

**操作系统7-12周作业报告**

**2022-2023学年 第一学期**

学生姓名： 徐丁

学 号： 202205010419

专业班级： 计算机224班

所在学院： 数学与计算机科学学院

目录

[理论课作业 3](#_Toc186202215)

[理论课作业第1回： 3](#_Toc186202216)

[理论课作业第2回： 5](#_Toc186202217)

[理论课作业第3回： 6](#_Toc186202218)

[理论课作业第4回： 10](#_Toc186202219)

[理论课作业第5回： 12](#_Toc186202220)

[理论课作业第6回： 13](#_Toc186202221)

[Peterson 正确性证明 14](#_Toc186202222)

[理论课作业第7回： 15](#_Toc186202223)

[理论课作业第8回： 16](#_Toc186202224)

[实验课作业 18](#_Toc186202225)

[实验课作业第1回： 18](#_Toc186202226)

[实验课作业第2回： 19](#_Toc186202227)

[实验课作业第3回： 30](#_Toc186202229)

[实验课作业第4回： 48](#_Toc186202230)

[实验课作业第5回： 50](#_Toc186202231)

[实验课作业第6回： 54](#_Toc186202232)

[实验课作业第7回： 57](#_Toc186202233)

[实验课作业第8回： 61](#_Toc186202234)

[实验课作业第9回： 64](#_Toc186202235)

[实验课作业第10回： 70](#_Toc186202236)

[实验课作业第11回： 78](#_Toc186202237)

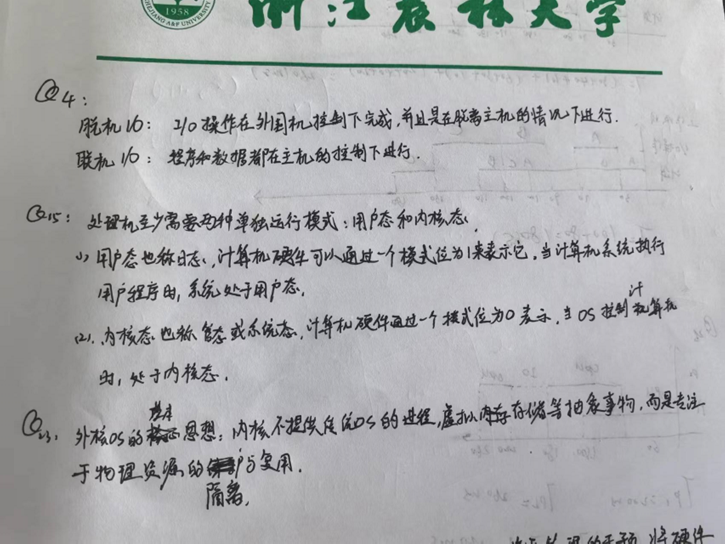
[实验课作业第12回： 84](#_Toc186202238)

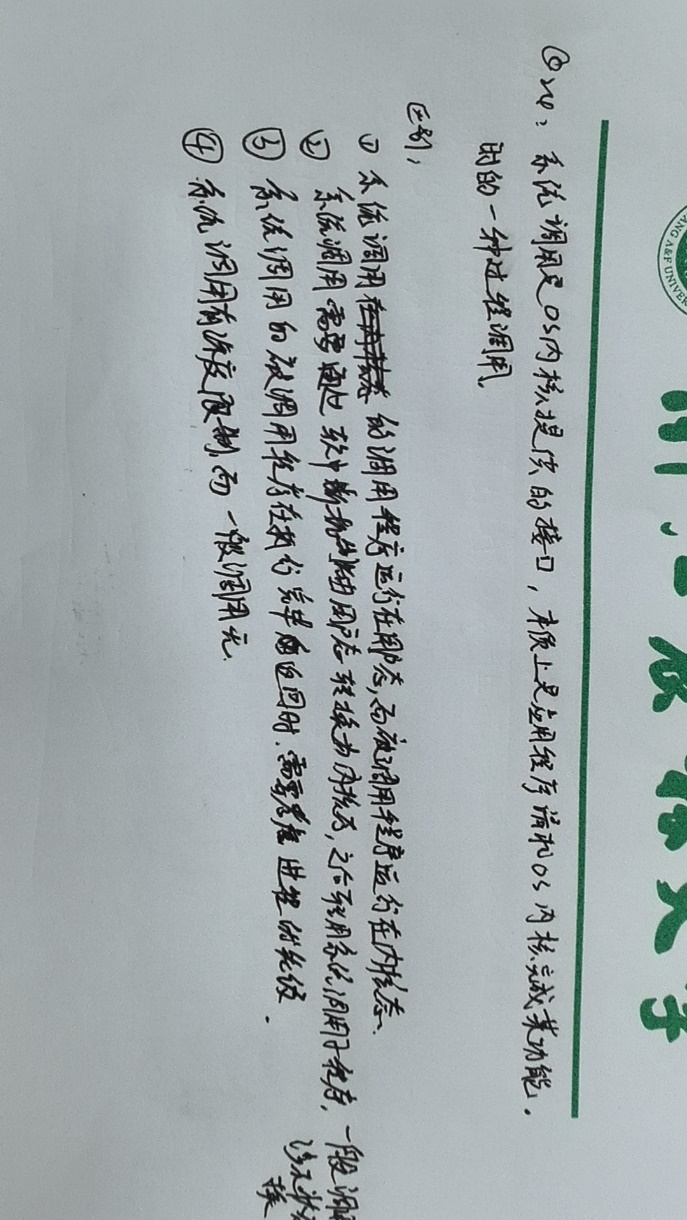
[实验课作业第13回： 96](#_Toc186202239)

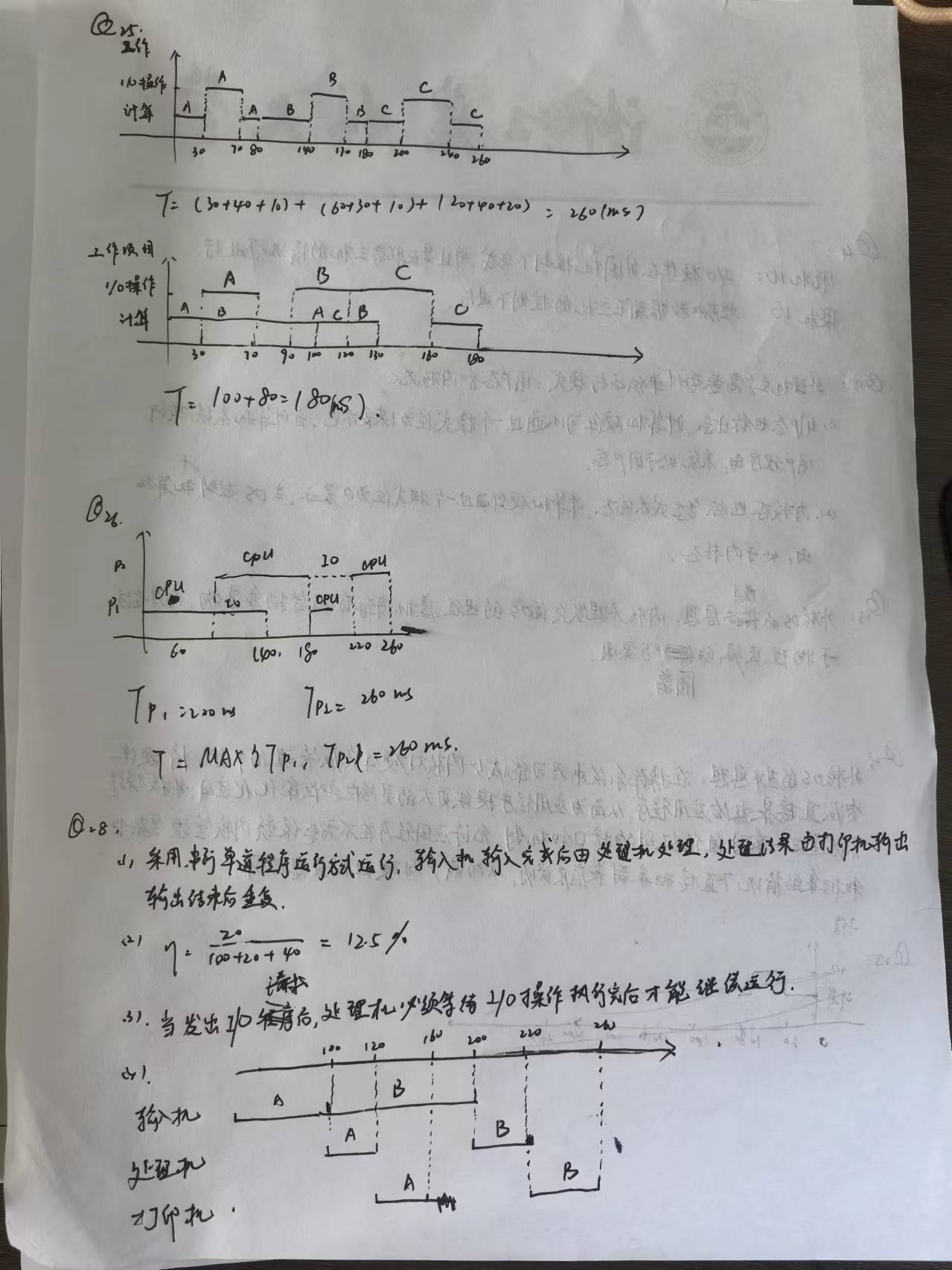
[随堂测试 111](#_Toc186202245)

# 理论课作业

## 理论课作业第1回：







## 理论课作业第2回：

**4.为什么要引入进程的挂起状态?**

(1)终端用户的需要。终端用户自己的程序在运行期间发现有可疑问题，希望暂停程序运行，以便用户研究其执行情况或对其进行修改。

(2)父进程的需要。有时父进程希望挂起自己的某个子进程，以便考查和修改该子进程或者协调各子进程间的活动。

(3)负荷调节的需要。当实时系统中的工作负荷较重，可能会影响到对实时任务的控制时，系统可把一些不重要的进程挂起，以保证自身能正常运行。

(4)OS的需要。OS有时希望挂起某些进程，以便检查在进程运行过程中资源的使用情况或进行记账。所记录的信息包括CPU时间、实际使用时间、作业或进程数量等。

**6.(考研真题)请给出 PCB 的主要内容。描述当进程状态发生转换(就绪→运行、运行→阻塞)时，OS需要使用/修改 PCB 的哪些内容?**

PCB ：用来存储进程的标志信息、现场信息和控制信息。

就绪->运行：将PCB当前的就绪改为运行态，修改PCB的队列指针，将PCB从就绪队列移出；利用PCB中的CPU现场信息布置CPU现场并投入运行

运行->阻塞：将CPU的当前状态保存到PCB中，将进程状态由运行改为就绪，并修改PCB中相应的队列指针，然后将其放入PCB就绪队列

**8.(考研真题)在创建一个进程时，OS需要完成的主要工作是什么?**

操作系统在检测到创建新进程的请求后，首先调用进程创建原语。接着，会申请一个空白的进程控制块（PCB），并在其中填写用于控制和管理该进程的信息。然后，为进程分配运行所需的资源。最后，将进程的PCB设置为就绪状态，并插入到就绪队列中。

**15.试比较直接通信方式和间接通信方式。**

1.发送原语和接收原语

2.直接通信要求发送双发显示提供对方的标识符；对于接收过程，如果允许它同时接收多个进程发来的消息，则接受原语中的发送进程表示符可以是通信完成后的返回值

3.直接通信时，进程只须提供对方的标识符便可进行通信，在收发双方之间建立通信链路的任务的系统自动完成，此外在收发双发之间有且仅有一条通信链路

间接通信时，仅当一堆进程共享某个信箱时它们之间才有通信与链路，每对进程间有多条链路

4.直接通信通常只能提供实时的通信，间接通信既可以实现实时通信，也可以实现非实时通信

**19.(考研真题)用户级线程和内核支持线程有何区别?**

1.用户级线程:

仅存在于用户空间中的线程,无须内核支持。这种线程的创建、撤销、线程间的同步与通信等功能, 都无需利用系统调用实现。用户级线程的切换通常发生在一个应用进程的诸多线程之间,同样无需内核支持。

2.内核支持线程:

在内核支持下运行的线程。无论是用户进程中的线程, 还是系统线程中的线程, 其创建、撤销和切换等都是依靠内核,在内核空间中实现的。 在内核空间里还为每个内核支持线程设置了线程控制块, 内核根据该控制块感知某线程的存在并实施控制。

**22.(考研真题)现代 OS 一般都提供多进程(或称多任务)运行环境，回答以下问题。**

**(1)为支持多进程的并发执行，系统必须建立哪些关于进程的数据结构?**

**(2)为支持进程状态的变迁，系统至少应提供哪些进程控制原语?**

**(3)在执行每一个进程控制原语时，进程状态会发生什么变化?相应的数据结构会发生什么变化?**

1）为支持多进程的并发执行，系统必须建立关于进程的相关数据结构，包括 PCB 和队列结构（如就绪队列、等待队列、运行指针等）。

2）为支持进程状态的变迁，系统应提供的进程控制原语包括：创建原语、阻塞原语、唤醒原语、撤销原语。

3）在执行每一个进程控制原语时，进程状态及相应的数据结构有 4 种变化情况。

1．创建原语：系统为进程创建 PCB ，并对它进行初始化。进程状态由无变为就绪状态，新创建的进程加人就绪队列中。

2. 阻塞原语：进程状态从运行状态变为阻塞状态，并将阻塞进程的 PCB 插人相应的阻塞队列中。

3. 唤醒原语：进程状态从阻塞状态变为就绪状态．从阻塞队列中删除该进程，并将其插入就绪队列中。

4. 撤销原语：进程状态从运行状态变为消亡状态，系统撤销该进程的PCB。

## 理论课作业第3回：

**1. 高级调度与低级调度的主要任务是什么?为什么要引入中级调度?**

①高级调度的对象是作业。它的主要任务是根据某种算法，决定将外存上处于后备队列中的哪几个作业调入内存，为它们创建进程、分配必要的资源，并将它们放入就绪队列。

②低级调度的对象是进程(或内核级线程)。它的主要任务是根据某种算法，决定就绪队列中的哪个进程获得处理机，并由分派程序将处理机分配给被选中的进程。

引人中级调度的主要目的是提高内存利用率和系统吞吐量。为此，应把那些暂时不能运行的进程调至外存等待，把进程状态改为就绪驻外存状态或挂起状态。当它们已具备运行条件目内存又有空间时，由中级调度来决定把外存上的那些已具备运行条件的就绪进程再重新调人内存，并修改它们的状态为就绪状态，挂在就绪队列上等待。

**11.在基于时间片的 RR(round robin,轮转)调度算法中,应如何确定时间片的大小?**

响应时间：时间片越短，系统的响应速度越快，因为每个进程在等待其下次执行的时间会更短。这在实时系统或交互性要求高的系统中尤为重要。但过短的时间片会导致频繁的进程切换，增加系统开销。

系统开销：每次上下文切换都会消耗 CPU 资源。若时间片过短，系统频繁切换进程，导致开销大增。相反，时间片过长，则 RR 调度算法趋近于 FCFS（先来先服务）调度算法，失去公平性。

任务类型和数量：在负载较高（进程数量多）的系统中，通常适合选择较短的时间片，以保证每个进程都能得到相对公平的执行机会；而在负载较轻（进程数量少）或任务执行时间较长的情况下，可以适当增大时间片。

**17.(考研真题)什么是死锁?产生死锁的原因和必要条件是什么?如何预防死锁?**

死锁是指两个或多个进程互相等待对方释放资源，从而导致无法继续执行的状态。要发生死锁，必须满足四个必要条件：

1. **互斥条件**：至少有一个资源不能被共享，只能被一个进程占用。
2. **持有并等待**：至少有一个进程持有一个资源，同时又在等待其他资源。
3. **不可抢占**：已获得的资源不能被抢占，只能由持有的进程主动释放。
4. **循环等待**：存在一个进程链，使得每个进程都在等待链中下一个进程所占用的资源。

预防死锁的方法有：

* **资源预分配**：在进程执行前分配所需的全部资源。
* **破坏必要条件**：如限制资源共享，避免持有并等待，允许资源剥夺，或打破循环等待。
* **银行家算法**：通过动态检查进程的资源申请，确保系统始终处于安全状态。

****

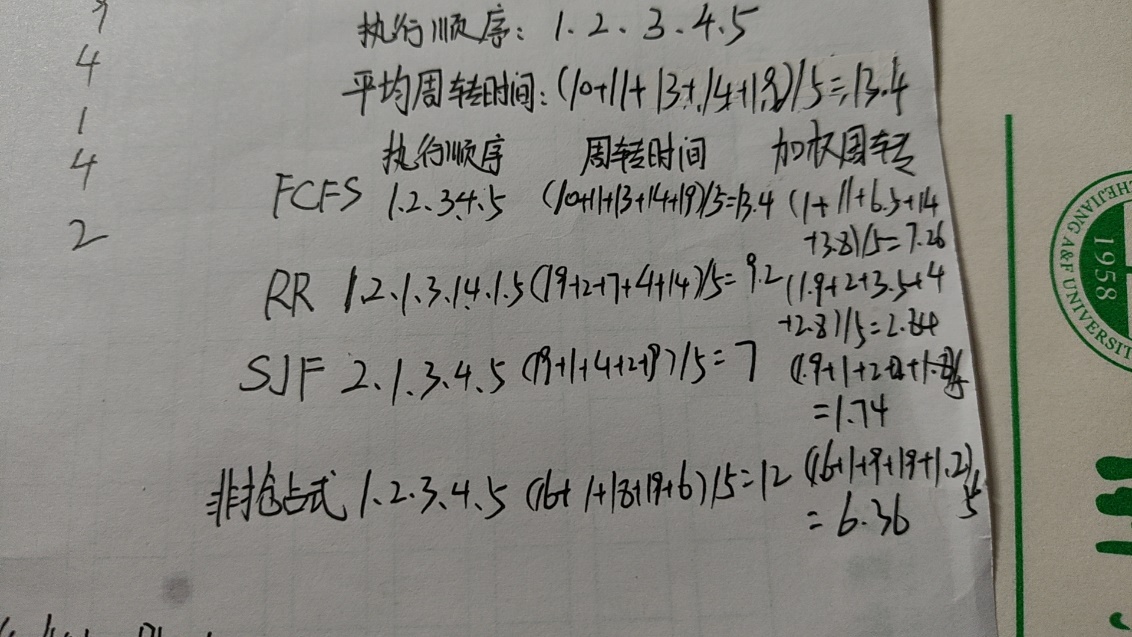
P1：9 P2：15.6 P3：9 P4：14.5 P5：5

（9+15.6+9+14.5+5）/ 5=10.62

**20.(考研真题)假定要在一台处理机上执行表1-3-2所示的作业，且假定这些作业在时刻0以1，2，3，4，5的顺序到达。请说明分别采用FCFS、RR(时间片为1)、SJF及非抢占式优先级调度算法时，这些作业的执行情况(优先级的高低顺序依次为1到5)。针对上述每种调度算法，给出平均周转时间和平均带权周转时间。**

**表格

描述已自动生成**



FCFS： 12345

周转时间：（10+11+13+14+19）/ 5 = 13.4 加权周转：（1+11+6.5+14+3.8）/ 5 = 7.26

RR：12131415

周转时间：（19+2+7+4+14）/ 5 = 9.2 加权周转：(1.9+2+3.5+4+2.8)/5=2.64

SJF：21345

周转时间：(19+1+4+2+9) / 5 = 7 加权周转：(1.9+1+2+2+1.2)/5=1.74

非抢占式：12345

周转时间：（16+1+18+19+6）/ 5 = 12 加权周转：(16+1+9+19+1.2)/5=6.36

**图片包含 图形用户界面

描述已自动生成**

各类进程之间采用优先级调度算法，而同类进程部采用时间片轮转调度算法，因此，系统首先对优先级为4的进程P1、P2、P3采用时间片轮转调度算法运行；当P1、P2、P3均运行结束或没有可运行的进程,即P1、P2、P3都处于等待状态,或其中部分进程已运行结束，其余进程处于等待状态时，则对优先级为3的进程P4、P5采用时间片轮转调度算法运行。在此期间，如果未结束的P1、P2、P3有一个转为就绪状态，则当前时间片用完后又回到优先级4进行调度。类似地，当P1～P5均运行结束或没有可运行进程,即P1～P5都处于等待状态；或其中部分进程已运行结束，其余进程处于等待状态,时，则对优先级为2的进程P6、P7、P8采用时间片轮转调度算法运行，一旦P1～P5中有一个转为就绪状态，则当前时间片用完后立即回到相应的优先级进行时间片轮转调度。

**表格

描述已自动生成**

这段程序在不同的运行推进速度下，可能会产生死锁。例如，进程 p1 先申请资源 R 1，得到资源 R1，然后进程申请资源 R2 ，得到资源 R2 进程 p1又申请资源 R2 ，因资源 R2 己分配，使得进程 p1 阻塞。进程 p1 和进程 p2 都区申请不到资源而形成死锁。若改变进程的运行顺序，则这两个进程就不会出现死锁现象。

产生死锁的原因可归结为两点：

1）竞争资源

2）进程推进顺序非法

产生死锁的必要条件：

1）互斥条件

2）请求并保持条件

3）不可抢占条件

4）循环等待条件

## 理论课作业第4回：

习题四P134：

5. 忙等指的是进程要访问的临界资源被其他进程占用，不断地消耗CPU 时间 来检测某个条件是否成立，而不是将 CPU 控制权交给其他进程占用CPU资源

缺点：导致 CPU 资源浪费，系统响应性下降、以及可能的饥饿等问题

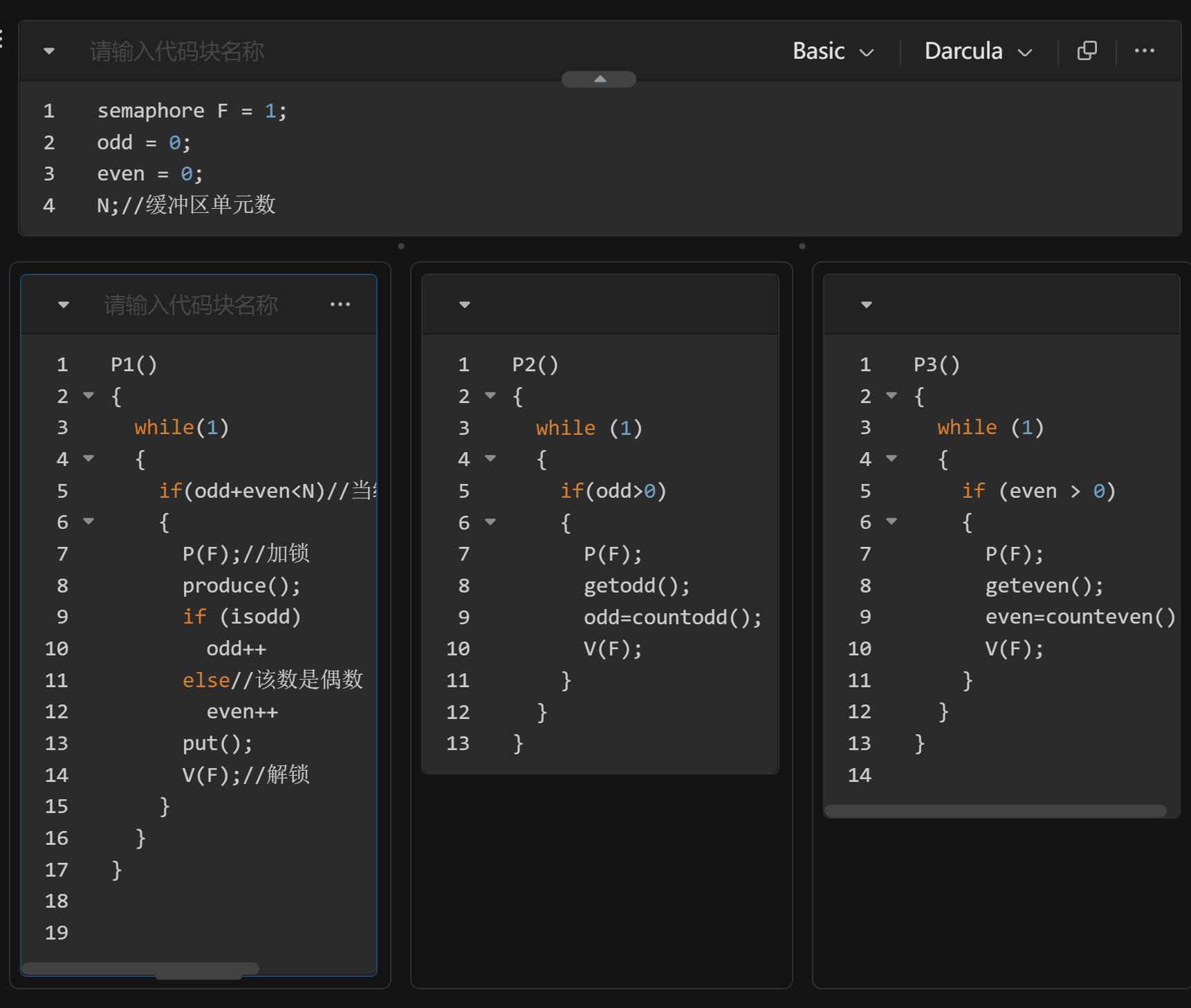
8．竞态条件、死锁或数据不一致

如果 P 和 V 操作不是原子性的，且信号量为1，可能发生进程 A 和进程 B 同时执行P操作，此时它们都看到 信号量= 1。这时，两个进程都会进入临界区，导致缓冲区的资源被多个进程同时访问，产生竞态条件。

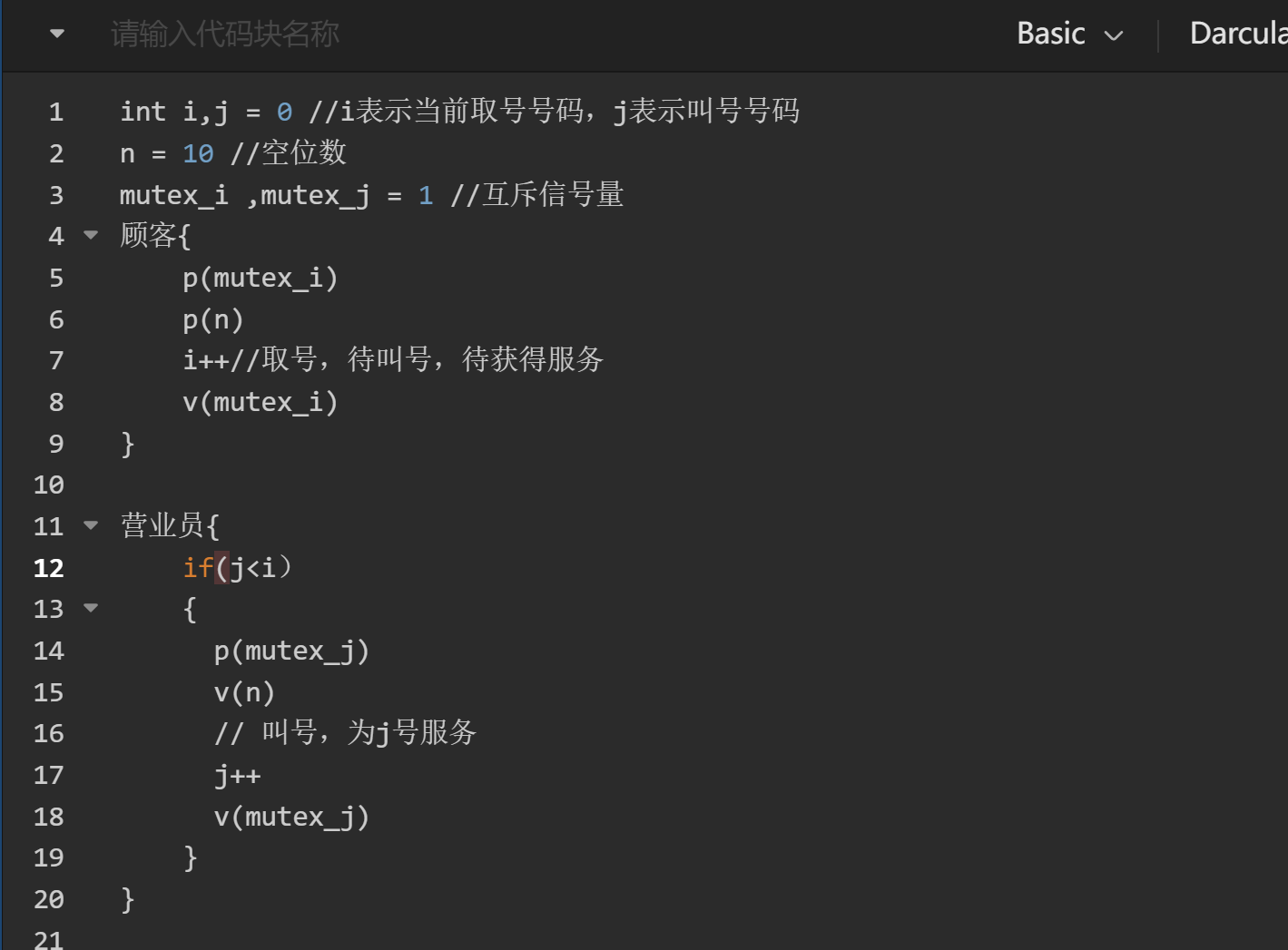
12．作用：为了解决管程中的进程不能被阻塞，而其他进程不能访问管程的问题。  
含义：管程为何种阻塞的原因都设置了条件变量，进程若因此原因而阻塞，会把这个进程挂在此条件变量的队列上，若是阻塞条件发生变换，则启动一个进程。

15.信号量初值为3。当第1个进程申请进入时，信号量值变为2；第2个进程申请进入时，信号量值变为1；第3个进程申请进时，信号量值变为0；第4个进程申请进入时，信号量值变为-1。因此，信号量的变化范围是[-1，3]

16.



