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：** 21计科七班

**姓 名：** 罗玉婷

**学 号：** 2101020326

**指导教师：** 肖小聪

**时 间**： 17、18周

**地 点**： 逸夫楼416

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

1. **实验题目**

**实验一 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。

1. **实验内容**

**（1）编写基本的 Win32 Consol Application**

**步骤 1**：登录进入 Windows 系统，启动 VC++ 6.0。

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

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

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

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

**（2） 创建进程**

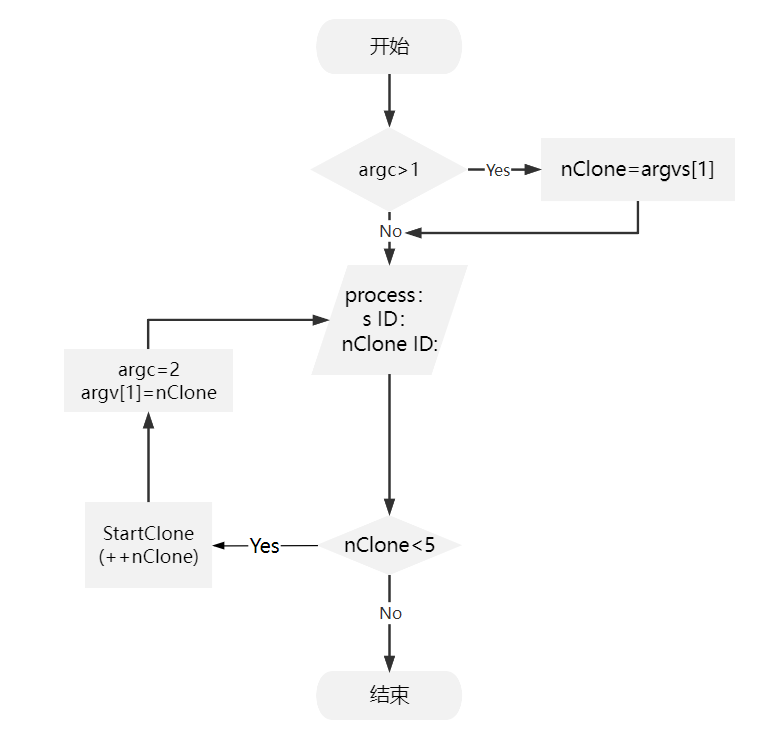
本实验显示了创建子进程的基本框架。该程序只是再一次地启动自身，显示它的系统进程 ID和它在进程列表中的位置。

**步骤 1**：创建一个“Win32 Consol Application”工程，然后拷贝清单 2-2 中的程序，编译成可执行4文件。

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

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

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



**（3） 父子进程的简单通信及终止进程**

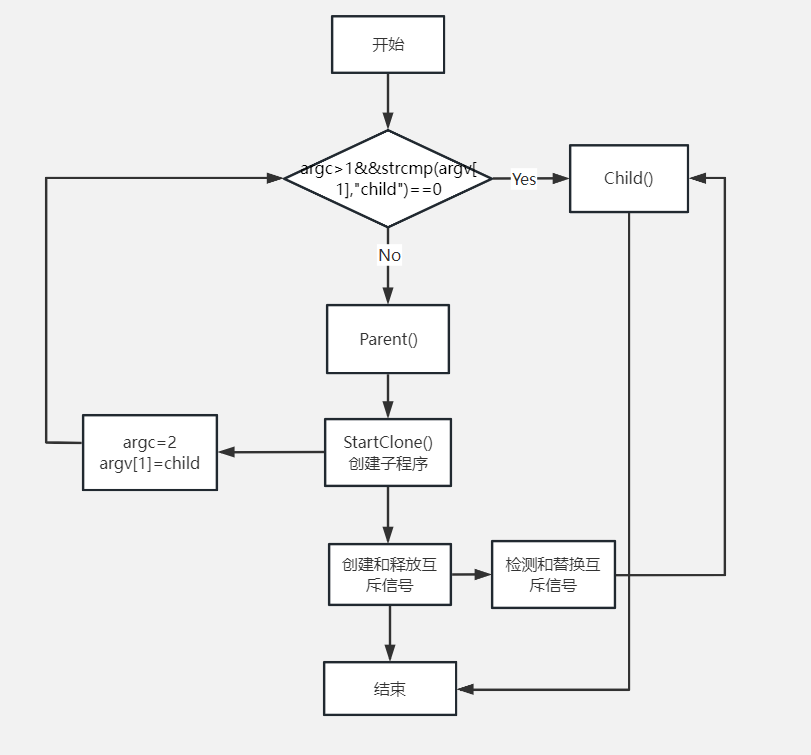
**步骤 1：**创建一个“Win32 Consol Application”工程，然后拷贝清单 2-3 中的程序，编译成可执行文件。

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

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

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

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

--3程序执行的大概描述：

从main()函数开始，首先判断argc的值（argc初始值默认为1），决定进行父进程还是子进程，因为argc不满足大于1，所以调用parent()函数，在执行parent（）函数过程中调用StartClone() ;然后通过sprintf(szCmdLine, “”%s\"child\" , szFilename)将argv[1]赋值child，后面满足条件后调用child()函数；由于设置了互斥信号，则只允许一个进程进行，所以只有当父进程释放互斥信号hMutexSuicide时，子进程检测获得才结束进程。

--4父子进程同步的大概描述：

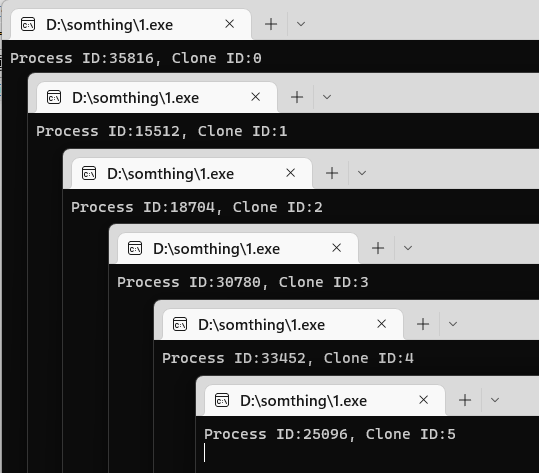
CreateMutex() 创建互斥体hMutexSuicide信号、 OpenMutex()打开互斥体、 ReleaseMutex()释放互斥体、WaitForSingleObject()检测Hhandle信号状态，通过这些只允许有一个状态被创建或者使用也就是信号量唯一，实现进程同步。

1. **实验结果与分析**

**（1）**在将void改为int后编译执行

****

1. 未修改运行

****

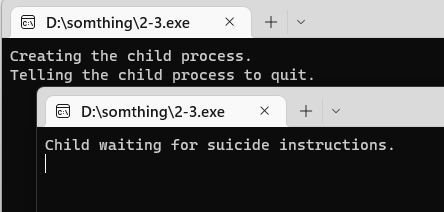
****

修改后运行：（由于关机了并未保存截图）

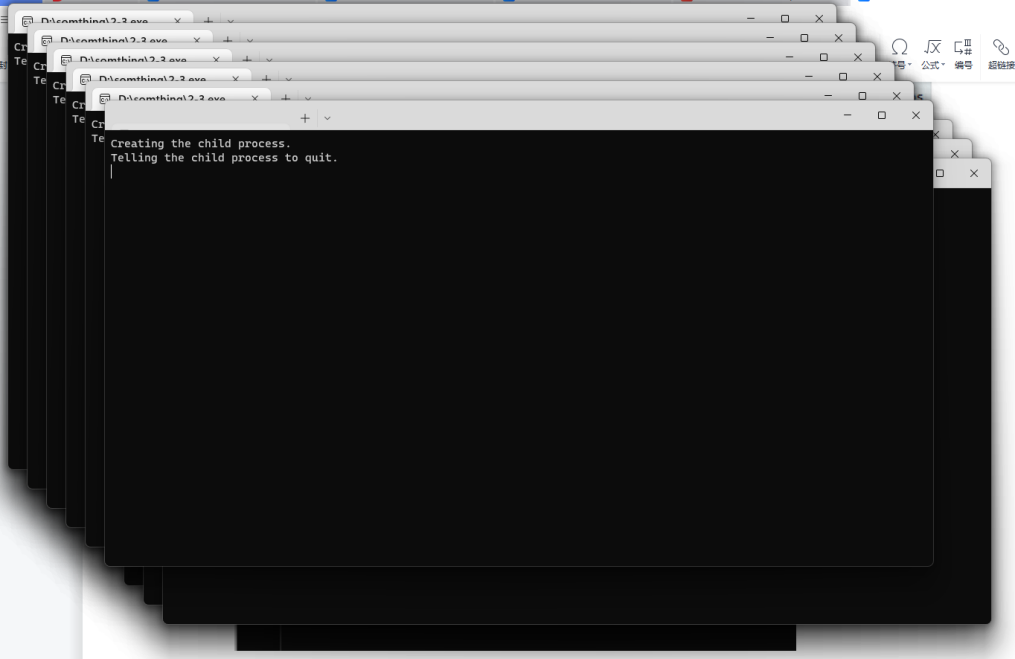
会一直创建进程，直到CPU利用率达到100%

nClone的作用是控制执行过程；未修改前，当nClone>5跳出循环，创建子进程就会结束；修改后，每次nClone会被重置为0，所以会陷入死循环，一直创建子进程

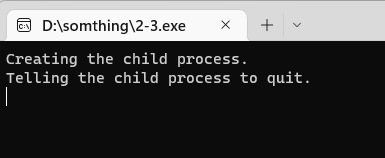
1. 未修改前运行：



修改后运行（1）：



修改后（2）：



**五、小结与心得体会**

1.加深了对操作系统进程概念的理解，Windows 所创建的每个进程都从调用 CreateProcess() API 函数开始，都以调用ExitProcess() 或TerminateProcess() API函数终止。可利用CreateMutex()API可创建互斥体，放弃共享资源时可调用ReleaseMute()API。

2. 修改为sscanf\_s出现了编译错误 

原因是sscanf\_s函数是C11标准中新增的函数，因此不是所有的编译器都支持它。我使用的是旧版本的编译器，无法使用sscanf\_s函数。

sscanf-s与sscanf函数的功能基本相同，但是sscanf\_s函数增加了一些安全检查。具体来说，sscanf\_s函数在解析字符串时会进行缓冲区边界检查，以防止缓冲区溢出。sscanf\_s函数还要求在format字符串中指定每个可变参数的最大长度，以便进行安全检查。

