一、填空

1. Fork 系统调用的返回是__子进程 PID 或 0_____。
2. 使用动态优先级做调度决策的系统调度算法有，___MLFQ_____、_步幅调度___、_和____CFS_____。
3. RISC-V 架构下全局页目录表的首地址由___SATP（页表基地址寄存器）____保存。
4. Linux 将中断管理例程分为___上半部(top half)_____和_____下半部(bottom half)_____两个阶段。
5. 信号量包括两种典型的操作：___P（down 或者 wait）_____和___V（up 或者 signal）_____。
6. GNU 的全称是____GNU's Not Unix_____。
7. 32 位 Linux 操作系统采用__2_级页式管理虚拟内存，分别是___页目录表__、__页中间表_____。
8. RISC-V 架构下分为___4___个特权级。
9. Linux 将软中断管理实现包括__软中断号、中断向量表、软中断处理函数、软中断通信机制、软中断调度_____。 （答出 3 个即可）
10. Linux 命名空间子系统包括_UTS_、__ipc__、_mnt__、_pid_和_net_。
11. 操作系统级虚拟化技术分为____Type-1_____和_____Type-2____。

二、选择

1.采用分段管理的操作系统，若地址用 24 位表示，其中 8 位表示段号，则允许每段的最大长度为（B）。

A 2^{24}　　B 2^{16}　　C 2^{8}　　D 2^{32}

2. 进程和程序的本质区别是 (D)。

A 存储在内存和外存　　B 顺序和非顺序执行机器指令

C 分时使用和独占使用计算机资源　　D 动态和静态特征

3. 若 P、V 操作的信号量 S 初值为 2，当前值为 -1，则表示有(B)个等待进程。

A 0　　B 1　　C 2　　D 3

4. 通常，用户编写的程序中所使用的地址是(A)。

A 逻辑地址　　B 物理地址　　C 绝对地址　　D 内存地址

5. 把逻辑地址转变为内存的物理地址的过程称作(D)。

A 编译　　B 连接　　C 运行　　D 重定位

6. 进程控制块是描述进程状态和特性的数据结构，一个进程(D)。

A 可以有多个进程控制块　　B 可以和其他进程共用一个进程控制块

C 可以没有进程控制块　　D 只能有惟一的进程控制块

7. 在分时系统中，时间片一定，则（ B ），响应时间越长。

A 内存越大　　B 任务数越多　　C 后备队列越短　　D 任务数越少

8. 以下说法正确的是 （ C ）。

A 话题通讯只能通过 UDP 的方式实现

B ROS 节点编程语言仅支持 C++

C 单个机器人中节点的名称必须唯一

D rviz 和 gezebo 是对同一逻辑功能的不同实现

9. 关于 Linux 内核，以下描述不正确的是（ C ）。

A 是一个 GPL 许可协议下的开源软件B 是一个宏内核

C 作者是 Richard Stallman　　D 被 Android 操作系统所采用

10. 某计算机采用二级页表的分页存储管理方式，按字节编制，页大小为 2^{10} 字节，页表项大小为 2 字节，逻辑地址结构为：

逻辑地址空间大小为 2^{16} 页，则表示整个逻辑地址空间的页目录表中包含表项的个数至少是（ B ）。

A 64　　B 128　　C 256　　D 512

11. 在支持多线程的系统中，进程 P 创建的若干个线程不能共享的是（ D ）。

A 进程 P 的代码段　　B 进程 P 中打开的文件

C 进程 P 的全局变量　　D 进程 P 中某线程的栈指针

12. 在请求分页系统中，LRU 算法是指（ B ）。

A 最早进入内存的页先淘汰

B 近期最长时间以来没被访问的页先淘汰

C 近期被访问次数最少的页先淘汰

D 以后再也不用的也先淘汰

13. 下列选项中，操作系统提供的给应用程序的接口是（ A ）。

A 系统调用　　B 中断　　C 库函数　　D 原语

14. 关于操作系统内核的核心功能，以下描述不正确的是（ B ）。

A 进程调度　　B 数据管理　　C 内存管理　　D 设备驱动

15. 本地用户通过键盘登录系统时，首先获得键盘输入信息的程序是（ B ）。

A 命令解释程序　B 中断处理程序C 系统调用程序D 用户登录程序

16. 在虚拟内存管理中，地址变换机构将逻辑地址变换为物理地址，形成该逻辑地址的阶段是（ C ）。

A 编辑　　B 编译　　C 连接　　D 装载

17. 以下有关进程描述不正确的是（ D ）。

A 是一个程序的动态执行上下文　　B 是一种执行的实体

C 是调度的基本单元　　D 是系统调度的单位

