1. 进程、线程、协程

1. 为什么使用线程：服务器为进程配置的资源太多，且切换开销较大，所以使用线程
2. 进程和线程的区别
    1. 进程是操作系统资源分配的最小单位，线程是操作系统调度的最小单位
    2. 进程拥有独立的地址空间，在启动时分配，同时包含进程控制块、代码段、数据段、堆、栈

       线程的地址空间共享，仅拥有自己的线程栈
    3. 通信方式不同，线程共享全局变量、静态变量，共享更方便

       进程需要使用信号等等手段
    4. 进程间较为独立，进程崩溃一般不会影响其他进程

       线程崩溃，该进程的所有线程崩溃，因为线程共享地址空间
    5. 进程切换开销较大，线程切换开销较小
    6. 创建和撤销进程时，系统要为之分配/回收资源、内存空间、IO设备，开销远大于创建和撤销线程的开销，线程的切换只需要保存和设置少量寄存器的内容，开销小
3. 如何选择进程和线程

   线程切换代价小，所以如果频繁创建、销毁却又包含大量计算，使用线程。例如界面显示和及时响应消息。

   允许并发且有阻塞的socket/磁盘通常也是用线程
4. 线程间的通信方式

   临界区，使用互斥锁/信号量解决临界区变量
5. 进程间的通信方式
    1. 管道
        1. 无名管道：半双工，只能用于父子进程间的通信，使用read/write、文件描述符进行读取，是内存中的特殊文件

           int fd[2]　pipe(fd)创建无名管道，其中fd[0]为读端，fd[1]为写端

           <1>当写端存在时，管道中没有数据时，读取管道时将阻塞

           <2>当读端请求读取的数据大于管道中的数据时，此时读取管道中实际大小的数据

           <3>当读端请求读取的数据小于管道中的数据时，此时放回请求读取的大小数据

           <4>向管道中写入数据时，linux不保证写入的原子性，管道缓冲区一有空闲区域，写进程就会试图向管道写入数据。当管道满时，读进程不读走管道缓冲区中的数据，那么写操作将一直阻塞。

           如果没有读端，写进程收到SIGPIPE信号
        2. 命名管道：FIFO半双工，是文件系统的特殊文件（存在于内核之中，无亲缘关系也可以使用其进行通信）

```
mkfifo("tp", 0644);
int infd;
infd = open("abc", O_RDONLY);
if (infd == -1) ERR_EXIT("open");

int outfd;
outfd = open("tp", O_WRONLY);
```

创建命名管道

```
unlink("zieckey_fifo");
```

```
mkfifo("zieckey_fifo",0777);
```

读写数据

```
fd=open("zieckey_fifo",O_RDONLY);
```

特点：面向字节流，生命周期随内核，自带同步互斥机制，半双工、单向通信

2. 消息队列：存放于内核之中，不一定按照顺序进行读取

   消息队列由标识符唯一标记

   1. 消息队列可以认为是一个全局的一个链表，链表节点钟存放着数据报的类型和内容，有消息队列的标识符进行标记。
   2. 消息队列允许一个或多个进程写入或者读取消息。
   3. 消息队列的生命周期随内核。
   4. 消息队列可实现双向通信。

3. 信号量：进程间的互斥与同步

   同一进程中互斥，不同进程中同步

   在保证互斥的同时提高了并发，可用于进程、线程间的同步

   sem_init sem_wait sem_post

4. 共享内存：最快的IPC方式

   但是要保证共享内存处的线程安全问题。（epoll使用mmap进行共享内存，减少了从内核中拷贝数据到进程段的时间）

   1. 不用从用户态到内核态的频繁切换和拷贝数据，直接从内存中读取就可以。
   2. 共享内存是临界资源，所以需要操作时必须要保证原子性。使用信号量或者互斥锁都可以。
   3. 生命周期随内核。

5. socket

6. 信号

7. 文件（fork后使用相同的文件描述符）

6. 线程同步与线程异步

   1. 线程同步：一个线程需要等待另一个线程执行结束后才能进行
   2. 线程异步：无需等待另一个线程结束
   3. 同步的目的：多线程操作数据保证前后一致性
   4. 互斥的目的：多线程之间进行数据操作不会相互影响

7. 多线程同步、互斥的方法

   互斥锁、信号量，信号，条件变量

   互斥锁： pthread_mutex_init

   pthread_mutex_destroy

   pthread_mutex_lock

   pthread_mutex_unlock

   信号量：阻塞sem_wait

P操作的主要动作是： [1]

①S减1； [1]

②若S减1后仍大于或等于0，则进程继续执行； [1]

③若S减1后小于0，则该进程被阻塞后放入等待该信号量的等待队列中，然后转进程调度。 [1]

V操作的主要动作是： [1]

①S加1； [1]

②若相加后结果大于0，则进程继续执行； [1]

③若相加后结果小于或等于0，则从该信号的等待队列中释放一个等待进程，然后再返回原进程继续执行或转进程调度。

信号量
信号量是一种特殊的变量，可用于线程同步。它只取自然数值，并且只支持两种操作：
P(SV):如果信号量SV大于0，将它减一；如果SV值为0，则挂起该线程。
V(SV)：如果有其他进程因为等待SV而挂起，则唤醒，然后将SV+1；否则直接将SV+1。
其系统调用为：
sem_wait（sem_t *sem）：以原子操作的方式将信号量减1，如果信号量值为0，则sem_wait将被阻塞，直到这个信号量具有非0值。
sem_post（sem_t *sem)：以原子操作将信号量值+1。当信号量大于0时，其他正在调用sem_wait等待信号量的线程将被唤醒。

互斥量
互斥量又称互斥锁，主要用于线程互斥，不能保证按序访问，可以和条件锁一起实现同步。当进入临界区　　时，需要获得互斥锁并且加锁；当离开临界区时，需要对互斥锁解锁，以唤醒其他等待该互斥锁的线程。其主要的系统调用如下：
pthread_mutex_init:初始化互斥锁
pthread_mutex_destroy：销毁互斥锁
pthread_mutex_lock：以原子操作的方式给一个互斥锁加锁，如果目标互斥锁已经被上锁，pthread_mutex_lock调用将阻塞，直到该互斥锁的占有者将其解锁。
pthread_mutex_unlock:以一个原子操作的方式给一个互斥锁解锁。

条件变量
条件变量，又称条件锁，用于在线程之间同步共享数据的值。条件变量提供一种线程间通信机制：当某个共享数据达到某个值时，唤醒等待这个共享数据的一个/多个线程。即，当某个共享变量等于某个值时，调用 signal/broadcast。此时操作共享变量时需要加锁。其主要的系统调用如下：
pthread_cond_init:初始化条件变量
pthread_cond_destroy：销毁条件变量
pthread_cond_signal：唤醒一个等待目标条件变量的线程。哪个线程被唤醒取决于调度策略和优先级。
pthread_cond_wait：等待目标条件变量。需要一个加锁的互斥锁确保操作的原子性。该函数中在进入wait状态前首先进行解锁，然后接收到信号后会再加锁，保证该线程对共享资源正确访问。

8. 进程上限：4096，使用limit -u 进行修改

   最终上限：pid的上限32768（short）

   线程上限：依据虚拟内存大小/线程栈的大小决定（大约300个） 3072M/8M = 384个

   由于代码段和数据段-1个，主线程-1个为382个

   使用ulimit -s减少栈的大小，但是无法突破1024的上限

9. 协程

   一个进程包含多个线程，一个线程包含多个协程

   协程又称微线程、纤程。在执行过程中，在子程序内部可中断。然后去执行别的子程序，再返回来继续执行

   协程的子程序之间可以随时中断去执行另一个协程，特点为在一个线程执行

   协程切换不消耗资源在用户态进行

   线程内的多个协程进行切换，但是是串行执行的

   协程为非抢占式调度，切换由程序员控制

10. 进程、线程、协程的切换对比

    1. 进程的切换者--操作系统，用户无感知，使用进程的调度算法（FCFS，SJF，时间片.....）

       切换内容：全局目录，内核栈，硬件上下文保存在内存中，nei内核态-用户态-内核态

       上下文切换补充：

       1. 切换页目录使用新的地址空间--虚拟空间变换
       2. 切换内核栈、硬件上下文
       3. 更换PCB

    4. 页表
    5. 刷新TLB（是页表的cache）
2. 线程的切换者--操作系统，用户无感知

   切换内容：线程栈、硬件上下文。线程切换内容保存在内核栈中，nei内核态-用户态-内核态
3. 协程的切换者--用户，只是切换硬件上下文，切换内存保存在用户自定义变量中，没有进入内核态

