

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： 计算机科学与技术七班

姓 名： 陈琪琪

学 号： 2102010629

指导教师： 肖小聪

时 间： 2023.06.05~2023.06.17

地 点： 逸夫楼416

指导教师评语：

成绩：

等级：

签名：

年 月 日

目 录

实验一 Windows 进程管理	1
实验二 Linux 进程控制	6
实验三 Linux 进程间通信	10
实验四 Windows 线程的互斥与同步	13
实验五 内存管理	17
实验六 银行家算法的模拟与实现	22
实验七 磁盘调度算法的模拟与实现	27
实验八* 虚拟内存系统的页面置换算法模拟	32
实验九* 基于信号量机制的并发程序设计	37
实验十* 实现一个简单的 shell 命令行解释器	40
实验日志	45

实验一 Windows 进程管理

一. 实验题目

Windows进程管理

二. 实验目的

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

三. 实验内容

1. 实验原理

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。

2. 实验步骤

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

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

代码:

```
#include <iostream>
using namespace std;

int main()
{
    cout << "Hello, Win32 Consol Application" << endl;
```

```
    return 0;
}
```

(2) 创建进程

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

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

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

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

步骤4: 修改清单 2-2 中的程序，将 nClone 的定义和初始化方法按程序注释中的修改方法进行修改，编译成可执行文件（执行前请先保存已经完成的工作）。再按步骤 2 中的方式运行，看看结果会有什么不一样。列出运行结果。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？

代码：

```
// 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;
//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;
/*此处修改无限生成子进程*/
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();
```

(3)父子进程的简单通信及终止进程

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

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

步骤3: 按源程序中注释中的提示，修改源程序 2-3，编译执行（执行前请先保存已经完成的工作），列出运行结果。在程序中加入跟踪语句，或调试运行程序，同时参考 MSDN 中的帮助文件CreateProcess()的使用方法，理解父子进程如何传递参数。给出程序执行过程的大概描述。

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

步骤5: 参考MSDN中的帮助文件CreateMutex()、OpenMutex()、ReleaseMutex()和 WaitForSingleObject()的使用方法，理解父子进程如何利用互斥体进行同步的。给出父子进程同步过程的一个大概描述。

代码:

```
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
        /*INFINITE表示最大, 参数表为对象句柄和毫秒数*/
        //WaitForSingleObject(hMutexSuicide, INFINITE);
        WaitForSingleObject(hMutexSuicide, 0);
        //WaitForSingleObject(hMutexSuicide, 5000);
        // 准备好终止，清除句柄
        std::cout << "Child quitting." << std::endl;
        CloseHandle(hMutexSuicide);
    }
}
```

四．实验结果与分析

实验1-1:

```
D:\CompileSoftware\Dev-C++5.9.2+便携版\OS\test1\test1-1\main.exe
Hello, Win32 Console Application

-----
Process exited after 0.2635 seconds with return value 0
请按任意键继续. . .
```

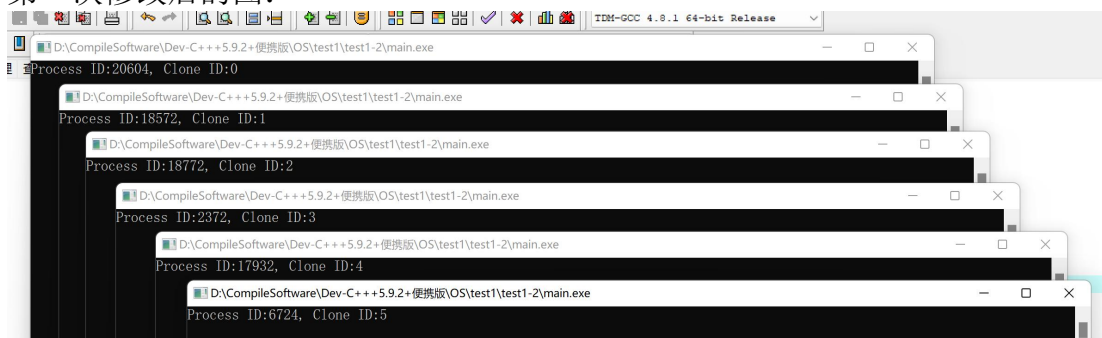
return是返回值，如果越界就会返回一个数，后边execution time 是 C/C++控制台程序执行的时间。

实验1-2:

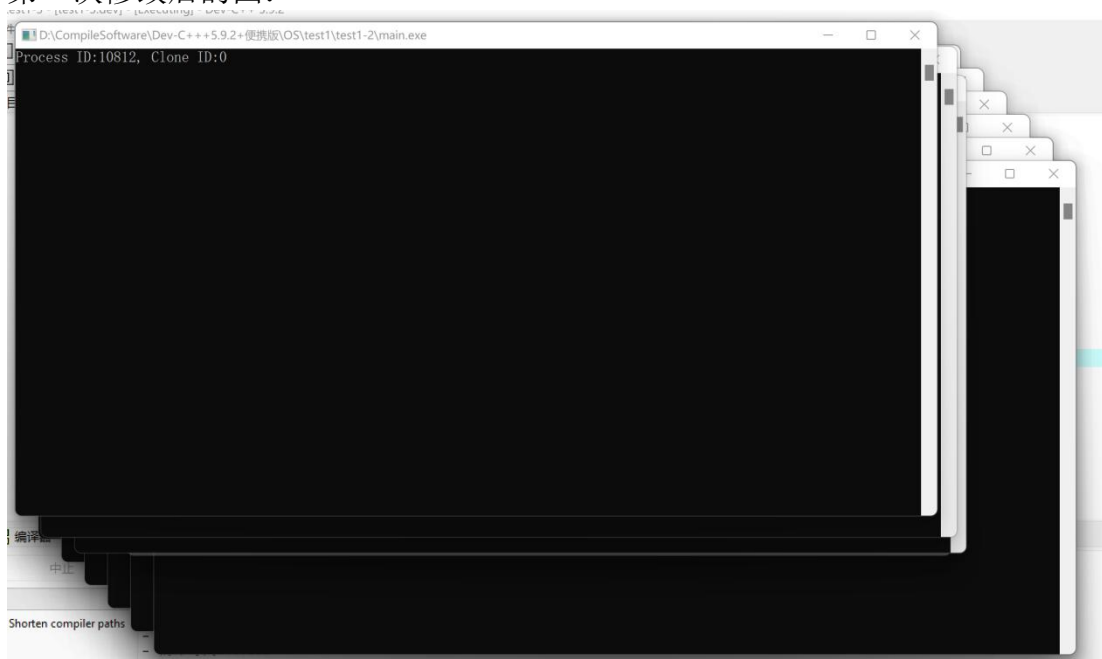
会出现6个窗口，其中5个是子进程，且0号进程结束后其余进程不消失。

第一次修改后与原先展示一样，只是进程号变化了；第二次修改后，子进程无限生成。

第一次修改后的图:



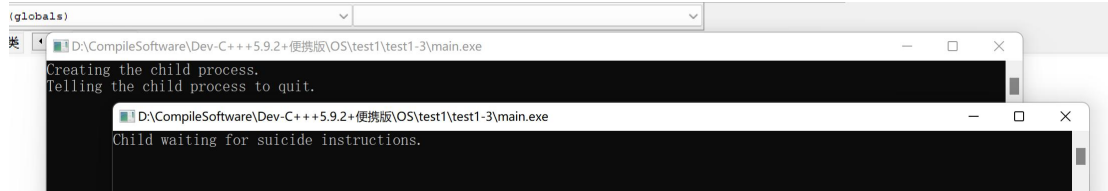
第二次修改后的图:



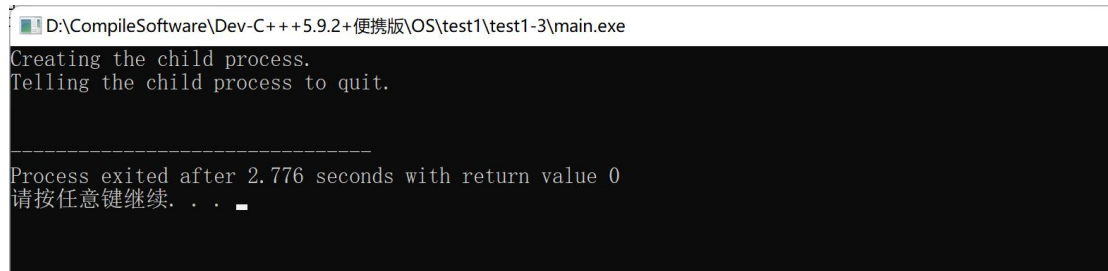
原因: 这是一个父进程生成子进程，再由子进程生成子进程的过程，直至nClone的值等于5为止。由代码可得第二个参数是进程编号，且是它的父进程传进来的值；修改前，将nClone的值赋值为0，当符合if语句中的条件就会将父进程传进来的参数赋值给nClone，会被sscanf()覆盖掉。第二次修改则是在if判断之后，会覆盖掉if中的赋值，将nClone

