

OSExperiment

AI

Published
with GitBook



目錄

说明	0
实验一、Linux及其使用环境	1
1-1Linux常用命令简介	1.1
1-2Linux下C语言使用、编译和调试	1.2
实验二、进程管理	2
2-1创建进程	2.1
2-2进程的控制与互斥	2.2
(1)进程的控制	2.2.1
(2)进程的互斥	2.2.2
2-3信号机制	2.3
2-4自己动手做操作系统	2.4
实验三、进程通信	3
3-1管道通信	3.1
3-2消息队列通信	3.2
3-3共享存储区通信	3.3
3-4信号量机制	3.4
实验四、内核编译和系统调用	4
4-1编译内核	4.1
4-2系统调用	4.2
实验五、内核模块	5
5-1实际hello内核模块	5.1
5-2设计内核模块显示系统中所有的进程	5.2

说明

操作系统实验

授课老师：李岚

学生姓名：闫致敬

学号：8000114089

班级：软工142班

实验一、Linux及其使用环境

1-1Linux常用命令简介

实验目的

- 1、了解Linux的Shell命令及使用格式。
- 2、学习如何连接Linux系统。
- 3、熟悉Linux的常用基本命令。
- 4、学会如何得到帮助信息。

实验内容

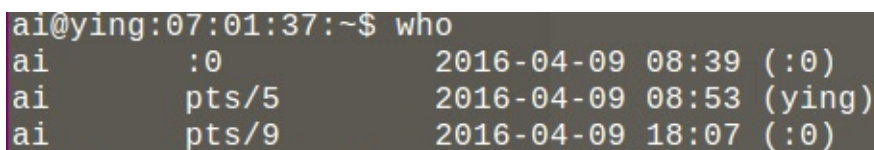
- 1· 通过WINDOWS操作系统中的远程登录程序telnet.exe 登录linux。
- 2· 使用man命令

使用man命令来获得每个Linux命令的帮助手册

用man ls，man passwd，man pwd命令得到ls、passwd、pwd三个命令的帮助手册。

用who 命令显示当前正在你的Linux系统中使用的用户名字：

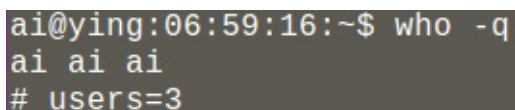
- 3.用who 命令显示当前正在你的Linux系统中使用的用户名字：



```
ai@ying:07:01:37:~$ who
ai      :0                2016-04-09 08:39 (:0)
ai      pts/5            2016-04-09 08:53 (ying)
ai      pts/9            2016-04-09 18:07 (:0)
```

图 1-1-1

- 3.1有多少用户正在使用你的Linux系统？给出显示的结果



```
ai@ying:06:59:16:~$ who -q
ai ai ai
# users=3
```

图 1-1-2

3.2哪个用户登录的时间最长？给出该用户登录的时间和日期。

```
ai@ying:06:54:39:~$ who -T
ai      ? :0          2016-04-09 08:39 (:0)
ai      + pts/5      2016-04-09 08:53 (ying)
ai      + pts/9      2016-04-09 18:07 (:0)
```

图 1-1-3

4.使用下面的命令显示有关你计算机系统信息：uname（显示操作系统的名称），uname -n（显示系统域名），uname -p（显示系统的CPU名称）

4.1你的操作系统名字是什么？

```
ai@ying:06:09:47:~$ uname
Linux
```

图 1-1-4

4.2你计算机系统的域名是什么？

```
ai@ying:06:15:43:~$ uname -n
ying
```

图 1-1-4

4.3你计算机系统的CPU名字是什么？

```
ai@ying:06:15:56:~$ uname -p
x86_64
```

图 1-1-5

5.使用whoami命令找到用户名。然后使用who -a命令来看看你的用户名和同一系统其他用户的列表。

```
ai@ying:06:55:33:~$ whoami
ai
ai@ying:06:55:46:~$ who -a
      system boot 2016-04-09 08:38
      run-level 5 2016-04-09 08:39
LOGIN tty1       2016-04-09 08:39      1054 id=tty1
ai      ? :0      2016-04-09 08:39      ?      1072 (:0)
ai      + pts/5   2016-04-09 08:53      .      2559 (ying)
ai      + pts/9   2016-04-09 18:07 00:33 13310 (:0)
```

图 1-1-6

6.使用passwd命令修改你的登录密码。

```
ai@ying:06:57:35:~$ sudo passwd
[sudo] password for ai:
Enter new UNIX password:
Retype new UNIX password:
passwd: password updated successfully
```

图 1-1-7

7.用命令date显示当前的时间，给出显示的结果。

```
ai@ying:06:57:51:~$ date
2016年 04月 09日 星期六 18:58:18 CST
```

图 1-1-8

8.用pwd显示你的主目录(home directory)名字，给出pwd显示的结果。

```
ai@ying:06:58:38:~$ pwd
/home/ai
```

图 1-1-9

9.使用uptime命令判断系统已启动运行的时间和当前系统中有多少登录用户，给出显示的结果。

```
ai@ying:06:59:14:~$ uptime
18:59:16 up 10:20, 3 users, load average: 0.29, 0.17, 0.15
ai@ying:06:59:16:~$
```

图 1-1-10

1-2Linux下C语言使用、编译和调试

实验目的

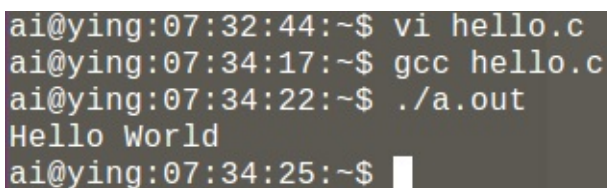
- 1、练习用vi编辑器编辑文本文件；
- 2、复习C语言程序基本知识
- 3、练习并掌握UNIX提供的vi编辑器来编译C程序

实验内容

- 1、用vi编写一个简单的、显示"Hello,World!"的C程序，用gcc编译并观察编译后的结果

```
#include <stdio.h>
int mian()
{
    printf("Hello World!\n");
    return 0;
}
```

- 2、运行生成的可执行文件。



```
ai@ying:07:32:44:~$ vi hello.c
ai@ying:07:34:17:~$ gcc hello.c
ai@ying:07:34:22:~$ ./a.out
Hello World
ai@ying:07:34:25:~$
```

图 1-2-1

- 3、vi编辑器的使用：

- a. 在shell提示符下，输入vi firscrip并按键。vi的界面将出现在显示屏上；
- b. 输入a，输入ls -la，并按键；

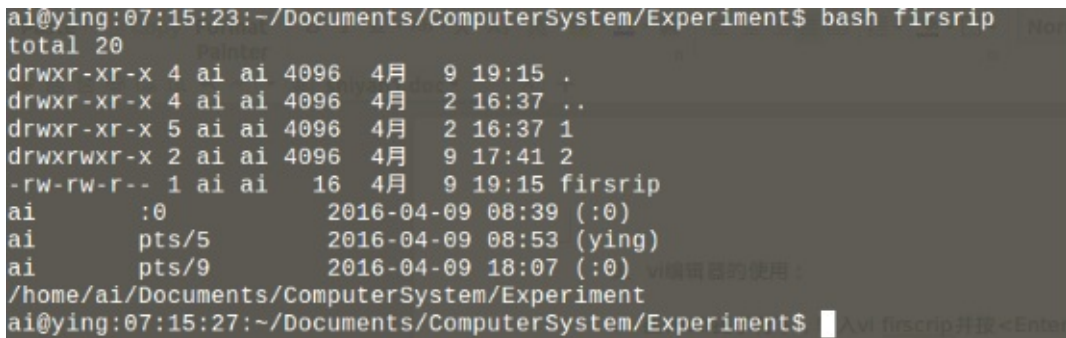
c. 输入who，并按键；

d. 输入pwd，再按键。这时屏幕将如下图所示：

e. 输入:wq，并按键；

f. 在shell提示符下，输入bash firscrip并按键；

g. 观察结果。当前的工作目录中有多少个文件？他们的名称和大小？还有 谁在使用你的计算机系统？当前的工作目录是什么？



```
ai@ying:07:15:23:~/Documents/ComputerSystem/Experiment$ bash firsrip
total 20
drwxr-xr-x 4 ai ai 4096 4月 9 19:15 .
drwxr-xr-x 4 ai ai 4096 4月 2 16:37 ..
drwxr-xr-x 5 ai ai 4096 4月 2 16:37 1
drwxrwxr-x 2 ai ai 4096 4月 9 17:41 2
-rw-rw-r-- 1 ai ai 16 4月 9 19:15 firsrip
ai      :0      2016-04-09 08:39 (:0)
ai      pts/5   2016-04-09 08:53 (ying)
ai      pts/9   2016-04-09 18:07 (:0)
/home/ai/Documents/ComputerSystem/Experiment
ai@ying:07:15:27:~/Documents/ComputerSystem/Experiment$
```

图 1-2-2

实验二、进程管理

2-1进程的创建

实验目的

- 1、掌握进程的概念,明确进程的含义
- 2、认识并了解并发执行的实质

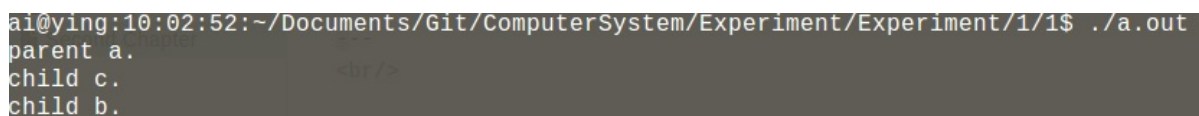
实验内容

- 1、编写一段程序,使用系统调用 `fork()` 创建两个子进程。当此程序运行时,在系统中有一个父进程和两个子进程活动。让每一个进程在屏幕上显示一个字符:父进程显示'a',子进程分别显示字符'b'和字符'c'。

```
/*lab1.c*/

#include <stdio.h>
#include <unistd.h>
int main()
{
    int p, q;
    p = fork();
    if (p < 0)
        printf("fork failed!\n");
    else if(p > 0)
    {
        q = fork();
        if (q < 0)
            printf("fork failed!\n");
        else if (q > 0)
            printf("parent a.\n");
        else
            printf("child b.\n");
    }
    else
        printf("child c.\n");
    return 0;
}
```

结果截图：



```
ali@ying:10:02:52:~/Documents/Git/ComputerSystem/Experiment/Experiment/1/1$ ./a.out
parent a.
child c.
child b.
```

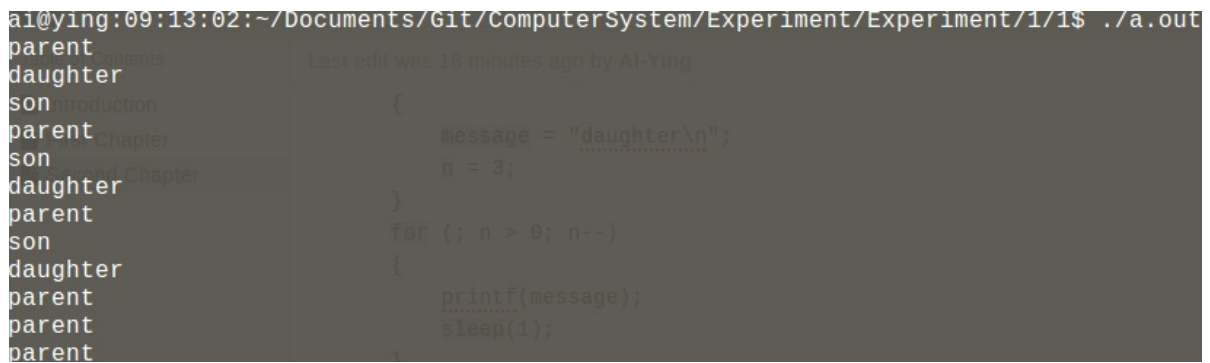
图2-1-1

2、修改上述程序,每一个进程循环显示一句话。子进程显示'daughter 及'son',父进程显示 'parent ...',观察结果,分析原因。

```
/*lab2.c*/

#include<stdio.h>
#include<unistd.h>
int main()
{
    int p, q, n;
    char *message;
    p = fork();
    if (p < 0)
        printf("fork failed!\n");
    else if(p > 0)
    {
        q = fork();
        if (q < 0)
            printf("fork failed!\n");
        else if (q > 0)
        {
            message = "parent\n";
            n = 6;
        }
        else
        {
            message = "son\n";
            n = 3;
        }
    }
    else
    {
        message = "daughter\n";
        n = 3;
    }
    for (; n > 0; n--)
    {
        printf(message);
        sleep(1);
    }
    return 0;
}
```

结果截图：



```
ai@ying:09:13:02:~/Documents/Git/ComputerSystem/Experiment/Experiment/1/1$ ./a.out
parent
daughter
son
parent
son
daughter
parent
son
daughter
parent
parent
parent
parent
```

图2-1-2

fork函数有以下特点：

1. 一次调用，两次返回。
2. 并发执行。父进程与子进程并发进行，互补干扰。
3. 相同的但是独立的地址空间。调用**fork**结束后，父进程和子进程交替执行，跟内核的调度算法有关。所以图2-1-2的结果也说明了这一点。
4. 共享文件。打开或关闭文件的时候，两者是共享的。

实验指导

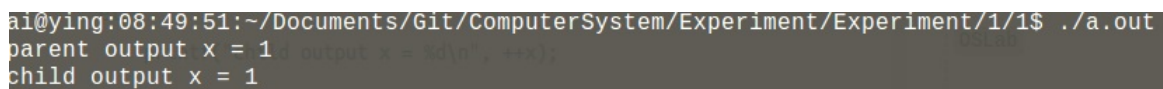
1.1 注意观察下面的程序执行结果,并分析其原因?

```
#include <stdio.h>
#include <unistd.h>
int main()
{
    int p, x;
    p = fork();
    if (p > 0)
        printf("parent output x = %d\n", ++x);
    else
        printf("child output x = %d\n", ++x);

    return 0;
}
```

