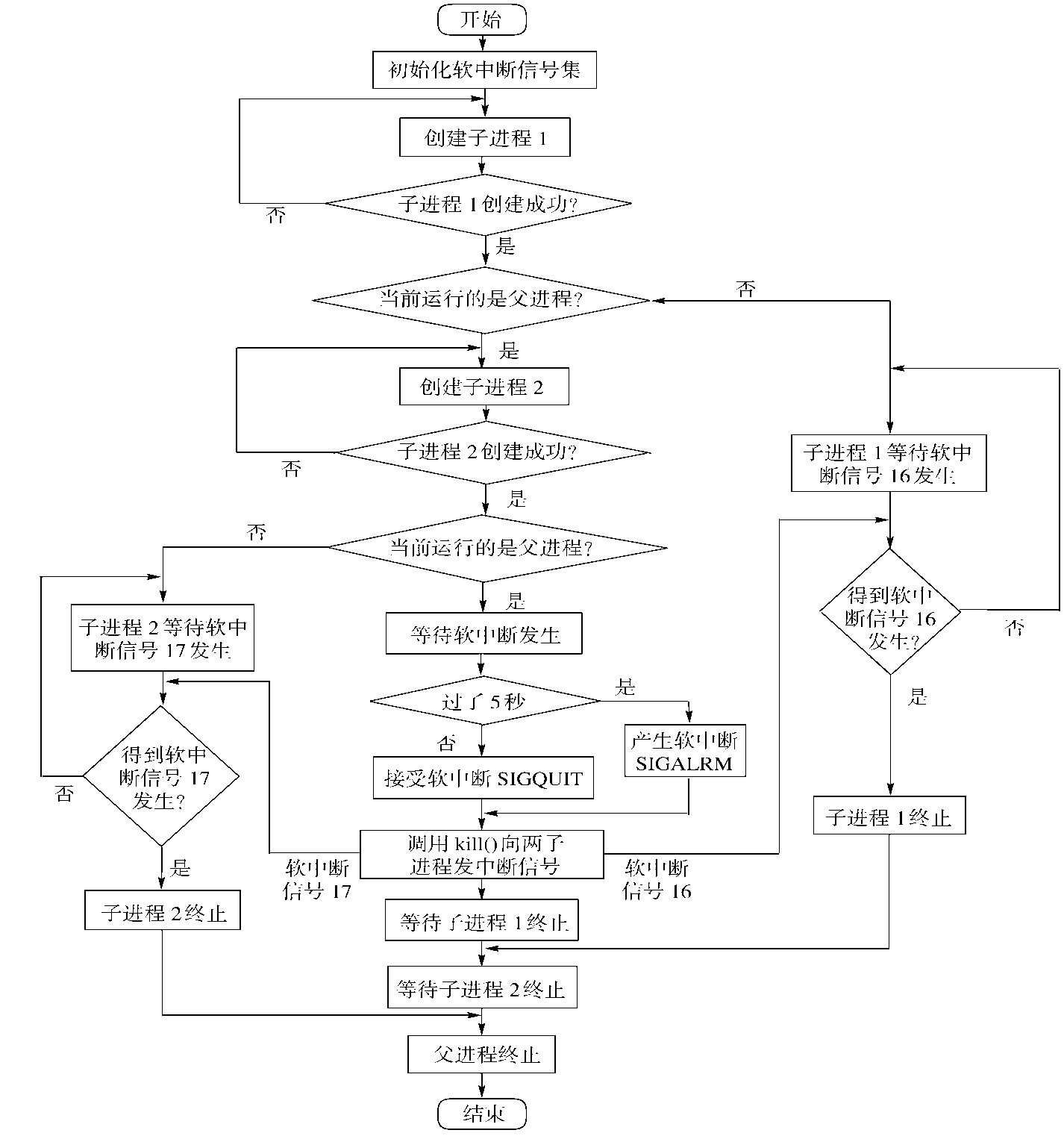
### 实验二 进程通信与内存管理

#### 1.进程的软中断通信

#### **实验内容：**

1. 根据流程图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!
* 

1. 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
2. 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