置为0，比较 $nClone < c_nCloneMax$ 会恒成立，然后就会无限生成子进程，同时子进程的 $nClone$ 参数永远都是1。

实验1-3:



修改后的实验结果:



原因：首先INFINITE在C++表示的正无穷，而函数WaitForSingleObject()的第二参数是等待时间，如果超过这个时间，不管互斥体是否可用都继续运行。所以未修改前子进程在一直等待着互斥信号量的释放，所以处于阻塞状态，所以会出现如图所示的两个窗口，而进行第一次修改后，将时间改至为0，说明父进程几乎等于立即释放锁，所以子进程无需进行等待就可获得，所以看到的窗口只有一个。

五. 小结与心得体会

通过本次实验，更加深刻的理解了windows操作系统下的相关函数的使用，理解了父进程生成子进程，某些相关变量修改后造成的影响，以及这些函数中参数的意义与作用，同时也理解了互斥信号量的使用，可谓收获颇多！

实验二 Linux 进程控制

一. 实验题目

Linux进程控制

二. 实验目的

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

三. 实验内容

1.实验原理

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 execl( 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像置换当前的进程映像. 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 `execl` 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。

2. 实验步骤

(1) 任务一：进程的创建

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

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

步骤 2：在命令行输入`./fork_demo` 运行该程序。

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

代码：

```
srand((unsigned)time(NULL));
while((x=fork())==1);
if (x==0)
{
    sleep(rand() % 2);
```

```

        printf("a");
    }
    else
    {
        sleep(rand() % 3);
        printf("b");
    }
    printf("c");
}

```

(2) 任务二：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 3-1 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 3-2 中的程序 (该程序的执行如图 3-1 所示)，使用 `cc` 或者 `gcc` 编译成可执行文件 `exec_demo`。例如，可以使用 `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。

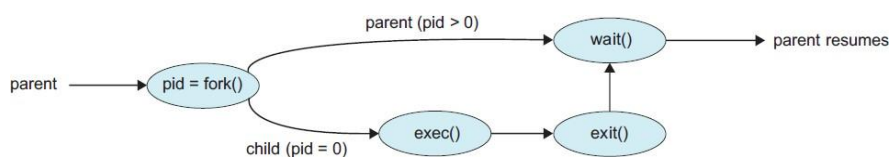


图 3-1 `exec_demo.c` 程序的执行过程

代码：

```

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\n");
}

```

四. 实验结果与分析

实验2-1:

```

[cqq@cqq test]$ gcc test2-1.c -o test2-1.exe
[cqq@cqq test]$ ./test2-1.exe
acbc[cqq@cqq test]$
[cqq@cqq test]$ ./test2-1.exe
bc[cqq@cqq test]$ ac
[cqq@cqq test]$ ./test2-1.exe
bc[cqq@cqq test]$ ac

[cqq@cqq test]$ ./test2-1.exe
bcac[cqq@cqq test]$ ./test2-1.exe
acbc[cqq@cqq test]$ ./test2-1.exe
bcac[cqq@cqq test]$ ./test2-1.exe
bcac[cqq@cqq test]$ ./test2-1.exe
acbc[cqq@cqq test]$

```

原因: 由于fork函数的调用成功, 子进程会复制一份一样父进程的代码执行, 且两者的代码中都是会输出c的, 不同点在于if语句里的执行代码, 都是休眠函数, 且响应时间不同, 于此同时两者会竞争资源从而出现了结果每次执行完结果不一致的效果, 所以有时候bc在前, 有时候ac在前, 而像第一张图的ac之所以在命令行出现之后才出现, 是因为子进程的休眠时间有些许长的缘故。

实验2-2:

```

[cqq@cqq test]$ gcc test2-2.c -o test2-2.exe
[cqq@cqq test]$ ./test2-2.exe
test2-1.c test2-1.exe test2-2.c test2-2.exe
Child Complete
[cqq@cqq test]$

```

原因: 同样是调用了fork函数, 不同也是区别于父子进程之间执行的代码, 对于父进程因为调用了wait函数, 所以会等待一个进程结束再继续下面代码的执行, 从而此时进入了阻塞状态; 对于子进程, 它执行的是execlp函数, execlp可根据path搜索合适的程序运行, 这个函数是负责调用命令行的函数, 输入相应的参数, 则在相应的目录下, 执行相应的命令行命令, 实验则是执行的罗列目录结构的操作, 即ls, 从而出现了此结果。

五. 小结与心得体会

①了解进程状态转换很重要。在Linux系统中, 每个进程都有自己的状态, 如运行态、就绪态和阻塞态等。了解这些状态之间的转换关系可以帮助我们更好地理解进程控制机制。

②信号是一种重要的通信方式。通过发送信号可以实现对目标进程的控制和通知, 如结束进程、暂停/恢复执行等。

通过本次实验, 进一步掌握了有关fork(), wait(), exec系列函数的使用, 以及深刻了解了其相关参数的意义。

实验三 Linux 进程间通信

一. 实验题目

Linux进程间通信

二. 实验目的

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

三. 实验内容

1. 实验原理

fork 系统调用的原理参考实验二。

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。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：

- ①semget()—获得信号量的IPC标识符。
- ②msgget()—获得消息队列的IPC标识符。
- ③shmget()—获得共享内存IPC标识符。

控制 IPC 资源的系统调用有：

- ①semctl()—对信号量资源进行控制的函数。
- ②msgctl()—对消息队列进行控制的函数。
- ③shmctl()—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：

- ①IPC_SET 命令：设置属主的用户标识符和组标识符。
- ②IPC_STAT 和 IPC_INFO 命令：获得资源状态信息。
- ③IPC_RMID 命令：释放这个资源。

操作IPC资源的系统调用有：

- ①semop()—获得或释放一个IPC 信号量。 可以实现P、V 操作
- ②msgsnd()—发送一个 IPC 消息。
- ③msgrcv()—接收一个 IPC 消息。

- ④shmat()——将一个 IPC 共享内存段添加到 进程的地址空间
- ⑤shmdt()——将 IPC 共享内存段从私有的地址空间剥离。

2. 实验要求

- 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观察参考程序, 说明控制消息队列系统调用 msgctl() 在此起什么作用?

3. 实验步骤

步骤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。

步骤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”。

步骤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”。

步骤4: 父进程在 SERVER 和 CLIENT 均退出后结束。

代码:

```
void CLIENT()
{
    int i;
    //flag 本身由操作允许权和控制命令值相“或”得到。
    msgqid=msgget(MSGKEY,0777);
    for (i=10; i>=1; i--)
    {
        msg.mtype=i;
        printf("(client) sent \n");
        //flag 规定当核心用尽内部缓冲空间时应执行的动作
        msgsnd(msgqid,&msg,1024,0);
    }
    exit(0);
}

void SERVER()
{
    //IPC_CREAT | 0400 是否该队列应被创建;
    msgqid=msgget(MSGKEY,0777|IPC_CREAT);
    do
    {
        //type 是用户要读的消息类型:
        msgrcv(msgqid,&msg,1030,0,0);
        printf("(Server) recieved\n");
    }
}
```

```

        while(msg.mtype!=1);
    msgctl(msgqid, IPC_RMID, 0);
    exit(0);
}

```

四. 实验结果与分析

1. 实验结果:

```

[cqq@cqq test]$ gcc test3-1.c -o test3-1.exe
[cqq@cqq test]$ ./test3-1.exe
(client) sent
(client) sent
(client) sent
(client) sent
(client) sent
(client) sent
(client) sent
(client) sent
(client) sent
(client) sent
(client) sent
(Server) recieved
(client) sent
(Server) recieved
(Server) recieved
(Server) recieved
(Server) recieved
(Server) recieved
(Server) recieved
(Server) recieved
(Server) recieved
(Server) recieved
[cqq@cqq test]$

```

多数情况下连续出现sent和连续出现recieved, 理论上应当是两者交替出现(此情景极少出现)。

2. 实验分析

原因: message的传送和控制并不保证完全同步, 当一个程序不再激活状态的时候, 它完全可能继续睡眠, 造成上面现象, 在多次send message后才receive message. 这一点有助于理解消息转送的实现机理。

消息通信的特点: 消息队列是由消息的链表, 存放在内核中并由消息队列标识符标识。消息队列克服了信号传递信息少、管道只能承载无格式字节流以及缓冲区大小受限等缺点。消息的传递, 自身就带有同步的控制。当等到消息的时候, 进程进入睡眠状态, 不再消耗CPU资源。

