西安交通大学

操作系统专题实验报告

班级： 计算机2102

学号： 2215012469

姓名： 向胤兴

2023年 12 月 15 日

目 录

1 openEuler 系统环境实验

1.1进程相关实验

1.1.1 实验目的

1.1.2 实验内容

1.1.3 实验思想

1.1.4 实验步骤

1.1.5 测试数据设计

1.1.6 程序运行初值及运行结果分析

1.1.7 实验总结

1.1.7.1 实验中的问题与解决过程

1.1.7.2 实验收获

1.1.7.3 意见与建议

1.2线程相关实验

1.2.1 实验目的

1.2.2 实验内容

1.2.3 实验思想

1.2.4 实验步骤

1.2.5 测试数据设计

1.2.6 程序运行初值及运行结果分析

1.2.7 实验总结

1.2.7.1 实验中的问题与解决过程

1.2.7.2 实验收获

1.2.7.3 意见与建议

1.3自旋锁相关实验

1.3.1 实验目的

1.3.2 实验内容

1.3.3 实验思想

1.3.4 实验步骤

1.3.5 测试数据设计

1.3.6 程序运行初值及运行结果分析

1.3.7 实验总结

1.3.7.1 实验中的问题与解决过程

1.3.7.2 实验收获

1.3.7.3 意见与建议

1.4附件程序及README

2 进程通信与内存管理

2.1进程的软中断通信

2.1.1 实验目的

2.1.2 实验内容

2.1.3 实验思想

2.1.4 实验步骤

2.1.5 回答问题

2.2进程的管道间通信

2.2.1 实验目的

2.2.2 实验内容

2.2.3 实验思想

2.2.4 程序运行初值及运行结果分析

2.2.5 回答问题

2.3内存分配与回收

2.3.1 实验目的

2.3.2 实验内容

2.3.3 实验思想

2.3.4 程序运行初值及运行结果分析

2.3.5 回答问题

2.4 实验总结

2.5 附件程序及README

3 Liunx动态模块与设备驱动

3.1Liunx动态模块

3.1.1 实验目的

3.1.2 实验内容

3.1.3 实验思想

3.1.4 实验步骤

3.2Liunx设备驱动

3.2.1 实验目的

3.2.2 实验内容

3.2.3 实验思想

3.2.4 实验步骤

3.2.5 主要功能的实现方式

3.3实验总结

3.4附件程序及README

### openEuler 系统环境实验

#### 1.1进程相关编程实验

#### 1.1.1实验目的

（1）熟悉 Linux 操作系统的基本环境和操作方法，通过运行系统命令查看  
系统基本信息以了解系统；  
（2）编写并运行简单的进程调度相关程序，体会进程调度、进程间变量的管  
理等机制在操作系统实际运行中的作用

#### 1.1.2 实验内容

（1）熟悉操作命令、编辑、编译、运行程序。完成图 1-1 程序的运行验证，  
多运行几次程序观察结果；去除 wait 后再观察结果并进行理论分析。  
图 1-1 教材中所给代码（p103 作业 3.7）  
（2）扩展图 1-1 的程序：  
a） 添加一个全局变量并在父进程和子进程中对这个变量做不同操作，输出  
操作结果并解释；  
b） 在 return 前增加对全局变量的操作并输出结果，观察并解释；  
c） 修改程序体会在子进程中调用 system 函数和在子进程中调用 exec 族函  
数；

#### 1.1.3 实验原理

（1）进程：进程是计算机科学中的一个重要概念，它是操作系统中的基本执  
行单位。进程代表着一个正在执行的程序实例，它包括了程序的代码、数据和执  
行状态等信息。操作系统通过进程管理来实现对计算机资源的有效分配和控制；

（2）PID：PID 是进程标识符（Process Identifier）的缩写，它是用来唯  
一标识一个操作系统中的进程的数值。每个正在运行或已经终止的进程都会被分  
配一个唯一的 PID，这个标识符可以用来在操作系统内部识别和管理进程；

（3）fork()函数：fork() 是一个在类 Unix 操作系统中常见的系统调用，  
用于创建一个新的进程，新进程是原进程（父进程）的副本。新进程被称为子进  
程，它与父进程共享很多资源，但也有一些独立的属性。fork() 被用于实现多  
进程编程，常见于操作系统和并发编程中。函数返回一个整数，如果返回值为负  
数，则表示创建进程失败。如果返回值为 0，表示当前正在执行的代码是在子进  
程中。如果返回值大于 0，表示当前正在执行的代码是在父进程中，返回值是子  
进程的 PID。调用 fork() 函数时，操作系统会创建一个新的进程，该进程是调  
用进程的一个副本，称为子进程。子进程几乎与父进程相同，包括代码、数据、  
文件描述符等。但是子进程拥有自己的独立的内存空间和资源。

#### 1.1.4 实验步骤

本实验通过在程序中输出父、子进程的 pid，分析父子进程 pid 之间的关系，  
进一步加入 wait()函数分析其作用。  
步骤一：编写并多次运行图 1-1 中代码。

步骤二：删去图 1-1 代码中的 wait()函数并多次运行程序，分析运行结  
果。

步骤三：修改图 1-1 中代码，增加一个全局变量并在父子进程中对其进行  
不同的操作（自行设计），观察并解释所做操作和输出结果。  
  
步骤四：在步骤三基础上，在 return 前增加对全局变量的操作（自行设  
计）并输出结果，观察并解释所做操作和输出结果。  
  
步骤五：修改图 1-1 程序，在子进程中调用 system()与 exec 族函数。编写  
system\_call.c 文件输出进程号 PID，编译后生成 system\_call 可执行文

件。在子进程中调用 system\_call,观察输出结果并分析总结

#### 1.1.5 测试数据设计

本实验的五个步骤中，只有步骤3、4需要给全局变量设置初值。

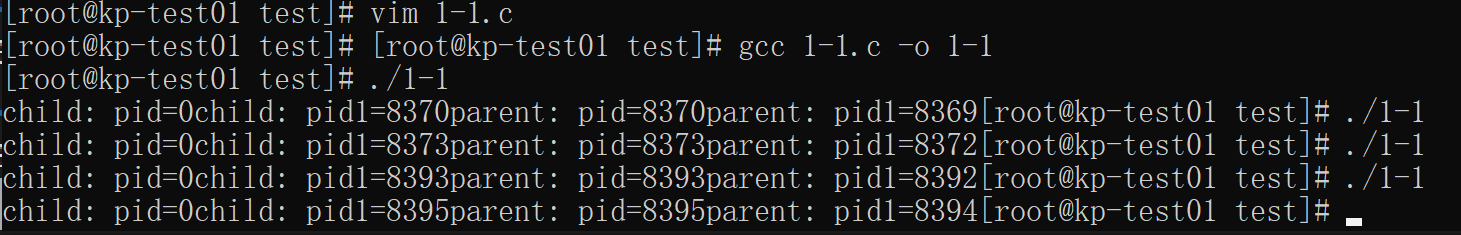
我在该实验中声明了int类型全局变量value，并初始化为0。在子进程中进行减一操作并打印子进程中value的值和其存储地址，在父进程中加一并同样打印其值和地址。

#### 1.1.6 程序运行初值及运行结果分析

步骤1:执行原始代码

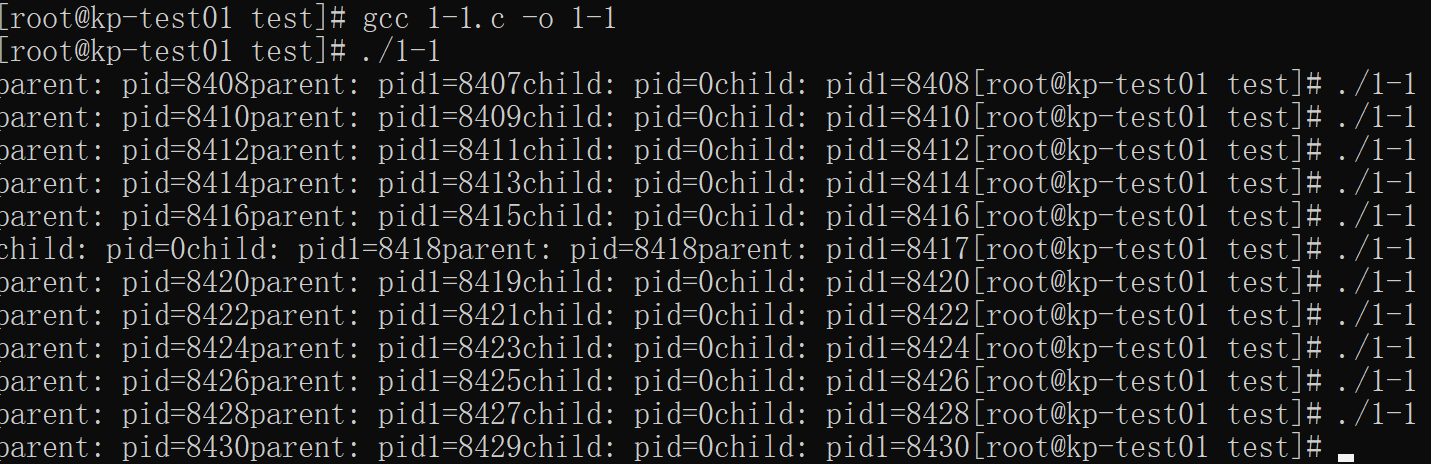
该步骤无需设置初值

运行结果：



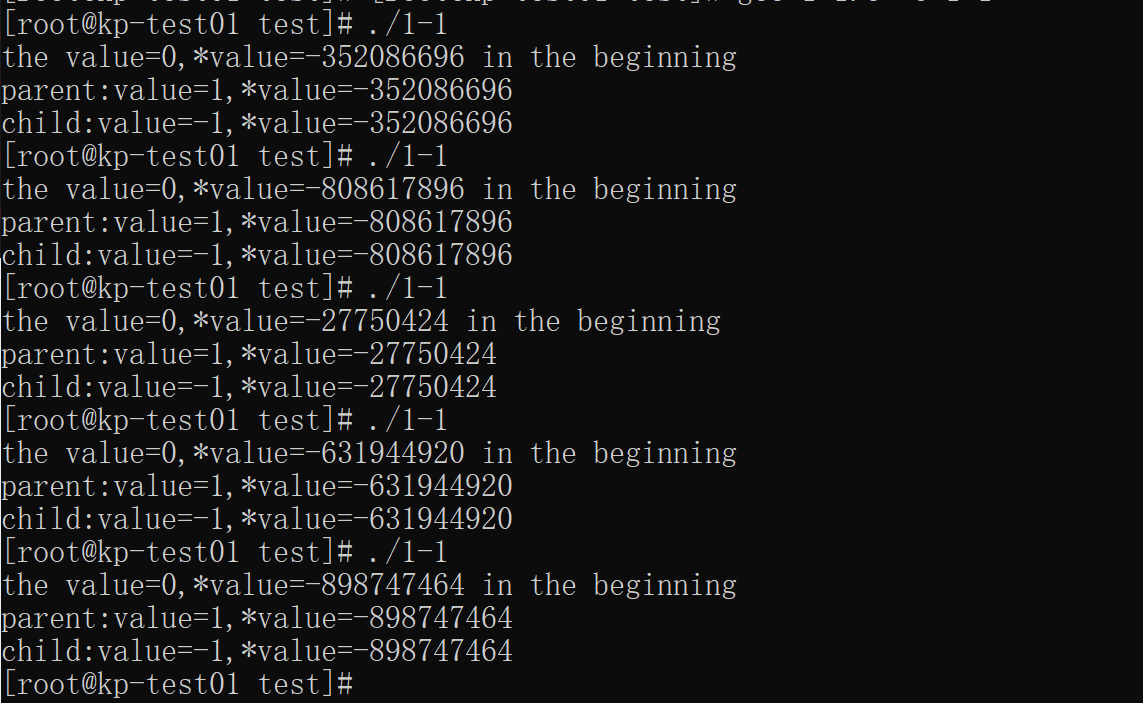
结果分析：pid\_t类型变量是用来标识进程的，每个进程都有它的pid。在父进程中调用pid=fork()分裂产生子进程时，子进程中pid的值会返回为0，而父进程中的pid的值返回为子进程的pid，在父、子进程中分别可以调用getpid()来返回当前进程的pid值。子进程比父进程后创建，对应的pid值比父进程的pid值大。

步骤2：删除父进程中的wait(NULL)

结果：

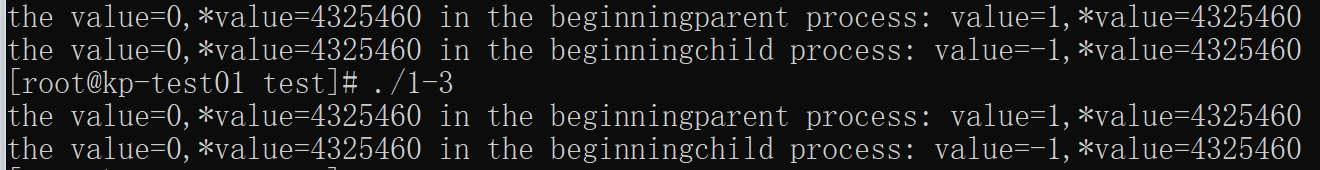
结果分析：多次执行文件，出现了一次先打印child后打印parent的情况，但经过分析，应该与是否删除wait(NULL)无关，因为wait(NULL)语句出现在打印内容之后，wait()不会影响打印顺序。

步骤3：父子进程分别对一全局变量进行不同操作，打印不同进程中全局变量的值和地址

结果：

结果分析：根据程序运行结果可以观察到父进程子进程中的全局变量的地址相同，但操作之间相互影响。

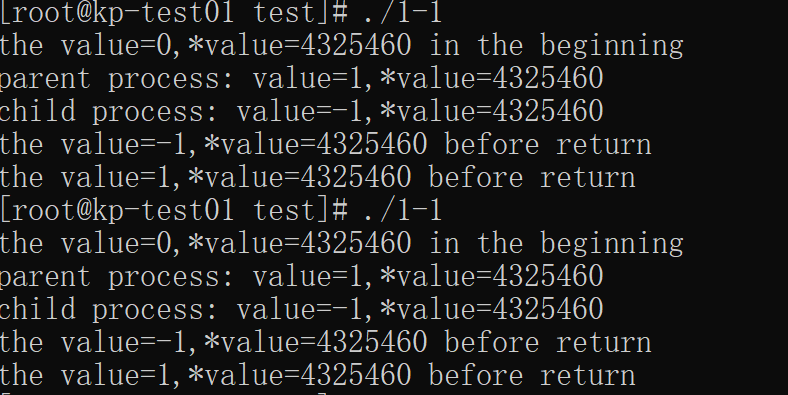
拓展：如果初始打印value时不加入换行符”\n”，fork之前的print内容会被打印两次。



原因是字符”\n”起到刷新缓冲区的作用，如果没有\n，在fork创建子进程时，在构建子进程的缓冲区时会复制一份父进程的缓冲区，会导致在fork之前的原本在父进程种的内容再次被打印一次，而打印内容种”\n”会在父进程打印刷新缓冲区，使得新得到的子进程的缓冲区为空，子进程就不会再打印一次。

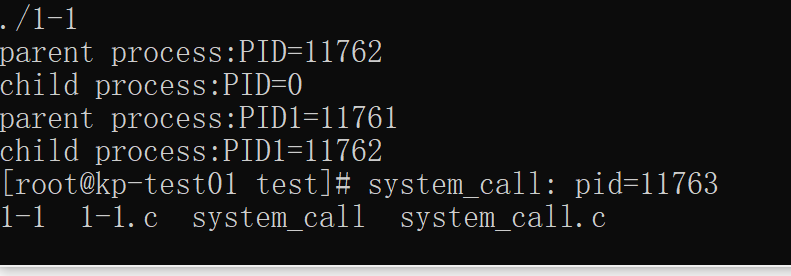
步骤4：在步骤3的基础上，在return之前打印全局变量value的值和地址

结果：



结果分析：由实验结果可知，父子进程都会进行一次return，并且在return前全局变量的值仍然为本进程处理后的值。

步骤5：在子进程中调用 system()与 exec 族函数

结果：

结果分析：在子进程中调用了system()和execlp()函数。根据结果可知，system()可以在进程中调用其他可运行文件，产生新的进程。而execlp()函数可以将指定的命令或代码用来替换本进程中剩余的代码，比如此例中用ls命令代替了剩下的代码，使得在execlp()之后的printf命令无法运行，运行结果中没有出现"this sentences won't be executed\n"。

#### 1.1.7 实验总结

###### 1.7.1 实验中的问题与解决过程

在步骤1的实验中，我产生了如下疑惑：

fork()是从原进程中分裂出了子进程（新增一个进程），还是kill原进程，新增两个进程，并使其一个为父进程一个为子进程？

于是我设计代码进行了实验，在调用fork()之前使用getpid()获得当前进程的pid值并且打印，根据此时pid值和fork之后父子进程之间的pid1值来探究它们之间的关系。

代码如下：#include<sys/types.h>

#include<sys/wait.h>

#include<stdio.h>

#include<unistd.h>

int main()

{

pid\_t pid,pid1;

pid=getpid();

printf("pid=%d before fork\n",pid);

pid=fork();

if (pid<0){/\*error occured\*/

fprintf(stderr,"Fork failed");

return 1;

}

else if(pid==0){/\*child process\*/

pid1=getpid();

printf("child pid=%d,pid1=%d\n",pid,pid1);

}

else{/\*Parent process\*/

pid1=getpid();

printf("parent pid=%d,pid1=%d\n",pid,pid1);

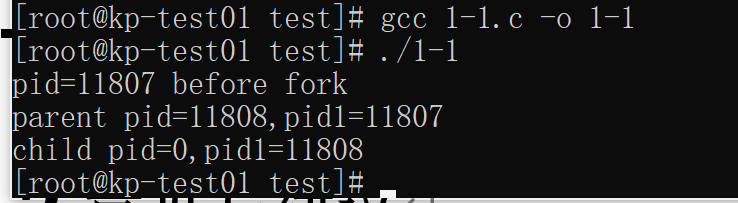
wait(NULL);

}

return 0;

}

实验结果如下：



可以观察到fork之后父进程的pid1与fork之前的原进程的pid值相同，可以得出结论fork之后的父进程即为原进程，只新产生了一个进程，即为子进程。

在步骤2的实验中，我认为是否删除在printf之后的wait(NULL)没有影响，wait(NULL)出现在parent process的printf之前会影响打印顺序，设计了如下代码验证：

#include<sys/types.h>

#include<sys/wait.h>

#include<stdio.h>

#include<unistd.h>

int main()

{

pid\_t pid,pid1;

pid=fork();

if (pid<0){/\*error occured\*/

fprintf(stderr,"Fork failed");

return 1;

}

else if(pid==0){/\*child process\*/

pid1=getpid();

printf("child pid=%d,pid1=%d\n",pid,pid1);

}

else{/\*Parent process\*/

wait(NULL);

pid1=getpid();

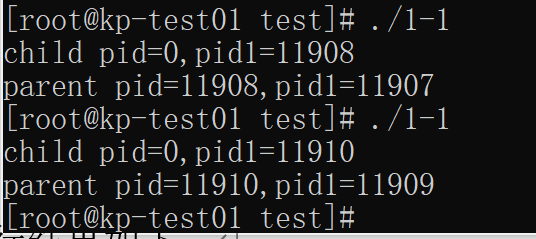
printf("parent pid=%d,pid1=%d\n",pid,pid1);

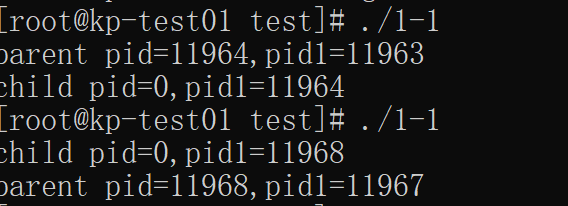
}

return 0;

}

运行结果如下：



删除parent process中的wait(NULL)后，

可能先出现parent也可能先出现child

###### 1.7.2 实验收获

通过本次实验，我更加深入的了解了fork()函数的运行原理，知道了父子进程pid值的关系。对多个进程对内存的管理有了更清晰直观的认知，了解全局变量在不同进程中是在同一个地址空间，但会分开操作保证互斥。

通过本次实验，让我了解了exec族的函数的作用，是用指定命令或代码来替换掉剩余的代码。了解了system()可以用来调用创建新进程其他的可运行文件。

总之，这次实验不光让我对多进程的内容有了更多的了解，还让我对各种对进程管理的函数有了更深入的认识。

###### 1.7.3 意见与建议

可以在指导书里简要说明一下每个步骤的目的是什么，是要验证什么原理，了解什么知识，让学生在进行实验的每一步时不光知道要写什么代码，还知道为什么要这样设计代码。

#### 1.2线程相关编程实验

#### 1.2.1实验目的

探究多线程编程中的线程共享进程信息。在计算机编程中，多线程是一种常  
见的并发编程方式，允许程序在同一进程内创建多个线程，从而实现并发执行。  
由于这些线程共享同一进程的资源，包括内存空间和全局变量，因此可能会出现  
线程共享进程信息的现象。本实验旨在通过创建多个线程并使其共享进程信息，  
以便深入了解线程共享资源时可能出现的问题。

#### 1.2.2 实验内容

（1）在进程中给一变量赋初值并成功创建两个线程；  
（2）在两个线程中分别对此变量循环五千次以上做不同的操作（自行设计）  
并输出结果；  
（3）多运行几遍程序观察运行结果，如果发现每次运行结果不同，请解释原  
因并修改程序解决，考虑如何控制互斥和同步；  
（4）将任务一中第一个实验调用 system 函数和调用 exec 族函数改成在线  
程中实现，观察运行结果输出进程 PID 与线程 TID 进行比较并说明原因。

#### 1.2.3 实验原理

本实验旨在通过创建两个线程，它们分别对一个共享的变量进行多次循环操

作，并观察在多次运行实验时可能出现的不同结果。在观察到结果不稳定的情况  
下，引入互斥和同步机制来确保线程间的正确协同操作。  
（1）线程创建与变量操作： 首先，在一个进程内创建两个线程，并在进程  
内部初始化一个共享的变量。这两个线程将并发地对这个共享变量进行循环操作，  
执行不同的操作。  
（2）竞态条件和不稳定结果： 由于线程并发执行，存在竞态条件，即两个  
线程可能同时读取和修改共享变量。在没有适当的同步措施的情况下，不同线程  
的操作可能会交叉执行，导致结果不稳定，每次运行可能都会得到不同的结果。  
（3）互斥与同步： 为了解决竞态条件带来的问题，可以使用互斥锁（Mutex）  
来保护共享变量的访问。在每个线程对变量进行操作之前，先获取互斥锁，操作  
完成后再释放锁。这样一来，每次只有一个线程能够访问变量，从而避免了并发  
访问带来的不稳定性。  
（4）观察结果与比较： 运行多次实验，观察使用互斥锁后的运行结果。应  
该可以发现，通过互斥锁的保护，不再出现不稳定的结果，每次运行得到的结果  
都是一致的。  
（5）调用系统函数和线程函数的比较： 在任务一中，如果将调用系统函数  
和调用 exec 族函数改成在线程中实现，观察运行结果。可以发现，调用系统函  
数和 exec 族函数时，会输出进程的 PID（Process ID），而在线程中运行时，会  
输出线程的 TID（Thread ID）。这是因为线程是进程的子任务，它们共享进程的  
资源，但有自己的执行流程。