注:观察 x 值在父子进程中输出的结果。

结果截图：



```
ai@ying:08:49:51:~/Documents/Git/ComputerSystem/Experiment/Experiment/1/1$ ./a.out
parent output x = 1
child output x = 1
```

图 2-1-3

分析：父进程在创立子进程之后，子进程得到父进程的拷贝，二者并发运行时互不干扰，子进程结束后pid返回0，父进程返回子进程的pid。因此if语句和else语句都被执行一遍。又因为， x 默认初始化为0,所以结果输出 x 均自动加一。

1.2分析下面程序所产生的进程树的结构

```

#include <stdio.h>
#include <unistd.h>
main()
{
    int p,x;
    p=fork();
    if (p>0)
    {
        fork();
    }
    else
    {
        fork();
        fork();
    }
    sleep(5);
}

```

注:程序运行时采用后台执行,并及时用 `ps -l` 查看进程树的结构。

截图结果：

```

ai@ying:05:37:38:~/Documents/Git/ComputerSystem/Experiment/Experiment/1/1$ ps -l
F S  UID  PID  PPID  C PRI  NI ADDR SZ  WCHAN  TTY          TIME CMD
0 S  1000  5449  2175  0  80   0 -  7049 wait   pts/20    00:00:00 bash
0 S  1000  7781  5449  0  80   0 -  1055 hrtime  pts/20    00:00:00 a.out
1 S  1000  7782  7781  0  80   0 -  1055 hrtime  pts/20    00:00:00 a.out
1 S  1000  7783  7781  0  80   0 -  1055 hrtime  pts/20    00:00:00 a.out
1 S  1000  7784  7782  0  80   0 -  1055 hrtime  pts/20    00:00:00 a.out
1 S  1000  7785  7782  0  80   0 -  1055 hrtime  pts/20    00:00:00 a.out
1 S  1000  7786  7784  0  80   0 -  1055 hrtime  pts/20    00:00:00 a.out
4 R  1000  7787  5449  0  80   0 -  3561 -      pts/20    00:00:00 ps

```

图 2-1-4

分析：由图可知，一共有6个a.out进程。执行 `p = fork()` 代码时，父进程0创建一个子进程0-1。执行 `if(p > 0)` 时，父进程0又创建了一个子进程0-2。执行 `else` 语句中的第一个 `fork()` 时，子进程0-1创建了子进程0-1-1。执行 `else` 语句中的第二个 `fork()` 时，子进程0-1又创建了子进程0-1-2。而子进程0-1-1也创建了子进程0-1-1-1。转化成图形为为：


```
//六个进程。
0
0---->0-1
0---->0-2
0---->0-1---->0-1-1
0---->0-1---->0-1-2
0---->0-1---->0-1-1---->0-1-1-1
0---->0-1---->0-1-1---->0-1-1-2
```

1.3 分析vfork()函数的使用

```
#include <stdio.h>
{
    int p,x;
    p=vfork();
    if (p>0)
        printf("parent output x=%d",++x);
    else
        printf("child output x=%d",++x);
}
```

说明:注意分析该程序的执行结果与上面的使用 fork 的区别,并分析其原因。

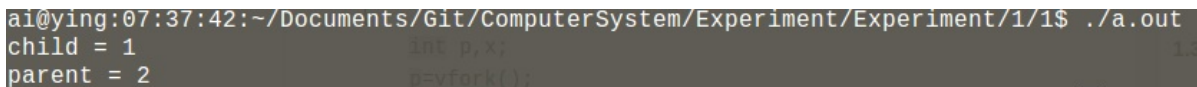
截图结果：

```
ai@ying:07:37:02:~/Documents/Git/ComputerSystem/Experiment/Experiment/1/1$ ./a.out
child = 1
parent = -788517498
```

图 2-1-5

parent打印的是一个随机数。修改源代码，在 else 语句后面添加一行代码 `exit(0);`。执行结果如下：

截图结果：



```
ai@ying:07:37:42:~/Documents/Git/ComputerSystem/Experiment/Experiment/1/1$ ./a.out
child = 1
parent = 2
```

图 2-1-6

对比图2-1-5和图2-1-6。分析fork和vfork的区别如下：

fork：新创建的子进程将得到与父进程用户级虚拟地址空间相同的且独立的一份拷贝，包括文本、数据段和bss段、堆以及用户栈。也就是说，子进程在对父进程中的同名变量修改是，不会影响父进程中的值。这也同时说明了图2-1-3的结果。

vfork：新建的子进程将与父进程共享地址空间。也就是说，子进程如果修改了父进程中的同名变量，将会影响父进程的输出结果。这也是解释了图2-1-5和图2-1-6的结果。但仍会有疑问？图2-1-5和图2-1-6的父进程输出结果不一样。这是因为在调用vfork后，首先会执行子进程，直到子进程调用exec或exit函数后，父进程才执行。图2-1-5的结果因为子进程修改了同名变量后却没有调用exec或exit函数，使得子进程意外中断，父进程中的同名变量变成了随机数。

2-2进程的控制与互斥

(1)进程的控制

实验目的

- 1、掌握使用 `exec` 调用进程的方法
- 2、熟悉进程的睡眠、同步、撤消等进程控制方法

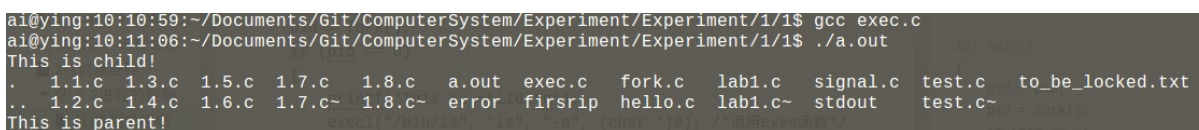
实验内容

- 1、用 `fork()` 创建一个进程,再调用 `exec()` 用新的程序替换该子进程的内容
- 2、利用 `wait()` 来控制进程执行顺序

```
/*lab2*/
#include <stdio.h>
#include <sys/wait.h>
#include <unistd.h>

int main()
{
    pid_t pid;
    pid = fork();
    if (pid == 0)
    {
        printf("This is child!\n");
        execl("/bin/ls", "ls", "-a", (char *)0); /*调用exec函数*/
    }
    if (pid > 0)
    {
        wait(0); /*同步，保证子进程先执行*/
        printf("This is parent!\n");
    }
    return 0;
}
```

结果截图：



```
ai@ying:10:10:59:~/Documents/Git/ComputerSystem/Experiment/Experiment/1/1$ gcc exec.c
ai@ying:10:11:06:~/Documents/Git/ComputerSystem/Experiment/Experiment/1/1$ ./a.out
This is child!
.  1.1.c 1.3.c 1.5.c 1.7.c 1.8.c a.out exec.c fork.c lab1.c signal.c test.c to_be_locked.txt
.. 1.2.c 1.4.c 1.6.c 1.7.c~ 1.8.c~ error firsrup hello.c lab1.c~ stdout test.c~
This is parent!
```

图2-2-1

实验指导

例 4:

```

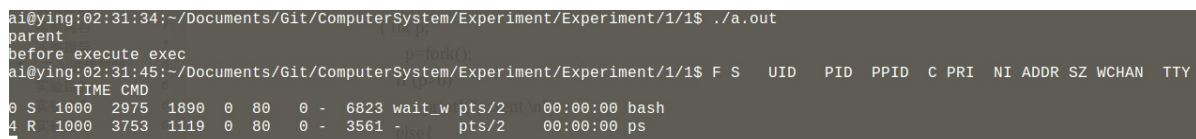
#include<stdio.h>
#include <unistd.h>
int main()
{
    int p;
    p = fork();
    if (p>0)
        printf("parent\n");
    else
    {
        printf("before execute exec\n");
        execl("/bin/ps", "ps", "-l", (char *)0);
        printf("after execute exec\n");
    }

    return 0;
}

```

注意:调用 `execl` 函数的前后输出语句是否都输出,分析其原因。

结果截图：



```

ai@ying:02:31:34:~/Documents/Git/ComputerSystem/Experiment/Experiment/1/1$ ./a.out
parent
before execute exec
ai@ying:02:31:45:~/Documents/Git/ComputerSystem/Experiment/Experiment/1/1$ F S UID PID PPID C PRI NI ADDR SZ WCHAN TTY
TIME CMD
0 S 1000 2975 1890 0 80 0 - 6823 wait_w pts/2 00:00:00 bash
4 R 1000 3753 1119 0 80 0 - 3561 - pts/2 00:00:00 ps

```

图 2-2-2

分析：根据图2-2-1可知，结果并不是想象的那样会在结尾输出 `after execute exec` 这句话。这根`exec`函数有关，查资料其实一共有6种以`exec`开头的函数，统称`exec`函数。如下：

```
#include <unistd.h>
int execl(const char *path, const char *arg, ...);
int execlp(const char *file, const char *arg, ...);
int execl_e(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[]);
```

这些函数如果调用成功则加载新的程序从启动代码开始执行,不再返回,如果调用出错则返回-1。因此例4中的子进程执行 `execl("/bin/ps","ps","-l",(char *)0);` 代码后,子进程会调用 `ps` 这个进程。则子进程的内容全部被 `ps` 进程替换了。也就说子进程 `a.out` 被 `ps` 进程替换了,如果再填上一行代码 `sleep(5)` 会发现,打印结果如图1-4-2中,指出现一个 `a.out` 进程,而不是两个 `a.out` 进程。这也说明了为什么图2-2-2中不会出现 `after execute exec`。因为内容被 `ps` 进程替换, `execl` 函数下面的内容继而也不会被执行了。

在例4的 `return 0` 语句前添加一行 `sleep(5)` 结果截图:

```
ai@ying:01:49:16:~/Documents/Git/ComputerSystem/Experiment/Experiment/1/1$ ./a.out
parent
before execute exec
```

F	S	UID	PID	PPID	C	PRI	NI	ADDR	SZ	WCHAN	TTY	启动代码开始	TIME	CMD	如果调用出错则返
0	S	1000	2555	2550	0	80	0	-	6857	wait	pts/5	00:00:00	00:00:00	bash	
0	S	1000	2598	915	0	80	0	-	236651	poll_s	pts/5	00:00:02	00:00:02	evince	代码后 子
0	S	1000	4265	2555	0	80	0	-	1056	hrttime	pts/5	00:00:00	00:00:00	a.out	说子进程a.out被
4	R	1000	4266	4265	0	80	0	-	3561	-	pts/5	00:00:00	00:00:00	ps	

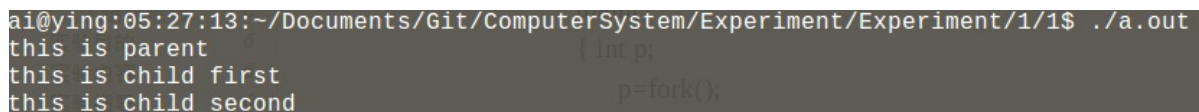
图 2-2-3

例5:

```
#include<stdio.h>
#include<unistd.h>
#include<stdlib.h>
int main()
{
    int p;
    p = fork();
    if (p > 0)
        printf("this is parent \n");
    else
    {
        printf("this is child first \n");
        printf("this is child second \n");
        _exit(0);
    }
    return 0;
}
```

注意:观察子进程的两个输出语句的执行结果。

截图结果:



```
ai@ying:05:27:13:~/Documents/Git/ComputerSystem/Experiment/Experiment/1/1$ ./a.out
this is parent
this is child first
this is child second
```

图2-2-4

分析：对比图2-2-2，发现子进程这次执行了 `else` 语句中所有代码。这也说明了，在子进程未调用`exec`函数或`exit`函数之前，子进程的内容没有改变。调用后，子进程中的所有内容将被`exec`或`exit`进程代替，且不返回子进程。

(2)进程的互斥

实验目的

- 1、进一步认识并发执行的实质
- 2、分析进程竞争资源的现象,学习解决进程互斥的方法

实验内容

- 1、修改实验(二)中的程序 2,用 lockf()来给每一个进程加锁,以实现进程之间的互斥;观察并分析出现的现象
- 2、分析以下程序的输出结果:

```
#include<stdio.h>
#include<unistd.h>
#include<stdlib.h>
int main()
{
    int p1, p2, i;
    int *fp;
    fp = fopen("to_be_locked.txt", "w+");
    if (fp == NULL)
    {
        printf("Fail to create file \n");
        exit(-1);
    }
    while ((p1 = fork()) == -1);    /*创建子进程*/

    if (p1 == 0)
    {
        lockf(*fp, 1, 0);          /*加锁*/
        for (i = 0; i < 10; i++)
            fprintf(fp, "daughter %d\n", i);
        lockf(*fp, 0, 0);          /*解锁*/
    }
    else
```



```
{
    while ((p2 = fork()) == -1); /*创建子进程*/
    if (p2 == 0)
    {
        lockf(*fp, 1, 0);      /*加锁*/
        for (i = 0; i < 10; i++)
            fprintf(fp, "son %d\n", i);
        lockf(*fp, 0, 0);      /*解锁*/
    }

    else
    {
        wait(NULL);
        lockf(*fp, 1, 0);      /*加锁*/
        for (i = 0; i < 10; i++)
            fprintf(fp, "parent %d\n", i);
        lockf(*fp, 0, 0);      /*解锁*/
    }

}
fclose(fp);
return 0;
}
```

