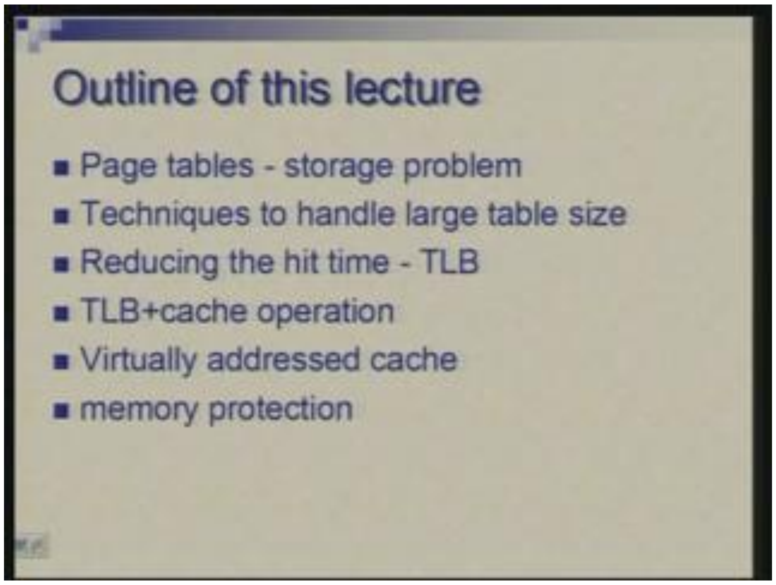
Computer Architecture

Prof. Anshul Kumar
Department of Computer Science and Engineering
Indian Institute of Technology, Delhi
Lecture - 32
Memory Hierarchy: Virtual Memory (contd.)

We have discussed virtual memory as a mechanism to give an illusion of a large addressable space to a program with an attempt to keep the performance as good as a physical memory. Today we will continue with that. We have seen that a page table is the key mechanism in organizing a virtual memory which helps you in translating virtual addresses to physical addresses.

(Refer Slide Time: 01:25)

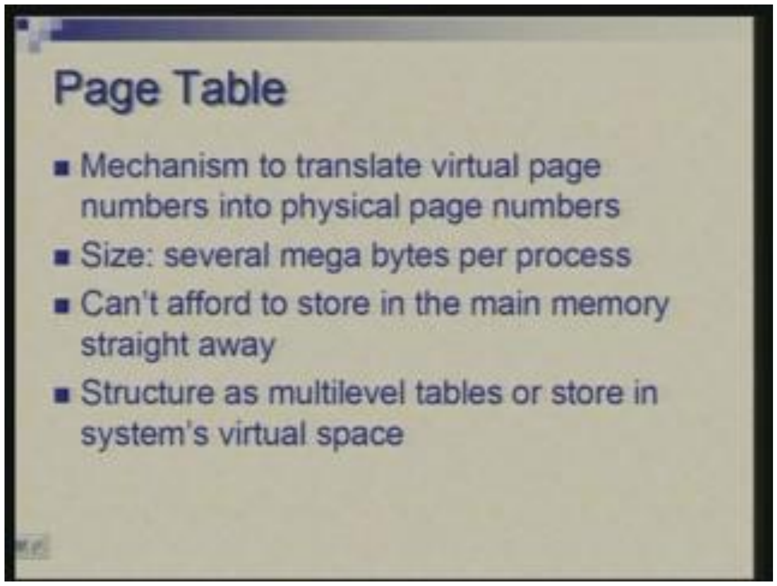


Outline of this lecture

- Page tables - storage problem
- Techniques to handle large table size
- Reducing the hit time - TLB
- TLB+cache operation
- Virtually addressed cache
- memory protection

We notice that storage of page tables was a big issue because they are large in size. We looked at that and we will continue on that topic little further to see in little more details. We will introduce another structure which is present in hardware called TLB or translation look aside buffer whose role is to speed up the access. We will see how TLB works together with cache. We will also look at the possibility of addressing cache by virtual addresses instead of physical addresses. And finally we will look at the role of virtual memory organization in memory protection.

(Refer Slide Time: 02:18)



Page Table

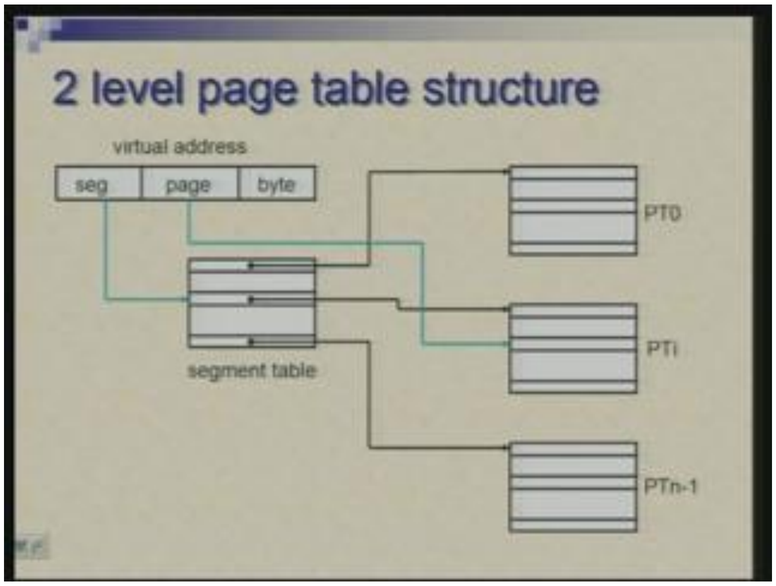
- Mechanism to translate virtual page numbers into physical page numbers
- Size: several mega bytes per process
- Can't afford to store in the main memory straight away
- Structure as multilevel tables or store in system's virtual space

We have seen that page table is basically a lookup table where given virtual page number you look up in the table and you will find out the physical page number corresponding to it. That happens if the page is present in the physical memory. But in case page is absent we get page fault and then we expect that this table will give us an indication of where the page is stored on hard disk. So either directly disk track and sector indices would be there in the page table or it will point to an area from where these could be picked up.

