**广州大学学生实验报告**

**开课学院及实验室：**计算机科学与网络工程学院软件实验室 **2023年 5月 日**

|  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- |
| **学院** | **计算机科学与网络工程学院** | **年级/专业/班** | **软件211** | **姓名** | 张景致 | **学号** | 32106300004 |
| **实验课程名称** | 操作系统实验 | | | | | **成绩** |  |
| **实验项目名称** | 进程管理与进程通信 | | | | | **指导老师** |  |

**实验一 进程管理与进程通信**

1. **实验目的**

1、掌握进程的概念，明确进程的含义。

2、认识并了解进程并发执行的实质，进程的阻塞与唤醒，终止与退出的过程。

3、熟悉进程的睡眠、同步、撤消等进程控制方法。

4、分析进程竞争资源的现象，学习解决进程互斥的方法 。

5、了解什么是信号，利用信号量机制熟悉进程间软中断通信的基本原理，

6、熟悉消息传送的机理 ，共享存储机制 。

1. **实验环境**

Linux archlinux 6.3.1-zen1-1-zen #1 ZEN SMP PREEMPT\_DYNAMIC Mon, 01 May 2023 17:42:12 +0000 x86\_64 GNU/Linux

clang version 15.0.7

Target: x86\_64-pc-linux-gnu

Thread model: posix

InstalledDir: /usr/bin

1. **实验内容**
2. **实验原理 实验中用到的系统调用函数（包括实验原理中介绍的和自己采用的），实验步骤。**

本实验的用到的主要系统调用函数：

fork, exec, wait, exit, getpid, sleep, lockf, kill, signal, read, write, msgget, msgsnd, msgrcv, msgctl，shmget, shmat, shmdt, shmctl

***·实验步骤·***

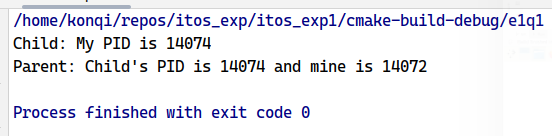
共享头文件common.h：  
#ifndef **ITOS\_EXP1\_COMMON\_H**#define **ITOS\_EXP1\_COMMON\_H**#include <stdio.h>  
#include <string.h>  
#include <stdlib.h>  
  
#include <fcntl.h>  
#include <unistd.h>  
  
#include <sys/wait.h>  
#include <sys/file.h>  
#include <sys/ipc.h>  
#include <sys/msg.h>  
#include <sys/shm.h>  
#include <sys/mman.h>  
  
#endif *//ITOS\_EXP1\_COMMON\_H*

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并发执行，观察实验结果并分析原因。

所用代码：

#include "common.h"  
  
int main() {  
 pid\_t pid = fork();  
 if (pid == 0) {  
 *// child* printf("Child: My PID is %d\n", getpid());  
 } else if (pid > 0) {  
 *// parent* wait(0);  
 printf("Parent: Child's PID is %d and mine is %d\n", pid, getpid());  
 } else {  
 *// fork failed* printf("Fork failed.");  
 }  
 return 0;  
}

运行结果：



原因分析：

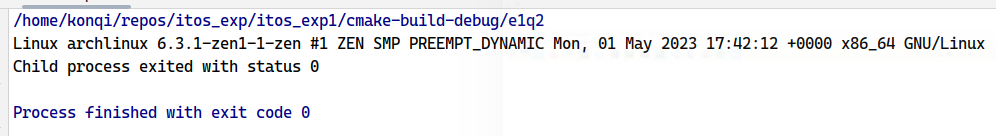
fork系统调用成功发生后，程序分为两个代码上相同的子母进程运行，而母进程从fork系统调用获得了子进程的pid。

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，利用wait( )来控制进程执行顺序，掌握进程的睡眠、同步、撤消等进程控制方法，并根据实验结果分析原因。

所用代码：

#include "common.h"  
  
int main() {  
 pid\_t pid = fork();  
 if (pid == 0) {  
 *// child process* char \*args[] = {"/usr/bin/uname", "-a", NULL};  
 int ret = execv(args[0], args);  
 if (ret == -1) {  
 perror("execv");  
 exit(**EXIT\_FAILURE**);  
 }  
 } else if (pid > 0) {  
 *// parent process* int status;  
 wait(&status);  
 if (**WIFEXITED**(status)) {  
 printf("Child process exited with status %d\n", **WEXITSTATUS**(status));  
 }  
 } else {  
 *// fork failed* perror("fork");  
 exit(**EXIT\_FAILURE**);  
 }  
 return 0;  
}