结果截图：

```
ai@ying:09:53:28:~/Documents/Git/ComputerSystem/Experiment/Experiment/1/1$ ./a.out
ai@ying:09:53:30:~/Documents/Git/ComputerSystem/Experiment/Experiment/1/1$ cat to_be_locked.txt
daughter 0
daughter 1
daughter 2
daughter 3
daughter 4
daughter 5
daughter 6
daughter 7
daughter 8
daughter 9
son 0
son 1
son 2
son 3
son 4
son 5
son 6
son 7
son 8
son 9
parent 0
parent 1
parent 2
parent 3
parent 4
parent 5
parent 6
parent 7
parent 8
parent 9
```

图 2-2-5

实验指导

1、所涉及的系统调用 `lockf(files,function,size)` 用作锁定文件的某些段或者整个文件。本函数的头文件为

```
#include "unistd.h"
```

参数定义：

```
int lockf(files,function,size)
int files,function;
long size;
```

其中：`files` 是文件描述符；`function` 是锁定和解锁：1 表示锁定，0 表示解锁。 `size` 是锁定或解锁的字节数，为 0，表示从文件的当前位置到文件尾。

2、参考程序

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

int main()
{
    int p1, p2, i;
    p1 = fork(); /*创建子进程*/
    if (p1 == 0)
    {
        lockf(1, 1, 0); /*加锁，这里第一个参数为stdout
                        (标准输出设备的描述符)*/
        for (i = 0; i < 10; i++)
        {
            printf("child1 = %d\n", i);
            sleep(1);
        }
        lockf(1, 0, 0); /*解锁*/
    }
    else
    {
        p2 = fork(); /*创建子进程*/
        if (p2 == 0)
        {
            lockf(1, 1, 0); /*加锁*/
            for (i = 0; i < 10; i++)
            {
                printf("child2 = %d\n", i);
                sleep(1);
            }
            lockf(1, 0, 0); /*解锁*/
        }
        else
        {
            lockf(1, 1, 0); /*加锁*/
            for (i = 0; i < 10; i++)
            {
                printf("parent %d\n", i);
                sleep(1);
            }
            lockf(1, 0, 0); /*解锁*/
        }
    }
}
```

```

    }
}
return 0;
}

```

注意:

- (1)查看程序执行的结果并估计程序执行所需要时间。
- (2)将程序中所有的 `lockf` 函数加上注释,再观察程序执行的结果和估算程序执行所需的时间。
- (3)分析这两次执行的结果与时间的区别。

结果截图：

```

ai@ying:11:19:05:~/Documents/Git/ComputerSystem/Experiment/Experiment/1/1$ time ./a.out
parent 0
parent 1
parent 2
parent 3
parent 4
parent 5
parent 6
parent 7
parent 8
parent 9
child2 = 0
real    0m10.003s
user    0m0.000s
sys     0m0.000s
ai@ying:11:19:20:~/Documents/Git/ComputerSystem/Experiment/Experiment/1/1$ child2 = 1
child2 = 2
child2 = 3
child2 = 4
child2 = 5
child2 = 6
child2 = 7
child2 = 8
child2 = 9
child1 = 0
child1 = 1
child1 = 2
child1 = 3
child1 = 4
child1 = 5
child1 = 6
child1 = 7
child1 = 8
child1 = 9

```

注释`lockf()`前：图2-2-6

```

ai@ying:10:53:00:~/Documents/Git/ComputerSystem/Experiment/Experiment/1/1$ time ./a.out
parent 0
child1 = 10
child2 = 0
parent 1
child2 = 1
parent 2
child2 = 2
parent 3
child2 = 3
parent 4
child2 = 4
parent 5
child2 = 5
parent 6
child2 = 6
parent 7
child2 = 7
parent 8
child2 = 8
parent 9
child2 = 9

real    0m10.003s
user    0m0.000s
sys     0m0.004s

```

注意：

- (1) 查看程序执行的结果并估计程序执行所需时间。
- (2) 将程序中所有的 `lockf` 函数加上注释，再观察程序执行所需的时间。
- (3) 分析这两次执行的结果与时间的区别。

注释`lockf()`后：图2-2-7

分析:分析:根据图2-2-6可知，每个进程结束大概需要10秒，所以一共大概用了30秒左右的时间。而根据图2-2-7可知，整个程序用了10秒左右。`lockf()`函数的作用就是，锁定进程之间共享的文件夹，只准当前进程使用，其他进程需当前进程解锁后可以使用共享文件。在这个实验中，共享文件为标准输出流文件`stdout`。当子进程`child1`锁定时，其他进程需要等待，当子进程`child1`解锁后，其他使用这个文件的进程才可以运行。所以，在没有注释`lockf()`之前，因为每个进程都使用了`stdout`,每个进程依次执行。而图2-2-7的结果，显示了三个进程是并发，几乎是同步执行，因此时间用的也较少。

2-3信号机制

实验目的

- 1、了解什么是信号
- 2、熟悉 LINUX 系统中进程之间软中断通信的基本原理

实验内容

编写程序:用 `fork()` 创建两个子进程,再用系统调用 `signal()` 让父进程捕捉 键盘上来的中断信号(即按`^c` 键);捕捉到中断信号后,父进程用系统调用 `kill()` 向两个子进程发出信号,子进程捕捉到信号后分别输出下列信息后终止:

```
Child process1 is killed by parent!  
Child process2 is killed by parent!
```

父进程等待两个子进程终止后,输出如下的信息后终止:

```
Parent process is killed!
```

实验指导

参考程序

```
#include<stdio.h>  
#include<signal.h>  
#include<unistd.h>  
#include<stdlib.h>  
#include<wait.h>  
  
void waiting();  
void stop();  
int wait_mark;
```

```
int main()
{
    int p1, p2, stdout;
    signal(SIGINT, SIG_IGN);
    while ((p1 = fork()) == -1); /*创建子进程p1*/
    if (p1 > 0)
    {
        while ((p2 = fork()) == -1); /*创建子进程p2*/
        if (p2 > 0)
        {
            wait_mark = 1;
            signal(SIGINT, stop); /*接受到^c信号,转stop*/
            waiting();
            kill(p1, 10);          /*向p1发软中断信号10*/
            kill(p2, 12);          /*向p2发软中断信号12*/
            wait(0);               /*同步*/
            wait(0);
            printf("Parent process is killed!\n");
            exit(0);
        }
        else
        {
            wait_mark = 1;
            signal(12, stop); /*接收到软中断信号12,转stop*/
            waiting();
            lockf(stdout, 1, 0);
            printf("Child process 2 is killed by parent!\n");
            lockf(stdout, 0, 0);
            exit(0);
        }
    }

    else
    {
        wait_mark = 1;
        signal(10, stop);
        waiting();
        lockf(stdout, 1, 0);
        printf("Child process 1 is killed by parent!\n");
    }
}
```

```

        lockf(stdout, 0, 0);
        exit(0);
    }
    return 0;
}

void waiting()
{
    while (wait_mark != 0);
}

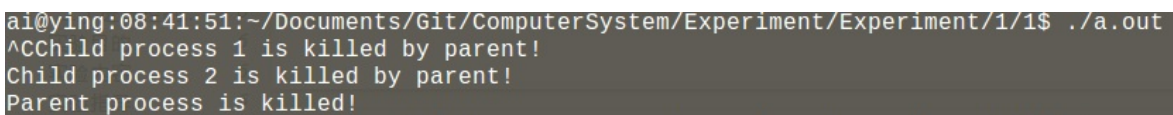
void stop()
{
    wait_mark = 0;
}

```

注意:

- (1) 运行编译后的程序,查看输出;按 CTRL+C 键再查看程序执行结果。
- (2) 将编译后的程序放在后台执行,查看输出;按 CTRL+C 键再查看程序执行结果;发送 kill -INT pid(放入后台后显示的进程号)

结果截图：

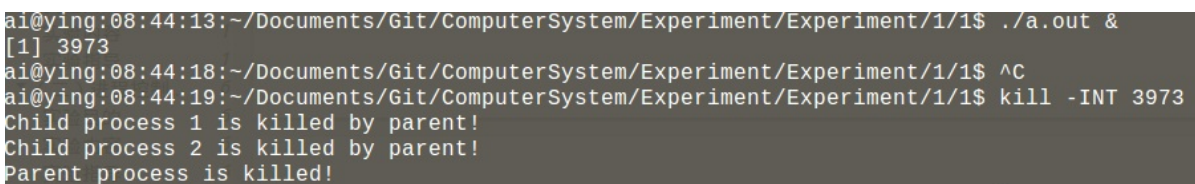


```

ai@ying:08:41:51:~/Documents/Git/ComputerSystem/Experiment/Experiment/1/1$ ./a.out
^CChild process 1 is killed by parent!
Child process 2 is killed by parent!
Parent process is killed!

```

图 2-3-1



```

ai@ying:08:44:13:~/Documents/Git/ComputerSystem/Experiment/Experiment/1/1$ ./a.out &
[1] 3973
ai@ying:08:44:18:~/Documents/Git/ComputerSystem/Experiment/Experiment/1/1$ ^C
ai@ying:08:44:19:~/Documents/Git/ComputerSystem/Experiment/Experiment/1/1$ kill -INT 3973
Child process 1 is killed by parent!
Child process 2 is killed by parent!
Parent process is killed!

```

图 2-3-2


```

ai@ying:11:02:43:~/Documents/Git/ComputerSystem/Experiment/Experiment/1/1$ 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

```

图 2-3-3

```

ai@ying:10:17:24:~/Documents/Git/ComputerSystem/Experiment/Experiment/1/1$ gcc signal.c
ai@ying:10:17:27:~/Documents/Git/ComputerSystem/Experiment/Experiment/1/1$ ./a.out
^CParent process is killed!

```

图 2-3-4

分析：图2-3-1和图2-3-2是运行结果。图2-3-3显示的是系统信号，其中编号为10和12是用户自定义信号。图2-3-4是注释掉程序开头 `signal(SIGINT, SIG_IGN);` 这段代码截图。

程序开始为什么会有这一段代码

```
signal(SIGINT, SIG_IGN);
```

这一段代码，目的是忽略子进程的中断信号，当程序运行过程中出现中断信号时，子进程会忽略此中断信号。为什么要对子进程忽略中断信号？因为程序中子进程是接受父进程发的中断信号，如果不忽略的话，中断信号会影响子进程接受父进程发的信号。为了更明白原因，把这段代码注释掉，结果截图2-3-4。根据截图可以看到，子进程并没有输出

```

Child process1 is killed by parent!
Child process2 is killed by parent!

```

而是只是输出了

```
Parent process is killed!
```

而是因为子进程接受Ctrl+C中断信号，早于父进程发起的中断信号，以至于父进程还未发起信号，子进程就终止了，看不到图2-3-1的结果。在程序开始加入这一行代码还有另一个作用，也即防止子进程变成僵尸进程。这又是为什么呢？

如果不加入这一行代码？子进程可能在接受到Ctrl+c中断新号时，直接被杀死。而此时如果父进程又没有调用wait()函数，也即父进程没有对子进程回收，此时子进程很有可能成为僵尸进程。

为什么父进程不会忽略中断信号？

因为linux中，父进程默认是忽略中断信号的，所以此段代码对父进程不会有影响。

2-4自己动手做操作系统

实验目的

- 1、了解操作系统的启动步骤
- 2、熟悉汇编语言构建操作系统的步骤和方法

实验内容

使用汇编语言实现最小的“操作系统”

实验指导

在**linux**环境下

- 1.安装nasm。
- 2.安装bochs。
- 3.编辑文件boot.asm。

```

org 07c00h                ; 告诉编译器程序加载到 7c00 处
mov ax, cs
mov ds, ax
mov es, ax
call DispStr              ; 调用显示字符串例程
jmp $                    ; 无限循环
DispStr:
mov ax, BootMessage
mov bp, ax                ; ES:BP = 串地址
mov cx, 16                ; CX = 串长度
mov ax, 01301h            ; AH = 13, AL = 01h
mov bx, 000ch             ; 页号为 0 (BH = 0) 黑底红字 (BL = 0Ch, 高亮)
mov dl, 0
int 10h                  ; 10h 号中断
ret
BootMessage:
db "Hello, OS world!"
times
510-($-$$)
db 0                    ; 填充剩下的空间, 使生成的二进制代码恰好为 512 字节
dw 0xaa55               ; 结束标志

```

4. 生成 bin 文件

```
nasm boot.asm -o boot.bin
```

5. 编写 bochs 配置文件。bochsrc(在默认的原文件上改写)

6. 将 bin 文件写入到软盘。

```
dd if=boot.bin of=a.img bs=512 count=1 conv=notrunc
```

7. 观察操作系统输出。

结果截图

把 *bin* 文件写入到软盘。

```

ai@ying:04:10:13:~/Documents/Git/ComputerSystem/MakeOS/MakeOS/chapter1$ dd if=boot.bin of=a.img bs=512 count=1 conv=notrunc
1+0 records in
1+0 records out
512 bytes (512 B) copied, 0.050368 s, 10.2 kB/s

```

图 1-8-1

*bochrc*文件配置。

