Parallel Computer Architecture
Prof. Hemangee K. Kapoor
Department of Computer Science and Engineering
Indian Institute of Technology Guwahati
Week - 03
Lecture - 17

Lec 17: Virtual Memory (1)

Hello everyone. We are doing module 2 on memory hierarchy. This lecture is about virtual memories. Let's first understand the concept. We had started with this example of writing a term paper referring to the library books when we started with the cache hierarchy. We will continue the same one right now.

So recollect that we are writing a term paper sitting at our desk and we have collected a set of books to refer to while writing the term paper. Okay. As and when you require new books, you will go to the shelf and bring the new books. If your desk space falls short, you will want to replace some of your earlier collected books back to the shelf and bring newer books.

Right. So this was the concept of the cache memory. All right. So now your term paper demands more books and your shelf or rather your library doesn't support those books. So what you want to do is you want to access more books which are not in your library and therefore you take special permissions and visit libraries apart from your own college or institute.

So you will visit other libraries to bring books. All right. So this connection here is the cache connection. Right? You are bringing books from the shelf to your desk and then when your main memory, the so-called shelf falls short, you want to go elsewhere to other libraries. So this is a library somewhere else in the town, another library somewhere else in the town or another town and so on.

So you can refer to several other libraries and then bring books or issue books so that you can complete your work. Okay. So what is this idea of accessing other libraries tells us that as a person, I have an access to an infinite collection of books because as and when I need, I can go fetch the book and complete my task. Okay. So bringing back the same example to the memory hierarchy, the bookshelf is now referring to my RAM and it is of limited size.

Every program may fit or may not fit in the RAM because your physical memory size is

limited. Now if your physical memory size is limited, the process which can run using this physical memory also has its own limitations because it cannot fit all the instructions or data. Essentially the complete process state cannot be saved in the given RAM capacity. So should we stop there or can we do better? Right. So similar to the example that we went to other libraries, we go elsewhere to search for storage or rather access the storage. So this is done using the secondary memory or the disk, which helps us to logically expand the RAM.

So my RAM is small, but I use the secondary disk space to access those locations and essentially expanding my RAM capacity. Right. So we can move data between the RAM and the secondary storage so that we will be able to access that as well. And what decides this movement of data? The access patterns. So the data items which you want, you're going to bring them from the disk to the RAM, access them.

And when you require something else, you might want to replace the earlier brought data items. Okay. So essentially this virtual memory is giving you an illusion of a very large storage available with the help of the secondary memory. Okay. And what all things does it give me apart from extra storage? It gives me safe sharing of the memory among multiple programs because the RAM is shared by everybody and every process is going to store its data into the RAM. So is it possible that another process might read my data? Yes, it is definitely possible, but the virtual memory concept protects us from this because it puts a demarcation about what portion belongs to which process. Right. So it gives me a safe sharing of the memory.

It allows a single program to exceed the size of the RAM. We discussed this as a motivation that my program size can be bigger than the RAM size. Okay. And how does this help? This helps me in the programming that I don't need to write programs which are small enough to sit in the RAM. Right. My programs can be bigger than the RAM size. And if earlier when this concept wasn't there, programmers thought that given a limited capacity RAM, I will either write small programs or I will split my program into smaller programs or modules which will be loaded turn by turn to be executed. Okay.

Essentially, the memory management or virtual memory management is giving this to us in a transparent or seamless manner. Okay. So we don't need to split the programs and execute them in modules, but they will be done automatically. So this feature of bringing in small modules and executing is done by default by this logically big virtual address space. Okay. So apart from giving more memory, it is helping me to translate the program address from the physical, between the physical and the virtual addresses. As we always, as we discussed earlier, the virtual address is generated by the processor and eventually it will be translated to the physical address when you access the RAM. Right.

So this is when you access the RAM, you need the physical address and the processor is going to generate the virtual address. So this address translation which will happen will also help in saving or demarcating the different processes. That is, I have these two. So this is the physical address of process P1, that's the physical address of process P2. And even if the virtual addresses would be the same, that is I have VA equal to 1 for process P1 and I have the same virtual address for process P2. Right?

So these two would map to different physical locations and the memory management unit will make sure that P1 cannot access P2's data. Right. So if they want to go and read or write the data, this is not permitted. Okay. So this address translation enforces protection of programs from each other's address space. Okay. So how does a virtual memory look like? So this picture shows that this is the virtual address space for a process P1. Okay. So virtual address space of process P1 and as you see, dimensionally it is very large and this physical address is the RAM.

