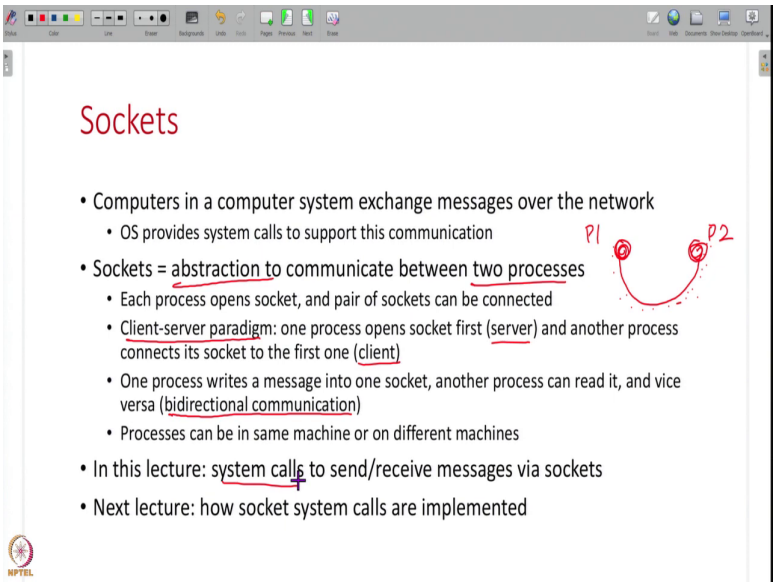
Design and Engineering of Computer Systems
Professor Mythili Vutukuru
Department of Computer Science and Engineering
Indian Institute of Technology, Bombay
Lecture-26
Network I/O via Sockets

Hello everyone, welcome to the 18th lecture on the course Design and Engineering of Computer Systems. In the previous two lectures, we have studied how file systems work, in this lecture and the next we are going to study how network I/O works in an operating system. So, let us get started. So, in all modern operating systems provide this abstraction of communication between two machines via a concept known as sockets.

(Refer Slide Time: 00:48)



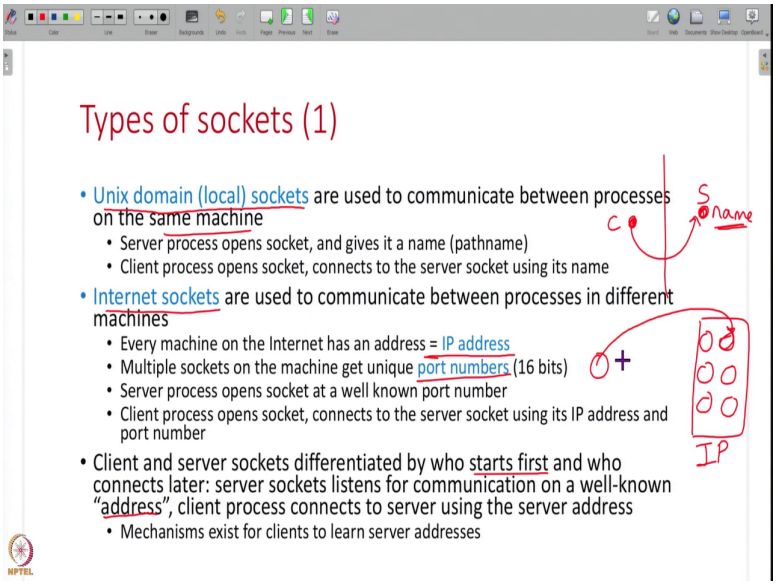
So, any computer system you need machines to exchange messages with each other. And a socket is an abstraction using which you can actually exchange these messages. And a socket is a way to communicate between two processes. These processes can be in the same machine and different machines does not matter, the idea of the sockets remains the same, it is a way for two processes to communicate with each other.

So, one process P1 opens a socket, another process P2 opens a socket, then these two sockets connect to each other. And then P1 can send messages in the socket P2 will receive it, P2 can send messages into the socket P1 will be able to read it from its socket. So, you can think of sockets as two endpoints of a pipe. So, normally there is the notion of a client and a server in a socket that is the process that opens the socket first is called the server.

And another process which connects to the first guys process is called a client. So, anybody can be a client, anybody can be a server, it is just that who starts first and who starts second. So, that is why sockets are set to follow the client-server communication paradigm. Then the other thing to note about sockets is it is bidirectional, just because one person started first the other person started second there is really no difference between them once they establish a connection. Anybody can read into a socket, can write into a socket, the other side will read, this side can write, this side will read, it is a bidirectional communication.