```

ai@ying:04:20:38:~/Documents/Git/ComputerSystem/MakeOS/MakeOS/chapter1$ bochs -f bochsout.txt
=====
Bochs x86 Emulator 2.6.8
Built from SVN snapshot on May 3, 2015
Compiled on Apr 14 2016 at 13:15:02
=====
00000000000i[CPU0] BXSHARE not set. using compile time default '/usr/local/share/bochs'
00000000000i[CPU0] reading configuration from bochsout.txt
00000000000p[CPU0] >>PANIC<< bochsout.txt:1: a bochssrc option needs at least one parameter
00000000000e[SIM] notify called, but no bxevent_callback function is registered
=====
Bochs is exiting with the following message:
[CPU0] bochsout.txt:1: a bochssrc option needs at least one parameter
=====
00000000000i[CPU0] CPU is in real mode (active)
00000000000i[CPU0] CS.mode = 16 bit
00000000000i[CPU0] SS.mode = 16 bit
00000000000i[CPU0] EFER = 0x00000000
00000000000i[CPU0] EAX=00000000 EBX=00000000 ECX=00000000 EDX=00000000
00000000000i[CPU0] ESP=00000000 EBP=00000000 ESI=00000000 EDI=00000000
00000000000i[CPU0] IOPL=0 id vip vif ac vm rf nt of df if tf sf ZF af PF cf
00000000000i[CPU0] SEG sltr(index|ti|rpl) base limit G D
00000000000i[CPU0] CS:0000( 0000| 0| 0) 00000000 00000000 0 0
00000000000i[CPU0] DS:0000( 0000| 0| 0) 00000000 00000000 0 0
00000000000i[CPU0] SS:0000( 0000| 0| 0) 00000000 00000000 0 0
00000000000i[CPU0] ES:0000( 0000| 0| 0) 00000000 00000000 0 0
00000000000i[CPU0] FS:0000( 0000| 0| 0) 00000000 00000000 0 0
00000000000i[CPU0] GS:0000( 0000| 0| 0) 00000000 00000000 0 0
00000000000i[CPU0] EIP=00000000 (00000000)
00000000000i[CPU0] CR0=0x00000000 CR2=0x00000000
00000000000i[CPU0] CR3=0x00000000 CR4=0x00000000
bx_dbg_read_linear: physical memory read error (phy=0x000000000000, lin=0x00000000)
00000000000i[SIM] quit_sim called with exit code 1

```

图 1-8-2

输出显示结果

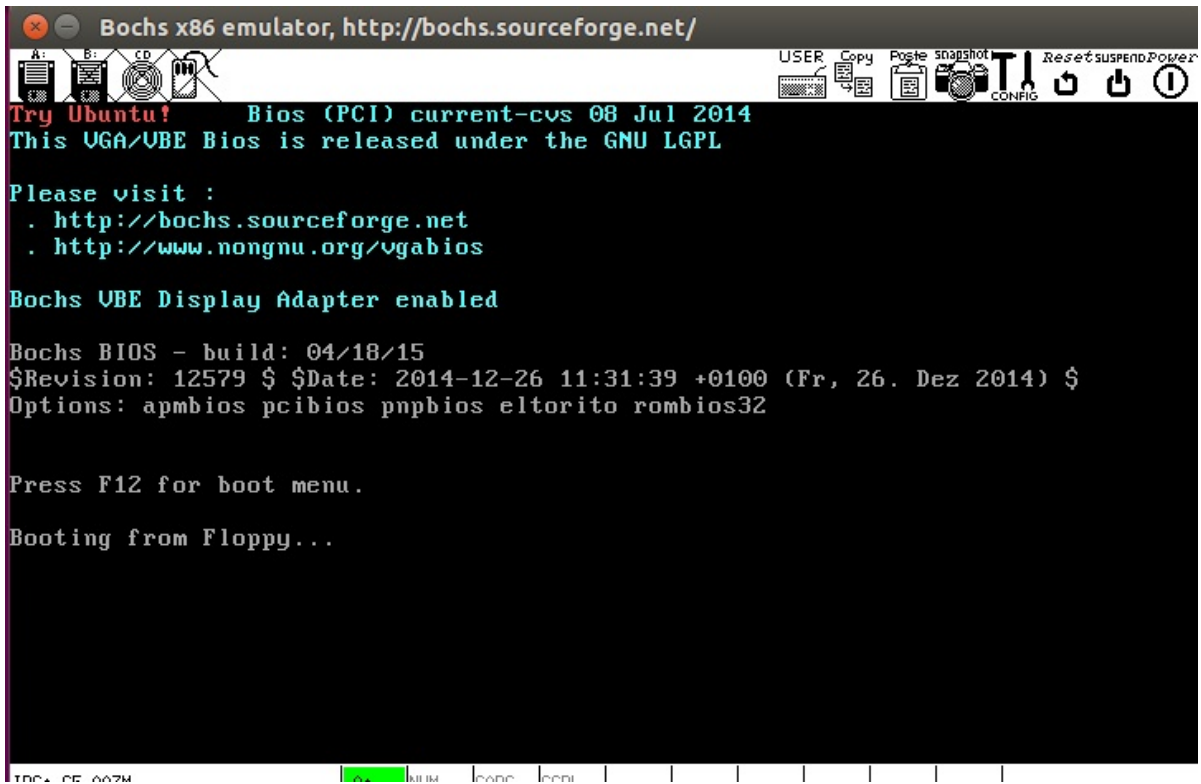


图 1-8-3

实验三、进程通信

3-1 管道通信

实验目的

- 1、了解什么是管道
- 2、熟悉 UNIX/LINUX 支持的管道通信方式

实验内容

编写程序实现进程的管道通信。用系统调用 `pipe()` 建立一管道,二个进程 P1 和 P2 分别向管道各写一句话:

```
Child 1 is sending a message!  
Child 2 is sending a message!
```

父进程从管道中读出二个来自子进程的信息并显示(要求先接收 P1,后 P2)。

```
/*lib1*/  
#include <unistd.h>  
#include <stdio.h>  
#include <stdlib.h>  
#include <string.h>  
#include <wait.h>  
  
int main( )  
{  
    int pid1, pid2;  
    int fd[2];  
    char outpipe[100], inpipe[100];  
    pipe(fd);  
    pid1 = fork();  
    if(pid1==0)  
    {  
        lockf(fd[1], 1, 0);
```



```
    sprintf(outpipe, "child 1 process is sending message!");
    write(fd[1], outpipe, 50);
    lockf(fd[1], 0, 0);
    exit(0);
}
else
{
    pid2 = fork();
    if(pid2==0)
    {
        lockf(fd[1], 1, 0);
        sprintf(outpipe, "child 2 process is sending message!");
        write(fd[1], outpipe, 50);
        lockf(fd[1], 0, 0);
        exit(0);
    }
    if (pid2 > 0)
    {
        wait(0);
        read(fd[0], inpipe, 50);
        printf("%s\n", inpipe);
        wait(0);
        read(fd[0], inpipe, 50);
        printf("%s\n", inpipe);
        exit(0);
    }
}
}
```

结果截图：

```
ai@ying:03:10:15:~/Documents/Git/ComputerSystem/Experiment/Experiment/2$ vi 3.1.c
ai@ying:03:12:29:~/Documents/Git/ComputerSystem/Experiment/Experiment/2$ gcc 3.1.c
ai@ying:03:13:00:~/Documents/Git/ComputerSystem/Experiment/Experiment/2$ ./a.out
child 1 process is sending message!
child 2 process is sending message!
```

图 3-1-1

实验指导

所涉及的系统调用

1、pipe()

建立一无名管道，系统调用格式

```
pipe(filedes)
```

参数定义

```
int pipe(filedes);  
int filedes[2];
```

其中, `filedes[1]` 是写入端, `filedes[0]` 是读出端。该函数使用头文件如下:

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

2、read()

系统调用格式

```
read(fd,buf,nbyte)
```

功能:从 `fd` 所指示的文件中读出 `nbyte` 个字节的数据,并将它们送至由指针 `buf` 所指示的缓冲区中。如该文件被加锁,等待,直到锁打开为止。

参数定义

```
int read(fd,buf,nbyte);  
int fd;  
char *buf;  
unsigned nbyte;
```

3、write()

系统调用格式

```
read(fd,buf,nbyte)
```

功能:把 **nbyte** 个字节的数据,从 **buf** 所指向的缓冲区写到由 **fd** 所指向的 文件中。如文件加锁,暂停写入,直至开锁。参数定义同 **read()**。

例 1 父子进程基于管道的简单通信。

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

int main()
{
    int fd[2],pid,n;
    char buffer[256],dat[20]="hello world\n";
    pipe(fd);
    pid=fork();
    if(pid==0)
    {
        close(fd[1]);
        n=read(fd[0],buffer,256);
        printf("child %d read %d bytes:%s",getpid(),n,buffer);
    }
    else
    {
        close(fd[0]);
        write(fd[1],dat,strlen(dat));
        printf("parent write%d byte: %s\n",strlen(dat),dat);
    }
    return 0;
}
```

观察程序执行的结果

结果截图：

```
ai@ying:03:32:07:~/Documents/Git/ComputerSystem/Experiment/Experiment/2$ ./a.out
parent write 12 bytes: hello world
child 8682 read 12 bytes:hello world
```

图 3-1-2

例 2 同一个进程树的兄弟进程通信

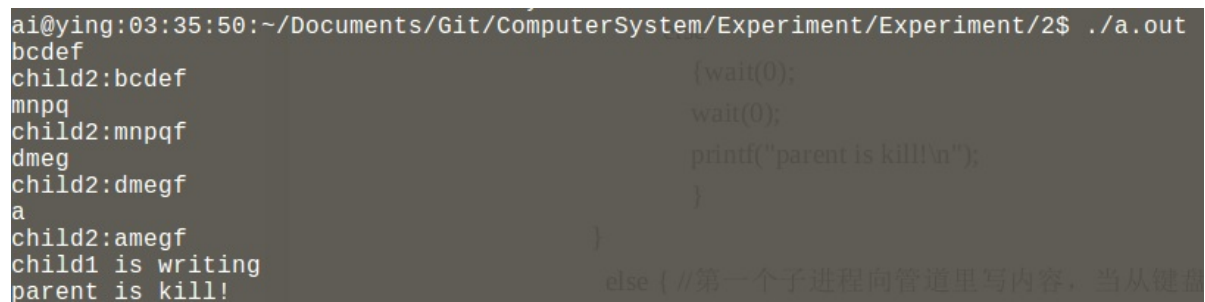
```
#include<stdio.h>
#include<unistd.h>
#include<stdlib.h>
#include<string.h>
#include<wait.h>

int main()
{
    int fd[2], i, n;
    pid_t pid, pir;
    char b[20] = "", dat[20] = "";
    pipe(fd);
    pid = fork();
    if (pid > 0)
    {
        pir = fork();
        if (pir == 0)
        {
            while (b[0] != 'a')
            {
                n = read(fd[0], b, 20);
                printf("child2:%s\n", b);
                sleep(1);
            }
        }
        else
        {
            wait(0);
            wait(0);
            printf("parent is kill!\n");
        }
    }
}
```

```
    }  
}  
  
else  
{  
    while (dat[0] != 'a')  
    {  
        scanf("%s", dat);  
        write(fd[1], dat, strlen(dat));  
        sleep(1);  
    }  
    printf("child1 is writing\n");  
}  
return 0;  
}
```

观察程序执行的结果

结果截图：



```
ai@ying:03:35:50:~/Documents/Git/ComputerSystem/Experiment/Experiment/2$ ./a.out  
bcdef  
child2:bcdef  
mnpq  
child2:mnpqf  
dmeg  
child2:dmegf  
a  
child2:amegf  
child1 is writing  
parent is kill!
```

Code snippets visible in the background of the terminal:

```
{ wait(0);  
wait(0);  
printf("parent is kill!\n");  
}  
else { //第一个子进程向管道里写内容，当从键盘
```

图 3-1-3

3-2消息队列通信

实验目的

- 1、了解什么是消息、消息队列
- 2、熟悉消息传送的机理

实验内容

消息的创建、发送和接收。使用系统调用 `msgget()`,`msgsnd()`,`msgrcv()`,及 `msgctl()` 编制一长度为1k 的消息发送和接收的程序。

实验指导

一、什么是消息

消息(message)是一个格式化的可变长的信息单元。消息机制允许由一个进程给其它任意的进程发送一个消息。当一个进程收到多个消息时,可将它们排成一个消息队列。消息使用二种重要的数据结构:一是消息首部,其中记录了一些与消息有关的信息,如消息数据的字节数;二个消息队列头表,其每一表项是作为一个消息队列的消息头,记录了消息队列的有关信息。

1、消息机制的数据结构

(1)消息首部

记录一些与消息有关的信息,如消息的类型、大小、指向消息数据区的指针、消息队列的链接指针等。

(2)消息队列头表

其每一项作为一个消息队列的消息头,记录了消息队列的有关信息如指向消息队列中第一个消息和指向最后一个消息的指针、队列中消息的数目、队列中消息数据的总字节数、队列所允许消息数据的最大字节总数,还有最近一次执行发送操作的进程标识符和时间、最近一次执行接收操作的进程标识符和时间等。

2、消息队列的描述符

UNIX 中,每一个消息队列都有一个称为关键字(key)的名字,是由用户指定的;消息队列有一消息队列描述符,其作用与用户文件描述符一样,也是为了方便用户和系统对消息队列的访问。

二、涉及的系统调用

1.msgget()

创建一个消息,获得一个消息的描述符。核心将搜索消息队列头表,确定是否有指定名字的消息队列。若无,核心将分配一新的消息队列头,并对它进行初始化,然后给用户返回一个消息队列描述符,否则它只是检查消息队列的许可权便返回。

系统调用格式:

```
msgqid=msgget(key, flag)
```

该函数使用头文件如下:

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

参数定义

```
int msgget(key, flag)
key_t key;
int flag;
```

其中:

key 是用户指定的消息队列的名字;flag 是用户设置的标志和访问方式。如 是否该队列已被创建。无则创建,是则打开; IPC_CREAT | 0400 IPC_EXCL | 0400 是否该队列的创建应是互斥的。 msgqid 是该系统调用返回的描述符,失败则返回-1。

2.msgsnd()

发送一消息。向指定的消息队列发送一个消息,并将该消息链接到该消息队列的尾部。

系统调用格式:

```
msgsnd(msgqid,msgp,size,flag)
```

该函数使用头文件如下:

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

参数定义:

```
int msgsnd(msgqid,msgp,size,flag)
int msgqid,size,flag;
struct msgbuf * msgp;
```

