

湖南科技大学计算机科学与工程学院

操作系统 课程设计报告

专业班级： 计算机科学与技术专业五班

姓 名： 李龙华

学 号： 2105010614

指导教师： 刘敏

时 间： 2023-06-22

地 点： 逸夫楼 431

指导教师评语：

成绩：

等级：

签名： _____
年 月 日

实验一、Windows 进程管理

- 1、实验题目
- 2、实验目的
- 3、实验内容
 - 3-1、编写基本的 Win32 Consol Application
 - 3-2、创建进程
 - 3-3、父子进程的简单通信及终止进程
- 4、小结与心得体会

实验二、Linux 进程控制

- 1、实验题目
- 2、实验目的
- 3、实验内容
 - 3-1、进程的创建
 - 3-2、子进程执行新任务
- 4、心得与体会

实验三、Linux 进程间通信

- 1、实验题目
- 2、实验目的
- 3、实验内容
- 4、心得与体会

实验四、Windows 线程的互斥与同步

- 1、实验题目
- 2、实验目的
- 3、实验内容
- 4、心得与体会

实验五、内存管理

- 1、实验题目
- 2、实验目的
- 3、实验内容
- 4、心得体会

实验六、银行家算法的模拟与实现

- 1、实验题目
- 2、实验目的
- 3、背景知识
- 4、模块设计
- 5、详细设计
 - 5.1、数据结构
 - 5.2、流程图
 - 5.3、算法思路
 - 5.3.1、安全性检查算法:
 - 5.3.2、主程序:
- 6、实验结果分析
- 7、小结与心得体会

实验七、磁盘调度算法的模拟与实现

- 1、实验题目
- 2、实验目的
- 3、背景知识
- 4、模块设计
- 5、详细设计
 - 5.1、数据结构
 - 5.2、流程图
 - 5.3、算法思路
 - 5.3.1、先进先出算法

5.3.1、最短服务时间优先算法

5.3.1、SCAN（电梯算法）

5.3.1、C-SCAN（循环扫描）算法

6、实验结果与分析

7、小结与心得体会

实验八、虚拟内存系统的页面置换算法模拟

1、实验题目

2、实验目的

3、背景知识

4、模块设计

5、详细设计

5.1、数据结构

5.2、流程图

5.3、算法思路

5.3.1、指令序列的设计

6、实验结果与分析

7、小结与心得体会

实验九、基于信号量机制的并发程序设计

1、实验题目

2、实验目的

3、背景知识

4、模块设计

5、详细设计

5.1、数据结构

5.2、流程图

5.3、算法思路

6、实验结果与分析

7、小结与心得体会

实验十、实现一个简单的 shell 命令行解释器

1、实验题目

2、实验目的

3、背景知识

4、模块设计

5、详细设计

5.1、数据结构

5.2、流程图

5.3、算法思路

6、实验结果与分析

7、小结与心得体会

实验十一、简单二级文件系统设计

1、实验题目

2、实验目的

3、背景知识

4、模块设计

5、详细设计

5.1、数据结构

5.2、流程图

5.3、算法思路

6、实验结果与分析

7、小结与心得体会

实验一、Windows 进程管理

1、实验题目

Windows 进程管理

2、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。

3、实验内容

由于我平常用的都是Dev-C++来开发C/C++程序的，对于VC没怎么接触过，因此对于这次课设的验证性实验和设计类我都是用的Dev-C++来完成的。

3-1、编写基本的 Win32 Consol Application

这个实验的主要目的是为了熟悉编写C/C++控制台应用程序，对于创建一个C/C++控制台应用程序，我一般有俩种方式。

- 第一种，进入Dev-C++主界面，点击文件->新建->源代码。
- 第二种方式，点击文件->新建->项目->选择控制台应用程序。

前面是运行该程序所花费的时间，后面是程序的返回值，如果返回值为零，说明该程序运行正常，否则表示程序异常，或者非法访问等异常错误。

3-2、创建进程

- 用来创建进程的API函数CreateProcess()

```
BOOL bCreateOK=::CreateProcess(  
    NULL,                //子进程名称  
    szCmdLine,           //命令行参数  
    NULL,                // 不继承父进程的进程句柄  
    NULL,                // 不进程父进程的线程句柄  
    FALSE,               // 句柄继承标识为FALSE  
    CREATE_NEW_CONSOLE,  // 使用新的控制台  
    NULL,                // 使用父进程的环境  
    NULL,                // 使用父进程的当前目录  
    &si,                 // 启动信息  
    &pi);                // 返回的进程信息
```

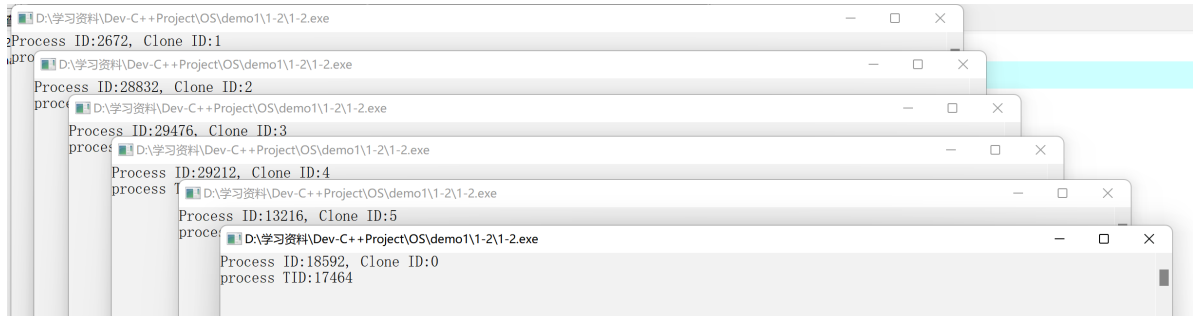
- 用来获得当前进程标识符的API函数GetCurrentProcessId()
- 用来获得当前进程已加载的模块的API函数GetModuleFileName()

```
// 提取用于当前可执行文件的文件名
TCHAR szFilename[MAX_PATH] ;
GetModuleFileName(NULL, szFilename, MAX_PATH) ;
```

- 进程终止的API函数ExitProcess() 或TerminateProcess()

拷贝实验给出的源代码到自己新建的工程下

1. 不修改的运行结果：会生成五个子进程



在“命令提示符”窗口加入参数重新运行生成的可执行文件，会生成相应个数的子进程，如果输入的命令行参数大于1的话，那么所克隆出来的子进程为(5-命令行参数)



2. 第一次修改nClone=0结果和源代码一样。
3. 第二次修改的运行结果会无限生成子进程，不管输入的命令行参数等于多少。

```
if (argc > 1)
{
    // 从第二个参数中提取克隆ID
    :: sscanf(argv[1] , "%d" , &nClone) ; //用sscanf_s代替
}
```

nClone=0

原因在于：上述nclone语句是在if语句后面，导致父进程给子进程传递的参数不起作用，nclone一直为0。

3-3、父子进程的简单通信及终止进程

用于管理事件对象的 API

- CreateEvent()在内核中创建一个新的事件对象。

```
HANDLE hMutexSuicide=CreateMutex(  
    NULL,                // 缺省的安全性  
    TRUE,                // 最初拥有的  
    g_szMutexName) ;     // 互斥体名称
```

- OpenEvent()创建对已经存在的事件对象的引用。

```
HANDLE hMutexSuicide = OpenMutex(  
    SYNCHRONIZE,        // 打开用于同步  
    FALSE,              // 不需要向下传递  
    g_szMutexName) ;    // 名称
```

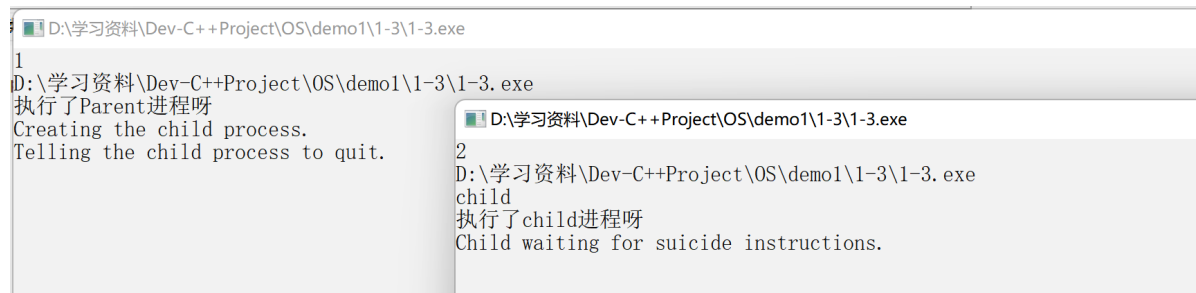
运行结果显示，不断克隆出子进程，这是因为程序路径名与命令行的参数没有空格分开，导致父进程给子进程传递的参数没有达到预期的效果。

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

改为：

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

运行结果显示：俩个进程，一个父进程，一个子进程，子进程通过接受父进程的信号而终止。



```
//父进程  
ReleaseMutex(hMutexSuicide)
```

```
//子进程  
waitForSingleObject(hMutexSuicide, INFINITE)
```

- 第一次修改，不断克隆出子进程。
- 第二次修改，子进程立即结束。

```
waitForSingleObject(hMutexSuicide, 0)
```

