

西安交通大学

操作系统专题实验报告

班级： 计算机 005

学号： 2201412018

姓名： 许凌皓

2022 年 12 月 15 日

目 录

1.	openEuler 系统环境实验.....	1
1.1.	实验目的	1
1.2.	实验内容	1
1.3.	实验思想	1
1.4.	实验步骤	2
1.5.	测试数据设计.....	4
1.6.	在进程中创建线程.....	5
1.7.	程序运行初值及运行结果分析.....	5
1.8.	实验总结	11
	1.8.1. 实验中的问题与解决过程.....	11
	1.8.2. 实验收获.....	11
	1.8.3. 意见与建议.....	12
1.9.	附件	12
	1.9.2. 附件 1 程序.....	12
	1.9.1. 附件 2 Readme	21
2.	进程通信与内存管理.....	24
2.1.	实验目的	24
2.2.	实验内容	24
2.3.	实验思想	25
2.4.	实验步骤	26
2.5.	测试数据设计.....	29
2.6.	程序运行初值及运行结果分析.....	29
2.7.	页面置换算法复杂度分析.....	35
2.8.	回答问题	35
	2.8.1. 软中断通信	35
	2.8.2. 管道通信	39
2.9.	实验总结	40
	2.9.1. 实验中的问题与解决过程.....	40
	2.9.2. 实验收获.....	40
	2.9.3. 意见与建议.....	41
2.10.	附件	41
	2.10.1. 附件 1 程序.....	41

2.10.2.	附件 2 Readme	49
3.	Linux 的动态模块与设备驱动	52
3.1.	实验目的	52
3.2.	实验内容	52
3.3.	实验思想	52
3.4.	实验步骤	54
3.5.	程序运行初值及运行结果分析.....	58
3.6.	实验总结	60
3.6.1.	实验中的问题与解决过程.....	60
3.6.2.	实验收获.....	64
3.6.3.	意见与建议.....	64
3.7.	附件	64
3.7.1.	附件 1 程序.....	64
3.7.2.	附件 2 Readme	73

1. openEuler 系统环境实验

1.1. 实验目的

1.1.1. 进程相关编程实验

- a) 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是什么
- b) 观察进程调度中的全局变量改变，输出父子进程共享变量地址了解物理地址与虚地址概念
- c) 在子进程中调用 `system` 函数
- d) 在子进程中调用 `exec` 族函数

1.1.2. 线程相关编程实验

创建两个线程运行后体会线程共享进程信息、线程对共享变量操作中同步与互斥的知识。

1.2. 实验内容

1.2.1. 进程相关编程实验

(1) 熟悉操作命令、编辑、编译、运行程序。完成操作系统原理课程教材 P103 作业 3.7 (采用图 3-32 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) 扩展图 3-32 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，同时输出两种变量的地址观察并分析；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` 族函数执行自己写的一段程序，在此程序中输出进程 PID 进行比较并说明原因。

1.2.2. 线程相关编程实验

- 1、在进程中给一变量赋初值并创建两个线程；
- 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。

1.3. 实验思想

进程的定义：进程的经典定义就是一个执行中程序的实例，是计算机科学中最深刻、最成

功的概念之一。

假象： 在现代系统上运行一个程序时，我们会得到一个假象，就好像我们的程序是系统当中运行的唯一程序一样。我们的程序好像独占的使用处理器和内存。处理器就好像是无间断地一条接一条的执行我们程序中的指令。最后，我们程序中的数据和代码好像是系统中内存的唯一对象。然而，这些都是假象，都是进程带给我们的。

真相： 关键在于进程是轮流使用处理器的。每个进程执行它的流一部分，然后被抢占（暂时挂起），然后轮到其他进程。对于一个运行在这些进程之一的上下文程序，它看上去像是在独占的使用处理器。即多个程序同时运行时，并不是某个程序从开始到结束连续的进行，而是某个程序执行一部分指令后，先暂时挂起，轮到另一个程序执行一部分，依此类推。

父进程与子进程： 父进程是通过 fork 函数创建一个新的子进程。

新建的子进程几乎但不完全与父进程相同。子进程得到与父进程用户级虚拟地址相同的（但是独立的）一份副本，包括代码和数据段、堆、共享库以及用户栈。

fork 函数只调用一次，却返回两次：一次是在调用进程（父进程）中，一次是在新建的子进程中。在父进程中，fork 返回子进程的 PID。在子进程中，fork 返回 0。因为子进程的 PID 总是为非零，返回值就提供一个明确的方法来分辨程序是在父进程中执行还是在子进程中执行。

线程具有许多传统进程所具有的特征，故又称为轻型进程 (Light—Weight Process) 或进程元；而把传统的进程称为重型进程 (Heavy—Weight Process)，它相当于只有一个线程的任务。在引入了线程的操作系统中，通常一个进程都有若干个线程，至少包含一个线程。

线程与进程的区别：

进程是操作系统资源分配的基本单位，而线程是处理器任务调度和执行的基本单位

资源开销： 每个进程都有独立的代码和数据空间（程序上下文），程序之间的切换会有较大的开销；线程可以看做轻量级的进程，同一类线程共享代码和数据空间，每个线程都有自己独立的运行栈和程序计数器（PC），线程之间切换的开销小。

包含关系： 如果一个进程内有多个线程，则执行过程不是一条线的，而是多条线（线程）共同完成的；线程是进程的一部分，所以线程也被称为轻权进程或者轻量级进程。

内存分配： 同一进程的线程共享本进程的地址空间和资源，而进程之间的地址空间和资源是相互独立的

影响关系： 一个进程崩溃后，在保护模式下不会对其他进程产生影响，但是一个线程崩溃整个进程都死掉。所以多进程要比多线程健壮。

执行过程： 每个独立的进程有程序运行的入口、顺序执行序列和程序出口。但是线程不能独立执行，必须依存在应用程序中，由应用程序提供多个线程执行控制，两者均可并发执行

1.4. 实验步骤

1.4.1. 搭建华为实验环境

1.4.1.1. 服务器配置



1.4.1.2. 远程控制服务器

```
yunkino@yunkino-virtual-machine:~$ ssh root@121.36.48.125
The authenticity of host '121.36.48.125 (121.36.48.125)' can't be established.
ED25519 key fingerprint is SHA256:F258BwzLqpt6K98b+69iVNi0pwhsLhFqxFWHaxvga+E.
This key is not known by any other names
Are you sure you want to continue connecting (yes/no/[fingerprint])? y
Please type 'yes', 'no' or the fingerprint: yes
Warning: Permanently added '121.36.48.125' (ED25519) to the list of known hosts.

Authorized users only. All activities may be monitored and reported.
root@121.36.48.125's password:

Welcome to Huawei Cloud Service

Welcome to 4.19.90-2003.4.0.0036.oe1.aarch64

System information as of time: 2022年 10月 24日 星期一 13:04:57 CST

System load: 0.70
Processes: 119
Memory used: 10.0%
Swap used: 0.0%
Usage On: 9%
IP address: 192.168.1.133
Users online: 1
```

使用 ssh 命令控制服务器

```
yunkino@yunkino-virtual-machine:~$ scp /home/yunkino/文档/lab1/process1.c root@121.36.48.125:
Authorized users only. All activities may be monitored and reported.
root@121.36.48.125's password:
process1.c 100% 638 22.6KB/s 00:00
yunkino@yunkino-virtual-machine:~$ scp /home/yunkino/文档/lab1 root@121.36.48.125:
Authorized users only. All activities may be monitored and reported.
root@121.36.48.125's password:
/home/yunkino/文档/lab1: not a regular file
yunkino@yunkino-virtual-machine:~$ scp -r /home/yunkino/文档/lab1 root@121.36.48.125:
Authorized users only. All activities may be monitored and reported.
root@121.36.48.125's password:
ReadMe.md 100% 2909 31.5KB/s 00:00
system_child_process 100% 16KB 310.6KB/s 00:00
pthread4.c 100% 1308 28.1KB/s 00:00
pthread3 100% 16KB 178.1KB/s 00:00
process4.c 100% 679 14.6KB/s 00:00
process4 100% 16KB 175.5KB/s 00:00
pthread3.c 100% 1253 27.1KB/s 00:00
process3.c 100% 655 13.3KB/s 00:00
pthread1 100% 16KB 175.4KB/s 00:00
pthread2.c 100% 1190 25.8KB/s 00:00
process3 100% 16KB 173.3KB/s 00:00
```

使用 scp 命令将本地写好的文件传输到远程服务器

1.4.2. 进程相关编程实验

1.4.2.1. 课程教材上 P104 页图 3-32 所示程序

完成课程教材上 P104 页图 3-32 所示程序, 多次运行验证

所对应代码为 process1.c

使用命令 `gcc process1.c -o process1` 编译, `./process1` 执行。

1.4.2.2. 扩展程序

所对应代码为 process2.c

定义全局变量, 在父进程与子进程中分别对其进行操作。

定义 `int` 类型变量 `global` 初始化为 0, 在父进程中对其进行++操作, 在子进程中对其进行--操作。

在 `return` 前增加对全局变量*2 的操作。

1.4.2.3. 在子进程中调用 `system` 函数运行自己写的程序

所对应代码为 process3.c 和 `system_child_process.c`

在子进程中使用 `system` 函数调用 `system_child_process` 程序, 在 `system_child_process` 程序中获取当前进程的 `pid` 值并返回 `pid` 值。

1.4.2.4. 在子进程中调用 `exec` 函数运行自己写的程序

所对应代码为 process4.c 和 `exec_child_process.c`

在子进程中使用 `exec` 函数调用 `exec_child_process` 程序, 在 `exec_child_process` 程序中获取当前进程的 `pid` 值并返回 `pid` 值。

1.4.2.5. 在进程中创建线程

所对应代码为 `pthread1.c`

在进程中给变量 `global` 赋初值 0 并创建两个线程, 在线程一中对 `global` 进行++操作, 在线程二中对 `global` 进行自加 2 操作。

在两个线程中 `global` 循环五千次操作并输出结果

1.4.2.6. 在线程中实现互斥与同步

所对应代码为 `pthread2.c`

在线程中实现互斥与同步

1.4.2.7. 在线程中调用 `system` 函数

所对应代码为 `pthread3.c` 和 `system_pthread.c`

在线程一二中分别调用 `system` 函数, 输出进程的 `pid`, 线程的 `tid`。

1.4.2.8. 在线程中调用 `exec` 函数

所对应代码为 `pthread4.c` 和 `exec_pthread.c`

在线程一二中分别调用 `exec` 函数, 输出进程的 `pid`, 线程的 `tid`。

1.5. 测试数据设计

1.4.3. 父子进程对同一全局变量进行操作

所对应代码为 process2.c

定义全局变量, 在父进程与子进程中分别对其进行操作。

定义 int 类型变量 global 初始化为 0, 在父进程中对其进行++操作, 在子进程中对其进行--操作。

在 return 前增加对全局变量*2 的操作。

1.6. 在进程中创建线程

在进程中给变量 global 赋初值 0 并创建两个线程, 在线程一中对 global 进行++操作, 在线程二中对 global 进行自加 2 操作。

在两个线程中 global 循环五千次操作并输出结果。

1.7. 程序运行初值及运行结果分析

1.7.1. 进程相关编程实验

1.7.1.1. 课程教材上 P104 页图 3-32 所示程序

完成课程教材上 P104 页图 3-32 所示程序, 多次运行验证

所对应代码为 process1.c

使用命令 gcc process1.c -o process1 编译, ./process1 执行

```
[root@kp-test01 lab1]# gcc process1.c -o process1
[root@kp-test01 lab1]# ./process1
parent: pid = 3362
child: pid = 0
parent: pid1 = 3361
child: pid1 = 3362
[root@kp-test01 lab1]# ./process1
parent: pid = 3376
child: pid = 0
parent: pid1 = 3375
child: pid1 = 3376
[root@kp-test01 lab1]# ./process1
parent: pid = 3390
child: pid = 0
parent: pid1 = 3389
child: pid1 = 3390
[root@kp-test01 lab1]# ./process1
parent: pid = 3404
parent: pid1 = 3403
child: pid = 0
child: pid1 = 3404
[root@kp-test01 lab1]# ./process1
parent: pid = 3418
child: pid = 0
parent: pid1 = 3417
child: pid1 = 3418
[root@kp-test01 lab1]#
```

从运行结果来看既有可能 child 先执行, 又有可能 parent 先执行。


```
[root@kp-test01 lab1]# gcc process1.c -o process1
[root@kp-test01 lab1]# ./process1
parent: pid = 3561
child: pid = 0
parent: pid1 = 3560
child: pid1 = 3561
[root@kp-test01 lab1]# ./process1
parent: pid = 3575
child: pid = 0
parent: pid1 = 3574
child: pid1 = 3575
[root@kp-test01 lab1]# ./process1
parent: pid = 3589
parent: pid1 = 3588
child: pid = 0
child: pid1 = 3589
[root@kp-test01 lab1]# ./process1
parent: pid = 3603
child: pid = 0
parent: pid1 = 3602
child: pid1 = 3603
[root@kp-test01 lab1]# ./process1
parent: pid = 3617
child: pid = 0
parent: pid1 = 3616
child: pid1 = 3617
[root@kp-test01 lab1]#
```

去除 wait() 再运行

在去掉 wait() 后, 同样也是既有可能 child 先执行, 又有可能 parent 先执行。

由于 fork() 函数会将父进程复制产生子进程, 并同时执行父进程与子进程, 所以才会出现结果中, child 和 parent 进程交错输出的情况。

对于父进程, fork() 函数返回子进程的 pid, 对于子进程, fork() 函数返回的是 0。getpid() 返回当前进程的 pid 值, 所以 parent 的 pid 和 child 的 pid1 值相同, 为子进程的 pid 值。

在去掉 wait() 前后, 都是既有可能 parent 先输出, 又有可能 child 先输出, 因为父子进程的执行顺序是不确定的。wait() 的作用是让父进程挂起等待子进程结束。所以 wait() 放在父进程输出之前才能确保子进程先输出。

1.7.1.2. 扩展程序

所对应代码为 process2.c

定义全局变量, 在父进程与子进程中分别对其进行操作。

定义 int 类型变量 global 初始化为 0, 在父进程中对其进行++操作, 在子进程中对其进行--操作。

```
[root@kp-test01 lab1]# gcc process2.c -o process2
[root@kp-test01 lab1]# ./process2
gobal = 1
gobal = -1
gobal address = 0x42005c
gobal address = 0x42005c
parent: pid = 3032
child: pid = 0
parent: pid1 = 3031
child: pid1 = 3032
[root@kp-test01 lab1]# ./process2
gobal = 1
gobal = -1
gobal address = 0x42005c
gobal address = 0x42005c
parent: pid = 3088
child: pid = 0
parent: pid1 = 3087
child: pid1 = 3088
```

在 return 前增加对全局变量*2 的操作

```
[root@kp-test01 lab1]# gcc process2.c -o process2
[root@kp-test01 lab1]# ./process2
gobal = 1
gobal = -1
gobal address = 0x420054
gobal address = 0x420054
parent: pid = 3659
child: pid = 0
parent: pid1 = 3658
child: pid1 = 3659
gobal = 2
gobal = -2
gobal address = 0x420054
gobal address = 0x420054
[root@kp-test01 lab1]# ./process2
gobal = 1
gobal = -1
gobal address = 0x420054
gobal address = 0x420054
parent: pid = 3673
child: pid = 0
parent: pid1 = 3672
child: pid1 = 3673
gobal = 2
gobal = -2
gobal address = 0x420054
gobal address = 0x420054
```

从结果可以看出来 global 全局变量地址是相同的, 原因在于这个地址是虚地址, 而因为子进程的创建是复制了父进程的虚拟地址空间的, 因为这两个变量的虚地址也是一样的。

如果父进程对其进行修改, 那么在真正的物理内存中会拷贝一份 global 的副本, 父进程中修改的其实是副本。对子进程而言也是如此。

1.7.1.3. 在子进程中调用 system 函数运行自己写的程序

所对应代码为 process3.c 和 system_child_process.c

在子进程中使用 system 函数调用 system_child_process 程序, 在 system_child_process 程序中获取当前进程的 pid 值并返回 pid 值。

```
[root@kp-test01 lab1]# gcc system_child_process.c -o system_child_process
[root@kp-test01 lab1]# gcc process3.c -o process3
[root@kp-test01 lab1]# ./process3
parent: pid = 3855
child: pid1 = 3855
parent: pid1 = 3854
在子进程中调用system()函数, pid为3856
[root@kp-test01 lab1]# ./process3
parent: pid = 3870
parent: pid1 = 3869
child: pid1 = 3870
在子进程中调用system()函数, pid为3871
[root@kp-test01 lab1]# ./process3
parent: pid = 3885
child: pid1 = 3885
parent: pid1 = 3884
在子进程中调用system()函数, pid为3886
```

可以看到调用 system() 之后 pid 和之前不同, 是因为 system() 函数会启动一个新的进程, 拥有独立的代码空间, 内存空间等待新的进程执行完毕, system 才返回。