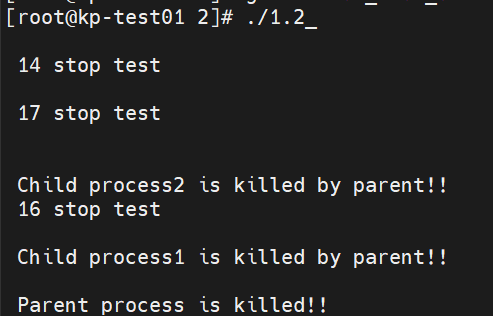
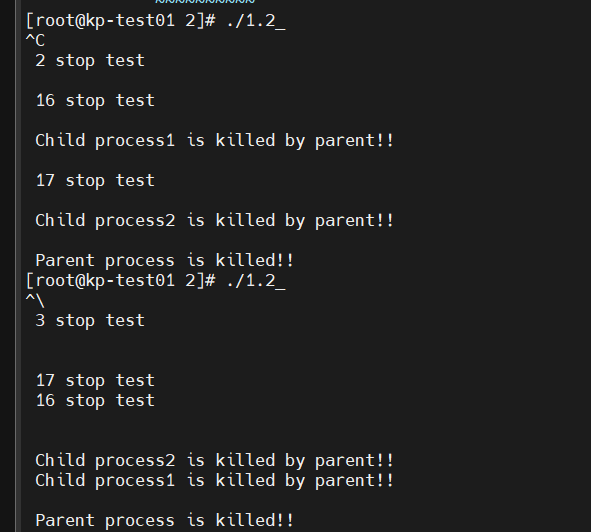
#### 编写实验代码需要考虑的问题：

1. 如何阻塞住子进程，让子进程等待父进程发来信号？
2. 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？

* void inter\_handler(int sig)  
  {  
   flag = 0;  
   printf(" \n %d stop test \n", sig);  
  }  
  void waiting()  
  {  
   while (flag == 1)  
   ;  
  }   
    
  signal(17, inter\_handler);  
  waiting();  
  printf("\n Child process2 is killed by parent!! \n");
* 子进程通过waiting()函数等待信号到来，当flag为1时，一直实现阻塞；（接收到信号时flag值改变），依次确保子进程已经准备好接收信号。

#### 运行结果与分析：

1. 最初认为运行结果：
   * 子进程并行执行顺序不固定，
   * 5s内接收到终止信号时父进程用系统调用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输出相应信息。
   * 5s后进行闹钟中断，产生sigalarm信号使父进程用系统调用kill()向两个子进程分别发出整数值为16 和17 软中断信号。
2. 实际运行结果：

* 该图表示：5s后执行闹钟中断（信号14），父进程向两个子进程分别发出整数值为16 和17 软中断信号子进程结束。
* 
* 执行闹钟中断
* 该图表示5s内执行sigquit信号（信号2）或sigint（信号3），父进程向两个子进程分别发出整数值为16 和17 软中断信号子进程结束。
* 
* 为实现子进程接收父进程的软中断信号，在子进程中创建了一个新的信号屏蔽字newmask，其中阻塞了SIGQUIT和SIGINT信号。接着使用sigprocmask函数将这个新的屏蔽字应用到了进程中。
* long unsigned int newmask ;  
  signal(16, inter\_handler);  
  sigemptyset(&newmask);  
  sigaddset(&newmask, SIGQUIT);  
  sigaddset(&newmask, SIGINT);  
  sigprocmask(0, &newmask, NULL);

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

* 主动退出通过调用exit()系统调用来实现。exit()系统调用会终止进程的执行，并将控制权交还给操作系统。主动调用exit()系统调用更好一些。因为这种方式可以确保进程在退出之前完成清理工作，释放资源，关闭文件等操作，而使用kill命令可能会导致进程突然被终止，无法完成必要的清理工作，可能会导致资源泄漏或者数据丢失。

