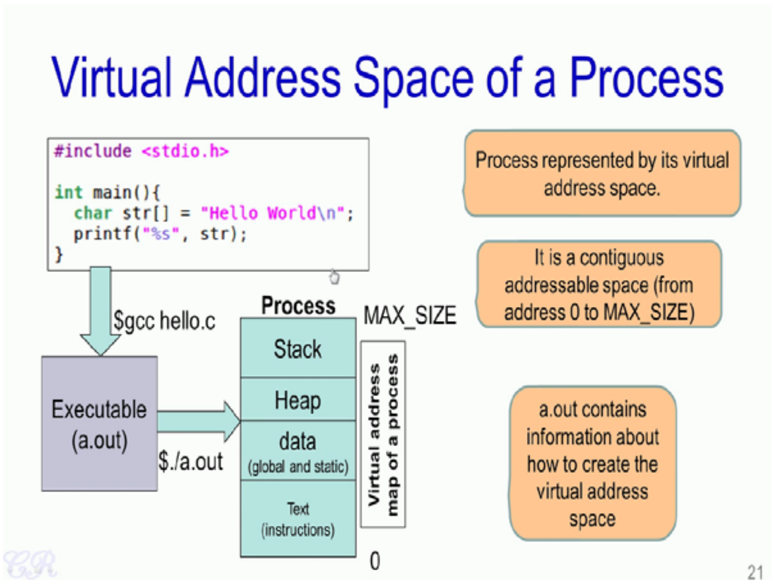
Introduction to Operating Systems
Prof. Chester Rebeiro
Department of Computer Science and Engineering
Indian Institute of Technology, Madras

Week - 02
Lecture – 07
Virtual Memory (MMU and Mapping)

Hello. In this video, we will discuss how the virtual memory mapping takes place between the Virtual Memory and the RAM.

(Refer Slide Time: 00:32)



