操作系统 3

jask

09/14/2024

磁盘 IO

零拷贝

DMA 技术：在进行 IO 设备和内存的数据传输的时候，数据搬运工作全部交给 DMA 控制器，即 CPU 不再参与与任何数据搬运相关的事情，让 CPU 处理别的事情

实现

mmap+write：mmap 系统调用函数直接把内核缓冲区里面的数据映射到用户空间，这样操作系统和内核与用户空间就不需要任何数据拷贝操作

sendfile：可以替代 read 和 write 两个调用，减少一次系统开销

kafka 和 nginx 都是零拷贝技术实现的项目之一

提升：pagecache

pagecache 使用了预读功能，来缓存最近被访问的数据

在传输大文件的时候不会用，主要用来传输小文件

对于大文件简单的调用 read，通过异步 IO 和直接 IO 解决

IO 多路复用

socket 编程

首先调用 socket 函数，创建网络协议，和 tcp 的 socket

接着调用 bind 函数，给 socket 绑定一个 ip 地址和端口：

1. 绑定端口的目的：当内核受到 TCP 报文，通过 TCP 头里面的端口号，来找到应用程序传输数据

2. 绑定 IP 的目的：一个主机可能对应多个网卡，每个网卡都有对应的 IP，绑定 IP 才能在内核上收到对应网卡的包

接着调用 listen 函数监听

进入监听状态后调用 accept，来从内核获取客户端的连接

客户端在创建好 socket 以后，调用 connect 函数发起连接指明 IP 和端口号，开始三次握手

在 TCP 连接的时候，服务器内核实际上创建了两个队列：

1.TCP 半连接队列，指的是没有完成三次握手的连接，此时服务端出于 syn-rcvd 状态

2.TCP 全连接队列，指的是已经完成三次握手的连接，此时服务器处于 established 状态

当 TCP 全连接队列不为空，服务端的 accept 函数，就会从内核中的 TCP 全连接队列里拿出一个已经完成连接的 socket 返回应用程序，后续数据传输都用这个 socket

监听的 socket 和已经连接的 socket 是两个不一样的 socket

多进程，多线程

多进程 主进程监听客户端的连接，一旦连接完成,accept 函数就会返回一个已连接的 socket，通过 fork 函数创建一个子进程，把父进程所有相关的东西都复制一份，根据返回值来区分父进程和子进程

因为子进程会复制父进程的文件描述符，因此可以直接使用已连接的 socket 来进行通信了

当其退出的时候，如果做不好回收的工作，就会变成僵尸进程，会慢慢消耗我们的资源

多线程 多线程相比多进程的优点就是多线程可以共享资源部分，相对比多进程开销小得多，但是涉及到上下文切换，如果过于频繁也会不大行

可以使用线程池的方式避免线程的频繁销毁

当然因为线程池是全局的，这个时候操作线程池是需要加锁的

IO 多路复用

select 原理：通过传入一组文件描述符集，内核检查这些描述符的状态变化（如是否可读、可写等），并将就绪的描述符返回给应用程序

特点：

可移植性强：select 是 POSIX 标准的一部分，几乎在所有操作系统上都支持。

限制：select 对文件描述符的数量有限制（通常为 1024），不适合处理大量连接。

效率问题：每次调用 select 时，应用程序都需要重新传递文件描述符集，内核需要遍历整个集合，导致性能瓶颈

poll 原理：poll 使用一个结构体数组（struct pollfd）来监视多个文件描述符，并在这些描述符上设置感兴趣的事件。

特点：

无文件描述符数量限制：相比 select，poll 不限制文件描述符的数量，适合处理更多连接。
性能提升：poll 使用链表或数组存储文件描述符，避免了 select 的位图操作，但在高并发情况下性能仍有限。
一次性传递：每次调用 poll 都需要传递整个文件描述符数组，仍然存在遍历开销。

epoll 原理：epoll 是 Linux 特有的高效 IO 多路复用机制，通过内核维护一个事件表（epoll 实例），应用程序可以向内核注册感兴趣的文件描述符和事件。内核会在文件描述符状态变化时通知应用程序。

