**6-操作系统**

**基本概念**

**操作系统的特征**

* **并发：**指的是两个或者多个事件在同一时间间隔内发生
* **共享：**指的是系统中的资源可以供内存中多个并发执行的进程共同使用
* **虚拟：**把一个物理上的实体变为若干个逻辑上的对应物
* **异步：**允许多个程序并发执行，但是由于资源有限 ，进程的执行不一定是一起执行完成，而是走走停停，以不可预知的速度向前推进

|  |
| --- |
| **并发**强调的是多个任务同时进行，可以使用**多进程**或者**多线程**实现 **异步**指的是任务完成不会阻塞其他任务，可以使用**非阻塞的方式**实现 |

**操作系统的两种指令**

**特权指令**

不允许用户程序使用，只允许操作系统使用。如:IO指令，中断指令

**非特权指令**

普通的运算指令，用户只能使用非特权指令

**处理器两种状态**

**用户态**

CPU只能执行非特权指令

**核心态**

CPU任何指令都可以执行

**用户态到核心态**

通过中断(是由硬件完成的)。例如:访管指令，发生在用户态。

**核心态到用户态**

通过标志位的转换完成，0用户态，1核心态

**用户态和内核态切换流程**

1. 上下文切换：

当系统从用户态切换到内核态时，需要保存当前运行的用户态程序的上下文（如CPU寄存器、程序计数器等），然后加载内核态的上下文。这包括：

保存用户态的寄存器状态：在切换之前，当前用户进程的寄存器状态需要保存，以便以后切换回来时能够继续执行。

加载内核态的寄存器状态：切换到内核态后，系统需要加载内核态的寄存器状态，以执行内核代码。

上下文切换是一个复杂的过程，需要时间来完成这些操作。

1. 模式切换：

用户态和内核态有不同的权限级别和访问范围。用户态程序受到严格限制，不能直接访问硬件或系统关键资源，而内核态可以执行所有的指令和访问所有资源。切换到内核态时，CPU需要改变运行模式，这涉及：

特权级的改变：现代CPU有多个特权级别，用户态和内核态对应不同的特权级。切换需要改变CPU的特权级别。

中断处理和陷阱：通常，切换是通过系统调用或中断触发的。这些机制需要执行额外的指令来处理中断或系统调用请求。

模式切换涉及到CPU的状态变化，这增加了额外的开销。

1. 内存管理：

内核态和用户态可能有不同的内存地址空间和页面映射：

页面表的切换：如果内核和用户空间有不同的页面表，切换过程中需要更改页面表，这会导致TLB（Translation Lookaside Buffer，翻译后援缓冲区）失效，需要重新加载页面表信息。

地址空间转换：CPU需要在内核和用户空间地址之间进行转换，这些操作需要额外的处理时间。

1. 缓存的影响：

缓存失效：在切换过程中，由于不同态下的代码和数据可能占用不同的缓存，当从一个态切换到另一个态时，缓存内容可能需要重新加载，这会导致性能损失。

1. 安全性检查：

权限验证：从用户态进入内核态时，系统通常需要执行权限检查，以确保用户进程有权限执行该操作。这些安全检查需要额外的处理时间。

**中断**

**概念**

发生中断，意味着操作系统需要介入展开工作，cpu 会进入核心态，中断是 cpu 从用户态进入核心态的唯一途径

**出现原因**

为了实现多道程序并发执行而引入的技术

**中断分类**

* **内中断：**
* 用户态执行特权指令，会被系统中断阻止
* 执行非法指令（数组越界，除 0 等）
* 陷入（用户通过系统调用，将用户态转为核心态，由操作系统代为执行）
* **外中断：**
* 时钟中断（cpu 执行时间间隔）
* 外设中断 (打印机等）

**中断执行过程**

首先指令执行完后，cpu 检查是否有中断，如果有，需要保护被中断进程的 cpu 环境，根据中断信号转入处理中断处理程序，处理完成后退出中断，返回原进程继续执行

**进程**

**进程基本介绍**

**进程概念**

进程是系统进行资源分配和调用的一个独立单位。进程实体由程序段、数据段、PCB 三部分组成。创建进程就是创建 PCB，PCB 是进程存在的唯一标志。

**PCB的组成**

* **进程描述信息：**进程标识符 PID、用户标识符 UID、进程的父进程 PPID（祖先会是 init 进程，是操作系统的引导进程）
* **进程控制和管理信息：**进程当前状态、进程优先级
* **资源分配清单：**程序段指针、数据段指针、键盘、鼠标
* **处理及相关信息：**各种寄存器

**进程的组织形式**

* **链接方式：**按进程状态将PCB分为多个队列
* **索引方式：**按照进程状态建立几张索引表，各表项指向一个PCB

**进程的特征**

* **动态性：**进程与程序的区别在于，程序只是一个静态的指令集合，而进程是一个正在系统中活动的指令集合。在进程中加入了时间的概念。进程具有自己的生命周期和各种不同的状态，这些概念在程序中都是不具备的。
* **并发性：**多个进程可以在单个处理器上并发执行，多个进程之间不会互相影响。
* **独立性：**进程是系统中独立存在的实体，它可以拥有自己独立的资源，每一个进程都拥有自己私有的地址空间。在没有经过进程本身允许的情况下，一个用户进程不可以直接访问其他进程的地址空间。
* **异步性：**由于进程间的相互制约，使进程具有执行的间断性，即进程按各自独立的、不可预知的速度向前推进
* **结构性：**进程由程序、数据和进程控制块三部分组成。

**进程状态与转换**

**进程的状态分类**

运行状态、就绪状态、阻塞状态、创建状态、终止状态

**进程状态间的转换**

* **就绪态 ------> 运行态：**分配到 cpu 资源
* **运行态 ------> 就绪态：**没有空闲 cpu
* **运行态 ------> 阻塞态：**没有资源分配
* **阻塞态 ------> 就绪态：**获得资源

**进程控制与通信**

进程控制就是要是实现进程的状态切换，使用原语来控制

**创建原语**

申请空白 PCB，为新进程分配资源，初始化 PCB，把 PCB 插入到就绪队列

**终止原语**

在 PCB 集合中找到终止进程的 PCB，如果进程正在运行，立即剥夺 CPU，将 CPU 分配给其他进程，并终止所有子进程。将进程拥有的所有资源归还给父进程或者操作系统，删除 PCB

**阻塞和唤醒原语**

找到阻塞进程对应的 PCB，保护进程运行环境，设置PCB状态为阻塞态，停止进程运行，将 PCB 插入相应的事件的等待队列

**进程切换**

将运行环境信息存入 PCB，将 PCB 移入相应队列，选择另一个进程执行，更新其 PCB，根据 PCB 恢复新进程所需的运行环境

[进程切换](https://www.baidu.com/s?wd=%E8%BF%9B%E7%A8%8B%E5%88%87%E6%8D%A2&usm=5&ie=utf-8&rsv_pq=d267ee8402707451&oq=%E8%BF%9B%E7%A8%8B%E5%88%87%E6%8D%A2%E8%BF%87%E7%A8%8B&rsv_t=8a34y9d%2BOB%2B%2FCKmqDFK7lbw%2B26m0epJ%2Bi%2B%2BZzrrAJk%2FRM20uGBrpM3cfvA8&sa=re_dqa_generate)过程涉及从正在运行的进程中收回处理器，然后让待运行的进程占用处理器。这一过程实质上是上下文的切换，包括保存当前进程的执行状态（如程序计数器、寄存器内容、[堆栈指针](https://www.baidu.com/s?wd=%E5%A0%86%E6%A0%88%E6%8C%87%E9%92%88&usm=5&ie=utf-8&rsv_pq=d267ee8402707451&oq=%E8%BF%9B%E7%A8%8B%E5%88%87%E6%8D%A2%E8%BF%87%E7%A8%8B&rsv_t=8a34y9d%2BOB%2B%2FCKmqDFK7lbw%2B26m0epJ%2Bi%2B%2BZzrrAJk%2FRM20uGBrpM3cfvA8&sa=re_dqa_generate)等），并恢复下一个进程的执行状态。进程切换是[操作系统](https://www.baidu.com/s?wd=%E6%93%8D%E4%BD%9C%E7%B3%BB%E7%BB%9F&usm=5&ie=utf-8&rsv_pq=d267ee8402707451&oq=%E8%BF%9B%E7%A8%8B%E5%88%87%E6%8D%A2%E8%BF%87%E7%A8%8B&rsv_t=d43fF1XPAMaQzc5CAiSGRSnFmyA4J0rR2YQI0VVqPM3zjX8uVYd%2BKZ4SE3g&sa=re_dqa_generate)为了控制进程执行所必须具备的基本功能，它允许操作系统挂起正在[CPU](https://www.baidu.com/s?wd=CPU&usm=5&ie=utf-8&rsv_pq=d267ee8402707451&oq=%E8%BF%9B%E7%A8%8B%E5%88%87%E6%8D%A2%E8%BF%87%E7%A8%8B&rsv_t=d43fF1XPAMaQzc5CAiSGRSnFmyA4J0rR2YQI0VVqPM3zjX8uVYd%2BKZ4SE3g&sa=re_dqa_generate)上运行的进程，并恢复以前挂起的某个进程的执行。这种行为也被称为任务切换或上下文切换。12

在进程切换过程中，一个进程存储在处理器各寄存器中的中间数据被称为进程的上下文。当进程未占用处理器时，这些上下文是存储在进程的私有堆栈中。整个过程依赖于堆栈指针的切换，以确保正确保存和恢复进程的状态。进程切换的时机可以是主动的，如[IO语句](https://www.baidu.com/s?wd=IO%E8%AF%AD%E5%8F%A5&usm=5&ie=utf-8&rsv_pq=d267ee8402707451&oq=%E8%BF%9B%E7%A8%8B%E5%88%87%E6%8D%A2%E8%BF%87%E7%A8%8B&rsv_t=ec3c9pVcYNb8tiErjZEvIaIFggfXOhZyLMgaqsoKNOehvP3ZnlxvSz1PTNs&sa=re_dqa_generate)或系统调用，也可以是被动的，如高优先级进程的到来或时间片到期。

