《openEuler内核编程》

课程讲稿

第七章 第2讲

内核网络的关键数据结构和函数

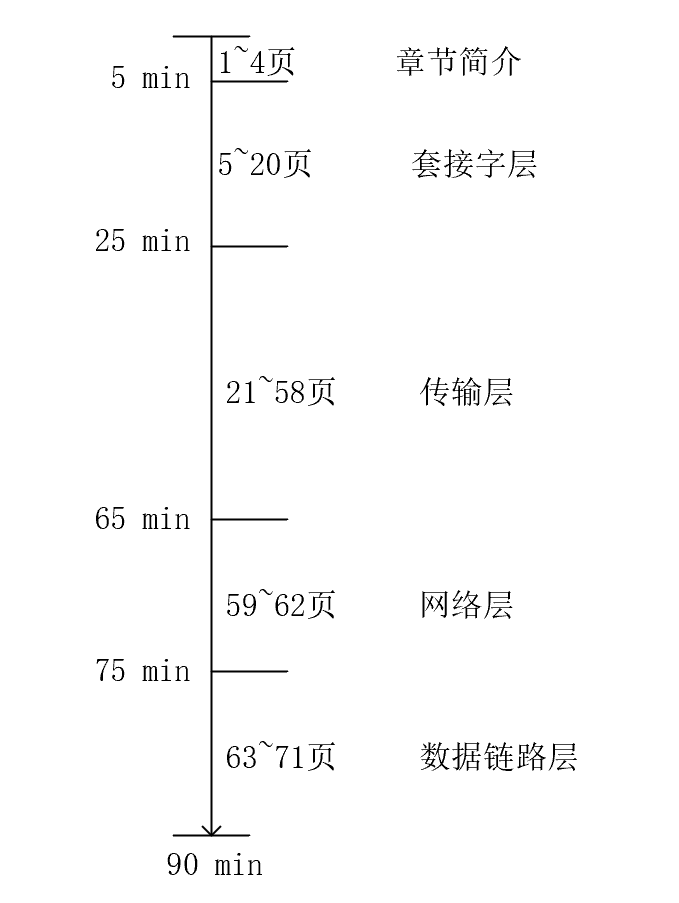
软件所制

第七章 第2讲 内核网络的关键数据结构和函数

**学时：**2学时

**教学目的：**深入了解内核协议栈原理，从源码的角度看网络数据的传递

**课程时间线：**



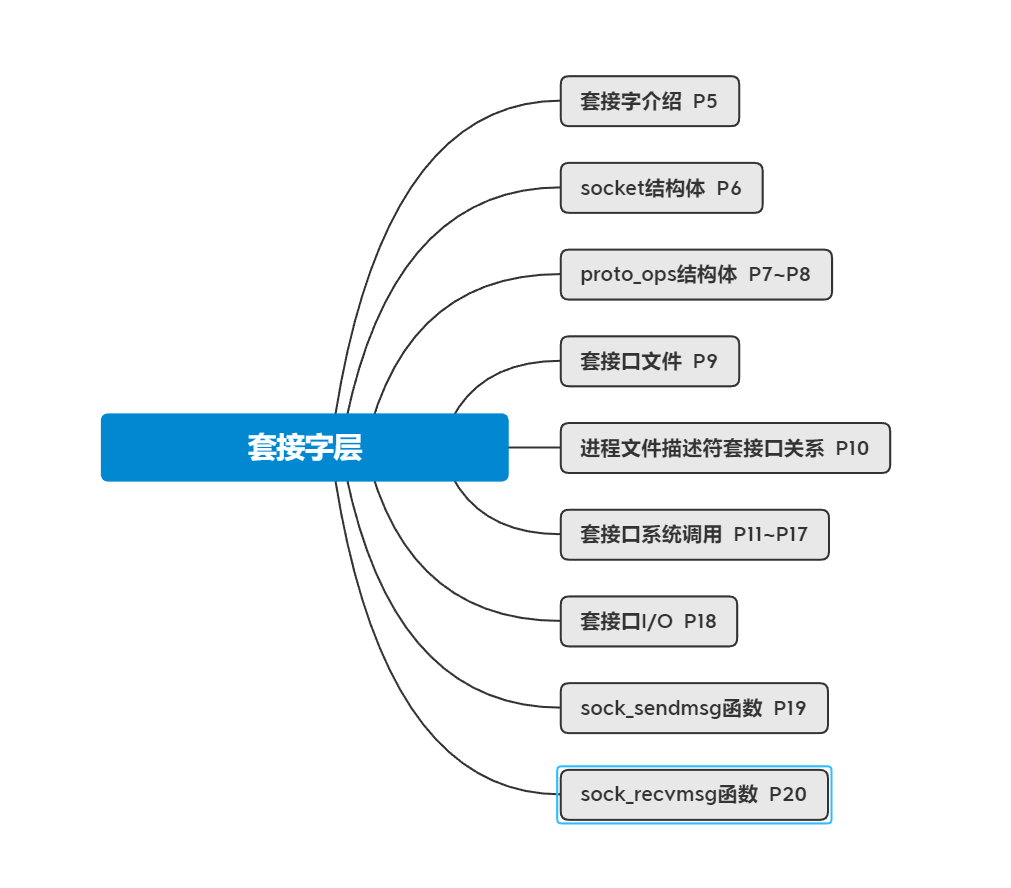
**课外参考读物：**

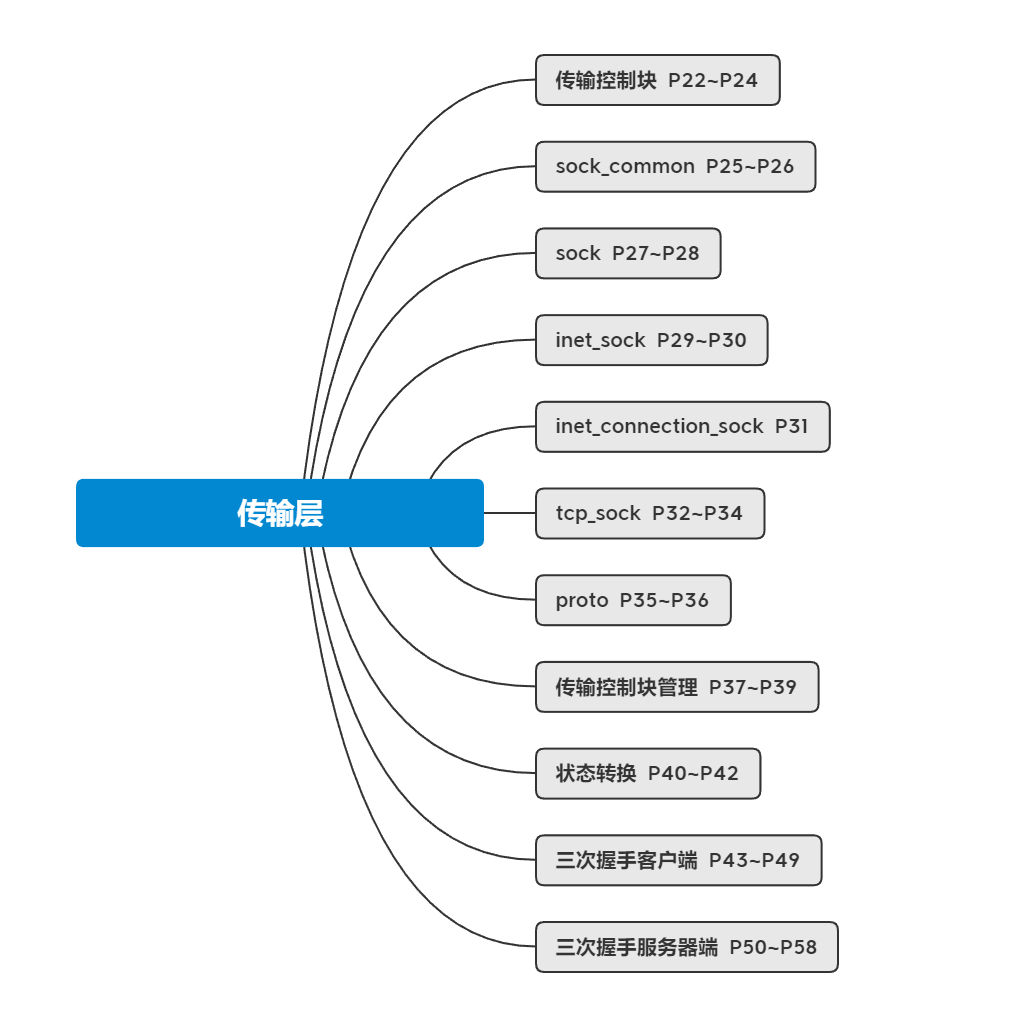
《深入理解Linux网络技术内幕》

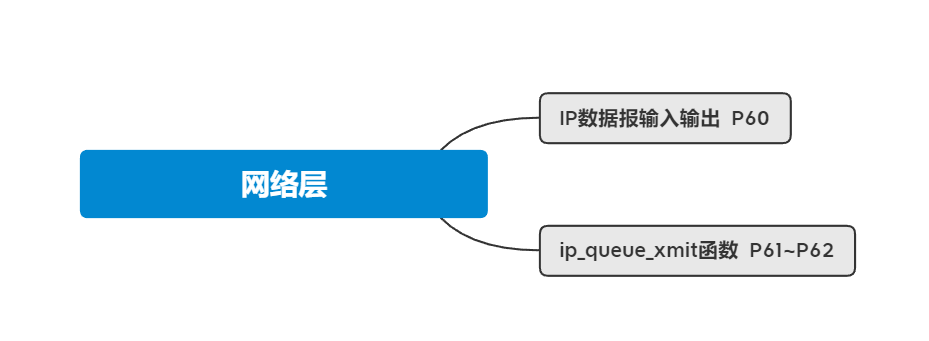
《Linux内核源码剖析 TCPIP实现(上册)》

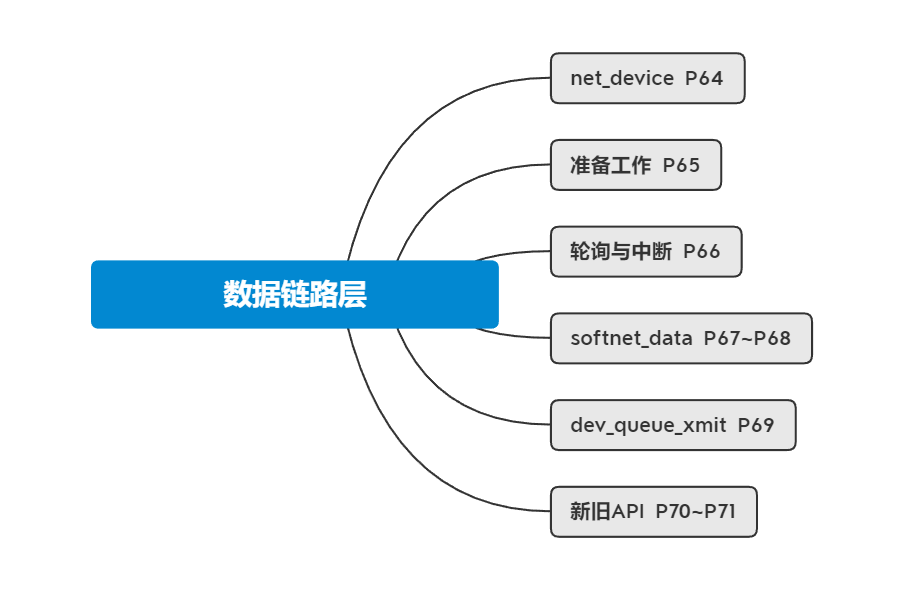
《Linux内核源码剖析 TCPIP实现(下册)》

**知识框图：**









**PPT讲稿：**

1. 我们继续第七章网络知识的学习。
2. 这一节我们主要学习网络数据在内核中的传递。
3. 首先再来回顾一下这张图，从数据从应用层到数据链路层，每一次传输都会增加一个报头，然后进入下一层传输。而我们这一届讲的就是如何增加这一报头以及数据的传输过程。
4. 这一节我们将从四个方面来介绍，分别是：套接字层、传输层、网络层和数据链路层。
5. 首先是套接字层。学习过网络编程的同学对socket这个函数一定并不陌生。Socket编程第一步当然是创建一个socket，但很多网络编程的开发者很容易去网上抄这样的例子，但是为什么第一个参数要使用AF\_INET，为什么第二个参数要使用SOCK\_STREAM或者SOCK\_DGRAM，为什么第三个参数要填0呢。首先我们需要先了解socket。
6. 用户使用socket系统调用编写应用程序时，通过一个数字来表示一个socket，所有的操作都在该数字上进行，这个数字称为套接字描述符。在系统调用 的实现函数里，这个数字就会被映射成一个表示socket的结构体，该结构体保存了该socket的所有属性和数据。state是用于表示套接口所处的状态的标志。该标志有些状态只对TCP有效如这里的ss\_connecting和ss\_disconnecting等等，因为只有TCP是面向连接的协议；Flag是一组标志位，有标识套接口发送队列是否已满的，有标识是否设置某些选项的；ops是指向套接口系统调用中选择对应类型的套接口接口，用来将套接口层系统调用映射到相应传输层协议，ops中对应的是tcp，udp和raw三种套接口类型对应的proto\_ops结构实例；file是指向与该套接口相关联的file结构的指针。Sk是指向与该套接口关联的传输控制块。wait是等待该套接字的进程队列。还有最有一个type是套接口的类型，其中比较常用的有基于连接的套接口sock\_stream、基于数据报的套接口sock\_dgram和原始套接口sock\_raw。
7. 上一节课在讲socket层中，我们也提到了socket结构体中最关键的成员是ops和sk，前者是指向特定传输协议的操作集，后者是指向与该套接口相关的传输控制块。这里我们具体来看。这是proto\_ops结构体的代码，可以看到代码分为三部分，第一部分是family表示协议族，第二部分是owner表示所属模块，第三部分是一组与套接口系统调用相对应的传输层函数指针，其中bind、connect、accept等都是我们比较常用，从图中我们可以看出这一组指针可以看作是一张套接口系统调用到传输层函数的跳转表。
8. 每一种文件都有各自的文件类型，如设备文件包括字符设备文件和块设备文件等，而与套接口关联的文件类型为套接口文件。套接口有一套独立的系统调用，包括建立套接口、连接和IO操作等，由于在建立套接口后返回的是文件描述符，因此也可以通过标准的文件IO操作进行对套接口的读写，例如用send进行数据的发送，而事实上也可以通过write系统调用达到同样的效果。这是由于在创建套接口文件时，使file结构中的f\_op指向了socket\_file\_ops。通过socket\_file\_ops可以看到套接口文件支持哪些系统调用，它支持read、write、poll、ioctl。这其实就是上节课我们提到的可以把普通文件操作作为网络I/O。
9. 在描述套接口系统调用之前，需要介绍将进程、文件描述符和套接口联系在一起的数据结构。下图给出了这些结构以及有关的结构成员。在进程的task\_struct结构中，files指向file\_struct结构，该结构主要功能是管理fd\_array指针数组指向的描述符，每个file结构描述一个打开的文件。图中仅显示了一个file结构，通过current->files->fd\_array[fd]就可以访问当前进程某个文件描述符了。在file结构中有两个成员比较关键，一个是private\_data另一个是f\_op。f\_op指向file\_operations结构，该结构包含一张实现read、write、ioctl、select、close系统调用的函数指针表，图中显示f\_op指向一个全局的file\_operations结构，这个结构就是上面我们讲的socket\_file\_ops，该结构包含指向套接口文件关联的函数的指针。private\_data指向相关I/O对象专用数据。对于套接口而言，private\_data指向与描述符相关的socket结构。socket结构中的ops指向产生套接口时选中的协议的proto\_ops结构。最后传输控制块tcp\_sock结构中的skc\_prot指向产生套接口时选中的协议的proto结构。
