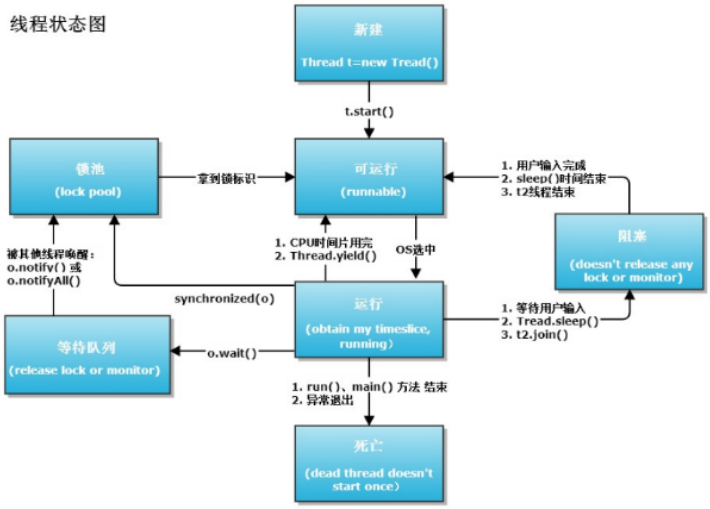
操作系统（1）

并发

- 加锁会导致一下问题：
  - 加多线程竞争下，加锁和释放锁会导致较多的上下文切换，引起性能问题。
  - 多线程可以导致死锁的问题。
  - 多线程持有的锁会导致其他需要此锁的线程挂起。
- ThreadLocal
  - 每个 Thread 维护一个 ThreadLocalMap 映射表，这个映射表的 key 是 ThreadLocal 实例本身，value 是真正需要存储的 Object。ThreadLocalMap 是使用 ThreadLocal 的弱引用作为 Key 的，弱引用的对象在 GC 时会被回收。
  - ThreadLocal 好的使用习惯，是每次使用完 ThreadLocal，都调用它的 remove() 方法，清除数据。
  - 内存泄漏
    - ThreadLocalMap使用ThreadLocal的弱引用作为key，如果一个ThreadLocal没有外部强引用来引用它，那么系统 GC 的时候，这个ThreadLocal势必会被回收，这样一来，ThreadLocalMap中就会出现key为null的Entry，就没有办法访问这些key为null的Entry的value，如果当前线程再迟迟不结束的话，这些key为null的Entry的value就会一直存在。
    - 其实，ThreadLocalMap的设计中已经考虑到这种情况，也加上了一些防护措施：在ThreadLocal的get(),set(),remove()的时候都会清除线程ThreadLocalMap里所有key为null的value。
    - 情景：
      - 使用static的ThreadLocal，延长了ThreadLocal的生命周期，可能导致的内存泄漏。
      - 分配使用了ThreadLocal又不再调用get(),set(),remove()方法，那么就会导致内存泄漏。
  - 为什么不适用强引用
    - key 使用强引用：引用的ThreadLocal的对象被回收了，但是ThreadLocalMap还持有ThreadLocal的强引用，如果没有手动删除，ThreadLocal不会被回收，导致Entry内存泄漏。
  - 如何避免内存泄漏：
    - 每次使用完ThreadLocal，都调用它的remove()方法，清除数据。
- 锁
  - synchronized 关键字
    - moniterenter 和 moniterexit这两个指令划分了一片同步块，具有排他性，当有线程进入该同步块后，其他线程必须等待在 monitereneter 指令上，直到进入同步块的线程通过 moniterexit 指令退出后，其他线程才可以进入同步块。
  - ReentrantLock 重入锁
    - ReentrantLock 提供了显式加解锁操作。提供了 lock(),unlock() 方法进行加解锁的操作，而 synchronized 是隐式进行加锁与解锁操作（依赖于编译器将其编译为 moniterenter 与 moniterexit）。
    - 对锁的等待可以中断，在持有锁的线程长时间不释放锁时，等待锁的线程可以选择放弃等待，这样就避免了 synchronized 可能带来的死锁问题。
      ReentrantLock.tryLock() 可以设置等待时间。
    - ReentrantLock 提供了公平锁与非公平锁，而 synchronized 的实现是非公平锁。
    - lock(), 如果获取了锁立即返回，如果别的线程持有锁，当前线程则一直处于休眠状态，直到获取锁
    - tryLock(), 如果获取了锁立即返回 true，如果别的线程正持有锁，立即返回 false；
- 死锁
  - 在死锁时，线程 / 进程间相互等待资源，而又不释放自身的资源，导致无穷无尽的等待，其结果是任务永远无法执行完成
  - 产生死锁的四个必要条件：
    - 互斥条件：一个资源每次只能被一个线程 / 进程使用。
    - 请求与保持条件：一个线程 / 进程因请求资源而阻塞时，对已获得的资源保持不放。
    - 不剥夺条件：线程 / 进程已获得的资源，在未使用完之前，不能强行剥夺。
    - 循环等待条件：若干线程 / 进程之间形成一种头尾相接的循环等待资源关系。
- Executors框架
  -