#### 2.**进程的管道通信**

#### **实验内容：**

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。

完整代码

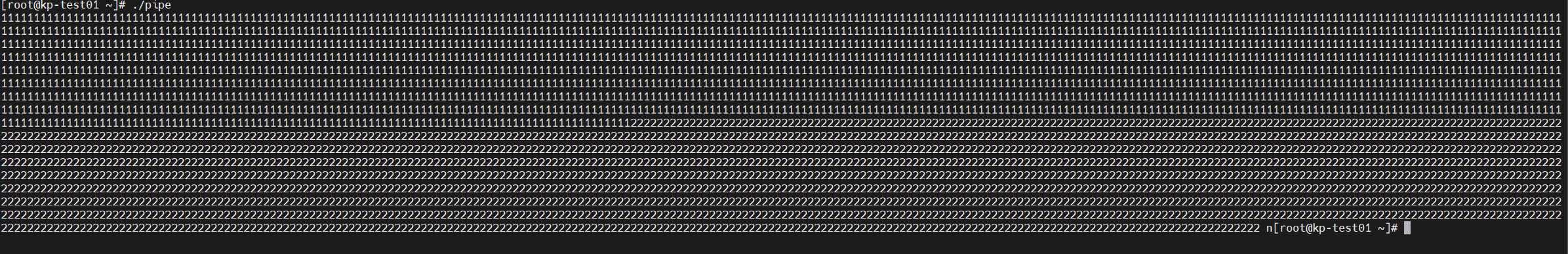
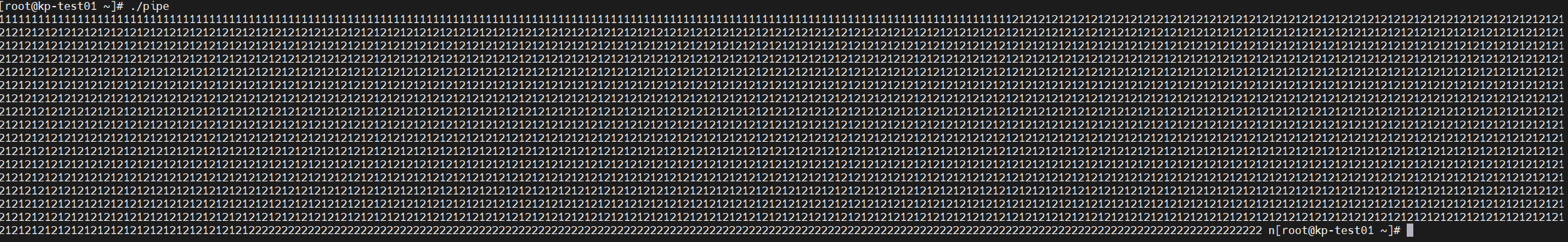
#include <unistd.h>  
#include <sys/types.h>   
#include <sys/wait.h>  
#include <signal.h>  
#include<stdlib.h>  
#include <stdio.h>  
int pid1, pid2; // 定义两个进程变量  
int main()  
{  
 int fd[2];  
 char InPipe[4000]; // 定义读缓冲区  
 char c1 = '1', c2 = '2';  
 pipe(fd);  
 // 创建管道  
 while ((pid1 = fork()) == 1)  
 ; // 如果进程1 创建不成功 则空循环  
 if (pid1 == 0)  
 { // 如果子进程1 创建成功, pid1 为进程号  
  
 // 锁定管道  
 lockf(fd[1], 1, 0);  
 // TODO 分2000 次每次向管道写入字符’1’  
 for (int i = 0; i < 2000; i++)  
 {  
 write(fd[1], &c1, 1);  
 }  
 sleep(5); // 等待读进程读出数据  
  
 // TODO 解除管道的锁定  
 lockf(fd[1], 0, 0);  
  
 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);  
 // 分2000 次每次向管道写入字符’2’  
 for (int i = 0; i < 2000; i++)  
 {  
 write(fd[1], &c2, 1);  
 }  
  
 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  
 int bytesRead = read(fd[0], InPipe, 4000); // 从管道中读出4000 个字符  
 InPipe[bytesRead] = '\0'; // 加字符串结束符  
 printf("%s n", InPipe); // 显示读出的数据  
 exit(0); // 父进程结束  
 }  
 }  
}

#### 运行结果与分析：

1. 你最初认为运行结果会怎么样？

* 输出2000个1再输出2000个2。因为存在锁所以不会交替输出。

1. 实际的结果什么样？有什么特点？试对产生该现象的原因进行分析。

* **实际结果：**先输出2000个1再输出2000个2，原因：子进程1先锁住占用了管道，导致子进程2无法访问管道，在子进程1完成写操作并解锁后，子进程才能使用管道。
* 
* 实验过程中发现许运行一段时间才有输出，原因是：由于pipe 读写的互锁产生延时，pipe在写完2000个1和2000个2后再读的时候才产生输出。
* **去锁结果：**
* 
* 由图中可以看出子进程1先执行，再子进程2开始运行后来哦你跟着开始交替输出。

