Computer Organization and Architecture: A Pedagogical Aspect

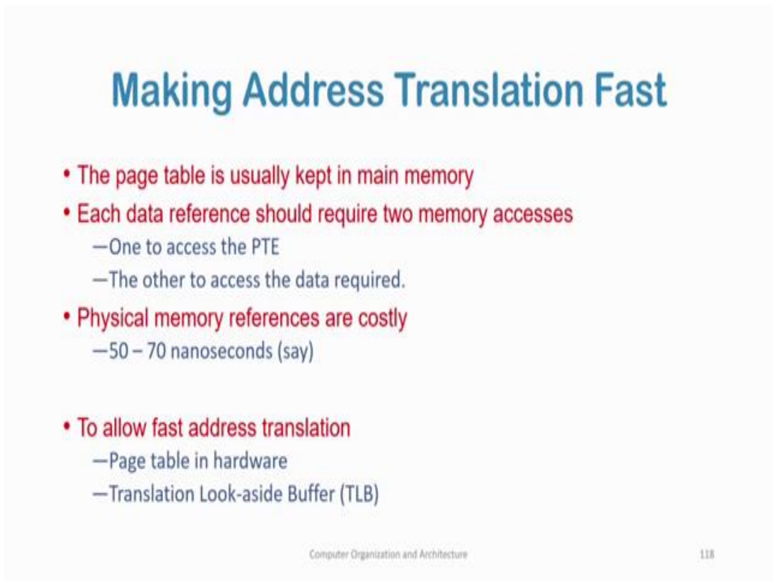
Prof. Jatindra Kr. Deka
Dr. Santosh Biswas
Dr. Arnab Sarkar
Department of Computer Science & Engineering
Indian Institute of Technology, Guwahati

Lecture – 29
TLBs and Page Fault Handling

In the last lecture we saw that in the absence of any measure the sizes of page tables can be huge. Therefore, we studied different techniques by which the sizes of page tables can be controlled.

This lecture we will start with the discussion on how address translation using page tables can be made faster.

(Refer Slide Time: 00:52)



Making Address Translation Fast

- The page table is usually kept in main memory
- Each data reference should require two memory accesses
  - —One to access the PTE
  - —The other to access the data required.
- Physical memory references are costly
  - —50 – 70 nanoseconds (say)
- To allow fast address translation
  - —Page table in hardware
  - —Translation Look-aside Buffer (TLB)

Computer Organization and Architecture 118

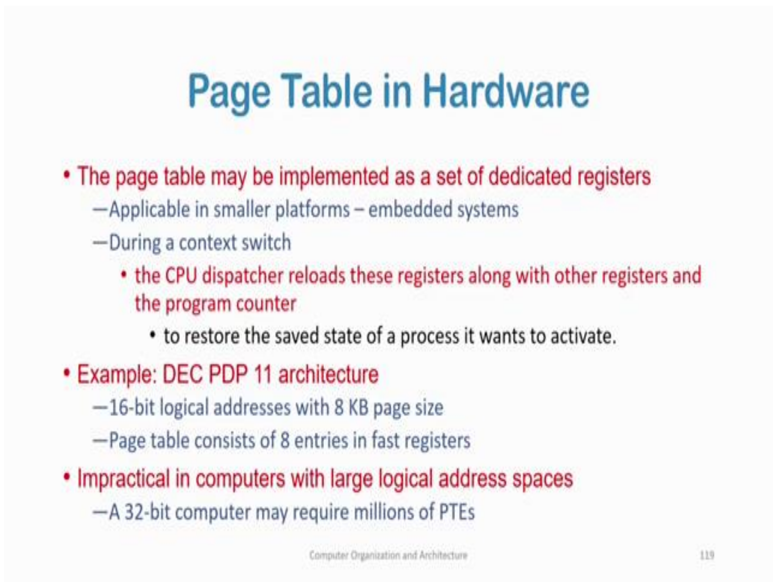
What is the motivation? As we discussed page tables are usually kept in main memory; therefore, for each data reference we will typically require two memory accesses if we do not take any measure.

One access of which will be to access the page table entry itself and then when we access the page table entry we get the main memory address or we get the main memory page number and we generate the physical address subsequent to that and then the second memory reference will be to access the actual data that we require from main memory.

