操作系统原理 实用实验教程

部分示例代码更正

2022.2.12

本文档对2020版《操作系统原理 实用实验教程》中存在的部分有bug或不足的示例代码，进行了更正。因为时间精力等原因，这些更正的代码并未完成详细彻底的严格测试，欢迎同学们批评指正。

**本文档是《操作系统原理 实用实验教程》的修订附件，需要与《操作系统原理 实用实验教程》一起阅读。本文档出现的带小节号的部分(比如：1.3 示例实验)，替代了原教程中的相应小节；对本文档不存在的小节，请看原教程的内容。**

**本文档包含第二部分算法实验中的实验一、三、四、五及七的示例实验小节。**

部分同学发现了2020版实验教程中的实例代码存在bug，并提供了修改后的代码。部分同学对示例代码的改进，提出了很好的意见建议。在此表示深切的感谢！这些同学包括：殷梓淇、张凌霄、李卓栋、雍奥等。

**第二部分 操作系统算法实验**

实验一、进程控制实验

1.3 示例实验

以下实验示例程序应实现一个类似子shell子命令的功能,它可以从执行程序中启动另一个新的子进程并执行一个新的命令和其并发执行.

**1) 打开一终端命令行窗体，新建一个文件夹，在该文件夹中建立以下名为pctl.c的C语言程序：**

/\*

\* Filename : pctl.c

\* copyright : (C) 2006 by zhanghonglie

\* Function : 父子进程的并发执行

\*/

#include "pctl.h"

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

{

int i;

int pid; //存放子进程号

int status; //存放子进程返回状态

char \*args[] = {"/bin/ls", "-a", NULL}; //子进程要缺省执行的命令

signal(SIGINT, (sighandler\_t)sigcat); //注册一个本进程处理键盘中断的函数

pid = fork(); //建立子进程

if (pid < 0) { //建立子进程失败？

printf("Create Process fail!\n");

exit(EXIT\_FAILURE);

}

if (pid == 0) { //子进程执行代码段

//报告父子进程进程号

printf("I am Child process %d\nMy father is %d\n", getpid(), getppid());

pause(); //暂停，等待键盘中断信号唤醒

//子进程被键盘中断信号唤醒继续执行

printf("%d child will Running: ", getpid());

if (argv[1] != NULL) {

//如果在命令行上输入了子进程要执行的命令

//则执行输入的命令

for (i = 1; argv[i] != NULL; i++)

printf("%s ", argv[i]);

printf("\n");

//装入并执行新的程序

status = execve(argv[1], &argv[1], NULL);

}

else {

//如果在命令行上没输入子进程要执行的命令

//则执行缺省的命令

for (i = 0; args[i] != NULL; i++)

printf("%s ", args[i]);

printf("\n");

//装入并执行新的程序

status = execve(args[0], args, NULL);

}

}

else { //父进程执行代码段

printf("\nI am Parent process %d\n", getpid()); //报告父进程进程号

if (argv[1] != NULL) {

//如果在命令行上输入了子进程要执行的命令

//则父进程等待子进程执行结束

printf("%d Waiting for child done.\n\n", getpid());

waitpid(pid, &status, 0); //等待子进程结束

printf("\nMy child exit! status = %d\n\n", status);

}

else {

//如果在命令行上没输入子进程要执行的命令

//唤醒子进程，与子进程并发执行不等待子进程执行结束，

**sleep(3);** //仅用于演示，注意并非完全可靠！

//----如果不加上面一行让父进程等待一会，

//----在父进程先执行的情况下，

//----父进程还没等到子进程暂停，

//----就向子进程发出SIGINT信号，导致子进程无法继续

if (kill(pid, SIGINT) >= 0)

printf("%d Wakeup %d child.\n", getpid(), pid);

printf("%d don't Wait for child done.\n\n", getpid());

}

}

return EXIT\_SUCCESS;

}

**2) 再建立以下名为pctl.h的C语言头文件：**

#include <signal.h>

#include <stdio.h>

#include <stdlib.h>

#include <sys/types.h>

#include <unistd.h>

#include <wait.h>

//进程自定义的键盘中断信号处理函数

typedef void (\*sighandler\_t) (int);

void sigcat() { printf("%d Process continue\n", getpid()); }

**3) 建立以下项目管理文件Makefile**

head = pctl.h

srcs = pctl.c

objs = pctl.o

opts = -g -c

all: pctl

pctl: $(objs)

gcc $(objs) -o pctl

pctl.o: $(srcs) $(head)

gcc $(opts) $(srcs)

clean:

rm pctl \*.o

**在Makefile中，一个规则可以有多个命令行，每一条命令占一行。注意：每一个命令行必须以[Tab]字符开始，[Tab]字符告诉make此行是一个命令行。make按照命令完成相应的动作。这也是书写Makefile中容易产生，而且比较隐蔽的错误。**

**4) 输入make命令编译连接生成可执行的pctl程序**

$ *make*

gcc -g -c pctl.c

gcc pctl.o -o pctl

**5) 执行pctl程序(注意进程号是动态产生的，每次执行都不相同)**

*$ ./pctl*

I am Parent process 3088

I am Child process 3089

My father is 3088

3088 Wakeup 3089 child.

3088 don't Wait for child done.

3089 Process continue

3089 child will Running: /bin/ls -a

. .. Makefile pctl pctl.c pctl.h pctl.o

以上程序的输出说明父进程3088创建了一个子进程3089，子进程执行被暂停。父进程向子进程发出键盘中断信号唤醒子进程并与子进程并发执行。父进程并没有等待子进程的结束继续执行先行结束了（此时的子进程成为了孤儿进程，不会有父进程为它清理退出状态了）。而子进程继续执行，它变成了列出当前目录所有文件名的命令ls -a。在完成了列出文件名命令之后，子进程的执行也结束了。此时子进程的退出状态将由初始化进程为它清理。

**6) 再次执行带有子进程指定执行命令的pctl程序:**

$ ./pctl /bin/ls -l

I am Parent process 3165

3165 Waiting for child done.

I am Child process 3166

My father is 3165

可以看到这一次子进程仍然被挂起，而父进程则在等待子进程的完成。为了检测父子进程是否都在并发执行，请输入ctrl+z将当前进程放入后台并输入ps命令查看当前系统进程信息，显示如下：

[1]+ Stopped ./pctl /bin/ls -l

$ *ps -l*

F S UID PID PPID C PRI NI ADDR SZ WCHAN TTY TIME CMD

0 S 1000 3116 3109 0 80 0 - 4812 do\_wai pts/0 00:00:00 bash

0 T 1000 3165 3116 0 80 0 - 624 do\_sig pts/0 00:00:00 pctl

1 T 1000 3166 3165 0 80 0 - 624 do\_sig pts/0 00:00:00 pctl

0 R 1000 3167 3116 0 80 0 - 5013 - pts/0 00:00:00 ps

可以看到当前系统中同时有两个叫pctl的进程，它们的进程号分别是3165和3166。它们的状态都为“T”,说明当前都被挂起。3166的父进程是3165，而3165的父进程是3109，也就是bash-shell。为了让pctl父子进程继续执行，请输入fg命令让pctl再次返回前台，显示如下：

$ fg

./pctl /bin/ls -l

现在pctl父子进程从新返回前台。我们可以通过键盘发键盘中断信号来唤醒pctl父子进程继续执行，输入ctrl+c，将会显示：

3165 Process continue

3166 Process continue

3166 child will Running: /bin/ls -l

total 52

-rw-rw-r-- 1 u1 u1 165 Jan 15 18:09 Makefile

-rwxrwxr-x 1 u1 u1 20512 Jan 17 19:26 pctl

-rw-rw-r-- 1 u1 u1 2936 Jan 15 18:52 pctl.c

-rw-rw-r-- 1 u1 u1 268 Jan 15 18:08 pctl.h

-rw-rw-r-- 1 u1 u1 9544 Jan 17 19:26 pctl.o

My child exit! status = 0

以上输出说明了子进程在捕捉到键盘中断信号后继续执行了指定的命令，按我们要求的长格式列出了当前目录中的文件名，父进程在接收到子进程执行结束的信号后将清理子进程的退出状态并继续执行，它报告了子进程的退出编码（0 表示子进程正常结束)最后父进程也结束执行。