特点：

高效处理大规模连接：epoll 使用事件驱动模型，避免了每次调用时遍历所有文件描述符的开销，特别适合高并发场景。
支持边缘触发（Edge-Triggered）和水平触发（Level-Triggered）：
    水平触发（LT）：默认模式，类似 select 和 poll，只要文件描述符可读可写，就会不断通知。
    边缘触发（ET）：仅在状态发生变化时通知，减少了不必要的通知次数，提高了效率，但需要应用程序以非阻塞方式处理。
持久化事件：一旦注册，文件描述符在事件表中持续存在，除非显式移除，减少了频繁的注册操作。

API 主要函数：

epoll_create1(int flags)：创建一个 epoll 实例。
epoll_ctl(int epoll_fd, int op, int fd, struct epoll_event *event)：控制 epoll 实例，添加、修改或删除文件描述符。
epoll_wait(int epoll_fd, struct epoll_event *events, int maxevents, int timeout)：等待事件发生。

注意，epoll 并不是异步的

同步与异步的区分

同步：调用方主动发起 I/O 操作，并等待或者轮询 I/O 结果。epoll 就是同步的，因为应用程序还是需要主动调用 epoll_wai
异步：调用方不需要主动等待结果，系统会在 I/O 操作完成后通过回调或信号的方式通知调用方。

虽然 epoll 使用事件通知机制（即注册感兴趣的事件后，程序会在事件发生时被唤醒），但在等待事件的过程中，程序还是需要调用 epoll_wait()，这意味着程序处于同步等待状态——也就是说，它在等待事件发生的时候还是阻塞的。

AIO

Posix AIO POSIX AIO 是符合 POSIX 标准的异步 I/O 接口，主要通过 aio_read()、aio_write()、aio_error() 和 aio_return() 等函数提供异步读写能力。特点：

POSIX 标准：POSIX AIO 是 POSIX 标准的一部分，旨在为各种 UNIX 系统（包括 Linux、BSD、Solaris 等）提供一致的异步 I
实现方式：在 Linux 上，POSIX AIO 通常通过线程模拟异步 I/O，即当你调用 aio_read() 或 aio_write() 时，内核可能会在
线程开销：由于 POSIX AIO 在 Linux 上依赖于后台线程池，因此它的性能在处理大量并发请求时可能会受到影响。线程创建和
阻塞问题：由于其本质上依赖线程，某些系统调用（如 open）仍然会是阻塞的，这与完全异步的 I/O 不同。

优点：

兼容性好：POSIX AIO 是跨平台的，可以在不同的 UNIX 系统上使用。
提供了熟悉的 POSIX API，对于已经依赖 POSIX 的系统来说，可以轻松集成。

缺点：

由于使用线程模拟异步操作，因此无法充分发挥异步 I/O 的全部优势，性能相对较差。
并不完全支持所有类型的异步 I/O 操作（例如，文件打开操作仍然是阻塞的）。

Kernel AIO Kernel AIO 是 Linux 特有的、完全基于内核实现的异步 I/O 接口，主要通过 io_submit(), io_getevents() 和 io_cancel() 等系统调用提供异步 I/O 功能。特点：

真正的异步 I/O：与 POSIX AIO 不同，Kernel AIO 是真正的内核级异步 I/O。它直接在内核中处理 I/O 请求，不依赖于后台线
文件 I/O 限制：Kernel AIO 最初设计是为了服务于高性能 I/O 场景，特别是直接对块设备进行操作的应用程序（如数据库、文
性能：Kernel AIO 在处理大量并发 I/O 请求时，性能远远优于 POSIX AIO，特别是在处理大规模的块设备读写时。它的性能提
接口复杂性：相比 POSIX AIO 的简单 API，Kernel AIO 的接口相对复杂，需要更多的低级管理。编写基于 Kernel AIO 的程序

优点：

真正的内核异步 I/O：相比 POSIX AIO，Kernel AIO 提供了更高效的异步 I/O 操作，特别适合高性能场景，如数据库、存储系
性能优势：因为不依赖线程，Kernel AIO 在处理大量并发 I/O 操作时，具有更好的可伸缩性和性能。

