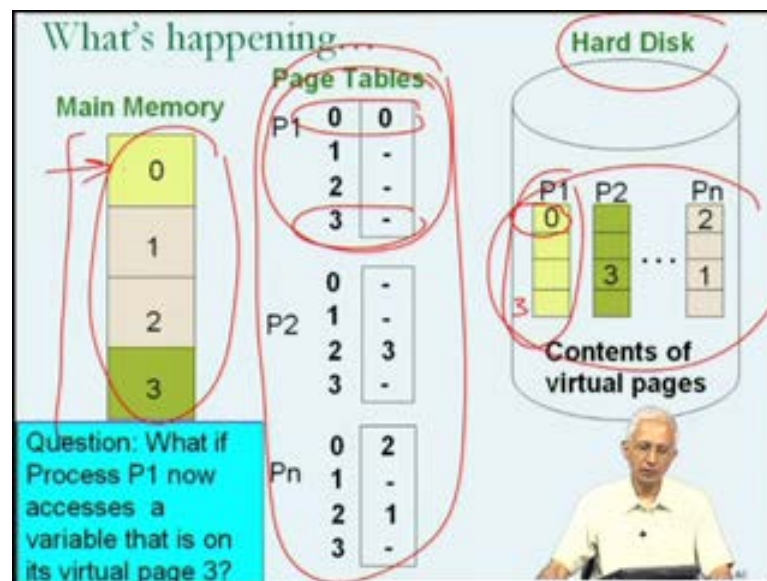


**High Performance Computing**  
**Prof. Matthew Jacob**  
**Department of Computer Science and Automation**  
**Indian Institute of Science, Bangalore**

**Lecture No. #14**

Welcome to lecture 14 of the course on, High Performance Computing. You will remember that, in lecture 13, we looked in some detail at what happens behind the scenes in a paged virtual memory system. We call that, this is the part of the operating system responsibility, which takes care of the sharing of main memory among many programs which might be on in execution on a computer system at a time. So, the summary slide, which totally showed you what is happening behind the scenes, we will go back to the slide right now.

(Refer Slide Time: 00:49)



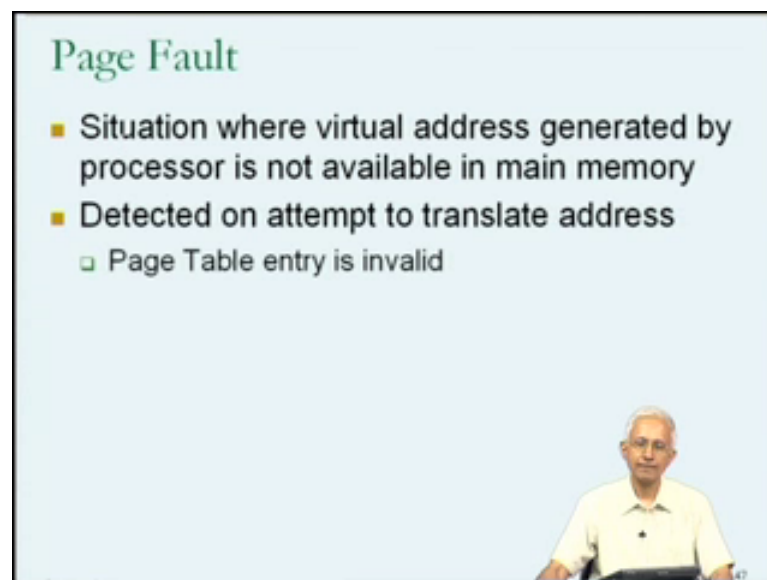
So, in this slide what we had seen was a situation where there are **four processes P 1 the I am sorry** number of processes P 1 through P n. All actually, in some state of execution in that all are in memory, in some form or the other. At this particular point in time, since, obviously, all the pages, all the virtual pages of all the processes cannot be in main memory, which in this example was quite small. Main memory in the small example can

hold only four pages and the sum total of the pages of all these processes is many times of that.

So, what page virtual memory does is, it remembers all the page virtual pages of all the processes, which should be a memory on hard disk, which has a very large capacity and at any given point in time, sum of the those pages would be present in the main memory. And the mapping between the pages that are present in main memory and the virtual addresses that they correspond to, is maintained in the page tables of the processes, as this example showed.

So, we were looking in more detail at one particular problematic situation that could arise and that is the situation where particular processes say P 1 is running and makes a reference to a particular memory page that is not currently present in main memory. For example, if process P 1 was to refer to its page three, if you look at the page table entry for process P 1, you notice that virtual page three is not present in main memory. Among all the virtual pages of process is 0, only its virtual page 0 is present in main memory. In fact, at main memory physical page 0, this particular situation is what is known as a Page Fault, and it is detected when the attempt to translate the address is made.

(Refer Slide Time: 02:28)



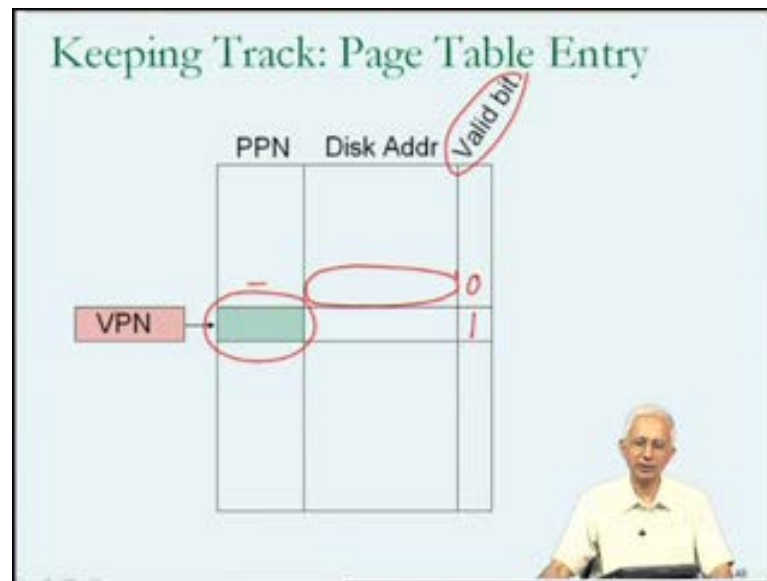
**Page Fault**

- Situation where virtual address generated by processor is not available in main memory
- Detected on attempt to translate address
  - Page Table entry is invalid

The slide features a light blue background with a black border. In the bottom right corner, there is a small inset image of a man with white hair and glasses, wearing a light-colored shirt, sitting at a desk. The text is in a dark, sans-serif font.

So, the page table entry would then be referred to, it would be notice that there is no meaningful mapping and hence the page fault would be identified.