So, in concept it is very simple but the problem comes up because of the large size. We noticed that it could be several megabytes and there may be many processes simultaneously active; you would require a separate page table for each of these. And then you do not want to fill up the entire physical memory with just page tables. So we discussed a few possibilities and we will elaborate on that today.

Two of the possibilities which I mentioned last time; one was a multilevel page table structure that is it is not a simple flat array but you structure it as a hierarchy; it could be two levels or larger levels; in a simple case of two level we saw that it basically means there are several tables in a tree like structure and that helps in reducing the requirement of the main memory. You also looked at the possibility of keeping the page table in virtual space itself which is much larger and therefore the problem is eased out.

(Refer Slide Time: 04:06)



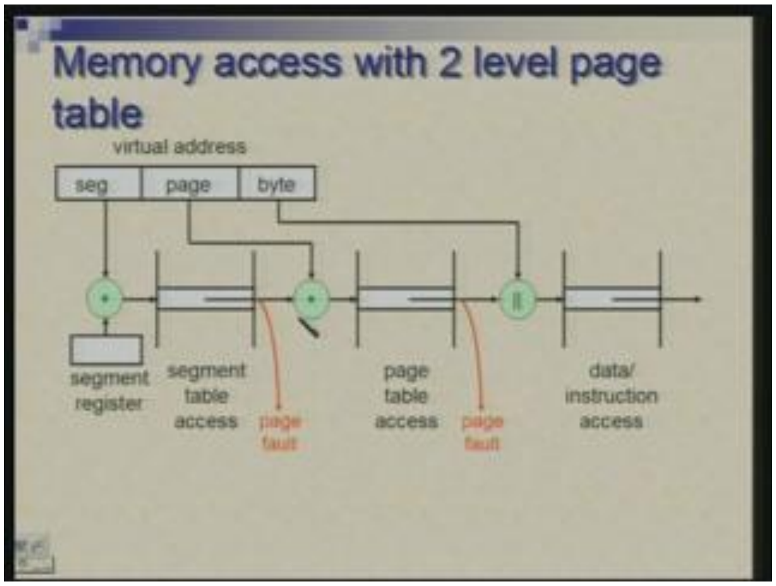
2 level page table structure

So, coming back to the organization of two level page table you have basically the entire virtual space divided into segments. So, for example, suppose you have 1 million pages then you can think of it as thousand segments so each segment containing thousand pages so something of that nature. Then for each segment you could have a much smaller page table which will point to those thousand pages or which will have information about those thousand pages. So essentially we will have thousand such small small tables and depending upon your requirement you will only have a few of these in the physical memory, you do not have to have all of them and that is where the saving comes. So you have these existing somewhere but at any time you need to maintain maybe a limited few in the main memory.

So, to keep track of these there as to be another table which you call as segment table and the entry in the segment table will point to the starting points of these page tables. So it will also have indication valid bit and so on; indicate whether that table is actually present in the memory or not present in the memory; rest could be lying in the disk. So in its functionality it is the same thing as a page table. So you can call this division as segments and pages or you could call it level 1 and level 2 so idea is same.

The way it works is that the virtual address gets now divided into three parts; higher few bits will decide which segment it is, then one field will be referral to the page and then byte within a page. So using the segment field, you address this segment table and basically this leads you to appropriate page table; within that page table you index through the page number and then you get information about the corresponding physical page of that.

(Refer Slide Time: 06:23)

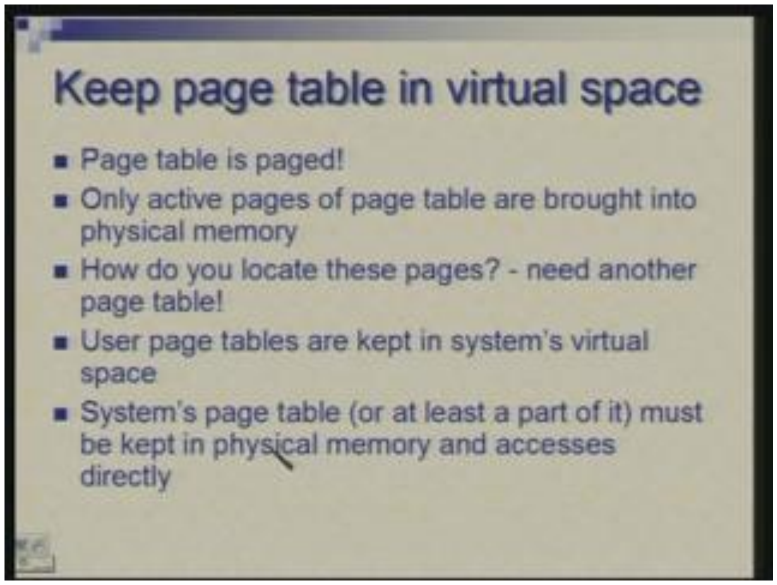

Memory access with 2 level page table

This can be seen more clearly with this diagram where I am trying to show exactly how the accesses would be made. So starting with the virtual address looking at the segment field you have.... imagine a register which we call segment register which has the starting address of the segment table, so, that starting address with the segment number has an offset, you had these two (Refer Slide Time: 6:52) you reach the required entry of the segment table. I am not showing a constant multiplying factor here. Suppose each segment table entry is 4 bytes then this number will have to be multiplied by 4 before you add it. So conceptually I am just directly putting it here.

