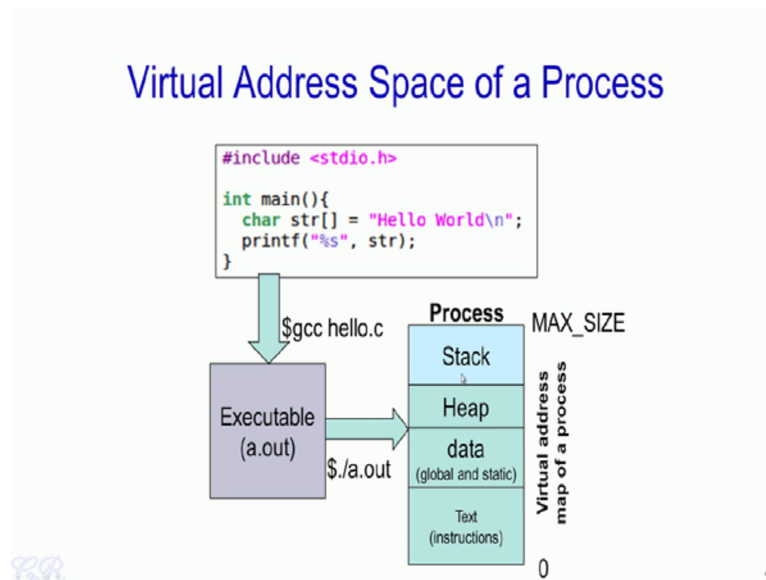
Introduction to Operating Systems
Prof. Chester Rebeiro
Department of Computer Science and Engineering
Indian Institute of Technology, Madras

Week - 03
Lecture - 11
Operating Systems (Processes)

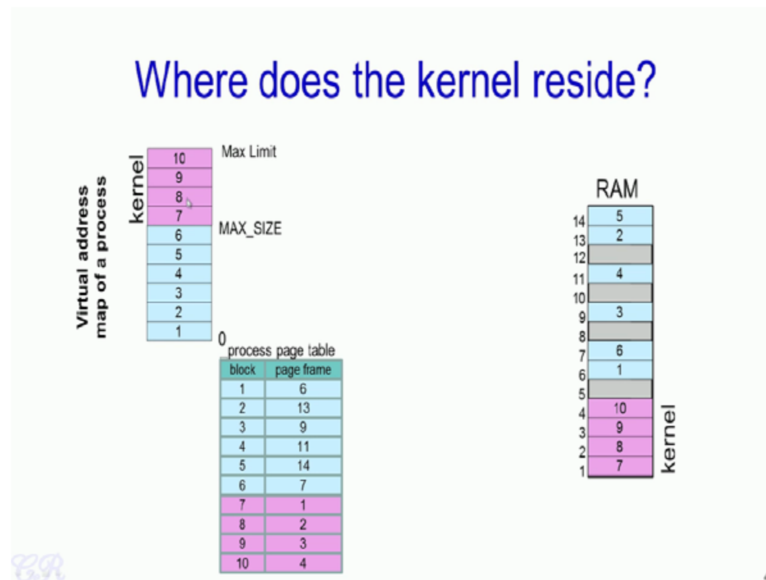
Hello, and welcome to this video. In this video we will looked at Processes, which is possibly the most crucial part of operating systems. So, processes as we know are a program in execution. We will see today how operating systems manage processes.

(Refer Slide Time: 00:38)



Let us start with this now famous example of printing "Hello world" on to a screen. So, when compiled with gcc hello dot c it creates an executable a dot out. When a dot out is executed a process is created, part of this process will be in the RAM and it is identified by a virtual address map. The virtual address map is a sequence of contiguous addressable memory locations starting from 0 to a limit of MAX SIZE. So, within this virtual address map we have various aspects of the process including the instructions, global and static data, heap, as well as the stack.