Virtual Address Space of a Process

```
#include <stdio.h>

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

$gcc hello.c
Executable (a.out)
$./a.out
Process
Stack
Heap
data (global and static)
Text (instructions)
MAX_SIZE
0
Virtual address map of a process
Process represented by its virtual address space.
It is a contiguous addressable space (from address 0 to MAX_SIZE)
a.out contains information about how to create the virtual address space
21

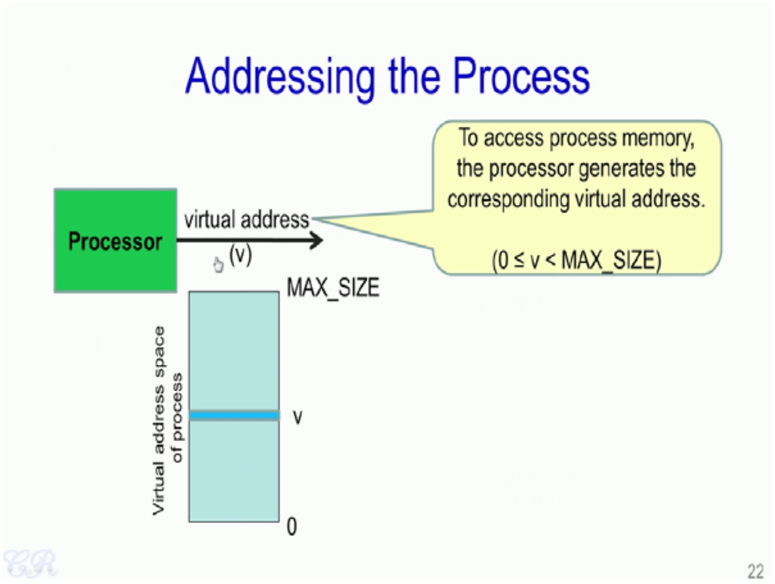
Essentially, let us start with this very famous slide by now; that when you write a program and compile it, you get the executable a dot out. And, when you execute this particular executable, a dot out executable, the operating system will create a process for you. Now, the process is represented by what is known as the virtual address space. Now, the virtual address space is a contiguous address space starting from 0 and extending up to an address, defined by something known as the MAX SIZE. This MAX SIZE comes from the xv6 nomenclature.

Now, within this contiguous virtual address space for the process are various sections like the Text, Data, Heap and the Stack. So, all these sections are created by the operating system. Essentially, the operating system will need to know where the heap would start, where the stack would start and so on. So, how does the operating system know all these information? So, this information is given to the OS through the - a dot out. Essentially what happens is that when the program is compiled and linked, the compiler will put a lot of this information in the executable a dot out.

So when executed, the operating system will read out this information and then determines where in the virtual address space should these various segments be present. So, now when this process is executing all these instructions corresponding to printf, for instance, gets executed. And, all addresses corresponding to ‘str’ is accessed. So, it may be noted that all these addresses, all these instructions, would be mapped into this particular virtual address space. For instance, if I were to print the address of “Hello World”, essentially the address of this ‘str’, then the result would be the virtual address with respect to this particular mapping.

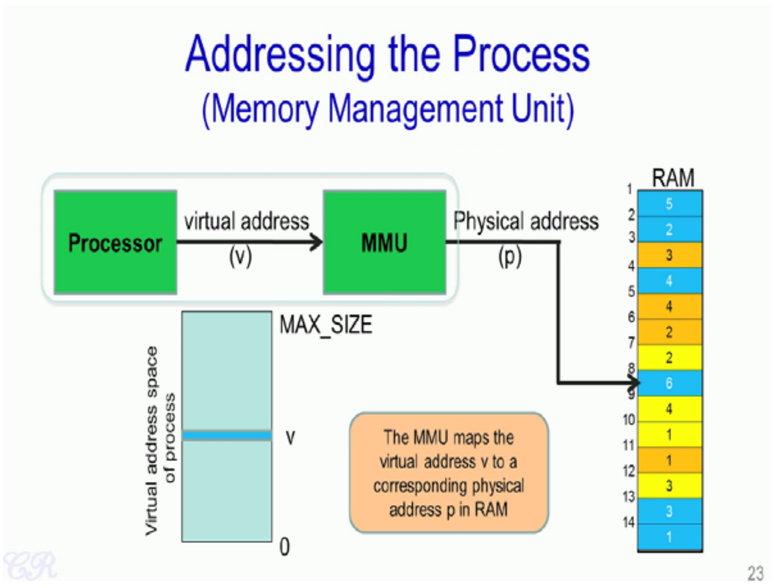
So, what you will next see is how this virtual address mapped; gets mapped into the main memory of the system. So, let us say that the processor wants to access this particular string, at least the base address of this string, and as we know that that would contain the letter ‘H’.

(Refer Slide Time: 03:41)


Addressing the Process
To access process memory, the processor generates the corresponding virtual address.
(0 ≤ v < MAX_SIZE)