11. 孤儿进程和僵尸进程
    1. 孤儿进程：父进程先于子进程结束，子进程由init收养，回收由init进行
    2. 僵尸进程：子进程退出，资源未被父进程回收，但是占用了进程号等资源。如果大量存在僵尸进程，那么可能会导致进程号分配不足而无法fork
    3. 解决方案
        1. kill父进程，让子进程转由init收养
        2. 子进程退出时向父进程发送SIGCHRD信号，信号处理函数中进行回收
12. 进程调度算法：
    1. FCFS：每次从就绪队列中选取头部调度
    2. SJF：平均等待时间、平均周转时间最低

       对长作业不利，容易造成饥饿现象

       程序运行时间由用户提供，不一定真正做到短作业优先
    3. 优先级调度算法
    4. 时间片轮转算法
    5. 多级反馈队列调度算法：多个FCFS+时间片轮转
13. 多进程与多线程

对比维度	多进程	多线程	总结
数据共享、同步	数据共享复杂，需要用IPC；数据是分开的，同步简单	因为共享进程数据，数据共享简单，但也是因为这个原因导致同步复杂	各有优势
内存、CPU	占用内存多，切换复杂，CPU利用率低	占用内存少，切换简单，CPU利用率高	线程占优
创建销毁、切换	创建销毁、切换复杂，速度慢	创建销毁、切换简单，速度很快	线程占优
编程、调试	编程简单，调试简单	编程复杂，调试复杂	进程占优
可靠性	进程间不会互相影响	一个线程挂掉将导致整个进程挂掉	进程占优
分布式	适应于多核、多机分布式；如果一台机器不够，扩展到多台机器比较简单	适应于多核分布式	进程占优

    1. 频繁的创建、销毁：线程 -- web服务器
    2. 大量计算：线程 -- 切换频繁
    3. 强相关：进程 -- 消息解码、消息处理
    4. 弱相关：线程 -- 消息接受、消息发送

5. 多机分布：进程　　多核分布：线程

IO密集型：多线程

CPU密集型：多进程

14. 进程和线程的概念
    1. 进程是运行时对程序的封装，是操作系统资源调度和分配的基本单位，实现了操作系统的并发
    2. 线程是进程的子任务，是CPU调度的基本单位，实现进程的并发。线程独自占用虚拟存储器，拥有独立的寄存器组、PC和处理器状态，但是公用地址空间和文件队列
15. 线程存在的必要性
    1. 资源节省 -- 切换开销小 -- 通信方便
    2. 进程存在缺点：同一时间只能干一件事，如果有阻塞整个进程会被挂起

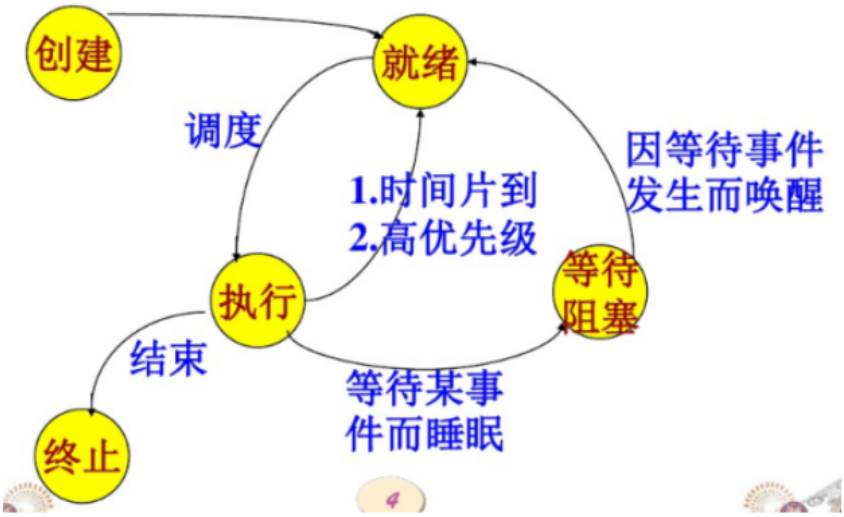
       引入线程作为并发调度的基本单位，减少程序并发执行付出的时空开销
    3. 从资源上来讲，线程是一种非常"节俭"的多任务操作方式。在linux系统下，启动一个新的进程必须分配给它独立的地址空间，建立众多的数据表来维护它的代码段、堆栈段和数据段，这是一种"昂贵"的多任务工作方式。
    4. 从切换效率上来讲，运行于一个进程中的多个线程，它们之间使用相同的地址空间，而且线程间彼此切换所需时间也远远小于进程间切换所需要的时间。据统计，一个进程的开销大约是一个线程开销的30倍左右
    5. 从通信机制上来讲，线程间方便的通信机制。对不同进程来说，它们具有独立的数据空间，要进行数据的传递只能通过进程间通信的方式进行，这种方式不仅费时，而且很不方便。线程则不然，由于同一进城下的线程之间贡献数据空间，所以一个线程的数据可以直接为其他线程所用，这不仅快捷，而且方便。
    6. 使多CPU系统更加有效。操作系统会保证当线程数不大于CPU数目时，不同的线程运行于不同的CPU上。
    7. 改善程序结构。一个既长又复杂的进程可以考虑分为多个线程，成为几个独立或半独立的运行部分，这样的程序才会利于理解和修改。
16. 单核程序写多线程是否要加锁

    需要，因为线程执行顺序不定

    抢占式操作系统中，每个线程分配时间片，当某线程时间片耗尽时，会将其转入就绪队列，然后运行另一个线程。如果共享数据，必须加锁保证线程安全
17. 如果进程在时间片结束前阻塞/结束，CPU立即进行切换，该进程调出就绪队列
18. 线程需要的上下文
    1. 当前线程id，线程状态、堆栈、寄存器（SP，PC，EAX）
    2. 线程栈大小固定，运行时决定，开辟在堆区
    3. 进程栈大小运行时决定
    4. SP：堆栈寄存器，指向栈顶地址

       PC：程序计数器，存储指令地址

       EAX：累加计数器，加乘的缺省计数器
19. 进程的五种状态



执行状态才能进入阻塞，阻塞完进入就绪状态

创建状态：PCB分配，进程创建，分配虚拟内存

就绪状态：进程加入就绪队列**等待CPU调度**

执行状态：进程正在运行

阻塞状态：IO等原因阻塞等待

终止状态：进程运行完毕

20. 交换技术

    多个进程竞争资源，在内存中全部阻塞导致等待IO的情况。此时交换技术将进程换入换出

    1. 交换技术 ： 进程换出一部分到外存

       讲暂时不能运行的进程换出到外存--挂起

    2. 虚拟存储技术：每个进程只能转入一部分程序和数据

    3. 交换技术的几种内存状态

        1. 活动阻塞：进程活跃中，且被阻塞
        2. 静止阻塞：进程在外存，且被阻塞
        3. 活动就绪：进程在内存，就绪状态
        4. 静止就绪：进程在外存，就绪状态

        活动就绪 —— 静止就绪　　(内存不够，调到外存)

        活动阻塞 —— 静止阻塞　　(内存不够，调到外存)

        执行　—— 静止就绪　　(时间片用完)