**一、实验题目**

**实验二 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）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。

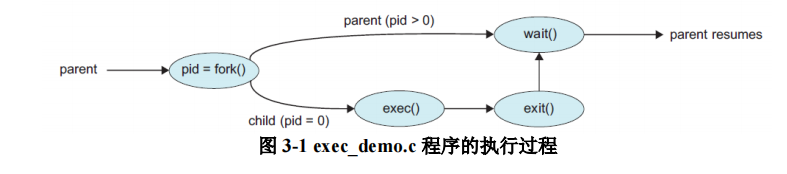
**（2）任务二：子进程执行新任务**

**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。

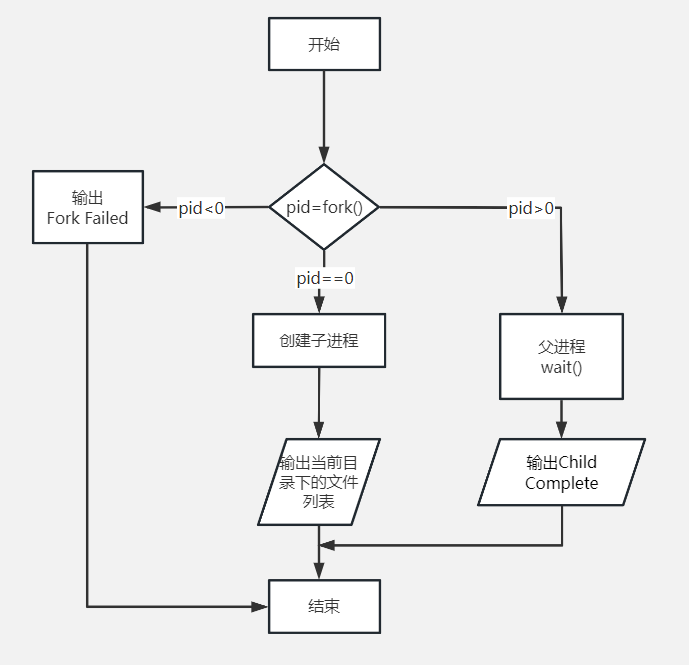
**步骤 1**：使用 vi 或 gedit 新建一个 exec\_demo.c 程序，然后拷贝清单 3-2 中的程序（该程序的执行如图 3-1 所示），使用cc或者 gcc 编译成可执行文件exec\_demo。例如，可以使用 gcc –o exec\_demo exec\_demo.c 完成编译。

**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。

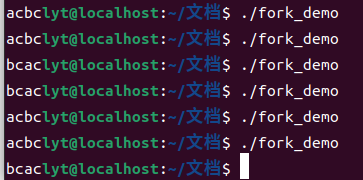
****

流程图：

****

**四、实验结果与分析**

**（1）**在main前加int后可编译运行

****

这是一个使用 fork() 函数创建进程的 C 语言程序，它会创建两个进程并打印出字符 "a" 和 "b"，最后打印一个字符 "c"。

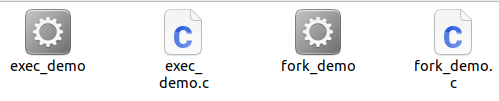
使用 srand() 函数初始化随机数种子，以便在后面的程序中使用 rand() 函数生成不同的随机数。在一个 while 循环中调用 fork() 函数创建子进程，如果 fork() 函数返回值为 -1，则表示创建进程失败，继续等待创建进程直到成功。

如果 fork() 函数返回值为 0，则表示当前进程为子进程，调用 sleep() 函数等待一个随机时间后输出字符 "a"。如果 fork() 函数返回值不为 0，则表示当前进程为父进程，调用 sleep() 函数等待一个随机时间后输出字符 "b"。最后，父进程和子进程都输出字符 "c"。

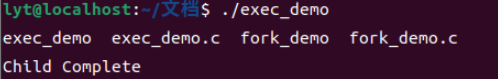
由于父进程和子进程同时运行，因此输出的字母顺序可能会不同，例如会输出 "acbc" 或 "bcac" 不同的顺序。

**（2）**

该目录下的文件列表：



编译执行结果：



调用fork()函数创建一个子进程，如果返回值小于 0，则表示创建进程失败，输出错误信息并退出程序。如果返回值等于 0，则表示当前进程为子进程，在子进程中通过调用 execlp() 函数来执行 /bin/ls 命令，显示当前目录下的文件列表。其中，第一个参数 "/bin/ls" 是要执行的命令，第二个参数 "ls" 是命令的名称，NULL 表示命令参数列表结束。如果返回值大于 0，则表示当前进程为父进程，在父进程中调用 wait() 函数来等待子进程的执行，直到子进程运行完毕后输出 "Child Complete"。

**五、小结与心得体会**

加深了我对Linux操作系统的进程概念的了解，也学会了在Linux基本运行，也使我明白了在Linux系统中子进程的创建，以及父子进程的运行过程，加深了对进程运行的理解。在Linux中利用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,while((x=fork())==-1)这句话是用来判断子进程是否能创建成功，而且当x=0时运行子进程，当x>0时父进程执行，而x<0时，则进程创建不成功，通过代码确定父子进程的先后执行顺序。

**一、实验题目**

**实验三 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**

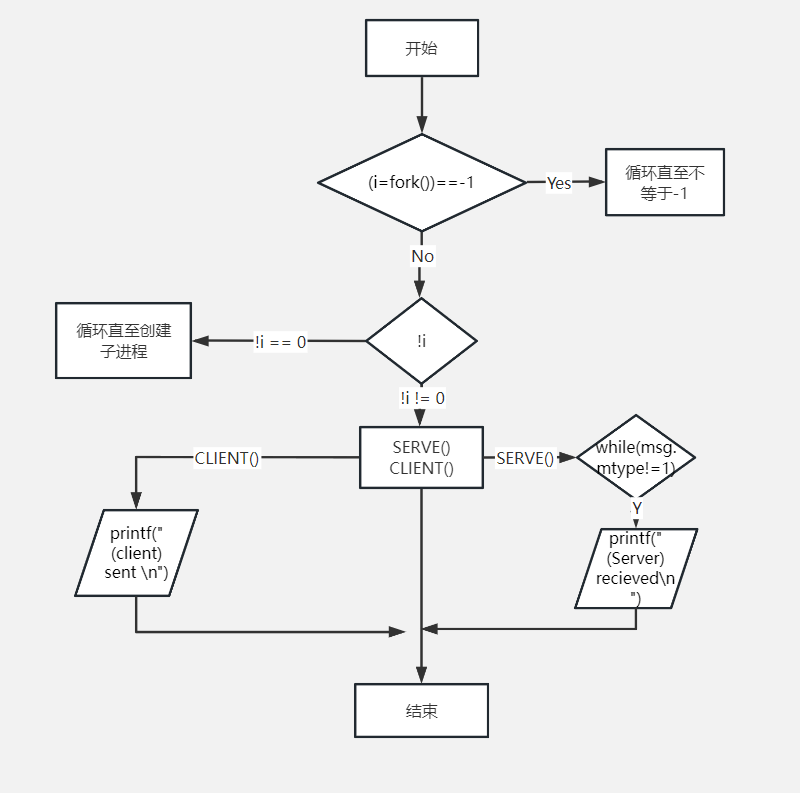
**消息的创建、发送和接收的程序设计**

（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。



**四、实验结果与分析**

****

**（1）**控制消息队列系统调用 msgctl()在此起的作用

msgctl()系统调用在SERVER函数中被调用，用于删除消息队列。具体来说，当SERVER函数接收到mtype为1的消息时，它调用msgctl()函数将消息队列从系统中删除，以便释放系统资源。在使用完消息队列之后，需要通过调用msgctl()函数将其从系统中删除，否则它将一直存在于系统中，占用系统资源。

删除消息队列可能会导致正在等待该消息队列的进程无法继续进行，因此在删除消息队列之前需要确保所有进程都已经完成对该消息队列的操作。这个程序中使用了wait()函数等待子进程退出，以确保在删除消息队列之前所有进程都已经完成对消息队列的操作。

**（2）**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。

在这个程序中，客户端进程和服务器进程都是并发运行的，它们之间的通信是通过共享的消息队列来实现的。由于进程的调度是由操作系统决定的，因此进程的执行顺序是不确定的。这导致在实际运行中，客户端发送的多条消息和服务器接收的多条消息可能不是交替出现的，而是在一段时间内连续出现，这是很正常的行为。另外，由于消息队列是一个共享的资源，多个进程都可以对它进行操作，因此在实际运行中可能会出现一些竞争条件。例如，当客户端进程和服务器进程同时试图从空队列中读取消息时，服务器进程可能会阻塞等待，直到有消息到达为止。同样地，当客户端进程试图向满队列中发送消息时，它可能会阻塞等待，直到有空闲空间为止。这些情况都可能导致程序的行为与预期不同。因此，在编写使用消息队列进行进程间通信的程序时，需要考虑这些竞争条件，并采取相应的措施来避免它们。

**五、小结与心得体会**

加深了我对Linux操作系统的进程进程间消息通信机制的了解，也能更加熟练的使用fork()函数，也学会了系统调用函数的使用以及相关命令的运用。值得注意的是多个进程是并发运行时，通信是通过共享的消息队列来实现，调度是由操作系统来决定，因此进程执行顺序是不确定的。

**一、实验题目**

**实验四 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**

**生产者消费者问题**

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

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

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

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

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

步骤 6：根据步骤 4 的结果，并查看 MSDN，回答下列问题：

1）CreateMutex 中有几个参数，各代表什么含义。

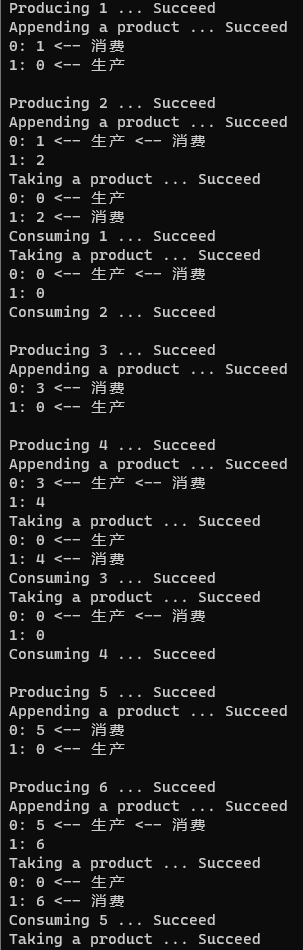
2）CreateSemaphore 中有几个参数，各代表什么含义，信号量的初值在第几个参数中。

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

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

1. **实验结果与分析**

**（1）步骤2：**

****

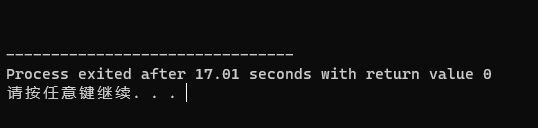
**（2）步骤3**：

创建线程的 WINDOWS API 函数是CreateThread函数，线程的第一个执行函数是Produce()，它位于CreateThread的第三个参数中。这个参数是一个指向线程函数的指针，当线程被创建时，它将从这个函数开始执行。

