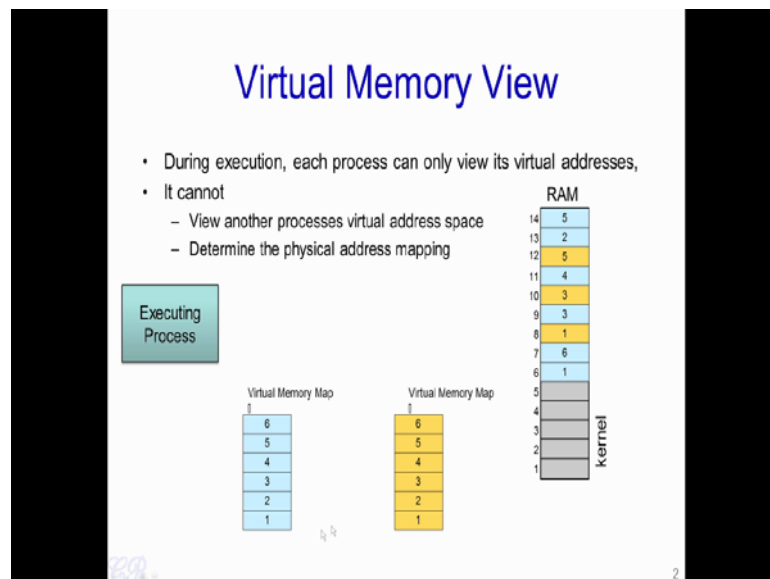


**Introduction to Operating Systems**  
**Prof. Chester Rebeiro**  
**Department of Computer Science and Engineering**  
**Indian Institute of Technology, Madras**

**Week – 06**  
**Lecture – 23**  
**Inter Process Communication**

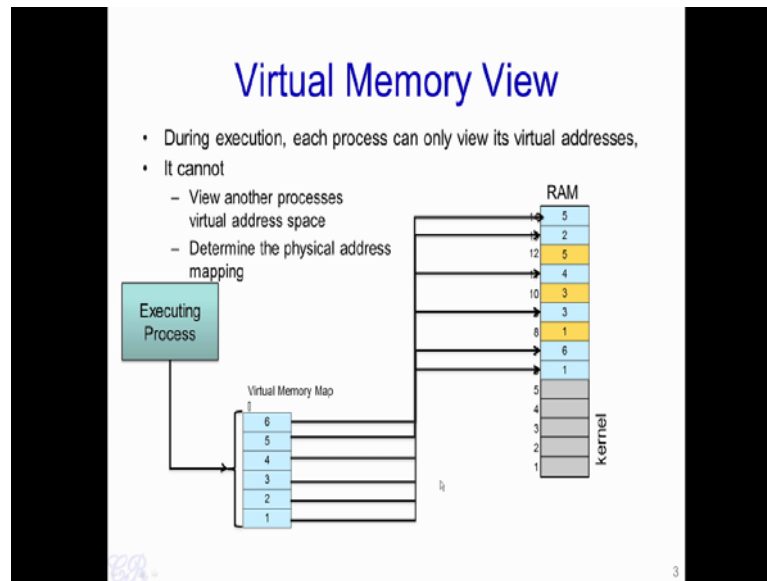
In this video we will look at Inter Process Communication. Essentially, when we write large applications it is often quite useful to write them as separate processes. In order to have an efficient communication between these processes, in order to make things happen efficiently we use inter process communication. In this particular video, we will look at a brief introduction to IPC's that is Inter Process Communication and we will see a few examples of the same.

(Refer Slide Time: 00:50)



So we have seen this Virtual Memory View of a process. We said when a process is executing it sends out virtual addresses which get mapped into this virtual memory map. And we had seen that is a MMU which uses page table stored in the memory and converts these virtual addresses into physical addresses. Now we have also seen that each process that executes in the CPU will have it is own virtual memory map.