18. 下列选项中，满足短任务优先且不会发生饥饿现象的调度算法是（ B ）。

A 先来先服务　　B 高响应比优先

C 时间片轮转　　D 非抢占式短任务优先

19. 下列选项中会导致进程从执行态变为就绪态的事件是（ D ）。

A 执行 P(wait)操作　　B 申请内存失败

C 启动 I/O 设备　　D 被高优先级进程抢占

20. 以下那个选项不是进程中信号处理流程的一个阶段（ D ）。

A 对信号进行处理　　B 信号在进程中注册

C 信号诞生　　D 将信号进行复制

21. 设备的分类方法很多，下列错误的是（ D ）

A、按数据传输单位可以分为字符设备和块设备

B、按使用特性可以分为存储设备和 I/O 设备

C、按共享特性可分为独占设备和共享设备

D、按传输速度可分为低速设备、匀速设备和加速设备

22. 在请求分页系统中，LRU 算法是指（ B ）。

A 最早进入内存的页先淘汰　　B 近期最长时间以来没被访问的页先淘汰

C 近期被访问次数最少的页先淘汰　　D 以后再也不用的也先淘汰

23 多道程序设计（Multiprogramming）的主要目标是（ A ）。
A 提高系统性能和资源利用率　B 提供图形用户界面
C 增加存储容量　　D 管理网络连接

24. 用户态（User mode）和内核态（Kernel mode）之间的主要区别是（ C ）。
A 用户态可以访问系统资源，而内核态不能
B 用户态具有更高的权限级别
C 用户态只能运行用户应用程序，而内核态可以执行特权指令和访问系统资源
D 用户态和内核态没有明显区别

25. 死锁（Deadlock）是指（ D ）。
A 一种计算机病毒
B 一个运行缓慢的程序
C 一种系统崩溃
D 一种资源竞争的情况，导致进程无法继续执行

26. 操作系统中有一组特殊的程序，它们不能被系统中断，在操作系统中称为（B ）。
A、初始化程序 B、原语 C、子程序 D、控制模块

三、简答（共 30 分，每题 10 分）

1. 什么是页表？它在地址转换中起什么作用？

页表是一种特殊的数据结构，用于记录虚拟页与物理页之间的映射。在分页机制下，页表用于将虚拟地址转换为物理地址，以便进程能够正确地访问内存中的数据和指令。每个进程都拥有一个自己的页表，PCB（进程控制块）表中通常会有一个指针指向该页表。当程序需要访问某个虚拟地址时，CPU 会首先将该地址转换为逻辑地址。然后通过查找页表，找到对应逻辑地址所对应的物理页号。根据系统设置好的分页大小、换进/换出机制等规则，确定最终要访问的具体物理位置（即实际的起始地址），从而实现对真实内存的有效管理。

2. 请简述全虚拟化和半虚拟化的主要区别。

全虚拟化是一种完全模拟硬件的虚拟化方式，它能够通过创建全新的虚拟系统来实现底层物理系统的全部抽象化。全虚拟化计算效率高，而且不需要修改客户机操作系统或应用程序，因此在客户端看来，它就像在硬件上直接运行一样。然而，由于 Hypervisor 的使用，全虚拟化的性能可能逊色于纯虚拟机。全虚拟化的最大优势在于无需对客户端操作系统进行任何改动，只需要对基础的硬件进行支持。

半虚拟化基于全虚拟化技术新增一个能够优化客户端 OS 指令的 API 系统，以减轻 Hypervisor 的工作量，从而快速实现底层硬件的访问。半虚拟化解决了敏感函数不下陷的问题，协同设计的思想也可以提升某些场景下的系统性能。但是半虚拟化需要修改操作系统代码，难以在闭源系统中应用。此外即使是开源系统，也难以同时在不同版本中实现。

3. 请简述分段和分页式内存管理的过程。

在分段式内存管理中，程序被划分为多个逻辑段，每个段的大小并不固定。当一个段被加载到内存中时，操作系统会在内存中为其分配一个连续的区域。由于段的长度不固定，因此可能会产生内部碎片。为了实现地址转换，系统需要维护一个段表，该表记录了每个段的起始地址和长度。当程序需要访问某个地址时，操作系统会通过查找段表来找到对应的物理地址。

而在分页式内存管理中，程序被划分为固定大小的页。当程序运行时，操作系统将这些页映射到物理内存中。与分段不同，分页将程序的各个部分分散到物理内存的不同位置，有效减少内部碎片的产生。操作系统维护一个页表，该表记录了每个页的起始地址和长度。当程序需要访问某个地址时，操作系统会通过查找页表来找到对应的物理地址。

4. 请解释虚拟内存（Virtual Memory）的概念及其作用。

虚拟内存是一种操作系统技术，它将主存和磁盘结合使用，使得程序认为它拥有连续的、私有的地址空间。它通过将程序的逻辑地址映射到物理地址，允许程序访问超出物理内存大小的数据。虚拟内存的作用包括扩展可用的内存空间、实现进程间的内存隔离、提供更高的安全性和方便的内存管理。