此外，进程切换通常需要较多的开销，因为它涉及到整个进程上下文的切换，包括[内存映射](https://www.baidu.com/s?wd=%E5%86%85%E5%AD%98%E6%98%A0%E5%B0%84&usm=5&ie=utf-8&rsv_pq=d267ee8402707451&oq=%E8%BF%9B%E7%A8%8B%E5%88%87%E6%8D%A2%E8%BF%87%E7%A8%8B&rsv_t=ec3c9pVcYNb8tiErjZEvIaIFggfXOhZyLMgaqsoKNOehvP3ZnlxvSz1PTNs&sa=re_dqa_generate)、[文件描述符](https://www.baidu.com/s?wd=%E6%96%87%E4%BB%B6%E6%8F%8F%E8%BF%B0%E7%AC%A6&usm=5&ie=utf-8&rsv_pq=d267ee8402707451&oq=%E8%BF%9B%E7%A8%8B%E5%88%87%E6%8D%A2%E8%BF%87%E7%A8%8B&rsv_t=ec3c9pVcYNb8tiErjZEvIaIFggfXOhZyLMgaqsoKNOehvP3ZnlxvSz1PTNs&sa=re_dqa_generate)等，以及可能进行的[页表切换](https://www.baidu.com/s?wd=%E9%A1%B5%E8%A1%A8%E5%88%87%E6%8D%A2&usm=5&ie=utf-8&rsv_pq=d267ee8402707451&oq=%E8%BF%9B%E7%A8%8B%E5%88%87%E6%8D%A2%E8%BF%87%E7%A8%8B&rsv_t=c635Uco5sXzzw3CITnsdDTQQjd7Az8pU85ARKA%2FUNc49Yx75TN1OkpmCsdQ&sa=re_dqa_generate)和[缓存刷新](https://www.baidu.com/s?wd=%E7%BC%93%E5%AD%98%E5%88%B7%E6%96%B0&usm=5&ie=utf-8&rsv_pq=d267ee8402707451&oq=%E8%BF%9B%E7%A8%8B%E5%88%87%E6%8D%A2%E8%BF%87%E7%A8%8B&rsv_t=c635Uco5sXzzw3CITnsdDTQQjd7Az8pU85ARKA%2FUNc49Yx75TN1OkpmCsdQ&sa=re_dqa_generate)等操作。相比之下，‌[线程切换](https://www.baidu.com/s?wd=%E7%BA%BF%E7%A8%8B%E5%88%87%E6%8D%A2&usm=5&ie=utf-8&rsv_pq=d267ee8402707451&oq=%E8%BF%9B%E7%A8%8B%E5%88%87%E6%8D%A2%E8%BF%87%E7%A8%8B&rsv_t=c635Uco5sXzzw3CITnsdDTQQjd7Az8pU85ARKA%2FUNc49Yx75TN1OkpmCsdQ&sa=re_dqa_generate)的开销较小，因为线程共享进程的资源，不需要额外的内存映射和页表切换。

**进程通信**

* **共享内存：**两个或者多个进程可以访问公共内存空间。在使用共享内存前，需要经过一系列的调用流程，流程如下
* 创建共享内存段或者使用已创建的共享内存段(shmget())
* 将进程附加到已经创建的内存段中(shmat())
* 从已连接的共享内存段分离进程(shmdt())
* 对共享内存段执行控制操作(shmctl())
* **信号：**通过向一个或多个进程发送异步事件信号来实现，信号可以从键盘或者访问不存在的位置等地方产生；信号通过 shell 将任务发送给子进程。进程可以选择忽略发送过来的信号，但是有两个是不能忽略的：SIGSTOP 和 SIGKILL 信号。SIGSTOP 信号会通知当前正在运行的进程执行关闭操作，SIGKILL 信号会通知当前进程应该被杀死。除此之外，进程可以选择它想要处理的信号，进程也可以选择阻止信号，如果不阻止，可以选择自行处理，也可以选择进行内核处理。
* **管道通信：**在两个进程之间，可以建立一个通道，一个进程向这个通道里写入字节流，另一个进程从这个管道中读取字节流。管道是同步的，当进程尝试从空管道读取数据时，该进程会被阻塞，直到有可用数据为止。
* **命名管道：**先入先出队列 FIFO 通常被称为 命名管道(Named Pipes)，命名管道的工作方式与常规管道非常相似，但是确实有一些明显的区别。未命名的管道没有备份文件：操作系统负责维护内存中的缓冲区，用来将字节从写入器传输到读取器。一旦写入或者输出终止的话，缓冲区将被回收，传输的数据会丢失。相比之下，命名管道具有支持文件和独特 API ，命名管道在文件系统中作为设备的专用文件存在。当所有的进程通信完成后，命名管道将保留在文件系统中以备后用。
* **消息队列：**消息队列是用来描述内核寻址空间内的内部链接列表。可以按几种不同的方式将消息按顺序发送到队列并从队列中检索消息信号。消息队列有两种模式，一种是严格模式， 严格模式就像是 FIFO 先入先出队列似的，消息顺序发送，顺序读取。还有一种模式是 非严格模式，消息的顺序性不是非常重要。
* **套接字：**两个进程间通信的是使用 socket，socket 提供端到端的双向通信。

**进程调度**



进程调度发生时机是在主动放弃或者被动放弃，比如资源被剥夺，cpu 时间片到期。在处理中断、进程在系统内核的临界区，原则操作过程中是不能进行调度的。调度方式有剥夺式调度和非剥夺式调度。

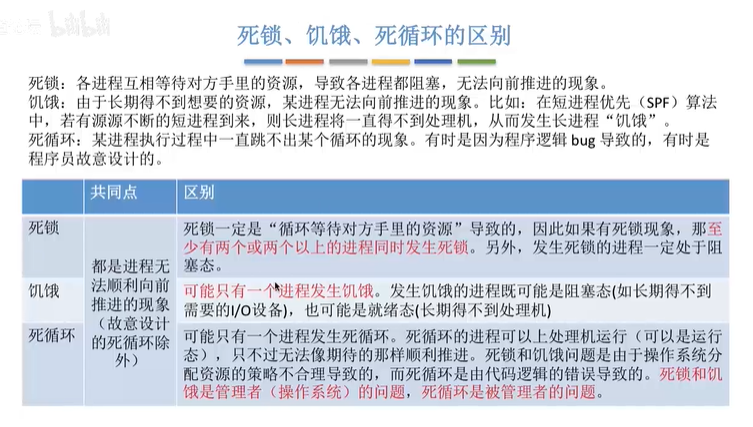
**调度算法**

* **先来先服务：**先来的先进行服务，对长作业不利，短作业有利，不会饥饿
* **短作业优先算法：**最短作业优先服务，非抢占，会产生饥饿
* **高相应比优先算法：**相应比为(等待时间+要求服务事件)/要求服务事件，非抢占式，不会饥饿
* **时间片轮转调度算法：**每个任务得到一个时间片运行任务，抢占式，不会饥饿，高频率的进程切换会有开销。
* **优先级调度算法：**调度选择优先级最高的，可以抢占也可以不抢占，会饥饿
* **多级反馈队列调度算法：**设置多级优先级队列，每一个队列优先级从高到低，每个队列都有时间片，从小到大。先执行完一个队列的所有任务，再去执行下一个队列任务。会饥饿，抢占式。

**进程同步与互斥**

管程是专门来解决进程的同步问题

**进程死锁**



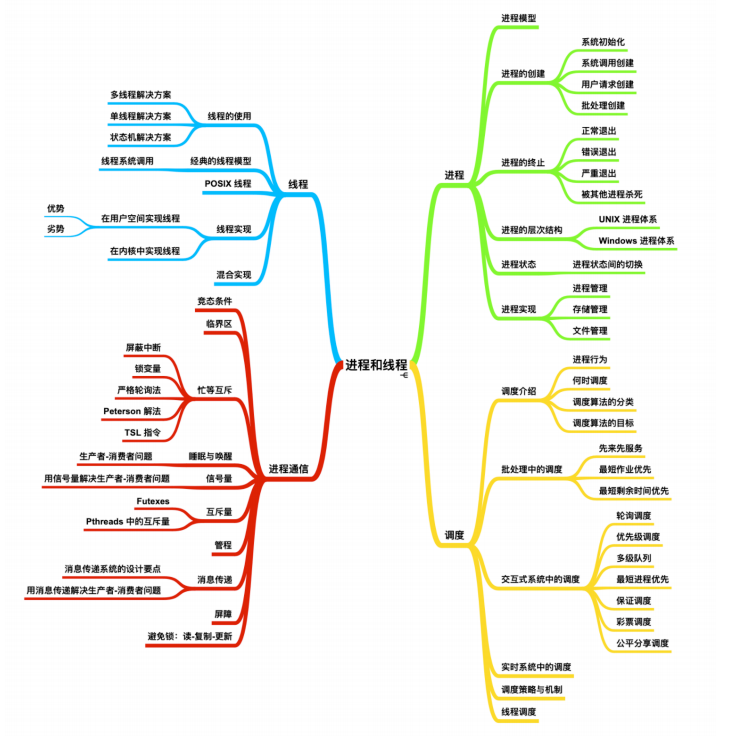
**死锁的四个条件**

* **互斥条件**
* **不剥夺条件**
* **请求和保持条件**
* **循环等待条件**

**死锁处理**

* **预防死锁：**破坏四个条件中的一个
* **避免死锁：**使用银行家算法预先先检测是否会发生死锁
* **死锁检测和接触：**通过资源分配图检查死锁，检查出原因后使用资源剥夺，撤销进程或者进程回退来解除死锁