线程池	特性	任务队列 (BlockingQueue)	核心线程数	最大线程数	线程保活时间
FixedThreadPool	创建固定线程数线程池	LinkedBlockingQueue()	nThread	无效	0L
SingleThreadPool	单线程任务执行线程池	LinkedBlockingQueue()	1	1	0
CachedThreadPool	动态分配线程数	SynchronousQueue()	0	Integer.MAX_VALUE	60s

    - 线程池的优点：
        - 通过重用已存在的线程，降低线程创建和销毁的消耗
        - 提升系统的响应速度，有任务来时可以直接执行
        - 方便线程并发数的管控，不会无限制的创建线程消耗服务器资源
    - 线程池抛出异常处理办法：
        - 1）自定义线程池并实现 afterExecute 方法
        - 2）给线程池中的每个线程指定一个 UncaughtExceptionHandler, 由 handler 来统一处理异常。
- 线程的状态转换
    - 在 Java 当中，线程通常都有五种状态，创建、就绪、运行、阻塞和死亡。
    - 如何保证线程安全
        - 对非安全的代码进行加锁控制；
        - 使用线程安全的类；
        - 多线程并发情况下，线程共享的变量改为方法级的局部变量。
- Thread 类的 interrupt,interrupted,isInterrupted 方法的区别
    - interrupt标志中断标志位，并不会立即中断线程。等待会抛出 InterruptedException的方法（sleep和wait）。
    - interrupted 会清除中断标志位，interrupted 是作用于当前线程
    - isInterrupted 不会清除中断标志位，sInterrupted 是作用于调用该方法的线程对象所对应的线程
-



- 最大线程数与JVM的关系
    - -Xms 初始堆大小与-Xmx 最大堆大小
        - 当给 JVM 的堆内存分配的越大，系统可创建的线程数量就越少，因为线程占用的是系统空间，所以当 JVM 的堆内存越大，系统本身的内存就越少，自然可生成的线程数量就越少。
    - -Xss 每个线程栈大小
        - 这个理解也很简单，线程可用空间保持不变，每个线程占用的栈内存大小变小，自然可生成的线程数量就越多。
    - 系统本身限制
- synchronized
    - 锁的形式
        - 1、对于普通同步方法，锁是当前实例对象。
        - 2、对于静态同步方法，锁是当前类的 Class 对象。
            - 方法同步则是依靠方法修饰符上的 ACC_SYNCHRONIZED 实现。

- 3、对于同步代码块，锁是 Synchronized 括号中配置的对象.
  - 代码块同步是使用 monitorenter 和 monitorexit 指令实现的。
- 为了减少获得锁和释放锁带来的性能消耗，引入了 “偏向锁” 和 “轻量级锁”：
  - 偏向锁（无任何竞争）
    - 大多数情况下，锁不仅不存在多线程竞争，而且总是由同一线程多次获得，为了让线程获得锁的代价更低引入了偏向锁。
    - 当一个线程访问同步块并获取锁时，会在对象头和栈帧中的锁记录里存储锁偏向的线程 ID，以后该线程在进入和退出同步块时不需要进行 CAS 操作来加锁和解锁。轻量级锁的获取及释放依赖多次 CAS 原子指令，而偏向锁只需要一次 CAS 原子指令来置换线程 ID。
  - 轻量级锁（有竞争，但交替执行）
    - 在多线程交替执行同步块时，尽量避免重量级锁引起的性能消耗，但是如果多个线程在同一时刻进入临界区，会导致轻量级锁膨胀升级为重量级锁。
  - 重量级锁（同时竞争同一把锁）
    - 重量级锁通过对象内部的监视器（monitor）实现，其中 monitor 的本质是依赖于底层操作系统的 Mutex Lock 实现，操作系统实现线程之间的切换需要从用户态切换到内核态，切换成本非常高。
-

锁	优点	缺点	适用场景
偏向锁	加锁和解锁不需要额外的消耗，和执行非同步方法相比仅存在纳米级的差距	如果线程间存在锁竞争，会带来额外的锁撤销的消耗	适用于只有一个线程访问同步块的场景
轻量级锁	竞争的线程不会阻塞，提高了程序的响应速度	如果始终得不到锁竞争的线程，使用自旋会消耗 CPU	追求响应时间，同步块执行速度非常快
重量级锁	线程竞争不使用自旋，不会消耗 CPU	线程阻塞，响应时间缓慢	追求吞吐量，同步块执行速度较长

