湖南科技大学计算机科学与工程学院

操作系统 课程设计报告

**专业班级：** 22计科六班

**姓 名：** 周俊哲

**学 号：** 2205010711

**指导教师：** 黄卫红

**时 间**：

**地 点**：

|  |
| --- |
| **指导教师评语：**  **成绩： 等级：**  **签名：**  **年 月 日** |

**实验一**

1. **实验题目：Windows进程管理**

**二、实验目的**

（1）学会使用 VC 编写基本的 Win32 Consol Application（控制台应用程序)。

（2）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”。

（3）通过阅读和分析实验程序，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。

**三、实验内容**

**3.1实验内容和步骤**

**（1）编写基本的 Win32 Consol Application**

**步骤 1**：登录进入 Windows 系统，启动 VC++ 6.0。

**步骤 2**：在“FILE”菜单中单击“NEW”子菜单，在“projects”选项卡中选择“Win32 Consol Application”,然后在“Project name”处输入工程名，“Location” 处输入工程目录。创建一个新的控 制台应用程序工程。

**步骤 3**：在“FILE”菜单中单击“NEW”子菜单，在“Files”选项卡中选择“C++ Source File”, 然后在“File”处输入 C/C++源程序的文件名。

**步骤 4**：将清单 2-1 所示的程序清单复制到新创建的 C/C++源程序中。编译成可执行文件。

**步骤 5**：在“开始”菜单中单击“程序”-“附件”-“命令提示符”命令，进入 Windows“命令提示符”窗口，然后进入工程目录中的 debug 子目录，执行编译好的可执行程序，列出运行结果 (如果运行不成功，则可能的原因是什么？)

**（2） 创建进程**

本实验显示了创建子进程的基本框架。该程序只是再一次地启动自身，显示它的系统进程 ID和它在进程列表中的位置。

**步骤 1**：创建一个“Win32 Consol Application”工程，然后拷贝清单 2-2 中的程序，编译成可执行文件。

**步骤 2**：在“命令提示符”窗口运行步骤 1 中生成的可执行文件，列出运行结果。按下 ctrl+alt+del，调用 windows 的任务管理器，记录进程相关的行为属性。

**步骤 3**：在“命令提示符”窗口加入参数重新运行生成的可执行文件，列出运行结果。按下ctrl+alt+del，调用 windows 的任务管理器，记录进程相关的行为属性。

**步骤 4**：修改清单 2-2 中的程序，将 nClone 的定义和初始化方法按程序注释中的修改方法进行修改，编译成可执行文件（执行前请先保存已经完成的工作）。再按步骤 2 中的方式运行，看看结果会有什么不一样。列出行结果。从中你可以得出什么结论？说明 nClone 的作用。变量的定义和初始化方法（位置）对程序的执行结果有影响吗？为什么？

**（3） 父子进程的简单通信及终止进程**

**步骤 1：**创建一个“Win32 Consol Application”工程，然后拷贝清单 2-3 中的程序，编译成可执行文件。

**步骤 2：**在 VC 的工具栏单击“Execute Program”(执行程序) 按钮，或者按 Ctrl + F5 键，或者在“命令提示符”窗口运行步骤 1 中生成的可执行文件，列出运行结果。

**步骤 3：**按源程序中注释中的提示，修改源程序 2-3，编译执行（执行前请先保存已经完成的工作），列出运行结果。在程序中加入跟踪语句，或调试运行程序，同时参考 MSDN 中的帮助文件

CreateProcess()的使用方法，理解父子进程如何传递参数。给出程序执行过程的大概描述。

**步骤 4：**按源程序中注释中的提示，修改源程序 2-3，编译执行，列出运行结果。

**步 骤 5 ：** 参 考 MSDN 中 的 帮 助 文 件 CreateMutex()、 OpenMutex()、 ReleaseMutex()和WaitForSingleObject()的使用方法，理解父子进程如何利用互斥体进行同步的。给出父子进程同步过程的一个大概描述

**3.2关键代码**

// 创建当前进程的克隆进程的简单方法

void StartClone(){

    TCHAR szFilename[MAX\_PATH] ;

    GetModuleFileName(NULL, szFilename, MAX\_PATH) ;

    TCHAR szCmdLine[MAX\_PATH] ;

    sprintf(szCmdLine, "\"%s\" child" , szFilename) ;

    STARTUPINFO si;

    ZeroMemory(&si,sizeof(si)) ;

    si.cb = sizeof(si) ;

    PROCESS\_INFORMATION pi;

    BOOL bCreateOK=CreateProcess(

        szFilename,             // 产生的应用程序的名称 (本EXE文件)

        szCmdLine,              // 告诉我们这是一个子进程的标志

        NULL,                   // 用于进程的缺省的安全性

        NULL,                   // 用于线程的缺省安全性

        FALSE,                  // 不继承句柄

        CREATE\_NEW\_CONSOLE,     //创建新窗口

        NULL,                   // 新环境

        NULL,                   // 当前目录

        &si,                    // 启动信息结构

        &pi ) ;                 // 返回的进程信息

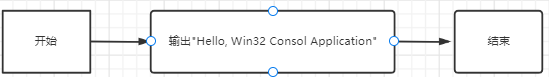
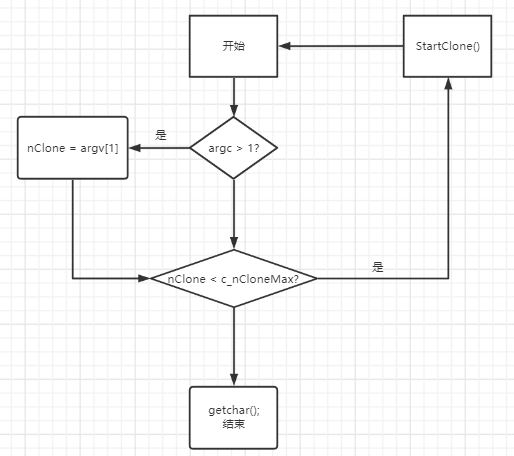
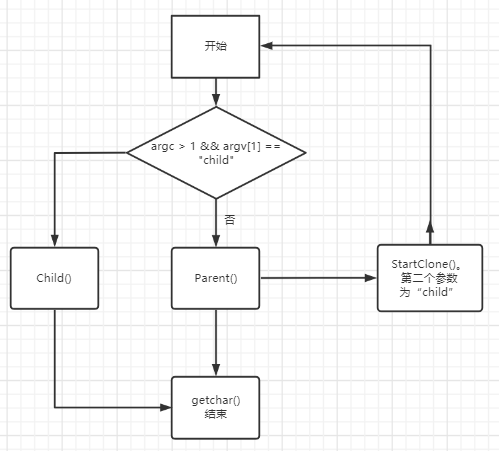
    if (bCreateOK){