And these processes can be on the same machine or on different machines. So, in this lecture, what we are going to do is we are going to understand these system calls that correspond to communicating using sockets. And in the next lecture, we are going to study how these system calls are actually implemented inside the operating system. So, now, let us see what are the different types of sockets. So, roughly depending on where these two communicating processes are, there are two types of sockets.

(Refer Slide Time: 03:00)



One is, what are called Unix domain sockets are local sockets that are used to communicate between processes on the same machine. And internet sockets, which are used to communicate between processes on different machines. So, in the case of Unix domain sockets or local sockets, one process opens a socket and gives it a certain name, some path name slash something it will give a name.

And another process will also open a socket and it will ask it will make a system call to connect this socket to the first socket by giving this name. So, this is your server socket and

this is the client. And once these two processes are connected, they can exchange information with each other. So, typically processes on the same machine also have to communicate with each other, if you have a large computer system, it is not like all the logic will be there just in one process, the logic will be split across many different processes on a machine.

And all of these will have to exchange information and that is done one of the ways to do that is Unix domain sockets. We are going to study more about this in a later lecture. Then the next thing is if these two processes are on different machines. A process on one machine has to communicate with another process on another machine, then these processes will open Internet sockets, a different type of sockets will be open to communicate across machines.

Now, you can no longer use some name or some path name to address a socket on another machine. How do you know on that machine, what path name is being used and all of that it is very difficult. So, you have a different way of addressing sockets across machines. So, on the internet, every machine has a unique address that is called the IP address. So, next week when we study networking, we are going to study this in a lot more detail.

But for now, understand that an IP address is nothing but a unique address that is given to every machine on the internet. So, now on the server machine, this machine has a certain IP address. And at that IP address on this machine there could be multiple sockets open, multiple processors could be communicating. And each of these sockets will get a port number, a 16-bit unique number that is unique within a machine that is called the port number.

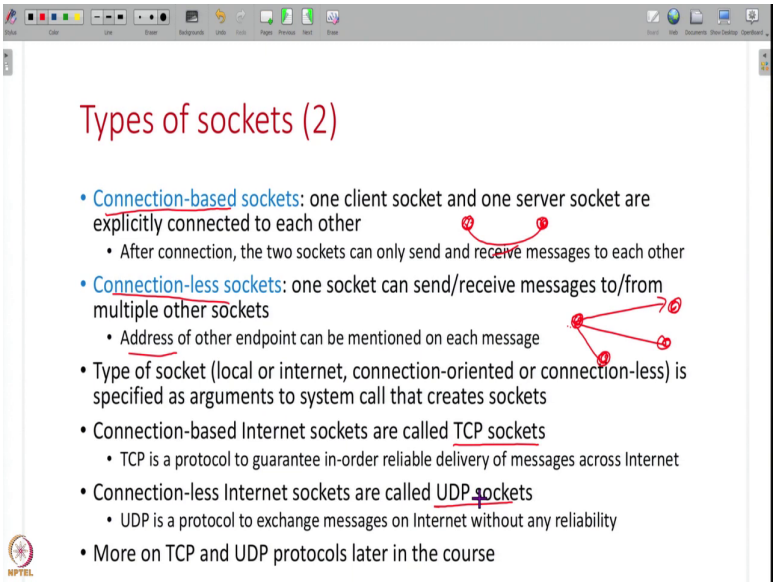
So, when a server opens a socket, this IP address plus this port number is a unique way to identify the server process. And therefore, when a client opens a socket, it will also another client on another machine, there is a process if it knows the IP address and port number of the server socket, then it can connect to that server socket and communicate with it. So, this is how sockets are addressed differently based on whether they are local sockets or internet sockets.

So, note that the client and server they are primarily differentiated by who starts first. The server starts first and it opens it socket at a well-known address and the client process must know that server's address in order to connect to it. So, how does the client know the

server's address, there are many ways to learn the server's addresses that in when we study about the internet in the next week, we are going to know more about this.

But for now, understand that when you want to talk to somebody else, you need to know where they are. if a client socket wants to connect to a server socket, the server socket must be at a well-known address that the client can talk to.

(Refer Slide Time: 06:37)



Next type of sockets that you have, the next classification is based on whether these sockets are connection based or connectionless. What does this mean? In a connection-based sockets two sockets the client and server are connected with each other, are tied to each other are sort of like married to each other. So, that when this socket you write a message into this socket, you can read it only from here, when you write a message here you can only read it from here.

