西安交通大学

操作系统专题实验报告

班级：计算机 2201

学号：牛虹瑾

姓名：2226124051

2024 年 12 月 13 日

目录

1 进程、线程相关编程实验

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。完成下图程序的运行验证， 多运行几次程序观察结果；去除 wait()函数后再观察结果并进行理论分析。

```
#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;
}
```

2) 扩展上图的程序：

   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 实验步骤与运行结果

在程序中输出父子进程的 pid，分析两者的关系，进一步加入 wait()函数、调用 system 函数、exec 族函数，并观察实验结果、分析相关作用。

步骤一：编写并多次运行上图代码：

```
[root@kp-test01 ex1]# gcc 1-11.c -o 1-11
[root@kp-test01 ex1]# ./1-11
child: pid = 0  child: pid1 = 3726  parent: pid = 3726  parent: pid1 = 3725  [root@kp-test01 ex1]# ./1-11
child: pid = 0  child: pid1 = 3771  parent: pid = 3771  parent: pid1 = 3770  [root@kp-test01 ex1]# ./1-11
child: pid = 0  child: pid1 = 3787  parent: pid = 3787  parent: pid1 = 3786  [root@kp-test01 ex1]#
```

步骤二：删去图 1-1 代码中的 wait()函数并多次运行程序。

```
[root@kp-test01 ex1]# gcc 1-12.c -o 1-12
[root@kp-test01 ex1]# ./1-12
parent: pid = 3881  parent: pid1 = 3880  child: pid = 0  child: pid1 = 3881  [root@kp-test01 ex1]# ./1-12
parent: pid = 3905  parent: pid1 = 3904  child: pid = 0  child: pid1 = 3905  [root@kp-test01 ex1]# ./1-12
parent: pid = 3909  parent: pid1 = 3908  child: pid = 0  child: pid1 = 3909  [root@kp-test01 ex1]# ./1-12
parent: pid = 3932  parent: pid1 = 3931  child: pid = 0  child: pid1 = 3932  [root@kp-test01 ex1]#
```

步骤三：修改图 1-1 中代码， 增加全局变量 value，在父进程中修改 value 值为 3，子进程中修改 value 值为 1，分别打印`value`的地址与值。

```
[root@kp-test01 ex1]# gcc 1-13.c -o 1-13
[root@kp-test01 ex1]# ./1-13
child: global value = 1  child: *value = 0x420064  parent: global value = 3  parent: *value = 0x420064  [root@kp-test01 ex1]# ./1-13
child: global value = 1  child: *value = 0x420064  parent: global value = 3  parent: *value = 0x420064  [root@kp-test01 ex1]# ./1-13
child: global value = 1  child: *value = 0x420064  parent: global value = 3  parent: *value = 0x420064  [root@kp-test01 ex1]#
```

步骤四：在步骤三基础上，在 return 前对全局变量 value 加 1 并输出结果。

```
[root@kp-test01 ex1]# gcc 1-14.c -o 1-14
[root@kp-test01 ex1]# ./1-14
parent: value = 3  parent: *value = 0x420064
child: value = 1  child: *value = 0x420064
before return: value = 2  before return: *value = 0x420064
before return: value = 4  before return: *value = 0x420064
[root@kp-test01 ex1]# ./1-14
child: value = 1  child: *value = 0x420064
before return: value = 2  before return: *value = 0x420064
parent: value = 3  parent: *value = 0x420064
before return: value = 4  before return: *value = 0x420064
[root@kp-test01 ex1]# ./1-14
parent: value = 3  parent: *value = 0x420064
child: value = 1  child: *value = 0x420064
before return: value = 2  before return: *value = 0x420064
before return: value = 4  before return: *value = 0x420064
```

步骤五：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。

调用 system()函数：

```
[root@kp-test01 ex1]# gcc 1-151.c -o 1-151 -lpthread
[root@kp-test01 ex1]# ./1-151
parent PID = 3310
child PID = 3311
system_call: PID = 3312
child PID = 3311
[root@kp-test01 ex1]# ./1-151
parent PID = 3334
child PID = 3335
system_call: PID = 3336
child PID = 3335
[root@kp-test01 ex1]# ./1-151
parent PID = 3345
child PID = 3346
system_call: PID = 3347
child PID = 3346
```

调用 exec 族函数：

```
[root@kp-test01 ex1]# gcc 1-152.c -o 1-152 -lpthread
[root@kp-test01 ex1]# ./1-152
parent PID = 3450
child PID = 3451
system_call: PID = 3451
[root@kp-test01 ex1]# ./1-152
parent PID = 3475
child PID = 3476
system_call: PID = 3476
[root@kp-test01 ex1]# ./1-152
parent PID = 3485
child PID = 3486
system_call: PID = 3486
```

1.1.4 结果分析

1) **步骤一：**调用 fork()创建一个子进程。父进程中，pid 返回子进程的进程号，pid1 为父进程的进程号；子进程中，pid 返回 0，pid1 返回子进程本身的进程号。

   调用了 wait()，父进程会等子进程结束后再结束。wait()运行时父进程已经打印了输出结果，所以其实父子进程间的打印顺序应该是随机的，父进程先于子进程打印的输出结果出现得更频繁，猜测可能是因为子进程代码少，所以执行速度更快。

2) **步骤二：**wait()会暂时停止目前进程的执行，直到有信号来到或子进程结束。删去 wait()函数后，父进程不会等待子进程结束，它将会和子进程同

时运行。因此，父进程和子进程的打印顺序是不确定的。而且如果父进程先于子进程结束，子进程会变成孤儿进程，此时子进程将立刻结束。

3) **步骤三：**增加全局变量 value，在父进程中修改 value 值为 3，子进程中修改 value 值为 1，分别打印 value 的地址与值。

   观察到，父子进程打印的值不同，但地址是相同的。因为 fork()后子进程和父进程的数据空间是独立的，不同进程中 value 的修改互不影响。但在父进程和子进程中，指针 p 所指的地址是一样的，因为 fork()会复制整个地址空间。

4) **步骤四：**在 1-13 的基础上，在 return 前对全局变量`value`加 1。父子进程都会执行这个操作，而且由于`fork()`后子进程和父进程的数据空间是独立的，所以不同的加 1 操作互不影响。子进程的`value`从 1 增加到 2，父进程的 `value`从 3 增加到 4。

   因为调用了`wait()`函数，父进程将等待子进程结束后再继续运行，因此`before return: value = 4`语句将最后输出。父进程和子进程间执行加 1 操作的顺序是随机的，但进程内部需要先进行赋值，再进行加 1 操作。因此，进程内部的打印顺序是确定的。

5) **步骤五：**在子进程内部分别使用 system()和 execvp()函数调用 system_call.c 输出进程号，与直接调用 getpid()获得的进程号对比。观察到 execvp()输出的进程号是子进程的 pid，而 system()输出的进程号并非子进程的 pid。同时，调用 execvp()函数后，进程不会继续执行接下来的输出。

   这是因为`system()`会新创建一个子进程来执行指定的命令，其 pid 与原先的进程不同；而 execvp()是一个进程替换函数，它会替换当前进程的映像，并用新程序的映像来覆盖现有的进程，所以 execvp()输出的进程号与原先的进程相同。

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 实验步骤与运行结果

步骤一：创建两个子线程，定义共享变量初始值为 0，两个线程分别对其进行 100000 次+/- 100 操作，最终在主进程中输出处理后的变量值，观察输出结果。

```
[root@kp-test01 ex1]# gcc 1-21.c -o 1-21 -lpthread
[root@kp-test01 ex1]# ./1-21
thread1 create success!
thread2 create success!
variable result: 158000
[root@kp-test01 ex1]# ./1-21
thread1 create success!
thread2 create success!
variable result: -157900
[root@kp-test01 ex1]# ./1-21
thread1 create success!
thread2 create success!
variable result: 155000
```

步骤二：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。

```
[root@kp-test01 ex1]# gcc 1-22.c -o 1-22 -lpthread
[root@kp-test01 ex1]# ./1-22
thread1 create success!
thread2 create success!
variable result: 0
[root@kp-test01 ex1]# ./1-22
thread1 create success!
thread2 create success!
variable result: 0
[root@kp-test01 ex1]# ./1-22
thread1 create success!
thread2 create success!
variable result: 0
```

步骤三：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。

调用 system()函数：

```
[root@kp-test01 ex1]# ./1-231
thread1 create success!
thread1 tid = 3181, pid = 3180
thread2 create success!
thread2 tid = 3182, pid = 3180
system_call: PID = 3183
thread2 systemcall return
system_call: PID = 3184
thread1 systemcall return
[root@kp-test01 ex1]# ./1-231
thread1 create success!
thread2 create success!
thread2 tid = 3198, pid = 3196
thread1 tid = 3197, pid = 3196
system_call: PID = 3200
system_call: PID = 3199
thread1 systemcall return
thread2 systemcall return
```

调用 exec 族函数：

```
[root@kp-test01 ex1]# gcc 1-232.c -o 1-232 -lpthread
[root@kp-test01 ex1]# ./1-232
thread1 create success!
thread2 create success!
thread1 tid = 3771, pid = 3770
thread2 tid = 3772, pid = 3770
system_call: PID = 3770
[root@kp-test01 ex1]# ./1-232
thread1 create success!
thread1 tid = 3794, pid = 3793
thread2 create success!
thread2 tid = 3795, pid = 3793
system_call: PID = 3793
[root@kp-test01 ex1]# ./1-232
thread1 create success!
thread1 tid = 3823, pid = 3822
thread2 create success!
system_call: PID = 3822
```

1.2.4 结果分析

1) **步骤一：**创建两个子线程，在不设置线程锁和信号量的情况下，分别对全局变量 shared_var 进行 100000 次+100 和-100 操作。在没有竞争的理想情况下，预计输出结果为 0，但实际输出非 0，这是因为两个线程间可能并发访问相同的内存，比如全局变量 shared_var，从而造成数据竞争。

2) **步骤二：**使用信号量来保证两个线程不会在没有协调的情况下同时访问和修改`shared_var`。对于线程 increment，每次循环开始，首先执行 P 操作，如果 signal 值为 1，则 P 操作成功，线程进入临界区，并将信号量值减至 0。

   临界区操作完成后，V 操作将信号量的值加回 1，释放临界区。线程 decrement 与 increment 线程类似。

   信号量保证了每次只有一个线程可以访问和修改全局变量 shared_var，从而避免了数据竞争。

3) **步骤三：**两个线程的 TID 不同，因为每个线程都有自己的 TID；PID 相同，因为两个线程属于同一个进程。调用 system()函数输出进程的 PID，发现 PID 不同，因为 system()会新创建一个子进程来执行指定的命令。

   调用 execvp()时，观察到输出不完全。因为调用 execvp() 会导致当前线程的代码空间被替换，主进程剩余的代码不会被继续执行。所以，只要有一个线程开始执行 execvp()，新进程将会覆盖原先的进程，另一个线程也不会继续执行原先的代码。

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 实验步骤

步骤一：编写模拟自旋锁程序代码 spinlock.c；

步骤二：编译并运行程序，分析运行结果；

1.3.4 运行结果与分析

自旋锁的设计是为了防止多个线程在同时访问 shared_value 时发生数据竞争。如果没有自旋锁，两个线程可能会同时读取 shared_value，并各自进行+1 或-1 操作，而这时会造成数据冲突，从而导致最终结果错误。

增加自旋锁后，每次只能有一个线程访问共享资源，因此两个线程在操作共享变量时不会发生数据竞争，从而正确地修改 shared_value。

```
[root@kp-test01 ex1]# gcc spinlock.c -o spinlock -lpthread
[root@kp-test01 ex1]# ./spinlock
Shared value: 0
thread1 create success!
thread2 create success!
Shared value: 0
[root@kp-test01 ex1]# ./spinlock
Shared value: 0
thread1 create success!
thread2 create success!
Shared value: 0
[root@kp-test01 ex1]# ./spinlock
Shared value: 0
thread1 create success!
thread2 create success!
Shared value: 0
```

1.4 实验总结

1.4.1 遇到的问题

1 对于 system 函数与 exec 函数的原理不了解，对于输出 PID 存在疑惑。

解决方法：查阅官方文档，了解这两个函数的详细说明和示例。编写程序来实际调用这两个函数，观察它们的执行情况和进程行为。

2 调用 pthread 库时，gcc 编译链接不成功。

解决方法：网上查阅相关博客之后，发现 gcc 编译时需要使用-ipthread 指令来链接 pthread 库。

1.4.2 实验收获

在完成操作系统实验后，我深感收获颇丰。通过本次实验，我熟悉了 Linux 操作系统的基本环境和操作方法。通过运行系统命令，如 ls、cat、ps 等，我能够查看系统基本信息，了解系统的运行状态。这为我后续进行更深入的操作系统学习奠定了基础。

编写并运行简单的进程调度相关程序，让我对进程调度、进程间变量的管理有了更直观的认识。在实际操作中，我体会到了进程调度算法的重要性，以及如何通过合理安排进程的执行顺序来提高系统性能。

在探究多线程编程中的线程共享进程信息方面，我深刻理解了多线程并发执行的原理。通过创建多个线程并使其共享进程信息，我发现线程间的资源共享确实存在一定的问题，如竞态条件和数据不一致等。

关于自旋锁的实验，我实了解了自旋锁的基本概念，明白了自旋锁的工作原理和特点。相较于其他锁机制，自旋锁在特定场景下具有优势，但也

存在局限性。在多线程实验环境中，我设计了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。通过这个过程，我更加明确了线程同步的重要性。c. 使用自旋锁实现了线程间的同步，确保了同一时间只有一个线程能够访问竞争资源，有效避免了数据不一致和竞态条件的发生。

总之，本次实验让我对操作系统中的进程调度、线程共享资源以及线程同步等方面有了更加深入的了解。同时，我也意识到操作系统领域还有很多知识等待我去探索，这将激励我不断前行，追求更高的技术水平。

2 进程管理与内存通信

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 实验步骤与运行结果

使用 pause()阻塞子进程，直到父进程发来信号。pause()是一个阻塞函

数，用于让进程挂起，直到捕获一个信号，并且存在该信号的处理程序。

步骤一：5s 内按下 ctrl+c 发送 SIGINT 信号：

```
[root@kp-test01 ex2]# ./2.1
^C
2 stop test
17 stop test
16 stop test


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

Parent process is killed!!
```

步骤二：5s 内按下 ctrl+\发送 SIGQUIT 信号：

```
[root@kp-test01 ex2]# ./2.1
^\
3 stop test
16 stop test

Child process1 is killed by parent!!
17 stop test

Child process2 is killed by parent!!

Parent process is killed!!
```

步骤三：5s 内不发送任何信号，等待闹钟中断：

```
[root@kp-test01 ex2]# ./2.1

14 stop test

16 stop test

Child process1 is killed by parent!!

17 stop test

Child process2 is killed by parent!!

Parent process is killed!!
```

2.1.4 结果分析

1. **你最初认为运行结果会怎么样？写出你猜测的结果。**

   我最初认为子进程将直接终止，不会输出“Child process 1 is killed by parent !! ”语句。

2. **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。

实际运行结果和截图如 2.1.3 所示，父进程将打印中断信号或者超时信号的值，子进程将打印收到的信号的值与终止信息。

接收不同中断前后的差别：SIGINT 或 SINQUIT 信号未触发时，子进程处于 pause，父进程等待中断信号或超时信号。中断信号触发后，父进程向两个子进程分别发送信号 SIGSTKFLT 和 SIGCHLD 子进程从`pause` 中唤醒，执行信号处理程序，完成资源清理并退出。

3. 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？

改为闹钟中断后，超时之后，父进程向子进程发送 SIGALRM 信号。程序运行结果如 2.1.3 所示，父进程收到闹钟信号 (SIGALRM)后，通过 kill(pid, SIGUSR1) 和 kill(pid, SIGUSR2) 通知子进程退出。

修改前，父进程收到的中断信号来自键盘的 SIGINT 或者 SIGQUIT；修改后，父进程收到的闹钟中断是来自操作系统的 SIGALRM。

4. 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？

kill()命令在程序中被使用了两次。

1) 接收到中断信号：父进程在捕获 SIGINT 或 SIGQUIT 信号时，发送 SIGSTKFL 信号给 pid1 对应的子进程 1；发送 SIGCHLD 信号给 pid2 对应的子进程 2。子进程 1 捕获到 SIGSTKFL 信号后，执行 handle_usr1() 函数，打印 16 stop test；子进程 2 捕获到 SIGCHLD 信号后，执行 handle_usr2() 函数，打印 17 stop test。

2) 超时，父进程强制结束子进程：当超时发生（5 秒后，父进程收到 SIGALRM ），发送 SIGSTKFL 信号给 pid1 对应的子进程 1；发送 SIGCHLD 信号给 pid2 对应的子进程 2。

5. 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？

父进程可以通过`kill(pid, SIGTERM)`或`kill(pid, SIGKILL)`信号强制结束子进程。SIGTERM 用于请求进程正常退出，进程可以捕获此信号并执行必要的清理操作（如关闭文件、释放内存等），然后自行终止。SIGKILL 用于强制终止进程，无条件杀死进程，操作系统会立即清理资源，强制释放进程占用的内存，该进程自身无法执行退出时的清理工作。

进程可以调用`exit()`函数主动终止运行，并释放所有占用的资源，并向操作系统返回状态码。

一般来说，主动退出能可清理资源（文件、内存等）并记录日志，更适合大多数情况；但当进程失去响应（如进入死循环、死锁）或需要

强制终止时，在外部杀死进程更适合。

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

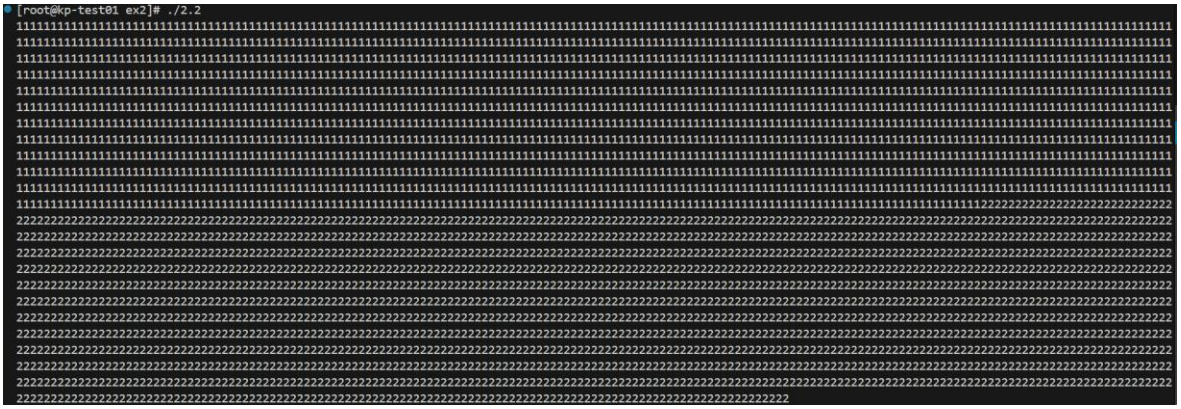
1) 学习 man 命令的用法，通过它查看管道创建、同步互斥系统调用的在线帮助，并阅读 参考资料。

