

# Memory Management

Kartik Gopalan

## References:

- Chapter 3, Modern Operating Systems, Andrew S. Tanenbaum
- [https://en.wikipedia.org/wiki/Page\\_\(computer\\_memory\)](https://en.wikipedia.org/wiki/Page_(computer_memory))
- [https://en.wikipedia.org/wiki/Page\\_table](https://en.wikipedia.org/wiki/Page_table)
- [https://en.wikipedia.org/wiki/Virtual\\_memory](https://en.wikipedia.org/wiki/Virtual_memory)

# Memory Management

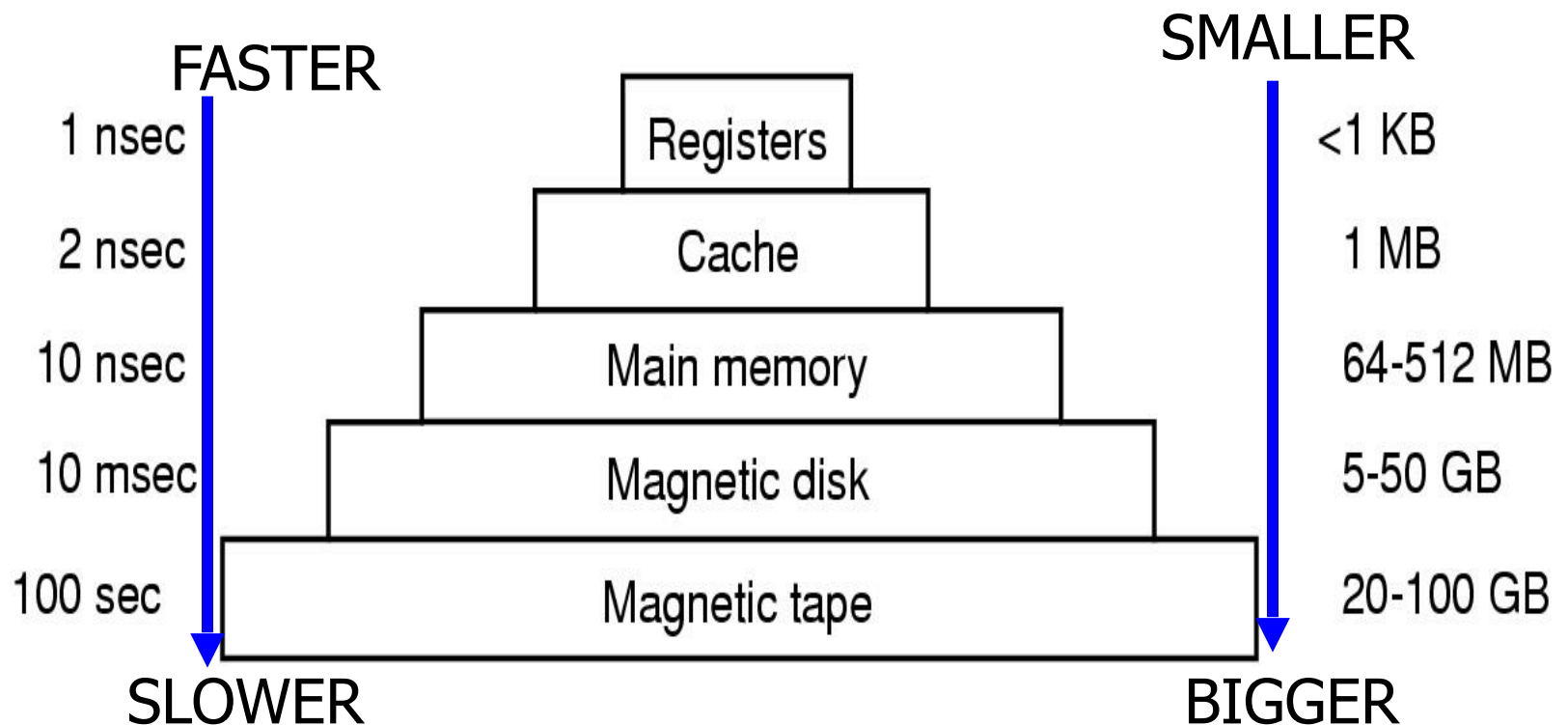
- Ideally programmers want memory that is
  - large
  - fast
  - persistent (non-volatile)

# Memory Hierarchy

- Registers & Cache
  - small amount of fast, expensive, volatile memory
- Main memory
  - some medium-speed, medium price, volatile/persistent memory
- Disk & Tape
  - Lots of slow, cheap, persistent, storage