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

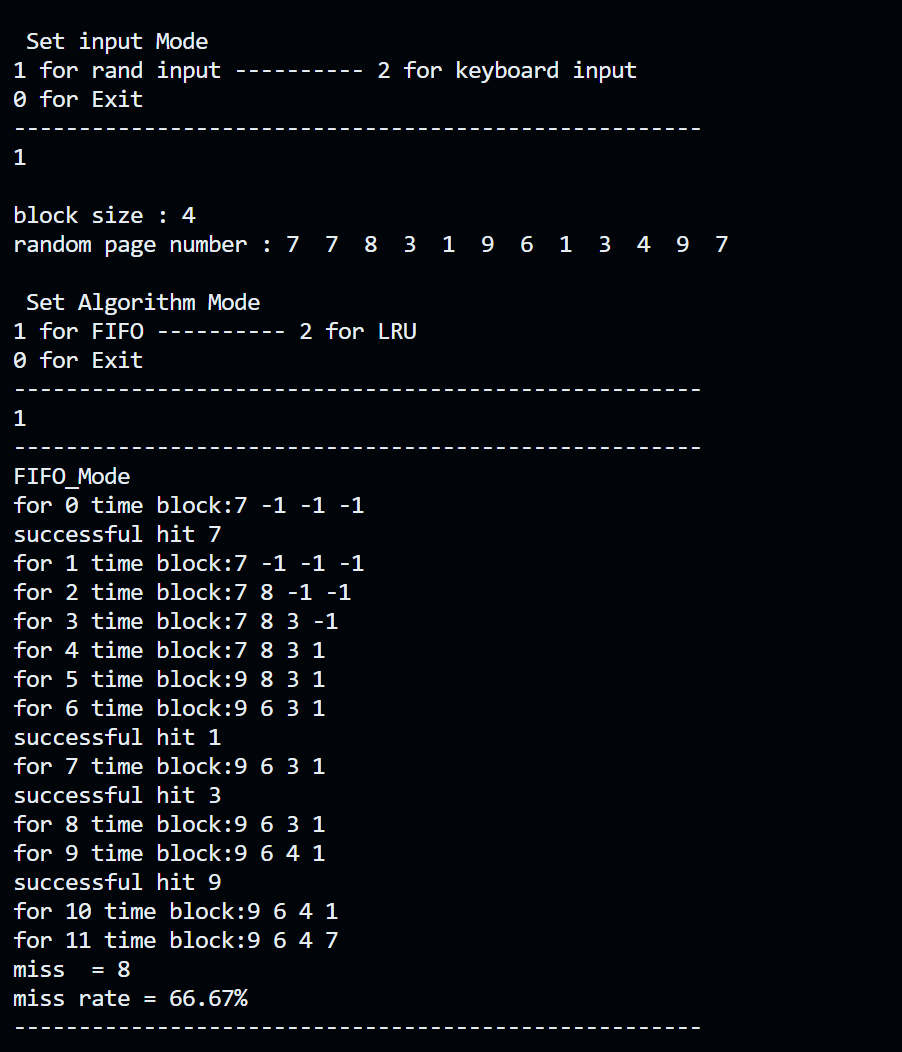
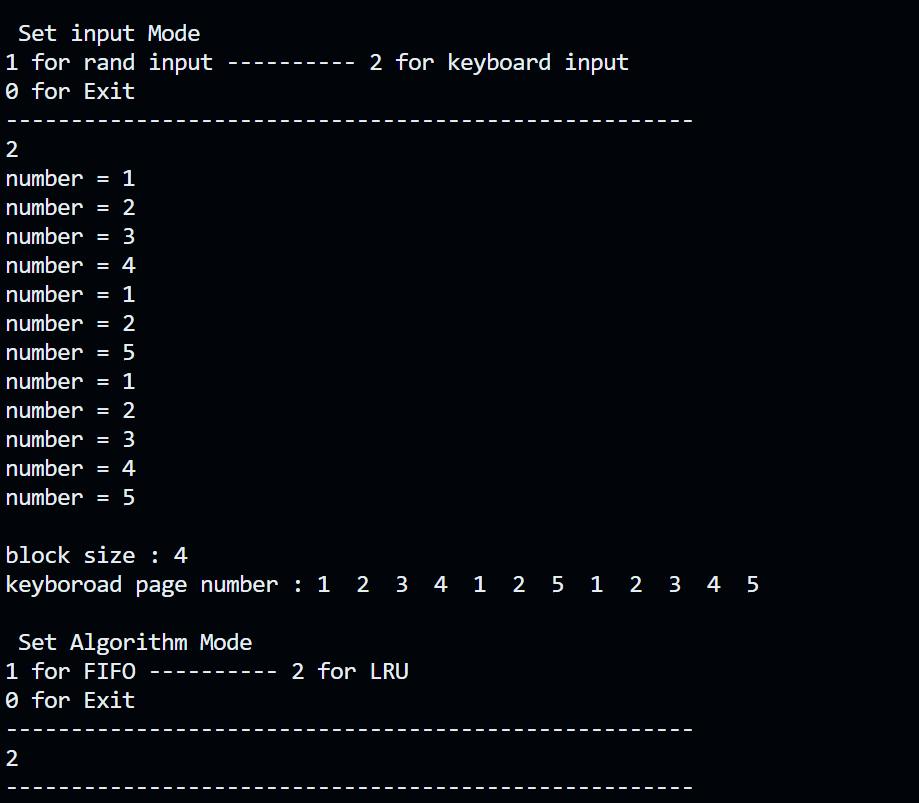