运行结果：



原因分析：

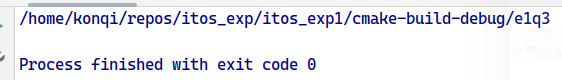
fork调用成功后，子进程execv调用启动了uname -a命令，母进程通过wait等待并获取了进程退出状态。另设有检查系统调用成功与否的结构。

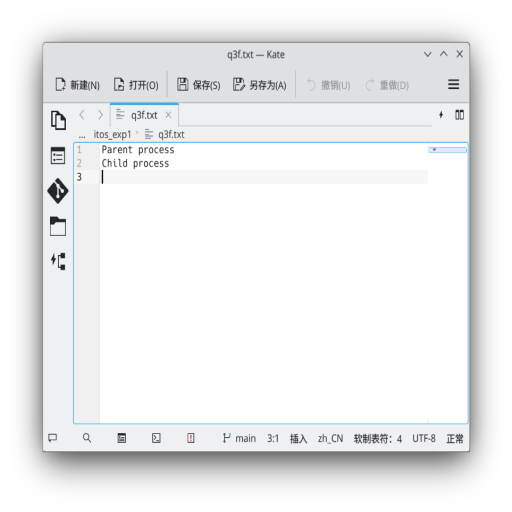
1. 编写一段多进程并发运行的程序，用lockf( )来给每一个进程加锁，以实现进程之间的互斥，观察并分析出现的现象及原因。

所用代码：

#include "common.h"  
  
int main() {  
 pid\_t pid;  
 int fd;  
 char buf[20];  
  
 fd = open("q3f.txt", **O\_WRONLY** | **O\_CREAT**, 0644);  
 if (fd == -1) {  
 perror("open");  
 exit(**EXIT\_FAILURE**);  
 }  
  
 pid = fork();  
 if (pid == -1) {  
 perror("fork");  
 exit(**EXIT\_FAILURE**);  
 } else if (pid == 0) {  
 *//lock* if (lockf(fd, **F\_LOCK**, 0) == -1) {  
 perror("lockf");  
 exit(**EXIT\_FAILURE**);  
 }  
  
 sprintf(buf, "Child process\n");  
 write(fd, buf, strlen(buf));  
  
 *//unlock* if (lockf(fd, **F\_ULOCK**, 0) == -1) {  
 perror("lockf");  
 exit(**EXIT\_FAILURE**);  
 }  
  
 exit(**EXIT\_SUCCESS**);  
 }  
  
 *//lock* if (lockf(fd, **F\_LOCK**, 0) == -1) {  
 perror("lockf");  
 exit(**EXIT\_FAILURE**);  
 }  
  
 sprintf(buf, "Parent process\n");  
 write(fd, buf, strlen(buf));  
  
 *// unlock* if (lockf(fd, **F\_ULOCK**, 0) == -1) {  
 perror("lockf");  
 exit(**EXIT\_FAILURE**);  
 }  
  
 exit(**EXIT\_SUCCESS**);  
}

运行结果：





原因分析：

首先一个文件被打开，然后子进程被创建。子母进程分别在尝试使用sprintf写入内容前使用lockf锁定了文件，使用完之后释放之。

4、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：

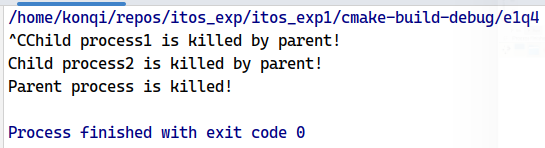
Parent process is killed!

分析利用信号量机制中的软中断通信实现进程同步的机理。

使用代码：

#include "common.h"  
  
pid\_t pid1, pid2;  
  
void handle\_signal(int signal) {  
 if (signal != **SIGINT**) {  
 return;  
 }  
 kill(pid1, **SIGINT**);  
 kill(pid2, **SIGINT**);  
}  
  
void handle\_child1\_signal(int signal) {  
 if (signal != **SIGINT**) {  
 return;  
 }  
 printf("Child process1 is killed by parent!\n");  
 exit(0);  
}  
  
void handle\_child2\_signal(int signal) {  
 if (signal != **SIGINT**) {  
 return;  
 }  
 printf("Child process2 is killed by parent!\n");  
 exit(0);  
}  
  