**（3）步骤4：**

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

**（4）步骤5：**



结果为空，因为在信号量被创建时，缓冲区中没有空闲位置可用，即初始状态下缓冲区是满的。

1. **步骤6：**
2. CreateMutex 中有几个参数，各代表什么含义。

三个

第一个参数是一个指向 SECURITY\_ATTRIBUTES 结构体的指针，用于指定互斥量的安全属性，通常可以设为 NULL，表示使用默认安全属性。

第二个参数指定互斥量的初始拥有者。 FALSE表示互斥量没有初始拥有者。

第三个参数指定互斥量对象的名称，通常可以设为 NULL，表示互斥量对象没有名称。

1. 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。

四个

第一个参数是一个指向 SECURITY\_ATTRIBUTES 结构体的指针，用于指定信号量的安全属性，通常可以设为 NULL，表示使用默认安全属性。

第二个参数指定信号量的初始计数器值。当参数的值为 SIZE\_OF\_BUFFER，即缓冲区的大小，表示在信号量被创建时，缓冲区中有 SIZE\_OF\_BUFFER 个空闲位置可用。

第三个参数指定信号量的最大计数器值。参数的值为 SIZE\_OF\_BUFFER，表示信号量的计数器最大值与缓冲区的大小相等。

第四个参数指定信号量对象的名称，通常可以设为 NULL，表示信号量对象没有名称。

1. 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。

P(full)：WaitForSingleObject(FullSemaphore,INFINITE)

V(full)：ReleaseSemaphore(FullSemaphore,1,NULL)

P(mutex)：WaitForSingleObject(Mutex,INFINITE)

V(mutex)：ReleaseMutex(Mutex)

p(empty)：WaitForSingleObject(EmptySemaphore,INFINITE)

V(empty)：ReleaseSemaphore(EmptySemaphore,1,NULL)

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

可以

Mutex = CreateMutex(NULL,FALSE,NULL)修改为Mutex =CreateSemaphore(NULL,FA

LSE,FALSE,NULL);生产者、消费者体内：ReleaseMutex(Mutex)修改为ReleaseSemaphore(Mutex,1,NULL)。

**五、小结与心得体会**

加深了我对Windows线程的了解，了解互斥体对象，基本掌握了互斥与同步操作编写生产者-消费者问题的并发程序，加深对 P (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。

**一、实验题目**

**实验五 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**

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 6-1 中的程序，编译成可执行文件。

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

**步骤 3**：根据运行结果，回答下列问题

虚拟内存每页容量为：\_\_\_\_\_4.00 KB\_\_\_\_\_ 最小应用地址：\_\_\_0x00010000\_\_\_\_

最大应用地址：\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_0x7ffffffeffff\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

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：\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_32G\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

理论上每个 Windows 应用程序可以独占的最大存储空间是：\_\_\_\_\_\_4GB\_\_\_\_\_\_\_

|  |
| --- |
| **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 6-3 格式记录

表 6-3 实验记录

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| 地址 | 大小 | 虚拟地址  空间类型 | 访问权限 | 描述 |

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| 00023000-00030000 | (52.0KB) | Free | NOACCESS | 拒绝访问 |
| 0004f000-00050000 | (4.00KB) | Free | NOACCESS | 拒绝访问 |
| 00054000-00060000 | (48.0KB) | Free | NOACCESS | 拒绝访问 |
| 00062000-00070000 | (56.0KB) | Free | NOACCESS | 拒绝访问 |
| 000a1000-000b0000 | (60.0KB) | Free | NOACCESS | 拒绝访问 |
| 000b3000-000c0000 | (52.0KB) | Free | NOACCESS | 拒绝访问 |
| 00122000-00130000 | (56.0KB) | Free | NOACCESS | 拒绝访问 |
| 001fe000-00200000 | (8.00KB) | Free | NOACCESS | 拒绝访问 |
| 0041e000-00420000 | (8.00KB) | Free | NOACCESS | 拒绝访问 |
| 00651000-00660000 | (60.0KB) | Free | NOACCESS | 拒绝访问 |
| 00661000-00670000 | (60.0KB) | Free | NOACCESS | 拒绝访问 |
| 00671000-00680000 | (60.0KB) | Free | NOACCESS | 拒绝访问 |
| 00681000-00690000 | (60.0KB) | Free | NOACCESS | 拒绝访问 |
| 006f2000-00700000 | (56.0KB) | Free | NOACCESS | 拒绝访问 |
| 00708000-00710000 | (32.0KB) | Free | NOACCESS | 拒绝访问 |
| 00719000-00740000 | (156KB) | Free | NOACCESS | 拒绝访问 |
| 00750000-007a0000 | (320KB) | Free | NOACCESS | 拒绝访问 |
| 01221000-01230000 | (60.0KB) | Free | NOACCESS | 拒绝访问 |
| 02631000-02640000 | (60.0KB) | Free | NOACCESS | 拒绝访问 |
| 02740000-61440000 | (1.48GB) | Free | NOACCESS | 拒绝访问 |
| 6145a000-64940000 | (52.8MB) | Free | NOACCESS | 拒绝访问 |
| 64955000-6fc40000 | (178MB) | Free | NOACCESS | 拒绝访问 |
| 6fda3000-7ffe0000 | (258MB) | Free | NOACCESS | 拒绝访问 |
| 7ffe1000-7ffe4000 | (12.0KB) | Free | NOACCESS | 拒绝访问 |
| 7ffe5000-7ff4fdec0000 | (1.96GB) | Free | NOACCESS | 拒绝访问 |
| 7ff5fffe1000-7ff5ffff0000 | (60.0KB) | Free | NOACCESS | 拒绝访问 |
| 7ff5ffff1000-7ffdd9cc0000 | (3.40GB) | Free | NOACCESS | 拒绝访问 |
| 7ffdd9dc0000-7ffddc740000 | (41.5MB) | Free | NOACCESS | 拒绝访问 |
| 7ffddc7d7000-7ffddf6f0000 | (47.0MB) | Free | NOACCESS | 拒绝访问 |
| 7ffddf801000-7ffddf810000 | (60.0KB) | Free | NOACCESS | 拒绝访问 |
| 7ffddf8aa000-7ffddf8b0000 | (24.0KB) | Free | NOACCESS | 拒绝访问 |
| 7ffddfc53000-7ffddfd90000 | (1.23MB) | Free | NOACCESS | 拒绝访问 |
| 7ffddfea8000-7ffddfeb0000 | (32.0KB) | Free | NOACCESS | 拒绝访问 |
| 7ffddfed6000-7ffde00b0000 | (1.85MB) | Free | NOACCESS | 拒绝访问 |
| 7ffde00e1000-7ffde00f0000 | (60.0KB) | Free | NOACCESS | 拒绝访问 |
| 7ffde014e000-7ffde09d0000 | (8.50MB) | Free | NOACCESS | 拒绝访问 |
| 7ffde0d59000-7ffde0f10000 | (1.71MB) | Free | NOACCESS | 拒绝访问 |
| 7ffde0fb7000-7ffde0fc0000 | (36.0KB) | Free | NOACCESS | 拒绝访问 |
| 7ffde10b1000-7ffde1260000 | (1.68MB) | Free | NOACCESS | 拒绝访问 |
| 7ffde1322000-7ffde13d0000 | (696KB) | Free | NOACCESS | 拒绝访问 |
| 7ffde157b000-7ffde17f0000 | (2.45MB) | Free | NOACCESS | 拒绝访问 |
| 7ffde1819000-7ffde20f0000 | (8.83MB) | Free | NOACCESS | 拒绝访问 |
| 7ffde2207000-7ffde2250000 | (292KB) | Free | NOACCESS | 拒绝访问 |
| 7ffde2464000-7fffffff0000 | (475MB) | Free | NOACCESS | 拒绝访问 |

|  |
| --- |
| **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？ |

解决：可以通过将数据复制到电子表格中，然后进行数据的分列，数据合并。

将系统当前的已调配区 (committed) 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 |

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| 00010000-00020000 | (64.0KB) | Committed | READWRITE | Mapped |
| 00020000-00023000 | (12.0KB) | Committed | READONLY | Mapped |
| 00030000-0004f000 | (124KB) | Committed | READONLY | Mapped |
| 00050000-00054000 | (16.0KB) | Committed | READONLY | Mapped |
| 00060000-00062000 | (8.00KB) | Committed | READWRITE | Private |
| 00070000-000a1000 | (196KB) | Committed | READONLY | Mapped |
| 000b0000-000b3000 | (12.0KB) | Committed | READONLY | Mapped |
| 000c0000-000c2000 | (8.00KB) | Committed | READWRITE | Private |
| 00130000-001fe000 | (824KB) | Committed | READONLY | Mapped |
| 003f7000-00400000 | (36.0KB) | Committed | READWRITE | Private |
| 00400000-00401000 | (4.00KB) | Committed | READONLY | Image |
| 00401000-00404000 | (12.0KB) | Committed | EXECUTE\_READ | Image |
| 00404000-00405000 | (4.00KB) | Committed | READWRITE | Image |
| 00405000-00408000 | (12.0KB) | Committed | READONLY | Image |
| 00408000-0040a000 | (8.00KB) | Committed | READWRITE | Image |
| 0040a000-0040c000 | (8.00KB) | Committed | WRITECOPY | Image |
| 0040c000-0041e000 | (72.0KB) | Committed | READONLY | Image |
| 00619000-0061c000 | (12.0KB) | Committed | GUARD | READWRITE |
| 0061c000-00620000 | (16.0KB) | Committed | READWRITE | Private |
| 00620000-00651000 | (196KB) | Committed | READONLY | Mapped |
| 00660000-00661000 | (4.00KB) | Committed | READONLY | Mapped |
| 00670000-00671000 | (4.00KB) | Committed | READONLY | Mapped |
| 00680000-00681000 | (4.00KB) | Committed | READONLY | Mapped |
| 00690000-00691000 | (4.00KB) | Committed | READWRITE | Private |
| 00700000-00704000 | (16.0KB) | Committed | READONLY | Mapped |
| 00710000-00719000 | (36.0KB) | Committed | READONLY | Mapped |
| 00740000-00747000 | (28.0KB) | Committed | READWRITE | Private |
| 007a0000-007c6000 | (152KB) | Committed | READWRITE | Private |
| 00a99000-00a9c000 | (12.0KB) | Committed | GUARD | READWRITE |
| 00a9c000-00aa0000 | (16.0KB) | Committed | READWRITE | Private |
| 00c9b000-00c9e000 | (12.0KB) | Committed | GUARD | READWRITE |
| 00c9e000-00ca0000 | (8.00KB) | Committed | READWRITE | Private |
| 00e9b000-00e9e000 | (12.0KB) | Committed | GUARD | READWRITE |
| 00e9e000-00ea0000 | (8.00KB) | Committed | READWRITE | Private |
| 00ea0000-00f1b000 | (492KB) | Committed | READONLY | Mapped |
| 010a0000-01221000 | (1.50MB) | Committed | READONLY | Mapped |
| 01230000-015d1000 | (3.62MB) | Committed | READONLY | Mapped |
| 02640000-02654000 | (80.0KB) | Committed | READWRITE | Private |
| 61440000-61441000 | (4.00KB) | Committed | READONLY | Image |
| 61441000-61450000 | (60.0KB) | Committed | EXECUTE\_READ | Image |
| 61450000-61451000 | (4.00KB) | Committed | READWRITE | Image |
| 61451000-61454000 | (12.0KB) | Committed | READONLY | Image |
| 61454000-61455000 | (4.00KB) | Committed | READWRITE | Image |
| 61455000-61456000 | (4.00KB) | Committed | READONLY | Image |
| 61456000-61457000 | (4.00KB) | Committed | READWRITE | Image |
| 61457000-61459000 | (8.00KB) | Committed | WRITECOPY | Image |
| 61459000-6145a000 | (4.00KB) | Committed | READONLY | Image |
| 64940000-64941000 | (4.00KB) | Committed | READONLY | Image |
| 64941000-64949000 | (32.0KB) | Committed | EXECUTE\_READ | Image |
| 64949000-6494a000 | (4.00KB) | Committed | READWRITE | Image |
| 6494a000-6494d000 | (12.0KB) | Committed | READONLY | Image |
| 6494d000-6494e000 | (4.00KB) | Committed | READWRITE | Image |
| 6494e000-64950000 | (8.00KB) | Committed | READONLY | Image |
| 64950000-64951000 | (4.00KB) | Committed | READWRITE | Image |
| 64951000-64954000 | (12.0KB) | Committed | WRITECOPY | Image |
| 64954000-64955000 | (4.00KB) | Committed | READONLY | Image |
| 6fc40000-6fc41000 | (4.00KB) | Committed | READONLY | Image |
| 6fc41000-6fd08000 | (796KB) | Committed | EXECUTE\_READ | Image |
| 6fd08000-6fd0c000 | (16.0KB) | Committed | READWRITE | Image |
| 6fd0c000-6fd43000 | (220KB) | Committed | READONLY | Image |
| 6fd43000-6fd45000 | (8.00KB) | Committed | READWRITE | Image |
| 6fd45000-6fd9d000 | (352KB) | Committed | READONLY | Image |
| 6fd9d000-6fd9e000 | (4.00KB) | Committed | READWRITE | Image |
| 6fd9e000-6fda1000 | (12.0KB) | Committed | WRITECOPY | Image |
| 6fda1000-6fda3000 | (8.00KB) | Committed | READONLY | Image |
| 7ffe0000-7ffe1000 | (4.00KB) | Committed | READONLY | Private |
| 7ffe4000-7ffe5000 | (4.00KB) | Committed | READONLY | Private |
| 7ff4fdec0000-7ff4fdec5000 | (20.0KB) | Committed | READONLY | Mapped |
| 7ff5fffe0000-7ff5fffe1000 | (4.00KB) | Committed | READWRITE | Private |
| 7ff5ffff0000-7ff5ffff1000 | (4.00KB) | Committed | READONLY | Mapped |
| 7ffdd9cc0000-7ffdd9cc1000 | (4.00KB) | Committed | READONLY | Image |
| 7ffdd9cc1000-7ffdd9d69000 | (672KB) | Committed | EXECUTE\_READ | Image |
| 7ffdd9d69000-7ffdd9dad000 | (272KB) | Committed | READONLY | Image |
| 7ffdd9dad000-7ffdd9daf000 | (8.00KB) | Committed | READWRITE | Image |
| 7ffdd9daf000-7ffdd9dc0000 | (68.0KB) | Committed | READONLY | Image |
| 7ffddc740000-7ffddc741000 | (4.00KB) | Committed | READONLY | Image |
| 7ffddc741000-7ffddc793000 | (328KB) | Committed | EXECUTE\_READ | Image |
| 7ffddc793000-7ffddc7b7000 | (144KB) | Committed | READONLY | Image |
| 7ffddc7b7000-7ffddc7ba000 | (12.0KB) | Committed | READWRITE | Image |
| 7ffddc7ba000-7ffddc7d7000 | (116KB) | Committed | READONLY | Image |