五. 总结与心得体会

①不同的进程间通信方式适用于不同的场景。例如, 管道适用于单向数据传输, 而共享内存则适用于大规模数据共享。

②进程间通信需要考虑并发性和同步问题。多个进程同时操作共享资源时, 需要使用锁或者其他同步机制来避免竞争条件的出现。

③错误处理非常重要。当进程间通信出现问题时, 需要及时处理并给出错误提示, 以便快速排查和解决问题。

总之, 在学习和实践过程中, 需要注重细节和规范, 以确保程序的正确性、可靠性和高效性。

实验四 Windows 线程的互斥与同步

一. 实验题目

Windows线程的互斥与同步

二. 实验目的

- (1) 回顾操作系统进程、线程的有关概念，加深对 Windows 线程的理解。
- (2) 了解互斥体对象，利用互斥与同步操作编写生产者-消费者问题的并发程序，加深对 P（即 semWait）、V（即 semSignal）原语以及利用 P、V 原语进行进程间同步与互斥操作的理解。

三. 实验内容

1. 实验步骤和要求:

- 步骤1:** 创建一个“Win32 Consol Application”工程，然后拷贝清单 5-1 中的程序，编译成可执行文件。
- 步骤2:** 在“命令提示符”窗口运行步骤 1 中生成的可执行文件，列出运行结果。
- 步骤3:** 仔细阅读源程序，找出创建线程的 WINDOWS API 函数，回答下列问题：线程的第一个执行函数是什么（从哪里开始执行）？它位于创建线程的API 函数的第几个参数中？
- 步骤4:** 修改清单 5-1 中的程序，调整生产者线程和消费者线程的个数，使得消费者数目大与生产者，看看结果有何不同。察看运行结果，从中你可以得出什么结论？
- 步骤5:** 修改清单 5-1 中的程序，按程序注释中的说明修改信号量 EmptySemaphore 的初始化方法，看看结果有何不同。
- 步骤6:** 根据步骤 4 的结果，并查看MSDN，回答下列问题：
- ①CreateMutex 中有几个参数，各代表什么含义。
 - ②CreateSemaphore 中有几个参数，各代表什么含义，信号量的初值在第几参数中。
 - 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。
 - ④CreateMutex 能用 CreateSemaphore 替代吗？尝试修改程序 5-1，将信号量 Mutex完全用CreateSemaphore 及相关函数实现。写出要修改的语句。

2. 代码:

主要代码:

//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;
}
```

```

//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;
}
//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4-1修改部分:

```
const unsigned short CONSUMERS_COUNT = 5; //消费者的个数
```

实验4-2修改部分:

```
EmptySemaphore = CreateSemaphore(NULL, 0, SIZE_OF_BUFFER-1, NULL);
```

四．实验结果与分析

未修改前:


```
D:\CompileSoftware\Dev-C++5.9.2+便携版\OS\test4\test4-1\main.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

Producing 4 ... Succeed
Appending a product ... Succeed
0: 3 <-- 生产 <-- 消费
1: 4
Taking a product ... Succeed
0: 0 <-- 生产
```

实验4-1:

```
D:\CompileSoftware\Dev-C++5.9.2+便携版\OS\test4\test4-1\main.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

Producing 4 ... Succeed
Appending a product ... Succeed
0: 3 <-- 生产 <-- 消费
1: 4
```

原因：结果呈现跟未修改前类似，两者呈现的效果都是生产者生产一个产品，消费者消费一个产品，所以修改消费者的数量是不影响实验结果的。

实验4-2:

原因：将信号量如上修改，结果无输出。原因是将buff空信号量修改为实验要求，那么生产者以为一开始的buff是满的，但是其实是空的，从而生产者就处于阻塞状态，也没有生产产品，与此同时，消费者这里以为生产者生产了产品，但是没有拿到产品，也处于阻塞状态，从而控制台展示的效果就是什么都没有输出。

问题一：CreateMutex 中有几个参数，各代表什么含义。

共有3个参数；

①指向安全属性的指针

- ②初始化互斥对象的所有者
- ③指向互斥对象名的指针。

问题二：CreateSemaphore中有几个参数，各代表什么含义，信号量的初值在第几参数中。共有4个参数，

- ①结构体指针
- ②信号量对象的初始计数
- ③信号量对象的最大计数
- ④信号量对象的名称。

信号量的初值为第2个参数中。

问题三：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。

程序中 P、V 原语所对应的实际 Windows API 函数是WaitForSingleObject()、ReleaseMutex() 和ReleaseSemaphore()。

- ①ReleaseMutex(Mutex);
- ②ReleaseSemaphore(FullSemaphore, 1, NULL);
- ③ReleaseSemaphore(EmptySemaphore, 1, NULL);
- ④WaitForSingleObject(EmptySemaphore, INFINITE);
- ⑤WaitForSingleObject(Mutex, INFINITE);

问题四：CreateMutex能用CreateSemaphore替代吗？尝试修改程序 5-1，将信号量Mutex完全用CreateSemaphore及相关函数实现。写出要修改的语句。

可以使用CreateSemaphore 来替代CreateMutex。因为达到的效果一致。

- ①Mutex=CreateMutex(NULL, FALSE, NULL) --> Mutex=CreateSemaphore(NULL, 1, 1, NULL);
- ②ReleaseMutex(Mutex) --> ReleaseSemaphore(Mutex, 1, NULL);

五. 小结与心得体会

通过本实验加深了我对Windows线程的理解。了解了互斥体对象，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。

实验五 内存管理

一. 实验题目

内存管理

二. 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。

三. 实验内容

1. 实验背景知识

优化所拥有的内存的办法。

- ①分页过程
- ②内存共享
- ③未分页合并内存与分页合并内存
- ④提高分页性能
- ⑤Windows虚拟内存

2. 实验步骤

步骤 1: 创建一个“Win32 Consol Application”工程，然后拷贝清单 6-1 中的程序，编译成可执行文件，在弹出的窗口选择 - linker settings - other linker options 中添加-lshlwapi；把报错位置的 (DWORD) 改成 (DWORD64)。

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。

3. 实验任务：

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：_____

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。

将系统当前的自由区（free）虚拟地址空间按表 6-3 格式记录。

表 6-3 实验记录，例如：

将系统当前的虚拟地址空间按表 6-4 格式记录。

表 6-4 实验记录

地址	大小	虚拟地址 空间类型	访问权限	描述
		committed		

4. 代码:

```
// 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) << (DWORD64) pBlock
                << "-"
                << std :: hex << std :: setw(8) << (DWORD64) 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个块
    pBlock = pEnd;
    }
}
}

```

四. 实验结果与分析

注意：由于数据量偏大，这里仅展示部分数据

虚拟内存每页容量：4.00 KB

最小应用地址：0x00010000

最大应用地址：0x7fffffffeffff

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：16GB
理论上每个 Windows 应用程序可以独占的最大存储空间是：19.99GB

虚拟内存的自由区(free)：

地址	大小	虚拟地址空间类型	访问权限	描述
00011000-00020000	(60.0 KB)	Free	NOACCESS	
00021000-00030000	(60.0 KB)	Free	NOACCESS	
0004f000-00050000	(4.00 KB)	Free	NOACCESS	
00054000-00060000	(48.0 KB)	Free	NOACCESS	
00062000-00070000	(56.0 KB)	Free	NOACCESS	
000a1000-000b0000	(60.0 KB)	Free	NOACCESS	
000b3000-000c0000	(52.0 KB)	Free	NOACCESS	

虚拟内存的已调配区(committed)：

地址	大小	虚拟地址空间类型	访问权限	描述
00010000-00011000	(4.00 KB)	Committed	READONLY	Mapped
00020000-00021000	(4.00 KB)	Committed	READONLY	Mapped
00030000-0004f000	(124 KB)	Committed	READONLY	Mapped
00050000-00054000	(16.0 KB)	Committed	READONLY	Mapped
00060000-00062000	(8.00 KB)	Committed	READWRITE	Private
00070000-000a1000	(196.0 KB)	Committed	READONLY	Mapped
000b0000-000b3000	(12.0 KB)	Committed	READONLY	Mapped

虚拟内存的保留区(reserved)：

地址	大小	虚拟地址空间类型	访问权限	描述
000c2000-00122000	(384.0 KB)	Reserved	READONLY	Private
00200000-002f1000	(964.0 KB)	Reserved	READONLY	Private
002f8000-00400000	(1.03 MB)	Reserved	READONLY	Private
005e0000-007d9000	(1.97 MB)	Reserved	READONLY	Private
007e1000-00842000	(388 KB)	Reserved	READONLY	Private
008be000-009a0000	(904 KB)	Reserved	READONLY	Private
7ff4fdec5000-7ff4fdfc0000	(292 KB)	Reserved	READONLY	Mapped

