CSE 3320
Operating Systems
Memory Management

Jia Rao

Department of Computer Science and Engineering

http://ranger.uta.edu/~jrao

Recap of Previous Classes

- Multiprogramming
  - Requires multiple programs to run at the “same” time
  - Programs must be brought into memory and placed within a process for it to be run
  - How to manage the memory of these processes?
  - What if the memory footprint of the processes is larger than the physical memory?

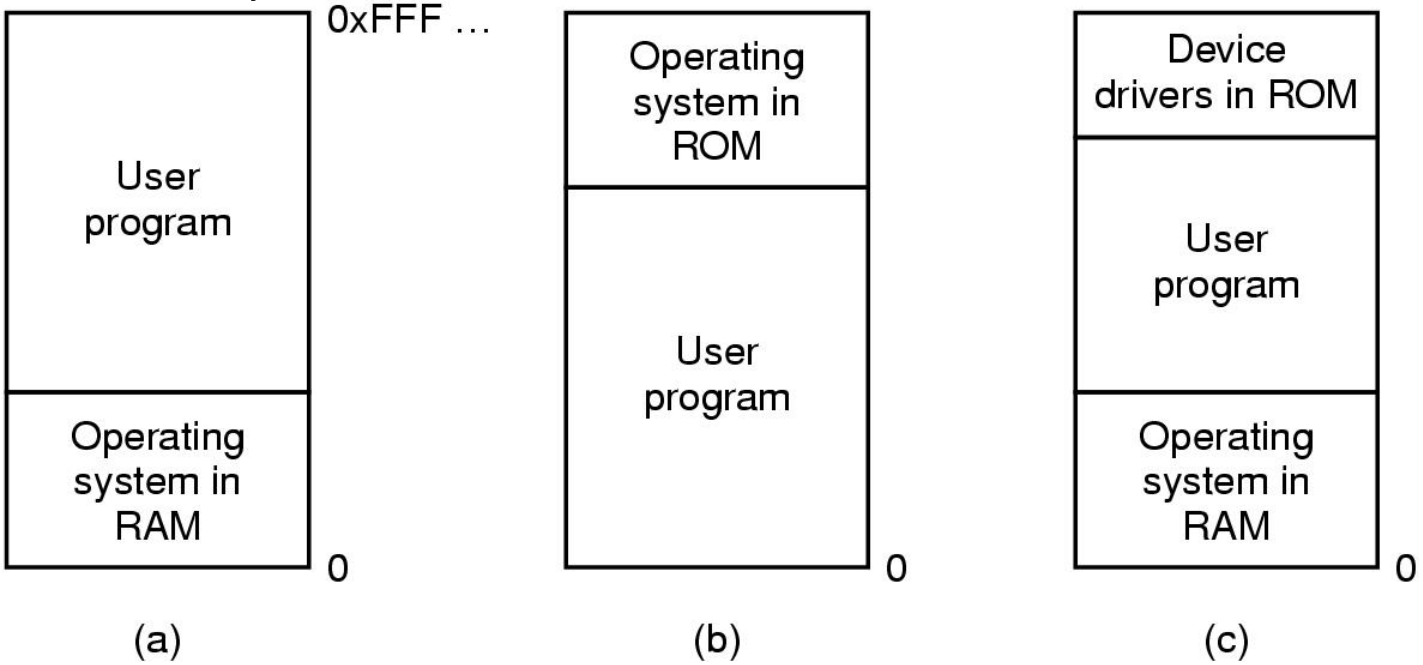
Memory Management

- Ideally programmers want memory that is
  - large
  - fast
  - non volatile
  - and cheap
- Memory hierarchy
  - small amount of fast, expensive memory – cache
  - some medium-speed, medium price main memory
  - gigabytes of slow, cheap disk storage
- Memory management tasks
  - Allocate and de-allocate memory for processes
  - Keep track of used memory and by whom
  - Address translation and protection

Memory is cheap and large in today's desktop, why memory management is still important?

No Memory Abstraction

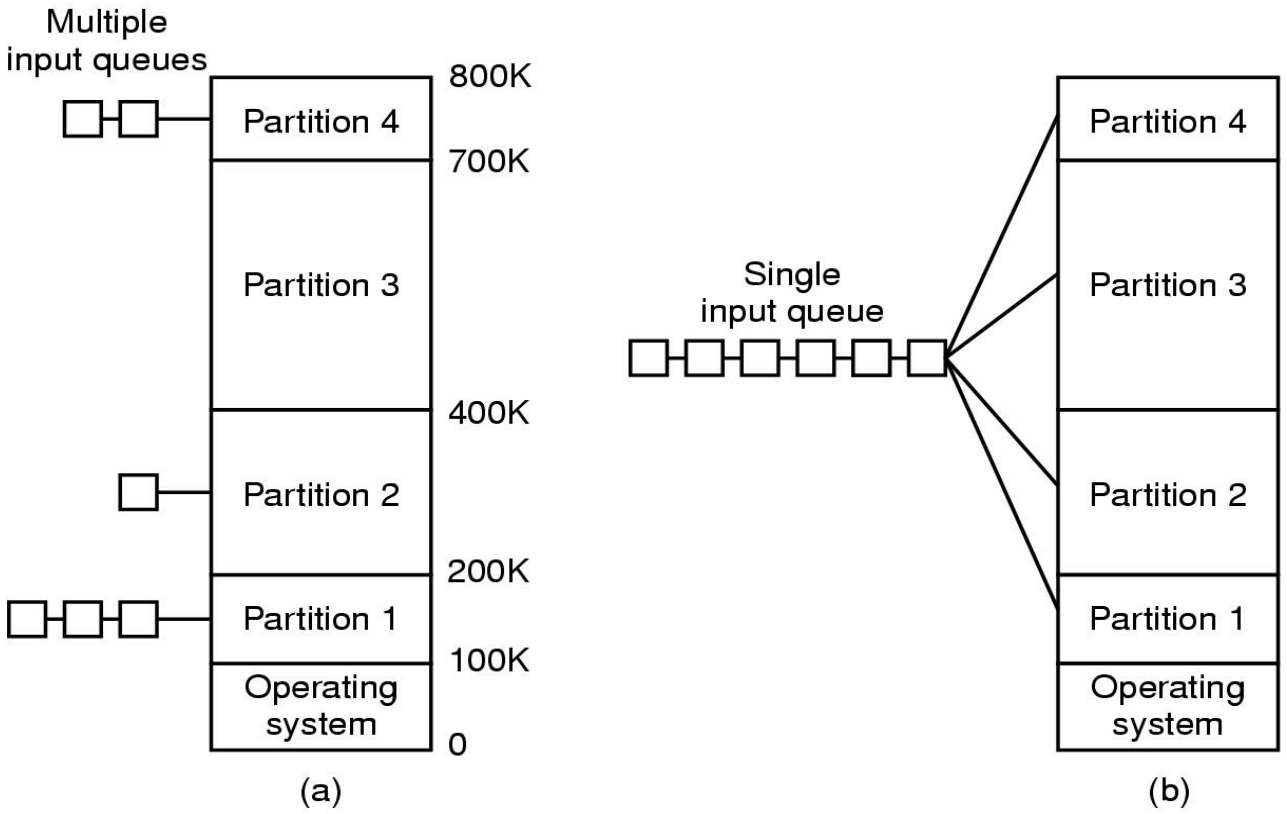
- Mono-programming: One program at a time, sharing memory with OS



Three simple ways of organizing memory. (a) early mainframes. (b) Handheld and embedded systems. (c) early PC.

