `kill -l` 命令可以查看系统的中断型号。可以使用 `kill` 命令发送一个中断信号给某个指定进程。

如有问题，请联系 duanwh@126.com

```
root@duanwh-virtual-machine: ~  
File Edit View Search Terminal Help  
root@duanwh-virtual-machine: ~# kill -l  
1) SIGHUP      2) SIGINT      3) SIGQUIT      4) SIGILL      5) SIGTRAP  
6) SIGABRT     7) SIGBUS     8) SIGFPE      9) SIGKILL     10) SIGUSR1  
11) SIGSEGV    12) SIGUSR2    13) SIGPIPE     14) SIGALRM     15) SIGTERM  
16) SIGSTKFLT  17) SIGCHLD   18) SIGCONT     19) SIGSTOP     20) SIGTSTP  
21) SIGTTIN    22) SIGTTOU    23) SIGURG      24) SIGXCPU     25) SIGXFSZ  
26) SIGVTALRM  27) SIGPROF   28) SIGWINCH    29) SIGIO       30) SIGPWR  
31) SIGSYS     34) SIGRTMIN   35) SIGRTMIN+1  36) SIGRTMIN+2  37) SIGRTMIN+3  
38) SIGRTMIN+4 39) SIGRTMIN+5 40) SIGRTMIN+6 41) SIGRTMIN+7 42) SIGRTMIN+8  
43) SIGRTMIN+9 44) SIGRTMIN+10 45) SIGRTMIN+11 46) SIGRTMIN+12 47) SIGRTMIN+13  
48) SIGRTMIN+14 49) SIGRTMIN+15 50) SIGRTMAX-14 51) SIGRTMAX-13 52) SIGRTMAX-12  
53) SIGRTMAX-11 54) SIGRTMAX-10 55) SIGRTMAX-9 56) SIGRTMAX-8 57) SIGRTMAX-7  
58) SIGRTMAX-6 59) SIGRTMAX-5 60) SIGRTMAX-4 61) SIGRTMAX-3 62) SIGRTMAX-2  
63) SIGRTMAX-1 64) SIGRTMAX  
root@duanwh-virtual-machine: ~#
```

在系统中打开两个终端，一个终端执行 `./signal.out` 命令创建一个进程，另一个终端使用 `kill` 命令给该进程发送中断信号。进程的进程号可以使用 `ps -el|grep signal` 查看。



```
root@duanwh-virtual-machine: ~  
File Edit View Search Terminal Help  
Hello, world!  
Hello, world!  
Hello, world!  
Hello, world!  
Hello, world!  
Hello, world!  
Hello, world!  
Hello, world!  
Hello, world!  
Hello, world!  
Hello, world!  
Hello, world!  
Hello, world!  
Hello, world!  
Hello, world!  
Hello, world!  
Hello, world!  
Hello, world!  
Hello, world!  
Hello, world!  
OK!  
pid: 26300, ppid: 26084  
root@duanwh-virtual-machine: ~  
root@duanwh-virtual-machine: ~  
File Edit View Search Terminal Help  
root@duanwh-virtual-machine: ~# ps -el|grep signal  
0 S 0 26300 26084 6 80 0 - 422 n_tty pts/12 00:00:01 signal.out  
root@duanwh-virtual-machine: ~# kill -2 26300  
root@duanwh-virtual-machine: ~#
```

上图中可以看到执行 `signal.out` 创建的进程的进程号是 26300。`kill -2 26300`，是给进程 26300 发送一个 `SIGINT` 信号，26300 进程接收到信号后，将 `k` 置 0，跳出循环后打印 OK 和进程号，进程正常终止。

6. 简单的变量不能实现通信

在父进程中设定一个变量。子进程中修改，父进程中保持不变。因为两个进程中，他们的存储空间是独立的，有变量名一样也是两个不同的拷贝。

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

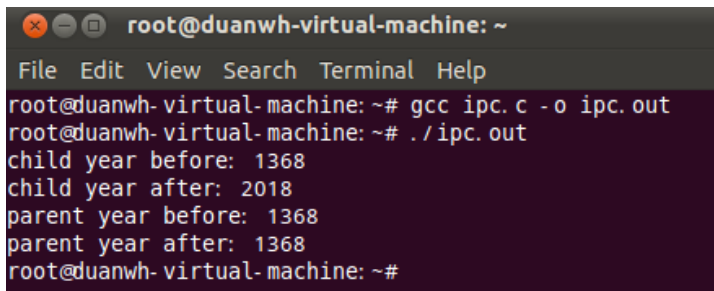
```
int main()
{
    int pid;

    int year = 1368;

    pid = fork();
```

```
if (pid == 0)
{
    printf("child year before: %d\n", year);
    year = 2018;
    printf("child year after: %d\n", year);
}
else
{
    printf("parent year before: %d\n", year);
    sleep(4);
    printf("parent year after: %d\n", year);
}

return 0;
}
```



```
root@duanwh-virtual-machine: ~
File Edit View Search Terminal Help
root@duanwh-virtual-machine: ~# gcc ipc.c -o ipc.out
root@duanwh-virtual-machine: ~# ./ipc.out
child year before: 1368
child year after: 2018
parent year before: 1368
parent year after: 1368
root@duanwh-virtual-machine: ~#
```