1.7.1.4. 在子进程中调用 exec 函数运行自己写的程序

所对应代码为 process4.c 和 exec_child_process.c

在子进程中使用 exec 函数调用 exec_child_process 程序, 在 exec_child_process 程序中获取当前进程的 pid 值并返回 pid 值。

```
[root@kp-test01 lab1]# gcc exec_child_process.c -o exec_child_process
[root@kp-test01 lab1]# gcc process4.c -o process4
[root@kp-test01 lab1]# ./process4
parent: pid = 3935
parent: pid1 = 3934
child: pid1 = 3935
在子进程中调用exec()函数, pid为3935
[root@kp-test01 lab1]# ./process4
parent: pid = 3949
child: pid1 = 3949
parent: pid1 = 3948
在子进程中调用exec()函数, pid为3949
[root@kp-test01 lab1]# ./process4
parent: pid = 3963
child: pid1 = 3963
parent: pid1 = 3962
在子进程中调用exec()函数, pid为3963
```

可以看到调用 `exec()` `pid` 值没有改变，这是因为。执行 `exec` 系列函数后，原来的进程将不再执行，改为执行对应路径可执行文件内容，新的进程的 `PID` 值与原先的完全一样。相当与用新进程替换当前进程，但 `pid` 值不做改变。

1.7.1.5. 在进程中创建线程

所对应代码为 `pthread1.c`

在进程中给变量 `global` 赋初值 0 并创建两个线程, 在线程一中对 `global` 进行++操作, 在线程二中对 `global` 进行自加 2 操作。

在两个线程中 `global` 循环五千次操作并输出结果。

```
线程2中global的值为8533
线程2中global的值为8535
线程2中global的值为8537
线程2中global的值为8539
线程2中global的值为8541
线程1中global的值为8443
线程1中global的值为8544
线程1中global的值为8545
线程1中global的值为8546
线程1中global的值为8547
线程1中global的值为8548
```

从图中部分运行结果我们能看出线程的执行顺序是不一定的，线程是并发执行的。

1.7.1.6. 在线程中实现互斥与同步

所对应代码为 `pthread2.c`

在线程中实现互斥与同步

```
线程1中global的值为4995
线程1中global的值为4996
线程1中global的值为4997
线程1中global的值为4998
线程1中global的值为4999
线程1中global的值为5000
线程2中global的值为5002
线程2中global的值为5004
线程2中global的值为5006
线程2中global的值为5008
线程2中global的值为5010
```

从图中运行结果可知，实现互斥与同步后，线程有序执行，使用 `global` 的值使它有规律增加。

1.7.1.7. 在线程中调用 system 函数

所对应代码为 pthread3.c 和 system_pthread.c

在线程一二中分别调用 system 函数, 输出进程的 pid, 线程的 tid。

getpid() 得到的是进程的 pid, 在内核中, 每个线程都有自己的 PID, 要得到线程的 PID, 必须用 syscall(SYS_gettid) pthread_self 函数获取的是线程 ID, 线程 ID 在某进程中是唯一的, 在不同的进程中创建的线程可能出现 ID 值相同的情况。

```
[root@kp-test01 lab1]# gcc system_pthread.c -o system_pthread
[root@kp-test01 lab1]# gcc pthread2.c -o pthread2 -lpthread
[root@kp-test01 lab1]# gcc pthread3.c -o pthread3 -lpthread
[root@kp-test01 lab1]# ./pthread3
主函数中创建第一个线程
global的值为5000
thread:int 4181 main process, the tid=281470341097696,pid=4181
主函数中创建第二个线程
global的值为15000
thread:int 4183 main process, the tid=281464591689952,pid=4183
[root@kp-test01 lab1]# ./pthread3
主函数中创建第一个线程
global的值为5000
thread:int 4198 main process, the tid=281473307781344,pid=4198
主函数中创建第二个线程
global的值为15000
thread:int 4200 main process, the tid=281461160749280,pid=4200
[root@kp-test01 lab1]# ./pthread3
主函数中创建第一个线程
global的值为5000
thread:int 4215 main process, the tid=281456792905952,pid=4215
主函数中创建第二个线程
global的值为15000
thread:int 4217 main process, the tid=281466935584992,pid=4217
```

1.7.1.8. 在线程中调用 exec 函数

所对应代码为 pthread4.c 和 exec_pthread.c

在线程一二中分别调用 exec 函数, 输出进程的 pid, 线程的 tid。

getpid() 得到的是进程的 pid, 在内核中, 每个线程都有自己的 PID, 要得到线程的 PID, 必须用 syscall(SYS_gettid) pthread_self 函数获取的是线程 ID, 线程 ID 在某进程中是唯一的, 在不同的进程中创建的线程可能出现 ID 值相同的情况。

```
[root@kp-test01 lab1]# gcc exec_pthread.c -o exec_pthread
[root@kp-test01 lab1]# gcc pthread4.c -o pthread4 -lpthread
[root@kp-test01 lab1]# ./pthread4
主函数中创建第一个线程
global的值为5000
thread:int 4279 main process, the tid=281467765795040,pid=4279
[root@kp-test01 lab1]# ./pthread4
主函数中创建第一个线程
global的值为5000
thread:int 4293 main process, the tid=281463908870368,pid=4293
[root@kp-test01 lab1]# ./pthread4
主函数中创建第一个线程
global的值为5000
thread:int 4307 main process, the tid=281469166364896,pid=4307
```

1.8. 实验总结

1.8.1. 实验中的问题与解决过程

1.8.1.1. 不清楚 system 函数和 exec 函数的使用方法

在查询资料后解决。

system:<https://blog.csdn.net/dark_cy/article/details/89715625>

exec:<<https://blog.csdn.net/zjwson/article/details/53337212>>

1.8.1.2. 编译 execl 报错

没有加对应头文件。

1.8.1.3. 编译 wait 报错

添加头文件#include<wait.h>解决。

1.8.1.4. 编译有关 pthread 的代码报错

查询资料后得知 pthread 不是 Linux 下的默认的库，也就是在链接的时候，无法找到 pthread 库中函数的入口地址，于是链接会失败。

```
[root@kp-test01 lab1]# gcc pthread1.c -o pthread1
/usr/bin/ld: /tmp/ccE8IksI.o: in function `main':
pthread1.c:(.text+0x104): undefined reference to `pthread_create'
/usr/bin/ld: pthread1.c:(.text+0x138): undefined reference to `pthread_join'
/usr/bin/ld: pthread1.c:(.text+0x160): undefined reference to `pthread_create'
/usr/bin/ld: pthread1.c:(.text+0x194): undefined reference to `pthread_join'
collect2: 错误: ld 返回 1
```

解决:在 gcc 编译加-lpthread 参数。

1.8.1.5. scp 命令向远程服务器传文件夹出错

```
yunkino@yunkino-virtual-machine:~$ scp /home/yunkino/文档/lab1 root@121.36.48.125:
Authorized users only. All activities may be monitored and reported.
root@121.36.48.125's password:
/home/yunkino/文档/lab1: not a regular file
yunkino@yunkino-virtual-machine:~$ scp -r /home/yunkino/文档/lab1 root@121.36.48.125:
```

解决: scp 命令传送文件夹要加-r 参数

1.8.2. 实验收获

原本对 linux 下的编译和调试环境不是很熟悉，但通过这次的实验，让我熟悉了 linux 下的编译器和调试器的使用。

通过实验，我了解了 linux 的系统调用机制，加深了对系统调用的理解。

通过本次实验了解了一些常用进程管理命令的使用，例如 ps、kill 命令，了解到 kill 与 killall 的不同，对于 linux 操作系统下的进程的学习打下基础，更好的学习进程。

本次实验是熟练掌握 Linux 系统常用进程创建与管理的系统调用，linux 下使用 fork() 创建子进程，通过比较更好的理解和掌握了进程的创建，对于进程的管理的理解也有了清晰地认识。

1.8.3. 意见与建议

输出检验的方法可以再优化一下，过于繁琐的输出反而会影响对进程与线程运行情况的判断。

1.9. 附件

1.9.2. 附件 1 程序

```
process1.c
#include<sys/types.h>
#include<stdio.h>
#include<unistd.h>
#include<wait.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\n",pid);//A
        printf("child: pid1 = %d\n",pid1);//B
    }
    else
    {    //parent process
        pid1 = getpid();
        printf("parent: pid = %d\n",pid);//C
        printf("parent: pid1 = %d\n",pid1);//D
        wait(NULL);
    }
}
```

```
}

    return 0;
}

process2.c
#include<sys/types.h>
#include<stdio.h>
#include<unistd.h>
#include<wait.h>

int gobal = 0;

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();
        gobal--;
        printf("gobal = %d\n", gobal);
        printf("gobal address = %p\n",&gobal);
        printf("child: pid = %d\n",pid);//A
        printf("child: pid1 = %d\n",pid1);//B
    }
    else
    {    //parent process
        pid1 = getpid();
        gobal++;
        printf("gobal = %d\n", gobal);
        printf("gobal address = %p\n",&gobal);
        printf("parent: pid = %d\n",pid);//C
        printf("parent: pid1 = %d\n",pid1);//D
        wait(NULL);
    }

    return 0;
}
```

```
process3.c
#include<sys/types.h>
#include<stdio.h>
#include<unistd.h>
```



```
#include<stdlib.h>
#include<wait.h>

int main()
{
    pid_t pid,pid1;

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

    if(pid < 0)
    {    //error occured
        fprintf(stderr, "Fork Failed");
        return 1;
    }
    else if (pid == 0)
    {    //child process
        pid1 = getpid();
        printf("child: pid1 = %d\n",pid1);//B
        system("./system_child_process");
    }
    else
    {    //parent process
        pid1 = getpid();
        printf("parent: pid = %d\n",pid);//C
        printf("parent: pid1 = %d\n",pid1);//D
        wait(NULL);
    }

    return 0;
}
```

system_child_process.c

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

int main()
{
    pid_t pid = getpid();
    printf("在子进程中调用 system() 函数，pid 为%d\n",pid);

    return 0;
}
```

process4.c

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

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

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

    if(pid < 0)
    {    //error occured
        fprintf(stderr, "Fork Failed");
        return 1;
    }
    else if (pid == 0)
    {    //child process
        pid1 = getpid();
        printf("child: pid1 = %d\n",pid1);//B
        execl("./exec_child_process","exec_child_process" ,NULL);
    }
    else
    {    //parent process
        pid1 = getpid();
        printf("parent: pid = %d\n",pid);//C
        printf("parent: pid1 = %d\n",pid1);//D
        wait(NULL);
    }

    return 0;
}
```

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

```
int main()
{
    pid_t pid = getpid();
    printf("在子进程中调用 exec() 函数，pid 为%d\n",pid);

    return 0;
}
```

```
pthread1.c
#include<stdio.h>
#include<stdlib.h>
#include<pthread.h>
```

```
int global = 0;
```

```
void* thread1_func()
{
    int i;
```

```
for(i = 0;i < 5000;i++)
{
    global++;
    printf("线程 1 中 global 的值为%d\n",global);
}

}

void* thread2_func()
{
    int i;
    for(i = 0;i < 5000;i++)
    {
        global = global + 2;
        printf("线程 2 中 global 的值为%d\n",global);
    }
}

int main()
{
    int status;
    pthread_t tid_one,tid_two;

    printf("主函数中创建第一个线程\n");
    status = pthread_create(&tid_one ,NULL ,thread1_func ,NULL);
    if(status != 0)
    {    //线程创建不成功，打印错误信息
        printf("线程创建失败 %d\n",status);
        return 1;
    }

    printf("主函数中创建第二个线程\n");
    status = pthread_create(&tid_two ,NULL ,thread2_func ,NULL);
    if(status != 0)
    {    //线程创建不成功，打印错误信息
        printf("线程创建失败 %d\n",status);
        return 1;
    }

    pthread_join(tid_one,NULL);
    pthread_join(tid_two,NULL);

    return 0;
}

pthread2.c
#include<stdio.h>
#include<stdlib.h>
#include<pthread.h>

int global = 0;
```

```
pthread_mutex_t mutex = PTHREAD_MUTEX_INITIALIZER;
```

```
void* thread1_func()
{
    pthread_mutex_lock(&mutex);
    int i;
    for(i = 0; i < 5000; i++)
    {
        global++;
        printf("线程 1 中 global 的值为%d\n", global);
    }
    pthread_mutex_unlock(&mutex);
}
```

```
void* thread2_func()
{
    pthread_mutex_lock(&mutex);
    int i;
    for(i = 0; i < 5000; i++)
    {
        global = global + 2;
        printf("线程 2 中 global 的值为%d\n", global);
    }
    pthread_mutex_unlock(&mutex);
}
```

```
int main()
{
    int status;
    pthread_t tid_one, tid_two;

    printf("主函数中创建第一个线程\n");
    status = pthread_create(&tid_one, NULL, thread1_func, NULL);
    if(status != 0)
    {
        //线程创建不成功，打印错误信息
        printf("线程创建失败 %d\n", status);
        return 1;
    }

    printf("主函数中创建第二个线程\n");
    status = pthread_create(&tid_two, NULL, thread2_func, NULL);
    if(status != 0)
    {
        //线程创建不成功，打印错误信息
        printf("线程创建失败 %d\n", status);
        return 1;
    }

    pthread_join(tid_one, NULL);
    pthread_join(tid_two, NULL);

    return 0;
}
```

```
}
```

```
pthread3.c
```

```
#include<stdio.h>
```

```
#include<stdlib.h>
```

```
#include<pthread.h>
```

```
int global = 0;
```

```
pthread_mutex_t mutex = PTHREAD_MUTEX_INITIALIZER;
```

```
void* thread1_func()
```

```
{
```

```
    pthread_mutex_lock(&mutex);
```

```
    int i;
```

```
    for(i = 0; i < 5000; i++)
```

```
    {
```

```
        global++;
```

```
    }
```

```
    printf("global 的值为%d\n", global);
```

```
    system("./system_pthread");
```

```
    pthread_mutex_unlock(&mutex);
```

```
}
```

```
void* thread2_func()
```

```
{
```

```
    pthread_mutex_lock(&mutex);
```

```
    int i;
```

```
    for(i = 0; i < 5000; i++)
```

```
    {
```

```
        global = global + 2;
```

```
    }
```

```
    printf("global 的值为%d\n", global);
```

```
    system("./system_pthread");
```

```
    pthread_mutex_unlock(&mutex);
```

```
}
```

```
int main()
```

```
{
```

```
    int status;
```

```
    pthread_t tid_one, tid_two;
```

```
    printf("主函数中创建第一个线程\n");
```

```
    status = pthread_create(&tid_one, NULL, thread1_func, NULL);
```

```
    if(status != 0)
```

```
    { //线程创建不成功, 打印错误信息
```

```
        printf("线程创建失败 %d\n", status);
```

```
        return 1;
```

```
    }
```

```
    printf("主函数中创建第二个线程\n");
```

```
    status = pthread_create(&tid_two, NULL, thread2_func, NULL);
```

```
    if(status != 0)
```

```
{ //线程创建不成功, 打印错误信息
    printf("线程创建失败 %d\n", status);
    return 1;
}

pthread_join(tid_one, NULL);
pthread_join(tid_two, NULL);

return 0;
}

system_thread.c
#include <stdio.h>
#include <pthread.h>
#include <stdlib.h>
#include <unistd.h>
#include <sys/syscall.h>

int main()
{
    printf("thread:int %d main process, the
tid=%lu, pid=%ld\n", getpid(), pthread_self(), syscall(SYS_gettid));
    return 0;
}

pthread4.c
#include<stdio.h>
#include<stdlib.h>
#include<pthread.h>
#include<unistd.h>

int global = 0;
pthread_mutex_t mutex = PTHREAD_MUTEX_INITIALIZER;

void* thread1_func()
{
    pthread_mutex_lock(&mutex);
    int i;
    for(i = 0; i < 5000; i++)
    {
        global++;
    }
    printf("global 的值为%d\n", global);
    execl("./exec_pthread", "exec_pthread", NULL);
    pthread_mutex_unlock(&mutex);
}

void* thread2_func()
{
    pthread_mutex_lock(&mutex);
    int i;
```

```
for(i = 0;i < 5000;i++)
{
    global = global + 2;
}
printf("global 的值为%d\n",global);
execl("./exec_pthread","exec_pthread",NULL);
pthread_mutex_unlock(&mutex);
}

int main()
{
    int status;
    pthread_t tid_one,tid_two;

    printf("主函数中创建第一个线程\n");
    status = pthread_create(&tid_one ,NULL ,thread1_func ,NULL);
    if(status != 0)
    {    //线程创建不成功, 打印错误信息
        printf("线程创建失败 %d\n",status);
        return 1;
    }

    printf("主函数中创建第二个线程\n");
    status = pthread_create(&tid_two ,NULL ,thread2_func ,NULL);
    if(status != 0)
    {    //线程创建不成功, 打印错误信息
        printf("线程创建失败 %d\n",status);
        return 1;
    }

    pthread_join(tid_one,NULL);
    pthread_join(tid_two,NULL);

    return 0;
}
```