So, after connection, the two sockets can only communicate with each other. The other type of sockets are connection less sockets, that is you have a socket here, this can send a message to this socket or this socket or this socket, it can send messages to different sockets at different points of time. On every message as long as you mentioned the address of the other end point, you can the message will reach the other end point that is this socket is not connected to any one socket, but it can communicate with any of the other sockets.

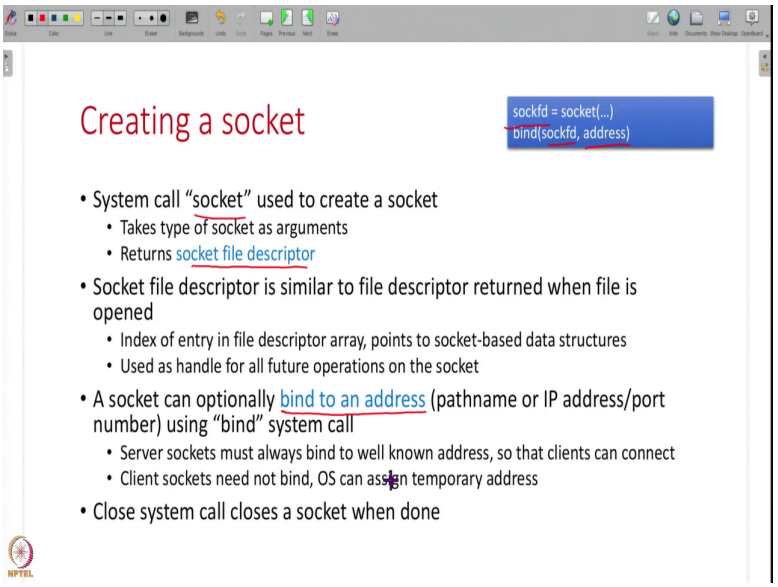
So, these are called connection less sockets. The difference here is that when you send and receive a message here, once the connection is done, you do not have to explicitly say the address on every message whereas here for every message you have to give an address of

which endpoint it is going to. So, now depending on your type of socket, whether it is local or internet or connection oriented or connectionless.

All of these types of sockets are specified as an argument when you create a socket to the system call that actually creates the sockets. The one thing I would like to point out is connection-based sockets are also called TCP sockets and connectionless sockets are also called UDP sockets. So, what are TCP and UDP? They are nothing but protocols that are used on the internet for example, TCP will ensure that on a connection data is being transferred in a reliable manner.

Whereas with UDP since there is no notion of a connection, there is no reliability there are all of these different properties out there for the sockets. That next week when we study how the internet works, we are going to understand what is a TCP and what is a UDP socket in more detail. But for now, you can remember that TCP sockets are connection-based sockets and UDP sockets or connection less sockets.

(Refer Slide Time: 08:41)



So now that we have seen that type of sockets, let us understand how you create sockets. So, there is a system call, the socket system call which is used to create a socket, it takes various arguments like the type of the socket and so on, and it returns a socket file descriptor. So, this is very similar to a file descriptor that is returned when you open a file.

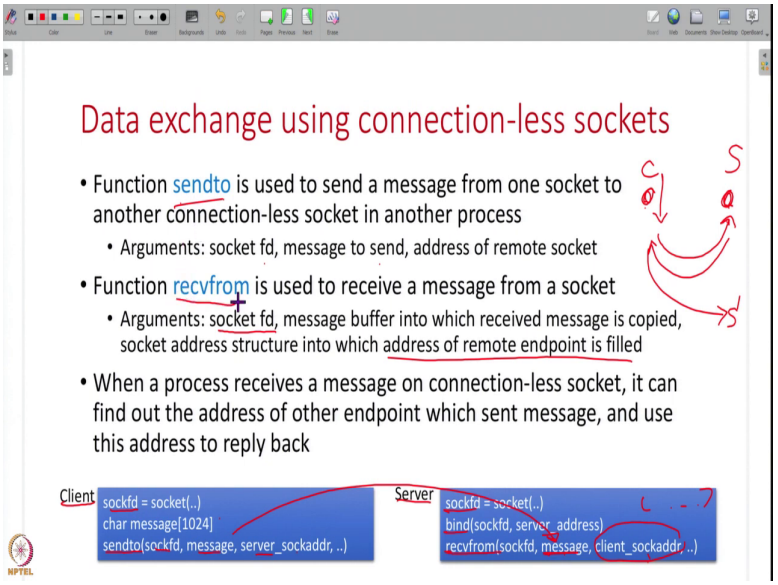
So, the socket file descriptor is also the index in the file descriptor array of a process which points to some socket-based data structures, which are useful for future operations on the socket. Therefore, whenever you open a socket, you get a file descriptor and then any

system call you want to do on the socket you will provide this file descriptor as an argument. So, that is about opening a socket.