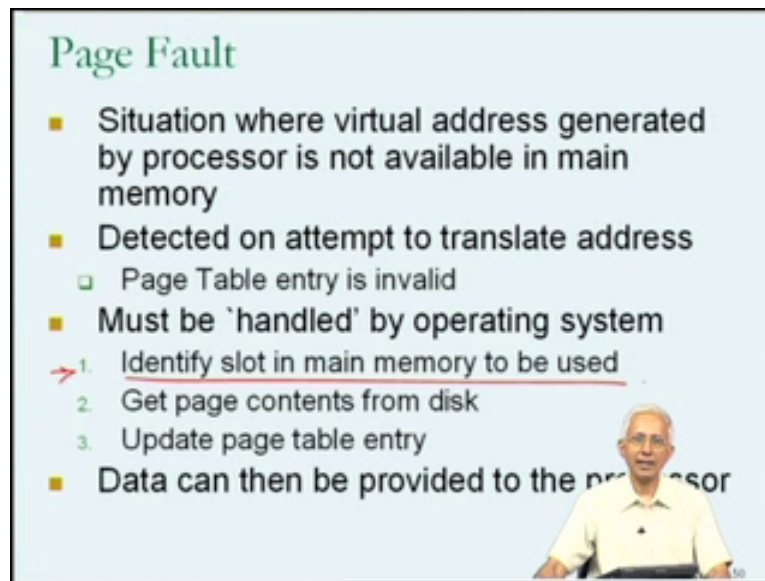
(Refer Slide Time: 02:45)



And in order to do this, it is useful to have a particular bit in the page table entry called the Valid bit, which would have a value of one for those pages which have a meaningful mapping. In other words, the PPN entry for that page is the correct translation information and a value of 0 for which the page table entry does not contain meaningful information and therefore, the copy of the page on disk is the meaningful piece of information regarding that particular page.

So, we have included a valid bit, which is more frequently referred abbreviated as the V-bit. The notation which I am using is to put a question mark to indicate that it is a bit, in any of these table notations.

(Refer Slide Time: 03:29)



**Page Fault**

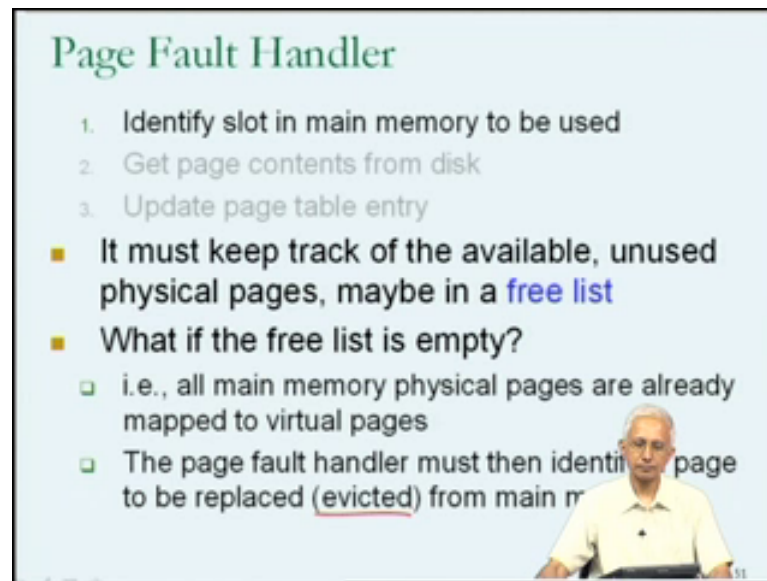
- Situation where virtual address generated by processor is not available in main memory
- Detected on attempt to translate address
  - Page Table entry is invalid
- Must be 'handled' by operating system
  - 1. Identify slot in main memory to be used
  - 2. Get page contents from disk
  - 3. Update page table entry
- Data can then be provided to the processor

50

So, the page fault is a situation that must be handled and we understood that the handling is done by a part of the operating system, which we would call the page fault handler. And what the Page Fault handler would have to do is to find or make space in main memory for the particular page. In this case, page one of process P 1 to be copied from the disk into main memory. So, the first step is to identify place in main memory to be used for this purpose. Subsequently, copy the page **from it** from disk to main memory and subsequently update the page table entry.

So, these are the tasks of the page fault handler and the problematic situation which could arise in this case is that they may be no free slot in the main memory to be used at this point in time.

(Refer Slide Time: 04:15)



### Page Fault Handler

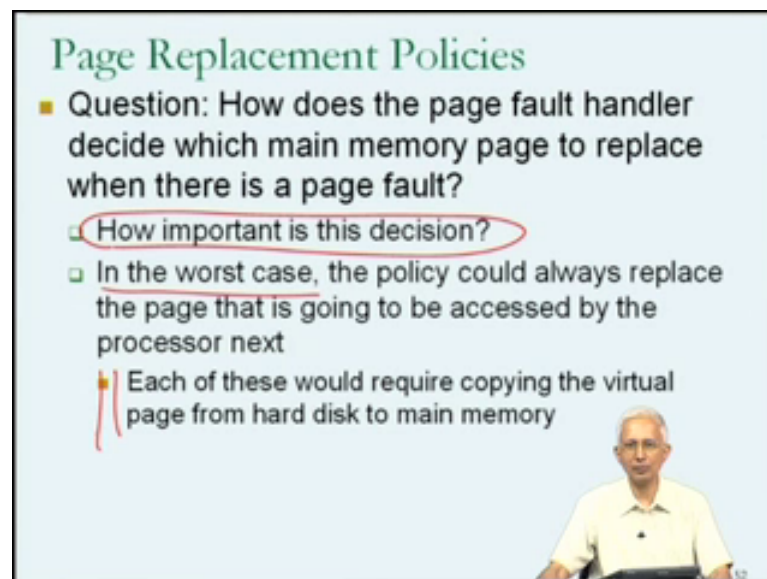
1. Identify slot in main memory to be used
2. Get page contents from disk
3. Update page table entry

- It must keep track of the available, unused physical pages, maybe in a **free list**
- What if the free list is empty?
  - i.e., all main memory physical pages are already mapped to virtual pages
  - The page fault handler must then identify a page to be replaced (evicted) from main memory

51

And if that is the case then the page fault handler would have to actually replace one of the pages which are currently in use, in order to make space for this newly requested page to be copied from disk to the memory. And this is the task of the page fault handler known as page replacement and the task may involve evicting a page from main memory.

