

---

## Contents

<b>实验 1-Windows 进程管理</b>	<b>3</b>
实验目的	3
实验内容	3
子实验 1-编写基本的 Win32 Console Application	3
子实验 2-创建进程	3
子实验 3-父子进程的简单通信和终止进程	4
实验结果与分析	6
小结与心得体会	8
<b>实验 2-Linux 进程控制</b>	<b>8</b>
实验目的	8
实验内容	8
子实验 1-进程的创建	8
子实验 2-子进程执行新任务	8
实验结果与分析	9
子实验 1	9
子实验 2	10
小结与心得	10
<b>实验 3-Linux 进程间通信</b>	<b>10</b>
实验目的	10
实验内容	10
管道	11
来自 SystemV 的 IPC 设施	11
实验结果与分析	13
小结和心得	14
<b>实验 6-银行家算法的模拟和实现</b>	<b>14</b>
实验目的	14
总体设计	14
详细设计	15
实验结果与分析	16
小结和心得	17
<b>实验 7-磁盘调度算法的模拟与实现</b>	<b>17</b>
实验目的	17
总体设计	18

---

详细设计 . . . . .	18
实验结果与分析 . . . . .	20
小结和心得 . . . . .	21
<b>实验 8-虚拟内存系统的页面置换算法模拟</b>	<b>21</b>
实验目的 . . . . .	21
总体设计 . . . . .	21
详细设计 . . . . .	22
实验结果与分析 . . . . .	25
小结和心得 . . . . .	25
<b>实验 9-基于信号量机制的并发程序设计</b>	<b>25</b>
实验目的 . . . . .	25
总体设计 . . . . .	25
详细设计 . . . . .	25
实验结果与分析 . . . . .	27
小结和心得 . . . . .	28
<b>实验 10-实现一个简单的 shell 命令行解释器</b>	<b>29</b>
实验目的 . . . . .	29
总体设计 . . . . .	29
详细设计 . . . . .	29
实验结果与分析 . . . . .	31
小结和心得 . . . . .	32

---

## 实验 1-Windows 进程管理

### 实验目的

- 学会使用 VC 编写基本的 Win32 Console Application
- 通过创建进程、观察正在运行的进程和终止进程的程序设计和调试操作，进一步熟悉操作系统的进程概念，理解 Windows 进程的“一生”
- 通过阅读和分析试验程序，学习创建进程、观察进程、终止进程以及父子进程同步的基本程序设计方法

### 实验内容

#### 子实验 1-编写基本的 Win32 Console Application

```
1  #include <iostream>
2  int main()
3  {
4      std::cout << "Hello, Win32 Console Application" << std::endl;
5  }
```

#### 子实验 2-创建进程

```
1  #include <windows.h>
2  #include <iostream>
3  #include <stdio.h>
4  // 创建传递过来的进程的克隆过程并赋予其 ID 值
5  void StartClone(int nCloneID){
6      // 提取用于当前可执行文件的文件名
7      TCHAR szFilename[MAX_PATH];
8      GetModuleFileName(NULL, szFilename, MAX_PATH);
9      // 格式化用于子进程的命令行并通知其 EXE 文件名和克隆 ID
10     TCHAR szCmdLine[MAX_PATH];
11     sprintf(szCmdLine, "\\\"%s\\\" %d", szFilename, nCloneID);
12     // 用于子进程的 STARTUPINFO 结构
13     STARTUPINFO si;
14     ZeroMemory(&si, sizeof(si));
15     si.cb = sizeof(si); // 必须是本结构的大小
16     // 返回的用于子进程的进程信息
17     PROCESS_INFORMATION pi;
18     // 利用同样的可执行文件和命令行创建进程，并赋予其子进程的性质
19     BOOL bCreateOK = ::CreateProcess(
20         szFilename, // 产生这个 EXE 的应用程序的名称
21         szCmdLine, // 告诉其行为像一个子进程的标志
22         NULL, // 缺省的进程安全性
23         NULL, // 缺省的线程安全性
24         FALSE, // 不继承句柄
```

```

25     CREATE_NEW_CONSOLE, // 使用新的控制台
26     NULL, // 新的环境
27     NULL, // 当前目录
28     &si, // 启动信息
29     &pi // 返回的进程信息
30 );
31 // 对子进程释放引用
32 if (bCreateOK){
33     CloseHandle(pi.hProcess);
34     CloseHandle(pi.hThread);
35 }
36 }
37 int main(int argc, char* argv[]){
38     // 确定派生出几个进程，及派生进程在进程列表中的位置
39     int nClone=0;
40     //修改语句: int nClone;
41     //第一次修改: nClone=0;
42     if (argc > 1){
43         // 从第二个参数中提取克隆ID
44         sscanf(argv[1], "%d", &nClone);
45     }
46     //第二次修改: nClone=0;
47     // 显示进程位置
48     std::cout << "Process ID:" << GetCurrentProcessId()
49         << ", Clone ID:" << nClone
50         << std::endl;
51     // 检查是否有创建子进程的需要
52     const int c_nCloneMax=5;
53     if (nClone < c_nCloneMax){
54         // 发送新进程的命令行和克隆号
55         StartClone(++nClone);
56     } // 等待响应键盘输入结束进程
57     getchar();
58     return 0;
59 }

```

### 子实验 3-父子进程的简单通信和终止进程

```

1 // procterm 项目
2 # include <windows.h>
3 # include <iostream>
4 # include <stdio.h>
5 static LPCTSTR g_szMutexName = "w2kdg.ProcTerm.mutex.Suicide" ;
6 // 创建当前进程的克隆进程的简单方法
7 void StartClone(){
8     // 提取当前可执行文件的文件名
9     TCHAR szFilename[MAX_PATH];
10    GetModuleFileName(NULL, szFilename, MAX_PATH);
11    // 格式化用于子进程的命令行，字符串 "child" 将作为形参传递给子进程的main 函数
12    TCHAR szCmdLine[MAX_PATH];
13    //实验2-3 步骤3: 将下句中的字符串child 改为别的字符串，重新编译执行，执行前
    请先保存已经完成的工作
14    sprintf(szCmdLine, "\"%s\" child", szFilename);

