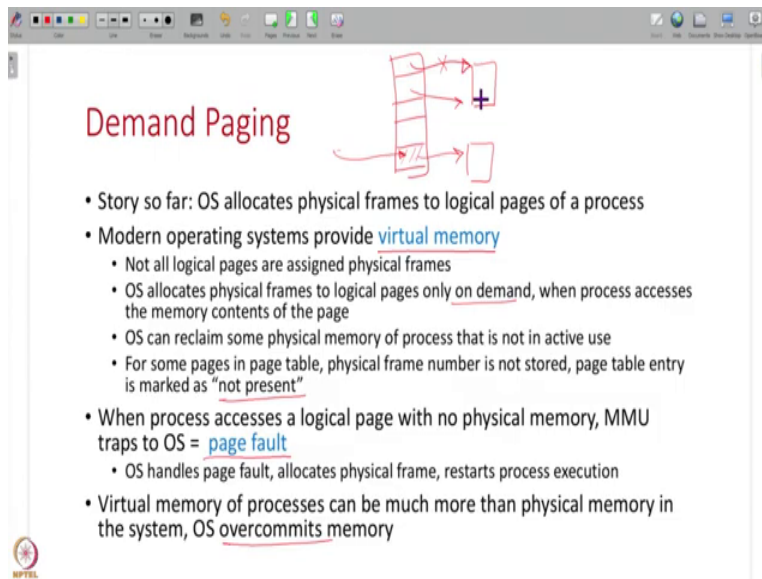


Design and Engineering of Computer Systems
Professor. Mythili Vutukuru
Computer Science and Engineering
Indian Institute of Technology, Bombay
Lecture 13
Demand Paging

Hello everyone, welcome to the thirteenth lecture in the course Design and Engineering of Computer Systems. In this lecture, we are going to continue our discussion on memory management and paging. And we will study a topic called demand paging in this lecture. So, let us get started.

(Refer Slide Time: 00:36)



The slide is titled "Demand Paging" in red text. To the right of the title is a hand-drawn diagram in red ink showing a vertical stack of five boxes representing logical pages. Arrows point from the first, third, and fifth boxes to a single box on the right, which contains a plus sign (+), representing physical frames. The second and fourth boxes in the stack have an 'X' over them, indicating they are not currently mapped to physical frames. Below the diagram is a list of bullet points:

- Story so far: OS allocates physical frames to logical pages of a process
- Modern operating systems provide virtual memory
 - Not all logical pages are assigned physical frames
 - OS allocates physical frames to logical pages only on demand, when process accesses the memory contents of the page
 - OS can reclaim some physical memory of process that is not in active use
 - For some pages in page table, physical frame number is not stored, page table entry is marked as "not present"
- When process accesses a logical page with no physical memory, MMU traps to OS = page fault
 - OS handles page fault, allocates physical frame, restarts process execution
- Virtual memory of processes can be much more than physical memory in the system, OS overcommits memory

So, the mental model we have so far the story we have seen so far is, every process has a virtual address space that is divided into fixed size chunks called pages and for every page to store the contents of this page to store the contents of these virtual addresses, the OS will allocate a physical frame for every page of the process and this mapping is kept track of in the page table. That is what we have seen so far. But the actual truth is what we have seen so far is only an approximation, it is not exactly correct.

The truth is that the operating system does not allocate memory for every page of the process, it only allocates memory on demand that is, the operating systems provide you with what is called virtual memory. So, you can say that I have these addresses at this page, there is code, data, stack, heap everything you can say that all of these addresses are assigned to you assigned to

your program. But until you use them, the OS may not allocate actual physical memory for you until the CPU is running that code you may not actually get physical memory allocated.

So, the OS allocates physical frames to pages only on demand when you access that memory contents until then you may not be given a physical frame. And also, if you are not using some physical memory, the OS can also take it away from you even though you think there is code data stack keep something stored in that page, actual physical memory may not be given to you it can be taken away even after giving it to you. So, this concept is called virtual memory.

And therefore, in your page table, some for some pages, even though the page is valid and there is valid data, the programmer thinks there is valid data stored in that page, the physical frame number may not be allocated and the page can be marked does not present in RAM until you use it. Of course, if you use it once you begin using it CPU says get me this data then, of course, it has to be in RAM, but until then it may not be in RAM and the page table entry can be marked as not present.

