西安交通大学

操作系统专题实验报告

班级：网安 2301

学号：2233611665

姓名：叶政

2025 年 12 月 20 日

目录

1 openEuler 系统环境实验

1.1 实验目的

（1）熟悉华为云服务器上 openEuler 操作系统基本系统环境；

（2）理解进程的创建、进程地址空间的概念、父子进程资源的关系；

（3）观察进程调度，了解进程调度的过程，了解孤儿进程和僵尸进程的区别；

（4）理解线程与进程的关系，对等线程间的关系。

（5）理解并运用信号量及其 PV 操作、自旋锁实现线程间的同步互斥。

1.2 实验思想

openEuler 基于 Linux 内核，支持多种架构，支持多种虚拟化技术，支持多种容器技术，支持多种云计算技术。openEuler 采用 Linux 内核，通过了解 openEuler 系统下的 shell 命令和系统命令后，所学习的知识可以轻松迁移到其他 Linux 发行版本中，如 Ubuntu。

通过配置华为云服务器上 openEuler 操作系统基本系统环境，运行 shell 命令查看系统信息以达到了解和使用 openEuler 操作系统的目的。

通过进程相关编程实验，编写和运行简单的进程、线程相关程序，理解进程与线程概念、进程空间与物理内存空间、进程调度、进程间变量管理、进程调用其他程序、如何实现正确的并发（同步互斥）等方面在操作系统中的实际操作。具体来讲，通过输出子进程和父进程的 PID，观察进程调度，了解进程调度的过程，了解孤儿进程和僵尸进程的区别。观察并发进程中的全局变量改变，输出父子进程共享变量地址以了解物理地址与虚地址概念，从而理解进程地址空间的概念，了解关于地址绑定的基础知识。创建两个线程运行后体会线程与进程的关系、对等线程间的资源共享以及同步与互斥的知识。

1.3 实验内容

（1）在华为云上搭建 openEuler 操作系统环境，通过运行 shell 命令查看系统信息以了解并熟悉 openEuler 操作系统。

（2）熟悉操作命令、编辑、编译、运行程序。完成给定程序（图 1.1 所示的程序）的运行验证，多运行程序几次观察结果；去除 wait 后再观察结果并进行理论分析。

（2）在所给程序中添加一个全局变量，在父进程和子进程中分别对这个变量做不同操作并输出操作结果，对结果进行观察并解释，同时输出两种变量的地址观察并分析。

（3）修改程序，在子进程中调用 system 函数和在子进程中调用 exec 族函数以执行自己写的一段程序，在此程序中输出进程 PID，进行比较分析。

（4）线程实验:在进程中给一变量赋初值并创建两个线程，在两个线程中分别对此变量循

环 5000 次以上做不同的操作并输出结果；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；在线程中调用 system 函数/exec 族函数以执行自己写的一段程序，输出进程 PID 和线程 TID，进行比较分析。

（5）自旋锁实验：在进程中给一变量赋初值并成功创建两个线程；在两个线程中分别对此变量循环 5000 次以上做不同的操作（自行设计）并输出结果；使用自旋锁实现互斥和同步。

```
#include <sys/types.h>
#include <stdio.h>
#include <unistd.h>

int main()
{
pid_t pid, pid1;

   /* fork a child process */
   pid = fork();

   if (pid < 0) { /* error occurred */
      fprintf(stderr, "Fork Failed");
      return 1;
   }
   else if (pid == 0) { /* child process */
      pid1 = getpid();
      printf("child: pid = %d",pid); /* A */
      printf("child: 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;
}
```

图 1.1 给定程序

1.4 实验步骤

（1）登录华为云,搭建 openEuler 操作系统环境。

（2）编辑、编译、运行给定的程序，多次运行观察实验结果。

（2）去除 wait()后观察结果并进行理论分析，比较程序在有无 wait()函数时的运行结果，分析 wait()函数的作用。

（3）添加一个全局变量，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，同时输出两种变量的地址并进行分析。

（4）在子进程中调用 system 函数执行自己写的一段程序，在此程序中输出进程 PID 和其父进程 PID 进行比较分析。

（5）在子进程中调用 exec 函数执行自己写的一段程序，在此程序中输出进程 PID 和其父进程 PID 进行比较分析。

（6）修改给定程序，给一变量赋初值并创建两个线程，在两个线程中分别对此变量循环 5000 次做不同的操作并输出结果。

（7）多运行几遍程序观察运行结果。

（8）完成上述过程的同步和互斥操作。

（9）在两个线程中调用 system 函数执行自己写的一段程序，在此程序中输出线程 TID 及进程 PID,进行分析。

（10）在两个线程中调用 exec 函数执行自己写的一段程序，在此程序中输出线程 TID 及进程 PID,进行分析。

（11）参考实验指导书，编写模拟自旋锁程序代码，补充主函数代码，用自旋锁实现线程间的同步。编译并运行程序，分析运行结果

1.5 测试数据设计

(1)实验测试数据为父子进程分别利用 fork 生成不同的 pid 号，并且和本进程的 pid 号(pid1)，如果是子进程 pid 是 0，父进程就是子进程的 ID 号，父进程 pid1 生成的是自己进程的 pid1 号。

(2) 删除 1.1 中的 wait 函数，父进程不必等待子进程结束才结束。

用于观察函数输出是否又区别对比有 wait()函数。

(3) 增加全局变量 glo 初始 100，父进程减 50，子进程加 50。观察每个进程中最后 glo 的输出值。

(4) 在 1.3 的基础上增加父进程除于 10，子进程乘于 10。用于对最后返回前的观察。

(5) 在进程调用 system()函数和 exec()函数，观察不同函数输出的进程号以及父进程号。

(6)创建两个线程分别对一个公共变量 count 操作，分别进行加 100 减 100 各 5000 次。观察最后的结果是多少。

(7)运用信号量对 1.6 实验 pv 操作下进行。定义一个信号量 signal。

(8)在两个线程中调用分别调用 system()函数和 exec 函数分别输出 pid 和 tid。由于 exec()会使当前进程为 system_call 的进程导致另一个线程无法实现调用。

(9)运用自旋锁并创造两个线程对一个共享变量操作分别加 100 减 100 各 5000 次。自旋锁是忙等待的锁。

1.6 程序运行初值及运行结果分析

1.1

```
[root@kp-mytest test1]# ./103
child: pid is 0child: pid1 is 2449parent: pid is 2449parent: pid1 is 2448[root@kp-mytest test1]# ./103
child: pid is 0child: pid1 is 2451parent: pid is 2451parent: pid1 is 2450[root@kp-mytest test1]# ./103
child: pid is 0child: pid1 is 2453parent: pid is 2453parent: pid1 is 2452[root@kp-mytest test1]# ./103
child: pid is 0child: pid1 is 2455parent: pid is 2455parent: pid1 is 2454[root@kp-mytest test1]# ./103
child: pid is 0child: pid1 is 2457parent: pid is 2457parent: pid1 is 2456[root@kp-mytest test1]# ./103
child: pid is 0child: pid1 is 2459parent: pid is 2459parent: pid1 is 2458[root@kp-mytest test1]# ./103
child: pid is 0child: pid1 is 2461parent: pid is 2461parent: pid1 is 2460[root@kp-mytest test1]# ./103
child: pid is 0child: pid1 is 2463parent: pid is 2463parent: pid1 is 2462[root@kp-mytest test1]# ./103
child: pid is 0child: pid1 is 2465parent: pid is 2465parent: pid1 is 2464[root@kp-mytest test1]#
```

1.2

```
[root@kp-mytest test1]# gcc -o 104 104.c
[root@kp-mytest test1]# ./104
parent: pid = 2473 parent: pid1 = 2472 child: pid = 0 child: pid1 = 2473 [root@kp-mytest test1]# ./104
parent: pid = 2475 parent: pid1 = 2474 child: pid = 0 child: pid1 = 2475 [root@kp-mytest test1]# ./104
child: pid = 0 child: pid1 = 2477 parent: pid = 2477 parent: pid1 = 2476 [root@kp-mytest test1]# ./104
parent: pid = 2479 parent: pid1 = 2478 child: pid = 0 child: pid1 = 2479 [root@kp-mytest test1]# ./104
parent: pid = 2481 parent: pid1 = 2480 child: pid = 0 child: pid1 = 2481 [root@kp-mytest test1]# ./104
parent: pid = 2483 parent: pid1 = 2482 child: pid = 0 child: pid1 = 2483 [root@kp-mytest test1]# ./10
-bash: ./10: No such file or directory
[root@kp-mytest test1]# ./104
parent: pid = 2486 parent: pid1 = 2485 child: pid = 0 child: pid1 = 2486 [root@kp-mytest test1]# ./104
parent: pid = 2488 parent: pid1 = 2487 child: pid = 0 child: pid1 = 2488 [root@kp-mytest test1]# ./104
parent: pid = 2490 parent: pid1 = 2489 child: pid = 0 child: pid1 = 2490 [root@kp-mytest test1]# ./104
parent: pid = 2492 parent: pid1 = 2491 child: pid = 0 child: pid1 = 2492 [root@kp-mytest test1]#
```

第一个是没有去除 wait，第二个是去除 wait 后的结果。

观察结果发现有 wait(NULL)的函数其中父进程由于 wait(NULL)的作用，导致父进程必须等待子进程结束后再结束，所以结果表现出输出父进程的 printf 晚于子进程的 printf，