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。

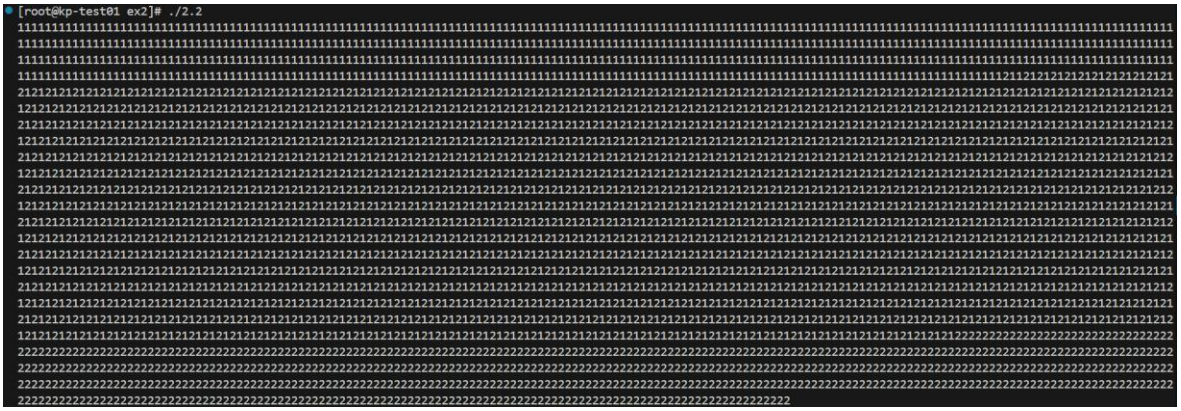
3) 修改上述程序，让其中两个子进程各向管道写入 2000 个字符，父进程从管道中读出， 分有锁和无锁的情况。运行程序，分别观察有锁和无锁情况下的写入读出情况。

2.2.3 运行结果

1. 加锁：



2. 不加锁：



2.2.4 结果分析

1. **你最初认为运行结果会怎么样？**

   实际运行结果和最初预期相同。

2. **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。有锁情况下：先输出 2000 个 1，后输出 2000 个 2。**

   - 有锁情况下，先输出 2000 个 1，后输出 2000 个 2（或者先输出 2000 个 2，后输出 2000 个 1）。

     原因：lockf(fd[1], 1, 0)锁住写端，确保当前子进程独占管道的写操作。在锁释放之前，其他子进程无法对管道写端执行写操作。

     lockf(fd[1], 0, 0)解锁写端，允许其他子进程写入。

     进程 1 启动并加锁后，写入字符 '1'，直到写完 2000 个字符并解锁。子进程 2 等待子进程 1 释放锁后才加锁并写入字符 '2'。锁机制确保了同一时刻只有一个子进程在写入管道。

```
lockf(fd[1], 1, 0); // 锁定管道
        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); // 解除管道的锁定
```

   - 无锁情况下，1 和 2 乱序输出。

     原因：两个子进程不再使用 `lockf()`，写入管道的顺序完全依赖操作系统的调度机制。子进程可以在任何时间点进行写入，导致两者之间可能发生竞争。可能出现两个子进程轮流执行，写入字符交替出现。

3. **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？**

   - **同步：**

如果父进程早于子进程读取管道：管道中的数据可能尚未完全写入，导致父进程读取到的数据不完整。父进程使用 wait(0) 等待子进程结束，从而确保父进程在所有子进程完成写入后再进行管道的读取操作。

```
    wait(0); // 等待子进程 1 结束
    wait(0); // 等待子进程 2 结束
    read(fd[0], InPipe, 4000); // 从管道中读出 4000 个字符
    InPipe[4000] = '\0'; // 加字符串结束符
    printf("%s\n",InPipe); // 显示读出的数据
```

- **互斥：**

  如果不控制互斥，子进程之间可能发生竞争，导致写入数据混乱（比如，write() 的内容被覆盖或交错）。

  使用 lockf(fd[1], 1, 0) 锁定管道的写端，确保某个子进程在写入时独占管道的写权限。lockf(fd[1], 0, 0) 解锁，释放写权限，允许其他子进程进行写入。

```
lockf(fd[1], 1, 0); // 锁定管道
 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); // 解除管道的锁定
```

```
   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);
```

2.3 内存的分配与回收

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

1) 常见的动态分区分配算法

   a) 首次适应算法

   b) 循环首次适应算法

   c) 最佳适应算法

   d) 最差匹配算法

2) 内存碎片

   a) 外碎片

   b) 内碎片

3) 内存紧缩技术

2.3.4 实验步骤与运行结果

补充完整实验指导书所给框架，运行结果如下：

1. 首次适应算法（FF）：

a) 顺序插入大小为 200、300、400 的进程。

```
-------------------------------------------------------------
Free Memory:
          start_addr                 size
                 900                  124

Used Memory:
       PID          ProcessName start_addr       size
         3           PROCESS-03        500        400
         2           PROCESS-02        200        300
         1           PROCESS-01          0        200
-------------------------------------------------------------
```

b) 删除 1 号进程，插入大小为 150 的新进程，此时新进程的内存起始位置为 0。

```
------------------------------------------------------------
Free Memory:
          start_addr                  size
                 150                    50
                 900                   124

Used Memory:
       PID          ProcessName start_addr       size
         4           PROCESS-04          0        150
         3           PROCESS-03        500        400
         2           PROCESS-02        200        300
------------------------------------------------------------
```

c) 再分别插入大小为 200、180 的进程，此时超出默认内存大小，分配失败。

```
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-05:200
Allocation fail

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-06:180
Allocation fail
```

d) 插入大小为 170 的新进程，此时采用内存紧缩技术，将全部空闲块

合并在一起。

```
-------------------------------------------------------
Free Memory:
          start_addr                  size

Used Memory:
       PID          ProcessName start_addr       size
         7           PROCESS-07          0        170
         4           PROCESS-04        174        150
         3           PROCESS-03        324        400
         2           PROCESS-02        724        300
-------------------------------------------------------
```

2. 最佳适应算法（BF）：

a) 顺序插入大小为 200、300、400 的进程。删除第一个进程，此时有两个大小分别为 200、124 的空闲块。插入大小为 100 的新进程，此时选择第二个空闲块，新进程的内存起始位置为 900。

```
-------------------------------------------------------
Free Memory:
          start_addr                  size
               1000                    24
                  0                   200

Used Memory:
       PID          ProcessName start_addr       size
         4           PROCESS-04        900        100
         3           PROCESS-03        500        400
         2           PROCESS-02        200        300
-------------------------------------------------------
```

b) 插入大小为 20 的进程，起始位置为 1000。

```
-------------------------------------------------------------
Free Memory:
          start_addr                  size
                   0                   200

Used Memory:
       PID           ProcessName start_addr        size
         5            PROCESS-05       1000          20
         4            PROCESS-04        900         100
         3            PROCESS-03        500         400
         2            PROCESS-02        200         300
-------------------------------------------------------------
```

3. 最差适应算法（WF）：

a) 顺序插入大小为 100、200、300 的进程。删除 1 号进程，此时有两个大小分别为 200、424 的空闲块。插入大小为 150 的进程，此时选择后一个个空闲块，新进程的内存起始位置为 600。

```
-------------------------------------------------------------
Free Memory:
          start_addr                  size
                 750                   274
                 100                   200

Used Memory:
       PID           ProcessName start_addr        size
         4            PROCESS-04        600         150
         3            PROCESS-03        300         300
         1            PROCESS-01          0         100
-------------------------------------------------------------
```

b) 插入大小为 100 的进程，起始位置为 750。

```
------------------------------------------------------
Free Memory:
          start_addr                 size
                 850                  174
                 100                  200

Used Memory:
       PID           ProcessName start_addr       size
         5            PROCESS-05        750        100
         4            PROCESS-04        600        150
         3            PROCESS-03        300        300
         1            PROCESS-01          0        100
------------------------------------------------------
```

2.3.5 结果分析

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。**

   1) **FF**：优先从空闲块链表的开头查找第一个可以满足请求的空闲块。若找到适合的块，则分配请求大小的内存，并更新该空闲块的剩余大小。

   - 优点：

     查找速度较快：由于从头开始查找，找到第一个适配块即可返回，无需遍历完整链表。

     实现简单：只需维护一个单向链表，直接按顺序搜索即可。

   - 缺点：

     容易产生外部碎片：由于每次分配都选择靠前的小块，内存会逐渐形成许多零散的小碎片。

     随着时间推移，前半部分的内存会被频繁访问，可能导致性能下降。

   - 如何提高性能：

     记录上次分配位置（Next Fit）：在链表中记录上一次分配的位置，从该位置继续查找，避免频繁从头搜索。

     空闲块合并：定期检查并合并相邻的空闲块，减少碎片。

   2) **BF**：从空闲块链表中查找最接近请求大小的空闲块进行分配，即空闲块大小与请求大小的差值最小。

- 优点：

  减少碎片：通过选择最适合的块进行分配，尽可能减少未使用空间。

  更高效利用内存：比 FF 更适合在内存紧张的情况下使用。

- 缺点：

  查找速度慢：必须遍历整个链表以找到最优块，增加查找开销。

  容易形成小碎片：由于优先选择最合适的块，剩余的小块可能不足以满足未来请求。

- 如何提高性能：

  维护有序链表：按块大小升序排列空闲块链表，避免每次分配都遍历整个链表。

  索引结构：使用二叉搜索树或堆等数据结构存储空闲块，以加速查找过程。

3) **WF**：选择最大的空闲块进行分配，以期望剩余空间足够大，减少形成小碎片的可能性。

- 优点：

  减少小碎片的可能性：优先使用最大块，留下的碎片通常大到可以满足后续分配需求。

  更适合大块内存分配：对于需要大内存的进程，Worst Fit 更容易满足需求。

- 缺点：

  内存利用率低：大块空闲分区被分割后，剩余的空闲内存可能不够其他大块分配请求使用。长期运行可能导致内存利用率降低，剩余空间不足以满足大请求。

  产生更多的内存碎片：虽然 WF 算法试图避免小碎片，但它可能在划分最大的空闲分区后留下多块大小不均的碎片。这些碎片不一定能够满足其他请求，导致内存碎片化问题。

- 如何提高性能：

  维护降序链表：按块大小降序排列空闲块链表，使得第一个块就是最大的块。

  分段管理：将空闲块按大小分段存储（如：10-50、51-100

等），减少遍历范围。

2. 3 种算法的空闲块排序分别是如何实现的。

FF 无需排序。只需要按链表的插入顺序保存空闲块，分配时从头到尾线性查找第一个满足需求的块。

BF 按块大小升序排序。

WF 按块大小降序排序。

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。

1) 内碎片：

内碎片是指分配给进程的内存块中，实际使用的内存小于分配的内存，导致剩余部分无法被利用。

假设进程需要 13 KB 内存，而系统分配的最小块是 16 KB，剩余的 3 KB 是内碎片。

本次实验中，当空闲块大小足够，但分配后的剩余空间小于 MIN_SLICE，则将剩余空间一起分配给进程，此时未被使用的剩余空间则为内碎片。内碎片不能被其他进程使用。

2) 外碎片：

外碎片是指空闲内存总量足够，但被分散成多个不连续的小块，导致无法分配给需要连续大块内存的进程。

假设系统中有 3 个空闲块：5 KB、6 KB 和 4 KB，总计 15 KB，但如果一个进程需要 12 KB 的连续内存，这些碎片无法满足需求。

本次实验中，当空闲块大小足够，而且分配后的剩余空间大于 MIN_SLICE，此时剩余空间则为外碎片。外碎片可以被其他进程使用。

3) 紧缩功能解决的是外碎片。

4. 在回收内存时，空闲块合并是如何实现的？

在回收内存时，新释放的空闲块会插入到空闲链表中，并按照地址有序排列。合并操作通过遍历空闲链表来判断相邻的空闲块是否可以合并，并在条件满足时将它们合并为一个更大的空闲块。

1) 先按 FF 策略重排整个链表。

2) 将新释放的空闲块插入到空闲块链表中合适的位置。如果链表为空，新释放的空闲块成为链表的头节点。

3) 遍历空闲块链表，检查当前块和下一块是否相邻（即 当前块的结束地址 == 下一块的起始地址）。如果相邻，将两个块合并成一个。当前块的大小更新为当前块大小 + 下一块大小，然后跳过下一块节点（即将其从链表中删除）。

4) 在合并完成后，根据当前分配算法，对链表进行重新排列。

2.4 页面的置换

2.4.1 实验目的

通过模拟实现页面置换算法（FIFO、LRU），理解请求分页系统中，页面置换的实现思路， 理解命中率和缺页率的概念，理解程序的局部性原理，理解虚拟存储的原理。

2.4.2 实验内容

1) 理解页面置换算法 FIFO、LRU 的思想及实现的思路。

2) 参考给出的代码思路，定义相应的数据结构，在一个程序中实现上述 2 种算法，运行 时可以选择算法。算法的页面引用序列要至少能够支持随机数自动生成、手动输入两种生成 方式；算法要输出页面置换的过程和最终的缺页率。

3) 运行所实现的算法，并通过对比，分析 2 种算法的优劣。

4) 设计测试数据，观察 FIFO 算法的 BELADY 现象；设计具有局部性特点的测试数据，分 别运行实现的 2 种算法，比较缺页率，并进行分析。

2.4.3 实验思想

1) FIFO 算法：

FIFO 算法的核心思想是：最早进入内存的页面最先被置换。每当有新页面需要加载且内存已满时，优先将最早加载的页面移除。

2) LRU 算法：

LRU 算法假设最近使用的页面在未来仍可能被访问，而较长时间未使用的页面被访问的可能性更低。每当需要置换页面时，优先移除最近最少被使用的页面。

2.4.4 实验步骤和运行结果

1. FIFO Belady 现象：

引用序列：1 2 3 4 1 2 5 1 2 3 4 5

a) 设置帧数为 3，缺页次数为 9，缺页率为 75.00%。

```
FIFO:
Page 1 arrives but miss.
frame: 1
Page 2 arrives but miss.
frame: 1 2
Page 3 arrives but miss.
frame: 1 2 3
Page 4 arrives but miss.
frame: 2 3 4
Page 1 arrives but miss.
frame: 3 4 1
Page 2 arrives but miss.
frame: 4 1 2
Page 5 arrives but miss.
frame: 1 2 5
Page 1 arrives and hit.
frame: 1 2 5
Page 2 arrives and hit.
frame: 1 2 5
Page 3 arrives but miss.
frame: 2 5 3
Page 4 arrives but miss.
frame: 5 3 4
Page 5 arrives and hit.
frame: 5 3 4
Page miss count: 9
Page miss ratio: 75.00%
```

b) 设置帧数为 4，缺页次数为 10，缺页率为 83.33%，高于帧数为 3 时。

```
FIFO:
Page 1 arrives but miss.
frame: 1
Page 2 arrives but miss.
frame: 1 2
Page 3 arrives but miss.
frame: 1 2 3
Page 4 arrives but miss.
frame: 1 2 3 4
Page 1 arrives and hit.
frame: 1 2 3 4
Page 2 arrives and hit.
frame: 1 2 3 4
Page 5 arrives but miss.
frame: 2 3 4 5
Page 1 arrives but miss.
frame: 3 4 5 1
Page 2 arrives but miss.
frame: 4 5 1 2
Page 3 arrives but miss.
frame: 5 1 2 3
Page 4 arrives but miss.
frame: 1 2 3 4
Page 5 arrives but miss.
frame: 2 3 4 5
Page miss count: 10
Page miss ratio: 83.33%
```

2. LRU：

a) 引用序列：0 1 2 1 2 3 1 0 4 5 4 6 5 4 5 6 4 0 1 2

设置帧数为 4。

FIFO：缺页率 50.00%

LRU：缺页率 45.00%，说明这个引用序列局部性良好。

```
reference:
0 1 2 1 2 3 1 0 4 5 4 6 5 4 5 6 4 0 1 2

LRU:
Page 0 arrives but miss.
frame: 0
Page 1 arrives but miss.
frame: 0 1
Page 2 arrives but miss.
frame: 0 1 2
Page 1 arrives and hit.
Page 2 arrives and hit.
Page 3 arrives but miss.
frame: 0 1 2 3
Page 1 arrives and hit.
Page 0 arrives and hit.
Page 4 arrives but miss.
frame: 0 1 4 3
Page 5 arrives but miss.
frame: 0 1 4 5
Page 4 arrives and hit.
Page 6 arrives but miss.
frame: 0 6 4 5
Page 5 arrives and hit.
Page 4 arrives and hit.
Page 5 arrives and hit.
Page 6 arrives and hit.
Page 4 arrives and hit.
Page 0 arrives and hit.
Page 1 arrives but miss.
frame: 0 6 4 1
Page 2 arrives but miss.
frame: 0 2 4 1
Page miss count: 9
Page miss ratio: 45.00%
```

b) 随机生成引用序列：

```
reference:
0 1 1 1 3 0 3 2 2 1

LRU:
Page 0 arrives but miss.
frame: 0
Page 1 arrives but miss.
frame: 0 1
Page 1 arrives and hit.
Page 1 arrives and hit.
Page 3 arrives but miss.
frame: 0 1 3
Page 0 arrives and hit.
Page 3 arrives and hit.
Page 2 arrives but miss.
frame: 0 2 3
Page 2 arrives and hit.
Page 1 arrives but miss.
frame: 1 2 3
Page miss count: 5
Page miss ratio: 50.00%
```

