何种场合需要创建新的线程，线程如何创建和等待，线程和进程的关系，线程局部存储(TLS或者叫thread local),多线程访问资源产生竞态的原因和解决方案，线程同步原语。

对于linux平台:需要熟悉mutex,semaphore, condition\_variable,read-write-lock,自旋锁

进程与线程：

进程与线程区别： 1)进程是资源分配的基本单位，线程是cpu调度或者说是程序执行的最小单位，实现并发功能的单位是线程；2)进程有独立的地址空间,比如早linux下面包括代码段，堆栈段和数据段。而运行一个进程中的线程,它们之间共享大部分数据，共享相同的地址空间(共享代码段(代码和常量)，数据段(全局变量和静态变量)，堆)。但是每个线程有自己的栈(存放局部变量和临时变量)因此启动一个线程，切换一个线程比进程快，花费小。3）线程间通信方便。同一进程下的线程共享数据；而进程间的通信只能通过进程通信的方式进行

安全性(线程不安全),健壮性，性能3方面比较：

安全性：相比于线程，进程之间是不共享资源和地址空间的，所以不会存在太多的安全问题。由于多个线程共享着相同的地址空间和资源，所以有可能恶意修改或者获取非授权数据的可能。

健壮性：由于多个线程共享同一个进程的地址空间和相关的资源，所以当一个线程出现crash，那么可能会导致相应的地址空间和资源会出现问题。从而导致其它的线程也crash。(浏览器使用多进程架构：打开浏览器时，当一个tab页突然崩溃时，所有的tab都会奔溃，这时通常浏览器要重启，重启进程，重新生成线程)而多进程不存在这个问题，因为不同的地址空间和资源，当一个进程崩溃时，并不会影响到其他进程。

性能：进程的安全性，健壮性是建立在独立的地址空间和独立的资源的条件下的，所以进程的启动，关闭，切换相比于线程会有更多的开销。

线程安全：

如果多线程的程序运行结果是可预期的，而且与单线程的程序运行结果一样，那么说明是“线程安全”的

如何保证线程安全？

互斥同步(加锁)；无需同步方案：可重入代码(可允许多个进程访问的代码，因此不允许任何进程对它进行修改。如果要修改，需要把修改的部分拷贝到自己的数据区，并不去改变共享的代码)；线程本地存储(TLS)；原子变量(无锁编程)

线程的3个基本状态：执行，就绪，阻塞，中止

线程两个基本类型：用户级线程；系统级线程（操作系统内核进行管理）

多线程同步与互斥实现：互斥锁，读写锁boost::shared\_lock，原子操作，信号量，自旋锁spin\_lock，条件锁

回收线程

线程函数执行完毕推出，或以其他非常方式终止，线程进入终止态，但是为线程分配的资源不一定释放，可能在系统重启前，一直不能释放。

thread.join() thread.detach()

线程是可结合(joinable)或者可分离的(detached),一个可结合的线程能够被其它线程回收资源和杀死，在被其它线程回收之前,它的资源比如栈，是不可释放的；相反，一个分离的线程是不能被其它线程回收或者杀死的，它的栈在它终止时由系统自动释放

一个进程,包括代码段(可执行代码段，只读常量)，BSS段(未初始化的静态数据，未初始化的全局数据)，数据段(初始化的静态数据，初始化的全局数据)，栈和堆；fork()函数通过系统调用创建一个与原来进程，也就是2个进程可以做完全相同代码段，但如果初始参数或者传入的变量不同,两个进程也可以做不同的事。一个进程调用fork()函数后，子进程完全复制了父进程的地址空间的内容:包括堆栈段和数据段的内容(静态存储区和动态存储区:全局变量，静态变量，堆栈)。但是子进程并没有复制代码段，代码段是只读的，不存在被修改的问题。

pid\_t fork(); 返回2个值(两次)。子进程返回0；父进程返回子进程ID。出现错误，返回负值。Linux的fork采用写时拷贝，只有在需要写入时，数据才会被复制。从而使各个进程拥有各自的拷贝。当fork()后调用exec()就无须复制了。

exec(): 一个进程一旦执行exec类函数，它本身就死亡了。系统把代码段替换成新的程序的代码，废弃原有的数据段和堆栈段，并为新程序分配新的数据段和堆栈段，唯一留下的，就是进程号(还是同一个进程，不过是另一个程序了)

