**第四章：linux进程管理**

**4.3 linux内核模块的编程入门**

内核模块是linux内核向外提供的一个插口，其全称是动态可加载内核模块，简称模块。模块是具有独立功能的程序，它可以被单独编译，但不能独立运行。它在运行时被连接到内核作为内核的一部分在内核空间运行，这与运行在用户空间的进程是不同的。模块通常由一组函数和数据结构组成，可用来实现一种文件系统、一个驱动程序或其他内核上层的功能。

**一、模块编程步骤：**

（1）编辑源程序，保存到某一目录下；

（2）编辑Makefile文件，注意保存到与源程序同一目录下；

（3）使用make命令编译模块程序；

（4）使用insmod命令插入模块，让其工作

（5）使用dmesg命令查看模块运行结果；

（6）使用rmmod命令从内核中移除模块

**二、模块编程举例：**

**第一步：编辑源程序：hello.c**，保存到某一目录下

#include<linux/module.h>

#include<linux/kernel.h>

#include<linux/init.h>

MODULE\_LICENSE("Dual BSD/GPL");

MODULE\_AUTHOR("MDAXIA");

static int \_\_init hello\_init(void)

{

printk(KERN\_ALERT "Hello world!\n");

return 0;

}

static void \_\_exit hello\_exit(void)

{

printk(KERN\_ALERT "Goodbye,cruel world!");

}

module\_init(hello\_init);

module\_exit(hello\_exit);

**第二步：编辑Makefile文件，保存到与hello.c同一目录下：**

ifeq ($(KERNELRELEASE),)

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

PWD := $(shell pwd)

modules:

$(MAKE) -C $(KDIR) M=$(PWD) modules /\*编译模块\*/

modules\_install:

$(MAKE) -C $(KDIR) M=$(PWD) modules\_install

clean:

rm -rf \*.o \*.ko .depend \*.mod.o \*.mod.c Module.\* modules.\*

.PHONY:modules modules\_install clean

else

obj-m :=hello.o /\*产生hello模块的目标文件\*/

endif

**第三步：编译模块：依次执行下面两个命令**

**#sudo make**

这时，在hello.c 所在目录就会有 hello.ko ，这就是我们需要的内核模块

**#sudo make clean**

清理编译垃圾。

**第四步：插入模块，让其工作**。注意必须是**root**权限

**#sudo insmod ./hello.ko**

**第五步：查看执行结果：**

用**dmesg命令**就可以看到 产生的内核信息：Hello world!

**注意**：如果没有输出" hello world"，可能是因为在终端模拟器下运行，这时会把内核消息输出到日志文件/var/log/kern.log中。可用cat命令查看。

**第六步：从内核中移除模块：**

**#sudo rmmod ./hello**  
再用dmesg命令 可以看到内核输出信息：Goodbye,cruel world!

**程序详细说明：**

**（1）源程序文件说明：**

**module.h：**包含了模块的结构定义及版本控制，任何模块程序的编写都要包含该文件。

k**ernel.h：**包含了常用的内核函数

**init.h：**包含了宏\_\_init和\_\_exit的定义，宏\_\_init告诉编译程序相关的函数和变量仅用于初始化，编译程序将标有宏\_\_init的所以代码存储到特殊的内存段中，初始化结束后就释放这段内存。

**Hello\_init()**：模块的初始化函数

**Hello\_exit()**：模块的退出和清理函数

**module\_init()和module\_exit()**：是模块编程中最基本的也是必须的两个函数，前者是模块加载函数，这个是安装模块时被系统自动调用的函数，通过module\_init宏来指定。后者是模块卸载函数，这是在卸载模块时被系统自动调用的函数，通过module\_exit来指定。

**（2）Makefile文件说明：必须放在与hello.c同一个目录下**

ifeq ($(KERNELRELEASE),)

判断KERNELRELEASE是否为空，该变量是内核源码的顶层Makefile中定义的一个变量，是描述内核版本的字符串。只有执行make命令的当前目录为内核源代码目录时，该变量才不为空字符。

KDIR := /lib/modules/$(shell uname -r)/build /\*linux内核源码的当前版本的绝对路径\*/

PWD := $(shell pwd) /\*模块所在的当前路径\*/

modules:

$(MAKE) -C $(KDIR) M=$(PWD) modules /\*编译模块\*/

/\*这句是Makefile的规则：这里的$(MAKE)就相当于make，-C选项的作用是指将当前工作目录转移到你所指定的位置。“M=”选项的作用是，当用户需要以某个内核为基础编译一个外部模块的话，需要在make modules 命令中加入“M=dir”，程序会自动到你所指定的**dir**目录中查找模块源码，将其编译，生成KO文件。\*/

modules\_install:

$(MAKE) -C $(KDIR) M=$(PWD) modules\_install /\*模块的安装\*/

clean:

rm -rf \*.o \*.ko .depend \*.mod.o \*.mod.c Module.\* modules.\*

.PHONY:modules modules\_install clean

/\*这句话是的作用是保证modules,modules\_install,clean这三个命令能正常完成。.PHONY 这是一个特殊目标名称\*/

else

obj-m :=hello.o /\*产生hello模块的目标文件\*/

endif

**（3）命令说明：**

**1．insmod命令**  
调用insmod程序把需要插入的模块以目标代码的形式插入到内核中。在插入的时候，insmod自动调用init\_module()函数运行。注意，只有超级用户才能使用这个命令，其命令格式为：  
# insmod  [path] modulename.ko  
**2. rmmod命令**  
调用rmmod程序将已经插入内核的模块从内核中移出，rmmod会自动运行cleanup\_module()函数，其命令格式为：  
#rmmod  [path] modulename.ko  
**3．lsmod命令**  
调用lsmod程序将显示当前系统中正在使用的模块信息。实际上这个程序的功能就是读取/proc文件系统中的文件/proc/modules中的信息，其命令格式为：  
#lsmod

**4.4 linux进程管理系统调用**

**一、 linux进程控制**

**1、创建进程**

函数原型：

#include <sys/types.h>

#include <unistd.h>

**pid\_t fork(void);**

参数:无。

功能：创建子进程。fork()用来创建一个子进程，被创建的子进程是父进程精确的副本。创建成功后，父子进程采用“写时复制”共享代码和数据，直到一方要写入时才对相应内容进行复制。

返回值：创建成功，父进程返回子进程的进程，子进程中返回0；创建失败，则不会产生子进程，函数的返回值为-1。

注意：如果创建成功，子进程将是父进程的精确副本，因此，子进程的下一条指令的地址跟父进程相同，都将从fork()调用返回的位置继续执行。但在执行时为了区分父子进程，操作系统给它们赋了不同的fork()返回值。

用户在使用Linux系统的时候，每次在终端下面输入一行命令，就由shell进程接收这个命令并创建一个新的进程，这个新的进程还可以通过fork()系统调用，继续创建自己的子进程。系统中的多个进程构成一棵进程树。实际上，在Linux系统启动时，最早产生的进程是idle进程，其pid号为0，该进程会创建一个内核线程，该线程进行一系列初始化动作后最终会执行/sbin/init文件，执行该文件的结果是运行模式从核心态切换到用户态，该线程演变成了用户进程init，pid为1。init进程是一个非常重要的进程，一切用户态进程都是它的后代进程。

* + 1. **更改进程运行的程序**

函数原型：

#include <unistd.h>

int execl( const char \*path, const char \*arg, ...);

int execlp( const char \*file, const char \*arg, ...);

int execle( const char \*path, const char \*arg, ..., char\* const envp[]);

int execv( const char \*path, char\* const argv[]);

int execve( const char \*path, char\* const argv[], char\* const envp[]);

int execvp( const char \*file, char\* const argv[]);

参数:

path是新的被执行程序的路径名；

file 是新的被执行程序的文件名；

arg 是若干个字符串，作为执行程序的参数，其中第一个字符串为可执行程序的程序名，最后一个字符串为ＮＵＬＬ。

argv 是字符串数组，作为执行程序的参数（可参看main函数的参数argv），同样它也必须以ＮＵＬＬ作为结束字符串

envp 是字符串数组，为执行文件的环境变量参数，同样也必须以ＮＵＬＬ作为结束标记。

功能： exec系列函数，用来将进程执行的程序更换成路径名所指定的新程序。在exec调用之前存在的用户级上下文的内容，在exec调用成功之后，将不能再被访问。

返回值：执行成功时无返回值，失败时返回-1。

* + 1. **终止进程**

函数原型：#include <stdlib.h>

**void exit(int status);**

参数：status是返回给父进程的状态值。

功能：终止调用进程。

返回值：exit()函数没有返回值，其实它根本不会返回。

* + 1. **等待一个进程终止**

函数原型：**#include <sys/types.h>**

**#include <sys/wait.h>**

**pid\_t wait(int \*status)；**

**pid\_t waitpid(pid\_t pid, int \*status, int options);**

参数：\*status 如果不为NULL，则保存结束的子进程的状态相关的信息；

pid ＞0,只等进程标识数为pid的子进程终止；

=0，等任一组标识数等于当前进程标识数的子进程终止；

-1，等任一子进程终止；

＜-1，等任一组标识数等于|pid|的子进程终止；

**options** 可以为0，也可以是下列常量或它们的组合；

WNOHANG，表示如果没有已经终止的子进程，调用者立即返回；

WUTRACED,跟进程跟踪调试相关。

功能：wait函数挂起当前进程，直到它的一个子进程终止，或者该进程收到一个终止自己的信号为止，如果调用时子进程已经处于僵尸状态，则当前进程无需挂起，子进程所占用的系统资源将被释放。waitpid类似于wait，但它可指定所等待的子进程