其中 `msgqid` 是返回消息队列的描述符; `msgp` 是指向用户消息缓冲区的一个结构体指针。缓冲区中包括消息类型和消息正文,即

```
{
    long mtype; /*消息类型*/
    char mtext[ ]; /*消息的文本*/
}
```

`size` 指示由 `msgp` 指向的数据结构中字符数组的长度;即消息的长度。这个数组的最大值由 `MSG-MAX()` 系统可调用参数来确定。`flag` 规定当核心用尽内部缓冲空间时应执行的动作:进程是等待,还是立即返回。若在标志 `flag` 中未设置 `IPC_NOWAIT` 位,则当该消息队列中的字节数超过最大值时,或系统范围的消息数超过某一最大值时,调用 `msgsnd` 进程睡眠。若是设置 `IPC_NOWAIT`,则在此情况下,`msgsnd` 立即返回。`Flag` 值为 0:忽略标志位。为 `IPC_NOWAIT`:如果消息队列已满,消息将不被写入队列,控制权返回调用函数的进程。如果不指定这个参数,线程将被阻塞直到消息可以被写入。对于 `msgsnd()`,核心须完成以下工作:

(1)对消息队列的描述符和许可权及消息长度等进行检查。若合法才继续执行,否则返回;

(2)核心为消息分配消息数据区。将用户消息缓冲区中的消息正文,拷贝到消息数据区;

(3)分配消息首部,并将它链入消息队列的末尾。在消息首部中须填写消息类型、消息大小和指向消息数据区的指针等数据;

(4)修改消息队列头中的数据,如队列中的消息数、字节总数等。最后,唤醒等待消息的进程。

3.msgrcv()

接受一消息。从指定的消息队列中接收指定类型的消息。

系统调用格式:

```
msgrcv(msgqid,msgp,size,type,flag)
```

本函数使用的头文件如下:

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

参数定义:

```
int msgrcv(msgqid,msgp,size,type,flag)
int msgqid,size,flag;
struct msgbuf *msgp;
long type;
```

其中,msgqid,msgp,size,flag 与 msgsnd 中的对应参数相似,type 是规定要读的消息类型,flag 规定倘若该队列无消息,核心应做的操作。如此时设置了 IPC_NOWAIT 标志,则立即返回,若在 flag 中设置了 MS_NOERROR,且所接收的消息大于 size,则核心截断所接收的消息。对于 msgrcv 系统调用,核心须完成下述工作:

(1)对消息队列的描述符和许可权等进行检查。若合法,就往下执行;否 则返回;

(2)根据 `type` 的不同分成三种情况处理:

`type=0`,接收该队列的第一个消息,并将它返回给调用者;

`type` 为正整数,接收类型 `type` 的第一个消息;

`type` 为负整数,接收小于等于 `type` 绝对值的最低类型的第一个消息。

(3)当所返回消息大小等于或小于用户的请求时,核心便将消息正文拷贝 到用户区,并从消息队列中删除此消息,然后唤醒睡眠的发送进程。但如果消息 长度比用户要求的大时,则做出错返回。

4.msgctl()

消息队列的操纵。读取消息队列的状态信息并进行修改,如查询消息队列描 述符、修改它的许可权及删除该队列等。

系统调用格式:

```
msgctl(msgqid,cmd,buf);
```

本函数使用的头文件如下:

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

参数定义:

```
int msgctl(msgqid,cmd,buf);
int msgqid,cmd;
struct msgqid_ds *buf;
```

其中,函数调用成功时返回 0,不成功则返回-1。`buf` 是用户缓冲区地址,供用户存放控制参数和查询结果;`cmd` 是规定的命令。命令可分三类:

(1) `IPC_STAT`。查询有关消息队列情况的命令。如查询队列中的消息数目、队列中的最大字节数、最后一个发送消息的进程标识符、发送时间等;

(2)IPC_SET。按 buf 指向的结构中的值,设置和改变有关消息队列属性的命令。如改变消息队列的用户标识符、消息队列的许可权等;

(3)IPC_RMID。消除消息队列的标识符。

msgqid_ds结构定义如下:

```
struct msgqid_ds
{
    struct ipc_perm msg_perm; /*许可权结构*/
    short pad1[7]; /*由系统使用*/
    ushort msg_qnum; /*队列上消息数*/
    ushort msg_qbytes; /*队列上最大字节数*/
    ushort msg_lspid; /*最后发送消息的 PID*/
    ushort msg_lrpid; /*最后接收消息的 PID*/
    time_t msg_stime; /*最后发送消息的时间*/
    time_t msg_rtime; /*最后接收消息的时间*/
    time_t msg_ctime; /*最后更改时间*/
};

struct ipc_perm
{
    ushort uid; /*当前用户*/
    ushort gid; /*当前进程组*/
    ushort cuid; /*创建用户*/
    ushort cgid; /*创建进程组*/
    ushort mode; /*存取许可权*/
    { short pid1; long pad2; } /*由系统使用*/
}
```

三、参考程序

```
/*client.c*/
#include <unistd.h>
#include <sys/types.h>
#include <sys/msg.h>
#include <sys/ipc.h>
#include <stdio.h>
#include <stdlib.h>
#define MSGKEY 75          //定义一个消息队列

struct msgform            //创建一个消息结构体
{
    long mtype;
    char mtext[1000];
}msg;
int msgqid;

void client()
{
    int i;
    msgqid = msgget(MSGKEY, 0777); //创建一个消息队列，队列号为75，
    //其中0777表示消息队列的执行权限，表示所有用户都可以运行
    for (i = 10; i >= 1; i--)
    {
        msg.mtype = i;
        printf("(client)sent!\n");
        msgsnd(msgqid, &msg, 1024, 0); //发送消息
        sleep(1);
    }
    exit(0);
}

int main()
{
    client();
    return 0;
}
```

```
/*server.c*/
#include <unistd.h>
#include <sys/types.h>
#include <sys/msg.h>
#include <sys/ipc.h>
#include <stdio.h>
#include <stdlib.h>
#define MSGKEY 75

struct msgform
{
    long mtype;
    char mtext[1000];
}msg;

int msgqid;

void server()
{
    msgqid = msgget(MSGKEY, 0777|IPC_CREAT); //创建消息队列
    do
    {
        msgrcv(msgqid, &msg, 1030, 0, 0); //接收消息
        printf("(server)received!\n");
        sleep(1);
    }while(msg.mtype != 1);
    msgctl(msgqid, IPC_RMID, 0); //消除消息队列
    exit(0);
}

int main()
{
    server();
    return 0;
}
```

四、程序说明

- 1、为了便于操作和观察结果,编制二个程序 `client.c` 和 `server.c`,分别用于 消息的发送与接收。
- 2、`server` 建立一个 Key 为 75 的消息队列,等待其它进程发来的消息。当 遇到类型为 1 的消息,则作为结束信号,取消该队列,并退出 `server`。`server` 每 接收到一个消息后显示一句“(server)received。”
- 3、`client` 使用 key 为 75 的消息队列,先后发送类型从 10 到 1 的消息,然 后退出。最后一个消息,即是 `server` 端需要的结束信号。`client` 每发送一条消息 后显示一句“(client)sent”。
- 4、注意: 二个程序分别编辑、编译为 `client` 与 `server`。为了看清结果要在两个终端 执行:

```
./server  
ipcs -q  
./client。
```

结果截图：



```
ai@ying:~/Documents/Git/ComputerSystem/Experiment/Experiment/2$ ls  
3.1.c 3.3.c client.c message.c server.c  
ai@ying:~/Documents/Git/ComputerSystem/Experiment/Experiment/2$ ./client  
(client)sent!,mtype = 10  
(client)sent!,mtype = 9  
(client)sent!,mtype = 8  
(client)sent!,mtype = 7  
(client)sent!,mtype = 6  
(client)sent!,mtype = 5  
(client)sent!,mtype = 4  
(client)sent!,mtype = 3  
(client)sent!,mtype = 2  
(client)sent!,mtype = 1  
ai@ying:~/Documents/Git/ComputerSystem/Experiment/Experiment/2$  
ai@ying:~/Documents/Git/ComputerSystem/Experiment/Experiment/2$ ls  
3.1.c 3.3.c client.c message.c server.c  
ai@ying:~/Documents/Git/ComputerSystem/Experiment/Experiment/2$ ./server  
(server)received! mtype = 10  
(server)received! mtype = 9  
(server)received! mtype = 8  
(server)received! mtype = 7  
(server)received! mtype = 6  
(server)received! mtype = 5  
(server)received! mtype = 4  
(server)received! mtype = 3  
(server)received! mtype = 2  
(server)received! mtype = 1  
ai@ying:~/Documents/Git/ComputerSystem/Experiment/Experiment/2$
```

图 3-2-1

```

ai@ying:~/Documents/Git/ComputerSystem/Experiment/Experiment/2$ ls
3.1.c 3.2.c 3.3.c client client.c lab1.c message.c server server.c
ai@ying:~/Documents/Git/ComputerSystem/Experiment/Experiment/2$ ./server &
[1] 3213
ai@ying:~/Documents/Git/ComputerSystem/Experiment/Experiment/2$ ipcs -q

----- Message Queues -----
key          msqid        owner        perms        used-bytes   messages
0x0000004b  262144       ai           777          0             0

ai@ying:~/Documents/Git/ComputerSystem/Experiment/Experiment/2$ ./client
(client)sent!,mtype = 10
(server)received! mytype = 10
(client)sent!,mtype = 9
(server)received! mytype = 9
(client)sent!,mtype = 8
(server)received! mytype = 8
(client)sent!,mtype = 7
(server)received! mytype = 7
(client)sent!,mtype = 6
(server)received! mytype = 6
(client)sent!,mtype = 5
(server)received! mytype = 5
(client)sent!,mtype = 4
(server)received! mytype = 4
(client)sent!,mtype = 3
(server)received! mytype = 3
(client)sent!,mtype = 2
(server)received! mytype = 2
(client)sent!,mtype = 1
(server)received! mytype = 1
[1]+  Done                  ./server
ai@ying:~/Documents/Git/ComputerSystem/Experiment/Experiment/2$

```

图 3-2-2

五、运行结果

分析：

为了更容易看清结果，对程序进行部分修改，两个源程序在原来的基础上添加了，`sleep()` 函数，并且打印消息类型，更能容易看清消息队列运行的机制。

根据运行结果可以看出，`client`每发送一条消息，`server`就接受一条消息。依次执行下去直至运行结束。

3-3共享存储区通信

实验目的

了解和熟悉共享存储机制

实验内容

编制一长度为 1k 的共享存储区发送和接收的程序。

实验指导

一、共享存储区

共享存储区机制的概念

共享存储区(ShareMemory)是 UNIX 系统中通信速度最高的一种通信机制。该机制可使若干进程共享主存中的某一个区域,且使该区域出现(映射)在多个进程的虚地址空间中。另一方面,一个进程的虚地址空间中又可连接多个共享存储区,每个共享存储区都有自己的名字。当进程间欲利用共享存储区进行通信时,必须先在主存中建立一共享存储区,然后将它附接到自己的虚地址空间上。此后,进程对该区的访问操作,与对其虚地址空间的其它部分的操作完全相同。进程之间便可通过对共享存储区中数据的读、写来进行直接通信。

二、涉及的系统调用

1、shmget()

创建、获得一个共享存储区。

系统调用格式:

```
shmid=shmget(key, size, flag)
```

该函数使用头文件如下:


```
#include<sys/types.h>
#include<sys/ipc.h>
#include<sys/shm.h>
```

参数定义

```
int shmget(key, size, flag);
key_t key;
int size, flag;
```

其中, **key** 是共享存储区的名字; **size** 是其大小(以字节计); **flag** 是用户设置的标志, 如 **IPC_CREAT**。 **IPC_CREAT** 表示若系统中尚无指名的共享存储区, 则由核心建立一个共享存储区; 若系统中已有共享存储区, 便忽略 **IPC_CREAT**。 附:

操作允许权	八进制数
-------	------

用户可读	00400
------	-------

用户可写	00200
------	-------

小组可读	00040
------	-------

小组可写	00020
------	-------

其它可读	00004
------	-------

其它可写	00002
------	-------

控制命令	值
------	---

IPC_CREAT	0001000
-----------	---------

IPC_EXCL	0002000
----------	---------

例:

```
shmid=shmget(key, size, (IPC_CREAT|0400))
```

创建一个关键字为 **key**, 长度为 **size** 的共享存储区

2、shmat()

共享存储区的附接。从逻辑上将一个共享存储区附接到进程的虚拟地址空间上。

系统调用格式:

```
virtaddr=shmat(shmid, addr, flag)
```

该函数使用头文件如下:

```
#include<sys/types.h>
#include<sys/ipc.h>
#include<sys/shm.h>
```

参数定义

```
char *shmat(shmid, addr, flag);
int shmid, flag;
char * addr;
```

其中,shmid 是共享存储区的标识符;addr 是用户给定的,将共享存储区附接到进程的虚地址空间;flag 规定共享存储区的读、写权限,以及系统是否应对用户规定的地址做舍入操作。其值为 SHM_RDONLY 时,表示只能读;其值为 0 时,表示可读、可写;其值为 SHM_RND(取整)时,表示操作系统在必要时舍去这个地址。该系统调用的返回值是共享存储区所附接到的进程虚地址 viraddr。

3、shmdt()

把一个共享存储区从指定进程的虚地址空间断开。

系统调用格式:

```
shmdt(addr)
```

该函数使用头文件如下:

```
#include<sys/types.h>
#include<sys/ipc.h>
#include<sys/shm.h>
```

参数定义

```
int shmdt(addr);
char addr;
```

其中,addr 是要断开连接的虚地址,亦即以前由连接的系统调用 shmat()所返回的虚地址。调用成功时,返回 0 值,调用不成功,返回-1。

4、shmctl()