实验三、进程调度算法实验

3.3 示例实验

以下示例实验程序要测试在linux系统中不同调度策略和不同优先数的调度效果。

**1) 在新建文件夹中建立以下名为psched.c的C语言程序：**

/\*

\* Filename : psched.c

\* Copyright : (C) 2006 by zhanghonglie

\* Last update : 2022.1.19

\* Function : 父进程创建3个子进程，为它们设置不同的优先数和调度策略

\* 注意提高进程优先级需要超级用户的权限才可以起作用

\* Usage:

sudo ./psched [pri0 [pri1 [pri2 [pol0 [pol1 [pol2]]]]]]

\*/

#include <stdio.h>

#include <stdlib.h>

#include <unistd.h>

#include <sched.h>

#include <sys/resource.h>

#define LOOPS 200000000UL

unsigned long cal(unsigned long nLoops)

{

unsigned long i, sum = 0UL;

for(i = 0UL; i < nLoops; i++)

sum++;

return sum;

}

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

{

int i;

int pid[3]; //存放进程号

struct sched\_param p[3]; //设置调度策略时使用的数据结构

int pol[3], pri[3]; // policy & priority

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

//循环创建3个子进程

if((pid[i] = fork()) > 0) { // Parent

//取进程优先数放在调度策略数据结构中

if(argc > (i + 4))

pol[i] = atoi(argv[i + 4]);

else

pol[i] = SCHED\_OTHER;

if(argc > (i + 1))

pri[i] = atoi(argv[i + 1]);

else

pri[i] = 10;

if(pol[i] == SCHED\_OTHER)

p[i].sched\_priority = 0;

else

p[i].sched\_priority = pri[i];

//父进程设置子进程的调度策略.如果命令行第4,5,6 参数指定了3个策略值则按指定的数设置,否则都为默认策略

sched\_setscheduler(pid[i], pol[i], &p[i]);

//父进程设置子进程的优先数,如果命令行第1,2,3 参数指定了3个优先数则按指定的数设置,否则都为10

setpriority(PRIO\_PROCESS, pid[i], pri[i]);

}

//各子进程循环报告其优先数和调度策略

else{ // Child

sleep(3); // Wait parent run. Not reliable, for demo only!

//每个子进程报告10次进程号和优先级，两次报告之间调用一次耗费CPU的cal函数

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

printf("Child PID = %d %d priority = %d @%lu\n", getpid(), i, getpriority(PRIO\_PROCESS, 0), cal(LOOPS));

exit(EXIT\_SUCCESS);

}

}

//父进程报告子进程调度策略后先行退出

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

printf("My child %d policy is %d\n", pid[i], sched\_getscheduler(pid[i]));

return EXIT\_SUCCESS;

}

2) **在当前目录中建立以下Makefile文件：**

srcs = psched.c

objs = psched.o

opts = -g -c

all: psched

psched: $(objs)

gcc $(objs) -o psched

psched.o: $(srcs)

gcc $(opts) $(srcs)

clean:

rm psched \*.o

**3) 使用make命令编译连接生成可执行文件psched:**

$ make

gcc -c psched.c

gcc psched.o -o psched

**4) 以超级用户权限运行psched，指定3个子进程的优先数为10, 5, -10；调度策略都是默认策略**

注意提高进程优先级需要超级用户的权限才可以起作用。

$ **sudo** ./psched 10 5 -10 0 0 0

My child 3195 policy is 0

My child 3196 policy is 0

My child 3197 policy is 0

Child PID = 3197 0 priority = -10 @200000000

Child PID = 3196 0 priority = 5 @200000000

Child PID = 3197 1 priority = -10 @200000000

Child PID = 3196 1 priority = 5 @200000000

Child PID = 3197 2 priority = -10 @200000000

Child PID = 3195 0 priority = 10 @200000000

Child PID = 3197 3 priority = -10 @200000000

Child PID = 3196 2 priority = 5 @200000000

Child PID = 3197 4 priority = -10 @200000000

Child PID = 3196 3 priority = 5 @200000000

Child PID = 3197 5 priority = -10 @200000000

Child PID = 3196 4 priority = 5 @200000000

Child PID = 3197 6 priority = -10 @200000000

Child PID = 3195 1 priority = 10 @200000000

Child PID = 3197 7 priority = -10 @200000000

Child PID = 3196 5 priority = 5 @200000000

Child PID = 3197 8 priority = -10 @200000000

Child PID = 3196 6 priority = 5 @200000000

Child PID = 3197 9 priority = -10 @200000000

Child PID = 3195 2 priority = 10 @200000000

Child PID = 3196 7 priority = 5 @200000000

Child PID = 3195 3 priority = 10 @200000000

Child PID = 3196 8 priority = 5 @200000000

Child PID = 3195 4 priority = 10 @200000000

Child PID = 3196 9 priority = 5 @200000000

Child PID = 3195 5 priority = 10 @200000000

Child PID = 3195 6 priority = 10 @200000000

Child PID = 3195 7 priority = 10 @200000000

Child PID = 3195 8 priority = 10 @200000000

Child PID = 3195 9 priority = 10 @200000000

可以看出各子进程在相同的运算量及调度策略下，优先数小的子进程的printf报告输出相对更靠前，说明其进程推进的比较快，也就是得到了相对优先的调度。

**5) 再次以超级用户权限运行psched，指定3个子进程的优先数为10, 5, 18；调度策略分别是0,0,1**

$ **sudo** ./psched 10 5 18 0 0 1

My child 3205 policy is 0

My child 3206 policy is 0

My child 3207 policy is 1

Child PID = 3207 0 priority = 18 @200000000

Child PID = 3206 0 priority = 5 @200000000

Child PID = 3207 1 priority = 18 @200000000

Child PID = 3207 2 priority = 18 @200000000

Child PID = 3206 1 priority = 5 @200000000

Child PID = 3207 3 priority = 18 @200000000

Child PID = 3206 2 priority = 5 @200000000

Child PID = 3205 0 priority = 10 @200000000

Child PID = 3207 4 priority = 18 @200000000

Child PID = 3207 5 priority = 18 @200000000

Child PID = 3206 3 priority = 5 @200000000

Child PID = 3207 6 priority = 18 @200000000

Child PID = 3206 4 priority = 5 @200000000

Child PID = 3207 7 priority = 18 @200000000

Child PID = 3206 5 priority = 5 @200000000

Child PID = 3207 8 priority = 18 @200000000

Child PID = 3205 1 priority = 10 @200000000

Child PID = 3207 9 priority = 18 @200000000

Child PID = 3206 6 priority = 5 @200000000

Child PID = 3205 2 priority = 10 @200000000

Child PID = 3206 7 priority = 5 @200000000

Child PID = 3205 3 priority = 10 @200000000

Child PID = 3206 8 priority = 5 @200000000

Child PID = 3205 4 priority = 10 @200000000

Child PID = 3206 9 priority = 5 @200000000

Child PID = 3205 5 priority = 10 @200000000

Child PID = 3205 6 priority = 10 @200000000

Child PID = 3205 7 priority = 10 @200000000

Child PID = 3205 8 priority = 10 @200000000

Child PID = 3205 9 priority = 10 @200000000

可以看出虽然3207子进程其优先数最大，但由于其调度策略为先进先出，因此其printf报告输出相对更靠前，说明该子进程推进的比较快，也就是得到了相对优先的调度。

实验四、进程同步实验

4.3 示例实验

以下示例实验程序应能模拟多个生产/消费者在有界缓冲上正确的操作。它利用N 个字节的共享内存作为有界循环缓冲区，利用写一字符模拟放一个产品，利用读一字符模拟消费一个产品。当缓冲区空时消费者应阻塞睡眠，而当缓冲区满时生产者应当阻塞睡眠。一旦缓冲区中有空单元，生产者进程就向空单元中入写字符,并报告写的内容和位置。一旦缓冲区中有未读过的字符，消费者进程就从该单元中读出字符，并报告读取位置。生产者不能向同一单元中连续写两次以上相同的字符，消费者也不能从同一单元中连续读两次以上相同的字符。

