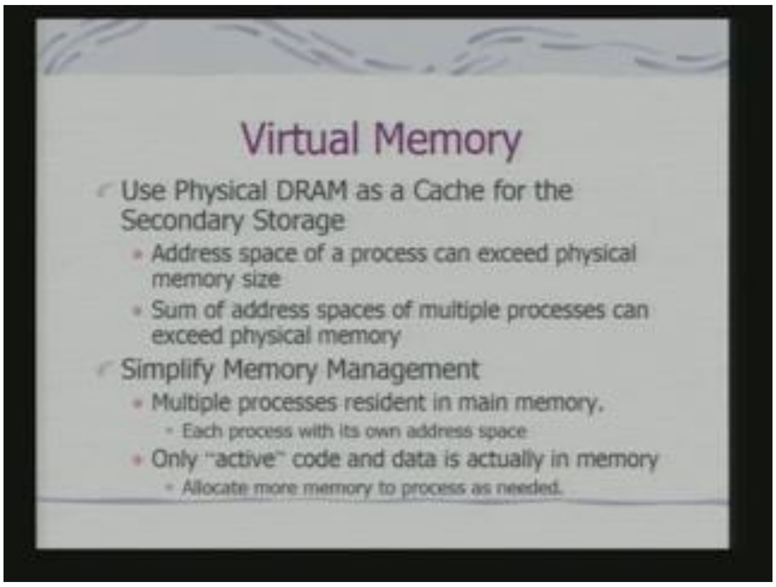
Embedded Systems

Dr. Santanu Chaudhury
Department of Electrical Engineering
Indian Institute of Technology, Delhi

Lecture - 13
Virtual memory and memory management unit

In the last class, we had discussed cache memory and cache memory arrangements. Today, we shall look at virtual memory and how we can use virtual memory in an embedded system, the basic motivation for having virtual memory to use physical DRAM like a cache for the secondary storage.

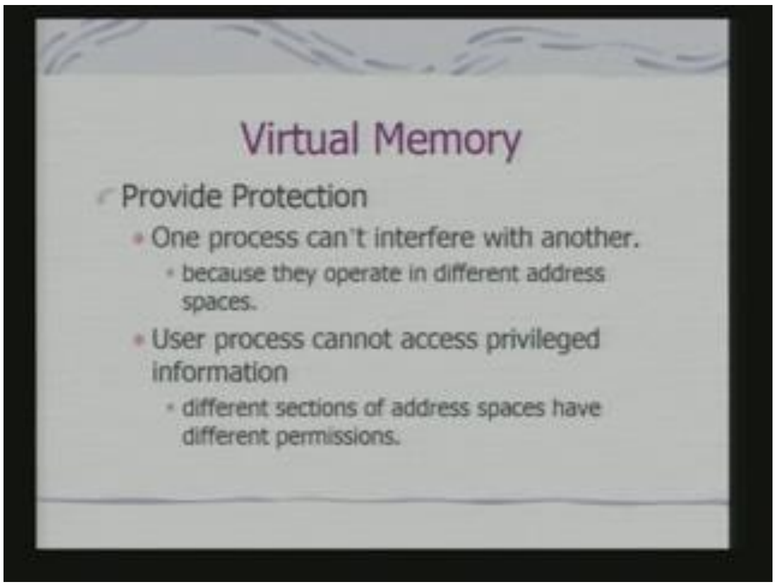
(Refer Slide Time: 01:35)


Virtual Memory
- Use Physical DRAM as a Cache for the Secondary Storage
  - Address space of a process can exceed physical memory size
  - Sum of address spaces of multiple processes can exceed physical memory
- Simplify Memory Management
  - Multiple processes resident in main memory.
    - Each process with its own address space
  - Only "active" code and data is actually in memory
    - Allocate more memory to process as needed.

The secondary storage in case of embedded system can be a disc or may be even flash memory being used as a disc. So, what is required is that address space of a process can exceed the physical memory size, when you are using virtual memory. So, I may have the DRAM or SDRAM not occupy completely the address space may be occupying part of the address space. And I may have secondary storage, which may be Flash or may be disc. And I can use the addresses in the secondary memory for the realization of a memory space bigger than that what is available in terms of SDRAM on the Embedded System. The virtual memory also simplifies the memory management system, because when I am having a multitasking OS and particularly in an Embedded System, when any to deal with multiple concurrent tasks as demanded by the physical environment. I need to have these tasks active in the embedded system itself. So, I need to manage the memory spaces of these different processors.

So, each processor will have its own address space now, a virtual memory system enables as to manage the processors process as a programs in execution in its own address space. Effectively only active core and data is actually meet resident in the main memory or DRAM. So, you can actually allocate more memory to the process as needed known active code that is code belonging to the process, which currently non active may be resident in the secondary storage. And when they become active can be brought into the main memory, which is your DRAM. So, effectively DRAM is functioning as a cache with respect to secondary storage, when you are using virtual memory. But another very important property is provided through virtual memory is that of protection.

(Refer Slide Time: 04:33)

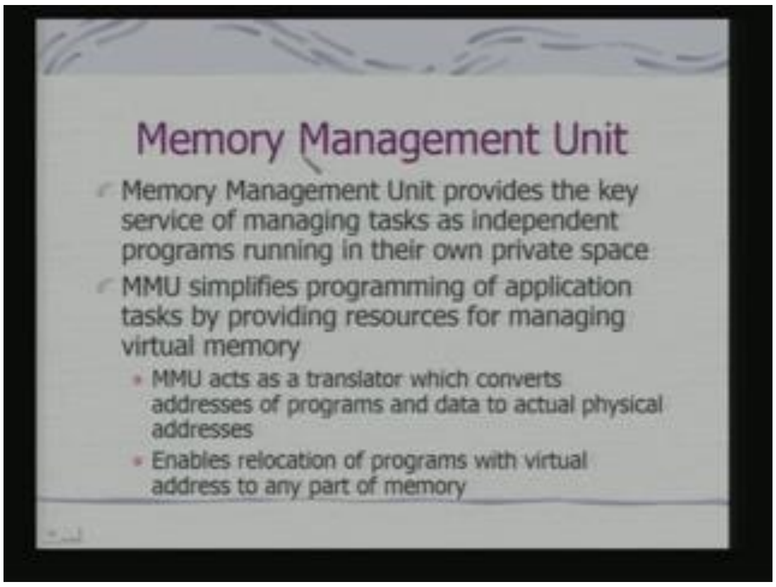


Virtual Memory

- Provide Protection
  - One process can't interfere with another.
    - because they operate in different address spaces.
  - User process cannot access privileged information
    - different sections of address spaces have different permissions.

One process should not interfere with another process, because they operate in different spaces. These we can make sure once we are the implemented virtual memory, because then we shall have a clear partitioning of the address space of different processors. Further user processor cannot access privileged information, because different sections of address spaces can have different permissions. This is particularly important to have and in a security feature on or embedded systems. Today, we have got Embedded Systems, we can which can download code and execute only. So, this download it code available from variety of sources should not corrupt the actual software, which is running in the embedded plants. In order to ensure that protection virtual memory is important, how is virtual memory realized, using primarily memory management Unit.

(Refer Slide Time: 05:47)

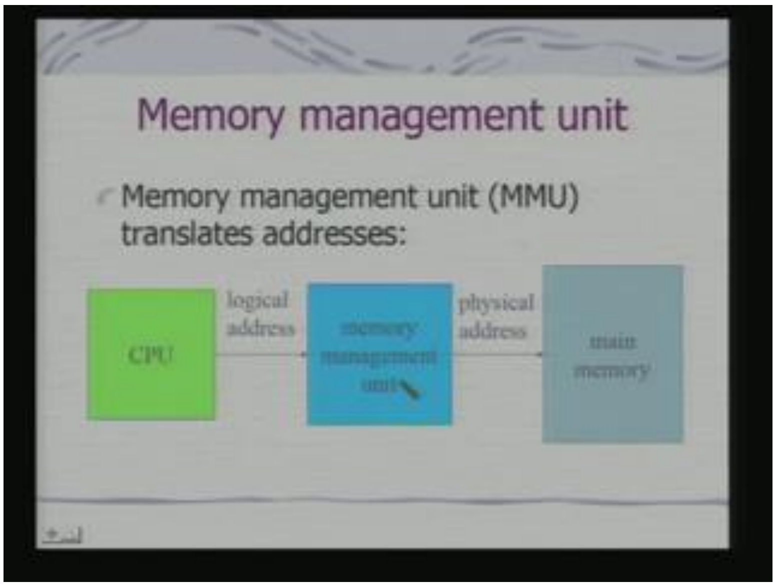


Memory Management Unit

- Memory Management Unit provides the key service of managing tasks as independent programs running in their own private space
- MMU simplifies programming of application tasks by providing resources for managing virtual memory
  - MMU acts as a translator which converts addresses of programs and data to actual physical addresses
  - Enables relocation of programs with virtual address to any part of memory

This is provided with many of this modern microcontroller, which are targeted for use in embedded applications. So, memory management Unit provides the key service of managing tasks as independent programs are running in their own private space. MMU simplifies programming of application tasks by providing resources for managing virtual memory. MMU translates the addresses the logical addresses the program generate to physical addresses to map them onto the actual DRAM. It also enables the relocation of programs with virtual address to any part of my primary memory the DRAM memory without actually of physically moving around the code.

(Refer Slide Time: 06:54)

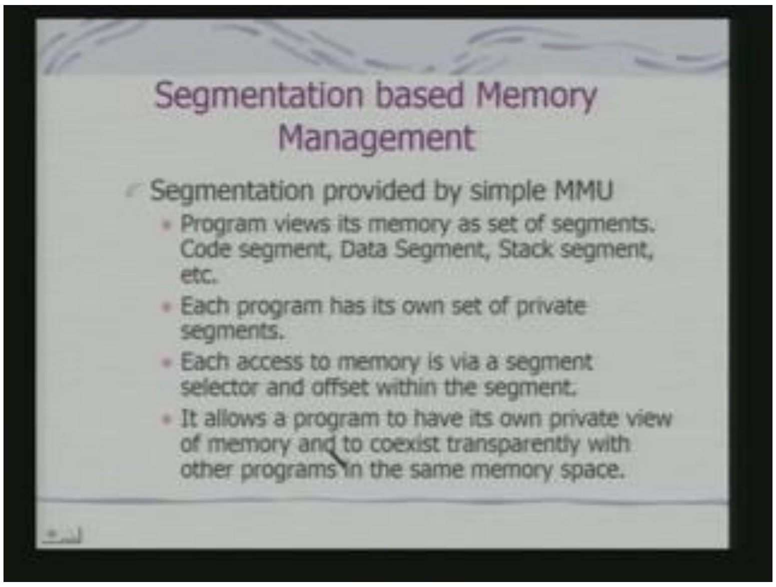


Memory management unit

- Memory management unit (MMU) translates addresses:

So, conceptually memory management unit lies between the CPU and main memory, when an instruction is executed the address is that CPU generates for accessing instructions or data a logical address. These logical addresses are received by the memory management unit and this memory management unit translates this logical address to physical address. And these physical addresses or addresses of the main memory locations from where the instructions or data are actually fetched.

