操作系统

课程设计指导书

(2016 级)

计算机科学与工程学院

内部使用

湖南科技大学

2018 年 6 月

一、课程设计目的

通过课程设计，加深学生对教材中的重要算法的理解，同时通过用 C 语言编程实现这些算法，并在 Linux 或 Windows 平台上实现，让学生更好地掌握操作系统的原理及实现方法，提高学生综合运用各专业课知识的能力。

二、课程设计要求

1. 每位同学准备实验本，上机前作好充分的准备工作，预习本次实验的内容，事先熟悉与实验有关的软硬件环境。
2. 实验时遵守实验室的规章制度，爱护实验设备，不得在实验室做与实验无关的事情，影响其他同学的上机。对于实验设备出现的问题，要及时向指导老师汇报。
3. 最终的实验报告要求格式规范，语言通顺，仔细记录实验中的数据、源程序、实验结果，对于实验过程中出现的问题或疑惑要一并书写，并作为重点加以思考。
4. 课程设计程序、实验报告要求独立完成、不允许抄袭。
5. 及时提交课程设计报告（主要包含课程设计目的、内容、步骤、结果及其分析，请参见最末的课程设计格式要求）和源程序文件。

三、课程设计考核

1. 建议实验一~实验六必做，其它实验选做。指导老师也可以根据实际情况进行调整。
2. 课程设计成绩主要由以下几部分组成：实验期间的表现情况、程序测试情况、实验报告质量。

目 录

实验一 Windows 进程管理 1
实验二 Linux 进程管理 8
实验三 互斥与同步 11
实验四 银行家算法的模拟与实现 15
实验五 内存管理 17
实验六 磁盘调度 27
实验七 进程间通信 32
实验八 简单二级文件系统设计 39
附录 1：Linux 编程基础 46
附录 2：课程设计报告参考格式 48

实验一 Windows 进程管理

1、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。

2、背景知识

Windows 所创建的每个进程都从调用 CreateProcess() API 函数开始，该函数的任务是在对象管理器子系统内初始化进程对象。每一进程都以调用 ExitProcess() 或 TerminateProcess() API 函数终止。通常应用程序的框架负责调用 ExitProcess() 函数。对于 C++ 运行库来说，这一调用发生在应用程序的 main() 函数返回之后。

（1） 创建进程

CreateProcess() 调用的核心参数是可执行文件运行时的文件名及其命令行。表 1-1 详细地列出了每个参数的类型和名称。

表 1-1 CreateProcess() 函数的参数

参数名称	使用目的
LPCTSTR lpApplivationName	全部或部分地指明包括可执行代码的 EXE 文件的文件名
LPCTSTR lpCommandLine	向可执行文件发送的参数
LPSECURIITY_ATTRIBUTES lpProcessAttributes	返回进程句柄的安全属性。主要指明这一句柄是否应该由其他子进程所继承
LPSECURIITY_ATTRIBUTES lpThreadAttributes	返回进程的主线程的句柄的安全属性
BOOL bInheritHandle	一种标志，告诉系统允许新进程继承创建者进程的句柄
DWORD dwCreationFlage	特殊的创建标志 (如 CREATE_SUSPENDED) 的位标记
LPVOID lpEnvironment	向新进程发送的一套环境变量；如为 null 值则发送调用者环境
LPCTSTR lpCurrentDirectory	新进程的启动目录
STARTUPINFO lpStartupInfo	STARTUPINFO 结构，包括新进程的输入和输出配置的详情
LPPROCESS_INFORMATION lpProcessInformation	调用的结果块；发送新应用程序的进程和主线程的句柄和 ID

可以指定第一个参数，即应用程序的名称，其中包括相对于当前进程的当前目录的全路径或者利用搜索方法找到的路径；lpCommandLine 参数允许调用者向新应用程序发送数据；接下来的三个参数与进程和它的主线程以及返回的指向该对象的句柄的安全性有关。

然后是标志参数，用以在 dwCreationFlags 参数中指明系统应该给予新进程什么行为。经常使用的标志是 CREATE_SUSPNDED，告诉主线程立刻暂停。当准备好时，应该使用 ResumeThread() API 来启动进程。另一个常用的标志是 CREATE_NEW_CONSOLE，告诉新进程启动自己的控制台窗口，而不是利用父窗口。这一参数还允许设置进程的优先级，用以向系统指明，相对于系统中所有其他的活动进程来说，给此进程多少 CPU 时间。

接着是 CreateProcess() 函数调用所需要的三个通常使用缺省值的参数。第一个参数是

lpEnvironment 参数，指明为新进程提供的环境；第二个参数是 lpCurrentDirectory，可用于向主创进程发送与缺省目录不同的新进程使用的特殊的当前目录；第三个参数是 STARTUPINFO 数据结构所必需的，用于在必要时指明新应用程序的主窗口的外观。

CreateProcess() 的最后一个参数是用于新进程对象及其主线程的句柄和 ID 的返回值缓冲区。以 PROCESS_INFORMATION 结构中返回的句柄调用 CloseHandle() API 函数是重要的，因为如果不将这些句柄关闭的话，有可能危及主创进程终止之前的任何未释放的资源。

（2） 正在运行的进程

如果一个进程拥有至少一个执行线程，则为正在系统中运行的进程。通常，这种进程使用主线程来指示它的存在。当主线程结束时，调用 ExitProcess() API 函数，通知系统终止它所拥有的所有正在运行、准备运行或正在挂起的其他线程。当进程正在运行时，可以查看它的许多特性，其中少数特性也允许加以修改。

首先可查看的进程特性是系统进程标识符 (PID) ，可利用 GetCurrentProcessId() API 函数来查看，与 GetCurrentProcess() 相似，对该函数的调用不能失败，但返回的 PID 在整个系统中都可使用。其他的可显示当前进程信息的 API 函数还有 GetStartupInfo()和 GetProcessShutdownParameters() ，可给出进程存活期内的配置详情。

通常，一个进程需要它的运行期环境的信息。例如 API 函数 GetModuleFileName() 和 GetCommandLine() ，可以给出用在 CreateProcess() 中的参数以启动应用程序。在创建应用程序时可使用的另一个 API 函数是 IsDebuggerPresent() 。

可利用 API 函数 GetGuiResources() 来查看进程的 GUI 资源。此函数既可返回指定进程中的打开的 GUI 对象的数目，也可返回指定进程中打开的 USER 对象的数目。进程的其他性能信息可通过 GetProcessIoCounters()、GetProcessPriorityBoost() 、GetProcessTimes() 和 GetProcessWorkingSetSize() API 得到。以上这几个 API 函数都只需要具有 PROCESS_QUERY_INFORMATION 访问权限的指向所感兴趣进程的句柄。

另一个可用于进程信息查询的 API 函数是 GetProcessVersion() 。此函数只需感兴趣进程的 PID (进程标识号) 。这一 API 函数与 GetVersionEx() 的共同作用，可确定运行进程的系统的版本号。

（3） 终止进程

所有进程都是以调用 ExitProcess() 或者 TerminateProcess() 函数结束的。但最好使用前者而不要使用后者，因为进程是在完成了它的所有的关闭“职责”之后以正常的终止方式来调用前者的。而外部进程通常调用后者即突然终止进程的进行，由于关闭时的途径不太正常，有可能引起错误的行为。

TerminateProcess() API 函数只要打开带有 PROCESS_TERMINATE 访问权的进程对象，就可以终止进程，并向系统返回指定的代码。这是一种“野蛮”的终止进程的方式，但是有时却是需要的。

