实验 2.1 进程的软中断通信

```
[root@kp-test01 src]# ./test
^C
 2 stop test


 16 stop test
 17 stop test

 Child process 2 is killed by parent !!

 Child process 1 is killed by parent !!

 Parent process is killed !!
[root@kp-test01 src]# ./test

 14 stop test

 16 stop test

 Child process 1 is killed by parent !!

 17 stop test

 Child process 2 is killed by parent !!

 Parent process is killed !!
[root@kp-test01 src]#
```

分析：创建两个子进程后在父进程中设置 signal(3,stop);signal(2,stop); signal(14,stop);在 stop 函数中将 wait_flag 设置为 0，然后 sleep(5)之后对 wait_flag 进行判断，如果 wait_flag 为 0,说明已经父进程已经收到 SIGINT 信号或者 SIGQUIT 信号，跳出 if 语句即可；如果 wait_flag 为 1，说明父进程还没有收到 SIGINT 信号或者 SIGQUIT 信号，此时需要给父进程发送一个 SIGALARM 信号，然后父进程调用 signal(14,stop)；if 语句结束后父进程给两个子进程分别发送信号 16 和信号 17,同时在两个子进程内设置 signal(2,SIG_IGN); signal(3,SIG_IGN);来屏蔽 SIGINT 信号和 SIGQUIT 信号，子进程收到父进程的中断信号之后调用 stop 函数将 wait_flag 置 0，跳出了 while(wait_flag)；循环，打印之后调用 exit(0)返回父进程。

困难：不知道 signal 在程序中的作用，不知道怎么去设置那个 5 秒闹钟中断指令。

实验 2.2 进程的管道通信

有锁：

```
[root@kp-test01 src]# ./test

 Child process 1 is sending message!
111111111111111111111111111111111111111111111111111111111111111111111111111111111111111111111111111111111111111111111111
111111111111111111111111111111111111111111111111111111111111111111111111111111111111111111111111111111111111111111111111
111111111111111111111111111111111111111111111111111111111111111111111111111111111111111111111111111111111111111111111111
111111111111111111111111111111111111111111111111111111111111111111111111111111111111111111111111111111111111111111111111
111111111111111111111111111111111111111111111111111111111111111111111111111111111111111111111111111111111111111111111111
111111111111111111111111111111111111111111111111111111111111111111111111111111111111111111111111111111111111111111111111
111111111111111111111111111111111111111111111111111111111111111111111111111111111111111111111111111111111111111111111111
111111111111111111111111111111111111111111111111111111111111111111111111111111111111111111111111111111111111111111111111
111111111111111111111111111111111111111111111111111111111111111111111111111111111111111111111111111111111111111111111111
111111111111111111111111111111111111111111111111111111111111111111111111111111111111111111111111111111111111111111111111
111111111111111111111111111111111111111111111111111111111111111111111111111111111111111111111111111111111111111111111111
111111111111111111111111111111111111111111111111111111111111111111111111111111111111111111111111111111111111111111111111
111111111111111111111111111111111111111111111111111111111111111111111111111111111111111111111111111111111111111111111111
111111111111111111111111111111111111111111111111111111111111111111111111111111111111111111111111111111111111111111111111
111111111111111111111111111111111111111111111111111111111111111111111111111111111111111111111111111111111111111111111111
111111111111111111111111111111111111111111111111111111111111111111111111111111111111111111111111111111111111111111111111
11111111111111111111111111111111111111111111111111111111111111111111111111111111
 Child process 2 is sending message!
222222222222222222222222222222222222222222222222222222222222222222222222222222222222222222222222222222222222222222222222
222222222222222222222222222222222222222222222222222222222222222222222222222222222222222222222222222222222222222222222222
222222222222222222222222222222222222222222222222222222222222222222222222222222222222222222222222222222222222222222222222
222222222222222222222222222222222222222222222222222222222222222222222222222222222222222222222222222222222222222222222222
222222222222222222222222222222222222222222222222222222222222222222222222222222222222222222222222222222222222222222222222
222222222222222222222222222222222222222222222222222222222222222222222222222222222222222222222222222222222222222222222222
222222222222222222222222222222222222222222222222222222222222222222222222222222222222222222222222222222222222222222222222
222222222222222222222222222222222222222222222222222222222222222222222222222222222222222222222222222222222222222222222222
222222222222222222222222222222222222222222222222222222222222222222222222222222222222222222222222222222222222222222222222
222222222222222222222222222222222222222222222222222222222222222222222222222222222222222222222222222222222222222222222222
222222222222222222222222222222222222222222222222222222222222222222222222222222222222222222222222222222222222222222222222
222222222222222222222222222222222222222222222222222222222222222222222222222222222222222222222222222222222222222222222222
222222222222222222222222222222222222222222222222222222222222222222222222222222222222222222222222222222222222222222222222
222222222222222222222222222222222222222222222222222222222222222222222222222222222222222222222222222222222222222222222222
222222222222222222222222222222222222222222222222222222222222222222222222222222222222222222222222222222222222222222222222
222222222222222222222222222222222222222222222222222222222222222222222222222222222222222222222222222222222222222222222222
22222222222222222222222222222222222222222222222222222222222222222222222222222222
[root@kp-test01 src]#
```

