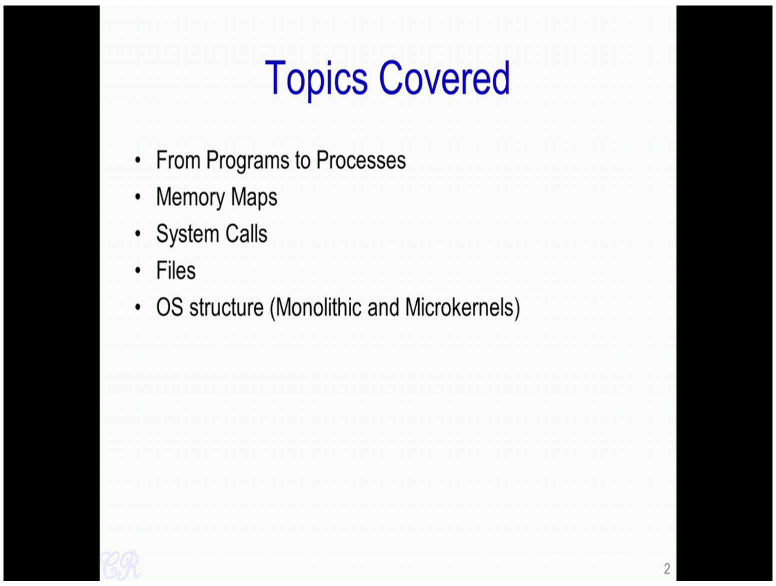
Introduction to Operating Systems

Prof. Chester Rebeiro
Department of Computer Science and Engineering
Indian Institute of Technology, Madras

Week - 01
Lecture - 03
From Programs to Processes

Hello. In today's class we will be having a very brief introduction to an operating concept called Processes.

(Refer Slide Time: 00:23)

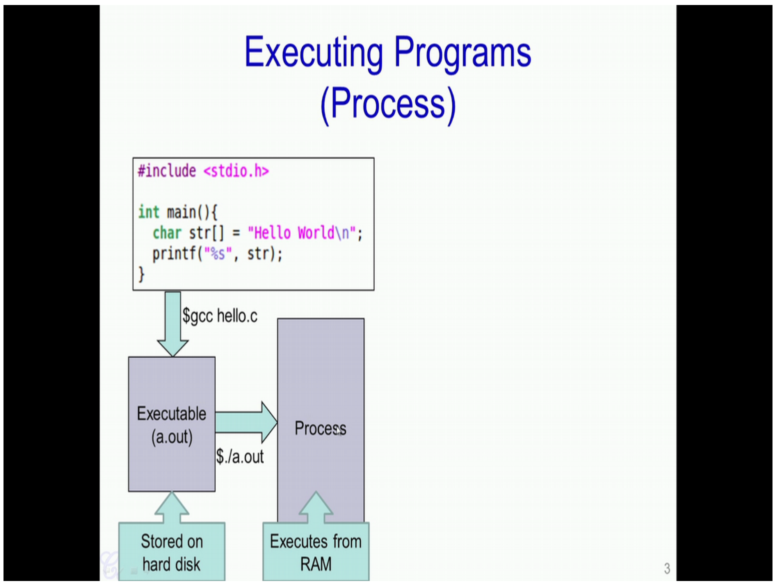


Topics Covered

- From Programs to Processes
- Memory Maps
- System Calls
- Files
- OS structure (Monolithic and Microkernels)

So, the topics which we will cover, is from Programs to Processes, Memory Maps, a System Calls, Files and essentially the structure of the Operating System.