协程：单个线程中的多个用户态线程。实现方式：利用汇编代码切换上下文。原理：通过保存和恢复寄存器的状态。来进行各协程上下文的保存和切换。

协程相较于函数和线程的优点：

相比于函数：协程避免了传统的函数调用栈，几乎可以无限地递归。

相比于线程：协程没有内核态地上下文切换(用户态切换)，近乎可以无限并发。把异步操作转换为同步操作意味着不需要加锁避免了加锁过程中不必要地开销。

进程，线程和协程地设计都是为了并发任务可以更好地利用CPU资源，它们之间最大地区别在于CPU资源地使用上；

进程调度时上下文切换：

os夺取CPU使用权；保存当前用户进程的上下文；调用调度函数，找到下一个占用CPU的进程；恢复下一个进程的上下文；将CPU交回给待继续的进程。

创建协程：

分配栈空间：协程执行起来就像线程一样，需要有堆栈来实现函数调用。线程的堆栈是由操作系统分配的；协程由于工作在用户态，因此由程序员写代码分配。一般用malloc。

CPU寄存器保存区：

当切换协程时，需要切换函数的上下文。切换上下文称为保护现场和恢复现场。所谓的现场就是必要的CPU寄存器值，这些寄存器里包含了协程的堆栈(栈指针指向栈顶，栈基址指针，数据寄存器，程序运行的下一个指令地址)。

进程和线程地任务调度是由内核控制的，是抢占式的；

协程的任务调度是在用户态完成，需要代码里显示地将CPU交给其他协程，是协作式地；

由于可以在用户态调度协程任务，所以可以把一组相互依赖地任务设计为协程。这样，当一个协程任务完成后，可以手动地进行任务切换，把当前任务挂起(yield)，切换到另一个协程区工作。由于可以控制程序主动让出资源，很多情况下不需要对资源进行加锁。

fork和vfork的区别：

fork()的子进程拷贝父进程的数据段；vfork()的子进程与父进程共享数据段

fork()的父子进程的执行次序不确定；vfork()保证子进程先运行，在调用exec或exit之前与父进程数据是共享的，在它调用exec或exit之后父进程才可能被调度运行；当需要改变共享数据段中变量的值，则拷贝父进程。

Linux创建线程：从内核角度讲，Linux没有线程这个概念。Linux把所有线程当作进程。线程仅仅被视为一个与其他进程共享某些资源的进程。

子进程通过父进程创建，子进程的结束和父进程的运行是一个异步过程:即父进程永远无法预测子进程到底什么时候结束。于是产生了孤儿进程和僵尸进程。

孤儿进程:指一个父进程退出后,它的一个或多个子进程还在运行，那么这些子进程成为孤儿进程。孤儿进程将被init进程(进程号为1)所收养,并由init进程对他们完成状态收集工作。

僵尸进程：子进程退出，而父进程并没有调用wait或waitpid获取子进程的状态信息。那么子进程的进程描述符任然保存在系统中，这种进程称为僵尸进程。

避免产生僵尸进程：当一个进程终止后，它的父进程需要调用wait()或者waitpid()系统调用取得子进程的终止状态。当子进程退出后，内核给父进程发送一个sigchild信号，可以在父进程捕获sigchild信号的信号处理函数中调用wait或者waitpid,就可以清理退出的子进程达到防止僵尸进程。

孤儿进程和僵尸进程的区别:孤儿进程是父进程已退出，而子进程未退出；僵尸进程是父进程未退出，而子进程已退出。

Init进程会周期性地调用wait系统调用清除各个僵尸子进程

进程采用exit()退出的时候，OS系统会进行一系列的处理工作，包括关闭打开的FD，占用的内存等。但是，OS也会为该进程保留少量的信息，比如进程id号等信息，因而占用了系统的资源。在一种极端的情况下，当僵尸进程过多的时候，占用了大量的进程id，系统将无法产生新的进程，相当于系统的资源被耗尽。