### 1.2.4 实验步骤

步骤1：设计程序，创建两个子线程，两线程分别对同一个共享变量多次操  
作，观察输出结果。

步骤2：修改程序，定义信号量 signal，使用 PV 操作实现共享变量的访问  
与互斥。运行程序，观察最终共享变量的值。

步骤3：在第一部分实验了解了 system()与 exec 族函数的基础上，将这两  
个函数的调用改为在线程中实现，输出进程 PID 和线程的 TID 进行分析。

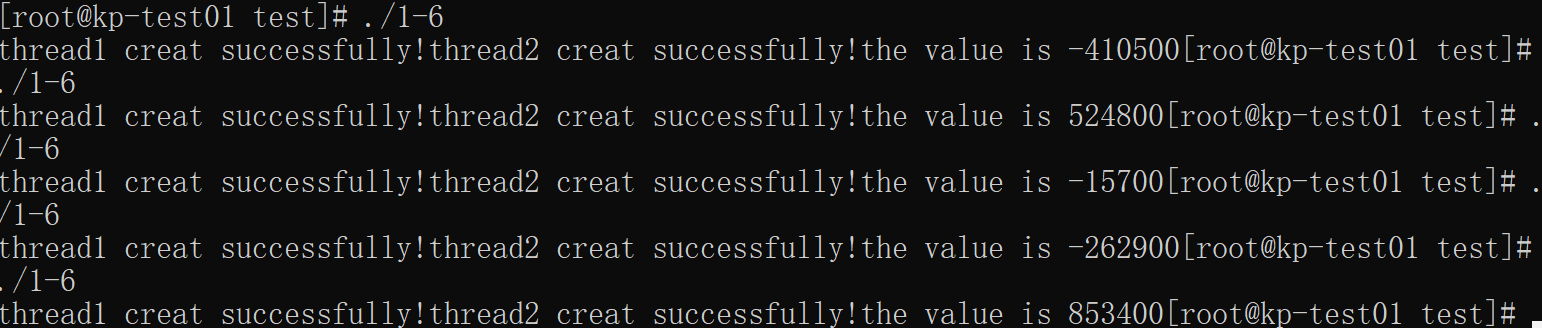
### 1.2.5 测试数据设计

在创建多线程前声明全局变量value并初始化其为0，再在后续多个线程中操作value。

### 1.2.6 程序运行初值及运行结果分析

步骤一：

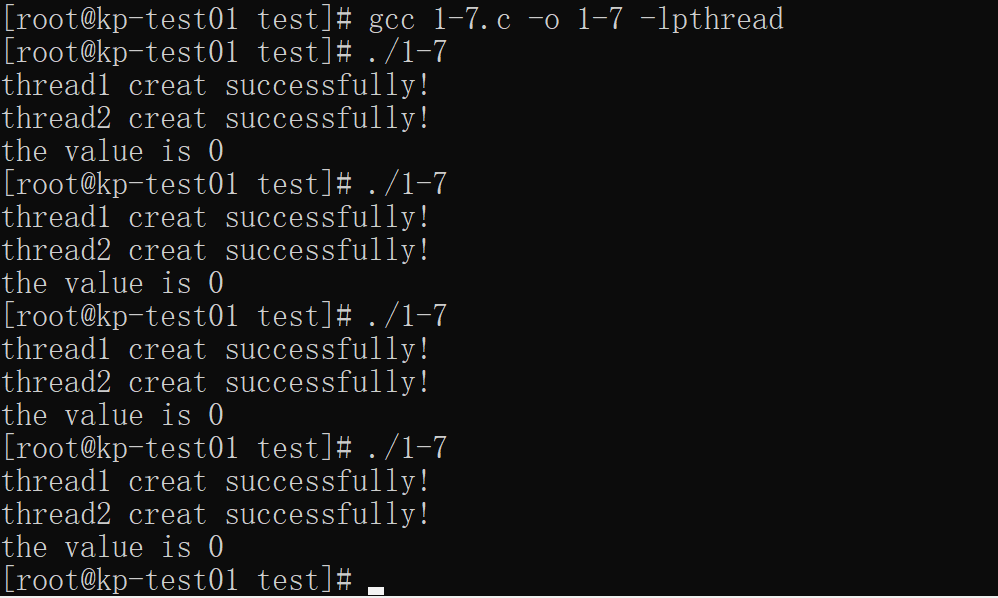
运行初值：value初值设为0，在thread1中进行100000次自增，在thread2中进行100000次自减



发现每次返回的value值不同，原因是两个线程同时对同一个变量进行操作导致value的值异常。

步骤二：

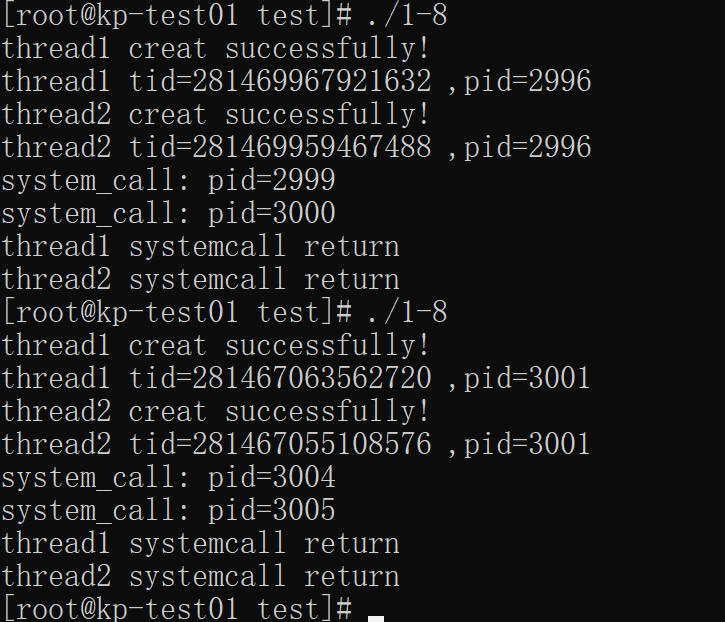
声明set\_t类型变量，使用sem\_init初始化，并使用sem\_wait(&set\_t)和sem\_post(&set\_t)分别作为P操作和V操作。实验结果如下：



根据实验结果，发现引入互斥机制后，value值最终回归正常。

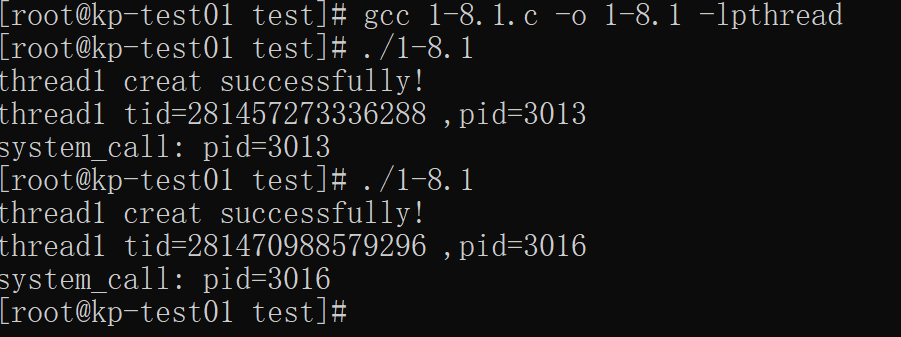
步骤3：

在两个线程中分别调用system(“./system\_call”)，运行结果如下



可以看到的pid值相同而tid值不同，说明此为两个属于相同进程的不同线程。在两个线程中分别调用了./system\_call，产生的进程的pid不同，说明两个线程可以分别创建不同的进程。

使用execlp()函数调用system\_call:



没有观察到thread2的内容，原因是thread1先创建，并且调用execlp()函数把其所在进程的在其之后的代码全部由system\_call文件中的内容覆盖，所以没有观察到thread2。

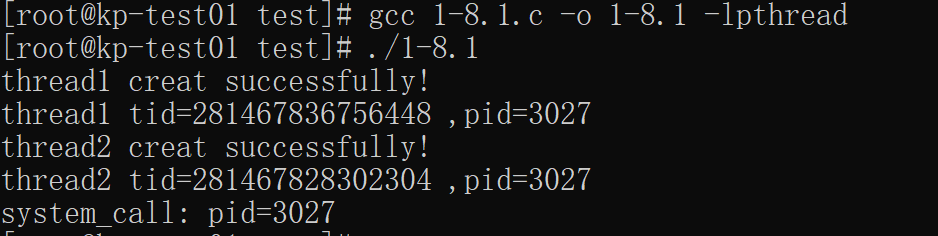
q

### 1.2.7 实验总结

#### 1.7.1 实验中的问题与解决过程

如何证明步骤3中使用execlp()时没有出现thread2的原因是还未来得及执行thread2的内容就被替换？为了验证，我进行了以下实验：

在thread1的execl()前添加sleep(10),实验结果如下图所示：



此时thread2创建成功并且成功打印，可以说明之前没有出现thread2的原因是还未创建就被替换。

#### 1.7.2 实验收获

本次实验中让我学会了怎样创建线程、结束线程。通过本次实验，对课本中的知识进行了验证，让我对进程和线程的共同点和差异性有了更深入和全面的认识。

本次实验让我更加熟练的使用system()和exec族的各种函数，从而在以后的各种实践编程中我能够更加自如的调用这两种函数。

## 1.3自旋锁实验

### 1.3.1实验目的

自旋锁作为一种并发控制机制，可以在特定情况下提高多线程程序的性  
能。本实验旨在通过设计一个多线程的实验环境，以及使用自旋锁来实现线程  
间的同步，从而实现以下目标：  
（1）了解自旋锁的基本概念： 通过研究自旋锁的工作原理和特点，深入  
理解自旋锁相对于其他锁机制的优势和局限性；  
（2）实验自旋锁的应用： 在一个多线程的实验环境中，设计一个竞争资  
源的场景，让多个线程同时竞争对该资源的访问；  
（3）实现自旋锁的同步： 使用自旋锁来保护竞争资源的访问，确保同一  
时间只有一个线程可以访问该资源，避免数据不一致和竞态条件；

### 1.3.2 实验内容

（1）在进程中给一变量赋初值并成功创建两个线程；

（2）在两个线程中分别对此变量循环五千次以上做不同的操作（自行设  
计）并输出结果；  
（3）使用自旋锁实现互斥和同步；

### 1.3.3 实验原理

自旋锁是一种基于忙等待（busy-waiting）的同步机制，用于在线程竞争  
共享资源时，不断尝试获取锁，而不是阻塞等待。它的工作原理可以简单地概  
括为以下几个步骤：  
（1）初始化锁： 自旋锁的开始是一个共享的标志变量（flag），最初为未  
锁定状态（0）。这个标志变量用于表示资源是否已被其他线程占用。  
（2）获取锁： 当一个线程尝试获取锁时，它会循环检查标志变量的状  
态。如果发现标志变量是未锁定状态（0），那么该线程将通过原子操作将标志  
变量设置为锁定状态（1），从而成功获取锁。如果标志变量已经是锁定状态，  
线程会一直在循环中等待，直到标志变量变为未锁定状态为止。  
（3）释放锁： 当持有锁的线程完成对共享资源的操作后，它会通过原子  
操作将标志变量设置回未锁定状态（0），从而释放锁，允许其他等待的线程尝  
试获取锁。  
自旋锁的工作原理中关键的部分在于“自旋”这一概念，即等待获取锁的  
线程会循环忙等待，不断检查标志变量的状态，直到能够成功获取锁。这种方  
式在锁的占用时间很短的情况下可以减少线程切换的开销，提高程序性能。

### 1.3.4 实验步骤

步骤1：根据实验内容要求，编写模拟自旋锁程序代码 spinlock.c

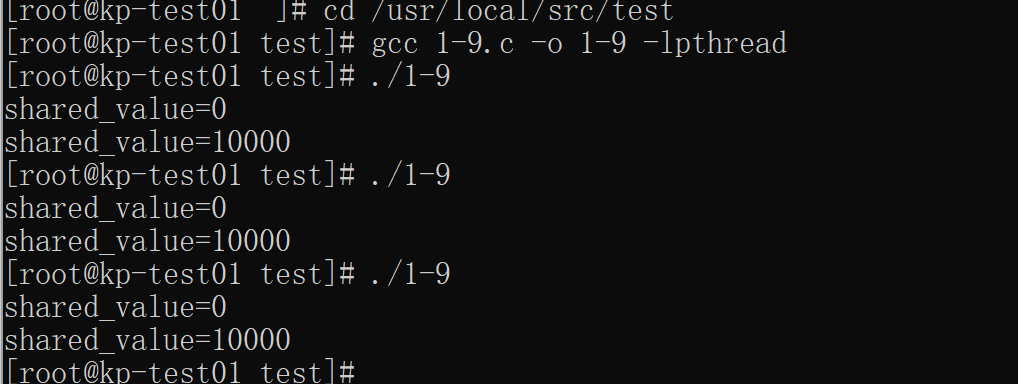
步骤2：运行补充后的文件

### 1.3.5 测试数据设计

初始时将shared\_value设置为0，两个线程分别对其进行500次加1操作。

### **1.3.6 程序运行初值及运行结果分析**

初始时将shared\_value设置为0，两个线程分别对其进行500次加1操作1.3.7 实验总结 ，实验结果如下：



由于两个进程中每次操作value值时，都申请了自旋锁，避免了同时操作数据，使得数据值不会发生异常。

### 1.3.7实验总结

### 实验收获

通过自行设计自旋锁，让我对自旋锁的结构，以及实现功能的原理有了更深入的了解。即在有一个线程操作多个线程共同申请操作的变量是，其他线程使用while()循环等待，直到占用资源的线程释放锁，某个线程获取访问变量的权限后再次上锁，如此往复。

实验1附件：

### 1.8附件：README以及代码：

1.1附件程序

步骤1程序：

###1-1.c###

#include<sys/types.h>

#include<sys/wait.h>

#include<stdio.h>

#include<unistd.h>

int main()

{

pid\_t pid,pid1;

pid=fork();

if (pid<0){/\*error occured\*/

fprintf(stderr,"Fork failed");

return 1;

}

else if(pid==0){/\*child process\*/

pid1=getpid();

printf("child: pid=%d",pid);//A

printf("chile: pid1=%d",pid1);//B

}

else{/\*Parent process\*/

pid1=getpid();

printf("parent: pid=%d",pid);//C

printf("parent: pid1=%d",pid1);//D

wait(NULL);

}

return 0;

}

步骤2程序：

###1-1.c###

#include<sys/types.h>

#include<sys/wait.h>

#include<stdio.h>

#include<unistd.h>

int main()

{

pid\_t pid,pid1;

pid=fork();

if (pid<0){/\*error occured\*/

fprintf(stderr,"Fork failed");

return 1;

}

else if(pid==0){/\*child process\*/

pid1=getpid();

printf("child: pid=%d",pid);//A

printf("chile: pid1=%d",pid1);//B

}

else{/\*Parent process\*/

pid1=getpid();

printf("parent: pid=%d",pid);//C

printf("parent: pid1=%d",pid1);//D

}

return 0;

}

步骤3程序：

###1-1.c###

#include<sys/types.h>

#include<sys/wait.h>

#include<stdio.h>

#include<unistd.h>

int value=0;

int main()

{

printf("the value=%d,\*value=%d in the beginning\n",value,&value);

pid\_t pid;

pid=fork();

if (pid<0){/\*error occured\*/

fprintf(stderr,"Fork failed");

return 1;

}

else if(pid==0){/\*child process\*/

value--;

printf("child process: value=%d,\*value=%d\n",value,&value);

}

else{/\*Parent process\*/

value++;

printf("parent process: value=%d,\*value=%d\n",value,&value);

wait(NULL);

}

return 0;

}

步骤4程序：

###1-1.c###

#include<sys/types.h>

#include<sys/wait.h>

#include<stdio.h>

#include<unistd.h>

int value=0;

int main()

{

printf("the value=%d,\*value=%d in the beginning\n",value,&value);

pid\_t pid;

pid=fork();

if (pid<0){/\*error occured\*/

fprintf(stderr,"Fork failed");

return 1;

}

else if(pid==0){/\*child process\*/

value--;

printf("child process: value=%d,\*value=%d\n",value,&value);

}

else{/\*Parent process\*/

value++;

printf("parent process: value=%d,\*value=%d\n",value,&value);

wait(NULL);

}

printf("the value=%d,\*value=%d before return\n",value,&value);

return 0;

}

步骤五程序：

###1-1.c###

#include<sys/types.h>

#include<stdio.h>

#include<unistd.h>

#include<sys/wait.h>

#include<stdlib.h>

int main()

{

pid\_t pid,pid1;

pid=fork();

if (pid<0){/\*error occured\*/

fprintf(stderr,"Fork failed");

return 1;

}

else if(pid==0){/\*child process\*/

pid1=getpid();

printf("child process:PID=%d\n",pid);

printf("child process:PID1=%d\n",pid1);

system("./system\_call");

execlp("ls","ls",NULL);

printf("this sentences won't be executed\n");

}

else{/\*Parent process\*/

pid1=getpid();

printf("parent process:PID=%d\n",pid);

printf("parent process:PID1=%d\n",pid1);

}

return 0;}

### system\_call.c ###

#include<stdio.h>

#include<unistd.h>

int main()

{

pid\_t pid;

pid=getpid();

printf("system\_call: pid=%d\n",pid);

return 0;

}

1.2 程序

步骤1程序

#include<stdio.h>

#include<stdlib.h>

#include<pthread.h>

int value=0;

void \*worker1(){

int i;

for(i=0;i<100000;i++)

value++;

pthread\_exit(NULL);

}

void \*worker2(){

int i;

for(i=0;i<100000;i++)

value--;

pthread\_exit(NULL);

}

int main(){

pthread\_t thread1,thread2;

int rc;

rc=pthread\_create(&thread1,NULL,worker1,NULL);

if(rc){

printf("ERROR; return code from creat thread1 is %d\n",rc);

exit(-1);

}

rc=pthread\_create(&thread2,NULL,worker2,NULL);

if(rc){

printf("ERROR; return code from creat thread2 is %d\n",rc);

exit(-1);

}

rc=pthread\_join(thread1,NULL);

if(rc){

printf("ERROR; return code from join thread1 is %d\n",rc);

exit(-1);

}

rc=pthread\_join(thread2,NULL);

if(rc){

printf("ERROR; return code from join thread2 is %d\n",rc);

exit(-1);

}

printf("the value is %d",value);

return 0;

}

步骤2程序：

#include<stdio.h>

#include<stdlib.h>

#include<pthread.h>

#include<semaphore.h>

int value = 0;

sem\_t sem;

void \*worker1(void \*arg) {

int i;

for (i = 0; i < 100000; i++) {

sem\_wait(&sem); // P操作

value++;

sem\_post(&sem); // V操作

}

pthread\_exit(NULL);

}

void \*worker2(void \*arg) {

int i;

for (i = 0; i < 100000; i++) {

sem\_wait(&sem); // P操作

value--;

sem\_post(&sem); // V操作

}

pthread\_exit(NULL);

}

int main() {

pthread\_t thread1, thread2;

int rc;

sem\_init(&sem, 0, 1); // 初始化信号量为1

rc = pthread\_create(&thread1, NULL, worker1, NULL);

if (rc) {

printf("ERROR; return code from creat thread1 is %d\n", rc);

exit(-1);

}

rc = pthread\_create(&thread2, NULL, worker2, NULL);

if (rc) {

printf("ERROR; return code from creat thread2 is %d\n", rc);

exit(-1);

}

rc = pthread\_join(thread1, NULL);

if (rc) {

printf("ERROR; return code from join thread1 is %d\n", rc);

exit(-1);

}

rc = pthread\_join(thread2, NULL);

if (rc) {

printf("ERROR; return code from join thread2 is %d\n", rc);

exit(-1);

}

printf("the value is %d\n", value);

sem\_destroy(&sem); // 销毁信号量

return 0;

}

步骤3程序：

使用system()的步骤3代码：

#include<stdio.h>

#include<stdlib.h>

#include<pthread.h>

#include<semaphore.h>

#include<unistd.h>

pid\_t pid;

void \*worker1(void \*arg) {

int i;

printf("thread1 creat successfully!\n");

pid=getpid();

pthread\_t tid=pthread\_self();

printf("thread1 tid=%d ,pid=%d\n",tid,pid);

system("./system\_call");

printf("thread1 systemcall return\n");

pthread\_exit(NULL);

}

void \*worker2(void \*arg) {

int i;

printf("thread2 creat successfully!\n");

pid=getpid();

pthread\_t tid=pthread\_self();

printf("thread2 tid=%d ,pid=%d\n",tid,pid);

system("./system\_call");

printf("thread2 systemcall return\n");

pthread\_exit(NULL);

}

int main() {

pthread\_t thread1, thread2;

int rc;

rc = pthread\_create(&thread1, NULL, worker1, NULL);

if (rc) {

printf("ERROR; return code from creat thread1 is %d\n", rc);

exit(-1);

}

rc = pthread\_create(&thread2, NULL, worker2, NULL);

if (rc) {

printf("ERROR; return code from creat thread2 is %d\n", rc);

exit(-1);

}

rc = pthread\_join(thread1, NULL);

if (rc) {

printf("ERROR; return code from join thread1 is %d\n", rc);

exit(-1);

}

rc = pthread\_join(thread2, NULL);

if (rc) {

printf("ERROR; return code from join thread2 is %d\n", rc);

exit(-1);

}

return 0;

}

使用execlp()的步骤3

#include<stdio.h>

#include<stdlib.h>

#include<pthread.h>

#include<semaphore.h>

#include<unistd.h>

pid\_t pid;

void \*worker1(void \*arg) {

int i;

printf("thread1 creat successfully!\n");

pid=getpid();

pthread\_t tid=pthread\_self();

printf("thread1 tid=%d ,pid=%d\n",tid,pid);

execlp("./system\_call","system\_call",NULL);

printf("thread1 systemcall return\n");

pthread\_exit(NULL);