21. 守护进程：用于执行特定的系统任务、在系统引导时启动，并且一直运行知道系统关闭

2. 死锁

1. 死锁产生的四个条件

   互斥、请求并保持、不可剥夺、循环等待

2. 死锁预防：限制条件严格、效率低

    1. 破坏互斥条件：不可破荒
    2. 破坏请求并保持：一次性分配所有资源给进程，可能导致饥饿
    3. 破坏不可剥夺：如果进程资源请求不能满足，那么释放进程
    4. 破坏循环等待：顺序资源分配法，给资源标号，进程按照标号顺序进行申请

3. 死锁避免：避免不安全状态--银行家算法

安全状态：动态检查内存资源分配，如果存在一条可以合理分配所有资源的安全序列，则为安全状态

不安全状态：没有安全序列存在

不安全状态也可能不导致死锁

4. 死锁检测：资源分配图查看环路
5. 死锁解除：资源剥夺，进程撤销，进程回退

3. 线程池

线程池：创建好的大量线程，减少调用时再创建的时间。初始后处于空闲状态，当有新任务进来时，取出进行处理。（生产者消费者模型）

1. 设置生产者消费者队列，作为临界资源
2. 初始化n个线程并运行，加锁取任务
3. 当任务队列为空时，所有阻塞
4. 生产者来了任务后，对队列加锁，任务挂在队列上，然后使用条件变量通知阻塞的线程

4. fork

1. fork、vfork、clone的区别
    1. fork：调用一次返回两次，复制一份进程映像（写时复制原则）

        父进程返回子进程pid，子进程返回0，出错返回-1

        写时复制原则：fork创建时不进行内存的copy，二者虚拟内存相同但是物理内存不同。此时共享页面为只读。当子进程对共享内容需要进行修改时，os再对内存进行拷贝复制。如果子进程调用exec或者exit，就减少了一次内存复制开销

    2. vfork：创建一个子进程

        父子进程共享空间，但是函数运行后父进程阻塞，只有子程序调用exec/exit时，父进程结束阻塞。（由于子进程没有独立地址空间，认为其不是一个进程）

    3. clone：创建一个线程、进程

        clone与fork的区别：不再复制父进程的空间，而是创建新的，在函数中指定大小

2. fork、vfork、clone的联系
    1. 都是linux系统调用
    2. 分别调用了sys_fork,sys_vfork,sys_clone，最终调用do_fork

写时复制原则--只拷贝页表

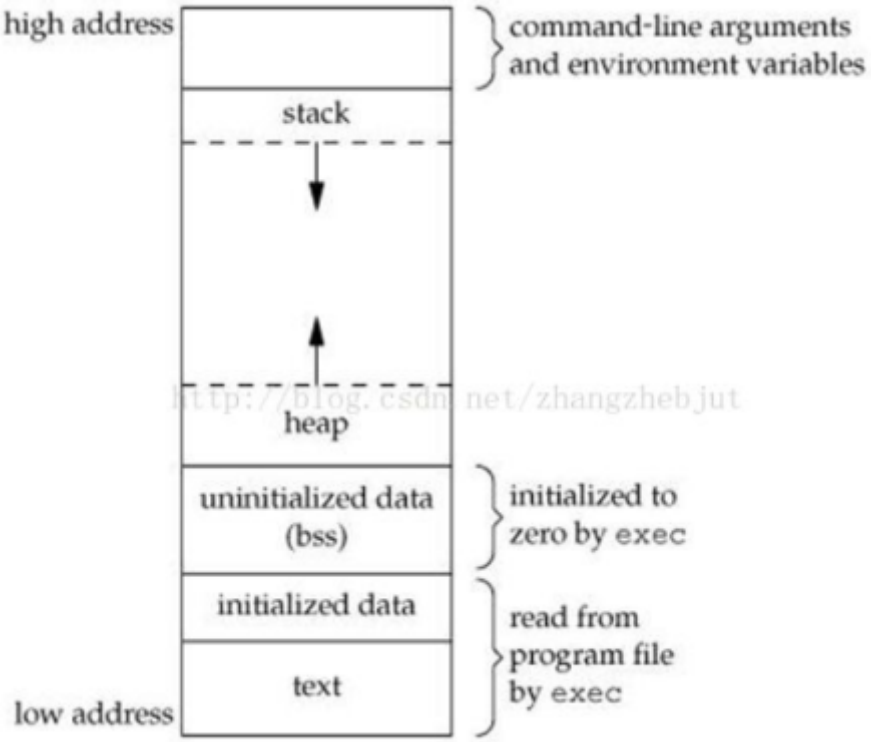
写时复制采用惰性优化方法避免复制时的系统开销，如果有多个进程要读取它们自己的那部分的副本，那么复制是不必要的。每个进程只要保存一个指向这个资源的指针就可以了。只要没有进程要去修改自己的“副本”

写时复制的主要好处在于：如果进程从来就不需要修改资源，则不需要进行复制。惰性算法的好处就在于它们尽量推迟代价高昂的操作，直到必要的时刻才会去执行。

在使用虚拟内存的情况下，写时复制（Copy-On-Write）是以页为基础进行的。所以，只要进程不修改它全部的地址空间，那么就不必复制整个地址空间。在fork( )调用结束后，父进程和子进程都相信它们有一个自己的地址空间，但实际上它们共享父进程的原始页，接下来这些页又可以被其他的父进程或子进程共享。

写时复制在内核中的实现非常简单。与内核页相关的数据结构可以被标记为只读和写时复制。如果有进程试图修改一个页，就会产生一个缺页中断。内核处理缺页中断的方式就是对该页进行一次透明复制。这时会清除页面的COW属性，表示着它不再被共享。

5. Linux进程地址空间



1. 低3G地址空间：用户态

   地址空间分布--通过页表投射到实际内存中

   代码段.text：只读二进制代码，同时包含一些静态字符串常量

   数据段.data：已初始化的全局和静态变量

   .bss：未初始化的全局和静态变量--运行时初始化为0

   常量

   堆：动态内存分配，采用隐式链表和显式空闲链表

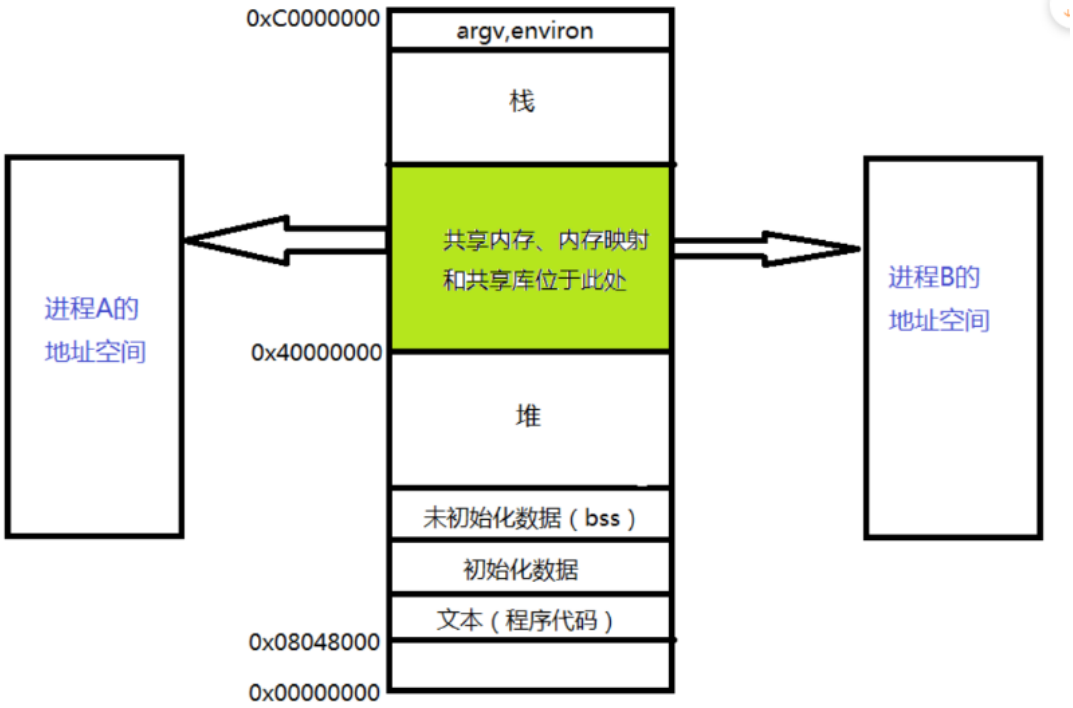
   映射区：存储动态链接库、调用mmap进行的文件映射

   栈：存储自动变量，函数的调用和返回，自动释放

2. 高1G地址空间：内核态
3. 栈溢出：在函数调用的过程中，如果不断向栈中压入函数栈帧，可能导致栈满，页故障，调用栈增长函数，如果达到了上限，触发段错误（栈溢出）
4. 堆和栈的区别