And once you open a socket the next thing you can do is you can actually bind it to an address that is you can give a path name or some IP address, port number as arguments and you can bind the socket to an address. So, server sockets must always bind to an address. Why? Because the client must know how to connect to the server socket, the client must know that the server socket is there at a specific address.

Therefore, server sockets must bind to well-known addresses and the client should know about these addresses. But client sockets need not bind. If you just want to be a client then connect to somebody then you do not have to bind, do not have to do the bind system call, the OS will just assign some temporary address to you as needed. So, this is about opening a socket. Similarly, there is also a closed system call just like with files to close a socket.

(Refer Slide Time: 10:23)



So, now that we have created sockets, how do we exchange data across sockets? So, let us begin with understanding how data is exchanged on connectionless sockets. So, you have a client and a server that have both opened a socket using the socket system call and the server has done this bind system call to bind it socket to a specific address a well-known address, and the client knows about it somehow.

Now, the client can use or the server also can use the sendto function to send a message to the other side, now the client knows the server's address. So, into the socket, it will send this message to this address. You have the client socket, you have the server socket, and

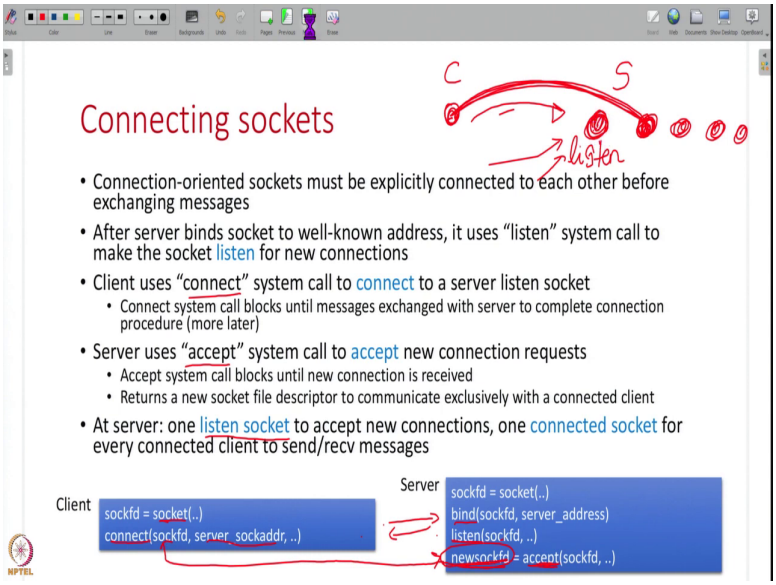
when the client writes a message into the socket gives the server's address the message will reach this server and the server can do receive from and read this message whatever message was sent here, that message will be received here at the server process.

So, the sendto system call is used to send a message from one socket to the other. Note that these are both connection less sockets, why? Because on every message you are providing the address, after sending one message to the server, the next message can go to another server from this client, from the same socket you can send to another server also. Therefore, there is no notion of a connection here between these two sockets.

Therefore, with every message every time you send a message, you will specify which socket, what is the message to send, some user space buffer will be there which has the message and which address to send it to. Similarly, when you receive from a socket, the receive system call will use as arguments a socket file descriptor, the message the buffer into which you want to read this message that is received you will also have some character array or something into which this message has to be copied by the system call.

And you also provide an address structure into which the address of the remote endpoint is filled. Now, when the server receives a message, the kernel, the system call will also tell this server this message came from so and so client at so and so address. So, why does the server need this address because of the server wants to send a reply back to the client then it will use this address of the client and then do a sendto to this address so that the message once again will reach the client. In this way you can exchange two processes on two different machines can exchange data using connectionless sockets, using this sendto and receive from system calls.

(Refer Slide Time: 13:17)



The next thing is what if you want to connect sockets. So, if I want two sockets to only explicitly communicate with each other and to nobody else and on every message, I do not want to put an address then you will have to connect these sockets to each other. So, this connection oriented or TCP sockets must be explicitly connected before they can start exchanging messages. So, how do you do it? Once a server opens a socket and binds it to an address it will use a system call called listen, this listen system called says that this server is open for new connections from clients.