17:



## 理论课作业第5回：

P170 习题五

3

将虚拟的逻辑地址映射为真实存在的物理地址。  
进程中的地址都是从0开始的虚拟地址，在多道程序环境中必须依靠重定位寄存器将逻辑地址映射为物理地址。

7

链接：将编译后的目标模块与库函数链接为一个可装入的模块。

解决问题：将目标模块与库函数链接起来，目标函数中只有调用的库函数名，参数等，并没有实际内容，链接后便形成完整的函数。

静态链接：已经拥有所有需要的库函数，运行速度快但体积大，有很多冗余代码。

装入时动态链接：装入内存时一边装入一边链接,若装入时发生调用，在将被调用的模块装入并修改地址。便于修改与更新，便于模块共享，但是运行时性能会有损失

动态运行时链接：体积小，装入速度快。运行时性能会有损失。

12

T= 0.85×1.1ns+0.15×2.1ns= 1.25ns

16

由题意可知，该系统的用户地址空间为2“B，而页的大小为4KB，故作业最多可有2“/2”2-2“个页，其页表的大小则为2”x4-2“B。因此，又可将页表分成2”个页表项，并为它建立二级页表，二级页表的大小为2“B。依次类推，可知道它的3、4、5、6级页表的长度分别是2-B、2“B、2'B、2B，故必须采取6层分页策略。

18



## 理论课作业第6回：

P197 习题六

13

(1)逻辑地址0A5CH转化为二进制为000101001011100，页号为00010，页号合法。从页表中找到对应的内存物理块号为4，即0100;与页内地址1001011100拼接而形成物理地址01001001011100，即125CH。

(2)逻辑地址103CH的页号为4，页号合法，但该页未装入内存，产生缺页中断。

(3)逻辑地址1A5CH的页号为6，页号非法，产生越界中断。

15

（1）80%的访问的页表项在联想寄存器中，访问耗时1us。

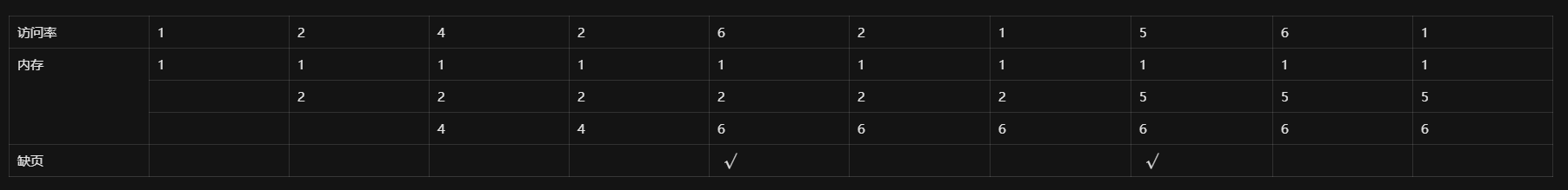
（2）18%的访问的页表项不在联想寄存器中，但在内存中，访问耗时(1ms+1ms)

（3）2%的访问产生缺页中断，访问耗时(1μs+1us+20ms+lus)。

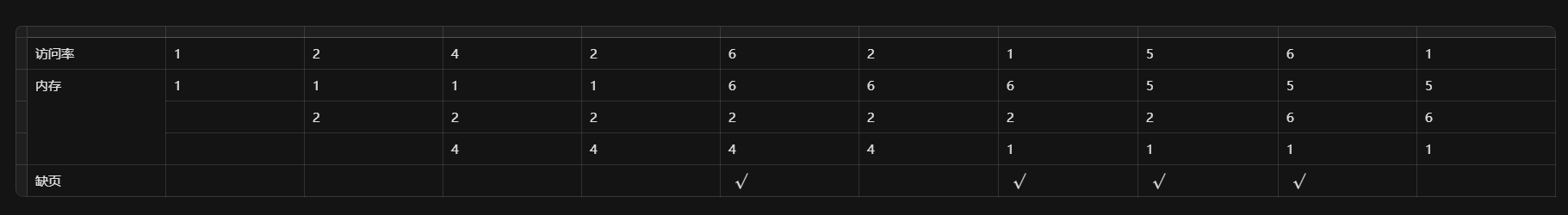
所以有效访问时间为80%x1+(1-80%)x[(1-10%)x1x2]+2%x(1x3+20x1000)=401.22us

18

F=2/10=20%

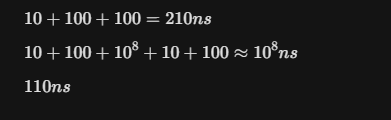


F=4/10=40%



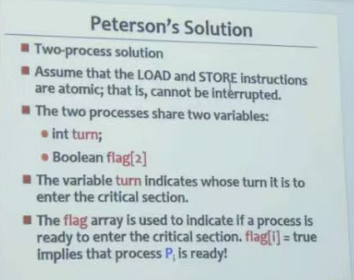
21

（1）



（2）

在访问1565H时，由于内存已满且合法驻留集为2，使用LRU页面置换算法淘汰0号页，将101H号块号分配给1号页，拼接成物理地址101565H。



## Peterson 正确性证明

需要证明:

1. 互斥成立。
2. 满足进入（progress即不死锁）需求
3. 满足限界等待需求

证明1：

Pi进程只有在flag[j] == false或者turn == i时候才会进入临界区。

如果两个进程同时在其临界区内执行，那么flag[0] = flag[1] = true .

但是turn的值只能是0或者1,所以P0和P1同一时间不能成功地执行它们的while语句。因此只能有一个进程（如Pj）能成功的执行完while语句，而另一个进程（Pi）则至少必须执行一个附加的语句("turn = j")。

而且，由于只要Pj在其临界区内，flag[j] = true和turn = j就会同时成立。

互斥成立

证明2：

只要flag[j] = true && turn = j成立，进程Pi陷入while循环语句.

如果Pj不准备进入临界区，那么flag[j] = false,Pi就可以进入临界区

如果Pj已经设置flag[j] = true,且也在其while语句中执行，那么turn = j 或者 turn = i.如果turn = i，那么Pi进入临界区；如果turn = j，那么Pj进入临界区。然而当Pj退出临界区，它会设置flag[j] = false,以允许Pi进入其临界区

如果Pj重新设置flag[j] = true,那么它也必须设置turn为i.

因此由于进程Pi执行while语句时并不改变turn的值，所以Pi会进入临界区（证明2成立），并且Pi最多在Pj进入临界区一次后就能进入（证明3成立）

## 理论课作业第7回：

P245

4

1. 通道是一种特殊的处理机，它具有执行I/O指令的能力，并且可以通过执行通道I/O程序来控制I/O操作。
2. 交叉连接主要是为了解决通道的瓶颈问题。之所以会产生瓶颈问题，是因为通道价格昂贵，设置的通道数量较少，导致系统吞吐量下降。交叉连接的多通路方式不仅解决了瓶颈问题，还提高了系统可靠性。

6

设备驱动程序与硬件密切相关，主要负责接收上层软件发来的I/O指令，并将其转换成具体要求发送给设备控制器；反之，也将来自设备控制器的信号传送给上层软件。采用设备驱动程序实现I/O系统的高层与设备控制器之间的通信，驱动I/O设备工作。

用户进程通过设备驱动程序控制设备工作的过程为：

1. 接收由用户进程发来的I/O命令和参数，并将命令中的抽象要求转换为具体要求，如将磁盘盘块号转换为磁盘的盘面、磁道和扇区号：
2. 检查用户1/0请求的合法性，如果请求不合法，则拒绝接收I/O0请求并反馈用户进程；
3. 了解I/O设备的状态，如果设备准备就绪，则可由设备控制器设置设备的工作方式、传递有关参数：否则，将请求者的请求块挂到设备请求队列上等待：
4. 发出I/O命令，如果设备空闲，则立即启动I/O设备，完成指定的I/O操作。

15

查询扇区的时间是3ms,传输时间4KB/20MB/s=0.2ms,总时间:3+6+0.2+0.2=9.4ms

17

（1）FCFS调度算法：访问顺序为30、145、120、78、82、140、20、42、165、55、65；移动距离为80、115、25、42、4、58、120、22、123、110.10：平均移动磁道数为（80+115+ 25+42+4+58+120+22+123+110+10)/11=64.45。

（2）SCAN调度算法：访问顺序为120、140、145、165、82、78、65、55、42、30、 20：移动距离为10、20、5、20、83、4、13、10、13、12、10；平均移动磁道数为（10+20+5+20+83+4+13+10+13+10+10)/11=18.18。

18

（1）FCFS调度算法：磁头移动顺序为30、38、6、37、100、14、124、65、67；移动磁道

数为8+32+31+63+86+10+59+2=391。

（2）SSTF调度算法：磁头移动顺序为30、37、38、14、6、65、67、100、124；移动磁道

数为7+1+24+8+59+2+33+24=158。

（3）SCAN调度算法：磁头移动顺序为30、37、38、65、67、100、124、14、6；移动磁道数为7+1+27+2+33+24+110+8=212。

## 理论课作业第8回：

P279

15

1

建立符号链接（软链接）时，引用计数值直接被复制，因此F的引I用计数

值为1。

2建立硬链接时，引用计数值加1，F的引1用计数值为2；但在删除文件时，对于硬链接，引用计数值减1，因此删除E后F的引I用值变为1。

16

若顺序文件中所含有的记录数为N，则检索到指定关键字的记录平均须查找 N/2个记录。但对于索引顺序文件，检索到指定关键字的记录平均须查找√N个记录，因而其检索效率比顺序文件提高了约√N/2倍。因此，包含40000条记录的文件，采用索引顺序文件组织方式的平均检索效率比顺序文件提高了√40000/2-100倍。

17

创建的文件数量上限等于索引节点数量上限，索引节点为4B，即为32位，故最多有2^n个索引节点，即最多能创建2^n个文件。

20

目录检索的大致过程为：①在根目录中查找usr目录项：②找到usr目录项后，根据其索引节点编号，获得索引节点位置并将其内容读人内存（若不在内存），然后根据索引节点中的文件外存地址读人usr的内容：③在usr中查找student目录项，获取索引节点并读人student的内容（过程类似于第②步，若还有更深层次的目录，则中间目录也采用类似的方法）：④在student中查找myproc.c目录项，获取其索引节点（过程类似于第②步）。

# 实验课作业

## 实验课作业第1回：

1. 实验内容：

安装Linux虚拟机系统（Ubuntu）

1. 实验目标：

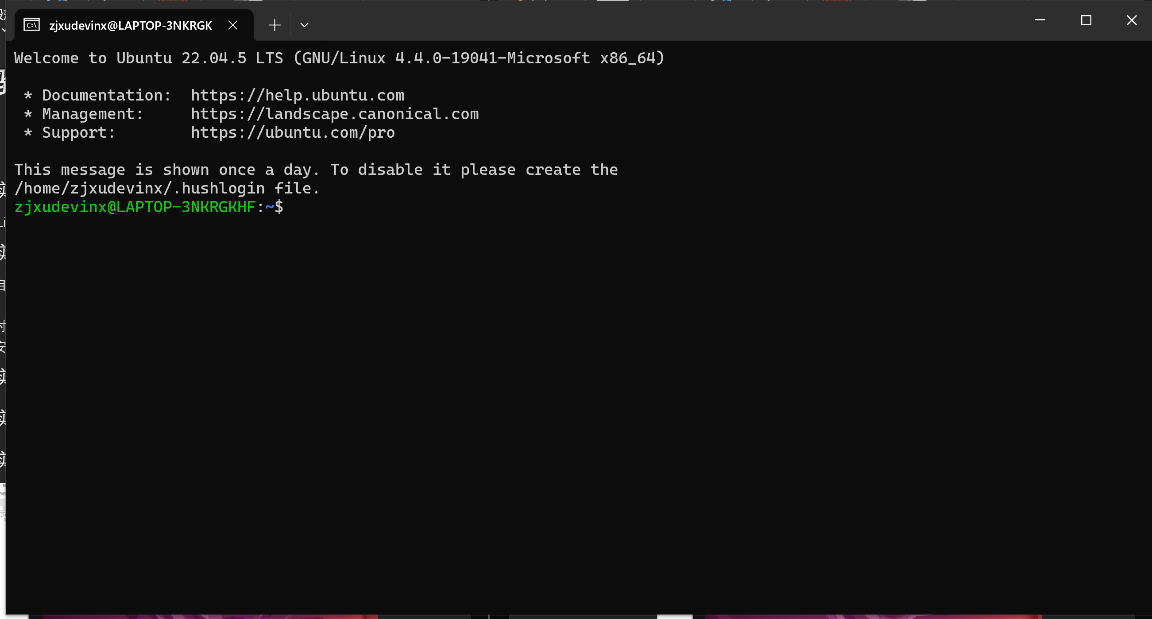
学会自己动手安装Linux虚拟机系统（Ubuntu）；

安装顺序：WSL

加深对开源操作系统的认识和体会；

通过安装过程提高自己动手调查问题、分析问题、解决问题的能力。

1. 实验要求：
2. 实验代码：
3. 实验结果



1. 心得体会

使用 WSL 安装 Ubuntu 后，我感受到其便捷性与高效性。安装过程十分简洁，通过简单的命令即可完成配置，避免了传统虚拟机的复杂设置。同时，WSL 能够与 Windows 文件系统无缝集成，方便在两个系统间快速切换与共享资源。在实际使用中，WSL 提供了完整的 Linux 操作体验，使我能够在熟悉的环境中运行常用的 Linux 命令与工具，而无需脱离 Windows 系统。无论是 Web 开发还是系统管理，Ubuntu 在 WSL 中的运行表现都十分稳定，为日常开发工作带来了极大的便利与效率提升。

## 实验课作业第2回：

## 实验目标

* 熟悉 Ubuntu 系统的基本操作与界面。
* 掌握常用的 Linux 命令行操作技巧。
* 了解文件系统的基础结构与常见管理操作。
* 学会使用基本文本编辑工具如 nano 和 vi。
* 熟悉软件安装和系统更新的流程。

**实验步骤**

**1. 创建与管理目录和文件**

1. **创建实验目录 os\_lab 并进入该目录：**

mkdir os\_lab

cd os\_lab



1. **创建子目录 test\_dir 并在其中创建文件 test\_file.txt：**

mkdir test\_dir

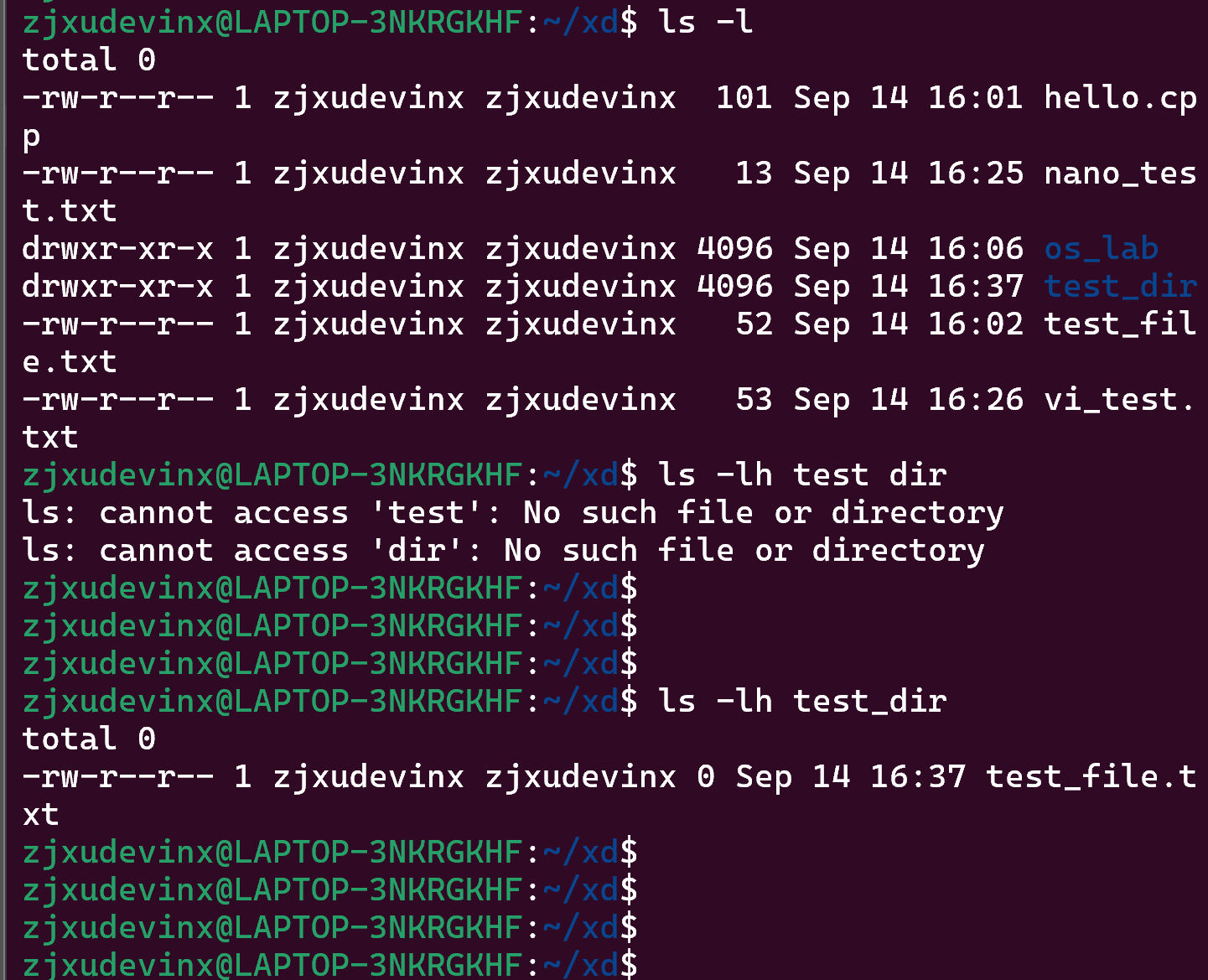
touch test\_dir/test\_file.txt



1. **查看文件和目录的详细信息：**

ls -l

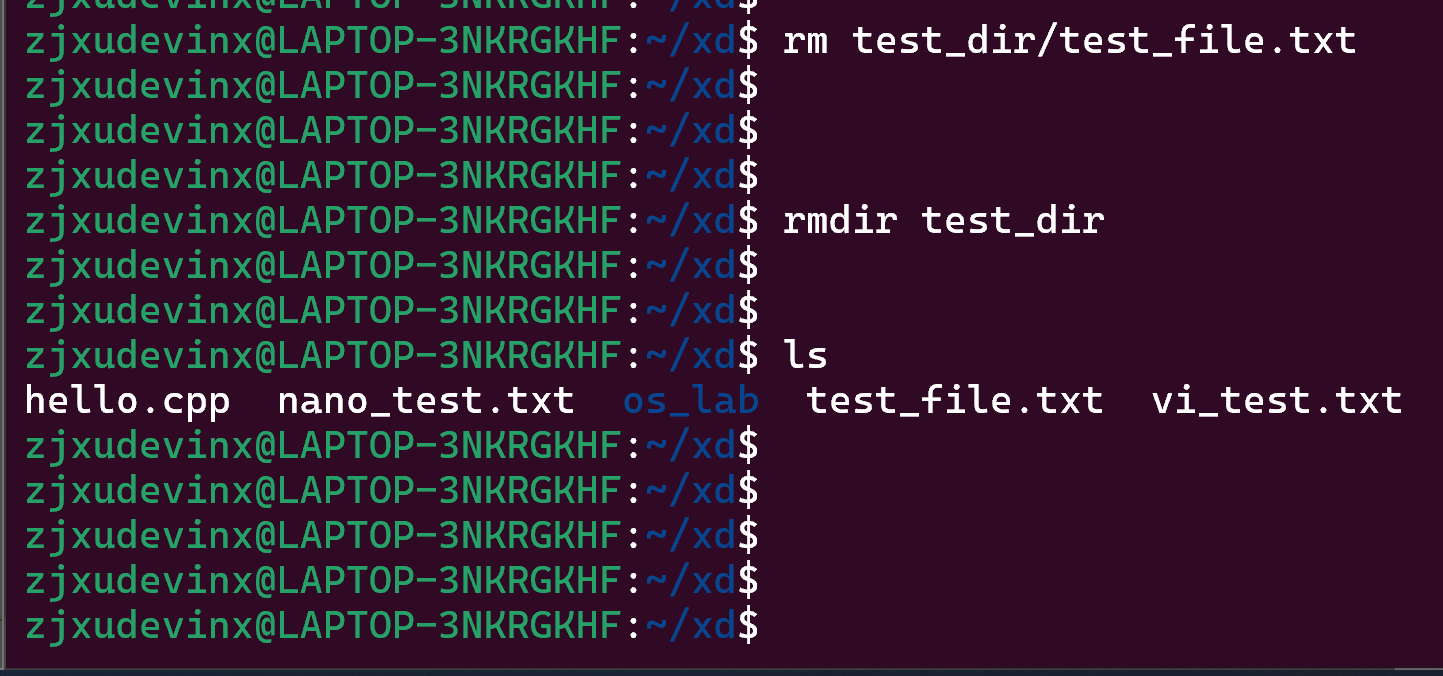
ls -lh test\_dir



1. **删除文件和目录：**

rm test\_dir/test\_file.txt

rmdir test\_dir



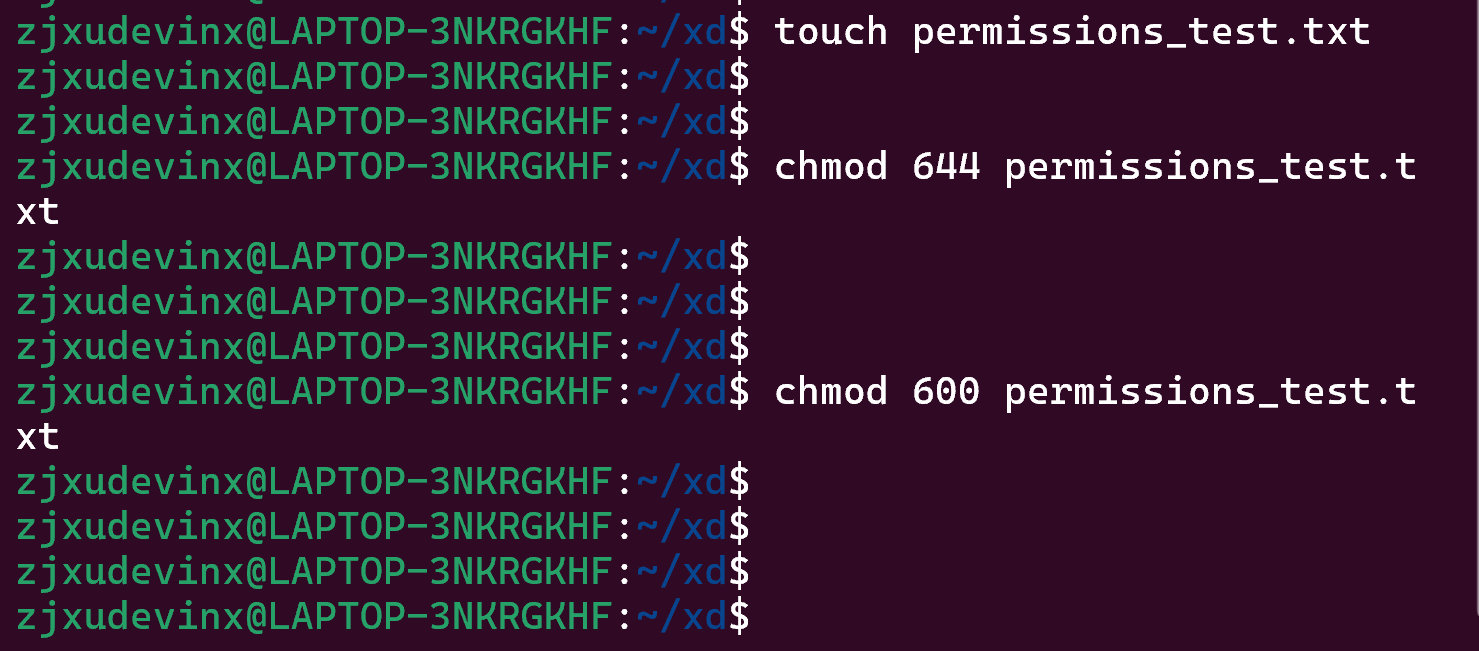
**2. 文件权限与用户管理**

1. **创建文件 permissions\_test.txt 并设置不同权限：**

touch permissions\_test.txt

chmod 644 permissions\_test.txt # 所有者读写，其他人只读

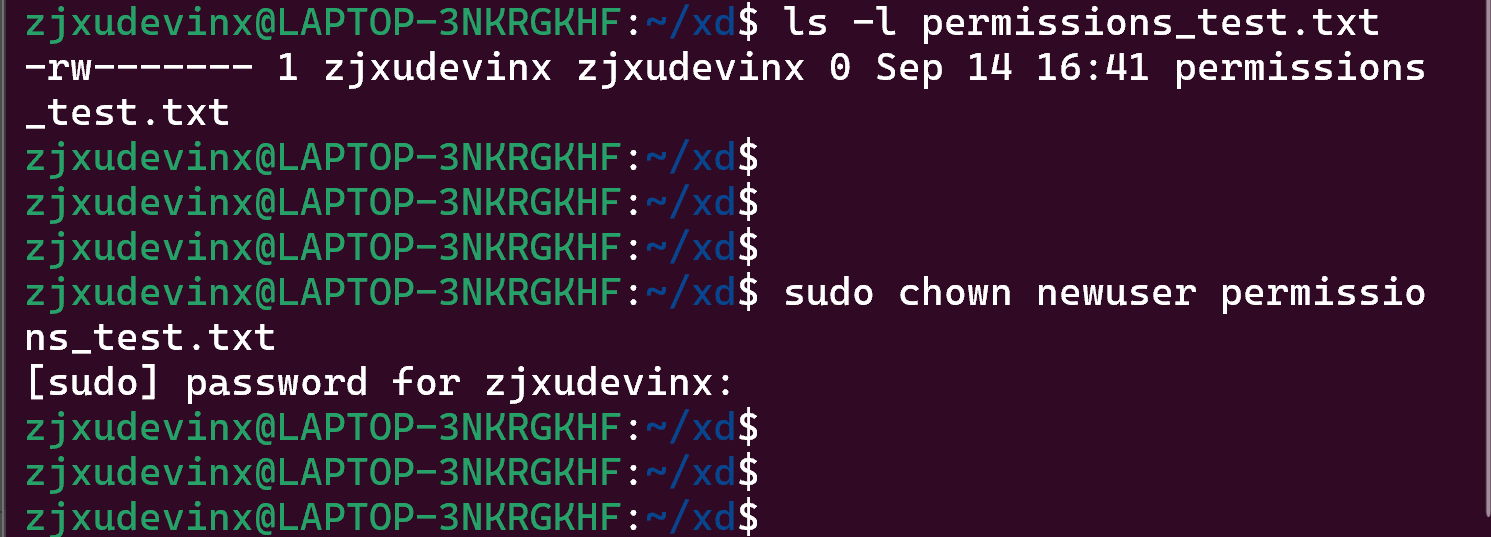
chmod 600 permissions\_test.txt # 只有所有者可读写