	栈	堆
管理方式	栈由编译器自动管理，无需程序员手工控制	堆空间的申请释放工作由程序员控制，容易产生内存泄漏
空间大小	栈是向低地址扩展的数据结构，是一块连续的内存区域。栈顶的地址和栈的最大容量是系统预先规定好的，当申请的空间超过栈的剩余空间时，将提示溢出。因此，用户能从栈获得的空间较小。	堆是向高地址扩展的数据结构，是不连续的内存区域。因为系统是用链表来存储空闲内存地址的，且链表的遍历方向是由低地址向高地址。由此可见，堆获得的空间较灵活，也较大。栈中元素都是一一对应的，不会存在一个内存块从栈中间弹出的情况
是否产生碎片	对于栈来讲，则不会存在这个问题	对于堆来讲，频繁的 malloc/free（new/delete）势必会造成内存空间的不连续，从而造成大量的碎片，使程序效率降低
增长方向	栈的增长方向是向下的，即向着内存地址减小的方向	堆的增长方向是向上的，即向着内存地址增加的方向
分配方式	栈的分配和释放是由编译器完成的，栈的动态分配由 alloca()函数完成，但是栈的动态分配和堆是不同的，它的动态分配是由编译器进行申请和释放的，无需手工完成。	堆都是程序中由 malloc()函数动态申请分配并由 free()函数释放
分配效率	栈是操作系统提供的数据结构，计算机会在底层对栈提供支持：分配专门的寄存器存放栈的地址，压栈出栈都有专门的指令执行。	堆则是 C 函数库提供的，它的机制很复杂，例如为了分配一块内存，库函数会按照一定的算法在堆内存中搜索可用的足够大的空间，如果没有足够大的空间（可能是由于内存碎片太多），需要操作系统来重新整理内存空间，这样就有机会分到足够大小的内存，然后返回。显然，堆的效率比栈要低得多

5. malloc和alloc的区别
    1. malloc向堆申请内存，需要手动释放，同时使用memset初始化申请内存，realloc调整大小，sbrk增加数据段大小，free释放
    2. alloc向栈中申请内存，同时自动初始化，无需手动释放
    3. calloc自动分配内存空间同时初始化为0
6. 共享内存的位置



7. 内存页面置换方法--发生缺页中断的时候需要进行替换

    在更换页面时，如果更换页面是一个很快会被再次访问的页面，则再次缺页中断后又很快会发生新的缺页中断。

    1. 最佳置换方法OPT：未来最长不使用
    2. FIFO置换算法：表现差：只考虑了调入时间，没有考虑使用频率

3. LRU置换算法：颠簸，缓存污染，偶发性缓存大量存在
4. LFU最不经常访问：颠簸
5. CLOCK算法

8. 页面抖动

   在页面置换过程中的一种最糟糕的情形是，刚刚换出的页面马上又要换入主存，刚刚换入的页面马上就要换出主存，这种频繁的页面调度行为称为抖动，或颠簸。如果一个进程在换页上用的时间多于执行时间，那么这个进程就在颠簸。

   频繁的发生缺页中断（抖动），其主要原因是某个进程频繁访问的页面数目高于可用的物理页帧数目。虚拟内存技术可以在内存中保留更多的进程以提高系统效率。在稳定状态，几乎主存的所有空间都被进程块占据，处理机和操作系统可以直接访问到尽可能多的进程。但如果管理不当，处理机的大部分时间都将用于交换块，即请求调入页面的操作，而不是执行进程的指令，这就会大大降低系统效率。

9. mmap映射：将用户空间的一段程序映射到内核中，进程对其的修改将同步到内核

```
void *mmap(void *adrr, size_t length, int prot, int flags, int fd, off_t offset);
//成功：返回创建的映射区首地址；失败：MAP_FAILED宏
```

   参数

   - addr: 建立映射区的首地址，由Linux内核指定。使用时，直接传递NULL
   - length： 欲创建映射区的大小
   - prot： 映射区权限`PROT_READ`、`PROT_WRITE`、`PROT_READ|PROT_WRITE`
   - flags：标志位参数(常用于设定更新物理区域、设置共享、创建匿名映射区)
     `MAP_SHARED`：会将映射区所做的操作反映到物理设备（磁盘）上。
     `MAP_PRIVATE`：映射区所做的修改不会反映到物理设备。
   - fd： 用来建立映射区的文件描述符
   - offset： 映射文件的偏移(4k的整数倍)

   mmap分配的空间也需要进行释放

   mmap建立的映射区在使用结束后也应调用类似free的函数来释放。

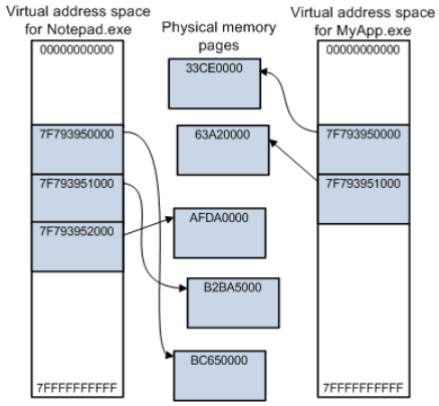
```
int munmap(void *addr, size_t length);
//成功：返回0； 失败：返回-1
```

10. 使用虚拟地址空间的好处

    1. 内存保护：防止不同进程同一时刻对物理地址进行修改，每个进程拥有独立的虚拟内存，互相不能进行干扰，同时对特定内存地址提供写保护，防止代码/数据篡改
    2. 不同进程在运行过程中，所看到的是自己独自占有了当前的32位4G内存，所有进程共享物理内存，每个进程只把自己需要的映射并存储到物理内存中。
    3. 进程创建和加载时，只是创建了虚拟内存布局，初始化了内存链表，而不把对应位置的程序数据拷贝到内存。

       只有运行到该程序时，通过缺页异常来拷贝

       在运行过程中，动态分配内存也是分配虚拟内存，只有对虚拟内存对应的页表项做出设置，并且访问到该数据时，才会触发缺页异常。
    4. 扩大了地址空间
    5. 无需再连续分配物理空间，虚拟空间连续，物理空间不连续



11. 使用虚拟内存的代价
    1. 建立数据结构管理内存
    2. 虚地址向物理地址的转换增加事件
    3. 页面换入换出需要IO事件
    4. 一页只有一部分数据可能有浪费
12. malloc分配空间的算法
    1. 首次适应：每次从低地址找能满足大小的
    2. 最佳适应：找到大小满足的，最小的空间
    3. 最坏适应：防止小碎片产生，分配最大的连续空闲区
    4. 临近时应：从当前查找第一个满足的
13. 缺页中断
    1. malloc和mmap分配时只建立虚拟的地址空间，需要使用时，触发缺页中断
    2. 缺页中断：请求分页系统中，查询页表状态位来确定访问的页面是否在内存中，不在内存中时，产生缺页中断，将其调入内存
    3. 缺页中断步骤
        1. 保护CPU现场
        2. 分析中断原因
        3. 缺页中断处理程序
        4. 恢复CPU现场
    4. 缺页中断是硬件中断，与普通中断存在区别
        1. 指令执行期间进行中断
        2. 一条指令可以产生多个中断
        3. 中断返回当前指令，其他返回下一条指令
14. CPU内存管理机制