Processor
virtual address
(v)
MAX_SIZE
v
0
Virtual address space of process
22

So, what would happen is that the processor will put out a virtual address on its bus. So, let us say this virtual address is v and this virtual address could be in the range from 0 to MAX SIZE. So, MAX SIZE is the largest address that a user space process could have in its virtual address space. So said another way, the virtual address v would be some address in the virtual address space of the process.

(Refer Slide Time: 04:19)

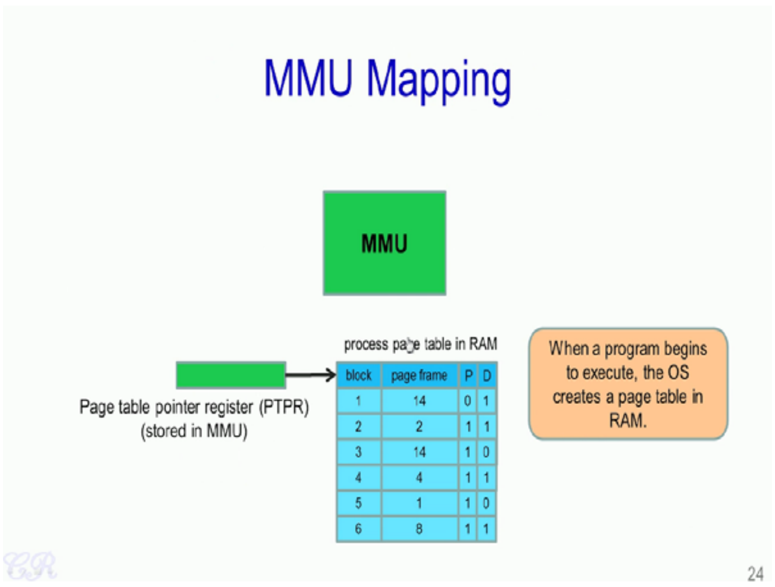

Addressing the Process
(Memory Management Unit)

Processor
virtual address (v)
MMU
Physical address (p)
RAM
MAX_SIZE
v
0
Virtual address space of process
The MMU maps the virtual address v to a corresponding physical address p in RAM
1
2
3
4
5
6
7
8
9
10
11
12
13
14
5
2
3
4
4
2
2
6
4
1
1
3
3
1
23

Next, another unit in the CPU known as the MMU or the Memory Management Unit, so this entire thing would typically be in the same package or jointly called the CPU or processor would then take the virtual address and convert it to a physical address ‘p’, which will then be used to access the RAM.

So, what we will see next is how this virtual address gets converted to the corresponding physical address. So, it may be noted that the virtual address corresponds to the virtual address map, but the physical address correspondence to 1, physical address in the main memory or the RAM.

(Refer Slide Time: 05:09)



MMU Mapping

MMU

process page table in RAM

block	page frame	P	D
1	14	0	1
2	2	1	1
3	14	1	0
4	4	1	1
5	1	1	0
6	8	1	1

Page table pointer register (PTPR)
(stored in MMU)

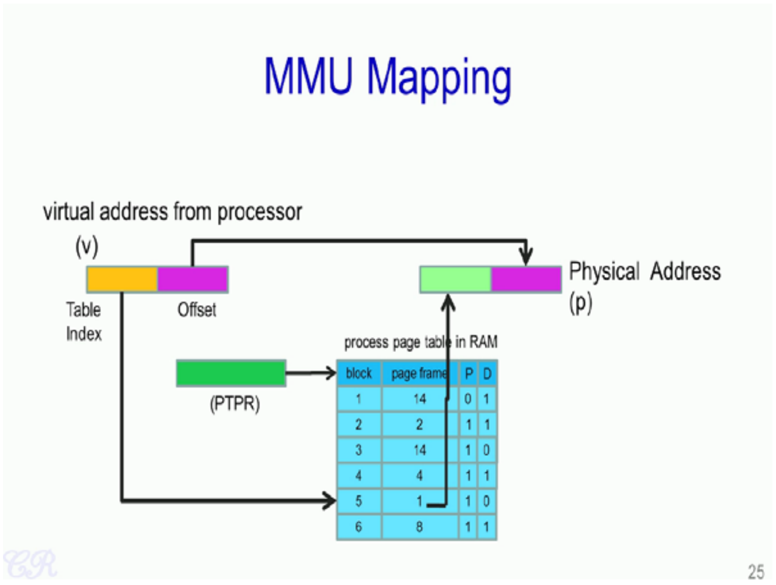
When a program begins to execute, the OS creates a page table in RAM.

24

So when a process begins to execute, the operating system would create a page table for that process in RAM. So, this is the page table. And as we have seen in the previous video, the page table holds the mapping from the virtual blocks or the virtual address space of the process to the physical page frames. And, we have other bits like the present bit, dirty bit and the protected bit, which is not shown here.

Now in the memory management unit, there is a register known as the page table pointer register or PTPR. So, in the Intel systems this PTPR or page table pointer register is known as the CR 3 register. And, we will see more details of this in a future video. So, this PTPR register present in the MMU will have a pointer to the process page table. Let us see how these things are used to provide the mapping from the virtual address space to the physical space.

(Refer Slide Time: 06:18)



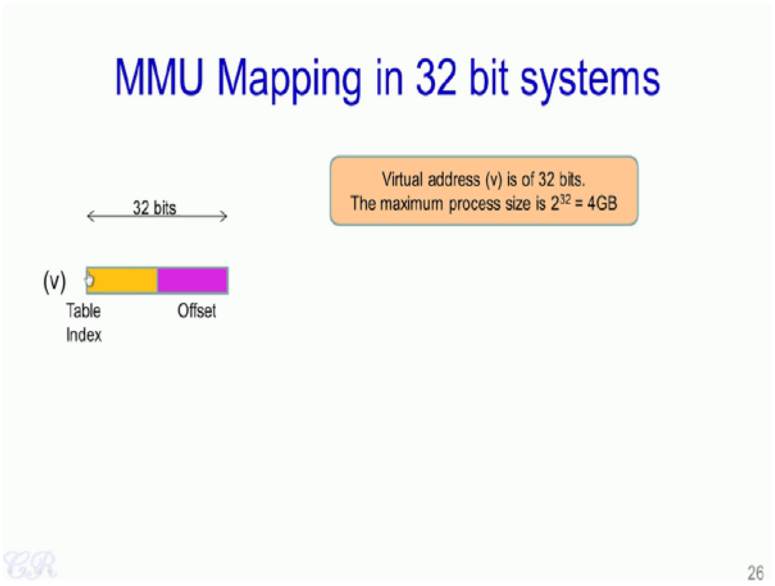
MMU Mapping

block	page frame	P	D
1	14	0	1
2	2	1	1
3	14	1	0