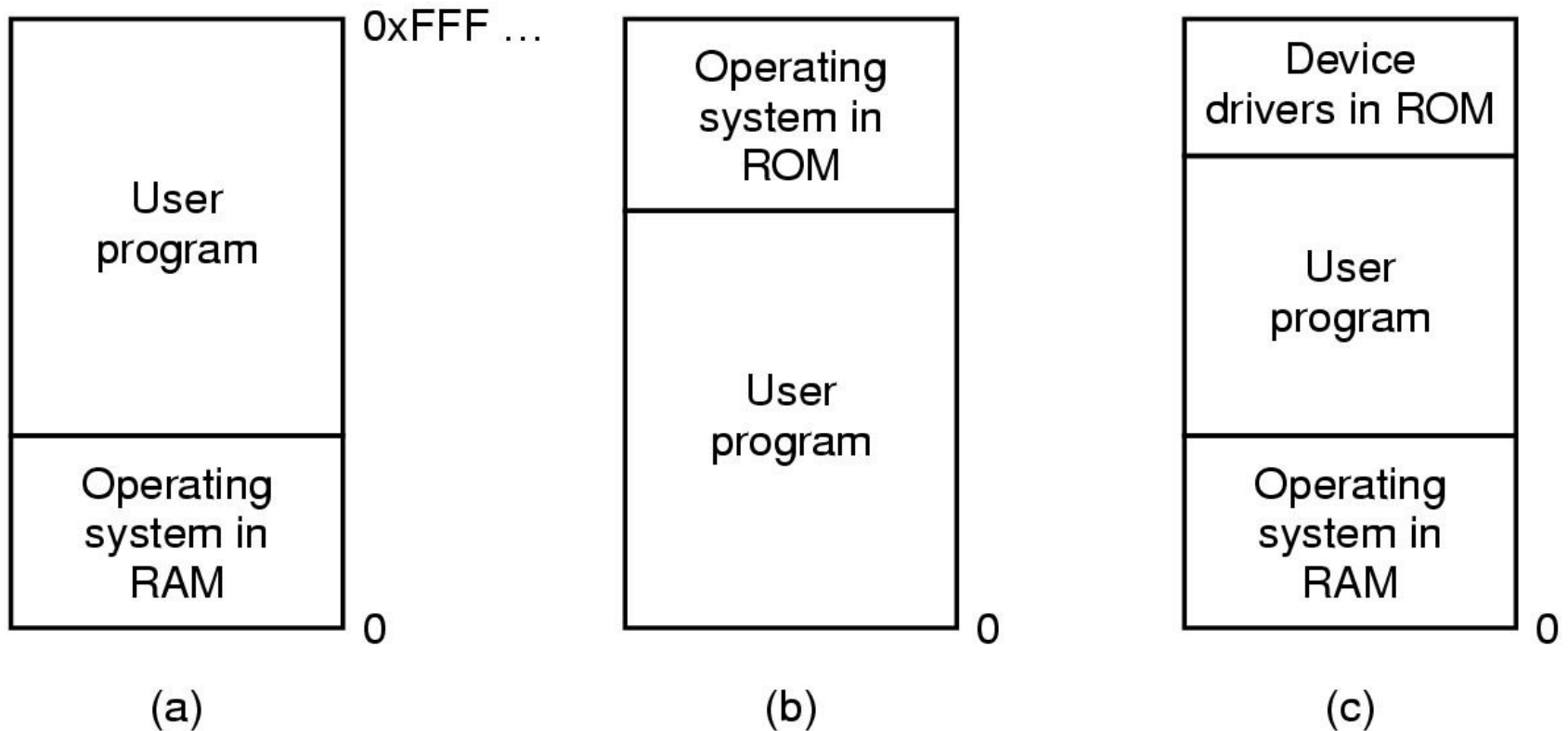
Typical access time

Typical capacity



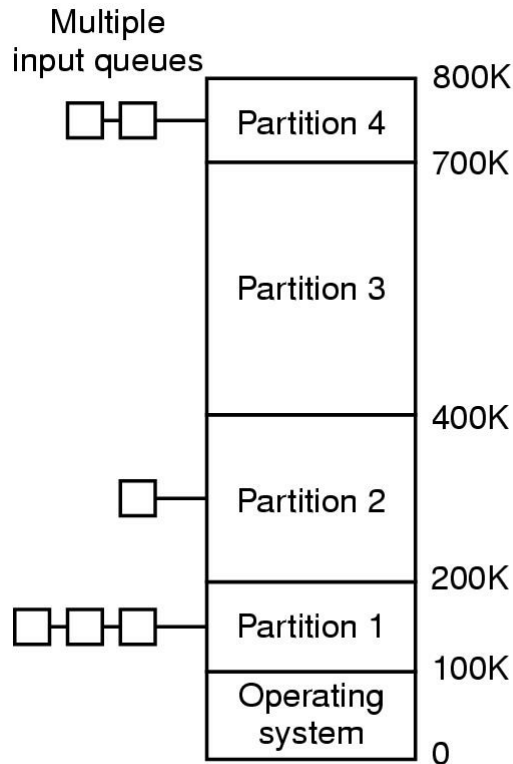
# Basic Memory Management

"Mono-programming" without Swapping or Paging

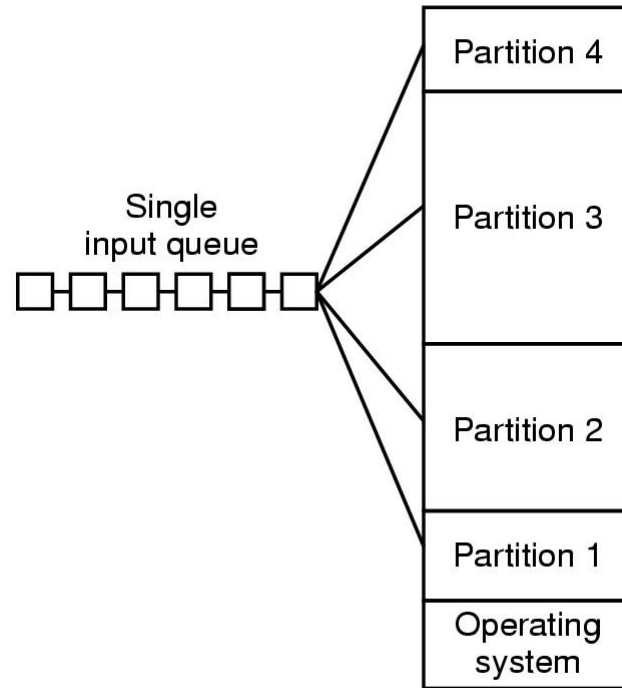


Three simple ways of organizing memory  
- an operating system with one user process

