Memory Management

Week 06 – Lecture 2

Virtual Memory

Team

Instructors

Giancarlo Succi

Joseph Brown

Teaching Assistants

Vladimir Ivanov

Stanislav Litvinov

Alexey Reznik

Munir Makhmutov

Hamna Aslam

Sources

- These slides have been adapted from the original slides of the adopted book:
  - Tanenbaum & Bo, Modern Operating Systems: 4th edition, 2013
    Prentice-Hall, Inc.

  and customised for the needs of this course.
- Additional input for the slides are detailed later

Virtual Memory (1)

- There is a need to run programs that are too large to fit in memory
- Swapping is not a solution since transfer rate of a hard disk is to slow and it will take seconds to swap in or out a large program
- Solution adopted in the 1960s, split programs into little pieces, called **overlays**
  - Kept on the disk, swapped in and out of memory

Virtual Memory (2)

- When a program started, only the overlay manager was loaded into memory
- It ran overlay 0 and, when it was done, the overlay manager loaded overlay 1 either above overlay 0 or on top of it (if there was no space)
- The work of splitting the program into pieces had to be done manually by the programmer

Virtual Memory (3)

- **Virtual memory**: each program has its own address space, broken up into chunks called **pages**
- Each page is a contiguous range of addresses and is mapped onto physical memory
- Not all pages have to be in physical memory at the same time to run the program. When the program references a part of its address space that is not in physical memory, the OS gets the missing piece and re-executes the instruction that failed

Paging (1)

- On any computer, programs reference a set of memory addresses which can be generated using indexing, base registers, segment registers, and other ways
- These program-generated addresses are called **virtual addresses** and form the **virtual address space**
- When virtual memory is used, the virtual addresses go to an MMU (Memory Management Unit) that maps the virtual addresses onto the physical memory addresses (Fig. 3-8)
- MMU is commonly a part of the CPU chip nowadays. However, logically it could be a separate chip and was years ago

Paging (2)

