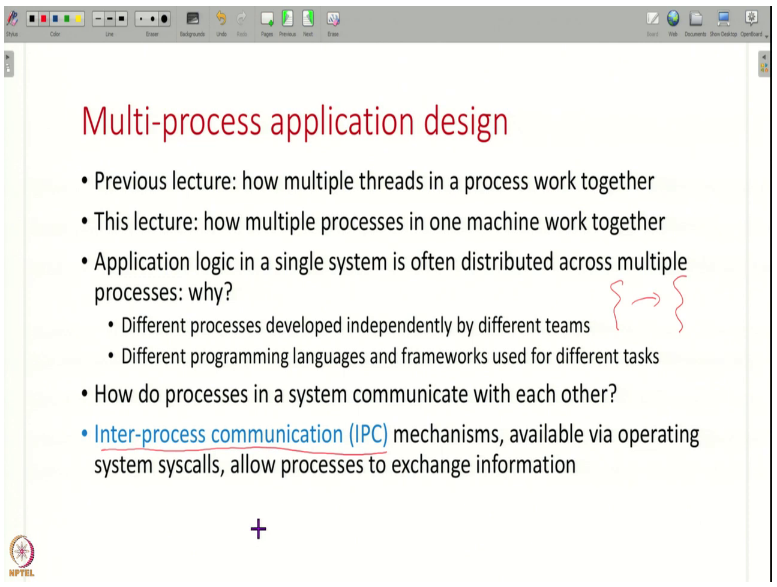
Design and Engineering of Computer Systems
Professor Mythili Vutukuru
Department of Computer Science and Engineering
Indian Institute of Technology, Bombay
Lecture 39
Inter-Process Communication

Hello everyone, welcome to the 27th lecture in the course Design and Engineering of Computer Systems. So, in the previous lecture, we have seen how different threads in a single process can co-ordinate with each other in order to do some work. Now, in this lecture, we are going to study inter-process communication, that is, we are going to see how different processes in a single system can communicate with each other and coordinate with each other.

(Refer Slide Time: 00:47)



Multi-process application design

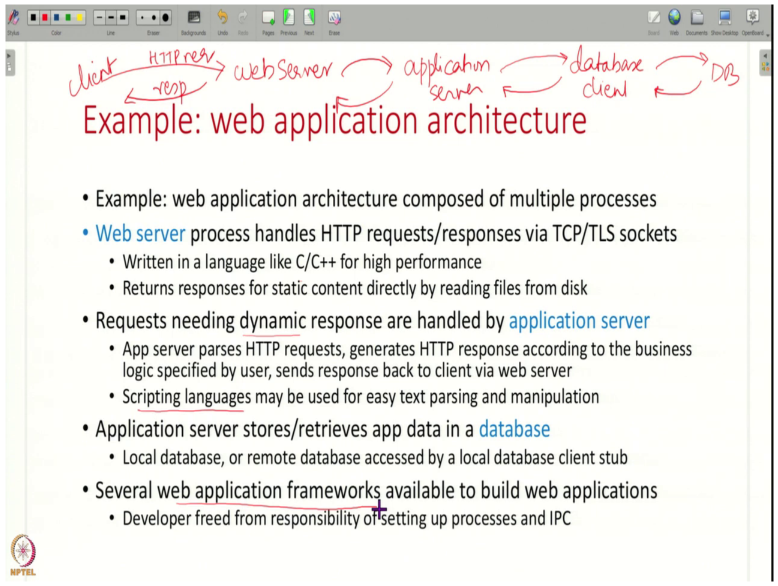
- Previous lecture: how multiple threads in a process work together
- This lecture: how multiple processes in one machine work together
- Application logic in a single system is often distributed across multiple processes: why?
  - Different processes developed independently by different teams
  - Different programming languages and frameworks used for different tasks
- How do processes in a system communicate with each other?
- Inter-process communication (IPC) mechanisms, available via operating system syscalls, allow processes to exchange information

So, the question comes up if you have application logic that has to be done in one machine, why do you need multiple processes? Why cannot you just do all the work in a single process and use multiple threads. So, there are many reasons why we use multiple processes instead of multiple threads.

For example, it could be the case that these different processes were developed independently by different teams or by different companies or so on. You might also want to use different programming languages, different frameworks for different tasks. So, therefore, in a single system, in a single computer, you might have multiple processes that are doing the same work of the same application and have to somehow coordinate with each other one process might have to do some work, then the other process may have to do something else. Some mechanism for coordination may be needed.

So, those mechanisms for coordination and communication between processes are called inter-process communication or IPC mechanisms. And these are available via the operating system or other libraries. So, just like you have mechanisms for different threads in a process to synchronize with each other, like for example, condition variables. Similarly, IPC mechanisms are used for different processes to synchronize with each other.