```

---

```

15     // 子进程的启动信息结构
16     STARTUPINFO si;
17     ZeroMemory(&si, sizeof(si));
18     si.cb = sizeof(si); // 应当是此结构的大小
19     // 返回的用于子进程的进程信息
20     PROCESS_INFORMATION pi;
21     // 用同样的可执行文件名和命令行创建进程，并指明它是一个子进程
22     BOOL bCreateOK = CreateProcess(
23         szFilename, // 产生的应用程序的名称(本EXE 文件)
24         szCmdLine, // 告诉我们这是一个子进程的标志
25         NULL, // 用于进程的缺省的安全性
26         NULL, // 用于线程的缺省安全性
27         FALSE, // 不继承句柄
28         CREATE_NEW_CONSOLE, // 创建新窗口
29         NULL, // 新环境
30         NULL, // 当前目录
31         &si, // 启动信息结构
32         &pi // 返回的进程信息
33     );
34     // 释放指向子进程的引用
35     if (bCreateOK) {
36         CloseHandle(pi.hProcess);
37         CloseHandle(pi.hThread);
38     }
39 }
40 void Parent() {
41     // 创建“自杀”互斥程序体
42     HANDLE hMutexSuicide = CreateMutex(
43         NULL, // 缺省的安全性
44         TRUE, // 最初拥有的
45         g_szMutexName // 互斥体名称
46     );
47     if (hMutexSuicide != NULL) {
48         // 创建子进程
49         std::cout << "Creating the child process." << std::endl;
50         StartClone();
51         // 指令子进程“杀”掉自身
52         std::cout << "Telling the child process to quit." << std::endl;
53         // 等待父进程的键盘响应
54         getchar();
55         // 释放互斥体的所有权，这个信号会发送给子进程的WaitForSingleObject 过程
56         ReleaseMutex(hMutexSuicide);
57         // 消除句柄
58         CloseHandle(hMutexSuicide);
59     }
60 }
61 void Child() {
62     // 打开“自杀”互斥体
63     HANDLE hMutexSuicide = OpenMutex(
64         SYNCHRONIZE, // 打开用于同步
65         FALSE, // 不需要向下传递
66         g_szMutexName); // 名称
67     if (hMutexSuicide != NULL) {
68         // 报告我们正在等待指令
69         std::cout << "Child waiting for suicide instructions. " << std::endl;
70         // 子进程进入阻塞状态，等待父进程通过互斥体发来的信号

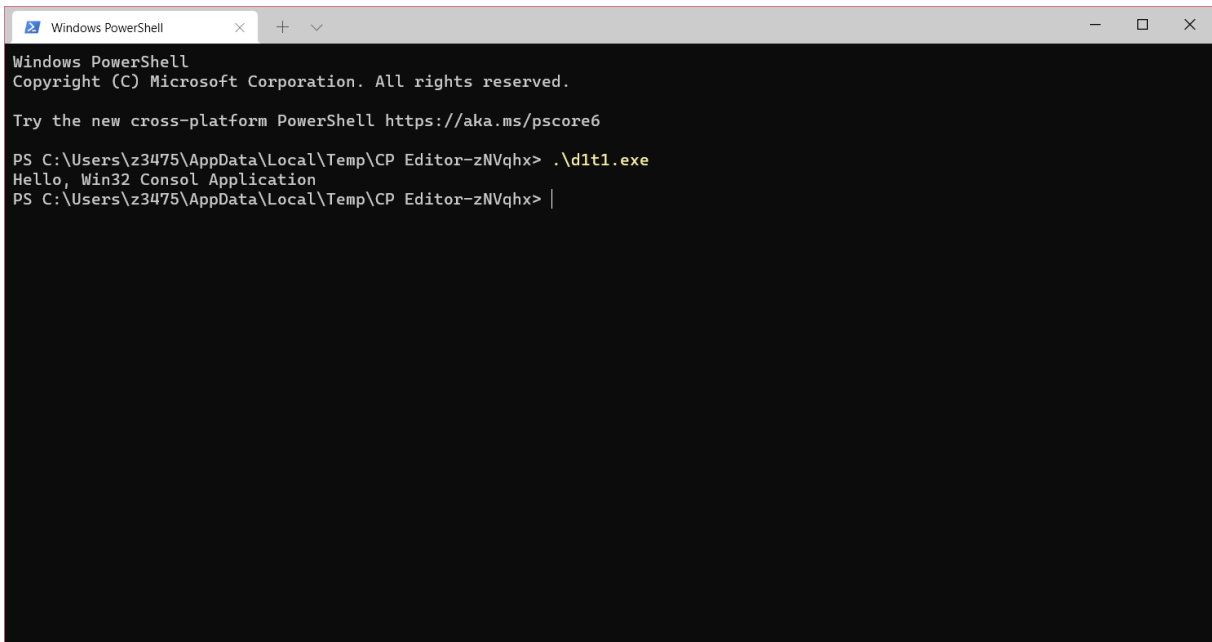
```

---

```
71         //WaitForSingleObject(hMutexSuicide, INFINITE);
72         WaitForSingleObject(hMutexSuicide, 0);
73         //实验2-3 步骤4: 将上句改为WaitForSingleObject(hMutexSuicide, 0) , 重新
           编译执行
74         // 准备好终止, 清除句柄
75         std::cout << "Child quitting." << std::endl;
76         CloseHandle(hMutexSuicide);
77     }
78 }
79 int main(int argc, char* argv[] ){
80     // 决定其行为是父进程还是子进程
81     if (argc>1&&strcmp(argv[1],"child")==0){
82         Child() ;
83     }else{
84         Parent() ;
85     }
86     return 0;
87 }
```

## 实验结果与分析

### • 子实验 1



```
Windows PowerShell
Copyright (C) Microsoft Corporation. All rights reserved.

Try the new cross-platform PowerShell https://aka.ms/pscore6

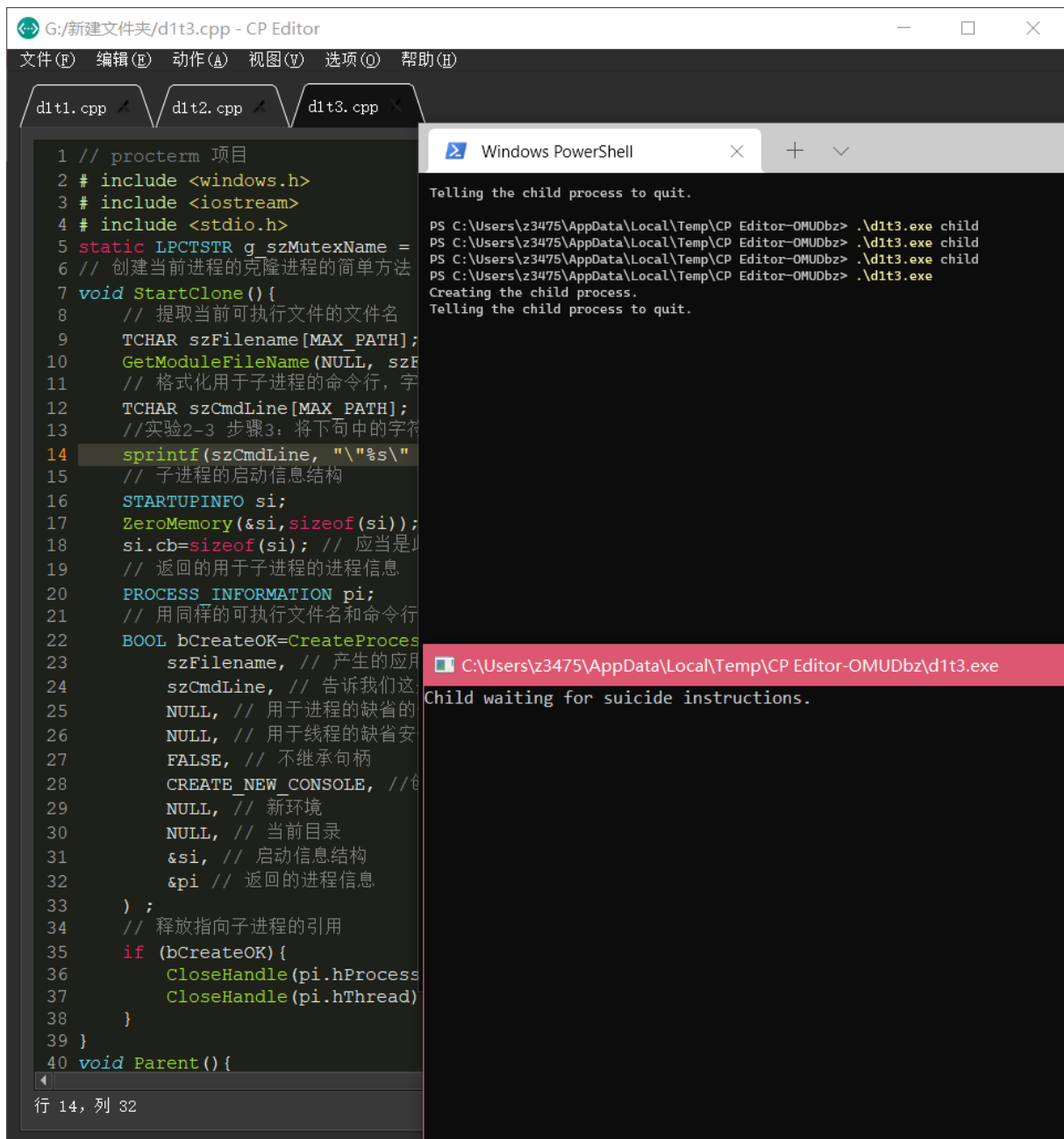
PS C:\Users\z3475\AppData\Local\Temp\CP Editor-zNVqh> .\d1t1.exe
Hello, Win32 Console Application
PS C:\Users\z3475\AppData\Local\Temp\CP Editor-zNVqh> |
```