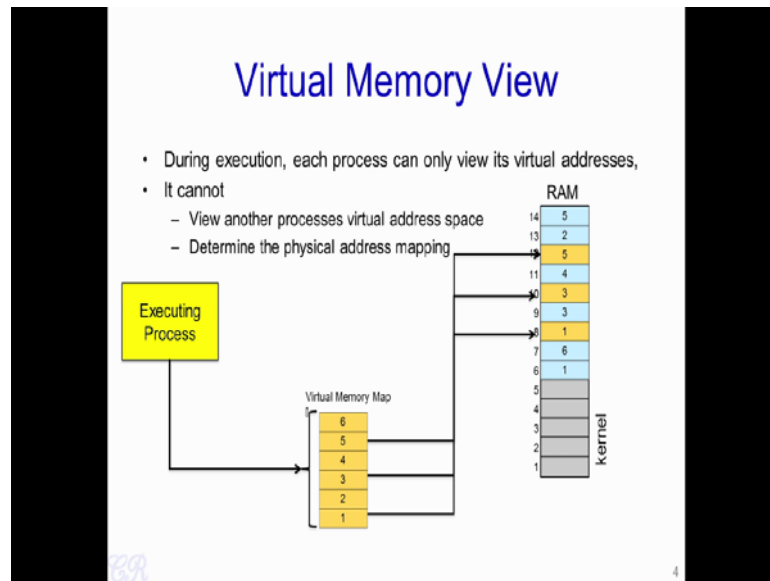
(Refer Slide Time: 01:23)



The use of virtual memory map provides some level of abstraction. Essentially, the executing process would only know it is virtual addresses and with that virtual address, it can only access the user space of the virtual memory map. We had seen that the MMU then transfers this mapping into corresponding physical addresses. So, the executing process will have no way to determine what the corresponding physical addresses for its memory accesses are. For instance, if you declare an integer `i` in your program you can print out the virtual address corresponding to `i` for example, with a `printf` ampersand `i`. However, it is not possible for you in the user space to determine what the corresponding physical address for that particular variable `i`.

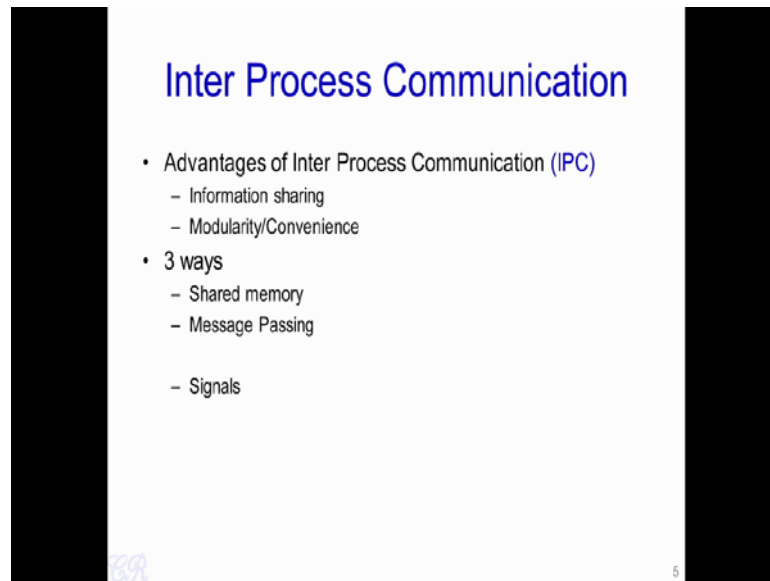
The second thing what the virtual memory map provides is that the executing process it only have accessed to it is own virtual memory map. If there is another process it has with it is own table, now in user space there is no way for the executing process to determine anything of the other process.

(Refer Slide Time: 02:42)



So, when there is a context switch and a new process executes it is the second processes virtual address map or virtual memory map which is then used. This process has no way to determine any other processes virtual address map. So the thing to conclude from these slides is that, a process it has no way to determine any information or any data about another process. So given this, how does one process communicate with another process?

