

epoll背后的原理_lianhunqianr1的博客

-CSDN博客

23-29 minutes

1 简介

Epoll 是个很老的知识点，是后端工程师的经典必修课。这种知识具备的特点就是研究的人多，所以研究的趋势就会越来越深。当然分享的人也多，由于分享者水平参差不齐，也产生的大量错误理解。

今天我再次分享 epoll，肯定不会列个表格，对比一下差异，那就太无聊了。我将从线程阻塞的原理，中断优化，网卡处理数据过程出发，深入的介绍 epoll 背后的原理，最后还会 diss 一些流行的观点。相信无论你是否已经熟悉 epoll，本文都会对你有价值。

2 引言

正文开始前，先问大家几个问题。

1、epoll 性能到底有多高。很多文章介绍 epoll 可以轻松处理几十万个连接。而传统 IO 只能处理几百个连接 是不是说 epoll 的性能就是传统 IO 的千倍呢？

2、很多文章把网络 IO 划分为阻塞，非阻塞，同步，异步。并表示：非阻塞的性能比阻塞性能好，异步的性能比同步性能好。

- 如果说阻塞导致性能低，那传统 IO 为什么要阻塞呢？
- epoll 是否需要阻塞呢？
- Java 的 NIO 和 AIO 底层都是 epoll 实现的，这又怎么理解同步和异步的区别？

3、都是 IO 多路复用。

- 既生瑜何生亮，为什么会有 select, poll 和 epoll 呢？
- 为什么 epoll 比 select 性能高？

PS:

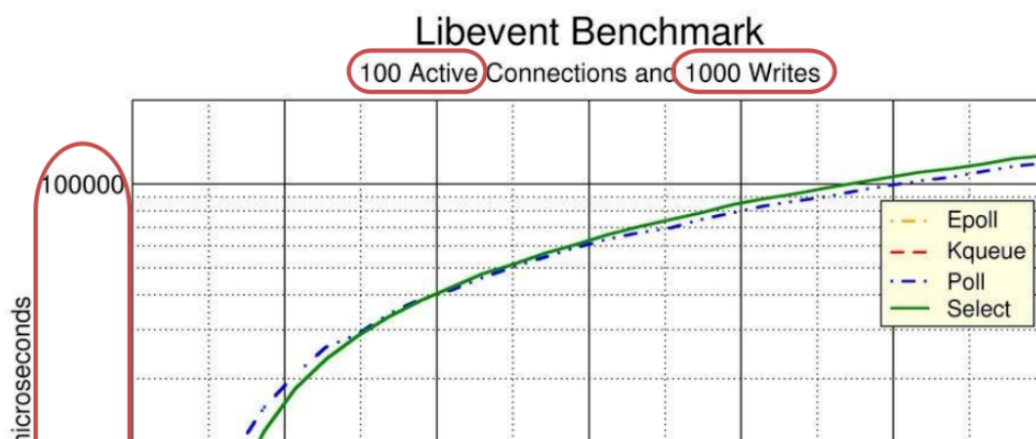
本文共包含三大部分：**初识 epoll、epoll 背后的原理、Diss 环节。**

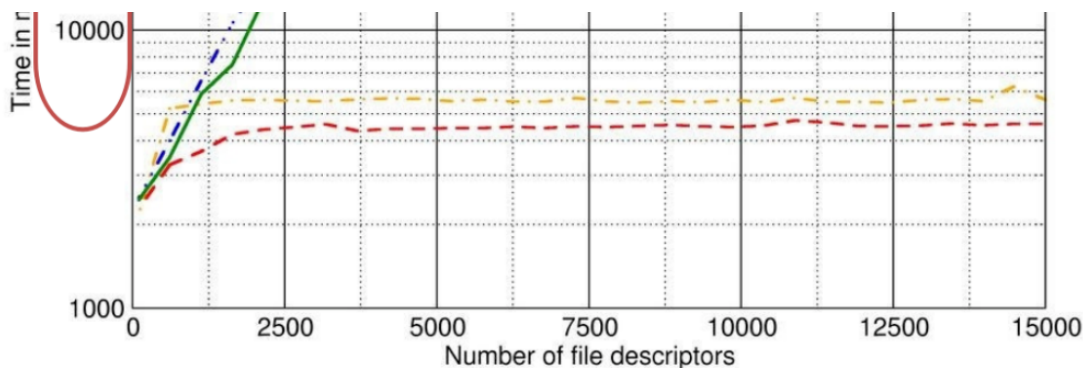
本文的重点是介绍原理，建议读者的关注点尽量放在：“为什么”。

Linux 下进程和线程的区别其实并不大，尤其是在讨论原理和性能问题时，因此本文中“进程”和“线程”两个词是混用的。

3 初识 epoll

epoll 是 Linux 内核的可扩展 I/O 事件通知机制，其最大的特点就是性能优异。下图是 **libevent**(一个知名的异步事件处理软件库)对 select, poll, epoll, kqueue 这几个 I/O 多路复用技术做的性能测试。





