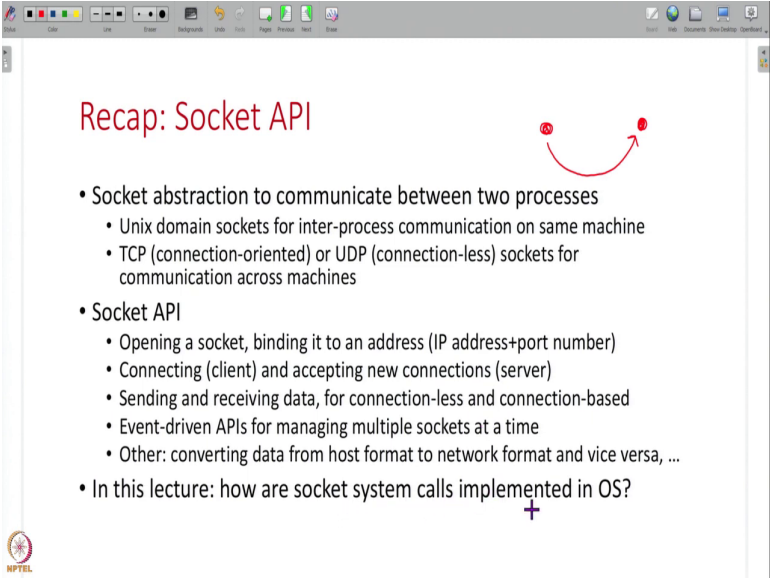


Design and Engineering of Computer Systems
Professor Mythili Vutukuru
Department of Computer Science and Engineering
Indian Institute of Technology, Bombay
Lecture 27
Network I/O Implementation

Hello everyone, welcome to the 19th lecture in the course Design and Engineering of Computer Systems. So, in the previous lecture, we have studied what is the socket API that is provided by the kernel to user processes to help them communicate with each other. In this lecture, we are going to understand how the socket API is implemented inside the operating system. And this understanding is crucial for us to understand how applications run in real life and what kind of performance they get and so on. So, let us get started.

(Refer Slide Time: 00:55)

A presentation slide titled "Recap: Socket API" in red text. To the right of the title is a diagram showing two red circular nodes connected by a red curved arrow, representing communication between two processes. Below the title is a bulleted list of topics. The slide is displayed within a window that has a standard operating system taskbar at the top with various application icons. At the bottom left of the slide content is a small circular logo with the text "IITB" inside. At the bottom center of the slide content is a small purple plus sign.

Recap: Socket API

- Socket abstraction to communicate between two processes
 - Unix domain sockets for inter-process communication on same machine
 - TCP (connection-oriented) or UDP (connection-less) sockets for communication across machines
- Socket API
 - Opening a socket, binding it to an address (IP address+port number)
 - Connecting (client) and accepting new connections (server)
 - Sending and receiving data, for connection-less and connection-based
 - Event-driven APIs for managing multiple sockets at a time
 - Other: converting data from host format to network format and vice versa, ...
- In this lecture: how are socket system calls implemented in OS?

So, what is the socket API? This is a small recap. We have seen that the socket is an abstraction that is provided as a way for two processes to communicate with each other. So, there is one process that opens a socket, server socket, another process that has a client socket, connects to this server socket and then they can exchange messages with each other whatever you send here will be received here and vice versa.

And you can also send messages to other sockets that you are not connected to. So, we have seen that there are local sockets for communication this way, between processes in the same machine, as well as internet sockets, or TCP or UDP sockets for communicating with processes across machines.

And we have seen various components of the socket API, how you open a socket, how you bind it to a well-known address, like an IP address plus port number at a server, then how you connect a client socket to a server socket, how you send and receive data. This slightly varies between connectionless and connection-based sockets, then how we use event driven APIs for managing multiple sockets at the same time if a server has to handle multiple clients concurrently.

There are also other functions that are provided in any communication a socket like API or any other programming language API. Like for example, the data that you send over the network is in a specific format and it could be different from the format in which the data is stored at the host. So, you have functions to convert from the host format to the network format and vice versa.

There are many other helper functions that we have not gone into more detail, but when you are using any library for communication, you should understand these things. So, this is the API. In this lecture, we want to understand how this system calls are implemented inside the operating system.

