1.Linux的用户态与内核态的转换方法。（选择题）

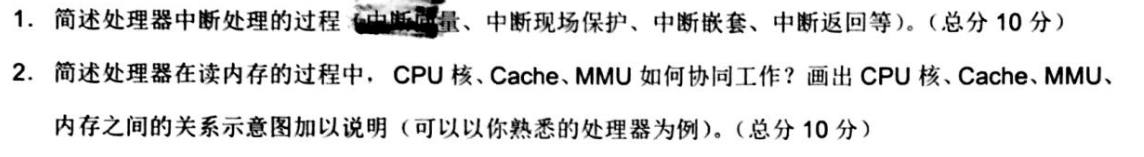
解答：Linux下内核空间与用户空间进行通信的方式主要有syscall（system call）、procfs、ioctl和netlink等。

syscall：一般情况下，用户进程是不能访问内核的。它既不能访问内核所在的内存空间，也不能调用内核中的函数。Linux内核中设置了一组用于实现各种系统功能的子程序，用户可以通过调用他们访问linux内核的数据和函数，这些系统调用接口（SCI）称为系统调用；

procfs：是一种特殊的伪文件系统 ，是Linux内核信息的抽象文件接口，大量内核中的信息以及可调参数都被作为常规文件映射到一个目录树中，这样我们就可以简单直接的通过echo或cat这样的文件操作命令对系统信息进行查取；

netlink：用户态应用使用标准的 socket API 就可以使用 netlink 提供的强大功能；

ioctl：函数是文件结构中的一个属性分量，就是说如果你的驱动程序提供了对ioctl的支持，用户就可以在用户程序中使用ioctl函数控制设备的I/O通道。

2

3. 为什么需要uboot？不用行不行？

  用uboot的目的是引导内核启动。  我理解的，理论是可以的。把uboot中所做的一些工作写进内核里，板子也能启动。但是很少有人这么做，毕竟内核很庞大，大面积修改难度比较大。

4. 两个Linux操作系统之间使用什么命令进行文件的传递？

当两台LINUX主机之间要互传文件时可使用SCP命令来实现

scp传输速度较慢,但使用ssh通道保证了传输的安全性

5. bootloader、内核 、根文件的关系

启动顺序：bootloader->linux kernel->rootfile->app

Bootloader全名为启动引导程序，是第一段代码，它主要用来初始化处理器及外设，然后调用Linux内核。Linux内核在完成系统的初始化之后需要挂载某个文件系统作为根文件系统（RootFilesystem），然后加载必要的内核模块，启动应用程序。（一个嵌入式Linux系统从软件角度看可以分为四个部分：引导加载程序（Bootloader），Linux内核，文件系统，应用程序。）

6. Bootloader启动的两个阶段：

Stage1:汇编语言

1）基本的硬件初始化（关闭看门狗和中断，MMU（带操作系统），CACHE。配置系统工作时钟）

2）为加载stage2准备RAM空间

3）拷贝内核映像和文件系统映像到RAM中

4）设置堆栈指针sp

5）跳到stage2的入口点

Stage2:c语言

1）初始化本阶段要使用到的硬件设备（led uart等）

2）检测系统的内存映射

3）加载内核映像和文件系统映像

4）设置内核的启动参数

嵌入式系统中广泛采用的非易失性存储器通常是Flash，而Bootloader就位于该存储器的最前端，所以系统上电或复位后执行的第一段程序便是Bootloader。