(Refer Slide Time: 02:06)



Example: web application architecture

- Example: web application architecture composed of multiple processes
- Web server process handles HTTP requests/responses via TCP/TLS sockets
  - Written in a language like C/C++ for high performance
  - Returns responses for static content directly by reading files from disk
- Requests needing dynamic response are handled by application server
  - App server parses HTTP requests, generates HTTP response according to the business logic specified by user, sends response back to client via web server
  - Scripting languages may be used for easy text parsing and manipulation
- Application server stores/retrieves app data in a database
  - Local database, or remote database accessed by a local database client stub
- Several web application frameworks available to build web applications
  - Developer freed from responsibility of setting up processes and IPC

So, let us see an example of where this is needed, where you have multiple processes to do different tasks of an application. So, if you look at a web application, a web server or something that is receiving requests from users and has to handle those requests, then such web applications usually in a single system, they are composed of multiple processes.

For example, you can have a web server process that is actually getting HTTP requests from clients and sending out HTTP responses. So, the main thing in a web application is you will have a web server process. And this process can have multiple threads can be using TCP or if it is a secure server TLS sockets, sending and receiving HTTP request responses, and so on.

And you can have multiple threads, it can be written in a language like C or C++ for high performance. And, of course, if the request is just for a file, the web server can simply read the file and return the response. But if you have dynamic requests, where the user has provided some parameters, and you have to do some work, run some functions and return a response, dynamic response, maybe this web server process may not be able to do all of that work and it might use another process on the same system that is called an application server.

So, for some specific tasks, there is some application server. So, this web server will send these dynamic requests to the application server, the application server will send the responses back, and the web server will then pass them on to the client, you can have such a situation. And this application servers, you can write them in, like scripting languages like Python or something where these scripting languages are not very fast, but they are very good for text parsing and text manipulation, parse an HTTP request, extract various parameters, get the information corresponding to the request, generate a construct a HTTP response, all of that may be easy to do in scripting languages.

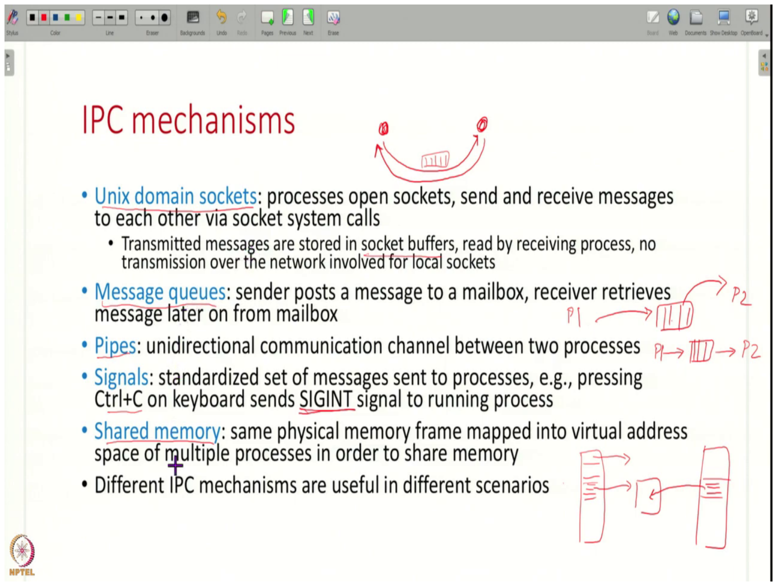
So, you might write your application server process in some other language that is also possible. Now, this application server may want to get or store data in some database. So, this database could be a separate process that you get from some vendor or some open source software, something you get and you install it and this application server has to send request to this database and get responses back.

Again, you need some IPC mechanism here or this you might have a remote database somewhere else and this could just be a database client, you talk to this client, this client will talk to some other database over the network get back responses. All of this also can be possible. So, overall there are multiple components, you cannot build all of this in a single process, you have different databases written by somebody else's, application logic is written by somebody else, web server you get from somewhere else. All of these things have to different processes have to communicate with each other.

So, normally when you use any application frameworks to build web applications, there are many web application frameworks available out there. For example, Python, Django, React, Node JS, you might have heard some of these names, these are all frameworks that make it easy for you to build web applications. So, what these frameworks do is they do all of the setting up multiple processes, setting up the IPC mechanisms between these processes, they do it for you.

So, you may not realize that these things are happening under the hood. But any system, any computer system component that is running on a single machine will most likely have multiple processes for different purposes. And you will need some mechanism to send requests get responses back across these different components, across these different processes in one system.