无锁：

```
[root@kp-test01 src]# ./test

 Child process 1 is sending message!
1111111111111111111111111111111111111111111111111111111111111111111111111111111111111111111111111111111111111111111111
11111111111111111111111111111111111111111111111111
 Child process 2 is sending message!
1212121212121212121212121212121212121212121212121212121212121212121212121212121212121212121212121212121212121212121212
1212121212121212121212121212121212121212121212121212121212121212121212121212121212121212121212121212121212121212121212
1212121212121212121212121212121212121212121212121212121212121212121212121212121212121212121212121212121212121212121212
1212121212121212121212121212121212121212121212121212121212121212121212121212121212121212121212121212121212121212121212
1212121212121212121212121212121212121212121212121212121212121212121212121212121212121212121212121212121212121212121212
1212121212121212121212121212121212121212121212121212121212121212121212121212121212121212121212121212121212121212121212
1212121212121212121212121212121212121212121212121212121212121212121212121212121212121212121212121212121212121212121212
1212121212121212121212121212121212121212121212121212121212121212121212121212121212121212121212121212121212121212121212
1212121212121212121212121212121212121212121212121212121212121212121212121212121212121212121212121212121212121212121212
1212121212121212121212121212121212121212121212121212121212121212121212121212121212121212121212121212121212121212121212
1212121212121212121212121212121212121212121212121212121212121212121212121212121212121212121212121212121212121212121212
1212121212121212121212121212121212121212121212121212121212121212121212121212121212121212121212121212121212121212121212
1212121212121212121212121212121212121212121212121212121212121212121212121212121212121212121212121212121212121212121212
1212121212121212121212121212121212121212121212121212121212121212121212121212121212121212121212121212121212121212121212
1212121212121212121212121212121212121212121212121212121212121212121212121212121212121212121212121212121212121212121212
1212121212121212121212121212121212121212121212121212121212121212121212121212121212121212121212121212121212121212121212
1212121212121212121212121212121212121212121212121212121212121212121212121212121212121212121212121212121212121212121212
1212121212121212121212121212121212121212121212121212121212121212121212121212121212121212121212121212121212121212121212
1212121212121212121212121212121212121212121212121212121212121212121212121212121212121212121212121212121212121212121212
1212121212121212121212121212121212121212121212121212121212121212121212121212121212121212121212121212121212121212121212
1212121212121212121212121212121212121212121212121212121212121212121212121212121212121212121212121212121212121212121212
1212121212121212121212121212121212121212121212121212121212121212121212121212121212121212121212121212121212121212121212
1212121212121212121212121212121212121212121212121212121212121212121212121212121212121212121212121212121212121212121212
1212121212121212121212121212121212121212121212121212121212121212121212121212121212121212121212121212121212121212121212
1212121212121212121212121212121212121212121212121212121212121212121212121212121212121212121212121212121212121212121212
1212121212121212121212121212121212121212121212121212121212121212121212121212121212121212121212121212121212121212121212
1212121212121212121212121212121212121212121212121212121212121212121212121212121212121212121212121212121212121212121212
1212121212121212121212121212121212121212121212121212121212121212121212121212121212121212121212121212121212121212121212
1212121212121212121212121212121212121212121212121212121212121212121212121212121212121212121212121212121212121212121212
1212121212121212121212121212121212121212121212121212121212121212121212121212121212121212121212121212121212121212121212
1212121212121212121212121212121212121212121212121212121212121212121212121212121212121212121212121212121212121212121212
1212122222222222222222222222222222222222222222222222222222222222222222222222222222222222222222222222222222222222222222
22222222222222222222222222222222222222222222222222222222
```

分析：

1．预想中的结果：有锁的时候，先输出 2000 个 1，然后输出 2000 个 2。无锁的时候，1 和 2 随机输出，没有规律，无法预测。

2．实际中的结果和预想中的结果几乎一致，当无锁的时候输出非常混乱，两个子进程的输出在父进程的输出中交错出现。由于没有进行同步和互斥控制，子进程 1 和子进程 2 在写入数据时，没有对管道进行互斥锁定，因此它们可以同时写入数据。而父进程也可以同时从管道中读取数据，父进程在读取数据时也没有进行同步操作，因此它可能在子进程 1 和子进程 2 还未完全写入数据时就开始读取。缺乏同步和互斥机制，导致进程之间的操作相互干扰，产生混乱的结果。导致数据混乱。

当有锁的时候，子进程 1 先上锁,将 2000 个 1 写到管道中，然后释放锁，接着子进程 2 上锁，将 2000 个 2 写到管道中。

3．在这个实验中，使用了 lockf 函数对管道进行锁定和解锁的操作，实现了对管道通信的同步与互斥。lockf(fd[1], 1, 0)：锁定管道写入端，确保在一个进程写入时其他进程无法写入。lockf(fd[1], 0, 0)：解锁管道写入端，允许其他进程写入。这样做的目的是为了防止两个子进程同时向管道写入数据，造成混乱。

