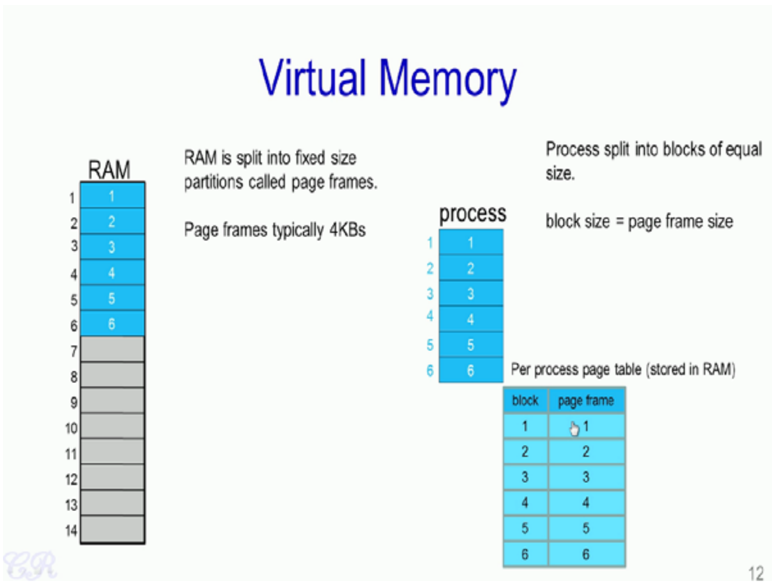
Introduction to Operating Systems

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

Week – 02
Lecture – 06
Virtual Memory

Hello. In this video, we will look at Virtual Memory; which is by far the most commonly used memory management techniques in systems these days.

(Refer Slide Time: 00:31)


Virtual Memory

RAM is split into fixed size partitions called page frames.

Page frames typically 4KBs

Process split into blocks of equal size.

block size = page frame size

Per process page table (stored in RAM)

block	page frame
1	1
2	2
3	3
4	4
5	5
6	6

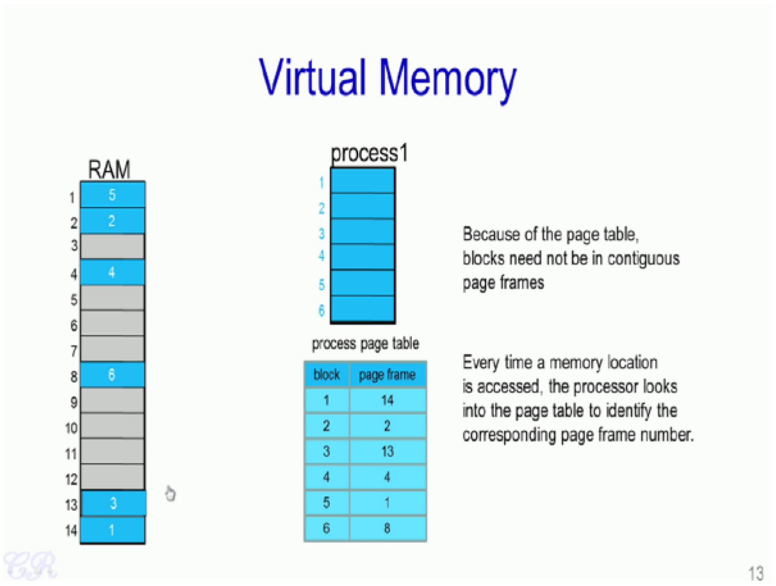
In a virtual memory system, the entire RAM which is present in the system is split into equal sized partitions called Page frames. So typically, page frames would be of 4 kilobytes each. So in this RAM, for instance, we have 14 page frames; which are numbered 1 to 14. And, each of these page frames have the same size.

In the previous Intel processor, all pages were fixed at 4 kilobytes. But, in more recent processes we could have pages or page frames, which are of a larger size. In a similar way, the process which executes in the CPU or the process map corresponding to the process is also split into equal sized blocks. Now, the split of a process is in such a way that the block size of a process, that is, this size is equal to the page frame size; that is the size of each block in this process is equal to the page frame size.