很多文章在描述 epoll 性能时都引用了这个基准测试，但少有文章能够清晰的解释这个测试结果。

这是一个限制了100个活跃连接的基准测试，每个连接发生1000次读写操作为止。纵轴是请求的响应时间，横轴是持有的 socket 句柄数量。随着句柄数量的增加，epoll 和 kqueue 响应时间几乎无变化，而 poll 和 select 的响应时间却增长了非常多。

可以看出来，epoll 性能是很高的，并且随着监听的文件描述符的增加，epoll 的优势更加明显。

不过，这里限制的100个连接很重要。epoll 在应对大量网络连接时，只有活跃连接很少的情况下才能表现的性能优异。换句话说，epoll 在处理大量非活跃的连接时性能才会表现的优异。如果15000个 socket 都是活跃的，epoll 和 select 其实差不了太多。

为什么 epoll 的高性能有这样的局限性？

问题好像越来越多了，看来我们需要更深入的研究了。

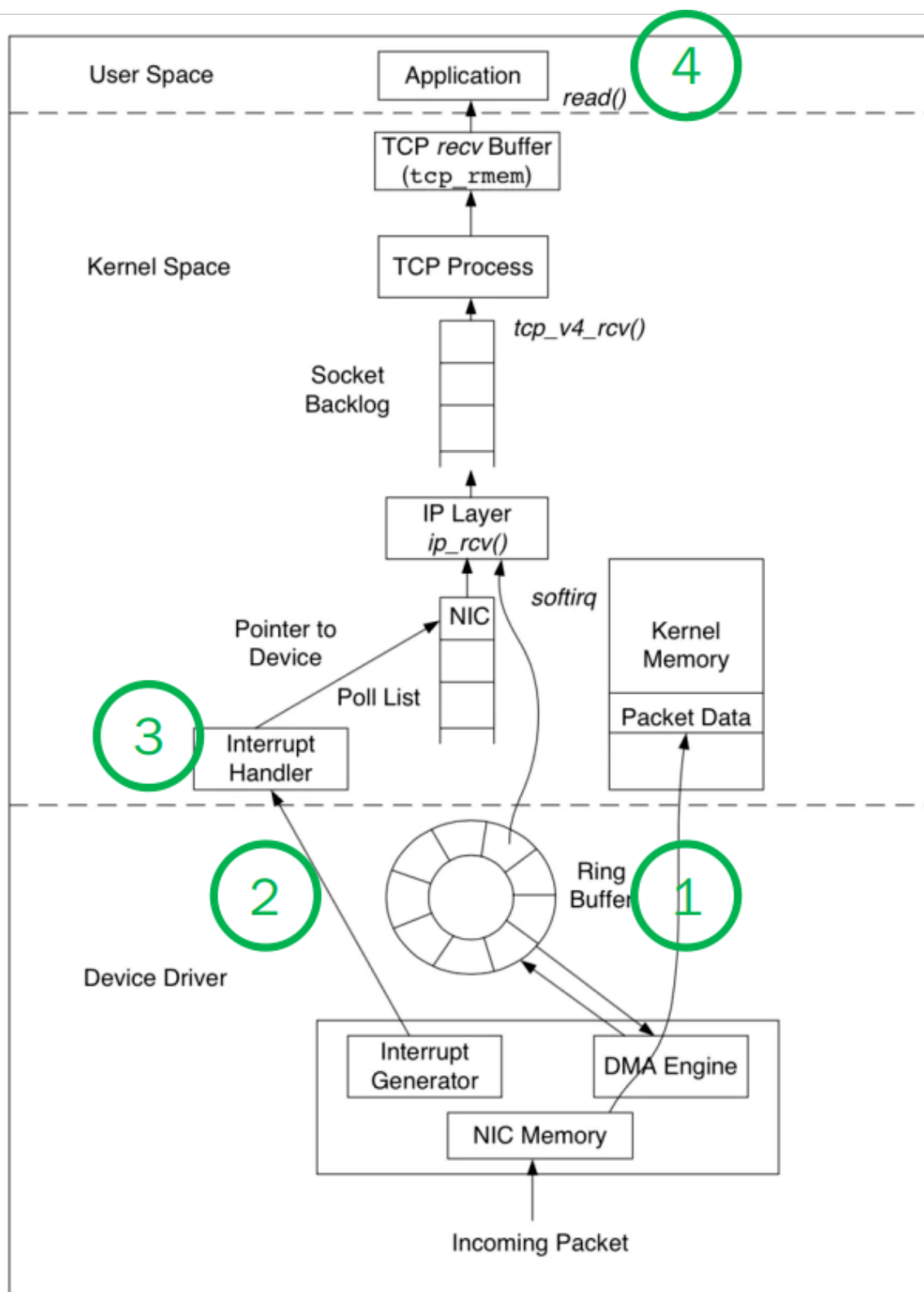
4 epoll背后的原理

4.1 阻塞

4.1.1 为什么阻塞

我们以网卡接收数据举例，回顾一下之前我分享过的网卡接收数据

的过程。



为了方便理解，我尽量简化技术细节，可以把接收数据的过程分为4步：

1. NIC（网卡）接收到数据，通过 DMA 方式写入内存(Ring Buffer 和 sk_buff)。
2. NIC 发出中断请求（IRQ），告诉内核有新的数据过来了。
3. Linux 内核响应中断，系统切换为内核态，处理 Interrupt Handler，从RingBuffer 拿出一个 Packet，并处理协议栈，填充 Socket 并交给用户进程。
4. 系统切换为用户态，用户进程处理数据内容。

