1. 操作系统

1.1 Linux里如何查看一个想知道的进程?

参考回答

查看进程运行状态的指令：ps命令。“**ps -aux | grep PID**”，用来查看某PID进程状态

1.2 Linux里如何查看带有关键字的日志文件?

参考回答

1. **cat 路径/文件名 | grep 关键词**

```
# 返回test.log中包含http的所有行 cat test.log | grep "http"
```

1. **grep -i 关键词 路径/文件名** （与方法一效果相同，不同写法而已）

```
# 返回test.log中包含http的所有行(-i忽略大小写） grep -i "http" ./test.log
```

1.3 说说你对grep命令的了解?

参考回答

grep 命令。强大的文本搜索命令，grep(Global Regular Expression Print) 全局正则表达式搜索。

grep 的工作方式是这样的，它在一个或多个文件中搜索字符串模板。如果模板包括空格，则必须被引用，模板后的所有字符串被看作文件名。搜索的结果被送到标准输出，不影响原文件内容。

1.4 Linux修改主机名的命令是什么?

参考回答

1. 如果只需要临时更改主机名，可以使用hostname命令。

```
sudo hostname <new-hostname> # 例如： sudo hostname myDebian #myDebian为修改名
```

2. 如果想永久改变主机名，可以使用hostnamectl命令

```
sudo hostnamectl set-hostname myDebian #myDebian为修改名
```

1.5 Linux开机自动执行命令如何实现?

参考回答

1. **方法 #1 - 使用 cron 任务**

   除了常用格式（分 / 时 / 日 / 月 / 周）外，cron 调度器还支持 @reboot 指令。这个指令后面的参数是脚本（启动时要执行的那个脚本）的绝对路径。

   然而，这种方法需要注意两点：

   a) cron 守护进程必须处于运行状态（通常情况下都会运行），同时

   b) 脚本或 crontab 文件必须包含需要的环境变量。

2. **方法 #2 - 使用 /etc/rc.d/rc.local**

   这个方法对于 systemd-based 发行版 Linux 同样有效。不过，使用这个方法，需要授予 /etc/rc.d/rc.local 文件执行权限：

```
# chmod +x /etc/rc.d/rc.local
```

   然后在这个文件底部添加脚本。

1.6 Linux查看内存的命令是什么?

参考回答

查看内存使用情况的指令：**free命令。**“**free -m**”，命令查看内存使用情况。

查看进程运行状态、查看内存使用情况的指令均可使用**top指令。**

答案解析

1. **free命令**

   Linux free命令用于**显示内存状态。**

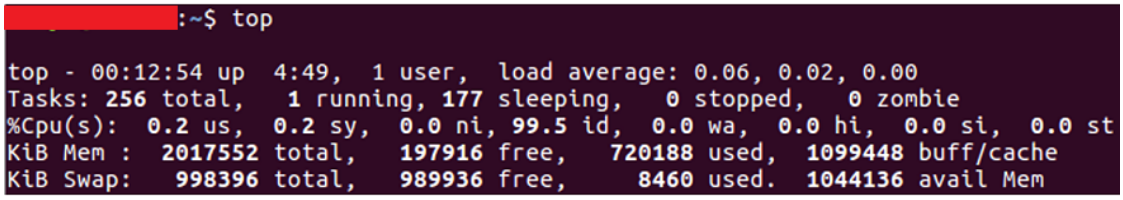
   free指令会显示内存的使用情况，**包括实体内存，虚拟的交换文件内存，共享内存区段，以及系统核心使用的缓冲区等。**

   实例：显示内存使用情况

```
# free //显示内存使用信息
total used free shared buffers cached
Mem: 254772 184568 70204 0 5692 89892
-/+ buffers/cache: 88984 165788
Swap: 524280 65116 459164
```

2. **top命令**

   top命令。**显示当前系统正在执行的进程的相关信息，包括进程 ID、内存占用率、CPU 占用率等**



   前五行是当前系统情况整体的统计信息区。

   1. 第一行，任务队列信息，同 uptime 命令的执行结果，具体参数说明情况如下：

      00:12:54 — 当前系统时间

      up ？ days, 4:49 — 系统已经运行了？天4小时49分钟（在这期间系统没有重启过）

      21users — 当前有1个用户登录系统

      load average: 0.06, 0.02, 0.00 — load average后面的三个数分别是1分钟、5分钟、15分钟的负载情况。load average数据是每隔5秒钟检查一次活跃的进程数，然后按特定算法计算出的数值。如果这个数除以逻辑CPU的数量，结果高于5的时候就表明系统在超负荷运转了。

   2. 第二行，Tasks — 任务（进程），具体信息说明如下：

      系统现在共有256个进程，其中处于运行中的有1个，177个在休眠（sleep），stoped状态的有0个，zombie状态（僵尸）的有0个。

3. 第三行，cpu状态信息，具体属性说明如下：

   0.2%us — 用户空间占用CPU的百分比。

   0.2% sy — 内核空间占用CPU的百分比。

   0.0% ni — 改变过优先级的进程占用CPU的百分比

   99.5% id — 空闲CPU百分比

   0.0% wa — IO等待占用CPU的百分比

   0.0% hi — 硬中断（Hardware IRQ）占用CPU的百分比

   0.0% si — 软中断（Software Interrupts）占用CPU的百分比

4. 第四行，内存状态，具体信息如下：

   2017552 total — 物理内存总量

   720188 used — 使用中的内存总量

   197916 free — 空闲内存总量

   1099448 cached — 缓存的总量

5. 第五行，swap交换分区信息，具体信息说明如下：

   998396 total — 交换区总量

   989936 free — 空闲交换区总量

   8460 used — 使用的交换区总量

   1044136 cached — 缓冲的交换区总量

1.7 free命令有哪些选项?

参考回答

Linux free命令用于**显示内存状态。**

free指令会显示内存的使用情况，包括实体内存，虚拟的交换文件内存，共享内存区段，以及系统核心使用的缓冲区等。

参数如下：

```
-b   以Byte为单位显示内存使用情况。
-k   以KB为单位显示内存使用情况。
-m   以MB为单位显示内存使用情况。
-h   以合适的单位显示内存使用情况，最大为三位数，自动计算对应的单位值。单位有：
        B = bytes
        K = kilos
        M = megas
        G = gigas
        T = teras
-o   不显示缓冲区调节列。
-s<间隔秒数>    持续观察内存使用状况。
-t   显示内存总和列。
-V   显示版本信息。
```

答案解析

实例：显示内存使用情况

```
# free //显示内存使用信息
total used free shared buffers cached
Mem: 254772 184568 70204 0 5692 89892
-/+ buffers/cache: 88984 165788
Swap: 524280 65116 459164
```

1.8 Linux中压缩文件的命令是什么?

参考回答

Linux中压缩文件与解压文件的命令有：**tar命令、gz命令、bz2命令、compress命令、zip命令、unzip命令。**

答案解析

1. **tar 命令详解**

   Linux tar（英文全拼：tape archive ）命令用于备份文件。

   tar 是用来建立，还原备份文件的工具程序，它可以加入，解开备份文件内的文件。

2. gz命令详解**

   Linux gzip命令用于压缩文件。

   gzip是个使用广泛的压缩程序，文件经它压缩过后，其名称后面会多出".gz"的扩展名。

3. **bz2命令详解**

   bzip2(选项)（参数）：用于创建和管理.bz2格式的压缩包。

4. **compress命令详解**

   Linux compress命令是一个相当古老的 unix 档案压缩指令，压缩后的档案会加上一个 .Z 延伸档名以区别未压缩的档案，压缩后的档案可以以 uncompress 解压。若要将数个档案压成一个压缩档，必须先将档案 tar 起来再压缩。由于 gzip 可以产生更理想的压缩比例，一般人多已改用 gzip 为档案压缩工具。

5. **zip 命令详解**

6. **unzip 命令详解**

   Linux unzip命令用于解压缩zip文件

   unzip为.zip压缩文件的解压缩程序。

1.9 Linux查询连接数的命令是什么?

参考回答

1. **netstat**

2. ```
   //示例
   查看Web服务器（Nginx Apache）的并发请求数及其TCP连接状态：
   netstat -n | awk '/^tcp/ {++S[$NF]} END {for(a in S) print a, S[a]}'

   解释：
   返回结果示例：
   LAST_ACK 5   (正在等待处理的请求数)
   SYN_RECV 30
   ESTABLISHED 1597 (正常数据传输状态)
   FIN_WAIT1 51
   FIN_WAIT2 504
   ```
   ```

```
TIME_WAIT 1057 (处理完毕，等待超时结束的请求数)

状态：描述
CLOSED：无连接是活动的或正在进行
LISTEN：服务器在等待进入呼叫
SYN_RECV：一个连接请求已经到达，等待确认
SYN_SENT：应用已经开始，打开一个连接
ESTABLISHED：正常数据传输状态
FIN_WAIT1：应用说它已经完成
FIN_WAIT2：另一边已同意释放
ITMED_WAIT：等待所有分组死掉
CLOSING：两边同时尝试关闭
TIME_WAIT：另一边已初始化一个释放
LAST_ACK：等待所有分组死掉
```

1.10 Linux中top命令有哪些参数?

参考回答

top命令。显示当前系统正在执行的进程的相关信息，包括进程 ID、内存占用率、CPU 占用率等

参数:

```
-d 指定每两次屏幕信息刷新之间的时间间隔。当然用户可以使用s交互命令来改变之。
-p 通过指定监控进程ID来仅仅监控某个进程的状态。
-q 该选项将使top没有任何延迟的进行刷新。如果调用程序有超级用户权限，那么top将以尽可能高的优先级运行。
-S 指定累计模式
-s 使top命令在安全模式中运行。这将去除交互命令所带来的潜在危险。
-i 使top不显示任何闲置或者僵死进程。
-c 显示整个命令行而不只是显示命令名
```

答案解析

```
[redacted]:~$ top

top - 00:12:54 up  4:49,  1 user,  load average: 0.06, 0.02, 0.00
Tasks: 256 total,   1 running, 177 sleeping,   0 stopped,   0 zombie
%Cpu(s):  0.2 us,  0.2 sy,  0.0 ni, 99.5 id,  0.0 wa,  0.0 hi,  0.0 si,  0.0 st
KiB Mem :  2017552 total,   197916 free,   720188 used,  1099448 buff/cache
KiB Swap:   998396 total,   989936 free,     8460 used.  1044136 avail Mem
```

前五行是当前系统情况整体的统计信息区。

1. 第一行，任务队列信息，同 uptime 命令的执行结果，具体参数说明情况如下：

   00:12:54 — 当前系统时间

   up ？ days, 4:49 — 系统已经运行了？天4小时49分钟（在这期间系统没有重启过）

   21users — 当前有1个用户登录系统

   load average: 0.06, 0.02, 0.00 — load average后面的三个数分别是1分钟、5分钟、15分钟的负载情况。load average数据是每隔5秒钟检查一次活跃的进程数，然后按特定算法计算出的数值。如果这个数除以逻辑CPU的数量，结果高于5的时候就表明系统在超负荷运转了。

2. 第二行，Tasks — 任务（进程），具体信息说明如下：

   系统现在共有256个进程，其中处于运行中的有1个，177个在休眠（sleep），stoped状态的有0个，zombie状态（僵尸）的有0个。

3. 第三行，cpu状态信息，具体属性说明如下：

0.2%us — 用户空间占用CPU的百分比。

0.2% sy — 内核空间占用CPU的百分比。

0.0% ni — 改变过优先级的进程占用CPU的百分比

99.5% id — 空闲CPU百分比

0.0% wa — IO等待占用CPU的百分比

0.0% hi — 硬中断（Hardware IRQ）占用CPU的百分比

0.0% si — 软中断（Software Interrupts）占用CPU的百分比

4. 第四行，内存状态，具体信息如下：

   2017552 total — 物理内存总量

   720188 used — 使用中的内存总量

   197916 free — 空闲内存总量

   1099448 cached — 缓存的总量

5. 第五行，swap交换分区信息，具体信息说明如下：

   998396 total — 交换区总量

   989936 free — 空闲交换区总量

   8460 used — 使用的交换区总量

   1044136 cached — 缓冲的交换区总量

1.11 Linux中，如何通过端口查进程，如何通过进程查端口?

参考回答

1. **linux下通过进程名查看其占用端口**： （1）先查看进程pid

```
ps -ef | grep 进程名
```

   （2）通过pid查看占用端口

```
netstat -nap | grep 进程pid
```

2. **linux通过端口查看进程**：

```
netstat -nap | grep 端口号
```

1.12 请你说说ping命令?

参考回答

Linux ping命令用于检测主机。

执行ping指令会使用ICMP传输协议，发出要求回应的信息，若远端主机的网络功能没有问题，就会回应该信息，因而得知该主机运作正常。

答案解析

语法：

```
ping [-dfnqrRv][-c<完成次数>][-i<间隔秒数>][-I<网络界面>][-l<前置载入>][-p<范本样式>][-s<数据包大小>][-t<存活数值>][主机名称或IP地址]
```

参数说明：

```
-d 使用Socket的SO_DEBUG功能。
-c<完成次数> 设置完成要求回应的次数。
-f 极限检测。
-i<间隔秒数> 指定收发信息的间隔时间。
-I<网络界面> 使用指定的网络接口送出数据包。
-l<前置载入> 设置在送出要求信息之前，先行发出的数据包。
-n 只输出数值。
-p<范本样式> 设置填满数据包的范本样式。
-q 不显示指令执行过程，开头和结尾的相关信息除外。
-r 忽略普通的Routing Table，直接将数据包送到远端主机上。
-R 记录路由过程。
-s<数据包大小> 设置数据包的大小。
-t<存活数值> 设置存活数值TTL的大小。
-v 详细显示指令的执行过程。
```

1.13 什么是协程?

参考回答

协程：协程是微线程，在子程序内部执行，可在子程序内部中断，转而执行别的子程序，在适当的时候再返回来接着执行。

答案解析

1. **线程与协程的区别：**

   （1）协程执行效率极高。协程直接操作栈基本没有内核切换的开销，所以上下文的切换非常快，切换开销比线程更小。

   （2）协程不需要多线程的锁机制，因为多个协程从属于一个线程，不存在同时写变量冲突，效率比线程高。

   （3）一个线程可以有多个协程。

2. **协程的优势：**

   （1）**协程调用跟切换比线程效率高**：协程执行效率极高。协程不需要多线程的锁机制，可以不加锁的访问全局变量，所以上下文的切换非常快。

   （2）**协程占用内存少**：执行协程只需要极少的栈内存（大概是4～5KB），而默认情况下，线程栈的大小为1MB。

   （3）**切换开销更少**：协程直接操作栈基本没有内核切换的开销，所以切换开销比线程少。

1.14 为什么协程比线程切换的开销小?

参考回答

（1）协程执行效率极高。协程直接操作栈基本没有内核切换的开销，所以上下文的**切换非常快**，切换开销比线程更小。

（2）协程不需要多线程的锁机制，因为多个协程从属于一个线程，不存在同时写变量冲突，效率比线程高。**避免了加锁解锁的开销。**

1.15 线程和进程的区别?

参考回答

（1）一个线程从属于一个进程；一个进程可以包含多个线程。

（2）一个线程挂掉，对应的进程挂掉；一个进程挂掉，不会影响其他进程。

（3）进程是系统资源调度的最小单位；线程CPU调度的最小单位。

（4）进程系统开销显著大于线程开销；线程需要的系统资源更少。

（5）进程在执行时拥有独立的内存单元，多个线程共享进程的内存，如代码段、数据段、扩展段；但每个线程拥有自己的栈段和寄存器组。

（6）进程切换时需要刷新TLB并获取新的地址空间，然后切换硬件上下文和内核栈，线程切换时只需要切换硬件上下文和内核栈。

（7）通信方式不一样。

（8）进程适应于多核、多机分布；线程适用于多核

1.16 进程切换为什么比线程更消耗资源?