# Multiprogramming with Fixed Partitions



(a)



(b)

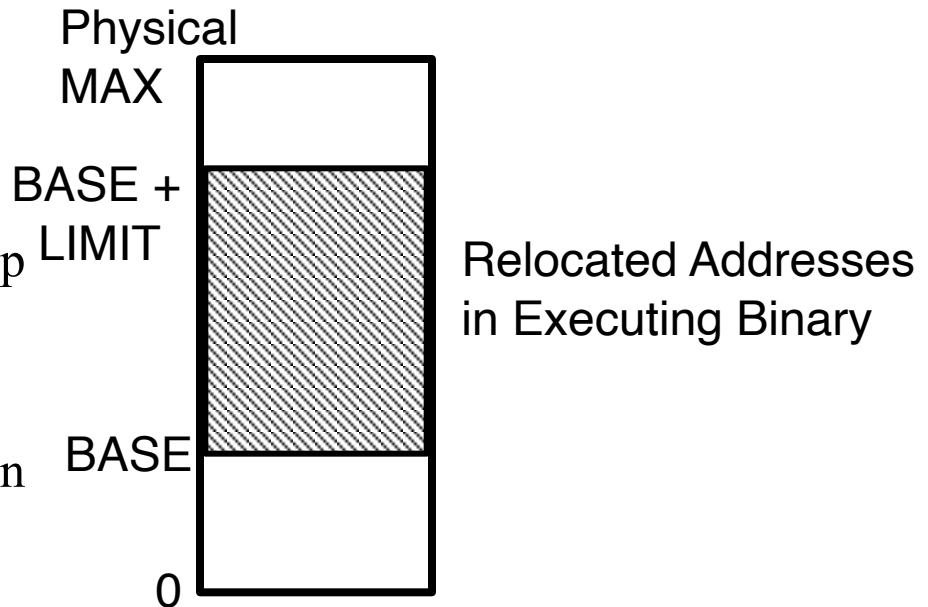
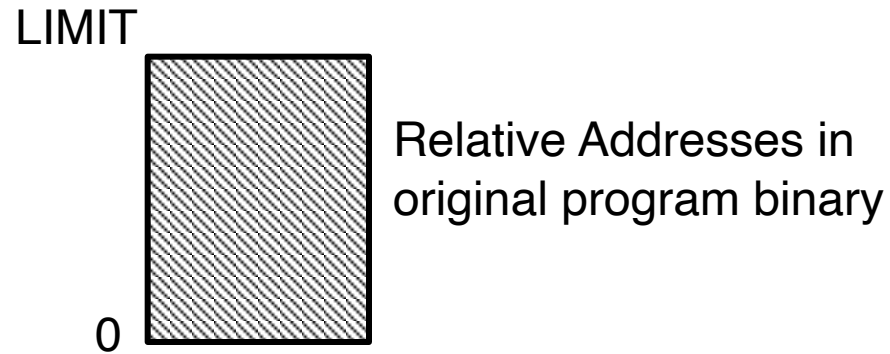
- Fixed memory partitions
  - (a) separate input queues of processes for each partition
  - (b) single input queue

# Physical Memory addressing

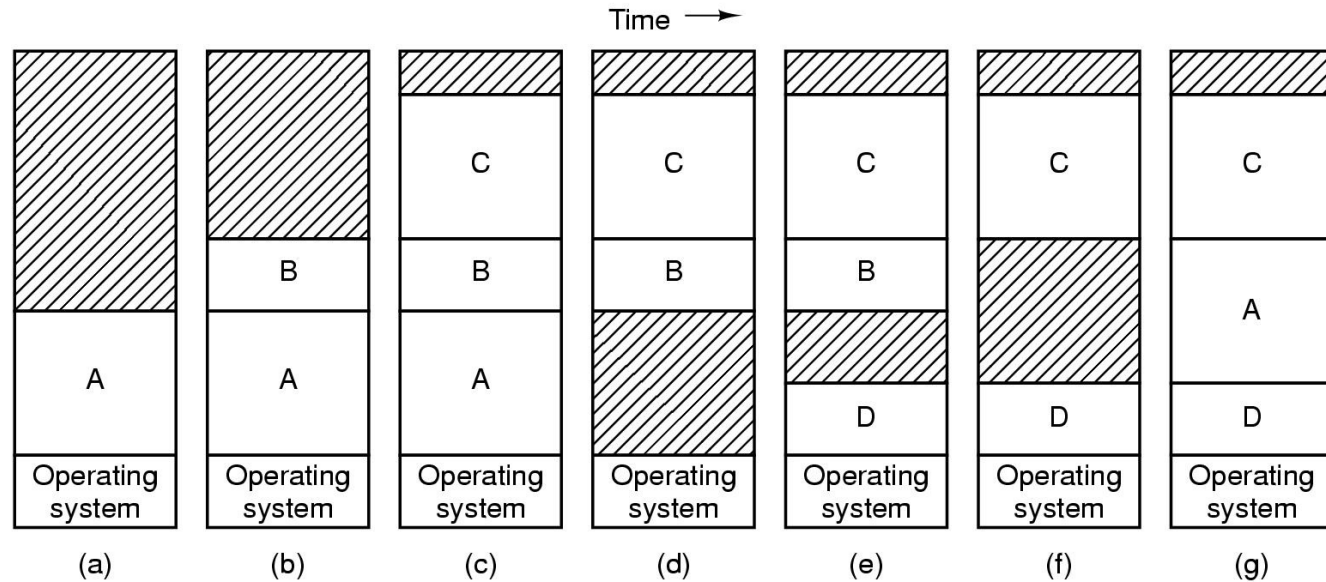
- Consider an instruction that reads from a memory location
  - `load reg memory_address`
- But programmer doesn't know the `memory_address` where data will be stored when the process runs!
- Solution: Relocation
  - Programmer assumes a “relative” address, which is converted to a “physical” address by the OS+hardware when the process runs.