(Refer Slide Time: 00:37)

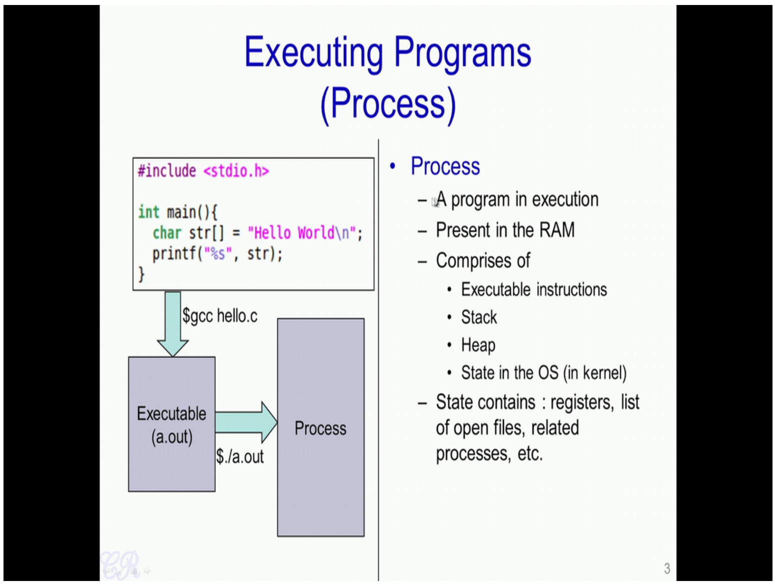


Executing Programs (Process)

```
#include <stdio.h>

int main(){
  char str[] = "Hello World\n";
  printf("%s", str);
}
```

Consider this particular program written in C. So, this program prints "Hello World" on to the screen. In order to compile and run this program, we first need to use a compiler such as gcc and specify the c code name such as hello.c in this case and what we will get is an executable, in this case it is called a.out. This executable is stored on the hard disk. In order to run this particular program we specify a command like ./a.out and it results in a process being created in the RAM. So, this process is essentially the - a.out program under execution, which is present in the RAM.

(Refer Slide Time: 01:18)



Executing Programs (Process)

```
#include <stdio.h>

int main(){
  char str[] = "Hello World\n";
  printf("%s", str);
}
```

$gcc hello.c
Executable (a.out)
$./a.out
Process

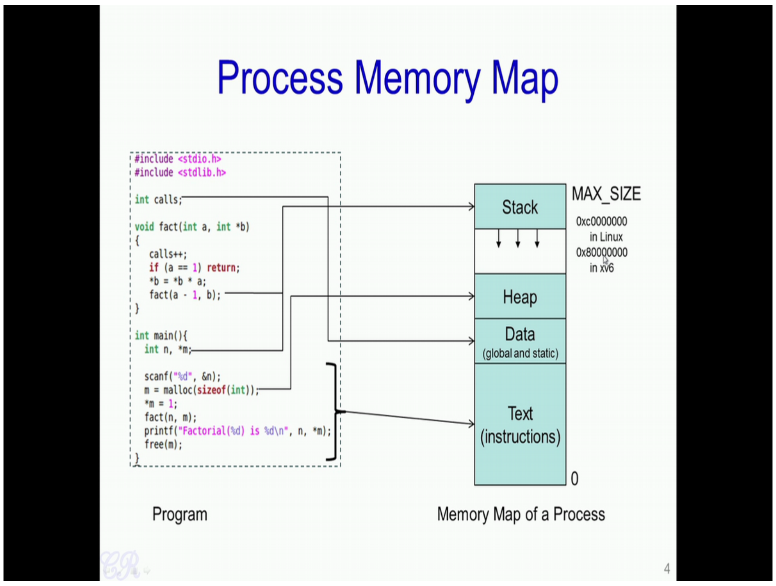
- Process
  - A program in execution
  - Present in the RAM
  - Comprises of
    - Executable instructions
    - Stack
    - Heap
    - State in the OS (in kernel)
  - State contains : registers, list of open files, related processes, etc.

3

To define it formally, a process is a program under execution which is executed from RAM and essentially comprises of various sections, such as the Executable instructions, Stack, Heap and also a hidden section known as the State. So, this state is actually maintained by the operating system and contains various things like the registers, list of open files, process, list of related processes etc..

So, in today's class we will look more into detail about what this particular process contains and how it is managed.