So, when such pages are accessed, the MMU will trap to the OS, because there is no physical frame number MMU cannot translate, there will be a page fault, this particular trap is called a page fault. And then the OS will allocate a page, suppose you think you have your some data stored in this page over here, your page table entry will mark this page as valid, you think these virtual addresses you can use, but there will be no physical memory allocated corresponding to this.

When you access this page, when the CPU accesses some virtual address here, it will trap to the OS, then the OS will say if you are using it, fine, I will give you the physical memory, you can put the contents inside it and then the translation can happen. So, this memory allocation is done on demand. Because of this, some of the virtual memories of all the processes in the system can actually be more than the physical RAM you have the OS overcommits its memory.

This allows the OS to run many more processes, then there is actual memory in the system. And this is because every process is only using a small amount of memory at a time, you are executing some code over here, then all of these other physical frames, you do not need them if you are not using them. So, this is an optimization that modern operating systems do in order to

support many more processes over a limited amount of RAM. So, how is this memory allocation done on demand? Let us understand that in more detail.

(Refer Slide Time: 04:17)

The slide is titled "File-backed vs. anonymous pages" in red. To the right of the title is a diagram showing a vertical stack of memory pages. The top page is labeled "code" and has an arrow pointing to it from the word "code" in red. The bottom page is labeled "stack" and has an arrow pointing to it from the word "stack" in red. A red circle with a plus sign is drawn around the "stack" page, and an arrow labeled "swap" points from it to a box labeled "swap" in red. The slide contains the following bullet points:

- Pages in the memory image of a process are of two types
 - File-backed pages contain data from files on disk (e.g., page containing executable code)
 - Anonymous pages are not backed by files on disk (e.g., pages containing stack, heap)
- On-demand allocation of file backed pages
 - Can be fetched from disk into empty frame only when accessed for first time
 - Physical frame can be reclaimed when not in use by process (copy exists on disk)
- On-demand allocation of anonymous pages
 - Empty physical frame can be allocated when page is used for first time
 - Once page is modified, cannot simply reclaim physical frame (data can be lost)
- Swap space: space on hard disk used to store copies of modified "dirty" anonymous pages (different from file storage)
 - Dirty anonymous pages are written to swap space when not in use by process, read back from swap into main memory when required

So, before that, to understand that we need to classify the pages in the memory image of a process into two types. If you look at all your pages in the memory image, they are of one of two types. One set of pages are what are called file backed pages. That is these pages contain data from files on disk. For example, if you have a page containing your code of the executable. So, this data that is there in this page is already also there on the disk, you will simply copy the disk contents into this page.

And on the other hand, there are some pages which do not have any copy on disk, for example, the pages of your stack, the stack you are using at runtime, as you are running, you are pushing arguments, you are popping arguments, it does not correspond to any file on disk, such pages are called Anonymous pages. So, in your memory image, some pages are file backed pages, which are your executable code your language library code operating system code.

There are many such pages that can be filed back and there are some pages that are anonymous that are not filed back. Now, for these two kinds of pages, the on demand allocation will be different. So, for file pack pages, how can on demand allocation happen? Suppose you have created a process and there is some this page of the processes file backed it actually corresponds to the code in the executable.

So, when the process is created, the OS will create this virtual address space that will create a page table entry saying, from this address 0 to some 4 KB, this is your code, the OS will tell you that and the CPU will start to execute this code program counter points to some address over here, all of that will happen, but until the CPU says give me the instruction at this location, you will not actually allocate a physical memory frame for this page.

So, when the CPU says give me this code, then you will copy it from the disk into some free physical frame and then you will add a page table entry from this page number to this frame number. So, you will fetch from disk only when that code is accessed for the first time not when the process is created.

When the process is created itself, you do not give it a bunch of physical frames and copy all the code and data into it only when the CPU starts to run out it says give me an instruction at this address, then you will quickly find a free physical frame copy the instructions into that frame at this page table entry and then the CPU will access it. So, file back pages can be fetched from disk on demand.

And if the OS is running out of memory, it wants you to reclaim some memory take away some physical memory from you then what is there it can simply take away this physical frame, the contents are there on disk anyways it will remove this page table mapping it will mark this page is not present in RAM and you are done the physical frame can be reclaimed easily when it is not in use by the process.

And the next time the process wants to run this code again once again you will allocate a physical frame to create this page table mapping. So, that is on demand allocation for file back pages. Now, what about anonymous pages suppose you have a stack or something here it has some data on it that is not there on disk. Now, when you access the stack for the first time, then you will allocate an empty page you will not give an empty page for the stack when the process is created. Instead, when it starts to actually use the stack then you will give it an empty page.