4	4	1	1
5	1	1	0
6	8	1	1

So, what happens is that the virtual address which has sent out by the processor core comprises of two parts. It has a table index, which are a few bits. Typically, they are more significant bits of the address and an offset. So, when the MMU obtains this virtual address. It is going to look up the process page table, corresponding to the offset present in the table index. So, the PTPR would be the base address of this particular process page table. And, the higher bits in the virtual address would correspond to the offset in the process page table. So from here, the corresponding page frame is taken, and it forms part of the physical address.

Now, the second part of the physical address is directly taken from the offset. So, in this way we would then create what is known as the physical address. And, it is this physical address that is used to access or to address the RAM on the main memory.

(Refer Slide Time: 07:40)



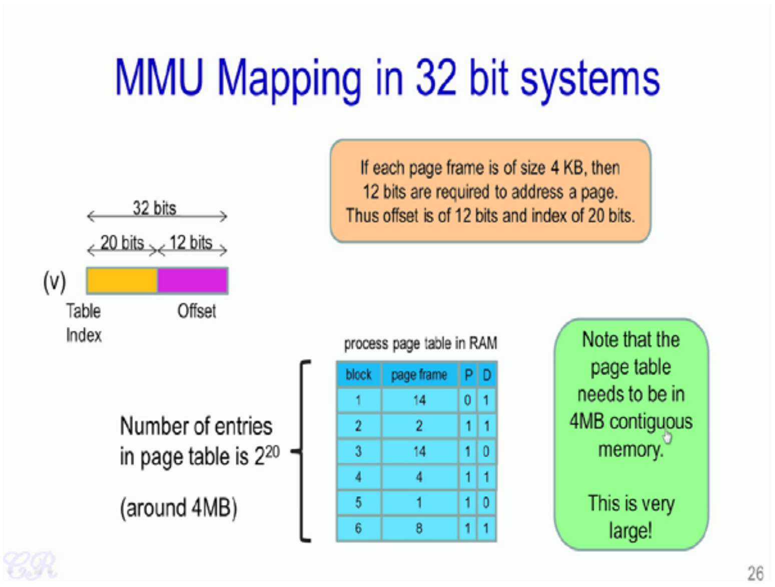
MMU Mapping in 32 bit systems

Virtual address (v) is of 32 bits.
The maximum process size is 2^{32} = 4GB

32 bits
(v)
Table Index
Offset
26

Let us see how this MMU mapping works for a 32-bit system. So in a 32-bit processor, the virtual address space could be utmost 2 power 32, that is, 4 giga bytes. The virtual addresses of 32 bits are shown over here.

(Refer Slide Time: 08:03)



MMU Mapping in 32 bit systems

If each page frame is of size 4 KB, then 12 bits are required to address a page. Thus offset is of 12 bits and index of 20 bits.

Number of entries in page table is 2^{20}

(around 4MB)

process page table in RAM

block	page frame	P	D
1	14	0	1
2	2	1	1
3	14	1	0
4	4	1	1
5	1	1	0
6	8	1	1

Note that the page table needs to be in 4MB contiguous memory.

This is very large!

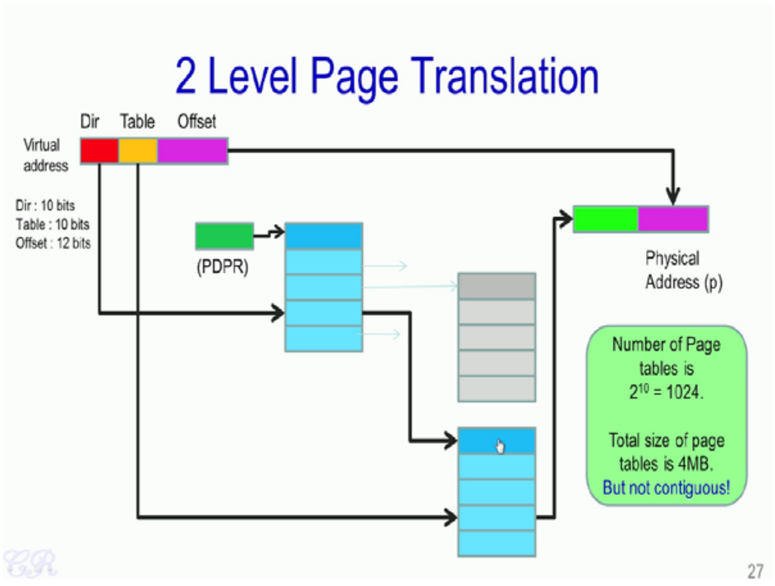
And, this is typically divided into two parts; the table index and the offset, which is of 20 bits and 12 bits respectively. So, how did we get this particular thing is by the fact that each page frame as we had mentioned in the earlier video is of size 4 kilobytes. Thus, to access any offset within the page frame would require 12 bits. 12 bits; because 2 power

12 is 4096, which is 4 kilobytes. And therefore, we have an offset which is of 12 bits, which essentially would give the offset within a page. The remaining bits that is, 32 minus 12, which is 20 bits, would be used for the table index. Thus, the process page table would be, which is indexed by these 20 bits would have 2 power 20 entries.

Now, assuming that each entry is of 4 bytes, then the entire size of this page table is 4 megabytes. And, what essentially is required is that this page table has to be contiguous, that is, we need to have these 4 MB of process page table to be in contiguous memory locations.

