

# 操作系统 -- 实验报告

## 实验一：Windows进程管理

### 一、实验题目

(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，在“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 1-1 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 1-2 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 1-2 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。列出运行结果。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 1-3 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 1-3，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 1-3，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。

## 二、实验目的

- (1) 学会使用 VC 编写基本的 Win32 Console 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、总体设计

- 主函数 `main()` 中判断当前处于父进程还是子进程，如果处于子进程则调用 `Parent()` 函数创建子进程并传入参数，实现**进程间通信**。
- `Parent()` 函数中主要包括创建子进程并调用 `ReleaseMutex()` 函数释放互斥体的所有权，以及调用 `CloseHandle()` 消除句柄。
- `Child()` 子进程中则主要包括打开自杀互斥体并进入**阻塞状态**，等待父进程通过互斥体发来的信号。收到信号后准备终止并清除句柄。

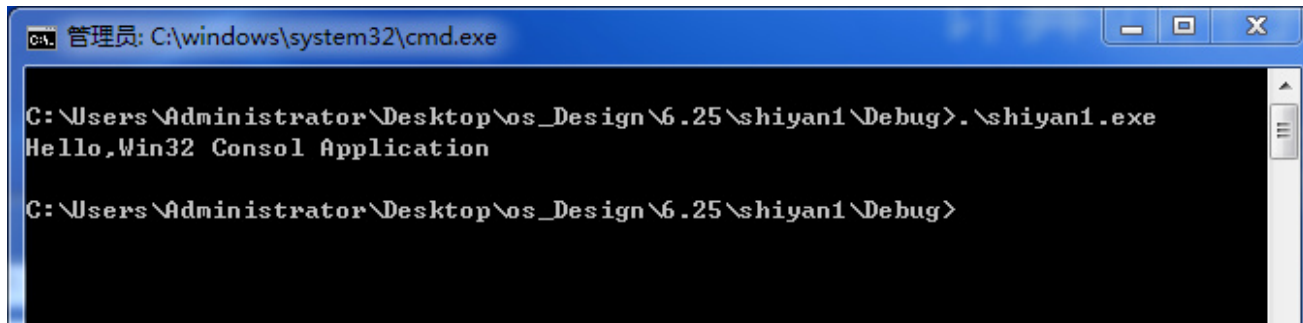
## 四、详细设计

### 1. 编写基本的 Win32 Console Application

实验代码：

```
#include "stdafx.h"
#include<iostream>
void main()
{
    std::cout<<"Hello,Win32 Consol Application"<<std::endl;
}
```

运行截图



### 2. 创建进程

实验代码：

```
// shiyang1_2.cpp : Defines the entry point for the console application.
```

```
//

#include "stdafx.h"
#include<windows.h>
#include<iostream>
#include<stdio.h>

// 创建传递过来的进程的克隆过程并赋予其ID值
void StartClone(int nCloneID)
{
    // 提取用于当前可执行文件的文件名
    // szFilename: [D:\testProject\Debug\testProject.exe]
    // MAX_PATH: [MAX_PATH是C语言运行时库中通过#define指令定义的一个宏常量, 它定义了编译器所支持的最长全路径名的长度]
    // Windows的MAX_PATH: [MAX_PATH的解释: 文件名最长256 (ANSI), 加上盘符 (X:\) 3字节, 259字节, 再加上结束符1字节, 共260]
    TCHAR szFilename[MAX_PATH];
    GetModuleFileName(NULL, szFilename, MAX_PATH);

    // 格式化用于子进程的命令行并通知其EXE文件名和克隆ID
    // szCmdLine: ["D:\testProject\Debug\testProject.exe"5]
    TCHAR szCmdLine[MAX_PATH];
    sprintf(szCmdLine, "\"%s\" \"%d\"", szFilename, nCloneID);

    // 用于子进程的STARTUPINFO结构
    STARTUPINFO si;
    ZeroMemory(&si, sizeof(si));
    si.cb=sizeof(si); // 必须是本结构的大小

    // 返回的用于子进程的进程信息
    PROCESS_INFORMATION pi;

    // 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
    BOOL bCreateOK=::CreateProcess(
        szFilename,    // 产生这个EXE文件的应用程序的名称
        szCmdLine,    // 告诉其行为像一个子进程的标志
        NULL,         // 缺省的进程安全性
        NULL,         // 缺省的线程安全性
        FALSE,        // 不继承句柄
        CREATE_NEW_CONSOLE, // 使用新的控制台
        NULL,         // 新的环境
        NULL,         // 当前目录
        &si,          // 启动信息
        &pi);         // 返回的进程信息

    // 对子进程释放引用
    if(bCreateOK)
    {
        CloseHandle(pi.hProcess);
        CloseHandle(pi.hThread);
    }
}
```

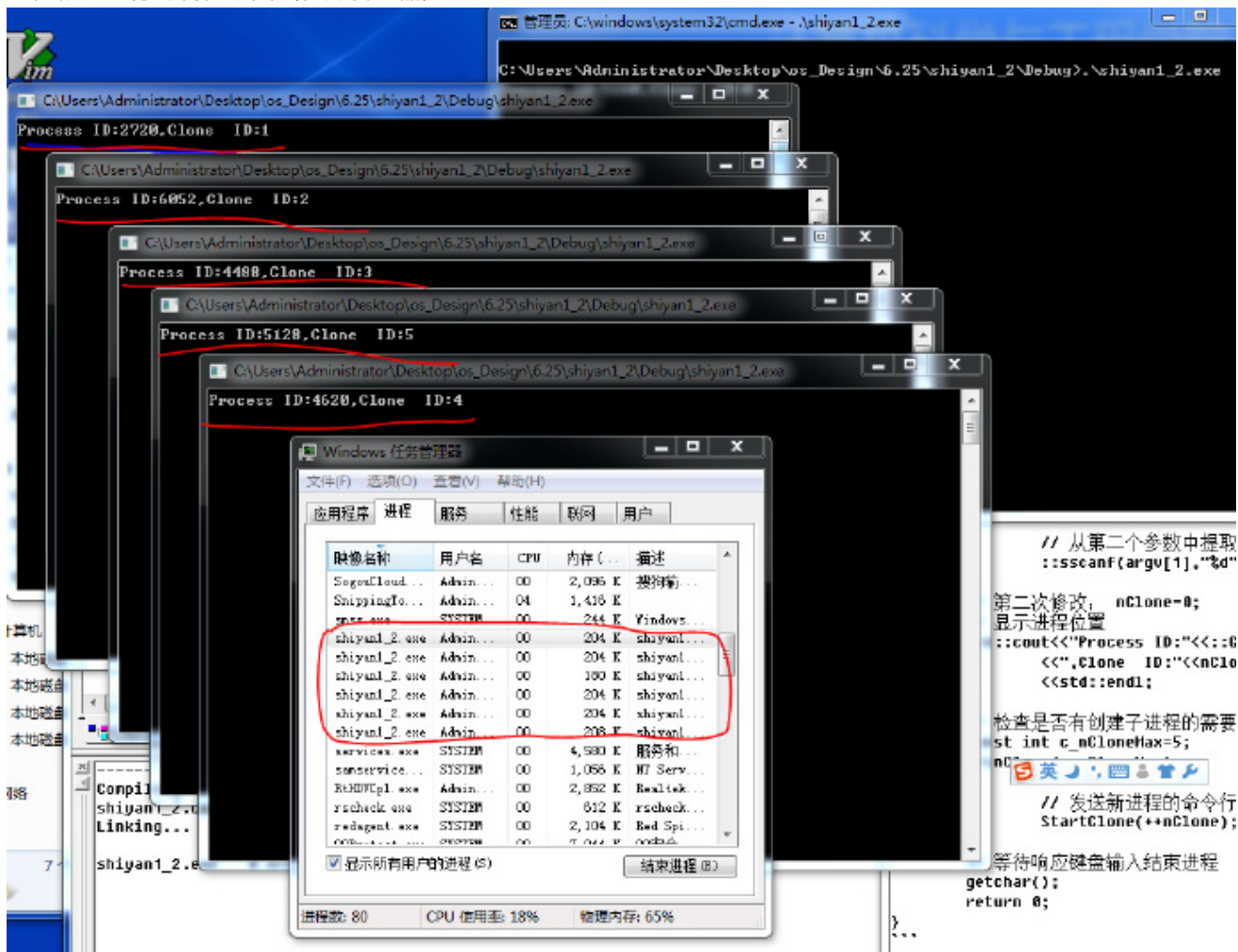
```

int main(int argc, char* argv[])
{
    // 确定派生出几个进程，及派生进程在进程列表中的位置
    int nClone=0;
    // 修改语句: int nClone;
    // 第一次修改: nClone=0;
    if(argc>1)
    {
        // 从第二个参数中提取克隆 ID
        // 之所以按照注释修改代码就会死循环，原因在于父进程在创建子进程时会把nClone当作命令行参数传入
        // 子进程，nClone=0的位置就显得尤为重要!! (参考line 20注释，命令行后附带参数[cClone])
        ::sscanf(argv[1], "%d", &nClone);
        printf("子进程读取到命令行参数:%d\n", nClone);
    }
    // 第二次修改: nClone=0;
    //nClone=0;
    // 显示进程位置
    std::cout<<"Process ID:"<<::GetCurrentProcessId()
        <<" , Clone ID:"<<nClone
        <<std::endl;

    // 检查是否有创建子进程的需要
    const int c_nCloneMax=5;
    if(nClone<c_nCloneMax)
    {
        // 发送新进程的命令行和克隆号
        StartClone(++nClone);
    }
    // 等待响应键盘输入结束进程
    getchar();
    return 0;
}

```

进程相关的行为属性截图（任务管理器）



### 3. 父进程的简单通信及终止进程

实验代码：

```
/*
 * shiyani_3.cpp : Defines the entry point for the console application.
 *
 * procterm 项目
 */

#include "stdafx.h"
#include<windows.h>
#include<iostream>
#include<stdio.h>
static LPCTSTR g_szMutexName="w2kdg.ProcTerm.mutex.Suicide";

// 创建当前进程的克隆进程的简单方法
void StartClone()
{
    // 提取当前可执行文件的文件名
    TCHAR szFilename[MAX_PATH];
    GetModuleFileName(NULL, szFilename, MAX_PATH);
```

```

// 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
TCHAR szCmdLine[MAX_PATH];
//// 实验1-3 步骤3：将下句中的字符创child改为别的字符创，重新编译执行，执行前请先保存已经完成的工作
sprintf(szCmdLine, "\\%s\\"child", szFilename);

// 子进程的启动信息结构
STARTUPINFO si;
ZeroMemory(&si, sizeof(si));
si.cb=sizeof(si); // 应当是此结构的大小

// 返回的用于子进程的进程信息
PROCESS_INFORMATION pi;

// 用同样的可执行文件名和命令行创建进程，并指明他是一个子进程
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, // 用于线程的缺省的安全性
    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); // 启动的进程信息

// 释放指向子进程的引用
if(bCreateOK)
{
    CloseHandle(pi.hProcess);
    CloseHandle(pi.hThread);
}
}

void Parent()
{
    // 创建"自杀"互斥程序体
    HANDLE hMutexSuicide=CreateMutex(
        NULL, // 缺省的安全性
        TRUE, // 最初拥有的
        g_szMutexName); // 互斥体名称
    if(hMutexSuicide!=NULL)
    {
        // 创建子进程
        std::cout<<"Creating the child process."<<std::endl;
        StartClone();
        // 指令子进程"杀"掉自身
        std::cout<<"Telling the child process to quit."<<std::endl;
        // 等待父进程的键盘响应
        getchar();
        // 释放互斥体的所有权，这个信号会送给子进程的WaitForSingleObject过程

        ReleaseMutex(hMutexSuicide);
    }
}

