1、进程、线程和协程的区别和联系

	进程	线程	协程
定义	资源分配和拥有的基本单位	程序执行的基本单位	用户态的轻量级线程，线程内部调度的基本单位
切换情况	进程 CPU 环境(栈、寄存器、页表和文件句柄等)的保存以及新调度的进程 CPU 环境的设置	保存和设置程序计数器、少量寄存器和栈的内容	先将寄存器上下文和栈保存，等切换回来的时候再进行恢复
切换者	操作系统	操作系统	用户
切换过程	用户态->内核态->用户态	用户态->内核态->用户态	用户态(没有陷入内核)
调用栈	内核栈	内核栈	用户栈
拥有资源	CPU 资源、内存资源、文件资源和句柄等	程序计数器、寄存器、栈和状态字	拥有自己的寄存器上下文和栈
并发性	不同进程之间切换实现并发，各自占有 CPU 实现并行	一个进程内部的多个线程并发执行	同一时间只能执行一个协程，而其他协程处于休眠状态，适合对任务进行分时处理
系统开销	切换虚拟地址空间，切换内核栈和硬件上下文，CPU 高速缓存失效、页表切换，开销很大	切换时只需保存和设置少量寄存器内容，因此开销很小	直接操作栈则基本没有内核切换的开销，可以不加锁的访问全局变量，所以上下文的切换非常快
通信方面	进程间通信需要借助操作系统	线程间可以直接读写进程数据段(如全局变量)来进行通信	共享内存、消息队列

1、进程是资源分配的基本单位，运行一个可执行程序会创建一个或多个进程，进程就是运行起来的可执行程序

2、线程是资源调度的基本单位，也是程序执行的基本单位，是轻量级的进程。每个进程中都有唯一的主线程，且只能有一个，主线程和进程是相互依存的关系，主线程结束进程也会结束。多提一句：协程是用户态的轻量级线程，线程内部调度的基本单位

2、线程与进程的比较

1、线程启动速度快，轻量级

2、线程的系统开销小

3、线程使用有一定难度，需要处理数据一致性问题

4、同一线程共享的有堆、全局变量、静态变量、指针，引用、文件等，而独自占有栈

2.2、补充另一种问法

线程和进程的区别？

1. 调度：线程是调度的基本单位（PC，状态码，通用寄存器，线程栈及栈指针）；进程是拥有资源的基本单位（打开文件，堆，静态区，代码段等）。
2. 并发性：一个进程内多个线程可以并发（最好和 CPU 核数相等）；多个进程可以并发。
3. 拥有资源：线程不拥有系统资源，但一个进程的多个线程可以共享隶属进程的资源；进程是拥有资源的独立单位。
4. 系统开销：线程创建销毁只需要处理 PC 值，状态码，通用寄存器值，线程栈及栈指针即可；进程创建和销毁需要重新分配及销毁 task_struct 结构。

3、一个进程可以创建多少线程，和什么有关？

这个要分不同系统去看：

- 如果是 32 位系统，用户态的虚拟空间只有 3G，如果创建线程时分配的栈空间是 10M，那么一个进程最多只能创建 300 个左右的线程。
- 如果是 64 位系统，用户态的虚拟空间大到有 128T，理论上不会受虚拟内存大小的限制，而会受系统的参数或性能限制。

顺便多说一句，过多的线程将会导致大量的时间浪费在线程切换上，给程序运行效率带来负面影响，无用线程要及时销毁。

4、外中断和异常有什么区别？

外中断是指由 CPU 执行指令以外的事件引起，如 I/O 完成中断，表示设备输入/输出处理已经完成，处理器能够发送下一个输入/输出请求。此外还有时钟中断、控制台中断等。

而异常时由 CPU 执行指令的内部事件引起，如非法操作码、地址越界、算术溢出等。

5、进程线程模型你知道多少？

对于进程和线程的理解和把握可以说基本奠定了对系统的认知和把控能力。其核心意义绝不仅仅是“线程是调度的基本单位，进程是资源分配的基本单位”这么简单。

多线程

我们这里讨论的是用户态的多线程模型，同一个进程内部有多个线程，所有的线程共享同一个进程的内存空间，进程中定义的全局变量会被所有的线程共享，比如有全局变量 int i = 10，这一进程中所有并发运行的线程都可以读取和修改这个 i 的值，而多个线程被 CPU 调度的顺序又是不可控的，所以对临界资源的访问尤其需要注意安全。

我们必须知道，**做一次简单的 i = i + 1 在计算机中并不是原子操作，涉及内存取数，计算和写入内存几个环节，**而线程的切换有可能发生在上述任何一个环节中间，所以不同的操作顺序很有可能带来意想不到的结果。

但是，虽然线程在安全性方面会引入许多新挑战，但是线程带来的好处也是有目共睹的。首先，原先顺序执行的程序（暂时不考虑多进程）可以被拆分成几个独立的逻辑流，这些逻辑流可以独立完成一些任务（最好这些任务是不相关的）。

比如 QQ 可以一个线程处理聊天一个线程处理上传文件，两个线程互不干涉，在用户看来是同步在执行两个任务，试想如果线性完成这个任务的话，在数据传输完成之前用户聊天被一直阻塞会是多么尴尬的情况。

对于线程，我认为弄清以下两点非常重要：

线程之间有无先后访问顺序（线程依赖关系）

多个线程共享访问同一变量（同步互斥问题）

另外，我们通常只会去说同一进程的多个线程共享进程的资源，但是每个线程特有的部分却很少提及，除了标识线程的 tid，每个线程还有自己独立的栈空间，线程彼此之间是无法访问其他线程栈上内容的。

而作为处理机调度的最小单位，线程调度只需要保存线程栈、寄存器数据和 PC 即可，相比进程切换开销要小很多。

线程相关接口不少，主要需要了解各个参数意义和返回值意义。

1. 线程创建和结束
    - 背景知识：

      在一个文件内的多个函数通常都是按照main函数中出现的顺序来执行，但是在分时系统下，我们可以让每个函数都作为一个逻辑流并发执行，最简单的方式就是采用多线程策略。在main函数中调用多线程接口创建线程，每个线程对应特定的函数（操作），这样就可以不按照main函数中各个函数出现的顺序来执行，避免了忙等的情况。线程基本操作的接口如下。

    - 相关接口：
        - 创建线程：int pthread_create(pthread_t *tidp,const pthread_attr_t *attr, void *(*start_rtn)(void),void *arg);

          创建一个新线程，pthread和start_routine不可或缺，分别用于标识线程和执行体入口，其他可以填NULL。

            - pthread：用来返回线程的tid，*pthread值即为tid，类型pthread_t == unsigned long int。
            - attr：指向线程属性结构体的指针，用于改变所创线程的属性，填NULL使用默认值。
            - start_routine：线程执行函数的首地址，传入函数指针。
            - arg：通过地址传递来传递函数参数，这里是无符号类型指针，可以传任意类型变量的地址，在被传入函数中先强制类型转换成所需类型即可。
        - 获得线程ID：pthread_t pthread_self();

          调用时，会打印线程ID。
        - 等待线程结束：int pthread_join(pthread_t tid, void** retval);

          主线程调用，等待子线程退出并回收其资源，类似于进程中wait/waitpid回收僵尸进程，调用pthread_join的线程会被阻塞。

            - tid：创建线程时通过指针得到tid值。
            - retval：指向返回值的指针。
        - 结束线程：pthread_exit(void *retval);

          子线程执行，用来结束当前线程并通过retval传递返回值，该返回值可通过pthread_join获得。

            - retval：同上。
        - 分离线程：int pthread_detach(pthread_t tid);

          主线程、子线程均可调用。主线程中pthread_detach(tid)，子线程中pthread_detach(pthread_self())，调用后和主线程分离，子线程结束时自己立即回收资源。

            - tid：同上。

2. 线程属性值修改

- 背景知识：

  线程属性对象类型为pthread_attr_t，结构体定义如下：

```
typedef struct{
    int detachstate;    // 线程分离的状态
    int schedpolicy;    // 线程调度策略
    struct sched_param schedparam;    // 线程的调度参数
    int inheritsched;    // 线程的继承性
    int scope;    // 线程的作用域
    // 以下为线程栈的设置
    size_t guardsize;    // 线程栈末尾警戒缓冲大小
    int stackaddr_set;    // 线程的栈设置
    void *    stackaddr;    // 线程栈的位置
    size_t stacksize;    // 线程栈大小
}pthread_attr_t;
```

- 相关接口：

  对上述结构体中各参数大多有：pthread_attr_get()和pthread_attr_set()系统调用函数来设置和获取。这里不一一罗列。

多进程

每一个进程是资源分配的基本单位。

进程结构由以下几个部分组成：代码段、堆栈段、数据段。代码段是静态的二进制代码，多个程序可以共享。

实际上在父进程创建子进程之后，父、子进程除了 pid 外，几乎所有的部分几乎一样。

父、子进程共享全部数据，但并不是说他们就是对同一块数据进行操作，子进程在读写数据时会通过写时复制机制将公共的数据重新拷贝一份，之后在拷贝出的数据上进行操作。

如果子进程想要运行自己的代码段，还可以通过调用 execv()函数重新加载新的代码段，之后就和父进程独立开了。

我们在 shell 中执行程序就是通过 shell 进程先 fork()一个子进程再通过 execv()重新加载新的代码段的过程。

进程创建与结束

背景知识：

进程有两种创建方式，一种是操作系统创建的一种是父进程创建的。从计算机启动到终端执行程序的过程为：0 号进程 -> 1 号内核进程 -> 1 号用户进程(init 进程) -> getty 进程 -> shell 进程 -> 命令行执行进程。所以我们在命令行中通过 ./program 执行可执行文件时，所有创建的进程都是 shell 进程的子进程，这也就是为什么 shell 一关闭，在 shell 中执行的进程都自动被关闭的原因。从 shell 进程到创建其他子进程需要通过以下接口。

相关接口：

- 创建进程：pid_t fork(void);
- 返回值：出错返回-1；父进程中返回 pid > 0；子进程中 pid == 0
- 结束进程：void exit(int status);

  status 是退出状态，保存在全局变量中 S?，通常 0 表示正常退出。

- 获得 PID：pid_t getpid(void);

  返回调用者 pid。

- 获得父进程 PID：pid_t getppid(void);

  返回父进程 pid。

其他补充：

- 正常退出方式：exit()、_exit()、return（在 main 中）。

  exit()和_exit()区别：exit()是对__exit()的封装，都会终止进程并做相关收尾工作，最主要的区别是_exit()函数关闭全部描述符和清理函数后不会刷新流，但是 exit()会在调用_exit()函数前刷新数据流。

  return 和 exit()区别：exit()是函数，但有参数，执行完之后控制权交给系统。return 若是在调用函数中，执行完之后控制权交给调用进程，若是在 main 函数中，控制权交给系统。

- 异常退出方式：abort()、终止信号。

Linux 进程控制

- 进程地址空间（地址空间）

  虚拟存储器为每个进程提供了独占系统地址空间的假象。

  尽管每个进程地址空间内容不尽相同，但是他们的都有相似的结构。X86 Linux 进程的地址空间底部是保留给用户程序的，包括文本、数据、堆、栈等，其中文本区和数据区是通过存储器映射方式将磁盘中可执行文件的相应段映射至虚拟存储器地址空间中。

  有一些"敏感"的地址需要注意下，对于 32 位进程来说，代码段从 0x08048000 开始。从 0xC0000000 开始到 0xFFFFFFFF 是内核地址空间，通常情况下代码运行在用户态（使用 0x00000000 ~ 0xC0000000 的用户地址空间），当发生系统调用、进程切换等操作时 CPU 控制寄存器设置模式位，进入内和模式，在该状态（超级用户模式）下进程可以访问全部存储器位置和执行全部指令。

也就说 32 位进程的地址空间都是 4G，但用户态下只能访问低 3G 的地址空间，若要访问 3G ~ 4G 的地址空间则只有进入内核态才行。

- 进程控制块（处理机）

  进程的调度实际就是内核选择相应的进程控制块，被选择的进程控制块中包含了一个进程基本的信息。

- 上下文切换

  内核管理所有进程控制块，而进程控制块记录了进程全部状态信息。每一次进程调度就是一次上下文切换，所谓的上下文本质上就是当前运行状态，主要包括通用寄存器、浮点寄存器、状态寄存器、程序计数器、用户栈和内核数据结构（页表、进程表、文件表）等。

  进程执行时刻，内核可以决定抢占当前进程并开始新的进程，这个过程由内核调度器完成，当调度器选择了某个进程时称为该进程被调度，该过程通过上下文切换来改变当前状态。

  一次完整的上下文切换通常是进程原先运行于用户态，之后因系统调用或时间片到切换到内核态执行内核指令，完成上下文切换后回到用户态，此时已切换到进程 B。

6、进程调度算法你了解多少？

1、 先来先服务 first-come first-serverd（FCFS）

非抢占式的调度算法，按照请求的顺序进行调度。

有利于长作业，但不利于短作业，因为短作业必须一直等待前面的长作业执行完毕才能执行，而长作业又需要执行很长时间，造成了短作业等待时间过长。

2、 短作业优先 shortest job first（SJF）

非抢占式的调度算法，按估计运行时间最短的顺序进行调度。

长作业有可能会饿死，处于一直等待短作业执行完毕的状态。因为如果一直有短作业到来，那么长作业永远得不到调度。

3、最短剩余时间优先 shortest remaining time next（SRTN）

最短作业优先的抢占式版本，按剩余运行时间的顺序进行调度。 当一个新的作业到达时，其整个运行时间与当前进程的剩余时间作比较。

如果新的进程需要的时间更少，则挂起当前进程，运行新的进程。否则新的进程等待。

4、时间片轮转

将所有就绪进程按 FCFS 的原则排成一个队列，每次调度时，把 CPU 时间分配给队首进程，该进程可以执行一个时间片。

当时间片用完时，由计时器发出时钟中断，调度程序便停止该进程的执行，并将它送往就绪队列的末尾，同时继续把 CPU 时间分配给队首的进程。

时间片轮转算法的效率和时间片的大小有很大关系：

- 因为进程切换都要保存进程的信息并且载入新进程的信息，如果时间片太小，会导致进程切换得太频繁，在进程切换上就会花过多时间。
- 而如果时间片过长，那么实时性就不能得到保证。

5、优先级调度

为每个进程分配一个优先级，按优先级进行调度。

为了防止低优先级的进程永远等不到调度，可以随着时间的推移增加等待进程的优先级。

6、多级反馈队列

一个进程需要执行 100 个时间片，如果采用时间片轮转调度算法，那么需要交换 100 次。

多级队列是为这种需要连续执行多个时间片的进程考虑，它设置了多个队列，每个队列时间片大小都不同，例如 1,2,4,8,..。进程在第一个队列没执行完，就会被移到下一个队列。

这种方式下，之前的进程只需要交换 7 次。每个队列优先权也不同，最上面的优先权最高。因此只有上一个队列没有进程在排队，才能调度当前队列上的进程。

可以将这种调度算法看成是时间片轮转调度算法和优先级调度算法的结合。

8、Linux 下同步机制？

POSIX 信号量：可用于进程同步，也可用于线程同步。

POSIX 互斥锁 + 条件变量：只能用于线程同步。

10、内存交换和覆盖有什么区别？

交换技术主要是在不同进程（或作业）之间进行，而覆盖则用于同一程序或进程中。

7、Linux 下进程间通信方式？

- 管道：

  无名管道（内存文件）：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之间使用。进程的亲缘关系通常是指父子进程关系。

  有名管道（FIFO 文件，借助文件系统）：有名管道也是半双工的通信方式，但是允许在没有亲缘关系的进程之间使用，管道是先进先出的通信方式。

- 共享内存：共享内存就是映射一段能被其他进程所访问的内存，这段共享内存由一个进程创建，但多个进程都可以访问。共享内存是最快的 IPC 方式，它是针对其他进程间通信方式运行效率低而专门设计的。它往往与信号量，配合使用来实现进程间的同步和通信。
- 消息队列：消息队列是有消息的链表，存放在内核中并由消息队列标识符标识。消息队列克服了信号传递信息少、管道只能承载无格式字节流以及缓冲区大小受限等缺点。
- 套接字：适用于不同机器间进程通信，在本地也可作为两个进程通信的方式。
- 信号：用于通知接收进程某个事件已经发生，比如按下 ctrl + C 就是信号。
- 信号量：信号量是一个计数器，可以用来控制多个进程对共享资源的访问。它常作为一种锁机制，实现进程、线程的对临界区的同步及互斥访问。