返回值：通常返回终止子进程的PID。如果使用了WNOHANG而且没有子进程终止，则返回0；如果出错则返回-1。

* + 1. **使进程进入睡眠状态：**

函数原型：#include <unistd.h>

**Unsigned int sleep(insigned int seconds);**

参数：seconds为睡眠的时间，单位为秒。

功能：使当前进程主动进入睡眠状态，直到过去指定的秒数seconds，或者它收到一个不能忽略的信号为止。

返回值：如果过了指定的秒数，则返回0；否则返回剩余的秒数。

* + 1. **获取进程的标识数、父进程的标识数**

函数原型：#include <sys/types.h>

#include <unistd.h>

**pid\_t get\_pid(void)**

**pid\_t get\_ppid(void)**

参数：无。

功能：get\_pid函数用来获取调用进程的进程标识数，get\_ppid则用来获取调用进程父进程的标识数。

返回值：调用进程的标识数，或调用进程父进程的标识数。

* + 1. **调整进程优先级**

函数原型：#include <unistd.h>

**int nice(int inc)**

参数：进程nice值的增量。

功能：将进程的nice值加上增量值inc。nice值越大，进程调度的优先权越低，只有超级用户root可以设置负值增量以提高优先权，普通用户只能设置正增量，即降低优先级。

返回值：成功返回0，否则返回-1。

* + 1. **system函数**

用户可以使用该函数来在自己的程序中调用系统提供的各种命令。

#include<stdlib.h>

int system(const char \*cmdstring);

参数cmdstring 是一个字符串指针，指向表示命令行的字符串。该函数的实现是通过调用fork、exec和waitpid函数来完成的，其中任意一个调用失败则system函数的调用失败，故返回值较复杂。

例：

#include<stdlib.h>

#include<stdio.h>

int main() {

printf("running ps with system.\n");

system("ps –ax");

printf("Done.\n");

exit(0);

}

* + 1. **Pthread线程库**

**必须包含头文件：<pthread.h>**

(1) int pthread\_create (pthread\_t \*thread,const pthread\_attr\_t \*attr, void \*(\*routines)(void \*), void \*arg);

功能：创建以函数routines为线程体，以arg为参数，具有attr线程属性#include<stdlib.h>

#include<stdio.h>

int main() {

printf("running ps with system.\n");

system("ps –ax");

printf("Done.\n");

exit(0);

}

#include<stdlib.h>

#include<stdio.h>

int main() {

printf("running ps with system.\n");

system("ps –ax");

printf("Done.\n");

exit(0);

}

的线程。

参数表：thread：返回参数，新线程的句柄；attr：新生成线程的属性，如果值为NULL，则具有默认的线程属性设置；routines：线程指定运行的参数，该参数必须具有void \*返回值；arg：该线程运行函数的参数。

返回值：如果成功创建该线程，则函数返回0，否则返回一个非0的错误码。

(2) int pthread\_join (pthread\_t thread, void \*\*status);

功能：等待一个线程结束，并将结束时的状态写入status。

参数：thread: 被等待的线程；status: 输出参数，线程结束时的状态。

返回值：0（成功）；否则，返回非0的错误码。

(3) int pthread\_mutex\_lock (pthread\_mutex\_t \*mutex);

功能：如果信号量mutex未加锁，则为其加锁，否则，该线程阻塞直到mutex解锁。

参数：mutex: 需要加锁的信号量。 返回值：0（成功）；否则，返回非0的错误码。

(4) int pthread\_mutex\_unlock (pthread\_ t \*mutex);

功能：为指定的信号量解锁。

参数：mutex: 需要加锁的信号量。 返回值：0（成功）；否则，返回非0的错误码。

* + 1. **进程控制系统调用使用示例**

**实例1：**

**/\* process\_state.c \*/**

#include <stdio.h>

#include <sys/types.h>

#include <unistd.h>

#include <stdlib.h>

int main()

{

int pid;

if(!(pid = fork())) /\* 子进程始执行\*/

{

int i,j,k;

printf("C: This is son, my pid is: %d, my father's pid is %d \n", getpid(), getppid());

printf("C: I'm so busy...\n");

for(i=0; i<10000; i++)

for(j=0; j<10000; j++)

for(k=0; k<50; k++);

printf("C: Now, Let me have a rest. \n");

sleep(15);

printf("C: I want to exit. \n");

exit(0);

}

else /\* 父进程执行\*/

{

printf("F: This is father, my pid is: %d\n", getpid());

sleep(50);

pid=wait(NULL);

printf("F: My son %d terminated.\n",pid);

sleep(10);

printf("F: Ok, I'm almost finished.\n");

}

}

1. 认真阅读上述程序，了解程序的基本功能，并仔细思考父进程创建子进程后将从哪里开始继续运行；子进程被创建后，首次得到CPU后将从哪里开始运行。
2. 打开一个终端窗口，编辑、编译并运行上述程序，观察程序运行的结果是否如你所料。思考，子进程首次得到CPU后从哪个位置开始执行，这到底是通过什么做到的。
3. 观察父子进程是顺序执行，还是并发执行。
4. 请再打开另一个终端窗口，用top观察系统中进程的情况，观察所有进程是按什么项目从大到小的顺序依次显示的。
5. 将top的进程显示顺序按PID从新排序，并在显示界面中只保留USER 、PID、S、COMMAND四个项目。
6. 在前一个终端窗口中再次运行process\_state程序，同时仔细观察top终端窗口显示的内容，了解该程序对应的父子进程在它们的生命周期中的状态经历了哪些变化。

**实例2：**

#include <stdio.h>

#include <sys/types.h>

#include <unistd.h>

int count = 1;

int main()

{

int child;

if(!(child = fork())) /\* 子进程将从此点开始运行\*/

{

int i;

printf("This is son, my pid is: %d, my father's pid is %d ", getpid(), getppid());

for(i=0; i<10; i++)

{

printf(" Son's count is: %d. \n", count++);

sleep(5);

}

}

else /\* 父进程执行\*/

{

printf("This is father, my pid is: %d, my father's pid is %d\n", getpid(), getppid());

printf(" Father's count is: %d. \n", count);

sleep(10);

printf(" Father's count is: %d. \n", count);

}

}

该程序运行时创建一个子进程，子进程显示自己和父进程的进程标识数后，进行循环，每轮循环将显示并修改count的值、并进入睡眠状态5秒钟。父进程显示自己和父进程的父进程标识数后，显示count的值，睡眠5秒钟后，在没修改count变量的情况下再一次显示它的值。

（1）在一个终端窗口中，编辑、编译并运行上述程序，观察父子进程的count变量是共享内存中的同一个变量；还是父子进程各有自己的空间，它们的count变量存放在自己不同的空间中，因此并不相互影响。

（2）在另一个终端窗口使用top命令观察系统进程的动态变化，并思考top窗口中显示的某些内容如PID，S，PR，NI等，是存放在父进程的空间中；还是子进程的空间中；还是分别在父子进程的空间中；还是均不在父子进程的空间中，而在操作系统的空间中。请思考，如果OS没提供显示进程信息的top，ps等命令，也没提供getpid()等函数，用户有没有可能直接获得自己某个进程的进程控制块内的信息（如进程标识数的值等）。

（3）将top的进程显示顺序按PID进行排序。

（4）选择top显示的项目内容并调整显示顺序，使它们从左到右依次显示PPID、PID、S、USER、%CPU、%MEN、COMMAND等内容；

（5）在前一个终端窗口中再次运行上述程序，并在运行top的终端窗口中仔细观察本次程序的进程情况，尤其是关注父子进程的状态变化、子进程的PPID和父子进程的终止情况，了解父进程是否能在子进程终止前先终止；父进程如果先终止了，那么子进程将托付给哪个进程成为它的儿子；子进程会不会因为父进程先终止而成为永远的僵尸进程。

**实例3：**

**C程序1：**

**/\* forkExecDemo.c \*/**

**#include <stdio.h>**

**#include <sys/types.h>**

**#include <unistd.h>**

**#include <stdlib.h>**

**#include <sys/wait.h>**

**main()**

**{**

**int pid;**

**if((pid=fork())==0)**

**{**

**execl("./newprog","newprog",0);**

**}**

**printf("F: This is father,my pid is %d.\n",getpid());**

**printf("F: I've a child, his pid is %d.\n",pid);**

**printf("F: Now, I'm waiting for his termination.\n");**

**pid=wait(NULL);**

**printf("F: My son %d terminated.\n",pid);**

**exit(0);**

**printf("F: Never print out.\n");/\* 语句1 \*/**

**}**

**C程序2：**

/\*newprog.c\*/

#include <stdio.h>

main()

{

printf("S: My pid is %d.\n",getpid());

printf("S: My father's pid is %d.\n",getppid());

}

（1）认真阅读上述程序，思考：子进程被创建后是运行跟父进程一样的程序，还是从父进程的程序转去运行新的程序了？子进程在执行execl()前后，其进程标识数有无变化，它使用的是同一个进程描述符（PCB），还是前后分别使用了不同的PCB？程序forkExecDemo.c中对应于注释/\* 语句1 \*/的那条语句会不会被执行?