Now because we have split the RAM like this, as well as the processor's memory in a similar way, what we can then do is allocate blocks in a process to page frames in the RAM. Additionally, what the operating system maintains is a table. This table is also present in the RAM. And, not shown over here. But, what it contains is the mapping from the process blocks to the page frames. for instance, over here the mapping is very simple. Block 1 of the process gets mapped into page frame number 1, that is, block 1 of the process gets mapped into page frame number 1, block 2 gets mapped to page frame 2, block 3 gets mapped to page frame 3, and so on.

Now this, I would like to highlight again; is a third process page table. So, third process means that every process running on the system will have a similar page table as shown over here.

(Refer Slide Time: 03:23)



Virtual Memory

Because of the page table, blocks need not be in contiguous page frames

process page table

block	page frame
1	14
2	2
3	13
4	4
5	1
6	8

Every time a memory location is accessed, the processor looks into the page table to identify the corresponding page frame number.

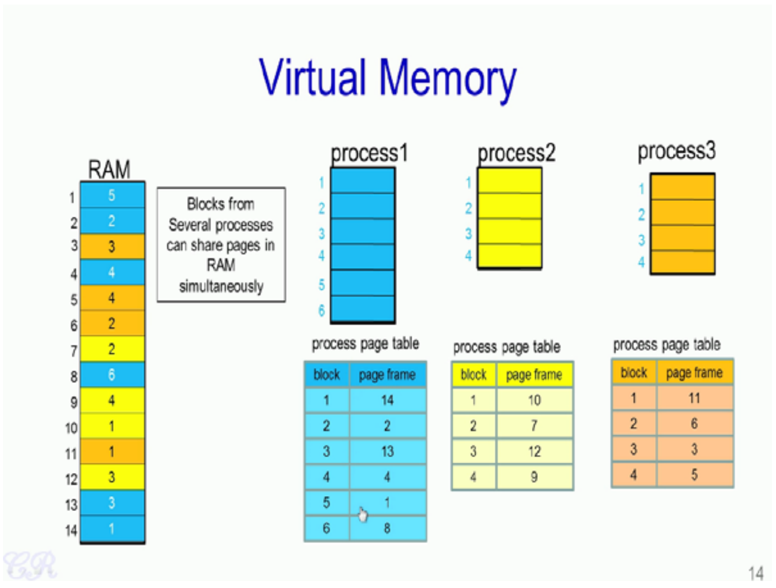
Now because we have such a table which provides the mapping between blocks of a process to the corresponding page frame, what we can then do is we could have any kind of mapping that we choose. For instance, now we have block 1 of the process present in block 14. That is over here. Block 2 of the process present in page frame 2, block 3 of the process in page frame 13 and so on.

Essentially, what we are able to achieve with this particular process page table is that the blocks of the process need not be in contiguous page frames in the RAM. However, the overhead that we incur is that every time a particular memory location needs to be accessed in the processor memory map, the CPU would need to look up the processor page table obtain the corresponding page frame number for that particular address. And, only then can the RAM be addressed or accessed. For instance, let us say that the program executing in the CPU accesses a memory location inside the block 3 of the process. So, this could be an instruction or a load or store to some data.

Now, when such an operation is executed a memory management unit present in the processor would intercept this access. And then, it is going to look up in to the page table and find that the corresponding page frame, which towards block number 3 is 13. Then, it is going to generate something known as a physical address, which will then look into the thirteenth page frame in RAM. As a result, every memory access has the additional overhead of looking into the page table, before the access into RAM can be made possible.

This look up into the page table is the extra overhead. And, typically this overhead is partially mitigated by using something known as a TLB cache or a translational look aside buffer cache. So, we will not go into details about the TLB cache. But, for our understanding with respect to the course we are studying, we need to remember that every memory access or every load or store of instructions fetched by the process during its execution would need to look up a particular page table. And, only then can the RAM be accessed.