**1) 在当前新建文件夹中建立以下名为ipc.h的C程序的头文件，该文件中定义了生产者/消费者共用的IPC 函数的原型和数据结构：**

/\*

\* Filename : ipc.h

\* copyright : (C) 2006 by zhonghonglie, revised 2022.1.19

\* Function : 声明IPC机制的函数原型和全局变量

\*/

#include <stdio.h>

#include <stdlib.h>

#include <unistd.h>

#include <sys/types.h>

#include <sys/ipc.h>

#include <sys/shm.h>

#include <sys/sem.h>

#include <sys/msg.h>

#define BUFSZ 256

//建立或获取ipc的一组函数的原型说明

int get\_ipc\_id(char \*proc\_file, key\_t key);

char \*set\_shm(key\_t shm\_key, int shm\_num, int shm\_flag);

int set\_msq(key\_t msq\_key, int msq\_flag);

int set\_sem(key\_t sem\_key, int sem\_val, int sem\_flag);

int down(int sem\_id);

int up(int sem\_id);

/\*信号灯控制用的共同体\*/

typedef union semuns {

int val;

} Sem\_uns;

/\* 消息结构体\*/

typedef struct msgbuf {

long mtype;

char mtext[1];

} Msg\_buf;

**2) 在当前新建文件夹中建立以下名为ipc.c的C程序，该程序中定义了生产者/消费者共用的IPC函数和全局变量：**

/\*

\* Filename : ipc.c

\* copyright : (C) 2006 by zhonghonglie, revised 2022.1.19

\* Function : 一组建立IPC机制的函数

\*/

#include "ipc.h"

//生产消费者共享缓冲区即其有关的变量

key\_t buff\_key;

int buff\_num;

char \*buff\_ptr;

//生产者放产品位置的共享指针

key\_t pput\_key;

int pput\_num;

int \*pput\_ptr;

//消费者取产品位置的共享指针

key\_t cget\_key;

int cget\_num;

int \*cget\_ptr;

//生产者有关的信号量

key\_t prod\_key;

key\_t pmtx\_key;

int prod\_sem;

int pmtx\_sem;

//消费者有关的信号量

key\_t cons\_key;

key\_t cmtx\_key;

int cons\_sem;

int cmtx\_sem;

int sem\_val;

int sem\_flg;

int shm\_flg;

/\*

\* get\_ipc\_id() 从/proc/sysvipc/文件系统中获取IPC的id号

\* pfile: 对应/proc/sysvipc/目录中的IPC文件分别为

\* msg-消息队列,sem-信号量,shm-共享内存

\* key: 对应要获取的IPC的id号的键值

\*/

int get\_ipc\_id(char \*proc\_file, key\_t key)

{

FILE \*pf;

int i, j;

char line[BUFSZ], colum[BUFSZ];

if((pf = fopen(proc\_file,"r")) == NULL) {

perror("Proc file not open");

exit(EXIT\_FAILURE);

}

fgets(line, BUFSZ, pf);

while(!feof(pf)) {

i = j = 0;

fgets(line, BUFSZ, pf);

while(line[i] == ' ')

i++;

while(line[i] !=' ')

colum[j++] = line[i++];

colum[j] = '\0';

if(atoi(colum) != key)

continue;

j = 0;

while(line[i] == ' ')

i++;

while(line[i] !=' ')

colum[j++] = line[i++];

colum[j] = '\0';

i = atoi(colum);

fclose(pf);

return i;

}

fclose(pf);

return -1;

}

/\*

\* 信号灯上的down/up操作

\* semid:信号灯数组标识符

\* semnum:信号灯数组下标

\* buf:操作信号灯的结构

\*/

int down(int sem\_id)

{

struct sembuf buf;

buf.sem\_op = -1;

buf.sem\_num = 0;

buf.sem\_flg = SEM\_UNDO;

if((semop(sem\_id, &buf, 1)) < 0) {

perror("down error ");

exit(EXIT\_FAILURE);

}

return EXIT\_SUCCESS;

}

int up(int sem\_id)

{

struct sembuf buf;

buf.sem\_op = 1;

buf.sem\_num = 0;

buf.sem\_flg = SEM\_UNDO;

if((semop(sem\_id, &buf, 1)) < 0) {

perror("up error ");

exit(EXIT\_FAILURE);

}

return EXIT\_SUCCESS;

}

/\*

\* set\_sem函数建立一个具有n个信号灯的信号量

\* 如果建立成功，返回一个信号灯数组的标识符sem\_id

\* 输入参数：

\* sem\_key 信号灯数组的键值

\* sem\_val 信号灯数组中信号灯的个数

\* sem\_flag 信号等数组的存取权限

\*/

int set\_sem(key\_t sem\_key, int sem\_val, int sem\_flg)

{

int sem\_id;

Sem\_uns sem\_arg;

//测试由sem\_key标识的信号灯数组是否已经建立

if((sem\_id = get\_ipc\_id("/proc/sysvipc/sem", sem\_key)) < 0) {

//semget新建一个信号灯,其标号返回到sem\_id

if((sem\_id = semget(sem\_key, 1, sem\_flg)) < 0) {

perror("semaphore create error");

exit(EXIT\_FAILURE);

}

//设置信号灯的初值

sem\_arg.val = sem\_val;

if(semctl(sem\_id,0, SETVAL, sem\_arg) < 0) {

perror("semaphore set error");

exit(EXIT\_FAILURE);

}

}

return sem\_id;

}

/\*

\* set\_shm函数建立一个具有n个字节 的共享内存区

\* 如果建立成功，返回 一个指向该内存区首地址的指针shm\_buf

\* 输入参数：

\* shm\_key 共享内存的键值

\* shm\_val 共享内存字节的长度

\* shm\_flag 共享内存的存取权限

\*/

char \* set\_shm(key\_t shm\_key, int shm\_num, int shm\_flg)

{

int i, shm\_id;

char \* shm\_buf;

//测试由shm\_key标识的共享内存区是否已经建立

if((shm\_id = get\_ipc\_id("/proc/sysvipc/shm", shm\_key)) < 0) {

//shmget新建 一个长度为shm\_num字节的共享内存,其标号返回到shm\_id

if((shm\_id = shmget(shm\_key, shm\_num, shm\_flg)) < 0) {

perror("shareMemory set error");

exit(EXIT\_FAILURE);

}

//shmat将由shm\_id标识的共享内存附加给指针shm\_buf

if((shm\_buf = (char \*)shmat(shm\_id, 0, 0)) < (char \*)0) {

perror("get shareMemory error");

exit(EXIT\_FAILURE);

}

for(i = 0; i < shm\_num; i++) shm\_buf[i] = 0; //初始为0

}

//shm\_key 标识的共享内存区已经建立,将由shm\_id 标识的共享内存附加给指针shm\_buf

if((shm\_buf = (char \*)shmat(shm\_id, 0, 0)) < (char \*)0) {

perror("get shareMemory error");

exit(EXIT\_FAILURE);

}

return shm\_buf;

}

/\*

\* set\_msq 函数建立一个消息队列

\* 如果建立成功，返回一个消息队列的标识符msq\_id

\* 输入参数：

\* msq\_key 消息队列的键值

\* msq\_flag 消息队列的存取权限

\*/

int set\_msq(key\_t msq\_key, int msq\_flg)

{

int msq\_id;

//测试由msq\_key标识的消息队列是否已经建立

if((msq\_id = get\_ipc\_id("/proc/sysvipc/msg", msq\_key)) < 0) {

//msgget新建一个消息队列,其标号返回到msq\_id

if((msq\_id = msgget(msq\_key, msq\_flg)) < 0) {

perror("messageQueue set error");

exit(EXIT\_FAILURE);

}

}

return msq\_id;

}

**3) 在当前新文件夹中建立生产者程序producer.c**

/\*

\* Filename : producer.c

\* copyright : (C) 2006 by zhonghonglie, revised 2022.1.19

\* Function : 建立并模拟生产者进程

\*/

#include "ipc.h"

//生产消费者共享缓冲区即其有关的变量

extern key\_t buff\_key;

extern int buff\_num;

extern char \*buff\_ptr;

//生产者放产品位置的共享指针

extern key\_t pput\_key;

extern int pput\_num;

extern int \*pput\_ptr;