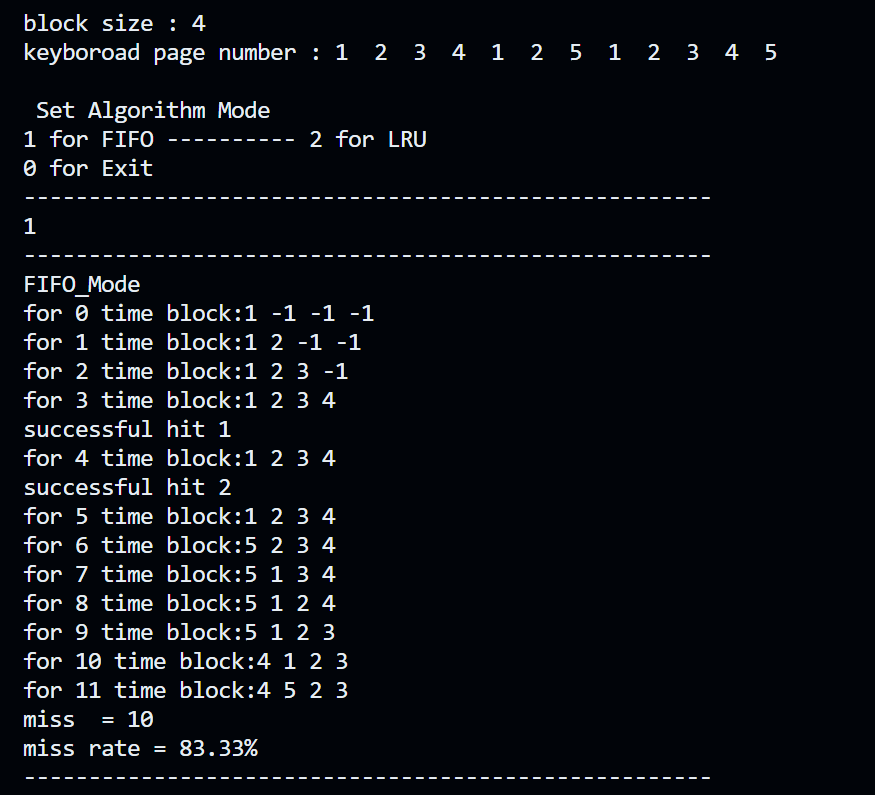
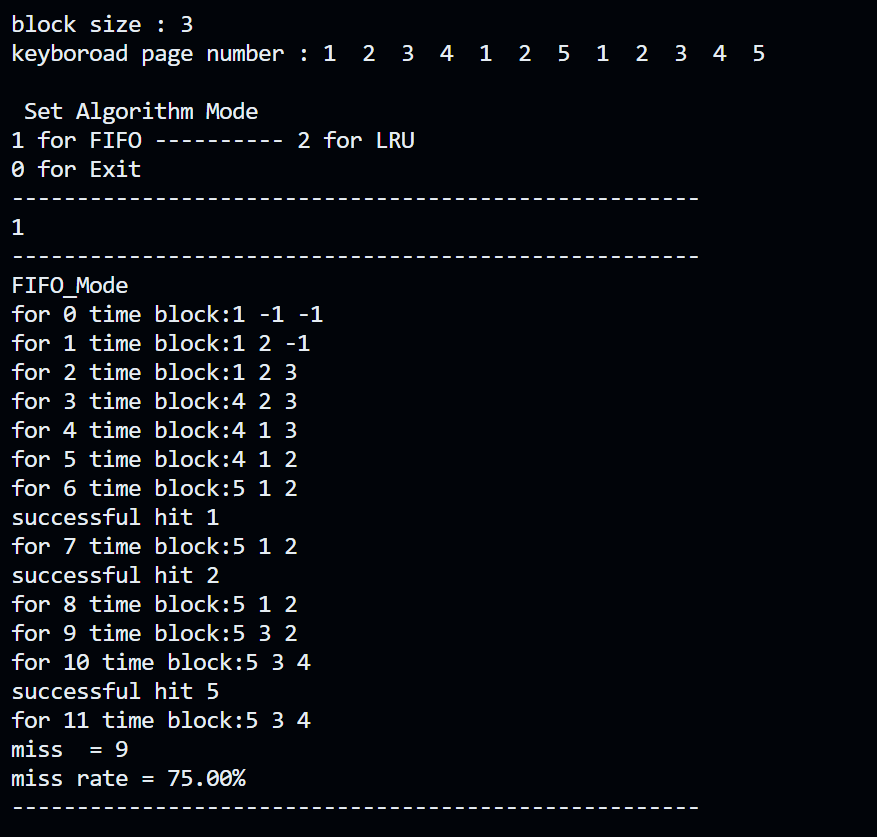
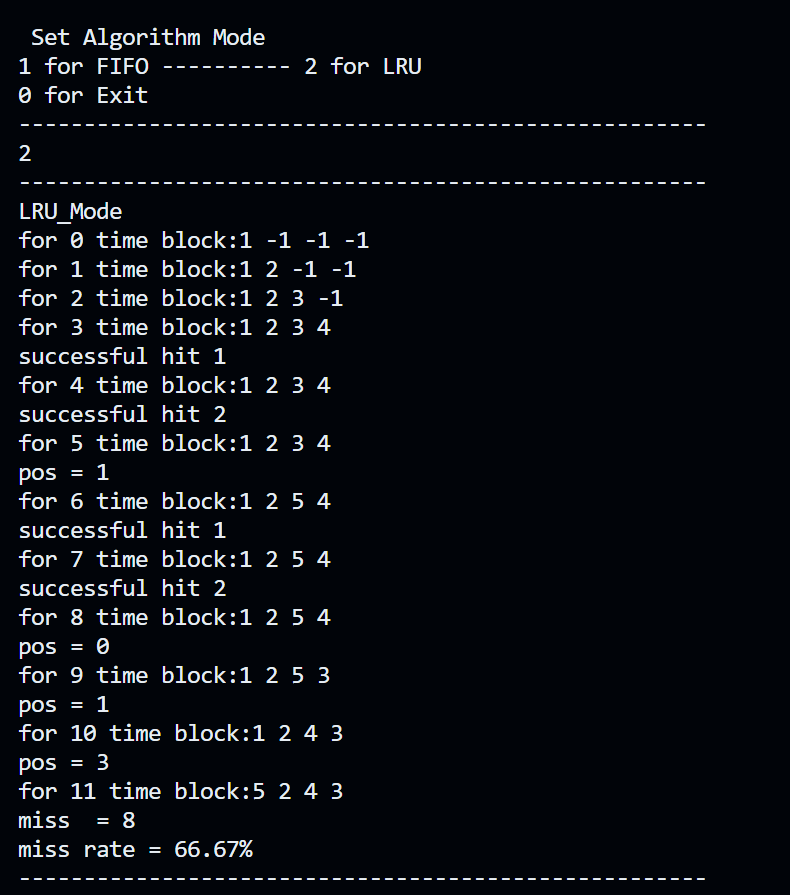