由于第二个参数为0，所以子进程并不会因为等待父进程的信号而阻塞，而是继续往下执行。

4、小结与心得体会

通过此时实验了解基本的控制台程序的创建和控制台程序传参的过程，通过简单的修改nClone的值，实现完全不同的效果，以及在Windows下如何创建进程，进程之间如何相互通信，如何实现互斥等操作。

实验二、Linux 进程控制

1、实验题目

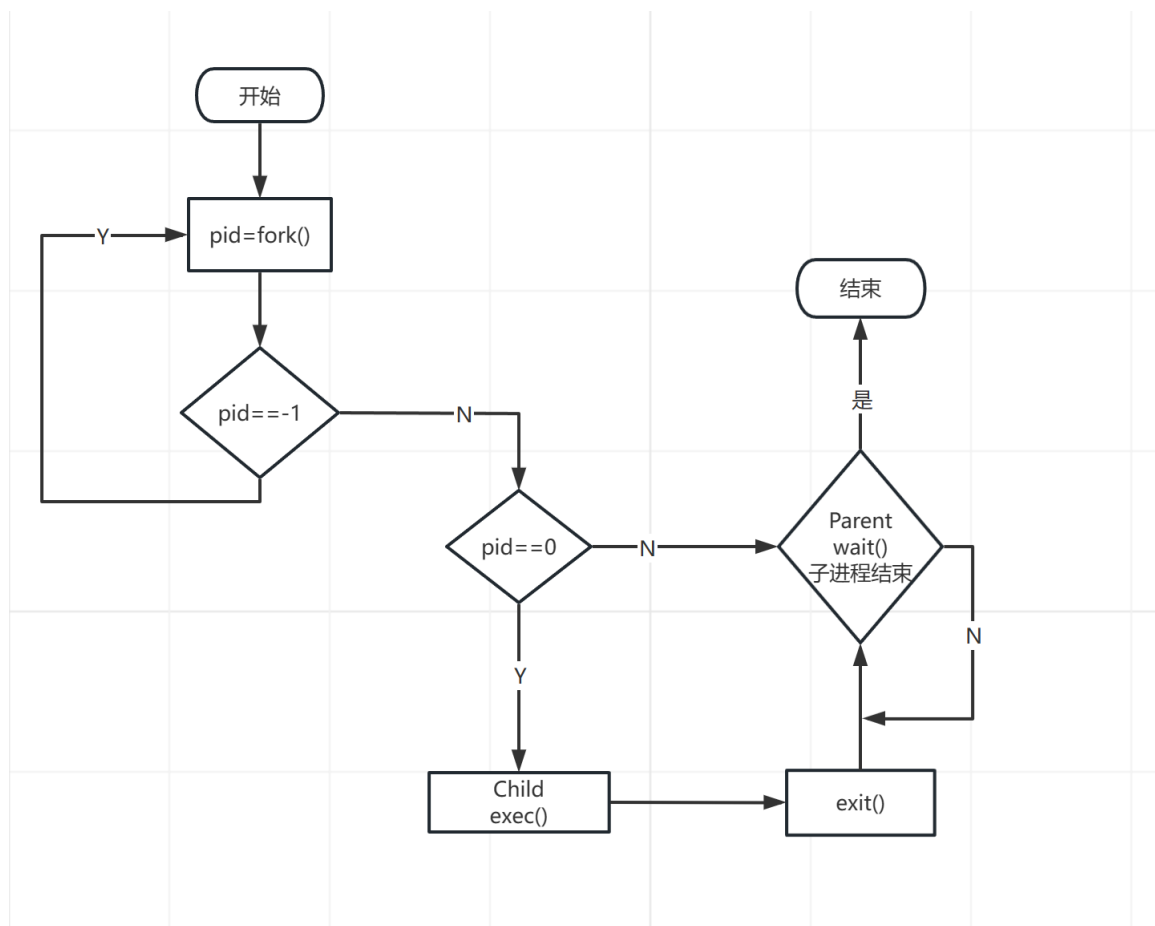
Linux 进程控制

2、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。

3、实验内容

流程图：



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。

3-1、进程的创建

运行结果如下

```
llh@llh-virtual-machine:~/demo/demo$ ./demo1
acbc llh@llh-virtual-machine:~/demo/demo$ ./demo1
bcac llh@llh-virtual-machine:~/demo/demo$ ./demo1
bcac llh@llh-virtual-machine:~/demo/demo$ ./demo1
acbc llh@llh-virtual-machine:~/demo/demo$ ./demo1
acbc llh@llh-virtual-machine:~/demo/demo$ S
```

通过fork()创建子进程，父进程打印出b，子进程打印a，最后各自都打印c。结果可以看出由于sleep的原因以及调度方法导致打印结果的顺序是不一样的。

```
sleep(rand() % 3);
```

3-2、子进程执行新任务

运行结果如下

```
llh@llh-virtual-machine:~/demo/demo$ ls
demo1  demo1.c
llh@llh-virtual-machine:~/demo/demo$ ./demo1
demo1  demo1.c
Child Complete
llh@llh-virtual-machine:~/demo/demo$
```

可以看出执行程序后，首先父进程fork()创建子进程，由于执行完fork()函数后父进程返回的值是大于0，而子进程返回的值为0，让子进程使用exec系列的函数去执行ls命令，而父进程则等待子进程而阻塞，直到子进程的结束，最后打印出子进程结束的字符串。

4、心得与体会

通过这次实验又熟悉了Linux环境的搭建以及编译命令和vim的使用，大致了解了Linux下如何创建进程以及创建进程的过程，如何通过exec系列的函数执行相应的程序。

实验三、Linux 进程间通信

1、实验题目

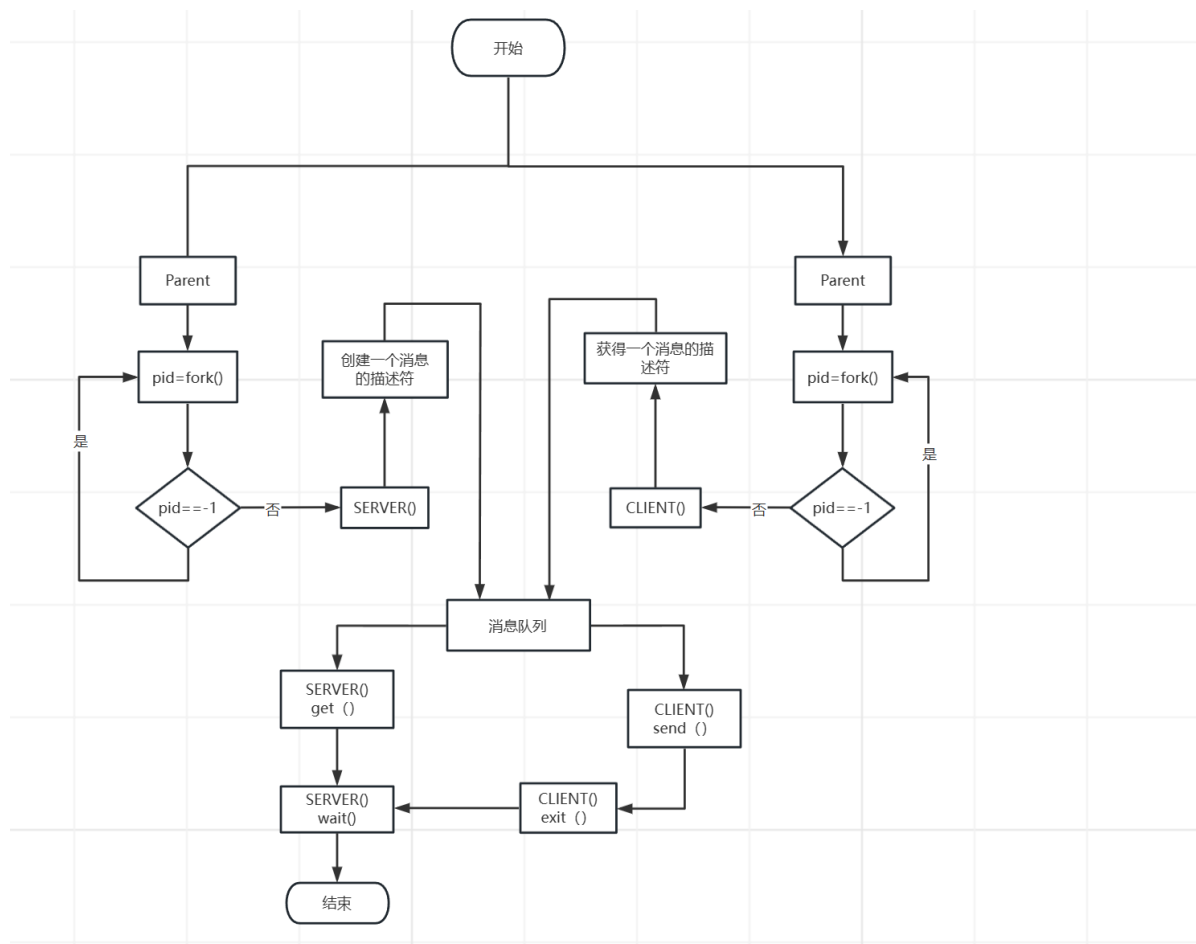
Linux 进程间通信

2、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。

3、实验内容

流程图：



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。

- 首先父进程`fork()`两个子进程`SERVER()`和`CLIENT()`
- `SERVER()`通过系统调用`msgget()`去创建消息队列并得到消息描述符，然后再通过系统调用`msgrcv()`等待消息队列里面的消息，最后通过系统调用`msgctl()`去删除消息描述符
- `CLIENT()`通过系统调用`msgget()`去得到对应消息队列的标识符，然后再通过系统调用`msgsnd()`发送消息到消息队列当中