5. 请简述共享内存和消息传递的区别。

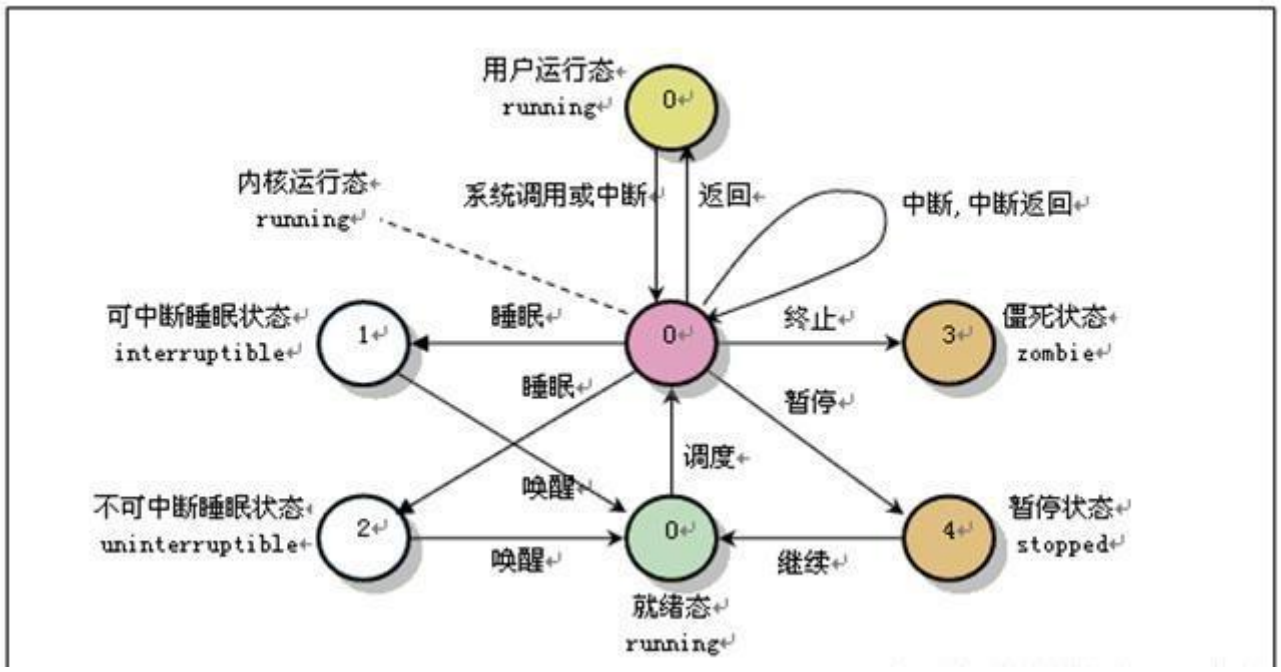
共享内存允许两个或多个进程共享同一块物理内存。通过共享内存，进程可以直接读写这块内存区域，从而实现数据交换。共享内存的优点在于，多核场景下，用户态无需切换到内核态即可完成通信；完全有用户态程序控制，具有更强的定制能力；速度快，因为它避免了数据的复制操作，可以直接读写内存，实现零内存拷贝然而，共享内存也带来了同步和数据一致性的问题，需要使用信号量等机制来确保不同进程对共享内存的访问不会发生冲突。

消息传递是一种更为灵活的进程间通信方式。在消息传递中，进程通过发送和接收消息来进行通信。消息可以是任意格式的数据，并且可以包含控制信息或者实际数据。消息传递的优点在于它能够处理不同类型和格式的数据，并且可以支持分布式系统中的进程间通信。此外，消息传递也提供了更好的灵活性和异步性，因为消息的发送和接收可以是异步进行的，不需要等待对方进程的响应。

6. 画出 Linux 进程的主要执行状态和生命周期图。

运行状态（TASK_RUNNING）
可中断睡眠状态（TASK_INTERRUPTIBLE）
不可中断睡眠状态（TASK_UNINTERRUPTIBLE）
暂停状态（TASK_STOPPED）

僵死状态（TASK_ZOMBIE）



7. 现代 CPU 至少有两个特权级，分别用来运行内核和用户程序。为什么要这样设计?

答：使用特权级别可以实现对系统资源和关键数据的保护。内核运行在较高的特权级别，能够执行一些敏感的操作和访问受保护的系统资源，而用户程序运行在较低的特权级别，受到限制，无法直接访问关键系统资源。这有助于防止用户程序对系统造成意外或恶意的破坏，确保系统的安全性和可靠性。

8. 进行系统调用时，Exception 会带来哪些额外的开销？试设计一个 Exception-Less 的系统调用方案，并与传统方案作比较（提示：利用多核）。

答：在传统的系统调用方案中，当用户程序需要执行系统调用时，会触发一个 exception，导致从用户态切换到内核态，并进入内核中相应的系统调用处理程序。额外开销涉及到状态的保存和恢复、TLB（Translation Lookaside Buffer）刷新、调度开销等（合理即可）。

方案：可以在系统中驻留处理系统调用的内核线程，在系统调用时用消息传递机制将系统调用请求和参数等信息传递给相应的内核线程。这可以使用一些 IPC（进程间通信）机制，如消息队列、共享内存等。

9. 用户程序执行的过程会触发同步异常或被异步中断打断，此时 CPU 特权级会切换到内核态。用户程序的上下文被保存在内核栈中。这里的内核栈是每个线程独立的还是所有线程共享的？为什么？