# Relocation and Protection

- Problem: A programmer doesn't know where a program will be loaded in memory
  - address locations of variables and code routines cannot be absolute
  - must keep a program out of other processes' partitions
- Solution: Use base and limit values
- Relocation
  - Address locations in a program are relative.
  - They are added to a **base value** to map to physical addresses.
- Protection
  - Access to address locations larger than **limit value** results in an error



# What if physical memory is not enough to hold all processes? — Swapping



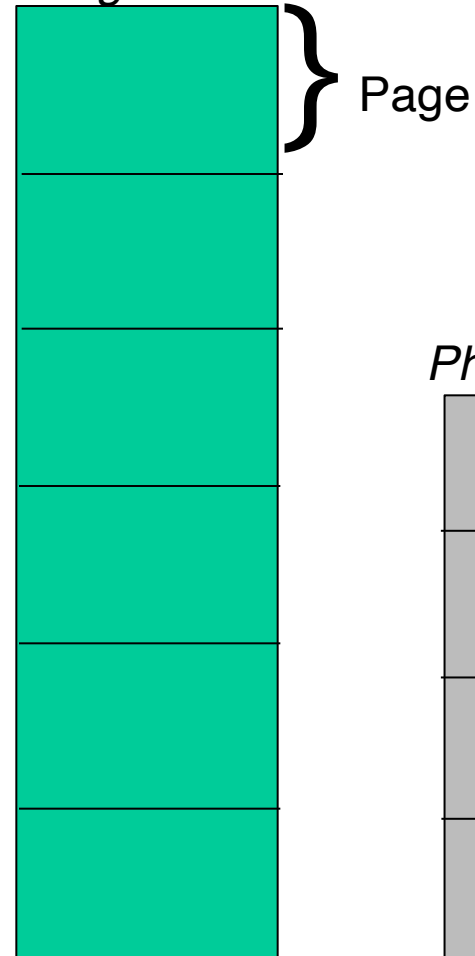
- Physical memory may not be enough to accommodate the needs of all processes
- Memory allocation changes as
  - processes come into memory
  - leave memory and are *swapped out* to disk
  - Re-enter memory by getting *swapped-in* from disk
- Shaded regions are unused memory



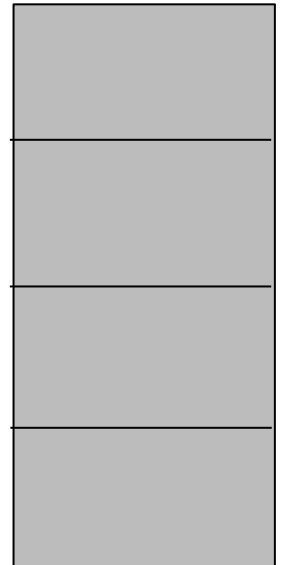
# Virtual Memory

- Swapping the memory of an entire process is useful when the sum of memory needed by all processes is greater than the total RAM available in the system.
- But sometimes, a single process might require more memory than the total RAM in the system.
- In such cases swapping an entire process is not enough.
- Rather, we need to break up the memory space of a process into smaller equal-sized pieces, called PAGES.
- OS then decides which pages stay in memory and which get moved to disk.
- **Virtual memory**: means that each process gets an illusion that it has more memory than the physical RAM in the system.

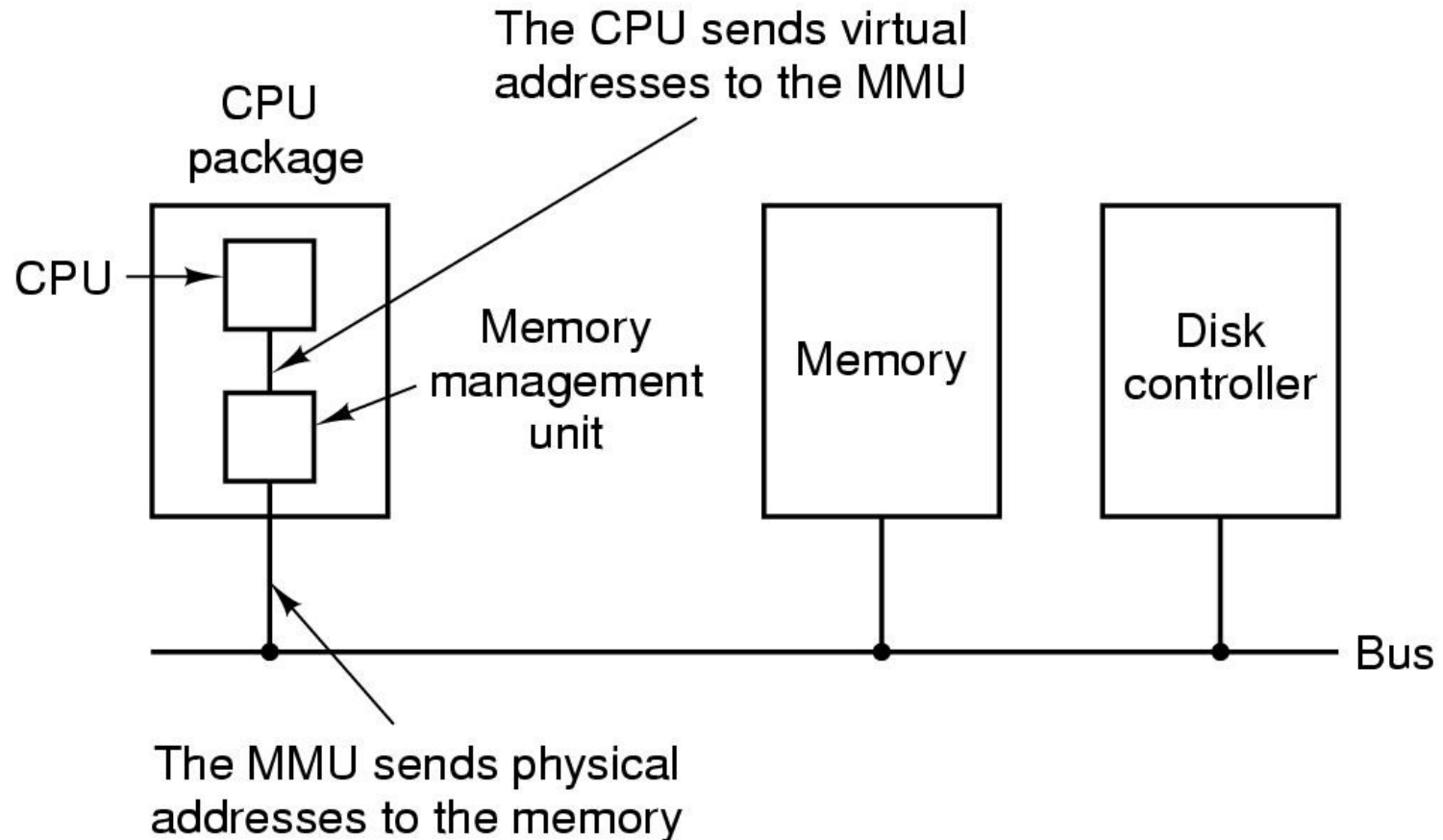
*Virtual Address Space  
of a single Process*