(Refer Slide Time: 02:49)

Overview of network communication

- Data is exchanged on a network in units of packets = sequence of bytes
- Communicating processes have unique network addresses
 - Computers on Internet have IP (Internet Protocol) addresses
 - Multiple processes at one IP address are differentiated by port numbers
- Every packet has sender/receiver IP addresses, port numbers
 - Network routes packets from source to destination using these addresses
- Machines on a network communicate using multiple protocols
 - Protocol specifies which message to exchange with each other, format of messages
 - Example: TCP+IP protocols used for reliable connection-oriented communication
 - Example: UDP+IP protocols used for unreliable connection-less communication
- Packet sent over network = payload (actual data sent by users) + protocol-specific headers (IP address, port numbers, other metadata)
 - User program reads/writes payload using socket API
 - Protocol processing, encapsulating/decapsulating headers done by OS

Handwritten red annotations on the slide include: 'payload' and 'socket' near the bottom node, and 'header' near the central node.

So, before we understand how these system calls or the socket-based system calls are implemented inside an OS, I would like to give you a brief overview of how network communication itself works. So, this is a topic that we will study in more detail next week. But here I would like to give you a brief introduction to how network communication works.

Of course, if you have taken a networking course before you should be familiar with it, but if you are not also here is a summary. So, over the network like the internet data is exchanged in units of packets that is you will take some number of bytes put them into a packet and you will send this packet, the next set of bytes you will send another packet. So, data flows in the form of a series of packets between multiple machines on the internet.

And these communicating processes that are sending and receiving these packets, they all have unique network addresses. So, every machine has an IP address and inside a machine the different processes when they open sockets, they will get different port numbers. So, this combination of this IP address and port number uniquely identifies a server socket, uniquely identifies a server process.

So, therefore, whenever you have to send a message to any server, you will put the server's IP address and port number as part of your packet so that the message reaches the server. So, every packet has the sender's addresses, IP address port number as well as the receiver's IP address port number. And using this information, the network has a series of routers.

So, this machine is sending some series of packets and all of these routers will look at, okay this packet has to go to this person to this address and they will route the packet and finally your packet reaches the server. So, you will have both the sender's address as well as the receiver's address in the packet.

And machines on the internet, they communicate using various protocols, what is a protocol, is nothing but some rules we have to exchange messages like this, this is how the message should look like, this is the format of the message. You have to agree on some things, otherwise if this guy sends some random gibberish bytes then this guy cannot understand what he is saying.

So, there is a common language that these processes used to communicate over the network which is called a protocol. There are many protocols for example, the TCP IP protocol is used for reliable communication, the UDP IP protocol is used for unreliable communication, the HTTP protocol is used to exchange information about websites, we will study all of this next week.

But for now, you need to understand that there are some protocols and each protocol will have its own headers. For example, the IP protocol may add its IP address, the TCP protocol may add port number. So, there is all of this other protocol specific information

that is there in a packet. So, every packet has what is called the payload, which is the actual data that the user sends into the socket, you are sending some message to the other side, you will write some message into a socket that is your payload.

And once you write the payload into your socket, before it is actually sent out over the network, a few other extra pieces of information are added to the packet which are called the headers. These headers are usually added by the operating system when it does this protocol processing like TCP processing or UDP processing, it will add these headers here and on the other side, you will remove the headers.

So, the user program only sees the payload, but the OS takes care of adding and removing these headers. And finally, the packet that gets sent over the network has both payload as well as headers. So, what these protocols are, what headers do they add, what information is there in the headers, all of this we are going to see next week when we study the design of the internet in more detail.

So, now with this knowledge that there are certain packets that are sent and received which have certain payload and certain headers corresponding to various protocols now that we have this information, now let us understand how the socket API is implemented in operating systems.

(Refer Slide Time: 07:31)

Socket buffers and queues

- Opening a socket returns socket file descriptor
 - Index into file descriptor array of process, which points to open file table
 - Open file table contains pointers to inode for files, socket structures for sockets, ...
- Socket buffer (skb or sk_buff in Linux) is a kernel data structure to store network packets (payload + headers)
- Every socket has two socket buffer queues (transmit and receive queues)
- Send/write data into socket
 - New skb is added to socket TX queue, payload copied from user memory into skb, OS processes packet (adds headers) and transmits via device driver
 - When packet transmitted over network and no longer needed, skb is freed
- Receive/read from socket
 - Dequeue skb from RX queue, payload copied from skb to user memory, skb is freed
 - If socket RX queue empty, process blocks until: data received from network, OS processes packet, adds skb to socket RX queue

Handwritten diagrams include: a 'socket' label with arrows pointing to TX and RX queues; a diagram showing a message (msg) being added to a TX queue and then to a skb; and a diagram showing a skb being dequeued from an RX queue and its payload being copied to a user memory block.

So, when you open a socket, we have seen this before you will get a socket file descriptor. This is nothing but your index into the file descriptor array which points to open file table all of it is the same that high level sockets are treated the same as files, just that instead of

inode for a file for a socket you have separate structures, socket does not need inode, it has some other pieces of data associated with it which is kept track of in the open file table. So, every socket has queues of socket buffers.

