COMP4611: Design and Analysis of Computer Architectures

Memory System

Virtual Memory

Lin Gu

CSE, HKUST

Why Virtual Memory?

- An example: MIPS64 is a 64-bit architecture allowing an address space defined by 64 bits
  - Maximum address space:
    - $2^{64} = 16 \times 10^{18}$ =16,000 petabytes
    - peta = 10^{15}
- This is several orders of magnitude larger than any realistic and economical (or necessary for that matter) physical memory system

Virtual Memory

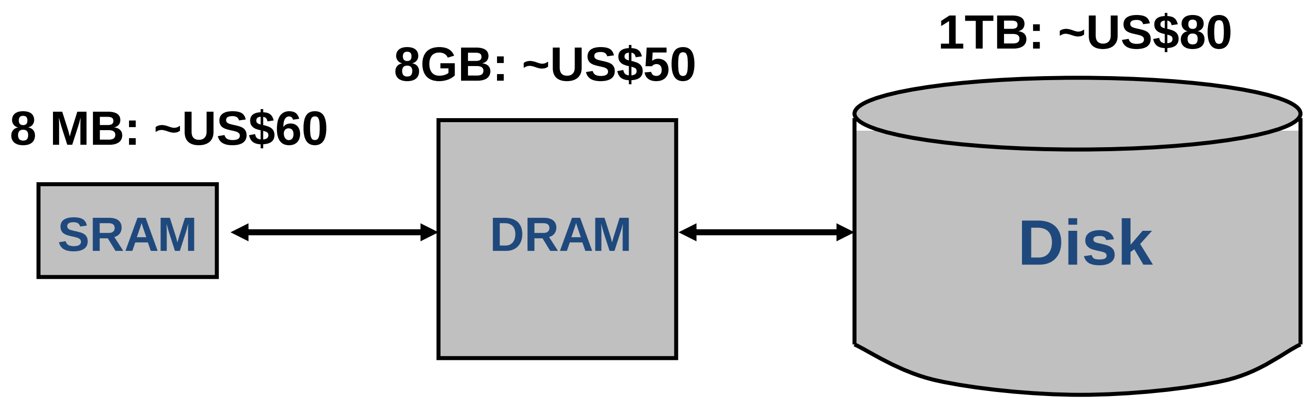
- Originally invented to support program sizes larger than then-available physical memory
  - later on it finds applications in multi-programming and virtual machines
- Virtual memory is as large as the address space allowed by the ISA…but
  - only a portion of the address space resides in physical memory at any given time
  - the rest is kept on disks and brought into physical memory as needed

Motivations for Virtual Memory

- (1) Use Physical DRAM as a Cache for the Disk
  - Address space of a process (program) can exceed physical memory size
  - Only “active” code and data are actually in memory
    - Allocate more memory to process as needed.
  - Sum of address spaces of multiple processes can exceed physical memory
- (2) Simplify Memory Management
  - Multiple processes reside in main memory.
    - Each process with its own address space

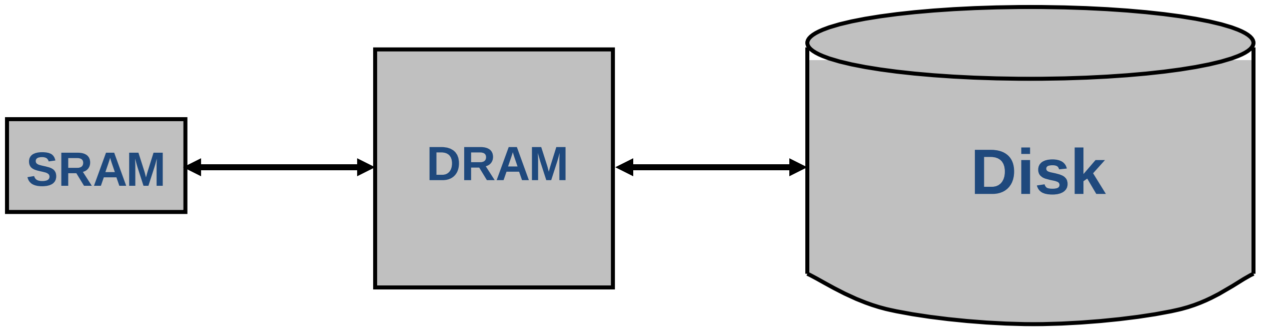
Motivation #1: DRAM as "Cache" for Disk

- Full address space is quite large:
  - 32-bit addresses: 0—4,294,967,295 (~ 4 billion bytes)
  - 64-bit addresses: 0—18,446,744,073,709,551,615 (~ 16,000 petabytes)
- Disk storage is much cheaper than DRAM storage
  - 1TB DRAM: ~ US$6,400
  - 1TB disk: ~ US$80
- To access large amounts of data in a cost-effective manner, the bulk of the data must be stored on disk

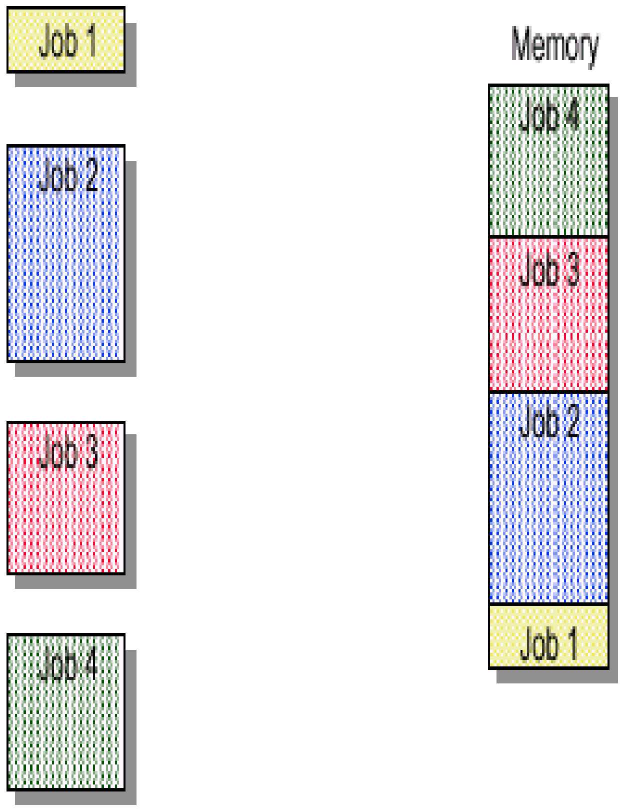


DRAM vs. SRAM as a "Cache"

- DRAM vs. disk is more extreme than SRAM vs. DRAM
  - Access latencies:
    - DRAM ~10X slower than SRAM
    - Disk **~100,000X** slower than DRAM
  - Design decisions made for DRAM caches driven by enormous cost for misses

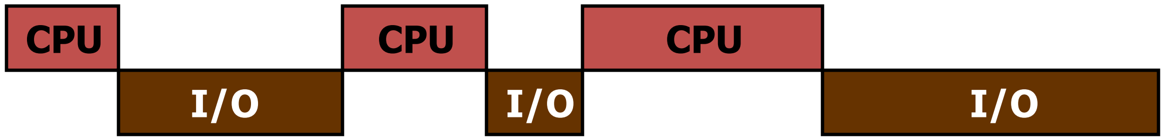


Multi-programming



- Multiple programs run concurrently, ideally each with the illusion that it owns all the machine resources
- Each program requires memory space in the main memory