**线程**



线程是cpu调度的最小单位。引入线程后提高了并发度，减小了系统开销。

**线程控制块TCB**

1. 线程标识符
2. 一组寄存器，包括程序计数器、状态寄存器和通用寄存器
3. 线程运行状态
4. 优先级
5. 线程专有存储区：保护现场
6. 堆栈指针：用于过程调度时保持局部变量及返回地址

**线程属性**

* 线程几乎不拥有系统资源
* 同一进程的不同线程间共享进程的资源，共享内存地址空间
* 线程间通信无需系统干预
* 同一进程中线程的切换不会引起进程的切换，不同进程的线程切换会引起线程切换
* 切换进程内的线程系统开销小

**多线程模型**

* **多对一模型：**进程管理开销小效率高，但是一个线程阻塞会导致整个进程阻塞，并发小
* **一对一：**进程管理开销大，线程可以分配到多核处理机并行执行，并发高
* **多对多：**进程管理开销小，并发高

**线程同步方式**

**c++:**

* 互斥锁
* 信号量
* 条件变量
* 读写锁
* 自旋锁

**POSIX**

POSIX 线程 通常称为 pthreads是一种独立于语言而存在的执行模型，以及并行执行模型。



**点击图片可查看完整电子表格**

**协程**

比线程更加轻量级，由用户的程序控制，没有线程切换的开销，在协程控制共享资源不加锁，因为协程是基于一个线程的，不存在同时写变量冲突，协程执行过程中可以中断，协程开销是由语言自己运行时候管理，因此开销更低。切换开销更小，线程需要切换资源，需要和内核打交道，协程不用。并发更强，速度和线程差不多。

**协程同步方式**

**go**：

* channel
* waitgroup
* context
* mutex

**内存**

**程序到程序运行过程**

编译-->链接-->装入

**链接的三种方式**

* **静态链接：**装入前链接成一个完整的装入模块
* **装入时动态链接：**运行前边装入边链接
* **运行时动态链接：**运行时需要目标模块才装入并链接

**装入的三种方式**

* **绝对装入：**编译时产生绝对地址
* **可重定位装入：**装入时将逻辑地址转换为物理地址
* **动态运行时装入：**运行时将逻辑地址转换为物理地址，需要重定位寄存器

**内存的分配与回收**

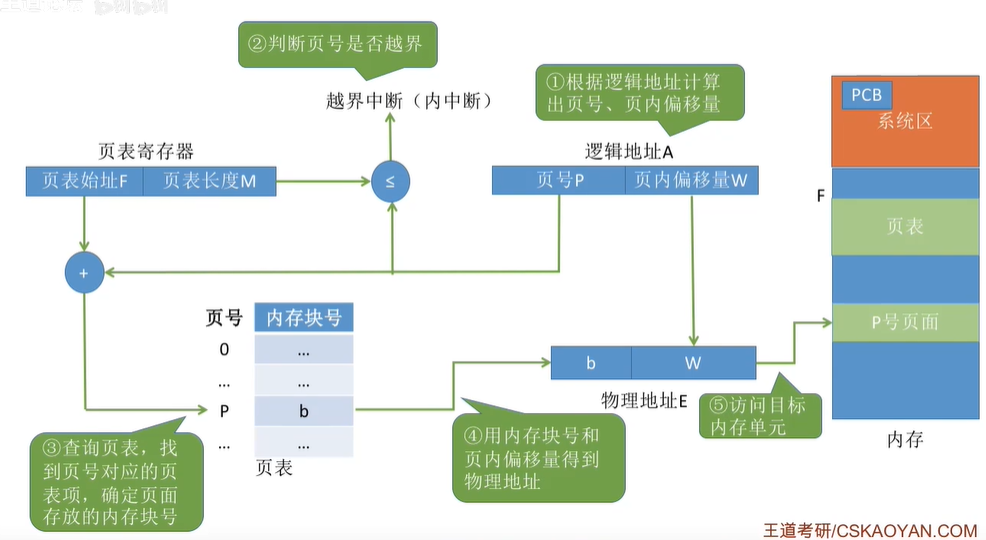
**连续分配方式**

* **单一连续分配**
* **固定分区分配**
* **动态分区分配**
* 首次适应算法：从头到尾找最合适的分区
* 最佳适应算法：优先使用最小的分区，开销大会产生很多碎片
* 最坏适应算法：优先使用最大的分区，开销大，大分区会一下用完
* 临近适应算法：由首次适应演变，从上次查找结束位置开始找，大分区会一下用完

**非连续分配方式**

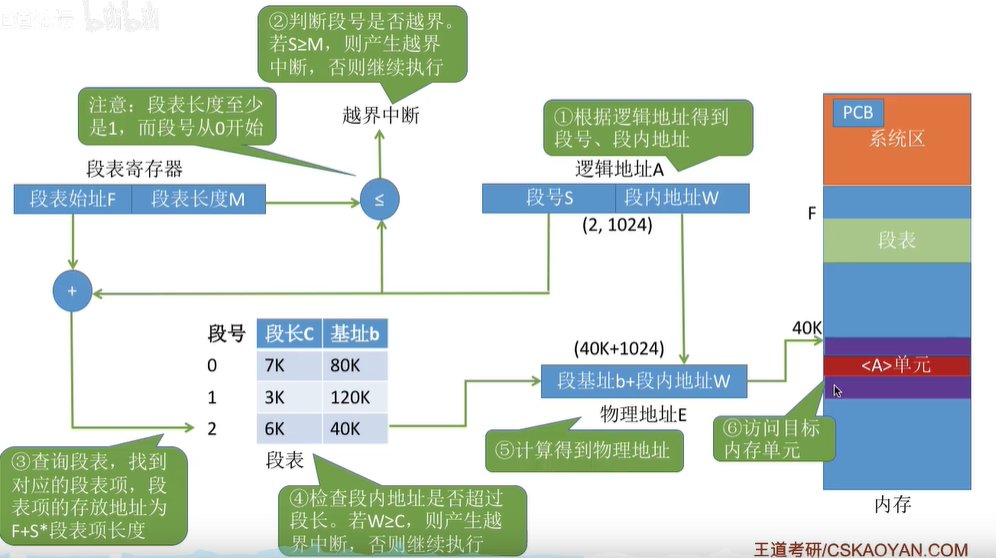
**基本分页存储管理**

为每一个进程建立一张页表，记录每个页表对应的块号。内存根据系统的页面大小划为N个页面，每个页面对应内存中的某一块。下图是计算访问内存中目标元素时候的访问方法。



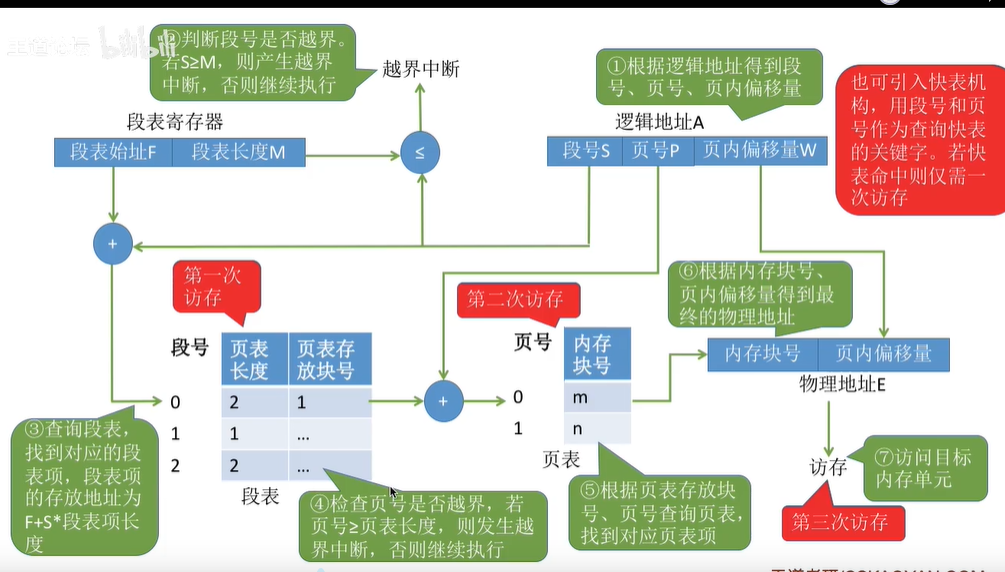
**基本分段存储管理**

为每一个进程建立一张段表，记录每个段表对应的块号。内存根据系统的段大小划为N个段，每个段对应内存中的某一块，段的大小是不同的。下图是计算访问内存中目标进程时候的访问方法。