9、如果系统中具有快表后，那么地址的转换过程变成什么样了？

①CPU 给出逻辑地址，由某个硬件算得页号、页内偏移量，将页号与快表中的所有页号进行比较。

②如果找到匹配的页号，说明要访问的页表项在快表中有副本，则直接从中取出该页对应的内存块号，再将内存块号与页内偏移量拼接形成物理地址，最后，访问该物理地址对应的内存单元。因此，若快表命中，则访问某个逻辑地址仅需一次访存即可。

③如果没有找到匹配的页号，则需要访问内存中的页表，找到对应页表项，得到页面存放的内存块号，再将内存块号与页内偏移量拼接形成物理地址，最后，访问该物理地址对应的内存单元。因此,若快表未命中，则访问某个逻辑地址需要两次访存(注意:在找到页表项后，应同时将其存入快表,以便后面可能的再次访问。但若快表已满，则必须按照-定的算法对旧的页表项进行替换)

由于查询快表的速度比查询页表的速度快很多，因此只要快表命中，就可以节省很多时间。因为局部性原理，–般来说快表的命中率可以达到 90%以上。

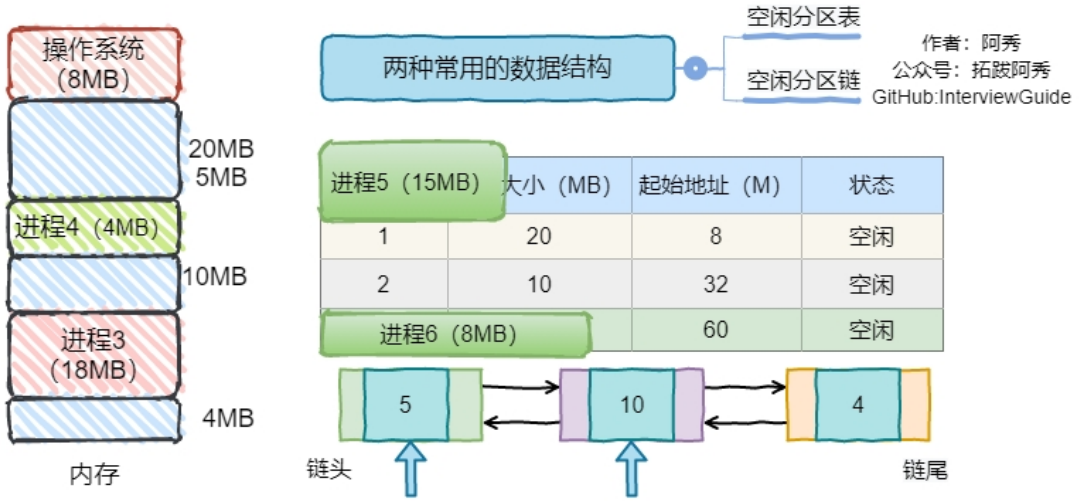
例:某系统使用基本分页存储管理，并采用了具有快表的地址变换机构。访问一次快表耗时1us， 访问一次内存耗时 100us。若快表的命中率为 90%，那么访问一个逻辑地址的平均耗时是多少? (1+100) * 0.9 + (1+100+100) * 0.1 = 111 us 有的系统支持快表和慢表同时查找，如果是这样，平均耗时应该是(1+100) * 0.9+ (100+100) *0.1=110.9 us 若未采用快表机制，则访问一个逻辑地址需要 100+100 = 200us 显然，引入快表机制后，访问一个逻辑地址的速度快多了。

11、动态分区分配算法有哪几种？可以分别说说吗？

1、首次适应算法

算法思想：每次都从低地址开始查找，找到第–个能满足大小的空闲分区。

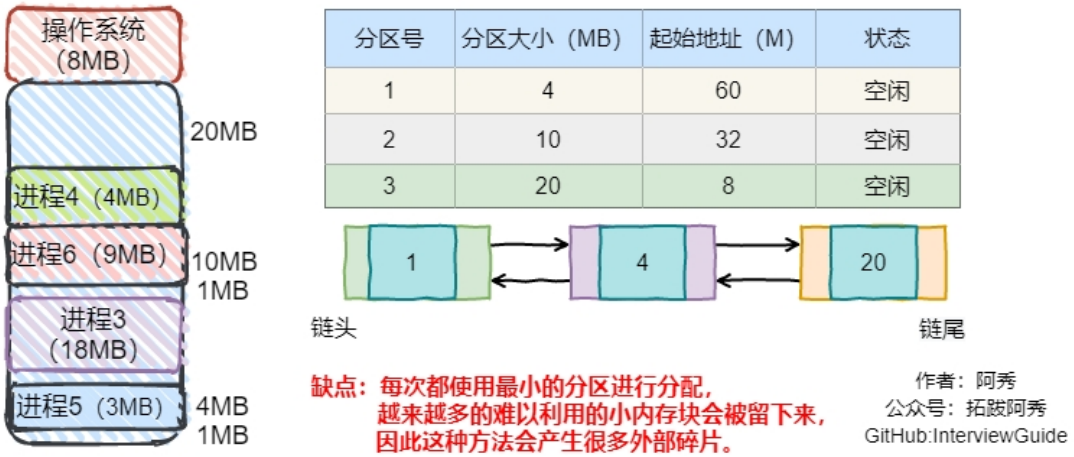
如何实现：空闲分区以地址递增的次序排列。每次分配内存时顺序查找空闲分区链(或空闲分[表)，找到大小能满足要求的第-一个空闲分区。



2、最佳适应算法

算法思想:由于动态分区分配是一种连续分配方式，为各进程分配的空间必须是连续的一整片区域。因此为了保证当"大进程"到来时能有连续的大片空间，可以尽可能多地留下大片的空闲区,即，优先使用更小的空闲区。

如何实现:空闲分区按容量递增次序链接。每次分配内存时顺序查找空闲分区链(或空闲分区表)，找到大小能满足要求的第-一个空闲分区。

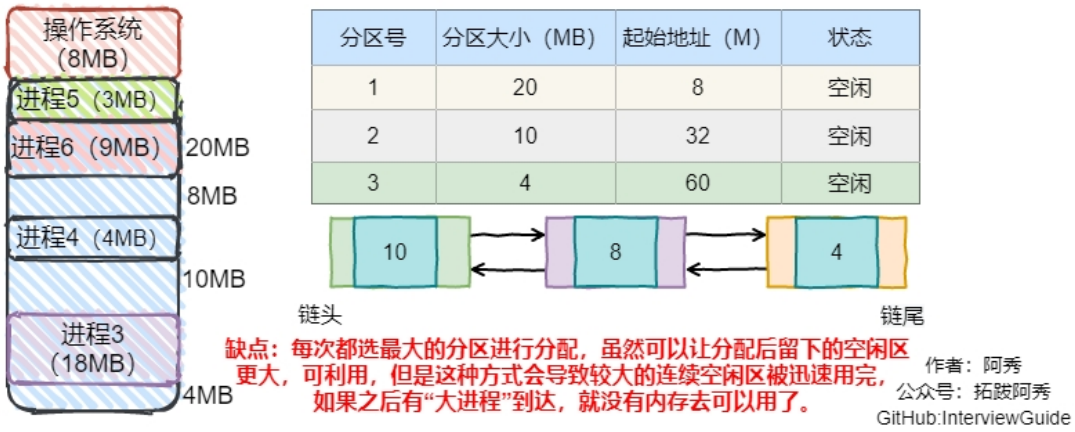


3、最坏适应算法

又称最大适应算法(Largest Fit)

算法思想:为了解决最佳适应算法的问题—即留下太多难以利用的小碎片，可以在每次分配时优先使用最大的连续空闲区，这样分配后剩余的空闲区就不会太小，更方便使用。

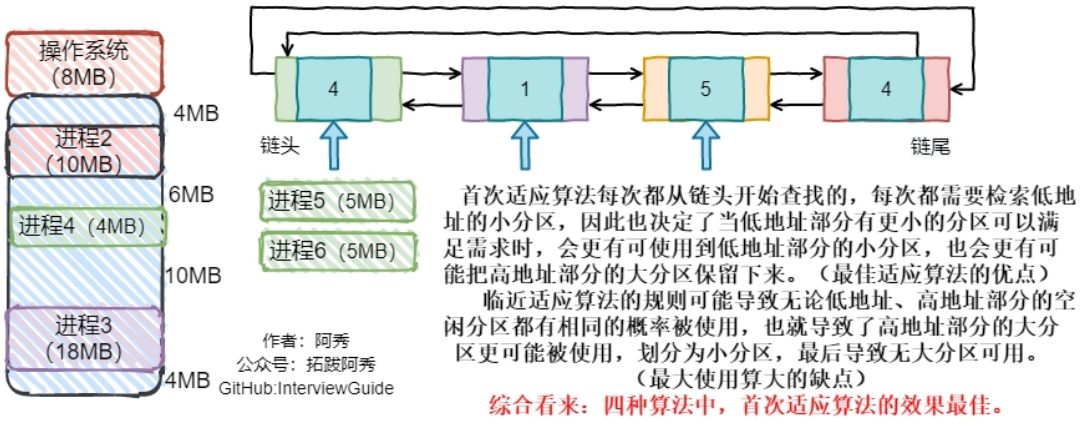
如何实现:空闲分区按容量递减次序链接。每次分配内存时顺序查找空闲分区链(或空闲分区表)，找到大小能满足要求的第-一个空闲分区。



4、邻近适应算法

算法思想：首次适应算法每次都从链头开始查找的。这可能会导致低地址部分出现很多小的空闲分区，而每次分配查找时，都要经过这些分区，因此也增加了查找的开销。如果每次都从上次查找结束的位置开始检索，就能解决上述问题。

如何实现：空闲分区以地址递增的顺序排列(可排成-一个循环链表)。每次分配内存时从上次查找结束的位置开始查找空闲分区链(或空闲分区表)，找到大小能满足要求的第一个空闲分区。



5、总结

首次适应不仅最简单，通常也是最好最快，不过首次适应算法会使得内存低地址部分出现很多小的空闲分区，而每次查找都要经过这些分区，因此也增加了查找的开销。

邻近算法试图解决这个问题，但实际上，它常常会导致在内存的末尾分配空间分裂成小的碎片，它通常比首次适应算法结果要差。

最佳导致大量碎片，最坏导致没有大的空间。

进过实验，首次适应比最佳适应要好，他们都比最坏好。

算法	算法思想	分区排列顺序	优点	缺点
首次适应	从头到尾找适合的分区	空闲分区以地址递增次序排列	综合看性能最好。**算法开销小**，回收分区后一.般不需要对空闲分区队列重新排序	
最佳适应	优先使用更小的分区，以保留更多大分区	空闲分区以容量递增次序排列	会有更多的大分区被保留下来，更能满足大进程需求	会产生很多太小的、难以利用的碎片;**算法开销大**，回收分区后可能需要对空闲分区队列重新排序
最坏适应	优先使用更大的分区，以防止产生太小的不可用的碎片	空闲分区以容量递减次序排列	可以减少难以利用的小碎片	大分区容易被用完，不利于大进程;**算法开销大**(原因同上)
邻近适应	由首次适应演变而来，每次从上次查找结束位置开始查找	空闲分区以地址递增次序排列(可排列成循环链表)	不用每次都从低地址的小分区开始检索。**算法开销小**(原因同首次适应算法)	会使高地址的大分区也被用完

12、虚拟技术你了解吗？

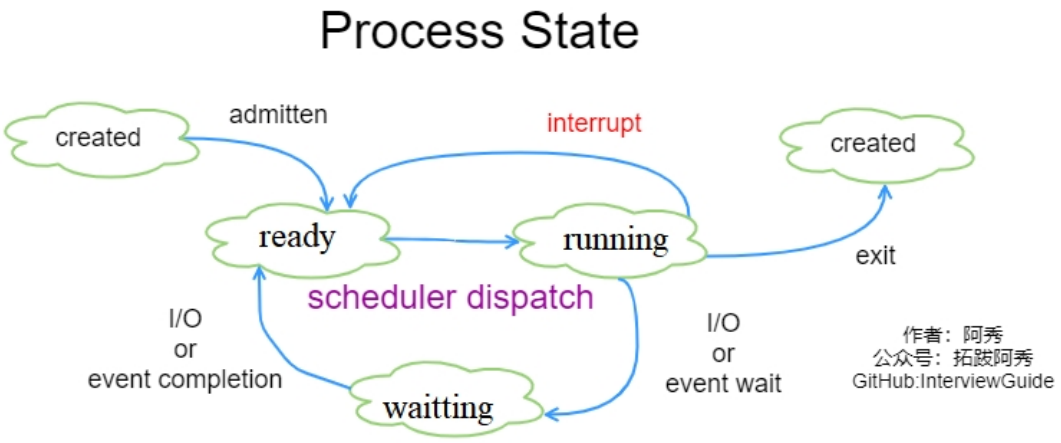
虚拟技术把一个物理实体转换为多个逻辑实体。

主要有两种虚拟技术：时（时间）分复用技术和空（空间）分复用技术。

多进程与多线程：多个进程能在同一个处理器上并发执行使用了时分复用技术，让每个进程轮流占用处理器，每次只执行一小个时间片并快速切换。

虚拟内存使用了空分复用技术，它将物理内存抽象为地址空间，每个进程都有各自的地址空间。地址空间的页被映射到物理内存，地址空间的页并不需要全部在物理内存中，当使用到一个没有在物理内存的页时，执行页面置换算法，将该页置换到内存中。

13、进程状态的切换你知道多少？



- 就绪状态（ready）：等待被调度
- 运行状态（running）
- 阻塞状态（waiting）：等待资源

应该注意以下内容：

- 只有就绪态和运行态可以相互转换，其它的都是单向转换。就绪状态的进程通过调度算法从而获得 CPU 时间，转为运行状态；而运行状态的进程，在分配给它的 CPU 时间片用完之后就会转为就绪状态，等待下一次调度。
- 阻塞状态是缺少需要的资源从而由运行状态转换而来，但是该资源不包括 CPU 时间，缺少 CPU 时间会从运行态转换为就绪态。

14、一个 C/C++程序从开始编译到生成可执行文件的完整过程，你能说出来多少？

四个过程：

（1）预编译 主要处理源代码文件中的以“#”开头的预编译指令。处理规则见下

1、删除所有的#define，展开所有的宏定义。

2、处理所有的条件预编译指令，如“#if”、“#endif”、“#ifdef”、“#elif”和“#else”。

3、处理“#include”预编译指令，将文件内容替换到它的位置，这个过程是递归进行的，文件中包含其他 文件。

4、删除所有的注释，“//”和“/**/”。

5、保留所有的#pragma 编译器指令，编译器需要用到他们，如：#pragma once 是为了防止有文件被重 复引用。

6、添加行号和文件标识，便于编译时编译器产生调试用的行号信息，和编译时产生编译错误或警告是 能够显示行号。

（2）编译 把预编译之后生成的 xxx.i 或 xxx.ii 文件，进行一系列词法分析、语法分析、语义分析及优化后，生成相应的汇编代码文件。

1、词法分析：利用类似于“有限状态机”的算法，将源代码程序输入到扫描机中，将其中的字符序列分割成一系列的记号。

2、语法分析：语法分析器对由扫描器产生的记号，进行语法分析，产生语法树。由语法分析器输出的语法树是一种以表达式为节点的树。

3、语义分析：语法分析器只是完成了对表达式语法层面的分析，语义分析器则对表达式是否有意义进行判断，其分析的语义是静态语义——在编译期能分期的语义，相对应的动态语义是在运行期才能确定的语义。

4、优化：源代码级别的一个优化过程。

5、目标代码生成：由代码生成器将中间代码转换成目标机器代码，生成一系列的代码序列——汇编语言表示。

6、目标代码优化：目标代码优化器对上述的目标机器代码进行优化：寻找合适的寻址方式、使用位移来替代乘法运算、删除多余的指令等。

（3）汇编

将汇编代码转变成机器可以执行的指令(机器码文件)。 汇编器的汇编过程相对于编译器来说更简单，没有复杂的语法，也没有语义，更不需要做指令优化，只是根据汇编指令和机器指令的对照表一一翻译过来，汇编过程有汇编器 as 完成。