Multiprogramming and Memory Protection

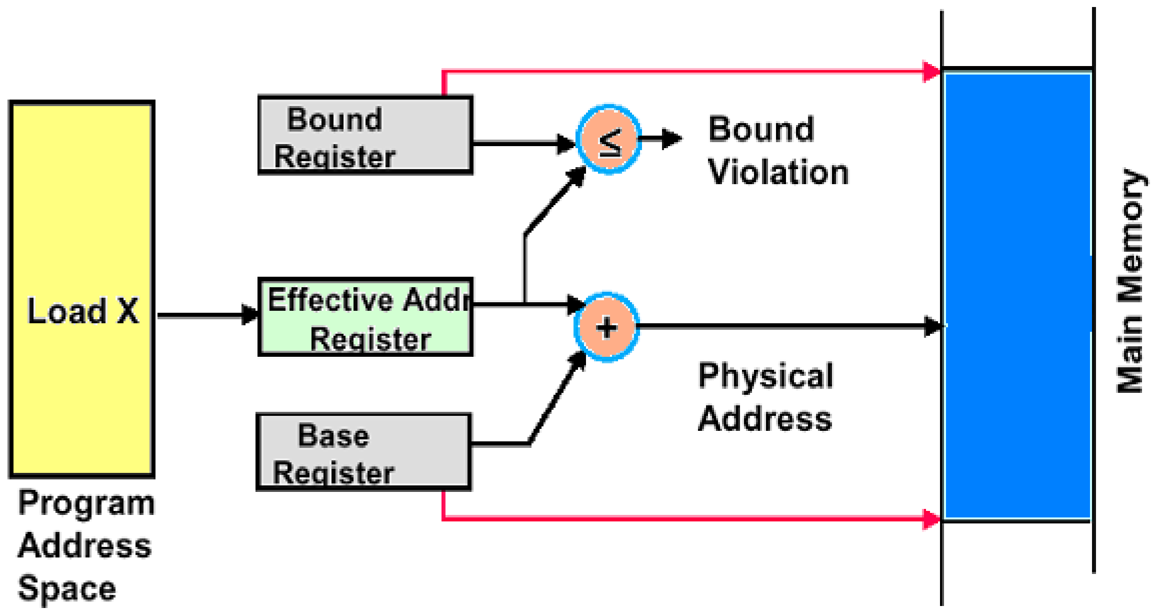


- In the early machines, I/O operations were slow
  - having the CPU wait pending completion of I/O operation was a waste of precious machine cycles
- Higher throughput is achieved if CPU and I/O of 2 or more programs are overlapped
- How to overlap execution of multiple programs?
  - use multiprogramming
- How to protect programs from one another in a multiprogramming environment?
  - use a bound register?

Away With Absolute Addressing

- In early (1950) machines only one program ran at any given time, with unrestricted access to the entire machine (RAM + I/O devices)
- Addresses in a program were location-dependent
  - they depended upon where the program was to be loaded in memory
  - so when programmers wrote code, they had to have a pretty good idea where in memory they should load the program
  - but it was more convenient for programmers to write location-independent programs
- How to achieve location-independence?
  - a base register was used to support re-locatable code

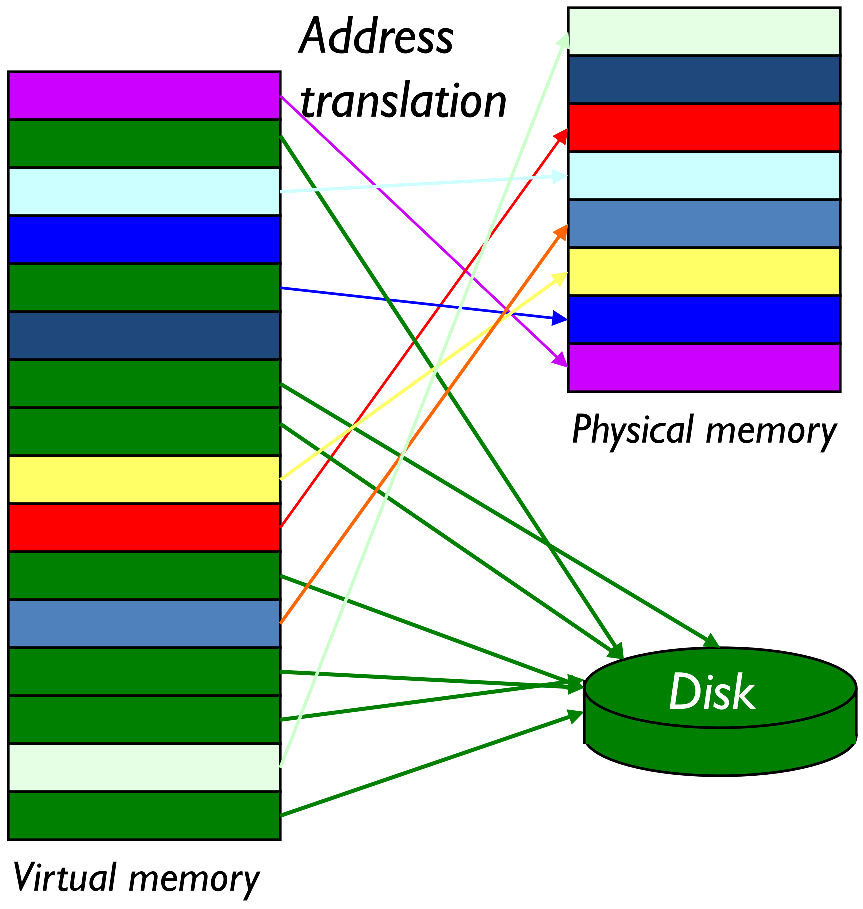
Base & Bound Registers



- Base & bound registers loaded from the process table when the program is context-switched
- Permits multiple memory-resident concurrent programs
- Protects programs from one another

A better solution: virtual memory

Virtual Memory



- Address translation
- Program "sees" entire VM address space
- Program is run in physical memory which is typically smaller than the address space
- Pages of address space are swapped in/out of disk storage as needed
- Strictly speaking VM is required to overcome limitation on the size of physical memory
  - but VM is extended in a natural way to support multiprogramming & memory protection
  - There are machines where the physical memory space is larger than the virtual memory space (e.g., PDP-11)

Advantages of Virtual Memory

- Abstraction of large and flat memory
  - program has a consistent view of a contiguous memory, even though physical memory is scrambled
  - Allows multiprogramming
  - relocation: allows the same program to run in any location in physical memory
- Protection
  - different processes are protected from each other
  - different pages can have different behavior (read-only; user/supervisor)
    - kernel code/data protected from user programs
- Sharing
  - can map same physical memory to multiple processes (shared memory)

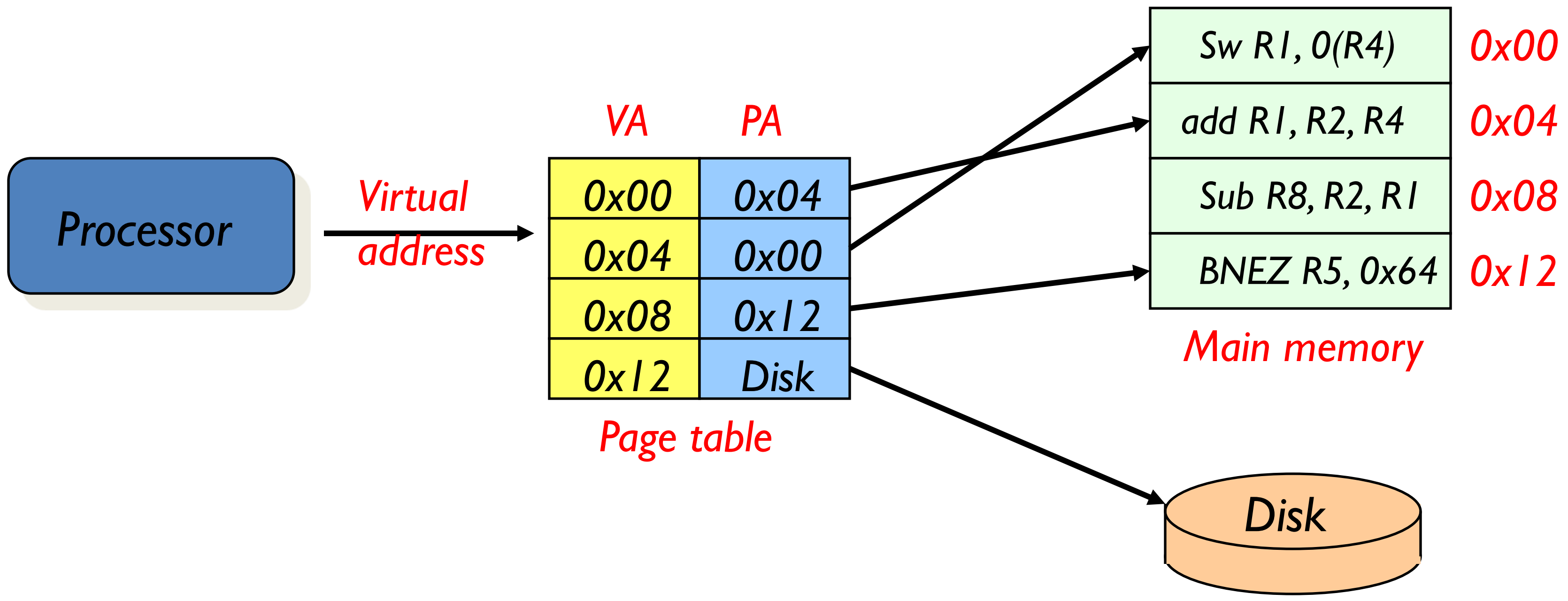
How VM Works

- On program startup
  - OS loads part of the program into RAM; this includes enough code to start execution
  - if program size exceeds allocated RAM space the remainder is maintained on disk
- During execution
  - if program needs a code/data not resident in RAM, it fetches the segment from disk into RAM
  - if there is not enough room in RAM, some resident code/data is evicted from memory to make room
  - if evicted contents are “dirty”, they are written to disk