如果开发人员确实有机会来设计“谋杀”(终止别的进程的进程) 和“受害”进程 (被终止的进程) 时，应该创建一个进程间通讯的内核对象——如一个互斥程序——这样一来，“受害”进程只在等待或周期性地测试它是否应该终止。

（4）进程同步

Windows 提供的常用对象可分成三类：核心应用服务、线程同步和线程间通讯。其中，开发人员可以使用线程同步对象来协调线程和进程的工作，以使其共享信息并执行任务。此类对象包括互锁数据、临界段、事件、互斥体和信号等。

多线程编程中关键的一步是保护所有的共享资源，工具主要有互锁函数、临界段和互斥体等；另一个实质性部分是协调线程使其完成应用程序的任务，为此，可利用内核中的事件对象和信号。

在进程内或进程间实现线程同步的最方便的方法是使用事件对象，这一组内核对象允许一个线程对其受信状态进行直接控制 (见表 1-2) 。

而互斥体则是另一个可命名且安全的内核对象，其主要目的是引导对共享资源的访问。拥有单

一访问资源的线程创建互斥体，所有想要访问该资源的线程应该在实际执行操作之前获得互斥体，而在访问结束时立即释放互斥体，以允许下一个等待线程获得互斥体，然后接着进行下去。

与事件对象类似，互斥体容易创建、打开、使用并清除。利用 CreateMutex() API 可创建互斥体，创建时还可以指定一个初始的拥有权标志，通过使用这个标志，只有当线程完成了资源的所有的初始化工作时，才允许创建线程释放互斥体。

表 1-2 用于管理事件对象的 API

API 名称	描述
CreateEvent()	在内核中创建一个新的事件对象。此函数允许有安全性设置、手工还是自动重置的标志以及初始时已接受还是未接受信号状态的标志
OpenEvent()	创建对已经存在的事件对象的引用。此 API 函数需要名称、继承标志和所需的访问级别
SetEvent()	将手工重置事件转化为已接受信号状态
ResetEvent()	将手工重置事件转化为非接受信号状态
PulseEvent()	将自动重置事件对象转化为已接受信号状态。当系统释放所有的等待它的线程时此种转化立即发生

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；然后，线程将这个句柄提供给一个等待函数。当内核将互斥体对象发送给等待线程时，就表明该线程获得了互斥体的拥有权。当线程获得拥有权时，线程控制了对共享资源的访问——必须设法尽快地放弃互斥体。放弃共享资源时需要在该对象上调用 ReleaseMute() API。然后系统负责将互斥体拥有权传递给下一个等待着的线程 (由到达时间决定顺序) 。

3、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，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。

4、给出实验结论

5、 程序清单

清单 1-1 一个简单的 Windows 控制台应用程序

```
// hello 项目
# include <iostream>
void main()
{
        std::cout << “Hello, Win32 Consol Application” << std :: endl ;
}
```

清单 1-2 创建子进程

```
#include <windows.h>
#include <iostream>
#include <stdio.h>

// 创建传递过来的进程的克隆过程并赋于其 ID 值
void StartClone(int nCloneID)
{
    // 提取用于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并通知其 EXE 文件名和克隆 ID
    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，并赋于其子进程的性质
    BOOL bCreateOK=::CreateProcess(
        szFilename,                 // 产生这个 EXE 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
        :: sscanf(argv[1] , "%d" , &nClone) ;
    }

//第二次修改：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;
}
```

清单 1-3　父子进程的简单通信及终止进程的示例程序

```
// procterm 项目
# 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
    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，并指明它是一个子进程
    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

    // 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，这个信号会发送给子进程的 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) ，重新编译执行

            // 准备好终止，清除句柄
            std :: cout << "Child quiting." << 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;

}
```

实验二 Linux 进程管理

1、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。

2、背景知识

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 系列有 6 个函数，原型如下：

```
extern char **environ;
int execl( const char *path, const char *arg, ...);
int execlp( const char *file, const char *arg, ...);
int execle( const char *path, const char *arg , ..., char * const envp[]);
int execv( const char *path, char *const argv[]);
int execve (const char *filename, char *const argv [], char *const envp[]);
int execvp( const char *file, char *const argv[]);
```

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 和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。

在函数 execl,execlp,和 execle 中，const char *arg 以及省略号代表的参数可被视为 arg0, arg1, ...,argn。它们合起来描述了指向 NULL 结尾的字符串的指针列表，即执行程序的参数列表。作为约定,第一个 arg 参数应该指向执行程序名自身,参数列表必须用 NULL 指针结束。

execv 和 execvp 函数提供指向 NULL 结尾的字符串的指针数组作为新程序的参数列表。作为约定，指针数组中第一个元素应该指向执行程序名自身。指针数组必须用 NULL 指针结束。

execle 函数同时说明了执行进程的环境(environment),它在 NULL 指针后面要求一个附加参数,NULL 指针用于结束参数列表,或者说,argv 数组。这个附加参数是指向 NULL 结尾的字符串的指针数组, 它必须用 NULL 指针结束。其它函数从当前进程的 environ 外部变量中获取新进程的环境。

execlp和execvp可根据path搜索合适的程序运行，其它则需要给出程序全路径。

execve（）类似 execv（），但是加上了环境的处理。

wait（）, waitpid（）可用来等待子进程结束。函数原型：

```
#include <sys/wait.h>
pid_t wait(int *stat_loc);
pid_t waitpid(pid_t pid, int *stat_loc,int options);
```

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。

waitpid 的第一个参数 pid 的意义：

pid > 0: 等待进程 id 为 pid 的子进程。

pid == 0: 等待与自己同组的任意子进程。

pid == -1: 等待任意一个子进程

pid < -1: 等待进程组号为-pid 的任意子进程。

因此，wait(&stat)等价于 waitpid(-1, &stat, 0)，waitpid 第三个参数 option 可以是 0，WNOHANG, WUNTRACED 或这几者的组合。

3、实验内容和步骤

请先快速阅读并完成附录 1：Linux 编程基础

（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/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。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。

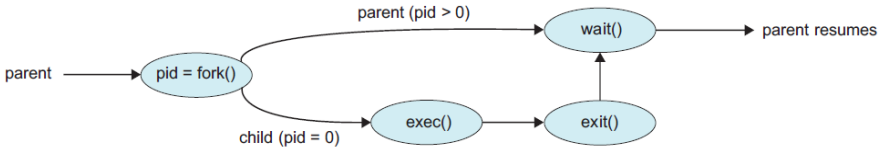


图 2-1 exec_demo.c 程序的执行过程

（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。

提示：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；
    }
```

4、给出实验分析与结论

5、 参考程序清单

清单 2-1　创建进程

```
int main ()
{
    int x;
    while((x=fork())==-1);
    if (x==0)
        printf("a");
    else
        printf("b");
    printf("c");
}
```

清单 2-2 子进程执行新任务

```
#include <sys/types.h>
#include <stdio.h>
#include <unistd.h>
int main()
{
    pid_t pid;
    /* fork a child process */
    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");
        }
    return 0;
}
```

实验三 互斥与同步

1、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。

2、实验内容和步骤

（1） 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 3-1 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 3-1 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 3-1 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 3-1，将信号量 Mutex 完全用 CreateSemaphore 及相关函数实现。写出要修改的语句。

（2） 读者写者问题（选做）

根据实验（1）中所熟悉的 P、V 原语对应的实际 Windows API 函数，并参考教材中读者、写者问题的算法原理，尝试利用 Windows API 函数实现第一类读者写者问题（读者优先）。

3、 给出实验分析与结论

4、 参考程序清单

清单 3-1 生产者消费者问题