(Refer Slide Time: 07:34)



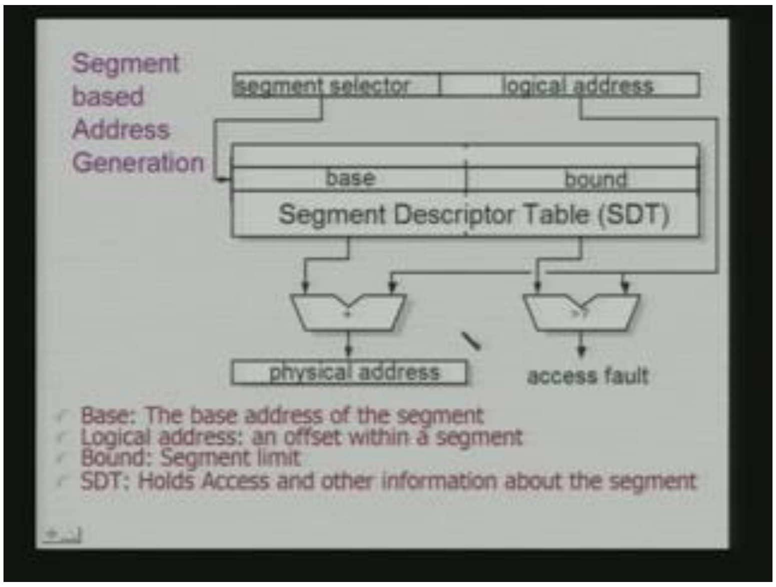
Segmentation based Memory Management

- Segmentation provided by simple MMU
  - Program views its memory as set of segments. Code segment, Data Segment, Stack segment, etc.
  - Each program has its own set of private segments.
  - Each access to memory is via a segment selector and offset within the segment.
  - It allows a program to have its own private view of memory and to coexist transparently with other programs in the same memory space.

Now, how is this virtual may memory realized? This simplest form is that of Segmentation based memory organization. Whether we are having really the mapping from secondary storage to the primary storage or not independent of that fact I can have my primary memory organized in terms of segments. The segmentation is primarily provided by the simple MMU. In fact, you are familiar with 8086, 8086 makes use of in it is program a program running on it is can make use of 4 logical segments Data segment, Code segment, Extra segment and Stack segment. The segmentation based memory organization that we are discussing now, is a simple extension of that basic concept. So, program views it is memory as set of segments, Code segment, Data segment and stack segment.

Each program has it is own set of private segments so, these clearly define different addressing space for different processors. Each access to memory is via what you call a segment selector and on offset within the segment. Now, all these things permits a program to have what we have already telling to have it is own private view of memory and to exists transparently with other programs in the same memory space. Obviously, what is implied by this statement that a program, which is an execution, will not access a data segment, which belongs to some other process or program in execution. Another whole infrastructure, which is provided by the MMU will ensure that such access is a really denied and if any such access attempt is made an error is flat.

(Refer Slide Time: 10:10)

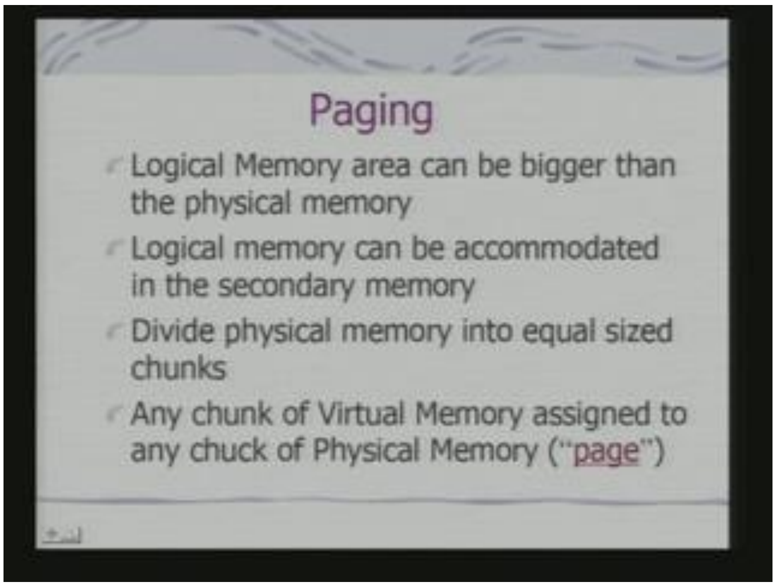


- Base: The base address of the segment
- Logical address: an offset within a segment
- Bound: Segment limit
- SDT: Holds Access and other information about the segment

So, the address generation is through this process so, in fact, the address actually can be divided into 2 parts. The address, which the CPU generates can be logically looked at consisting of 2 parts. The first part is segment selector, which actually indicates the segments the other part is a logical address, which is really an offset in the segment. Now, the segment selector is used to access the base address of the segment in what we call Segment Descriptor Table. So, these give me the starting point of the segment. So, using the segment selector the starting point is opting from this table that starting point is added with the offset to get the physical address. This is the address using which allocation is accessed in the primary memory like DRAM or SDRAM.

So, base is the base address of the segment the logical address is offset within the segment. I am the next thing, which is to be noted is a bound, because each segment can be associated with a fixed size. So, this bound information can also be kept in the segment descriptor table. So, what is checked is that once we do this sum that is offset with the base that should not exceed the bound of the segment. If it exceeds the bound of the segment then there will be an access fault so, this is an exception. We are already discussed in drops an exception, this is a internally generated exception, which is to be handle by the operating system. So, this mechanism ensures that each segment is clearly separated out and you cannot access beyond the segment boundary. The next form or the next way to implement virtual memory is by use of what is call paging.

(Refer Slide Time: 12:44)



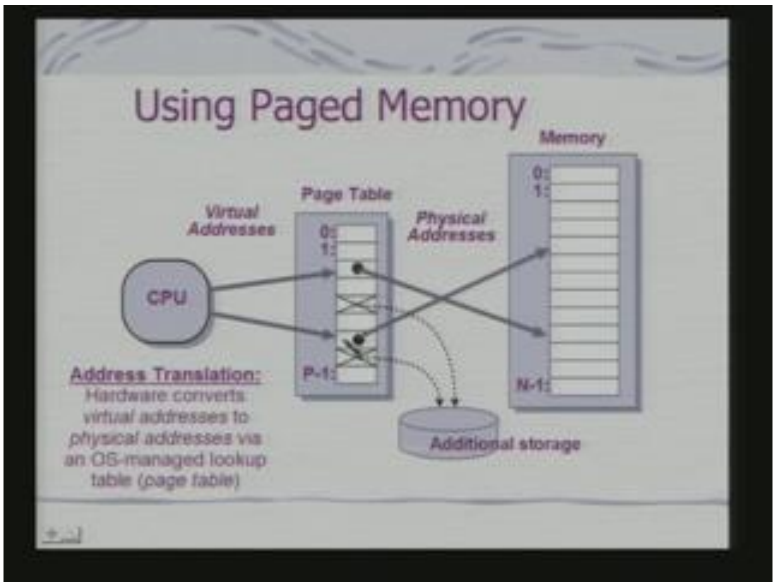
Paging

- Logical Memory area can be bigger than the physical memory
- Logical memory can be accommodated in the secondary memory
- Divide physical memory into equal sized chunks
- Any chunk of Virtual Memory assigned to any chuck of Physical Memory ("page")

Now, the core concept is logical memory area can be bigger than the physical memory. This is the one of the motivation with which we started on the concept of virtual memory. Logical memory can be accommodated in the secondary storage and what can be done is divide physical memory into equal sized chance and this chance really pages. Any chunk of virtual memory is assigned to any chunk of physical memory that is the basic principle of paging for implementation of virtual memory. So, what can be inferred from this let us say I have a pretty large secondary storage. The secondary storage is also the logically being divided in terms of chunks or pages.

And I have space for few pages in the primary memory and depending on which process is a currently active page from secondary memory will be loaded into pages of the primary memory. In fact, conceptually this is very similar to that of cache blocks; you mapped memory locations to cache blocks. This exactly what is happening? But here now talking about the secondary storage and the SDRAM or DRAM introduction. The difference between the segmentation and paging is what segmentation can be arbitrary length here when we are talking about pages we are talking about chunks of fixed length.

(Refer Slide Time: 14:48)



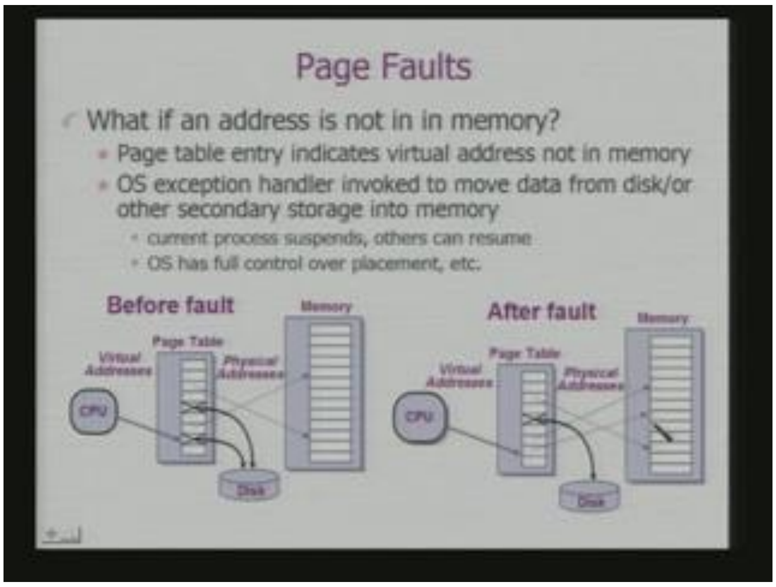
Using Paged Memory

Address Translation: Hardware converts virtual addresses to physical addresses via an OS-managed lookup table (page table)

So, pictorially the process is like this CPU generates the logical or virtual address. We used for logical or virtual address to map into what we call a page table. This is these page table is representation of the storage area required by a process. So, a process occupies memory area. So, that memory is divided into a number of pages here we are talking about P pages for a process P. So, a process is divided into P pages and information about all these pages at we maintained in a page table. So, there are P entries in these page tables. Now, all these pages belonging to a process and not necessarily resident in the physical memory, some of the pages, a resident in physical memory. So, here we are showing one example these page is resident in physical memory at these location, because memory is also logically divided into pages and another important thing to be noted. This pages can be accommodated anywhere in the memory.

So, therefore, pages need not be stored at contiguous locations in memory. Some of the pages, which are note in the physical memory can be resident in the additional storage of the secondary memory with additional storage can be disc. In fact, for a general purpose computing system, it is invariably a disc for an embedded system it can be a disc or it can be a flash memory area. In fact, the hardware, which manages the page table, converts the virtual addresses to physical addresses. And these page table the software wise this page table what will be the mapping or the influence on the page table is managed by the operating system. So, operating system actually knows where the pages are getting mapped to physical memory an accordingly, it will load the addresses in this page table. So, what happens, when CPU generates an address of a page, which is not in page table then we have what you call page faults.