经汇编之后，产生目标文件(与可执行文件格式几乎一样)xxx.o(Linux 下)、xxx.obj(Windows 下)。

（4）链接

将不同的源文件产生的目标文件进行链接，从而形成一个可以执行的程序。链接分为静态链接和动态链接：

1、静态链接

函数和数据被编译进一个二进制文件。在使用静态库的情况下，在编译链接可执行文件时，链接器从库中复制这些函数和数据并把它们和应用程序的其它模块组合起来创建最终的可执行文件。

空间浪费：因为每个可执行程序中对所有需要的目标文件都要有一份副本，所以如果多个程序对同一个目标文件都有依赖，会出现同一个目标文件都在内存存在多个副本；

更新困难：每当库函数的代码修改了，这个时候就需要重新进行编译链接形成可执行程序。

运行速度快：但是静态链接的优点就是，在可执行程序中已经具备了所有执行程序所需要的任何东西，在执行的时候运行速度快。

2、动态链接

动态链接的基本思想是把程序按照模块拆分成各个相对独立部分，在程序运行时才将它们链接在一起形成一个完整的程序，而不是像静态链接一样把所有程序模块都链接成一个单独的可执行文件。

共享库：就是即使需要每个程序都依赖同一个库，但是该库不会像静态链接那样在内存中存在多份副本，而是这多个程序在执行时共享同一份副本；

更新方便：更新时只需要替换原来的目标文件，而无需将所有的程序再重新链接一遍。当程序下一次运行时，新版本的目标文件会被自动加载到内存并且链接起来，程序就完成了升级的目标。

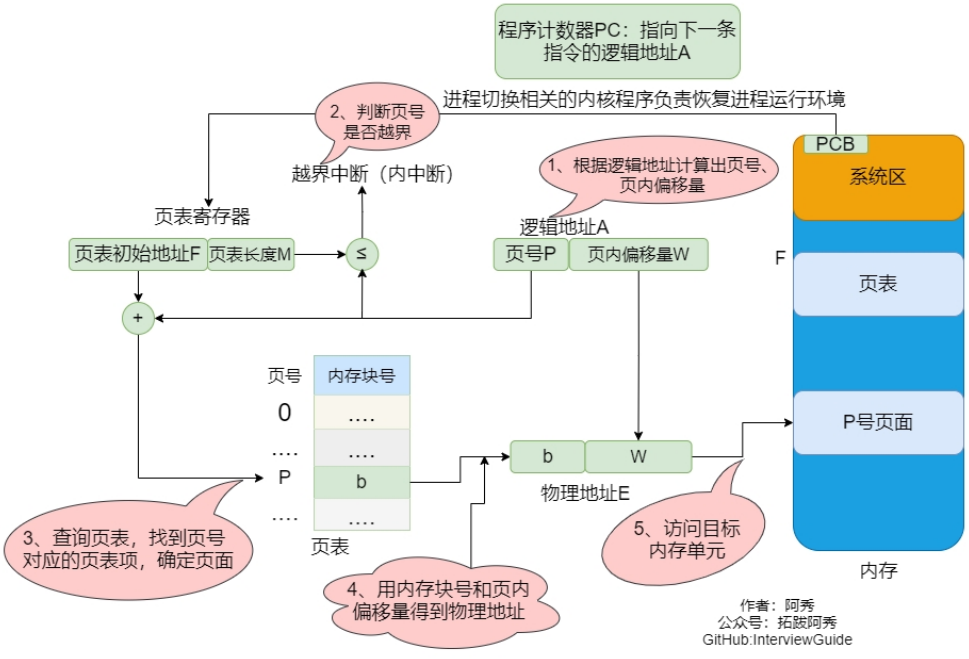
性能损耗：因为把链接推迟到了程序运行时，所以每次执行程序都需要进行链接，所以性能会有一定损失。

15、通过例子讲解逻辑地址转换为物理地址的基本过程

可以借助进程的页表将逻辑地址转换为物理地址。

通常会在系统中设置一个页表寄存器(PTR)，存放页表在内存中的起始地址 F 和页表长度 M。进程未执行时，页表的始址和页表长度放在进程控制块(PCB) 中，当进程被调度时，操作系统内核会把它们放到页表寄存器中。

注意:页面大小是 2 的整数幂 设页面大小为 L，逻辑地址 A 到物理地址 E 的变换过程如下:



例:若页面大小 L 为 1K 字节，页号 2 对应的内存块号 b=8，将逻辑地址 A=2500 转换为物理地址 E。 等价描述：某系统按字节寻址，逻辑地址结构中，页内偏移量占 10 位(说明一个页面的大小为 2^10B = 1KB)，页号 2 对应的内存块号 b=8，将逻辑地址 A=2500 转换为物理地址 E。

> ①计算页号、页内偏移量 页号 P=A/L = 2500/1024 = 2; 页内偏移量 W= A%L = 2500%1024 = 452
>
> ②根据题中条件可知，页号 2 没有越界，其存放的内存块号 b=8
>
> ③物理地址 E=b*L+W=8 * 1024+ 425 = 8644
>
> 在分页存储管理(页式管理)的系统中，只要确定了每个页面的大小，逻辑地址结构就确定了。因此，页式管理中地址是-维的。即，只要给出一个逻辑地址，系统就可以自动地算出页号、页内偏移量两个部分，并不需要显式地告诉系统这个逻辑地址中，页内偏移量占多少位。

16、进程同步的四种方法？

1. 临界区

对临界资源进行访问的那段代码称为临界区。

为了互斥访问临界资源，每个进程在进入临界区之前，需要先进行检查。

```
// entry section
// critical section;
// exit section
```

2. 同步与互斥

- 同步：多个进程因为合作产生的直接制约关系，使得进程有一定的先后执行关系。
- 互斥：多个进程在同一时刻只有一个进程能进入临界区。

3. 信号量

信号量（Semaphore）是一个整型变量，可以对其执行 down 和 up 操作，也就是常见的 P 和 V 操作。

- **down**：如果信号量大于 0 ，执行 -1 操作；如果信号量等于 0，进程睡眠，等待信号量大于 0；
- **up** ：对信号量执行 +1 操作，唤醒睡眠的进程让其完成 down 操作。

down 和 up 操作需要被设计成原语，不可分割，通常的做法是在执行这些操作的时候屏蔽中断。

如果信号量的取值只能为 0 或者 1，那么就成为了 **互斥量（Mutex）** ，0 表示临界区已经加锁，1 表示临界区解锁。

```
typedef int semaphore;
semaphore mutex = 1;void P1() {
    down(&mutex);
    // 临界区
    up(&mutex);
}
void P2() {
    down(&mutex);
    // 临界区
    up(&mutex);
}
```

使用信号量实现生产者-消费者问题

问题描述：使用一个缓冲区来保存物品，只有缓冲区没有满，生产者才可以放入物品；只有缓冲区不为空，消费者才可以拿走物品。

因为缓冲区属于临界资源，因此需要使用一个互斥量 mutex 来控制对缓冲区的互斥访问。

为了同步生产者和消费者的行为，需要记录缓冲区中物品的数量。数量可以使用信号量来进行统计，这里需要使用两个信号量：empty 记录空缓冲区的数量，full 记录满缓冲区的数量。

其中，empty 信号量是在生产者进程中使用，当 empty 不为 0 时，生产者才可以放入物品；full 信号量是在消费者进程中使用，当 full 信号量不为 0 时，消费者才可以取走物品。

注意，不能先对缓冲区进行加锁，再测试信号量。也就是说，不能先执行 down(mutex) 再执行 down(empty)。如果这么做了，那么可能会出现这种情况：生产者对缓冲区加锁后，执行 down(empty) 操作，发现 empty = 0，此时生产者睡眠。

消费者不能进入临界区，因为生产者对缓冲区加锁了，消费者就无法执行 up(empty) 操作，empty 永远都为 0，导致生产者永远等待下，不会释放锁，消费者因此也会永远等待下去。

```
#define N 100typedef int semaphore;
semaphore mutex = 1;
semaphore empty = N;
semaphore full = 0;
void producer() {
    while(TRUE) {
        int item = produce_item();
        down(&empty);
        down(&mutex);
        insert_item(item);
        up(&mutex);
        up(&full);
    }
}
void consumer() {
    while(TRUE) {
        down(&full);
        down(&mutex);
        int item = remove_item();
        consume_item(item);
        up(&mutex);
        up(&empty);
    }
}
```

4. 管程

使用信号量机制实现的生产者消费者问题需要客户端代码做很多控制，而管程把控制的代码独立出来，不仅不容易出错，也使得客户端代码调用更容易。

c 语言不支持管程，下面的示例代码使用了类 Pascal 语言来描述管程。示例代码的管程提供了 insert() 和 remove() 方法，客户端代码通过调用这两个方法来解决生产者-消费者问题。

```
monitor ProducerConsumer
    integer i;
    condition c;

    procedure insert();
    begin
        // ...
    end;

    procedure remove();
    begin
        // ...
    end;end monitor;
```

管程有一个重要特性：在一个时刻只能有一个进程使用管程。进程在无法继续执行的时候不能一直占用管程，否则其它进程永远不能使用管程。

管程引入了 **条件变量** 以及相关的操作：**wait()** 和 **signal()** 来实现同步操作。对条件变量执行 wait() 操作会导致调用进程阻塞，把管程让出来给另一个进程持有。signal() 操作用于唤醒被阻塞的进程。

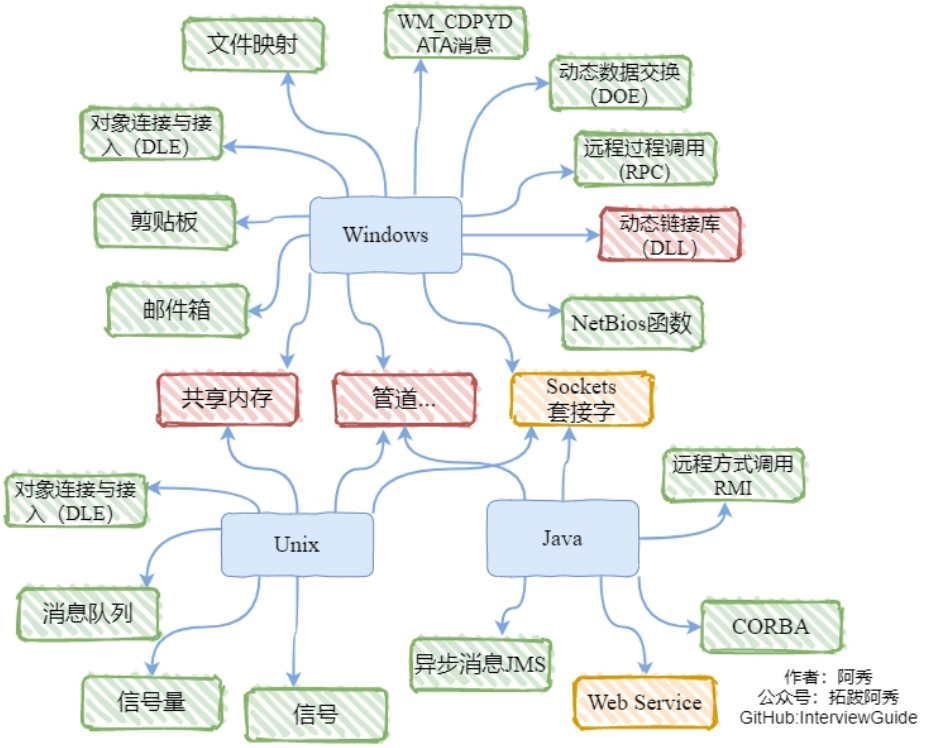
使用管程实现生产者-消费者问题

```
// 管程
monitor ProducerConsumer
    condition full, empty;
    integer count := 0;
    condition c;

    procedure insert(item: integer);
    begin
        if count = N then wait(full);
        insert_item(item);
        count := count + 1;
        if count = 1 then signal(empty);
    end;

    function remove: integer;
    begin
        if count = 0 then wait(empty);
        remove = remove_item;
        count := count - 1;
        if count = N -1 then signal(full);
    end;end monitor;
// 生产者客户端procedure producerbegin
    while true do
    begin
        item = produce_item;
        ProducerConsumer.insert(item);
    endend;
// 消费者客户端procedure consumerbegin
    while true do
    begin
        item = ProducerConsumer.remove;
        consume_item(item);
    endend;
```

17、操作系统在对内存进行管理的时候需要做些什么?

- 操作系统负责内存空间的分配与回收。
- 操作系统需要提供某种技术从逻辑上对内存空间进行扩充。
- 操作系统需要提供地址转换功能，负责程序的逻辑地址与物理地址的转换。
- 操作系统需要提供内存保护功能。保证各进程在各自存储空间内运行，互不干扰

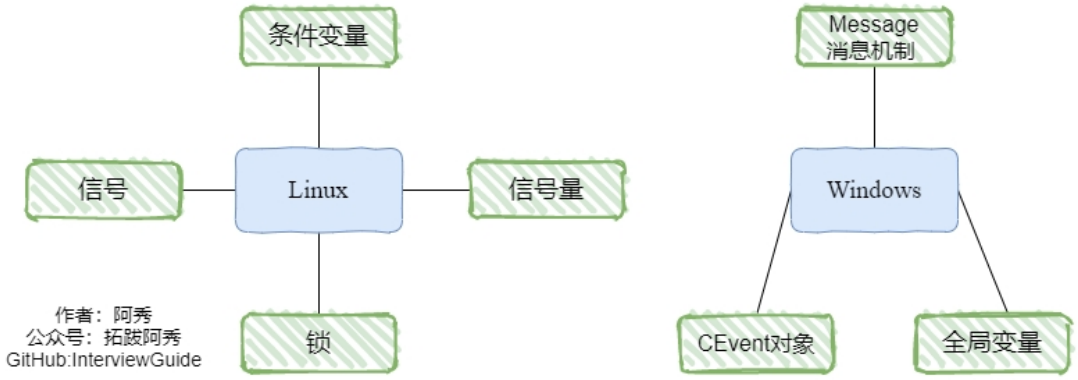
18、进程通信方法（Linux 和 windows 下），线程通信方法（Linux 和 windows 下）



名称及方式
管道(pipe)：允许一个进程和另一个与它有共同祖先的进程之间进行通信
命名管道(FIFO)：类似于管道，但是它可以用于任何两个进程之间的通信，命名管道在文件系统中有对应的文件名。命名管道通过命令 mkfifo 或系统调用 mkfifo 来创建
消息队列(MQ)：消息队列是消息的连接表，包括 POSIX 消息对和 System V 消息队列。有足够权限的进程可以向队列中添加消息，被赋予读权限的进程则可以读走队列中的消息。消息队列克服了信号承载信息量少，管道只能成该无格式字节流以及缓冲区大小受限等缺点；
信号量(semaphore)：信号量主要作为进程间以及同进程不同线程之间的同步手段；
共享内存(shared memory)：它使得多个进程可以访问同一块内存空间，**是最快的可用 IPC

形式。**这是针对其他通信机制运行效率较低而设计的。它往往与其他通信机制，如信号量结合使用，以达到进程间的同步及互斥
信号(signal)：信号是比较复杂的通信方式，用于通知接收进程有某种事情发生，除了用于进程间通信外，进程还可以发送信号给进程本身
内存映射(mapped memory)：内存映射允许任何多个进程间通信，每一个使用该机制的进程通过把一个共享的文件映射到自己的进程地址空间来实现它
Socket：它是更为通用的进程间通信机制，可用于不同机器之间的进程间通信

线程通信方法



名称及含义
Linux：
信号：类似进程间的信号处理
锁机制：互斥锁、读写锁和自旋锁
条件变量：使用通知的方式解锁，与互斥锁配合使用
信号量：包括无名线程信号量和命名线程信号量
Windows：
全局变量：需要有多个线程来访问一个全局变量时，通常我们会在这个全局变量前加上 volatile 声明，以防编译器对此变量进行优化
Message 消息机制：常用的 Message 通信的接口主要有两个：PostMessage 和 PostThreadMessage，PostMessage 为线程向主窗口发送消息。而 PostThreadMessage 是任意两个线程之间的通信接口。
CEvent 对象：CEvent 为 MFC 中的一个对象，可以通过对 CEvent 的触发状态进行改变，从而实现线程间的通信和同步，这个主要是实现线程直接同步的一种方法。