int main() {  
 *// Register signal handler for parent process* signal(**SIGINT**, handle\_signal);  
  
 pid1 = fork();  
 if (pid1 == -1) {  
 perror("fork");  
 exit(1);  
 } else if (pid1 == 0) {  
 *// Child process 1* signal(**SIGINT**, handle\_child1\_signal);  
 while (1) sleep(1);  
 }  
  
 pid2 = fork();  
 if (pid2 == -1) {  
 perror("fork");  
 exit(1);  
 } else if (pid2 == 0) {  
 *// Child process 2* signal(**SIGINT**, handle\_child2\_signal);  
 while (1) sleep(1);  
 }  
  
 *// Parent process* int status;  
 waitpid(pid1, &status, 0);  
 waitpid(pid2, &status, 0);  
 printf("Parent process is killed!\n");  
 return 0;  
}

运行结果：



其中输出第一行的^C为用户输入的Control-C控制字符 / SIGINT信号。

原因分析：

在使用signal调用正确注册了SIGINT信号处理程序之后，两个子程序均对用户输入的SIGINT做出了响应（结束自身）。其中软中断使得用户程序的运作暂停，从而得以实现互斥。

每个进程在运行时，都要通过信号机制来检查是否有信号到达。若有，便中断正在执行的程序，转向与该信号相对应的处理程序，以完成对该事件的处理；处理结束后再返回到原来的断点继续执行。

1. 使用系统调用msgget( ),msgsnd( ),msgrev( ),及msgctl( )编制一长度为1k的消息发送和接收的程序，并分析消息的创建、发送和接收机制及控制原理。

使用代码：

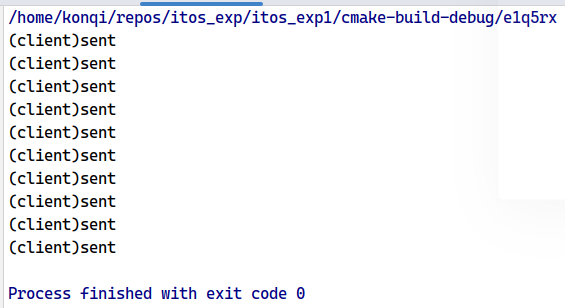
[TX]

#include "common.h"  
  
#define **MSGKEY** 7775  
  
struct msgform  
{  
 long mtype;  
 char mtext[1000];  
} msg;  
  
int msgqid;  
  
void client()  
{  
 int i;  
 msgqid=msgget(**MSGKEY**,0777); */\*打开75#消息队列\*/* for(i=10;i>=1;i--)  
 {  
 msg.mtype=i;  
 printf("(client)sent\n");  
 msgsnd(msgqid,&msg,1024,0); */\*发送消息\*/* }  
 exit(0);  
}  
  
int main()  
{  
 client();  
}

[RX]

#include "common.h"  
  
#define **MSGKEY** 7775  
  
struct msgform  
{  
 long mtype;  
 char mtext[1000];  
} msg;  
  
int msgqid;  
  
void server( )  
{  
 msgqid=msgget(**MSGKEY**,0777|**IPC\_CREAT**); */\*创建75#消息队列\*/* do  
 {  
 msgrcv(msgqid,&msg,1030,0,0); */\*接收消息\*/* printf("(server)received\n");  
 }while(msg.mtype!=1);  
 msgctl(msgqid,**IPC\_RMID**,0); */\*删除消息队列，归还资源\*/* exit(0);  
}  
  
int main()  
{  
 server();  
}

运行结果：





原因分析：

操作系统托管消息队列头表，并在应用程序需要时在其中创建新表或根据标识符寻找已有表。消息创建并发送后会被加入消息头并链接到队列中等待接收，另一个进程接收时操作系统便会将其从队列弹出并向接收进程提供其内容。

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。

1. 编制一长度为1k的共享存储区发送和接收的程序，并设计对该共享存储区进行互斥访问及进程同步的措施，必须保证实现正确的通信。

使用代码：

#include "common.h"  
  