```
#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_ccontinue = 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_ccontinue){
        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;
}
```

实验四 银行家算法的模拟与实现

1、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。

2、实验基本知识及原理

（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。

3、实验内容

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。

4、 银行家算法中的数据结构与流程图

（1）数据结构（详见教材 P177）：

资源总量向量 *Resource*，*m* 维，表示 *m* 种资源的总量。

可用资源向量 *Available*，*m* 维，表示未分配的各种可用资源数量。

需求矩阵 *Claim*，*n***m* 矩阵，表示 *n* 个进程对 *m* 类资源的最大需求。

分配矩阵 *Allocation* ，*n***m* 矩阵，表示 *n* 个进程已分配的各种资源数。

（2）算法流程图：请补充画出算法流程图。

5、 银行家算法编程实现

根据银行家算法及其数据结构，利用 C/C++语言进行编程实现。

6、 实验结果与分析

7、 实验总结

实验五 内存管理

1、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。

2、背景知识

耗尽内存是 Windows 系统中最常见的问题之一。当系统耗尽内存时，所有进程对内存的总需求超出了系统的物理内存总量。随后，Windows 必须借助它的虚拟内存来维持系统和进程的运行。虚拟内存机制是 Windows 操作系统的重要组成部分，但它的速度比物理内存慢得多，因此，应该尽量避免耗尽物理内存资源，以免导致性能下降。

解决内存不足问题的一个有效的方法就是添加更多的内存。但是，一旦提供了更多的内存，Windows 很可以会立即“吞食”。而事实上，添加更多的内存并非总是可行的，也可能只是推迟了实际问题的发生。因此，应该相信，优化所拥有的内存是非常关键的。

（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，并在必要时将其他信息传输到硬盘上，而硬盘与物理内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。

（4） 提高分页性能

只有一个物理硬盘驱动器的系统限制了优化分页性能的能力。驱动器必须处理系统和应用程序的请求以及对分页文件的访问。虽然物理驱动器可能有多个分区，但是将分页文件分布到多个分区的分页文件并不能提高硬盘驱动器的能力。只有当一个分区没有足够的空间来包含整个分页文件时，才将分页文件放在同一个硬盘的多个分区上。

拥有多个物理驱动器的服务器可以使用多个分页文件来提高分页性能。关键是将分页请求的负载分布到多个物理硬盘上。实际上，使用独立物理驱动器上的分页文件，系统可以同时处理多个分页请求。各个物理驱动器可以同时访问它自己的分页文件并写入信息，这将增加可以传输的信息量。多个分页文件的最佳配置是将各个分页文件放在拥有自己的控制器的独立驱动器上。不过，由于额外的费用并且系统上的可用中断很有限，因此对于大多数基于服务器的配置来说，这可能是不切实际的解决方案。

分页文件最重要的配置参数是大小。无论系统中有多少个分页文件，如果它们的大小不合适，那么系统就可能遇到性能问题。

如果初始值太小，那么系统可能必须扩大分页文件，以补偿额外的分页活动。当系统临时增加分页文件时，它必须在处理分页请求的同时创建新的空间。这时，系统将出现大量的页面错误，甚至可能出现系统失效。当系统必须在进程的工作区外部 (在物理内存或分页文件中的其他位置) 查找信息时，就会出现页面错误。当系统缺乏存储资源 (物理内存及虚拟内存) 来满足使用需求，从而遇到过多的分页时，就会出现系统失效。系统将花更多的时间来分页而不是执行应用程序。当系统失效时，Memory：Pages/see 计数器将持续高于每秒 100 页。系统失效严重降低了系统的性能。此外，动态扩展分页文件将导致碎片化。分页文件将散布在整个磁盘上而不是在启动时的连续空间中创建，从而增加了系统的开销，并导致系统性能降低。因此，应该尽量避免系统增加分页文件的大小。

> **提示：**
>
> 1) WINDOWS 中采用的虚拟存储管理方案是请求页式存储管理，分页文件就是我们原理课中所说的交换/对换文件，存放的内容是暂时被交换到外存中的进程页面。UNIX 使用的是交换分区，WINDOWS 使用的是交换文件。
>
> 2)在 NTFS 驱动器上，总是至少保留 25%的空闲驱动器空间，以确保可以在连续的空间中创建分页文件。
>
> 3) Windows xp 使用内存数量的 1.5 倍作为分页文件的最小容量，这个最小容量的两倍作为最大容量。它减少了系统因为错误配置的分页文件而崩溃的可能性。系统在崩溃之后能够将内存转储写入磁盘，所以系统分区必须有一个至少等于物理内存数量加上 1 的分页文件。

（5） Windows 虚拟内存

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 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。

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。

表 5-1 MEMORY_BASIC_INFORMATION 结构的成员

成员名称	目的
PVOID BaseAddress	虚拟内存区域开始处的指针
PVOID AllocationBase	如果这个特定的区域为子分配区的话，则为虚拟内存外面区域的指针；否则此值与 BaseAddress 相同
DWORD AllocationProtect	虚拟内存最初分配区域的保护属性。其可能值包括：PAGE_NOACCESS，PAGE_READONLY，PAGE_READWRITE 和 PAGE _EXECUTE_READ
DWORD RegionSize	虚拟内存区域的字节数
DWORD State	区域的当前分配状态。其可能值为 MEM_COMMIT，MEM_FREE 和 MEM_RESERVE
DWORD Protect	虚拟内存当前区域的保护属性。可能值与 AllocationProtect 成员的相同
DWORD Type	虚拟内存区域中出现的页面类型。可能值为 MEM_IMAGE，MEM_MAPPED 和 MEM_PRIVATE

Windows 还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。一些用于虚拟内存操作及检测的 API 见表 5-2 所示。

表 5-2 虚拟内存的 API

API 名称	描述
VirtualQueryEx()	通过填充 MEMORY_BASIC_INFORMATION 结构检测进程内虚拟内存的区域
VirtualAlloc()	保留或调配进程的部分虚拟内存，设置分配和保护标志
VirtualFree()	释放或收回应用程序使用的部分虚拟地址
VirtualProtect()	改变虚拟内存区域保护规范
VirtualLock()	防止系统将虚拟内存区域通过系统交换到页面文件中
VirtualUnlock()	释放虚拟内存的锁定区域，必要时，允许系统将其交换到页面文件中

程序清单 5-1 还显示了如何理解 Virtual QueryEX() API 填充的 MEMORY_BASIC_ INFORMATION 结构，如表 5-l 所示。此数据描述了进程虚拟内存空间中一组虚拟内存页面的当前

状态。其中 State 项表明这些区域是否为自由区、已调配区或保留区；Protect 项则包含了 Windows 系统为这些区域添加了何种访问保护；Type 项则表明这些区域是可执行图像、内存映射文件还是简单的私有内存。VirtualQueryEX() API 能让用户在指定的进程中，对虚拟内存地址的大小和属性进行检测。