//消费者取产品位置的共享指针

extern key\_t cget\_key;

extern int cget\_num;

extern int \*cget\_ptr;

//生产者有关的信号量

extern key\_t prod\_key;

extern key\_t pmtx\_key;

extern int prod\_sem;

extern int pmtx\_sem;

//消费者有关的信号量

extern key\_t cons\_key;

extern key\_t cmtx\_key;

extern int cons\_sem;

extern int cmtx\_sem;

extern int sem\_val;

extern int sem\_flg;

extern int shm\_flg;

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

{

int rate;

//可在在命令行第一参数指定一个进程睡眠秒数，以调解进程执行速度

if(argv[1] != NULL) rate = atoi(argv[1]);

else rate = 3; //不指定为3秒

//共享内存使用的变量

buff\_key = 101; //缓冲区任给的键值

buff\_num = 8; //缓冲区任给的长度

pput\_key = 102; //生产者放产品指针的键值

pput\_num = 1; //指针数

shm\_flg = IPC\_CREAT | 0644; //共享内存读写权限

//获取缓冲区使用的共享内存，buff\_ptr 指向缓冲区首地址

buff\_ptr = (char \*)set\_shm(buff\_key, buff\_num, shm\_flg);

//获取生产者放产品位置指针pput\_ptr

pput\_ptr = (int \*)set\_shm(pput\_key, pput\_num, shm\_flg);

//信号量使用的变量

prod\_key = 201; //生产者同步信号灯键值

pmtx\_key = 202; //生产者互斥信号灯键值

cons\_key = 301; //消费者同步信号灯键值

cmtx\_key = 302; //消费者互斥信号灯键值

sem\_flg = IPC\_CREAT | 0644;

//生产者同步信号灯初值设为缓冲区最大可用量

sem\_val = buff\_num;

//获取生产者同步信号灯，引用标识存prod\_sem

prod\_sem = set\_sem(prod\_key, sem\_val, sem\_flg);

//消费者初始无产品可取，同步信号灯初值设为0

sem\_val = 0;

//获取消费者同步信号灯，引用标识存cons\_sem

cons\_sem = set\_sem(cons\_key, sem\_val, sem\_flg);

//生产者互斥信号灯初值为1

sem\_val = 1;

//获取生产者互斥信号灯，引用标识存pmtx\_sem

pmtx\_sem = set\_sem(pmtx\_key, sem\_val, sem\_flg);

//循环执行模拟生产者不断放产品

while(1) {

//如果缓冲区满则生产者阻塞

down(prod\_sem);

//如果另一生产者正在放产品，本生产者阻塞

down(pmtx\_sem);

//用写一字符的形式模拟生产者放产品，报告本进程号和放入的字符及存放的位置

buff\_ptr[\*pput\_ptr] = 'A' + \*pput\_ptr;

printf("%d producer put: %c to Buffer[%d]\n", getpid(), buff\_ptr[\*pput\_ptr], \*pput\_ptr);

//存放位置循环下移

\*pput\_ptr = (\*pput\_ptr + 1) % buff\_num;

//唤醒阻塞的生产者

up(pmtx\_sem);

//唤醒阻塞的消费者

up(cons\_sem);

sleep(rate);

}

return EXIT\_SUCCESS;

}

**4) 在当前新文件夹中建立消费者程序consumer.c**

/\*

Filename : consumer.c

copyright : (C) by zhanghonglie, revised 2022.1.19

Function : 建立并模拟消费者进程

\*/

#include "ipc.h"

//生产消费者共享缓冲区即其有关的变量

extern key\_t buff\_key;

extern int buff\_num;

extern char \*buff\_ptr;

//生产者放产品位置的共享指针

extern key\_t pput\_key;

extern int pput\_num;

extern int \*pput\_ptr;

//消费者取产品位置的共享指针

extern key\_t cget\_key;

extern int cget\_num;

extern int \*cget\_ptr;

//生产者有关的信号量

extern key\_t prod\_key;

extern key\_t pmtx\_key;

extern int prod\_sem;

extern int pmtx\_sem;

//消费者有关的信号量

extern key\_t cons\_key;

extern key\_t cmtx\_key;

extern int cons\_sem;

extern int cmtx\_sem;

extern int sem\_val;

extern int sem\_flg;

extern int shm\_flg;

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

{

int rate;

//可在在命令行第一参数指定一个进程睡眠秒数，以调解进程执行速度

if(argv[1] != NULL) rate = atoi(argv[1]);

else rate = 3; //不指定为3秒

//共享内存 使用的变量

buff\_key = 101; //缓冲区任给的键值

buff\_num = 8; //缓冲区任给的长度

cget\_key = 103; //消费者取产品指针的键值

cget\_num = 1; //指针数

shm\_flg = IPC\_CREAT | 0644; //共享内存读写权限

//获取缓冲区使用的共享内存，buff\_ptr 指向缓冲区首地址

buff\_ptr = (char \*)set\_shm(buff\_key, buff\_num, shm\_flg);

//获取消费者取产品指针，cget\_ptr 指向索引地址

cget\_ptr = (int \*)set\_shm(cget\_key, cget\_num, shm\_flg);

//信号量使用的变量

prod\_key = 201; //生产者同步信号灯键值

pmtx\_key = 202; //生产者互斥信号灯键值

cons\_key = 301; //消费者同步信号灯键值

cmtx\_key = 302; //消费者互斥信号灯键值

sem\_flg = IPC\_CREAT | 0644; //信号灯操作权限

//生产者同步信号灯初值设为缓冲区最大可用量

sem\_val = buff\_num;

//获取生产者同步信号灯，引用标识存prod\_sem

prod\_sem = set\_sem(prod\_key,sem\_val,sem\_flg);

//消费者初始无产品可取，同步信号灯初值设为0

sem\_val = 0;

//获取消费者同步信号灯，引用标识存cons\_sem

cons\_sem = set\_sem(cons\_key,sem\_val,sem\_flg);

//消费者互斥信号灯初值为1

sem\_val = 1;

//获取消费者互斥信号灯，引用标识存pmtx\_sem

cmtx\_sem = set\_sem(cmtx\_key,sem\_val,sem\_flg);

//循环执行模拟消费者不断取产品

while(1) {

//如果无产品消费者阻塞

down(cons\_sem);

//如果另一消费者正在取产品，本消费者阻塞

down(cmtx\_sem);

//用读一字符的形式模拟消费者取产品，报告本进程号和获取的字符及读取的位置

printf("%d consumer get: %c from Buffer[%d]\n", getpid(), buff\_ptr[\*cget\_ptr], \*cget\_ptr);

//读取位置循环下移

\*cget\_ptr = (\*cget\_ptr + 1) % buff\_num;

//唤醒阻塞的消费者

up(cmtx\_sem);

//唤醒阻塞的生产者

up(prod\_sem);

sleep(rate);

}

return EXIT\_SUCCESS;

}

**5) 在当前文件夹中建立Makefile项目管理文件**

hdrs = ipc.h

opts = -g -c

c\_src = consumer.c ipc.c

c\_obj = consumer.o ipc.o

p\_src = producer.c ipc.c

p\_obj = producer.o ipc.o

all: producer consumer

consumer: $(c\_obj)

gcc $(c\_obj) -o consumer

consumer.o: $(c\_src) $(hdrs)

gcc $(opts) $(c\_src)

producer: $(p\_obj)

gcc $(p\_obj) -o producer

producer.o: $(p\_src) $(hdrs)

gcc $(opts) $(p\_src)

clean:

rm consumer producer \*.o

**6) 使用make命令编译连接生成可执行的生产者、消费者程序**

$ make

gcc -g -c producer.c ipc.c

gcc producer.o ipc.o -o producer

gcc -g -c consumer.c ipc.c

gcc consumer.o ipc.o -o consumer

**7) 在当前终端窗体中启动执行速率为1秒的一个生产者进程**

$ ./producer 1

2509 producer put: A to Buffer[0]

2509 producer put: B to Buffer[1]

2509 producer put: C to Buffer[2]

2509 producer put: D to Buffer[3]

2509 producer put: E to Buffer[4]

2509 producer put: F to Buffer[5]