10. 介绍完进程、文件描述符和套接口的联系，就可以来说说套接口系统调用了，套接字系统调用出现在tcp三次握手之前，tcp服务器端需要经过初始化socket，端口绑定，端口监听以及accept阻塞来等待客户端连接，这四个步骤分别对应socket，bind，listen和accept四个系统调用函数。完成后tcp服务器端处于阻塞状态，当tcp客户端有connect请求来建立连接，这个过程就是tcp三次握手来建立连接。三次握手我们会在下面的传输层具体介绍。
11. 四个系统调用我们一个一个来看，socket系统调用是把套接口的创建和此套接口关联的文件描述符的分配做简单的封装，从而完成创建套接口的功能。一般的，系统调用会通过syscall中的socket进行sys\_socket调用，在实现系统调用的函数调用链中，关键的是sock\_create函数。
12. 在sock\_create中又会再一次的调用\_\_sock\_create函数。在这里我们应该注意，内核网络模块中有很多函数的实现接口都是dosomething，而具体的实现都在\_\_dosomething中，这个大家在阅读代码的时候可以关注一下。Sock\_create函数中的参数大家应该已经比较熟了，family是协议族，type是套接口类型、protocol是传输层协议，最后一个sock是输出参数，是最后创建成功的套接口指针。而\_\_sock\_create的参数要比sock\_create多几个，其中net\_ns是一种namespace机制，提供一种资源隔离方案。我们这里简单介绍一下，Linux Namespaces机制提供一种资源隔离方案。PID,IPC,Network等系统资源不再是全局性的，而是属于某个特定的Namespace。每个namespace下的资源对于其他namespace下的资源都是透明，不可见的，具体的大家可以参考namespace机制。res是输出参数与sock\_create中的sock一致，kern标识由应用程序还是内核创建该套接口，一般为0/1，主要用来标识创建的套接口用于内核还是普通进程。
13. 在\_\_sock\_create中会首先对参数进行检查，然后调用sock\_allco函数为socket结构体分配内存，之后对传入的参数进行设置，比如设置套接口类型的type，根据family获取已注册到net\_families中对应的net\_proto\_family指针，最后当调用pf->create时即调用inet\_create对已创建的套接口进行初始化，下面是inet\_create的部分代码，我们可以看到大都是一些设置操作，有设置状态为ss\_UNCONNECTED，有设置套接口和传输层直接的接口ops，还有分配传输控制块并对传输控制块做一些设置，比如是否需要检验和是否是面向连接的传输控制块。
14. bind系统调用也是从syscall中的bind来的，作用是讲一个本地的ip地址即传输层端口与套接字关联起来。在他的函数调用链中，首先会在sockfd\_lookuo\_light函数中根据fd查找socket指针。如果当前套接口在传输层接口上有bind的实现，则直接调用传输层接口上的bind()，直接进行bind()操作即可，目前只有SOCK\_RAW类型的套接口的传输层接口实现了bind接口，为raw\_bind()；对于其他类型，则将IP地址拷贝到传输控制块中，然后调用传输控制块提供的get\_port()函数，绑定端口。
15. listen系统调用是用于通知进程准备接受套接口上的连接请求，同时也指定套接口上可以排队等待的连接数的门限值。超过门限值时，套接口将拒绝新的连接请求，TCP将忽略进入的连接请求。在sys\_listen实现中，设置的somaxconn使它为sysctl\_somaxconn，sysctl\_somaxconn存储的是服务器监听时，允许每个套接字连接队列长度的最大值，默认值是SOMAXCONN，即128，在sysctl\_core\_net\_init()函数中初始化。当指定的最大连接数超过系统限制，则使用系统当前允许的连接队列中连接的最大数。再下面，如果是TCP套接字，sock->ops指向的是inet\_stream\_ops，sock->ops是在inet\_create()函数中初始化，所以listen接口调用的是inet\_listen()函数。