参考回答

进程切换时需要**刷新TLB**并获取新的地址空间，然后切换硬件上下文和内核栈；线程切换时只需要切换硬件上下文和内核栈。

答案解析

进程是程序的动态表现。 一个程序进行起来后，会使用很多资源，比如使用寄存器，内存，文件等。每当切换进程时，必须要考虑保存当前进程的状态。状态包括存放在内存中的程序的代码和数据，它的栈、通用目的寄存器的内容、程序计数器、环境变量以及打开的文件描述符的集合，这个状态叫做上下文（Context）。可见，想要切换进程，保存的状态还不少。不仅如此，由于虚拟内存机制，进程切换时需要**刷新TLB**并获取新的地址空间。

线程存在于进程中，一个进程可以有一个或多个线程。**线程是运行在进程上下文中的逻辑流**，这个线程可以独立完成一项任务。同样线程有自己的上下文，包括唯一的整数线程ID， 栈、栈指针、程序计数器、通用目的寄存器和条件码。可以理解为线程上下文是进程上下文的子集。由于保存线程的上下文明显比进程的上下文小，因此系统切换线程时，必然开销更小。

1.17 介绍一下进程之间的通信。

参考回答

为了提高计算机系统的效率．增强计算机系统内各种硬件的并行操作能力．操作系统要求程序结构必须适应并发处理的需要．为此引入了进程的概念。而进程并行时，需要考虑进程间的通信，进程间通信主要有以下几种方式：匿名管道、命名管道、信号、消息队列、共享内存、信号量、Socket。

匿名管道：管道是一种半双工的通信方式，数据只能单向流动，而且只能在具有亲缘关系的进程间使用。进程的亲缘关系通常是指父子进程关系。

有名管道

匿名管道，由于没有名字，只能用于亲缘关系的进程间通信。为了克服这个缺点，提出了有名管道(FIFO)。

有名管道不同于匿名管道之处在于它提供了一个路径名与之关联，**以有名管道的文件形式存在于文件系统中**，这样，**即使与有名管道的创建进程不存在亲缘关系的进程，只要可以访问该路径，就能够彼此通过有名管道相互通信**，因此，通过有名管道不相关的进程也能交换数据。值的注意的是，有名管道严格遵循**先进先出(first in first out)** ,对匿名管道及有名管道的读总是从开始处返回数据，对它们的写则把数据添加到末尾。它们不支持诸如lseek()等文件定位操作。**有名管道的名字存在于文件系统中，内容存放在内存中。**

信号

- 信号是Linux系统中用于进程间互相通信或者操作的一种机制，信号可以在任何时候发给某一进程，而无需知道该进程的状态。
- 如果该进程当前并未处于执行状态，则该信号就有内核保存起来，知道该进程回复执行并传递给它为止。
- 如果一个信号被进程设置为阻塞，则该信号的传递被延迟，直到其阻塞被取消是才被传递给进程。

以下列出几个常用的信号：

信号	描述
SIGHUP	当用户退出终端时，由该终端开启的所有进程都退接收到这个信号，默认动作为终止进程。
SIGINT	程序终止(interrupt)信号, 在用户键入INTR字符(通常是Ctrl+C)时发出，用于通知前台进程组终止进程。
SIGQUIT	和SIGINT类似, 但由QUIT字符(通常是Ctrl+)来控制. 进程在因收到SIGQUIT退出时会产生core文件, 在这个意义上类似于一个程序错误信号。
SIGKILL	用来立即结束程序的运行. **本信号不能被阻塞、处理和忽略。**
SIGTERM	程序结束(terminate)信号, 与SIGKILL不同的是该信号可以被阻塞和处理。通常用来要求程序自己正常退出。
SIGSTOP	停止(stopped)进程的执行. 注意它和terminate以及interrupt的区别:该进程还未结束, 只是暂停执行. **本信号不能被阻塞, 处理或忽略**.

消息队列

- 消息队列是**存放在内核中的消息链表**，每个消息队列由消息队列标识符表示。
- 与管道（无名管道：只存在于内存中的文件；命名管道：存在于实际的磁盘介质或者文件系统）不同的是消息队列**存放在内核中**，只有在内核重启(即，操作系统重启)或者显示地删除一个消息队列时，该消息队列才会被真正的删除。
- 另外与管道不同的是，消息队列在某个进程往一个队列写入消息之前，并**不需要另外某个进程在该队列上等待消息的到达。**

> **消息队列特点总结：**
>
> （1）消息队列是消息的链表,具有特定的格式,存放在内存中并由消息队列标识符标识.
>
> （2）消息队列允许一个或多个进程向它写入与读取消息.
>
> （3）管道和消息队列的通信数据都是先进先出的原则。
>
> （4）消息队列可以实现消息的随机查询,消息不一定要以先进先出的次序读取,也可以按消息的类型读取.比FIFO更有优势。

> （5）消息队列克服了信号承载信息量少，管道只能承载无格式字 节流以及缓冲区大小受限等缺。
>
> （6）目前主要有两种类型的消息队列：POSIX消息队列以及System V消息队列，系统V消息队列目前被大量使用。系统V消息队列是随内核持续的，只有在内核重起或者人工删除时，该消息队列才会被删除。

共享内存

进程间本身的内存是相互隔离的，而共享内存机制相当于给两个进程开辟了一块二者均可访问的内存空间，这时，两个进程便可以共享一些数据了。但是，多进程同时占用资源会带来一些意料之外的情况，这时，我们往往会采用上述的信号量来控制多个进程对共享内存空间的访问。

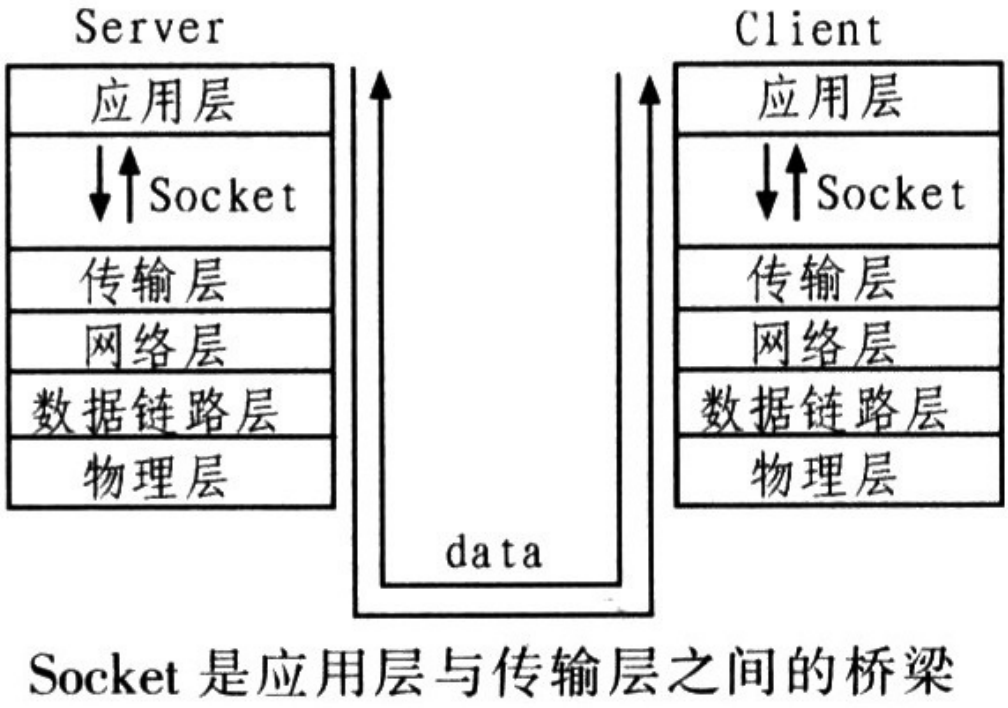
信号量

信号量主要用来解决进程和线程间并发执行时的同步问题，进程同步是并发进程为了完成共同任务采用某个条件来协调他们的活动，这是进程之间发生的一种直接制约关系。

对信号量的操作分为P操作和V操作，P操作是将信号量的值减一，V操作是将信号量的值加一。当信号量的值小于等于0之后，再进行P操作时，当前进程或线程会被阻塞，直到另一个进程或线程执行了V操作将信号量的值增加到大于0之时。锁也是用的这种原理实现的。

信号量我们需要定义信号量的数量，设定初始值，以及决定何时进行PV操作。

1. socket



Socket 是应用层与传输层之间的桥梁

套接字可以看做是：不同主机之间的进程进行双向通信的端点。（套接字 = IP地址 + 端口号）

1.18 介绍一下信号量。

参考回答

1. 在多进程环境下，为了防止多个进程同时访问一个公共资源而出现问题，需要一种方法来协调各个进程，保证它们能够合理地使用公共资源。信号量就是这样一种机制。

   信号量的数据类型为结构sem_t，它本质上是一个长整型的数。函数sem_init（）用来初始化一个信号量。它的原型为：

   extern int sem_init *P ((sem_t *sem, int* _pshared, unsigned int _*value));

sem为指向信号量结构的一个指针；pshared不为 0 时此信号量在进程间共享，否则只能为当前进程的所有线程共享；value给出了信号量的初始值。

（1）函数sem_post(sem_t *sem)用来增加信号量的值。当有线程阻塞在这个信号量上时，调用这个函数会使其中的一个线程不在阻塞，选择机制同样是由线程的调度策略决定的。

（2）函数sem_wait(sem_t *sem)被用来阻塞当前线程直到信号量sem的值大于0，解除阻塞后将sem的值减一，表明公共资源经使用后减少。函数sem_trywait (sem_t *sem)是函数sem_wait（）的非阻塞版本，它直接将信号量sem的值减一。

（3）函数sem_timedwait(sem_t *sem, const struct timespec *abs_timeout) 与 sem_wait() 类似，只不过 abs_timeout 指定一个阻塞的时间上限，如果调用因不能立即执行递减而要阻塞。

（4）函数sem_destroy(sem_t *sem)用来释放信号量sem。

1.19 说说僵尸进程和孤儿进程。

参考回答

1. 我们知道在unix/linux中，正常情况下，子进程是通过父进程创建的，子进程在创建新的进程。子进程的结束和父进程的运行是一个异步过程,即父进程永远无法预测子进程 到底什么时候结束。 当一个 进程完成它的工作终止之后，它的父进程需要调用wait()或者waitpid()系统调用取得子进程的终止状态。
2. 孤儿进程：一个父进程退出，而它的一个或多个子进程还在运行，那么那些子进程将成为孤儿进程。孤儿进程将被init进程(进程号为1)所收养，并由init进程对它们完成状态收集工作。
3. 僵尸进程：一个进程使用fork创建子进程，如果子进程退出，而父进程并没有调用wait或waitpid获取子进程的状态信息，那么子进程的进程描述符仍然保存在系统中。这种进程称之为僵尸进程。

1.21 请介绍线程之间的通信方式。

参考回答

1. 锁机制：包括互斥锁、条件变量、读写锁互斥锁提供了以排他方式防止数据结构被并发修改的方法。读写锁允许多个线程同时读共享数据，而对写操作是互斥的。条件变量可以以原子的方式阻塞进程，直到某个特定条件为真为止。对条件的测试是在互斥锁的保护下进行的。条件变量始终与互斥锁一起使用。
2. 信号量机制(Semaphore)：包括无名线程信号量和命名线程信号量
3. 信号机制(Signal)：类似进程间的信号处理线程间的通信目的主要是用于线程同步，所以线程没有像进程通信中的用于数据交换的通信机制。

1.22 说一说进程的状态。

参考回答

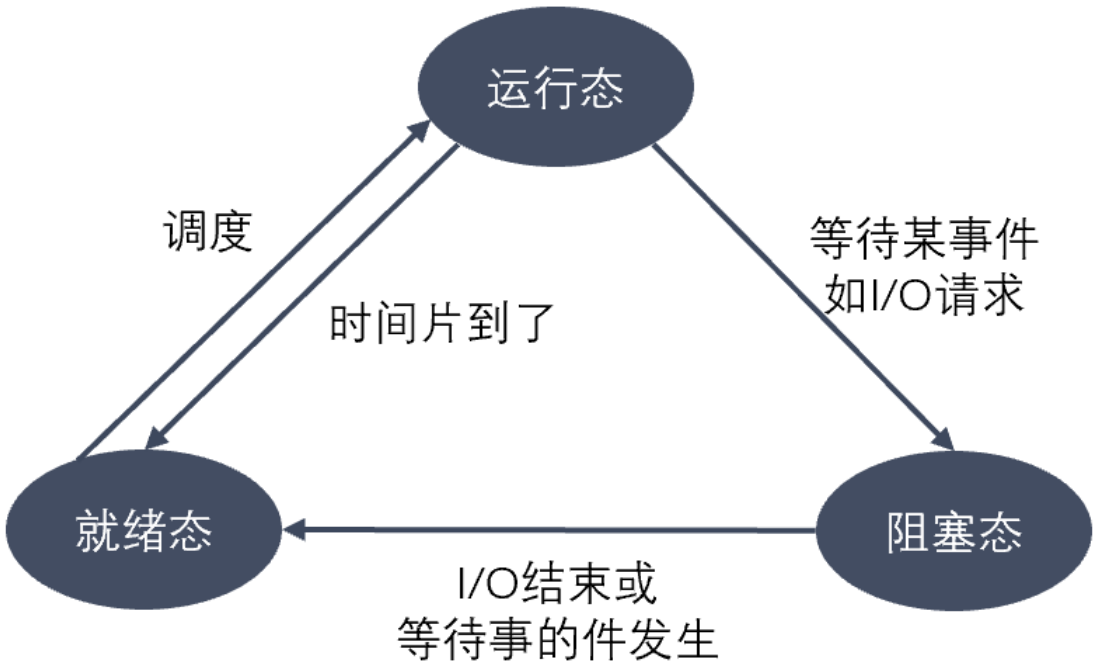
1. 进程的3种基本状态：**运行、就绪和阻塞。**

（1）就绪：当一个进程获得了除处理机以外的一切所需资源，一旦得到处理机即可运行，则称此进程处于就绪状态。就绪进程可以按多个优先级来划分队列。例如，当一个进程由于时间片用完而进入就绪状态时，排入低优先级队列；当进程由I / O操作完成而进入就绪状态时，排入高优先级队列。

（2）运行：当一个进程在处理机上运行时，则称该进程处于运行状态。处于此状态的进程的数目小于等于处理器的数目，对于单处理机系统，处于运行状态的进程只有一个。在没有其他进程可以执行时（如所有进程都在阻塞状态），通常会自动执行系统的空闲进程。

（3）阻塞：也称为等待或睡眠状态，一个进程正在等待某一事件发生（例如请求I/O而等待I/O完成等）而暂时停止运行，这时即使把处理机分配给进程也无法运行，故称该进程处于阻塞状态。

其转移图如下：



1. 进程的五种状态

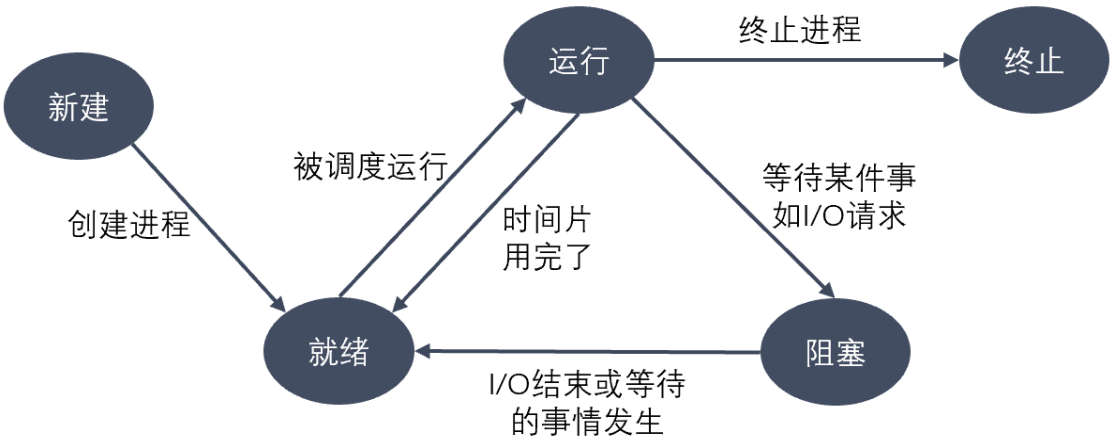
创建状态：进程在创建时需要申请一个空白PCB，向其中填写控制和管理进程的信息，完成资源分配。如果创建工作无法完成，比如资源无法满足，就无法被调度运行，把此时进程所处状态称为创建状态

就绪状态：进程已经准备好，已分配到所需资源，只要分配到CPU就能够立即运行

执行状态：进程处于就绪状态被调度后，进程进入执行状态

阻塞状态：正在执行的进程由于某些事件（I/O请求，申请缓存区失败）而暂时无法运行，进程受到阻塞。在满足请求时进入就绪状态等待系统调用

终止状态：进程结束，或出现错误，或被系统终止，进入终止状态。无法再执行



1.23 CPU调度的最小单位是什么？线程需要CPU调度吗？

参考回答

1. 进程是CPU分配资源的最小单位，线程是CPU调度的最小单位。
2. 线程是比进程更小的能独立运行的基本单位，需要通过CPU调度来切换上下文，达到并发的目的。

1.24 进程之间共享内存的通信方式有什么好处？

参考回答

采用共享内存通信的一个显而易见的好处是效率高，因为进程可以直接读写内存，而不需要任何数据的拷贝。对于像管道和消息队列等通信方式，则需要在内核和用户空间进行四次的数据拷贝，而共享内存则只拷贝两次数据：一次从输入文件到共享内存区，另一次从共享内存区到输出文件。

实际上，进程之间在共享内存时，并不总是读写少量数据后就解除映射，有新的通信时，再重新建立共享内存区域。而是保持共享区域，直到通信完毕为止，这样，数据内容一直保存在共享内存中，并没有写回文件。共享内存中的内容往往是在解除映射时才写回文件的。因此，采用共享内存的通信方式效率是非常高的。