        CloseHandle(pi.hProcess) ;

        CloseHandle(pi.hThread) ;

}}

**3.3程序流程图**

1. 
2. 
3. 
4. **实验结果与分析**

（1）编写基本的 Win32 Consol Application

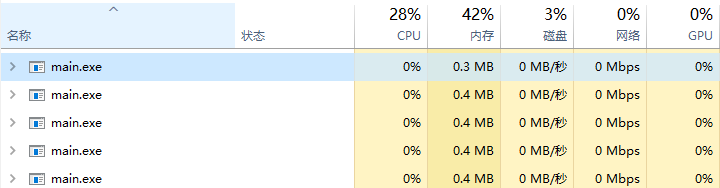


1. 创建进程

步骤2：cmd中运行结果：



任务管理器：



进程：

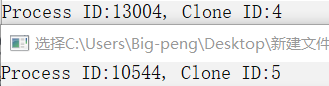


步骤3：

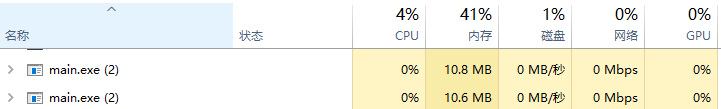
cmd：



进程：



任务管理器：



步骤4：

第一次修改不会对程序造成影响，第二次修改后会不断的弹出新的进程，因为修改后nClone永远为0。

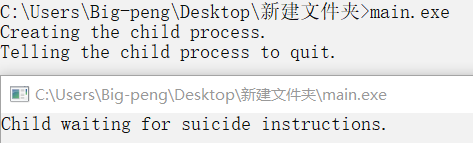
nClone的作用：指明当前进程的ID，能决定创建进程的数量。

变量的定义和初始化方法（位置）对程序的执行结果有影响吗？为什么？

答：有影响。当给变量赋值在变量初始化之前时，初始化的值会覆盖新赋的值，可能导致程序出错。

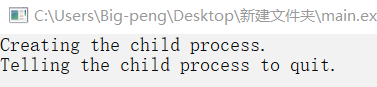
1. 父子进程的简单通信及终止进程

步骤2：运行结果:



步骤3：

会不断的生成父进程。



步骤4：

只会生成一个父进程，因为设置了子进程的等待时间为0。

步骤5：

父进程通过CreateMutex()创建一个互斥程序体，子进程通过OpenMutex（）打开互斥体，通过WaitForSingleObject（）进入阻塞状态，等待父进程通过互斥体发来的信号，当父进程调用ReleaseMutex（）释放互斥体的所有权时，这个信号会发送给子进程。

**五、小结与心得体会**

通过这个题目的练习，我学会了使用c++编写基本的控制台应用程序。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”。并且通过阅读和分析实验程序，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。

**实验二**

**一、实验题目：Linux进程控制**

**二、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。

**三、实验内容**

**3.1 运用的理论知识**

在 Linux 中创建子进程要使用 fork()函数，执行新的命令要使用 exec()系列函数，等待子进程结束使用 wait()函数，结束终止进程使用 exit()函数。

fork()原型如下：pid\_t fork(void);

fork 建立一个子进程，父进程继续运行，子进程在同样的位置执行同样的程序。对于父进程，fork()返回子进程的 pid, 对于子进程，fork()返回 0。出错时返回-1。

exec 系列函数用新的进程映象置换当前的进程映象.这些函数的第一个参数是待执行程序的路径名(文件名)。这些函数调用成功后不会返回,其进程的正文(text),数据(data)和栈(stack)段被待执行程序程序覆盖。但是进程的 PID所有打开的文件描述符没有改变,同时悬挂信号被清除，信号重置为缺省行为。

当进程调用 wait，它将进入睡眠状态直到有一个子进程结束。wait 函数返回子进程的进程 id，stat\_loc 中返回子进程的退出状态。

**3.2实验内容和步骤**

**（1）进程的创建：**

编写一段程序，使用系统调用 fork（）创建一个子进程。当此程序运行时，在系统中有一个父进程和一个子进程活动。让每一个进程在屏幕上分别显示字符：父进程显示字符“b”；子进程显示字符“a”，另外父子进程都显示字符“c”。

**步骤 1：**使用 vi 或 gedit 新建一个 fork\_demo.c 程序，然后拷贝清单3-1 中的程序，使用 cc 或者gcc 编译成可执行文件 fork\_demo。例如，可以使用 gcc –o fork\_demo fork\_demo.c 完成编译。

**步骤 2：**在命令行输入./fork\_demo 运行该程序。

**步骤 3：**多次运行程序，观察屏幕上的显示结果，并分析多次运行为什么会出现不同的结果。

1. **子进程执行新任务：**

编写一段程序，使用系统调用fork ()创建一个子进程。子进程通过系统调用exec更换自己原有的执行代码，转去执行Linux 命令/bin/ls(显示当前目录的列表)，然后调用exit () 函数结束。父进程则调用waitpid()等待子进程结束，并在子进程结束后显示子进程的标识符，然后正常结束。

**步骤1:**使用vi或gedit新建一个exec\_demo.c程序，然后拷贝清单3-2中的程序（该程序的执行如图3-1所示),使用cc或者 gcc编译成可执行文件exec\_demo。例如,可以使用gee-o exec\_demoexec\_demo.c完成编译。

**步骤2:**在命令行输入./exec\_demo运行该程序。步康3:观察该程序在屏幕上的显示结果，并分析。

**步骤3：**观察该程序在屏幕上的显示结果，并分析。

**3.3关键代码**

（1）

int x;

  srand((unsigned)time(NULL));

  while((x=fork())==-1);

  if (x==0)

  {

    sleep(rand() % 2);

    printf("a");

  }

  else

  {

    sleep(rand() % 3);

    printf("b");

  }

  printf("c");

（2）

  pid\_t pid;

  pid = fork();

  if (pid < 0)

  { /\* error occurred \*/

    fprintf(stderr, "Fork Failed");

    return 1;

  }

  else if (pid == 0)

  { /\* 子进程 \*/

    execlp("/bin/ls","ls",NULL);