(Refer Slide Time: 01:55)



Process Memory Map

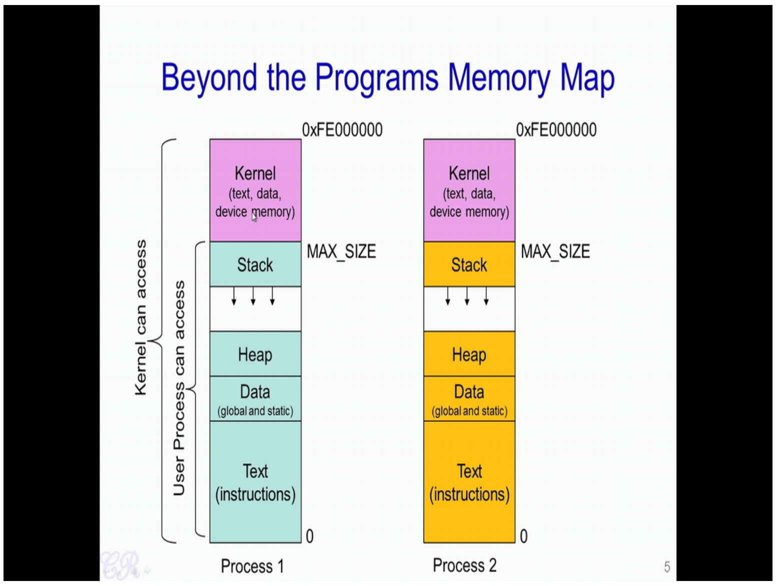
```c
#include <stdio.h>
#include <stdlib.h>

int calls;

void fact(int a, int *b)
{
   calls++;
   if (a == 1) return;
   *b = *b * a;
   fact(a - 1, b);
}

int main(){
  int n, *m;

  scanf("%d", &n);
  m = malloc(sizeof(int));
  *m = 1;
  fact(n, m);
  printf("Factorial(%d) is %d\n", n, *m);
  free(m);
}
```

So, Let us take a very simple example. So, this is a program and this is a process that is created when this program is executed. Now the process has various sections for example, it has the Text, Data, Heap and the Stack. Now various parts of this program when executed get mapped into the various sections of the process. For instance, all the instruction such as the instructions involved in the function main will get mapped into the text section of the process. Similarly, other functions such as the fact() function (mentioned in above slide) the instructions involved in this will also get mapped into the text section.

Now the global data and also static data gets mapped into the data section of the process. So, this section is actually divided into two parts where called as initialized and non-initialized sections. Third section is the heap, now any dynamically allocated memory such as m (mentioned in above slide) which is dynamically allocated using malloc gets created in the heap. Now the final section is called the stack, which contains all the local variables such as n and m and also information about function invocation. For example, in this case we have a recursive function which is getting invoked. So, all this information is present in the stack.

Now, the memory map of a process comprising all of the sections has a maximum limit called the MAX SIZE. So, typically at least in processes which are used in typical operating systems these days, this MAX SIZE is going to be fixed by the OS. For instance in a 32 bit Linux operating system, the MAX SIZE of every process is fixed at 0xc0000000. In the xv6 operating system which we are looking at for this course, the MAX SIZE of a process is fixed at the address 0x80000000.

(Refer Slide Time: 04:07)



Beyond the Programs Memory Map

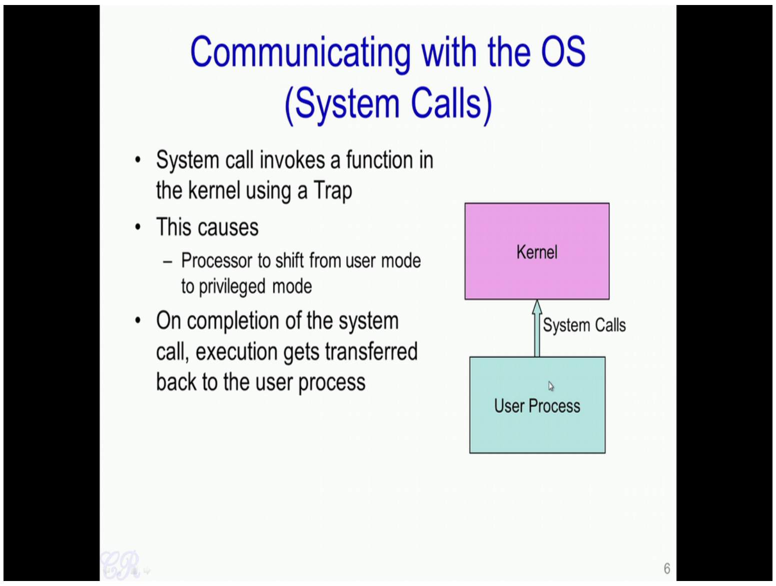
So, what we had seen is that if every process, a program that is a program under execution gets mapped into an area which starts at 0 and ends at MAX SIZE. So, what is present beyond this MAX SIZE of the process? So, typically the Kernel or the operating system gets mapped to the memory region from the MAX SIZE to the maximum limit. The entire thing like Text i.e the instructions of the operating system, operating system data, the OS heap and also device memory gets mapped into this upper region of the OS.