守护进程:

在linux的系统引导时会开启很多服务，这些服务叫做守护进程(特点:脱离于终端并且在后台运行的进程；守护进程脱离终端是为了避免进程在执行过程中的信息在任何终端上显示并且进程也不会被任何终端所产生的终端信息所打断)；用处：周期性地执行某种人物或等待处理某些发生地事。守护进程常常在系统引导装入时启动，在系统关闭时终止。Linux系统大多数服务通过守护进程实现，同时守护进程还能完成许多系统任务(作业规划进程crond(定时任务服务:定期执行某些命令或者程序),打印进程lqd; d:daemon); 每一个从终端开始运行地进程都会依附于这个终端，这个终端就称为这些进程地控制终端，当控制终端被关闭时，相应地进程都会自动关闭。但是守护进程却能突破这种限制，它从被执行时开始运转，直到整个系统关闭时才退出。

创建守护进程：1)创建子进程，父进程退出2)在子进程中创建新会话3)改变当前目录为根目录4)重设文件权限掩码5)关闭文件描述符

给进程设置僵尸状态的目的是维护子进程的信息，以便父进程在以后某个时间获取，这些信息包括子进程的进程ID，终止状态以及资源利用信息(cpu时间，内存使用量)

进程通信方法：管道，消息队列，共享内存，信号量,socket，信号，文件锁

管道也是一种文件，这个文件只存在于内存中。通过管道通信的两个进程，一个进程向管道写数据，另外一个进程从中读数据。写入的数据每次都添加到管道缓冲区地末尾，读数据的时候都是从缓冲区地头部读出数据。

管道：1)匿名管道(无名管道)：在内核中申请一块固定大小的缓冲区，程序拥有写入和读取的权利，一般使用fork函数实现父子进程的通信；2）命名管道(named pipe或FIFO):它不同于无名管道之处在于它提供一个路径名与之关联,以FIFO地文件形式存在于文件系统中。这样，即使与FIFO的创建进程不存在亲缘关系的进程，只要可以访问该路径，就能彼此通过FIFO相互通信。因此，通过FIFO不相关的进程也能交换数据。在内核中申请一块固定大小的缓冲区，程序拥有写入和读取的权利，没有血缘关系的进程也可以进程间通信。

有名管道特点:

1)使互不相关的两个进程间实现彼此通信；2)该管道可以通过路径名来指出,并且在文件系统中是可见的。在建立了管道之后，两个进程就可以把它当作普通文件一样进行读写操作。3)FIFO严格遵循先进先出规则，对管道及FIFO的读操作总是从开始处返回数据，对它们的写操作则是把数据添加到末尾。

Int mkfifo(const char\* pathname, mode\_t mode);

共享内存：允许两个不相关的进程访问同一个逻辑内存，共享内存是两个正在运行的进程之间共享和传递数据的一种非常有效的方式。共享内存没有提供同步机制(缺点)，需要自己提供，一般和信号量配合用。共享内存是最有用的进程间通信方式，因为不需要数据来回复制，所以是最快的IPC方式

原理：不同进程之间共享的内存通常为同一段物理内存。进程可以将同一段物理内存连接到他们自己的地址空间中。所有的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中写入数据，所做的改动将立即影响到可以访问同一段共享内存的任何其他进程。

在linux中，每个进程都有自己的地址空间，并且有一个与之对应的页表，负责将进程的虚拟地址与物理地址进行映射。不同进程各自不同的虚拟地址通过页表映射到物理空间的同一区域，它们所指向的这块区域即共享内存。