- LINUX 内存分配
  - glibc库实现了malloc，它实现linux系统的堆管理。在Linux中，大部分的系统调用都是通过C库函数来体现了，因此glibc就显得尤为重要。glibc的实现策略和Windows的类似，都是维护一个全局链表，每个链表元素由不定长的内存块链表。
  - 与Windows不同的是，在glibc中，维护了多个不定长的内存块链表，每一个链表负责一个大小范围，这种做法有效减少了分配大内存时的遍历开销，类似于哈希的方式，将很大的范围的数据散列到有限的几个小的范围内而不是所有数据都放在一起，虽然最终还是要在小的范围内查找，但是最起码省去了很多的开销，如果只有一个不定长链表那么就要全部遍历，如果分成3个，就省去了2/3的开销，总之这个策略十分类似于散列。glibc另外的策略就是不止维护一类空闲链表，而是另外再维护一个缓冲链表和一个高速缓冲链表，在分配的时候首先在高速缓存中查找，失败之后再在空闲链表查找，如果找到的内存块比较大，那么将切割之后的剩余内存块插入到缓存链表，如

果空闲链表查找失败那么就往缓存链表中查找. 如果还是没有合适的空闲块，就向内存申请比请求数更大的内存块，然后把剩下的内存放入链表中。这种方式是glibc自己实现的策略。

- malloc函数，它内部有一个将多个可用内存块连接为一个空闲链表。在调用时，它沿链表寻找一个大到足以满足用户请求所需要的内存块。然后，将该内存块一分为二（一块的大小与用户请求的大小相等，另一块的大小就是剩下的字节）。接下来，将分配给用户的那块内存传给用户，并将剩下的那块（如果有的话）返回到链接表上。
- 调用free函数时，它将用户释放的内存块回收到空闲链表。到最后，空闲链会被切成很多的小内存片段，如果这时用户申请一个大的内存片段，那么空闲链上可能没有可以满足用户要求的片段了。于是，malloc函数请求延时，并开始在空闲链上翻箱倒柜地检查各内存片段，对它们进行整理，将相邻的小空闲块合并成较大的内存块。

- 三组I/O复用函数的比较
  - （1）事件集：
    - Select：Select的参数类型fd_set没有将文件描述符和事件进行绑定，它仅仅是一个文件描述符的集合，因此select的参数需要提供3个这种类型的参数来分别传入和输出可读、可写及异常事件。这一方面使得select不能处理更多类型的事件，另一方面由于内核对fd_set集合的在线修改，应用程序下次调用select前不得不重置3个fd_set集合。
    - Poll：poll的参数pollfd，它把文件描述符和事件都定义在其中，任何事件都被统一处理，从而使编程接口简单的多，并且内核每次修改的是pollfd结构体的revents成员，而events 成员保持不变，因此下次调用poll时应用程序无需重置pollfd类型的事件集合。
    - Epoll：epoll采用完全不同与select和poll的方式来管理用户注册的事件。它在内核中维护一张事件表，并提供一个独立的系统调用epoll_ctl来控制往其中添加、删除、修改事件。这样，每次epoll_wait调用都直接从内核事件表中取得用户注册的事件，而无需反复从用户空间读入这些事件。
  - （2）最大支持文件描述符数：
    - Select：select打开的文件描述符fd是由一定限制的，有 fd_size设置，默认为2048,对于那些需要支持上万连接数目的服务器来说显然太少了。
    - Poll和epoll：poll和epoll分别用nfds和maxevents参数指定最多监听多少个文件描述符和事件，这两个值都能达到系统允许打开的最大文件描述符数目，即65535。
  - （3）工作模式：
    - Select和poll都只能工作在相对低效的LT模式，而epoll则可以工作在ET高效模式。
  - （4）实现原理：
    - Select和poll采用的都是轮询的方式，即每次调用都要扫描整个注册文件描述符集合，并将其中就绪的文件描述符返回给用户，因此他们检测就绪事件的算法时间复杂度是O(n)。
    - Epoll的epoll_wait则不同，它采用的是回调函数的方式。内核检测到就绪的文件描述符时，将触发回调函数，回调函数就将该文件描述符上对应的时间插入内核就绪事件队列。内核最后在适当的时候将就绪事件队列中的内容拷贝到用户空间。因此epoll_wait无需轮询整个文件描述符集合来检测那些事件已经准备就绪，其算法事件复杂度O(1)。但是，当活动链接比较多时，epoll_wait的效率未必比select和poll高，因为回调函数被触发得过于频繁。所以epoll_wait适用于连接数量多，但是活动连接少的情况。
  - (5)使用mmap加速内核与用户空间的消息传递。

    - 这点实际上涉及到epoll的具体实现了。无论是select,poll还是epoll都需要内核把FD消息通知给用户空间，如何避免不必要的内存拷贝就很重要，在这点上，epoll是通过内核于用户空间mmap同一块内存实现的。