The idea here is that segment register contains the address of starting point of the segment table and this tells which particular entry you want in that table. So two added with the appropriate weightage here leads you to the location of segment table which you want to refer to. So you make an access to the segment table. We are assuming that segment table would be in the physical memory; it is comparatively much smaller, let us say if you can manage in few kilo bytes it is not much of storage to dedicate to this.

So when you make an access to this entry it will either tell you that the corresponding page table is not present; we will look at that as a page fault or it will tell you where it is present. So now in case of a hit this address will be the starting address of the relevant page table and to that we can add the page number again with appropriate weightage here to get the exact location of the appropriate page table entry. So we look up that entry; now we are referring to let us say some ith page table and here again we might either be successful or we might find a page fault. If we are successful then what we are getting here is the physical page number of the word we are looking for or the byte we are looking for. So this would be.... well, I should say it would be...... okay.... yeah I think this is this is the physical page number concatenated with the byte address within the page, you get complete physical address which you can use to access instruction or data whatever you are looking for. So these are the, roughly speaking, three steps: One step is this (Refer Slide Time: 9:12) making one access, second access and third access.

(Refer Slide Time: 9:21)

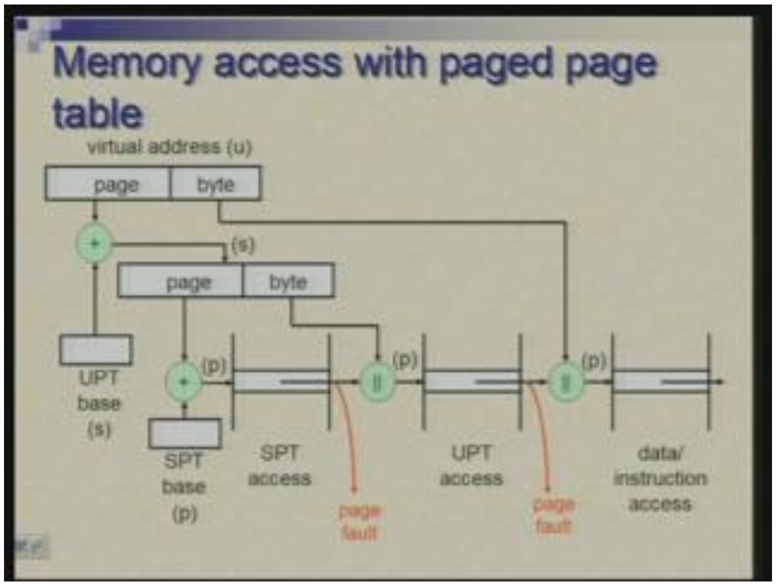


Keep page table in virtual space

- Page table is paged!
- Only active pages of page table are brought into physical memory
- How do you locate these pages? - need another page table!
- User page tables are kept in system's virtual space
- System's page table (or at least a part of it) must be kept in physical memory and accesses directly

The second approach which I mentioned was not structuring into multiple levels but allowing the page table to be fragmented. Think of page table also as consisting of pages. After all it is some storage area, some data structure and it will be spreading over several pages. So allow the paging mechanism to work on this itself. That means only a few relevant pages or a few active pages of the page table are present in the memory, the rest are in the disk. So, effectively we are placing the page table in virtual space and to avoid cyclic references we said that this could be in the system's virtual space, not in the user's virtual space but in the system's virtual space. So to access that we would need access to system's page table; we made an assumption that either that table would be in the physical memory or at least the relevant part of that table would be in the physical memory. So we will start with that assumption and again look at this mechanism and little more details.

(Refer Slide Time: 10:35)



Memory access with paged page table

So here we have two fields only. This is a virtual address and in the parenthesis you will notice I am putting (u) (s) or (p); so (u) stands for user's virtual space, (s) stands for system's virtual space and (p) is for physical space. So there are many addresses flowing in this diagram, so just to distinguish those I have qualified these with (u) (s) or (p). So starting point is virtual address which is in user's virtual space and it consists of page part, page number and the byte number (Refer Slide Time: 11:17). Notice that we are not dividing this into two parts. So the whole thing represents one of the many many many pages.

Now we need to make a reference to the user's page table to know about this and we have the starting address of user page table here in this register. UPT stands for User Page Table. So its base address is available in the register. But now it is not a physical address. Since you have decided to keep user's page table in system virtual space this is an address but it is in system's virtual space it is a virtual address. So this starting address plus page offset gives us another virtual address in the system space which we can again look upon as a page part and a byte part. So this sum we are getting is let us say again a 32-bit number. We interpret this again as a page number part and a byte number part.

So the page number is basically saying that it is a page of the system virtual space and if we have system page table base address, together we can find a pointer to the entry in system's page table. Now we are assuming that system page table's base is a physical address. So this physical address plus page number offset we get another physical address with which we access memory and this location (Refer Slide Time: 12:57) either tells us that there is a fault, what we are looking for is not present or it gives a physical page number and that physical page number with this byte gives us a complete physical address and what is this address this is the address of user's page table which we were looking for to begin with.