### • 子实验 2

观察到不断有进程被创建, 关掉最新创建的进程, 进程创建被终止。

### • 子实验 3

#### - 步骤 1



执行./d1t3.exe

观察到一个子进程被创建，在父进程输入回车，子进程和父进程退出。

- 步骤 2

修改 StartClone 中命令行参数，观察到不断有子进程创建，关掉最新创建的进程，进程创建被终止。

- 步骤 3

---

修改子进程 WaitForSingleObject 第二个参数为 0，观察到子进程马上退出。

## 小结与心得体会

windowsAPI 命名风格为单词组合，单词首字母大写。

Windows 进程间使用名字来区分互斥锁。

## 实验 2-Linux 进程控制

### 实验目的

通过进程的创建、撤销和运行加深对进程概念和进程并发执行的理解，明确进程和程序之间的区别。

### 实验内容

#### 子实验 1-进程的创建

多次运行程序，观察屏幕上的显示结果，分析。

```
1  #include <bits/stdc++.h>
2  using namespace std;
3
4  int main() {
5      int x;
6      srand((unsigned)time(NULL));
7      while((x=fork())!=-1);
8      if (x==0) {
9          sleep(rand() % 2);
10         printf("a");
11     }else {
12         sleep(rand() % 3);
13         printf("b");
14     }
15     printf("c");
16 }
```

#### 子实验 2-子进程执行新任务

观察该程序在屏幕上的显示结果，并分析。



---

```
1 #include <sys/wait.h>
2 #include <bits/stdc++.h>
3 using namespace std;
4 int main (){
5     pid_t pid; /* fork a child process */
6     pid = fork ();
7     if (pid < 0)
8     { /* error occurred */
9         fprintf (stderr, "Fork Failed");
10        return 1;
11    }
12    else if (pid == 0)
13    { /* 子进程 */
14        execlp ("/bin/ls", "ls", NULL);
15    }
16    else
17    { /* 父进程 */ /* 父进程将一直等待，直到子进程运行完毕*/
18        wait (NULL);
19        printf ("Child Complete");
20    }
21    return 0;
22 }
```

## 实验结果与分析

### 子实验 1

编写 makefile，在src目录下运行make d1t2data，适当时 Ctrl+C 退出。在 test/d1t2 中运行a.zsh，分析结果。

149 次运行得到的结果为

```
1 acbc -> 79
2 bcac -> 70
```

分析： 子进程会在 sleep 之后输出 ac，父进程会在 sleep 之后输出 bc。以下是所有可能的情况

```
1 abcc
2 bacc
3 acbc
4 bcac
```

因为输出的时间远小于 sleep 的时间，所以 xxcc 不可能出现。父进程 sleep 的时间期望大于子进程的时间期望。因此 acbc 出现的次数应该高于 bcac。符合实验结果

---

## 子实验 2

运行结果为

```
1 d1t2s1 d1t2s1.cpp d1t2s2 d1t2s2.cpp makefile
2 Child Complete
```

**分析** 因为本地环境原因，fork 不会失败。子进程执行ls命令，父进程等待子进程结束再输出。所以Child Complete一定会在ls命令之后输出。

## 小结与心得

Linux 进程操作摘要

1. fork() 启动新进程，返回 0 为子进程，小于 0 为错误，大于 1 为子进程 PID
2. excelp() 用一个新的进程镜像替换当前镜像
3. argv[0] 不一定等于程序在文件系统中的名字
4. wait 用于等待子进程状态改变（比如终止，被信号停止，被信号恢复）

## 实验 3-Linux 进程间通信

### 实验目的

Linux 系统的进程通信机构（IPC）允许在任意进程间大批量交换数据，通过本实验，理解 Linux 支持的消息通信机制。

### 实验内容

Linux 给用户提供的 IPC 资源为以下种

- 管道 (pipe)/先入先出队列
- 信号量 (sem)\*
- 消息队列 (msg)\*
- 共享内存 (shm)\*
- 内存文件映射 (mmap)
- 网络 (sockets)

---

## 管道

管道单向传输。先入先出队列是具名管道，管道利用虚拟文件系统进行数据传输。使用 `pipe(int fd[2])` 创建管道文件描述符。之后就可以看做文件使用。

```
1  #include <stdio.h>
2  #include <stdlib.h>
3  #include <errno.h>
4  #include <unistd.h>
5
6  int main(void)
7  {
8      int pfd[2];
9      char buf[30];
10
11     if (pipe(pfd) == -1) {
12         perror("pipe");
13         exit(1);
14     }
15
16     printf("writing to file descriptor #%d\n", pfd[1]);
17     write(pfd[1], "test", 5);
18     printf("reading from file descriptor #%d\n", pfd[0]);
19     read(pfd[0], buf, 5);
20     printf("read \"%s\"\n", buf);
21     sleep(60);
22     return 0;
23 }
```

## 来自 SystemV 的 IPC 设施

标星的设施 (信号量 (sem), 消息队列 (msg), 共享内存 (shm)) 设计取自 SystemV 的 IPC 设施，对于每一种设施的资源使用 `key(key_t, alias long)-id(int)` 管理/使用，key 可以指定也可以使用 `ftok(const char* filepath, int type)` 利用文件协助创建。

使用 `xxxget` 创建相关设施的资源，`xxxctl` 控制对应 IPC 资源。

`xxxget` 中的 `flag` 设置权限 0666 (8 进制) =rw-rw-rw-。flag 位或上以下 macro 具有一些功能。

- IPC\_PRIVATE: 不管 key 对应的资源 id 存不存在必须创建
- IPC\_CREAT: 可以创建一个新的资源
- IPC\_EXCL(exclusive): 必须创建一个新的资源 (需要和 IPC\_CREATE 共用)

一旦 `xxxget` 成功创建资源，操作系统就会设置对应资源结构体 `msqid_ds` 的 `cuid, guid, flag` (权限), `ctime`。

