**#include <stdio.h>**

**#include <signal.h>**

**#include <unistd.h>**

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

**int wait\_flag;**

**void stop( );**

**main( ) {**

**int pid1, pid2; // 定义两个进程号变量**

**signal(3,stop); // 或者 signal(14,stop);**

**while((pid1 = fork( )) == -1); // 若创建子进程1不成功,则空循环**

**if(pid1 > 0) { // 子进程创建成功,pid1为进程号**

**while((pid2 = fork( )) == -1); // 创建子进程2**

**if(pid2 > 0) {**

**wait\_flag = 1;**

**sleep(5); // 父进程等待5秒**

**kill(pid1,16); // 杀死进程1**

**kill(pid2,17); // 杀死进程2**

**wait(0); // 等待第1个子进程1结束的信号**

**wait(0); // 等待第2个子进程2结束的信号**

**printf("\n Parent process is killed !!\n");**

**exit(0); // 父进程结束**

**}**

**else {**

**wait\_flag = 1;**

**signal(17,stop); // 等待进程2被杀死的中断号17**

**printf("\n Child process 2 is killed by parent !!\n");**

**exit(0);**

**}**

**}**

**else {**

**wait\_flag = 1;**

**signal(16,stop); // 等待进程1被杀死的中断号16**

**printf("\n Child process 1 is killed by parent !!\n");**

**exit(0);**

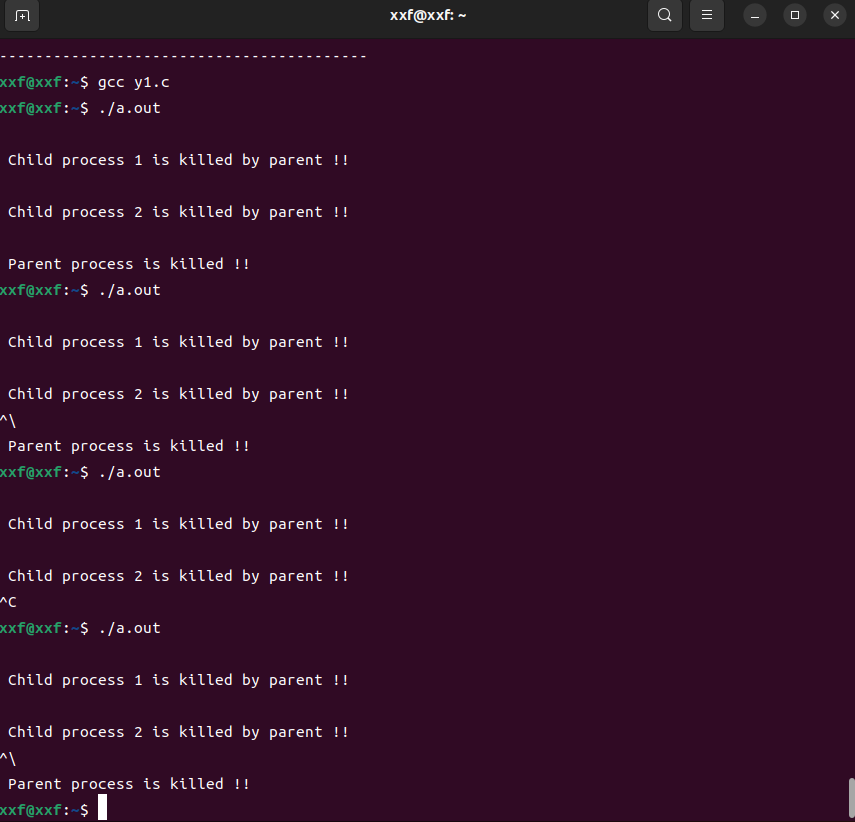
**}**

**}**

**void stop( ) {**

**wait\_flag = 0;**

**}**



猜想：本程序应该开始不显示任何文字，等待键盘按下Delete/Quit键后显示“Child process 1 is killed by parent !! Child process 2 is killed by parent !!”，五秒之后显示“Parent process is killed !!”。