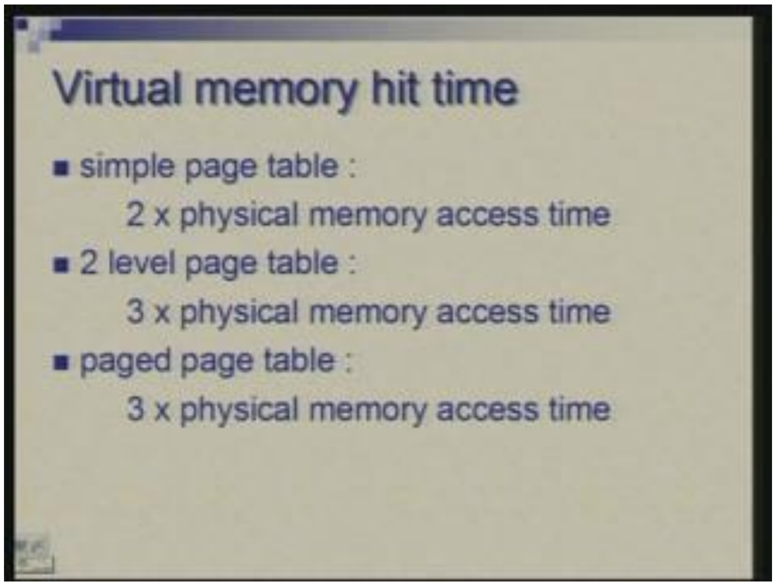
So actually this address which we have got (Refer Slide Time: 13:25) is the physical counterpart of this virtual address. So this virtual address has been translated by this portion into a physical address here. Now we make access to user’s page table and get a physical page number or get a page fault so this number together with byte give you another physical address. So this physical address is translated part of this virtual address. So you notice that there are two address translations going on; one in system space, one in user space. So this translation means one offset edition and a memory lookup so the first one is here; this is translation from virtual to physical in system virtual space and we have used system’s page table for that.

This translation which is basically that addition and this lookup (Refer Slide Time: 14:33) together are translating this virtual address into a physical address using user’s page table. So once you have that then you make a final access to data or instruction whatever the case is. So it may look a bit complicated if you try to look at it at a glance but if you go through it gradually and carefully basically they are nothing but two address translations.

Are there any questions in this; something which is not clear?
So, first translation is actually in this area. Here is a virtual address in the system space.
What is this address of?
This is the address of user page table entry which we are looking for. (Refer Slide Time: 15:30). So basically we want to take the virtual page number of user address and user page table base. but unfortunately this is a virtual address so we cannot use this to access memory so we need to translate this into a physical address by this activity and then we are looking up the physical memory to know whether the page we were looking for here........ whether this page is present or not is indicated here and whether this page (Refer Slide Time: 16:12) is present or not is indicated here. So this addition and the memory access here with concatenation of byte address; this part is one address translation and this is forming another address translation. But within as a, let us say as a sub-routine or as a next level function we have this itself translated before we apply here.
Had the user page table been in the physical memory then we would simply take this address and lookup and proceed. But because it is not in physical space it is in system virtual space we need a lookup in system’s page table.

So now suppose we use one of these two mechanisms then where do we land up?
Let us look at the hit time.

(Refer Slide Time: 17:20)



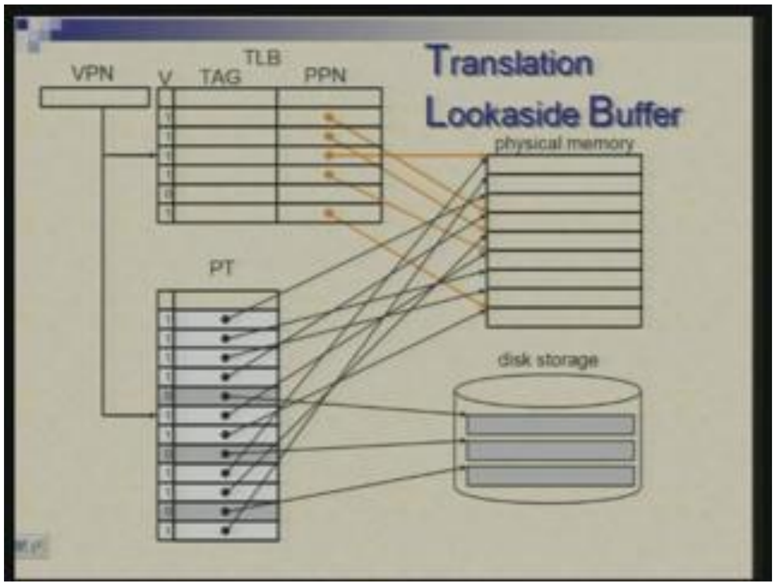
Virtual memory hit time

- simple page table :
  - 2 x physical memory access time
- 2 level page table :
  - 3 x physical memory access time
- paged page table :
  - 3 x physical memory access time

The miss penalty here is pretty large as I mentioned because you need to make an access to the disk. But hopefully with very small miss rate or page fault rate we can reduce the effect of that. But what do we do about the hit time which as you will notice here is something which is not very acceptable. In a simple page table structure suppose the page table was in physical memory still we will require two accesses to physical memory to read a word or to get an instruction. So the hit time even if everything is fine you have to have two memory accesses made. so straightaway we have reduce the performance by factor of 2 by going through this process, whatever you may completely amortize the miss penalty but the hit time is going to hit us; it is two times the physical memory access time.

On the other hand, in a two level page table as well as paged page table you know both these both these pictures (Refer Slide Time: 18:36) you notice that there are three memory accesses which are made in the best case: one memory access, second memory access, third memory access assuming that there are no page faults. Here also this is memory access, this is memory access and that is memory access. To get to read one word you need to make three accesses. So in reality things are pretty bad. So what do we do for this? The solution for this is that we keep a buffer, have a special memory as part of the CPU which keeps track of some of the recent address translations which we do. So it is like a cache of the page table. So some recently accessed entries of the page table can be preserved for future reference. Again the idea of locality is what will make this successful and this is called a Translation Lookaside Buffer.