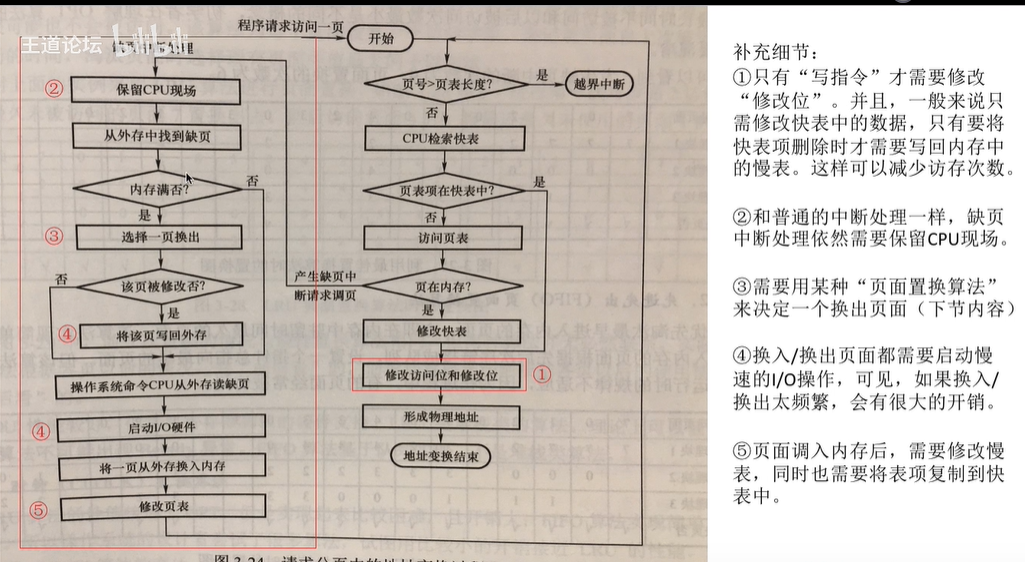
**段页式存储管理**

为每一个进程建立一张段表，记录每个段表对应的块号。内存根据系统的段大小划为N个段，每个段划为成大小相同的页数，每个段的页大小都是一样的，每个页对应内存中的某一块，段的大小是不同的。下图是计算访问内存中目标进程时候的访问方法。



**虚拟内存技术**

当内存不够的时候，会将一部分页面存入磁盘，当需要这部分的时候再进行页面置换，这叫做页面置换机制，使用了中断机制来实现，实现流程如下



**页面置换的方法**

**最佳置换算法**

每次选择的淘汰页面是以后用不使用，或者长时间内不再被访问的页面。不可能实现。

**先进先出算法（FIFO）**

淘汰页面是最早进入内存的页面，会导致页面抖动（刚刚换出去的页面又要换入内存，可能页面置换比执行任务事件还长）

**最近最久未使用置换算法（LRU）**

每次淘汰就是最近最久没有使用的页面，但是需要硬件支持，实现困难大，开销大

**最不经常使用(LFU)**

页面置换算法要求置换具有最小计数的页面。

这种选择的原因是，积极使用的页面应当具有大的引用计数。然而，当一个页面在进程的初始阶段大量使用但是随后不再使用时，会出现问题。由于被大量使用，它有一个大的计数，即使不再需要却仍保留在内存中。一种解决方案是，定期地将计数右移1位，以形成指数衰减的平均使用计数。

**时钟置换算法**

每个页面设置一个访问位，再用指针将页面链接成一个循环队列。当被访问的时候，将访问位置为1。当需要置换的时候首先将为0的置换出去，如果都是1，则将所有置为0，然后将第一个1置换出去，最多两轮会进行置换

**改进型时钟置换算法**

设置一个访问位和修改位，首先淘汰没有修改和访问的，其次是没访问但是修改过，然后是访问过但是没有修改过，最后是访问过修改过。

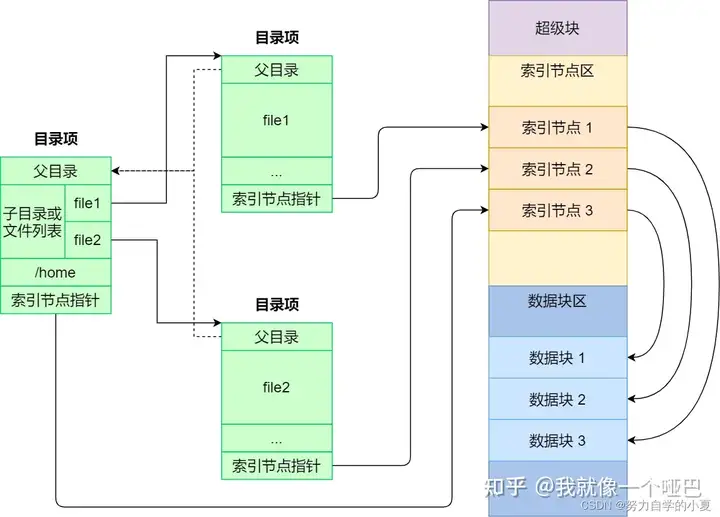
**文件系统**

文件系统主要负责管理和组织计算机存储设备上的文件和目录，其功能包括以下几个方面：

1. **存储管理**：将文件数据存储到物理存储介质中，并且管理空间分配，以确保每个文件都有足够的空间存储，并避免文件之间发生冲突。
2. **文件管理**：文件的创建、删除、移动、重命名、压缩、加密、共享等等。
3. **目录管理**：目录的创建、删除、移动、重命名等等。
4. **文件访问控制**：管理不同用户或进程对文件的访问权限，以确保用户只能访问其被授权访问的文件，以保证文件的安全性和保密性。

**文件存放形式**

磁盘块号+块内地址，操作系统同样需要将逻辑地址转换为物理地址。



Linux 文件系统会为每个文件分配两个数据结构：索引节点（index node）和目录项（directory entry），它们主要用来记录文件的元信息和目录层次结构。

* 索引节点，也就是 inode，用来记录文件的元信息，比如 inode 编号、文件大小、访问权限、创建时间、修改时间、数据在磁盘的位置等等。索引节点是文件的唯一标识，它们之间一一对应，也同样都会被存储在硬盘中，所以索引节点同样占用磁盘空间。
* 目录项，也就是 dentry，用来记录文件的名字、索引节点指针以及与其他目录项的层级关联关系。多个目录项关联起来，就会形成目录结构，但它与索引节点不同的是，目录项是由内核维护的一个数据结构，不存放于磁盘，而是缓存在内存。

目录项和索引节点的关系是多对一。注意，目录也是文件，也是用索引节点唯一标识，和普通文件不同的是，普通文件在磁盘里面保存的是文件数据，而目录文件在磁盘里面保存子目录或文件。目录是个文件，持久化存储在磁盘，而目录项是内核一个数据结构，缓存在内存。

磁盘读写的最小单位是扇区，扇区的大小只有 512B 大小，所以，文件系统把多个扇区组成了一个逻辑块，每次读写的最小单位就是逻辑块（数据块），Linux 中的逻辑块大小为 4KB，也就是一次性读写 8 个扇区，这将大大提高了磁盘的读写的效率。

**文件的使用**

打开了一个文件后，操作系统会跟踪进程打开的所有文件，所谓的跟踪呢，就是操作系统为每个进程维护一个打开文件表，文件表里的每一项代表「文件描述符」，所以说文件描述符是打开文件的标识。

而操作系统会在打开文件表中维护打开文件的状态信息： - 文件指针：系统跟踪上次读写位置作为当前文件位置指针，这种指针对打开文件的某个进程来说是唯一的； - 文件打开计数器：文件关闭时，操作系统必须重用其打开文件表条目，否则表内空间不够用。因为多个进程可能打开同一个文件，所以系统在删除打开文件条目之前，必须等待最后一个进程关闭文件，该计数器跟踪打开和关闭的数量，当该计数为 0 时，系统关闭文件，删除该条目； - 文件磁盘位置：绝大多数文件操作都要求系统修改文件数据，该信息保存在内存中，以免每个操作都从磁盘中读取； - 访问权限：每个进程打开文件都需要有一个访问模式（创建、只读、读写、添加等），该信息保存在进程的打开文件表中，以便操作系统能允许或拒绝之后的 I/O 请求；

**文件系统结构**

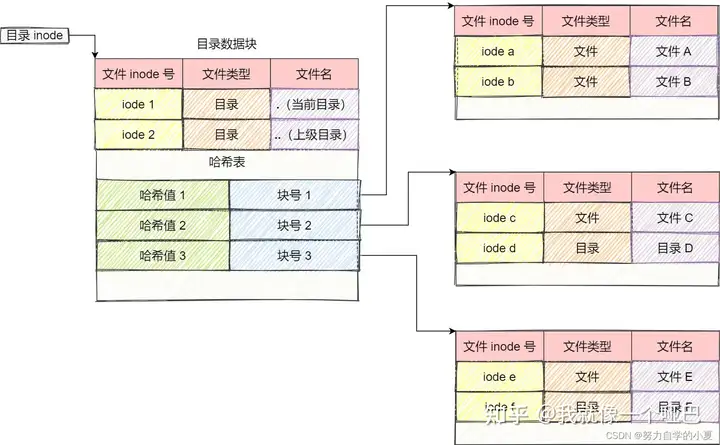
采用「一个块的位图 + 一系列的块」，外加「一个块的 inode 的位图 + 一系列的 inode 的结构」能表示的最大空间也就 128M。

在 Linux 文件系统，把这个结构称为一个块组，那么有 N 多的块组，就能够表示 N 大的文件。

**目录的存储**

普通文件的块里面保存的是文件数据，而目录文件的块里面保存的是目录里面一项一项的文件信息。

在目录文件的块中，最简单的保存格式就是列表，就是一项一项地将目录下的文件信息（如文件名、文件 inode、文件类型等）列在表里。列表中每一项就代表该目录下的文件的文件名和对应的 inode，通过这个 inode，就可以找到真正的文件。