```

```

        // 消除句柄
        CloseHandle(hMutexSuicide);
    }
}

void Child()
{
    // 打开“自杀”互斥体
    HANDLE hMutexSuicide=OpenMutex(
        SYNCHRONIZE, // 打开用于同步
        FALSE, // 不需要向下传递
        g_szMutexName); // 名称
    if(hMutexSuicide!=NULL)
    {
        // 报告我们正在等待指令
        std::cout<<"Child waiting for suicide instructions."<<std::endl;

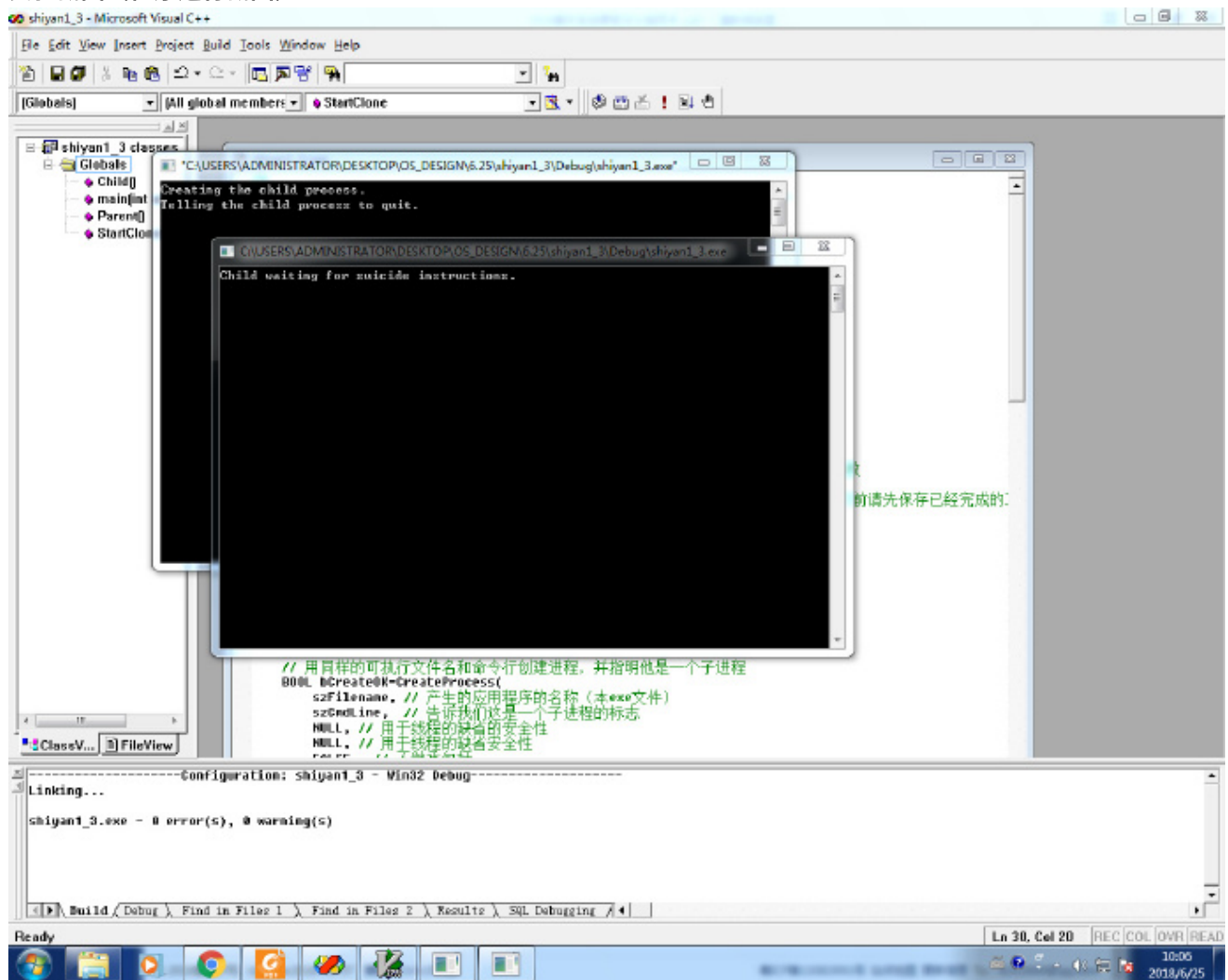
        // 子进程进入阻塞状态，等待父进程通过互斥体发来的信号
        WaitForSingleObject(hMutexSuicide, INFINITE);
        // 实验1-3步骤4：将上句改为 WaitForSingleObject(hMutexSuicide, 0), 重新百衲衣执行

        // 准备好终止，清除句柄
        std::cout<<"Child quitting."<<std::endl;
        CloseHandle(hMutexSuicide);
    }
}

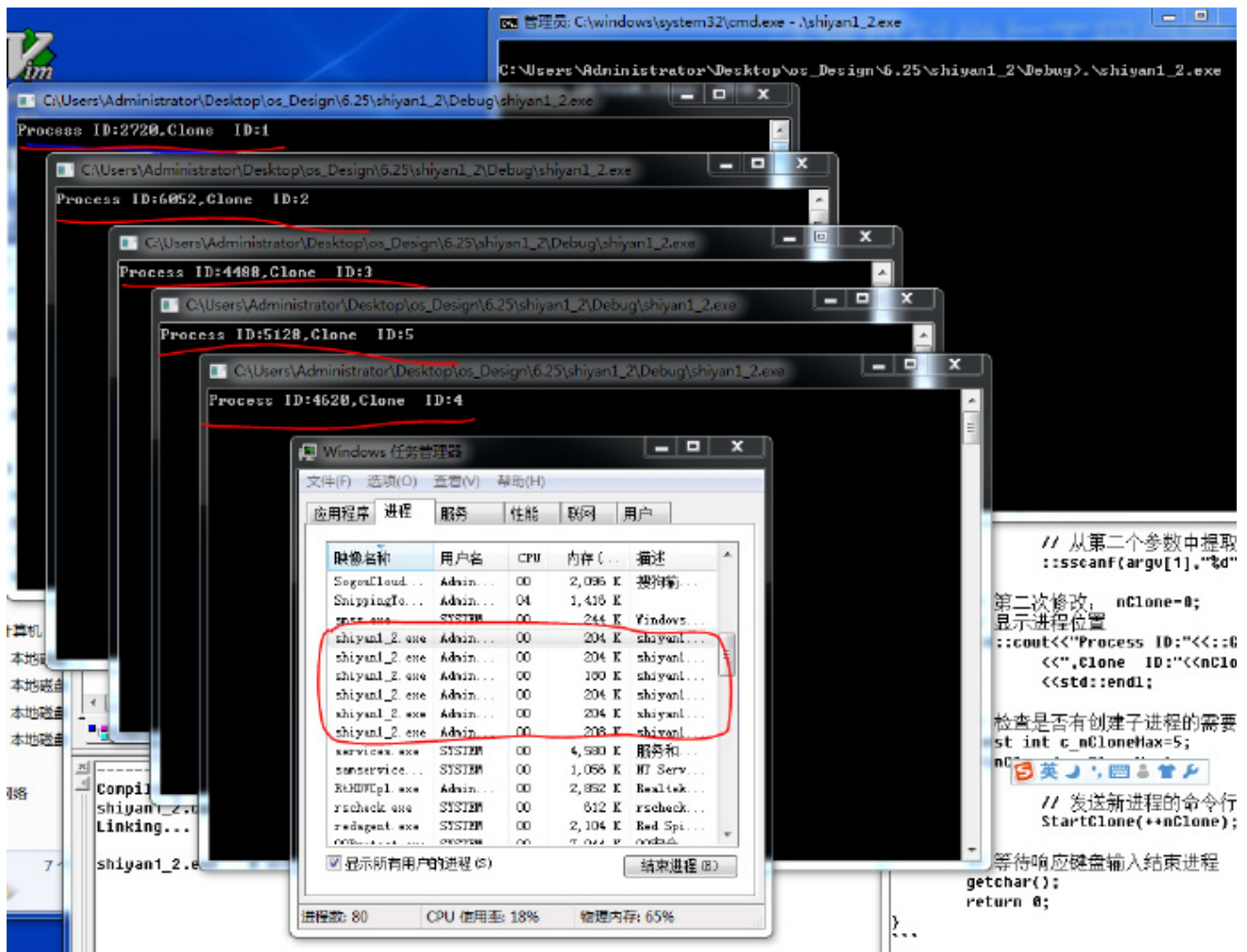
int main(int argc, char* argv[])
{
    // 决定其行为是父进程还是子进程
    if(argc>1 && ::strcmp(argv[1], "child")==0)
    {
        Child();
    }
    else
    {
        Parent();
    }
    return 0;
}

```

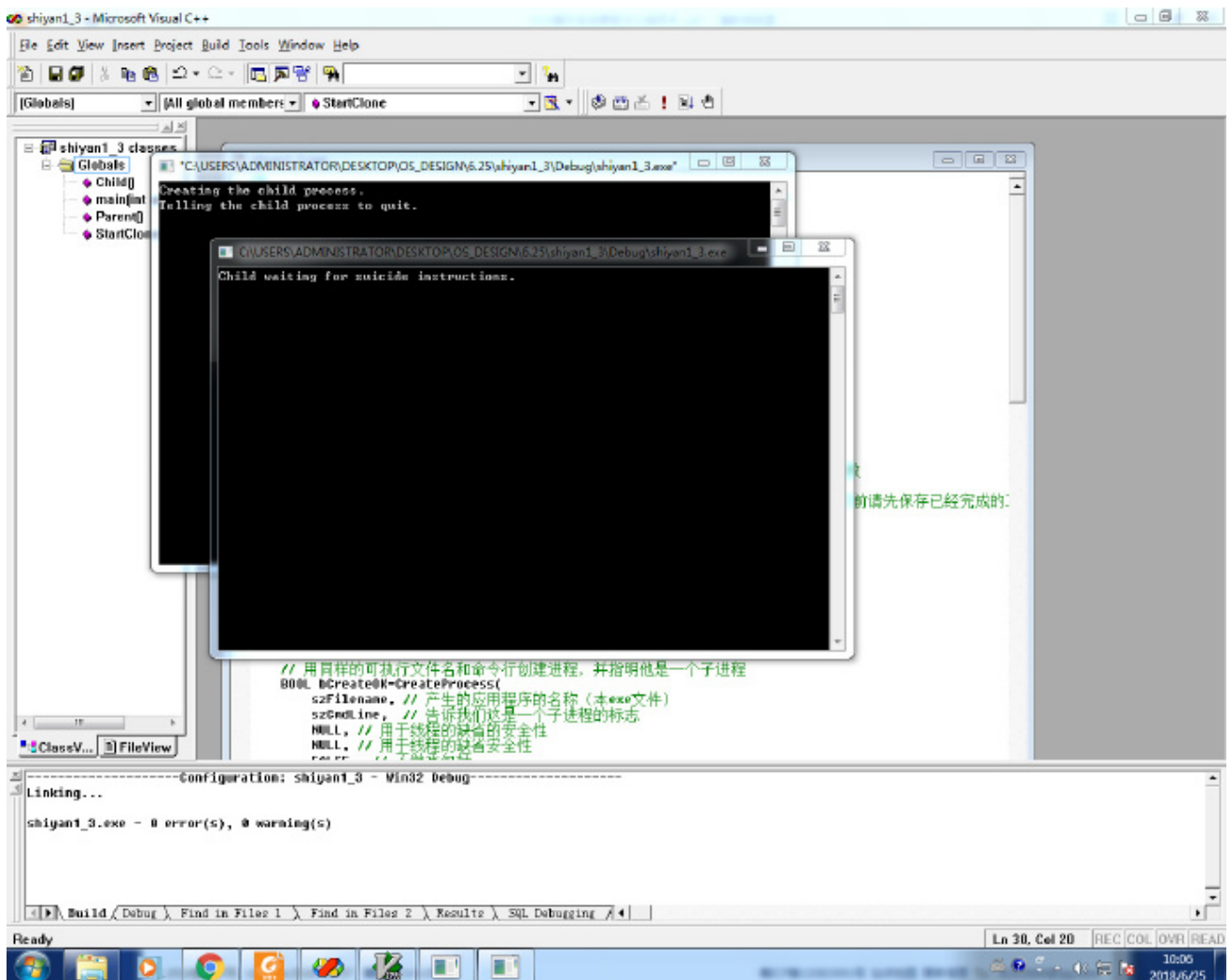
**实验截图（程序运行截图）：**







(2)父子进程间的简单通信及进程终止运行结果：



## 实验分析:

- nClone 的作用为确定创建子进程的数量;
- 之所以按照注释修改代码就会死循环, 原因在于父进程在创建子进程时会把nClone当作命令行参数传入子进程, nClone=0的位置就显得尤为重要!
- 将代码 `WaitForSingleObject(hMutexSuicide, INFINITE);` 改为 `WaitForSingleObject(hMutexSuicide, 0);` 后, 子进程不再等待父进程通过互斥体发来的信号 (等待时间为0), 因此直接往下执行代码。清除句柄, 放弃进程。

## 六、小结与心得体会

- MAX\_PATH:[MAX\_PATH是C语言运行时库中通过#define指令定义的一个宏常量, 它定义了编译器所支持的最长全路径名的长度]
- Windows的MAX\_PATH:[MAX\_PATH的解释: 文件名最长256 (ANSI), 加上盘符 (X:\) 3字节, 259字节, 再加上结束符1字节, 共260]
- 为了获得互斥体, 首先, 想要访问调用的线程可使用 `OpenMutex()` API 来获得指向对象的句柄;