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

int main()
{
    printf("thread:int %d main process, the
tid=%lu,pid=%ld\n",getpid(),pthread_self(),syscall(SYS_gettid));
    return 0;
}
```

1.9.1.附件 2 Readme

实验一 华为云上 openEuler 操作系统环境

1 搭建华为实验环境

1.1 服务器配置

![服务器配置] (/home/yunkino/文档/lab1/服务器配置.png)

1.2 远程控制服务器

使用 ssh 命令控制服务器

![使用 ssh 命令控制服务器] (/home/yunkino/文档/lab1/ssh 控制远程服务器.png)

使用 scp 命令将本地写好的文件传输到远程服务器

![文件上传] (/home/yunkino/文档/lab1/使用 scp 命令将本地文件上传至远程服务器.png)

2 进程相关编程实验

2.1 课程教材上 P104 页图 3-32 所示程序

完成课程教材上 P104 页图 3-32 所示程序, 多次运行验证

所对应代码为 process1.c

使用命令 gcc process1.c -o process1 编译, ./process1 执行

![课本上程序多次运行结果] (/home/yunkino/文档/lab1/process1 不删除 wait.png)

从运行结果来看既有可能 child 先执行, 又有可能 parent 先执行。

去除 wait() 再运行

![去除 wait() 再多次运行结果] (/home/yunkino/文档/lab1/去除 wait 后.png)

在去掉 wait() 后, 同样也是既有可能 child 先执行, 又有可能 parent 先执行。

由于 fork() 函数会将父进程复制产生子进程, 并同时执行父进程与子进程, 所以才会出现结果中, child 和 parent 进程交错输出的情况。

对于父进程, fork() 函数返回子进程的 pid, 对于子进程, fork() 函数返回的是 0。getpid() 返回当前进程的 pid 值, 所以 parent 的 pid 和 child 的 pid 值相同, 为子进程的 pid 值。

在去掉 wait() 前后, 都是既有可能 parent 先输出, 又有可能 child 先输出, 因为父子进程的执行顺序是不确定的。wait() 的作用是让父进程挂起等待子进程结束。所以 wait() 放在父进程输出之前才能确保子进程先输出。

2.2 扩展程序

所对应代码为 process2.c

定义全局变量, 在父进程与子进程中分别对其进行操作。

定义 int 类型变量 global 初始化为 0, 在父进程中对其进行++操作, 在子进程中对其进行--操作。

![运行结果] (/home/yunkino/文档/lab1/没加 return.png)

在 return 前增加对全局变量*2 的操作

![运行结果] (/home/yunkino/文档/lab1/加 return.png)

从结果可以看出来 global 全局变量地址是相同的, 原因在于这个地址是虚地址, 而因为子进程的创建是复制了父进程的虚拟地址空间的, 因为这两个变量的虚地址也是一样的。

如果父进程对其进行修改, 那么在真正的物理内存中会拷贝一份 global 的副本, 父进程中修改的其实是副本。对于子进程而言也是如此。

2.3 在子进程中调用 system 函数运行自己写的程序

所对应代码为 process3.c 和 system_child_process.c

在子进程中使用 system 函数调用 system_child_process 程序, 在 system_child_process 程序中获取当前进程的 pid 值并返回 pid 值。

![运行结果] (/home/yunkino/文档/lab1/子进程 system.png)

可以看到调用 system() 之后 pid 和之前不同, 是因为 system() 函数会启动一个新的进程, 拥有独立的代码空间, 内存空间等待新的进程执行完毕, system 才返回。

2.4 在子进程中调用 exec 函数运行自己写的程序

所对应代码为 process4.c 和 exec_child_process.c

在子进程中使用 exec 函数调用 exec_child_process 程序, 在 exec_child_process 程序中获取当前进程的 pid 值并返回 pid 值。

![运行结果] (/home/yunkino/文档/lab1/子进程 exec.png)

可以看到调用 exec() pid 值没有改变, 这是因为。执行 exec 系列函数后, 原来的进程将不再执行, 改为执行对应路径可执行文件内容, 新的进程的 PID 值与原先的完全一样。相当与用新进程替换当前进程, 但 pid 值不做改变。

2.5 在进程中创建线程

所对应代码为 pthread1.c

在进程中给变量 global 赋初值 0 并创建两个线程, 在线程一中对 global 进行++操作, 在线程二中对 global 进行自加 2 操作。

在两个线程中 global 循环五千次操作并输出结果

![部分运行结果] (/home/yunkino/文档/lab1/pthread1.png)

从图中部分运行结果我们能看出线程的执行顺序是不一定的。

2.6 在线程中实现互斥与同步

所对应代码为 pthread2.c

在线程中实现互斥与同步

![运行结果] (/home/yunkino/文档/lab1/pthread2.png)

从图中运行结果可知, 实现互斥与同步后, 线程有序执行, 使用 global 的值使它有规律增加。

2.7 在线程中调用 system 函数

所对应代码为 pthread3.c 和 system_pthread.c

在线程一二中分别调用 system 函数, 输出进程的 pid, 线程的 tid。

getpid() 得到的是进程的 pid, 在内核中, 每个线程都有自己的 PID, 要得到线程的 PID, 必须用 syscall(SYS_gettid) pthread_self 函数获取的是线程 ID, 线程 ID 在某进程中是唯一的, 在不同的进程中创建的线程可能出现 ID 值相同的情况。

![运行结果] (/home/yunkino/文档/lab1/pthread_system.png)

2.8 在线程中调用 exec 函数

所对应代码为 pthread4.c 和 exec_pthread.c

在线程一二中分别调用 exec 函数, 输出进程的 pid, 线程的 tid。

getpid() 得到的是进程的 pid, 在内核中, 每个线程都有自己的 PID, 要得到线程的 PID, 必须用 syscall(SYS_gettid) pthread_self 函数获取的是线程 ID, 线程 ID 在某进程中是唯一的, 在不同的进程中创建的线程可能出现 ID 值相同的情况。

![运行结果] (/home/yunkino/文档/lab1/pthread_exec.png)

3 遇到的问题与解决方法

1. 不清楚 system 函数和 exec 函数的使用方法

在查询资料后解决。

system:<https://blog.csdn.net/dark_cy/article/details/89715625>

exec:<<https://blog.csdn.net/zjwson/article/details/53337212>>

2. 编译 execl 报错

没有加对应头文件。

3. 编译 wait 报错

添加头文件#include<wait.h>解决。

4. 编译有关 pthread 的代码报错

![报错截图](/home/yunkino/文档/lab1/pthread 编译错误.png)

查询资料后得知 pthread 不是 Linux 下的默认的库，也就是在链接的时候，无法找到 pthread 库中函数的入口地址，于是链接会失败。

解决:在 gcc 编译加-lpthread 参数。

5. scp 命令向远程服务器传文件夹出错

![报错截图](/home/yunkino/文档/lab1/scp 命令报错.png)

解决: scp 命令传送文件夹要加-r 参数

2. 进程通信与内存管理

2.1. 实验目的

本实验在用户态下，根据教材所学习的操作系统原理，完成 Linux 下进程通信与内存管理算法的实现，通过实验，进一步理解所学理论知识。

2.1.1. 进程的软中断通信

编程实现进程的创建和软中断通信，通过观察、分析实验现象，深入理解进程及进程在调度执行和内存空间等方面的特点，掌握在 POSIX 规范中系统调用的功能和使用。

2.1.2. 进程的管道通信

编程实现进程的管道通信，通过观察、分析实验现象，深入理解进程管道通信的特点，掌握管道通信的同步和互斥机制。

2.1.3. 页面的替换

模拟实现 FIFO 算法，LRU 算法。

2.2. 实验内容

2.2.1. 进程的软中断通信

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（即按 `delete` 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，输入以下信息，结束进程执行：

Parent process is killed!!

多运行几次编写的程序，简略分析出现不同结果的原因。

2.2.2. 进程的管道通信

编程实现进程的管道通信。

为了协调双方的通信，管道机制必须提供以下三方面的协调能力：

①互斥，即当一个进程正在对 pipe 执行读/写操作时，其它(另一)进程必须等待。

②同步，指当写(输入)进程把一定数量(如 4KB)的数据写入 pipe，便去睡眠等待，直到读(输出)进程取走数据后，再把他唤醒。当读进程读一空 pipe 时，也应睡眠等待，直至写进程将

数据写入管道后，才将之唤醒。

③确定对方是否存在，只有确定了对方已存在时，才能进行通信。

2.2.3. 页面的替换

模拟实现 FIFO 算法，LRU 算法。

2.3. 实验思想

2.3.1. 进程的软中断通信

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（即按 `delete` 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，然后分别输出信息后终止。

2.3.2. 进程的管道通信

所谓“管道”，是指用于连接一个读进程和一个写进程以实现他们之间通信的一个共享文件，又名 `pipe` 文件。向管道(共享文件)提供输入的发送进程(即写进程)，以字符流形式将大量的数据送入管道；而接受管道输出的接收进程(即读进程)，则从管道中接收(读)数据。由于发送进程和接收进程是利用管道进行通信的，故又称为管道通信。这种方式首创于 UNIX 系统，由于它能有效地传送大量数据，因而又被引入到许多其它操作系统中。

为了协调双方的通信，管道机制必须提供以下三方面的协调能力：

①互斥，即当一个进程正在对 `pipe` 执行读/写操作时，其它(另一)进程必须等待。

②同步，指当写(输入)进程把一定数量(如 4KB)的数据写入 `pipe`，便去睡眠等待，直到读(输出)进程取走数据后，再把他唤醒。当读进程读一空 `pipe` 时，也应睡眠等待，直至写进程将数据写入管道后，才将之唤醒。

③确定对方是否存在，只有确定了对方已存在时，才能进行通信。

管道是进程间通信的一种简单易用的方法。管道分为匿名管道和命名管道两种。下面首先介绍匿名管道。

匿名管道只能用于父子进程之间的通信，它的创建使用系统调用 `pipe()`：

```
int pipe(int fd[2])
```

其中的参数 `fd` 用于描述管道的两端，其中 `fd[0]` 是读端，`fd[1]` 是写端。两个进程分别使用读端和写端，就可以进行通信了。

一个父子进程间使用匿名管道通信的例子。

匿名管道只能用于父子进程之间的通信，而命名管道可以用于任何管道之间的通信。命名管道实际上就是一个 FIFO 文件，具有普通文件的所有性质，用 `ls` 命令也可以列表。但是，它只是一块内存缓冲区。

`pipe` 使用：

pipe 系统调用的语法格式是： `int pipe (filedes)`

`int filedes [2];`

核心创建一条管道完成下述工作：

- a. 分配一个隶属于 root 文件系统的磁盘和内存索引结点 inode.
- b. 在系统打开文件表中分别分配一读管道文件表项和一写管道文件表项。
- c. 在创建管道的进程控制块的文件描述表（进程打开文件表 u-ofile） 中分配二表项，表项中的偏移量 `filedes[0]`和 `filedes[1]`分别指向系统打开文件表的读和写管道文件表项。

2.3.3. 页面的替换

FIFO 算法：

在分配内存页面数（AP）小于进程页面数（PP）时，当然是最先运行的 AP 个页面放入内存；这时又需要处理新的页面，则将原来放的内存中的 AP 个页中最先进入的调出（FIFO），再将新页面放入；

以后如果再有新页面需要调入，则都按上述规则进行。

算法特点：所使用的内存页面构成一个队列。

要得到命中率，必然应该有一个常量 `total_instruction` 来记录页面总共使用的次数，此外还需要一个变量记录总共换入页面的次数 `diseffect`（需要换出页面总是因为缺页中断而产生）。利用公式 $1 - \text{diseffect} / \text{total_instruction} * 100\%$ 可以得到命中率。

LRU 算法：

当内存分配页面数（AP）小于进程页面数（PP）时，把最先执行的 AP 个页面放入内存。

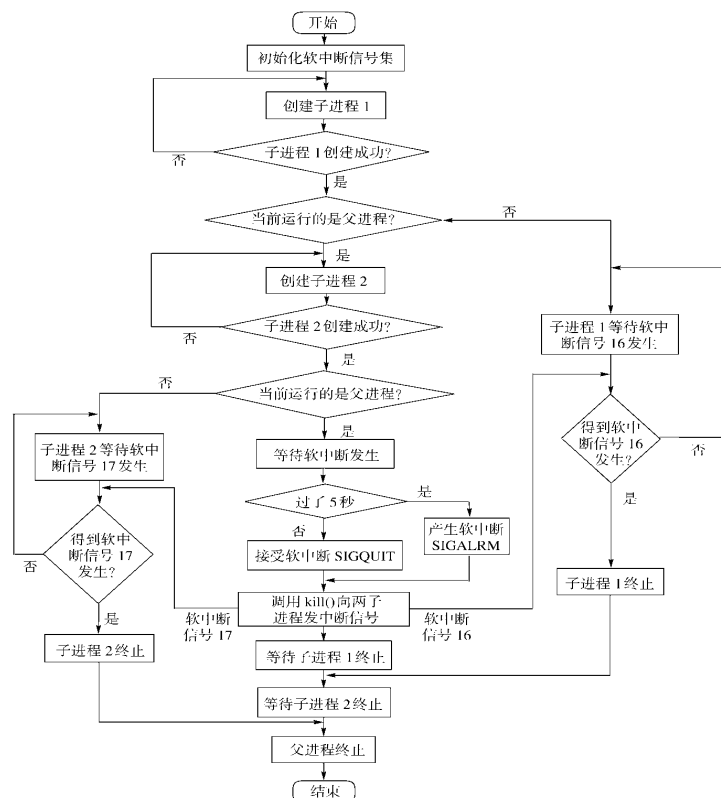
当需调页面进入内存，而当前分配的内存页面全部不空闲时，选择将其中最长时间没有用到的那一页调出，以空出内存来放置新调入的页面（LRU）。

算法特点：每个页面都有属性来表示有多长时间未被 CPU 使用的信息。

与前述算法一样，只有先得到 `diseffect` 才能获得最终的命中率。

2.4. 实验步骤

2.4.1. 软中断通信



软中断通信程序流程图

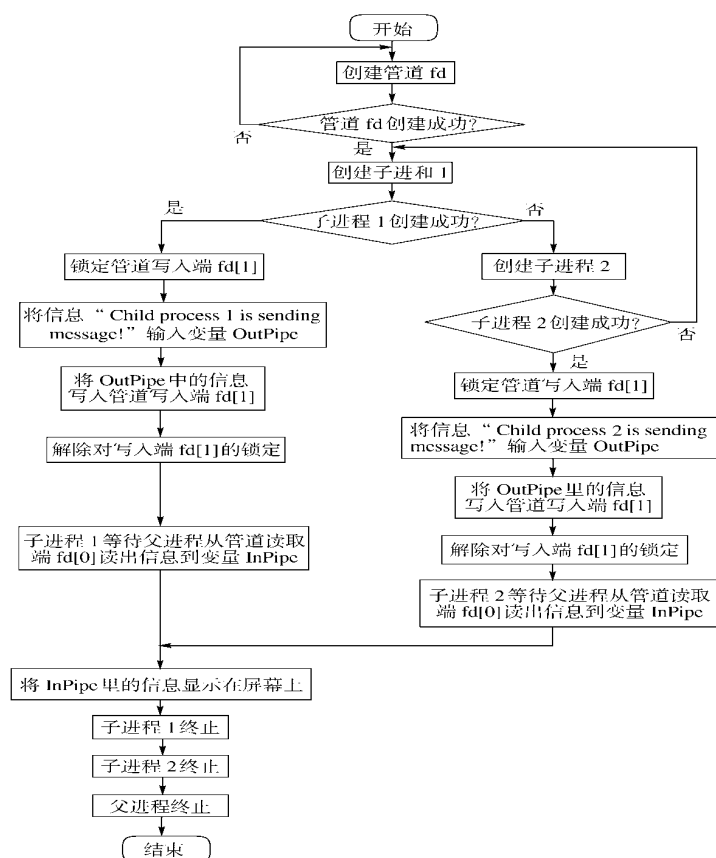
1、根据流程图编写程序，猜想一下这个程序的运行结果，然后多次运行，观察 Delete/quit 键前后出现的结果并分析原因。

2、如果程序运行界面上显示 “Child process 1 is killed by parent !! Child process 2 is killed by parent !!”，五秒之后显示 “Parent process is killed !!”，修改程序使得只有接收到相应的中断信号后再发生跳转，执行输出。

3、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，将 `signal(3, stop)` 当中数字信号变为 2，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。

注：delete 会向进程发送 SIGINT 信号，quit 会向进程发送 SIGQUIT 信号。ctrl+c 为 delete，ctrl+\ 为 quit。https://blog.csdn.net/mylzh/article/details/38385739。

2.4.2. 管道通信



管道通信程序流程图

- 1) 先猜想一下这个程序的运行结果。分析管道通信是怎样实现同步与互斥的；
- 2) 然后按照注释里的要求把代码补充完整，运行程序；
- 3) 修改程序并运行，体会互斥锁的作用，比较有锁和无锁程序的运行结果，并解释之。

2.4.3. 页面的替换

FIFO 算法：

(1) 初始化。设置两个数组 `page[ap]` 和 `pagecontrol[pp]` 分别表示进程页面数和内存分配的页面数，并产生一个随机数序列 `main[total_instruction]`（这个序列由 `page[ap]` 的下标随机构成）表示待处理的进程页面顺序，`diseffect` 置 0。

(2) 看 `main[]` 中是否有下一个元素，若有，就由 `main[]` 中获取该页面下标，并转 (3)，如果没有则转 (7)。

(3) 如果该页已在内存中，就转 (2)，否则转 (4)，同时未命中的 `diseffect` 加 1。

(4) 观察 `pagecontrol` 是否占满，如果占满则须将使用队列（在第 (6) 步中建立的）中最先进入的（就是队列的第一个单元）`pagecontrol` 单元“清干净”，同时将 `page[]` 单元置为“不在内存中”。

(5) 将该 `page[]` 与 `pagecontrol[]` 建立对应关系（可以改变 `pagecontrol[]` 的标志位，也可以采用指针链接，总之至少要使对应的 `pagecontrol` 单元包含两个信息：一是它被使用了，二是哪个 `page[]` 单元使用的。`page[]` 单元也包含两个信息：对应的 `pagecontrol` 单元号和本

page[]单元已在内存中)。

(6) 将用到的 pagecontrol 置入使用队列 (这里的队列是一种 FIFO 的数据结构), 返回 (2)。

(7) 显示计算 $1 - \text{diseffect} / \text{total_instruction} * 100\%$, 完成。

LRU 算法:

(1) 初始化。设置两个数组 page[ap]和 pagecontrol[pp]分别表示进程页面数和内存分配的页面数, 并产生一个随机数序列 main[total_instruction] (这个序列由 page[ap]的下标随机构成) 表示待处理的进程页面顺序, diseffect 置 0。

(2) 看序列 main[]中是否有下一个元素, 如果有, 就由 main[]中获取该页面下标, 并转 (3), 如果没有则转 (6)。

(3) 如果该 page[]单元在内存中便改变页面属性, 使它保留 “最近使用” 的信息, 转 (2), 否则转 (4), 同时 diseffect 加 1。

(4) 看是否有空闲页面, 如果有, 就返回页面指针, 并转到 (5), 否则, 在内存页面中找出最长时间没有使用到的页面, 将其 “清干净”, 并返回该页面指针。

(5) 在需处理的 page[]与 (4) 中得到的 pagecontrol[]之间建立联系, 同时让对应的 page[]单元保存 “最新使用” 的信息, 转 (2)。

(6) 如果序列处理完成, 就输出计算 $1 - \text{diseffect} / \text{total_instruction} * 100\%$ 的结果, 完成。

2.5. 测试数据设计

2.5.1. 进程的管道通信

创建 pipe 管道来让两个子进程与父进程完成通信, 子进程 1 向父进程发送 2000 个 1, 子进程 2 向父进程发送 2000 个 2。

2.5.2. 页面的替换

进程页面数为: 10

内存分配的页面数为: 4

2.6. 程序运行初值及运行结果分析

2.6.1. 进程的软中断通信

2.6.1.1. 进程的软中断通信——delete 与 quit

代码参考 interrupt1.c

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 (即按 delete 与 quit, 分别对应 ctrl+c 与 ctrl+\), 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，输入以下信息，结束进程执行：Parent process is killed!!