提供虚拟内存分配功能的是 VirtualAlloc() API。该 API 支持用户向系统要求新的虚拟内存或改变已分配内存的当前状态。用户若想通过 VirtualAlloc() 函数使用虚拟内存，可以采用两种方式通知系统：

1) 简单地将内存内容保存在地址空间内；

2) 请求系统返回带有物理存储区 (RAM 的空间或换页文件) 的部分地址空间。

用户可以用 flAllocation Type 参数 (commit 和 reserve) 来定义这些方式，用户可以通知 Windows 按只读、读写、不可读写、执行或特殊方式来处理新的虚拟内存。

与 VirtualAlloc() 函数对应的是 VirtualFree() 函数，其作用是释放虚拟内存中的已调配页或保留页。用户可利用 dwFree Type 参数将已调配页修改成保留页属性。

VirtualProtect() 是 VirtualAlloc() 的一个辅助函数，利用它可以改变虚拟内存区的保护规范。

3、 实验内容和步骤

（1）观察和调整 Windows XP/7 的内存性能。

步骤 1：阅读“背景知识”，请回答：

1) 什么是“分页过程”？

2) 什么是“内存共享”？

3) 什么是“未分页合并内存”和“分页合并内存”？

Windows xp 中，未分页合并内存的最大限制是多少？

4) Windows xp 分页文件默认设置的最小容量和最大容量是多少？

步骤 2：登录进入 Windows xp。

步骤 3：查看包含多个实例的应用程序的内存需求。

1) 启动想要监视的应用程序，例如 Word。

2) 右键单击任务栏以启动“任务管理器”。

3) 在“Windows 任务管理器”对话框中选定“进程”选项卡。

4) 向下滚动在系统上运行的进程列表，查找想要监视的应用程序。

请在表 5-3 中记录：

表 5-3 实验记录

映像名称	PID	CPU	CPU 时间	内存使用

“内存使用”列显示了该应用程序的一个实例正在使用的内存数量。

5) 启动应用程序的另一个实例并观察它的内存需求。

请描述使用第二个实例占用的内存与使用第一个实例时的内存对比情况。

步骤 4：未分页合并内存。

估算未分页合并内存大小的最简单方法是使用“任务管理器”。未分页合并内存的估计值显示在“任务管理器”的“性能”选项卡的“核心内存”部分。

总数 (K) ：_____ 分页数：_____

未分页 (K) ：_____

还可以使用“任务管理器”查看一个独立进程正在使用的未分页合并内存数量和分页合并内存数量。操作步骤如下：

1) 单击“Windows 任务管理器”的“进程”选项卡，然后从“查看”菜单中选择“选择列”命令，显示“进程”选项卡的可查看选项。

2) 在“选择列”对话框中，选定“页面缓冲池”选项和“非页面缓冲池”选项旁边的复选框，然后单击“确定”按钮。

返回 Windows Xp“任务管理器”的“进程”选项卡时，将看到其中增加显示了各个进程占用的分页合并内存数量和未分页合并内存数量。

仍以刚才打开观察的应用程序 (例如 Word) 为例，请在表 5-4 中记录：

表 5-4 实验记录

映像名称	PID	内存使用	页面缓冲池	非页面缓冲池

从性能的角度来看，未分页合并内存越多，可以加载到这个空间的数据就越多。拥有的物理内存越多，未分页合并内存就越多。但未分页合并内存被限制为 256MB，因此添加超出这个限制的内存对未分页合并内存没有影响。

步骤 5：提高分页性能。

在 Windows xp 的安装过程中，将使用连续的磁盘空间自动创建分页文件(pagefile.sys) 。用户可以事先监视变化的内存需求并正确配置分页文件，使得当系统必须借助于分页时的性能达到最高。

虽然分页文件一般都放在系统分区的根目录下面，但这并不总是该文件的最佳位置。要想从分页获得最佳性能，应该首先检查系统的磁盘子系统的配置，以了解它是否有多个物理硬盘驱动器。

1) 在“开始”菜单中单击“设置” – “控制面板”命令，双击“管理工具”图标，再双击“计算机管理”图标。

2) 在“计算机管理”窗口的左格选择“磁盘管理”管理单元来查看系统的磁盘配置。

请在表 5-5 中记录：

表 5-5 实验记录

卷	布局	类型	文件系统	容量	状态

如果系统只有一个硬盘，那么建议应该尽可能为系统配置额外的驱动器。这是因为：Windows xp 最多可以支持在多个驱动器上分布的 16 个独立的分页文件。为系统配置多个分页文件可以实现对不同磁盘 I/O 请求的并行处理，这将大大提高 I/O 请求的分页文件性能。

步骤 6：计算分页文件的大小。

要想更改分页文件的位置或大小配置参数，可按以下步骤进行：

1) 右键单击桌面上的“我的电脑”（Win7 为计算机）图标并选定“属性”（Win7 为高级系统设置）。

2) 在“高级”选项卡上单击“性能选项”按钮。

3) 单击对话框中的“虚拟内存”区域中的“更改”按钮。

请记录：

所选驱动器 (C:) 的页面文件大小：

驱动器：_____ 可用空间：_____ MB

初始大小 (MB) ：_____ 最大值 (MB) ：_____

所选驱动器 (D:) 的页面文件大小：(如果有的话)

驱动器：_____ 可用空间：_____MB

初始大小 (MB) ：_____ 最大值 (MB) ：_____

所有驱动器页面文件大小的总数：

允许的最小值：_____ MB　　推荐：_____ MB

当前已分配：_____ MB

4) 要想将另一个分页文件添加到现有配置，在“虚拟内存”对话框中选定一个还没有分页文件的驱动器，然后指定分页文件的初始值和最大值 (以兆字节表示) ，单击“设置”，然后单击“确定”。

5) 要想更改现有分页文件的最大值和最小值，可选定分页文件所在的驱动器。然后指定分页文件的初始值和最大值，单击“设置”按钮，然后单击“确定”按钮。

6) 在“性能选项”对话框中单击“确定”按钮。

7) 单击“确定”按钮以关闭“系统特性”对话框。

（2）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，然后拷贝清单 5-1 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

虚拟内存每页容量为：_____　　最小应用地址：_____

最大应用地址：_____

当前可供应用程序使用的内存空间为：_____

当前计算机的实际内存大小为：_____

理论上每个 Windows 应用程序可以独占的最大存储空间是：_____

> **提示：**可供应用程序使用的内存空间实际上已经减去了开头与结尾两个 64KB 的保护区。虚拟内存空间中的 64KB 保护区是防止编程错误的一种 Windows 方式。任何对内存中这一区域的访问 (读、写、执行) 都将引发一个错误陷阱，从而导致错误并终止程序的执行。

按 committed、reserved、free 等三种虚拟地址空间分别记录实验数据。其中“描述”是指对该组数据的简单描述，例如，对下列一组数据：

00010000 – 00012000　　<8.00KB>　Committed,　READWRITE,　Private

可描述为：具有 READWRITE 权限的已调配私有内存区。

将系统当前的自由区 (free) 虚拟地址空间按表 5-6 格式记录。

表 5-6 实验记录

地址	大小	虚拟地址空间类型	访问权限	描述
		free		

> **提示：**详细记录实验数据在实验活动中是必要的，但想想是否可以简化记录的办法？

将系统当前的已调配区 (committed) 虚拟地址空间按表 5-7 格式记录。

表 5-7　实验记录

地址	大小	虚拟地址空间类型	访问权限	描述
		committed		

将系统当前的保留区 (reserved) 虚拟地址空间按表 5-8 格式记录。

表 5-8　实验记录

地址	大小	虚拟地址空间类型	访问权限	描述
		reserved		

4、 实验结论

简单描述 windows 进程的虚拟内存管理方案。

5、 程序清单

清单 5-1　了解和检测进程的虚拟内存空间