19、进程间通信有哪几种方式？把你知道的都说出来

Linux 几乎支持全部 UNIX 进程间通信方法，包括管道（有名管道和无名管道）、消息队列、共享内存、信号量和套接字。其中前四个属于同一台机器下进程间的通信，套接字则是用于网络通信。

管道

无名管道

无名管道特点：

无名管道是一种特殊的文件，这种文件只存在于内存中。

无名管道只能用于父子进程或兄弟进程之间，必须用于具有亲缘关系的进程间的通信。

无名管道只能由一端向另一端发送数据，是半双工方式，如果双方需要同时收发数据需要两个管道。

相关接口：

int pipe(int fd[2]);

- fd[2]：管道两端用 fd[0]和 fd[1]来描述，读的一端用 fd[0]表示，写的一端用 fd[1]表示。通信双方的进程中写数据的一方需要把 fd[0]先 close 掉，读的一方需要先把 fd[1]给 close 掉。

有名管道：

有名管道特点：

有名管道是 FIFO 文件，存在于文件系统中，可以通过文件路径名来指出。

有名管道可以在不具有亲缘关系的进程间进行通信。

相关接口：

int mkfifo(const char *pathname, mode_t mode);

pathname：即将创建的 FIFO 文件路径，如果文件存在需要先删除。

mode：和 open()中的参数相同。

消息队列

相比于 FIFO，消息队列具有以下优点：

- 消息队列可以独立于读写进程存在，从而避免了 FIFO 中同步管道的打开和关闭时可能产生的困难；
- 避免了 FIFO 的同步阻塞问题，不需要进程自己提供同步方法；
- 读进程可以根据消息类型有选择地接收消息，而不像 FIFO 那样只能默认地接收。

共享内存

进程可以将同一段共享内存连接到它们自己的地址空间，所有进程都可以访问共享内存中的地址，如果某个进程向共享内存内写入数据，所做的改动将立即影响到可以访问该共享内存的其他所有进程。

相关接口

创建共享内存：int shmget(key_t key, int size, int flag);

成功时返回一个和 key 相关的共享内存标识符，失败范湖范围-1。

key：为共享内存段命名，多个共享同一片内存的进程使用同一个 key。

size：共享内存容量。

flag：权限标志位，和 open 的 mode 参数一样。

连接到共享内存地址空间：void *shmat(int shmid, void *addr, int flag);

返回值即共享内存实际地址。

shmid：shmget()返回的标识。

addr：决定以什么方式连接地址。

flag：访问模式。

从共享内存分离：int shmdt(const void *shmaddr);

调用成功返回 0，失败返回-1。

- shmaddr：是 shmat()返回的地址指针。

其他补充

共享内存的方式像极了多线程中线程对全局变量的访问，大家都对等地有权去修改这块内存的值，这就导致在多进程并发下，最终结果是不可预期的。所以对这块临界区的访问需要通过信号量来进行进程同步。

但共享内存的优势也很明显，首先可以通过共享内存进行通信的进程不需要像无名管道一样需要通信的进程间有亲缘关系。其次内存共享的速度也比较快，不存在读取文件、消息传递等过程，只需要到相应映射到的内存地址直接读写数据即可。

信号量

在提到共享内存方式时也提到，进程共享内存和多线程共享全局变量非常相似。所以在使用内存共享的方式是也需要通过信号量来完成进程间同步。多线程同步的信号量是 POSIX 信号量，而在进程里使用 SYSTEM V 信号量。

相关接口

创建信号量：int semget(key_t key, int nsems, int semflag);

创建成功返回信号量标识符，失败返回-1。

key：进程 pid。

nsems：创建信号量的个数。

semflag：指定信号量读写权限。

改变信号量值：int semop(int semid, struct sembuf *sops, unsigned nsops);

我们所需要做的主要工作就是串讲 sembuf 变量并设置其值，然后调用 semop，把设置好的 sembuf 变量传递进去。

struct sembuf 结构体定义如下：

```
struct sembuf{
    short sem_num;
    short sem_op;
    short sem_flg;
};
```

成功返回信号量标识符，失败返回-1。

semid：信号量集标识符，由 semget()函数返回。

sops：指向 struct sembuf 结构的指针，先设置好 sembuf 值再通过指针传递。

nsops：进行操作信号量的个数，即 sops 结构变量的个数，需大于或等于 1。最常见设置此值等于 1，只完成对一个信号量的操作。

直接控制信号量信息：int semctl(int semid, int semnum, int cmd, union semun arg);

semid：信号量集标识符。

semnum：信号量集数组上的下标，表示某一个信号量。

arg：union semun 类型。

辅助命令

ipcs 命令用于报告共享内存、信号量和消息队列信息。

ipcs -a：列出共享内存、信号量和消息队列信息。

ipcs -l：列出系统限额。

ipcs -u：列出当前使用情况。

套接字

与其它通信机制不同的是，它可用于不同机器间的进程通信。

20、虚拟内存的目的是什么？

虚拟内存的目的是为了让物理内存扩充成更大的逻辑内存，从而让程序获得更多的可用内存。

为了更好的管理内存，操作系统将内存抽象成地址空间。每个程序拥有自己的地址空间，这个地址空间被分割成多个块，每一块称为一页。

这些页被映射到物理内存，但不需要映射到连续的物理内存，也不需要所有页都必须在物理内存中。当程序引用到不在物理内存中的页时，由硬件执行必要的映射，将缺失的部分装入物理内存并重新执行失败的指令。

从上面的描述中可以看出，虚拟内存允许程序不用将地址空间中的每一页都映射到物理内存，也就是说一个程序不需要全部调入内存就可以运行，这使得有限的内存运行大程序成为可能。

例如有一台计算机可以产生 16 位地址，那么一个程序的地址空间范围是 0~64K。该计算机只有 32KB 的物理内存，虚拟内存技术允许该计算机运行一个 64K 大小的程序。

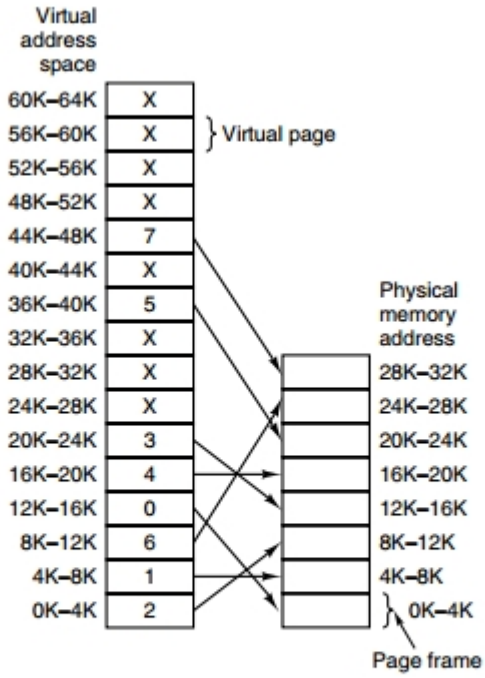
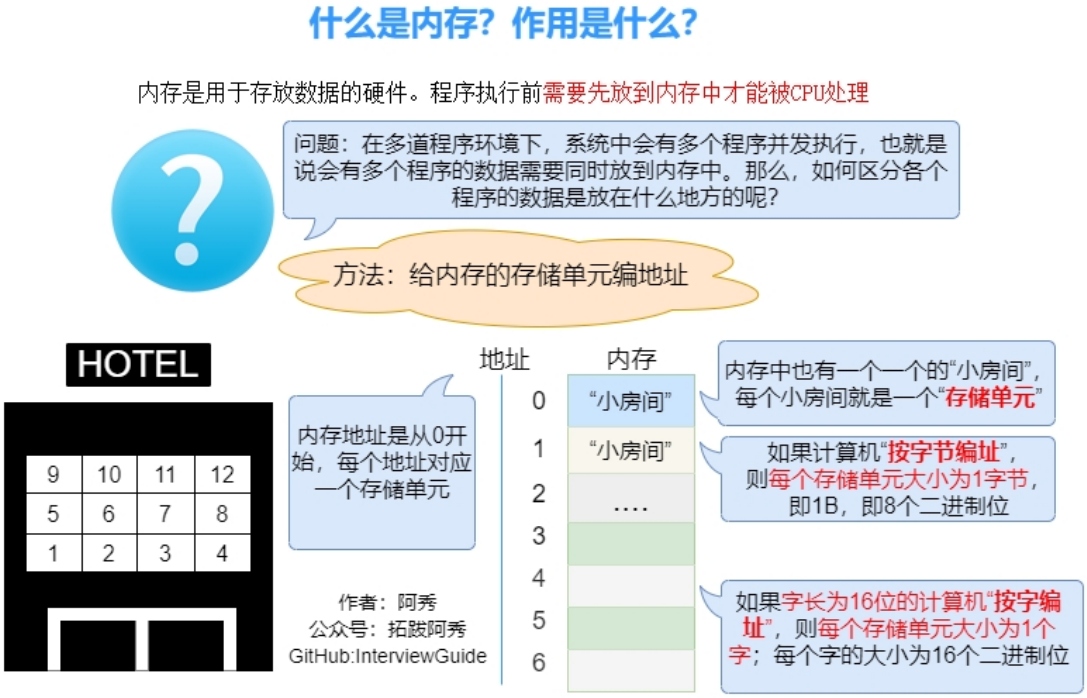
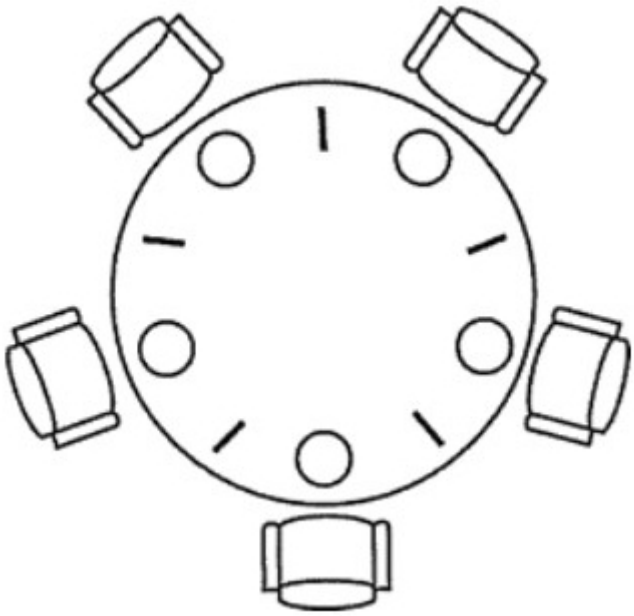


Figure 3-9. The relation between virtual addresses and physical memory addresses is given by the **page table**. Every page begins on a multiple of 4096 and ends 4095 addresses higher, so 4K–8K really means 4096–8191 and 8K to 12K means 8192–12287.

21、说一下你理解中的内存？他有什么作用呢？



22、操作系统经典问题之哲学家进餐问题



五个哲学家围着一张圆桌，每个哲学家面前放着食物。哲学家的生活有两种交替活动：吃饭以及思考。当一个哲学家吃饭时，需要先拿起自己左右两边的两根筷子，并且一次只能拿起一根筷子。

下面是一种错误的解法，如果所有哲学家同时拿起左手边的筷子，那么所有哲学家都在等待其它哲学家吃完并释放自己手中的筷子，导致死锁。

```
#define N 5
void philosopher(int i) {
    while(TRUE) {
        think();
        take(i);       // 拿起左边的筷子
        take((i+1)%N); // 拿起右边的筷子
        eat();
        put(i);
        put((i+1)%N);
    }
}
```

为了防止死锁的发生，可以设置两个条件：

- 必须同时拿起左右两根筷子；
- 只有在两个邻居都没有进餐的情况下才允许进餐。

```
#define N 5#define LEFT (i + N - 1) % N // 左邻居#define RIGHT (i + 1) % N
// 右邻居#define THINKING 0#define HUNGRY   1#define EATING   2typedef int
semaphore;int state[N];                 // 跟踪每个哲学家的状态
semaphore mutex = 1;          // 临界区的互斥，临界区是 state 数组，对其修改需
要互斥
semaphore s[N];               // 每个哲学家一个信号量
void philosopher(int i) {
    while(TRUE) {
        think(i);
        take_two(i);
        eat(i);
        put_two(i);
    }
}
void take_two(int i) {
    down(&mutex);
    state[i] = HUNGRY;
    check(i);
    up(&mutex);
    down(&s[i]); // 只有收到通知之后才可以开始吃，否则会一直等下去
}
void put_two(i) {
    down(&mutex);
    state[i] = THINKING;
    check(LEFT); // 尝试通知左右邻居，自己吃完了，你们可以开始吃了
    check(RIGHT);
    up(&mutex);
}
void eat(int i) {
    down(&mutex);
    state[i] = EATING;
    up(&mutex);
}
// 检查两个邻居是否都没有用餐，如果是的话，就 up(&s[i])，使得 down(&s[i]) 能
够得到通知并继续执行void check(i) {
    if(state[i] == HUNGRY && state[LEFT] != EATING && state[RIGHT] !=EATING)
{
        state[i] = EATING;
        up(&s[i]);
    }
}
```

23、操作系统经典问题之读者-写者问题

允许多个进程同时对数据进行读操作，但是不允许读和写以及写和写操作同时发生。

一个整型变量 count 记录在对数据进行读操作的进程数量，一个互斥量 count_mutex 用于对 count 加锁，一个互斥量 data_mutex 用于对读写的数据加锁。

```
typedef int semaphore;
semaphore count_mutex = 1;
semaphore data_mutex = 1;int count = 0;
void reader() {
    while(TRUE) {
        down(&count_mutex);
        count++;
        if(count == 1) down(&data_mutex); // 第一个读者需要对数据进行加锁，防止写进程访问
        up(&count_mutex);
        read();
        down(&count_mutex);
        count--;
        if(count == 0) up(&data_mutex);//最后一个读者要对数据进行解锁，防止写进程无法访问
        up(&count_mutex);
    }
}
void writer() {
    while(TRUE) {
        down(&data_mutex);
        write();
        up(&data_mutex);
    }
}
```

24、介绍一下几种典型的锁？

读写锁

- 多个读者可以同时进行读
- 写者必须互斥（只允许一个写者写，也不能读者写者同时进行）
- 写者优先于读者（一旦有写者，则后续读者必须等待，唤醒时优先考虑写者）

互斥锁

一次只能一个线程拥有互斥锁，其他线程只有等待

互斥锁是在抢锁失败的情况下主动放弃 CPU 进入睡眠状态直到锁的状态改变时再唤醒，而操作系统负责线程调度，为了实现锁的状态发生改变时唤醒阻塞的线程或者进程，需要把锁交给操作系统管理，所以互斥锁在加锁操作时涉及上下文的切换。

互斥锁实际的效率还是可以让人接受的，加锁的时间大概 100ns 左右，而实际上互斥锁的一种可能的实现是先自旋一段时间，当自旋的时间超过阀值之后再将线程投入睡眠中，因此在并发运算中使用互斥锁（每次占用锁的时间很短）的效果可能不亚于使用自旋锁

条件变量

互斥锁一个明显的缺点是他只有两种状态：锁定和非锁定。而条件变量通过允许线程阻塞和等待另一个线程发送信号的方法弥补了互斥锁的不足，他常和互斥锁一起使用，以免出现竞态条件。

当条件不满足时，线程往往解开相应的互斥锁并阻塞线程然后等待条件发生变化。一旦其他的某个线程改变了条件变量，他将通知相应的条件变量唤醒一个或多个正被此条件变量阻塞的线程。

总的来说**互斥锁是线程间互斥的机制，条件变量则是同步机制。**

自旋锁

如果进线程无法取得锁，进线程不会立刻放弃 CPU 时间片，而是一直循环尝试获取锁，直到获取为止。

如果别的线程长时期占有锁，那么自旋就是在浪费 CPU 做无用功，但是自旋锁一般应用于加锁时间很短的场景，这个时候效率比较高。

24.1、你知道哪几种线程锁（POSIX）？

互斥锁（mutex）

互斥锁属于 sleep-waiting 类型的锁。例如在一个双核的机器上有两个线程 A 和 B，它们分别运行在 core 0 和 core 1 上。假设线程 A 想要通过 pthread_mutex_lock 操作去得到一个临界区的锁，而此时这个锁正被线程 B 所持有，那么线程 A 就会被阻塞，此时会通过上下文切换将线程 A 置于等待队列中，此时 core 0 就可以运行其他的任务（如线程 C）。