Now once later on, once you have given the stack, you have given a physical frame to the stack, there is some contents are stored here. Now, if you want to reclaim this physical frame, the OS wants to take away this physical memory then you cannot simply take it away there is some data in this ram it will be lost what do you do.

So, once you have a modified or also called a dirty anonymous page, it cannot simply be reclaimed or as cannot take away the memory then you will lose the data. So, in such cases, if the OS wants to take away some of the physical memory given to you it can store a copy of your memory frame on a special area of disk called the swap space. That is you have certain physical frames assigned to you that are storing some anonymous pages contents of some anonymous pages that are not there anywhere else on disk.

And what the OS wants to free up this physical memory in such cases, this content in this memory physical frame will be written to disk into the swap space. And now this memory frame is free, it can be erased, this content can be erased given to store some other content given to some other processor and so on.

So, dirty anonymous pages are returned to the swap space when the OS wants to reclaim the memory. And when again, when the process accesses the stack. Again, you want the contents here, then the OS will give it back to you again, it will read from swap space and give you your memory page again. In this way on demand allocation is done.

(Refer Slide Time: 09:35)

Information in page table entry

Handwritten diagram: A box labeled 'PTE' containing a box labeled 'PFN' followed by several vertical bars representing bits.

- Page table entry maps page number to physical frame number
- Page table entry also contains several bits to indicate status of page
 - Valid bit indicates if the page is in use by process in its virtual address space (invalid addresses should never be accessed by a process)
 - Present bit indicates if a physical frame number is assigned to the logical page
- Process accesses page with
 - Valid bit not set → illegal memory access (segmentation fault)
 - Valid bit set, present bit not set → OS has not allocated memory yet
- Other bits set by MMU to indicate usage of page to OS (since OS does not know of every memory access)
 - Whenever a process writes to a page, MMU sets dirty bit
 - Whenever a process accesses a page, MMU sets accessed bit

Handwritten diagram on the right: A vertical stack of boxes representing a page table. An arrow points from one box to another, and a question mark is next to the top box.

So, now we have to keep track of all of this. We have to know that is this page does this have corresponding physical memory or not has the OS yet had done the allocation or not? You have to keep track of all this information so you have some extra bits in your page table entry that will help you keep track of all of these things. So, the page table entry if you look at the page table

entry, one single page entry a page table has many such page table entries in an array, if you look at one page table entry, it will have the it will have some space for the physical frame number and it will also have a few extra bits of information.

What are these bits? One bit is the valid bit which indicates if this virtual addresses in this page are in use by the process or not if the process is storing anything code data stack heap, anything in those virtual addresses, then those virtual addresses are in use the valid bit is set otherwise the valid bit is not set. Then there is the other bit, which is the present bit that is even for pages that are assigned valid data even for valid pages, you may not actually assign a free physical frame yet.

So, the present bit indicates if a physical frame number is assigned to the page or not note that these are two different things. So, if a process accesses a page, where the valid bit itself is not set, this means that you have some say you have your heap all of these addresses are there in your heap and this is the end of the heap and you have access to an address here beyond your array you have accessed for such pages the valid bit itself is not set that is an illegal memory access like a segmentation fault.

On the other hand, for some valid pages, there may not be any physical memory allocated yet. So, for such pages, the valid bit is set, but the present bit is not set in such cases also the memory address translation cannot be done MMU traps to the OS in such cases, this is not the fault of the programmer, the programmer is accessing correct memory, but the OS has not yet allocated physical memory here in such cases the OS will allocate memory on demand.

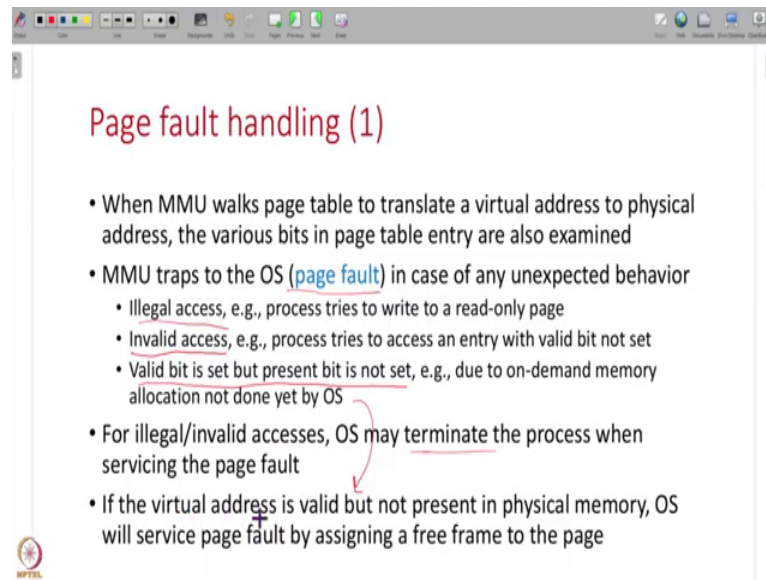
So, such pages when they are accessed, the OS has not yet allocated memory and it will allocate memory on demand. So, these two are different understand the difference between a valid bit which indicates all the virtual addresses that the process thinks it can access and the present bit, which indicates if actual physical memory is there for these valid addresses or not. So, with demand paging, these two are different the set of valid pages and the present pages are different.