<pre> llh@llh-virtual-machine:~/demo/demo\$./demo (client) sent (client) sent (client) sent (client) sent (client) sent (client) sent (client) sent (client) sent (client) sent (client) sent (client) sent (Server) recieved (Server) recieved (Server) recieved (Server) recieved (Server) recieved (Server) recieved (Server) recieved (Server) recieved (Server) recieved (Server) recieved llh@llh-virtual-machine:~/demo/demo\$ </pre>	<pre> llh@llh-virtual-machine:~/demo/demo\$./demo (client) sent (Server) recieved (client) sent (Server) recieved (client) sent (Server) recieved (client) sent (Server) recieved (client) sent (Server) recieved (client) sent (Server) recieved (client) sent (Server) recieved (client) sent (Server) recieved (client) sent (Server) recieved (client) sent (Server) recieved llh@llh-virtual-machine:~/demo/demo\$ </pre>
---	---

运行结果可以看出sent和recieved的打印次序并不是交替出现的，而是连续出现的，这是因为处理器通过时间片轮流执行进程，而执行相应的进程时，由于时间片还没到就会一直执行下去，因此我在打印后面加了一句代码 `sleep(1)`，当打印完之后该进程就会去休眠，因此就能实现交替。

4、心得与体会

通过本此实验熟悉了对Linux终端下命令的使用和加深了对进程的创建的理解，了解了Linux下进程之间的通信过程以及如何进程通信，通过使用系统调用 `msgget()`，`msgsnd()`，`msgrcv()`及 `msgctl()`来实现进程通信资源的分配。

实验四、Windows 线程的互斥与同步

1、实验题目

Windows 线程的互斥与同步

2、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。

3、实验内容

根据实验的要求，写出相关的生产者消费者伪代码

```
Semaphore:
empty=n;           //empty表示当前缓冲区空闲的个数
full=0;            //full表示当前缓冲区产品的个数
mutex;             //mutex用来对访问缓冲区实施互斥
//生产者
Producer:
while(1){
P(empty)
P(mutex)
生产产品
将产品放到缓冲区中
V(mutex)
V(full)
}
//消费者
Consumer:
while(1){
P(full)
P(mutex)
从缓冲区中拿产品
消费产品
V(mutex)
V(empty)
}
```

首先了解一下基本信号量对象和互斥对象、线程如何创建的以及加锁解锁操作。

1. 创建或打开命名或未命名的信号量对象

```
HANDLE CreateSemaphore(
    LPSECURITY_ATTRIBUTES lpSemaphoreAttributes, // SD
    LONG lInitialCount,           // initial count
    LONG lMaximumCount,          // maximum count
    LPCTSTR lpName                // object name
)
```

2. CreateMutex () 函数用来创建一个有名或无名的互斥量对象

```
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
)
```

3. 创建线程

```
HANDLE CreateThread(
    LPSECURITY_ATTRIBUTES lpThreadAttributes,
    SIZE_T dwStackSize,
    LPTHREAD_START_ROUTINE lpStartAddress,
    __drv_aliasesMem LPVOID lpParameter,
    DWORD dwCreationFlags,
    LPDWORD lpThreadId
)
```

```
//加锁
WaitForSingleObject(hMutex, INFINITE)
//解锁
BOOL ReleaseMutex(HANDLE hSemaphore)
BOOL ReleaseSemaphore(
    [in] HANDLE hSemaphore,
    [in] LONG lReleaseCount,
    [out, optional] LPLONG lpPreviousCount
)
```

了解了基本的知识之后接下来就是根据上述伪代码的逻辑具体的实现代码，实现也很简单，只需根据上述逻辑将上述的知识链接到里面即可。

运行结果：

- 信号量 EmptySemaphore 的初始化方法如果使得数量为零，则将导致全部进程都阻塞。
- 调整生产者线程和消费者线程的个数，当消费者数目大于生产者，消费者将经常等待生产者，反之，生产者经常等待消费者。

4、心得与体会

之前都是在Linux下实现生产者和消费者问题的，而现在是在Windows下实现，了解了Windows下如何创建进程和信号量和互斥量以及如何进行加锁和解锁操作，通过简单的修改，就会出现难以预料的结果，很有趣，学到很多知识。

实验五、内存管理

1、实验题目

内存管理

2、实验目的

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

3、实验内容

将实验代码拷贝到项目中运行，发现出现类型错误，将DWORD换成DWORD64即可。

- 虚拟内存每页容量为 4.00KB
- 最小应用地址为 0x00010000
- 最大应用地址为0x7fffffff
- 当前可供应用程序使用的内存空间为 3.99 GB
- 当前计算机的实际内存大小为 13.9GB
- 理论上每个Windows 应用程序可以独占的最大存储空间是 4GB

```
D:\学习资料\Dev-C++\Project\OS\demo\main.exe
Virtual memory page size: 4.00 KB
Minimum application address: 0x00010000
Maximum application address: 0x7fffffff
Total available virtual memory: 3.99 GB
00010000-00020000 (64.0 KB) Committed, READWRITE, Mapped
00020000-00023000 (12.0 KB) Committed, READONLY, Mapped
00023000-00030000 (52.0 KB) Free, NOACCESS
00030000-0004f000 (124 KB) Committed, READONLY, Mapped
0004f000-00050000 (4.00 KB) Free, NOACCESS
00050000-00054000 (16.0 KB) Committed, READONLY, Mapped
00054000-00060000 (48.0 KB) Free, NOACCESS
00060000-00062000 (8.00 KB) Committed, READWRITE, Private
00062000-00070000 (56.0 KB) Free, NOACCESS
00070000-000a1000 (196 KB) Committed, READONLY, Mapped
000a1000-000b0000 (60.0 KB) Free, NOACCESS
000b0000-000b3000 (12.0 KB) Committed, READONLY, Mapped
000b3000-000c0000 (52.0 KB) Free, NOACCESS
000c0000-000c1000 (4.00 KB) Committed, READONLY, Mapped
000c1000-000d0000 (60.0 KB) Free, NOACCESS
000d0000-000ee000 (120 KB) Committed, READWRITE, Private
000ee000-001d0000 (904 KB) Reserved, READONLY, Private
001d0000-001d1000 (4.00 KB) Committed, READONLY, Mapped
001d1000-001e0000 (60.0 KB) Free, NOACCESS
001e0000-001e1000 (4.00 KB) Committed, READONLY, Mapped
001e1000-001f0000 (60.0 KB) Free, NOACCESS
001f0000-001fc000 (48.0 KB) Committed, READONLY, Mapped
001fc000-00200000 (16.0 KB) Free, NOACCESS
00200000-003bb000 (1.73 MB) Reserved, READONLY, Private
003bb000-003c2000 (28.0 KB) Committed, READWRITE, Private
003c2000-00400000 (248 KB) Reserved, READONLY, Private
```