  }

  else { /\* 父进程 \*/

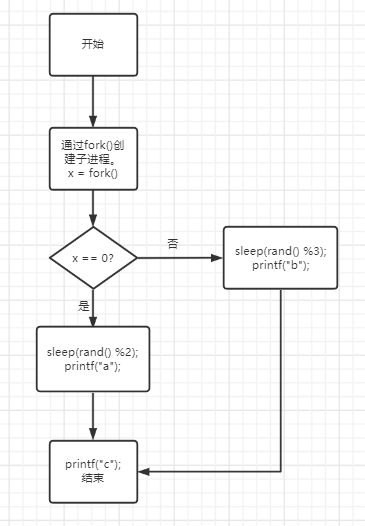
  /\* 父进程将一直等待，直到子进程运行完毕\*/

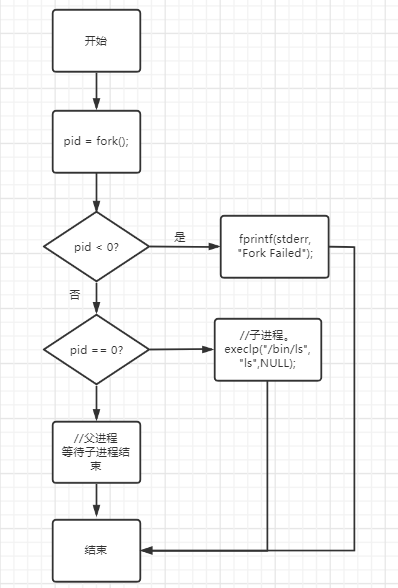
    wait(NULL);

    printf("Child Complete");

  }

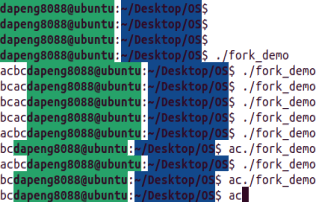
**3.4流程图**

1）

2）

**四、实验结果与分析**

（1）进程的创建：



为什么多次运行出现不同的结果？

答：因为子进程和父进程中都有休眠语句，不过休眠时间是随机的。

（2）子进程执行新任务：



父进程运行到fork（）处后，父进程执行wait（）等待子进程结束，子进程通过exec（）去执行Linux命令，然后调用exit（）结束。

**五、小结与心得体会**

通过本题，我学会了在Linux环境下进程的创建、撤销和运行，加深了对进程概念和进程并发执行的理解，明确了进程和程序之间的区别。

**实验三**

**一、实验题目：Linux进程间通信**

**二、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地交换数据，通过本实验，理解熟悉 Linux 支持的消息通信机制。

**三、实验内容**

**3.1背景知识**

UINX/Linux 系统把信号量、消息队列和共享资源统称为进程间通信资源(IPC resource)。提供给用户的 IPC 资源是通过一组系统调用实现的。这组系统调用为用户态进程提供了以下三种服务：

 用信号量对进程要访问的临界资源进行保护。

 用消息队列在进程间以异步方式发送消息。

 用一块预留出的内存区域供进程之间交换数据。

创建 IPC 资源的系统调用有：

 semget()—获得信号量的 IPC 标识符。

 msgget()—获得消息队列的 IPC 标识符。

 shmget()—获得共享内存的 IPC 标识符。

控制 IPC 资源的系统调用有：

 semctl()—对信号量资源进行控制的函数。

 msgctl()—对消息队列进行控制的函数。

 shmctl()—对共享内存进行控制的函数。

上述函数为获得和设置资源的状态信息提供了一些命令。例如：

 IPC\_SET 命令：设置属主的用户标识符和组标识符。

 IPC\_STAT 和 IPC\_INFO 命令：获得资源状态信息。

 IPC\_RMID 命令：释放这个资源。

操作 IPC 资源的系统调用有：

 semop()—获得或释放一个 IPC 信号量。 可以实现 P、V 操作

 msgsnd()—发送一个 IPC 消息。

 msgrcv()—接收一个 IPC 消息。

 shmat()—将一个 IPC 共享内存段添加到 进程的地址空间

 shmdt()——将 IPC 共享内存段从私有的地址空间剥离。

**3.2实验内容和步骤**

**消息的创建、发送和接收的程序设计：**

使用系统调用 msgget()，msgsnd()，msgrcv()及 msgctl()编制一长度为 1K 的消息的发送和接收程序。观察参考程序，说明控制消息队列系统调用 msgctl()在此起什么作用？

（1） 为了便于操作和观察结果，用一个程序作为“引子”，先后 fork()两个子进程 SERVER和 CLIENT，进行通信。

（2） SERVER 端建立一个 key 为 75 的消息队列，等待其他进程发来的消息。当遇到类型为 1 的消息，则作为结束信号，取消该队列，并退出 SERVER。SERVER 每接收到一个消息后显示一句“（server） received”。

（3） CLIENT 端使用 key 为 75 的消息队列，先后发送类型从 10 到 1 的消息，然后退出。最后的一个消息，即是 SERVER 端需要的结束信号。CLIENT 每发送一条消息后显示一句“(client)sent”。

（4） 父进程在 SERVER 和 CLIENT 均退出后结束。

**3.3关键代码**

struct msgform{

long mtype;

char mtext[1030];

}msg;

int msgqid,i;

void CLIENT(){

int i;

msgqid=msgget(MSGKEY,0777);

for (i=10;i>=1;i--)

{

msg.mtype=i;

printf("(client) sent \n");

msgsnd(msgqid,&msg,1024,0);

}

exit(0);

}

void SERVER(){

msgqid=msgget(MSGKEY,0777|IPC\_CREAT);

do{

msgrcv(msgqid,&msg,1030,0,0);

printf("(Server) recieved\n");

} while(msg.mtype!=1);

msgctl(msgqid,IPC\_RMID,0);

exit(0);

}

void main()

{

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

if(!i) SERVER();

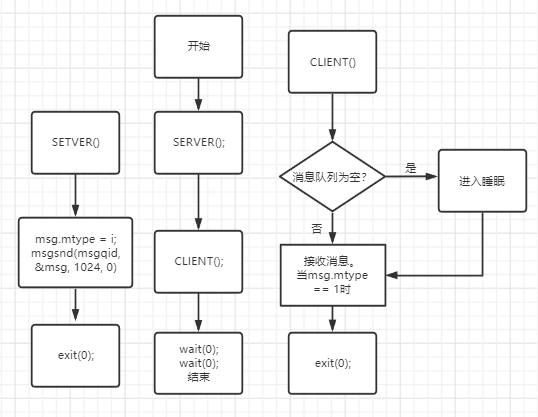
while((i=fork())==-1);

if(!i) CLIENT();

wait(0);

}

**3.4流程图**



**四、实验结果与分析**



从理论上说，上述程序应当是每当 client 发送一条消息后，server 接收该消息，client 再发送下一条，也即是应该交替出现“（client）sent”和“（server）received”，但实际结果大多不是这样，会出现几个“（client）sent”连续后再几个“（server）received”，请分析原因。

答：消息在client和server之间的传递并不是完全同步的当消息队列为空时，server进程进入睡眠，当client发送信息到消息队列中时，并不会通知server，所以server很可能继续睡眠。