So, there are also other bits in the page table in every page table entry that are set by the MMU, which indicate how the page is being used to the OS. Because the OS does not know whenever any time or memory pages access. So, the MMU will tell some extra information to the OS for example, if a page has been written to the MMU sets a dirty bit, if the page has just been

accessed read or write the MMU sets the access bit with this the OS will know or this is, an anonymous page this is a dirty anonymous page, this is an unmodified page all of this information the OS can know.

So, which is why these bits are set by the MMU. So, these are all the extra information in addition to the frame number that is present in the page table entry.

(Refer Slide Time: 13:15)



Page fault handling (1)

- When MMU walks page table to translate a virtual address to physical address, the various bits in page table entry are also examined
- MMU traps to the OS (page fault) in case of any unexpected behavior
 - Illegal access, e.g., process tries to write to a read-only page
 - Invalid access, e.g., process tries to access an entry with valid bit not set
 - Valid bit is set but present bit is not set, e.g., due to on-demand memory allocation not done yet by OS
- For illegal/invalid accesses, OS may terminate the process when servicing the page fault
- If the virtual address is valid but not present in physical memory, OS will service page fault by assigning a free frame to the page

Now, let us see what happens on a page fault. So, what is a page fault anytime the MMU cannot translate an address, it raises a trap, the CPU executes the intent or the trap instruction jumps to OS code we have seen this before. So, when the MMU cannot translate, the virtual address to a physical address, then it raises a page fault it traps to the OS with a page fault. And this can happen for many reasons.

And what are those reasons one could be some kind of illegal access the processes accessing memory it should not be accessing for example, in user mode, you are trying to access a kernel OS virtual address you are trying to write to a read only page any such cases there will be illegal accesses, there could also be invalid accesses. The MMU looks at a page table entry and it finds that the valid bit is not set this virtual address is not even in used by the process, but somehow the CPU is requesting that virtual address then that is an invalid access.

Then sometimes the addresses are valid, the program thinks it has put some code data something in that memory address, but the OS has not yet allocated a free physical frame has not yet copied

that content into memory from disk or wherever in such cases MMU when it is translating the address it will find that the valid bit is set, but the present bit is not set. Because the memory allocation on demand allocation is not yet done by the OS.

These are all the possible reasons why MMU address translation can fail when it is walking the page table and it can trap to the OS for a page fault. So, how does the OS handle these cases if the OS if the program is doing any illegal or invalid access then the OS can simply terminate the process it can say, if there is a segmentation fault your program crashes. So, the OS can terminate the process.

But in this case, in this case, when it is not the fault of the programmer, a valid bit is set, but the present bit is not set, then the OS will service this page fault and try to allocate a free physical frame to the process on demand. So, now, let us understand this case in a little bit more detail.

(Refer Slide Time: 15:38)

Page fault handling (2)

- If page not assigned a frame, OS finds free physical frame to assign to logical page
 - OS maintains list of free physical frames to assign during page faults
 - If there are no free physical frames, OS can evict a victim page (i.e., take away a physical frame assigned to another page) to free up a physical frame
- Page replacement policy helps OS identify victim pages
- Reclaiming victim page involves writing victim page contents to swap space (in case of modified anonymous page) and deleting old page table mappings
- Next, OS populates contents of free physical frame with new page content
 - If page is in swap space or file-backed, contents are read from disk into free page
 - Reading the page contents from disk may block the process
- Once physical frame is ready with page content, OS updates page table mapping with new physical frame number, MMU updated, process execution restarted
- Minor page fault: physical frame already in memory (e.g., shared library being used by another process), just need to add page table mapping and restart

major

So, you have your virtual address space of a process, and for some CPU has requested some virtual address and this virtual address is not yet assigned any memory free by the OS. So, this is a valid virtual address, the program thinks it has some code or data or stack something over here, but actually no memory has been allocated or maybe memory was allocated by the OS in the past, but when it needed to free up some memory, it copied it into disk into the swap space.