4、心得体会

对于经常用Windows的我来说，以前只知道用，却从来没有关心过程序内存的大小，经过这次实验了解了Windows下程序内存的空间的基本结构。

实验六、银行家算法的模拟与实现

1、实验题目

银行家算法的模拟与实现

2、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。

3、背景知识

(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。

4、模块设计

本次代码设计包含两个部分，第一个部分就是要检查进程所需要的资源是否超过最大值以及系统是否有足够的资源满足该进程的请求，第二个部分就是检查是否存在安全序列。

5、详细设计

5.1、数据结构

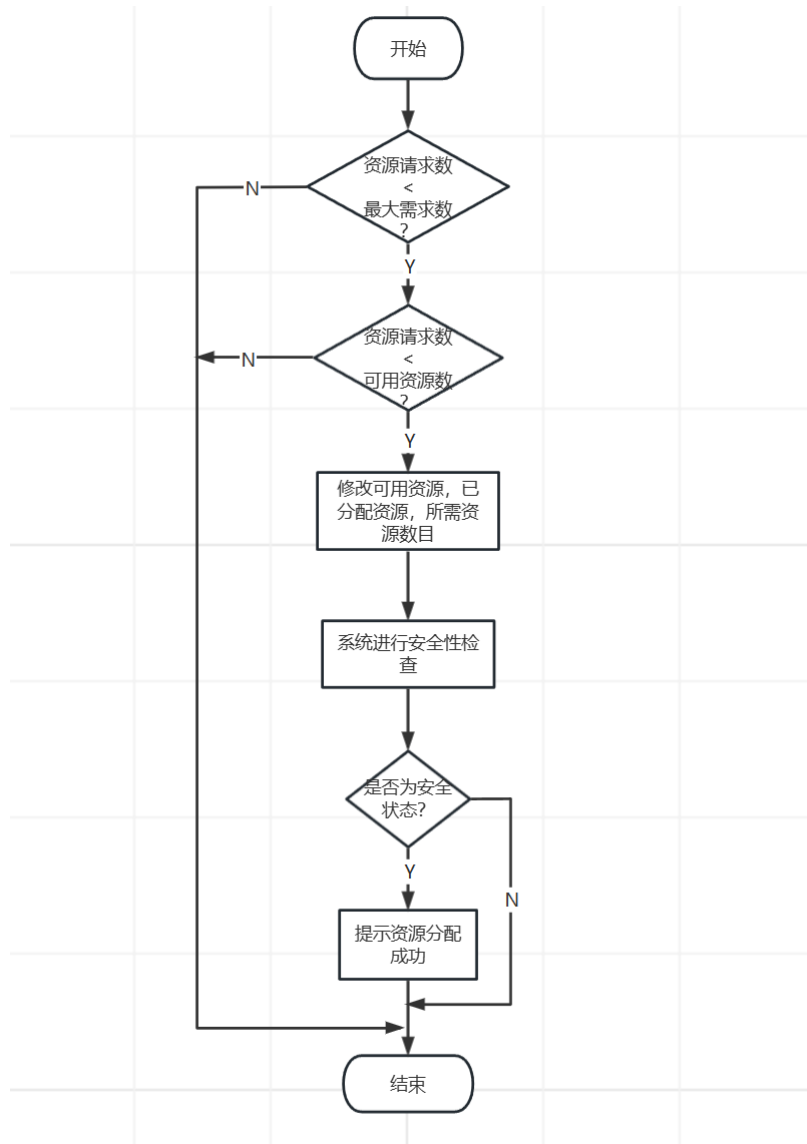
```
int resource[N];           //总资源
int available[N];          //未分配的资源
int claim[M][N];           //进程声明的最大资源
int alloc[M][N];           //已经分配给进程的资源
int need[M][N];            //进程所需要的资源数量
```

由于在寻找安全序列的时候，未分配的资源 and 进程已经分配的资源是动态变化的，因此可以寻找一个替代和标记该进程的资源状态

```
int List[M];               //存放安全序列
int work[N];               //未分配资源的代替
int finish[M];             //标记是否被加入到安全序列中
```

```
void init()           //初始化进程资源状况
void printstate()     //打印进程资源表
int safe()            //安全性检查算法
```

5.2、流程图



5.3、算法思路

5.3.1、安全性检查算法:

1. 初始化work:=available; finish:=0;
2. 寻找满足条件的i: A、finish[i]=0; B、need[i]≤work; 如果不存在, 则转步骤4
3. work:=work+alloc[i]; finish[i]=1; 同时记录安全序列; 转步骤2
4. 若对所有i,finish[i]=1,则系统处于安全状态, 否则处于不安全状态

5.3.2、主程序：

- 1、首先进行特殊情况判断申请资源超过最大请求和资源不足
- 2、然后尝试进行资源分配

```
alloc[n][0]+=x;
alloc[n][1]+=s;
alloc[n][2]+=j;
available[0]-=x;
available[1]-=s;
available[2]-=j;
for(int i=0;i<M;i++){
    for(int j=0;j<N;j++){
        need[i][j]=claim[i][j]-alloc[i][j];
    }
}
```

3. 系统进行安全性算法，检查此次分配后，系统是否还处于安全状态，若安全，把资源分配给进程p[i]；否则，恢复原来的资源分配状态，让进程p[i]等待。

6、实验结果分析

程序运行时初值：

```
3种资源的总量：
9 3 6
4个进程所需要的最大资源数量：
3 2 2
6 1 3
3 1 4
4 2 2
4个进程已经拥有的资源数量：
1 0 0
5 1 1
2 1 1
0 0 2
进程声明的最大资源Claim：
3 2 2
6 1 3
3 1 4
4 2 2

进程已经拥有的资源Alloc：
1 0 0
5 1 1
2 1 1
0 0 2

need:
2 2 2
1 0 2
1 0 3
4 2 0

总资源Resource:
9 3 6

未分配的资源Available:
1 1 2

init is safe!
安全序列：1 0 2 3
请输入要申请的进程号： ■
```

进程0申请R1和R3的一个资源，由于不存在安全序列，因此拒绝分配该资源给0号进程，恢复原来的资源分配状况：

```
请输入要申请的进程号: 0
请输入要申请的资源:1 0 1
它不是安全状态,拒绝分配资源
请输入要申请的进程号:
```

进程1申请R1和R3的一个资源, 存在安全序列, 允许分配该资源给进程1, 并打印当前资源分配情况:

```
它不是安全状态,拒绝分配资源
请输入要申请的进程号: 1
请输入要申请的资源:1 0 1
它是安全状态
安全序列: 1 0 2 3
进程声明的最大资源Claim:
3 2 2
6 1 3
3 1 4
4 2 2

进程已经拥有的资源Alloc:
1 0 0
6 1 2
2 1 1
0 0 2

need:
2 2 2
0 0 1
1 0 3
4 2 0

总资源Resource:
9 3 6

未分配的资源Available:
0 1 1

请输入要申请的进程号: _
```

7、小结与心得体会

这次实验模拟了操作系统的银行家算法, 让我更加深刻的理解了进程死锁这一概念, 也学习了如何进行死锁预防和死锁避免, 而操作系统的银行家算法正是通过死锁避免来实现的, 通过自己编写代码, 大大加深了对于操作系统的银行家算法细节的理解, 理解了如何通过安全性算法来检查系统状态是否安全。

实验七、磁盘调度算法的模拟与实现

1、实验题目

磁盘调度算法的模拟与实现

2、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。

3、背景知识

- (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。

4、模块设计

本实验主要分为两个模块，第一个模块主要是大量随机数的生成，第二个模块则是实现各类磁盘调度算法，先进先出算法、最短服务时间优先算法、SCAN（电梯算法）和C-SCAN（循环扫描）算法。

5、详细设计

5.1、数据结构

```
#define MAX 1001//定义磁盘请求数最大值为MAX-1
int queue1[MAX],queue2[MAX],queue3[MAX],queue4[MAX];//存放磁盘请求的磁道号队列
int HEAD;//磁道号的初始位置
```

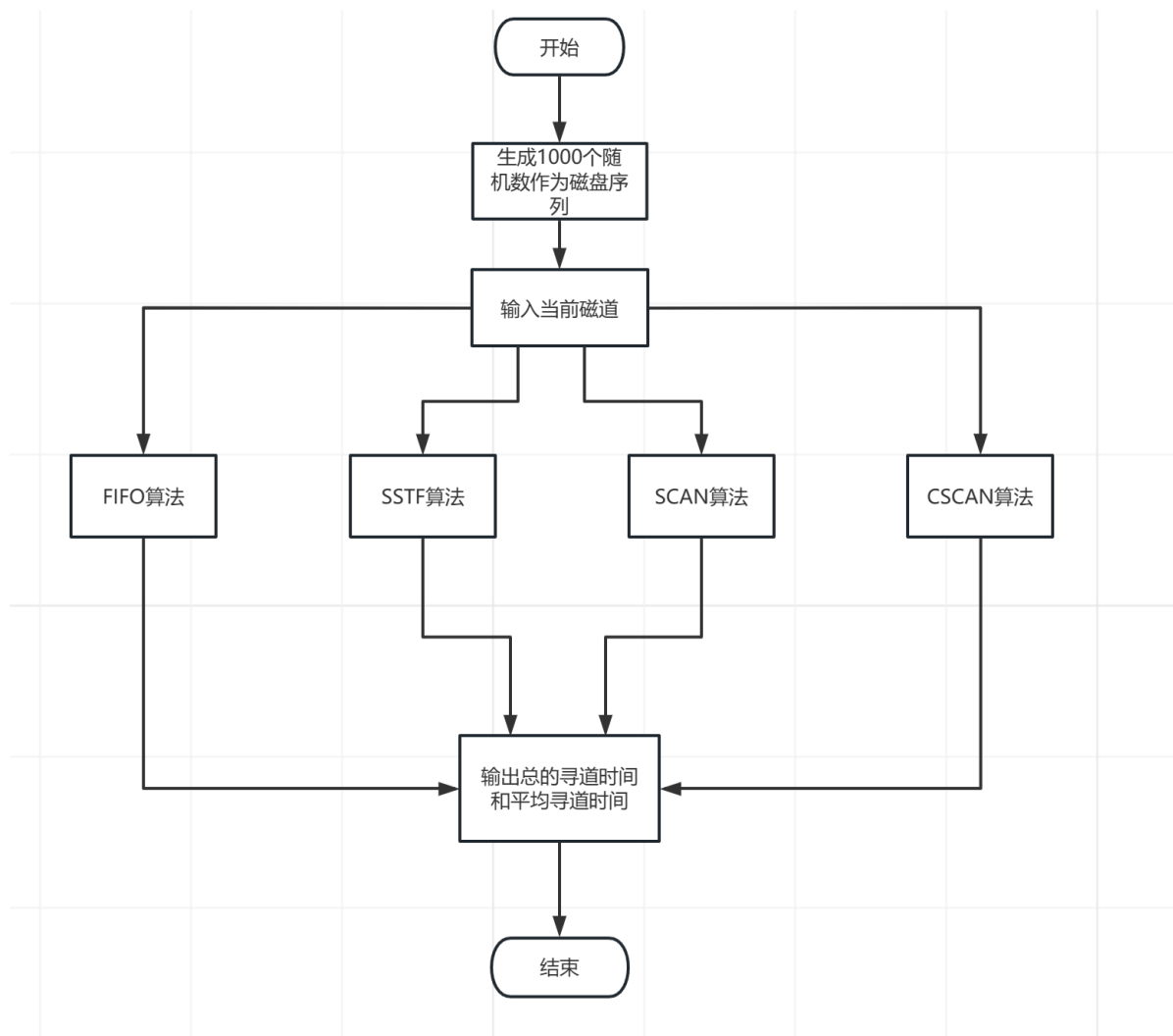
随机种子的生成

```
srand(time(0)); //初始化种子
rand()%100; //磁道号的范围为0~99之间
```