但是在去除后父进程和子进程两个的执行顺序就是不固定的可能父进程的 printf 会早于子进程也可能会晚于。

1.3

```
child: pid is 0 child: pid1 is 2542 glo= 150 parent: pid is 2542 parent: pid1 is 2541 glo= 50 [root@kp-mytest test1]# ./
1-3
child: pid is 0 child: pid1 is 2544 glo= 150 parent: pid is 2544 parent: pid1 is 2543 glo= 50 [root@kp-mytest test1]# ./
1-3
child: pid is 0 child: pid1 is 2546 glo= 150 parent: pid is 2546 parent: pid1 is 2545 glo= 50 [root@kp-mytest test1]# ./
1-3
child: pid is 0 child: pid1 is 2548 glo= 150 parent: pid is 2548 parent: pid1 is 2547 glo= 50 [root@kp-mytest test1]# ./
1-3
```

我们添加了一个全局变量加入了 glo，初始设定为 100，在父进程减去 50，子进程加 50，最后得到的结果为子进程 150，父进程 50。

1.4

```
child: pid is 0 child: pid1 is 2676 glo= 150 child before return glo= 1500 parent: pid is 2676 parent: pid1 is 2675 glo=
 50 parent before return glo= 5 [root@kp-mytest test1]# ./1-4
child: pid is 0 child: pid1 is 2678 glo= 150 child before return glo= 1500 parent: pid is 2678 parent: pid1 is 2677 glo=
 50 parent before return glo= 5 [root@kp-mytest test1]# ./1-4
child: pid is 0 child: pid1 is 2680 glo= 150 child before return glo= 1500 parent: pid is 2680 parent: pid1 is 2679 glo=
 50 parent before return glo= 5 [root@kp-mytest test1]# ./1-4
child: pid is 0 child: pid1 is 2682 glo= 150 child before return glo= 1500 parent: pid is 2682 parent: pid1 is 2681 glo=
 50 parent before return glo= 5 [root@kp-mytest test1]# ./1-4
child: pid is 0 child: pid1 is 2684 glo= 150 child before return glo= 1500 parent: pid is 2684 parent: pid1 is 2683 glo=
 50 parent before return glo= 5 [root@kp-mytest test1]# ./1-4
child: pid is 0 child: pid1 is 2686 glo= 150 child before return glo= 1500 parent: pid is 2686 parent: pid1 is 2685 glo=
 50 parent before return glo= 5 [root@kp-mytest test1]#
```

在步骤三的基础上我们通过增加在子进程和父进程乘 10 和除 10 的不同操作最终得到不同的结果。观察发现在 150 和 50 的基础上最后变化为 1500 和 5 分别由 150*10 和 50/10 得到的。

1.5

```
[root@kp-mytest test1]# ./1-5
parent: pid = 3047, pid1 = 3046
child: pid = 0, pid1 = 3047

--- 子进程调用 system() ---
system_call: pid=3048, ppid=3047

--- 子进程调用 exec() ---
system_call: pid=3047, ppid=3046
[root@kp-mytest test1]# ./1-5
parent: pid = 3050, pid1 = 3049
child: pid = 0, pid1 = 3050

--- 子进程调用 system() ---
system_call: pid=3051, ppid=3050

--- 子进程调用 exec() ---
system_call: pid=3050, ppid=3049
[root@kp-mytest test1]# ./1-5
parent: pid = 3053, pid1 = 3052
child: pid = 0, pid1 = 3053

--- 子进程调用 system() ---
system_call: pid=3054, ppid=3053

--- 子进程调用 exec() ---
system_call: pid=3053, ppid=3052
[root@kp-mytest test1]#
```

我们分别调用了 system()和 excel()两个函数。

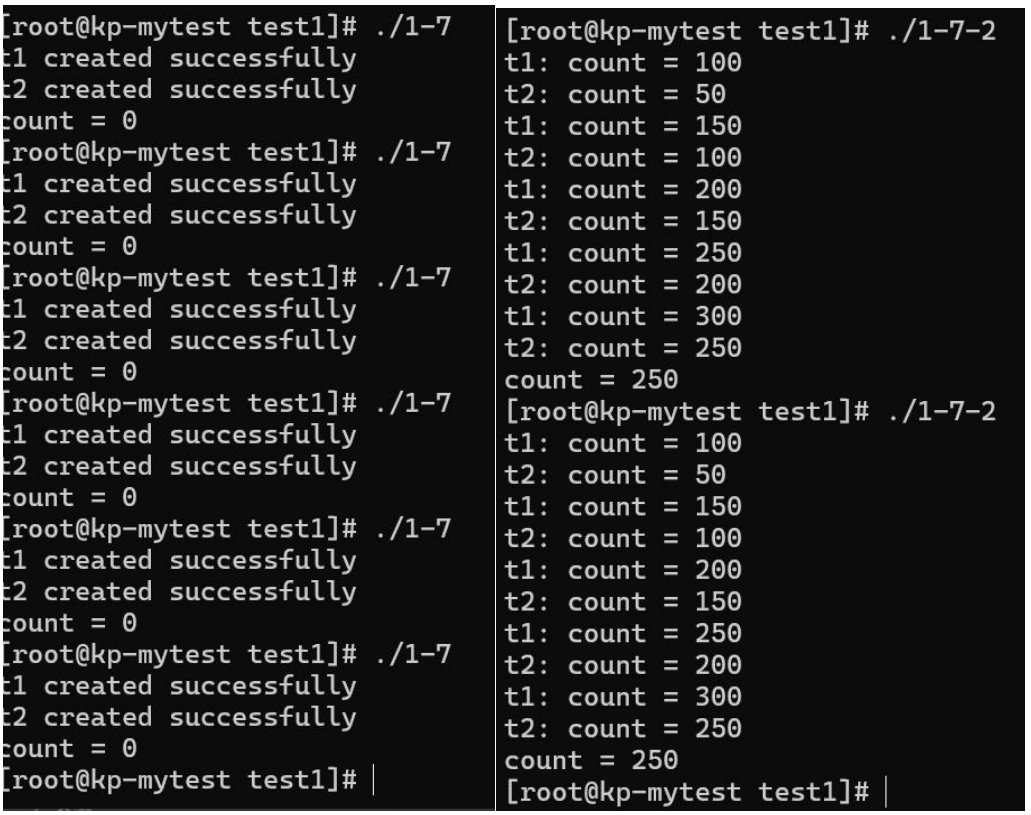
观察得知在 system()函数是在原来子进程新开辟一个新进程，原来子进程的 pid 为 3053，但是 system()调用的 system_call 的子进程是新开辟的，所以 pid 号是 3054，但是我们观察在 execl ()调用的 system_call()的 pid 号是 3053 说明和原来子进程是同一个进程，由于 execl()是在原来子进程上替换说明 pid 号也是相等的。

1.6

```
[root@kp-mytest test1]# gcc -o 1-6 1-6.c -pthread
[root@kp-mytest test1]# ./1-6
count = 19900
[root@kp-mytest test1]# ./1-6
count = -22500
[root@kp-mytest test1]# ./1-6
count = -700
[root@kp-mytest test1]# ./1-6
count = 4100
[root@kp-mytest test1]# ./1-6
count = 0
[root@kp-mytest test1]# ./1-6
count = -3000
[root@kp-mytest test1]# ./1-6
count = -7600
[root@kp-mytest test1]#
```

我们观察结果发现由于没有任何锁，导致代码最终输出结果不相同，按照正常的预想 count=0，但是线程对公共变量的访问出现问题，导致最终出现竞态，从而出现 count 的结果不一致。

1.7



观察发现使用 pv 操作后，两个线程没有出现竞态，最终的输出的结果为 0。符合正常的预期。与步骤一不相同的结果是由于在进行每个线程时由于 pv 操作，另一个线程无法访问临界区，只能在等待另一个线程结束后再访问。这是因为再没有 pv 操作时，加入同时访问临界区同时读取 count=0，线程一的计算寄存器结果是+100,而线程二的计算寄存器的结果是-100，同时写回内存结果可能是-100 或者+100，但是有了 pv 操作每次只有一个线程能进入修改 count，所以最终结果是 0。

在图二中我们规定线程 2 必须等线程 1 执行完再执行。我们修改代码线程 1 和 2 各只有五次循环。线程一每次循环+100，线程 2 每次循环-50。每次循环必须等线程 1 的一次循环结束后线程二才可以进行一次循环从而交替。这是由于

```
sem_wait(&signal1); // 等待信号量 1
            count += 100;
            printf("t1: count = %d\n", count);
            sem_post(&signal2); // 线程 2 可以执行
  sem_wait(&signal2); // 等待线程 1 释放信号量
```

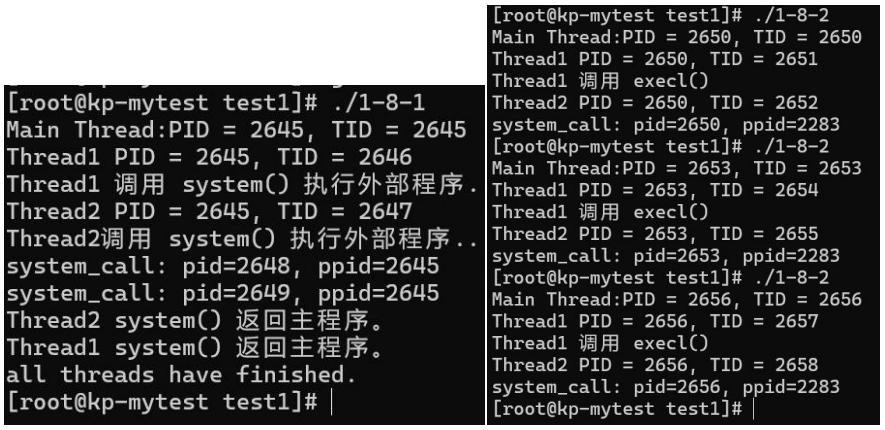
```
count -= 50;

printf("t2: count = %d\n", count);

sem_post(&signal1); // 线程 1 继续
```