So, why do we need it to be contiguous is essentially that the table index is added to the process page table pointer, in order to get the location of a particular block and the corresponding page frame. And therefore, it needs to be in contiguous memory. Thus, we see that each process that executes on the system will have the additional overhead of having a 4-megabyte process page table, while 4 MB is not a very large space with today's RAM sizes of 32 and 64 giga bytes. However, the requirement that it needs to be contiguous could, would be an issue.

(Refer Slide Time: 10:36)



So, what some systems do? Especially, the intel system is to have two level page translations. Essentially, instead of having just table and offset for the virtual address, now we have 3 components in the virtual address. We have a directory entry which is of 10 bits, a table entry which is also of 10 bits and the offset which is as usual is of 12 bits.

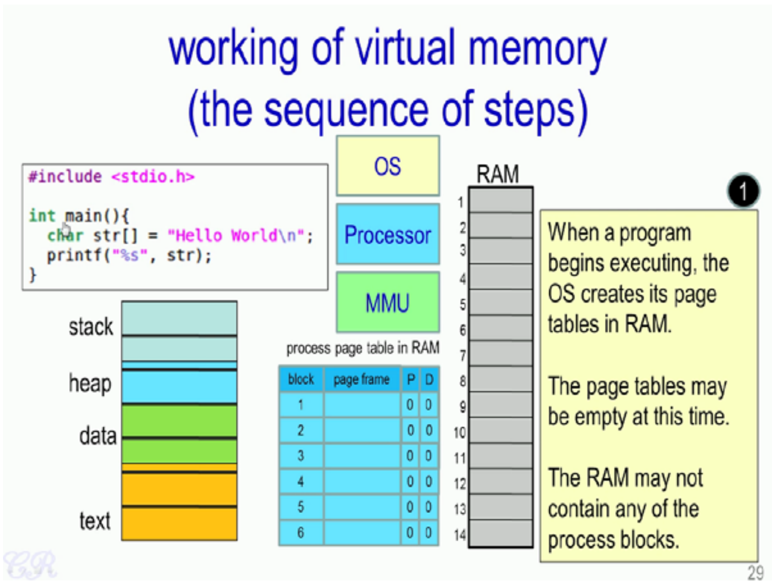
Now, corresponding to the virtual address what would happen is that the directory entry is used to index into something known as a page directory. Now, this page directory is pointed to by the page directory pointer register present in the MMU. Now, the contents of this entry in the page directory would point to a particular page table. So, this page table is of 4 kilobytes. And, this is also of 4 kilobytes; that is, the directories also of 4 kilobytes.

Next, the table entry is used to index into this particular 4 kilobyte table, and from there, we obtain the first part of the physical address. And as it shows, the second part of the physical address is taken from the offset. So, how does the scheme actually help us?

Now, what we see is that the number of page tables that can be present, that is the number of these, is 2 power 10 or 1024. And, how do we get 1024 is that we have a 10-bit directory entry over here. And, 2 power 10 is 1024. Thus, the directory page table is of 4 kilobytes having 1024 entries. So, 2 power 10 entries. Since each entry points to a different table, thus we can have utmost 2 power 10 different page tables. Each entry in the directory would point to a different page table. Now, how is the scheme actually helping us?

In total, although we still have 4 megabytes of page tables that is required, but the advantage we get is that they are not contiguous. So, we just need to have chunks of 4 kilo bytes, which need to be contiguous. And, this is easily obtained because each frame in the RAM is of 4 kilobytes. So, the two level page translations would allow us to have non-contiguous page tables present.

(Refer Slide Time: 13:48)



working of virtual memory (the sequence of steps)