}

void \*worker2(void \*arg) {

int i;

printf("thread2 creat successfully!\n");

pid=getpid();

pthread\_t tid=pthread\_self();

printf("thread2 tid=%d ,pid=%d\n",tid,pid);

execlp("./system\_call","system\_call",NULL);

printf("thread2 systemcall return\n");

pthread\_exit(NULL);

}

int main() {

pthread\_t thread1, thread2;

int rc;

rc = pthread\_create(&thread1, NULL, worker1, NULL);

if (rc) {

printf("ERROR; return code from creat thread1 is %d\n", rc);

exit(-1);

}

rc = pthread\_create(&thread2, NULL, worker2, NULL);

if (rc) {

printf("ERROR; return code from creat thread2 is %d\n", rc);

exit(-1);

}

rc = pthread\_join(thread1, NULL);

if (rc) {

printf("ERROR; return code from join thread1 is %d\n", rc);

exit(-1);

}

rc = pthread\_join(thread2, NULL);

if (rc) {

printf("ERROR; return code from join thread2 is %d\n", rc);

exit(-1);

}

return 0;

}

1.3 附件程序

#include <stdio.h>

#include <pthread.h>

#include<stdlib.h>

// 定义自旋锁结构体

typedef struct {

int flag;

} spinlock\_t;

// 初始化自旋锁

void spinlock\_init(spinlock\_t \*lock) {

lock->flag = 0;

}

// 获取自旋锁

void spinlock\_lock(spinlock\_t \*lock) {

while (\_\_sync\_lock\_test\_and\_set(&lock->flag, 1)) {

// 自旋等待

}

}

// 释放自旋锁

void spinlock\_unlock(spinlock\_t \*lock) {

\_\_sync\_lock\_release(&lock->flag);

}

// 共享变量

int shared\_value = 0;

// 线程函数

void \*thread\_function(void \*arg) {

spinlock\_t \*lock = (spinlock\_t \*)arg;

for (int i = 0; i < 5000; ++i) {

spinlock\_lock(lock);

shared\_value++;

spinlock\_unlock(lock);

}

return NULL;

}

int main() {

pthread\_t thread1, thread2;

spinlock\_t lock;

// 输出共享变量的值

printf("shared\_value=%d\n",shared\_value);

// 初始化自旋锁

spinlock\_init(&lock);

// 创建两个线程

int rc;

rc=pthread\_create(&thread1,NULL,thread\_function,(void \*)&lock);

if(rc){printf("ERROR in create thread1\n");

exit(-1);

}

rc=pthread\_create(&thread2,NULL,thread\_function,(void \*)&lock);

if(rc){printf("ERROR in create thread2\n");

exit(-1);

}

// 等待线程结束

rc=pthread\_join(thread1,NULL);

if(rc){

printf("ERROR in wait thread1\n");

exit(-1);

}

rc=pthread\_join(thread2,NULL);

if(rc){

printf("ERROR in wait thread2\n");

exit(-1);

}

// 输出共享变量的值

printf("shared\_value=%d\n",shared\_value);

return 0;

}

### 2、进程通信与内存管理

**2.1 进程的软中断通信**

**2.1.1 实验目的**

编程实现进程的创建和软中断通信，通过观察、分析实验现象，深入理解进程及进程在

调度执行和内存空间等方面的特点，掌握在 POSIX 规范中系统调用的功能和使用。

**2.1.2 实验内容**

（1）使用 man 命令查看 fork 、kill 、signal、sleep、exit 系统调用的帮助手册。

（2）根据流程图（如图 2.1 所示）编制实现软中断通信的程序：使用系统调用 fork()创建

两个子进程，再用系统调用 signal()让父进程捕捉键盘上发出的中断信号（即 5s 内按下

delete 键或 quit 键），当父进程接收到这两个软中断的某一个后，父进程用系统调用 kill()

向两个子进程分别发出整数值为 16 和 17 软中断信号，子进程获得对应软中断信号，然后分

别输出下列信息后终止：

Child process 1 is killed by parent !!

Child process 2 is killed by parent !!

父进程调用 wait()函数等待两个子进程终止后，输出以下信息，结束进程执行：

Parent process is killed!! 。

（3）多次运行所写程序，比较 5s 内按下 Ctrl+\或 Ctrl+Delete 发送中断，或 5s 内不进行

任何操作发送中断，分别会出现什么结果？分析原因。

（4）将本实验中通信产生的中断通过 14 号信号值进行闹钟中断，体会不同中断的执行样 式，从而对软中断机制有一个更好的理解

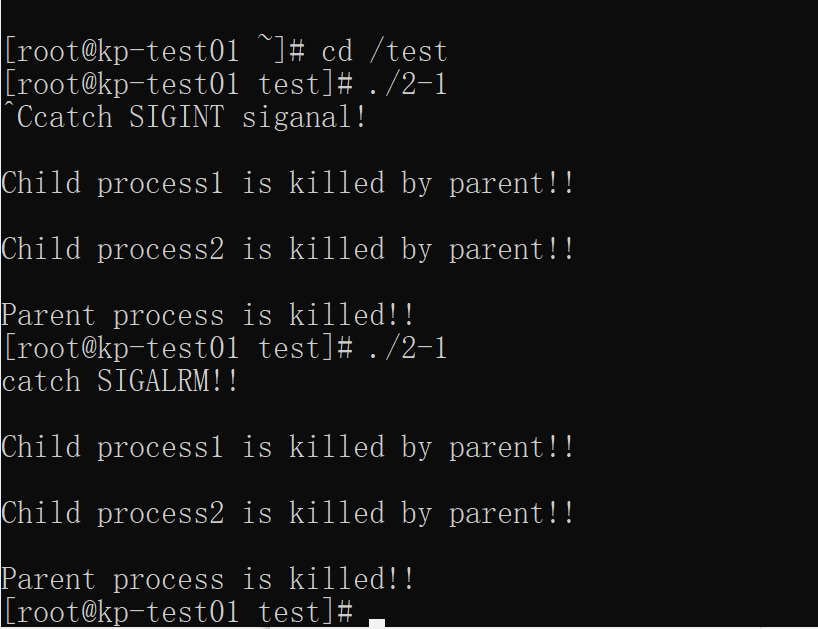
**2.1.3 实验思想**

通过合理使用kill、signal函数，实现进程间的控制与通信，实现特定进程收到特定信号后做出特定行为。

**2.1.4 程序运行初值及运行结果分析**

本实验无需要设置的程序运行初值。

结果及其分析



注：第一个为手动输入ctrl+c信号实现中断，第二个为5秒后被时钟信号打断实现中断。

结果分析：两种情况下父进程都在子进程结束返回之后才结束。

问题回答：

1. 你最初认为运行结果会怎么样？写出你猜测的结果。

答：最初认为的运行结果与我猜测的一致，即两个子进程结束以后父进程才结束。

1. 实际的结果什么样？有什么特点？在接收不同中断前后有什么差别？请将 5 秒内中断 和 5 秒后中断的运行结果截图，试对产生该现象的原因进行分析。

答：实际结果即：不管父进程捕捉到SIGINT还是SIGALRM信号，都是子进程结束返回之后父进程才结束

1. 改为闹钟中断后，程序运行的结果是什么样子？与之前有什么不同？

答：为了区分闹钟中断和键盘键入的SIGINT中断，在设计程序时，父进程捕捉到不同的信号的会在屏幕上打印不同的内容不同。被键盘键入的SIGINT信号打断会在终端屏幕输出”catch SIGINT signal!”，被闹钟中断会在终端屏幕输出”catch SIGALRM!!”。

1. kill 命令在程序中使用了几次？每次的作用是什么？执行后的现象是什么？

答：在我的代码中kill命令使用了4次。

第1、2次是为了保证父进程向子进程发送信号时，如何确保子进程已经准备好接收信号。父进程在创建完两个子进程后进入连续两次的挂起状态，而两个新创建的进程会在创建成功后向父进程发送信号，唤醒父进程，在父进程被完全唤醒后，就能保证此时子进程一定以及创建成功了。执行后的现象是父进程的闹钟信号开始计时

第3、4次是父进程向子进程发送中断信号，结束子进程。执行后的现象是两个子进程结束并在屏幕上打印自己结束的信息。

1. 使用 kill 命令可以在进程的外部杀死进程。进程怎样能主动退出？这两种退出方式哪种更好一些？

答：进程可以在程序内部的return或者exit实现主动退出。两种退出方式各有其好处和使用场景：

主动退出：进程内部主动退出能保证进程在按照预期完成工作后才退出，而不会因为被打断还没有完成希望完成的工作就被打杀死。

外部控制：某些时候我们希望能够通过一些外部进程来达到控制另一些进程的效果，例如在本实验中，希望在父进程收到中断信号后去中断结束掉子进程，这种情况下kill的灵活性就体现出来了。

2.1.5 回答问题

(1)父进程向子进程发送信号时，如何确保子进程已经准备好接收信号？

答：子进程完成signal绑定后，通过kill向父进程发送SIGUSR信号，打断父进程的pause()挂起，父进程才执行之后的命令，保证父进程向子进程发送信号时，子进程已经准备好接收信号。

(2)如何阻塞住子进程，让子进程等待父进程发来信号？

答：使用while(flag)pause();初始时flag=1，只有接收到父进程发来的特定信号才会使flag=0，退出循环。使用pause()可以避免忙等待，加快处理速度。

**2.2 进程的管道通信**

**2.2.1 实验目的**

编程实现进程的管道通信，通过观察、分析实验现象，深入理解进程管道通信的特点，

掌握管道通信的同步和互斥机制。

**2.2.2 实验内容**

（1）学习 man 命令的用法，通过它查看管道创建、同步互斥系统调用的在线帮助，并阅读

参考资料。

（2）根据流程图（如图 2.2 所示）和所给管道通信程序，按照注释里的要求把代码补充完

整，运行程序，体会互斥锁的作用，比较有锁和无锁程序的运行结果，分析管道通信是如何

实现同步与互斥的。

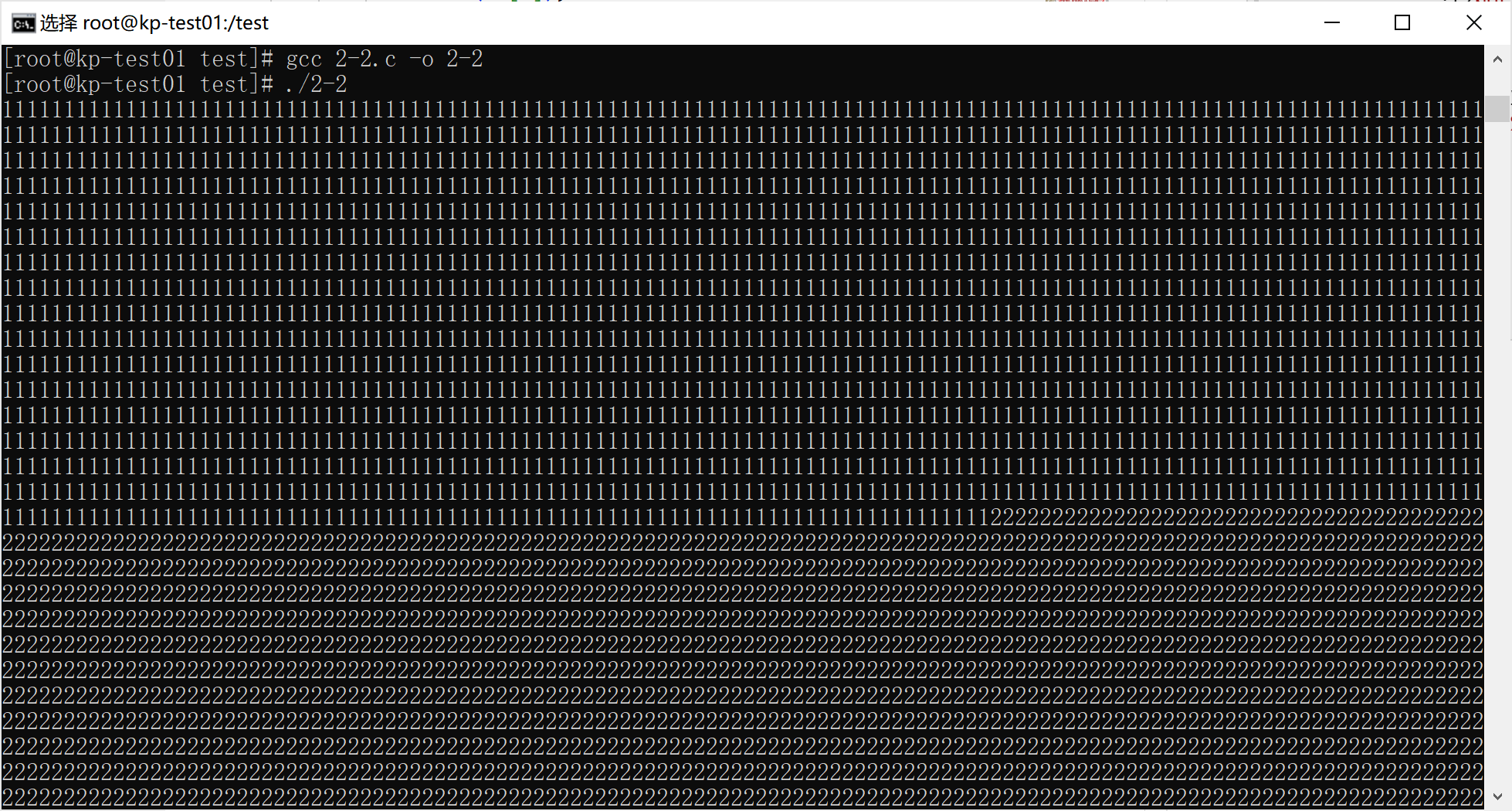
**2.2.3 实验思想**

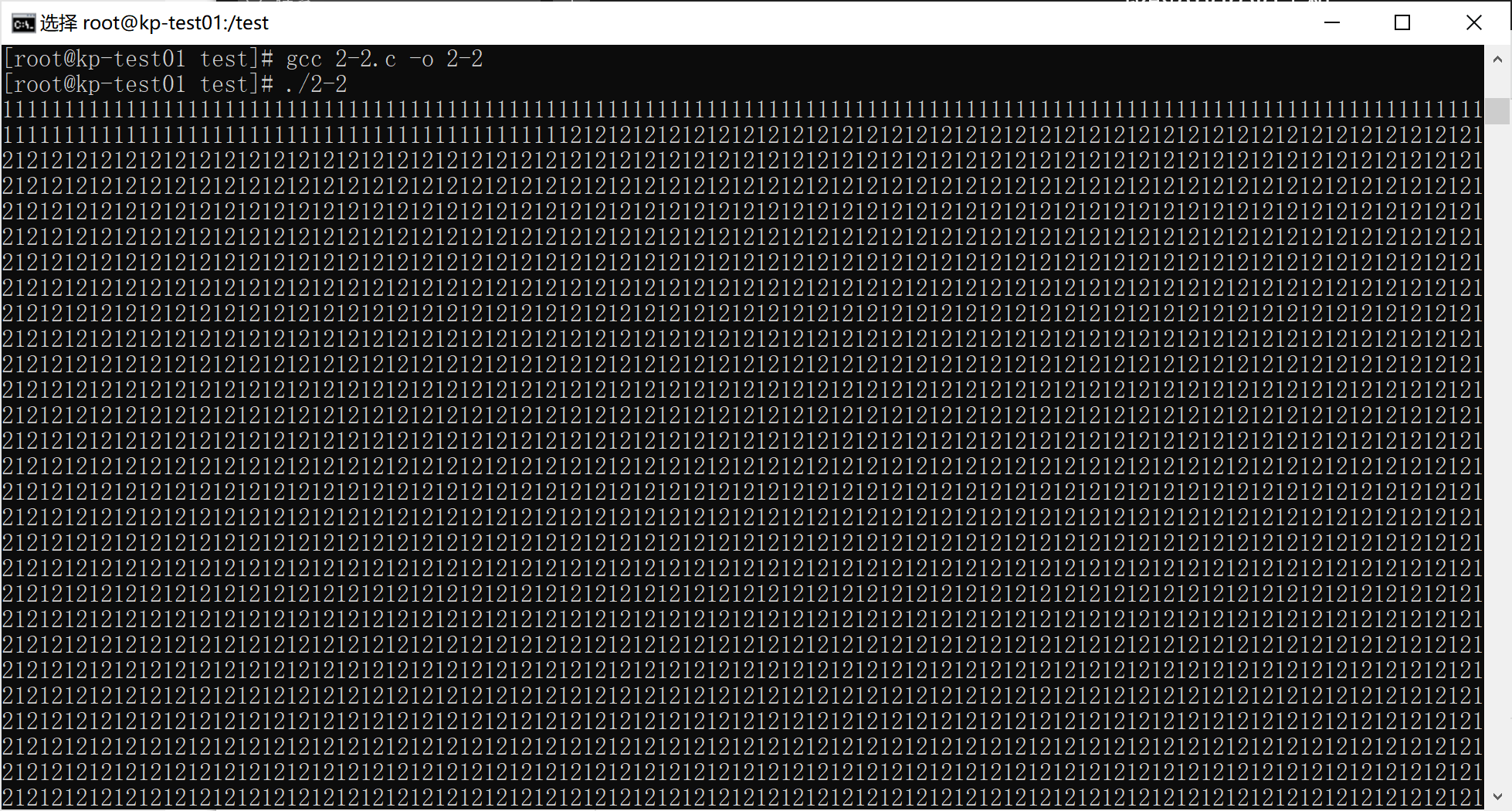
通过pipe实现管道通信，两个进程共同向管道写入数据。观察不同互斥锁情况下管道中数据的情况。

**2.2.4 程序运行初值及运行结果分析**

程序设计向管道中写入的数据量不能太少，太少可能导致先创建的进程在另一个进程开始向管道中写入数据之前就已经把数据全部写入了，导致互斥锁没有起作用。

结果分析：





实验问题的回答：

1. 你最初认为运行结果会怎么样？

答：我最初也认为不加互斥锁的情况下，两进程会交替向管道中写入数据。但是没有想到两个进程是严格的轮换输入数据的。思考原因可能是因为管程在设计的时候偏好于把权限分配给之前没有被分配到的进程。

1. 实际的结果什么样？有什么特点？试对产生该现象的原因进行分析。

答：在两个进程都申请互斥锁的情况下，先申请到互斥锁的进程写完自己所有的数据后，另一个进程才会向管道中写入。在两个进程只有一个申请互斥锁或者都不申请互斥锁的情况下，先创建的进程在另一个写入进程还没有做写入之前做连续输入，等到另一个进程也开始向管道中写入后，两进程交叉写入。这是由于没有互斥锁来保证进程的连续写入，两进程只能交叉使用管道的写入端。

1. 实验中管道通信是怎样实现同步与互斥的？如果不控制同步与互斥会发生什么后果？

答：进程在使用管道前使用lockf(fd[1],1,0)对管道上锁，使用结束后使用lockf(fd[1],0,0)对管道进行解锁。如果不控制同步互斥，则会导致写入管道的信息混乱，从管道的读出端无法读出有效数据。（如一个单词word从管道写入端写入，接收方收到的w/o/r/d四个字母中间夹杂着很多其他消息，就无法正确收到word这个单词了。）