**文件的逻辑结构**

**无结构文件**

流式文件，比如windows的txt文件

**有结构文件**

**顺序文件**

* 链式存储：只能从第一个往后找，可变长，没办法随即查找
* 顺序存储：定长，可以用折半等方法快速查找，它是可变长

**索引文件**

索引表是一个定长记录的顺序文件，用来记录索引文件，每一个文件对应一个索引项

**索引顺序文件**

每个记录对应一个索引表项，每一个索引表项对应一组文件

**文件的物理结构**

**连续分配**

分配连续的磁盘块，删除的时候可能会产生过多的碎片

**链接分配**

每一个盘块指向下一个盘块号，只能顺序访问（隐式分配）。建立一张文件分配表，显示记录盘块的先后关系，但是需要额外的空间来记录（显示分配），支持随机索引。

**索引分配**

为文件块建立索引表，如果太大，就用链接分配来结合使用。

**文件存储空间管理**

**空闲表法**

使用一张表来记录空闲空间

**空闲链表法**

空闲的块使用链表连接在一起

**位示图法**

每个二进制代表一个块号，对应记录空闲的块

**成组链接法**

将空闲块分组，每组记录空闲块号，并用链表连接起来

**文件共享**

**硬链接**

基于索引节点的共享方式，只有所有索引删除才会删除文件

**软链接**

基于符号链的共享方式，删除文件也不会删除链接

**磁盘**

磁盘分为磁片，磁片分为磁道，磁道分为扇区

**磁盘调度算法**

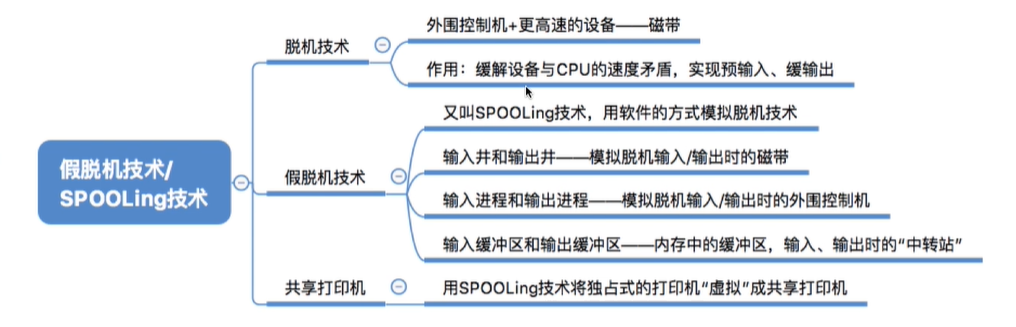
* **先来先服务：**按照请求到达磁盘调度器的顺序进行处理，先到达的请求的先被服务。容易出现饥饿问题，即一些后到的磁盘请求可能需要等待很长时间才能得到服务。
* **最短寻找时间优先：**优先选择距离当前磁头位置最近的请求进行服务。SSTF 算法能够最小化磁头的寻道时间，但容易出现饥饿问题，即磁头附近的请求不断被服务，远离磁头的请求长时间得不到响应。
* **扫描算法：**磁头沿着一个方向扫描磁盘，如果经过的磁道有请求就处理，直到到达磁盘的边界，然后改变移动方向，依此往复。SCAN 算法能够保证所有的请求得到服务，解决了饥饿问题。但是，如果磁头从一个方向刚扫描完，请求才到的话。这个请求就需要等到磁头从相反方向过来之后才能得到处理。
* **循环扫描算法：**只在磁盘的一侧进行扫描，并且只按照一个方向扫描，直到到达磁盘边界，然后回到磁盘起点，重新开始循环。
* **c-look调度算法：**C-LOOK 算法对 C-SCAN 算法进行了改进，如果磁头移动的方向上已经没有磁道访问请求了，就可以立即让磁头返回，并且磁头只需要返回到有磁道访问请求的位置即可。

**IO管理**

**IO控制方式**

* 程序直接控制方式
* 中断驱动方式
* DMA方式
* 通道控制方式





**常备系统知识**

**fork函数如何理解，exec函数如何理解**

fork函数是可以赋值当前进程，并创建一个子进程，子进程并从调用fork函数后的代码开始执行。该函数会返回两次，在父进程返回pid，子进程返回0。如果失败父进程返回-1。exec函数是立即生成子进程并取代父进程，并从main函数开始执行。

**read一个文件的全过程**

首先根据名字在系统的打开表中查找，找到了就在打开表为文件分一个表项，然后指向找到的表项。PCB为文件分配一个文件描述符再返回如果没有找到，就看a在不在内存，不在就把目录表装入内存，在目录表中找文件，找到后就把文件的FCB装入内存，系统打开表为文件建立一个表项，指向FCB，在进程文件打开表中分配新的一项，指向系统文件打开表，然后PCB为a分配一个文件描述符返回

**什么是僵尸进程，什么是孤儿进程**

在当前进程中生成一个子进程一般需要调用fork，但是fork需要返回两次。如果子进程比父进程先退出，父进程没有调用wait，则子进程的描述符停留在系统中就会变成僵尸进程。用PS查看进程状态为z则直接Kill。孤儿进程是因为父进程先比子进程退出，因为父进程并没有等待子进程退出，此时进程会被系统中的init进程管理，不会对系统照成危害。

**操作系统的锁一共有几种**

**互斥锁**

只有一个线程可以访问

**读写锁**

读进程之间不互斥，读写和写写互斥

**自旋锁**

只有一个线程可以访问，但是如果获取失败不会进入睡眠，而是占用 cpu 等待锁的释放

**操作系统相比 32 位，64 位的优势是什么**

32 位和 64 位指的是计算机一次可以处理更大的数据量。32 位有符号整型最大到 20 亿。32 位宽的 CPU 最多操作 232 个内存地址，也就是 4G 内存地址，64 位有更大的寻指空间。

**进程开销比线程大在哪里**

Linux 中创建一个进程自然会创建一个线程，也就是主线程。创建进程需要为进程划分出一块完整的内存空间，有大量的初始化操作，比如要把内存分段（堆栈、正文区等）。创建线程则简单得多，只需要确定 PC 指针和寄存器的值，并且给线程分配一个栈用于执行程序，同一个进程的多个线程间可以复用堆栈。因此，创建进程比创建线程慢，而且进程的内存开销更大

**Linux 中文件的权限**

读权限（r），写权限（w），执行权限（x）

文件权限一共有十位，第一位是文件类型，'-' 表示普通文件，'d' 表示目录，'p' 表示管道

2-4 表示用户权限，5-7 表示组的权限，8-10 表示所有用户权限

* - rwx rwx rwx

111 表示当前一组权限，转化为十进制就是7，所以权限全开就是 777

**Linux 常用的指令**

**Xargs**

批量处理工具，如重命名文件夹下 a.txt b.txt 为 pre\_a.txt pre\_b.txt

|  |
| --- |
| Plain Text ls | xargs -I GG echo "mv GG pre\_GG" |

**scp**

拷贝文件

**ssh**

连接远程服务器

**host**

查询 dns

**awk**

逐行分析日志

**less**

看文件的少部分数据，支持向上或者向下翻页

**head/tail**

看头部或者尾部多少行的数据

**cat**

查看文件里的所有内容

**堆和栈是否是连续的**

堆通常是不连续的，而栈是连续的。用于存储函数的局部变量、参数、返回地址等信息。栈是连续的内存区域，由编译器自动管理。

**线程在切换时的主要步骤**

1. 保存当前线程的上下文信息：操作系统会保存当前执行线程的[寄存器](https://www.baidu.com/s?wd=%E5%AF%84%E5%AD%98%E5%99%A8&usm=1&ie=utf-8&rsv_pq=feb54df101036e3e&oq=%E7%BA%BF%E7%A8%8B%E5%9C%A8%E5%88%87%E6%8D%A2%E7%9A%84%E6%97%B6%E5%80%99%2C%E8%A6%81%E5%81%9A%E7%9A%84%E4%BA%8B%E6%83%85&rsv_t=d1f6NMGExwhcZkfhyEE6KpIj0GZmqlhc9futtWIP9ogVam2RQx%2FR5knXSqQ&sa=re_dqa_generate)的值、[程序计数器](https://www.baidu.com/s?wd=%E7%A8%8B%E5%BA%8F%E8%AE%A1%E6%95%B0%E5%99%A8&usm=1&ie=utf-8&rsv_pq=feb54df101036e3e&oq=%E7%BA%BF%E7%A8%8B%E5%9C%A8%E5%88%87%E6%8D%A2%E7%9A%84%E6%97%B6%E5%80%99%2C%E8%A6%81%E5%81%9A%E7%9A%84%E4%BA%8B%E6%83%85&rsv_t=87b3JTV7YJS4FiI5Fk5yUMWHp4A6OWprZthI1CZM9dYhF5DTPJtMU%2BveepM&sa=re_dqa_generate)、堆栈指针和其他必要的状态信息到该线程的[控制块](https://www.baidu.com/s?wd=%E6%8E%A7%E5%88%B6%E5%9D%97&usm=1&ie=utf-8&rsv_pq=feb54df101036e3e&oq=%E7%BA%BF%E7%A8%8B%E5%9C%A8%E5%88%87%E6%8D%A2%E7%9A%84%E6%97%B6%E5%80%99%2C%E8%A6%81%E5%81%9A%E7%9A%84%E4%BA%8B%E6%83%85&rsv_t=0b32LbMJ4hkZBc29CMw58FVDztSI549L73ZH2iUur2jhA4OlGmv0OUruv6o&sa=re_dqa_generate)或[线程表](https://www.baidu.com/s?wd=%E7%BA%BF%E7%A8%8B%E8%A1%A8&usm=1&ie=utf-8&rsv_pq=feb54df101036e3e&oq=%E7%BA%BF%E7%A8%8B%E5%9C%A8%E5%88%87%E6%8D%A2%E7%9A%84%E6%97%B6%E5%80%99%2C%E8%A6%81%E5%81%9A%E7%9A%84%E4%BA%8B%E6%83%85&rsv_t=b6c0d3peMLcktMkxsK7WKn0VASTTW%2BwKSYtBu6PylP5EXKsp1WldMolnerg&sa=re_dqa_generate)中。
2. 加载下一个线程的上下文信息：操作系统从待执行线程的控制块或线程表中获取该线程的上下文信息，将寄存器的值、程序计数器、堆栈指针和其他状态信息加载到[CPU](https://www.baidu.com/s?wd=CPU&usm=1&ie=utf-8&rsv_pq=feb54df101036e3e&oq=%E7%BA%BF%E7%A8%8B%E5%9C%A8%E5%88%87%E6%8D%A2%E7%9A%84%E6%97%B6%E5%80%99%2C%E8%A6%81%E5%81%9A%E7%9A%84%E4%BA%8B%E6%83%85&rsv_t=a95daWe9YF9KwjMh2HWAqRgHqJ%2FJBMdHW5dOS9lVmv5nNchfgwgNJ1ywxaM&sa=re_dqa_generate)中。
3. 切换堆栈：当线程切换时，堆栈指针会指向不同的栈空间，操作系统需要将从当前线程的用户栈切换到即将执行线程的用户栈。
4. 更新页表和地址空间：如果切换的线程使用了虚拟内存技术，操作系统需要更新页表和地址空间的映射关系。
5. 执行下一个线程：最后，操作系统将控制权转交给即将执行的线程，该线程从切换前保存的上下文信息恢复执行。