（2）编辑并将上述程序分别保存到两个文件中，再分别编译后将可执行文件保存到forkExecDemo和newprog中；

（3）运行forkExecDemo程序，仔细观察，并再次理解进程和程序的区别和联系、顺序执行和并发执行的概念、进程控制块、CPU现场等内容。

**实例4：**用fork()创建一个子进程，由其调用execve（）启动shell命令ps查看系统当前进程的信息。

**#include<unistd.h>**

**#include<stdio.h>**

**#include<sys/types.h>**

**Main()**

**{**

**int pid;**

**char \*path=”/bin/ps”;**

**char \*argv[5]={“ps”,”-a”,”-x”,null};**

**printf(“Run ps with execv() by child process:\n”);**

**if ((pid=fork())<0)**

**{**

**Printf(“fork error!”)”**

**Exit(0);**

**}**

**Else if(pid==0)**

**{**

**If(exceve(path,argv[],o)<0)**

**{**

**Printf(exceve() error!”);**

**Exit(0);**

**}**

**Printf(“child is OK!\n”);**

**Exit(0);**

**}**

**Wait();**

**Printf(“father is OK !\n”);**

**Exit(0);**

**}**

**实例5：Pthread线程库的应用：**

例：创建一个线程，进行运算和显示输出信息。主程序也对相同的变量进行运算，输出相应信息。

#include<pthread.h>

#include<stdlib.h>

#include<unistd.h>

#include<stdio.h>

int myglobal;

pthread\_mutex\_t mymutex=PTHREAD\_MUTEX\_INITIALIZER; //定义静态互斥

void \*thread\_function(void \*arg) {

int i,j;

for(i=0;i<20;i++) {

pthread\_mutex\_lock(&mymutex); //加锁

j=myglobal;

j++;

printf(".");

fflush(stdout);

sleep(1);

myglobal=j;

pthread\_mutex\_unlock(&mymutex); //解锁

}

return NULL;

}

int main(void) {

pthread\_t mythread;

int i;

if(pthread\_create(&mythread, NULL, thread\_function, NULL)) {

printf("error creating thread.");

abort();

}

for(i=0;i<20;i++) {

pthread\_mutex\_lock(&mymutex); //加锁

myglobal++;

pthread\_mutex\_unlock(&mymutex); //解锁

printf(".");

fflush(stdout);

sleep(1);

}

if(pthread\_join(mythread, NULL)) {

printf("error joining thread.");

abort();

}

printf("\nmyglobal equals %d\n",myglobal);

exit(0);

}

**二、进程通信**

**（一）Linux管道通信机制**

管道是所有UNIX及linux都提供的一种进程间通信机制，它是进程之间的一个单向数据流，一个进程可向管道写入数据，另一个进程则可以从管道中读取数据，从而达到进程通信的目的。

**1.无名管道**

无名管道通过pipe()系统调用创建，它具有如下特点：

1. 它只能用于具有亲缘关系的进程（如父子进程或者兄弟进程）之间的通信。
2. 管道是半双工的，具有固定的读端和写端。虽然pipe()系统调用返回了两个文件描述符，但每个进程在使用一个文件描述符之前仍需先将另一个文件描述符关闭。如果需要双向的数据流，则必须通过两次pipe()建立起两个管道。
3. 管道可以看成是一种特殊的文件，对管道的读写与文件的读写一样使用普通的read、write等函数，但它不是普通的文件，也不属于任何文件系统，而只存在于内存中。

**2.pipe系统调用**

（1）函数原型

#include <unistd.h>

int pipe(int filedes[2]);

（2）参数

filedes参数是一个输出参数，它返回两个文件描述符，其中filedes[0]指向管道的读端，filedes[1]指向管道的写端。

（3）功能

pipe在内存缓冲区中创建一个管道，并将读写该管道的一对文件描述符保存在filedes所指的数组中，其中filedes[0]用于读管道，filedes[1]用于写管道。

（4）返回值

成功返回0；失败返回-1，并在error中存入错误码。

（5）错误代码

EMFILE：进程使用的文件描述符过多

ENFILE ：系统文件表已满

EFAULT ：非法参数filedes

**3.无名管道的阻塞型读写**

管道缓冲区有4096B的长度限制，因此，采用阻塞型读写方式时，当管道已经写满时，写进程必须等待，直到读进程取走信息为止。同样，读空的管道时，也可能会引起进程阻塞。

**当管道大小（管道缓冲区中待读的字节数）为p，而用户进程请求读n个字节时：**

1. 若不存在写进程：

① p=0，则返回0；

② 0<p<n，则读得p个字节，返回p，管道缓冲区中还剩0个字节；

③ p≥n，则读得n个字节，返回n，管道缓冲区中还剩p－n个字节；

1. 若存在写进程，且写进程没因写管道而阻塞时：

① p=0，读进程阻塞等待数据被写入管道；

② 0<p<n，则读得p个字节，返回p，管道缓冲区中还剩0个字节；

③ p≥n，则读得n个字节，返回n，管道缓冲区中还剩p－n个字节；

1. 若存在写进程，且写进程因写管道而阻塞时：

①0<p<n，则读管道，当管道缓冲区变空时，阻塞，等待数据被写入。最后，返回实际读得的字节数；

②p≥n，则读得n个字节，返回n，管道缓冲区中还剩p－n个字节。

**当管道缓冲区中有u个字节未用，而用户进程请求写入n个字节时：**

1. 若不存在读进程，则向写管道的进程将发SIGPIPE信号，并返回－EPIPE。
2. 若存在至少一个读进程：

① u＜n≤4096 则写进程等待，直到有n-u个字节被释放为止，写入n个字节，返回n；

② n＞4096 则写入n个字节（必要时等待）并返回n；

③ u≥n 写入n个字节，返回n。

**4.无名管道使用实例**

//pipeDemo.c

#include <errno.h>

#include <stdio.h>

#include <stdlib.h>

#include <unistd.h>

#define BUFNUM 60

int main(void)

{

int n;

int fd[2];

pid\_t pid;

char buf[BUFNUM];

if (pipe(fd) < 0)

{

fprintf(stderr,"Creat Pipe Error：%s \n",strerror(errno));

exit(EXIT\_FAILURE);

}

if ((pid = fork()) < 0)

{

fprintf(stderr,"Fork Error：%s \n",strerror(errno));

exit(EXIT\_FAILURE);

}

if (pid > 0) /\* parent \*/

{

close(fd[0]);

write(fd[1], "I'm your father.\n", 17);

write(fd[1], "I'm waiting for your termination.\n", 35);

wait(NULL);

}

else /\* child \*/

{

close(fd[1]);

n = read(fd[0], buf, BUFNUM);

printf("%d bytes read:%s",n,buf);

}

return 0;

}

上述程序父进程创建一个管道，然后再创建一个子进程，由于子进程是父进程的精确拷贝，因此子进程也复制了父进程的文件描述符表信息，从而可以访问到父进程创建的管道。接着父进程关闭管道的读端，并向管道写入一些信息；而子进程则关闭管道的写端，并从读端获得父进程写入的信息。

请思考：

（1）如果将pipe()调用放在fork()之后进行，那么上述父子进程是否还能进行管道通信？

（2）上述程序运行后，子进程从管道中读到的信息是17个字节，还是52个字节，还是60个字节？为什么？

（3）如果子进程执行read()时，第3个参数的值为5，那么程序运行的结果会怎样？管道中未取走的信息在读端未关闭以前会不会消失？

（4）父子进程的执行顺序是否会影响到程序运行的结果？

**5.有名管道**

管道应用的一个重大限制是它没有名字，因此，只能用于具有亲缘关系的进程间通信，在有名管道（named pipe或FIFO）提出后，该限制得到了克服。FIFO不同于管道之处在于它提供一个路径名与之关联，以FIFO的文件形式存在于文件系统中。这样，即使与FIFO的创建进程不存在亲缘关系的进程，只要可以访问该路径，就能够彼此通过FIFO相互通信（能够访问该路径的进程以及FIFO的创建进程之间），因此，通过FIFO不相关的进程也能交换数据。值得注意的是，FIFO严格遵循先进先出（first in first out），对管道及FIFO的读总是从开始处返回数据，对它们的写则把数据添加到末尾。它们不支持诸如lseek()等文件定位操作。

有名管道通过mkfifo创建：

#include <sys/types.h>

#include <sys/stat.h>

int mkfifo(const char \* pathname, mode\_t mode)

该函数的第一个参数是一个普通的路径名，也就是创建后FIFO的名字。第二个参数与打开普通文件的open()函数中的mode 参数相同。 有名管道创建成功，mkfifo()返回0；否则返回-1。如果mkfifo的第一个参数是一个已经存在的路径名时，错误代码中会返回EEXIST错误，所以一般典型的调用代码首先会检查是否返回该错误，如果确实返回该错误，那么只要调用打开FIFO的函数就可以了。

与普通文件类似，有名管道在使用之前必须先进行open操作，具体类似于文件的打开方式：

#include <sys/types.h>

#include <sys/stat.h>

#include <fcntl.h>

int open(const char \*pathname, int flags);

对有名管道的open操作必须遵循下列规则：（1）如果当前打开操作是为读而打开FIFO时，若已经有相应进程为写而打开该FIFO，则当前打开操作将成功返回；否则，可能阻塞直到有相应进程为写而打开该FIFO（当前打开操作设置了阻塞标志）；或者，成功返回（当前打开操作没有设置阻塞标志）。（2）如果当前打开操作是为写而打开FIFO时，如果已经有相应进程为读而打开该FIFO，则当前打开操作将成功返回；否则，可能阻塞直到有相应进程为读而打开该FIFO（当前打开操作设置了阻塞标志）；或者，返回ENXIO错误（当前打开操作没有设置阻塞标志）。

一旦打开操作成功，便可通过返回的文件描述符，利用read、write系统调用对管道进行读写操作，读写完成应使用close系统调用关闭有名管道。

（3）有名管道使用实例

/\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

\* namedPipeDemo.c \*

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/

#include <sys/types.h>

#include <sys/stat.h>

#include <errno.h>

#include <fcntl.h>

#include <string.h>

#include <stdio.h>

#define FIFO\_NAME "/tmp/myfifo"

main()

{

int fd;

char w\_buf[50];

int w\_num;

// 若fifo已存在，则直接使用，否则创建它

if((mkfifo(FIFO\_NAME,0777)<0)&&(errno!=EEXIST))

{

printf("cannot create fifo...\n");

exit(1);

}

//以阻塞型只写方式打开fifo

fd=open(FIFO\_NAME,O\_WRONLY,0);

if(fd==-1)

if(errno==ENXIO)

{

printf("cannot open fifo for read...\n");

exit(1);

}

// 通过键盘输入字符串，再将其写入fifo，直到输入"exit"为止

while(1)

{

printf("please input something:");

scanf("%s",w\_buf);

w\_num=write(fd,w\_buf,strlen(w\_buf));

printf("real write num is %d\n",w\_num);

if(strcmp(w\_buf,"exit")==0) break;