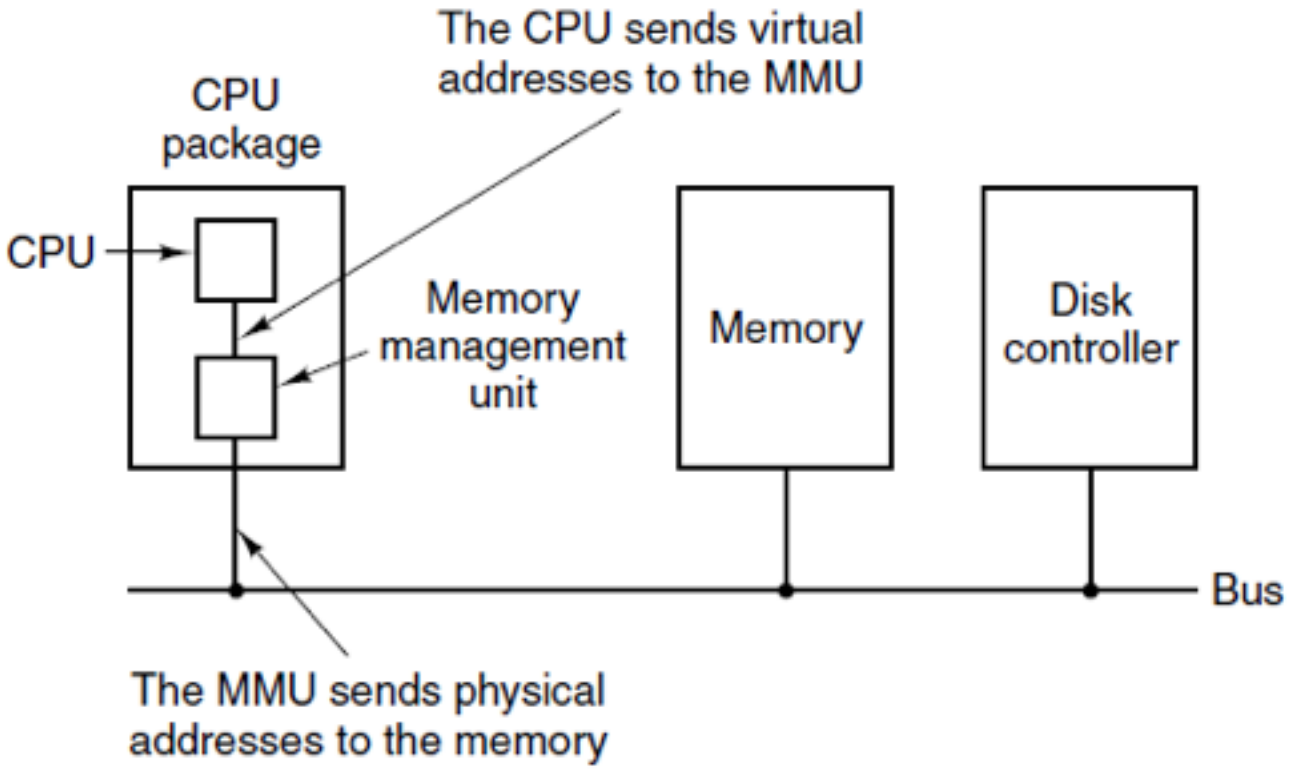

Figure 3-8. The position and function of the MMU

Paging (3)

- Paging example:
  - The computer has 32 KB of physical memory
  - It generates a set of 16-bit virtual addresses from 0 to 64K - 1
  - The virtual address space consists of fixed-size units called pages
  - The corresponding units in the physical memory are called **page frames**
  - Transfers between RAM and disk are always in whole pages

Paging (4)

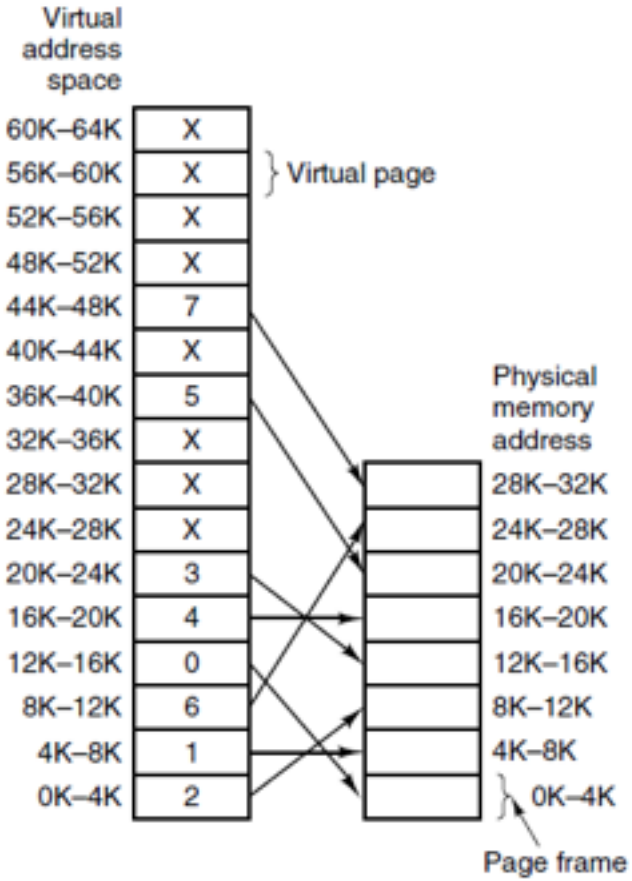

Figure 3-9. The relation between virtual addresses and physical memory addresses

Paging (5)

- The range marked 0K–4K means that the virtual or physical addresses in that page are 0 to 4095, the range 4K–8K refers to addresses 4096 to 8191, and so on
- When the program tries to access address 0, for example, using the instruction MOV REG,0 virtual address 0 is sent to the MMU which according to the mapping converts it to 8192 (first address of page frame 2)

Paging (6)

- Other examples:
  - the instruction MOV REG,8192 is converted into MOV REG,24576
  - virtual address 20500 is 20 bytes from the start of virtual page 5 (virtual addresses 20480 to 24575) and maps onto physical address 12288 + 20 = 12308

Paging (7)