7. 中断和异常的区别

内中断：同步中断（异常）是由cpu内部的电信号产生的中断，其特点为当前执行的指令结束后才转而产生中断，由于有cpu主动产生，其执行点必然是可控的。

外中断：异步中断是由cpu的外设产生的电信号引起的中断，其发生的时间点不可预期。

8. 中断怎么发生，中断处理流程

请求中断→响应中断→关闭中断→保留断点→中断源识别→保护现场→中断服务子程序→恢复现场→中断返回。

9. 在有数据cache情况下，DMA数据链路为：

外设-DMA-DDR-cache-CPU

10. 1）多线程的优点：

无需跨进程边界；

程序逻辑和控制方式简单；

所有线程可以直接共享内存和变量等；

线程方式消耗的总资源比进程方式好；

（2）多线程缺点：

每个线程与主程序共用地址空间，受限于2GB地址空间；

线程之间的同步和加锁控制比较麻烦；

一个线程的崩溃可能影响到整个程序的稳定性；

到达一定的线程数程度后，即使再增加CPU也无法提高性能；

线程能够提高的总性能有限，而且线程多了之后，线程本身的调度也是一个麻烦事儿，需要消耗较多的CPU

（3）多进程优点：

每个进程互相独立，不影响主程序的稳定性，子进程崩溃没关系；

可以尽量减少线程加锁/解锁的影响，极大提高性能，就算是线程运行的模块算法效率低也没关系；

通过增加CPU，就可以容易扩充性能；

每个子进程都有2GB地址空间和相关资源，总体能够达到的性能上限非常大

（4）多进程缺点：

逻辑控制复杂，需要和主程序交互；

多进程调度开销比较大；

需要跨进程边界，如果有大数据量传送，就不太好，适合小数据量传送、密集运算

11. 目录下查找符合文件权限的文件

find /boot -perm 755

12. 多个进程访问同一个文件时，文件描述符是一样的吗

两个进程中分别产生生成两个独立的fd

2.两个进程可以任意对文件进行读写操作，操作系统并不保证写的原子性

3.进程可以通过系统调用对文件加锁，从而实现对文件内容的保护

4.任何一个进程删除该文件时，另外一个进程不会立即出现读写失败

5.两个进程可以分别读取文件的不同部分而不会相互影响

6.一个进程对文件长度和内容的修改另外一个进程可以立即感知

13. 查看linux系统是64位的还是32系统：

getconf LONG\_BIT

uname -a

cat /proc/cpuinfo

c语言编写函数，输出指针大小，为4则是32位，为8则是64位

14. 一个进程中最多可以有多少个线程？  
在上一篇文章Linux中线程占用内存中，我们知道了创建一个线程会占用多少内存，这取决于分配给线程的调用栈大小，可以用ulimit -s命令来查看大小（一般常见的有10M或者是8M）。我们还知道，一个进程的虚拟内存是4G，在Linux32位平台下，内核分走了1G，留给用户用的只有3G，于是我们可以想到，创建一个线程占有了10M内存，总共有3G内存可以使用。于是可想而知，最多可以创建差不多300个左右的线程。因此，进程最多可以创建的线程数是根据分配给调用栈的大小，以及操作系统（32位和64位不同）共同决定的。

Linux中最多可以有多少个进程？  
一.Linux中有一个命令可以帮助我们查看系统中的进程上限[pigff@izbp13yd4r85qvk53t04mbz ~]$ ulimit -u  
4096这属于软限制，是可以改变的。也就是说在我的机器上最多可以有4096个进程，但是我可以通过改变这个参数的值来修改对于进程数量的软限制，比如说用下面的命令将软限制改到5120。 ulimit -u 5120

15. freeRTOS里的信号量，邮箱讲讲

16. 中断函数能不能申请内存；

一个是只要是内存动态申请，必然涉及到了堆，所以必然是要做共享资源保护的，FreeRTOS里，用了临界区，没毛病。第二，只要是内存的动态管理，必然涉及到了复杂的算法，他的运行时间是带有很强的不确定性的。在中断里跑算法，跑耗时的操作，只会是坑你自己而已。

当你在一个多线程环境内使用堆时，会出现一类新的问题。堆变成一个共享的资源，因此引发了全部和资源共享有关的麻烦，新问题是：  
λ malloc() 和 free() （常常）是不可重入的，也就是说，它们不能从多个执行线程被安全的调用。  
λ 可以通过使用一个互斥体保护 malloc() ， free() ， realloc() 这类函数来弥补重入的问题，互斥体 mutex 仅允许每次一个线程存取共享的堆。然而，这个方案可能造成过度的线程阻塞（特别是如果内存管理是不可确定的）并且可以显著的降低并行性。互斥体也可以造成优

先级倒置。一般的，被一个互斥体保护的堆管理函数不适用于 ISR 因为 ISR 不能阻塞。

17. 为了让程序拥有更好的性能，有时候需要将进程或线程绑定到特定的CPU，这样可以减少调度的开销和保护关键进程或线程。

**进程绑定到CPU**

Linux提供一个接口，可以将进程绑定到特定的CPU：

#include <sched.h>

int sched\_setaffinity(pid\_t pid, size\_t cpusetsize, const cpu\_set\_t \*mask);

int sched\_getaffinity(pid\_t pid, size\_t cpusetsize, cpu\_set\_t \*mask);