Now, main memory references are typically very costly with respect to if we had found the data in cache let us say; main memory accesses we said would be of the order of 50 to 70 nanoseconds; as against cache which could be around 5 to 10, 1 to 10 nanoseconds ok, around 5 to 10 nanoseconds.

And therefore, it is necessary that we take action to reduce this page table access in main memory. Now, there are two typical strategies which are employed here. The first one is to implement the page table in hardware and the other one is to use a translation lookaside buffer.

(Refer Slide Time: 02:20)



Page Table in Hardware

- The page table may be implemented as a set of dedicated registers
  - —Applicable in smaller platforms – embedded systems
  - —During a context switch
    - the CPU dispatcher reloads these registers along with other registers and the program counter
      - to restore the saved state of a process it wants to activate.
- Example: DEC PDP 11 architecture
  - —16-bit logical addresses with 8 KB page size
  - —Page table consists of 8 entries in fast registers
- Impractical in computers with large logical address spaces
  - —A 32-bit computer may require millions of PTEs

Computer Organization and Architecture 119

When we implement the page table in hardware how do we do it? We implement the page table using a dedicated set of registers and obviously, it is applicable for systems where the page table sizes will typically be smaller. For example, in embedded systems. Now, during a context switch when a new process has to be brought into the CPU.

The CPU dispatcher will reload these page table registers along with other registers and the program counter. In order to restore the save state of a process and activated; so, basically during a context switch what happens? The saved state of a process is brought into cache.

And what are what is the typical what do we mean by the saved state of a process? The contents of its page table, the contents of its other registers and also the program counter.

So, therefore, after the context switch when the process is brought into memory because we have saved program counter known, so the process can start from the from the place where it was previously evicted from CPU.

Now, when the page table is in hardware I have to reload all the registers in page table during a context switch because that is part of the saved state. If the page table is in memory it is sufficient to load the page table base register corresponding to this process. Because the page table is in memory; I only need where in memory the page table starts.

So, therefore the page table base register only needs to be brought in during a context switch. However, when I am implementing this page table in hardware I need to bring in the entire set of registers which is the page table into the hardware after the during the context switch.

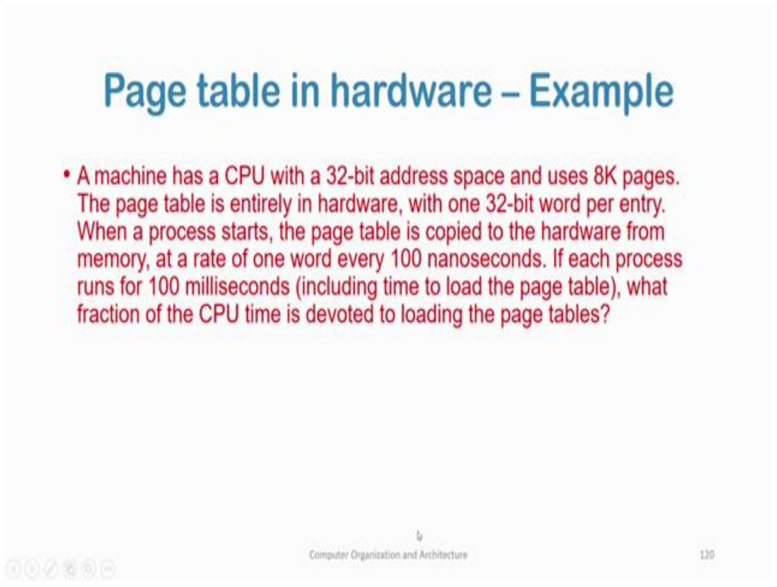
An example of such hardware implemented page tables is the DEC PDP 11 architecture. The DEC PDP 11 architecture is a 16 bit small computer. So, it has 16 bit logical address space with 8 KB page size. So, therefore, if it is a page 8 KB page size, so therefore, the offset part of the virtual address is 2^{13} bits. So, $2^{10\times3}$, 8 KB; so, 2^{13} bits is the size of the page offset part in the virtual address.

So, the number of pages is just, it consumes only 3 bits; so, 16 - 13, 3 bits. So, therefore, the page table the system contains only 8 pages and therefore, the page table consists of 8 entries in fast registers. However obviously, such hardware implementation of page tables are impractical for computers with very large address spaces.