网卡何时接收到数据是依赖发送方和传输路径的，这个延迟通常都很高，是毫秒(ms)级别的。而应用程序处理数据是纳秒(ns)级别的。也就是说整个过程中，内核态等待数据，处理协议栈是个相对很慢的过程。这么长的时间里，用户态的进程是无事可做的，因此用到了“阻塞（挂起）”。

4.1.2 阻塞不占用 cpu

阻塞是进程调度的关键一环，指的是进程在等待某事件发生之前的等待状态。请看下表，在 Linux 中，进程状态大致有7种（在 include/linux/sched.h 中有更多状态）：

进程状态	中文名	说明
TASK_RUNNING	可运行状态	进程要么在CPU上执行，要么准备执行（等待cpu时间片的调度）。
TASK_INTERRUPTIBLE	可中断的睡眠状态（挂起）	进程被挂起(睡眠)，直到某个条件满足(产生一个硬件中断，释放进程等待的系统源，或传递一个信号)都是可以唤醒进程的条件（状态放回到TASK_RUNNING）
TASK_UNINTERRUPTIBLE	不可中断的睡眠状态	不常见，与TASK_INTERRUPTIBLE类似，但不会被信号中断
TASK_STOPPED	暂停状态	收到某种信号，运行被停止（SIGSTOP，SIGTSTP，SIGTTIN或SIGTTOU）
TASK_TRACED	被跟踪状态	进程的执行，被Debugger程序暂停
EXIT_ZOMBIE	僵死状态	僵尸状态。进程已经退出但尚未被父进程收尸，内核不能丢弃进程描述符中的数据
EXIT_DEAD	僵死撤销状态	最终状态，内核释放进程描述符中的数据，状态非常短暂，几乎不可能同步ps捕捉

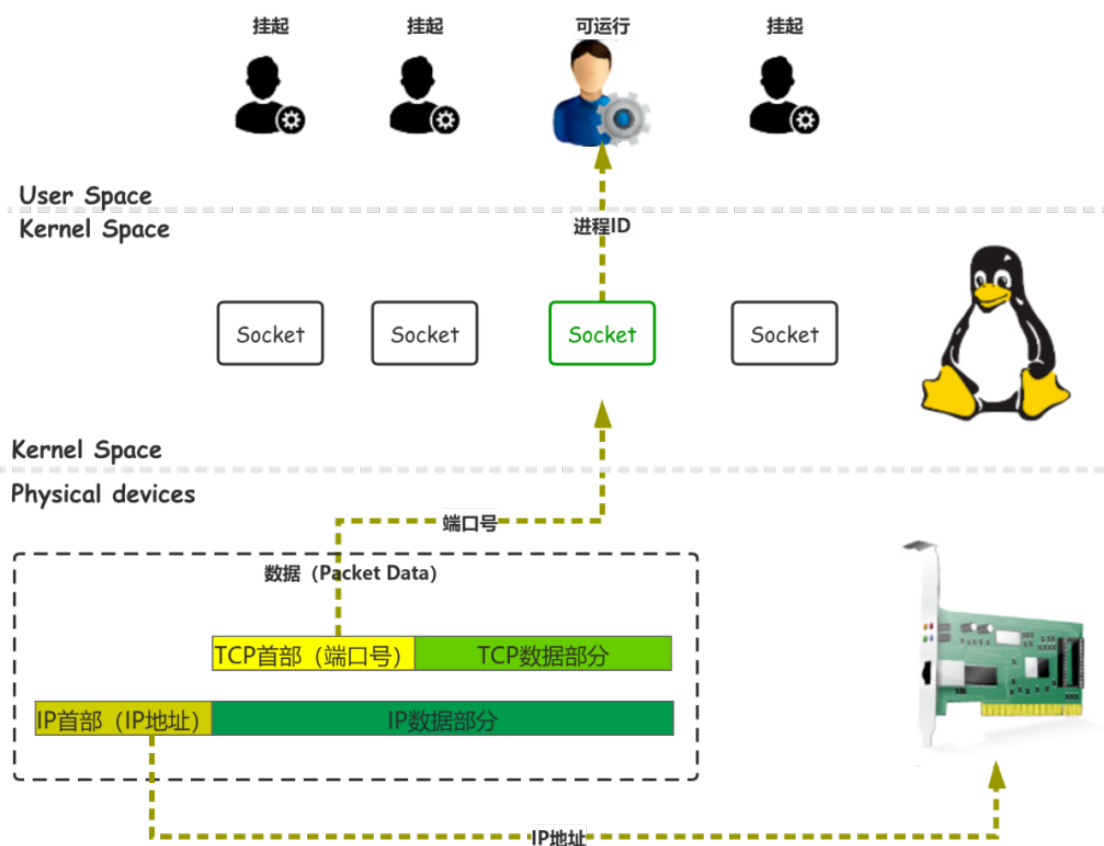
从说明中其实就可以发现，“可运行状态”会占用 CPU 资源，另外创建和销毁进程也需要占用 CPU 资源（内核）。重点是，当进程被“阻塞/挂起”时，是不会占用 CPU 资源的。