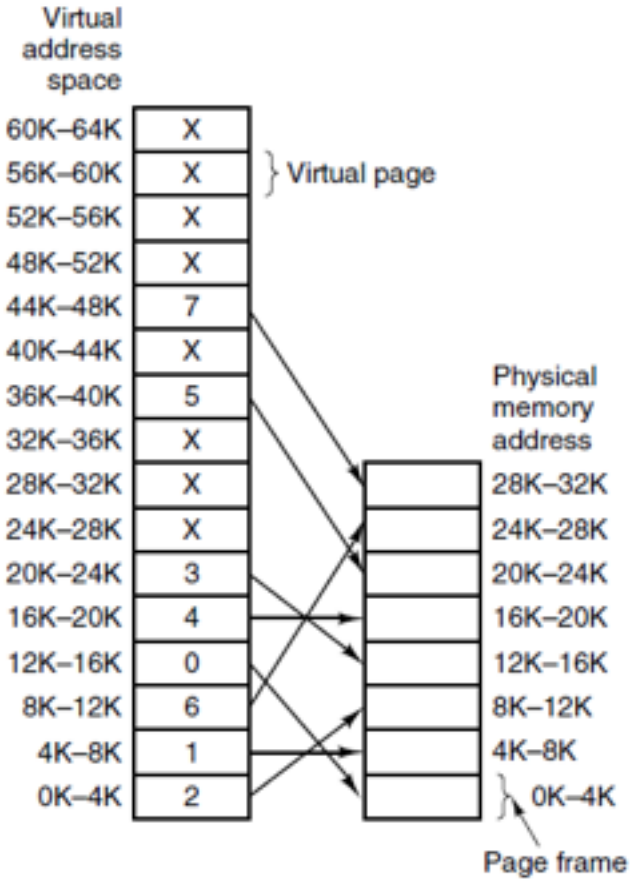

Figure 3-9. The relation between virtual addresses and physical memory addresses (notice X signs in virtual memory)

Paging (8)

- MMU's map does not solve the problem that the virtual address space is larger than the physical memory
- In the actual hardware, a **Present/absent bit** keeps track of which pages are physically present in memory
- If the program references an unmapped address, the MMU causes the CPU to trap to the OS
- Such a trap is called a **page fault**
- The OS swaps the page that is needed with a little-used frame, changes the map and restarts the instruction

Page Tables (1)

- The purpose of the page table is to map virtual pages onto page frames
- For example, with a 16-bit address and a 4-KB page size, the upper 4 bits could specify one of the 16 virtual pages and the lower 12 bits would then specify the byte offset (0 to 4095) within the selected page

Page Tables (2)

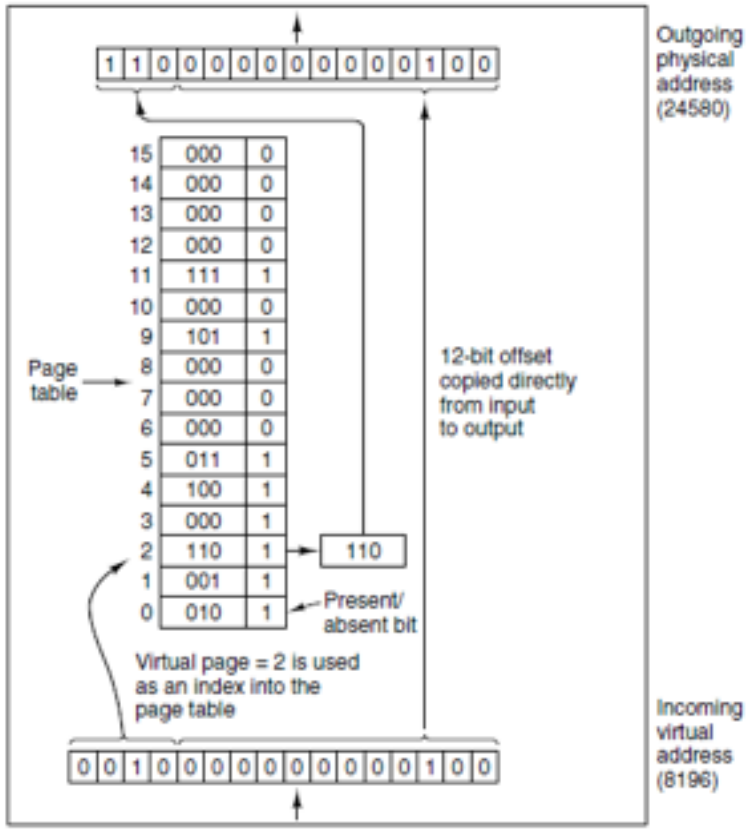

Figure 3-10. The internal operation of the MMU with 16 4-KB pages

Page Tables (3)

- **Structure of a Page Table Entry:**
  - The exact layout of an entry in the page table is highly machine dependent, but the information is roughly the same (Fig. 3-11):
    - Page frame number
    - Present/absent bit
    - Protection bits (r/w — one bit or r/w/x — three bits)
    - Modified bit (dirty bit)
    - Referenced bit
    - Caching disabled bit

Page Tables (4)