(Refer Slide Time: 04:42)



### Page Replacement Policies

- Question: How does the page fault handler decide which main memory page to replace when there is a page fault?
  - How important is this decision?
  - In the worst case, the policy could always replace the page that is going to be accessed by the processor next
- Each of these would require copying the virtual page from hard disk to main memory

52

So, **we were looking at** you are about to look into what kinds of criteria could be used by the operating system page fault handler, in deciding which particular page to eliminate

from main memory, in the event of a page fault. And this particular decision making part of the page fault handler would be an implementation of what is called as Page Replacement Policy.

By Policy, you can think of a policy as being a strategy or in this particular case an algorithm that is used in order to make the page replacement decision and so we are going to address this question, how does the page fault handler decide which main memory page to replace, when there is a page fault and there are no free main memory pages, in other words the free list of free memory pages is empty?

Now, before we actually look at possible policies and get a feel for what the operating system where our program is running might be doing behind the scenes, might be useful to understand how important this decision is. The way that it will understand how important this decision is by thinking of what the worst case policy might do, in other words, if the operating system did the worst job possible what would it do?

One way to think about this is, in the worst case, the operating systems page replacement policy could be so bad that it always replaces a page that is going to be the next page access by the processor, and this were of course, be little difficult to implement precisely, because in order to implement this particular policy, one would have to know the operating system page fault handler would have to know what is going to happen in the future. But this is hypothetically the worst way the things could go. Now, why do I consider this to be the worst possible scenario?

The scenario where every time a page is replaced from memory and it just happens to be the page that would have been needed next. If you think about this a little bit, you will realize that if every time a page has to be remove from main memory that page, which is going to be needed next, is the one that is replaced and this would mean that in the near future when that particular page that was replaced, is requested by the processor, that would generate another page fault, and every one of this page faults as we know, involves copying a page from the hard disk to main memory.

So, in order to get an idea of how important the page replacement decision is, we have to know little bit more about how much of a penalty it is to copy a page from hard disk to the main memory.

(Refer Slide Time: 07:11)

**Aside: Disk Access Speed**

- We saw that there is a speed disparity of about 2 orders of magnitude between Processor (nsec) and Main Memory (~100 ns)
  - Recall: nano  $10^{-9}$
- Hard disk
  - Remembers things by the state of magnetic material
  - Disk is a mechanical device: motors rotating a firm plate coated with magnetic material
  - Aside: Computer noises
  - Reading a page from hard disk could take -msecs (milli:  $10^{-3}$ ) if not longer
  - i.e.,  $10^4$  times slower than main memory!

53

So, let us just step aside little bit and try to get a feel for how slow or how fast the hard disks are. Now, when we were talking about processors and main memory, we were concerned about the speed disparity between the processor and the main memory. and I had given you an idea about approximately how big the speed disparity is today, by suggesting that the time scale on which the processor operates is a nanosecond time scale, whereas, the current main memories operate time scale of about hundred nanoseconds.

In other words, one event of interest may take about one hundred nanoseconds, which is what I would describe by this speed disparity of about two orders of magnitude, one nanosecond compared to 100 or 10 to the power of 2 nanoseconds, and that was of major concern to us. There was a two order of magnitudes speed disparity. So, what is the situation and I will remind you that nanosecond is 10 to the power minus 9; nano stands for 10 to the power of minus 9. So, what is the situation as far as hard disks is concerned?

Let me tell you a little bit about hard disks. I will be telling you a lot more about hard disks later, when we talk a little bit about the input and the output management task of the operating system. But, from what we have seen already, you will recall that on the current magnetic hard disks, such as those used in personal computers, laptops, mini

laptops, as well as lot of servers, the remembering of instructions or data is done by the current state of certain magnetic material.

So, these are magnetic storage devices and one thing which is important to know to know here is that the disk is a mechanical device. In the sense that the mechanical material is on surface and that surface is actually rotated, in order to access the different parts or the different pieces of data stored on the hard disk and this is done by motors.

The rotation of this plate or disk is done using motors and **the motors that** the plate itself has a coating of a magnetic material. So, in some sense, one could view the disk as being a mechanical device, unlike many of the other components of the computer system that we have talked about; we talked about processor, we talked about main memory, which were purely electrical or electronic devices. They contain circuits. In some cases, the circuits could have been of quite different kinds, as we can see, from the fact that the speed disparity between processors and memory are fairly large, but in many events they were all circuits.

Here we have something quite different. This is actually what I would call a mechanical device and again on the smaller side, let me just mention that many of you would notice that your computer, your laptop, or your desktop does make a lot of noise and noise is not something that we typically associate. The kind of noise which you get from a laptop or desktop is not something that you would normally associate with electronic circuits, because in electronic circuits there is some flow of current, some charge accumulation, things of that kind, things happens in very high speed and there is no particular reason to think that noise might be generated.

Whereas, we know that our computers are quite noisy and these noises are generated by some of the mechanical devices inside the computer. One of those mechanical devices, which generates a good part of the noise that your computer is guilty of are the disks. There is a disk to rotate, the magnetic coated surface, there are other motors for other purposes within the disk drive. Now, some of the other noises that you hear from your computer since we are just on smaller side talking about the noises are actually due to fans. Fans once again, a mechanical devices and they too have motors, but you may have wonder why are the fans inside your computer and the answer to that question comes from the electronic, electrical circuits in the computer.



Now, when the electrical circuits are operating they consume energy. **They from** You know that the electric bill at home is probably higher than it used to be years ago, before you had a computer, and this is largely because the computer does consume electricity and good percentage of the electricity is consumed by the electronics, by the processor. If the processor is doing things it needs energy to do things. But as a result, as you know even current passes through a wire, heat is generated. In other words, some of the energy is dissipated in the form of heat.

So, as this heat for a high speed processor, the amount of heat generated could be substantial and therefore, there is a need for cooling, hence the fans and therefore, the noise. So, at the moment we are not too concerned about power. In this course, we would not talk about problems of heat, but very clearly there are mechanical devices inside the computer and currently were worrying about one of them the disk.