将系统当前的保留区 (reserved) 虚拟地址空间按表 6-5 格式记录。

表 6-5 实验记录

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| 地址 | 大小 | 虚拟地址  空间类型 | 访问权限 | 描述 |

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| 000c2000-00122000 | (384KB) | Reserved | READONLY | Private |
| 00200000-003f7000 | (1.96MB) | Reserved | READONLY | Private |
| 00420000-00619000 | (1.97MB) | Reserved | READONLY | Private |
| 00691000-006f2000 | (388KB) | Reserved | READONLY | Private |
| 00704000-00708000 | (16.0KB) | Reserved | READONLY | Mapped |
| 00747000-00750000 | (36.0KB) | Reserved | READONLY | Private |
| 007c6000-008a0000 | (872KB) | Reserved | READONLY | Private |
| 008a0000-00a99000 | (1.97MB) | Reserved | READONLY | Private |
| 00aa0000-00c9b000 | (1.98MB) | Reserved | READONLY | Private |
| 00ca0000-00e9b000 | (1.98MB) | Reserved | READONLY | Private |
| 00f1b000-010a0000 | (1.51MB) | Reserved | READONLY | Mapped |
| 015d1000-02631000 | (16.3MB) | Reserved | READONLY | Mapped |
| 02654000-02740000 | (944KB) | Reserved | READONLY | Private |
| 7ff4fdec5000-7ff4fdfc0000 | (0.98MB) | Reserved | READONLY | Mapped |
| 7ff4fdfc0000-7ff5fdfe0000 | (128KB) | Reserved | READONLY | Private |
| 7ff5fdfe0000-7ff5fffe0000 | (32.0MB) | Reserved | READONLY | Private |

1. **实验结果与分析**

**（1）报错：**

问题1：将void改为int

问题2：error: cast from 'LPCVOID' {aka 'const void\*'} to 'DWORD' {aka 'long unsigned int'} loses precision [-fpermissive]|

解决：将(DWORD)改为(DWORD64)

问题3：显示StrFormatByteSize()不可以用

解决：没有链接库，右击项目名称—>build options—>link settings—>other linker options—>添加lshlwapi

**（2）**该程序从主函数Main()出发，调用void WalkVM(HANDLE hProcess)函数，void WalkVM(HANDLEhProcess)函数获得系统信息，分配应用程序内存地址空间。然后开始做循环，从函数运行开始每次获得下一个虚拟程序内存块的信息，之后计算块的结尾及大小，然后再显示块的大小与位置，状态，显示保护方式(void ShowProtection(DWORD dwTarget))，将文件名显示出来，移动块指针获得下一块，依次这样循环下去，直至结束。

**五、小结与心得体会**

了解了内存结构和虚拟内存的管理，理解了进程的虚拟内存空间和物理内存的映射关系。了解了Virtual QueryEX() API 等虚拟内存API的作用及使用方法

1. **实验题目**

**实验六 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（含背景知识或基本原理与算法、或模块介绍、设计步骤等）**

1. 基本原理

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。

2. 模块设计：

1）初始化矩阵void Init()（Max、Allocation、Available矩阵）

2）判断初始化矩阵的合法性bool Judge()（Allocation[i][j] <= Max[i][j]）

3）判断当前work向量是否大于Need[i]向量bool Judge(int k)

（Need[k][j] <=Work[j]）

4）安全性检查的时候虚拟处理void VirtureDeal(int k)（循环Work[j] += Allocation[k][j]）

5）安全性检查bool SecurityCheck(queue<int>& q)

6）试探性分配资源void Alloc(int k)（Allocation[k][j] += Request[j];

Need[k][j] -= Request[j];Available[j] -= Request[j];）

1. 恢复原状 void Recover(int k)（Allocation[k][j] -= Request[j];

Need[k][j] += Request[j];Available[j] += Request[j];）

8）请求资源void Query()

**四、详细设计（含主要的数据结构、程序流程图、关键代码等）**

**数据结构：**

·资源总量向量 Sum，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 Need，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。

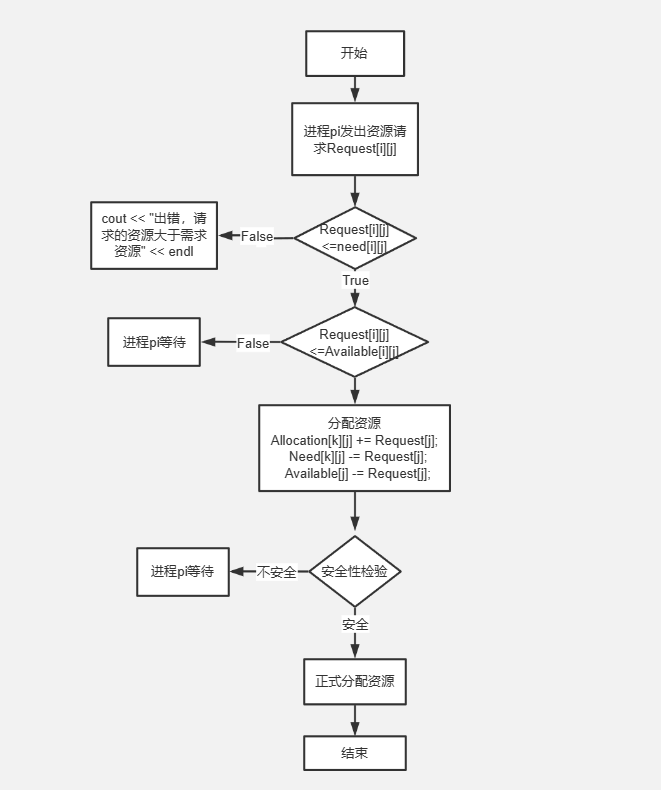
·最大需求总量 Max，n\*m矩阵，表示进程n对m类资源最大的需求总量。

·工作向量Work，m维，表示系统可提供给进程继续运行所需的m类资源的数目。

·进程状态向量Finish，m维，表示每一个进程是否运行完成，用布尔值来表示进程的状态，True表示以完成运行。

·请求资源向量Request，m\*n维

**程序流程图：**



**关键代码：**

/\*请求资源\*/

void Query(){

int ID;

cout << "请输入进程ID(1 -- " << n << "):" << endl; cin >> ID;

cout << "请输入请求向量(输入" << m << "个有效整数):" << endl;

for (int j = 1;j <= m;j++) cin >> Request[j];

for (int j = 1;j <= m;j++)

{

if (Request[j] > Need[ID][j]){

cout << "出错，请求的资源大于需求资源" << endl;

return;

}

else if (Request[j] > Available[j]){

cout << "进程无法得到资源，继续处于等待状态！" << endl;

return;

}

}

Alloc(ID);//试探性分配

queue<int> q;

if (SecurityCheck(q)){

cout << "分配资源成功！" << endl;

cout << "其中的一个安全序列是：";

bool mark = false;

while (q.size())

{

if (mark) cout << "->";

cout << q.front();q.pop();

mark = true;

}

cout << endl;

}

else {

cout << "无法执行分配任务！" << endl;

Recover(ID);//无法执行分配，恢复原状

}

}

int main(){

cout << "请输入进程个数: "; cin >> n;

cout << "请输入资源的种类数: "; cin >> m;

Init();

system("pause");

while (1)

{

cout << "1、请求资源" << endl;

cout << "2、检查当前状态的安全性" << endl;

cout << "0、退出主程序" << endl;

while (1){

int op; cin >> op;

if (op < 0 || op > 2) {

cout << "操作错误，请重新输入！！！" << endl;

system("pause");