16. accept系统调用是用于等待连接请求，返回一个新的文件描述符，指向一个连接到客户的新套接字文件，再系统调用链中会首先sockfd\_lookup\_light函数根据文件描述符fd获取套接口指针，然后调用sock\_alloc分配一个新的套接口，准备处理来自客户端的连接，之后初始化创建的socket，包括其type和ops字段，对应的是套接口类型和系统调用的操作跳转表，然后给newsock分配对应的文件描述符并跟套接口绑定，最后通过sock->ops->accept调用对应参数协议中的accept操作。
17. 上面讲了这么多连接，很多人可能会有疑问，那数据在哪里呢，这就要说到套接口I/O，不管是数据的传输还是接收，用户层的数据最终都是以msghdr消息头来描述的，消息头中的成员包括msg\_name指向sock\_addr结构的目的地址指针；msg\_namelen是目的地址或源地址的缓存区大小；msg\_iov和msg\_iovlen是提供待发送数据或接收数据的缓冲区。它由一个分散/聚合数组构成，msg\_iov是指向这个iovec结构数组的指针，msg\_iovlen是该数组的大小；msg\_control是指向一个附加的数据结构，通常情况下为cmsghdr结构类型的数组；msg\_controllen是整个控制信息考虑对齐后的长度；msg\_flag：接收标志。整个结构如下图所示。
18. 数据的处理主要是调用sock\_sendmsg()函数和sock\_recvmsg()函数。几个常用的发送数据的函数中，send、sendto、sendmsg系统调用大都是这部分代码中的操作，先初始化消息头，然后把套接字地址从用户空间拷贝到内核空间中，最终都调用sock\_sendmsg()来输出数据，这些系统调用本身只是准备和校验消息头，然后调用sock\_sendmsg()，而sock\_sendmsg()最终则调用proto\_ops中的相应接口，然后由proto来选择恰当的协议。
19. 接收数据的过程和发送数据相似，recv、recvfrom、recvmsg系统调用最终都调用sock\_recvmsg()来接收数据，然后sock\_recvmsg最终则用proto\_ops中的相应接口，然后由proto来选择恰当的协议。
20. 数据通过套接口层，便到了传输层。
21. 各协议族传输层使用各自的传输控制块存放套接字所要求的信息，如TCP传输控制块，UDP传输控制块，RAW传输控制块。内核会根据协议族和传输层协议的特点，分层次定义了多个结构来组成传输控制块。以tcp\_sock为例，其中从小到大依次为sock\_common结构、sock结构、inet\_sock结构、inet\_connection\_sock结构。
22. Sock\_common结构是传输控制块信息的最小集合，由sock和inet\_timewait\_sock结构前面相同部分单独构成。sock结构是比较通用的网络层描述块，构成传输控制块的基础，与具体的协议族无关。它描述了各协议族传输层协议的公共信息，不同协议族的传输层在使用该结构时会对其进行扩展，来适合各自的传输特性。inet\_sock结构由sock结构及其它特性组成，构成IPv4协议族传输控制块。
23. inet\_sock结构是比较通用的IPV4协议族描述块，包含IPV4协议族基础传输层，即UDP、TCP、RAW控制块共有的信息，其中包括外部和本地IP地址、外部和本地端口号、IP首部原型、本地使用的IP选项等。tcp\_sock结构，TCP传输控制块，支持TCP的完整特性，保存了TCP为各连接维护的所有节点信息，其中两个方向的序列号、窗口大小、重传次数等等都是保存在tcp\_sock结构中的。同理，upd\_sock结构，UDP传输控制块，支持UDP的完整特性。
24. 上面是传输控制块的总体结构，现在我们从sock\_common结构体来看，sock\_common结构体比较复杂，，包含了多个属性。skc\_hash是存储TCP状态为established时加入到散列表的关键字键值。由于计算键值相对耗时，因此用一个成员来存储键值有利于提高效率。skc\_family是所属协议族。skc\_state等同于TCP状态而UDP不存在连接和传输状态。skc\_reuse标识是否可以重用地址和端口，在SO\_REUSEADDR中设置，linxu系统中设置地址可重用，端口也可以重用。skc\_bound\_dev\_if如果不为0，即为输出报文的网络设备索引号，这一般是通过应用程序的setsockopt里面的选项设置。
25. 已绑定端口的传输控制块利用skc\_bind\_node插入到与之绑定端口信息结构为头节点的链表中，释放端口时，会从中删除，但是它仅用于基于连接的传输控制块，如TCP。skc\_prot指向网络接口层的指针。skc\_refcnt是引用计数器。为0时才能被释放。TCP维护一个所有TCP传输控制块的散列表tcp\_hashinfo，而sk\_node用来将所属TCP传输控制块链接到该散列表；UDP维护一个已经绑定端口的UDP控制块的散列表udp\_hash，而skc\_node用来将所属UDP传输控制块链接到该散列表。以上这些都是sock\_common结构中比较重要的成员。