答：内核栈是每个线程独立的。每个线程可能在执行内核代码时需要使用不同的局部变量和临时数据，存储中断处理过程中产生的临时数据和状态。独立的内核栈确保每个线程在内核

模式下执行时都有自己的执行上下文，隔离线程之间的内核执行，防止它们之间相互干扰导致数据混乱，可以支持多线程的并发执行而不会发生冲突。

10. Linux 把中断处理过程分为了上半部和下半部。

a) 这样设计有什么优点？

通过将中断处理分为两部分，只在中断发生时立刻处理上半部的紧急硬件操作，可延缓的硬件操作放到下半部分。 能够有效提高系统吞吐量， 避免在中断处理例程耗费过多时间。

b) 考虑一次中断发生后，以下操作哪些应该放在上半部，哪些应该放在下半部？请简要解释原因。

i. 读设备寄存器检查设备状态

上半部；属于需要马上进行的同步操作。

ii. 调用一些可能导致睡眠的函数

下半部；属于可能耗时的操作，不应该长时间阻塞中断处理例程

iii. 处理设备通过 DMA 发送的数据

下半部；处理数据属于耗时操作，应该通过工作队列等在下半部处理

iv. 解除中断屏蔽

下半部；上半部处理过程中应该保持中断屏蔽，而下半部可以再次被中断

11. 并行和并发程序中通常需要进行线程间的同步，请举出两个需要同步的例子。

答：举出两个合适的例子即可。

共享资源的访问：

当多个线程需要访问共享资源（如共享变量、数据结构或文件）时，为了防止数据竞争和确保数据一致性，需要进行同步。例如，考虑多个线程同时读写共享的计数器变量，需要使用同步机制（如互斥锁、信号量等）确保在任何时刻只有一个线程能够修改计数器，以防止数据不一致或丢失更新。

生产者-消费者问题：

在生产者-消费者问题中，有一个共享的缓冲区，生产者将数据放入缓冲区，而消费者从缓冲区取出数据。多个生产者和消费者线程并发执行时，需要进行同步以确保正确的数据传递和缓冲区管理。例如，使用互斥锁来保护对共享缓冲区的访问，以防止多个线程同时修改缓冲区的不一致状态。

12. 互斥锁是为了方便并行编程而抽象出来的一种同步原语。Ticket Spinlock、MCS Lock 是 常见的互斥锁实现。请简要说明他们各自的特点和适用场景

答：Ticket Spinlock 是一种基于自旋的互斥锁实现。它使用两个指针：一个用于指示当前服务号（ticket），另一个用于指示下一个服务号。线程按顺序获取服务号，如果当前服务号不是自己的服务号，线程会自旋等待直到轮到自己。优点：简单且易于实现。缺点：在高竞争情况下，可能存在多次无效的 cache line 刷新，降低性能。

MCS Lock 是一种基于链表的互斥锁实现。每个线程都拥有自己的节点，节点中保存了一个指向后继节点的指针。获取锁时，线程只需要在自己的节点上自旋等待，而释放锁时只需要修改后继节点的链表值，避免了对共享的锁变量的竞争。优点：避免了无效的 cache line 刷新，提高性能。缺点：相对于 Ticket Spinlock，实现相对复杂。

适用场景：Ticket Spinlock 适用于简单、低竞争的场景，而 MCS Lock 在高竞争时能够提供更好的性能。

13. 软链接和硬链接是在文件系统中创建引用的方式。请分别说明是否可以通过软链接或硬

链接的方式在本机目录下（例如“/home/student”）创建一个到外部存储（例如 U 盘）内文件的引用。
软链接是通过路径名字指向，可以创建。
硬链接是直接链接到同一个 inode 实体，无法跨文件系统进行链接。

14. 在 Windows 系统下的“本地磁盘(C:)”中有一个非常大的文件其路径为 “C:/os/ubuntu_20.04.iso”，当你尝试将其移动到“C:/study/”目录下时，这个操作很快就完成 了，而尝试将其移动到“U 盘(E:)”下时却需要花费很多时间，请解释发生这种现象的原因。

答：在 Windows 系统中，当将一个大文件在同一个磁盘分区中移动时，操作通常会很迅速，因为这涉及的只是元数据的更新，而不需要实际复制文件内容。然而，当将文件移动到一个外部设备时，需要真正做数据的复制和删除，所以操作时间较长。

15. Ordered Mode 和 Journaling Mode 分别是 Ext4 文件系统的两个运行模式。请简述这两种模式的流程。请从性能和崩溃一致性两个方面说明两种模式的区别。

答：Ordered Mode：文件系统首先将数据直接写入磁盘，然后将相应的元数据更新写入日志，再写入磁盘相应位置。
Journaling Mode：所有的数据和元数据写入都会先写入日志，然后再写入磁盘相应的位置。
性能：相较于 Ordered Mode，Journaling Mode 的性能开销较大，因为所有的磁盘写操作都必须在日志提交时才能进行。
崩溃一致性：Ordered Mode 只保证元数据一致性，文件数据可能丢失，而 Journaling Mode 对数据和元数据都保持一致性。