**五、小结与心得体会**

通过本题，理解熟悉了Linux支持的消息通信机制。

**实验四**

**一、实验题目：Windows线程的互斥与同步**

**二、实验目的**

(1) 回顾操作系统进程、线程的有关概念，加深对 Windows 线程的理解。

(2) 了解互斥体对象，利用互斥与同步操作编写生产者-消费者问题的并发程序，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。

**三、实验内容**

**3.1实验内容和步骤：**

**生产者消费者问题**

**步骤 1**：创建一个“Win32 Consol Application”工程，然后拷贝清单 5-1 中的程序，编译成可执行文件。

**步骤 2**：在“命令提示符”窗口运行步骤 1 中生成的可执行文件，列出运行结果。

**步骤 3**：仔细阅读源程序，找出创建线程的 WINDOWS API 函数，回答下列问题：线程的第一个执行函数是什么（从哪里开始执行）？它位于创建线程的 API 函数的第几个参数中？

**步骤 4**：修改清单 5-1 中的程序，调整生产者线程和消费者线程的个数，使得消费者数目大与生产者，看看结果有何不同。察看运行结果，从中你可以得出什么结论？

**步骤 5**：修改清单 5-1 中的程序，按程序注释中的说明修改信号量 EmptySemaphore 的初始化方法，看看结果有何不同。

**步骤 6**：根据步骤 4 的结果，并查看 MSDN，回答下列问题：

1）CreateMutex 中有几个参数，各代表什么含义。

2）CreateSemaphore 中有几个参数，各代表什么含义，信号量的初值在第几个参数中。

3）程序中 P、V 原语所对应的实际 Windows API 函数是什么，写出这几条语句。

4）CreateMutex 能用 CreateSemaphore 替代吗？尝试修改程序 5-1，将信号量 Mutex 完全用

CreateSemaphore 及相关函数实现。写出要修改的语句。

**3.2 关键代码**

//生产者

DWORD WINAPI Producer(LPVOID lpPara)

{

while(p\_ccontinue){

WaitForSingleObject(EmptySemaphore,INFINITE); //p(empty);

WaitForSingleObject(Mutex,INFINITE); //p(mutex);

Produce();

Append();

Sleep(1500);

ReleaseMutex(Mutex); //V(mutex);

ReleaseSemaphore(FullSemaphore,1,NULL); //V(full);

}

return 0;

}

//消费者

DWORD WINAPI Consumer(LPVOID lpPara)

{

while(p\_ccontinue){

WaitForSingleObject(FullSemaphore,INFINITE); //P(full);

WaitForSingleObject(Mutex,INFINITE); //P(mutex);

Take();

Consume();

Sleep(1500);

ReleaseMutex(Mutex); //V(mutex);

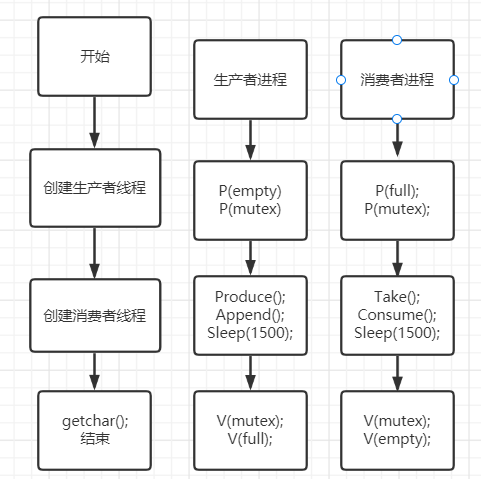
ReleaseSemaphore(EmptySemaphore,1,NULL); //V(empty);

}

return 0;

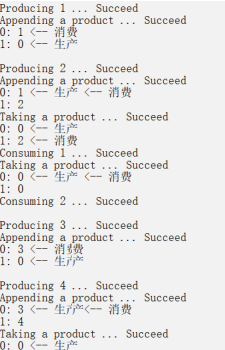
}

**3.3程序流程图**



**四、实验结果与分析**

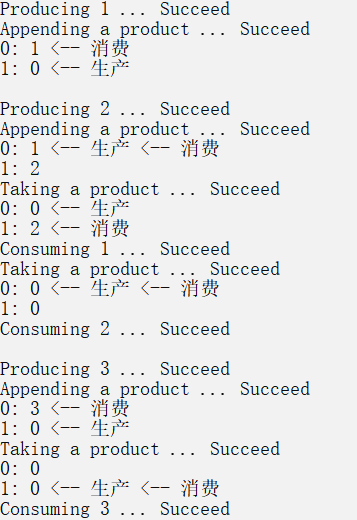
**步骤2：**运行结果



**步骤3：**

答：线程的第一个执行函数是Producer（），位于创建线程API函数的第三个参数中。

**步骤4：**运行结果



可以看出当生产者个数多于消费者个数时，生产速度快，生产者经常等待消费者；反之，消费者经常等待。

**步骤5：**

答：不输出任何结果，因为EmptySemaphore的初始值被设置为0，代表缓冲区空位为0，所以生产者等待，而缓冲区此时为空，消费者也等待，所以不输出任何结果。

**步骤6：**

答：1）有三个参数，分别是指向安全属性的指针、初始化互斥对象的所有者、指向互斥对象名的指针。

1. 有四个参数，分别是指向SECURITY\_ATTRIBUTES结构的指针、信号量对象的初值、信号量对象的最大值、信号量对象的名称。
2. WaitForSingleObject(EmptySemaphore,INFINITE); //p(empty);

WaitForSingleObject(Mutex,INFINITE); //p(mutex);

ReleaseMutex(Mutex); //V(mutex);

ReleaseSemaphore(FullSemaphore,1,NULL); //V(full);

1. 可以。

Mutex = CreateMutex(NULL,FALSE,NULL);

改为：Mutex = CreateSemaphore(NULL, 1, 1, NULL);

ReleaseMutex(Mutex);

改为：ReleaseSemaphore(Mutex,1,NULL);

**五、小结与心得体会**

通过本题，我回顾了操作系统进程、线程的有关概念，加深了对Windows线程的理解。了解了互斥体对象，并利用了互斥与同步操作编写生产者-消费者问题的并发程序，加深对P（semWait）、V(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。