Address Translation

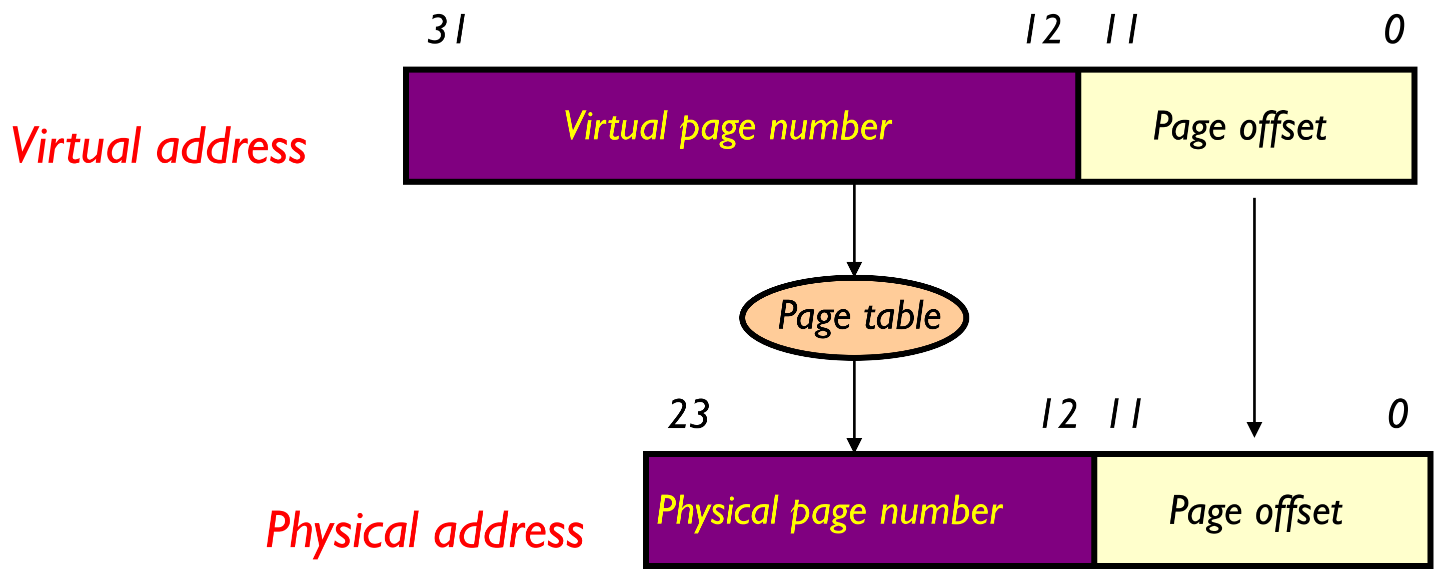
VA-to-PA Address Translation

- Programs use virtual addresses (VA) for data & instructions
- VA translated to physical addresses (PA)
  - Usually organized in pages (paging) or segments
  - Paging systems use a "page table" for the translation
- Instruction/data fetched/updated in memory

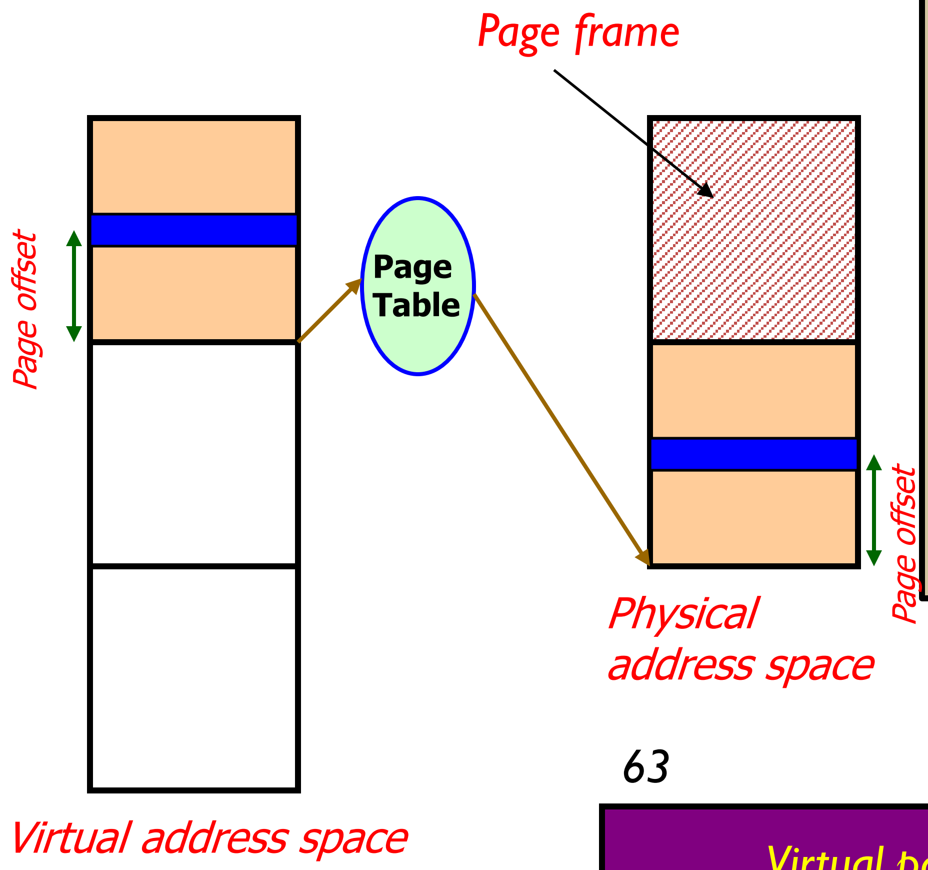


Page Table

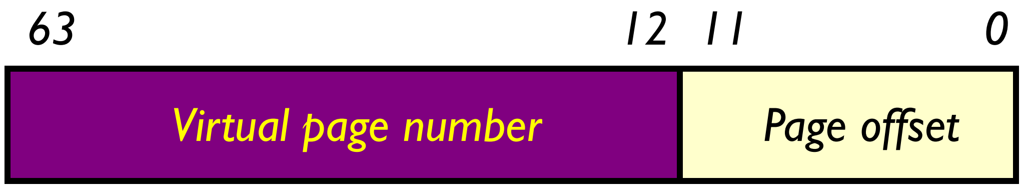
- Memory organized in pages (similar to blocks in cache)
- Page size is usually 4-8 Kbytes