- Need for multi-programming: 1. Use multiple CPUs. 2. Overlapping I/O and CPU

Multiprogramming with Fixed Partitions



- Fixed-size memory partitions, without swapping or paging
  - Separate input queues for each partition
  - Single input queue
  - Various job schedulers

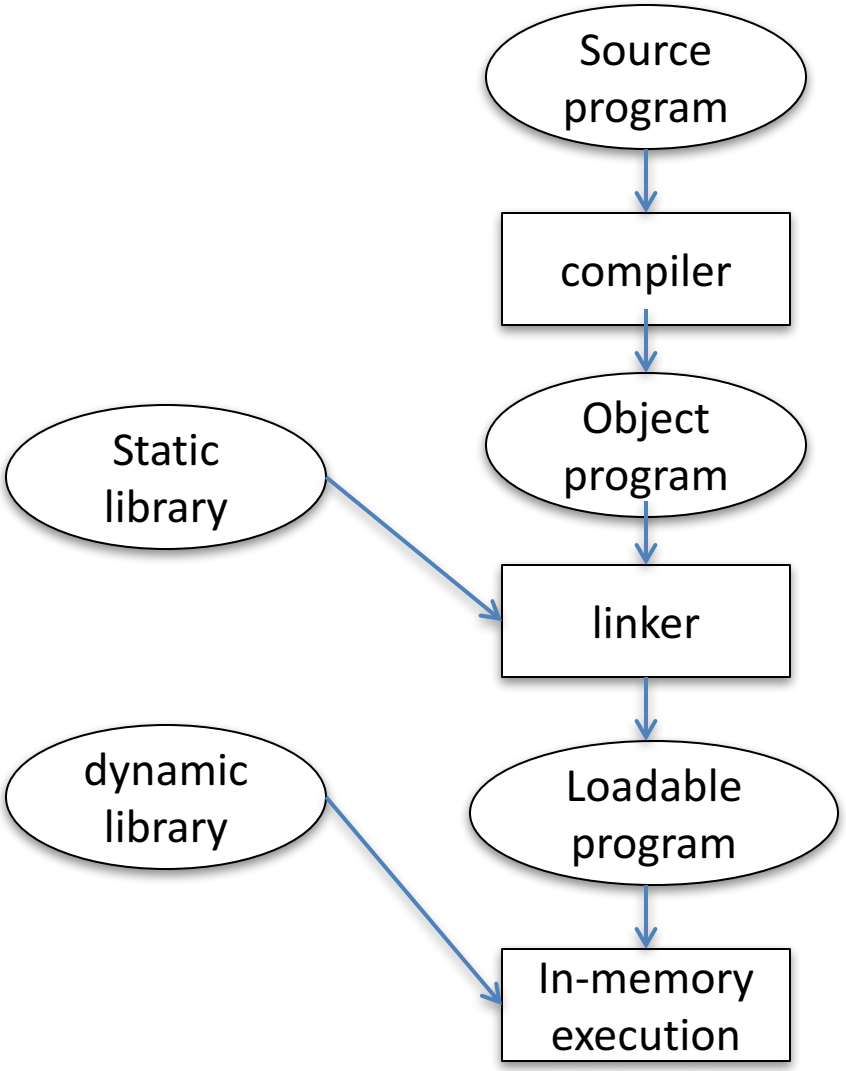
What are the disadvantages?

Multiprogramming w and w/o Swapping

- Swapping
  - One program at one time
  - Save the entire memory of the previous program to disk
  - No address translation is required and no protection is needed
- w/o swapping
  - Memory is divided into blocks
  - Each block is assigned a protection key
  - Programs work on absolute memory, no address translation
  - Protection is enforced by trapping unauthorized accesses

Running a Program*

- Compile and link time
  - A linker performs relocation if the memory address is known
- Load time
  - Must generate relocatable code if memory location is not known at compile time



Relocation and Protection

- Relocation: what address the program will begin in in memory
  - Static relocation: OS performs one-time change of addresses in program
  - Dynamic relocation: OS is able to relocate a program at runtime
- Protection: must keep a program out of other processes' partitions

Static relocation – OS can not move it once a program is assigned a place in memory

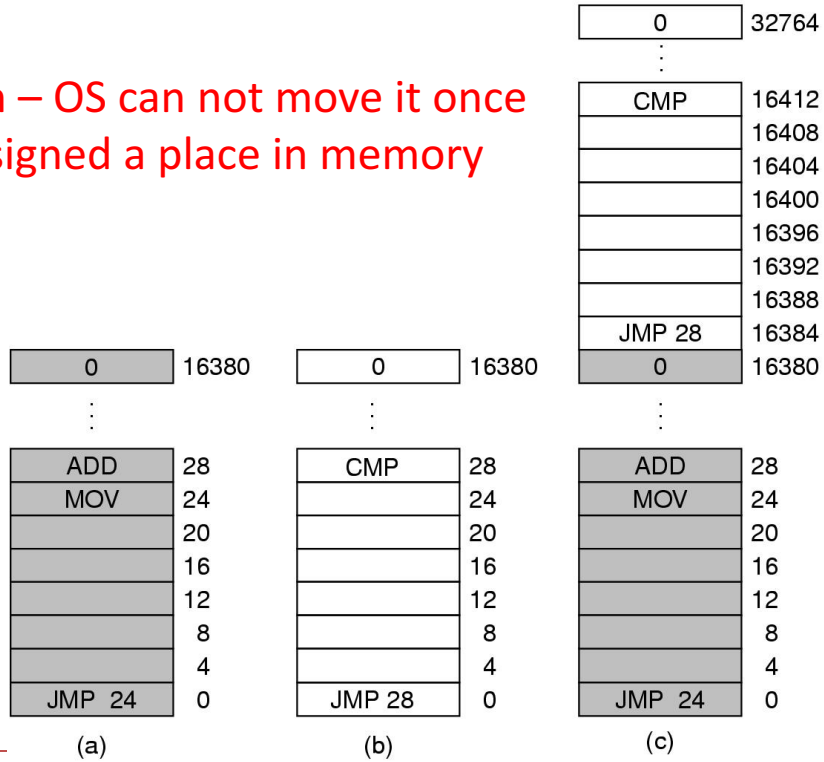


Illustration of the relocation problem.