分析：

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。

五. 小结与心得体会

了解Windows的内存结构和虚拟内存管理对于操作系统内存管理和虚拟存储管理的理论知识都是非常重要的。它能够帮助我们更好地理解计算机系统中内存的使用和管理方式, 并且能够提高我们在开发过程中对内存问题的敏感度和处理能力。

实验六 银行家算法的模拟与实现

一. 实验题目

银行家算法的模拟与实现

二. 实验目的

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

三. 总体设计

1. 背景知识

- ①**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。
- ②**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。
- ③**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。
- ④**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. 实验原理

本实验采用的是银行家算法，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。

3. 实验步骤

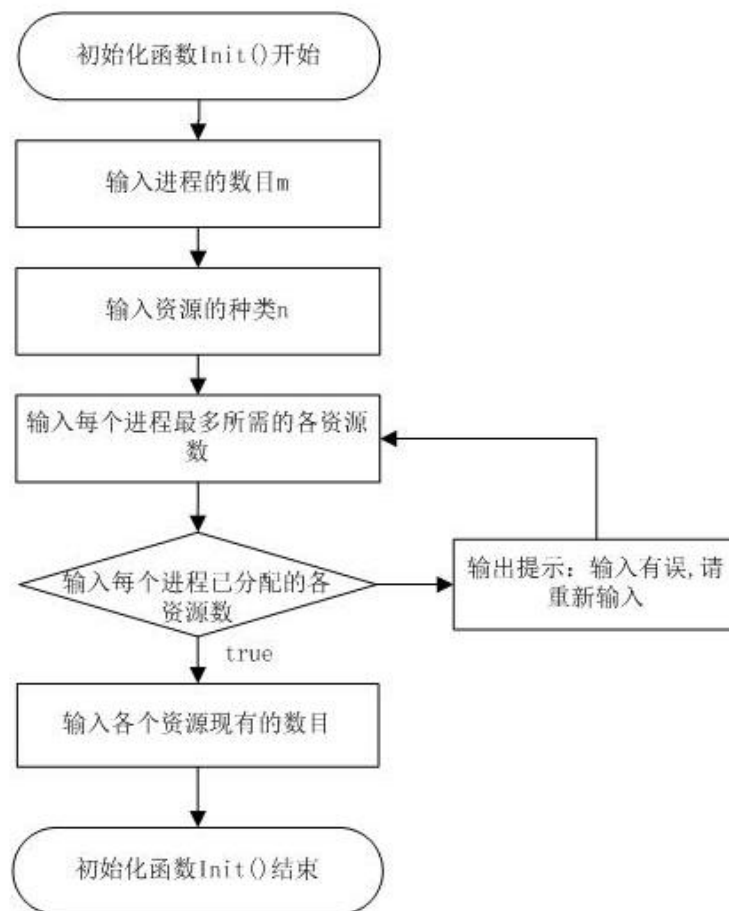
- ①检查该进程所需要的资源是否已超过它所宣布的最大值。
- ②检查系统当前是否有足够资源满足该进程的请求。
- ③系统试探着将资源分配给该进程，得到一个新状态。
- ④执行安全性算法，若该新状态是安全的，则分配完成；若新状态是不安全的，则恢复原状态，阻塞该进程。

四. 详细设计

1. 数据结构

数据结构	表示含义	示例说明
n	系统中的进程个数	
m	系统中的资源类数	
$Available(m)$	现有资源向量	$Available(j)=k$ 表示 k 个未分配的 j 类资源
$Max(n, m)$	资源最大申请量矩阵	$Max(i, j)=k$ 表示第 i 个进程在运行过程中对第 j 类资源的最大申请量为 k
$Allocation(n, m)$	资源分配矩阵	$Allocation(i, j)=k$ 表示进程 i 已占有 k 个 j 类资源
$Need(n, m)$	进程以后还需要的资源矩阵	$Need(i, j)=k$ 表示进程 i 以后还需要 k 个第 j 类资源 显然有 $Need[i, j]=Max[i, j]-Allocation[i, j]$
$Request(n, m)$	进程申请资源矩阵	$Request(i, j)=k$ 表示进程 i 申请 k 个第 j 类资源

2. 流程图



3. 关键代码

①尝试分配资源

```
for(int j=0;j<N;j++)
{
    Available[j]=Available[j]-Request[j];
    Allocation[i][j]=Allocation[i][j]+Request[j];
    Need[i][j]=Need[i][j]-Request[j];
}
```

②分配资源失败，资源返还

```
for(int j=0; j<N; j++)
{
    Available[j] = Available[j] + Request[j];
    Allocation[i][j] = Allocation[i][j] - Request[j];
    Need[i][j] = Need[i][j] + Request[j];
}
```

③安全性序列算法

```
for (int s=0; s<M; s++)
{
    for(i=0;i<M;i++)
    {
        apply=0;
        for(j=0;j<N;j++)
        {
            if(Finish[i]==False && Need[i][j]<=Work[j])
            {
                apply++;
                //直到每类资源尚需数都小于系统可利用资源数才可分配
                if(apply==N)
                {
                    for (m=0;m<N;m++)
                        Work[m]=Work[m]+Allocation[i][m];
                    Finish[i]=True;
                    Security[k++]=i;
                }
            }
        }
    }
    if (k == M)
        break;
}
```

④判断银行家算法的前两条件是否成立

```
for (j=0;j<N;j++)
{
    if(Request[j]>Need[i][j])//判断申请是否大于需求，若大于则出错
    {
        printf("进程P%d申请的资源大于它需要的资源",i);
        printf("分配不合理，不予分配！\n");
        flag = False;
        break;
    }
}
```

```

    }
    else
    {
        if (Request[j]>Available[j])//判断申请是否大于当前可分配资源，若大于则出错
        {
            printf("进程%d申请的资源大于系统现在可利用的资源", i);
            printf("\n");
            printf("系统尚无足够资源，不予分配!\n");
            flag = False;
            break;
        }
    }
}

```

五. 实验结果与分析

实验6-1:

```

-----银行家算法实现-----

请输入可用资源种类数m为:3
资源0的名称为:p
资源p的可用个数为:3
资源1的名称为:q
资源q的可用个数为:3
资源2的名称为:r
资源r的可用个数为:2

请输入进程的数量n为:4
请输入各进程的最大需求矩阵的值[Max] (提示: n×m的形式, 中间以空格进行分隔):
7 2 3
5 3 2
2 1 1
0 4 3
请输入各进程已分配的资源数[Allocation] (提示: n×m的形式, 中间以空格进行分隔):
1 2 1
1 1 0
0 0 1
0 0 1

-----

系统当前的资源分配情况如下:

```

进程名	Max			Allocation			Need			Available		
	p	q	r	p	q	r	p	q	r	p	q	r
P0	7	2	3	1	2	1	6	0	2	3	3	2
P1	5	3	2	1	1	0	4	2	2			
P2	2	1	1	0	0	1	2	1	0			
P3	0	4	3	0	0	1	0	4	2			

```

系统不安全

```

由图可知, 先进行p2进程的分配, 此时资源进行了回收, 即Available=[3, 3, 3], 进行下一轮的分配, 此时无论是p0, p2还是p3进程, 对其进行分配都是不安全的, 因为对p0来说, p资源数不够分配, 对于p1来说, p资源不够分配, 对于p3来说, q资源不够分配, 所以是不安全序列。

实验6-2:

```
-----银行家算法实现-----

请输入可用资源种类数m为:3
资源0的名称为:p
资源p的可用个数为:3
资源1的名称为:q
资源q的可用个数为:3
资源2的名称为:r
资源r的可用个数为:2

请输入进程的数量n为:4
请输入各进程的最大需求矩阵的值[Max] (提示: n×m的形式, 中间以空格进行分隔):
6 2 3
4 2 3
2 1 1
1 4 3
请输入各进程已分配的资源数[Allocation] (提示: n×m的形式, 中间以空格进行分隔):
4 0 1
2 1 1
1 0 1
0 2 1

-----

系统当前的资源分配情况如下:

```

	Max			Allocation			Need			Available		
进程名	p	q	r	p	q	r	p	q	r	p	q	r
P0	6	2	3	4	0	1	2	2	2	3	3	2
P1	4	2	3	2	1	1	2	1	2			
P2	2	1	1	1	0	1	1	1	0			
P3	1	4	3	0	2	1	1	2	2			

```

系统安全!
存在一个安全序列:P0 --> P1 --> P2 --> P3

```

由图可知, 先对p0进行分配, 此时资源进行了回收, 即Available=[7, 3, 3], 接下来对p1进行分配, 资源是够分配的, 资源进行回收, 此时Available=[9, 4, 4], 接下来再对p2进行分配, 资源是也够分配的, 资源进行回收, 此时Available=[10, 4, 1], 最后对p3进行分配, 资源是够分配的, 资源进行回收, 此时Available=[10, 6, 2], 所以是安全序列, 得出的安全序列为p0 -> p1 -> p2 -> p3。

