西安交通大学

操作系统专题实验报告

班级：计算机 2101

学号：2212312393

姓名：李浩伟

2023 年 12 月 15 日

目 录

1 open Eule 系统环境实验

1.1 实验目的

（1）熟悉基于鲲鹏架构弹性云服务器 ECS 上 openEuler 操作系统基本系统环境；
（2）理解进程的创建、进程地址空间的概念、父子进程资源的关系；
（3）观察进程调度，了解进程调度的过程，了解孤儿进程和僵尸进程的区别；
（4）理解线程与进程的关系，对等线程间的关系。
（5）理解并运用信号量及其 PV 操作、自旋锁实现线程间的同步互斥。

1.2 实验内容

（1）在华为云上搭建 openEuler 操作系统环境，通过运行 shell 命令查看系统信息以了解并熟悉 openEuler 操作系统。
（2）熟悉操作命令、编辑、编译、运行程序。完成给定程序（图 1.1 所示的程序）的运行验证，多运行程序几次观察结果；去除 wait 后再观察结果并进行理论分析。
（2）在所给程序中添加一个全局变量，在父进程和子进程中分别对这个变量做不同操作并输出操作结果，对结果进行观察并解释，同时输出两种变量的地址观察并分析。
（3）修改程序，在子进程中调用 system 函数和在子进程中调用 exec 族函数以执行自己写的一段程序，在此程序中输出进程 PID，进行比较分析。
（4）线程实验:在进程中给一变量赋初值并创建两个线程，在两个线程中分别对此变量循环五千次以上做不同的操作并输出结果 ；多运行几遍程序观察运行结果，如果发现每次运行结果不同，请解释原因并修改程序解决，考虑如何控制互斥和同步；在线程中调用 system 函数/exec 族函数以执行自己写的一段程序，输出进程 PID 和线程 TID, 进行比较分析。
（5）自旋锁实验：在进程中给一变量赋初值并成功创建两个线程；在两个线程中分别对此变量循环五千次以上做不同的操作（自行设计）并输出结果；使用自旋锁实现互斥和同步。

1.3 实验思想

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

通过配置基于鲲鹏架构弹性云服务器 ECS 上 openEuler 操作系统基本系统环境，运行 shell 命令查看系统信息以达到了解和使用 openEuler 操作系统的目的。

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

1.4 实验步骤

（1）登录华为云,搭建 openEuler 操作系统环境。

（2）编辑、编译、运行给定的程序，多次运行观察实验结果。

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

（3）添加一个全局变量，在父进程和子进程中对这个变量做不同操作，输出操作结果，同时输出两种变量的地址并进行分析。

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

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

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

（7）多运行几遍程序观察运行结果。

（8）完成上述过程的同步和互斥操作。

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

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

PID,进行分析。

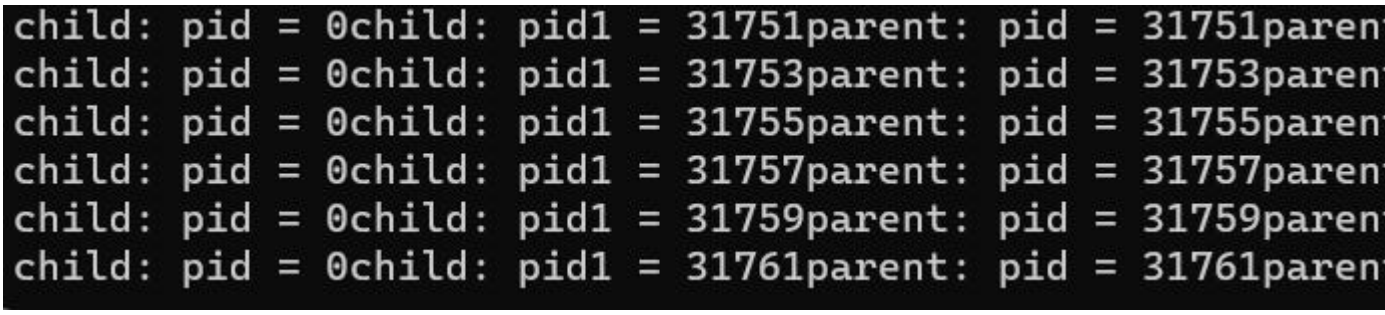
（11）参考实验指导书，编写模拟自旋锁程序代码，补充主函数代码，用自旋锁实现线程间的同步。编译并运行程序，分析运行结果

1.5 测试数据设计

1. 步骤一操作时，一个进程执行 5 次+1，另一个进程赋值位 127
2. 线程相关实验中，一个线程执行 5000 次+1，另一个进程执行 5000 次-1
3. PV 操作实现同步时，两个线程都执行 5000 次+1
4. 自旋锁相关实验，两个线程都执行 5000 次+1

1.6 程序运行初值及运行结果分析

含 wait 函数

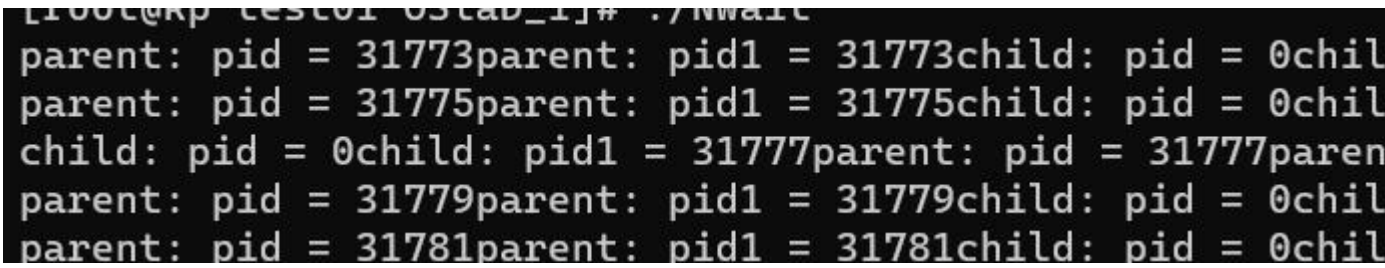


Fork()创建子进程并用 pid 作为 fork()函数的返回值，若返回位负值则 fork 失败退出程序；

若返回为 0，则进入子进程，此时 pid1=getpid()得到的即为子程序的 pid 值;

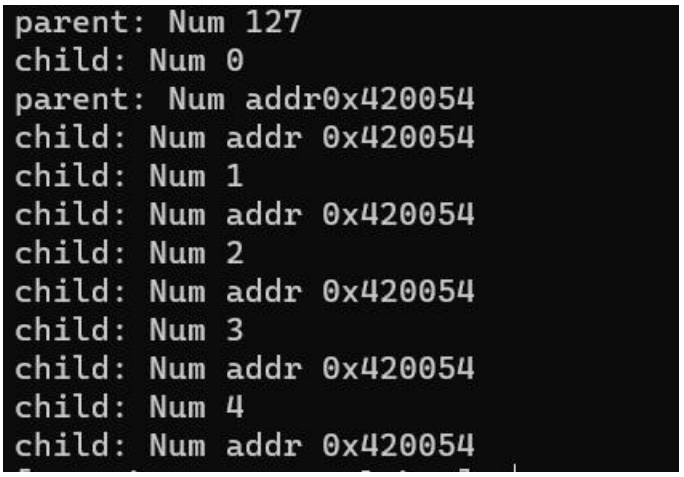
若返回不为 0，则进入父进程，同时 pid 返回的值为子进程的 pid 值，而父进程中的 pid1=getpid()则返回的为父进程的 pid 值；

去除 wait 函数



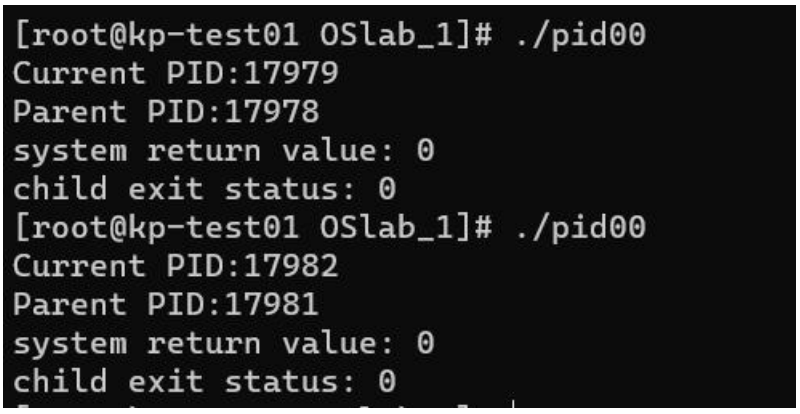
根据运行结果，父子进程的输出顺序与含 wait 函数时相反，这是因为 wait()函数会阻塞父进程，知道它的子进程终止，父进程才可以终止。因此在含 wait 函数的运行结果里，子进程优先于父进程输出；删去 wait 函数后，结果恰好相反。

