Parallel Computer Architecture
Prof. Hemangee K. Kapoor
Department of Computer Science and Engineering
Indian Institute of Technology Guwahati
Week - 03
Lecture - 18

Lec 18: Virtual Memory (2)

Hello everyone. We are doing module 2 on memory hierarchy. Continuing the lecture on virtual memory. Okay. So we are discussing page fault in this lecture. So page fault occurs when the page is not there in the RAM and that page needs to be brought from the main memory. Okay. So here as the virtual memory concept goes, objects reside in the disk as well as in the RAM and the space is divided across by pages. So when you divide the RAM or the disk, we the unit of division is called a page and when you don't find the page in the RAM, it is called the page fault.

So when the page fault occurs, we need to take time to handle it. So it is going to take longer to handle and hence it can't be done in the hardware. We give the charge of handling the page fault to the operating system. And why? Because page fault is going to take several clock cycles and while the page fault is handled, that is you go to the main, you go to the disk, bring the page, load it in the RAM somewhere, the processor can be scheduled to do some other process. Okay.

So another process gets scheduled when the operating system handles the page fault. And analogously if you think of cache miss, cache miss is the analogy of a page fault, right? But cache misses are handled in the hardware and page faults are handled in software. In case of cache miss, if we have an in-order processor, then we can stall that particular instruction until the cache is updated with the data item. But if we have an out of order processor, then we can schedule other instructions or other threads can be scheduled while the cache miss gets handled by the hardware. So now during page fault, our task is to get the pages from the disk. Okay.

How do you know that there is a page fault? When we refer to the page table using the virtual page number, the valid bit tells me whether the page is present or not. If the page is absent, we give control to the operating system to go and bring this page. Now how does the OS know where the page is sitting on the disk? So the task of the OS is to find the page on the disk in the next level of memory or rather disk or secondary storage, whatever we call it. So locating the page is an important task the OS has to do. And if you see the virtual address, virtual address space of any program starts from zero and

may go up to some number. All right.

So these numbers are same across all processes. And if I see a virtual address, I can't identify where it will be sitting on the disk. So the virtual address or the virtual page number tells me nothing of the location of the physical page in the disk. Given the virtual page number, if it hits in the page table, I know the physical page number. But if it misses in the page table, we do not know where it is sitting in the disk. All right.

So for this, the OS is creating extra space, all the, all the pages which could not sit in the RAM and we said that they will sit on the disk. So where do they sit on the disk? They sit in a special space called the swap space. Okay. So a memory which cannot hold all physical pages of the process will spill over the remaining into some location of the disk called the swap space. Now how do I identify the location of the page inside the RAM? We need to keep additional mapping of the virtual page number to the disk location. Right? So this has to be kept and the OS keeps this because the OS knows which part of the swap space is used for which process. Okay.

So we need a separate data structure which will maintain the mapping between the virtual page number and the disk location. So when additional mapping is also required. The first mapping was virtual page number to physical page number. Now I need a virtual page number to the disk location based mapping. You could maintain it separately or as part of the page table.

So if I maintain it, maintain it as a part of the page table, then how do I manage this storage? If you recollect, the page table has got the valid bit. If this valid bit is 1, you get the physical page number. If this valid bit is 0, it says that the page is not present in the RAM. So can I use this space here? Right. I will use this space to tell me the disk location. Okay. So this is how we can integrate the two informations together.

Whenever the V is 0, the address is giving me the disk address. If you want a separate data structure, then even that can be maintained as a separate storage element. Okay. And as we know that pages in the disk and pages in the main memory have the same size. They have to be because that is the unit of transfer. Then we are going to use LRU replacement policy in the RAM.

Whenever a page, new page is brought into the RAM, you can use LRU to replace that page with a new page. And the victim page, that is the page which is going out of the RAM, is moved to the swap space. Okay. So if you are removing a page from the RAM, keep it back to the swap and bring the new page from the swap space to the RAM. So that's how a page table is managed. You have valid bit, so tells the page is present.

Valid bit is 0, it will tell the disk location. OS takes care of bringing the page back from the swap space to the RAM, evicts a victim from the RAM to the swap space. So now we are talking of page table scalability, which depends on the size of the virtual addresses. Okay. So in my previous example, my virtual address was 32 bits, and it was handling 4 kilobytes of pages. And therefore, the page table had 2 power 20 entries, leading to a 4 megabytes of storage.