特点：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。

共享内存API:int shmget(size\_t size, key\_t key)(创建共享内存);返回一个共享内存标识符，用于后续的共享内存函数

void\* shmat(int shm\_id)(挂接共享内存);将先前创建的共享内存连接到当前进程中的地址位置。返回一个指向共享内存第一个字节的地址。

shmdt():去关联共享内存

产生死锁的原因：

系统资源不足；进程运行推进的顺序不合适；

死锁避免：

加锁顺序:确保所有的线程都是按照相同的顺序获得锁，那么死锁就不会发生。例如线程1企图加锁顺序是lock1,lock2,lock3。如果线程2加锁顺序是lock3,lock,lock1可能会死锁。必须相同顺序加锁。

加锁时限：在尝试获取锁的时候加一个超时时间，这意味着在尝试获取锁的过程中若超过了这个时限该线程则放弃对该锁的请求。若一个线程没有在给定的时限内成功获得所有需要的锁，则会进行回退并释放所有已经获得的锁，然后等待一段随机的时间再重试。

避免嵌套锁

避免在锁定了一个mutex后调用用户提供的代码：无法保证用户代码做了什么。

race condition(竞争条件):多线程对共享的数据进行读写操作时,最终的结果取决于这些线程的执行顺序；避免并发和防止竞争条件称为同步。

多线程同步机制如何实现：

多线程有3种同步方式：互斥锁，条件变量，读写锁，自旋锁(spinlock)

互斥锁: int pthread\_mutex\_lockpthread\_mutex\_trylock pthread\_mutex\_unlock

条件变量：互斥锁用于上锁,而条件变量用于等待,通常它都会跟互斥锁一起使用

Int pthread\_cond\_wait(pthread\_cond\_t\* cptr, pthread\_mutex\_t\* mptr) int pthread\_cond\_signal(pthread\_cond\_t\* cptr) int pthread\_cond\_broadcast(pthread\_cond\_t\* cptr)

条件变量是线程的一种同步机制

Pthread\_cond\_t qready = PTHREAD\_COND\_INITIALIZER;

Pthread\_mutex\_t qlock = PTHREAD\_MUTEX\_INITIALIZER;

//process\_msg

for (;;)

{

Pthread\_mutex\_lock(&qlock);

While (workq==nullptr)

Pthread\_cond\_wait(&qready, &qlock); //1.把调用线程放到条件等待队列上。2.释放mutex这两个操作是原子性的(捆绑在一起)

Pthread\_mutex\_unlock(&qlock);

//do the work

}

//enqueue\_msg

{

Pthread\_mutex\_lock(&qlock);

//………………….

Pthread\_mutex\_unlock(&qlock);

Pthread\_cond\_signal(&qready);

}

pthread\_cond\_wait(cond,mutex)的功能有3个:

调用者线程首先释放mutex;然后阻塞,等待被别的线程唤醒;当调用者线程被唤醒后,调用者线程会再次获取mutex

为什么ptrhead\_cond\_wait要加锁(mutex)？

当前线程执行pthread\_cond\_wait时,处于临界区访问共享资源,存在一个mutex与该临界区相关联。因此pthread\_cond\_wait要带有mutex参数。

当前线程执行pthread\_cond\_wait前,已经获得了和临界区相关联的mutex；执行pthread\_cond\_wait会阻塞,但是在进入阻塞状态前,必须释放已经获得的mutex;让其他线程能够进入临界区(pthread\_cond\_wait第一步先解锁)

当前线程执行pthread\_cond\_wait后，阻塞条件满足，被唤醒后，仍处于临界区，因此被唤醒后腰获得和临界区相关联的mutex

为什么判断条件要while (workq == NULL) ,不能换成if?当有多个消费线程时:因为生产者生产物品后,signal消息前，东西已经被另一个线程拿掉了；signal后，其他消费线程要重新判断队列空的条件(惊群效应)

当有多个消费线程时:

void enqueue\_msg(struct msg\* mp)

{

pthread\_mutex\_lock(&qlock);

////////////////////

pthread\_mutex\_unlock(&qlock);

//此时另外一个消费线程在signal之前,执行了process\_msg,刚好拿走了msg

pthread\_cond\_signal(&qready);