#define **SHMKEY** 7755  
  
int shmid, i;  
int \*addr;  
  
void client()  
{  
 int i;  
 shmid=shmget(**SHMKEY**,1024,0777); */\*打开共享存储区\*/* addr=shmat(shmid,0,0); */\*获得共享存储区首地址\*/* 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**); */\*创建共享存储区\*/* addr=shmat(shmid,0,0); */\*获取首地址\*/* do  
 {  
 \*addr=-1;  
 while (\*addr==-1);  
 printf("(server) received\n");  
 }while (\*addr);  
 shmctl(shmid,**IPC\_RMID**,0); */\*撤消共享存储区，归还资源\*/* exit(0);  
}  
  
int main( )  
{  
 while ((i=fork())==-1);  
 if (!i) server();  
 system("ipcs -m");  
 while ((i=fork())==-1);  
 if (!i) client();  
 wait(0);  
 wait(0);  
}

运行结果：

/home/konqi/repos/itos\_exp/itos\_exp1/cmake-build-debug/e1q6rx

------------ 共享内存段 --------------

键 shmid 拥有者 权限 字节 连接数 状态

0x00000000 6 konqi 606 4848984 2 目标

0x00000000 7 konqi 606 4848984 2 目标

0x00000000 8 konqi 600 1323008 2 目标

0x00000000 9 konqi 600 1323008 2 目标

0x00000000 393226 konqi 600 4 2 目标

0x00000000 393228 konqi 600 454152 2 目标

0x00000000 65551 konqi 600 7872000 2 目标

0x00000000 98325 konqi 600 176952 2 目标

0x00000000 98326 konqi 600 1301304 2 目标

0x00000000 38 konqi 600 7872000 2 目标

0x5122023b 39 konqi 600 1 0

0x00000000 98353 konqi 600 629104 2 目标

0x00000000 589878 konqi 600 41760 2 目标

0x00001e4b 589879 konqi 777 1024 1

(client) sent

(server) received

(client) sent

(server) received

(client) sent

(server) received

(client) sent

(server) received

(client) sent

(server) received

(client) sent

(server) received

(client) sent

(server) received

(client) sent

(server) received

(client) sent

(server) received

(client) sent

(server) received

原因分析：

该应用程序成功使用shmget和shmat调用开启了7755号共享内存，并向其中存储数据，实现了两个进程之间的数据发送功能，最后使用shmctl调用撤销了共享内存使用。

**思考题**

**1、进程创建与进程并发执行**

（1）系统是怎样创建进程的？

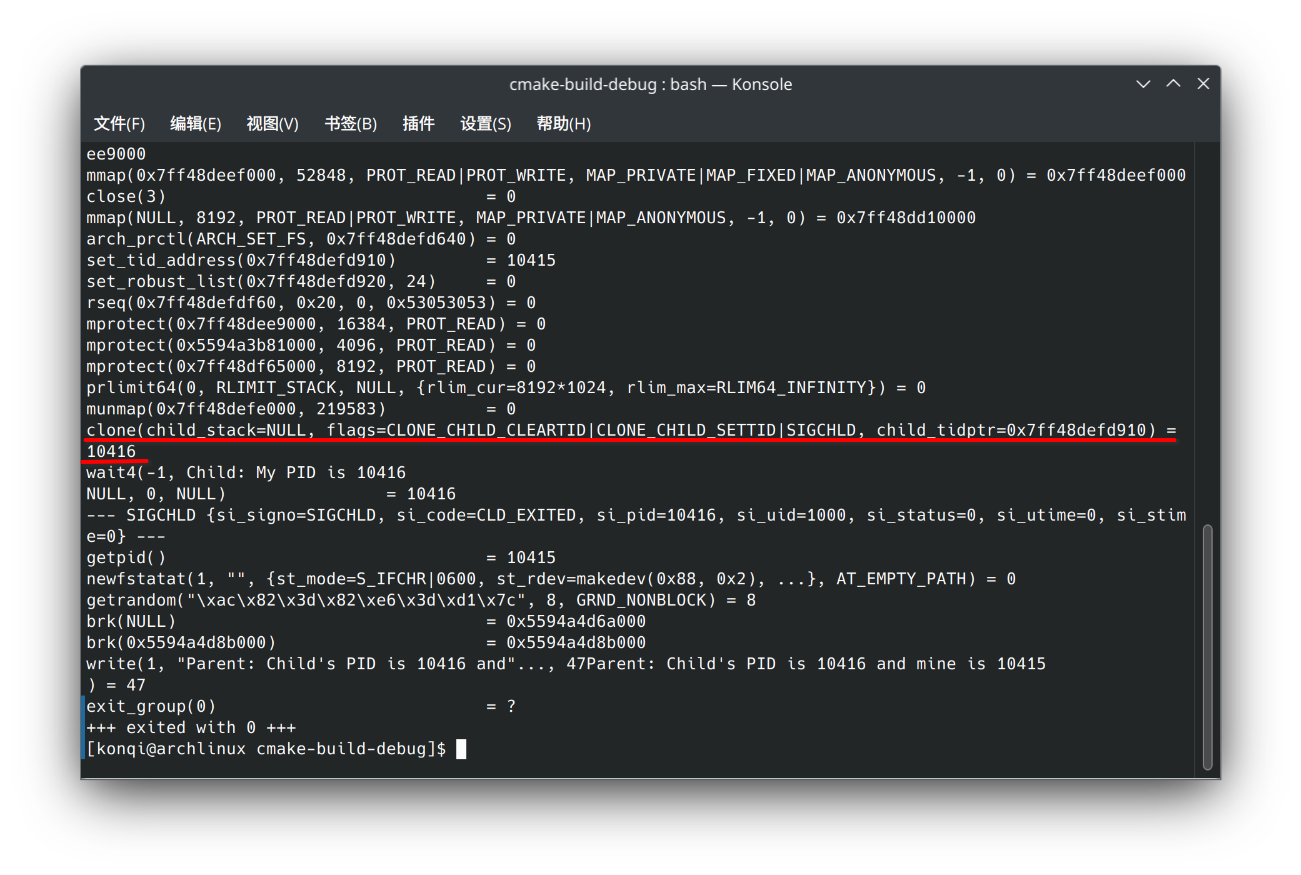
系统创建并初始化新进程的信息（id、母进程id等），并分配其内存空间，然后开始执行对应的任务。