- 用户级线程与内核级线程比较
    - 用户级线程不需要陷入内核，因此切换速度很快
    - 用户级线程允许每个进程有自己定制的调度算法
    - 用户级线程跨平台很方便
    - 用户级线程问题之一是如何现实阻塞
    - 用户级线程问题之二是没有时钟如何调度
- 多进程程序与多线程程序区别，优缺点，使用场合
    - 多进程程序，一个进程崩溃不会影响其他进程，但是进程之间的切换和通信代价较大；
    - 多线程程序，一个线程崩溃会导致整个进程死掉，其他线程也不能正常工作，但是线程之前数据共享和通信更加方便。
    - 进程需要开辟独立的地址空间，多进程对资源的消耗很大，而线程则是“轻量级进程”，对资源的消耗更小，对于大并发的情况，只有线程加上IO复用技术才能适应。
    - 对于需要频繁交互数据的，频繁的对同一个对象进行不同的处理，选择多线程合适，对于一些并发编程，不需要很多数据交互的采用多进程。
- 有了进程为什么还要线程
    - 1. 一个任务可以分成多个子任务并行执行，他们是对一个对象在操作。
    - 2. 线程不需要像进程一样维护那么多信息，因此创建和销毁速度更快，拥有同一个地址空间，访问很容易
    - 3. 任务有CPU密集和IO等待，的过程，最大化利用CPU
- 进程之间的通信方式
    - 数据传递，关键部位不会交叉，顺序
    - 信号量：信号量用于实现进程间的互斥与同步，而不是用于存储进程间通信数据。
    - 管道(pipe)：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。
    - 命名管道 (named pipe)： 有名管道也是半双工的通信方式，但是它允许无亲缘关系进程间的通信。
    - 消息队列(message queue)： 消息队列是由消息的链表，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。
    - 信号 (sinal)： 信号是一种比较复杂的通信方式，用于通知接收进程某个事件已经发生。
    - 共享内存(shared memory)：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。它往往与其他通信机制，如信号两，配合使用，来实现进程间的同步和通信。
    - 套接字(socket)： 套解口也是一种进程间通信机制，与其他通信机制不同的是，它可用于不同机器间的进程通信。
- 线程的基本概念、线程的基本状态及状态之间的关系
    - 线程被称为轻量级进程，除了没有独立的地址空间管理资源，其余都与进程相同。进程与线程的区别前面已经阐述。

- 线程与进程相同，都具有三个基本状态：运行、阻塞、和就绪，状态之间有四种转换关系：运行–> 阻塞；阻塞–>就绪；就绪–>运行；运行–>就绪；只有运行与就绪可以相互转换。
- 而阻塞一定要经过就绪态，我的理解是，如果没有就绪态，那么操作系统在调度的时候就不知道哪些是已经阻塞完成哪些还在阻塞，那等待IO来说，如果没有就绪态，操作系统就不知道哪些IO上已经有了输入，哪些还需要继续等待。

- 多线程有几种实现方法，都是什么?
  - 1.调用系统的API，windows下是CreateThread，_beginThread（不要用），_beginThreadex（推荐使用），Linux使用POSIX线程
  - 2.使用第三方库的多线程函数，比如Boost库，实现用户级线程，MFC库中的afxBeginThread实际是调用系统API
  - 3.使用C/C++库里的函数，包含头文件中，thread函数创建线程（用户级线程）
- 多线程同步和互斥有几种实现方法，都是什么?
  - 信号量：对应一个down和up操作，down使信号量减1，up使信号量加1，如果信号量大于0，则down后继续执行，如果down等于0，则down后睡眠，但是并不会将信号量减到负数。down和up都是原子操作
  - 互斥量：互斥量是信号量的一种特例，他只有0和1两种状态（解锁和加锁）。
  - 关键区域：关键区域与互斥量类似，但是最大的区别在于，关键区域会进行忙等待，而互斥量如果不能解锁会自动让出CPU
  - Windows平台下
    - 关键区域（Critical Section）：关键节不是内核对象，在用户态实现了同一进程中线程的互斥。由于使用时不需要从用户态切换到核心态，所以速度很快（X86系统上约为20个指令周期），但其缺点是不能跨进程同步，同时不能指定阻塞时的等待时间，只能无限等待。
    - 互斥体（Mutex）：互斥体实现了和关键节类似的互斥功能，但区别在于：互斥体是内核对象，可以实现跨进程互斥，但需要在用户态和核心态之间切换，速度比关键节慢得多（X86系统上约为600个指令周期），同时可以指定阻塞时的等待时间。
    - 事件（Event）：事件也是内核对象，具有“信号态”和“无信号态”两种状态。当某一线程等待一个事件时，如果事件为信号态，将继续执行，如果事件为无信号态，那么线程被阻塞。线程能够指定阻塞时的等待时间。
    - 信号量（Semaphore）：信号量是一个资源计数器，当某线程获取某信号量时，信号量计数首先减1，如果计数小于0，那么该线程被阻塞；当某线程释放某信号量时，信号量计数首先加1，如果计数小于或等与0，那么唤醒某被阻塞的线程并执行之。
  - Linux平台下（加锁）
    - 互斥锁
    - 读写锁
    - 自旋锁
    - 屏障
-