`xxxctl` 中 `cmd` 可取以下 macro

- IPC\_STAT: 将操作系统中的 `msqid_ds` 复制到 `buf`

- IPC\_SET: 上述操作的逆操作
- IPC\_RMID: 立即删除当前资源

**消息队列** 使用 `int msgget(key_t key, int flag)` 创建消息队列的资源, `msgctl(int id, int cmd, msqid_ds *buf)` 控制消息队列。

- `int msgsnd(int msqid, const void *msgp, size_t msgsz, int msgflg)`; 一向消息队列发送消息
- `ssize_t msgrcv(int msqid, void *msgp, size_t msgsz, long msgtyp, int msgflg)`; 从消息队列接受消息

`msgp` 是一个内存布局是以下格式的结构体, `msgsz` 标识 `mtext` 长度

```
1 struct msgbuf{
2     long mtype; //给用户使用的消息类型标识 must >0
3     char mtext[N]; //data
4 }
```

- `msgflag` 一般设置成 0, 表示阻塞。

```
1 #include <bits/stdc++.h>
2 #include <sys/ipc.h>
3 #include <sys/wait.h>
4 #include <sys/msg.h>
5 #include <sys/types.h>
6 #include <unistd.h>
7 int MSGKEY;
8 struct msgT
9 {
10     long mtype;
11     char mtext[8];
12 };
13 int msgqid, i;
14 std::atomic<int> cnt{0};
15 void
16 CLIENT ()
17 {
18     int i;
19     msgqid = msgget (MSGKEY, 0777);
20     for (i = 10; i >= 1; i--)
21     {
22         msgT msg;
23         msg.mtype = i;
24         std::cin >> msg.mtext;
25         printf ("%d (client) sent \n", ++cnt);
26         msgsnd (msgqid, &msg, strlen(msg.mtext), 0);
27         if (i!=10) sleep(1);
28     }
29     exit (0);
30 }
31 void
```

```

32 SERVER ()
33 {
34     msgqid = msgget (MSGKEY, 0777 | IPC_CREAT);
35     msgT msg;
36     do
37     {
38         msg.mtext[msgrcv (msgqid, &msg, 8, 0, 0)]=0;
39         printf ("%d (Server) recieved %s\n",++cnt,msg.mtext);
40     }
41     while (msg.mtype != 1);
42     msgctl (msgqid, IPC_RMID, 0);
43     exit (0);
44 }
45 int
46 main ()
47 {
48     MSGKEY=ftok("/tmp/t",'b');
49     std::cout<<MSGKEY<<std::endl;
50     while ((i = fork ()) == -1)
51     ;
52     if (!i)
53         SERVER ();
54     while ((i = fork ()) == -1)
55     ;
56     if (!i)
57         CLIENT ();
58     wait (0);
59     wait (0);
60 }

```

## 实验结果与分析

```

1  z3475@z3475Laptop ► ~/ACM/Operating-System-Curriculum-Design/src ► ⌵ master ►
   ./dl1t3
2  1254
3  1 (client) sent
4  1 (Server) recieved 1254
5  124
6  2 (client) sent
7  2 (Server) recieved 124
8  23
9  3 (client) sent
10 3 (Server) recieved 23
11 5335
12 4 (client) sent
13 4 (Server) recieved 5335
14 12
15 5 (client) sent
16 5 (Server) recieved 12
17 24
18 6 (client) sent
19 6 (Server) recieved 24
20 34
21 7 (client) sent

```

---

```
22 7 (Server) recieved 34
23 54
24 8 (client) sent
25 8 (Server) recieved 54
26 123
27 9 (client) sent
28 9 (Server) recieved 123
29 23
30 10 (client) sent
31 10 (Server) recieved 23
```

成功实现了进程之间的通讯

## 小结和心得

Linux 为 C 语言编程人员提供了多种多样的进程间通讯设施，有利用文件系统的管道/具名管道；沿袭自 SystemV 的 IPC 设施——信号量，消息队列，共享内存。也有现代的网络和内存文件映射等等。

SystemV 的 IPC 设施包含一组创建资源的 xxxget 函数，一组修改资源权限/删除资源的 xxxctl 函数，一组操作资源的函数。三个设施分开管理。

## 实验 6-银行家算法的模拟和实现

### 实验目的

1. 进一步了解进程的并发执行。
2. 加强对进程死锁的理解，理解安全状态与不安全状态的概念。
3. 掌握使用银行家算法避免死锁问题。

### 总体设计

#### 1. 基本概念

- 死锁：多个进程在执行过程中，因为竞争资源会造成相互等待的局面。如果没有外力作用，这些进程将永远无法向前推进。此时称系统处于死锁状态或者系统产生了死锁。
- 安全序列：系统按某种顺序并发进程，并使它们都能达到获得最大资源而顺序完成的序列为安全序列。
- 安全状态：能找到安全序列的状态称为安全状态，安全状态不会导致死锁。
- 不安全状态：在当前状态下不存在安全序列，则系统处于不安全状态。

#### 2. 银行家算法

银行家算法顾名思义是来源于银行的借贷业务，一定数量的本金要满足多个客户的借贷周转，为了防止银行家资金无法周转而倒闭，对每一笔贷款，必须考察其是否能限期归还。在操作系统中研究资源分配策略时也有类似问题，系统中有限的资源要供多个进程使用，必须保证得到的资源的进程能在有限的时间内归还资源，以供其它进程使用资源。如果资源分配不当，就会发生进程循环等待资源，则进程都无法继续执行下去的死锁现象。当一进程提出资源申请时，银行家算法执行下列步骤以决定是否向其分配资源：

1. 检查该进程所需要的资源是否已超过它所宣布的最大值。
2. 检查系统当前是否有足够资源满足该进程的请求。
3. 系统试探着将资源分配给该进程，得到一个新状态。
4. 执行安全性算法，若该新状态是安全的，则分配完成；若新状态是不安全的，则恢复原状态，阻塞该进程。

## 详细设计

```
1  #include <bits/stdc++.h>
2  using namespace std;
3  array<int,2> lens;
4  auto &n=lens[0],&m=lens[1];
5  template<class T,size_t id>
6  struct vec:public vector<T>{
7      constexpr static int& len=lens[id];
8      vec():vector<T>(len){
9          if constexpr (is_same_v<T,int>) fill(this->begin(),this->end(),0);
10     }
11 };
12 using rvec=vec<int,1>;//资源向量类型
13 struct bank{
14     rvec resource,available;
15     vec<rvec,0> claim,allocation;//声明和分配矩阵
16     bank add(int id,const rvec& a){//为id为id的进程分配资源
17         bank b=*this;
18         for (int i=0;i<m;i++){
19             b.available[i]-=a[i];
20             b.allocation[id][i]+=a[i];
21         }
22         return b;
23     }
24     bool vaild(){return ((bank)*this)._vaild();}//判断当前情况是否为安全状态
25     bool _vaild(){//判断安全状态内部实现
26         auto check=[&](int id){//判断当前进程能否执行完毕
27             for (int i=0;i<m;i++)
28                 if (claim[id][i]-allocation[id][i]>available[i])
29                     return false;
30             return true;
31         };
32         auto allzero=[&](auto &ma,int id){//判断当前进程是否执行完毕
33             for (int i=0;i<m;i++)
34                 if (ma[id][i]!=0) return false;
```

```

35         return true;
36     };
37     while ([&](){
38         bool f=false;//循环到无进程执行完毕
39         for (int i=0;i<n;i++)
40             if (!allzero(allocation,i))//如果当前进程还持有资源
41                 if (check(i)){//判断能否执行完毕
42                     f=true;
43                     for (int j=0;j<m;j++){//执行
44                         available[j]+=exchange(allocation[i][j],0);
45                     }
46                 }
47         return f;
48     }());
49     for (int i=0;i<n;i++)
50         if (!allzero(allocation,i)) return false;//无法取得进展情况下，如果
51         //存在进程还持有资源则为不安全状态
52     return true;
53 };
54 int main(int argc,char *argv[]){
55     cerr<<"Please Enter Process Count and Resource Count"<<endl;
56     cin>>n>>m;
57     bank b;
58     cerr<<"Please Enter Resource Vector" << endl;
59     for (int i=0;i<m;i++) cin>>b.resource[i];
60     b.available=b.resource;
61     cerr<<"Please Enter Claim Matrix" << endl;
62     for (auto &i:b.claim) for (auto &j:i) cin>>j;
63     while (cin){
64         cerr<<"Please Enter Process ID"<<endl;
65         int i;cin>>i;
66         cerr<<"Please Enter Need Resource Vector"<<endl;
67         rvec need;
68         for (int i=0;i<m;i++)
69             cin>>need[i];
70         if (!cin) return 0;
71         auto bn=b.add(i,need);
72         if (bn.vaild()){//判断添加之后是否为安全状态
73             cout << "Accept" << endl;
74             b=bn;
75         }else{
76             cout << "Refuse" << endl;
77         }
78     }
79 }