如果不控制同步与互斥，可能导致以下后果：

数据混乱：多个进程同时向管道写入数据，可能导致数据混在一起，难以区分哪个进程写入的哪部分数据。

读取不完整：父进程在子进程还未完全写入数据时就开始读取，可能导致读取不完整或不正确的数据。

程序崩溃：多个进程同时对管道进行读写，可能引发竞争条件，导致程序崩溃或产生未定义的行为。

实验 2.3 页面置换

FIFO 和 LRU：

```
[root@kp-test01 src]# ./test
0
3  0  2  2  3  6  3  1  6  3  0  0
FIFO:
3
3  0
3  0  2
3  0  2
3  0  2
3  0  2  6
3  0  2  6
0  2  6  1
0  2  6  1
2  6  1  3
6  1  3  0
6  1  3  0
0.583333
LRU:
3
3  0
3  0  2
3  0  2
3  0  2
3  0  2  6
3  0  2  6
3  1  2  6
3  1  2  6
3  1  2  6
3  1  0  6
3  1  0  6
0.500000
[root@kp-test01 src]#
```

```
[root@localhost ~]# ./m
0
13  2  5  18  25  14  18  23  12  4  23  10  12  28  24
FIFO:
13
13  2
13  2  5
2  5  18
5  18  25
18  25  14
18  25  14
25  14  23
14  23  12
23  12  4
23  12  4
12  4  10
12  4  10
4  10  28
10  28  24
0.800000
LRU:
13
13  2
13  2  5
18  2  5
18  25  5
18  25  14
18  25  14
18  23  14
18  23  12
4  23  12
4  23  12
4  23  10
12  23  10
12  28  10
12  28  24
0.866667
```

FIFO 之 BLEADY 现象

```
[root@kp-test01 src]# gcc -o test test.c
[root@kp-test01 src]# ./test
1
1 2 3 4 1 2 5 1 2 3 4 5
1
1  2
1  2  3
2  3  4
3  4  1
4  1  2
1  2  5
1  2  5
1  2  5
2  5  3
5  3  4
5  3  4
0.750000
[root@kp-test01 src]# gcc -o test test.c
[root@kp-test01 src]# ./test
1
1 2 3 4 1 2 5 1 2 3 4 5
1
1  2
1  2  3
1  2  3  4
1  2  3  4
1  2  3  4
2  3  4  5
3  4  5  1
4  5  1  2
5  1  2  3
1  2  3  4
2  3  4  5
0.833333
[root@kp-test01 src]#
```

分析：

①运行截图的前两张图表示在随机数生成页面序列的情况下，FIFO 和 LRU 两种页面置换算法的详细置换过程以及最终的缺页率。在第一张图中 FIFO 的缺页率差于 LRU 的缺页率，在第二张图中 FIFO 的缺页率优于 LRU 的缺页率，导致这种情况的根本原因是 LRU 的优秀缺页率是建立在页面序列满足一定的时间局部性和空间局部性的前提之上。由于页面序列是随机生成的，所以当页面序列不具有局部性的时候，LRU 算法可能还不如 FIFO 算法。
但是程序在运行时用到的数据通常都具有一定的时间局部性和空间局部性，所以 LRU 算法很常用。

②在运行截图的最后一张图中，我们可以发现 FIFO 之 BLEADY 现象，同样的页面序列在内存中的页面帧数增大的情况下缺页率不减反增，当发生页面缺页时，FIFO 算法会替换内存中最老的页面（存在时间最长的页面）。有了更多的页面帧，更多的页面会在内存中保留更长的时间。在某些情况下，增加页面帧数会导致本应被替换的页面在内存中停留的时间更长。结果是更多的页面缺页，因为老的页面被保留，而新的页面无法在可用的页面帧中找到位置。所以向系统添加更多资源（在这种情况下是页面帧）可能并不总是会带来预期的性能改善，而选择页面置换算法对结果起着至关重要的作用。

③在实现 LRU 的过程中，我在页面帧数数组 pc 中设置了一个 lastest 表示最近一次加入的时间，当需要选择牺牲页时，选择页面帧数数组 pc 中 lastest 最小的元素作为牺牲页，这样做使得越是最近加入数组的元素越晚被当成牺牲页，当页面序列具有局部性的时候，可以使得重复使用的数据尽可能的不会缺页。这种实现方式符合 LRU 算法的本质，即根据页面的最近使用情况来进行页面替换，以更好地利用程序的局部性原理。

问题：不知道采用什么样的数据结构来保存数据，不知道用什么数据结构来描述页面帧数

解决过程：在看完教材之后，我设置了一个数组来描述页面帧数，对于 FIFO 算法，我把这个数组当成循环队列来使用，队头所指的元素可以看成是在页面帧数中最早进入的元素，同时也当成了牺牲页，队尾所指的元素是待进入的元素。对于 LRU 算法，每次有元素进入数组的时候，都把数组对应元素的 lastest 设置成最新值，在选择牺牲页的时候，将数组中 latest 最小的元素作为牺牲页。