Now, the client cannot directly start sending messages to the server if you are using connection-based sockets, if the socket is a connection-based socket, you cannot just directly send message, you have to first connect it to somebody. So, the client will connect the socket to the server, you will know the server address and you will say connect the socket to this server socket.

And this connect system call, what will it do, it will cause this machine to send some messages will be exchanged back and forth at the end of which this connection is established as socket at the client and a socket at the server are bound to each other. And this connect system call blocks until all of this message exchange happens, and after the connect system call returns now whenever you write anything into the socket, it will only reach this server at the other side it will not go to anybody else.

And similarly, similar to connected the client you have the accept system call at the server, which is this system call when the server says accept this system call will block until a new connection arrives and when a new connection arrives then the client and server can talk to

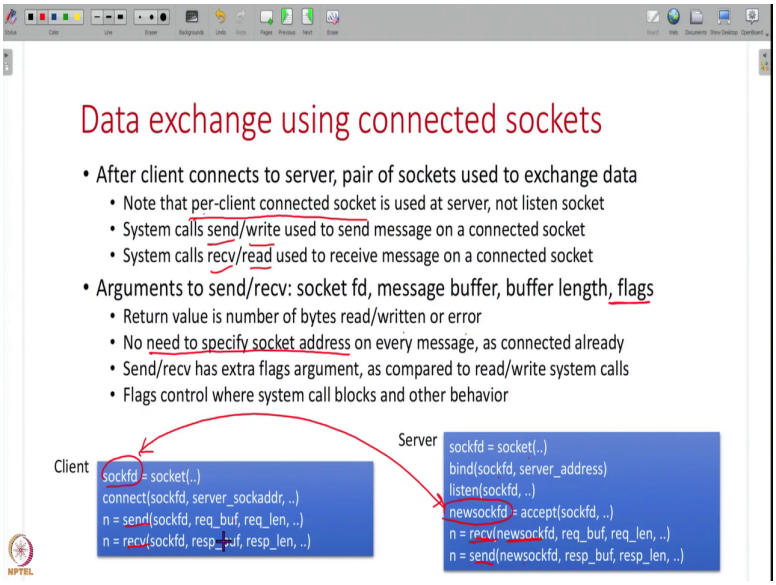
each other. And one very important thing to note here is that this accept system call returns a new socket file descriptor, that is explicitly and exclusively connected to this client. That is your client has one socket and your server has one socket that is listening for new connections.

And when the client sends a message to connect to the server then what happens the server will create a separate socket after it accepts this connection from the client, after it tells the client okay fine, let us talk, it will create a separate connected socket that is bound to this client socket. Why? Because you want to keep this listen socket free for other clients. Now, you cannot say I will only talk to one client at a time, this server may be having to talk to multiple clients and multiple clients may be sending requests to connect to the server.

So, it will keep this listen socket free to listen to requests from all clients. And for every connected client it will assign a separate socket that is exclusively dedicated to talk to this client. So, at the server there will be one listen socket which will continue to accept. Now, this listen socket will continue to accept new connections. On this socket you will only keep on doing accepting and every time there is a new connection this accept will return. And in addition to this there is also a separate connected socket.

Now, this new sock fd you write into this it will reach the client, you write here it will reach. Now, these two sockets are connected to each other. So, there is a listen socket and there is a connected socket, there is one connected socket for every client. So, if you have multiple clients who are connected to you, there will be multiple connected sockets and one listen socket at the server.

(Refer Slide Time: 16:58)



So, how do you do data exchange using connected sockets? The client will open a socket connected to the server, the server will open a socket bind it to a well-known address, do listen and accept the connection. Once the connection is accepted this new sockfd and this socket here these two are directly bound to each other. Now, the client can send a message on the socket, the server can receive it.

Note that it is the connected socket here not the listen socket. The connected socket will receive this message then the server sends a message on this connected socket, the client will receive it. The pairs of sockets are used to exchange data. And note that this is at the server you are reading and writing into the per client socket, on the listen socket you can continue to do accept, you can accept for new connections and get more connected sockets. But this connected socket will be used to send and receive messages.

And the system calls here are send and receive also called read and write, there is very just slight difference between these two. And note that for these system calls, you do not have to provide the address of the other side, you are just sending a message because it is connected you know where it has to go to, there is no need to explicitly specify a address on these sockets whenever you send data.