父子进程对所添加变量进行不同操作



如运行结果所示，在父进程和子进程中对于同一变量的写入时独立的，这说明父子进程是两个独立的进程，互不影响；两进程中对于同一变量的地址是相同的，而该地址是虚拟地址，实际上两个进程中的 Num 的绝对地址是不同的，因而父子进程写入不会相互影响

System



System 函数不会替换当前进程，只是调用系统的命令并返回命令的状态，子进程和父进程分别调用 systeme 函数进而调用程序，再将被调用程序执行结果返回，继续执行当前进程,因此 sys 函数调用后会输出 ret 状态。

Exec

```
[root@kp-test01 OSlab_1]# ./pid00
Current PID:17959
Parent PID:17958
child exit status: 0
[root@kp-test01 OSlab_1]# ./pid00
Current PID:17961
Parent PID:17960
child exit status: 0
[root@kp-test01 OSlab_1]#
```

exec 会将当前进程替换为新的程序继续执行，子进程和父进程分别执行 exec 函数，两个进程都替换为新的程序执行，由于实现了进程替换，子进程抛弃掉原进程，因而再 excel 函数以后的部分不在执行，因此不输出 ret 状态。

Thread

```
[root@kp-test01 OSlab_1]# ./Thread00
Thread1 terminated, temp = 4262
Thread2 terminated, temp = 92
All termintaed, temp = 92
[root@kp-test01 OSlab_1]# ./Thread00
Thread1 terminated, temp = 3368
Thread2 terminated, temp = -369
All termintaed, temp = -369
[root@kp-test01 OSlab_1]# ./Thread00
Thread1 terminated, temp = 5000
Thread2 terminated, temp = 0
All termintaed, temp = 0
[root@kp-test01 OSlab_1]# ./Thread00
Thread1 terminated, temp = 5000
Thread2 terminated, temp = 0
All termintaed, temp = 0
[root@kp-test01 OSlab_1]# ./Thread00
Thread2 terminated, temp = -5000
Thread1 terminated, temp = 0
All termintaed, temp = 0
[root@kp-test01 OSlab_1]# ./Thread00
Thread1 terminated, temp = 4593
Thread2 terminated, temp = 400
All termintaed, temp = 400
[root@kp-test01 OSlab_1]# ./Thread00
Thread1 terminated, temp = 4895
Thread2 terminated, temp = 484
All termintaed, temp = 484
[root@kp-test01 OSlab_1]#
```

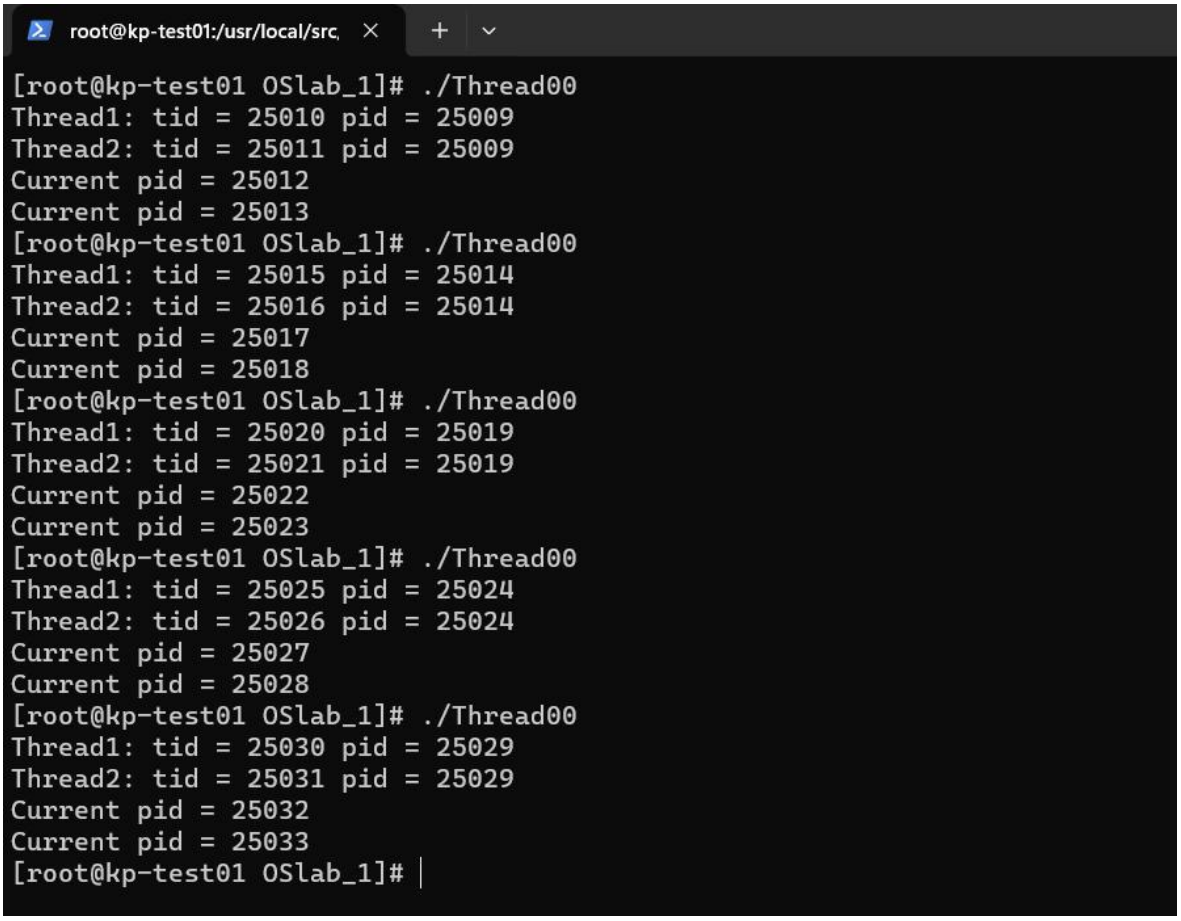
两线程不存在任何同步机制，在运行的时间顺序上是随机的，两线程产生竞争现象，因而多次运行结果不会完全相同。当然也不排除一些特殊情况即其中一个线程运行完毕后，另一个线程继续运行，此时我们最终可以得到一个正确结果。

加入同步和互斥操作

```
[root@kp-test01 OSlab_1]# ./Thread_signal
Thread1 terminated, temp = 5000
Thread2 terminated, temp = 0
All termintaed, temp = 0
[root@kp-test01 OSlab_1]# ./Thread_signal
Thread1 terminated, temp = 5000
Thread2 terminated, temp = 0
All termintaed, temp = 0
[root@kp-test01 OSlab_1]# ./Thread_signal
Thread1 terminated, temp = 5000
Thread2 terminated, temp = 0
All termintaed, temp = 0
[root@kp-test01 OSlab_1]# ./Thread_signal
Thread1 terminated, temp = 5000
Thread2 terminated, temp = 0
All termintaed, temp = 0
[root@kp-test01 OSlab_1]# ./Thread_signal
Thread1 terminated, temp = 5000
Thread2 terminated, temp = 0
All termintaed, temp = 0
[root@kp-test01 OSlab_1]# ./Thread_signal
Thread1 terminated, temp = 5000
Thread2 terminated, temp = 0
All termintaed, temp = 0
[root@kp-test01 OSlab_1]# ./Thread_signal
Thread1 terminated, temp = 5000
Thread2 terminated, temp = 0
All termintaed, temp = 0
```

如运行结果所示，在加入 signal 并进行 PV 操作后，确保了同一时间只有一个线程工作，不会出现竞争现象，因此最后得到正确结果 0。

System



在原进程创建的两个线程共享一个进程，因而两线程得到的进程号相同；同时线程 id 与进程 id 统一编号，不同线程又具有不同 tid，因而两线程的 tid 在原进程的基础上分别加 1 和加 2；而两线程在调用 system 函数时，分别创建了一个新的进程，因而创建了两个不同的进程，pid 也因此不同。

exec 族

```
[root@kp-test01 OSlab_1]# ./exec
Thread1: tid = 26906 pid = 26905
Current pid = 26905
[root@kp-test01 OSlab_1]# ./exec
Thread1: tid = 26909 pid = 26908
Current pid = 26908
[root@kp-test01 OSlab_1]# ./exec
Thread1: tid = 26912 pid = 26911
Current pid = 26911
[root@kp-test01 OSlab_1]# ./exec
Thread1: tid = 26915 pid = 26914
Current pid = 26914
[root@kp-test01 OSlab_1]# ./exec
Thread1: tid = 26918 pid = 26917
Current pid = 26917
[root@kp-test01 OSlab_1]# ./exec
Thread1: tid = 26921 pid = 26920
Current pid = 26920
```

根据输出结果，只有一个线程运行并创建了新进程，这是因为 excel 函数会将当前进程替换为另一个可执行程序，而两线程共处一个进程之下，因而当其中一个线程调用了 excel 函数后，都会先将当前进程替换为另一个可执行程序，并在新程序中继续运行。这意味着原来的线程就不存在了。