(Refer Slide Time: 06:29)



Virtual Memory

process page table

block	page frame
1	14
2	2
3	13
4	4
5	1
6	8

process page table

block	page frame
1	10
2	7
3	12
4	9

process page table

block	page frame
1	11
2	6
3	3
4	5

Now, what we mentioned was that every process executing in the system would have its own process page table. Thus, if you have a second process, that is process 2, a process 2 will also have associated page table; process 3 will also have its associated page table. Now, these processes or these processes as we have seen the memory associated with these processes are present in the user space region of the memory. However, the process page tables are present in the (Refer Time: 07:09) space. And therefore, any program you write in the user space, but not be able to determine what the process page table mappings are.

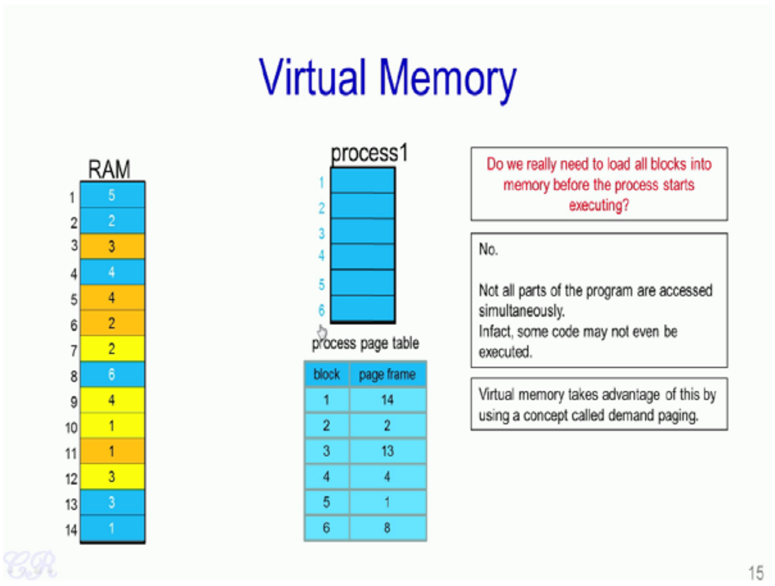
Now, since we have multiple processes executing in the system, each process could have various page frame mappings. For instance, process 1 has a mapping of all the blue page frames in this particular RAM; while process 2 has all the yellow page frames; when the third, process three has all the orange page frames.

Now, what we can see is that we are capable of sharing the RAM with multiple processes simultaneously. Now, assume that we have a system with a single CPU. this means that at a particular point in time, only a single process will execute. Now, this process will continue to execute until its time slice completes.

During the execution of the process, that is when for instance, process 1 is executing in the CPU, this particular process page table will be the active page table. Therefore, any instruction or data which is loaded or stored will have, will look up this particular process page table to get the corresponding page frame in RAM. when there is a context switch from process one to say process 2, it would be this processor’s page table, which will become active.

So, any address that is accessed in process two will get the corresponding page frame number from this active page frame. Similarly, when process 3 gets executed in the CPU, this processor’s page table will be the active page table. So, what we can see is that it is not possible for process 2 to access any of the page frames corresponding to process 1 or process 3.

(Refer Slide Time: 09:37)



Virtual Memory

Do we really need to load all blocks into memory before the process starts executing?

No.

Not all parts of the program are accessed simultaneously.
Infact, some code may not even be executed.

Virtual memory takes advantage of this by using a concept called demand paging.

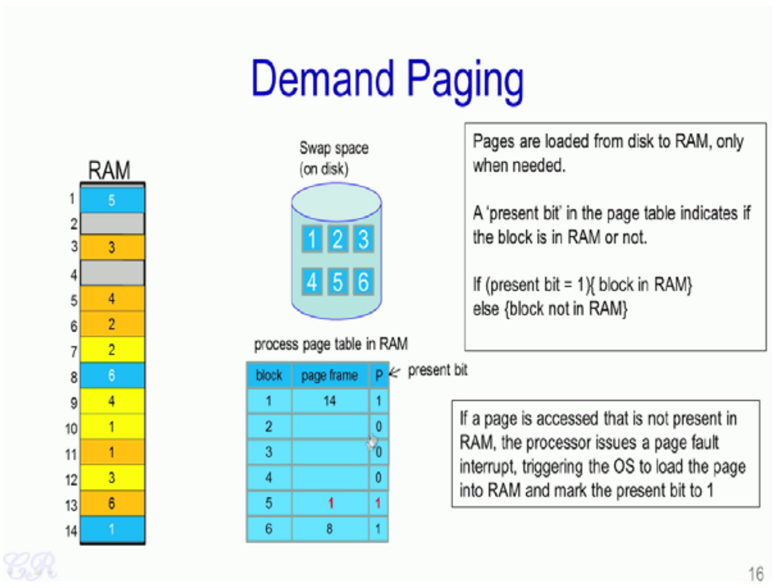
block	page frame
1	14
2	2
3	13
4	4
5	1
6	8