(Refer Slide Time: 17:36)



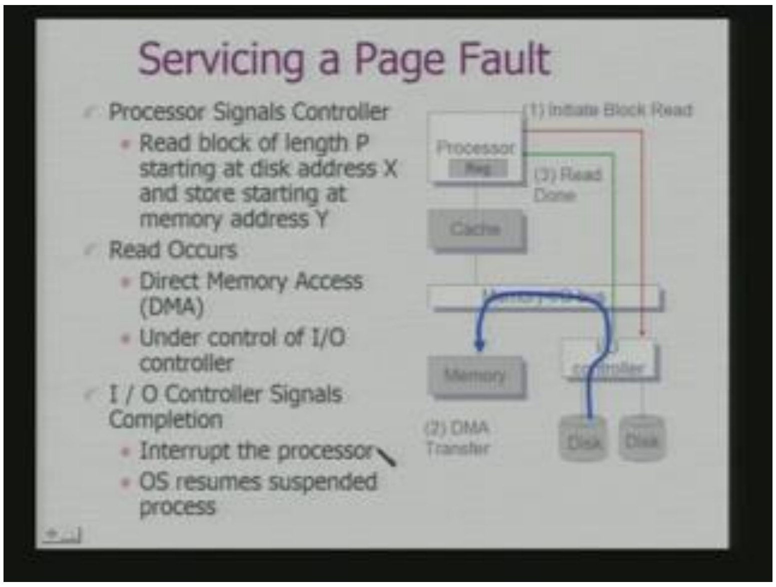
Page Faults

- What if an address is not in in memory?
  - Page table entry indicates virtual address not in memory
  - OS exception handler invoked to move data from disk/or other secondary storage into memory
    - current process suspends, others can resume
    - OS has full control over placement, etc.

Before fault
Memory
Page Table
Virtual Addresses
Physical Addresses
CPU
Disk
After fault
Memory
Page Table
Virtual Addresses
Physical Addresses
CPU
Disk

So, page fault occurs when CPU generates an address, which is not in memory. So, page table entry indicates that virtual address is currently not in memory so, this is a memory exception. So, this exception handler is invoked to move data from disk or other secondary storage into memory. In fact, if you remember that ARM has also got a corresponding exception and the corresponding mode to handle these kind of exception. So, current process, which generated is request cannot execute. So, it has to be suspended, but since we have talking about a multitasking system. There will be other active process in the system. So, that can now, be resume and executed, in fact, OS manages this. So, what we have this OS also has a full control over the placement that is require the pages are to be obtained and where to active map. So, what happens is that when the page for exception occurs. So, the data from the disc is fetched that page what you fetch at a given pointing time a page again the same principle that we have discussed in the context of cache the temporal and spatial locality is exploited. So, I do not fetch a single instruction, but I fetch a page and that is put in the memory.

(Refer Slide Time: 19:29)



Servicing a Page Fault

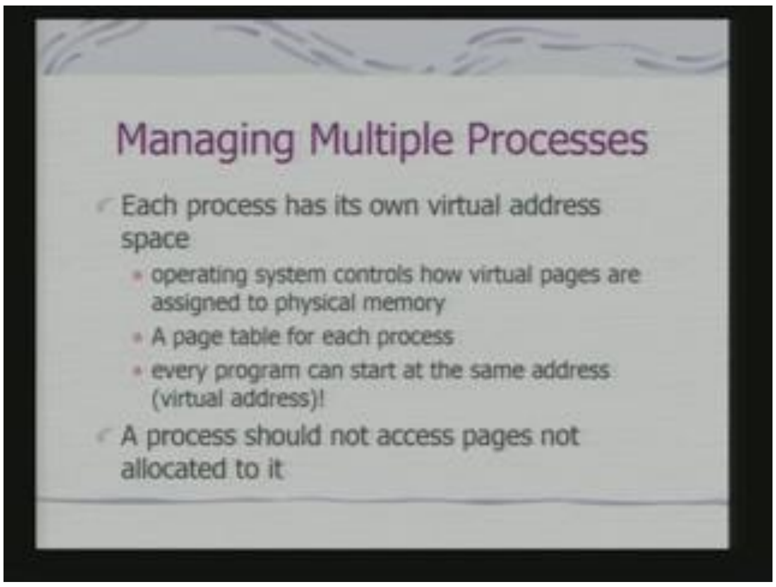
- Processor Signals Controller
  - Read block of length P starting at disk address X and store starting at memory address Y
- Read Occurs
  - Direct Memory Access (DMA)
  - Under control of I/O controller
- I / O Controller Signals Completion
  - Interrupt the processor
  - OS resumes suspended process

So, the exact servicing sequence, if you want to look at it in terms of this hardware operation. So, what really happens? So, processor signals the controller, which provides a read block of length P starting at disk address X and store starting at memory address Y. So, this is the action, which is actually taken care by the exception handler, because this is what is stole to the controller. That is your IO controller, which is managing the disc here ((refer time: 20:08)) with reference to the disc. It can be also with reference to a

Flash controller of Flash interface. Then the IO controller takes the data from the disc and then once the data has been obtained at DMA direct memory access is requested.

This direct memory access is executed under the control of IO controller. So, from the IO controller buffer actually, which in a way logically from the disc. The data goes into the appropriate location in a memory. In fact, where it will going to the memory is provided by the exception handler. Once the DMA is over the IO controller interrupts the processor in generates an interrupt telling the DMA process is complete. So, once the DMA process is complete now, the page is memory is a memory. So, the process, which was suspended, because it generated a page fault exception can now be resume. Resuming means what? It can be put into an active for ready queue.

(Refer Slide Time: 21:34)

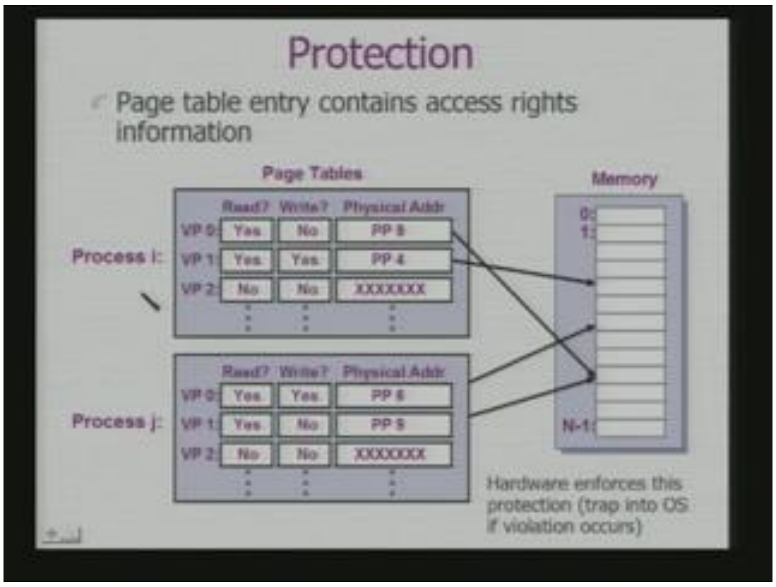


Managing Multiple Processes

- Each process has its own virtual address space
  - operating system controls how virtual pages are assigned to physical memory
  - A page table for each process
  - every program can start at the same address (virtual address)!
- A process should not access pages not allocated to it

So, when there are multiple processes what happens with this page table? I have already told you that each process has its own virtual address space. An operating system controls how virtual pages are assigned to physical memory. So, the simplest way to implement this is that each process will have its own page table. In fact, under that condition every program can actually start at the same address same virtual address. But where, that the same virtual address is getting map one to the physical memory, that information is contained in the page table. And the process should not access pages not allocated to it. So, all the protection information now will be associated with the pages. So, for each process I shall have its own page table and there would be permissions associated with the pages. And the permission should ensure protection of the code in the Embedded System.

(Refer Slide Time: 22:50)



Protection

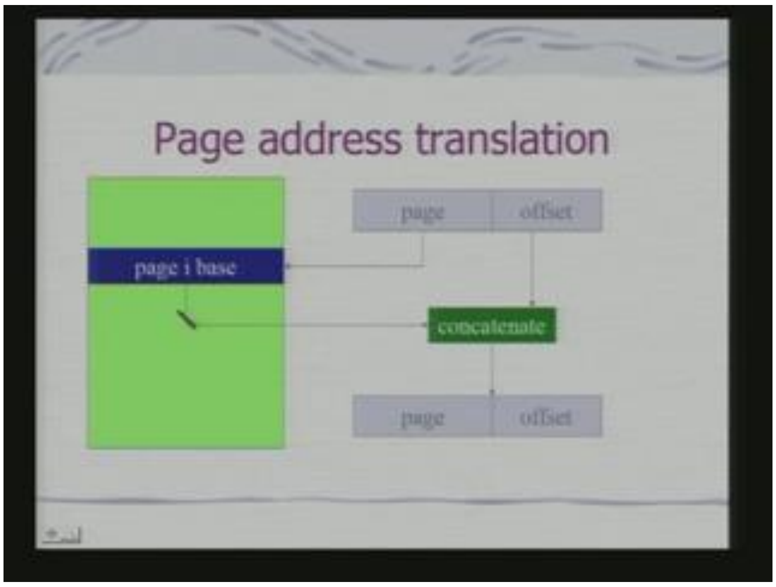
- Page table entry contains access rights information

So, the pictorially, it looks like this that process this is process I and this is process j. So, it each of it this processes have their own virtual page tables. Now what we have shown here is a kind of permission information. The can be much more a ((refer time: 23:15)), but here it is showing that read write whether process I has got a read permission for this page, because this is virtual page 0. This is getting mapped onto a page 9 in the physical memory and it tells these process, I whether it has got the permission to write onto this page. It does not have a permission to write onto the page. But it can read from that page VP 1 for these page, it has got the permission to both Read as well as Write, Similar thing would be true for process j. Now, even understand this enables also shading of pages across process. So, if I have a library routine or a common routine or a proceed here, which is to be used by more than one process.

I can keep it resident in a physical address space and for that physical address page, I shall give read permission to all the processes, that need that page, but I shall not provide write permission any one of that. In fact, typically a code page even it is a private to a particular process that page containing your code typically does not have write permission. Only the data area the pages corresponding to the data area do have write permissions, the other point you should keep in mind is that; obviously, when an executing a process. A process has got logically a code area; it will have an area for store the globes. It will have the stack to accommodate local variables as well as the return addresses for subroutine calls.