(Refer Slide Time: 06:27)



IPC mechanisms

- Unix domain sockets: processes open sockets, send and receive messages to each other via socket system calls
  - Transmitted messages are stored in socket buffers, read by receiving process, no transmission over the network involved for local sockets
- Message queues: sender posts a message to a mailbox, receiver retrieves message later on from mailbox
- Pipes: unidirectional communication channel between two processes
- Signals: standardized set of messages sent to processes, e.g., pressing Ctrl+C on keyboard sends SIGINT signal to running process
- Shared memory: same physical memory frame mapped into virtual address space of multiple processes in order to share memory
- Different IPC mechanisms are useful in different scenarios

So, what are these IPC mechanisms that are available there are many, we will briefly see a few popular ones in this lecture. So, one is UNIX domain sockets we have already studied this. We have seen how processes can open sockets, and you read and you send message into a socket, it can be read from the other socket, you send a message here it can be received here. You can connect two different sockets and exchange messages with each other.

We have already seen this in the context of networking and the same concept applies within a system also. So, you can open TCP, UDP sockets to communicate across machines. But on the same machine, you can use UNIX domain sockets, local sockets to communicate between processes on the same machine.

So, of course, for UNIX domain sockets, if you write something into a socket, it will not be put in a packet and sent over the network, it will be stored in some buffer inside the OS and it can be read from the other process from this buffer. So, transmitted messages are stored in socket buffers and read by the receiving process, they are no making packets, adding headers, all of that does not apply here.

So, this is something we have seen before the system calls and all of that. So, there are also other mechanisms for IPC besides UNIX domain sockets, one is what is called message queues. So, this is almost like a mailbox, one process P1 will post some message into some mailbox and another process P2 can contact this mailbox and get this message.

In this way P1 puts messages P2 will read messages from the mailbox that is the concept of message queues. Then you have pipes, pipes is a unidirectional channel of communication that is one process will write into a pipe write some push some bytes into a pipe another process will read from this pipe, that is the concept of pipes.

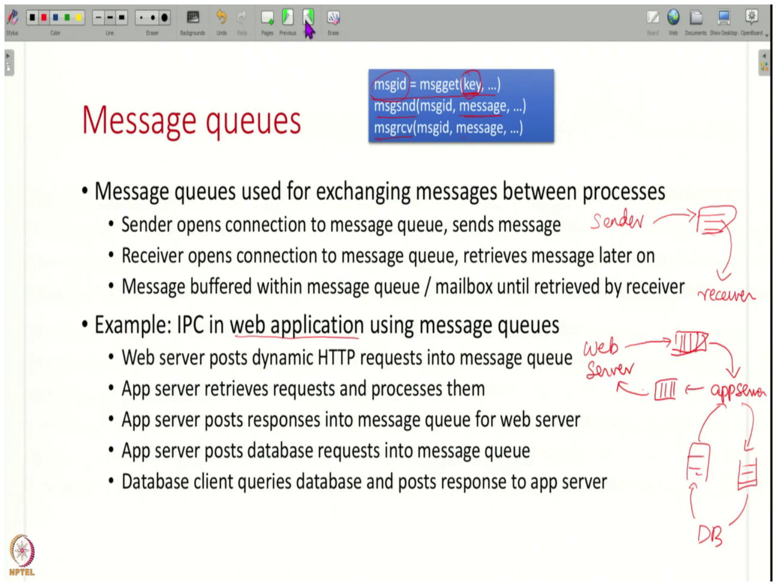
The other concept is of signals, you can send some specific messages standard set of messages to processes. For example, when you press Ctrl C on the keyboard, an interrupt signal called SIGINT is sent to the process. Similarly, there are many different types of signals defined, of course, you cannot send any arbitrary signal there the OS will give you, these other 20, 30 whatever number of signals you can send one of these signal to one of the processes in your system.

Then you have shared memory that is normally a process has some virtual address space and all of these pages are mapping to some physical memory, what you can do is for two different processes, you can map the same physical memory frame into the address spaces of two different processes.

So, that when this process writes to these addresses or this process writes to these addresses, it is written into the same physical memory. So, they can address access the same memory and put some data here and read and write and so on. That is the concept of shared memory. So, these mechanisms are all different. So, sockets message queues, pipes, they are somewhat similar, but these two are very different signals and shared memory and they are useful in different scenarios.

So, what we will do now is we will try to go over these in a little bit more detail sockets, I will not cover because we have studied them enough. But the other mechanisms I will go into a little more detail, so that when you are designing your application that has multiple different processes if you understand each of these mechanisms in some detail you can decide. In this situation this mechanism is suitable, in this situation some other mechanism is suitable and so on. So, let us study this in a little bit more detail.