The next thing we would look at is that do we really need to load all blocks into memory before the process starts to execute. So, in the earlier video we had seen that when a process executes, the entire process was loaded into a RAM. So, now we ask ourselves that is such a loading actually required? And, the answer is no. Essentially, what we notice about programs is that not all parts of the programs are access simultaneously.

In other words, there is a locality aspect in the way a particular program executes. What this means is that if some instructions are executed in let us say this particular block, it is highly likely that the next set of instructions or the next few instructions that follow this would also be present in this block. So, once there is an instruction to another block, say block 3, it is quite likely that because of the locality of the program, there would be quite a few instructions executed from this block and so on.

In fact, you may have, and it is quite often the case that there may be several parts of the process memory, which are not even accessed at all. That is, they are neither executed, loaded or stored from memory. So, what the virtual memory concept does is that it takes advantage of this locality aspect during the execution by using a concept known as demand paging.

(Refer Slide Time: 11:43)



Demand Paging

Pages are loaded from disk to RAM, only when needed.

A 'present bit' in the page table indicates if the block is in RAM or not.

If (present bit = 1){ block in RAM}
else {block not in RAM}

process page table in RAM

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

If a page is accessed that is not present in RAM, the processor issues a page fault interrupt, triggering the OS to load the page into RAM and mark the present bit to 1

16

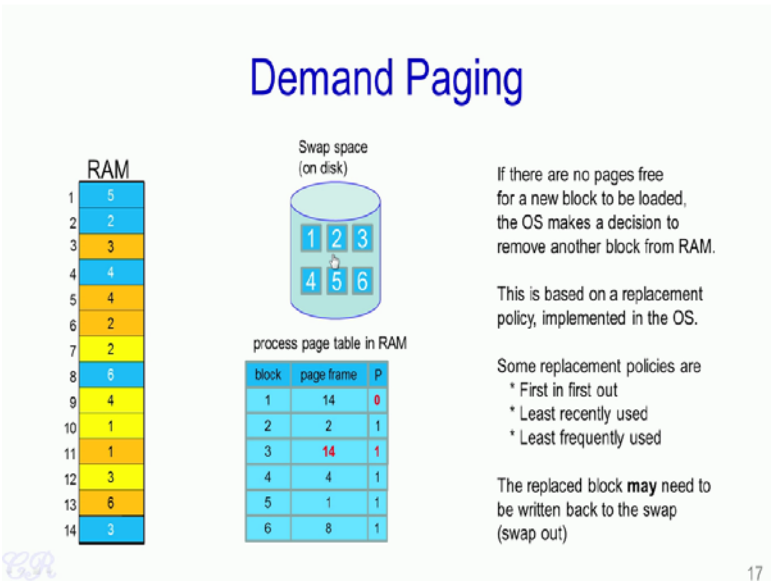
In a demand paging scheme, what would happen is that we would have on a secondary storage device like a hard disk, there would be a particular space are located as the swap space. So, in this swap space all blocks of the executable will be present. So, for instance if the process has 6 blocks, 1 to 6, and all the 6 blocks will be present in the swap space. And only on demand, will blocks be loaded from the swap space into the RAM.

Now let us assume that there is an instruction which is executing in page frame 8 that corresponds to block 6 of the process. And, it has resulted in a load or a store to a particular memory location in block number 5. So, what happens is that the processor is going to look up the process page table corresponding to block number 5 and see that the present bit is set to 0. So, this would result in something known as a page fault trap, also called as the page fault interrupt.