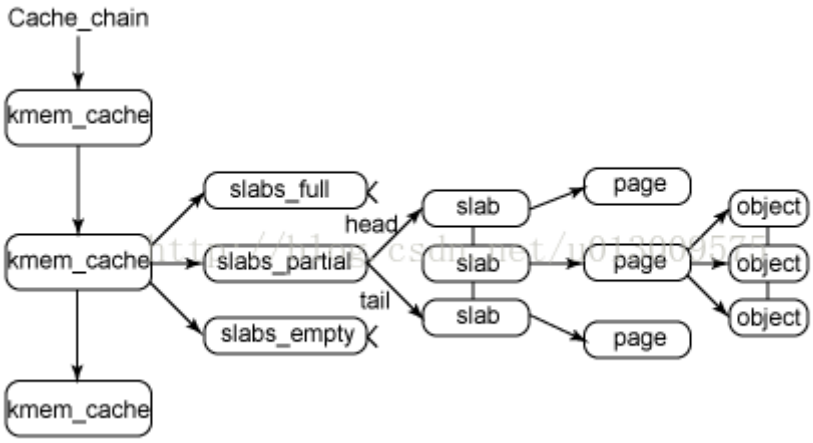
进程和线程(POSIX)的系统调用

进程原语	线程原语	描述
fork	pthread_create	创建线程或进程
waitpid	pthread_join	获取线程进程或线程退出状态
getpid	pthread_self	获取进程ID或线程ID
exit	pthread_exit	退出进程或线程

- 线程池是什么?
  - 线程池多用于并发服务的情况下，一般的并发是新增一个连接就为新的连接开一个线程为其服务，但是开启线程和服务完毕后关闭线程都需要耗费一定的时间。因此，为了提高效率，预先创建一定数量的线程，并让线程处于阻塞状态，当新来了一个连接就从线程池中挑一个线程为其服务，服务完毕后线程也不关闭，重新放回池子。线程池就是这样采用一个队列或其他容器，维持一定数量线程。
- 多线程同步和互斥有何异同，在什么情况下分别使用他们？举例说明。
  - 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，如“第一类读写者模型”。
  - 火车票售票是互斥还是同步？ 互斥
  - 生产者消费者模式是主要是同步问题，也有互斥的问题。
- 内存管理
  - 1.小块内存单独分配,大块内存有系统自动分配.(nginx和stl就是使用这种方法)
    - 内存池的实现原理
      - 内存池的先调用malloc函数申请一大块内存，然后维护一个空闲链表，该链表是一个个小的空闲内存片，没当需要内存时就从空闲链表上拿过来一个小片内存使用。如果空闲链表为空了，就从之前分配的大块内存去取几个插入到空闲链表上。如果分配的大块内存也用光了，就继续用malloc申请一大块。
      - 实现
        - 在学习STL源码时，了解了allocator类，于是打算自己实现一个简单的内存池
        - 对于用户主要使用两个接口：
          - newElement();newElement(const U& args);
          - 该函数用于创建一个对象，并调用构造函数，最后返回对象指针。有带参数版本和无参数版本。
          - void deleteElement(pointer p);
          - 该函数先调用析构函数，然后回收内存到内存池;
  - 2.伙伴算法.
    - 1.将空闲页面分为m个组,第1组存储2^0个单位的内存块,,第2组存储2^1个单位的内存块,第3组存储2^2个单位的内存块,第4组存储2^3个单位的内存块,以此类推.直到m组.
    - 2.每个组是一个链表,用于连接同等大小的内存块.

- 3.伙伴块的大小是相等的,并且第1块和第2块是伙伴,第三块和第四块是伙伴.以此类推.
- 伙伴算法分配内存:
  - 若申请的内存大小为n则将n向上取整为2的幂设次数为s,则需要分配s大小的内存块,定位大相应数组,
  - 1.如果该数组有剩余内存块,则分配出去.
  - 2.若没有剩余内存块就沿数组向上查找,然后再将该内存块分割出来s并将剩余的内存块放入相应大小的数组中.
  - 例如分配5大小的内存块
  - ---------->定位到大小为8的链表中 -------->若该链表中之中没有空余元素,则定位到16的链表中,16中有剩余元素,则取出该元素,并分割出大小为8的内存块供用户使用,然后将剩余的8连接到大小为8的数组中.
- 伙伴算法的内存合并:
  - 当用户用完内存后会归还,然后根据该内存块实际大小(向上取整为2的幂)归入链表中,在归入之前,
  - 1.我们还要检测他的伙伴内存块是否空闲,
  - 2.如果空闲就合并在一起,合并后转到1,继续执行.
  - 3.若果不是空闲的就直接归入链表中.
  - 一般来说,伙伴算法实现中会用位图记录内存块是否被使用,用于伙伴内存的合并.