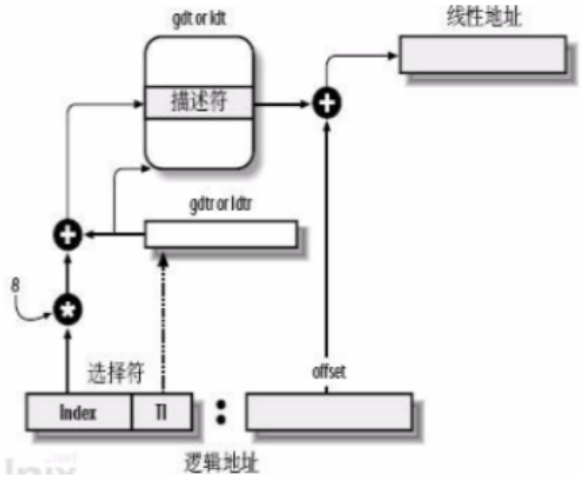
    物理内存：内存芯片的寻址，与地址总线对应

    虚拟地址：是对内存地址的抽象描述

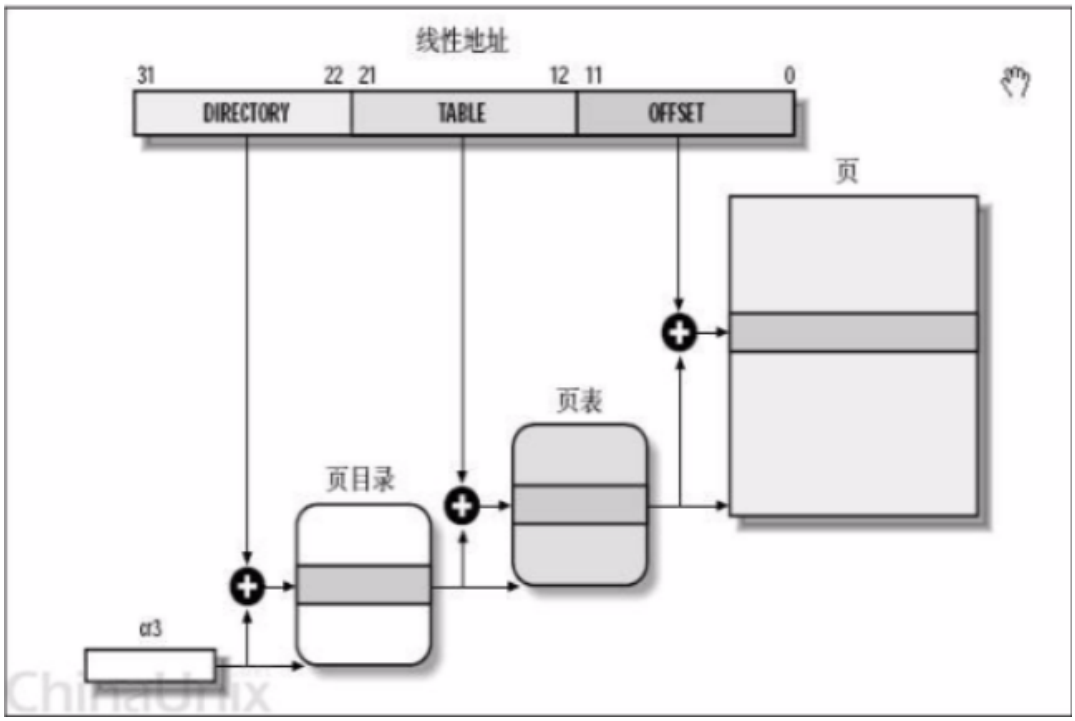
    地址转换方式：

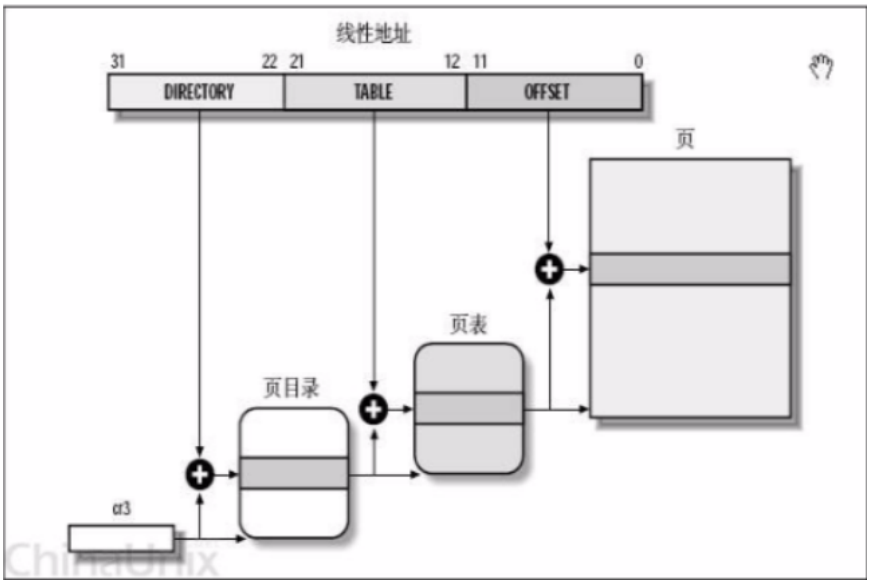
    1. 段式存储 -- 代码段、数据段.....

       段标识符+段内偏移



2. 页式存储





3. 段页式结合，开销增加，硬件内存增加

4. 各自的优缺点

    1. 页式存储：没有外部碎片，内部碎片最大为页大小4K
    2. 段式存储：没有内部碎片，存在外部碎片

对每个模块进行编写编译，对不同的段采取不同的保护方式

15. 堆：当进程未调用malloc是没有堆段的，只有调用malloc时分配一个堆，并且程序运行时可以动态增加堆大小（移动break指针），从低地址向高地址增长

    栈：使用栈空间存储函数的返回地址，参数、局部变量、返回值，从高地址向低地址增长，有一个最小栈大小

16.

6. 信号

1. 信号与信号量的区别

   信号：由用户、系统向目标进程发送的信息

   信号量：本质计数器，记载了临界资源的数目，进程的PV均为原子操作。协调进程对共享资源的访问，实现进程间的互斥、同步

7. 用户栈和内核栈

用户态：堆栈指针指向用户栈

内核态：堆栈指针指向内核栈

系统调用时进程进入内核态，将用户态的堆栈指针保存在内核栈中，然后修改堆栈指针

8. 查看端口占用

lsof -i查看所有进程　　lsof -i xxx：查看特定进程

netstat -tunlp查看所有tcp、udp端口　　netstat -tunlp | grep 端口号

9. 静态链接和动态链接

1. 静态链接：将所有程序模块链接为一个单独的可执行文件，每个模块在程序中都有一份

   优点：运行速度快

   缺点：占用空间大，更新的话需要对所有用到模块的文件重新编译链接

2. 动态链接：将程序模块划分为单独独立的部分，只有程序运行时才进行链接

   优点：多个模块共享库模块，占用空间小，更新方便

   缺点：执行速度慢（差距很小5%）

10. KILL函数

kill函数的参数 -9 -15

kill -9 xxxx ： 向进程发送SIGKILL信号，强制关闭

kill -15 xxxx：向进程发送SIGTERM信号，进程进行退出准备工作，此时如果遇到阻塞等无法退出，进程选择忽略并终止该信号，则会导致进程无法杀死的现象（还有一种可能是自定了信号处理函数）

linux默认参数kill -15

11. 线程安全问题

1. 线程安全问题是指多线程访问临界资源，可能会出现不一致的问题
2. 解决：使用锁、信号量进行PV操作，原子操作
3. 多线程同步：多线程共享统一资源不受其他线程干扰
4. 改进：使用消息队列模型，多个线程将数据发送至队列中，加锁保证线程安全，然后让一个线程取出消息发送给对端
5. 系统调用都是线程安全的

   printf、malloc、free不是原子性

12. /bin是用户常用指令，/sbin是系统管理员常用的指令

13. 进程最大文件句柄的修改

ulimit -n xxxx：只对当前进程有效

vi /etc/security/limits.conf添加

soft nofile 65536　　hard nofile 65536　　修改后保存重新登陆

14. 并发和并行的区别

1. 并发：宏观上两个程序同时运行，微观上交织运行，提高效率
2. 并行：物理以上的并行，进程运行在多核上，互相不影响，同时执行两条指令，提高了运行效率

15. 操作系统的页表寻址

1. 页式内存管理：内存分成长度相等4K的页片，操作系统维护从虚拟地址到物理地址的映射关系表--页表，页表每一项记录页的基地址，通过基地址+偏移量得到物理地址
2. Linux的两级页表机制--最初