六. 小结与心得体会

通过学习银行家算法, 增加了耐心和细心, 同时让我明白要善于思考问题并且勤于练习。同时还要注意理论与实践结合, 在理论基础上进行实践操作, 加深了对该算法的理解和掌握。

实验七 磁盘调度算法的模拟与实现

一. 实验题目

磁盘调度算法的模拟与实现

二. 实验目的

- (1) 了解磁盘结构以及磁盘上数据的组织方式。
- (2) 掌握磁盘访问时间的计算方式。
- (3) 掌握常用磁盘调度算法及其相关特性。

三. 总体设计

1. 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。

①**先进先出算法**：按访问请求到达的先后次序进行调度。

②**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。

③**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。

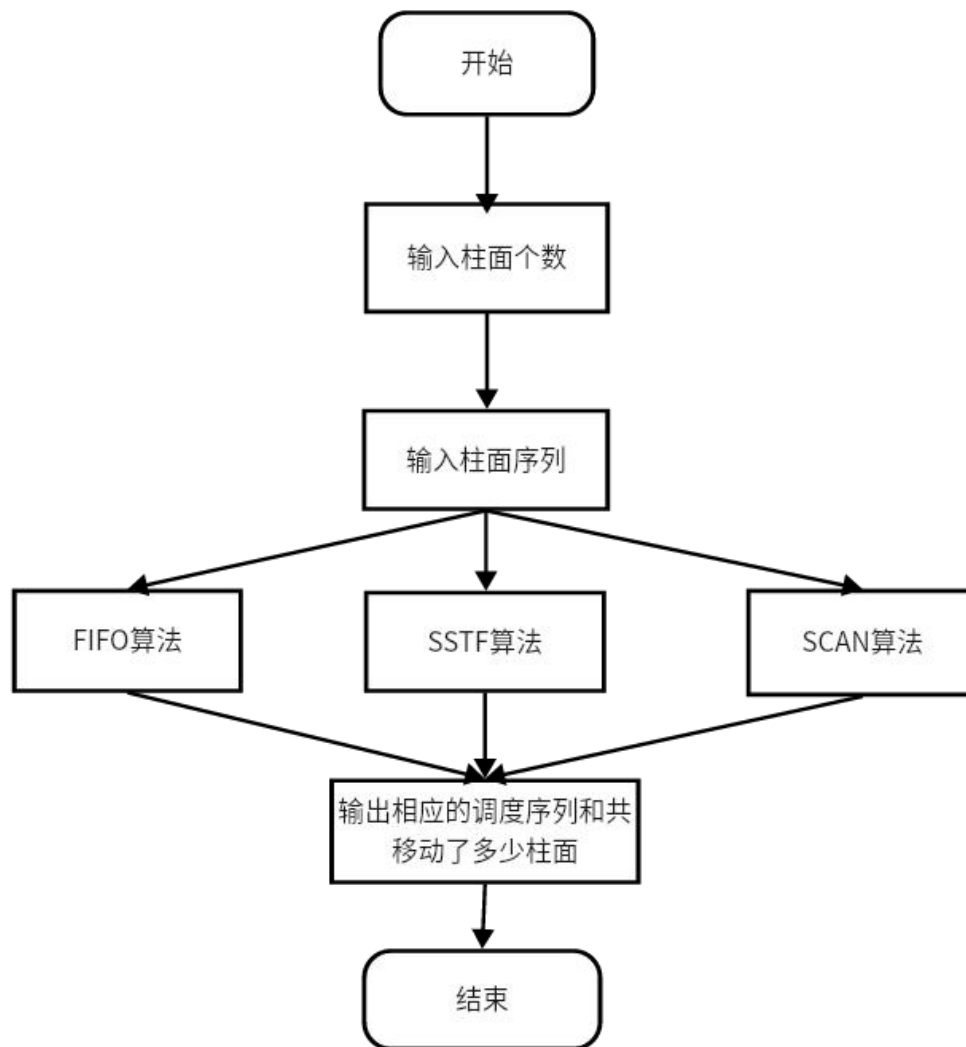
④**C-SCAN（循环扫描）算法**：在磁盘调度时，把扫描限定在一个方向，当沿某个方向访问到最后一个磁道时，磁头臂返回到磁盘的另一端，并再次开始扫描。

2. 实验步骤

- ①根据实验要求依次输入柱面的个数以及其序列
- ②设计先进先出调度算法
- ③设计最短服务时间优先调度算法
- ④设计电梯算法

四. 详细设计

1. 程序流程图:



2. 实验代码:

①FCF0:

```
printf("柱面顺序: ");
for (i = 1; i < CN; i++)
    printf("%-4d", CO[i]);
printf("\n移动距离: ");
for (i = 1; i < CN; i++)
{
    int j = abs(CO[i] - CO[i - 1]);
    printf("%-4d", j);
    sum += j;
}
printf("\n一共移动了%d个柱面\n", sum);
```

②SSTF:

```
for (i = 0; i < CN; i++)
{
```

```

    copy[i] = CO[i];
}
for (i = 1; i < CN; i++)
{
    pos = i;
    min = abs(copy[i] - copy[i - 1]);
    for (j = i + 1; j < CN; j++)
    {
        if (abs(copy[j] - copy[i - 1]) < min)
        {
            min = abs(copy[j] - copy[i - 1]);
            pos = j;
        }
    }
    distance[i] = min;
    if (pos != i)
    {
        temp = copy[pos];
        copy[pos] = copy[i];
        copy[i] = temp;
    }
}
printf("柱面顺序: ");
for (i = 1; i < CN; i++)
    printf("%-4d", copy[i]);
printf("\n移动距离: ");
for (i = 1; i < CN; i++)
{
    j = distance[i];
    printf("%-4d", j);
    distance[0] += j;
}
printf("\n一共移动了%d个柱面\n", distance[0]);

```

③SCAN:

```

if (choice == 1)
{
    j = 0;
    for (i = 0; i < CN; i++)
        if (CO[i] >= CO[0])
            copy[j++] = CO[i];

    int n = j - 1;
    bool flag = true;
    while (n > 1 && flag)
    {
        flag = false;
        for (i = 0; i < n - 1; i++)
            if (copy[i] > copy[i + 1])
            {
                int temp = copy[i];
                copy[i] = copy[i + 1];
                copy[i + 1] = temp;
            }
    }
}

```

```

        flag = true;
    }
    n--;
}
int m = j;
for (i = 1; i < CN; i++)
    if (CO[i] < CO[0])
        copy[j++] = CO[i];
n = j;
flag = true;
while (n >= m && flag)
{
    flag = false;
    for (i = m; i < n - 1; i++)
        if (copy[i] < copy[i + 1])
        {
            int temp = copy[i];
            copy[i] = copy[i + 1];
            copy[i + 1] = temp;
            flag = true;
        }
    n--;
}
}

```

五. 实验结果与分析

D:\CompileSoftware\Dev-C++ 5.9.2+便携版\OS\test7\main.exe

输入柱面个数: 10
 输入柱面序列: 125 86 147 91 177 94 150 102 175 130

-----FCFS-----
 柱面顺序: 86 147 91 177 94 150 102 175 130
 移动距离: 39 61 56 86 83 56 48 73 45
 一共移动了547个柱面

-----SSTF-----
 柱面顺序: 130 147 150 175 177 102 94 91 86
 移动距离: 5 17 3 25 2 75 8 3 5
 一共移动了143个柱面

-----SCAN-----
 请选择磁头移动方向 1. 由外向里 2. 由里向外 : 1
 柱面顺序: 147 150 175 177 130 102 94 91 86
 移动距离: 22 3 25 2 47 28 8 3 5
 一共移动了143个柱面

由图可知，输入柱面顺序时，第一个为柱头当前位置；

①对于先进先出的磁盘调度算法来说，实质上就是根据输入次序依次计算到达下一柱面的距离，然后进行相加，所以第一个需要到达的柱面是86，所以移动距离为125-86=39，

此时柱头在86处，移动到下一柱面147，则需要移动距离 $147-86=61$ ，依次类推，得到共移动547个柱面的距离；

②对于最短时间优先算法来说，起初在柱头在125，通过遍历此柱头到各柱面的距离，得出最小距离的那个柱面就是需要下次移动到的柱面，由输入序列可知130是距离125最近的，从而移动距离为 $130-125=5$ ，此时柱头在130处，再通过遍历发现距离130最近的为147，从而移动距离为 $147-130=17$ ，依次类推，得出最后的移动距离为143；