参数

pid：进程的id号，如果pid为0，则表示本进程

cpusetsize：mask的大小

mask：运行进程的CPU，可以通过以下函数操作mask

#define CPU\_SET(cpu, cpusetp) //设置cpu

#define CPU\_CLR(cpu, cpusetp) //删除cpu

#define CPU\_ISSET(cpu, cpusetp) //判断cpu

#define CPU\_ZERO(cpusetp) //初始化为0

18. 现在的fork都在用写时拷贝技术，顾名思义就是数据段，堆，栈一开始并不复制，由父、字进程共享，并将这些内存设置为只读。直到父，子进程一方尝试写这些区域，内核才为需要修改的那片内存拷贝副本，这样做可以提高fork的效率。

19 用一行命令查找某个文件并重命名它

find ./ -type f -name "\*" -exec rename 's/\.h/\.j/' \* {} \;

#### 20. 调用函数时，有那些内容需要压栈？

参数入栈：将参数从右向左依次压入系统栈中  
返回地址入栈：将当前代码区调用指令的下一条指令地址压入栈中，供函数返回时继续执行  
代码区跳转：处理器从当前代码区跳转到被调用函数的入口处  
栈帧调整：具体包括  
保存当前栈帧状态值，已备后面恢复本栈帧时使用（EBP入栈）  
将当前栈帧切换到新栈帧。（将ESP值装入EBP，更新栈帧底部）  
给新栈帧分配空间。（把ESP减去所需空间的大小，抬高栈顶）

21. **ThreadLocal与其它同步机制的比较：**

Threadlocal和其他所有的同步机制都是为了解决多线程中的对同一变量的访问冲突，在普通的同步机制中，是通过对对象加锁来实现多个线程对同一变量的安全访问的。这时该变量是多个线程共享的，使用这种同步机制需要很细致的分析在什么时候对变量进行读写，什么时候需要锁定某个对象，什么时候释放该对象的索等等。所有这些都是因为多个线程共享了该资源造成的。Threadlocal就从另一个角度来解决多线程的并发访问，Threadlocal会为每一个线程维护一个和该线程绑定的变量副本，从而隔离了多个线程的数据共享，每一个线程都拥有自己的变量副本，从而也就没有必要对该变量进行同步了。ThreadLocal提供了线程安全的共享对象，在编写多线程代码时，可以把不安全的变量封装进ThreadLocal。

总结：当然ThreadLocal并不能替代同步机制，两者面向的问题领域不同。同步机制是为了同步多个线程对相同资源的并发访问，是为了多个线程之间进行通信的有效方式；而ThreadLocal是隔离多个线程的数据共享，从根本上就不在多个线程之间共享资源（变量），这样当然不需要对多个线程进行同步了。所以，如果你需要进行多个线程之间进行通信，则使用同步机制；如果需要隔离多个线程之间的共享冲突，可以使用ThreadLocal，这将极大地简化你的程序，使程序更加易读、简洁。

22**在 Linux 上，在相同进程的不同线程之间，则只使用 POSIX 信号量；在进程之间，可以使用 System V 信号量。**

**23互斥量和信号量的区别：**

**互斥量用于线程的互斥，信号量用于线程的同步。这是互斥量和信号量的根本区别，也就是互斥和同步之间的区别；**

**互斥量值只能为0/1，信号量值可以为非负整数。也就是说，一个互斥量只能用于一个资源的互斥访问，它不能实现多个资源的多线程互斥问题。信号量可以实现多个同类资源的多线程互斥和同步。当信号量为单值信号量是，也可以完成一个资源的互斥访问；**

**互斥量的加锁和解锁必须由同一线程分别对应使用，信号量可以由一个线程释放，另一个线程得到。**

**这里区分一下互斥和同步：**

**互斥：是指某一资源同时只允许一个访问者对其进行访问，具有唯一性和排它性。但互斥无法限制访问者对资源的访问顺序，即访问是无序的；**

**同步：是指在互斥的基础上（大多数情况），通过其它机制实现访问者对资源的有序访问。在大多数情况下，同步已经实现了互斥，特别是所有写入资源的情况必定是互斥的。少数情况是指可以允许多个访问者同时访问资源。**

### 24简述处理器在读内存的过程中，CPU核、cache、MMU如何协同工作？画出CPU核、cache、MMU、内存之间的关系示意图加以说明（可以以你熟悉的处理器为例）

25 为什么栈的空间不连续

局部变量的在栈上的布局受众多因素影响，比如字节对齐，release/debug 编译器一些check选项都会影响局部变量栈上的布局。不过先定义的局部变量处在高地址，后定义的处于地址，这个在绝大多数编译器基本上是确定的。