}

}

/\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

\* namedPipeDemo2.c \*

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/

#include <sys/types.h>

#include <sys/stat.h>

#include <errno.h>

#include <fcntl.h>

#define FIFO\_NAME "/tmp/myfifo"

main()

{

char r\_buf[50];

int fd;

int r\_num;

// 若fifo已存在，则直接使用，否则创建它

if((mkfifo(FIFO\_NAME,0777)<0)&&(errno!=EEXIST))

{

printf("cannot create fifo...\n");

exit(1);

}

//以阻塞型只读方式打开fifo

fd=open(FIFO\_NAME,O\_RDONLY,0);

if(fd==-1)

{

printf("open %s for read error\n");

exit(1);

}

// 通过键盘输入字符串，再将其写入fifo，直到输入"exit"为止

while(1)

{

memset(r\_buf,0,sizeof(r\_buf));

r\_num=read(fd,r\_buf,50);

if(r\_num==-1)

if(errno==EAGAIN)

printf("no data avlaible\n");

printf(" %d bytes read:%s\n",r\_num,r\_buf);

if(strcmp(r\_buf,"exit")==0) break;

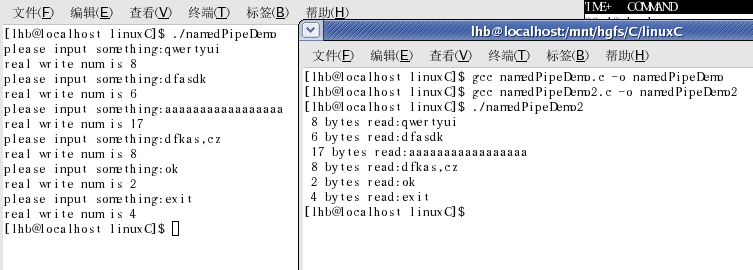
sleep(1);

}

unlink(FIFO\_NAME);//删除fifo

}

程序编译和运行结果：



**（二）Linux消息队列通信机制**

Linux系统中，若干个进程可以共享一个消息队列，系统允许其中的一个或多个进程向消息队列写入消息，同时也允许一个或多个进程从消息队列中读取消息，从而完成进程之间的信息交换，这种通信机制被称作消息队列通信机制。消息队列通信机制是客户/服务器模型中常用的进程通信方式：客户向服务器发送请求信息，服务器读取消息并执行相应的请求。消息可以是命令，也可以是数据。

**1.数据结构**

**（1）消息缓冲区struct msgbuf**

消息缓冲区是用来存放消息内容的结构体，而且这个结构体的第一个成员必须是一个大于0的长整数，表示对应消息的类型；不过，系统对结构体中其余成员的类型不做任何限制。include/linux/msg.h中给出的消息缓冲格式如下：

struct msgbuf{ /\* 消息定义的参照格式 \*/

long mtype; /\* 消息类型（大于0的长整数） \*/

char mtext[1]; /\*消息正文\*/

};

应用程序员可以重新定义消息缓冲区结构体，其中，成员(mtext)不仅能定义为长度为1的字符数组，也可以定义成长度大于1的字符数组，或定义成其他的数据类型，Linux也允许消息正文的长度为0，即结构体中没有mtext域。

虽然，Linux没限定mtext的类型，但却限定了消息的长度，一个消息的最大长度由宏MSGMAX决定，根据版本的不同，其取值可能为8192或其他值。

**（2）消息结构struct msg**

消息队列中的每个消息节点中不仅包含了消息内容，还包含了一些其他信息，消息节点由消息结构来描述。include/linux/msg.h中给出的消息结构格式如下：

struct msg {

struct msg \*msg\_next; /\*消息队列链接指针，指向队列中的下一条消息 \*/

long msg\_type; /\*消息类型，同struct msgbuf中的mtype\*/

char \*msg\_spot; /\* 消息正文的地址,指向msgbuf的消息正文 \*/

time\_t msg\_stime; /\* 消息发送的时间 \*/

short msg\_ts; /\* 消息正文的大小 \*/

};

**（3）IPC对象访问权限struct ipc\_perm**

struct ipc\_perm{

key\_t key; /\* IPC对象键值 \*/

ushort uid; /\* owner euid and egid \*/

ushort gid;

ushort cuid; /\* creator euid and egid \*/

ushort cgid;

ushort mode; /\* 访问权限 \*/

ushort seq; /\* slot usage sequence number，即IPC对象使用频率信息 \*/

};

其中：

key 是IPC对象（例如消息队列，共享存储器等）的键值，每个IPC对象(活动在内核级别的一种[进程间通信](http://baike.baidu.com/view/1492468.htm" \t "/home/william/文档\\x/_blank)的工具)都关联着一个唯一的长整型的键值，不同的进程通过相同的键值可访问到同一个IPC对象。用户进程在创建IPC对象时可以指定key为某个大于0的整数，此时，需要用户自己保证该key值不与系统中存在的其他IPC键值相冲突。更常用的方式是通过函数调用ftok（pathname，proj\_jd）请求系统为用户进程生成一个键值，其中的pathname是一个实际存在的文件的路径名，而且用户进程具有对该文件的访问权限，proj\_jd是一个整数，但ftok只会用到其低8位的值（该值不能为0），只要路径名访问到的是同一个文件，而且proj\_jd的低8位的值相同，则ftok()调用便将产生相同的键值；如果使用不同的文件路径名和proj\_jd，虽然系统不能保证、但通常生成的键值是不同的。

mode 中给出了该IPC对象的访问权限，由9个二进制位表示所有者、同组用户、其他组用户的访问权限。它可以是下列权限的组合：

**访问权限 八进制整数**

拥有者可读 0400

拥有者可写 0200

同组用户可读 0040

同组用户可写 0020

其他用户可读 0004

其他用户可写 0002

由于系统规定任何IPC结构都不存在可执行权限，因此一个IPC对象的权限最大值为**0666**（八进制）。

**（4）消息队列结构体struct msqid\_ds**

系统中每个消息队列由一个struct msqid\_ds类型的变量来描述，struct msqid\_ds的格式如下：

struct msqid\_ds {

struct ipc\_perm msg\_perm; /\* 消息队列访问权限\*/

struct msg \*msg\_first; /\* 队列上第一条消息，即链表头\*/

struct msg \*msg\_last; /\* 队列中的最后一条消息，即链表尾 \*/

time\_t msg\_stime; /\* 发送给队列的最后一条消息的时间 \*/

time\_t msg\_rtime; /\* 从消息队列接收到最后一条消息的时间 \*/

time\_t msg\_ctime; /\* 最后修改队列的时间\*/

…

ushort msg\_cbytes; /\*队列上所有消息总的字节数 \*/

ushort msg\_qnum; /\*当前队列上消息的个数 \*/

ushort msg\_qbytes; /\* 队列允许的最大的字节数 \*/

ushort msg\_lspid; /\* 发送最后一条消息的进程的pid \*/

ushort msg\_lrpid; /\* 接收最后一条消息的进程的pid \*/

};

Linux还通过宏MSGMNB限定了一个消息队列的最大长度（队列中所有消息总的字节数）。

**2.消息队列相关的系统调用**

Linux提供了一组消息队列相关的系统调用来方便用户进行消息通信。

**（1）o系统调用**

①函数原型：

#include <sys/types.h>

#include <sys/ipc.h>

#include <sys/msg.h>

int msgget(key\_t key, int msgflg)；

②参数:

key key为0(IPC\_PRIVATE)，则创建一个新的消息队列；否则，key为一个大于0的长整数，它对应于消息队列的键值，通常是通过ftok()函数生成的。

msgflg 对消息队列的访问权限和控制命令的组合。其中访问权限见“IPC对象访问权限struct ipc\_perm”部分的说明。而控制命令IPC\_CREAT表示，如果key对应的消息队列不存在，则创建它；而IPC\_EXCL必须与IPC\_CREATT一起使用，它表示：如果key对应的消息队列不存在，则创建一个新的队列，否则返回-1。

③功能： 如果IPC\_CREAT单独使用，semget()为一个新创建的消息队列返回标识数，或者返回具有相同键值的已存在消息队列标识数。如果IPC\_EXCL与IPC\_CREAT一起使用，要么创建一个新的队列并返回它的标识数，如果队列已存在，则返回-1。

④返回值： 成功，返回消息队列的标识数；出错，返回-1，同时将错误代码存放在error中。对于新创建的消息队列，其msqid\_ds结构成员变量的初值设置如下：

msg\_qnum、msg\_lspid、msg\_lrpid设置为0；

msg\_stime、msg\_rtime设置为0；

msg\_ctime设置为当前时间；

msg\_qbytes设成系统的限制值，即宏MSGMNB；

msgflg的读写权限写入msg\_perm.mode中；

msg\_perm结构的uid和cuid成员被设置成当前进程的有效用ID，id和cuid成员被设置成当前进程的有效组ID。

⑤错误代码：EACCES：指定的消息队列已存在，但调用进程没有权限访问它

EEXIST：key指定的消息队列已存在，而msgflg中同时指定IPC\_CREAT和IPC\_EXCL标志

ENOENT：key指定的消息队列不存在同时msgflg中没有指定IPC\_CREAT标志

ENOMEM：需要建立消息队列，但内存不足

ENOSPC：需要建立消息队列，但已达到系统的限制

**（2）msgsnd系统调用**

①函数原型：

#include <sys/types.h>

#include <sys/ipc.h>

#include <sys/msg.h>

int msgsnd(int msqid, struct msgbuf \*msgp, size\_t msgsz, int msgflg);