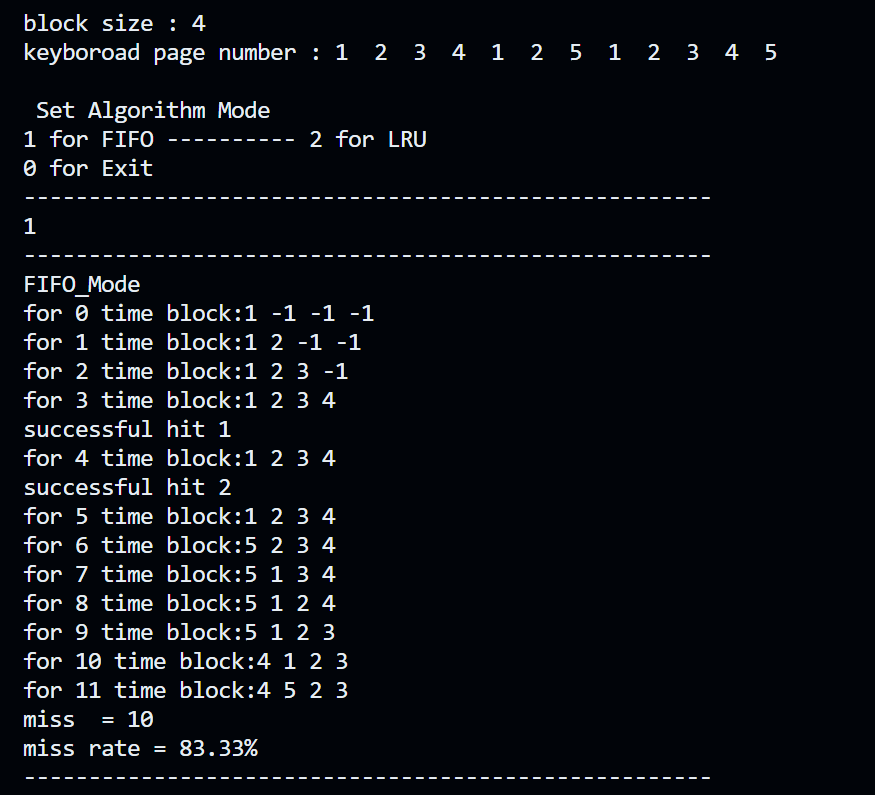
* 通过控制lock(),lockf(fd[1],1,0)代表上锁，此时其他进程无法访问管道，lockf(fd[1],0,0)代表解锁，此时其他进程能访问管道。不控制同步与互斥会让子进程1、2同步运行，出现1,2交替输出的情况。