共享存储区的控制,对其状态信息进行读取和修改。

系统调用格式:

```
shmctl(shmid,cmd,buf)
```

该函数使用头文件如下:

```
``c
#include<sys/types.h>
#include<sys/ipc.h>
#include<sys/shm.h>
```

参数定义

```
int shmctl(shmid,cmd,buf);
int shmid,cmd;
struct shmid_ds *buf;
```

其中,buf 是用户缓冲区地址,cmd 是操作命令。命令可分为多种类型:

(1)用于查询有关共享存储区的情况。如其长度、当前连接的进程数、共享区的创建者标识符等;

(2)用于设置或改变共享存储区的属性。如共享存储区的许可权、当前连接的进程计数等;

(3)对共享存储区的加锁和解锁命令;

(4)删除共享存储区标识符等。

上述的查询是将 `shmid` 所指示的数据结构中的有关成员,放入所指示的缓冲区中;而设置是用由 `buf` 所指示的缓冲区内容来设置由 `shmid` 所指示的数据结构 中的相应成员。

三、参考程序

```
/*lab3*/

#include <stdio.h>
#include <stdlib.h>
#include <wait.h>
#include <sys/types.h>
#include <unistd.h>
#include <sys/shm.h>
#include <sys/ipc.h>
#define SHMKEY 75      //定义共享存储区号
int shmid, i;
int *addr;

void client()
{
    int i;
    shmid = shmget(SHMKEY, 1024, 0777); //打开一个共享存储区
    addr = shmat(shmid, 0, 0);          //获取共享存储区的首地址
    for (i = 9; i >= 0; i--)
    {
        while(*addr != -1);
        printf("(client)sent\n");
        *addr = i;
    }
    exit(0);
}
```

```

void server()
{
    shmid = shmget(SHMKEY, 1024, 0777|IPC_CREAT); //创建一个共享存储区
    addr = shmat(shmid, 0, 0); //获取贡献存储区的地址
    do
    {
        *addr = -1;
        while (*addr == -1);
        printf("(server)receive\n");
    }while(*addr);
    shmctl(shmid, IPC_RMID, 0);
    exit(0);
}

int main()
{
    i = fork();
    if (!i)
        server();
    system("ipcs -m");
    i = fork();
    if (!i)
        client();
    wait(0);
    wait(0);
    return 0;
}

```

四、程序说明

1、为了便于操作和观察结果,用一个程序作为“引子”,先后 fork() 两个子进程,server 和 client,进行通信。

2、server 端建立一个 key 为 75 的共享区,并将第一个字节置为-1,作为数据空的标志。等待其他进程发来的消息。当该字节的值发生变化时,表示收到了信息,进行处理。然后再次把它的值设为-1,如果遇到的值为 0,则视为为结束信号,取消该

队列,并退出 server。server 每接收到一次数据后显示“(server)received”。

3、client 端建立一个 key 为 75 的共享区,当共享取得第一个字节为-1 时,server 端空闲,可发送请求。client 随即填入 9 到 0。期间等待 server 端的再次空闲。进行完这些操作后,client 退出。client 每发送一次数据后显示“(client)sent”。

4、父进程在 server 和 client 均退出后结束。

五、运行结果

结果截图：

```
ai@ying:10:06:24:~/Documents/Git/ComputerSystem/Experiment/Experiment/2$ ./shm
----- Shared Memory Segments -----
key          shmid      owner    perms    bytes       nattch     status
0x00000000   851968     ai       600      524288      2          dest
0x00000000   524289     ai       600      524288      2          dest
0x00000000   196610     ai       600      16777216    2          dest
0x00000000   1277955    ai       600      524288      2          dest
0x00000000   655364     ai       600      524288      2          dest
0x00000000   950277     ai       600      524288      2          dest
0x00000000   1310726    ai       600      524288      2          dest
0x00000000   1179655    ai       600      67108864    2          dest
0x00000000   1409032    ai       600      524288      2          dest
0x00000000   1507337    ai       600      524288      2          dest
0x00000000   3112970    ai       600      2097152     2          dest
0x00000000   1638411    ai       600      393216      2          dest
0x00000000   1671180    ai       600      524288      2          dest
0x00000000   1900557    ai       600      524288      2          dest
0x00000000   2129934    ai       700      141148      2          dest
0x00000000   1933327    ai       600      4194304     2          dest
0x00000000   1966096    ai       600      33554432    2          dest
0x00000004b  3244049    ai       777      1024        1          dest

(client)sent
(server)receive
(client)sent
(server)receive
(client)sent
(server)receive
(client)sent
(server)receive
(client)sent
(server)receive
(client)sent
(server)receive
(client)sent
(server)receive
(client)sent
(server)receive
(ai@ying: ~/Documents/Git/ComputerSystem/Experiment/Experiment/2
```

图 3-3-1

3-4信号量机制

实验目的

了解和熟悉基于信号量机制实现 P、V 原语的相关操作。

实验内容

利用信号量机制和共享存储区实现生产者和消费者模型。

实验指导

一、信号量机制介绍

信号量机制是 UNIX/Linux 的 IPC 机制的一种,它同消息队列和共享存储区不同的是,它不是主要用于通信的双方进程之间进行数据通信的,而是用于通信双方之间的同步或互斥控制。在操作系统原理书中讲述的使用 PV 原语实现进程同步与互斥机制,在本实验中可用基于信号量机制实现 PV 原语。

二、涉及的系统调用

下面的系统调用函数均用到以下头文件:

```
#include <sys/types.h>
#include <sys/ipc.h>
#include <sys/sem.h>
```

1、信号量结构

```
struct sem
{
    ushort semval ; /* 信号量的值,总是大于 1 */
    pid_t sempid ; /* 最后一个操作信号量的进程 id 号*/
    ushort semncnt ; /* 等待着信号量值增加的进程数 */
    ushort semzcnt ; /* 等待着信号量值为零的进程数 */
} ;
```

2. 信号量创建

```
int semget(key_t key, int nsems, int flag) ;
```

key,flag 两个参数类似于 msgget 和 shmget 中的 key 和 flag。nsems 是该集合中的信号量数。如果是创建新集合(一般在服务器中),则必须指定 nsems。例如创建一个含有两个元素的信号量为:

```
int semid=semget(567,2,0777|IPC_CREAT);
```

3.信号量控制函数

```
int semctl(int semid, int semnum, int cmd, union semun arg) ;
```

该函数用于完成对信号量的删除、设置初始值或读取信号量的值等操作,具体的操作是由 cmd 参数决定的。

(1)联合体参数 arg 的定义如下:

```
union semun
{
    int val ;/* for SETVAL */
    struct semid_ds * buf ; /* for IPC_STAT and IPC_SET */
    ushort *array ; /* for GETALL and SETALL */
} ;
```


(2)命令参数 `cmd` 有如下命令:

- `GETVAL` 返回成员 `semnum` 的 `semval` 值。
- `SETVAL` 设置成员 `semnum` 的 `semval` 值。该值由 `arg.val` 指定。
- `GETPID` 返回成员 `semnum` 的 `sempid` 值。
- `GETNCNT` 返回成员 `semnum` 的 `semncnt` 值。
- `GETZCNT` 返回成员 `semnum` 的 `semzcnt` 值。
- `GETALL` 取该集合中所有信号量的值,并将它们存放在由 `arg.array` 指向的数组中。
- `SETALL` 按 `arg.array` 指向的数组中的值设置该集合中所有信号量的值。

4.信号量操作

```
int semop(int semid, struct sembuf *sops, size_t nops) ;
```

其中结构体参数 `sops` 定义如下:

```
struct sembuf
{
    short sem_num ;/* 信号量数组的序号为(0,1,...,nsems-1*/
    short sem_op ;/* 信号量操作-1 相当于 P 操作,+1 相当于 V 操作*/
    short sem_flg ;/* IPC_NOWAIT, SEM_UNDO */
} ;
```

`sem_op` 规定。此值可以是负值、0 或正值。

>0 : `sem_op` 值加到信号量的值上。1 相当于 v 操作。
 <0: 则从信号量值中减去 `sem_op` 的绝对值, -1 相当于 p 操作
 如果信号量的值 $\geq \text{abs}(\text{sem_op})$, 操作成功, 否则, 当未置 `IPC_NOWAIT` 时,
 进程将等待信号量的值 $\geq \text{abs}(\text{sem_op})$
 =0, 这表示希望等待到该信号量值变成 0

如果信号量值小于 `sem_op` 的绝对值(资源不能满足要求), 则:

(a) 若指定了 `IPC_NOWAIT`, 则出错返回 `EAGAIN` ;

(b) 若未指定 `IPC_NOWAIT`, 则该信号量的 `semncnt` 值加 1 (因为将进入睡眠状态), 然后调用进程被挂起直至下列事件之一发生:

i. 此信号量变成大于或等于 `sem_op` 的绝对值 (即某个进程已释放了某些资源)。此信号量的 `semncnt` 值减 1 (因为已结束等待), 并且从信号量值中减去 `sem_op` 的绝对值。如果指定了 `undo` 标志, 则 `sem_op` 的绝对值也加到该进程的此信号量调整值上。

ii. 从系统中删除了此信号量。在此情况下, 函数出错返回 `ERMID`。

iii. 进程捕捉到一个信号, 并从信号处理程序返回, 在此情况下, 此信号量的 `semncnt` 值减 1 (因为不再等待), 并且函数出错返回 `EINTR`。

三、参考程序

注意：源代码部分错误，已修改。

```
/*sem_head.h*/

#ifndef _SEM_HEAD_H
#define _SEM_HEAD_H
#include <sys/types.h>
#include <sys/ipc.h>
#include <unistd.h>
#include <sys/sem.h>
#include <sys/shm.h>
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <wait.h>

union semun
{
    int val;
    struct semid_ds *buf;
    unsigned short *array;
};
```

```
//创建信号量
int open_semaphore_set(key_t keyval, int numsems)
{
    int sid;
    sid = semget(keyval, numsems, IPC_CREAT|0660);
    return sid;
}

//初始化信号量
void init_a_semaphore(int sid, int semnum, int initval)
{
    union semun semopts;
    semopts.val = initval;
    semctl(sid, semnum, SETVAL, semopts);
}

//删除信号量
int rm_semaphore(int sid)
{
    return (semctl(sid, 0, IPC_RMID, 0));
}

//信号量的P操作
int semaphore_P(int sem_id)
{
    struct sembuf sb;
    sb.sem_num = 0;
    sb.sem_op = -1;
    sb.sem_flg = SEM_UNDO;
    if (semop(sem_id, &sb, 1) == -1)
    {
        printf("semaphore_P failed.\n");
        return 0;
    }
    return 1;
}

//信号量的V操作
int semaphore_V(int sem_id)
```

```
{
    struct sembuf sb;
    sb.sem_num = 0;
    sb.sem_op = 1;
    sb.sem_flg = SEM_UNDO;
    if (semop(sem_id, &sb, 1) == -1)
    {
        printf("semaphore_V failed.\n");
        return 0;
    }
    return 1;
}
#endif
```

```
/*consumer.c*/

#include "sem_head.h"

int main()
{
    //定义共享存储区结构体
    struct exchange
    {
        char buf[BUFSIZ+80];
        int seq;
    }shm;

    int shmid;
    unsigned char *retval;
    int producer, consumer, i;

    //创建信号量consumer
    consumer = open_semaphore_set(ftok("consumer", 0), 1);

    //初始化信号量consumer的值为1
    init_a_semaphore(consumer, 0, 1);
```

```

//创建信号量producer
producer = open_semaphore_set(ftok("producer", 0), 1);

//初始化信号量producer的值为1
init_a_semaphore(producer, 0, 1);

//创建共享存储区用于存放生产者产生的数据
shmid = shmget(ftok("share", 0), sizeof(struct exchange),
0666|IPC_CREAT);
retval = shmat(shmid, (unsigned char *)0, 0);

//进行生产与消费的同步
for (i = 0; ; i++)
{
    semaphore_V(consumer);
    printf("data receive:%s ,sequence:%d\n", retval,
shm.seq);
    semaphore_P(producer);
    if (strncmp(retval, "end", 3) == 0)
        break;
}

rm_semaphore(producer);
rm_semaphore(consumer);
exit(0);
return 0;
}

```

```

/*producer.c*/

#include "sem_head.h"

int main()
{
    struct exchange
    {
        char buf[BUFSIZ+80];

```

```
    int seq;
}shm;
int shmidx;
unsigned char *retval;
int producer, consumer, i;
char readbuf[BUFSIZ];

//创建信号量consumer
consumer = open_semaphore_set(ftok("consumer", 0), 1);

//初始化信号量consumer的值为1
init_a_semaphore(consumer, 0, 1);

//创建信号量producer
producer = open_semaphore_set(ftok("producer", 0), 1);

//初始化信号量producer的值为1
init_a_semaphore(producer, 0, 1);

//创建共享存储区用于存放生产者产生的数据
shmidx = shmget(ftok("share", 0), sizeof(struct exchange), 0666|IPC_CREAT);
retval = shmat(shmidx, (unsigned char *)0, 0);

//生产者与消费者同步
for (i = 0; ; i++)
{
    printf("enter some text:");
    fgets(readbuf, BUFSIZ, stdin);
    semaphore_P(consumer);
    sprintf(retval, "messge %2d form producer %d is \"%s\"", i, producer, readbuf);
    semaphore_V(producer);
    if (strncmp(readbuf, "end", 3) == 0)
        break;
}
exit(0);

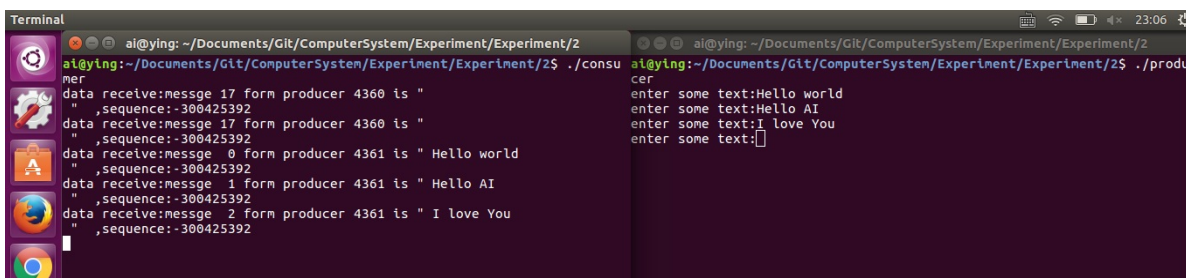
return 0;
}
```