1.25 如何杀死一个进程？

参考回答

1. 杀死父进程并不会同时杀死子进程：每个进程都有一个父进程。可以使用 pstree 或 ps 工具来观察这一点。

   调用 ps 命令可以显示 PID（进程 ID） 和 PPID（父进程 ID）。

   杀死父进程后，子进程将会成为孤儿进程，而 init 进程将重新成为它的父进程。

2. 杀死进程组或会话中的所有进程

   ```
   $ kill -SIGTERM -- -19701
   ```

   这里用一个负数 -19701 向进程组发送信号。如果传递的是一个正数，这个数将被视为进程 ID 用于终止进程。如果传递的是一个负数，它被视为 PGID，用于终止整个进程组。负数来自系统调用的直接定义。

   杀死会话中的所有进程与之完全不同。即使是具有会话 ID 的系统，例如 Linux，也没有提供系统调用来终止会话中的所有进程。需要遍历 /proc 输出的进程树，收集所有的 SID，然后一一终止进程。

   Pgrep 实现了遍历、收集并通过会话 ID 杀死进程的算法。可以使用以下命令：

   ```
   pkill -s <SID>
   ```

1.26 说一说kill的原理。

参考回答

- kill 命令的执行原理是这样的，kill 命令会向操作系统内核发送一个信号（多是终止信号）和目标进程的 PID，然后系统内核根据收到的信号类型，对指定进程进行相应的操作。kill 命令的基本格式如下：

  [root\@localhost ~]# kill [信号] PID

- kill 命令是按照 PID 来确定进程的，所以 kill 命令只能识别 PID，而不能识别进程名。

- kill 命令只是“发送”一个信号，因此，只有当信号被程序成功“捕获”，系统才会执行 kill 命令指定的操作；反之，如果信号被“封锁”或者“忽略”，则 kill 命令将会失效。

1.27 介绍下你知道的锁。

参考回答

悲观锁

悲观锁并不是某一个锁，是一个锁类型，无论是否并发竞争资源，都会锁住资源，并等待资源释放下一个线程才能获取到锁。 这明显很悲观，所以就叫悲观锁。这明显可以归纳为一种策略，只要符合这种策略的锁的具体实现，都是悲观锁的范畴。

乐观锁

与悲观锁相对的，乐观锁也是一个锁类型。当线程开始竞争资源时，不是立马给资源上锁，而是进行一些前后值比对，以此来操作资源。例如常见的CAS操作，就是典型的乐观锁。示例如下

```
int cas(long *addr, long old, long new) {
    /* 原子执行 */
    if(*addr != old)
        return 0;
    *addr = new;
    return 1;
}
```

自旋锁

自旋锁是一种基础的同步原语，用于保障对共享数据的互斥访问。与互斥锁的相比，在获取锁失败的时候不会使得线程阻塞而是一直自旋尝试获取锁。当线程等待自旋锁的时候，CPU不能做其他事情，而是一直处于轮询忙等的状态。

自旋锁主要适用于被持有时间短，线程不希望在重新调度上花过多时间的情况。实际上许多其他类型的锁在底层使用了自旋锁实现，例如多数互斥锁在试图获取锁的时候会先自旋一小段时间，然后才会休眠。如果在持锁时间很长的场景下使用自旋锁，则会导致CPU在这个线程的时间片用尽之前一直消耗在无意义的忙等上，造成计算资源的浪费。

公平锁

多个线程竞争同一把锁，如果依照先来先得的原则，那么就是一把公平锁。

非公平锁

多个线程竞争锁资源，抢占锁的所有权。

共享锁

多个线程可以共享这个锁的拥有权。一般用于数据的读操作，防止数据被写修改。共享锁的代码示例如下：

- 如果一个线程已经获取了*共享锁*，则其他任何线程都无法获取*互斥锁*，但是可以获取*共享锁*。
- 从这个输出可以看出，验证了如果一个线程已经获取了*互斥锁*，则其他线程都无法获取该锁。

死锁

死锁是指两个或两个以上的进程在执行过程中，由于竞争资源或者由于彼此通信而造成的一种阻塞的现象，若无外力作用，它们都将无法推进下去。此时称系统处于死锁状态或系统产生了死锁，这些永远在互相等待的进程称为死锁进程。

1.28 什么情况下会产生死锁?

参考回答

如果在计算机系统中同时具备下面四个必要条件时，那么会发生死锁。换句话说，只要下面四个条件有一个不具备，系统就不会出现死锁。

1. 互斥条件。即某个资源在一段时间内只能由一个进程占有，不能同时被两个或两个以上的进程占有。这种独占资源如CD-ROM驱动器，打印机等等，必须在占有该资源的进程主动释放它之后，其它进程才能占有该资源。这是由资源本身的属性所决定的。如独木桥就是一种独占资源，两方的人不能同时过桥。

   代码中有一个全局变量①，这个全局变量被一个全局的互斥量保护②。add_to_list()③和list_contains()④函数中使用std::lock_guardstd::mutex，使得这两个函数中对数据的访问是互斥的：list_contains()不可能看到正在被add_to_list()修改的列表。

2. 不剥夺条件。进程所获得的资源在未使用完毕之前，资源申请者不能强行地从资源占有者手中夺取资源，而只能由该资源的占有者进程自行释放。如过独木桥的人不能强迫对方后退，也不能非法地将对方推下桥，必须是桥上的人自己过桥后空出桥面（即主动释放占有资源），对方的人才能过桥。

3. 请求和保持条件。进程至少已经占有一个资源，但又申请新的资源；由于该资源已被另外进程占有，此时该进程阻塞；但是，它在等待新资源之时，仍继续占用已占有的资源。还以过独木桥为例，甲乙两人在桥上相遇。甲走过一段桥面（即占有了一些资源），还需要走其余的桥面（申请新的资源），但那部分桥面被乙占有（乙走过一段桥面）。甲过不去，前进不能，又不后退；乙也处于同样的状况。

4. 循环等待条件。存在一个进程等待序列{P1，P2，...，Pn}，其中P1等待P2所占有的某一资源，P2等待P3所占有的某一源，.....，而Pn等待P1所占有的的某一资源，形成一个进程循环等待环。就像前面的过独木桥问题，甲等待乙占有的桥面，而乙又等待甲占有的桥面，从而彼此循环等待。

   上述过程可能导致在2处上锁，然后子线程在1处发生阻塞，最后主线程在3处一直等待子线程结束，无穷等待下去。 上面提到的这四个条件在死锁时会同时发生。也就是说，只要有一个必要条件不满足，则死锁就可以排除。

1.29 说一说你对自旋锁的理解。

参考回答

旋锁的定义：当一个线程尝试去获取某一把锁的时候，如果这个锁此时已经被别人获取(占用)，那么此线程就无法获取到这把锁，该线程将会等待，间隔一段时间后会再次尝试获取。这种采用循环加锁 -> 等待的机制被称为自旋锁(spinlock)。

自旋锁有以下特点

- 用于临界区互斥
- 在任何时刻最多只能有一个执行单元获得锁
- 要求持有锁的处理器所占用的时间尽可能短
- 等待锁的线程进入忙循环

自旋锁存在的问题

- 如果某个线程持有锁的时间过长，就会导致其它等待获取锁的线程进入循环等待，消耗CPU。使用不当会造成CPU使用率极高。
- 无法满足等待时间最长的线程优先获取锁。不公平的锁就会存在“线程饥饿”问题。

自旋锁的优点

- 自旋锁不会使线程状态发生切换，一直处于用户态，即线程一直都是active的；不会使线程进入阻塞状态，减少了不必要的上下文切换，执行速度快
- 非自旋锁在获取不到锁的时候会进入阻塞状态，从而进入内核态，当获取到锁的时候需要从内核态恢复，需要线程上下文切换。（线程被阻塞后便进入内核（Linux）调度状态，这个会导致系统在用户态与内核态之间来回切换，严重影响锁的性能）

自旋锁与互斥锁的区别

- 自旋锁与互斥锁都是为了实现保护资源共享的机制。
- 无论是自旋锁还是互斥锁，在任意时刻，都最多只能有一个保持者。
- 获取互斥锁的线程，如果锁已经被占用，则该线程将进入睡眠状态；获取自旋锁的线程则不会睡眠，而是一直循环等待锁释放。

1.30 说一说你对悲观锁的理解。

参考回答

悲观锁总是假设最坏的情况，每次去拿数据的时候都认为别人会修改，所以每次在拿数据的时候都会上锁，这样别人想拿这个数据就会阻塞直到它拿到锁（**共享资源每次只给一个线程使用，其它线程阻塞，用完后再把资源转让给其它线程**）。传统的关系型数据库里边就用到了很多这种锁机制，比如行锁，表锁等，读锁，写锁等，都是在做操作之前先上锁。

1.31 说一说你对乐观锁的理解。

参考回答

乐观锁总是假设最好的情况，每次去拿数据的时候都认为别人不会修改，所以不会上锁，但是在更新的时候会判断一下在此期间别人有没有去更新这个数据，可以使用版本号机制和CAS算法实现。**乐观锁适用于多读的应用类型，这样可以提高吞吐量**，像数据库提供的类似于**write_condition机制**，其实都是提供的乐观锁。

1.32 CAS在什么地方用到过吗?

参考回答

- CAS是英文单词**CompareAndSwap**的缩写，中文意思是：比较并替换。CAS需要有3个操作数：内存地址V，旧的预期值A，即将要更新的目标值B。CAS指令执行时，当且仅当内存地址V的值与预期值A相等时，将内存地址V的值修改为B，否则就什么都不做。整个比较并替换的操作是一个原子操作。
- 高并发环境下，对同一个数据的**并发读**（两边都读出余额是100）与**并发写**（一个写回28，一个写回38）导致的数据一致性问题。

  解决方案是在set写回的时候，加上初始状态的条件compare，只有初始状态不变时，才允许set写回成功，这是一种常见的降低读写锁冲突，保证数据一致性的方法。

1.33 谈谈IO多路复用。

参考回答

1. IO多路复用是一种同步IO模型，实现一个线程可以监视多个文件句柄；一旦某个文件句柄就绪，就能够通知应用程序进行相应的读写操作；没有文件句柄就绪时会阻塞应用程序，交出cpu。多路是指网络连接，复用指的是同一个线程。
2. IO多路复用有三种实现方式:select, poll, epoll

(1)select：时间复杂度O(n)，它仅仅知道了，有I/O事件发生了，却并不知道是哪那几个流（可能有一个，多个，甚至全部），只能无差别轮询所有流，找出能读出数据，或者写入数据的流，对他们进行操作。所以**select具有O(n)的无差别轮询复杂度**，同时处理的流越多，无差别轮询时间就越长。

```
 int select (int n, fd_set *readfds, fd_set *writefds, fd_set
*exceptfds, struct timeval *timeout);
```

select 函数监视的文件描述符分3类，分别是writefds、readfds、和exceptfds。调用后select函数会阻塞，直到有描述副就绪（有数据 可读、可写、或者有except），或者超时（timeout指定等待时间，如果立即返回设为null即可），函数返回。当select函数返回后，可以 通过遍历fdset，来找到就绪的描述符。

select目前几乎在所有的平台上支持，其良好跨平台支持也是它的一个优点。select的一 个缺点在于单个进程能够监视的文件描述符的数量存在最大限制，在Linux上一般为1024，可以通过修改宏定义甚至重新编译内核的方式提升这一限制，但是这样也会造成效率的降低。

(2)poll：时间复杂度O(n)，poll本质上和select没有区别，它将用户传入的数组拷贝到内核空间，然后查询每个fd对应的设备状态， **但是它没有最大连接数的限制**，原因是它是基于链表来存储的。

```
int poll (struct pollfd *fds, unsigned int nfds, int timeout);
```

不同与select使用三个位图来表示三个fdset的方式，poll使用一个 pollfd的指针实现。

```
struct pollfd {
    int fd;
    /* file descriptor */
    short events;
    /* requested events to watch */
    short revents;
    /* returned events witnessed */
};
```

pollfd结构包含了要监视的event和发生的event，不再使用select“参数-值”传递的方式。同时，pollfd并没有最大数量限制（但是数量过大后性能也是会下降）。 和select函数一样，poll返回后，需要轮询pollfd来获取就绪的描述符。

(3)epoll：时间复杂度O(1)，**epoll可以理解为event poll**，不同于忙轮询和无差别轮询，epoll会把哪个流发生了怎样的I/O事件通知我们。所以说epoll实际上是**事件驱动（每个事件关联上fd）** 的，此时对这些流的操作都是有意义的。

epoll操作过程需要三个接口，分别如下：

```
int epoll_create(int size);
//创建一个epoll的句柄，size用来告诉内核这个监听的数目一共有多大
int epoll_ctl(int epfd, int op, int fd, struct epoll_event *event);
int epoll_wait(int epfd, struct epoll_event * events, int maxevents, int
timeout);
```

- int epoll_create(int size)：创建一个epoll的句柄，size用来告诉内核这个监听的数目一共有多大，这个参数不同于select()中的第一个参数，给出最大监听的fd+1的值，参数size并不是限制了epoll所能监听的描述符最大个数，只是对内核初始分配内部数据结构的一个建议。当创建好epoll

句柄后，它就会占用一个fd值，在linux下如果查看/proc/进程id/fd/，是能够看到这个fd的，所以在使用完epoll后，必须调用close()关闭，否则可能导致fd被耗尽。

- int epoll_ctl(int epfd, int op, int fd, struct epoll_event *event)：函数是对指定描述符fd执行op操作。- epfd：是epoll_create()的返回值。- op：表示op操作，用三个宏来表示：添加EPOLL_CTL_ADD，删除EPOLL_CTL_DEL，修改EPOLL_CTL_MOD。分别添加、删除和修改对fd的监听事件。- fd：是需要监听的fd（文件描述符）- epoll_event：是告诉内核需要监听什么事，
- int epoll_wait(int epfd, struct epoll_event * events, int maxevents, int timeout)：等待epfd上的io事件，最多返回maxevents个事件。参数events用来从内核得到事件的集合，maxevents告之内核这个events有多大，这个maxevents的值不能大于创建epoll_create()时的size，参数timeout是超时时间（毫秒，0会立即返回，-1将不确定，也有说法说是永久阻塞）。该函数返回需要处理的事件数目，如返回0表示已超时。

1. select、poll、epoll区别

- 支持一个进程所能打开的最大连接数

select	单个进程所能打开的最大连接数有FD_SETSIZE宏定义，其大小是32个整数的大小（在32位的机器上，大小就是32*32，同理64位机器上FD_SETSIZE为32*64），当然我们可以对进行修改，然后重新编译内核，但是性能可能会受到影响，这需要进一步的测试。
poll	poll本质上和select没有区别，但是它没有最大连接数的限制，原因是它是基于链表来存储的
epoll	虽然连接数有上限，但是很大，1G内存的机器上可以打开10万左右的连接，2G内存的机器可以打开20万左右的连接

- FD剧增后带来的IO效率问题

select	因为每次调用时都会对连接进行线性遍历，所以随着FD的增加会造成遍历速度慢的“线性下降性能问题”。
poll	同上
epoll	因为epoll内核中实现是根据每个fd上的callback函数来实现的，只有活跃的socket才会主动调用callback，所以在活跃socket较少的情况下，使用epoll没有前面两者的线性下降的性能问题，但是所有socket都很活跃的情况下，可能会有性能问题。

- 消息传递方式

select	内核需要将消息传递到用户空间，都需要内核拷贝动作
poll	同上
epoll	epoll通过内核和用户空间共享一块内存来实现的。

1.34 谈谈poll和epoll的区别。

参考回答

1. poll将用户传入的数组拷贝到内核空间，然后查询每个fd对应的设备状态，如果设备就绪则在设备等待队列中加入一项并继续遍历，如果遍历完所有fd后没有发现就绪设备，则挂起当前进程，直到设备就绪或者主动超时，被唤醒后它又要再次遍历fd。这个过程经历了多次无谓的遍历。

   它没有最大连接数的限制，原因是它是基于链表来存储的，但是同样有缺点：

   1. 大量的fd的数组被整体复制于用户态和内核地址空间之间，而不管这样的复制是不是有意义。
   2. poll还有一个特点是“**水平触发**”，如果报告了fd后，没有被处理，那么下次poll时会再次报告该fd。
2. epoll是在2.6内核中提出的，是之前的select和poll的增强版本。相对于select和poll来说，epoll更加灵活，没有描述符限制。epoll使用一个文件描述符管理多个描述符，将用户关系的文件描述符的事件存放到内核的一个事件表中，这样在用户空间和内核空间的copy只需一次。

   epoll支持水平触发和边缘触发，最大的特点在于边缘触发，它只告诉进程哪些fd刚刚变为就绪态，并且只会通知一次。还有一个特点是，epoll使用“事件”的就绪通知方式，通过epoll_ctl注册fd，一旦该fd就绪，内核就会采用类似callback的回调机制来激活该fd，epoll_wait便可以收到通知。其优点有：

   1. 没有最大并发连接的限制，能打开的FD的上限远大于1024（1G的内存上能监听约10万个端口）。
   2. 效率提升，不是轮询的方式，不会随着FD数目的增加效率下降。只有活跃可用的FD才会调用callback函数；即Epoll最大的优点就在于它只管你“活跃”的连接，而跟连接总数无关，因此在实际的网络环境中，Epoll的效率就会远远高于select和poll。
   3. 内存拷贝，利用mmap()文件映射内存加速与内核空间的消息传递；即epoll使用mmap减少复制开销。

1.35 谈谈select和epoll的区别。

参考回答

1. select 函数监视的文件描述符分3类，分别是writefds、readfds、和exceptfds。调用后select函数会阻塞，直到有描述符就绪（有数据 可读、可写、或者有except），或者超时（timeout指定等待时间，如果立即返回设为null即可），函数返回。当select函数返回后，可以通过遍历fdset，来找到就绪的描述符。

   select目前几乎在所有的平台上支持，其良好跨平台支持也是它的一个优点。select的一个缺点在于单个进程能够监视的文件描述符的数量存在最大限制，在Linux上一般为1024，可以通过修改宏定义甚至重新编译内核的方式提升这一限制，但是这样也会造成效率的降低。

   内核需要传递消息到用户空间，需要内存拷贝

2. 相对于select和poll来说，epoll更加灵活，没有描述符限制。epoll使用一个文件描述符管理多个描述符，将用户关系的文件描述符的事件存放到内核的一个事件表中，这样在用户空间和内核空间的copy只需一次。

   epoll能打开的FD的上限远大于1024（1G的内存上能监听约10万个端口）。

效率提升，不是轮询的方式，不会随着FD数目的增加效率下降。只有活跃可用的FD才会调用callback函数；即Epoll最大的优点就在于它只管你“活跃”的连接，而跟连接总数无关，因此在实际的网络环境中，Epoll的效率就会远远高于select和poll。

1.36 epoll有哪两种模式?