To store the page table for 32-bit virtual addresses, we require 4 megabytes of storage and 4 megabytes for one process. And you imagine so many processes run in an operating system. So so many processes, everyone will need a 4 MB storage. So if I increase the virtual page, sorry, virtual address size to a 64-bit address. Okay. so you. look at the scale now. So 64-bit virtual address will need 2 power 52 entries, and you can imagine the size of the page table for one process.

Of course, there are optimizations present which limit, which limit the virtual addresses. Even if you have a 64-bit address space, you will limit it to some number of locations. We can go for segmentation, inverted page tables, multiple levels of page tables, or even paging the page tables. Right. So these are optimizations which you will popularly learn in an operating systems course. So we will not go into the details of this.

So how do I access a page table? The page table, as we said, is kept in the main memory. Okay. So this side is the RAM. So inside the RAM, we have the data items and the page tables. So here I have drawn three processes, p1, p2, p3. The blue process has got, this is the page table of the process p1, and that's the data for process p1.

So if this process p1 wants to access something, so process p1 will generate a virtual address. This virtual address will be translated to a physical address. Probably this physical address will be something here. Suppose this. Okay. It says that go to location A100 in the RAM. Now for doing this translation, I need to access the page table to do this translation.

The VA to PA mapping is stored inside the page table. So what did we do? When an access came from P1, first go to the page table, bring back the entry, translate the virtual to the physical address, and then go to the RAM to read it. So I do one trip for reading the page table, and once I get the translation, I do second trip to access the data. Okay. So here I am going to do read the translation entry. Given a virtual address of P1, first go to the page table, read the translation entry, come back, and then go again to the RAM to access the data.

So we are going to make two trips to the RAM consuming twice the time to do the access. Okay. So how do I improve on these two trips? Okay. So there is a concept called TLB or the Translation Lookaside Buffer which saves these two trips which we are making to the RAM. Again it follows the principle of locality of reference that is if I have used a VA to PA translation for process P1, I might reuse that same translation again and again. Right? So there will be spatial and temporal locality in these address translations. So can I construct a cache out of it? Right. So we would have a spatial and temporal locality among these address translations.

And therefore, we, most of the modern processors use a special cache to keep track of the most recently used translations. The most recent translations are kept somewhere as a cache and that cache is called the Translation Lookaside Buffer. So this buffer is going to keep the most recent translations happening so that you don't need to go to the main memory for accessing the page table. Okay. So TLB is a cache. Right. So it looks something like this.

It has a valid bit to tell whether that entry is valid or not. It has a dirty bit saying that in case you have modified the particular physical page, when this entry gets evicted, that physical page needs to be written back to the swap space, the dirty bit is there, then there is a reference which which says that this particular entry was referenced. Okay. So these are the three extra information bits. Now as I said this is a cache. So when there is a cache, you need a tag because the TLB is supposed to be of small size.

If you recollect page tables of 500,000 entries of that order and we could afford to index it using the virtual page number. But here I can't afford to do that because this being a cache which is small in size, I am going to use the virtual page number but not the full of it. Right? So the VPN will be used to index into this but I am not going to use the complete 20 bits of the VPN. I am going to use only part of it to index. The remaining part is kept as the tag in the TLB. Okay.

So if I integrate this with the page table, this is how it will look like. The VPN virtual page number comes, it gets indexed here. We will check for valid or not. If it is valid, then this is the PPN which we are interested in, the physical page number, go read the data. So this is where the data will be kept, go and read the data.

Similarly, if it happens to be a 0, then the TLB does not have this entry, so you have to go to the page table. So in case of this, this is called a TLB miss, you will have to go to the page table, then find out the physical page number from here. So either it is in the RAM or it is on the disk. Right. So these are the two options available here. So depending on the physical page address or the disk address, we will access the data.

So once we access this data, we will bring in this entry and store it somewhere here in the TLB because this was a recently accessed data item. Okay. So you go to the TLB, you find the physical page number good. If you do not find that is the data entry is not valid, go to the page table, access the correct VPN, VPN to PPN mapping, bring that mapping into the TLB. Okay. So this TLB is acting as a cache and hence it has a tag and the data. What is the tag? Tag is part of the virtual address, a portion of the virtual address is the tag, portion of the virtual page number is the tag.

And what is the data? Data is the physical page number. It has a valid bit, dirty bit, reference bit used for corresponding purposes and then if there is no tag match that is you have a TLB miss, then go to the page table. Page table gives you the new physical page number. In case the page table does not have the page number, you will incur a page fault in which case invoke the operating system to do this. And once this page comes into the RAM, update the page table with the corresponding PPN entry. Right.