A Memory Abstraction: Address Spaces

- Exposing the entire physical memory to processes
  - Dangerous, may trash the OS
  - Inflexible, hard to run multiple programs simultaneously
- Program should have their own views of memory
  - The address space – logical address
  - Non-overlapping address spaces – protection
  - Move a program by mapping its addresses to a different place - relocation

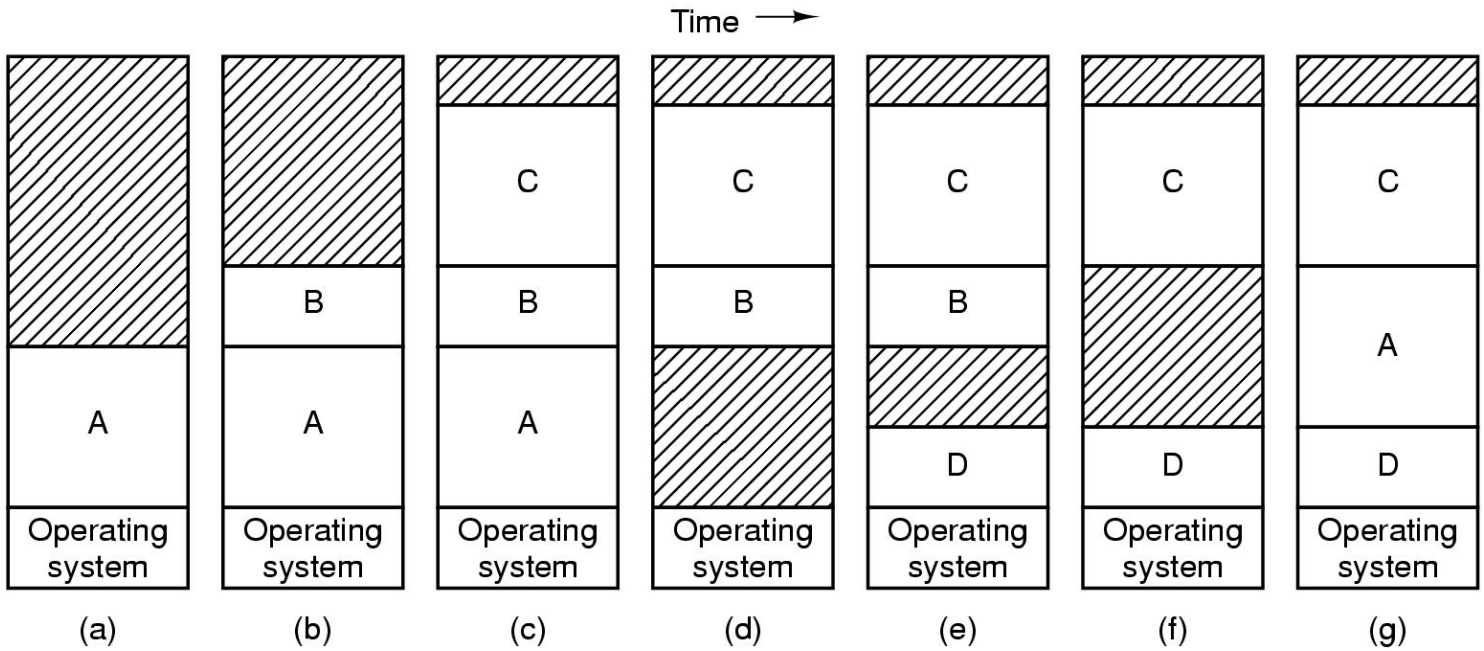
Dynamic Relocation

- OS dynamically relocates programs in memory
  - Two hardware registers: base and limit
  - Hardware adds relocation register (base) to virtual address to get a physical address
  - Hardware compares address with limit register address must be less than limit

Disadvantage: two operations on every memory access and the addition is slow

Dealing with Memory Overload - Swapping

- Swapping: bring in each process in its *entirety*, M-D-M- …
- Key issues: allocating and de-allocating memory, keep track of it



Memory allocation changes as

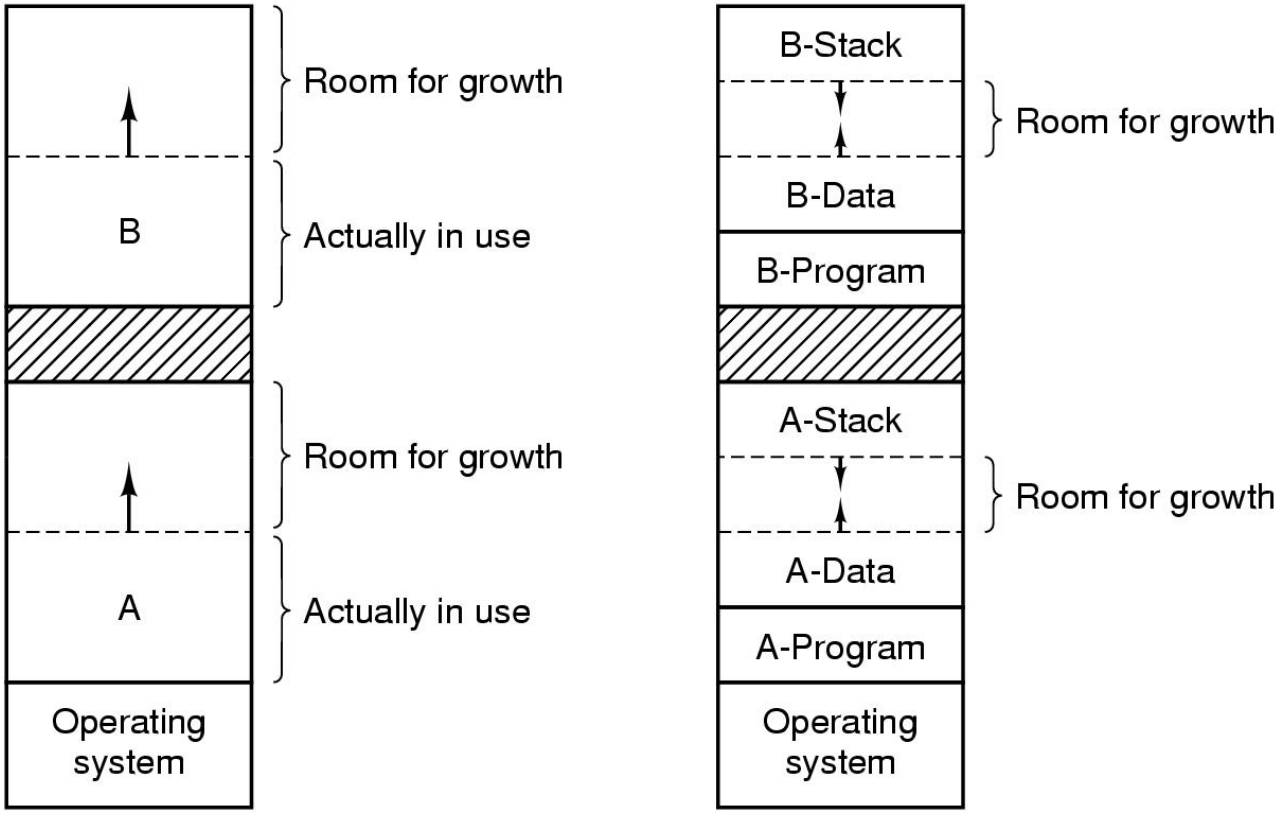
- processes come into memory
- leave memory

Shaded regions are unused memory (memory holes)

Why not memory compaction?