按下Delete/Quit键后程序直接输出“Child process 1 is killed by parent !! Child process 2 is killed by parent !!”，五秒之后显示“Parent process is killed !!”。键盘的输入不起作用，这是因为代码里signal函数并没有对输入的中断进行判断。

多次运行程序，发现子进程被杀死的顺序不总是一样的。这是因为两个子进程同时运行，并没有先后的区别。

在stop函数中将wait\_flag置为0，在kill函数前面加上while(wait\_flag==1)，这样只有在接收到中断信号后才会跳转。

#include <stdio.h>

#include <signal.h>

#include <unistd.h>

#include <sys/types.h>

int wait\_flag;

void stop(int signum);

int main(){

//定义两个进程号变量

int pid1, pid2;

alarm(1);

//signal(SIGINT,stop);

//signal(SIGQUIT,stop);

signal(SIGALRM,stop);

while((pid1 = fork( )) == -1);

//子进程创建成功,pid1为进程号

//在父进程里

if(pid1 > 0) {

//创建子进程2

while((pid2 = fork( )) == -1);

//父进程

if(pid2 > 0) {

wait\_flag = 1;

// 父进程等待5秒

//sleep(5);

//收到中断信号后stop函数中将wait\_flag置为0，再杀死子进程

//signal(SIGALRM,stop);

while(wait\_flag == 1);

// 杀死进程1发中断号16

//kill的作用不是使进程结束，而是发送信号让子进程自己结束。

kill(pid1,16);

// 杀死进程2发中断号17

kill(pid2,17);

// 等待第1个子进程1结束的信号

wait(0);

// 等待第2个子进程2结束的信号

wait(0);

printf("\n Parent process is killed !!\n");

printf("\n");

// 父进程结束

exit(0);

}

//在pid2里

else {

wait\_flag = 1;

// 等待进程2被杀死的中断号17

signal(17,stop);

while(wait\_flag==1);

printf("\n Child process 2 is killed by parent !!\n");

exit(0);

}

}

//pid1

else {

wait\_flag = 1;

// 等待进程1被杀死的中断号16

signal(16,stop);

while(wait\_flag==1);

printf("\n Child process 1 is killed by parent !!\n");

exit(0);

}

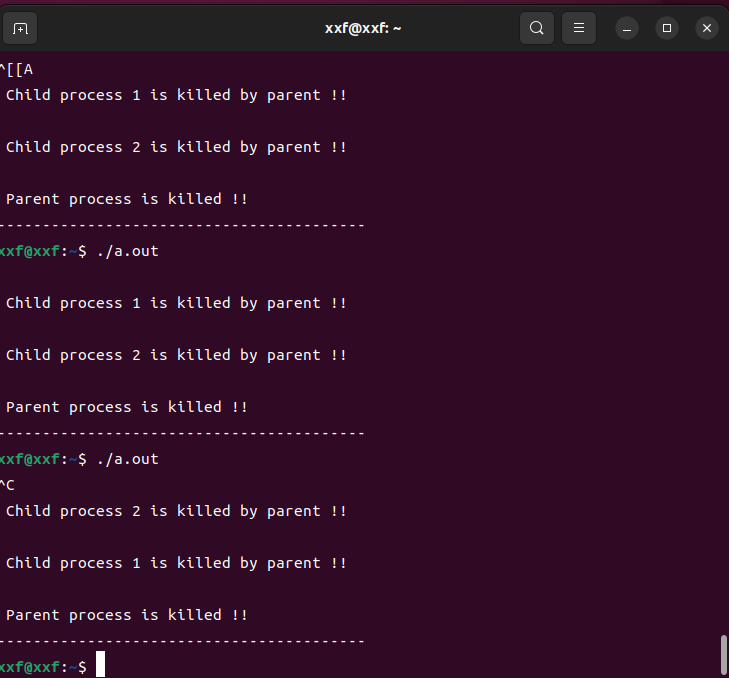
}

void stop(int signum) {

wait\_flag = 0;

printf("\n %d stop test \n",signum);