```

## 实验结果与分析

```

1 Please Enter Process Count and Resource Count
2 4 3
3 Please Enter Resource Vector
4 9 3 6
5 Please Enter Claim Matrix

```



---

```
6 3 2 2
7 6 1 3
8 3 1 4
9 4 2 2
10 Please Enter Process ID
11 0
12 Please Enter Need Resource Vector
13 1 0 0
14 Accept
15 Please Enter Process ID
16 1
17 Please Enter Need Resource Vector
18 5 1 1
19 Accept
20 Please Enter Process ID
21 2
22 Please Enter Need Resource Vector
23 2 1 1
24 Accept
25 Please Enter Process ID
26 3
27 Please Enter Need Resource Vector
28 0 0 2
29 Accept
30 Please Enter Process ID
31 0
32 Please Enter Need Resource Vector
33 1 0 1
34 Refuse
35 Please Enter Process ID
36 0
37 Please Enter Need Resource Vector
38 1 0 1
39 Refuse
```

成功实现了银行家算法。

## 小结和心得

银行家算法建立在已知作业的最大资源申请数量上，以较少作业下能够接受的时间复杂度，可以有效地避免死锁情况的发生。

## 实验 7-磁盘调度算法的模拟与实现

### 实验目的

- 了解磁盘结构以及磁盘上数据的组织方式。
- 掌握磁盘访问时间的计算方式。

- 掌握常用磁盘调度算法及其相关特性。

## 总体设计

### 1. 磁盘数据的组织

磁盘上每一条物理记录都有唯一的地址，该地址包括三个部分：磁头号（盘面号）、柱面号（磁道号）和扇区号。给定这三个量就可以唯一地确定一个地址。

### 2. 磁盘访问时间的计算方式

磁盘在工作时以恒定的速率旋转。为保证读或写，磁头必须移动到所要求的磁道上，当所要求的扇区的开始位置旋转到磁头下时，开始读或写数据。对磁盘的访问时间包括：寻道时间、旋转延迟时间和传输时间。

### 3. 磁盘调度算法

磁盘调度的目的是要尽可能降低磁盘的寻道时间，以提高磁盘 I/O 系统的性能。

- 先进先出算法：按访问请求到达的先后次序进行调度。
- 最短服务时间优先算法：优先选择使磁头臂从当前位置开始移动最少的磁盘 I/O 请求进行调度。
- SCAN（电梯算法）：要求磁头臂先沿一个方向移动，并在途中满足所有未完成的请求，直到它到达这个方向上的最后一个磁道，或者在这个方向上没有别的请求为止，后一种改进有时候称作 LOOK 策略。然后倒转服务方向，沿相反方向扫描，同样按顺序完成所有请求。
- C-SCAN（循环扫描）算法：在磁盘调度时，把扫描限定在一个方向，当沿某个方向访问到最后一个磁道时，磁头臂返回到磁盘的另一端，并再次开始扫描。

## 详细设计

```
1  #include <bits/stdc++.h>
2
3  using namespace std;
4  #define V_MAX 200//磁盘磁道号数量
5  namespace scheduler{
6      struct base_scheduler{//基本规划器
7          deque<int> v;//请求队列
8          base_scheduler(vector<int> _v):
9              v(_v.begin(),_v.end()){
10             }
11         virtual ~base_scheduler(){}
12         virtual pair<int,deque<int>::iterator> take(int pos)=0;//寻找在pos位置
            的下一个方向
13         int deal(int pos){//处理初始位置为pos时，清空请求队列下的总请求等待时间
14             int step=0,cnt=0;
15             while (v.size()){
```

```

16         auto p=take(pos);
17         pos+=p.first>pos?1:-1;
18         step++;
19         if (p.second!=v.end()&&*p.second==pos) {
20             v.erase(p.second);
21             cnt+=exchange(step,0);
22         }
23     }
24     return cnt;
25 }
26 };
27 struct FIFO_scheduler:public base_scheduler{//FIFO调度器（先进先出算法）
28     using base_scheduler::base_scheduler;
29     virtual pair<int,deque<int>::iterator> take(int pos){
30         return {v.front(),v.begin()};
31     }
32 };
33
34 struct SSTF_scheduler:public base_scheduler{//SSTF调度器（最短服务时间优先
35     算法）
36     using base_scheduler::base_scheduler;
37     virtual pair<int,deque<int>::iterator> take(int pos){
38         int minn=0x3f3f3f3f,mini=0;
39         for (size_t i=0;i<v.size();i++)
40             if (abs(v[i]-pos)<minn){
41                 minn=abs(v[i]-pos);
42                 mini=i;
43             }
44         return {v[mini],v.begin()+mini};
45     }
46 };
47 struct SCAN_scheduler:public base_scheduler{//SCAN调度器（电梯算法）
48     using base_scheduler::base_scheduler;
49     bool f=false;
50     virtual pair<int,deque<int>::iterator> take(int pos){
51         deque<int>::iterator w=v.end();
52         for (auto i=v.begin();i<v.end();i++)
53             if (f==(*i>pos))
54                 if (w==v.end()||f==(*i<*w)) w=i;
55
56         if (w!=v.end()) {
57             return {*w,w};
58         }else{
59             f=!f;
60             return take(pos);
61         }
62     }
63 };
64
65 struct C_SCAN_scheduler:public base_scheduler{//C-SCAN调度器（循环扫描算
66     法）
67     using base_scheduler::base_scheduler;
68     bool f=false;
69     virtual pair<int,deque<int>::iterator> take(int pos){
70         if (!f){