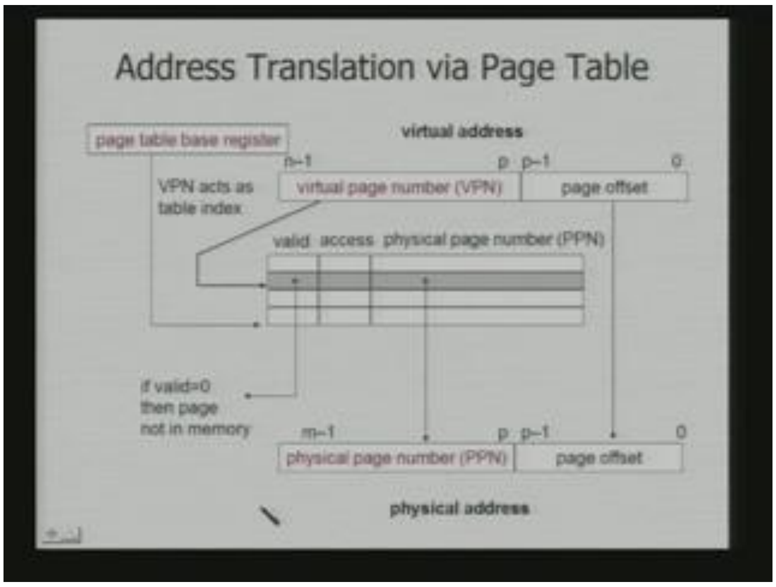
It will also have keep to accommodate dynamically allocated storages that is the storages that you allocate for implementation of linked list. Now, all these are different logical memory areas that a required buys a process to execute. And all these memory areas are again divided into pages the chance. So, when we are looking at pages like segmentation system, we do not talk about logical code segment data segment extra. We talk about the pages, which belong to a processor this pages can have logical relevance in terms of that is they can have code, they can have data. That is they can actually implement the stack their part of the stack of the private of the process. They can be part of the keep for the process. Now, how is the page address really translated?

(Refer Slide Time: 26:33)



The basic principle is same as that of the segmentation. So, you have a page now, we look at the address, which is being generated the logical address, which is being generated by the CPU as combination of page and offset. So, this page number is mapped onto a page table that provides the base of the ith base if I have these I as a page indexes it would provide the base of the ith page and that base is concatenated with the offset to access the exact location in the page. So, I can have a logical address and the corresponding page can be located anywhere in the memory. What I need? I need to have actually the page table appropriately operated.

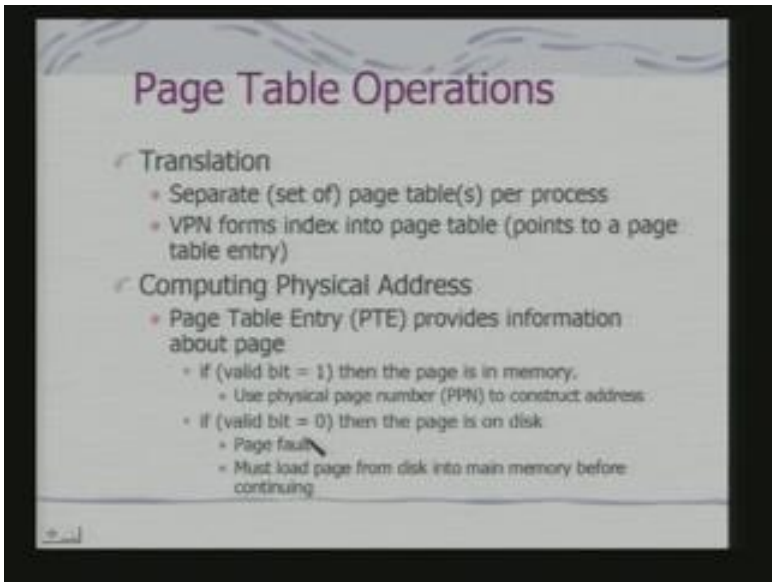
(Refer Slide Time: 27:35)


Address Translation via Page Table

So, in more detail form, we have these kind of a structure. So, I this is my address the virtual address or the logical address have a virtual page number. This is part of the logical address, this is the page offset I use this virtual page number to access a location in a page table. Now, there is a page table base register now, we can understand that when I switch from one process to another process my page table would change, because each process has got its own page table. So, changing page table would essentially imply changing page table base register value. So, the virtual page number is like at offset added to the content of the page table base register to access the location in the page table, which has got the physical page number. But in the corresponding entry in the page table please note that along with physical page number. We have other information's as well we have a flag, call valid flag, which says that if valid is 0 then page is not in memory.

Therefore, conceptually this is the same as that of your valid bit in your cache. Then there are few bits, which are used for defining the access permissions. In fact, what we have not also shown is the dirty bit, because if I have write permission for that page, then the process can actually update contained of that page in the memory. So, when it has to be stored back to the secondary storage, if that page is evicted from the memory I need to write it back. So, I need to know whether content of that page has been modified or not. So, dirty bit would flag that condition. So, here you have the physical page number and the page offset added to it will give you the actual location in the page corresponding to the logical or virtual address generated by a CPU. In maturity of the modern microcontrollers, which are used the MMU support paged memory and page table associated with individual processes. So, what are the days for operations of a page table?

(Refer Slide Time: 30:29)

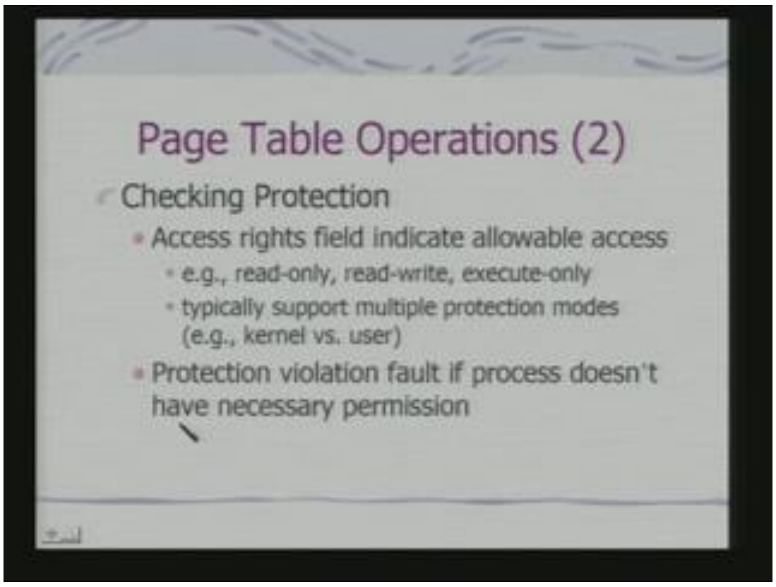


Page Table Operations

- Translation
  - Separate (set of) page table(s) per process
  - VPN forms index into page table (points to a page table entry)
- Computing Physical Address
  - Page Table Entry (PTE) provides information about page
    - If (valid bit = 1) then the page is in memory.
      - Use physical page number (PPN) to construct address
    - If (valid bit = 0) then the page is on disk
      - Page fault
      - Must load page from disk into main memory before continuing

The basic operation of the page table is translation, because you have separate set of page tables' purposes. And we have already seen that virtual page number forms index into page table; that means, it coincide to a particular page table entry. And then you compute the physical address so, page table entry provides the following information. If valid bit one the page is a memory so, use physical page number to construct address. If valid bit is 0 then the page is on disk, hence you raise an exception and in response to the exception the page would be loaded. And how the page is loaded how this exception is serviced we have already discussed.

(Refer Slide Time: 31:33)



Page Table Operations (2)

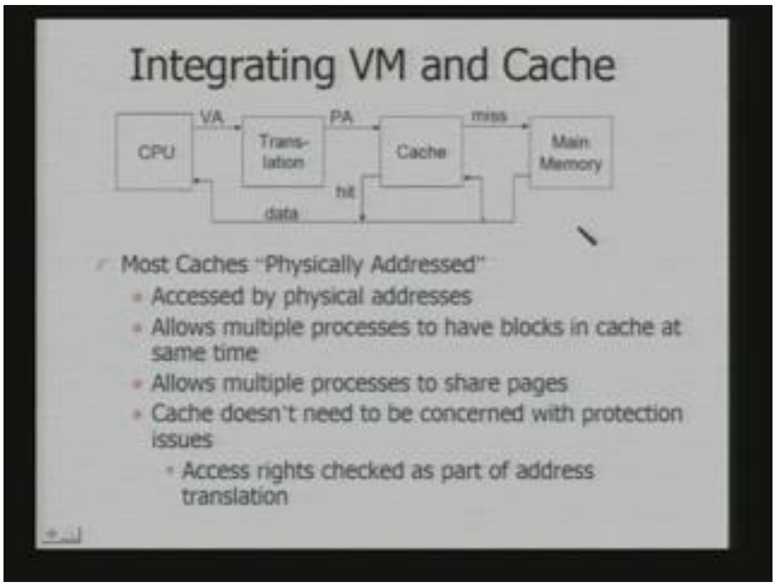
- Checking Protection
  - Access rights field indicate allowable access
    - e.g., read-only, read-write, execute-only
    - typically support multiple protection modes (e.g., kernel vs. user)
  - Protection violation fault if process doesn't have necessary permission

The other important aspect of page table operation is that of checking protections, the Access right field indicate allowable access read-only, read-write and execute-only. So, I can have pages marked no read, no write, but execute. So, I can simply execute the content of the page, but I cannot read value from that page into register or some other memory location. And typically, it supports in hardware multiple protection modes. There are different modes in which a processor operates, we have seen with reference to an example of ARM I have basically a supervisor in mode of operation and a user mode of operation when a process is operating in user mode. It will have some restricted permissions it cannot execute instructions located in a page, which is protected via supervisor you are the kernel domain.

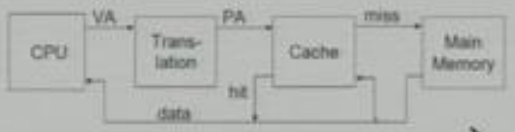
So, this separation is done again to ensure protection of the code, you would like the basic operating system or the basic operational code of an embedded appliance to be protected against malicious attack. Because today, say on your cell phone, you can actually receive the code on the fly and you can execute them. But you have to protect your cell phone against a kind of a virus attack or any other kind of malicious attack through this on the fly codes set your downloading an executing. So, these kind of production mechanism as implemented through this virtual memory becomes critically important. And protection violation fault occurs, if process does not have necessary permission and it is try to access a location, which is not really allowed.

Now, with regard to this a primary observation is critical that in maturity of Embedded Systems. You may not really have demand paging, but what you will have definitely is multitasking. And for ensuring protection and organization of the code of multiple task into separate logical address space. The virtual memory systems implementation and use of MMU and that is why it is very critical for embedded system designer to know about virtual memory. Although in many of the cases, they will not to be using hard disks as a logical storage space like in a general purpose computing.