代码就是首先运行线程一的一次循环完从而 signal2+1 就可以运行线程二的循环。运行线程二的一次循环后 signal1+1 就可以再次运行线程一循环而线程二还在等待线程一释放信号量。

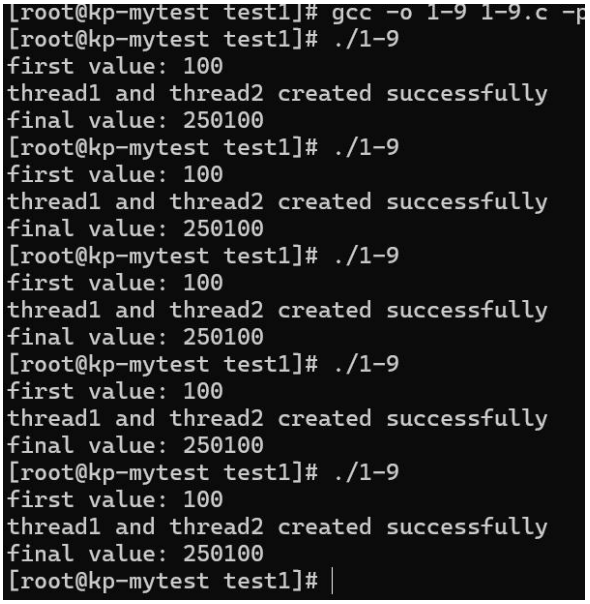
1.8



图一调用的是 system()，主进程的 pid 为 2645，主线程的 tid 等于进程 pid。两个线程由于是同一个进程所以进程 pid 号相等，但是 tid 号不相等，代表不同的线程。System()调用的是创立两个新进程所以 pid 号与主进程不相等。

图二是调用 execl()，因此当线程一调用的时候因此该进程变成 system_call 的进程。Pid 号还是原来的主进程的 pid 号，但是线程二无法调用因为整个进程已经变成 system_call()。所以只有线程一调用 system_call()输出 pid 号。

1.9



我们发现多次运行自旋锁的主进程，分别运行两个线程并且观察结果是正确的，没有出现同时访问临界区的情况。我们初始设置代码的共享变量为 100，分别进行+100 和-50 各 5000 次操作操作得到最终结果相当于+250000。

1.7 实验总结

1.7.1 实验中的问题与解决过程

实验中遇到编译错误发现需要引入一个库函数就可以解决。

发现是未导入库函数#include<sys/wait.h>

实验中发现加入 wait 函数后并没有出现先子进程和父进程原因是由于加入了/立马刷新了缓冲区。因为 Linux 下缓冲区的作用（将输出结果打印到终端） 把 linux 进程日志 printf 调取到 终端显示-CSDN 博客

从而代码出现每次刷新缓冲区导致父进程的 printf 和子进程的 printf 是交错输出。

1.7.2 实验收获

在华为云上成功搭建 openEuler 环境，通过 uname -a、cat /etc/os-release 等命令查看系统与硬件信息，熟悉了常用操作命令、编辑编译与程序运行流程。通过 fork() 实验理解父子进程从同一位置继续执行但返回值不同、调度顺序不确定；多次运行发现输出顺序可能变化，加入 wait() 后父进程等待子进程结束可减少输出交错与父进程先退出等现象，去除 wait() 后随机性更明显，从而认识到进程同步的重要性。在程序中添加全局变量并在父子进程分别修改后观察到互不影响，说明 fork() 后地址空间逻辑独立；同时打印变量地址发现父子进程可能出现相同虚拟地址但数据不共享，结合写时复制（COW）机制解释了“地址看似相同但修改互不影响”的原因。进一步在子进程中分别调用 system() 与 exec 族执行自编程序并输出 PID 对比，验证 system() 通常会通过 shell 间接创建新进程并返回继续执行，而 exec 会替换当前进程映像使 PID 不变且后续代码不再执行。在线程实验中创建两个线程对共享变量循环 5000 次以上做不同操作，多次运行若结果不同则表明存在数据竞争（读-改-写非原子），

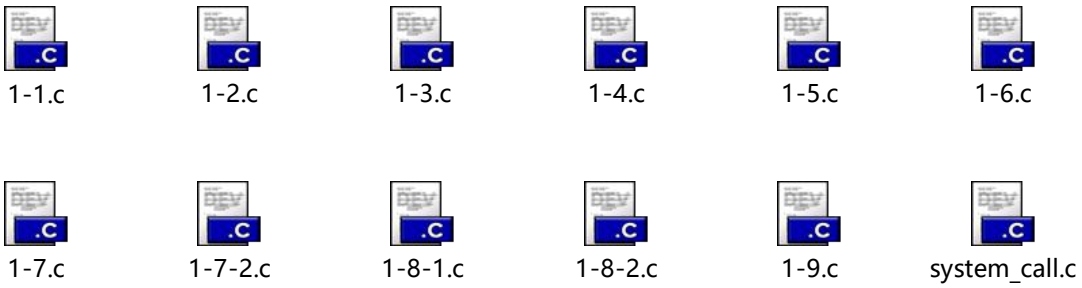
通过引入互斥机制（如信号量/互斥锁）后结果稳定；同时利用同步手段可实现严格交替等顺序控制。在线程中调用 system/exec 并输出 PID 与 TID，确认同一进程内线程 PID 相同而 TID 不同，system() 会派生新进程执行命令；最后通过自旋锁实现互斥与同步，体会到其适合短临界区但会忙等占用 CPU，需结合临界区长度与系统负载选择自旋锁或阻塞型锁。

1.7.3 意见与建议

希望多增加代码实践。

1.8 附件

1.8.1 附件 1 程序



1.8.2 附件 2 Readme

2 进程通信与内存管理

2.1 实验目的

进程的软中断通信

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。

进程的管道通信

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。

内存的分配与回收

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。

2.2 实验内容

进程的软中断通信

（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!!

注： delete 会向进程发送 SIGINT 信号，quit 会向进程发送 SIGQUIT 信号。 ctrl+c 为

delete，ctrl+\为 quit 。

参考资料 https://blog.csdn.net/mylizh/article/details/38385739

（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。

进程的管道通信

（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。

（3）修改上述程序，让其中两个子进程各向管道写入 2000 个字符，父进程从管道中读出，

分有锁和无锁的情况。运行程序，分别观察有锁和无锁情况下的写入读出情况。

内存的分配与回收

（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 实验思想

（软中断通信）

用 fork 创建并发执行的进程实体

父进程先后调用两次 fork() 得到两个子进程。父子进程拥有各自独立的执行流与内存空间，

但可通过信号这种内核机制进行“异步通知”。

用 signal 建立“中断事件→处理函数”的映射

父进程通过 signal(SIGINT, handler) 和 signal(SIGQUIT, handler) 捕捉来自键盘的软中断（Ctrl+C 对应 SIGINT，Ctrl+\ 对应 SIGQUIT）。这样父进程不再采用默认处理（直接终止），而是转而执行自定义 handler，实现“收到中断后执行通信逻辑”。

把“键盘中断”转换为“父向子发信号”的通信触发器

实验的通信本质是：

键盘产生的 SIGINT/SIGQUIT → 父进程捕获 → 父进程用 kill 向两个子进程发送指定信号（16/17）

因此信号在这里扮演了“消息/事件”的角色，完成父子进程之间的软中断通信。

子进程用阻塞等待保证“已准备好接收信号”

因为信号可能在子进程尚未进入等待状态前就发出，所以子进程需要先注册对 16/17 的处理方式，并进入阻塞等待（常见做法是 pause() 或循环 sleep()/检查标志位），确保：

子进程“先准备（装好 handler）”

再等待父进程发信号

收到信号后，子进程在 handler 中设置标志位/直接打印并退出，从而体现“信号到达→异步打断→执行处理”的特点。

父进程用 wait 回收子进程，体现进程生命周期管理

父进程在发出 kill 后调用 wait()/waitpid() 等待两个子进程退出，避免产生僵尸进程，并在确认子进程终止后输出 “Parent process is killed!!” 完成收尾。

“5 秒窗口”体现信号的两种触发路径：外部事件 or 超时事件

实验要求 5s 内按键触发，或 5s 内无操作也触发，这体现两类软中断来源：

外部输入事件：SIGINT / SIGQUIT（键盘）

计时事件：SIGALRM（14 号闹钟信号），用 alarm(5) 在超时后自动产生中断

二者都能进入同一套 handler 逻辑，帮助理解“信号是统一的异步事件机制，只是来源不同”。

多次运行现象总结的思想：不确定性来自调度与异步信号

子进程打印“被杀死”的先后顺序不固定，是因为：

内核调度顺序不可控

信号投递与处理是异步的

但通过“子进程阻塞等待 + 父进程 wait 回收”可保证最终行为稳定：两子进程都会被父进程发送的信号终止，父进程最后退出。

（进程的管道通信）

用 pipe 建立父子进程间的数据通道

通过 pipe(fd) 创建匿名管道，得到两个文件描述符：fd[0] 读端、fd[1] 写端。再用 fork() 创建子进程，使父子进程继承同一管道端点，从而形成“写进程→管道→读进程”的字符流通