2509 producer put: G to Buffer[6]

2509 producer put: H to Buffer[7]

可以看到2509号进程在向共享内存中连续写入了8个字符后因为缓冲区满而

阻塞。

**8) 打开另一终端窗体，进入当前工作目录，从中再启动另一执行速率为3的生产**

**者进程：**

$ ./producer 3

可以看到该生产者进程因为缓冲区已满而立即阻塞。

**9) ）再打开另外两个终端窗体，进入当前工作目录，从中启动执行速率为2和4的两**

**个消费者进程：**

$ ./consumer 2

2511 consumer get: A from Buffer[0]

2511 consumer get: B from Buffer[1]

2511 consumer get: D from Buffer[3]

2511 consumer get: E from Buffer[4]

2511 consumer get: G from Buffer[6]

……

$ ./consumer 4

2512 consumer get: C from Buffer[2]

2512 consumer get: F from Buffer[5]

2512 consumer get: A from Buffer[0]

……

在第一个生产者窗体中生产者1被再次唤醒输出:

2509 producer put: A to Buffer[0]

2509 producer put: C to Buffer[2]

2509 producer put: D to Buffer[3]

2509 producer put: F to Buffer[5]

2509 producer put: G to Buffer[6]

2509 producer put: A to Buffer[0]

……

在第二个生产者窗体中生产者2也被再次被唤醒输出：

2510 producer put: B to Buffer[1]

2510 producer put: E to Buffer[4]

2510 producer put: H to Buffer[7]

2510 producer put: C to Buffer[2]

2510 producer put: F to Buffer[5]

……

可以看到由于消费者进程读出了写入缓冲区的字符，生产者重新被唤醒继续向读过的缓冲区单元中同步地写入字符。

请用ctrl+C将两生产者进程打断，观察两消费者进程是否在读空缓冲区后而被阻塞。反之，请用ctrl+C将两消费者进程打断，观察两生产者进程是否在写满缓冲区后而被阻塞。

信号量是保存在Linux内核中的，在多次用ctrl+c退出生产者/消费者，并重启后，有时会导致信号量异常。现象为目测生产者生产的速度，与消费者消费的速度明显不匹配，甚至无生产者时消费者依然能长时间消费，或者无消费者时生产者依然能长时间生产的问题。

此时可先用ctrl+c退出所有的生产者/消费者，再执行下列操作：

$ ipcs -s

------ Semaphore Arrays --------

key semid owner perms nsems

0x000000c9 0 u1 644 1

0x0000012d 32769 u1 644 1

0x000000ca 65538 u1 644 1

0x0000012e 98307 u1 644 1

再用ipcrm命令，删除上面ipcs命令显示的4个key值分别为201(0xc9), 202(0xca), 301(0x12d), 302(0x12e)的信号量。注意这里ipcrm命令的选项是大写的S，对应着删除某个特定key值的信号量。

$ ipcrm -S 201

$ ipcrm -S 202

$ ipcrm -S 301

$ ipcrm -S 302

$ ipcs -s

------ Semaphore Arrays --------

key semid owner perms nsems

这里的显示应该不再有上述4个key值的信号量了。

此时再重启生产者/消费者即可。

实验五、进程互斥实验

5.3 示例实验

我们可以利用上节实验中介绍的IPC机制中的消息队列来实验一下以上使用消息传递算法的读写者问题的解法，看其是否能够满足我们的要求。仍采用共享内存模拟要读写的对象，一写者向共享内存中写入一串字符后，多个读者可同时从共享内存中读出该串字符。

**1）在新建的文件夹中建立以下ipc.h 头文件**

/\*

\* Filename: ipc.h

\* Copyright: (C) 2006 by zhonghonglie, revised 2022.1.19

\* Function: 声明 IPC 机制的函数原型和全局变量

\*/

#include <stdio.h>

#include <stdlib.h>

#include <unistd.h>

#include <sys/types.h>

#include <sys/ipc.h>

#include <sys/shm.h>

#include <sys/sem.h>

#include <sys/msg.h>

#define BUFSZ 256

#define MAXVAL 100

#define STRSIZ 8

#define WRITERQUEST 1 // 写请求标识

#define READERQUEST 2 // 读请求标识

#define FINISHED 3 // 读写完成标识

/\*信号灯控制用的共同体\*/

typedef union semuns {

int val;

} Sem\_uns;

/\* 消息结构体\*/

typedef struct msgbuf {

long mtype;

int mid;

} Msg\_buf;

int get\_ipc\_id(char \*proc\_file, key\_t key);

char \*set\_shm(key\_t shm\_key, int shm\_num, int shm\_flag);

int set\_msq(key\_t msq\_key, int msq\_flag);

int set\_sem(key\_t sem\_key, int sem\_val, int sem\_flag);

int down(int sem\_id);

int up(int sem\_id);

**2）在当前新建文件夹中建立以下名为ipc.c的C程序，该程序中定义了读者/写者/控制者共用的IPC函数**

/\*

\* Filename : ipc.c

\* copyright : (C) 2006 by zhonghonglie, revised 2022.1.19

\* Function : 一组建立IPC机制的函数

\*/

#include "ipc.h"

key\_t buff\_key;

int buff\_num;

char \*buff\_ptr;

int shm\_flg;

int quest\_flg;

key\_t quest\_key;

int quest\_id;

int respond\_flg;

key\_t respond\_key;

int respond\_id;

/\*

\* get\_ipc\_id() 从/proc/sysvipc/文件系统中获取IPC的id号

\* pfile: 对应/proc/sysvipc/目录中的IPC文件分别为

\* msg-消息队列,sem-信号量,shm-共享内存

\* key: 对应要获取的IPC的id号的键值

\*/

int get\_ipc\_id(char \*proc\_file, key\_t key)

{

FILE \*pf;

int i, j;

char line[BUFSZ], colum[BUFSZ];

if((pf = fopen(proc\_file,"r")) == NULL) {

perror("Proc file not open");

exit(EXIT\_FAILURE);

}

fgets(line, BUFSZ, pf);

while(!feof(pf)) {

i = j = 0;

fgets(line, BUFSZ, pf);

while(line[i] == ' ')

i++;

while(line[i] !=' ')

colum[j++] = line[i++];

colum[j] = '\0';

if(atoi(colum) != key)

continue;

j = 0;

while(line[i] == ' ')

i++;

while(line[i] !=' ')

colum[j++] = line[i++];

colum[j] = '\0';

i = atoi(colum);

fclose(pf);

return i;

}

fclose(pf);

return -1;

}

/\*

\* 信号灯上的down/up操作

\* semid:信号灯数组标识符

\* semnum:信号灯数组下标

\* buf:操作信号灯的结构

\*/

int down(int sem\_id)

{

struct sembuf buf;

buf.sem\_op = -1;

buf.sem\_num = 0;

buf.sem\_flg = SEM\_UNDO;

if((semop(sem\_id, &buf, 1)) < 0) {

perror("down error ");

exit(EXIT\_FAILURE);

}

return EXIT\_SUCCESS;

}

int up(int sem\_id)

{

struct sembuf buf;

buf.sem\_op = 1;

buf.sem\_num = 0;

buf.sem\_flg = SEM\_UNDO;

if((semop(sem\_id, &buf, 1)) < 0) {

perror("up error ");

exit(EXIT\_FAILURE);

}

return EXIT\_SUCCESS;

}

/\*

\* set\_sem函数建立一个具有n个信号灯的信号量

\* 如果建立成功，返回一个信号灯数组的标识符sem\_id

\* 输入参数：

\* sem\_key 信号灯数组的键值

\* sem\_val 信号灯数组中信号灯的个数

\* sem\_flag 信号等数组的存取权限

\*/

int set\_sem(key\_t sem\_key, int sem\_val, int sem\_flg)

{

int sem\_id;

Sem\_uns sem\_arg;

//测试由sem\_key标识的信号灯数组是否已经建立

if((sem\_id = get\_ipc\_id("/proc/sysvipc/sem", sem\_key)) < 0) {

//semget新建一个信号灯,其标号返回到sem\_id

if((sem\_id = semget(sem\_key, 1, sem\_flg)) < 0) {

perror("semaphore create error");

exit(EXIT\_FAILURE);