(Refer Slide Time: 03:15)

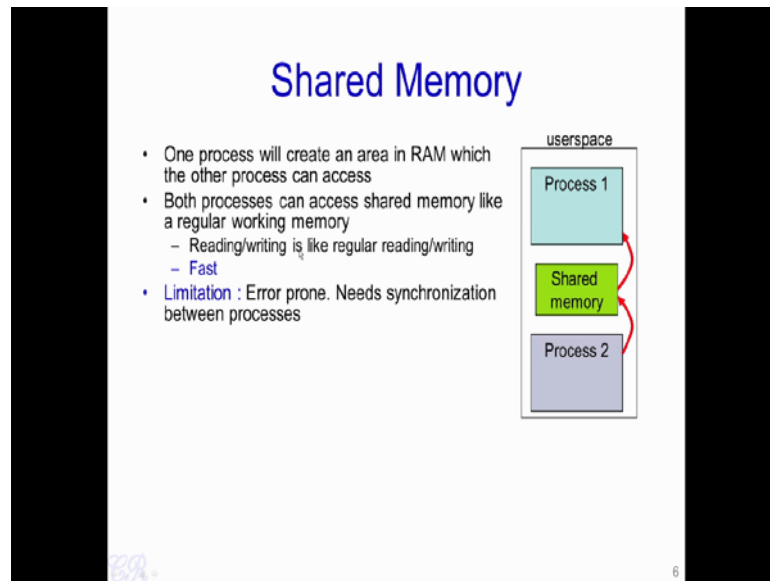


In order to do this, there is a mechanism known as Inter Process Communication. Essentially with IPC's or inter process communication two processes will be able to send and receive data between themselves. The advantage of IPC is that the processes could be return to be modular. So essentially each process is meant to do a single job and processes could then communicate with each other through IPC's.

For an example, let us say we have a data acquisition system and a control system. Essentially we could write one process which acquires data from the external world, such as the temperature, pressure, or the speed, and then it could then send this information or the data collected to a second process which then analyzes the data and determines some particular parameters. These parameters could be sent to a third process which then actuates some external data, for instance it could open a valve or close a valve or adjust the temperature of the room and so on. Thus, we see we are able to achieve a modular structure in our application.

Each processes job is to only focus on a single aspect. The communication between the processes is achieved by inter process communication. In typical operating systems there are three common ways in which IPC's are implemented. These are through shared memory, message passing, and signals. So let us look at each of these things.

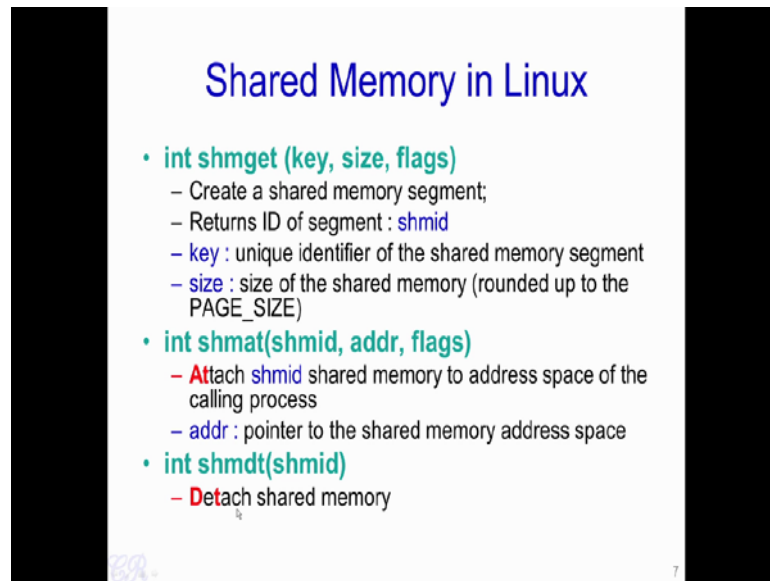
(Refer Slide Time: 05:00)



So, with a shared memory we have one process which creates an area in RAM which is then used by another process. Essentially, the communication between process 1 and process 2 is happening through this particular shared memory. In both processes can access the shared memory like a regular working memory, so they could either read or write to this particular shared memory independent of each other. The advantage with this is that the communication is extremely fast, there are no system calls which are involved. And, the only requirement is that you could define an array over here and then fill the array in the shared memory which can then be read by the other process.