**2.3内存分配与回收**

**2.3.1 实验目的**

通过设计实现内存分配管理的三种算法（FF，BF，WF），理解内存分配及回收的过程及

实现思路，理解如何提高内存的分配效率和利用率。

**2.3.2 实验内容**

（1）理解内存分配 FF，BF，WF 策略及实现的思路。

（2）参考给出的代码思路，定义相应的数据结构，实现上述 3 种算法。每种算法要实现内

存分配、回收、空闲块排序以及合并、紧缩等功能。

（3）充分模拟三种算法的实现过程，并通过对比，分析三种算法的优劣。

**2.3.3 实验思想**

通过数据结构的管理模拟内存的分配与回收机制与过程。

**2.3.4 程序运行初值及运行结果分析**

struct allocated\_block \*allocated\_block\_head= NULL; //进程分配内存块链表的首指针

int free\_block\_count= 0;    //空闲块的个数

int mem\_size= DEFAULT\_MEM\_SIZE;   //内存大小

int current\_free\_mem\_size= 0;     //当前空闲内存大小

int ma\_algorithm= MA\_FF;          //当前分配算法，初始值设为First Fit

int flag= 0;       //设置内存大小标志，表示内存大小是否设置

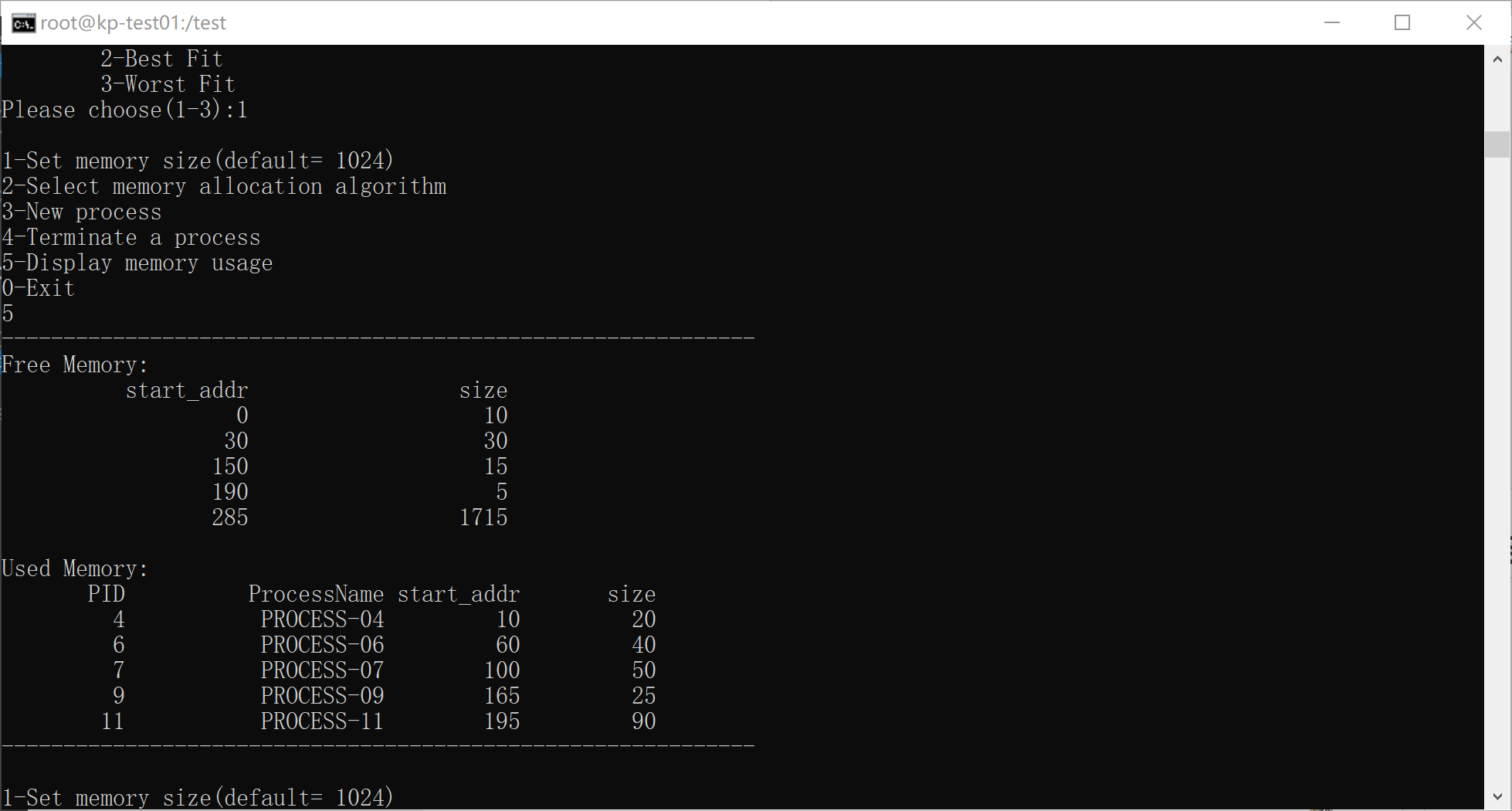
运行结果分析：

实验结果：

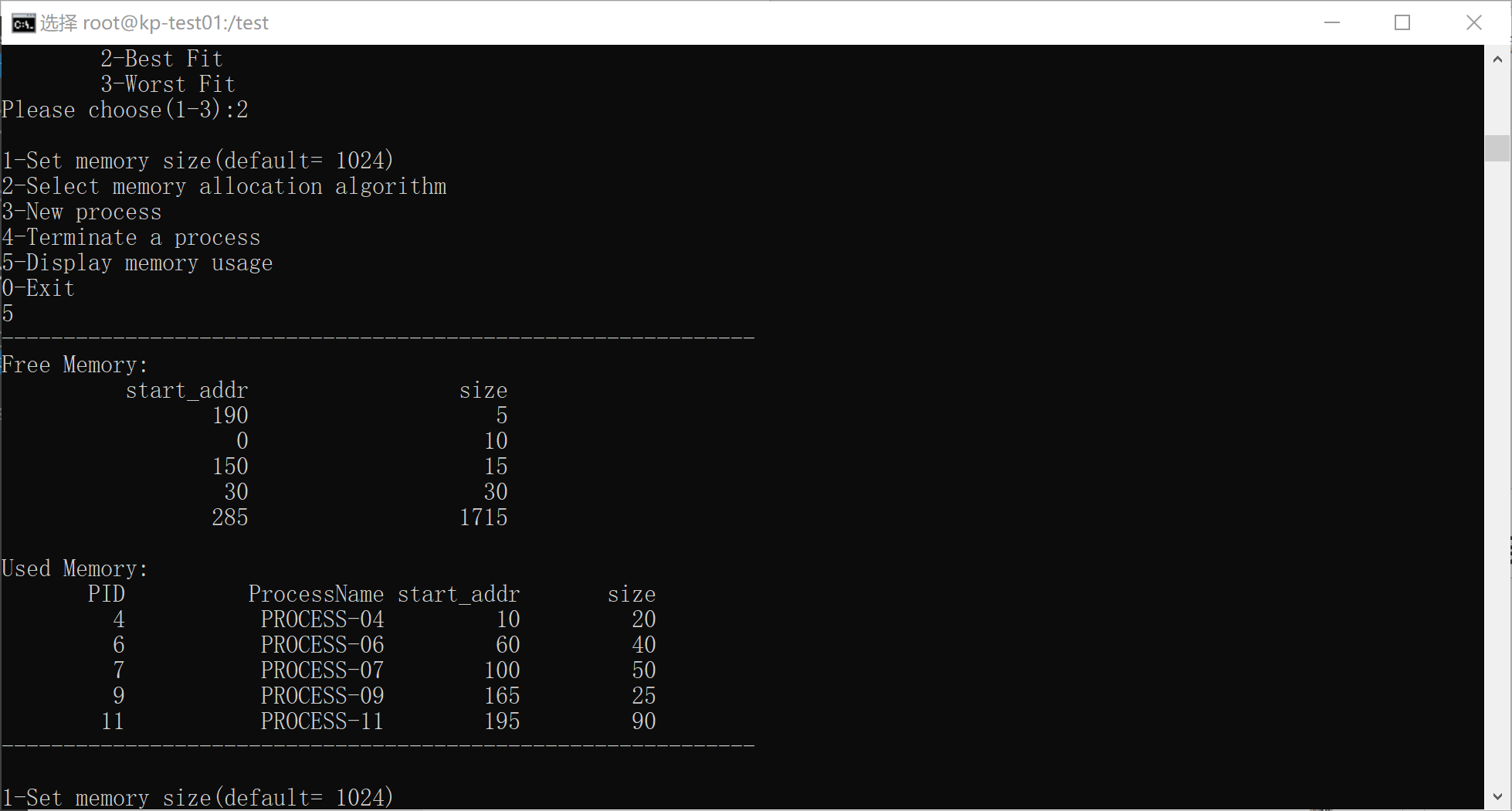
设计分配算法为总选择空闲块队列里第一个符合要求的空闲块，通过对空闲队列进行不同的排序从而实现不同的选择策略。

同样的空闲块在不同算法下不同的组织情况：

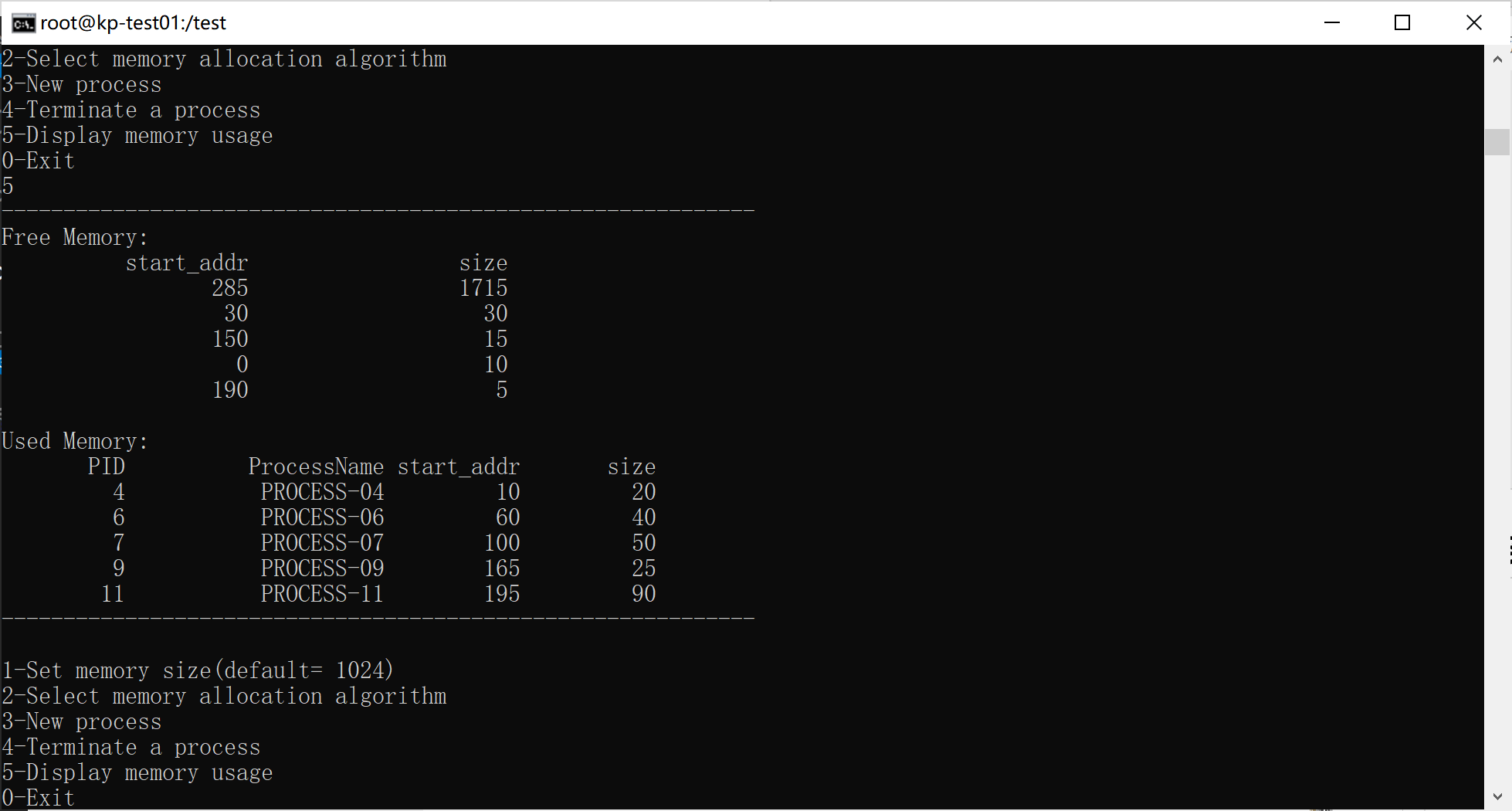
First Fit算法，按起始地址从小到大组织空闲块：



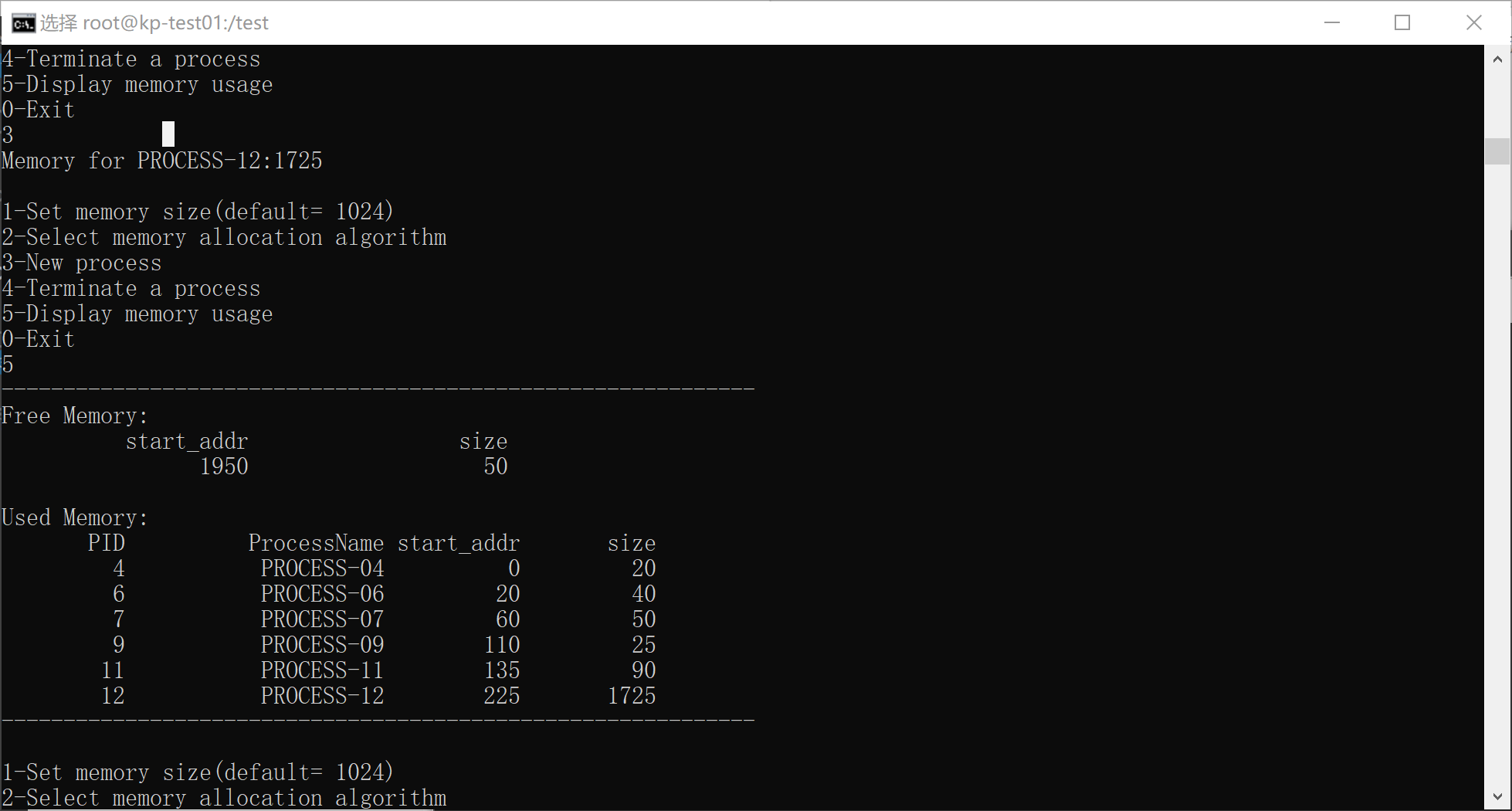
Best Fit算法，按空闲块的大小从小到大组织空闲块：



Worst Fit算法，按空闲块的大小从大到小排序：

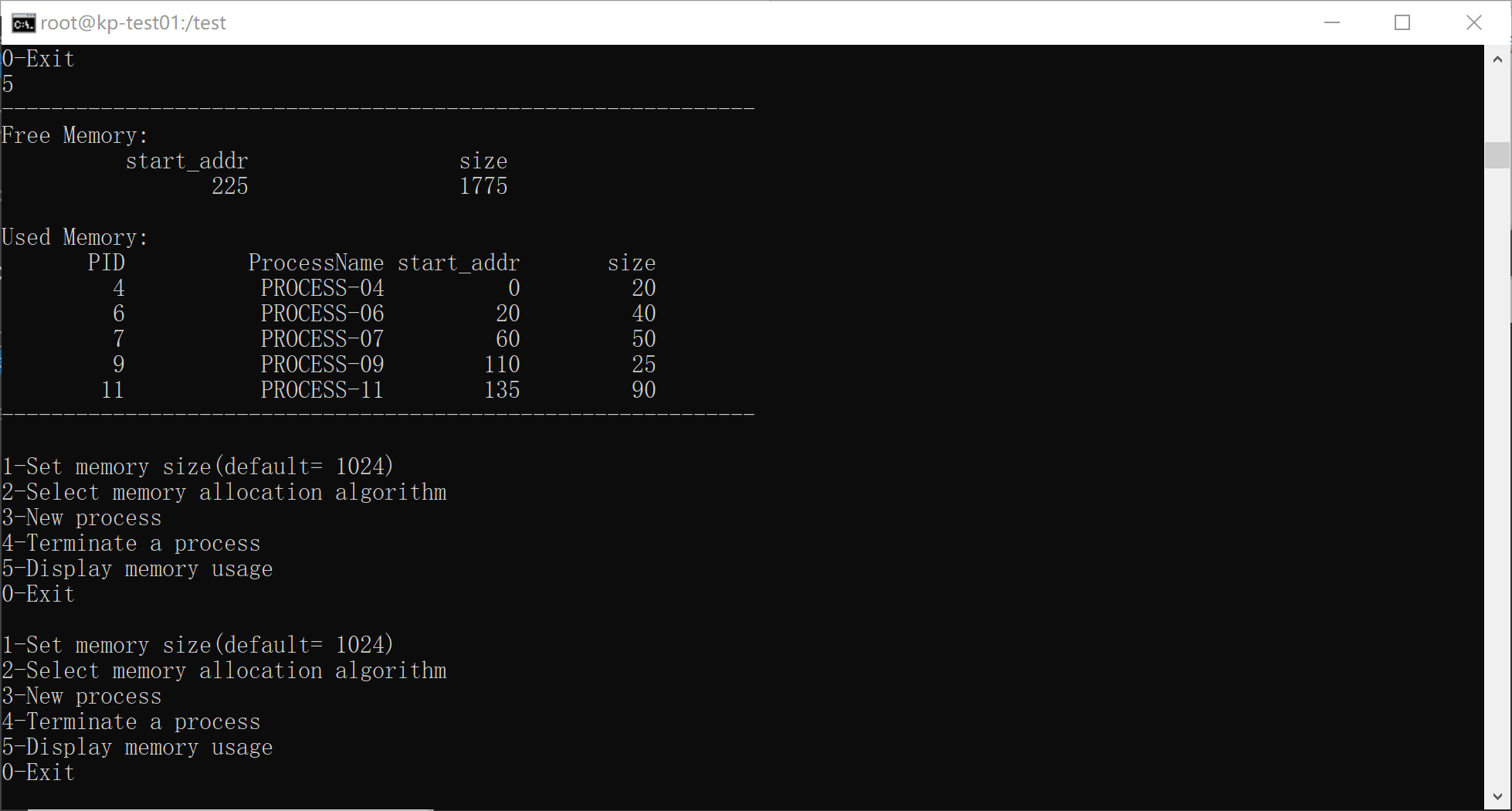


内存紧缩，当没有足够大的单个空闲块分配但总空闲空间还足以分配时候启用内存紧缩机制：



可以看到在新创建了一个大小为1725的进程之后，由于原本最大的空闲块只有1715，不足以分配，但总空闲空间足够分配，所以采取内存紧缩机制，内存紧缩后的内存分配情况如上。

进程结束归还内存时，合并空闲块：



进程12结束后，归还内存空间，归还的内存空间与原本的大小为50的空闲块相邻，两空闲块合并为一个大的空闲块。

**2.3.5回答问题**

（1）对涉及的 3 个算法进行比较，包括算法思想、算法的优缺点、在实现上如何提高算法的查找性能。

答： 算法思想：

FF---从起始地址开始第一个有足够大空间的空闲块的空间分配内存。

BF---从所有大于分配进程要求空间空闲块中选择最小的空闲块进行分配

WF---从所有大于分配进程要求空间空闲块中选择最大的空闲块进行分配

算法的优缺点：

FF，实现方式最简单，但是没有按照实际方式来管理，可能造成很多外碎片的情况。

BF，总是尽可能分配最小的空闲块的空间，可以保证最大空闲块的内存不会被消耗，有较大进程需要分配时有大的空闲块进行分配，但使用BF算法可能导致零散的外碎片较多或者产生很多内碎片的问题。

WF，总是尽可能分配最大的空闲块的空间，与BF算法相反，尽可能避免的零散的外碎片和内碎片，但当许多占空间不大的小进程分配后，可能导致没有足够大的空闲块给较大进程分配空间。

（2）3 种算法的空闲块排序分别是如何实现的。

答：

First Fit：空闲块按起始地址从小到大组织

Best Fit：空闲块按空闲块大小从小到大组织

Worst Fit：空闲块按空闲块大小从大到小组织

（3）结合实验，举例说明什么是内碎片、外碎片，紧缩功能解决的是什么碎片。

答： 内碎片是分配给进程但没有使用的空间，外碎片是虽然没有分配给进程，但是由于本身太小不足以以单个块给任何一个进程分配空间，导致了实际上也无法被使用的空闲空间。

举例说明，例如本实验要求，当给进程分配空闲块的空间时，如果分配后的空闲块剩余内存小于10，则直接分配给进程，尽管该空间进程不会使用。

例如，把一个大小为25的空闲块分配给一个大小为20的进程，由于25-20=5<10，则把整个空闲块的所有25空间都分配给进程，但实际使用中进程不会使用5个空间，导致5个空间被浪费了这就是内碎片。

再例如，把一个大小为35的空闲块分配给一个大小为20的进程，由于35-20=15>10，10的空间不会被分配给这个进程，仍然以一个空闲块的方式被管理再空闲块链表中。但是，由于整个空闲块太小了，基本不存在任何一个进程可以使用这个大小为15的空闲块，导致这个空间被浪费了，这就是外碎片。

（4）在回收内存时，空闲块合并是如何实现的？

答：每当有进程结束释放分配空间块时，先在空闲块链表中加入新的空闲块，起始地址和大小即为原分配块的起始地址和大小。然后按FF算法管理空闲块链表，从头到尾检查每一个空闲块，自己的起始地址＋长度?=下一个块的起始地址，如果等于，则在链表中删除下一个空闲块，更新各类指针，并且把本空闲块的长度更新为原本此空闲块长度与下一个空闲块长度之和。注意，针对最后一个空闲块不能做此操作，否则会导致越界。从头到尾检索一次以后，就可以保证所有相邻的块都被合并了，此时还需要返回原本的组织算法ma\_algorithm，保证后续其他操作的正常进行。

除了实验指导书的提问以外，以下还有几个就我认为有价值的问题

如何实现动态调整多种分配算法？

答：在创建进程时候，总是先分配空闲块链表中第一个足够大的空闲块，选择不同的分配算法，实质上是按照不同的管理方式管理空闲链表。然后总是用链表从头到尾的第一个大于目标分配进程的要求大小的空闲块来进行分配。

如何实现内存紧缩机制？

答：首先需要记录内存还有多少空间可以分配，因为必须要是没有任何单个空闲块可以满足要求且总的空闲空间可以满足要求的情况下，才会出现内存紧缩情况，所以记录总空闲空间的大小的是有必要的。在判断需要进行内存紧缩机制的情况下，删除原本所有的空闲块管理信息，然后修改已分配进程的队列，使他们连续排序，直至最后一个已分配进程的结束位置开始，作为其实位置分配给新创建的进程，最后剩下的空间即为剩下的最后的唯一一个空闲块。

**2.4实验总结**

**实验中遇到的问题与解决过程**

在进行内存的分配与回收实验中，一开始时总是没有注意好指针越界的问题，总是访问NULL->next导致报错。最后先用vscocde调试才找到自己出错的地方。

**实验收获**

了解了软中断这一Linux进程间的通信方式，了解了管程这一不同进程之间发送与接收消息的机制。收获最大的内存分配与回收实验，锻炼了代码能力，此外更加深入的了解到了内存管理空闲空间的方式，最内存分配、内碎片与外碎片、FF/WF/BF算法等诸多概念有了更深入的了解。