```
// 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,                       // 相关的进程
            pBlock,                         // 开始位置
            &mbi,                           // 缓冲区
            sizeof(mbi))==sizeof(mbi) )     // 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,        // 实际虚拟内存的模块句柄
                szFilename,              //完全指定的文件名称
                MAX_PATH)>0)             //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下个块
            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) ;
```

```
    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());
}
```

实验六 磁盘调度

1、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。

2、实验基本知识及原理

（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。

3、实验内容

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）。参考代码如下：

```
#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);
}
// 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;
}
```

实验七 进程间通信

1、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、共享存储区机制及信息量机制。

2、实验题目

1）消息的创建，发送和接收。

（1） 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。

（2） 观察上面程序，说明控制消息队列系统调用 msgctl()在此起什么作用？

2）共享存储区的创建、附接和断接。

使用系统调用 shmget()，shmat()，sgmdt()，shmctl()，编制一个与上述功能相同的程序。

3）比较上述 1，2 两种消息通信机制中数据传输的时间。

3、背景知识

系统调用函数说明、参数值及定义

- fork( )

  创建一个新进程。

  int fork( )

  其中返回 int 取值意义如下：

  0：创建子进程，从子进程返回的 id 值

  大于 0：从父进程返回的子进程 id 值

  -1：创建失败

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。

下面对部分系统调用说明如下：

- msgget(key,flag)

  获得一个消息的描述符，该描述符指定一个消息队列以便用于其他系统调用。该函数使用头文件如下：

  # include <sys/types.h>

  #include <sys/ipc.h>

  #include <sys/msg.h>

  参数定义

  int msgget(key,flag)

  key_t key;

  int flag;

  语法格式：msgqid=msgget(key,flag)

  其中：

  msgqid 是该系统调用返回的描述符，失败则返回-1；

  flag 本身由操作允许权和控制命令值相“或”得到。

  如：IPC_CREAT | 0400　　是否该队列应被创建；

  　　IPC_EXCL | 0400　　是否该队列的创建映象是互斥的；等。

- msgsnd(id,msgp,size,flag)

  发送一消息。

  该函数使用头文件如下：

  #include <sys/types.h>

  #include <sys/ipc.h>

  #include <sys/msg.h>

  参数定义：

  int msgsnd(id,msgp,size,flag)

  int id,size,flag;

  struct msgbuf *msgp;

  其中：id 是返回消息队列的描述符；msgp 是指向用户存储区的一个构造体指针，size 批示由 msgp 指向的数据结构中字符数组的长度，即消息的长度。这个数组的最大值由 MSG_MAX 系统可调用参数来确定。flag 规定当核心用尽内部缓冲空间时应执行的动作；若在标志范围 flag 中未设置 IPC_NOWAIT 位，则当该消息队列中的字节数超过一最大值时，或系统范围的消息数超过某一最大值时，调用 msgsnd 进程睡眠。若是设置 IPC_NOWAIT ，则在此情况下，msgsnd 立即返回。

- msgrcv(id,msgp,size,flag)

  接受一消息。

  该函数调用使用头文件如下：

#include <sys/types.h>
#include <sys/ipc.h>
#include <sys/msg.h>
参数定义：
int msgrcv(id,msgp,size,flag)
int id,size,type,flag;
struct msgbuf *msgq;
struct msgbuf{ long mtype;char mtext[];};
语法格式：
count=msgrcv(id,msgp,size,type,flag)
其中：id 是消息描述符，msgp 是用来存放欲接收消息的拥护数据结构的地址；size 是 msgp 中数据数组的大小；type 是用户要读的消息类型：
type 为 0：接收该项队列的第一个消息；
type 为正：接收类型 type 的第一个消息；
type 为负：接收小于或等于 type 绝对值的最低类型的第一个消息。
flag 规定倘若该队列无消息，核心应当做什么事，如果此时设置了 IPC_NOWAIT 标志，则立即返回，若在 flag 中设置了 MSG_NOERROR，且所接收的消息大小大于 size ，核心截断所接收的消息。
count 是返回消息正文的字节数。

- msgctl(id,cmd,buf)

查询一个消息描述符的状态，设置它的状态及删除一个消息描述符。
调用该函数使用头文件如下：
#include <sys/types.h>
#include <sys/ipc.h>
#include <sys/msg.h>
参数定义：
int msgctl(id,cmd,buf)
int id,cmd;
struct msqid_ds *buf;
其中：函数调用成功时返回 0，调用不成功时返回-1。
id 用来识别该消息的描述符；cmd 规定命令的类型。
IPC_STAT 将与 id 相关联的消息队列首标读入 buf。
IPC_SET 为这个消息序列设置有效的用户和小组标识及操作允许权和字节的数量。
IPC_RMID 删除 id 的消息队列。
buf 是含有控制参数或查询结果的用户数据结构的地址。
附：msgid_ds 结构定义如下：

```
struct msgid_ds
{struct ipc_perm msg_perm;      /* 许可权结构 */
short pad1[7];                  /* 由系统调用 */
ushort onsg_qnum;               /* 队列上消息数 */
ushort msg_qbytes;              /* 队列上最大字节数 */
ushort msg_lspid;               /* 最后发送消息的 PID */
ushort msg_lrpid;               /* 最后接收消息的 PID */
time_t msg_stime;               /* 最后发送消息的时间 */
time_t msg_rtime;               /* 最后接收消息的时间 */
time_t msg_ctime;               /* 最后更改时间 */
```

```
}
struct ipc_perm
{ushort uid;                    /*  当前用户 id */
ushort gid;                     /*  当前进程组 id */
ushort cuid;                    /*  创建用户 id */
ushort cgid;                    /*  创建进程组 id */
ushort mode;                    /*  存取许可权  */
{short pad1;long pad2}  /*  由系统调用  */
};
```

- shmget(key,size,flag)

  获取一个共享存储区。

  该函数使用头文件如下：

  #include <sys/types.h>

  #include <sys/ipc.h>

  #include <sys/shm.h>

  语法格式：

  shmid=shmget(key,size,flag)

  参数定义：

  int shmget(key,size,flag)

  key_t key;

  int size,flag;

  其中：size 是存储区的字节数，key 和 flag 与系统调用 msgget 中的参数含义相同。

  附：

操作允许权	八进制数
用户可读	00400
用户可写	00200
小组可读	00040
小组可写	00040
其他可读	00004
其他可写	00002

控制命令	值
IPC_CREAT	0001000
IPC_EXCL	0002000

  如：shmid=shmget(key,size,(IPC_CREAT | 0400);

  创建一个关键字为 key,长度为 size 的共享存储区。

- shmat(id,addr,flag)

  从逻辑上将一个共享存储区附接到进程的虚拟地址空间上。

  该函数使用头文件如下：

  #include <sys/types.h>

  #include <sys/ipc.h>

  #include <sys/msg.h>

  语法格式：

  virtaddr=shmat(id,addr,flag)

  参数定义：

  char *shmat(id,addr,flag)

char *addr;
int id,flag;
其中：id 是共享存储区的标识符，addr 是用户要使用共享存储区附接的虚地址，若 addr 是 0，系统选择一个适当的地址来附接该共享区。flag 规定对此区的读写权限，以及系统是否应对用户规定的地址做舍入操作。如果 flag 中设置了 shm_rnd 即表示操作系统在必要时舍去这个地址。如果设置了 shm_rdonly，即表示只允许读操作。viraddr 是附接的虚地址。

- shmdt(addr)

把一个共享存储区从指定进程的虚地址空间断开。
该函数使用头文件如下：
#include <sys/types.h>
#include <sys/ipc.h>
#include <sys/mhm.h>
参数定义：
int shmdt(addr)
char *addr
其中：当调用成功时，返回 0 值，调用不成功，返回-1，addr 是系统调用 shmat 所返回的地址。

- shmctl(id,cmd,buf)

对与共享存储区关联的各种参数进行操作，从而对共享存储区进行控制。
该函数使用头文件如下：
#include <sys/types.h>
#include <sys/ipc.h>
#include <sys/shm.h>
参数定义：
int shmctl(id,cmd,buf)
int id,cmd;
struct shmid_ds *buf;
其中：调用成功时返回 0，否则返回-1。id 为被共享存储区的标识符。cmd 规定操作的类型。规定如下：
IPC_STAT：返回包含在指定的 shmid 相关数据结构中的状态信息，并且把它放置在用户存储区中的*buf 指针所指的数据结构中。执行此命令的进程必须读取允许权。
IPC_SET：对于指定的 shmid，为它设置有效用户和小组标识和操作存取权。
IPC_RMID：删除指定的 shmid 以及与它相关的共享存储区的数据结构。
SHM_LOCK：在内存中锁定指定的共享存储区，必须是超级用户才可以进行此项操作。
buf 是一个用户级数据结构地址。
附： shmid_ds