*Entire  
Physical RAM*



# Memory Management Unit (MMU)



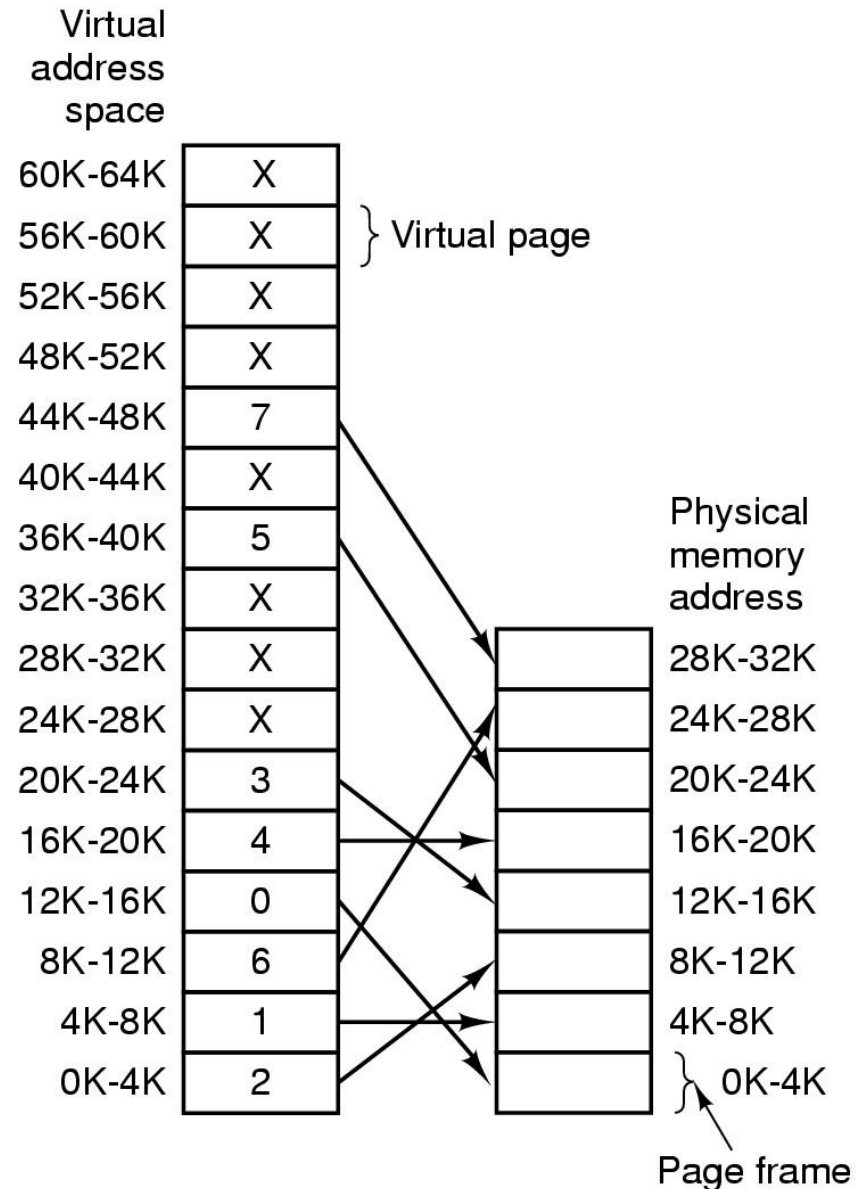
- MMU is a hardware module that accompanies the CPU
- It translates the Virtual Address used by executing instructions to Physical Addresses in the main memory.