```
yunkino@yunkino-virtual-machine:~/OS/lab2$ ./interrupt1

Child process 2 is killed by parent!!

Child process 1 is killed by parent!!

Parent process is killed!!
yunkino@yunkino-virtual-machine:~/OS/lab2$ ./interrupt1

Child process 1 is killed by parent!!

Child process 2 is killed by parent!!

Parent process is killed!!
yunkino@yunkino-virtual-machine:~/OS/lab2$ ./interrupt1

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

Parent process is killed!!
```

运行过程中，首先出现“Child process 1 is killed by parent!!”“Child process 2 is killed by parent!!”两条结果，且两个结果输出的先后顺序是不确定的。几秒后，出现结果“Parent process is killed!!”。

分析可知，两子进程的运行先后顺序并不确定，因为两个子进程同时运行，并没有先后的区别。但由于 `wait(0)` 和 `sleep(5)` 函数的存在，父进程会挂起等到两子进程结束后在进行。故父进程输出的结果在最后。

子进程结果输出后，父进程结果输出前，键盘输入 `delete` 或者 `quit`，分别出现以下情况

```
yunkino@yunkino-virtual-machine:~/OS/lab2$ ./interrupt1

Child process 2 is killed by parent!!

Child process 1 is killed by parent!!
^C
Parent process is killed!!
```

输入 `delete` 后

```
yunkino@yunkino-virtual-machine:~/OS/lab2$ ./interrupt1

Child process 2 is killed by parent!!

Child process 1 is killed by parent!!
^\\退出（核心已转储）
```

输入 quit 后

根据 signal 信号表可知，输入 delete 后，触发中断，会执行中断对应程序，然后在返回原先程序运行处继续运行，故仍会输出“Parent process is killed!!”

而输入 quit 后，进程直接终止，也就不会输出“Parent process is killed!!”

若想让程序运行界面上显示“Child process 1 is killed by parent!! Child process 2 is killed by parent!!”，五秒之后显示“Parent process is killed!!”，只有接收到相应的中断信号后再发生跳转，执行输出。

修改代码后，代码参考 interrupt2.c

由于存在全局变量 wait_flag，在中断中将其值赋为 0。故在执行输出前，先给 wait_flag 赋值为 1，然后用 while(wait_flag == 1); 循环判断，直至中断产生再输出结果。

运行程序，程序开始等待键盘输入 delete 键或者 quit 键。

按下对应键后，程序运行完毕，父进程与子进程输出完毕。

```
yunkino@yunkino-virtual-machine:~/OS/lab2$ ./interrupt2
^C
中断信号数为：2

中断信号数为：16

中断信号数为：2

Child process 1 is killed by parent!!

中断信号数为：17

中断信号数为：2

Child process 2 is killed by parent!!

Parent process is killed!!
```

输入 delete 键后

```

yunkino@yunkino-virtual-machine:~/OS/lab2$ ./interrupt2
^\\
中断信号数为: 3

中断信号数为: 17
♦中断信号数为: 3

Child process 2 is killed by parent!!

中断信号数为: 16

中断信号数为: 3

Child process 1 is killed by parent!!

Parent process is killed!!

```

输入 quit 键后

为了观察实际运行的先后情况，在父进程向子进程通过 `kill()` 发送中断信号后，输出 “kill 1/2 success”，在子进程中断信号初始化后，输出 “signal 1/2 initial success”

```

yunkino@yunkino-virtual-machine:~/OS/lab2$ ./interrupt2

signal 1 initail success

signal 2 initail success
^C
中断信号数为: 2

Child process 1 is killed by parent!!

中断信号数为: 2

kill 1 success

kill 2 success

中断信号数为: 17

中断信号数为: 2

中断信号数为: 16
Child process 2 is killed by parent!!

Parent process is killed!!

```

从输出来看子进程中断先初始化成功，然后父进程 “kill” 子进程，最后父进程结束。

2.6.1.2. 进程的软中断通信——alarm

代码参考 interrupt3.c

alarm(1) 让进程在开始一秒后发出中断。程序运行结果来看，等待大约一秒后出现输出。

```
yunkino@yunkino-virtual-machine:~/05/lab2$ ./interrupt3
signal 1 initail success
signal 2 initail success
中断信号数为: 14
kill 1 success
中断信号数为: 17
Child process 2 is killed by parent!!
kill 2 success
中断信号数为: 16
Child process 1 is killed by parent!!
Parent process is killed!!
```

alarm 中断

2.6.2. 进程的管道通信

参考代码 pipe.c

管道通信是指用于连接一个读进程和一个写进程以实现他们之间通信的一个共享文件，又名 pipe 文件。向管道(共享文件)提供输入的发送进程(即写进程)，以字符流形式将大量的数据送入管道；而接受管道输出的接收进程(即读进程)，则从管道中接收(读)数据。创建 pipe 管道来让两个子进程与父进程完成通信，子进程 1 向父进程发送 2000 个 1，子进程 2 向父进程发送 2000 个 2。

预估输出会为 2000 个 1 和 2000 个 2，1 和 2 的顺序不一定。由于锁机制的存在，并不会存在 1、2 交替输出的问题。

```
yunkino@yunkino-virtual-machine:~/05/lab$ ./pipe
```

The output of the program is a large grid of characters. The first 28 rows consist entirely of the character '1'. The remaining 72 rows are composed of a mix of '1' and '2' characters, forming a grayscale-like representation of a face. The '2's are concentrated in the center of the grid, corresponding to the facial features.

pipe 运行结果

在去除互锁及之后，预计将会出现 1,2 交替输出的情况。

```
yunkino@yunkino-virtual-machine:~/OS/lab2$ ./pipe
```

[illegible]

pipe 去除互锁后的结果

从结果来看，子进程 2 先进运行，随后子进程 1 开始运行后出现了 1, 2 交替输出的情况。

2.6.3. 页面的替换

参考代码 fifo

页面替换实现 FIFO 算法和 LRU 算法。

FIFO 算法其原理为：

在当前内存分配的页面数未满足时，将新加入的页面直接放在空位中。

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先加入的页面。

LRU 算法其原理为：

在当前内存分配的页面数未满足时，将新加入的页面直接放在空位中。

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。

其中 FIFO 算法由于其先进先出的特性，用队列来实现。

LRU 算法则需要按页面的访问次数进行排序，则用栈对页面访问频繁程度进行更新。

分别 `page[ap]` 和 `pagecontrol[pp]` 分别表示进程页面数和内存分配的页面数

定义常量 `total_instruction` 来记录页面总共使用的次数，变量 `diseffect` 记录总共换入页面的次数（需要换出页面总是因为缺页中断而产生）。

利用公式 $1 - \text{diseffect} / \text{total_instruction} * 100\%$ 可以得到命中率。

页面替换实现 FIFO 算法和 LRU 算法。

FIFO 算法其原理为:

在当前内存分配的页面数未满足时，将新加入的页面直接放在空位中。

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先加入的页面。

LRU 算法其原理为:

在当前内存分配的页面数未满足时，将新加入的页面直接放在空位中。

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。

其中 FIFO 算法由于其先进先出的特性，用队列来实现。

LRU 算法则需要按页面的访问次数进行排序, 则用栈对页面访问频繁程度进行更新。

分别 `page[ap]`和 `pagecontrol[pp]`分别表示进程页面数和内存分配的页面数

定义常量 `total_instruction` 来记录页面总共使用的次数，变量 `diseffect` 记录总共换入页面的次数（需要换出页面总是因为缺页中断而产生）。

利用公式 $1 - \text{diseffect} / \text{total_instruction} * 100\%$ 可以得到命中率。

进程页面数为：10	进程页面数为：10
内存分配的页面数为：4	内存分配的页面数为：4
页面访问顺序为：2 2 5 9 4 4 9 3 3 9 3 9 7 6 4	页面访问顺序为：9 4 7 7 4 8 8 6 0 3 7 4 1 2 3
请输入替换算法类型：1为FIFO，2为LRU。	请输入替换算法类型：1为FIFO，2为LRU。
1	2
当前算法为FIFO算法	当前算法为LRU算法
2 -1 -1 -1	9 -1 -1 -1
2 -1 -1 -1	9 4 -1 -1
2 5 -1 -1	9 4 7 -1
2 5 9 -1	9 4 7 -1
2 5 9 4	9 4 7 -1
2 5 9 4	9 4 7 8
2 5 9 4	9 4 7 8
3 5 9 4	6 4 7 8
3 5 9 4	6 4 0 8
3 5 9 4	6 3 0 8
3 5 9 4	6 3 0 7
3 5 9 4	4 3 0 7
3 7 9 4	4 3 1 7
3 7 6 4	4 2 1 7
3 7 6 4	4 2 1 3
未命中次数为：7	未命中次数为：12
命中率为：53.33%	命中率为：20.00%

FIFO 算法的结果

LRU 算法的结果

2.7. 页面置换算法复杂度分析

从代码来看，由于 FIFO 算法队列的运用和 LRU 算法栈的运用，算法的时间复杂度主要取决于在当前内存中的页面中查找空位与当前页面是否在内存中，故时间复杂度为 $O(n)$ ，其中 n 为内存中的页面数。

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；

程序运行界面上显示“Child process 1 is killed by parent !! Child process 2 is killed by parent !!”，但先后顺序是不固定的，五秒之后显示“Parent process is killed !!”

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。

和猜测的结果一样，由于进程的并发，子进程输出的前后顺序不同。

运行过程中，首先出现“Child process 1 is killed by parent!!”“Child process 2 is killed by parent!!”两条结果，且两个结果输出的先后顺序是不确定的。几秒后，出现结果“Parent

process is killed!!”。

分析可知，两子进程的运行先后顺序并不确定，因为两个子进程同时运行，并没有先后的区别。但由于 `wait(0)` 和 `sleep(5)` 函数的存在，父进程会挂起等到两子进程结束后在进行。故父进程输出的结果在最后。

子进程结果输出后，父进程结果输出前，键盘输入 `delete` 或者 `quit`，分别出现以下情况

```
yunkino@yunkino-virtual-machine:~/OS/lab2$ ./interrupt1

Child process 2 is killed by parent!!

Child process 1 is killed by parent!!
^C
Parent process is killed!!
```

输入 `delete` 后

```
yunkino@yunkino-virtual-machine:~/OS/lab2$ ./interrupt1

Child process 2 is killed by parent!!

Child process 1 is killed by parent!!
^退出（核心已转储）
```

输入 `quit` 后

根据 `signal` 信号表可知，输入 `delete` 后，触发中断，会执行中断对应程序，然后在返回原先程序运行处继续运行，故仍会输出 “Parent process is killed!!”

而输入 `quit` 后，进程直接终止，也就不会输出 “Parent process is killed!!”

若想让程序运行界面上显示 “Child process 1 is killed by parent!! Child process 2 is killed by parent!!”，五秒之后显示 “Parent process is killed!!”，只有接收到相应的中断信号后再发生跳转，执行输出。

3. 针对实验过程 2，怎样修改的程序？修改前后程序的运行结果是什么？请截图说明。

修改代码后，代码参考 `interrupt2.c`

由于存在全局变量 `wait_flag`，在中断中将其值赋为 0。故在执行输出前，先给 `wait_flag` 赋值为 1，然后用 `while(wait_flag == 1);` 循环判断，直至中断产生再输出结果。

运行程序，程序开始等待键盘输入 `delete` 键或者 `quit` 键。

按下对应键后，程序运行完毕，父进程与子进程输出完毕。

```
yunkino@yunkino-virtual-machine:~/OS/lab2$ ./interrupt2
^C
中断信号数为: 2

中断信号数为: 16

中断信号数为: 2

Child process 1 is killed by parent!!

中断信号数为: 17

中断信号数为: 2

Child process 2 is killed by parent!!

Parent process is killed!!
```

输入 delete 键后

```
yunkino@yunkino-virtual-machine:~/OS/lab2$ ./interrupt2
^\\
中断信号数为: 3

中断信号数为: 17
◆中断信号数为: 3

Child process 2 is killed by parent!!

中断信号数为: 16

中断信号数为: 3

Child process 1 is killed by parent!!

Parent process is killed!!
```

输入 quit 键后

为了观察实际运行的先后情况,在父进程向子进程通过 kill()发送中断信号后,输出“kill 1/2 success”,在子进程中断信号初始化后,输出“signal 1/2 initial success”


```

yunkino@yunkino-virtual-machine:~/05/lab2$ ./interrupt2

signal 1 initail success

signal 2 initail success
^C
中断信号数为：2

Child process 1 is killed by parent!!

中断信号数为：2

kill 1 success

kill 2 success

中断信号数为：17

中断信号数为：2

中断信号数为：16
Child process 2 is killed by parent!!

Parent process is killed!!

```

从输出来看子进程中断先初始化成功，然后父进程“kill”子进程,最后父进程结束。

4. 针对实验过程 3，程序运行的结果是什么样子？时钟中断有什么不同？

代码参考 interrupt3.c

alarm(1)让进程在开始后一秒后发出中断。程序运行结果来看，等待大约一秒后出现输出。

```

yunkino@yunkino-virtual-machine:~/05/lab2$ ./interrupt3

signal 1 initail success

signal 2 initail success

中断信号数为：14

kill 1 success

中断信号数为：17

Child process 2 is killed by parent!!
kill 2 success

中断信号数为：16

Child process 1 is killed by parent!!