16. DMA 是一种快速传送数据的机制。DMA 传输将数据从一个地址空间复制到另外一个地址空间。CPU 负责初始化这个传输动作，而传输动作本身是由 DMA 控制器来实行和完成。最初的 DMA 地址是物理地址，但这会带来安全问题，因此出现了 IOMMU 硬件。

a) 使用 IOMMU 之后，设备看到的还是物理地址吗？设备访问内存的过程是怎样的？

答：当使用 IOMMU 时，设备看到的不是物理地址，而是虚拟地址。
设备首先发起对系统内存某个地址的访问请求（设备认为是物理地址，实际是虚拟地址）；IOMMU 会对设备访问的虚拟地址进行地址翻译，这个过程涉及将设备请求的虚拟地址映射到真正的物理地址（经过页表或其他地址转换机制）。这样就实现了设备访问内存地址中的数据，又不暴露真实的物理地址。

b) 在使用 IOMMU 之前，设备访问主机内存会有什么安全问题？IOMMU 是如何解决它 的？除了能够解决安全问题，IOMMU 提供的抽象还有什么别的好处？

答：在没有 IOMMU 的情况下，设备对系统内存的访问是采用物理地址的，缺乏有效的隔离。设备可以访问整个物理内存空间，从而增加了对系统内存的非法或不受控制的访问的风险。IOMMU 的主要目的是提供硬件级别的内存虚拟化，以增强设备访问系统内存的控制和隔离，防止设备访问未经授权的内存区域。除了安全性之外，IOMMU 使得系统对设备访问的内存进行更细粒度的权限控制；允许将物理地址映射到更大的虚拟地址空间，提高系统的可扩展性，支持更多的设备和更大的内存容量；使得设备对主机内存的访问更为独立，不受其他设备的影响，增加了系统的可靠性和稳定性。

17. 虚拟机监控器分为 Type-1 和 Type-2 两种，它们的设计各有什么优点？你能各举出两种类型虚拟机监控器的一个例子吗？

答：Type-1 VMM 直接运行在裸机上，有直接的硬件访问权限。这意味着它可以更好地利用硬件虚拟化技术，提供更高性能的虚拟化体验。由于 Type-1 VMM 不运行在操作系统之上，它能够更有效地利用系统资源，减少不必要的中间层。这通常导致更高的性能和更低的虚拟化开销。例子：VMware ESXi 是一个 Type-1 的虚拟机监控器

Type-2 VMM 运行在主机操作系统之上，允许用户在常规操作系统环境中轻松部署和管理虚拟机。这提供了更大的灵活性和便利性，特别适用于开发和测试场景。Type-2 VMM 通常不依赖硬件虚拟化支持，因此可以在不支持硬件虚拟化的系统上运行。这增加了兼容性，使得 Type-2 VMM 更适合普通用户和开发者。例子：Oracle VirtualBox 是一个 Type-2 的虚拟机监控器。

四、解答题

1. 详述给出“哲学家用餐”问题的求解过程，并用伪代码方式写出主要流程。

```
semaphore chopstick[n];

void philosopher(int i) //哲学家进程
{
  while(true)
  {
      think(); //思考问题
      //规定奇数哲学家先竞争左边的筷子，偶数哲学家竞争右边的筷子
      if(i%2==0)
      {
          //竞争筷子
          wait(chopstick[(i+1)%5]);
          wait(chopstick[i]);
          //竞争成功，用餐
          eat();
          //用餐结束，归还筷子
          ********************
          ********************
      }
      else
      {
          wait(chopstick[i]);
          wait(chopstick[(i+1)%5]);
          eat();
          signal(chopstick[(i+1)%5]);
          signal(chopstick[i]);
```

```
    }

  }
}
```

2. 写出以下操作的主体源代码（使用 Linux 内核现有的数据结构. 函数和宏定义）：在 Linux 内核中，从当前进程开始，遍历所有进程，为每一个进程分配一段内存，内存大小（以字节记）等于该进程父进程的 PID 号（例如 PID=294，则分配 294 字节内存），并让该进程的 security 字段指向该内存空间。

这里只展示主体部分：

```
static int allocate(void) {
  struct task_struct *task=current;
  unsigned long size;
  void *memblock;

  **********************
      size = ********************;
      memblock = kmalloc(size, GFP_KERNEL);
      if(!memblock){
          printk(KERN_ERR “Failed to allocate memory.\n”);
          continue;
      ***************************
      ******************** = memblock;
  }

  return 0;
}
```

3. 考虑一个处理网络请求的场景：（左为用户态程序，右为简化的内核态的网卡驱动）

```
// usermode
…
while (working) {
    read(sockfd, request, len);
    handle(request);
}
…
```

```
// kernel
void do_read() {
…
    if (packet_buffer is empty) {
        wait_event(queue);
    }
    raw_packet = get(packet_buffer);
    packet = protocol_stack(raw_packet);
    copy_to_user(userbuf, packet, len);
…
}
void handle_irq_netdev() {
    put(packet_buffer, dma_buffer);
    wake_up(queue);
}
```