So, socket has a queue, a transmit queue and a receive queue and this queue has what are called socket buffers. This is the socket buffers you are transmitting and this is the socket buffers you are receiving. So, when you write socket buffers go here, when you read socket buffers come from here. But what is a socket buffer?

A socket buffer is nothing but a data structure that is used to store network packets. It is called as skb or sk_buff in Linux and it has nothing but some payload as well as headers. A socket buffer will have variable size they will by many different headers that are added to the payload and all of these together is called the socket buffer or skb in Linux. And every socket stores two queues of all the socket buffers or packets that it is transmitting and all the socket buffers are packets that it is receiving.

Every socket is a bi directional communication, you can send as well as receive. So, when you use the send or the write system call to write into a socket, what happens a new skb is added to the transmit queue. And in this skb, you have some payload and some headers. You have headers and payload into this skb. When you write you are giving some message to the socket API to the write API, this message is copied into the payload and these headers are added later by the operating system.

So, when you write into a socket, a new skb is created. The payload is copied from the message, the users space message the buffer that you have given to the write system call from that you create the payload, then the OS will later based on whatever protocols it is running, it will add some headers.

And finally, this skb is given to the device driver and then the device driver will send it out over the network, you will have some network card like an Ethernet card or a Wi Fi card, which will have its own device driver, and then this skb finally, we will go over the network as a packet.

At a high level, there is of course, a simplified description, but this is roughly what happens. And once you send this bits in this skb we are sent over the wire or wireless, over the network they are sent then this skb is freed up. So, the skb is allocated when you write

into the socket, it is constructed using the payload of the user and the headers and when it is sent out this skb when it is no longer needed, it is freed up.

Then what happens when you receive, when you read from a socket, there are already skb's that have been received that are queued up from this queue, you will take the payload information that is there in the skb and copied into the user provided buffer. Once again there is an skb, this payload the reader the receive system call gives some buffer some message as an argument you will copy this payload into that read system calls argument.

But what if your received queue does not have any packets, then what if the user says read from a socket but the receive queue of the socket is empty, in which case the process will block, you will block until some data is received from the network then the OS will put an skb queue it up here until that is done this process will block.

So, this is roughly how sending and receiving data through a socket happens where during the send time you will create a skb's send it out, during the receive time the OS where the packets are coming OS will queue up the skb's here and the socket read system call or receive system call will copy the data into user space from the skb.

(Refer Slide Time: 12:15)

Network device driver

- Device driver talks to network interface card (NIC) to exchange packets
- Device driver maintains TX/RX rings of packet descriptors in main memory
 - Ring = circular array (loop back to start upon reaching end)
 - Packet descriptor = pointer to socket buffer and other info about packet
 - Head/Tail = pointers to start/end of occupied slots in ring
 - NIC knows locations of TX/RX rings and can access it in memory
- TX ring is queue of skb waiting to be transmitted
 - Device driver adds skb to TX ring when packet is ready to be sent
 - When NIC free to transmit, NIC reads address of next skb from TX ring, DMA packet from skb into device hardware
 - When transmit complete, NIC raises interrupt, skb in TX ring is freed
- RX ring is queue of empty skb waiting to be filled by received packets
 - Device driver initializes RX ring with pointers to empty skb
 - NIC receives packet, find empty skb on RX ring, DMA received packet into skb, raises interrupt
 - OS handles interrupt, processes received packet, hands it over to corresponding socket RX queue
 - When received packet dequeued from RX ring, new empty skb is replenished by OS

Handwritten notes and diagrams include: 'device driver' with arrows pointing to TX and RX rings; 'NIC' with arrows pointing to the TX and RX rings; and a circular diagram representing the ring structure with 'TX' and 'RX' labels.

So, next is how are actually packets sent over the network, for that we have device drivers. So, a device driver is a piece of software that talks to I/O devices, we have seen this before. And network cards also called NICs or network interface cards have their own device drivers, which basically exchange packets with the network. So, every device driver maintains what are called transmit and receive rings.

So, the device driver will have a transmit ring and a receive ring. So, ring is nothing but a circular array. So, that is you will write in the array like this when you reach the end you will once again overwrite. So, it is nothing but a fixed size circular array and this is the transmit ring and this is the receive ring. What do these rings contain? So, these are maintained by the device driver that is the software inside the OS.

These rings will contain pointers to the socket buffers, these rings contain packet descriptors which are nothing but pointers to `sk_buffs`, `skb`, socket buffers. These are all the socket buffers that have to be transmitted, these are all the socket buffers that have been received. So, you have a ring of socket buffers the TX ring and the RX ring. And the actual hardware device itself, the NIC itself knows the location of these transmit and receive rings in memory.