2.4.5 结果分析

1 从实现和性能方面，比较分析 FIFO 和 LRU 算法。

a) 实现复杂度：

- FIFO：通常使用一个队列来维护页面的加载顺序。队列中的页面按访问顺序排列，页面访问时只是简单地检查页面是否在队列中，如果不在，就替换队列头部的页面并将新的页面加入队列尾部。入队和出队操作都非常简单且高效，时间复杂度为 O(1)。
- LRU：要求跟踪每个页面的访问顺序，并总是淘汰最近最少被访问的页面。为了做到这一点，常见的实现方法是使用链表（或哈希表加双向链表）来快速更新页面的访问顺序。在本次实验中，因为帧数较少，选择依次遍历每个帧，找出最近最少访问的页面。

b) 性能分析

- FIFO：当引用串呈现较强的局部性时，FIFO 的缺页率通常较高。FIFO 可能会导致 Belady 现象，即增加帧数时，缺页次数反而增加。FIFO 算法没有考虑页面的访问频率或时间，它仅仅按照页面加载的顺序进行淘汰。对于具有时间局部性的引用串，FIFO 无法有效优化，导致页面经常被不适时地淘汰。
- LRU：LRU 的缺页率通常比 FIFO 低，尤其是在有较强时间局部性的引用串中。LRU 能够根据页面的使用频率来淘汰最不常用的页面，因此能够更好地利用局部性原理，导致 Belady 现象。

2 LRU 算法是基于程序的局部性原理而提出的算法，你模拟实现的 LRU 算法有没有体现出该特点？如果有，是如何实现的？

在 LRU（最近最少使用） 算法中，局部性原理是指程序访问数据时，通常会频繁访问最近访问过的数据（时间局部性），而不常访问的则被淘汰。LRU 算法正是基于这一原则，通过维护页面的访问顺序，淘汰最近最少使用的页面，从而利用局部性原理来优化页面置换的性能。

find_lru 函数的核心逻辑是通过遍历当前帧（内存中的页面），找到访问时间戳最小的页面。这个页面在 LRU 算法中是最近最少使用的页面，符合时间局部性原则，应该被置换出去。

3 在设计内存管理程序时,应如何提高内存利用率。

a) 合理划分内存空间：分配适当大小的内存块：避免申请过多的过大内存块以及过小的内存块。过大的内存块可能导致内存浪费，过小的内存块可能会导致碎片化。在内存分配时，使用合适的内存块大小可以减少内存的浪费。

b) 优化内存分配与回收机制：在不确定内存需求的情况下，尽量避免提前分配过多内存。及时回收不再使用的内存空间。使用适当的垃圾回收机制来释放不再使用的内存，避免内存泄漏。手动内存管理时，可以确保每次分配内存后都进行释放操作。

c) 减少内存碎片：内存管理程序可以合并相邻的空闲块，避免因为分配和释放导致内存碎片，选择合适的内存分配算法来减少内存碎片。使用动态调整内存分配策略（如堆内存和栈内存的动态扩展与收缩），在程序运行时根据需求调整内存的分配和释放。

d) 使用虚拟内存管理：使用分页和分段技术来管理内存。分页机制能够将内存划分为固定大小的页面，而分段机制则根据程序的逻辑结构将内存划分为不同的段。这样可以有效利用内存，并避免外部碎片。通过虚拟内存技术，操作系统可以在硬盘上创建虚拟的内存空间，部分数据可以交换到硬盘，释放内存供其他任务使用。虚拟内存通过页表管理，将程序使用的内存地址空间映射到实际的物理内存，从而提高内存的利用率。

2.5 实验总结

2.5.1 遇到的问题

1. 实验 1 中，子进程也可能收到 SIGINT 或 SIGQUIT 信号。

   解决方法：使用信号屏蔽字和 sigprocmask 函数在子进程中阻塞 SIGINT 和 SIGQUIT，子进程只接受并处理父进程发来的信号 16 或 17。

2. 实验 1 中，父进程向子进程发送信号时，需要保证子进程被阻塞，等待

父进程发来信号。

解决方法：调用 pause 函数阻塞子进程。pause()使进程进入休眠状态，直到收到一个信号并且存在该信号的处理函数。

3. 实验 3 中，紧缩空闲块之后，没有紧缩已分配块，导致空闲块与已分配块冲突。

   解决方法：同时紧缩空闲块和已分配块，紧缩之后重新调用 allocate_mem 按照原有的策略分配新内存。

4. 实验 3 中，当归还已分配块时，需要对空闲块进行重排与合并。

   解决方法：按地址顺序将空闲块插进合适的位置，若采用的是 WF 或 BF 算法，再进行重排。

5. 实验 4 中，实现 FIFO 算法时，需要找到最早调入的页面。

   解决方法：使用循环队列实现，队头即为最早调入的页面。

2.5.2 实验体会

在完成本次操作系统实验后，我对进程的创建、通信、内存管理以及页面置换算法有了更为深刻的理解。

在编程实现进程的创建和软中断通信的过程中，我通过系统调用如 fork()和 signal()等，成功创建了子进程，并实现了父子进程间的软中断通信。通过观察实验现象，我深入理解了进程的调度执行、内存空间分配以及在操作系统中的状态转换。我认识到，进程是操作系统中资源分配和调度的基本单位，而软中断通信则为进程间提供了灵活的同步和通信手段。

在实现进程的管道通信时，我使用了 pipe()调用创建管道，并通过实验现象的分析，理解了管道通信的特点。我掌握了管道通信的同步和互斥机制，认识到管道作为一种先进先出的数据结构，在进程间通信中具有重要作用。同时，我也注意到了管道通信中可能出现的死锁和资源竞争问题。

设计并实现了内存分配管理的三种算法，让我对内存分配及回收的过程有了直观的认识。通过这些算法的实现，我理解了如何根据不同的需求选择合适的内存分配策略，以提高内存的分配效率和利用率。这一过程让我深刻体会到，合理的内存管理对于系统性能的重要性。

在模拟实现页面置换算法的过程中，我对请求分页系统中的页面置换有了更深入的理解。通过实验，我掌握了缺页率的概念，并认识到程序的局部性原理在虚拟存储管理中的重要作用。

本次实验让我对操作系统的核心概念和原理有了更加全面的认识，提高了我的编程能力和系统分析能力。我也意识到实验过程中存在的不足，需要在今后的学习中不断改进和完善。

3 动态模块与设备驱动

3.1 实验目的

1) 学习内核通过动态模块来动态加载内核新功能的机制；
2) 理解 LINUX 字符设备驱动程序的基本原理；
3) 掌握字符设备的驱动运作机制；
4) 学会编写字符设备驱动程序；
5) 学会编写用户程序通过对字符设备的读写完成不同用户间的通信。

3.2 实验内容

1) 编译、安装与卸载动态模块；
2) 实现系统调用的篡改；
3) 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；
4) 编写聊天程序实现不同用户通过该设备的一对一、一对多、多对多聊天。

3.3 实验相关背景

3.3.1 内核模块

Linux 的动态模块（Dynamic Kernel Modules）是指在内核运行时可以被加载或卸载的模块，它们通常用于扩展 Linux 内核的功能，而无需重新编译或重启系统。动态模块有助于减少内核的大小，提高系统的灵活性，能够根据需要添加或删除特性。

动态模块可以在系统运行时加载和卸载，使用 insmod 加载模块，使用和 rmmod 卸载模块。

3.3.2 字符设备驱动

字符设备驱动是 Linux 内核中用于处理字符设备的一种驱动程序。字符设备以字节为单位进行读写，而不像块设备按块操作。字符设备通过一组简单的系统调用接口（如 read、write、ioctl 等）与用户空间交互。

字符设备驱动设计流程：

1) 初始化字符设备：用 cdev_init 初始化 cdev，并将其与 file_operations 绑定。

2) 注册字符设备：使用 register_chrdev_region 或 alloc_chrdev_region 分配设备号，并将设备注册到内核。

3) 实现设备操作函数：实现 file_operations 中的回调函数，例如 read 和 write。

4) 创建设备节点：使用 mknod 创建 /dev 下的设备节点，供用户空间访问。

3.3.3 相关函数

1) wait_event_interruptible()

函数原型：`int wait_event_interruptible(wait_queue_head_t *queue, int condition);`

queue：指向等待队列头部的指针。等待队列是一个数据结构，用于管理所有等待某个条件的进程。内核中的多个进程可能在不同的等待队列上等待不同的条件。

condition：表示进程需要等待的条件。如果这个条件为 false，进程就会被阻塞（即挂起）。当条件为 true 时，进程将从等待队列中被唤醒。

2) 字符串转整型函数 kstrtouint()

函数原型：int kstrtouint(const char *s, unsigned int base, unsigned int *res);

s 是输入的字符串，base 是数值基数，可以是 10（代表十进制）、16（代表十六进制）或者是 0（自动识别基数），res 是用来存放转换后的整型值的指针。当转换过程中没有错误发生时，函数返回 0。

3.4 篡改系统调用的实现

3.4.1 核心思想

arm64 架构下，gettimeofday 的模块号为 169。修改系统调用 gettimeofday 为自定义的 hello 函数。模块加载时会进行替换，卸载时会恢复原有的系统调用。

使用 update_mapping_prot 函数修改特定虚拟地址区域的读/写/执行权限。Sys_call_table 位于内核中的.rodata 段，修改时要临时解除写保护。为了方便操作，直接将整个系统调用表的操作权限修改为读写。

3.4.2 运行结果

- **篡改系统调用前：**

```
[root@kp-test01 ex3]# ./modify_old_syscall
tv_sec:1733152031
tv_usec:67318
[root@kp-test01 ex3]# ./modify_new_syscall
-1
```

- **篡改系统调用后：**

```
[root@kp-test01 ex3]# ./modify_old_syscall
tv_sec:4196032
tv_usec:0
[root@kp-test01 ex3]# ./modify_new_syscall
30
```

3.4.3 遇到的问题

1. 运行 insmod modify_syscall.ko，内核崩溃。

   原因：sys_call_table 在内核中是只读的。没有禁用写保护，因此不能直接更改系统调用表。需要提前解除写保护，修改系统调用后再恢复。使用 update_mapping_prot 函数修改特定虚拟地址区域的读/写/执行权限。

   代码中没有更改系统调用表的地址，使用`rep sys_call_table /boot/System.map-$(uname -r)静态查询系统调用表，或使用 kallsyms_lookup_name("sys_call_table")动态查询。

2. 运行./modify_new_syscall，输出一个非 30 的整数。

   原因：参数 10、20 没有直接传递，不能直接在 hello 函数中使用 (int a, int b) ，需要从寄存器中读取参数。

3.5 字符设备驱动程序的实现

3.5.1 设计思想

创建了一个新的字符设备驱动，修改原有设备驱动中的文件操作，重载 `read()` `write()` `open()` `release()`四个函数。

- globalvar_open()：打开设备时调用，通过 pid 注册客户端并初始化客户端结构体的相关数据。
- globalvar_release()：释放设备时调用，释放对应的客户端数据。

- globalvar_read()：读取设备时调用，客户端通过该函数读取消息。如果有消息，消息被复制到目标客户端的用户空间，否则等待新消息。
- globalvar_write()：写入设备时调用，客户端可以通过该函数发送消息。支持群发消息（发送给所有客户端）和私聊消息（只发送给指定的客户端）。

对于私聊消息，globalvar_write()解析出目标 pid，将消息拷贝进目标客户端的消息队列。

对于广播消息，globalvar_write()将消息拷贝进所有目标客户端的消息队列。

当进程从内核空间将未读消息拷贝进自己的用户空间之后，该消息在消息队列中将被删除。

3.5.2 多对多通信的实现

每个客户端（进程）都有一个唯一的 pid，和一个消息队列用于存储消息。客户端通过设备文件与内核通信，系统会为每个客户端分配一个 client_data 结构体来存储相关信息。

每当一个进程打开设备文件时，会调用 globalvar_open 来注册该进程并创建或查找相应的 client_data 结构体。

每个客户端通过向设备写入消息来向其他客户端发送信息。消息的格式包括发送者的 pid 和消息内容。globalvar_write 函数处理消息的发送。对于私聊消息，只会被放入目标客户端的消息队列；对于群发消息，会放进所有客户端的消息队列。

globalvar_read() 函数从客户端的消息队列中读取消息并拷贝用户空间。每个客户端的消息队列按 FIFO 顺序读取。读过的消息会从消息队列里删除。

3.5.3 同步与互斥的实现

```
struct client_data {
    pid_t pid;
    struct message message_queue[MAX_MESSAGES];
    int message_count;
```

每个客户端都有独立的消息队列，其消息不会受到其他客户端的影响，不同客户端之间的读写操作可以互不干扰。如果使用全局的单一消息队列，当队列满或阻塞时，所有客户端的操作都会受影响。

```
struct chat_device_data {
    struct client_data *clients[MAX_CLIENTS];
    struct semaphore read_sem;
    struct semaphore write_sem;
    int readers_count;
    wait_queue_head_t read_queue;
    wait_queue_head_t write_queue;
```

```
    int client_count;
};
```

读写锁：read_sem 和 write_sem 确保同一时刻只有一个进程可以进行写入操作，写入的时候其他进程不能开始读。

等待队列：read_queue 和 write_queue 协调读进程和写进程的操作。

当某个进程调用 read，但当前没有未读的消息时，该进程会被加入到 read_queue 中，并进入睡眠状态；同理，当某个进程调用 write，但设备或缓冲区暂时无法接受数据时（如当前有进程在读），该进程会被加入到 write_queue 中，并进入睡眠状态。

在设备数据结构体 chat_device_data 中，加入整型变量 readers_count，实时监测当前读进程的个数。当 readers_count==0 时，才允许写入消息。

3.6 测试程序的实现

3.6.1 设计思想

测试程序使用多线程来实现基于字符设备的并发通信。通过两个线程分别处理消息的发送和接收，程序验证了字符设备的基本功能、消息传递能力以及线程间的协作。

私聊消息格式：sender_pid|@target_pid message

群聊消息格式：sender_pid|message

在每条消息前添加上发送进程的 pid，这样方便目标进程解析出本条消息的发送方。

3.6.2 读写进程的实现

创建读线程和写线程，一个用于接收消息，另一个用于发送消息。写线程等待用户写入消息，读线程实时打印其他进程发来的消息。

写线程从用户输入中读取消息。使用 snprintf 格式化消息为 sender_pid|message_content，并使用 write() 将消息写入设备文件；接收消息线程（receive_message）；读线程从设备文件中读取消息、解析消息，提取发送者 PID 和消息内容，打印出来。

```
    // 接收消息线程
    if (pthread_create(&receive_thread, NULL, receive_message, (void
*)&args) != 0) {
        perror("Failed to create receive thread");
        close(fd);
        return -1;
    }

    // 发送消息线程
```

```
    if (pthread_create(&send_thread, NULL, send_message, (void *)&args) !=
0) {
        perror("Failed to create send thread");
        close(fd);
        return -1;
    }
```

3.7 运行结果

在终端分别创建三个进程，pid 分别为 3633、3661、3700。

1) 进程 3633 向进程 3700 发送"Hello! I am 3633."。

```
My PID is 3700.
Enter message to send (or type 'quit' to exit):
Child PID 3700 received message from PID 3633: @3700
Hello! I am 3633.
```

```
My PID is 3633.
Enter message to send (or type 'quit' to exit):
@3700 Hello! I am 3633.
Message sented.
```

2) 进程 3700 向进程 3661 发送"Hi!"。

```
My PID is 3700.
Enter message to send (or type 'quit' to exit):
Child PID 3700 received message from PID 3633: @3700
Hello! I am 3633.
@3661 Hi!
Message sented.
```

```
My PID is 3661.
Enter message to send (or type 'quit' to exit):
Child PID 3661 received message from PID 3700: @3661
Hi!
```

3) 进程 3661 向 3633、3700 广播"It is a good day." "Nice to meet you."。

```
My PID is 3661.
Enter message to send (or type 'quit' to exit):
Child PID 3661 received message from PID 3700: @3661
Hi!
It is a good day.
Message sented.
Enter message to send (or type 'quit' to exit):
Nice to meet you.
Message sented.
```

```
My PID is 3633.
Enter message to send (or type 'quit' to exit):
@3700 Hello! I am 3633.
Message sented.
Enter message to send (or type 'quit' to exit):
Child PID 3633 received message from PID 3661: It is
a good day.
Child PID 3633 received message from PID 3661: Nice t
o meet you.
```

```
My PID is 3700.
Enter message to send (or type 'quit' to exit):
Child PID 3700 received message from PID 3633: @3700
Hello! I am 3633.
@3661 Hi!
Message sented.
Enter message to send (or type 'quit' to exit):
Child PID 3700 received message from PID 3661: It is
a good day.
Child PID 3700 received message from PID 3661: Nice t
o meet you.
```

测试程序实现了进程之间一对一、一对多、多对多的私聊、广播通信，运行结果符合预期。

3.8 实验总结

3.8.1 遇到的问题

1. 运行测试程序时，出现报错 Failed to open device: No such file or directory。

   原因与解决方法：代码中没有创建设备类和设备节点，需要手动创建。`mknod /dev/globalvar c 290 0`，在 /dev 下创建设备文件，290 是主设备号，与驱动中的 MAJOR_NUM 相同。0 是次设备号。后续在代码里加入了 class_create 和 device_create 函数，可以自动创建设备类和设备节点，不再使用 mknod 指令手动创建。

2. 实现群发功能时，如果只使用一个缓冲区，难以判断所有目标进程是否都读取到消息并拷贝到用户空间。

   解决方法：给每个客户端（进程）创建一个消息队列，群发时把消息放进所有进程的消息队列。这样虽然重复占用内核空间，但是能保证每个进程对于消息的读取是独立的。

3. 测试时，向另一个进程发送消息，接收方不能实时打印收到的消息

原因：接受方正在等待写操作的输入，进程卡在 scanf。

解决方法：创建读线程和写线程，一个用于接收消息，另一个用于发送消息。写线程不断等待用户写入消息，读线程实时打印其他进程发来消息。

3.8.2 实验收获

通过完成动态模块和设备驱动的实验，我对 Linux 内核模块的设计与开发有了更加深入的理解，同时也掌握了字符设备驱动的基本实现方法和调试技巧。

在实验中，我将曾经学习的模块加载、卸载理论应用到实践中，学会了如何编写一个简单的动态模块，并通过命令 insmod 和 rmmod 实现模块的动态加载与卸载。这让我深刻体会到 Linux 内核的可扩展性和模块化设计的优势。掌握了字符设备驱动的核心功能，包括 open、read、write 和 close 函数的实现。

在实验中设计了用户态测试程序，通过多线程实现对字符设备的读写操作，验证了设备驱动程序的并发处理能力。这让我对线程安全和设备驱动的并发设计有了新的认识。

通过这次实验，我不仅对 Linux 内核模块和设备驱动的原理有了更直观的认识，还锻炼了自己的实际编程能力和调试技巧。同时，我意识到设备驱动开发是一项需要理论与实践深度结合的技术工作，为我未来进一步学习内核开发和系统编程打下了良好的基础。