信链路。

用“关闭不用的端口”明确通信方向并避免死锁

通信双方各自只保留需要的一端：

写进程关闭读端 close(fd[0])

读进程关闭写端 close(fd[1])

这样既能避免误用端口，也能保证当所有写端关闭时，读端能读到 EOF，体现“对方是否存在”的判定机制。

管道自身提供同步：满则写阻塞、空则读阻塞

管道是内核缓冲区：

当管道缓冲区满时，写进程 write() 会阻塞，直到读进程读走数据；

当管道缓冲区空时，读进程 read() 会阻塞，直到写进程写入数据。

因此管道天然具备“生产者-消费者”式的同步特性。

互斥不等同于同步：多写者场景需要额外互斥控制

若只有一个写进程、一个读进程，管道的阻塞机制通常足以保证基本同步。但当存在多个写进程同时写同一管道时，会出现“写入交错/数据混杂”的现象：

无锁：两个子进程的输出可能穿插，导致父进程读出的字符流顺序不稳定、甚至分块混乱；

有锁：用互斥锁/文件锁（如 lockf、信号量等）把“写一次/写一段”作为临界区保护起来，保证一次写入的整体性，从而实现互斥，让读出结果更可控、更可复现。

对比实验的核心：观察“有锁 vs 无锁”的现象差异

按流程图补全程序后，通过多次运行比较：

无锁：数据更容易交错，读出的内容顺序随机，体现并发竞争；

加锁：写操作串行化，读出更接近“按块/按次完整输出”，体现互斥的作用。

修改为两个子进程各写 2000 字符后，现象更明显：无锁更易混杂，有锁更清晰，直观展示“同步（管道阻塞）+互斥（锁保护写临界区）”共同保证通信正确性。

（内存的分配与回收：FF / BF / WF）

用“动态分区 + 空闲分区表/链”模拟连续内存管理

把用户区内存抽象为一段连续地址空间，用**空闲分区链表（free list）**记录当前所有空闲块（起始地址 start、大小 size）。每次分配就是从空闲表中挑一块“足够大”的空闲分区进行切割；每次回收就是把释放区插回空闲表并尝试与相邻空闲块合并，从而模拟 OS 的动态分区管理过程。

三种分配算法的核心差异：选哪一块空闲分区

FF（First Fit 首次适应）：空闲表按地址递增，从头找第一个能放下请求的块。思想是“尽快找到、减少查找开销”。

BF（Best Fit 最佳适应）：空闲表按大小递增（或扫描找最小可用块），选择“刚好够用”的最小块。思想是“尽量少浪费，降低外部碎片的即时增长”。

WF（Worst Fit 最差适应）：空闲表按大小递减（或扫描找最大块），优先用最大空闲块分配。思想是“避免产生大量小碎片，尽量保留中等块的可用性”。

分配的通用步骤：查找 → 切割 → 更新表

对每次请求大小 R：

在空闲表中按对应策略找到满足 size >= R 的空闲块；

若 size == R：整块分配，删除该空闲结点；

若 size > R：执行分割，分配前 R 给进程，剩余部分生成/更新为空闲块结点；

维护空闲表的排序规则（FF 按地址；BF/WF 按大小）。

回收的通用步骤：插入 → 相邻合并（消除碎片）

释放一块 [start, size]：

先把该块按规则插入空闲表（通常按地址更利于合并）；

判断是否与前驱/后继空闲块在地址上相邻（prev.end == start、start+size == next.start），出现三种情况：只与前合并、只与后合并、前后都合并；

合并后更新结点，必要时删除多余结点。

实验强调：回收是否合并直接影响外部碎片积累速度与后续分配成功率。

碎片与紧缩：当“总空闲足够但不连续”时如何处理

外部碎片会导致：虽然 sum(free) >= R，但没有任何单块 >= R。这时需要紧缩（compaction）：

将已分配分区向某一方向移动（在模拟中可把已分配块重新排列到低地址连续区），

把分散空闲块合并成一个大空闲区，再尝试分配。

紧缩能提高可分配性，但代价高（真实系统涉及搬移与重定位开销），实验用于体会“效率 vs 利用率”的权衡。

对比分析的实验重点：效率、利用率、碎片化趋势

通过同一组申请/释放序列模拟三种算法，比较：

查找开销（FF 通常更快；BF/WF 可能需要更长扫描或维护排序）

外部碎片情况（BF 容易留下很多很小碎片；WF 倾向保留较大的剩余块；FF 介于两者之间且易在低地址产生碎片带）

分配成功率与紧缩触发频率（碎片越多越容易触发紧缩）

从而理解不同策略在“速度（分配效率）”与“空间（利用率）”上的取舍。

2.4 实验步骤

(1)写全软中断通信代码，观察 5s 内运行不同中断和 5s 后中断的运行结果

(2)改为闹钟中断后，观察 5s 后中断和 5s 内中断的结果，并与之前比较。

(3)补充完整管道通信代码，观察有锁和无锁情况下的管道通信程序的结果。

(4)修改上述程序实现两个子进程向管道分别写入 2000 个字符，父进程向管道读出，并且分有锁和无锁的情况。

(5)补全内存的分配和回收的代码，实现 FF,BF,WF 三种策略，并且每个要实现内存分配、回收、空闲块排序以及合并、紧缩等功能。

2.5 测试数据设计

(1)软中断通信观察不同中断后的输出，如果是 5s 内中断则结果输出一致，如果不是则会输出无信号量输入在 5s 内。

(2)管道通信通过观察父进程输出的结果就能观察是否有锁的输出下的结果。

(3)内存的分配和回收实验，我们通过观察随机的生成内存分布下三个算法下对新的内存如何分配结果进行观察。

2.6 程序运行初值及运行结果分析

(1)进程的软中断通信

```
[root@kp-mytest test2]# ./2-1
Parent waiting 5 seconds
^CParent received delete or quit


Child process1 is killed by parent!!
Child process2 is killed by parent!!

Parent process is killed!!
[root@kp-mytest test2]# ./2-1
Parent waiting 5 seconds
^CParent received delete or quit


Child process2 is killed by parent!!
Child process1 is killed by parent!!

Parent process is killed!!
[root@kp-mytest test2]# ./2-1
Parent waiting 5 seconds
^\Parent received delete or quit

Child process1 is killed by parent!!

Child process2 is killed by parent!!

Parent process is killed!!
[root@kp-mytest test2]#
```

代码结果是这样的

我们观察子进程 1 和 2 在收到信号 16 17 后终止

我们通过建立管道 pipefd 保证父进程在子进程安装好信号处理器后才开始发送

因为

```
char buf;
      read(pipefd[0], &buf, 1);
      read(pipefd[0], &buf, 1);
      close(pipefd[0]);
```

只有父进程在 pipefd 中读取到东西才运行下面的代码否则就阻塞住在这里。

为什么有两个 read,是因为有两个进程，所以保证每个子进程都写入才可以发送信号。

/ 子进程等待父进程信号

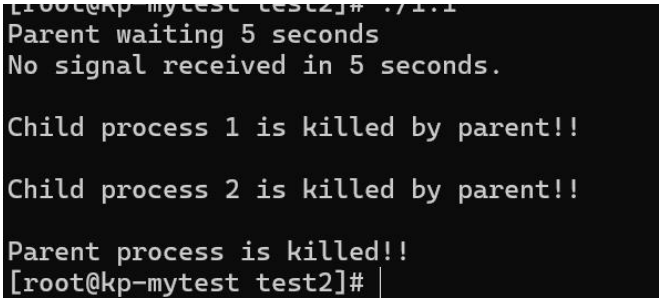
```
void waiting() {
	while (1) pause();
}
```

子进程通过 pause()函数等待，保证子进程一直阻塞中，等待父进程的信号。

不等待则会出现子进程结束的情况。

这个是过 5s 后自动中断的结果。通过 5s 过后父进程自动给子进程发送终止信号。

```
printf("Parent waiting 5 seconds\n");
	sleep(5);
	if (flag == 1)
		printf("Parent received DELETE   or QUIT signal\n");
	else
		printf("No signal received in 5 seconds.\n");
	// 向两个子进程发送信号
	kill(pid1, 16);
	kill(pid2, 17);
```



这个是修改为闹钟中断的信号。分别是 5s 内中断和 5s 过后中断。

```
[root@kp-mytest test2]# ./1.2
Parent waiting 5 seconds
No signal received in 5 seconds.

Child process1 is killed by parent!!

Child process2 is killed by parent!!

Parent process is killed!!
[root@kp-mytest test2]# ./1.2
Parent waiting 5 seconds
No signal received in 5 seconds.

Child process1 is killed by parent!!

Child process2 is killed by parent!!

Parent process is killed!!
[root@kp-mytest test2]#
```

结果基本一致。有输入中断没什么区别，无输入时是由于

设置

alarm(5);

pause();

配合

signal(SIGALRM, inter_handlerf) ;