```
#include <stdio.h>

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

process page table in RAM

block	page frame	P	D
1		0	0
2		0	0
3		0	0
4		0	0
5		0	0
6		0	0

1

When a program begins executing, the OS creates its page tables in RAM.

The page tables may be empty at this time.

The RAM may not contain any of the process blocks.

29

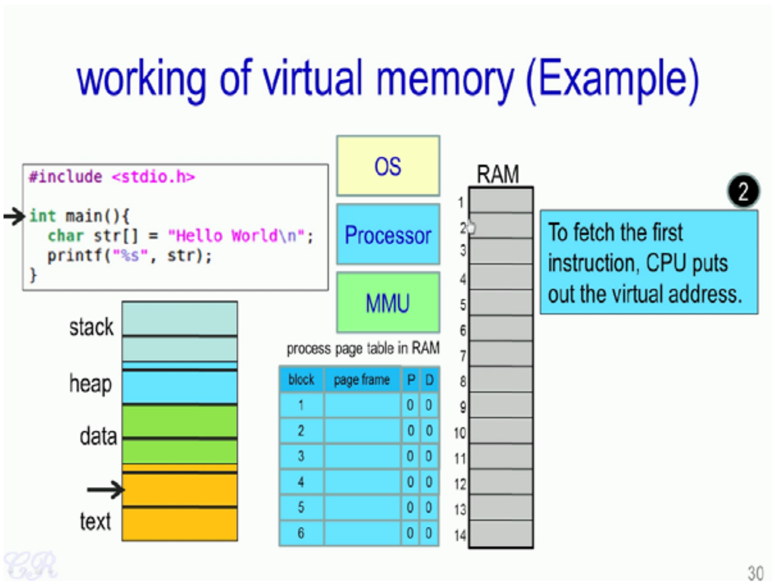
Next, we will look at the sequence of steps that occur during the virtual memory mapping. So, let us start with the particular program. And, this is its virtual address space with different colors, corresponding to each section of the program. Also, we have the various blocks. And as we have mentioned, each of these blocks is of 4 kilo bytes.

Then, what we have seen is that also there is a process page table present in the RAM, and there is the RAM itself which is again split into equal size blocks known as the page frames. Now, there are actually three entities which are involved with the working of the virtual memory. That is the operating system, which is the software component of this while the processor and the MMU, the memory management unit are the hardware components.

Let us say that we have started to execute this program. And, this particular line in the program is the first instruction being executed. Now, let us see what is the sequence of steps that occur as the program executes. So, essentially when the user runs this particular program, it triggers the operating system to create this particular process page table in RAM. So, ideally this particular page table will have the present bits set to all 0. And, the other parts of the page table may or may not be empty at this particular point in time. Third, the RAM may or may not contain any of the process blocks corresponding to the newly created process.

Next, what happens is that the operating system would transfer control into the main function of the program.

(Refer Slide Time: 15:50)



working of virtual memory (Example)

```
#include <stdio.h>

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

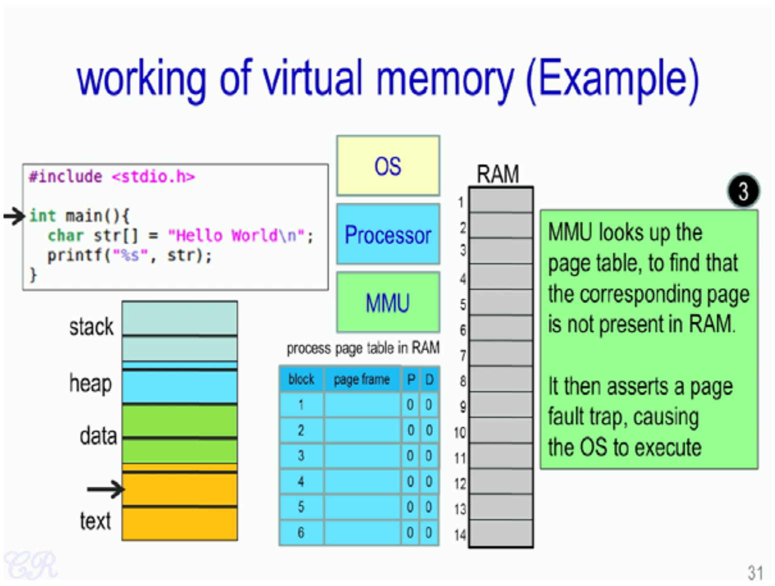
process page table in RAM

block	page frame	P	D
1		0	0
2		0	0
3		0	0
4		0	0
5		0	0
6		0	0

2 To fetch the first instruction, CPU puts out the virtual address.

This main function in the C code would probably point to a particular instruction in the virtual address space. So, this would result in the processor sending a particular virtual address corresponding to the main of function. So, this virtual address would be sent on to the bus, and this virtual address is then intercepted by the memory management unit.

(Refer Slide Time: 16:27)



working of virtual memory (Example)