缺点：

普通文件支持有限：Kernel AIO 最初是为块设备优化的，因此对普通文件的异步 I/O 支持有限，尤其是使用文件系统缓存的场
API 复杂性：Kernel AIO 的 API 比 POSIX AIO 要复杂一些，需要开发者对内核行为有更多的了解。

io_uring io_uring 的本质

io_uring 本质上是基于环形缓冲区（ring buffer）的 I/O 完成通知机制，允许用户空间与内核通过共享的缓冲区进行高效通信，从而实现真正的异步 I/O。它的设计目标是减少内核和用户空间之间的系统调用开销，简化异步 I/O 操作，并支持更广泛的 I/O 操作（不仅限于文件操作，也包括网络操作等）。io_uring 的工作原理

io_uring 使用两个环形队列：

提交队列（Submission Queue，SQ）：用户空间将 I/O 请求放入该队列，然后将这些请求提交给内核。
完成队列（Completion Queue，CQ）：内核处理完成的 I/O 请求，并将结果放入该队列中，用户空间可以从中获取完成的 I/O

由于 io_uring 通过内核与用户空间共享这两个队列，用户空间可以直接将请求写入提交队列，而无需频繁进行系统调用，极大减少了系统调用的开销。io_uring 的工作流程

创建 io_uring 实例：通过系统调用 io_uring_setup() 创建一个 io_uring 实例，初始化提交和完成队列。
提交 I/O 请求：用户将 I/O 请求写入提交队列。常见操作包括文件读写、网络操作、超时、信号等。
内核处理 I/O：内核从提交队列中读取 I/O 请求并执行处理（这部分是异步的，用户空间不需要等待）。
获取结果：当内核完成 I/O 请求后，会将结果写入完成队列。用户空间可以异步检查完成队列，获取结果。

io_uring 的优势 1. 减少系统调用开销

在传统的异步 I/O（如 aio 接口）中，尽管 I/O 是异步的，但每次提交 I/O 请求时，仍然需要进行一次系统调用。而 io_uring 使用共享的环形缓冲区，允许用户空间直接将请求写入内存映射的缓冲区内，无需每次进行系统调用。这显著减少了系统调用的频率，从而提高了性能。2. 真正的异步 I/O

io_uring 提供了真正的异步 I/O。用户无需阻塞等待 I/O 完成，I/O 操作可以立即返回，I/O 完成后通过完成队列获取通知，完全非阻塞。3. 支持更丰富的操作

除了常见的文件读写操作，io_uring 还支持更广泛的操作，如网络 I/O、超时、信号等待等。其设计不仅限于文件 I/O，还可以处理网络连接等异步事件。4. 批量处理

io_uring 支持批量提交 I/O 请求和批量获取结果，可以同时提交和处理多个请求，进一步提高了 I/O 操作的效率。5. 减少内核锁争用

由于用户空间和内核共享的队列数据结构可以直接进行更新，内核锁争用大大减少，从而进一步提高了并发性能。io_uring 的使用场景

高性能网络服务器：对于需要处理大量并发请求的网络服务器，io_uring 的高效异步 I/O 特性使其非常适合用于高性能网络应
文件操作密集型应用：例如数据库系统、文件系统等需要频繁进行大量读写操作的应用，可以使用 io_uring 大幅提高吞吐量。
事件驱动的异步编程：与事件驱动编程模型（如 epoll）类似，io_uring 提供了对多种异步事件（包括超时、信号）的支持，

io_uring 与 epoll 的对比

异步模型：io_uring 是异步的，而 epoll 是同步的。epoll 需要程序调用 epoll_wait() 等函数来主动等待事件，io_uring 则
系统调用开销：epoll 每次处理事件需要系统调用，而 io_uring 则利用内存映射和共享队列减少了系统调用次数。
操作范围：epoll 主要用于监听文件描述符上的事件，适用于网络 I/O，而 io_uring 支持更多类型的操作，包括文件、网络、

io_uring 的局限性

虽然 io_uring 在性能上有很大提升，但它有一些局限性：

内核版本要求：io_uring 需要 Linux 5.1 及以上的内核才能使用，且某些功能依赖于更高版本的内核。
复杂性：尽管性能上有优势，但 io_uring 的编程模型更复杂，需要对环形缓冲区的机制和底层异步编程有较深入的理解。