   地址空间分为

   20位两级页表索引 - 12位页内偏移量

   PGD：高10位全局页目录索引　-- CR3寄存器

   PTE：中间10位页表入口索引
3. Linux的三级页表机制

   2位PGD，9位PMD，9位PTE，12位页内偏移

16. 游戏服务器与用户

游戏服务器应给每个用户开辟一个进程：同一时间的线程之间会相互影响，线程死掉会影响其他线程导致进程崩溃，为了不让用户之间相互影响，应为每个用户开辟一个进程

17. 结构体对齐与字节对齐

1. 对齐原因：不是所有硬件平台都能随机访问数据（指操作系统字内部）

   需要对齐地址
2. 对齐规则：
   1. 结构体成员对齐规则：

      第一个成员放在offset=0的地方

之后的按照#pragma pack(n)指定的数值和自身长度取最小对齐

2. 结构体整体对齐规则

   按照#pragma pack指定的数值和结构体最大数据成员长度较小的对齐

```
#pragma pack(2)

struct AA {

int a;      //长度4 > 2 按2对齐；偏移量为0；存放位置区间[0,3]

char b;  //长度1 < 2 按1对齐；偏移量为4；存放位置区间[4]

short c;    //长度2 = 2 按2对齐；偏移量要提升到2的倍数6；存放位置区间[6,7]

char d;  //长度1 < 2 按1对齐；偏移量为7；存放位置区间[8]；共九个字节

};

#pragma pack()
```

3. #pragma pack用法

   1. 默认采用#pragma pack(8)，可取1，2，4，8，16
   2. 使用#pragma pack(show)查看对齐字节数，编译时给出警告

      #pragma pack(push)：会将当前的对齐字节数压入栈顶，并设置这个值为新的对齐方式

      #pragma pack(push n)：会将当前字节数压入栈顶，设置n

      #pragma pack(pop)：弹出栈顶对齐字节数，设置为新的对齐字节

      #pragma pack(pop，n)：弹出栈顶并丢弃，设置n为新的字节数
   3. 对齐后总长度必须为最大对齐参数的整数倍
   4. 不做对其处理#attribute(packed)

18. 互斥锁与读写锁

1. 互斥锁：保证任意时刻只有一个线程访问对象
2. 读写锁：分为读锁和写锁

   读锁：多个进程可以同时获得读锁对同一对象进行读操作，获取失败阻塞等待

   写锁：同一时刻只能由一个对象获得写锁，其他进入睡眠状态等待写锁释放被唤醒

   写锁会阻塞其他锁，且写锁优先级更高（同时到达读写锁，优先执行写锁）

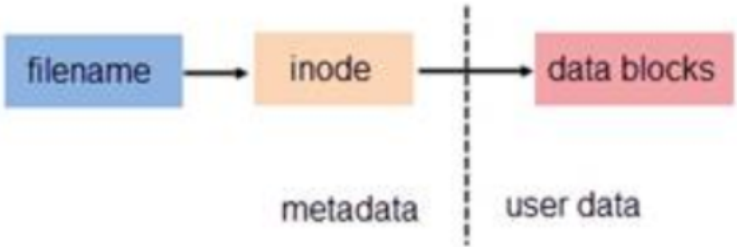
   适用于读频率远大于写频率
3. 互斥锁与读写锁的区别

   1. 读写锁区分读者和写者，互斥锁不区分
   2. 互斥锁同一时间只能由一个线程访问，读写锁可以同时容纳多个读锁

19. Linux的四种锁机制

1. 互斥锁：任意时刻只能有一个线程访问
2. 读写锁：读频率远大于写频率
3. 自旋锁：任意时刻只能有一个线程访问对象，获取锁失败不睡眠而是自选不断请求，适用于加解锁时间短暂，减少线程唤醒的开销
4. RCU：修改数据时：读取数据生成一个副本，对副本进行修改，然后再update生成新的。

20. 硬链接与软连接



1. 硬链接

   多个文件名对应一个inode

   只要有一个连接存在，那么删除链接不会导致文件的删除，只有最后一个链接删除且有新数据存储，才会释放删除的数据块

   只可对文件创建，可以防止误删除

   硬链接：

   - 1. 硬链接，以文件副本的形式存在。但不占用实际空间。
   - 2. 不允许给目录创建硬链接。
   - 3. 硬链接只有在同一个文件系统中才能创建。
   - 4. 删除其中一个硬链接文件并不影响其他有相同 inode 号的文件。

2. 软连接

   软连接视为普通文件，但是inode内容放置同一个路径

   每个文件名都有自己的inode，但是指向相同

   软连接可对文件、目录创建

   软链接：

   - 1.软链接是存放另一个文件的路径的形式存在。
   - 2.软链接可以 跨文件系统 ， 硬链接不可以。
   - 3.软链接可以对一个不存在的文件名进行链接，硬链接必须要有源文件。
   - 4.软链接可以对目录进行链接。

21. 大小端

大端：低字节地址在高字节区域：网络字节序

小端：低字节地址在低字节区域

```
union big_small {
    int a;
    char c;
};

big_small t;
t.a = 1;
if (t.c == 1)
    cout << "small";
else
{
    cout << "big";
}
```

22. 静态变量

1. 静态变量位置：已初始化的存放在.data，未初始化的存放在.bss，由exec初始化

23. 用户态与内核态

二者权限不同，用户态仅拥有最低的特权级别，内核态拥有较高的特权级别，用户态程序不能直接访问内核数据和程序

转换方式：

```
1. 系统调用：主动切换
```

2. 异常：缺页异常
3. 中断：键盘读写完毕后，转到中断处理程序

24. socket

1. 如何设计server舍得能够接受多个客户端

   多线程，多进程，IO多路复用

2. 死循环+连接时新建线程效率低，如何改进

   1. 改进连接时新建线程：使用线程池，使用生产者消费者模型，创建任务队列
   2. 改进死循环+连接：使用select 、poll、epoll

25. 查看线程状态

1. ps命令

参数	作用
-a	显示所有进程（包括其他用户的进程）
-u	用户以及其他详细信息
-x	显示没有控制终端的进程

2. 状态行

   1.R——Runnable（运行）：正在运行或在运行队列中等待

   2.S——sleeping（中断）：休眠中，受阻，在等待某个条件的行程或接收到信号

   3.D——uninterruptible sleep(不可中断)：收到信号不唤醒和不可运行，进程必须等待知道有种段发生

   4.Z——zombie（僵死）：进程已终止，但进程描述还在，直到父进程调用wait4()系统调用后史昂

   5.T——traced or stoppd(停止)：进程收到SiGSTOP,SIGSTP,SIGTOU信号后停止运行

3. 查看线程

   top -H -p xx：查看ppidxx的线程

   ps -T

26. 互斥锁的缺陷--排队加解锁

访问共享资源前对互斥量加锁，访问完成后释放锁，此时其他线程申请锁会被阻塞，如果当前锁阻塞大量线程，释放时都会变成可运行状态，第一个再加锁，其他继续阻塞

27. 两个进程访问临界区可能都会获得自旋锁

单核cpu，并且开了抢占可以造成这种情况。

28. windows的消息机制

1. windows的消息本身是一个结构体，包含消息类型和其他信息

   例如：事件类型号：WM_LBUTTONDOWN，坐标：（TMsg）

2. 当用户有操作时，系统将其转化成消息，每个打开的进程维护一个消息队列。将消息放入消息队列中，进程取出消息完成操作

29. 内存溢出、内存泄露

1. 内存溢出：申请的内存空间超过了系统分配的空间

   原因：内存加载的数据量过大

   内存存在死循环

   函数调用递归导致栈溢出

2. 内存泄露

   1. 内存泄露表示由于疏忽/错误导致申请的动态内存未被释放。内存泄漏并非指内存在物理上的消失，而是应用程序分配某段内存后，由于设计错误，失去了对该段内存的控制，因而造成了内存的浪费。
   2. 分类：