自旋锁相关实验

```
[root@kp-test01 OSlab_1]# ./Thread_mutex
Thread1 terminated, temp = 5000
Thread2 terminated, temp = 0
All termintaed, temp = 0
[root@kp-test01 OSlab_1]# ./Thread_mutex
Thread1 terminated, temp = 5000
Thread2 terminated, temp = 0
All termintaed, temp = 0
[root@kp-test01 OSlab_1]# ./Thread_mutex
Thread1 terminated, temp = 5000
Thread2 terminated, temp = 0
All termintaed, temp = 0
[root@kp-test01 OSlab_1]# ./Thread_mutex
Thread1 terminated, temp = 5000
Thread2 terminated, temp = 0
All termintaed, temp = 0
[root@kp-test01 OSlab_1]# ./Thread_mutex
Thread1 terminated, temp = 5000
Thread2 terminated, temp = 0
All termintaed, temp = 0
[root@kp-test01 OSlab_1]# ./Thread_mutex
Thread2 terminated, temp = -1007
Thread1 terminated, temp = 0
All termintaed, temp = 0
```

由运行结果得，两线程操作中加入互斥锁后，最终输出结果始终正确，确保了同一时间只有一个线程工作，消除了竞争现象。

1.7 实验总结

1.7.1 实验中的问题与解决过程

问题 1：在进程相关实验时，在 printf 函数中是否加入\n 会导致输出结果不同

解决方法：在对 wait 函数作用分析的实验中删除 printf 函数中的\n

问题 2：线程相关实验时正常 gcc 编译链接失败

解决方法：编译时在末尾加入-lpthread 以链接到-pthread 库

问题 3：在线程相关实验时无法调用 gettid()函数输出线程的 tid 值

解决方法：将 gettid()函数替换为 syscall(SYS_gettid)以获取当前线程的 tid 值

1.7.2 实验收获

1. exec 族函数会实现进程替换，在进程相关实验时，由于父子进程是两个进程，因此一个进程调用 exec 函数不会影响另一进程；而线程则是在进程以下，多个线程共享一个进程，因此一旦某一线程调用 exec 族函数，整个进程就会替换到另一程序，所有线程都会被终止。

2. 本次实验让我较为深刻的体会到了线程之间的工作机制

1.7.3 意见与建议

希望能有更清晰的指导设计，并增加一些启发式内容

2 进程通信与内存管理

2.1 实验目的

（1）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）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。

（3）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。

（4）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。

2.2 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+ c 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。

（5）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。

（6）根据流程图（如图 2.2 所示）和所给管道通信程序，按照注释里的要求把代码补充完整，运行程序，体会互斥锁的作用，比较有锁和无锁程序的运行结果，分析管道通信是如何实现同步与互斥的。

（7）理解内存分配 FF，BF，WF 策略及实现的思路。

（8）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。

（9）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。

（10）理解页面置换算法 FIFO、LRU 的思想及实现的思路。

（11）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。

（12）运行所实现的算法，并通过对比，分析 2 种算法的优劣。

（13）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。

2.3 实验思想

2.3.1 软中断通信

信号是一种软件中断，可以用来通知进程发生了某些事件，或者让进程执行某些操作。利用 Linux 系统提供的 signal 和 kill 函数，实现进程间通过发送和接收信号的方式进行通信。

2.3.2 进程的管道通信

利用 Linux 系统提供的 pipe 和 lockf 函数，实现进程间通过创建和操作管道的方式进行通信有锁和无锁情况下的通信。

2.3.3 内存的分配与回收

根据 BF、WF、FF 三种算法的逻辑，设计数据结构与函数，实现内存的动态分配和释放。这些函数可以根据需要为不同的数据结构分配合适的内存空间，并在使用完毕后及时释放，以提高内存的利用效率和安全性，同时实现内存紧缩技术。

2.3.4 页面的置换

分页管理是一种内存管理技术，可以将内存和外存划分为固定大小的单元，称为页面和块。当进程访问的页面不在内存中时，就会发生缺页中断，此时需要从外存中调入所需的页面，并替换掉内存中的某个页面。页面置换算法就是决定替换哪个页面的方法，不同的算法有不同的性能和优缺点。利用 FIFO 和 LRU 两种页面置换算法，模拟内存的分页管理。

2.4 实验步骤

（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+ c 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。

（5）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。

（6）根据流程图（如图 2.2 所示）和所给管道通信程序，按照注释里的要求把代码补充完整，运行程序，体会互斥锁的作用，比较有锁和无锁程序的运行结果，分析管道通信是如何实现同步与互斥的。

（7）理解内存分配 FF，BF，WF 策略及实现的思路。

（8）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。

（9）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。

（10）理解页面置换算法 FIFO、LRU 的思想及实现的思路。

（11）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。

（12）运行所实现的算法，并通过对比，分析 2 种算法的优劣。

（13）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。

2.5 测试数据设计

(1) 在内存分配与管理实验中，我对各种特殊情况进行尝试，比如让进程内存之和超过总内存大小、让分配内存两侧夹有小碎片等等。

(2) 在页面置换算法实验中，我在一种算法中通过随机数来生成页面序列，而在另一种算法的执行中将前一步生成的随机序列作为此次的输入序列，以实现两种算法优劣的对比。

2.6 程序运行初值及运行结果分析

2.6.1 软中断通信

根据流程图编制实现软中断通信的程序

在运行结果看来，两种中断包括 5s 后中断所形成的效果是相同的，如果不给予软中断信号，父进程在 5s 后会向子进程发送 kill 信号，子进程打印输出并退出；当给予^C 或^\，父进程想要进行中断，父程序向子进程发送 kill 信号，并等待子进程结束后终止该进程。

```
bloodee@ubuntu:~/os$ ./kill
16 stop test
17 stop test

Child process1 is killed by parent!!

Child process2 is killed by parent!!

Parent process is killed!!
bloodee@ubuntu:~/os$ ./kill
^C
2 stop test
16 stop test

Child process1 is killed by parent!!
17 stop test

Child process2 is killed by parent!!

Parent process is killed!!
bloodee@ubuntu:~/os$ ./kill
^\
3 stop test
17 stop test
16 stop test

Child process2 is killed by parent!!

Child process1 is killed by parent!!

Parent process is killed!!
bloodee@ubuntu:~/os$
```

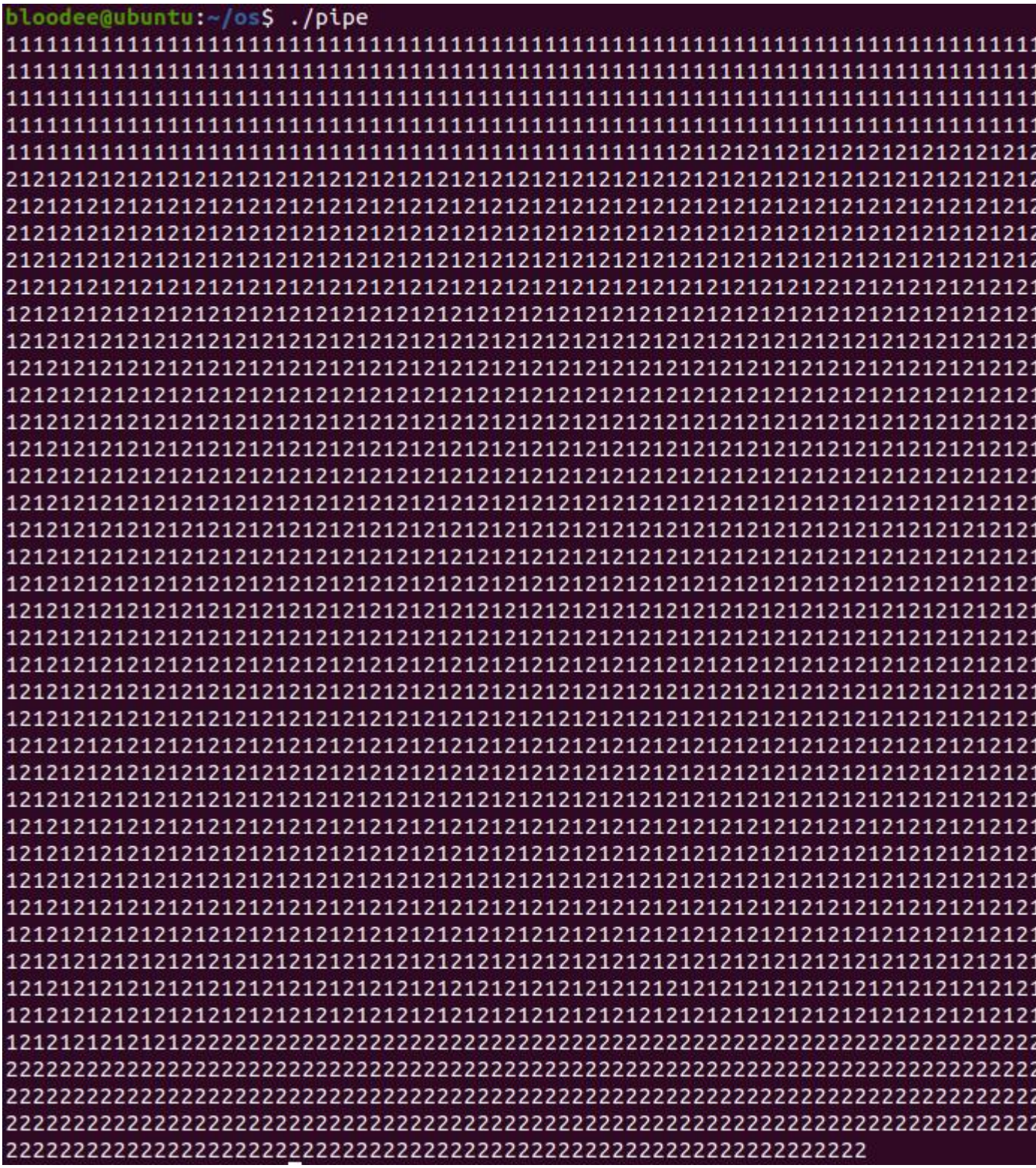
通过 14 号信号值进行闹钟终端

改为闹钟中断后，到达所设时钟时间值时，向父进程发送时钟中断信号，父进程处理后向子进程发送 kill 信号，并等待子进程终止。这种做法类似于前面等待休眠 5s 后自动运行 kill 命令，但通过设置时钟信号，可以对进程进行异步的终止，以更好的控制进程运行。