}

else {

if (op == 0) exit(0);

else if (op == 1) {

Query();

system("pause");

}

else if (op == 2){

queue<int> q;

if (SecurityCheck(q)){

cout << "当前状态安全" << endl;

cout << "其中的一个安全序列是：";

bool mark = false;

while (q.size()){

if (mark) cout << "->";

cout << q.front();q.pop();

mark = true;

}

cout << endl;

}

else {

cout << "当前状态不安全，极有可能进入死锁状态" << endl;

}

system("pause");

}

}

break;

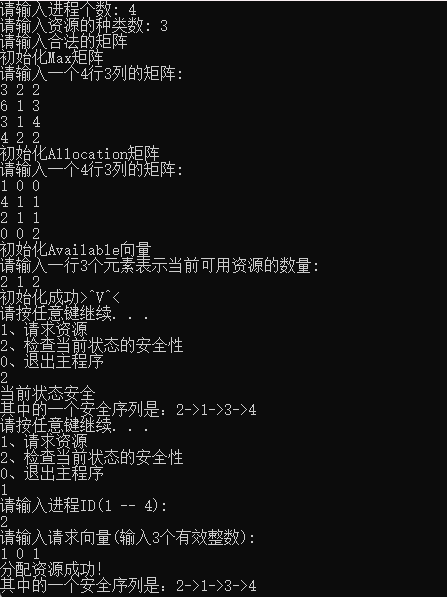
}

}

return 0;

}

**五、实验结果与分析**

qa

**六、小结与心得体会**

学会了银行家算法的基本原理，对银行家算法有了更深一步的了解，银行家算法是避免死锁的重要方法。避免死锁关键在于对于请求资源不能大于剩余资源的数量，而且要时刻检查资源是否足够，还有系统尝试分配资源给进程然后得到新的安全序列，不安全则恢复。

对进程死锁、安全状态与不安全状态的概念理解，进程死锁的条件是互斥、占有等待、不可抢占、循环等待，可通过破坏任一条件避免死锁。银行家算法相比于其他，代价更小。

**一、实验题目**

**实验七 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. 基本原理**

**（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（FIFO）：**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（SSTF）：**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。

**四、详细设计（含主要的数据结构、程序流程图、关键代码等）**

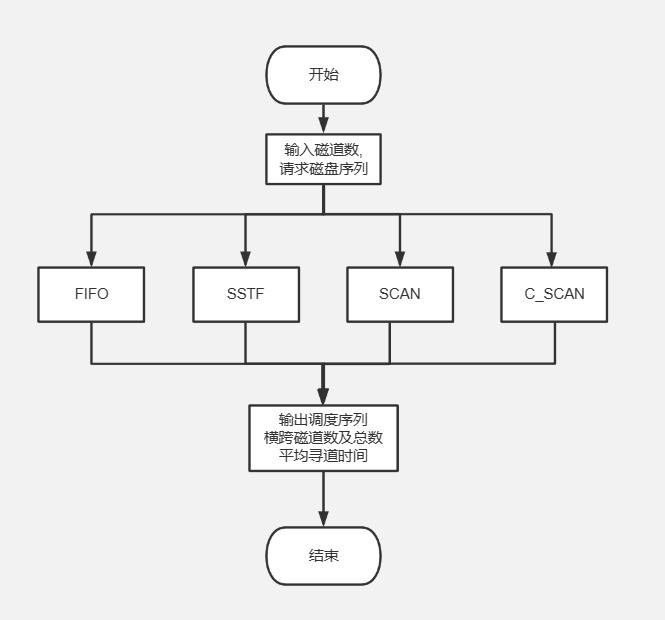
**数据结构：**

主要采用数组结构

·int request[100]： 请求磁盘序列

·int re[100]： 复制初始序列

·int r[100]： 记录每个算法执行后序列



**关键代码：**

**·先进先出（FIFO）**

void FIFO(){ //先进先出

kua=abs(begin-request[0]);

printf("\n先进先出算法：\n FIFO调度： %3d",begin);

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

printf(" %3d",request[i]);

printf("\n 横跨磁道数为： %3d",abs(begin-request[0]));

for(int i=1;i<num;i++){

k=abs(request[i-1]-request[i]);

printf(" %3d",k);

kua+=k;

}

printf("\n 横跨的总磁道数： %3d",kua);

printf("\n 平均寻道长度： %.5f\n",1.0\*kua/num);

}

**·最短服务时间优先（SSTF）**

int Smin(int b,int re[]){ //返回离 开始磁盘b最近的磁盘下标

int min=abs(b-re[0]);

int j=0;

for(int i=1;i<num;i++)

if(abs(b-re[i]) < min){

min=abs(b-re[i]);

j=i;

}

return j;

}

void SSTF(){ //最短服务时间优先

int c=0,b=begin;

printf("\n最短服务时间优先算法：\n SSTF调度： %3d",begin);

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

c=Smin(b,re); //返回最近的磁道下标

b=re[c]; //将最近的磁盘作为开始

re[c]=9999999; //将已经访问过的磁盘 设为很大值

printf(" %3d",b);

r[i]=b;

}

kua=abs(begin-r[0]);

printf("\n 横跨磁道数为： %3d",abs(begin-r[0]));

for(int i=1;i<num;i++){

k=abs(r[i-1]-r[i]);

printf(" %3d",k);

kua+=k;

}

printf("\n 横跨的总磁道数： %3d",kua);

printf("\n 平均寻道时间： %.5f\n",1.0\*kua/num);

}

**·扫描算法（SCAN）**

void SCAN(){ //扫描算法

int c=0,b=begin;

for(int i=0;i<num;i++) //SSTF时re[]已改变

re[i]=request[i];

printf("\n扫描算法：\n SCAN调度： %3d",begin);

for(int i=0;i<num-1;i++){

for (int j=0;j<num-i-1;j++){

if(re[j]>re[j+1]){

re[j]=re[j]+re[j+1];

re[j+1]=re[j]-re[j+1];

re[j]=re[j]-re[j+1];

}

}

}

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

if(re[i]>b){

printf(" %3d",re[i]);

r[c++]=re[i];

}

for(int i=num-1;i>=0;i--)

if(re[i]<b){

printf(" %3d",re[i]);

r[c++]=re[i];

}

kua=abs(begin-r[0]);

printf("\n 横跨磁道数为： %3d",abs(begin-r[0]));

for(int i=1;i<num;i++){

k=abs(r[i-1]-r[i]);

printf(" %3d",k);

kua+=k;

}

printf("\n 横跨的总磁道数： %3d",kua);

printf("\n 平均寻道时间： %.5f\n",1.0\*kua/num);

}

**·循环扫描算法（C\_SCAN）**

void C\_SCAN(){ //循环扫描

int c=0,b=begin;

printf("\n循环扫描算法：\n CSCAN调度： %3d",begin);

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

if(re[i]>b){

printf(" %3d",re[i]);

r[c++]=re[i];

}

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

if(re[i]<b){

printf(" %3d",re[i]);

r[c++]=re[i];

}

kua=abs(begin-r[0]);

printf("\n 横跨磁道数为： %3d",abs(begin-r[0]));

for(int i=1;i<num;i++){

k=abs(r[i-1]-r[i]);

printf(" %3d",k);

kua+=k;

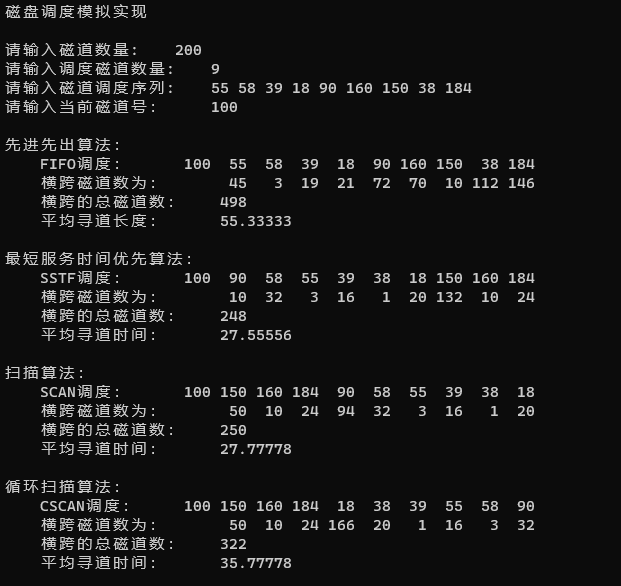
}

printf("\n 横跨的总磁道数： %3d",kua);

printf("\n 平均寻道时间： %.5f\n",1.0\*kua/num);

}

**五、实验结果与分析**



**六、小结与心得体会**

通过使用实现磁盘调度算法，我深入了解了不同算法的原理和特点，并提高了自己的编程技能和学习能力。

在使用代码模拟磁盘调度算法时，我需要根据具体的应用场景选择合适的算法，并根据输入的数据进行计算和输出结果。同时，我也需要注意代码的细节问题，如变量的类型、循环的边界、数组下标的范围等，以避免出现错误和异常情况。

通过实现不同的磁盘调度算法，我了解到不同算法的优缺点和适用范围。例如，先进先出算法适用于请求分布较均匀的情况；最短服务时间优先算法可以使磁头移动的距离最短，从而提高磁盘的读写效率；扫描算法和循环扫描算法适用于请求分布集中的情况等。

总之，通过使用上述代码实现磁盘调度算法，我不仅了解到不同算法的特点和使用方法，还提高了自己的编程能力和实践能力，这对我今后的学习和工作都有很大的帮助。

**一、实验题目**

**实验八\* 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. 基本原理**

需要调入新页面时，选择内存中哪个物理页面被置换，称为置换策略。页面置换算法的目标：

把未来不再使用的或短期内较少使用的页面调出，通常应在局部性原理指导下依据过去的统计数据进行预测，减少缺页次数。

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 的折中。

为了兼顾程序的局部性原理，要通过随机数产生一个指令序列  
**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； 这样生成的指令序列是随机且有局部性