```
{struct ipc_perm shm_perm;        /* 许可权结构 */
int shm_segsz;                    /* 段大小 */
int pad1;                         /* 由系统使用 */
ushort shm_lpid;                  /* 最后操作的进程 id; */
ushort shm_cpid;                  /* 创建者的进程 id; */
ushort shm_nattch;                /* 当前附界数 */
short pad2;                       /* 由系统使用 */
time_t shm_atime;                 /* 最后断接时间 */
time_t shm_dtime;                 /* 最后修改时间 */
time_t shm_ctime;                 /* 最后修改时间 */
```

}

4、实验内容与步骤

1）任务 1（消息的创建、发送和接收）的程序设计

（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。

2）任务 2（共享存储区的创建、附接和断接）的程序设计

（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，并将第一个字节置为-1，作为数据空的标志，等待其他进程发来的消息。当该字节的值发生变化时，表示收到了信息，进行处理。然后再次把它的值设为-1。当遇到的值为 0，则视为结束信号，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，当共享取得第一个字节为-1 时，SERVER 端空闲，可发送请求。CLIENT 随即填入 9 到 0。期间等待 SERVER 端的再次空闲。进行完这些操作后，CLIENT 退出。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）任务 3（基于信号量机制的进程同步问题）的程序设计（此任务可以选做）

使用系统调用 semget（），semop（），及 semctl（）编制一个哲学家进餐问题的程序。思路：为了便于操作和观察结果，用主程序创建一个信号量集(五个筷子信号量)，然后先后 fork（）五个哲学家子进程，使它们通过对信号量进行 P、V 操作并发地进行 thinking 与 eating.

5、任务 1 的参考的程序清单