在这里,间隔增加的话,可以防止一些错误.  
比如  
int a, b;  
如果相邻,那么  
\*((&b) + 1) = 1;  
就很有可能影响到a.  
如果间隔增大,就影响的可能性"小"

在release 底下，就是正常的

26进程是具有一定独立功能的程序关于某个数据集合上的一次运行活动，它是系统进行资源分配和调度的一个独立单位。例如，用户运行自己的程序，系统就创建一个进程，并为它分配资源，包括各种表格、内存空间、磁盘空间、IO设备等，然后该进程被放入到进程的就绪队列，进程调度程序选中它，为它分配CPU及其他相关资源，该进程就被运行起来。

  线程是进程的一个实体，是CPU调度和分配的基本单位，线程自己基本上不拥有系统资源，只拥有一些在运行中必不可少的资源（如程序计数器、一组寄存器和栈），但是，它可以与同属一个进程的其他的线程共享进程所拥有的全部资源。

  在没有实现线程的操作系统中，进程既是资源分配的基本单位，又是调度的基本单位，它是系统中并发执行的单元。而在实现了线程的操作系统中，进程是资源分配的基本单位而线程是调度的基本单位，是系统中并发执行的单元。

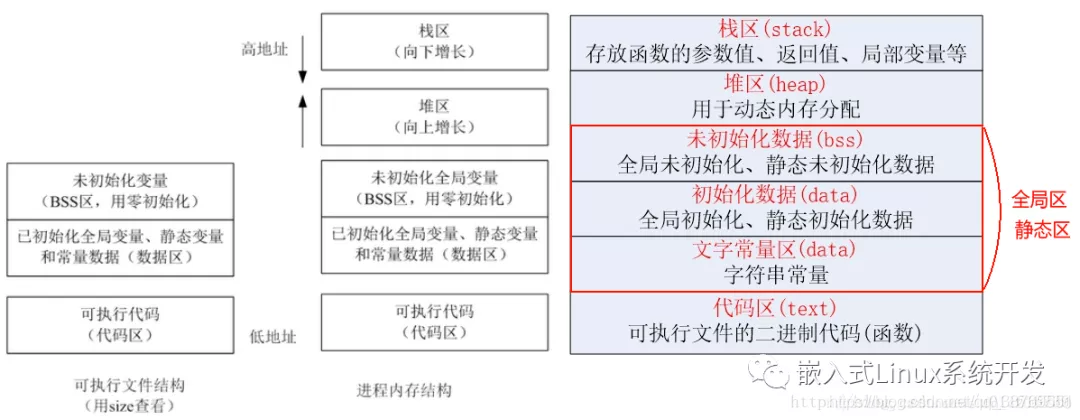
  不全。面试官又提醒了。进程间通信方式有那些，也回答上了。管道，FIFO，信号，信号量，消息队列，共享内存（最快），套接字。

27

不能用 sizeof（）函数，如何判断操作系统是16位，还是32位

  16位系统中，int变量的范围-32768到+32767,32767+1变为-32768。可以利用这个特性来判断。

28一个进程能不能绑定到一个cpu？系统调用接口是什么？线程呢？



29**如何申请一块超大连续内存？**

进程的4G内存空间都放着或被map着什么:

第0G和第1G:用户态地址空间

第2G:库函数映射等

第3G:内核态内存空间

用户态地址空间中还包含了进程代码本身占用的地址空间, 栈的空间等等.

第2G中, 库函数映射等只占用了很少的一部分空间，还有很多的空闲空间.

现在让我们解释这4个问题:

第(1)个问题, 由上图可以看出, 没有连续的2G的内存, 所以申请2G的连续内存是肯定失败的.