# Size of address space (in bytes) as a function of address size (in bits)

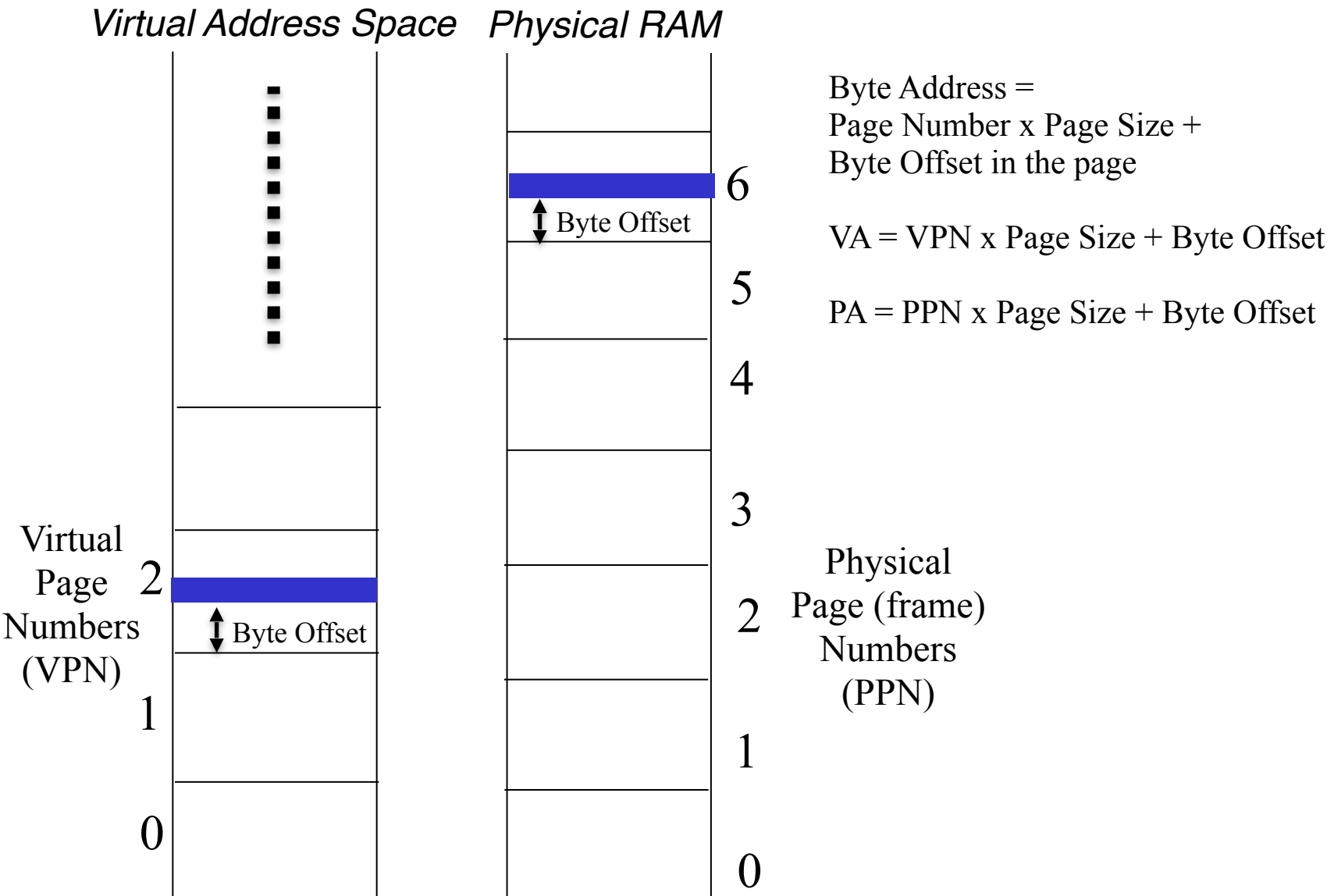
Number of bits in address	Maximum address space size (bytes)
0	$2^0 = 1$ byte
1	$2^1 = 2$ bytes
2	$2^2 = 4$ bytes
10	$2^{10} = 1024 = 1\text{KiB}$
12	$2^{12} = 4\text{KiB}$
16	$2^{16} = 64\text{ KiB}$
32	$2^{32} = 4\text{GiB (Gibibytes)}$
64	$2^{64} = 16\text{ EiB (Exbibytes)}$