```

```

70         deque<int>::iterator w=v.end();
71         for (auto i=v.begin();i<v.end();i++)
72             if (f==( *i>pos))
73                 if (w==v.end()||(f==( *i<*w))) w=i;
74
75         if (w!=v.end()) {
76             return { *w,w};
77         }else {
78             f=true;
79             return take(pos);
80         }
81     }else{
82         deque<int>::iterator w=v.end();
83         for (auto i=v.begin();i<v.end();i++)
84             if (w==v.end()||( *i>*w)) w=i;
85         if (pos!=*w)
86             return { *w,w};
87         else{
88             f=false;
89             return take(pos);
90         }
91     }
92 }
93 };
94 }
95 using namespace scheduler;
96
97 int main(){
98     int n,pos;
99     cin>>n>>pos;pos=200-pos;
100     vector<int> v(n);
101     for (auto &i:v) {cin>>i;i=200-i;}
102     vector<pair<unique_ptr<base_scheduler>,string>> l;
103     l.emplace_back(new FIFO_scheduler(v),"FIFO");
104     l.emplace_back(new SSTF_scheduler(v),"SSTF");
105     l.emplace_back(new SCAN_scheduler(v),"SCAN");
106     l.emplace_back(new C_SCAN_scheduler(v),"C_SCAN");
107     for (auto& [schi,name]:l)
108         cout << fixed << setprecision(1) << schi->deal(pos)/((double)n << " "
109             << name << endl;

```

## 实验结果与分析

以书上数据作为输入运行结果为

```

1  9 100
2  55 58 39 18 90 160 150 38 184
3  55.3 FIFO
4  27.6 SSTF
5  27.8 SCAN
6  35.8 C_SCAN

```

---

和书上数据一致，SSTF 和 SCAN 算法表现最好，C\_SCAN 较差，FIFO 性能最差。

## 小结和心得

磁盘调度算法直接决定了单个应用程序的等待时间，发明好的磁盘调度算法能有助于增加 cpu 利用率，降低单个应用程序的运行时间。SSTF 和 SCAN 调度算法均为表现好的算法，实践中应该尽量使用。

## 实验 8-虚拟内存系统的页面置换算法模拟

### 实验目的

通过对页面、页表、地址转换和页面置换过程的模拟，加深对虚拟页式内存管理系统的页面置换原理和实现过程的理解。

### 总体设计

需要调入新页面时，选择内存中哪个物理页面被置换，称为置换策略。页面置换算法的目标：把未来不再使用的或短期内较少使用的页面调出，通常应在局部性原理指导下依据过去的统计数据预测，减少缺页次数。

教材给出的常用的页面置换算法包括：

- 最佳置换算法 (OPT)：置换时淘汰“未来不再使用的”或“在离当前最远位置上出现的”页面。
- 先进先出置换算法 (FIFO)：置换时淘汰最先进入内存的页面，即选择驻留在内存时间最长的页面被置换。
- 最近最久未用置换算法 (LRU)：置换时淘汰最近一段时间最久没有使用的页面，即选择上次使用距当前最远的页面淘汰
- 时钟算法 (Clock)：也称最近未使用算法 (NRU, Not Recently Used)，它是 LRU 和 FIFO 的折衷。

通过随机数产生一个指令序列，共 320 条指令。

1. 50% 的指令是顺序执行的；
2. 25% 的指令是均匀分布在前地址部分；
3. 25% 的指令是均匀分布在后地址部分；

具体的实施方法是：

1. 在  $[0, 319]$  的指令地址之间随机选取一起点  $m$ ;
2. 顺序执行一条指令，即执行地址为  $m+1$  的指令;
3. 在前地址  $[0, m+1]$  中随机选取一条指令并执行，该指令的地址为  $m_1$ ;
4. 顺序执行一条指令，其地址为  $m_1+1$ ;
5. 在后地址  $[m_1+2, 319]$  中随机选取一条指令并执行;
6. 重复上述步骤 1~5，直到执行 320 条指令。

## 详细设计

```
1  #include <bits/stdc++.h>
2
3  using namespace std;
4  #define VMEM_N 320
5
6  random_device rd;
7
8  mt19937 mt(rd());
9
10 vector<int> gen(){//生成指令序列
11     auto rall=uniform_int_distribution(0,VMEM_N-1);
12     vector<int> w;
13     w.push_back(rall(mt));
14     for (int j=0;w.size()<320;){
15         w.push_back([&]() {
16             if (j==0||j==2){
17                 return w.back()+1;
18             }else if (j==1&&0<w.back()-1){
19                 return uniform_int_distribution(0,w.back()-1)(mt);
20             }else if (j==3&&w.back()+1<VMEM_N-1){
21                 return uniform_int_distribution(w.back()+1,VMEM_N-1)(mt);
22             }else return rall(mt);
23         })%VMEM_N;
24         if (++j>=4) j=0;
25     }
26     return w;
27 }
28 namespace memscheduler{
29     constexpr int n=4;
30     struct base_memscheduler{//基本抽象页面置换调度器
31         int clocks;//当前时钟
32         array<int,n> pages;//内存中的页面
33         vector<int> v;//指令序列
34         base_memscheduler(const vector<int>& v):v(v){}
35         virtual ~base_memscheduler(){}
36         virtual int choosePage()=0;//选要置换的一页
37         virtual void usePage(int index){} //使用内存中的一页
38         virtual void loadPage(int index,int page){} //加载一页
39         pages[index]=page;
40     }
```

```

41     int deal(){//处理指令序列
42         clocks=0;
43         int cnt=0;//加载页面计数
44         pages.fill(-1);
45         while (v.size()){
46             cnt+=[&](){}
47             auto addr=v.front();v.erase(v.begin());//读取指令
48             for (int i=0;i<n;i++)//检查当前内存中是否有指令对应的页
49                 if (addr/10==pages[i]){
50                     usePage(i);
51                     return false;//无需加载
52                 }
53             for (int i=0;i<n;i++)//置换掉内存中无效页面
54                 if (i==--1){
55                     loadPage(i,addr/10);
56                     usePage(i);
57                     return true;//加载一个页面
58                 }
59             int index=choosePage();//选中即将置换掉的页
60             loadPage(index,addr/10);//加载页
61             usePage(index);//使用页
62             return true;//加载一个页面
63         }();
64         clocks++;
65     }
66     return cnt;//返回计数
67 }
68 };
69 struct OPT_memscheduler:public base_memscheduler{//最佳置换算法(OPT)
70     using base_memscheduler::base_memscheduler;
71     int choosePage(){
72         array<int,n> w;//w[i]表示内存中第i个页所指向的虚拟地址页最近使用是
           第几次
73         for (int i=0;i<n;i++)
74             w[i]=find_if(v.begin(),v.end(),[&](auto& j){
75                 return j/10==pages[i];
76             })-v.begin();
77         return max_element(w.begin(),w.end())-w.begin();//取最近最晚使用的
           页面
78     }
79 };
80 struct FIFO_memscheduler:public base_memscheduler{//先进先出置换算法(FIFO)
81     using base_memscheduler::base_memscheduler;
82     array<int,n> pageClk;//pageClk表示内存中页面上一次调入的时间戳
83     void loadPage(int index,int page){
84         pages[index]=page;
85         pageClk[index]=clocks;
86     }
87     int choosePage(){
88         return min_element(pageClk.begin(),pageClk.end())-pageClk.begin();
           //取最早调入的页面
89     }
90 };
91 struct LRU_memscheduler:public base_memscheduler{//最近最久未用置换算法(LRU)
92     using base_memscheduler::base_memscheduler;