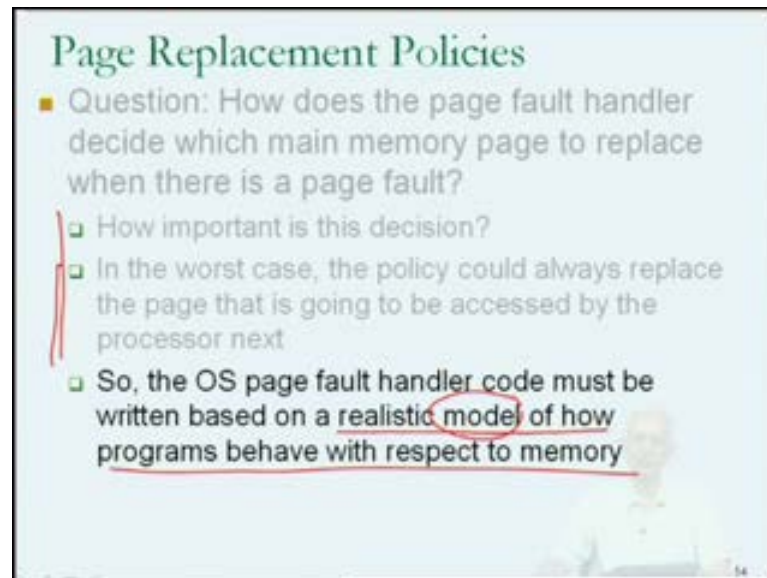
So, the disk is the mechanical device, which means that its rotation is going to be done by a motor and hence the speed of the disk is to some extent going to depend on how fast the motor is rotating the disk, and you may have heard about disks which rotate at 7200 revolutions per minute or RPM. Faster disks, which rotated at 10000 RPM, and the faster the disk, conceivably, the faster the rate at which data can be transferred from the disk to the memory. But, how fast is the disk access, if one assumes and makes reasonable assumptions what might see in a current laptop or desktop.

Now, the answer is that reading a page from a hard disk would take not nanoseconds, but on the order of milliseconds, and the millisecond is  $10$  to the power of minus  $3$  and in fact, the amount of time to read a page from the disk could be substantially larger than milliseconds, it could even run into seconds or more depending on the current state of the disk and again, I will go into the more about disks later. So, when we talk about reading page from a disk, at the very best way talking about milliseconds, we could be talking about seconds, compare to the  $100$  nanoseconds that we were concerned about with memory.

So, disk accesses are, let us say 4 orders of magnitude, I will just use that as a ballpark figure for the moment. If I assume that disk access takes millisecond and that main memory is on the scale of  $10$  to the power of  $2$  nanoseconds and you do the calculation

here, you will find that the disks are at are several orders of magnitude slower than main memory; main memory, itself is of two orders of magnitude, slower than the processor.

(Refer Slide Time: 14:01)



**Page Replacement Policies**

- Question: How does the page fault handler decide which main memory page to replace when there is a page fault?
  - How important is this decision?
  - In the worst case, the policy could always replace the page that is going to be accessed by the processor next
  - So, the OS page fault handler code must be written based on a realistic model of how programs behave with respect to memory

So, going back to the question which raise this aside about disk access speed, the question of how important is decision, the decision of which page to replace from memory, in the event of a page fault. We realize that this is a very important decision, because with the bad decision many more page faults will be generated, and the more page faults there are, the more disk accesses have to take place, and each disk access is four, may be more order of at least four orders of magnitude, slower than a main memory access. So, this is a very important decision.

So, the bottom line that we learnt from this, Page replacement policies are important, they have a very big impact on the execution time of our programs. We do need to know something about how the page replacement part of the operating system works. In case that will give us some insight into how we should modify the way that we write our programs. But, its stands to reason from this big speed disparity between main memory and disk that the OS page fault handler, must be written in some intelligent way, and one strategy, which is often used is to have some kind of a model or some kind of an understanding about how programs behave with respect to memory.

So, if the person writing the page fault handler has some way of reasoning about how programs behave, then he can use that knowledge, he or she can use that knowledge to decide how to write the page fault handler. In other words, how to decide which page could be replaced from memory if the need arises during the handling of a page fault.

So, what we need now is some knowledge about what are considered being realistic models of how programs behave with respect to memory. A model is some kind of an abstract understanding of a physical phenomenon. In fact, abstract description of a physical phenomenon.

(Refer Slide Time: 15:47)

The slide is titled "Page Replacement Policies" and focuses on the "Principle of Locality of Reference". It contains the following text and annotations:

- Principle of Locality of Reference** (circled in red)
- A commonly believed/seen program property
- If memory address  $A$  is referenced at time  $t$ , then it and its neighbouring memory locations are likely to be referenced in the near future (circled in red)

Handwritten annotations include:

- "Low" written above the title.
- "Memory Addresses" written above the title.
- A diagram of a memory stack with addresses  $A-3, A-2, A-1, A, A+1, A+2, A+3$  and a red circle around  $A$ .
- "now" written above the text "at time  $t$ ".
- "by word" written above the text "neighbouring memory locations".
- "A" circled in red with an arrow pointing to "Temporal Locality of reference".
- "A-1, A-2" and "A+1, A+2" written below the text with arrows pointing to "Spatial Locality of reference".
- "Time" and "Space" circled in red.

A small video inset of a man in a white shirt is visible in the bottom right corner of the slide.

Now, today one of the most widely respected, most widely believed models for how programs behave with respect to memory access is something known as the Principle of Locality of Reference. It is referred to, it is a principle, and it is not called a law. A law is something which holds now, always held, will always hold anywhere. This is just a principle, which means something which we believe holds a lot of the time to some extent. Its technically what principle is use for here.

The word reference is here referring to a memory access or a memory reference. So, instead of reference, you could think that the word memory accesses. So, here principle is being stated as principle as some kind of a generalization about how programs behave in terms of their memory access behavior, and the key word here is something called

locality and I will state a principle of locality of reference for us to just get a feel for what...

So, basically this is some kind of a commonly believed or commonly observed program property, and this is a statement. Now, the statement says, let us suppose that a particular memory address, which will be called A is referenced at some particular time t, think of it as being now. So, as the program is executing some point in time, let us call it t<sub>mem</sub>, the program accesses memory address A.

Now, what the principle of locality of reference tells us is that it is likely that particular address, as well as its neighboring memory locations, will be referenced in the near future. So, this particular principle is giving us some kind of a hint or it is giving us some kind of a model about what is going to happen in the near future as far as a typical program is concerned, and this is what we would like to have.

You will recall that in describing the worst case policy we were, because of our uncertainty about what was going to happen in the near future, we could have used of a policy which did the worst possible thing. So, with a principle like this you get some kind of a hint about what is conceivably going to happen in the near future that could be used to prevent page replacement policy for making bad decisions.

So, again read this slowly, the principle is suggesting that if you want to get some feel for what, which, pages or which memory locations are going to be accessed or referenced in the near future, look at what is happening now, and if particular address is being accessed now then, it is quite likely that that particular address, as well as its neighboring memory locations. So, what could neighboring memory locations be? Now, you will recall that we, whenever, talked about memory, memory was an address space starting with address 0 and going up to some maximum possible address.

So, if I was talking about particular address A that is basically some unsigned integer some number. A could be the memory address hex 1 0 0 0, for example, it is a number. What do I mean by its neighboring memory locations? Quite clearly, looking at the picture of memory, I must be referring to the memory addresses which are close to A. In other words, A minus one, as well as A plus one, as well as A minus two, as well as A

plus two and so on. The addresses around including, and around A would constitute the neighborhood of A.