2.6.2 进程的管道通信

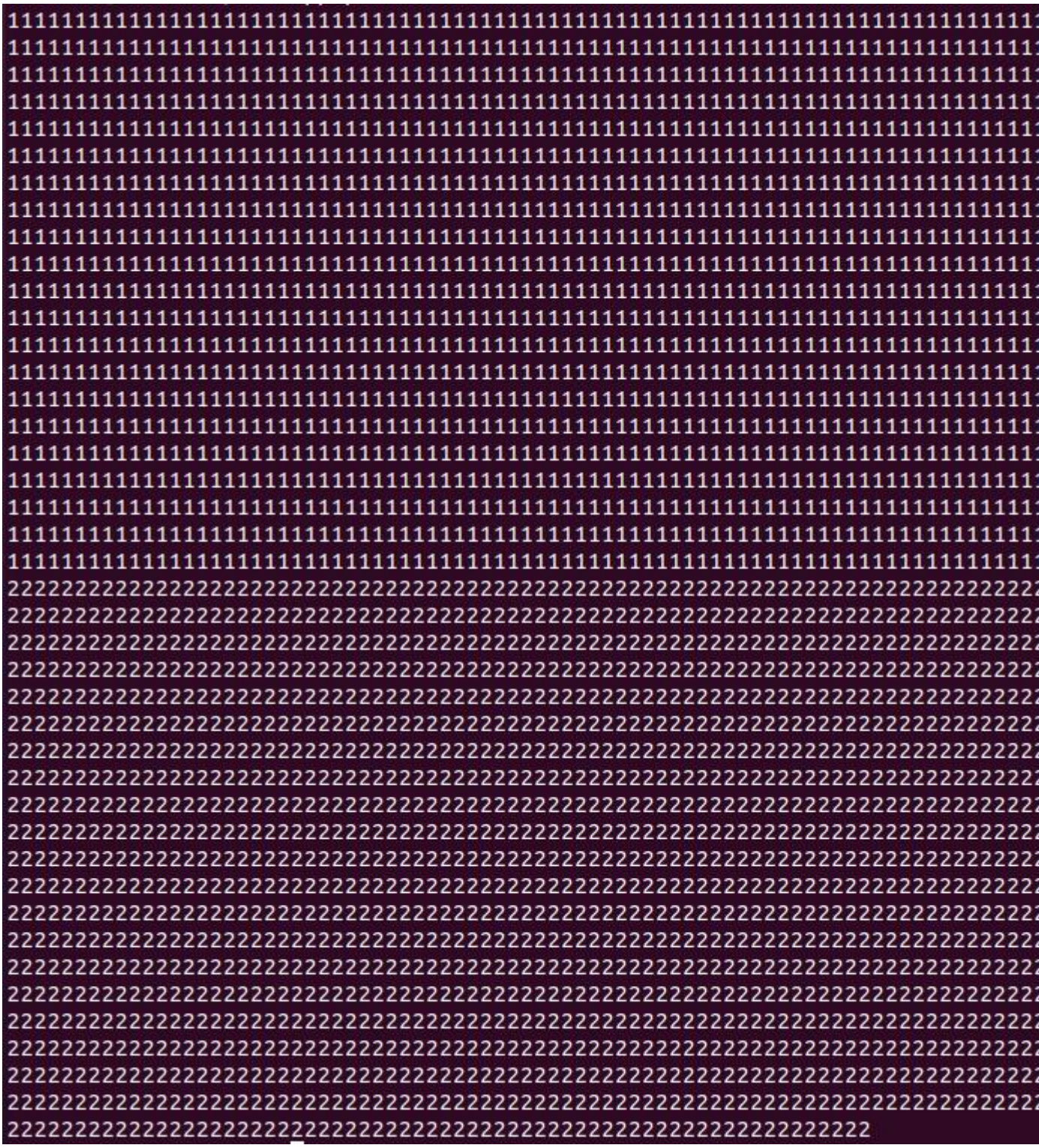
无锁：

无锁状态下，两进程对 InPipe 的写入在时序上是随机的，因而打印结果也是 1 与 2 交错打印。



有锁：

有锁状态下，同一时间内只有一个进程可以写入，因而是先写入了 2000 个 1 而后写入了 2000 个 2



2.6.3 内存的分配与回收

内碎片：

本次实验的实现是通过动态分配和释放内存，唯一产生内碎片的点是当分配内存到所在空闲块后，空闲块剩余大小小于等于 MIN_SLICE 时，该内存会把这整片空闲块占据，而多出的那一小部分即是内碎片

```
------------------------------------------------------------
Free Memory:
          start_addr                size
                   0                1024

Used Memory:
       PID          ProcessName start_addr       size
------------------------------------------------------------

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-02:1020

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

Used Memory:
       PID          ProcessName start_addr       size
         2           PROCESS-02          0       1024
------------------------------------------------------------
```

外碎片：

多个进程被动态分配和释放时，会产生不连续的空闲分区

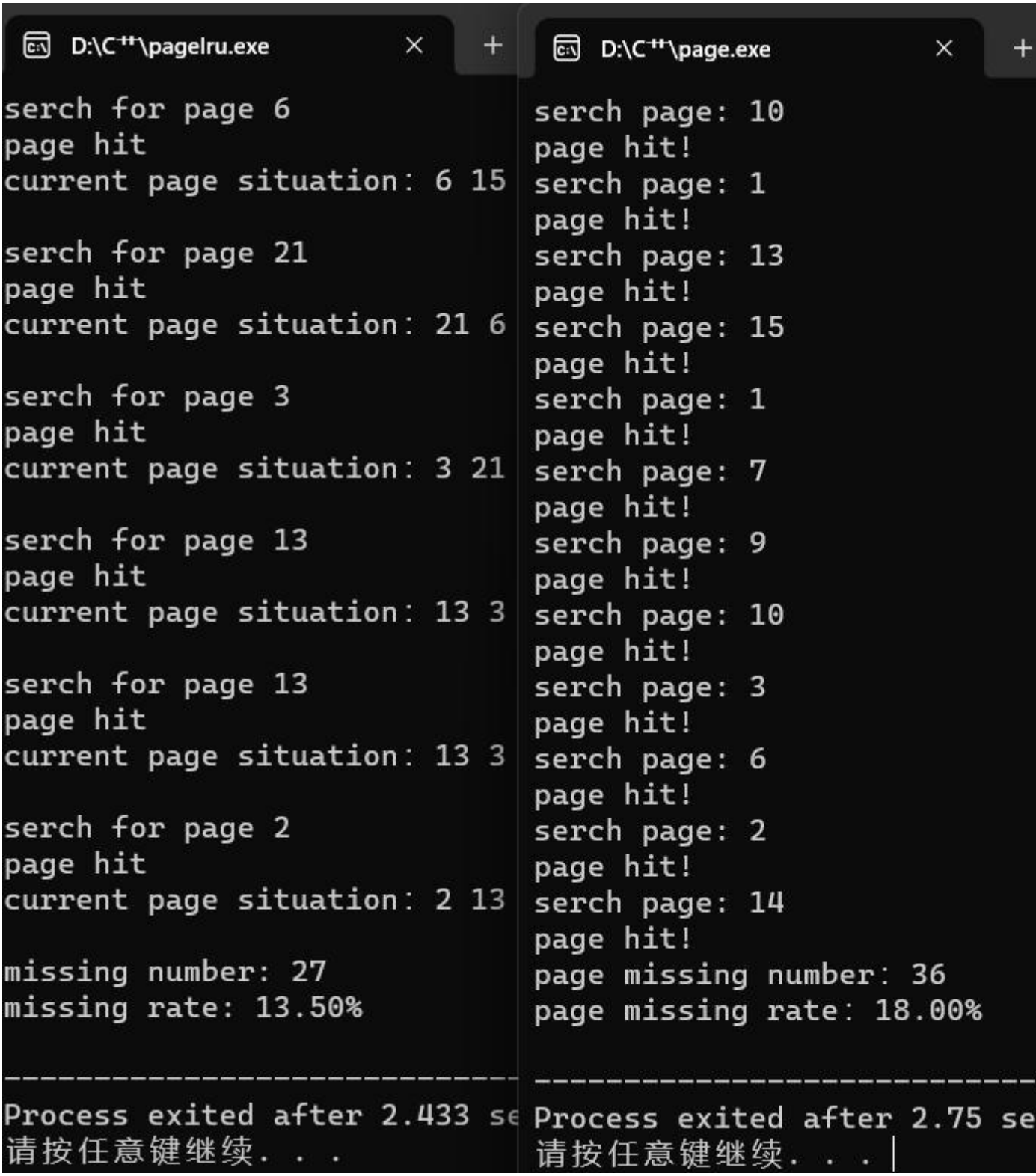
```
Free Memory:
          start_addr                  size
                 467                   111
                 934                    90

Used Memory:
       PID           ProcessName start_addr        size
         5            PROCESS-05        800         134
         4            PROCESS-04        578         222
         2            PROCESS-02        123         344
         1            PROCESS-01          0         123
```