**2.模块设计：**

1. 指令序列的生成
2. 判断判断第i条指令是否在内存中以及判断内存块是否存满
3. 三个算法的实现

**四、详细设计（含主要的数据结构、程序流程图、关键代码等）**

**数据结构：**

主要采用数组结构

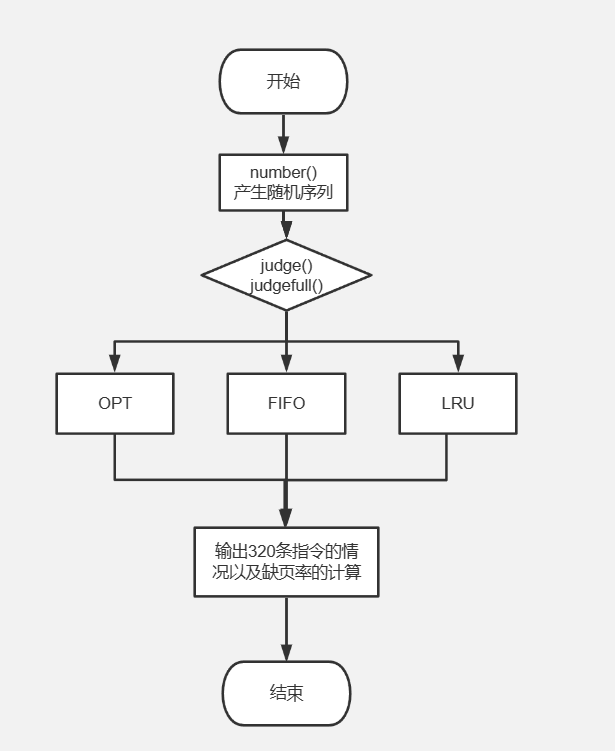
int page[4]={-1,-1,-1,-1}：表示进程内存页面4页,初始值为-1

int number[321]：存放320要执行的指令,方便模3，凑个整数，最后一条舍弃

int start[4]：用于记录页面刚被调入内存时的时间

int recent[4]：用与记录页面最近被调用的时间

**程序流程图：**



**关键代码：**

**·随机产生指令序列**

void random(){

int m1,m2,m3;

for(int i=0;i<321;){ //每次产生3条

m1=rand()%320; //rand()函数没有参数，不能规定范围

number[i++]=m1+1; //计算得的指令就存入数组

m2=rand()%(m1+1); //后置++先用后加

number[i++]=m2+1;

m3=rand()%(318-m2)+m2+2; //前后部分表示0-（317-m2)加上就变成指定区间了

number[i++]=m3; //不要后移

}

printf("随机的320条指令序列为：\n");

for(int i=0;i<320;i++){

if(i%45==0)printf("\n");

printf("%3d ",number[i]);

} printf("\n\n");

}

**·最佳置换算法(OPT)**

void OPT(){

count=0;

printf("最佳置换算法(OPT):\n");

for(int i=0;i<320;i++){

bool jieguo1=judge(i);

if(!jieguo1) { //表示指令所在页面不在内存中

count++;

bool jieguo2=judgefull(i);

if(!jieguo2) { //表示内存块已经存满

int juli[4]={-1,-1,-1,-1}; //存储四个页面到下一次被调度的距离

for(int j=0;j<4;j++){

if(juli[j]==-1) { //表示还未出现过

for(int k=i+1;k<320;k++){

if(page[j]==number[k]/10) {//下一次第一次被调度，记录距离

juli[j]=k-i;break;

}

}

}

}

int flag=0,index=-1;

for(int j=0;j<4;j++){

if(juli[j]==-1){//表示此页以后都未被调用过，则替换此页

index=j; flag=1;break;

}

}

if(flag==0) { //表示内存中的页都出现过，则取距离最远

int max=-1;

for(int j=0;j<4;j++){

if(juli[j]>max){

max=juli[j];index=j;

}

}

}

page[index]=number[i]/10; //模拟把指令所在页面调入内存

printf("该指令所在页不在内存中,内存块已满，采用OPT算法，其物理地址为：页号：%d,偏移量：%d\n",index,number[i]%10);

}

}

}

printf("\nOPT算法的缺页率为：%f",1.0\*count/320);

}

**·先进先出置换算法(FIFO)**

void FIFO(){

count=0;

printf("先进先出置换算法(FIFO):\n");

for(int i=0;i<320;i++){

bool jieguo1=judge(i);

if(!jieguo1) { //表示指令所在页面不在内存中

count++;

bool jieguo2=judgefull(i);

if(!jieguo2){//表示内存满了

int min=9999,index=-1;

for(int j=0;j<4;j++){

if(start[j]<min){

min=start[j];index=j; //取最早被调入内存的页面

}

}page[index]=number[i]/10; //模拟把指令所在页面调入内存

start[index]=i;

printf("该指令所在页不在内存中,内存块已满，采用FIFO算法，其物理地址为：页号：%d,偏移量：%d\n",index,number[i]%10);

}

}

}

printf("\nFIFO算法的缺页率为：%f",1.0\*count/320);

}

**·最近最久未用置换算法(LRU)**

最近最久未用置换算法(LRU):\n");

for(int i=0;i<320;i++){

bool jieguo1=judge(i);

if(!jieguo1) {//表示指令所在页面不在内存中

count++;

bool jieguo2=judgefull(i);

if(!jieguo2) {//表示内存满了

int min=999,index=-1;

for(int j=0;j<4;j++){

if(recent[j]<min){

min=recent[j];index=j; //取最近最远被调入的页面

}

}page[index]=number[i]/10;//模拟把指令所在页面调入内存

recent[index]=i;

printf("该指令所在页不在内存中,内存块已满，采用FIFO算法，其物理地址为：页号：%d,偏移量：%d\n",index,number[i]%10);

}

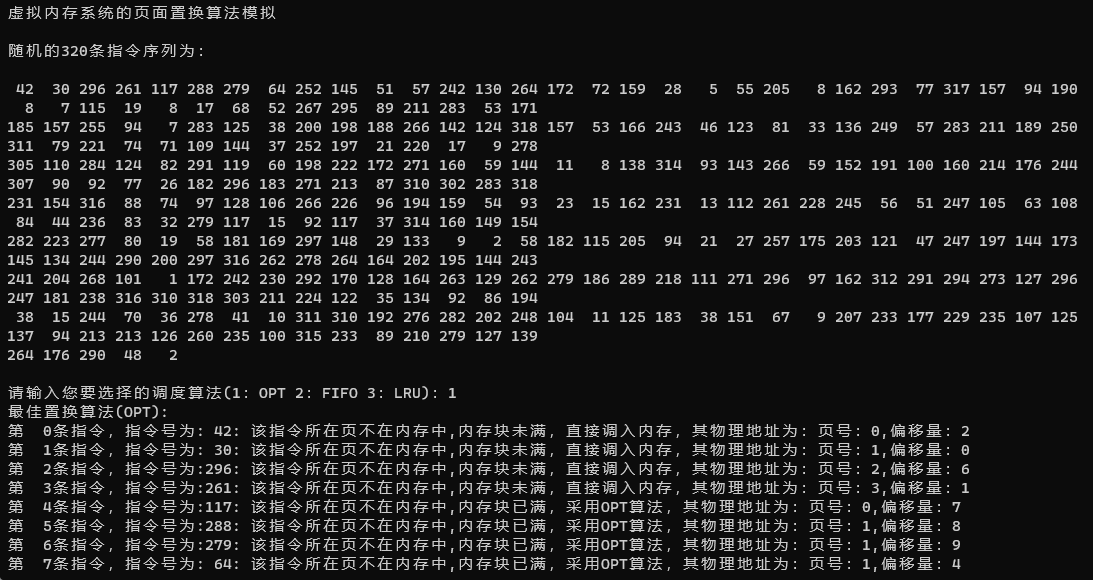
}

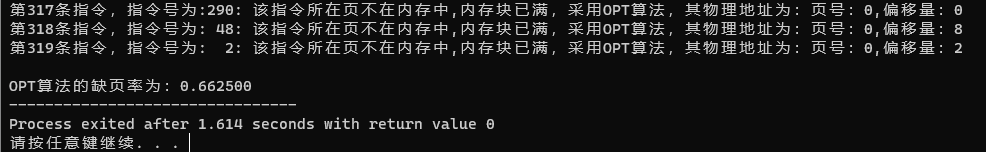
}

printf("\nLRU算法的缺页率为：%f",1.0\*count/320);

}

**五、实验结果与分析**









从结果上来看，OPT的缺页率是最低的，FIFO和LRU算法的缺页率近乎相等，但是LRU还是略优于FIFO算法，但是实现相较下更为复杂，而FIFO的实现是最为简单的。OPT只是一种理论上最优的算法，由于要预测未来，所以实现也最为复杂。

**六、小结与心得体会**

通过对虚拟内存系统的页面置换算法模拟的实现，我深入了解了不同置换算法的原理和特点，并提高了自己的编程技能和学习能力。在编程过程中需要注意代码的细节问题，如变量的类型、循环的边界、数组下标的范围等，以避免出现错误和异常情况。

**OPT：**OPT算法是一种理论上最优的页面置换算法，然而，由于无法预测未来的页面访问情况，因此OPT算法在实际应用中很难实现。OPT算法常常被用作评估其他页面置换算法的性能。

**FIFO：**它按照页面进入内存的时间顺序进行替换。优点是实现简单、开销小，适用于内存资源较为充足的系统。缺点是它无法很好地反映页面的使用情况，可能会将最常用的页面替换出去，从而导致性能下降。因此，在内存资源较为紧张的系统中，FIFO算法往往表现不佳。

**LRU：**优点是它能够很好地反映页面的使用情况，可以有效地提高系统的性能。缺点是它需要维护一个页面访问时间的链表，这会增加算法的开销。因此，在内存资源紧张的系统中，其性能可能会受到影响。

**一、实验题目**

**实验九\* 基于信号量机制的并发程序设计**

**二、实验目的**

(1) 回顾操作系统进程、线程的有关概念，针对经典的同步、互斥、死锁与饥饿问题进行并发程序设计。

(2) 了解互斥体对象，利用互斥与同步操作编写读者-写者问题的并发程序，加深对 P (即semWait)、V(即 semSignal)原语以及利用 P、V 原语进行进程间同步与互斥操作的理解。

(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**

**读者-写者问题的并发程序设计**

根据实验五中所熟悉的 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）。

**1. 基本原理**

有读者和写者两组并发进程，共享一个文件，当两个或两个以上的读进程同时访问共享数据时不会产生副作用，但若是某个写进程和其他进程同时访问共享数据时则可能会导致数据不一致的错误。要求：

1. 允许多个读者进程同时执行读操作。
2. 只允许一个写者往文件中写信息；
3. 任一写者在完成写操作之前不允许其他读者或写者工作；
4. 写者执行写操作前应让所有读者和写者全部退出。

**模块设计：**

**（1）**设置三个互斥信号量，和一个count计数器。

**（2）**写进程，一个P(w)判断是否有其他访问者，然后P(m)对文件加锁，开始写文件，完成释放V(w)，V(m)。

**（3）**读进程，先用P(w)判断是否有写者进程再写或者申请写，然后一个P©实现对count的判断和修改，if判断如果count==0，就用P(m)对文件加锁，然后count自加，释放V©，V(w)，开始读文件，完成先获得P©然后count自减，if判断count等于0，就解锁V(w)，然后释放V©。

**（4）**Main函数通过创建读写进程，并实际程序结束出口。

**四、详细设计（含主要的数据结构、程序流程图、关键代码等）**

**关键代码：**

**·写进程**

DWORD WINAPI Writer(LPVOID lpPara)

{

while(p\_ccontinue){//程序在运行中

WaitForSingleObject(Writer\_Mutex,INFINITE); //p(mutex);

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

printf("%d开始写文件",WriterID);

Sleep(1000);

printf("...%d完成写文件\n",WriterID);

WriterID++;

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

ReleaseMutex(Writer\_Mutex); //V(mutex);

}return 0;

}

**·读进程**

DWORD WINAPI Reader(LPVOID lpPara){

while(p\_ccontinue){

WaitForSingleObject(Writer\_Mutex,INFINITE); //P(mutex1);