**2.5附件：代码及README2**

**# 2.1 进程的软中断通信**

**## 代码**

```C

#include <stdio.h>

#include <unistd.h>

#include <sys/wait.h>

#include <stdlib.h>

#include <signal.h>

#include<time.h>

int flag = 1;

pid\_t pid1 = -1, pid2 = -1;

void inter\_handler() {

    if(pid1>0&&pid2>0)

    {

    printf("\ncatch SIGINT siganal!\n");

    kill(pid1,16);

    kill(pid2,17);

    }

}

void alrm\_handler() {

    // TODO

    printf("\ncatch SIGALRM!!\n");

        kill(pid1, 16);

        kill(pid2, 17);

}

void inter\_handler1(){

    printf("\nChild process1 is killed by parent!!\n");

    flag=0;

}

void inter\_handler2(){

    printf("\nChild process2 is killed by parent!!\n");

    flag=0;

}

void child\_handler(int signum) {

    // 子进程收到 SIGUSR1 信号表示已经准备好

    if (signum == SIGUSR1) {

        printf("\nChild1 process is ready to receive signals.\n");

    }

    else if(signum == SIGUSR2){

         printf("\nChild2 process is ready to receive signals.\n");

    }

}

int main() {

    signal(SIGUSR1, child\_handler);

    signal(SIGUSR2, child\_handler);

    signal(SIGINT, inter\_handler);

    signal(SIGQUIT, inter\_handler);

    while (pid1 == -1) {

        pid1 = fork();

    }

    if (pid1 > 0) {//父进程

        while (pid2 == -1)

     {

            pid2 = fork();

          }

        if (pid2 > 0) {//父进程

    pause();

    pause();

    alarm(5);

    signal(SIGALRM, alrm\_handler);

    wait(NULL);

    wait(NULL);

    printf("\nParent process is killed!!\n");

        }

    else {//pid 2 子进程

        sleep(1);

        signal(17,inter\_handler2);

        kill(getppid(),SIGUSR2);

        while(flag)pause();

        return 0;

        }

    }

        else

    {//pid1子进程

    sleep(2);

    signal(16,inter\_handler1);

    kill(getppid(),SIGUSR1);

    while(flag)pause();

    return 0;

        }

    return 0;

}

```

**### 多次运行所写程序，比较 5s 内按下 Ctrl+\或 Ctrl+Delete 发送中断或 5s 内不进行任何操作发送中断，分别会出现什么结果？分析原因。**

运行结果如下

![2-1](2-1.png)

如果在5s内按下Ctrl+\或Ctrl+Delete发送中断，会先打印catch SIGNIT siganl！后，子进程被结束然后父进程结束，说明是通过inter\_handler()结束的子进程。

如果在5s内没有进行任何中断发送操作，alarm(5)会在父进程创建5s后自动向父进程发送SIGALRM信号，通过alrm\_handler()结束子进程。

**### 父进程向子进程发送信号时，如何确保子进程已经准备好接收信号？**

子进程完成signal绑定后，通过kill向父进程发送SIGUSR信号，打断父进程的pause()挂起，父进程才执行之后的命令，保证父进程向子进程发送信号时，子进程已经准备好接收信号。

**### 如何阻塞住子进程，让子进程等待父进程发来信号？**

使用while(flag)pause();初始时flag=1，只有接收到父进程发来的特定信号才会使flag=0，退出循环。使用pause()可以避免忙等待，加快处理速度。

**# 2.2 进程的管道通信**

**## 代码**

```c

/\*管道通信实验程序完整版 \*/

#include <unistd.h>

#include <signal.h>

#include <stdio.h>

#include<stdlib.h>

#include<sys/wait.h>

int pid1, pid2; // 定义两个进程变量

int main( ) {

    int fd[2];

    char InPipe[4000]; // 定义读缓冲区

    char c1 = '1', c2 = '2';

    pipe(fd); // 创建管道

    while ((pid1 = fork( )) == -1); // 如果进程 1 创建不成功,则空循环

    if (pid1 == 0) { // 如果子进程 1 创建成功, pid1 为进程号

        close(fd[0]);

    lockf(fd[1],1,0);

        for (int i = 0; i < 2000; i++) {

            write(fd[1], &c1, 1); // 分 2000 次每次向管道写入字符 '1'

        }

        sleep(5); // 等待读进程读出数据

    lockf(fd[1],0,0);

        close(fd[1]);

        exit(0); // 结束进程 1

    } else {

        while ((pid2 = fork( )) == -1); // 若进程 2 创建不成功,则空循环

        if (pid2 == 0) {

            close(fd[0]);

            lockf(fd[1], 1, 0);

            for (int i = 0; i < 2000; i++) {

                write(fd[1], &c2, 1); // 分 2000 次每次向管道写入字符 '2'

            }

            sleep(5);

            lockf(fd[1], 0, 0);

            close(fd[1]);

            exit(0);

        } else {

            close(fd[1]);

            wait(NULL); // 等待子进程 1 结束

            wait(NULL); // 等待子进程 2 结束

            int bytesRead = read(fd[0], InPipe, 4000); // 从管道中读出 4000 个字符

            InPipe[bytesRead] = '\0'; // 加字符串结束符

            printf("%s\n", InPipe); // 显示读出的数据

            close(fd[0]);

            exit(0); // 父进程结束

        }

    }

return 0;

}

```

两进程都上锁

![all lock](2-2alllock.png)

完全互斥访问

只有一个进程上锁或者都不上锁

![not lock](2-2notlock.png)

**# 2.3内存分配与回收**

```c

#include <stdio.h>

#include <malloc.h>

#include <stdlib.h>

#include <unistd.h>

#define PROCESS\_NAME\_LEN 32    //进程名长度

#define MIN\_SLICE 10           //最小碎片的大小

#define DEFAULT\_MEM\_SIZE 1024  //内存大小

#define DEFAULT\_MEM\_START 0    //起始位置

//内存分配算法

#define MA\_FF 1

#define MA\_BF 2

#define MA\_WF 3

//描述每一个空闲块的数据结构

struct free\_block\_type

{

    int size;        //空闲块大小

    int start\_addr;  //空闲块起始位置

    struct free\_block\_type \*next;  //指向下一个空闲块

};

//指向内存中空闲块链表的首地址

struct free\_block\_type \*free\_block= NULL;

//每个进程分配到的内存块的描述

struct allocated\_block

{

    int pid;

    int size;  //进程大小

    int start\_addr;   //进程分配到的内存块的起始地址

    char process\_name[PROCESS\_NAME\_LEN];  //进程名

    struct allocated\_block \*next;   //指向下一个进程控制块

};

//进程分配内存块链表的首指针

struct allocated\_block \*allocated\_block\_head= NULL;

int free\_block\_count= 0;    //空闲块的个数

int mem\_size= DEFAULT\_MEM\_SIZE;   //内存大小

int current\_free\_mem\_size= 0;     //当前空闲内存大小

int ma\_algorithm= MA\_FF;          //当前分配算法

static int pid= 0;

int flag= 0;       //设置内存大小标志，表示内存大小是否设置

struct free\_block\_type\* init\_free\_block(int mem\_size);

void display\_menu();

int set\_mem\_size();

void set\_algorithm();

void rearrange(int algorithm);

int rearrange\_WF();

int rearrange\_BF();

int rearrange\_FF();

int new\_process();

int allocate\_mem(struct allocated\_block \*ab);

void kill\_process();

int free\_mem(struct allocated\_block \*ab);

int dispose(struct allocated\_block \*free\_ab);

int display\_mem\_usage();

struct allocated\_block\* find\_process(int pid);

int do\_exit();

int allocate(struct free\_block\_type \*pre,struct free\_block\_type \*allocate\_free\_nlock,struct allocated\_block \*ab);

int mem\_retrench(struct allocated\_block \*ab);

int main(){

 char choice; pid=0;

 free\_block = init\_free\_block(mem\_size); //初始化空闲区

 while(1) {

 display\_menu(); //显示菜单

 fflush(stdin);

 choice=getchar(); //获取用户输入

 switch(choice){

 case '1': set\_mem\_size(); break; //设置内存大小

 case '2': set\_algorithm();flag=1; break;//设置算法

 case '3': new\_process(); flag=1; break;//创建新进程

 case '4': kill\_process(); flag=1; break;//删除进程

 case '5': display\_mem\_usage(); flag=1; break; //显示内存使用

 case '0': do\_exit(); exit(0);//释放链表并退出

 default: break; } }

 return 0;

}

//初始化空闲块，默认为一块，可以指定大小及起始地址

struct free\_block\_type\* init\_free\_block(int mem\_size)

{

    struct free\_block\_type \*fb;

    fb= (struct free\_block\_type\*)malloc(sizeof(struct free\_block\_type));

    if(fb== NULL)

    {

        printf("No mem\n");

        return NULL;

    }

    current\_free\_mem\_size= mem\_size;

    fb->size= mem\_size;

    fb->start\_addr= DEFAULT\_MEM\_START;

    fb->next= NULL;    //首地址指向NULL

    free\_block\_count=1;

    return fb;

}

//设置内存的大小

int set\_mem\_size()

{

    int size;

    if(flag!= 0)     //防止重复设置

    {

        printf("you've already set the memory size\n");

        return 0;

    }

    printf("Total memory size=");

    for(;;)

    {

        scanf("%d",&size);

        getchar();

        if(size> 0)

        {

            current\_free\_mem\_size= size;

            mem\_size= size;

            free\_block->size= mem\_size;

            break;

        }

        else

        {

            printf("The size must greater than zero!Please input again:");

        }

    }

    flag= 1;    //内存大小已经设置

    return 1;

}

//设置当前的分配算法

void set\_algorithm()

{

    int algorithm;

    //system("clear");

    printf("\t1-First Fit\n");   //首次适应算法

    printf("\t2-Best Fit\n");    //最佳适应算法

    printf("\t3-Worst Fit\n");   //最坏适应算法

    printf("Please choose(1-3):");

    for(;;)

    {

        scanf("%d",&algorithm);

        getchar();

        if(algorithm>= 1&&algorithm<= 3)

        {

            ma\_algorithm= algorithm;

            break;

        }

        else

        {

            printf("\nCannot input %d,Please input 1-3:",algorithm);

        }

    }

    rearrange(ma\_algorithm);

}

void rearrange(int algorithm)

{

    switch(algorithm)

    {

        case MA\_FF:

            rearrange\_FF();

            break;

        case MA\_BF:

            rearrange\_BF();

            break;

        case MA\_WF:

            rearrange\_WF();

            break;

    }

}

//按FF算法重新整理内存空闲块链表，按空闲块首地址排序

int rearrange\_FF()

{

    struct free\_block\_type \*head= free\_block;

    struct free\_block\_type \*forehand,\*pre,\*rear;

    int i;

    if(head== NULL)

        return -1;

    for(i= 0;i< free\_block\_count-1;i++)

    {

        forehand= head;

        pre= forehand->next;

        rear= pre->next;

        if(forehand== head&&forehand->start\_addr>= pre->start\_addr)

            {

                //比较空闲链表中第一个空闲块与第二个空闲块的开始地址的大小

                head->next= pre->next;

                pre->next= head;

                head= pre;

                forehand= head;

                pre= forehand->next;

            }

        while(pre->next!= NULL)

        {

            if(pre->start\_addr>= rear->start\_addr)

            {

                //比较链表中其它相邻两个结点的开始地址的大小

                pre->next= rear->next;

                forehand->next= rear;

                rear->next= pre;

                forehand= rear;

                rear= pre->next;

            }

            else

            {

                forehand= pre;

                pre= rear;

                rear= rear->next;

            }

        }

    }

    free\_block=head;

    return 0;

}

//按BF算法重新整理内存空闲块链表，按空闲块大小从小到大排序

int rearrange\_BF()

{

    struct free\_block\_type \*head= free\_block;

    struct free\_block\_type \*forehand,\*pre,\*rear;

    int i;

    if(head== NULL)

        return -1;

    for(i= 0;i< free\_block\_count-1;i++)

    {

        forehand= head;

        pre= forehand->next;

        rear= pre->next;

        if(forehand== head&&forehand->size>= pre->size)//排在考前的空闲块的size比其之后的更大，需要交换

            {

                //比较空闲链表中第一个空闲块与第二个空闲块的空间的大小，size从小到大排序

                head->next= pre->next;

                pre->next= head;

                head= pre;

                forehand= head;

                pre= forehand->next;

                rear= pre->next;

            }

        while(pre->next!= NULL)

        {

            if(pre->size>= rear->size)

            {

                //比较链表中其它相邻两个结点的空间的大小

                pre->next= rear->next;

                forehand->next= rear;

                rear->next= pre;

                forehand= rear;

                rear= pre->next;

            }

            else

            {

                forehand= pre;

                pre= rear;

                rear= rear->next;

            }

        }

    }

    free\_block=head;

    return 0;

}

//按WF算法重新整理内存空闲块链表，按空闲块大小从大到小排序

int rearrange\_WF()

{

    struct free\_block\_type \*head= free\_block;

    struct free\_block\_type \*forehand,\*pre,\*rear;

    int i;

    if(head== NULL)

        return -1;

    for(i= 0;i< free\_block\_count-1;i++)

    {

        forehand= head;

        pre= forehand->next;

        rear= pre->next;

        if(forehand== head&&forehand->size<=pre->size)

            {

                //比较空闲链表中第一个空闲块与第二个空闲块空间的大小,第一个空闲块比第二个空闲块小时交换

                head->next= pre->next;

                pre->next= head;

                head= pre;

                forehand= head;

                pre= forehand->next;

                rear= pre->next;

            }

        while(pre->next!= NULL)

        {

            if(pre->size<= rear->size)

            {

                //比较链表中其它相邻两个结点的空间的大小

                pre->next= rear->next;

                forehand->next= rear;

                rear->next= pre;

                forehand= rear;

                rear= pre->next;

            }

            else

            {

                forehand= pre;

                pre= rear;

                rear= rear->next;

            }

        }

    }

    free\_block=head;

    return 0;

}

//创建一个新进程

int new\_process()

{

    struct allocated\_block \*ab;

    int size;

    int ret;

    ab= (struct allocated\_block\*)malloc(sizeof(struct allocated\_block));

    if(!ab)

        exit(-5);

    ab->next= NULL;

    pid++;

    sprintf(ab->process\_name,"PROCESS-%02d",pid);//将格式化的数据写入某字符串中

    ab->pid= pid;

    printf("Memory for %s:",ab->process\_name);

    for(;;)

    {

        scanf("%d",&size);

        getchar();

        if(size> 0)

        {

            ab->size= size;

            break;

        }

        else

            printf("The size have to greater than zero!Please input again!");

    }

    ret= allocate\_mem(ab);

    if((ret== 1)&&(allocated\_block\_head== NULL))  //如果此时未赋值，则赋值

    {

        allocated\_block\_head= ab;

        return 1;

    }

    else if(ret== 1)  //分配成功，将该已分配块的描述插入已分配链表

    {

        struct allocated\_block \*p=allocated\_block\_head;

        while(p->next!=NULL)p=p->next;

        p->next=ab;

        return 2;

    }

    else if(ret== -1)  //分配不成功

    {

        printf("Allocation fail.\n");

        free(ab);

        return -1;

    }

    return 3;

}

//分配内存模块

int allocate\_mem(struct allocated\_block \*ab)

{

    int ret;

    struct free\_block\_type \*pre= NULL,\*f= free\_block;

    if(f== NULL)

        return -1;

    while(f!= NULL)

    {

        if(f->size>= ab->size)

        {

            ret= allocate(pre,f,ab);

            rearrange(ma\_algorithm);

            return ret;

        }

        pre= f;

        f= pre->next;

    }

    if(f== NULL&&current\_free\_mem\_size> ab->size)

        ret= mem\_retrench(ab);

    else

        {

            ret= -2;

            printf("no such spare rooms!!\n");

        }

    return ret;

}

//给新进程分配内存空间

int allocate(struct free\_block\_type \*pre,struct free\_block\_type \*allocate\_free\_block,struct allocated\_block \*ab)

{

    //struct allocated\_block \*p= allocated\_block\_head;

    ab->start\_addr= allocate\_free\_block->start\_addr;

    if(allocate\_free\_block->size-ab->size< MIN\_SLICE)

    {

        ab->size= allocate\_free\_block->size;

        if(pre!= NULL)

        {

            pre->next= allocate\_free\_block->next;

        }

        else

        {

            free\_block= allocate\_free\_block->next;

        }

        free(allocate\_free\_block);

        free\_block\_count--;

    }

    else

    {

        allocate\_free\_block->start\_addr+= ab->size;

        allocate\_free\_block->size-= ab->size;

    }

    /\*if(p== NULL)

    {

        allocated\_block\_head= ab;

    }

    else

    {

        while(p->next!= NULL)

            p= p->next;

        p->next= ab;

    }\*/

    current\_free\_mem\_size-= ab->size;

    if(current\_free\_mem\_size== 0)

        free\_block= NULL;

    return 1;

}

//通过内存紧缩技术给新进程分配内存空间

int mem\_retrench(struct allocated\_block \*ab)

{

    struct allocated\_block \*allocated\_work,\*allocated\_pre= allocated\_block\_head;

    struct free\_block\_type \*free\_rare,\*free\_work= free\_block->next;

    if(allocated\_pre== NULL)

        return -1;

    allocated\_pre->start\_addr= 0;

    allocated\_work= allocated\_pre->next;

    while(allocated\_work!= NULL)

    {

        allocated\_work->start\_addr= allocated\_pre->start\_addr+ allocated\_pre->size;

        allocated\_pre= allocated\_work;

        allocated\_work= allocated\_work->next;

    }

    free\_block->start\_addr= allocated\_pre->start\_addr+ allocated\_pre->size;

    free\_block->size= current\_free\_mem\_size;

    free\_block->next=NULL;

    free\_rare= free\_work->next;

    while(free\_rare!= NULL)

    {

        free(free\_work);

        free\_work= free\_rare;

        free\_rare=free\_rare->next;

        if(free\_rare== NULL)

            free(free\_work);

    }

    free\_block\_count=1;

    allocate(NULL,free\_block,ab);

    return 1;

}

//删除进程，归还分配的存储空间，并删除描述该进程内存分配的结点

void kill\_process()

{

    struct allocated\_block \*ab;

    int pid;

    printf("Kill Process,pid=");

    scanf("%d",&pid);

    getchar();

    ab= find\_process(pid);

    if(ab!= NULL)

    {

        free\_mem(ab);  //释放ab所表示的分配区(free区)

        dispose(ab);   //释放ab数据结构结点

    }

}

struct allocated\_block\* find\_process(int pid){

    struct allocated\_block \*p=allocated\_block\_head;

    while(p!=NULL){

        if(p->pid==pid)return p;

        p=p->next;

    }

    printf("no such a pid process!\n");

    return NULL;

}

//将ab所表示的已分配区归还，并进行可能的合并,对空闲块进行操作

int free\_mem(struct allocated\_block \*ab){

    current\_free\_mem\_size+= ab->size;

    struct free\_block\_type \*pre,\*work,\*rear;

    work=(struct free\_block\_type\*)malloc(sizeof(struct free\_block\_type));

    work->start\_addr=ab->start\_addr;

    work->size=ab->size;

    work->next=NULL;

    pre=free\_block;

    if(pre==NULL){free\_block=work;free\_block\_count++;return 1;}

    rear=pre->next;

    while(rear!=NULL){

        pre=rear;

        rear=rear->next;

    }

    pre->next=work;

    free\_block\_count++;

    rearrange\_FF();

    pre=free\_block;

    rear=free\_block->next;

    while(rear!=NULL)

    {

        while(pre->start\_addr+pre->size==rear->start\_addr)

        {

            pre->size+=rear->size;

            pre->next=rear->next;

            free(rear);

            free\_block\_count--;

            rear=pre->next;

            if(rear==NULL)break;

        }

        if(rear!=NULL){

        pre=rear;

        rear=rear->next;

        }

    }

    rearrange(ma\_algorithm);

    return 1;

}

int dispose(struct allocated\_block \*free\_ab){

    struct allocated\_block \*pre,\*work;

    pre=allocated\_block\_head;

    if(pre==NULL)return -1;

    //如果是第一个节点

    if(pre->pid==free\_ab->pid){allocated\_block\_head=pre->next;free(pre);return 1;}

    work=pre->next;

    while(work!=NULL){

        if(work->pid==free\_ab->pid){

            pre->next=work->next;

            free(work);

            return 1;

        }

        else{

            pre=work;

            work=work->next;

        }

    }

}

//显示当前内存的使用情况，包括空闲区的情况和已经分配的情况

int display\_mem\_usage()

{

    struct free\_block\_type \*fbt= free\_block;

    struct allocated\_block \*ab= allocated\_block\_head;

    printf("-------------------------------------------------------------\n");

    printf("Free Memory:\n");

    printf("%20s %20s\n","     start\_addr","     size");

    while(fbt!= NULL)

    {

        printf("%20d %20d\n",fbt->start\_addr,fbt->size);

        fbt= fbt->next;

    }

    printf("\nUsed Memory:\n");

    printf("%10s %20s %10s %10s\n","PID","ProcessName","start\_addr","size");

    while(ab!= NULL)

    {

        printf("%10d %20s %10d %10d\n",ab->pid,ab->process\_name,ab->start\_addr,ab->size);

        ab= ab->next;

    }

    printf("-------------------------------------------------------------\n");

    return 1;

}

int do\_exit()

{

    struct allocated\_block \*allocated\_ab,\*allocated\_pre;

    struct free\_block\_type \*free\_ab,\*free\_pre;

    free\_pre= free\_block;

    allocated\_pre= allocated\_block\_head;

    if(free\_pre!= NULL)

    {

        free\_ab= free\_pre->next;

        while(free\_ab!= NULL)

        {

            free(free\_pre);

            free\_pre= free\_ab;

            free\_ab= free\_ab->next;

        }

        free(free\_pre);

    }

    if(allocated\_pre!= NULL)

    {

        allocated\_ab= allocated\_pre->next;

        while(allocated\_ab!= NULL)

        {

            free(allocated\_pre);

            allocated\_pre= allocated\_ab;

            allocated\_ab= allocated\_ab->next;

        }

        free(allocated\_pre);

    }

    return 0;

}

//显示主菜单

void display\_menu()

{

    printf("\n");

    //system("clear");

    printf("1-Set memory size(default= %d)\n",DEFAULT\_MEM\_SIZE);

    printf("2-Select memory allocation algorithm\n");

    printf("3-New process\n");

    printf("4-Terminate a process\n");

    printf("5-Display memory usage\n");

    printf("0-Exit\n");

}

```

**## 实验结果**

**### 设计分配算法为总选择空闲块队列里第一个符合要求的空闲块，通过对空闲队列进行不同的排序从而实现不同的选择策略**

**#### 空闲块按起始地址排序**

![first\_fit](./first\_fit.png)

**#### 空闲块按空闲块大小 从小到大排序**

![best\_fit](./best\_fit.png)

**#### 空闲块按空闲块大小 从大到小排序**

![worst\_fit](./worst\_fit.png)

**#### 单个空闲块不足但总剩余空闲内存足够时，进行内存紧缩分配内存**

![jinsuo](./jinsuo.png)

**#### 进程结束时归还内存，对连续的空闲块合并**

![hebing](./hebing.png)