**实验五**

1. **实验题目：内存管理**
2. **实验目的**

了解 Windows 的内存结构和虚拟内存的管理，理解进程的虚拟内存空间和物理内存的映射关

**三、 实验内容和步骤**

**3.1背景知识**

请参考教材《操作系统精髓与设计原理（中文第九版）》P240 8.5 Windows 内存管理。

Windows Xp 是 32 位的操作系统，它使计算机 CPU 可以用 32 位地址对 32 位内存块进行操作。

内存中的每一个字节都可以用一个 32 位的指针来寻址。这样，最大的存储空间就是 2 32字节或 4G

字节。这样，在 Windows 下运行的每一个应用程序最大可能占有 4GB 大小的空间。

然而，实际上每个进程一般不会占有 4GB 内存。Windows 在幕后将虚拟内存 (virtual memory，

VM) 地址映射到了各进程的物理内存地址上。而所谓物理内存是指计算机的 RAM 和由 Windows

分配到用户驱动器根目录上的换页文件。物理内存完全由系统管理。

在 Windows 环境下，4GB 的虚拟地址空间被划分成两个部分：低端 2GB 提供给进程使用，高

端 2GB 提供给系统使用。这意味着用户的应用程序代码，包括 DLL 以及进程使用的各种数据等，

都装在用户进程地址空间内 (低端 2GB) 。用户进程的虚拟地址空间也被分成三部分：

1) 虚拟内存的已调配区 (committed) ：具有备用的物理内存，根据该区域设定的访问权限，用

户可以进行写、读或在其中执行程序等操作。

2) 虚拟内存的保留区 (reserved) ：没有备用的物理内存，但有一定的访问权限。

3) 虚拟内存的自由区 (free) ：不限定其用途，有相应的 PAGE\_NOACCESS 权限。

与虚拟内存区相关的访问权限告知系统进程可在内存中进行何种类型的操作。例如，用户不能

在只有 PAGE\_READONLY 权限的区域上进行写操作或执行程序；也不能在只有 PAGE\_EXECUTE

权限的区域里进行读、写操作。而具有 PAGE\_ NOACCESS 权限的特殊区域，则意味着不允许进程

对其地址进行任何操作。

在进程装入之前，整个虚拟内存的地址空间都被设置为只有 PAGE\_NOACCESS 权限的自由区

域。当系统装入进程代码和数据后，才将内存地址的空间标记为已调配区或保留区，并将诸如

EXECUTE、READWRITE 和 READONLY 的权限与这些区域相关联。

**3.2实验内容和步骤**

**了解和检测进程的虚拟内存空间。**

**步骤 1：**创建一个“Win32 Consol Application”工程，拷贝清单 6-1 中的程序，编译成可执行文件。

**步骤 2：**在 VC 的工具栏单击“Execute Program”(执行程序) 按钮，或者按 Ctrl + F5 键，或者在“命

令提示符”窗口运行步骤 1 中生成的可执行文件。

**步骤 3：**根据运行结果，回答下列问题。

为什么显示的地址都以 3 个 0 结尾：\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

虚拟内存每页大小为：\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

应用程序有权限访问（读或写）的最小地址：\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

应用程序有权限访问（读或写）的最大地址：\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

应用程序有权限访问（读或写）的虚拟地址空间大小为：\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_24

当前计算机的实际内存大小为：\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

通过网络搜索自主学习，请自行组织语言，指出已调配区 (committed)、保留区 (reserved) 、

自由区 (free) 三者之间的区别：\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_。

**3.3关键代码**

void ShowVirtualMemory()

{

// 首先，让我们获得系统信息

SYSTEM\_INFO si;

:: ZeroMemory(&si, sizeof(si) ) ;

:: GetSystemInfo(&si) ;

// 使用外壳辅助程序对一些尺寸进行格式化

TCHAR szPageSize[MAX\_PATH];

::StrFormatByteSize(si.dwPageSize, szPageSize, MAX\_PATH) ;

DWORD dwMemSize = (DWORD)si.lpMaximumApplicationAddress -

(DWORD) si.lpMinimumApplicationAddress;

TCHAR szMemSize [MAX\_PATH] ;

:: StrFormatByteSize(dwMemSize, szMemSize, MAX\_PATH) ;

// 将内存信息显示出来

std :: cout << "Virtual memory page size: " << szPageSize << std :: endl;

std :: cout.fill ('0') ;

std :: cout << "Minimum application address: 0x"

<< std :: hex << std :: setw(8)

<< (DWORD) si.lpMinimumApplicationAddress

<< std :: endl;

std :: cout << "Maximum application address: 0x"

<< std :: hex << std :: setw(8)28

<< (DWORD) si.lpMaximumApplicationAddress

<< std :: endl;

std :: cout << "Total available virtual memory: "

<< szMemSize << std :: endl ;

}

void main()

{

//显示虚拟内存的基本信息

ShowVirtualMemory();

// 遍历当前进程的虚拟内存

::WalkVM(::GetCurrentProcess());

}}

**四、 实验结果与分析**

**步骤2：**运行结果



**步骤3：**

1. 因为地址通常以页为单位来管理，而页的大小通常是4KB（4096字节），这意味着地址必须是4096的倍数，因此地址的最后三位是零
2. 4KB
3. 0x00000000
4. 0Xffffffff
5. 0xFFFFFFFF−0x00000000+1=4,294,967,296 字节，约为 4 GB。
6. 15.6GB
7. **调配区（Committed）**：

表示已分配并可以使用的内存，如 "Committed, READONLY, Private"。

**保留区（Reserved）**：

表示已保留但未分配的内存，如 "Reserved, READONLY, Private"。

**自由区（Free）**：

表示未使用的内存，如 "Free, NOACCESS"。

Windows 操作系统的虚拟内存管理方案旨在提供每个进程一个独立的、连续的虚拟地址空间，同时有效利用物理内存和磁盘存储。其主要特点和机制如下：

1. **虚拟地址空间**：
   * 每个进程都有一个独立的虚拟地址空间。对于32位系统，这个地址空间通常为4GB；对于64位系统，这个地址空间可以达到非常大的值，具体大小依赖于操作系统和硬件配置。
2. **分页机制**：
   * 虚拟内存被分为固定大小的页面（通常为4KB）。操作系统通过页面表将虚拟地址映射到物理地址。每个页面可能被存储在物理内存或磁盘上的页面文件中。
3. **地址空间分布**：
   * 虚拟地址空间通常分为用户空间和内核空间。在32位系统中，通常是2GB用户空间和2GB内核空间；在64位系统中，用户空间和内核空间的比例可以根据配置不同有所变化。
4. **页面状态**：
   * **自由（Free）**：未使用的内存，应用程序可以请求使用。
   * **保留（Reserved）**：已保留但未分配实际物理内存，防止其他应用程序使用这段地址空间。
   * **提交（Committed）**：已分配物理内存或页面文件，应用程序可以使用这些内存。
5. **内存保护**：
   * Windows 提供多种内存保护机制，以确保进程之间的隔离和内存访问的安全性。常见的保护类型包括只读、可写和可执行。
6. **页面文件**：
   * 当物理内存不足时，Windows 会将不常用的页面移到磁盘上的页面文件中。这种技术称为交换（swapping）或分页（paging），可以扩展可用内存的容量。
7. **内存分配函数**：
   * Windows 提供了一些内存管理函数，如 VirtualAlloc、VirtualFree、VirtualProtect，用于申请、释放和保护内存。
8. **工作集管理**：
   * 操作系统维护每个进程的工作集（在物理内存中的页面集合）。系统会根据进程的需求和物理内存的压力动态调整每个进程的工作集大小。