4 附录

4.1 实验完整代码

4.1.1 实验一

```c
// 1.11.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){
        fprintf (stderr, "Fork Failed");
        return 1;
    }

    else if (pid == 0){
        pid1 = getpid();
        printf("child: pid = %d  ", pid);
        printf("child: pid1 = %d  ", pid1);
    }

    else{
        pid1 = getpid();
        printf("parent: pid = %d  ", pid);
        printf("parent: pid1 = %d  ", pid1);
        wait(NULL);
    }

    return 0;
}
```

```
// 1.12.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){
        fprintf (stderr, "Fork Failed");
        return 1;
    }

    else if (pid == 0){
        pid1 = getpid();
        printf("child: pid = %d  ", pid);
        printf("child: pid1 = %d  ", pid1);
    }

    else{
        pid1 = getpid();
        printf("parent: pid = %d  ", pid);
        printf("parent: pid1 = %d  ", pid1);
        //wait(NULL);
```

```
    }

    return 0;
}
```

```
// 1.13.c
#include <sys/types.h>
#include <sys/wait.h>
#include <stdio.h>
#include <unistd.h>

int value = 0;
int* p = &value;

int main(){
    pid_t pid, pid1;

    pid = fork();

    if (pid < 0){
        fprintf (stderr, "Fork Failed");
        return 1;
    }

    else if (pid == 0){
        pid1 = getpid();
        value = 1;
        //printf("child: pid = %d  ", pid);
        //printf("child: pid1 = %d  ", pid1);
        printf("child: global value = %d  ", value);
        printf("child: *value = %p  ", p);
    }

    else{
        pid1 = getpid();
        value = 3;
        //printf("parent: pid = %d  ", pid);
        //printf("parent: pid1 = %d  ", pid1);
        printf("parent: global value = %d  ", value);
        printf("parent: *value = %p  ", p);
        wait(NULL);
    }

    return 0;
}
```

```
// 1.14.c
#include <sys/types.h>
```

```c
#include <sys/wait.h>
#include <stdio.h>
#include <unistd.h>

int value = 0;
int* p = &value;

int main(){
    pid_t pid, pid1;

    pid = fork();

    if (pid < 0){
        fprintf (stderr, "Fork Failed");
        return 1;
    }

    else if (pid == 0){
        pid1 = getpid();
        value = 1;
        printf("child: value = %d  ", value);
        printf("child: *value = %p  \n", p);
    }

    else{
        pid1 = getpid();
        value = 3;
        printf("parent: value = %d  ", value);
        printf("parent: *value = %p  \n", p);
        wait(NULL);
    }

    // value++
    value += 1;
    printf("before return: value = %d  ", value);
    printf("before return: *value = %p  \n", p);

    return 0;
}
```

```c
// 1.21.c
#include <stdio.h>
#include <pthread.h>

int shared_var = 0;

// +100
void* increment(void* arg) {
    for (int i = 0; i < 100000; i++) {
```

```
        shared_var += 100;
    }
    return NULL;
}

// -100
void* decrement(void* arg) {
    for (int i = 0; i < 100000; i++) {
        shared_var -= 100;
    }
    return NULL;
}

int main() {
    pthread_t thread1, thread2;

    pthread_create(&thread1, NULL, increment, NULL);
    if (thread1)
        printf("thread1 create success!\n");
    else return 1;

    pthread_create(&thread2, NULL, decrement, NULL);
    if (thread2)
        printf("thread2 create success!\n");
    else return 1;

    pthread_join(thread1, NULL);
    pthread_join(thread2, NULL);

    printf("variable result: %d\n", shared_var);

    return 0;
}
```

```
// 1.22.c
#include <stdio.h>
#include <pthread.h>
#include <semaphore.h>

int shared_var = 0;
sem_t signal;

// +100
void* increment(void* arg) {
    for (int i = 0; i < 100000; i++) {
        sem_wait(&signal);     // P
        shared_var += 100;
        sem_post(&signal);     // V
```

```
    }
    return NULL;
}

// -100
void* decrement(void* arg) {
    for (int i = 0; i < 100000; i++) {
        sem_wait(&signal);     // P
        shared_var -= 100;
        sem_post(&signal);     // V
    }
    return NULL;
}

int main() {
    pthread_t thread1, thread2;

    sem_init(&signal, 0, 1);
    pthread_create(&thread1, NULL, increment, NULL);
    if (thread1)
        printf("thread1 create success!\n");
    else return 1;

    pthread_create(&thread2, NULL, decrement, NULL);
    if (thread2)
        printf("thread2 create success!\n");
    else return 1;

    pthread_join(thread1, NULL);
    pthread_join(thread2, NULL);

    printf("variable result: %d\n", shared_var);

    sem_destroy(&signal);

    return 0;
}
```

```
// 1.151.c
#include <sys/types.h>
#include <sys/wait.h>
#include <stdio.h>
#include <unistd.h>
#include <stdlib.h>

int main(){
    pid_t pid, pid1;
```

```
    pid = fork();

    if (pid < 0){
        fprintf (stderr, "Fork Failed");
        return 1;
    }

    else if (pid == 0){
        pid1 = getpid();
        printf("child PID = %d\n", pid1);

        // 使用 system() 调用外部命令执行 system_call 程序
        system("./system_call");
        printf("child PID = %d\n", pid1);
    }

    else{
        pid1 = getpid();
        printf("parent PID = %d\n", pid1);
        wait(NULL);
    }

    return 0;
}
```

```
// 1.152.c
#include <sys/types.h>
#include <sys/wait.h>
#include <stdio.h>
#include <unistd.h>
#include <stdlib.h>

int main(){
    pid_t pid, pid1;

    pid = fork();

    if (pid < 0){
        fprintf (stderr, "Fork Failed");
        return 1;
    }

    else if (pid == 0){
        pid1 = getpid();
        printf("child PID = %d\n", pid1);

        // 使用 exec() 系列函数调用 system_call 程序
        char *args[] = {"./system_call", NULL};
        execvp(args[0], args);
```

```
        printf("child PID = %d\n", pid1);
    }

    else{
        pid1 = getpid();
        printf("parent PID = %d\n", pid1);
        wait(NULL);
    }

    return 0;
}
```

```
// 1.231.c
#include <stdio.h>
#include <pthread.h>
#include <unistd.h>
#include <sys/types.h>
#include <sys/syscall.h>
#include <sys/wait.h>
#include <stdlib.h>

void* pthread_1(void* arg) {
    printf("thread1 tid = %d, pid = %ld\n", syscall(SYS_gettid), getpid());
    system("./system_call");
    printf("thread1 systemcall return\n");
    return NULL;
}

void* pthread_2(void* arg) {
    printf("thread2 tid = %d, pid = %ld\n", syscall(SYS_gettid), getpid());
    system("./system_call");
    printf("thread2 systemcall return\n");
    return NULL;
}

int main() {
    pthread_t thread1, thread2;

    pthread_create(&thread1, NULL, pthread_1, NULL);
    if (thread1)
        printf("thread1 create success!\n");
    else return 1;

    pthread_create(&thread2, NULL, pthread_2, NULL);
    if (thread2)
        printf("thread2 create success!\n");
    else return 1;

    pthread_join(thread1, NULL);
```

```
    pthread_join(thread2, NULL);

    return 0;
}
```

```c
// 1.232.c
#include <stdio.h>
#include <pthread.h>
#include <unistd.h>
#include <sys/types.h>
#include <sys/syscall.h>
#include <sys/wait.h>
#include <stdlib.h>

void* pthread_1(void* arg) {
    printf("thread1 tid = %d, pid = %ld\n", syscall(SYS_gettid), getpid());
    char *args[] = {"./system_call", NULL};
    execvp(args[0], args);
    printf("thread1 systemcall return\n");
    return NULL;
}

void* pthread_2(void* arg) {
    printf("thread2 tid = %d, pid = %ld\n", syscall(SYS_gettid), getpid());
    char *args[] = {"./system_call", NULL};
    execvp(args[0], args);
    printf("thread2 systemcall return\n");
    return NULL;
}

int main() {
    pthread_t thread1, thread2;

    pthread_create(&thread1, NULL, pthread_1, NULL);
    if (thread1)
        printf("thread1 create success!\n");
    else return 1;

    pthread_create(&thread2, NULL, pthread_2, NULL);
    if (thread2)
        printf("thread2 create success!\n");
    else return 1;

    pthread_join(thread1, NULL);
    pthread_join(thread2, NULL);

    return 0;
}
```

```c
/**
*spinlock.c
*in xjtu
*2024.10
*/
#include <stdio.h>
#include <pthread.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
// 线程 1: value++
void *thread1_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;
}
// 线程 2: value--
void *thread2_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;

    printf("Shared value: %d\n", shared_value);

    // 初始化自旋锁
    spinlock_init(&lock);

    // 创建两个线程
    pthread_create(&thread1, NULL, thread1_function, &lock);
    if (thread1)
        printf("thread1 create success!\n");
    else return 1;
    pthread_create(&thread2, NULL, thread2_function, &lock);
    if (thread2)
        printf("thread2 create success!\n");
    else return 1;

    // 等待线程结束
    pthread_join(thread1, NULL);
    pthread_join(thread2, NULL);

    // 输出共享变量的值
    printf("Shared value: %d\n", shared_value);
    return 0;
}
```

```
// system_call.c
#include <stdio.h>
#include <unistd.h>

int main() {
    printf("system_call: PID = %d\n", getpid());
    return 0;
}
```