So that this box, grey box is the total RAM size I have and if you look at this left hand side box, the P1's virtual address space, it is rather big storage which is demanded. So I cannot cater to this complete storage in the small size RAM. If you see only these green colored pages are presently available in the RAM, whereas the pages which are red in color or orange in color, these are not loaded right now. So they are not in the RAM. So where are they right now? Okay. They are sitting in the disk.

So they are sitting inside the disk. So the disk and the RAM together help me to access this complete virtual address space of process P1. Okay. So if we extend the same concept, you would have the virtual address space, VA, virtual address space of process P1. Similarly, a virtual address space of process P2 and there will be several such processes running and all of them will essentially be mapped somewhere into the RAM. Okay. So if this is the RAM, suppose P1 is able to get this much share, P2 got a slightly smaller share of the RAM, P3 got even a smaller and so on.

And remember that P1 might need several locations, but it only gets this much share. So where is the remaining share? The remaining share will be stored and accessed from the disk like this. Okay. So these demarcations which are present in the RAM, the address translation unit which will translate the virtual address to the physical address will also check for the boundaries that is whether a process is trying to access another process's data or not. So it prevents this from happening. Okay. So some terminologies related to virtual memory.

In the case of cache, every block which we brought was called a cache block or a cache

line, whereas in virtual memory, a block is called a page. And if you miss in a virtual memory, that is if that particular page is not loaded in the RAM, we call it a page fault. A cache was called a cache miss, here it is called a page fault. Okay. Then processor produces the virtual address. This virtual address has to be translated to the physical address and with the help of both hardware and software.

It is not purely done in hardware. Hardware as well as software both help to generate the translation. Okay. So the process of doing this translation is called address mapping or address translation. Okay. So we will understand this further. So this is the virtual address.

A virtual address now consists of a virtual page number and a page offset. So I will call it a VA because it's a virtual address. So this virtual address has now been given to us by the processor consisting of the page number and the page offset. So the number of bits in the page offset here, so these bits here in the page offset, they tell us the size of the page. Right now we can see we have 12 bits, so we will have 2 power 12 bytes of information which makes it a 4 kilobyte page.

Right, so that is the page offset. Then VPN, that is the virtual page number. If you see these bits, so these many bits are used to generate the page number. So we have several virtual pages accessible and this number of pages will definitely be bigger than the physical addresses. That is the VPN, that is the virtual page numbers.

The number of virtual pages will be always greater than the number of physical pages or physical addresses because that's our requirement that our RAM is smaller and we want a bigger virtual address space. Okay. So this is the basis that my small RAM is now accessible to many more virtual pages which gives me the illusion of having an essentially unbounded amount of memory. Right. So more virtual pages fitting into lesser physical pages which gives me an unbounded memory available to a processor. Okay. Right. So the virtual address has a virtual page number. It goes through the address translation unit and generates a physical page number.

If you see the physical page number, number of bits representing is less here. Right. So if we do some calculations, look at the page offset. The page offset gives me the page size which is here. So we have 12 bits, so 2 power 12 that is 4 kilobytes is my page size. Okay. Then number of physical pages, how many pages or what is the size of my RAM size. Okay.

So the number of physical pages which can be accessed is equal to the number of bits here. So we have the physical address of 30 bits. So 30 bit physical address and 12 bits

of offset is gone. So 30 minus 12 gives me 18 bits. So we have 2 power 18 pages available in the RAM and each page is 4 kilobytes wide which gives me a 1 GB RAM.

And what is the size of the virtual memory? If you go to the virtual memory, you have the virtual address as 32 bits minus the page offset which is 12 bits which becomes 20 bits to access the VPN. So VPN is 20 bits, so we have 2 power 20 virtual pages. And if you see, multiply this with the 4 kilobytes per page, you will get a 4 GB of virtual memory. Okay. So we can run a program which is 4 GB wide using only 1 GB size RAM. Okay. So this is how the virtual memory is used for a single process. Okay.

So now consider that we have these two processes, process A and B. A has got 4 pages A1 to A4, B has B1 to B4 and they are randomly loaded in the RAM. Right. Not all pages are there. If you see the color coding, you will know that certain pages of each process are there in the RAM and the remaining could be in the disk. So when a process A wants to access suppose A4 and A4 is not there in the RAM, what will we do? Okay. So this is called the page fault.

So when a page fault comes, the RAM does not have the data, we have to go to the disk first find out where the page is, bring the page, put it somewhere in the RAM and then start reading that page. Okay. So this is a task about finding where the page is on the disk and bringing it back. Again this is going to take millions of clock cycles because the disk is slow, even the RAM is slow, the disk is even slower. So this is going to take lots of time and when we need lot of latency to do something, we want to do more work in one trip. Okay. So the idea is if the disk is very far, once I reach the disk I will bring lots of data and hence the page sizes are rather very large. Okay.