## 实验二：Linux进程管理

# 一、实验题目

## 子进程执行新任务

任务要求：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/lis` (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。程序执行过程如图 2-1 所示。

## 进程的创建

任务要求：编写一段程序，使用系统调用 `fork()` 创建两个子进程。当此程序运行时，在系统中有一个父进程和两个子进程活动。让每一个进程在屏幕上显示一个字符：父进程显示字符“a”；两子进程分别显示字符“b”和字符“c”。

实现一个简单的shell（命令行解释器）(此任务有一些难度，可选做)。

要设计的 shell 类似于 `sh`, `bash`, `csh` 等，必须支持以下内部命令：

**cd** <目录> 更改当前的工作目录到另一个<目录>。如果<目录>未指定，输出当前工作目录。如果<目录>不存在，应当有适当的错误信息提示。这个命令应该也能改变 **PWD** 的环境变量。

**environ** 列出所有环境变量字符串的设置（类似于 Unix 系统下的 **env** 命令）。

**echo** <\*\*内容\*\*> 显示 echo 后的内容且换行

**help** 简短概要的输出你的 shell 的使用方法和基本功能。

**jobs** 输出 shell 当前的一系列子进程，必须提供子进程的命名和 PID 号。

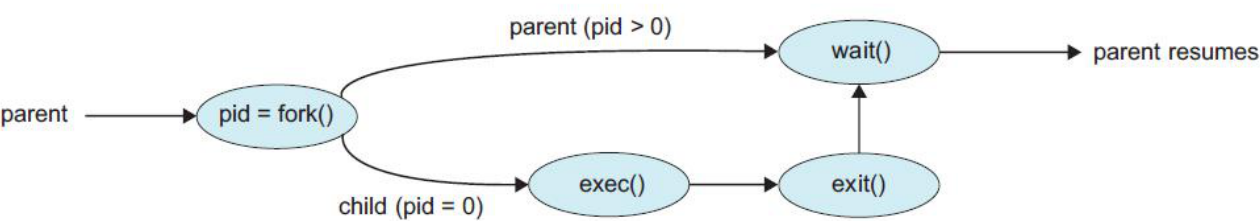
**quit,exit,bye** 退出 shell。

# 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、总体设计

进程创建部分程序流程图如下：



Shell程序设计：

shell 的主体就是反复下面的循环过程

```
while(1){
    //接收用户输入的命令行；
    //解析命令行；
```

```
if(用户命令为内部命令)

//直接处理;

else if(用户命令为外部命令)

//创建子进程执行命令;      //参考清单 2-2

else

//提示错误的命令;

}
```

## 四、详细设计

### (1) 进程的创建

**任务要求：**编写一段程序，使用系统调用 `fork()` 创建两个子进程。当此程序运行时，在系统中有一个父进程和两个子进程活动。让每一个进程在屏幕上显示一个字符：父进程显示字符“a”；两子进程分别显示字符“b”和字符“c”。

**步骤1：**使用 `vi` 或 `gedit` 新建一个 `fork_demo.c` 程序，然后拷贝清单\* 2-1 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。

### (2) 子进程执行新任务

**任务要求：**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/l`s (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。程序执行过程如图2-1 所示。

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

**步骤 2：**在命令行输入 `./exec_demo` 运行该程序。

**步骤 3：**观察该程序在屏幕上的显示结果，并分析。

### (3) 实现一个简单的shell（命令行解释器）（此任务有一些难度，可选做）。

要设计的 shell 类似于 `sh`, `bash`, `csh` 等，必须支持以下内部命令：

**cd**<目录>更改当前的工作目录到另一个<目录>。如果<目录>未指定，输出当前工作目录。如果<目录>不存在，应当有适当的错误信息提示。这个命令应该也能改变 **PWD** 的环境变量。

**environ** 列出所有环境变量字符串的设置（类似于\*\* Unix 系统下的 `env` \*\*命令）。

**echo <内容>**显示 **echo**后的内容且换行

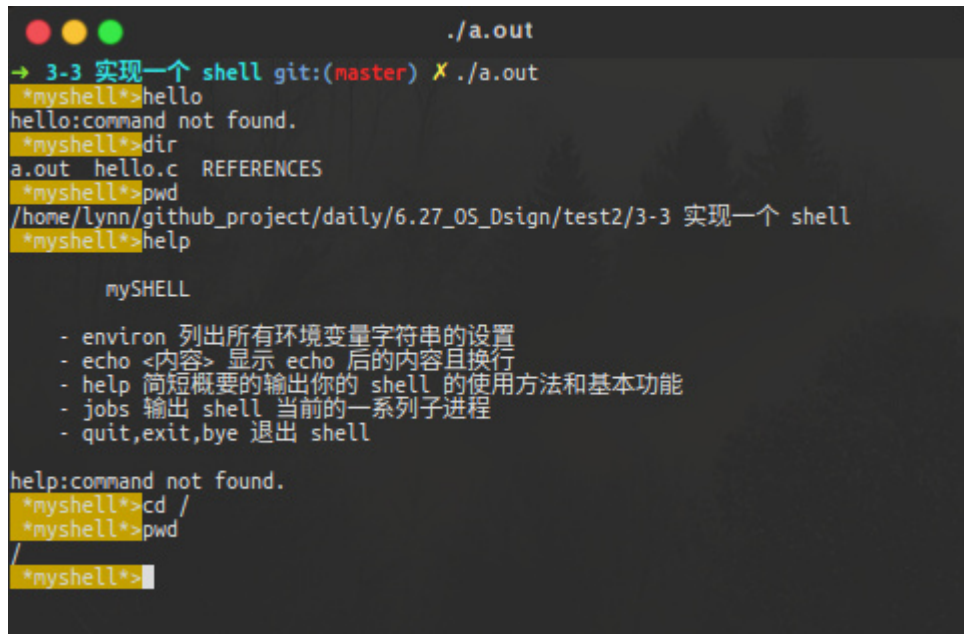
**help**简短概要的输出你的 **shell** \*\*的使用方法和基本功能。

**jobs**输出 **shell** 当前的一系列子进程，必须提供子进程的命名和 **PID**\*\*号。

**quit,exit,bye**退出shell。

## 五、实验结果与分析

实验结果图：



```
./a.out
→ 3-3 实现一个 shell git:(master) X ./a.out
*myshell*>hello
hello:command not found.
*myshell*>dir
a.out hello.c REFERENCES
*myshell*>pwd
/home/lynn/github_project/daily/6.27_OS_Dsign/test2/3-3 实现一个 shell
*myshell*>help

mySHELL

- environ 列出所有环境变量字符串的设置
- echo <内容> 显示 echo 后的内容且换行
- help 简短概要的输出你的 shell 的使用方法和基本功能
- jobs 输出 shell 当前的一系列子进程
- quit,exit,bye 退出 shell

help:command not found.
*myshell*>cd /
*myshell*>pwd
/
*myshell*>
```

实验结果分析：

整体框架为：一个死循环,一直在等待这用户输入命令.主要的工作都在eval函数里面。

### eval函数实现

Linux中命令分为内置命令和外置命令,内置命令不需要开进程就可以直接执行,比如cd,pwd.外置命令需要重开个进程才能执行,比如vim,/bin/ls,/bin/echo. eval函数中有两个外调函数就是parseline,解析命令行cmdstring,储存到argv数组中.bulidin\_command函数判断命令是否是内置命令,如果是,则执行.不是则返回false.

### parseline函数实现

就是一个字符串操作,根据空格分割,写的比较屎.读者可以自己实现,不需要模仿我的.记得最后的argv数组要以NULL结尾。

## 六、小结与心得体会

- 关于fork()函数:

fork () 函数调用时,在调用的地方立马创建一个子进程,子进程同时立马开始执行代码(全部代码)。对于以下代码：

```

12 int main()
13 {
14     int x;
15     printf("1111");
16     x=fork();
17     printf("222");
18     printf("x=%d\n",x);
19     printf("333\n");
20     int i=0;
21     int b=9;
22     while(i<9)
23     {
24         i++;
25         b=i;
26     }
27     printf("finally\n");
28 }
...

```

当22行为 `while(i<999999)` 时，运行结果为：

```

1111222x=22093
333
1111222x=0
333
finally
finally

```

当22行为 `while(i<9)` 时，运行结果为：

```

1111222x=22093
333
finally
1111222x=0
333
finally

```

由此可见，执行父进程执行 `fork()` 函数之后，当即产生一个子进程，并且同时子进程开始运行(相同的代码)。

## 实验三：互斥与同步

### 一、实验题目

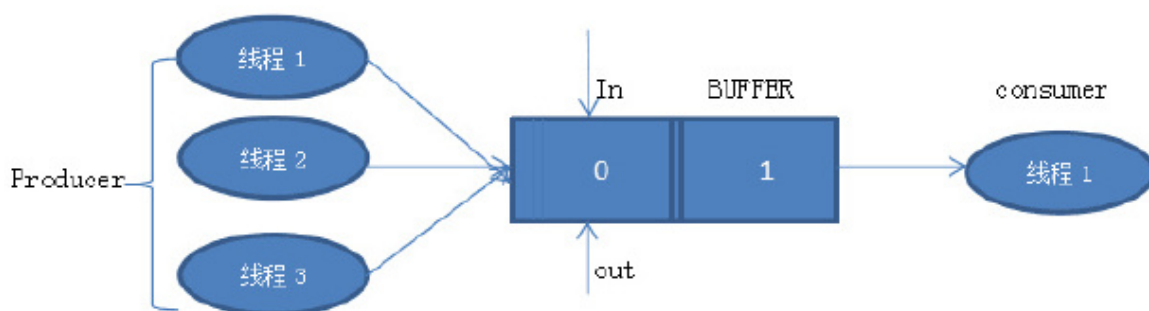
- (1) 生产者消费者问题
- (2) 读者写者问题（选做）

## 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、总体设计

程序流程图：



## 四、详细设计

- 程序主要包括 main(), Produce(), Append(), Consume() 和 Take() 函数
- main() 函数控制整个程序的流程，创建生产者线程、消费者线程等
- Produce() 函数主要功能为：

生产一个产品。简单模拟了一下，仅输出新产品的 ID 号

- Append() 函数主要功能为：

把新生产的产品放入缓冲区

- Take() 函数主要功能为：

从缓冲区中取出一个产品

- Consume() 函数的主要功能为：

模拟消耗一个产品

- Producer() 函数的主要功能为：

模拟生产者功能，生产一个产品

- Consumer() 函数的主要功能为：

模拟消费者，控制信号量，调用 Consume() 函数

实验代码：

```
// test3_1.cpp : Defines the entry point for the console application.
//
```



```

#include "stdafx.h"
#include <windows.h>
#include <iostream>
const unsigned short SIZE_OF_BUFFER = 2; //缓冲区长度
unsigned short ProductID = 0; //产品号
unsigned short ConsumeID = 0; //将被消耗的产品号
unsigned short in = 0; //产品进缓冲区时的缓冲区下标
unsigned short out = 0; //产品出缓冲区时的缓冲区下标
int buffer[SIZE_OF_BUFFER]; //缓冲区是个循环队列
bool p_continue = true; //控制程序结束

HANDLE Mutex; //用于线程间的互斥
HANDLE FullSemaphore; //当缓冲区满时迫使生产者等待
HANDLE EmptySemaphore; //当缓冲区空时迫使消费者等待
DWORD WINAPI Producer(LPVOID); //生产者线程
DWORD WINAPI Consumer(LPVOID); //消费者线程

int main()
{
    //创建各个互斥信号
    //注意，互斥信号量和同步信号量的定义方法不同，互斥信号量调用的是 CreateMutex 函数，
    //同步信号量调用的是 CreateSemaphore 函数，函数的返回值都是句柄。
    Mutex = CreateMutex(NULL, FALSE, NULL);
    EmptySemaphore = CreateSemaphore(NULL, SIZE_OF_BUFFER, SIZE_OF_BUFFER, NULL);
    //将上句做如下修改，看看结果会怎样
    //EmptySemaphore = CreateSemaphore(NULL, 0, SIZE_OF_BUFFER-1, NULL);
    FullSemaphore = CreateSemaphore(NULL, 0, SIZE_OF_BUFFER, NULL);
    //调整下面的数值，可以发现，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
    const unsigned short PRODUCERS_COUNT = 3; //生产者的个数
    const unsigned short CONSUMERS_COUNT = 1; //消费者的个数
    //总的线程数
    const unsigned short THREADS_COUNT = PRODUCERS_COUNT+CONSUMERS_COUNT;
    HANDLE hThreads[THREADS_COUNT]; //各线程的 handle
    DWORD producerID[PRODUCERS_COUNT]; //生产者线程的标识符
    DWORD consumerID[CONSUMERS_COUNT]; //消费者线程的标识符
    //创建生产者线程
    for (int i=0; i<PRODUCERS_COUNT; ++i)
    {
        hThreads[i]=CreateThread(NULL, 0, Producer, NULL, 0, &producerID[i]);
        if (hThreads[i]==NULL) return -1;
    }
    //创建消费者线程
    for (i=0; i<CONSUMERS_COUNT; ++i)
    {
        hThreads[PRODUCERS_COUNT+i]=CreateThread(NULL, 0, Consumer, NULL, 0, &consumerID[i]);
        if (hThreads[i]==NULL) return -1;
    }
    while(p_continue)
    {
        if(getchar()) //按回车后终止程序运行