So, typically any user program could access any part of this lower region i.e the green region (mentioned in above slide). So, there would not be any problem to actually read data from any of these user space regions or even write data to parts of these user space regions. But, however, the process cannot access any data present in the Kernel memory that is beyond the MAX SIZE limit. However, the Kernel or the operating system which is executing from this upper region can access data from any part of the region that is it could execute or access data from in this kernel region as well as in the user space region.

Now what happens when we actually have multiple processes running in the same system? So, each process would have its own memory map, we having its own instructions, data, heap and stack, and also the kernel component is also present beyond the MAX SIZE. So, what you see is that every process in the system would have the kernel starting at MAX SIZE and extending beyond. Only the lower parts and this kernel part is going to be same for every process that is executing in the system. Below this MAX SIZE is going to vary from one process to another.

So, what does this mean? So, what it means is that when you execute one process and then executing another process, the regions above the MAX SIZE is going to be similar, while the regions below the MAX SIZE is going to change from one process to another. So, we mention that user programs will not be able to access any data in the kernel space. So, in that case how does the user program actually invoke the operating system?

(Refer Slide Time: 06:40)



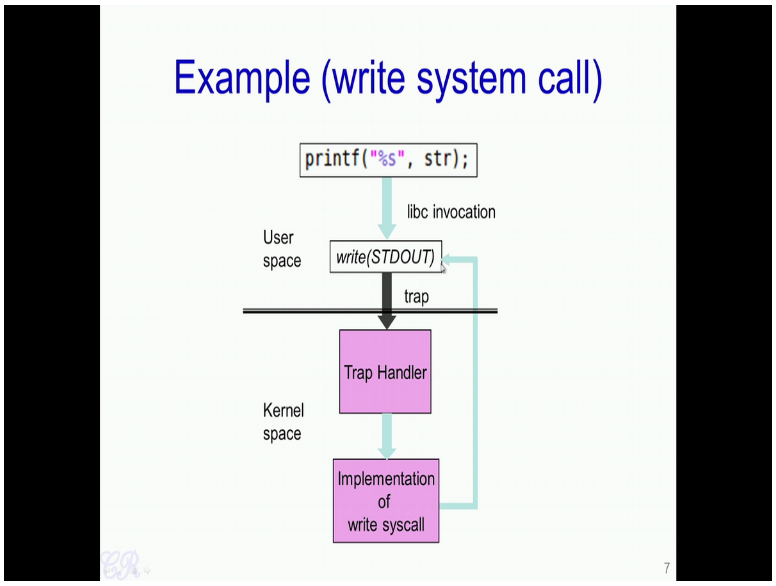
Communicating with the OS (System Calls)

- System call invokes a function in the kernel using a Trap
- This causes
  - Processor to shift from user mode to privileged mode
- On completion of the system call, execution gets transferred back to the user process

So, there are special invocation functions which the operating system support these are known as System Calls. So a System Calls are a set of functions which the OS support and a User Process could invoke any of the system calls to get information or to access hardware for other resources within the Kernel.

So, what happens when a system call is invoked; is that a process which is generally running in a user mode gets shifted to something known as a kernel mode or a privilege mode which will allow the kernel or the operating system to actually execute. When the system call completes execution then the user process will resume its execution from where it actually stopped.