The limitation of the shared memory approach is that it is highly prone to error; it requires the two processes to be synchronized. So, we will take a small example of how shared memory is implemented in Linux.

(Refer Slide Time: 06:05)



## Shared Memory in Linux

- **int shmget (key, size, flags)**
  - Create a shared memory segment;
  - Returns ID of segment : **shmid**
  - **key** : unique identifier of the shared memory segment
  - **size** : size of the shared memory (rounded up to the PAGE\_SIZE)
- **int shmat(shmid, addr, flags)**
  - **Attach** **shmid** shared memory to address space of the calling process
  - **addr** : pointer to the shared memory address space
- **int shmdt(shmid)**
  - **Detach** shared memory

Essentially, in shared memory in the Linux operating system we have three system calls which are used first is the shmget or the shared memory get which takes three parameters; a key, size, and flags. This system call creates a shared memory segment. It returns an ID of the segment that is shmid the shared memory ID of the segment. The parameters key is a unique identifier for the segment, while size is the size of the shared memory. This is typically rounded up to the page size that is 4 kilobytes. So, this is how a shared memory gets created in a particular process.

Now another process could attach to this shared memory by this particular system call that is shared memory attach, shmat. This particular call requires the shared memory ID, and address, and flags. So, essentially system call would attach the shared memory to the address space of the calling process. The address is the pointer to the shared memory address space. We will understand more of this through an example. The opposite of the shared memory attach is the shared memory detach where a process could detach the shared memory from it is user space. So, the detach system call takes the shared memory ID.

(Refer Slide Time: 07:40)

### Example

```
server.c
1 #include <sys/types.h>
2 #include <sys/ipc.h>
3 #include <sys/shm.h>
4 #include <stdio.h>
5 #include <unistd.h>
6
7 #define SHM_SIZE 27 /* size of shared memory */
8
9 int main()
10 {
11     char c;
12     int shmfd;
13     key_t key;
14     char *shm;
15
16     key = 5678; /* some key to uniquely identify the shared memory */
17
18     /* create the segment */
19     if ((shmfd = shmget(key, SHM_SIZE, IPC_CREAT | 0666)) < 0) {
20         perror("shmget");
21         exit(1);
22     }
23
24     /* attach the segment to our data space */
25     if ((shm = shmat(shmfd, NULL, 0)) == (char *) -1) {
26         perror("shmat");
27         exit(1);
28     }
29
30     /* Now put some things into the shared memory */
31     int i = 0;
32     for (i = 'A'; i <= 'Z'; i++)
33         *shm++ = i;
34     *shm = '\0'; /* end with a NULL termination */
35
36     /* wait until the other process changes the first character
37      * to 'a' in the shared memory */
38     while (*shm != 'a')
39         sleep(1);
40     exit(0);
41 }
```

```
client.c
1 #include <sys/types.h>
2 #include <sys/ipc.h>
3 #include <sys/shm.h>
4 #include <stdio.h>
5 #include <unistd.h>
6
7 #define SHM_SIZE 27
8
9 int main()
10 {
11     int shmfd;
12     key_t key;
13     char *shm;
14
15     /* we need to get the segment name "5678", created by the server
16     key = 5678;
17
18     /* create the segment */
19     if ((shmfd = shmget(key, SHM_SIZE, 0666)) < 0) {
20         perror("shmget");
21         exit(1);
22     }
23
24     /* attach the segment to our data space */
25     if ((shm = shmat(shmfd, NULL, 0)) == (char *) -1) {
26         perror("shmat");
27         exit(1);
28     }
29
30     /* read what the server put in the memory */
31     for (i = 0; i <= 26; i++)
32         putchar(*shm);
33
34     /* Finally, change the first character of the
35     * segment to 'a', indicating we have read
36     * the segment.
37     */
38     *shm = 'a';
39     exit(0);
40 }
```