③对于电梯算法来说，则要先确定磁头移动的方向，是由外向里还是由里向外，这个根据输入来进行判定，这里可以根据数组的下标来确定是向里还是向外，由图可知，为由外向里，先将柱面顺序进行一次排序，然后根据方向来依次移动柱头，由于一开始柱头位置在125，且是由外向里，所以先到达147，从而移动距离为 $147-125=22$ ，此时柱头在147，向150移动，移动距离为 $150-147=3$ ，由于SCAN算法和C-SCAN算法相似，所以这里采用的是SCAN算法，从而到达边缘时的距离直接就等于两个柱面之间的距离，无需计算到达最边缘的距离，从而得出最终距离为143个柱面。

1. 先来先服务FCFS

此算法的优点是公平、简单，且每个进程的请求都能依次得到处理，不会出现某进程的请求长期得不到满足的情况。但此算法由于未对寻道进行优化，致使平均寻道时间可能较长。FCFS算法仅适用于请求磁盘I/O的进程数目较少的场合。

2. 最短寻道时间优先SSTF

该算法选择这样的进程，其要求访问的磁道与当前磁头所在的磁道距离最近，以使每次的寻道时间最短，该算法可以得到比较好的吞吐量，但却不能保证平均寻道时间最短。其缺点是对用户的服务请求的响应机会不是均等的，因而导致响应时间的变化幅度很大。在服务请求很多的情况下，对内外边缘磁道的请求将会无限期的被延迟，有些请求的响应时间将不可预期。

3. 循环扫描算法CSCAN。

循环扫描算法是对扫描算法的改进。如果对磁道的访问请求是均匀分布的，当磁头到达磁盘的一端，并反向运动时落在磁头之后的访问请求相对较少。这是由于这些磁道刚被处理，而磁盘另一端的请求密度相当高，且这些访问请求等待的时间较长，为了解决这种情况，循环扫描算法规定磁头单向移动。

六. 小结与心得体会

通过本次实验，深刻了解了磁盘调度算法，并学会使用代码去实现这几种算法，同时也在实践中深深体会到不同算法之间的优缺点，以及异同之处，以及不同调度算法的使用场景，收获颇多！

实验八* 虚拟内存系统的页面置换算法模拟

一. 实验题目

虚拟内存系统的页面置换算法模拟

二. 实验目的

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

三. 总体设计

1. 背景知识

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

教材给出的常用的页面置换算法包括：

- ①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。
- ②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。
- ③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
- ④时钟算法(Clock)：也称最近未使用算法(NRU, Not Recently Used)，它是 LRU 和 FIFO 的折衷。

2. 设计步骤

- (1) 每个页面中可存放10条指令，分配给一作业的内存块（页框）数为 4。
- (2) 用C语言模拟一作业的执行过程，该作业共有320 条指令，即它的地址空间为32页，目前它的所有页都还未调入内存。在模拟过程中，如果所访问的指令已在内存，则显示其物理地址，并转下一条指令。如果所访问的指令还未装入内存，则发生缺页，此时须记录缺页的次数，并将相应页调入内存；如果4个内存块中均已装入该作业，则需进行页面置换；最后显示其物理地址，并转下一条指令。在所有 320 条指令执行完毕后，计算并显示作业运行过程中发生的缺页率。
- (3) 置换算法分别考虑 OPT、FIFO 和 LRU 算法。
- (4) 测试用例设计：通过随机数产生一个指令序列，共 320 条指令。
 - ① 50%的指令是顺序执行的；
 - ② 25%的指令是均匀分布在前地址部分；
 - ③ 25%的指令是均匀分布在后地址部分；具体的实施方法是：
 - ① 在[0, 319]的指令地址之间随机选取一起点 m；

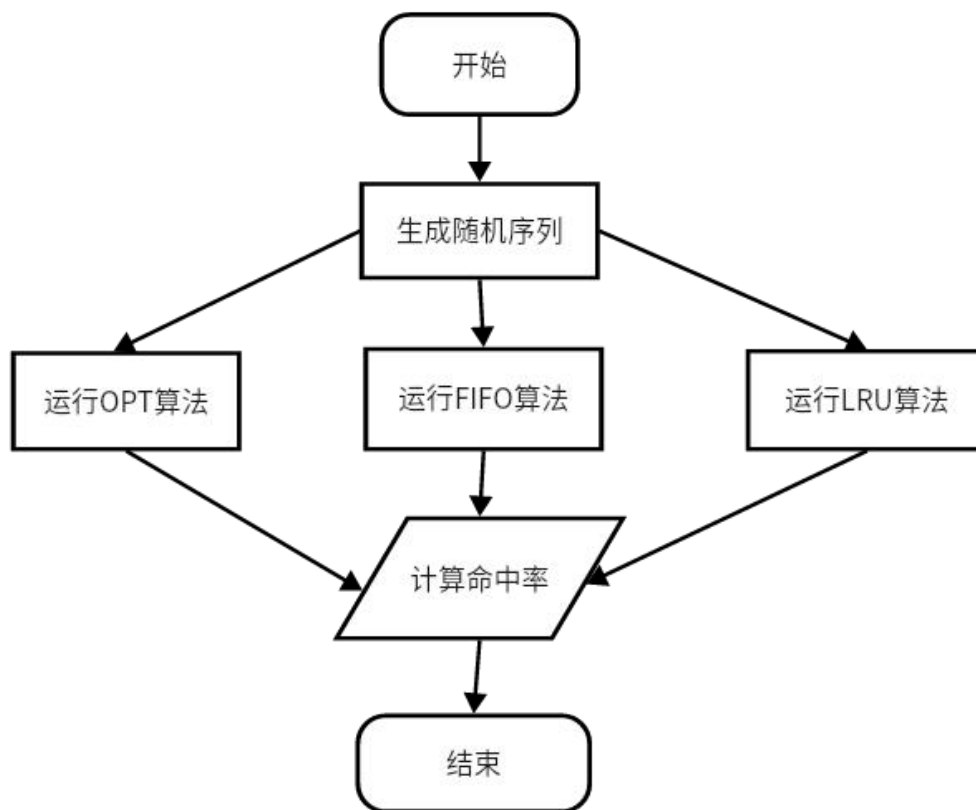
- ②顺序执行一条指令，即执行地址为 $m+1$ 的指令；
- ③在前地址 $[0, m+1]$ 中随机选取一条指令并执行，该指令的地址为 $m1$ ；
- ④顺序执行一条指令，其地址为 $m1+1$ ；
- ⑤在后地址 $[m1+2, 319]$ 中随机选取一条指令并执行；

3. 模块设计

- ①数据结构的设计：页框、指令的数据类型等。
- ②指令序列的生成。
- ③模拟实现OPT，FIFO，LRU算法。

四．详细设计

1. 流程图



2. 关键代码

①随机指令序列：

```

srand((unsigned)time(NULL));
int commd = 0;
while(commd < COMMD_NUM)
{
    //范围是[0, 319]

```

```

int m = rand()%COMMD_NUM;
order_commd.push_back(m);
commd++;
if(commd >= COMMD_NUM)
    break;
if(m == COMMD_NUM-1)
    continue;
order_commd.push_back(m+1);
commd++;
if(commd >= COMMD_NUM)
    break;

//范围是[0, m+1]
m = rand()%(m+2);
order_commd.push_back(m);
commd++;
if(commd >= COMMD_NUM)
    break;
if(m == COMMD_NUM-1)
    continue;
order_commd.push_back(m+1);
commd++;
if(commd >= COMMD_NUM)
    break;

//范围是[m+2, 319]
m = rand()%(COMMD_NUM -m -2) + m+2;
order_commd.push_back(m);
commd++;
if(commd >= COMMD_NUM)
    break;
}

```

②OPT算法:

```

if(!flag)    //没找到, 发生缺页
{
    err++;
    if(memory_page_num < MEM_PAHE_NUM)
    {
        memory[memory_page_num++].index = order_page[i];
    }
    else    //利用置换算法开始置换
    {
        int far[MEM_PAHE_NUM] ;//存储距离
        int furthest = 0;//找最远的位置;
        for(int j=0;j<MEM_PAHE_NUM;j++) //初始化为最大值
            far[j] = MAX_FAR;
        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;
        }
    }
    for(int j=0;j<memory_page_num;j++) //找最远的位置
        if(far[furthest] < far[j])
            furthest = j;
    //找到之后，置换
    memory[furthest].index = order_page[i];
}
}

```

③FIFO算法:

```

if(!flag) //没找到,发生缺页
{
    err++;
    if(memory_page_num < MEM_PAHE_NUM)
    {
        memory[memory_page_num++].index = order_page[i];
    }
    else //利用置换算法开始置换
    {
        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];
    }
}

```

④LRU算法:

```

if(!flag) //没找到,发生缺页
{
    int minn=0; //标记最小的
    err++;
    if(memory_page_num < MEM_PAHE_NUM)
    {
        memory[memory_page_num].time = i; //留下时间戳
        memory[memory_page_num++].index = order_page[i];
    }
    else //利用置换算法开始置换
    {
        for(int j=0;j<memory_page_num;j++)
        {
            if(memory[minn].time > memory[j].time)
            {
                minn = j;
            }
        }
    }
}