1. **查看文件权限并更改所有者：**

ls -l permissions\_test.txt

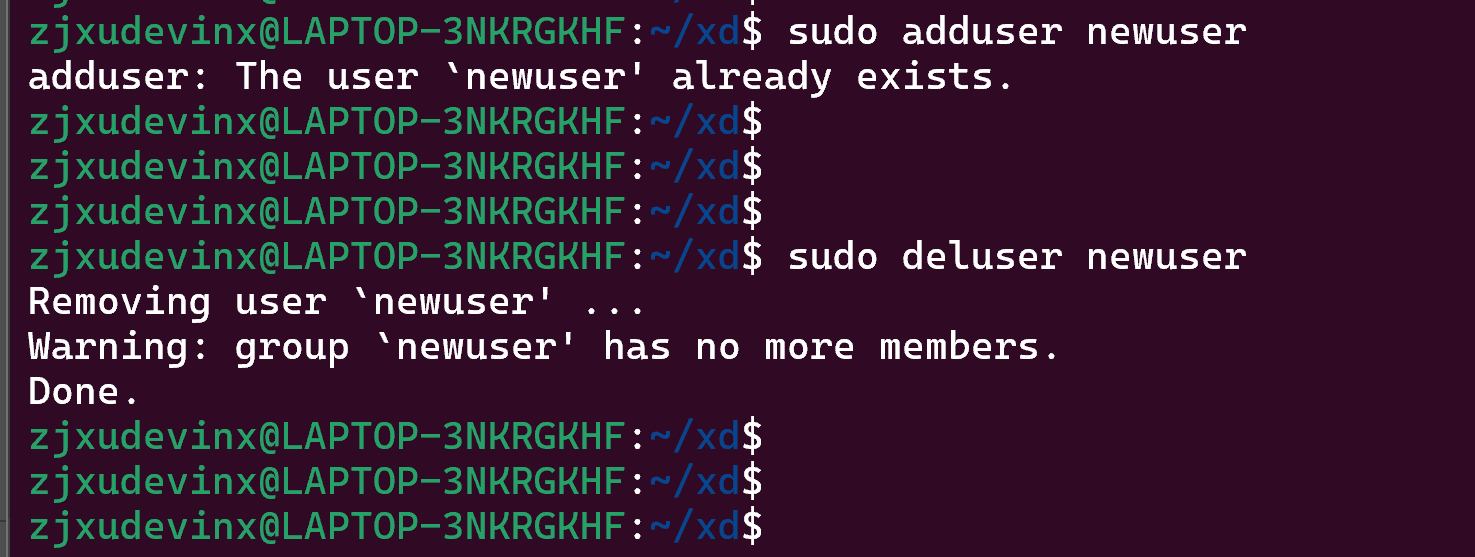
sudo chown newuser permissions\_test.txt



1. **添加和删除用户：**

sudo adduser newuser

sudo deluser newuser

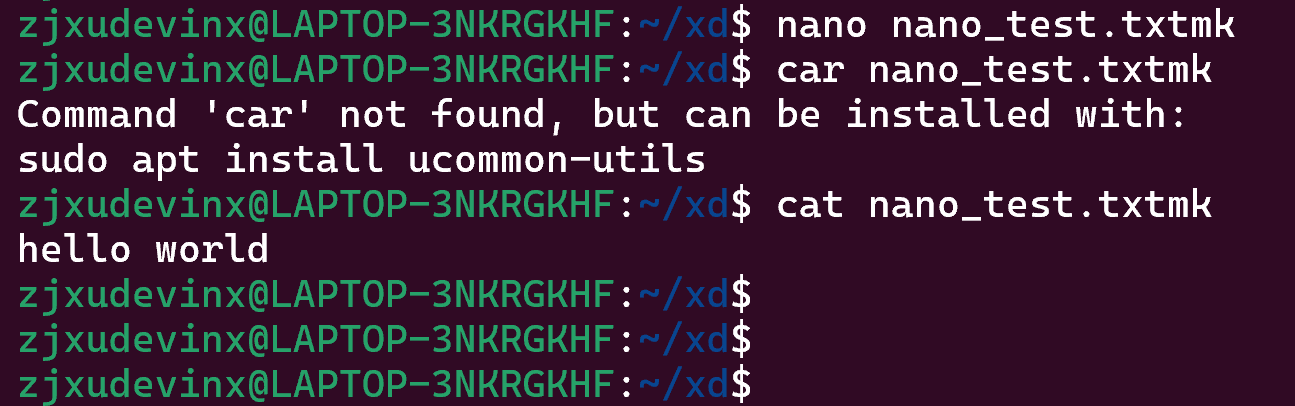


**3. 文本编辑器的使用：nano 和 vi**

1. **使用 nano 编辑文件 nano\_test.txt：**

nano nano\_test.txtmk

* + **练习**：在 nano 中输入内容，保存并退出（Ctrl+O 保存，Ctrl+X 退出）。

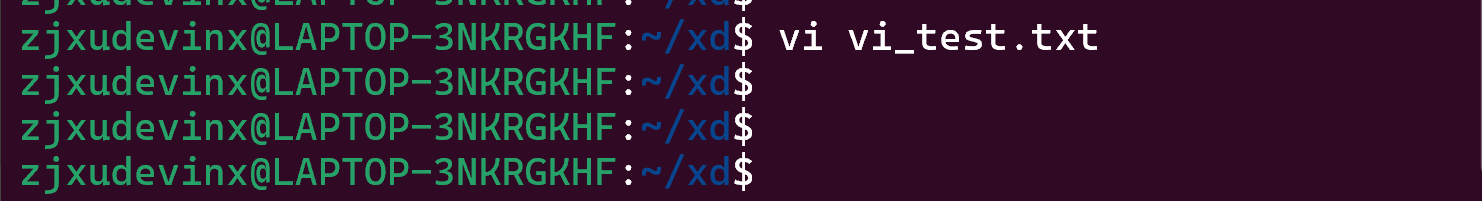


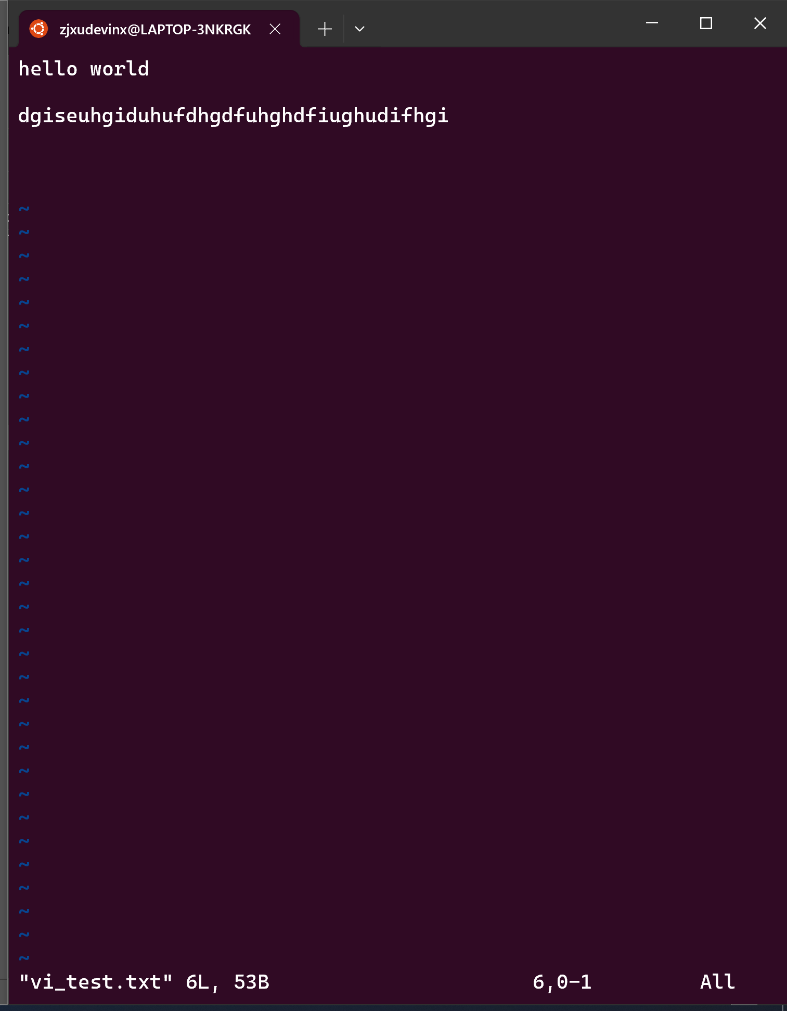


1. **使用 vi 编辑文件 vi\_test.txt：**

vi vi\_test.txt

* + **练习**：进入插入模式（按 i），输入文本，退出并保存（按 Esc 键，输入 :wq）。
  + **常用命令**：
    - i 进入插入模式
    - Esc 退出编辑模式
    - :w 保存文件
    - :q 退出
    - :wq 保存并退出
    - :q! 强制退出不保存





**4. 系统信息与进程管理**

1. **查看当前用户、系统时间和磁盘使用情况：**

whoami

date

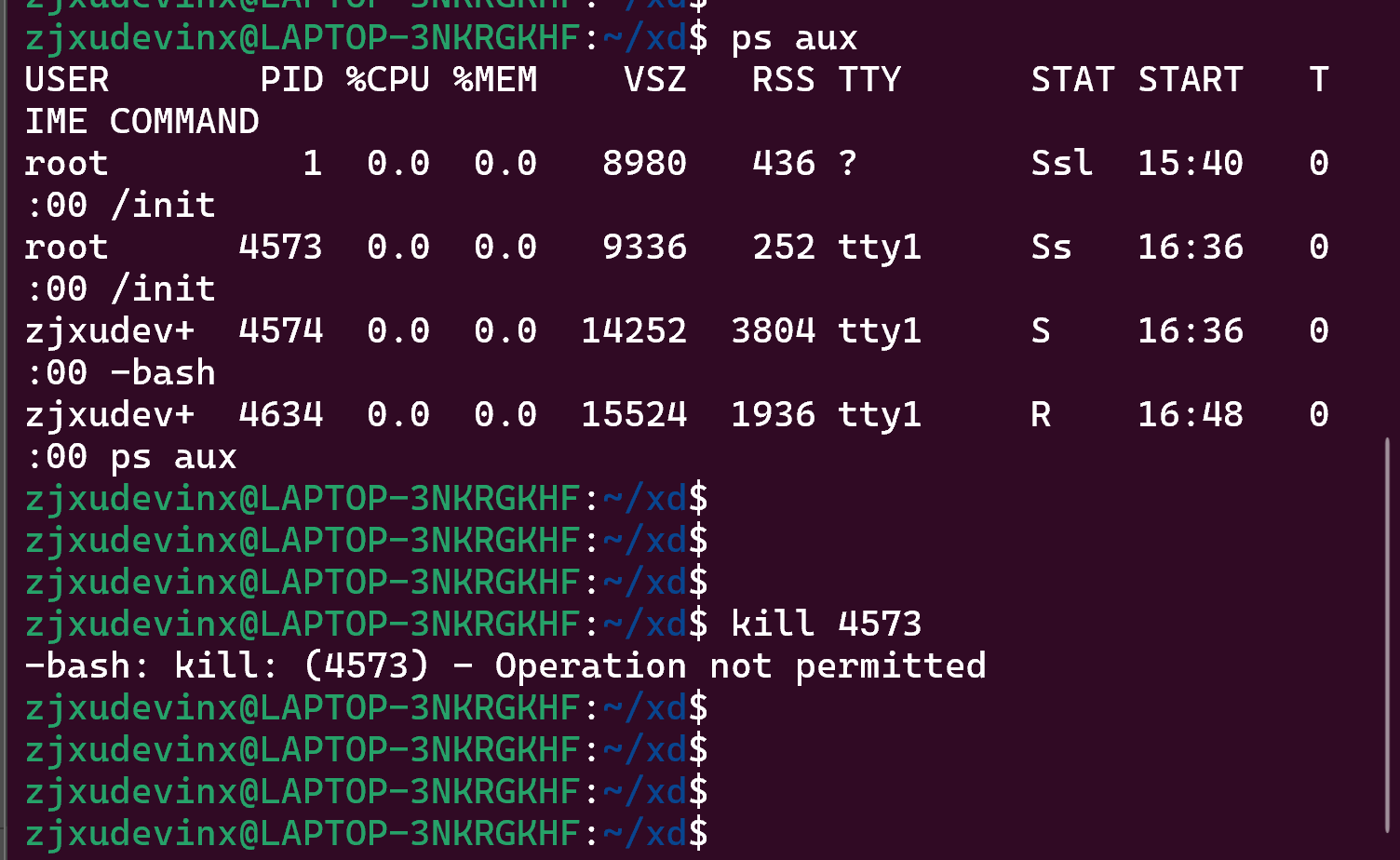
df -h



1. **查看当前运行的进程并杀掉某个进程：**

ps aux

kill <进程ID>



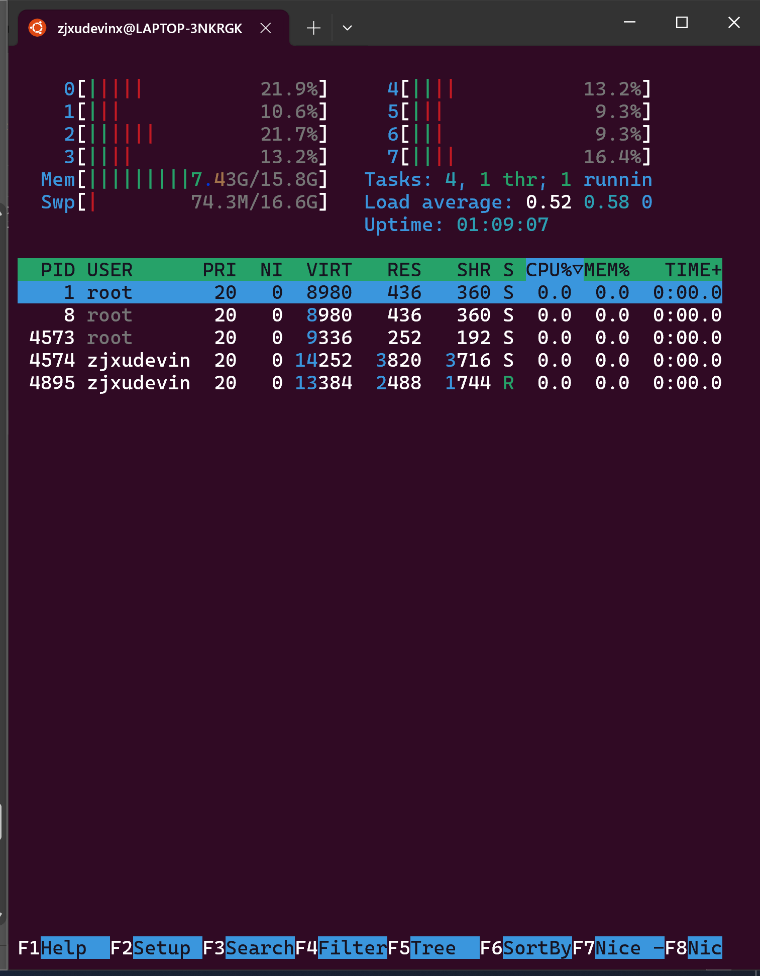
1. **使用 htop 查看系统资源：**

sudo apt update

sudo apt install htop

htop

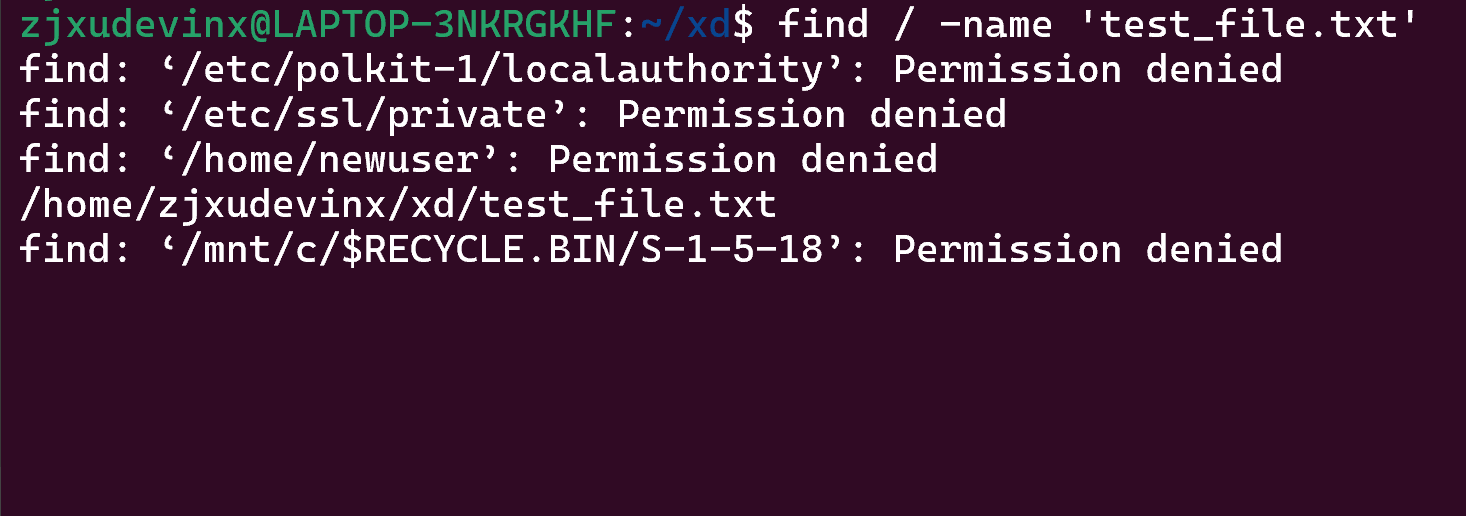




**5. 文件搜索与内容查找**

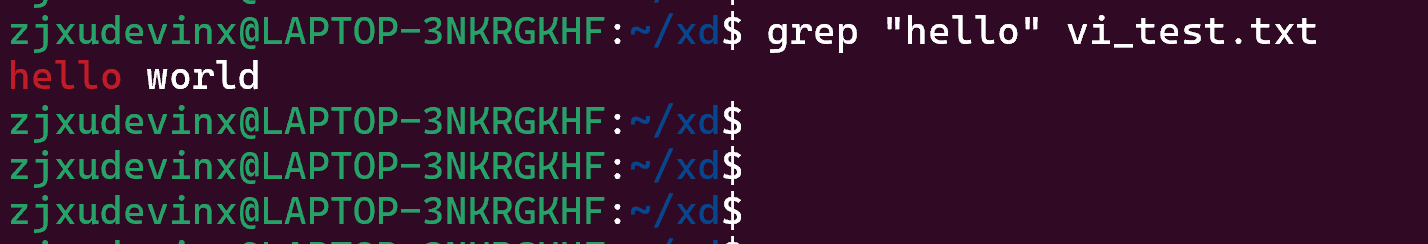
1. **查找文件：**

find / -name "test\_file.txt" # 在根目录下查找



1. **查找文件中的内容：**

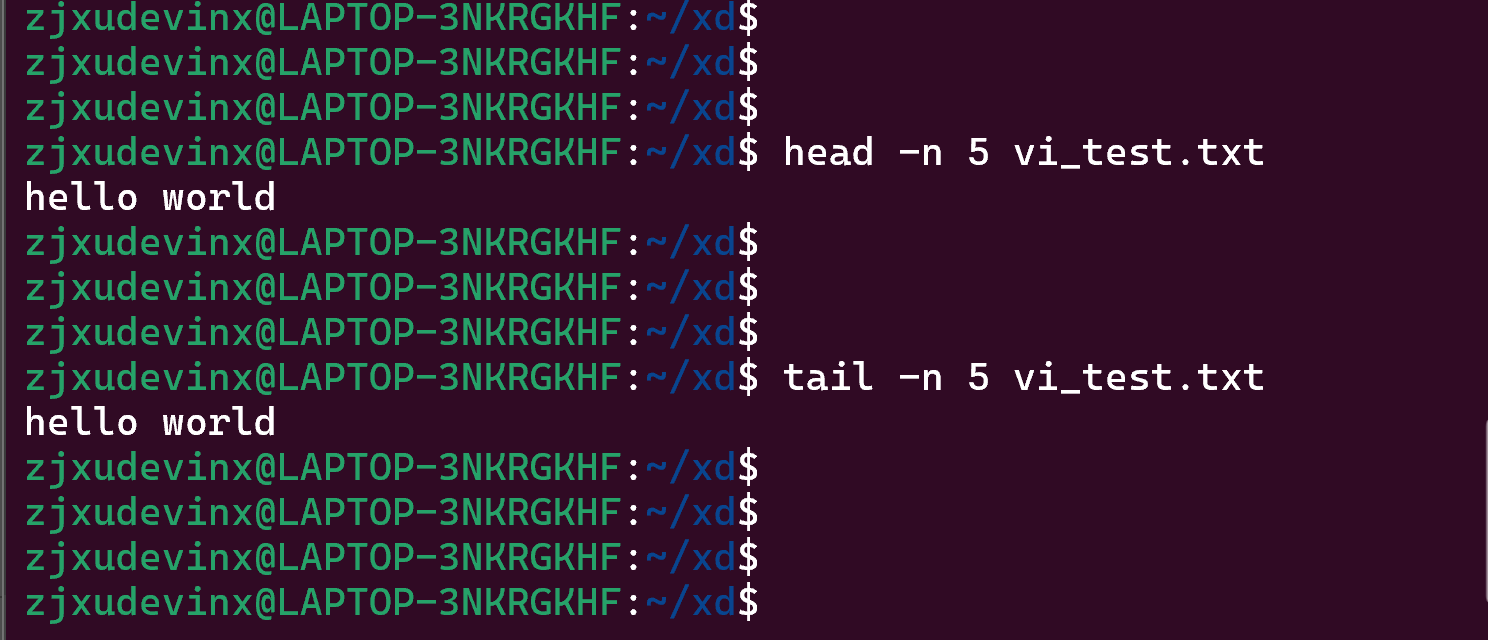
grep "Hello" vi\_test.txt # 在文件中查找包含 "Hello" 的行