换个角度来讲。为了支持多任务，Linux 实现了进程调度的功能（CPU 时间片的调度）。而这个时间片的切换，只会在“可运行状态”的进程间进行。因此“阻塞/挂起”的进程是不占用 CPU 资源的。

另外讲个知识点，为了方便时间片的调度，所有“可运行状态”状态的进程，会组成一个队列，就叫“**工作队列**”。

4.1.3 阻塞的恢复

内核当然可以很容易的修改一个进程的状态，问题是网络 IO 中，内核该修改那个进程的状态。



socket 结构体，包含了两个重要数据：进程 ID 和端口号。进程 ID 存放的就是执行 connect, send, read 函数，被挂起的进程。在 socket 创建之初，端口号就被确定了下来，操作系统会维护一个端口号到 socket 的数据结构。

当网卡接收到数据时，数据中一定会带着端口号，内核就可以找到对应的 socket，并从中取得“挂起”进程的 ID。将进程的状态修改为“可运行状态”（加入到工作队列）。此时内核代码执行完毕，将控制权交还给用户态。通过正常的“CPU 时间片的调度”，用户进程得以处理数据。

4.1.4 进程模型

上面介绍的整个过程，基本就是 BIO（阻塞 IO）的基本原理了。用户进程都是独立的处理自己的业务，这其实是一种符合进程模型的处理方式。

4.2 上下文切换的优化

上面介绍的过程中，有两个地方会造成频繁的上下文切换，效率可能会很低。

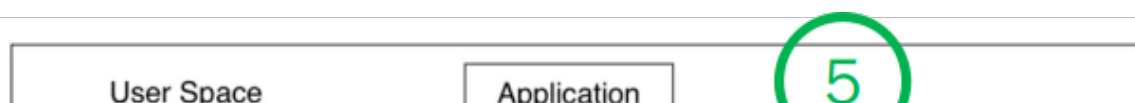
1. 如果频繁的收到数据包，NIC 可能频繁发出中断请求（IRQ）。CPU 也许在用户态，也许在内核态，也许还在处理上一条数据的协议栈。但无论如何，CPU 都要尽快的响应中断。这么做实际上非常低效，造成了大量的上下文切换，也可能导致用户进程长时间无法获得数据。（即使是多核，每次协议栈都没有处理完，自然无法交给用户进程）
2. 每个 Packet 对应一个 socket，每个 socket 对应一个用户态的进程。这些用户态进程转为“可运行状态”，必然要引起进程间的上下文切换。