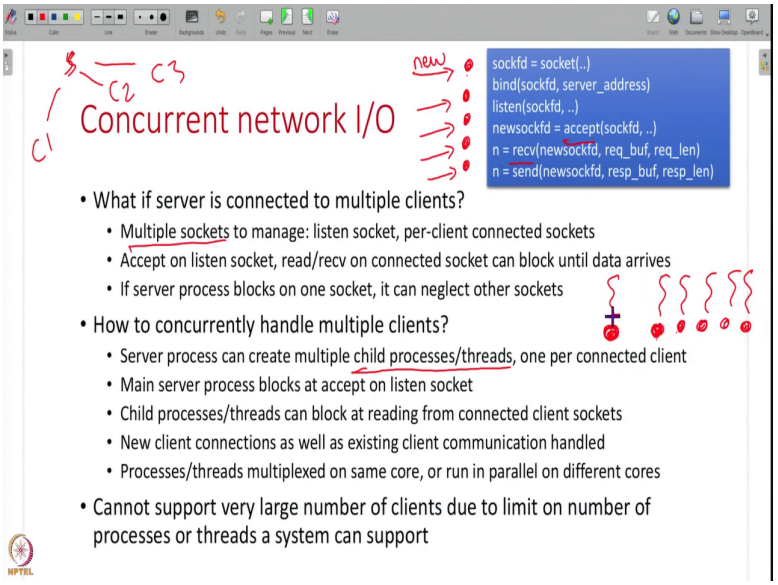
And the arguments are of course a socket file descriptor whatever message you want to send the size of the message and you will read it on the other side, whatever buffer and length you have you write into the socket you will read it here. And the return value will basically say how many bytes have been read or written or if there is some error. And these

flags control behaviour like for example, should I block what if there is no data here, should I block, all of that information is put in over here.

And there is no need to specify a socket address on every message, why? Because the sockets are already connected. So, that is about data exchange using connected sockets, you will first connect, accept a connection, you have separate sockets that are bound to each other which are used to exchange information.

So, now the next question comes up what if a server has to simultaneously talk to multiple clients? So, this is a valid scenario, you have a web server or an e-commerce website. It is not like you are talking to one client at a time. So, now let us look at the issues that happen when a server has to talk to multiple clients.

(Refer Slide Time: 19:28)



So, a server is connected to client c1, client c2, multiple clients at the same time. So, in this case, what is happening at the server you have a listen socket, and you have one connected socket for every client that is currently connected. And the server is listening for new connections on the socket. And on all of these sockets, you might be getting messages you have to read and reply and all of that. So, the server has to manage multiple sockets at the same time.

And note that each of these system calls can block, the accept system call will block until a new connection comes, the receive system call can block until some data comes. So, now how does a server know which socket should I look at? Suppose the server there is a single

process at the server, should I be waiting for new connections? Should I receive data on the socket? Should I receive data on that socket?

And if I keep waiting here, what if I miss messages coming from some other client, if the server pays attention to one guy, the other guys can get neglected. So, then in such cases, what should a server do, you can constantly keep switching between all of these sockets, the server becomes very confused as to what to do. So, therefore, you need more techniques at the server, you cannot just write a simple program like this to handle multiple concurrent clients, you need to do more in order to correctly handle multiple clients.

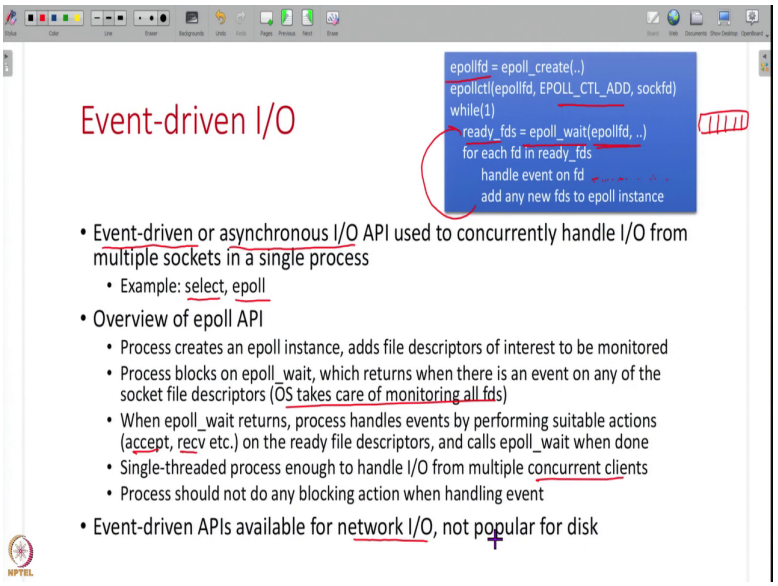
If you just have a single threaded single process server like this, then if you are waiting on one socket, other sockets can get neglected. So, typically what real life computer systems do is, one way to solve this problem is you create multiple child processes or threads. So, there is the main server process or thread and there are multiple child processes or threads created, this main server process will wait will keep monitoring this listen socket, keep listening for new connections.