四、程序说明

上面的 3 个程序是用于模拟生产者与消费者的一个例子,生产者从键盘输入产生数据,将数据存入缓冲区中,并通过信号量唤醒消费者。消费者从缓冲区中读出数据并将其输出,并唤醒生产者使其再生产,如此循环,直到生产者输入“end”后都结束。

无、运行结果

结果截图：



```

Terminal
ai@ying: ~/Documents/Git/ComputerSystem/Experiment/Experiment/2
ai@ying:~/Documents/Git/ComputerSystem/Experiment/Experiment/2$ ./consumer
data receive:msgsg 17 form producer 4360 is "
",sequence:-300425392
data receive:msgsg 17 form producer 4360 is "
",sequence:-300425392
data receive:msgsg 0 form producer 4361 is " Hello world
",sequence:-300425392
data receive:msgsg 1 form producer 4361 is " Hello AI
",sequence:-300425392
data receive:msgsg 2 form producer 4361 is " I love You
",sequence:-300425392

ai@ying:~/Documents/Git/ComputerSystem/Experiment/Experiment/2$ ./producer
enter some text:Hello world
enter some text:Hello AI
enter some text:I love You
enter some text:

```

图 3-4-1

```

ai@ying:11:10:16:~/Documents/Git/ComputerSystem/Experiment/Experiment/2$ ls
2client.c 3.2.c a.out client.c consumer.c message.c producer.c server shm
3.1.c 3.3.c client consumer lab1.c producer sem_head.h server.c singals
ai@ying:11:10:18:~/Documents/Git/ComputerSystem/Experiment/Experiment/2$ ./consumer &
[3] 4483
data receive:msgsg 4 form producer 4478 is "
",sequence:-1100328096
data receive:msgsg 4 form producer 4478 is "
",sequence:-832499008
ai@ying:11:10:26:~/Documents/Git/ComputerSystem/Experiment/Experiment/2$ ./producer
data receive:msgsg 4 form producer 4478 is "
",sequence:-1100328096
enter some text:Hello World!
enter some text:data receive:msgsg 0 form producer 4484 is " Hello World!
",sequence:-832499008
Hello AI!
data receive:msgsg 1 form producer 4484 is " Hello AI!
",sequence:-1100328096
enter some text:I love You!
enter some text:data receive:msgsg 2 form producer 4484 is " I love You!
",sequence:-832499008

```

图 3-4-2

实验四、内核编译和系统调用

4-1 编译内核

实验目的

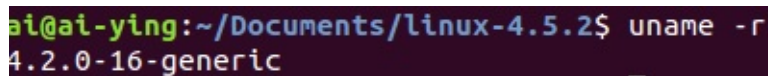
- 1、学习和动手编译 Linux 内核
- 2、熟悉编写系统调用的过程

实验内容

下载并编译 Linux 内核,将新内核安装到已有的 Linux 发行版中。

实验指导

- 1、从 www.kernel.org 上下载内核文档 `linux-4.0.0.tar.gz`。查看编译新内核前,系统内核版本。



```
ai@ai-ying:~/Documents/linux-4.5.2$ uname -r
4.2.0-16-generic
```

图 4-1-1

- 2、解压内核文档 `linux-4.0.0.tar.gz`
- 3、配置内核

```
make menuconfig
```

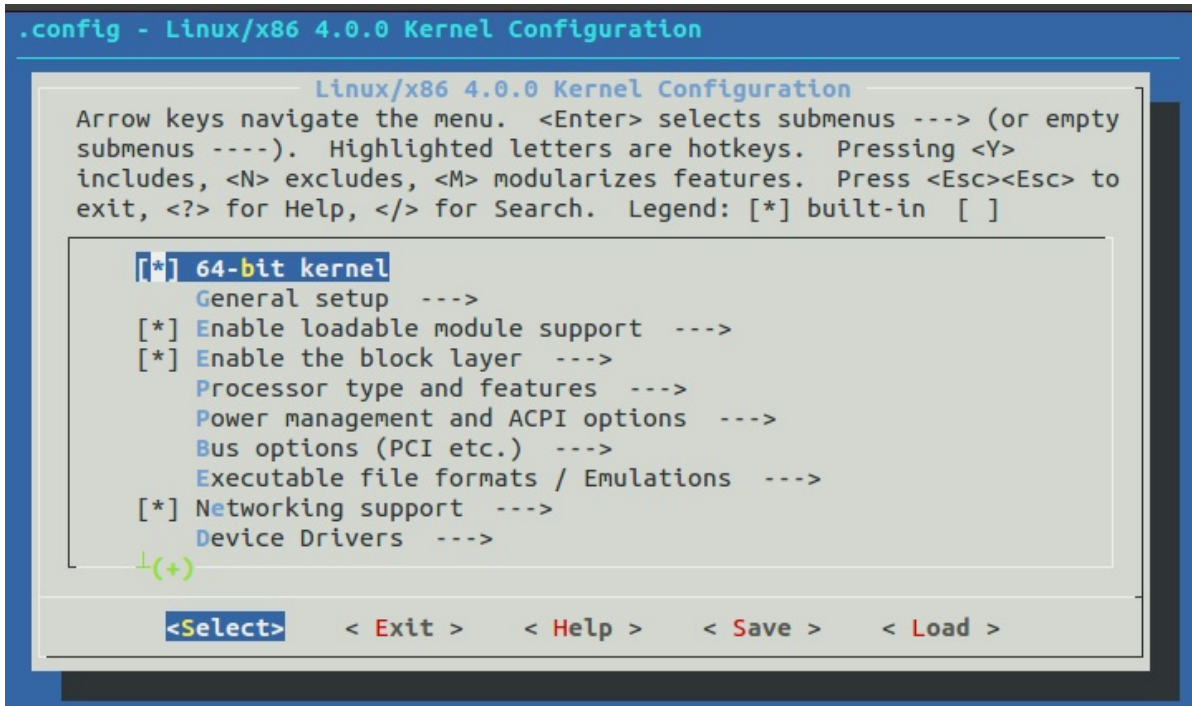


图 4-1-2

4、编译内核(需要较长时间)

```
make
make modules
```

```
root@ai-ying:/usr/src/linux-4.0# make modules
CHK      include/config/kernel.release
CHK      include/generated/uapi/linux/version.h
CHK      include/generated/utsrelease.h
CALL     scripts/checksyscalls.sh
Building modules, stage 2.
MODPOST 4305 modules
```

图 4-1-3

5、安装内核

```
make modules_install
make install
```

6、重启,查看当前内核版本

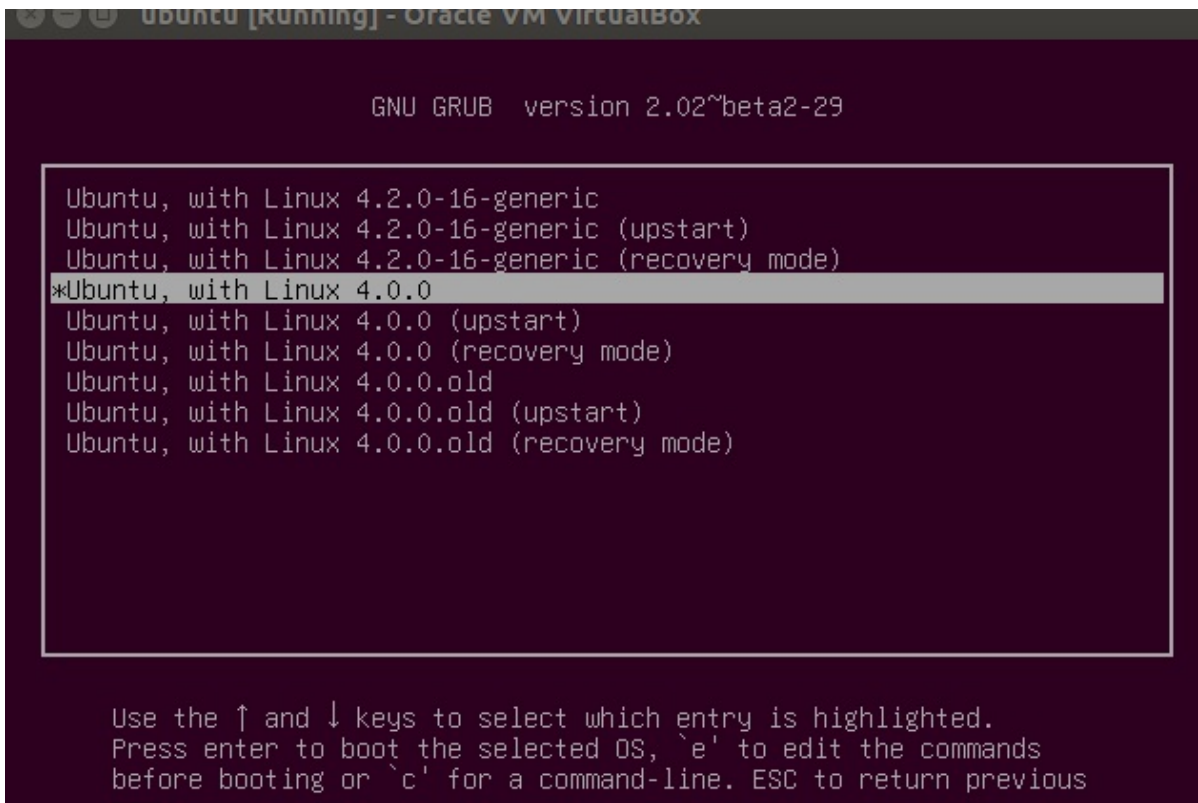


图 4-1-4



图 4-1-5

分析：编译内核时出现了一些错误，会需要一些依赖库的安装。

错误1.scripts/kconfig/lxdialog/dialog.h:38:20: fatal error: curses.h: No such file or directory

解决方法：sudo apt install ncurses-dev

错误2.scripts/sign-file.c:23:30: fatal error: openssl/opensslv.h: No such file or directory

解决方法：sudo apt install libssl-dev

编译安装后，需要对grub进行部分修改，否则系统默认使用的内核仍是原来版本。对grub的修改目的是，开机进入设置选项。修改如下：把第二行的注释掉即可。

```
GRUB_DEFAULT=0
#GRUB_HIDDEN_TIMEOUT=0
GRUB_HIDDEN_TIMEOUT_QUIET=true
GRUB_TIMEOUT=10
GRUB_DISTRIBUTOR=`lsb_release -i -s 2> /dev/null || echo Debian`
GRUB_CMDLINE_LINUX_DEFAULT="quiet splash"
GRUB_CMDLINE_LINUX=""
```

为了提高编译速度，节约编译时间。可以开机使用文本界面进行编译，编译好后再切换到图形界面。（时间几乎可以节约一半）

1.开机进入文本字符界面。

```
sudo systemctl set-default multi-user.target
```

2.开机进入图形界面。

```
sudo systemctl set-default graphical.target
```

4-2 系统调用

实验目的

- 1、了解系统调用
- 2、编写自己定义的系统调用,将其编译进内核

实验内容

(1)分配系统调用号:include/uapi/asm-generic/unistd.h。

```
#define __NR_ai 666  
__SYSCALL(__NR_ai, sys_ai)
```

同时更改 `#define __NR_syscalls 282` 的数目。因为系统调用增加1，总数目也要增加。

```
#define __NR_syscalls 283
```

```
__SYSCALL(__NR_bpf, sys_bpf)  
#define __NR_execveat 281  
__SC_COMP(__NR_execveat, sys_execveat, compat_sys_execveat)  
  
/*my syscalls*/  
#define __NR_ai 666  
__SYSCALL(__NR_ai, sys_ai)  
  
#undef __NR_syscalls  
#define __NR_syscalls 283_
```

图4-2-1

(2)修改系统调用表:arch/x86/syscalls/syscall_64.tbl。(如果是32位虚拟机的话，应该修改syscall_32.tbl)

```
666      x64      ai      sys_ai
```

注意一定要在**x64**的模块添加（一般是在系统调用号为**322**后面添加），如果在文件尾部添加，编译时会报错

```
319      common  memfd_create      sys_memfd_create
320      common  kexec_file_load    sys_kexec_file_load
321      common  bpf                 sys_bpf
322      64      execveat            stub_execveat

#my syscalls
666      64      ai                 sys_ai_
```

图4-2-2

(3)修改头文件 include/linux/syscalls.h,添加函数声明

```
asmlinkage int sys_ai(void);
```

```
asmlinkage long sys_execveat(int dirfd, const char __user *filename,
                             const char __user *const __user *argv,
                             const char __user *const __user *envp, int flags);

/*my syscalls*/
asmlinkage int sys_ai(void);
```

图4-2-3

(4)添加处理函数:kernel/sys.c

```

asmlinkage int sys_ai(void)
{
    struct task_struct *p;
    printk("*****\n");
    printk("-----the output of rkcall-----\n");
    printk("*****\n");
    printk("%-20s %-6s %-6s %-20s", "Name", "pid", "state", "Parent");
    for (p = &init_task; (p = next_task(p)) != &init_task;)
    {
        printk("%-20s %-6d %-6d %-20s\n", p->comm, p->pid,
            p->state, p->parent->comm);
    }
    return 0;
}