26. sock结构是构成传输控制块的基础，跟具体的协议族无关，包含各协议族传输层协议的公共信息，因此不能直接作为传输层的控制块来使用，不同协议族的传输层在使用sock结构时都会对其进行扩展，使其适合各自的传输特性，例如inet\_sock结构就是由sock结构和其他一些特性组成，时IPv4协议族传输控制块的基础。Sock结构的第一个肯定是sock\_common结构，之后一些是sock的属性设置，包括所属协议族，状态，是否可以重用，标志位等等。
27. 之后一段在一开始有一个后备接收队列sk\_backlog，这个队列目前只用于tcp，传输控制块呗上锁后当由新的报文传递到传输控制块时，只能把报文放到后备接收队列中，之后由用户进程读取tcp数据时，再从该队列中取出复制到用户空间中。一旦用户进程解锁传输控制块，就会立即处理后备队列，讲tcp段处理之后添加到接收队列中。就在sk\_backlog下面还有一个sk\_forward\_alloc预分配缓存长度，这是一个标识而且目前只用于tcp。当分配的缓存小于该值时，分配必然成功，否则需要重新确认分配的缓存是否有效。共用体中\*sk\_wq和\*sk\_wq\_raw是进程等待队列。进程等待连接、等待输出缓冲区、等待读数据时都会讲进程暂存到此队列中。这个成员最初实在sk\_clone函数中初始化为null，该成员实际存储的socket结构中的wait成员，这个操作在sock\_init\_data中完成。还有就是上一页ppt中的sk\_receive\_queue和这里的sk\_write\_queue为接受和发送队列。Sk\_socket指向对应套接字的指针。之后这四个sock sk指针是当传输控制块的状态发生变化时或者数据准备就位时又或者是发生错误时，唤醒那些等待本套接字的进程。
28. inet\_sock结构是IPv4协议专用的传输控制块，是对sock结构的扩展，在传输控制块的基本属性已具备的基础上，进一步提供了IPv4协议专有的一些属性，如TTL、组播列表、IP地址、端口等。他的第一项是一个sock结构。Inet\_daddr和inet\_rcv\_saddr分别标识目的IP地址和已绑定的本地IP地址；inet\_dport和inet\_num分别标识目的端口和主机字节序存储的本地端口；uc\_ttl：单播报文的TTL，默认值为-1，在输出IP数据报时，TTL值首选会从这里获取；inet\_id是一个递增的值，用来赋给IP首部中的id域；inet\_opt指向ip数据报选项的指针；tos用于设置IP首部的TOS域；is\_icsk:1：是否是基于连接的传输控制块。最后还有一个inet\_cork\_full结构。
29. cork是用于udp或原始ip在每次发送时缓存的一些临时信息。如udp数据报或原始ip数据报分片的大小。其成员主要是一个inet\_cork结构和用flowi结构来缓存目的地址、目的端口、源地址和源端口，构造UDP报文时有关信息就从这里获取。Inet\_cork结构中包含了输出IP数据报的目的地址、指向此次发送数据报的IP选项、UDP数据报或原始IP数据报分片大小以及当前发送的数据报的数据长度。
30. inet\_connection\_sock结构体是所有面向连接传输控制块的表示，在inet\_sock基础上增加了有关连接、确认和重传的成员。大家可以从成员的作用中看出都是一些与连接、确认和重传有关的属性。
31. 最后是最大的tcp\_sock结构，也就是tcp协议的传输控制块，在inet\_connection\_sock基础上增加了滑动窗口、拥塞控制算法等一系列TCP专有属性，到这里我们可以看出这是一个一步一步的过程，不断的在控制块上添加不同属性对应不同的功能。Tcp\_sock中值得我们学习的有rcv\_nxt等待接收的下一个tcp段的序号，没接收一个tcp段之后都会更新该值；snd\_nxt等待发送的下一个tcp段的序号。
32. snd\_una在输出的段中，最早一个未确认段的序号；rcv\_tstamp最近一次收到ack段的时间，用于ack保活；lsndtime最近一次发送数据包的时间，主要用于拥塞窗口的设置；snd\_wll记录更新发送窗口的那个ack段的序号；snd\_wnd接收方提供的接收窗口的大小；max\_window接收方通告过的最大接收窗口值；mss\_cache发送方当前有效的MSS；window\_clamp滑动窗口最大值；rcv\_ssthresh当前接收窗口大小的阈值。
33. snd\_ssthresh拥塞控制时慢启动的阈值；snd\_cwnd当前拥塞窗口的大小；snd\_cwnd\_cnt自上次调整拥塞窗口到目前为止接收到的总ack段数；snd\_cwnd\_clamp允许最大拥塞窗口值；snd\_cwnd\_stamp最近一次检测拥塞窗口的时间；rcv\_wnd当前接收窗口大小；write\_seq已加入到发送队列中的最后一个字节序号。课后大家可以仔细研究代码后面的注释，相信这些名词学过计算机网络的各位都是非常熟悉的。
34. 接下来我们来讲讲proto和poto\_ops两个结构，proto结构为网络接口层，结构中的函数实现了传输层的操作和从传输层到网络层调用的跳转，每一种传输层协议都有对应的proto实例，有tcp\_proto, udp\_proto, raw\_proto，在网络模块初始化时，通过proto\_register()注册到全局proto\_list链表上。