1. **查看文件的前几行和后几行：**

head -n 5 vi\_test.txt

tail -n 5 vi\_test.txt



**6. 压缩与解压缩文件**

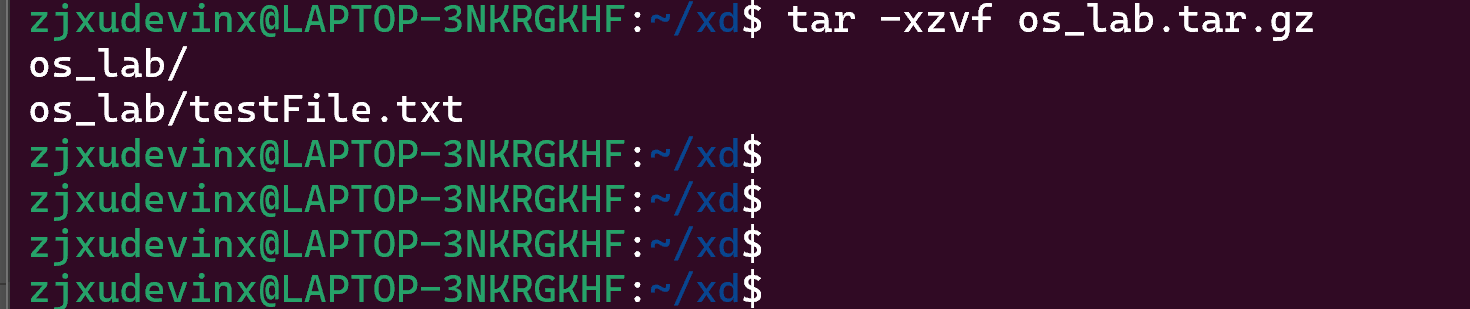
1. **压缩目录 os\_lab：**

tar -czvf os\_lab.tar.gz os\_lab



1. **解压缩文件：**

tar -xzvf os\_lab.tar.gz

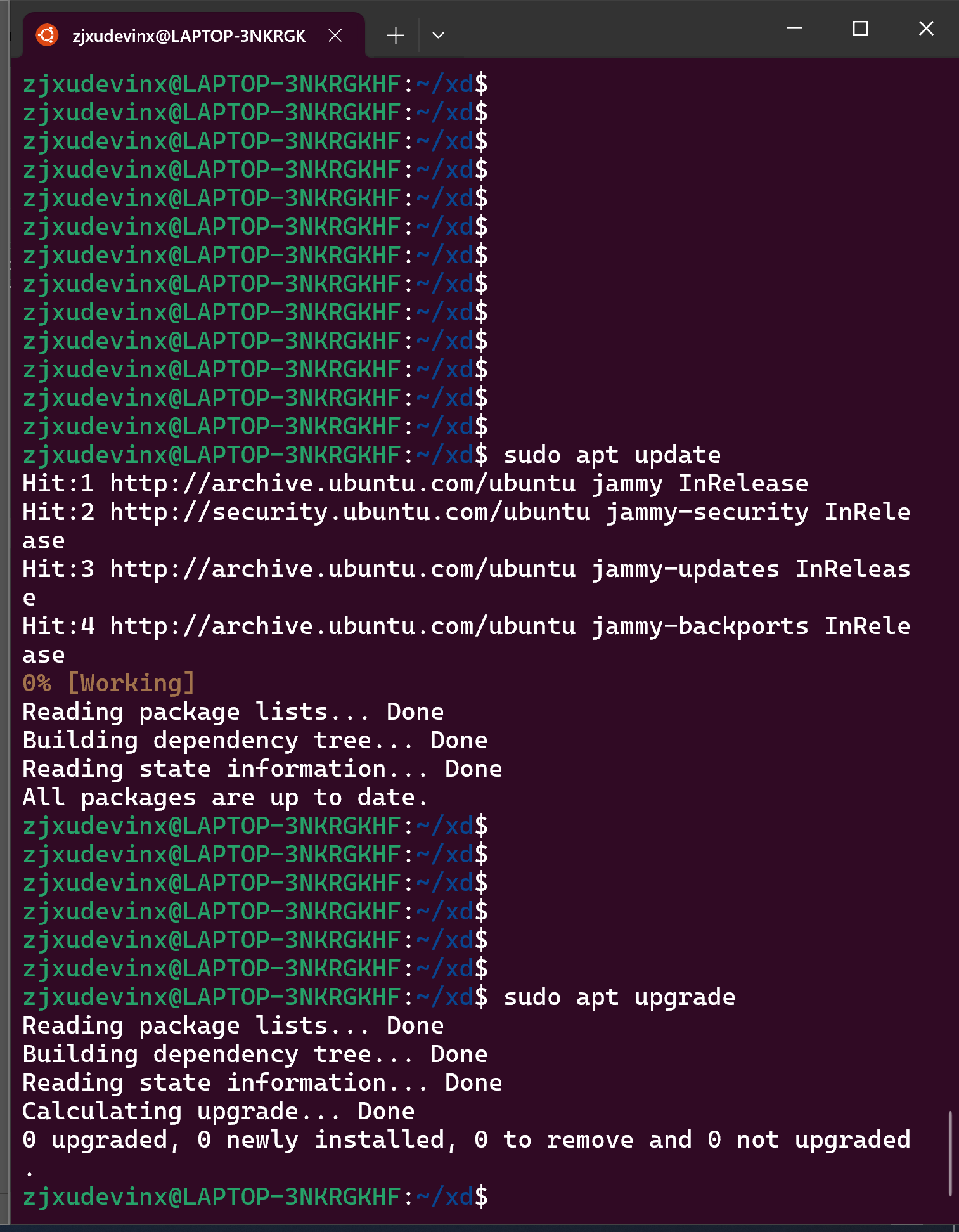


**7. 软件包管理与系统更新**

1. **更新系统软件源和升级系统：**

sudo apt update

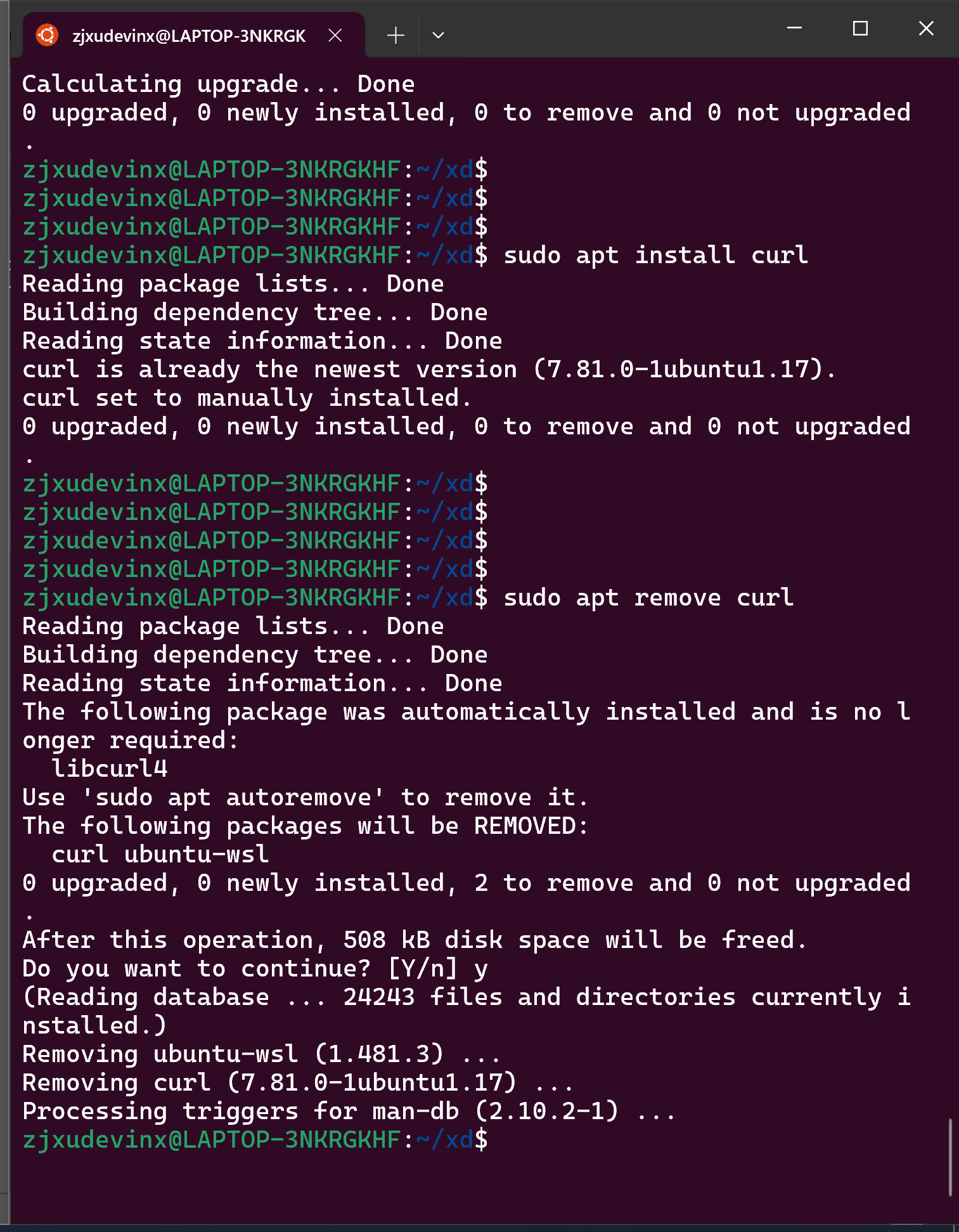
sudo apt upgrade



1. **安装和卸载软件：**

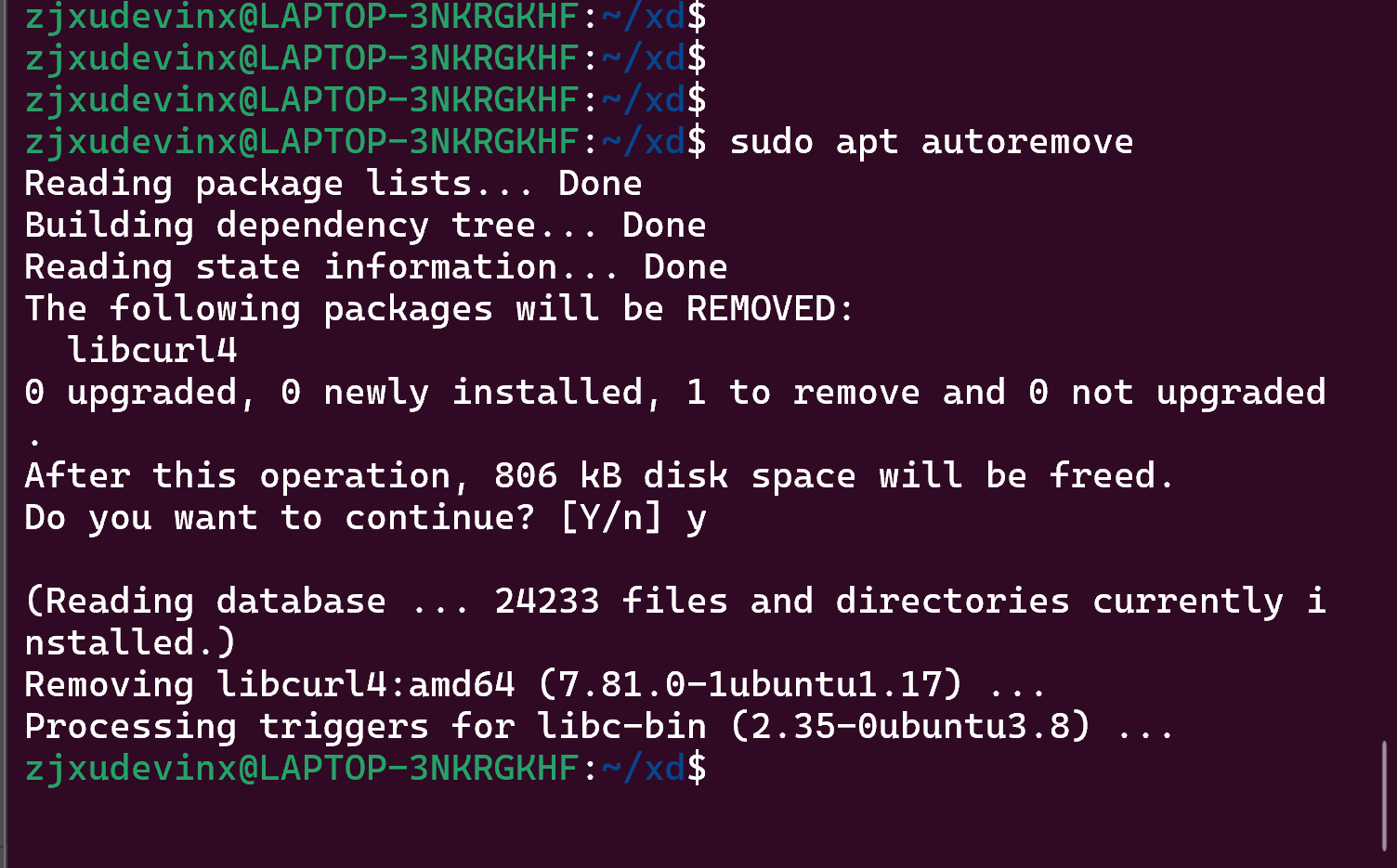
sudo apt install curl

sudo apt remove curl



1. **清理不再需要的软件包：**

sudo apt autoremove



**常见错误提示**

* 权限不足时请使用 sudo 命令提升权限。
* 编辑器 vi 中，按 Esc 键可以退出编辑模式，输入 :q! 强制退出。
* 安装软件时遇到网络问题，请检查网络连接或更换软件源。

心得体会

在这节实验课中，我按照文档要求输入执行各种linux下的常用指令，以及熟悉了linux中常用的文本编辑工具vi、nano，通过这次实验，我对各种指令更加熟悉，并且发现在linux中的操作虽然刚上手时会比windows慢很多，但熟练之后是非常便捷的，有利于提高开发效率。

## 实验课作业第3回：

**操作系统实验：利用 Shell 编程**

**目标：**

通过本次实验，将学会使用 Shell 脚本来编写简单的程序，理解 Shell 编程的基本语法，并掌握如何通过脚本执行一系列操作系统任务，如文件操作、进程管理等。

**实验内容大纲：**

1. **Shell 基础知识**
   * Shell 是什么？
   * 常见的 Shell 类型（如 Bash、Zsh 等）
   * Shell 脚本的工作原理
   * 如何编写和运行 Shell 脚本
2. **Shell 编程的基本语法**
   * 变量定义与使用
   * 输入与输出
   * 条件语句（if-else）
   * 循环（for、while）
   * 函数定义
   * 文件与目录操作
   * 管道与重定向
3. **实验任务**
   * 编写简单的 Shell 脚本
   * 实现自动化的文件管理
   * 利用 Shell 脚本控制进程
   * 使用 Shell 脚本进行系统监控
4. **程序示例与详细分析**
   * 文件操作示例
   * 进程控制示例
   * 系统监控示例

**Shell 基础知识**

**Shell 是什么？**

* **Shell** 是用户与操作系统内核交互的接口。它是一种命令行解释器，用户可以通过 Shell 输入命令并执行。
* 在 Linux 系统中，最常见的 Shell 是 **Bash**（Bourne Again Shell），其他 Shell 包括 Zsh、Fish 等。

**Shell 脚本的工作原理**

* Shell 脚本是一种包含命令行指令的文本文件。通过执行这些命令，用户可以自动化处理任务。
* **如何编写并运行 Shell 脚本：**
  1. 创建一个文本文件，并将 Shell 命令写入其中。
  2. 使文件可执行：chmod +x script.sh
  3. 运行脚本：./script.sh

**Shell 编程的基本语法**

**1. 变量定义与使用**

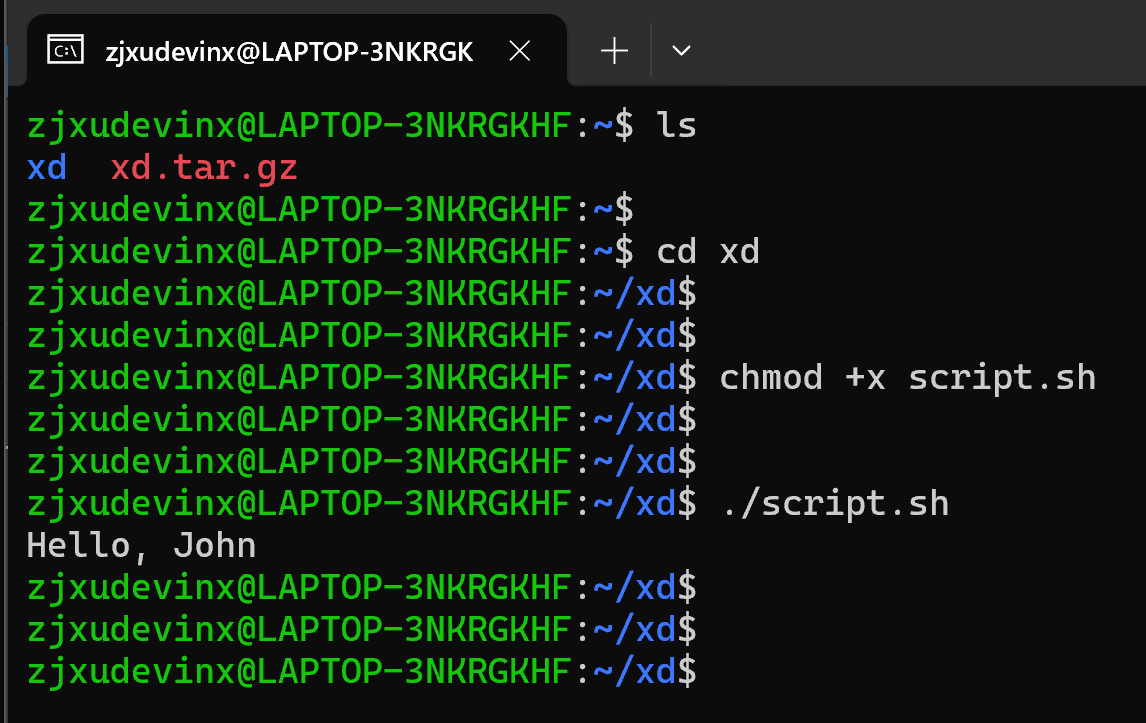
* Shell 中的变量不需要显式声明，用 = 进行赋值：

# 赋值

NAME="John"

# 使用变量

echo "Hello, $NAME"



**2. 条件语句**

* Shell 使用 if-else 语句进行条件判断。可以结合 test 命令或 [ ] 进行条件判断。

# 条件判断示例

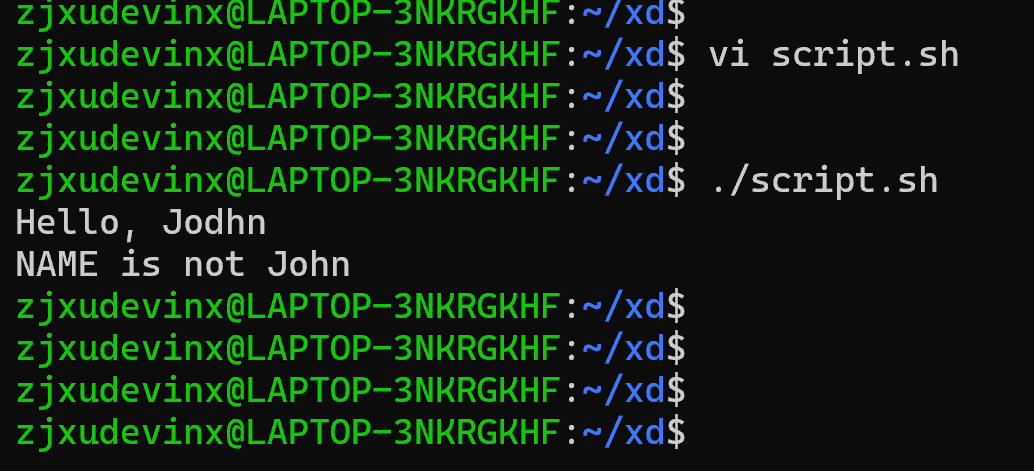
if [ $NAME = "John" ]; then

echo "Name is John"

else

echo "Name is not John"

fi



**3. 循环**

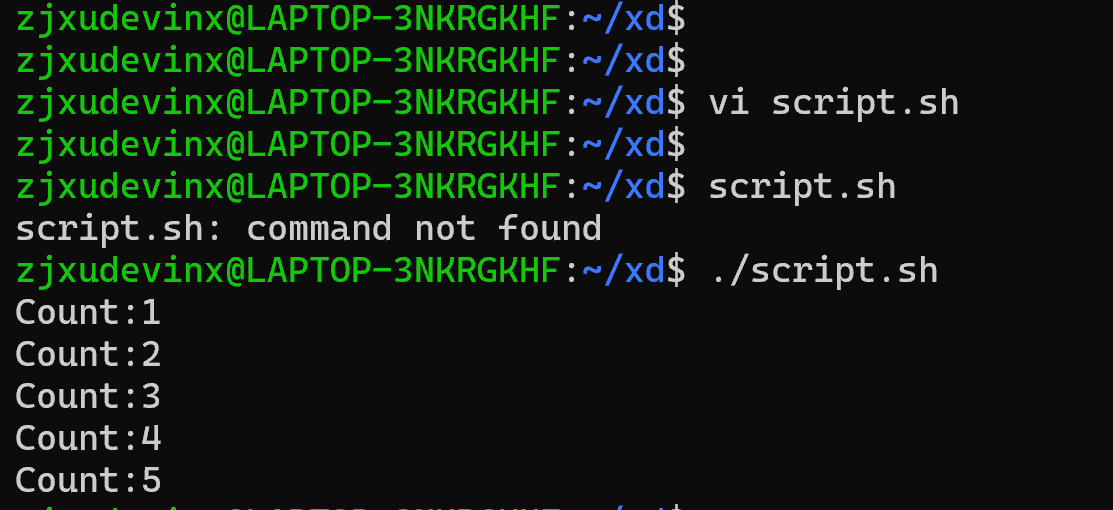
* Shell 提供多种循环结构，最常用的是 for 和 while。

# for 循环示例

for i in 1 2 3 4 5; do

echo "Number: $i"

done



# while 循环示例

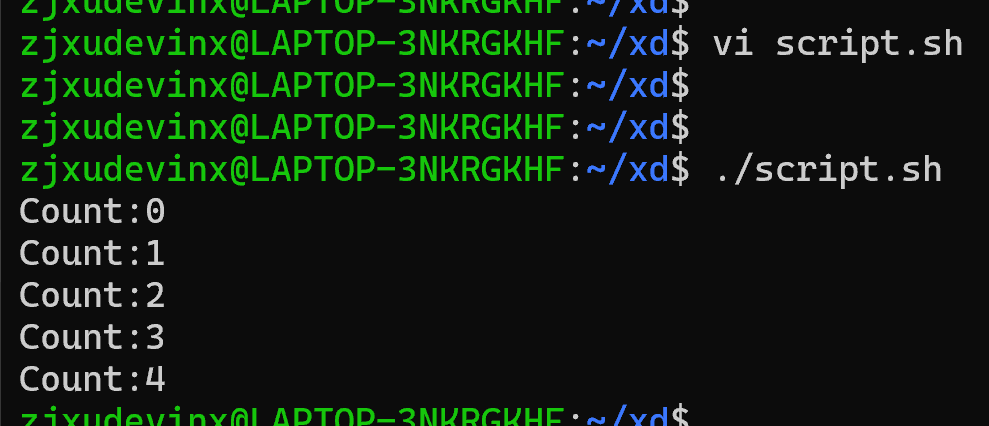
COUNT=0

while [ $COUNT -lt 5 ]; do

echo "Count: $COUNT"

COUNT=$((COUNT + 1))

Done



**4. 函数**

* 可以定义 Shell 函数来复用代码：

# 函数定义

my\_function() {

echo "This is a function"

}

# 调用函数

my\_function



**5. 文件与目录操作**

* 文件操作是 Shell 脚本的重要组成部分。

# 创建目录

mkdir my\_directory

# 创建文件

touch my\_file.txt

# 删除文件

rm my\_file.txt

# 显示文件内容

cat my\_file.txt

**6. 管道与重定向**

* **管道 (|)**：将一个命令的输出作为另一个命令的输入。
* **重定向 (>, >>)**：将输出重定向到文件。

# 管道示例：将 ls 输出传给 grep 进行过滤

ls | grep "my\_file"

# 重定向示例：将命令的输出写入文件

echo "Hello, World" > output.txt

**实验任务**

**实验 1：创建自动化文件管理的 Shell 脚本**

**目标**：编写一个脚本自动化地创建目录、移动文件、备份文件等操作。

#!/bin/bash

# 创建目录

mkdir project\_backup

# 移动文件到新目录

mv \*.txt project\_backup/

# 压缩备份目录

tar -czvf project\_backup.tar.gz project\_backup/

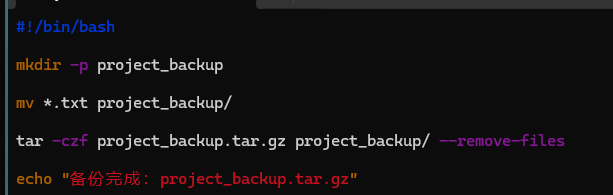
# 显示完成信息

echo "文件备份完成，存储在 project\_backup.tar.gz"

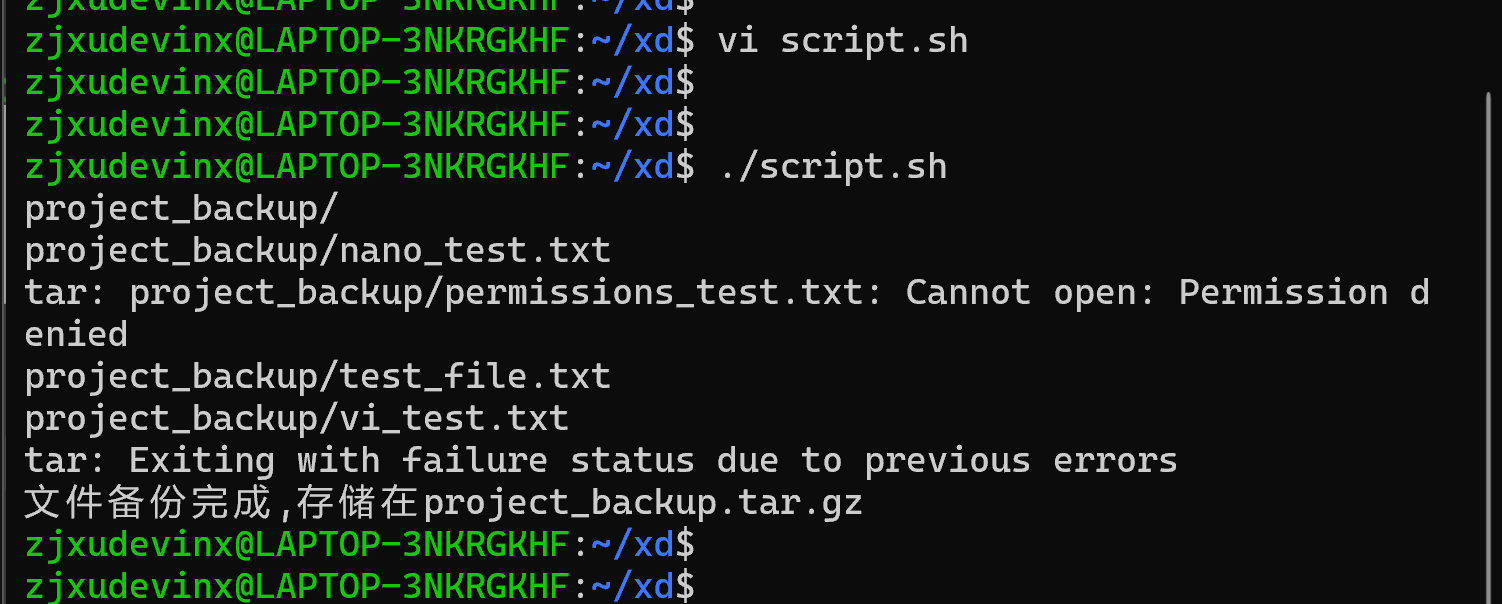
**解释：**

* 脚本首先创建一个新的目录 project\_backup，然后将所有 .txt 文件移动到该目录中。最后将该目录压缩为 project\_backup.tar.gz。

实验代码



实验结果



**实验 2：利用 Shell 脚本控制进程**

**目标**：编写一个脚本，启动指定的程序，并检查它们的运行状态。

#!/bin/bash

# 启动程序

echo "启动 Firefox 浏览器..."

firefox & # 将程序放入后台执行

# 获取进程 PID

PID=$!

# 检查进程是否在运行

if ps -p $PID > /dev/null; then

echo "Firefox 正在运行，进程 ID: $PID"

else

echo "Firefox 没有运行"

fi

# 停止进程

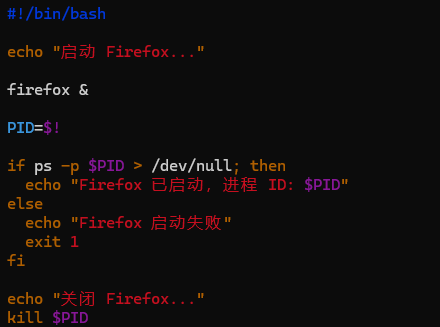
echo "关闭 Firefox 浏览器..."

kill $PID

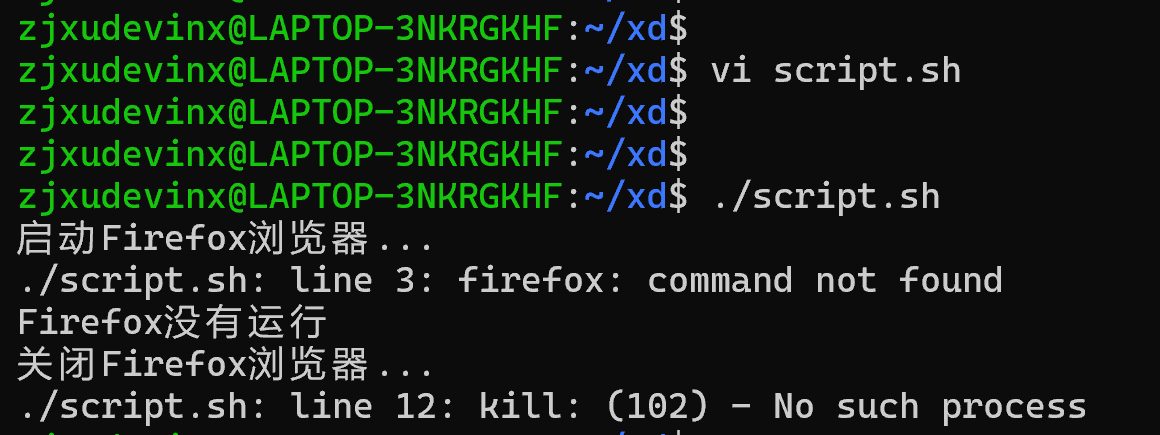
**解释：**

* 脚本启动 Firefox 浏览器并将其放入后台执行。通过 $!，可以获得 Firefox 进程的 PID，然后使用 ps 命令检查该进程是否正在运行。最后，脚本通过 kill 命令关闭 Firefox。

**实验代码**

****

**实验结果**

****

**由于我是用的是wsl的ubuntu,为未集成firefox**

**实验 3：使用 Shell 脚本进行系统监控**

**目标**：编写一个脚本，监控系统资源使用情况（如 CPU 使用率、内存使用情况等）。

#!/bin/bash

# 显示 CPU 使用情况

echo "CPU 使用情况："

mpstat

# 显示内存使用情况

echo "内存使用情况："

free -h

# 显示磁盘使用情况

echo "磁盘使用情况："

df -h

# 显示当前系统中运行的进程数

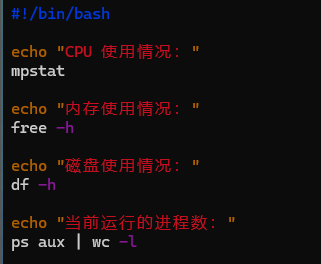
echo "当前运行的进程数："

ps aux | wc -l

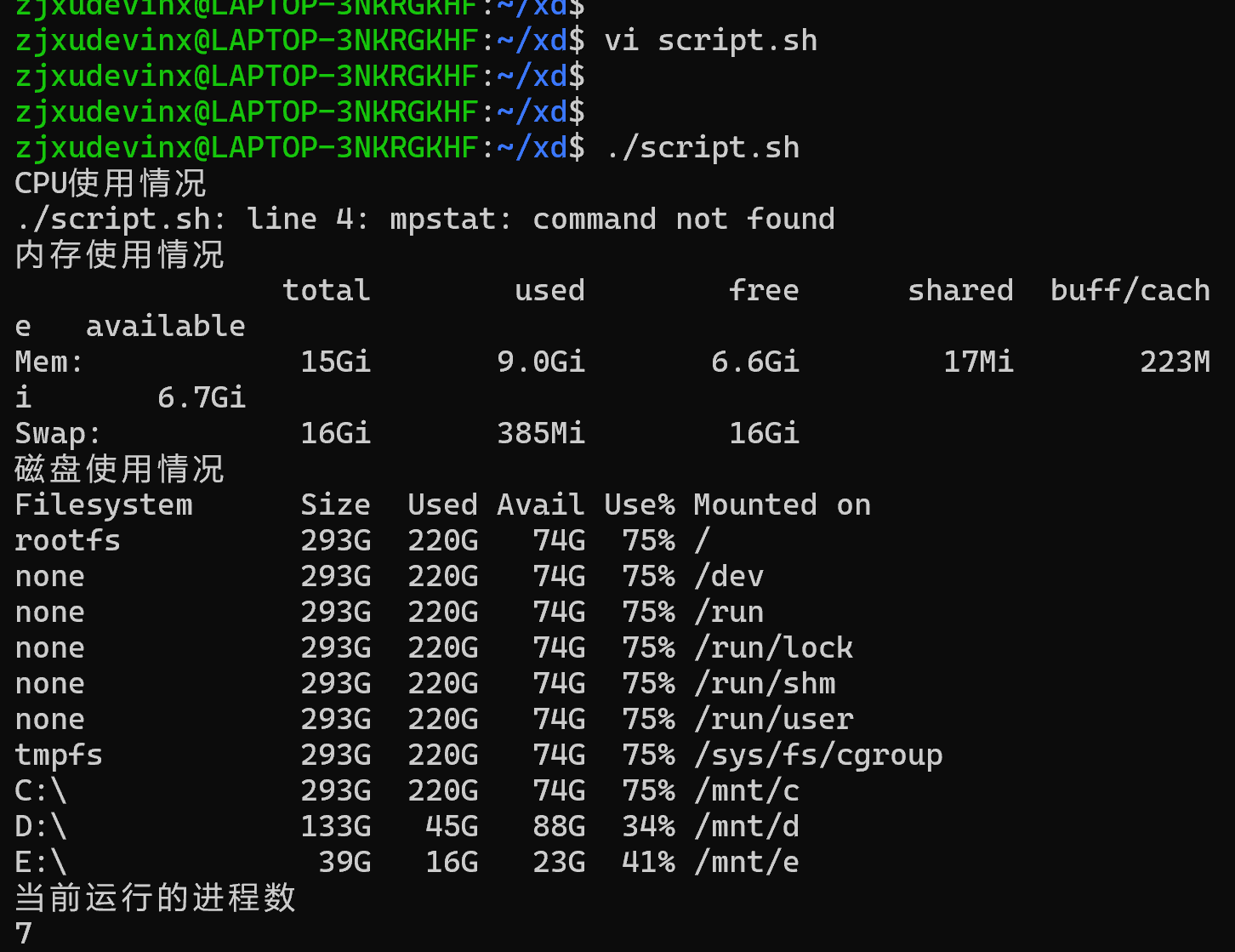
**解释：**

* 该脚本使用 mpstat、free 和 df 命令显示系统的 CPU、内存和磁盘使用情况，最后使用 ps aux 统计当前运行的进程数。

实验代码



实验结果



**程序示例与详细分析**

**示例 1：文件自动整理脚本**

#!/bin/bash

# 文件类型分类整理

echo "开始整理文件..."