      1. 堆内存泄露：malloc、free没有释放
      2. 系统资源泄露：socket，bitmap没有释放
      3. 对象内存泄漏：虚析构函数

30. 常用的线程模型

1. future模型
2. fork join模型
3. actor模型
4. 生产者消费者

使用一个缓存来保存任务。开启一个/多个线程来生产任务，然后再开启一个/多个来从缓存中取出任务进行处理。这样的好处是任务的生成和处理分隔开，生产者不需要处理任务，只负责向生成任务然后保存到缓存。而消费者只需要从缓存中取出任务进行处理。使用的时候可以根据任务的生成情况和处理情况开启不同的线程来处理。比如，生成的任务速度较快，那么就可以灵活的多开启几个消费者线程进行处理，这样就可以避免任务的处理响应缓慢的问题。

5. master-worker模型

master-worker模型类似于任务分发策略，开启一个master线程接收任务，然后在master中根据任务的具体情况进行分发给其它worker子线程，然后由子线程处理任务。如需返回结果，则worker处理结束之后把处理结果返回给master。

31. 系统调用

1. 系统调用：运行在用户态，向操作系统内核请求更高权限运行的服务，系统调用提供了用户程序和操作系统内核的接口
2. 特权指令：一类只能在内核态进行而不能在用户态执行的指令。若程序在用户态下执行需要访问系统核心功能，这时通过系统调用接口
3. 原因：有些特权指令交给用户执行很危险，使用系统调用包装指令

   在cpu的一些指令中，有的指令如果用错，将会导致整个系统崩溃。分了内核态和用户态后，当用户需要操作这些指令时候，内核为其提供了API，可以通过系统调用陷入内核，让内核去执行这些操作。

32. 源代码到可执行文件的过程

1. 预编译：
   1. 删除#define，展开宏定义
   2. 处理所有预编译指令#ifdef #ifndef #endif
   3. 处理include预编译指令，将文件递归替换到该位置
   4. 删除注释
   5. 保留#pragma等编译指令，#pragma once防止文件重复引用
   6. 删除行号、文件表示
2. 编译：
   1. 词法分析：有限状态机算法，将源码输入扫描机，将字符序列分割为记号
   2. 语法分析：根据扫描器产生的记号，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：对表达式是否有意义进行判断（静态语义）

   源代码级别的优化

   目标代码生成：由代码生成器将中间代码转化为目标机器代码，生成汇编语言表示

   目标代码优化：目标代码优化器对上述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，删除多余指令
3. 汇编：汇编转机器码
4. 链接
5. 执行

33. 宏内核与微内核

1. 宏内核：linux内核

   基本：进程、线程管理、内存管理

   额外：文件管理、驱动、网络协议

   优点：效率高

   缺点：稳定性差，开发时bug会导致系统崩溃
2. 微内核

   只有基本的调度、内存管理

   驱动和文件系统由用户态的守护进程实现

   优点：稳定，驱动的错误只会导致进程死掉

   缺点：效率低

34. 僵尸进程

1. 正常进程：子进程的结束和父进程运行异步，父进程无法预测子进程的结束，进程结束后由父进程wait或waitpid调用取得终止状态
2. unix提供了一种机制可以保证**只要父进程想知道子进程结束时的状态信息， 就可以得到**：在每个进程退出的时候，内核释放该进程所有的资源，包括打开的文件，占用的内存等。 但是仍然为其保留一定的信息，直到父进程通过wait / waitpid来取时才释放。保存信息包括：
    1. 进程号the process ID
    2. 退出状态the termination status of the process
    3. 运行时间the amount of CPU time taken by the process等
3. 每个进程都有僵尸状态存在
4. 如果不进行回收释放进程描述符，那么该进程号一直被占用，可能导致新进程无进程号使用

    外部消灭：SIGTERM或者SIGKILL消灭父进程，那么僵尸进程由init收养

    内部解决：子进程退出时向父进程发送SIGCHILD信号，父进程处理时调用wait

    fork两次，kill父进程，让父进程变为init进程

35. 5种IO模型

1. 阻塞IO：阻塞等待某个函数是否返回：accept
2. 非阻塞IO：设置fcntl的nonblocking的socket，不断检查IO事件是否就绪
3. 信号驱动IO：linux使用套接字驱动IO，IO就绪时收到SIGIO信号
4. IO复用：单个线程实现对多个socket的就绪检查和使用。select或poll实现IO复用模型。使进程阻塞，但是可以阻塞多个IO操作，可以同时对多个读操作、写操作的IO函数进行检测，有数据可读可写时才调用IO操作函数
5. 异步IO：使用aio_read告知内核缓冲区指针大小和通知方式，待拷贝完成后再通知进程

36. Linux内核的Timer定时器机制

1）低精度时钟

Linux 2.6.16之前，内核只支持低精度时钟，内核定时器的工作方式：

1、系统启动后，会读取时钟源设备(RTC, HPET，PIT...)，初始化当前系统时间。

2、内核会根据HZ(系统定时器频率，节拍率)参数值，设置时钟事件设备，启动tick(节拍)中断。HZ表示1秒种产生多少个时钟硬件中断，tick就表示连续两个中断的间隔时间。

3、设置时钟事件设备后，时钟事件设备会定时产生一个tick中断，触发时钟中断处理函数，更新系统时钟,并检测timer wheel，进行超时事件的处理。

在上面工作方式下，Linux 2.6.16 之前，内核软件定时器采用timer wheel多级时间轮的实现机制，维护操作系统的所有定时事件。timer wheel的触发是基于系统tick周期性中断。

所以说这之前，linux只能支持ms级别的时钟，随着时钟源硬件设备的精度提高和软件高精度计时的需求，有了高精度时钟的内核设计。

2）高精度时钟

Linux 2.6.16 ，内核支持了高精度的时钟，内核采用新的定时器hrtimer，其实现逻辑和Linux 2.6.16 之前定时器逻辑区别：

hrtimer采用红黑树进行高精度定时器的管理，而不是时间轮；

高精度时钟定时器不在依赖系统的tick中断，而是基于事件触发。

旧内核的定时器实现依赖于系统定时器硬件定期的tick，基于该tick，内核会扫描timer wheel处理超时事件，会更新jiffies，wall time(墙上时间，现实时间)，process的使用时间等等工作。

新的内核不再会直接支持周期性的tick，新内核定时器框架采用了基于事件触发，而不是以前的周期性触发。新内核实现了hrtimer(high resolution timer)：于事件触发。

hrtimer的工作原理：

通过将高精度时钟硬件的下次中断触发时间设置为红黑树中最早到期的Timer 的时间，时钟到期后从红黑树中得到下一个 Timer 的到期时间，并设置硬件，如此循环反复。

在高精度时钟模式下，操作系统内核仍然需要周期性的tick中断，以便刷新内核的一些任务。hrtimer是基于事件的，不会周期性出发tick中断，所以为了实现周期性的tick中断(dynamic tick)：系统创建了一个模拟 tick 时钟的特殊 hrtimer，将其超时时间设置为一个tick时长，在超时回来后，完成对应的工作，然后再次设置下一个tick的超时时间，以此达到周期性tick中断的需求。