4.1.2 实验二

```c
// 2.1.c
#include <stdio.h>
#include <unistd.h>
#include <sys/wait.h>
#include <stdlib.h>
#include <signal.h>


int handled = 0;
//int flag = 0;
pid_t pid1 = -1, pid2 = -1;
int terminated_1 = 0, terminated_2 = 0;

void inter_handler(int signum) {
    // TODO: 处理不同信号
    if (handled) return; // 保证只处理一次信号
    handled = 1;

    printf("\n%d stop test\n", signum); // SIGINT 对应 Ctrl + C, SIGQUIT 对
应 Ctrl + '\'

    if (signum == SIGQUIT || signum == SIGINT || signum == SIGALRM){
        kill(pid1, SIGSTKFLT); // 发送信号 16
        kill(pid2, SIGCHLD); // 发送信号 17
        // kill(pid1, SIGALRM); // 闹钟中断 14
        // kill(pid2, SIGALRM); // 闹钟中断 14
    }
}

void handle_usr1(int signum) {
    terminated_1 = 1;
    printf("\n%d stop test\n", signum); // SIGSTKFLT
}

void handle_usr2(int signum) {
    terminated_2 = 1;
    printf("\n%d stop test\n", signum); // SIGCHLD
}

void block_sig() {
    // 在子进程中屏蔽 SIGINT 和 SIGQUIT 信号
    sigset_t set;
    sigemptyset(&set);
    sigaddset(&set, SIGINT);
    sigaddset(&set, SIGQUIT);
    sigprocmask(SIG_BLOCK, &set, NULL);
}
```

```
int main() {
    // TODO: 五秒之后或接收到两个信号
    while (pid1 == -1) pid1=fork();
    if (pid1 > 0) {
        while (pid2 == -1) pid2=fork();
        if (pid2 > 0) {
            // TODO: 父进程
                    // 信号处理函数
            signal(SIGINT, inter_handler);  // 捕捉 SIGINT（如 Ctrl + C）
            signal(SIGQUIT, inter_handler); // 捕捉 SIGQUIT（如 Ctrl + \）
            signal(SIGALRM, inter_handler);  // 捕捉 SIGALRM（超时信号）
            alarm(5);
            pause();

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

            return 0;
        }
        else {
            // TODO: 子进程 2
            block_sig();
            signal(SIGCHLD, handle_usr2);
            // signal(SIGALRM, handle_usr2);
            pause();
            if (terminated_2){
                printf("\nChild process2 is killed by parent!!\n");
                exit(0);
            }
        }
    }
    else {
        // TODO: 子进程 1
        block_sig();
        signal(SIGSTKFLT, handle_usr1);
        // signal(SIGALRM, handle_usr1);
        pause();
        if (terminated_1){
            printf("\nChild process1 is killed by parent!!\n");
            exit(0);
        }
 }
 return 0;
}
```

```
// 2.2.c
#include <unistd.h>
```

```c
#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[4001]; // 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
        // lockf(fd[1], 1, 0); // 锁定管道
        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); // 解除管道的锁定
        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) {
            // 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);
            exit(0);
        }
        else {
            wait(0); // 等待子进程 1 结束
            wait(0); // 等待子进程 2 结束
            read(fd[0], InPipe, 4000); // 从管道中读出 4000 个字符
            InPipe[4000] = '\0'; // 加字符串结束符
            printf("%s\n",InPipe); // 显示读出的数据
            exit(0); // 父进程结束
        }
    }
}
```

```c
// 2.3.c
/*
1 - Set memory size (default=1024)
2 - Select memory allocation algorithm
3 - New process
4 - Terminate a process
```

```
5 - Display memory usage
0 - Exit
*/
#include <stdio.h>
#include <stdlib.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
int mem_size=DEFAULT_MEM_SIZE; /*内存大小*/
int ma_algorithm = MA_FF; /*当前分配算法*/
static int pid = 0; /*初始 pid*/
int flag = 0; /*设置内存大小标志*/

/*描述每一个空闲块的数据结构*/
struct free_block_type{
    int size;
    int start_addr;
    struct free_block_type *next;
};
/*指向内存中空闲块链表的首指针*/
struct free_block_type *free_block;

/*每个进程分配到的内存块的描述*/
struct allocated_block{
    int pid; int size;
    int start_addr;
    char process_name[PROCESS_NAME_LEN];
    struct allocated_block *next;
};
/*进程分配内存块链表的首指针*/
struct allocated_block *allocated_block_head = NULL;

/*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;
    }
    fb->size = mem_size;
    fb->start_addr = DEFAULT_MEM_START;
    fb->next = NULL;
    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("Cannot set memory size again\n");
        return 0;
    }
    printf("Total memory size =");
    scanf("%d", &size);
    if(size>0) {
        mem_size = size;
        // init_free_block(mem_size);
        free_block->size = mem_size;
    }
    flag=1;
    return 1;
}

/*按 FF 算法重新整理内存空闲块链表*/
//
void rearrange_FF() {
    struct free_block_type *i, *j;
    int temp_size, temp_start;

    // 按起始地址升序排列空闲块链表
    for (i = free_block; i != NULL; i = i->next) {
        for (j = i->next; j != NULL; j = j->next) {
            if (i->start_addr > j->start_addr) {
                // 交换 i 和 j 的大小和起始地址
                temp_size = i->size;
                temp_start = i->start_addr;

                i->size = j->size;
                i->start_addr = j->start_addr;

                j->size = temp_size;
                j->start_addr = temp_start;
            }
        }
    }
}

/*按 BF 算法重新整理内存空闲块链表*/
void rearrange_BF(){
    struct free_block_type *i, *j;
    int temp_size, temp_start;
    // 按照空闲块大小升序排列
    for(i = free_block; i != NULL; i = i->next){
```

```
        for(j = i->next; j != NULL; j = j->next){
            if(i->size > j->size){
                temp_size = i->size;
                temp_start = i->start_addr;
                i->size = j->size;
                i->start_addr = j->start_addr;
                j->size = temp_size;
                j->start_addr = temp_start;
            }
        }
    }
}

/*按 WF 算法重新整理内存空闲块链表*/
void rearrange_WF(){
    struct free_block_type *i, *j;
    int temp_size, temp_start;
    // 按照空闲块大小降序排列
    for(i = free_block; i != NULL; i = i->next){
        for(j = i->next; j != NULL; j = j->next){
            if(i->size < j->size){
                temp_size = i->size;
                temp_start = i->start_addr;
                i->size = j->size;
                i->start_addr = j->start_addr;
                j->size = temp_size;
                j->start_addr = temp_start;
            }
        }
    }
}

/*按指定的算法整理内存空闲块链表*/
void rearrange(int algorithm){
    switch(algorithm){
        case MA_FF: rearrange_FF(); break;
        case MA_BF: rearrange_BF(); break;
        case MA_WF: rearrange_WF(); break;
    }
}

/* 设置当前的分配算法 */
void set_algorithm(){
    int algorithm;
    printf("\t1 - First Fit\n");
    printf("\t2 - Best Fit \n");
    printf("\t3 - Worst Fit \n");
    scanf("%d", &algorithm);
    if(algorithm>=1 && algorithm <=3)
    ma_algorithm=algorithm;
```

```
    //按指定算法重新排列空闲区链表
    rearrange(ma_algorithm);
}

/* 内存紧缩函数 */
void compact_free_blocks(int free_block_size) {
    struct free_block_type *cur = free_block;
    struct free_block_type *new_block;
    struct allocated_block *ab = allocated_block_head;
    int current_address = 0;

    // 创建一个新的空闲块表示紧缩后的连续区域
    new_block = (struct free_block_type *)malloc(sizeof(struct
free_block_type));
    if (!new_block) {
        printf("Memory compaction failed: no sufficient system memory.\n");
        return;
    }
    new_block->start_addr = DEFAULT_MEM_START; // 紧缩后从内存起始地址开始
    new_block->size = free_block_size;
    new_block->next = NULL;

    // 释放原有空闲块链表
    cur = free_block;
    while (cur != NULL) {
        struct free_block_type *temp = cur;
        cur = cur->next;
        free(temp);
    }

    // 更新空闲块链表为紧缩后的新块
    free_block = new_block;

    printf("Memory compaction completed: total free size = %d.\n",
free_block_size);

    // 紧缩已分配块
    current_address = free_block->start_addr + free_block->size;
    while (ab != NULL) {
        ab->start_addr = current_address;
        current_address += ab->size;
        ab = ab->next;
    }

}

/*分配内存模块*/
int allocate_mem(struct allocated_block *ab){
    struct free_block_type *fbt, *pre, *tfbt; //pre:前一个 tfbt = fbt
    int request_size=ab->size;
```

```
    int free_block_size = 0;
    fbt = free_block;
    pre = NULL;
    tfbt = fbt;
    //计算所有剩余空闲块的大小
    while (tfbt != NULL){
        free_block_size += tfbt->size;
        tfbt = tfbt->next;
    }
    //在空闲分区链表中搜索合适空闲分区进行分配
    while (fbt != NULL){
        //找到可满足空闲分区且分配后剩余空间足够大，则分割
        if (fbt->size - request_size >= MIN_SLICE){
            ab->start_addr = fbt->start_addr;
            fbt->start_addr += request_size;
            fbt->size -= request_size;
            return 1;
        }
        //找到可满足空闲分区且但分配后剩余空间比较小，则一起分配
        else if (fbt->size >= request_size && fbt->size - request_size <
MIN_SLICE){
            ab->start_addr = fbt->start_addr;
            if(pre == NULL) free_block = fbt->next;
            else pre->next = fbt->next;
            free(fbt);
            return 1;
        }
        //找不可满足需要的空闲分区但空闲分区之和能满足需要，则采用内存紧缩技术，进
行空闲分区的合并，然后再分配
        else if (fbt->size < request_size && free_block_size >=
request_size){
            compact_free_blocks(free_block_size);
            return allocate_mem(ab);
        }
        //分配不成功
        else {
            pre = fbt;
            fbt = fbt->next;
        }
    }
    return -1;
}

/*创建新的进程，主要是获取内存的申请数量*/
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);
    ab->pid = pid;
    printf("Memory for %s:", ab->process_name);
    scanf("%d", &size);
    if(size>0) ab->size=size;
    ret = allocate_mem(ab); /* 从空闲区分配内存，ret==1 表示分配 ok*/
    /*如果此时 allocated_block_head 尚未赋值，则赋值*/
    if((ret==1) &&(allocated_block_head == NULL)){
        allocated_block_head=ab;
        return 1; }
    /*分配成功，将该已分配块的描述插入已分配链表*/
    else if (ret==1) {
        ab->next=allocated_block_head;
        allocated_block_head=ab;
        return 2; }
    else if(ret==-1){ /*分配不成功*/
        printf("Allocation fail\n");
        free(ab);
        return -1;
    }
    return 3;
}

/*将 ab 所表示的已分配区归还，并进行可能的合并*/
int free_mem(struct allocated_block *ab){
    struct free_block_type *fbt, *pre = NULL;
    struct free_block_type *cur = free_block;
    fbt=(struct free_block_type*) malloc(sizeof(struct free_block_type));
    if(!fbt) return -1;
    fbt->size = ab->size;
    fbt->start_addr = ab->start_addr;
    fbt->next = NULL;

    // 插入空闲块列表并按地址排序
    rearrange_FF();
        // 找到合适的位置
    while(cur != NULL && cur->start_addr < fbt->start_addr){
        pre = cur;
        cur = cur->next;
    }
        // 插入
    if(pre == NULL){
        fbt->next = free_block;
        free_block = fbt;
    } else {
        fbt->next = pre->next;
        pre->next = fbt;
    }
```

```
    // 合并相邻空闲块
        cur = free_block;
        while(cur != NULL && cur->next != NULL){
            if(cur->start_addr + cur->size == cur->next->start_addr){
                cur->size += cur->next->size;
                struct free_block_type *temp = cur->next;
                cur->next = cur->next->next;
                free(temp);
            } else {
                cur = cur->next;
            }
        }
    rearrange(ma_algorithm); // 按当前算法重新排列空闲链表
    return 1;
}

/*释放 ab 数据结构节点*/
int dispose(struct allocated_block *free_ab){
    struct allocated_block *pre, *ab;
    if(free_ab == allocated_block_head) { /*如果要释放第一个节点*/
        allocated_block_head = allocated_block_head->next;
        free(free_ab);
        return 1;
    }
    pre = allocated_block_head;
    ab = allocated_block_head->next;
    while(ab!=free_ab){
        pre = ab;
        ab = ab->next;
    }
    pre->next = ab->next;
    free(ab);
    return 2;
}

/* 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;
    // if(fbt == NULL && ab == NULL) {
    if(fbt == NULL) {
        printf("No free memory.\n");
        // return(-1);
    }
    printf("----------------------------------------------------------\n");
    /* 显示空闲区 */
    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;
    }
    /* 显示已分配区 */
    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 0;
}

/*找到 pid 对应的存储空间*/
struct allocated_block* find_process(int pid) {
    struct allocated_block *ab = allocated_block_head;
    while (ab != NULL) {
        if (ab->pid == pid) {
            return ab;
        }
        ab = ab->next;
    }
    return NULL; // 找不到该 pid
}

/*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节点*/
void kill_process(){
    struct allocated_block *ab;
    int pid;
    printf("Kill Process, pid=");
    scanf("%d", &pid);
    ab=find_process(pid);
    if(ab!=NULL){
        free_mem(ab); /*释放 ab 所表示的分配区*/
        dispose(ab); /*释放 ab 数据结构节点*/
    }
    else printf("This PID is invalid.\n");
}

void do_exit() {
    struct free_block_type *free_ptr;
    struct allocated_block *alloc_ptr;

    // 释放空闲块链表
    while (free_block != NULL) {
        free_ptr = free_block;
        free_block = free_block->next;
        free(free_ptr);
```

```
    }

    // 释放已分配块链表
    while (allocated_block_head != NULL) {
        alloc_ptr = allocated_block_head;
        allocated_block_head = allocated_block_head->next;
        free(alloc_ptr);
    }

    printf("Memory successfully released. Exiting program.\n");
}

/*显示菜单*/
void display_menu(){
    printf("\n");
    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");
}

int main(){
    char choice; pid=0;
    free_block = init_free_block(mem_size); //初始化空闲区
    while(1) {
        if (choice != '\n') // 清除缓冲区的'\n'
            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;
        }
    }
}
```

```
// 2.4.c
#include <stdio.h>
#include <stdbool.h>
#include <stdlib.h>
#include <time.h>
```

```c
#include <limits.h>

// 页表项结构体
typedef struct page {
    int frame;      // 帧号
    int flag;       // （是否换入）
    int counter;
} page;

page *page_table = NULL;
int *reference = NULL;
int table_size;
int frame_size;
int ref_len;

void initial_page_table() {
    page_table = (page *)malloc(table_size * sizeof(page));
    if (!page_table) {
        printf("page_table: malloc failed!\n");
        exit(0);
    }
    for (int i = 0; i < table_size; i++) {
        page_table[i].flag = 0;
        page_table[i].counter = 0;
    }
}

// 输出引用串
void print_reference() {
    printf("\nreference:\n");
    for (int i = 0; i < ref_len; i++) {
        printf("%d ", reference[i]);
    }
    printf("\n");
}

// 随机生成引用串
void generate_random_reference() {
    srand(time(NULL));
    for (int i = 0; i < ref_len; i++) {
        reference[i] = rand() % table_size;
    }
    // print_reference();
}

// 释放页表和引用串的内存
void release() {
    if (page_table) {
        free(page_table);
    }
```

```
    if (reference) {
        free(reference);
    }
}

// FIFO 页面置换算法
int FIFO(void) {
    int miss = 0;    // 缺页次数
    int front = 0;   // 循环队列的队头
    int rear = 0;    // 循环队列的队尾
    int *queue = (int *)malloc(frame_size * sizeof(int));
    if (!queue) {
        printf("queue: malloc failed!\n");
        exit(0);
    }

    for (int i = 0; i < ref_len; i++) {
        printf("Page %d arrives ", reference[i]);
        // 页面不在帧中
        if (!page_table[reference[i]].flag) {
            printf("but miss.\n");
            miss++;
            if ((rear + 1) % frame_size == front) {
                // 帧满，换出页面
                page_table[queue[front]].flag = 0;
                front = (front + 1) % frame_size;
            }
            // 页面换入
            page_table[reference[i]].flag = 1;
            page_table[reference[i]].frame = rear;
            queue[rear] = reference[i];
            rear = (rear + 1) % frame_size;
        } else {
            // 页面在帧中
            printf("and hit.\n");
        }
        // 输出当前帧内容
        printf("frame: ");
        for (int k = front; k != rear; k = (k + 1) % frame_size) {
            printf("%d ", queue[k]);
        }
        printf("\n");
    }

    free(queue);
    return miss;
}

// 查找最近最少使用的页面
int find_lru(int *frame) {
```

```
    int max = -1, frame_num = 0;
    for (int i = 0; i < frame_size; i++) {
        if (page_table[frame[i]].counter < max || max == -1) {
            max = page_table[frame[i]].counter;
            frame_num = i;
        }
    }
    return frame_num;
}

int LRU(void) {
    frame_size--;
    int miss = 0;      // 缺页次数
    int clock = 0;     // 逻辑时钟
    int full = 0;      // 帧满标志
    int *frame = (int *)malloc(frame_size * sizeof(int));  // 帧数组
    if (!frame) {
        printf("frame: malloc failed!\n");
        exit(0);
    }

    for (int i = 0; i < frame_size; i++) {
        frame[i] = -1; // 初始化为空帧
    }

    for (int i = 0; i < ref_len; i++) {
        printf("Page %d arrives ", reference[i]);
        page_table[reference[i]].counter = clock;
        clock++;

        // 页面不在帧中
        if (!page_table[reference[i]].flag) {
            printf("but miss.\n");
            miss++;
            if (full < frame_size) {
                // 帧未满，直接加载
                page_table[reference[i]].flag = 1;
                page_table[reference[i]].frame = full;
                frame[full] = reference[i];
                full++;
            } else {
                // 帧已满，选择最近最少使用的页面替换
                int replace = find_lru(frame);  // 选择最近最少使用的页面
                page_table[frame[replace]].flag = 0; // 页面换出
                page_table[reference[i]].flag = 1;  // 页面换入
                page_table[reference[i]].frame = replace;
                frame[replace] = reference[i];
            }
            // 输出当前帧内容
            printf("frame: ");
```

```
            for (int k = 0; k < full; k++) {
                printf("%d ", frame[k]);
            }
            printf("\n");
        } else {
            // 页面在帧中
            printf("and hit.\n");
        }
    }

    free(frame);
    return miss;
}

int main(void) {
    printf("\nChoice: 1)FIFO      2)LRU\n");
    char choice1 = getchar();
    while (getchar() != '\n');

    // 页表大小
    printf("Size of the page table: (1 <= length <= 128, default = 10) ");
    if (scanf("%d", &table_size) != 1 || table_size < 1 || table_size > 128)
{
        table_size = 10;
    }
    while (getchar() != '\n');

    // 帧大小
    printf("Size of the frame: (1 <= size <= 32, default = 4) ");
    if (scanf("%d", &frame_size) != 1 || frame_size < 1 || frame_size > 32)
{
        frame_size = 4;
    }
    while (getchar() != '\n');
    frame_size++;

    // 引用串长度
    printf("Length of the reference: ");
    if (scanf("%d", &ref_len) != 1 || ref_len < 1 || ref_len > 100) {
        ref_len = 20;
    }
    while (getchar() != '\n');

    reference = (int *)malloc(ref_len * sizeof(int));
    if (!reference) {
        printf("Malloc failed!\n");
        exit(0);
    }

    printf("Generate reference randomly? (y/n): ");
```

```
    char random_choice;
    scanf(" %c", &random_choice);
    while (getchar() != '\n');

    if (random_choice == 'y' || random_choice == 'Y') {
        generate_random_reference();
    } else {
        printf("Enter reference manually: ");
        for (int i = 0; i < ref_len; i++) {
            scanf("%d", &reference[i]);
            if (reference[i] >= table_size || reference[i] < 0) {
                printf("Invalid page number.\n");
                exit(1);
            }
        }
    }

    print_reference();
    initial_page_table();

    int miss; // 缺页次数
    switch (choice1) {
        case '1':
            printf("\nFIFO:\n");
            miss = FIFO();
            break;
        case '2':
            printf("\nLRU:\n");
            miss = LRU();
            break;

        default:
            printf("\ndefault = FIFO:\n");
            miss = FIFO();
    }

    printf("Page miss count: %d\n", miss);
    printf("Page miss ratio: %.2f%%\n", ((float)miss / ref_len) * 100.0);
    release();

    return 0;
}
```