(Refer Slide Time: 01:28)



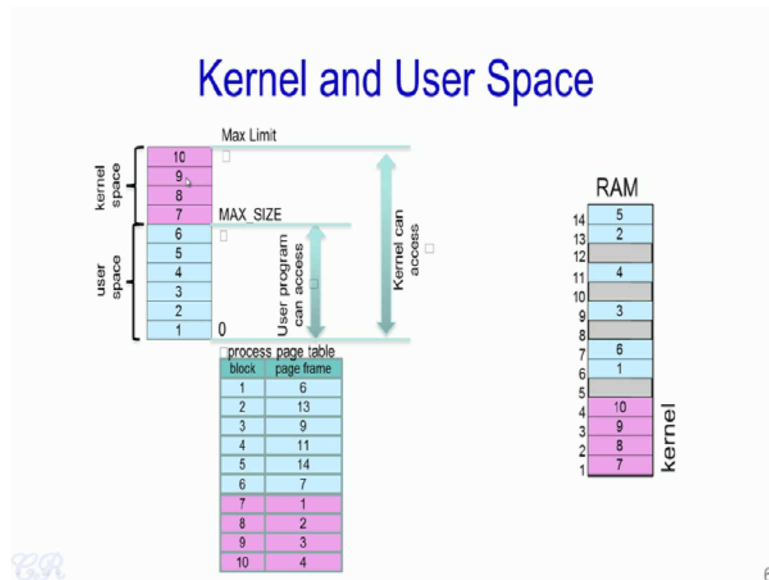
As we have seen in an earlier video, the virtual address space or virtual address map of a process is divided into equally sized blocks. Typically, the size of each block is 4 kilo bytes. Again each process would also have process page table in memory which maps each block of the process into a corresponding page frame. The RAM, as we have seen is divided into page frames of size 4 KB similar to the block size. And these page frames contain the actual code and data of the process which is being executed.

Now we have seen these in a previous video, but the question which we need to ask is; where does the operating system or where does the kernel reside in this entire a scheme. As we know the kernel is other software and has to be present in the RAM to execute. Thus, in most operating systems such as Linux as well as in the operating systems which we are studying that is xv6, the kernel resides in the lower part of the memory starting from page prints 1, 2, 3, and so on.

Just like every other page frame the kernel two is divided into page frames of equal size. Now a sense we are using the virtual addressing in the system the page frames corresponding to the kernel are mapped into the virtual address space of the process. So, the kernel code and data are present above the max size and below limit known as max limit. Now, again in the processes page table there are entries corresponding to this map.

For instance 7, 8, 9, and 10 corresponding to the blocks that have the kernel code and data and the page table tells us that they are mapped into page prints 1, 2, 3, and 4.

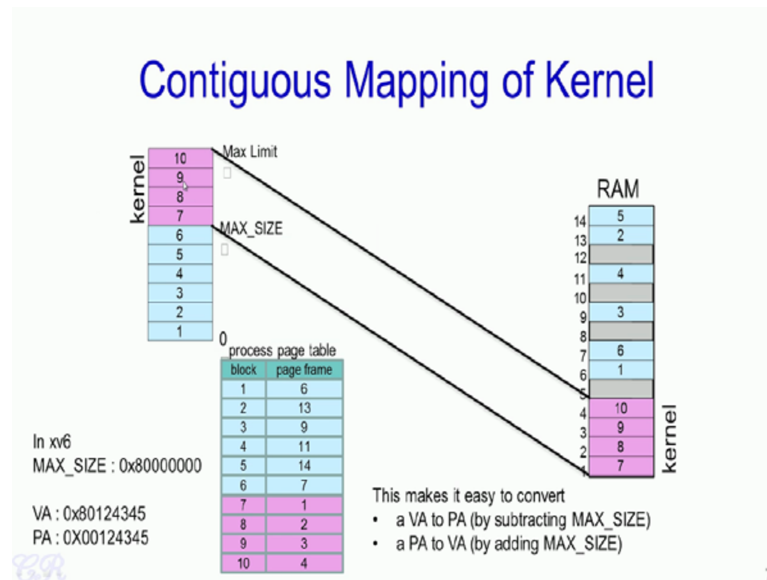
(Refer Slide Time: 03:42)



Now, we could divide this particular virtual address base into two components. One is the user space which corresponds to this blue area which contains the user processes, code, data, and other segments such as the stack and heap. Again, there is the kernel space which corresponds to the kernel code data and other aspects of the kernel.