35. 而proto\_ops结构中包含了某个特定的“protocol family"的一系列functions，其包含的functions和struct proto很类似，但是其在socket层，当socket相关的systemcall发生时， 首先从"proto\_ops"结构体中调用相应的function, 然后对应的ip-protocol function从proto结构体中被调用。对比两者代码，可以看到很多相似的函数。
36. 以上就是tcp传输控制块的结构，既然有结构那么就会有管理，TCP传输控制块存在多种状态：SYN\_SEND, SYN\_RECV, LISTEN, ESTABLISHED, CLOSE等等，其中LISTEN和ESTABLISHED这两种状态相对于其他状态是一种常态，这些状态会在后续的tcp三次握手和四次挥手中详细讲解。为了能对处于不同状态的传输控制块进行合理的管理与访问，TCP根据状态将传输控制块存储到多个不同的散列表中，比较重要的两个，一个是ehash另一个是bhash。
37. ehash指向大小为ehash\_size的inet\_ehash\_bucket类型的散列表，用来管理除LISTEN状态之外的传输控制块；而bhash指向大小为bhash\_size的inet\_bind\_bucket类型的散列表，管理端口绑定信息块。其结构体代码如下图。在TCP中实现hash接口的函数是tcp\_v4\_hash()和tcp\_unhash()。
38. 还有一个listening\_hash散列表用于存储LISTEN状态的传输控制块。对于listening状态以外的传输控制块都存储在ehash中，每一个又链接着tcp\_sock结构；而linstening状态的传输控制块都保存在listening\_hash中，链接方式与ehash相同。
39. 上半部分是TCP三路握手过程的状态变迁，下半部分是TCP四次挥手过程的状态变迁。首先是起始点closed在超时或者连接关闭时候进入此状态，这并不是一个真正的状态，而是这个状态图的假想起点和终点。然后服务器经过socket、bind、listen 函数之后进入listen状态，开始监听客户端发过来的连接请求。SYN\_SENT是第一次握手发生阶段，客户端发起连接。客户端调用 connect，发送 SYN 给服务器端，然后进入 SYN\_SENT 状态，等待服务器端确认（三次握手中的第二个报文）。如果服务器端不能连接，则直接进入CLOSED状态。SYN\_RCVD是第二次握手发生阶段，跟上面对应，这里是服务器端接收到了客户端的 SYN，此时服务器由 LISTEN 进入 SYN\_RCVD状态，同时服务器端回应一个 ACK，然后再发送一个 SYN 即 SYN+ACK 给客户端。状态图中还描绘了这样一种情况，当客户端在发送 SYN 的同时也收到服务器端的 SYN请求，即两个同时发起连接请求，那么客户端就会从 SYN\_SENT 转换到 SYN\_REVD 状态。ESTABLISHED是第三次握手发生阶段，客户端接收到服务器端的 ACK 包（ACK，SYN）之后，也会发送一个 ACK 确认包，客户端进入 ESTABLISHED 状态，表明客户端这边已经准备好，但TCP 需要两端都准备好才可以进行数据传输。服务器端收到客户端的 ACK 之后会从 SYN\_RCVD 状态转移到 ESTABLISHED 状态，表明服务器端也准备好进行数据传输了。这样客户端和服务器端都是 ESTABLISHED 状态，就可以进行后面的数据传输了。所以 ESTABLISHED 也可以说是一个数据传送状态。以上是三次握手过程的状态迁移。下面是四次挥手动作。第一次挥手FIN\_WAIT\_1，主动关闭的一方（执行主动关闭的一方既可以是客户端，也可以是服务器端，这里以客户端执行主动关闭为例），终止连接时，发送 FIN 给对方，然后等待对方返回 ACK 。调用 close() 第一次挥手就进入此状态。接收到FIN 之后，被动关闭的一方进入CLOSE\_WAIT状态。具体动作是接收到 FIN，同时发送 ACK。之所以叫 CLOSE\_WAIT 可以理解为被动关闭的一方此时正在等待上层应用程序发出关闭连接指令。前面已经说过，TCP关闭是全双工过程，这里客户端执行了主动关闭，被动方服务器端接收到FIN 后也需要调用 close 关闭，这个 CLOSE\_WAIT 就是处于这个状态，等待发送 FIN，发送了FIN 则进入 LAST\_ACK 状态。主动端（接收到被动方返回的 ACK 后进入FIN\_WAIT\_2状态。还有一种情况是两边同时发起关闭请求时（即主动方发送FIN，等待被动方返回ACK，同时被动方也发送了FIN，主动方接收到了FIN之后，发送ACK给被动方），主动方会由FIN\_WAIT\_1 进入CLOSING 状态，等待被动方返回ACK。四次挥手最后都会经过TIME\_WAIT进入CLOSED状态。
40. 然后我们从代码的角度来看看三次握手和四次挥手。首先是三次握手。