分析：

子进程中，父进程中 `year` 保持 1368 年不变。

7. 简单的指针不能实现通信

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

```
int main()
{
    int pid;

    int year = 1368;
    int *pyear = &year;

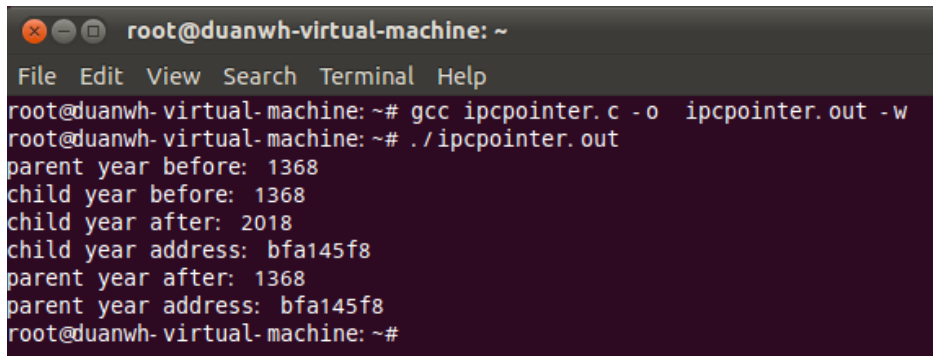
    pid = fork();

    if (pid == 0)
    {
        printf("child year before: %d\n", year);
        *pyear = 2018;
        printf("child year after: %d\n", year);
    }
}
```



```
        printf("child year address: %x\n", &year);
    }
    else
    {
        printf("parent year before: %d\n", year);
        sleep(4);
        printf("parent year after: %d\n", year);
        printf("parent year address: %x\n", &year);
    }

    return 0;
}
```



```
root@duanwh-virtual-machine: ~
File Edit View Search Terminal Help
root@duanwh-virtual-machine: ~# gcc ipcpointer.c -o ipcpointer.out -w
root@duanwh-virtual-machine: ~# ./ipcpointer.out
parent year before: 1368
child year before: 1368
child year after: 2018
child year address: bfa145f8
parent year after: 1368
parent year address: bfa145f8
root@duanwh-virtual-machine: ~#
```

总结：

父子两个进程中，`year` 变量的地址值是一样的，但是在各自的内存空间中（可以类比为教 2 和教 3 的两个 101 教室）。子进程中修改自己空间的数据，修改不了父进程中的数据。

关于实验 1 和实验 2

实验 1 和实验 2 都是认知型的实验，主要是通过实验理解操作系统中的某些概念。

实验 1 的基本要求：

- （1）熟悉 Linux 的环境，测试若干命令。
- （2）学会使用 `gedit` 文本编辑器编写 c 语言程序，`gcc` 编译运行 c 语言程序。
- （3）编写创建进程的程序（参考程序中的程序 4），观察运行结果。

4. → 进程创建 2，增加 `sleep` 语句使得打印结果有交叉。
/*fork2.c*/

实验 1 的收获：

- （1）熟悉了新的操作系统
- （2）熟悉了新的 c 语言编写编译环境。
- （3）进程的创建
- （4）进程执行的并发性和异步性。

如有问题，请联系 duanwh@126.com

实验 2 的基本要求：

- (1) 了解进程通信的方式。
- (2) 实现进程的消息队列通信方式，查看运行结果。
- (3) 实现进程的另外一种通信方式。(或者是共享存储区、或者是管道文件、或者是信号)，查看运行结果。