内存紧缩：

通过将内存连续的外碎片合并到一起，来实现内存紧缩，当进程 7 被删除时，其两侧的外碎片会被紧缩到一起。

```
------------------------------------------------------------
Free Memory:
          start_addr                size
                702                 111
                925                  99

Used Memory:
       PID          ProcessName start_addr       size
         7           PROCESS-07        813        112
         5           PROCESS-05        357        345
         4           PROCESS-04        123        234
         3           PROCESS-03          0        123
------------------------------------------------------------

1 - Set memory size (default=1024)
2 - Select memory allocation algorithm
3 - New process
4 - Terminate a process
5 - Display memory usage
0 - Exit
4
Kill Process, pid=7

Process 7 has been killed

1 - Set memory size (default=1024)
2 - Select memory allocation algorithm
3 - New process
4 - Terminate a process
5 - Display memory usage
0 - Exit
5
------------------------------------------------------------
Free Memory:
          start_addr                size
                702                 322

Used Memory:
       PID          ProcessName start_addr       size
         5           PROCESS-05        357        345
         4           PROCESS-04        123        234
         3           PROCESS-03          0        123
------------------------------------------------------------

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

2.6.4 页面的置换


D:\C++\pagelru.exe
serch for page 6
page hit
current page situation: 6 15
serch for page 21
page hit
current page situation: 21 6
serch for page 3
page hit
current page situation: 3 21
serch for page 13
page hit
current page situation: 13 3
serch for page 13
page hit
current page situation: 13 3
serch for page 2
page hit
current page situation: 2 13
missing number: 27
missing rate: 13.50%
Process exited after 2.433 se
请按任意键继续. . .
D:\C++\page.exe
serch page: 10
page hit!
serch page: 1
page hit!
serch page: 13
page hit!
serch page: 15
page hit!
serch page: 1
page hit!
serch page: 7
page hit!
serch page: 9
page hit!
serch page: 10
page hit!
serch page: 3
page hit!
serch page: 6
page hit!
serch page: 2
page hit!
serch page: 14
page hit!
page missing number: 36
page missing rate: 18.00%
Process exited after 2.75 se
请按任意键继续. . .

2.7 页面置换算法复杂度分析

2.7.1 FIFO 算法

因为只需要在队列的头尾进行插入和删除操作，所以时间复杂度为 O(1)。

2.7.2 LRU 算法

在实现时只需要把最近访问过的节点置入当前块页的尾端，再将其他部分按顺序连接起来，故时间复杂度位 O(1)。

2.8 回答问题

2.8.1 软中断通信

（1）你最初认为运行结果会怎么样？写出你猜测的结果。

我最初认为运行结果是 3/2 stop test 在最前，17 stop test 在 Child process 2 is killed 前，16 stop test 在 Child process 1 is killed 前，parent process is killed 在最后，其余顺序可以随意出现。

（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，并对产生该现象的原因进行分析。

```
bloodee@ubuntu:~/os$ ./kill
16 stop test
17 stop test

Child process1 is killed by parent!!

Child process2 is killed by parent!!

Parent process is killed!!
bloodee@ubuntu:~/os$ ./kill
^C
2 stop test
16 stop test

Child process1 is killed by parent!!
17 stop test

Child process2 is killed by parent!!

Parent process is killed!!
bloodee@ubuntu:~/os$ ./kill
^\
3 stop test
17 stop test
16 stop test

Child process2 is killed by parent!!

Child process1 is killed by parent!!

Parent process is killed!!
bloodee@ubuntu:~/os$
```

实际结果大致呈现一个拓补结构，两种中断包括 5s 后中断所形成的效果是相同的，如果不给予软中断信号，父进程在 5s 后会向子进程发送 kill 信号，子进程打印输出并退出；当给予^C 或^\，父进程想要进行中断，父程序向子进程发送 kill 信号，并等待子进程结束后终止该进程。

（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？

```
bloodee@ubuntu:~/os$ ./kill

After 2, send alarm signal to process1

14 stop test
16 stop test

Child process1 is killed by parent!!

After 4, send alarm signal to process2.

14 stop test
17 stop test

Child process2 is killed by parent!!