(Refer Slide Time: 19:44)

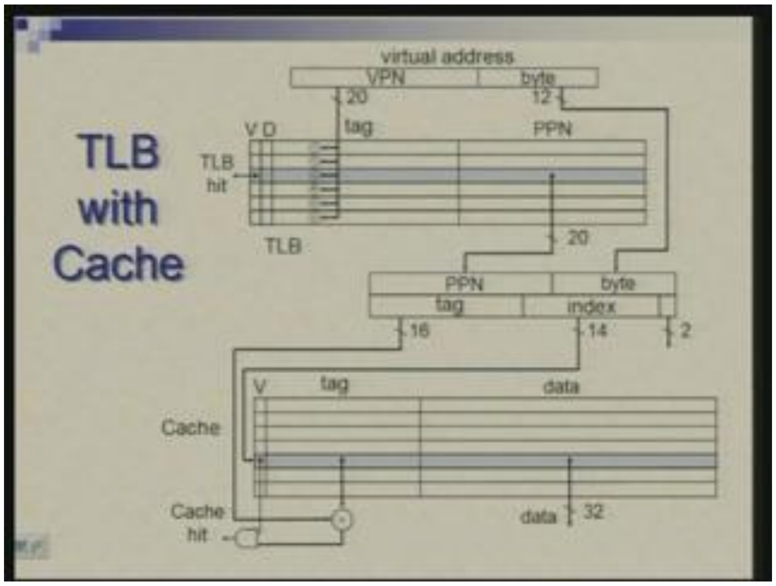


This buffer is another lookup table. TLB stands for Translation Lookaside Buffer and what it has is basically............ look upon this as a two column table. Given a Virtual Page Number we compare it with all entries in the first field. So here, what I am calling as tag here is basically nothing but a VPN Virtual Page Number. So suppose you have some address translation done recently this can be captured by remembering the virtual page number, physical page number pairs. So last few pairs if we store we have an associative memory where we can match it with all of them in parallel, if it is found then we do not need to go any further. So imagine this as an associative memory where you have three things: one is a valid bit, then there is a virtual page number field and a physical page number field. So with this given virtual page number of an address we try looking at all these simultaneously. If it matches somewhere we simply pick up physical page number and make access to physical memory. If we do not find it here then of course we go through the user process of translation through page table. So page table will always help out but TLB will have incomplete information. But hopefully the relevant part will be there.

Therefore, you could have a miss at this level that means you are now dividing the process in two steps: first TLB; you are hopeful and you expect that something will be found in the TLB if that works out there is nothing like it. The time spent here will be typically let us say one clock cycle of CPU. And if you succeed hoping that most of the time you would then the time spent is only one access to physical memory. So we have reduced the hit time from 3 or 2 times memory access time to 1. And only when you fail here then you make an access here. Again this may lead to a success or a failure so that that we have seen here that the failure here should be very very small.

[Conversation between Student and Professor.............22:58] yeah you are right. Suppose you miss out on the TLB and you go to page table you have a hit there then you will update your TLB. So it is like a cache; in principle it is exactly like a cache. Imagine that this is your main memory and this is your cache for that. Again same questions will come. When you are bringing in a new entry in this it is a fully associative cache so you have a question of which entry should be thrown off and replaced by a new entry so again you have same possibilities. LRU you try to use as far as possible.

(Refer Slide Time: 23:48)

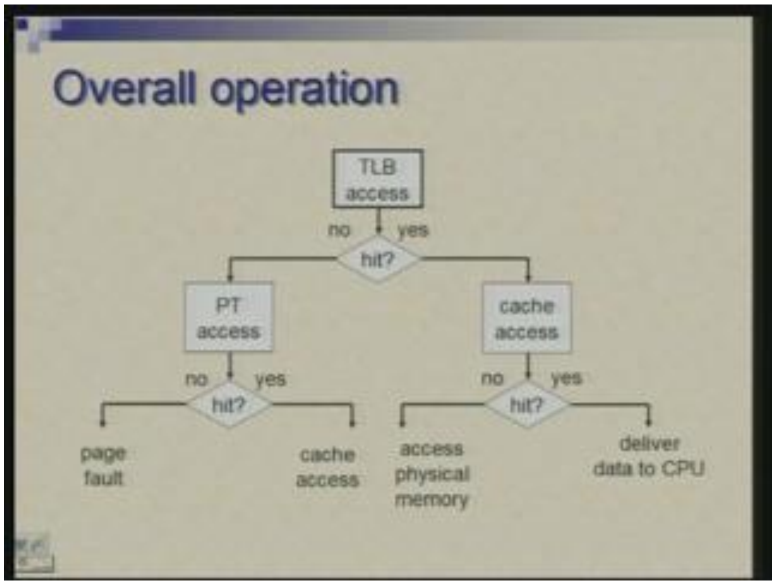


This shows the overall flow; how you will use TLB along with cache. You start with virtual address consisting of VPN the page number and the byte. For example, if address was a 32-bit address, the page number would be typically 20 bits if the page size is 12 bits and the remaining 12 bits specify the byte within a page. So first thing you are doing is looking at all the tags in this associative memory comparing them with VPN. So you also check the valid bit because there may be.... to begin with, at some of time there may be blank entries in this. This D bit is being used to indicate dirty bit. So dirty bit is
or clean bit comes from the write operation.