实验 2 的收获：

- (1) 进程间通信有别于同一个进程中两个函数之间的数据传递。
- (2) 进程的执行是彼此独立的，使用的是操作系统所虚拟出来的存储空间。

实验 3 页面调度算法模拟

1. 参考程序：

```
/*
    title: 页面置换算法
    time: 2015.11.18
    author: duanwh@126.com
*/

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

#define M 12

/*给数组赋值*/
void setarray(int arr[], int num, int val)
{
    int i;
    for (i = 0; i < num; ++i)
        arr[i] = val;
}

/*查找内存 mem 中是否存在页面 page*/
int findexist(int mem[], int mnum, int page)
{
    int i;
    for (i = 0; i < mnum; i++)
    {
        if (mem[i] == page) return i;
    }
    return -1;
}

/*查找内存 mem 中是否存空的位置*/
int findempty(int mem[], int mnum)
{
    int i;
    for (i = 0; i < mnum; i++)
    {
        if (mem[i] == -1) return i;
    }
    return -1;
}
```

```
/*先进先出页面置换算法*/
int fifo(int req[], int rnum, int mnum)
{
    int count;
    int i, j;
    int pos;
    int max;

    /*分配内存空间*/
    int *mem = (int *)malloc(sizeof(int) * mnum);
    setarray(mem, mnum, -1);

    /*time 记录内存中每个页面进入的时间*/
    int *time = (int *)malloc(sizeof(int) * mnum);
    setarray(time, mnum, M);

    /*缺页率初始时为 0*/
    count = 0;

    for (i = 0; i < rnum; ++i)
    {
        /*发现页面请求是否存在内存中*/
        pos = findexist(mem, mnum, req[i]);

        /*如果内存中存在页面请求*/
        if (pos != -1) continue;

        /*如果内存中不存在缺页次数加一*/
        count++;

        /*内存中是否存在空位置可以存放新页面*/
        pos = findempty(mem, mnum);

        /*存在空位置，直接存入新页面，不做置换*/
        if (pos != -1)
        {
            mem[pos] = req[i];
            time[pos] = i;
            continue;
        }

        /*不存在空位置，寻找要置换的页面，fifo*/
    }
}
```

```
        pos = 0;
        for (j = 1; j < mnum; ++j)
        {
            if (time[j] < time[pos])
            {
                pos = j;
            }
        }

        mem[pos] = req[i];
        time[pos] = i;
    }

    free(time);
    free(mem);
    return count;
}

/*最近最久未使用页面置换算法*/
int lru(int req[], int rnum, int mnum)
{
    int count;
    int i, j;
    int pos;
    int max;

    /*分配内存空间*/
    int *mem = (int *)malloc(sizeof(int) * mnum);
    setarray(mem, mnum, -1);

    /*time 记录内存中每个页面最后访问的时间*/
    int *time = (int *)malloc(sizeof(int) * mnum);
    setarray(time, mnum, M);

    /*缺页率初始时为 0*/
    count = 0;

    for (i = 0; i < rnum; ++i)
    {
        /*发现页面请求是否存在内存中*/
        pos = findexist(mem, mnum, req[i]);

        /*如果内存中存在页面请求，此处与 fifo 有些许区别*/
    }
}
```

```
    if (pos != -1)
    {
        time[pos] = i; /*此处 lru 记录的是最后一次的访问时间，而 fifo 中记录的是最
先进入的时间*/
        continue;
    }
    /*如果内存中不存在缺页次数加一*/
    count++;

    /*内存中是否存在空位置可以存放新页面*/
    pos = findempty(mem, mnum);

    /*存在空位置，直接存入新页面，不做置换*/
    if (pos != -1)
    {
        mem[pos] = req[i];
        time[pos] = i;
        continue;
    }

    /*不存在空位置，寻找要置换的页面，lru*/
    pos = 0;
    for (j = 1; j < mnum; ++j)
    {
        if (time[j] < time[pos])
        {
            pos = j;
        }
    }

    mem[pos] = req[i];
    time[pos] = i;
}

free(time);
free(mem);
return count;
}

/*最佳页面置换算法*/
int opt(int req[], int rnum, int mnum)
{
    int count;
    int i, j;
```

```
int pos;
int max;

/*分配内存空间*/
int *mem = (int *)malloc(sizeof(int) * mnum);
setarray(mem, mnum, -1);

/*time 记录内存中每个页面将来要访问的时间*/
int *time = (int *)malloc(sizeof(int) * mnum);

/*缺页率初始时为 0*/
count = 0;

for (i = 0; i < rnum; ++i)
{
    /*发现页面请求是否存在内存中*/
    pos = findexist(mem, mnum, req[i]);

    /*如果内存中存在页面请求，此处与 fifo 相同*/
    if (pos != -1)
        continue;

    /*如果内存中不存在缺页次数加一*/
    count++;

    /*内存中是否存在空位置可以存放新页面*/
    pos = findempty(mem, mnum);

    /*存在空位置，直接存入新页面，不做置换*/
    if (pos != -1)
    {
        mem[pos] = req[i];
        continue;
    }

    /*不存在空位置，寻找要置换的页面，opt*/
    setarray(time, mnum, M);

    int k;

    /*查找内存中每个页面在将来出现的最近时间*/
```

```
for (j = 0; j < mnum; ++j)
{
    for (k = i+1; k < rnum; k++)
    {
        if (mem[j] == req[k])
        {
            time[j] = k;
            break;
        }
    }
}

pos = 0;
for (j = 1; j < mnum; ++j)
{
    if (time[j] > time[pos]) /*此处与 fifo 和 lru 有区别*/
    {
        pos = j;
    }
}

mem[pos] = req[i];
}

free(time);
free(mem);
return count;
}

int main()
{
    int request[M] = {2, 3, 2, 1, 5, 2, 4, 5, 3, 2, 5, 2};

    int count;
    int n = 3;

    count = fifo(request, M, n);
    printf("fifo: %d \n", count);

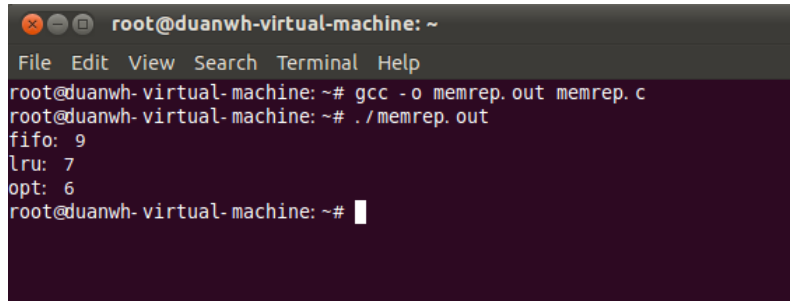
    count = lru(request, M, n);
    printf("lru: %d \n", count);
}
```



```
count = opt(request, M, n);
printf("opt: %d\n", count);

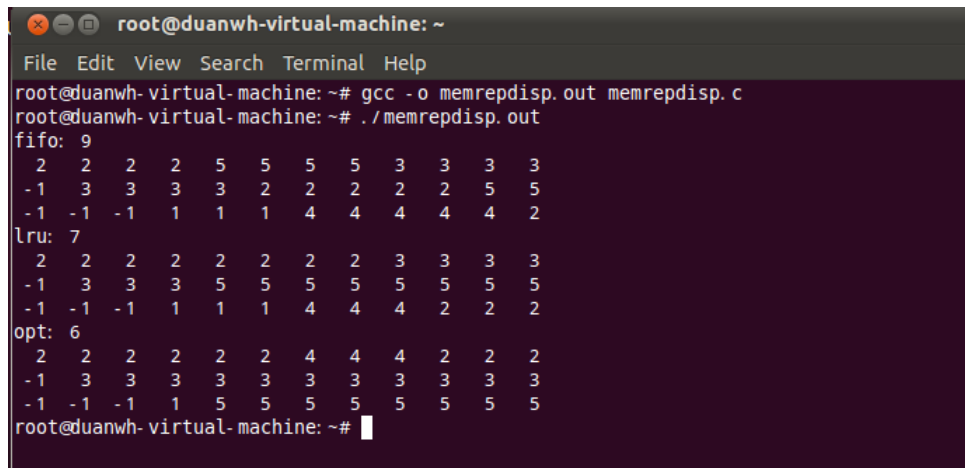
return 0;
}
```