通过这些机制，Windows 能够为每个进程提供一个独立且受保护的虚拟地址空间，同时高效地管理和利用系统的物理内存和磁盘资源。

**五、小结与心得体会**

虚拟内存管理是操作系统的重要组成部分，通过学习这个机制，我对操作系统的整体工作原理有了更深入的了解。

**实验六**

**一、实验题目：银行家算法的模拟与实现**

**二、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。

**三、总体设计**

**3.1基本原理与算法**

银行家算法由荷兰学者Dijksra为银行系统设计的，以确保银行在发行现金贷款时，不会发生不能满足所有顾客需要的情况。后来该算法被用在操作系统中，用来避免死锁。

核心思想：在进程提出资源申请时，先预判此次分配是否会导致系统进入不安全状态。如果会进入不安全状态，就暂时不答应这次请求，让该进程先阻塞等待。

**3.2模块介绍**

void init();     //初始化state

void m\_print(); //输出state

int distribute();  //分配资源

int safe();     //判断分配完后是否为安全状态

void recycle();  //当不安全时回收分配出的资源

**3.3设计步骤**

银行家算法步骤：

①检查此次申请是否超过了之前声明的最大需求数

②检查此时系统剩余的可用资源是否还能满足这次请求

③试探着分配，更改各数据结构

④用安全性算法检查此次分配是否会导致系统进入不安全状态

安全性算法步骤：

检查当前的剩余可用资源是否能满足某个进程的最大需求，如果可以，就把该进程加入安全序列，并把该进程持有的资源全部回收。不断重复上述过程，看最终是否能让所有进程都加入安全序列。

**四、详细设计（含主要的数据结构、程序流程图、关键代码等）**

**4.1主要的数据结构**

struct state {

int resource[m]; //表示 m 种资源的总量

int available[m]; //表示未分配的各种可用资源数量

int claim[n][m]; //表示 n 个进程对 m 类资源的最大需求

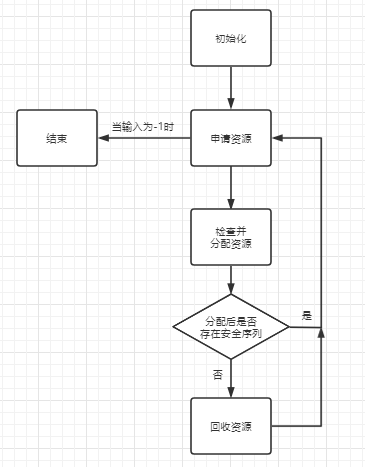
int alloc[n][m]; //表示 n 个进程已分配的各种资源数

int need[n][m]; //表示n个进程仍需要的各种资源数

} state;

int request[m + 1]; //表示申请的各类资源数

**4.2程序流程图**



**4.3关键代码**

//分配

//检查申请加已分配的资源是否超过之前声明的最大需求

//检查此时系统剩余的可用资源是否还能满足这次请求

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

if(state.alloc[index][i] + request[i+1] > state.claim[index][i] || state.available[i] < request[i+1] || request[i+1] < 0) {

cout << "申请+已分配的资源超过之前声明的最大需求" << endl;

cout << "或，此时系统剩余的可用资源是否还能满足这次请求" << endl;

cout << "或，申请的资源数小于0" << endl;

return 0;

}

}

//分配资源，更改各数据结构

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

state.available[i] -= request[i+1];

state.alloc[index][i] += request[i+1];

state.need[index][i] -= request[i+1];

}

//检查分配后系统是否安全

//len:安全序列长度；flag0=0表示没有能加入安全序列的进程

//flag=1表示进程可以加入安全序列

while(len < n) {

flag0 = 0;

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

if(safe\_list[i]) continue;

flag = 1;

for(j = 0; j < m; j++) {

if(state.need[i][j] > remaining[j]) {

flag = 0;

break;

}

}

if(flag) {

for(j = 0; j < m; j++) {

remaining[j] += state.alloc[i][j];

}

safe\_list[i] = 1;

len++;

flag0 = 1;

}

}

if(!flag0)

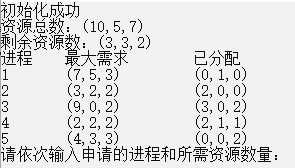
return 0;

}

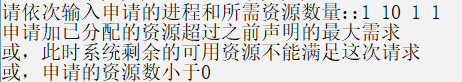
return 1;

1. **实验结果与分析**

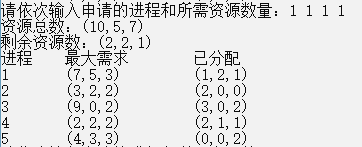
初始化：



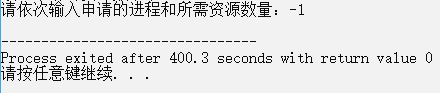
分配资源（不能分配或分配后系统状态不安全）：



分配资源（分配后系统状态任然是安全的）：



结束程序：



1. **小结与心得体会**

通过这个实验，我进一步了解了进程的并发执行，加强了对进程死锁的理解，理解了安全状态与不安全状态的概念。并且掌握了使用银行家算法避免死锁问题。

**实验七**

1. **实验题目：磁盘调度算法的模拟与实现**

**二、实验目的**

(1) 了解磁盘结构以及磁盘上数据的组织方式。

(2) 掌握磁盘访问时间的计算方式。

(3) 掌握常用磁盘调度算法及其相关特性。

**三、总体设计**

**3.1基本原理与算法**

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。磁盘调度算法影响的是移动磁头所花的事件。

**FIFO（先进先出算法）**：按访问请求到达的先后次序进行调度。

**SSTF（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。

**3.2模块介绍S**

void init(); //初始化

void FIFO(); //先进先出算法

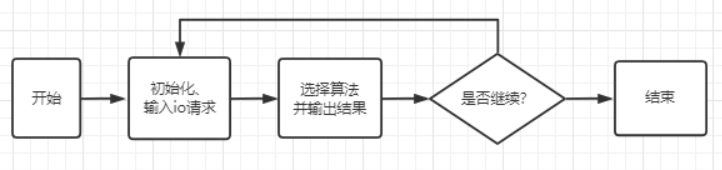
void SSTF(); //最短服务时间优先算法

void SCAN(); //电梯算法

void C\_SCAN(); //循环扫描算法

1. **详细设计**

**4.1程序流程图**



**4.2关键代码**

//FIFO

for(i = 0; i < io\_len; i++) {

len = abs(now - io[i]);

now = io[i];

sum += len;

printf("%d\t%d\n", io[i], len);