**3实验 3 Linux 动态模块与设备驱动**

**3.1Linux 的动态模块**

**3.1.1实验目的**

a)实现动态模块的插入

b)实现对Makefile文件实现的了解。

**3.1.2实验内容**

使用动态模块，篡改内核中的模块，实现系统内核的篡改，实现新的功能。

**3.1.3 实验思想**

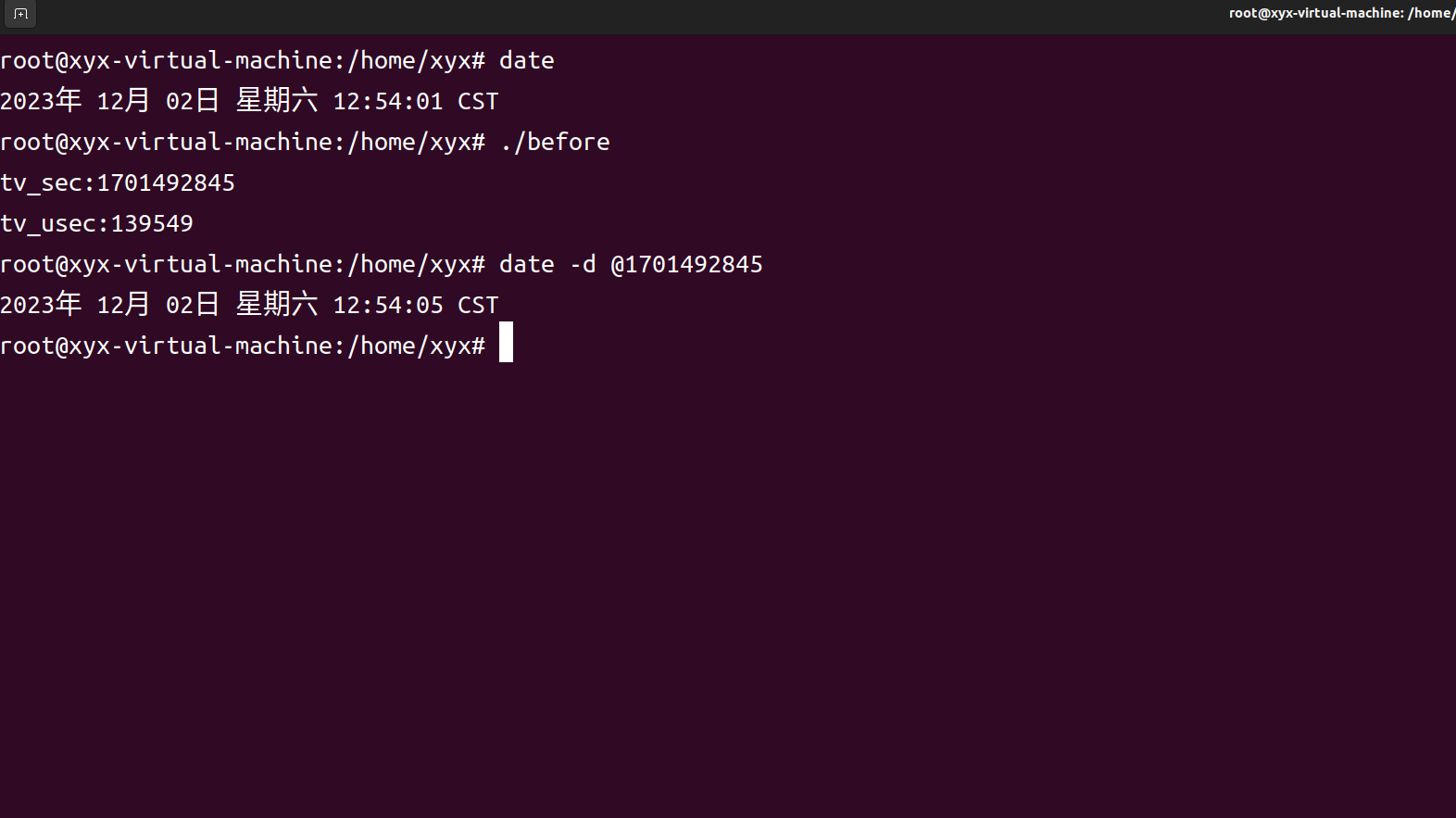
篡改系统模块，并使用测试程序检验是否达到成功篡改模块的功能。

**3.1.4 程序运行初值及运行结果分析**

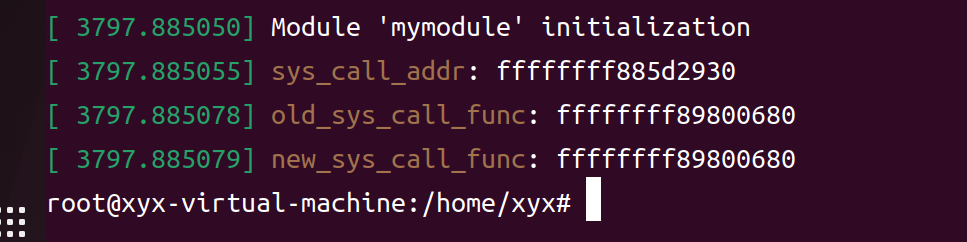
本实验无运行初值的设置。

运行结果分析：

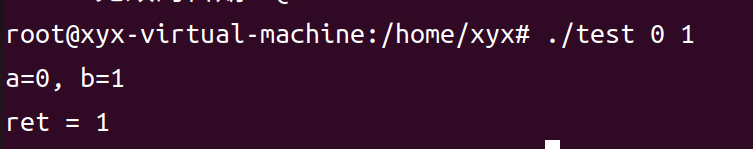
在插入动态模块前，运行测试程序，转换时间如下：



插入动态模块之后，使用sudo dmesg查看系统日志



成功篡改内核后，执行调用了syscall接口的test文件，观察是否实现对应的功能。



通过篡改之后的内核实现了加功能。

需要注意的地方：

在模块的init中调用modify\_syscall();

在模块的exit中一定要调用restore\_syscall()，还原操作系统原本的内核，避免影响操作系统的使用。

**3.2Linux的设备驱动**

**3.2.1 实验目的**

a) 理解 LINUX 字符设备驱动程序的基本原理；

b) 掌握字符设备的驱动运作机制；

c) 学会编写字符设备驱动程序。

**3.2.2 实验内容**

编写一个简单的字符设备驱动程序，以内核空间模拟字符设备，完成对该设备的打开，读写和释放操作，并编写聊天程序实现对该设备的同步和互斥操作。

**3.2.3 实验思想**

通过编写设备文件的代码设计设备，并且动态加载到内核中，通过设备的读读写端实现聊天进程的消息发送。

**3.2.4 程序运行初值及运行结果分析**

初值设置：

globalvar.private\_chat\_pid=0;记录私聊进程号，为0时表示一个写者发送信息给多个读者，处于群发状态，初始化为0，标识最初设备处在群发状态。

    INIT\_LIST\_HEAD(&globalvar.readers);初始化list\_head类型变量globalvar.readers，用来记录目前设备中读进程的进程号，以便检测发送的消息所指定的进程号是否合法。

    globalvar.read\_count=0;在一个写者发送信息给多个读者（群发）时，用来统计已经收信息的读者数量。 初始化为0。

    globalvar.total\_readers=0;记录总的读者的数量，是实现群发功能的基础。每当设备检测到有进程以读权限访问设备时，globalvar.total\_readers++，每当有进程结束（close）时，如果是访问的有读权限的端口，globalvar.total\_readers--，初始化为0.

    cdev\_init(&globalvar.devm, &globalvar\_fops);初始化内置字符设备devm。

    sema\_init(&globalvar.sem, 1);初始化互斥锁为1，标示互斥锁此时可用。

    init\_waitqueue\_head(&globalvar.outq);初始化读进程等待队列outq。

    globalvar.rd = globalvar.buffer;

    globalvar.wr = globalvar.buffer;

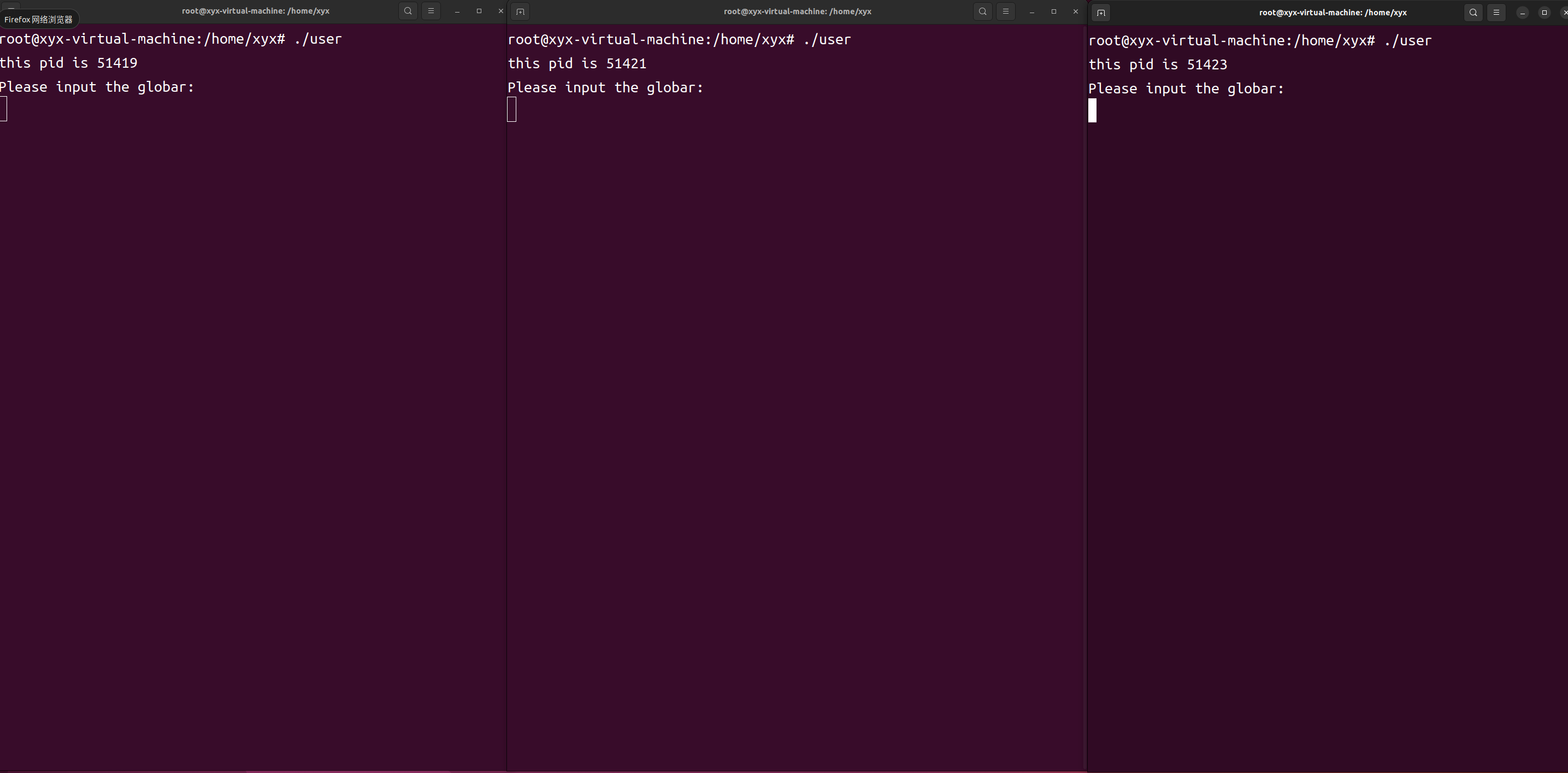
globalvar.end = globalvar.buffer + MAXNUM;

以上三行初始化设备内核空间中的读写指针和尾指针。初始时读、写指针同时指向字符缓冲区头部，尾指针（end）指向缓冲区尾部。

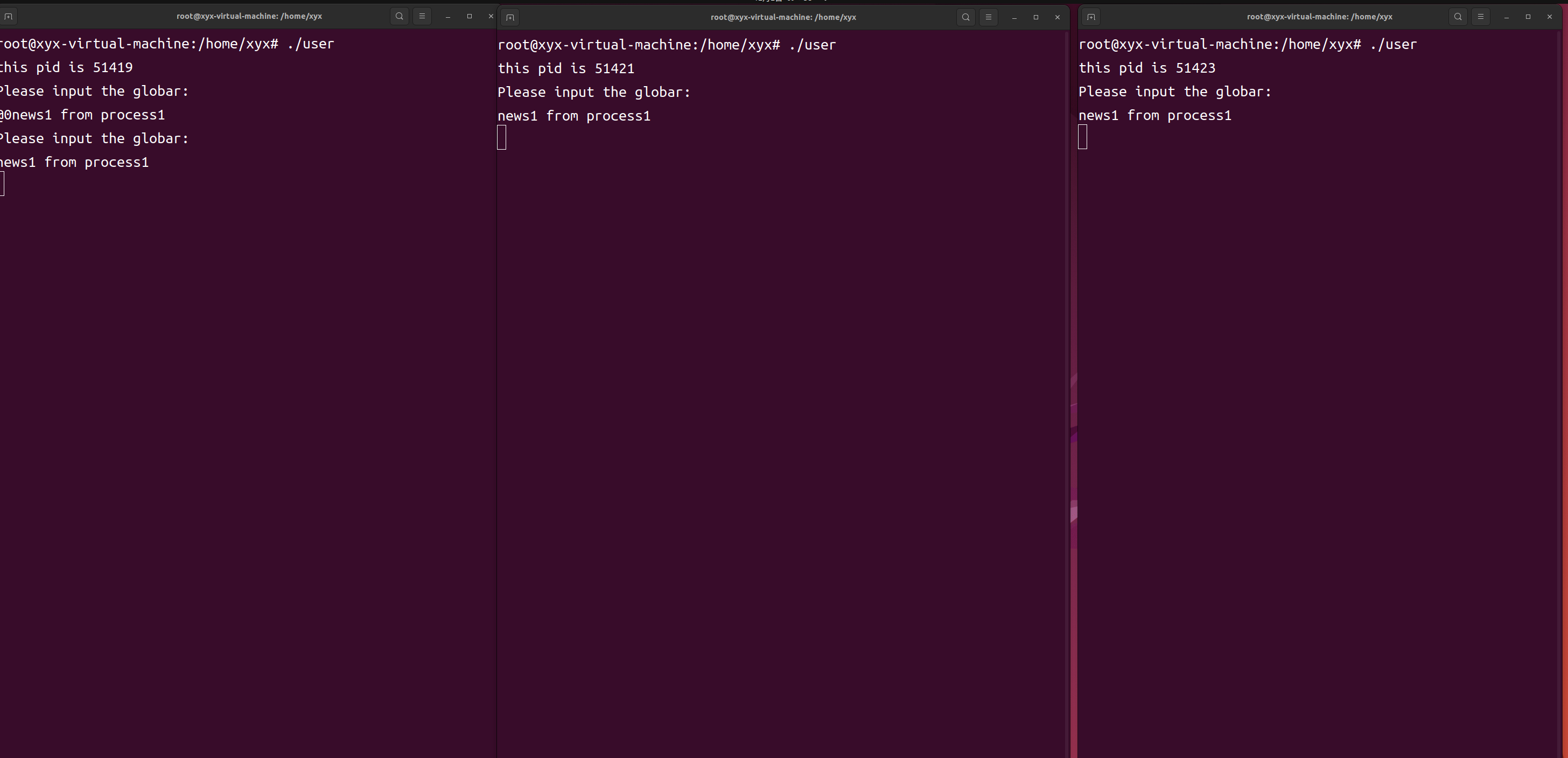
globalvar.flag = 0;用来控制读进程是否可读的信号量，设备初始化时由于还未写入任何信息，初始化为0.

运行结果及其分析：

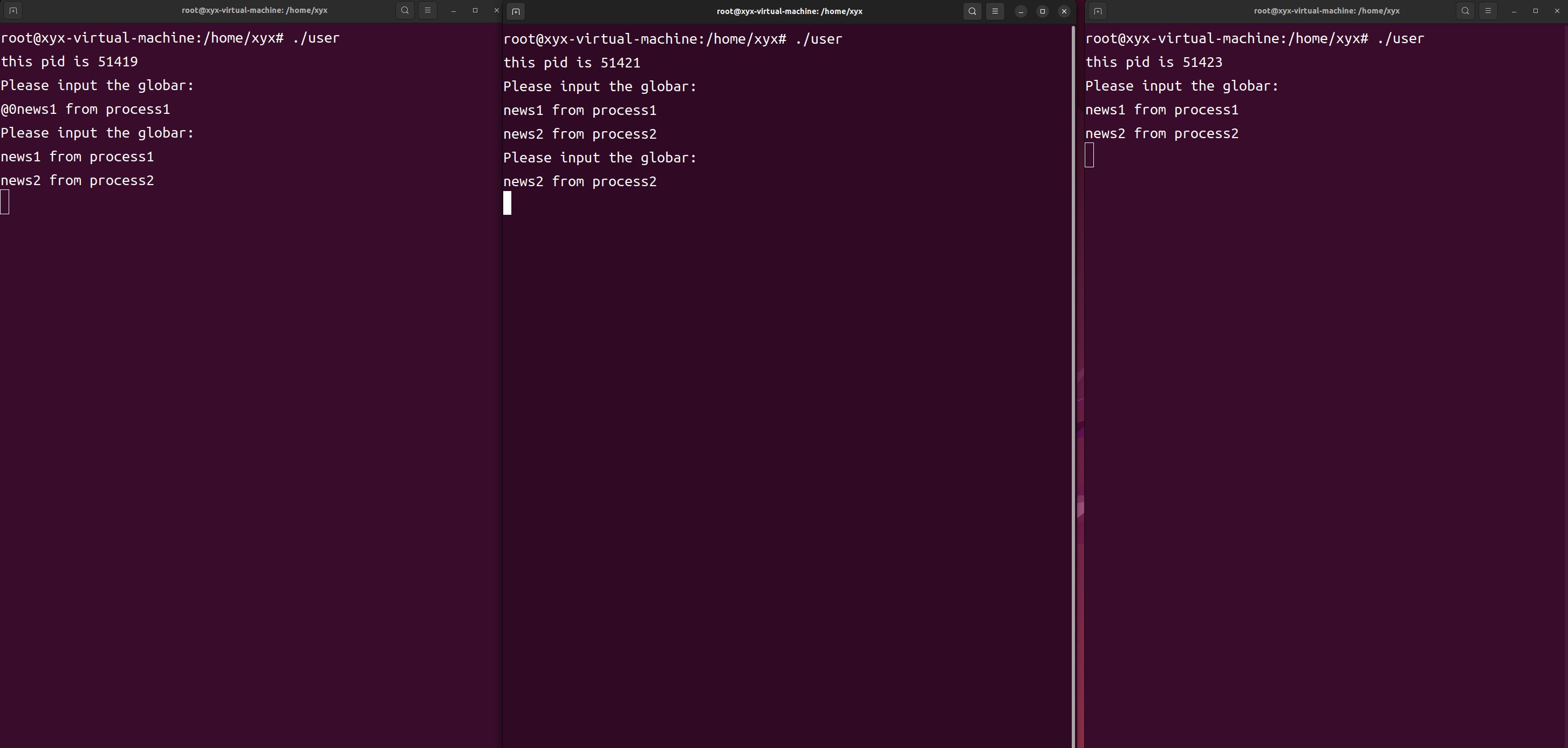
打开三个测试终端，为了方便通过进程的进程号进行私聊通信，测试程序在创建进程时会打印进程号。