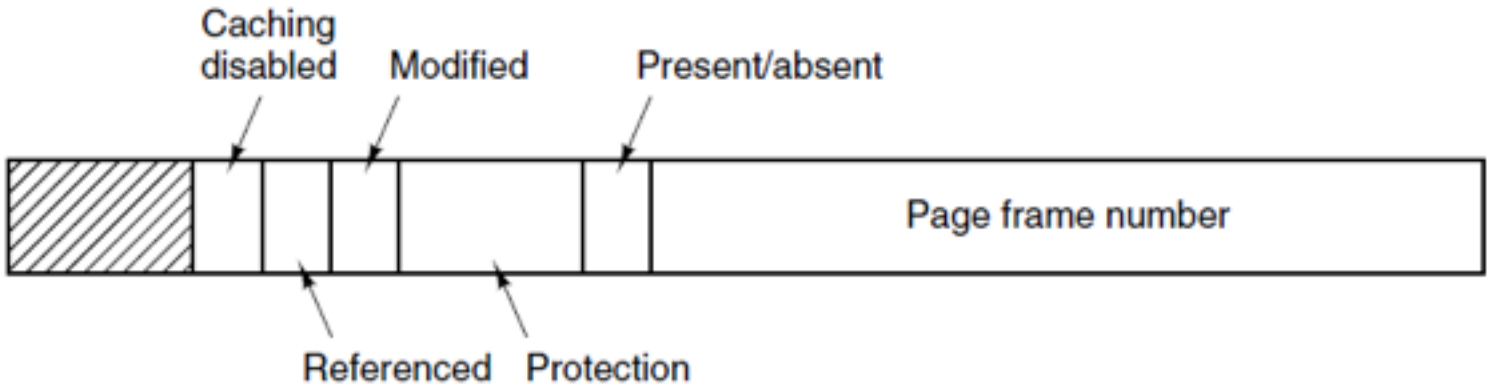

Figure 3-11. A typical page table entry.

Speeding Up Paging

- Major issues faced:
  - The mapping from virtual address to physical address must be fast: the virtual-to-physical mapping must be done on every memory reference
  - If the virtual address space is large, the page table will be large: a 32-bit address space has 1 million pages, the page table must have 1 million entries and each process requires its own page table

Translation Lookaside Buffers (1)

- In the absence of paging, a 1-byte instruction makes only one memory reference, to fetch the instruction
- With paging, at least one additional memory reference will be needed, to access the page table which would reduces performance by half

Translation Lookaside Buffers (2)

- The solution is to equip computers with a small hardware device for mapping virtual addresses to physical addresses without going through the page table
- It is based on the observation that most programs tend to make a large number of references to a small number of pages, and not the other way around
- The device is called a **TLB** (**Translation Lookaside Buffer**) or an **associative memory** (Fig. 3-12)

Translation Lookaside Buffers (3)

Valid	Virtual page	Modified	Protection	Page frame
1	140	1	RW	31
1	20	0	R X	38
1	130	1	RW	29
1	129	1	RW	62
1	19	0	R X	50
1	21	0	R X	45
1	860	1	RW	14
1	861	1	RW	75

Figure 3-12. A TLB structure

Translation Lookaside Buffers (4)

- hen a virtual address is presented to the MMU for translation, the hardware first checks to see if its virtual page number is present in the TLB by comparing it to all the entries in parallel.
- If a valid match is found and the access does not violate the protection bits, the page frame is taken directly from the TLB (a protection fault is generated otherwise)

Translation Lookaside Buffers (5)

- When the virtual page number is not in the TLB, the MMU does an ordinary page table lookup
- It removes one of the entries from the TLB and replaces it with the page table entry just looked up
- When an entry is purged from the TLB, only the **modified bit** is copied back into the page table entry in memory since the other values are already there except the reference bit
- When the TLB is loaded from the page table, all the fields are taken from memory

Software TLB Management (1)

- If the TLB is moderately large (say, 64 entries) to reduce the miss rate, software management of the TLB turns out to be acceptably efficient
- The main gain is a much simpler MMU, which frees up a considerable amount of area on the CPU chip for caches and other features that can improve performance

Software TLB Management (2)

- When software TLB management is used, it is essential to understand the difference between different kinds of misses:
  - A **soft miss** occurs when the page referenced is not in the TLB, but is in memory
  - A hard miss occurs when the page itself is neither in TLB nor in memory. A **page table walk** (a process of looking up the mapping in the page table hierarchy) is needed

Software TLB Management (3)