```

```

93     array<int,n> pageClk;//pageClk表示内存中页面上一次调入的时间戳
94     void usePage(int index){
95         pageClk[index]=clocks;
96     }
97     int choosePage(){
98         return min_element(pageClk.begin(),pageClk.end())-pageClk.begin();
99         //取内存中最近最久未使用的页
100     }
101 };
102 struct CLK_memscheduler:public base_memscheduler{//时钟算法(Clock)
103     using base_memscheduler::base_memscheduler;
104     array<unsigned char,n> buffer{};int i=0;//初始化全部置0
105     void usePage(int index){
106         buffer[index]=1;//刚使用置1
107     }
108     void loadPage(int index,int page){
109         buffer[index]=1;//加载置1
110         pages[index]=page;
111     }
112     int choosePage(){
113         while (exchange(buffer[i],0)) { //旋转
114             if (++i==n) i=0;
115         }
116         return i;
117     }
118 };
119 using namespace memscheduler;
120 vector<int> mkdata(){
121     vector<unique_ptr<base_memscheduler>> l;
122     auto data=gen();
123     l.emplace_back(new OPT_memscheduler(data));
124     l.emplace_back(new FIFO_memscheduler(data));
125     l.emplace_back(new LRU_memscheduler(data));
126     l.emplace_back(new CLK_memscheduler(data));
127     vector<int> v(l.size());
128     for (int i=0;i<l.size();i++){
129         v[i]=l[i]->deal();
130     }
131     return v;
132 }
133 int main(){
134     vector<pair<int,string>> l(4,pair<int,string>{0,""});
135     l[0].second=("OPT");
136     l[1].second=("FIFO");
137     l[2].second=("LRU");
138     l[3].second=("CLK");
139     for (int i=0;i<1000;i++){
140         auto v=mkdata();
141         for (int j=0;j<v.size();j++){
142             l[j].first+=v[j];
143         }
144         for (auto& [schi,name]:l){
145             cout<<schi<<" "<<name<<endl;
146         }
147     }

```



---

## 实验结果与分析

运行 1000 次测试

```
1 115955 OPT
2 165713 FIFO
3 171546 LRU
4 150937 CLK
```

可知，性能上  $CLK < FIFO < LRU$ 。

## 小结和心得

页面置换机制允许应用程序使用比当前内存更大的内存，在页面即将调出时，页面置换算法直接决定了应用程序的运行速度，实践中应该尽量使用 CLK 算法。页面置换问题和 cache 置换问题属于同类问题。

## 实验 9-基于信号量机制的并发程序设计

### 实验目的

1. 回顾操作系统进程、线程的有关概念，针对经典的同步、互斥、死锁与饥饿问题进行并发程序设计。
2. 了解互斥体对象，利用互斥与同步操作编写读者-写者问题的并发程序，加深对 P (即 semWait)、V (即 semSignal) 原语以及利用 P、V 原语进行进程间同步与互斥操作的理解。
3. 理解 Linux 支持的信息量机制，利用 IPC 的信号量系统调用编程实现哲学家进餐问题。

### 总体设计

本次设计选择“读者优先的读者-写者问题的并发程序”。基于 POSIX 信号量编程，封装为信号量和互斥量。（虽然标准已经定义了相关设施）。使用 C++11 提供的 `std::thread` 实现线程并发。

两个等待，一个为读者等没有写者，写者等没有读者。设计两个互斥量，读者数量锁和写者锁，读者数量锁维护读者数量，当读者数量变成 0 时解写者锁。当读者数量变成 1 时尝试锁写者锁。

### 详细设计

---

```

1  #include <iostream>
2  #include <cstdio>
3  #include <random>
4  #include <thread>
5  #include <atomic>
6  #include <semaphore.h>
7  using namespace std;
8  namespace z3475{
9  struct sem{//封装信号量
10     sem_t a;
11     sem(int v=0){sem_init(&a,0,v);}
12     ~sem(){sem_destroy(&a);}
13     sem& operator++(){
14         sem_post(&a);
15         return *this;
16     }
17     sem& operator--(){
18         sem_wait(&a);
19         return *this;
20     }
21 };
22
23 struct mutex{//封装互斥量
24     sem a{1};
25     void lock(){--a;}
26     void unlock(){++a;}
27 };
28
29 struct unique_mutex{//基于作用域的上锁
30     mutex &m;
31     unique_mutex(mutex &m):m(m){m.lock();}
32     ~unique_mutex(){m.unlock();}
33 };
34
35 random_device rd;
36 mt19937 mt(rd());
37
38 void random_sleepms(int l,int r){//等待随机事件
39     this_thread::sleep_for(uniform_int_distribution(l,r)(mt)*1ms);
40 }
41
42 struct file{//抽象文件操作
43     int readers=0;
44     mutex writeMutex;
45     mutex readersMutex;
46     void process_read(int i){//执行读操作
47         printf("reader i:%d reading\n",i);
48         random_sleepms(0,1000);
49         printf("reader i:%d read done\n",i);
50     }
51     void process_write(int i){//执行写操作
52         printf("writer i:%d writing\n",i);
53         random_sleepms(0,1000);
54         printf("writer i:%d write done\n",i);
55     }
56     void read(int i){

```

---

---

```

57         {unique_mutex u(readersMutex); // 维护读者数量
58             if (!readers++)
59                 writeMutex.lock(); // 第一个读者获得写锁
60         }
61         process_read(i);
62         {unique_mutex u(readersMutex);
63             if (!--readers)
64                 writeMutex.unlock(); // 最后一个读者解除写锁
65         }
66     }
67     void write(int i){
68         unique_mutex u(writeMutex); // 获得写锁开始写
69         process_write(i);
70     }
71 };
72
73 void reader(file &f){ // 进行4次读操作
74     static atomic<int> i=0;
75     int w=i++;
76     for (int i=0;i<4;i++){
77         f.read(w);
78         random_sleepms(0,1000);
79     }
80 }
81 void writer(file &f){ // 进行2次写操作
82     static atomic<int> i=0;
83     int w=i++;
84     for (int i=0;i<2;i++){
85         f.write(w);
86         random_sleepms(0,2000);
87     }
88 }
89 }
90 using namespace z3475;
91 int main(){
92     vector<jthread> vthread;
93     file f;
94     for (int i=0;i<5;i++) // 5个读线程
95         vthread.emplace_back(jthread(reader,ref(f)));
96     for (int i=0;i<3;i++) // 3个写线程
97         vthread.emplace_back(jthread(writer,ref(f)));
98 }

```

## 实验结果与分析

### 可能的运行结果

```

1 reader i:0 reading
2 reader i:3 reading
3 reader i:2 reading
4 reader i:1 reading
5 reader i:4 reading
6 reader i:2 read done
7 reader i:4 read done

```

---

```
8 reader i:4 reading
9 reader i:0 read done
10 reader i:3 read done
11 reader i:1 read done
12 reader i:2 reading
13 reader i:1 reading
14 reader i:4 read done
15 reader i:0 reading
16 reader i:2 read done
17 reader i:3 reading
18 reader i:2 reading
19 reader i:3 read done
20 reader i:2 read done
21 reader i:4 reading
22 reader i:1 read done
23 reader i:0 read done
24 reader i:0 reading
25 reader i:4 read done
26 reader i:1 reading
27 reader i:4 reading
28 reader i:1 read done
29 reader i:3 reading
30 reader i:4 read done
31 reader i:2 reading
32 reader i:3 read done
33 reader i:1 reading
34 reader i:0 read done
35 reader i:2 read done
36 reader i:1 read done
37 writer i:1 writing
38 writer i:1 write done
39 writer i:0 writing
40 writer i:0 write done
41 writer i:2 writing
42 writer i:2 write done
43 reader i:3 reading
44 reader i:0 reading
45 reader i:0 read done
46 reader i:3 read done
47 writer i:1 writing
48 writer i:1 write done
49 writer i:0 writing
50 writer i:0 write done
51 writer i:2 writing
52 writer i:2 write done
```