```
#include <stdio.h>
#include <sys/types.h>
#include <sys/msg.h>
#include <sys/ipc.h>
#define MSGKEY 75
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);
  wait(0);
}
```

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。

6、实验结果与分析

7、实验总结

实验八 简单二级文件系统设计

1、实验内容

为 Linux/Unix 设计一个简单的二级文件系统，要求做到以下几点：

（1）可以实现下列几条命令（至少 4 条）

Login	用户登录
Dir	列文件目录
Create	创建文件
Delete	删除文件
Open	打开文件
Close	关闭文件
Read	读文件
Write	写文件

（2）列目录时要列出文件名、物理地址、保护码和文件长度

（3）源文件可以进行读写保护

2、实验准备

(1) 首先应确定文件系统的数据结构：主目录、子目录及活动文件等。主目录和子目录都以文件的形式存放于磁盘，这样便于查找和修改；

(2) 用户创建的文件，可以编号存储于磁盘上。如 file0,file1,file2……并以编号作为物理地址，在目录中进行登记。

3、实验指导

一、文件管理

要将文件存储在磁盘（带）上，必须为之分配相应的存储空间，这就涉及到对文件存储空间的管理；采取何种方式存储，又涉及到文件的物理结构；为了简化对文件的访问和共享，还应设置相应的用户文件描述表及文件表。

1、文件存储空间的管理

（1）文件卷的组织

UNIX 中，把每个磁盘（带）看作是一个文件卷，每个文件卷上可存放一个具有独立目录结构的文件系统。一个文件卷包含许多物理块，并按块号排列如下图：

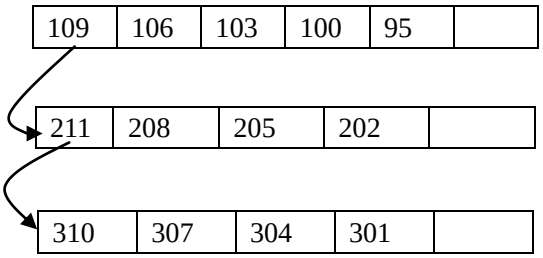
0#	1#	2#	3# ……K#	K+1#	……N#

其中，0#块用于系统引导或空闲，1#为超级块(superblock)，存放文件卷的资源管理信息，如整个文件卷的盘块数、磁盘索引结点的盘块数、空闲盘块号栈及指针等。2#~K#存放磁盘索引结点。每个索引结点 64B，第 K+1#~N#存放文件数据。

（2）空闲盘块的组织

UNIX 采用成组链接法组织空闲盘块。它将若干个空闲盘块划归一个组，将每组中所有盘块号存放在其前一组的第一个空闲盘块中，而第一组中所有空闲盘块号放入超级块的空闲盘块号栈中。

例：　　超级块表



（3）空闲盘块的分配与回收

内核要从文件系统中分配一盘块时，先检查超级块空闲盘块号栈是否已上锁。是则调用 sleep 睡眠，否则将超级块中空闲盘块栈栈顶盘块号分配出去。

回收时，若空闲盘块号栈未满，直接将回收盘块编号记入空闲盘块号栈中。若回收时栈已满，须先将栈中的所有空闲盘块号复制到新回收的盘块中，再将新回收盘块的编号作为新栈的栈底块号进栈。

2、文件的物理结构

UNIX 未采用传统的三种文件结构形式，而是将文件所占用盘块的盘块号，直接或间接地存放在该文件索引结点的地址项中。查找文件时，只需找到该文件的索引结点，便可直接或间接的寻址方式获得指定文件的盘块。

过程 bmap 可将逻辑文件的字节偏移量转换为文件的物理块号。先将字节偏移量转换为文件逻辑块号及块内偏移量，再把逻辑块号转换为文件的物理块号。

3、用户文件描述符表和文件表的管理

每个进程的 U 区中设置一张用户文件描述符表。只在首次打开文件时才需给出路径名。内核在该进程的用户文件描述符表中，分配一空项，取其在该表中的位移量作为文件描述符 fd(file discriptor)返还给用户。当用户再次访问该文件时，只需提供 fd，系统根据 fd 便可找到相应文件的内存索引结点。fd 表项的分配由 ufalloc 完成。

为了方便用户对文件进行读/写及共享，系统中设置了一张文件表。每个用户在打开文件时，都要在文件表中获得一表项，其中包含下述内容：

f.flag：文件标志，指示该文件打开是为了读或写；

f.inode：指向打开文件的内存索引结点指针；

f.offset：文件读写指针偏移值；

f.count：文件引用计数。

二、目录管理

UNIX 中，为了加速对文件目录的查找，将文件名和文件说明分开，由文件说明形成一个称为索引结点的数据结构，而相应的文件目录项则只由文件符号名和指向索引结点的指针构成。

对目录的管理应包括的功能有：

1、对索引结点的管理

每个文件都有一唯一的磁盘索引结点(di_node)。文件被打开后，还有一个内存索引结点(i_node)。创建一新文件时，就为之建立一个磁盘索引结点，以将文件的有关信息记入其中，并将用户提供的文件名和磁盘索引结点号一并组成一个新目录项，记入其父目录文件中。文件被撤消时，系统要回收该文件的磁盘索引结点，从其父目录中删除该目录项。随着文件的打开与关闭，系统还要为之分配和回收内存索引结点。

（1）磁盘索引结点

磁盘索引结点中，包含有关文件的下述一系列信息：

文件模式 di_mode。可以是正规文件、目录文件、字符特别文件、块特别文件和管道文件等几种；

文件所有者用户标识符 di_uid。指拥有该文件的用户标识符；

同组用户标识符 di_gid。与拥有该文件的用户在同一小组的用户标识符；

文件长度 di_size。以字节计数的文件大小；

文件的联接计数 di_nlink。表明在本文件系统中所有指向该文件的文件名计数；

文件的物理地址 di_addr。di_addr 地址项共有 13 项，即 di_addr(0) 到 di_addr(12)，每个地址项占 3 字节；

文件的访问时间 di_atime。指文件最近被进程访问的时间；

文件的修改时间 di_mtime。指文件和索引结点最近被进程修改的时间；

文件的建立时间 di_citime。

（2）内存索引结点

文件被打开后，系统为它在内存索引结点表区中建一内存索引结点，以方便用户和系统对文件的访问。其中，一部分信息是直接从磁盘索引结点拷贝过来的，如 i_mode、i_uid、i_gid、i_size、i_addr、i_nlink 等，并又增加了如下各信息：

索引结点编号 i_number。作为内存索引结点的标识符；

状态 i_flag。指示内存索引结点是否已上锁、是否有进程等待此 i 结点解锁、i 结点是否被修改、是否有最近被访问等标志；

引用计数 i_count。记录当前有几个进程正在访问此 i 结点，每当有进程访问此 i 结点时，对 i_count+1，退出-1；

设备号 i_dev。文件所属文 S 件系统的逻辑设备号；

前向指针 i_forw。Hash 队列的前向指针；

后向指针 i_back。Hash 队列的后向指针；

（3）磁盘索引结点的分配与回收

分配过程 ialloc：当内核创建一新文件时，要为之分配一空闲磁盘 i 结点。如分配成功，便再分配一内存 i 结点。其过程如下：

检查超级块上锁否。由于超级块是临界资源，诸进程必须互斥地访问它，故在进入 ialloc 后，要先检查它是否已上锁，若是则睡眠等待；

检查 i 结点栈空否。若 i 结点栈中已无空闲结点编号，则应从盘中再调入一批 i 结点号进栈。若盘中已无空闲 i 结点，则出错处理，返回；

从空闲 i 结点编号栈中分配一 i 结点，并对它初始化、填写有关文件的属性；

分配内存 i 结点；

将磁盘 i 结点总数-1，置超级块修改标志，返回。

回收过程 ifree：

当删除文件时，应回收其所占用的盘块及相应的磁盘 i 结点。具体有：

检查超级块上锁否。若是，直接返回，即不把本次回收的 i 结点号记入空闲 i 结点编号栈中；

检查 i 结点编号栈满否。若已满，无法再装入新回收的 i 结点号，立即返回，若未满，便将回收的 i 结点编号进栈，并使当前空闲结点数+1；

置超级块修改标志，返回。

（4）内存索引结点的分配与回收

分配过程 iget：

虽然 iget 用在打开文件时为之分配 i 结点，但由于允许文件被共享，因此，如果一文件已被其他用户打开并有了内存 i 结点，则此时只需将 i 结点中的引用计数+1。如果文件尚未被任何用户（进程）打开，则由 iget 过程为该文件分配一内存 i 结点，并调用 bread 过程将其磁盘 i 结点的内容拷贝到内存 i 结点中并进行初始化。

回收过程 iput：

进程要关闭某文件时，须调用 iput 过程，先对该文件内存 i 结点中的引用计数-1。若结果为 0，便回收该内存 i 结点，再对该文件的磁盘 i 结点中的连接计数减 1，若其结果也为 0，便删除此文件，并回收分配给该文件的盘块和磁盘 i 结点。

2、构造目录 make_node

文件系统的一个基本功能是实现按名存取，它通过文件目录来实现。为此须使每一个文件都在文件目录中有一个目录项，通过查找文件目录可找到该文件的目录项和它的索引结点，进而找到文件的物理位置。对于可供多个用户共享的文件，则可能有多个目录项。如果要将文件删除，其目录项也应删除。

构造目录先调用 ialloc 为新建文件分配一磁盘 i 结点及内存 i 结点。若分配失败则返回，分配成功时须先设置内存 i 结点的初值（含拷贝），调用写目录过程 wdir，将用户提供的文件名与分配给该文件的磁盘 i 结点号一起，构成一新目录项，再将它记入其父目录文件中。

3、检索目录 namei

用户在第一次访问某文件时，需要使用文件的路径名，系统按路径名去检索文件目录，得到该文件的磁盘索引结点，且返回给用户一个 fd。以后用户便可利用该 fd 来访问文件，这时系统不需再去检索文件目录。

namei 根据用户给出的路径名，从高层到低层顺序地查找各级目录，寻找指定文件的索引结点号。检索时，对以'/'开头的路径名，须从根目录开始检索，否则，从当前目录开始，并把与之对应的 i 结点作为工作索引结点，然后用文件路径名中的第一分量名与根或与当前目录文件中的各目录项的文件名，逐一进行比较。由于一个目录文件可能占用多个盘块，在检索完一个盘块中所有目录项而未找到匹配的文件分量名时，须调用 bmap 和 bread 过程，将下一个盘块中的所有目录项读出后，再逐一检索。若检索完该目录文件的所有盘块而仍未找到，才认为无此文件分量名。

检索方式采用 Hash 方法。

三、主要文件操作的处理过程

1、打开文件 open

检索目录。内核调用 namei 从根目录或从当前目录，沿目录树查找指定的索引结点。若未找到或该文件不允许存取，则出错处理返回 NULL，否则转入下一步；

分配内存索引结点。如果该文件已被其它用户打开，只需对上一步中所找到的 i 结点引用计数+1，否则应为被打开文件分配一内存 i 结点，并调用磁盘读过程将磁盘 i 结点的内容拷贝到内存 i 结点中，并设置 i.count=1；

分配文件表项。为已打开的文件分配一文件表项，使表项中的 f.inode 指向内存索引结点；

分配用户文件描述表项。

2、创建文件 creat

核心调用 namei，从根目录或当前目录开始，逐级向下查找指定的索引结点。此时有以下二种情况：

重写文件。namei 找到了指定 i 结点，调用 free 释放原有文件的磁盘块。此时内核忽略用户指定的许可权方式和所有者，而保持原有文件的存取权限方式和文件主。最后打开。

新建。namei 未找到。调用 ialloc，为新创建的文件分配一磁盘索引结点，并将新文件名及所分配到的 i 结点编号，写入其父目录中，建立一新目录项。利用与 open 相同的方式，把新文件打开。

3、关闭文件 close

根据用户文件描述符 fd，从相应的用户文件描述符表项中，获得指向文件表项的指针 fp，再对该文件表项中的 f.count-1。

四、主要数据结构

1、i 节点

```
struct  inode
{  struct  inode  *i_forw;
```

```
    struct   inode   *i_back;
    char   i_flag;
    unsigned   int   i_ino;                  /*磁盘 i 节点标号*/
    unsigned   int   i_count;                /*引用计数*/
    unsigned   short   di_number;            /*关联文件数，当为 0 时，则删除该文件*/
    unsigned   short   di_mode;              /*存取权限*/
    unsigned   short   di_uid;               /*磁盘 i 节点用户 id*/
    unsigned   short   di_gid;               /*磁盘 i 节点组 id*/
    unsigned   int   di_addr[NADDR];         /*物理块号*/
```

2、磁盘 i 节点