So, the MAX SIZE defines the boundary between the user space and the kernel space. A user program can only access any code or data present in this user space. The user program cannot access anything in the kernel space. On the other hand, the Kernel can access code as well as data in both the kernel space as well as the user space. So, this prevents that user space programs from maliciously modifying data or modifying kernel structures.

(Refer Slide Time: 04:51)

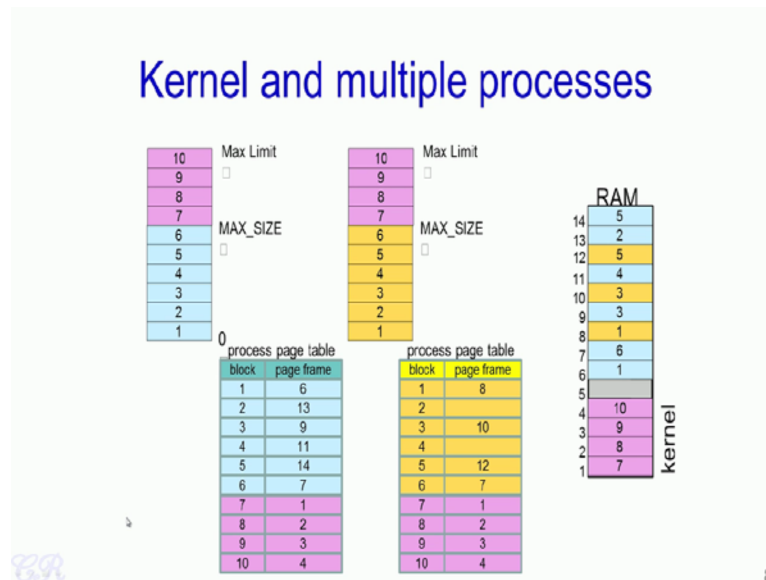


So, another thing to notice is that there is a contiguous mapping between the kernel addresses in the virtual space of the process to the corresponding physical frames in which the kernel gets mapped into. For instance, the kernel blocks 7, 8, 9, and 10 get mapped into the contiguous page frames 1, 2, 3, and 4. So, why is this contiguous mapping actually reduced? So, one most important aspect is that given this contiguous mapping it is easy for the kernel to make conversions from virtual address to physical address and vice versa.

For instance, to convert from virtual address in the kernel space to the corresponding physical address in the page frames of the kernel a simple subtraction by max size would do the trick. In xv6 where the max is defined as 0x80000000, a virtual address of 0x80124345 can be converted to the corresponding physical address by subtracting the max size. So, the physical address would be simply written as 0x00124345.

Similarly, a physical address corresponding in the kernel code and a data in the kernel page frames can be converted to the corresponding virtual address in the kernel space by adding max size. For example, in this case the physical address 0x00124345 can be converted to the corresponding virtual address in the kernel space by adding this max size to get - 0x80124345.

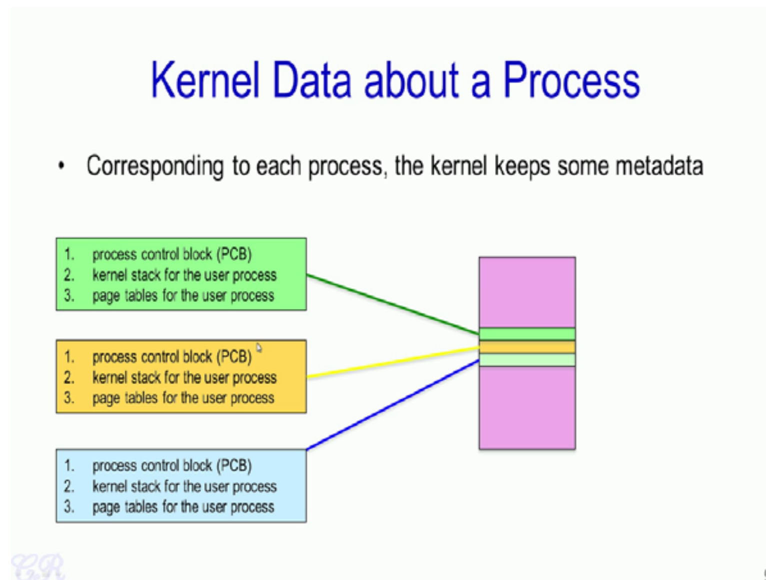
(Refer Slide Time: 06:52)



So, what happens when we have multiple processes in the system? The kernel space is mapped identically in all virtual address spaces of every process. For instance, above max size and below max limit the kernel space is present in all processes. Similarly, the page table in each process also has an identical mapping between the kernel page tables and the corresponding page frames that the kernel occupies, as can be seen in these few entries as well as these two entries.