**线程切换的开销**

线程切换是一个相对轻量级的操作，因为它不需要像进程切换那样保存和恢复整个进程的上下文信息。然而，即使如此，线程切换仍然会有一定的开销，主要包括保存和加载寄存器值、更新堆栈和页表等操作的时间消耗。在实际应用中，频繁的线程切换可能会影响性能，尤其是在高并发的场景下。

**写文件时，从内存到磁盘步骤**

1. 写入缓存：当应用程序向文件系统发出写入请求时，数据会首先被写入到系统内核的缓存中，也称为页缓存（page cache）。这个缓存位于内存中，它包含了即将写入磁盘的数据。
2. 写入磁盘缓存：数据被写入到磁盘缓存（disk cache）中，这是一个硬盘内部的缓存区域。数据在写入到磁盘缓存之前可能会被重新排序、合并或延迟写入，以提高性能和减少磁盘I/O操作的次数。
3. 写入磁盘控制器：一旦数据被写入磁盘缓存，操作系统会通知磁盘控制器开始将数据从磁盘缓存写入到实际的磁盘存储设备中。磁盘控制器负责将数据传输到硬盘的正确位置，并确保数据的完整性和一致性。
4. 确认写入：一旦数据成功写入磁盘存储设备，操作系统会收到确认写入的通知。此时，该文件的数据已经被永久地保存在磁盘中，并且可以保证数据的持久性。

需要注意的是，Linux操作系统具有缓存机制，其中包括页缓存和磁盘缓存等，这些缓存会提高文件系统的性能和速度。但是，在某些情况下，如果系统发生异常关机或崩溃，可能会导致部分数据丢失或损坏。因此，在关键数据的写入过程中，建议使用适当的数据同步和持久性保证机制，如fsync()函数或O\_SYNC标志，以确保数据的完整性和可靠性。

采用默认方法，则在以下几种情况会触发内核缓存写入磁盘： - 在调用 write 的最后，当发现内核缓存的数据太多的时候，内核会把数据写到磁盘上； - 用户主动调用 sync，内核缓存会刷到磁盘上； - 当内存十分紧张，无法再分配页面时，也会把内核缓存的数据刷到磁盘上； - 内核缓存的数据的缓存时间超过某个时间时，也会把数据刷到磁盘上；

**零拷贝**

零拷贝一共 2 种实现方法： - mmap + write - sendfile

* mmap + write

read() 系统调用的过程中会把内核缓冲区的数据拷贝到用户的缓冲区里，于是为了减少这一步开销，我们可以用 mmap() 替换 read() 系统调用函数。

|  |
| --- |
| C buf = mmap(file, len);write(sockfd, buf, len); |

mmap() 系统调用函数会直接把内核缓冲区里的数据「映射」到用户空间，这样，操作系统内核与用户空间就不需要再进行任何的数据拷贝操作。此时 mmap() 通过 DMA 把磁盘数据拷贝到内核缓冲区并与进程共享缓冲区，之后调用 write() 那么操作系统会直接将内核缓冲区数据拷贝到 socket 缓冲区，最后由 DMA 把 socket 缓冲区数据拷贝到网卡缓冲区。那么 mmap() 替换 read() 就可以减少一次数据拷贝。

* sendfile

在 LInux 内核 2.1 中，提供了专门发送文件的系统调用 sendfile() 如下：

|  |
| --- |
| C #include <sys/socket.h> ssize\_t sendfile(int out\_fd, int in\_fd, off\_t \*offset, size\_t count); |

可以替代前面的 read() 和 write() 这两个系统调用，这样就可以减少一次系统调用，也就减少了 2 次上下文切换的开销。并且，该系统调用，可以直接把内核缓冲区里的数据拷贝到 socket 缓冲区里，不再拷贝到用户态，这样就只有 2 次上下文切换，和 3 次数据拷贝。

**为什么会有线程池，到底解决了什么问题**

1. 减少线程的创建与销毁（线程的角度）
2. [异步解耦](https://zhida.zhihu.com/search?q=%E5%BC%82%E6%AD%A5%E8%A7%A3%E8%80%A6&zhida_source=entity&is_preview=1)的作用（设计的角度）

**在linux下 OOM 日志**

OOM 指的是 Out of Memory，即内存溢出。在 Linux 系统中，当系统没有足够的内存来满足请求时，会触发 OOM Killer，它会选择并杀死一些进程以释放内存。

OOM 日志通常记录在 /var/log/messages 或者 /var/log/syslog 文件中，格式如

**线程的应用场景和协程的应用场景**

协程的应用场景如下：

* 并发处理：协程可以处理大量的并发任务，比传统的线程模型更符合高并发场景的需求。
* 非阻塞IO：协程可以利用非阻塞IO技术，在等待IO操作时切换到其他协程，提高程序的响应性能。
* 微服务：协程可以被用于构建高性能的微服务架构，处理大量的请求。

线程的应用场景如下：

* CPU密集型任务：对于需要大量计算的任务，使用多线程可以充分利用多核处理器提升计算性能。
* 并发IO：对于IO密集型的任务，使用多线程可以提高IO的效率，缩短等待时间。
* 阻塞IO：在需要进行阻塞IO操作时，线程可以等待IO完成再继续执行。

**linux锁的底层实现机制**

Linux锁的底层实现机制依赖于硬件和使用的具体锁类型。在最底层，Linux 使用原子操作来实现自旋锁（spinlock），而对于读写锁（rwlock），则使用了一种称为“读优先”或“写优先”的策略。

自旋锁通常用于 BH 锁（软中断锁），原理是通过原子操作来确保在同一时刻只有一个核心可以获得自旋锁。如果锁已经被占用，那么获取锁的核心将在一个小循环内不停地尝试获取锁，直到锁变为可用为止。

读写锁实现了锁的分级，允许多个读者同时持有锁，但在同一时刻只允许一个写者或者写者与读者同时。实现方式可能包括使用一个计数器来跟踪读者和写者的数量，以及一个队列来管理等待的进程。

**并发编程可能会遇到的问题**

1.数据同步

2.死锁

**同步io和非同步io，阻塞io和非阻塞io有什么区别**

IO读取数据的过程，内核将数据准备好到内核缓冲区，内核将数据拷贝到用户态，进程首先和内核打交道，之后内核再和硬件（如网卡）打交道

**阻塞IO**，无论用户进程和内核打交道还是内核和硬件打交道，这两个过程都阻塞，也就是说，进程要等待内核将数据在内核缓冲区准备好（第一个阶段），并且还要等待内核缓冲区的数据拷贝到用户态（第二个阶段）时才返回

**非阻塞IO** 是指在第一个过程里不阻塞等待内核的结果，而是采用轮询的方式不断询问内核缓冲区中是否有数据，直到内核中有数据后，就在第二个阶段进行阻塞，保证数据成功从内核拷贝到用户

**同步IO** 是指当进程发起IO后，必须等待IO操作完成 无论是阻塞IO还是非阻塞IO都是同步IO，因为二者都需要在第二个阶段里阻塞，等待内核将数据拷贝到用户，由于进程必须在第二个阶段等待，那么也就是说，进程在io的整个过程里都必须要等待内核将数据拷贝到用户才算成功，而在这个过程里，进程是不能去做其他事情的，无论你是轮询内核也好，等待内核也好

**异步IO**是指，进程在调用io操作后，我无需关注结果在什么时候返回，届时你准备好了数据给我就可以了（通过回调机制完成），这个过程里我可以去做别的事情

**设计消息缓存机制**

设计消息缓存机制以应对高并发请求，需要综合考虑缓存的使用场景、缓存策略、以及缓存与数据库之间的同步机制。以下是一个设计思路的实现方案：

1. 缓存使用场景‌：

* 消息缓存应适用于所有需要频繁读取但更新不频繁的消息数据。例如，系统通知、新闻更新等。
* 在消息从数据库读取后，应将其缓存起来，以便快速响应后续的读取请求，减少数据库访问，从而提高处理效率。