So, now all of these rings this is all just memory, it is located in main memory, these are all OS data structures, data structures maintained by the device driver. So, all of this is there in RAM, but the NIC knows how to access this transmit and receive rings. For example, the NIC knows here is the first packet that have to transmit, here is the next packet after transmit, it can read, it can access these transmit and receive rings.

So, every ring has a head and tail pointer that is these are all the occupied slot, these are all the packets that are there to be transmitted, all of that information is known to the NIC. So, once the NIC knows the transmit and receive rings and how to access it, where are the packets, where is the queue starting, where is the queue ending, it can access these rings in order to do the transmission and reception.

So, let us look at it in a little bit more detail. What is happening at the transmit ring? The transmit ring is nothing but the queue of socket buffers that are waiting to be transmitted. Whenever the user has created packets to be transmitted, they get queued up at this transmit ring. Now, the device driver it will add the `skb`'s to this transmit ring when the packet is ready to be sent.

And then the NIC whenever it is free to send it will access the `skb`, do a DMA of the packet from the `skb` into the actual device hardware. So, the NIC has its own hardware where it converts these bits into electrical signals, to send it over the wire and all of that. So, the NIC is doing its own processing on the packet. So, the first thing the NIC will do is it will DMA the packet from ram into its own hardware memory and from there it will go ahead and transmit the packet.

So, how does the NIC know DMA, where in memory this skb is located, that is where this transmit ring, that is why the transmit ring has all the pointers. This is the address of the first packet to send, this is the address of the second packet to send, all of that is there in the transmit ring. So, the NIC accesses the transmit ring, sees the address of the first skb to send then gives that address to its DMA hardware which will copy the packet into the device and then the packet is sent.

And once the packet is transmitted then the NIC raises an interrupt then this skb can be freed up, you are done transmitting this packet, it is done. Similarly, we will be done with the next packet, the next packet and so on. Is that clear? So, the transmit ring is a queue of skb to be transmitted, the device driver will add skb's and the NIC will DMA these skb's and raise an interrupt at which point these skb's will be freed up on the transmitter. Now, what about the receive ring?

The receive ring is a queue of empty skb's that are waiting to be filled. So, the device driver it will create a bunch of empty packets and keep them in the RX ring. And whenever a packet arrives at the NIC it will find the first empty packet, find the address of this packet, DMA the received packet into this skb and then raise an interrupt. So, this RX string is used, is looked up by the NIC when it has a packet, now that it has a packet it needs to know where to DMA it into, it cannot just randomly dump it anywhere in memory.

So, it will read the RX ring, find out which is the first free slot, put the packet there, then the next packet it will put it in the next free slot and the next free slot and so on. And then when the OS handles the packet of course it will take this received packet do whatever processing and take it back to the socket queue. You remember the socket queue in the previous slide. So, from the RX ring this skb has to go to the socket queue that is where the user program will read it from.

So, once the NIC receives the packet it will DMA it into the skb that is there on the ring and then the OS will handle this interrupt, once that the DMA is done the interrupt is raised, the OS will handle the interrupt take this skb off to the RX queue. And when you pull an skb, filled up skb from this RX ring, of course, you will replenish it with a new empty skb, why? So, that the next time later on in the future another packet can be received. So, this RX ring has a fixed number of slots. So, when you take a packet away you will replenish that with a new empty skb. So, this is about the device driver and the TX and the RX ring.

(Refer Slide Time: 18:40)

The slide is titled "Packet transmission" in red. At the top, there is a handwritten diagram showing a "Socket" with "TX" and "RX" queues, and a "driver" with "TX" and "RX" queues. Below the title, there is a list of bullet points. The first bullet point is "Summary: TX/RX queues at socket, TX/RX rings at device driver". The second bullet point is "How is a network packet transmitted?". This is followed by four sub-bullets: "Write/send system call allocates new skb in socket TX queue, copies data from user memory into skb", "OS performs network protocol processing, adds headers to skb", "When network protocol decides to send packet, OS adds skb to TX ring (note that some network protocols may slow down transmission for congestion control and other reasons)", and "When device is free to transmit, DMA packet from TX ring into device, raise interrupt when transmission complete". The final bullet point is "OS handles interrupt, frees up skb in TX ring". There is a handwritten red circle around "device NIC" and a red box around a diagram of a packet with headers. A purple plus sign is at the bottom center.