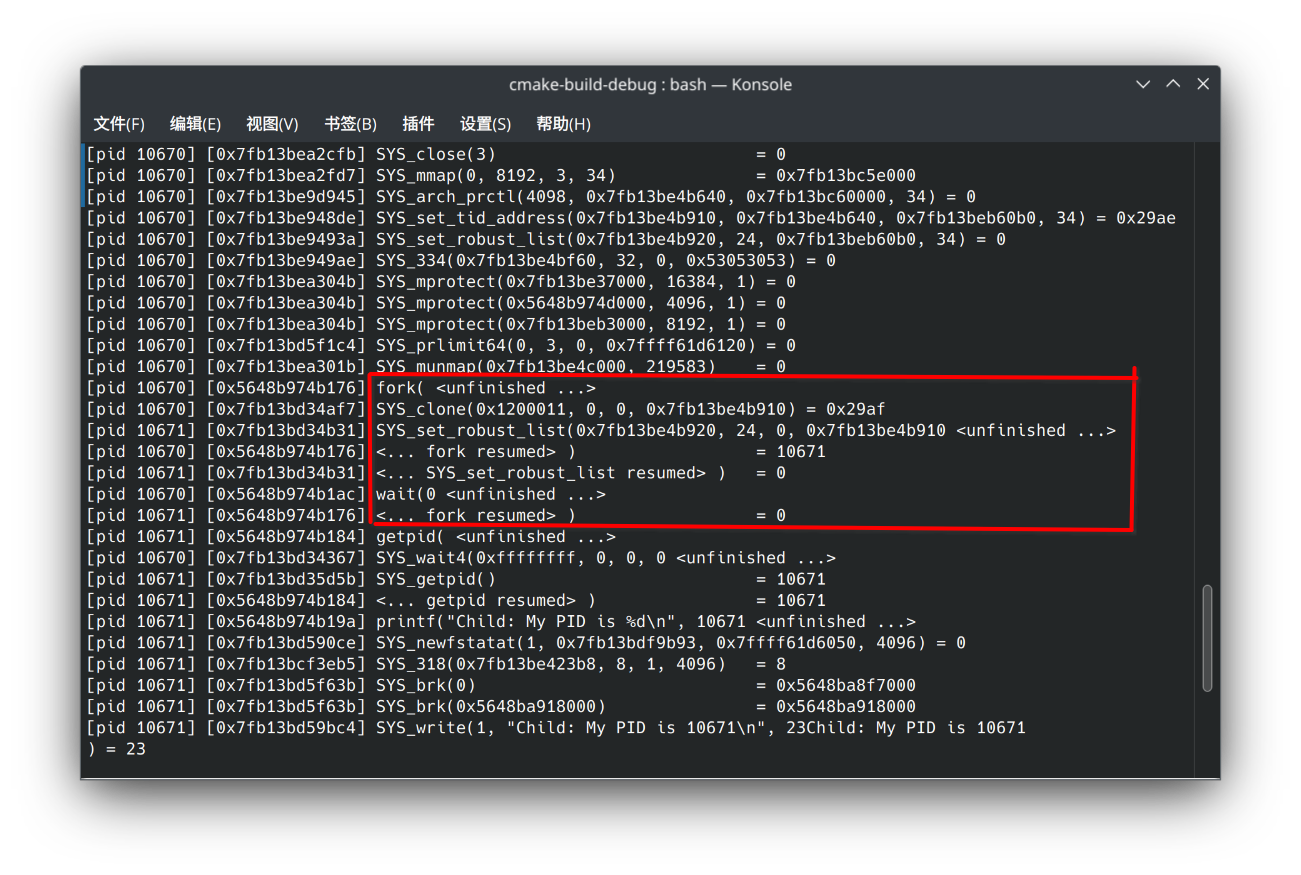
1. 当首次调用新创建进程时，其入口在哪里？

入口位于fork系统调用返回处（其创建者的执行位置）。

（3）利用strace 和ltrace -f -i -S ./executable-file-name查看程序

执行过程，并分析原因，画出进程家族树。

strace可见有1处由fork引发的clone系统调用。



ltrace可见fork及其对应的clone调用。

可见进程家族为 母进程-->子进程

**2、进程的睡眠、同步、撤消等进程控制**

（1）可执行文件加载时进行了哪些处理？

其资源状态、内存结构得到初始化。同时应用程序在运行前期动态链接了libc等库。

1. 什么是进程同步？wait( )是如何实现进程同步的？

进程同步指的是多个进程在共享资源的过程中保持一致性的机制。wait通过使得当前进程等待另一个进程退出来实现进程同步。

1. wait( )和exit（）是如何控制实验结果的随机性的？

这两个函数使得前后代码的执行具备一定可预见的先后顺序（等待、被等待、主动退出等）。

**3、多进程通过加锁互斥并发运行**

（1）进程加锁和未上锁的输出结果相同吗？ 为什么？

不同。未加锁会发生同时写入和读取的问题。

**4、进程间通过信号机制实现软中断通信**

（1）为了得到实验内容要求的结果，需要用到哪些系统调用函数来实现及进程间的通信控制和同步？

signal、kill、exit、wait

1. kill( )和signal( )函数在信号通信中的作用是什么？如果分别注释掉它们，结果会如何？

作用是发送信号和提供信号处理程序。会导致信号不被发送、不被处理（或采用默认处理方式）。