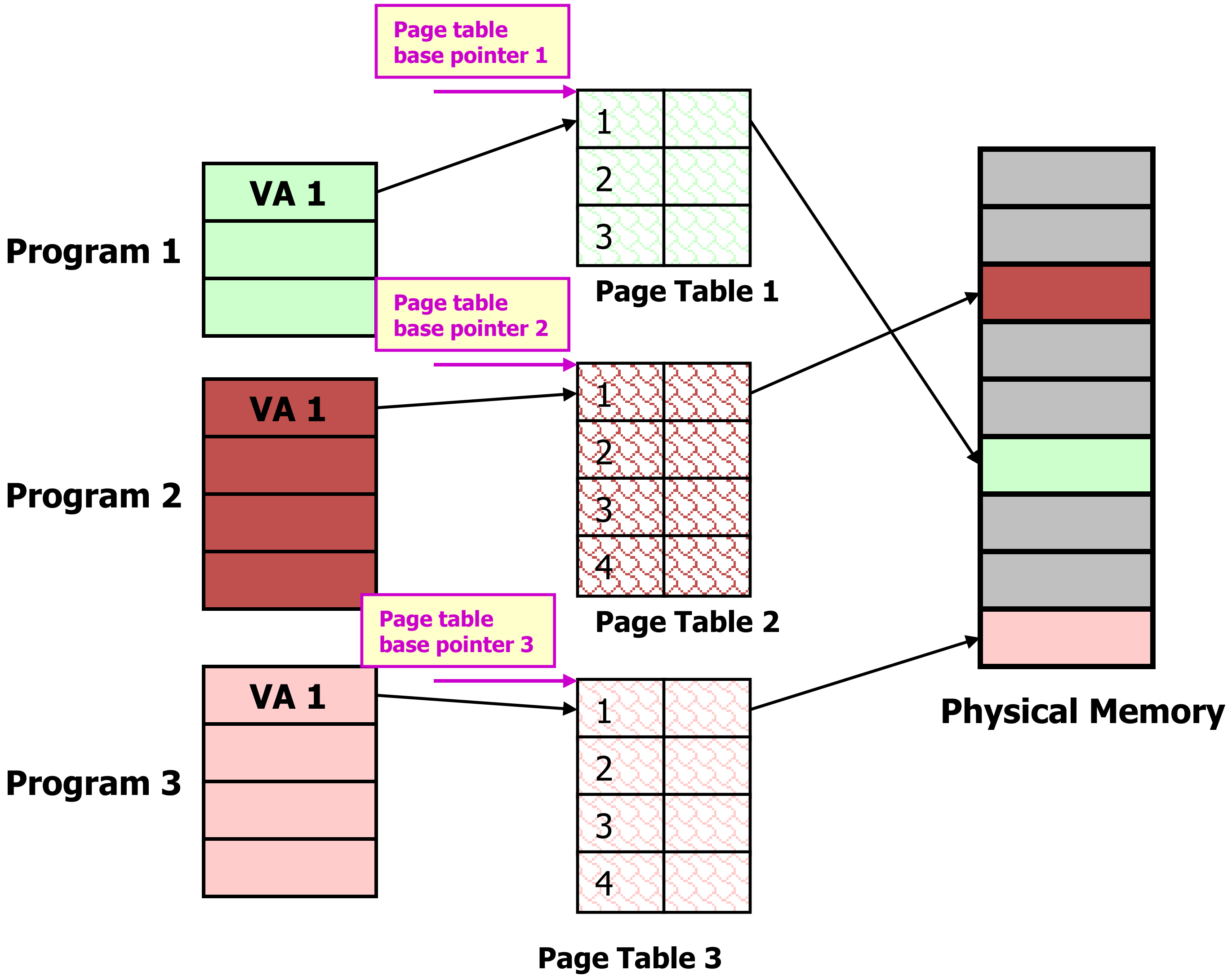
VA to PA Translation



- Page table maps the base address of virtual and physical page frames
- Page offset need not be used in VA=>PA translation because virtual and physical block sizes are the same



Multiprogramming View of VM

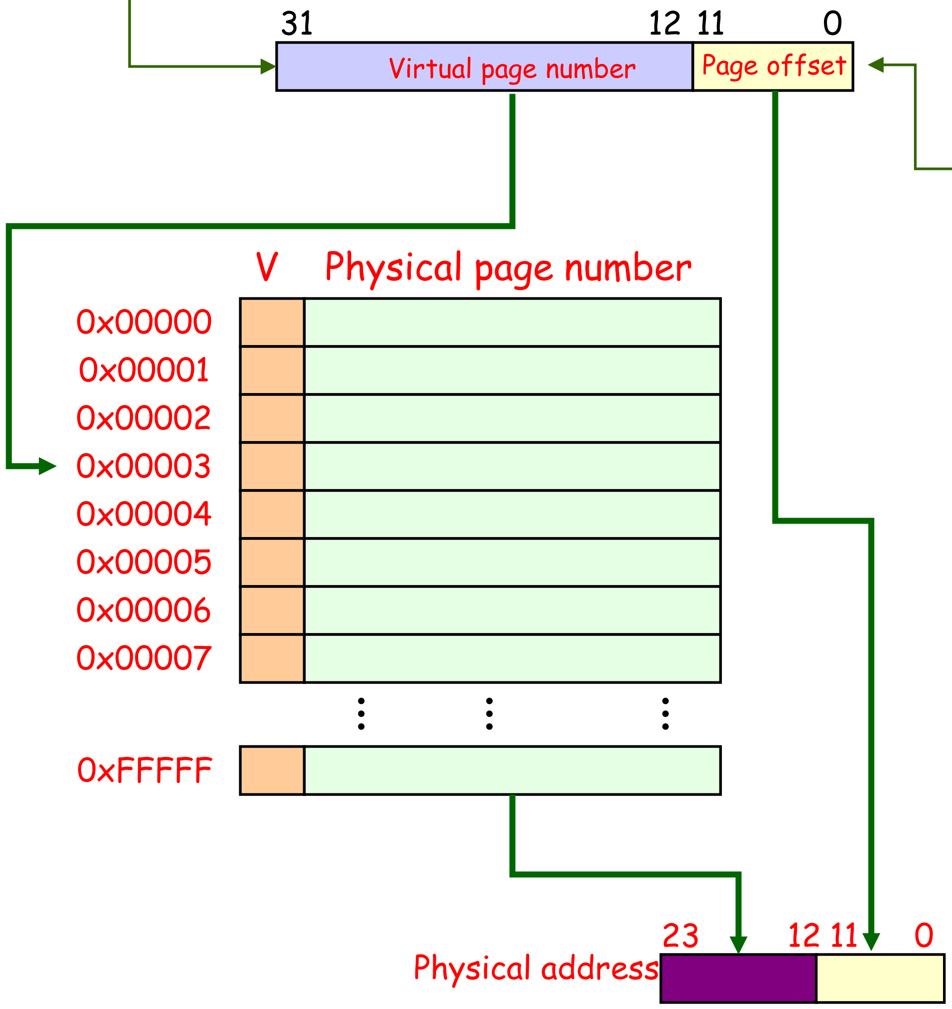


- Each application has a separate page table
- Page table contains an entry for each page
- Page tables are identified by the base pointers maintained by OS for individual processes

Page Table Structure

Virtual page number <=> page table size
20 bits: ~1 million

Page offset <=> page size
12 bits: 4096 bytes



Determining Page Table Size

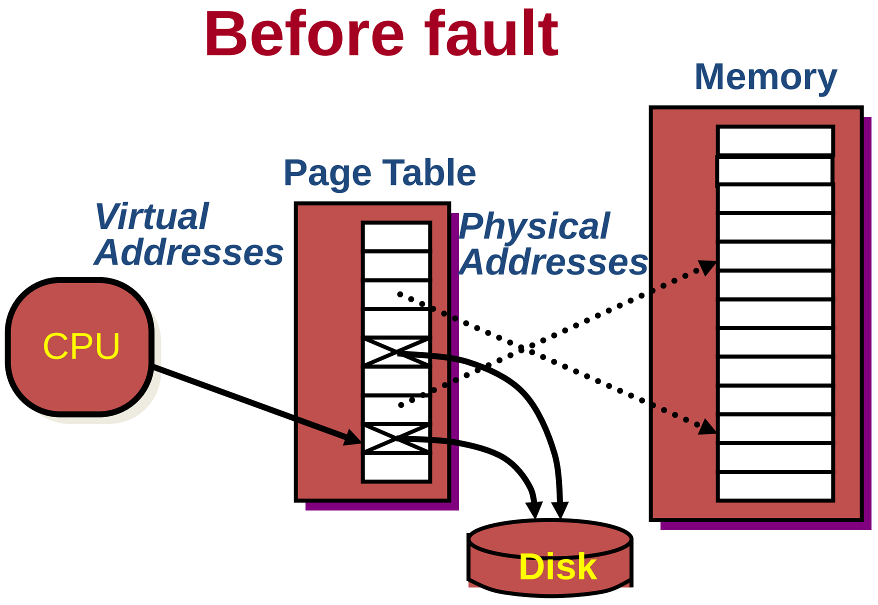
- Assume
  - 32-bit virtual address
  - 30-bit physical address
  - 4 KB pages => 12 bit page offset
  - Each page table entry is one word (4 bytes)
- How large is the page table?
  - Virtual page number = 32 - 12 = 20 bits
  - Number of entries = number of pages = 2^20
  - Total size = number of entries x bytes/entry

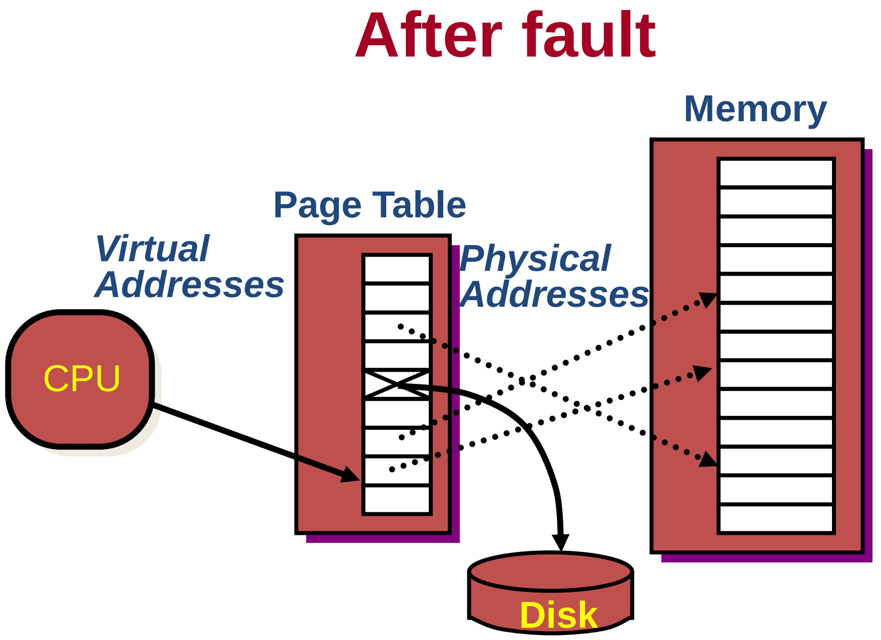
    = 2^20 x 4 = 4 Mbytes
  - Each process running needs its own page table