# Page Table

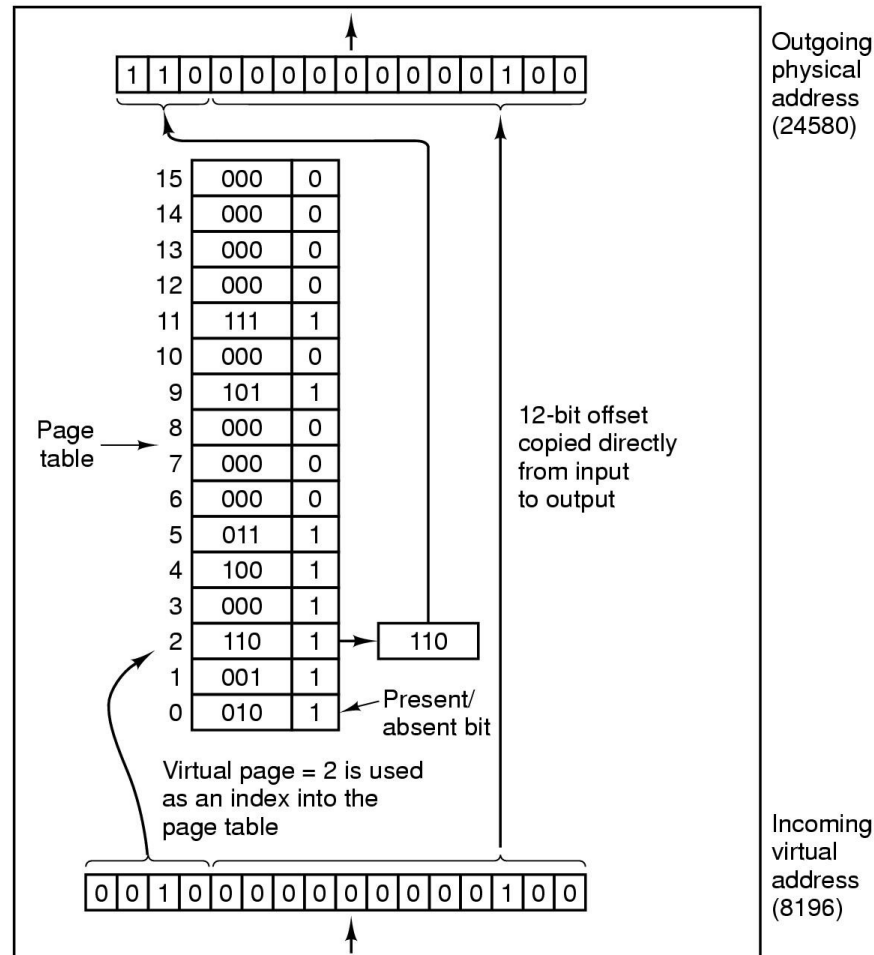
- An array that stores the mapping from virtual page numbers to physical numbers
- The OS maintains
  - One page table per userspace process.
  - And usually another page table for kernel memory.