Now, there are two aspects of the statement that have to be understood. One is the suggestion that if memory address A, is being referenced now, there is slightly that it (Refer Slide Time: 19:22), in other words, memory address A itself will be reference again in the near future. There is a second aspect of this statement, which is that is memory address A is reference now, then it is likely that its neighbors in other words A minus one, A plus 1, A minus 2, A plus 2 etcetera are going to be reference again and are going to be referenced in the near future.

So, there are these two aspects to this statement of the principle and in reading literature related to memory, you may come across two terms, which are used. One is to talk about the first part of the principle, in other words, the idea that A itself is likely to be reference in the near future, as a statement of Temporal Locality of Reference. Temporal is a word relating to time and another term which is used is to talk about the likelihood that the neighbors of A.

In other words, A minus 1, A minus 2, A plus 1, A plus 2 etcetera, will be referenced in the near future from time t, is referred to as the principle of Spatial Locality of Reference(Refer Slide Time: 20:25), which is referring to space, and you can quite clearly understand where the two terms are coming from. So, people may separately talk about how the programs show good temporal locality of reference or the program show good Spatial Locality of Reference and this gives us an idea of what they mean.

Now, before going ahead and looking at page replacement policies, we do need to get some confidence in this principle, **the in the leggie** I mean that this principal is legitimate means that the program that we write are likely to show the properties which are described here.

(Refer Slide Time: 21:00)

### Locality of Reference

- Based on your experience, why do you expect that programs will display locality of reference?

The diagram illustrates the memory layout of a process. It shows a vertical stack of memory segments. At the top, the address 0 is marked. The segments are labeled from top to bottom: 'text', 'data', 'heap', and 'stack'. The 'text' segment is circled in red and labeled 'instructions'. The 'data' segment is circled in red and labeled 'data'. The 'heap' segment has a downward-pointing arrow below it, and the 'stack' segment has an upward-pointing arrow above it. At the bottom, the address  $2^n - 1$  is marked. A person is visible in the bottom right corner of the slide.

So, once let us do that, let us ask a question based on our experience, why do we expect that the programs that we write will display locality of reference. Now, the picture that we have of the programs that we write, let me just remind you is that when I write a program I write it in C, and subsequently that program is compiled into something, which in the machine language, which deals with memory references. The program that I write does not directly deal with memory references. I could actually write a program in C without knowledge that there is a memory at all. Some of you may have started this course without a clear understanding of why a computers system had main memory.

So, clearly it is possible to write C programs without any knowledge of main memory. Now, we have a better understanding of main memory. We understood that in the Unix, Linux, world if I look at exactly how main memory is used, this is a virtual address space of a process going from **some address 0** the address 0 up to some address of two the power n minus one, and I know the some of the address space is used for the instructions of my program.

Some of the address space is used for the statically allocated variables of my program, some of the address space is used for the heap allocated variables of my program, the heap might grow and some of the address space is used for the stack allocated variables of my program, the local variables of functions parameters of functions and so on. So,

what we need to try to understand is **as far as the, remember,** the text relates to the instructions of my program.

So, the memory reference is that my program makes could be instruction references and the memory references could be data references. When your program executes, it accesses memory both ways, to fetch instructions as well as to fetch data or to store data. Therefore, in trying to assess what extent our programs show good locality of reference we have to look both at the behavior of programs, these are the instructions and the behavior of our programs vis-à-vis data, which is what we will do next.

(Refer Slide Time: 23:02)

**Locality of Reference**

Based on your experience, why do you expect that programs will display locality of reference?

**Same address (temporal)** *A itself*

**Neighbours (spatial)** *A-2 A-1 A+1 A+2 ...*

<b>Instructions</b>	Small loop Function	Sequential code Loop
<b>Data</b>	Local variable Loop index	Stepping through array

*int A; A; A+1; A-1*

*for (i=0; i<=N; i++) x[i]*

We will try to come up with examples of things, which are programs commonly do and try to access whether they are going to be good or bad from the perspective of locality of reference. So, in doing this, I am going to look separately at instructions and data, because they may have different properties as far as locality is concerned, for the typical program. And we looked separately at the same address type of locality, which I call temporal locality, and the neighbor locality which I called spatial locality. Remember, the temporal locality was the fact that it is highly likely the A itself will be reference again in the near future. Spatial locality was the likelihood that A minus one, A plus one, A minus two, A plus two etcetera, in other words, the neighbors of the address A, which is the current reference would be accessed in the near future.

So, we are going to come up with the table in which we are going to try to list things, which happen in the programs that we write that are likely to produce good instruction temporal locality of reference or good instruction spatial locality of reference and so on. And if this table is satisfying then we can say that we believe the principal of locality of reference, as far as, the programs that we write are concerned, so this going to be useful exercise. This will also give us some insight into what kinds of things we could do in writing our programs in order to enhance their locality of reference.

So, let us think about instructions first, why would it be or why would it happen that instruction accesses could show good temporal locality of reference? In other words, why would it ever be the case that a particular instruction  $i$ , is currently being fetched from memory, it is likely that the same instruction  $i$  will have to be fetched sometime in the near future. That is essentially what the question of temporal locality would address.

So, if instruction  $i$  is being executed now, apparently it is likely that instruction  $i$  might be executed in the near future. Now, what kind of program features might cause this to happen? You think about this little bit and clearly we are talking here about either some repetitive behavior, in which case, a small loop would cause instructions to show good temporal locality of reference.

If I have a program in which there is instructions  $i$ , and instruction  $i$  plus one, and instruction  $i$  plus two, possibly also instruction  $i$  minus one, all forming a loop. In other words, after executing instruction  $i$  plus two, it loops back to  $i$  minus one, possibly after checking some condition. Now, if I do have such a small loop, let suppose that at time  $t$ , instruction  $i$  is executed then possibly shortly after that instruction  $i$  plus one, shortly after that instruction  $i$  plus two, shortly after that instruction  $i$  minus one, and after that in the very near future instruction  $i$  being executed.

So, programs which have small loops, if the loops have a high iteration count, then they would show excellent temporal locality of reference. Now, what are the kinds of behavior of, what are the kinds of programs might show good temporal locality of reference?

The key here is that there are some kinds of repetitive behavior and one way to have repetitive behavior is to have a loop. Another way to have repetitive behavior is actually



to have a piece of code which is frequently called. In other words, if I have a small function or a function which is called very often then **if I looked it one of the...**, I have a function and I consider one of the instructions in that function, **it is instruction, it is** if the function is small, then every call to the function may just involve executing, a small number of instructions, is soon after that the function is called again, then the instruction  $i$  is going to be executed again.

