（往下拉）

目录

[I. 进程管理实验 2](#_Toc27654)

[A. 实验目的 2](#_Toc21130)

[B. 实验内容 2](#_Toc6597)

[C. 实验要求 2](#_Toc9522)

[D. 软中断 3](#_Toc11070)

[E. 管道通信 5](#_Toc32017)

[II. 存储器管理实验 8](#_Toc1248)

[A. 实验目的 8](#_Toc14864)

[B. 实验内容 8](#_Toc10072)

[C. 实验要求 8](#_Toc8421)

[D. 实验原理 8](#_Toc30784)

[E. 实验代码 9](#_Toc19863)

[F. 实验结果 15](#_Toc1293)

[G. 遇到的问题 17](#_Toc6951)

[III. 体会 17](#_Toc7591)

# 进程管理实验

## 实验目的

1) 加深对进程概念的理解，明确进程和程序的区别。

2) 进一步认识并发执行的实质。

3) 分析进程争用资源的现象，学习解决进程互斥的方法。

4) 了解Linux系统中进程通信的基本原理。

进程是操作系统中最重要的概念，贯穿始终，也是学习现代操作系统的关键。通过本次实验，要求理解进程的实质和进程管理的机制。在Linux系统下实现进程从创建到终止的全过程，从中体会进程的创建过程、父进程和子进程的关系、进程状态的变化、进程之间的同步机制、进程调度的原理和以信号和管道为代表的进程间通信方式的实现。

## 实验内容

1) 编制实现软中断通信的程序

使用系统调用fork()创建两个子进程，再用系统调用signal()让父进程捕捉键盘上发出的中断信号（即按delete键），当父进程接收到这两个软中断的某一个后，父进程用系统调用kill()向两个子进程分别发出整数值为16和17软中断信号，子进程获得对应软中断信号，然后分别输出下列信息后终止：

Child process 1 is killed by parent !!

Child process 2 is killed by parent !!

父进程调用wait()函数等待两个子进程终止后，输入以下信息，结束进程执行：

Parent process is killed!!

多运行几次编写的程序，简略分析出现不同结果的原因。

2) 编制实现进程的管道通信的程序

使用系统调用pipe()建立一条管道线，两个子进程分别向管道写一句话：

Child process 1 is sending a message!

Child process 2 is sending a message!

而父进程则从管道中读出来自于两个子进程的信息，显示在屏幕上。

要求：父进程先接收子进程P1发来的消息，然后再接收子进程P2发来的消息。

## 实验要求

1) 按照实验内容，认真完成各项实验，并完成实验报告。

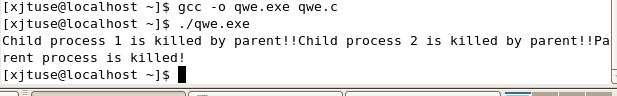
2) 实验报告必须包括：程序清单（含注释）、实验结果、实验中出现的问题、观察到的现象的解释和说明，以及实验体会。

## 软中断

1. **程序清单**



1. **实验结果**

捕获1.PNG

1. **实验中出现的问题**
   1. Signal函数的使用

问题:

在实验中,需要使用signal函数接收键盘上DELETE键的中断信号和父进程传递的16、17,而对于如何捕捉该信号,却不清楚.

解决:

经查询资料,得知:

signal(SIGHUP, SIG\_IGN);

signal信号函数，第一个参数表示需要处理的信号值（SIGHUP），第二个参数为处理函数或者是一个表示，这里，SIG\_IGN表示忽略SIGHUP那个注册的信号。

SIGHUP和控制台操作有关，当控制台被关闭时系统会向拥有控制台sessionID的所有进程发送HUP信号，默认HUP信号的action是 exit，如果远程登陆启动某个服务进程并在程序运行时关闭连接的话会导致服务进程退出，所以一般服务进程都会用nohup工具启动或写成一个 daemon。

在signals中,有:

|  |  |
| --- | --- |
| SIGINT | 由Interrupt Key产生，通常是CTRL+C或者DELETE。发送给所有ForeGround Group的进程 |

故在接收键盘信号时,使用”SIGINT"信号.

在子进程中，使用直接16、17即可。

* 1. 加库

问题:

Fork,signal,wait等函数并不属于默认库中,需引进这些库.