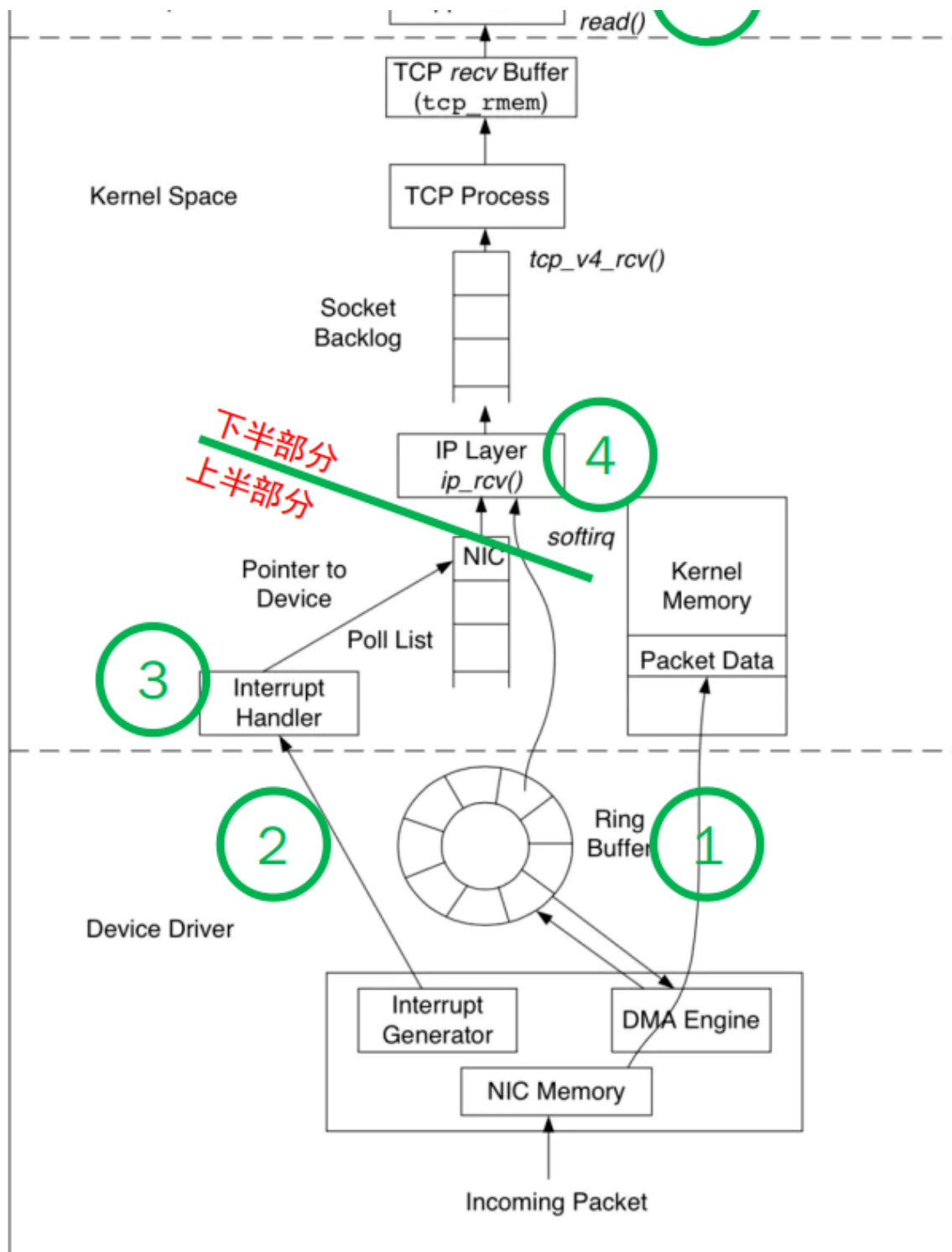
4.2.1 网卡驱动的 NAPI 机制

在 NIC 上，解决频繁 IRQ 的技术叫做 New API(NAPI)。原理其实特别简单，把 Interrupt Handler 分为两部分。

1. 函数名为 `napi_schedule`，专门快速响应 IRQ，只记录必要信息，并在合适的时机发出软中断 `softirq`。
2. 函数名为 `netrxaction`，在另一个进程中执行，专门响应 `napi_schedule` 发出的软中断，批量的处理 RingBuffer 中的数据。

所以使用了 NAPI 的驱动，接收数据过程可以简化描述为：





1. NIC 接收到数据，通过 DMA 方式写入内存(Ring Buffer 和 sk_buff)。
2. NIC 发出中断请求 (IRQ) ，告诉内核有新的数据过来了。
3. driver 的 napi_schedule 函数响应 IRQ，并在合适的时机发出软中断 (NET_RX_SOFTIRQ)