(Refer Slide Time: 34:36)



Integrating VM and Cache



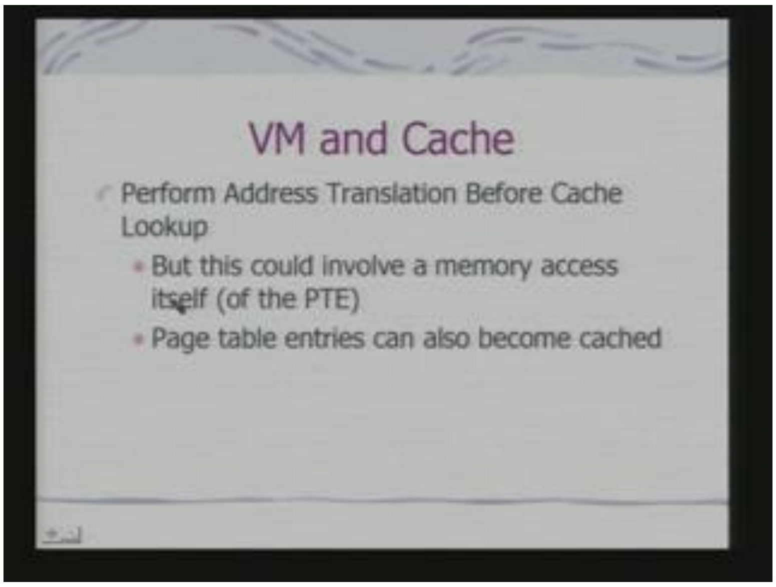
- Most Caches "Physically Addressed"
  - Accessed by physical addresses
  - Allows multiple processes to have blocks in cache at same time
  - Allows multiple processes to share pages
  - Cache doesn't need to be concerned with protection issues
    - Access rights checked as part of address translation

Next thing we shall look at how to integrate virtual memory and cache, because what CPU is generating is generating the logical address. Now, these logical address is being translated by the page table, which is under hardware control of MMU an under software control of operating system. This translation unit is actually generating the physical address. Now, finding out whether there is a cache hit or miss these physical address is to be used. And this physical address is what is being generated by the translation unit is the input to the cache controller. Here, we are not shown the cache controller we are just shown the cache. Now, if there is a miss then they load from the main memory to the cache otherwise you have a hit and you get the data from the cache.

Now, this translation process itself you can understand pretty well a page, which is not in main memory can not be in cache. So, whenever there is a fault page fault. So, page fault has to be serviced and then only when the process gets resumed. It may access a location so, what will happen in that case there will be a case of a hold miss for the cache. So, that data will not be in cache so, that has to be loaded from main memory into the cache. And you may have the data steamy the data flowing into the CPU and referring to data, but it will true as well for instructions. So, what we say is that caches or physically addressed maturity of the cases are really physically addressed. So, you access by physical addresses.

It allows multiple processes to have locks in cache at the same time. It also allows multiple processes to share pages and cache does not need to be concerned with protection issues. Because is protection issues a taken care of at the level of translation processes itself. So, here the organization is when you are using the physical addresses and physical addresses for accessing the cache. But what you can realize is that where will be the page tables really stored? Because when I using the translation you have to access the page tables, otherwise you cannot do the translation from logical address to the physical address. Typically, page table would be somewhere in the main memory MMU cannot really store the page table. So, an access to each location therefore, implies what action access to main memory. Then you are actually defeating in that case the basic purpose of using cache. So, there has to be some architectural innovation to overcome this problem.

(Refer Slide Time: 38:00)

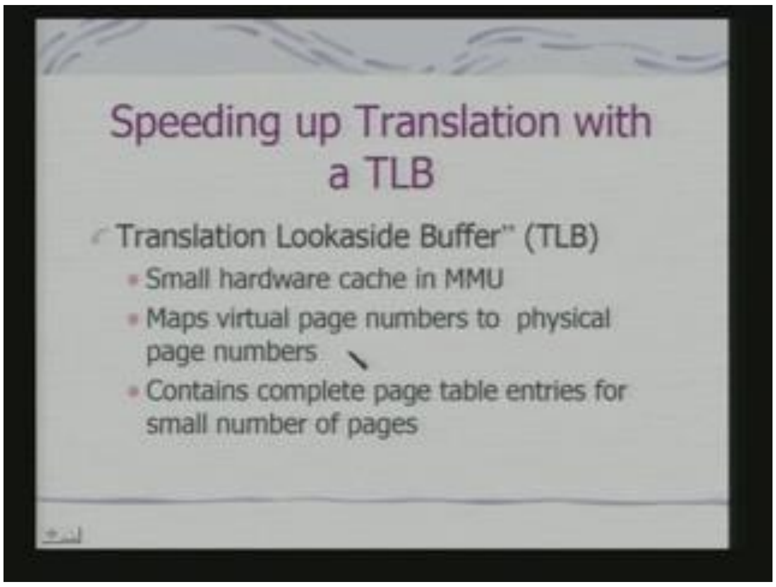


VM and Cache

- Perform Address Translation Before Cache Lookup
  - But this could involve a memory access itself (of the PTE)
  - Page table entries can also become cached

The how is this problem over overcomes so, you perform address translation before cache lookup. So, what you need to do therefore, make the page table entries cached. What we are telling that since is lookup involved memory access, it is not long or efficient so, make page table entries cached.

(Refer Slide Time: 38:20)

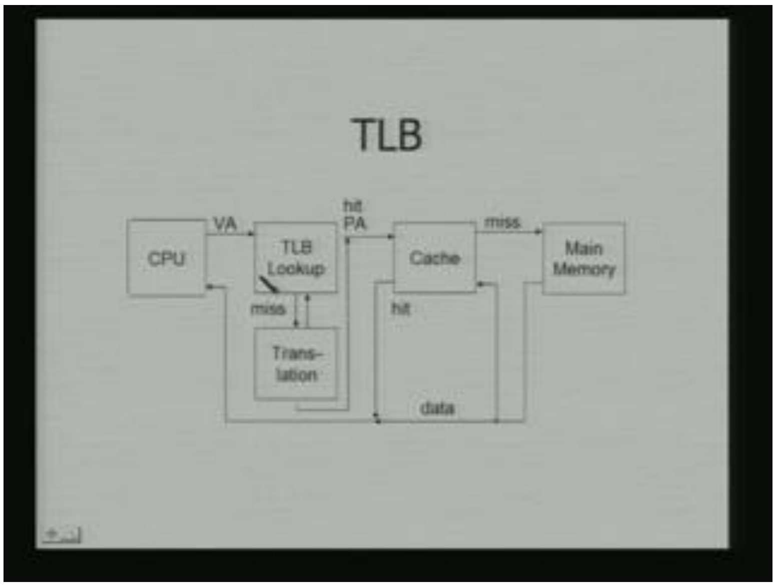


Speeding up Translation with a TLB

- Translation Lookaside Buffer" (TLB)
  - Small hardware cache in MMU
  - Maps virtual page numbers to physical page numbers
  - Contains complete page table entries for small number of pages

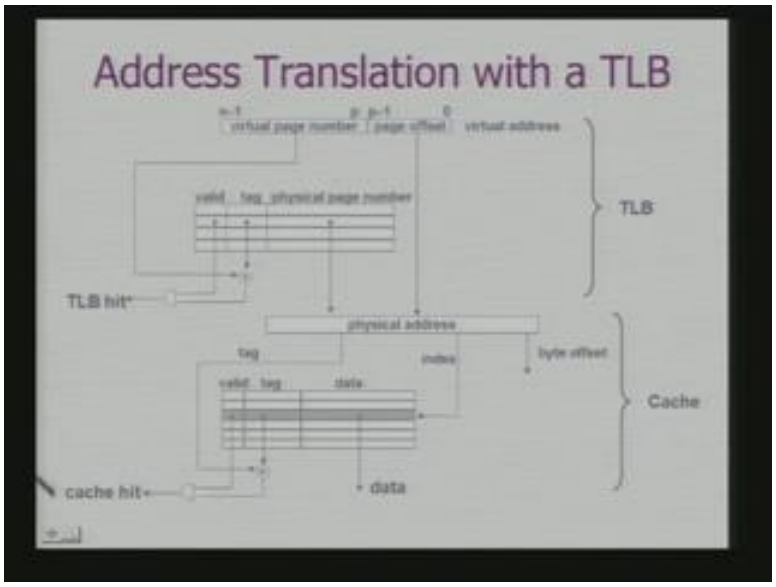
So, you have the concept of what is called Translation Look aside Buffer or TLB. So, TLB is typically as small hardware cache in MMU. TLB can also be in L1 cache as well, but in many cases it is a hardware cache within MMU that is your memory management Unit. So, which maps virtual page numbers to physical page numbers and contains complete page table entries for small number of pages? So, there has to be a caching policy for these pages as well.

(Refer Slide Time: 39:10)



So, now, what we have got CPU generates the virtual address. So, if have got a TLB translation look aside buffer. Now, if there is a miss for these they can be miss, because I may not have the complete page table loaded. So, if there is a miss then I have to get that data either true from the cache or I have to get that information from the main memory. So, I may have part of the page table stored in the cache and remaining part may be stored in these TLB. I then I do the translation so, once I look at the TLB I know the information. So, TLB is what TLB is nothing but a page table entry put into cache. So, using that information you have to go through the process of physical address generation cache access hit and so on and so far. Now, these miss you should understand has got two kind of significance. One is the page table contain itself can be missing from the cache else it can be actually entry in the page table missing. Because that block has not been loaded as of it ok. So, why this two kind of miss may come in? We shall understand gradually.

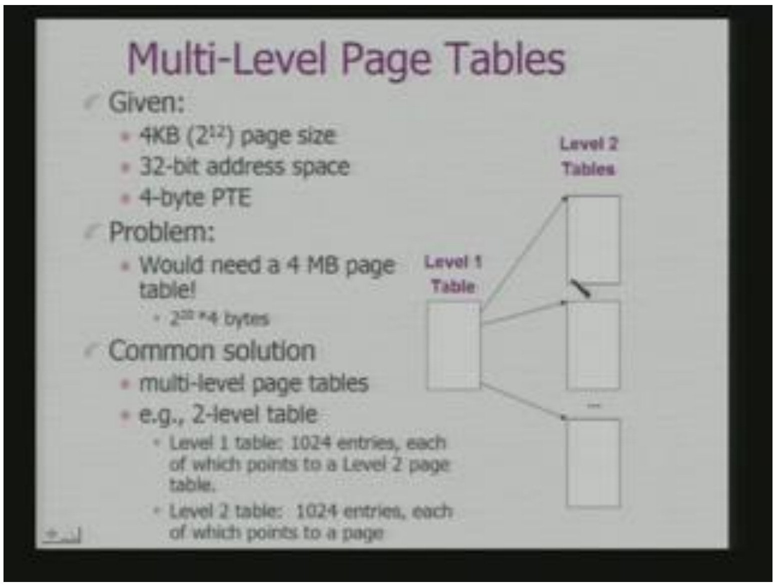
(Refer Slide Time: 40:43)



First let us look at the basic translation process so now, your virtual page number is refer to two TLB so, TLB is like a cache. So, it will use the tag in fact, tag part from the virtual page number to check whether the tag matches or not. If the tag matches then there is a TLB hit and you get the physical page number. So, I may not have all these pages information about all virtual pages stored in the TLB. So, there will be a miss and even if I can store them you can understand the other logical situation is what the physical page may not be allocated. So, that is also up that case of a page fault.