(Refer Slide Time: 10:12)



Message queues

```
msgid = msgget(key, ...)
msgsnd(msgid, message, ...)
msgrcv(msgid, message, ...)
```

- Message queues used for exchanging messages between processes
  - Sender opens connection to message queue, sends message
  - Receiver opens connection to message queue, retrieves message later on
  - Message buffered within message queue / mailbox until retrieved by receiver
- Example: IPC in web application using message queues
  - Web server posts dynamic HTTP requests into message queue
  - App server retrieves requests and processes them
  - App server posts responses into message queue for web server
  - App server posts database requests into message queue
  - Database client queries database and posts response to app server

So, first message queues. So, message queues are used to exchange messages you can think of it like a mailbox. So, what one process can do is it can open a connection to a message queue you will get back some identifier to talk to a message queue. So, this is again like a file descriptor, you will get back some handle to a message queue, there can be multiple message queues in your system that are all identified by different keys.

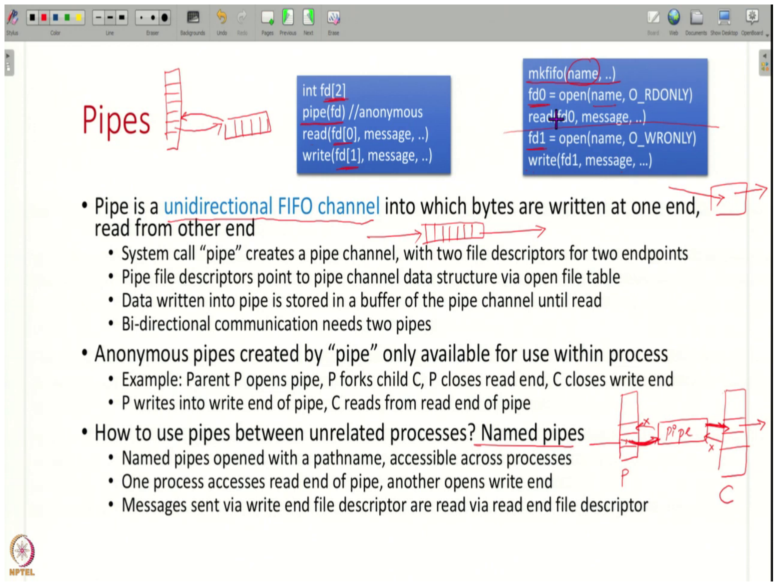
So, the sender will open a message queue with a certain key and it will send a message on that message queue. The receiver can also open the same mailbox and receive the message from that message queue, when you send a message you are giving some set of bytes they will be stored inside the message queue and whoever and whenever some other process does receive those bytes will be passed on to this receiving process.

And in the interim, when the sender sends, the sender sends the message in to the message queue and the receiver will receive from the message queue. In the interim before this receiver process does this receive the message will be stored in the message queue in some memory inside the OS or whichever and whoever is implementing these send receive system calls there the message will be stored temporarily.

And these message queues can be used to do all the IPC in a web application that we have seen before. For example, your web server is getting requests from clients and it will put these HTTP requests into some message queue and the application server will fetch requests from this message queue and handle them and whenever there is a response, it will put that response into probably another message queue and the web server will read these responses and send them back to the client.

Similarly, the application server may post, hey, I want this data it can pose that to a message queue to the database and this database can run this SQL query or whatever and return back the responses and the app server can use this response. So, you can have one or more message queues in the system in this way to communicate to pass messages across all of these different processes in your application, of course, this is one design you can have many different such types of designs.

(Refer Slide Time: 12:46)



Pipes

```
int fd[2]
pipe(fd) //anonymous
read(fd[0], message, ..)
write(fd[1], message, ..)
```

```
mkfifo(name, ..)
fd0 = open(name, O_RDONLY)
read(fd0, message, ..)
fd1 = open(name, O_WRONLY)
write(fd1, message, ...)
```

- Pipe is a unidirectional FIFO channel into which bytes are written at one end, read from other end
  - System call "pipe" creates a pipe channel, with two file descriptors for two endpoints
  - Pipe file descriptors point to pipe channel data structure via open file table
  - Data written into pipe is stored in a buffer of the pipe channel until read
  - Bi-directional communication needs two pipes
- Anonymous pipes created by "pipe" only available for use within process
  - Example: Parent P opens pipe, P forks child C, P closes read end, C closes write end
  - P writes into write end of pipe, C reads from read end of pipe
- How to use pipes between unrelated processes? Named pipes
  - Named pipes opened with a pathname, accessible across processes
  - One process accesses read end of pipe, another opens write end
  - Messages sent via write end file descriptor are read via read end file descriptor

So, the next IPC mechanism is, what is called a pipe? A pipe is a unidirectional FIFO channel, what does it mean? You can put bytes into the pipe and in FIFO order the bytes will be read from the other side. So, how do you create these pipes, these pipes are created with this pipe system call. What this pipe system call will do is it will return two file descriptors. So, if you remember every process has a file descriptor array, which points to the inode or which points to the open file table which will point to the inode or socket buffer or whatever many things can be created and managed using these file descriptors.