编译运行结果如下图所示。



```
root@duanwh-virtual-machine: ~
File Edit View Search Terminal Help
root@duanwh-virtual-machine:~# gcc -o memrep.out memrep.c
root@duanwh-virtual-machine:~# ./memrep.out
fifo: 9
lru: 7
opt: 6
root@duanwh-virtual-machine:~#
```

2. 试着增加代码打印出如下图所示的内存中页面的变化情况。



```
root@duanwh-virtual-machine: ~
File Edit View Search Terminal Help
root@duanwh-virtual-machine:~# gcc -o memrepdisp.out memrepdisp.c
root@duanwh-virtual-machine:~# ./memrepdisp.out
fifo: 9
 2  2  2  2  5  5  5  5  3  3  3  3
-1  3  3  3  3  2  2  2  2  2  5  5
-1 -1 -1  1  1  1  4  4  4  4  4  2
lru: 7
 2  2  2  2  2  2  2  2  3  3  3  3
-1  3  3  3  5  5  5  5  5  5  5  5
-1 -1 -1  1  1  1  4  4  4  2  2  2
opt: 6
 2  2  2  2  2  2  4  4  4  2  2  2
-1  3  3  3  3  3  3  3  3  3  3  3
-1 -1 -1  1  5  5  5  5  5  5  5  5
root@duanwh-virtual-machine:~#
```