- There are three types of situations that might occur when the page walk does not find the page in the process' page table:
  - A **minor page fault** occurs when the page is actually in memory, but not in this process' page table (it may have been brought in from disk by another process)
  - A **major page fault** occurs if the page needs to be brought in from disk
  - A **segmentation fault** occurs when the program simply accesses an invalid address and no mapping needs to be added in the TLB

Multilevel Page Tables (1)

- The purpose of the multilevel page table method is to avoid keeping all the page tables in memory all the time
- Each top-level page table entry contains PT1 field, PT2 field and offset (Fig. 3-13a)
- When a virtual address is presented to the MMU, it first extracts the PT1 field and uses this value as an index into the top-level page table (Fig. 3-13b)

Multilevel Page Tables (2)

- The entry located by indexing into the top-level page table yields the address or the page frame number of a second-level page table
- The PT2 field is now used as an index into the selected second-level page table to find the page frame number for the page itself
- After the frame page is found accessing the necessary memory address is pretty straightforward by using the offset

Multilevel Page Tables (3)

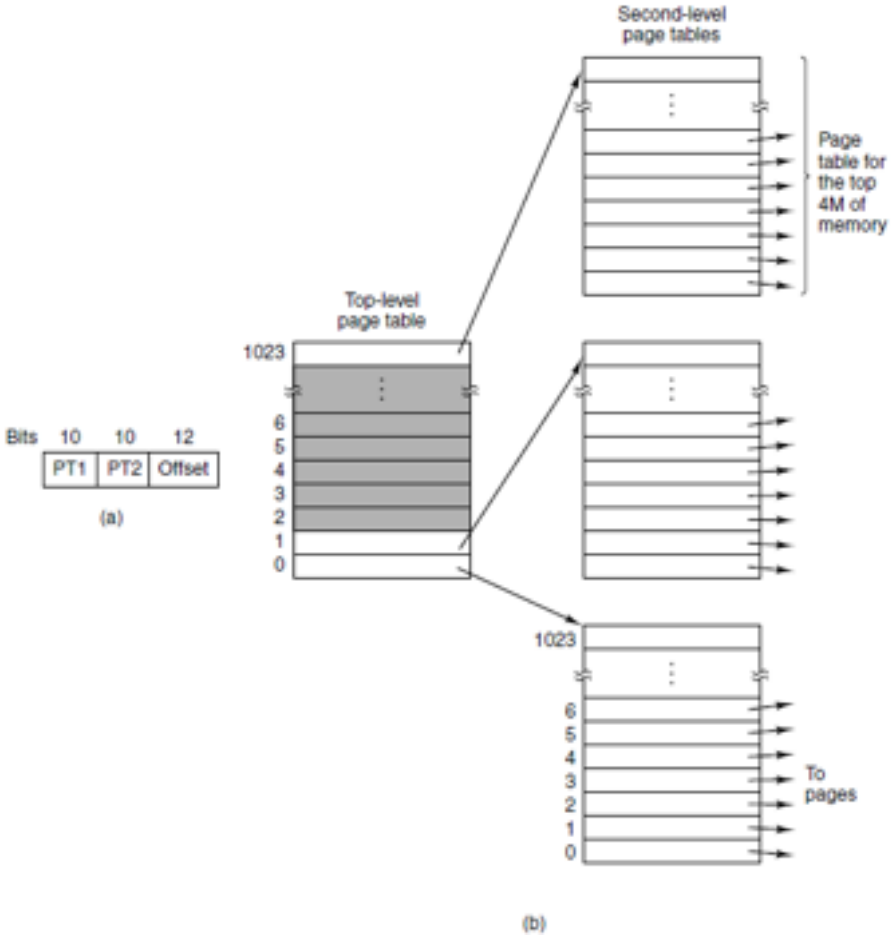

Figure 3-13. (a) A 32-bit address with two page table fields. (b) Two-level page tables.

Inverted Page Tables (1)

- An alternative to a paging hierarchy is known as **inverted page tables**
- There is one entry per page frame in real memory, rather than one entry per page of virtual address space, so the size is proportional to physical memory, not the virtual address space
- The entry contains a pair (process ID, virtual page address) and refers to a page frame

Inverted Page Tables (2)

- Such an approach helps saving space, but has a serious pitfall:
  - when process *n* references virtual page p, it is no longer possible to find the physical page by using *p* as an index into the page table. Instead, we must search the entire inverted page table for an entry (n, p). This search must be done on every memory reference, not just on page faults

Inverted Page Tables (3)

- To solve this problem we can use the TLB. If the TLB can hold all of the heavily used pages, translation is as fast as with regular page tables
- On a TLB miss the inverted page table has to be searched in software
- We can use hash tables to make the search faster (Fig. 3-14)