}

//设置信号灯的初值

sem\_arg.val = sem\_val;

if(semctl(sem\_id,0, SETVAL, sem\_arg) < 0) {

perror("semaphore set error");

exit(EXIT\_FAILURE);

}

}

return sem\_id;

}

/\*

\* set\_shm函数建立一个具有n个字节 的共享内存区

\* 如果建立成功，返回 一个指向该内存区首地址的指针shm\_buf

\* 输入参数：

\* shm\_key 共享内存的键值

\* shm\_val 共享内存字节的长度

\* shm\_flag 共享内存的存取权限

\*/

char \* set\_shm(key\_t shm\_key, int shm\_num, int shm\_flg)

{

int i, shm\_id;

char \* shm\_buf;

//测试由shm\_key标识的共享内存区是否已经建立

if((shm\_id = get\_ipc\_id("/proc/sysvipc/shm", shm\_key)) < 0) {

//shmget新建 一个长度为shm\_num字节的共享内存,其标号返回到shm\_id

if((shm\_id = shmget(shm\_key, shm\_num, shm\_flg)) < 0) {

perror("shareMemory set error");

exit(EXIT\_FAILURE);

}

//shmat将由shm\_id标识的共享内存附加给指针shm\_buf

if((shm\_buf = (char \*)shmat(shm\_id, 0, 0)) < (char \*)0) {

perror("get shareMemory error");

exit(EXIT\_FAILURE);

}

for(i = 0; i < shm\_num; i++) shm\_buf[i] = 0; //初始为0

}

//shm\_key 标识的共享内存区已经建立,将由shm\_id 标识的共享内存附加给指针shm\_buf

if((shm\_buf = (char \*)shmat(shm\_id, 0, 0)) < (char \*)0) {

perror("get shareMemory error");

exit(EXIT\_FAILURE);

}

return shm\_buf;

}

/\*

\* set\_msq 函数建立一个消息队列

\* 如果建立成功，返回一个消息队列的标识符msq\_id

\* 输入参数：

\* msq\_key 消息队列的键值

\* msq\_flag 消息队列的存取权限

\*/

int set\_msq(key\_t msq\_key, int msq\_flg)

{

int msq\_id;

//测试由msq\_key标识的消息队列是否已经建立

if((msq\_id = get\_ipc\_id("/proc/sysvipc/msg", msq\_key)) < 0) {

//msgget新建一个消息队列,其标号返回到msq\_id

if((msq\_id = msgget(msq\_key, msq\_flg)) < 0) {

perror("messageQueue set error");

exit(EXIT\_FAILURE);

}

}

return msq\_id;

}

**3) 在当前目录中建立如下的控制者程序control.c**

/\*

\* Filename: control.c

\* Copyright: (C) 2006 by zhonghonglie, revised 2022.1.19

\* Function: 建立并模拟控制者进程

\*/

#include "ipc.h"

extern key\_t buff\_key;

extern int buff\_num;

extern char \*buff\_ptr;

extern int shm\_flg;

extern int quest\_flg;

extern key\_t quest\_key;

extern int quest\_id;

extern int respond\_flg;

extern key\_t respond\_key;

extern int respond\_id;

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

{

int i;

int count = MAXVAL;

Msg\_buf msg\_arg;

int wr\_mid\_next;

// 建立一个共享内存先写入一串 A 字符模拟要读写的内容

buff\_key = 101;

buff\_num = STRSIZ + 1;

shm\_flg = IPC\_CREAT | 0644;

buff\_ptr = (char \*)set\_shm(buff\_key, buff\_num, shm\_flg);

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

buff\_ptr[i] = 'A';

buff\_ptr[i] = '\0';

// 建立一条请求消息队列

quest\_flg = IPC\_CREAT | 0644;

quest\_key = 201;

quest\_id = set\_msq(quest\_key, quest\_flg);

// 建立一条响应消息队列

respond\_flg = IPC\_CREAT | 0644;

respond\_key = 202;

respond\_id = set\_msq(respond\_key, respond\_flg);

// 控制进程准备接收和响应读写者的消息

printf("Wait quest\n");

while(1) {

wr\_mid\_next = 0; // 无需要延迟处理的对写者的响应

// 状态1：初始状态；或空闲状态；或有读者在读，但无写者正在写，也无需要延迟处理的对写者的响应

if(count > 0) {

// 以非阻塞方式接收消息

if(msgrcv(quest\_id, &msg\_arg, sizeof(msg\_arg.mid), FINISHED, IPC\_NOWAIT) >= 0) {

// 有读者完成

count++;

printf("%d reader finished\n", msg\_arg.mid);

}

else if(msgrcv(quest\_id, &msg\_arg, sizeof(msg\_arg.mid), READERQUEST, IPC\_NOWAIT) >= 0) {

// 有读者请求, 允许读者读

count--;

msg\_arg.mtype = msg\_arg.mid;

msgsnd(respond\_id, &msg\_arg, sizeof(msg\_arg.mid), 0);

printf("%d quest read\n", msg\_arg.mid);

}

else if(msgrcv(quest\_id, &msg\_arg, sizeof(msg\_arg.mid), WRITERQUEST, IPC\_NOWAIT) >= 0) {

// 有写者请求, 看是否立即允许写者写，还是需要等读者(们)读完再写

if(count == MAXVAL) { // 当前无读者，立即响应写者

msg\_arg.mtype = msg\_arg.mid;

msgsnd(respond\_id, &msg\_arg, sizeof(msg\_arg.mid), 0);

printf("%d quest write\n", msg\_arg.mid);

}

else // 当前有读者，延迟对写者的响应

wr\_mid\_next = msg\_arg.mid;

count -= MAXVAL;

}

} // if(count > 0)

// 状态2：当 count == 0 时说明1个写者正在写, 我们等待写完成

if(count == 0) {

msgrcv(quest\_id, &msg\_arg, sizeof(msg\_arg.mid), FINISHED, 0); // 以阻塞方式接收消息

count = MAXVAL;

printf("%d write finished\n", msg\_arg.mid);

}

// 状态3：当 count < 0 时说明有读者(们)正在读, 且有1个写者的请求已收到但尚未响应

if(count < 0) {

// 等待读者(们)读完

while(count < 0) {

msgrcv(quest\_id, &msg\_arg, sizeof(msg\_arg.mid), FINISHED, 0); // 以阻塞方式接收消息

count++;

printf("%d reader finish\n", msg\_arg.mid);

}

// count == 0

if(wr\_mid\_next) { // 有需要延迟处理的对写者的响应

msg\_arg.mtype = msg\_arg.mid = wr\_mid\_next;

msgsnd(respond\_id, &msg\_arg, sizeof(msg\_arg.mid), 0);

printf("%d quest write\n", wr\_mid\_next);

}

}

}

return EXIT\_SUCCESS;

}

**4）在当前目录中建立如下的读者程序reader.c**

/\*

\* Filename: reader.c

\* Copyright: (C) 2006 by zhonghonglie, revised 2022.1.19

\* Function: 建立并模拟读者进程

\*/

#include "ipc.h"

extern key\_t buff\_key;

extern int buff\_num;

extern char \*buff\_ptr;

extern int shm\_flg;

extern int quest\_flg;

extern key\_t quest\_key;

extern int quest\_id;

extern int respond\_flg;

extern key\_t respond\_key;

extern int respond\_id;

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

{

int i = 0;

int t\_reading = 3, t\_break = 0;

Msg\_buf msg\_arg;

// 可在在命令行第1个参数指定一个模拟读过程的秒数, 第2个参数指定一个模拟离开的秒数

if(argc > 1) {

t\_reading = atoi(argv[1]);

if(argc > 2)

t\_break = atoi(argv[2]);

}

//附加一个要读内容的共享内存

buff\_key = 101;

buff\_num = STRSIZ + 1;

shm\_flg = IPC\_CREAT | 0644;

buff\_ptr = (char \*)set\_shm(buff\_key, buff\_num, shm\_flg);

//联系一个请求消息队列

quest\_flg = IPC\_CREAT | 0644;

quest\_key = 201;

quest\_id = set\_msq(quest\_key, quest\_flg);