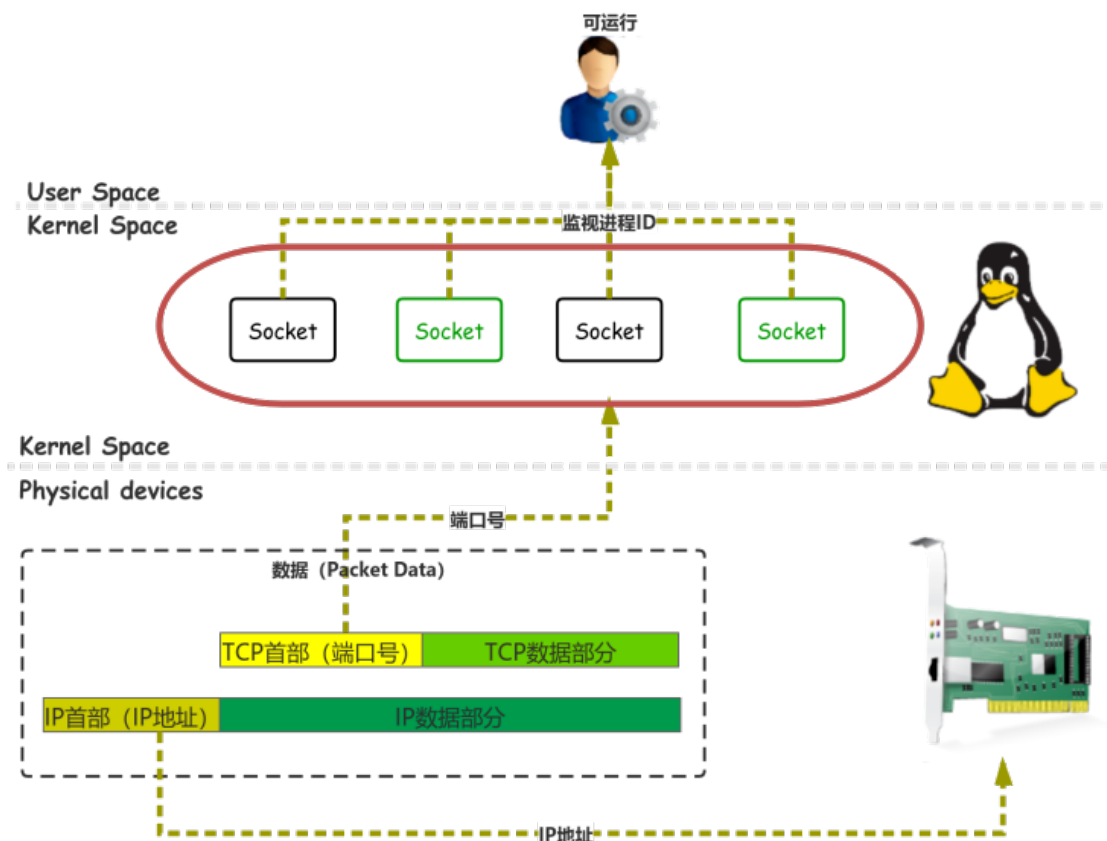
4. driver 的 net_rx_action 函数响应软中断，从 Ring Buffer 中批量拉取收到的数据。并处理协议栈，填充 Socket 并交给用户进程。
5. 系统切换为用户态，多个用户进程切换为“可运行状态”，按 CPU 时间片调度，处理数据内容。

一句话概括就是：等着收到一批数据，再一次批量的处理数据。

4.2.2 单线程的 IO 多路复用

内核优化“进程间上下文切换”的技术叫的“IO 多路复用”，思路和 NAPI 是很接近的。

每个 socket 不再阻塞读写它的进程，而是用一个专门的线程，批量的处理用户态数据，这样就减少了线程间的上下文切换。



作为 IO 多路复用的一个实现，select 的原理也很简单。所有的 socket 统一保存执行 select 函数的（监视进程）进程 ID。任何一个 socket 接收了数据，都会唤醒“监视进程”。内核只要告诉“监视进

程”，那些 socket 已经就绪，监视进程就可以批量处理了。

4.3 IO 多路复用的进化

4.3.1 对比 epoll 与 select

select, poll 和 epoll 都是“IO 多路复用”，那为什么还会有性能差距呢？篇幅限制，这里我们只简单对比 select 和 epoll 的基本原理差异。

对于内核，同时处理的 socket 可能有很多，监视进程也可能有多个。所以监视进程每次“批量处理数据”，都需要告诉内核它“关心的 socket”。内核在唤醒监视进程时，就可以把“关心的 socket”中，就绪的 socket 传给监视进程。

换句话说，在执行系统调用 select 或 epoll_create 时，入参是“关心的 socket”，出参是“就绪的 socket”。

而 select 与 epoll 的区别在于：

- **select (一次 $O(n)$ 查找)**

1. 每次传给内核一个用户空间分配的 fd_set 用于表示“关心的 socket”。其结构（相当于 bitset）限制了只能保存1024个 socket。
2. 每次 socket 状态变化，内核利用 fd_set 查询 $O(1)$ ，就能知道监视进程是否关心这个 socket。
3. 内核是复用了 fd_set 作为出参，返还给监视进程（所以每次 select 入参需要重置）。