(1) 实验中涉及到了二维数组的操作。二维数组作为函数参数时，必须指定二维数组列的纬度。

```
void ini_tarr(int arr[][M], int rnum, int cnum)
{
    int r, c;

    for (r = 0; r < rnum; ++r)
    {
        for (c = 0; c < cnum; ++c)
            arr[r][c] = -1;
    }
}
```

3. 程序所处理的请求序列是在 main 函数中，试着修改程序从外部的 txt 文件中读取请求序列，并计算缺页次数和缺页率。

实验 4 文件系统的模拟

1. 文件的读写

/*fileoper.c 文件操作*/

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

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

```
#include <string.h>
```

```
#include <fcntl.h>
```

```
int main()
```

```
{
```

```
    int fd0, record_len, buff_len, cnt;
```

```
    char filename[50], f_buff[2000];
```

```
    /*打开当前目录下的一个文件 example.txt，确保存在*/
```

```
    strcpy(filename, "./example.txt");
```

```
    fd0 = open(filename, O_RDWR|O_CREAT, 0644); /*打开文件*/
```

```
    if (fd0 < 0)
```

```
    {
```

```
        printf("Can't create example.txt file!\n");
```

```
        exit(0);
```

```
    }
```

```
    buff_len = 20;
```

```
    record_len = 10;
```

```
    lseek(fd0, 0, SEEK_SET);    /*定位到文件开始位置*/
```

```
    /*读文件，长度为 record_len * buff_len 的内容到 f_buff 中*/
```

```
    cnt = read(fd0, f_buff, record_len * buff_len);
```

```
    cnt = cnt / record_len;
```

```
    printf("%s\n", f_buff);
```

```
    strcpy(f_buff, "1234567890");    /*设置要写入文件的信息*/
```

```
    write(fd0, f_buff, strlen(f_buff)); /*写文件*/
```

```
    write(fd0, "\n\r", 2);
```

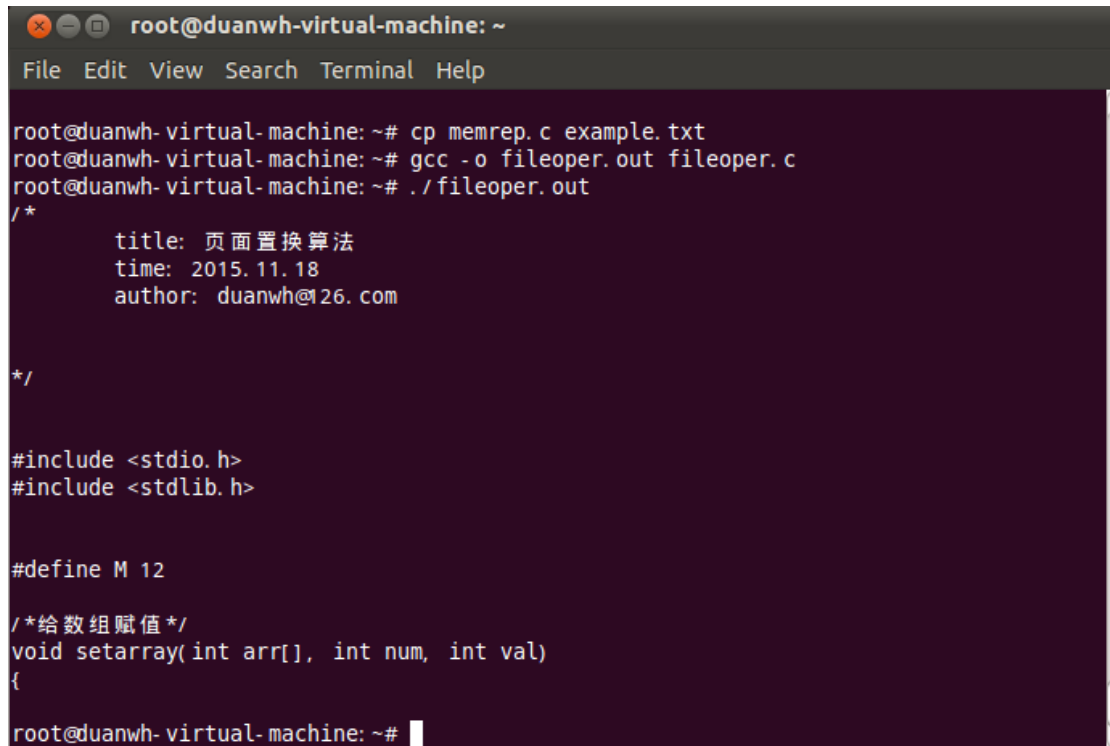
```
    close(fd0);    /*关闭文件*/
```

```
    return 0;
```

```
}
```