So, frequently execute small functions once again, I going to show very good temporal locality of reference. So, if these are features of your programs, the programs that you did normally write, small loops, functions then you know that your program is going to show good temporal locality of reference as far as instructions are concerned, and you could in fact, enhance the temporal locality of your program as far as instructions are concerned, by using these features, a little bit more you could also try to think about.

Remember, when I talk about small loops, I am talking about while loops, if loops, `for` loops, everything comes under that. You could also think about to what extent other features of your program enhance the temporal locality of reference, given the definition of temporal locality. But let us move on, let us next try to think about **why it would be why** to what extent the programs that we write shows spatial locality of reference as far as instructions are concerned?

In other words, if instruction  $i$  at address  $A$ , I am referring it to as instruction  $i$ . Until now, I was referring to it as the thing at address  $A$ , but let suppose instruction  $i$  is instruction currently being executed. The question of spatial locality is that why is it likely that instruction  $i$  plus one, instruction  $i$  plus two, instruction  $i$  minus one, instruction  $i$  minus two would be reference again in the near future, and we have seen from our discussion of temporal locality that why instruction  $i$  minus one and  $i$  plus one might be reference again in the near future as far as loops are concerned.

So, in immediate answer in the next square which we are going to fill up, the square about why instructions might show good spatial locality of reference, is going to be relating to loops, as we have just seen. But more frequently occurring scenario is the ordinary code that we write where there is no control transfer, which is sequential code. After instruction  $i$ , if I have a piece of sequential code, in other words, I forget about this loop, I just have code, which is executing sequentially, then after instruction  $i$  is

executed, instruction  $i$  plus one is executed and after that instruction  $i$  plus two is executed. In other words, the instructions in the neighborhood of  $i$  going forward in memory,  $i$  executed.

So, sequential code shows good spatial locality of reference. We see that loops cause instruction  $i$  itself to be executed again in the near future, particularly if loops are small, but loops in general are going to cause good behavior as far as spatial locality of reference is concerned too.

Because after instruction  $i$  plus two has been executed instruction  $i$  minus one is going to be executed, in this example, if we have a loop. So loops enhance the spatial and temporal locality of your program. So, if your programs contained functions, loops, sequential code, and then you know that your program is going to go, show good properties as far as the locality of reference is concerned with respective instructions.

And again a little bit of thought about this tells you that if you write your programs under principles that were taught you in a course on programming, such as modular programming or object oriented programming, then these things happen automatically. In other words, the idea of modular programming or object oriented programming by definition enhances the temporal and spatial locality of reference of your programs and you do not possibly have to worry about specifically as far as instructions are concerned. Good programming practices will result in good locality properties for instructions, but does the same hold for data? Let us fill the table a little bit more to get some understanding of what kinds of features of data accesses would cause our programs to show good temporal and spatial locality of reference.

Now, let us think first about temporal locality. So, the issue here is I have a variable  $x$  and currently in a situation  $x$  is the variable at address  $A$ . So, I am currently in a situation where my program is just accessed address  $A$ . As the program shows good temporal locality of reference, suggestion is that in the near future it will reference that variable again. So, you need to ask our self that what kind of scenarios and programs would cause this to happen. A variable which is currently accessed is accessed again in near feature and you can think of situations where this happens. For example, if I have a function, we had the example of our function  $B$  from our function called scenario, where they were

two local variables. I think there were called x and y. They may have been called A and B, but, they were two local variables.

Now, why did I declare those as local variables? I declare them as local variables because I wanted those two pieces of data to be used while the function was executing. Therefore, it is very likely that those two variables will be accessed within the body of the function. So, if one of them is accessed at time t, my reference to memory location A, then it is very likely that some time in a near future it will be accessed again, because it is a local variable of a function. There was a small number of local variables of the function they were included their specifically, they are likely to be accessed again and again within the function. Therefore, local variables are likely to show this property.

Another kind of a variable that you use which may well show temporal locality of reference, if you think about a bit, is a loop index. What do I mean by loop index? Consider a for loop. In a for loop, you have a variable called i which is initialized to some value, and then there is some value which is checked. So, i is compared to some value to identify the termination condition and every time through the loop the variable i is incremented.

Here that the technical term for the variable i is to refer to it as a loop index. It is the loop the variable which is used for a for loop to specify how many times a loop should iterate and the value of i could also be used within the loop. For example, if this is a variable a loop which is doing something to an array then it often happens that you index the array in this case, I am calling x, using the loop index or some function of loop index, which is i plus one.

So, if I am inside a loop, at the beginning of the loop, the loop variable, the loop index is going to be accessed and very frequently within the loop it happens, often happens that the loop index is accessed again and again, due to the fact that it is the loop index. So, loop indices are good examples of variables, which will likely to show good temporal locality of reference and many of you write programs which use such loops.

So, these concepts, the idea that functions or loop, for loops have variables, which are likely to be used a lot within that function or that loop suggest a program, which uses these features will show reasonable temporal locality of reference. What about spatial locality

of reference as far as data is concerned? In other words, if a particular address  $A$  is accessed or is being accessed, now the idea that it is quite likely that address  $A$  plus one or address  $A$  minus one, would be referenced in the near future. Remember that these are the addresses of data items.

So, what kind of data objects might cause this property to be seen? Now, immediately the thought which make come be coming into your mind is about arrays. If you do have, let say one dimensional array. Remember, an array is up on the top, I am going to draw an example of an array. This might be the array  $x$  (Refer Slide Time: 34:01). So, each element of the array has an index. This might be an array of size 100, in which case the index of the last element will be 99, the index of the second last element would be 98 and so on.

So, let us suppose at a time  $t$ , I was accessing the element  $x$  of three. Now, if I have the reason that I was setting up this array was to let say step through a series of data, then could well happen that after accessing one particular element I will access the next element and so on. So, this notion of stepping through an array is very clearly going to show very good spatial locality of reference.

The same could be said about things like struct or structures. If I do have a collection of different kinds of data which are associated with each other and defined as a struct declared as a struct, **see** struct, then the elements of the struct are going to be stored in neighboring memory locations, which means they are going to have addresses by which one could call them neighbors of each other and if I access one field of the struct and then another field of the struct, and then another field of the struct, then all of these references if they happen close to each other in time, would enhance to the spatial locality of reference of my program.

So, putting all these together, it looks like there are certain principles which may fallout. As far as instructions are concerned, good practices might be all that we have to worry about for the moment. As far as data is concerned we may have to be aware that in the indices, loop indices, local variables are special and that in stepping through an array, it might be go to try step to the array, let say sequentially, rather than stepping through the array in some random fashion.

By random fashion, I mean first accessing the 98th element, then the zeroth element, then the 6th element and so on. Because they would be a benefit from the perspective of spatial locality of reference as far as the sequential stepping through the, serial stepping through the array is concerned and therefore, if that is under my control, I could enhance the spatial locality of reference of my program by doing the stepping through the array.