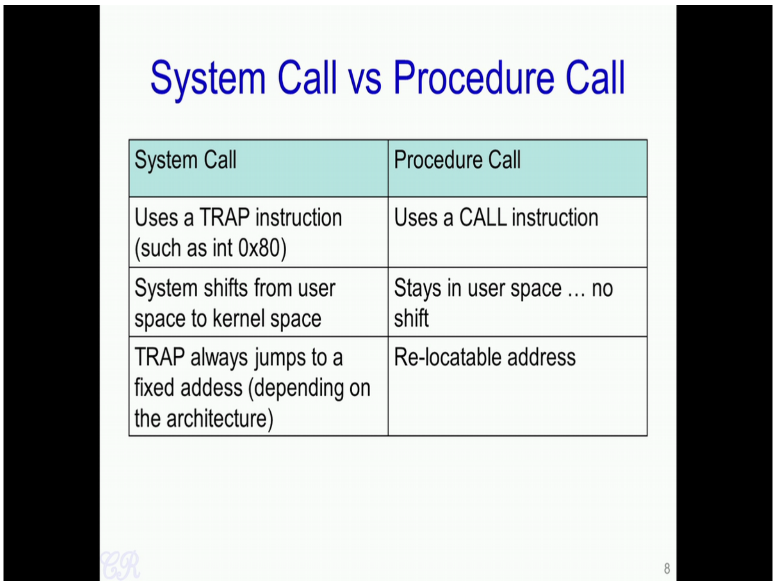
(Refer Slide Time: 07:25)


Example (write system call)

So Let us take an example of the printf statement. So, printf in fact is a library call. So, it is present in this libc and it results in particular function in the User space known as write() to be invoked and printf() function will then invoke a system call called write, with the parameter call STDOUT. So, STDOUT is a special parameter which essentially tells the operating system that this string provided by printf should be displayed on to the standard output that is the screen.

So, the write is the system call which causes a trap to be triggered, and this trap will result in something known as a Trap Handler in the Kernel space to be executed and the Trap Handler would then invoke a function which will correspond to the write system call. So, this write system call will then be responsible for actually printing the string provided by str on to the screen. After the write system call completes, then the execution is transferred back to the user space and the process continues to execute.

(Refer Slide Time: 08:40)



System Call vs Procedure Call

System Call	Procedure Call
Uses a TRAP instruction (such as int 0x80)	Uses a CALL instruction
System shifts from user space to kernel space	Stays in user space … no shift
TRAP always jumps to a fixed addess (depending on the architecture)	Re-locatable address

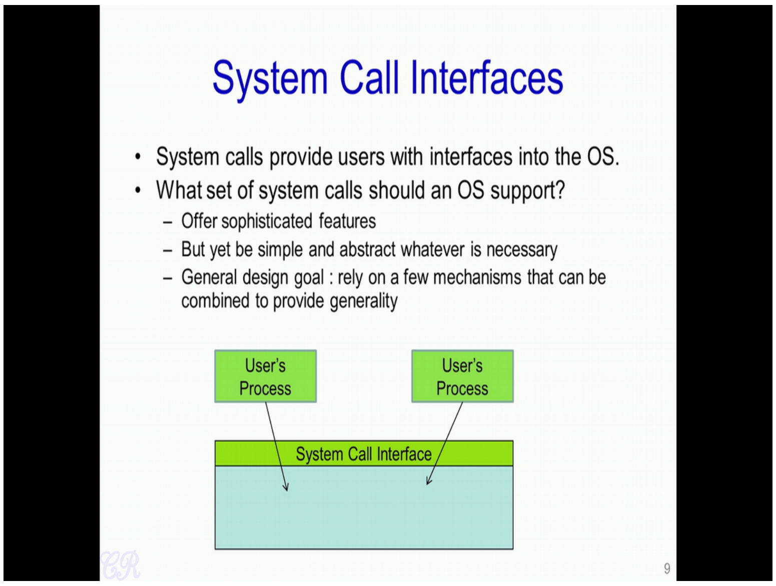
8

What is the difference between a System Call verses a Standard Function Call or Procedure Call. So, one important difference is that, when we want to invoke a function in a program or in a process we use an instruction such as a CALL instruction this is a standard x68 assembly instruction, and this will result in the function getting called and after that function gets invoked it returns back to the calling function.