- 3.slab算法.
  -

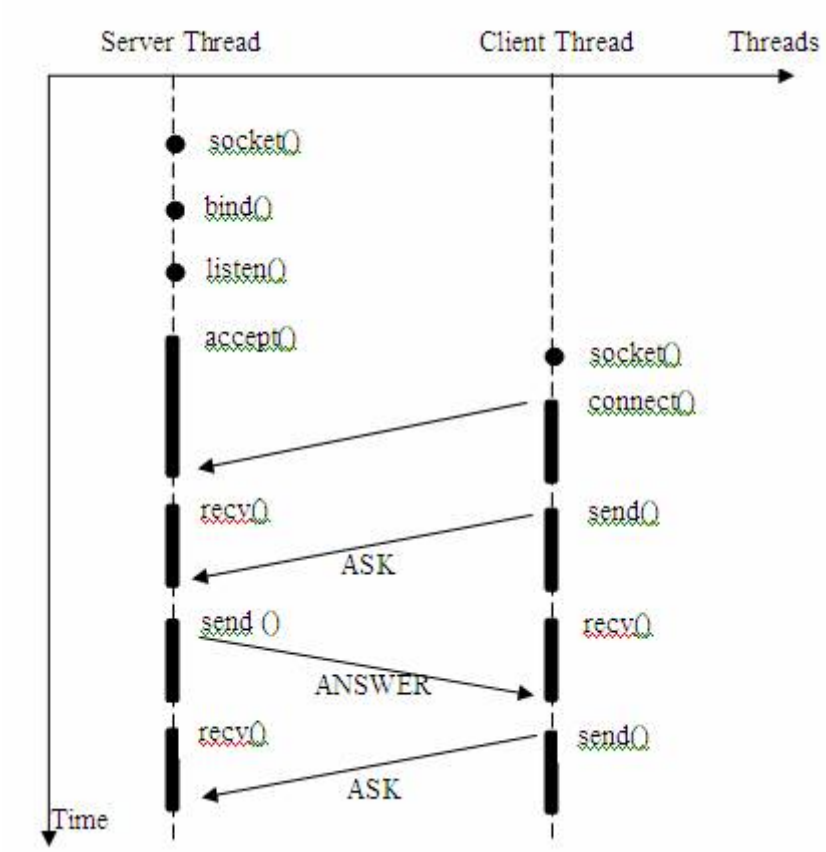


- 高端内存的最基本思想
  - 借一段地址空间，建立临时地址映射，用完后释放，达到这段地址空间可以循环使用，访问所有物理内存。
- 用户空间（进程）是否有高端内存概念
  - 用户进程没有高端内存概念。只有在内核空间才存在高端内存。用户进程最多只可以访问3G物理内存，而内核进程可以访问所有物理内存。
- 用户进程能访问多少物理内存？内核代码能访问多少物理内存？
  - 32位系统用户进程最大可以访问3GB，内核代码可以访问所有物理内存。
  - 64位系统用户进程最大可以访问超过512GB，内核代码可以访问所有物理内存。