Page Fault

- How is it known whether the page is in memory?
  - Maintain a valid bit per page table entry
  - valid bit is set to INVALID if the page is not in memory
  - valid bit is set to VALID if the page is in memory
- Page fault occurs when a page is not in memory
  - fault results in OS fetching the page from disk into DRAM
  - if DRAM is full, OS must evict a page (victim) to make room
  - if victim is dirty OS updates the page on disk before fetch
  - OS changes page table to reflect turnover
- After a page fault and page-fetching, execution resumes at the instruction which caused the fault

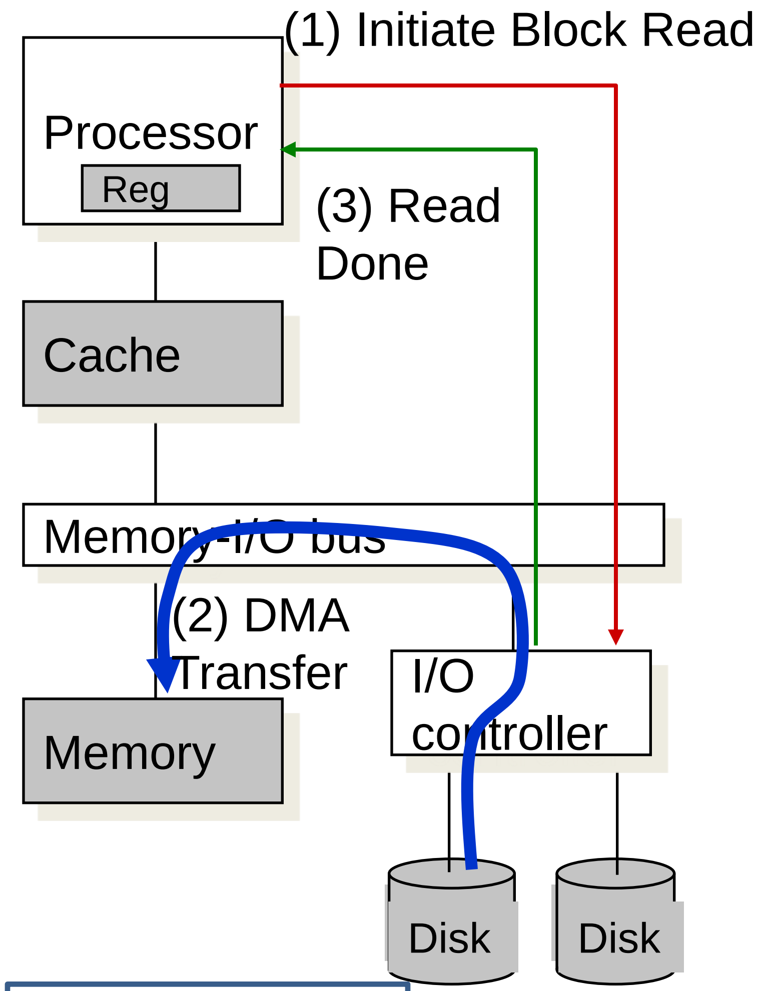
Page Faults (like “Cache Misses”)





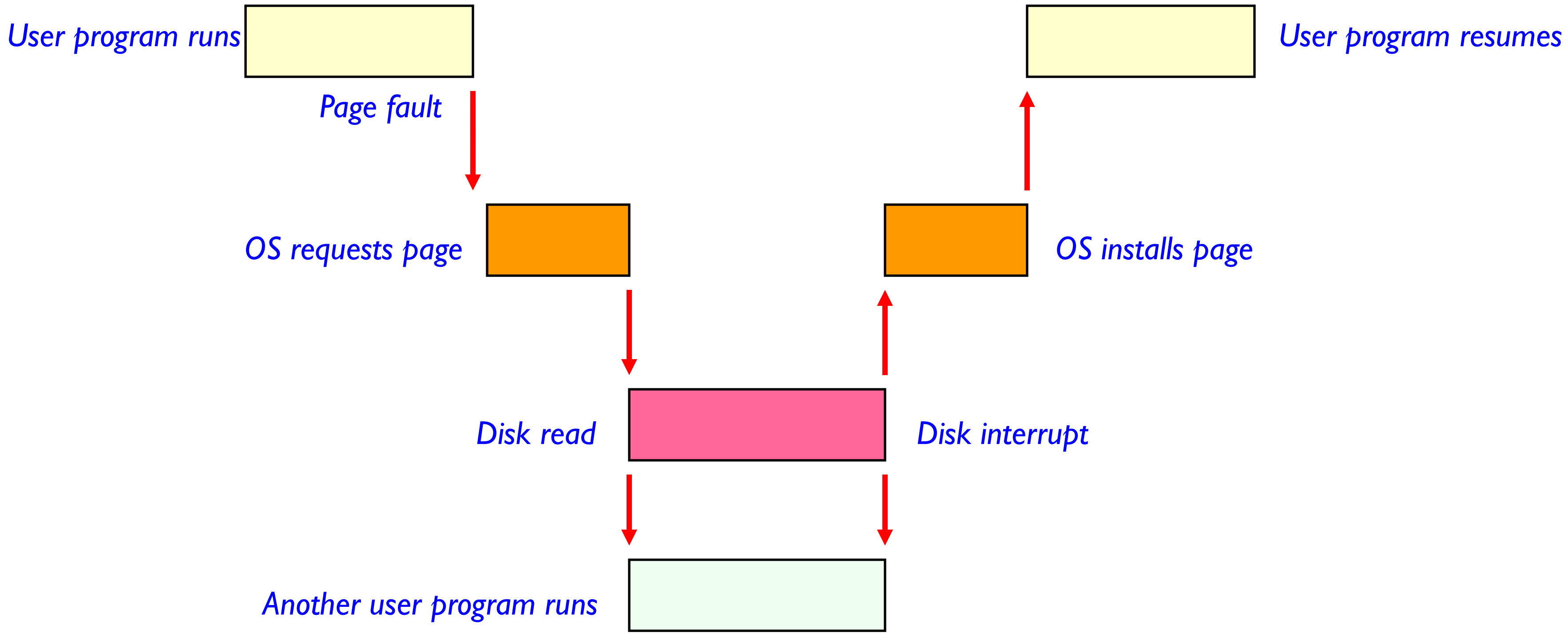
Page Fault Handling

- Processor signals Controller
  - Read block of length P starting at disk address X and store data starting at memory address Y
- Read
  - Direct Memory Access (DMA)
  - Under control of I/O controller
- I / O controller signals completion
  - Interrupts processor
  - OS resumes suspended process



Differences from cache?