41. 对于服务器端，从接收到客户端的SYN请求起，必须等到客户端对服务端SYN+ACK段的确认，连接才算成功建立，在此期间，要将TCP连接的信息保存起来，直到用户调用accept()获取套接字为止，TCP控制块用连接请求块队列request\_sock\_queue结构来保存正在建立连接和已建立连接但未被accept的传输控制块，正在建立连接的即处于SYN\_RECV状态为半连接队列。这里再Linux更新中有一个commit，Linux4.4之前会将tcp\_request\_sock挂在listen\_sock下，收到 ACK 之后从 listening\_hash 找到 listen socket 再进一步找到 tcp\_request\_sock，而4.4之后直接把 tcp\_request\_sock 挂在 ehash 中，这样收到 ACK 之后可以直接找到 tcp\_request\_sock。而已经建立连接但未accept的即处于ESTABLISHED状态为全连接队列，rskq\_accept\_head指向的链表保存了已建立连接的sock，rskq\_accept\_tail指向此链表的尾部。
42. 三次握手我们以客户端主动打开为例，客户端主动打开是通过connect系统调用等一系列操作完成的，这一系统调用最终会调用传输层的tcp\_v4\_connect函数。在这期间，客户端共做了三个动作。发送syn段，接收syn+ack段，发送ack段。
43. 第一次握手构造并发送syn段。这是整个函数调用链。Tcp\_v4\_connect会初始化一个连接，主要作用根据用户提供的目的地址，设置好了传输控制块，为传输做好准备。最后调用tcp\_connect对syn包进行发送。
44. Tcp\_v4\_connect位于net/ipv4/tcp\_ipvs4.c文件中，参数分别表示传输控制块、地址结构和目的地址长度。因为是客户端主动打开，所以需要将tcp的状态设置为syn\_sent，在一些属性填充进传输控制块后，调用tcp\_connect函数。
45. Tcp\_connect首先会初始化tcp连接，然后将当前的sk\_buff添加到发送队列中，并调用tcp\_send\_syn\_data将syn包发送出去。因为之前write\_seq自增过，所以这表示的是下一个待发送字节的序列号，发送完syn后snd\_nxt就赋值下一个待发送的序列号，为下一次发送做准备。
46. 第二次握手是客户端接收syn+ack包，主要由tcp\_rcv\_state\_process函数完成。
47. 在tcp\_rcv\_state\_process函数中，因为之前的状态是sym\_sent，所以此处会先调用tcp\_rcv\_synsent\_state\_process来处理接收到的数据段。tcp\_rcv\_synsent\_state\_process会解析tcp选项，获取服务端的支持情况， 比如sack, TFO, wscale, MSS， timestamp等；如果有ack， 进行tcp\_ack， 这时候可能fastopen确认了之前的数据；调用tcp\_finish\_connect，TCP\_SYN\_SENT->TCP\_ESTABLISHED；如果包里没有ack，只有syn，则表示相互connect， TCP\_SYN\_SENT->TCP\_SYN\_RECV, 并发送synack。
48. 第三次握手是发送ack包，主要是由tcp\_send\_ack函数实现。先为数据报分配空间，然后通过skb\_reserve创建报头tcp\_init\_nondata\_skb()初始化一个ACK回复的TCP报文, 中间的tcp\_acceptable\_seq构造该报文的seq,TCPHDR\_ACK设置TCP报文的标志位；最后tcp\_transmit\_skb发送ack包。以上是从客户端的角度来看三次握手。
49. 从服务器端的角度来看，tcp想要被动打开，就必须得先进行listen调用，经过listen调用之后，系统内部其实创建了一个监听套接字，专门负责监听是否有数据发来，而不会负责传输数据。这个过程同样会经历三个步骤，等待接收syn段，回复syn+ack段，接收ack段。
50. 服务器端的第一次握手是接收客户端发来的syn段，主要由tcp\_v4\_do\_rcv实现。
51. 这里与上面接收syn+ack段相同，在Linux的tcp实现中，tcp\_rcv\_state\_process负责根据接收到的包维护tcp的状态机，这里不止能接收syn段，其他接收的段也都由这个函数处理。
52. Tcp\_v4\_conn\_request函数又调用了tcp\_conn\_request。在tcp\_conn\_request中回分配request\_sock空间，进入TCP\_NEW\_SYN\_RECV状态。之后将req sock插入ehsh散列表中，并发送synack。
53. 其实第二次握手和上一页ppt中的第一次握手是同一个函数实现，上一个ppt中最后调用了send\_synack，实际调用的就是tcp\_v4\_send\_synack。