Another way is to use virtual memory

Swapping – Memory Growing



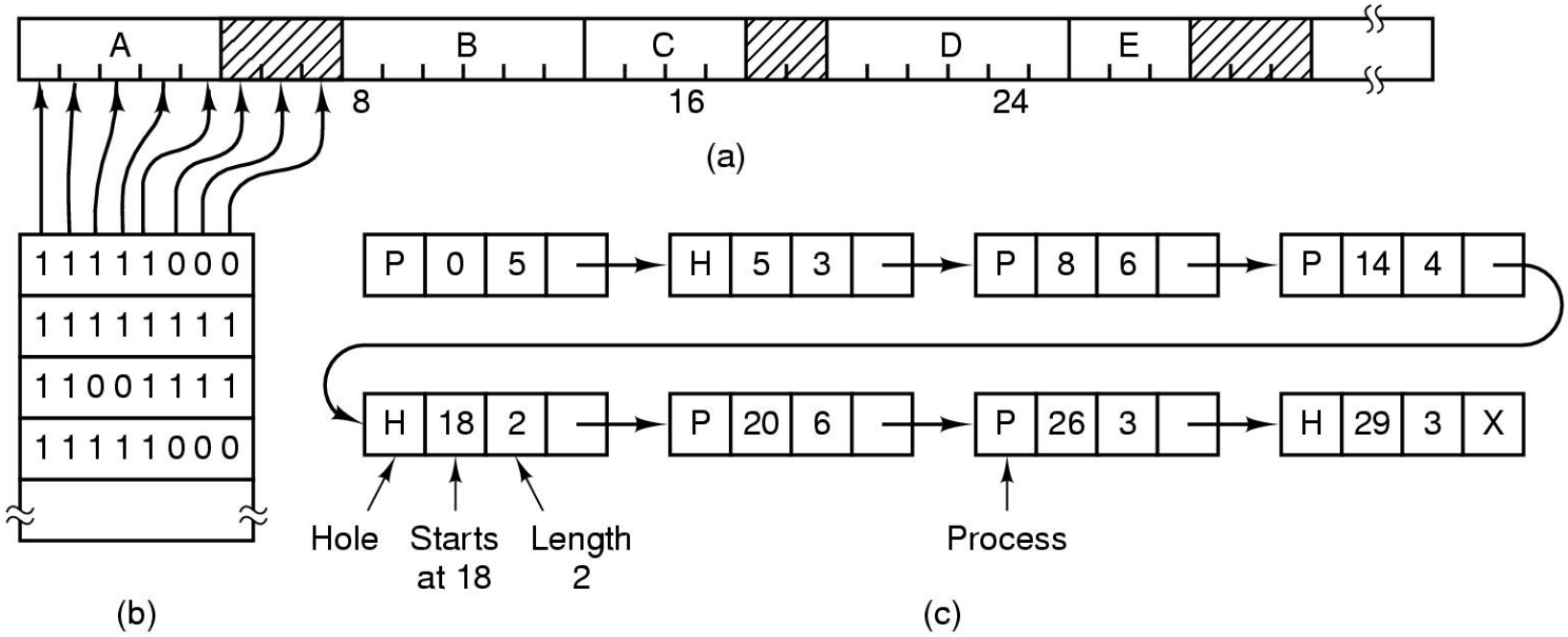
Why stack grows downward?

(a) Allocating space for growing *single (*data) segment

(b) Allocating space for growing stack & data segment

Memory Management with Bit Maps

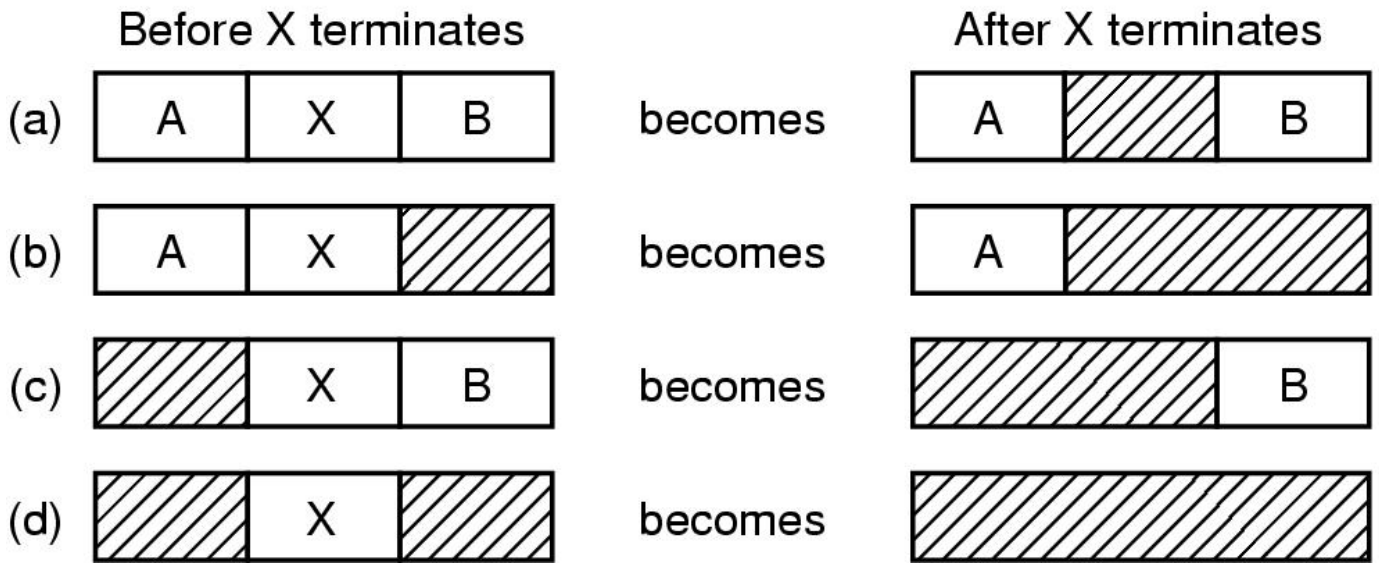
- Keep track of dynamic memory usage: bit map and free lists



- Part of memory with 5 processes, 3 holes
  - tick marks show allocation units (what is its desirable size?)
  - shaded regions are free
- Corresponding *bit map* (searching a bitmap for a run of n 0s?)
- Same information as a *list* (better using a doubly-linked list)

Memory Management with Linked Lists

- *De-allocating* memory is to update the list



Four neighbor combinations for the terminating process X

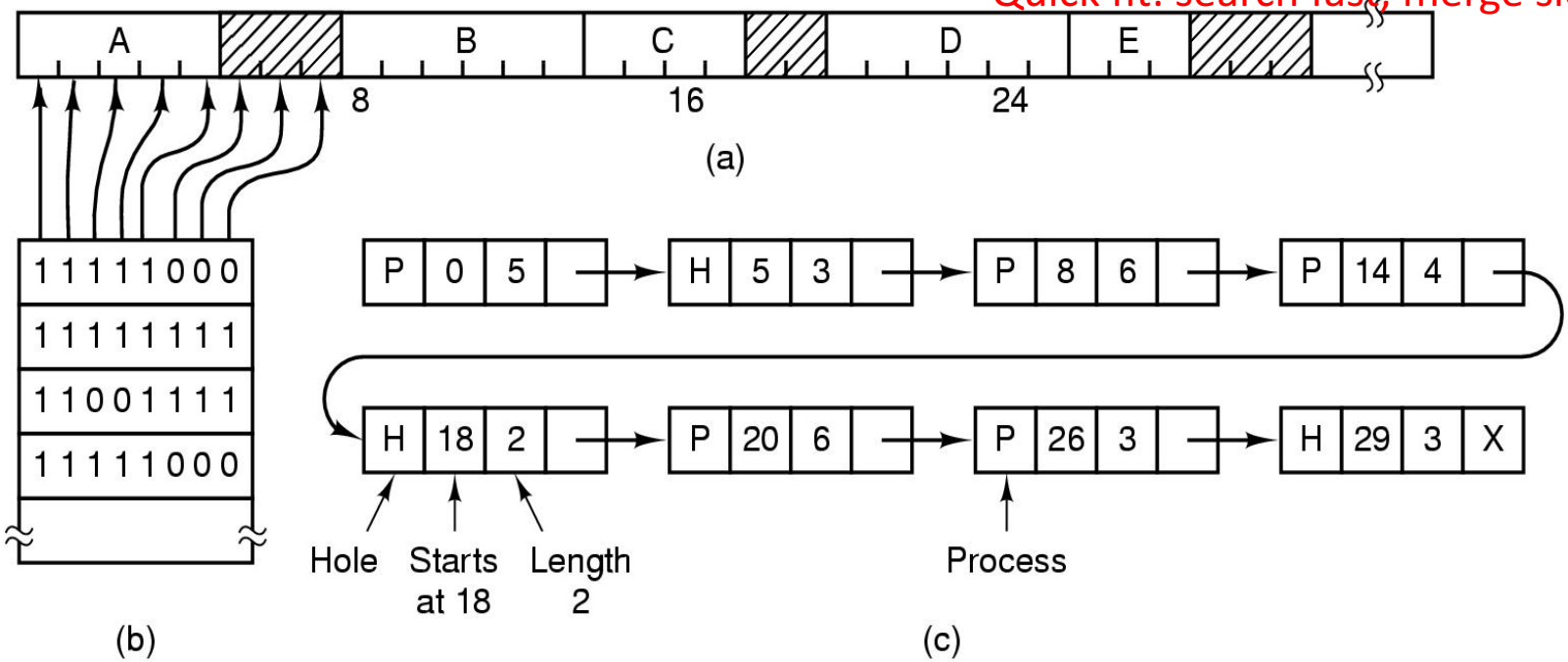
Memory Management with Linked Lists (2)