Packet transmission

Socket TX RX
driver TX RX

- Summary: TX/RX queues at socket, TX/RX rings at device driver
- How is a network packet transmitted?
 - Write/send system call allocates new skb in socket TX queue, copies data from user memory into skb
 - OS performs network protocol processing, adds headers to skb
 - When network protocol decides to send packet, OS adds skb to TX ring (note that some network protocols may slow down transmission for congestion control and other reasons)
 - When device is free to transmit, DMA packet from TX ring into device, raise interrupt when transmission complete
 - OS handles interrupt, frees up skb in TX ring

device NIC

+

Now, let us put everything together. Let us see how packet transmission and packet reception happens end to end. So, to summarize, you have your socket and this socket has TX and RX queues which are skb's that are queued up at the socket for transmission and reception. And then underneath this TX and RX queues at the device driver level you have TX and RX rings and then finally you have your device your NIC itself. This is the driver.

So, how is the network packet transmitted? When the user does a right or a send system call, a new skb is allocated in the TX queue. And the data is copied from the user space into this skb then the OS performs all the network protocol processing like the user has filled in the payload, the OS will add the various headers and then this skb is finally pulled out from this TX queue and queued up at the TX ring of the device driver when the OS is ready to send the packet.

Note that as soon as the user writes you may not give the packet to the device driver to the TX ring because as we will see next week some protocols might want to slow down transmission when there is congestion, when too many packets are getting lost or some for various reasons, you might want to also slow down. So, the OS will wait for a little while when it thinks it is ready to send the packet it will take it from this TX queue and put it into the TX ring that the device has access too.

Then the device will DMA the packet from the TX ring and when it has sent the packet over the wire or wireless whatever the case may be, then it will raise an interrupt at which point this skb is freed up. Once the OS handles interrupt it frees up the skb because the

packet has been transmitted fully. This is the life cycle of a packet when you send it over the socket interface. Of course, what these protocols are and all of that you do not have to understand now, this we will see next week.

(Refer Slide Time: 21:06)

Packet reception

- How is a network packet received?
 - Device driver populates device RX ring with empty skb
 - If process makes read/recv system call before data is received, process is blocked
 - When packet received, NIC performs DMA of packet into empty skb in RX ring, raises interrupt
 - OS handles interrupt, performs minimal processing (e.g., acknowledge interrupt) in interrupt handler (top half), schedules another interrupt handler process (bottom half) to run when CPU is free
 - Bottom half interrupt handler removes skb from RX ring, replenishes RX ring with fresh skb, processes received packet, adds received skb in socket RX queue
 - When recv system call returns, data copied from skb into user memory, skb freed up
- How is socket identified based on received packet?
 - For connection-less sockets, receiver IP address/port number uniquely identifies socket
 - For connected sockets, sender/receiver IP address/port number (4-tuple) identifies socket

The diagram shows a 'Socket' box with 'TX' and 'RX' queues. A 'NIC' (Network Interface Card) is shown below, with a 'DMA' arrow pointing from the NIC to the RX queue. A 'listen' label is at the bottom right.

The next thing is packet reception. How is a network packet received? We start with once again there is your socket has TX and RX queues and then you have your device driver that has TX and RX rings and finally you have your NIC, your hardware. So, initially the device driver on the RX ring it will put a bunch of empty skb's which are just placeholders where received data can be put in.

Now, when the process makes a read system call if this receive queue is empty, if some packets have been received before well and good otherwise if this received queue is empty, anybody who makes a read system call will block until this received queue has packets until the socket received queue has packets. Now, when a packet arrives, the NIC will find the RX ring, find an empty buffer in the RX ring and DMA the packet into this skb and it will raise an interrupt.

Now, when the OS has received this interrupt what it has to do, it has to process this received packet. It has to do various protocol processing, look at the headers everything, this will usually take time. So, what the OS does is, it will split its interrupt processing into two parts. First is it will run what is called the top half which does the bare minimum work, which is telling the NIC, okay, got it. If the NIC knocks at your door, you say, wait, I am coming, I heard you I am coming.

That is all it does at first acknowledge the interrupt, why? Because you have interrupted a process and you do not want to do a lot of work as part of handling the interrupt. Therefore, the OS as part of handling the interrupt it will first acknowledge the interrupt in a piece of code that is called the top half. And later on, when it is free, it will run the remaining part of the interrupt handling which is called the bottom half.

That is the interrupt handling is split in modern operating systems into two parts called the top half and the bottom half. And when this bottom half interrupt handler runs that will remove the skb from this RX ring, run it through whatever protocol processing needs to be done. And finally queue it up at the RX queue of the socket. And of course, when you pluck out a slot from here, you will replenish it with an empty skb.

So, that in the future once again, when the circular buffer wraps around, another packet can be put over there. And when the receive system call returns. Now, once something comes in the RX queue this process that block will be woken up and it will copy data from the skb into its user memory and then the skb is freed up. So, this is the lifecycle of a received packet.

The NIC DMA's into one of the socket buffers that are whose pointers there in the RX ring, then the OS interrupt handler the top half and bottom half together handle this skb queue it up at the RX queue of the socket and then when the read system call or the receive system call of the process returns, it will copy from this skb to user memory.