```
struct dinode
{   unsigned   short   di_number;          /*关联文件数*/
    unsigned   short   di_mode;            /*存取权限*/
    unsigned   short   di_uid
    unsigned   short   di_gid;
    unsigned   long    di_size;            /*文件大小*/
    unsigned   int     di_addr[NADDR];     /*物理块号*/
}
```

3、目录项结构

```
struct direct
{   char   d_name[DIRSIZ];               /*目录名*/
    unsigned   int   d_ino;              /*目录号*/
}
```

4、超级块

```
struct filsys
{   unsigned short   s_isize;                /*i 节点块块数*/
    unsigned long    s_fsize;                /*数据块块数*/
    unsigned int     s_nfree;                /*空闲块块数*/
    unsigned short   s_pfree;                /*空闲块指针*/
    unsigned int     s_free[NICFREE];        /*空闲块堆栈*/
    unsigned int     s_ninode;               /*空闲 i 节点数*/
    unsigned short   s_pinode;               /*空闲 i 节点指针*/
    unsigned int     s_inode[NICINOD];       /*空闲 i 节点数组*/
    unsigned int     s_rinode;               /*铭记 i 节点*/
    char   s_fmod;                           /*超级块修改标记*/
}
```

5、用户密码

```
struct   pwd
{   unsigned   short   p_uid;
    unsigned   short   p_gid;
    char   password[PWOSIZ];
};
```

6、目录

```
struct dir
{   struct   direct   direct [DIRNUM];
    int   size;
```

};

7、查找内存 i 节点的 hash 表

```
struct hinode
{     struct    inode    *i_forw;
};
```

8、系统打开表

```
struct file
{     char    f_flag;                    /*文件操作标志*/
      unsigned    int    f_count;        /*引用计数*/
      struct    inode    *f_inode;       /*指向内存 i 节点*/
      unsigned    long    f_off;         /*读/写指针*/
};
```

9、用户打开表

```
struct user
{        unsigned short u_default_mode;
         unsigned short u_uid;                  /*用户标志*/
         unsigned short u_gid;                  /*用户组标志*/
         unsigned short u_ofile[NOFILE];        /*用户打开表*/
};
```

三、主要函数

1、i 节点内容获取函数 iget( )
2、i 节点内容释放函数 iput( )
3、目录创建函数 mkdir( )
4、目录搜索函数 namei( )
5、磁盘块分配函数 balloc( )
6、磁盘块释放函数 bfree( )
7、分配 i 节点区函数 ialloc( )
8、释放 i 节点区函数 ifree( )
9、搜索当前目录下文件的函数 iname( )
10、访问控制函数 access( )
11、显示目录和文件用函数_dir( )
12、改变当前目录用函数 chdir( )
13、打开文件函数 open( )
14、创建文件函数 create( )
15、读文件用函数 read( )
16、写文件用函数 write( )
17、用户登录函数 login( )
18、用户退出函数 logout( )
19、文件系统格式化函数 format( )
20、进入文件系统函数 install( )
21、关闭文件系统函数 close( )
22、退出文件系统函数 halt( )
23、文件删除函数 delete( )

四、主程序说明

```
begin
   step1          对磁盘进行格式化
```

step2　　调用 install( ),进入文件系统
step3　　调用_dir( ),显示当前目录
step4　　调用 login( ),用户注册
step5　　调用 mkdir( )和 chdir( )创建目录
step6　　调用 creat( )，创建文件 0
step7　　分配缓冲区
step8　　写文件 0
step9　　关闭文件 0 和释放缓冲
step10　　调用 mkdir( )和 chdir( )创建子目录
step11　　调用 creat( ),创建文件 1
step12　　分配缓冲区
step13　　写文件 1
step14　　关闭文件 1 和释放缓冲
step15　　调用 chdir 将当前目录移到上一级
step16　　调用 creat( ) ，创建文件 2
step17　　分配缓冲区
step18　　调用 write( ),写文件 2
step19　　关闭文件 2 和释放缓冲
step20　　调用 delete( ) ,删除文件 0
step21　　调用 creat( ) ，创建文件 3
step22　　为文件 3 分配缓冲区
step23　　调用 write( ),写文件 3
step24　　关闭文件 3 和释放缓冲
step25　　调用 open( ),打开文件 2
step26　　为文件 2 分配缓冲区
step27　　调用 open( ),打开文件 2
step28　　释放缓冲
step29　　用户退出(logout)
step30　　关闭(halt)

end

由上述描述过程可知，该文件系统实际是为用户提供一个解释执行相关命令的环境。主程序中的大部分语句都被用来执行相应的命令。

下面，我们给出每个过程的相关 C 语言程序。读者也可以使用这些子过程，编写出一个用 shell 控制的文件系统界面。

五、参考程序

具体参见 http://www.doc88.com/p-918959409953.html

附录 1：Linux 编程基础

1、实验目的

要求学生熟悉 Linux 系统，熟练使用 vi 编辑器或 gedit 编辑器，掌握如何在 Linux 下编写并编译、调试、运行程序。

2、实验内容

（1）学习使用 vi 编辑器，作为替代，也可以选择学习使用 gedit 编辑器。

（2）使用 c 语言编写一个显示输入文件内容的程序，要求显示时删除输入文件中多余的空行。

3、实验步骤

（1）输入以下 c 程序

输入 c 程序可以用 vi 命令来完成。下面就使用 vi 命令来输入 c 程序 cowpact.c。vi 或 gedit 的使用请上网查询相关的指南。

```
/*    Remove newlines */
#include <stdio.h>
main()
{
   int c,n=0,max=1;
   while((c=getchar())!=EOF)
   {
       if (c==’\n’)
          n++;
       else
          n=0;
       if (n<=max)
          putchar( c);
    }
}
```

上述程序是用来删除空行的，连续两个以上的空行只保留一个空行。

（2）编译 c 程序

由于 c 语言是一种高级语言，所以输入完 c 程序后就要对它进行编译。cc 命令可以用来编译 c 程序。如果在 cc 命令后面直接跟上文件名，则编译后的输出结果将存放在标准的 a.out 文件中。如果 cc 命令使用-o 任选项，则可以将编译结果存放在自己命名的文件中。为方便起见，我们使用带-o 任选项的 cc 命令来进行编译。当然，在本次课程设计中也可以使用 gcc 命令进行编译。

```
$cc –o compact compact.c
```

这样，编译结果就存放在 compact 文件中。如果出现编译错误，则需要使用 GDB（参见下页的参考资料）进行调试并找出错误，然后再利用 vi 命令来对程序进行修改。

(3) 执行 c 程序

由于存放编译如果的文件本身就是可执行文件，所以可以在 shell 提示符敲入该文件的名字来执行它，在有些情况下还要提供输入内容。下面给出 compact 的执行过程。

首先创建测试文件 testfile。

```
$cat > testfile
this is line 1
this is line 2
this is line 4
<ctrl+d>     （这里表示的是在键盘上同时按下 ctrl 键和 d，终止输入）
$
```

然后，对 testfile 文件使用 compact。

```
$./compact < testfile
  this is line 1
  this is line 2
  this is line 4
$
```

这样就从输入文件中删除了多余的空行。

参考资料：

[1] Linux 操作系统下 C 语言编程快速入门， http://www.doc88.com/p-43941787254.html

[2] 程序调试工具 GDB 用法，http://www.cnblogs.com/itech/archive/2011/02/12/1952888.html

附录 2：课程设计报告参考格式

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

学　　号：_____

姓　　名：_____

班　　级：_____

指导老师：_____

完成时间：_____

一、实验题目

二、实验目的

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）

四、详细设计（含主要的数据结构、程序流程图、关键代码等）

五、实验结果与分析

六、小结与心得体会