And now there is no memory corresponding to this virtual address. In such cases, what the OS will do is when this page fault happens, it will find a free physical frame and it will give it to this

particular page. It will find a free physical frame to assign to this page. So, where will you get a free physical frame from the OS usually maintains a list of free physical frames, RAM, there are some empty locations in RAM that nobody is using, it will have a free list from that it will find some frame number and give it to this page.

Sometimes all of your memory is full, there is no free physical page at all, then what do you do? Suppose there is this physical frame that has been used by some other process, then the OS will take away a physical frame some from some other processes logical page and give it to this process that is called evicting a victim page. So, now some other process, its page contents, its logical page, certain virtual addresses are stored in this physical frame, in this frame.

This frame number is there in the page table of some of the process, the contents of some other processes are stored in this physical frame. The OS can take this away from that process, say, if this page has some modified anonymous data, then it can write it to the swap space. Or if it has file back data, it can simply erase the data, whatever it is, it will take away this physical frame from some other page of some other process and give it to this process to handle the page fault.

So, how do you pick this frame? Which poor guy will you make your victim, they will there will be a policy, every OS has a page replacement policy that tells the OS which is the best page to pick, we will see what these are in a little bit. So, the OS will run this, look up this page replacement policy find the victim page and free up this physical frame, it may have to write it to swap space if some other process has stored some anonymous modified anonymous content over here.

And then now that this frame is free, the OS will fill it with whatever data suppose this has to have the code of this process, then you will get it either from the file or from swap space. If this page, if it is an empty page, well and good you will just clean it up and give it to this process. Otherwise, if it had some content beforehand, then you may have to read it from disk read from disk. And initially, you may have to first write to the disk.

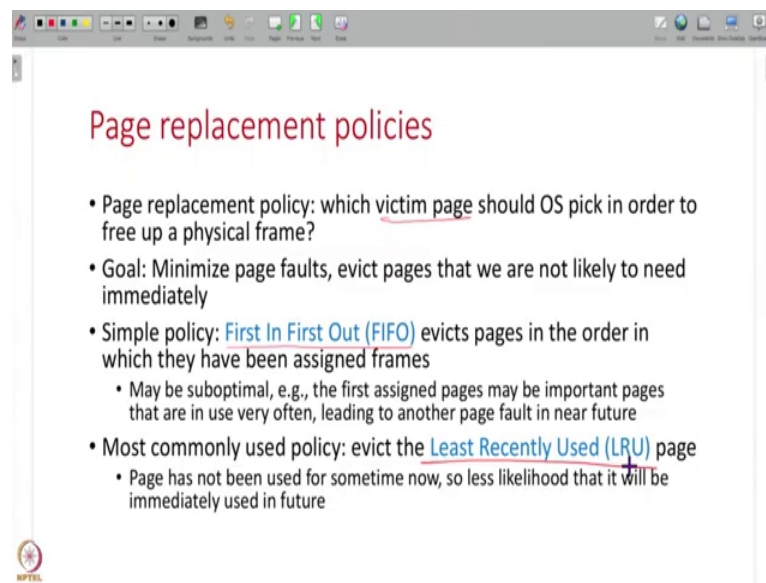
You will have to write this victim page contents either to swap space or somewhere and read some extra content into this physical frame and then add a page table entry from this virtual address to this physical address. Once the frame is ready with content, you update the page table

mapping and you tell the MMU hey look now please try this address translation again it will work because I have added this address translation in your page table it will work now.

And you will restart the process execution. So, this is called servicing a page fault. It involves identifying a free frame probably by evicting its contents to disks reading the contents of this page from disk if required, updating the page table mapping and restarting the process. Some page faults so all of this is what is called a major page fault. So, this is a major page fault. In contrast, sometimes you will have minor page faults, which is this frame is already there in memory. Maybe it is a shared library frame or a frame having OS content.

It is already there in memory, all you have to do is just add a new page table mapping to it that is all. You do not have to do anything like reading it from disk or something you just content is there you just have to add a page table mapping. So, such page faults are easier to fix they are called minor page faults. Otherwise, if it is the page is not there in memory already, then it is a major page fault.