Now, the system can follow a policy that it store only in the TLB information about those pages, which are currently memory resident. So, I am if I am accessing a page, which is not in the memory resident there will be a TLB miss. Then, I shall go back to the physical memory to get the remaining part of the page table to find out whether the page is really memory resident or not and generate a page fault. So, this two aspects that we are referring to that in this case I have got a TLB hit if it is a TLB hit. I get the physical page number I get the physical address use that physical address to check whether that location is in the cache. So, if it is a cache hit then you execute otherwise you have to load the data onto the cache and then only you can execute.

(Refer Slide Time: 42:34)



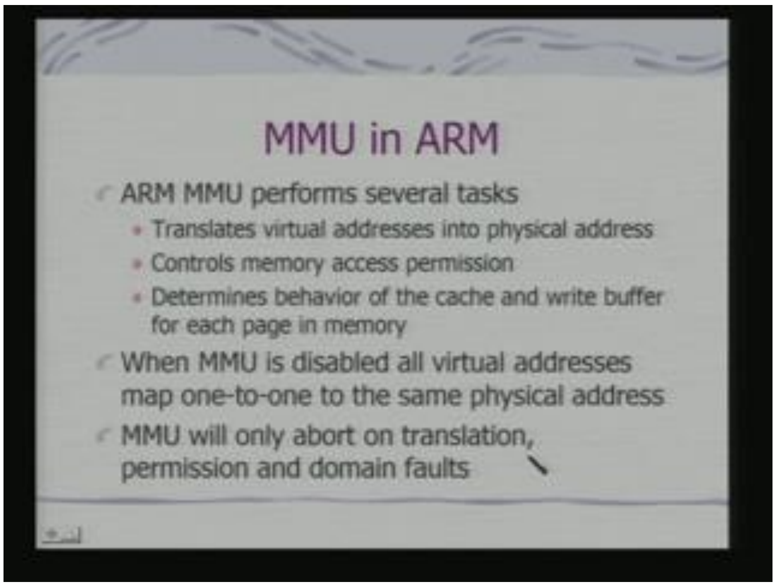
Multi-Level Page Tables

- Given:
  - 4KB (2^{12}) page size
  - 32-bit address space
  - 4-byte PTE
- Problem:
  - Would need a 4 MB page table!
    - 2^{20} *4 bytes
- Common solution
  - multi-level page tables
  - e.g., 2-level table
    - Level 1 table: 1024 entries, each of which points to a Level 2 page table.
    - Level 2 table: 1024 entries, each of which points to a page

Level 2 Tables
Level 1 Table
...

In fact, when my memory addressing capability increases, you can realize that my page table itself can become pretty large. So, if I have given let us look at these example 4KB page size 32 bit address space and you have got 4 byte entry page table entry 4 byte length. So, you need a 4 MB page table, which is default to manage and I cannot definitely have may be a 4 MB page table in Translation Look aside Buffer so, what do you have multilevel page tables? So, you have got this one page table so, level 1 table may have 1024 entries. Each of which points to a level to page table and level 2 table has got 1024 entries each of which point points to an actual page. So, in the TLB you actually, put in level 1 table and may be if you can accommodate 1 of these level 2 tables, when it is really a case of really multi level page tables. In fact, maturity of the system, if they really implementing virtual memory they would need to implement multi level page tables.

(Refer Slide Time: 44:10)

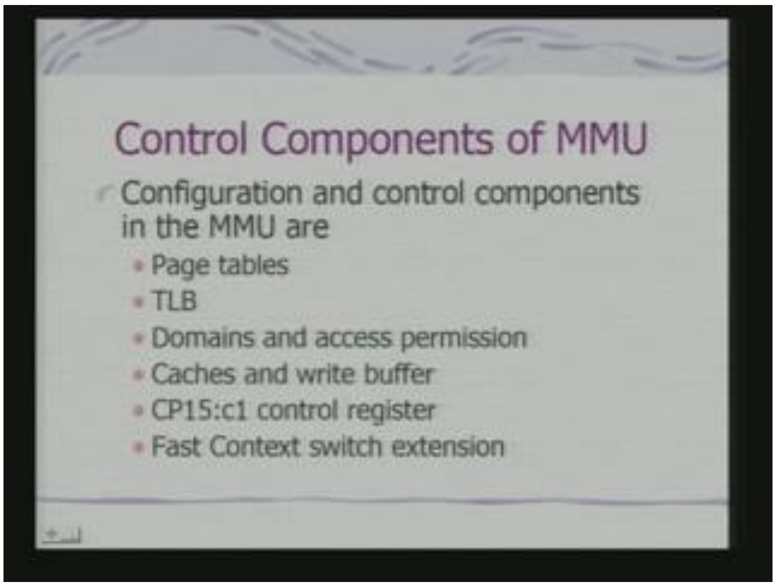


MMU in ARM

- ARM MMU performs several tasks
  - Translates virtual addresses into physical address
  - Controls memory access permission
  - Determines behavior of the cache and write buffer for each page in memory
- When MMU is disabled all virtual addresses map one-to-one to the same physical address
- MMU will only abort on translation, permission and domain faults

So, ARM, ARM core comes with MMU to support virtual memory and the primary motivation. So, providing this kind of MMU capability is not only to have demand paging, but most importantly to implement protection. So, like any other MMU, ARM, MMU also translates; obviously, virtual address into physical address controls memory access permission determines behavior of the cache. And write buffer for each page in memory, when MMU is disabled all virtual addresses map one to one to the same physical address. That means, for a simplest system you have the provision to disable MMU. So, your 32 bit addresses, which ARM generates, will be mapped onto directly physical addresses. And MMU will apart translation permission and domain faults. So, these are the exceptions that can MMU would generate under error conditions, so, configuration and control components in the MMU are obviously.

(Refer Slide Time: 45:19)



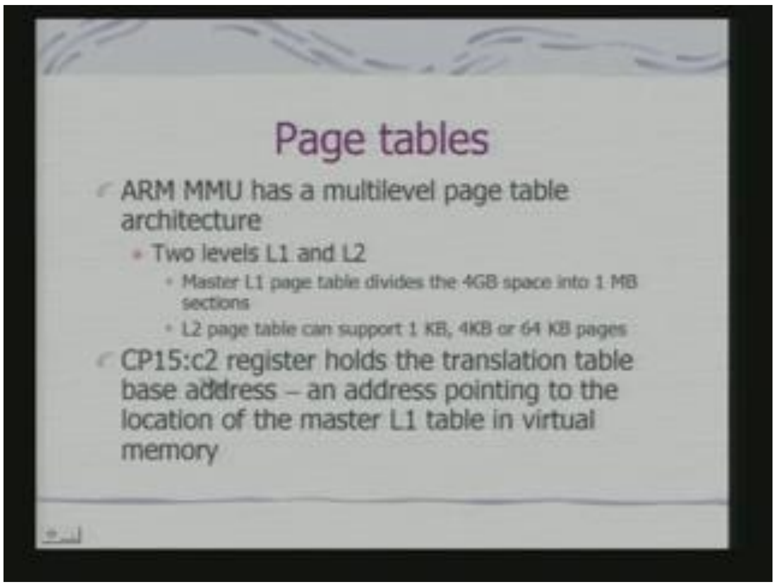
Control Components of MMU

- Configuration and control components in the MMU are
  - Page tables
  - TLB
  - Domains and access permission
  - Caches and write buffer
  - CP15:c1 control register
  - Fast Context switch extension

The page tables, the TLB domains and access permission. So, here we got of a concept of domain, which use in ARM used in ARM to define the permissions caches. And write buffers, how they are to be configured. Then CP15 this is I told you that system control coprocessor that manages again the MMU. It provides the management interface to the MMU and there is also additional hardware or extensions of this architecture, which provides for fast context switch. What is context switch? When I switch for one process to another process; obviously, you can understand the lots of things have to be saved and lot of things are needed onto the registers.

The simplest thing as that page table definitions have to be loaded. So, that introducers and overhead, because of these we may need to have hardware support. Particularly when we are talking about embedded applications, why because any interrupt actually would cause context switch. So, if I do not have a support for context switch then interrupt latency would be more. So, there are hardware supports in some of the architectures, which enables fast context switch. There is when your virtual memory is implemented with MMU you require first context switch provisions. So, that the overhead of loading the page tables ensuring consistence of the cache is taken care of.

(Refer Slide Time: 47:06)

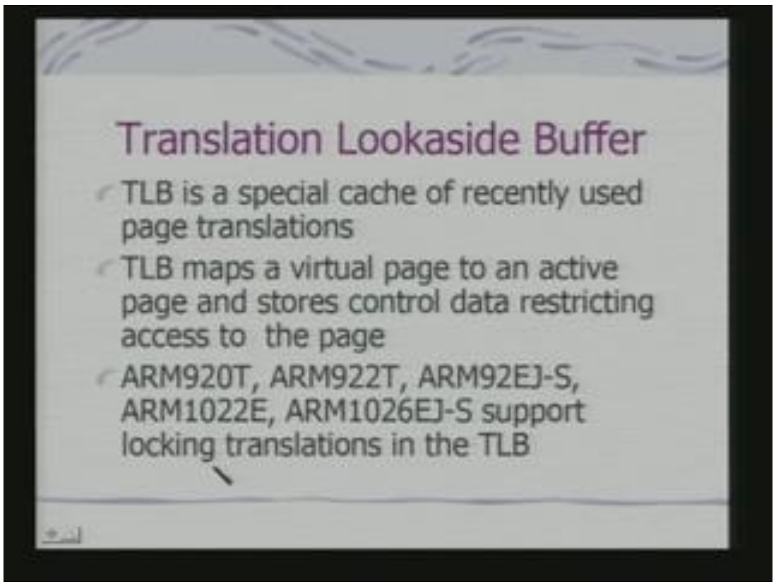


Page tables

- ARM MMU has a multilevel page table architecture
  - Two levels L1 and L2
    - Master L1 page table divides the 4GB space into 1 MB sections
    - L2 page table can support 1 KB, 4KB or 64 KB pages
- CP15:c2 register holds the translation table base address – an address pointing to the location of the master L1 table in virtual memory

In fact, ARM MMU supports multilevel page table, because this is you can realize that it is got to the 32 bit addressing address well locations. It has 2 levels we call them L1 level 1 and L2. The master L1 page table divides the 4GB space into 1 MB sections and L2 page table can support 1 KB 4 KB or 64 KB pages. So, different processes can use actually different size pages depending on the way L2 page table is configured. CP 15 C2 register holds the translation table base address an address pointing to the location of the master L1 table in virtual memory. This is the starting point fine so far a doing any kind of translation this base address is absolutely bottom. So, these base address I hope you remember I have talked about the starting address of the Page Table. So, this address is critically important for doing any kind of translation process and this is stored in the register of your coprocessor 50.