Now, the page fault interrupt would then trigger the operating system to execute. Now, the operating system will then detect that this was a page fault interrupt. And, it will determine the cause of this page fault interrupt and it will load the corresponding page from the disk into RAM. Consequently, it will also modify the processor’s page table.

For instance, over here a value of one is added to the page frame number and the present bit is set to 1. Now, all later accesses to this particular block 5 will not cause a page fault because the present bit is set to 1. However, the first accesses to block 2, 3 and 4 would result in page faults. And, it would cause the operating system to execute and load the corresponding blocks into the RAM. Consequently, the page tables for that particular process will be updated.

(Refer Slide Time: 14:10)



Demand Paging

process page table in RAM

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

If there are no pages free for a new block to be loaded, the OS makes a decision to remove another block from RAM.

This is based on a replacement policy, implemented in the OS.

Some replacement policies are
* First in first out
* Least recently used
* Least frequently used

The replaced block **may** need to be written back to the swap (swap out)

Now we have reached a state where every page frame in the RAM occupies some block. So, these blocks could either be from the process one or process 2 or process 3. Now, what happens if let us say there is a memory access to process one's third block. So, as we can see that since the present bit is set to 0 in the process page table, it would result in a page fault interrupt that occurs. And, it would trigger the operating system to execute.

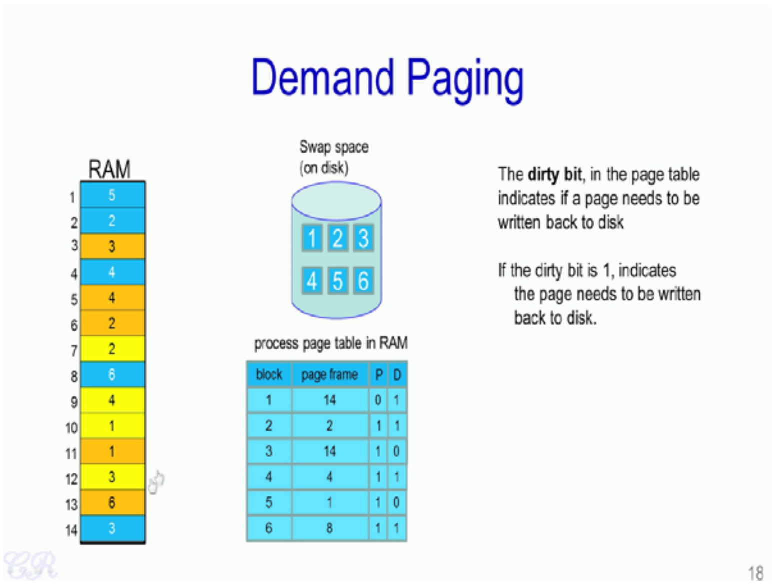
Now, the thing is what will the operating system do. So, in order to (Refer Time: 15:10) to this particular memory access in the third block, the operating system would need to remove one of these pages. Essentially, it needs to clear up one of these page frames in order to make way for the new block. So, the obvious decision that the operating system would need to make is which of these blocks should it remove, in order to make way for the new block.

So, this choice is based on some algorithm present in the operating system. So, these are known as the page replacement algorithm. And, there are quite a few choices for that; such as the First In First Out, Least Recently Used or Least Frequently Used. So, based on the decision that the replacement algorithm makes, the OS would swap out a block from the RAM and replace it with the third block that was recently accessed.

Consequently, the present bit corresponding to the block that has been removed will be set to 0. So, we had removed, for instance, the block number one from the RAM to the swap. And therefore, the present bit corresponding to block one will be set to 0. Now, block three which has just got loaded into the particular page frame will have its page frame number set and the present bit set to 1. So, this process of removing a block from the RAM and replacing it by another is known as the swap out and swap in, respectively.