1. 缓存策略‌：

* 采用[分布式缓存技术](https://www.baidu.com/s?sa=re_dqa_generate&wd=%E5%88%86%E5%B8%83%E5%BC%8F%E7%BC%93%E5%AD%98%E6%8A%80%E6%9C%AF&rsv_pq=e151d80f027e87db&oq=%E8%AE%A9%E4%BD%A0%E8%AE%BE%E8%AE%A1%E4%B8%80%E7%A7%8D%E6%B6%88%E6%81%AF%E7%BC%93%E5%AD%98%E6%9C%BA%E5%88%B6%2C%E4%BD%A0%E5%A6%82%E4%BD%95%E8%AE%BE%E8%AE%A1%3F%E5%A6%82%E4%BD%95%E5%AE%9E%E7%8E%B0%E9%AB%98%E5%B9%B6%E5%8F%91%3F&rsv_t=a9edNWvIKKcUXEzI7SCLeyKj8qdVCuMrha3CiYHgISUYhT1AfPXS2p2kbBQ&tn=baidu&ie=utf-8)，如[Redis](https://www.baidu.com/s?sa=re_dqa_generate&wd=Redis&rsv_pq=e151d80f027e87db&oq=%E8%AE%A9%E4%BD%A0%E8%AE%BE%E8%AE%A1%E4%B8%80%E7%A7%8D%E6%B6%88%E6%81%AF%E7%BC%93%E5%AD%98%E6%9C%BA%E5%88%B6%2C%E4%BD%A0%E5%A6%82%E4%BD%95%E8%AE%BE%E8%AE%A1%3F%E5%A6%82%E4%BD%95%E5%AE%9E%E7%8E%B0%E9%AB%98%E5%B9%B6%E5%8F%91%3F&rsv_t=a9edNWvIKKcUXEzI7SCLeyKj8qdVCuMrha3CiYHgISUYhT1AfPXS2p2kbBQ&tn=baidu&ie=utf-8)或[Memcached](https://www.baidu.com/s?sa=re_dqa_generate&wd=Memcached&rsv_pq=e151d80f027e87db&oq=%E8%AE%A9%E4%BD%A0%E8%AE%BE%E8%AE%A1%E4%B8%80%E7%A7%8D%E6%B6%88%E6%81%AF%E7%BC%93%E5%AD%98%E6%9C%BA%E5%88%B6%2C%E4%BD%A0%E5%A6%82%E4%BD%95%E8%AE%BE%E8%AE%A1%3F%E5%A6%82%E4%BD%95%E5%AE%9E%E7%8E%B0%E9%AB%98%E5%B9%B6%E5%8F%91%3F&rsv_t=a9edNWvIKKcUXEzI7SCLeyKj8qdVCuMrha3CiYHgISUYhT1AfPXS2p2kbBQ&tn=baidu&ie=utf-8)，以分散存储和访问压力，提高缓存的可用性和扩展性。
* 设置合理的缓存失效时间，避免数据长时间不更新导致缓存中的数据与数据库中的数据不一致。可以通过设置超时时间或采用[缓存淘汰策略](https://www.baidu.com/s?sa=re_dqa_generate&wd=%E7%BC%93%E5%AD%98%E6%B7%98%E6%B1%B0%E7%AD%96%E7%95%A5&rsv_pq=e151d80f027e87db&oq=%E8%AE%A9%E4%BD%A0%E8%AE%BE%E8%AE%A1%E4%B8%80%E7%A7%8D%E6%B6%88%E6%81%AF%E7%BC%93%E5%AD%98%E6%9C%BA%E5%88%B6%2C%E4%BD%A0%E5%A6%82%E4%BD%95%E8%AE%BE%E8%AE%A1%3F%E5%A6%82%E4%BD%95%E5%AE%9E%E7%8E%B0%E9%AB%98%E5%B9%B6%E5%8F%91%3F&rsv_t=a9edNWvIKKcUXEzI7SCLeyKj8qdVCuMrha3CiYHgISUYhT1AfPXS2p2kbBQ&tn=baidu&ie=utf-8)来实现。
* 对于需要频繁更新的数据，可以采用[缓存更新通知机制](https://www.baidu.com/s?sa=re_dqa_generate&wd=%E7%BC%93%E5%AD%98%E6%9B%B4%E6%96%B0%E9%80%9A%E7%9F%A5%E6%9C%BA%E5%88%B6&rsv_pq=e151d80f027e87db&oq=%E8%AE%A9%E4%BD%A0%E8%AE%BE%E8%AE%A1%E4%B8%80%E7%A7%8D%E6%B6%88%E6%81%AF%E7%BC%93%E5%AD%98%E6%9C%BA%E5%88%B6%2C%E4%BD%A0%E5%A6%82%E4%BD%95%E8%AE%BE%E8%AE%A1%3F%E5%A6%82%E4%BD%95%E5%AE%9E%E7%8E%B0%E9%AB%98%E5%B9%B6%E5%8F%91%3F&rsv_t=a9edNWvIKKcUXEzI7SCLeyKj8qdVCuMrha3CiYHgISUYhT1AfPXS2p2kbBQ&tn=baidu&ie=utf-8)，即当数据库中的数据发生变化时，通知缓存系统更新相应数据，以保证缓存与数据库数据的一致性。

1. 缓存与数据库同步机制‌：

* 采用定时任务或[消息队列](https://www.baidu.com/s?sa=re_dqa_generate&wd=%E6%B6%88%E6%81%AF%E9%98%9F%E5%88%97&rsv_pq=e151d80f027e87db&oq=%E8%AE%A9%E4%BD%A0%E8%AE%BE%E8%AE%A1%E4%B8%80%E7%A7%8D%E6%B6%88%E6%81%AF%E7%BC%93%E5%AD%98%E6%9C%BA%E5%88%B6%2C%E4%BD%A0%E5%A6%82%E4%BD%95%E8%AE%BE%E8%AE%A1%3F%E5%A6%82%E4%BD%95%E5%AE%9E%E7%8E%B0%E9%AB%98%E5%B9%B6%E5%8F%91%3F&rsv_t=a9edNWvIKKcUXEzI7SCLeyKj8qdVCuMrha3CiYHgISUYhT1AfPXS2p2kbBQ&tn=baidu&ie=utf-8)来定期同步数据库与缓存中的数据，确保数据的实时性和准确性。
* 对于写入操作，可以先更新数据库，然后通知缓存系统更新相应数据，以保证数据的最新状态在缓存中可用。
* 采用[事务机制](https://www.baidu.com/s?sa=re_dqa_generate&wd=%E4%BA%8B%E5%8A%A1%E6%9C%BA%E5%88%B6&rsv_pq=e151d80f027e87db&oq=%E8%AE%A9%E4%BD%A0%E8%AE%BE%E8%AE%A1%E4%B8%80%E7%A7%8D%E6%B6%88%E6%81%AF%E7%BC%93%E5%AD%98%E6%9C%BA%E5%88%B6%2C%E4%BD%A0%E5%A6%82%E4%BD%95%E8%AE%BE%E8%AE%A1%3F%E5%A6%82%E4%BD%95%E5%AE%9E%E7%8E%B0%E9%AB%98%E5%B9%B6%E5%8F%91%3F&rsv_t=a9edNWvIKKcUXEzI7SCLeyKj8qdVCuMrha3CiYHgISUYhT1AfPXS2p2kbBQ&tn=baidu&ie=utf-8)来保证数据库与缓存操作的一致性，即在事务中先更新数据库，然后更新缓存，如果数据库更新失败，则缓存更新也回滚，反之亦然。

1. 高并发处理‌：

* 利用[负载均衡技术](https://www.baidu.com/s?sa=re_dqa_generate&wd=%E8%B4%9F%E8%BD%BD%E5%9D%87%E8%A1%A1%E6%8A%80%E6%9C%AF&rsv_pq=e151d80f027e87db&oq=%E8%AE%A9%E4%BD%A0%E8%AE%BE%E8%AE%A1%E4%B8%80%E7%A7%8D%E6%B6%88%E6%81%AF%E7%BC%93%E5%AD%98%E6%9C%BA%E5%88%B6%2C%E4%BD%A0%E5%A6%82%E4%BD%95%E8%AE%BE%E8%AE%A1%3F%E5%A6%82%E4%BD%95%E5%AE%9E%E7%8E%B0%E9%AB%98%E5%B9%B6%E5%8F%91%3F&rsv_t=a9edNWvIKKcUXEzI7SCLeyKj8qdVCuMrha3CiYHgISUYhT1AfPXS2p2kbBQ&tn=baidu&ie=utf-8)来分散请求压力，通过多个缓存服务器共同处理请求，提高系统的并发处理能力。
* 采用[异步处理](https://www.baidu.com/s?sa=re_dqa_generate&wd=%E5%BC%82%E6%AD%A5%E5%A4%84%E7%90%86&rsv_pq=e151d80f027e87db&oq=%E8%AE%A9%E4%BD%A0%E8%AE%BE%E8%AE%A1%E4%B8%80%E7%A7%8D%E6%B6%88%E6%81%AF%E7%BC%93%E5%AD%98%E6%9C%BA%E5%88%B6%2C%E4%BD%A0%E5%A6%82%E4%BD%95%E8%AE%BE%E8%AE%A1%3F%E5%A6%82%E4%BD%95%E5%AE%9E%E7%8E%B0%E9%AB%98%E5%B9%B6%E5%8F%91%3F&rsv_t=a9edNWvIKKcUXEzI7SCLeyKj8qdVCuMrha3CiYHgISUYhT1AfPXS2p2kbBQ&tn=baidu&ie=utf-8)方式，将请求先放入队列中，然后由后台服务异步处理，避免因等待处理而阻塞主线程，从而提高系统的响应速度和并发处理能力。
* 对缓存进行水平扩展，通过增加更多的缓存服务器来分担请求压力，同时采用[负载均衡算法](https://www.baidu.com/s?sa=re_dqa_generate&wd=%E8%B4%9F%E8%BD%BD%E5%9D%87%E8%A1%A1%E7%AE%97%E6%B3%95&rsv_pq=e151d80f027e87db&oq=%E8%AE%A9%E4%BD%A0%E8%AE%BE%E8%AE%A1%E4%B8%80%E7%A7%8D%E6%B6%88%E6%81%AF%E7%BC%93%E5%AD%98%E6%9C%BA%E5%88%B6%2C%E4%BD%A0%E5%A6%82%E4%BD%95%E8%AE%BE%E8%AE%A1%3F%E5%A6%82%E4%BD%95%E5%AE%9E%E7%8E%B0%E9%AB%98%E5%B9%B6%E5%8F%91%3F&rsv_t=a9edNWvIKKcUXEzI7SCLeyKj8qdVCuMrha3CiYHgISUYhT1AfPXS2p2kbBQ&tn=baidu&ie=utf-8)来合理分配请求到各个缓存服务器上，避免单点压力过大。