And these other child processes will keep monitoring their individual connected sockets, every child process or thread is given its own socket. You keep taking care of the socket, read from the socket, process this request and reply to the client, you take care of the socket. And if a new connection comes in, you create a new process or thread and assign it, its connected socket.

In this way, multiple threads or processes is each monitoring its own socket, and it can monitor it effectively without any one socket getting neglected. So, this is called the one process or thread per connection model. This is one way of designing real life web servers or application servers. And the advantage of doing this is now you have multiple processes and threads, even if one of them blocks the other can run, they can run in parallel on multiple cores.

So, you can effectively use your CPU. And new connections as well as existing client connections are all equally handled with the equal amount of attention. But the problem with this method is that you cannot support a large number of clients if you have millions of clients connected, there is a limit on how many processes and threads your system can support. Therefore, for very large number of connections, this model can get somewhat inefficient.

(Refer Slide Time: 22:53)



So, then what are the other options we have? The other option is what is called event driven or a synchronous I/O. This is a different way of managing multiple sockets, where a single process can correctly handle multiple sockets. And this uses a different set of system calls, which are called select or epoll. There are many APIs to do this, let us look at the epoll API here.

So, what this epoll API does is it gives the server the single process a way to give the OS the multiple file descriptors and say that I will not constantly keep looking at each of them, you tell me which of these I have to look at, if a new connection has occurred or new data has arrived, whatever has happened, you tell me what to do, I will look at that. That is a way in which that is the high-level idea of how the epoll API works.

For example, so a server that has multiple sockets to monitor what it will do is it will create what is called an epoll instance. And it will add all the file descriptors to this epoll instance, there is this listen socket, some other socket, all of that information is provided to the operating system using this epoll API. And then the server itself it will not block on either accept or receive or read, it does not know where will it block it does not know. So, therefore, it will call this function epoll wait.

And it will give it all of these file descriptors. And this epoll weight will block until there is an event on any of these file descriptors. So, now, the process does not have to constantly check each file descriptor, the OS is taking care of this, and this OS will make this process

block. And when any event happens on any of these file descriptors, when any of these file descriptors become ready if a new connection has occurred.

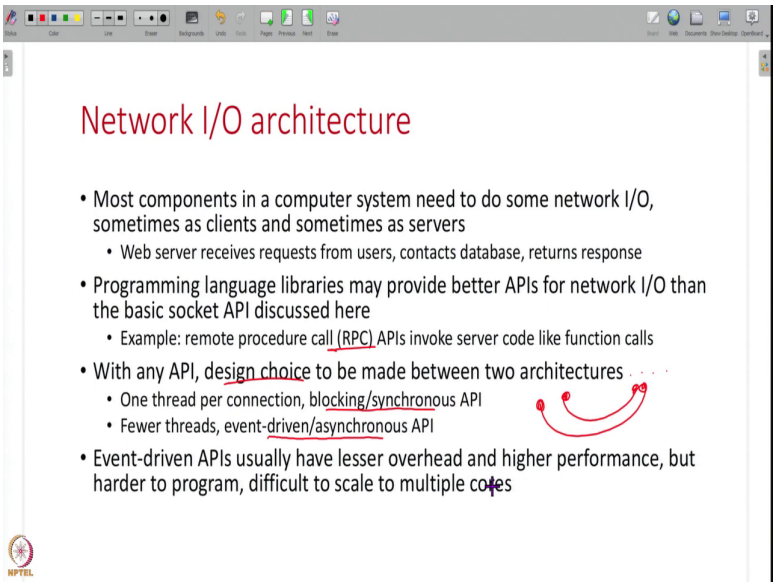
Or if data has arrived on a connected socket, whatever whenever any event occurs, then this epoll wait will return, the OS will tell you which are the file descriptors that are ready that have events and then the process will go handle that event it will accept a connection or read or whatever it has to do and then it will go back into this epoll wait loop again. When epoll wait returns, you will handle the event.

Now, that you know there is an event you will not block, accept will not block because there is a connection, receive need not blocked because the data has already arrived. When things are ready, then epoll wait returns the process the single process at the server can handle all of these events, and it can go back into epoll wait again. In this way, our process is efficiently handling multiple sockets, efficiently handling multiple clients connected clients at the same time, concurrently connected clients.