实现父进程由闹钟信号唤醒，代码继续走，向子进程发送信号，结束子进程，最后结束父进程。

(2)　　进程的管道通信

左边是有锁的结果，右边是无锁的情况。

```
[root@kp-mytest test2]# ./2.1
Child Process 1 is sending message
Child Process 2 is sending message

child process 2 is exiting
child process 1 is exiting
Parent process exit
[root@kp-mytest test2]# ./2.1
Child Process 1 is sending message
Child Process 2 is sending message

child process 2 is exiting
child process 1 is exiting
Parent process exit
[root@kp-mytest test2]# ./2.1
Child Process 1 is sending message
Child Process 2 is sending message

child process 2 is exiting
child process 1 is exiting
Parent process exit
[root@kp-mytest test2]#
```

```
CChhiilldd  PPrrooccessss  12  iiss  sseennddiingn gm emsessasaggee


child process 2 is exiting
child process 1 is exiting
Parent process exit
[root@kp-mytest test2]# ./2.2
CChhiilldd  PPrrooccessss  12  iiss  sseennddiinngg m emsessasgaeg
e

child process 1 is exiting
child process 2 is exiting
Parent process exit
```

两个进程通过锁管道进行互斥。首先其中一个管道对管道写时进行锁

```
lockf(fd[1], 1, 0);
```

从而保证另一个进程想要对管道写时，被阻塞等待直到另一个进程写完。并解锁。

```
lockf(fd[1], 0, 0);
```

增加两个管道进行同步。

```
pipe(wa); // 父进程提醒子进程结束信号
pipe(fi);   //子进程提醒父进程开始信号
```

子进程向父进程写，而父进程一直等待直到读出数据负责堵塞保证父进程在子进程写完数据再开始读数据通道。

```
write(fi[1],"1",1);  // 子
read(fi[0],&sig,1);  //父
```

父进程也通过管道提醒子进程结束信号

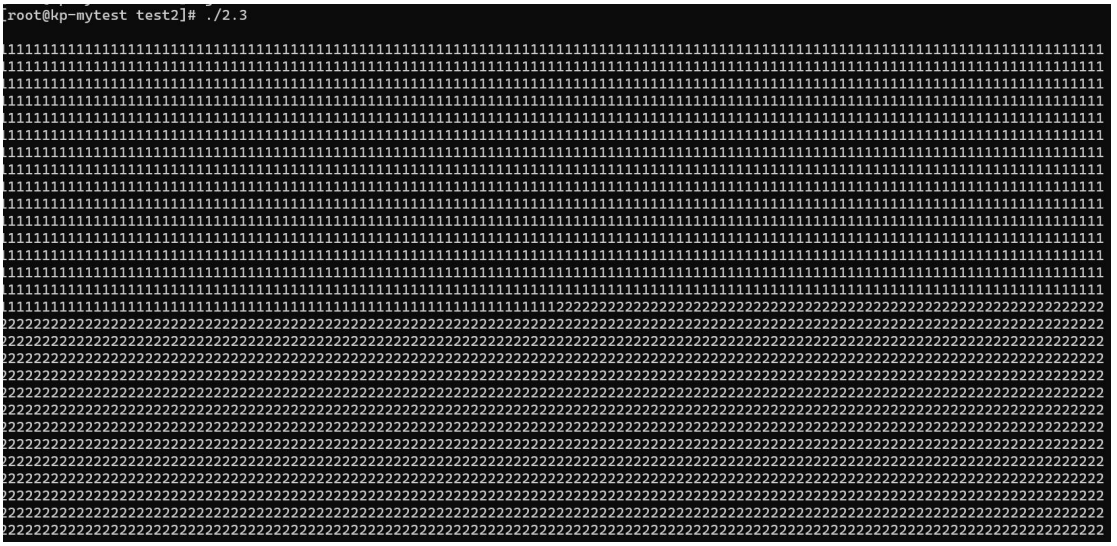
```
write(wa[1], "1", 1);  //父
read(wa[0], &buf, 1);  //子
```

当没有上锁时每个子进程自然就可以访问缓冲区，传递数据，但是不互斥，导致乱码。

现在修改代码改成输入 2000 字符串

第一个是有锁第二个是无锁。

当没有锁的时候结果是 1 和 2 输出是混乱的，是由于进程一和进程二向通道写入的时候没有先后顺序导致结果自然也是随机的。

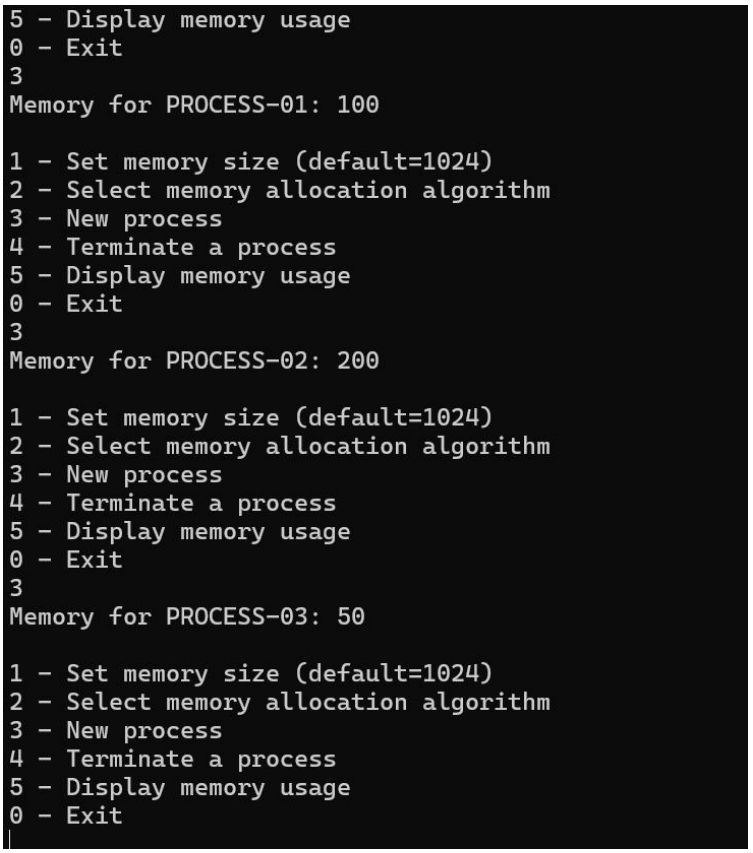


(3) 内存的分配与回收

三个算法的主要区别就是在选择分配块的区别

FF:在链表结构中选取按顺序，选取第一个能适应大小的空闲分区。

开始创建三个进程，分别 100 ，200 ，50



我们删除第二个进程,并创建第四个进程 150 大小。

```
1 - Set memory size (default=1024)
2 - Select memory allocation algorithm
3 - New process
4 - Terminate a process
5 - Display memory usage
0 - Exit
4
Kill Process, pid=2

1 - Set memory size (default=1024)
2 - Select memory allocation algorithm
3 - New process
4 - Terminate a process
5 - Display memory usage
0 - Exit
3
Memory for PROCESS-04: 150

1 - Set memory size (default=1024)
2 - Select memory allocation algorithm
3 - New process
4 - Terminate a process
5 - Display memory usage
0 - Exit
```

我们观察内存，发现进程 4 的内存放在删去了进程 2 的空闲区。

```
1 - Set memory size (default=1024)
2 - Select memory allocation algorithm
3 - New process
4 - Terminate a process
5 - Display memory usage
0 - Exit
5
------------------------------------------------------------
Free Memory:
         start_addr                size
                250                  50
                350                 674

Used Memory:
      PID          ProcessName start_addr       size
        4           PROCESS-04        100        150
        3           PROCESS-03        300         50
        1           PROCESS-01          0        100
------------------------------------------------------------
```

BF:

我们创建五个进程，分别大小为 100 200 50 150 50。

我们删去 2 和 4 进程。

留下 1 3 5，创立新进程大小为 120 现在按顺序有 200 和 150 空闲区域。

```
------------------------------------------------------------
Free Memory:
          start_addr                size
                 550                 474

Used Memory:
       PID          ProcessName start_addr       size
         5           PROCESS-05        500         50
         4           PROCESS-04        350        150
         3           PROCESS-03        300         50
         2           PROCESS-02        100        200
         1           PROCESS-01          0        100
------------------------------------------------------------

1 - Set memory size (default=1024)
2 - Select memory allocation algorithm
3 - New process
4 - Terminate a process
5 - Display memory usage
0 - Exit
```

我们删去 2 ，4 进程

空闲区域为 100-300 和 350-500。

由于是 bset fit 算法首先填充最小的空闲分区。

```
------------------------------------------------------------
Free Memory:
          start_addr                size
                 350                 150
                 100                 200
                 550                 474

Used Memory:
       PID          ProcessName start_addr       size
         5           PROCESS-05        500         50
         3           PROCESS-03        300         50
         1           PROCESS-01          0        100
------------------------------------------------------------

1 - Set memory size (default=1024)
2 - Select memory allocation algorithm
3 - New process
4 - Terminate a process
5 - Display memory usage
0 - Exit
```

我们观察进程 6 填充在 350-500 的空间，而 100-300 的空间没有使用。

```
------------------------------------------------------------
Free Memory:
          start_addr                size
                 470                  30
                 100                 200
                 550                 474

Used Memory:
       PID          ProcessName start_addr       size
         6           PROCESS-06        350        120
         5           PROCESS-05        500         50
         3           PROCESS-03        300         50
         1           PROCESS-01          0        100
------------------------------------------------------------

1 - Set memory size (default=1024)
2 - Select memory allocation algorithm
3 - New process
4 - Terminate a process
5 - Display memory usage
0 - Exit
```

WF:

我们创建三个进程 1 2 3 分别大小为 100 150 200。

```
1 - Set memory size (default=1024)
2 - Select memory allocation algorithm
3 - New process
4 - Terminate a process
5 - Display memory usage
0 - Exit
5
------------------------------------------------------------
Free Memory:
          start_addr                size
                 450                 574

Used Memory:
       PID          ProcessName start_addr       size
         3           PROCESS-03        250        200
         2           PROCESS-02        100        150
         1           PROCESS-01          0        100
------------------------------------------------------------
```

我们删去进程 2,得到的空闲空间为 100-250 和 450-1024 这两个空闲分区。

```
0 - Exit
4
Kill Process, pid=2

1 - Set memory size (default=1024)
2 - Select memory allocation algorithm
3 - New process
4 - Terminate a process
5 - Display memory usage
0 - Exit
5
------------------------------------------------------------
Free Memory:
          start_addr                size
                 450                 574
                 100                 150

Used Memory:
       PID          ProcessName start_addr       size
         3           PROCESS-03        250        200
         1           PROCESS-01          0        100
------------------------------------------------------------
```

创建进程 4 ，大小为 120 ，观察填充区域

```
1 - Set memory size (default=1024)
2 - Select memory allocation algorithm
3 - New process
4 - Terminate a process
5 - Display memory usage
0 - Exit
3
Memory for PROCESS-04: 120
```

查看内存使用图，发现进程 4 没有分配在 100-250 空间而是在 450-1024 的空闲分区中分割出一部分使用。

```
1 - Set memory size (default=1024)
2 - Select memory allocation algorithm
3 - New process
4 - Terminate a process
5 - Display memory usage
0 - Exit
5
------------------------------------------------------------
Free Memory:
          start_addr                size
                 570                 454
                 100                 150

Used Memory:
       PID          ProcessName start_addr       size
         4           PROCESS-04        450        120
         3           PROCESS-03        250        200
         1           PROCESS-01          0        100
------------------------------------------------------------
```

我们观察空闲分区的合并，当前进程分别占用 0-100 和 200-300，我们观察释放进程 1 后，空闲分区的分布。

```
-------------------------------------------------------------
Free Memory:
          start_addr                 size
                 100                  100
                 300                  724

Used Memory:
       PID          ProcessName start_addr       size
         3           PROCESS-03        200        100
         1           PROCESS-01          0        100
-------------------------------------------------------------
```

删除进程 1 后，我们发现还是只有两个空闲分区。

原本是 100-200，但是进程 1 被释放后，产生 0-100 的空闲分区，0-100，100-200 被合并后产生 0-200 的空闲分区。

```
Kill Process, pid=1

1 - Set memory size (default=1024)
2 - Select memory allocation algorithm
3 - New process
4 - Terminate a process
5 - Display memory usage
0 - Exit
5
-------------------------------------------------------------
Free Memory:
          start_addr                 size
                   0                  200
                 300                  724

Used Memory:
       PID          ProcessName start_addr       size
         3           PROCESS-03        200        100
-------------------------------------------------------------
```

紧缩验证：

总内存大小 800

```
1 - Set memory size (default=1024)
2 - Select memory allocation algorithm
3 - New process
4 - Terminate a process
5 - Display memory usage
0 - Exit
1
Total memory size = 800
```

当前两个进程 1 3 分别大小 200 150

空闲分区 200-350 和 500-800

现在创立进程 4 大小 400 但是当前空闲分区内存不够需要进行紧缩。

```
1 - Set memory size (default=1024)
2 - Select memory allocation algorithm
3 - New process
4 - Terminate a process
5 - Display memory usage
0 - Exit
3
Memory for PROCESS-04: 400
```

观察紧缩后的内存使用情况。结果如图，发现进程 1 和 3 进行紧缩留出空闲分区 350-800

用于进程 4 的内存分配。

```
-----------------------------------------------------------
Free Memory:
          start_addr                size
                 750                  50

Used Memory:
       PID          ProcessName start_addr       size
         4           PROCESS-04        350        400
         3           PROCESS-03        200        150
         1           PROCESS-01          0        200
-----------------------------------------------------------
```

2.7 页面置换算法复杂度分析

做的是内存分配和回收

2.8 回答问题

2.8.1 软中断通信

1.

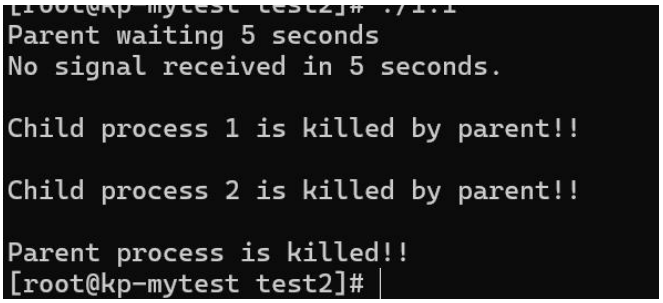
初始我认为出现的结果是出现提醒父进程五秒内等待，然后我们选择按键 delete 或者 quit 键。然后按键后出现父进程接受信号，然后子进程 1 和 2 结束，最后父进程结束。

2.

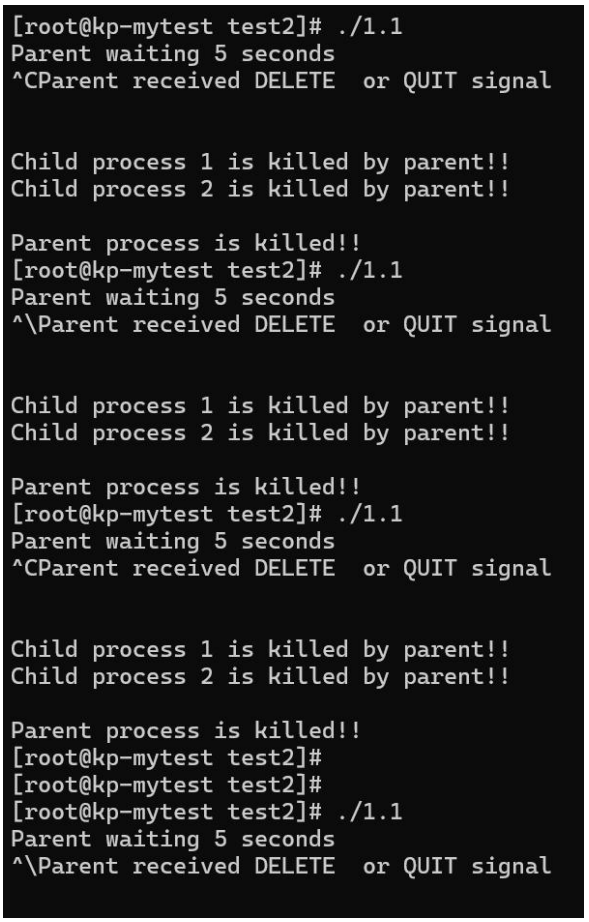
实际结果是子进程 1 和 2 的结束顺序不一致，是先结束子进程再结束父进程。接受 delete 和 quit 信号没有什么区别，结果是一致的，没有什么区别，但是当 5s 内没有输入中断信号

结束是

这个是过 5s 后自动中断的结果。



有信号输入的结果。



这个是分别有 5s 内中断和 5s 后中断的运行结果

```
[root@kp-mytest test2]# ./1.1
Parent waiting 5 seconds
No signal received in 5 seconds.

Child process 1 is killed by parent!!

Child process 2 is killed by parent!!

Parent process is killed!!
[root@kp-mytest test2]# ./1.1
Parent waiting 5 seconds
^CParent received DELETE  or QUIT signal


Child process 1 is killed by parent!!
Child process 2 is killed by parent!!

Parent process is killed!!
```

结果不同的原因是，因为不同的输入信号导致最后的 flag 不同从而输出结果不同。

3.

改成闹钟中断后结果主要是 5s 后中断不一样。

```
  printf("Parent waiting 5 seconds\n");
      alarm(5);
      pause();


void inteer_handlerf(int sig){
      falg=0;
}
  signal(SIGALRM, inter_handlerf) ;
```

修改了这三个部分。保证 5s 后生成闹钟信号 sigalrm,程序结果一样，但是代码修改。

```
[root@kp-mytest test2]# ./1.2
Parent waiting 5 seconds
No signal received in 5 seconds.

Child process1 is killed by parent!!

Child process2 is killed by parent!!

Parent process is killed!!
[root@kp-mytest test2]# ./1.2
Parent waiting 5 seconds
No signal received in 5 seconds.

Child process1 is killed by parent!!

Child process2 is killed by parent!!

Parent process is killed!!
[root@kp-mytest test2]#
```

4.

使用了两次，主要是将信号 16 17 传输给子进程 1 和子进程 2。

由于子进程

signal(16, child1_handler);

signal(17, child2_handler);

会自动调用函数进行处理退出。

5.

通过 exit(status)指令

一般是主动退出更好保证资源释放和数据安全。

异常情况最好通过外部 kill，比如死锁。

2.8.2 管道通信

1.我最初认为结果是有锁的时候字符串不是混乱的，无锁的时候输出是混乱的

2.实际第一个实验输出在没有锁的时候也是正常的，由于可能是输入字符过少，时间片分配的时间足够将所有的字符一起输入缓冲区。

3.通过 lockf 保证互斥们也就是在其中一个进程写入的时候另一个进程被阻塞在外。

```
lockf(fd[1],1,0)
lockf(fd[1],0,0)
```

锁住通道 fd[1]保证写入时唯一。

同步主要利用通道的写入读出，当子进程向通道写入大量数据完导致管道阻塞后进入休眠，直到父进程读完，子进程开始唤醒，但是我们的数据只有 4000 不足与导致缓冲区 4005 满，我们选择新建两个管道，保证只有子进程一和进程二写完 父进程再运行接下来的代码。

```
write(fu[1], "2", 1);  //子进程发送 写完数据信号
read(fu[0],&buf,1);   //父进程接受  子进程写完数据信号
```

并且保证父进程在没有输出数据，子进程不会结束。使用

```
// 通知两个子进程退出
      write(zi[1], "1", 1);
      write(zi[1], "1", 1);
// 等待父进程信号
      read(zi[0], &buf, 1);
```

如果不互斥或则同步将会导致出现读方在读出数据是乱码。

2.8.3 内存的分配与回收

1.