And then we also need to identify where each page is sitting in the RAM that is where is the orange color page and where is the blue color page. So each process will have a table which will tell the location of this A1 inside the physical memory. Suppose it is sitting at page 6. Okay. And here the page table for B, process B, so B1 suppose it is sitting at page B3. All right. So these are called the page tables which are private to every process because we do not want A to access the contents of B.

So the process A should never know where the pages of process B are because it is not permitted to access that. Okay. So that is the concept of the page table which helps every process to identify the location in the RAM. All right. So we discussed that the sizes are large because the disk is far and it is going to take more time to load. So what are the typical page sizes? Normally 4 KB or 16 KB but certain modern systems also have 64 KB pages.

However embedded systems have tiny pages of 1 KB. So depending on your system requirements the page size can be adjusted. Right. Then when you bring the pages into the RAM, look at the right hand side RAM, where do I place the fourth page? Suppose I bring the fourth orange page, where do I place this page? Is there a fixed position like a direct map cache? Is there a set of small locations like a group assignment that is an associative cache? So that is the question to answer. However because the accesses are random, the requirements of every process may be different. We end up saying that let me go for a fully associative design and have no restrictions on where I want to place.

So this RAM is acting as a fully associative cache. Whenever a page comes, it can be placed anywhere in the RAM. Okay. So it has to be fully associative so that we can reduce the page faults. As we discussed that similar to the reduction in cache misses, if you recollect, we said to reduce cache misses increase the associativity. So we are going to the extreme in the case of virtual memory. To reduce the page faults, I am going to make the RAM a fully associative cache for the disk. All right.

So this page fault is handled by the software because when the page comes in, we have to decide where to place. Definitely if there is a empty location in the RAM, I will place it. But if orange page comes, I would rather want to keep the new orange page here closer to the other orange pages and not somewhere there. Right? I do not want to put it here, but I want to put it close to the other orange pages or if your RAM is a banked RAM and so on. So there are many algorithms which decide where to place a page after it is brought in. Okay.

So that is why we need help from the software to do this. And the operating system helps me to assign the physical addresses because the OS knows which area belongs to which process so that there is no overlap among the process states of different processes. All right. Then the other concept is that write through is not used. What is write through? Whenever a change is made in the cache, that same write goes all the way to the next level. In the case of virtual memory, what is your next level? Your next level is the disk and any change in the physical memory's page, you do not want to go all the way to the disk and write because it is going to take millions of clock cycles to do this.

Hence a write through policy is not used. Finally, every process has got its own page table to ensure protection and quick access. Right. So this page table which we are discussing is a fully associative page table. And when it is fully associative, the drawback of a fully associative or a set associative design is that the hit time increases, hit time is searching time. So my search time increases because of a fully associative thing.

So I will give you an example. So let us follow the color code, the orange is the physical page number and the green is the virtual page number. Okay, recollect your classroom, this is your classroom where there are tables and chairs and these are the table numbers, right. So there is identity associated with each chair. So we have table number 1, 2, 3.

So these are seating positions. And when you enter the classroom, you are virtual pages, your students are treated, can be treated as virtual pages in this analogy and they will sit anywhere in this classroom. Okay. So this is the setup. We have the physical chairs given by numbers and the green are the students sitting anywhere in the class. Now I ask you how will you search for a given student? Okay. So if I have a fully associative table, because you need a mapping table, this this picture has only 12 chairs, but in a main memory, you will have around 500,000 pages, right.

So that is the scale of which we are talking about. So if I have a fully associative page table, which is catering to 500,000 entries, then how do we do this? Okay, coming back to this example, I have generated the small table, which is catering to the fully associative arrangement. How does it, how do we read this? You read the first line, the first row is saying B8, right. So the student B came and sat in seat number 8. So we made one entry into this table. Then the student L came, sat in row, sorry, sat in the table number 2 and so on.

Then eventually some student K came, then the student sat in seat number 3. So this is the order of arrival, right. So for example, if I maintain the arrival order, then I will make a table like this. If a new student comes, I might want to remove the student and put a new student here, okay.

So the arrangements are dynamic and keep changing. So this is the page table, which is of fully associative nature. And if you want to search for student A, right, you do not want to go to the classroom and search, first you refer to this chart, find out the desk number and then go and address the student. So I have I have come here to search where is A sitting in this classroom. So I will start searching from here, right.

I will check B, B is not there. No, L is not there, C is not there. Then I will say okay, A is found, then you go check. So that is table number 4. So when you enter the classroom, you go to table number 4 and then identify the student. Okay. So here we were lucky to find it in the fourth entry. But suppose we were searching for some other student who was sitting at the last desk or who came very late.