So, some ideas which come to mind about how it might impact the way that we write programs. But, all of this discussion about locality of reference came about because we were trying to understand what considerations an operating system page fault handler might use in deciding, which page to replace from memory, in the event of a page fault and recall this locality of reference was a principle about how the typical behavior of programs as far as memory accesses is are concerned.

(Refer Slide Time: 36:34)

**Page Replacement Policies**

- For a program that displays good locality of reference what would be a good page replacement policy?

A diagram shows a horizontal timeline labeled  $t$  at the right end. A vertical line labeled "now" intersects the timeline. To the left of "now", there are points labeled  $P_3$ ,  $P_2$ ,  $P_1$ , and  $P_0$ . To the right of "now", there is a point labeled  $P_x$ . A red circle is drawn around  $P_3$ . A red wavy line is drawn under the timeline between  $P_1$  and  $P_x$ .

A page fault occurs on reference to page  $P_x$   
Which page should be replaced from memory to make space for page  $P_x$ ?

Candidates:  $P_x, P_0, P_1, P_2, P_3$  all the pages in main memory

**Pick from them the page that was referenced least recently**

So, stepping back to the page of replacement policy, we now have this question, for a program to displace good locality of reference, spatial and temporal, what would be a good page replacement policy? In other words, what would be a good consideration for the operating system page fault handler to use in deciding, which page to replace from memory? Now, let us try to understand the principle in a slightly different way. Now let us look at this line which is a time line. So, the time line starts at some point in the past it goes to through to the indefinite future and somewhere in the middle is now, which might be labeled as 0, just sometime in some intermediate time.

Now, what do we know from the principle of locality of reference? We know that let suppose that I know the memory reference, which is happening now. Let we most specifically say, let suppose that at the time called now, a page fault has occurred, due to the reference to a particular page, which I will call  $P_x$ . I am not going to talk about particular memory variables or addresses anymore, because we know that from the perspective of page virtual memory, both the virtual the physical address spaces are viewed as pages and therefore, and the replacement of entities from memory is not going to happen on in terms of bytes or words, it is going to happen in chunks of pages.

So, I will refer to the individual references as being page references and the offset within the pages is of secondary importance at that the moment. So, at now what has happened is or what is happening now is at page fault has occurred due to a reference to some variable which is on the page  $P_x$  that **could be p p** could be page  $P_3$ , whatever it is.

Now, what do we know about the consequences from the principle of locality of reference, about the fact that page  $P_x$ , has been reference. We know that the next reference is lightly to have the next, if I look into the future slightly that I will see references to memory locations, which are either on page  $P$  or the neighbors of page  $P$ . That is one way to look at the principle of the locality of reference, but does not help us very much in making the page replacement decision. The page replacement decision is the decision that has to be made is among all the pages which are currently present in main memory, which one should I replace from main memory, in order to make space for page  $P_x$ .

So, the decision has to be may based not on the identity of page  $P_x$ , but on the identity of the pages, which are currently present in main memory. So, let us assume that well. So, this is the question that which page should be replace from memory to make space for page  $P_x$ .

Now, let me assume that I can look back in time. I know that I cannot look forward in time, but I do know that I can look back in time, because in order to look back in time, I really just have to remember, I mean before time now, if I think about time now minus one, if I remember what happen that time now minus one somewhere, then when time moves forward, I will be able to look back in time. So, I can look back in time by just remembering what happened in the past when it happened.

Now, let suppose that I knew something about among all the places, which were in memory, which once were actually accessed **in the near future, I am sorry** in the near past, I cannot look into the future, but I can apparently look into the past. Now, let suppose that currently the pages which are in main memory are the pages P 1, P 2, P 3, through P n. So, there n pages in main memory and my objective is to identify one of them as the page to be evicted. Now, let suppose that the page which was referenced just before and now, in other words the previous memory reference was to page P 1. P 1 is one of the pages which is currently in main memory. Now, by the principle of locality of reference, I know that if page P 1 was referenced there at time now minus one, then it is quite likely that it will be referenced again in the near future, which means that as far as now is concerned page P 1 is not a good candidate for replacement. Page P 1 is one of the pages, which is likely to be reference sometime in the near future. It was referenced at now minus one and is therefore, likely to be referenced at now plus one or now plus two, sometime in the near future.

Similarly, **if I know that page P 2, sorry** P 5 was reference just before Page P 1 in the recent past that I know that page P 5 also is not a good candidate for replacement, since it is likely to be reference sometime in the near future. This gives me the base for identifying a page replacement policy and the idea might be, I look back at the recent references an any page in main memory, which was referenced recently, is not a good candidate for replacement, by the principle of locality of reference. It would not be good for me to replace page P 1 or page P 5, since they are likely to be reference in the near future.

So, then the question arises among all the pages, which are currently present in main memory, I have eliminated page P 1, I have eliminated page P 5, as good candidates for eviction. So, which of these pages would be the best one to evict and the answer as you can quickly see is, I want to evict that page from main memory that was referenced least recently. In other words, I continue this looking back in time and I eliminate page P 1 because it was referenced at now minus one. I eliminate page P 5 because it was reference at time now minus two. Similarly, I eliminate other pages by looking back in time and I am left to the one page. That is the page which was referenced least recently and that would in fact, be the page which is best to replace. So, let suppose that the page which was referenced is recently was page three. The previous reference of page P 3 was

some time in the distance past, which means that it very unlikely the page p 3 is going to be reference any time in the near future.

Among all the pages which are present in main memory, P 3 is the one, which is least slightly to be referenced in the near future based on the principle of locality of reference. So, if my programs show good locality of reference then would make a lot of sense to use this red statement at the bottom as the basis for defining a page replacement policy. In other words, pick from all the pages in memory that page that was referenced least recently. What would you call a page replacement policy that uses this idea? You would call it the least recently used page replacement policy.

(Refer Slide Time: 42:53)

The slide is titled "Least Recently Used (LRU) Policy" in blue text. It contains two main bullet points, each with sub-points. The first bullet point is "Keep track of when each page was last used", with sub-points: "With a timestamp" (circled in red, with "time" written next to it), "LRU page: the one with the smallest timestamp", and "Requires a large number of comparisons" (circled in red). The second bullet point is "Or, keep track of the stack of recently used pages", with sub-points: "LRU page: at the bottom of the stack" and "Stack must be updated on every memory access" (circled in red). At the bottom, a third bullet point states "So, LRU might be too expensive in practise" (circled in red). To the right of the text is a hand-drawn diagram of a stack with five levels, numbered 1 to 5 from top to bottom. The number 7 is written above the top level, and the number 3 is written above the second level. A vertical line is drawn to the right of the stack, and a red circle is drawn around the bottom level (level 5).