解决:

[kill]函数 int kill(pid\_t pid, int sig);

用于向任何进程组或进程发送信号。

#include <sys/types.h>

#include <signal.h>

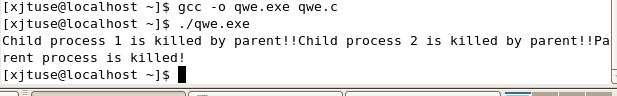
[fork]函数

#include <unistd.h>

[signal]函数 signal(SIGHUP, SIG\_IGN);

#include <signal.h>

1. **实验现象**
   1. 现象

捕获1.PNG

* 1. 解释说明

子进程1,2被创建后,使用signal函数等待父进程把信号传入,待传入后依次输出并退出子进程.

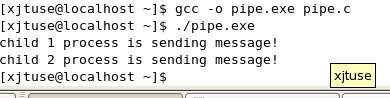
父进程使用两个wait等待两个子进程分别结束后,输出并退出.

## 管道通信

1. **程序**



1. **实验结果**



1. **出现的问题**
   * 1. 互斥问题

问题：两个子进程同时向管道发送信息，如何平衡两者以保证不冲突。

解决：使用lockf函数，对子进程加锁，通过锁的开关解决互斥问题。

* + 1. Exit()函数报错

问题：

解决：exit并不属于<stdio.h>库，而是属于<stdlib.h>，故在使用时需加入此库。

【附】exit(0)与exit(1),return三者区别

exit（0）：正常运行程序并退出程序；

exit（1）：非正常运行导致退出程序；

return（）：返回函数，若在主函数中，则会退出函数并返回一值。

详细说：

1. return返回函数值，是关键字； exit 是一个函数。

2. return是语言级别的，它表示了调用堆栈的返回；而exit是系统调用级别的，它表示了一个进程的结束。

3. return是函数的退出(返回)；exit是进程的退出。

4. return是C语言提供的，exit是操作系统提供的（或者函数库中给出的）。

5. return用于结束一个函数的执行，将函数的执行信息传出个其他调用函数使用；exit函数是退出应用程序，删除进程使用的内存空间，并将应用程序的一个状态返回给OS，这个状态标识了应用程序的一些运行信息，这个信息和机器和操作系统有关，一般是 0 为正常退出，非0 为非正常退出。

6. 非主函数中调用return和exit效果很明显，但是在main函数中调用return和exit的现象就很模糊，多数情况下现象都是一致的。

exit和\_exit

进程终止有5种方法：

1正常终止

（1）从main函数返回

（2）调用exit

（3）调用\_exit

2异常终止

（1）调用abort

（2）由一个信号来终止

exit和\_exit就是用来正常终止一个进程的，主要区别是\_exit会立刻进入内核，而exit先执行一些清除工作（包括执行各种终止处理程序，关闭所有标准I／O等，一旦关闭了IO，例如Printf等函数就不会输出任何东西了），然后才进入内核。这两个函数会对父子进程有一定的影响，当用vfork创建子进程时，子进程会先在父进程的地址空间运行（这跟fork不一样），如果子进程调用了exit就会把父进程的IO给关掉。

这两个函数都带一个参数表示终止状态，这跟我们平时写的return效果是一样的，如果不返回一个终止状态，那表示这个进程的终止状态就是未定义的。

1. **观察到的现象**

子进程1，2分别依次进行输出。

# 存储器管理实验

## 实验目的

1) 理解内存页面调度的机理

2) 掌握几种理论页面置换算法的实现方法

3) 了解HASH数据结构的使用

4) 通过实验比较几种调度算法的性能优劣

页面置换算法是虚拟存储管理实现的关键，通过本次实验理解内存页面调度的机制，在模拟实现FIFO、LRU、NRU和OPT几种经典页面置换算法的基础上，比较各种页面置换算法的效率及优缺点，从而了解虚拟存储实现的过程。