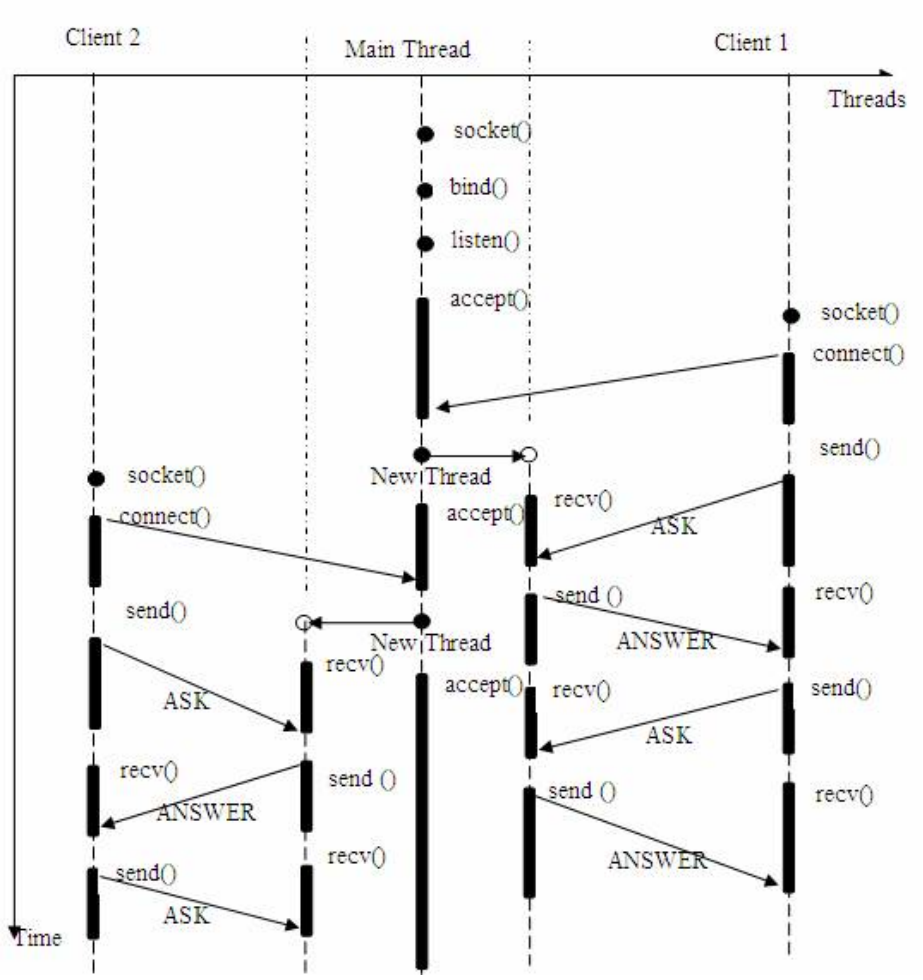
- 进程调度
  - Linux进程分为两种，实时进程和非实时进程；

- 优先级分为静态优先级和动态优先级，优先级的范围；
- 调度策略
  - FIFO，LRU，时间片轮转
- 交互进程通过平均睡眠时间而被奖励；

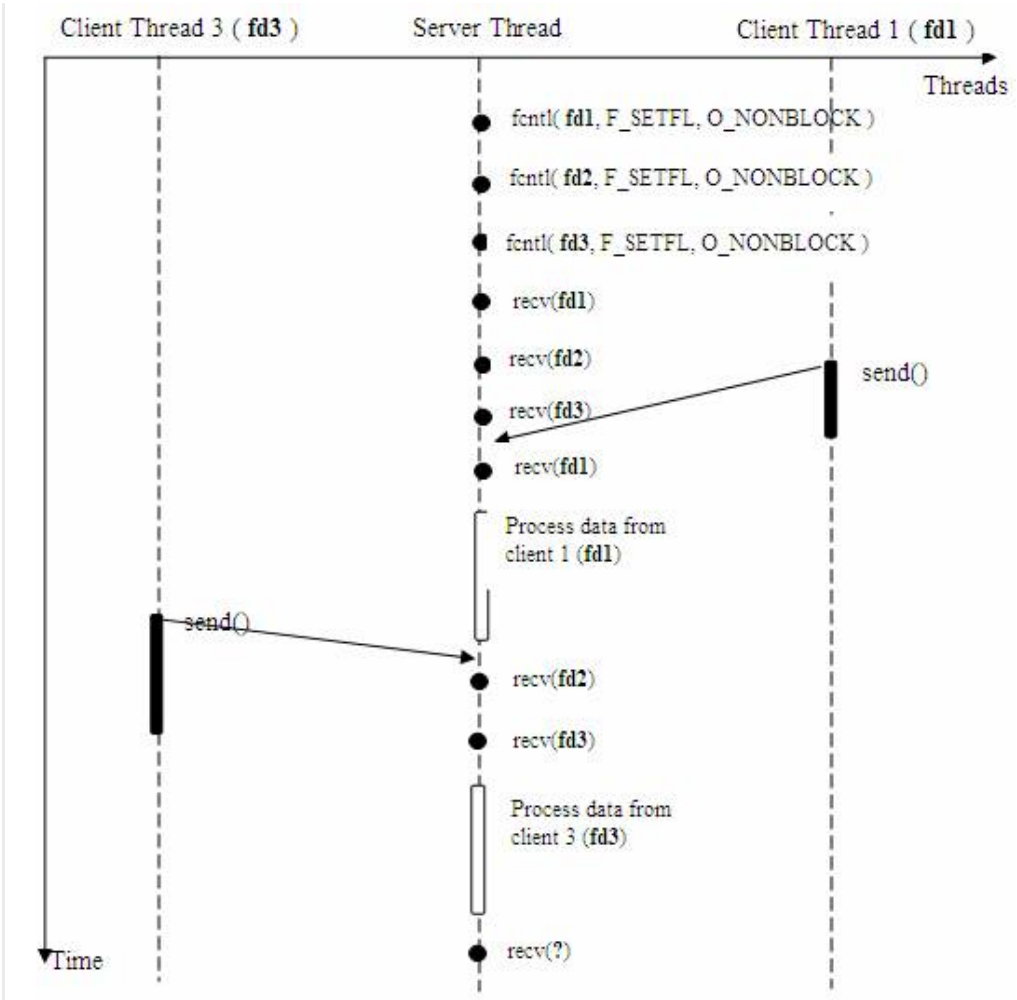
- 命令行
  - Linux命令 在一个文件中，倒序打印第二行前100个大写字母
    - cat filename | head -n 2 | tail -n 1 | grep '[[:upper:]]' -o | tr -d '\n'| cut -c 1-100 | rev
  - 与CPU，内存，磁盘相关的命令(top，free, df, fdisk)
  - 网络相关的命令netstat，tcpdump等
  - sed, awk, grep三个超强大的命名，分别用与格式化修改，统计，和正则查找
  - ipcs和ipcrm命令
  - 查找当前目录以及字母下以.c结尾的文件，且文件中包含”hello world”的文件的路径
  - 创建定时任务
- IO模型
  - 五种IO模型：阻塞IO，非阻塞IO，IO复用，信号驱动式IO，异步IO
  - select，poll，epoll的区别
    - select：
      - 是最初解决IO阻塞问题的方法。用结构体fd_set来告诉内核监听多个文件描述符，该结构体被称为描述符集。由数组来维持哪些描述符被置位了。对结构体的操作封装在三个宏定义中。通过轮寻来查找是否有描述符要被处理，如果没有返回**
      - 存在的问题：
        - 1. 内置数组的形式使得select的最大文件数受限与FD_SIZE；
        - 2. 每次调用select前都要重新初始化描述符集，将fd从用户态拷贝到内核态，每次调用select后，都需要将fd从内核态拷贝到用户态；
        - 3. 轮寻排查当文件描述符个数很多时，效率很低；
    - poll：
      - 通过一个可变长度的数组解决了select文件描述符受限的问题。数组中元素是结构体，该结构体保存描述符的信息，每增加一个文件描述符就向数组中加入一个结构体，结构体只需要拷贝一次到内核态。poll解决了select重复初始化的问题。轮寻排查的问题未解决。**
    - epoll：
      - 轮寻排查所有文件描述符的效率不高，使服务器并发能力受限。因此，epoll采用只返回状态发生变化的文件描述符，便解决了轮寻的瓶颈。
    - - 为什么使用IO多路复用，最主要的原因是什么?
    - - epoll有两种触发模式？这两种触发模式有什么区别？编程的时候有什么区别?
    - - 上一题中编程的时候有什么区别，是在边缘触发的时候要把套接字中的数据读干净，那么当有多个套接字时，在读的套接字一直不停的有数据到达，如何保证其他套接字不被饿死(面试网易游戏的时候问的一个问题，答不上来，印象贼深刻)。
  - 阻塞IO
    -