条件变量(cond)

自旋锁(spin)

自旋锁属于 busy-waiting 类型的锁，如果线程 A 是使用 pthread_spin_lock 操作去请求锁，如果自旋锁已经被线程 B 所持有，那么线程 A 就会一直在 core 0 上进行忙等待并不停的进行锁请求，检查该自旋锁是否已经被线程 B 释放，直到得到这个锁为止。因为自旋锁不会引起调用者睡眠，所以自旋锁的效率远高于互斥锁。

虽然它的效率比互斥锁高，但是它也有些不足之处：

> 自旋锁一直占用 CPU，在未获得锁的情况下，一直进行自旋，所以占用着 CPU，如果不能在很短的时间内获得锁，无疑会使 CPU 效率降低。
>
> 在用自旋锁时有可能造成死锁，当递归调用时有可能造成死锁。
>
> 自旋锁只有在内核可抢占式或 SMP 的情况下才真正需要，在单 CPU 且不可抢占式的内核下，自旋锁的操作为空操作。自旋锁适用于锁使用者保持锁时间比较短的情况下。

25、逻辑地址 VS 物理地址

编译时只需确定变量 x 存放的相对地址是 100 (也就是说相对于进程在内存中的起始地址而言的地址)。

CPU 想要找到 x 在内存中的实际存放位置，只需要用进程的起始地址+100 即可。

相对地址又称逻辑地址，绝对地址又称物理地址。

26、怎么回收线程？有哪几种方法？

等待线程结束： int pthread_join(pthread_t tid, void** retval);

主线程调用，等待子线程退出并回收其资源，类似于进程中 wait/waitpid 回收僵尸进程，调用 pthread_join 的线程会被阻塞。

tid：创建线程时通过指针得到 tid 值。

retval：指向返回值的指针。

结束线程： void pthread_exit(void *retval);

子线程执行，用来结束当前线程并通过 retval 传递返回值，该返回值可通过 pthread_join 获得。

- retval：同上。

分离线程： int pthread_detach(pthread_t tid);

主线程、子线程均可调用。主线程中 pthread_detach(tid)，子线程中 pthread_detach(pthread_self())，调用后和主线程分离，子线程结束时自己立即回收资源。

- tid：同上。

28、内存交换是什么？有什么特点？

交换(对换)技术的设计思想：内存空间紧张时，系统将内存中某些进程暂时换出外存，把外存中某些已具备运行条件的进程换入内存(进程在内存与磁盘间动态调度)

换入：把准备好竞争 CPU 运行的程序从辅存移到内存。

换出：把处于等待状态（或 CPU 调度原则下被剥夺运行权力）的程序从内存移到辅存，把内存空间腾出来。

29、什么时候会进行内存的交换？

内存交换通常在许多进程运行且内存吃紧时进行，而系统负荷降低就暂停。

例如:在发现许多进程运行时经常发生缺页，就说明内存紧张，此时可以换出一些进程;如果缺页率明显下降，就可以暂停换出。

30、终端退出，终端运行的进程会怎样

终端在退出时会发送 SIGHUP 给对应的 bash 进程，bash 进程收到这个信号后首先将它发给 session 下面的进程，

如果程序没有对 SIGHUP 信号做特殊处理，那么进程就会随着终端关闭而退出.

31、如何让进程后台运行

（1）命令后面加上&即可，实际上，这样是将命令放入到一个作业队列中了

（2）ctrl + z 挂起进程，使用 jobs 查看序号，在使用 bg %序号后台运行进程

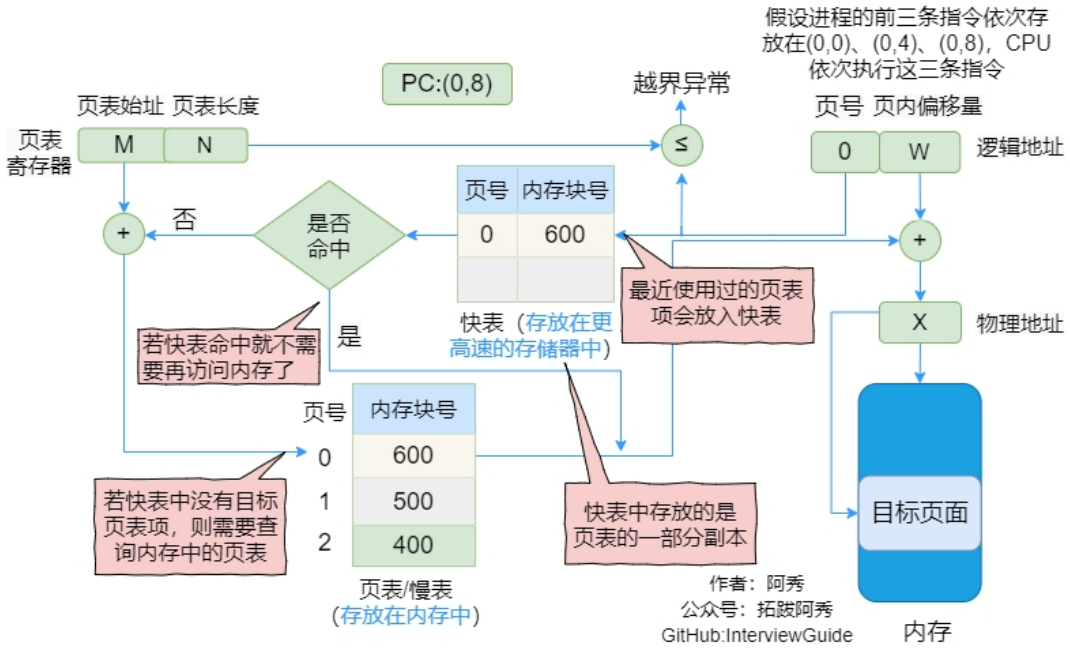
（3）nohup + &，将标准输出和标准错误缺省会被重定向到 nohup.out 文件中，忽略所有挂断（SIGHUP）信号

（4）运行指令前面 + setsid，使其父进程编程 init 进程，不受 HUP 信号的影响

（5）将 命令+ &放在()括号中，也可以是进程不受 HUP 信号的影响

32、什么是快表，你知道多少关于快表的知识？

快表，又称联想寄存器(TLB) ，是一种访问速度比内存快很多的高速缓冲存储器，用来存放当前访问的若干页表项，以加速地址变换的过程。与此对应，内存中的页表常称为慢表。



33、地址变换中，有快表和没快表，有什么区别？

	地址变换过程	访问一个逻辑地址的访存次数
基本地址变换机构	①算页号、页内偏移量 ②检查页号合法性 ③查页表，找到页面存放的内存块号 ④根据内存块号与页内偏移量得到物理地址 ⑤访问目标内存单元	两次访存
具有快表的地址变换机构	①算页号、页内偏移量 ②检查页号合法性 ③查快表。若命中，即可知道页面存放的内存块号，可直接进行⑤;若未命中则进行④ ④查页表，找到页面存放的内存块号，并且将页表项复制到快表中 ⑤根据内存块号与页内偏移量得到物理地址 ⑥访问目标内存单元	快表命中，只需一次访存 快表未命中，需要两次访存

34、在执行 malloc 申请内存的时候，操作系统是怎么做的？

从操作系统层面上看，malloc 是通过两个系统调用来实现的： brk 和 mmap

- brk 是将进程数据段(.data)的最高地址指针向高处移动，这一步可以扩大进程在运行时的堆大小
- mmap 是在进程的虚拟地址空间中寻找一块空闲的虚拟内存，这一步可以获得一块可以操作的堆内存。

通常，分配的内存小于 128k 时，使用 brk 调用来获得虚拟内存，大于 128k 时就使用 mmap 来获得虚拟内存。

进程先通过这两个系统调用获取或者扩大进程的虚拟内存，获得相应的虚拟地址，在访问这些虚拟地址的时候，通过缺页中断，让内核分配相应的物理内存，这样内存分配才算完成。

35、 守护进程、僵尸进程和孤儿进程

守护进程

指在后台运行的，没有控制终端与之相连的进程。它独立于控制终端，周期性地执行某种任务。Linux 的大多数服务器就是用守护进程的方式实现的，如 web 服务器进程 http 等

创建守护进程要点：

（1）让程序在后台执行。方法是调用 fork（）产生一个子进程，然后使父进程退出。

（2）调用 setsid（）创建一个新对话期。控制终端、登录会话和进程组通常是从父进程继承下来的，守护进程要摆脱它们，不受它们的影响，方法是调用 setsid（）使进程成为一个会话组长。setsid（）调用成功后，进程成为新的会话组长和进程组长，并与原来的登录会话、进程组和控制终端脱离。

（3）禁止进程重新打开控制终端。经过以上步骤，进程已经成为一个无终端的会话组长，但是它可以重新申请打开一个终端。为了避免这种情况发生，可以通过使进程不再是会话组长来实现。再一次通过 fork（）创建新的子进程，使调用 fork 的进程退出。

（4）关闭不再需要的文件描述符。子进程从父进程继承打开的文件描述符。如不关闭，将会浪费系统资源，造成进程所在的文件系统无法卸下以及引起无法预料的错误。首先获得最高文件描述符值，然后用一个循环程序，关闭 0 到最高文件描述符值的所有文件描述符。

（5）将当前目录更改为根目录。

（6）子进程从父进程继承的文件创建屏蔽字可能会拒绝某些许可权。为防止这一点，使用 unmask（0）将屏蔽字清零。

（7）处理 SIGCHLD 信号。对于服务器进程，在请求到来时往往生成子进程处理请求。如果子进程等待父进程捕获状态，则子进程将成为僵尸进程（zombie），从而占用系统资源。如果父进程等待子进程结束，将增加父进程的负担，影响服务器进程的并发性能。在 Linux 下可以简单地将 SIGCHLD 信号的操作设为 SIG_IGN。这样，子进程结束时不会产生僵尸进程。

孤儿进程

如果父进程先退出，子进程还没退出，那么子进程的父进程将变为 init 进程。（注：任何一个进程都必须有父进程）。

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。

僵尸进程

如果子进程先退出，父进程还没退出，那么子进程必须等到父进程捕获到了子进程的退出状态才真正结束，否则这个时候子进程就成为僵尸进程。

设置**僵尸进程的目**的是维护子进程的信息，以便父进程在以后某个时候获取。这些信息至少包括进程 ID，进程的终止状态，以及该进程使用的 CPU 时间，所以当终止子进程的父进程调用 wait 或 waitpid 时就可以得到这些信息。如果一个进程终止，而该进程有子进程处于僵尸状态，那么它的所有僵尸子进程的父进程 ID 将被重置为 1（init 进程）。继承这些子进程的 init 进程将清理它们（也就是说 init 进程将 wait 它们，从而去除它们的僵尸状态）。

36、如何避免僵尸进程？

通过 signal(SIGCHLD, SIG_IGN)通知内核对子进程的结束不关心，由内核回收。如果不想让父进程挂起，可以在父进程中加入一条语句：signal(SIGCHLD,SIG_IGN);表示父进程忽略 SIGCHLD 信号，该信号是子进程退出的时候向父进程发送的。

父进程调用 wait/waitpid 等函数等待子进程结束，如果尚无子进程退出 wait 会导致父进程阻塞。waitpid 可以通过传递 WNOHANG 使父进程不阻塞立即返回。

如果父进程很忙可以用 signal 注册信号处理函数，在信号处理函数调用 wait/waitpid 等待子进程退出。

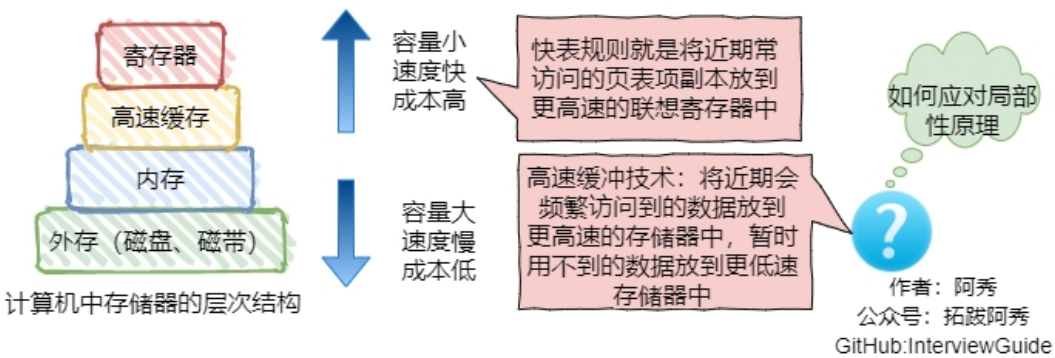
通过两次调用 fork。父进程首先调用 fork 创建一个子进程然后 waitpid 等待子进程退出，子进程再 fork 一个孙进程后退出。这样子进程退出后会被父进程等待回收，而对于孙子进程其父进程已经退出所以孙进程成为一个孤儿进程，孤儿进程由 init 进程接管，孙进程结束后，init 会等待回收。

第一种方法忽略 SIGCHLD 信号，这常用于并发服务器的性能的一个技巧因为并发服务器常常 fork 很多子进程，子进程终结之后需要服务器进程去 wait 清理资源。如果将此信号的处理方式设为忽略，可让内核把僵尸子进程转交给 init 进程去处理，省去了大量僵尸进程占用系统资源。

37、局部性原理你知道吗？主要有哪两大局部性原理？各自是什么？

主要分为**时间局部性和空间局部性**。

时间局部性:如果执行了程序中的某条指令，那么不久后这条指令很有可能再次执行;如果某个数据被访问过，不久之后该数据很可能再次被访问。(因为程序中存在大量的循环) 空间局部性:一旦程序访问了某个存储单元，在不久之后，其附近的存储单元也很有可能被访问。(因为很多数据在内存中都是连续存放的，并且程序的指令也是顺序地在内存中存放的)



38、父进程、子进程、进程组、作业和会话

父进程

已创建一个或多个子进程的进程

子进程

由 fork 创建的新进程被称为子进程（child process）。该函数被调用一次，但返回两次。两次返回的区别是子进程的返回值是 0，而父进程的返回值则是新进程（子进程）的进程 id。

将子进程 id 返回给父进程的理由是：因为一个进程的子进程可以多于一个，没有一个函数使一个进程可以获得其所有子进程的进程 id。

对子进程来说，之所以 fork 返回 0 给它，是因为它随时可以调用 getpid()来获取自己的 pid；也可以调用 getppid()来获取父进程的 id。(进程 id 0 总是由交换进程使用，所以一个子进程的进程 id 不可能为 0)。

fork 之后，操作系统会复制一个与父进程完全相同的子进程，虽说是父子关系，但是在操作系统看来，他们更像兄弟关系，这 2 个进程共享代码空间，但是数据空间是互相独立的，子进程数据空间中的内容是父进程的完整拷贝，指令指针也完全相同，子进程拥有父进程当前运行到的位置（两进程的程序计数器 pc 值相同，也就是说，子进程是从 fork 返回处开始执行的）。

但有一点不同，如果 fork 成功，子进程中 fork 的返回值是 0，父进程中 fork 的返回值是子进程的进程号，如果 fork 不成功，父进程会返回错误。

子进程从父进程继承的有：

1.进程的资格(真实(real)/有效(effective)/已保存(saved)用户号(UIDs)和组号(GIDs))

2.环境(environment)

3.堆栈

4.内存

5.进程组号

独有：

1.进程号；

2.不同的父进程号(译者注：即子进程的父进程号与父进程的父进程号不同， 父进程号可由 getppid 函数得到)；

3.资源使用(resource utilizations)设定为 0

进程组

进程组就是多个进程的集合，其中肯定有一个组长，其进程 PID 等于进程组的 PGID。只要在某个进程组中一个进程存在，该进程组就存在，这与其组长进程是否终止无关。