主要函数说明:

```
void FIFO(); //先进先出算法
void SSF(); //最短服务时间优先算法
void SCAN(); //SCAN(电梯算法)
void CSCAN(); //C-SCAN(循环扫描)算法
```

5.2、流程图



5.3、算法思路

5.3.1、先进先出算法

1. 从磁盘队列中依次取出下一步要访问的磁道
2. 记录磁道移动的距离

```
//seek寻道总时间，diff磁道偏移量，head当前所处磁道号
for (i = 1; i <= n; i++)
{
    diff = abs(head - queue1[i]);
    seek += diff;
    head = queue1[i];
}
```

3. 序列中是否还存在未被访问的磁道,如果存在则转到步骤1执行,否则转到步骤4执行
4. 输出移动的总磁道距离

5.3.1、最短服务时间优先算法

1. 在磁道中找出未被访问的,且距离当前磁道最短的磁道queue2[pos]
2. 记录磁道移动的距离

```
diff = abs(head - queue2[pos]);    //pos为最小的磁盘号所在的位置
seek += diff;
```

3. 更新当前磁道号位置

```
head = queue2[pos];           //更新当前磁道号位置
queue2[pos] = queue2[tail];    //将移除的磁道号用最末尾的代替，tail--
tail--;                       //tail为当前磁道请求队列中最后一个请求所在的位置
```

4. 序列中是否还存在未被访问的磁道,如果存在则转到步骤1执行,否则转到步骤5执行

5. 输出移动的总磁道距离

5.3.1、SCAN (电梯算法)

1. 初始化磁头移动的位置

2. 在磁道中找出未被访问的,跟磁头方向移动一致的且距离当前磁道最短的磁道queue3[pos]

```
for (j = 1; j <= tail; j++){
    if(queue3[j] >= head && abs(queue3[j] - head) < diff)    //找到离当前磁道号最近的位置
    {
        diff = abs(queue3[j] - head);
        pos = j;
    }
}
```

3. 能否在磁道序列中找出这样的磁道,如果不能则执行以下代码调转磁头反向转到步骤4执行,否则直接转到步骤4执行

```
//当前磁道号最大，处在最右边的情况
if (pos == -1)
{
    int max=queue3[1];
    pos=1;
    for(int k=2;k<=tail;k++){           //寻找最大的磁盘号
        if(queue3[k] >max){
            pos=k;
            max=queue3[k];
        }
    }
}
```

4. 记录磁道移动的距离

```
diff = abs(head - queue3[pos]);
seek += diff;
```

5. 更新当前磁道号位置

```
head = queue3[pos];           //更新磁道号位置
queue3[pos] = queue3[tail];    //将移除的磁道号用最末尾的代替，tail--
tail--;                       //tail为当前磁道请求队列中最后一个请求所在的位置
```

6. 序列中是否还存在未被访问的磁道,如果存在则转到步骤2执行,否则转到步骤7执行

7. 输出移动的总磁道距离

5.3.1、C-SCAN（循环扫描）算法

1. 初始化磁头移动的位置
2. 在磁道中找出未被访问的,跟磁头方向移动一致的且距离当前磁道最短的磁道`queue4[pos]`

```
for (j = 1; j <= tail; j++){
    if (queue4[j] >= head && abs(queue4[j] - head) < diff)
    {
        diff = abs(queue4[j] - head);
        pos = j;
    }
}
```

3. 能否在磁道序列中找出这样的磁道,如果不能则执行以下代码将磁头回到初始方向的最左边且方向不变转到步骤4执行,否则直接转到步骤4执行

```
//当前磁道号最大，处在最右边的情况
if (pos == -1)
{
    int min=queue4[1];
    pos=1;
    for(int k=2;k<=tail;k++){
        if(queue4[k]<min){
            pos=k;
            min=queue4[k];
        }
    }
}
```

4. 记录磁道移动的距离

```
diff = abs(head - queue4[pos]);
seek += diff;
```

5. 更新当前磁道号位置

```
head = queue4[pos];
queue4[pos] = queue4[tail];
tail--;
```

//更新磁道号位置
//将移除的磁道号用最末尾的代替，`tail--`
//`tail`为当前磁道请求队列中最后一个请求所在的位置

6. 序列中是否还存在未被访问的磁道,如果存在则转到步骤2执行,否则转到步骤7执行
7. 输出移动的总磁道距离

6、实验结果与分析

首先测试算法的正确性，测试俩组数据结果如下：

```
int queue1[MAX]={0,1,5,2,9,7}, queue2[MAX]={0,1,5,2,9,7}, queue3[MAX]={0,1,5,2,9,7}, queue4[MAX]={0,1,5,2,9,7};
选择 D:\学习资料\Dev-C++\Project\OS\demo7\7.exe
输入刚开始的磁道号: 10
FIFO总的寻道时间: 25
FIFO平均寻道时间: 5.000000
• SSF总的寻道时间: 9
SSF平均寻道时间: 1.800000
SCAN总的寻道时间: 9
SCAN平均寻道时间: 1.800000
CSCAN总的寻道时间: 17
CSCAN平均寻道时间: 3.400000
```

道号

```
int queue1[MAX]={0,1,5,2,4,9},queue2[MAX]={0,1,5,2,4,9},queue3[MAX]={0,1,5,2,4,9},queue4[MAX]={0,1,5,2,4,9};
D:\学习资料\Dev-C++\Project\OS\demo7\7.exe
输入刚开始的磁道号: 3
FIFO总的寻道时间: 16
FIFO平均寻道时间: 3.200000
SSF总的寻道时间: 10
SSF平均寻道时间: 2.000000
SCAN总的寻道时间: 14
SCAN平均寻道时间: 2.800000
CSCAN总的寻道时间: 15
CSCAN平均寻道时间: 3.000000
```

明显可以看出结果的正确性，接下来计算大批数据的测试，得出一般规律：

输入刚开始的磁道号: 6 FIFO总的寻道时间: 32484 FIFO平均寻道时间: 32.484001 SSF总的寻道时间: 105 SSF平均寻道时间: 0.105000 SCAN总的寻道时间: 192 SCAN平均寻道时间: 0.192000 CSCAN总的寻道时间: 197 CSCAN平均寻道时间: 0.197000	输入刚开始的磁道号: 34 FIFO总的寻道时间: 32831 FIFO平均寻道时间: 32.831001 SSF总的寻道时间: 133 SSF平均寻道时间: 0.133000 SCAN总的寻道时间: 164 SCAN平均寻道时间: 0.164000 CSCAN总的寻道时间: 197 CSCAN平均寻道时间: 0.197000	输入刚开始的磁道号: 50 FIFO总的寻道时间: 33310 FIFO平均寻道时间: 33.310001 SSF总的寻道时间: 148 SSF平均寻道时间: 0.148000 SCAN总的寻道时间: 148 SCAN平均寻道时间: 0.148000 CSCAN总的寻道时间: 197 CSCAN平均寻道时间: 0.197000	输入刚开始的磁道号: 70 FIFO总的寻道时间: 32692 FIFO平均寻道时间: 32.692001 SSF总的寻道时间: 169 SSF平均寻道时间: 0.169000 SCAN总的寻道时间: 128 SCAN平均寻道时间: 0.128000 CSCAN总的寻道时间: 197 CSCAN平均寻道时间: 0.197000
Process exited after 6.765 seconds with return value 0 请按任意键继续. . .	Process exited after 1.797 seconds with return value 0 请按任意键继续. . .	Process exited after 1.88 seconds with return value 0 请按任意键继续. . .	Process exited after 3.699 seconds with return value 0 请按任意键继续. . .