Now one thing to be noticed is that all though the virtual address space of each process has different entries for the kernel, however all processes eventually map their kernel space into the same page frames in the RAM. So, what this means is that, we have just single copy of the kernel present in the RAM. However, there can be multiple identical entries in each processes page table corresponding to the kernel code and data.

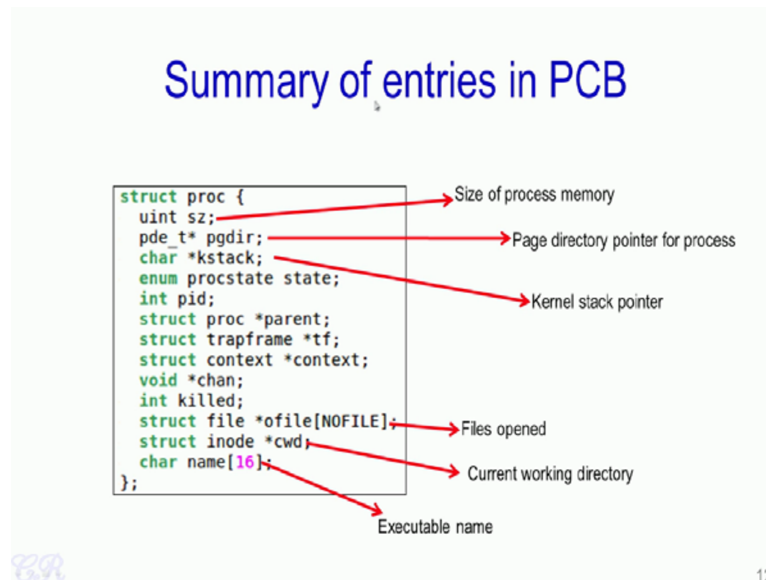
(Refer Slide Time: 08:12)



Now that we have seen where the kernel exist in the RAM as well as where it gets map to in the virtual address space of each process, now we will look at what metadata the kernel has corresponding to each process that runs in the system.

So, each process in the system has 3 metadata known as; the process control block, a kernel stack for that user process and the corresponding page table for that user process. So, each process that runs in the system will have these three locks that are unique for that process. We have already seen page tables map the virtual addressable space of that user process to the corresponding page frames that the process occupies. Now we will look at the other two metadata.

(Refer Slide Time: 09:06)



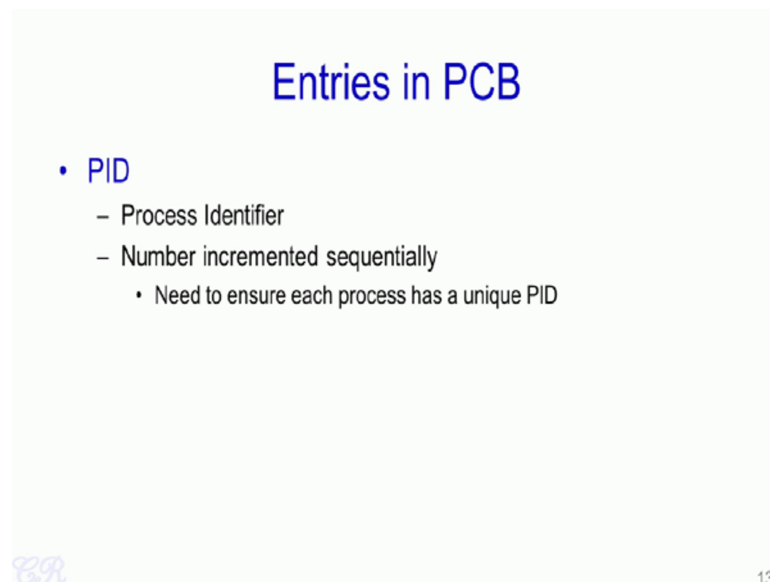
So, we have learnt that corresponding to each process there are various segments, and one important segment is the stack of the process. This process stack present in the user space would have information such as the local variables and also information about function calls. So this we will now call as the user space stack. In addition to the user space stack each process will also have something known as the kernel space stack or the kernel stack for that process. This kernel stack is used when the kernel executes in the context of a process. For instance, when the process executes a system call it results in some one kernel code executing and these kernel code would use the kernel stack for it is local variables as well as function calls.