4.1.3 实验三

```
// modify_syscall.c
#include <linux/init.h>
#include <linux/kernel.h>
```

```c
#include <linux/module.h>
#include <linux/unistd.h>
#include <linux/time.h>
#include <linux/kallsyms.h>
#include <linux/syscalls.h>
#include <linux/sched.h>
#include <asm/uaccess.h>
#include <asm/memory.h>
#include <asm/pgtable.h>

// 169: gettimeofday
#define sys_No 169

unsigned long old_sys_call_func;
unsigned long *sys_call_table = NULL;
void (*update_mapping_prot)(phys_addr_t phys, unsigned long virt,
phys_addr_t size, pgprot_t prot);

unsigned long start_rodata; //指向内核只读数据段（.rodata）
unsigned long init_begin; //指向初始化代码段的起始地址

asmlinkage int hello(const struct pt_regs *regs)
{
    // 在寄存器中读参数
    int a = regs->regs[0];
    int b = regs->regs[1];
    printk("No 169 syscall has changed to hello\n");
    return a + b;
}

// 动态获得 sys_call_name 地址
unsigned long *get_sys_call_table(void)
{
    if (!sys_call_table) {
        sys_call_table = (unsigned long
*)kallsyms_lookup_name("sys_call_table");
    }
    return sys_call_table;
}

// 读写
static inline void unprotect_memory(void)
{
    if (update_mapping_prot && start_rodata && init_begin) {
        update_mapping_prot(__pa_symbol(start_rodata),
                            (unsigned long)start_rodata,
                            init_begin - start_rodata,
                            PAGE_KERNEL);
    } else {
        printk("Failed to unprotect memory: missing symbols\n");
```

```
    }
}

// 只读
static inline void protect_memory(void)
{
    if (update_mapping_prot && start_rodata && init_begin) {
        update_mapping_prot(__pa_symbol(start_rodata),
                            (unsigned long)start_rodata,
                            init_begin - start_rodata,
                            PAGE_KERNEL_RO);
    } else {
        printk("Failed to protect memory: missing symbols\n");
    }
}

void modify_syscall(void)
{
    unsigned long *sys_call_addr;

    sys_call_table = get_sys_call_table();
    if (!sys_call_table) {
        printk(KERN_INFO "Failed to find sys_call_table\n");
        return;
    }

    sys_call_addr = &sys_call_table[sys_No];
    old_sys_call_func = sys_call_table[sys_No];

    unprotect_memory();  // 解除写保护
    *sys_call_addr = (unsigned long)&hello;
    protect_memory();  // 恢复写保护

    printk(KERN_INFO "Syscall 169 hooked successfully!\n");
}

void restore_syscall(void)
{
    unsigned long *sys_call_addr;

    sys_call_table = get_sys_call_table();
    if (!sys_call_table) {
        printk(KERN_INFO "Failed to find sys_call_table\n");
        return;
    }

    sys_call_addr = &sys_call_table[sys_No];

    unprotect_memory();
    *sys_call_addr = old_sys_call_func;
```

```
    protect_memory();

    printk(KERN_INFO "Syscallintk 169 restored successfully!\n");
}

// 通过 kallsyms_lookup_name 查找 update_mapping_prot、start_rodata 和
init_begin
static int __init mymodule_init(void)
{
    printk(KERN_INFO "Loading module...\n");

    update_mapping_prot = (void
*)kallsyms_lookup_name("update_mapping_prot");
    start_rodata = (unsigned long)kallsyms_lookup_name("__start_rodata");
    init_begin = (unsigned long)kallsyms_lookup_name("__init_begin");

    if (!update_mapping_prot || !start_rodata || !init_begin) {
        return -EINVAL;
    }

    modify_syscall();
    return 0;
}

static void __exit mymodule_exit(void)
{
    printk(KERN_INFO "Unloading module...\n");
    restore_syscall();
}

module_init(mymodule_init);
module_exit(mymodule_exit);
MODULE_LICENSE("GPL");
```

```
// modify_new_syscall.c
#include<stdio.h>
#include<sys/time.h>
#include<unistd.h>
int main()
{
    int ret=syscall(169,10,20); //after modify syscall 169
    printf("%d\n",ret);
    return 0;
}
```

```
// modify_old_syscall.c
#include<stdio.h>
```

```c
#include<sys/time.h>
#include<unistd.h>
int main()
{
    struct timeval tv;
    syscall(169,&tv,NULL); //before modify syscall 169 :gettimeofday
    printf("tv_sec:%d\n",tv.tv_sec);
    printf("tv_usec:%d\n",tv.tv_usec);
    return 0;
}
```

```c
// driver.c
#include <linux/kernel.h>
#include <linux/module.h>
#include <linux/fs.h>
#include <linux/uaccess.h>
#include <linux/init.h>
#include <linux/slab.h>
#include <linux/semaphore.h>
#include <linux/sched.h>
#include <linux/wait.h>
#include <linux/device.h>

#define MAJOR_NUM 290 //主设备号
#define DEVICE_NAME "globalvar"
#define MAX_CLIENTS 10 //最大客户端数量：10
#define MAX_MESSAGES 10 //最大消息数量：10
#define MESSAGE_SIZE 256 //最大消息长度：256


// 消息结构体
struct message {
    char content[MESSAGE_SIZE];
    size_t length;
};

// 客户端结构体
struct client_data {
    pid_t pid;
    struct message message_queue[MAX_MESSAGES]; //每个客户端的消息队列
    int message_count;                  //当前收到的消息数量
};

// 设备数据结构体
struct chat_device_data {
    struct client_data *clients[MAX_CLIENTS];
    struct semaphore read_sem;
    struct semaphore write_sem;
```

```
    int readers_count;
    wait_queue_head_t read_queue;
    wait_queue_head_t write_queue;
    int client_count;                           //当前连接的客户端个数
};

static struct chat_device_data chat_device = {
    .write_sem = __SEMAPHORE_INITIALIZER(chat_device.write_sem, 1),
    .read_sem = __SEMAPHORE_INITIALIZER(chat_device.read_sem, 1),
    .client_count = 0,
    .readers_count = 0,
};

static int globalvar_open(struct inode *, struct file *);
static int globalvar_release(struct inode *, struct file *);
static ssize_t globalvar_read(struct file *, char *, size_t, loff_t *);
static ssize_t globalvar_write(struct file *, const char *, size_t, loff_t
*);
static struct client_data *get_client_data(pid_t pid);
static void send_message_to_all(struct client_data *sender, const char
*message);
static void send_message_to_pid(pid_t target_pid, struct client_data
*sender, const char *message);

// 文件操作
static struct file_operations chat_fops = {
    .open = globalvar_open,
    .release = globalvar_release,
    .read = globalvar_read,
    .write = globalvar_write,
};

//设备类和节点
static struct class *chat_class = NULL;
static struct device *chat_device_node = NULL;

//找到当前 pid 对应的客户端，若找不到则重新创建
static struct client_data *get_client_data(pid_t pid) {
    int i = 0;
    for (i = 0; i < MAX_CLIENTS; i++) {
        if (chat_device.clients[i] && chat_device.clients[i]->pid == pid) {
            return chat_device.clients[i];
        }
    }

    // 如果不存在，则重新创建
    if (chat_device.client_count < MAX_CLIENTS) {
        for (i = 0; i < MAX_CLIENTS; i++) {
            if (!chat_device.clients[i]) {
```

```
                struct client_data *new_client =
kzalloc(sizeof(*new_client), GFP_KERNEL);
                if (!new_client) return NULL;
                new_client->pid = pid;
                chat_device.clients[i] = new_client;
                chat_device.client_count++;
                printk(KERN_INFO "globalvar: Client with PID %d
registered.\n", pid);
                return new_client;
            }
        }
    }

    return NULL;
}

// 打开设备
static int globalvar_open(struct inode *inode, struct file *file) {
    pid_t pid = current->pid;
    struct client_data *client;

    if (down_interruptible(&chat_device.read_sem)) {
        return -ERESTARTSYS;
    }

    //查找当前 pid 对应的客户端
    client = get_client_data(pid);
    if (!client) {
        up(&chat_device.read_sem);
        return -ENOMEM;
    }

    file->private_data = client;
    up(&chat_device.read_sem);
    return 0;
}

// 释放设备
static int globalvar_release(struct inode *inode, struct file *file) {
    pid_t pid = current->pid;
    int i = 0;

    if (down_interruptible(&chat_device.read_sem)) {
        return -ERESTARTSYS;
    }

    //释放当前 pid 对应的客户端内存
    for (i = 0; i < MAX_CLIENTS; i++) {
        if (chat_device.clients[i] && chat_device.clients[i]->pid == pid) {
            kfree(chat_device.clients[i]);
```

```
            chat_device.clients[i] = NULL;
            chat_device.client_count--;
            printk(KERN_INFO "globalvar: Client with PID %d
disconnected.\n", pid);
            break;
        }
    }

    up(&chat_device.read_sem);
    return 0;
}

// 群发：发送消息给所有客户端（忽略发送者自身）
static void send_message_to_all(struct client_data *sender, const char
*message) {
    int i = 0;
    for (i = 0; i < MAX_CLIENTS; i++) {
        struct client_data *client = chat_device.clients[i];
        if (client && client != sender) { //不能给自己发送消息
            if (client->message_count < MAX_MESSAGES) { //消息队列未满
                struct message *msg =
&client->message_queue[client->message_count++];
                strncpy(msg->content, message, MESSAGE_SIZE - 1); //写入当前
pid 的消息队列
                msg->content[MESSAGE_SIZE - 1] = '\0';
                msg->length = strnlen(msg->content, MESSAGE_SIZE);
                printk(KERN_INFO "globalvar: Sent message to client
PID %d.\n", client->pid);
            } else {
                printk(KERN_ALERT "globalvar: Message queue for client
PID %d is full.\n", client->pid);
            }
        }
    }
}

// 私发：发送消息给指定 pid 的客户端
static void send_message_to_pid(pid_t target_pid, struct client_data
*sender, const char *message) {
    int i;

    for (i = 0; i < MAX_CLIENTS; i++) {
        struct client_data *client = chat_device.clients[i];
        if (client && client->pid == target_pid) { // 找到目标 pid
            if (client->message_count < MAX_MESSAGES) {
                struct message *msg =
&client->message_queue[client->message_count++];
                strncpy(msg->content, message, MESSAGE_SIZE - 1); //写入当前
pid 的消息队列
                msg->content[MESSAGE_SIZE - 1] = '\0';
```

```
                msg->length = strnlen(msg->content, MESSAGE_SIZE);
                printk(KERN_INFO "globalvar: Sent message to client
PID %d.\n", target_pid);
            } else {
                printk(KERN_ALERT "globalvar: Message queue for PID %d is
full.\n", target_pid);
            }
            return;
        }
    }

    printk(KERN_ALERT "globalvar: No client found with PID %d.\n",
target_pid);
}

// 读取消息
static ssize_t globalvar_read(struct file *filp, char *buf, size_t len,
loff_t *off) {
    struct client_data *client = filp->private_data; //客户端数据结构
    ssize_t bytes_read = 0;

    if (down_interruptible(&chat_device.read_sem)) {
        return -ERESTARTSYS;
    }

    // wait_event_interruptible(chat_device.write_queue,
chat_device.readers_count == 0); //没有进程正在读
    chat_device.readers_count++; //开始读，读进程数量+1

    //检查是否有未读消息
    if (client->message_count > 0) {
        struct message *msg = &client->message_queue[0];
        size_t to_copy = min(len, msg->length);

        //拷贝到用户空间
        if (copy_to_user(buf, msg->content, to_copy)) {
            chat_device.readers_count--;
            up(&chat_device.read_sem);
            return -EFAULT;
        }

        bytes_read = to_copy;

        //拷贝结束之后，从消息队列里删除
        memmove(&client->message_queue[0], &client->message_queue[1],
                sizeof(struct message) * (client->message_count - 1)); //前
移未读消息
        client->message_count--;
        printk(KERN_INFO "globalvar: Read %zd bytes for client PID %d.\n",
bytes_read, client->pid);
```

```
    }

    chat_device.readers_count--; //结束，读进程数量-1
    if (chat_device.readers_count == 0)
        wake_up_interruptible(&chat_device.write_queue); //没有进程在读，则允
许写操作

    up(&chat_device.read_sem);
    return bytes_read;
}

//写入消息
static ssize_t globalvar_write(struct file *filp, const char *buf, size_t
len, loff_t *off) {
    struct client_data *sender = filp->private_data;
    char message[MESSAGE_SIZE];
    char *msg_content;
    char *sep;
    pid_t sender_pid;
    pid_t target_pid;
    int flag = 0; //标志是否为私聊

    // 检查消息长度
    if (len >= MESSAGE_SIZE) {
        return -EINVAL; // 消息太长
    }

    // 从用户空间拷贝 message
    if (copy_from_user(message, buf, len)) {
        return -EFAULT;
    }
    message[len] = '\0'; // 确保消息以 NULL 结尾

    // 获取信号量
    if (down_interruptible(&chat_device.write_sem)) {
        return -ERESTARTSYS;
    }
    wait_event_interruptible(chat_device.read_queue,
chat_device.readers_count == 0); //保证没有进程在读

    // 查找第一个分隔符 '|'
    sep = strchr(message, '|');
    if (!sep) {
        up(&chat_device.write_sem);
        return -EINVAL; // 缺少第一个分隔符
    }

    *sep = '\0'; // 将第一个分隔符替换为字符串结束符
    //提取发送者 pid
    if (kstrtoint(message, 10, &sender_pid) < 0) {
```

```
        up(&chat_device.write_sem);
        return -EINVAL; // 无效的 PID
    }

    //用@判断是否私聊
    msg_content = sep + 1;
    if (msg_content[0] == '@'){
        flag = 1;
        char *pid_start = msg_content + 1;
        char *space_pos = strchr(pid_start, ' ');
        if (!space_pos) {
            up(&chat_device.write_sem);
            return -EINVAL; // 缺少消息内容
        }

        *space_pos = '\0'; // 将空格替换为字符串结束符
        if (kstrtoint(pid_start, 10, &target_pid) < 0) { //提取目标 pid
            up(&chat_device.write_sem);
            return -EINVAL; // 无效的目标 PID
        }
        msg_content = space_pos + 1; //获取消息内容

    }

    char full_message[MESSAGE_SIZE];
    //重建消息内容
    if (flag) {
        snprintf(full_message, sizeof(full_message), "%d|@%d %s",
sender_pid, target_pid, msg_content);
        send_message_to_pid(target_pid, sender, full_message);
    } else {
        snprintf(full_message, sizeof(full_message), "%d|%s", sender_pid,
msg_content);
        send_message_to_all(sender, full_message);
    }

    // 释放信号量
    wake_up_interruptible(&chat_device.read_queue); //已经写完，可以开始读
    up(&chat_device.write_sem);
    return len; // 返回写入的字节数
}

// 初始化模块
static int __init globalvar_init(void) {
    int ret;

    //设备号使用 MAJOR_NUM 不重新申请设备号
    ret = register_chrdev(MAJOR_NUM, DEVICE_NAME, &chat_fops);
    if (ret < 0) {
        printk(KERN_ALERT "globalvar: Registration failed.\n");
```

```
        return ret;
    }

    //初始化信号量和队列
    sema_init(&chat_device.read_sem, 1);
    sema_init(&chat_device.write_sem, 1);
    init_waitqueue_head(&chat_device.read_queue);
    init_waitqueue_head(&chat_device.write_queue);

    //注册设备类
    chat_class = class_create(THIS_MODULE, "globalvar_class");
    if (IS_ERR(chat_class)) { //创建失败
        unregister_chrdev(MAJOR_NUM, DEVICE_NAME);
        return PTR_ERR(chat_class);
    }

    //注册设备节点
    chat_device_node = device_create(chat_class, NULL, MKDEV(MAJOR_NUM, 0),
NULL, DEVICE_NAME);
    if (IS_ERR(chat_device_node)) { //创建失败
        class_destroy(chat_class);
        unregister_chrdev(MAJOR_NUM, DEVICE_NAME);
        return PTR_ERR(chat_device_node);
    }

    printk(KERN_INFO "globalvar: Registered with major number %d.\n",
MAJOR_NUM);
    return 0;
}

// 卸载模块
static void __exit globalvar_exit(void) {
    int i;

    //释放每个客户端的内存
    for (i = 0; i < MAX_CLIENTS; i++) {
        if (chat_device.clients[i]) {
            kfree(chat_device.clients[i]);
        }
    }

    //销毁设备节点和类
    device_destroy(chat_class, MKDEV(MAJOR_NUM, 0));
    class_destroy(chat_class);

    //注销字符设备
    unregister_chrdev(MAJOR_NUM, DEVICE_NAME);

    printk(KERN_INFO "globalvar: Unregistered device.\n");
}
```

```
module_init(globalvar_init);
module_exit(globalvar_exit);
MODULE_LICENSE("GPL");
```

```
// test.c
#include <sys/types.h>
#include <unistd.h>
#include <sys/stat.h>
#include <stdio.h>
#include <fcntl.h>
#include <string.h>
#include <stdlib.h>
#include <signal.h>
#include <sys/wait.h>
#include <pthread.h>

int fd;
#define DEVICE_NAME "/dev/globalvar"
#define BUFFER_SIZE 256

typedef struct {
    int fd;
    int running;
} thread_args;

// 线程函数：接收消息
void *receive_message(void *arg) {
    thread_args *args = (thread_args *)arg;
    int fd = args->fd;
    char recv_message[BUFFER_SIZE];

    while (args->running) {
        ssize_t bytes_read = read(fd, recv_message, sizeof(recv_message) -
1);
        if (bytes_read > 0) {
            recv_message[bytes_read] = '\0'; // 确保消息以 '\0' 结尾

            // printf("Orient message:%s\n", recv_message);

            // 解析消息
            pid_t sender_pid;
            char *msg_content = strchr(recv_message, '|');
            if (msg_content) {
                *msg_content = '\0'; // 分离发送者 PID 和消息内容

                sender_pid = atoi(recv_message); // 获取 sender_pid
                msg_content++; // 跳过 ':' 符号，指向消息内容
```

```
                // 检查消息是否属于当前进程
                if (sender_pid != getpid()) {
                    printf("PID %d received message from PID %d: %s\n",
getpid(), sender_pid, msg_content);
                }
            }
        } else if (bytes_read == 0) {
            usleep(100000); // 延时等待新消息
        } else {
            perror("Failed to read message");
        }
    }

    return NULL;
}

// 线程函数：发送消息
void *send_message(void *arg) {
    thread_args *args = (thread_args *)arg;
    int fd = args->fd;
    char send_message[BUFFER_SIZE];
    char full_message[BUFFER_SIZE];

    while (args->running) {
        // 输入消息
        printf("Enter message to send (or type 'quit' to exit):\n");
        fgets(send_message, sizeof(send_message), stdin);
        send_message[strcspn(send_message, "\n")] = '\0'; // 去除换行符

        if (strcmp(send_message, "quit") == 0) {
            args->running = 0;  // 退出条件
            break;
        }

        // 格式化消息
        snprintf(full_message, sizeof(full_message), "%d|%s", getpid(),
send_message);

        // 发送消息
        if (write(fd, full_message, strlen(full_message)) < 0) {
            perror("Failed to write message");
        } else {
            printf("Message sented.\n");
        }
    }

    return NULL;
}
```

```c
int main() {
    int fd = open(DEVICE_NAME, O_RDWR, S_IRUSR | S_IWUSR);
    if (fd == -1) {
        perror("Failed to open device");
        return -1;
    }

    printf("My PID is %d.\n",getpid());

    pthread_t receive_thread, send_thread;
    thread_args args = {fd, 1}; // 传递给线程的参数

    // 接收消息线程
    if (pthread_create(&receive_thread, NULL, receive_message, (void
*)&args) != 0) {
        perror("Failed to create receive thread");
        close(fd);
        return -1;
    }

    // 发送消息线程
    if (pthread_create(&send_thread, NULL, send_message, (void *)&args) !=
0) {
        perror("Failed to create send thread");
        close(fd);
        return -1;
    }

    // 等待两个线程结束
    pthread_join(receive_thread, NULL);
    pthread_join(send_thread, NULL);

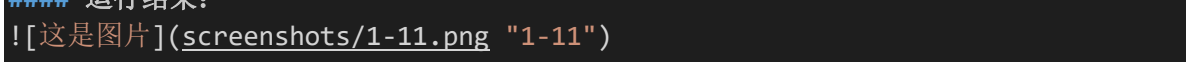
    close(fd);
    return 0;
}
```

4.2 Readme

4.2.1 实验一

```
## 1.1 进程相关编程实验
### 步骤 1
#### 代码：
```c
#include <sys/types.h>
#include <sys/wait.h>
#include <stdio.h>
#include <unistd.h>
```
```

```c
int main(){
    pid_t pid, pid1;

    pid = fork();

    if (pid < 0){
        fprintf (stderr, "Fork Failed");
        return 1;
    }

    else if (pid == 0){
        pid1 = getpid();
        printf("child: pid = %d  ", pid);
        printf("child: pid1 = %d  ", pid1);
    }

    else{
        pid1 = getpid();
        printf("parent: pid = %d  ", pid);
        printf("parent: pid1 = %d  ", pid1);
        wait(NULL);
    }

    return 0;
}
```

运行结果：


结果分析：
调用`fork()`创建一个子进程。

父进程中，`pid`返回子进程的进程号，`pid1`为父进程的进程号；子进程中，`pid`返回 0，`pid1`返回子进程本身的进程号。

调用了`wait()`，父进程会等子进程结束后再结束。

（`wait()`运行时父进程已经打印了输出结果，所以其实父子进程间的打印顺序应该是随机的，但是我运行的时候并没有跑出父进程先于子进程打印的输出结果，猜测可能是因为子进程代码少，所以执行速度更快。）

步骤 2
代码：
```c
#include <sys/types.h>
#include <sys/wait.h>
#include <stdio.h>
#include <unistd.h>
```

```c
int main(){
    pid_t pid, pid1;

    pid = fork();

    if (pid < 0){
        fprintf (stderr, "Fork Failed");
        return 1;
    }

    else if (pid == 0){
        pid1 = getpid();
        printf("child: pid = %d  ", pid);
        printf("child: pid1 = %d  ", pid1);
    }

    else{
        pid1 = getpid();
        printf("parent: pid = %d  ", pid);
        printf("parent: pid1 = %d  ", pid1);
        //wait(NULL);
    }

    return 0;
}
```

运行结果：

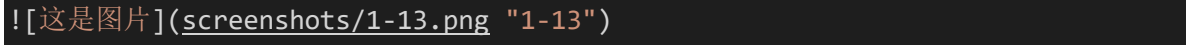

结果分析：

`wait()`会暂时停止目前进程的执行，直到有信号来到或子进程结束。

删去`wait()`函数后，父进程不会等待子进程结束，它将会和子进程同时运行。因此，父进程和子进程的打印顺序是不确定的。而且如果父进程先于子进程结束，子进程会变成孤儿进程，此时子进程将立刻结束。

步骤 3
代码：
```c
#include <sys/types.h>
#include <sys/wait.h>
#include <stdio.h>
#include <unistd.h>

int value = 0;
int* p = &value;
```

```c
int main(){
    pid_t pid, pid1;

    pid = fork();

    if (pid < 0){
        fprintf (stderr, "Fork Failed");
        return 1;
    }

    else if (pid == 0){
        pid1 = getpid();
        value = 1;
        //printf("child: pid = %d  ", pid);
        //printf("child: pid1 = %d  ", pid1);
        printf("child: global value = %d  ", value);
        printf("child: *value = %p  ", p);
    }

    else{
        pid1 = getpid();
        value = 3;
        //printf("parent: pid = %d  ", pid);
        //printf("parent: pid1 = %d  ", pid1);
        printf("parent: global value = %d  ", value);
        printf("parent: *value = %p  ", p);
        wait(NULL);
    }

    return 0;
}
```

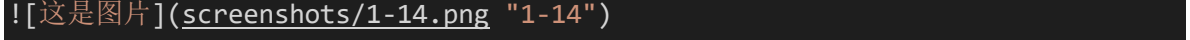
运行结果：


结果分析：
增加全局变量`value`，在父进程中修改`value`值为 3，子进程中修改`value`值为 1，分别打印`value`的地址与值。观察到，父子进程打印的值不同，但地址是相同的。因为`fork()`后子进程和父进程的数据空间是独立的，不同进程中`value`的修改互不影响。但在父进程和子进程中，指针 p 所指的地址是一样的，因为`fork()`会复制整个地址空间。

步骤 4
代码：
```c
#include <sys/types.h>
#include <sys/wait.h>
#include <stdio.h>
#include <unistd.h>
```

```
int value = 0;
int* p = &value;

int main(){
    pid_t pid, pid1;

    pid = fork();

    if (pid < 0){
        fprintf (stderr, "Fork Failed");
        return 1;
    }

    else if (pid == 0){
        pid1 = getpid();
        value = 1;
        printf("child: value = %d  ", value);
        printf("child: *value = %p  \n", p);
    }

    else{
        pid1 = getpid();
        value = 3;
        printf("parent: value = %d  ", value);
        printf("parent: *value = %p  \n", p);
        wait(NULL);
    }

    // value++
    value += 1;
    printf("before return: value = %d  ", value);
    printf("before return: *value = %p  \n", p);

    return 0;
}
```

运行结果：



结果分析：

在`1-13`的基础上，在 return 前对全局变量`value`加 1。父子进程都会执行这个操作，而且由于`fork()`后子进程和父进程的数据空间是独立的，所以不同的加 1 操作互不影响。子进程的`value`从 1 增加到 2，父进程的 `value`从 3 增加到 4。