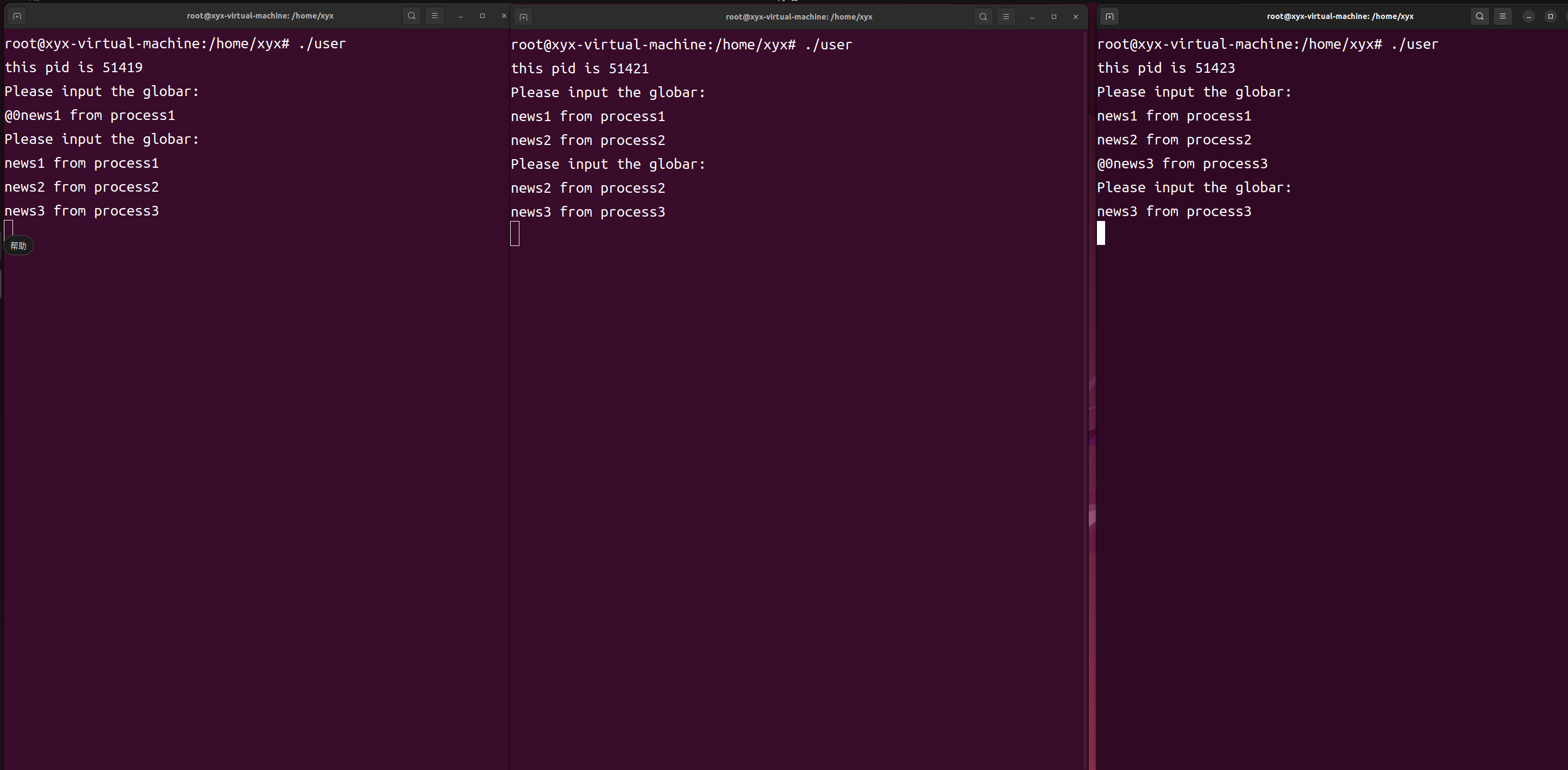
终端1群发消息给所有终端进程（包括自己）：



终端2进程群发消息给所有终端进程（包括自己）：

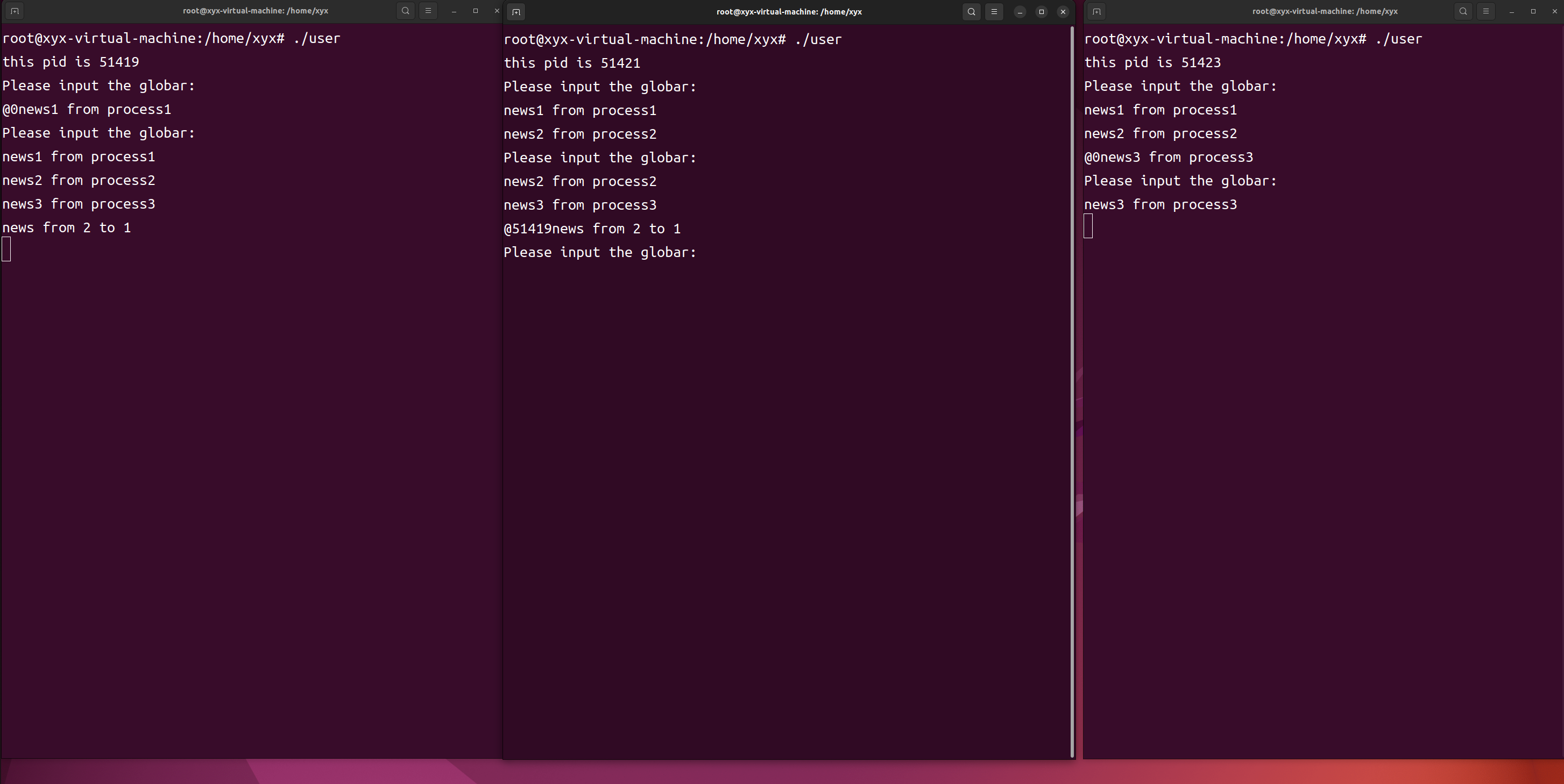


终端3进程群发消息给所有终端进程（包括自己）：

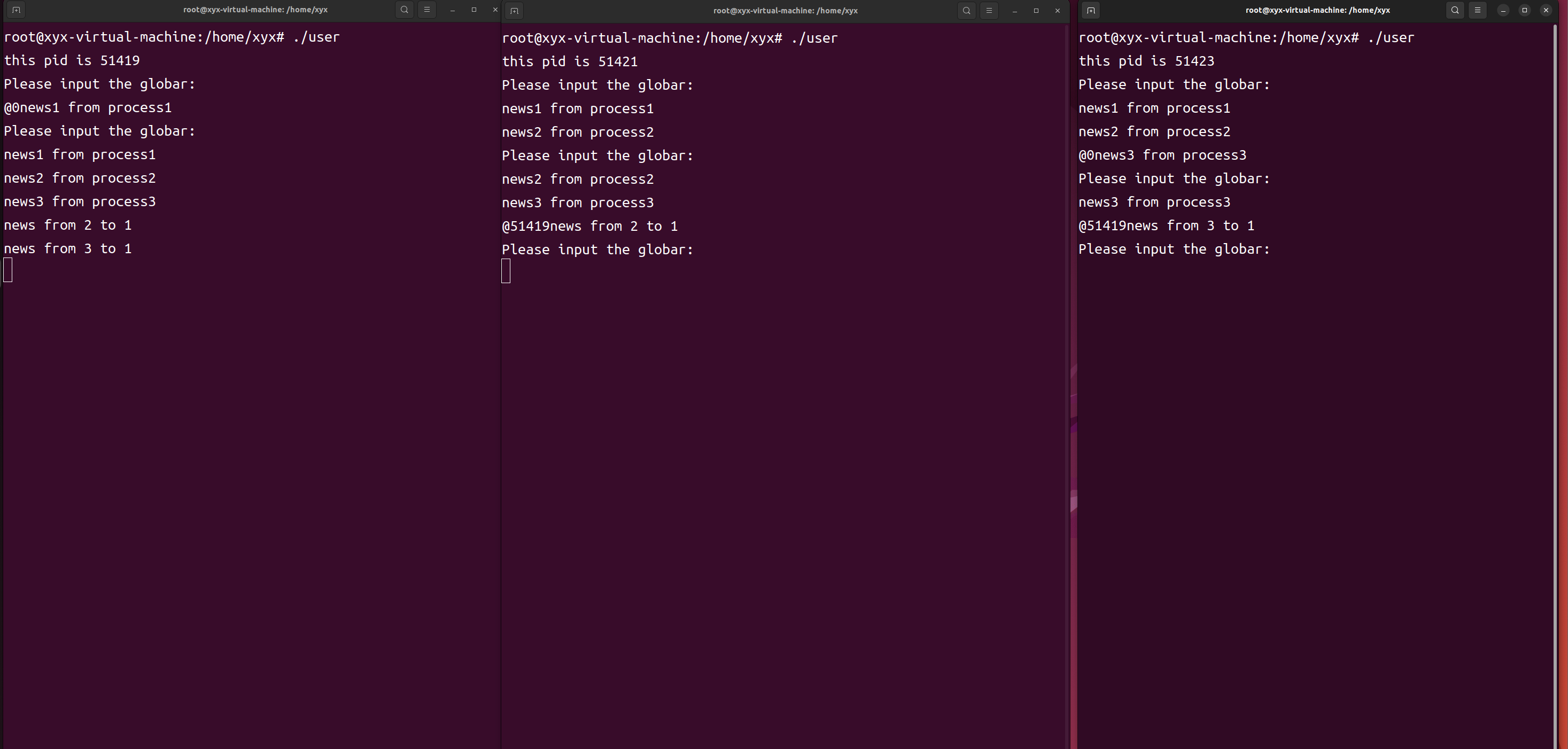


私聊功能：

终端2向终端1发送私聊消息：

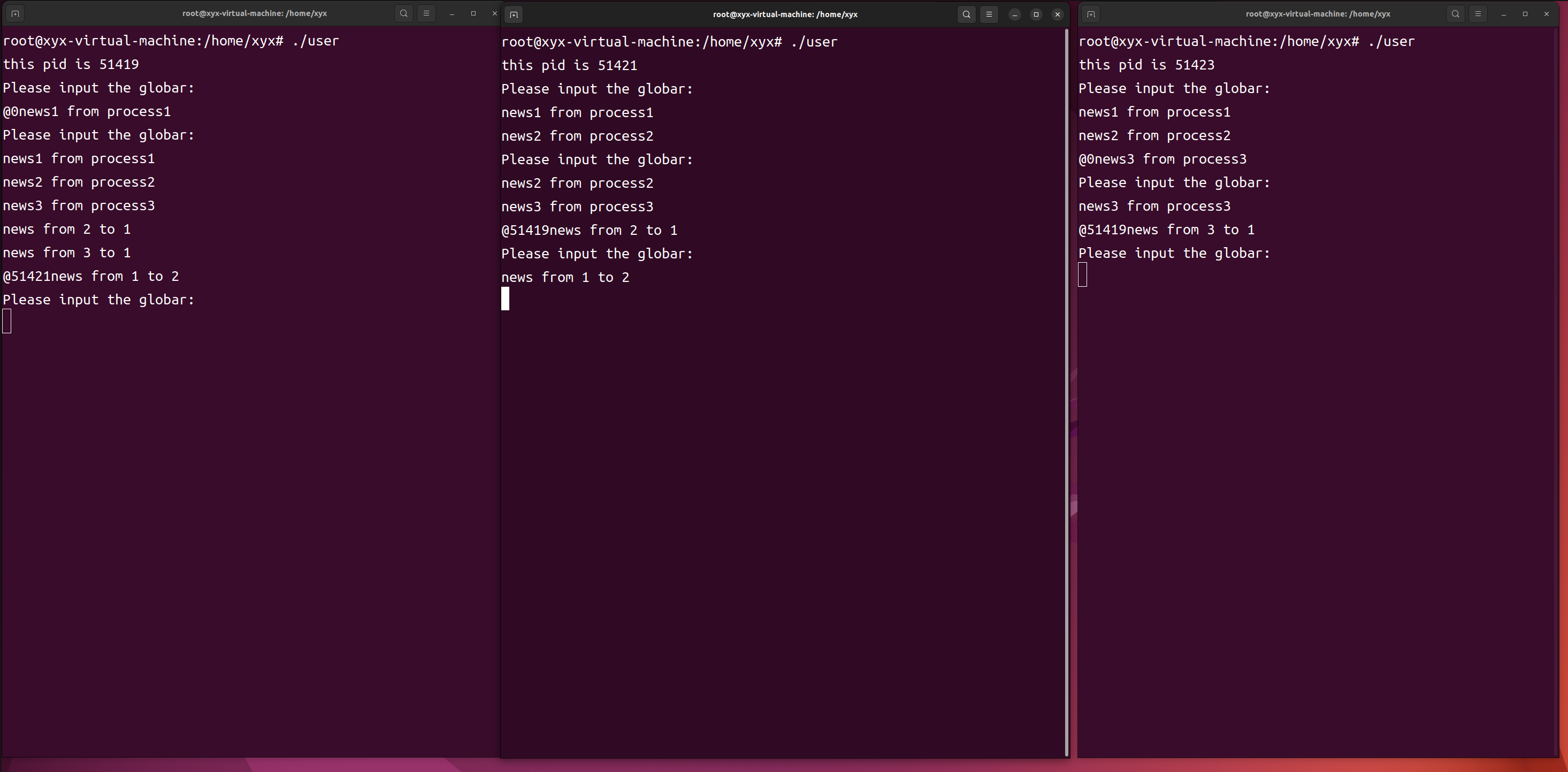


终端3向终端1发送私聊消息：

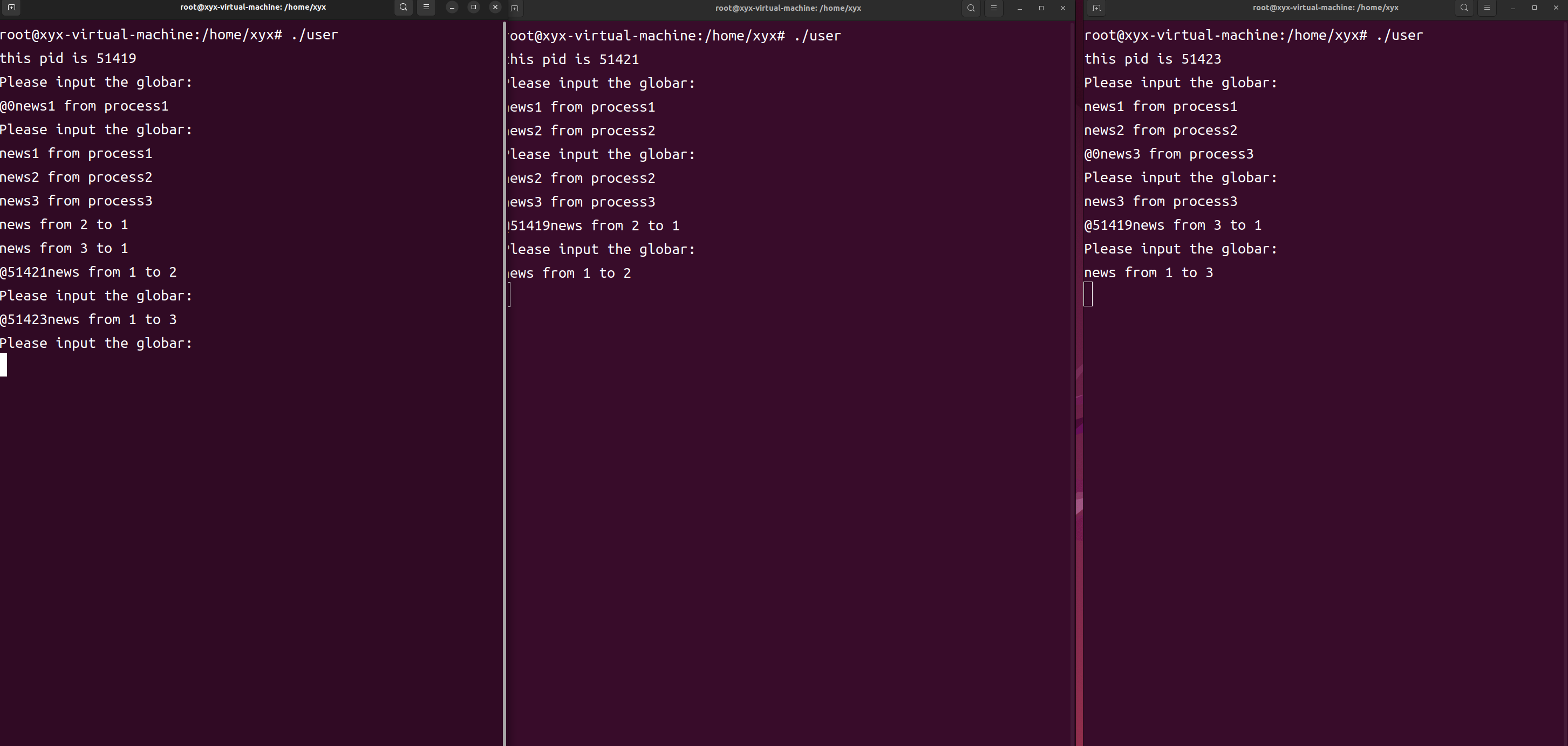


终端1向终端2和终端3分别发送消息：

向终端2：

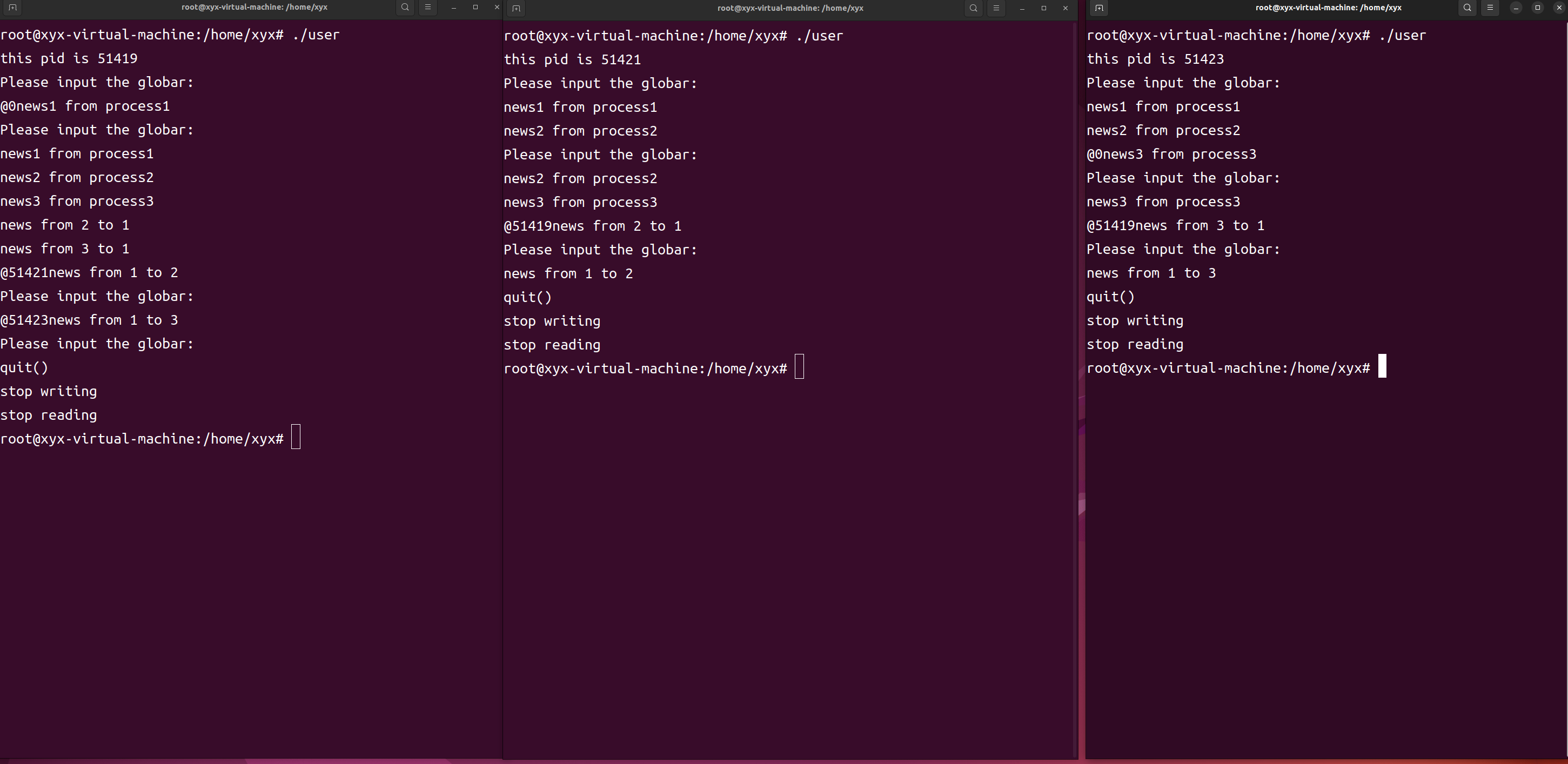


向终端3：



结束退出：

三个终端输入quit()退出：



结果分析：实现了一对多和私聊消息的功能，并且测试文件编写的每个终端拥有控制自己读写的权力。

**3.2.5主要功能的实现方式**

#### 实现群发（一对多）功能的方式

通过再globalvar实例中添加全局变量total\_readers，实时维护读者的数量，提供了每个进程都能读到信息的基础。群发模式下，所有在读进程等待队列中阻塞的读进程都会被唤醒，依次按顺序实现对缓冲区内容的读出，每一个读进程读一次缓冲区后globalvar.reader\_count计数加一，最后一个读进程读之前不修改读指针，最后一个读进程检测到计数值与读进程数相同时，才把flag改为0（标识为不可读），修改读指针（保证下一次读取正确），把globalvar.reader\_count置零（保证下一次计数正确）。

面临的问题：编程过程中主要遇到的问题：如何避免一个读进程多次先抢占到PV锁读缓冲区，导致有的读进程读多次缓冲区，而有的进程没有读过一次缓冲区？

解决方法：

    在设备都端口添加以下代码块：

while(globalvar.flag == 1&&globalvar.read\_count!=0)

    {

        if(down\_interruptible(&globalvar.sem)) //P 操作

    {

        return -ERESTARTSYS;

    }

        up(&globalvar.sem); //V 操作

    }

在群发模式下，如果有进程先完成了一次读缓冲区，在进行第二次时，会由于globalvar.flag == 1&&globalvar.read\_count!=0判别生效，陷入在while陷阱中。此外，在内核中直接使用while忙等待会导致系统卡死，此处在while循环中使用PV操作避免卡死，尚未读过缓冲区的读进程总会申请到PV锁，这样在有新消息到达时候，所有读进程一定能且只能收到一条消息。待到最后一个读进程从缓冲区读完消息后globalvar.flag == 1&&globalvar.read\_count!=0判别失败，所有读进程跳出陷阱，恢复正常。

如何保证在有多个写进程时，一个新的写入不会打断群发消息模式多个进程依次读缓冲区的过程？

必要性：想象这样一个场景，在刚刚描述的一对多场景下，有一部分读进程已经读取了缓冲区，而有一部分进程还没有读取缓冲区，如果此时有一个写进程抢占了PV锁并发送了私密消息，会修改写指针的位置，同时修改接受进程号标示，导致还未读缓冲区的读进程收不到消息，globalvar.reader\_count、flag等一系列变量出错，接收端也接受不到正确的私密信息，所以，避免写进程打断，一部分读进程已经读取了缓冲区，而有一部分进程还没有读取缓冲区的情况，是有必要的。

解决方法：类似于之前的处理方法，写进程开始申请PV锁进行对内核空间的缓冲区进行写操作之前，添加以下代码块：  
 while(globalvar.flag==1&&globalvar.read\_count!=0)

    {

            if(down\_interruptible(&globalvar.sem)) //P 操作

        {

            return -ERESTARTSYS;

        }

            up(&globalvar.sem); //V 操作

    }

即，如果globalvar.flag==1&&globalvar.read\_count!=0，标示此时处在群发状态且有读进程还未读到缓冲区，写进程掉入陷阱，直到全部读进程正常接收到消息之后再正常执行写操作。

#### 实现私发功能的方式

在私发模式下，只有是相应进程号的进程才会被唤醒。

此外，为了防止使用发送端口发送给非法进程（没有对应的进程在等待读），如果没有目标进程，我们并不希望私发的消息被写入缓冲区中被其他读端口读取，所以需要引入检测信号是否合法的机制