实验参考资料：<https://blog.csdn.net/studyhardi/article/details/89852839>

#### 3.页面的置换

#### 实验内容：

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。

1. 运行所实现的算法，并通过对比，分析2 种算法的优劣。
   * 实现随机数自动生成、手动输入**两种生成方式**，并**输出过程和最终缺页率**
   * 
   * 随机生成模式
   * 
   * 手动输入模式
2. 设计测试数据，观察FIFO 算法的BLEADY 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。
   * **FIFO 算法的BLEADY 现象：**当使用序列 **{1，2，3，1，2，5，1，2，3，4，5}**发现出现BLEADY 现象：blocksize 增大 ，缺页率升高。
   * 
   * block size = 4
   * 
   * block size = 3
   * **出现原因：FIFO算法的置换特征与进程访问内存的动态特征是矛盾的，即被置换的页面并不是进程不会访问的**
   * **具有局部性特点的测试数据两种算法的优劣性：**
   * **{1，2，3，1，2，5，1，2，3，4，5}**序列具有较好的局部性，在block =4 情况下观察两种算法：
   * 
   * LRU
   * 
   * FIFO
   * FIFO的缺页率(83.33%)高于LRU(75%)，原因：LRU算法依据局部性原理若当前内存分配的页面数已满，则用新加入的页面直接替换掉最不常访问的页面，**既对某些被频繁地访问的页面有较好的利用率**；FIFO算法则是用新加入的页面直接替换掉最先加入的页面，这种算法没有考虑局部性（**被它替换出去的页面并不一定是进程不会访问的**）。所以对于局部性较好的数据（某一部分序列频率较高），LRU算法效果好。

#### 运行结果与分析：

1. 从实现和性能方面，比较分析FIFO和LRU算法。

#### FIFO

#### 算法实现：

* + void FIFO()  
    {  
     int miss = 0; // miss number  
     double miss\_rate;  
     int num;  
     int empty\_flag;  
     queue.head = block\_num-1;  
     queue.rear = block\_num;  
     for (num = 0; num < total\_input; num++) // input coming  
     {  
     empty\_flag = FIFO\_Empty(block, page[num]);  
     if (empty\_flag >= block\_num) // hit  
     {  
     printf("successful hit %d\n", page[num]);  
     }  
     else if (empty\_flag == -1) // full but not hit  
     {  
     miss++;  
     block[queue.head] = page[num];  
     if (queue.rear==0) {  
     queue.rear =block\_num-1;  
     }  
     else{  
     queue.rear = (queue.rear - 1) % (block\_num);  
     }  
     if (queue.head==0) {  
     queue.head =block\_num-1;  
     }  
     else{  
     queue.head = (queue.head - 1) % (block\_num);  
     }  
     }  
      