可以看出当磁盘请求很多时，磁盘调度的效率高依次是最短服务时间优先、SCAN算法、CSCAN算法、先进先出算法，且CSCAN算法的平均寻道时间稳定在0.197。

7、小结与心得体会

平时我们用的最多的就是文件操作，那么文件是存取磁盘的哪个位置呢，如何读取文件呢，通过这次实验我对磁盘的组成有了一定的了解，知道了从三个参数磁头号、柱面号、扇区号就可以唯一确定一个地址以及彻底理解了磁盘调度算法，知道什么样的情形下选用什么样的算法效率最高。

实验八、虚拟内存系统的页面置换算法模拟

1、实验题目

虚拟内存系统的页面置换算法模拟

2、实验目的

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

3、背景知识

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

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

4、模块设计

本次设计一共包括俩大部分，第一大部分就是要设计出一个体现局部性原理的随机数列，然后将它当作指令序列；第二大部分则是要模拟实现OPT，FIFO，LRU算法。

5、详细设计

5.1、数据结构

页框的数据结构：

```
struct MemoryCell
{
    int index;           //页号
    int time;            //记录页面使用的时间
};
```

OPT算法和FIFO算法的实现只需知道页号是多少即可，而LRU算法的实现则需知道上次使用距离当前的时间，因此增加的一个时间变量来记录。

相关的宏定义：

```
#define MEM_PAHE_NUM 4           //最大的内存页面数
#define COMMD_NUM 320           //指令序列的数量
#define PAGE_COMMD 10           //一个页面所含有的指令数
#define MAX_FAR 1000000000
```

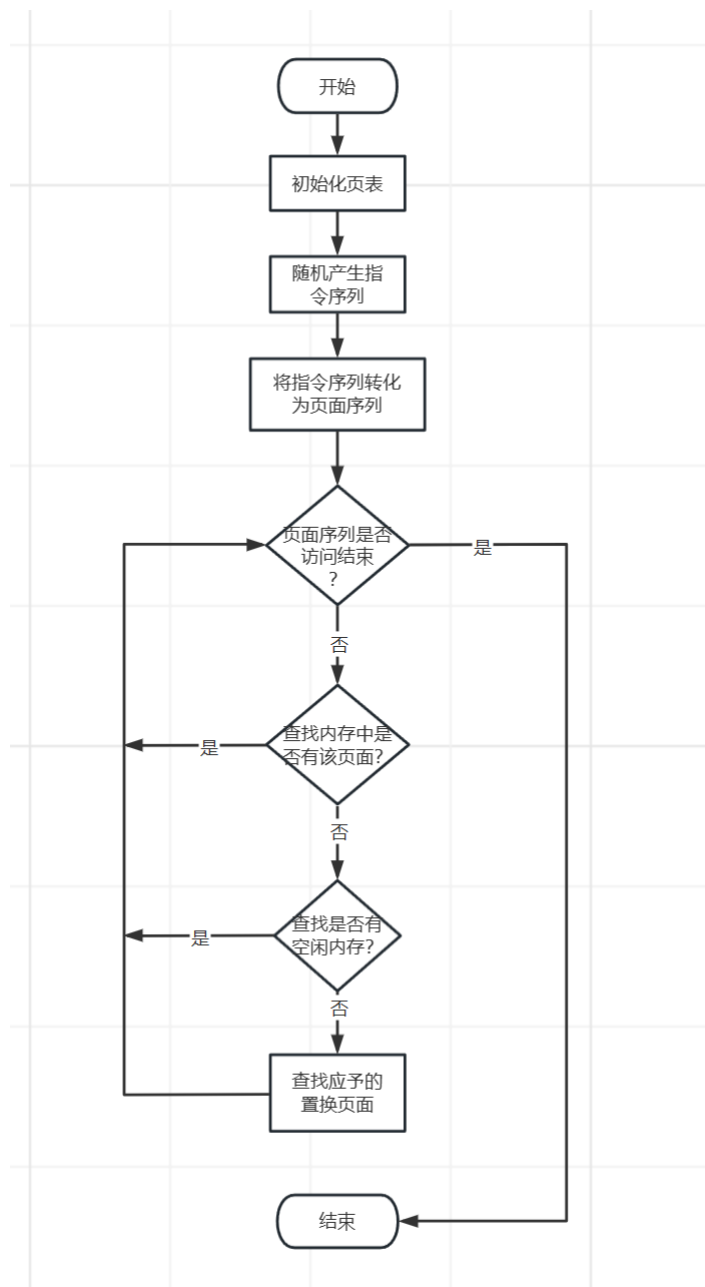
相关的全局变量：

```
int commds[COMMD_NUM];           //存放的是每一条指令的页号
MemoryCell memory[MEM_PAHE_NUM]; //内存块
vector<int> order_commd;          //指令序列
vector<int> order_page;           //指令序列对应的指令页号
```

函数说明：

```
void initPage();           //初始化页表
void createArray();        //生成序列
double FIFO();             //用FIFO置换算法
double LRU();              //用LRU置换算法
double OPT();              //用OPT置换算法
```

5.2、流程图



5.3、算法思路

5.3.1、指令序列的设计

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

1. 根据上述规则，实现其代码，逻辑很简单，需要注意的是 $m=319$ 的特殊情况。

```

if(m == COMMD_NUM-1)
    continue;           //m=319
  
```

2. 将指令序列转为页面序列

```
//将指令序列转化为页号序列
for(int i=0;i<order_commd.size();i++)
{
order_page.push_back(commnds[order_commd[i]]);
}
```

3. 缺页时，内存页面有空闲则执行下面关键代码

```
if(memory_page_num < MEM_PAHE_NUM)
{
memory[memory_page_num++].index = order_page[i];
}
```

4. 缺页时，内存页面没有空闲则需要执行置换算法

```
for(int j=1;j<memory_page_num;j++)
{
memory[j-1] = memory[j];    //将第一个置换出去
}
memory[memory_page_num-1].index=order_page[i];
```

FIFO：将内存中的第一个页面置换出去，后面的往前移动，末尾添加新的页面。

```
for(int j=0;j<memory_page_num;j++)
{
if(memory[minn].time > memory[j].time)
{
minn = j;
}
}
memory[minn].time = i;    //更新时间戳
memory[minn].index = order_page[i];
```

LRU：寻找历史页面距离当前内存中的页面的最近距离，更新时间戳。

```
for(int j=0;j<memory_page_num;j++)
{
for(int k=i+1;k<order_page.size();k++)
{
if(memory[j].index == order_page[k]){ //未来出现的位置
far[j] = k;
break;
}
}
}
```

OPT：寻找未来页面距离当前内存页面最远的距离。

6、实验结果与分析

运行结果如下：


```
D:\学习资料\Dev-C++ 19 17 31* 11
27 9* 0 4 19 17 31* 11
27 9 0* 4 19 17 31 11*
27 9 0* 4 19 17 31 11*
27 9 13* 4 F 19* 17 31 11
27* 9 13 4 23* 17 31 11 F
28* 9 13 4 F 23* 17 31 11
1* 9 13 4 F 23 20* 31 11 F
16 17* 14 4 F 23 20* 31 11
16 17 14 15* F 23* 20 31 11
16 17 14 15* 18* 20 31 11 F
16 17 14 24* F 18* 20 31 11
16* 17 14 24 3* 20 31 11 F
16* 17 14 24 3* 20 31 11
16 17 14* 24 25* 20 31 11 F
16 17 14 24* 4* 20 31 11 F
8* 17 14 24 F 5* 20 31 11 F
8* 17 14 24 2* 20 31 11 F
1* 17 14 24 F 2* 20 31 11
1* 17 14 24 14* 20 31 11 F
1 17* 14 24 OPT err =135
1 17* 14 24 fifo = 0.550000 lru = 0.540625 opt = 0.421875
11* 17 14 24 F 12* 9 3 4*
11* 17 14 24 12 9* 3 4
13* 17 14 24 F OPT err =141
6* 17 14 24 F fifo = 0.571875 lru = 0.568750 opt = 0.440625
6* 17 14 24
6 4* 14 24 F
6 4* 14 24
6* 4 14 24
OPT err =128
fifo = 0.521875 lru = 0.521875 opt = 0.400000

Process exited after 1.123 seconds with return value 0
请按任意键继续. . .

Process exited after 1.055 seconds with return value 0
请按任意键继续. . .
```

可以很明显的看出页面置换算法中效率最高的是OPT算法，其次是LRU，最后是FIFO，LRU和FIFO的效率差距不是很明显，虽然OPT算法的效率很高，但是对于未来页面的预测是很难实现的。

7、小结与心得体会

以前对于虚拟内存的页面是一个很模糊的概念，通过这次实验熟悉了虚拟内存页面，更加了解了页面的置换算法，可以说收获多多，感觉自己的知识又得到更进一步的提高。

实验九、基于信号量机制的并发程序设计

1、实验题目

基于信号量机制的并发程序设计

2、实验目的

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

3、背景知识

- UNIX/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。使用 IPC 中提供的 semget () , semop () , 及 semctl () 等信号量系统调用。

4、模块设计

主要分为俩大模块，第一个模块主要是定义和初始化信号量，找到其中蕴含的同步和互斥关系；第二个模块主要实现哲学家就餐的逻辑部分，使其不会发生死锁。

5、详细设计

5.1、数据结构

哲学家的数量以及左右筷子的编号

```
#define N 5 // 哲学家数量
#define LEFT (i + N - 1) % N // 左邻居编号
#define RIGHT (i + 1) % N // 右邻居编号
```

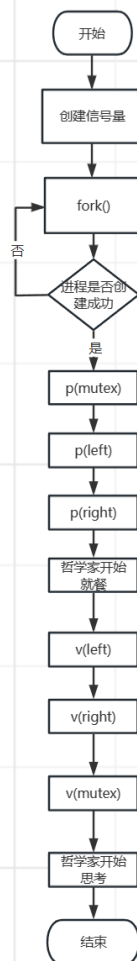
主要定义了五个信号量分别代表哲学家，和一个互斥信号量。

```
sem_t mutex; // 互斥信号量
sem_t s[N]; // 哲学家信号量
```