But one thing to remember is that inside this epoll processing you cannot block, you cannot do any operation that blocks like for example, you cannot do a disk read or something because why, now this is your only process only thread in your process. And you know, the OS is coming back to you only when there are events to be handled.

So, you have to quickly handle the event, go back to check on epoll wait again. So, this is how event driven APIs work for network I/O. This also similar things exist for disk I/O, but they are not very popular with this you still use the blocking-based Read Write APIs. But with network I/O, this event driven APIs are fairly popular.

(Refer Slide Time: 26:23)



So, most components in any computer system need to do some network I/O, as clients or servers, for example, there is a web server, a client connects to the web server, then the web server becomes the client again and connects to some database to get some information then returns a reply. There is a lot of network I/O happening in real systems. And there are many different APIs, many different libraries and different programming languages also available for network I/O.

You need not use simple this socket-based APIs that I have described in this lecture. There are also more complicated, simpler APIs available, which are basically wrappers around this, the OS provides socket, you can add more functionalities over it. For example, there are RPC libraries, where you can invoke the server code like it is a regular function call without actually worrying about the messages, and reading and writing and all of that.

So, we will study these when we study application design later on. But there are many ways for machines to communicate with each other. But with any API, if you look deep enough, you have this design choice. Either you have one thread per connection, and you use a blocking synchronous API, or you use fewer threads, and you use an event driven asynchronous API.

Whichever programming language you are using, whichever framework you are using, you will always have the choice between a client and a server, the server is connected to multiple clients, should the server have a separate thread, or something for each of these

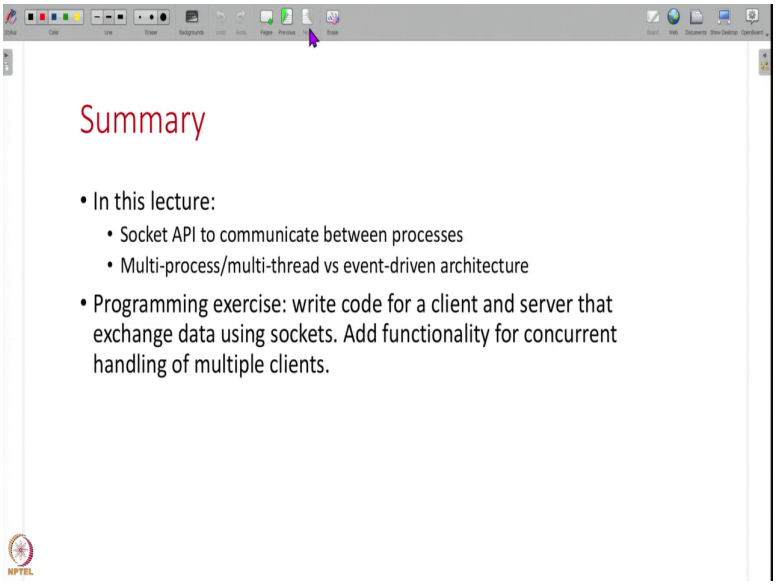
clients and block on them or should it use a fewer number of threads and use an event driven API. This is always a design choice that has to be made.

And what is the trade-off here? Event driven APIs usually have lesser overheads because they have fewer threads to manage, and so on. But they are also somewhat harder to program. Because every time there will be an event, then your application code is actually split across these event call backs. And it can get kind of difficult to program, they are usually non-intuitive.

And they are also difficult to scale to multiple cores. If you have just one process thread that is doing or handling all of these events, then how do you scale it to multiple cores becomes a little bit harder than when you have multiple threads, one per connection. So, the summary is that these are two different ways of doing network I/O.

And any API you use any language, any programming language, any library you use for network I/O, always think about is it giving you a blocking synchronous API or a asynchronous event driven API. And accordingly, you have to think about how to structure your application.

(Refer Slide Time: 29:06)



So, in this lecture, I have given you a summary of the socket API that is used to communicate between two processes, we have seen local, remote sockets, connection less, connection-oriented sockets, we have seen how to do synchronous and asynchronous communication also across the sockets.

As a programming exercise, I will ask you to try to write code for a simple client and server that communicate with each other, then try to extend that code to make your server communicate with multiple clients say using epoll or by creating a separate process or thread for every connection.

So, try to play around with these APIs so that you understand these concepts better. So, thank you all that is all I have for this lecture. In the next lecture, we are going to understand how the OS actually implements these system calls. So, see you all then, thank you.