- How to *allocate* memory for a newly created process (or swapping) ?
  - First fit: allocate the first hole that is big enough
  - Best fit: allocate the smallest hole that is big
  - Worst fit: allocate the largest hole
  - How about separate P and H lists for searching speedup? But with what cost?

Which strategy is the best?

- Example: a block of size 2 is needed for memory allocation

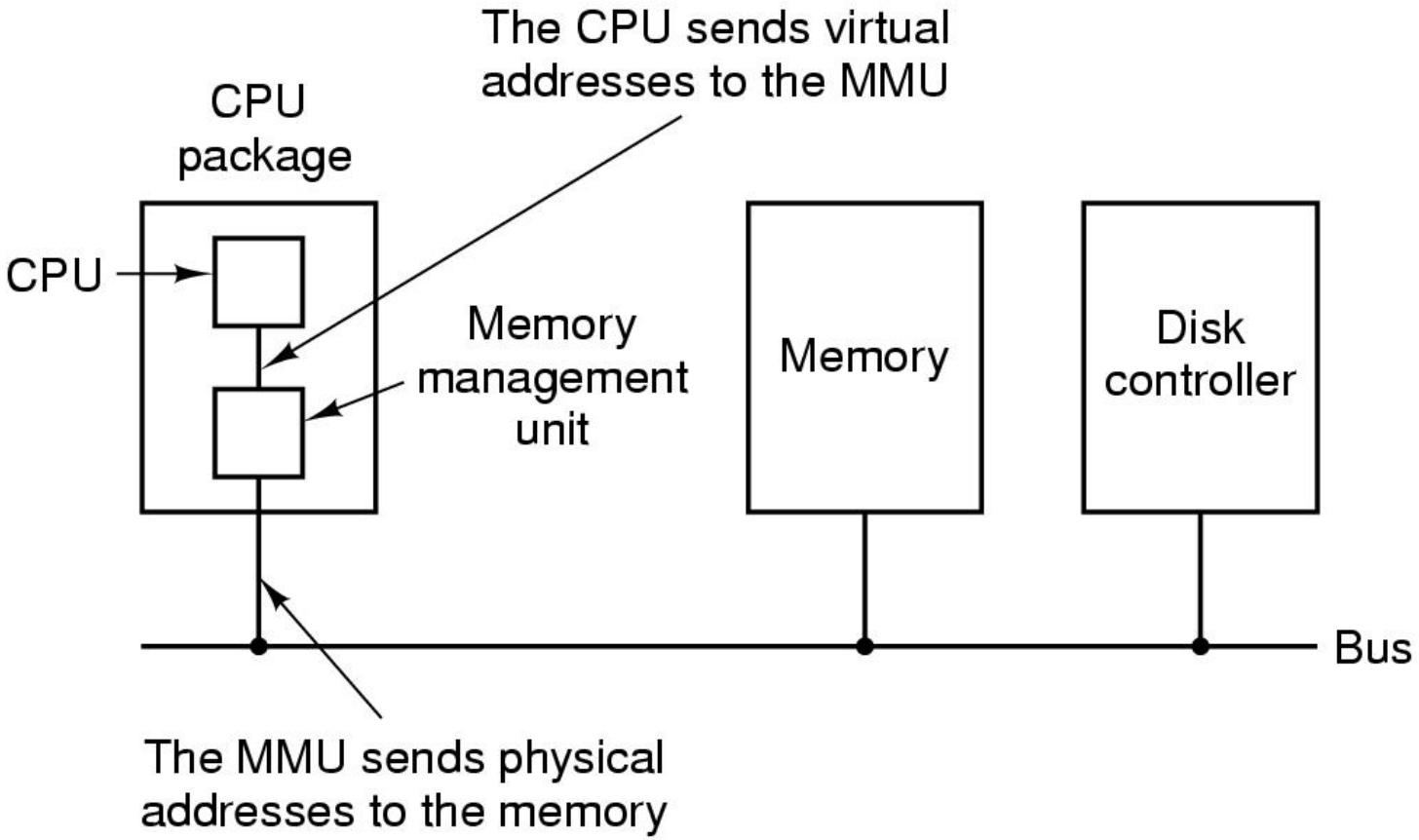
Quick fit: search fast, merge slow



Virtual Memory

- Virtual memory: the combined size of the program, data, and stack may exceed the amount of physical memory available.
  - Swapping with overlays; but hard and time-consuming to split a program into overlays by the programmer
  - What to do more efficiently?