因为调用了`wait()`函数，父进程将等待子进程结束后再继续运行，因此`before return: value = 4`语句将最后输出。父进程和子进程间执行加 1 操作的顺序是随机的，但进程内部需要先进行赋值，再进行加 1 操作。因此，进程内部的打印顺序是确定的。

步骤 5

代码：
调用`system`函数：
```c
#include <sys/types.h>
#include <sys/wait.h>
#include <stdio.h>
#include <unistd.h>
#include <stdlib.h>

int main(){
    pid_t pid, pid1;

    pid = fork();

    if (pid < 0){
        fprintf (stderr, "Fork Failed");
        return 1;
    }

    else if (pid == 0){
        pid1 = getpid();
        printf("child PID = %d\n", pid1);

        // 使用 system() 调用外部命令执行 system_call 程序
        system("./system_call");
        printf("child PID = %d\n", pid1);
    }

    else{
        pid1 = getpid();
        printf("parent PID = %d\n", pid1);
        wait(NULL);
    }

    return 0;
}
```

调用`exec`族函数：
```c
#include <sys/types.h>
#include <sys/wait.h>
#include <stdio.h>
#include <unistd.h>
#include <stdlib.h>
```

```c
int main(){
    pid_t pid, pid1;

    pid = fork();

    if (pid < 0){
        fprintf (stderr, "Fork Failed");
        return 1;
    }

    else if (pid == 0){
        pid1 = getpid();
        printf("child PID = %d\n", pid1);

        // 使用 exec() 系列函数调用 system_call 程序
        char *args[] = {"./system_call", NULL};
        execvp(args[0], args);
        printf("child PID = %d\n", pid1);
    }

    else{
        pid1 = getpid();
        printf("parent PID = %d\n", pid1);
        wait(NULL);
    }

    return 0;
}
```

运行结果：
调用`system`函数：

调用`exec`族函数：

结果分析：
在子进程内部分别使用`system()`和`execvp()`函数调用`system_call.c`输出进程号，与直接调用`getpid()`获得的进程号对比。观察到`execvp()`输出的进程号是子进程的`pid`，而`system()`输出的进程号并非子进程的`pid`。同时，调用`execvp()`函数后，进程不会继续执行接下来的输出。

这是因为`system()`会新创建一个子进程来执行指定的命令，其`pid`与原先的进程不同；而`execvp()`是一个进程替换函数，它会替换当前进程的映像，并用新程序的映像来覆盖现有的进程，所以`execvp()`输出的进程号与原先的进程相同。

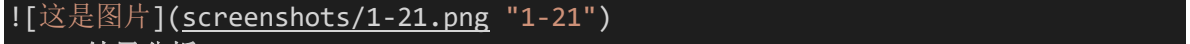
1.2 线程相关编程实验

步骤 1

代码：

```c
#include <stdio.h>
#include <pthread.h>
#include <unistd.h>
#include <sys/types.h>
#include <sys/syscall.h>
#include <sys/wait.h>
#include <stdlib.h>

void* pthread_1(void* arg) {
    printf("thread1 tid = %d, pid = %ld\n", syscall(SYS_gettid), getpid());
    system("./system_call");
    printf("thread1 systemcall return\n");
    return NULL;
}

void* pthread_2(void* arg) {
    printf("thread2 tid = %d, pid = %ld\n", syscall(SYS_gettid), getpid());
    system("./system_call");
    printf("thread2 systemcall return\n");
    return NULL;
}

int main() {
    pthread_t thread1, thread2;

    pthread_create(&thread1, NULL, pthread_1, NULL);
    if (thread1)
        printf("thread1 create success!\n");
    else return 1;

    pthread_create(&thread2, NULL, pthread_2, NULL);
    if (thread2)
        printf("thread2 create success!\n");
    else return 1;

    pthread_join(thread1, NULL);
    pthread_join(thread2, NULL);

    return 0;
}
```

运行结果：



结果分析：

创建两个子线程，在不设置线程锁和信号量的情况下，分别对全局变量`shared_var`进行 100000 次+100 和-100 操作。在没有竞争的理想情况下，预计输出结果为 0，但实际输出非

0，这是因为两个线程间可能并发访问相同的内存，比如全局变量`shared_var`，从而造成数据竞争。

遇到的问题：gcc 编译的时候没加`-lpthread`，导致没链接 pthread 库。

步骤 2
代码：
```c
#include <stdio.h>
#include <pthread.h>
#include <semaphore.h>

int shared_var = 0;
sem_t signal;

// +100
void* increment(void* arg) {
    for (int i = 0; i < 100000; i++) {
        sem_wait(&signal);     // P
        shared_var += 100;
        sem_post(&signal);     // V
    }
    return NULL;
}

// -100
void* decrement(void* arg) {
    for (int i = 0; i < 100000; i++) {
        sem_wait(&signal);     // P
        shared_var -= 100;
        sem_post(&signal);     // V
    }
    return NULL;
}

int main() {
    pthread_t thread1, thread2;

    sem_init(&signal, 0, 1);
    pthread_create(&thread1, NULL, increment, NULL);
    if (thread1)
        printf("thread1 create success!\n");
    else return 1;

    pthread_create(&thread2, NULL, decrement, NULL);
    if (thread2)
        printf("thread2 create success!\n");
    else return 1;

    pthread_join(thread1, NULL);
```

```
    pthread_join(thread2, NULL);

    printf("variable result: %d\n", shared_var);

    sem_destroy(&signal);

    return 0;
}
```

运行结果：


结果分析：
使用信号量来保证两个线程不会在没有协调的情况下同时访问和修改`shared_var`。对于线程`increment`，每次循环开始，首先执行 P 操作，如果 signal 值为 1，则 P 操作成功，线程进入临界区，并将信号量值减至 0。
临界区操作完成后，V 操作将信号量的值加回 1，释放临界区。线程`decrement`与`increment`线程类似。

信号量保证了每次只有一个线程可以访问和修改全局变量`shared_var`，从而避免了数据竞争。

步骤 3
代码：
调用`system`函数：
```c
#include <stdio.h>
#include <pthread.h>
#include <unistd.h>
#include <sys/types.h>
#include <sys/syscall.h>
#include <sys/wait.h>
#include <stdlib.h>

void* pthread_1(void* arg) {
    printf("thread1 tid = %d, pid = %ld\n", syscall(SYS_gettid), getpid());
    system("./system_call");
    printf("thread1 systemcall return\n");
    return NULL;
}

void* pthread_2(void* arg) {
    printf("thread2 tid = %d, pid = %ld\n", syscall(SYS_gettid), getpid());
    system("./system_call");
    printf("thread2 systemcall return\n");
    return NULL;
}

int main() {
    pthread_t thread1, thread2;
```

```
    pthread_create(&thread1, NULL, pthread_1, NULL);
    if (thread1)
        printf("thread1 create success!\n");
    else return 1;

    pthread_create(&thread2, NULL, pthread_2, NULL);
    if (thread2)
        printf("thread2 create success!\n");
    else return 1;

    pthread_join(thread1, NULL);
    pthread_join(thread2, NULL);

    return 0;
}
```

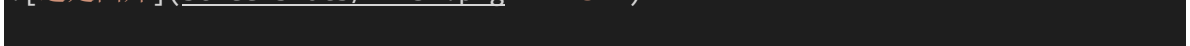
调用`exec`族函数：

```c
#include <stdio.h>
#include <pthread.h>
#include <unistd.h>
#include <sys/types.h>
#include <sys/syscall.h>
#include <sys/wait.h>
#include <stdlib.h>

void* pthread_1(void* arg) {
    printf("thread1 tid = %d, pid = %ld\n", syscall(SYS_gettid), getpid());
    char *args[] = {"./system_call", NULL};
    execvp(args[0], args);
    printf("thread1 systemcall return\n");
    return NULL;
}

void* pthread_2(void* arg) {
    printf("thread2 tid = %d, pid = %ld\n", syscall(SYS_gettid), getpid());
    char *args[] = {"./system_call", NULL};
    execvp(args[0], args);
    printf("thread2 systemcall return\n");
    return NULL;
}

int main() {
    pthread_t thread1, thread2;

    pthread_create(&thread1, NULL, pthread_1, NULL);
    if (thread1)
        printf("thread1 create success!\n");
    else return 1;
```

```
    pthread_create(&thread2, NULL, pthread_2, NULL);
    if (thread2)
        printf("thread2 create success!\n");
    else return 1;

    pthread_join(thread1, NULL);
    pthread_join(thread2, NULL);

    return 0;
}
```

运行结果与分析：


两个线程的 TID 不同，因为每个线程都有自己的 TID；PID 相同，因为两个线程属于同一个进程。调用`system()`函数输出进程的 PID，发现 PID 不同，因为`system()`会新创建一个子进程来执行指定的命令。



调用`execvp()`时，观察到输出不完全。因为调用 `execvp()` 会导致当前线程的代码空间被替换，主进程剩余的代码不会被继续执行。所以，只要有一个线程开始执行`execvp()`，新进程将会覆盖原先的进程，另一个线程也不会继续执行原先的代码。

1.3 自旋锁实验
代码：
```c
#include <stdio.h>
#include <pthread.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
// 线程 1: value++
void *thread1_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;
}
// 线程 2: value--
void *thread2_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;

    printf("Shared value: %d\n", shared_value);

    // 初始化自旋锁
    spinlock_init(&lock);

    // 创建两个线程
    pthread_create(&thread1, NULL, thread1_function, &lock);
    if (thread1)
        printf("thread1 create success!\n");
    else return 1;
```

```
    pthread_create(&thread2, NULL, thread2_function, &lock);
    if (thread2)
        printf("thread2 create success!\n");
    else return 1;

    // 等待线程结束
    pthread_join(thread1, NULL);
    pthread_join(thread2, NULL);

    // 输出共享变量的值
    printf("Shared value: %d\n", shared_value);
    return 0;
}
```

运行结果：

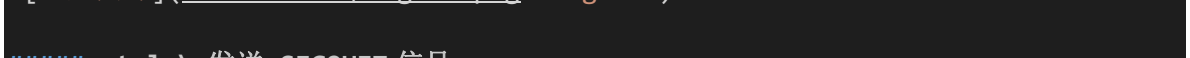
结果分析：

自旋锁的设计是为了防止多个线程在同时访问`shared_value`时发生数据竞争。如果没有自旋锁，两个线程可能会同时读取 `shared_value`，并各自进行+1 或-1 操作，而这时会造成数据冲突，从而导致最终结果错误。

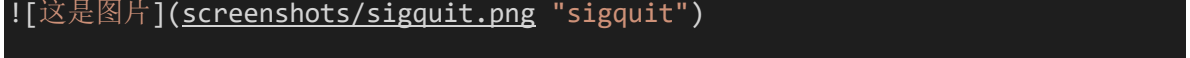
增加自旋锁后，每次只能有一个线程访问共享资源，因此两个线程在操作共享变量时不会发生数据竞争，从而正确地修改 `shared_value`。

4.2.2 实验二

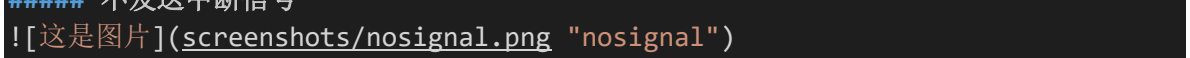
```
### 2.1 进程的软中断通信
#### 运行截图
##### ctrl+c 发送 SIGINT 信号
```



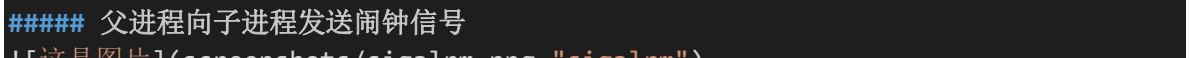
```
##### ctrl+\ 发送 SIGQUIT 信号
```



```
##### 不发送中断信号
```



```
##### 父进程向子进程发送闹钟信号
```



```
#### 结果分析
##### 在接收不同中断前后的差别
SIGINT 或 SINQUIT 信号未触发时，子进程处于`pause`，父进程等待中断信号或超时信号。中断信号触发后，父进程向两个子进程分别发送信号 SIGSTKFLT 和 SIGCHLD 子进程从`pause`中唤醒，执行信号处理程序，完成资源清理并退出。

##### 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？
```

1. 父进程向子进程发送 SIGALRM 信号
    将父进程通过 SIGUSR1 和 SIGUSR2 通知子进程的逻辑替换为使用闹钟信号(SIGALRM)。子进程通过 SIGALRM 来接受父进程的中断通知。收到中断信号时，父进程通过 `kill(pid, SIGALRM)` 通知子进程退出。
    修改前，子进程能明确知道收到的中断信号是 SIGUSR1（父进程发给子进程 1）还是 SIGUSR2（父进程发给子进程 2）。修改后，子进程无法区分具体来自父进程的哪个通知，只能统一处理 SIGALRM。

2. 父进程被 SIGALRM 信号中断
    父进程收到 SIGALRM 信号后，子进程直接被 SIGKILL 强制终止 ；父进程收到 SIGINT/SIGQUIT 信号后，子进程通过 SIGUSR1/SIGUSR2 通知退出。

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？
`kill()` 命令被使用了两次。
1. 接收到中断信号
    父进程在捕获 SIGINT 或 SIGQUIT 信号时，发送 SIGSTKFL 信号给 pid1 对应的子进程 1；发送 SIGCHLD 信号给 pid2 对应的子进程 2。
    子进程 1 捕获到 SIGSTKFL 信号后，执行 `handle_usr1()` 函数，打印 16 stop test；子进程 2 捕获到 SIGCHLD 信号后，执行 handle_usr2() 函数，打印 17 stop test。

2. 超时，父进程强制结束子进程
    当超时发生（5 秒后，父进程收到 SIGALRM ），发送 SIGSTKFL 信号给 pid1 对应的子进程 1；发送 SIGCHLD 信号给 pid2 对应的子进程 2。

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？
父进程可以通过`kill(pid, SIGTERM)`或`kill(pid, SIGKILL)`信号强制结束子进程。SIGTERM 用于请求进程正常退出，进程可以捕获此信号并执行必要的清理操作（如关闭文件、释放内存等），然后自行终止。SIGKILL 用于强制终止进程，无条件杀死进程，操作系统会立即清理资源，强制释放进程占用的内存，该进程自身无法执行退出时的清理工作。

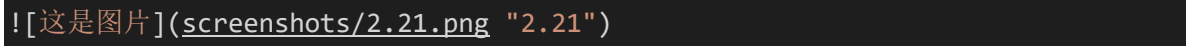
进程可以调用`exit()`函数主动终止运行，并释放所有占用的资源，并向操作系统返回状态码。

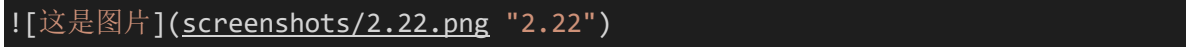
一般来说，主动退出能可清理资源（文件、内存等）并记录日志，更适合大多数情况；但当进程失去响应（如进入死循环、死锁）或需要强制终止时，在外部杀死进程更适合。

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？
子进程完成初始化后主动通知父进程，比如通过 pipe、signal 或共享内存（设置标志位，父进程轮询）等方式。

如何阻塞住子进程，让子进程等待父进程发来信号？
使用`pause()`阻塞子进程，直到父进程发来信号。
`pause()` 是一个阻塞函数，用于让进程挂起，直到捕获一个信号，并且存在该信号的处理程序。

2.2 进程的管道通信
运行截图

有锁


无锁


结果分析
有锁情况下：
`lockf(fd[1], 1, 0)`：
锁住写端，确保当前子进程独占管道的写操作。
在锁释放之前，其他子进程无法对管道写端执行写操作。

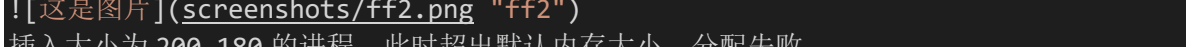
`lockf(fd[1], 0, 0)`：
解锁写端，允许其他子进程写入。

子进程 1 启动并加锁后，写入字符 '1'，直到写完 2000 个字符并解锁。子进程 2 等待子进程 1 释放锁后才加锁并写入字符 '2'。锁机制确保了同一时刻只有一个子进程在写入管道。

无锁情况下；
两个子进程不再使用 `lockf()`，写入管道的顺序完全依赖操作系统的调度机制。子进程可以在任何时间点进行写入，导致两者之间可能发生竞争。
可能出现两个子进程轮流执行，写入字符交替出现。

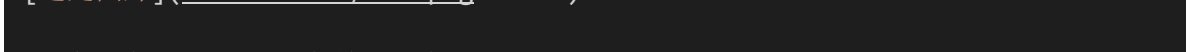
实验中管道通信是怎样实现同步与互斥的？如果不控制同步与互斥会发生什么后果？
同步：
如果父进程早于子进程读取管道：管道中的数据可能尚未完全写入，导致父进程读取到的数据不完整。
父进程使用 `wait(0)` 等待子进程结束，从而确保父进程在所有子进程完成写入后再进行管道的读取操作。
互斥：
如果不控制互斥，子进程之间可能发生竞争，导致写入数据混乱（比如，`write()` 的内容被覆盖或交错）。
使用 `lockf(fd[1], 1, 0)` 锁定管道的写端，确保某个子进程在写入时独占管道的写权限。
`lockf(fd[1], 0, 0)` 解锁，释放写权限，允许其他子进程进行写入。

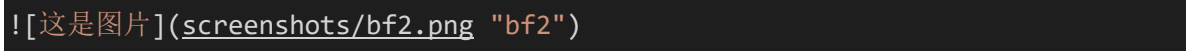
2.3 内存的分配与回收
运行截图
FF：
顺序插入大小 200 300 400 的进程

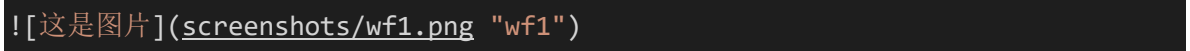
删除 200，插入 150，此时新进程的内存起始位置为 0

插入大小为 200 180 的进程，此时超出默认内存大小，分配失败

插入大小为 170 的进程，此时采用内存紧缩技术，将全部空闲块合并在一起


BF：

顺序插入大小 200 300 400 的进程。删除 200，此时有两个大小分别为 200、124 的空闲块。插入 100，此时选择第二个空闲块，新进程的内存起始位置为 900。


插入大小为 20 的进程，起始位置为 1000。


WF:
顺序插入大小为 100 200 300 的进程。删除 200，此时有两个大小分别为 200、424 的空闲块。插入大小为 150 的进程，此时选择后一个个空闲块，新进程的内存起始位置为 600。