Page Fault Handling



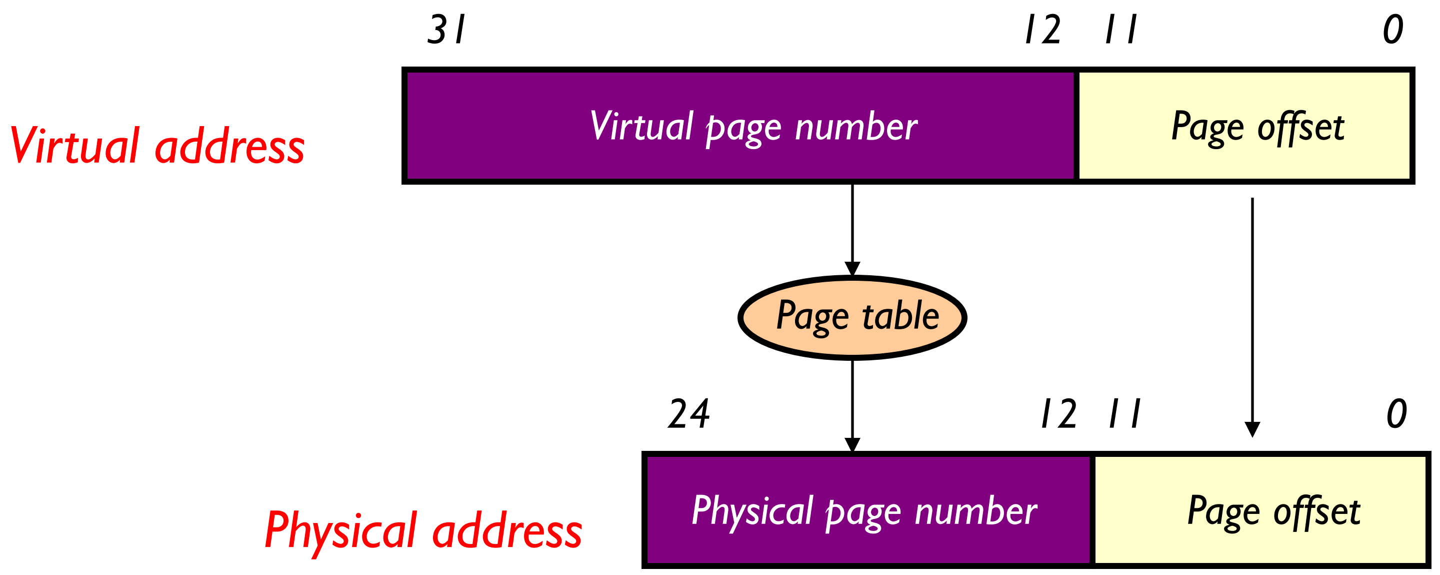
Accelerating Virtual Memory Operations

Virtual Memory Hardware

- Protection via virtual memory
  - Keeps processes in their own memory space

- Role of architecture:
  - Provide user mode and supervisor mode
  - Protect certain aspects of CPU state
  - Provide mechanisms for switching between user mode and supervisor mode
  - Provide mechanisms to limit memory accesses
  - Provide TLB to accelerate the address translation

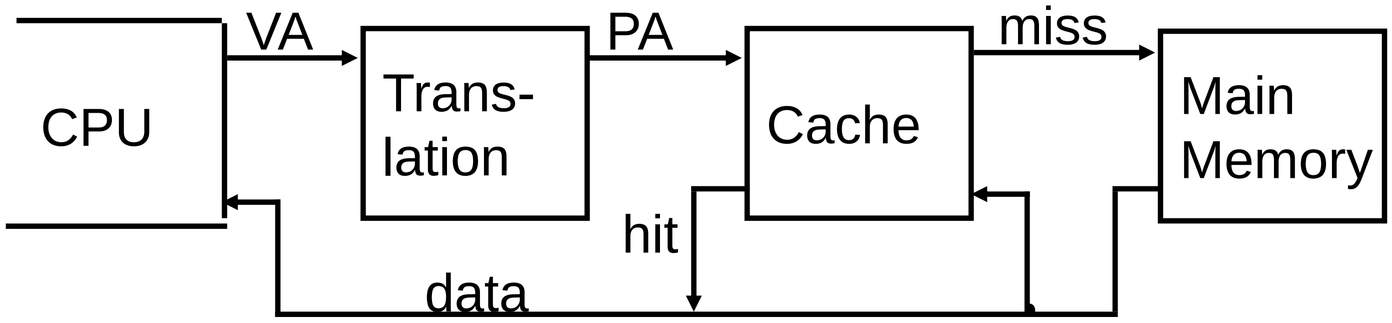
VM => PM Translation



- Each program memory reference requires two memory accesses: one for VM => PM mapping and one for actual data/instruction
  - must make page table lookup as fast as possible
- Page table too big to keep in fast memory (SRAM) in its entirety
  - store page table in main memory
  - cache a portion of the page table in TLB (Translation Look-Aside Buffer)

Virtual Addressing with a Cache

- Thus it takes an *extra* memory access to translate a VA to a PA

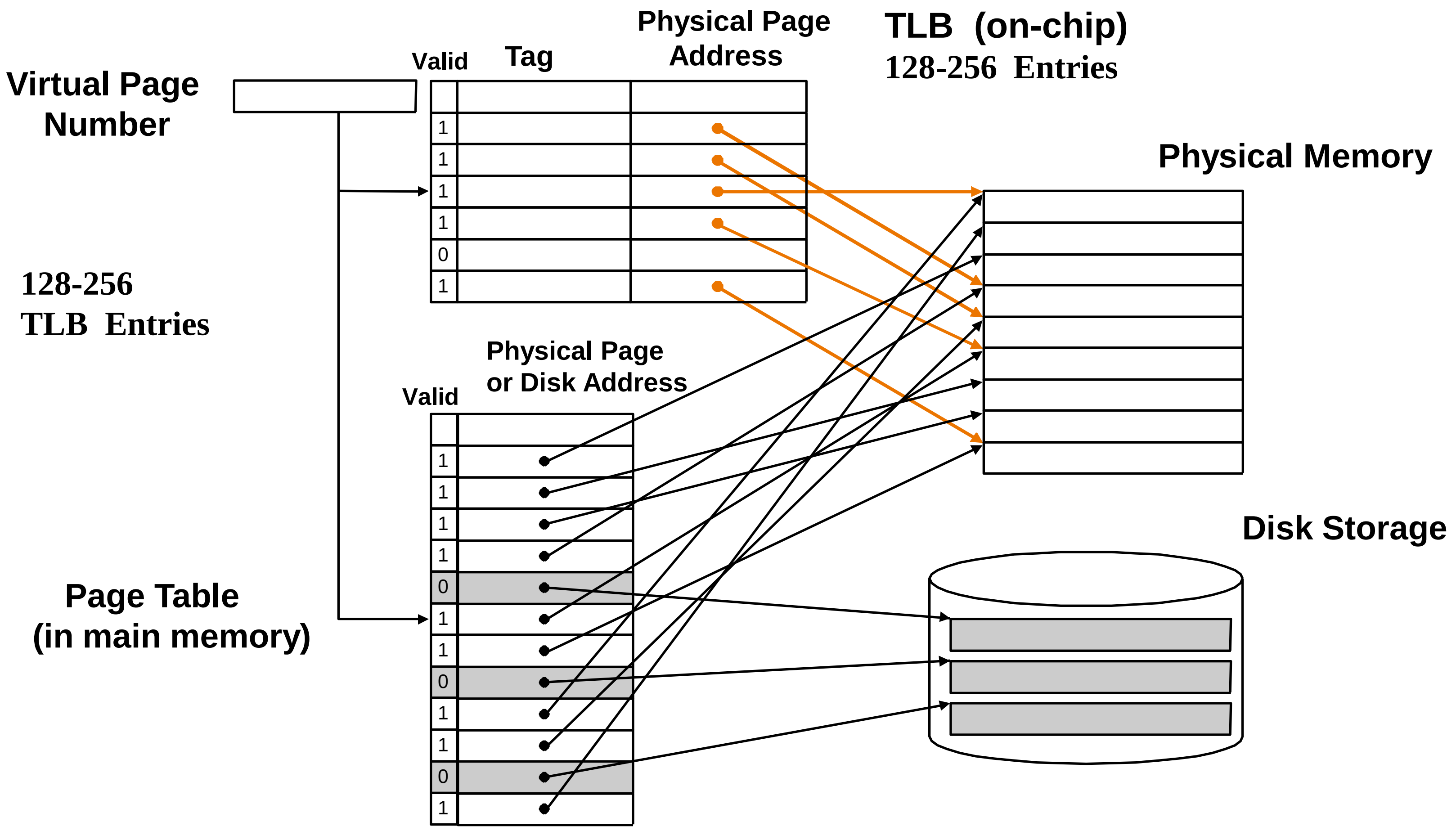


- This makes memory (cache) accesses very expensive (if every access was really *two* accesses)
- Translation Lookaside Buffer (TLB) keeps track of recently used address mappings to avoid having to do a page table lookup

Speeding Up Address Translation:

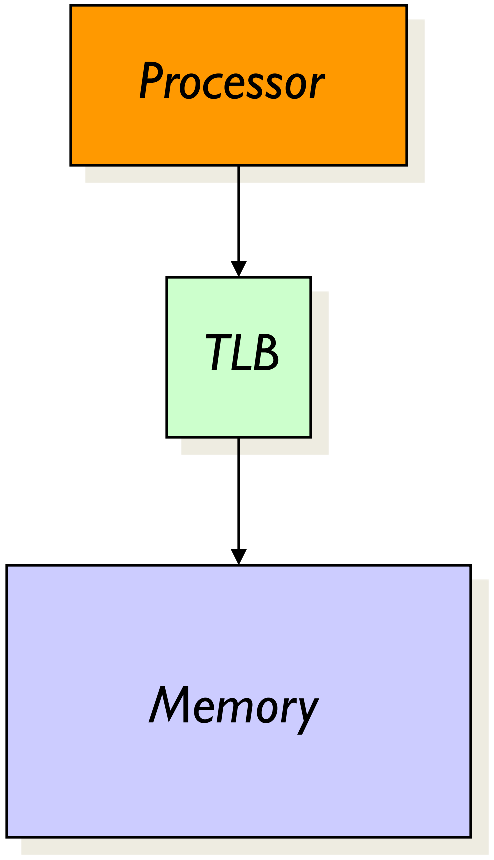
Translation Lookaside Buffer (TLB)

- TLB: A small and fast on-chip memory structure used for address translations.
- If a virtual address is found in TLB (a TLB hit), the page table in main memory is not accessed.