```



```

        {
            p_ccontinue = false;
        }
    }

    return 0;
}

//生产一个产品。简单模拟了一下，仅输出新产品的 ID 号
void Produce()
{
    std::cout << std::endl<< "Producing " << ++ProductID << " ... ";
    std::cout << "Succeed" << std::endl;
}

//把新生产的产品放入缓冲区
void Append()
{
    std::cerr << "Appending a product ... ";
    buffer[in] = ProductID;
    in = (in+1)%SIZE_OF_BUFFER;
    std::cerr << "Succeed" << std::endl;
    //输出缓冲区当前的状态
    for (int i=0; i<SIZE_OF_BUFFER; ++i)
    {
        std::cout << i <<": " << buffer[i];
        if (i==in) std::cout << " <-- 生产";
        if (i==out) std::cout << " <-- 消费";
        std::cout << std::endl;
    }
}

//从缓冲区中取出一个产品
void Take()
{
    std::cerr << "Taking a product ... ";
    ConsumeID = buffer[out];
    buffer[out] = 0;
    out = (out+1)%SIZE_OF_BUFFER;
    std::cerr << "Succeed" << std::endl;
    //输出缓冲区当前的状态
    for (int i=0; i<SIZE_OF_BUFFER; ++i)
    {
        std::cout << i <<": " << buffer[i];
        if (i==in) std::cout << " <-- 生产";
        if (i==out) std::cout << " <-- 消费";
        std::cout << std::endl;
    }
}

```

```

//消耗一个产品
void Consume()
{
    std::cout << "Consuming " << ConsumeID << " ... ";
    std::cout << "Succeed" << std::endl;
}

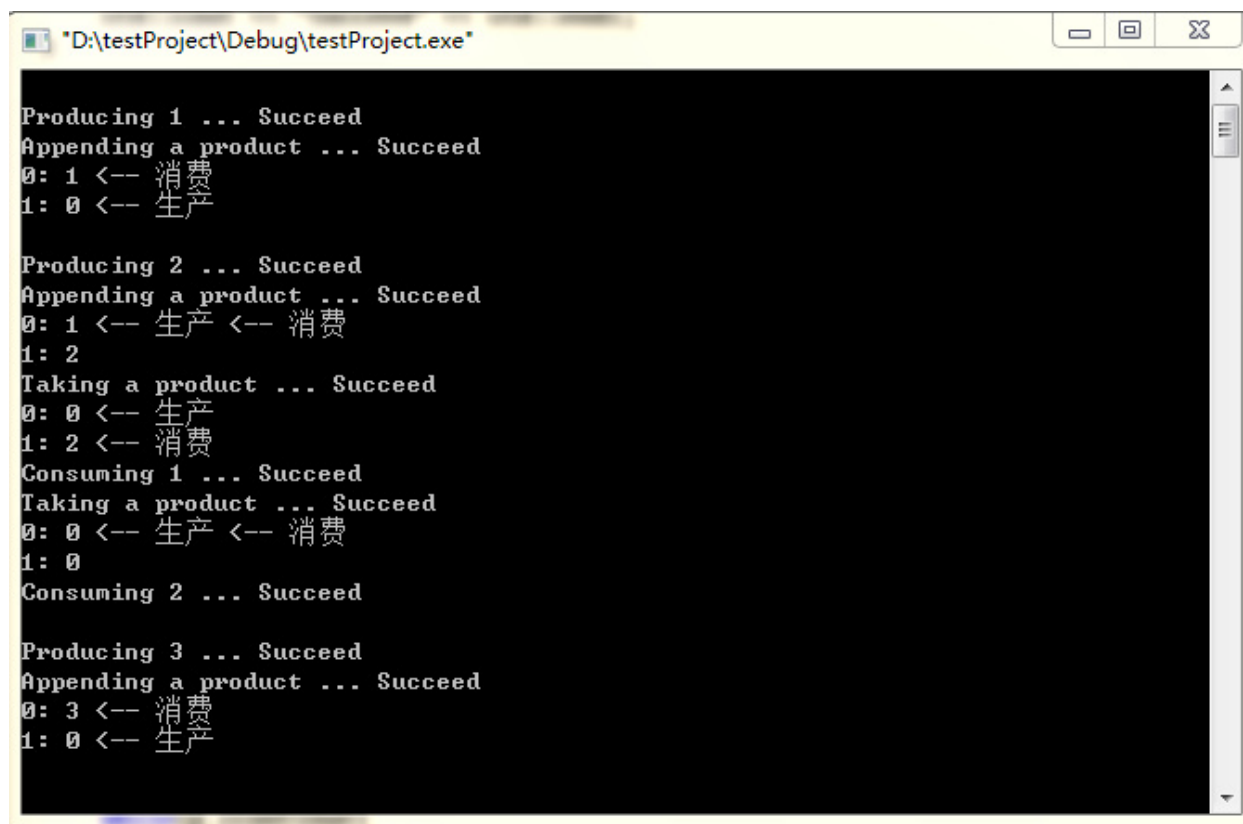
//生产者
DWORD WINAPI Producer(LPVOID lpPara)
{
    while(p_ccontinue)
    {
        WaitForSingleObject(EmptySemaphore, INFINITE); //p(empty);
        WaitForSingleObject(Mutex, INFINITE); //p(mutex);
        Produce();
        Append();
        Sleep(1500);
        ReleaseMutex(Mutex); //V(mutex);
        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;
}

```

## 五、实验结果与分析

- 生产者消费者问题运行结果截图：



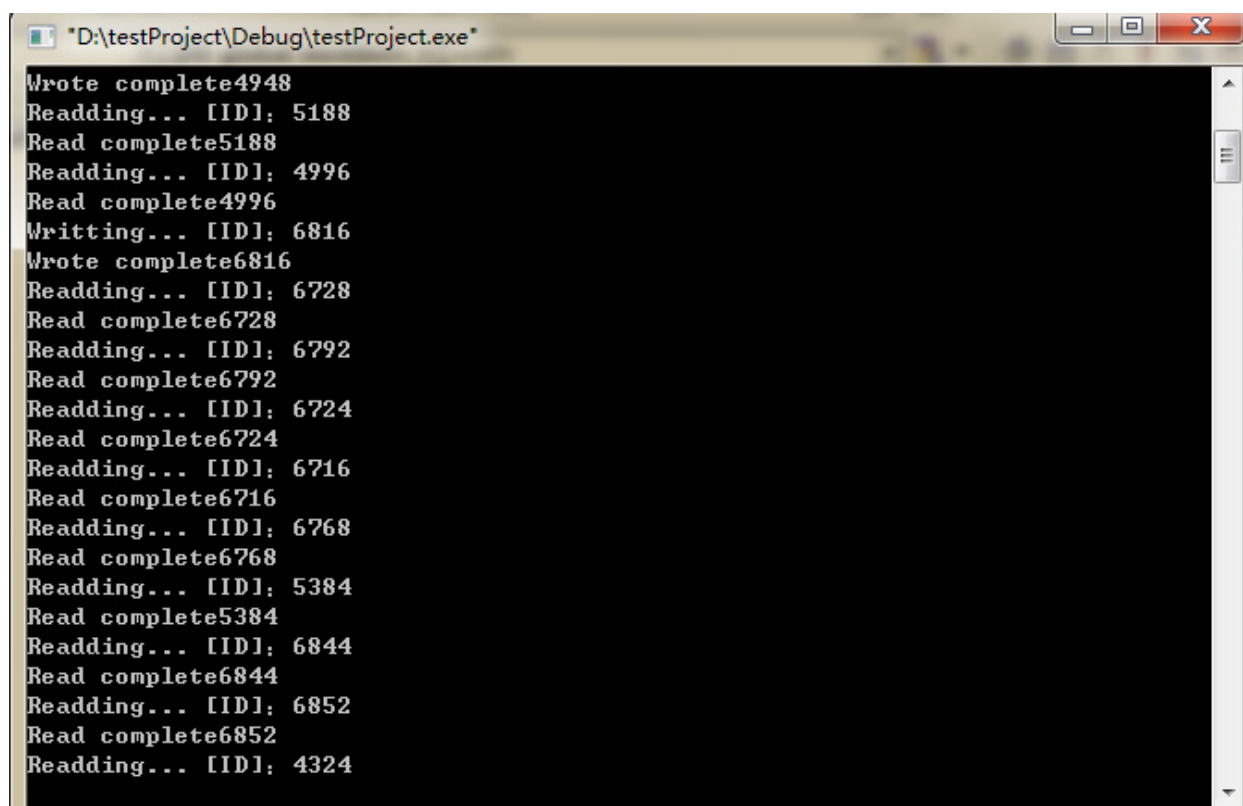
```
"D:\testProject\Debug\testProject.exe"

Producing 1 ... Succeed
Appending a product ... Succeed
0: 1 <-- 消费
1: 0 <-- 生产

Producing 2 ... Succeed
Appending a product ... Succeed
0: 1 <-- 生产 <-- 消费
1: 2
Taking a product ... Succeed
0: 0 <-- 生产
1: 2 <-- 消费
Consuming 1 ... Succeed
Taking a product ... Succeed
0: 0 <-- 生产 <-- 消费
1: 0
Consuming 2 ... Succeed

Producing 3 ... Succeed
Appending a product ... Succeed
0: 3 <-- 消费
1: 0 <-- 生产
```

- 读写者问题运行结果截图：



```
"D:\testProject\Debug\testProject.exe"

Wrote complete4948
Readding... [ID]: 5188
Read complete5188
Readding... [ID]: 4996
Read complete4996
Writting... [ID]: 6816
Wrote complete6816
Readding... [ID]: 6728
Read complete6728
Readding... [ID]: 6792
Read complete6792
Readding... [ID]: 6724
Read complete6724
Readding... [ID]: 6716
Read complete6716
Readding... [ID]: 6768
Read complete6768
Readding... [ID]: 5384
Read complete5384
Readding... [ID]: 6844
Read complete6844
Readding... [ID]: 6852
Read complete6852
Readding... [ID]: 4324
```

- 实验结果分析：

- 程序主要包括 main(), Produce(), Append(), Consume() 和 Take() 函数
  - main() 函数控制整个程序的流程，创建生产者线程、消费者线程等

- Produce() 函数主要功能为：  
生产一个产品。简单模拟了一下，仅输出新产品的 ID 号
- Append() 函数主要功能为：  
把新生产的产品放入缓冲区
- Take() 函数主要功能为：  
从缓冲区中取出一个产品
- Consume() 函数的主要功能为：  
模拟消耗一个产品
- Producer() 函数的主要功能为：  
模拟生产者功能，生产一个产品
- Consumer() 函数的主要功能为：  
模拟消费者，控制信号量，调用 Consume() 函数

## 六、小结与心得体会

### 1. CreateMutex():

有几个参数，各代表什么含义？

功能：

找出当前系统是否已经存在指定进程的实例。如果没有则创建一个互斥体。CreateMutex () 函数可用来创建一个有名或无名的互斥量对象。

函数原型：

```
HANDLE CreateMutex(
    LPSECURITY_ATTRIBUTES lpMutexAttributes, // 指向安全属性的指针
    BOOL bInitialOwner, // 初始化互斥对象的所有者
    LPCTSTR lpName // 指向互斥对象名的指针
);
```

返回值：

Long，如执行成功，就返回互斥体对象的句柄；0表示出错。会设置GetLastError。即使返回的是一个有效句柄，但倘若指定的名字已经存在，GetLastError 也会设为**ERROR\_ALREADY\_EXISTS**

参数及其说明：

**lpMutexAttributes SECURITY\_ATTRIBUTES**，指定一个SECURITY\_ATTRIBUTES结构，或传递零值（将参数声明为ByVal As Long，并传递零值），表示使用不允许继承的默认描述符。

**bInitialOwner BOOL**，如创建进程希望立即拥有互斥体，则设为**TRUE**。一个互斥体同时只能由一个线程拥有。**FALSE**，表示刚刚创建的这个Mutex不属于任何线程。也就是没有任何线程拥有他，一个Mutex在没有任何线程拥有他的时候，他是处于**激发态**的，所以处于**有信号状态**。