- 具体使用多进程还是多线程，并没有一个特定的模式。传统意义上，进程的开销要远远大于线程，所以，如果需要同时为较多的客户机提供服务，则不推荐使用多进程；如果单个服务执行体需要消耗较多的 CPU 资源，譬如需要进行大规模或长时间的数据运算或文件访问，则进程较为安全。通常，使用 pthread_create () 创建新线程，fork() 创建新进程。
  -


Client 2
Main Thread
Client 1
Threads
socket()
bind()
listen()
accept()
socket()
connect()
send()
socket()
New Thread
recv()
connect()
accept()
ASK
send ()
recv()
send()
New Thread
ANSWER
recv()
accept()
recv()
send()
ASK
ASK
recv()
send ()
send ()
recv()
ANSWER
ANSWER
recv()
send()
ASK
Time

- 非阻塞IO
  -



- 共享内存的实现原理?
  - 共享内存实现分为两种方式一种是采用mmap，另一种是采用XSI机制中的共享内存方法。mmap是内存文件映射，将一个文件映射到进程的地址空间，用户进程的地址空间的管理是通过vm_area_struct结构体进行管理的。mmap通过映射一个相同的文件到两个不同的进程，就能实现这两个进程的通信，采用该方法可以实现任意进程之间的通信。mmap也可以采用匿名映射，不指定映射的文件，但是只能在父子进程间通信。XSI的内存共享实际上也是通过映射文件实现，只是其映射的是一种特殊文件系统下的文件，该文件是不能通过read和write访问的。
- ++i是否是原子操作
  - 明显不是，++i主要有三个步骤，把数据从内存放在寄存器上，在寄存器上进行自增，把数据从寄存器拷贝会内存，每个步骤都可能被中断
- 判断大小端
  - union un { int i; char ch; };
  - void fun() {
  - union un test;
  - test.i = 1;
  - if(ch == 1)
  - cout << "小端" << endl;
  - else
  - cout << "大端" << endl; }
- 负载均衡器
  - nginx

- HAporxy
- LVS

- select、poll、epoll区别
-

1、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

2、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。

3、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。

- 综上，在选择select，poll，epoll时要根据具体的使用场合以及这三种方式的自身特点：
- 1、表面上看epoll的性能最好，但是在连接数少并且连接都十分活跃的情况下，select和poll的性能可能比epoll好，毕竟epoll的通知机制需要很多函数回调。
- 2、select低效是因为每次它都需要轮询。但低效也是相对的，视情况而定，也可通过良好的设计改善。