面对高速网卡，上述代码难以达到最高性能，此时的性能瓶颈在什么地方？应该如何改进？

答：中断处理过程中同步处理数据，在高速网卡场景下数据体量大，导致系统在中断处理程序中停留时间长，降低响应速度，也没有利用并行性。应该在收到 netdev 中断时标注“数据已到来”就返回，继续运行和服务其它进程；后续使用软中断或工作队列处理数据，处理完成后再唤醒原进程。

4. 请分别简要描述读取磁盘时，read()系统调用在宏内核和微内核的执行流程。

宏内核：以 Linux 为例，read 系统调用触发之后，程序下陷到内核态，内核保存上下文信息到栈，调用内核中的 do_read 等函数。先经过 VFS 层权限检查，如果检查成功，调用特定文件系统实现的 read 接口，接口内部会先查询 page cache 中是否有需要内容，如果存在，拷贝对应内容到缓冲区，然后返回。如果发生 page cache miss，则需要调用磁盘驱动读取对应磁盘块内容。

微内核：read 系统调用会以 IPC 的方式将请求发送到文件系统用户进程，然后等待文件系统用户进程返回结果。在文件系统用户进程中，收到请求后会根据请求来源和内容进行权限检查，然后读取对应文件内容，以 IPC 的形式返回结果。

5. 下面的代码是 ticket spinlock 的简单实现，我们发现 lock 结构中的变量都注明了 volatile 关键字，volatile 关键字会告诉编译器不要对相关的变量进行优化。当去掉 volatile 关键字 之后，我们发现，有时候本应该轮到某个线程拿到锁，即 lock->owner 已经等于该线程拿到的 ticket，但这个线程仍然一直停留在在第 8 行的循环，无法跳出，导致整个程序无法继续进行。请从编译优化的角度，根据代码具体分析该现象出现的原因，并总结出 volatile 关键字的使用场景。

```
1  struct lock {
2    volatile int owner;
3    volatile int next;
4  };
5
6  void lock(struct lock *lock) {
7    int my_ticket = atomic_FAA(&lock->next, 1);
8    while(lock->owner != my_ticket) {}
9    barrier();
10 }
11
12 void unlock() {
13   barrier();
14   lock->owner++;
15 }
```

答：编译器在进行寄存器分配和优化时，可能将变量值存储在寄存器中而不是主内存中。这就可能导致多个线程访问同一个共享变量时，各自使用的是寄存器中的缓存值，而不是主内存中的最新值。使用 volatile 关键字可以防止编译器对变量进行这种优化，确保读写操作都

在主内存中进行。本例中，没有 volatile 关键字时编译器将 lock->owner 优化为寄存器存储和访问，导致无法获取更新后的值。

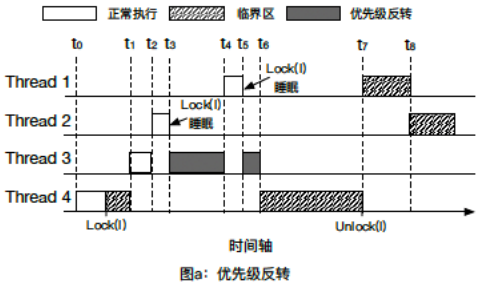
Volatile 关键字适用于：多线程环境中，寄存器缓存优化导致线程之间的共享数据修改不可见的场景。

6. 同步原语的使用不当也会导致某些问题。下面代码描述了一个银行账户转账的例子，其 使用了互斥锁来保护账户余额的修改。请从同步的角度，分析多个线程执行下面代码时 可能会出现什么问题，并提出一种解决方法。

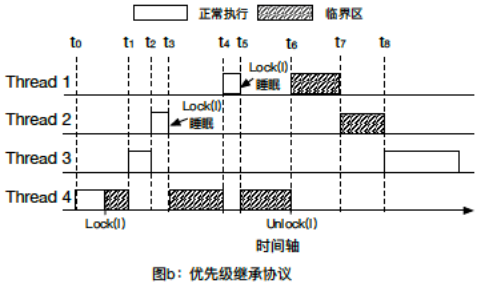
```
1  struct Account {
2    int id;
3    double balance;
4    struct lock *lock;
5  };
6
7
8  int transfer(Account *from, Account *to, double amount) {
9    int err = 0;
10
11   lock(from->lock);
12   lock(to->lock);
13
14   if (from->balance >= amount) {
15     from->balance -= amount;
16     to->balance += amount;
17   } else {
18     err = 1;
19   }
20
21   unlock(to->lock);
22   unlock(from->lock);
23
24   return err;
25 }
```

答：from->lock 和 to->lock 会死锁。假设两个账户同时互相给对方转账，都分别执行了 11 行，各自获取了自己账户的 lock。此时双方都要在 12 行等待对方释放 lock，从而引发死锁。请以伪代码形式提出解决方案。

7. 优先级反转是同步可能会导致的一类问题，它是由于同步导致线程执行顺序违反线预设优先级的问题。假设一个 CPU 核心上有 Thread1、Thread2、Thread3 和 Thread4 四个线程，其优先级分别为 1、2、3、4（数字越小，优先级越大），其中 Thread1、Thread2、Thread4 竞争同一把锁，Thread3 不申请锁。下图 a 展现了出现优先级反转的一种场景，图 b 展现了使用优先级继承协议来避免优先级反转问题的例子。请填写完成下面的表格，判断在使用优先级继承协议后，图 b 中 Thread 4 的在下表各时刻的优先级。