## 实验内容

对比以下几种算法的命中率：

1) 先进先出算法FIFO（First In First Out）

2) 最近最少使用算法LRU（Least Recently Used）

3) 最近未使用算法NUR（Never Used Recently）

4) 最佳置换算法OPT（Optimal Replacement）

## 实验要求

1) 按照实验内容，认真完成各项实验，并完成实验报告。

2) 实验报告必须包括：程序清单（含注释）、实验结果、实验中出现的问题、观察到的现象的解释和说明，以及实验体会。

## 实验原理

1. **FIFO算法**

（1）在分配内存页面数（AP）小于进程页面数（PP）时，当然是最先运行的AP个页面放入内存；

（2）这时又需要处理新的页面，则将原来放的内存中的AP个页中最先进入的调出（FIFO），再将新页面放入；

（3）以后如果再有新页面需要调入，则都按上述规则进行。

（4）算法特点：所使用的内存页面构成一个队列。

1. **LRU算法**

（1）当内存分配页面数（AP）小于进程页面数（PP）时，把最先执行的AP个页面放入内存。

（2）当需调页面进入内存，而当前分配的内存页面全部不空闲时，选择将其中最长时间没有用到的那一页调出，以空出内存来放置新调入的页面（LRU）。

（3）算法特点：每个页面都有属性来表示有多长时间未被CPU使用的信息。

1. **NUR算法**

所谓“最近未使用”，首先是要对“近”做一个界定，比如CLEAR\_PERIOD=50，便是指在CPU最近的50次进程页面处理工作中，都没有处理到的页面。那么可能会有以下几种情况：

（1）如果这样的页面只有一个，就将其换出，放入需要处理的新页面。

（2）如果有这样的页面不止一个，就在这些页面中任取一个换出（可以是下标最小的或者最小的），放入需要处理的页面。

（3）如果没有一个这样的页面，就随意换出一个页面（可以是下标最小的或者最大的）。

算法特点：有一个循环周期，每到达这个周期，所有页面存放是否被CPU处理的信息的属性均被置于初始态（没有被访问）。

1. **OPT算法**

所谓的最佳算法是一种理想状况下的算法，它要求先遍历所有的CPU待处理的进程页面序列。在这些页面中，如果有些已经在内存中，而CPU不再处理的，就将其换出；而有些页面已在内存中而CPU即将处理，就从当前位置算起，取最后才会处理到的页面，将其换出。如CPU待处理的页面序列为：

|  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| 1 | 3 | 2 | 2 | 4 | 5 | 2 | 5 | 1 | 4 | 3 | 4 | 1 | 1 | 5 | 5 | 3 | 4 | 2 | 1 |

如果已经处理了前5个页面（底纹为黑色），那么随后的页面5是第一个待处理的页面，这时若要将页面5调入内存，需选择页面3换出，因为页面3 是最后才会被处理到的。

## 实验代码

#include <iostream>

#include<map>

#include<set>

#include <algorithm>

#include<cstdio>

#include<cstring>

#include<cmath>

#define N 200

using namespace std;

int page[N];//页面引用号

int block[N];//物理块，内存

int dist[N][N];//表示第i次访问内存的时候，内存中的页面j 在以后被访问的最小时间

int n;//页面引用号个数

int m;//物理块数目

int page\_max;//最大页面号

int pre[N];//page[i]在page中的索引

int opt(){//最佳页面置换算法

printf("OPT页面置换情况: \n");

int page\_lack = 0;

memset(pre, 0, sizeof(pre));

memset(dist, 0x3f, sizeof(dist));

memset(block, -1, sizeof(block));

for(int i=n; i>=1; --i){

for(int j=0; j<=page\_max; ++j)

if(pre[j])

dist[i][j] = pre[j] - i;

pre[page[i]] = i;

}

for(int i=1; i<=n; ++i){//开始访问页面，初始是内存中没有分页

int j;

int max\_dist = 0, p;

for(j=1; j<=m; ++j){

if(block[j] == -1){//改块没有放入页面，则直接放入, 产生缺页

block[j] = page[i];

//cout<<"页面"<<page[i]<<"不在内存，直接放入物理块"<<j<<"中!"<<endl;

page\_lack++;

break;

} else if(block[j] == page[i])//页面存在内存中

break;

if(max\_dist < dist[i][block[j]]){

max\_dist = dist[i][block[j]];//说明block[j] 对应的页面以后会长时间不会用到

p = j;//block[] 第j个页面会被替换掉

}

}

if(j > m){//此时内存中不能在放入新的分页，而且没有找到page[i]对应的分页，接下来进行页面替换

//cout<<"页面"<<page[i]<<"不在内存，将物理块"<<p<<"中的页面" <<block[p]<<"替换掉!"<<endl;

block[p] = page[i];

page\_lack++;

}

//cout<<endl<<"当前内存中页面的情况:"<<endl;

for(int k=1; k<=m; ++k)//内存中页面加载情况

cout<<block[k]<<" ";

cout<<endl<<endl;