54. 在tcp\_v4\_send\_synack中会根据当前的传输控制块、路由信息和请求信息等来构建syn+ack段，构建中会首先给skb分配空间，然后用到skb\_reserve函数来为MAC层、IP层、TCP层首部预留必要的空间，之后就是对tcp报的一写设置，最后返回一个skb。最后经过tcp\_v4\_send\_check函数进行验证并发送。
55. 服务器端的最后一次握手是接收ack段。由tcp\_v4\_do\_rcv实现。与之前接收syn段相同，调用tcp\_rcv\_stae\_process来实现接收段操作。
56. 回到tcp\_rcv\_stae\_process我们一定还记得函数中有一个switch-case语句，所以首先我们需要知道此时的状态。tcp\_rcv\_stae\_process是从tcp\_child\_process来的，传入的参数是child和skb。而child是从父socket生成的。如果child的状态和父socket状态一样，肯定是有问题的。那么child的状态是何时改变的呢？这里要回到回到创建child的函数tcp\_create\_openreq\_child->inet\_csk\_clone\_lock，因此，此时tcp状态是TCP\_SYN\_RECV，那下面让我们来看一下tcp\_rcv\_state\_process处理TCP\_SYN\_RECV状态的代码。
57. 首先会先检查该ack包，通过了才会进入下面操作。这里值得注意的是tcp\_set\_state(sk, TCP\_ESTABLISHED)，表示完成了三次握手，sk的状态自然改为 TCP\_ESTABLISHED，下面一句sk\_state\_change默认调用sock\_def\_wakeup, 会唤醒sleep在该socket上的进程。然后又是一处switch-case语句，但此时tcp状态已经被设置为TCP\_ESTABLISHED，所以只需要执行TCP\_ESTABLISHED处的代码tcp\_data\_queue。处理完tcp\_rcv\_state\_process后回到tcp\_child\_process中，此时state的状态即为child之前的状态，即TCP\_SYN\_RECV。而child->sk\_state在处理完这个ack包后变为TCP\_ESTABLISHED了，因此进入parent->sk\_data\_ready准备进行数据的传输。
58. 传输层下面就是网络层了。
59. 这是网络层的函数调用关系，其中最为重要的netfilter架构中的钩子，我们会在下一节netfilter架构中学习。这里先大致介绍一下各个函数的作用并主要介绍ip\_queue\_xmit函数。ip\_queue\_xmit函数是将数据包从传输层发送到网络层，设置路由、创建IP协议头和IP选项，调用dst\_output发送。dst\_output函数是一个函数指针，它指向ip\_output函数。ip\_finish\_output2函数最终通过dst->neighbour->output函数指针或hh->hh\_output函数指针调用数据链路层的dev\_queue\_xmit函数发送，这两个函数指针的初始化在/net/ipv4/arp.c文件中。
60. ip\_queue\_xmit()函数是ip层提供给tcp层发送回调，大多数tcp发送都会使用这个回调，tcp层使用tcp\_transmit\_skb封装了tcp头之后，调用该函数，该函数提供了路由查找校验、封装ip头和ip选项的功能，封装完成之后调用ip\_local\_out发送数据包。从代码中我们也可以看到ip\_queue\_xmit根据目的地址查找路由缓存，并将路由设置到skb中。
61. 而后就是对ip的设置，构造ip头，设置ip选项，设置id，最后通过ip\_local\_out输出。\_\_ip\_local\_out函数中除了设置ip协议另一个就是经过NF的local\_out的钩子点，这是netfilter中的内容，我们会在下一节重点讲解。
62. 最后是数据链路层。
63. 上节课讲过，数据链路层最重要的是net\_device结构体，存储着特定网络设备的所有信息，比如说硬件信息：与网络设备相关的底层硬件信息，如果是虚拟网络设备，则这部分信息无效；设备操作接口：包括操作数据或控制设备的函数接口，如发送数据报的接口、激活和管理设备的接口。注册后的net\_device实例会链入一个全局链表dev\_base中，同时，还会加入到dev\_name\_head和dev\_index\_head这两个散列表中。这里用不同的颜色代表不同的流。蓝色的是全局链表dev\_base，指向net\_device结构，又通过net\_device中的next指针指向下一个net\_device。红色的dev\_name\_head方便快速的找到对应的net\_device链。紫色的是dev\_index\_head散列表与dev\_name\_head作用相似，不同的是dev\_name\_head是以设备的名字为键值，而dev\_index\_head是以设备的索引为键值。