Mapping of Virtual addresses to Physical addresses



Logical program works in its contiguous virtual address space

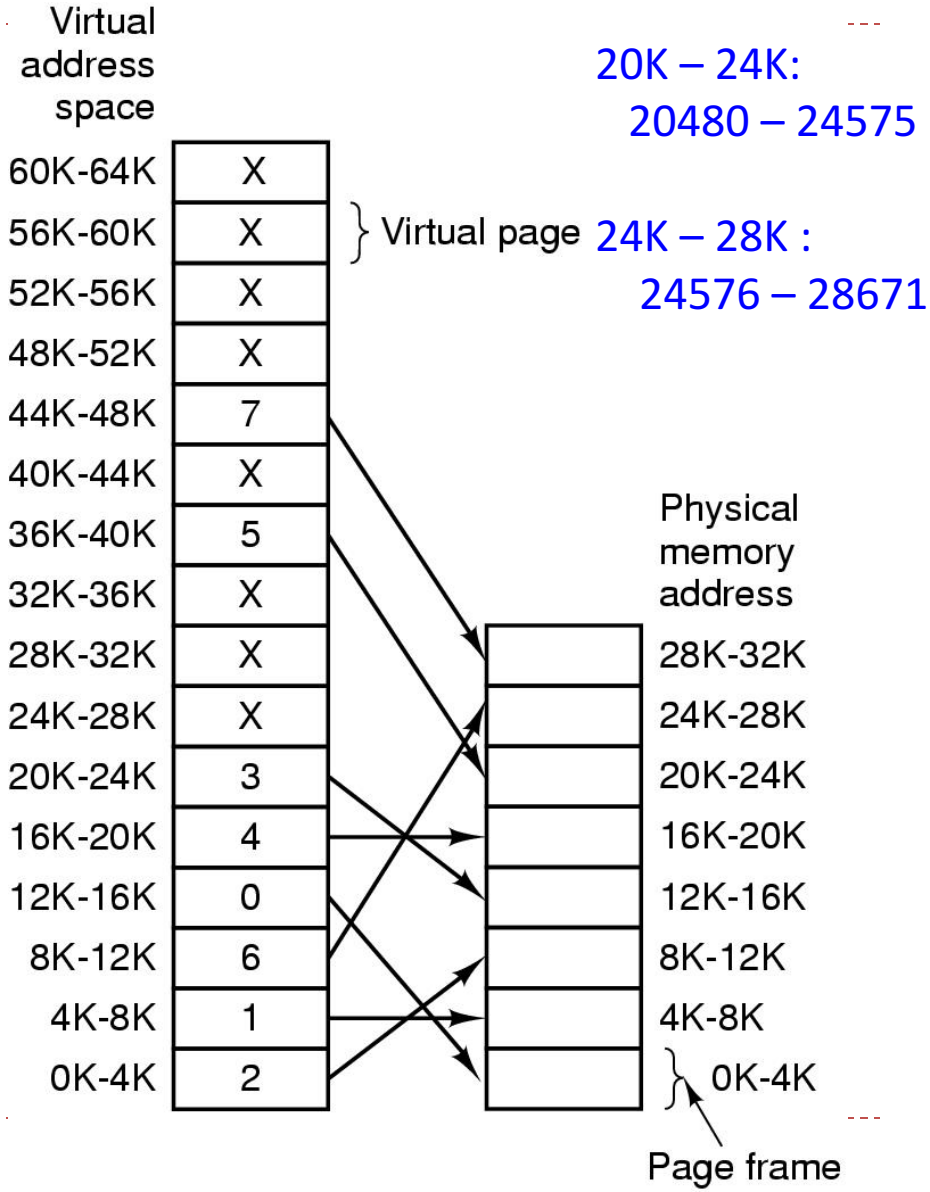
Address translation done by MMU

Actual locations of the data in physical memory

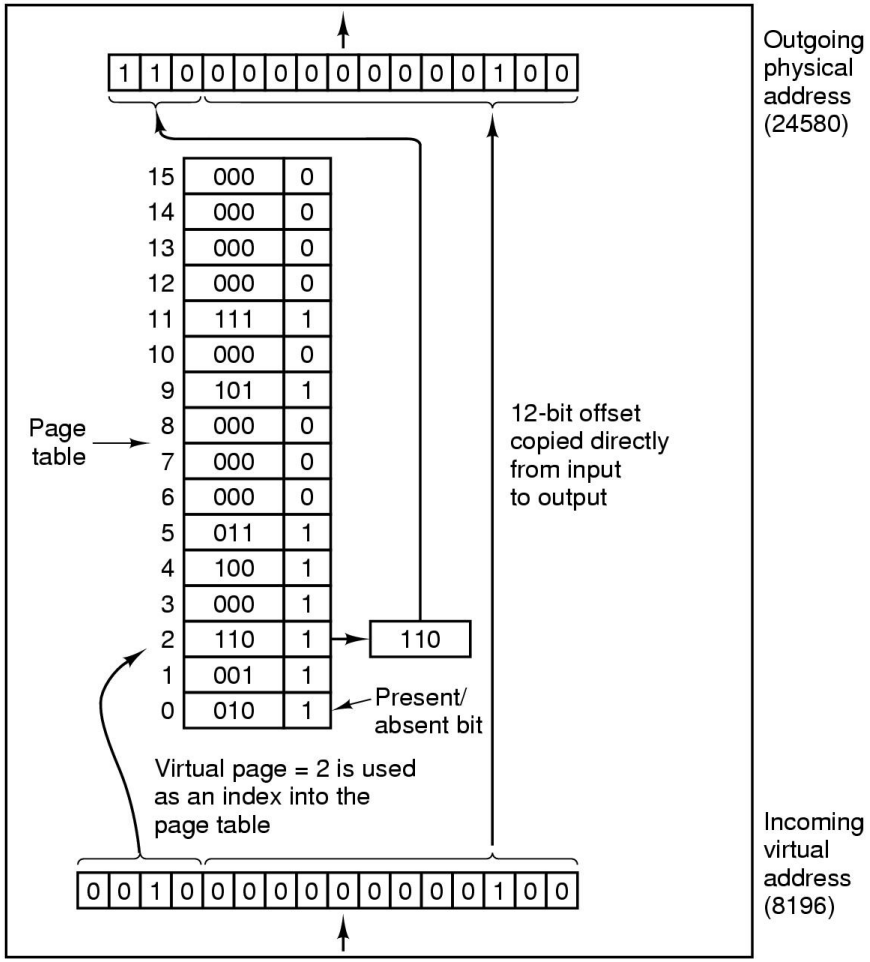
Paging and Its Terminology

- Terms
  - Pages
  - Page frames
  - Page hit
  - Page fault
  - Page replacement
- Examples:
  - MOV REG, 0
  - MOV REG, 8192
  - MOV REG, 20500
  - MOV REG, 32780

Page table gives the relation between virtual addresses and physical memory addresses



Page Tables



Internal operation of MMU with 16 4 KB pages

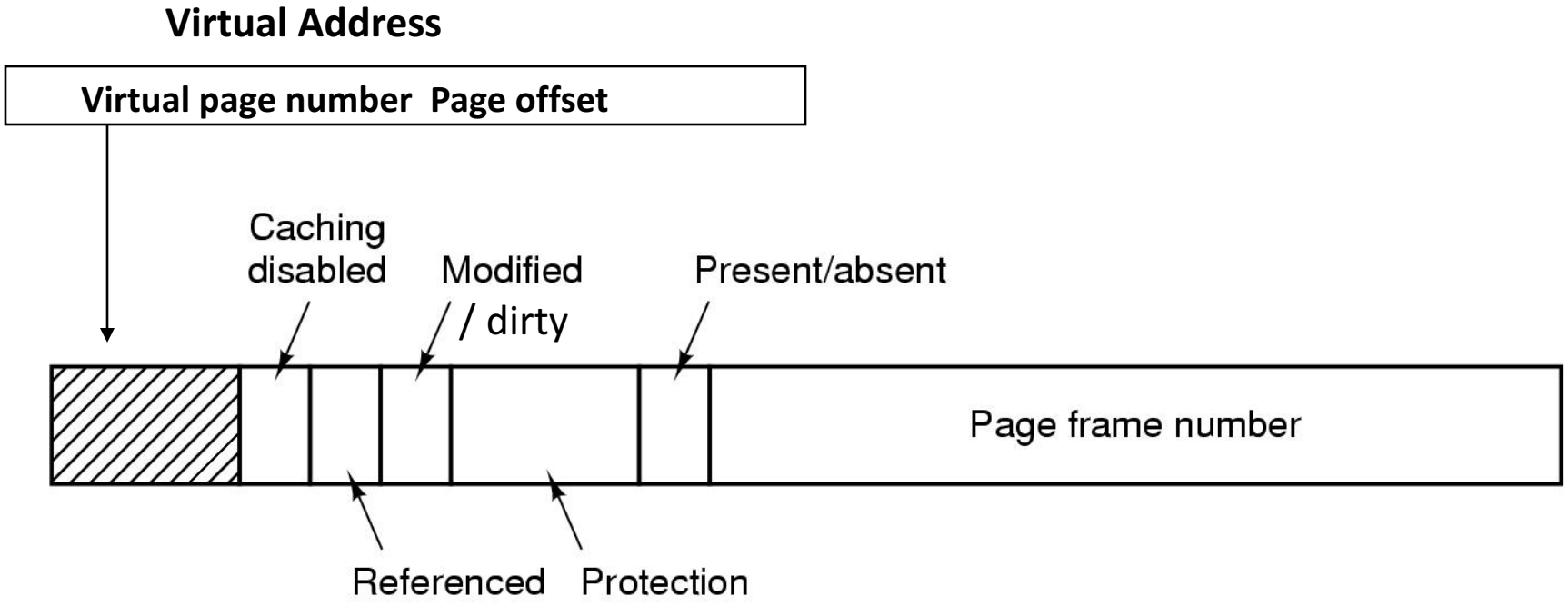
Two issues:

1. Mapping must be fast

2. Page table can be large

Who handles page faults?