# 创建分类目录

mkdir -p images documents others

# 移动文件到对应目录

mv \*.jpg \*.png \*.gif images/ 2>/dev/null

mv \*.doc \*.pdf \*.txt documents/ 2>/dev/null

mv \* others/ 2>/dev/null

echo "文件整理完成"

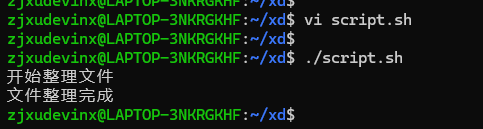
**解释与分析：**

* 脚本根据文件类型（图片、文档等）自动分类整理文件，并将它们分别移动到对应的目录下。如果文件类型不属于任何已分类的类型，则将其移动到 others 目录中。2>/dev/null 用于忽略找不到文件时的错误输出。

实验代码



实验结果



**示例 2：定时备份脚本**

#!/bin/bash

# 获取当前日期时间

DATE=$(date +%Y-%m-%d\_%H-%M-%S)

# 备份目录

BACKUP\_DIR="backup\_$DATE"

# 创建备份目录

mkdir $BACKUP\_DIR

# 复制文件到备份目录

cp -r ~/Documents/\* $BACKUP\_DIR

# 打包备份文件

tar -czvf $BACKUP\_DIR.tar.gz $BACKUP\_DIR

# 删除原始备份目录

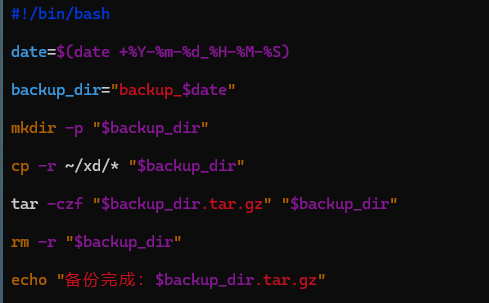
rm -r $BACKUP\_DIR

echo "备份完成：$BACKUP\_DIR.tar.gz"

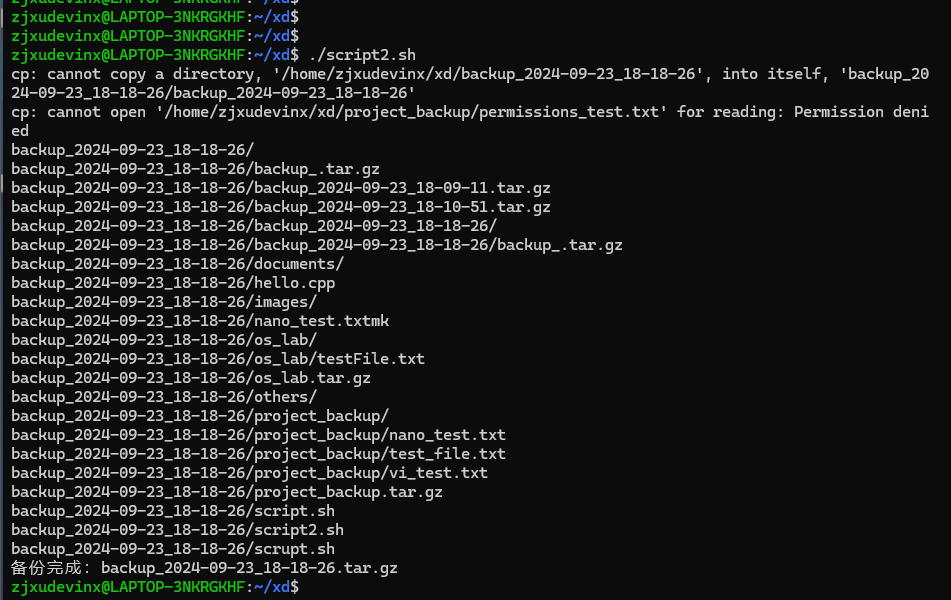
**解释与分析：**

* 该脚本定期备份用户的 Documents 目录。它首先根据当前时间创建一个备份目录，然后将所有文件复制到该目录并打包为压缩文件。最后，删除原始备份目录，只保留压缩包。

实验代码



实验结果



**总结**

通过本次实验课，将掌握以下技能：

1. **编写基本的 Shell 脚本**：能够独立编写脚本，实现简单的任务自动化。
2. **利用 Shell 脚本管理系统资源**：将学会如何通过脚本进行文件操作、进程控制和系统监控。
3. **深入理解 Shell 脚本中的控制流**：能够灵活使用条件语句、循环和函数，编写更复杂的程序。

通过实验和程序的分析，不仅能够掌握 Shell 编程的基本语法，还可以将这些知识应用到日常任务的自动化处理中。

补充内容：

在 Linux 和其他类 Unix 系统中，2>/dev/null 是一种**重定向**操作，用于将错误输出（标准错误流，stderr）重定向到 /dev/null，从而**忽略错误信息**。具体来说，它的作用是将命令执行过程中产生的错误输出丢弃，不显示在终端上。

**详细说明：**

* **标准输出 (stdout)**：在 Shell 中，程序执行成功时通常会输出一些信息，默认情况下，这些信息会显示在终端上。这些输出被称为**标准输出**，标识符为 1。
* **标准错误输出 (stderr)**：当程序执行过程中遇到错误时，会输出错误信息。默认情况下，这些错误信息也会显示在终端上。这些输出被称为**标准错误输出**，标识符为 2。
* **/dev/null**：/dev/null 是一个特殊的设备文件，它类似于一个“黑洞”。任何写入 /dev/null 的数据都会被丢弃，没有任何返回或显示。因此，**将输出重定向到 /dev/null 意味着忽略该输出**。

**2>/dev/null 的含义：**

* **2>**：这个部分指的是将**标准错误流 (stderr)** 重定向。2 是标准错误流的标识符。
* **/dev/null**：表示将错误输出丢弃。

因此，2>/dev/null 的完整意思是：**将标准错误输出重定向到 /dev/null，即忽略错误输出，不在终端上显示。**

**示例解释：**

bash

复制代码

mv \*.jpg images/ 2>/dev/null

**含义：**

* 这个命令试图将当前目录下所有 .jpg 文件移动到 images/ 目录中。
* 如果当前目录下没有任何 .jpg 文件，mv 命令会产生一个错误信息，例如：“没有这样的文件”。
* 通过添加 2>/dev/null，这些错误信息会被丢弃，用户在终端上不会看到任何错误提示，即使没有找到 .jpg 文件也不会显示错误信息。

**具体例子：**

**场景 1：不使用 2>/dev/null**

bash

复制代码

mv \*.txt documents/

假设当前目录下没有任何 .txt 文件，终端会显示类似如下的错误信息：

bash

复制代码

mv: cannot stat '\*.txt': No such file or directory

这表示 mv 无法找到匹配的文件，并输出了错误信息。

**场景 2：使用 2>/dev/null**

bash

复制代码

mv \*.txt documents/ 2>/dev/null

如果没有 .txt 文件，错误信息将不会显示，系统会忽略这些错误，而不会在终端中输出任何内容。这种方式适用于你不希望用户看到错误提示的情况，特别是在不确定文件是否存在时。

**应用场景：**

* **静默处理错误**：在自动化脚本中，如果某些操作失败但不会影响整体逻辑，使用 2>/dev/null 可以避免不必要的错误提示，保持脚本的输出简洁。
* **文件或命令的条件执行**：如果某些命令可能会报错，而这些错误是可以预期的且不影响后续流程，使用 2>/dev/null 可以让脚本继续运行，而不会因为错误而中断。

**总结：**

2>/dev/null 是一种常见的重定向方法，用于**忽略命令执行过程中产生的错误输出**。这在某些情况下非常有用，尤其是当我们期望脚本自动处理可能产生错误的场景而不希望用户看到错误提示时。

Shell进阶：

在掌握了 Shell 编程的基础知识之后，可以进一步学习一些更高级和复杂的 Shell 编程概念。这些知识点不仅能提高脚本的灵活性和效率，还能处理复杂的任务和系统操作。以下是一些常见的高级 Shell 编程概念及其相应的代码示例和详细解说。

**1. 数组的使用**

虽然 Shell 脚本中不像其他编程语言那样有强大的数据结构，但数组是 Shell 中非常实用的一种数据存储方式。

**数组的声明与使用**

# 声明数组

my\_array=("apple" "banana" "cherry")

# 访问数组元素

echo "第一个元素是：${my\_array[0]}"

# 获取数组长度

echo "数组长度是：${#my\_array[@]}"

# 遍历数组

for fruit in "${my\_array[@]}"; do

echo "水果: $fruit"

done

**解释：**

* Shell 中的数组可以通过 () 进行定义，数组中的每个元素用空格分隔。
* 使用 ${my\_array[index]} 来访问数组元素。
* ${#my\_array[@]} 可以获取数组的长度。

**2. 命令替换与嵌套**

Shell 支持通过命令替换的方式将命令的执行结果赋给变量。常见的命令替换有两种语法形式：`command` 和 $(command)。在高级编程中，命令替换是处理动态数据的基础。

**命令替换示例**

#!/bin/bash

# 获取当前日期并存入变量

current\_date=$(date +"%Y-%m-%d %H:%M:%S")

echo "当前时间是: $current\_date"

# 使用命令替换计算目录下文件数量

file\_count=$(ls | wc -l)

echo "当前目录下有 $file\_count 个文件"

**解释：**

* $(command) 将命令的输出作为字符串返回。date 命令获取当前时间，ls | wc -l 通过管道计算文件数量。
* $(command) 和反引号 `command` 效果相同，但推荐使用 $() 语法，因为它更容易嵌套使用。

**3. 进程控制与后台执行**

Shell 提供了启动和管理后台任务的能力，通过将命令放入后台运行来提高程序的效率。同时，Shell 也允许我们对后台任务进行监控和管理。

**后台运行与进程控制**

#!/bin/bash

# 启动一个后台进程

sleep 60 & # 将 sleep 命令放入后台运行，延迟60秒

# 获取后台进程的 PID

pid=$!

echo "后台进程 PID: $pid"

# 等待进程结束

wait $pid

echo "进程 $pid 结束"

**解释：**

* sleep 60 & 表示将 sleep 命令放入后台运行，& 是放入后台运行的符号。
* $! 可以获取最近放入后台运行的进程的 PID。
* wait $pid 会让脚本等待后台进程完成再继续执行。

**4. trap 捕捉信号处理**

在高级 Shell 脚本中，我们可以使用 trap 来捕捉和处理信号（例如 SIGINT, SIGTERM 等）。这对于编写健壮的脚本非常有用，特别是在脚本被外部终止时执行清理任务。

**trap 捕捉信号**

#!/bin/bash

# 定义当脚本接收到 SIGINT（Ctrl+C） 时的处理行为

trap "echo '脚本被中断！'; exit" SIGINT

# 进入一个无限循环

while true; do

echo "运行中... 按 Ctrl+C 停止"

sleep 2

done

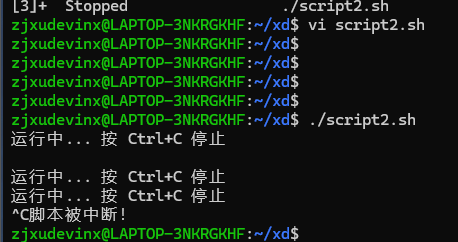
**解释：**

* trap 命令用于定义当脚本收到某个信号时要执行的操作。这里定义了当收到 SIGINT 信号时，显示一条消息并退出。
* SIGINT 是当用户按 Ctrl+C 时发送的信号，trap 捕捉到后执行相应的命令。

实验代码



实验结果



**5. Here Document**

**Here Document** 是一种将多行文本传递给命令的方式。它可以简化复杂命令的输入，特别适用于生成长文本、批处理任务或自动化配置。

**Here Document 示例**

#!/bin/bash

# 使用 here document 输出多行文本到文件

cat <<EOF >output.txt

这是一个多行文本

可以用于输出到文件或者作为命令的输入

多行文本结束

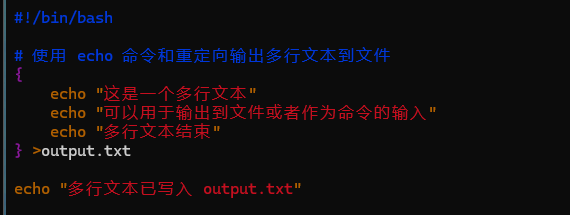
EOF

echo "多行文本已写入 output.txt"

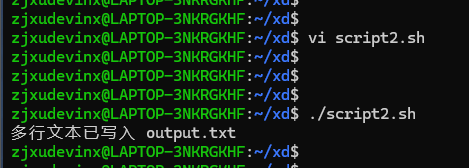
**解释：**

* <<EOF 定义了 Here Document 的起始符，EOF 是结束标记，可以替换成其他标记符。cat 命令会将这些多行内容输出到 output.txt。
* Here Document 通常用于生成脚本中的多行文本，避免多次使用 echo。

实验代码



实验结果



**6. 并发执行与管道结合**

Shell 可以通过管道将多个命令串联执行，并支持并发执行任务。高级 Shell 脚本可以通过组合管道与并发技术实现高效任务处理。

**并发执行与管道示例**

#!/bin/bash

# 并发下载多个文件

curl -O http://example.com/file1.txt &

curl -O http://example.com/file2.txt &

curl -O http://example.com/file3.txt &

# 等待所有下载完成

wait

# 使用管道将多个命令的输出串联处理

cat file1.txt file2.txt file3.txt | grep "关键字" | sort > result.txt

echo "所有文件已下载并处理，结果保存到 result.txt"

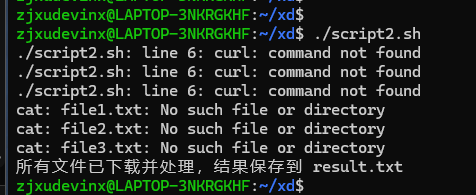
**解释：**

* curl -O 命令用于下载文件，这里通过 & 将多个文件的下载放入后台并行执行。
* wait 命令用于等待所有后台任务完成后继续执行。
* 通过 cat 将三个文件的内容合并，grep 用于过滤关键字，sort 对结果进行排序，并输出到 result.txt。

实验代码



实验结果



**7. 递归函数与递归脚本**

Shell 也支持递归函数和递归调用，可以用于处理递归任务（如遍历目录树或递归处理文件）。

**递归遍历目录**

#!/bin/bash

# 定义递归函数遍历目录

traverse\_dir() {

local dir=$1

echo "进入目录: $dir"

# 遍历当前目录中的所有文件和子目录

for file in "$dir"/\*; do

if [ -d "$file" ]; then

# 如果是目录，则递归调用

traverse\_dir "$file"

else

# 如果是文件，则输出文件名

echo "文件: $file"

fi

done

}

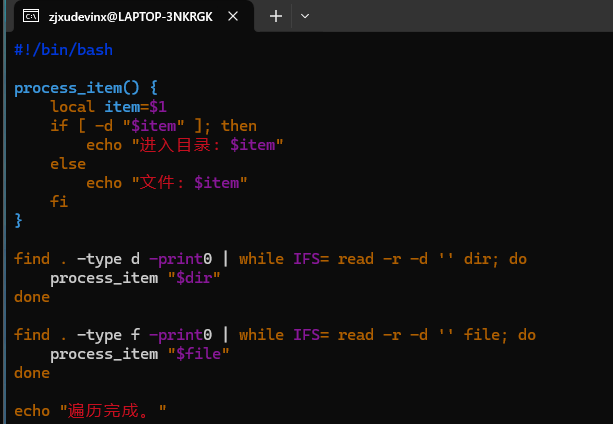
# 调用递归函数，传入当前目录

traverse\_dir .

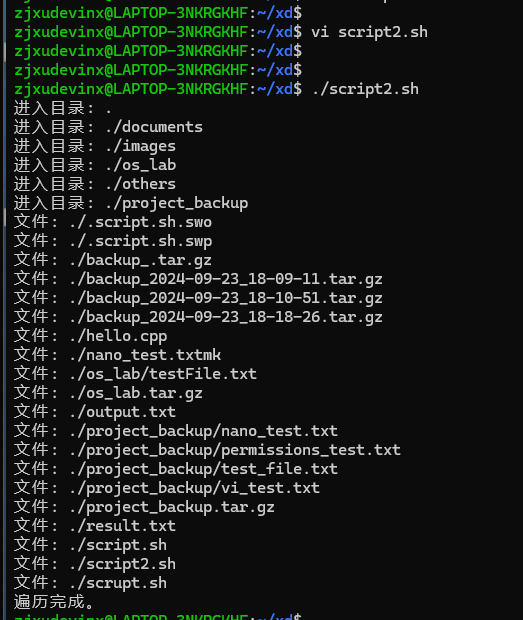
**解释：**

* traverse\_dir 是一个递归函数，它会遍历传入的目录，输出该目录中的所有文件和子目录。如果遇到子目录，则会递归调用自己。
* 递归函数非常适合处理层级结构的数据，如文件系统中的目录结构。

实验代码



实验结果



**8. 正则表达式与模式匹配**

正则表达式是处理文本的强大工具，结合 Shell 的 grep、sed 和 awk 等命令，可以实现复杂的文本处理任务。

**使用 grep 进行模式匹配**

#!/bin/bash

# 搜索包含特定模式的行

grep -E "^user[0-9]+" users.txt

# 使用 `sed` 替换文件中的内容

sed -i 's/old\_word/new\_word/g' file.txt

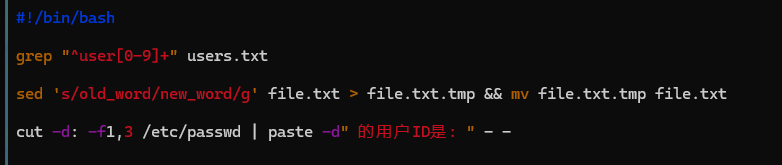
# 使用 `awk` 提取并处理文件中的字段

awk -F: '{ print $1 " 的用户ID是: " $3 }' /etc/passwd

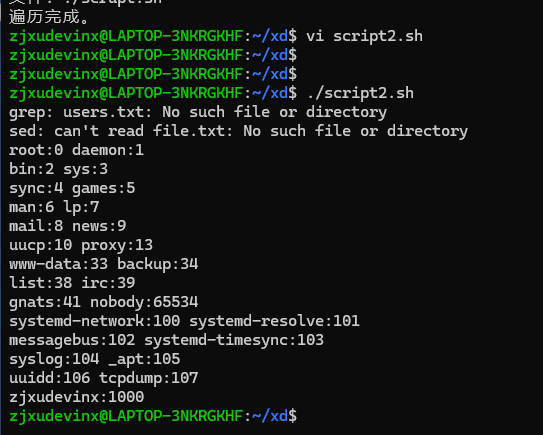
**解释：**

* grep -E 使用正则表达式搜索匹配特定模式的行，例如搜索以 "user" 开头，后面是数字的字符串。
* sed 用于对文件进行流式编辑，s/old\_word/new\_word/g 用于全局替换。
* awk 是一个强大的文本处理工具，用于提取并处理文件中的字段。这里 -F: 指定以冒号分隔字段，print 用于输出字段。

实验代码



实验结果



**总结：**

通过学习这些高级 Shell 编程技术，将掌握如何编写更高效、复杂的 Shell 脚本，处理文件、进程、网络等操作系统任务。这些高级技术在自动化运维、系统管理、批量处理等领域非常实用。

以下是涵盖的知识点：

* 数组和命令替换
* 进程控制与后台执行
* 捕捉信号（trap）
* Here Document
* 并发执行与管道
* 递归函数
* 正则表达式与模式匹配

这些技术能够编写功能强大的 Shell 脚本，解决实际问题并提升系统操作的效率。

实验报告：

--------------------

* 1. 学习并理解本文档中的示例程序；
  2. 动手操作：实验1，实验2，实验3，示例1，示例2，（分别执行各代码，得出相应的结果，将代码和测试结果写进报告）
  3. 动手操作：
     + **4. trap 捕捉信号处理**
     + **7. 递归函数与递归脚本**
     + **8. 正则表达式与模式匹配**

（分别执行各代码，得出相应的结果，将代码和测试结果写进报告）

\*注意：在不改变代码功能的基础上，必须对代码做相应修改和变动。

心得体会

本次实验通过一些示例学习了基本的shell编程，完成一定的逻辑任务，从中，我体会到了shell命令的便利之处，可以通过编写自动化shell脚本代码，来自动完成一些重复性的工作，比如备份、整理文件夹、显示文件信息等等。同时，我发现shell语法对空格非常敏感，很多错误都是因为错误引起的，导致前面两个实验花费了很长时间，之后会更加注意。

## 实验课作业第4回：

文本

描述已自动生成

代码：

#include <unistd.h>

#include <sys/types.h>

#include <errno.h>

#include <sys/wait.h>

#include <stdlib.h>

#include <stdio.h>

int main(){

pid\_t childpid;

int retval;

int status;

for (int i = 1; i <= 3; i++){

childpid = fork();

if (childpid == 0){

printf("CHILD: I am the child process! \n");

printf("CHILD: Here's my PID: %d \n", getpid());

printf("CHILD: My parent's PID is: %d \n", getppid());

printf("CHILD: The value of fork return is: %d \n", childpid);

}

else {

printf("PARENT: I am the parent process! \n");

printf("PARENT: Here's my PID: %d \n", getpid());

printf("PARENT: The value of my child's PID is: %d \n", childpid);

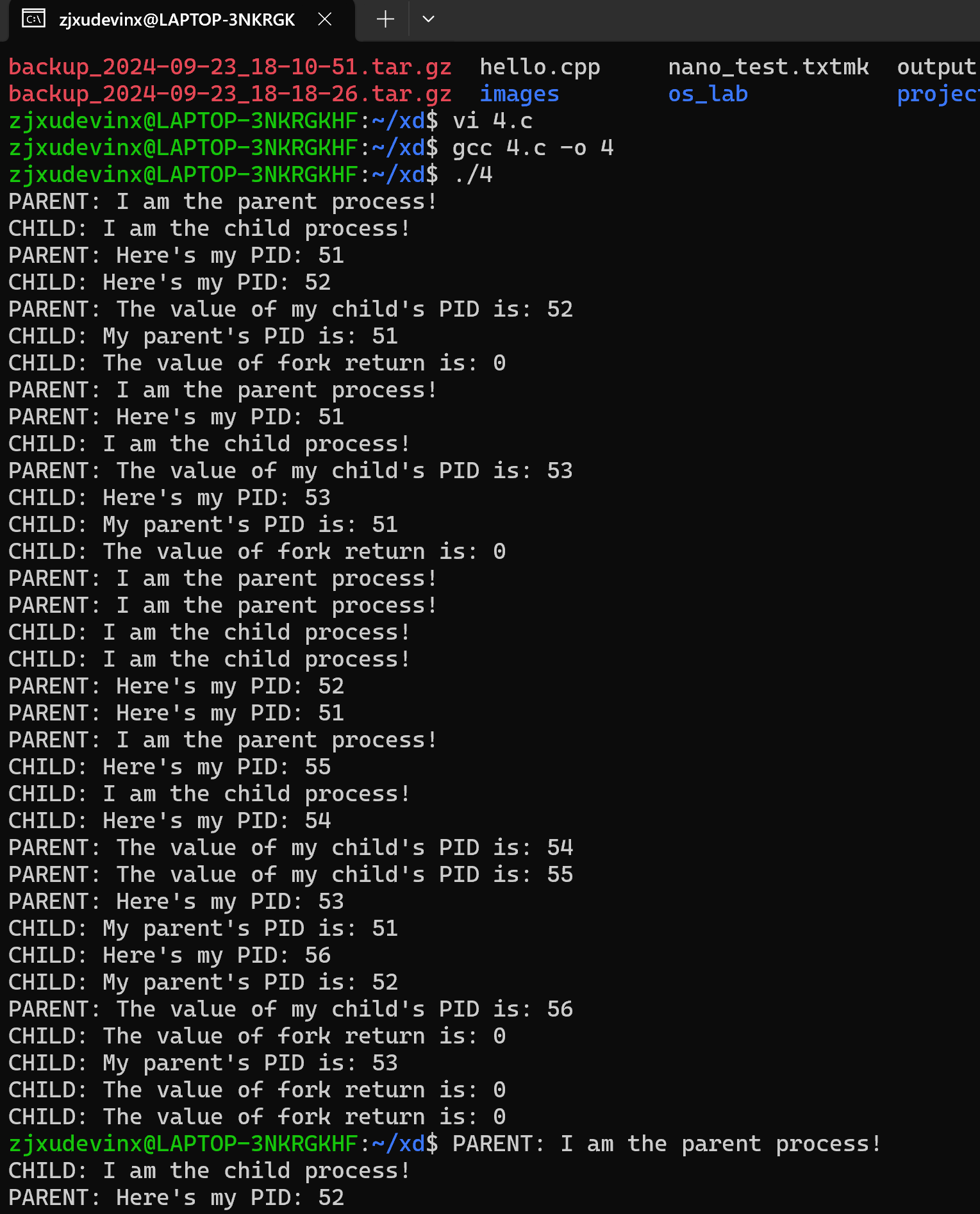
}

}

exit(0);