```

```

        }
    }
    memory[minn].time = i;
    memory[minn].index = order_page[i];
}
}

```

⑥ 重复上述步骤①~⑤，直到执行 320 条指令。

五. 实验结果与分析

1. 实验结果:

```

OPT = 0.506250 FIFO = 0.559375 LRU = 0.565625
OPT = 0.512500 FIFO = 0.553125 LRU = 0.556250
OPT = 0.503125 FIFO = 0.546875 LRU = 0.543750

```

2. 实验分析

①增加分配给作业的内存块数，将会对作业运行过程中的缺页率会降低

②FIFO 算法将页面进入内存后的时间长短作为淘汰依据，而LRU则是以使用频率作为淘汰依据，一般情况下，LRU之所以比FIFO具有更好的性能是因为长时间驻留并不一定代表不使用，FIFO的淘汰机制有可能增加了缺页率。

六. 小结与心得体会

通过本次实验，更加清晰的了解了FIFO，OPT以及LRU置换算法的实现机制，在代码实现的过程中确实有很多令人头疼的时刻，但是沉下心来慢慢搞，最后还是实现了，同时也掌握了随机序列的生成，收获了很多！

实验九* 基于信号量机制的并发程序设计

一. 实验题目

基于信号量机制的并发程序设计（读者/写者问题）

二. 实验目的

- ①回顾操作系统进程、线程的有关概念，针对经典的同步、互斥、死锁与饥饿问题进行并发程序设计。
- ②了解互斥体对象，利用互斥与同步操作编写读者-写者问题的并发程序，加深对 P（即semWait）、V（即semSignal）原语以及利用P、V原语进行进程间同步与互斥操作的理解。
- 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。

三. 总体设计

1. 背景知识

- ①相关并发控制的 Windows API 函数参考实验四。
- ②UNIX/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(IPCresource)。提供给用户的 IPC 资源是通过一组系统调用实现的。任务二中将使用 IPC 中提供的semctl()等信号量系统调用。

2. 实验步骤

(1)限制条件设置

- ①任意数量的读进程可同时读这个文件。
- ②一次只有一个写进程可以写文件
- ③若写进程正在写文件，则禁止任何读进程读文件

(2)读者部分代码实现

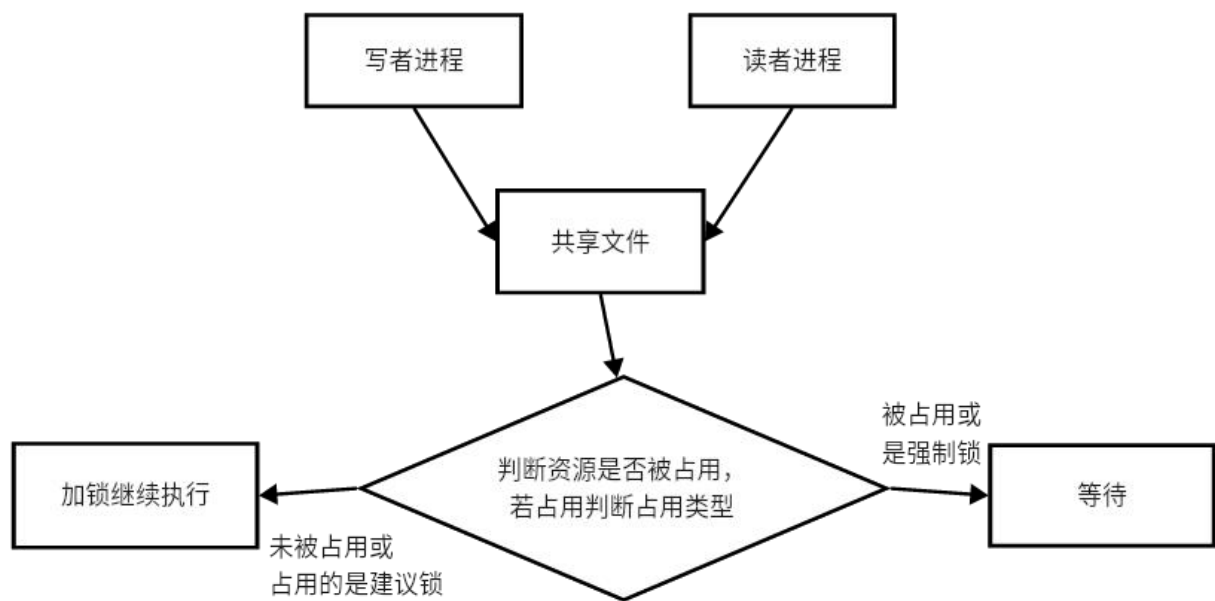
(3)写者部分代码实现

3. 算法思路

读者和写者是互斥的，写者和写者是互斥的，读者和读者不互斥；两类进程，一种是写者，另一种是读者。写者很好实现，因为它和其他任何进程都是互斥的，因此对每一个写者进程都给一个互斥信号量的P、V操作即可；而读者进程的问题就较为复杂，它与写者进程是互斥的，而又与其他的读者进程是同步的，因此不能简单的利用P、V操作解决。

四. 详细设计

1. 流程图



2.关键代码

①读者

```

DWORD WINAPI reader(LPVOID lpPara)
{
    while(p_ccontinue)
    {
        WaitForSingleObject(XSemaphore, INFINITE); //semWait(x);
        readcount++;
        if(readcount == 1) //第一个读者来了
            WaitForSingleObject(WriteSemaphore, INFINITE); //semWait(wsem);
        ReleaseSemaphore(XSemaphore, 1, NULL); //semSignal(x);
        READUNIT();
        //阅读完毕
        WaitForSingleObject(XSemaphore, INFINITE); //semWait(x);
        readcount--;
        //无读者，释放资源
        if(readcount == 0)
        {
            ReleaseSemaphore(WriteSemaphore, 1, NULL); //semSignal(wsem);
            ReleaseSemaphore(XSemaphore, 1, NULL); //semSignal(x);
            Sleep(3000);
        }
    }
    return 0;
}
  
```

②写者

```

DWORD WINAPI writer(LPVOID lpPara)
{
    while(p_ccontinue)
  
```



```

    {
        WaitForSingleObject(WriteSemaphore, INFINITE); //semWait(wsem);
        WRITEUNIT();
        ReleaseSemaphore(WriteSemaphore, 1, NULL); //semSignal(wsem);
        Sleep(2000);
    }
    return 0;
}

```

五. 实验结果与分析

```

写者开始写:1
写者开始写:2
写者开始写:3
写者开始写:4
一个读者开始阅读:4
一个读者开始阅读:4一个读者开始阅读:4

写者开始写:5
写者开始写:6
一个读者开始阅读:6
一个读者开始阅读:6
一个读者开始阅读:6
写者开始写:7
写者开始写:8

```

只有写者进行写作时，信号量才会自增，对于读者来说是不会自增的，所以实验结果符合预期。

六. 小结与心得体会

由于在前面的实验已经多少接触了有关同步和互斥相关函数的调用，这次实验就轻松了一些，之前的同步与互斥是验证性实验，这次是设计性实验，所以在实操性来说，更能加深对同步互斥相关知识的理解与掌握。

实验十* 实现一个简单的 shell 命令行解释器

一. 实验题目

实现一个简单的 shell 命令行解释器

二. 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。

三. 总体设计

1. 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。以上相关的系统函数调用的说明请参见实验三的背景知识。

2. 实验要求

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Linux 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。

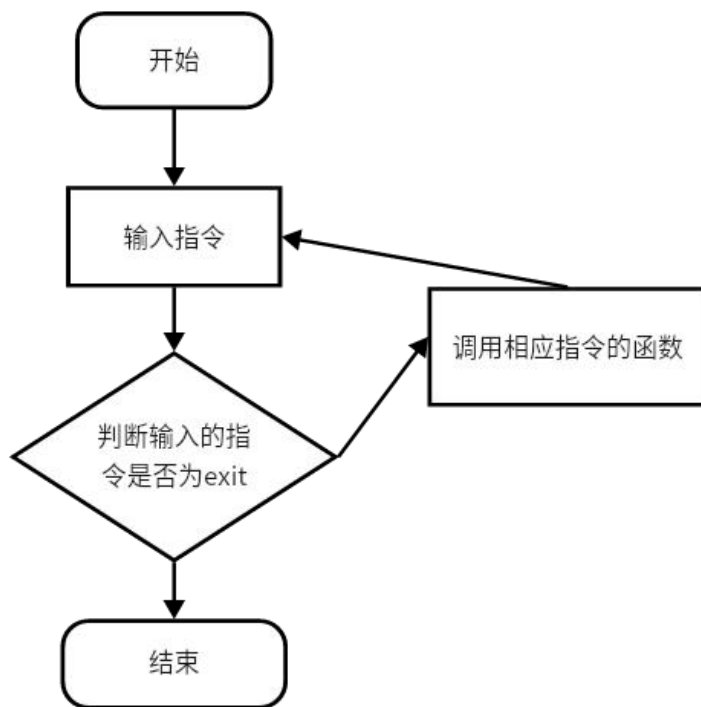
`quit, exit, bye` 退出 shell。

3. 设计思路

通过拆分指令，将其存放到数组里面，然后调用linux里面自带的指令函数，将相应的指令实现。

四. 详细设计

1. 流程图



2. 关键代码

Cd指令:

```

void exec_cd(int argcount, char arglist[100][256]) {
    if(argcount != 2) {
        printf("%s : command not found\n", arglist[0]);
        return;
    }
    chdir(arglist[1]);
}