In order to invoke a system call however, we use a TRAP instruction such as the int 0x80. So, int here stands for interrupt or software interrupt and it results in the system shifting from the user space or the user space mode of operation to the kernel space. So, the trap instruction causes the kernel to be invoked and it causes instructions in the kernel to then be executed. However, when we use the standard function call or the procedure call using the CALL instruction there is no change or shift from user space to kernel space and so on. So, the execution continues to remain in the user space as it was before.

Another very settle difference between the system call and a standard procedure call or a function call is that the destination address or the destination function which is invoked can be at a relocatable address. So, it could change every time the program is compiled and so on. However with the system call when a trap instruction is used the hardware actually or the processor actually decides where the next instruction in the kernel space should get invoked. So, this is going to be fixed irrespective of what program is running, what operating system is running, and so on.

(Refer Slide Time: 10:30)



System Call Interfaces

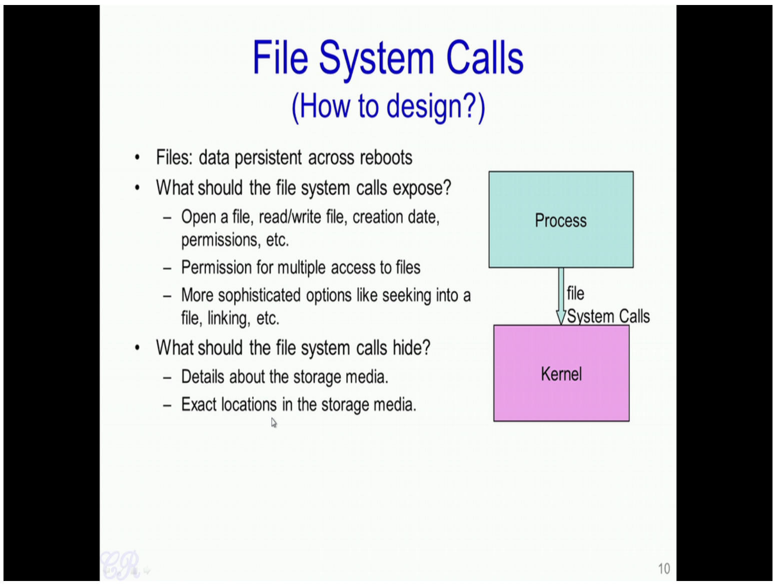
- System calls provide users with interfaces into the OS.
- What set of system calls should an OS support?
  - Offer sophisticated features
  - But yet be simple and abstract whatever is necessary
  - General design goal : rely on a few mechanisms that can be combined to provide generality

User's Process
User's Process
System Call Interface
9

So, one crucial aspect when actually designing operating systems is what system call should the operating system supports? We had seen that the only way a user process could invoke a particular functionality in the operating system is through the system call interface. So, the question now comes that if a person is actually designing an operating system, so what are the interface that the system call should support.

So, one obvious requirement is that the system call interface should have several sophisticated features so that a user process could actually very easily be able to interface or invoke several important functionality in the operating system. However a different approach is to have a very simple system call interface and abstract whatever is necessary from the operating system. So, we will see in the next slide, a particular example in this.

(Refer Slide Time: 11:29)



File System Calls
(How to design?)

- Files: data persistent across reboots
- What should the file system calls expose?
  - Open a file, read/write file, creation date, permissions, etc.
  - Permission for multiple access to files
  - More sophisticated options like seeking into a file, linking, etc.
- What should the file system calls hide?
  - Details about the storage media.
  - Exact locations in the storage media.