图a：优先级反转



图b：优先级继承协议

时刻	T4 优先级
T0	4
T1	4
T2	4
T5	1

8. 不同架构的处理器使用不同的内存模型，例如 ARM 架构使用弱一致性模型（弱一致性模型不保证任何对不同的地址的读写操作顺序），Intel x86 架构使用 TSO（Total Store Ordering）模型（TSO 模型保证对不同地址的读-读、读-写、写-写顺序，不保证写-读顺序）。下面的代码是并发哈希表的一种实现，其仅仅用锁保护了 put 操作，我们发现上述实现在某些处理器上总能正确运行，在另一些处理器上，lookup 操作有时会返回错误的值或者找不到已经 put 的值。

a) 在不考虑编译乱序的情况下，请从内存一致性的角度出发，分析两种处理器架构下出现不同现象的原因。

在 x86 处理器架构下能够正确运行，因为在新的 entry 被插入到 slot 头部之前，能够保证对 entry 的 key/value 设置已经 store 到内存。

而 arm 架构的弱一致性模型允许 CPU 对 load/store 指令进行重排，这导致了 lookup 看到的 entry 可能还没有被其他线程的 insert 设置好对应的 key/value 内容。即 insert 函数内部的 load/store 重排导致的。

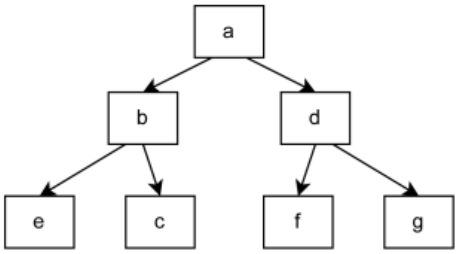
b) 请分别用添加锁（lock()和 unlock()）和添加内存屏障（barrier()）的方法修改下述代码，使得修改后的代码能在不同架构的处理器上都能够正确运行，并使得不同的读者可以同时查询同一个 bucket 中的元素。

```
1  struct entry {
2    int key;
3    int value;
4    struct entry *next;
5  };
6
7  struct entry *table[N_BUCKET];
8  struct lock *bucket_locks[N_BUCKET];
9
10 void insert(int key, int value, struct entry **p) {
11   struct entry *e = malloc(sizeof(struct entry));
12   e->key = key;
13   e->value = value;
14   e->next = *p;
15   *p = e;
16 }
17
18 void put(int key, int value) {
19   lock(&bucket_locks[key % N_BUCKET]);
20
21   struct entry **p = &table[key % N_BUCKET];
22   insert(key, value, p);
23
24   unlock(&bucket_locks[key % N_BUCKET]);
25 }
26
27 struct entry* lookup(int key) {
28   struct entry *e = table[key % N_BUCKET];
29   for (; e != NULL; e = e->next) {
30     if(e->key == key) break;
31   }
32   return e;
33 }
```

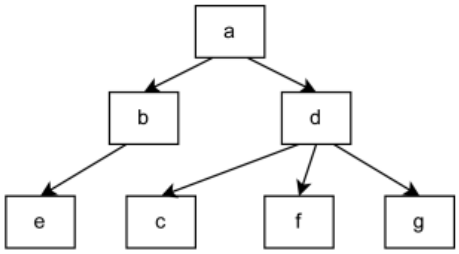
锁方式：
```
void put(int key, int value) {
    struct entry *e = malloc(sizeof(struct entry));
    e->key = key;
    e->value = value;
retry:
    e->next = table[key % N_BUCKET];
    lock(&bucket_locks[key % N_BUCKET]);
    if (e->next != table[key % N_BUCKET]) {
        unlock(&bucket_locks[key % N_BUCKET]);
        goto retry;
    }
    table[key % N_BUCKET] = e;
    unlock(&bucket_locks[key % N_BUCKET]);
}
```

barrier 方式：
```
void insert(int key, int value, struct entry **p) {
    struct entry *e = malloc(sizeof(struct entry));
    e->key = key;
    e->value = value;
    e->next = *p;
    barrier();
    *p = e;
}
```

9. Copy-on-write 是一种在 NVM 文件系统上常用的保证操作原子性的方法，请描述 Copy-on-write 原子性地完成 rename /a/b/c /a/d/c 操作的过程，将下图目录树(1)转化为(2)。



(1) Original File Tree



(2) Final File Tree

1) 复制根目录（/）、/a 目录、 /a/b 目录和/a/d 目录的元数据。
2) 在/a/b 和/a/d 元数据副本中，将/a/b/c 的信息从/a/b 目录的元数据中删除，并将其信息添加到/a/d 目录的元数据中。
3) 在/a 的元数据副本中，将/a/b 和/a/d 目录的信息，修改为指向新的元数据副本位置。
4) 在根目录的元数据副本中，将/a 目录的信息，修改为指向新的元数据副本位置。
5) 原子性地依次修改/a/b，/a/d，/a，根目录的元数据