作业

shell 分前后台来控制的不是进程而是作业（job）或者进程组（Process Group）。

一个前台作业可以由多个进程组成，一个后台也可以由多个进程组成，shell 可以运行一个前台作业和任意多个后台作业，这称为作业控制

为什么只能运行一个前台作业？

当我们在前台新起了一个作业，shell 就被提到了后台，因此 shell 就没有办法再继续接受我们的指令并且解析运行了。

但是如果前台进程退出了，shell 就会有被提到前台来，就可以继续接受我们的命令并且解析运行。

作业与进程组的区别：如果作业中的某个进程有创建了子进程，则该子进程是不属于该作业的。 一旦作业运行结束，shell 就把自己提到前台（子进程还存在，但是子进程不属于作业），如果原来的前台进程还存在（这个子进程还没有终止），他将自动变为后台进程组

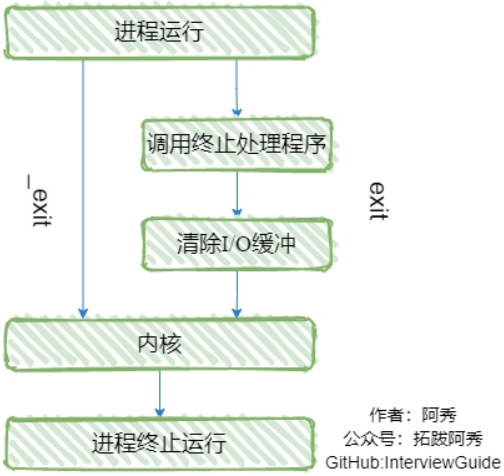
会话

会话（Session）是一个或多个进程组的集合。一个会话可以有一个控制终端。在 xshell 或者 WinSCP 中打开一个窗口就是新建一个会话。

39、进程终止的几种方式

1、main 函数的自然返回，return 2、调用 exit 函数，属于 c 的函数库 3、调用_exit 函数，属于系统调用 4、调用 abort 函数，异常程序终止，同时发送 SIGABRT 信号给调用进程。 5、接受能导致进程终止的信号：ctrl+c (^C)、SIGINT(SIGINT 中断进程)

exit 和_exit 的区别



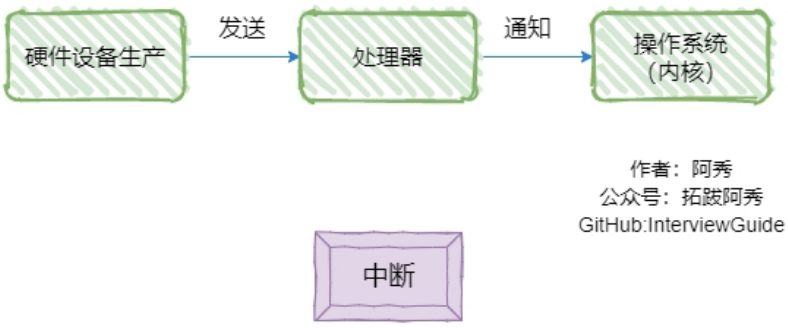
40、Linux 中异常和中断的区别

中断

大家都知道，当我们在敲击键盘的同时就会产生中断，当硬盘读写完数据之后也会产生中断，所以，我们需要知道，中断大多数是由硬件设备产生的，而它们从物理上说就是电信号。

之后，它们通过中断控制器发送给 CPU，接着 CPU 判断收到的中断来自于哪个硬件设备（这定义在内核中）。

最后，由 CPU 发送给内核，有内核处理中断。下面这张图显示了中断处理的流程：



软件中断与软中断 除了硬件中断，还存在部分由软件产生的中断，称之为软件中断（Software Interrupt），最常见的有引发系统调用的 Int 0x80。

同时，软件中断又区别于软中断（SoftIRQ），软中断主要用于中断处理的下半程（Bottom Halves），非关键逻辑的部分，来提高中断处理的效率与实时性，最常用于 I/O 相关的中断处理。

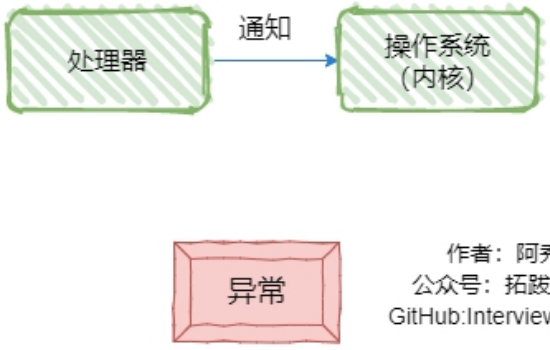
处理中断下半程的方法除了软中断，还有 tasklet，二者存在一定的区别与联系：

- 二者都可以被注册用于中断处理下半程的延时任务；
- 同一种类型的 tasklet 只能串行执行，而同类型的软中断可以多个 CPU 并发执行，不同类型的软中断和 tasklet 均可并发执行；
- tasklet 底层是基于两种软中断来实现的，分别是 HI_SOFTIRQ 和 TASKLET_SOFTIRQ；

异常

我们在学习《计算机组成原理》的时候会知道两个概念，CPU 处理程序的时候一旦程序不在内存中，会产生缺页异常；当运行除法程序时，当除数为 0 时，又会产生除 0 异常。

所以，大家也需要记住的是，**异常是由 CPU 产生的，同时，它会发送给内核，要求内核处理这些异常**，下面这张图显示了异常处理的流程：



相同点

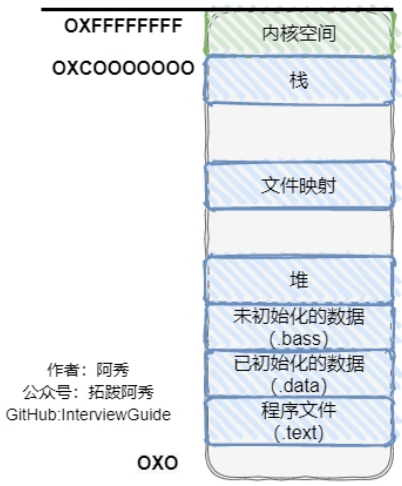
最后都是由 CPU 发送给内核，由内核去处理

处理程序的流程设计上是相似的

不同点

- 产生源不相同，异常是由 CPU 产生的，而中断主要是由硬件设备产生的
- 内核需要根据是异常还是中断调用不同的处理程序
- 中断不是时钟同步的，这意味着中断可能随时到来；异常由于是 CPU 产生的，所以它是时钟同步的
- 当处理中断时，处于中断上下文中；处理异常时，处于进程上下文中

41、Windows 和 Linux 环境下内存分布情况



通过这张图你可以看到，用户空间内存，从**低到高**分别是 7 种不同的内存段：

- 程序文件段，包括二进制可执行代码；
- 已初始化数据段，包括静态常量；
- 未初始化数据段，包括未初始化的静态变量；
- 堆段，包括动态分配的内存，从低地址开始向上增长；
- 文件映射段，包括动态库、共享内存等，从低地址开始向上增长（跟硬件和内核版本有关）
- 栈段，包括局部变量和函数调用的上下文等。栈的大小是固定的，一般是 8 MB。当然系统也提供了参数，以便我们自定义大小；

42、一个由 C/C++编译的程序占用的内存分为哪几个部分？

1、栈区（stack）— 地址向下增长，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数据结构中的栈，先进后出。

2、堆区（heap）— 地址向上增长，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。 - 程序结束后有系统释放

4、文字常量区 —常量字符串就是放在这里的。程序结束后由系统释放

5、程序代码区(text)—存放函数体的二进制代码。

43、一般情况下在 Linux/windows 平台下栈空间的大小

Linux 环境下有操作系统决定，一般是 8MB，8192KB，通过 ulimit 命令查看以及修改

Windows 环境下由编译器决定，VC++6.0 一般是 1M

Linux

linux 下非编译器决定栈大小，而是由操作系统环境决定，默认是 8192KB（8M）；而在 Windows 平台下栈的大小是被记录在可执行文件中的（由编译器来设置)，即：windows 下可以由编译器决定栈大小，而在 Linux 下是由系统环境变量来控制栈的大小的。

在 Linux 下通过如下命令可查看和设置栈的大小：

```
$ ulimit -a           # 显示当前栈的大小 （ulimit 为系统命令，非编译器命令）
$ ulimit -s 32768     # 设置当前栈的大小为 32M
```

Windows

下程序栈空间的大小，VC++ 6.0 默认的栈空间是 1M。

VC6.0 中修改堆栈大小的方法：

- 选择 "Project->Setting"
- 选择 "Link"
- 选择 "Category"中的 "Output"
- 在 "Stack allocations"中的"Reserve:"中输栈的大小

44、程序从堆中动态分配内存时，虚拟内存上怎么操作的

页表：是一个存放在物理内存中的数据结构，它记录了虚拟页与物理页的映射关系

在进行动态内存分配时，例如 malloc()函数或者其他高级语言中的 new 关键字，操作系统会在硬盘中创建或申请一段虚拟内存空间，并更新到页表（分配一个页表条目（PTE），使该 PTE 指向硬盘上这个新创建的虚拟页），通过 PTE 建立虚拟页和物理页的映射关系。

45、常见的几种磁盘调度算法

读写一个磁盘块的时间的影响因素有：

- 旋转时间（主轴转动盘面，使得磁头移动到适当的扇区上）
- 寻道时间（制动手臂移动，使得磁头移动到适当的磁道上）
- 实际的数据传输时间

其中，寻道时间最长，因此磁盘调度的主要目标是使磁盘的平均寻道时间最短。

1. 先来先服务

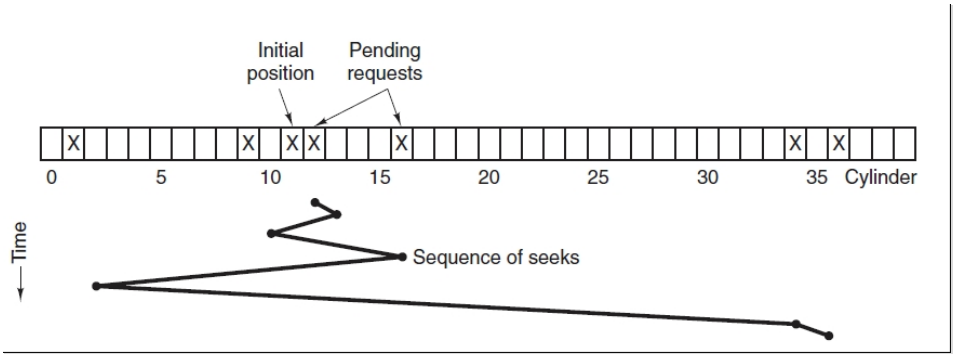
按照磁盘请求的顺序进行调度。

优点是公平和简单。缺点也很明显，因为未对寻道做任何优化，使平均寻道时间可能较长。

2. 最短寻道时间优先

优先调度与当前磁头所在磁道距离最近的磁道。

虽然平均寻道时间比较低，但是不够公平。如果新到达的磁道请求总是比一个在等待的磁道请求近，那么在等待的磁道请求会一直等待下去，也就是出现饥饿现象。具体来说，两端的磁道请求更容易出现饥饿现象。

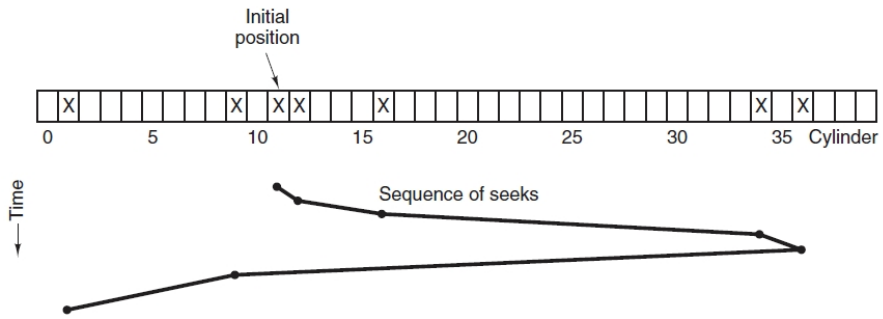


3. 电梯扫描算法

电梯总是保持一个方向运行，直到该方向没有请求为止，然后改变运行方向。

电梯算法（扫描算法）和电梯的运行过程类似，总是按一个方向来进行磁盘调度，直到该方向上没有未完成的磁盘请求，然后改变方向。

因为考虑了移动方向，因此所有的磁盘请求都会被满足，解决了 SSTF 的饥饿问题。



46、交换空间与虚拟内存的关系

交换空间

Linux 中的交换空间（Swap space）在**物理内存**（RAM）被充满时被使用。如果系统需要更多的内存资源，而物理内存已经充满，内存中不活跃的页就会被移到交换空间去。

虽然交换空间可以为带有少量内存的机器提供帮助，但是这种方法不应该被当做是对内存的取代。交换空间位于硬盘驱动器上，它比进入物理内存要慢。

交换空间可以是一个专用的交换分区（推荐的方法），交换文件，或两者的组合。

交换空间的总大小应该相当于你的计算机内存的两倍和 32 MB 这两个值中较大的一个，但是它不能超过 2048MB（2 GB）。

虚拟内存

虚拟内存是文件数据交叉链接的活动文件。是 WINDOWS 目录下的一个"WIN386.SWP"文件，这个文件会不断地扩大和自动缩小。

就速度方面而言,CPU 的 L1 和 L2 缓存速度最快，内存次之，硬盘再次之。但是**虚拟内存使用的是硬盘的空间**，为什么我们要使用速度最慢的硬盘来做 为虚拟内存呢？

因为电脑中所有运行的程序都需要经过内存来执行，如果执行的程序很大或很多，就会导致我们只有可怜的 256M/512M 内存消耗殆尽，而硬盘空间动辄几十 G 上百 G。

为了解决这个问题，Windows 中运用了虚拟内存技术，即拿出一部分硬盘空间来充当内存使用。

47、抖动你知道是什么吗？它也叫颠簸现象

刚刚换出的页面马上又要换入内存，刚刚换入的页面马上又要换出外存，这种频繁的页面调度行为称为抖动，或颠簸。

产生抖动的主要原因是进程频繁访问的页面数目高于可用的物理块数(分配给进程的物理块不够)

为进程分配的物理块太少，会使进程发生抖动现象。为进程分配的物理块太多，又会降低系统整体的并发度，降低某些资源的利用率。

为了研究为应该为每个进程分配多少个物理块，Denning 提出了进程工作集”的概念

48、从堆和栈上建立对象哪个快？（考察堆和栈的分配效率比较）

从两方面来考虑：

1、分配和释放，堆在分配和释放时都要调用函数（malloc,free)，比如分配时会到堆空间去寻找足够大小的空间（因为多次分配释放后会造成内存碎片），这些都会花费一定的时间。具体可以看看 malloc 和 free 的源代码，函数做了很多额外的工作，而栈却不需要这些。

2、访问时间，访问堆的一个具体单元，需要两次访问内存，第一次得取得指针，第二次才是真正的数据，而栈只需访问一次。另外，堆的内容被操作系统交换到外存的概率比栈大，栈一般是不会被交换出去的。

49、常见内存分配方式有哪些？

内存分配方式

（1） 从静态存储区域分配。内存在程序编译的时候就已经分配好，这块内存在程序的整个运行期间都存在。例如全局变量，static 变量。

（2） 在栈上创建。在执行函数时，函数内局部变量的存储单元都可以在栈上创建，函数执行结束时这些存储单元自动被释放。栈内存分配运算内置于处理器的指令集中，效率很高，但是分配的内存容量有限。

（3） 从堆上分配，亦称动态内存分配。程序在运行的时候用 malloc 或 new 申请任意多少的内存，程序员自己负责在何时用 free 或 delete 释放内存。动态内存的生存期由我们决定，使用非常灵活，但问题也最多。

50、常见内存分配内存错误

（1）内存分配未成功，却使用了它。

编程新手常犯这种错误，因为他们没有意识到内存分配会不成功。常用解决办法是，在使用内存之前检查指针是否为 NULL。如果指针 p 是函数的参数，那么在函数的入口处用 assert(p!=NULL)进行检查。如果是用 malloc 或 new 来申请内存，应该用 if(p==NULL) 或 if(p!=NULL)进行防错处理。

（2）内存分配虽然成功，但是尚未初始化就引用它。

犯这种错误主要有两个起因：一是没有初始化的观念；二是误以为内存的缺省初值全为零，导致引用初值错误（例如数组）。内存的缺省初值究竟是什么并没有统一的标准，尽管有些时候为零值，我们宁可信其无不可信其有。所以无论用何种方式创建数组，都别忘了赋初值，即便是赋零值也不可省略，不要嫌麻烦。