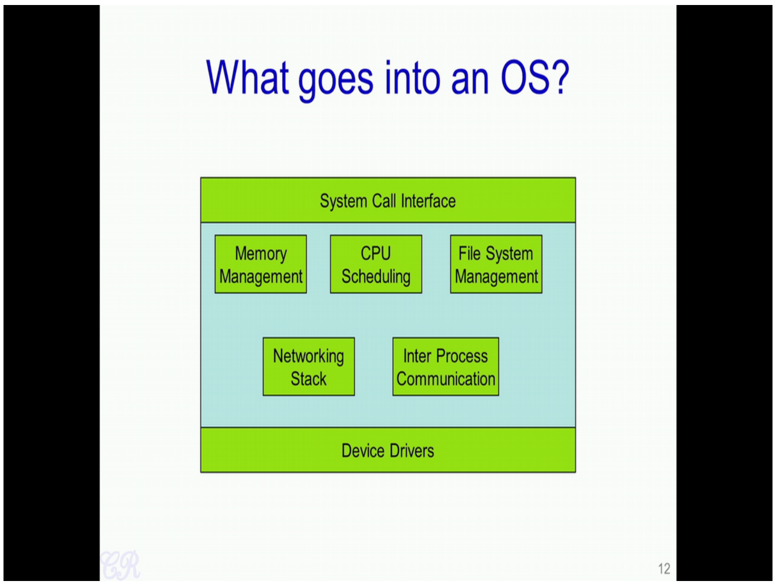
10

Let us take an example of system calls that an OS supports for accessing files. So, as we know files are a data which is persistent across reboots, so these are data which is stored in the hard disk and could be read, written or accessed using function such as fopen, close, read, write and so on. Every time we do a file open, it would require that the hard disk be accessed, so a process would need to invoke a system call into the kernel and the kernel should actually take care of accessing the hard disk or a hard disk buffers or any other storage medium to open the file and return back a pointer to the process.

Now the question comes is how does a operating system designer decide what system call should be provided or supported in order to access files? So, some of the obvious things are like there should be system calls to open a file, read or write to a file, there should be system calls to actually modify the creation date, set permissions and so on. The operating system could also support more complicated or more sophisticated operations such as being able to seek into a particular offset within a file, be able to link between files and so on. So, these are the essential requirements that a system call for handling files should support.

On the other hand, operating system should be able to hide some details about the file. For instance, details which should not be supported by system calls is like things such as like details about the storage media where the file is stored, for instance whether the file is stored on a USB drive or a Hard disk or a CD-ROM. This is actually abstracted out by the operating system, and the user process would not be easily able to know where the file is stored. Another aspect which is abstracted out by the operating system is the exact locations in the storage media.

(Refer Slide Time: 13:40)

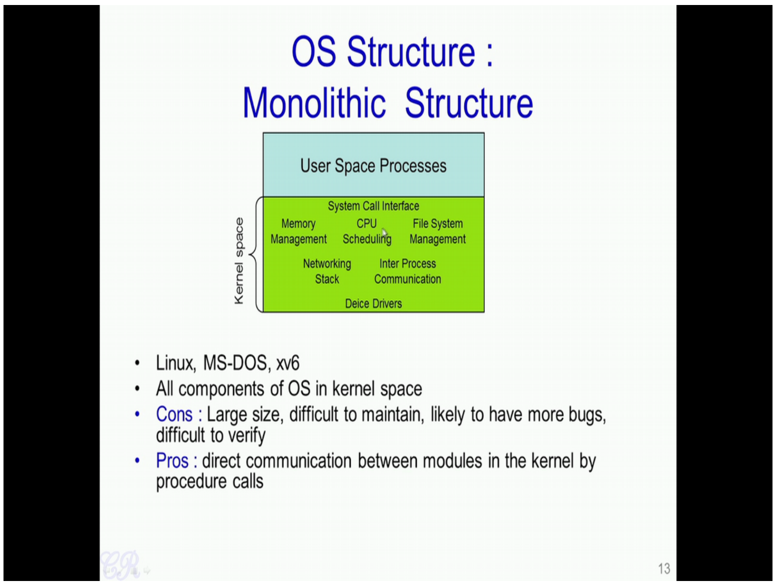


We will now look at how a typical Operating System structure looks like. So, the Operating System, suppose we consider this as a big green block have a several modules built internally. For example, it would have a memory management module which manages all the memory in the system, it would have a CPU scheduling block that is also the file system management module which will control how the file system such as the those present in hard disk or a CD-ROMs are managed. So, you have a networking stack which manages the TCP/IP network and you have something known as the inter process communication module which would take care of processes communicating with each other.

So, two important things which have not been mentioned as yet is the System Call

Interface, which allows user processes to actually access features or functionalities within each of these modules. Another aspect is the Device Drivers which would take care of communicating with the hardware devices and other hardware resources within the system. So, this essentially is all the different modules that an operating system supports.