Now, you might have many sockets in our system then how do you identify which socket received you should I dump this packet into, that is where your IP address port number will be useful. If you have connectionless sockets of course, you will only have one socket at a particular IP address and port number. So, you will uniquely identify the socket.

But if you have connected sockets, then what at the same port number you have a listen socket and you have multiple connected sockets all of these are at the same server address at the same port number. So, then which of these socket queues will you add the packet to, this is where you will also look at the sender IP address.

All of these connected sockets have different senders based on that information you will distinguish and based on which sender it came from you will find the suitable socket that is you will use what is called the 4-tuple of a connection that is four pieces of information which is the sender IP address, sender port number, receiver IP address, receiver port

number, these four pieces of information will be used to identify one of the connected sockets and then you will queue up the skb at that receive queue. So, this is the lifecycle of packet reception.

(Refer Slide Time: 25:44)

Optimizations to packet reception

- Receive-side processing of network packets split into top half and bottom half interrupt handlers: why?
 - Top half does minimal processing, to avoid disruption to interrupted process
 - Bottom half (soft IRQ) is separate process that is scheduled when CPU is free, does processing related to various network protocols
- One CPU core may not be able to keep up with interrupt processing on high speed network cards (~100Gbps today)
 - Receive side scaling (RSS) feature in NICs allows NIC to have multiple RX/TX rings
 - NIC splits received packets among multiple RX rings (packets of one connection are kept in the same ring, use hash of connection 4-tuple to pick RX ring)
 - Each RX ring is assigned to separate core, interrupt handling distributed to CPU cores
- Another optimization: NAPI (new API) to reduce interrupt load
 - Once interrupt is raised, all future interrupts disabled till bottom half runs
 - Bottom half polls all packets received until then, reenables interrupts

Now in general, this packet reception is a very overhead filled process, why? Because there is a lot of protocols, you have to look at headers, you have to do various computations, calculations, check sums, we will see all of this later. So, when all of this is being done, your OS can get overwhelmed and therefore there are many techniques that are available to speed this up.

Especially, today you have multi Gbps, 10s to 100s of gigabits per second, lot of packets coming in. So, how do you efficiently process all of these packets? That is the question. So, some techniques that the operating system uses are one we have seen, which is splitting the interrupt processing into a top half and a bottom half. As soon as the packet comes when an interrupt is raised, you are stopping an existing running process to handle this interrupt.

Therefore, you do not want to take much time, you will do a minimal processing, which is called the top half which just involves acknowledging the interrupt then you will schedule a separate process whose only job is to run these bottom half interrupt handlers. And when that process runs, the bottom half runs which is also called the soft IRQ. When that process runs, you will schedule it whenever the CPU is free when nobody else is running.

And that will do all the protocol related processing and take the skb from the RX ring to the RX queue of the socket. So, this splitting will help you avoid interruptions to existing

processes. Then the other optimization we do is that one CPU core may not be able to keep up with all the interrupt processing normally every device raises an interrupt on a particular CPU core.

But if you have hundreds of Gbps, millions of packets coming in per second then one CPU core will be like overwhelmed in doing all this interrupt processing, therefore what modern systems do is they use a feature of NIC is called receive side scaling. What is RSS? Instead of one TX/RX queue you have multiple TX/RX queues and the NIC when it gets a packet it will split the packets into these multiple queues.

And each of these queues will be handled by a different CPU core, C0, C1, C2 and so on. There are multiple TX/RX queues and each queue, so NIC will DMA one packet into this queue then the CPU will core will be interrupted, next packet goes here this core will handle the interrupt, so you are splitting the load of interrupt processing across multiple CPU cores.

And how do you split packets? You try to ensure that the packets of the same connection are in the same ring so that you do not jumble things around too much. So, you will typically use the hash of this 4-tuple source receiver, source destination, IP port numbers in order to split packets into these receive queues at the RX ring level. And each RX ring these interrupts are handled by a separate CPU core that way you can use your multiple CPU cores to handle this large number of interrupts that are coming in on high-speed network cards.

And the other optimization that is commonly used is what is called the NAPI optimization. What this means is that suppose at your RX ring, you have multiple slots and your NIC has put a packet here, DMA the packet here it has raised an interrupt. Now, the CPU is very busy it has not yet handled this packet, then another packet has come again you raise an interrupt, again you raise an interrupt instead of constantly knocking on the door when the other person is not opening the door you should stop knocking because you should realize that they are busy.

So, that is what this NAPI does. Once an interrupt is raised until the bottom half runs and handles interrupt all future interrupts are disabled, that does not mean packets will not stop coming, packets will keep coming, you will just keep adding them to the RX ring but you will not constantly keep raising interrupts and interrupting the CPU that is already overwhelmed.