然而监视进程必须遍历一遍 socket 数组 $O(n)$ ，才知道哪些 socket 就绪了。

- **epoll (全是 $O(1)$ 查找)**

1. 每次传给内核一个实例句柄。这个句柄是在内核分配的红黑树 rbr+

双向链表 rdllist。只要句柄不变，内核就能复用上次计算的结果。

2. 每次 socket 状态变化，内核就可以快速从 rbr 查询O(1)，监视进程是否关心这个 socket。同时修改 rdllist，所以 rdllist 实际上是“就绪的 socket”的一个缓存。
3. 内核复制 rdllist 的一部分或者全部（LT 和 ET），到专门的 epoll_event 作为出参。

所以监视进程，可以直接一个个处理数据，无需再遍历确认。

Select 示例代码

```
int fds[] = ... //socket数组;
fd_set read_fds,temp_read_fds; //用户空间分配的bitset,既作为与内核交互的入参，又作为出参
FD_ZERO(&read_fds);
for(int i=0; i < fds.count; i++){
    FD_SET(fds[i], &read_fds);
}
while(1){
    //因为内核复用fd_set作为出参，这里需要重置fd_set，利用另一个fd_set赋值，是最快的重置方法。
    temp_read_fds=read_fds;

    //1、这个就是监视器，会阻塞线程
    //2、作为入参时，复制read_fds给内核（long[32]）
    //3、内核把全部符合就绪条件的socket填充到read_fds又作为出参。
    int n = select(..., &temp_read_fds, ...)
    for(int i=0; i < fds.count; i++){ // 用户态遍历socket数组O(n)，查找需要处理的socket
        if(FD_ISSET(fds[i],&temp_read_fds)){ // 用户态操作：通过bitset确定是否需要处理
            //处理业务逻辑
            FD_CLR(fds[i], &read_fds);
        }
    }
}
```

Epoll 示例代码

```
int fds[] = ... //socket数组;
int efd = epoll_create(...); // 系统调用，在内核空间创建epoll实例（红黑树rbr + 就绪链表rdllist）
for(int i=0; i < fds.count; i++){ // 在while(1)循环体外,所以只操作一次
    epoll_ctl(efd, ...,fds[i],...); //系统调用，红黑树上添加、修改、删除每一个节点（socket）
}
struct epoll_event events[MAX_EVENTS]; //用户空间分配的一段内存，events用作出参
while(1){
    //1、这个就是监视器，会阻塞线程
    //2、内核会利用红黑树，快速查找select需要的socket，放入就绪链表
    //3、把就绪链表中一定数量的内容复制到events
    int n = epoll_wait(efd,&events...)
    for(int i = 0; i < n; i++){
        events[i].data.fd; //events全是需要处理的socket
    }
}
```

}

另外，`epoll_create` 底层实现，到底是不是红黑树，其实也不太重要（完全可以换成 hashtable）。重要的是 `efd` 是个指针，其数据结构完全可以对外透明的修改成任意其他数据结构。

4.3.2 API 发布的时间线

另外，我们再来看看网络 IO 中，各个 api 的发布时间线。就可以得到两个有意思的结论。

1983, `socket` 发布在 Unix(4.2 BSD)
1983, `select` 发布在 Unix(4.2 BSD)
1994, Linux的1.0, 已经支持`socket`和`select`
1997, `poll` 发布在 Linux 2.1.23
2002, `epoll`发布在 Linux 2.5.44

1、`socket` 和 `select` 是同时发布的。这说明了，`select` 不是用来代替传统 IO 的。这是两种不同的用法(或模型)，适用于不同的场景。

2、`select`、`poll` 和 `epoll`，这三个“IO 多路复用 API”是相继发布的。这说明了，它们是 IO 多路复用的3个进化版本。因为 API 设计缺陷，无法在不改变 API 的前提下优化内部逻辑。所以用 `poll` 替代 `select`，再用 `epoll` 替代 `poll`。

4.4 总结

我们花了三个章节，阐述 `Epoll` 背后的原理，现在用三句话总结一下。

1. 基于数据收发的基本原理，系统利用阻塞提高了 CPU 利用率。
2. 为了优化上线文切换，设计了“IO 多路复用”（和 `NAPI`）。
3. 为了优化“内核与监视进程的交互”，设计了三个版本的

API(select,poll,epoll)。

5 Diss 环节


讲完“Epoll 背后的原理”，已经可以回答最初的几个问题。这已经是一个完整文章，很多人劝我删掉下面的 diss 环节。

我的观点是：学习就是个研究+理解的过程。上面是研究，下面再讲一下我的个人“理解”，欢迎指正。

5.1 关于 IO 模型的分类

关于阻塞，非阻塞，同步，异步的分类，这么分自然有其道理。但是在操作系统的角度来看“**这样分类，容易产生误解，并不好**”。

	Blocking	Non-blocking
Synchronous	Read/write	Read/wirte (O_NONBLOCK)
Asynchronous	i/O multiplexing (select/poll)	AIO