So if I had to go all the way to the end, you imagine the search time. So the search time

increases if I have a page table which looks like this. Now what to do? Okay. So we can have a solution for this saying that instead of searching for the student names that is the student B sitting ? or is the student A sitting here?, A sitting here, I could index this table using the student's name. So I will change this table to look something like this, where the green ones that is the student names are in alphabetical order. So when I come looking for person A, person A will be indexed using A.

If I come for D, D indexes to the fourth location. So as soon as I have the D, I can use a formula to find out its location. Right? If it was D, depending on the size of the table, you can use D to index into the table. So this example shows that we can index it using the virtual page number, the greens are my virtual page numbers. So is this easy? So when I come with the student name, the student's name is used to index into the table and then we immediately in one step get the seat number of the student. Okay. So the page tables should not be fully associative, although the RAM is fully associative, but the page tables searching has to be fast and hence we will index it on the virtual page number.

Once we index it on the virtual page number, you will realize that storing the green entries is useless because we are going to use it to index. So I will come here to search for D and I know I have come here to D, search for D and D must map there, it cannot map anywhere else. Hence this portion I do not need to store. So my page table only reduces to the orange entries.

So this also saves on the space required to store the page table. Okay. So this slide summarizes the discussion. So your fully associative placement would cause overhead of the search time. Right? So it is it is rather impractical to search for so many entries into the page table. Then the page table is as we discussed indexed using the virtual address. You have to use the virtual address to index so that you can quickly find the location of the corresponding physical page number.

So you can use the VPN, you use the virtual page number to index into the page table and immediately you will get the physical page number. Okay. So that translation becomes very quick. Every page table belongs to a particular program. So every program has a different page table.

So these page tables are again kept in the main memory. So we also need to know where is the page table. If I am talking of process A versus process B, A needs to know where its page table is kept. It is there in the RAM but where. Okay. So that is maintained in the page table register which is also part of the process state of every process. So the page table register puts the address, the physical address of where the

page table is kept.

Then inside this page table there are several entries of physical pages but certain pages may be absent. That is, if this is my physical, if this is my page table, it is indexed using the virtual page number and the entries here are storing the physical page numbers. But this page may be present, this particular virtual address is not present in the RAM, this could be present in the RAM and so on. So we need to maintain some information about this.

Hence, the valid bit is used here. If the valid bit is 1, it says that the particular virtual address mapping to a particular physical address is present in the RAM. That is that page is there in the RAM. Otherwise we treat it as 0. So 0 means the page is absent and then this leads to a page fault. Okay. Inside this we are indexing using the VPN and once we index using the VPN the complete virtual page number is used and hence I do not need a tag.

So no tag is required in the page table because the complete address is used. We need a tag only when I am using part of the address to go somewhere and the remaining address bits need to be compared as part of the tag. But page table uses the complete virtual page number and hence no tag is required for the page table. So we can see an example. So this is the same virtual address, a 32-bit virtual address, 4 kilobytes page and this is the page table.

I have drawn a small one. So this is a representative page table. It has got a valid entry. So 0 or 1 will decide whether this entry is valid or not and it gives me the physical page number. So the virtual page number of 20 bits comes here and it is used to index the page table. This is used for indexing. So we will go to suppose whatever is the combination of these 20 bits that will be the row number in the page table which I will access.

That is the meaning of index. So once you reach to that particular row number, you have to check the valid bit. Okay. Is this valid bit 0 or 1? If it is 0 means the page is not there in the RAM, you have to handle the page fault. If the bit is 1, you will get the physical page number. Now the physical page number in this example consists of 18 bits.

So we move those 18 bits out and then construct the physical address. And the page offset here is same as this one. Okay. So if we put some numbers and see the sizes, the same example is continuing. So I have a virtual memory of 4 GB size, a physical memory of 1 GB. How many entries are there in the page table? This is the page table. How many entries are there in this page table? As we discussed, we are going to use these 20 bits to index inside that and hence I am going to have 2 power 20 entries in the

page table which is close to 1,000,000. Okay.

So I have 1,000,000 entries in the page table. Now what is the size of this page table? I am storing 18 bits here which is the physical page number and I am storing 1 bit here for the valid bit. So total 19 bits are stored but 19 is an odd number and we normally deal with multiples of 2 or powers of 2. Hence the page table normally stores up to 4 bytes of information per entry because we will also have some other fields getting added to the page table later on. All right. So hence normally 32 bits are stored per entry and you would actually need 2 power 20 into 32 bits worth of storage to keep the page table.

We will need some extra bits also so these 32 bits will be useful for that. Okay So that is how one page table looks like and every process will have this big page table. So with this we finished part 1 of the virtual memory. We will deal with page faults and some other concepts in the next lecture. Thank you.