**lpName String**, 指定互斥体对象的名字。用vbNullString创建一个未命名的互斥体对象。如已经存在拥有这个名字的一个事件, 则打开现有的**已命名互斥体**。这个名字可能不与现有的事件、信号机、可等待计时器或文件映射相符, 该名称可以有一个"Global\" 或"Local\"前缀, 明确地建立在全局或会话命名空间的对象。剩余的名称可以包含任何字符, 除反斜杠字符 (\) 。

#### 注意:

一旦不再需要, 注意必须用CloseHandle函数将互斥体句柄关闭。从属于它的所有句柄都被关闭后, 就会删除对象。进程中止前, 一定要释放互斥体, 如不慎未采取这个措施, 就会将这个互斥体标记为废弃, 并自动释放所有权。共享这个互斥体的其他应用程序也许仍然能够用它, 但会接收到一个废弃状态信息, 指出上一个所有进程未能正常关闭。这种状况是否会造成影响取决于涉及到的具体应用程序\*\*

## 2. CreateSemaphore():

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。

#### 功能:

创建信号量

#### 原型:

```
HANDLE CreateSemaphore(  
    LPSECURITY_ATTRIBUTES lpSemaphoreAttributes,  
    LONG lInitialCount,  
    LONG lMaximumCount,  
    LPCTSTR lpName  
);
```

#### 函数说明:

第一个参数表示安全控制, 一般直接传入NULL。第二个参数表示初始资源数量。第三个参数表示最大并发数量。第四个参数表示信号量的名称, 传入NULL表示匿名信号量。

## 3. CreateThread():

#### 功能:

当使用CreateProcess调用时, 系统将创建一个进程和一个主线程。CreateThread将在主线程的基础上创建一个新线程。

#### 原型:

```
HANDLE CreateThread(  
    LPSECURITY_ATTRIBUTES lpThreadAttributes,  
    DWORD dwStackSize,  
    LPTHREAD_START_ROUTINE lpStartAddress,  
    LPVOID lpParameter,  
  
    DWORD dwCreationFlags,
```

```
LPDWORD lpThreadId
);

/*
- 第一个参数是指向SECURITY_ATTRIBUTES型态的结构指针。在Windows 98中忽略该参数。在Windows NT
中，它被设为NULL。

- 第二个参数是用于新线程的初始堆栈大小，默认值为0。在任何情况下，Windows根据需要动态延长堆栈的大小。

- 第三个参数是指向线程函数的指标。函数名称没有限制，但是必须以下列形式声明：
DWORD WINAPI ThreadProc (PVOID pParam) ;

- 第四个参数为传递给ThreadProc的参数。这样主线程和从属线程就可以共享数据。

- 第五个参数通常为0，但当建立的线程不马上执行时为旗标CREATE_SUSPENDED。线程将暂停直到呼叫
ResumeThread来恢复线程的执行为止。

- 第六个参数是一个指标，指向接受执行线程ID值的变量。
*/
```

#### 注意：

临界区要在线程执行前初始化，因为线程一旦被建立即开始运行（除非手工挂起），但线程建立后在初始化临界区可能出现问題。

## 实验四：银行家算法的模拟与实现

### 一、实验题目

本实验的内容是要通过编写和调试一个模拟系统动态分配资源的银行家算法程序，有效地避免死锁发生。具体要求如下：（1）初始化时让系统拥有一定的资源；（2）用键盘输入的方式允许进程动态申请资源；（3）如果试探分配后系统处于安全状态，则修改系统的资源分配情况，正式分配资源；（4）如果试探分配后系统处于不安全状态，则提示不能满足请求，恢复原状态并阻塞该进程。

### 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

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。

### 四、详细设计

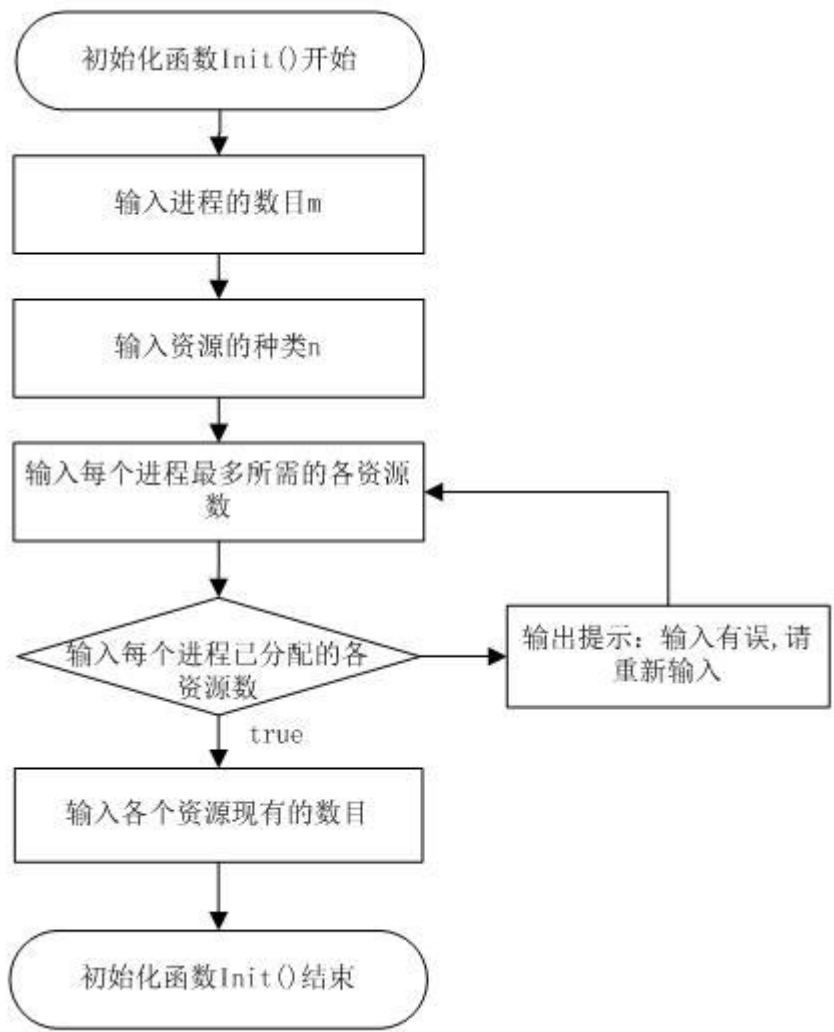
## 1. 数据结构：

- 1) 可利用资源向量Available 是个含有m个元素的数组，其中的每一个元素代表一类可利用的资源数目。如果 $Available[j]=K$ ，则表示系统中现有 $R_j$ 类资源K个。
- 2) 最大需求矩阵Max 这是一个 $n \times m$ 的矩阵，它定义了系统中n个进程中的每一个进程对m类资源的最大需求。如果 $Max[i,j]=K$ ，则表示进程i需要 $R_j$ 类资源的最大数目为K。
- 3) 分配矩阵Allocation 这也是一个 $n \times m$ 的矩阵，它定义了系统中每一类资源当前已分配给每一进程的资源数。如果 $Allocation[i,j]=K$ ，则表示进程i当前已分得 $R_j$ 类资源的数目为K。
- 4) 需求矩阵Need。 这也是一个 $n \times m$ 的矩阵，用以表示每一个进程尚需的各类资源数。如果 $Need[i,j]=K$ ，则表示进程i还需要 $R_j$ 类资源K个，方能完成其任务。  
 $Need[i,j] = Max[i,j] - Allocation[i,j]$

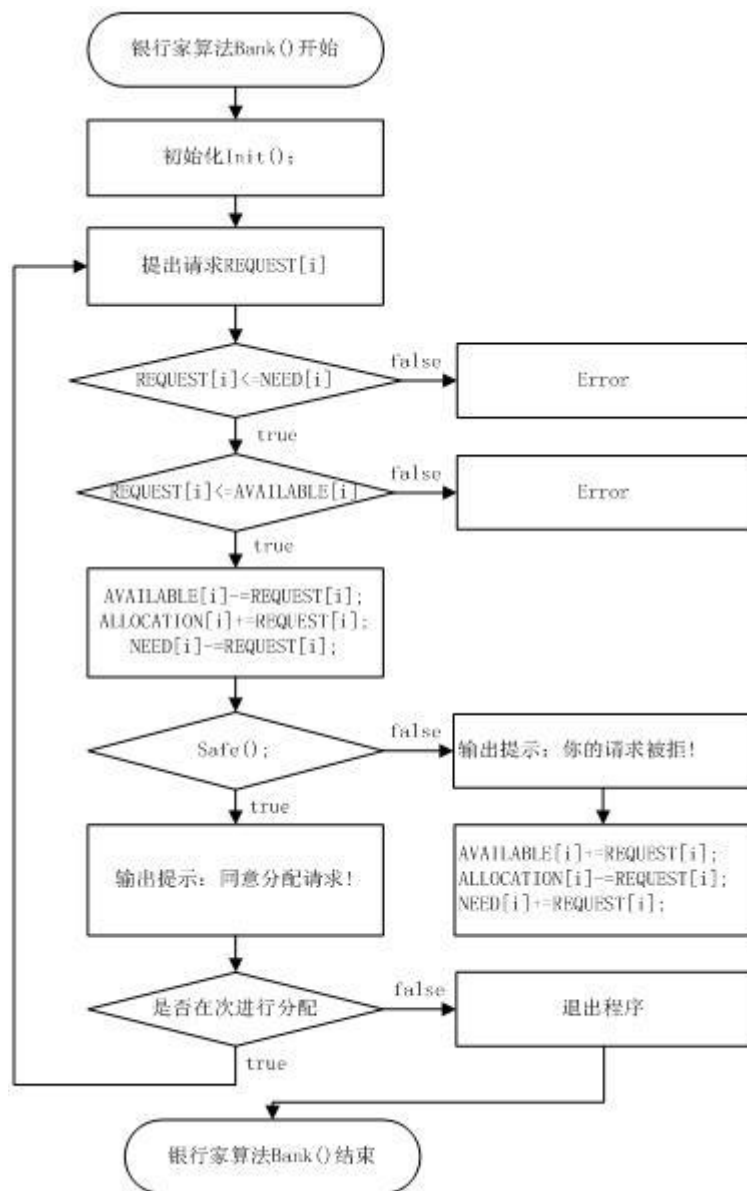
## 2. 算法流程图：

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。

Init 初始化算法流程图：

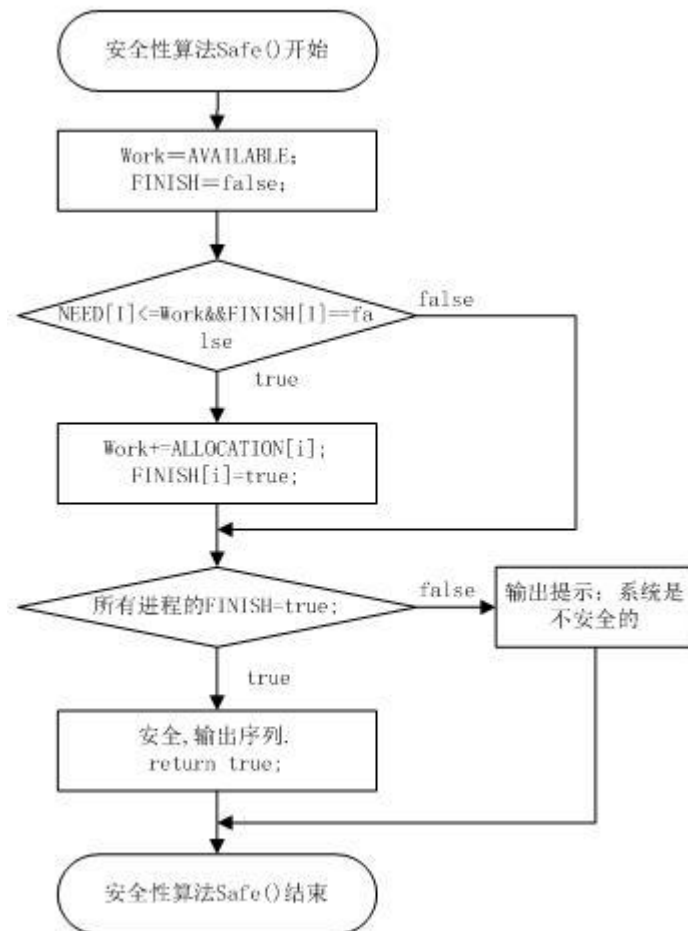


Bank 银行家算法流程图：



Safe 安全性算法流程图:





## 五、实验结果与分析

### 1. 程序运行结果：

```

E:\233\bin\Debug\233.exe
2 3
4 5
[ALLOCATION] 已分配各个资源数量3*2的矩阵
1 0
2 2
2 3
请输入系统各种资源初始化数量2列
6 7

-----Allocation-----
port0    1      0
port1    2      2
port2    2      3
-----Need-----
port0    0      1
port1    0      1
port2    2      2

-----
哪个进程请求申请资源
1
此进程每类资源的请求个数
0 1
T0时刻系统安全安全序列为 0-->1-->2
输入任意字符继续
  
```

## 2. 实验结果分析：

在避免死锁的方法中，所施加的限制条件较弱，有可能获得令人满意的系统性能。在该方法中把系统的状态分为安全状态和不安全状态，只要能使系统始终都处于安全状态，便可以避免发生死锁。

银行家算法的基本思想是分配资源之前，判断系统是否是安全的；若是，才分配。它是最具有代表性的避免死锁的算法。

设进程 $cusneed$  提出请求 $REQUEST[i]$ ，则银行家算法按如下规则进行判断。

(1) 如果 $REQUEST[cusneed][i] \leq NEED[cusneed][i]$ ，则转(2)；否则，出错。

(2) 如果 $REQUEST[cusneed][i] \leq AVAILABLE[cusneed][i]$ ，则转(3)；否则，出错。

(3) 系统试探分配资源，修改相关数据：

```
AVAILABLE[i] -= REQUEST[cusneed][i];
```

```
ALLOCATION[cusneed][i] += REQUEST[cusneed][i];
```

```
NEED[cusneed][i] -= REQUEST[cusneed][i];
```

(4) 系统执行安全性检查，如安全，则分配成立；否则试探性分配作废，系统恢复原状，进程等待。

## 六、小结与心得体会

### 1.1 银行家算法的实现思想

允许进程动态地申请资源，系统在每次实施资源分配之前，先计算资源分配的安全性，若此次资源分配安全（即资源分配后，系统能按某种顺序来为每个进程分配其所需的资源，直至最大需求，使每个进程都可以顺利地完），便将资源分配给进程，否则不分配资源，让进程等待。

### 1.2 死锁的概念

死锁是指两个或两个以上的进程在执行过程中，由于竞争资源或者由于彼此通信而造成的一种阻塞的现象，若无外力作用，它们都将无法推进下去。此时称系统处于死锁状态或系统产生了死锁，这些永远在互相等待的进程称为死锁进程。

银行家算法是避免死锁的一种重要方法。操作系统按照银行家制定的规则为进程分配资源，当进程首次申请资源时，要测试该进程对资源的最大需求量，如果系统现存的资源可以满足它的最大需求量则按当前的申请量分配资源，否则就推迟分配。当进程在执行中继续申请资源时，先测试该进程已占用的资源数与本次申请的资源数之和是否超过了该进程对资源的最大需求量。若超过则拒绝分配资源，若没有超过则再测试系统现存的资源能否满足该进程尚需的最大资源量，若能满足则按当前的申请量分配资源，否则也要推迟分配。

### 1.3 产生死锁的必要条件

① 互斥条件：指进程对所分配到的资源进行排它性使用，即在一段时间内某资源只由一个进程占用。如果此时还有其它进程请求资源，则请求者只能等待，直至占有资源的进程用毕释放。

② 请求和保持条件：指进程已经保持至少一个资源，但又提出了新的资源请求，而该资源已被其它进程占有，此时请求进程阻塞，但又对自己已获得的其它资源保持不放。

- ③ 不剥夺条件：指进程已获得的资源，在未使用完之前，不能被剥夺，只能在使用完时由自己释放。
- ④ 环路等待条件：指在发生死锁时，必然存在一个进程——资源的环形链，即进程集合{P0, P1, P2, ..., Pn}中的P0正在等待一个P1占用的资源；P1正在等待P2占用的资源，.....，Pn正在等待已被P0占用的资源。

## 实验五：内存管理

### 一、实验题目

- (1) 观察和调整 Windows XP/7 的内存性能。
- (2) 了解和检测进程的虚拟内存空间。

### 二、实验目的

(1) 通过对 Windows xp/7“任务管理器”、“计算机管理”、“我的电脑”属性、“系统信息”、“系统监视器”等程序的应用，学习如何察看和调整 Windows 的内存性能，加深对操作系统内存管理、虚拟存储管理等理论知识的理解。(2) 了解 Windows xp/7 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系。

### 三、总体设计

1. Windows 提供了一整套能使用户精确控制应用程序的虚拟地址空间的虚拟内存 API。
2. 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

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

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。

程序主要包括：

- main() 函数：控制程序整体执行流程，显示虚拟内存的基本信息，遍历当前进程的虚拟内存。
- ShowVirtualMemory()
- WalkVM()
- ShowProtection()
- TestSet()

### 四、详细设计

实验代码：

```
// 工程 vmwalker
#include <windows.h>
#include <iostream>

#include <shlwapi.h>
```

```

#include <iomanip>
#pragma comment(lib, "Shlwapi.lib")
// 以可读方式对用户显示保护的辅助方法。
// 保护标记表示允许应用程序对内存进行访问的类型
// 以及操作系统强制访问的类型
inline bool TestSet(DWORD dwTarget, DWORD dwMask)
{
    return ((dwTarget & dwMask) == dwMask) ;
}
# define SHOWMASK(dwTarget, type) if (TestSet(dwTarget, PAGE_##type) ) {std :: cout <<
", " << type; }
void ShowProtection(DWORD dwTarget)
{
    SHOWMASK(dwTarget, READONLY) ;
    SHOWMASK(dwTarget, GUARD) ;
    SHOWMASK(dwTarget, NOCACHE) ;
    SHOWMASK(dwTarget, READWRITE) ;
    SHOWMASK(dwTarget, WRITECOPY) ;
    SHOWMASK(dwTarget, EXECUTE) ;
    SHOWMASK(dwTarget, EXECUTE_READ) ;
    SHOWMASK(dwTarget, EXECUTE_READWRITE) ;
    SHOWMASK(dwTarget, EXECUTE_WRITECOPY) ;
    SHOWMASK(dwTarget, NOACCESS) ;
}
// 遍历整个虚拟内存并对用户显示其属性的工作程序的方法
void WalkVM(HANDLE hProcess)
{
    // 首先, 获得系统信息
    SYSTEM_INFO si;
    :: ZeroMemory(&si, sizeof(si) ) ;
    :: GetSystemInfo(&si) ;
    // 分配要存放信息的缓冲区
    MEMORY_BASIC_INFORMATION mbi;
    :: ZeroMemory(&mbi, sizeof(mbi) ) ;
    // 循环整个应用程序地址空间
    LPCVOID pBlock = (LPVOID) si.lpMinimumApplicationAddress;
    while (pBlock < si.lpMaximumApplicationAddress)
    {
        // 获得下一个虚拟内存块的信息
        if (:: VirtualQueryEx(
            hProcess,
            pBlock,
            &mbi,
            sizeof(mbi))==sizeof(mbi) )
        // 相关的进程
        // 开始位置
        // 缓冲区
        // 大小的确认
        {
            // 计算块的结尾及其大小
            LPCVOID pEnd = (PBYTE) pBlock + mbi.RegionSize;
            TCHAR szSize[MAX_PATH];

            :: StrFormatByteSize(mbi.RegionSize, szSize, MAX_PATH) ;

```

```

// 显示块地址和大小
std :: cout.fill ('0') ;
std :: cout
    << std :: hex << std :: setw(8) << (DWORD) pBlock << "-"
    << std :: hex << std :: setw(8) << (DWORD) pEnd << (::
strlen(szSize)==7? " (" : " (") << szSize << " ) " ;
// 显示块的状态
switch(mbi.State)
{
case MEM_COMMIT :
    std :: cout << "Committed" ;
    break;
case MEM_FREE :
    std :: cout << "Free" ;
    break;
case MEM_RESERVE :
    std :: cout << "Reserved" ;
    break;
}

// 显示保护
if(mbi.Protect==0 && mbi.State!=MEM_FREE)
{
    mbi.Protect=PAGE_READONLY;
}
ShowProtection(mbi.Protect);

// 显示类型
switch(mbi.Type)
{
case MEM_IMAGE :
    std :: cout << ", Image" ;
    break;
case MEM_MAPPED:
    std :: cout << ", Mapped";
    break;
case MEM_PRIVATE :
    std :: cout << ", Private" ;
    break;
}

// 检验可执行的影像
TCHAR szFilename [MAX_PATH] ;
if (:: GetModuleFileName (
    (HMODULE) pBlock,
    szFilename,
    MAX_PATH)>0)

// 实际虚拟内存的模块句柄
//完全指定的文件名称
//实际使用的缓冲区大小
{
// 除去路径并显示
    :: PathStripPath(szFilename) ;
    std :: cout << ", Module: " << szFilename;
}

std :: cout << std :: endl;