}

float missRate = (float)page\_lack/n;

printf("缺页率：%.2f%%\n",missRate\*100);

return page\_lack;//返回缺页次数

}

int lru(){//最近最久未使用算法和opt算法差不多，只不过是lru是向前看， opt是向后看

printf("LRU页面置换情况: \n");

int page\_lack = 0;

memset(pre, 0, sizeof(pre));

memset(dist, 0x3f, sizeof(dist));

memset(block, -1, sizeof(block));

for(int i=1; i<=n; ++i){

for(int j=0; j<=page\_max; ++j)

if(pre[j])

dist[i][j] = i - pre[j];

pre[page[i]] = i;

}

for(int i=1; i<=n; ++i){//开始访问页面，初始是内存中没有分页

int j;

int max\_dist = 0, p;

for(j=1; j<=m; ++j){

if(block[j] == -1){//改块没有放入页面，则直接放入, 产生缺页

block[j] = page[i];

//cout<<"页面"<<page[i]<<"不在内存，直接放入物理块"<<j<<"中!"<<endl;

page\_lack++;

break;

} else if(block[j] == page[i])//页面存在内存中

break;

if(max\_dist < dist[i][block[j]]){

max\_dist = dist[i][block[j]];//说明block[j] 对应的页面以后会长时间不会用到

p = j;//block[] 第j个页面会被替换掉

}

}

if(j > m){//此时内存中不能在放入新的分页，而且没有找到page[i]对应的分页，接下来进行页面替换

//cout<<"页面"<<page[i]<<"不在内存，将物理块"<<p<<"中的页面" <<block[p]<<"替换掉!"<<endl;

block[p] = page[i];

page\_lack++;

}

//cout<<endl<<"当前内存中页面的情况:"<<endl;

for(int k=1; k<=m; ++k)//内存中页面加载情况

cout<<block[k]<<" ";

cout<<endl<<endl;

}float missRate = (float)page\_lack/n;

printf("缺页率：%.2f%%\n",missRate\*100);

return page\_lack;//返回缺页次数

}

set<int>page\_set;

int fifo(){//先进先出页面置换算法

printf("FIFO页面置换情况: \n");

int page\_lack = 0;

memset(block, -1, sizeof(block));

int index = 1;

for(int i=1; i<=n; ++i){

if(index > m) index = 1;

set<int>::iterator it;

it = page\_set.find(page[i]);

if(it == page\_set.end()){

if(block[index] != -1)

page\_set.erase(block[index]);

page\_set.insert(page[i]);

block[index++] = page[i];

++page\_lack;

}

for(int k=1; k<=m; ++k)

cout<<block[k]<<" ";

cout<<endl;

} float missRate = (float)page\_lack/n;

printf("缺页率：%.2f%%\n",missRate\*100);

return page\_lack;

}

int nru[N];//表示 物理块 i 最近时候被访问过

int page\_in\_block[N];//页面 i 在 block的下标索引

int clock(){

printf("NRU页面置换情况: \n");

int index = 1;

int page\_lack = 0;

memset(block, -1, sizeof(block));

for(int i=1; i<=n; ++i){

if(page\_in\_block[page[i]]){//如果page[i]已经在内存中

nru[page\_in\_block[page[i]]] = 1;//重新标记这个物理块中的页面被访问过了

//cout<<endl<<"第"<<i<<"次: 页面"<<page[i]<<"已经存在物理块"<< page\_in\_block[page[i]] << "中!"<<endl;

}

else {

while(true){

if(index > m) index = 1;

if(block[index] == -1) {

nru[index] = 1;

page\_in\_block[page[i]] = index;

block[index++] = page[i];

++page\_lack;

break;

}

if(block[index] == page[i]){

nru[index++] = 1;

break;

} else {

if(nru[index] == 0){//替换该页面

nru[index] = 1;

page\_in\_block[block[index]] = 0;

//cout<<endl<<"第"<<i<<"次: 物理块"<<index<<"中的页面"<< block[index] <<"最近未被使用，将要被页面"<<page[i]<<"替换!"<<endl;

page\_in\_block[page[i]] = index;

block[index++] = page[i];

++page\_lack;

break;

} else

nru[index++] = 0;

}

}

}

for(int k=1; k<=m; ++k)

cout<<block[k]<<" ";

cout<<endl;

}float missRate = (float)page\_lack/n;

printf("缺页率：%.2f%%\n",missRate\*100);

return page\_lack;

}

int main(){

cin>>n>>m;

//printf("%d\n",m);

for(int i=1; i<=n; ++i){

cin>>page[i];

page\_max = max(page\_max, page[i]) ;