Now, when you open a pipe, when you create a pipe, what is happening is two file descriptors are created. One file descriptor is used to write into this pipe buffer and another file descriptor is used to read from this pipe buffer. So, when you create a pipe, the two file descriptor values are filled in and using one file descriptor, you can read messages from this pipe buffer using the other file descriptor, you can write messages into the pipe buffer.

So, you write here, you can read from a file descriptor. So, normally when you read from a file descriptor you are reading from a file or socket or something but here you are actually reading from the pipe buffer. Similarly, when you write also you are storing it in this pipe buffer. And this pipe buffer is of course, point the pointer to this pipe buffer is stored in the open file table.

So, given a file descriptor, you go to open file table you can locate the pipe buffer and you can copy the message there. And this is one pipe is a unidirectional communication you can only write from one file descriptor and read from the other file descriptor. If you want bidirectional you have to open two pipes.

So, now you might be thinking, fine, I have opened a pipe I am sending messages on one file descriptor reading from the other file descriptor in the same process, what sense does it make? Why will a process want to send messages to itself? This is not useful. So, this these are called anonymous pipes that are created like this, within the same process, they are only useful to communicate if a process has other child processes which are all sharing the same file descriptor.

So, for example, you can have a situation like this parent process has opened a pipe. So, there is a pipe buffer and there is a write file descriptor there is a read file descriptor. Now, this parent has forked a child and remember that the child gets a copy of everything of the parent including the file descriptor array.

So, the child processes file descriptors also will have one file descriptor to read from this pipe one file descriptor to write into this pipe. So, the parent and child both have hooks into this pipe buffer. So, now, what you can do is the parent can close say this read end and the child can close this write end so, then what do you have the parent is writing into the pipe here and the child is reading from the pipe here.

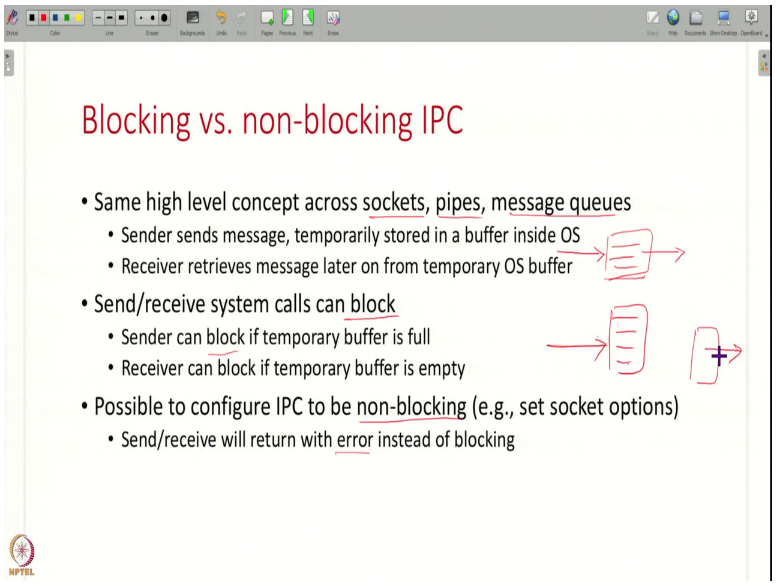
Using this file descriptor, the parent writes into the pipe using this file descriptor the child reads from the pipe. So, in this way, you create a pipe then you fork some child processes and the parent and the child can communicate over these pipes. Note that these anonymous pipes are only available within a process and its descendants.

Because there this pipe you cannot refer to it by any name there is no identifier some other process cannot use this pipe. But if you want to communicate between unrelated processes, then you can use named pipes. So, here is an example of a named pipe you create a named pipe with a certain name, this can be a path name, some string. Then different processes can open if you have a name pipe different processes can open a write file descriptor and a read file descriptor into the same pipe.

So, one process can say into this pipe, give me a read file descriptor using this file descriptor this process can read from this pipe. Another process can say give me a write end of this pipe, it can get a file descriptor to write into the file. So, when one process. So, these two pieces of code can be in different processes, when one process writes into a pipe, the other process can read from the pipe.

So, if you want to communicate between unrelated processes, that is not parent child such kind of relationships do not exist some random two different processes in a system then you can use named pipes where one process where both these processes open the same pipe with the same name and one of them writes into the pipe the other will read from the pipe. And if you want bidirectional communication, you will need two pipes. Because a pipe by itself is only unidirectional channel.