```

```

// 移动块指针以获得下一个块
        pBlock = pEnd;
    }
}

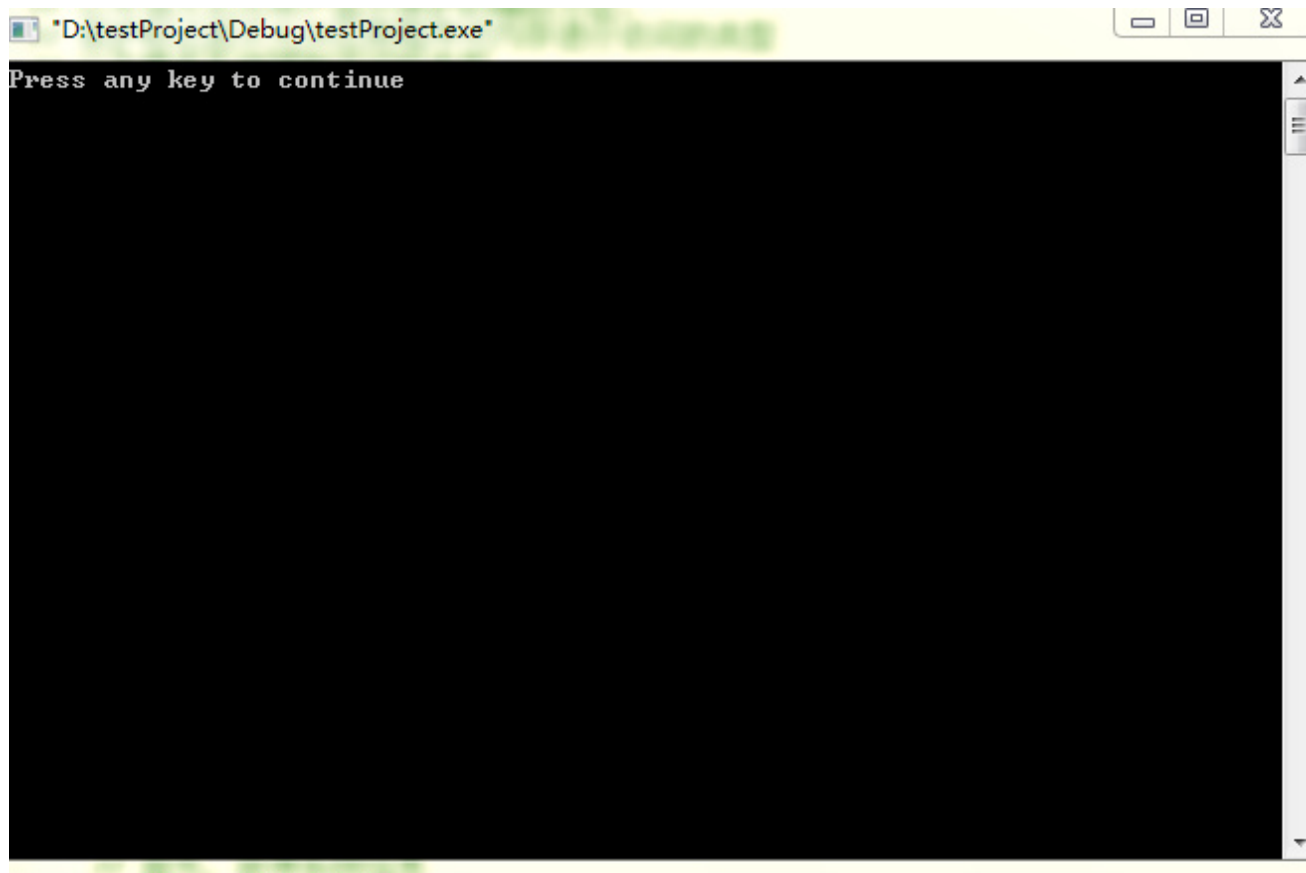
void ShowVirtualMemory()
{
// 首先, 让我们获得系统信息
    SYSTEM_INFO si;
    ::ZeroMemory(&si, sizeof(si) );
    ::GetSystemInfo(&si) ;
// 使用外壳辅助程序对一些尺寸进行格式化
    TCHAR szPageSize[MAX_PATH];
    ::StrFormatByteSize(si.dwPageSize, szPageSize, MAX_PATH) ;
    25
    DWORD dwMemSize = (DWORD)si.lpMaximumApplicationAddress - (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)
        << (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());
}

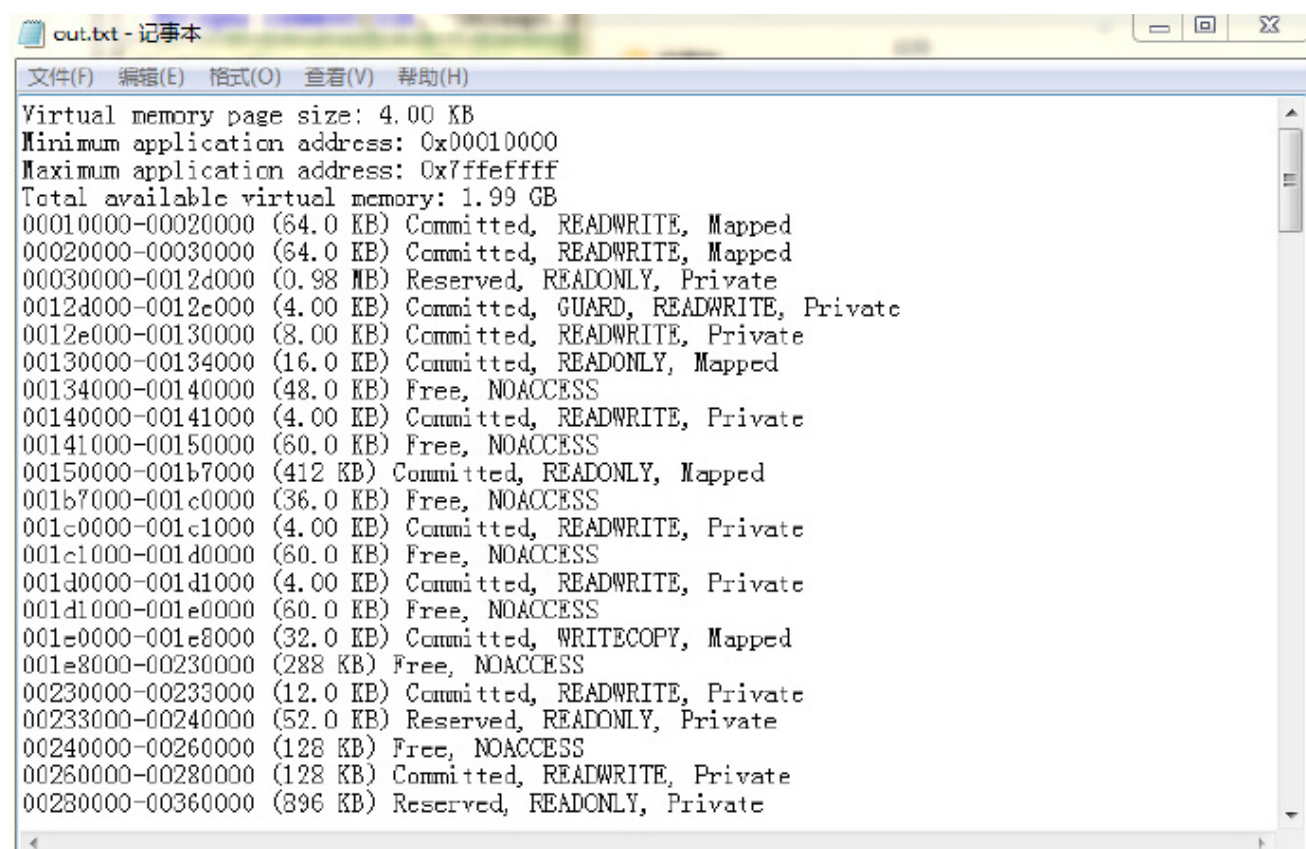
```

## 五、实验结果与分析

程序运行截图：



程序新建文件内容截图：



实验结果分析：

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 232 字节或 4000 兆字节 (4GB)。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。

## 六、小结与心得体会

(1) 分页过程 当 Windows 求助于硬盘以获得虚拟内存时，这个过程被称为分页 (paging)。分页就是将信息从主内存移动到磁盘进行临时存储的过程。应用程序将物理内存和虚拟内存视为一个独立的实体，甚至不知道 Windows 使用了两种内存方案，而认为系统拥有比实际内存更多的内存。例如，系统的内存数量可能只有 256MB，但每一个应用程序仍然认为有 4GB 内存可供使用。使用分页方案带来了很多好处，不过这是有代价的。当进程需要已经交换到硬盘上的代码或数据时，系统要将数据送回物理内存，并在必要时将其其他信息传输到硬盘上，而硬盘与物理内存存在性能上的差异极大。例如，硬盘的访问时间通常大约为 4-10 毫秒，而物理内存的访问时间为 60 us，甚至更快。(2) 内存共享 应用程序经常需要彼此通信和共享信息。为了提供这种能力，Windows 必须允许访问某些内存空间而不危及它和其他应用程序的安全性和完整性。从性能的角度来看，共享内存的能力大大减少了应用程序使用的内存数量。运行一个应用程序的多个副本时，每一个实例都可以使用相同的代码和数据，这意味着不必维护所加载应用程序代码的单独副本并使用相同的内存资源。无论正在运行多少个应用程序实例，充分支持应用程序代码所需求的内存数量都相对保持不变。(3) 未分页合并内存与分页合并内存 Windows 决定了系统内存组件哪些可以以及哪些不可以交换到磁盘上。显然，不应该将某些代码 (例如内核) 交换出主内存。因此，Windows 将系统使用的内存进一步划分为未分页合并内存和分页合并内存。分页合并内存是存储迟早需要的可分页代码或数据的内存部分。虽然可以将分页合并内存中的任何系统进程交换到磁盘上，但是它临时存储在主内存的这一部分，以防系统立刻需要它。在将系统进程交换到磁盘上之前，Windows 会交换其他进程。未分页合并内存包含必须驻留在内存中的占用代码或数据。这种结构类似于早期的 MS-DOS 程序使用的结构，在 MS-DOS 中，相对较小的终止并驻留程序 (Terminate and Stay Resident, TSR) 在启动时加载到内存中。这些程序在系统重新启动或关闭之前一直驻留在内存的特定部分中。例如，防病毒程序将加载为 TSR 程序，以预防可能的病毒袭击。未分页合并内存中包含的进程保留在主内存中，并且不能交换到磁盘上。物理内存的这个部分用于内核模式操作 (例如，驱动程序) 和必须保留在主内存中才能有效工作的其他进程。没有主内存的这个部分，内核组件就将是可分页的，系统本身就有变得不稳定的危险。分配到未分页内存池的主内存数量取决于服务器拥有的物理内存数量以及进程对系统上的内存地址空间的需求。不过，Windows XP 将未分页合并内存限制为 256MB (在 Windows NT 4 中的限制为 128MB)。根据系统中的物理内存数量，复杂的算法在启动时动态确定 Windows XP 系统上的未分页合并内存的最大数量。Windows XP 内部的这一自我调节机制可以根据当前的内存配置自动调整大小。例如，如果增加或减少系统中的内存数量，那么 Windows Xp 将自动调整未分页合并内存的大小，以反映这一更改。

## 实验六：磁盘调度

### 一、实验题目

(1) 观察和调整 Windows XP/7 的内存性能。



- (2) 了解和检测进程的虚拟内存空间。

## 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

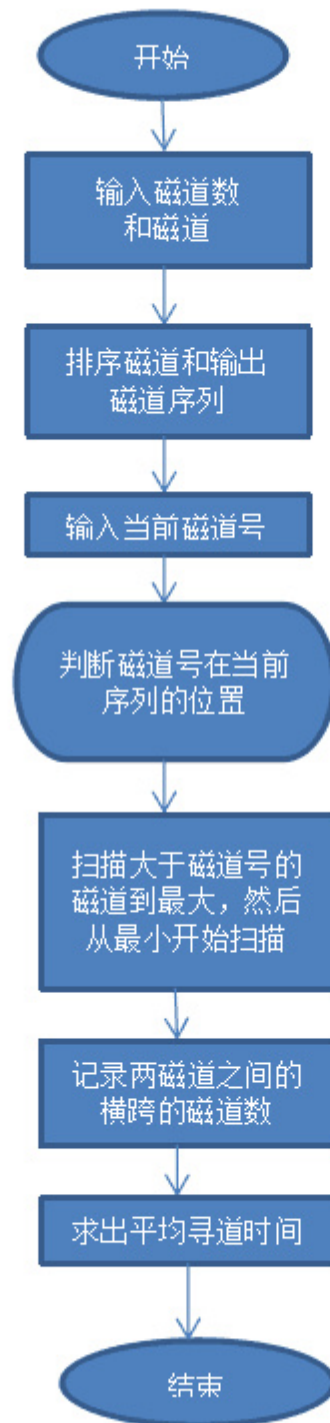
本实验通过编程模拟实现几种常见的磁盘调度算法。（1）测试数据：参见教材 P319-320，测试结果参见表 11.2。（2）使用 C 语言编程实现 FIFO、SSTF、SCAN、C-SCAN 算法（选做）。

程序主要包括：

- main() 主函数：控制整个程序流程；
- FFIO() 先进先出算法；
- SSTF() 最短服务时间算法；
- SCAN() 电梯扫描算法；
- CSCAN() 循环电梯算法。

## 四、详细设计

程序流程图：



程序代码：

```

/*
 *   Description: ---
 *   Author: Lynn
 *   Email: lgang219@gmail.com
 *   Create: 2018-07-05 21:08:39
 *   Last Modified: 2018-07-05 22:02:32
 */

```

```

#include "stdio.h"

```

```

#include<stdlib.h>
#define maxsize 1000 //定义最大数组域

//先进先出调度算法
void FIFO(int array[],int m)
{
    int sum=0,j,i,now;
    float avg;
    printf("\n 请输入当前的磁道号:");
    scanf("%d",&now);
    printf("\n FIFO 调度结果: ");
    printf("%d ",now);
    for(i=0; i<m; i++)
        printf("%d ",array[i]);
    sum=abs(now-array[0]);
    for(j=1; j<m; j++)
        sum+=abs(array[j]-array[j-1]); //累计总的移动距离
    avg=(float)sum/m; //计算平均寻道长度
    printf("\n 移动的总道数: %d \n",sum);
    printf(" 平均寻道长度: %f \n",avg);
}

//最短服务时间优先调度算法
void SSTF(int array[],int m)
{
    int temp;
    int k=1;
    int now,l,r;
    int i,j,sum=0;
    float avg;
    for(i=0; i<m; i++)
    {
        for(j=i+1; j<m; j++) //对磁道号进行从小到大排列
        {
            if(array[i]>array[j]) //两磁道号之间比较
            {
                temp=array[i];
                array[i]=array[j];
                array[j]=temp;
            }
        }
    }
    for( i=0; i<m; i++) //输出排序后的磁道号数组
        printf("%d ",array[i]);
    printf("\n 请输入当前的磁道号:");
    scanf("%d",&now);
    printf("\n SSTF 调度结果: ");
    if(array[m-1]<=now) //判断整个数组里的数是否都小于当前磁道号
    {
        for(i=m-1; i>=0; i--) //将数组磁道号从大到小输出
            printf("%d ",array[i]);
        sum=now-array[0]; //计算移动距离
    }
}

```

```

else if(array[0]>=now)//判断整个数组里的数是否都大于当前磁道号
{
    for(i=0; i<m; i++) //将磁道号从小到大输出
        printf("%d ",array[i]);
    sum=array[m-1]-now;//计算移动距离
}
else
{
    while(array[k]<now)//逐一比较以确定 k 值
    {
        k++;
    }

    l=k-1;
    r=k;
    //确定当前磁道在已排的序列中的位置
    while((l>=0)&&(r<m))
    {
        if((now-array[l])<=(array[r]-now))//判断最短距离
        {
            printf("%d ",array[l]);
            sum+=now-array[l];//计算移动距离
            now=array[l];
            l=l-1;
        }
        else
        {
            printf("%d ",array[r]);
            sum+=array[r]-now;//计算移动距离
            now=array[r];
            r=r+1;
        }
    }
    if(l== -1)
    {
        for(j=r; j<m; j++)
        {
            printf("%d ",array[j]);
        }
        sum+=array[m-1]-array[0];//计算移动距离
    }
    else
    {
        for(j=l; j>=0; j--)
        {
            printf("%d ",array[j]);
        }
        sum+=array[m-1]-array[0];//计算移动距离
    }
}
avg=(float)sum/m;
printf("\n 移动的总道数: %d \n",sum);

printf(" 平均寻道长度: %f \n",avg);