参考回答

epoll对文件描述符的操作有两种模式：LT（level trigger）和ET（edge trigger）。LT模式是默认模式，LT模式与ET模式的区别如下：

LT模式：当epoll_wait检测到描述符事件发生并将此事件通知应用程序，应用程序可以不立即处理该事件。下次调用epoll_wait时，会再次响应应用程序并通知此事件。

ET模式：当epoll_wait检测到描述符事件发生并将此事件通知应用程序，应用程序必须立即处理该事件。如果不处理，下次调用epoll_wait时，不会再次响应应用程序并通知此事件。

1. **LT模式（水平模式）**

LT(level triggered)是缺省的工作方式，并且同时支持block和no-block socket。在这种做法中，内核告诉你一个文件描述符是否就绪了，然后你可以对这个就绪的fd进行IO操作。如果你不作任何操作，内核还是会继续通知你的。

1. **ET模式（边缘模式）**

ET(edge-triggered)是高速工作方式，只支持no-block socket。在这种模式下，当描述符从未就绪变为就绪时，内核通过epoll告诉你。然后它会假设你知道文件描述符已经就绪，并且不会再为那个文件描述符发送更多的就绪通知，直到你做了某些操作导致那个文件描述符不再为就绪状态了(比如，你在发送，接收或者接收请求，或者发送接收的数据少于一定量时导致了一个EWOULDBLOCK 错误）。但是请注意，如果一直不对这个fd作IO操作(从而导致它再次变成未就绪)，内核不会发送更多的通知(only once)。

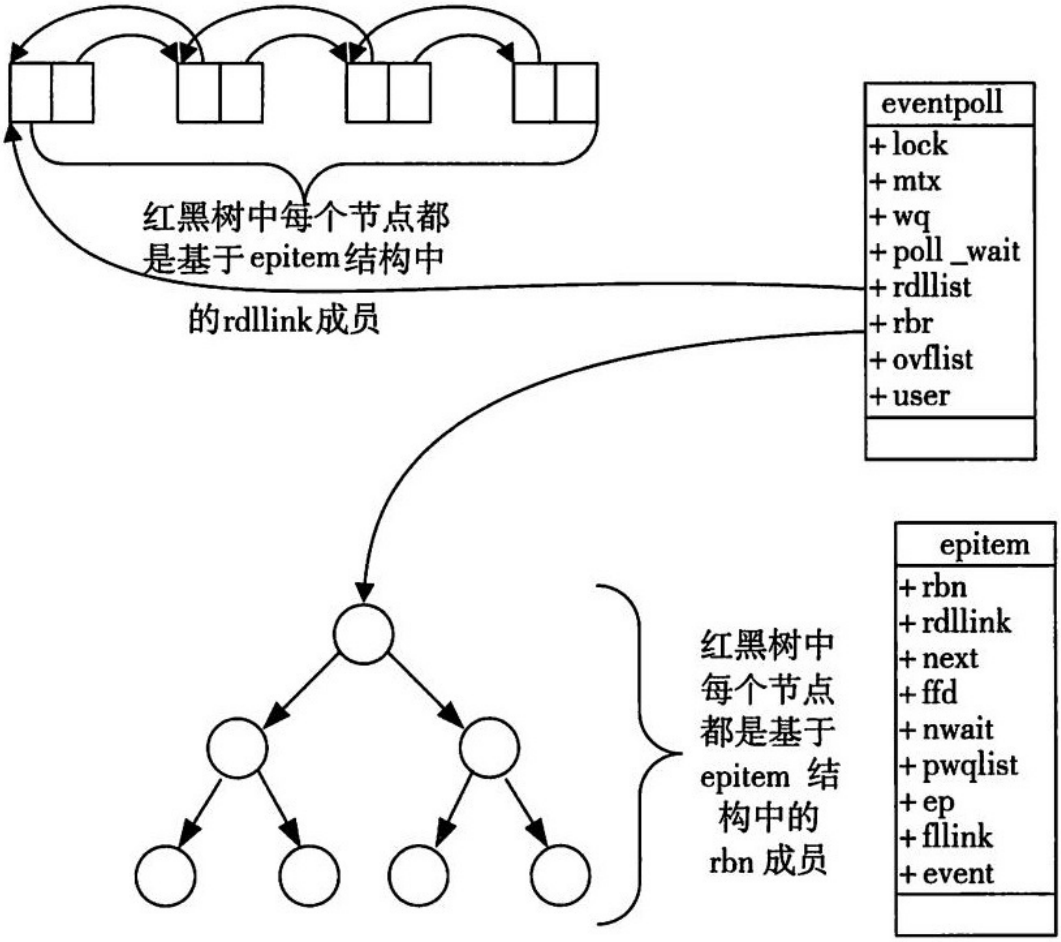
ET模式在很大程度上减少了epoll事件被重复触发的次数，因此效率要比LT模式高。epoll工作在ET模式的时候，必须使用非阻塞套接口，以避免由于一个文件句柄的阻塞读/阻塞写操作把处理多个文件描述符的任务饿死。

1.37 说一下epoll的原理，它的查询速度是O(1)的吗?

参考回答

epoll是一种更加高效的IO多路复用的方式，不同于忙轮询和无差别轮询，epoll会把哪个流发生了怎样的I/O事件通知我们。时间复杂度为O(1)。

epoll的执行过程如图所示：



1. 创建红黑树，调用epoll_create()创建一颗空的红黑树，用于存放FD及其感兴趣事件；
2. 注册感兴趣事件，调用epoll_ctl()向红黑树中添加节点（FD及其感兴趣事件），时间复杂度O(logN)，向内核的中断处理程序注册一个回调函数，告诉内核，如果这个句柄的中断到了，就把它添加到就绪队列中。所以，当一个socket上有数据到了，内核在把网卡上的数据copy到内核中后就来把socket插入到就绪队列中了；
3. 获取就绪事件，调用epoll_wait()返回就绪队列中的就绪事件，时间复杂度O(1)；

1.38 介绍域名解析成IP的全过程。

参考回答

第一步：检查浏览器缓存中是否缓存过该域名对应的IP地址

用户通过浏览器浏览过某网站之后，浏览器就会自动缓存该网站域名对应的地址，当用户再次访问的时候，浏览器就会从缓存中查找该域名对应的IP地址，因为缓存不仅是有大小限制，而且还有时间限制（域名被缓存的时间通过属性来设置），所以存在域名对应的找不到的情况。当浏览器从缓存中找到了该网站域名对应的地址，那么整个解析过程结束，如果没有找到，将进行下一步骤。对于的缓存时间问题，不宜设置太长的缓存时间，时间太长，如果域名对应的发生变化，那么用户将在一段时间内无法正常访问到网站，如果太短，那么又造成频繁解析域名。

第二步：如果在浏览器缓存中没有找到IP，那么将继续查找本机系统是否缓存过IP

如果第一个步骤没有完成对域名的解析过程，那么浏览器会去系统缓存中查找系统是否缓存过这个域名对应的地址，也可以理解为系统自己也具备域名解析的基本能力。在系统中，可以通过设置文件来将域名手动绑定到某上，文件位置在。对于普通用户，并不推荐自己手动绑定域名和，对于开发者来说，通过绑定域名和，可以轻松切换环境，可以从测试环境切换到开发环境，方便开发和测试。在系统中，黑客常常修改他的电脑的文件，将用户常常访问的域名绑定到他指定的上，从而实现了本地解析，导致这些域名被劫持。在或者系统中，文件在，修改该文件也可以实现同样的目的。

前两步都是在本机上完成的，所以没有在上面示例图上展示出来，从第三步开始，才正在地向远程DNS服务器发起解析域名的请求。

第三步：向本地域名解析服务系统发起域名解析的请求

如果在本机上无法完成域名的解析，那么系统只能请求本地域名解析服务系统进行解析，本地域名系统一般都是本地区的域名服务器，比如你连接的校园网，那么域名解析系统就在你的校园机房里，如果你连接的是电信、移动或者联通的网络，那么本地域名解析服务器就在本地区，由各自的运营商来提供服务。对于本地服务器地址，系统使用命令就可以查看，在和系统下，直接使用命令来查看服务地址。一般都缓存了大部分的域名解析的结果，当然缓存时间也受域名失效时间控制，大部分的解析工作到这里就差不多已经结束了，负责了大部分的解析工作。

第四步：向根域名解析服务器发起域名解析请求

本地域名解析器还没有完成解析的话，那么本地域名解析服务器将向根域名服务器发起解析请求。

第五步：根域名服务器返回gTLD域名解析服务器地址

本地域名解析向根域名服务器发起解析请求，根域名服务器返回的是所查域的通用顶级域（）地址。

第六步：向gTLD服务器发起解析请求

本地域名解析服务器向gTLD服务器发起请求。

第七步：gTLD服务器接收请求并返回Name Server服务器

服务器接收本地域名服务器发起的请求，并根据需要解析的域名，找到该域名对应的域名服务器，通常情况下，这个服务器就是你注册的域名服务器，那么你注册的域名的服务商的服务器将承担起域名解析的任务。

第八步：Name Server服务器返回IP地址给本地服务器

服务器查找域名对应的地址，将地址连同值返回给本地域名服务器。

第九步：本地域名服务器缓存解析结果

本地域名服务器缓存解析后的结果，缓存时间由时间来控制。

1.39 如何在Linux上配置一个IP地址，如果给定端口号如何解析出域名?

参考回答

1. 配置Linux系统的IP地址的方法，主要有以下三种：

- ifconfig

  ifconfig 命令主要是用来查看网卡的配置信息，因为用它来配置网卡的IP地址时，只会临时生效（Linux服务器重启后就会失效）

- setup

  setup 命令是 redhat 系列的linux系统（如CentOS）中专有的命令工具。可以使用 setup 命令，来对网络配置中的IP地址、子网掩码、默认网关、DNS服务器进行设置。而且，setup 网络配置工具设置的IP地址会永久生效。

- 修改网卡的配置文件

  直接修改网卡的配置文件，设置方法有两种：

  - 自动获取动态IP地址
  - 手工配置静态的IP地址

1. 使用dig命令解析域名

1.40 解释一下IP地址、子网掩码、网关。

参考回答

1. IP地址

IP地址有一个32位的连接地址,由4个8位字段组成,8位字段称为8位位组,每个8位位组之间用点号隔开,用于标识TCP/IP宿主机。每个IP地址都包含两部分:网络ID和主机ID,网络ID 标识在同一个物理网络上的所有宿主机,主机ID标识网络上的每一个宿主机,运行TCP/IP的每个计算机都需要唯一的IP地址。

Intenet委员会定义了五种地址类型以适应不同尺寸的网络。地址类型定义网络ID使用哪些位,它也定义了网络的可能数目和每个网络可能的宿主机数目.

1. 子网掩码(Subnet Mask)

使用子网可以把单个大网分成多个物理网络,并用路由器把它们连接起来。子网掩码用于屏蔽IP地址的一部分,使得TCP/IP能够区别网络ID和宿主机ID。当TCP/IP宿主机要通信时,子网掩码用于判断一个宿主机是在本地网络还是在远程网络。

缺省的子网掩码用于不分成子网的TCP/IP网络,对应于网络ID的所有位都置为1,每个8位位组的十进制数是255,对应于宿主机ID的所有位都置为0。

用于子网掩码的位数决定可能的子网数目和每个子网的宿主机数目,子网掩码的位数越多,则子网越多,但是宿主机也较少。

例:假设A类地址子网数是14,则所需位数至少为4,用于子网的位为: 11111111, 11110000, 00000000, 00000000, 子网掩码为255.240.0.0,每个子网的宿主机数目为2^20-2=1,048, 574个。

1. 网关（Gateway）

网关就是一个网络连接到另一个网络的“关口”。 按照不同的分类标准，网关也有很多种。TCP/IP协议里的网关是最常用的，在这里我们所讲的“网关”均指TCP/ IP协议下的网关。

网关实质上是一个网络通向其他网络的IP地址。比如有网络A和网络B，网络A的IP地址范围为“192.168.1.1~192. 168.1.254”，子网掩码为255.255.255.0；网络B的IP地址范围为“192.168.2.1~192. 168.2.254”，子网掩码为255.255.255.0。在没有路由器的情况下，两个网络之间是不能进行TCP/IP通信的，即使是两个网络连接在同一台交换机（或集线器）上，TCP/IP协议也会根据子网掩码（255.255.255.0）判定两个网络中的主机处在不同的网络里。而要实现这两个网络之间的通信，则必须通过网关。

如果网络A中的主机发现数据包的目的主机不在本地网络中，就把数据包转发给它自己的网关，再由网关转发给网络B的网关，网络B的网关再转发给网络B的某个主机。网络B向网络A转发数据包的过程也是如此。而要实现这两个网络之间的通信，则必须通过网关。如果网络A中的主机发现数据包的目的主机不在本地网络中，就把数据包转发给它自己的网关，再由网关转发给网络B的网关，网络B的网关再转发给网络B的某个主机。网络B向网络A转发数据包的过程也是如此 所以说，只有设置好网关的IP地址，TCP/IP协议才能实现不同网络之间的相互通信。那么这个IP地址是哪台机器的IP地址呢？网关的IP地址是具有路由功能的设备的IP地址，具有路由功能的设备有路由器、启用了路由协议的服务器（实质上相当于一台路由器）、代理服务器（也相当于一台路由器）。

1.41 说说IP如何寻址?