通过上述设计思路，可以构建一个高效、可扩展的消息缓存机制，以应对高并发请求的挑战。

**进程和线程共享什么，不共享什么**

进程和线程共享的东西包括[代码段](https://www.baidu.com/s?sa=re_dqa_generate&wd=%E4%BB%A3%E7%A0%81%E6%AE%B5&rsv_pq=d603f6090081885d&oq=%E8%BF%9B%E7%A8%8B%E5%92%8C%E7%BA%BF%E7%A8%8B%E5%85%B1%E4%BA%AB%E4%BB%80%E4%B9%88%E4%B8%9C%E8%A5%BF%E4%BB%A5%E5%8F%8A%E4%B8%8D%E5%85%B1%E4%BA%AB%E4%BB%80%E4%B9%88%E5%86%85%E5%AE%B9&rsv_t=f1f4b08Fa7zziqawE7yhvP5DDB/w7hyf+sGzpAdlW/Qqjvdu2eJdOYmAvOQ&tn=baidu&ie=utf-8)、[数据段](https://www.baidu.com/s?sa=re_dqa_generate&wd=%E6%95%B0%E6%8D%AE%E6%AE%B5&rsv_pq=d603f6090081885d&oq=%E8%BF%9B%E7%A8%8B%E5%92%8C%E7%BA%BF%E7%A8%8B%E5%85%B1%E4%BA%AB%E4%BB%80%E4%B9%88%E4%B8%9C%E8%A5%BF%E4%BB%A5%E5%8F%8A%E4%B8%8D%E5%85%B1%E4%BA%AB%E4%BB%80%E4%B9%88%E5%86%85%E5%AE%B9&rsv_t=f1f4b08Fa7zziqawE7yhvP5DDB/w7hyf+sGzpAdlW/Qqjvdu2eJdOYmAvOQ&tn=baidu&ie=utf-8)、[堆](https://www.baidu.com/s?sa=re_dqa_generate&wd=%E5%A0%86&rsv_pq=d603f6090081885d&oq=%E8%BF%9B%E7%A8%8B%E5%92%8C%E7%BA%BF%E7%A8%8B%E5%85%B1%E4%BA%AB%E4%BB%80%E4%B9%88%E4%B8%9C%E8%A5%BF%E4%BB%A5%E5%8F%8A%E4%B8%8D%E5%85%B1%E4%BA%AB%E4%BB%80%E4%B9%88%E5%86%85%E5%AE%B9&rsv_t=f1f4b08Fa7zziqawE7yhvP5DDB/w7hyf+sGzpAdlW/Qqjvdu2eJdOYmAvOQ&tn=baidu&ie=utf-8)、[文件描述符](https://www.baidu.com/s?sa=re_dqa_generate&wd=%E6%96%87%E4%BB%B6%E6%8F%8F%E8%BF%B0%E7%AC%A6&rsv_pq=d603f6090081885d&oq=%E8%BF%9B%E7%A8%8B%E5%92%8C%E7%BA%BF%E7%A8%8B%E5%85%B1%E4%BA%AB%E4%BB%80%E4%B9%88%E4%B8%9C%E8%A5%BF%E4%BB%A5%E5%8F%8A%E4%B8%8D%E5%85%B1%E4%BA%AB%E4%BB%80%E4%B9%88%E5%86%85%E5%AE%B9&rsv_t=f1f4b08Fa7zziqawE7yhvP5DDB/w7hyf+sGzpAdlW/Qqjvdu2eJdOYmAvOQ&tn=baidu&ie=utf-8)、[套接字](https://www.baidu.com/s?sa=re_dqa_generate&wd=%E5%A5%97%E6%8E%A5%E5%AD%97&rsv_pq=d603f6090081885d&oq=%E8%BF%9B%E7%A8%8B%E5%92%8C%E7%BA%BF%E7%A8%8B%E5%85%B1%E4%BA%AB%E4%BB%80%E4%B9%88%E4%B8%9C%E8%A5%BF%E4%BB%A5%E5%8F%8A%E4%B8%8D%E5%85%B1%E4%BA%AB%E4%BB%80%E4%B9%88%E5%86%85%E5%AE%B9&rsv_t=f1f4b08Fa7zziqawE7yhvP5DDB/w7hyf+sGzpAdlW/Qqjvdu2eJdOYmAvOQ&tn=baidu&ie=utf-8)和[端口](https://www.baidu.com/s?sa=re_dqa_generate&wd=%E7%AB%AF%E5%8F%A3&rsv_pq=d603f6090081885d&oq=%E8%BF%9B%E7%A8%8B%E5%92%8C%E7%BA%BF%E7%A8%8B%E5%85%B1%E4%BA%AB%E4%BB%80%E4%B9%88%E4%B8%9C%E8%A5%BF%E4%BB%A5%E5%8F%8A%E4%B8%8D%E5%85%B1%E4%BA%AB%E4%BB%80%E4%B9%88%E5%86%85%E5%AE%B9&rsv_t=f1f4b08Fa7zziqawE7yhvP5DDB/w7hyf+sGzpAdlW/Qqjvdu2eJdOYmAvOQ&tn=baidu&ie=utf-8)、[进程标识信息](https://www.baidu.com/s?sa=re_dqa_generate&wd=%E8%BF%9B%E7%A8%8B%E6%A0%87%E8%AF%86%E4%BF%A1%E6%81%AF&rsv_pq=d603f6090081885d&oq=%E8%BF%9B%E7%A8%8B%E5%92%8C%E7%BA%BF%E7%A8%8B%E5%85%B1%E4%BA%AB%E4%BB%80%E4%B9%88%E4%B8%9C%E8%A5%BF%E4%BB%A5%E5%8F%8A%E4%B8%8D%E5%85%B1%E4%BA%AB%E4%BB%80%E4%B9%88%E5%86%85%E5%AE%B9&rsv_t=f1f4b08Fa7zziqawE7yhvP5DDB/w7hyf+sGzpAdlW/Qqjvdu2eJdOYmAvOQ&tn=baidu&ie=utf-8)、[信号处理器](https://www.baidu.com/s?sa=re_dqa_generate&wd=%E4%BF%A1%E5%8F%B7%E5%A4%84%E7%90%86%E5%99%A8&rsv_pq=d603f6090081885d&oq=%E8%BF%9B%E7%A8%8B%E5%92%8C%E7%BA%BF%E7%A8%8B%E5%85%B1%E4%BA%AB%E4%BB%80%E4%B9%88%E4%B8%9C%E8%A5%BF%E4%BB%A5%E5%8F%8A%E4%B8%8D%E5%85%B1%E4%BA%AB%E4%BB%80%E4%B9%88%E5%86%85%E5%AE%B9&rsv_t=f1f4b08Fa7zziqawE7yhvP5DDB/w7hyf+sGzpAdlW/Qqjvdu2eJdOYmAvOQ&tn=baidu&ie=utf-8)、[环境变量](https://www.baidu.com/s?sa=re_dqa_generate&wd=%E7%8E%AF%E5%A2%83%E5%8F%98%E9%87%8F&rsv_pq=d603f6090081885d&oq=%E8%BF%9B%E7%A8%8B%E5%92%8C%E7%BA%BF%E7%A8%8B%E5%85%B1%E4%BA%AB%E4%BB%80%E4%B9%88%E4%B8%9C%E8%A5%BF%E4%BB%A5%E5%8F%8A%E4%B8%8D%E5%85%B1%E4%BA%AB%E4%BB%80%E4%B9%88%E5%86%85%E5%AE%B9&rsv_t=f1f4b08Fa7zziqawE7yhvP5DDB/w7hyf+sGzpAdlW/Qqjvdu2eJdOYmAvOQ&tn=baidu&ie=utf-8)、[程序计数器](https://www.baidu.com/s?sa=re_dqa_generate&wd=%E7%A8%8B%E5%BA%8F%E8%AE%A1%E6%95%B0%E5%99%A8&rsv_pq=d603f6090081885d&oq=%E8%BF%9B%E7%A8%8B%E5%92%8C%E7%BA%BF%E7%A8%8B%E5%85%B1%E4%BA%AB%E4%BB%80%E4%B9%88%E4%B8%9C%E8%A5%BF%E4%BB%A5%E5%8F%8A%E4%B8%8D%E5%85%B1%E4%BA%AB%E4%BB%80%E4%B9%88%E5%86%85%E5%AE%B9&rsv_t=f1f4b08Fa7zziqawE7yhvP5DDB/w7hyf+sGzpAdlW/Qqjvdu2eJdOYmAvOQ&tn=baidu&ie=utf-8)。这些共享资源使得多个线程能够在同一进程中并发执行，实现高效的资源利用和并行处理。

然而，进程和线程之间也有不共享的内容，包括：

* 线程ID‌：每个线程都有唯一的标识，用于在进程中区分不同的线程。
* 寄存器组的值‌：由于线程间是并发运行的，每个线程有自己的运行线索，寄存器组的状态必须在线程切换时保存和恢复。
* 线程的堆栈‌：保证线程独立运行所必需，线程必须拥有自己的函数堆栈，以便函数调用可以正常执行，不受其他线程的影响。
* 错误返回码‌：由于多个线程可能同时运行，每个线程应该有其自己的错误返回码变量，以避免错误值被意外修改。
* 线程的信号屏蔽码‌：由于每个线程感兴趣的信号不同，线程的信号屏蔽码应该由线程自己管理。
* 线程的优先级‌：线程需要有自己的优先级，以便像进程一样被调度。

这些不共享的内容确保了线程的独立性和互操作性，使得多线程应用程序能够更加灵活和可靠地运行12。