Inverted Page Tables (4)

- Now the algorithm to process the misses looks like this:
  - We organize the inverted page table as a hash table
  - We use hash function for given (PID, page number)
  - If we have a match, translate the address. If not, use collision resolution technique (rehash, search, linear probing) and search again

Inverted Page Tables (5)

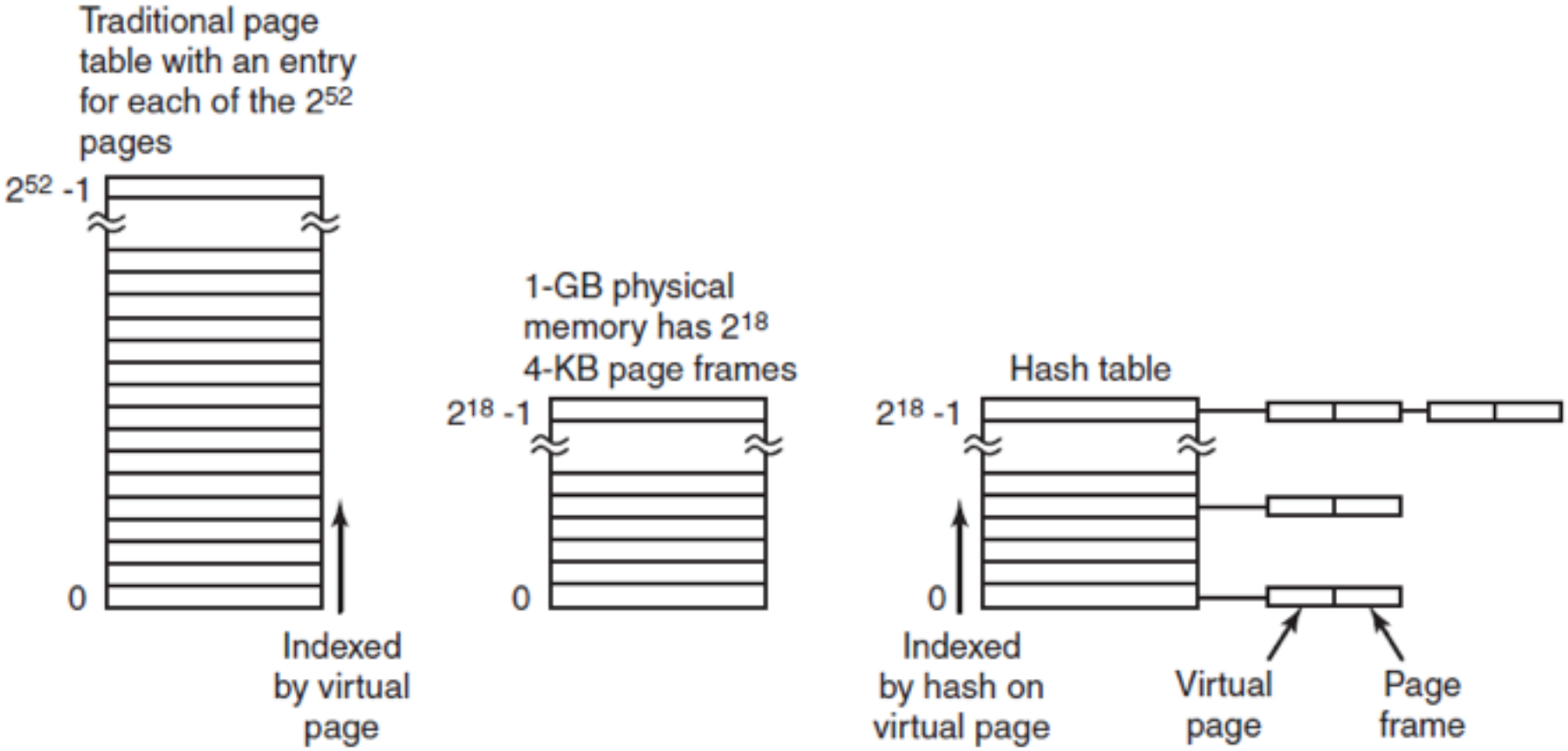

Figure 3-14. Comparison of a traditional page table with an inverted page table.

End

Week 06 – Lecture 2

References

- Tanenbaum & Bo, Modern Operating Systems: 4th edition, 2013 Prentice-Hall, Inc.
- http://www.cs.cmu.edu/~gkesden/412-18/fall01/ln/lecture11.html