The process of moving a block from the RAM to the disk is known as a swap out process; while, the process of loading a block from the disk to the RAM is known as swap in. Now, during the process of swap out, that is storing block one to the hard disk, all the changes that happened during the execution of the process, all changes that occurred in block 1 will get updated in the swap space. Thus, the block which eventually gets stored in the swap space will have the latest view of the data. So, the next thing that we are going to ask is that if the swap out, that is, the copying of the block 1 from the RAM to the disk is actually required.

(Refer Slide Time: 18:12)



Demand Paging

The **dirty bit**, in the page table indicates if a page needs to be written back to disk

If the dirty bit is 1, indicates the page needs to be written back to disk.

process page table in RAM

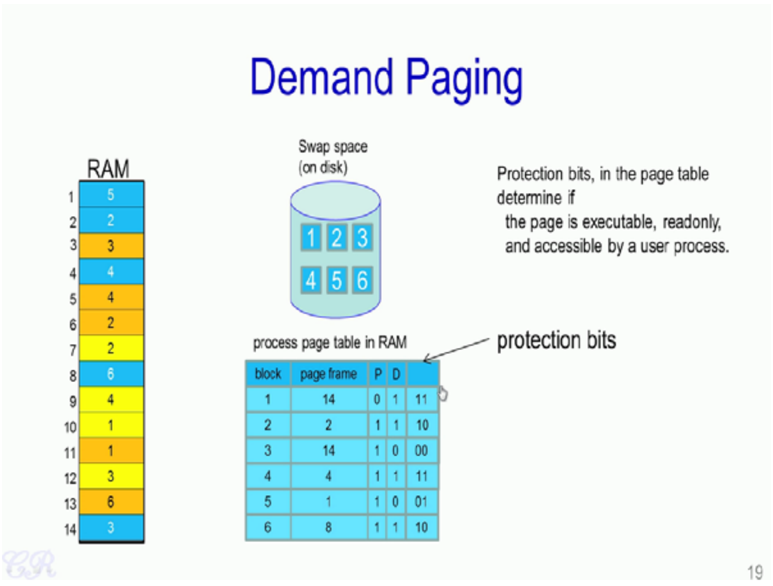
block	page frame	P	D
1	14	0	1
2	2	1	1
3	14	1	0
4	4	1	1
5	1	1	0
6	8	1	1

Let us assume that block one, which was present over in the fourteenth frame had no data that is changed. For instance, it could be corresponding to instructions or read only data. And therefore, none of this data actually changes. In such a case, the swap out process of copying the contents of this page frame back to the disk will not be required. On the other hand, suppose we had data in block number 1 which was modified, so this would indicate that the copy of block 1 in the RAM is different from that in the disk. And therefore, the entire page frame should be swapped out and return back into the disk.

So, how will the operating system know whether a swap out is required or not? So, this is done by an additional bit, which is present in the page table known as the D-bit or the dirty bit. So, the dirty bit essentially indicates whether the contents of a block in the RAM has been modified with respect to its contents in the disk or in the swap space.

So, if the dirty bit is set to 1, it indicates that indeed there is a difference between the contents in the RAM corresponding to the contents in the swap space, and therefore during a page fault, the contents of the page frame will be returned back to RAM. If the dirty bit has a value of 0, then the entire swap out process will not be required. It will just require to swap in the new process and replace that corresponding block in that corresponding page frame.

(Refer Slide Time: 20:19)



Demand Paging

Protection bits, in the page table determine if
the page is executable, readonly, and accessible by a user process.

process page table in RAM

block	page frame	P	D	
1	14	0	1	11
2	2	1	1	10
3	14	1	0	00
4	4	1	1	11
5	1	1	0	01
6	8	1	1	10

In addition to the p and d bits, that is the present and dirty bits, the page table for a process has some additional bits known as the protection bits. So, these protection bits would determine various attributes for a particular block. For instance, the process of and the operating system could set various blocks as executable; which means that the data present in that block corresponds to executable code.

And therefore, the CPU could then execute it. Other bits could also tell whether the block is read only. And therefore, modifying of any data in that block will lead to a fault or an error. And, bits over here will also determine whether a particular block corresponds to an operating system code or it is a regular user code. So, this additional information is present by the protection bits.

Thank you.