You remember, in cache also we talked about that you keep track of which cache blocks have been modified so that when those blocks are replaced in a write back scheme, when they are being evicted then the memory can be updated. So D would mean that there is something which has been modified here but it is not up to date with respect to page table. So before that entry is thrown page table would need to be updated. So this shows that comparison is made simultaneously and if somewhere it is a match it is a hit and if there is a hit we read out this physical page number stored here, concatenate that with the byte number to get a physical address.

Now this physical address would be reinterpreted from the point of view of cache access. Now we divide differently into fields; there is a tag field, index field and number of bytes within a block. So with index field you access the cache. What is shown here (Refer Slide Time: 26:05) is a direct mapped cache but other organizations are also possible. In this particular location we match the tag with the tag as part of the address and if there is a match also valid bit is set then there is a cache hit. So you can have a hit or a miss at this level then hit or miss at this level. So how do we put all these together.

(Refer Slide Time: 26:35)



Overall operation

The overall operation is where you are looking at three things. There is a TLB, there is a cache and there is a page table. In which order do you access them and what is the consequence of finding a success or failure at any of these levels.

As I mentioned that in the previous diagram first thing you are doing is you are doing a TLB access. TLB access is basically a shortcut to do in page table access. So if you are successful at TLB level then you do not need to go to page table. So I am showing two outcomes here. After TLB access you notice either a success or a failure. If it is a success then you can go to cache directly, if it is a failure then you have to resort to page table access. Let us see what happens when you had a success and you are looking at cache. So at cache level you may again find a hit or a miss. If you find a hit then your job is done you can get the information from cache itself and deliver that to CPU so there is no access made to physical memory at all, your search ends at cache.

If there is a miss here then you need to make an access to physical memory. So now what you have, you already have a physical address so you can go ahead and make an access to physical memory and the matter ends there. The other event is that you made a TLB access but there was a failure. In this case you need to go to page table where you might see a page fault or a hit. If there is a hit then you get the translation done. Basically it is now you are getting physical address and you need to make an access to cache now. We do not go straight to memory, we go to cache and again face both these possible outcomes. Just not to clutter up I have not connected here but effectively this branch is leading us to the same point here.

If there is a fault here then of course we have no choice but to service this page fault so context switch will occur and page will be brought from the physical memory so that will be a long long process here. Now you can see that fastest or the most desirable path is this that you have a hit at both levels; no physical memory access is required. If you come this way (Refer Slide Time: 29:30) one physical memory access, if you come this way there are...... because here you are further going here so if you suppose have a hit here sorry a miss here, a hit here and a hit here then you have in the process just done one physical memory access. But you might have a failure here and you may have to make another physical memory access. So, two physical memory accesses are made here.

On the other hand, if you go in here then you have made one physical memory access then you service page fault and make another memory access after that. If you look at these three tests together each throws up two possibilities: a hit or a miss so total there are eight possibilities theoretically. You have TLB, you have cache, you have page table; each will be having a miss or a hit. So all put together there are eight combinations of all these possibilities.

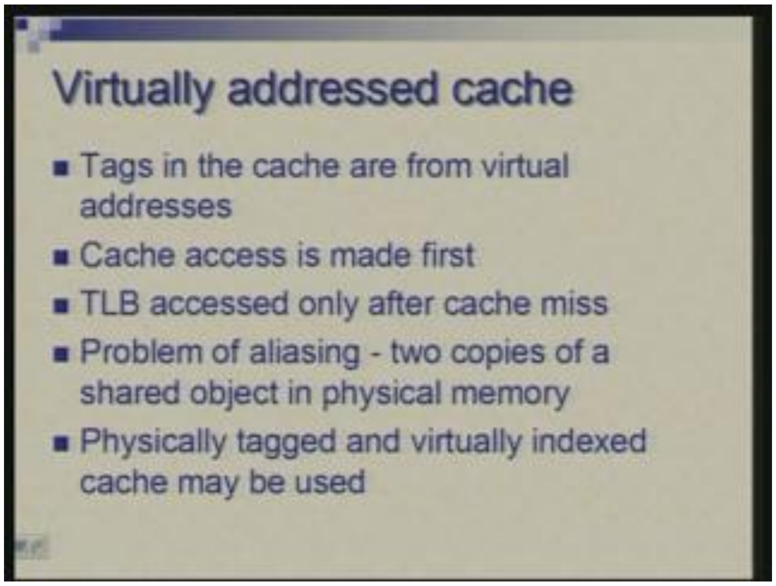
Question is have we covered all or we have not covered all?
Just take a moment and see; have we covered all eight cases?
[Conversation between student and Professor: (31:06)] ....which one? Okay something is not in TLB, no, it could be in the cache and we have covered that case. You may have a miss at TLB level and eventually you suppose you succeed at PT level then you access cache and we are taking care of both the possibilities.

If you simply count I mean it is like a tree, how many outcomes are there?
One is leading here, one is leading here, imagine repetition of this part here also so there are there are two leaves there and one leaf here. So we have 1 2 3 4 5 out of eight basically if you want to look at it as a tree leading to eight outcomes we have shown only five so what is missing here? What is missing here? Or there is nothing missing? [Conversation between student and Professor: (32:53)] Yes, yeah, if it is not in the page table it is not in the cache. So we are assuming that inclusive property here.