②参数:

msqid 消息队列的标识数

msgp 存放欲发送消息内容的消息缓冲区指针

msgsz 消息正文(而非整个消息结构)的长度

msgflg： 0 ——消息队列满时，msgsnd将会阻塞

IPC\_NOWAIT ——消息队列满时，msgsnd立即返回-1

MSG\_NOERROR——消息正文长度超过msgsz字节时，不报错，而是直接截去其中多余的部分，并只将前面的msgsz字节发送出去

③功能： 在标识数为msqid的消息队列中添加一个消息，即向标识数为msqid的消息队列发送一个消息。

④返回值： 消息发送成功，返回0；否则返回-1，同时error中存有错误代码

⑤错误代码：EAGAIN ——参数msgflg设为IPC\_NOWAIT，而消息队列已满

EACCESS——无权限写入消息队列

EFAULT ——参数msgp指向的地址无法访问

EIDRM ——标识符为msqid的消息队列已被删除

EINTR ——队列已满而处于阻塞的情况下，被信号唤醒

EINVAL ——无效的参数msqid、或消息类型type小于等于0、或msgsz为负数或超过系统限制值MSGMAX

ENOMEM ——系统无足够内存空间存放msgbuf消息的副本

**（3）msgrcv系统调用**

①函数原型：

#include <sys/types.h>

#include <sys/ipc.h>

#include <sys/msg.h>

ssize\_t msgrcv(int msqid, struct msgbuf \*msgp, size\_t msgsz, long msgtyp, int msgflg);

②参数:

msqid 消息队列的标识数

msgp 存放欲接收消息内容的消息缓冲区指针

msgsz 消息正文(而非整个消息结构)的长度

msg-typ 0 ——接收消息队列中的第一个消息

>0 ——接收第一个类型为msgtyp的消息

<0 ——接收第一个类型小于等于msgtyp的绝对值的消息

msgflg 0 ——没有可以接收的消息时，msgrcv阻塞

IPC\_NOWAIT ——没有可以接收的消息时，立即返回-1

MSG\_EXCEPT ——返回第一个类型不为msgtyp的消息

MSG\_NOERROR——消息正文长度超过msgsz字节时，将直接截去其中多余的部分

③功能： 如果传递给参数msgflg的值为IPC\_NOWAIT，并且没有可取的消息，那么给调用进程返回ENOMSG错误码，否则，调用进程阻塞，直到一条满足要求的消息到达消息队列。如果进程正在等待消息，而相应的消息队列被删除，则返回EIDRM。如果当进程正在等待消息时，捕获到了一个信号，则返回EINTR

④返回值： 接收成功，返回实际接收到的消息正文的字节数；否则返回-1，同时error中存有错误代码

⑤错误代码：E2BIG ——消息长度超过msgsz，且MSG\_NOERROR标志没被使用

EACCESS——无权限读取消息队列

EFAULT ——参数msgp指向的地址无法访问

EIDRM ——标识符为msqid的消息队列已被删除

EINTR ——等待消息的情况下，被信号唤醒

EINVAL ——无效的参数msqid、或msgsz为负数

ENOMSG ——参数msgflg设为IPC\_NOWAIT，但无满足要求的消息可接收

**（4）msgctl系统调用**

①函数原型：

#include <sys/types.h>

#include <sys/ipc.h>

#include <sys/msg.h>

int msgctl(int msqid, int cmd, struct msqid\_ds \*buf);

②参数:

msqid 消息队列的标识数

cmd IPC\_STAT ——将对应消息队列结构体的值复制到一份到buf所

指的结构体中，调用者必须有读消息队列的权限

IPC\_SET —— 将buf所指结构体中的部分信息:

msg\_perm.uid,msg\_perm.gid,mst\_perm.mode,msg\_qbytes写到消息队列结构体中，并且更新消息队列结构体msg\_ctime成员的值。调用者必须有相应的权限。

IPC\_RMD —— 删除消息队列，并唤醒该消息队列上等待读或等待写的进程。调用者必须有相应的权限。

③功能： 获取或设置消息队列的属性信息，或者删除消息队列

④返回值： 成功，返回0；否则返回-1，同时error中存有错误代码

⑤错误代码：EACCESS—— cmd为IPC\_STAT，但调用进程无读消息队列的权限

EFAULT —— 参数cmd为IPC\_STAT或IPC\_SET ，但buf指向的地址无法访问

EIDRM ——标识符为msqid的消息队列已被删除

EINTR ——等待消息的情况下，被信号唤醒

EINVAL ——cmd或msqid为无效的参数

EPERM —— 参数cmd为IPC\_RMD或IPC\_SET ，但调用进程无足够的权限

**3.实例**

//sender.c

#include <stdio.h>

#include <string.h>

#include <stdlib.h>

#include <errno.h>

#include <sys/types.h>

#include <sys/ipc.h>

#include <sys/msg.h>

#include <sys/stat.h>

#define MSG\_FILE "sender.c"

#define BUFFER 255

#define PERM S\_IRUSR|S\_IWUSR

struct msgbuf

{

long mtype;

char mtext[BUFFER+1];

};

char \*message[3]= {"I'm Sender,there are some messages for you.",\

"This is 1st message.",\

"This is 2nd message."

};

int main()

{

struct msgbuf msg;

key\_t key;

int msgid;

int i;

if((key=ftok(MSG\_FILE,66))==-1)

{

fprintf(stderr,"Creat Key Error：%s \n",strerror(errno));

exit(EXIT\_FAILURE);

}

if((msgid=msgget(key,PERM|IPC\_CREAT))==-1)

{

fprintf(stderr,"Creat MessageQueue Error：%s \n",strerror(errno));

exit(EXIT\_FAILURE);

}

msg.mtype=1;

for(i=0; i<3; i++)

{

strncpy(msg.mtext,message[i],BUFFER);

msgsnd(msgid,&msg,sizeof(struct msgbuf),0);

}

memset(&msg,'\0',sizeof(struct msgbuf));

msgrcv(msgid,&msg,sizeof(struct msgbuf),2,0);

printf("Sender receive: %s\n",msg.mtext);

if(msgctl(msgid, IPC\_RMID, 0) == -1)

{

fprintf(stderr, "Remove MessageQueue Error%s \n",strerror(errno));

exit(EXIT\_FAILURE);

}

exit(EXIT\_SUCCESS);

}

//receiver.c

#include <stdio.h>

#include <string.h>

#include <stdlib.h>

#include <errno.h>

#include <unistd.h>

#include <sys/types.h>

#include <sys/ipc.h>

#include <sys/stat.h>

#include <sys/msg.h>

#define MSG\_FILE "sender.c"

#define BUFFER 255

#define PERM S\_IRUSR|S\_IWUSR

struct msgbuf

{

long mtype;

char mtext[BUFFER+1];

};

int main()

{

struct msgbuf msg;

key\_t key;

int msgid;

int i;

char \*myask="I'm receiver, 3 messages received from you.";

if((key=ftok(MSG\_FILE,66))==-1)

{

fprintf(stderr,"Creat Key Error：%s \n",strerror(errno));

exit(EXIT\_FAILURE);

}

if((msgid=msgget(key,PERM|IPC\_CREAT))==-1)

{

fprintf(stderr,"Creat MessageQueue Error：%s \n",strerror(errno));

exit(EXIT\_FAILURE);

}

for(i=0; i<3; i++)

{

msgrcv(msgid,&msg,sizeof(struct msgbuf),1,0);

printf("Receiver receive: %s\n",msg.mtext);

}

msg.mtype=2;

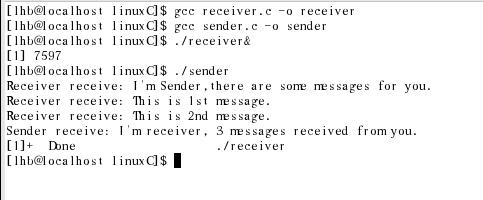
strncpy(msg.mtext,myask,BUFFER);

msgsnd(msgid,&msg,sizeof(struct msgbuf),0);

exit(EXIT\_SUCCESS);

}

**程序的运行结果如下：**



上述程序运行时，第三行命令名后的添加的‘&’，可以使命令“./receiver”在后台运行，shell会立即显示命令提示符，等待用户输入下一条命令，再输入“./sender”便可让上述两个程序同时运行。请思考：

1. 如果不使用后台进程，那应如何运行上述程序，达到进程通信的目的？
2. 发送给sender的消息、和发送给receiver的消息是存放在相同的消息队列中，还是存放在各自的消息队列中？
3. Sender、receiver获取/创建消息队列时使用的键值是否必须相同，如果键值不同会发生什么情况？
4. 程序中函数调用msgctl(msgid, IPC\_RMID, 0)的目的是什么？如果没有执行该函数调用，那程序多次运行会不会创建多个消息队列？

**（三）linux共享内存通信**

共享内存是Linux支持的三种进程间通信机制（IPC）中的一种。它实际上是一段特殊的内存区域。这一段内存区域可以被两个或者两个以上的进程映射到自身的地址空间中。一个进程写入共享内存中的信息，可以被其它使用这些共享内存的进程，通过一个简单点的内存读操作（memory read operation）读出，从而实现了进程间的通信。共享一个或多个进程通过同时出现在它们的虚拟地址空间的内存进程通信。这块虚拟内存的页面在每一个共享进程的页表中都有页表项目引用。但是不需要在所有进程的虚拟内存都有相同的地址。

1. **共享内存的系统调用**

任何Linux进程创建时，都有很大的虚拟地址空间，这块虚拟地址空间只有一部分放着代码、数据、堆和堆栈，剩余的那些部分在初始化时是空闲的。一块共享内存一旦被连接（attach），即会被映射入空闲的虚拟地址空间。随后，进程即可像对待普通的内存区域那样读、写共享内存。共享内存一共有四个系统调用，它们分别是：

* shmget（）创建一块共享内存。
* shmat（）将一块已存在共享内存映射（map）到一个进程的地址空间。
* shmdt（）取消一个进程的地址空间中一块共享内存块的映射（unmap）。
* shmctl（）是管理共享内存的函数，用于执行对共享内存的各种控制命令。