Also this kernel stack is use for many other important aspects such as to store the context of process as well as to store. In addition to this standard use of these stack such as for local and auto variables as well as for function calls, the kernel stacks plays a crucial role in storing the context of a process which would allow the process to restart after a periods of time. So, why do we have two separate stacks? Why do we have a user stack for the process as well as the kernel stack? The advantage that we achieve is that the kernel can execute even if the user stack is corrupted. So attacks that target the stack, such as buffer overflow attack will not affect the kernel in such a case.

So, let us look at some of the important components in the PCB. This particular structure is taken from the xv6 operating systems a PCB which is defined as struct proc. Some of the important elements or aspects of this particular structure is as said, which is the size of the process memory pgdir which is the pointer to the page directory for the process. Kstack, which is the pointer to the kernel stack which we have defined few slides earlier.

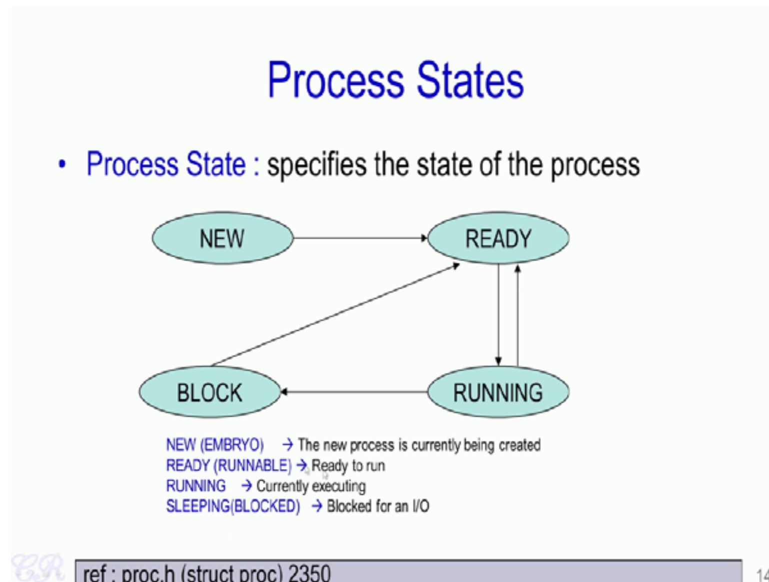
And there are other aspects such as, the list of files that are opened by the process, the current working directory of the process, and the executable name; for instance, a dot out in our example. So, we will look at some of these other parameters in the fourth coming slides.

(Refer Slide Time: 12:04)



An important entry in the PCB corresponding to each process is the PID or Process Identifier. This is an identifier for the process essentially defined as an integer and each process would have unique PID. Typically, the number would be incremented sequentially in such a manner that when a process is created it gets unique number.

(Refer Slide Time: 12:35)