Parent process is killed!!
```

改为闹钟中断后，到达所设时钟时间值时，向父进程发送时钟中断信号，父进程处理后向子进程发送 kill 信号，并等待子进程终止。这种做法类似于前面等待休眠 5s 后自动运行 kill 命令，但通过设置时钟信号，可以对进程进行异步的终止，以更好的控制进程运行。

（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 命令在程序中使用了两次，作用是向子进程发送中断信号结束此进程，第一次执行后，子进程接受 16 信号并打印 16 stop test，并打印子进程 1 被杀死的信息，第二次执行同理，打印 17 stop test，并打印子进程 2 被杀死的信息。

（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？

进程可以通过 exit 或 return 函数来自主退出，进程自主退出的方式会更好一些，这样可以保证进程数据的完整性，使用 kill 命令在进程外部强制其退出可能导致进程的数据丢失或资源泄漏，但当进程出现异常无法退出时，可能需要 kill 命令来中止进程。

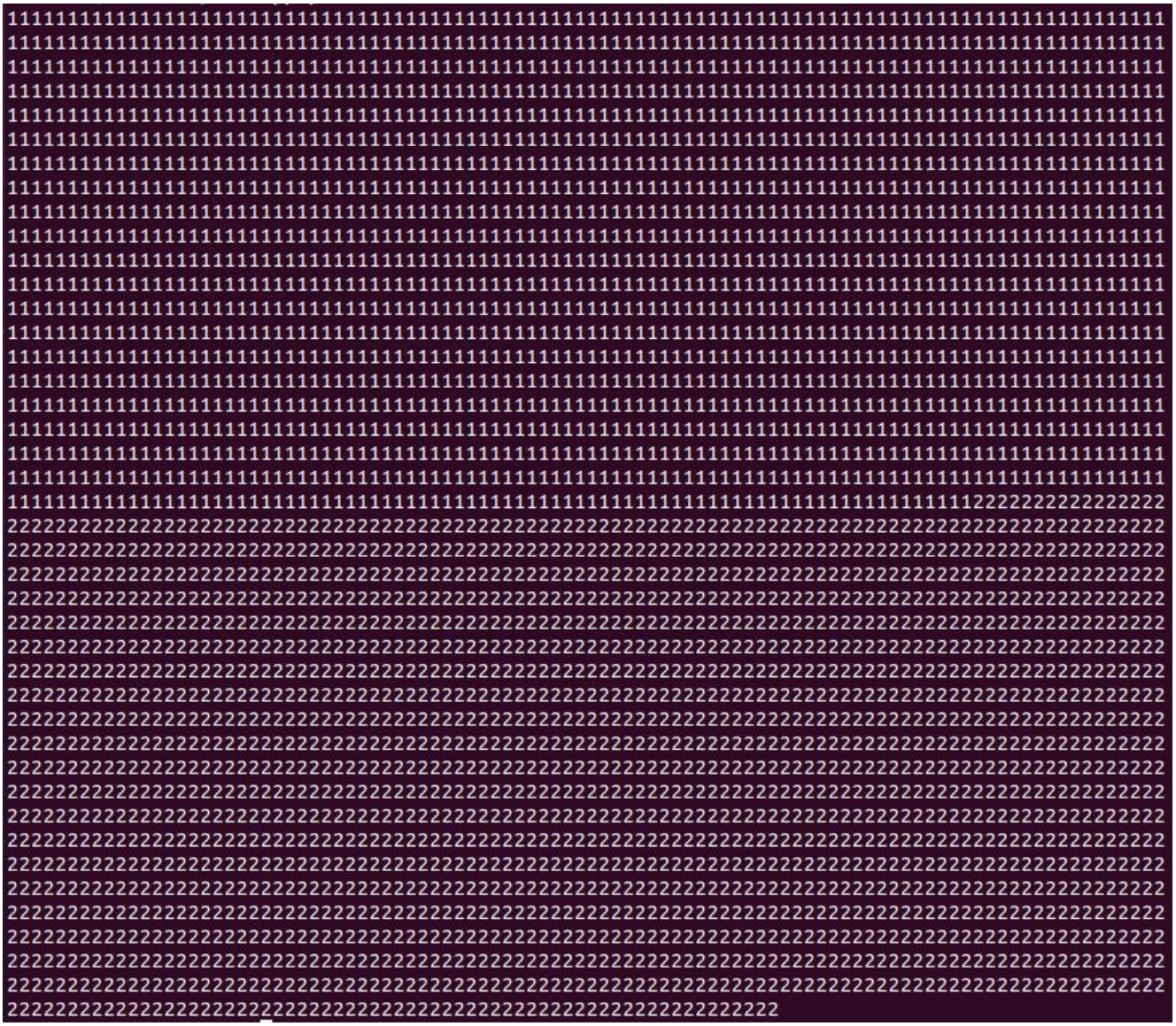
2.8.2 管道通信

(1)你最初认为运行结果会怎么样？

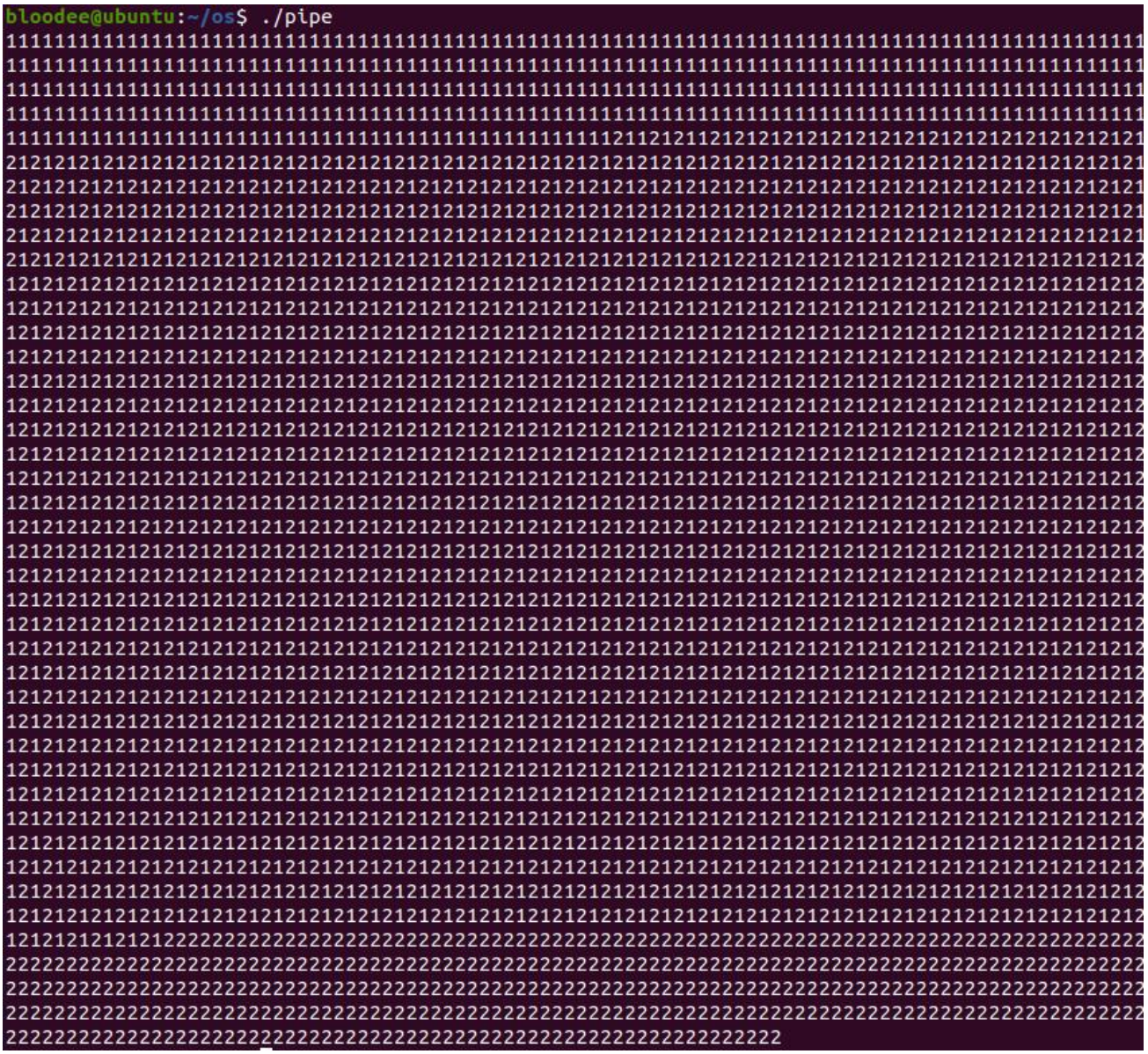
在有锁状态下，我认为会先打印 2000 个 1，再打印 2000 个 2，或者先打印 2000 个 2，再打印 2000 个 1；在无锁状态下，我认为 1 和 2 会交错打印。

(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：



无锁：



实际结果：有锁状态下，同一时间内只有一个进程可以写入，因而是先写入了 2000 个 1 而后写入了 2000 个 2，最后打印结果如图；无锁状态下，两进程对 InPipe 的写入在时序上是随机的，因而打印结果也是 1 与 2 交错打印。

(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？

同步与互斥的实现：在对管道写入前，先对管道用 lockf 函数加锁，等管道写入完毕后，在将管道解锁。

不控制的后果：由于多个进程同时向管道中写入数据，那么数据就很容易发生交错和覆盖，导致数据错误。

2.9 实验总结

2.9.1 实验中的问题与解决过程

遇见的问题：

1、进程的软中断通信时，软中断信号还未发出，子进程退出。

2、进程的软中断通信时，软中断信号发出后，子进程不打印信息直接退出。

3、闹钟中断时，闹钟时间到后，不打印子进程退出信息。

4、在内存的分配与回收时，被释放的内存块无法连接到在其之前的内存块。

5、在内存的分配与回收时，分配的内存至空闲内存小于 MIN_SLICE 时，无法打印内存利用信息。

6、在 WF、BF 整理内存块算法时，无法将内存连续的空闲块连接到一起。

解决的办法：

1、给子进程加入 pause()函数阻塞子进程输出。

2、给子进程加入 signal(SIG_INT,SIG_IGN)以及 signal(SIG_QUIT,SIG_IGN),屏蔽掉应该被父进程接受的两个软中断信号，以使子进程只能接收到父进程 kill 传递过来的软中断信号。

3、将闹钟信号作为父进程的软中断信号，而不是通过 kill 函数传递给子进程。

4、当更换空闲块头节点时，将新更换的节点赋给 free_block.

5、在 display_mem_usage 时，即便空闲块中不存在节点，也应该打印信息，这代表程序中不再有空闲块，并打印出内存分配情况。

6、在 free_mem 时，在对连续地址空闲块的查询时，先调用一次 FF 算法整理空闲块，合并空闲块操作结束后，在进行 BF 或 WF 算法整理空闲块

2.9.2 实验收获

1、较为深入的理解了进程及进程在调度执行和内存空间等方面的特点。

2、较为深入的理解了进程管道通信的特点与管道通信的同步和互斥机制。

3、深入理解了 FF、BF、WF 三种内存分配管理算法的思想与实现策略，同时理解了

内存分配及回收的过程及实现思路以及如何提高内存的分配效率和利用率。

4、深入理解了 FIFO、LRU 两种页面置换算法的思想与实现策略，理解了请求分页系统中、页面置换的实现思路、命中率和缺页率的概念、程序的局部性原理以及虚拟存储的原理。

2.9.3 意见与建议

希望能有更清晰的指导设计，并增加一些启发式内容

2.10 附件

3 文件系统

3.1 实验目的

通过一个简单文件系统的设计，理解文件系统的内部结构、基本功能及实现。

3.2 实验内容

（1）分析 EXT2 文件系统的结构；

（2）基于 Ext2 的思想和算法，设计一个类 Ext2 的多级文件系统，实现 Ext2 文件系统的一个功能子集；

（3）用现有操作系统上的文件来代替硬盘进行硬件模拟。

3.3 实验思想

在分析 Linux 的文件系统的基础上，基于 Ext2 的思想和算法，设计一个类 Ext2 的虚拟多级文件系统，实现 Ext2 文件系统的一个功能子集。并且用现有操作系统上的文件来代替硬盘进行硬件模拟。设计文件系统应该考虑的几个层次：①介质的物理结构；②物理操作——设备驱动程序完成；③文件系统的组织结构（逻辑组织结构）；④对组织结构其上的操作；⑤为用户使用文件系统提供的接口。本实验只涉及后三个层次的内容。

3.4 实验步骤

(1) 定义类 EXT2 文件系统所需的数据结构，包括组描述符、索引结点和目录项。

(2) 实现底层函数，包括分配数据块等 操作。

(3) 实现命令层函数，包括 dir 等操作。

(4) 完成 shell 的设计。

(5) 测试整个文件系统的功能。

3.5 程序运行初值及运行结果分析

格式化硬盘：

调用 initialize_disk 和 initialize_memory 初始化磁盘和内存

```
[root@ /A/B/C/D/]#ls
items          type           mode           size           CreateTime
.              <DIR>          r_w__          ----           Tue Nov 28 20:53:54 2023
..             <DIR>          r_w__          ----           Tue Nov 28 20:50:33 2023
[root@ /A/B/C/D/]#format
Format will erase all the data in the Disk
Are you sure?y/n:
y
Creating the Ext2 file system
The ext2 file system has been installed!