1. shmget分配一块共享内存

* 函数声明：

#include<sys/ipc.h>

#include<sys/shm.h>

int shmget(key\_t, int size, int shmflg);

* 输入参数：

key:标识共享内存的键值，在调用前应赋初值

Size:所需的共享内存的最小尺寸（以字节为单位）

shmflg:将分配的共享内存属性标志

* 返回值：

若成功，则返回共享内存的标识符；否则返回-1，错误原因存在于errno中。

errno=EINVAL：参数size小于SHMMIN或者大于SHMMAX。

EEXIST：欲建立 key所指的共享内存，但已经存在。

EIDRM：参数key所指的共享内存已经删除。

ENOSPC：已经超过系统允许建立的共享内存的最大值（SHMALL）。

ENOENT：参数key所指的共享内存不存在，参数shmflg也未设IPC\_CREAT位。

EACCESS：没有权限。

ENOMEM：核心内存不足。

* 说明：

参数key的取值可以是一块已经存在的共享内存的键值、0、或IPC\_PRIVATE。如果key的取值为IPC\_PRIVATE，则函数shmget()将创建一块新的共享内存；如果key的取值为0，而参数shmflg中设置了IPC\_CREATE这标志，则同样将创建一块新的共享内存。在IPC的通信模式下，不管是使用消息队列或者是共享内存，甚至是信号量，每个IPC的对象都有唯一的名字，它被称做“键”（key）。

参数size是要建立的共享内存的长度。所有的内存分配都是以页为单位的。

参数shmflg主要和一些标志有关。其中有效的标志包括IPC\_CREAT和IPC\_EXCL，它们的功能与open(2)的O\_CREAT和O\_EXCL相当。

在Linux内核中，每一个新创建的共享内存都由一个shmid\_ds的数据结构表示。如果函数shmget( )成功创建了一块共享内存，则返回一个可以用于引用该共享内存的shmid\_ds数据结构的标识符。

每一个IPC对象，系统共用一个struct ipc\_perm的数据结构来存放权限信息，以确定一个ipc操作是否可以访问该IPC对象。

1. shmat连接（attach）一块共享内存

* 函数声明：

#include<sys/ipc.h>

#include<sys/shm.h>

void \*shmat(int shmid, const void \*shmaddr, int shmgflg);

* 输入参数：

shmid:欲连接（attach）的共享内存的标识符。

shmaddr: 欲连接（attach）的地址。

shmflg:一个标识符。

* 返回值：

若成功则返回已连接好的地址，否则返回-1，错误原因存在于errno。

errno=EINVAL：参数shmid无效；参数shmaddr并非页对齐（page aligned），而参数shmflg中并未设置SHM\_RND这个标志位。

ENOMEN：分配标识符或页表时，系统内存不足。

EACCESS：调用进程没有权限以指定的方式连接该共享内存。

* 说明：

函数shmat（）将以参数shmid为标识符的共享内存连接（attach）到调用进程的数据段，连接的地址由shmaddr指定。

参数shmflg除了可以设置SHM\_RND标志位外，还可以设置SHM\_RDONLY标志位。

1. shmdt断开（detach）一块共享内存的连接

* 函数声明：

#include<sys/ipc.h>

#include<sys/shm.h>

int shmdt（const void \*shmaddr）；

* 输入参数：

shmaddr：欲断开连接（detach）的共享内存的虚拟地址。

* 返回值：

若成功则返回已连接好的地址，否则返回-1，错误原因存在于errno中。

errno=EINVAL：参数shmaddr无效或参数shmaddr地址并非共享内存地址。

* 说明：

函数shmdt从调用进程的数据段，断开地址在shmaddr的共享内存。要分离的共享内存必须是当前连接到调用进程地址空间的共享内存质疑，也就是说shmaddr必须等于连接某个共享内存时调用shmat时的返回值。

1. shmctl对一块共享内存的控制操作

* 函数声明：

#include<sys/ipc.h>

#include<sys/shm.h>

int shmctl (int shmid, int cmd, struct shmid\_ds \*buff);

* 输入参数：

shmid：为欲处理的共享内存的标识符。

cmd：为欲进行的操作。

buf：缓存。

* 返回值：

若成功则返回共享内存的标识符，否则返回-1，错误原因存在于errno中。

errno= EINVAL：参数shmid是个无效的标识符或cmd为无效命令。

EFAULT：参数cmd为IPC\_SET或 IPC\_STAT，但是参数buf却

指向无效的地址。

EIDRM：参数shmid所指的共享内存已经删除。

EPERM：参数cmd为IPC\_SET或IPC\_RMID，但是调用进程并

不是创建者，所有者或超级用户，并且调用进程没有授予这些组

或域的权限。

EACCESS：参数cmd为IPC\_STAT，但是没有权限读写该共享内存。

* 说明

该系统调用允许用户得到一块共享内存的相关信息，设置一块共享内存的所有者，组以及读写权限，或者销毁一块共享内存。以shmid为标识符的共享内存的相关信息将被放在一个shmid\_ds的数据结构体中返回。

1. **使用实例**

我们要求建立一个利用共享内存来传递信息的程序，一个writer，一个reader，writer从用户处获得输入，然后将其写入共享内存，reader从共享内存获取信息，然后在屏幕上输出。

对于利用共享内存实现进程之间传递信息的同步问题，我们考虑用信号量（semaphore）来解决。以下程序中，我们使用两个信号量，一个用户读（SN\_READ），一个用于写（SN\_WRITE）。SN\_READ初始化为1， SN\_WRITE初始化为0。 详细源代码见reader\_writer1.c。

reader\_writer1.c

/\*\*\* reader\_writer1.c一个利用共享内存实现进程通信的程序\*\*\*/

#include<stdio.h>

#include<sys/types.h>

#include<sys/ipc.h>

#include<sys/sem.h>

#include<sys/shm.h>

#include<stdlib.h>

#include<errno.h>

#include<string.h>

#include<signal.h>

/\*The union for semctl may or may not be defined for us. This code, defined in linux’s semctl( ) manpage, is the proper way to attain it if necessary\*/

#if defined (\_GNU\_LIBRARY\_) && !defined (\_SEM\_SEMUN\_UNDEFINED)

/\*union semun is defined by including <sys/sem.h>\*/

#else

/\*according to X/OPEN we have to define it ourselves\*/

union semum{

int val; /\*value for SETVAL\* /

struct semid\_ds \*buff; /\*buffer for IPC\_STAT, IPC\_SET\*/

unsigned short int \*array; /\*array for GETALL, SETALL\*/

struct seminfo \*\_buf; /\*buffer for IPC\_INFO\*/

};

#endif

#define SHMDATASIZE 1000

#define BUFFERSIZE (SHMDATASIZE-sizeof(int))

#define SN\_READ 0

#define SN\_WRITE 1

int Semid=0; /\*用于最后删除这个信号量\*/

void reader(void);

void writer(int shmid);

void delete(void);

void sigdelete(int signum);

void locksem(int semid, int semnum);

void unlocksem(int semid, int semnum);

void waitzero(int semid, int semnum);

void write(int shmid, int semid, char \*buffer);

void mysemget(key\_t key, int nsems, int semflg);

int mysemctl(int semid, int semnum, int cmd, union semun arg);

int mysemop(int semid, struct sembuf \*sops, unsigned nsops);

int myshmget(key\_t key, int size, int shmflg);

void \*myshmat(int shmid, const void \*shmaddr, int shmflg);

int myshmctl( int shmid, int cmd, struct shmid\_ds \*buf);

int main(int argc, char \*argv[ ]){

/\*没有其它的参数，则为reader\*/

If (argc<2){

reader();

}else{

writer(atoi(argv[1]));

}

return 0;

}

void reader(void){

union semun sunion;

int semid, shmid;

void \*shmdata;

char \*buffer;

/\*首先：我们要创建信号量\*/

semid=mysemget(IPC\_PRIVATE, 2, SHM\_RISHM\_W);

Semid=semid;

/\*在进程离开时，删除信号量\*/

Atexit=(&delete);

Signal(SIGINT, &sigdelete);

/\*信号量SN\_READ初始化为1（锁定），SN\_WRITE初始化为0（未锁定）\*/

sunion.val=1;

mysemctl(semid,SN\_READ,SETVAL,sunion);

sunion.val=0;

mysemctl(semid, SN\_WRITE,SETVAL,sunion);

/\*现在创建一块共享内存\*/

shmid=myshmget(IPC\_PRIVATE, SHMDATASIZE, IPC\_CREAT|SHM\_R|SHM\_W);

/\*将该共享内存映射到进程的虚拟空间\*/

shmdata=shmat(shmid,0,0);

/\*将该共享内存标志为已销毁的，这样在使用完毕后，将被自动销毁\*/

Shmctl(shmid, IPC\_RMID, NULL);

/\*将信号量的标识符写入共享内存，以通知其它的进程\*/

\*(int\*)shmdata=semid;

buffer=shmdata+sizeof(int);

printf(“\n reader begin to run, and the id of the share memory is %d \*\*\n”,shmid);

/\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

reader的主循环

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/

While (1){

Printf( “\n wait for the writer’s output information…”);

Fflush(stdout);

Locksem(semid, SN\_WRITE);

Printf(“finish\n”);

Printf(“received information:%s\n”,buffer);

Unlocksem(semid,SN\_READ);

}

}

void writer(int shmid){

int semid;

void \*shmdata;

char \*buffer;

/\*将该共享内存映射到进程的虚拟空间 \*/

shmdata=myshmat(shmid, 0,0);

semid=\*(int \*) shmdata;

buffer=shmdata+sizeof (int);

printf(“\n, writer begin to run, the id of share memory is %d, the semaphore id is %d\n”,shmid, semid);

/\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

writer的主循环

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/

While(1){

char input[3];

printf(“\n menu\n 1.send a message\n”);

printf(“2.quit\n”);

printf(“input your choice (1-2):”);

fgets(input,sizeof (input),stdin);

switch (input[0]){

case ‘1’:write (shmid,semid,buffer);break;

case ‘2’:exit(0);break;