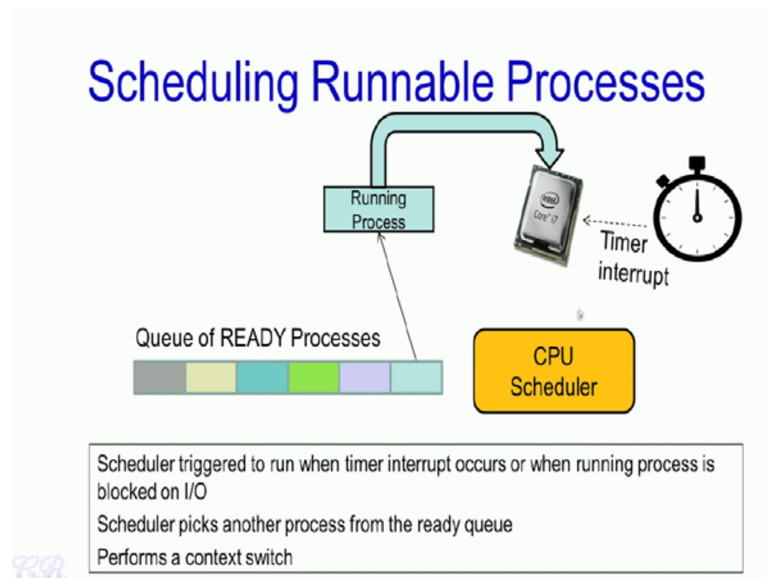
And other very important aspect a in the PCB is the state of the process. So, from the time process is created to the time it exists it moves through several states, such as the new, ready, block state, or running state. The xv6 calls these states by different names, such as the new is call the Embryo which means that a new process is currently being created, while Ready is known as the Runnable which means it is ready to run, while the Sleeping is known as the Block state and essentially blocked for an IO.

So, how and when does a process actually go from one state to another? When a new process is created it is initially in the state known as new, when it is ready to run the state is moved to what is known as the ready state, and when it finally runs on the processor it get shifted to the running state. After running for a while the process gets preempted from the processor in order to allow other processes to run, and in such a case it goes back from the running state to the ready state.

Now, suppose during the execution of the process there is some IO operation that is required, for instance the process could call invokes a scanner which requires the user to enter something through the keyboard. In such a case the process would be moved from a running state to a block state.

So, the process will remain in the block state until the event occurs. For instance, when the user enter something through the keyboard, when this event occurs the process moves from the block state back to the ready state, and this process of moving from one state to another from ready to running from running to back to ready or from running to blocked and then ready keeps going on through the entire life cycle of the process. At the end when the process exists or gets terminated it goes to what is known as an exit state, it is not shown in this diagram. So, you could actually look up the xv6 code proc dot h and which will tell give you more information about the various states. So, what is this ready state?

(Refer Slide Time: 15:19)



Operating systems maintain queue of processes which are all in the ready state; when an event such as the timer interrupt occurs module within the operating system known as the CPU scheduler gets triggered. This CPU scheduler then scans through these the queue of ready processes and selects one which then gets executed in the processor. This selected process then changes it is state from ready to running. The running process would continue to run until the next timer interrupt occurs, and the entire cycle repeats itself.

(Refer Slide Time: 16:03)

Entries in PCB

- Pointer to trapframe and pointer to context
  - Present as part of the kernel stack of a process.
  - Contains the state of all registers corresponding to the process
  - Used to restart a process after a context switch

SS
ESP
EFLAGS
CS
EIP
Error Code
Trap Number
ds
es
…
eax
ecx
…
esi
edi
esp
(empty)

trapframe →

16

Another entry in the PCB is pointers to what is known as a trapframe and context. So, these trapframe as well as context are part of the kernel stack and as seen in this figure they have lot of information about the current state of the running process. For instance, it would say the stack met segment, the stack pointer, the flag register, the code segment instruction pointer and so on. So, this particular trapframe and context is used when a process is restarted after a context switch.

(Refer Slide Time: 16:43)

Storing procs in xv6

- In a globally defined array present in ptable
- NPROC is the maximum number of processes that can be present in the system (#define NPROC 64)
- Also present in ptable is a lock that seralizes access to the array.

```
struct {
  struct spinlock lock;
  struct proc proc[NPROC];
} ptable;
```

ref : proc.c (struct ptable) 2409, params.h (NPROC) 0150

17

So, how are these various PCB stored in xv6? So, in xv6 structure known as ptable is defined. This structure has an array of struct procs, so remember that struct procs is actually the PCB structure in xv6. The array has NPROC entries, where NPROC is defined as 64. So, each process that was created in xv6 will have an entry in this particular array. So, you could have more information about in this particular structure by looking at the xv6 code proc dot c and the structure ptable. Also params dot h, is a file in xv6 which defines what NPROC is.

So, this gives us a brief introduction to how processes are managed in the operating system. In the next video, we will look at how process gets created, executes, and exits from the system.

Thank you.