So, once you access a new page which is not in the TLB, bring that entry and populate the TLB entry with that translation. So a TLB misses, you did not find the translation in the TLB cache. So what to do? You have to load the new entry into the TLB. Once you load the new entry, mostly it will be full, so you need to remove a victim. So when you remove a victim from there, you need to check for its dirty bit in case that translation indicates that this entry is dirty, meaning the physical page to which it is pointing has been modified.

So we need to write this page back to the disk. So this page has to be written back to update the page table, update the swap space and then this TLB is small because we want faster searches and it is mostly fully associative. To make the search fast and to reduce on the cost, we do not go for LRU, we simply go for a random replacement policy in case of TLBs, so that the search and replacement is quickly done.

Okay. So hopefully the TLB is now clear. So TLB sits between the translation from the virtual address to the physical address. So now we will see the interaction of the TLB with the cache. The pink one is showing a virtual address, the same 32-bit virtual address which has the VPN, virtual page number and the page offset. When we come with this virtual address, we first go to the TLB and then go to that particular location and search for the physical page number. In parallel, I have also shown the cache in which the actual data is there. Okay.

So now what we do is translate the virtual address to the physical address first. So this becomes the physical address. All right. So I will use the VPN. So this is a valid bit, so

it turns out into a hit. Okay.

So I will say this was a TLB hit in this case. I got the physical page number, copy the physical page number here and construct the physical address. Okay. So using the pink box of virtual address, I have constructed the orange box which is the physical address. Now this physical address is the new 32-bit address and this will be now used to read the cache because we are interested in the data. The processor gave the virtual address, we went to the TLB, did the translation, got the physical address. This physical address has to be further broken down or readjusted to find out the index of the cache. Right?

So depending on the cache index and tag bit information, I am going to translate this orange into this blue which is saying we simply reassign the partitions. So there was a partition here, now I have put the partition here and here. It is the same data. Okay. So the bit pattern is the same. Everything I have partitioned it at these locations to decide the byte offset, the index and the remaining other tag bits. Okay.

So now this is familiar to you. The cache tag is here, this is the index. So we will use this index to go in the cache somewhere here, then read the cache tag, bring it there and then compare the blue cache tag of my required address with this. You compare it and then declare a hit or a miss. So depending on that you are going to access this data.

So if hit then read the data. So read the data in case of a hit. So that is the interaction between the TLB and the cache. Okay. So overall how does the virtual memory gives us protection? So it allows sharing of the same memory across multiple processes. Right? So the same RAM is shared by multiple processes. It gives me protection that is every process can only access its own locations and not other processes' locations.

So it provides protection. It guarantees that one process cannot write into the space of another process. So same thing as protection. The right access bit helps us in doing this which tells that which process can write in which particular page. Then, again same thing as I cannot write in another processes space, I even cannot read another processes data.

So it reads across different processes is also not permitted. Okay. Now how does this work? How do you do this? By using extra bits that is called access write bits. So every TLB entry comes with these access permission bits which tell which process has got access to this particular entry and which does not have. Next question is do we have different TLBs for every process like we had different page tables for every process. Okay. So it is a design decision but mostly we have a single TLB. But then how do you identify that a particular entry belongs to which process? So it is easy you can simply

include the process identifier with every TLB entry saying that this particular VPN to PPN translation belongs to process P1 or process P2 and so on.

The same thing even occurs when I have one TLB, we have this entry which was used by process P1. Now this process got context switch or it was removed, it finished its work and now this process gets removed and a process P2 is scheduled and it wants to reuse this entry. Right. So even this is possible by using the process ID and other optimizations. We can reuse the TLB entries across different processes. And of course there are other solutions which you can study in some other operating systems course. Okay.

So that was an overview of the virtual memory. So what was the virtual memory? It was giving me an illusion of a large address space available to every process. It guarantees protection of data space across different programs so they cannot access each other's memory. Okay. So we get an illusion which is very important of a large memory. We get protection that is every process can access only its own data. It is managed with the help of secondary storage, so you need the disk space, you need page table, you need TLB, you also need the operating systems help to handle the page faults. Okay.

So it is handled using the OS in the software. We need secondary storage, page tables and TLBs which we discussed and all this work which we have been seeing, lot of things are happening in the background and they are completely transparent to the user program. So the user program doesn't even know the size of the RAM which is present in his or her system but still is able to run processes which are much larger than the physical capacity of the RAM. So that is the beauty of the virtual memory management system and with this we finish this topic. Thank you.