//联系一个响应消息队列

respond\_flg = IPC\_CREAT | 0644;

respond\_key = 202;

respond\_id = set\_msq(respond\_key, respond\_flg);

//循环请求读

msg\_arg.mid = getpid();

while(1) {

// 发读请求消息

msg\_arg.mtype = READERQUEST;

msgsnd(quest\_id, &msg\_arg, sizeof(msg\_arg.mid), 0);

printf("%d reader quest %d\n", msg\_arg.mid, ++i);

// 等待允许读消息

msgrcv(respond\_id, &msg\_arg, sizeof(msg\_arg.mid), msg\_arg.mid, 0);

printf("%d reading: %s\n", msg\_arg.mid, buff\_ptr);

sleep(t\_reading);

// 发读完成消息

msg\_arg.mtype = FINISHED;

msgsnd(quest\_id, &msg\_arg, sizeof(msg\_arg.mid), 0);

if(t\_break)

sleep(t\_break);

}

return EXIT\_SUCCESS;

}

**5）在当前目录中建立如下的写者程序writer.c**

/\*

\* Filename: writer.c

\* Copyright: (C) 2006 by zhonghonglie, revised 2022.1.19

\* Function: 建立并模拟写者进程

\*/

#include "ipc.h"

extern key\_t buff\_key;

extern int buff\_num;

extern char \*buff\_ptr;

extern int shm\_flg;

extern int quest\_flg;

extern key\_t quest\_key;

extern int quest\_id;

extern int respond\_flg;

extern key\_t respond\_key;

extern int respond\_id;

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

{

int i, j = 0, k = 0;

int t\_writing = 3, t\_break = 0;

Msg\_buf msg\_arg;

// 可在在命令行第1个参数指定一个模拟写过程的秒数, 第2个参数指定一个模拟离开的秒数

if(argc > 1) {

t\_writing = atoi(argv[1]);

if(argc > 2)

t\_break = atoi(argv[2]);

}

//附加一个要读内容的共享内存

buff\_key = 101;

buff\_num = STRSIZ + 1;

shm\_flg = IPC\_CREAT | 0644;

buff\_ptr = (char \*)set\_shm(buff\_key, buff\_num, shm\_flg);

//联系一个请求消息队列

quest\_flg = IPC\_CREAT | 0644;

quest\_key = 201;

quest\_id = set\_msq(quest\_key, quest\_flg);

//联系一个响应消息队列

respond\_flg = IPC\_CREAT | 0644;

respond\_key = 202;

respond\_id = set\_msq(respond\_key, respond\_flg);

//循环请求写

msg\_arg.mid = getpid();

while(1) {

//发写请求消息

msg\_arg.mtype = WRITERQUEST;

msgsnd(quest\_id, &msg\_arg, sizeof(msg\_arg.mid), 0);

printf("%d writer quest %d\n", msg\_arg.mid, ++k);

//等待允许写消息

msgrcv(respond\_id, &msg\_arg, sizeof(msg\_arg.mid), msg\_arg.mid, 0);

//写入 STRSIZ 个相同的字符

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

buff\_ptr[i] = 'A' + j;

j = (j + 1) % STRSIZ; //按 STRSIZ 循环变换字符

printf("%d writing: %s\n", msg\_arg.mid, buff\_ptr);

sleep(t\_writing);

//发写完成消息

msg\_arg.mtype = FINISHED;

msgsnd(quest\_id, &msg\_arg, sizeof(msg\_arg.mid), 0);

if(t\_break)

sleep(t\_break);

}

return EXIT\_SUCCESS;

}

**6) 在当前目录中建立如下Makefile文件**

hdrs = ipc.h

c\_src = control.c ipc.c

c\_obj = control.o ipc.o

r\_src = reader.c ipc.c

r\_obj = reader.o ipc.o

w\_src = writer.c ipc.c

w\_obj = writer.o ipc.o

opts = -g -c

all: control reader writer

control: $(c\_obj)

gcc $(c\_obj) -o control

control.o: $(c\_src) $(hdrs)

gcc $(opts) $(c\_src)

reader: $(r\_obj)

gcc $(r\_obj) -o reader

reader.o: $(r\_src) $(hdrs)

gcc $(opts) $(r\_src)

writer: $(w\_obj)

gcc $(w\_obj) -o writer

writer.o: $(w\_src) $(hdrs)

gcc $(opts) $(w\_src)

clean:

rm control reader writer \*.o

**7）在当前目录中执行make命令编译连接，生成读写者，控制者程序：**

$make

gcc -g -c control.c ipc.c

gcc control.o ipc.o -o control

gcc -g -c reader.c ipc.c

gcc reader.o ipc.o -o reader

gcc -g -c writer.c ipc.c

gcc writer.o ipc.o -o writer

$

**8）可打开四个以上的终端模命令窗体，都将进入当前工作目录。先在一窗体中启动./control程序：**

$ ./control

Wait quest

现在控制进程已经在等待读写者的请求。

**9）再在另两不同的窗体中启动两个读者，一个让它以1秒的延迟快一些读，一个让它以10秒的延迟慢一些读：**

$ ./reader 10

3903 reader quest

3903 reading: AAAAAAAA

......

$ ./reader 1

3904 reader quest

3904 reading: AAAAAAAA

3904 reader quest

3904 reading: AAAAAAAA

3904 reader quest

3904 reading: AAAAAAAA

3904 reader quest

......

现在可以看到控制进程开始响应读者请求，让多个读者同时进入临界区读：

Wait quest

3903 quest read

3904 quest read

3904 reader finished

3903 reader finished

3904 quest read

3904 reader finished

......

**10）再在另一终端窗体中启动一个延迟时间为8秒的写者进程**：

$ ./writer 8

3906 writer quest

3906 writing: AAAAAAAA

3906 writer quest

3906 writing: BBBBBBBB

3906 writer quest

3906 writing: CCCCCCCC

......

此时可以看到控制进程在最后一个读者读完后首先响应写者请求：

3904 reader finish

3903 reader finish

3906 quest write

3906 write finished

......

在写者写完后两个读者也同时读到了新写入的内容：

3903 reader quest

3903 reading: BBBBBBBB

3903 reader quest

3903 reading: CCCCCCCC

......

3904 reading: BBBBBBBB

3904 reader quest

3904 reading: CCCCCCCC

......

请仔细观察各个读者、写者的执行顺序：可以看出在写者写时不会有读者进入，在有读者读时不会有写者进入，但一旦读者全部退出，写者会首先进入。分析以上输出可以看出该算法实现了我们要求的读写者问题的功能。

注意重新启动前，请先用ctrl+c退出所有的reader、writer和control。

然后重启control，这时(还没有任何reader或writer启动)若control显示的除了“Wait quest”外，还有“quest read”或“quest write”，则需要重新用ctrl+c退出control，然后再次启动control。

在成功启动control后(control启动后仅显示了“Wait quest”)，就可以以任何次序启动多个reader及writer了。

与实验四中的ipcs及ipcrm命令类似，必要时可用下面3条命令分别显示和删除内核中留存的消息队列，key值分别为201(0xc9)和202(0xca)。

$ ipcs -q

------ Message Queues --------

key msqid owner perms used-bytes messages

0x000000c9 65536 u1 644 16 4

0x000000ca 98305 u1 644 12 3

$ ipcrm -Q 201

$ ipcrm -Q 202

**11）请按与以上不同的启动顺序、不同的延迟时间，启动更多的读写者。观察和分析是否仍能满足我们要求的读写者问题的功能。**

**12) 请修改以上程序，制造一个读者或写者的饥俄现象。观察什么是饥俄现象，说明为什么会发生这种现象。**

实验七、内存页面置换算法实验

7.3 示例实验

**1）在新建文件夹中建立以下vmrp.h文件：**

/\*

\* Filename : vmrp.h

\* copyright : (C) 2006 by zhonghonglie

\* Function : 声明虚拟内存页置换类

\*/

using namespace std;

#include <iostream>

#include <iomanip>

#include <malloc.h>

class Replace{

public:

Replace();