```

```

}

// 扫描算法
int SCAN(int array[], int m)
{
    int now=0;
    printf("SCAN 扫描算法");

    printf("\n 请输入当前的磁道号:");
    scanf("%d",&now);

    int temp=0;
    // 排序
    for(int i=0; i<m; i++)
    {
        for(int j=i+1; j<m; j++) //对磁道号进行从小到大排列
        {
            if(array[i]>array[j])//两磁道号之间比较
            {
                temp=array[i];
                array[i]=array[j];
                array[j]=temp;
            }
        }
    }

    // 输出结果
    int sum=0;
    float avg_steps=0;
    for(int i=0; i<m; i++)
    {
        if(array[i]>=now)
        {
            printf("\nSCAN 调度结果: %d ",now);
            for(int j=i; j<m; j++)
            {
                printf("%d ",array[j]);
                sum+=abs(array[j]-now);
                now=array[j];
            }

            if((i-1)!=0)
            {
                for(int j=i-1; j>=0; j--)
                {
                    printf("%d ",array[j]);
                    sum+=abs(array[j]-now);
                    now=array[j];
                }
            }
            avg_steps=float(sum/m);
            printf("\n移动的总道数: %d\n",sum);
            printf("平均寻道长度: %f\n\n",avg_steps);

            // 直接退出

```

```

        return 0;
    }
}
else
{
    if((i+1)==m)
    {
        printf("\nSCAN 调度结果: %d ", now);
        for(int j=i; j>=0; j--)
        {
            printf("%d ", array[j]);
            sum+=abs(array[j]-now);
            now=array[j];
        }
        avg_steps=float(sum/m);
        printf("\n移动的总道数: %d\n", sum);
        printf("平均寻道长度: %f\n\n", avg_steps);
    }
}

return 0;
}

// CSCAN 扫描算法
int CSCAN(int array[], int m)
{
    int now=0;
    printf("CSCAN 扫描算法");

    printf("\n 请输入当前的磁道号:");
    scanf("%d", &now);

    int temp=0;
    // 排序
    for(int i=0; i<m; i++)
    {
        for(int j=i+1; j<m; j++) //对磁道号进行从小到大排列
        {
            if(array[i]>array[j])//两磁道号之间比较
            {
                temp=array[i];
                array[i]=array[j];
                array[j]=temp;
            }
        }
    }

    // 输出结果
    for(int i=0; i<m; i++)
    {
        if(array[i]>=now)

```

```

    {
        printf("\nCSCAN 调度结果: %d ", now);
        for(int j=i; j<m; j++)
            printf("%d ", array[j]);
        for(int j=0; j<i; j++)
            printf("%d ", array[j]);
        // 直接退出
        return 0;
    }
    else
    {
        {
            if((i+1)==m)
            {
                printf("\nCSCAN 调度结果: %d ", now);
                for(int j=0; j<m; j++)
                    printf("%d ", array[j]);
            }
        }
    }
    return 0;
}

// 操作界面
int main()
{
    int c;
    int count;
    //int m=0;
    int cidao[maxsize]; //定义磁道号数组
    int i=0;
    int b;
    printf("\n ----- \n");
    printf("磁盘调度算法模拟");
    printf("\n ----- \n");
    printf("请先输入磁道数量: \n");
    scanf("%d", &b);
    printf("请先输入磁道序列: \n");

    for(i=0; i<b; i++)
    {
        scanf("%d", &cidao[i]);
    }
    printf("\n 磁道读取结果: \n");

    for(i=0; i<b; i++)
    {
        printf("%d ", cidao[i]); //输出读取的磁道的磁道号
    }

    count=b;
    printf("\n ");

```

```

while(1)
{
    printf("\n 算法选择:\n");
    printf(" 1、先进先出算法(FIFO)\n");
    printf(" 2、最短服务时间优先算法(SSTF)\n");
    printf(" 3、扫描算法(SCAN)\n");
    printf(" 4、循环扫描算法(C-SCAN)\n");
    printf(" 5. 退出\n");
    printf("\n");
    printf("请选择:");

    scanf("%d",&c);
    if(c>5)
        break;

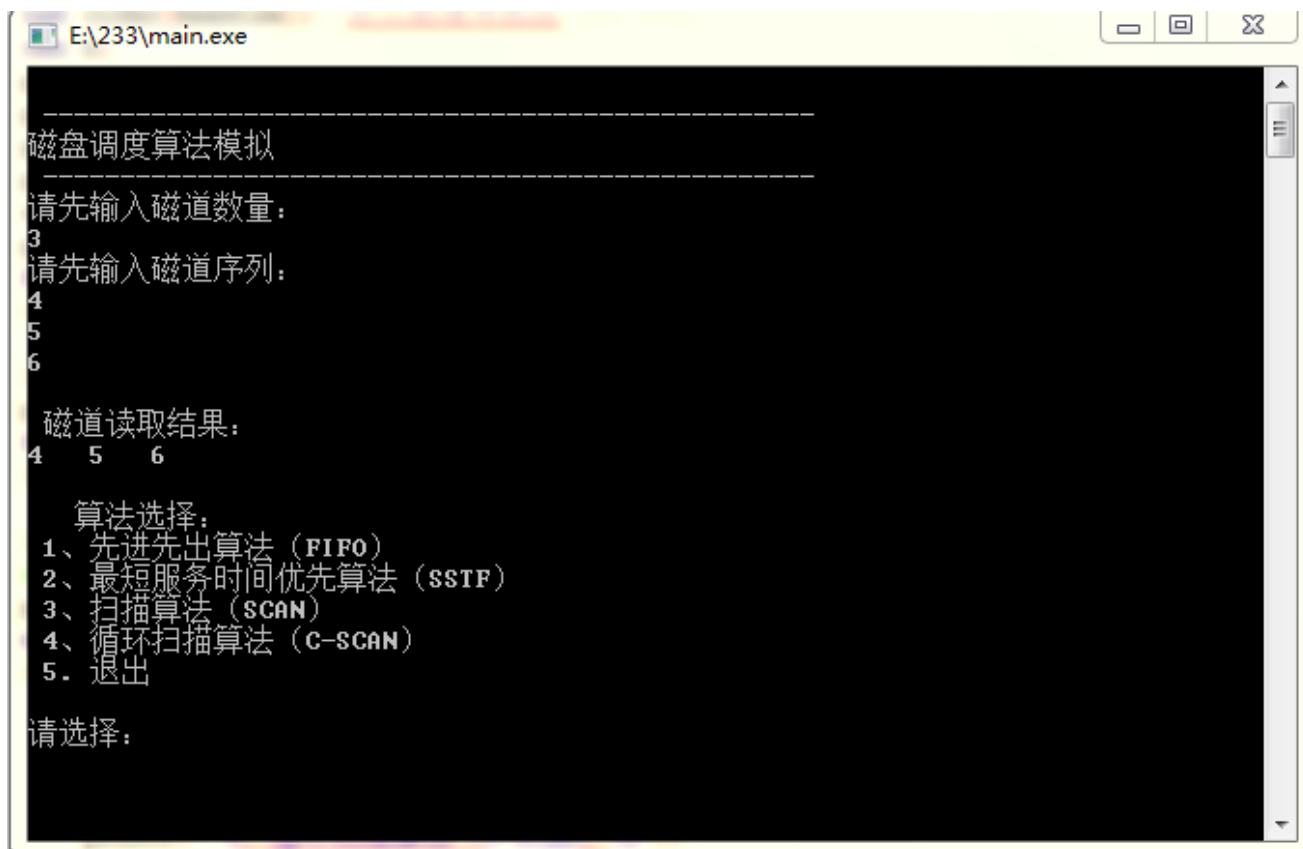
    switch(c)//算法选择
    {
        case 1:
            FIFO(cidao,count);//先进先出算法
            printf("\n");
            break;
        case 2:
            SSTF(cidao,count);//最短服务时间优先算法
            printf("\n");
            break;
        case 3:
            SCAN(cidao,count);//扫描算法,待补充!
            printf("\n");
            break;
        case 4:
            CSCAN(cidao,count);//循环扫描算法,待补充!
            printf("\n");
            break;
        case 5:
            exit(0);
            //
            //
    }
}
return 0;
}

```

## 五、实验结果与分析

程序运行结果：





#### 实验结果分析:

##### 1. 关于FIFO算法:

- 1、算法思想: 按访问请求到达的先后次序服务。
- 2、优点: 简单, 公平。
- 3、缺点: 效率不高, 相邻两次请求可能会造成最内到最外的柱面寻道, 使磁头反复移动, 增加了服务时间, 对机械也不利。

##### 2. 关于SSTF算法:

- 1、算法思想: 优先选择距当前磁头最近的访问请求进行服务, 主要考虑寻道优先。
- 2、优点: 改善了磁盘平均服务时间。
- 3、缺点: 造成某些访问请求长期等待得不到服务。

##### 3. 关于SCAN算法:

当设备无访问请求时, 磁头不动; 当有访问请求时, 磁头按一个方向移动, 在移动过程中对遇到的访问请求进行服务, 然后判断该方向上是否还有访问请求, 如果有则继续扫描; 否则改变移动方向, 并为经过的访问请求服务, 如此反复。

##### 4. 关于循环扫描算法:

采用SCAN算法和C-SCAN算法时磁头总是严格地遵循从盘面的一端到另一端, 显然, 在实际使用时还可以改进, 即磁头移动只需要到达最远端的一个请求即可返回, 不需要到达磁盘端点。这种形式的SCAN算法和C-SCAN算法称为LOOK和C-LOOK调度。这是因为它们在朝一个给定方向移动前会查看是否有请求。注意, 若无特别说明, 也可以默认SCAN算法和C-SCAN算法为LOOK和C-LOOK调度。

## 六、小结与心得体会

- FIFO

FCFS算法根据进程请求访问磁盘的先后顺序进行调度，这是一种最简单的[调度算法](#)。该算法的优点是具有公平性。如果只有少量进程需要访问，且大部分请求都是访问簇聚的文件扇区，则有望达到较好的性能；但如果大量进程竞争使用磁盘，那么这种算法在性能上往往接近于随机调度。所以，实际磁盘调度中考虑一些更为复杂的调度算法。[1][ ](undefined)

- 1、算法思想：按访问请求到达的先后次序服务。
- 2、优点：简单，公平。
- 3、缺点：效率不高，相邻两次请求可能会造成最内到最外的柱面寻道，使磁头反复移动，增加了服务时间，对机械也不利。

- SSTF

SSTF算法选择调度处理的磁道是与当前磁头所在磁道距离最近的磁道，以使每次的寻找时间最短。当然，总是选择最小寻找时间并不能保证平均寻找时间最小，但是能提供比FCFS算法更好的性能。这种算法会产生“饥饿”现象。

- 1、算法思想：优先选择距当前磁头最近的访问请求进行服务，主要考虑寻道优先。
- 2、优点：改善了磁盘平均服务时间。
- 3、缺点：造成某些访问请求长期等待得不到服务。

- SCAN 电梯算法

SCAN算法在磁头当前移动方向上选择与当前磁头所在磁道距离最近的请求作为下一次服务的对象。由于磁头移动规律与电梯运行相似，故又称为电梯调度算法。SCAN算法对最近扫描过的区域不公平，因此，它在访问局部性方面不如FCFS算法和SSTF算法好。

- CSCAN 循环电梯算法

在扫描算法的基础上规定磁头单向移动来提供服务，回返时直接快速移动至起始端而不服务任何请求。由于SCAN算法偏向于处理那些接近最里或最外的磁道的访问请求，所以使用改进型的C-SCAN算法来避免这个问题。

采用SCAN算法和C-SCAN算法时磁头总是严格地遵循从盘面的一端到另一端，显然，在实际使用时还可以改进，即磁头移动只需要到达最远端的一个请求即可返回，不需要到达磁盘端点。这种形式的SCAN算法和C-SCAN算法称为LOOK和C-LOOK调度。这是因为它们在朝一个给定方向移动前会查看是否有请求。注意，若无特别说明，也可以默认SCAN算法和C-SCAN算法为LOOK和C-LOOK调度。