Suppose we assume, let us assume inclusive property that cache will not have anything which is not there in the physical memory. What we have in cache is a subset of what is there in the physical memory. So with that assumption some of those eight get ruled out. That means a miss at P T and a hit at cache that combination gets ruled out which actually amounts to two combinations: miss at PT, hit at cache and TLB either a miss or a hit. So those two are let us say non-existent so that leaves six, do we still have something missing? We should account for that what is missing. [Conversation between student and Professor: (33:58)] Yeah, see, when there is a hit in the TLB there cannot be a miss in PT. TLB is also eventually a subset of PT so TLB will not be holding an entry which is not there in PT. so, that combination is also ruled out.

So I think we can think little deeper here. Now, so far we assumed that before you hit the cache, before you go to the cache you would need to translate virtual address to physical address. That means as far as cache is concerned, cache is organized on the basis of physical addresses, cache is organized as part of the physical memory and not as a cache of the virtual memory, but in principle we can do that also. We can straightaway access the cache by virtual address and all that is required is that the tags which you store in the cache should come out of the virtual address and not physical address so that is quite possible and if that is the case the first thing you will access is cache and not the TLB.

(Refer Slide Time: 36:04)



Virtually addressed cache

- Tags in the cache are from virtual addresses
- Cache access is made first
- TLB accessed only after cache miss
- Problem of aliasing - two copies of a shared object in physical memory
- Physically tagged and virtually indexed cache may be used

Straightaway by looking at the virtual address you can make an access to the cache and only if there is a cache miss then you need to go to main memory and before you go to main memory you need a physical address from either TLB or if not there from page table so that order could be reversed. What do we gain by that? What we gain by that is that the best case path is shortened.

In this case the best possibility which we hope will occur most of the time is this where we are going through the sequence of two things: TLB access and cache access. It we on the other hand reverse this and we get cache hit for the first time then there is nothing else we need to do and that is the fastest possible approach. So there is a merit in trying to do that but this comes with some problems. Of course there are solutions, although some of them cost somewhat, but it is possible and it is done. So the problem is that of aliasing.

But first of all let me mention that to achieve this you need to distinguish entries corresponding to different processes because you have same virtual address space or let us say it is numbered the same way; you have process A, you have process B. for process A you say that virtual address goes from 0 to 4 GB and for process B also you say that virtual address goes from 0 to 4 GB. So now in cache can you have entries corresponding to multiple processes. If you have it in the straightforward manner there will be confusion.

How would you know that a process which is taking its virtual address and trying to access cache is not seeing the virtual address; not seeing a data of another process?
Either we distinguish by extending another field that in the cache we have data, we have tag, we also put process ID; we have another field where we distinguish the entries in the cache corresponding to various processes so that is an extra overhead we need to incur. Either we do that or we see that when context switch occurs, when you go from one process to other process you start with clearing up the cache, you empty the cache, start afresh and there is no confusion at all. The only thing you might lose by doing so is that if these two processes genuinely shared some data and part of that data was there in the cache, at that time you will lose it and we will have to load it again; so that is the drawback. But otherwise, if you are willing to incur little more cost, spend more bits in the cache and the corresponding hardware, you can distinguish the entries corresponding to different processes in the cache. Having done that, we still have the problem of aliasing.

Aliasing means that you have shared object which might be looked upon as two different virtual addresses. Let us say there is a shared data structure. In process A's virtual space, it is lying somewhere, process B's virtual space, it is lying somewhere else and depending upon how these processes are running they may get mapped to two different areas in the cache and it may happen that two copies exist because this virtual address gets translated to some address here, this gets mapped to some other address in the cache and if two copies are there of the same thing then there is inconsistency; one process may update one copy but the other may not come to know of it and it may read a wrong copy. So this is aliasing.

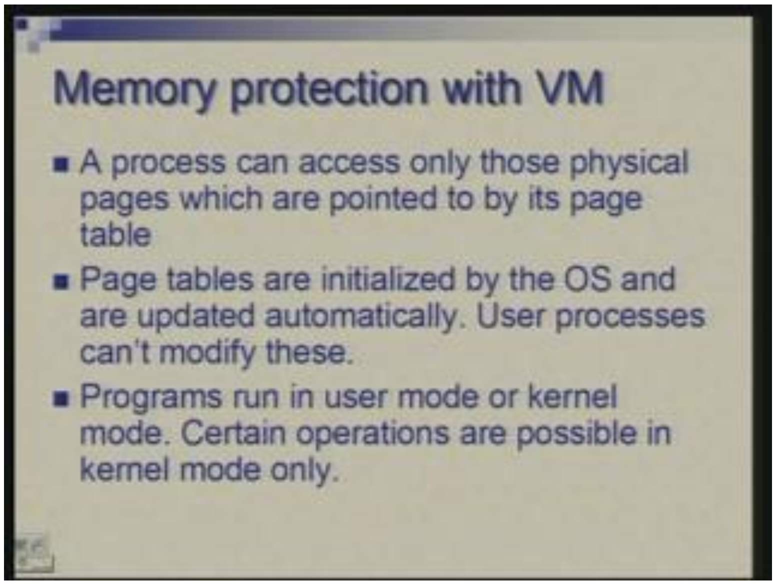
I will not go into the solution of this; there are more very involved solutions which can eliminate this kind of problem. There is another approach which is somewhat in between virtually addressed cache and physically addressed cache and it tries to get the best of both the worlds. We call it physically tagged and virtually indexed cache. That means......... I think let me explain this with reference to this diagram (Refer Slide Time: 41:04). If you look at this picture, cache requires two parts of the address. One is used as an index and the second is used as a tag. It is not difficult to imagine that what you need first is an index; you first need index to get to this point and then you read out and make a comparison.