}

//SSTF

for(i = 0; i < io\_len; i++) {

len = abs(now - io[i]);

now = io[i];

sum += len;

printf("%d\t%d\n", io[i], len);

}

//SCAN

for(i = now\_index; i < io\_len; i++) {

len = io[i] - now;

now = io[i];

sum += len;

printf("%d\t\t\t%d\n", io[i], len);

}

for(i = now\_index-1; i >= 0; i--) {

len = now - io[i];

now = io[i];

sum += len;

printf("%d\t\t\t%d\n", io[i], len);

}

//C\_SCAN

for(i = now\_index; i < io\_len; i++) {

len = io[i] - now;

now = io[i];

sum += len;

printf("%d\t\t\t%d\n", io[i], len);

}

for(i = now\_index-1; i >= 0; i--) {

len = now - io[i];

now = io[i];

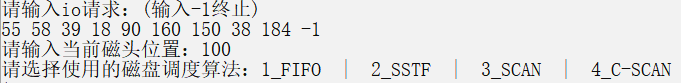
sum += len;

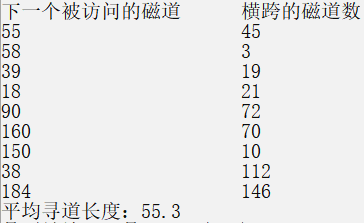
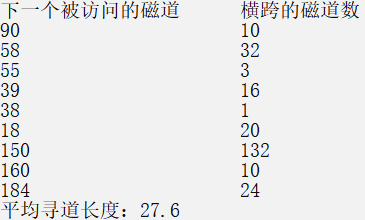
printf("%d\t\t\t%d\n", io[i], len);

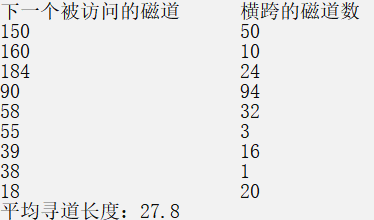
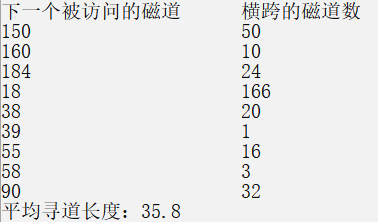
}

1. **实验结果与分析**

**5.1实验结果**



FIFO  SSTF 

SCAN  C\_SCAN

**5.2分析**

FIFO：公平，但是当访问的磁道分散时，算法性能很差。

SSTF：性能好，但会产生饥饿。

SCAN：性能较好且不会产生饥饿，但对于各个磁道的响应频率不平均。

C-SCAN：与SCAN相比，各个磁道的响应频率很平均，但平均寻道时间更长。

**六、小结与心得体会**

通过本题，我了解了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，还掌握常用磁盘调度算法及其相关特性。

**实验八**

**一、实验题目：虚拟内存系统的页面置换算法模拟**

**二、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。

**三、总体设计（含背景知识或基本原理与算法、或模块介绍、设计步骤等）**

**3.1基本原理与算法**

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进行预测，减少缺页次数。

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。

**3.2模块介绍**

void init(); //初始化

void option(); //选择使用哪种算法

void exchange(int str, int page, int num); //置换页面

int empty\_or\_in(int str, int page, int str\_num);//检查内存块为空或指令已在内存中

void OPT(int str, int page, int str\_num); //最佳置换算法

void FIFO(int str, int page, int str\_num); //先进先出置换算法

void LRU(int str, int page, int str\_num); //最近最久未用置换算法

1. **详细设计（含主要的数据结构、程序流程图、关键代码等）**

**4.1数据结构**

#define len 320 //作业指令数

#define nlen 4 //内存块数

int strs[len]; //存放指令地址

int pages[len]; //存放指令页

int count; //缺页次数

int time[nlen]; //记录进入内存时间

//创建一个单向循环列表来表示内存块

typedef struct Block {

int block\_num; //块号

int page\_num; //装进内存的作业页号

struct Block \*next;

} Block, \*BlockList;

BlockList blocks[nlen];

**4.2程序流程图**



**4.3关键代码**

//empty\_or\_in(int str, int page, int str\_num)

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

if(b->page\_num == -1) { //块为空

b->page\_num = page;

time[b->block\_num-1] = str\_num;

count++;

printf("指令未装入内存！页面置换完成\n");

printf("用户指令第%d页第%d条的物理地址为：第%d块第%d条\n\n", page, str % 10+1, b->block\_num, str % 10+1);

return 1;

}

if(b->page\_num == page) { //指令在内存中

printf("指令已在内存中！\n");

printf("用户指令第%d页第%d条的物理地址为：第%d块第%d条\n\n", page, str % 10+1, b->block\_num, str % 10+1);

return 1;

}

b = b->next;

}

void exchange(int str, int page, int num) {

//num表示被置换出的内存块号

BlockList b = blocks[0];

int i;

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

b = b->next;

b->page\_num = page;

count++;

printf("指令未装入内存且内存块已满！ 页面置换完成！\n");

printf("用户指令第%d页第%d条的物理地址为：第%d块第%d条\n\n", page, str % 10+1, b->block\_num, str % 10+1);

}

//OPT

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

for(j = str\_num; j < len; j++) {

if(b->page\_num == pages[j]) {

next[i] = j;

break;

}

}

b = b->next;

if(j == len) //当前页不再使用

next[i] = j;

if(next[max] < next[i]) //选出需要被置换出的内存页

max = i;

}

exchange(str, page, max);

// FIFO

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

if(time[earliest] > time[i])

earliest = i;

}

exchange(str, page, earliest+1);

time[b->block\_num-1] = str\_num;

// LRU

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

for(j = str\_num-2; j >=0; j --) {

if(b->page\_num == pages[j]) {

prior[i] = j;

break;

}

}

b = b->next;

if(prior[min] > prior[i])

min = i;

}

exchange(str, page, min);

1. **实验结果与分析**

**5.1实验结果**











**5.2分析**

OPT算法每次选择淘汰的页面是以后永不使用，或在最长时间内不被访问的页面。算法性能最好，可以保证最低的缺页率。但实际上操作系统无法提前预判页面访问序列，所以OPT算法是无法实现的。