初始化信号量，由于哲学家初始状态都没有拿筷子，因此都为1，都可以去拿筷子

```
sem_init(&mutex, 0, 1); // 初始化互斥信号量
for (i = 0; i < N; i++) {
    sem_init(&s[i], 0, 1); // 初始化哲学家信号量
}
```

5.2、流程图



5.3、算法思路

1. 哲学家按0到4编号,编号为i的哲学家左边的筷子left为i, 右边的筷子right为(i+1)%5
2. 定义互斥量数组S[5]和一个互斥量mutex
3. 伪代码

```
semaphore S[5]={1,1,1,1,1}
semaphore mutex=1
pi:
while(1){
P(mutex)
P(left)      //拿左筷子
P(right)     //拿右筷子
V(mutex)
//吃饭.....
V(left)      //放左筷子
V(right)     //放右筷子
//思考.....
}
```

根据上述伪代码实现即可

6、实验结果与分析

运行结果如下:

```
llh@llh-virtual-machine:~/demo/eat$ ./eat
Philosopher 0 is thinking.
Philosopher 1 is thinking.
Philosopher 3 is thinking.
Philosopher 2 is thinking.
Philosopher 4 is thinking.
Philosopher 2 is eating.
Philosopher 2 is thinking.
Philosopher 3 is eating.
Philosopher 2 is eating.
Philosopher 3 is thinking.
Philosopher 1 is eating.
Philosopher 1 is thinking.
Philosopher 4 is eating.
Philosopher 2 is thinking.
Philosopher 0 is eating.
Philosopher 4 is thinking.
SPhilosopher 0 is thinking.
Philosopher 3 is eating.
Philosopher 4 is eating.
```

可以看出运行一段时间后不会发生死锁，唯一的缺点就是如果调度太频繁的话，容易引起某位哲学家饥饿。

7、小结与心得体会

通过这次实验，我进一步的了解了Linux下如何使用信号量来解决同步互斥问题，多个进程并发进行如何去预防死锁，去避免死锁，只需要打破死锁的充分必要条件之一即可实现，死锁的充分必要条件分别是互斥，占有且等待、不可抢占、循环等待。

实验十、实现一个简单的 shell 命令行解释器

1、实验题目

实现一个简单的 shell 命令行解释器

2、实验目的

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

3、背景知识

本实验要使用创建子进程的 fork()函数,执行新的命令的 exec () 系列函数，通常 shell 是等待子进程结束后再接受用户新的输入，这可以使用 waitpid()函数。

4、模块设计

主要分为两个模块，第一个模块主要是完成Linux自带的一些命令，只需要执行exec命令即可完成，第二个模块则是需要深入底层，通过直接调用系统调用完成。

5、详细设计

5.1、数据结构

```
#define MAX_CMD_LEN 1024    //最大命令长度
#define MAX_ARG_NUM 64      //最大参数命令
```

函数说明：

打印标识符号

```
void print_prompt()
```

根据输入的参数解析命令

```
int read_command(char* cmd_buf, char ** arg_bufs, int max_arg_num)
```

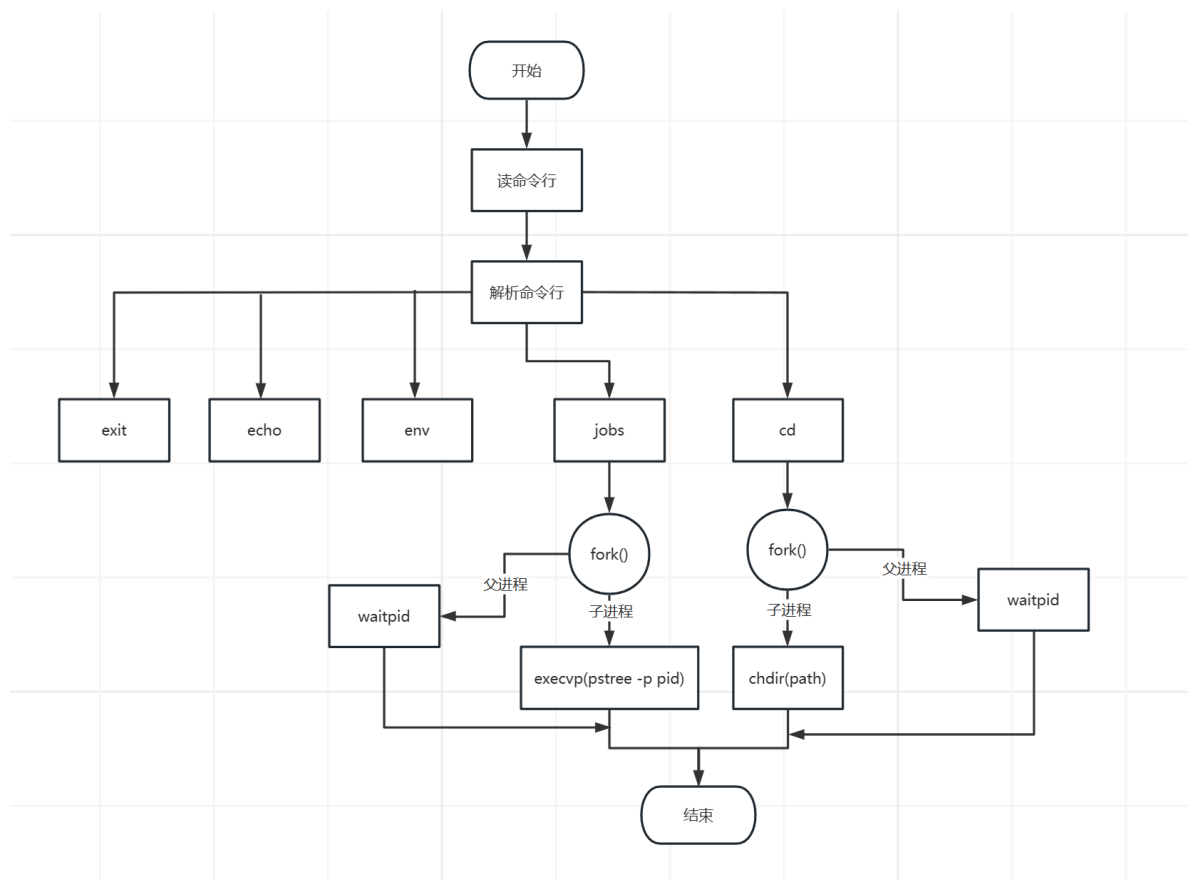
通过调用封装好的系统命令，执行exec命令

```
void execute_command(int arg_num, char ** args)
```

cd命令和jobs命令函数

```
cd(char *path)
void jobs()
```

5.2、流程图



5.3、算法思路

命令解释器首先是一个无限循环。

1. 首先在main函数中，我们需要设置一个无限循环，不断让用户输入命令
2. 使用fgets函数进行标准输入操作

```
fgets(cmd_buf, MAX_CMD_LEN, stdin);
```

3. 通过if-else结构进行解析命令字符串

```
//解析命令的关键代码
char *token = strtok(cmd_buf, " ");
token = strtok(NULL, " ");
```

3. 判断若输入的是exit命令，则直接break跳出循环，表示退出shell
4. 若判断为echo和env，则直接使用Linux封装的库函数即可
5. 若为jobs则需要调用自定义函数，创建子进程来到达该目的
6. 若为cd则直接调用系统调用chdir(path)来实现

6、实验结果与分析

```
llh@llh-virtual-machine:~/demo/shell$ ls
shell  shell.c
llh@llh-virtual-machine:~/demo/shell$ ./shell
>>>ls
shell  shell.c
>>>echo llh
llh
>>>jobs
cpid=3350
shell(3350)——pstree(3355)
>>>cd
>>>ls
公共的  模板  视频  图片  文档  下载  音乐  桌面  demo  snap
>>>qit
Fail to execute command qit.
>>>exit
llh@llh-virtual-machine:~/demo/shell$
```

可以看出基本的功能都可以实现

7、小结与心得体会

通过这次实验，了解了命令解释器的基本工作原理，实现了一些基本命令的解释，对Linux有了更深的了解，不过还有好多命令没有实现，原来Linux的强大之处还有好多需要探索，激发了我对Linux的极大兴趣。

实验十一、简单二级文件系统设计

1、实验题目

简单二级文件系统设计

2、实验目的

让学生自己动手设计一个简单的文件系统，进一步巩固操作系统的文件系统的理论知识。

3、背景知识

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

4、模块设计

主要分为三大模块，第一个模块为磁盘类的设计包含磁盘卷的结构和组织，第二个模块为文件类和目录类的设计，第三个模块则是文件系统的设计。

5、详细设计

5.1、数据结构

主要的数据结构

```
const int BLOCK_SIZE = 512; // 磁盘块大小
const int BLOCK_NUM = 1024; // 磁盘块数量
const int MAX_FILE_NUM = 128; // 最大文件数量
const int MAX_FILE_NAME_LEN = 32; // 最大文件名长度
const int MAX_FILE_SIZE = 1024 * 1024; // 最大文件大小
```