For example, if we have a 32 bit computer and let us say in that computer we use 4 KB pages for example, then I will use 12 bits for the page offset part and therefore, I will have 20 bits for the page numbers.

And therefore, we have $2^{10} \times 2^{10}$; 2^{20} entries in the page table which is 1 million; 2^{20}; 1 million page table entries and obviously, such very huge sizes of page tables cannot be implemented through registers in hardware.

(Refer Slide Time: 06:10)



Page table in hardware – Example

- A machine has a CPU with a 32-bit address space and uses 8K pages. The page table is entirely in hardware, with one 32-bit word per entry. When a process starts, the page table is copied to the hardware from memory, at a rate of one word every 100 nanoseconds. If each process runs for 100 milliseconds (including time to load the page table), what fraction of the CPU time is devoted to loading the page tables?

Computer Organization and Architecture 120

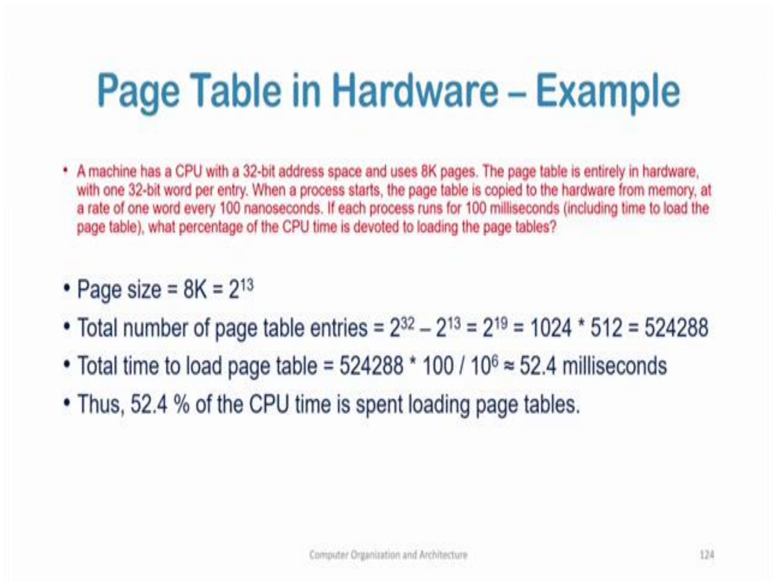
Before proceeding we take an example. A machine has a CPU with a 32 bit address space and uses 8K pages. The page table is entirely in hardware with one 32-bit word per entry. When a process starts the page table is copied to the hardware from memory. At a rate of one word every 100 nanoseconds.

So, therefore, the page table is in hardware. So, we are trying to look at what is the problem for big computers in a 32 bit computer if the entire page table is in memory. We will look at one of the computers in addition to space.

So, the machine has 32 bit address space and it uses 8K pages; the page table is entirely in hardware with one 32-bit entry per word 32-bit word per entry. When a process starts the page table is copied to the hardware from memory at a rate of one word every 100 nanoseconds.

So, to copy one word from memory it takes 100 nanoseconds. If each process runs for 100 milliseconds and this includes the time to load the page table what fraction of the CPU time is devoted to loading the page tables.

(Refer Slide Time: 07:34)



Page Table in Hardware – Example

- A machine has a CPU with a 32-bit address space and uses 8K pages. The page table is entirely in hardware, with one 32-bit word per entry. When a process starts, the page table is copied to the hardware from memory, at a rate of one word every 100 nanoseconds. If each process runs for 100 milliseconds (including time to load the page table), what percentage of the CPU time is devoted to loading the page tables?

- Page size = 8K = 2^{13}
- Total number of page table entries = $2^{32} – 2^{13} = 2^{19}$ = 1024 * 512 = 524288
- Total time to load page table = 524288 * 100 / $10^6 \approx$ 52.4 milliseconds
- Thus, 52.4 % of the CPU time is spent loading page tables.