The NIC will just silently keep DMAing packets without raising any further interrupts you have told the OS once you wait until it responds and once the bottom half runs, then it will not just look at this one packet but all the packets that have come until then, it was interrupted when one packet came but when it actually runs many more packets could have arrived and it will handle all of them. And once it has cleared the backlog it will re enable interrupts again. This avoids too much interrupt load in a system that is receiving packets at a very high speed.

(Refer Slide Time: 30:20)

The slide is titled "Performance tuning" in red. Above the title, there are hand-drawn diagrams: two vertical rectangles representing queues, and two circles representing rings, with arrows indicating data flow between them. The slide contains the following bullet points:

- Multiple queues: socket TX/RX queues, device TX/RX rings (finite size)
- Mismatch in speed of network, NIC, OS, application can lead to queues building up and overflowing, packet drops, poor performance
 - If packets arriving at very high rate on NIC, but OS not handling interrupts fast enough, device RX ring can overflow, packets dropped by device
 - If application reading packets very slowly from socket, socket RX queue hits maximum value, packets dropped
 - If device too slow in transmitting, device TX ring and socket TX queue become full, send/write into socket can block
- Best performance achieved when speeds of all components and queue sizes are matched (more on this topic later)
 - Sender must adjust sending speed based on capacity of network and receiver
 - Socket queue size and device driver ring size must be tuned for optimal performance

The NPTEL logo is visible in the bottom left corner of the slide.

So now, we have multiple queues in the system. And therefore, you might have the question what should I set these queue sizes to, you have at the socket you have these TX/RX queues then you have these TX/RX rings at the device driver and what should these sizes be? So, you have to set these sizes very carefully, because packets are coming over the network first they are sitting over here, then they are sitting over here, then the application is reading them. So, there is a pipeline.

And in the pipeline, you need to exactly match the speed of all the stages in the pipeline, if anybody is fast the other person is slow then what will happen your queue will build up. For example, if packets are coming very fast on the NIC and this RX ring is getting filled up. But the OS is not able to take the packets out and put them into the socket queues. Then what will happen your RX ring after you DMA packets into the entire ring.

Once you go around the ring, you will start overwriting packets, you will start dropping packets, there is no place to put them anymore, you will just drop the packets that are

coming in over the network that can happen. Or the other hand if the application is reading packets from the RX queue very slowly, then you know the OS is putting packets into the RX queue. But the application is not reading them, only when the application reads from the socket queue can you free of that slot.

So, that can also happen. It can also happen that the device is very slow, you could be writing very quickly into this TX ring, but the device is not sending packets. So, a lot of things can happen, queue can build up here, queue can build up here, queue can build up at many places. And whenever this queue builds up, packets will be dropped, you will get performance issues. You have sent some message to the server, the server is not getting your message and you have performance issues.

So, later on in the course, when we study about performance when we study about networking, we are going to see how to fix all of these sizes based on the speeds of the various components if your network is too fast if your network is too slow, if your application is too fast to slow, based on all of these things, how do you set all of these queues, how do you harmonize all of these stages of a pipeline such that you get good performance that is what we are going to study later in the course.

You must adjust your sending speed, you must adjust your queue sizes, your device driver ring sizes all of these must be tuned for optimal performance. And how you do this is something that we will see later on in the course. For now, it is enough for you to understand that there are many queues. And this is a pipeline one after the other and any time somebody is slow and somebody is fast in a pipeline queue will build up.

So, the final topic that I would like to touch upon today is that modern computer systems are doing I/O at increasing speed. So, initially when the operating systems were designed, you were doing I/O at few Mbps or Gbps but today you are doing I/O at 10s to 100s of Gbps especially in modern computer systems and data centers and the cloud. You have very high speed I/O. And the operating systems are not really designed to handle these millions of packets coming in per second. For every packet, there are inefficiencies.

(Refer Slide Time: 33:42)

Kernel bypass techniques

- I/O subsystem in OS not designed for high speed network I/O, has inefficiencies:
 - Interrupts, system calls for every packet, leads to expensive traps and context switches
 - Dynamic skb allocation for every packet, adds overhead
 - Packet data copied twice: device to skb, skb to user memory
- For high speed network I/O (~ 100Gbps), modern computer systems employ kernel bypass techniques, e.g., Data Plane Development Kit (DPDK)
 - Uses special device driver to access NIC, regular kernel processing fully bypassed
 - Packets DMA directly into userspace memory (zero copy)
 - Preallocate packet buffers in huge pages (efficient memory access)
 - Avoid interrupts, application itself checks RX ring periodically (polling)
 - Access multiple packets at a time from RX ring (batching)
- Disadvantages of kernel bypass: kernel isolation mechanisms and network stack processing fully bypassed, regular kernel networking tools do not work
 - Mainly useful in applications that process very high speed I/O

Handwritten annotations in red ink:

- user → skb → device
- device driver + NIC
- user buf

For every packet there is a system call, there is an interrupt you have to switch from user mode, kernel mode, context switch, and you know every packet you have to allocate skbs one by one, you have to copy the packet data, how, the device will DMA the data into skb which is there in the RX ring and this skb you will take you will put it into the RX queue and from here the data is copied into user whatever buffer user has given to the read system call, the data is copied there. So, you are copying data twice. So, there are all of these inefficiencies.