磁盘类：管理磁盘操作，从磁盘块中读出数据，将数据写入磁盘块

```
class Disk {
    //从磁盘块读出数据
    void read(int block, char* buf)
    // 将数据写入磁盘块
    void write(int block, char* buf)
private:
    char* data; //定义整个磁盘卷
};
```

文件类：文件的一些基本属性得到文件的名称、文件所处的磁盘块、文件的大小

```
class File {
    char* getName() //得到文件名
    int getBlock() //得到文件块
    int getSize() //得到文件大小
    void setSize(int size) //设置文件大小
private:
    char name[MAX_FILE_NAME_LEN]; //名称
    int block; //占用的磁盘块
    int size; //大小
};
```

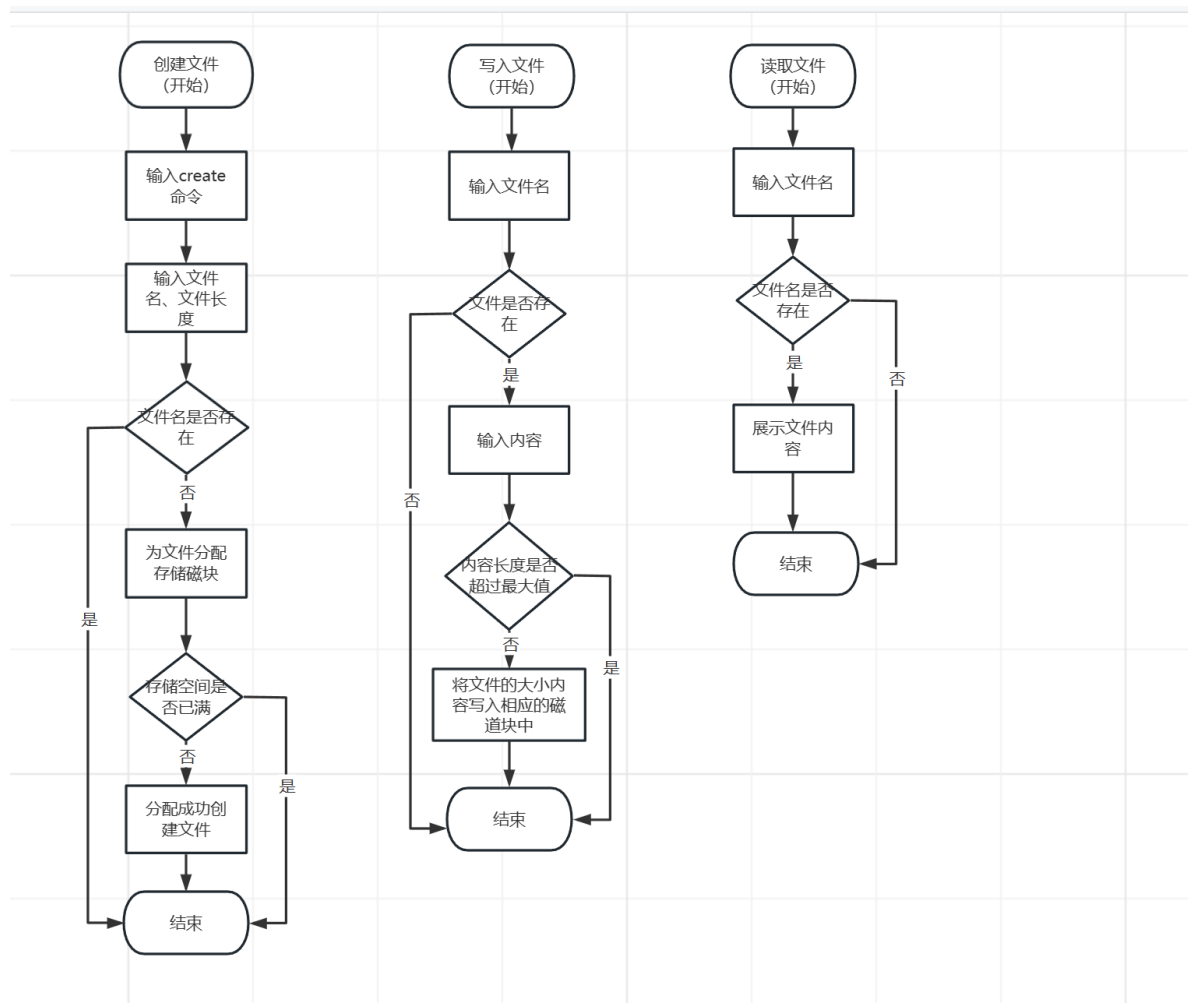
目录类：目录名称和所含的文件以及子目录项和父目录

```
class Directory {
private:
    char name[MAX_FILE_NAME_LEN]; //名称
    Directory* parent; //父目录
    vector<File*> files; //含有的文件项
    vector<Directory*> dirs; //含有的目录项
};
```

文件系统类：对文件的一些基本操作，读写文件创建删除文件以及创建目录。

```
class FileSystem {  
    //文件系统的私有属性  
    Disk* disk;  
    int fat[BLOCK_NUM];    //FAT表用于记录整个磁盘卷的使用情况  
    File* files[BLOCK_NUM]; //存放文件系统的所有文件  
    Directory* root;        //根目录  
    Directory* current_dir; //当前目录  
};
```

5.2、流程图



5.3、算法思路

1. 读取命令行参数，解析命令使用getline函数

```
cin.getline(cmd, 256);  
char* p = strtok(cmd, " ");    //设置标记符
```

2. 初始化FAT表，FAT表用于记录整个磁盘卷的使用情况以及根目录让当前目录指向根目录

```
memset(fat, -1, sizeof(fat));    //初始化为-1，即整个磁盘卷都是空的  
root = new Directory();          //根目录  
current_dir = root;             //当前目录等于根目录
```


3. 若命令是create，则将文件名存到文件系统中，并在FAT表中寻找空闲的块，将其块标记为-2

```
//查找空闲的磁盘块
int findFreeBlock() {
    for (int i = 0; i < BLOCK_NUM; i++) {
        if (fat[i] == -1) {
            return i;
        }
    }
    return -1;
}
```

4. 若命令为write，则先将磁盘中该文件的内容读出来，然后再将所有文件内容写入进去，如果文件内容大于一个块，则上一个块存取的编号则为下一个块的编号，最后一个存放-2表示文件的结束。

```
//关键部分
while (size > 0) {
    char buf[BLOCK_SIZE];
    disk->read(block, buf);
    int n = min(size, BLOCK_SIZE);
    memcpy(buf, data + offset, n);
    disk->write(block, buf);
    offset += n; //更新偏移量
    size -= n;
    if (size > 0) { //先取出来，在存到磁盘里面
        int next = findFreeBlock();
        if (next == -1) {
            cout << "No free block" << endl;
            return;
        }
        fat[block] = next;
        block = next;
        fat[block] = -2;
    }
}
```

5. 若命令为read，找到文件所对应的磁盘块，将文件内容从磁盘块中读出即可读出

6. 若命令为mkdir，则将其目录的名称存于文件系统当中

6、实验结果与分析

没有实现其删除功能，还有许多功能需要完善和改进，存在许多不足，不过基本功能还是实现了。

```
llh@llh-virtual-machine:~/demo/file$ ls
file  file.c
llh@llh-virtual-machine:~/demo/file$ ./file
> ls
> clear
Unknown command
> create llh
> mkdir llh
> ls
llh
llh
> write llh "1314520zxk" 20
> read llh 20
"1314520zxk" 20
> cd llh
> ls
> exit
llh@llh-virtual-machine:~/demo/file$
```

7、小结与心得体会

这是最后一个实验也是最让人痛苦的一个实验，不过通过这个实验让我对操作系统有了更清晰的认识了，了解了磁盘的基本组织方式和结构，使得操作系统的脉络一下子就清晰起来了，收获非常多，收益良浅。

附件 2：课程设计报告日志参考格式

时间		设计内容简要记录
第 17 周	星期一	熟悉了实验一和查询了实验一相关的 API 函数,了解了实验一是怎么样一步一步的完成的。
	星期二	首先搭建了 Linux 下的环境,熟悉了 Linux 下的一些操作,一些终端常用命令,对实验二和实验三的内容大概了解了一下,熟悉了 Linux 如何创建进程,如何控制进程。
	星期三	今天主要是完成实验四和实验五,查询了相关的 API 函数,熟悉了实验四和实验五的工作原理。
	星期四	向老师请求测试验证性实验,完成前五个验证性的实验报告。
	星期五	熟悉了实验六银行家算法和实验七磁盘调度算法的工作原理,仔细看了书上的介绍,对银行家算法和磁盘调度有了一个更清晰的认识,将流程图画了出来。然后晚上就实现了具体的代码。

周末		休息一天，没做什么事，主要调养了一下身体。
第 18 周	星期一	实验八虚拟内存页面置换算法这个实验查了很多资料，了解了算法底层的东西，实现了其算法。
	星期二	实验九和实验十这两个实验画了流程图和写了伪代码，过程不是很难，无非就是系统调用。
	星期三	花了我一上午时间用来设计文件系统的结构如何组织磁盘结构，很烧脑子，最终道晚上才完成该实验。
	星期四	向老师测试以及撰写实验报告
	星期五	撰写实验报告

注：请同学们及时填写，以便指导老师及时了解你的课程设计进展等情况。