(Refer Slide Time: 14:52)


User Space Processes
System Call Interface
Memory Management
CPU Scheduling
File System Management
Networking Stack
Inter Process Communication
Deice Drivers
Kernel space

OS Structure : Monolithic Structure

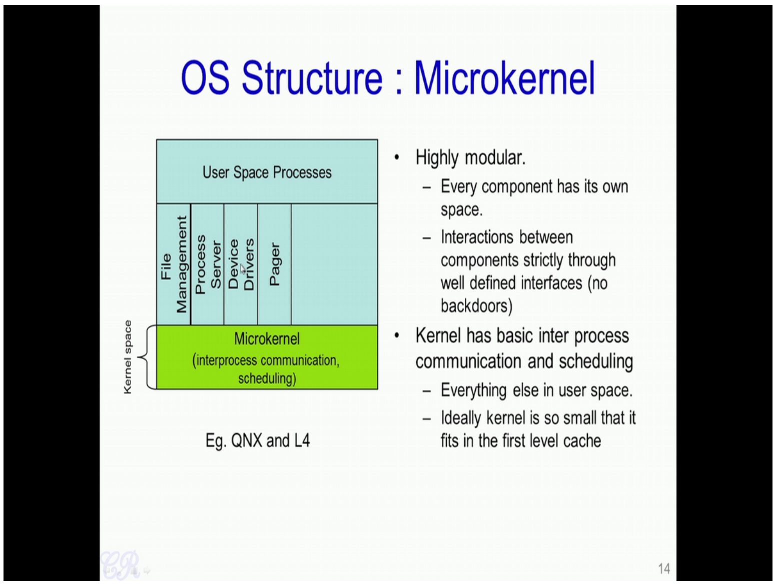
- Linux, MS-DOS, xv6
- All components of OS in kernel space
- Cons : Large size, difficult to maintain, likely to have more bugs, difficult to verify
- Pros : direct communication between modules in the kernel by procedure calls

13

So, In a Monolithic Structure of an operating system, all these various modules in the OS are present in a single addressable kernel space, so what this means is that this is just one large chunk of code and all of them are you could think of as one large program where all these modules are present in. So Therefore, calling any function from the memory management to say the networking stack would just mean a simple function call. Similarly from the networking stack to the device driver would be another function call.

This is essentially the advantage of having such a monolithic structure, is that you could there is a direct communication between one module and another. On the other hand the Kernel space becomes very large, and therefore, difficult to maintain and is likely to have more bugs. So, typical operating system such as Linux and xv6 and MS-DOS uses a monolithic structure. So, to take an example the Linux operating system or the Linux Kernel has around 10 million lines of code. So, all these 10 million lines of code comprises of the entire kernel Linux Kernel which is actually present in this area.

(Refer Slide Time: 16:14)



OS Structure : Microkernel

User Space Processes
File Management
Process Server
Device Drivers
Pager
Microkernel (interprocess communication, scheduling)
Kernel space

Eg. QNX and L4

- Highly modular.
  - Every component has its own space.
  - Interactions between components strictly through well defined interfaces (no backdoors)
- Kernel has basic inter process communication and scheduling
  - Everything else in user space.
  - Ideally kernel is so small that it fits in the first level cache

14

Another common structure of the operating system is known as the Microkernel structure, where the kernel is actually highly modular and every component in the kernel has its own addressable space. So, it is like having each of these as independent processes and you have a very small microkernel which actually runs in the kernel space, which is in charge of managing communication between each of these processes, and also communication between user process and the operating system processes.

So, the advantage here is that this microkernel is extremely small. So, ideally it is small enough to actually fit into the L1 cache of the system itself, so typically it would be quite fast. However, the drawback is that you now cannot have direct calls from say the file management to a device driver or rather like unlike the monolithic kernel where you could make direct function invocation from a file management module to a device driver function. Here every invocation of that form should be through a communication channel known as an IPC or Inter Process Communication channel.

With this we would actually end today's lecture, and from the next lecture we will actually look more about CPU sharing.

Thank you.