Let us see an example of shared memory with this example. So, let us say we have written this particular program called server dot C and we create a shared memory in this. We define a key called 5 6 7 8 this is some arbitrary value for key, but the requirement is that this key should uniquely identify the shared memory. We could use this key to invoke this function shmget, we pass it the key, we pass it shm size that is shm size is defined here as the size of the shared memory which we want to create. Note that; this size although we specified as only 27 bytes we will get extended to a page that is we will create a page of 4 kilobytes corresponding to this.

The third parameter is the permissions, that we have given it as IPC create that is we we have creating this particular shared memory and these are the permissions, read, write, execute permissions. Of course, if this function fails then it enters over here and exits. Otherwise, if it executes successfully we get a valid shared memory ID.

The next part is to invoke the shared memory attach, shmat providing this particular ID and we will get a pointer to this particular shared memory. So, shm so we have like char star shm this is a pointer to the shared memory. Now the shared memory attach would return a pointer to this particular shared memory.

Now in case it returns minus 1 then it is due to an error and we are exiting. Now at this particular point we have obtained a pointer to the shared memory and we can use that particular pointer just as a standard pointer as we use in a C program. For instance, over here we have moved the pointer to this variable S which is defined as char star S and we have put alphabets from - a to z in this thing. And finally, we have ended it with null termination. Then we are going in an in a loop continuously sleeping, so we will come back to these two statements later on.

Now, let us look at the client side of this code. We invoke this shmget and give with a key, that is key is the same 5 6 7 8, now the shared memory size is as before 27 and we are providing the permissions. Note that we do not require to give create over here because the shared memory region is already created. Then we attach to this particular shared memory segment as before we invoke the function shared memory attach, shmat and we pass it the shared memory ID. Then we get a pointer shm which is a pointer to this shared memory location.

So, essentially both the server as well as the client are pointing to the same physical memory page. Now we could actually read data from this particular shared memory region. So remember that the server has put values from a to z that is a b c d to z, while in this case we could read the values a to z into this particular pointer S and print it out in this scheme. Once we have completed reading all data from the shared memory we put this star into the shared memory location, it is the first location in the shared memory.

Now recall that over here in the server we have put a while loop which continuously loops until the first parameter pointed to in the shared memory is a star. So when we obtain a star it means that the client has completed reading all data and therefore the server can exit. In a similar way the client can exit as well. So, you see that we have these two processes; one is server, one a client.

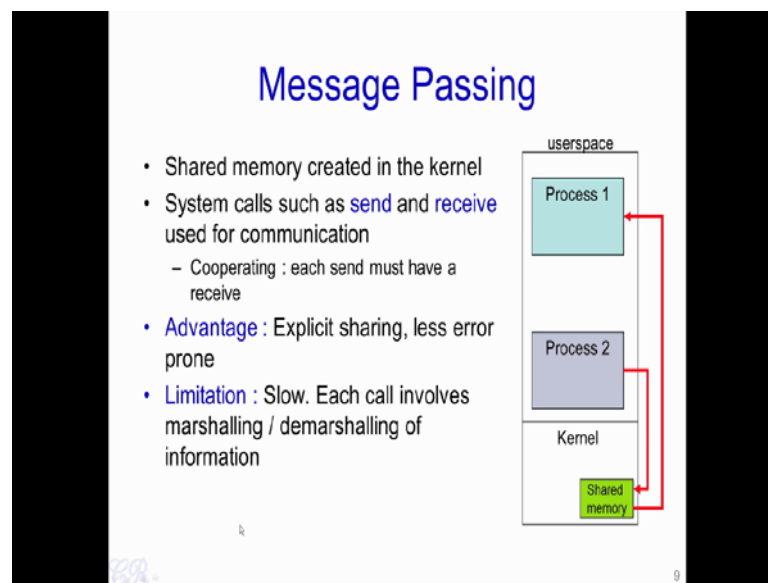
And these two processes are having their own virtual address space, but by the use of the shared memory which is created by the shared memory get and shared memory attach, we have able to a create shared memory region which is common between the server and the client. Then we have obtained a pointer to that particular shared memory region that



is S on shm and the server could put data into the shared memory region while the client could read data, the vice verse is also possible.