}

}

}

void delete(void){

union semun unused;

printf(“\n quit:delete the semaphore %d\n”,Semid );

/\*删除信号量\*/

If (mysemctl(Semid,0, IPC\_RMID,unused==-1) ){

Printf(“Error releasing semaphore.\n”);

}

}

void sigdelete (int signum){

/\*Calling exit will conveniently trigger the normal delete item.\*/

Exit(0);

}

void locksem(int semid, int semnum){

struct sembuf sb;

sb.sem\_num=semnum;

sb.sem\_op=-1;

sb.sem\_flg=SEM\_UNDO;

mysemop(semid,&sb,1);

}

void unlocksem(int semid, int semnum){

struct sembuf sb;

sb.sem\_num=senmum;

sb.sem\_op=1;

sb.sem\_flg=SEM\_UNDO;

mysemop(semid, &sb,1);

}

void waitzero (int semid, int semnum){

struct sembuf sb;

sb.sem\_num=semnum;

sb.sem\_op=0;

sb.sem\_flg=0; /\*No modification so no need to undo\*/

mysemop(semid,&sb,1);

}

void write (int shmid, int semid, char \*buffer){

printf(“\n wait for reader to read in information”);

fflush(stdout);

locksem(semid, SN\_READ);

printf(“finish\n”);

printf(“please input information”);

fgets(buffer, BUFFERSIZE,stdin);

unlocksem(semid,SN\_WRITE);

}

int mysemget(key\_t key, int nsems, int semflg){

int retval;

retval=semget(key, nsems, semflg);

if (retval==-1){

printf(“semget key %d, nsems %d failed:%s”,key,nsems,sterror(errno));

exit(255);

}

return retval;

}

int mysemctl(int semid, int semnum, int cmd, union semun arg){

int retval;

retval=semctl(semid, semnum, cmd, arg);

if (retval==-1){

printf(“semctl semid %d,semnum %d, cmd %d failed:%s”,semid,semnum,cmd, strerror (errno));

}

return retval;

}

int myshmget(key\_t key, int size, int shmflg){

int retval;

retval=smop(semid, sops, nsops);

if (retval==-1){

printf(“semop semid %d)%d operations)failed:%d”,semid,nsops,strerror(errno));

}

int myshmget(key\_t key, int size, int shmflg){

int retval;

retval=shmget(key, size, shmflg);

if (retval==-1){

printf(“shmget key %d, size %d failed:%s”,key,size,strerror(errno) );

exit(255);

}

return retval;

}

void \*myshmat(int shmid, const void \*shmaddr, int shmflg){

void \*retval;

retval=shmat(shmid,shmaddr,shmflg );

if (retval==(void\*)-1{

printf(“shmat shmid %d failed:%s”, shmid, strerror(errno));

exit(255);

}

return retval;

}

int myshmctl(int shmid, int cmd, struct shmid\_ds \*buf){

int retval;

retval=shmctl(shmid, cmd, buf);

if (retval==-1){

printf(“shmctl shmid %d, cmd %d failed:%s”, shmid, cmd, strerror(errno));

exit(255);

}

return retval;

}

假设上面这个程序名为reader\_writer1.c, 那么使用下面这个命令编译这个程序：

#gcc –o reader\_writer1 reader\_writer1.c

在此之后，即可启动reader，必须先启动reader，因为使用的信号量、共享内存都是在reader中申请的。

在命令行输入：./reader\_writer1

运行结果为：

writer begin to run, the id of share memory is 229376, semaphore id is 196608

menu

1. send a message
2. quit

input your choice1-2:1

wait for reader to read in information…finish

please input information:

在提示后面输入: hello, reader

随即，reader那边将打印这条信息

然后循环往复，不再累赘。

关于退出，writer可以通过菜单退出，reader可以在writer退出后，按Ctrl+C退出，退出reader时，它将自动删除最初申请信号量。

1. **实验思考**

上述程序只能支持一个reader和一个writer。请同学改进程序实现任意数目的reader和writer。

**三、进程/线程同步（信号量）**

信号量是用来解决进程间同步与互斥问题的一种进程间通信机制。程序对信号量的访问都是原子操作，且只允许对它进行等待（P操作）和发送（V操作）操作。

Linux信号量分为Posix信号量和System V信号量（或称为IPC信号量），相比较而言，Posix信号量的使用简单，而System V信号量的使用相对复杂，因此本实验中重点介绍Posix信号量，对System V信号量只坐简要介绍。

1. **System V信号量**

System V信号量可用于任何进程间的同步，使用时需包含两个头文件：sys/ipc.h、sys/sem.h。

**1、函数原型：**

System V 信号量的操作函数主要有3个，其函数原型分别为：

**（1）int semget(key\_t key, int nsems, int oflag)**

功能：创建一个新的信号量或取得一个已存在信号量。

**（2）int semop(int semid, struct sembuf \*opsptr, size\_t nops);**

功能：改变信号量的值，即使用/释放资源使用权。

（3）**int semctl(int semid, int semum, int cmd, ../\* union semun arg \*/);**

功能：删除信号量或初始化信号量的值。

**2、System V信号量的使用步骤：**

使用System V信号量通常分为以下4个步骤：

1. 使用semget()函数创建信号量，或获得在系统中以存在的信号量。不同进程通过使用同一个信号量键值来获得同一个信号量；
2. 使用semctl()函数的SETVAL操作初始化信号量；
3. 调用semop()函数进行信号量的PV操作，这是实现进程间同步和互斥的核心工作。
4. 如果不需要实验信号量，则使用semctl()函数的IPC\_RMID从系统中删除它。
5. **Posix信号量：**

Posix信号量分为无名信号量和有名信号量。无名信号量又称为基于内存的信号量，常用于多线程间的同步，也可用于相关进程间的同步。当无名信号量用于相关进程间同步时，需要放在进程间的共享内存区中。有名信号量通过IPC名字进行进程间的同步，其特点是把信号量保存在文件中，这决定了其用途非常广泛：既可用于线程，也可用于相关进程和不相关进程。

使用Posix信号量时必须包含头文件：**#include <semaphore.h>**

在编译使用Posix信号量的程序时，应加上“**-pthread**”选项，如要编译test1.c文件，可以使用命令：gcc –o test1 –pthread test1.c

1. **无名信号量：**
2. **无名信号量的创建：**

函数原型：int sem\_init(sem\_t \*sem, int pshared, unsigned int value);

函数功能：创建一个新的无名信号量，并进行初始化。  
 参数说明：

Sem：信号量名称；

Pshared：值为0：用于同一进程中多线程间的同步；

值>0：用于多个相关进程间的同步（如父子进程或兄弟进程）

Value：信号量的初始值

返回值：

执行成功返回0，执行失败返回-1.

1. **获取信号量当前值：**

函数原型：int sem\_getvalue(sem\_t \*sem, int \*sval);

函数功能：获取指定信号量当前值，并保存在sval中。

参数说明：

Sem：信号量名称

Sval：所取得的信号量的当前值。

若有1个或多个线程或进程调用sem\_wait()阻塞在该信号量上，该函数返回两种值给sval：  
 1) 返回0  
 2) 返回阻塞在该信号量上的进程或线程数目  
 linux采用了第一种返回策略，及在sval中保存“0”。

返回值：

成功时返回 0；错误时，返回 -1

1. **阻塞型申请资源操作，即P操作：**

函数原型：int sem\_wait(sem\_t \*sem); // 这是一个阻塞的函数  
 函数功能：申请资源，执行信号量的P操作：测试指定信号量sem的值，若大于0，则将sem的值减1后返回；若等于0，则调用线程（或进程）会进入阻塞状态直到另外一个相关的线程（或进程）执行sem\_post()调用，对sem加1，解除了对它的阻塞为止，此时立即将sem减1，然后返回。

参数说明：

Sem：指定的信号量名称

返回值：

函数执行成功返回0；执行错误返回-1，信号量的值不改动。

1. **非阻塞型申请资源操作，即P操作：**

函数原型：int sem\_trywait(sem\_t \*sem); // 非阻塞的函数

函数功能：申请资源，执行信号量的P操作：测试指定信号量sem的值，若大于0，则将sem的值减1后返回；若等于0，调用线程不会进入睡眠状态，直接返回，并标识EAGAIN错误。

参数说明：

Sem：指定的信号量名称

返回值：

函数执行成功返回0；执行错误返回-1，信号量的值不改动。

1. **预置阻塞时间型申请资源操作，即P操作：**

函数原型：int sem\_timedwait(sem\_t \*sem, const struct timespec \*abs\_timeout);

函数功能：申请资源，执行信号量的P操作：测试指定信号量sem的值，若大于0，则将sem的值减1后返回；若等于0，则调用线程会限时等待。当等待时间结束时，若信号量的值仍为0时，则返回错误。

参数说明：

Sem：指定的信号量名称；

Abs\_timeout：调用线程阻塞等待的时间限制。

返回值：

函数执行成功返回0；执行错误返回-1，信号量的值不改动。

1. **释放资源操作，即V操作**

函数原型：int sem\_post(sem\_t \*sem);

函数功能：将指定信号量的值加1，若有线程/进程在等待，则会唤醒其中的一个线程/进程。

参数说明：

Sem：信号量名称

返回值：成功时返回0；错误时，[信号量](http://baike.haosou.com/doc/6799211.html" \t "_blank)的值没有更改，返回-1。

1. **删除无名信号量操作：**

函数原型：int sem\_destroy(sem\_t \*sem);

函数功能：删除由sem\_init()创建的信号量sem。若此时有其他线程/进程阻塞在该信号量上，将导致信号量未定义错误。

返回值：成功时返回 0；错误时，返回 -1

1. **有名信号量**

有名信号量的特点是把信号量的值保存在文件中，所以对于相关进程来说，子进程是继承了父进程的文件描述符，那么子进程所继承的文件描述符所指向的文件是和父进程一样的，当然文件里面保存的有名信号量值就共享了。因此，可以使用有名信号量实现相关进程间的同步。