如有问题，请联系 duanwh@126.com

- (1) 确保当前目录下有 `example.txt` 文件
- (2) 运行结果，屏幕上打印了从文件 `example.txt` 中读取的内容。



```
root@duanwh-virtual-machine: ~
File Edit View Search Terminal Help

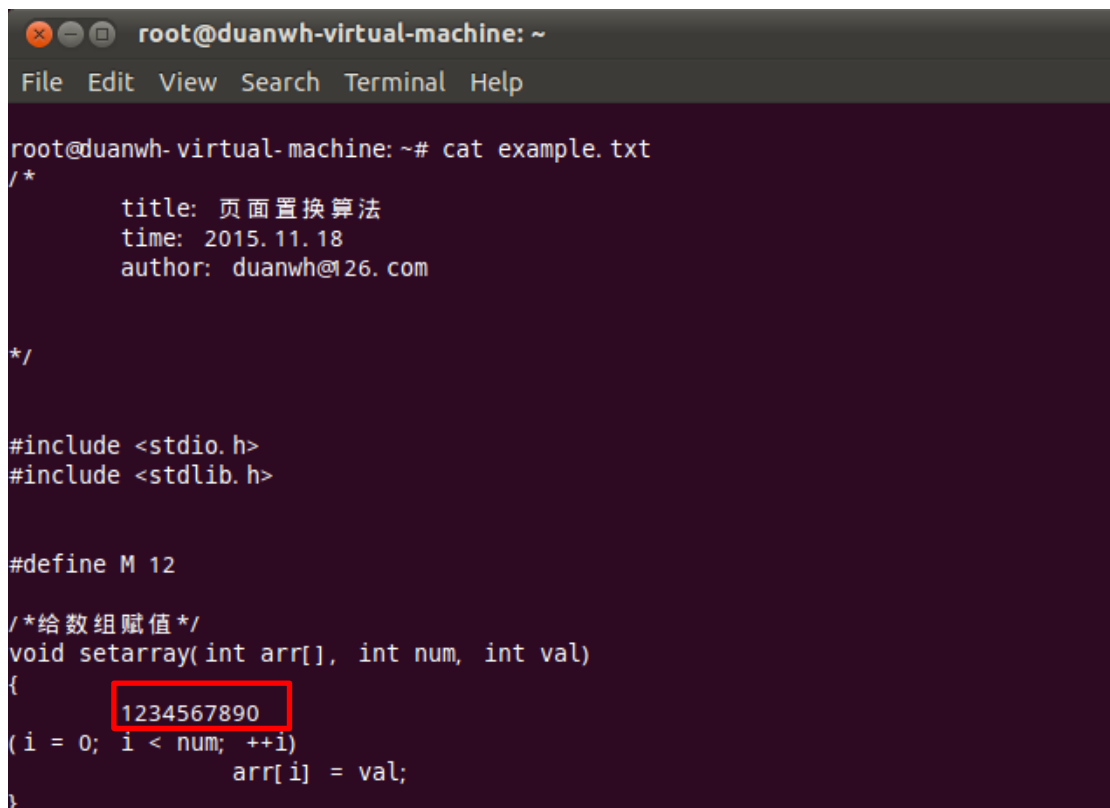
root@duanwh-virtual-machine:~# cp memrep.c example.txt
root@duanwh-virtual-machine:~# gcc -o fileoper.out fileoper.c
root@duanwh-virtual-machine:~# ./fileoper.out
/*
    title: 页面置换算法
    time: 2015.11.18
    author: duanwh@126.com
*/

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

#define M 12

/*给数组赋值*/
void setarray(int arr[], int num, int val)
{
root@duanwh-virtual-machine:~#
```

- (3) 运行结果，使用 `cat` 命令查看 `example.txt` 会发现该文件中多了字符串 `1234567890`，这是程序对文件的写操作。



```
root@duanwh-virtual-machine: ~
File Edit View Search Terminal Help

root@duanwh-virtual-machine:~# cat example.txt
/*
    title: 页面置换算法
    time: 2015.11.18
    author: duanwh@126.com
*/

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

#define M 12

/*给数组赋值*/
void setarray(int arr[], int num, int val)
{
    1234567890
    (i = 0; i < num; ++i)
        arr[i] = val;
}
```