(Refer Slide Time: 48:27)



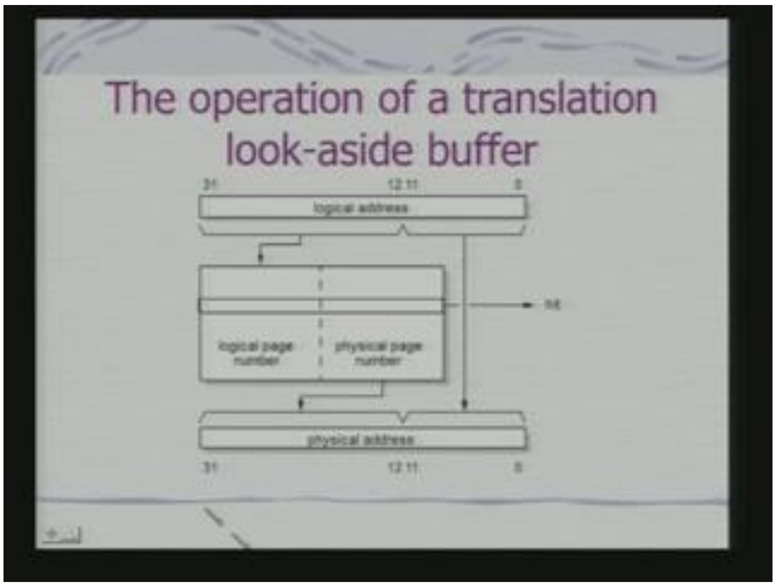
Translation Lookaside Buffer

- TLB is a special cache of recently used page translations
- TLB maps a virtual page to an active page and stores control data restricting access to the page
- ARM920T, ARM922T, ARM92EJ-S, ARM1022E, ARM1026EJ-S support locking translations in the TLB

Translation Look aside Buffer is that provision for that is in ARM MMU and like any expected operation. It maps a virtual page to an active page and stores control data about restricting access to the page. There is no difference in the function other than these part some of these ARM architectures; it supports locking translations in the TLB. Remember in the last class we have discussed the ability of the ARM to lock codes in the cache. The same thing is extended here for the TLB. So, TLB increase can be lock so, a pages a page tables, which are very frequently accessed for them the increase in the TLB would be locked. So, they will not be evicted so on.

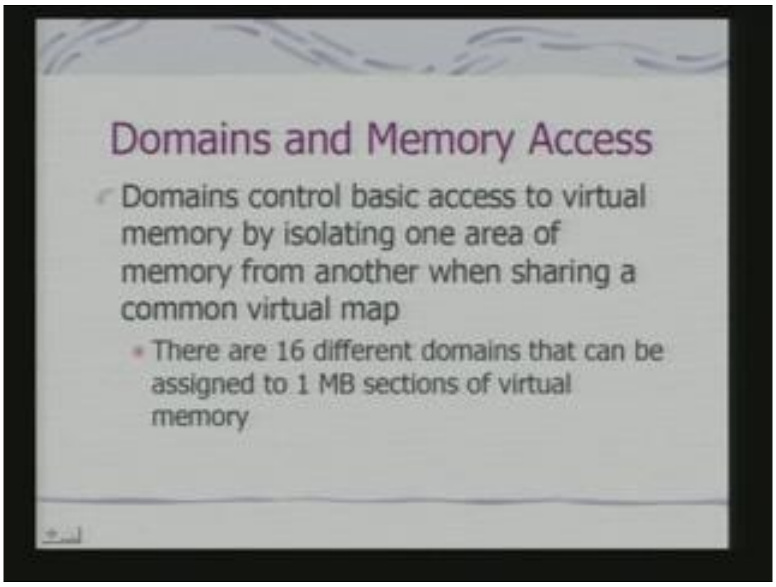
(Refer Slide Time: 49:21)

The operation of a translation look-aside buffer



So, the basic operation is this is your 2 bit address this a logical address these part is used for defining the logical page number. And then the table has got the physical page number. So, this physical page is added with this offset to get the physical address in an ARM system. So, this is how the complete operation takes place in an ARM TLB.

(Refer Slide Time: 49:51)

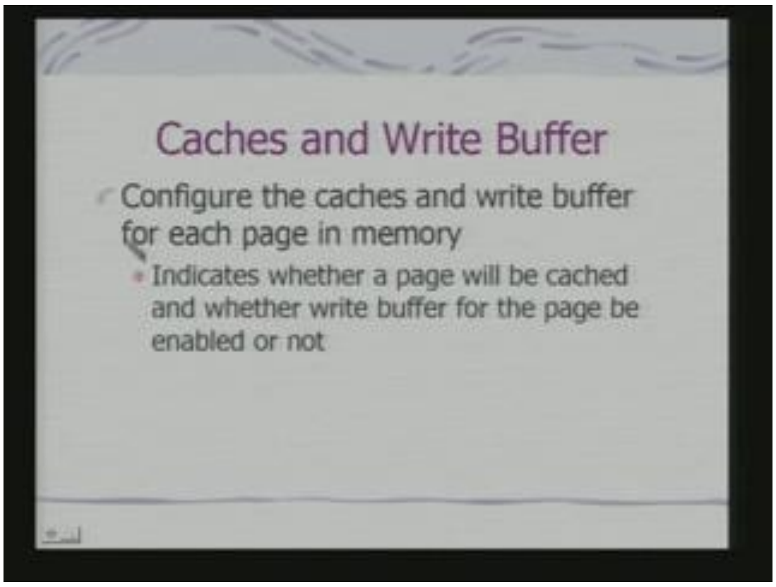


Domains and Memory Access

- Domains control basic access to virtual memory by isolating one area of memory from another when sharing a common virtual map
  - There are 16 different domains that can be assigned to 1 MB sections of virtual memory

Now, domains are used for defining the access permission domains control basics access to virtual memory by isolating one area of memory from another, when they are sharing a common virtual map. In fact, there are 16 different domains that can be assigned to 1 MB sections of virtual memory. So, domains actually define different kinds of access permissions. So, you can have a region so, that is what we call a 1 MB section, which is defined by the master page table. So, using this 1 MB sections, you can associate different kinds of access permissions. And based on this access permission the actual accesses will be permitted. In fact, here also you will find the ARM is providing 2 level of access permissions.

(Refer Slide Time: 50:48)

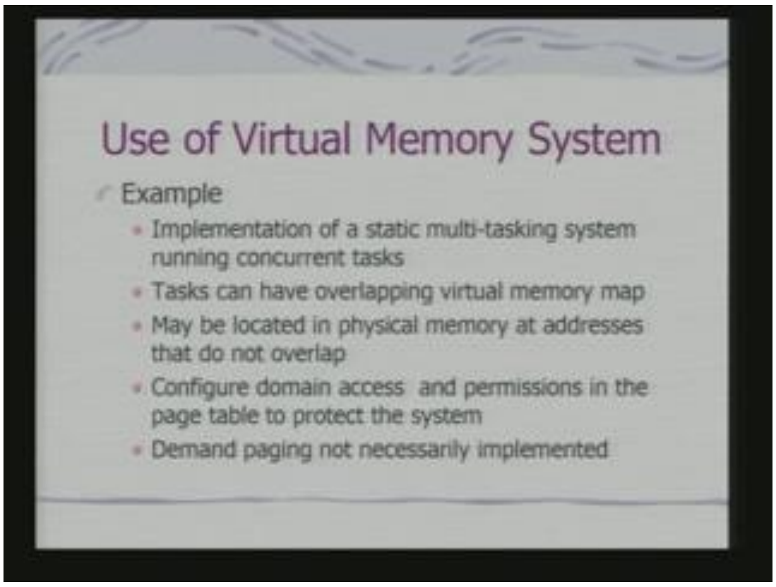


Caches and Write Buffer

- Configure the caches and write buffer for each page in memory
  - Indicates whether a page will be cached and whether write buffer for the page be enabled or not

In fact, one at one level there is access permission associated with definition of the domain. And getting domains associated with the sections and there would be also access permissions associated with pages. Now, you have caches and write buffer. So, you can configure whether a particular page will be cache or not or whether you can have a write buffer associated with a page or not. So, what we say that, when we have a page whether it will be cache or not or whether write buffer for the page be enabled or not. Now, these can be done by setting appropriate control can be sense in the MMU. So, I can have pages, but the page did not be really cache. In fact, this is reflection of the same condition that I discussed in the last class that some of the pages cannot be really cached.

(Refer Slide Time: 51:53)

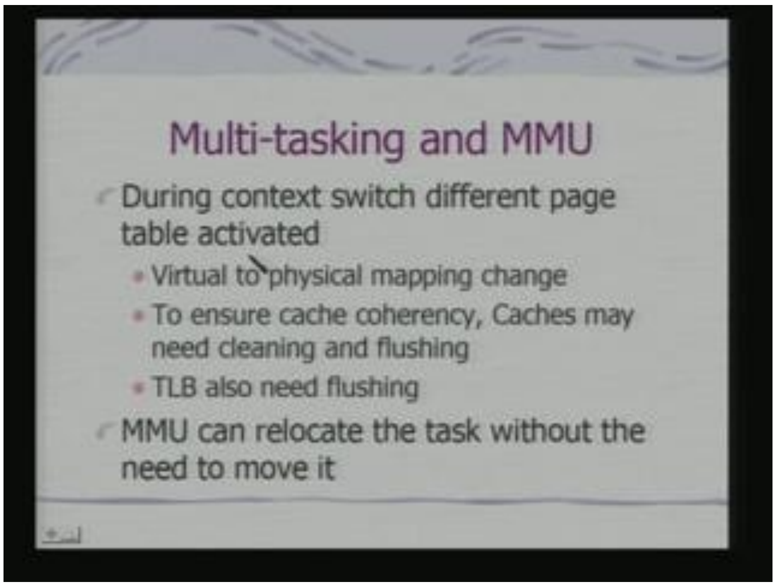


Use of Virtual Memory System

- Example
  - Implementation of a static multi-tasking system running concurrent tasks
  - Tasks can have overlapping virtual memory map
  - May be located in physical memory at addresses that do not overlap
  - Configure domain access and permissions in the page table to protect the system
  - Demand paging not necessarily implemented

The let us see how you will be using a virtual system a typically in a kind of an embedded application for implementation of a static multitasking system running concurrent tasks. So, an Embedded System, it is normally not expected you will be having a dynamic multitasking system. That it tasks be generated dynamically other than compute a like embedded systems like your PDS. In most of the cases in many of the dedicated systems that would be a set of fixed task, which in to run concurrently. Tasks can have overlapping virtual memory map overlapping virtual memory map. In fact, they can have same or identical virtual map, but they will be located in physical memory add addresses they do not overlap. It may also overlap, when they share pages when they share common pages that may also overlap then you configure your domain access. And permissions in the page table to protect the system so, each task will have associated with it a domain access. And that will restrict the access rights to the pages belonging to the task and demand paging is not necessarily implemented.