此时执行signal，在pthread\_cond\_wait等待的线程被唤醒,但是msg元素已经被另外一个线程拿走,所以workq 仍是NULL,因此需要继续等待。重新判断条件。

}

自旋锁(spinlock):linux内核中最常见的锁。短时间内进行轻量级加锁。让线程旋转而不是睡眠。中断上下文用自旋锁(中断过程不能睡眠)

与互斥锁类似：在任何时刻只能有一个保持者。但两者在调度机制上有不同：对于互斥锁，如果资源已经被占用，资源申请者只能进入睡眠状态。但是自旋锁不会引起调用者睡眠,如果自旋锁已经被别的线程保持,调用者就一直循环旋转，等待锁重新可用。

自旋锁可能存在两个问题：

死锁:试图递归的获得自旋锁必然会引起死锁

过多的占用cpu资源(一直在循环等待)。因此，一般自旋锁实现会有一个参数限定最多持续尝试次数，超出后，放弃当前time slice，等下次机会

自旋锁适用于保持锁时间比较短的情况(共享资源的访问时间非常短)，不应该被长时间持有。正是由于保持时间非常短，因此选择自旋而不是睡眠是非常重要的，自旋锁的效率远高于互斥锁

读写锁:(适用于对数据结构的读操作次数多于写操作次数的场合)

在一些程序中存在读者写者问题(对某些资源的访问会存在两种可能情况,一种访问是独占的，另一种访问可以是共享的)

读写锁比起互斥锁具有更高的适用性和并行性(可以有多个线程占用读模式的读写锁，只能有一个线程占用写模式的读写锁)

线程局部存储(TLS: Thread Local Storage)

主要是为了避免多个线程同时访问同一全局变量或者静态变量时导致的冲突，尤其是多个线程同时需要修改这一变量。通过TLS机制，为每一个使用该全局变量的线程都提供一个变量值的副本，每一个线程均可以独立改变自己的副本，而不会和其它线程的副本冲突。从线程的角度看，就好像每一个线程都完全拥有该变量。从全局变量的角度来看，好像一个全局变量被克隆成多份副本，每一份副本都可以被一个线程独立地改变。

程序如何转化为进程？

程序的生成：预编译，编译(将预处理文件转化成汇编文件)，汇编(将汇编文件转成目标文件)，链接->可执行程序

进程：操作系统将可执行程序复制到内存。内核将程序读入内存，为程序分配内存空间。内核为该进程分配进程标识符(PID)和其他所需资源。内核把进程放到运行队列中等待执行，程序转化为进程后可以被CPU调度了。

堆:new分配的内存块，编译器不会释放，由程序员控制(new ,delete)

栈:存放局部变量，函数参数， 函数的返回地址。

BSS段:未初始化的全局变量和静态局部变量

数据段:已初始化的全局变量和静态局部变量

代码段：可执行代码，字面常量，只读变量

Linux系统调用：用户程序->C库(API)：int 0x80->system\_call->系统调用服务程序->内核程序

Int 0x80:封装在C库函数中，执行此指令会让系统跳转到预设的内核空间地址，它指向系统调用处理程序

system\_call:根据具体的系统调用号转到执行具体的系统调用服务例程。

系统调用号作为参数放入寄存器中，syscall\_call函数读取寄存器获取参数，生成偏移地址，然后以sys\_call\_table(系统调用表)为基址，基址加偏移得到实际的系统调用服务程序地址 。系统调用完成后，把控制权交会给发起调用的用户进程。

Linux虚拟文件系统(vfs):把各种不同的文件系统抽象后采用统一的方式进行操作。使得用户可以直接使用open(),read()和write()这样的系统调用而无需考虑具体文件系统和实际物理介质。

用户空间(write())->VFS虚拟文件系统(sys\_write())->文件系统(文件系统特殊的写方法)->物理介质

unix系统将文件的相关信息和文件本身概念分开。访问控制权限，大小，拥有者，创建信息，文件信息等被称为文件的元数据，被存储在单独的数据结构中，该结构被称为索引节点(inode)