- (4) 用到的文件相关的系统调用
 - a) 打开文件 `open()`;

- b) 关闭文件 `close()`;
- c) 读文件 `read()`;
- d) 写文件 `write()`;
- e) 定位文件 `lseek()`;

2. 文件的复制

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

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

```
#include <fcntl.h>
```

```
#define BUF_SIZE 1024*8
```

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

```
{
```

```
    int fds, fdd;
```

```
    char buf[BUF_SIZE];
```

```
    size_t count = 0;
```

```
    fds = open(argv[1], O_RDONLY);
```

```
    fdd = open(argv[2], O_WRONLY|O_CREAT);
```

```
    if(fds && fdd)
```

```
    {
```

```
        while ((count = read(fds, buf, sizeof(buf))) > 0)
```

```
        {
```

```
            write(fdd, buf, count);
```

```
        }
```

```
    }
```

```
    close(fds);
```

```
    close(fdd);
```

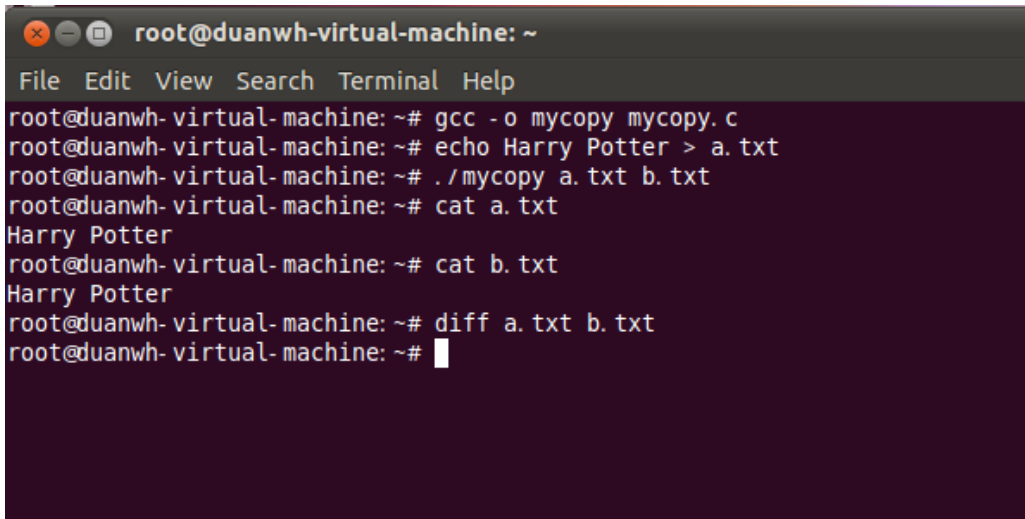
```
}
```

(1) 头文件

`size_t` 是一个整型类型，封装在 `stdio.h` 中。

`O_RDONLY`, `O_WRONLY`, `O_CREAT` 是常量，封装在 `fcntl.h` 头文件中。

(2) 可以使用 `echo`, `>`, `cat` 和 `diff` 等命令测试结果。`echo` 是显示一个字符串。`>`是重定向。`cat` 是显示文件内容。`diff` 是判断文件源文件和目标文件是否一致，如果一致，不输出任何信息。

A terminal window titled 'root@duanwh-virtual-machine: ~' with a menu bar (File, Edit, View, Search, Terminal, Help). The terminal shows the following commands and output:

```
root@duanwh-virtual-machine: ~# gcc -o mycopy mycopy.c
root@duanwh-virtual-machine: ~# echo Harry Potter > a.txt
root@duanwh-virtual-machine: ~# ./mycopy a.txt b.txt
root@duanwh-virtual-machine: ~# cat a.txt
Harry Potter
root@duanwh-virtual-machine: ~# cat b.txt
Harry Potter
root@duanwh-virtual-machine: ~# diff a.txt b.txt
root@duanwh-virtual-machine: ~#
```

(3) mycopy 程序实现了将源文件拷贝到目标文件。但是该程序还不完善，如果目标文件已经存在，该程序只是在目标文件的开头插入了源文件的内容。可以修改程序实现，如果目标文件已经存在，复制之前清空起先的内容。

(4) 此处主函数中有两个参数，一个是整型的 `argc`，一个是字符串数组 `argv`。在运行程序 `./mycopy a.txt b.txt` 时，`argc = 3`，表示此次执行有 3 个参数，`argc[0]=./mycopy`，`argc[1]=a.txt`，`argc[2]=b.txt`。