}



kill命令在程序中使用了几次？每次的作用是什么？执行后的现象是什么？

kill命令使用了两次，分别给子进程1和子进程2发送16，17信号，顺序不一定相同，但是子进程1先输出的概率大。通过将kill命令插入wait命令中间可以控制子进程执行顺序，因为父进程在第一个等待结束后才会发送下一个kill信号。

使用kill命令后，子进程接收到kill命令并调用stop函数，stop函数将wait\_flag置为0，输出被杀死的信号并结束。

使用kill命令可以在进程的外部杀死进程。进程怎样能主动退出？这两种退出方式哪种更好一些？

进程调用return函数和exit函数都可以主动退出，而kill是强制退出。主动退出比较好，如果在某个子进程退出前父进程被kill强制退出，则子进程会被init进程接管；如果用kill命令杀死某个子进程而其父进程没有调用wait函数等待，则该子进程为处于僵死状态占用资源。

1. 管道通信：

**#include <unistd.h>**

**#include <signal.h>**

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

**#include <stdlib.h>**

**#include <stdio.h>**

**int pid1,pid2; // 定义两个进程变量**

**Int main( ) {**

**int fd[2];**

**char OutPipe[100],InPipe[100]; // 定义两个字符数组**

**pipe(fd); // 创建管道**

**while((pid1 = fork( )) == -1); // 如果进程1创建不成功,则空循环**

**if(pid1 == 0) { // 如果子进程1创建成功,pid1为进程号**

**lockf(fd[1],1,0); // 锁定管道**

**sprintf(OutPipe,"\n Child process 1 is sending message!\n"); // 给Outpipe赋值**

**write(fd[1],OutPipe,50); // 向管道写入数据**

**sleep(5); // 等待读进程读出数据**

**lockf(fd[1],0,0); // 解除管道的锁定**

**exit(0); // 结束进程1**

**}**

**else {**

**while((pid2 = fork()) == -1); // 若进程2创建不成功,则空循环**

**if(pid2 == 0) {**

**lockf(fd[1],1,0);**

**sprintf(OutPipe,"\n Child process 2 is sending message!\n");**

**write(fd[1],OutPipe,50);**

**sleep(5);**

**lockf(fd[1],0,0);**

**exit(0);**

**}**

**else {**

**wait(0); // 等待子进程1 结束**

**read(fd[0],InPipe,50); // 从管道中读出数据**

**printf("%s\n",InPipe); // 显示读出的数据**

**wait(0); // 等待子进程2 结束**

**read(fd[0],InPipe,50);**

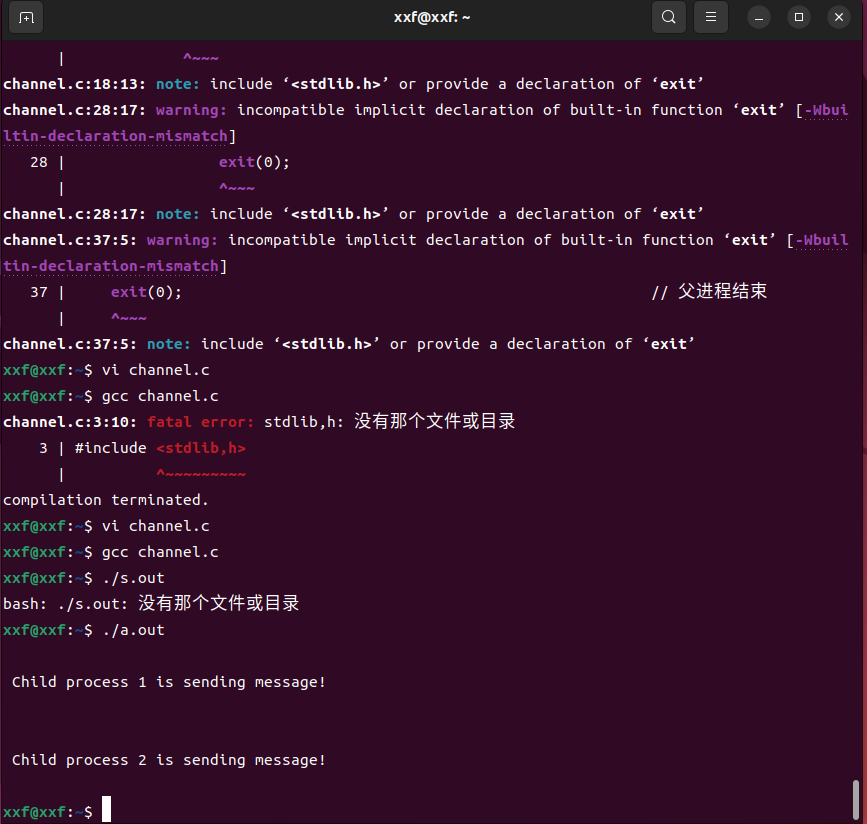