(Refer Slide Time: 20:12)



Page replacement policies

- Page replacement policy: which victim page should OS pick in order to free up a physical frame?
- Goal: Minimize page faults, evict pages that we are not likely to need immediately
- Simple policy: First In First Out (FIFO) evicts pages in the order in which they have been assigned frames
 - May be suboptimal, e.g., the first assigned pages may be important pages that are in use very often, leading to another page fault in near future
- Most commonly used policy: evict the Least Recently Used (LRU) page
 - Page has not been used for sometime now, so less likelihood that it will be immediately used in future

So, now we have seen a page replacement policy, it should make this difficult decision of if this process this page, these virtual addresses need to be put into RAM. Which other process which other page, should I evict out of RAM in order to make space of course, if you have free memory, well and good, but if you do not have free memory, you have to pick some other victim page to evict.

So, which one will you pick these policies have to pick pages such that in the future, immediately, there would not be another page fault. Suppose if you pick a very popular page to evict immediately that other process whose page you evicted will again have a page fault. Again, you have to bring back the contents. So, you do not want to do that you want to evict pages that will not result in page once again, in the immediate future, you want to reduce the number of page faults.

So, there are many page replacement policies. For example, again, a very simple policy is the FIFO policy. You see whichever page was assigned a frame in the past in the order in which it was assigned in that order, I will evict. So, this process for this page, it started long back, it took a frame long back, hey, you have had enough of this frame, give it back to me, you can do that. But this may be suboptimal because that process could actually be a very active process, that page could be very heavily used.

So, if you take away its memory and put it in swap space, immediately it will have a page fault. You do not want that. So, a better a more sensible policy is what is called LRU policy or Least Recently Used policy. That is if a process has been assigned some memory and it has not used that memory has not touched that quarter data for a long time, then you can say that there is a lesser likelihood that it would not be used in the near future, of course, we do not know it could be that immediately it might need it in the future, we do not know.

But the probability is that if you have not used it for a long time, maybe you will not need it again in the near future either. So, such Least Recently Used pages will be evicted. That is the LRU policy. And this is what most operating systems use some version of this LRU policy because this is a sensible policy.

So, how does the OS know which pages LRU now in your page table, you have all the list of all the pages of a process. But how do you know which page has been least recently used? Note that the OS is not informed every time there is a memory access. The MMU is translating the addresses the CPU is accessing the memory. The OS is not involved. So, how does the OS know which pages LRU? There is no like, the OS is not putting a timestamp on every page whenever it is being accessed.

(Refer Slide Time: 22:51)

The screenshot shows a presentation slide titled "LRU implementation" in red text. To the right of the title is a hand-drawn diagram of a vertical stack of five boxes, with a red arrow pointing to it from the word "MMU" written above. Below the title is a bulleted list:

- How does OS know which page is LRU?
 - OS is not involved in every memory access, so doesn't know which pages have been recently used
- Solution: MMU sets the accessed bit for every page table entry it accesses
 - Accessed bit is set implies page has been recently used
- Modern operating systems implement approximate LRU
 - Periodically, look at accessed bit of pages to classify pages into active and inactive pages
 - Pick pages that have been inactive for eviction
 - May also avoid dirty pages for eviction, since it requires extra disk write

A small NPTEL logo is visible in the bottom left corner of the slide.

So, OS has to make this difficult decision of finding the least recently used page. And there is no easy way to do it. So, most modern operating systems implement only an approximate version of LRU, how they do it is as follows in your page table, in your page table array, there is an accessed bit. So, every time the MMU looks up, a page table entry walks the page table finds a page table entry, whenever it touches this page table entry, it will set an access bit.

I access this page therefore I set the access bit I access this page, I access this page and then you will keep setting these access bits. And periodically, what the OS will do is it look at all these access bits in the page table, or these pages have been recently accessed, therefore they are active pages, I will find one of these pages that are not recently accessed and pick them for eviction. So, there is only an approximate LRU.

It is not exact LRU policy, but this is the best that modern operating systems can do. And of course, you can periodically reset these bits look at in the past few intervals when all bits have been set access bits have been set, you can do many improvements on this, but in the end, it will only be an approximate LRU policy that operating systems will use. So, you can pick pages that do not have this access bit set that are inactive or you can also use other heuristics, if the page is dirty, then I do not want to evict it because after all, once again write it to disk.

So, some combination of looking at this dirty bit and accessed bit modern operating systems make this decision for approximate LRU. So, now, we can put everything together and let us see what happens on a memory access. We have seen this before also now we have even more

information about page faults and on demand memory allocation. So, let us revisit the story again.