1. **创建有名信号量：**

函数原型：sem\_t \*sem\_open(const char \*name, int oflag, mode\_t mode ,

int value);

函数功能：打开一个已存在的有名信号量，或创建并初始化一个有名信号量，并将其引用计数加1。

参数说明：

name：用于标识信号量的名字。

**注意**：这里的name不能写成/tmp/aaa.sem这样的格式，因为在linux下，sem都是创建在/dev/shm目录下。你可以将name写成“/mysem”或“mysem”，创建出来的文件都是“/dev/shm/sem.mysem”，千万不要写路径。也千万不要写“/tmp/mysem”之类的。

Oflag：有O\_CREAT或O\_CREAT|EXCL两个取值

当oflag = O\_CREAT时，若name指定的信号量不存在时，则会创建一个，而且后面的mode和value参数必须有效。若name指定的信号量已存在，则直接打开该信号量，同时忽略mode和value参数。

当oflag = O\_CREAT|O\_EXCL时，若name指定的信号量已存在，该函数会直接返回error。

mode\_t：控制新的信号量的访问权限，如0644；

Value：指定信号量的初始化值

返回值：成功时返回指向有名信号量的指针，出错时为SEM\_FAILED。

1. **对信号量的操作函数：**

**sem\_wait()**

**sem\_trywait()**

**sem\_timedwait()**

**sem\_post()**

**sem\_getvalue(0**

**与无名信号量的相应操作函数完全一样。**

1. **有名信号量的删除：**

有名信号量的删除需要两个步骤：

* 1. **关闭信号量：**

函数原型：int sem\_close( sem\_t \*sem)

函数功能：

关闭引用信号量，信号量引用计数减1，但并没有将它从系统里删除，以后还可以使用sem\_open()打开它。

参数说明：

Sem：欲关闭的信号量名称，即在调用sem\_open()操作时的返回值。

返回值:成功时，返回0，失败，-1

* 1. **删除信号量：**

函数原型：int sem\_unlink(const char \*name);

函数功能：从系统中彻底删除该信号量。要注意的是，sem\_unlink()操作只对引用计数为0的信号量有用，对引用计数不为0 的信号量不会起任何作用。

参数说明：

Name：有名信号量的标示符，即在调用sem\_open()操作中的第一个参数值。

返回值:成功时，返回0，失败，-1

1. **信号量使用实例：**
2. **无名信号量的使用：**

实例1：利用无名信号量实现多线程间的同步。

利用无名信号量实现两个线程对共享变量number的互斥使用：

#include <pthread.h>

#include <semaphore.h>

#include <sys/types.h>

#include <stdio.h>

#include <unistd.h>

int number; // 被保护的全局变量

sem\_t sem\_id;

void\* thread\_one\_fun(void \*arg)

{

sem\_wait(&sem\_id); //信号量的P操作：申请资源

printf("thread\_one have the semaphore\n");

number++;

printf("number = %d\n",number);

sem\_post(&sem\_id); //信号量的V操作，释放资源

}

void\* thread\_two\_fun(void \*arg)

{

sem\_wait(&sem\_id); //信号量的P操作：申请资源

printf("thread\_two have the semaphore \n");

number--;

printf("number = %d\n",number);

sem\_post(&sem\_id); //信号量的V操作，释放资源

}

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

{

number = 1;

pthread\_t id1, id2;

sem\_init(&sem\_id, 0, 1);

pthread\_create(&id1,NULL,thread\_one\_fun, NULL);

pthread\_create(&id2,NULL,thread\_two\_fun, NULL);

pthread\_join(id1,NULL);

pthread\_join(id2,NULL);

printf("main,,,\n");

return 0;

}

实例2：上面的实例1中，到底哪个线程先申请到信号量资源，这是随机的。如果想要某个特定的顺序的话，可以用2个信号量来实现。例如下面的例程是线程1先执行完，然后线程2才继续执行，直至结束。

int number; // 被保护的全局变量

sem\_t sem\_id1, sem\_id2;

void\* thread\_one\_fun(void \*arg)

{

sem\_wait(&sem\_id1);

printf("thread\_one have the semaphore\n");

number++;

printf("number = %d\n",number);

sem\_post(&sem\_id2);

}

void\* thread\_two\_fun(void \*arg)

{

printf("thread\_two have the semaphore \n");

number--;

printf("number = %d\n",number);

sem\_post(&sem\_id1);

}

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

{

number = 1;

pthread\_t id1, id2;

sem\_init(&sem\_id1, 0, 1); // 空闲的

sem\_init(&sem\_id2, 0, 0); // 忙的

pthread\_create(&id1,NULL,thread\_one\_fun, NULL);

pthread\_create(&id2,NULL,thread\_two\_fun, NULL);

pthread\_join(id1,NULL);

pthread\_join(id2,NULL);

printf("main,,,\n");

return 0;

}

实例3：利用无名信号量实现多个相关进程间的同步。

在本实例中，共有2个进程，其中一个是另外一个的子进程（由fork产生）的。 本来对于fork来说，子进程只继承了父进程的代码副本，mutex在父子进程中应是相互独立的两个变量，但由于在初始化信号量mutex的时候，由参数pshared = 1指定了mutex处于共享内存区域，所以此时mutex变成了父子进程共享的一个变量。此时，mutex就可以用来同步相关进程了。

#include <semaphore.h>

#include <stdio.h>

#include <errno.h>

#include <stdlib.h>

#include <unistd.h>

#include <sys/types.h>

#include <sys/stat.h>

#include <fcntl.h>

#include <sys/mman.h>

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

{

int fd, i,count=0,nloop=10,zero=0,\*ptr;

sem\_t mutex;

//open a file and map it into memory

fd = open("log.txt",O\_RDWR|O\_CREAT,S\_IRWXU);

write(fd,&zero,sizeof(int));

ptr = mmap( NULL,sizeof(int),PROT\_READ |PROT\_WRITE,MAP\_SHARED,fd,0 );

close(fd);

/\* create, initialize semaphore \*/

if( sem\_init(&mutex,1,1) < 0) //

{

perror("semaphore initilization");

exit(0);

}

if (fork() == 0)

{ /\* child process\*/

for (i = 0; i < nloop; i++)

{

sem\_wait(&mutex);

printf("child: %d\n", (\*ptr)++);

sem\_post(&mutex);

}

exit(0);

}

/\* back to parent process \*/

for (i = 0; i < nloop; i++)

{

sem\_wait(&mutex);

printf("parent: %d\n", (\*ptr)++);

sem\_post(&mutex);

}

exit(0);

}

1. **有名信号量的使用：**

前面已经说过，有名信号量是位于共享内存区的，那么它要保护的资源也必须是位于共享内存区，只有这样才能被无相关的进程所共享。

实例4：在下面这个例子中，服务进程和客户进程是两个不相关进程，但它们都使用shmget和shmat来获取得一块共享内存资源。然后利用有名信号量来对这块共享内存资源进行互斥保护。

**<u>File1: server.c </u>**

#include <sys/types.h>

#include <sys/ipc.h>

#include <sys/shm.h>

#include <stdio.h>

#include <semaphore.h>

#include <sys/types.h>

#include <sys/stat.h>

#include <fcntl.h>

#define SHMSZ 27

char SEM\_NAME[]= "vik";

int main()

{

char ch;

int shmid;

key\_t key;

char \*shm,\*s;

sem\_t \*mutex;

key = 1000; //name the shared memory segment

//create & initialize semaphore

mutex = sem\_open(SEM\_NAME,O\_CREAT,0644,1);

if(mutex == SEM\_FAILED)

{

perror("unable to create semaphore");

sem\_unlink(SEM\_NAME);

exit(-1);

}

//create the shared memory segment with this key

shmid = shmget(key,SHMSZ,IPC\_CREAT|0666);

if(shmid<0)

{

perror("failure in shmget");

exit(-1);

}

//attach this segment to virtual memory

shm = shmat(shmid,NULL,0);

//start writing into memory

s = shm;

for(ch='A';ch<='Z';ch++)

{

sem\_wait(mutex);

\*s++ = ch;

sem\_post(mutex);

}

//the below loop could be replaced by binary semaphore

while(\*shm != '\*')

{

sleep(1);

}

sem\_close(mutex);

sem\_unlink(SEM\_NAME);

shmctl(shmid, IPC\_RMID, 0);

exit(0);

}

<u>**File 2: client.c**</u>

#include <sys/types.h>

#include <sys/ipc.h>

#include <sys/shm.h>

#include <stdio.h>

#include <semaphore.h>

#include <sys/types.h>

#include <sys/stat.h>

#include <fcntl.h>

#define SHMSZ 27

char SEM\_NAME[]= "vik";

int main()

{

char ch;

int shmid;

key\_t key;

char \*shm,\*s;

sem\_t \*mutex;

//name the shared memory segment

key = 1000;

//create & initialize existing semaphore

mutex = sem\_open(SEM\_NAME,0,0644,0);

if(mutex == SEM\_FAILED)

{

perror("reader:unable to execute semaphore");

sem\_close(mutex);

exit(-1);

}

//create the shared memory segment with this key

shmid = shmget(key,SHMSZ,0666);

if(shmid<0)

{

perror("reader:failure in shmget");

exit(-1);

}

//attach this segment to virtual memory

shm = shmat(shmid,NULL,0);

//start reading

s = shm;

for(s=shm;\*s!=NULL;s++)

{

sem\_wait(mutex);

putchar(\*s);

sem\_post(mutex);

}

//once done signal exiting of reader:This can be replaced by another semaphore

\*shm = '\*';

sem\_close(mutex);

shmctl(shmid, IPC\_RMID, 0);

exit(0);

}