5.1.1 阻塞和非阻塞

Linux 下所有的 IO 模型都是阻塞的，这是收发数据的基本原理导致的。阻塞用户线程是一种高效的方式。

你当然可以写一个程序，socket 设置成非阻塞模式，在不使用监视器的情况下，依靠死循环完成一次 IO 操作。但是这样做的效率实

在是太低了，完全没有实际意义。

换句话说，阻塞不是问题，运行才是问题，运行才会消耗 CPU。IO 多路复用不是减少了阻塞，是减少了运行。上下文切换才是问题，IO 多路复用，通过减少运行的进程，有效的减少了上下文切换。

5.1.2 同步和异步

Linux 下所有的 IO 模型都是同步的。BIO 是同步的，select 同步的，poll 同步的，epoll 还是同步的。

Java 提供的 AIO，也许可以称作“异步”的。但是 JVM 是运行在用户态的，Linux 没有提供任何异步支持。因此 JVM 提供的异步支持，和你自己封装成“异步”的框架是没有本质区别的（你完全可以使用 BIO 封装成异步框架）。

所谓的“同步”和“异步”只是两种事件分发器（event dispatcher）或者说是两个设计模式（Reactor 和 Proactor）。都是运行在用户态的，两个设计模式能有多少性能差异呢？

- Reactor 对应 java 的 NIO，也就是 Channel，Buffer 和 Selector 构成的核心的 API。
- Proactor 对应 java 的 AIO，也就是 Async 组件和 Future 或 Callback 构成的核心的 API。

5.1.3 我的分类

我认为 IO 模型只分两类：

1. 更加符合程序员理解和使用的，进程模型；
2. 更加符合操作系统处理逻辑的，IO 多路复用模型。

对于“IO多路复用”的事件分发，又分为两类：Reactor 和 Proactor。

5.2 关于 mmap

epoll 到底用没用到 mmap?

答案：没有！

这是个以讹传讹的谣言。其实很容易证明的，用 epoll 写个 demo。strace 一下就清楚了。

- END -

看完一键三连在看，转发，点赞

是对文章最大的赞赏，极客重生感谢你



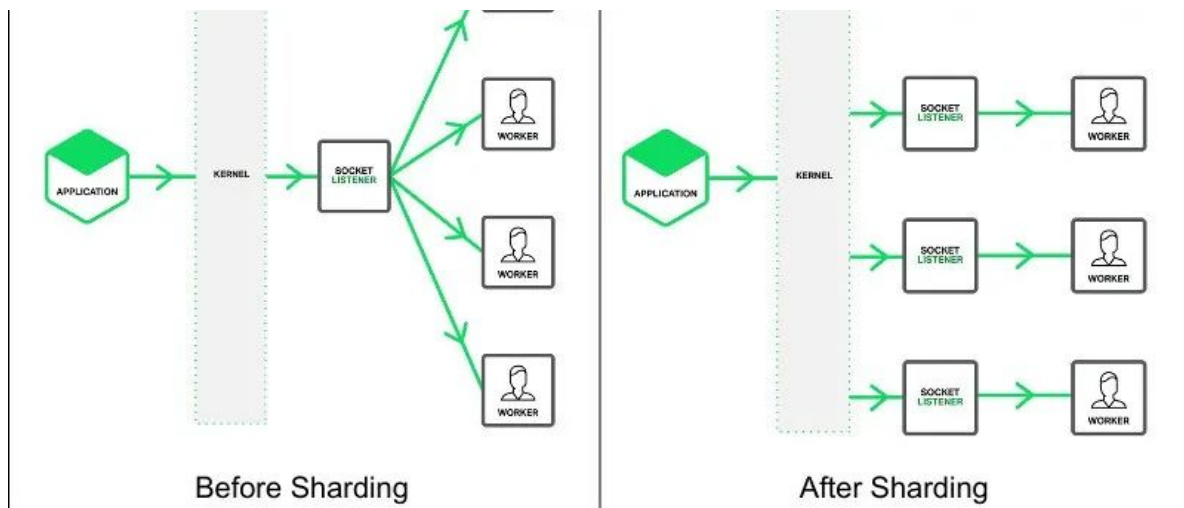
推荐阅读

	<u>BLOCKING</u>	<u>NON-BLOCKING</u>
<u>SYNCHRONOUS</u>	READ/ WRITE	READ/WRITE (O_NONBLOCK) I/O MULTIPLEXING
<u>ASYNCHRONOUS</u>	—	AIO

深入理解Linux异步I/O框架 io_uring



五个半小时



服务器性能优化之网络性能优化



求点赞，在看，分享三连