参考回答

IP寻址包括本地网络寻址和非本地网络寻址两部分

1. 本地网络寻址

假设有2个主机，他们是属于同一个网段。主机A和主机B，首先主机A通过本机的hosts表或者wins系统或dns系统先将主机B的计算机名转换为IP地址，然后用自己的IP地址与子网掩码计算出自己所出的网段，比较目的主机B的ip地址与自己的子网掩码，发现与自己是出于相同的网段，于是在自己的ARP缓存中查找是否有主机B的mac地址，如果能找到就直接做数据链路层封装并且通过网卡将封装好的以太网帧发送有物理线路上去。

如果arp缓存中没有主机B的的mac地址，主机A将启动arp协议通过在本地网络上的arp广播来查询主机B的mac地址，获得主机B的mac地址厚写入arp缓存表，进行数据链路层的封装，发送数据。

2. 非本地网络寻址

假设2个主机不是相同的网段，不同的数据链路层网络必须分配不同网段的IP地址并且由路由器将其连接起来。主机A通过本机的hosts表或wins系统或dns系统先主机B的计算机名转换为IP地址，然后用自己的IP地址与子网掩码计算出自己所处的网段，比较目的目的主机B的IP地址，发现与自己处于不同的网段。于是主机A将知道应该将次数据包发送给自己的缺省网关，即路由器的本地接口。

主机A在自己的ARP缓存中查找是否有缺省网关的MAC地址，如果能够找到就直接做数据链路层封装并通过网卡，将封装好的以太网数据帧发送到物理线路上去，如果arp缓存表中没有缺省网关的Mac地址，主机A将启动arp协议通过在本地网络上的arp广播来查询缺省网关的mac地址，获得缺省网关的mac地址后写入arp缓存表，进行数据链路层的封装，发送数据。

数据帧到达路由器的接受接口后首先解封装，变成IP数据包，对IP包进行处理，根据目的IP地址查找路由表，决定转发接口后做适应转发接口数据链路层协议帧的封装，并且发送到下一跳路由器，此过程继续直至到达目的的网络与目的主机。

1.42 操作系统的地址有几种，请具体说明。

参考回答

操作系统有物理地址、逻辑地址、线性地址（也叫虚拟地址）三种地址

1. 物理地址

在存储器里以字节为单位存储信息，为正确地存放或取得信息，每一个字节单元给以一个唯一的存储器地址，称为物理地址（Physical Address），又叫实际地址或绝对地址。

地址从0开始编号，顺序地每次加1，因此存储器的物理地址空间是呈线性增长的。它是用二进制数来表示的，是无符号整数，书写格式为十六进制数。它是出现在CPU外部地址总线上的寻址物理内存的地址信号，是地址变换的最终结果。用于内存芯片级的单元寻址，与处理器和CPU连接的地址总线相对应。

2. 逻辑地址

逻辑地址是指在计算机体系结构中是指应用程序角度看到的内存单元（memory cell）、存储单元（storage element）、网络主机（network host）的地址。 逻辑地址往往不同于物理地址（physical address），通过地址翻译器（address translator）或映射函数可以把逻辑地址转化为物理地址。

在有地址变换功能的计算机中,访问指令给出的地址 (操作数) 叫逻辑地址,也叫相对地址。要经过寻址方式的计算或变换才得到内存储器中的物理地址。把用户程序中使用的地址称为相对地址即逻辑地址。逻辑地址由两个16位的地址分量构成，一个为段基值，另一个为偏移量。两个分量均为无符号数编码。

3. 线性地址

线性地址（Linear Address）是逻辑地址到物理地址变换之间的中间层。在分段部件中逻辑地址是段中的偏移地址，然后加上基地址就是线性地址。

线性地址是一个32位无符号整数，可以用来表示高达4GB的地址，也就是，高达4294967296个内存单元。线性地址通常用十六进制数字表示，值的范围从0x00000000到0xffffffff）。程序代码会产生逻辑地址，通过逻辑地址变换就可以生成一个线性地址。如果启用了分页机制，那么线性地址可以再经过变换以产生一个物理地址。当采用4KB分页大小的时候，线性地址的高10位为页目录项在页目录表中的编号，中间10位为页表中的页号，其低12位则为偏移地址。如果是使用4MB分页机制，则高10位页号，低22位为偏移地址。如果没有启用分页机制，那么线性地址直接就是物理地址。

1.44 DNS用了哪些协议?

参考回答

1. DNS在进行区域传输的时候使用TCP协议，其它时候则使用UDP协议；

   DNS的规范规定了2种类型的DNS服务器，一个叫主DNS服务器，一个叫辅助DNS服务器。在一个区中主DNS服务器从自己本机的数据文件中读取该区的DNS数据信息，而辅助DNS服务器则从区的主DNS服务器中读取该区的DNS数据信息。当一个辅助DNS服务器启动时，它需要与主DNS服务器通信，并加载数据信息，这就叫做区传送（zone transfer）。

2. 为什么既使用TCP又使用UDP?

UDP报文的最大长度为512字节，而TCP则允许报文长度超过512字节。当DNS查询超过512字节时，协议的TC标志出现删除标志，这时则使用TCP发送。通常传统的UDP报文一般不会大于512字节。

1. 区域传送时使用TCP，主要有以下两点考虑：
   1. 辅域名服务器会定时（一般时3小时）向主域名服务器进行查询以便了解数据是否有变动。如有变动，则会执行一次区域传送，进行数据同步。区域传送将使用TCP而不是UDP，因为数据同步传送的数据量比一个请求和应答的数据量要多得多。
   2. TCP是一种可靠的连接，保证了数据的准确性。
2. 域名解析时使用UDP协议：

   客户端向DNS服务器查询域名，一般返回的内容都不超过512字节，用UDP传输即可。不用经过TCP三次握手，这样DNS服务器负载更低，响应更快。虽然从理论上说，客户端也可以指定向DNS服务器查询的时候使用TCP，但事实上，很多DNS服务器进行配置的时候，仅支持UDP查询包。

1.45 说一说你对Linux内核的了解。

参考回答

内核是操作系统的核心，具有很多最基本功能，它负责管理系统的进程、内存、设备驱动程序、文件和网络系统，决定着系统的性能和稳定性。

Linux 内核有 4 项工作：

1. **内存管理：** 追踪记录有多少内存存储了什么以及存储在哪里
2. **进程管理：** 确定哪些进程可以使用中央处理器（CPU）、何时使用以及持续多长时间
3. **设备驱动程序：** 充当硬件与进程之间的调解程序/解释程序
4. **系统调用和安全防护：** 从流程接受服务请求

在正确实施的情况下，内核对于用户是不可见的，它在自己的小世界（称为内核空间）中工作，并从中分配内存和跟踪所有内容的存储位置。用户所看到的内容（例如 Web 浏览器和文件则被称为用户空间。这些应用通过系统调用接口（SCI）与内核进行交互。

举例来说， 内核就像是一个为高管（硬件）服务的忙碌的个人助理。助理的工作就是将员工和公众（用户）的消息和请求（进程）转交给高管，记住存放的内容和位置（内存），并确定在任何特定的时间谁可以拜访高管、会面时间有多长。

为了更具象地理解内核，不妨将 Linux 计算机想象成有三层结构：

硬件：物理机（这是系统的底层结构或基础）是由内存（RAM）、处理器（或 CPU）以及输入/输出（I/O）设备（例如存储、网络和图形）组成的。其中，CPU 负责执行计算和内存的读写操作。

Linux 内核：操作系统的核心。它是驻留在内存中的软件，用于告诉 CPU 要执行哪些操作。

用户进程：这些是内核所管理的运行程序。用户进程共同构成了用户空间。用户进程有时也简称为进程。内核还允许这些进程和服务器彼此进行通信（称为进程间通信或 IPC）。

系统执行的代码通过以下两种模式之一在 CPU 上运行：内核模式或用户模式。在内核模式下运行的代码可以不受限制地访问硬件，而用户模式则会限制 SCI 对 CPU 和内存的访问。内存也存在类似的分隔情况（内核空间和用户空间）。这两个小细节构成了一些复杂操作的基础，例如安全防护、构建容器和虚拟机的权限分隔。

这也意味着：如果进程在用户模式下失败，则损失有限，无伤大雅，可以由内核进行修复。另一方面，由于内核进程要访问内存和处理器，因此内核进程的崩溃可能会引起整个系统的崩溃。由于用户进程之间会有适当的保护措施和权限要求，因此一个进程的崩溃通常不会引起太多问题。

1.46 说一说你对Linux内核态与用户态的了解。

参考回答

内核态其实从本质上说就是内核，它是一种**特殊的软件程序，控制计算机的硬件资源，例如协调CPU资源，分配内存资源，并且提供稳定的环境供应用程序运行。**

用户态就是提供应用程序运行的空间，为了使应用程序访问到内核管理的资源例如CPU，内存，I/O。内核必须提供一组通用的访问接口，这些接口就叫**系统调用。**

1. **系统调用**是操作系统的最小功能单位。根据不同的应用场景，不同的Linux发行版本提供的系统调用数量也不尽相同，大致在240-350之间。这些系统调用组成了用户态跟内核态交互的基本接口。
2. 从用户态到内核态切换可以通过三种方式：
    1. 系统调用：系统调用本身就是中断，但是是软件中断，跟硬中断不同。
    2. 异常：如果当前进程运行在用户态，如果这个时候发生了异常事件，就会触发切换。例如：缺页异常。
    3. 外设中断：当外设完成用户的请求时，会向CPU发送中断信号。

1.47 Linux负载是什么?

参考回答

负载(load)是linux机器的一个重要指标，直观了反应了机器当前的状态。

在UNIX系统中，系统负载是对当前CPU工作量的度量，被定义为特定时间间隔内运行队列中的平均线程数。load average 表示机器一段时间内的平均load。这个值越低越好。负载过高会导致机器无法处理其他请求及操作，甚至导致死机。

top 或 uptime 等命令会输出系统的平均负载 (Load Average)，一般会有三个值，分别代表 1 分钟，5 分钟和 15 分钟的平均负载。

负载记录的是 CPU 的负荷，能对 CPU 造成负荷的是进程（包括线程）的执行。负载的数值代表的是 CPU 还没处理完的进程的数目。

系统的负载采用的是指数移动平均，计算方法如下：

```
S(0) = 0 S(t) = a * X(t) + (1-a)*S(t-1)
```

其中，X(t) 为最近一次采样的值，a 为最近采样值占的比重，S(t) 则是系统最近一次采样的负载。

指数移动平均的计算方式会累计历史所有的采样值，但离现在越久，占的比重越小。更具体的，Linux 系统上对 1 分钟的平均负载取 a 的取值2为 1 - e^(-5/60)，5 分钟为 1 - e^(-5s/5min)，以此类推。

以一分钟为例，上面的取值能达到的效果是，最近一分钟的采样占所有历史值的比重约为 63%（准确值为 1 - 1/e），5 分钟和 15 分钟也一样。

单核满载是 1，有 n 核满载是 n。一般说线上运行的系统大于 0.7 的时候就要注意了。

1.48 Linux如何设置开机启动?

参考回答

1. 编辑rc.loacl脚本

   linux开机之后会执行/etc/rc.local文件中的脚本。

   所以可以直接在/etc/rc.local中添加启动脚本。

   ```
   $ vim /etc/rc.local
   ```

2. 添加一个开机启动服务。

   将启动脚本复制到 /etc/init.d目录下，并设置脚本权限, 假设脚本为test

   ```
   $ mv test /etc/init.d/test  $ sudo chmod 755 /etc/init.d/test
   ```

   将该脚本放倒启动列表中去

   ```
   $ cd .etc/init.d
   $ sudo update-rc.d test defaults 95
   ```

   注：其中数字95是脚本启动的顺序号，按照自己的需要相应修改即可。在有多个启动脚本，而它们之间又有先后启动的依赖关系时就知道这个数字的具体作用了。

   将该脚本从启动列表中剔除

   ```
   $ cd /etc/init.d
   $ sudo update-rc.d -f test remove
   ```

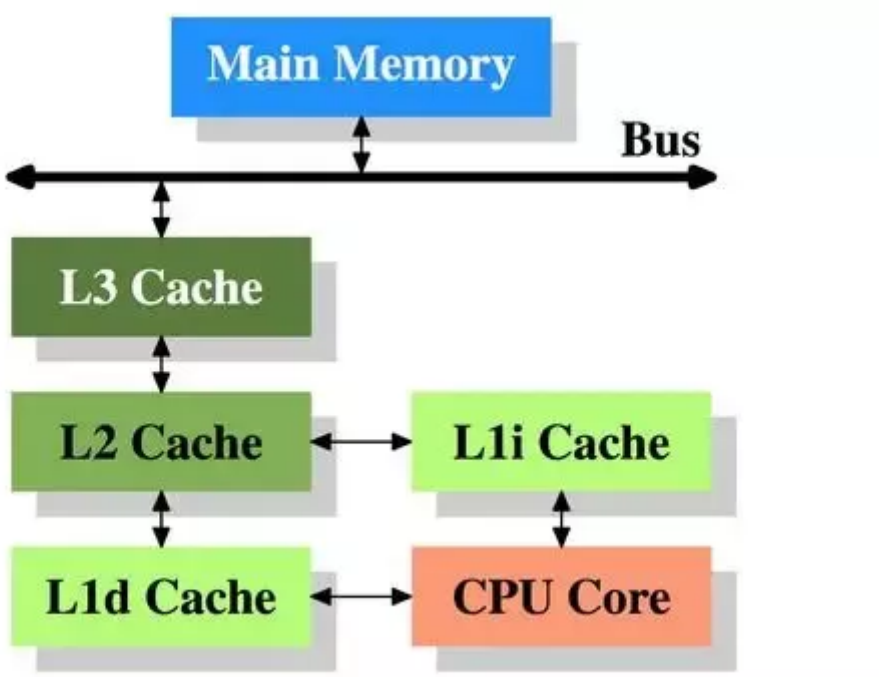
1.49 谈谈Linux的内存管理。

参考回答

常见的计算机存储层次如下：

- 寄存器：CPU提供的，读写ns级别，容量字节级别。
- CPU缓存：CPU和CPU间的缓存，读写10ns级别，容量较大一些，百到千节。
- 主存：动态内存，读写100ns级别，容量GB级别。
- 外部存储介质：磁盘、SSD，读写ms级别，容量可扩展到TB级别。

CPU内的缓存示意图如下：



其中 L1d 和 L1i 都是CPU内部的cache，

- L1d 是数据cache。
- L1i 是指令缓存。
- L2是CPU内部的，不区分指令和数据的。
- 由于现代PC有多个CPU，L3缓存多个核心共用一个。

对于编程人员来说，绝大部分观察主存和外部存储介质就可以了。如果要做极致的性能优化，可以关注L1、L2、L3的cache，比如nginx的绑核操作、pthread调度会影响CPU cache等。

1. 虚拟内存

物理内存是有限的（即使支持了热插拔）、非连续的，不同的CPU架构对物理内存的组织都不同。这使得直接使用物理内存非常复杂，为了降低使用内存的复杂度，引入了虚拟内存机制。

虚拟内存抽象了应用程序物理内存的细节，只允许物理内存保存所需的信息（按需分页），并提供了一种保护和控制进程间数据共享数据的机制。有了虚拟内存机制之后，每次访问可以使用更易理解的虚拟地址，让CPU转换成实际的物理地址访问内存，降低了直接使用、管理物理内存的门槛。

物理内存按大小被分成页框、页，每块物理内存可以被映射为一个或多个虚拟内存页。这块映射关系，由操作系统的页表来保存，页表是有层级的。层级最低的页表，保存实际页面的物理地址，较高层级的页表包含指向低层级页表的物理地址，指向顶级的页表的地址，驻留在寄存器中。当执行地址转换时，先从寄存器获取顶级页表地址，然后依次索引，找到具体页面的物理地址。

2. 大页机制

虚拟地址转换的过程中，需要好几个内存访问，由于内存访问相对CPU较慢，为了提高性能，CPU维护了一个TLB地址转换的cache，TLB是比较重要且珍稀的缓存，对于大内存工作集的应用程序，会因TLB命中率低大大影响到性能。

为了减少TLB的压力，增加TLB缓存的命中率，有些系统会把页的大小设为MB或者GB，这样页的数目少了，需要转换的页表项也小了，足以把虚拟地址和物理地址的映射关系，全部保存于TLB中。

3. 区域概念

通常硬件会对访问不同的物理内存的范围做出限制，在某些情况下设备无法对所有的内存区域做DMA。在其他情况下，物理内存的大小也会超过了虚拟内存的最大可寻址大小，需要执行特殊操作，才能访问这些区域。这些情况下，Linux对内存页的可能使用情况将其分组到各自的区域中（方便管理和限制）。比如ZONE_DMA用于指明哪些可以用于DMA的区域，ZONE_HIGHMEM包含未永久映射到内核地址空间的内存，ZONE_NORMAL标识正常的内存区域。

4. 节点

多核CPU的系统中，通常是NUMA系统（非统一内存访问系统）。在这种系统中，内存被安排成具有不同访问延迟的存储组，这取决于与处理器的距离。每一个库，被称为一个节点，每个节点Linux构建了一个独立的内存管理子系统。一个节点有自己的区域集、可用页和已用页表和各种统计计数器。

5. page cache

从外部存储介质中加载数据到内存中，这个过程是比较耗时的，因为外部存储介质读写性能毫秒级。为了减少外部存储设备的读写，Linux内核提供了Page cache。最常见的操作，每次读取文件时，数据都会被放入页面缓存中，以避免后续读取时所进行昂贵的磁盘访问。同样，当写入文件时，数据被重新放置在缓存中，被标记为脏页，定期的更新到存储设备上，以提高读写性能。

6. 匿名内存

匿名内存或者匿名映射表示不受文件系统支持的内存，比如程序的堆栈隐式创立的，或者显示通过mmap创立的。

7. 内存回收

贯穿系统的生命周期，一个物理页可存储不同类型的数据，可以是内核的数据结构，或是DMA访问的buffer，或是从文件系统读取的数据，或是用户程序分配的内存等。

根据页面的使用情况，Linux内存管理对其进行了不同的处理，可以随时释放的页面，称之为可回收页面，这类页面为：页面缓存或者是匿名内存（被再次交换到硬盘上）

大多数情况下，保存内部内核数据并用DMA缓冲区的页面是不能重新被回收的，但是某些情况下，可以回收使用内核数据结构的页面。例如：文件系统元数据的内存缓存，当系统处于内存压力情况下，可以从主存中丢弃它们。

释放可回收的物理内存页的过程，被称之为回收，可以同步或者异步的回收操作。当系统负载增加到一定程序时，kswapd守护进程会异步的扫描物理页，可回收的物理页被释放，并逐出备份到存储设备。

8. compaction

系统运行一段时间，内存就会变得支离破碎。虽然使用虚拟村内可以将分散的物理页显示为连续的物理页，但有时需要分配较大的物理连续内存区域。比如设备驱动程序需要一个用于DMA的大缓冲区时，或者大页内存机制分页时。内存compact可以解决了内存碎片的问题，这个机制将被占用的页面，从内存区域合适的移动，以换取大块的空闲物理页的过程，由kcompactd守护进程完成。