FF 主要按照空闲分区的链表按顺序遍历，找到适合大小的空闲分区就退出。

BF 主要排序空闲分区链表从小到大，找到第一个大于所需内存大小的空闲分区。

WF 主要排序空闲分区链表从大到小，找到第一个满足要求的空闲分区

三种算法的优劣：

FF:速度快，不需要排序，但是会产生外部碎片。

BF:减少每次分配造成的内存浪费，但是搜索成本高，需要排序一遍链表

WF:分配后的休闲块依然很大可以被继续使用，减少碎片聚集，但是浪费大块内存(可能导致大内存的分配失败)，且搜索成本高，需要排序。

通过对链表排序进行提高速度，BF 和 WF 通过排序保证按顺序后第一个满足要求的空闲分区就是需要的。

2.

FF：通过起始地址排序，地址小在前头

```
int compare_free_by_addr(const void *a, const void *b){
      struct free_block_type *pa = *(struct free_block_type **)a;
      struct free_block_type *pb = *(struct free_block_type **)b;
      if(pa->start_addr < pb->start_addr) return -1;
      else if(pa->start_addr > pb->start_addr) return 1;
      return 0;
}
```

BF：通过空闲分区大小排序，从小到大。

```
int compare_free_by_size_asc(const void *a, const void *b){
      struct free_block_type *pa = *(struct free_block_type **)a;
      struct free_block_type *pb = *(struct free_block_type **)b;
      if(pa->size < pb->size) return -1;
      else if(pa->size > pb->size) return 1;
      return 0;
}
```

WF：通过空闲分区大小排序，从大到小

```
int compare_free_by_size_desc(const void *a, const void *b){
      struct free_block_type *pa = *(struct free_block_type **)a;
      struct free_block_type *pb = *(struct free_block_type **)b;
      if(pa->size > pb->size) return -1;
      else if(pa->size < pb->size) return 1;
      return 0;
}
```

3.

内部碎片：指的是系统按块分配内存，假如 18k 的内存需要分配，系统内存块按 20k 分配，导致有 2k 内存无法利用，这就是内碎片。

外部碎片：指的是空闲分区由于分配内存不连续导致空闲分区浪费，比如 100k 的内存

在 0-46 和 52-75 的内存上分配出去，但是假如有 28k 的进程需要分配但是内存不连续可是加起来是够的 6+25=31>28。

紧缩解决是外部碎片。将已经分配的块往一端移动，腾出连续的空间。

因为内部碎片是块分配的，这是固定分配单位无法利用

4.首先将该分配块换成空闲块的结构体指针

```
fbt = (struct free_block_type*) malloc(sizeof(struct free_block_type));
    if(!fbt) return -1;


    fbt->start_addr = ab->start_addr;
    fbt->size = ab->size;
    fbt->next = NULL;
```

再通过将 fpt 指针插入队尾后，按地址算法排序后

```
rearrange_FF(); // 先按地址排序
```

创建新的指针指向表头，开始遍历，如果发现当前指针起始地址加长度等于该指针的下个指针的起始地址就可以将两个节点合并，合并空闲块。

```
while(p && p->next){
        if(p->start_addr + p->size == p->next->start_addr){
            // 合并 p 和 p->next
            struct free_block_type *to_remove = p->next;
            p->size += to_remove->size;
            p->next = to_remove->next;
            free(to_remove);
        } else {
            p = p->next;
        }
```

```
}
```

2.9 实验总结

2.9.1 实验中的问题与解决过程

在管道通信传输过程中，我注意第一个实验在无锁的情况下，依然是输出有序，与我们估计不符合，原因是可能输入数据过小，即使是全部数据传输，时间片也够传输完，我们通过增加一个 usleep(10)增长每个数据传输的时间。体现了无锁互斥的状态。

2.9.2 实验收获

成功实现软中断通信，主要是使用 16 17 用户自定义中断实现父子进程之间的通信，保证子进程收到父进程的信号，开始中断。使用 14 号时钟信号时保证 5s 后自动调用处理函数。子进程 1 2 的退出顺序是不确定的。通过管道实现父子进程对数据的传输，对同一个管道两个子进程输入大量不同数据，最后实现有序输出。在无锁时如果有两个进程向通同一个管道传输数据可能导致数据的无序和不完整。实现内存管理和分配的模拟系统，我们实现 FF，BF，WF 三种动态分区分配算法。实现了内存的回收，以及相邻空闲区的合并，并且通过紧缩减少外部碎片。我们模拟了三种情况分别对应不同的算法。

2.9.3 意见与建议

希望更新实验指导书以及查缺补漏。

2.10 附件

2.10.1 附件 1　程序


1-1.c


1-2.c


2-1.c


2-2.c


2-3.c


2-4.c

2-5.c

2.10.2 附件 2 Readme

实验二readme.docx

3 Linux 动态模块与设备驱动

3.1 实验目的

动态模块

1.理解 Linux 内核是 **monolithic** 架构，以及为什么需要 动态可加载内核模块（LKM）来避免反复重新配置/编译/重启内核。

2.掌握内核模块的基本组成与规范：

初始化/清理入口（module_init / module_exit 或旧版 init_module / cleanup_module）

模块信息（MODULE_AUTHOR / MODULE_DESCRIPTION / MODULE_ALIAS / MODULE_LICENSE）

3.掌握内核模块的基本操作与调试手段：

编译模块（Makefile + make）

加载/卸载（insmod / rmmod）

查看模块状态（lsmod、/proc/modules）

查看内核日志（dmesg + printk）

设备驱动

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 实验内容

动态模块

动态模块基础实验（Hello 模块）

编写一个最小内核模块，加载时 printk 输出信息，卸载时 printk 输出信息。

使用 Makefile（基于内核 build 系统）编译成 .ko。

使用 insmod 加载、rmmod 卸载，并用 dmesg 验证输出。

在模块加载时修改 sys_call_table 中某个 syscall 指向新函数 hello(a,b)。

在模块卸载时恢复原 syscall 指针。

编写用户态测试程序，在修改前后分别调用 syscall 验证行为变化。

设备驱动

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。

3.3 实验思想

动态模块

LKM（可加载模块）机制

模块在系统运行过程中动态装入内核空间，一旦装入就和内核其他代码一样运行在内核态、拥有同等权限。

通过模块机制可以让内核保持精简，同时按需加载驱动（网络、文件系统、外设驱动等）。

模块生命周期

V2.6 及之后：module_init() 注册初始化函数、module_exit() 注册清理函数。

初始化阶段完成资源申请/注册；清理阶段释放资源/注销注册项，确保可卸载。

编译原理（为什么要用内核源码/头文件）

模块编译依赖内核头文件（include/）与内核编译生成的符号信息；一般通过 /lib/modules/$(uname -r)/build 指向当前内核的构建目录来完成匹配编译。

Makefile 的 $(MAKE) -C $(KERNELDIR) M=$(PWD) modules 是标准外部模块编译方式。

调试原理

printk 输出到内核日志缓冲区，通过 dmesg 查看。

lsmod 和 /proc/modules 反映模块装载状态。

设备驱动

字符设备

a) 键盘——键盘驱动程序

b) 串口——串口驱动程序

c) 并口——并口驱动程序

d) 显卡——显卡驱动程序

块设备

e) 磁盘——磁盘驱动程序

f) 软盘——软盘驱动程序

g) 光盘——光盘驱动程序

h) 优盘——优盘驱动程序

驱动程序与操作系统内核的接口通过数据结构 file_operations 来完成

驱动程序与系统引导的接口

利用驱动程序对设备进行初始化

驱动程序与设备的接口

描述驱动程序如何与设备进行交互

设备文件的 VFS 处理:

普通文件——文件系统将用户的操作转换成对磁盘分区的数据块操作

设备文件——文件系统将用户的操作转换成对设备的驱动操作

VFS 中，每个文件都有一个 inode 与其对应，内核的 inode 结构中的 i_fop

成员，类型为 file_operations,file_operations 定义了文件的各种操作.