```
#include <stdio.h>

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

process page table in RAM

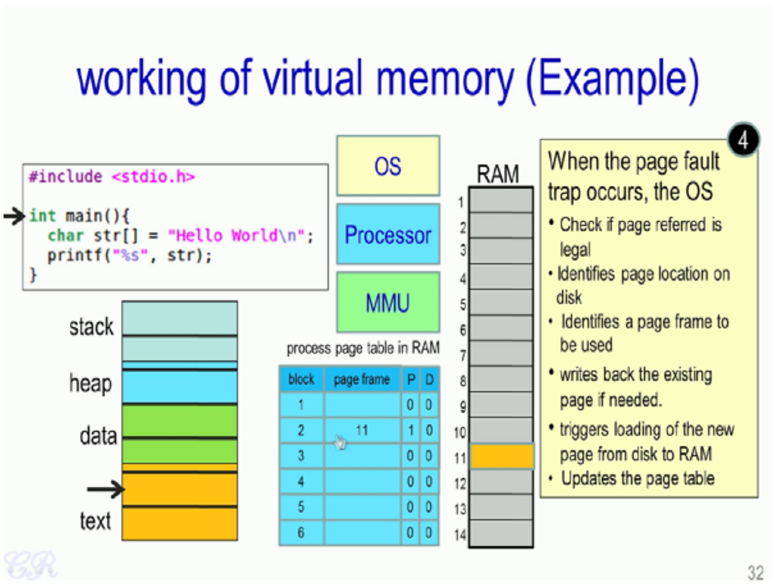
block	page frame	P	D
1		0	0
2		0	0
3		0	0
4		0	0
5		0	0
6		0	0

3 MMU looks up the page table, to find that the corresponding page is not present in RAM.

It then asserts a page fault trap, causing the OS to execute

And, the MMU would then look into the page table of the process and determine that that corresponding page or that corresponding block is not present in the RAM. So, this is determined by the 0 bit present in this particular bit. So, that is determined by a 0 in the present bit. So as a result of this, what the MMU is going to do is that it is going to cause a page fault to occur. So when a page fault occurs, it is going to trigger the operating system to execute.

(Refer Slide Time: 17:12)