使用

struct reader\_node {

pid\_t pid; // 进程的PID

struct list\_head list; // 列表节点

};

结构，动态维护在等待的读端口，如果私发消息@的进程号匹配不到检任何一个读端口，那么这条消息将被丢弃。

此外，通过@0方式可以将设备恢复到群发模式。

#### 测试文件实现既能发送又能接受消息的聊天端

在测试文件中，为了实现同时既能发送消息又能接收消息的功能，使用了多进程机制，父进程负责写功能，子进程负责读功能。为了进一步实现父进程对子进程的控制，以避免用户退出父进程后子进程陷入僵尸进程，当用户输入quit()时，父进程向子进程发送结束信号，等待子进程结束后再结束自身，避免了僵尸进程。

**3.3实验总结**

**实验中遇到的问题：**

初始时，在内核空间中访问用户空间的数据，造成权限错误，且由于在云服务器环境下进行实验，每次越界会造成云服务器连接卡死并断开服务，重新连接后无法查看日志调试。最后更换为本地机环境后才找到问题所在。

在解决群发功能时，即使保证能保证缓冲区只会在被读到指定次数后才会更新指针，但是无法保证每个进程只读一次。初始时直接设计while忙等待陷阱希望拦住已经读过一次缓冲区的读进程，但是单纯的while循环会使得设备卡死，猜测是由于在内核环境中运行速度更快，无效的while循环产生了异常，最后修改为在while陷阱状态下不断进行PV操作，解决了这一问题。

实验收获：

通过本次实验，让我了解到了Linux动态内核的作用和用法，通过设备驱动实验更是让我深刻认识到内核空间这一概念，之前只通过教科书对内核空间这一概念没有清晰的了解，在内核中进行编程实验后，对操作系统的内核空间以及其在日常使用计算机过程中起的作用有了更深入的认识。

**3.4附件代码及README：**

附件3：代码即READEME3

**# Linux的动态模块与设备驱动**

**## 动态模块**

**### Makefile文件**

```

obj-m += mymodule.o

all:

    make -C /lib/modules/$(shell uname -r)/build M=$(PWD) modules

clean:

    make -C /lib/modules/$(shell uname -r)/build M=$(PWD) clean

```

**### 设备模块mymodule.c**

```

#include <linux/init.h>

#include <linux/kernel.h>

#include <linux/module.h>

#include <linux/sched/task\_stack.h>

#include <linux/kallsyms.h>

#include <linux/module.h>

#define \_\_force\_order (\*(volatile char\*)0)

#define sys\_No 96

unsigned long long old\_sys\_call\_func;

unsigned long long p\_sys\_call\_table = 0xffffffffb7c00260;

inline void mywrite\_cr0(unsigned long cr0) {

  asm volatile("mov %0,%%cr0" : "+r"(cr0), "+m"(\_\_force\_order));

}

void enable\_write\_protection(void) {

  unsigned long cr0 = read\_cr0();

  set\_bit(16, &cr0);

  mywrite\_cr0(cr0);

}

void disable\_write\_protection(void) {

  unsigned long cr0 = read\_cr0();

  clear\_bit(16, &cr0);

  mywrite\_cr0(cr0);

}

asmlinkage long long hello(void)

{

    long long a, b;

    //register int a asm("rdi");

    //register int b asm("rsi");

    struct pt\_regs \*regs = task\_pt\_regs(current);

    a = regs->di;

    b = regs->si;

    printk("No 96 syscall has changed to hello, a = %lld, b = %lld, \n", a, b);

    return a + b;

}

void modify\_syscall(void)

{

    unsigned long long \*sys\_call\_addr;

    sys\_call\_addr = (unsigned long long \*)(p\_sys\_call\_table + sys\_No \* sizeof(void \*));

    old\_sys\_call\_func = \*(sys\_call\_addr);

    printk(KERN\_INFO "old\_sys\_call\_func: %llx\n", (unsigned long long)sys\_call\_addr);

    \*(sys\_call\_addr) = (unsigned long long)&hello; // 指向新函数

    printk(KERN\_INFO "new\_sys\_call\_func: %llx\n", (unsigned long long)sys\_call\_addr);

}

void restore\_syscall(void)

{

    unsigned long long \*sys\_call\_addr;

    sys\_call\_addr = (unsigned long long \*)(p\_sys\_call\_table + sys\_No \* sizeof(void \*));

    \*(sys\_call\_addr) = old\_sys\_call\_func; // 指向原始函数

}

static int mymodule\_init(void)

{

    unsigned long long \*sys\_call\_addr;

    printk(KERN\_INFO "Module 'mymodule' initialization\n");

    sys\_call\_addr = (unsigned long long \*)(p\_sys\_call\_table + sys\_No \* sizeof(void \*));

    printk(KERN\_INFO "sys\_call\_addr: %llx\n", \*sys\_call\_addr);

    disable\_write\_protection();

    modify\_syscall();

    enable\_write\_protection();

    return 0;

}

static void mymodule\_exit(void)

{

    printk(KERN\_INFO "Module 'mymodule' exit\n");

    disable\_write\_protection();

    restore\_syscall();

    enable\_write\_protection();

}

module\_init(mymodule\_init);

module\_exit(mymodule\_exit);

MODULE\_LICENSE("GPL");

MODULE\_AUTHOR("Kalo");

```

**### 篡改前before.c**

```

#include <stdio.h>

#include <sys/time.h>

#include <unistd.h>

int main()

{

    struct timeval tv;

    syscall(96, &tv, NULL); // before modify syscall 96 :gettimeofday

    printf("tv\_sec:%ld\n", tv.tv\_sec);

    printf("tv\_usec:%ld\n", tv.tv\_usec);

    return 0;

}

```

**### 篡改后test.c**

```

#include<stdio.h>

#include<sys/time.h>

#include<unistd.h>

#include<stdlib.h>

int main(int argc, char \*argv[]){

    long long arg1 = atoi(argv[1]);

    long long arg2 = atoi(argv[2]);

    printf("a=%lld, b=%lld\n", arg1, arg2);

    long long ret = syscall(96, arg1, arg2);

    if(ret != (arg1 + arg2)) {

        printf("Failed: %lld\n", ret);

        perror("syscall");

    } else {

        printf("ret = %lld\n", ret);

    }

    return 0;

}

```

**## 字符设备驱动**

**### Makefile文件**

```

CONFIG\_MODULE\_SIG=n

ifneq ($(KERNELRELEASE),)

        obj-m :=dev.o

else

        KERNELDIR := /lib/modules/$(shell uname -r)/build

PWD := $(shell pwd)

modules:

    $(MAKE) -C $(KERNELDIR) M=$(PWD) modules

clean:

    $(MAKE) -C $(KERNELDIR) M=$(PWD) clean

endif

```

**### Makefile文件代码解读**

```

obj-m :=dev.o  指定了要构建的目标模块为 dev.o

KERNELDIR := /lib/modules/$(shell uname -r)/build  设置了内核头文件的目录路径，它使用 uname -r 命令获取当前运行内核的版本

modules:: 这是一个构建目标，用于编译内核模块。下面的命令通过调用 make 在指定的内核头文件目录中进行模块的编译。M=$(PWD) 指定了模块源代码所在的当前工作目录

clean:: 这是一个构建目标，用于清理编译过程中产生的临时文件和输出文件。类似于 modules，它通过调用 make 在指定的内核头文件目录中进行清理操作。

```

**### 设备文件dev.c**

```c

#include <linux/module.h>

#include <linux/init.h>

#include <linux/fs.h>

#include <linux/uaccess.h>

#include <linux/semaphore.h>

#include <linux/sched.h>

#include <linux/wait.h>

#include <linux/cdev.h>

#include <linux/kdev\_t.h>

#include <linux/device.h>

#include <linux/types.h>

#include <linux/slab.h>

#include <linux/string.h>

#define MAXNUM 100

#define MAJOR\_NUM 290

//函数声明部分

static ssize\_t globalvar\_read(struct file \*filp, char \_\_user \*buf, size\_t len, loff\_t \*ppos);

static ssize\_t globalvar\_write(struct file \*filp, const char \_\_user \*buf, size\_t len, loff\_t \*ppos);

static int globalvar\_open(struct inode \*inode,struct file \*filp);

static int globalvar\_release(struct inode \*inode,struct file \*filp);

int globalvar\_init(void);

static void globalvar\_exit(void);

struct dev{

    struct cdev devm;

    struct semaphore sem;

    wait\_queue\_head\_t outq;

    int flag;

    char buffer[MAXNUM + 1];

    char \*rd, \*wr, \*end;

    pid\_t private\_chat\_pid;

    int read\_count; // 新增字段，用于跟踪已经完成读操作的进程数量

    int total\_readers; // 新增字段，用于存储读进程的总数

    struct list\_head readers; // 读进程的列表

    int length;//最后一个读进程退出时用来更新读指针

}globalvar;

struct reader\_node {

    pid\_t pid; // 进程的PID

    struct list\_head list; // 列表节点

};

static struct class \*my\_class;

static inline int isdigit(int ch)

{

    return ch >= '0' && ch <= '9';

}

int major = MAJOR\_NUM;

static ssize\_t globalvar\_read(struct file \*filp, char \_\_user \*buf, size\_t len, loff\_t \*ppos)

{

    while(globalvar.flag == 1&&globalvar.read\_count!=0)

    {

        if(down\_interruptible(&globalvar.sem)) //P 操作

    {

        return -ERESTARTSYS;

    }

        up(&globalvar.sem); //V 操作

    }

    if(wait\_event\_interruptible(globalvar.outq, globalvar.flag!=0&&(globalvar.private\_chat\_pid==0||(globalvar.private\_chat\_pid!=0&&current->pid == globalvar.private\_chat\_pid)))) //不可读时 阻塞读进程

    {

        return -ERESTARTSYS;

    }

    if(down\_interruptible(&globalvar.sem)) //P 操作

    {

        return -ERESTARTSYS;

    }

    if(globalvar.rd < globalvar.wr)

        len = min(len,(size\_t)(globalvar.wr - globalvar.rd)); //更新读写长度

    else

        len = min(len,(size\_t)(globalvar.end - globalvar.rd));

    globalvar.length=len;

    printk("in read process,len is %d\n",len);

    if(copy\_to\_user(buf,globalvar.rd,len))

    {

        printk(KERN\_ALERT"copy failed\n");

        up(&globalvar.sem);

        return -EFAULT;

    }

if(globalvar.private\_chat\_pid==0)

    {

        globalvar.read\_count++; // 增加计数器

        printk("count++\nnow reader count is %d\n",globalvar.read\_count);

        // 当计数器的值等于读进程的总数时，更新globalvar.rd的位置

        if(globalvar.read\_count == globalvar.total\_readers)

        {

            printk("in read process,before newing wr=%d,rd=%d\n",globalvar.wr-globalvar.buffer,globalvar.rd-globalvar.buffer);

            globalvar.rd = globalvar.rd + len;

            if(globalvar.rd == globalvar.end)

                globalvar.rd = globalvar.buffer; //字符缓冲区循环

            printk("in read process,after newing wr=%d,rd=%d\n",globalvar.wr-globalvar.buffer,globalvar.rd-globalvar.buffer);

            globalvar.read\_count = 0; // 重置计数器

            globalvar.flag=0;

        }

    }

else//指定私聊进程的操作

    {

        printk("in @ read process,before newing wr=%d,rd=%d\n",globalvar.wr-globalvar.buffer,globalvar.rd-globalvar.buffer);

        globalvar.rd = globalvar.rd + len;

        globalvar.flag=0;

            if(globalvar.rd == globalvar.end)

                globalvar.rd = globalvar.buffer; //字符缓冲区循环

        printk("in @ read process,after newing wr=%d,rd=%d\n",globalvar.wr-globalvar.buffer,globalvar.rd-globalvar.buffer);

    }

    up(&globalvar.sem); //V 操作

    return len;

}

static ssize\_t globalvar\_write(struct file \*filp, const char \_\_user \*buf, size\_t len, loff\_t \*ppos)

{

    while(globalvar.flag==1&&globalvar.read\_count!=0)

    {

            if(down\_interruptible(&globalvar.sem)) //P 操作

        {

            return -ERESTARTSYS;

        }

            up(&globalvar.sem); //V 操作

    }

    char \*kbuf = kmalloc(len + 1, GFP\_KERNEL); // 分配内核空间的内存

    if (!kbuf)

    {

        printk("kmalloc error\n");

        up(&globalvar.sem); //V 操作

        return -ENOMEM;

    }

    if (copy\_from\_user(kbuf, buf, len)) { // 将用户空间的数据复制到内核空间

            printk("copy\_from\_user error\n");

            kfree(kbuf);

            up(&globalvar.sem); //V 操作

            return -EFAULT;

        }

    kbuf[len] = '\0'; // 确保字符串以null字符结束

    int i=-1;

    if(down\_interruptible(&globalvar.sem)) //P 操作

    {

        return -ERESTARTSYS;

    }

    char first\_char;

   if (get\_user(first\_char, buf))

     {

        printk("get\_user error\n");

        up(&globalvar.sem); //V 操作

        return -EFAULT;

    }

    if(first\_char == '@') //检查是否为私聊消息

    {

        pid\_t pid;

        printk("kbuf is %s\n",kbuf);

        char pid\_str[10];

        long tmp;

        for (i = 0; i < len-1 && i < (sizeof(pid\_str) - 1) && isdigit(kbuf[i+1]); ++i)

        {

            pid\_str[i] = kbuf[i+1];

        }

        pid\_str[i] = '\0';

        // 转换为整数

        if (kstrtol(pid\_str, 10, &tmp)!= 0)

        {

            printk("kstrtoint error\n");

            kfree(kbuf);

            up(&globalvar.sem); //V 操作

            return -EFAULT;

        }

        pid=(pid\_t)tmp;

        // 检查指定的进程号是否在读进程的列表中

        struct list\_head \*pos;

        bool found = false;

        list\_for\_each(pos, &globalvar.readers)

        {

            struct reader\_node \*node = list\_entry(pos, struct reader\_node, list);

            if(node->pid == pid)

            {

                found = true;

                break;

            }

        }

            if(!found&&pid!=0)

            {

                // 指定的进程号不在读进程的列表中，忽略这次操作，但是如果为0，则改为群发

                kfree(kbuf);

                printk(KERN\_ALERT "Invalid pid: %d\n", pid);

                up(&globalvar.sem); //V 操作

                return -EINVAL;

            }

            // 指定的进程号在读进程的列表中，设置标志

            globalvar.private\_chat\_pid = pid;

    }

    if(globalvar.rd <= globalvar.wr)

        len = min(len-(i+1),(size\_t)(globalvar.end - globalvar.wr));

    else

        len = min(len-(i+1),(size\_t)(globalvar.rd - globalvar.wr-1));

    printk("in @write len is %d\n",len);

    /\*if(copy\_from\_user(globalvar.wr,buf,len))

    {

        up(&globalvar.sem); //V 操作

        return -EFAULT;

    }\*/

    strcpy(globalvar.wr, kbuf+i+1);

    printk("in write process,before newing,wr=%d,rd=%d\n",globalvar.wr-globalvar.buffer,globalvar.rd-globalvar.buffer);

    globalvar.wr = globalvar.wr + len;

    if(globalvar.wr == globalvar.end)

        globalvar.wr = globalvar.buffer; //循环

    printk("in write process,after newing,wr=%d,rd=%d\n",globalvar.wr-globalvar.buffer,globalvar.rd-globalvar.buffer);

    up(&globalvar.sem);//V 操作

    globalvar.flag = 1; //条件成立,可以唤醒读进程

    wake\_up\_interruptible(&globalvar.outq); //唤醒读进程

    printk("write send a number!");

    return len;

}

static int globalvar\_open(struct inode \*inode,struct file \*filp)

{

    try\_module\_get(THIS\_MODULE); //模块计数加一

    if (filp->f\_mode & FMODE\_READ) // 检查打开模式

    {

        globalvar.total\_readers++; //增加读进程的数量

        // 创建一个新的读进程节点

        struct reader\_node \*node = kmalloc(sizeof(struct reader\_node), GFP\_KERNEL);

        node->pid = current->pid;

        list\_add(&node->list, &globalvar.readers); // 添加到读进程的列表

        printk("total reader+1,now is%d\n",globalvar.total\_readers);

    }

    printk("This chrdev is in open\n");

    return(0);

}

static int globalvar\_release(struct inode \*inode,struct file \*filp)

{

    module\_put(THIS\_MODULE); //模块计数减一

    if (filp->f\_mode & FMODE\_READ) // 检查打开模式

    {

        globalvar.total\_readers--; //减少读进程的数量

        if(globalvar.read\_count!=0)globalvar.read\_count--;

        printk("total reader-1,now is%d\n",globalvar.total\_readers);

        struct list\_head \*pos, \*q;

        list\_for\_each\_safe(pos, q, &globalvar.readers)

        {

            struct reader\_node \*node = list\_entry(pos, struct reader\_node, list);

            if(node->pid == current->pid)

            {

                list\_del(pos);

                kfree(node);

                break;

            }

        }

        if(globalvar.total\_readers==0)//最后一个释放的读进程要负责完善善后工作

        {

            globalvar.read\_count = 0; // 重置计数器

            globalvar.flag=0;

        }

    }

    printk("now the total reader is %d\nthe count number%d\n",globalvar.total\_readers,globalvar.read\_count);

    printk("before release,wr=%d,rd=%d\n",globalvar.wr-globalvar.buffer,globalvar.rd-globalvar.buffer);

    printk("This chrdev is in release\n");

    return(0);

}

struct file\_operations globalvar\_fops =

{

    .read = globalvar\_read,

    .write = globalvar\_write,

    .open = globalvar\_open,

    .release = globalvar\_release,

};

int globalvar\_init(void)

{

    dev\_t dev = MKDEV(major, 0);

    int result;

    if(major)

    {

        //静态申请设备编号

        result = register\_chrdev\_region(dev, 1, "charmem");

    }

    else

    {

        //动态分配设备号

        result = alloc\_chrdev\_region(&dev, 0, 1, "charmem");

        major = MAJOR(dev);

    }

    if(result < 0)

        return result;

    globalvar.private\_chat\_pid=0;

    INIT\_LIST\_HEAD(&globalvar.readers);

    globalvar.read\_count=0;

    globalvar.total\_readers=0;

    cdev\_init(&globalvar.devm, &globalvar\_fops);

    globalvar.devm.owner = THIS\_MODULE;

    cdev\_add(&globalvar.devm, dev, 1);

    sema\_init(&globalvar.sem, 1);

    init\_waitqueue\_head(&globalvar.outq);

    globalvar.rd = globalvar.buffer;

    globalvar.wr = globalvar.buffer;

    globalvar.end = globalvar.buffer + MAXNUM;

    globalvar.flag = 0;

    my\_class = class\_create(THIS\_MODULE, "chardev0");

    device\_create(my\_class, NULL, dev, NULL, "chardev0");

    return 0;

}

static void globalvar\_exit(void)

{

    // 遍历读进程的列表，并释放每个节点的内存

    struct list\_head \*pos, \*q;

    list\_for\_each\_safe(pos, q, &globalvar.readers)

    {

        struct reader\_node \*node = list\_entry(pos, struct reader\_node, list);

        list\_del(pos);

        kfree(node);

    }

    device\_destroy(my\_class, MKDEV(major, 0));

    class\_destroy(my\_class);

    cdev\_del(&globalvar.devm);

    unregister\_chrdev\_region(MKDEV(major, 0), 1);

}

module\_init(globalvar\_init);

module\_exit(globalvar\_exit);

MODULE\_LICENSE("GPL");

```

**### 主要功能的实现方式**

**#### 实现群发（一对多）功能的方式**

```

通过再globalvar实例中添加全局变量total\_readers，实时维护读者的数量，提供了每个进程都能读到信息的基础。

```

**#### 实现私发功能的方式**

```

  在私发模式下，只有是相应进程号的进程才会被唤醒。

  此外，为了防止使用发送端口发送给非法进程（没有对应的进程在等待读），如果没有目标进程，我们并不希望私发的消息被写入缓冲区中被其他读端口读取，所以需要引入检测信号是否合法的机制

  使用

  struct reader\_node {

    pid\_t pid; // 进程的PID

    struct list\_head list; // 列表节点

};

 结构，动态维护在等待的读端口，如果私发消息@的进程号匹配不到检任何一个读端口，那么这条消息将被丢弃。

   此外，通过@0方式可以将设备恢复到群发模式。

```

**### 测试用户程序**

```c

#include<sys/types.h>

#include<unistd.h>

#include<sys/stat.h>

#include<stdio.h>

#include<fcntl.h>

#include<string.h>

#include <unistd.h>

#include <signal.h>

#include <stdlib.h>

#include <sys/wait.h>

int fd;

void signal\_handler(int signum) {

    printf("stop reading\n");

    close(fd);

    exit(0);

}

int main()

{

    pid\_t pid;

    pid=fork();

    if(pid!=0)//父进程，负责写，给别人发送消息

    {

        printf("this pid is %d\n",pid);

        int i;

        char msg[100];

        fd= open("/dev/ch\_device",O\_WRONLY,S\_IRUSR|S\_IWUSR);

        if(fd!=-1)

        {

            while(1)

            {

                for(i=0;i<101;i++)  //初始化

                    msg[i]='\0';

                printf("Please input the globar:\n");

                fgets(msg, sizeof(msg), stdin);

                msg[strcspn(msg, "\n")] = '\0';

                if(strcmp(msg,"quit()")==0)

                {

                    close(fd);

                    break;

                }

                write(fd,msg,strlen(msg));

            }

        }

        else

        {

            printf("device open failure\n");

        }

        printf("stop writing\n");

        kill(pid, SIGUSR1);

        wait(NULL);

        return 0;

    }

    else//子进程，负责读

    {

    signal(SIGUSR1, signal\_handler);

    int fd,i;

    char msg[101];

    fd= open("/dev/ch\_device",O\_RDONLY,S\_IRUSR|S\_IWUSR);

    if(fd!=-1)

    {

        while(1)

        {

            for(i=0;i<101;i++)  //初始化

                msg[i]='\0';

            read(fd,msg,100);

            printf("%s\n",msg);

        }

    }

    else

    {

        printf("device open failure,%d\n",fd);

    }

    return 0;

    }

}

```

**#### 测试文件实现既能发送又能接受消息的聊天端**

 ```测试文件为了模仿用户之间的通信，采用在一个终端创建子进程，父进程负责写（发送）消息，子进程负责读（接受）消息。

为了进一步实现父进程对子进程的控制，以避免用户退出父进程后子进程陷入僵尸进程，当用户输入quit()时，父进程向子进程发送结束信号，等待子进程结束后再结束自身，避免了僵尸进程。

```