At the end of it we require to note that there is a synchronization required between the server and client. This synchronization requirement should be explicitly program into this particular server client module.

(Refer Slide Time: 12:54)

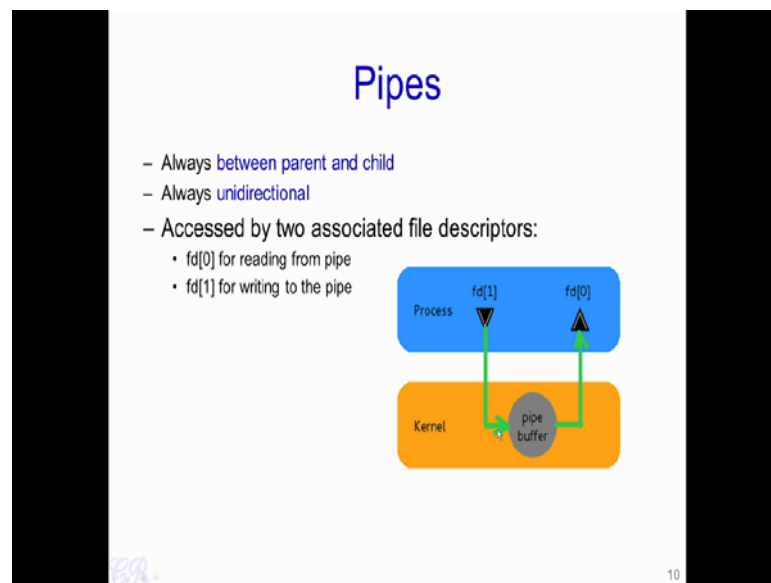


Next, we will look at the message passing. Unlike the shared memory that we just seen where the shared memory is created as part of the user space in message passing the shared memory is created in the kernel. Essentially we would then require system calls such as send and receive in order to communicate between the two processes. If a process 2 wants to send data to process 1 it will invoke the send system call. So this would then cause the kernel to execute and it will result in the data return into the shared memory, While, when process 1 invokes receive data from the shared memory would be read by process 1.

The advantage of this particular message passing is that the sharing is explicit. Essentially both process 1 and process 2 would require support for the kernel to transfer data between each other. The limitations is that it is slow, each call for the send or

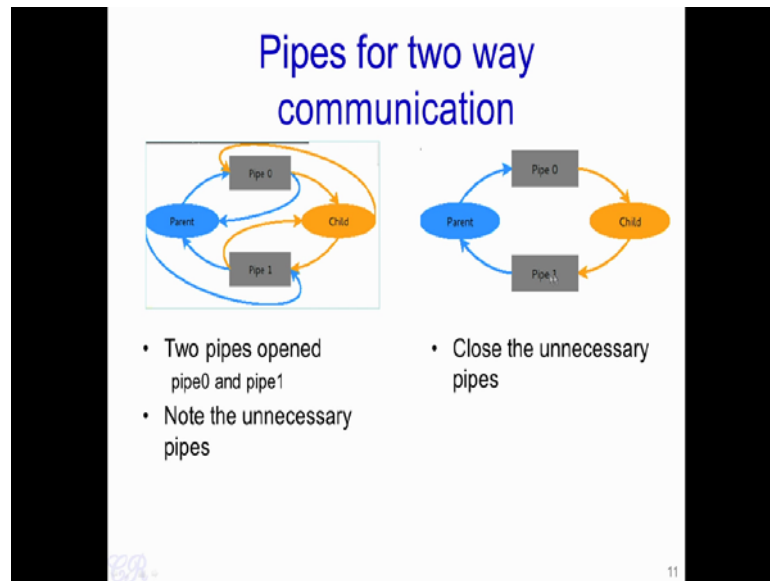
receive involves the marshalling or demarshalling of information. As we know in general a system call has significant overheads. Therefore, message passing is quite slow compared to shared memory. However, it is less error prone than shared memory because the kernel manages the sharing, and therefore would be able to do the synchronization between the process 1 and process 2.