working of virtual memory (Example)

```
#include <stdio.h>

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

process page table in RAM

block	page frame	P	D
1		0	0
2	11	1	0
3		0	0
4		0	0
5		0	0
6		0	0

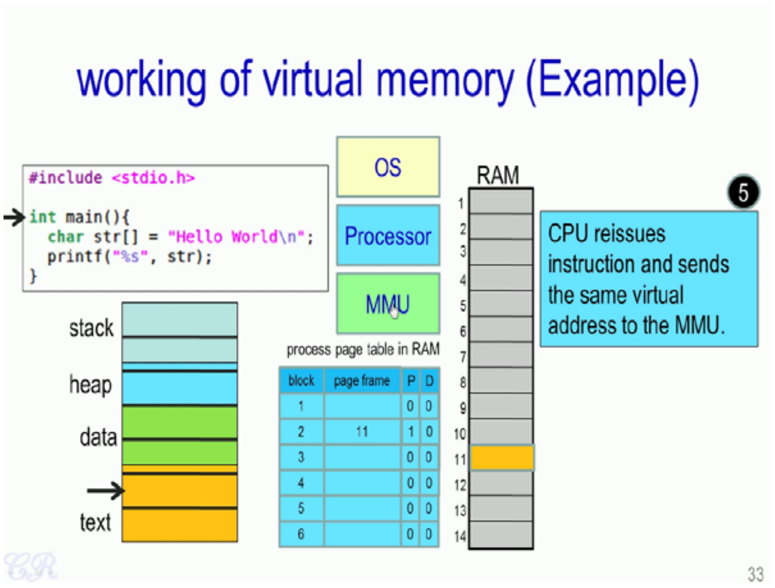
When the page fault trap occurs, the OS

- Check if page referred is legal
- Identifies page location on disk
- Identifies a page frame to be used
- writes back the existing page if needed.
- triggers loading of the new page from disk to RAM
- Updates the page table

Then, the operating system is going to execute and determine the cause of the page fault. So, essentially there is a sequence of steps that the operating system does whenever it occurs; whenever the page fault occurs. So, these are; that it will first check if the page referred is legal, and only if it is legal it will continue. Then, it will identify the page location on the disk corresponding to this particular block. Then, it will identify a page frame took in the RAM that need to be used. Then, essentially if it requires that a swap out process is needed.

And if the dirty bit is set then, the page which was previously present in the RAM would be returned back into the swap space. While this is followed by the block corresponding to this, a virtual address being accessed loaded from the disk into the RAM. So, this essentially is done by what is known as a DMA transfer from the hard disk to the RAM. And, the operating system would just need to trigger this particular DMA transfer to be initiated. And also, it is going to update the page table. So, essentially it is going to say that the first of all, the present bit is set to 1 and the page frame corresponding to block number two will be set to 11.

(Refer Slide Time: 19:02)



working of virtual memory (Example)

```
#include <stdio.h>

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

process page table in RAM

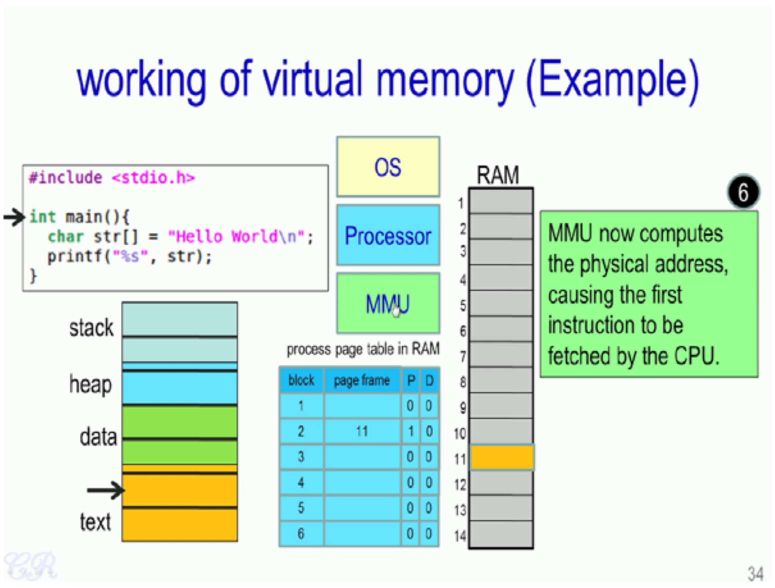
block	page frame	P	D
1		0	0
2	11	1	0
3		0	0
4		0	0
5		0	0
6		0	0

5 CPU reissues instruction and sends the same virtual address to the MMU.

33

Then, after the page gets loaded into the RAM, the transfer comes back to the CPU or the processor. And, the processor would reissue the instruction. So, this instruction would mean that there is a virtual address being sent on the processor bus, and it would cause the MMU to convert that virtual address into the corresponding physical address.

(Refer Slide Time: 19:30)



working of virtual memory (Example)

```
#include <stdio.h>

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

process page table in RAM

block	page frame	P	D
1		0	0
2	11	1	0
3		0	0
4		0	0
5		0	0
6		0	0

6 MMU now computes the physical address, causing the first instruction to be fetched by the CPU.

34

Now what the MMU is going to see is that it is going to look into the process page table, is going to see that the present bit is set to 1. Therefore, it would mean that the block is loaded into a page frame in the RAM. And, it would then be able to convert the virtual address to the physical address and cause the instruction to be loaded from the RAM into the processor. And, in this way the processor executes an instruction.

This is the sequence of steps that occur with the virtual addressing scheme. Essentially, it not only involves just the operating system, but there is a inter working between the operating system, the processor and the memory management unit, in order to achieve virtual memory. In other words, in order to make the virtual memory begin to work.

Thank you.