Parent process is killed!!

```

alarm 中断

5. 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几次，作用是让父进程向子进程通过 kill() 发送中断信号，

6. 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？

调用 `exit()` 或者 `exit`; (`exit()` 之后把控制权交给系统)

1. 你最初认为运行结果会怎么样？

2. 实际的结果怎么样? 有什么特点? 试对产生该现象的原因进行分析。

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？

管道为规模为二的数组，使用 `lockf(fd[i],1,0)` 锁定管道，`lockf(fd[i],0,0)`解除锁定，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。

2.9. 实验总结

2.9.1. 实验中的问题与解决过程

```
yunkino@yunkino-virtual-machine:~/OS/lab2$ ./interrupt2
^C
中断信号数为：2

中断信号数为：17

中断信号数为：2

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

如图，在发出 delete 中断信号后，子进程 2 正常结束，但子进程 1 异常，且此时输入光标闪烁，输入 delete 只会反复输出“中断信号数为：2”，程序不会结束。

修改代码在父进程发送完中断信号后，输出“kill 1/2”来查看中断信号是否正常发送。

```
yunkino@yunkino-virtual-machine:~/OS/lab2$ ./interrupt2
^C
中断信号数为：2

kill 1

kill 2

中断信号数为：16

中断信号数为：2

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

从结果来看，中断信号正常发送，但仍有子进程 2 未正常结束的情况。

猜测原因为：由于子进程与父进程同时运行，若子进程还未初始化接受父进程从 kill 传送的中断信号，而此时父进程已经发送过中断信号，则子进程无法正确收到中断信号，故子进程一直在等待父进程发送中断信号，而父进程又由于 wait() 函数挂起等待子进程结束，故此时出现了程序无法正常退出的情况。

故修改代码在子进程初始化中断信号后输出“signal 1/2 initial success”

但此后再也没出现这种问题。

2.9.2. 实验收获

在本次实验中，熟悉了进程的中断、管道机制和页面替换算法，更深入地理解了进程的结

束，实现了通过管道进行进程间通信，更熟练掌握了 FIFO 页面替换算法和 LRU 页面替换算法，编程能力进一步提高。

2.9.3. 意见与建议

实验的输出应该再详细一些，例如子进程的信号初始化是否成功。这样更方便观察结果，理解背后的运行逻辑。

2.10. 附件

2.10.1. 附件 1 程序

```
interrupt1.c
#include <stdio.h>
#include <stdlib.h>
#include <signal.h>
#include <unistd.h>
#include <sys/types.h>
#include <sys/wait.h>

int wait_flag;
void stop()
{
    wait_flag = 0;
}

int main()
{
    int pid1, pid2;
    signal(SIGINT, stop); // 父进程捕捉键盘上发出的中断信号 delete
    // signal(SIGQUIT, stop); // 父进程捕捉键盘上发出的中断信号 quit
    while ((pid1 = fork()) == -1)
        ; // 创建子进程 1 直到成功, 并获得进程号
    if (pid1 > 0) // 判断在父进程
    {
        while ((pid2 = fork()) == -1)
            ; // 创建子进程 2 直到成功, 并获得进程号
        if (pid2 > 0) // 判断在父进程
        {
            wait_flag = 1;
            sleep(5); // 父进程等待 5 秒
            kill(pid1, 16); // 杀死进程 1 发中断号 16
            kill(pid2, 17); // 杀死进程 2 发中断号 17
            wait(0); // 等待子进程 1 结束
            wait(0); // 等待子进程 2 结束
            printf("\nParent process is killed!!\n");
            exit(0);
        }
    }
}
```

```
}
else // 在子进程 2 里
{
    wait_flag = 1;
    signal(17, stop);
    printf("\nChild process 2 is killed by parent!!\n");
    exit(0);
}
}
else // 在子进程 1 里
{
    wait_flag = 1;
    signal(16, stop);
    printf("\nChild process 1 is killed by parent!!\n");
    exit(0);
}
}
```

interrupt2.c

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

int wait_flag;
void stop(int sig)
{
    wait_flag = 0;
    printf("\n 中断信号数为: %d\n", sig);
}

int main()
{
    int pid1, pid2;
    signal(SIGINT, stop); // 父进程捕捉键盘上发出的中断信号 delete
    signal(SIGQUIT, stop); // 父进程捕捉键盘上发出的中断信号 quit
    while ((pid1 = fork()) == -1)
        ; // 创建子进程 1 直到成功, 并获得进程号
    if (pid1 > 0) // 判断在父进程
    {
        while ((pid2 = fork()) == -1)
            ; // 创建子进程 2 直到成功, 并获得进程号
        if (pid2 > 0) // 判断在父进程
        {
            wait_flag = 1;
            while (wait_flag == 1)
                ; // 父进程等待捕捉键盘上发出的中断信号
            kill(pid1, 16); // 杀死进程 1 发中断号 16
            printf("\nkill 1 success\n");
            kill(pid2, 17); // 杀死进程 2 发中断号 17
            printf("\nkill 2 success\n");
        }
    }
}
```

```

        wait(0); // 等待子进程 1 结束
        wait(0); // 等待子进程 2 结束
        printf("\nParent process is killed!!\n");
        exit(0);
    }
    else // 在子进程 2 里
    {
        wait_flag = 1;
        signal(17, stop);
        printf("\nsignal 2 initial success\n");
        while (wait_flag == 1)
        ;
        printf("\nChild process 2 is killed by parent!!\n");
        exit(0);
    }
}
else // 在子进程 1 里
{
    wait_flag = 1;
    signal(16, stop);
    printf("\nsignal 1 initial success\n");
    while (wait_flag == 1)
    ;
    printf("\nChild process 1 is killed by parent!!\n");
    exit(0);
}
}

```

interrupt3.c

```

#include <stdio.h>
#include <stdlib.h>
#include <signal.h>
#include <unistd.h>
#include <sys/types.h>
#include <sys/wait.h>

```

```

int wait_flag;
void stop(int sig)
{
    wait_flag = 0;
    printf("\n 中断信号数为: %d\n", sig);
}

```

```

int main()
{
    int pid1, pid2;
    alarm(1);
    signal(SIGALRM, stop); // 父进程捕捉键盘上发出的中断信号 delete
    while ((pid1 = fork()) == -1)
    ; // 创建子进程 1 直到成功, 并获得进程号
    if (pid1 > 0) // 判断在父进程
    {
        while ((pid2 = fork()) == -1)

```

```

        ; // 创建子进程 2 直到成功, 并获得进程号
    if (pid2 > 0) // 判断在父进程
    {
        wait_flag = 1;
        while (wait_flag == 1)
            ; // 父进程等待捕捉键盘上发出的中断信号
        kill(pid1, 16); // 杀死进程 1 发中断号 16
        printf("\nkill 1 success\n");
        kill(pid2, 17); // 杀死进程 2 发中断号 17
        printf("\nkill 2 success\n");
        wait(0); // 等待子进程 1 结束
        wait(0); // 等待子进程 2 结束
        printf("\nParent process is killed!!\n");
        exit(0);
    }
    else // 在子进程 2 里
    {
        wait_flag = 1;
        signal(17, stop);
        printf("\nsignal 2 initial success\n");
        while (wait_flag == 1)
            ;
        printf("\nChild process 2 is killed by parent!!\n");
        exit(0);
    }
}
else // 在子进程 1 里
{
    wait_flag = 1;
    signal(16, stop);
    printf("\nsignal 1 initial success\n");
    while (wait_flag == 1)
        ;
    printf("\nChild process 1 is killed by parent!!\n");
    exit(0);
}
}

```

pipe.c

```

#include <stdio.h>
#include <sys/types.h>
#include <signal.h>
#include <unistd.h>
#include <stdlib.h>
#include <wait.h>

```

```

int pid1, pid2; // 定义两个进程变量

```

```

int main()

```

```

{
    int fd[2]; // fd[0]是读端, fd[1]是写端。
    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++) // 分 2000 次每次向管道写入字符' 1'
    {
        write(fd[1], &c1, 1);
    }
    sleep(5); // 等待读进程读出数据
    lockf(fd[0], 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++) // 分 2000 次每次向管道写入字符' 2'
        {
            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
        lockf(fd[0], 1, 0); // 从管道中读出 4000 个字符
        read(fd[0], InPipe, 4000);
        InPipe[4000] = '\0';
        lockf(fd[0], 1, 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
    }
}
}

```

```

fifo_lru.c
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <time.h>
#define ap 10
#define pp 4
#define total_instruction 15

```

```

struct Queue
{

```



```

    int head;
    int rear;
    int num[pp + 1];
} queue = {0, 0};

struct Stack
{
    int top;
    int num[pp + 1];
} stack = {0};

void FIFO(int cur, int *page, int *pagecontrol)
{
    int empty = -1; // 用来记录 pagecontrol 是否有空, 有空则为从左往右第一个空对应下
    标, 无空则为-1
    for (int i = 0; i < pp; i++)
    {
        if (pagecontrol[i] == -1)
        {
            empty = i;
            break;
        }
    }
    if (empty == -1)
    {
        page[pagecontrol[queue.num[queue.head]]] = 0; // 将先进入队列的页面从内存中
        清出
        pagecontrol[queue.num[queue.head]] = cur; // 将新页面插入
        page[cur] = 1;
        queue.num[queue.rear] = queue.num[queue.head];
        queue.head = (queue.head + 1) % (pp + 1);
        queue.rear = (queue.rear + 1) % (pp + 1);
    }
    else // pagecontrol 中有空位, 直接放入
    {
        pagecontrol[empty] = cur;
        page[cur] = 1;
        // 队列按照在当前内存中进入顺序记录, 记录的是 pagecontrol 的下标。
        queue.num[queue.rear] = empty;
        queue.rear = (queue.rear + 1) % (pp + 1);
    }
}

void LRU(int cur, int *page, int *pagecontrol)
{
    // 先检查在不在 pagecontrol 里面
    for (int i = 0; i < pp; i++)
    {
        if (pagecontrol[i] == cur)
        {
            int j;
            for (j = 0; j < pp; j++)
            {

```

```

        if (stack.num[j] == i)
            break;
    }
    for (; j < stack.top - 1; j++)
    {
        stack.num[j] = stack.num[j + 1];
    }
    stack.num[stack.top - 1] = i;
    return;
}
}
int empty = -1; // 用来记录 pagecontrol 是否有空, 有空则为从左往右第一个空对应下
标, 无空则为-1
for (int i = 0; i < pp; i++)
{
    if (pagecontrol[i] == -1)
    {
        empty = i;
        break;
    }
}
if (empty == -1) // 如果没有空位, 就将最近使用最少的页面替换为新页面
{
    page[pagecontrol[stack.num[0]]] = 0;
    pagecontrol[stack.num[0]] = cur;
    page[cur] = 1;
    int tmp = stack.num[0];
    for (int i = 0; i < stack.top - 1; i++)
    {
        stack.num[i] = stack.num[i + 1];
    }
    stack.num[stack.top - 1] = tmp;
}
else
{
    pagecontrol[empty] = cur;
    page[cur] = 1;
    stack.num[stack.top] = empty;
    stack.top++;
}
}

int main()
{
    int page[ap], pagecontrol[pp];
    int result[total_instruction][pp]; // 用于记录页面替换过程
    int algorithm;                      // 1 代表 FIFO 算法, 2 代表 LRU 算法
    for (int i = 0; i < pp; i++)
    {
        pagecontrol[i] = -1;
    }
    for (int i = 0; i < ap; i++)
    {

```

```
    page[i] = 0;
}
// int main[total_instruction] = {0, 5, 6, 4, 8, 8, 3, 5, 8, 7, 5, 8, 0, 4, 3};
int main[total_instruction];
srand(time(0));
for (int i = 0; i < total_instruction; i++)
{
    main[i] = rand() % ap;
}
printf("进程页面数为: %d\n", ap);
printf("内存分配的页面数为: %d\n", pp);
printf("页面访问顺序为: ");
for (int i = 0; i < total_instruction; i++)
{
    printf("%d ", main[i]);
}
printf("\n");
printf("请输入替换算法类型: 1 为 FIFO, 2 为 LRU. \n");
scanf("%d", &algorithm);
while (algorithm != 1 && algorithm != 2)
{
    printf("不支持此类算法, 请重新选择: \n");
    scanf("%d", &algorithm);
}
if (algorithm == 1)
    printf("当前算法为 FIFO 算法\n");
else
    printf("当前算法为 LRU 算法\n");
int cur = 0;
int diseffect = 0; // diseffect 是未命中次数
while (cur != total_instruction)
{
    if (page[main[cur]] == 1)
    {
        if (algorithm == 2)
        {
            LRU(main[cur], page, pagecontrol);
        }
    }
    else
    {
        diseffect++;
        if (algorithm == 1)
        {
            FIFO(main[cur], page, pagecontrol);
        }
        else
        {
            LRU(main[cur], page, pagecontrol);
        }
    }
    for (int i = 0; i < pp; i++)
    {
```

```

        result[cur][i] = pagecontrol[i];
    }
    cur++;
}
double hit_rate = 100 * (1 - ((double)diseffect / total_instruction));
for (int i = 0; i < total_instruction; i++)
{
    printf("%d", result[i][0]);
    for (int j = 1; j < pp; j++)
    {
        printf("\t%d", result[i][j]);
    }
    printf("\n");
}
printf("未命中次数为: %d\n", diseffect);
printf("命中率为: %.2f%%\n", hit_rate);
return 0;
}

```

2.10.2. 附件 2 Readme

实验二 Linux 进程通信与内存管理

1 进程的软中断通信

1.1 进程的软中断通信——delete 与 quit

代码参考 interrupt1.c

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（即按 delete 与 quit, 分别对应 ctrl+c 与 ctrl+\），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，输入以下信息，结束进程执行：Parent process is killed!!

![运行结果](./运行结果.png)

运行过程中，首先出现“Child process 1 is killed by parent!!”“Child process 2 is killed by parent!!”两条结果，且两个结果输出的先后顺序是不确定的。几秒后，出现结果“Parent process is killed!!”。

分析可知，两子进程的运行先后顺序并不确定，因为两个子进程同时运行，并没有先后的区别。但由于 wait(0) 和 sleep(5) 函数的存在，父进程会挂起等到两子进程结束后在进行。故父进程输出的结果在最后。

子进程结果输出后，父进程结果输出前，键盘输入 delete 或者 quit，分别出现以下情况

![输入 delete 后](./输入 delete 后.png)

输入 delete 后

![输入 quit 后](./输入 quit 后.png)

输入 quit 后

根据 signal 信号表可知，输入 delete 后，触发中断，会执行中断对应程序，然后在返回原先程序运行处继续运行，故仍会输出“Parent process is killed!!”

而输入 quit 后, 进程直接终止, 也就不会输出 “Parent process is killed!!”