Computer Organization and Architecture 124

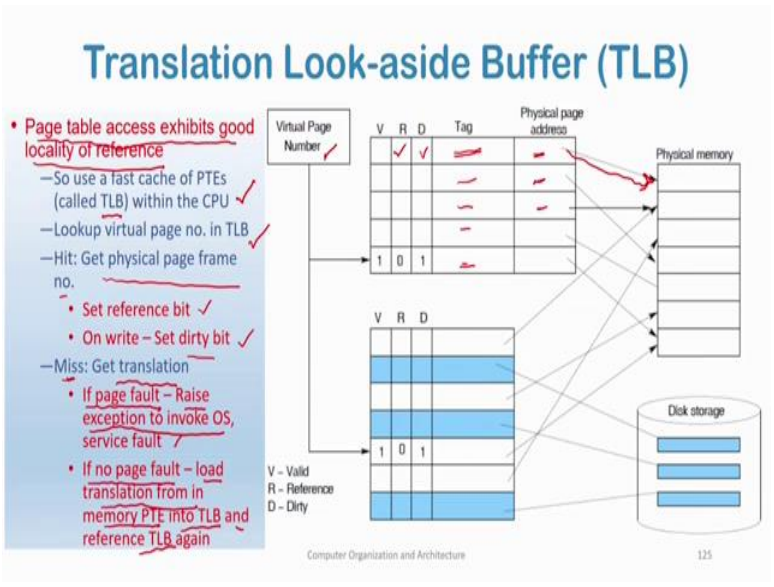
So, the page size is 8K or 2^{13}. So, 13 bits we consume for the page tables. So, therefore the total number of page table entries becomes $2^{32} - 2^{13}$ which is 2^{19}; which is $2^{10} \times 2^9$ or 524288. So, these many page table entries are there in the page table. All these have to be copied from memory into hardware during a context switch.

So, the total time to load the page table will be what; 524288 and each of them consumes 100 nanoseconds. If you convert it to milliseconds divide by 10^6, we get 52.4 milliseconds. So, therefore, 52.4 percent of the total CPU time in this case we spent loading page tables.

So, I have a time slot size in which I will execute a process and I execute a process for 100 milliseconds, for each process I give 100 milliseconds and out of that 52.4 milliseconds is only consumed for loading the page table from memory into hardware.

So, basically I am not being able to do anything any good work other than loading the page table from memory into hardware which is very costly. So, therefore, when the page table is in hardware it is only possible in cases where the address space virtual address space sizes are small and we have a small number of page table entries.

(Refer Slide Time: 09:25)



Translation Look-aside Buffer (TLB)

- Page table access exhibits good locality of reference
  - So use a fast cache of PTEs (called TLB) within the CPU
  - Lookup virtual page no. in TLB
  - Hit: Get physical page frame no.
    - Set reference bit
    - On write – Set dirty bit
  - Miss: Get translation
    - If page fault – Raise exception to invoke OS, service fault
    - If no page fault – load translation from in memory PTE into TLB and reference TLB again

Computer Organization and Architecture

125

So, then how to what is the other way around when we have large address spaces? So, how to control the time consumed for accessing page tables and doing address translation for large computers with large virtual address spaces?

For example, say 32 bit computers, how will we do that? We typically do that using in memory page tables. So, we now we don’t keep the page tables in hardware. We keep the page tables in memory itself. However; we use the property of the locality of reference of page tables. So, therefore, here is the first point page table access page table access exhibits good locality of reference.

So, therefore, once a page table entry is accessed it is supposed to be accessed soon again. So, the same page because our memory references, our memory accesses tend to be clustered in both time as well as space. So, the data that I am consuming nearby data I can I would consume in the recent future; nearby data I will access in the recent future and the same data I may access again in the recent future.

So, I have both temporal locality as well as spatial locality for page memory accesses. And therefore, the same happens for page table accesses page table entries, the same page table entries is likely to be accessed soon again. So, therefore we can use a fast cache for page table entries and this cache is called the translation lookaside buffer within the CPU.

So, we look up the virtual page number in the TLB. So, what do we have? We get the virtual page number is generated by the process that is the CPU and I look up that virtual I divide that virtual page number into the page offset part and the page number part. So, the page number part is now floated to the TLB. So, the TLB, the TLB the stack part of the TLB contains virtual page numbers.

So, we look up the virtual page number in the TLB. If I get a match, if I get a match with the tag I get the corresponding physical page number from the data part of this TLB. So, the tag part of the TLB contains a virtual page number and the data part of this TLB contains the corresponding physical page number. So, when I get the physical page number I can access the physical memory by adding up with the page offset which is directly obtained from the virtual address itself.