     else  
     {  
     miss++;  
     block[empty\_flag] = page[num];  
     if (queue.rear==0) {  
     queue.rear =block\_num-1;  
     }  
     else{  
     queue.rear = (queue.rear - 1) % (block\_num);  
     }  
     //printf("rear = %d\n",queue.rear);  
     }  
     printf("for %d time block:", num);  
     for (int i = block\_num-1; i >= 0; i--)  
     {  
     printf("%d ", block[i]);  
     }  
     printf("\n");  
     }  
      
     miss\_rate = 100 \* ((double)miss / total\_input);  
    }
  + 通过 FIFO\_Empty（）函数判断当前内存块状态（能命中，无命中有空块，无命中无空块）。维护一个先进先出的队列，记录页面进入顺序。
  + **无命中有空块：**将页面至于空块，更新队列（尾更新）。
  + **无命中无空块：**通过队列记录页面进入顺序，进行页面置换。选择队列最前面的页面进行替换，更新队列（头尾均需跟新）。

#### 性能：

* + **缺点：**FIFO算法只考虑页面进入内存的顺序，而不考虑页面的重要性和使用频率，导致性能较差。并存在Belady异常（无法根据页面的使用情况进行自适应的页面置换）
  + **优点：**只需要维护一个先进先出的队列，复杂度低，开销较低，可以确保每个页面都有被置换的机会，公平地对待每个页面。

#### LRU

* + **算法实现：**
  + void LRU()  
    {  
     int miss = 0; // miss number  
     double miss\_rate;  
     int num;  
     int pos = 0;  
     int empty\_flag;  
     for (num = 0; num < total\_input; num++)  
     {  
     empty\_flag = FIFO\_Empty(block, page[num]);  
     if (empty\_flag >= block\_num) // hit  
     {  
     for (int i = 0; i < stack.top; i++)  
     {  
     if (page[num] == stack.num[i])  
     {  
     pos = i;  
     break;  
     }  
     }  
     for (int i = pos + 1; i < stack.top; i++)  
     {  
     stack.num[i - 1] = stack.num[i];  
     }  
     stack.num[stack.top - 1] = page[num];  
     printf("successful hit %d\n", page[num]);  
     }  
      
     else if (empty\_flag == -1) // full but not hit   
     {  
     miss++;  
     // 找到最远使用替换  
     pos = FIFO\_Empty(block, stack.num[0]) - block\_num;  
     block[pos] = page[num];  
      
     // 更新结构体  
     for (int i = 0; i < stack.top - 1; i++)  
     {  
     stack.num[i] = stack.num[i + 1];  
     }  
     stack.num[stack.top - 1] = page[num];  
     }  
      
     else  
     {  
     miss++;  
     block[empty\_flag] = page[num];  
     stack.num[stack.top] = page[num];  
     stack.top++;  
     }  
     miss\_rate = 100 \* ((double)miss / total\_input);  
    }
  + 通过 FIFO\_Empty（）函数判断当前内存块状态（能命中，无命中有空块，无命中无空块）。维护一个栈，栈代表页面被使用时间的远近。
  + **无命中有空块：**将页面至于空块，更新栈（将新加入的页面至于栈顶）。
  + **无命中无空块：**找到栈底元素在block的位置然后将新加入的元素置换，更新栈（所有栈中元素向下平移）。
  + **命中：**找到命中的页面在栈中的位置，将其置于栈顶，其他页面向下平移。

#### 性能：

* + - LRU算法适合具有**较强时间局部性的访问序列**，即最近被访问的页面可能会在未来继续被访问的情况。例如，顺序访问或者循环访问的情况下，LRU算法能够比较好地预测未来的访问模式，提高缓存命中率。不适合的序列包括**周期性访问、随机访问**等无法很好地利用时间局部性的情况。
    - 与FIFO算法相比，LRU算法时间复杂度较高，需进行大量维护栈的操作，开销较大。

1. 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？

* 有，**体现在维护的数据结构----栈**。栈中元素位置由使用改页面的时间先后决定，符合时间局部性原理。

1. 在设计内存管理程序时 应如何提高内存利用率。

* 实现内存碎片整理：通过内存碎片整理技术，可以将内存中的碎片化空间进行整理，从而提高内存的利用率。
* 使用内存池技术：内存池技术可以预先分配一定大小的内存块，在需要时直接从内存池中获取，避免频繁的内存分配和释放操作，从而提高内存的利用率。
* 实施内存压缩技术：内存压缩技术可以通过对内存中的数据进行压缩，从而减少内存的使用量，提高内存的利用率。
* 使用内存共享技术：内存共享技术可以让多个进程共享同一块内存，避免多次复制相同的数据，从而提高内存的利用率。