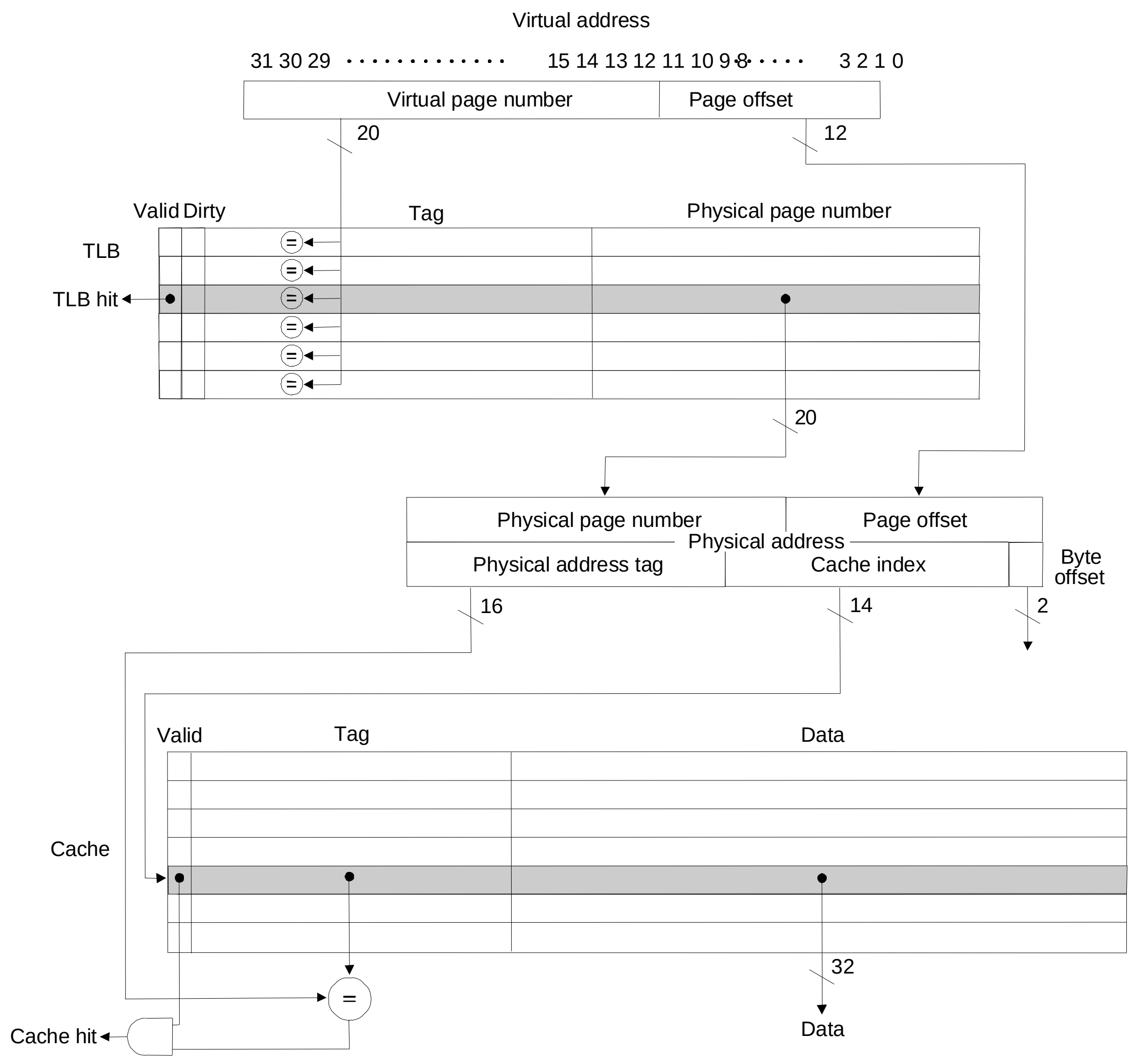


Translation Look-Aside Table (TLB)

- TLB maintains a list of most-recently used pages
- Similar to instruction & data cache
  - virtual address is used to index into the TLB
  - TLB entry contains physical address
  - takes ~ 1 clock cycle
- What if VM=>PM lookup and data/instruction lookup takes more than 1 clock cycle?
  - To avoid TLB lookup, remember the last VM => PM translation
  - if same page is referenced, TLB lookup can be avoided



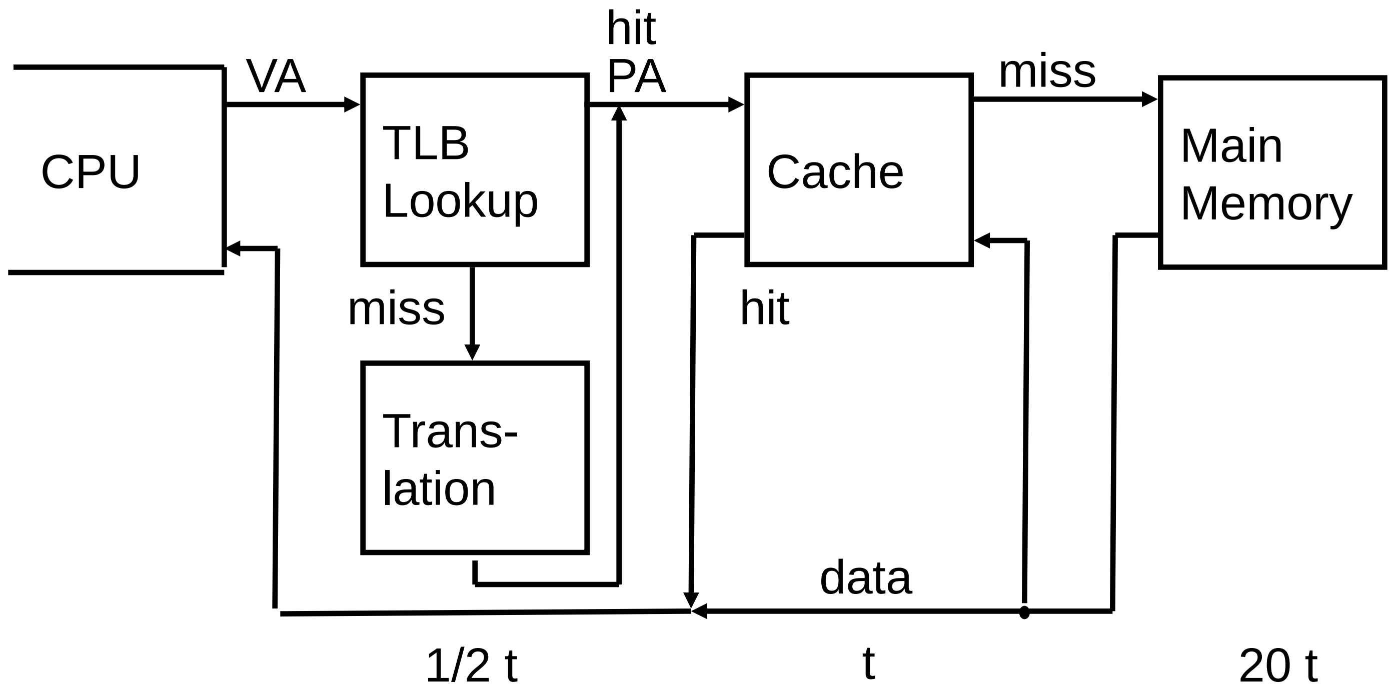
TLB / Cache Interaction



Techniques for Fast Address Translation

- Just like any other cache, the TLB can be organized as fully associative, set associative, or direct mapped
- TLBs are usually small, typically not more than 128 - 256 entries even on high end machines. This permits fully associative lookup on these machines. Most mid-range machines use small n-way set associative organizations.