Now which one of these two is dependent upon the address translation process?
In the given picture PPN is covering all the tag bits and a few bits of index. So we neither know tag nor do we know index fully unless we have found out PPN for the given VPN so here we cannot proceed with cache access unless we have done the translation. But suppose the picture was slightly different that is this..... let us say the tag was larger or PPN was smaller that is index bits were independent of PPN, what it means that the degree of associativity and the number of entries in the cache, the page size all these parameters are organized in such a manner that the index field does not overlap with PPN field. In that case index can be picked out from virtual address itself that is one part.

In the address translation this part (Refer Slide Time: 42:45) does not go through any modification it is being taken straightaway, it is this part VPN part which is getting translated into PPN. So the field sizes are such that index can be picked out of the non-changing part then you can start with the cache access immediately; whether there is a TLB access completed or not you can start with the cache access and by the time you are getting this here hopefully you have done the translation in TLB and you are ready for comparison. So here it is physically tagged and virtually indexed cache. That means the tag part is coming from the physical address but the index is coming from the virtual address or in fact it is a bit misnomer because the index part is same in virtual and physical address. We are picking up from virtual address but this part even if picked from the physical address would be unchanged so that is a very inexpensive arrangement and is very useful.

Finally let me talk little bit about what is the other role this organization plays. We primarily started with using virtual memory for expanding the address space in spite of limited physical memory. But another thing which it is doing is helping in putting the protection in place.

(Refer Slide Time: 44:18)



Memory protection with VM

- A process can access only those physical pages which are pointed to by its page table
- Page tables are initialized by the OS and are updated automatically. User processes can't modify these.
- Programs run in user mode or kernel mode. Certain operations are possible in kernel mode only.

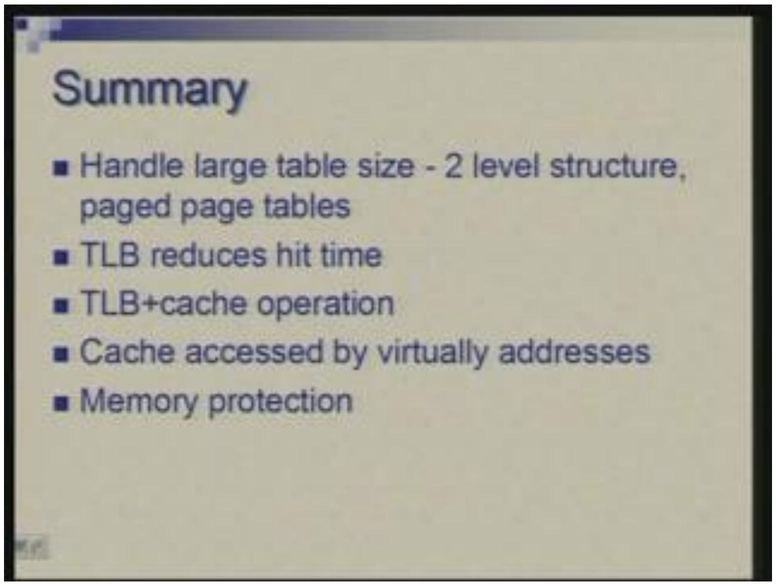
Protection means that we do not want one process to have unauthorized access to another processes data area or instruction area. Also, user processes should not be able to access restricted part in the system's area. So now since a process is able to access only those physical pages which are pointed to by its page table we have some protection inherently there. A program is allowed not to go to physical memory directly, it has to go through this transition process and therefore it can go to only those areas which are actually permitted by the page table and user program has no direct control over page table. Page table gets initialized by operating system and they get updated automatically;
as pages are brought in or swapped out, updation of the page table takes place automatically. User process does not have explicit access to that and therefore the

protection gets maintained. So this requires that a processor supports two modes of execution: the kernel mode or supervisor mode is one which has all accesses all permissions and a user mode where a few critical things are not permitted.

One of the things you can say is that initializing the page table. There will be a special instructions which are not accessible in user mode they are accessible only in kernel mode and how does this switching happens. So, for example, let us say user program is running....... first of all there will be some register some status register in the processor where a bit will be set to indicate whether currently it is a user mode or kernel mode and naturally user program will not be allowed to set a reset that way, it is only the OS which can do that. So when user program is running that bit is reset that means it is running in user mode. When it makes a system call that time, this bit will be set. That means when you are transiting from user program to system call you are going into system area, you are executing code which is part of the operating system and then you need to come back. So, at these two points the bit will flip and before you return to the user again you will reset this and come back.

So the user does not get control over that particular bit and user cannot change the mode that is the basis of this protection. So ultimately actually this protection or security is a very deep multilayered mechanism and you can see what is at the bottom of everything; bottom of everything is availability of these two modes and restriction on some basic operations. so from that, operating system derives its security, from that security the application programs or communication processes will derive their security at the network level and at that the application layers.

(Refer Slide Time: 49: 02)



Summary

- Handle large table size - 2 level structure, paged page tables
- TLB reduces hit time
- TLB+cache operation
- Cache accessed by virtually addresses
- Memory protection

So let me close at this point and to summarize we began today by looking at the techniques to handle the issue of large page table size. We looked at two level structure and paged page table structure. We noticed that the page table based approach asked us to have at least two memory accesses before you get to the physical data. But introducing these strategies makes it three memory accesses and to reduce this basically this three memory accesses means your hit time is very high; to improve on that the idea of TLB is introduced which is like a cache for the page table.

Then we saw how TLB and cache work together, then we saw everything put together TLB, cache, page table; what are the possible events and how one could sequence through these. We also looked at the possibility of accessing cache directly by virtual addresses; there are advantages and there are problems. Finally we saw a memory protection as an additional important role which is played by virtual memory mechanism.

That is all for today, thank you.