Create Your account

Usrname: bloodee
Password: ansme

Account creating successful!

usr name          : bloodee
volume name       : MYEXT2
disk size         : 4611(blocks)
free blocks       : 4095(blocks)
ext2 file size    : 2305(kb)
block size        : 512(kb)

Log in to the MYEXT2 file system
Or you can press q to exit

Username: bloodee
Password: ansme

Login successful!

[root@ /]#ls
items          type           mode           size           CreateTime
.              <DIR>          r_w__          ----           Tue Nov 28 20:54:54 2023
..             <DIR>          r_w__          ----           Tue Nov 28 20:54:54 2023
[root@ /]#
```

进入目录：

实际上就是改变当前路径，调用 reserch_file 判断是否存在文件，若存在，将路径改为相应路径

```
[root@ /A/B/]#ls
items          type          mode          size          CreateTime
.              <DIR>         r_w__         ----          Tue Nov 28 20:50:28 2023
..             <DIR>         r_w__         ----          Tue Nov 28 20:49:17 2023
C              <DIR>         r_w__           32 bytes    Tue Nov 28 20:50:33 2023
[root@ /A/B/]#cd ..
[root@ /A/]#ls
items          type          mode          size          CreateTime
.              <DIR>         r_w__         ----          Tue Nov 28 20:49:17 2023
..             <DIR>         r_w__         ----          Tue Nov 28 20:28:40 2023
B              <DIR>         r_w__           48 bytes    Tue Nov 28 20:50:28 2023
[root@ /A/]#cd B
[root@ /A/B/]#ls
items          type          mode          size          CreateTime
.              <DIR>         r_w__         ----          Tue Nov 28 20:50:28 2023
..             <DIR>         r_w__         ----          Tue Nov 28 20:49:17 2023
C              <DIR>         r_w__           32 bytes    Tue Nov 28 20:50:33 2023
[root@ /A/B/]#cd .
[root@ /A/B/]#ls
items          type          mode          size          CreateTime
.              <DIR>         r_w__         ----          Tue Nov 28 20:50:28 2023
..             <DIR>         r_w__         ----          Tue Nov 28 20:49:17 2023
C              <DIR>         r_w__           32 bytes    Tue Nov 28 20:50:33 2023
```

创建目录：

先寻找是否存在同名目录，若不存在，获取一个 inode 节点并分配 dir_entry

```
[root@ /A/B/C/]#mkdir D
[root@ /A/B/C/]#ls
items          type          mode          size          CreateTime
.              <DIR>         r_w__         ----          Tue Nov 28 20:50:33 2023
..             <DIR>         r_w__         ----          Tue Nov 28 20:50:28 2023
D              <DIR>         r_w__           32 bytes    Tue Nov 28 20:53:54 2023
[root@ /A/B/C/]#cd D
[root@ /A/B/C/D/]#ls
items          type          mode          size          CreateTime
.              <DIR>         r_w__         ----          Tue Nov 28 20:53:54 2023
..             <DIR>         r_w__         ----          Tue Nov 28 20:50:33 2023
```

创建文件：

实现同创建目录文件类似

```
[root@ /]#touch c
[root@ /]#ls
items          type          mode          size          CreateTime
.              <DIR>         r_w__         ----          Tue Nov 28 20:54:54 2023
..             <DIR>         r_w__         ----          Tue Nov 28 20:54:54 2023
c              <FILE>        r_w_x            0 bytes    Tue Nov 28 20:57:30 2023
[root@ /]#
```

删除目录文件:

调用 reserch_file 检索文件所在当前目录下的目录号，并加载其 inode_entry，将各信息初始化，若目录文件中无其他文件，则调用 remove_block 和 remove_inode,并检查指定 inode 的目录项，如果发现某个数据块中的所有目录项都为空，就删除该数据块并更新 inode 的相关信息，以清理数据块。

```
[root@ /]#mkdir A
[root@ /]#ls
items           type            mode            size            CreateTime
.               <DIR>           r_w__           ----            Tue Nov 28 20:54:54 2023
..              <DIR>           r_w__           ----            Tue Nov 28 20:54:54 2023
c               <FILE>          r_w_x              0 bytes      Tue Nov 28 20:57:30 2023
A               <DIR>           r_w__             32 bytes      Tue Nov 28 21:07:12 2023
[root@ /]#cd A
[root@ /A/]#ls
items           type            mode            size            CreateTime
.               <DIR>           r_w__           ----            Tue Nov 28 21:07:12 2023
..              <DIR>           r_w__           ----            Tue Nov 28 20:54:54 2023
[root@ /A/]#touch B
[root@ /A/]#cd ..
[root@ /]#rmdir A
Directory is not null!
[root@ /]#cd A
[root@ /A/]#rm B
[root@ /A/]#cd ..
[root@ /]#rmdir A
[root@ /]#ls
items           type            mode            size            CreateTime
.               <DIR>           r_w__           ----            Tue Nov 28 20:54:54 2023
..              <DIR>           r_w__           ----            Tue Nov 28 20:54:54 2023
c               <FILE>          r_w_x              0 bytes      Tue Nov 28 20:57:30 2023
```

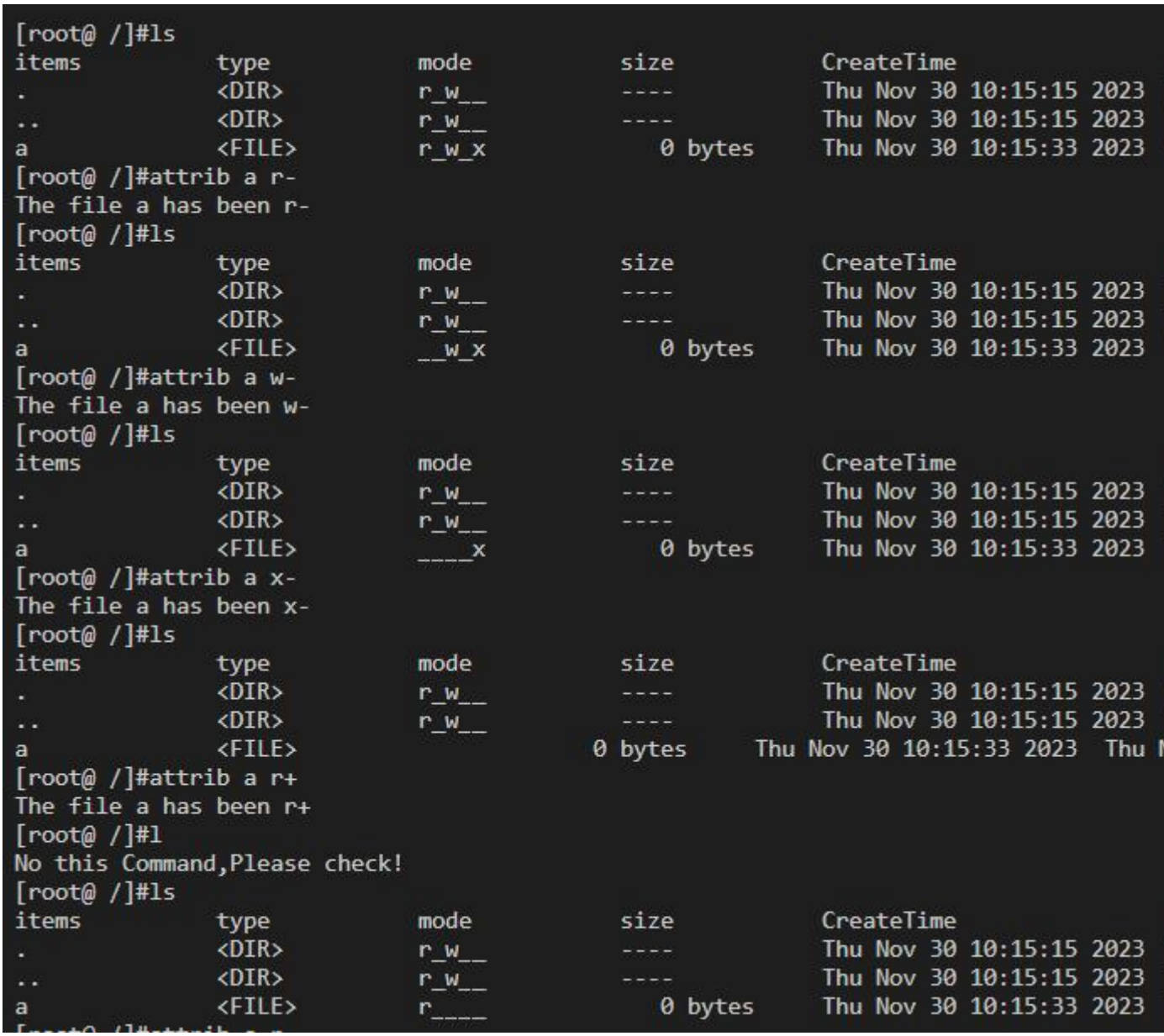
删除文件:

具体实现与 rmdir 函数类似，若文件此时处于打开状态，需索引文件打开表并将对应位置清 0