3. 显示目录命令 `myls`

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

```
#include <dirent.h>
```

```
int list(DIR *dir)
```

```
{
```

```
    struct dirent *pdir;
```

```
    while((pdir = readdir(dir)) != NULL)
```

```
    {
```

```
        if(strncmp(pdir->d_name, ".", 1) == 0 || strcmp(pdir->d_name, "..") == 0)
```

```
            continue;
```

```
        printf("%s\t", pdir->d_name);
```

```
    }
```

```
    printf("\n");
```

```
    return 0;
```

```
}
```

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

```
{
```

```
    DIR *dir;
```

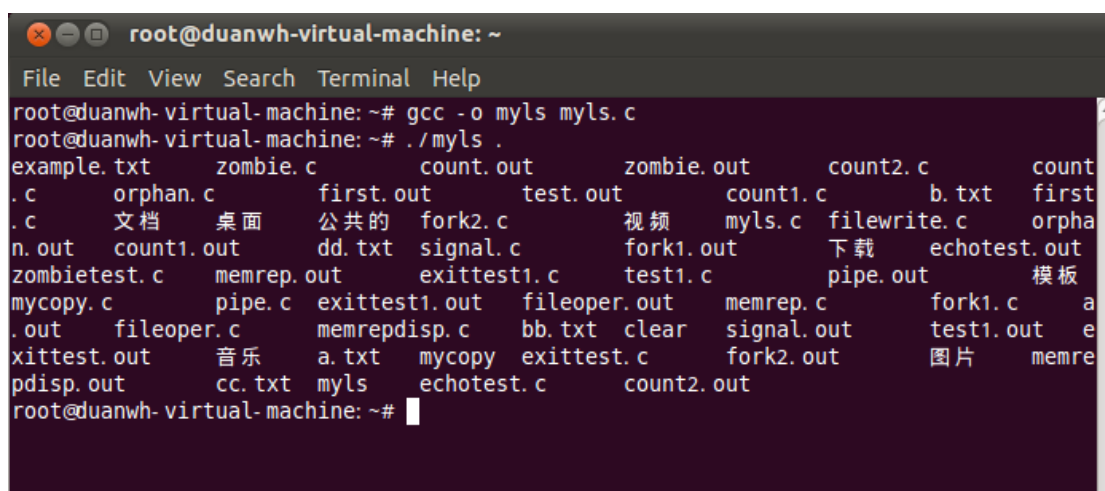
```
if(argc < 2)
{
    fprintf(stderr, "Usage : %s argv[1].\n", argv[0]);
    return -1;
}

if((dir = opendir(argv[1])) == NULL)
{
    perror("Fail to opendir");
    return -1;
}

list(dir);

return 0;
}
```

(1) 测试



```
root@duanwh-virtual-machine: ~
File Edit View Search Terminal Help
root@duanwh-virtual-machine:~# gcc -o myls myls.c
root@duanwh-virtual-machine:~# ./mysls .
example.txt  zombie.c  count.out  zombie.out  count2.c  count
.c  orphan.c  first.out  test.out  count1.c  b.txt  first
.c  文档 桌面 公共的  fork2.c  视频  myls.c  filewrite.c  orpha
n.out  count1.out  dd.txt  signal.c  fork1.out  下载  echotest.out
zombietest.c  memrep.out  exittest1.c  test1.c  pipe.out  模板
mycopy.c  pipe.c  exittest1.out  fileoper.out  memrep.c  fork1.c  a
.out  fileoper.c  memrepdisp.c  bb.txt  clear  signal.out  test1.out  e
xittest.out  音乐  a.txt  mycopy  exittest.c  fork2.out  图片  memre
pdisp.out  cc.txt  myls  echotest.c  count2.out
root@duanwh-virtual-machine:~#
```

(2) 设计到的系统调用和结构

opendir

```
#include <sys/types.h>
#include <dirent.h>

DIR *opendir(const char *name);
```

readdir

```
#include <dirent.h>

struct dirent *readdir(DIR *dirp);
```

struct dirent

如有问题，请联系 duanwh@126.com

```
struct dirent {
    ino_t      d_ino;      /* inode number */
    off_t      d_off;      /* offset to the next dirent */
    unsigned short d_reclen; /* length of this record */
    unsigned char d_type;   /* type of file; not supported
                           by all file system types */
    char        d_name[256]; /* filename */
};
```

(3) 该程序功能非常小，只能处理两个参数。在上图中第一个参数是命令名./myls，第二个参数是当前路径.。可以试着增加该程序的功能。

4. 试着去实现命令 cd, mkdir, rmdir 等命令。