And therefore, sometimes your computer cannot handle all the network data that is coming in. So, in such cases there are techniques available in modern computer systems which are called kernel bypass techniques, which basically say, okay, let me get rid of this OS layer in between I will directly access the network card and I will do my own work. So, some applications that need high speed I/O can bypass the operating system.

So, for example, you can use your own special device driver that directly accesses the NIC and the NIC will directly DMA packets into your user space. This device driver, special device driver can do that. So, there are frameworks like what is called the data plane development kit, or DPDK.

This is a one popular example where the device driver does this, it will directly access the NIC, you are no longer using the kernel, interrupt, system call all of that is gone, you directly access the NIC take all your data, put it into a user space buffer so that there is zero copy, there is no copying across the kernel and user memory, you will pre allocate buffers,

instead of doing skb one for every packet, you will allocate a large pool of buffers at the same time and you will use optimizations like huge pages we have seen this before large page size means good TLB performance, you will use all of those concepts, you will avoid interrupts.

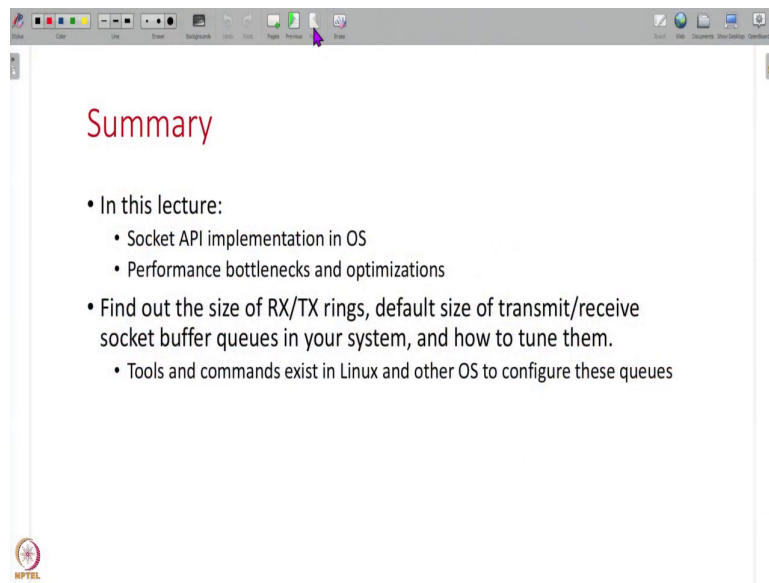
Whenever the application is free, it will directly poll the NIC instead of the NIC constantly interrupting the process and disturbing it, you might use polling, you might use techniques like batching where whenever you go to the NIC you will handle a batch of packets at a time, your protocol processing will happen on a batch of packets at a time instead of doing it one by one.

So, frameworks like DPDK today which are called kernel bypass techniques, employ all of these optimizations so that you can, you are able to handle hundreds of Gbps of packets in your application. So, these are all advanced topics that I will not cover in more detail here. But I just want you to know that this is an active area of research and if you are building a real-life system, you might have to employ these kernel bypass techniques sometimes.

Of course, not everything is very easy with kernel bypass techniques, the operating system is providing you various isolation mechanisms and there are various tools in the operating system that are useful. So, all of those are gone, the OS is out of the picture here. Directly, our application is using a device driver talking to the NIC, the OS is not involved. So, this has its own complications with respect to isolation, security or portability of your code and all of that, but that aside, if your application really needs high speed I/O, this is a technique that a lot of people are considering today in real life systems.

So, that is all I have for this lecture, I have talked to you about how the socket API is implemented inside the operating system and I have also spoken to you about what are some of the performance issues that can arise and what are some of the techniques used to optimize these performance issues in modern computer systems.

(Refer Slide Time: 37:23)



So, as a small exercise, you can do this you can try to find out the size of your RX/TX rings and your transmit receive queues of sockets in your system. There are simple commands in Linux that let you find out these numbers to tune them. So, play around with them to understand what these various things are in your system to understand the concepts of this lecture better. Thank you all, that is all I have for this lecture and see you in the next lecture. Thanks.