Structure of a Page Table Entry



Who sets all those bits?

Translation Look-aside Buffers (TLB)

Taking advantage of Temporal Locality:

A way to speed up address translation is to use a special cache of recently used page table entries -- this has many names, but the most frequently used is *Translation Lookaside Buffer* or *TLB*

Virtual page number (virtual page #)	Cache	Ref/use	Dirty	Protection	Physical Address (physical page #)

TLB access time comparable to cache access time;
much less than Page Table (usually in main memory) access time

Who handles TLB management and handling, such as a TLB miss?

Traditionally, TLB management and handling were done by MMU Hardware, today, some in software / OS (many RISC machines)

A TLB Example

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

A TLB to speed up paging (usually inside of MMU traditionally)

Page Table Size

Given a 32-bit virtual address,
4 KB pages,
4 bytes per page table entry (memory addr. or disk addr.)

What is the size of the page table?

The number of page table entries:

2^32 / 2^12 = 2^20

What if 2KB pages?

2^32/2^11*2^2 = 2^23 (8 MB)

The total size of page table:

2^20 * 2^2 = 2^22 (4 MB)

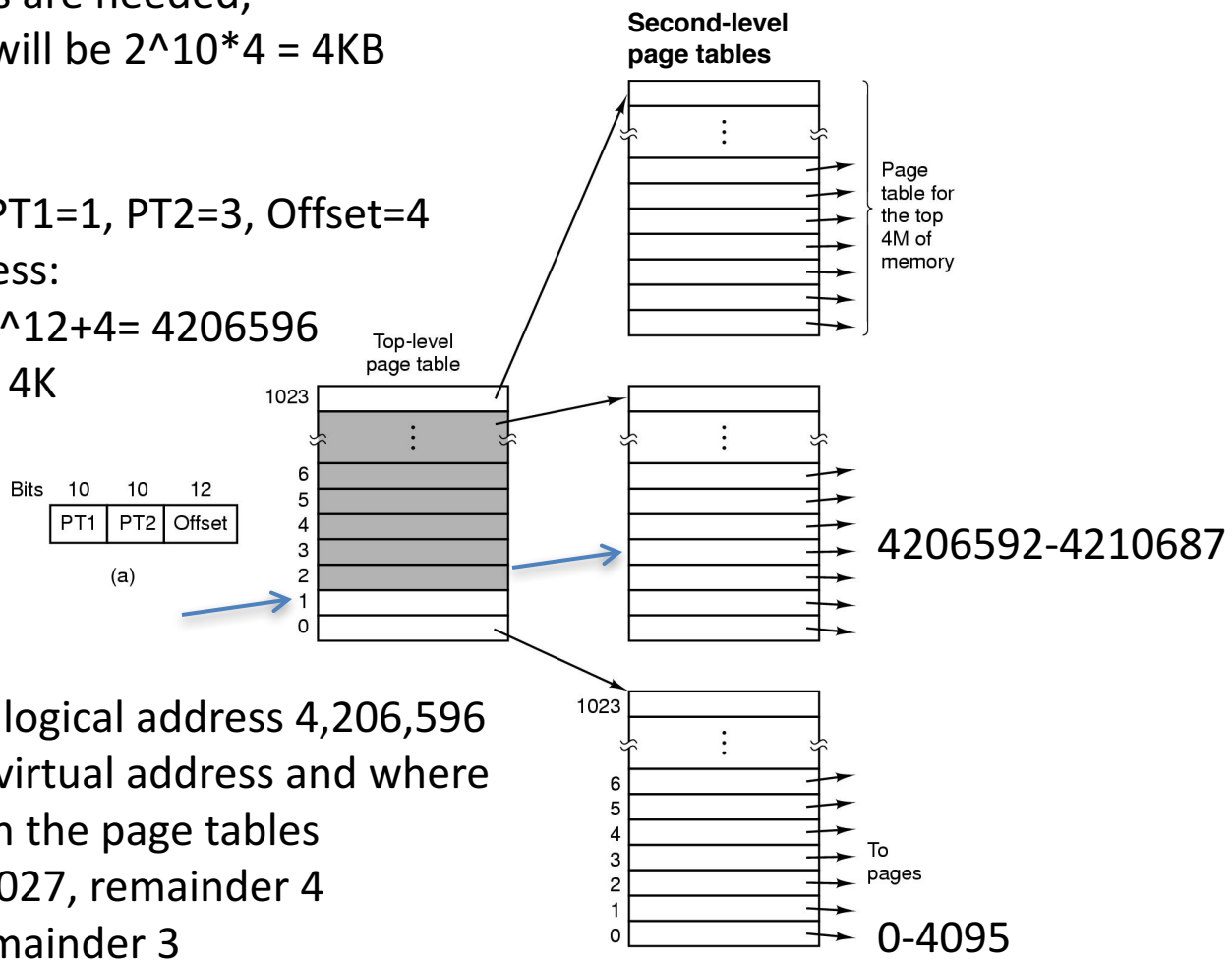
When we calculate Page Table size, the index itself (virtual page number) is often NOT included!

What if the virtual memory address is 64-bit? 2^64/2^12*2^2 = 2^24 GB

Multi-level Page Tables

If only 4 tables are needed,
the total size will be 2^10*4 = 4KB

Example-1: PT1=1, PT2=3, Offset=4
Virtual address:
1*2^22+3*2^12+4= 4206596
4M 4K



Example-2: given logical address 4,206,596
What will be the virtual address and where are its positions in the page tables
4206596/4096=1027, remainder 4
1027/1024=1, remainder 3