WaitForSingleObject(Count\_Mutex,INFINITE); //P(mutex1);

if(count==0) WaitForSingleObject(Mutex,INFINITE); //p(mutex);

count++;

ReleaseMutex(Count\_Mutex); //V(mutex);

ReleaseMutex(Writer\_Mutex); //V(mutex);

Sleep(1000);

WaitForSingleObject(Count\_Mutex,INFINITE); //P(mutex1);

printf("已有%d个读者完成读文件\n",ReaderID++);

count--;

if(count==0) ReleaseMutex(Mutex); //V(mutex);

ReleaseMutex(Count\_Mutex); //V(mutex);

}return 0;

}

**·主函数**

int main(){

Mutex = CreateMutex(NULL,FALSE,NULL);

Count\_Mutex = CreateMutex(NULL,FALSE,NULL);

Writer\_Mutex= CreateMutex(NULL,FALSE,NULL);

int WRITER\_COUNT = 5,READER\_COUNT = 10;

int THREADS\_COUNT = WRITER\_COUNT+READER\_COUNT;

HANDLE hThreads[THREADS\_COUNT]; //各线程的 handle

DWORD writerID[WRITER\_COUNT]; //写者线程的标识符

DWORD readerID[READER\_COUNT]; //读者线程的标识符

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

hThreads[i]=CreateThread(NULL,0,Writer,NULL,0,&writerID[i]);

if (hThreads[i]==NULL) return -1;}

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

hThreads[WRITER\_COUNT+i]=CreateThread(NULL,0,Reader,NULL,0,&readerID[i]);

if (hThreads[i]==NULL) return -1;}

while(p\_ccontinue){

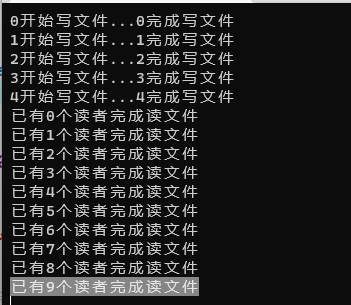
if(getchar()) p\_ccontinue = false;//按回车后终止程序运行

}

return 0;

}

**五、实验结果与分析**

****

**六、小结与心得体会**

用c语言实现对基于信号量机制的并发程序设计，让我进一步熟悉了Windows API，并且熟练掌握了并发程序的原理以及读写者问题，了解信号量的定义和初始化，线程的创建和PV操作

**一、实验题目**

**实验十\* 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.设计内容及要求：**

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。

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；

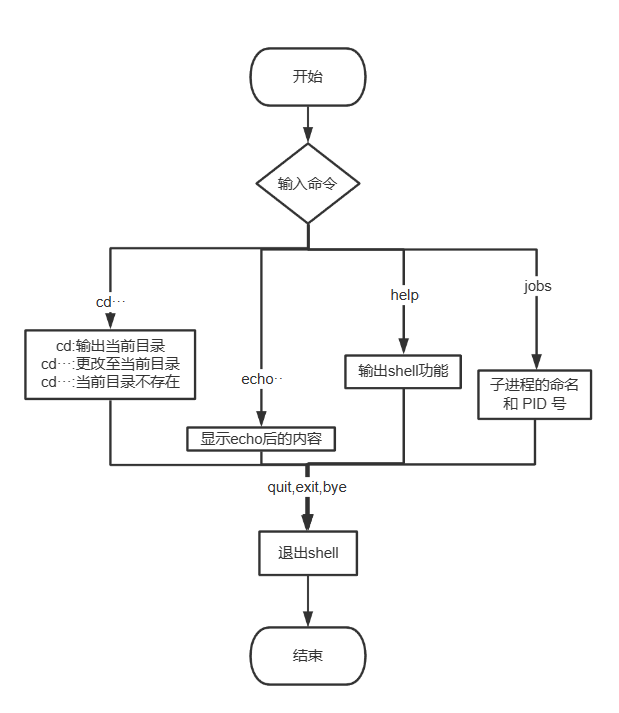
else

提示错误的命令；

}

**四、详细设计（含主要的数据结构、程序流程图、关键代码等）**

**程序流程图：**



**关键代码：**

**·cd**

void cd(){

printf("请输入目录,更改当前的工作目录到另一个目录\n");

char st[200];

memset(st,0,sizeof(st));

scanf("%s",st);

char mulu[200];

memset(mulu,0,sizeof(mulu));//清空mulu数组

getcwd(mulu,300);//调用函数保存当前目录

printf("当前目录为: %s\n",mulu);

if(strlen(st)==0)//没有输入，则输出当前目录

printf("没有输入路径,当前目录是：%s\n",mulu);

else{

if(chdir(st)==0)//跳转至指定目录{

memset(mulu,0,sizeof(mulu));//清空之前的存储内容

getcwd(mulu,200);//读取当前目录

printf("跳转目录至：%s\n",mulu);

}

else printf("跳转目录失败！请查看目录是否输入错误\n");

}

}

**·echo**

void echo() {

printf("请输入您要打印的内容：");

char st[200];

fgets(st,200,stdin);//键盘录入echo<内容>

for(i=0;i<strlen(st);i++){

putchar(st[i]); //依次输出每个字符

}

memset(st,0,sizeof(st));

}

**·help**

void help(){//输出shell命令行接口下的所以可供选择的功能

printf("shell的基本功能介绍如下：\n");

printf(" cd<目录>：更改当前的目录环境到另一指定目录。\n");

printf(" echo<内容>:显示echo后的内容且换行。\n");

printf(" help:简短概要的输出你的shell的使用方法和基本功能。\n");

printf(" 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。\n");

printf(" quit,exit,bye:退出shell。\n");

}

**·jobs**

void jobs(){

pid\_t pid=fork();//创建进程就是为了方便辨别当前进程为子进程还是父进程

if(pid==0)//表示是子进程

{

execlp("ps","",NULL);//execlp(constchar\*file,constchar\*arg,...);