9. OOM killer

机器上的内存可能会被耗尽，并且内核将无法回收足够的内存用于运行新的程序，为了保存系统的其余部分，内核会调用OOM killer杀掉一些进程，以释放内存。

1. 段页机制

段页机制是操作系统管理内存的一种方式，简单的来说，就是如何管理、组织系统中的内存。要理解这种机制，需要了解一下内存寻址的发展历程。

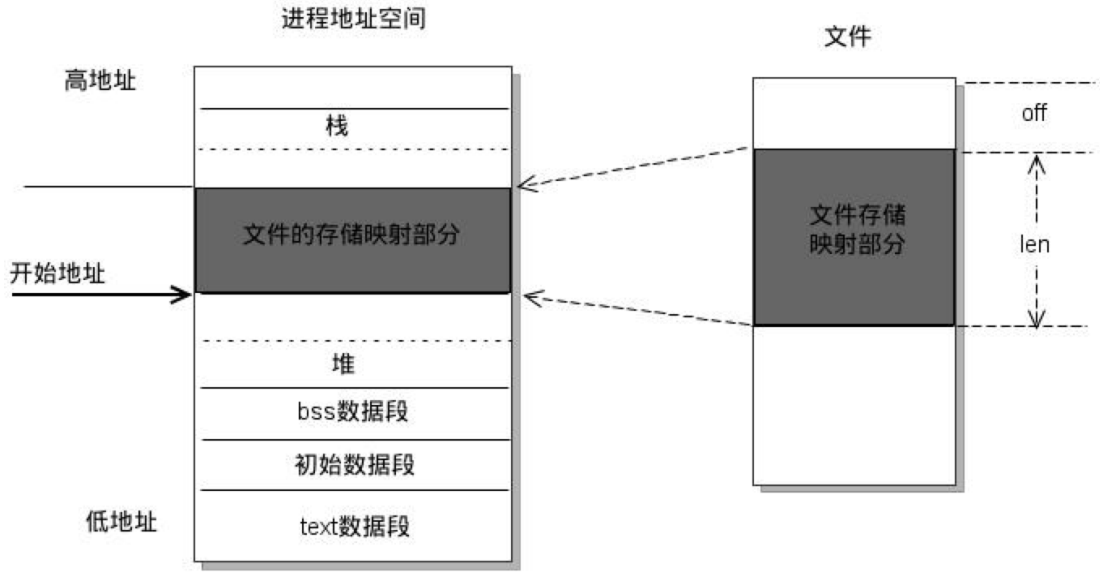
- 直接寻址：早期的内存很小，通过硬编码的形式，直接定位到内存地址。这种方式有着明显的缺点：可控性弱、难以重定位、难以维护

- 分段机制：8086处理器，寻址空间达到1MB，即地址线扩展了20位，由于制作20位的寄存器较为困难，为了能在16位的寄存器的基础上，寻址20位的地址空间，引入了段的概念，即内存地址=段基址左移4位+偏移
- 分页机制：随着寻址空间的进一步扩大、虚拟内存技术的引入，操作系统引入了分页机制。引入分页机制后，逻辑地址经过段机制转换得到的地址仅是中间地址，还需要通过页机制转换，才能得到实际的物理地址。逻辑地址 -->(分段机制) 线性地址 -->(分页机制) 物理地址。

1.50 谈谈内存映射文件。

参考回答

1. **内存映射（mmap）** 是一种内存映射文件的方法，即将一个文件或者其他对象映射到进程的地址空间，实现文件磁盘地址和应用程序进程虚拟地址空间中一段虚拟地址的一一映射关系。实现这样的映射关系后，进程就可以采用指针的方式读写操作这一段内存，而系统会自动回写脏页面到对应的文件磁盘上。应用程序处理映射部分如同访问主存。



1. mmap内存映射原理

（1）线程启动映射过程，并在虚拟地址空间中为映射创建虚拟映射区域。

先在用户空间调用库函数mmap，并在进程当前进程的虚拟地址空间中，寻找一段空闲的满足要求的连续虚拟地址作为内存虚拟映射区域，对此区域初始化并插入进程的虚拟地址区域链表或树中。

（2）系统在内核空间调用内核函数mmap，实现文件物理地址和进程虚拟地址之间的一一映射关系。

（3）进程发起堆这片映射空间的访问

进程读写操作访问虚拟地址，查询页表，发现这一段地址并不在内存的物理页面上，因为虽然建立了映射关系，但是还没有将文件从磁盘移到内存中。由此发生缺页中断，内核请求从磁盘调入页面。调页过程先在交换缓存空间（swap cache）中查找，若没有则通过nopage函数把缺失页从磁盘调入内存。之后进程会对其做读写操作，若写操作改变了页面内容，一段时间后系统会自动回写脏页面到磁盘中。(修改过的脏页面不会立即更新到文件中，可以调用msync来强制同步，写入文件)

1. mmap和分页文件操作的区别

区别在于分页文件操作在进程访存时是需要先查询页面缓存 **(page cache)** 的，若发生缺页中断，需要通过inode定位文件磁盘地址，先把缺失文件复制到page cache，再从page cache复制到内存中，才能进行访问。这样访存需要经过两次文件复制，写操作也是一样。总结来说，常规文件操作为了提高读写效率和保护磁盘，使用了页缓存机制。这样造成读文件时需要先将文件页从磁盘拷贝到页缓存中，由于

页缓存处在内核空间，不能被用户进程直接寻址，所以还需要将页缓存中数据页再次拷贝到内存对应的用户空间中。**但mmap的优势在于，把磁盘文件与进程虚拟地址做了映射，这样可以跳过page cache，只使用一次数据拷贝。**

1.51 谈谈虚拟内存模型。

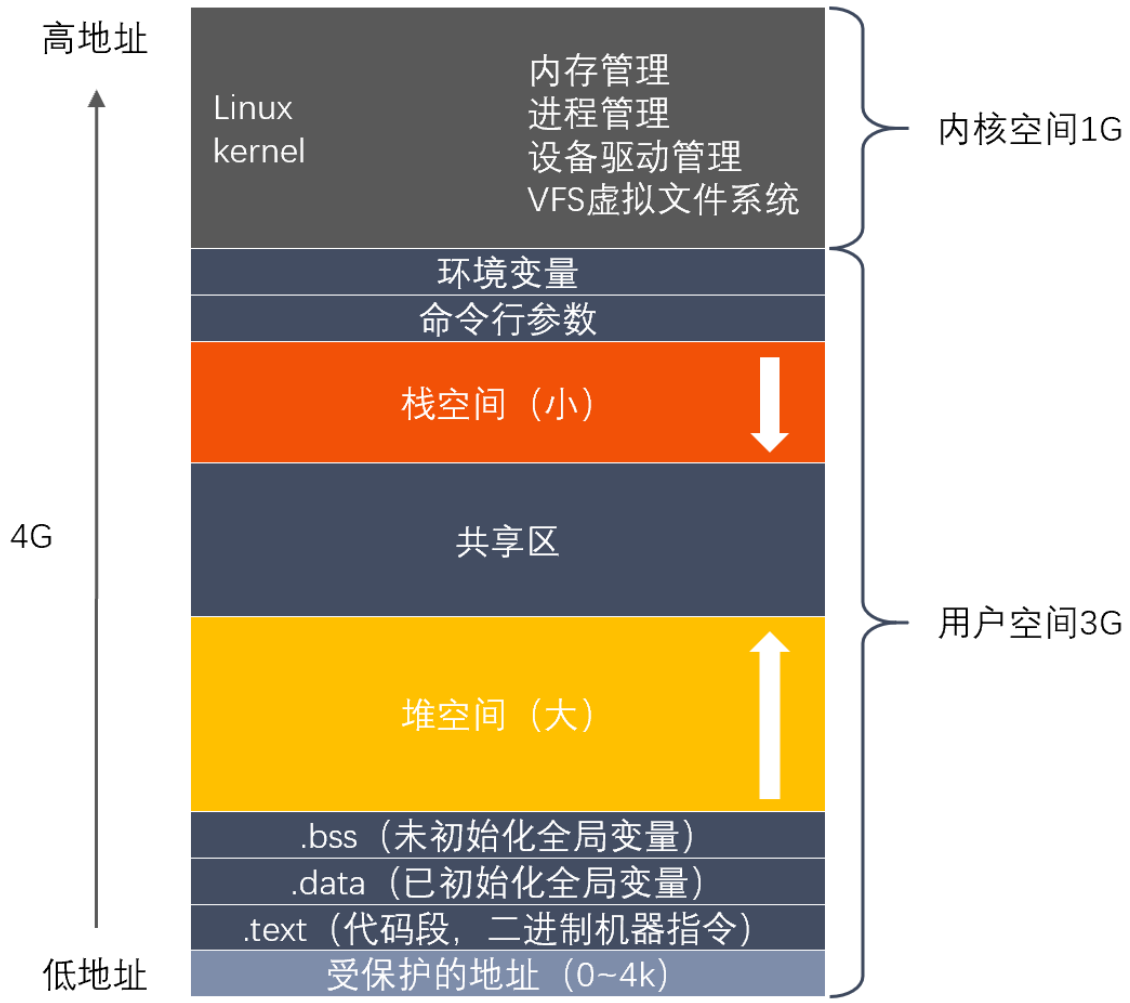
参考回答

虚拟内存分成五大区，分别为**栈区、堆区、全局区（静态区）、文字常量区（常量存储区）、程序代码区。**五大区特性如下：

1. 栈区（stack）： 由编译器自动分配释放 ，存放函数的参数值，局部变量的值等。其操作方式类似于数据结构中的栈。
2. 堆区（heap）： 一般由程序员分配释放， 若程序员不释放，程序结束时可能由OS回收 。注意它与数据结构中的堆是两回事，分配方式倒是类似于链表。
3. 全局区（静态区）（static）：全局变量和静态变量的存储是放在一块的，初始化的全局变量和静态变量在一块区域， 未初始化的全局变量和未初始化的静态变量在相邻的另一块区域。程序结束后由系统释放。
4. 文字常量区（**常量存储区**）：**常量字符串**就是放在这里的。 程序结束后由系统释放。这是一块比较特殊的存储区，他们里面存放的是常量，不允许修改。
5. 程序代码区：存放函数体的二进制代码。

答案解析

以32位的操作系统为例，32位的操作系统每个进程对应的虚拟内存为4G（2的32次方），其中内核区1G，用户区3G。结构图如下：



1.53 内存和缓存有什么区别?

参考回答

内存和缓存是计算机不同的组成部件。

1. 内存特性

   内存也被称作内存储器，其作用是用于暂时存放CPU的运算数据，以及与硬盘等外部存储交换的数据。只要计算机在运行中，CPU就会把需要进行运算的数据调到内存中进行运算，当运算完成后CPU再将结果传送出来，内存的运行也决定了计算机的稳定运行。

2. 缓存特性

   CPU芯片面积和成本的因素影响，决定了缓存都很小。现在一般的缓存不过几M，CPU缓存的运行频率极高，一般是和处理器同频运作，工作效率远远大于系统内存和硬盘。实际工作时，CPU往往需要重复读取读取同样的数据块，而缓存容量的增大，可以大幅度提升CPU内部读取数据的命中率，而不用再到内存或者硬盘上寻找，以此提高系统性能。

1.54 请你说说缓存溢出。

参考回答

1. 缓存溢出及其危害

缓存溢出是指输入到一个缓冲区或者数据保存区域的数据量超过了其容量，从而导致覆盖了其它区域数据的状况。攻击者造成并利用这种状况使系统崩溃或者通过插入特制的代码来控制系统。被覆盖的区域可能存有其它程序的变量、参数、类似于返回地址或指向前一个栈帧的指针等程序控制流数据。缓冲区可以位于堆、栈或进程的数据段。这种错误可能产生如下后果：

（1）破坏程序的数据；

（2）改变程序的控制流，因此可能访问特权代码。

最终很有可能造成程序终止。当攻击者成功地攻击了一个系统之后，作为攻击的一部分，程序的控制流可能会跳转到攻击者选择的代码处，造成的结果是被攻击的进程可以执行任意的特权代码（比如通过判断输入是否和密码匹配来访问特权代码，如果存在缓冲区漏洞，非法输入导致存放“密码”的内存区被覆盖，从而使得“密码”被改写，因此判断为匹配进而获得了特权代码的访问权）

缓冲区溢出攻击是最普遍和最具危害性的计算机安全攻击类型之一。

1. 如何预防缓存溢出

   广义上分为**两类**：

   （1）编译时防御系统，目的是强化系统以抵御潜伏于新程序中的恶意攻击

   （2）运行时预防系统，目的是检测并终止现有程序中的恶意攻击

1.55 深拷贝和浅拷贝的区别是什么，它们各自的使用场景是什么?

参考回答

浅拷贝只是对指针的拷贝，拷贝后两个指针指向同一个内存空间；**深拷贝**不断对指针进行拷贝，而且对指针指向的内容进行拷贝，经深拷贝后的指针是指向两个不同的地址空间。

1. 浅拷贝

   对一个已知对象进行拷贝时，编译系统会自动调用一次构造函数（拷贝构造函数），如果用户未定义拷贝构造函数，则会调用默认拷贝构造函数，调用一次构造函数，调用两次析构函数，两个对象的指针成员所指内存相同，但是程序结束时该内存被释放了两次，会造成内存泄漏问题。

2. 深拷贝

在对含有指针成员的对象进行拷贝时，必须要自己定义拷贝构造函数，使拷贝后的对象指针成员有自己的内存空间，即进行深拷贝，这样就避免了内存泄漏的发生，调用一次构造函数，一次自定义拷贝构造函数，两次析构函数。两个对象的指针成员所指内容不同。

1.56 说说IO模型。

参考回答

1. 什么是IO

我们都知道unix世界里，一切皆文件。而文件是什么呢？文件就是一串二进制流而已。无论是socket，还是FIFO、管道、终端，对我们来说，一切都是文件，一切都是流。在信息交换的过程中，我们都是对这些流进行数据的收发操作简称为**I/O操作**(input and output)。往流中读出数据，系统调用read；写入数据，系统调用write。

计算机里有这么多的流，我怎么知道要操作哪个流呢?

做到这个的就是**文件描述符**，即通常所说的fd，一个fd就是一个整数，所以对这个整数的操作就是对这个文件（流）的操作。我们创建一个socket，通过系统调用会返回一个文件描述符，那么剩下对socket的操作就会转化为对这个描述符的操作。不能不说这又是一种分层和抽象的思想。

2. IO交互

对于一个输入操作来说，进程IO系统调用后，内核会先看缓冲区有没有相应的缓存数据，没有的话再到设备中读取，因为设备IO一般速度较慢，需要等待，内核缓冲区有数据则直接复制到进程空间。所以，对于一个网络输入操作通常包括**两个不同阶段**:

（1）等待网络数据到达网卡->读取到内核缓冲区

（2）从内核缓冲区复制数据->用户空间

IO有内存IO、网络IO和磁盘IO三种，通常我们所说的IO指的是**网络IO**和**磁盘IO**两者。

3. 五大I/O模型

Linux有五大I/O模型，分别为**阻塞IO、同步非阻塞IO、IO多路复用、信号驱动IO、异步IO**。五种IO模型特性分别如下:

（1）阻塞IO（blocking IO）

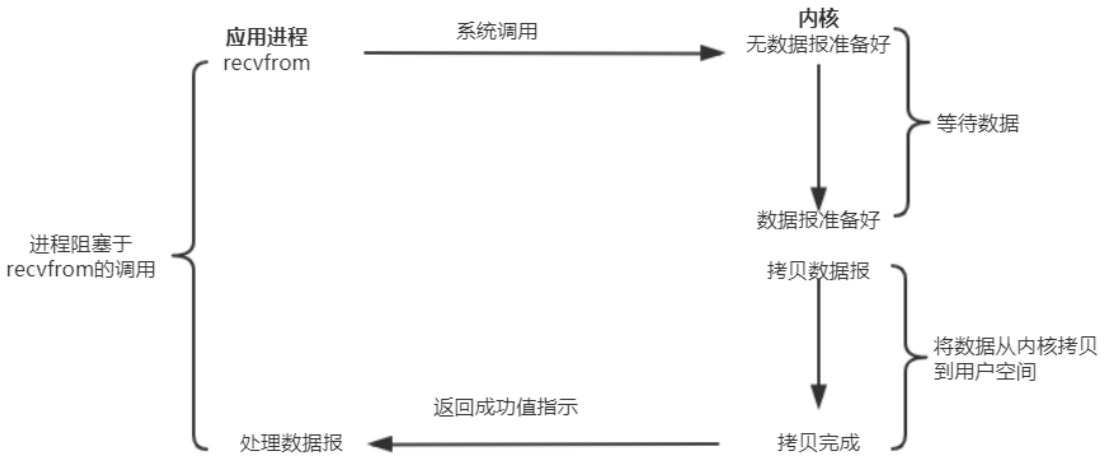
最传统的一种IO模型，即在读写数据过程中会发生阻塞现象。

当用户线程发出IO请求之后，内核会去查看数据是否就绪，如果没有就绪就会等待数据就绪，而用户线程就会处于阻塞状态，用户线程交出CPU。当数据就绪之后，内核会将数据拷贝到用户线程，并返回结果给用户线程，用户线程才解除block状态。