(Refer Slide Time: 14:27)



Another very common application of message passing is the use of Pipes. Now pipes are used between parent and child processes only. Essentially, you can only communicate data from a parent process to a child process and vice versa. Another aspect of pipes is that it is uni directional, now when a pipe is created generally there are two file descriptors which are associated with it; that is fd 0 which is used to read from the pipe, while fd 1 is used to write to the pipe. We know the uni directional nature of this particular IPC. So, fd 1 is exclusively used to write to the pipe buffer, while fd 0 is used to read from the pipe buffer.

(Refer Slide Time: 15:14)



In order to obtain two way communications between the parent and the child we would require two pipes to be obtained; pipe 0 and pipe 1. So, when pipe 0 is opened it would create two file descriptors; one to write into pipe 0 and the other one to read to from pipe 0. Similarly, there are two file descriptors to write to pipe 1 and read from pipe 1. Similarly there is two pair of file descriptors for pipes in the child process.

Now we know that these additional descriptors are not required, therefore we could actually close the extra file descriptors to obtain something like this. Now in order that the parent senses some information to the child, the parent will write to pipe 0 while the child will read from pipe 0. In order to that the child sense some information to the parent the child will write to pipe 1, while the while the parent will read from the pipe 1.

(Refer Slide Time: 16:23)

### Example (child process sending a string to parent)

```
#include <stdio.h>
#include <unistd.h>
#include <stdlib.h>

int main() {
    int pipefd[2];
    int pid;
    char recv[32];

    pipe(pipefd);

    switch(pid=fork()) {
        case -1: perror("fork");
                exit(1);
        case 0: /* in child process */
                close(pipefd[0]); /* close unnecessary pipefd */
                FILE *out = fdopen(pipefd[1], "w"); /* open pipe descriptor as stream */
                fprintf(out, "Hello world\n"); /* write to out stream */
                break;
        default: /* in parent process */
                close(pipefd[1]); /* close unnecessary pipefd */
                FILE *in = fdopen(pipefd[0], "r"); /* open descriptor as stream */
                fscanf(in, "%s", recv); /* read from in stream */
                printf("%s", recv);
                break;
    }
}
```

12

Let us see an example of the user pipes. This again is a standard linux example and as we have said, so we create a parent and child process so that there is a uni directional communication from the parent to the child. So let us look at this particular thing. First we invoke this particular system call called pipe and I will give this pipefd, so pipefd is defined over here and taking two elements and then we invoke the system called fork. As we have studied the fork system would call would create a child process which is the exact duplicate of the parent process. In the parent process the value of a pid that is a value what fork returns is the child's pid value, and it is a value which is greater than 0. On the other hand in the child process the value return by fork is 0.

So, let us see what happens in the child process first. In the child process because the pid value is 0 so it would come over here and the first thing what we do is we close pipefd 0. So, closing pipefd 0 would mean that we are closing the repipe to the child, that is so we are we having like two pipes over here corresponding to pipe 0 and we have just closing the read pipe while the pipe 1 that is the right file descriptor is still open. Now let us see what happens in the child process. In the child process the value of pid returned by fork is 0, therefore it will result in this particular code being executed.