And this is a commonly use term in computers circles and is frequently abbreviated as L R U. So, the least recently used page replacement policy builds heavily on the model defined by the principle of locality of reference, as we now understand it. Now, the question which arises now is, we do need to satisfy ourselves that the operating system page fault handler can actually use the least recently used policy. In other words that it is a feasible policy. We have seen that it makes a lot of sense from the perspective of principles of locality of reference, but if it is not feasible to implement it then we need not take it into consideration, operating systems would not use it.

So, how do we how do we argue about this.



So, you saw that what we were doing in our timeline was we were looking back in time and seeing when a page was used last. We look back at now minus one and saw that page P 1 was used. We look back at time now minus two and saw that page P 5 was used and so on. So, essentially to implement the least to recently used policy, the operating system page fault handler would have to keep track of when each page was last used. For example, for P 1, it would remember that page P 1 last used at time now minus one. For page P 5, it will remember that P 5 was referenced most recently at time now minus two and so on.

So, this is a kind of information that the page fault handler would have to keep track of in order to know which page among the P 1 through P n was least recently used, and you could do this by associating a timestamp. In other words, the time at which that page was most recently accessed, along with a page table entry corresponding to that page, then when the time comes to make a page replacement decision, **it could look at the** it could look among all the timestamps in the page table to find the page, which has the smallest timestamp. What does it mean to have the smallest timestamp? By time stamp, I mean some indication of time at which the reference was made.

So, if now is six'o clock the now minus 1 is six'o clock minus few nanoseconds and so on, these are times. So, to identify the least recently used page, we look for the page which has this smallest timestamp, and that will be the page, which was referenced farthest ago, in the past. That is why the timestamps is the smallest, these timestamps are monotonically increasing. Now the question arises, is it feasible to actually do both of these things? In other words, to among all the page table entries to find the one which is the smallest and also you will notice that we are also talking about updating the page table entries, in order to keep track of the timestamp. In other words, whenever page P 1 is accessed, its page table entry will have to be accessed to update the timestamp; otherwise, it will have a timestamp from the previous reference.

Therefore, there is a fair amount of work involved in updating the timestamps because for every memory reference page table entry will have to be updated. Timestamp entry will have to be updated for page P 1 and then potentially a huge number of comparisons to identify the timestamp which is smallest, when a page replacement decision has to be made.

Now, is this a big consideration? The fact, that in order to find this smallest timestamp you will have to compare a lot of timestamps. How do you find this smallest among hundred numbers? You have to compare them in some fashion and ultimately find out which of them in these comparisons ends up is the consensus minimum value. How many comparisons might we be talking about here? If you have only four pages is might not be a large number of comparisons, but if I have a main memory which is gigabytes in size, then the number of pages in the main memory could be substantial. It could be millions and if I have to find the smallest among millions of timestamps that could take a fair amount of time.

So, the large number of comparisons required to identify the least recently used page, which will be identified by the page, which has the smallest timestamp of recent reference in its page table entry, might mean that this becomes infeasible policy, and unless I could think of some other way to keep track of the least recently used page. Now, instead of keeping track of the least recently used page using a timestamp, what if I keep track least recently used page. In fact, of the order in which would be referenced using a stack, in other words, the stack of recently use pages and how might this work. You all know what is stack is; the stack of books on a table.

So, when a new book comes, you put it on the top of the stack. Why do I talk about a stack? Because, if I have a stack of references, every time a page is referenced, I put it on top of the stack. If it was somewhere in the middle, because it is reference from the recent past was not too recent, then I take it off in the middle of the stack and put it on to the top of this stack. Therefore, if I look at the stack at any given point in time, the top of stack will be the most recently reference page, the second and the stack would be the second most recently reference page and so on. The Page at the bottom of the stack would be the least would be the least recently reference page. In other words, candidate for replacement, so here we are talking about the idea of maintaining a separate stack.

This has nothing to do with a function call and return stack. This would be maintained by the page fault handler. This is just a stack of page numbers. So, if page 3 was least recently used, it will be on the bottom of the stack, page 1 was most recently used; it is on top of the stack. In our example, page 5 was the second most recently used page; it is on second position of the stack.

So, what will have to be done every time a page is referenced? I will have to make sure that this stack, this LRU stack, is it might be called, is kept correct. It should always contain the information of the order in which, if you think about it, this particular stack is keeping track of the order in which the recent references happened to these pages.

So, let us suppose that right now the page P sub x is being referenced, and it is not causing a page fault, but it actually happens to be page P 7, which is down here in this stack. As I mentioned, to update the LRU stack, I will have to take P 7 out from here and put the number 7 on top of the stack. Therefore, to keep track of the stack of least recently used pages, every time a page is referenced. It will potentially have to be searched for in this stack, moved from its current location and put on to the top of the stack, which once again if there are millions of pages in memory, might end up being extremely expensive, because this will have to be done on every memory access, not on every page fault. The updating of the time stamp would have to happen on every memory reference, not on every page fault, these could be very frequent events.

So, neither these seems to be a satisfactory policy for large main memories. They might be acceptable policies for very small main memories, but we know that main memories today are large and therefore, we might actually come up with the conclusion. So, the LRU page would be the one of the bottom of the stack in this description and there is a problem that the stack must be updated on every memory access.

So, the conclusion that we may come up with, I mean we looked at to what seem to be very reasonable ways to implement the LRU policy and we have also argue that neither of them is really a good idea in terms of implementation, because they will slow down the page fault handlers, substantially. Because for millions of pages, it will take a long time to do the comparisons and it will take a long time to keep updating the stack compare to the processor time, which is nanosecond. We do not want to slow things down too much and the bottom line conclusion might be, LRU might be too expensive in practice.

We might not find operating systems actually using LRU as a page replacement policy. This is unfortunate why, because we had come up with we learned about the principle of locality of reference, we understood that it is seems to make sense for the kinds of programs that we write, we came up with a page placement policy based on that model

of program behavior, but we find out that there it might not be feasible. It might be too expensive to implement in practice.

(Refer Slide Time: 50:58)



This looks like a dead end, but will move forward to understand that it is conceivable that operating systems can use the principle of locality of reference and use LRU like ideas without having to have the full overheads of implementing LRU.

So, they could be alternatives to the LRU as a page replacement policy. So, if there are going or if there could be efficient mechanisms for building page replacement policies that are LRU like, in that they allow the principle of locality of reference, to be used as a consideration in page replacement decisions, and prevent that worst case scenario from actually happening, then that would satisfy our requirements, and we will continue from this point in lecture 15.

Thank you.