```
[root@ /]#ls
items           type            mode            size            CreateTime
.               <DIR>           r_w__           ----            Tue Nov 28 20:54:54 2023
..              <DIR>           r_w__           ----            Tue Nov 28 20:54:54 2023
c               <FILE>          r_w_x              0 bytes      Tue Nov 28 20:57:30 2023
[root@ /]#rm c
[root@ /]#ls
items           type            mode            size            CreateTime
.               <DIR>           r_w__           ----            Tue Nov 28 20:54:54 2023
..              <DIR>           r_w__           ----            Tue Nov 28 20:54:54 2023
[root@ /]#
```

修改权限：

寻找文件所在目录项位置，加载其 inode_entry，对所修改权限位按要求取与或取或操作



打开文件：

调用 reserch_file 找到文件所在当前目录的位置，若寻找成功，检测该文件是否处于打开状态，若不是，则打开该文件

```
[root@ /]#ls
items          type          mode          size          CreateTime
.              <DIR>         r_w__         ----          Tue Nov 28 20:54:54 2023
..             <DIR>         r_w__         ----          Tue Nov 28 20:54:54 2023
sss            <FILE>        r_w_x            0 bytes    Tue Nov 28 21:18:54 2023
[root@ /]#open sss
File sss opened!
[root@ /]#open sss
The file sss has opened!
```

关闭文件：

调用 reserch_file 找到文件所在当前目录的位置，若寻找成功，检测该文件是否处于打开状态，若是，则关闭该文件

```
[root@ /]#ls
items          type          mode          size          CreateTime
.              <DIR>         r_w__         ----          Tue Nov 28 20:54:54 2023
..             <DIR>         r_w__         ----          Tue Nov 28 20:54:54 2023
sss            <FILE>        r_w_x            0 bytes    Tue Nov 28 21:18:54 2023
[root@ /]#open sss
File sss opened!
[root@ /]#close sss
File sss closed!
[root@ /]#close sss
The file sss has not been opened!
```

读取文件：

调用 reserch_file 找到文件所在当前目录的位置，若寻找成功，检测该文件是否处于打开状态，若是，则读取文件信息到缓冲区，并打印缓冲区内容

```
[root@ /]#ls
items          type          mode          size          CreateTime
.              <DIR>         r_w__         ----          Tue Nov 28 20:54:54 2023
..             <DIR>         r_w__         ----          Tue Nov 28 20:54:54 2023
sss            <FILE>        r_w_x            0 bytes    Tue Nov 28 21:18:54 2023
[root@ /]#open sss
File sss opened!
[root@ /]#write sss
#
llainwiresodope
#
[root@ /]#read sss
llainwiresodope
```

写入文件：

调用 reserch_file 找到文件所在当前目录的位置，若寻找成功，检测该文件是否处于

打开状态，若是，则向缓冲区处写入数据，在调用 memcpy 函数将缓冲区数据冲入数据块，实现文件的覆写

```
[root@ /]#ls
items           type            mode            size            CreateTime
.               <DIR>           r_w__           ----            Tue Nov 28 20:54:54 2023
..              <DIR>           r_w__           ----            Tue Nov 28 20:54:54 2023
sss             <FILE>          r_w_x              0 bytes      Tue Nov 28 21:18:54 2023
[root@ /]#open sss
File sss opened!
[root@ /]#write sss
#
llainwiresodope
#
[root@ /]#read sss
llainwiresodope

[root@ /]#write sss
#
aaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaa
#
[root@ /]#read sss
aaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaa
```

展示当前目录内容：

加载当前路径目录的 inode_entry，顺次检索目录下文件的各项属性，并按照一定格式进行打印

```
[root@ /]#mkdir A
[root@ /]#mkdir B
[root@ /]#mkdir C
[root@ /]#ls
items           type            mode            size            CreateTime
.               <DIR>           r_w__           ----            Tue Nov 28 20:54:54 2023
..              <DIR>           r_w__           ----            Tue Nov 28 20:54:54 2023
sss             <FILE>          r_w_x             38 bytes      Tue Nov 28 21:18:54 2023
A               <DIR>           r_w__             32 bytes      Tue Nov 28 21:27:34 2023
B               <DIR>           r_w__             32 bytes      Tue Nov 28 21:27:36 2023
C               <DIR>           r_w__             32 bytes      Tue Nov 28 21:27:38 2023
```

检查磁盘：

打印磁盘各项属性

```
[root@ /]#ckdisk
usr name          : bloodee
volume name       : MYEXT2
disk size         : 4611(blocks)
free blocks       : 4091(blocks)
ext2 file size    : 2305(kb)
block size        : 512(kb)
```

登入文件系统:

匹配用户名与密码，若正确则进入系统，错误可选择重新输入或退出系统

```
[root@ /]#login

Log in to the MYEXT2 file system
Or you can press q to exit

Username: bloodee
Password: ansme

Login successful!

[root@ /]#login

Log in to the MYEXT2 file system
Or you can press q to exit

Username: q
PS C:\Users\Lenovo\Desktop\OSLab\Lab_3\src_1>
```

修改密码:

加载组描述符，输入旧密码和新密码来修改密码

```
PS C:\Users\Lenovo\Desktop\OSLab\Lab_3\src_1> ./ext2

Log in to the MYEXT2 file system
Or you can press q to exit

Username: bloodee
Password: ansme

Login successful!

[root@ /]#password
No this Command,Please check!
[root@ /]#pwd
Please enter the original password: ansme
Please enter the new password: assme
[root@ /]#login

Log in to the MYEXT2 file system
Or you can press q to exit

Username: bloodee
Password: assme

Login successful!

[root@ /]#
```

3.6 实验总结

3.6.1 实验中的问题及解决过程

在设计文件系统时数据结构遇到的问题

1.inode 中的访问、修改和创建时间的获取难以用一 4 字节无符号数表示，于是我将其类型修改为字符串，并调整 inode 大小

2.在合并 superblock 和 group_desc 内容时，需对一些重复信息进行筛选

3.斟酌选用哪种数据结构来表示数据快位图和 inode 块位图，结果应采取简单易索引的数组来表示

实现底层函数时遇到的挑战问题

1.在创建文件时应对文件采取相应的初始化操作，若在其余函数中都实现相同的重

复操作会使代码非常繁琐，因而加入函数 dir_prepare 来实现这一操作

2.在初始化磁盘和内存时，对于各结构的加载和更新需要有一个清晰的操作过程，不然很容易出现难以发现的问题。

调试代码时遇到的挑战问题

问题：

1.使用 mkdir 建立文件夹时会清除当前文件中其他所有的文件，且所期待建立的文件夹也不会建立。

2.Format 格式化硬盘时，存储空间等信息恢复原样，但所创建过的文件内容和读写控制被情况，但文件仍旧存在。

3.在修改文件的 LastWritetime 时，无法对文件的上一级目录的该信息进行同步修改。

4.在进入二级目录时，无法成功创建文件。

5.利用 rmdir 删除目录时，若该目录中含有其他文件，会删除失败且进入一个错误的路径。

原因及解决的方法：

1.mkdir 函数调用时将当前文件夹的所有文件包括其自身在内全部被初始化，经修改后功能正常。

2.在 initializedisk 函数中没有格式化索引节点的的各项信息，导致又一些信息残存在系统又无法使用，经过正确填充后问题解决。

3.添加上级目录名称变量以及索引节点变量，以在修改时可以实现对上一级目录信息的修改。

4.我在创建文件函数中修改时间时加载了新的 inode_entry，导致原用于创建文件所加载到缓冲区的 inode_entry 被覆盖，导致创建出现问题，经调整后问题解决。

在 rmdir 中条件判断出现问题，导致不断递归的进行 rmdir，出现一些难以调试的问题，于是将 rmdir 函数调整为：若目录中含有其他文件，则不能进行删除操作。

3.6.2 实验收获

(1) 加深了对 EXT2 文件系统的理解和掌握，探究了其内部的数据结构和算法，体会了其设计的优势和局限。

(2) 提高了自己的编程能力和调试能力，学习了如何使用 C 语言实现一个复杂的系统设计

(3) 基于对 EXT2 文件系统的设计，我了解了更多文件系统及其它系统设计的相关知识，从中汲取了很多思想。

3.6.3 意见与建议

(1) 多给一些相关参考信息

(2) 设计流程可以更加明确