Now, during a hit what happens? We get the physical page number and we get the and as I said I get I can get the data from physical memory. I said the physic I said the reference bit.

So, this one R is the reference bit; I set the reference bit to indicate that in this in the in the in the current time epoch I have referenced this page to indicate that I set the reference bit. If this I am writing on to this on to main memory, if I am writing on to main memory I also set the dirty bit on; So, therefore, to indicate that this page has been modified.

However, when I get a miss, if my virtual page number does not match with any of the tags present in the in the in the translation lookaside buffer, then I have a miss in the page table ok. If I have a miss in the page table I need to get the translation from memory from the page table. Now, in this case there can be two distinct cases that I have a miss in the TLB, but the data is in memory.

Therefore, the data is also in the page table and I just get the translation from the corresponding page table entry in memory and I bring it back into the TLB and then reference the TLB again and get the data from physical memory. Otherwise I can induce a page fault.

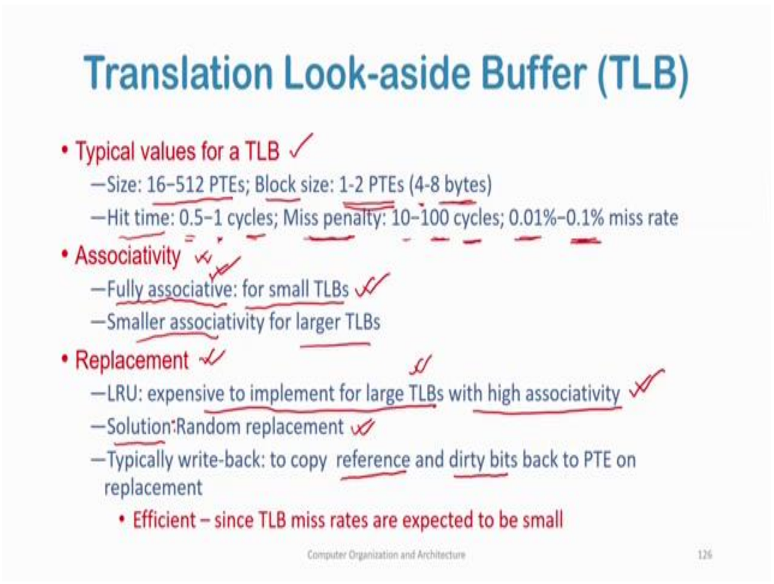
If the data that I am looking for the corresponding to the virtual page number there is this physical page number translation this virtual page number to physical page number translation is not there in the page table in the memory itself; which means that the page is not there in the memory I incur a page fault.

And therefore, then I need to find out a free page frame in the memory, bring the page from the disk into main memory and then populate the page table accordingly and half subsequently I need to bring that page table into the translation lookaside buffer and then subsequently when I access I get a hit in the TLB and I get the page number from the TLB itself and you and access the physical memory.

So, if there is a page fault; that means, that the page is not there in the memory. The CPU raises an exception; that means, it traps the OS; it traps to the OS it generates a trap instruction and the OS is invoked I go to the supervisor mode as the OS is invoked to service the fault in the way as we said we will see it in more depth in the in the subsequent slides.

How if there is no page fault; that means, I load the translation from where; from the in memory page table, from the in memory page table I bring the page table entry from the in memory page table into the TLB and reference the TLB again; I reference the TLB again. Now, I get a hit in the TLB and access physical memory.

(Refer Slide Time: 15:48)



Translation Look-aside Buffer (TLB)

- Typical values for a TLB
  - Size: 16–512 PTEs; Block size: 1-2 PTEs (4-8 bytes)
  - Hit time: 0.5–1 cycles; Miss penalty: 10–100 cycles; 0.01%–0.1% miss rate
- Associativity
  - Fully associative: for small TLBs
  - Smaller associativity for larger TLBs
- Replacement
  - LRU: expensive to implement for large TLBs with high associativity
  - Solution: Random replacement
  - Typically write-back: to copy reference and dirty bits back to PTE on replacement
    - Efficient – since TLB miss rates are expected to be small

Computer Organization and Architecture 126

So, a few more details with the translation lookaside buffer. Typical values for the TLB are as follows. Typical for a TLB: the size of a TLB are of the order of let us say 16 to 512 page table entry. So, page table entries and each such page table entry could be 4 to 8 bytes. So, therefore, TLBs are typically small in size. The block size for a TLB is 1 to 2 page table entries and if a page table entry is let us say 4 bytes; so, 2 page table entries; therefore, mean just 8 bytes.