引入了dynamic tick，是为了能够在使用高精度时钟的同时节约能源，这样会产生tickless 情况下，会跳过一些 tick。

新内核对相关的时间硬件设备进行了统一的封装，定义了主要有下面两个结构：

时钟源设备(closk source device)：抽象那些能够提供计时功能的系统硬件，比如 RTC(Real Time Clock)、TSC(Time Stamp Counter)，HPET，ACPI PM-Timer，PIT等。不同时钟源提供的精度不一样，现在pc大都是支持高精度模式(high-resolution mode)也支持低精度模式(low-resolution mode)。

时钟事件设备(clock event device)：系统中可以触发 one-shot（单次）或者周期性中断的设备都可以作为时钟事件设备。

当前内核同时存在新旧timer wheel 和 hrtimer两套timer的实现，内核启动后会进行从低精度模式到高精度时钟模式的切换，hrtimer模拟的tick中断将驱动传统的低精度定时器系统（基于时间轮）和内核进程调度。

38. 判断栈增长的方向

```
void second(int* b);
void first()
{
    int a = 0;
    second(&a);
}
void second(int* b)
{
    int c = 0;
    if (&c < b)
    {
        cout << "向低地址增长";
    }
    else
        cout << "向高地址增长";
}
```

函数栈默认为1M

39. 共享内存的建立

1. shmget和mmap
    1. mmap可以看到文件的实体，而shmget对应的文件在交换分区上的shm文件系统内，无法cat查看

2. 一致性

   mmap方式下，各进程映射文件的相同部分可以共享内存

   shmget时各个进程共享同一片内存区

3. 持续性

   进程挂了重启不丢失内容

   系统挂了重启，mmap可以不丢失内容，shmget会丢失

4. mmap保存到实际硬盘，并没有反映到主存上

   空间大但是速度慢

5. shmget保存到物理存储器，进程间访问速度快但是存储量小

2. mmap是通过操作内存来实现对文件的操作，这样可以加快执行速度，并不是专门用来进行数据通信
3. shm共享内存

   预留出内存空间，允许一组进程对其进行访问

4. 共享内存的api调用

```
int shmget(key_t key,size_t size,int shmflg);
key：共享内存键值，可以理解为共享内存的唯一性标记。
size：共享内存大小
shmflag：创建进程和其他进程的读写权限标识。
返回值：相应的共享内存标识符，失败返回-1
```

```
void *shmat(int shm_id,const void *shm_addr,int shmflg);
shm_id：共享内存标识符
shm_addr：指定共享内存连接到当前进程的地址，通常为0，表示由系统来选择。
shmflg：标志位
返回值：指向共享内存第一个字节的指针，失败返回-1
```

40. Linux文件读写原理

1. 进程调用read/write后陷入内核（系统调用），内核开始读写文件，内核首先把文件读入自己的内核空间，读完后进程在内核回归用户态，内核把读入内核内存的数据再复制进入进程的用户态空间
2. 使用mmap后，将文件映射到一段内存上，通过对内存的读取修改，实现对文件的读取修改
3. shm则是让每个进程最终映射到同一块物理内存

41. 单核、多核、多CPU

1. 单核中进程并发，多CPU中进程并行
2. 单核中线程并发，多核中线程并行

42. 中断

CPU对系统的某一事件做出的一种反应，暂停执行当前程序去中断向量入口地址处取地址进入中断处理程序，处理完毕后返回继续执行。

中断分为三类：

1. CPU外部引起的IO中断、时钟中断
2. CPU内部中断（非法操作，地址越界，浮点溢出）
3. 系统调用引起的、

中断处理分为中断响应和中断处理两个步骤，前者由硬件实施，中断处理由软件实施

43. 访问空指针问题

1. 在C中，定义了NULL表示空指针

   当访问了一个页表中不存在的地址时，触发一个异常，缺页异常由硬件自动触发

2. 用户态进程访问空指针，会收到segmentfault信号，然后进程杀死自己，或者执行对应的信号处理程 果是在内核态的其他上下文中（比如中断上下文），那么系统会执行panic动作。

   （3）如果内核中有配置panic_on_oops，那么上面发生oops的场景也会发生panic。

44. 栈溢出

1. 堆栈溢出是指进程向栈中变量写入的字节超过了变量申请数，导致相邻变量的值被改变
2. 原因：
   1. 函数局部数组过大，增大栈空间，改为用堆分配

      函数栈的大小大约为1M

   2. 递归调用层次太多：压入栈次数太多导致堆栈溢出
   3. 指针或者数组越界

45. 堆、栈

1. 栈的效率高原因：栈由操作系统提供，计算机底层提供了一系列支持，拥有专门寄存器存储，压栈、入栈都有专门指令。堆的机制由C/C++函数库提供。需要一些分配内存、合并内存、释放内存的算法
2. 栈由系统分配和管理，堆由程序员分配和管理
3. 栈从高地址向低地址扩展，堆从低地址向高地址扩展
4.

46. Linux

1. 一切皆文件：所有对象的表现形式都为文件，甚至包含磁盘、进程
2. 目录：
   1. /bin：全程binary，
   2. /dev：主要存放外接设备，不能直接使用，需要进行挂载（类似于windows下分配盘符）
   3. /etc：配置文件
   4. /home：存放用户的文件夹（除了root用户以外的用户目录）
   5.

47. 查看系统日志

1. tail

   tail -n 10 test.log查看日志最后10行

   tail -n +10 test.log查看10行后所有

48. 用户态线程和内核态线程

- 分类：用户级线程、内核级线程
- 用户级线程
  - 当线程在用户空间下实现时，OS对线程的存在一无所知，OS只能看到进程
  - 当一个线程完成了其工作或等待需要被阻塞时，其调用系统过程阻塞自身，然后将CPU交由其它线程

    - 这种模式的好处
        - 在用户空间下进行进程切换的速度 远快于 在操作系统内核中
        - 用户空间下实现线程使得程序员可以实现自己的线程调度算法
        - 由于操作系统不需要知道线程的存在，所以在任何操作系统上都能应用
    - 这种模式的缺点
        - 由于操作系统不知道线程的存在，因此当一个进程中的某一个线程进行系统调用时，比如缺页中断而导致线程阻塞，此时操作系统会阻塞整个进程
        - 造成编程困难，我们在写程序的时候必须仔细斟酌在什么时候应该让出CPU给别的线程使用
        - 假如进程中一个线程长时间不释放CPU，因为用户空间并没有时钟中断机制，会导致此进程中的其它线程得不到CPU而持续等待
- 内核级线程
    - 这种模式下，操作系统知道线程的存在
    - 实现线程造成阻塞的运行时调用(System runtime call)成本会高出很多
    - 当一个线程阻塞时，操作系统可以选择将CPU交给同一进程中的其它线程，或是其它进程中的线程。而在用户空间下实现线程时，调度只能在本进程中执行，直到操作系统剥夺了当前进程的CPU
    - 这种模式的优点
        - 用户编程简单，用户程序员在编程的时候无需关心线程的调度，即无需担心线程什么时候会执行，什么时候会挂起
        - 如果一个线程执行阻塞操作，操作系统可以从容地调度另外一个线程执行。因为操作系统能监控所有线程。
    - 这种模式的缺点
        - 效率较低。因为线程在内核态实现，每次线程切换都需要陷入到内核，由操作系统来进行调度
        - 占用内核稀缺的内存资源
- 混合模式 - 现在OS的实现方式
    - 用户态的执行系统负责进程内部线程在非阻塞时的切换
    - 内核态的操作系统负责阻塞线程的切换
    - 每个内核态线程可以服务一个或多个用户态线程
    - 在这种模式下，操作系统只能看到内核线程

49. mmap、fread、read

1. read：首先fopen一个文件，然后malloc一块内存，最后使用read函数将fp指向的文件读到内存中

   首先判断是否在内核中，如果没有，启动一次文件IO然后读入到内核的cache之中，最后再拷贝到用户的buffer里面

2. mmap：直接将内核态的地址map到用户态上，用户态通过指针直接访问内核中的内容