**printf("%s\n",InPipe);**

**exit(0); // 父进程结束**

**}**

**}**

**}**



先猜想一下这个程序的运行结果。

屏幕会输出200个1和200个2.

实际的结果什么样？有什么特点？试对产生该现象的原因进行分析。

多次实验发现有时先输出200个1，有时输出200个2.这是因为子进程1和子进程2执行的先后顺序是不一定的，如果进程1先执行则先输入1；如果进程2先执行则先输入2。但是并没有交错输出1或2，这是因为子进程在写入前线上了锁，保证只有一个进程在向管道输入数据。

实验中管道通信是怎样实现同步与互斥的？如果不控制同步与互斥会发生什么后果？

通过使用函数lockf来锁住管道的写端口。如果不加锁会交错输出1和2

#include<stdio.h>

#include<sys/types.h>

#include<signal.h>

#include<unistd.h>

#include<stdlib.h>

#include<wait.h>

int pid1,pid2; // 定义两个进程变量

int main( ) {

int fd[2];

char InPipe[1000]; // 定义读缓冲区

char c1='1',c2='2';

pipe(fd); // 创建管道

while((pid1 = fork( )) == -1); // 如果进程1创建不成功,则空循环

if(pid1 == 0) { // 如果子进程1创建成功,pid1为进程号

//在子进程1中

//补充； // 锁定管道

lockf(fd[1],1,0);

//补充; // 分200次每次向管道写入字符’1’

for(int i = 0;i<200;i++)

{

//sprintf(InPipe,1);

write(fd[1],&c1,1);

}

sleep(5); // 等待读进程读出数据

lockf(fd[1],0,0);

//补充; // 解除管道的锁定

exit(0); // 结束进程1

}

else {

while((pid2 = fork()) == -1); // 若进程2创建不成功,则空循环

if(pid2 == 0) {

//在子进程2中

lockf(fd[1],1,0);

for(int i = 0;i<200;i++)

{

write(fd[1],&c2,1);

}

//补充; // 分200次每次向管道写入字符’2’

sleep(5);

lockf(fd[1],0,0);

exit(0);

}

else {

//在父进程中

wait(0);

//补充; // 等待子进程1 结束

wait(0); // 等待子进程2 结束

//补充; // 从管道中读出400个字符

lockf(fd[0],1,0);

read(fd[0], InPipe, 400);

InPipe[400] = '\0';

lockf(fd[0],1,0);