第(2), 申请1.9G的空间是成功的, 这是因为前两个G可能会有1.9G的连续空间.

第(3), 申请了300M\*9 = 2.7G是成功的, 是的, 前3G中有可能空间着2.7G的空间, 前两个G中空闲的加上第3个G中空闲的部分. 但是如果一次申请2.7G是不行的, 因为没有连续的2.7G的地址空间. 最后一个300M没有申请成功的原因是, 申请的空间大小不能超过3G的用户态地址空间.

第(4), 比较有意思, 显然那个1.9G是在第1-2G这个地址空间中申请成功的, 后900M是第3个G这片地址空间中申请成功的. 我们一共申请到了2.8G的”内存”, 却也不是连续的

30

下面不属于cache的映射方式的是（）。

full-associative（全关联）

direct-mapping（直接映射）

indirect-mapping（间接映射）

set-associative（组相关）

KEY：C

解答：cache是一种高速缓冲寄存器，是为解决CPU和主存之间速度不匹配而采用的一项重要技术。主存与cache的地址映射方式有全相联方式、直接方式和组相联方式三种：

直接映射（direct-mapping）：将一个主存块存储到唯一的一个Cache行；

全相联映射（full-associative）：可以将一个主存块存储到任意一个Cache行；

组相联映射（set-associative）：可以将一个主存块存储到唯一的一个Cache组中任意一个行。

31Linux程序如何定位内存泄漏？

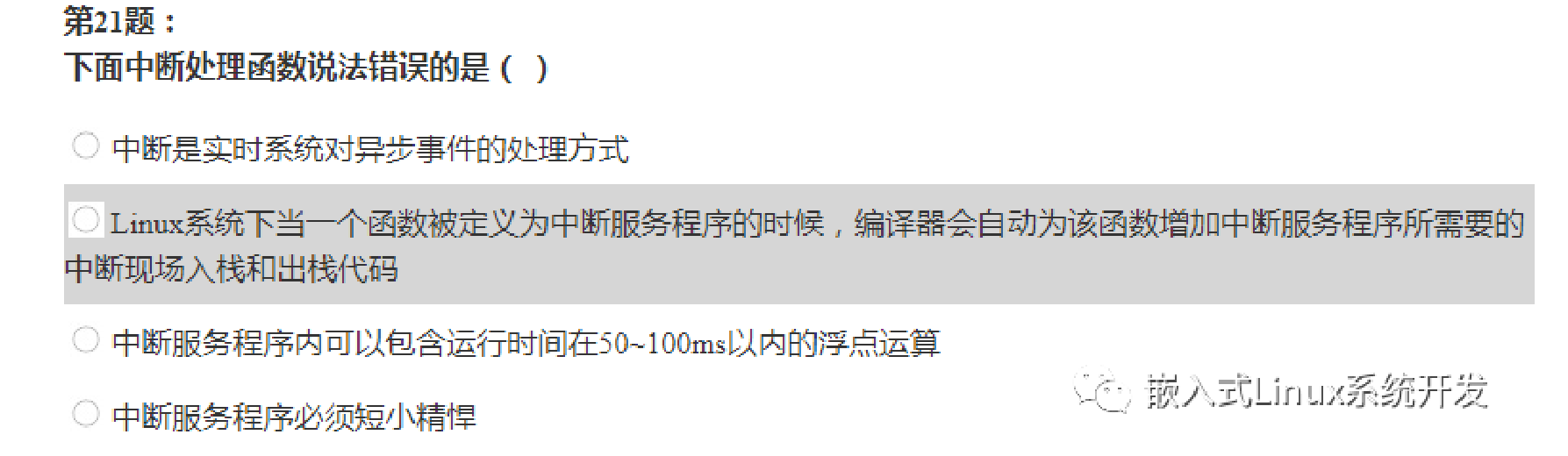
答案：如果在实际的调试过程中，怀疑某处发生了内存泄露，可以查看该进程的maps表，看进程的堆段或者mmap段的虚拟地址空间是否持续增加，如果是，说明很可能发生了内存泄露，如果mmap段虚拟地址空间持续增加，还可以看到各个段的虚拟地址空间的大小，从而可以确定是申请了多大的内存，对调试内存泄露类问题可以起到很好的定位作用