(Refer Slide Time: 53:13)

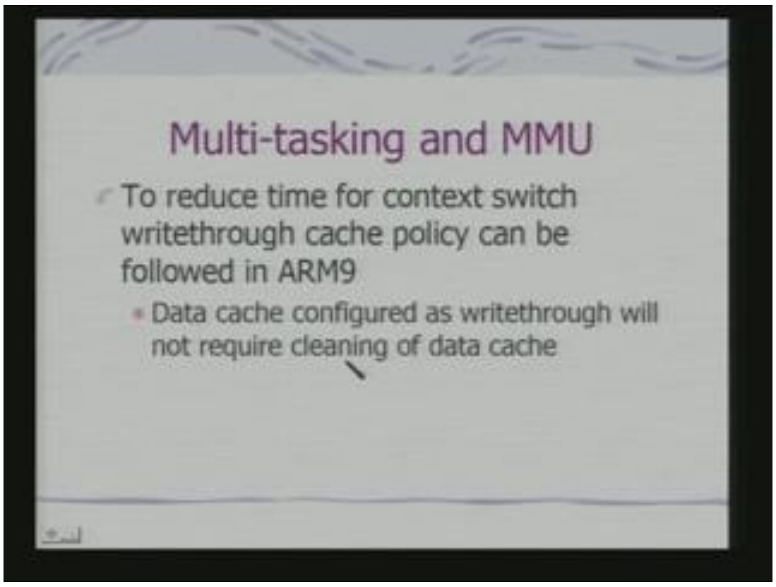


Multi-tasking and MMU

- During context switch different page table activated
  - Virtual to physical mapping change
  - To ensure cache coherency, Caches may need cleaning and flushing
  - TLB also need flushing
- MMU can relocate the task without the need to move it

So, during context switch different page tables is activated, so, it will switch from 1 page table to another page table. Because virtual to physical mapping change and to ensure cache coherency caches may need cleaning and flushing. Because I may not like the entries in the cache to remain, because they have they can be removed and flush advent go flushing would be effectively removing the contents advent go. TLB will also need flushing, because I am now, using different pages. So, this is the basic overhead for the context switching on. In fact, when we have context switching hardware first context switching extension in ARM. You have the hardware support to do this things fast. So, reduce the time for context switching. So, another policy, which is implemented, is write through cache policy.

(Refer Slide Time: 54:04)

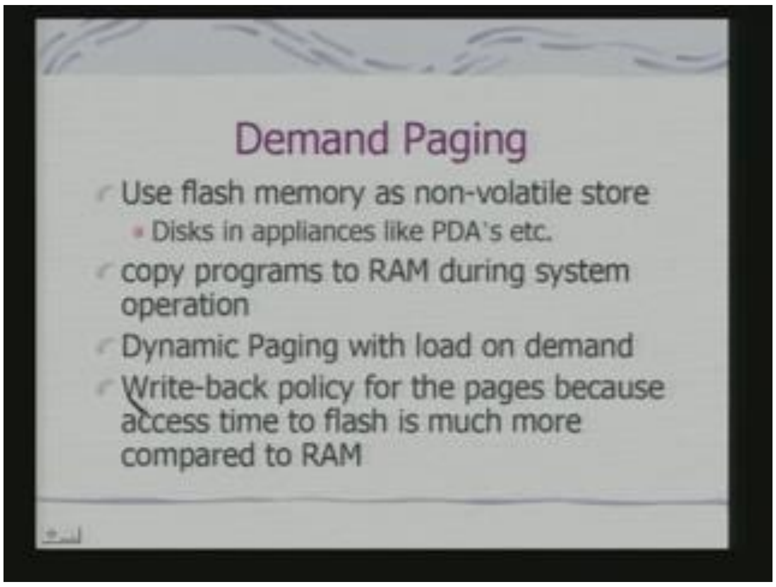


Multi-tasking and MMU

- To reduce time for context switch writethrough cache policy can be followed in ARM9
  - Data cache configured as writethrough will not require cleaning of data cache

So, in the case data cache is configured as write through will not require cleaning of data cache, why because I can simply forget about the contents of this data cache. Because every write to the data cache, I am writing back to the memory, but; obviously, it has got the drawback in terms of additional time consume for writing into the memory.

(Refer Slide Time: 54:35)

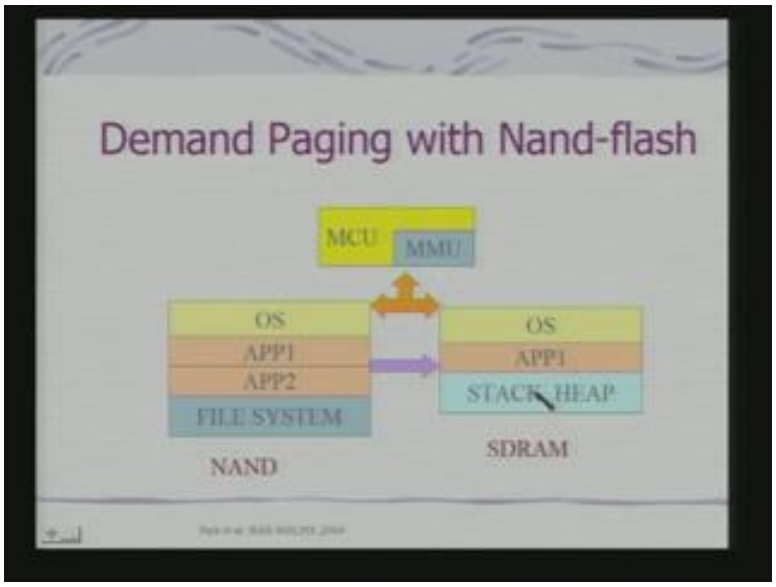


Demand Paging

- Use flash memory as non-volatile store
  - Disks in appliances like PDA's etc.
- copy programs to RAM during system operation
- Dynamic Paging with load on demand
- Write-back policy for the pages because access time to flash is much more compared to RAM

You use demand paging, you use in many cases in the maturity of the cases for embedded system flash memory as non volatile store. In fact, you have disc or file systems actually implemented on flash memory and you have programs, which are stored in the flash. Typically, your OS will be stored in the Flash and you copy from flash to RAM during execution for faster access. So, under those conditions, you can use dynamic paging with loading pages on demand. So, under those conditions you typically adopt write back policies for the pages. Because access time to flash is much more compared to RAM see would not like write through policy to be implemented for the pages. Now, this we are talking about the write back or write through policy of the pages and not of cache blocks please keep that in mind.

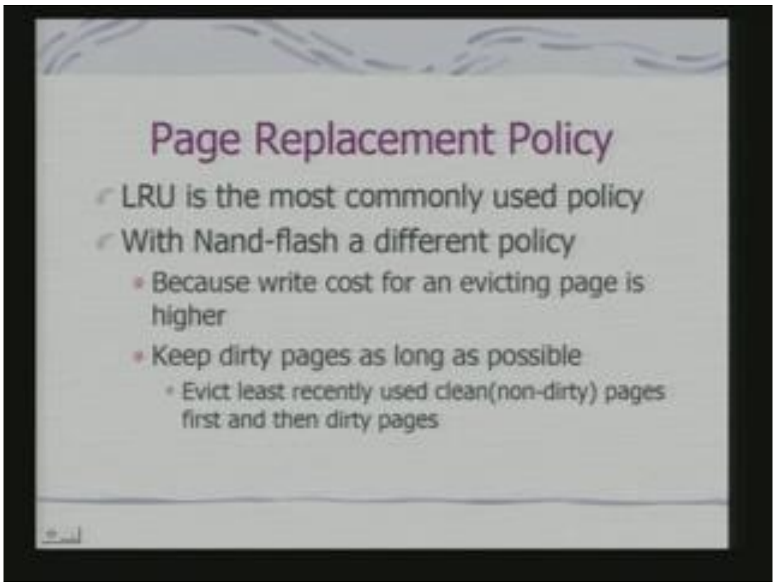
(Refer Slide Time: 55:38)



Demand Paging with Nand-flash

So, if you are using a NAND flash, NAND flash is that flash which has got very ((refer time 55:40)). It can have fast burst mode access, but its random mode access is not that efficient like your Nor flash. So, if you are really using NAND flash because of its density of storage then what you can really use is Demand Paging. So, in your NAND flash, you have the OS, you have the code for applications application 1 application 2. You have the complete file system and you have this MMU sitting with a microcontroller like your ARM MMU. And when you start of the system actually OS gets loaded onto your SDRAM. In fact, these pages can actually be locked. So, that the ((refer time: 56:25)) moved out of your SDRAM then depending on the application requirements pages of the application gets loaded onto SDRAM. On for running that application, you would need stack and hit that pages also could be accommodated in the SDRAM. So, the page gets loaded on demand on the SDRAM from the NAND file systems. So, this is an example where you can use Demand Paging in Embedded appliance. But when we are using NAND flash the new page replacement policy me needs some modification, because we know that LRU is that most commonly used policy.

(Refer Slide Time: 57:01)

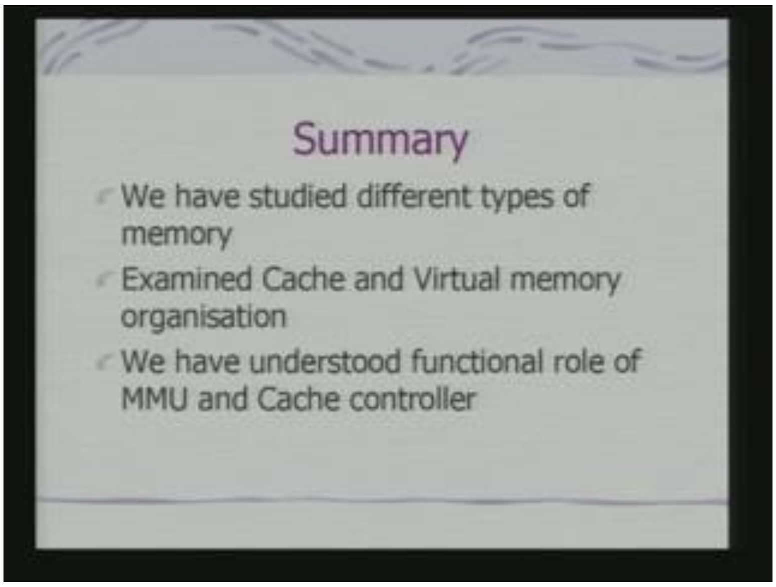


Page Replacement Policy

- LRU is the most commonly used policy
- With Nand-flash a different policy
  - Because write cost for an evicting page is higher
  - Keep dirty pages as long as possible
    - Evict least recently used clean(non-dirty) pages first and then dirty pages

What is that least recently used page to be replaced when do not have enough pages in the physical memory with NAND flash a different policy is followed. Because write cost for an evicting page is higher, because you are writing onto the flash. So, keep dirty pages as long as possible. So, you modify the policy and say that you evict least recently used clean pages first and then dirty pages. So, you ((refer time: 57:35)) overhead of Write back you can see the in the context of Embedded Systems all these well known policies also under ((refer time: 57:43)) And why LRU you set is an policy, because it is least recently access; that means, it will not be accessed in future, because of the principle of temporal locality and spatial locality. This finishes are study of memory systems in embedded systems.

(Refer Slide Time: 58:04)



Summary

- We have studied different types of memory
- Examined Cache and Virtual memory organisation
- We have understood functional role of MMU and Cache controller

We have examined cache and virtual memory organization also understood the functional role of MMU and cache controller. We shall look into other aspects of embedded system hardware particularly the bus definition in next few lectures.

.