插入大小为 100 的进程，起始位置为 750。


结果分析
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。
FF
优先从空闲块链表的开头查找第一个可以满足请求的空闲块。若找到适合的块，则分配请求大小的内存，并更新该空闲块的剩余大小。

优点：
+ 查找速度较快：由于从头开始查找，找到第一个适配块即可返回，无需遍历完整链表。
+ 实现简单：只需维护一个单向链表，直接按顺序搜索即可。

缺点：
+ 容易产生外部碎片：由于每次分配都选择靠前的小块，内存会逐渐形成许多零散的小碎片。
+ 随着时间推移，前半部分的内存会被频繁访问，可能导致性能下降。

如何提高性能：
+ 记录上次分配位置（Next Fit）：在链表中记录上一次分配的位置，从该位置继续查找，避免频繁从头搜索。
+ 空闲块合并：定期检查并合并相邻的空闲块，减少碎片。

BF
Best Fit 算法从空闲块链表中查找最接近请求大小的空闲块进行分配，即空闲块大小与请求大小的差值最小。

优点：
+ 减少碎片：通过选择最适合的块进行分配，尽可能减少未使用空间。
+ 更高效利用内存：比 First Fit 更适合内存紧张的情况下。

缺点：
+ 查找速度慢：必须遍历整个链表以找到最优块，增加查找开销。
+ 容易形成小碎片：由于优先选择最合适的块，剩余的小块可能不足以满足未来请求。

如何提高性能：
+ 维护有序链表：按块大小升序排列空闲块链表，避免每次分配都遍历整个链表。

+ 索引结构：使用二叉搜索树或堆等数据结构存储空闲块，以加速查找过程。

WF
Worst Fit 算法选择最大的空闲块进行分配，以期望剩余空间足够大，减少形成小碎片的可能性。

优点：
+ 减少小碎片的可能性：优先使用最大块，留下的碎片通常大到可以满足后续分配需求。
+ 更适合大块内存分配：对于需要大内存的进程，Worst Fit 更容易满足需求。

缺点：
+ 内存利用率低：大块空闲分区被分割后，剩余的空闲内存可能不够其他大块分配请求使用。长期运行可能导致内存利用率降低，剩余空间不足以满足大请求。
+ 产生更多的内存碎片：虽然 WF 算法试图避免小碎片，但它可能在划分最大的空闲分区后留下多块大小不均的碎片。这些碎片不一定能够满足其他请求，导致内存碎片化问题。

如何提高性能：
+ 维护降序链表：按块大小降序排列空闲块链表，使得第一个块就是最大的块。
+ 分段管理：将空闲块按大小分段存储（如：10-50、51-100 等），减少遍历范围。

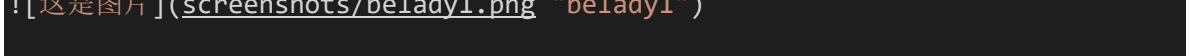
3 种算法的空闲块排序分别是如何实现的。
+ FF 无需排序。只需要按链表的插入顺序保存空闲块，分配时从头到尾线性查找第一个满足需求的块。
+ BF 按块大小升序排序。
+ WF 按块大小降序排序。

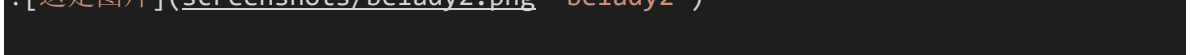
什么是内碎片、外碎片，紧缩功能解决的是什么碎片。
内碎片：
内碎片是指分配给进程的内存块中，实际使用的内存小于分配的内存，导致剩余部分无法被利用。
假设进程需要 13 KB 内存，而系统分配的最小块是 16 KB，剩余的 3 KB 是内碎片。
本次实验中，当空闲块大小足够，但分配后的剩余空间小于`MIN_SLICE`，则将剩余空间一起分配给进程，此时未被使用的剩余空间则为内碎片。内碎片不能被其他进程使用。
外碎片：
外碎片是指空闲内存总量足够，但被分散成多个不连续的小块，导致无法分配给需要连续大块内存的进程。
假设系统中有 3 个空闲块：5 KB、6 KB 和 4 KB，总计 15 KB，但如果一个进程需要 12 KB 的连续内存，这些碎片无法满足需求。
本次实验中，当空闲块大小足够，而且分配后的剩余空间大于`MIN_SLICE`，此时剩余空间则为外碎片。外碎片可以被其他进程使用。

紧缩功能解决的是外碎片。

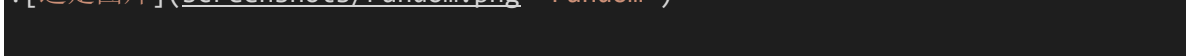
在回收内存时，空闲块合并是如何实现的？
在回收内存时，新释放的空闲块会插入到空闲链表中，并按照地址有序排列。合并操作通过遍历空闲链表来判断相邻的空闲块是否可以合并，并在条件满足时将它们合并为一个更大的空闲块。
+ 先按 FF 策略重排整个链表。
+ 将新释放的空闲块插入到空闲块链表中合适的位置。如果链表为空，新释放的空闲块成为链表的头节点。

+ 遍历空闲块链表，检查当前块和下一块是否相邻（即 当前块的结束地址 == 下一块的起始地址）。如果相邻，将两个块合并成一个。当前块的大小更新为当前块大小 + 下一块大小，然后跳过下一块节点（即将其从链表中删除）。
+ 在合并完成后，根据当前分配算法，对链表进行重新排列。

2.4 页面的置换
运行截图
FIFO Belady 现象：
引用序列：1 2 3 4 1 2 5 1 2 3 4 5
1. 设置帧数为 3，缺页次数为 9，缺页率为 75.00%。


2. 设置帧数为 4，缺页次数为 10，缺页率为 83.33%，高于帧数为 3 时。


LRU：
引用序列：0 1 2 0 1 2 3 4 0 1 0 2 3 0 1 2 0
这个序列局部性良好。
使用 LRU 算法，设置帧大小为 4，缺页次数为 9。


随机生成引用序列：


引用序列：0 1 2 1 2 3 1 0 4 5 4 6 5 4 5 6 4 0 1 2
FIFO：缺页率 50.00%
LRU：缺页率 45.00%

结果分析：
从实现和性能方面，比较分析 FIFO 和 LRU 算法。
实现复杂度：
+ FIFO：通常使用一个队列来维护页面的加载顺序。队列中的页面按访问顺序排列，页面访问时只是简单地检查页面是否在队列中，如果不在，就替换队列头部的页面并将新的页面加入队列尾部。入队和出队操作都非常简单且高效，时间复杂度为 O(1)。
+ LRU：要求跟踪每个页面的访问顺序，并总是淘汰最近最少被访问的页面。为了做到这一点，常见的实现方法是使用链表（或哈希表加双向链表）来快速更新页面的访问顺序。在本次实验中，因为帧数较少，选择依次遍历每个帧，找出最近最少访问的页面。

性能分析
+ FIFO：当引用串呈现较强的局部性时，FIFO 的缺页率通常较高。FIFO 可能会导致 Belady 现象，即增加帧数时，缺页次数反而增加。FIFO 算法没有考虑页面的访问频率或时间，它仅仅按照页面加载的顺序进行淘汰。对于具有时间局部性的引用串，FIFO 无法有效优化，导致页面经常被不适时地淘汰。
+ LRU：LRU 的缺页率通常比 FIFO 低，尤其是在有较强时间局部性的引用串中。LRU 能够根据页面的使用频率来淘汰最不常用的页面，因此能够更好地利用局部性原理，导致 Belady 现象。

LRU 算法是基于程序的局部性原理而提出的算法，你模拟实现的 LRU 算法有没有体现出该特点？如果有，是如何实现的？

在 LRU（最近最少使用） 算法中，局部性原理是指程序访问数据时，通常会频繁访问最近访问过的数据（时间局部性），而不常访问的则被淘汰。LRU 算法正是基于这一原则，通过维护页面的访问顺序，淘汰最近最少使用的页面，从而利用局部性原理来优化页面置换的性能。

find_lru 函数的核心逻辑是通过遍历当前帧（内存中的页面），找到访问时间戳最小的页面。这个页面在 LRU 算法中是最近最少使用的页面，符合时间局部性原则，应该被置换出去。

在设计内存管理程序时,应如何提高内存利用率。

1. 合理划分内存空间

分配适当大小的内存块：避免申请过多的过大内存块以及过小的内存块。过大的内存块可能导致内存浪费，过小的内存块可能会导致碎片化。在内存分配时，使用合适的内存块大小可以减少内存的浪费。

2. 优化内存分配与回收机制

在不确定内存需求的情况下，尽量避免提前分配过多内存。及时回收不再使用的内存空间。使用适当的垃圾回收机制来释放不再使用的内存，避免内存泄漏。手动内存管理时，可以确保每次分配内存后都进行释放操作。

3. 减少内存碎片

内存管理程序可以合并相邻的空闲块，避免因为分配和释放导致内存碎片，选择合适的内存分配算法来减少内存碎片。使用动态调整内存分配策略（如堆内存和栈内存的动态扩展与收缩），在程序运行时根据需求调整内存的分配和释放。

4. 使用虚拟内存管理

使用分页和分段技术来管理内存。分页机制能够将内存划分为固定大小的页面，而分段机制则根据程序的逻辑结构将内存划分为不同的段。这样可以有效利用内存，并避免外部碎片。
通过虚拟内存技术，操作系统可以在硬盘上创建虚拟的内存空间，部分数据可以交换到硬盘，释放内存供其他任务使用。虚拟内存通过页表管理，将程序使用的内存地址空间映射到实际的物理内存，从而提高内存的利用率。

4.2.3 实验三

模块签名

1. 生成一对公钥私钥

私钥地址：

/usr/src/kernels/4.19.90-2110.8.0.0119.oe1.aarch64/signing_key.priv

公钥地址：

/usr/src/kernels/4.19.90-2110.8.0.0119.oe1.aarch64/signing_key.x509

2. 配置内核启用模块签名

在/usr/src/kernels/4.19.90-2110.8.0.0119.oe1.aarch64 下运行`make menuconfig`

3. 生成签名密钥对：

`scripts/sign-file sha256 ./signing_key.priv ./signing_key.x509 module.ko`

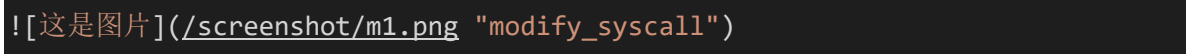
遇到的问题：

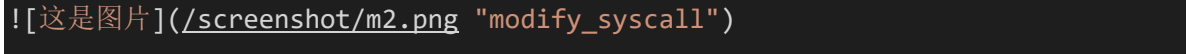
`At main.c:204:`

`SSL error:0909006C:PEM routines:get_name:no start line: crypto/pem/pem_lib.c:745`
`sign-file: ./signing_key.x509: No such file or directory`

篡改系统调用
arm64 架构下，gettimeofday 的模块号为 169。修改系统调用 gettimeofday 为自定义的 hello 函数。模块加载时会进行替换，卸载时会恢复原有的系统调用。

使用 update_mapping_prot 函数修改特定虚拟地址区域的读/写/执行权限。
sys_call_table 位于内核中的.rodata 段，修改时要临时解除写保护。为了方便操作，直接将整个系统调用表的操作权限修改为读写。

运行结果：
- 篡改系统调用前：


- 篡改系统调用后：


遇到的问题：
运行`insmod modify_syscall.ko`，内核崩溃。
原因：
1.sys_call_table 在内核中是只读的。没有禁用写保护，因此不能直接更改系统调用表。需要提前解除写保护，修改系统调用后再恢复。使用 update_mapping_prot 函数修改特定虚拟地址区域的读/写/执行权限。
2.代码中没有更改系统调用表的地址，使用`grep sys_call_table /boot/System.map-$(uname -r)`静态查询系统调用表，或使用`kallsyms_lookup_name("sys_call_table")`动态查询。

运行`./modify_new_syscall`，输出一个非 30 的整数。
原因：参数 10、20 没有直接传递，不能直接在 hello 函数中使用 (int a, int b) ，需要从寄存器中读取参数。

设备驱动
设计思路：
创建了一个新的字符设备驱动，修改原有设备驱动中的文件操作，重载`read()` `write()` `open()` `release()`四个函数。

`globalvar_open()`：打开设备时调用，通过 pid 注册客户端并初始化客户端结构体的相关数据。

`globalvar_release()`：释放设备时调用，释放对应的客户端数据。

`globalvar_read()`：读取设备时调用，客户端通过该函数读取消息。如果有消息，消息被复制到目标客户端的用户空间，否则等待新消息。

`globalvar_write()`：写入设备时调用，客户端可以通过该函数发送消息。支持群发消息（发送给所有客户端）和私聊消息（只发送给指定的客户端）。

多对多通信的实现：
每个客户端都有一个唯一的 pid，和一个消息队列用于存储消息。客户端通过设备文件与内核通信，系统会为每个客户端分配一个 client_data 结构体来存储相关信息。

每当一个进程打开设备文件时，会调用 globalvar_open 来注册该进程并创建或查找相应的 client_data 结构体。

每个客户端通过向设备写入消息来向其他客户端发送信息。消息的格式包括发送者的 pid 和消息内容。globalvar_write 函数处理消息的发送。对于私聊消息，只会被放入目标客户端的消息队列；对于群发消息，会放进所有客户端的消息队列。

globalvar_read 函数从客户端的消息队列中读取消息并拷贝用户空间。每个客户端的消息队列按 FIFO 顺序读取。读过的消息会从消息队列里删除。

消息发送格式：
私聊：sender_pid|@target_pid message
群发：sender_pid|message
在消息中加上发送方的 pid，方便接收方解析出这条消息从哪个进程发来的。

同步与互斥的实现：

- 每个客户端都有独立的消息队列，其消息不会受到其他客户端的影响，不同客户端之间的读写操作可以互不干扰。
  如果使用全局的单一消息队列，当队列满或阻塞时，所有客户端的操作都会受影响。

- 读写锁：read_sem 和 write_sem 确保同一时刻只有一个进程可以进行写入操作，写入的时候其他进程不能开始读。

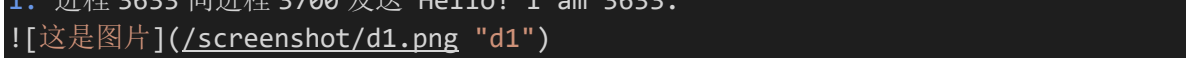
- 等待队列：read_queue 和 write_queue 协调读进程和写进程的操作。
  当某个进程调用 read，但当前没有未读的消息时，该进程会被加入到 read_queue 中，并进入睡眠状态；同理，当某个进程调用 write，但设备或缓冲区暂时无法接受数据时（如当前有进程在读），该进程会被加入到 write_queue 中，并进入睡眠状态。

- 在设备数据结构体 chat_device_data 中，加入整型变量 readers_count，实时监测当前读进程的个数。当 readers_count==0 时，才允许写入消息。

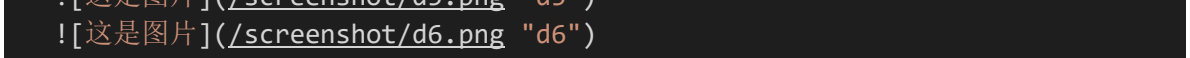
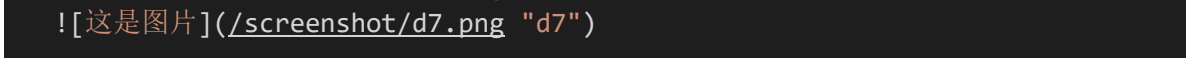
wait_event_interruptible 函数：
函数原型：`int wait_event_interruptible(wait_queue_head_t *queue, int condition);`
queue：指向等待队列头部的指针。等待队列是一个数据结构，用于管理所有等待某个条件的进程。内核中的多个进程可能在不同的等待队列上等待不同的条件。
condition：表示进程需要等待的条件。如果这个条件为 false，进程就会被阻塞（即挂起）。当条件为 true 时，进程将从等待队列中被唤醒。

字符串转整形函数 kstrtouint：
`int kstrtouint(const char *s, unsigned int base, unsigned int *res);`

s 是输入的字符串，base 是数值基数，可以是 10（代表十进制）、16（代表十六进制）或者是 0（自动识别基数），res 是用来存放转换后的整型值的指针。当转换过程中没有错误发生时，函数返回 0。

运行结果：
分别创建三个进程，pid 分别为 3633、3661、3700
1. 进程 3633 向进程 3700 发送"Hello! I am 3633."



2. 进程 3700 向进程 3661 发送"Hi!"

3. 进程 3661 向 3633、3700 广播"It is a good day." "Nice to meet you."
   
   
   

遇到的问题：
1. `Failed to open device: No such file or directory`
   代码中没有创建设备类和设备节点，需要手动创建。`mknod /dev/globalvar c 290 0`，在 /dev 下创建设备文件，290 是主设备号，与驱动中的 MAJOR_NUM 相同。0 是次设备号。
   后续在代码里加入了 class_create 和 device_create 函数，可以自动创建设备类和设备节点，不再使用 mknod 指令手动创建。
   检查设备节点是否创建：`ls /dev/globalvar`

2. 加锁时使用`mutex_lock`或者`mutex_trylock`容易导致死锁
   因为在读操作和写操作中需要加多把锁，最好使用信号量，这样方便定义不同的锁。

3. 用@标识私聊信息时，目标进程的 pid 不能作为一个单独的参数写入队列，只能包含在信息中
   使用字符串转整形函数 kstrtouint，在信息中提取目标 pid。
   `int kstrtoint(const char *s, unsigned int base, int *res);`
   其中，s 是输入的字符串，base 是数值基数，可以是 10（代表十进制）、16（代表十六进制）或者是 0（自动识别基数），res 是用来存放转换后的整型值的指针。当转换过程中没有错误发生时，函数返回 0。

4. 实现群发功能时，如果只使用一个缓冲区，难以判断所有目标进程是否都读取到消息并拷贝到用户空间。
   给每个客户端（进程）创建一个消息队列，群发时把消息放进所有进程的消息队列。这样虽然重复占用内核空间，但是能保证每个进程对于消息的读取是独立的。

5. 测试时，向另一个进程发送消息，接收方不能实时打印收到的消息
   接受方正在等待写操作的输入，进程卡在 scanf。
   创建读线程和写线程，一个用于接收消息，另一个用于发送消息。写线程等待用户写入消息，读线程实时打印其他进程发来的消息。

使用`dmesg | grep globalvar`查看内核日志

```
grep sys_call_table /boot/System.map-$(uname -r)
dmesg -c
```