}

运行结果：



## 实验课作业第5回：

**实验名称：高级进程间通信与同步实验**

**实验目的：**

1. 掌握 fork() 创建多个子进程的高级用法。
2. 学习 pipe() 与共享内存（Shared Memory, shmget()）结合使用来进行父子进程间的复杂数据传递。
3. 理解进程间同步的基本机制，使用信号（signal()）和信号量（sem\_t）来处理并发与同步问题。
4. 掌握如何处理父子进程的异常状态（如信号中断、资源不足等），并设计容错机制。

**实验任务：**

1. **创建三个子进程**：
   * 子进程 1 和子进程 2：分别通过管道与父进程通信，并通过共享内存交换数据。
   * 子进程 3：作为信号处理器，接收父进程发出的自定义信号，执行相关任务（如打印日志或触发某些功能）。
2. **进程间通信与共享内存**：
   * 子进程 1 和子进程 2 需要通过管道传递消息，并使用共享内存交换计算结果。
   * 父进程协调多个子进程的任务，保证通信与数据的一致性。
3. **进程同步与信号处理**：
   * 通过信号量和信号机制同步父子进程。
   * 父进程可以发送信号给子进程，触发某些动作或终止子进程。
4. **父子进程的容错机制**：
   * 如果某个子进程运行异常或收到外部中断信号（如 SIGINT），父进程需要处理异常并重启该子进程，保证系统的稳定性。

代码：

#include <stdio.h>

#include <stdlib.h>

#include <unistd.h>

#include <string.h>

#include <sys/wait.h>

#include <sys/ipc.h>

#include <sys/shm.h>

#include <semaphore.h>

#include <signal.h>

#define SHM\_SIZE 1024

sem\_t sem;

// 信号处理函数

void handle\_signal(int sig) {

if (sig == SIGUSR1) {

printf("子进程 3 接收到 SIGUSR1 信号，开始执行日志任务...\n");

}

}

// 创建子进程 1

void child1(int pipefd[2], int shm\_id) {

close(pipefd[0]); // 关闭读端

char msg[] = "Message from Child 1";

write(pipefd[1], msg, strlen(msg) + 1); // 向管道写入消息

// 访问共享内存

char \*shared\_memory = (char \*)shmat(shm\_id, NULL, 0);

sem\_wait(&sem); // 等待信号量

strcpy(shared\_memory, "Shared data from Child 1");

sem\_post(&sem); // 释放信号量

close(pipefd[1]);

exit(0);

}

// 创建子进程 2

void child2(int pipefd[2], int shm\_id) {

close(pipefd[1]); // 关闭写端

char buf[128];

read(pipefd[0], buf, sizeof(buf)); // 读取管道中的消息

printf("子进程 2 接收到消息: %s\n", buf);

// 访问共享内存

char \*shared\_memory = (char \*)shmat(shm\_id, NULL, 0);

sem\_wait(&sem); // 等待信号量

printf("子进程 2 从共享内存读取数据: %s\n", shared\_memory);

sem\_post(&sem); // 释放信号量

close(pipefd[0]);

exit(0);

}

int main() {

int pipefd[2];

pid\_t pid1, pid2, pid3;

int shm\_id;

// 创建管道

if (pipe(pipefd) == -1) {

perror("pipe");

exit(1);

}

// 创建共享内存

shm\_id = shmget(IPC\_PRIVATE, SHM\_SIZE, IPC\_CREAT | 0666);

if (shm\_id == -1) {

perror("shmget");

exit(1);

}

// 初始化信号量

sem\_init(&sem, 1, 1);

// 创建子进程 1

if ((pid1 = fork()) == 0) {

child1(pipefd, shm\_id);

}

// 创建子进程 2

if ((pid2 = fork()) == 0) {

child2(pipefd, shm\_id);

}

// 创建子进程 3 处理信号

if ((pid3 = fork()) == 0) {

signal(SIGUSR1, handle\_signal); // 设置信号处理函数

sleep(1);

printf(执行完毕\n”);

}vi 5

// 父进程：等待子进程执行完毕并发送信号

sleep(1);

printf("父进程发送 SIGUSR1 信号给子进程 3\n");

kill(pid3, SIGUSR1); // 发送信号

wait(NULL); // 等待子进程结束

wait(NULL);

wait(NULL);

// 销毁信号量和共享内存

sem\_destroy(&sem);

shmctl(shm\_id, IPC\_RMID, NULL); // 删除共享内存

return 0;

}结果如下：



## 实验课作业第6回：

编写C程序，使用linux上的ipc机制，完成“石头剪刀布”的游戏.

#include <time.h>

#include <unistd.h>

#include <sys/types.h>

#include <sys/wait.h>

#include <sys/ipc.h>

#include <sys/msg.h>

#include <stdio.h>

#include <stdlib.h>

struct Game{

int Round;

int Type;

};

void result\_send(int num){

struct Game game;

game.Type = 1;

game.Round = rand() % 3;

msgsnd(num, &game, sizeof(int), 0);

}

int result\_announce(int a, int b){

if ((a + 1 == b) || (a - 3 == b))

return -1;

else if (a == b)

return 0;

else

return 1;

}

void PrintResult(int \*result\_list, int len){

int count\_A = 0;

int count\_B = 0;

int count\_C = 0;

for (int i = 0; i < len; i++){

switch(result\_list[i]){

case -1:{

count\_A++;

printf("NO.%d:A win\n", i + 1);

break;

}

case 0:{

count\_C++;

printf("NO.%d:end in a draw\n", i + 1);

break;

}

case 1:{

count\_B++;

printf("NO.%d:B win\n", i + 1);

break;

}

}

}

printf("\nThe final result is A win: %d\nB win:%d \n end in a draw %d\n",count\_A, count\_B, count\_C);

}

int main(){

int times;

int key1 = 1234, key2 = 5678;

int \*result\_list;

pid\_t pid1, pid2;

int msgid1, msgid2;

msgid1 = msgget(key1, IPC\_CREAT | 0666);

msgid2 = msgget(key2, IPC\_CREAT | 0666);

printf("Game start, please input rounds:");

scanf("%d", &times);

result\_list = (int\*)malloc(times\*sizeof(int));

for (int i = 0; i < times; i++){

pid1 = fork();

if (pid1 == 0){

srand((unsigned)time(0) \* 3000);

result\_send(msgid1);

exit(-1);

}

pid2 = fork();

if (pid2 == 0){

srand((unsigned)time(NULL) \* i);

result\_send(msgid2);

exit(-1);

}

if (pid1 < 0 || pid2 < 0){

exit(-1);

}

else {

wait(NULL);

wait(NULL);

struct Game game1, game2;

msgrcv(msgid1, &game1, sizeof(game1) - sizeof(int), 0, 0);

msgrcv(msgid2, &game2, sizeof(game2) - sizeof(int), 0, 0);

int j = result\_announce(game1.Round, game2.Round);

result\_list[i] = j;

}

}

PrintResult(result\_list, times);

exit(EXIT\_SUCCESS);

}

result\_send: 生成随机的出拳结果（0、1或2）并发送到指定的消息队列。 result\_announce: 根据两个玩家的出拳结果判断胜负。

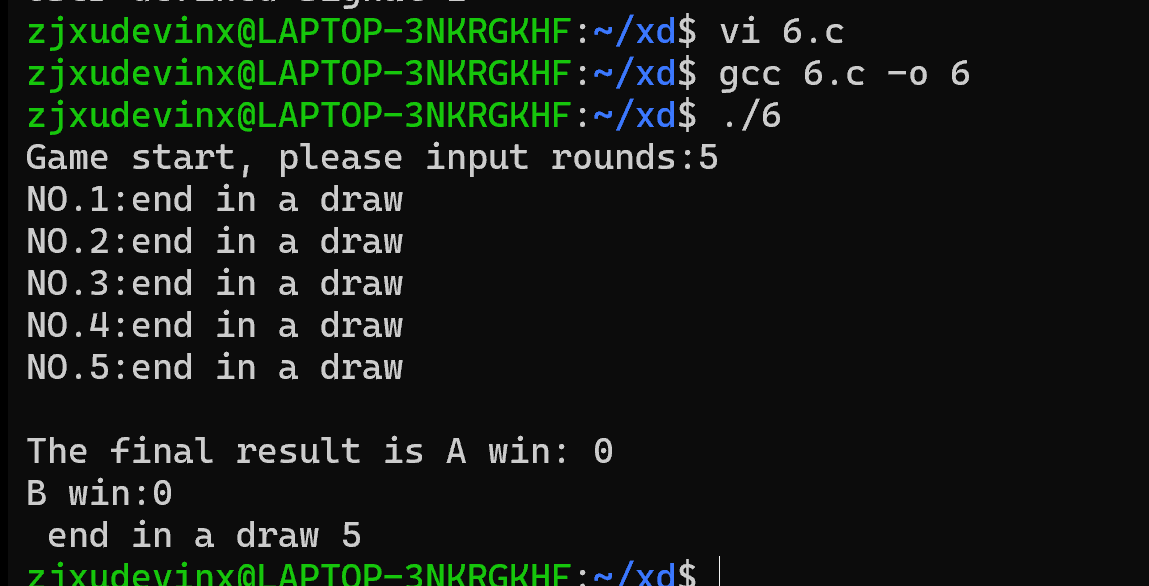
主函数逻辑：

a. 初始化： 设置初始回合数、消息队列的键值。 创建两个消息队列（msgid1和msgid2）。 提示用户输入回合数。

b. 主循环（每轮比赛）： 使用 fork() 创建两个子进程，分别代表两个玩家。 每个子进程： 重新设置随机数种子。 调用 result\_send 生成随机出拳并发送到对应的消息队列。 退出子进程。 父进程： 等待两个子进程结束。 从两个消息队列接收子进程发送的出拳结果。 调用 result\_announce 判断胜负，并将结果存入 result\_list。

c. 结果输出： 遍历 result\_list，打印每轮比赛的结果（平局、A胜或B胜）。

d. 清理： 删除创建的消息队列。 这个程序模拟了一个简单的猜拳游戏，使用进程间通信（IPC）中的消息队列来实现两个玩家的交互。每轮比赛都会创建新的子进程来代表玩家，这样可以确保每次出拳都是独立的随机事件。程序最后会统计并输出所有回合的结果。



## 实验课作业第7回：

1. 实验内容

编写 C 程序，使用 Linux 操作系统中的信号量机制模拟解决经典的进程同步问题：生

产者-消费者问题。假设有一个生产者和一个消费者，缓冲区可以存放产品，生产者不断生

产产品并存入缓冲区，消费者不断从缓冲区中取出产品并消费。

1. 实验目的

（1）加强对进程同步和互斥的理解，学会使用信号量解决资源共享问题。

（2）熟悉 Linux 进程同步原语。

（3）掌握信号量 wait/signal 原语的使用方法，理解信号量的定义、赋初值及 wait/signal

操作。

1. 实验代码

#include <stdio.h>

#include <stdlib.h>

#include <pthread.h>

#include <semaphore.h>

#include <unistd.h> // for sleep()

#define BUFFER\_SIZE 20

#define BATCH\_SIZE 5 // 每次生产的批次数目

// 全局变量

int buffer[BUFFER\_SIZE];

int count = 0;

int in = 0, out = 0;

// 信号量和互斥锁

sem\_t empty;

sem\_t full;

pthread\_mutex\_t mutex;

// 生产者函数

void \*producer(void \*arg) {

int items[BATCH\_SIZE];

while (1) {

// 准备一批产品

for (int i = 0; i < BATCH\_SIZE; ++i) {

items[i] = rand() % 100; // 模拟生产一个随机产品

}

// 等待有足够的空位

for (int i = 0; i < BATCH\_SIZE; ++i) {

sem\_wait(&empty);

}

pthread\_mutex\_lock(&mutex); // 锁定缓冲区

// 插入一批产品

for (int i = 0; i < BATCH\_SIZE; ++i) {

if (count < BUFFER\_SIZE) { // 防止缓冲区溢出

buffer[in] = items[i];

in = (in + 1) % BUFFER\_SIZE;

count++;

printf("生产者生产了: %d\n", items[i]);

} else {

break; // 如果缓冲区已满，停止插入

}

}

pthread\_mutex\_unlock(&mutex); // 解锁缓冲区

for (int i = 0; i < BATCH\_SIZE; ++i) {

sem\_post(&full); // 增加产品数量

}

sleep(rand() % 3 + 1); // 模拟生产时间

}

return NULL;

}

// 消费者函数

void \*consumer(void \*arg) {

int items[BATCH\_SIZE];

while (1) {

// 等待有足够的产品可取

for (int i = 0; i < BATCH\_SIZE; ++i) {

sem\_wait(&full);

}

pthread\_mutex\_lock(&mutex); // 锁定缓冲区

// 取出一批产品

for (int i = 0; i < BATCH\_SIZE; ++i) {

if (count > 0) { // 防止缓冲区欠载

items[i] = buffer[out];

out = (out + 1) % BUFFER\_SIZE;

count--;

printf("消费者消费了: %d\n", items[i]);

} else {

break; // 如果缓冲区为空，停止取出

}

}

pthread\_mutex\_unlock(&mutex); // 解锁缓冲区

for (int i = 0; i < BATCH\_SIZE; ++i) {

sem\_post(&empty); // 增加空位数量

}

sleep(rand() % 3 + 1); // 模拟消费时间

}

return NULL;

}

int main() {

pthread\_t prod\_thread, cons\_thread;

// 初始化信号量和互斥锁

sem\_init(&empty, 0, BUFFER\_SIZE);

sem\_init(&full, 0, 0);

pthread\_mutex\_init(&mutex, NULL);

// 创建生产者和消费者线程

pthread\_create(&prod\_thread, NULL, producer, NULL);

pthread\_create(&cons\_thread, NULL, consumer, NULL);

// 等待线程结束（这里不会真正退出，因为线程是无限循环）

pthread\_join(prod\_thread, NULL);

pthread\_join(cons\_thread, NULL);

// 清理资源（理论上应该在这里销毁信号量和互斥锁，但由于线程不会结束，这步不会执行）

sem\_destroy(&empty);

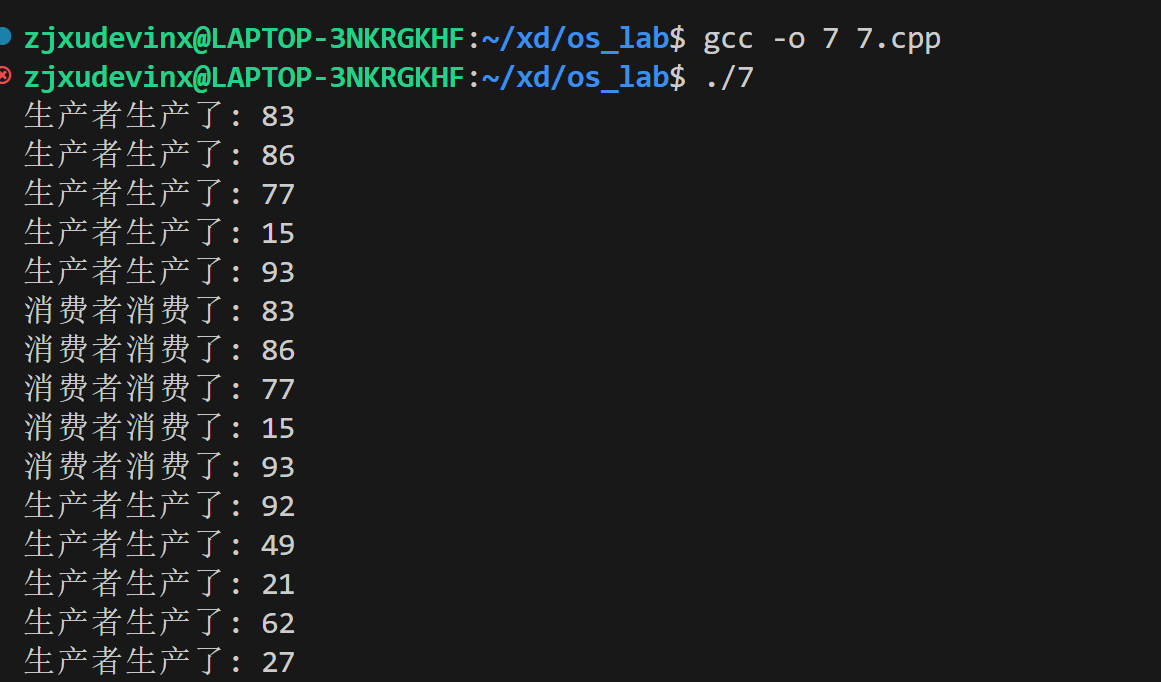
sem\_destroy(&full);

pthread\_mutex\_destroy(&mutex);

return 0;

}

1. 实验结果



**修改方案**

**批量生产和消费**：

* 1. 生产者每次生成 BATCH\_SIZE 个产品，并尝试将它们全部放入缓冲区。
  2. 消费者每次从缓冲区取出 BATCH\_SIZE 个产品进行消费。

**防止缓冲区溢出和欠载**：

* 1. 在批量插入时，检查缓冲区是否还有足够的空间；如果没有，则停止插入。
  2. 在批量取出时，检查缓冲区是否还有足够的产品；如果没有，则停止取出。

**信号量调整**：

* 1. 在批量操作前后，相应地调整 empty 和 full 信号量的值，以正确反映缓冲区的状态变化。

## 实验课作业第8回：

1. 实验内容

生产者消费者问题的扩展

1. 实验目的

在上节实验内容的基础上，按照下列要求修改逻辑功能，编码并且测试：

(1)存在多个生产者(P>1)

(2)存在多个消费者(C>1)

(3)一次放入产品的个数为Np

Np=1

(4)一次放入产品的个数为Nc

Nc=1

1. 实验代码

#include <stdio.h>

#include <stdlib.h>

#include <pthread.h>

#include <semaphore.h>

#include <unistd.h> // for sleep()

#include <signal.h> // for signal handling

#define BUFFER\_SIZE 20

#define NUM\_PRODUCERS 3 // 生产者的数量

#define NUM\_CONSUMERS 3 // 消费者的数量

#define Np 1 // 每次放入产品的个数

#define Nc 1 // 每次取出产品的个数

// 全局变量

int buffer[BUFFER\_SIZE];

int count = 0;

int in = 0, out = 0;

volatile sig\_atomic\_t running = 1; // 运行标志

// 信号量和互斥锁

sem\_t empty;

sem\_t full;

pthread\_mutex\_t mutex;

// 处理 SIGINT 信号

void handle\_sigint(int sig) {

if (sig == SIGINT) {

printf("收到中断信号，准备退出...\n");

running = 0;

}

}

// 生产者函数

void \*producer(void \*arg) {

while (running) {

int item = rand() % 100; // 模拟生产一个随机产品

sem\_wait(&empty); // 等待有空位

pthread\_mutex\_lock(&mutex); // 锁定缓冲区

// 插入产品

buffer[in] = item;

in = (in + 1) % BUFFER\_SIZE;

count++;

printf("生产者(%ld)生产了: %d\n", pthread\_self(), item);

pthread\_mutex\_unlock(&mutex); // 解锁缓冲区

sem\_post(&full); // 增加产品数量

sleep(rand() % 3 + 1); // 模拟生产时间

}

return NULL;

}

// 消费者函数

void \*consumer(void \*arg) {

while (running) {

sem\_wait(&full); // 等待有产品可取

pthread\_mutex\_lock(&mutex); // 锁定缓冲区

// 取出产品

int item = buffer[out];

out = (out + 1) % BUFFER\_SIZE;

count--;

printf("消费者(%ld)消费了: %d\n", pthread\_self(), item);

pthread\_mutex\_unlock(&mutex); // 解锁缓冲区

sem\_post(&empty); // 增加空位数量

sleep(rand() % 3 + 1); // 模拟消费时间

}

return NULL;

}

int main() {

pthread\_t prod\_threads[NUM\_PRODUCERS], cons\_threads[NUM\_CONSUMERS];

// 设置信号处理器

signal(SIGINT, handle\_sigint);

// 初始化信号量和互斥锁

sem\_init(&empty, 0, BUFFER\_SIZE);

sem\_init(&full, 0, 0);

pthread\_mutex\_init(&mutex, NULL);

// 创建多个生产者和消费者线程

for (int i = 0; i < NUM\_PRODUCERS; ++i) {

pthread\_create(&prod\_threads[i], NULL, producer, NULL);

}

for (int i = 0; i < NUM\_CONSUMERS; ++i) {

pthread\_create(&cons\_threads[i], NULL, consumer, NULL);

}

// 等待所有线程结束

for (int i = 0; i < NUM\_PRODUCERS; ++i) {

pthread\_join(prod\_threads[i], NULL);

}

for (int i = 0; i < NUM\_CONSUMERS; ++i) {

pthread\_join(cons\_threads[i], NULL);

}

// 清理资源

sem\_destroy(&empty);

sem\_destroy(&full);

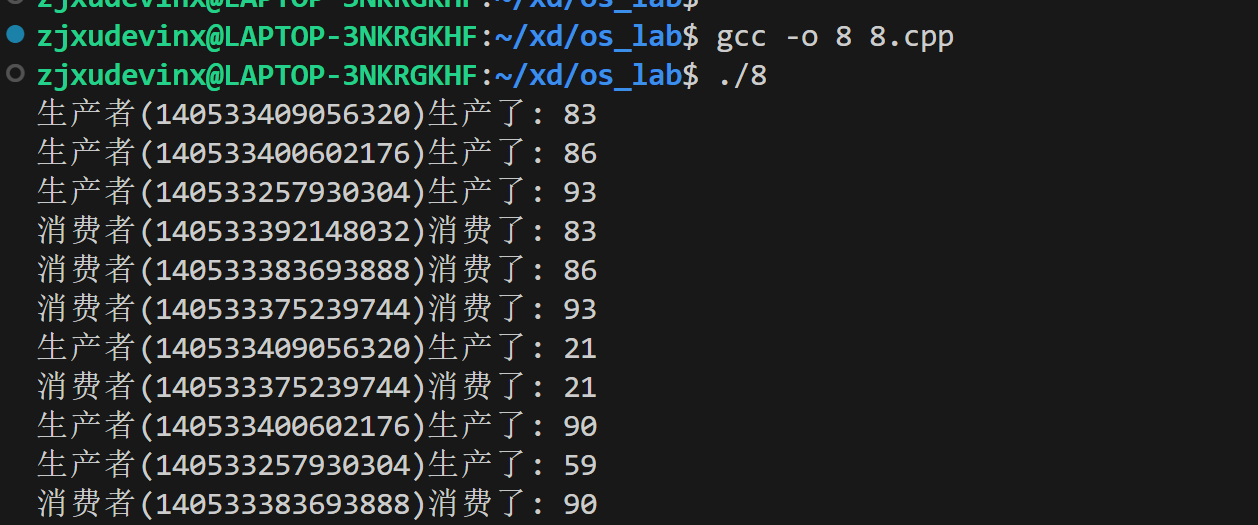
pthread\_mutex\_destroy(&mutex);

printf("程序已安全退出。\n");

return 0;

}

1. 实验结果



## 实验课作业第9回：

1. 实验目的：

掌握进程间死锁的相关概念；

掌握如何预防以及避免死锁的发生。

研究银行家算法的实现方法。

1. 实验内容：

为实现银行家算法，每个新进程在进入系统时，其都必须申明在运行过程中可能需要每种资源类型的最大单元数目，该数目不应超过系统所拥有的资源总数。当进程请求一组资源时，系统须首先确定是否有足够的资源可分配给该进程。若有，则进一步计算在将这些资源分配给该进程后，系统是否会处于不安全状态。如果不会，则将资源分配给该进程，否则让该进程等待。

该实验要求编程实现银行家算法，具体要求如下：

模拟一个银行家算法： 设置数据结构、安全性算法  
初始化时让系统拥有一定的资源  
用键盘输入的方式申请资源  
如果预分配后，系统处于安全状态，则修改系统的资源分配情况  
如果预分配后，系统处于不安全状态，则提示不能满足请求

1. 实验代码

#include <stdio.h>

#include <stdlib.h>

#include <stdbool.h> // 使用标准的 bool 类型

#include <stdarg.h> // 支持可变参数列表

#define MAXN 100 // 最大进程数

#define MAXM 100 // 最大资源种类数

int N; // 进程个数

int M; // 资源种类个数

int Avaliable[MAXM]; // 可利用资源数

int Max[MAXN][MAXM]; // 进程对资源的最大需求

int Allocation[MAXN][MAXM]; // 已分配资源

int Need[MAXN][MAXM]; // 需求矩阵

void Initialize();

bool Safe();

void Bank();

void PrintMatrix(int matrix[][MAXM], int rows, int cols, const char \*name);

void Log(const char \*format, ...);

int main()

{

Initialize();

if (!Safe()) {

printf("初始状态系统不安全！\n");

return EXIT\_FAILURE;

}

char ch;

do {

printf("是否继续发送请求 (y/n)? ");

fflush(stdout); // 确保输出缓冲区被清空

scanf(" %c", &ch); // 注意前面的空格用于跳过空白字符

if (ch == 'y' || ch == 'Y') {

Bank();

} else if (ch != 'n' && ch != 'N') {

printf("无效的选择，请输入 y 或 n。\n");

}

} while (ch != 'n' && ch != 'N');

return EXIT\_SUCCESS;

}

void Initialize()

{

int i, j;

printf("请输入进程个数: ");

if (scanf("%d", &N) != 1 || N <= 0 || N > MAXN) {

Log("错误：无效的进程数量。\n");

exit(EXIT\_FAILURE);

}

printf("请输入资源种类个数: ");

if (scanf("%d", &M) != 1 || M <= 0 || M > MAXM) {

Log("错误：无效的资源种类数量。\n");

exit(EXIT\_FAILURE);

}

for (i = 0; i < M; i++) {

printf("请输入资源 %d 的可利用数量: ", i + 1);

if (scanf("%d", &Avaliable[i]) != 1 || Avaliable[i] < 0) {

Log("错误：无效的资源数量。\n");

exit(EXIT\_FAILURE);

}

}

printf("请输入最大资源需求矩阵 (%d \* %d):\n", N, M);

for (i = 0; i < N; i++) {

for (j = 0; j < M; j++) {

if (scanf("%d", &Max[i][j]) != 1 || Max[i][j] < 0) {

Log("错误：无效的最大资源需求值。\n");

exit(EXIT\_FAILURE);

}

}

}

printf("请输入已分配资源矩阵 (%d \* %d):\n", N, M);

for (i = 0; i < N; i++) {

for (j = 0; j < M; j++) {

if (scanf("%d", &Allocation[i][j]) != 1 || Allocation[i][j] < 0) {

Log("错误：无效的已分配资源值。\n");

exit(EXIT\_FAILURE);

}

if (Allocation[i][j] > Max[i][j]) {

Log("错误：已经分配的资源超过了最大需求（进程 %d 资源种类 %d）。\n", i, j);

exit(EXIT\_FAILURE);

}

Need[i][j] = Max[i][j] - Allocation[i][j];

}

for (j = 0; j < M; j++) {

Avaliable[j] -= Allocation[i][j];

}

}

}

bool Safe()

{

int Work[MAXM];

bool Finish[MAXN] = {false};

int Security[MAXN];

int count = 0;

// 初始化工作向量为可用资源

for (int i = 0; i < M; i++)

Work[i] = Avaliable[i];

// 寻找安全序列

for (count = 0; count < N; ) {

bool found = false;

for (int i = 0; i < N; i++) {

if (!Finish[i]) {

bool can\_finish = true;

for (int j = 0; j < M; j++) {

if (Need[i][j] > Work[j]) {

can\_finish = false;

break;

}

}

if (can\_finish) {

for (int j = 0; j < M; j++) {

Work[j] += Allocation[i][j];

}

Finish[i] = true;

Security[count++] = i;

found = true;

break;

}

}

}

if (!found) {

Log("系统不安全!\n");

return false;

}

}

Log("系统是安全的!\n");

Log("进程结束的顺序: ");

for (int i = 0; i < N; i++) {

Log("%d", Security[i]);

if (i < N - 1)

Log("->");

}

Log("\n");

return true;

}

void Bank()

{

int process\_id;

int Request[MAXM];

printf("请选择进程的ID (0-%d): ", N - 1);

if (scanf("%d", &process\_id) != 1 || process\_id < 0 || process\_id >= N) {

Log("错误：无效的进程ID。\n");

return;

}

printf("请输入请求的资源量(%d): ", M);

for (int j = 0; j < M; j++) {

if (scanf("%d", &Request[j]) != 1 || Request[j] < 0) {

Log("错误：无效的资源请求值。\n");

return;

}

if (Request[j] > Need[process\_id][j]) {

Log("错误：请求大于最大需要的资源量！\n");

return;

}

if (Request[j] > Avaliable[j]) {

Log("错误：请求大于可用资源，进程正在等待！\n");

return;

}

// 检查请求后是否会超过最大需求

if (Allocation[process\_id][j] + Request[j] > Max[process\_id][j]) {

Log("错误：请求后将导致已分配资源超过最大需求（进程 %d 资源种类 %d）。\n", process\_id, j);

return;

}

}

// 假设分配资源

for (int j = 0; j < M; j++) {

Avaliable[j] -= Request[j];

Allocation[process\_id][j] += Request[j];

Need[process\_id][j] -= Request[j];

}

if (!Safe()) {

// 如果预分配后系统处于不安全状态，则回滚

for (int j = 0; j < M; j++) {

Avaliable[j] += Request[j];

Allocation[process\_id][j] -= Request[j];

Need[process\_id][j] += Request[j];

}

Log("请求被拒绝，系统将保持原状。\n");

} else {

Log("请求成功，资源已被分配。\n");

}

}

// 打印矩阵（用于调试）

void PrintMatrix(int matrix[][MAXM], int rows, int cols, const char \*name)

{

Log("%s Matrix:\n", name);

for (int i = 0; i < rows; i++) {

for (int j = 0; j < cols; j++) {

Log("%d ", matrix[i][j]);

}

Log("\n");

}

}

// 日志函数

void Log(const char \*format, ...)