}

cout<<"opt缺页中断次数:"<<opt()<<endl;

cout<<"\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*"<<endl;

cout<<"lru缺页中断次数:"<<lru()<<endl;

cout<<"\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*"<<endl;

cout<<"fifo缺页中断次数:"<<fifo()<<endl;

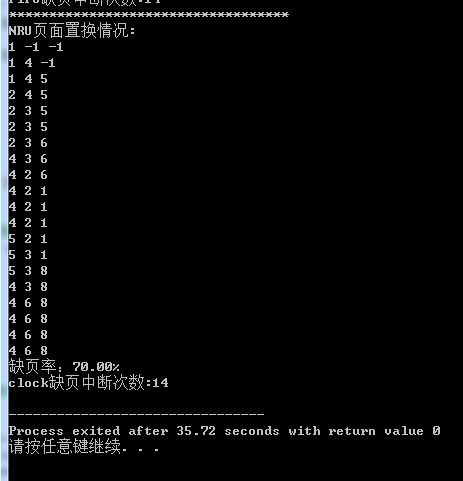
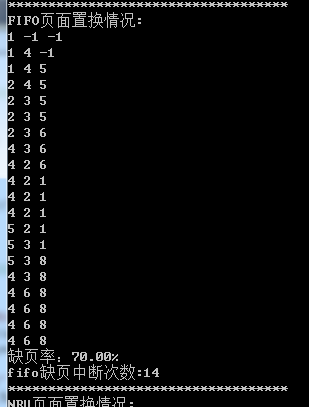
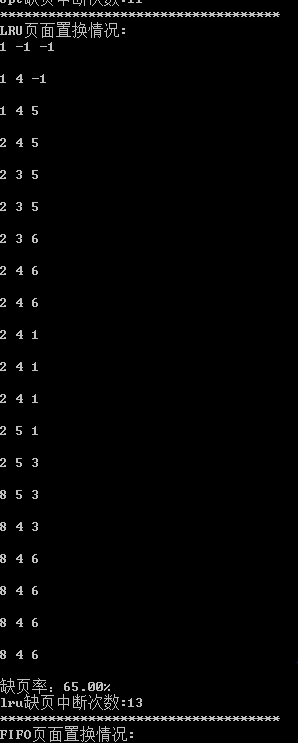
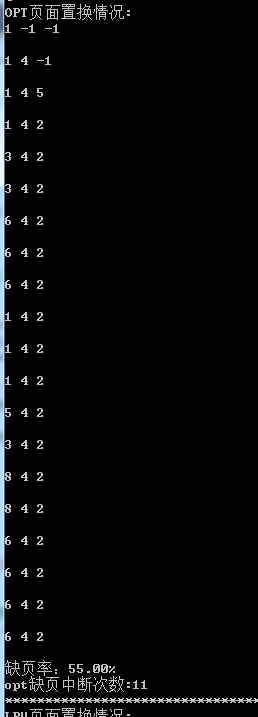
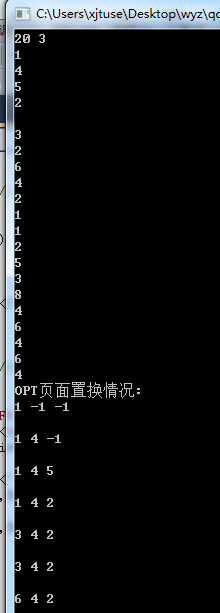
cout<<"\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*"<<endl;

cout<<"clock缺页中断次数:"<<clock()<<endl;

return 0;

}

## 实验结果



## 遇到的问题

### 缺页率

问题：缺页率在计算时，总为整数，输出误差极大。

解决：对计算结果进行强制类型转换。

### NUR算法

问题：nur算法中涉及对界定界限的判断，逻辑有些理不清，较复杂。

解决：在网上搜索解析，学习他人代码。

# 体会

所有看似复杂的功能和算法，都是由简单的步骤一步步组成的。

当发现一个特性不知如何表示时，添加一个变量会是不错的选择。

全局变量虽说会提高系统耦合性，但在不用考虑耦合性时，是一个很不错的方法。

算法对于性能的提升，起着相当重要的作用，好的算法可以极大的降低硬件成本。

参考文献

[1 ] exit (0)与exit(1),return三者区别(详解),https://www.cnblogs.com/ECJTUACM-873284962/p/6882448.html

1. 软中断通信https://download.csdn.net/download/qinchaohan/1104723
2. 操作系统之页面置换算法https://www.cnblogs.com/fkissx/p/4712959.html