典型的阻塞IO模型的例子为:

data = socket.read();

如果数据没有就绪，就会一直阻塞在read方法。



阻塞I/O模型

（2）同步非阻塞IO（nonblocking IO)

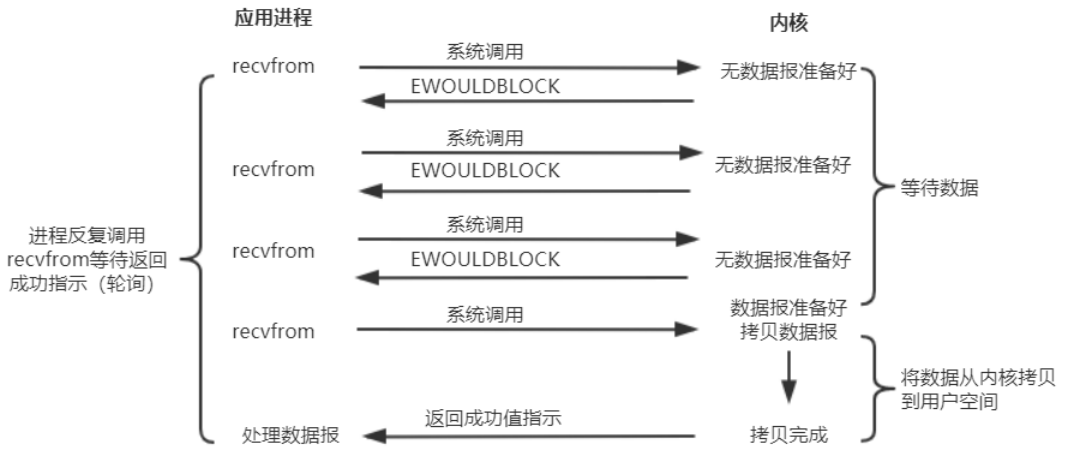
当用户线程发起一个read操作后，并不需要等待，而是马上就得到了一个结果。如果结果是一个error时，它就知道数据还没有准备好，于是它可以再次发送read操作。一旦内核中的数据准备好了，并且又再次收到了用户线程的请求，那么它马上就将数据拷贝到了用户线程，然后返回。

所以事实上，在非阻塞IO模型中，用户线程需要不断地询问内核数据是否就绪，也就说非阻塞IO不会交出CPU，而会一直占用CPU。

典型的非阻塞IO模型一般如下：

```
while(true){    data = socket.read();   if(data!= error){  处理数据  break;  }
}
```

但是对于非阻塞IO就有一个非常严重的问题，在while循环中需要不断地去询问内核数据是否就绪，这样会导致CPU占用率非常高，因此一般情况下很少使用while循环这种方式来读取数据。



同步非阻塞I/O模型

（3）IO多路复用（IO multiplexing)

多路复用IO模型是目前使用得比较多的模型。Java NIO实际上就是多路复用IO。

在多路复用IO模型中，会有一个线程不断去轮询多个socket的状态，只有当socket真正有读写事件时，才真正调用实际的IO读写操作。因为在多路复用IO模型中，只需要使用一个线程就可以管理多个socket，系统不需要建立新的进程或者线程，也不必维护这些线 程和 进程，并且只有在真正有socket读写事件进行时，才会使用IO资源，所以它大大减少了资源占用。

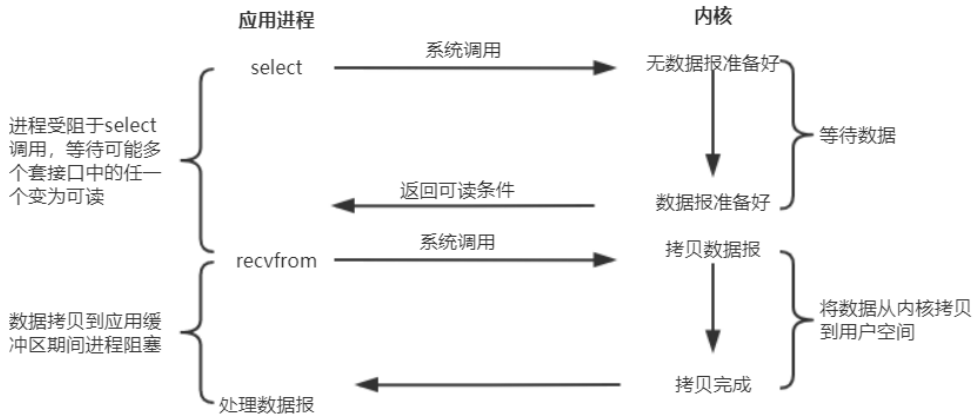
在Java NIO中，是通过selector.select()去查询每个通道是否有到达事件，如果没有事件，则一直阻塞在那里，因此这种方式会导致用户线程的阻塞。

也许有朋友会说，我可以采用多线程+ 阻塞IO 达到类似的效果，但是由于在多线程 + 阻塞IO 中，每个socket对应一个线程，这样会造成很大的资源占用，并且尤其是对于长连接来说，线程的资源一直不会释放，如果后面陆续有很多连接的话，就会造成性能上的瓶颈。

而多路复用IO模式，通过一个线程就可以管理多个socket，只有当socket真正有读写事件发生才会占用资源来进行实际的读写操作。因此，多路复用IO比较适合连接数比较多的情况。

另外多路复用IO为何比非阻塞IO模型的效率高是因为在非阻塞IO中，不断地询问socket状态时通过用户线程去进行的，而在多路复用IO中，轮询每个socket状态是内核在进行的，这个效率要比用户线程要高的多。

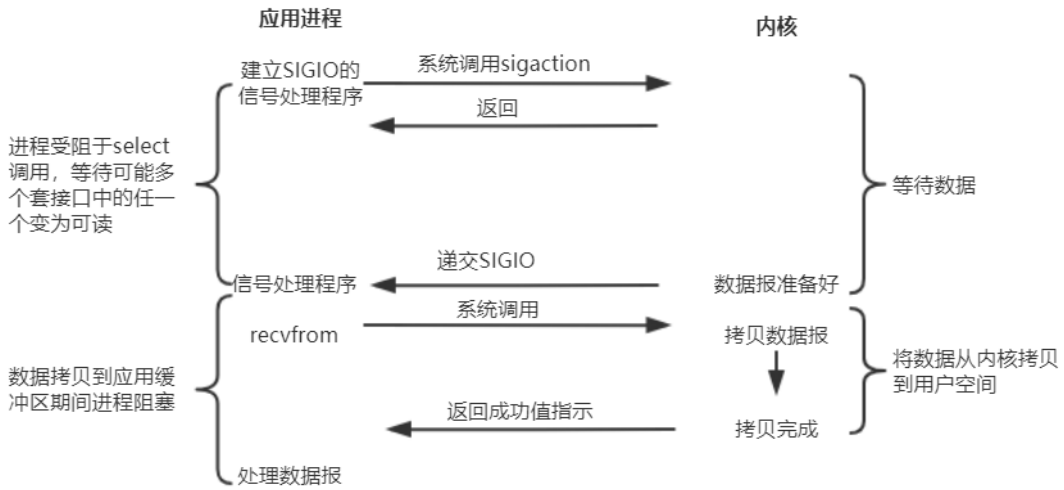
注意:多路复用IO模型是通过轮询的方式来检测是否有事件到达，并且对到达的事件逐一进行响应。因此对于多路复用IO模型来说，一旦事件响应体很大，那么就会导致后续的事件迟迟得不到处理，并且会影响新的事件轮询。



I/O多路复用模型

（4）信号驱动 IO（signal driven IO）

在信号驱动IO模型中，当用户线程发起一个IO请求操作，会给对应的socket注册一个信号函数，然后用户线程会继续执行，当内核数据就绪时会发送一个信号给用户线程，用户线程接收到信号之后，便在信号函数中调用IO读写操作来进行实际的IO请求操作。这个一般用于UDP中，对TCP套接口几乎是没用的，原因是该信号产生得过于频繁，并且该信号的出现并没有说明发生了什么事情。



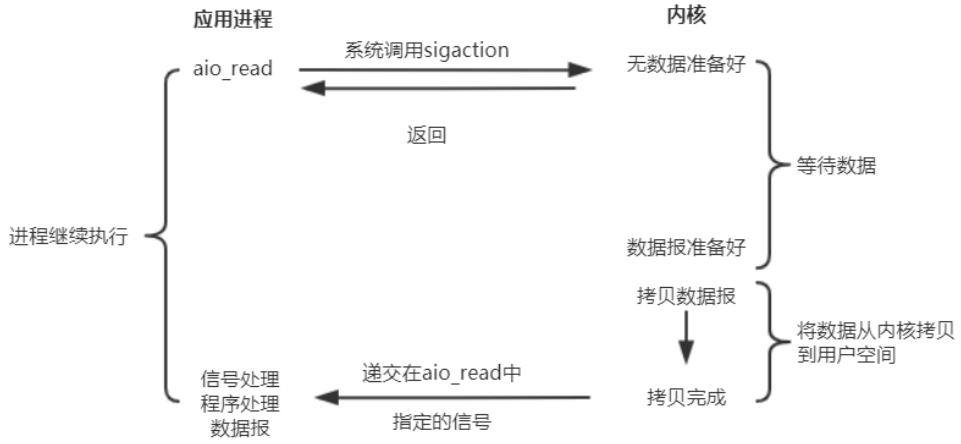
信号驱动I/O模型

（5）异步 IO（asynchronous IO）

异步IO模型才是最理想的IO模型，在异步IO模型中，当用户线程发起read操作之后，立刻就可以开始去做其它的事。而另一方面，从内核的角度，当它收到一个asynchronous read之后，它会立刻返回，说明read请求已经成功发起了，因此不会对用户线程产生任何阻塞。然后，内核会等待数据准备完成，再将数据拷贝到用户线程，当这一切都完成之后，内核会给用户线程发送一个信号，告诉它read操作完成了。也就说用户线程完全不需要关心实际的整个IO操作是如何进行的，只需要先发起一个请求，当接收内核返回的成功信号时表示IO操作已经完成，可以直接去使用数据了。

也就说在异步IO模型中，IO操作的两个阶段都不会阻塞用户线程，这两个阶段都是由内核自动完成，然后发送一个信号告知用户线程操作已完成。用户线程中不需要再次调用IO函数进行具体的读写。这点是和信号驱动模型有所不同的，在信号驱动模型中，当用户线程接收到信号表示数据已经就绪，然后需要用户线程调用IO函数进行实际的读写操作；而在异步IO模型中，收到信号表示IO操作已经完成，不需要再在用户线程中调用IO函数进行实际的读写操作。

注意:异步IO是需要操作系统的底层支持，在Java 7中，提供了Asynchronous IO（简称AIO）。



异步I/O模型

前四种IO模型实际上都属于同步IO，只有最后一种是真正的异步IO，因为无论是多路复用IO还是信号驱动模型，IO操作的第2个阶段都会引起用户线程阻塞，也就是内核进行数据拷贝的过程都会让用户线程阻塞。

1.57 Linux中的软链接和硬链接有什么区别?

参考回答

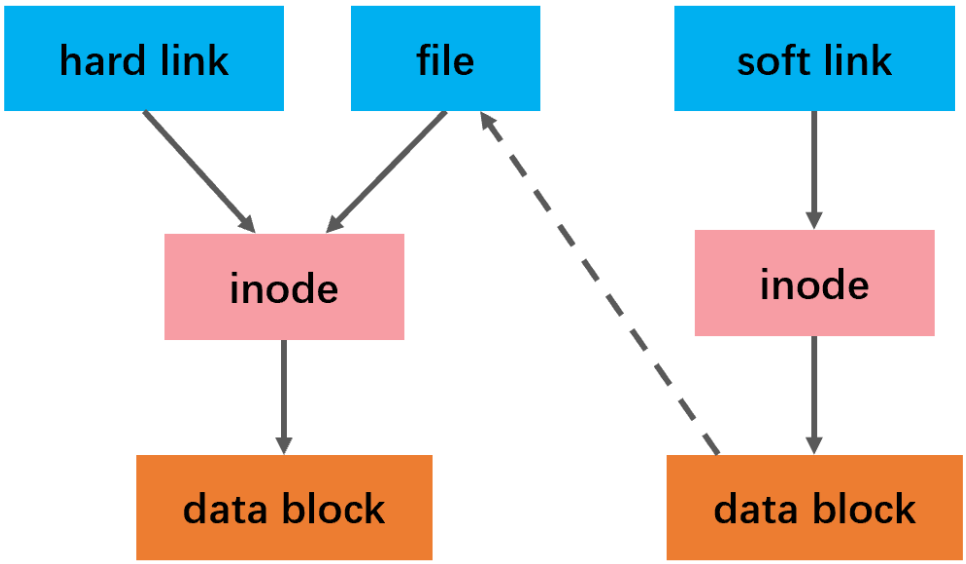
1. inode概念

   inode是文件系统中存储文件元信息的区域，中文叫节点索引，每个节点索引包含了文件的创建者，大小，日期等等。可以通过ls -i file 命令查看inode的值。

2. 根据 Linux 系统存储文件的特点，链接的方式分为**软链接和硬链接**2 种

   软链接相当于建立了一个新的快捷方式文件，该文件有自己的名称和inode以及物理存储的文件数据，文件数据里记录着如何跳转的设置数据，访问该快捷文件会被重新定向到原始文件，删除原始文件，软链文件失效；**硬链接**相当于为当前文件名对应的文件再建立了一个文件别名，别名对应的inode以及物理数据都是一样的，一旦建立，我们甚至根本无法区分谁是原始文件的原始名称，删除文件的其中一个名称，文件不会丢失，除非把所有的名称都删除。

   如下图:



   hard link(硬链) 和file 都指向同一个 inode，inode对应了一个实际物理存储的文件。soft link（软链） 对应一个新的inode， 新的inode对应一个新的物理存储文件，物理存储文件又指向了目标文件 file。

3. **软连接和硬链接的区别**

   (1) **软链接**可以为文件和目录（哪怕是不存在的）创建链接；**硬链接**只能为文件创建链接。 (2) **软链接**可以跨文件系统；**硬链接**必须是同一个文件系统 (3) **硬链接**因为只是文件的一个别名，所以不重复占用内存；**软链接**因为只是一个访问文件的快捷方式文件，文件内只包含快捷指向信息，所以占用很小的内存。 (4) **软链接**的文件权限和源文件可以不一样；**硬链接**文件权限肯定是一样的，因为他们本来就是一个文件的不同名称而已。

4. 二者使用场景

   一般比较重要的文件我们担心文件被误删除且传统复制备份方式占用double数量的空间会造成浪费，可以使用硬链做备份来解决；软链接一般被用来设置可执行文件的快捷方式的路径。

答案解析

1. 创建硬链接

```
[root@localhost ~]# touch cangls [root@localhost ~]# ln /root/cangls /tmp #建立硬链接文件，目标文件没有写文件名，会和原名一致 #也就是/tmp/cangls 是硬链接文件
```

2. 创建软链接

```
[root@localhost ~]# touch bols [root@localhost ~]# In -s /root/bols /tmp #建立软链接文件
```

注意：软链接文件的源文件必须写成绝对路径，而不能写成相对路径（硬链接没有这样的要求）；否则软链接文件会报错。

1.58 说说缺页中断机制。

参考回答

1. 缺页中断

   在请求分页系统中，可以通过查询页表中的状态位来确定所要访问的页面是否存在于内存中。当所要访问的页面不在内存时，会产生一次**缺页中断**，此时操作系统会根据页表中的外存地址在外存中找到所缺的一页，将其调入内存。 缺页中断的处理流程如下：

   A. 在内存中有空闲物理页面时，分配一物理页帧f，转第E步；

   B. 依据页面置换算法选择被替换的物理页帧f，对应逻辑页q；

   C. 如果q被修改过，则把它写回外存；

   D. 修改q的页表项中驻留位置为0；

   E. 将需要访问的页p装入到物理页f；

   F. 修改p的页表项驻留位为1，物理页帧号为f；

   G. 重新执行产生缺页的指令。

2. **缺页中断与一般的中断存在区别**

   （1）范围不同

一般中断只需要保护现场然后就直接跳到需及时处理的地方；

缺页中断除了保护现场之外，还要判断内存中是否有足够的空间存储所需的页或段，然后再把所需页调进来再使用。

（2）结果不同

一般中断在处理完之后返回时，执行下一条指令；

缺页中断返回时，执行产生中断的那一条指令。

（3）次数不同

一般中断只产生一次，发生中断指令后转入相应处理程序进行处理，恢复被中断程序现场;

在指令执行期间产生和处理缺页中断信号，一条指令在执行期间，可能产生多次缺页中断。

答案解析

产生缺页中断的几种情况：

1. 当内存管理单元（MMU）中确实没有创建虚拟物理页映射关系，并且在该虚拟地址之后再没有当前进程的线性区（vma）的时候，这将杀掉该进程；

2. 当MMU中确实没有创建虚拟页物理页映射关系，并且在该虚拟地址之后存在当前进程的线性区vma的时候，这很可能是缺页中断，并且可能是栈溢出导致的缺页中断；
3. 当使用malloc/mmap等希望访问物理空间的库函数/系统调用后，由于linux并未真正给新创建的vma映射物理页，此时若先进行写操作，将和2产生缺页中断的情况一样；若先进行读操作虽然也会产生缺页异常，将被映射给默认的零页，等再进行写操作时，仍会产生缺页中断，这次必须分配1物理页了，进入写时复制的流程；
4. 当使用fork等系统调用创建子进程时，子进程不论有无自己的vma，它的vma都有对于物理页的映射，但它们共同映射的这些物理页属性为只读，即linux并未给子进程真正分配物理页，当父子进程任何一方要写相应物理页时，导致缺页中断的写时复制。

1.59 软中断和硬中断有什么区别?