**5、消息的发送与接收**

（1）为了便于操作和观察结果，需要编制几个程序分别用于消息的发送与接收？

2个

1. 这些程序如何进行编辑、编译和执行？为什么？

使用编辑器程序（vi、nano、CLion等）进行编辑，使用C编译器（gcc、clang等）编译，通过Linux执行。因为程序代码需要被编译成对应体系结构、操作系统、ABI的机器代码才能被接受执行。

1. 如何实现消息的发送与接收的同步？

使用锁、信号、信号量等同步手段。

**6、进程的共享存储区通信**

（1）为了便于操作和观察结果，需要如何合理设计程序来实现子进程间的共享存储区通信？

使用同一个程序的不同子进程进行发送和接收操作。

（2）比较消息通信和共享存储区通信这两种进程通信机制的性能和优缺点。

消息通信通过操作系统托管，使用简单方便，但是性能因为繁琐的操作流程受到限制。共享内存由应用程序自行协商消息机制，自由灵活性能优，但是设置复杂度较高，难以做到安全使用。

1. **实验结果分析（截屏的实验结果，与实验结果对应的实验分析）**
2. 实验结果与实验程序、实验步骤、实验原理、操作系统原理的对应分析

（见上文）

1. 不同条件下的实验结果反应的问题及原因

由于创建了线程，在此之后代码的先后情况是难以确定的，但可能由于linux-zen内核的抢占特性，通常子进程在被创建之后首先得到执行。

1. 实验结果的算法时间、效率、鲁棒性等性能分析

每个实验的算法都较为直白，总体上呈现出顺序执行的特征，未见递归和循环分配、堆分配的现象，时间、空间效率上不认为存在明显的隐患。在使用fork和exec调用的时候适当检查了函数结果。

**六、实验总结**