~Replace();

void InitSpace(char \* MethodName); //初始化页号记录

void Report(void); // 报告算法执行情况

void Fifo(void); //先进先出算法

void Lru(void); //最近最旧未用算法

void Clock(void); //时钟(二次机会）置换算法

void Eclock(void); //增强二次机会置换算法

void Lfu(void); //最不经常使用置换算法

void Mfu(void); //最经常使用置换算法

private:

int \* ReferencePage ; //存放要访问到的页号

int \* EliminatePage ; //存放淘汰页号

int \* PageFrames ; //存放当前正在实存中的页号

int PageNumber; //访问页数

int FrameNumber; //实存帧数

int FaultNumber; //失败页数

};

**2）在新建文件夹中建立以下vmrp.cc文件：**

/\*

\* Filename : vmrp.cc

\* copyright : (C) 2006 by zhonghonglie

\* Function : 模拟虚拟内存页置换算法的程序

\*/

#include "vmrp.h"

Replace::Replace()

{

int i;

//设定总得访问页数,并分配相应的引用页号和淘汰页号记录数组空间

cout << "Please input page numbers :";

cin >> PageNumber;

ReferencePage = new int[sizeof(int) \* PageNumber];

EliminatePage = new int[sizeof(int) \* PageNumber];

//输入引用页号序列(页面走向),初始化引用页数组

cout << "Please input reference page string :";

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

cin >> ReferencePage[i]; //引用页暂存引用数组

//设定内存实页数(帧数),并分配相应的实页号记录数组空间(页号栈)

cout << "Please input page frames :";

cin >> FrameNumber;

PageFrames = new int[sizeof(int) \* FrameNumber];

}

Replace::~Replace(){

}

void Replace::InitSpace(char \* MethodName)

{

int i;

cout << endl << MethodName << endl;

FaultNumber = 0;

//引用还未开始,-1表示无引用页

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

EliminatePage[i] = -1;

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

PageFrames[i] = -1;

}

//分析统计选择的算法对于当前输入的页面走向的性能

void Replace::Report(void)

{

//报告淘汰页顺序

cout << endl << "Eliminate page:";

for(int i=0; EliminatePage[i]!=-1; i++)

cout << EliminatePage[i] << " ";

//报告缺页数和缺页率

cout << endl << "Number of page faults = " << FaultNumber << endl;

cout << setw(6) << setprecision(3) ;

cout << "Rate of page faults = " << 100\*(float)FaultNumber/(float)PageNumber <<

"%" <<endl;

}

//最近最旧未用置换算法

void Replace::Lru(void)

{

int i,j,k,l,next;

InitSpace("LRU");

//循环装入引用页

for(k = 0, l = 0; k < PageNumber; k++) {

next = ReferencePage[k];

//检测引用页当前是否已在实存

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

if(next == PageFrames[i]) {

//引用页已在实存将其调整到页记录栈顶

next = PageFrames[i];

for(j = i; j > 0; j--)

PageFrames[j] = PageFrames[j-1];

PageFrames[0]=next;

break;

}

}

if(PageFrames[0] == next) {

//如果引用页已放栈顶，则为不缺页，报告当前内存页号

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

if(PageFrames[j]>=0)

cout << PageFrames[j] << " ";

cout << endl;

continue; //继续装入下一页

}

else

// 如果引用页还未放栈顶，则为缺页，缺页数加1

FaultNumber++;

//栈底页号记入淘汰页数组中

EliminatePage[l] = PageFrames[FrameNumber-1];

//向下压栈

for(j = FrameNumber - 1; j > 0; j--)

PageFrames[j] = PageFrames[j - 1];

PageFrames[0]=next; //引用页放栈顶

//报告当前实存中页号

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

if(PageFrames[j] >= 0)

cout << PageFrames[j] << " ";

//报告当前淘汰的页号

if(EliminatePage[l] >= 0)

cout << "->" << EliminatePage[l++] << endl;

else

cout << endl;

}

//分析统计选择的算法对于当前引用的页面走向的性能

Report();

}

//先进先出置换算法

void Replace::Fifo(void)

{

int i,j,k,l,next;

InitSpace("FIFO");

//循环装入引用页

for(k = 0,j = l = 0; k < PageNumber; k++) {

next = ReferencePage[k];

//如果引用页已在实存中，报告实存页号

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

if(next == PageFrames[i])

break;

if(i < FrameNumber) {

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

if(PageFrames[i] >= 0)

cout << PageFrames[i] << " ";

cout << endl;

continue; // 继续引用下一页

}

//引用页不在实存中，缺页数加1

FaultNumber++;

EliminatePage[l] = PageFrames[j]; //最先入页号记入淘汰页数组

PageFrames[j] = next; //引用页号放最先入页号处

j = (j + 1) % FrameNumber; //最先入页号循环下移

//报告当前实存页号和淘汰页号

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

if(PageFrames[i] >= 0)

cout << PageFrames[i] << " ";

if(EliminatePage[l] >= 0)

cout << "->" << EliminatePage[l++] << endl;

else

cout << endl;

}

//分析统计选择的算法对于当前引用的页面走向的性能

Report();

}

//未实现的其他页置换算法入口

void Replace::Clock(void)

{

}

void Replace::Eclock (void)

{

}

void Replace::Lfu(void)

{

}

void Replace::Mfu(void)

{

}

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

{

Replace \* vmrp = new Replace();

vmrp->Fifo();

vmrp->Lru();

return 0;

}

**3）在新建文件夹中建立以下Makefile文件**

head = vmrp.h

srcs = vmrp.cc

objs = vmrp.o

opts = -w -g -c

all: vmrp

vmrp: $(objs)

g++ $(objs) -o vmrp

vmrp.o: $(srcs) $(head)

g++ $(opts) $(srcs)

clean:

rm vmrp \*.o

**4）执行make命令编译连接，生成可执行文件vmrp**

$make

g++ -g -c vmrp.cc vmrp.h

g++ vmrp.o -o vmrp

**5）执行vmrp命令，输入引用页数为12，引用串为Belady串，内存页帧数为3**

$ ./vmrp

Please input reference page numbers :12

Please input reference page string :1 2 3 4 1 2 5 1 2 3 4 5

Please input page frames :3

FIFO

1

1 2

1 2 3

4 2 3 ->1

4 1 3 ->2

4 1 2 ->3

5 1 2 ->4

5 1 2

5 1 2

5 3 2 ->1

5 3 4 ->2

5 3 4

Eliminate page:1 2 3 4 1 2

Number of page faults = 9

Rate of page faults = 75%

LRU

1

2 1

3 2 1

4 3 2 ->1

1 4 3 ->2

2 1 4 ->3

5 2 1 ->4

1 5 2

2 1 5

3 2 1 ->5

4 3 2 ->1

5 4 3 ->2

Eliminate page:1 2 3 4 5 1 2

Number of page faults = 10

Rate of page faults = 83.3%

以上输出报告了FIFO和LRU两种算法的页置换情况。其中每一行数字为当前实存中的页号，->右边的数字表示当前被淘汰的页号。每种算法最后3 行输出为：依次淘汰页号，缺页数，页出错率。

**6）再次执行vmrp命令，仍然输入Belady串，仅将页帧数改为4**

$ ./vmrp

Please input reference page numbers :12

Please input reference page string :1 2 3 4 1 2 5 1 2 3 4 5

Please input page frames :4

FIFO

1

1 2

1 2 3

1 2 3 4

1 2 3 4

1 2 3 4

5 2 3 4 ->1

5 1 3 4 ->2

5 1 2 4 ->3

5 1 2 3 ->4

4 1 2 3 ->5

4 5 2 3 ->1

Eliminate page:1 2 3 4 5 1

Number of page faults = 10

Rate of page faults = 83.3%

LRU

1

2 1

3 2 1

4 3 2 1

1 4 3 2

2 1 4 3

5 2 1 4 ->3

1 5 2 4

2 1 5 4

3 2 1 5 ->4

4 3 2 1 ->5

5 4 3 2 ->1

Eliminate page:3 4 5 1

Number of page faults = 8

Rate of page faults = 66.7%

从以上输出中可以看出FIFO 置换算法的Belady 异常现象，即在相同的引用串下，当内存页帧数从3 帧增加到4 帧，页出错率反而从75％增加到了83.3%。而在相同的情况下LUR置换算法无此异常现象。