10. 请阅读如下代码，并回答问题

```
1  int main() {
2      int fd;
3      char *addr;
4      char buf[256];
5
6      memset(buf, 0, 256);
7
8      fd = open("./b.txt", O_RDWR);
9      addr = mmap(NULL, 10, PROT_WRITE, MAP_SHARED, fd, 0);
10     printf("Before Assign:\t%s\n", (char *)addr);
11     addr[5] = 'Y';
12     printf("After Assign:\t%s\n", (char *)addr);
13     read(fd, buf, 10);
14     printf("Read From fd:\t%s\n", (char *)buf);
15
16     close(fd);
17 }
```

```
输出:
1  Before Assign:  1234567890
2  After Assign:   12345Y7890
3  Read From fd:   12345Y7890
```

a) 小雪在 ubuntu 20.04 中用 c 语言编写了如上代码并获得了如上的输出，她发现使用 mmap 将文件映

射成内存后修改，再立刻使用传统的 read 接口读取 fd 中的数据，仍 然可以获得修改后的最新数据。而她马上想到只是修改虚拟内存中的数据并不会触发 系统调用写磁盘文件，你能帮她解释为什么 read 函数仍然可以读到最新的数据吗？

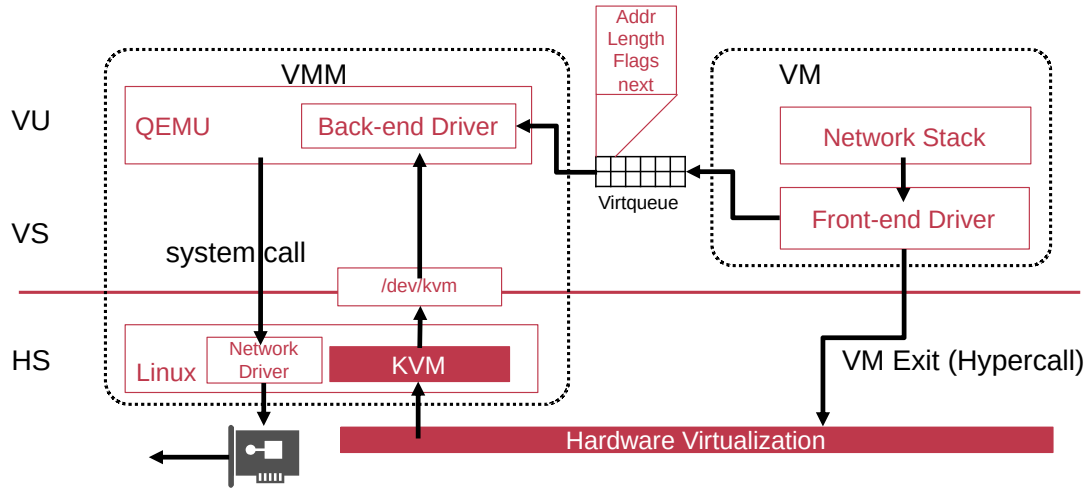
答：mmap 直接映射了页缓存中的文件数据页，修改会直接记录到页缓存中。而 read 系统调用在读取文件数据的时候，会先查看页缓存，因此能够读取到最新的还没持久化的数据。

b) 小雪随后在 ubuntu 20.04 中，尝试读取一个非常大的文件，她发现用 read 系统调用把文件读入内存需要很长的时间，但换一种方法，调用 mmap 时函数立刻就返回，并且随后马上可以在内存正常读写数据了，造成这种现象的原因是什么？

答：使用 read 系统调用读取大文件时，操作系统会将文件的数据从磁盘读取到内核的页缓存中，然后再从内核缓存中复制到用户空间。这个过程不仅需要进行两次拷贝，同时预读机制（readahead）的效果并不明显。

而在使用 mmap 时，内核并不会立即将整个文件内容加载到内存中，而是采用一种延迟加载的策略。具体地说，只有在真正访问映射区域的数据时，才会触发 PageFault 并将相应的数据加载到内存中。因此，mmap 在被调用时只需要在页表和文件描述符中做建立映射的修改，所以可以立即返回；并且此时预读机制可以发挥更好的作用，平摊读取整个大文件到内存的时间开销。

11. IO 虚拟化



上图展示了 guest 发送网络包时的传输路径(guest 和 qemu 通过 Virtuqueue 共享数据)，这种设计发送网络包的延迟很高。有人提出可以将 net-backend 放在 host kernel 中，以解决性能问题。请问为什么这种做法能够解决性能问题？

答：notification 先由 guest 发送到 KVM 再到 QEMU，有两次上下文切换，带来不必要的开销；net-backend 运行在用户态的 qemu 程序中，发送消息还需要进入 host kernel 并使用其网络栈，有一层额外的软件开销。将 net-backend 直接放在 host kernel 中，notification 只需要一次上下文切换，即从 guest 到 host kernel，减少了上下文切换的开销；net-backend 在 host kernel 中可以直接操作 NIC，避免了经过 host kernel 网络栈带来的软件开销。