可见所有写操作都得等读操作进行完毕也得等没有其他写操作。也观察到写操作明显被滞后了。

## 小结和心得

使用信号量写并发程序的一个关键在于判断出阻塞条件，如果阻塞条件是简单的生产者-消费者关系，直接用信号量本身的含义（资源数量）即可。题中写者阻塞条件是“存在读者”和“存在写者”。那么前者不能直接使用信号量本身含义，需要用互斥量保护一个维护读者数量的原子操作，这样可

---

以判断第一个读者和最后一个读者，第一个读者获得写锁，最后一个读者释放写锁即可，

## 实验 10-实现一个简单的 shell 命令行解释器

### 实验目的

本实验主要目的在于进一步学会如何在 Linux 系统下使用进程相关的系统调用，了解 shell 工作的基本原理，自己动手为 Linux 操作系统设计一个命令接口。

### 总体设计

主程序使用无限循环处理事件，以空格分割输入命令行参数。使用 fork 创建新进程。exec 系列函数执行程序。chdir 切换工作目录，opendir 判断目录是否存在。

exec 系列函数将执行进程镜像替换为对应的程序，并传入对应的参数和环境变量。如果不提供环境变量，会沿用前一个进程的环境变量。工作目录也可以沿用。

### 详细设计

```
1  #include <bits/stdc++.h>
2  #include "sys/wait.h"
3  #include <dirent.h>
4  #define FMT_HEADER_ONLY
5  #include "fmt/format.h"//:fmt:库
6  using namespace std;
7  using namespace fmt;
8
9  vector<string> prase(const string &a){//以空格解析参数
10     vector<string> v;
11     for (size_t i=0;i<a.size();){
12         while (i<a.size()&&a[i]==' ') i++;
13         int j=0;
14         while (j+i<a.size()&&a[j+i]!=' ') j++;
15         if (j>0) v.push_back(a.substr(i,j));
16         i+=j;
17     }
18     return v;
19 }
20
21 extern char **environ;
22 #define debug(...) print(__VA_ARGS__)
23 #define debug(...) ;
24 int exec(const string& file,
25         const vector<string>& args,
26         const map<string,string>& env){//以参数args，环境变量env开新进程执行
    file
```

```

27     debug("exec {}\\n",file);
28     vector<string> v;
29     vector<char*> arge(args.size()),enve(env.size());
30     for (size_t i=0;i<args.size();i++)
31         arge[i]=const_cast<char*>(args[i].c_str());
32     for (auto& [p,q]:env) v.push_back(p+"="+q);
33     for (size_t i=0;i<args.size();i++)
34         enve[i]=const_cast<char*>(v[i].c_str());
35     arge.push_back(0);
36     enve.push_back(0);
37     if (int id=fork()){//fork出一个新进程
38         wait(0);
39         return 0;
40     }
41     else{
42         return execvpe(file.c_str(),&arge[0],&enve[0]);//如果返回值为负数说明执
            行失败
43     }
44 }
45 void loop(istream& is,ostream& os,map<string,string> env){
46     string& workingdir=env["PWD"];
47     auto ls=[&]() {
48         return exec("/bin/ls",{ "ls"},env);
49     };
50     auto pwd=[&]() {
51         os<<workingdir<<endl;
52     };
53     while (1){
54         os<<format("[{}:{}]$",env["USER"],workingdir);//打印命令行前缀
55         string str;
56         getline(is,str);
57         auto args=prase(str);
58         for (auto i:args) debug("{}\\n",i);
59         auto F=[&]()->optional<int>{
60             if (args.size()>0){
61                 if (args[0]=="cd"){//切换切换工作目录
62                     if (args.size()==1){//不跟参数
63                         pwd();
64                         return 0;
65                     }else{//跟参数，执行切换工作目录
66                         string w=args[1][0]=='/'?args[1]:workingdir+"/"+args
67                             [1];
68                         if (DIR* dir=opendir(w.c_str())){
69                             chdir(w.c_str());
70                             closedir(dir);
71                             return 0;
72                         }else{
73                             os<<"Floder not exist!"<<endl;
74                             return 1;
75                         }
76                     }
77                 }else if (args[0]=="quit" ||
78                     args[0]=="exit" ||
79                     args[0]=="bye"){//推出
80                     exit(0);
81                 }else if (args[0]=="environ"){//打印环境变量

```

```

81         for (auto [p,q]:env)
82             os<<format("{}={}\n",p,q);
83         return 0;
84     }else if (args[0]=="jobs"){//打印当前执行进程
85         args[0]="ps";
86     }else if (args[0]=="help"){//打印帮助
87         os<<"帮助: "<<endl;
88         os<<"进入目录: cd [directory]"<<endl;
89         os<<"显示执行的程序: job"<<endl;
90         os<<"显示环境变量: environ"<<endl;
91         return 0;
92     }
93     if (exec(args[0],args,env)){//执行程序返回非0, 执行失败
94         os<<"Command Not Found!"<<endl;
95         exit(0);
96     }
97     return 0;
98 }
99 return {};
100 }();
101 char wd[1024];//替换环境变量PWD
102 if (getcwd(wd,1024))
103     workingdir=wd;
104 else{
105     os<<"Too long working path"<<endl;
106     exit(-1);
107 }
108 if (F){
109 }else{
110     os<<"Input Fail"<<endl;
111 }
112 }
113 }
114
115 int main(int argc,char *argv[]){
116     vector<string> args{argv,argv+argc};
117     map<string,string> env;
118     for (char **c=environ;*c&&**c;c++){//解析环境变量
119         string line=*c;
120         int w=line.find("=");
121         env[line.substr(0,w)]=line.substr(w+1);
122     }
123     loop(cin,cout,move(env));//开始事件循环
124 }

```

## 实验结果与分析

```

1 [z3475:/tmp]$cd fuck
2 [z3475:/tmp/fuck]$ls
3 a.cpp
4 [z3475:/tmp/fuck]$cat a.cpp
5 #include <bits/stdc++.h>
6
7 using namespace std;

```

---

```
8
9  int main(){
10      int a,b;
11      cin>>a>>b;
12      cout<<a+b<<endl;
13  }
14  [z3475:/tmp/fuck]$clang++ a.cpp -o a
15  [z3475:/tmp/fuck]$./a
16  100 -10
17  90
18  [z3475:/tmp/fuck]$enviro
19  ALL_PROXY=socks5://127.0.0.1:7891
20  COLORFGBG=15;0
21  COLORTERM=truecolor
22  CPLUS_INCLUDE_PATH=/home/z3475/ACM/ATL/base/include
23  DBUS_SESSION_BUS_ADDRESS=unix:path=/run/user/1000/bus
24  DISPLAY=:0
25  FTP_PROXY=http://127.0.0.1:7890
26  ** 略 **
27  [z3475:/tmp/fuck]$exit
```

成功实现基本的 shell

## 小结和心得

Linux 为用户提供了许多方便编写应用程序的 API，设计简单性能高。应用程序开发者能够轻松和操作系统交互，快速编写出运行在 Linux 上的应用程序。