用户对文件操作是通过调用 file_operations 来实现的，或者说内核使用

file_operations 来访问设备驱动程序中的函数,为了使用户对设备文件的操作

能够转换成对设备的操作，VFS 必须在设备文件打开时，改变其 inode 结构中

i_fop 成员的默认值，将其替换为与设备相关的具体函数操作。用户访问设备时，

文件系统读取设备文件在磁盘上相应的 inode，并存入主存 inode 结构中,内核

将文件的主设备号与次设备号写入 inode 结构中的 i_rdev 字段，并将 i_fop 字

段设置成 def_chr_fops（或 def_blk_fops），以便用户对设备文件的操作转换

成对设备的驱动操作。

3.4 实验步骤

(1)根据指导书完成代码，实现系统调用的动态模块。分别运行系统调用前后的用户程序，观察不同的结果。

(2)编写一个内核空间模拟字符 linux 设备的驱动模块，通过用户程序调用上面的字符设备驱动程序实现用户聊天。

3.5 程序运行初值及运行结果分析

第一个实验中我们发现找不到 适合的 linux 版本所以选择使用 x86 架构下的修改系统调用

但是首先需要修改写保护由于 CR0 寄存器的 16 位是写保护，我们需要将该位设置为 0。

执行我们的修改，修改完需要正常返回。

```
Makefile  modify_syscall.c  new.c  old.c
[root@kp-test2 test]# make
make -C /lib/modules/4.19.90-2003.4.0.0036.oe1.x86_64/build M=/root/test modules
make[1]: Entering directory '/usr/src/kernels/4.19.90-2003.4.0.0036.oe1.x86_64'
  CC [M]  /root/test/modify_syscall.o
/root/test/modify_syscall.c: In function ‘modify_syscall’:
/root/test/modify_syscall.c:48:22: warning: assignment makes integer from pointer without a cast [-Wint-conversion]
     p_sys_call_table = (unsigned long *)kallsyms_lookup_name("sys_call_table");
                      ^
  Building modules, stage 2.
  MODPOST 1 modules
  CC      /root/test/modify_syscall.mod.o
  LD [M]  /root/test/modify_syscall.ko
make[1]: Leaving directory '/usr/src/kernels/4.19.90-2003.4.0.0036.oe1.x86_64'
gcc old.c -o old
gcc new.c -o new
[root@kp-test2 test]# ./new
-1
[root@kp-test2 test]# ./old
tv_sec:1764591248
tv_usec:533603
[root@kp-test2 test]#
```

这个是正常结果

经过调用后

```
tv_usec:533603
[root@kp-test2 test]# sudo insmod modify_syscall.ko
[root@kp-test2 test]# ./new
30
[root@kp-test2 test]# ./old
tv_sec:2064412784
tv_usec:0
[root@kp-test2 test]#
```

结果正确，正确调用了 hello 函数，以及使用原来的函数出现问题。

完成设备驱动模块的实验中我们做出了一个聊天室，完成了可以进行聊天，以及修改自己名字显示当前聊天人数。

首先运行 chat 文件首先输入自己的初始名字。

```
[root@kp-mytest test3]# ./chat
================================================
            Welcome to Your Chat Room!
================================================
Enter your name: mike
No chat history available.
------------------------------------------------
Hello, mike!
Type '/quit'  to exit.
Type '/users' to check number of users.
Type '/name'  to change your name.
------------------------------------------------
Chat client is ready. Start typing your messages...
```

然后我们假如有多个人聊天，后来进入的人可以查看过去的聊天记录

```
Chat client is ready. Start typing your messages...

hi mike
[recv] [2025-12-03 14:40:39] will: hi mike
[recv] [2025-12-03 14:40:47] mike: hi will
```

过去的聊天记录

```
[root@kp-mytest ~]# cd /usr/local/src/test3
[root@kp-mytest test3]# ./chat
================================================
            Welcome to Your Chat Room!
================================================
Enter your name: nancy
------------------------------------------------
Chat History:
[2025-12-03 14:40:39] will: hi mike
[2025-12-03 14:40:47] mike: hi will
------------------------------------------------
------------------------------------------------
Hello, nancy!
Type '/quit'  to exit.
Type '/users' to check number of users.
Type '/name'  to change your name.
------------------------------------------------
Chat client is ready. Start typing your messages...
```

新增的 nancy 查看过去聊天

```
/name alice
Your name has been changed to alice.
hi everyone
[recv] [2025-12-03 14:42:55] alice: hi everyone
```

并且我们可以通过/name 修改自己的名字。

并且在后来访问的时候聊天记录里的话是来自修改后名字的。

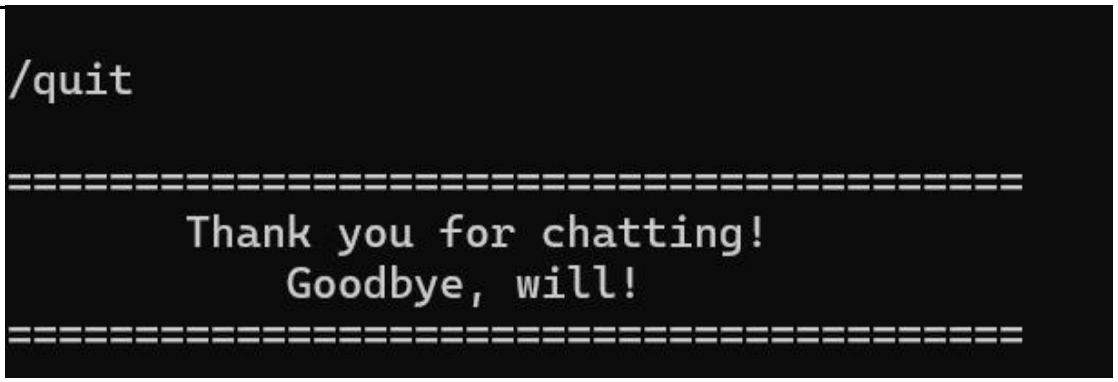
```
Enter your name: bob
------------------------------------------------
Chat History:
[2025-12-03 14:40:39] will: hi mike
[2025-12-03 14:40:47] mike: hi will
[2025-12-03 14:42:36] nancy: hi
[2025-12-03 14:42:55] alice: hi everyone
------------------------------------------------
------------------------------------------------
Hello, bob!
```

并且我们的可以查看当前聊天室人数

```
================================================
Enter your name: bob
------------------------------------------------
Chat History:
[2025-12-03 14:40:39] will: hi mike
[2025-12-03 14:40:47] mike: hi will
[2025-12-03 14:42:36] nancy: hi
[2025-12-03 14:42:55] alice: hi everyone
------------------------------------------------
------------------------------------------------
Hello, bob!
Type '/quit'  to exit.
Type '/users' to check number of users.
Type '/name'  to change your name.
------------------------------------------------
Chat client is ready. Start typing your messages...

/users
Current users=3
```

通过/quit 正常退出。



3.6 实验总结

3.6.1 实验中的问题与解决过程

1.系统调用实验的内核是 linux2.6，所以选择重新新建一个云服务器完成实验。以及我们需要查找新的系统调用表和对应的系统调用号，解除内存写保护后再修改。

1）它什么时候"可能可行"

一般需要同时满足这些条件（越老越容易满足）：

- **内核版本较老**（典型 2.4/早期 2.6 的年代），并且没有/较少启用内核自保护措施。
- **32 位 x86** 环境：你用的是 `sys_No*4` 这种按 4 字节指针表项计算的方式，隐含了 32 位指针大小。
- 你能拿到正确的 `sys_call_table` 地址，并且该地址在运行时**没有被随机化改变**（老内核没有 KASLR 或没启用）。
- `sys_call_table` 所在内存页在运行时**允许写**（老内核时期更可能；现代内核通常是只读/受保护）。

在这种"老内核 + 32 位 + 无强防护"的组合下，这类实验常被用来演示"模块能改内核行为"。

2.用户程序退出时出现双重 free()，原因是在程序判断输入/quit 时和最后都使用了/quit 从而导致出现双重 free()，我们通过增加新的 flag(cleaned)保证使用一次 cleanup()后无法再次 free()第二次。

```c
void cleanup(int sig) {
    if (cleaned) return; // 防止重复执行
    cleaned = 1;

    if (fd >= 0) {
        close(fd);
        fd = -1;
    }
    // history_file 在本程序中不长期保持打开（我们在使用后总会 fclose）
    if (history_file) {
        fclose(history_file);
        history_file = NULL;
    }

    printf("\n");
    print_line('=', 50);
    printf("            Thank you for chatting!\n");
    printf("                Goodbye, %s!\n", username);
    print_line('=', 50);
    printf("\n");

    exit(0);
}
```

3.6.2 实验收获

理解了 Linux 内核与动态模块的关系

通过学习知道 Linux 属于 monolithic 内核，内核功能部件共享内部数据结构与例程。传统方式把新功能编进内核需要重新配置、重新编译并重启，效率低；而动态模块（LKM）允许在系统运行中按需加载/卸载内核功能，使系统更灵活。

掌握了内核模块的基本生命周期与代码结构，明确了模块的“加载初始化”和“卸载清理”两阶段，并能理解 printk 输出在内核日志中查看的调试方式。

熟悉了模块编译与构建环境要求，实验中认识到编译内核模块必须具备：GCC 工具链、与当前运行内核匹配的内核源码/头文件（include 目录）、以及内核编译生成的符号信息（如 Module.symvers 等）。同时掌握通过 Makefile 调用内核构建系统（/lib/modules/$(uname -r)/build）来生成 .ko 模块的方法。

掌握了模块的管理与验证手段，能使用 insmod 加载模块、rmmod 卸载模块；能用 lsmod 或 /proc/modules 查看模块状态；并能通过 dmesg 检查 printk 输出，从而验证模块是否正确执行初始化与清理逻辑。

通过本实验理解了字符设备驱动在内核中的基本组成：设备注册/注销、设备文件节点的访问入口、以及 open/read/write/release 等关键操作函数在驱动中的位置与作用。认识到字符设备面向“字节流”，与块设备面向“数据块”的访问方式存在本质差异。

3.6.3 意见与建议

强烈建议更新指导书中相关条件，比如使用的操作系统版本过老

3.7 附件

3.7.1 附件 1　程序


modify_syscall.c


new.c


old.c


chat.c


dir.c

3.7.2 附件 2　Readme


实验三readme.docx