(Refer Slide Time: 18:06)



Blocking vs. non-blocking IPC

- Same high level concept across sockets, pipes, message queues
  - Sender sends message, temporarily stored in a buffer inside OS
  - Receiver retrieves message later on from temporary OS buffer
- Send/receive system calls can block
  - Sender can block if temporary buffer is full
  - Receiver can block if temporary buffer is empty
- Possible to configure IPC to be non-blocking (e.g., set socket options)
  - Send/receive will return with error instead of blocking

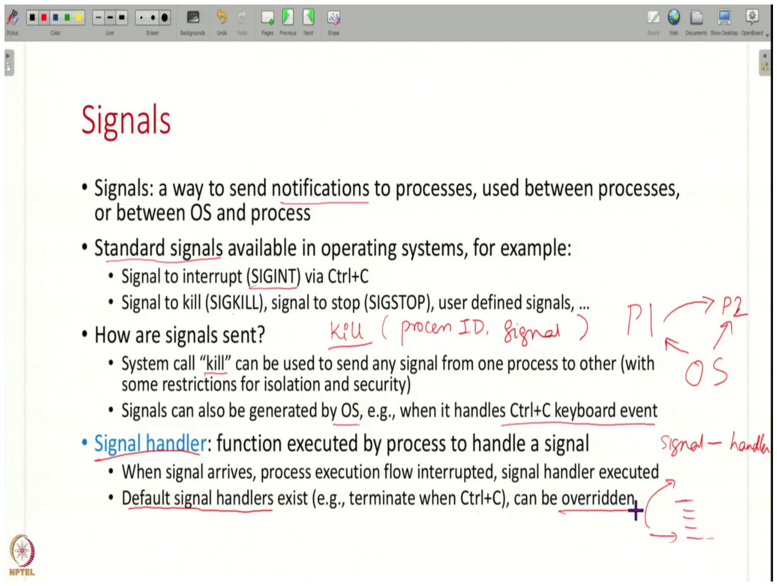
So, now, across all of these whether it is sockets, pipes, message queues across all of these the concept is the same, your one processes sending messages, this message is temporarily stored inside some buffer inside the OS, it could be the pipe buffer, the socket buffer the message queue, whatever there is some memory location inside the OS where whenever a process writes a message it is stored here and another process can read from this buffer later on using some system calls.

So, this is the basic concept sockets, pipes, message queues, the syntax might be different, but how they work is at a high level the same, with of course minor differences like sockets are bidirectional, whereas pipes are unidirectional, message queues are somewhat asynchronous, you can post a message and some of the processes can read the message later on.

So, they are slight differences, but the high-level concept is the same. Now, for all of these system calls that read or write into these sockets or pipes or message queues, you can have two variants of those system calls, the system calls can be blocking or non-blocking. For example, if you are writing into a pipe, but this maximum buffer size is filled this buffer is full.

The reading end has not yet read all the data in the pipe, in that case if your buffer is full and you write into a pipe or a socket, the write or the sending can block. Similarly, if your buffer does not have any content and you read this read you can also block. That is one option or you can configure all of these system calls to be non-blocking where you say if I write into one of these pipes or sockets, but the buffer is full, then you can return an error instead of blocking. Similarly, when reading also you can say 0 bytes are available and return some value without actually reading any data. So, it is possible to do this IPC either in a blocking fashion or a non-blocking fashion, it depends on the preferences of the programmer in his or her coat.

(Refer Slide Time: 20:09)



Signals

- Signals: a way to send notifications to processes, used between processes, or between OS and process
- Standard signals available in operating systems, for example:
  - Signal to interrupt (SIGINT) via Ctrl+C
  - Signal to kill (SIGKILL), signal to stop (SIGSTOP), user defined signals, ...
- How are signals sent? kill (process ID, signal)
  - System call "kill" can be used to send any signal from one process to other (with some restrictions for isolation and security)
  - Signals can also be generated by OS, e.g., when it handles Ctrl+C keyboard event
- Signal handler: function executed by process to handle a signal
  - When signal arrives, process execution flow interrupted, signal handler executed
  - Default signal handlers exist (e.g., terminate when Ctrl+C), can be overridden

So, the next IPC mechanism we are going to study is signals which is significantly different from the sockets message queues, pipes we have seen so far. Signal is a way to send some specific notifications or messages to processes. So, you cannot send arbitrary data like in a pipe you can write whatever bytes you want, but in a signal you cannot do that you cannot send arbitrary data, but there are some specified set of standard signals that are available that are defined in an operating system and you can only send these signals between processes and the signals can be sent between two processes or the OS and a process also.

So, what are the signals? So, whenever you want to interrupt a process you can send up SIGINT. For example, when you hit control C the user is sending SIGINT to some process or processes can also send SIGINT to each other. Then there are other signals, signal to KILL signal to STOP a process there can be user can define some signals.