若想让程序运行界面上显示 “Child process 1 is killed by parent!! Child process 2 is killed by parent!!”, 五秒之后显示 “Parent process is killed!!”, 只有接收到相应的中断信号后再发生跳转, 执行输出。

修改代码后, 代码参考 interrupt2.c

由于存在全局变量 wait_flag, 在中断中将其值赋为 0。故在执行输出前, 先给 wait_flag 赋值为 1, 然后用 while(wait_flag == 1); 循环判断, 直至中断产生再输出结果。

运行程序, 程序开始等待键盘输入 delete 键或者 quit 键。

按下对应键后, 程序运行完毕, 父进程与子进程输出完毕。

![等待键盘输入](./等待键盘输入.png)

等待键盘输入

![输入 delete 键后](./输入 delete 后 2.png)

输入 delete 键后

![输入 delete 键后](./输入 quit 后 2.png)

输入 quit 键后

为了观察实际运行的先后情况, 在父进程向子进程通过 kill() 发送中断信号后, 输出 “kill 1/2 success”, 在子进程中断信号初始化后, 输出 “signal 1/2 initial success”

![加入输出检验进程顺序](./加入输出检验进程顺序.png)

从输出来看子进程中断先初始化成功, 然后父进程 “kill” 子进程, 最后父进程结束。

1.2 进程的软中断通信——alarm

代码参考 interrupt3.c

alarm(1) 让进程在开始后一秒后发出中断。程序运行结果来看, 等待大约一秒后出现输出。

![alarm 中断](./alarm 中断.png)

alarm 中断

2 进程的管道通信

参考代码 pipe.c

管道通信是指用于连接一个读进程和一个写进程以实现他们之间通信的一个共享文件, 又名 pipe 文件。向管道(共享文件)提供输入的发送进程(即写进程), 以字符流形式将大量的数据送入管道; 而接受管道输出的接收进程(即读进程), 则从管道中接收(读)数据。创建 pipe 管道来让两个子进程与父进程完成通信, 子进程 1 向父进程发送 2000 个 1, 子进程 2 向父进程发送 2000 个 2。

预估输出会为 2000 个 1 和 2000 个 2, 1 和 2 的顺序不一定。由于锁机制的存在, 并不会存在 1、2 交替输出的问题。

![pipe 运行结果](./pipe 运行结果.png)

pipe 运行结果

程序运行中, 由于 pipe 读写的互锁产生延时, 在写完成后再读。故程序运行一段时间后才会有输出。

在去除互锁及之后, 预计将会出现 1, 2 交替输出的情况。

![pipe 去除互锁后的结果](./pipe 去除互锁后的结果.png)

pipe 去除互锁后的结果

从结果来看, 子进程 2 先进运行, 随后子进程 1 开始运行后出现了 1, 2 交替输出的情况。

3 页面的替换

参考代码 fifo_lru.c

页面替换实现 FIFO 算法和 LRU 算法。

FIFO 算法其原理为：

在当前内存分配的页面数未满足时，将新加入的页面直接放在空位中。

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先加入的页面。

LRU 算法其原理为：

在当前内存分配的页面数未满足时，将新加入的页面直接放在空位中。

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。

其中 FIFO 算法由于其先进先出的特性，用队列来实现。

LRU 算法则需要按页面的访问次数进行排序，则用栈对页面访问频繁程度进行更新。

分别 `page[ap]` 和 `pagecontrol[pp]` 分别表示进程页面数和内存分配的页面数

定义常量 `total_instruction` 来记录页面总共使用的次数，变量 `diseffect` 记录总共换入页面的次数（需要换出页面总是因为缺页中断而产生）。

利用公式 $1 - \text{diseffect} / \text{total_instruction} * 100\%$ 可以得到命中率。

![FIFO 算法的结果](./FIFO 算法的结果.png)

FIFO 算法的结果

![LRU 算法的结果](./LRU 算法的结果.png)

LRU 算法的结果

4 遇到的问题

![异常情况，子进程 1 中断无效](./异常情况，子进程 1 中断无效.png)

如图，在发出 `delete` 中断信号后，子进程 2 正常结束，但子进程 1 异常，且此时输入光标闪烁，输入 `delete` 只会反复输出“中断信号数为：2”，程序不会结束。

修改代码在父进程发送完中断信号后，输出“kill 1/2”来查看中断信号是否正常发送。

![异常情况，输出调试](./异常情况，输出调试.png)

从结果来看，中断信号正常发送，但仍有子进程 2 未正常结束的情况。

猜测原因为：由于子进程与父进程同时运行，若子进程还未初始化接受父进程从 `kill` 传送的中断信号，而此时父进程已经发送过中断信号，则子进程无法正确收到中断信号，故子进程一直在等待父进程发送中断信号，而父进程又由于 `wait()` 函数挂起等待子进程结束，故此时出现了程序无法正常退出的情况。

故修改代码在子进程初始化中断信号后输出“signal 1/2 initial success”

但此后再也没出现这种问题。

3. Linux 的动态模块与设备驱动

3.1. 实验目的

理解 LINUX 字符设备驱动程序的基本原理；

掌握字符设备的驱动运作机制；

学会编写字符设备驱动程序

3.2. 实验内容

编写一个简单的字符设备驱动程序，以内核空间模拟字符设备，完成对该设备的打开，读写和释放操作，并编写聊天程序实现对该设备的同步和互斥操作。

3.3. 实验思想

设备驱动程序

设备驱动程序是内核和硬件设备之间的接口，设备驱动程序屏蔽硬件细节，且设备被映射成特殊的文件进行处理。每个设备都对应一个文件名，在内核中也对应一个索引节点，应用程序可以通过设备的文件名来访问硬件设备。Linux 为文件和设备提供了一致性的接口，用户操作设备文件和操作普通文件类似。例如通过 `open()` 函数可打开设备文件，建立起应用程序与目标程序的连接；之后，可以通过 `read()`、`write()` 等常规文件函数对目标设备进行数据传输操作。设备驱动程序封装了如何控制设备的细节，它们可以使用和操作文件相同的、标准的系统调用接口来完成打开、关闭、读写和 I/O 控制等操作，而驱动程序的主要任务也就是要实现这些系统调用函数。设备驱动程序是一些函数和数据结构的集合，这些函数和数据结构是为实现设备管理的一个简单接口。操作系统内核使用这个接口来请求驱动程序对设备进行 I/O 操作。

字符驱动程序相关数据结构

字符设备是以字节为单位逐个进行 I/O 操作的设备，不经过系统的 I/O 缓冲区，所以需要管理自己的缓冲区结构。

在字符设备驱动程序中，主要涉及 3 个重要的内核数据结构，分别是 `file_operations`、`file` 和 `inode`。当用户访问设备文件时，每个文件的 `file` 结构都有一个索引节点 `inode` 与之对应。在内核的 `inode` 结构中，有一个名为 `i_fop` 成员，其类型为 `file_operations`。`file_operations` 定义文件的各种操作，用户对文件进行诸如 `open`、`close`、`read`、`write` 等操作时，Linux 内核将通过 `file_operations` 结构访问驱动程序提供的函数。内核通过这 3 个数据结构的关联，将用户对设备文件的操作转换为对驱动程序相关函数的调用。

file 代表一个打开了的文件。它由内核在使用 open() 函数时建立，并传递给该文件上操作的所有函数，直到最后的 close() 函数。当文件的所有操作结束后，内核会释放该数据结构。

内核用 inode 结构在内部标识文件，它和 file 结构不同，后者标识打开的文件描述符。对于单个文件，可能会有许多个表示打开的文件描述符的 file 结构，但它们都指向同一个 inode 结构，inode 结构中有两个重要的字段：

```
struct cdev *i_cdev;
dev_t i_rdev;
```

类型 dev_t 描述设备号，cdev 结构表示字符设备的内核数据结构，内核不推荐开发者直接通过结构的 i_rdev 结构获得主次设备号，而提供下面两个函数取得主、次设备号。

```
Unsigned int iminor(struct inode *inode);
Unsigned int imajor(struct inode *inode);
```

设备文件的创建

实验中需要对一个设备文件进行操作，这里有两种方案，一是在终端中人工使用命令进行创建，二是直接在模块程序中予以创建。

并发控制

在驱动程序中，当多个线程同时访问相同的资源时（驱动程序中的全局变量是一种典型的共享资源），可能会引发“竞争”，因此必须对共享资源进行并发控制。在此驱动程序中，可用信号量机制来实现并发控制。

相关结构与函数

设备驱动程序结构

字符设备的结构描述如下：

```
struct Scull_Dev{ //驱动程序结构体
    struct cdev devm; //字符设备
    struct semaphore sem; //信号量，实现读写时的 PV 操作
    wait_queue_head_t outq; //等待队列，实现阻塞操作
    int flag; //阻塞唤醒条件
    char buffer[MAXNUM+1]; //字符缓冲区
    char *rd,*wr,*end; //读，写，尾指针
};
```

scull (simple character utility for loading localities, “区域装载的简单字符工具”) 是一个操作内存区域的字符设备驱动程序，这片内存区域就相当于一个字符设备。scull 的优点在于他不和任何硬件相关，而只是操作从内核分配的一些内存。任何人都可以编译和运行 scull, 而且还看看可以将 scull 移植到 linux 支持的所有计算机平台上。但另一方面，除了

展示内核和字符设备驱动程序之间的接口并且让用户运行某些测试例程外，scull 设备做不了任何“有用”的事情。

字符设备的数据接口

字符设备的数据接口，将文件的读、写、打开、释放等操作映射为相应的函数。

```
struct file_operations globalvar_fops =
{
    .read=globalvar_read,
    .write=globalvar_write,
    .open=globalvar_open,
    .release=globalvar_release,
};
```

字符设备的注册与注销

字符设备的注册采用静态申请和动态分配相结合的方式，使用 `register_chrdev_region` 函数和 `alloc_chrdev_region` 函数来完成。字符设备的注销采用 `unregister_chrdev_region` 函数来完成。

字符设备的打开与释放

打开设备是通过调用 `file_operations` 结构中的函数 `open()` 来完成的。设备的打开提供给驱动程序初始化的能力，从而为以后的操作准备，此外还会递增设备的使用计数，防止在文件被关闭前被卸载出内核。

释放设备是通过调用 `file_operations` 结构中的函数 `release()` 来完成的。设备的释放作用刚好与打开相反，但基本的释放操作只包括设备的使用计数递减。

字符设备的读写操作

直接使用函数 `read()` 和 `write()`。文件读操作的原型如下：

```
Ssize_t device_read(struct file* filp, char __user* buff, size_t len, loff_t*
offset);
```

其中，`filp` 是文件对象指针；`buff` 是用户态缓冲区，用来接受读到的数据；`len` 是希望读取的数据量；`offset` 是用户访问文件的当前偏移。文件写操作的原型和读操作没有区别，只是操作方向改变而已。由于内核空间与用户空间的内存不能直接互访，因此借助函数 `copy_to_user()` 完成用户空间到内核空间的复制，函数 `copy_from_user()` 完成内核空间到用户空间的复制。

3.4. 实验步骤

3.4.1. 编写模块程序

首先编写模块程序，命名为 `globalvar.c`。

3.4.2. 定义虚拟字符设备驱动

struct Scull_Dev globalvar; 定义了一个虚拟字符设备驱动 globalvar。

声明虚拟字符设备的读、写、打开和释放操作函数

```
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 int globalvar_open(struct inode *inode,struct file *filp);
static int globalvar_release(struct inode *inode,struct file *filp);
```

将文件的读、写、打开、释放等操作映射为虚拟字符设备的函数

```
struct file_operations globalvar_fops =
{
    .read=globalvar_read,
    .write=globalvar_write,
    .open=globalvar_open,
    .release=globalvar_release,
};
```

3.4.3. 对设备进行初始化

函数 static int globalvar_init(void)对设备进行初始化。一是获取设备号，使用 dev_t dev = MKDEV(major, 0);定义一个设备号，用

```
if(major)
{
    //静态申请设备编号
    //第一个参数表示设备号，第二个参数表示注册的此设备数目，
    //第三个表示设备名称。
    result = register_chrdev_region(dev, 1, "charmeme");
}
else
{
    //动态分配设备号
    //第一个参数保存生成的设备号，第二个参数表示次设备号的基准，
    //即从哪个次设备号开始分配，第三个表示注册的此设备数目，
    //第四个表示设备名称。
    result = alloc_chrdev_region(&dev, 0, 1, "charmeme");
    major = MAJOR(dev);
}
```

//返回值：小于 0，则自动分配设备号错误。否则分配得到的设备号就被&dev 带出来。

```
if(result < 0)
```

```
return result;
```

申请或分配设备号。用 `class_create` 和 `device_create` 创建设备文件。

```
my_class = class_create(THIS_MODULE, "chardev0");
```

```
device_create(my_class, NULL, dev, NULL, "chardev0");
```

3.4.4. 向内核里面注册驱动

第一个输入参数代表即将被添加入 Linux 内核系统的字符设备，第二个输入参数是 `dev_t` 类型的变量，此变量代表设备的设备号，第三个输入参数是无符号的整型变量，代表想注册设备的设备号的范围。如果成功，则返回 0，如果失败，则返回 `ENOMEM`，`ENOMEM` 的被定义为 12。

```
err = cdev_add(&globalvar.devm, dev, 1);
```

```
if(err)
```

```
    printk(KERN_INFO "Error %d adding char_mem device", err);
```

```
else
```

```
{
```

```
    //设备注册成功
```

```
    printk("globalvar register success\n");
```

```
    sema_init(&globalvar.sem, 1); //初始化信号量
```

```
    init_waitqueue_head(&globalvar.outq); //初始化等待队列
```

```
    globalvar.rd = globalvar.buffer; //读指针
```

```
    globalvar.wr = globalvar.buffer; //写指针
```

```
    globalvar.end = globalvar.buffer + MAXNUM; //缓冲区尾指针
```

```
    globalvar.flag = 0; // 阻塞唤醒标志置 0
```

```
}
```

3.4.5. 打开操作

具体为模块计数加一。

```
static int globalvar_open(struct inode *inode, struct file *filp)
```

```
{
```

```
    //如果该模块处于活动状态且对它引用计数加 1 操作正确则返回 1，否则返回 0.
```

```
    try_module_get(THIS_MODULE); //模块计数加一
```

```
    printk("This chrdev is in open\n");
```

```
    return(0);
```

```
}
```

3.4.6. 释放操作

具体为模块计数减一。

```
static int globalvar_release(struct inode *inode, struct file *filp)
```

```

{
    module_put(THIS_MODULE); //模块计数减一
    printk("This chrdev is in release\n");
    return(0);
}

```

3.4.7. 注销设备操作

具体为用 `device_destroy` 注销创建的设备，用 `class_destroy` 注销设备类，用 `cdev_del` 释放 `cdev` 结构体空间，用 `unregister_chrdev_region` 来注销设备号。

```

static void globalvar_exit(void)
{
    //注销设备
    device_destroy(my_class, MKDEV(major, 0));
    class_destroy(my_class);
    cdev_del(&globalvar.devm);
    unregister_chrdev_region(MKDEV(major, 0), 1); //注销设备
}

```

3.4.8. 读操作

`globalvar.flag` 标志当前是否可读，若不可读，则用 `wait_event_interruptible` 将其挂起到等待队列 `globalvar.outq`。如果可以读，则使用 `down_interruptible(&globalvar.sem)` 进行 P 操作。接下来更新读指针，`len` 是读的字节数，如果读指针小于写指针，表明新写入了内容，应该读取从写指针到读指针的内容，即 `len = min(len, (size_t)(globalvar.wr - globalvar.rd))`。如果读指针大于等于写指针，表明在循环缓冲区中写指针已经过了一次循环，应令 `len = min(len, (size_t)(globalvar.end - globalvar.rd))` 将当前读指针到结尾的内容读完，下一次再读到写指针。用 `copy_to_user(buf, globalvar.rd, len)` 函数将内核空间的数据读取出来，并更新读指针位置，如果读指针在缓冲区末尾则将其循环地置为缓冲区首部。最后进行 V 操作退出临界区。

3.4.9. 写操作

写操作和读操作大致相同，只不过读写方向相反。首先用 P 操作进入临界区，计算写入长度 `len`。如果读指针小于写指针，则可以令 `len = min(len, (size_t)(globalvar.end - globalvar.wr))`；表示从当前写指针到缓冲区末尾都可以写入。如果读指针大于写指针，则只能写入从写指针到读指针之前的位置，即 `len = min(len, (size_t)(globalvar.rd - globalvar.wr - 1))`；，否则会破坏还未读取的内容，读取时也不能读取写指针后面、本应读取到的内容。最后更新写指针，进行 V 操作退出临界区，通过 `wake_up_interruptible(&globalvar.outq)` 唤醒读进程。采用这种方式实际上写入的优先级更高。

3.4.10. 模块程序

将以上部分汇总得到 `globalvar.c`。

3.4.11. 编写读程序

见代码 `read.c`。

3.4.12. 编写写程序

见代码 `write.c`。

3.4.13. 编写 Makefile 文件

为了编译模块程序 `globalvar.c`，建立 `makefile` 文件。

3.4.14. 编译模块程序 `globalvar.c`

使用 `make` 命令编译模块文件。

3.4.15. 加载模块

使用命令 `insmod globalvar.ko` 加载模块程序，值得注意的是加载模块程序需要 `root` 权限。加载模块程序如图。

3.4.16. 编译并打开读写程序

打开后在写窗口写入，便可以在读窗口看到输出内容。

但是这样不能实现两个窗口的通信，为此我们需要编写既能读又能写的程序。

3.4.17. 编写读写程序

编写读写程序 `read_write.c`，

3.4.18. 卸载模块

使用 `rmmmod globalvar` 卸载模块。

3.5. 程序运行初值及运行结果分析

3.5.1. 动态模块

代码参考 `mymoules.c` 和 `Makefile`

在动态模块中，在模块加载成功时，在日志中输出 “hello, my module wored!”，在模块卸载时在日志中输出 “goodbye, unloading my module.”

使用 `make` 命令编译模块，用 `inmod` 命令加载模块，用 `rmmmod` 命令卸载模块，用 `demsg` 命令查看日志。

```

yunkino@yunkino-virtual-machine:~/OS/lab3$ make
make -C /lib/modules/5.15.0-53-generic/build M=/home/yunkino/OS/lab3 modules
make[1]: 进入目录“/usr/src/linux-headers-5.15.0-53-generic”
warning: the compiler differs from the one used to build the kernel
The kernel was built by: gcc (Ubuntu 11.2.0-19ubuntu1) 11.2.0
You are using:          gcc (Ubuntu 11.3.0-1ubuntu1~22.04) 11.3.0
CC [M] /home/yunkino/OS/lab3/mymodules.o
MODPOST /home/yunkino/OS/lab3/Module.symvers
CC [M] /home/yunkino/OS/lab3/mymodules.mod.o
LD [M] /home/yunkino/OS/lab3/mymodules.ko
BTF [M] /home/yunkino/OS/lab3/mymodules.ko
Skipping BTF generation for /home/yunkino/OS/lab3/mymodules.ko due to unavailability of vmlinux
make[1]: 离开目录“/usr/src/linux-headers-5.15.0-53-generic”

```

make 运行结果

```

yunkino@yunkino-virtual-machine:~/OS/lab3$ sudo insmod mymodules.ko
[sudo] yunkino 的密码:
yunkino@yunkino-virtual-machine:~/OS/lab3$ lsmod
Module                Size  Used by
mymodules             16384  0
intel_rapl_msr        20480  0

```

加载模块成功

```

[11217.504121] mymodules: loading out-of-tree module taints kernel.
[11217.504314] mymodules: module verification failed: signature and/or required key missing - tainting kernel
[11217.504970] hello,my module wored!