//补充; // 加字符串结束符

printf("%s\n",InPipe); // 显示读出的数据

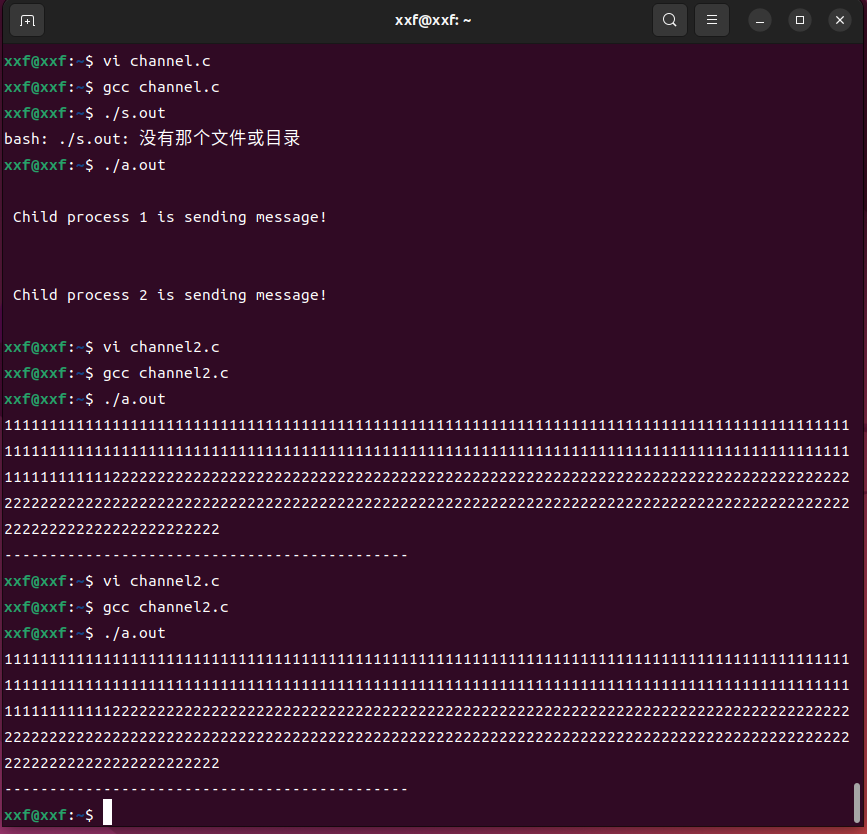
printf("---------------------------------------------\n");

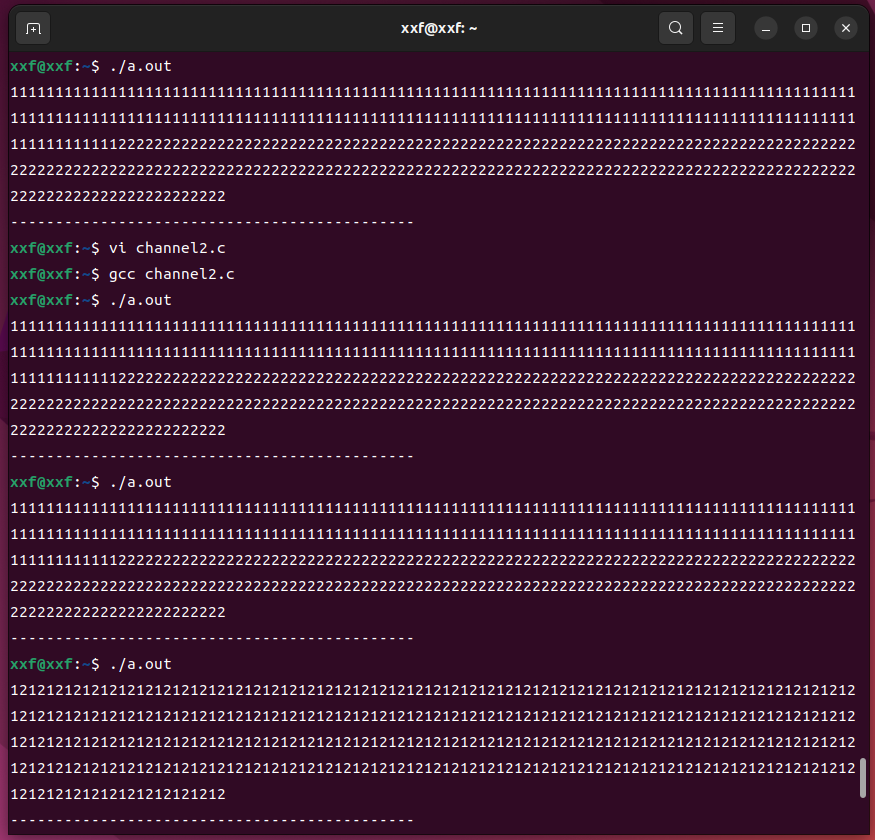
exit(0); // 父进程结束

}

}

}





3.