So, there are many different signals in an operating system, that but with signals you can only send these specific messages you cannot send a random array of bytes. You can only send some standard predefined signals. And, how do you send a signal? You can use the kill system call.

The kill system call can be used to send a signal you can say I want to send to this process to some process ID, I want to send this particular signal you can specify that and you can send that using the kill system call.

So, one process P1 can send some signal to another process P2 two using the kill system call. The system call is kind of unfortunately named, it is not just to send this KILL signal, but any signal, not dangerous signals, but regular signals also can be sent. And of course, there are certain restrictions on any random process cannot kill any other random process, there are certain restrictions in place. But most signals can be sent from one process to the other.

Signals can also be generated by operating systems, when you hit Ctrl C, whichever processes on the screen, the OS, whenever it handles this keyboard event, this interrupt handler, it will send the SIGINT, the Ctrl C signal to whichever processes on your screen at that point of time. So, signals can be sent between processes, they can also be sent by OS to processes.

And what happens when a process receives a signal? So, every process has a set of functions defined, which are called signal handlers that say, when I get the signal, I will do this work, when I get the signal for each signal, there will be a handler, some functions defined. So, you may be wondering, I have never written the signal handlers what is happening here. So, it turns out that there are many default signal handlers that already exist.

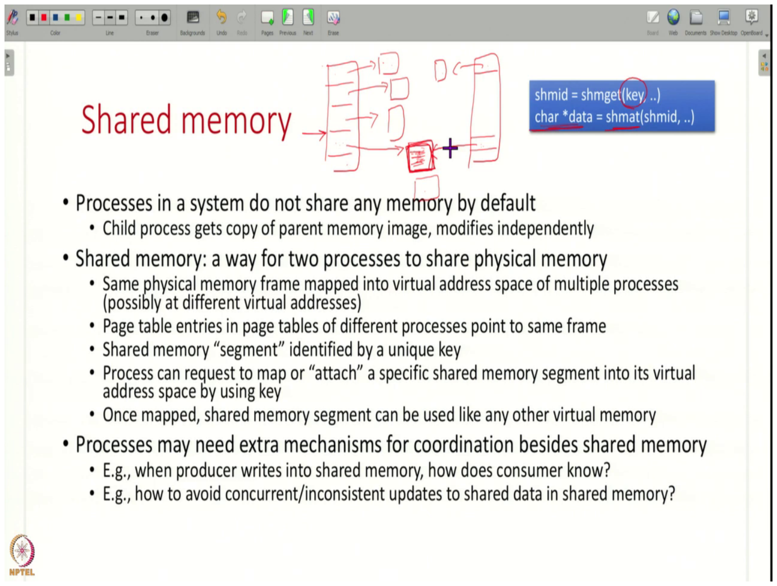
For every process, when you create a process in the template of the process there are certain default signal handlers defined for each process that when this signal happens do this, when the signal happens do this. When Ctrl C comes terminate such signal handlers are already defined. So, you as a programmer, you do not have to define every single signal handler, most of them are already defined for you.

And what happens when a signal arrives, the process, the program counter will jump from whatever instruction it has to run next to the signal handler. The process is executing some code. From here, it will jump to this signal handler, in order to handle a signal, and then it will come back here, if possible. So, that is how signal handlers work and default handlers exist.

For example, there is code to terminate a process when you hit Ctrl C. But you can also override the signal handlers. A process can say, when I get Ctrl C, I will do something else I will not terminate, maybe I will print the message and terminate or I will do something else. So, you can override these signal handlers.

In this way when a process P1 sends a message to process P2, P2 can do some extra work. Because it has defined what the work is in its signal handler. of course, you cannot override all signals. There are some signals which you cannot override, which you have to execute like, somebody OS wants to kill you cannot say, oh, no, I am not going to do that. So, some signals you cannot override, but for some other signals, you can override the default behaviour and specify your own signal handler so that the process can do some different work from the default work that is done when it receives a signal.

(Refer Slide Time: 25:07)



Shared memory

```
shmid = shmget(key, ..)
char *data = shmat(shmid, ..)
```

- Processes in a system do not share any memory by default
  - Child process gets copy of parent memory image, modifies independently
- Shared memory: a way for two processes to share physical memory
  - Same physical memory frame mapped into virtual address space of multiple processes (possibly at different virtual addresses)
  - Page table entries in page tables of different processes point to same frame
  - Shared memory "segment" identified by a unique key
  - Process can request to map or "attach" a specific shared memory segment into its virtual address space by using key
  - Once mapped, shared memory segment can be used like any other virtual memory
- Processes may need extra mechanisms for coordination besides shared memory
  - E.g., when producer writes into shared memory, how does consumer know?
  - E.g., how to avoid concurrent/inconsistent updates to shared data in shared memory?