```

```

        return 0;
    }

/*my syscalls*/
asmlinkage int sys_ai(void)
{
    struct task_struct *p;
    printk("*****\n");
    printk("-----the output of rkcall-----\n");
    printk("*****\n");
    printk("%-20s %-6s %-6s %-20s\n", "Name", "pid", "state", "ParentName");
    for (p = &init_task; (p = next_task(p)) != &init_task;)
        printk("%-20s %-6d %-6d %-20s\n", p->comm, p->pid, p->state, p->parent->
comm);
    return 0;
}

```

图 4-2-4

执行如下几个命令:

1. 编译内核

```

make
make modules

```

2. 安装内核

```
make modules_install
make install
```

3. 重新启动系统

写用户态程序 test.c

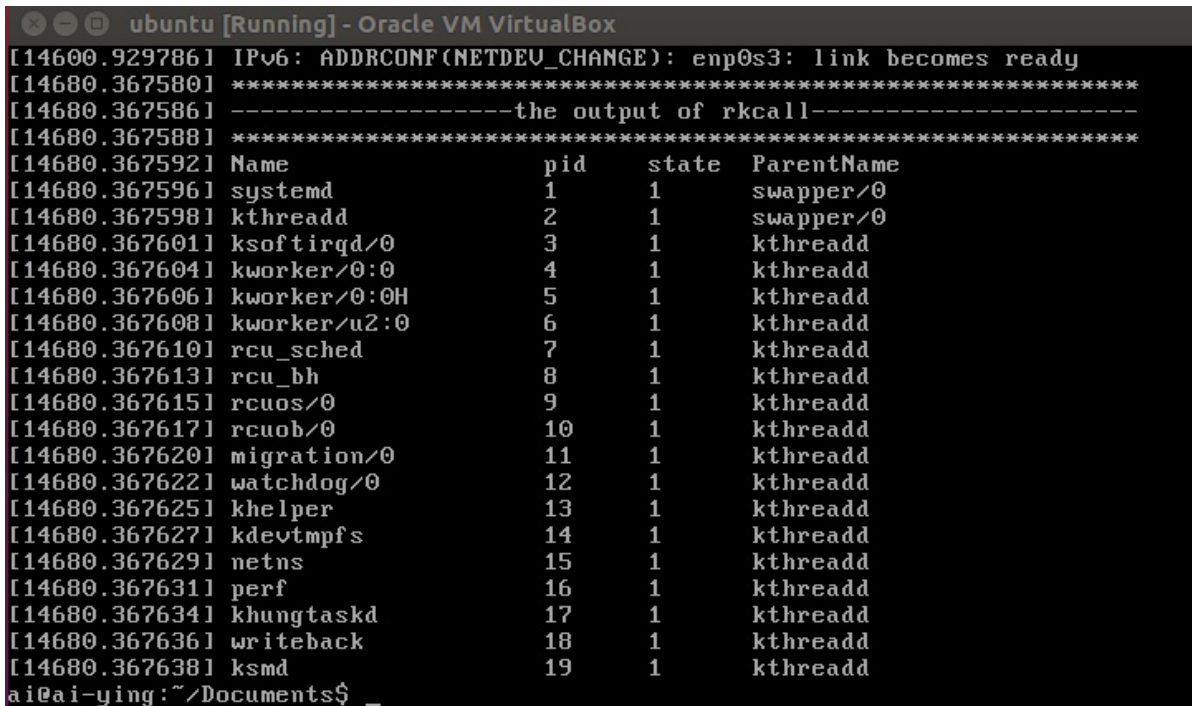
```
#include<linux/unistd.h>
#include<sys/syscall.h>

#define _NR_ai 666

int main()
{
    syscall(666);
    return 0;
}
```

运行 test.c。执行 dmesg 查看结果

结果截图：



```
ubuntu [Running] - Oracle VM VirtualBox
[14600.929786] IPv6: ADDRCONF(NETDEV_CHANGE): enp0s3: link becomes ready
[14680.367580] *****
[14680.367586] -----the output of rkcall-----
[14680.367588] *****
[14680.367592] Name                pid    state  ParentName
[14680.367596] systemd                1      1      swapper/0
[14680.367598] kthreadd                2      1      swapper/0
[14680.367601] ksoftirqd/0            3      1      kthreadd
[14680.367604] kworker/0:0             4      1      kthreadd
[14680.367606] kworker/0:0H            5      1      kthreadd
[14680.367608] kworker/u2:0            6      1      kthreadd
[14680.367610] rcu_sched               7      1      kthreadd
[14680.367613] rcu_bh                  8      1      kthreadd
[14680.367615] rcuos/0                 9      1      kthreadd
[14680.367617] rcuob/0                 10     1      kthreadd
[14680.367620] migration/0            11     1      kthreadd
[14680.367622] watchdog/0             12     1      kthreadd
[14680.367625] khelper                 13     1      kthreadd
[14680.367627] kdevtmpfs               14     1      kthreadd
[14680.367629] netns                   15     1      kthreadd
[14680.367631] perf                    16     1      kthreadd
[14680.367634] khungtaskd              17     1      kthreadd
[14680.367636] writeback               18     1      kthreadd
[14680.367638] ksmd                    19     1      kthreadd
ai@ai-ying:~/Documents$
```

图 4-2-5

分析：系统调用是由操作系统实现提供的所有系统调用所构成的集合即程序接口或应用编程接口(Application Programming Interface, API)。是应用程序同系统之间的接口。系统调用属于内核程序，同一台电脑上的不同程序都可以调用。

写系统调用时需要注意的几个方面，否则很容易编译多次扔调用不成功。

1.分配系统调用号的时候，注意尾部要对系统规定的系统调用数目进行加一，否则虽然在编译的时候不会报错，但最终调用不会成功。

2.修改系统调用表的时候也要注意，32位和64位修改的不是同一个文件，否则最终也调用不成功，修改64位的时候尽量不要写在文档尾部，编译的时候可能会报错。

实验五、内核模块

5-1 实际hello内核模块

实验目的

- 1、了解内核模块
- 2、熟悉内核模块的编写方法

实验内容

- (1)掌握内核模块基本编程技术
- (2)向内核中添加一个内核模块,编译模块
- (3)加载、卸载模块

实验指导

Linux 内核是具有微内核特点的宏内核。Linux 内核作为一个大程序在内核空间运行。太多的设备驱动和内核功能集成在内核中,内核过于庞大。Linux 内核引入内核模块机制。通过动态加载内核模块,使得在运行过程中扩展内核的功能。不需要的时候,卸载该内核模块。模块是 Linux 精心设计的一种机制,可以用来动态增加内核的功能,模块在内核空间运行。Linux 模块可以在内核启动过程中加载,这称为静态加载。也可以在内核运行的过程中随时加载,这称为动态加载。Linux 中的大多数设备驱动程序或文件系统都被编译成模块,因为它们数目繁多,体积庞大,不适合直接编译在内核中。而通过模块机制,在需要使用它们的时候,在临时加载,是最合适不过的。至少两个函数: `init_module()` 模块加载的时候调用, `cleanup_module()` 模块被卸载前调用。

- 1、编写.c 文件

```
/*hello.c*/

#include <linux/init.h>
#include <linux/kernel.h>
#include <linux/module.h>

//模块许可声明
MODULE_LICENSE("GPL");

//模块加载函数
static int __init hello_init(void)
{
    printk(KERN_ALERT "hello kernel!\n");
    return 0;
}

//模块卸载函数
static void __exit hello_exit(void)
{
    printk(KERN_ALERT "Goodbye!\n");
}

//模块注册
module_init(hello_init);
module_exit(hello_exit);
//可选
MODULE_AUTHOR("ai");
MODULE_DESCRIPTION("hello");
```

2、编写 Makefile 文件

```
obj-m :=hello.o
```

```
LINUX_KERNEL :=/lib/modules/$(shell uname -r)/build
```

```
all:
```

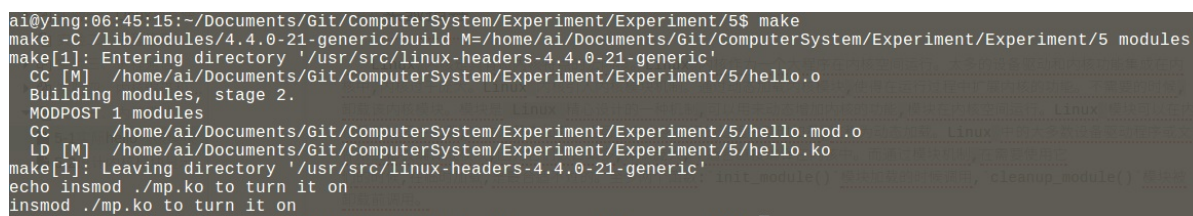
```
    make -C $(LINUX_KERNEL) M=$(shell pwd) modules
    echo insmod ./mp.ko to turn it on
```

```
clean:
```

```
    make -C $(LINUX_KERNEL) M=$(shell pwd) clean
```

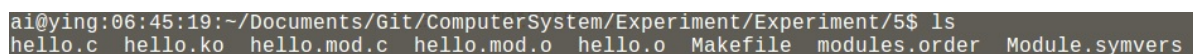
3、将上面两个文件放在同一目录下,使用 make 进行编译。

结果截图：



```
ai@ying:06:45:15:~/Documents/Git/ComputerSystem/Experiment/Experiment/5$ make
make -C /lib/modules/4.4.0-21-generic/build M=/home/ai/Documents/Git/ComputerSystem/Experiment/Experiment/5 modules
make[1]: Entering directory '/usr/src/linux-headers-4.4.0-21-generic'
CC [M] /home/ai/Documents/Git/ComputerSystem/Experiment/Experiment/5/hello.o
Building modules, stage 2.
MODPOST 1 modules
CC /home/ai/Documents/Git/ComputerSystem/Experiment/Experiment/5/hello.mod.o
LD [M] /home/ai/Documents/Git/ComputerSystem/Experiment/Experiment/5/hello.ko
make[1]: Leaving directory '/usr/src/linux-headers-4.4.0-21-generic'
echo insmod ./mp.ko to turn it on
insmod ./mp.ko to turn it on
```

图5-1-1



```
ai@ying:06:45:19:~/Documents/Git/ComputerSystem/Experiment/Experiment/5$ ls
hello.c hello.ko hello.mod.c hello.mod.o hello.o Makefile modules.order Module.symvers
```

图5-1-2

4、可以通过 insmod 和 rmmod 命令显式地(手工)将模块载入内核或从内核中将它卸载。

```
insmod hello.ko 加载这个模块
dmesg           查看内核日志
lsmod           查看内核所有模块
rmmod hello.ko  卸载这个模块
```

结果截图:

```
sudo insmod hello.ko
sudo dmesg
```

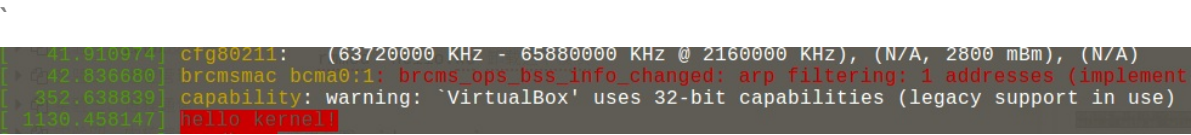


图 5-1-3

```
lsmod
```

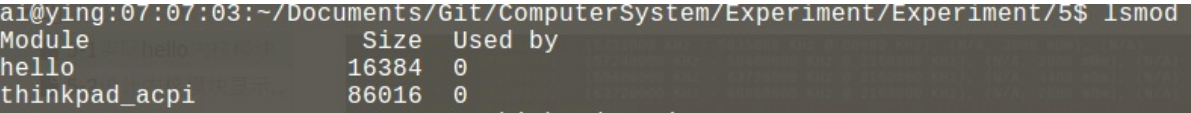


图 5-1-4

```
sudo rmmod hello.ko
sudo dmesg
```

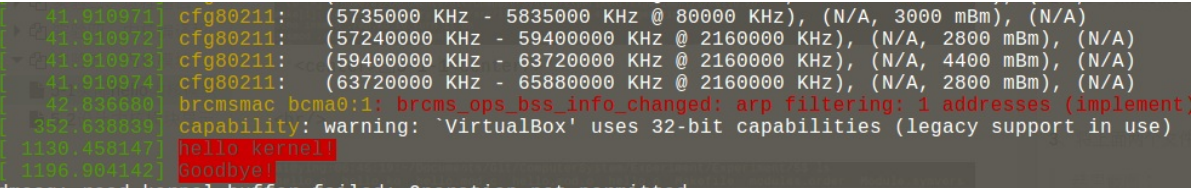


图 5-1-5

```
make clean
```

```
ai@ying:06:48:25:~/Documents/Git/ComputerSystem/Experiment/Experiment/5$ make clean
make -C /lib/modules/4.4.0-21-generic/build M=/home/ai/Documents/Git/ComputerSystem/Experiment/Experiment/5 clean
make[1]: Entering directory '/usr/src/linux-headers-4.4.0-21-generic'
  CLEAN   /home/ai/Documents/Git/ComputerSystem/Experiment/Experiment/5/.tmp_versions
  CLEAN   /home/ai/Documents/Git/ComputerSystem/Experiment/Experiment/5/Module.symvers
make[1]: Leaving directory '/usr/src/linux-headers-4.4.0-21-generic'
ai@ying:06:48:53:~/Documents/Git/ComputerSystem/Experiment/Experiment/5$ ls
hello.c  Makefile
```

图 5-1-3

5-2设计内核模块显示系统中所有的进程

实验目的

- 1、了解进程控制块
- 2、熟悉内核模块的实现机制

实验内容

- (1)掌握进程控制块的结构
- (2)编写内核模块,打印所有进程控制块的 PID 和可执行文件名信息。

实验指导