```

Help指令:

```

void exec_help(int argcount, char arglist[100][256]) {
    printf("-----help-----\n");
    printf("cd -- <目录>更改当前的工作目录到另一个<目录>\n");
    printf("env -- 列出所有环境变量字符串的设置\n");
    printf("jobs -- 输出 shell 当前的一系列子进程\n");
    printf("ls -- 列出当前目录中所有的子目录和文件\n");
    printf("man -- 命令说明书\n");
    printf("touch -- 新增文件\n");
    printf("vim/vi -- 编辑文件\n");
    printf("cat -- 查看文件\n");
    printf("mkdir -- 创建目录\n");
    printf("rm -- 删除目录和文件\n");
    printf("mv -- 修改目录\n");
}

```

```

    printf("cp -- 拷贝目录\n");
    printf("find -- 搜索目录\n");
    printf("pwd -- 查看当前目录\n");
    printf("exit -- 退出 shell\n");
}

```

Jobs指令:

```

void exec_jobs(int argcount, char arglist[100][256]) {
    execlp("pstree", "-p", NULL);
}

```

指令拆分:

```

void explain_input(char *buf, int *argcount, char arglist[100][256]) {
    char *p = buf;
    char *q = buf;
    int number = 0;
    while(1) {
        if(p[0] == '\n')
            break;
        if(p[0] == ' ')
            p++;
        else {
            q = p;
            number = 0;
            while((q[0] != ' ') && (q[0] != '\n')) {
                number++;
                q++;
            }
            strncpy(arglist[*argcount], p, number+1);
            arglist[*argcount][number] = '\0';
            *argcount = *argcount + 1;
            p = q;
        }
    }
}

```

指令实现:

```

int find_command(char *command) {
    DIR* dp;
    struct dirent* dirp;
    char* path[] = {".", "/bin", "/user/bin", NULL};
    if(strncmp(command, ".", 2) == 0)
        command = command + 2;
    int i = 0;
    while (path[i] != NULL) {

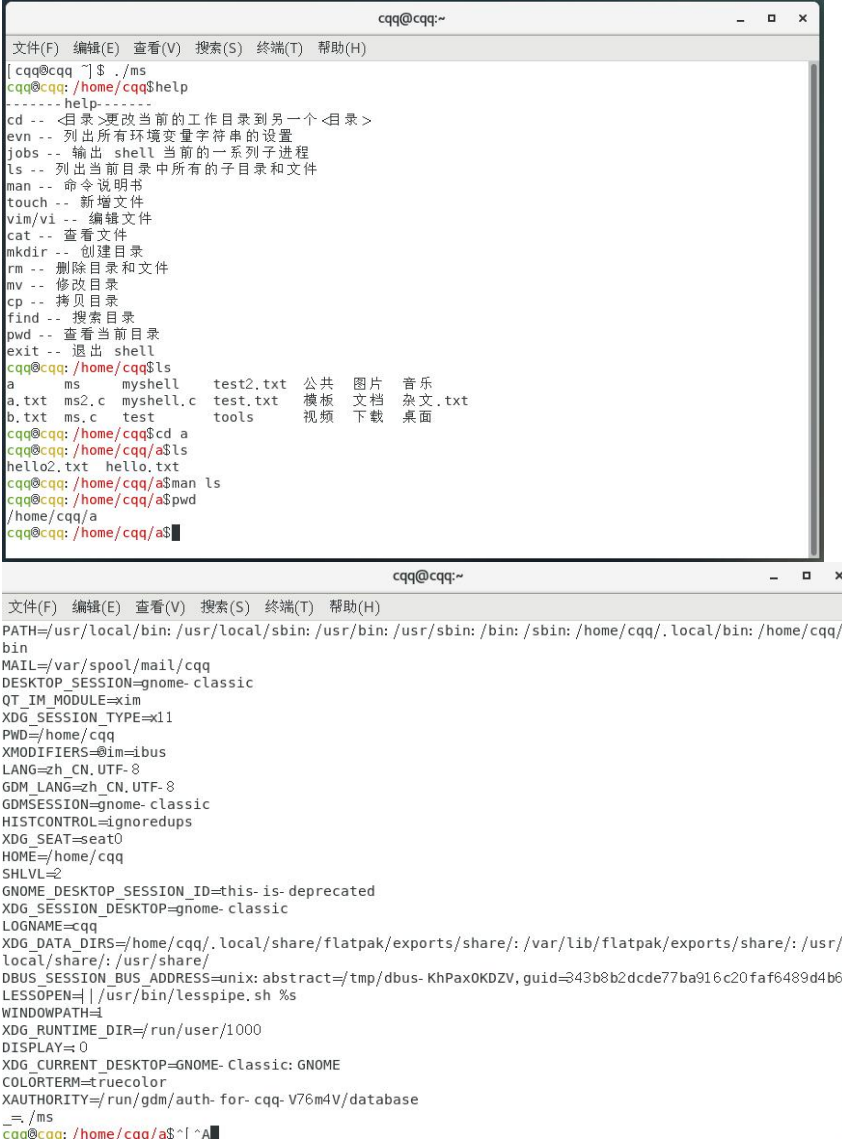
```

```

        if((dp = opendir(path[i])) == NULL)
            printf("can not open /bin \n");
        while((dirp = readdir(dp)) != NULL) {
            if(strcmp(dirp -> d_name, command) == 0) {
                closedir(dp);
                return 1;
            }
        }
        closedir(dp);
        i++;
    }
    return 0;
}

```

五. 实验结果与分析



The first screenshot shows the execution of a custom shell program. The user runs `./ms` in the `/home/cqq` directory. The program displays a help menu with various commands and their functions. The user then runs `ls`, and the program lists the contents of the current directory, including files like `ms`, `ms2.c`, `test2.txt`, and `公共`. The user then runs `cd a`, and the program changes the current directory to `/home/cqq/a`. The user then runs `ls`, and the program lists the contents of the new directory, including files like `hello2.txt` and `hello.txt`. The user then runs `pwd`, and the program displays the current directory path `/home/cqq/a`.

The second screenshot shows the output of the `env` command, displaying the environment variables for the user. The output includes variables such as `PATH`, `MAIL`, `DESKTOP_SESSION`, `QT_IM_MODULE`, `XDG_SESSION_TYPE`, `PWD`, `XMODIFIERS`, `LANG`, `GDM_LANG`, `GDMSESSION`, `HISTCONTROL`, `XDG_SEAT`, `HOME`, `SHLVL`, `GNOME_DESKTOP_SESSION_ID`, `XDG_SESSION_DESKTOP`, `LOGNAME`, `XDG_DATA_DIRS`, `DBUS_SESSION_BUS_ADDRESS`, `LESSOPEN`, `WINDOWPATH`, `XDG_RUNTIME_DIR`, `DISPLAY`, `XDG_CURRENT_DESKTOP`, `COLORTERM`, `XAUTHORITY`, and `_`.

实验结果符合预期。且在实验要求的基础上能实现更多额外的功能。
实验实现的本质其实就是调用linux自带的函数，进入到相应的路径执行相应的指令。

六. 小结与心得体会

由于之前操作系统老师布置过类似的shell命令的实验，有过相应的实验经验，对于本次实验，就会没有那么困难，反而会轻松点，当然在此期间还是有过小卡壳，但是最终还是解决了。通过本次实验，进一步掌握有关shell命令解释器的相关知识，为之后的linux进阶性的学习打上了一定的基础，收获颇多！

时间		设计内容简要记录
第 17 周	星期一	验证实验一、实验二，并完成相应的实验报告
	星期二	验证实验三、实验四，并完成相应的实验报告
	星期三	验证实验五并完成相应的实验报告
	星期四	设计实验六并完成相应实验报告
	星期五	设计实验七并完成相应实验报告

周末		设计实验八并完成相应实验报告
第18周	星期一	设计实验九并完成相应实验报告
	星期二	设计实验十并完成相应实验报告
	星期三	整理实验报告资料
	星期四	完善实验报告资料
	星期五	完善实验报告资料