```

模块加载后的日志记录

```

[11754.629842] goodbye,unloading my module.

```

模块卸载后的日志记录

3.5.2. 系统调用篡改

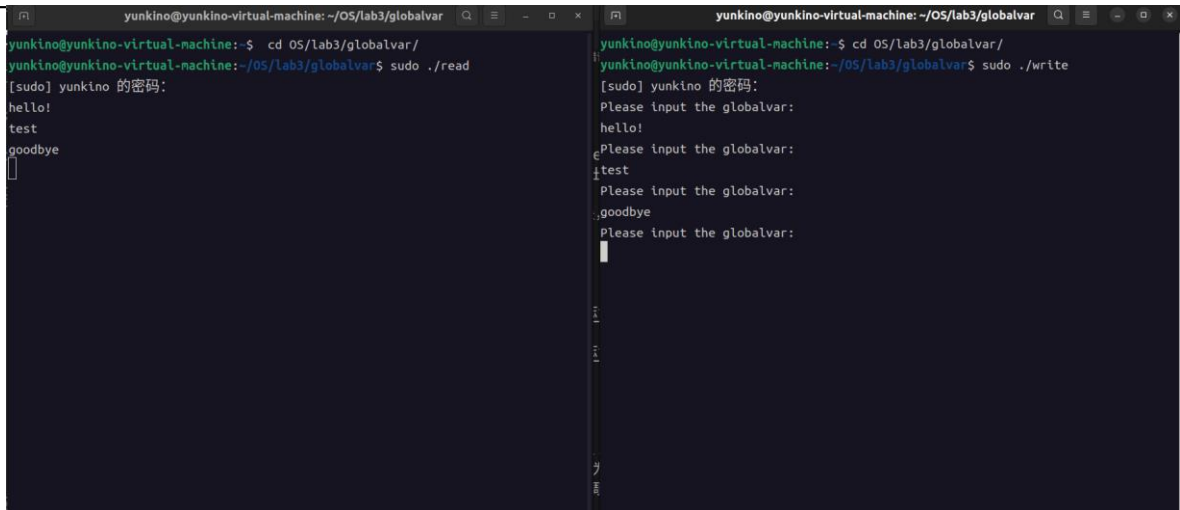
代码参考 `modify_syscall.c`, `Makefile`, `modify_new_syscall.c` 和 `modify_old_syscall.c`

尝试篡改系统调用 `gettimeofday`，使其调用改为输出三个参数之和。`modify_new_syscall` 用于测试篡改后的系统调用，`modify_old_syscall` 用于测试篡改前的系统调用。详见实验中间题与解决过程。

3.5.3. 字符设备驱动程序

代码参考 `globalvar.c`, `Makefile`, `read.c`, `write.c` 和 `read_write.c`

编写一个简单的字符设备驱动程序，以内核空间模拟字符设备，完成对该设备的打开，读写和释放操作，并编写聊天程序实现对该设备的同步和互斥操作。

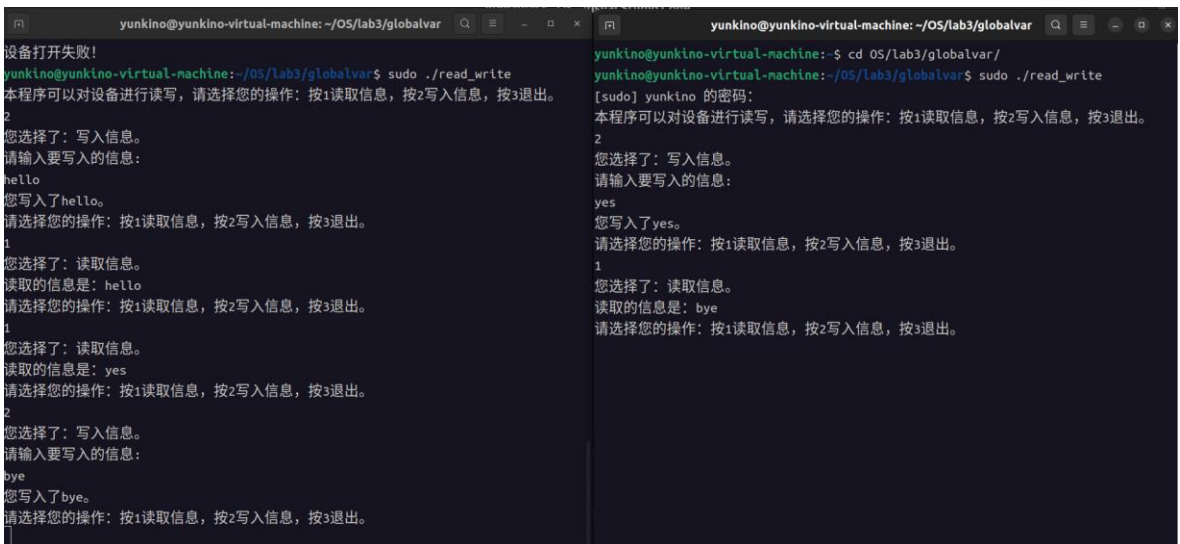


```
yunkino@yunkino-virtual-machine: ~/OS/lab3/globalvar
yunkino@yunkino-virtual-machine:~/OS/lab3/globalvar$ sudo ./read
[sudo] yunkino 的密码:
hello!
test
goodbye
[]

yunkino@yunkino-virtual-machine:~/OS/lab3/globalvar
yunkino@yunkino-virtual-machine:~/OS/lab3/globalvar$ sudo ./write
[sudo] yunkino 的密码:
Please input the globalvar:
hello!
Please input the globalvar:
test
Please input the globalvar:
goodbye
Please input the globalvar:
[]
```

在两个终端通信

聊天是一方可以说，也可以接收，故实现在同一终端中同时读写。



```
yunkino@yunkino-virtual-machine:~/OS/lab3/globalvar
yunkino@yunkino-virtual-machine:~/OS/lab3/globalvar$ sudo ./read_write
设备打开失败!
本程序可以对设备进行读写, 请选择您的操作: 按1读取信息, 按2写入信息, 按3退出。
2
您选择了: 写入信息。
请输入要写入的信息:
hello
您写入了hello。
请选择您的操作: 按1读取信息, 按2写入信息, 按3退出。
1
您选择了: 读取信息。
读取的信息是: hello
请选择您的操作: 按1读取信息, 按2写入信息, 按3退出。
1
您选择了: 读取信息。
读取的信息是: yes
请选择您的操作: 按1读取信息, 按2写入信息, 按3退出。
2
您选择了: 写入信息。
请输入要写入的信息:
bye
您写入了bye。
请选择您的操作: 按1读取信息, 按2写入信息, 按3退出。

yunkino@yunkino-virtual-machine:~/OS/lab3/globalvar
yunkino@yunkino-virtual-machine:~/OS/lab3/globalvar$ sudo ./read_write
[sudo] yunkino 的密码:
本程序可以对设备进行读写, 请选择您的操作: 按1读取信息, 按2写入信息, 按3退出。
2
您选择了: 写入信息。
请输入要写入的信息:
yes
您写入了yes。
请选择您的操作: 按1读取信息, 按2写入信息, 按3退出。
1
您选择了: 读取信息。
读取的信息是: bye
请选择您的操作: 按1读取信息, 按2写入信息, 按3退出。
```

在两个终端聊天

3.6. 实验总结

3.6.1. 实验中的问题与解决过程

3.6.1.1. make 命令错误

```
yunkino@yunkino-virtual-machine:~/OS/lab3$ make
make -C /lib/modules/5.15.0-53-generic/build M=/home/yunkino/OS/lab3 modules
make[1]: 进入目录“/usr/src/linux-headers-5.15.0-53-generic”
warning: the compiler differs from the one used to build the kernel
The kernel was built by: gcc (Ubuntu 11.2.0-19ubuntu1) 11.2.0
You are using: gcc (Ubuntu 11.3.0-1ubuntu1~22.04) 11.3.0
scripts/Makefile.build:44: /home/yunkino/OS/lab3/Makefile: 没有那个文件或目录
make[2]: *** 没有规则可制作目标“/home/yunkino/OS/lab3/Makefile”。 停止。
make[1]: *** [Makefile:1903: /home/yunkino/OS/lab3] 错误 2
make[1]: 离开目录“/usr/src/linux-headers-5.15.0-53-generic”
make: *** [makefile:7: modules] 错误 2
```

报错信息说找不到 Makefile 文件，考虑文件名大小写敏感，修改 makefile 文件名为 Makefile 后解决。

3.6.1.2. 加载模块错误

```
yunkino@yunkino-virtual-machine:~/OS/lab3$ insmod mymodules.ko
insmod: ERROR: could not insert module mymodules.ko: Operation not permitted
```

权限不足，把命令 insmod 改为 sudo insmod

```
yunkino@yunkino-virtual-machine:~/OS/lab3$ sudo insmod mymodules.ko
[sudo] yunkino 的密码:
yunkino@yunkino-virtual-machine:~/OS/lab3$ lsmod
Module                Size  Used by
mymodules              16384  0
intel_rapl_msr         20480  0
```

3.6.1.3. 输出时间错误

```
yunkino@yunkino-virtual-machine:~/OS/lab3/modify_syscall$ gcc modify_old_syscall.c -o modify_old_syscall
modify_old_syscall.c: In function 'main':
modify_old_syscall.c:8:21: warning: format '%d' expects argument of type 'int', but argument 2 has type '__time_t' {aka 'long int'} [-Wformat=]
   8 |     printf("tv_sec:%d\n",tv.tv_sec);
      |                ~^      ~~~~~
      |                |      |
      |                int   __time_t {aka long int}
      |                %ld
modify_old_syscall.c:9:22: warning: format '%d' expects argument of type 'int', but argument 2 has type '__suseconds_t' {aka 'long int'} [-Wformat=]
   9 |     printf("tv_usec:%d\n",tv.tv_usec);
      |                ~^      ~~~~~
      |                |      |
      |                int   __suseconds_t {aka long int}
      |                %ld
```


其类型为 long int，将%d 改为%ld。

3.6.1.4. 系统篡改错误

直接使用 ppt 中系统篡改代码，在模块加载后，运行 modify_new_syscall，得到返回值为-1, 运行 modify_old_syscall，得到返回值仍为原先的 gettimeofday 的结果。在尝试卸载模块时，发生错误，提示模块正在被使用，在查找资料以后，尝试强制卸载模块并重启虚拟机。后来在查找资料中发现错误，在 modify_syscall.c 中，用 unsigned long p_sys_call_table=0xc0361860;来表示系统调用表的起始地址，但在我的虚拟机上的系统调用表的起始地址不同，如图应为 0xffffffff8aa00300。而且实例代码中 gettimeofday 的系统调用号为 78,但在实际系统调用表中，32 位系统对应的 unistd_32.h 中为系统调用号为 78，而 64 位系统对应的 unistd_64.h 中系统调用号为 96。

```
#define __NR_getrusage 77
#define __NR_gettimeofday 78
#define __NR_settimeofday 79
#define __NR_getgroups 80
#define __NR_setgroups 81
```

unistd_32.h 中系统调用号

```
usr > include > x86_64-linux-gnu > asm > C unistd_64.h > __NR_getrusage
99  #define __NR_umask 95
100 #define __NR_gettimeofday 96
101 #define __NR_getrlimit 97
102 #define __NR_getrusage 98
```

unistd_64.h 中系统调用号

```
yunkino@yunkino-virtual-machine:/boot$ sudo cat /proc/kallsyms | grep sys_call_t
able
[sudo] yunkino 的密码:
ffffffff8aa00300 D sys_call_table
ffffffff8aa01420 D ia32_sys_call_table
ffffffff8aa02240 D x32_sys_call_table
```

查找系统调用表的起始地址

但在多次实验中发现，此法获得的系统调用表的地址是动态变化的。

在查询资料后得知使用 kallsyms_lookup_name () 函数动态获取地址。

但产生了如下报错

```
kallsyms_lookup_name undefined
```

```
ERROR: modpost: "kallsyms_lookup_name" [/home/yunkino/OS/lab3/modify_syscall/mod
ify_syscall.ko] undefined!
```

查询资料后得知

`[https://blog.csdn.net/Linux_Everything/article/details/104912356`](https://blog.csdn.net/Linux_Everything/article/details/104912356)

[https://blog.csdn.net/Mr0cheng/article/details/104890981`](https://blog.csdn.net/Mr0cheng/article/details/104890981)

out-of-tree 的驱动不能够利用 kallsyms_lookup_name 函数调用未导出的内核函数。Will Deacon 提交了一组 patch set，移除了对 kallsyms_lookup_name() 和 kallsyms_on_each_symbol() 的 export 声明。

固转尝试使用 Kprobe 方法

```
int noop_pre(struct kprobe *p, struct pt_regs *regs) { return 0; }
```

```
static struct kprobe kp = {  
    .symbol_name = "kallsyms_lookup_name",  
};
```

```
unsigned long *p_sys_call_table = NULL;
```

```
// 调用 kprobe 找到 kallsyms_lookup_name 的地址位置
```

```
int find_kallsyms_lookup_name(void)  
{  
    int ret = -1;  
    kp.pre_handler = noop_pre;  
    ret = register_kprobe(&kp);  
    if (ret < 0) {  
        printk(KERN_INFO "register_kprobe failed, error:%d\n", ret);  
        return ret;  
    }  
    printk(KERN_INFO "kallsyms_lookup_name addr: %p\n", kp.addr);  
    p_sys_call_table = (unsigned long*)kp.addr;  
    unregister_kprobe(&kp);  
    return ret;  
}
```

但仍未能产生正确结果，这是系统日志文件输出的错误

```
[ 151.211228] [drm:vmw_user_bo_synccpu_ioctl [vmwgfx]] *ERROR* Failed synccpu r
elease on handle 0x000001af.
[ 151.224289] [drm:vmw_user_bo_synccpu_ioctl [vmwgfx]] *ERROR* Failed synccpu r
elease on handle 0x000001af.
[ 155.107395] perf: interrupt took too long (2555 > 2500), lowering kernel.perf
_event_max_sample_rate to 78250
[ 162.704378] [drm:vmw_user_bo_synccpu_ioctl [vmwgfx]] *ERROR* Failed synccpu r
elease on handle 0x000001af.
[ 164.229192] [drm:vmw_user_bo_synccpu_ioctl [vmwgfx]] *ERROR* Failed synccpu r
elease on handle 0x000002d6.
```

3.6.1.5. ./read 或 ./write 打开失败

需要 root 权限

3.6.2. 实验收获

学习使用了 linux 的内核模块，掌握了 makefile 文件的编写，深入了解了 linux 系统调用篡改，实现了字符设备驱动程序。

3.6.3. 意见与建议

参考资料中的代码版本过早，在实际使用时有不少错误，实际去网上查找解决这些问题的时间长过实验的时间。

3.7. 附件

3.7.1. 附件 1 程序

```
mymodules.c
#include <linux/init.h> /*必须要包含的头文件*/
#include <linux/kernel.h>
#include <linux/module.h> /*必须要包含的头文件*/
static int mymodule_init(void) // 模块初始化函数
{
    printk("hello, my module wored! \n"); /*输出信息到内核日志*/
    return 0;
}
static void mymodule_exit(void) // 模块清理函数
{
    printk("goodbye, unloading my module.\n"); /*输出信息到内核日志*/
}
module_init(mymodule_init); // 注册初始化函数
module_exit(mymodule_exit); // 注册清理函数
```

```

MODULE_LICENSE("GPL");      // 模块许可声明

mymodules.c 对应的 Makefile
ifneq ($(KERNELRELEASE),)
obj-m := mymodules.o
else
KERNELDIR := /lib/modules/$(shell uname -r)/build
PWD := $(shell pwd)
modules:
    $(MAKE) -C $(KERNELDIR) M=$(PWD) modules
endif

modify_syscall.c
#include <linux/init.h>
#include <linux/kernel.h>
#include <linux/module.h>
#include <linux/kallsyms.h>
#include <linux/unistd.h>
#include <linux/kprobes.h>
//original, syscall 96 function: gettimeofday
// new syscall 96 function: print "No 96 syscall has changed to hello" and return
a+b
#define sys_No 96

unsigned long old_sys_call_func;
//unsigned long p_sys_call_table; // find in /boot/System.map-'uname -r'
unsigned long orig_cr0;

int noop_pre(struct kprobe *p, struct pt_regs *regs) { return 0; }

static struct kprobe kp = {
    .symbol_name = "kallsyms_lookup_name",
};

unsigned long *p_sys_call_table = NULL;

// 调用 kprobe 找到 kallsyms_lookup_name 的地址位置
int find_kallsyms_lookup_name(void)
{
    int ret = -1;
    kp.pre_handler = noop_pre;
    ret = register_kprobe(&kp);
    if (ret < 0) {
        printk(KERN_INFO "register_kprobe failed, error:%d\n", ret);
        return ret;
    }
    printk(KERN_INFO "kallsyms_lookup_name addr: %p\n", kp.addr);
    p_sys_call_table = (unsigned long*)kp.addr;
    unregister_kprobe(&kp);
    return ret;
}

static int clear_cr0(void)