64. 在数据链路层准备向网络设备传输前需要做好准备工作，首先是检查设备是否打开，当net\_device->state中的\_\_LINK\_STATE\_START标志被设置时，设备就可被视为已开启·。这个值会在设备dev\_open时被设置，也会在设备dev\_close时被清除。当准备好后，数据就可以进入ip栈和NIC之间的驱动队列，每个队列有一个指针指向其相关关联的设备，以及一个指针指向存储输入/输出缓冲区的skb\_buff。协议栈向设备发送数据包时需调用dev\_queue\_xmit()，该函数对SKB进行排队，最终由底层设备驱动程序进行传输。
65. 那么网络设备如何知道有数据来了呢？一般的，外部设备与中央处理器交互一般有两种手段：轮询和中断。轮询的方式很简单，很多I/O设备都有一个状态寄存器，用于描述设备当前的工作状态，每当设备状态发生改变时，设备将修改相应状态寄存器位。通过不断查询设备的状态寄存器，CPU就可以了解设备的状态，从而进行必要的I/O操作。为了节约CPU资源，查询工作往往不是连续的，而是定时进行。轮询方式具有简单、易实现、易控制等优势，在很多小型系统中有大量应用。对那些实时敏感性不高、具有大量CPU资源的系统来说，轮询方式有很广泛的应用。最典型的用途就是在那些任务比较单一的单片机上，嵌入式系统中也有应用。但是轮询的方式也有很多不足：<1>增加系统开销。无论是任务轮询还是定时器轮询都需要消耗对应的系统资源。<2>无法及时感知设备状态变化。在轮询间隔内的设备状态变化只有在下次轮询时才能被发现，这将无法满足对实时性敏感的应用场合。<3>浪费CPU资源。无论设备是否发生状态改变，轮询总在进行。在实际情况中，大多数设备的状态改变通常不会那么频繁，轮询空转将白白浪费CPU时间片。中断，顾名思义，就是打断正在进行中的工作。中断不需要处理器轮询设备的状态，设备在自己发生状态改变时将主动发送一个信号给处理器，后者在接收到这一通知信号时，会挂起当前正在执行的任务转而去处理响应外设的中断请求。中断解决了外部设备与处理器之间速度不匹配导致的资源浪费问题。 现代设备绝大多数采用中断的方式与处理器进行沟通，因此设备驱动程序必须能够支持设备的中断特性。中断模型也并非十全十美，其高优先级的VIP待遇和快速响应要求在极端条件下将造成“活锁”效应。有时候需要发挥粗暴中断和温和轮询各自的优势，根据实际应用情景，在两种模式之间切换。Linux网络设备驱动模型中的NAPI机制 ，它采用“中断+轮询”的处理方式代替纯中断处理方式，是中断和轮询的完美合体。
66. Softnet\_data结构体描述了与网络软中断处理相关的报文输入和输出队列，每个CPU有一个单独的softnet\_data实例，在接口层与网络层之间起着承上启下的作用。其中较为关键的成员变量有：output\_queue数据包输出软中断中输出数据包的网络设备队列；input\_pkt\_queue非NAPI的接口层缓存队列；poll\_list是网络轮询队列；completion\_queue是完成发送数据包的等待释放队列；backlog\_dev是用于非NAPI驱动的虚拟网络设备，不代表具体的网络设备，用来兼容NAPI的驱动。
67. 图中展示的是softnet\_data与接口层和网络层的关系，当网络层通过调用dev\_queue\_xmit()函数将数据包输出到设备层，数据包会被排入到输出网络设备的排队规则的队列中去，在数据包输出软中断中，将排队规则的队列中的数据包出队列，并输出到网络设备中，最后在合适的时机，释放已经成功输出的数据包。
68. dev\_queue\_xmit()函数的实现流程图，GSO是基于TSO概念的一个归纳，TSO是一种利用网卡来对大数据包进行自动分段，降低CPU负载的技术。 其主要是延迟分段。GSO是协议栈是否推迟分段，在发送到网卡之前判断网卡是否支持TSO，如果网卡支持TSO则让网卡分段，否则协议栈分完段再交给驱动。 如果TSO开启，GSO会自动开启。如果不支持则会对数据包进行聚合分散IO。若支持流量控制，则将待输出的数据包根据规则加入到输出网络设备队列中排队，并在合适的时机激活网络输出软中断，依次将报文从队列中取出通过网络设备输出。若不支持流量控制，则直接将数据包从网络设备输出。
69. 数据传输完毕后需要通知内核已接受，此时有两种实现，一种是旧API用的是netif\_rx函数，这个过程大致就是把data拷贝到sk\_buff中，并注意保留两个字节使ip报头对齐16字节边界。而新的New API是混合使用中断事件和轮询，而不使用纯粹的中断事件驱动模型具体的流程如右图，可以看到综合了中断和轮询两种方式。目前还有很多NIC设备驱动程序没有改用NAPI。但是这个新的架构已经整合到内核中了，而且连netif\_rx和内核其余部分之间的接口也都把NAPI考虑进来了。
70. 在新的api下，网络设备向上层传输数据时回想将网络设备添加到待轮询的队列中，完成后激活报文接收的软中断，最后通过软中断和轮询将poll\_list中的网络设备读取的报文传递到网络层。和有图b中的相比NAPI可以减少了cpu的负载，也使得设备处理更加公平。