参考回答

1. 硬中断

   由与系统相连的外设(比如网卡、硬盘)自动产生的。主要是用来通知操作系统外设状态的变化。比如当网卡收到数据包的时候，就会发出一个中断。我们通常所说的中断指的是硬中断(hardirq)。

2. 软终端

   为了满足实时系统的要求，中断处理应该是越快越好。Linux为了实现这个特点，当中断发生的时候，硬中断处理那些短时间就可以完成的工作，而将那些处理事件比较长的工作，放到中断之后来完成，也就是软中断(softirq)来完成。

3. 中断嵌套

   Linux下**硬中断是可以嵌套的**，但是没有优先级的概念，也就是说任何一个新的中断都可以打断正在执行的中断，但同种中断除外。**软中断不能嵌套**，但相同类型的软中断可以在不同CPU上并行执行。

4. **软中断与硬中断之间的区别**

   （1）**硬中断**是由外部事件引起的因此具有随机性和突发性；**软中断**是执行中断指令产生的，无外面事件中断请求信号，因此软中断的发生不是随机的而是由程序安排好的；

   （2）**硬中断**的中断号是由中断控制器提供的；**软中断**的中断号是由指令直接给出的，无需使用中断控制器。

   （3）**硬中断**的中断响应周期，CPU需要发中断回合信号；**软中断**的中断响应周期，CPU不需要发中断回合信号。

   （4）**硬中断**是可屏蔽的；**软中断**是不可屏蔽的。

1.60 介绍一下你对CopyOnWrite的了解。

参考回答

1. CopyOnWrite（写时拷贝技术）

   Linux的fork()使用写时拷贝页来实现新进程的创建，它是一种可推迟甚至避免数据拷贝的技术，开始时内核并不会复制整个地址空间，而是让父子进程共享地址空间，只有在写时才复制地址空间，使得父子进程都拥有独立的地址空间，即资源的复制是在只有需要写入时才会发生。在此之前都是以读的方式去和父进程共享资源，这样，在页根本不会被写入的场景下，fork()立即执行exec()，无需对地址空间进行复制，fork()的实际开销就是复制父进程的一个页表和为子进程创建一个进程描述符，也就是说只有当进程空间中各段的内存内容发生变化时，父进程才将其内容复制一份传给子进程，大大提高了效率。

2. 写时拷贝技术优缺点

写时拷贝技术是一种很重要的优化手段，核心是懒惰处理实体资源请求，在多个实体资源之间只是共享资源，起初是并不真正实现资源拷贝，只有当实体有需要对资源进行修改时才真正为实体分配私有资源。但写时拷贝技术技术也有它的优点和缺点：

优点：写时拷贝技术可以减少分配和复制大量资源时带来的瞬间延时，但实际上是将这种延时附加到了后续的操作之中。

缺点：写时拷贝技术可以减少不必要的资源分配。比如fork进程时，并不是所有的页面都需要复制，父进程的代码段和只读数据段都不被允许修改，所以无需复制。

答案解析

写时复制技术详述：

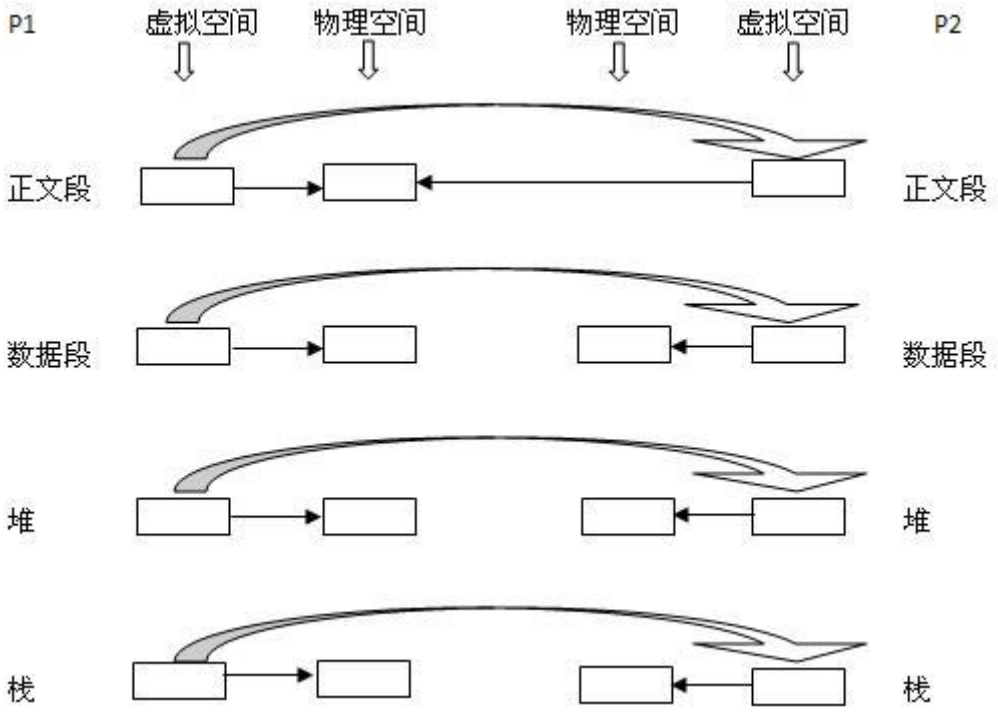
现在有一个父进程P1，这是一个主体，那么它是有灵魂也就身体的。现在在其虚拟地址空间（有相应的数据结构表示）上有：正文段，数据段，堆，栈这四个部分，相应的，内核要为这四个部分分配各自的物理块。即：正文段块，数据段块，堆块，栈块。至于如何分配，这是内核去做的事，在此不详述。

1. 现在P1用fork()函数为进程创建一个子进程P2，

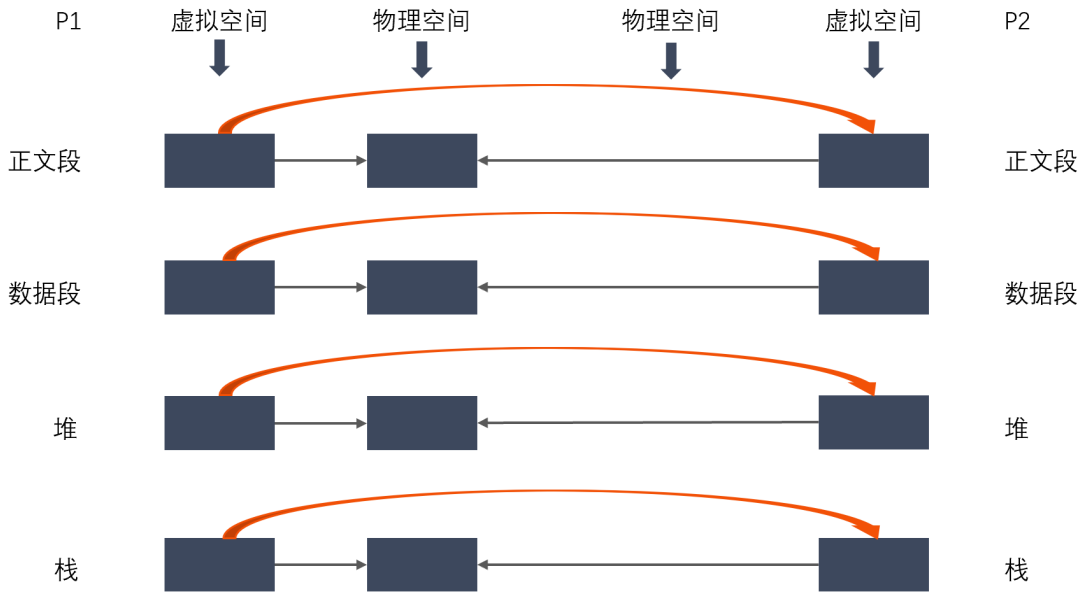
   内核：

   （1）复制P1的正文段，数据段，堆，栈这四个部分，注意是其内容相同。

   （2）为这四个部分分配物理块，P2的：正文段 - > PI的正文段的物理块，其实就是不为P2分配正文段块，让P2的正文段指向P1的正文段块，数据段 - > P2自己的数据段块（为其分配对应的块），堆 - > P2自己的堆块，栈 - > P2自己的栈块。如下图所示：从左到右大的方向箭头表示复制内容。

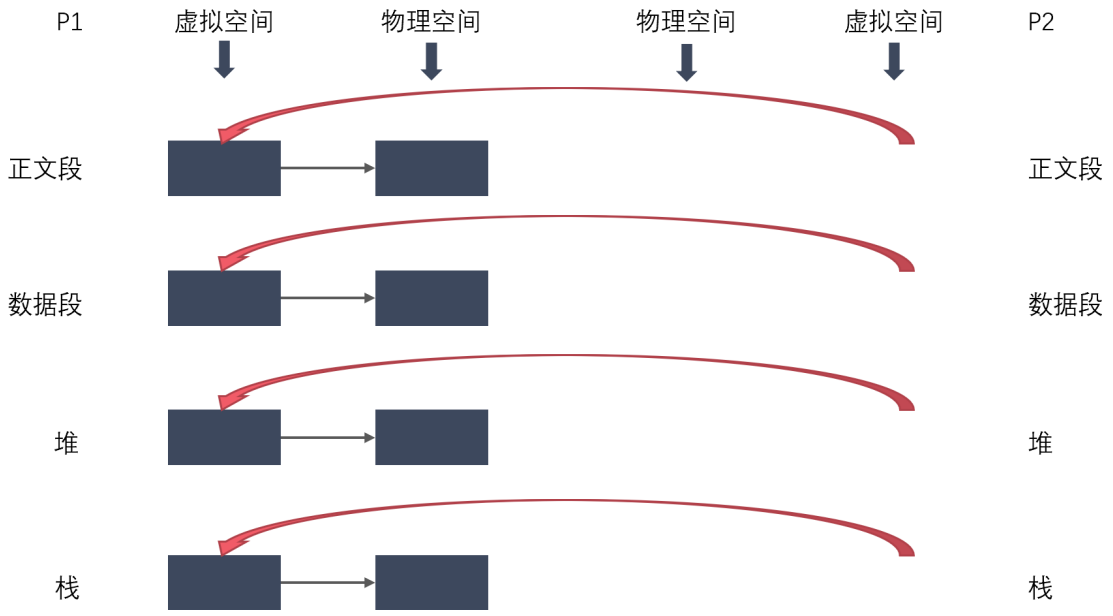


2. 写时复制技术：内核只为新生成的子进程创建虚拟空间结构，它们来复制于父进程的虚拟究竟结构，但是不为这些段分配物理内存，它们共享父进程的物理空间，当父子进程中有更改相应段的行为发生时，再为子进程相应的段分配物理空间。



3. vfork()：这个做法更加火爆，内核连子进程的虚拟地址空间结构也不创建了，直接共享了父进程的虚拟空间，当然了，这种做法就顺

   水推舟的共享了父进程的物理空间!



通过以上的分析，相信大家对进程有个深入的认识，它是怎么一层层体现出自己来的，进程是一个主体，那么它就有灵魂与身体，系统必须为实现它创建相应的实体， 灵魂实体与物理实体。这两者在系统中都有相应的数据结构表示，物理实体更是体现了它的物理意义。以下援引LKD

传统的fork()系统调用直接把所有的资源复制给新创建的进程。这种实现过于简单并且效率低下，因为它拷贝的数据也许并不共享，更糟的情况是，如果新进程打算立即执行一个新的映像，那么所有的拷贝都将前功尽弃。Linux的fork()使用写时拷贝（copy-on-write）页实现。写时拷贝是一种可以推迟甚至免除拷贝数据的技术。内核此时并不复制整个进程地址空间，而是让父进程和子进程共享同一个拷贝。只有在需要写入的时候，数据才会被复制，从而使各个进程拥有各自的拷贝。也就是说，资源的复制只有在需要写入的时候才进行，在此之前，只是以只读方式共享。这种技术使地址空间上的页的拷贝被推迟到实际发生写入的时候。在页根本不会被写入的情况下—举例来说，fork()后立即调用exec()—它们就无需复制了。fork()的实际开销就是复制父进程的页表以及给子进程创建惟一的进程描述符。在一般情况下，进程创建后都会马上运行一个可执行的文件，这种优化可以避免拷贝大量根本就不会被使用的数

据（地址空间里常常包含数十兆的数据）。由于Unix强调进程快速执行的能力，所以这个优化是很重要的。这里补充一点：**Linux COW与exec没有必然联系。**

实际上COW技术不仅仅在Linux进程上有应用，其他例如C++的String在有的IDE环境下也支持COW技术，即例如：

```
string str1 = "hello world"; string str2 = str1;
```

之后执行代码:

```
str1[1]='q'; str2[1]='w';
```

在开始的两个语句后，str1和str2存放数据的地址是一样的，而在修改内容后，str1的地址发生了变化，而str2的地址还是原来的,这就是C++中的COW技术的应用，不过VS2005似乎已经不支持COW。

1.61 Linux替换文本该如何操作呢?

参考回答

1. 通过vi编辑器来替换 **vi/vim 中可以使用 :s 命令来替换字符串。** :s/well/good/ 替换当前行第一个 well 为 good :s/well/good/g 替换当前行所有 well 为 good :n,$s/well/good/$替换第n行开始到最后一行中每一行的第一个$well$为$good$：n,s/well/good/g 替换第 n 行开始到最后一行中每一行所有 well 为 good n 为数字，若 n 为 .，表示从当前行开始到最后一行 :%s/well/good/（等同于 :g/well/s//good/） 替换每一行的第一个 well 为 good :%s/well/good/g（等同于 :g/well/s//good/g） 替换每一行中所有 well 为 good，可以使用 # 作为分隔符，此时中间出现的 / 不会作为分隔符 :s#well/#good/# 替换当前行第一个 well/ 为 good/ :%s#/usr/bin#/bin#g 可以把文件中所有路径/usr/bin换成/bin
2. 直接替换文件中的字符串。(此法不用打开文件即可替换字符串，而且可以批量替换多个文件。)

   （1）**perl命令替换**，参数含义如下： -a 自动分隔模式，用空格分隔$_并保存到@F中。相当于@F = split ”。分隔符可以使用-F参数指定 -F 指定-a的分隔符，可以使用正则表达式 -e 执行指定的脚本。 -i<扩展名> 原地替换文件，并将旧文件用指定的扩展名备份。不指定扩展名则不备份。 -l 对输入内容自动chomp，对输出内容自动添加换行 -n 自动循环，相当于 while(<>) { 脚本; } -p 自动循环+自动输出，相当于 while(<>) { 脚本; print; } **用法示例：** perl -p -i.bak -e 's/\bfoo\b/bar/g' *.c 将所有C程序中的foo替换成bar，旧文件备份成.bak

   perl -p -i -e "s/shan/hua/g" ./lishan.txt ./lishan.txt.bak 将当前文件夹下lishan.txt和lishan.txt.bak中的“shan”都替换为“hua”

   perl -i.bak -pe 's/(\d+)/ 1 + $1 /ge' file1 file2 将每个文件中出现的数值都加一

   （2）**sed命令下批量替换文件内容**

   格式: sed -i "s/查找字段/替换字段/g" grep 查找字段 -rl 路径 文件名

   -i 表示inplace edit，就地修改文件

   -r 表示搜索子目录

   -l 表示输出匹配的文件名 s表示替换，d表示删除

   用法示例：

   sed -i "s/shan/hua/g" lishan.txt

   把当前目录下lishan.txt里的shan都替换为hua

答案解析

sed的其他用法如下：

1. 删除行首空格 sed 's/^[ ]*//g' filename sed 's/^ *//g' filename sed 's/^[[:space:]]*//g' filename
2. 行后和行前添加新行 行后：sed 's/pattern/&\n/g' filename 行前：sed 's/pattern/\n&/g' filename &代表pattern
3. 使用变量替换(使用双引号) sed -e "s/$var1/var2/g" filename
4. 在第一行前插入文本 sed -i '1 i\插入字符串' filename
5. 在最后一行插入 sed -i '$ a\插入字符串' filename
6. 在匹配行前插入 sed -i '/pattern/ i "插入字符串"' filename
7. 在匹配行后插入 sed -i '/pattern/ a "插入字符串"' filename
8. 删除文本中空行和空格组成的行以及#号注释的行 grep -v ^# filename | sed /^[[:space:]]*$/d | sed /^$/d