```

```

{
    unsigned long cr0 = 0;
    unsigned int ret;
    asm volatile ("movq %%cr0, %%rax" : "=a"(cr0));
    ret = cr0;
    cr0 &= 0xffffefff;
    asm volatile ("movq %%rax, %%cr0" :: "a"(cr0));
    return ret;
}

static void setback_cr0(int val)//recover the value of WP
{
    asm volatile ("movq %%rax, %%cr0" :: "a"(val));
}

asmlinkage int hello(int a,int b,int c) //new function
{
    printk("No 96  syscall has changed to hello");
    return a+b+c;
}

void modify_syscall(void)
{
    unsigned long *sys_call_addr;
    orig_cr0 = clear_cr0();
    sys_call_addr=(unsigned long *) (p_sys_call_table[sys_No]);
    old_sys_call_func=*(sys_call_addr);
    *(sys_call_addr)=(unsigned long)&hello;        // point to new function
    setback_cr0(orig_cr0);
}

void restore_syscall(void)
{
    unsigned long *sys_call_addr;
    orig_cr0 = clear_cr0();
    sys_call_addr=(unsigned long *) (p_sys_call_table[sys_No]);
    *(sys_call_addr)=old_sys_call_func;            // point to original
function
    setback_cr0(orig_cr0);
}

static int mymodule_init(void)
{
    modify_syscall();
    return 0;
}

static void mymodule_exit(void)
{
    restore_syscall();
}
module_init(mymodule_init);
module_exit(mymodule_exit);

```

```
MODULE_LICENSE("GPL");
```

modify_syscall.c 对应的 Makefile

```
ifneq ($(KERNELRELEASE),)
obj-m := modify_syscall.o
else
KERNELDIR := /lib/modules/$(shell uname -r)/build
PWD := $(shell pwd)
modules:
$(MAKE) -C $(KERNELDIR) M=$(PWD) modules
endif
clean:
$(MAKE) -C $(KERNELDIR) M=$(PWD) clean
```

modify_new_syscall.c

```
#include<stdio.h>
#include<sys/time.h>
#include<unistd.h>
int main()
{
    int ret=syscall(96,10,20,30); //after modify syscall 78
    printf("%d\n",ret);
    return 0;
}
```

modify_old_syscall.c

```
#include<stdio.h>
#include<sys/time.h>
#include<unistd.h>
int main()
{
    struct timeval tv;
    syscall(96,&tv,NULL); //before modify syscall 78 :gettimeofday
    printf("tv_sec:%ld\n",tv.tv_sec);
    printf("tv_usec:%ld\n",tv.tv_usec);
    return 0;
}
```

globalvar.c

```
#include<linux/module.h>
#include<linux/init.h>
#include<linux/fs.h>
#include<linux/uaccess.h>
#include<linux/wait.h>
#include<linux/semaphore.h>
#include<linux/sched.h>
#include<linux/cdev.h>
#include<linux/types.h>
#include<linux/kdev_t.h>
#include<linux/device.h>
#define MAXNUM 100
#define MAJOR_NUM 456 //主设备号，没有被使用
//设备的结构体
```

```

struct Scull_Dev
{
    struct cdev devm; //字符设备
    struct semaphore sem; //信号量, 实现读写时的 PV 操作
    wait_queue_head_t outq; //等待队列, 实现阻塞操作
    int flag; //阻塞唤醒标志
    char buffer[MAXNUM+1]; //字符缓冲区
    char *rd,*wr,*end; //读, 写, 尾指针
};

struct Scull_Dev globalvar;
static struct class *my_class;
int major=MAJOR_NUM;
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 int globalvar_open(struct inode *inode,struct file *filp);
static int globalvar_release(struct inode *inode,struct file *filp);

struct file_operations globalvar_fops =
{
    .read=globalvar_read,
    .write=globalvar_write,
    .open=globalvar_open,
    .release=globalvar_release,
};

static int globalvar_init(void)
{
    int result = 0;
    int err = 0;
    dev_t dev = MKDEV(major, 0);
    if(major)
    {
        result = register_chrdev_region(dev, 1, "charmем");
    }
    else
    {
        result = alloc_chrdev_region(&dev, 0, 1, "charmем");
        major = MAJOR(dev);
    }
    if(result < 0)
        return result;
    cdev_init(&globalvar.devм, &globalvar_fops);
    globalvar.devм.owner = THIS_MODULE;
    err = cdev_add(&globalvar.devм, dev, 1);
    if(err)
        printk(KERN_INFO "Error %d adding char_мем device", err);
    else
    {
        printk("globalvar register success\n");
        sema_init(&globalvar.sem,1); //初始化信号量
        init_waitqueue_head(&globalvar.outq); //初始化等待队列
        globalvar.rd = globalvar.buffer; //读指针
        globalvar.wr = globalvar.buffer; //写指针
    }
}

```

```

        globalvar.end = globalvar.buffer + MAXNUM;//缓冲区尾指针
        globalvar.flag = 0; // 阻塞唤醒标志置 0
    }
    my_class = class_create(THIS_MODULE, "chardev0");
    device_create(my_class, NULL, dev, NULL, "chardev0");
    return 0;
}

static int globalvar_open(struct inode *inode, struct file *filp)
{
    try_module_get(THIS_MODULE); //模块计数加一
    printk("This chrdev is in open\n");
    return(0);
}

static int globalvar_release(struct inode *inode, struct file *filp)
{
    module_put(THIS_MODULE); //模块计数减一
    printk("This chrdev is in release\n");
    return(0);
}

static void globalvar_exit(void)
{
    device_destroy(my_class, MKDEV(major, 0));
    class_destroy(my_class);
    cdev_del(&globalvar.dev);
    unregister_chrdev_region(MKDEV(major, 0), 1); //注销设备
}

static ssize_t globalvar_read(struct file *filp, char *buf, size_t len, loff_t *off)
{
    if(wait_event_interruptible(globalvar.outq, globalvar.flag!=0)) //不可读时 阻塞
    读进程
    {
        return -ERESTARTSYS;
    }
    if(down_interruptible(&globalvar.sem)) //P 操作
    {
        return -ERESTARTSYS;
    }
    globalvar.flag = 0;
    printk("into the read function\n");
    printk("the rd is %c\n", *globalvar.rd); //读指针
    if(globalvar.rd < globalvar.wr)
        len = min(len, (size_t)(globalvar.wr - globalvar.rd)); //更新读写长度
    else
        len = min(len, (size_t)(globalvar.end - globalvar.rd));
    printk("the len is %d\n", len);
    if(copy_to_user(buf, globalvar.rd, len))
    {
        printk(KERN_ALERT"copy failed\n");
        up(&globalvar.sem); //V 操作
        return -EFAULT;
    }
}

```



```

    }
    printk("the read buffer is %s\n", globalvar.buffer);
    globalvar.rd = globalvar.rd + len;
    if(globalvar.rd == globalvar.end)
        globalvar.rd = globalvar.buffer; //字符缓冲区循环
    up(&globalvar.sem); //V 操作
    return len;
}

static ssize_t globalvar_write(struct file *filp, const char *buf, size_t len, loff_t
*off)
{
    if(down_interruptible(&globalvar.sem)) //P 操作
    {
        return -ERESTARTSYS;
    }
    if(globalvar.rd <= globalvar.wr)
        len = min(len, (size_t)(globalvar.end - globalvar.wr));
    else
        len = min(len, (size_t)(globalvar.rd - globalvar.wr - 1));
    printk("the write len is %d\n", len);
    if(copy_from_user(globalvar.wr, buf, len))
    {
        up(&globalvar.sem); //V 操作
        return -EFAULT;
    }
    printk("the write buffer is %s\n", globalvar.buffer);
    printk("the len of buffer is %d\n", strlen(globalvar.buffer));
    globalvar.wr = globalvar.wr + len;
    if(globalvar.wr == globalvar.end)
        globalvar.wr = globalvar.buffer; //循环
    up(&globalvar.sem); //V 操作
    globalvar.flag=1; //条件成立，可以唤醒读进程
    wake_up_interruptible(&globalvar.outq); //唤醒读进程
    return len;
}

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

globalvar.c 对应的 Makefile
ifneq ($(KERNELRELEASE),)
obj-m := globalvar.o
else
KERNELDIR := /lib/modules/$(shell uname -r)/build
PWD := $(shell pwd)
modules:
    $(MAKE) -C $(KERNELDIR) M=$(PWD) modules
endif
clean:
    $(MAKE) -C $(KERNELDIR) M=$(PWD) clean

```

```
read.c
#include<sys/types.h>
#include<sys/stat.h>
#include<stdio.h>
#include<fcntl.h>
#include<string.h>
main(void)
{
    int fd,i;
    char num[101];
    //打开文件
    fd = open("/dev/chardev0",O_RDWR,S_IRUSR|S_IWUSR);
    if(fd!=-1)
    {
        while(1)
        {
            for(i=0;i<101;i++)
                num[i]='\0';
            read(fd,num,100);
            printf("%s\n",num);
            if(strcmp(num,"quit")==0)
            {
                close(fd);
                break;
            }
        }
    }
    else
    {
        printf("device open failure,%d\n",fd);
    }
}
```

```
write.c
#include<sys/types.h>
#include<sys/stat.h>
#include<stdio.h>
#include<fcntl.h>
#include<string.h>
main()
{
    int fd;
    char num[100];

    fd = open("/dev/chardev0",O_RDWR,S_IRUSR|S_IWUSR);
    if(fd!=-1)
    {
        while(1)
        {
            printf("Please input the globalvar:\n");
            scanf("%s",num);
            write(fd,num,strlen(num));
            if(strcmp(num,"quit")==0)
                break;
        }
    }
}
```

```

        {
            close(fd);
            break;
        }
    }
}
else
{
    printf("device open failure\n");
}
}

```

read_write.c

```

#include<sys/types.h>
#include<unistd.h>
#include<sys/stat.h>
#include<stdio.h>
#include<fcntl.h>
#include<string.h>
#include<stdlib.h>
int fd,i;
char msg[101];
int main()
{

    printf("本程序可以对设备进行读写，");
    fd = open("/dev/chardev0",O_RDWR,S_IRUSR|S_IWUSR);
    int operation;
    while(1)
    {
        printf("请选择您的操作：按 1 读取信息，按 2 写入信息，按 3 退出。\\n");
        scanf("%d",&operation);
        getchar();
        switch (operation)
        {
            case 1:
            {
                printf("您选择了：读取信息。\\n");
                if(fd!=-1)
                {
                    {
                        for(i=0;i<101;i++) //初始化
                            msg[i]='\0';
                        printf("读取的信息是：");
                        read(fd,msg,100);
                        printf("%s\\n",msg);
                    }
                }
            }
            else
            {
                printf("设备打开失败！ %d\\n",fd);
                exit(1);
            }
        }
    }
}

```

```

        continue;
    }
    case 2:
    {
        printf("您选择了：写入信息。 \n");
        if(fd!=-1)
        {
            printf("请输入要写入的信息:\n");
            scanf("%s",msg);
            write(fd,msg,strlen(msg));
            printf("您写入了%s。 \n",msg);

        }
        else
        {
            printf("设备打开失败! \n");
            exit(1);
        }
        continue;
    }
    case 3:
    {
        printf("再见! \n");
        return 0;
    }
    default:
    {
        printf("不支持该操作! \n");
        continue;
    }
}
}
}

```

3.7.2. 附件 2 Readme

实验三 动态模块系统编程与字符设备驱动

1. 动态模块

代码参考 mymodules/mymoules.c 和 Makefile

在动态模块中，在模块加载成功时，在日志中输出“hello,my module worded!”，在模块卸载时在日志中输出“goodbye,unloading my module.”

使用 make 命令编译模块，用 inmod 命令加载模块，用 rmmod 命令卸载模块，用 demsg 命令查看日志。

![make 运行结果](./make 运行结果.png)

make 运行结果

![加载模块成功](./加载模块成功.png)

加载模块成功

![模块加载后的日志记录](./模块加载后的日志记录.png)

模块加载后的日志记录

![模块卸载后的日志记录](./模块卸载后的日志记录.png)

模块卸载后的日志记录

2. 系统调用篡改

代码参考 `modify_syscall/modify_syscall.c`, `Makefile`, `modify_new_syscall.c` 和 `modify_old_syscall.c`

尝试篡改系统调用 `gettimeofday`, 使其调用改为输出三个参数之和。`modify_new_syscall` 用于测试篡改后的系统调用, `modify_old_syscall` 用于测试篡改前的系统调用。

3. 字符设备驱动程序

代码参考 `globalvar/globalvar.c`, `Makefile`, `read.c`, `write.c` 和 `read_write.c`

编写一个简单的字符设备驱动程序, 以内核空间模拟字符设备, 完成对该设备的打开, 读写和释放操作, 并编写聊天程序实现对该设备的同步和互斥操作。

![在两个终端聊天](./在两个终端聊天.png)

在两个终端聊天

4. 错误

make 命令错误

![make 命令错误](./make 命令错误.png)

报错信息说找不到 `Makefile` 文件, 考虑文件名大小写敏感, 修改 `makefile` 文件名为 `Makefile` 后解决。

加载模块错误

![加载模块错误](./加载模块错误.png)

权限不足, 把命令 `insmod` 改为 `sudo insmod`

![加载模块成功](./加载模块成功.png)

输出时间错误

![输出时间错误](./输出时间错误.png)

其类型为 `long int`, 将 `%d` 改为 `%ld`。

系统篡改错误

直接使用 ppt 中系统篡改代码, 在模块加载后, 运行 `modify_new_syscall`, 得到返回值为 -1, 运行 `modify_old_syscall`, 得到返回值仍为原先的 `gettimeofday` 的结果。在尝试卸载模块时, 发生错误, 提示模块正在被使用, 在查找资料以后, 尝试强制卸载模块并重启虚拟机。后来在查找资料中发现错误, 在 `modify_syscall.c` 中, 用 `unsigned long` `p_sys_call_table=0xc0361860`; 来表示系统调用表的起始地址, 但在我的虚拟机上的系统调用表的起始地址不同, 如图应为 `0xffffffff8aa00300`。而且实例代码中 `gettimeofday` 的系统调用号为 78, 但在实际系统调用表中, 32 位系统对应的 `unistd_32.h` 中为系统调用号为 78, 而 64 位系统对应的 `unistd_64.h` 中系统调用号为 96。

![unistd_32](./unistd_32.png)

`unistd_32.h` 中系统调用号

![unistd_64](./unistd_64.png)

`unistd_64.h` 中系统调用号

![查找系统调用表的起始地址](./查找系统调用表的起始地址.png)

查找系统调用表的起始地址

但在多次实验中发现，此法获得的系统调用表的地址是动态变化的。

在查询资料后得知使用 `kallsyms_lookup_name()` 函数动态获取地址。

但产生了如下报错

`![kallsyms_lookup_name undefined](./kallsyms_lookup_name%20undefined.png)`

`kallsyms_lookup_name undefined`

查询资料后得知

``https://blog.csdn.net/Linux_Everything/article/details/104912356``

``https://blog.csdn.net/Mr0cheng/article/details/104890981``

out-of-tree 的驱动不能够利用 `kallsyms_lookup_name` 函数调用未导出的内核函数。Will Deacon 提交了一组 patch set，移除了对 `kallsyms_lookup_name()` 和 `kallsyms_on_each_symbol()` 的 export 声明。

固转尝试使用 Kprobe 方法

```
```c
```

```
int noop_pre(struct kprobe *p, struct pt_regs *regs) { return 0; }
```

```
static struct kprobe kp = {
 .symbol_name = "kallsyms_lookup_name",
};
```

```
unsigned long *p_sys_call_table = NULL;
```

```
// 调用 kprobe 找到 kallsyms_lookup_name 的地址位置
```

```
int find_kallsyms_lookup_name(void)
```

```
{
 int ret = -1;
 kp.pre_handler = noop_pre;
 ret = register_kprobe(&kp);
 if (ret < 0) {
 printk(KERN_INFO "register_kprobe failed, error:%d\n", ret);
 return ret;
 }
 printk(KERN_INFO "kallsyms_lookup_name addr: %p\n", kp.addr);
 p_sys_call_table = (unsigned long*)kp.addr;
 unregister_kprobe(&kp);
 return ret;
}
```
```

但仍未能产生正确结果，这是系统日志文件输出的错误

`![系统日志文件输出错误](./系统日志文件输出错误.png)`

`./read` 或 `./write` 打开失败

需要 root 权限