{

va\_list args;

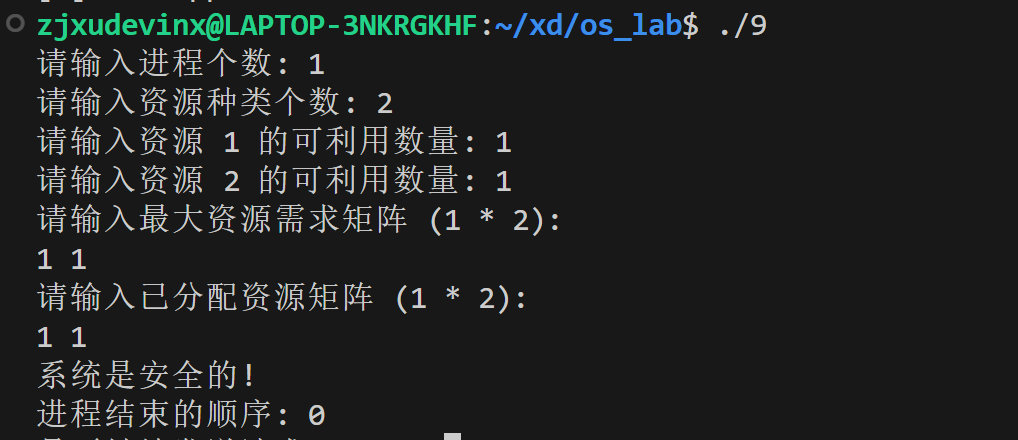
va\_start(args, format);

vfprintf(stderr, format, args); // 输出到 stderr 以便与正常输出区分

va\_end(args);

}

实验结果



## 实验课作业第10回：

1. 实验内容：动态分区分配方式的模拟
2. 实验目标：

（1）掌握动态分区分配方式使用的数据结构和分配算法（首次/最佳

/最坏适应算法）。

（2）进一步加深对动态分区分配管理方式及其实现过程的理解。

1. 实验要求：1.理解算法啊的核心概念和思想；
2. 画出算法流程图；

3.回答实验思考题

1. 实验代码：

#include<stdio.h>

#include<stdlib.h>

#include<string.h>

#define MAX 640

struct node //定义分区

{

int address, size;

struct node \*next;

};

typedef struct node RECT;

/\*-----函数定义-------\*/

RECT \*assignment(RECT \*head,int application); //分配分区

//针对首次适应算法回收分区

void firstfit(RECT \*head,RECT \*heada,RECT \*back1);

void bestfit(RECT \*head,RECT \*back1); //针对最佳适应算法回收分区，待扩充

int backcheck(RECT \*head,RECT \*back1); //合法性检查

void print(RECT \*head); //输出已分配分区表或空闲分区

/\*-----变量定义-------\*/

RECT \*head,\*heada,\*back,\*assign1,\*p;

int application1,maxblocknum;

char way; //用于定义分配方式：首次适应、最佳适应、最坏适应。目前未使用

int main()

{

char choose;

int check;

RECT \*allocated;

head=malloc(sizeof(RECT)); //建立空闲分区表的初始状态

p=malloc(sizeof(RECT));

head->size=MAX;

head->address=0;

head->next=p;

maxblocknum=1;

p->size=MAX;

p->address=0;

p->next=NULL;

print(head); //输出空闲分区表的初始状态

//printf("Enter the allocation way (best or first (b/f))\n");

//scanf("%c",&way);

way='f';

heada=malloc(sizeof(RECT)); //建立已分配分区表的初始状态

heada->size=0;

heada->address=0;

heada->next=NULL;

//print(heada); //输出空闲分区表的初始状态

do

{

printf("Enter the allocate or reclaim (a/r),or press other key to

exit.\n");

scanf(" %c",&choose); //选择分配或回收

if(tolower(choose)=='a') //a 为分配

{

printf("Input application:\n");

scanf("%d",&application1); //输入申请的空间大小

assign1=assignment(head,application1); //调用分配函数以分配内存

if (assign1->address==-1) //分配不成功

printf("Too large application! Allocation fails! \n\n");

else //分配成功

printf("Allocation Success! ADDRESS=%5d\n",assign1->address);

printf("\n\*\*\*\*\*\*\*\*\*Unallocated Table\*\*\*\*\*\*\*\*\*\*\*\*\*\n");

print(head); //输出

printf("\n\*\*\*\*\*\*\*\*\*Allocated Table\*\*\* \*\*\*\*\*\*\*\*\*\*\n");

print(heada);

}

else if (tolower(choose)=='r') //回收内存

{

back=malloc(sizeof(RECT));

printf("Input address and Size:\n");

scanf("%d%d",&back->address,&back->size);//输入回收地址和大小

check=backcheck(head,back);

if (check==1)

{

if(tolower(way)=='f')

firstfit(head,heada,back); //首次适应算法回收

printf("\n\*\*\*\*\*\*\*\*\*Unallocated Table\*\*\*\*\*\*\*\*\*\*\*\*\*\n");

print(head); //输出

printf("\n\*\*\*\*\*\*\*\*\*Allocated Table\*\*\* \*\*\*\*\*\*\*\*\*\*\n");

print(heada);

}

}

}while(tolower(choose)=='a'||tolower(choose)=='r');

exit(0);

} //main() end.

/\*-------内存分配函数-------\*/

RECT \*assignment(RECT \*head,int application)

{

RECT \*after,\*before,\*assign;

assign=malloc(sizeof(RECT)); //申请分配空间

assign->size=application;

assign->next=NULL;

if(application>head->size || application<0)

assign->address=-1; //申请无效

else

{

before=head;

after=head->next;

while(after->size < application) //遍历链表，查找合适的节点

{

before=before->next;

after=after->next;

}

if(after->size==application) //若节点大小等于申请大小，则完全分配

{

if(after->size==head->size) maxblocknum--;

before->next=after->next;

assign->address=after->address;

free(after);

}

else

{

if(after->size==head->size) maxblocknum--;

after->size=after->size-application; //大于申请空间时，截取相应大小并分配

assign->address=after->address+after->size;

}

if (maxblocknum==0) //修改最大数和头节点

{

before=head;

head->size=0;

maxblocknum=1;

while(before!=NULL)

{

if(before->size > head->size)

{

head->size=before->size;

maxblocknum=1;

}

else if(before->size==head->size)

maxblocknum++;

before=before->next;

}

}

}

assign1=assign;

//修改已分配分区表，添加节点

after=heada;

while(after->next!=NULL)

after=after->next;

after->next=assign;

heada->size++;

return assign1; //返回分配给用户的地址

}

/\*------------------首次适应算法------------\*/

void firstfit(RECT \*head,RECT \*heada,RECT \*back1)

{

RECT \*before,\*after,\*back2;

int insert,del;

back2=malloc(sizeof(RECT));

back2->address=back1->address;

back2->size=back1->size;

back2->next=back1->next;

before=head;

after=head->next;

insert=0;

while(!insert) //将回收区插入空闲分区表

{

if((after==NULL)||((back1->address<=after->address)&&(back1->

address>=before->address)))

{

before->next=back1;

back1->next=after;

insert=1;

}

else

{

before=before->next;

after=after->next;

}

}

if(back1->address==before->address+before->size) //与上一内存块合并

{

before->size=before->size+back1->size;

before->next=back1->next;

free(back1);

back1=before;

}

if((after!=NULL) &&(after->address==back1->address+back1->size))

//与下一内存块合并

{

back1->size=back1->size+after->size;

back1->next=after->next;

free(after);

}

if(head->size<back1->size) //修改最大块值和最大块个数

{

head->size=back1->size;

maxblocknum=1;

}

else

if(head->size==back1->size) maxblocknum++;

//修改已分配分区表，删除相应节点

before=heada;

after=heada->next;

del=0;

while(!del||after!=NULL) //将回收区从已分配分区表中删除

{

if((after->address==back2->address)&&(after->size==back2->size))

{

before->next=after->next;

free(after);

del=1;

}

else

{

before=before->next;

after=after->next;

}

}

heada->size--;

}

/\*-----------------打印输出链表--------------\*/

void print(RECT \*output)

{

RECT \*before;

int index;

before=output->next;

index=0;

if(output->next==NULL)

printf("NO part for print!\n");

else

{

printf("index\*\*\*\*address\*\*\*\*end\*\*\*\*\*size\*\*\*\* \n");

while(before!=NULL)

{

printf("------------------------------------\n");

printf("%-9d%- 9d%- 9d%- 9d\n", index, before->address, before->

address+ before->size-1,before->size);

printf("-------------------------------------\n");

index++;

before=before->next;

}

}

}

/\*检查回收块的合法性，back1 为要回收节点的地址\*/

int backcheck(RECT \*head,RECT \*back1)

{

RECT \*before;

int check=1;

if(back1->address<0 || back1->size<0) check=0; //地址和大小不能为负数

before=head->next;

while((before!=NULL)&&check) //地址不能和空闲分区表中的节点重叠

if(((back1->address<before->address)&&(back1->address+back1->size>before

->address))||((back1->address>=before->address)&&(back1->address<before

->address+before->size)))

check=0;

else

before=before->next;

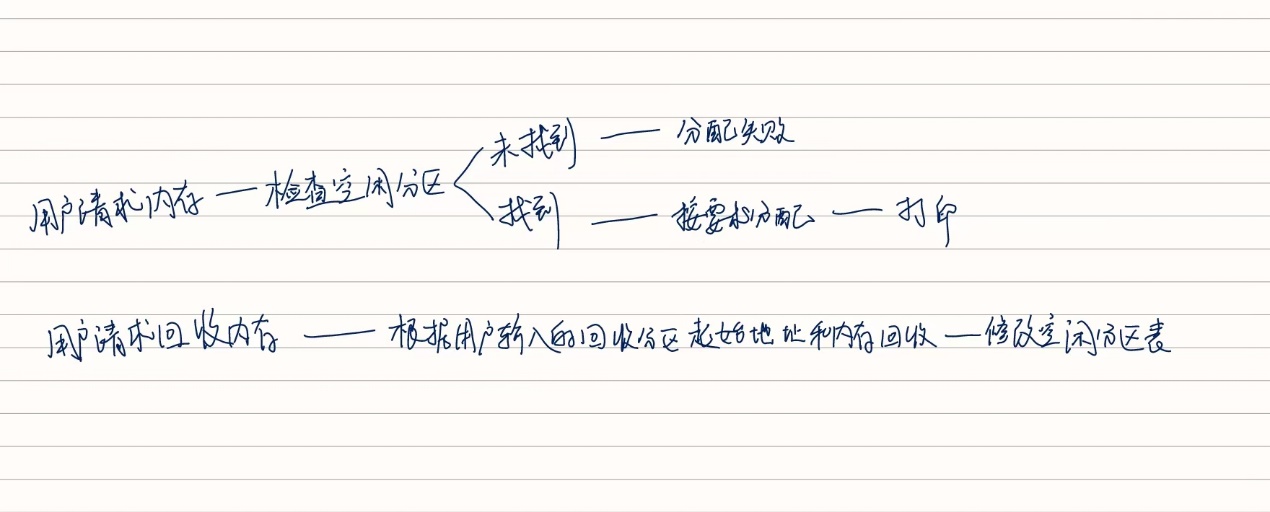
if(check==0) printf("Error input!\n");

return check;

}

1. 实验结果：

流程图：



实验思考：

**(1) 修改程序，使分配内存时从低地址开始**

修改 assignment 函数时，保留原有的遍历逻辑，确保内存分配时从链表头部开始。我们不改变遍历顺序，只需确保从链表头部开始分配内存，并且一旦找到合适的空闲块，就立即进行分配，避免跳过较小的空闲块。这样在分配内存时，首先检查低地址的空闲块，优先分配小的空闲块，保证内存的利用更加高效。

**(2) 完善程序，实现基于最佳适应算法和最坏适应算法的内存分配与回收**

**最佳适应算法：**

最佳适应算法要求遍历空闲分区链表，选择最小的能够容纳请求内存的空闲块。当遍历空闲内存链表时，我们需要找出所有能够满足申请内存的空闲块，并从中选择最合适的块，即与请求内存大小最接近的块。一旦找到合适的块，进行内存分配：

* 如果找到的块大小与请求内存大小完全匹配，则直接分配并移除该块。
* 如果块比请求内存大，则将该块分割，分配所需内存，剩余部分继续保留为新的空闲块。

在内存分配完成后，及时更新空闲分区链表和已分配分区链表，确保内存状态的正确管理。

**最坏适应算法：**

最坏适应算法要求遍历空闲分区链表，选择一个最大的空闲块进行内存分配。在遍历过程中，找到所有能够满足请求的空闲块，并从中选择最大的空闲块。一旦找到合适的块，分配内存的步骤与最佳适应算法类似：

* 如果块的大小与请求内存完全匹配，则直接分配并移除该块。
* 如果块的大小大于请求内存，则分割空闲块，分配所需的内存，剩余部分继续保留为新的空闲块。

## 实验课作业第11回：

1. 实验内容：

设计一个虚拟存储区和一个内存工作区，并使用下述常用页面置换算法计算访问命中率。

（1）先进先出（first in first out，FIFO）算法。

（2）最近最久未使用（least recently used，LRU）算法。

（3）最优（optimal，OPT）算法。

1. 实验目标：
2. 理解虚拟内存管理的原理和技术
3. 掌握请求分页存储管理的常用理论-页面置换算法；
4. 理解请求分页中的按需调页机制
5. 实验要求：
6. 有完整的实验流程
7. 三种算法至少实现两种
8. 实验思考题第三题
9. 实验代码：

#include <stdio.h>

#include <stdlib.h>

#include <time.h>

#define TOTAL\_INSTRUCTIONS 320

#define PAGE\_SIZE 10 // 每页包含10条指令

#define TOTAL\_PAGES 32 // 总共有32页

// 页面置换算法模拟

// FIFO (先进先出)

int fifo(int pages[], int num\_pages, int reference\_string[], int num\_references) {

int page\_frame[num\_pages];

int page\_faults = 0;

int page\_in\_frame = 0;

for (int i = 0; i < num\_pages; i++) page\_frame[i] = -1; // 初始化页面框架

for (int i = 0; i < num\_references; i++) {

int page = reference\_string[i];

int found = 0;

// 检查页面是否在内存中

for (int j = 0; j < num\_pages; j++) {

if (page\_frame[j] == page) {

found = 1;

break;

}

}

if (!found) {

// 页面不在内存中，发生缺页

page\_frame[page\_in\_frame] = page;

page\_in\_frame = (page\_in\_frame + 1) % num\_pages;

page\_faults++;

}

}

return page\_faults;

}

// LRU (最近最久未使用)

int lru(int pages[], int num\_pages, int reference\_string[], int num\_references) {

int page\_frame[num\_pages];

int last\_used[num\_pages];

int page\_faults = 0;

for (int i = 0; i < num\_pages; i++) {

page\_frame[i] = -1;

last\_used[i] = -1;

}

for (int i = 0; i < num\_references; i++) {

int page = reference\_string[i];

int found = 0;

int lru\_index = -1;

for (int j = 0; j < num\_pages; j++) {

if (page\_frame[j] == page) {

found = 1;

last\_used[j] = i;

break;

}

}

if (!found) {

// 页面不在内存中，发生缺页

for (int j = 0; j < num\_pages; j++) {

if (page\_frame[j] == -1) {

page\_frame[j] = page;

last\_used[j] = i;

page\_faults++;

break;

}

// 找到最久未使用的页面

if (lru\_index == -1 || last\_used[j] < last\_used[lru\_index]) {

lru\_index = j;

}

}

// 替换最久未使用的页面

if (lru\_index != -1) {

page\_frame[lru\_index] = page;

last\_used[lru\_index] = i;

page\_faults++;

}

}

}

return page\_faults;

}

// OPT (最优算法)

int opt(int pages[], int num\_pages, int reference\_string[], int num\_references) {

int page\_frame[num\_pages];

int page\_faults = 0;

for (int i = 0; i < num\_pages; i++) page\_frame[i] = -1;

for (int i = 0; i < num\_references; i++) {

int page = reference\_string[i];

int found = 0;

for (int j = 0; j < num\_pages; j++) {

if (page\_frame[j] == page) {

found = 1;

break;

}

}

if (!found) {

// 页面不在内存中，发生缺页

if (page\_faults < num\_pages) {

page\_frame[page\_faults] = page;

} else {

// 找到最优的页面进行替换

int farthest = -1;

int replace\_index = -1;

for (int j = 0; j < num\_pages; j++) {

int next\_use = -1;

for (int k = i + 1; k < num\_references; k++) {

if (page\_frame[j] == reference\_string[k]) {

next\_use = k;

break;

}

}

if (next\_use == -1) {

replace\_index = j;

break;

}

if (next\_use > farthest) {

farthest = next\_use;

replace\_index = j;

}

}

page\_frame[replace\_index] = page;

}

page\_faults++;

}

}

return page\_faults;

}

// 计算命中率

void calculate\_hit\_rate(int page\_frame\_size) {

int reference\_string[TOTAL\_INSTRUCTIONS];

int page\_sequence[TOTAL\_INSTRUCTIONS / PAGE\_SIZE];

// 随机生成指令序列

srand(time(NULL));

for (int i = 0; i < TOTAL\_INSTRUCTIONS; i++) {

reference\_string[i] = rand() % TOTAL\_PAGES; // 页面序号

}

// 模拟页面置换算法

int page\_faults\_fifo = fifo(page\_sequence, page\_frame\_size, reference\_string, TOTAL\_INSTRUCTIONS);

int page\_faults\_lru = lru(page\_sequence, page\_frame\_size, reference\_string, TOTAL\_INSTRUCTIONS);

int page\_faults\_opt = opt(page\_sequence, page\_frame\_size, reference\_string, TOTAL\_INSTRUCTIONS);

int hit\_rate\_fifo = (TOTAL\_INSTRUCTIONS - page\_faults\_fifo) \* 100 / TOTAL\_INSTRUCTIONS;

int hit\_rate\_lru = (TOTAL\_INSTRUCTIONS - page\_faults\_lru) \* 100 / TOTAL\_INSTRUCTIONS;

int hit\_rate\_opt = (TOTAL\_INSTRUCTIONS - page\_faults\_opt) \* 100 / TOTAL\_INSTRUCTIONS;

printf("FIFO 命中率: %d%%\n", hit\_rate\_fifo);

printf("LRU 命中率: %d%%\n", hit\_rate\_lru);

printf("OPT 命中率: %d%%\n", hit\_rate\_opt);

}

int main() {

int page\_frame\_size;

printf("请输入内存容量（4 到 32 页之间）：");

scanf("%d", &page\_frame\_size);

if (page\_frame\_size < 4 || page\_frame\_size > 32) {

printf("内存容量必须在4到32页之间！\n");

return 1;

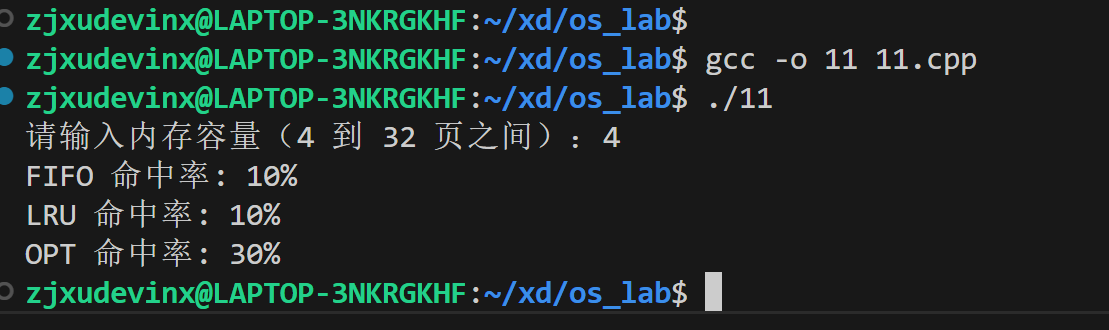
}

calculate\_hit\_rate(page\_frame\_size);

return 0;

}

1. 实验结果：



1. 实验思考：

（1）总结和比较：

表格

描述已自动生成

（2）结论：

**FIFO**：适用于较为简单的系统，但其性能较差，容易受到访问模式的影响，尤其是当程序访问方式较为随机时，性能会显著下降。

**LRU**：在大多数场景下，LRU表现良好，尤其适合具有局部性访问模式的程序。然而，其实现复杂，且开销较大。

**OPT**：理论上是最优的页面替换算法，但由于无法在实际应用中实现，它通常作为性能评估的基准或比较的参考。

## 实验课作业第12回：

**一、实验目标：熟练掌握操作系统中设备驱动程序的实现方法**

**二、实验目的：**

1. 掌握操作系统中设备驱动程序的实现方法，尤其是字符设备驱动的开发。
2. 学习如何通过Linux内核模块进行设备的操作，如打开、读取、写入和关闭操作。
3. 实现设备驱动的创新功能，增强驱动程序的可扩展性与健壮性，包括添加IOCTL支持、动态内存管理和同步机制。

**三、实验背景：**

设备驱动程序是操作系统与硬件设备之间的桥梁。它定义了应用程序与硬件交互的接口，允许操作系统通过一组函数与硬件设备进行通信。在Linux操作系统中，字符设备驱动通常是实现与外部设备的交互的重要方式之一。

传统的字符设备驱动程序包含基本的打开、读取、写入和关闭功能，能够处理用户空间应用程序对设备的操作请求。为了增加驱动程序的灵活性和扩展性，本实验中引入了更多的功能改进，如IOCTL操作支持、动态内存管理和互斥锁机制。

**四、实验内容与方法：**

1. 确定主设备号：

在Linux中，每个字符设备驱动程序都需要一个唯一的主设备号。

可以通过静态分配或动态分配的方式获取主设备号。

这里我选择通过register\_chrdev函数，我们动态地分配了主设备号，保证设备号的唯一性。

1. 定义file\_operations结构体：

file\_operations是Linux内核中用于字符设备驱动程序的结构体。

通过定义file\_operations结构体，我将设备驱动与内核的各种操作（如打开、读取、写入、关闭）进行绑定。

在这个实验中，我们的file\_operations结构体包括了设备驱动程序需要实现的各个函数：open、read、write、release和unlocked\_ioctl等。

驱动程序注册时，需要将file\_operations结构体的指针传递给内核。

1. 实现file\_operations结构体中的函数：

根据具体硬件设备的需求，实现open、read、write、release等函数。

这些函数负责处理用户空间的应用程序对硬件设备的操作请求。

1. 增加IOCTL命令支持：

为了使设备驱动能够接收更复杂的控制命令，我引入了IOCTL支持。

在该驱动程序中，增加了一个自定义的IOCTL命令MYDEV\_IOCTL\_SET\_MODE，用于设置设备模式或参数。

IOCTL操作可以在用户空间程序中使用，提供更加灵活的设备控制。

1. 动态内存分配：

传统的设备驱动程序通常使用静态缓冲区来存储数据。

在本实验中，我使用kmalloc函数为设备分配动态内存，并使用kfree函数在驱动卸载时释放内存。

这样能够减少静态缓冲区带来的内存限制，提高了驱动程序的灵活性。

1. 互斥锁机制：

为了防止多进程或多线程同时访问设备时出现竞态条件，我们为驱动程序添加了互斥锁（mutex）。

mutex确保了在同一时刻只有一个进程可以访问设备，避免了数据不一致的情况。

1. 驱动程序初始化与注销：

在驱动程序初始化时，通过register\_chrdev注册设备，并通过kmalloc动态分配内存。

注册成功后，内核会分配一个主设备号，并在/dev目录下创建相应的设备文件。

在驱动程序退出时，使用unregister\_chrdev注销设备，并使用kfree释放分配的内存。

**五、实验代码：**

#include <linux/init.h>

#include <linux/module.h>

#include <linux/kernel.h>

#include <linux/fs.h>

#include <linux/uaccess.h>

#include <linux/mutex.h>

#include <linux/ioctl.h>

#include <linux/slab.h>

#define DEVICE\_NAME "mychardev"

#define BUF\_LEN 1024

static int major;

static char \*kernel\_buf;

static int write\_ptr;

static struct mutex dev\_mutex;

// IOCTL命令定义

#define MYDEV\_IOCTL\_SET\_MODE \_IOW('M', 1, int)

static int my\_drv\_open(struct inode \*node, struct file \*file) {

printk(KERN\_INFO "Device opened\n");

write\_ptr = 0;

return 0;

}

static ssize\_t my\_drv\_read(struct file \*file, char \_\_user \*buf, size\_t size, loff\_t \*offset) {

int bytes\_read = 0;

if (\*offset > 0) {

return 0; // No-op if offset is non-zero

}

mutex\_lock(&dev\_mutex);

while (size && write\_ptr < BUF\_LEN) {

put\_user(kernel\_buf[write\_ptr++], buf++);

size--;

bytes\_read++;

}

mutex\_unlock(&dev\_mutex);

\*offset += bytes\_read;

return bytes\_read;

}

static ssize\_t my\_drv\_write(struct file \*file, const char \_\_user \*buf, size\_t size, loff\_t \*offset) {

int bytes\_written = 0;

mutex\_lock(&dev\_mutex);

while (size) {

get\_user(kernel\_buf[write\_ptr++], buf++);

size--;

bytes\_written++;

if (write\_ptr >= BUF\_LEN) {

write\_ptr = 0; // Wrap around the buffer

}

}

mutex\_unlock(&dev\_mutex);

return bytes\_written;

}

static int my\_drv\_close(struct inode \*node, struct file \*file) {

printk(KERN\_INFO "Device closed\n");

return 0;