（3）内存分配成功并且已经初始化，但操作越过了内存的边界。

例如在使用数组时经常发生下标“多 1”或者“少 1”的操作。特别是在 for 循环语句中，循环次数很容易搞错，导致数组操作越界。

（4）忘记了释放内存，造成内存泄露。

含有这种错误的函数每被调用一次就丢失一块内存。刚开始时系统的内存充足，你看不到错误。终有一次程序突然挂掉，系统出现提示：内存耗尽。动态内存的申请与释放必须配对，程序中 malloc 与 free 的使用次数一定要相同，否则肯定有错误（new/delete 同理）。

（5）释放了内存却继续使用它。常见于以下有三种情况：

> 程序中的对象调用关系过于复杂，实在难以搞清楚某个对象究竟是否已经释放了内存，此时应该重新设计数据结构，从根本上解决对象管理的混乱局面。
>
> 函数的 return 语句写错了，注意不要返回指向“栈内存”的“指针”或者“引用”，因为该内存在函数体结束时被自动销毁。
>
> 使用 free 或 delete 释放了内存后，没有将指针设置为 NULL。导致产生“空悬指针”。

51、内存交换中，被换出的进程保存在哪里？

保存在磁盘中，也就是外存中。具有对换功能的操作系统中，通常把磁盘空间分为文件区和对换区两部分。

文件区主要用于存放文件，主要追求存储空间的利用率，因此对文件区空间的管理采用离散分配方式;对换区空间只占磁盘空间的小部分，被换出的进程数据就存放在对换区。

由于对换的速度直接影响到系统的整体速度，因此对换区空间的管理主要追求换入换出速度，因此通常对换区采用连续分配方式(学过文件管理章节后即可理解)。

总之，对换区的 I/O 速度比文件区的更快。

52、在发生内存交换时，有些进程是被优先考虑的？你可以说一说吗？

可优先换出阻塞进程;可换出优先级低的进程;为了防止优先级低的进程在被调入内存后很快又被换出，有的系统还会考虑进程在内存的驻留时间... (注意: PCB 会常驻内存，不会被换出外存)

53、ASCII、Unicode 和 UTF-8 编码的区别？

ASCII

ASCII 只有 127 个字符，表示英文字母的大小写、数字和一些符号，但由于其他语言用 ASCII 编码表示字节不够。

例如：常用中文需要两个字节，且不能和 ASCII 冲突，中国定制了 GB2312 编码格式，相同的，其他国家的语言也有属于自己的编码格式。

Unicode

由于每个国家的语言都有属于自己的编码格式，在多语言编辑文本中会出现乱码，这样 Unicode 应运而生，Unicode 就是将这些语言统一到一套编码格式中。

通常两个字节表示一个字符，而 ASCII 是一个字节表示一个字符，这样如果你编译的文本是全英文的，用 Unicode 编码比 ASCII 编码需要多一倍的存储空间，在存储和传输上就十分不划算。

UTF-8

为了解决上述问题，又出现了把 Unicode 编码转化为"**可变长编码**"UTF-8 编码，UTF-8 编码将 Unicode 字符按数字大小编码为 1-6 个字节，英文字母被编码成一个字节，常用汉字被编码成三个字节。

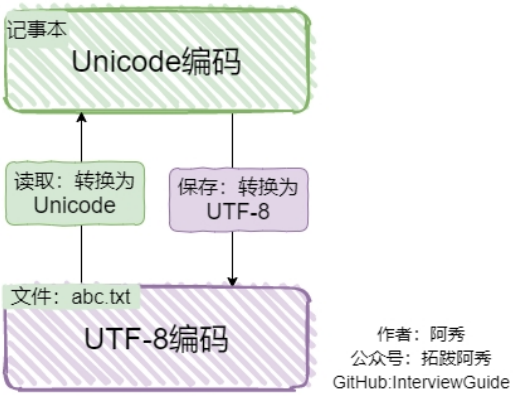
如果你编译的文本是纯英文的，那么用 UTF-8 就会非常节省空间，并且 ASCII 码也是 UTF-8 的一部分。

三者之间的联系

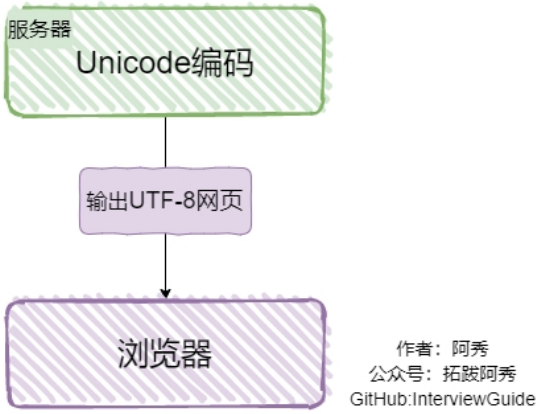
搞清楚了 ASCII、Unicode 和 UTF-8 的关系，我们就可以总结一下现在计算机系统通用的字符编码工作方式：

(1) 在计算机内存中，统一使用 Unicode 编码，当需要保存到硬盘或者需要传输的时候，就转换为 UTF-8 编码

(2)用记事本编辑的时候，从文件读取的 UTF-8 字符被转换为 Unicode 字符到内存里，编辑完成后，保存的时候再把 Unicode 转换为 UTF-8 保存到文件。如下图（截取他人图片）



浏览网页的时候，服务器会把动态生成的 Unicode 内容转换为 UTF-8 再传输到浏览器：



54、原子操作的是如何实现的

处理器使用基于对缓存加锁或总线加锁的方式来实现多处理器之间的原子操作。

首先处理器会自动保证基本的内存操作的原子性。处理器保证从系统内存中读取或者写入一个字节是原子的，意思是当一个处理器读取一个字节时，其他处理器不能访问这个字节的内存地址。

Pentium 6 和最新的处理器能自动保证单处理器对同一个缓存行里进行 16/32/64 位的操作是原子的，但是复杂的内存操作处理器是不能自动保证其原子性的，比如跨总线宽度、跨多个缓存行和跨页表的访问。但是，处理器提供总线锁定和缓存锁定两个机制来保证复杂内存操作的原子性。

（1）使用总线锁保证原子性

第一个机制是通过总线锁保证原子性。

如果多个处理器同时对共享变量进行读改写操作（i++就是经典的读改写操作），那么共享变量就会被多个处理器同时进行操作，这样读改写操作就不是原子的，操作完之后共享变量的值会和期望的不一致。

举个例子，如果 i=1，我们进行两次 i++操作，我们期望的结果是 3，但是有可能结果是 2，如图下图所示。

```
CPU1    CPU2
 i=1     i=1
 i+1     i+1
 i=2     i=2
```

原因可能是多个处理器同时从各自的缓存中读取变量 i，分别进行加 1 操作，然后分别写入系统内存中。

那么，想要保证读改写共享变量的操作是原子的，就必须保证 CPU1 读改写共享变量的时候，CPU2 不能操作缓存了该共享变量内存地址的缓存。

处理器使用总线锁就是来解决这个问题的。**所谓总线锁就是使用处理器提供的一个 LOCK＃信号，当一个处理器在总线上输出此信号时，其他处理器的请求将被阻塞住，那么该处理器可以独占共享内存。**

（2）使用缓存锁保证原子性

第二个机制是通过缓存锁定来保证原子性。在同一时刻，我们只需保证对某个内存地址的操作是原子性即可，但**总线锁定把 CPU 和内存之间的通信锁住了**。

这使得锁定期间，其他处理器不能操作其他内存地址的数据，所以总线锁定的开销比较大。目前处理器在某些场合下使用缓存锁定代替总线锁定来进行优化。

频繁使用的内存会缓存在处理器的 L1、L2 和 L3 高速缓存里，那么原子操作就可以直接在处理器内部缓存中进行，并不需要声明总线锁，在 Pentium 6 和目前的处理器中可以使用“缓存锁定”的方式来实现复杂的原子性。

所谓“缓存锁定”是指内存区域如果被缓存在处理器的缓存行中，并且在 Lock 操作期间被锁定，那么当它执行锁操作回写到内存时，处理器不在总线上声言 LOCK＃信号，而是修改内部的内存地址，并允许它的缓存一致性机制来保证操作的原子性。

因为**缓存一致性机制会阻止同时修改由两个以上处理器缓存的内存区域数据，当其他处理器回写已被锁定的缓存行的数据时，会使缓存行无效，在如上图所示的例子中，当 CPU1 修改缓存行中的 i 时使用了缓存锁定，那么 CPU2 就不能使用同时缓存 i 的缓存行。**

但是有两种情况下处理器不会使用缓存锁定。

- 第一种情况是：当操作的数据不能被缓存在处理器内部，或操作的数据跨多个缓存行（cache line）时，则处理器会调用总线锁定。
- 第二种情况是：有些处理器不支持缓存锁定。对于 Intel 486 和 Pentium 处理器，就算锁定的内存区域在处理器的缓存行中也会调用总线锁定。

55、内存交换你知道有哪些需要注意的关键点吗？

1. 交换需要备份存储，通常是快速磁盘，它必须足够大，并且提供对这些内存映像的直接访问。
2. 为了有效使用 CPU，需要每个进程的执行时间比交换时间长，而影响交换时间的主要是转移时间，转移时间与所交换的空间内存成正比。
3. 如果换出进程，比如确保该进程的内存空间成正比。
4. 交换空间通常作为磁盘的一整块，且独立于文件系统，因此使用就可能很快。
5. 交换通常在有许多进程运行且内存空间吃紧时开始启动，而系统负荷降低就暂停。
6. 普通交换使用不多，但交换的策略的某些变种在许多系统中（如 UNIX 系统）仍然发挥作用。

56、系统并发和并行，分得清吗？

并发是指宏观上在一段时间内能同时运行多个程序，而并行则指同一时刻能运行多个指令。

并行需要硬件支持，如多流水线、多核处理器或者分布式计算系统。

操作系统通过引入进程和线程，使得程序能够并发运行。

57、可能是最全的页面置换算法总结了

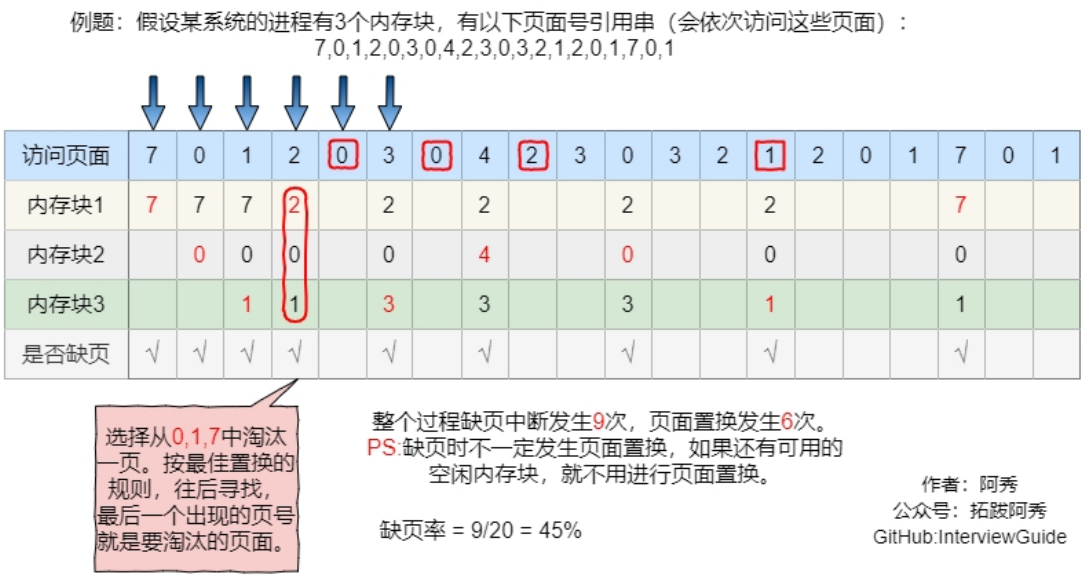
1、最佳置换法(OPT)

最佳置换算法(OPT，Optimal) :每次选择淘汰的页面将是以后永不使用，或者在最长时间内不再被访问的页面，这样可以保证最低的缺页率。

57、可能是最全的页面置换算法总结了

1、最佳置换法(OPT)

最佳置换算法(OPT，Optimal) :每次选择淘汰的页面将是以后永不使用，或者在最长时间内不再被访问的页面，这样可以保证最低的缺页率。



最佳置换算法可以保证最低的缺页率，但实际上，只有在进程执行的过程中才能知道接下来会访问到的是哪个页面。操作系统无法提前预判页面访问序列。因此，最佳置换算法是无法实现的

2、先进先出置换算法(FIFO)

先进先出置换算法(FIFO) :每次选择淘汰的页面是最早进入内存的页面 实现方法:把调入内存的页面根据调入的先后顺序排成一个队列，需要换出页面时选择队头页面队列的最大长度取决于系统为进程分配了多少个内存块。

访问页面	3	2	1	0	3	2	4	3	2	1	0	4
内存块1	3	3	3	3			4	4	4	4	0	0
内存块2		2	2	2			2	3	3	3	3	4
内存块3			1	1			1	1	2	2	2	2
内存块4				0			0	0	0	1	1	1
是否缺页	√	√	√	√			√	√	√	√	√	√

Belady 异常—当为进程分配的物理块数增大时，缺页次数不减反增的异常现象。只有 FIFO 算法会产生 Belady 异常，而 LRU 和 OPT 算法永远不会出现 Belady 异常。

另外，FIFO 算法虽然实现简单，但是该算法与进程实际运行时的规律不适应，因为先进入的页面也有可能最经常被访问。因此，算法性能差。

FIFO 的性能较差，因为较早调入的页往往是经常被访问的页，这些页在 FIFO 算法下被反复调入和调出，并且有 Belady 现象。

所谓 Belady 现象是指：采用 FIFO 算法时，如果对—个进程未分配它所要求的全部页面，有时就会出现分配的页面数增多但缺页率反而提高的异常现象。

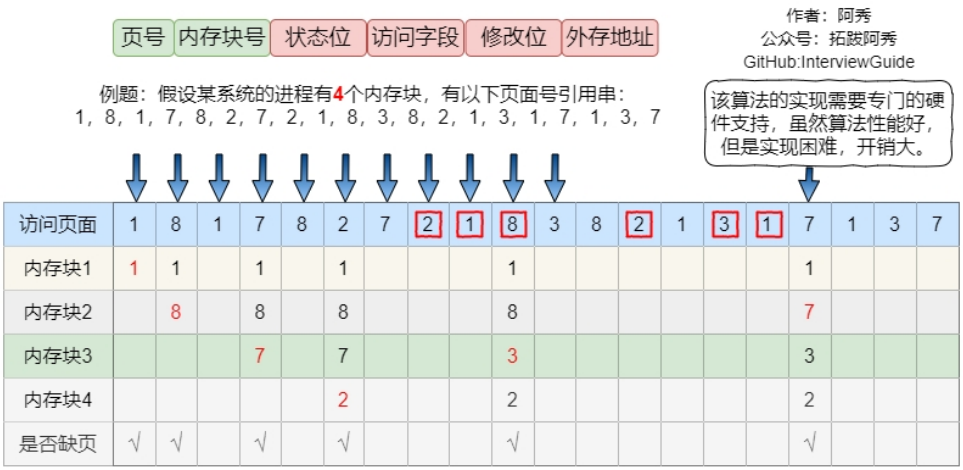
3、最近最久未使用置换算法(LRU)

最近最久未使用置换算法(LRU，least recently used) :每次淘汰的页面是最近最久未使用的页面。

实现方法:赋予每个页面对应的页表项中，用访问字段记录该页面自上次被访问以来所经历的时间 t(该算法的实现需要专门的硬件支持，虽然算法性能好，但是实现困难，开销大)。

当需要淘汰一个页面时，选择现有页面中 t 值最大的，即最近最久未使用的页面。

LRU 性能较好，但需要寄存器和栈的硬件支持。LRU 是堆栈类算法，理论上可以证明，堆栈类算法不可能出现 Belady 异常。



访问页面	1	8	1	7	8	2	7	2	1	8	3	8	2	1	3	1	7	1	3	7
内存块1	1	1		1		1				1							1			
内存块2		8		8		8				8							7			
内存块3				7		7				3							3			
内存块4						2				2							2			
是否缺页	√	√		√		√				√							√			