So, the next IPC mechanism is shared memory. So, we have seen the basic idea before. So, what is shared memory is every process it has some pages and the page table maps these pages into some physical frames. We have seen this before many weeks back and two different processes in general have a different set of frames that map to the same virtual addresses.

So, virtual address 0 can map to some physical address to this frame this page maps to this frame for this process, then virtual address 0 the first page maps to this frame for some other process that is possible and in general, they share different frames except for things like OS where they share the different pages point to the same physical frames. But what you can do is you can explicitly say that two different processes the virtual addresses will point to the same physical memory frame, you can do that using shared memory.

So, the OS can create a shared memory segment a special physical frame whose physical address is present in the page tables of two different processes. So, you can add page table entries where this page number points to the same frame number for two different processes of course, the virtual addresses can be different, but the physical address will be a same. So, how do you identify it, you can create a shared memory segment using a key.

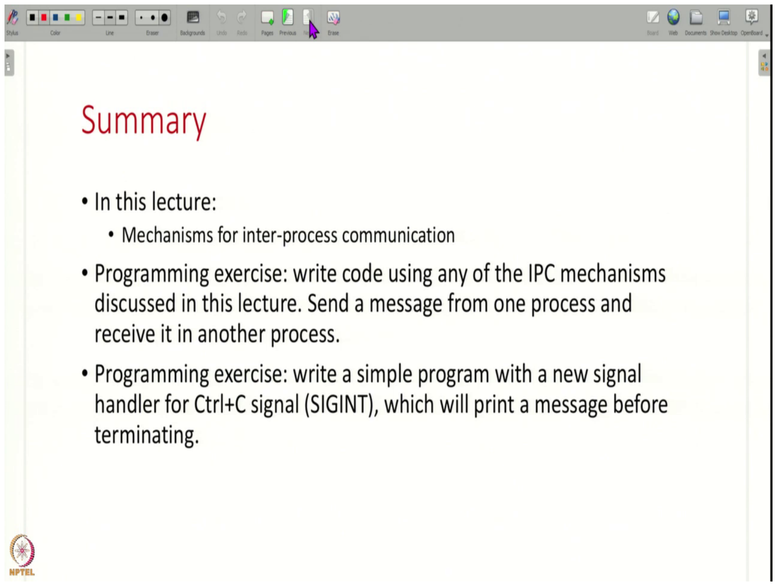
So, there can be many such shared memory segments, each of which has a name a key and when a process says, attach this shared memory segment into my virtual address space, the OS will map this shared memory segment into a certain virtual address and return that virtual address pointer some char* pointer, it will return to you.

So, now two different processes can attach the same shared memory segment using the same key and map it into their virtual memories. And now two different processes can just read or write, using this address you can put some data here this process can read from here and vice versa. Once the same memory shared, you can exchange data easily using this shared memory segment, of course, note that the shared memory by itself may not be always useful in all scenarios.

For example, if you want to do a producer consumer with the shared memory, the producer will put some data into the shared memory, how will the consumer know normally, the consumer has to read this memory and see has it been updated has it been updated. Similarly, if both processes are writing to the same data structure, you can have inconsistent updates.

So, with shared memory, there are all of these concurrency race conditions you have to take care of. But shared memory along with other mechanisms can be used. For example, if you want to do producer consumer with shared memory producer process puts the request here and maybe sends a signal to the consumer process, then the consumer process knows the data has arrived, it will read that shared memory data. So, in this way, so you can use some combination of all of these IPC mechanisms to do the coordination between processes.

(Refer Slide Time: 28:31)



Summary

- In this lecture:
  - Mechanisms for inter-process communication
- Programming exercise: write code using any of the IPC mechanisms discussed in this lecture. Send a message from one process and receive it in another process.
- Programming exercise: write a simple program with a new signal handler for Ctrl+C signal (SIGINT), which will print a message before terminating.

So, in this lecture, we have studied many different IPC mechanisms, and of course, there are more also that are available. So, as a programming exercise, try to write some code using any of these IPC mechanisms probably you can do like a producer consumer that we did with threads, you can do that with processes. Where one processor is producing some messages another processes, reading that data, you can use message queues or shared memory any of these mechanisms.

And understand the differences between these mechanisms, which mechanism is useful in which situation. And another programming exercise you can do is you can write signal handlers in your program. By default, when you get Ctrl C the program terminates, but you can write a new signal handler that will print a message instead of terminating. So, such things also you can understand how signals work by writing simple programs like this. So, that is all I have for this lecture. Let us continue our discussion in the next lecture. Thank you.