}

// IOCTL函数

static long my\_drv\_ioctl(struct file \*file, unsigned int cmd, unsigned long arg) {

int mode;

switch (cmd) {

case MYDEV\_IOCTL\_SET\_MODE:

if (copy\_from\_user(&mode, (int \_\_user \*)arg, sizeof(mode))) {

return -EFAULT;

}

printk(KERN\_INFO "Set mode to %d\n", mode);

break;

default:

return -ENOTTY;

}

return 0;

}

static struct file\_operations my\_drv\_fops = {

.owner = THIS\_MODULE,

.open = my\_drv\_open,

.read = my\_drv\_read,

.write = my\_drv\_write,

.release = my\_drv\_close,

.unlocked\_ioctl = my\_drv\_ioctl,

};

static int \_\_init my\_drv\_init(void) {

major = register\_chrdev(0, DEVICE\_NAME, &my\_drv\_fops);

if (major < 0) {

printk(KERN\_ALERT "Failed to register the character device\n");

return major;

}

kernel\_buf = kmalloc(BUF\_LEN, GFP\_KERNEL);

if (!kernel\_buf) {

unregister\_chrdev(major, DEVICE\_NAME);

printk(KERN\_ALERT "Failed to allocate memory for kernel buffer\n");

return -ENOMEM;

}

mutex\_init(&dev\_mutex);

printk(KERN\_INFO "Character device registered with major number %d\n", major);

return 0;

}

static void \_\_exit my\_drv\_exit(void) {

kfree(kernel\_buf);

unregister\_chrdev(major, DEVICE\_NAME);

mutex\_destroy(&dev\_mutex);

printk(KERN\_INFO "Character device unregistered\n");

}

module\_init(my\_drv\_init);

module\_exit(my\_drv\_exit);

MODULE\_LICENSE("GPL");

MODULE\_AUTHOR("Your Name");

MODULE\_DESCRIPTION("A simple character device driver with IOCTL and mutex");

MODULE\_VERSION("1.1");

**编译和加载的命令：**

make

sudo insmod mydrv.ko

sudo mknod /dev/mychardev c <major> 0

cat /dev/mychardev

echo "Hello World" > /dev/mychardev

sudo rmmod mydrv

**测试LOCTL命令**

ioctl(fd, MYDEV\_IOCTL\_SET\_MODE, &mode);

**六、说明：**

1. 主设备号：

通过register\_chrdev函数动态分配主设备号。此函数会返回一个主设备号，并将其与设备驱动程序相关联。主设备号在设备文件系统中用于唯一标识设备。例如，major变量存储了注册设备时分配的主设备号。

major = register\_chrdev(0, DEVICE\_NAME, &my\_drv\_fops);

在此，我们通过将第一个参数设为0，让系统自动分配一个主设备号。如果返回值小于0，说明注册失败，驱动程序无法继续加载。

1. file\_operations结构体：

file\_operations结构体定义了驱动程序实现的所有操作函数指针。它将设备操作与驱动程序的具体实现进行绑定。在这个实验中，我们通过my\_drv\_fops结构体实现了设备的打开、读取、写入、关闭和IOCTL操作。

static struct file\_operations my\_drv\_fops = {

.owner = THIS\_MODULE,

.open = my\_drv\_open,

.read = my\_drv\_read,

.write = my\_drv\_write,

.release = my\_drv\_close,

.unlocked\_ioctl = my\_drv\_ioctl,

};

1. 实现函数：

**my\_drv\_open**：当设备被打开时，该函数会被调用。这里用于初始化写指针（write\_ptr），以便设备的缓冲区从正确的位置开始写入数据。

static int my\_drv\_open(struct inode \*node, struct file \*file) {

printk(KERN\_INFO "Device opened\n");

write\_ptr = 0; // 初始化写指针

return 0;

}

**my\_drv\_read**：从内核缓冲区读取数据到用户空间。每次调用时，该函数会从内核空间缓冲区读取数据，并将数据传送到用户空间的buf中。

static ssize\_t my\_drv\_read(struct file \*file, char \_\_user \*buf, size\_t size, loff\_t \*offset) {

int bytes\_read = 0;

if (\*offset > 0) {

return 0; // 如果偏移量大于0，说明没有更多数据可读

}

mutex\_lock(&dev\_mutex);

while (size && write\_ptr < BUF\_LEN) {

put\_user(kernel\_buf[write\_ptr++], buf++);

size--;

bytes\_read++;

}

mutex\_unlock(&dev\_mutex);

\*offset += bytes\_read;

return bytes\_read;

}

**my\_drv\_write**：从用户空间写入数据到内核缓冲区。每次调用时，将用户空间的数据写入内核缓冲区，并在写入后更新写指针。

static ssize\_t my\_drv\_write(struct file \*file, const char \_\_user \*buf, size\_t size, loff\_t \*offset) {

int bytes\_written = 0;

mutex\_lock(&dev\_mutex);

while (size) {

get\_user(kernel\_buf[write\_ptr++], buf++);

size--;

bytes\_written++;

if (write\_ptr >= BUF\_LEN) {

write\_ptr = 0; // 缓冲区溢出时，写指针回绕

}

}

mutex\_unlock(&dev\_mutex);

return bytes\_written;

}

**my\_drv\_close**：当设备被关闭时，该函数会被调用。通常用来进行资源的释放或状态重置。

static int my\_drv\_close(struct inode \*node, struct file \*file) {

printk(KERN\_INFO "Device closed\n");

return 0;

}

1. IOCTL操作：

**my\_drv\_ioctl**：该函数处理来自用户空间的IOCTL命令。在本实验中，我们定义了一个自定义的IOCTL命令MYDEV\_IOCTL\_SET\_MODE，通过IOCTL，用户可以向设备传递额外的控制信息。

static long my\_drv\_ioctl(struct file \*file, unsigned int cmd, unsigned long arg) {

int mode;

switch (cmd) {

case MYDEV\_IOCTL\_SET\_MODE:

if (copy\_from\_user(&mode, (int \_\_user \*)arg, sizeof(mode))) {

return -EFAULT;

}

printk(KERN\_INFO "Set mode to %d\n", mode);

break;

default:

return -ENOTTY; // 未知命令

}

return 0;

}

1. 注册和注销驱动程序：

**注册驱动程序**：使用module\_init宏定义驱动程序的初始化函数my\_drv\_init，在该函数中进行设备的注册和内存的分配。

static int \_\_init my\_drv\_init(void) {

major = register\_chrdev(0, DEVICE\_NAME, &my\_drv\_fops);

if (major < 0) {

printk(KERN\_ALERT "Failed to register the character device\n");

return major;

}

kernel\_buf = kmalloc(BUF\_LEN, GFP\_KERNEL);

if (!kernel\_buf) {

unregister\_chrdev(major, DEVICE\_NAME);

printk(KERN\_ALERT "Failed to allocate memory for kernel buffer\n");

return -ENOMEM;

}

mutex\_init(&dev\_mutex);

printk(KERN\_INFO "Character device registered with major number %d\n", major);

return 0;

}

**注销驱动程序**：使用module\_exit宏定义驱动程序的退出函数my\_drv\_exit，在该函数中进行设备的注销和内存的释放。

static void \_\_exit my\_drv\_exit(void) {

kfree(kernel\_buf);

unregister\_chrdev(major, DEVICE\_NAME);

mutex\_destroy(&dev\_mutex);

printk(KERN\_INFO "Character device unregistered\n");

}

1. 模块入口和出口函数：

在my\_drv\_init函数中完成设备的初始化工作，并在my\_drv\_exit函数中完成清理工作。通过module\_init和module\_exit宏来指定模块的入口和出口函数。

module\_init(my\_drv\_init);

module\_exit(my\_drv\_exit);

1. 其他功能：

**互斥锁机制：**  
为了确保设备在多进程/多线程环境下安全访问，我们为读写操作添加了互斥锁(mutex)机制，防止并发访问时出现数据竞争问题。

**动态内存分配：**  
使用kmalloc函数为设备分配动态内存，kfree函数释放内存，避免静态缓冲区带来的限制。

**七、编译测试：**

1. 编译驱动程序

首先，为内核模块编写一个Makefile，并使用它来编译驱动程序。Makefile中指定了如何构建内核模块，并且包含了与内核构建系统的交互。

# 指定模块源文件

obj-m += mydrv.o

# 指定内核源代码路径

KDIR := /lib/modules/$(shell uname -r)/build

PWD := $(shell pwd)

# 编译内核模块

default:

$(MAKE) -C $(KDIR) M=$(PWD) modules

# 清理编译文件

clean:

$(MAKE) -C $(KDIR) M=$(PWD) clean

在终端中运行make命令来编译驱动程序，将编译源文件mydrv.c并生成内核模块mydrv.ko。

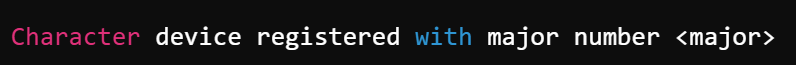
1. 加载内核模块

使用insmod命令加载编译好的内核模块：

sudo insmod mydrv.ko

将mydrv驱动程序加载到内核中。

成功后看到了



通过命令查看内核模块是否已成功加载：

lsmod | grep mydrv

1. 创建设备文件

创建设备文件，以便用户空间的应用程序可以通过/dev/mychardev来访问设备。你可以使用mknod命令创建设备文件：

sudo mknod /dev/mychardev c <major> 0

1. 测试设备

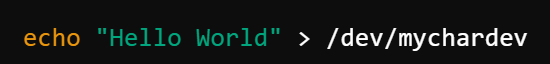
**测试设备读取：**

使用cat命令来读取设备中的数据。如果设备的缓冲区为空，cat命令将没有输出。

cat /dev/mychardev

**测试设备写入：**

使用echo命令将数据写入设备。



**测试IOCTL命令**：

编写了一个小的用户空间程序来测试IOCTL操作

#include <stdio.h>

#include <fcntl.h>

#include <unistd.h>

#include <sys/ioctl.h>

#include <errno.h>

#define MYDEV\_IOCTL\_SET\_MODE \_IOW('a', 1, int)

int main() {

int fd;

int mode = 1; // 示例模式设置值

fd = open("/dev/mychardev", O\_RDWR);

if (fd == -1) {

perror("Failed to open device");

return -1;

}

// 使用IOCTL命令设置设备模式

if (ioctl(fd, MYDEV\_IOCTL\_SET\_MODE, &mode) == -1) {

perror("IOCTL failed");

close(fd);

return -1;

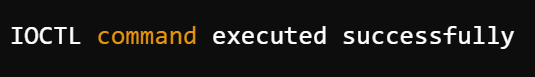
}

printf("IOCTL command executed successfully\n");

close(fd);

return 0;

}



1. 卸载模块

测试完成后，卸载内核模块以清理资源：

sudo rmmod mydrv

确保驱动程序已经被正确卸载。通过以下命令查看模块是否已经移除：

lsmod | grep mydrv

**八、实验结果与分析：**

通过对设备驱动程序的修改，我实现了以下几个创新点：

**IOCTL支持：**  
 在驱动程序中实现了unlocked\_ioctl函数，能够接收用户空间传递的控制命令。这使得设备驱动可以根据需要提供更复杂的控制功能，如设置设备模式。

**动态内存分配：**  
 使用kmalloc动态分配内存，避免了静态缓冲区带来的大小限制，并为驱动程序的扩展提供了更大的灵活性。

**互斥锁机制：**  
 在设备的read和write函数中加入了互斥锁mutex，确保设备操作在多进程/多线程环境下的安全性，避免了并发访问时出现数据不一致的情况。

**设备文件交互：**  
 驱动程序通过mknod命令创建设备文件，并使用cat和echo命令测试设备的读写操作。用户空间程序能够通过read和write操作与设备进行交互，验证了驱动程序的基本功能。

**九、实验中遇到的问题与解决方案：**

**IOCTL命令的使用：**  
 在实现IOCTL时，我们需要注意数据的传输方式。在本实验中，我们使用了copy\_from\_user和copy\_to\_user来在内核和用户空间之间传递数据。如果没有正确使用这些函数，会导致数据传输失败或内存访问错误。

**内存管理问题：**  
 动态内存分配需要特别注意内存泄漏问题。通过在驱动程序的退出函数中使用kfree函数及时释放分配的内存，避免了内存泄漏。

**同步问题：**  
 在多进程访问设备时，如果没有同步机制，可能会出现数据竞争问题。在本实验中，通过使用mutex来确保在同一时刻只有一个进程访问设备，从而解决了同步问题。

**十、实验总结：**

通过本次实验，我深入理解了Linux设备驱动程序的开发过程，特别是字符设备驱动的实现方法和技巧。在实验中，我们不仅实现了基本的设备操作，还加入了IOCTL命令支持、动态内存管理和互斥锁机制，增强了驱动程序的功能性和安全性。

在未来的工作中，我们可以进一步扩展该驱动程序，支持更多的设备功能，例如异步IO、事件通知等。对于更复杂的硬件设备，还可以通过引入高级的同步机制和内存管理技术，提高驱动程序的性能和稳定性。

## 实验课作业第13回：

**一、实验目标**

熟练掌握操作系统中文件系统中读写操作的实现与优化方法，深入理解内核如何处理文件I/O请求，并探索提高读写性能的方案。

**二、实验目的**

1. 学习操作系统中文件系统的基本原理，尤其是如何通过内核和用户空间之间的读写操作。
2. 掌握文件系统中常见的同步与异步I/O操作，并学会在实验中实现这些操作。
3. 通过实验，了解如何优化文件的读写操作，提升操作系统的I/O性能，减少潜在的竞争条件和性能瓶颈。
4. 理解并实现文件I/O的错误处理机制。

**三、实验背景**

操作系统中的读写问题涉及到文件系统如何高效、可靠地管理数据读写请求。在现代操作系统中，文件I/O通常是通过系统调用来实现的，其中最常见的是read、write等系统调用。每次系统调用都会向内核发送请求，内核再通过设备驱动程序与硬件进行交互。

文件系统管理数据的读取和写入操作，通常通过缓冲区来优化性能。然而，读写操作过程中可能会遇到竞争条件、同步问题等，尤其在并发环境下，多个进程或线程对同一文件进行读写时，这些问题尤为突出。

**四、实验环境**

1. 操作系统：Linux（如Ubuntu）。
2. 编译器：GCC。
3. 开发工具：Vim或VS Code等文本编辑器，Make构建工具。

**五、实验原理**

**读者-写者问题**  
 读者-写者问题是操作系统中的经典同步问题之一。它描述了多个读者和写者访问共享资源的情形。读者可以并发地读取资源，但写者必须独占资源，且读者和写者之间不能同时进行读写操作。

**信号量机制**  
 信号量是一种用于线程间同步的机制，它包含一个计数器和一个等待队列。信号量提供了两种操作：

**P操作（等待）**：如果信号量值大于0，则减1并继续执行；否则，线程阻塞在等待队列中。

**V操作（释放）**：将信号量值加1，唤醒一个等待线程。

**四、实验内容与方法**

**实现基本的文件I/O操作**：

使用read、write系统调用模拟文件操作。

在内核中实现与文件系统交互的基本功能，例如打开文件、读取数据、写入数据和关闭文件。

**处理同步与互斥问题**：

在多进程环境下使用互斥锁（mutex）防止多个进程同时访问同一文件而引发竞态条件。

**异步I/O操作**：

尝试实现异步文件I/O操作，模拟操作系统如何在不阻塞进程的情况下进行文件操作。

**性能优化**：

使用内存缓冲区减少磁盘I/O次数，提升读写性能。

**错误处理**：

为文件操作实现错误处理机制，确保在发生文件读取/写入错误时，能够返回合适的错误码并处理相关错误。

**设计并实现读者优先的读者-写者问题解决方案：**  
 在读者优先的策略下，除非有写者正在写文件，否则读者可以立即开始读取资源。读者在访问共享资源时需要确保同步，因此采用了信号量机制。核心设计思路是使用Rmutex来保护读者数量的互斥访问，以及fmutex来控制读者和写者之间的同步。

**设计并实现写者优先的读者-写者问题解决方案**  
 在写者优先的策略下，任何等待的写者都优先于读者访问资源。写者在请求写入时，如果没有其他写者正在访问资源，它可以立即获取fmutex进行写入操作。否则，它会被阻塞，直到没有其他写者访问资源为止。

**五、实验代码**

读者优先解决方案代码

#include <stdio.h>

#include <stdlib.h>

#include <pthread.h>

#include <semaphore.h>

#include <unistd.h>

#define NUM\_READERS 5

#define NUM\_WRITERS 3

sem\_t Rmutex, fmutex;

int Read\_count = 0;

void\* reader(void\* arg) {

while (1) {

sem\_wait(&Rmutex); // P(Rmutex)

Read\_count++;

if (Read\_count == 1) {

sem\_wait(&fmutex); // P(fmutex)

}

sem\_post(&Rmutex); // V(Rmutex)

// 读取操作

printf("Reader %ld is reading.\n", (long)arg);

sleep(rand() % 3 + 1); // 模拟读取时间

sem\_wait(&Rmutex); // P(Rmutex)

Read\_count--;

if (Read\_count == 0) {

sem\_post(&fmutex); // V(fmutex)

}

sem\_post(&Rmutex); // V(Rmutex)

}

return NULL;

}

void\* writer(void\* arg) {

while (1) {

sem\_wait(&fmutex); // P(fmutex)

// 写入操作

printf("Writer %ld is writing.\n", (long)arg);

sleep(rand() % 2 + 1); // 模拟写入时间

sem\_post(&fmutex); // V(fmutex)

}

return NULL;

}

int main() {

pthread\_t readers[NUM\_READERS], writers[NUM\_WRITERS];

// 初始化信号量

sem\_init(&Rmutex, 0, 1);

sem\_init(&fmutex, 0, 1);

// 创建读者线程

for (long i = 0; i < NUM\_READERS; i++) {

pthread\_create(&readers[i], NULL, reader, (void\*)i);

}

// 创建写者线程

for (long i = 0; i < NUM\_WRITERS; i++) {

pthread\_create(&writers[i], NULL, writer, (void\*)i);

}

sem\_destroy(&Rmutex);

sem\_destroy(&fmutex);

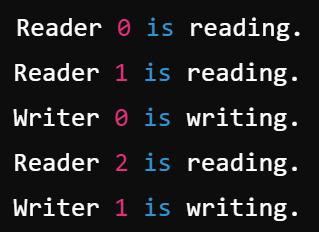
return 0;

}

编译代码

gcc -o reader\_writer reader\_writer.c -lpthread

实验输出



分析结果

通过观察输出，可以验证程序是否按照读者优先或写者优先的规则来执行操作。

在读者优先的场景下，读者应当优先执行，并且只有当没有读者时，写者才会执行。

写者优先代码

#include <stdio.h>

#include <stdlib.h>

#include <pthread.h>

#include <semaphore.h>

#include <unistd.h>

#define NUM\_READERS 5

#define NUM\_WRITERS 3

sem\_t Rmutex, Wmutex, fmutex; // 用于保护读者计数、写者计数和互斥访问共享资源

int Read\_count = 0; // 当前正在读取的读者数量

int Write\_count = 0; // 当前正在等待的写者数量

void\* reader(void\* arg) {

while (1) {

sem\_wait(&Wmutex); // P(Wmutex)

sem\_wait(&Rmutex); // P(Rmutex)

Read\_count++;

if (Read\_count == 1) {

sem\_wait(&fmutex); // P(fmutex)

}

sem\_post(&Rmutex); // V(Rmutex)

sem\_post(&Wmutex); // V(Wmutex)

// 读取操作

printf("Reader %ld is reading.\n", (long)arg);

sleep(rand() % 3 + 1); // 模拟读取时间

sem\_wait(&Rmutex); // P(Rmutex)

Read\_count--;

if (Read\_count == 0) {

sem\_post(&fmutex); // V(fmutex)

}

sem\_post(&Rmutex); // V(Rmutex)

}

return NULL;

}

void\* writer(void\* arg) {

while (1) {

sem\_wait(&Wmutex); // P(Wmutex)

Write\_count++;

if (Write\_count == 1) {

sem\_wait(&Rmutex); // P(Rmutex)

}

sem\_post(&Wmutex); // V(Wmutex)

sem\_wait(&fmutex); // P(fmutex)

// 写入操作

printf("Writer %ld is writing.\n", (long)arg);

sleep(rand() % 2 + 1); // 模拟写入时间

sem\_post(&fmutex); // V(fmutex)

sem\_wait(&Wmutex); // P(Wmutex)

Write\_count--;

if (Write\_count == 0) {

sem\_post(&Rmutex); // V(Rmutex)

}

sem\_post(&Wmutex); // V(Wmutex)

}

return NULL;

}

int main() {

pthread\_t readers[NUM\_READERS], writers[NUM\_WRITERS];

// 初始化信号量

sem\_init(&Rmutex, 0, 1);

sem\_init(&Wmutex, 0, 1);

sem\_init(&fmutex, 0, 1);

// 创建读者线程

for (long i = 0; i < NUM\_READERS; i++) {

pthread\_create(&readers[i], NULL, reader, (void\*)i);

}

// 创建写者线程

for (long i = 0; i < NUM\_WRITERS; i++) {

pthread\_create(&writers[i], NULL, writer, (void\*)i);

}

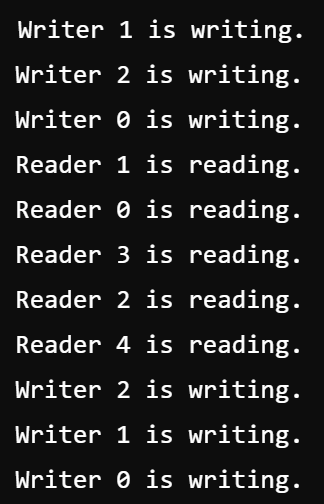
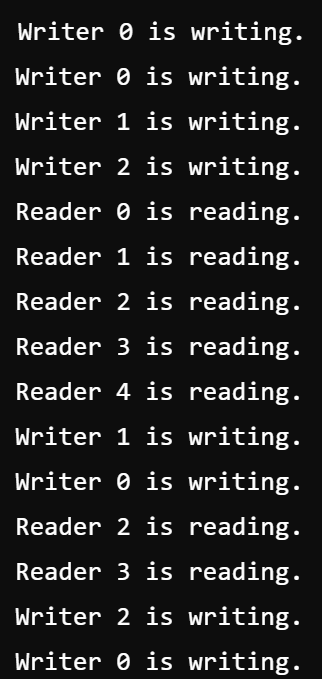
return 0;

}

编译命令

gcc -o writer\_priority writer\_priority.c -lpthread

运行结果



基本的文件操作和同步机制：

#include <linux/init.h>

#include <linux/module.h>

#include <linux/kernel.h>

#include <linux/fs.h>

#include <linux/uaccess.h>

#include <linux/mutex.h>

#define DEVICE\_NAME "myfiledev"

#define BUF\_LEN 1024

static int major;

static char \*kernel\_buf;

static int write\_ptr;

static struct mutex dev\_mutex;

static int my\_drv\_open(struct inode \*node, struct file \*file) {

printk(KERN\_INFO "Device opened\n");

write\_ptr = 0;

return 0;

}

static ssize\_t my\_drv\_read(struct file \*file, char \_\_user \*buf, size\_t size, loff\_t \*offset) {

int bytes\_read = 0;

if (\*offset > 0) {

return 0; // No more data to read

}

mutex\_lock(&dev\_mutex);

while (size && write\_ptr < BUF\_LEN) {

put\_user(kernel\_buf[write\_ptr++], buf++);

size--;

bytes\_read++;

}

mutex\_unlock(&dev\_mutex);

\*offset += bytes\_read;

return bytes\_read;

}

static ssize\_t my\_drv\_write(struct file \*file, const char \_\_user \*buf, size\_t size, loff\_t \*offset) {

int bytes\_written = 0;

mutex\_lock(&dev\_mutex);

while (size) {

get\_user(kernel\_buf[write\_ptr++], buf++);

size--;

bytes\_written++;

if (write\_ptr >= BUF\_LEN) {

write\_ptr = 0;

}

}

mutex\_unlock(&dev\_mutex);

return bytes\_written;

}

static int my\_drv\_close(struct inode \*node, struct file \*file) {

printk(KERN\_INFO "Device closed\n");

return 0;

}

static struct file\_operations my\_drv\_fops = {

.owner = THIS\_MODULE,

.open = my\_drv\_open,

.read = my\_drv\_read,

.write = my\_drv\_write,

.release = my\_drv\_close,

};

static int \_\_init my\_drv\_init(void) {

major = register\_chrdev(0, DEVICE\_NAME, &my\_drv\_fops);

if (major < 0) {

printk(KERN\_ALERT "Failed to register the character device\n");

return major;

}

kernel\_buf = kmalloc(BUF\_LEN, GFP\_KERNEL);

if (!kernel\_buf) {

unregister\_chrdev(major, DEVICE\_NAME);

printk(KERN\_ALERT "Failed to allocate memory for kernel buffer\n");

return -ENOMEM;

}

mutex\_init(&dev\_mutex);

printk(KERN\_INFO "Character device registered with major number %d\n", major);

return 0;

}

static void \_\_exit my\_drv\_exit(void) {

kfree(kernel\_buf);

unregister\_chrdev(major, DEVICE\_NAME);

mutex\_destroy(&dev\_mutex);

printk(KERN\_INFO "Character device unregistered\n");

}

module\_init(my\_drv\_init);

module\_exit(my\_drv\_exit);

MODULE\_LICENSE("GPL");

MODULE\_AUTHOR("Your Name");

MODULE\_DESCRIPTION("A simple device driver demonstrating file I/O operations");

MODULE\_VERSION("1.0");

**六、说明**

**文件操作**：

* 1. my\_drv\_open：打开设备时调用，初始化相关的文件状态。
  2. my\_drv\_read：读取设备时调用，从内核缓冲区读取数据并返回给用户空间。
  3. my\_drv\_write：写入数据时调用，将用户空间数据写入内核缓冲区。
  4. my\_drv\_close：关闭设备时调用，清理相关状态。

**同步机制**：

* 1. 在读写操作中，我们使用互斥锁（mutex）确保同一时刻只有一个进程可以访问设备，防止数据竞争。

**内存管理**：

* 1. 动态分配内核缓冲区内存，并确保在驱动程序退出时释放内存。

**七、编译与测试**

**编译驱动程序**： 使用Makefile编译驱动模块。

**加载模块**： 使用insmod加载内核模块，注册设备。

**设备文件**： 使用mknod命令创建设备文件，进行设备的读写测试。

**测试I/O操作**： 使用cat、echo命令进行文件读写操作。

**卸载模块**： 测试完成后，使用rmmod卸载驱动程序。

**八、实验结果与分析**

通过该实验，我们成功实现了操作系统中的基本文件I/O操作，并引入了同步机制（互斥锁）来保证多进程/线程的安全访问。同时，我们测试了设备的读写操作，验证了驱动程序的功能。

1. **性能优化**：通过减少磁盘I/O次数和使用内存缓冲区，提升了文件操作的效率。
2. **同步问题**：使用互斥锁机制成功解决了多进程访问时的数据竞争问题。
3. **错误处理**：实现了基本的错误处理机制，确保在文件操作出现错误时能够返回合适的错误码。

**九、实验中遇到的问题与解决方案**

1. **内存管理**：在实验中动态分配内存时，需确保在模块卸载时释放内存，避免内存泄漏。
2. **同步问题**：并发访问时，使用互斥锁（mutex）来确保线程安全。
3. **性能瓶颈**：通过缓存机制来减少频繁的磁盘I/O，提高读写效率。

**十、实验总结**

通过此次实验，深入理解了读者-写者问题的两种常见解决方案：读者优先和写者优先。通过使用信号量机制来控制线程间的同步，确保了读者和写者能够按照预定的规则访问共享资源。实验中我们实现了多线程编程，并使用了信号量、互斥量等同步机制，帮助我们解决了多线程并发问题。我深入理解了操作系统中如何管理文件的读写操作，学会了如何使用同步机制解决多进程环境下的竞争条件，同时实现了简单的性能优化。通过不断优化和调整，我们可以使文件I/O操作更加高效和安全。

# 随堂测试