58、共享是什么？

共享是指系统中的资源可以被多个并发进程共同使用。

有两种共享方式：互斥共享和同时共享。

互斥共享的资源称为临界资源，例如打印机等，在同一时刻只允许一个进程访问，需要用同步机制来实现互斥访问。

59、死锁相关问题大总结，超全！

死锁是指两个（多个）线程相互等待对方数据的过程，死锁的产生会导致程序卡死，不解锁程序将永远无法进行下去。

1、死锁产生原因

举个例子：两个线程 A 和 B，两个数据 1 和 2。线程 A 在执行过程中，首先对资源 1 加锁，然后再去给资源 2 加锁，但是由于线程的切换，导致线程 A 没能给资源 2 加锁。

线程切换到 B 后，线程 B 先对资源 2 加锁，然后再去给资源 1 加锁，由于资源 1 已经被线程 A 加锁，因此线程 B 无法加锁成功，当线程切换为 A 时，A 也无法成功对资源 2 加锁，由此就造成了线程 AB 双方相互对一个已加锁资源的等待，死锁产生。

理论上认为死锁产生有以下四个必要条件，缺一不可：

> **互斥条件**：进程对所需求的资源具有排他性，若有其他进程请求该资源，请求进程只能等待。
>
> **不剥夺条件**：进程在所获得的资源未释放前，不能被其他进程强行夺走，只能自己释放。
>
> **请求和保持条件**：进程当前所拥有的资源在进程请求其他新资源时，由该进程继续占有。
>
> **循环等待条件**：存在一种进程资源循环等待链，链中每个进程已获得的资源同时被链中下一个进程所请求。

2、死锁演示

通过代码的形式进行演示，需要两个线程和两个互斥量。

```
#include <iostream>
#include <vector>
#include <list>
#include <thread>
#include <mutex>  //引入互斥量头文件
using namespace std;
class A {
public:
        //插入消息，模拟消息不断产生
        void insertMsg() {
                for (int i = 0; i < 100; i++) {
                        cout << "插入一条消息:" << i << endl;
                        my_mutex1.lock(); //语句1
                        my_mutex2.lock(); //语句2
                        Msg.push_back(i);
                        my_mutex2.unlock();
                        my_mutex1.unlock();
                }
        }
        //读取消息
        void readMsg() {
                int MsgCom;
                for (int i = 0; i < 100; i++) {
                        MsgCom = MsgLULProc(i);
                        if (MsgLULProc(MsgCom)) {
                                //读出消息了
                                cout << "消息已读出" << MsgCom << endl;
                        }
                        else {
                                //消息暂时为空
                                cout << "消息为空" << endl;
                        }
                }
        }
        //加解锁代码
        bool MsgLULProc(int &command) {
                int curMsg;
                my_mutex2.lock();   //语句3
                my_mutex1.lock();   //语句4
                if (!Msg.empty()) {
                        //读取消息，读完删除
                        command = Msg.front();
                        Msg.pop_front();
```

```
                        my_mutex1.unlock();
                        my_mutex2.unlock();
                        return true;
                }
                my_mutex1.unlock();
                my_mutex2.unlock();
                return false;
        }private:
        std::list<int> Msg;  //消息变量
        std::mutex my_mutex1; //互斥量对象1
        std::mutex my_mutex2; //互斥量对象2
};
int main() {
        A a;
        //创建一个插入消息线程
        std::thread insertTd(&A::insertMsg, &a);
        //这里要传入引用保证是同一个对象
        //创建一个读取消息线程
        std::thread readTd(&A::readMsg, &a);
        //这里要传入引用保证是同一个对象
        insertTd.join();
        readTd.join();
        return 0;
}
```

语句 1 和语句 2 表示线程 A 先锁资源 1，再锁资源 2，语句 3 和语句 4 表示线程 B 先锁资源 2 再锁资源 1，具备死锁产生的条件。

3、死锁的解决方案

保证上锁的顺序一致。

4、死锁必要条件

- 互斥条件：进程对所需求的资源具有排他性，若有其他进程请求该资源，请求进程只能等待。
- 不剥夺条件：进程在所获得的资源未释放前，不能被其他进程强行夺走，只能自己释放
- 请求和保持条件：进程当前所拥有的资源在进程请求其他新资源时，由该进程继续占有。
- 循环等待条件：存在一种进程资源循环等待链，链中每个进程已获得的资源同时被链中下一个进程所请求。

5、处理方法

主要有以下四种方法：

- 鸵鸟策略
- 死锁检测与死锁恢复
- 死锁预防
- 死锁避免

鸵鸟策略

把头埋在沙子里，假装根本没发生问题。

因为解决死锁问题的代价很高，因此鸵鸟策略这种不采取任务措施的方案会获得更高的性能。

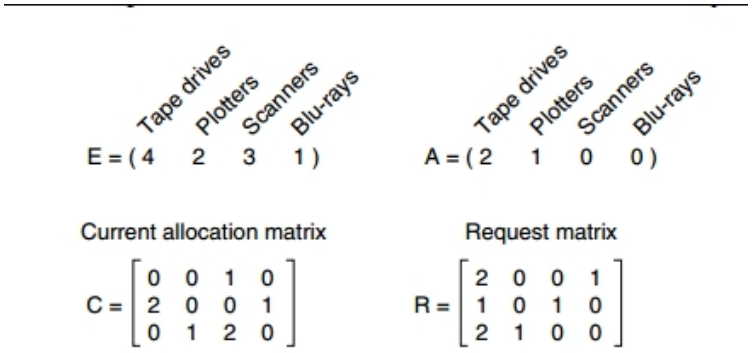
当发生死锁时不会对用户造成多大影响，或发生死锁的概率很低，可以采用鸵鸟策略。

大多数操作系统，包括 Unix，Linux 和 Windows，处理死锁问题的办法仅仅是忽略它。

死锁检测与死锁恢复

不试图阻止死锁，而是当检测到死锁发生时，采取措施进行恢复。

1、每种类型一个资源的死锁检测

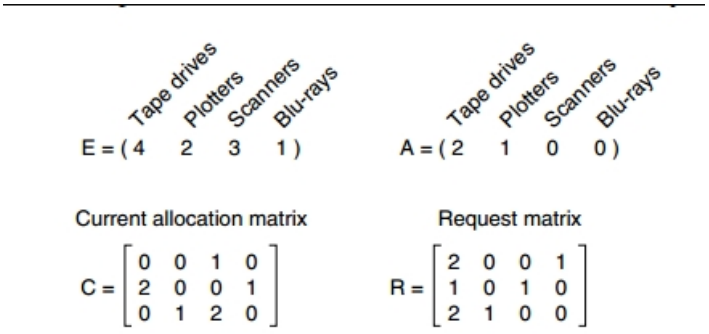


上图为资源分配图，其中方框表示资源，圆圈表示进程。资源指向进程表示该资源已经分配给该进程，进程指向资源表示进程请求获取该资源。

图 a 可以抽取出环，如图 b，它满足了环路等待条件，因此会发生死锁。

每种类型一个资源的死锁检测算法是通过检测有向图是否存在环来实现，从一个节点出发进行深度优先搜索，对访问过的节点进行标记，如果访问了已经标记的节点，就表示有向图存在环，也就是检测到死锁的发生。

2、每种类型多个资源的死锁检测



上图中，有三个进程四个资源，每个数据代表的含义如下：

- E 向量：资源总量
- A 向量：资源剩余量
- C 矩阵：每个进程所拥有的资源数量，每一行都代表一个进程拥有资源的数量
- R 矩阵：每个进程请求的资源数量

进程 P_1 和 P_2 所请求的资源都得不到满足，只有进程 P_3 可以，让 P_3 执行，之后释放 P_3 拥有的资源，此时 A = (2 2 2 0)。P_2 可以执行，执行后释放 P_2 拥有的资源，A = (4 2 2 1) 。P_1 也可以执行。所有进程都可以顺利执行，没有死锁。

算法总结如下：

每个进程最开始时都不被标记，执行过程有可能被标记。当算法结束时，任何没有被标记的进程都是死锁进程。

1. 寻找一个没有标记的进程 P_i，它所请求的资源小于等于 A。
2. 如果找到了这样一个进程，那么将 C 矩阵的第 i 行向量加到 A 中，标记该进程，并转回 1。
3. 如果没有这样一个进程，算法终止。

6、死锁恢复

- 利用抢占恢复
- 利用回滚恢复
- 通过杀死进程恢复

7、死锁预防

在程序运行之前预防发生死锁。

1. 破互斥条件

   例如假脱机打印机技术允许若干个进程同时输出，唯一真正请求物理打印机的进程是打印机守护进程。

2. 破坏请求和保持条件

   一种实现方式是规定所有进程在开始执行前请求所需要的全部资源。

3. 破坏不剥夺条件

   允许抢占资源

4. 破循环请求等待

   给资源统一编号，进程只能按编号顺序来请求资源。

8、死锁避免

1.在程序运行时避免发生死锁。

安全状态

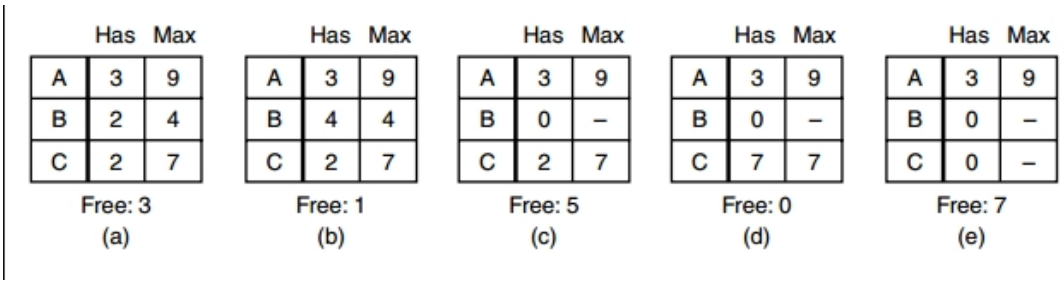


图 a 的第二列 Has 表示已拥有的资源数，第三列 Max 表示总共需要的资源数，Free 表示还有可以使用的资源数。

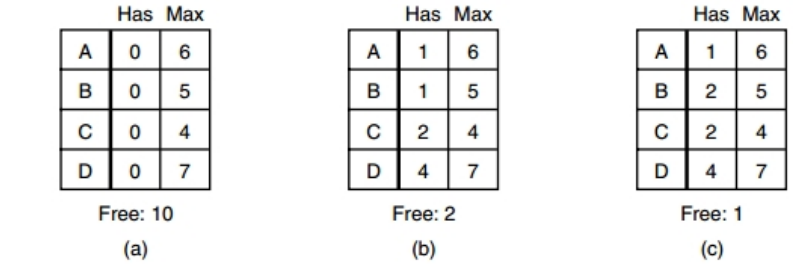
从图 a 开始出发，先让 B 拥有所需的所有资源（图 b），运行结束后释放 B，此时 Free 变为 5（图 c）；接着以同样的方式运行 C 和 A，使得所有进程都能成功运行，因此可以称图 a 所示的状态时安全的。

定义：如果没有死锁发生，并且即使所有进程突然请求对资源的最大需求，也仍然存在某种调度次序能够使得每一个进程运行完毕，则称该状态是安全的。

安全状态的检测与死锁的检测类似，因为安全状态必须要求不能发生死锁。下面的银行家算法与死锁检测算法非常类似，可以结合着做参考对比。

2. 单个资源的银行家算法

一个小城镇的银行家，他向一群客户分别承诺了一定的贷款额度，算法要做的是判断对请求的满足是否会进入不安全状态，如果是，就拒绝请求；否则予以分配。



上图 c 为不安全状态，因此算法会拒绝之前的请求，从而避免进入图 c 中的状态。

多个资源的银行家算法

上图中有五个进程，四个资源。

左边的图表示已经分配的资源，右边的图表示还需要分配的资源。最右边的 E、P 以及 A 分别表示：总资源、已分配资源以及可用资源，注意这三个为向量，而不是具体数值，例如 A=(1020)，表示 4 个资源分别还剩下 1/0/2/0。

4、检查一个状态是否安全的算法如下：

- 查找右边的矩阵是否存在一行小于等于向量 A。如果不存在这样的行，那么系统将会发生死锁，状态是不安全的。
- 假若找到这样一行，将该进程标记为终止，并将其已分配资源加到 A 中。
- 重复以上两步，直到所有进程都标记为终止，则状态时安全的。

如果一个状态不是安全的，需要拒绝进入这个状态。

60、为什么分段式存储管理有外部碎片而无内部碎片？为什么固定分区分配有内部碎片而不会有外部碎片？

分段式分配是按需分配，而固定式分配是固定分配的方式。

61、内部碎片与外部碎片

内碎片：分配给某些进程的内存区域中有些部分没用上，常见于固定分配方式

内存总量相同，100M

固定分配，将 100M 分割成 10 块，每块 10M，一个程序需要 45M，那么需要分配 5 块，第五块只用了 5M，剩下的 5M 就是内部碎片；

分段式分配，按需分配，一个程序需要 45M，就给分片 45MB，剩下的 55M 供其它程序使用，不存在内部碎片。

外碎片：内存中某些空闲区因为比较小，而难以利用上，一般出现在内存动态分配方式中

分段式分配：内存总量相同，100M，比如，内存分配依次 5M，15M，50M，25M，程序运行一段时间之后，5M，15M 的程序运行完毕，释放内存，其他程序还在运行，再次分配一个 10M 的内存供其它程序使用，只能从头开始分片，这样，就会存在 10M+5M 的外部碎片

62、如何消除碎片文件

对于外部碎片，通过**紧凑技术**消除，就是操作系统不时地对进程进行移动和整理。但是这需要动态重定位寄存器地支持，且相对费时。紧凑地过程实际上类似于 Windows 系统中地磁盘整理程序，只不过后者是对外存空间地紧凑

解决外部内存碎片的问题就是**内存交换**。

可以把音乐程序占用的那 256MB 内存写到硬盘上，然后再从硬盘上读回来到内存里。不过再读回的时候，我们不能装载回原来的位置，而是紧紧跟着那已经被占用了的 512MB 内存后面。这样就能空缺出连续的 256MB 空间，于是新的 200MB 程序就可以装载进来。

回收内存时要尽可能地将相邻的空闲空间合并。

63、冯诺依曼结构有哪几个模块？分别对应现代计算机的哪几个部分？（百度安全一面）

- 存储器：内存
- 控制器：南桥北桥
- 运算器：CPU
- 输入设备：键盘
- 输出设备：显示器、网卡

64、多进程和多线程的区别是什么？换句话说，什么时候该用多线程，什么时候该用多进程？

- 频繁修改：需要频繁创建和销毁的优先使用**多线程**
- 计算量：需要大量计算的优先使用**多线程** 因为需要消耗大量 CPU 资源且切换频繁，所以多线程好一点
- 相关性：任务间相关性比较强的用**多线程**，相关性比较弱的用多进程。因为线程之间的数据共享和同步比较简单。
- 多分布：可能要扩展到多机分布的用**多进程**，多核分布的用**多线程**。

但是实际中更常见的是进程加线程的结合方式，并不是非此即彼的。

65、服务器高并发的解决方案你知道多少？

- 应用数据与静态资源分离 将静态资源（图片，视频，js，css 等）单独保存到专门的静态资源服务器中，在客户端访问的时候从静态资源服务器中返回静态资源，从主服务器中返回应用数据。
- 客户端缓存 因为效率最高，消耗资源最小的就是纯静态的 html 页面，所以可以把网站上的页面尽可能用静态的来实现，在页面过期或者有数据更新之后再将页面重新缓存。或者先生成静态页面，然后用 ajax 异步请求获取动态数据。
- 集群和分布式 （集群是所有的服务器都有相同的功能，请求哪台都可以，主要起分流作用）
  （分布式是将不同的业务放到不同的服务器中，处理一个请求可能需要使用到多台服务器，起到加快请求处理的速度。）
  可以使用服务器集群和分布式架构，使得原本属于一个服务器的计算压力分散到多个服务器上。同时加快请求处理的速度。
- 反向代理 在访问服务器的时候，服务器通过别的服务器获取资源或结果返回给客户端。