Translation with a TLB



TLB and Context Switch

- In multi-programming we need to use TLB for the active process
  - What to do with TLB at context switch?
- Too costly to clear TLB on every context switch
- Keep track of PTE in TLB per process using ASID

TLB Organization

Tag

Virtual address	Physical address	Used	Dirty	Valid	Access	ASID
0xFA00	0x0003	N	Y	Y	R/W	34
0x0040	0x0010	Y	N	Y	R	0
0x0041	0x0011	Y1	N	Y	R	0

Additional info per page

- Dirty: page modified (if the page is swapped out, should the disk copy be updated?)
- Valid: TLB entry valid
- Used: Recently used or not (for selecting a page for eviction)
- Access: Read/Write
- ASID:Address Space ID

Example

Consider a virtual memory system with the following properties:

- 32-bit virtual address (4 Gbytes)
- 26-bit physical memory (64 Mbytes)
- 4 Kbyte pages (12 bits)

a. How big is the page table for this memory system, assuming each page table entry has 4 overhead bits *(valid, protection, dirty, use)* and holds one physical page number? Give your answer in bits.

b. Approximately how many pages of user data, at maximum, can we have in physical memory?

c. If we wanted to keep the page table under 1Mbit in size, how big would we have to make the pages? Assume a page needs to be 2^K bytes, and find the correct value of K.

Part a

$$AddressSpaceSize = 2^{32} = 4 \times 2^{30} \approx 4 \times 10^{9}\ \ Bytes$$

$$NumberOfPages_{virtual} = \frac{AddressSpaceSize}{PageSize} = \frac{2^{32}}{4 \times 1000} \approx \frac{2^{32}}{2^{12}} = 2^{20}$$

$$NumberOfPages_{virtual} \approx 1,000,000$$

The page table must have 1,000,000 entries. To compute the size in bits note that the page offset need not be stored in the page table. There are 4 Kbytes per page requiring 12 bits of offset ($4000 = 4 \times 10^{3} \approx 2^{12}$). Thus each page table entry must map 20 bits of virtual page number (32-12=20) to (26-12=14) bits of physical page number. Taking into account the 4 overhead bits, each entry requires 18 bits. With 1,000,000 entries, the page table size is 18 megabits.

Part b

$$NumberOfPages_{physical} = \frac{AddressSpaceSize_{physical}}{PageSize} = \frac{2^{26}}{4 \times 1000} \approx \frac{2^{26}}{2^{12}} = 2^{14}$$

$$NumberOfPages_{physical} \approx 16,000$$

Part c

$$PageTableSize = 1 \text{ Megabits} = \frac{10^6}{8} \text{ Megabytes} \approx \frac{2^{20}}{2^3} = 2^{17} \text{ Megabytes}$$

Assuming that the page is 2^K bytes, we need K bits for page offset in both the physical and virtual addresses. Thus each page entry requires:

$$PageEntrySize = (26 - K) + 4 = 30 - K \text{ bits} = \frac{30 - K}{8} Bytes$$

Now we must calculate the number of entries (corresponding to the number of virtual pages) that must be kept in the page table:

$$NumberOfPages_{virtual} = \frac{AddressSpaceSize}{PageSize} = \frac{2^{32}}{2^K} = 2^{32-K}$$

The following relationship holds:

$$PageTableSize = PageEntrySize \times NumberOfPages_{virtual}$$

$$2^{17} = \frac{30 - K}{8} \times 2^{32-K}$$

$$2^{K-12} = 30 - K \Rightarrow 15 < K < 16$$

We must pick K=16 to stay under the specified limit of 1 Mbits of page table.

Cache, Virtual Memory and Virtual Machines

Cache and Virtual Memory

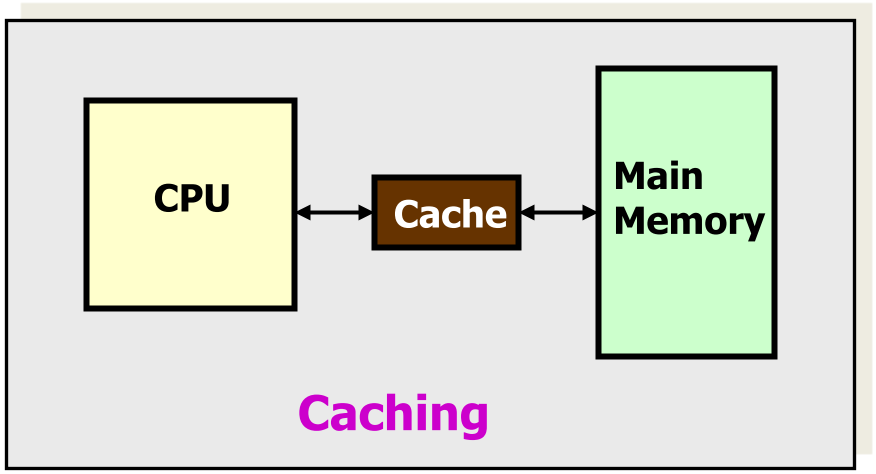
- Which address to use to index to cache sets?
  - Physical address? May be slow due to translation
  - Virtual? Protection, process switching, aliasing
- How to make a virtual cache work?
  - Page coloring
- Get the best of both: virtually indexed, physically tagged cache

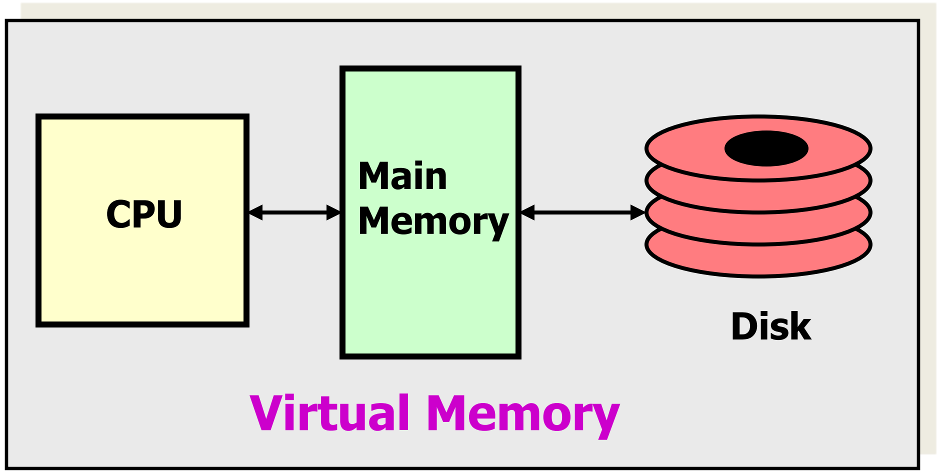
Cache vs. Virtual Memory

- Concept behind VM is almost identical to concept behind cache.
- But different terminology
  - Cache: Block VM: Page
  - Cache: Cache Miss VM: Page Fault
- Caches implemented completely in hardware. VM implemented in software, with hardware support from the processor.
- Cache speeds up main memory access, while main memory speeds up VM access.

VM & Caching Comparison

Parameter	First-level cache	Virtual memory
Block (page)	16-128 bytes	4096-65,536 bytes
Hit time	1-3 clock cycles	50-150 clock cycles
Miss penalty	8-150 clock cycles	1,000,000-10,000,000 clock cycles
Access time	6-130 clock cycles	800,000-8,000,000 clock cycles
Transfer time	2-20 clock cycles	200,000-2,000,000 clock cycles
Miss rate	0.1-10%	0.00001-0.001%
Address mapping	25-45 bit physical address to 14-20 bit cache address	32-64 bit virtual address to 25-45 bit physical address





Impact of These Properties on Design

- Comparing to cache, how would we set the following design parameters?
  - block size?
    - Large, since disks perform better at transferring large blocks
  - Associativity?
    - High, to minimize miss rate
  - Write through or write back?
    - Write back, since can't afford to perform small writes to disk
- What would the impact of these choices be on:
  - miss rate
    - Extremely low. << 1%
  - hit time
    - Must match cache/DRAM performance
  - miss latency (penalty)
    - Very high. ~20ms

Associativity of VM

- Cache miss penalty: 8-150 clock cycles
- VM miss penalty: 1,000,000 - 10,000,000 clock cycles
- Because of the high miss penalty, VM design minimizes miss rate by allowing full associativity for page placement

Write Strategies

- Disk I/O slow (millions of clock cycles)
- Always write-back; never write-through
- Use dirty bit to decide whether to write disk before eviction
- Smart disk controllers buffers writes
  – copy replaced page in buffer
  – read new page into main memory
  – write from buffer to disk

Virtual Machines

- Supports isolation and security
- Sharing a computer among many unrelated users
- Enabled by raw speed of processors, making the overhead more acceptable

- Allows different ISAs and operating systems to be presented to user programs
  - "System Virtual Machines"
  - SVM software is called "virtual machine monitor" or "hypervisor"
  - Individual virtual machines run under the monitor are called "guest VMs" or "guest OSes"

Impact of VMs on Virtual Memory

Each guest OS maintains its own set of page tables

- VMM adds a level of memory between physical and virtual memory called “real memory”
- VMM maintains shadow page table that maps guest virtual addresses to physical addresses
  - Requires VMM to detect guest’s changes to its own page table
  - Occurs naturally if accessing the page table pointer is a privileged operation