(Refer Slide Time: 24:38)

What happens on a memory access?

- CPU has requested data (or instruction) at a certain memory address
 - CPU checks caches. If hit in **CPU cache**, data is directly available. Otherwise, CPU has to access memory via MMU
 - MMU checks **TLB**. If TLB hit, the physical address is available, the data is fetched from memory. Otherwise, MMU has to **walk page table**
 - If address is valid and present in page table, permission checks pass, translate address and access memory (MMU adds translation to TLB)
 - For any **error** (address is valid but not present, or if valid bit is not set, or permissions do not permit access) MMU traps to OS for **page fault**, OS services page fault
- Overheads: CPU cache misses, TLB misses, page faults, ...

The CPU is executing the code of a process it periodically requests for either an instruction or some piece of data some variable at some memory location, either instruction or data the CPU requests. Now what happens when that happens? So, the CPU has made a request using some virtual address of course, you will first check in the CPU caches. If it is a cache your data is available you go back to the CPU, now it is a caches miss.

So, then what do you do? You will go to the MMU. So, the MMU will check the TLB to get a physical address corresponding to this virtual address, if the physical address is there, well and good, you will take that physical address go to RAM, you will go to main memory, access that memory contents, put back the contents in the cache and return the data to CPU. If this is not there, the physical address is not there in the TLB.

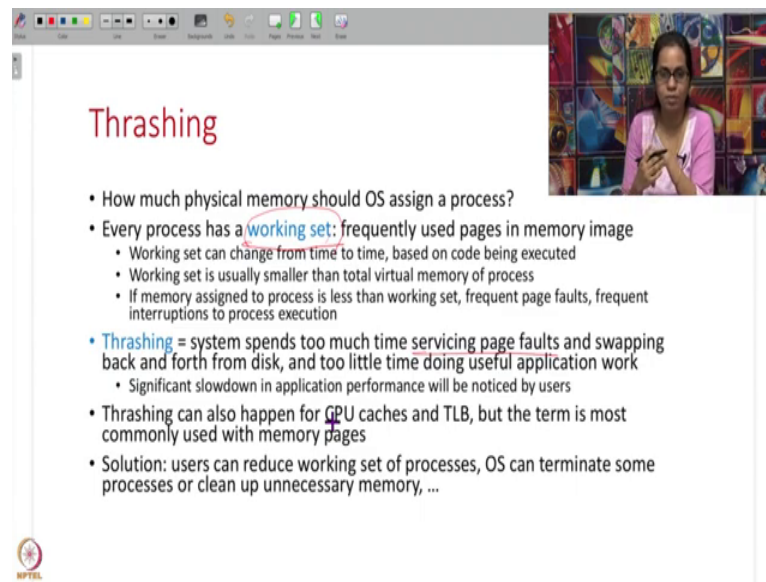
Then the MMU will walk the page table go to RAM, fetch the page table multiple levels of the page tables that will access it will find the page table entry translate the address, put a copy of the translation in TLB, and then access the memory contents. While doing this while walking the page table sometimes there could be an error. If the address is valid and present in the page table well and good you have translated the address and put it in TLB, all done.

But, if there is an error if, the address is either invalid, or valid, but it is not present, some permissions do not match in case of any error, the MMU will trap to the OS for a page fault. And if it is due to the on demand, memory allocation not being done by the OS, and the OS will have to, service, the page fault we have seen, when it will have to find a free frame may have to evict the contents of that frame to disks may have to read the contents of again, from disk from swap space, do all of this fix the page fault and restart.

So, there are many overheads in this memory access, you have CPU cache misses, TLB misses. And finally, page faults, page faults involve multiple disk accesses. So, imagine the CPU can actually access can run instructions very fast at like nanosecond timescales. And if, data is available in cache in a few nanoseconds, things go very fast. Otherwise, you have to do multiple accesses to ram which is once again, hundreds of nanoseconds, you might have to waste. And not even that if there is a page fault, you have to wait several milliseconds possibly to do various disk accesses back and forth from swap space.

So, now the CPU that can operate at a nanosecond timescale is actually waiting for so long for this memory access to happen for it to proceed. So, these things really kill the performance of your application. So, later on, when we study performance engineering, we are going to revisit this topic as to how you can reduce all of this. And a very common thing that you will notice, when a lot of page faults happen, that state is called thrashing.