# Translating Virtual address (VA) to physical address (PA)

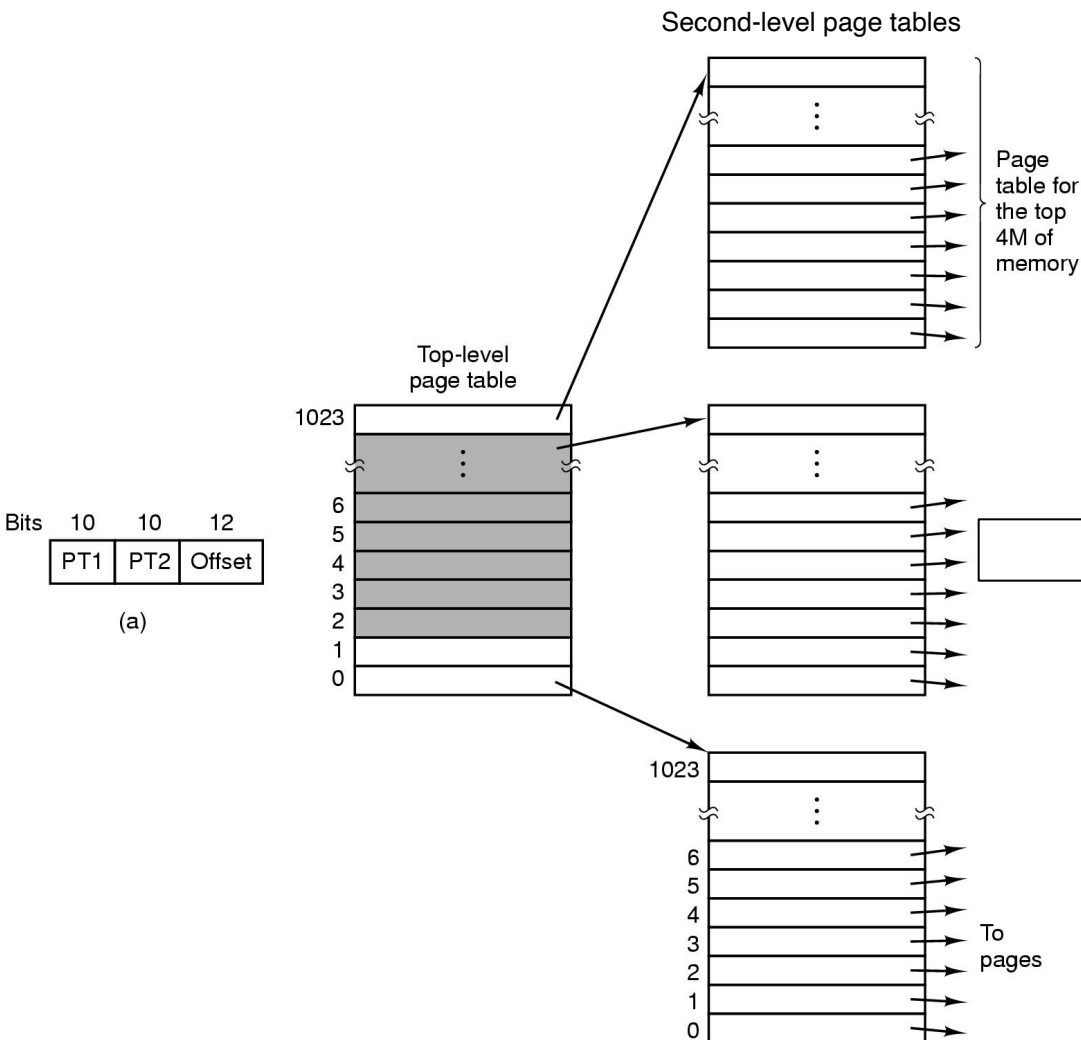


# Virtual Address Translation For Small Address Space



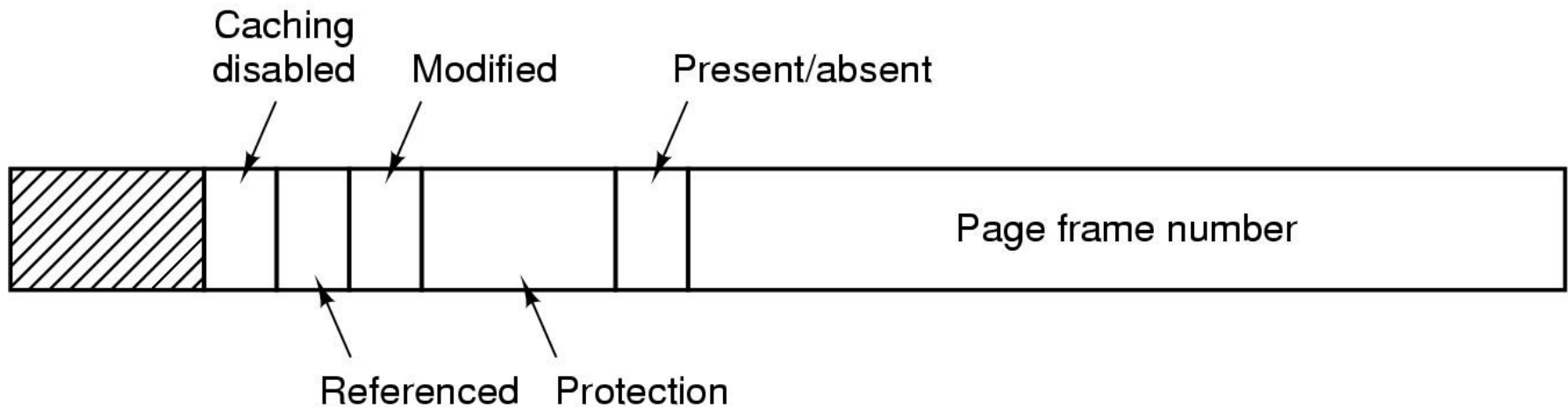
Internal operation of MMU with 16 4 KB pages

# Virtual Address Translation For Large Address Space



- 32 bit address with 2 page table fields
- Two-level page tables
- PT too Big for MMU
  - Keep it in main memory
- But how does MMU know where to find PT?
  - Registers (CR2 on Intel)

# Typical Page Table Entry (PTE)



- Page Frame number = physical page number for the virtual page represented by the PTE
- Referenced bit: Whether the page was accessed since last time the bit was reset.
- Modified bit: Also called “Dirty” bit. Whether the page was written to, since the last time the bit was reset.
- Protection bits: Whether the page is readable? writeable? executable? contains higher privilege code/data?
- Present/Absent bit: Whether the PTE contains a valid page frame number. Used for marking swapped/unallocated pages.



# TLBs – Translation Lookaside Buffers

<b>Valid</b>	<b>Virtual page</b>	<b>Modified</b>	<b>Protection</b>	<b>Page frame</b>
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

- TLB is a small cache that speeds up the translation of virtual addresses to physical addresses.
- TLB is part of the MMU hardware (comes with CPU)
- It is not a Data Cache or Instruction Cache. Those are separate.
- TLB simply caches translations from virtual page number to physical page number so that the MMU don't have to access page-table in memory too often.
- On older x86 processors, TLB had to be “flushed” upon every context switch because there is no field in TLB to identify the process context.
  - Tagged TLB can reduce this overhead

# Impact of Page Size on Page tables

## Small page size

- Advantages
  - less internal fragmentation
  - page-in/page-out less expensive
- Disadvantages
  - process that needs more pages has larger page table
  - Smaller “TLB Coverage” (next slide)

# Bit distribution in a memory address

- For a 32 bit address and ...
  - 4KiB page
    - 12 bit offset and 20 bit page number
  - 8KiB page
    - 13 bit offset and 19 bit page number
  - 64KiB page
    - 16 bit offset and 16 bit page number

# Quiz

- Consider a machine that has a 32-bit virtual address space and 8KiByte page size.
  1. What is the total size (in bytes) of the virtual address space for each process?
  2. How many bits in a 32-bit address are needed to determine the page number of the address?
  3. How many bits in a 32-bit address represent the byte offset into a page?
  4. How many page-table entries are present in the page table?

# Quiz Answers

- Consider a machine that has a 32-bit virtual address space and 8KiByte page size.
1. Total size (in bytes) of the virtual address space for each process =  $2^{32}$   
=  $4 * 1024 * 1024 * 1024$  bytes = 4 GiB
  2. Number of pages in virtual address space =  $4\text{GiB}/8\text{KiB} = 512 * 1024 = 2^9 * 2^{10} = 2^{19}$ 
    - So the number of bits in a 32-bit address are needed to determine the page number of the address =  $\log_2(4\text{GiB}/8\text{KiB}) = \log_2(2^{19}) = 19$  bits
  3. How many bits in a 32-bit address represent the byte offset into a page?
    - $\log_2(8\text{KiB}) = \log_2(2^{13}) = 13$  bits
    - Also,  $32 - 19 = 13$  bits
  4. How many page-table entries are present in the page table?
    - Number of PTEs = Number of pages in virtual address =  $4\text{GiB}/8\text{KiB} = 2^{19}$  pages

# References

- **Chapter 3: Modern Operating Systems, Andrew S. Tanenbaum**
- **X86 architecture**  
<http://en.wikipedia.org/wiki/X86>
- **Memory segment**  
[http://en.wikipedia.org/wiki/Memory\\_segment](http://en.wikipedia.org/wiki/Memory_segment)
- **Memory model**  
[http://en.wikipedia.org/wiki/Memory\\_model](http://en.wikipedia.org/wiki/Memory_model)
- **IA-32 Intel Architecture Software Developer's Manual, Volume 1: Basic Architecture**