#include<stdio.h>

#include<stdlib.h>

#define AP 10

#define PP 3

#define TOTAL\_INSTRUCTION 20

//队列数据结构

int Queue[PP+1]={};

int head = 0;

int tail = 0;

//特殊的栈

int Stack[PP] = {};

int top = PP-1;

int bottom = PP-1;

//用于存储中间结果

int temp[PP][TOTAL\_INSTRUCTION];

//检查pagecontrol是否还有空位

int isConEmpty(int\* first\_empty,int pagecontrol[],int control\_num)

{

int flag = 0;

for(int iter = 0;iter<control\_num;iter++)

{

if(pagecontrol[iter]==-1)

{

flag = 1;

\*first\_empty = iter;

break;

}

}

return flag;

}

/\*

\* FIFO算法

\* page[cur]是当前需要调入的页面

\*/

void FIFO(int curpage,int page[],int pagecontrol[],int control\_num)

{

int first\_empty = -1;

//如果pagecontrol中有空位，则直接放入空位

if(isConEmpty(&first\_empty,pagecontrol,control\_num))

{

pagecontrol[first\_empty]=curpage;

//该页面已在内存中

page[curpage] = 1;

//队列记录的是最先放入数据的位置而不是页面号

Queue[tail]=first\_empty;

if((tail+1)%(PP+1) != head)

{

tail = (tail + 1)%(PP+1);

}

else

{

//根据队列的设置不会出现队列满的情况

printf("队列已满！\n");

exit(1);

}

}

//如果没有空位，则替换最先

else

{

page[pagecontrol[Queue[head]]]=0;

pagecontrol[Queue[head]]=curpage;

page[curpage] = 1;

Queue[tail]=Queue[head];

if((head+1)%(PP+1)!=tail)

{

head = (head+1)%(PP+1);

}

else

{

//按照队列的设置不会出现队列空

printf("队列空！\n");

exit(1);

}

if((tail+1)%(PP+1) != head)

{

tail = (tail + 1)%(PP+1);

}

else

{

//根据队列的设置不会出现队列满的情况

printf("队列已满！\n");

exit(1);

}

}

}

//LRU算法

void LRU(int curpage,int page[],int pagecontrol[],int control\_num)

{

int first\_empty = -1;

//先检查在不在pagecontrol里面

for(int iter = 0; iter<PP;iter++)

{

if(pagecontrol[iter]==curpage)

{

int iter2;

for(iter2 = 0;iter2<PP;iter2++)

{

if(Stack[iter2]==iter)

break;

}

for(;iter2<PP-1;iter2++)

{

Stack[iter2]=Stack[iter2+1];

}

Stack[top]=iter;

return;

}

}

//如果pagecontrol中有空位，则直接放入空位

if(isConEmpty(&first\_empty,pagecontrol,control\_num))

{

pagecontrol[first\_empty]=curpage;

//该页面已在内存中

page[curpage] = 1;

//链表记录的是最先放入数据的位置而不是页面号

for(int iter = 0;iter<PP-1;iter++)

{

Stack[iter]=Stack[iter+1];

}

bottom--;

Stack[top]=first\_empty;

}

//如果没有空位，替换最近使用最少

else

{

page[pagecontrol[Stack[bottom+1]]]=0;

pagecontrol[Stack[bottom+1]]=curpage;

page[curpage] = 1;

int tmp = Stack[bottom+1];

for(int iter = 0;iter<PP-1;iter++)

{

Stack[iter]=Stack[iter+1];

}

Stack[top]=tmp;

}

}

int main()

{

//队列初始化为-1

for(int iter=0;iter<PP+1;iter++)

{

Queue[iter]=-1;

}

//栈初始化为-1

for(int iter=0;iter<PP;iter++)

{

Stack[iter]=-1;