(Refer Slide Time: 27:44)



Thrashing

- How much physical memory should OS assign a process?
- Every process has a **working set**: frequently used pages in memory image
 - Working set can change from time to time, based on code being executed
 - Working set is usually smaller than total virtual memory of process
 - If memory assigned to process is less than working set, frequent page faults, frequent interruptions to process execution
- **Thrashing** = system spends too much time servicing page faults and swapping back and forth from disk, and too little time doing useful application work
 - Significant slowdown in application performance will be noticed by users
- Thrashing can also happen for CPU caches and TLB, but the term is most commonly used with memory pages
- Solution: users can reduce working set of processes, OS can terminate some processes or clean up unnecessary memory, ...

So, what is thrashing? Thrashing is when a system spends too much time servicing page faults. There is very little memory in the system. And you do not have enough physical memory frames. And every time you access something, there is a page fault, you have to go to this, go back and forth from this, swap this output that and it is like musical chairs, you are spending a lot of time just moving things back and forth from disk and not doing any work at all. That is called thrashing.

And sometimes when that happens, your application performance will slow down a lot. So, when does the thrashing happen? Normally, every process has a working set of, some small number of pages that it is frequently using. So, you have to give every process at least as many frames as to store its working set. If you are working set to say four pages, you have to give the process at least four physical frames, the OS has to do that.

Otherwise, if you only give it three frames, what will happen, it lapses the fourth frame, there is a page fault, you take out one of those frames and it will access this frame, there is a page fault again page fault, you will have frequent page faults and frequent interruptions to the process execution.

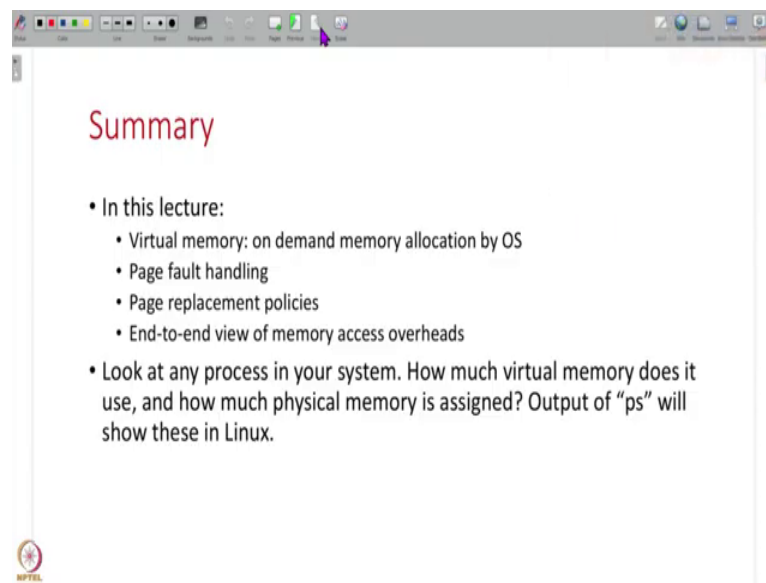
So, that is why it is a good idea for operating systems to estimate what is the working set of a process and give at least as many pages are there in the working set size of a process. That much

memory at least OS has to give processes. Otherwise, you are going to have this issue of too many page faults.

And this thrashing is not just for memory it can happen for CPU caches or TLB also, if you have many a lot of memory that is beyond the CPU cache size or many address translations beyond the TLB size also you can have frequent cache evictions, but the term is most commonly used for memory.

And when this thrashing happens, your system slows down a lot, in such cases what you should do is the users the application can reduce its working set size try to operate with lesser amount of memory OS can terminate a few processes reclaim memory. You have to do something you have to fix so that the performance of your system improves.

(Refer Slide Time: 29:50)



So, that is all for today's lecture. In this lecture we have studied the concept of demand paging and virtual memory how page faults are handled. What are page replacement policies and now, with this over the last three lectures, we have an end to end view of memory access when the CPU accesses a certain code or instruction or data at a virtual address, what are all the things that can happen in this process?

What are all the overheads and how do you go about fixing these overheads. Later on in the course when we study performance engineering, we are going to revisit this topic and come up with certain helpful tips to fix these overheads in real applications. So, that is all for today's

lecture. A small exercise for you is you can actually look at pick any process in your system look at how much virtual memory it thinks it has, and how much physical memory the OS has actually allocated to it.

So, this information if you are on a Linux machine, an output of a command like PS can actually give you this information. Please dig around it a little bit more to understand the concepts of this lecture better. Thank you all and see you in the next lecture.