文件对象：表示进程已打开的文件在内存中的表示。进程直接处理的是文件而不是索引节点和目录项。文件对象包含访问模式，当前偏移等。文件对象由open()系统调用创建，由close()系统调用撤销

Linux进程地址空间：Linux采用虚拟内存技术，系统中所有进程之间以虚拟方式共享内存(对于一个进程，仿佛可以访问整个系统的所有物理内存)。

页表数据结构：页表号+页内偏移；根据页表寄存器中的页表起始地址，页表长度，得到页表基地址的偏移量就是对应的页框。加上页内偏移就是最终的物理地址(内存地址)

多级页表：由于物理内存需要很大的连续空间存放页表。可以通过多级页表的方式，将页表分为多个部分，分别存放，这样就不需要连续的整段内存，只需要多个连续的小段内存即可。

分段，分页区别：

分页存储管理：用固定大小的页来描述逻辑地址空间，用相同大小的页框来描述物理内存空间，由操作系统实现从逻辑页到物理页框的页面映射。

分段存储管理：作业的地址空间被划分为若干个段，每个段定义了一组逻辑信息。每个段的长度不等。整个作业的地址空间由于分成多个段，其逻辑地址由段号和段内地址组成。由段表和段表寄存器实现从逻辑地址到物理地址的映射。

TLB：硬件缓冲。只包含页表中的一小部分条目。如果页号在TLB中，得到帧号，访问内存；否则从内存中的页表得到帧号，将其存入TLB，访问内存。

虚拟地址空间通过页表映射到物理内存，页表由OS维护，内核空间在页表中有高优先级，用户进程程序试图访问这些页表会导致页错误。在linux中内核空间是持续存在的，并且所有内核进程都映射到同样的物理内存。内核代码和数据总是可寻址，随时准备处理中断和系统调用。与此相反，用户进程地址空间的映射随进程的切换的发生而不断变化。

在将应用程序加载到内存空间执行时，OS负责代码段，数据段和BSS段的加载，并在内存中为这些段分配空间。栈也由OS分配和管理；堆由程序员管理，显式地申请和释放。

Linux虚拟地址空间：

内核空间：总是驻留在内存中，内核空间被内核保留，不允许应用程序读写该区域地内容。

栈：又称为堆栈，编译器自动分配和释放；

堆(不是数据结构中的堆，行为类似链表)；

BSS段(未初始化的全局变量)；

数据段(已初始化的全局变量)；

代码段；

内存映射段(mmap):内核将硬盘文件的内容直接映射到内存；高效的文件I/O

保留区:位于虚拟地址空间的最低部分,未赋予物理地址..任何对它的引用都是非法的。

缺页中断：

当软件试图访问虚拟地址空间中，但是未被加载到物理内存中的一个分页，由中断处理程序将相关分页从硬盘上的文件调入内存。系统检查是否有空闲页框，如果有直接从硬盘换入，如果没有，执行页面置换算法换页。

mmap共享内存与内存映射:映射一个文件描述符和指定的文件区域至调用进程的内存区域中。mmap的返回值为最后文件映射到进程空间的地址，进程可以直接操作起始地址为该值的有效地址(传统读写文件方式需要在进程的地址空间中保存文件的副本，浪费了存储空间)。

内存映射的使用场景：共享内存映射(一个进程改了，另一个进程可以看到)。使得进程之间通过映射同一个普通文件实现共享内存。普通文件被映射到进程地址空间后，进程可以像访问普通内存一样对文件进行访问，不必再调用read,write。

伙伴系统(buddy system)：解决外部碎片。大内存分配。连续页面。用最小的内存块来满足内核对于内存的请求。

实际应用中，经常需要分配一组连续的页框，而频繁申请和释放不同大小的连续页框，必然导致已分配页框地内存块中分散了许多小块的空闲页框。这样，即使这些页框是空闲的，其他需要连续分配页框的应用很难得到满足。