[**Linux**](http://lib.csdn.net/base/linux) 下面查看内存有多种渠道，比如通过命令 ps ,top,free 等，比如通过/proc系统，一般需要比较详细和精确地知道整机内存/某个进程内存的使用情况，最好通过/proc 系统

32系统“抖动”的原因是

如果进程没有需要[支持](https://www.jb51.cc/tag/zhichi/)活动使用[页面](https://www.jb51.cc/tag/yemian/)的帧数，那么它会很快产生缺页[错误](https://www.jb51.cc/tag/cuowu/)。此时，必须置换某个[页面](https://www.jb51.cc/tag/yemian/)。然而，由于它的所有[页面](https://www.jb51.cc/tag/yemian/)都在使用中，所以必须立即置换需要再次使用的[页面](https://www.jb51.cc/tag/yemian/)。因此，它会再次[快速](https://www.jb51.cc/tag/kuaisu/)产生缺页[错误](https://www.jb51.cc/tag/cuowu/)，再一次置换必须立即返回的[页面](https://www.jb51.cc/tag/yemian/)，如此[快速](https://www.jb51.cc/tag/kuaisu/)进行

33



34

linux内存碎片化的定义，可能的原因以及可以避免或者缓解的方式？

定义：内部碎片与外部碎片

在页式虚拟存储系统中, 用户作业的地址空间被划分成若干大小相等的页面, 存储空间也分成也页大小相

等的物理块, 但一般情况下, 作业的大小不可能都是物理块大小的整数倍, 因此作业的最后一页中仍有部分

空间被浪费掉了. 由此可知, 页式虚拟存储系统中存在内碎片.

在段式虚拟存储系统中, 作业的地址空间由若干个逻辑分段组成, 每段分配一个连续的内存区, 但各段之间

不要求连续, 其内存的分配方式类似于动态分区分配.由此可知, 段式虚拟存储系统中存在外碎片.

换一种问法也是一样的：

在内存管理中, “内零头”和”外零头”个指的是什么?在固定式分区分配, 可变式分区分配, 页式虚拟存储系统,

段式虚拟存储系统中, 各会存在何种碎片? 为什么?

解答:

在存储管理中内碎片是指分配给作业的存储空间中未被利用的部分。

在固定式分区分配中, 为将一个用户作业装入内存, 内存分配程序从系统分区表中找出一个能满足作业要

求的空闲分区分配给作业, 由于一个作业的大小并不一定与分区大小相等, 因此, 分区中有一部分存储空间

浪费掉了. 由此可知, 固定式分区分配中存在内碎片.

外碎片是指系统中无法利用的小存储块。

在可变式分区分配中, 为把一个作业装入内存, 应按照一定的分配算法从系统中找出一个能满足作业需求

的空闲分区分配给作业, 如果这个空闲分区的容量比作业申请的空间容量要大, 则将该分区一分为二, 一部

分分配给作业, 剩下的部分仍然留作系统的空闲分区。由此可知，可变式分区分配中存在外碎片.

简言之：

在固定分区管理算法中, 分给程序的内存空间往往大于程序所需的空间, 这剩余部分的空间不能被其他程

序所用, 这就是”内部碎片”

随着存储区的分配和释放过程的进行, 在各个被分配出去的分区之间会存在很多的小空闲区, 暂时不能被

利用, 这就是”外部碎片”.

避免方式：

1、依据可移动性组织页是防止物理内存碎片的一种可能方法.

2、虚拟可移动内存域避免内存碎片

35

假设某一虚拟存储系统采用先进先出（FIFO）页面淘汰算法，有一个进程在内存中占3页（开始时

内存为空），当访问如下页面序号（1,2，3,1,2,4,2,3,5,3,4,5）后，会产生几次缺页？

解析：

1  1 （一次）

2  1 2（一次）

3  1 2 3（一次）

1  1 2 3

2  1 2 3

4  2 3 4（一次）

2  2 3 4

3  2 3 4

5  3 4 5（一次）

3  3 4 5

4  3 4 5

5  3 4 5

B 5次

1,2,3  miss 3次 \*\*\* 为 1,2,3

1, 2  hit

4    miss 1次，此时1出4进 此时\*\*\*为2,3,4

2,3   hit

5    miss，2出5进，\*\*\*为3，4,5

3,4,5  hit

总计5次