}//tree显示进程树，也就输出了所以的子进程

else if(pid>0)//表示父进程

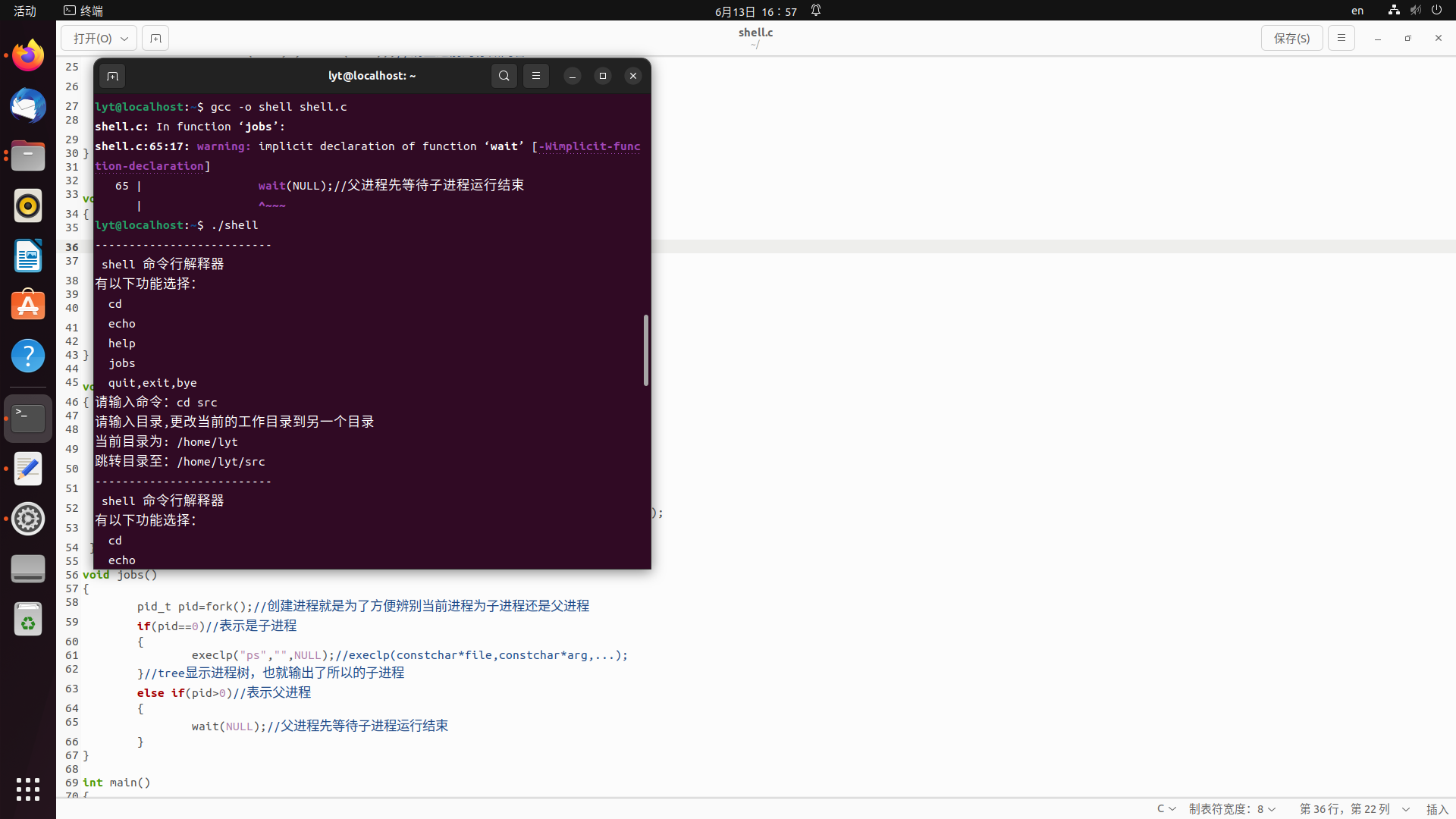
{

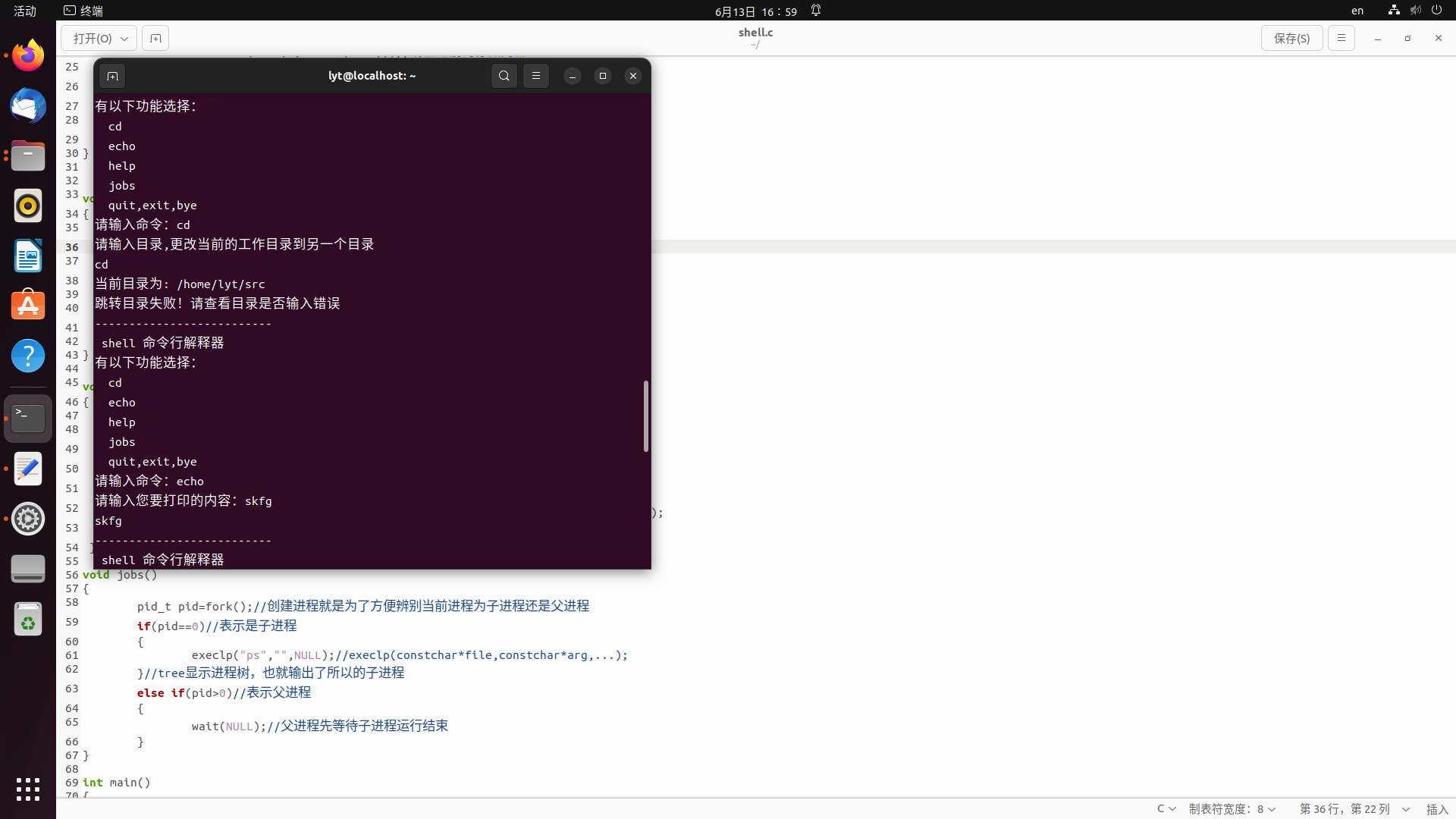
wait(NULL);//父进程先等待子进程运行结束

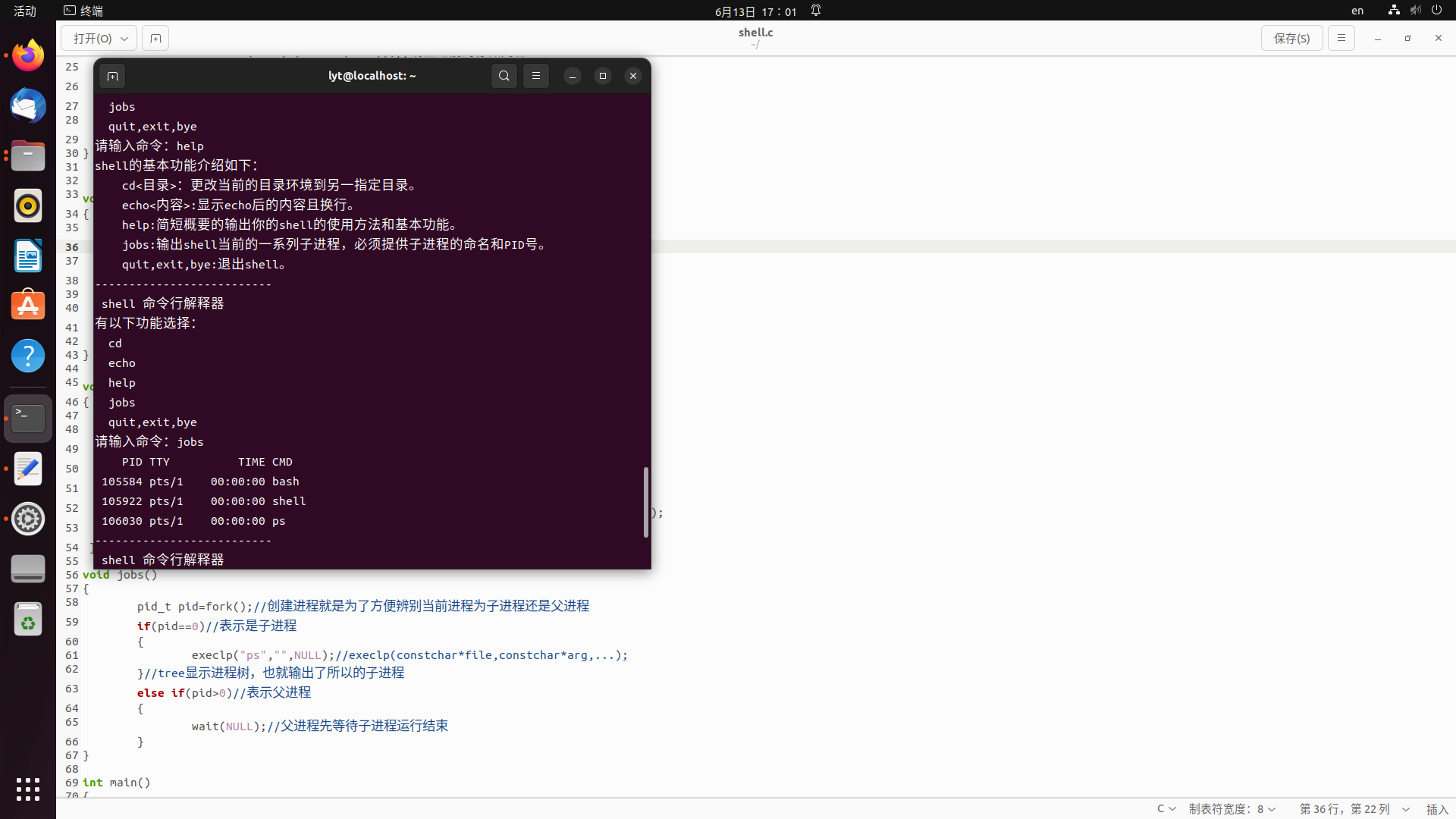
}

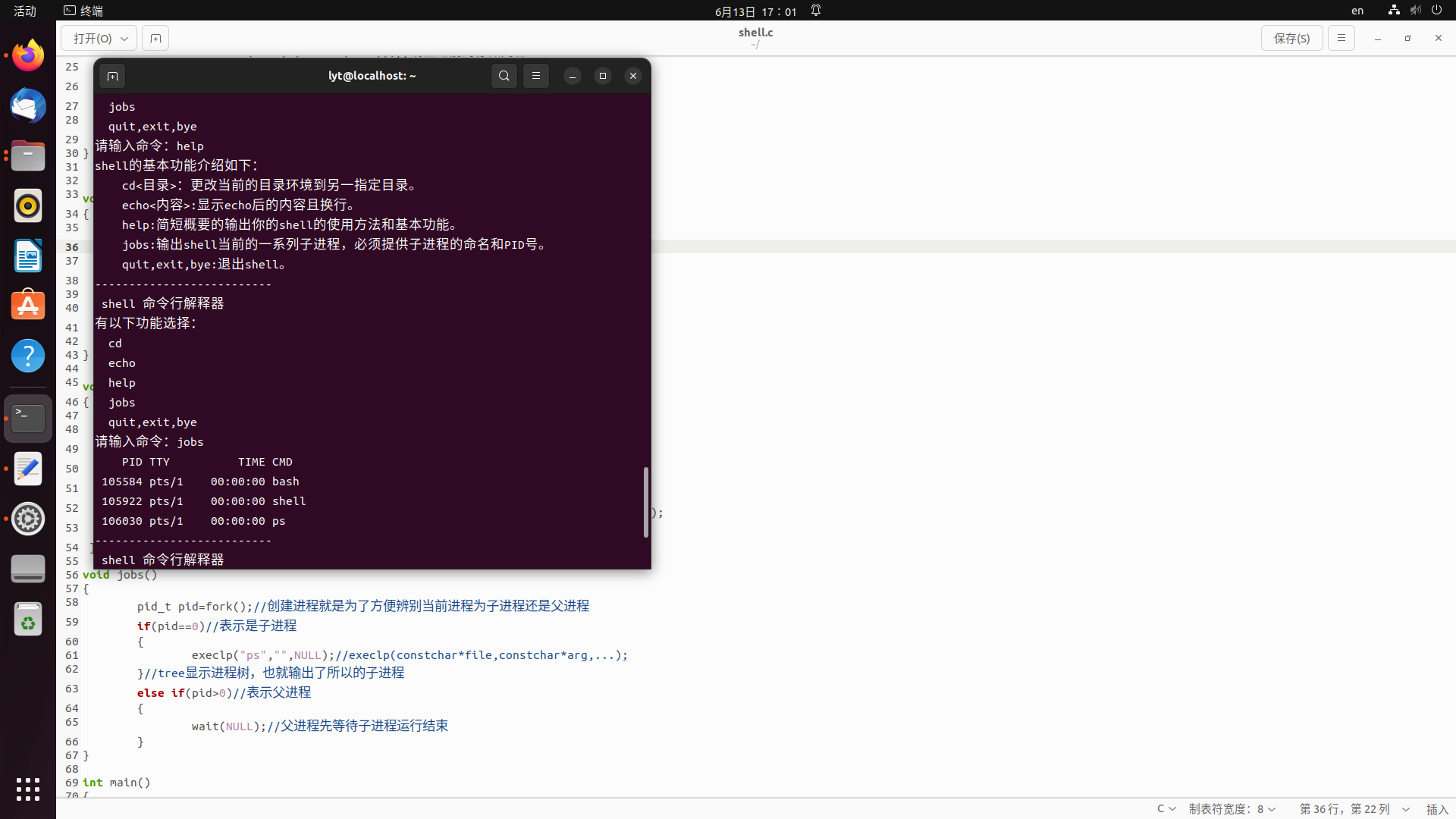
}

**五、实验结果与分析**

****

****

****

****

**六、小结与心得体会**

通过对实现简单的shell命令行解释器，我更加熟悉基本的 Linux 命令行操作：包括文件操作、进程管理、环境变量等，这是实现解释器所需要的基本知识。

熟悉进程的创建和管理：执行外部命令、管道、重定向等操作，需要熟悉进程的创建、管理和通信等基本原理。熟悉解释器的工作原理：解释器的核心是解析用户输入的命令，需要熟悉解释器的工作原理，包括解析命令行参数、执行外部命令、管道、重定向等操作，以及错误处理和信号处理等方面。