Linux内核引入了伙伴系统算法。把所有的空闲页框分组为11个块链表，每个块链表分别包含大小为1，2，4，8，16，32，64，128，256，1024个连续页框的页框块。比如第0个块链表每个元素包含大小为2^0个连续的页框。第1个块链表中，每个链表元素包含2^1个页框大小的连续地址空间，……..第10个块链表中，每个链表元素代表4M(4K\*2^10)的连续地址空间。最大可以申请1024个连续页框。每个页框块的第一个页框的物理地址是该块大小的整数倍。

假设要申请一个256个页框的块，先从256个页框的链表中查找空闲块，如果没有，就去512个页框的链表中找，找到就将页框块分为2个256个页框的块，一个分配给应用，另一个移到256个页框的链表中。若512页框的链表中仍没有空闲块，继续向1024个页框的链表查找，仍没有，返回错误。释放内存时，首先在链表中查找是否有伙伴存在，如果没有伙伴，直接将释放的块插入链表头。如果有伙伴的存在则将其从链表摘下，合并成一个大块，然后继续查找合并后的块在更大一级链表中是否有伙伴存在，直到不能合并为止。

slab算法:解决内部碎片，解决小块内存的分配。

伙伴算法实现的分区页框分配器适合大块内存的请求，它所分配的内存是以页框为基本单位的。对于内核中小块连续内存的请求，比如说几个字节或者几百个字节，如果依然分配一个页框来满足请求，是一种浪费，会产生内部碎片。

为了解决小块内存的分配，Linux内核的slab算法实现了自己的slab分配器。除此之外，slab分配器另一个主要功能是作为一个高速缓存，它用来存储内核中那些经常分配释放的对象。

slab分配器的三个目标：

减少伙伴算法在分配小块连续内存时所产生的内部碎片。

将频繁使用的对象缓存起来，减少分配，初始化和释放对象的时间开销。

通过着色技术调整对象以更好的使用硬件高速缓存。

slab分配器中用到了对象这个概念，所谓对象就是内核中的数据结构以及该数据结构进行创建和撤销的操作。它的基本思想是将内核中经常使用的对象放到高速缓存中，并且由系统保持为初始可利用状态。比如进程描述符，内核中会频繁对此数据进行申请和释放。当一个新进程创建时，内核会直接从slab分配器的高速缓存中获取一个已经初始化了的对象；当进程结束时，该结构所占的页框并不被释放，而是重新返回slab分配器中。如果没有基于对象的slab分配器，内核将花费更多的时间去分配，初始化以及释放一个对象。

slab分配器为每种对象分配一个高速缓存，这个缓存可以看作是同类型对象的一种储备。每个高速缓存所占的内存区又被划分为多个slab，每个slab是由一个或多个连续的页框组成。每个页框包含若干个对象，既有已分配的对象，也包含空闲的对象。

所有高速缓存通过双链表组织在一起形成高速缓存链表cache\_chain。当前缓存中的所有slab分为三个集合：空闲对象的slab链表slabs\_free，非空闲对象的slab链表slabs\_full以及部分空闲对象的slab链表slabs\_partial。

Linux内核锁：自旋锁(不会睡眠，忙循环等待)和信号量(会睡眠，适用于锁被长时间持有。只能在进程上下文使用，因为中断上下文中是不能被调度的)

Linux中的用户模式和内核模式：

在linux上，CPU要么处于受信任的内核模式，要么处于受限制的用户模式。除了内核本生处于内核模式之外，所有的用户进程都运行在用户模式之中。内核模式的代码可以无限制地访问所有处理器指令集和全部内存和IO空间。用户模式的进程必须通过系统调用向设备驱动程序或其它内核模块地代码发出请求。另外，用户模式地代码允许发生缺页，而内核模式地代码不允许。

栈溢出：局部数组过大(局部变量是存储在栈中的。解决方法：增大栈空间，使用堆不是栈)。递归调用层次太多。指针或数组越界(字符串拷贝，处理用户输入)。

栈溢出指向栈中(变量和数组)写入了超出限定长度的数据，溢出的数据会覆盖栈中其他数据，从而影响程序的运行。上溢：栈满时，进栈会产生空间溢出。下溢：栈空时，退栈产生空间溢出。