}

/\*定义变量

\* page是页面

\* pagecontrol是内存分配的页面，初始化为-1

\* pageseq是页面使用的顺序

\*/

printf("-----------------------------------------------------\n");

printf("两种替换算法：FIFO和LRU。\n");

int page[AP];

int pagecontrol[PP];

for(int iter = 0;iter<PP;iter++)

{

pagecontrol[iter] = -1;

}

for(int iter = 0;iter<AP;iter++)

{

page[iter] = 0;

}

int pageseq[TOTAL\_INSTRUCTION] = {};

//为pageseq的每一个元素赋随机值

srand((unsigned int)getpid());

for(int iter = 0;iter<TOTAL\_INSTRUCTION;iter++)

{

pageseq[iter]=rand()%AP;

}

//测试

//int pageseq[TOTAL\_INSTRUCTION] = {7,0,1,2,0,3,0,4,2,3,0,3,2,1,2,0,1,7,0,1};

//替换算法类型，0为FIFO，1为LRU

printf("可分配的物理块数量为：%d\n",PP);

printf("页面访问顺序为：\n%d",pageseq[0]);

for(int iter = 1;iter<TOTAL\_INSTRUCTION;iter++)

{

printf(" %d",pageseq[iter]);

}

printf("\n");

int algorithm\_type = 0;

printf("请输入替换算法类型：0为FIFO，1为LRU。\n");

scanf("%d",&algorithm\_type);

switch(algorithm\_type)

{

case 0: {printf("FIFO算法。\n");break;}

case 1: {printf("LRU算法。\n");break;}

default: {printf("Error！\n");exit(1);}

}

//cur是当前页面序列下标

int cur = 0;

//diseffect是未命中次数

int diseffect = 0;

while(cur!=TOTAL\_INSTRUCTION)

{

//如果序列当前页面在已经分配的页面中

if(page[pageseq[cur]]==1)

//FIFO什么都不用做

//LRU要更新记录

{

if(algorithm\_type==1)

{

LRU(pageseq[cur],page,pagecontrol,PP);

}

}

//如果不在当前已经分配的页面中

else

{

diseffect++;

//使用FIFO算法

if(algorithm\_type == 0)

{

FIFO(pageseq[cur],page,pagecontrol,PP);

}

//使用LRU算法

else

{

LRU(pageseq[cur],page,pagecontrol,PP);

}

}

//记录pagecontrol的值便于输出

for(int iter = 0;iter<PP;iter++)

{

temp[iter][cur]=pagecontrol[iter];

}

//查看下一个待调用的页面

cur++;

}

//计算命中率

double hit\_rate = 100\*(1-((double)diseffect/TOTAL\_INSTRUCTION));

//输出相关信息

printf("-----------------------------------------------------\n");

printf("替换次序：\n");

for(int iter = 0; iter<PP;iter++)

{

printf("%d",temp[iter][0]);

for(int iter2 = 1;iter2<TOTAL\_INSTRUCTION;iter2++)

{

printf("\t%d",temp[iter][iter2]);

}

printf("\n");

}

printf("-----------------------------------------------------\n");

printf("end：\n共访问页面次数：%d\n",TOTAL\_INSTRUCTION);

printf("未命中次数：%d\n",diseffect);

printf("命中率为：%.2f%%\n",hit\_rate);

printf("-----------------------------------------------------\n");

return 0;

}

本程序随机生成页面访问序列，用键盘输入使用的页面置换算法。在主函数中先将各个数据结构和变量初始化，再根据选择的算法类型和当前要使用的页面是否在pagecontrol内使用不同的策略。如果当前需要的页面在pagecontrol里面，则FIFO算法什么也不做，而LRU要更新Stack中的顺序；若当前需要的页面不在pagecontrol里，则需要调入页面。对于FIFO是调入队尾处；对于LRU是调入栈底处，并将栈底值取出置于栈顶，表示最近使用过。将每一次的替换记录在数组temp中，将未命中的次数记录在diseffect中，最后输出temp数组和相关计算数据