The hit time of a TLB is very fast typically of the order of 0.5 to 1 cycle. A miss penalty would result in what? Would result in going to the lower level of memory and therefore, I it will be around 10 to 100 cycles. Why, why this big difference? Firstly, because different architectures can support different and different access times and a miss penalty for corresponding to TLB miss, I may get the data either in the data cache for the corresponding TLB entry or I may not have the data corresponding to the TLB entry in the data cache. In that case the TLB entry must be brought from the memory itself.

So, therefore, there can be two possible cases when I have a miss and I want to get the page table entry into the TLB. And typical miss rates for a TLB are of the order of 0.01 to 0.1 1 percent. So, therefore, the locality of reference for corresponding to TLB entry is typically very high and so, smaller TLBs suffice. So, it’s around 99.9 percent to 99.99 percent is the hit rate.

The associativity of a TLB: TLBs are very often implemented in a fully associative fashion; which means that all the entries of the TLB; So, which means that because the TLB is small, I have the option because the TLB is small what happens is that; when it is fully associative the page table entry can be brought in to any entry of the TLB ok when it is fully associative.

I don’t restrict the page table a page table entry to go into certain specific locations in the TLB. The entire page table all entries in the page table can hold any all entries in the TLB can hold any page table entry when it is fully associative.

The drawback is that I when it is fully associative all the entries of the TLB must be searched in parallel for the tag. So, therefore, the search time and also the replacement time we will as we will see later becomes high and therefore, the cost of a fully associative TLB is high.

And this is affordable for small TLBs; however, when the sizes of TLBs grow, we nowadays we have a higher sizes of TLBs; smaller associativity’s are preferred as we understand why.

Replacement strategies how do we replace TLB entries? When I do not find when there is a miss in the TLB, I need to replace. So, least recently used TLB entry if I want to replace this is typically expensive to implement for large TLBs with high associativity. Why? Because when the TLB sizes are large finding the least recently used in implementing the least recently used in hardware finding which is the least recently used at any time which entry, keeping that account is expensive.

How do we keep account at any time which entry is least recently used? So, therefore, I need to maintain some kind of a clock or a queue corresponding to each TLB entry and that has to be implemented in hardware. So, that at any point in time I can keep track of the least recently used TLB entry.

And when the associativity becomes higher, this search or keeping this maintaining this becomes higher. Because the number of places within, for example, when it is fully associative any entry in the TLB can be least recently used. So, I may need to search the entire TLB. In a set associative one, I only need to search within the set. So therefore; when the associativity is high and when the associativity is high, it is expensive to implement for large TLBs; LRU is expensive to implement for large TLBs, especially and when the associativity is high.

So, what is the solution? It is a bypass. What is the solution? The bypass is that we use random replacement. I just find out a TLB and replace it randomly. I do not go for the least recently used. For a TLB we typically implement write back instead of a write through.

So, why do we write? We copy the reference and dirty bits back into the TLB entry on replacement. We said that when there is a TLB hit, what happens is that I may set the reference bit, I may set the reference bit to indicate that this page table entry was referenced. I may also said the dirty bit to indicate that that this page table this page table entry was written to this page was written to ok.

Now, when I am replacing this TLB entry I must write back the reference and dirty bits the values of the reference and dirty bits into the corresponding page table entry in memory. And I and I do not dos this and I am using a write back means that, I don't do this writing when I am basically changing the reference or dirty bits at a given page table entry access. That becomes very expensive because I have to write it back to the next level to the page table entry in memory if I want to write through and this becomes very costly.

And as we said that the page table miss rates are expected to be small as we saw it is of the order of 0.01 percent to 0.1 percent. So, page table miss rates are small. So, write back is very good in this case and it saves a lot of time and the implementation is much more efficient when we use a write back scheme instead of a write through and that is I write only during replacement and I don't write and I don't write on every reference to the page table entry, every reference to the page table.