FIFO算法每次选择淘汰的页面是最早进入内存的页面，但该算法与进程实际运行时的规律不适应，因为先进入的页面也有可能最经常被访问。因此算法性能差。

LRU算法每次选择淘汰的页面是最近最久未使用的页面，算法性能好。但该算法的实现需要专门的硬件支持，所以实现困难，开销大。

1. **小结与心得体会**

通过对页面、页表、地址转换和页面置换过程的模拟，加深了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。实现了OPT、FIFO、FRU算法，对页面置换算法有了更深的理解。

**实验九**

**一、实验题目：基于信号量机制的并发程序设计**

**二、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。

**三、总体设计**

**3.1基本原理与算法**

有读者和写者两组并发进程，共享一个文件，当两个或两个以上的读进程同时访问共享数据时不会产生副作用，但若某个写进程和其他进程（读进程或写进程）同时访问共享数据时则可能导致数据不一致的错误。因此要求：①允许多个读者可以同时对文件执行读操作；②只允许一个写者往文件中写信息；③任一写者在完成写操作之前不允许其他读者或写者工作；④写者执行写操作前，应让已有的读者和写者全部退出。

**3.2模块介绍**

DWORD WINAPI reader(LPVOID); //读者线程

DWORD WINAPI writer(LPVOID); //写者线程

**3.3设计步骤**

//伪代码

writer (){

while(flag){

P(rw); //写之前“加锁”

写文件…

V(rw); //写完了“解锁” }

}

reader (){

while(flag){

P(mutex); //各读进程互斥访问count

count++; //访问文件的读进程数+1

if(count==1) //由第一个读进程负责

P(rw); //读之前“加锁”

V(mutex);

读文件…

P(mutex); //各读进程互斥访问count

count--; //访问文件的读进程数-1

if(count==0) //由最后一个读进程负责

V(rw); //读完了“解锁”

V(mutex);

}}

**四、详细设计（含主要的数据结构、程序流程图、关键代码等）**

**4.1数据结构**

const int READER\_COUNT = 3; //读者个数

const int WRITER\_COUNT = 1; //写者个数

const int THREADS\_COUNT = READER\_COUNT + WRITER\_COUNT; //总的线程数

int count = 0; //记录当前有几个读线程在访问文件

int flag = 1, flag0[THREADS\_COUNT]; //flag标识子线程运行中；flag0标识子线程完成一次操作

HANDLE Mutex; //用于保证对count变量的互斥访问

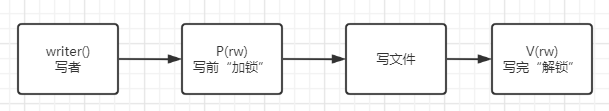
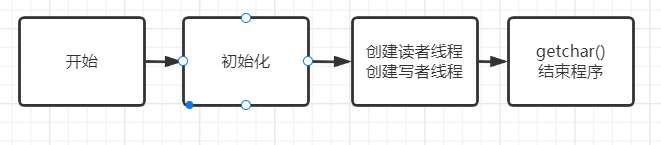
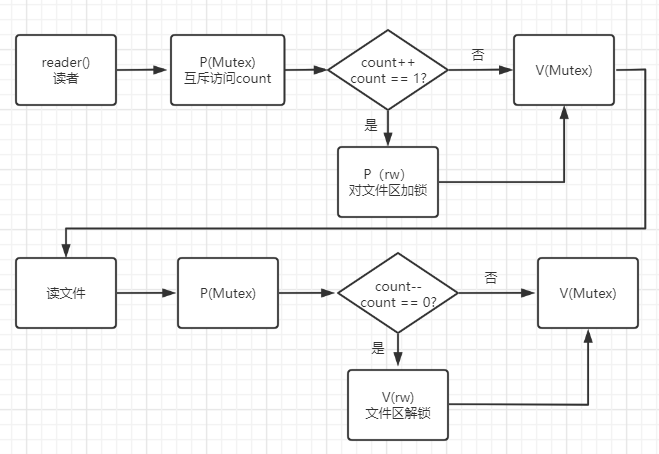
HANDLE rw; //用于实现对共享文件的互斥访问

typedef struct {

int id;

} threadP;

**4.2程序流程图**



**4.3关键代码**

**//main()**

//创建读者线程

for(i = 0; i < READER\_COUNT; i++) {

tp[i].id = i+1;

hThreads[i] = CreateThread(NULL, 0, reader, &tp[i], 0, &readerID[i]);

if(hThreads[i] == NULL)

return -1;

}

//创建写者线程

for(i = 0; i < WRITER\_COUNT; i++) {

tp[i].id = i+1;

hThreads[READER\_COUNT + i] = CreateThread(NULL, 0, writer, &tp[i], 0, &writerID[i]);

if(hThreads[READER\_COUNT + i] == NULL) return -1;

}

//reader()

flag0[id-1] = 1;

WaitForSingleObject(Mutex, INFINITE); //P(Mutex)

count++;

if(count == 1) //第一个读线程给共享文件加锁

WaitForSingleObject(rw, INFINITE); //P(rw)

if(!flag) { //当主程序终止时，还没开始操作子线程终止

flag0[id-1] = 0;

return 0;

}

printf("读线程%d读文件中...\n", id);

ReleaseMutex(Mutex); //V(Mutex)

//模拟读文件操作

Sleep(rand() % 3000);

WaitForSingleObject(Mutex, INFINITE); //P(Mutex)

printf("读线程%d读文件结束\n", id);

count--;

if(count == 0) //当没有读线程时给共享文件开锁

ReleaseSemaphore(rw, 1, NULL); //V(rw)

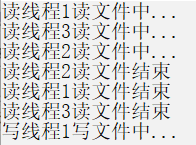
ReleaseMutex(Mutex); //V(Mutex)

flag0[id-1] = 0;

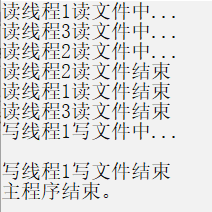
**五、实验结果与分析**

**5.1实验结果**

程序启动：



按下键盘，等待所有的子程序运行完成后，主程序结束：



**5.2分析**

可以看出，只有当读程序全部结束后，写程序才能运行。只要有读程序还在读，写程序就要一直阻塞等待，可能“饿死”。因此，在这种算法中，读者是优先的。

**六、小结与心得体会**

通过本题，我回顾了操作系统进程、线程有关概念，针对经典的同步、互斥、死锁和饥饿问题进行并发程序设计。了解了互斥对象并利用互斥与同步操作编写了读者-写者问题的并发程序，加深对P（semWait）、V（semSignal）原语以及利用PV原语进行线程间同步与互斥操作的理解。