内存溢出(out of memory)：

系统已经不能再分配出你需要的空间(比如需要100M的空间，系统只剩90M，叫做内存溢出)

可能发生内存溢出的位置：

检查对数据库查询中，是否一次获得全部数据的查询。因为一次取十万条记录到内存，可能引起内存溢出。

检查代码中是否有死循环或递归调用。

检查是否大循环重复产生新对象实体。

内存泄露(memory leak)：

使用资源的时候申请了资源，但没有手动释放资源。

如何检测内存泄露？

mutable(易变的，可变的,和const相反,突破const的限制) mutable修饰的成员变量在const成员函数中可以改变

volatile(c++关键字)：

易变形： 一个定义为volatile的变量是说这个变量可能会被意象不到的改变。当编译器遇到这个变量时，只能从变量的内存地址中读取这个变量，不可以从缓存，寄存器，或者其它任何地方读取。场景:多线程中被几个线程共享的变量。

顺序性：两个包含volatile变量的指令，编译后不可以乱序。注意是编译后不乱序，但是在执行的过程中还是可能会乱序的。这点需要由其它机制保证，例如memory-barriers。

C++中的volatile并不是用来解决多线程竞争问题的，而是用来修饰一些程序不可控制因素导致变化的变量，比如访问底层硬件设备的变量，来提醒编译器不要对该变量的访问擅自进行优化。 C++11标准明确指出解决多线程的数据竞争问题应该使用原子操作或互斥锁。

CAS(无锁算法)基本原理：为了解决多线程并行情况下使用锁造成性能损耗的问题，引入了CAS机制(campare and set)。有了这个原子操作，就可以实现各种无锁(lock free)的数据结构。

总结：

CAS(compare and set)比较并替换，是线程并发运行时用到的一种技术

CAS是原子安全，保证并发安全，而不保证并发同步。

CAS是CPU的一个指令。

CAS是非阻塞的，轻量级的乐观锁。

CAS操作包含三个操作数:当前值(old val)，预期值(expected val)，新值(new val)

bool oldval.compare\_exchange\_weak(T& expected, T newval); 比较并交换被封装的值与参数expected所指定的值是否相等。如果相等，则用val替换原来的旧值,返回true。不相等，则用原子对象的旧值替换expected，返回false。 一般情况下是一个自旋操作，不断重试

ABA问题：

并发1:获取出的数据初始值是A，后续计划实施CAS乐观锁期望数据仍是A的时候，修改才能成功。

并发2：将数据修改成B

并发3：将数据修改回A

并发1：CAS乐观锁，检测发现初始值还是A，进行数据修改

上述并发情况下，并发1在修改数据时，虽然还是A，但已经不是初始条件的A了，中间发生了A变B，B又变回了A，此A已非彼A，数据却成功修改，可能导致错误，这就是CAS引发的所谓的ABA问题。

ABA问题导致的原因，是CAS过程中只简单进行了值的比较，在有些情况下值相同不会带来错误，有些情况下，值虽然相同，却已经不是原来的数据了。

一个常用的方法：不能仅仅比较值，要必须确保的是原来的数据，才能修改成功。是添加额外的tag(版本号)或stamp来标记指针是否被修改过。

无锁队列

乐观锁：多读

总是假设最好的情况，每次拿数据的时候都认为别人不会修改，所以不会上锁，但是在更新时会判断在此期间别人有没有去更新这个数据，可以使用版本号机制和CAS算法实现。乐观锁适用于多读的应用类型，这样可以提高吞吐量。

悲观锁：多写

假设最坏的情况，每次去拿数据的时候都认为别人会修改，所以每次在拿数据的时候都会上锁，这样别人想拿数据就会阻塞直到它拿到锁。传统的关系型数据库用到了很多这种锁机制，比如行锁，表锁，读锁，写锁等，都是在做操作前先上锁。