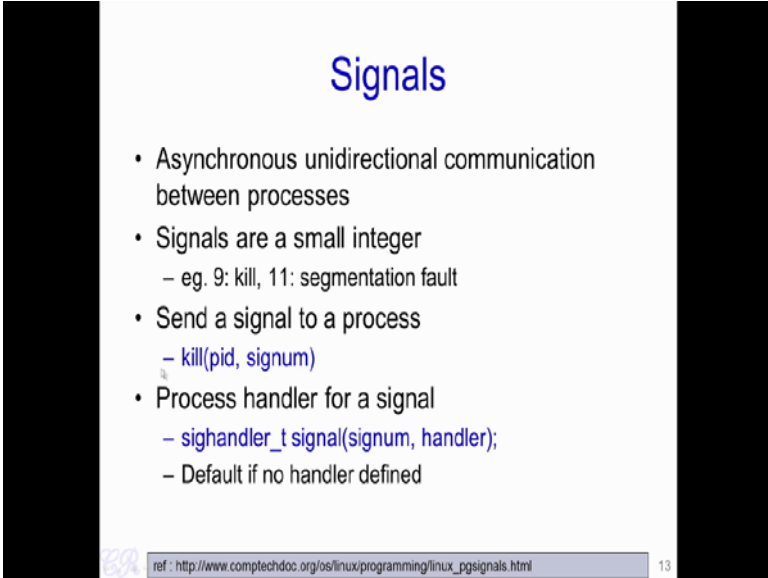
In this code first, there is we are closing the pipefd 0. So this means that the file

descriptor corresponding to the read pipe is closed. Second, we are then opening second file called fdopen corresponding to file descriptor 1 and it is opened in the write mode and we could then use fprintf out “Hello world.” Essentially what we are doing is do this pipe with descriptor 1 we are writing hello world. If you go back to this particular thing, so what we are seeing is that we are writing hello world into this pipe 1.

Now in the parent process which would execute over here because fork would return with the value which is greater than 0, so we are closing the write pipe and opening using fdopen the read pipe. Then we are using fscanf to read whatever has been pushed into the pipe, in this case it is hello world and printing it to the screen.

So, essentially what we have implemented in this program is this lower part. If the child opens this pipe in the write mode and puts hello world into this pipe while the parent then opens the pipe 1 in the read mode and reads the string hello world from the pipe. Thus, the string hello world has been transferred from the child to the parent. So you can actually try to implement this particular program and execute in a Linux system.

(Refer Slide Time: 19:58)



**Signals**

- Asynchronous unidirectional communication between processes
- Signals are a small integer
  - eg. 9: kill, 11: segmentation fault
- Send a signal to a process
  - `kill(pid, signal)`
- Process handler for a signal
  - `sighandler_t signal(signal, handler);`
  - Default if no handler defined

ref : [http://www.comptechdoc.org/os/linux/programming/linux\\_pgsignals.html](http://www.comptechdoc.org/os/linux/programming/linux_pgsignals.html)

Besides, the message passing, shared memory and pipes that we have studied so far. Third way of inter process communication is by what is known as signals. So, signals are

asynchronous unidirectional communication between processes. The operating system defines some predefined signals and these signals could be sent from the operating system to a process, or between one process to another. Signals are essentially small integers, for instance you would find that each of these integers has a predefined meaning. For instance 9 would mean to kill a process, while 11 would mean a segmentation fault and so on.

In order to send a signal to the process we could for a instance have kill or send to this pid and we could send this particular number which is an integer. In order that the process handles that signal it is define this particular function called signal, which takes this signal number for instance 9 or 1 and then the handler. So, handler is a pointer to the signal handler for that particular number. As a result we could send signals from one process to another and depending on the type of the signal the corresponding handler would be executed.

In this video we had seen a brief introduction to IPC's. Essentially, we had seen different types of IPC's like, message passing, shared memory, pipes and asynchronous transfers using signals. These IPC's are used extensively in systems to communicate between processes. And therefore, we are able to achieve or built applications which are extremely modular with small programs with each program focusing on only a certain aspect in the entire system.

Thank you.