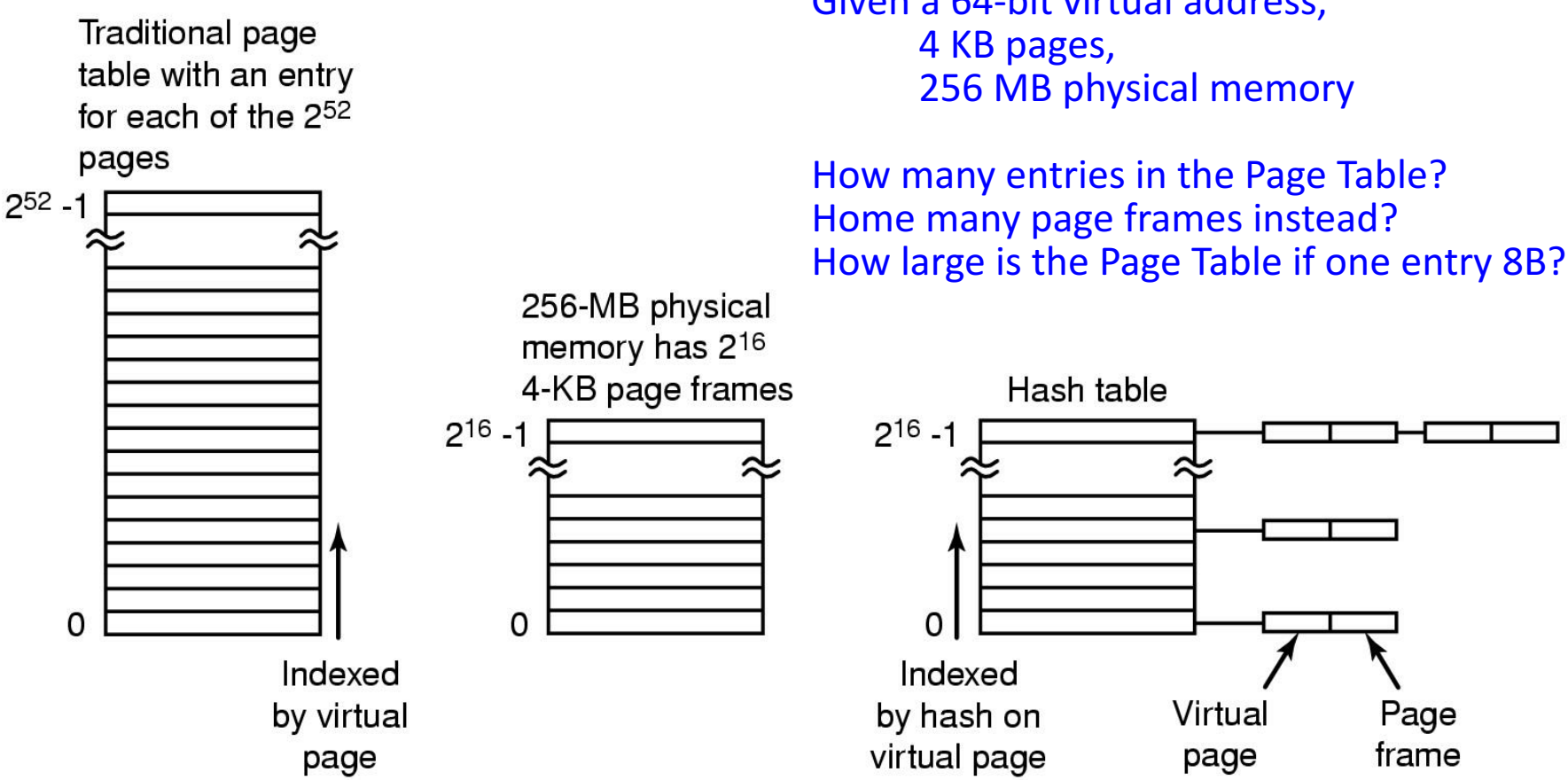
(a)32 bit address with 2 page table fields. (b)Two-level page tables

Inverted Page Tables

- Inverted page table: one entry per page frame in physical memory, instead of one entry per page of virtual address space.

Given a 64-bit virtual address,
4 KB pages,
256 MB physical memory

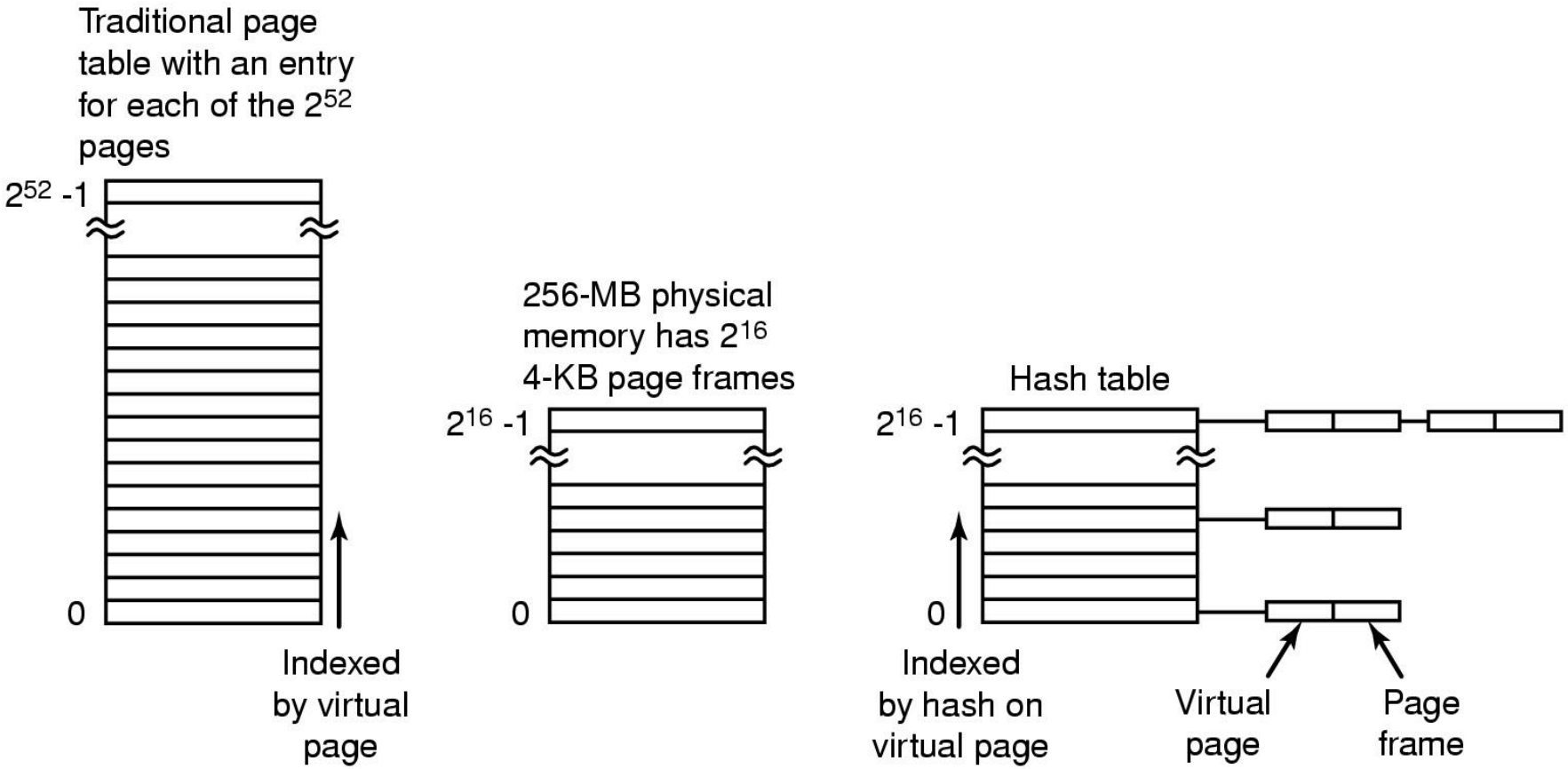
How many entries in the Page Table?
Home many page frames instead?
How large is the Page Table if one entry 8B?



Comparison of a traditional page table with an inverted page table

Inverted Page Tables (2)

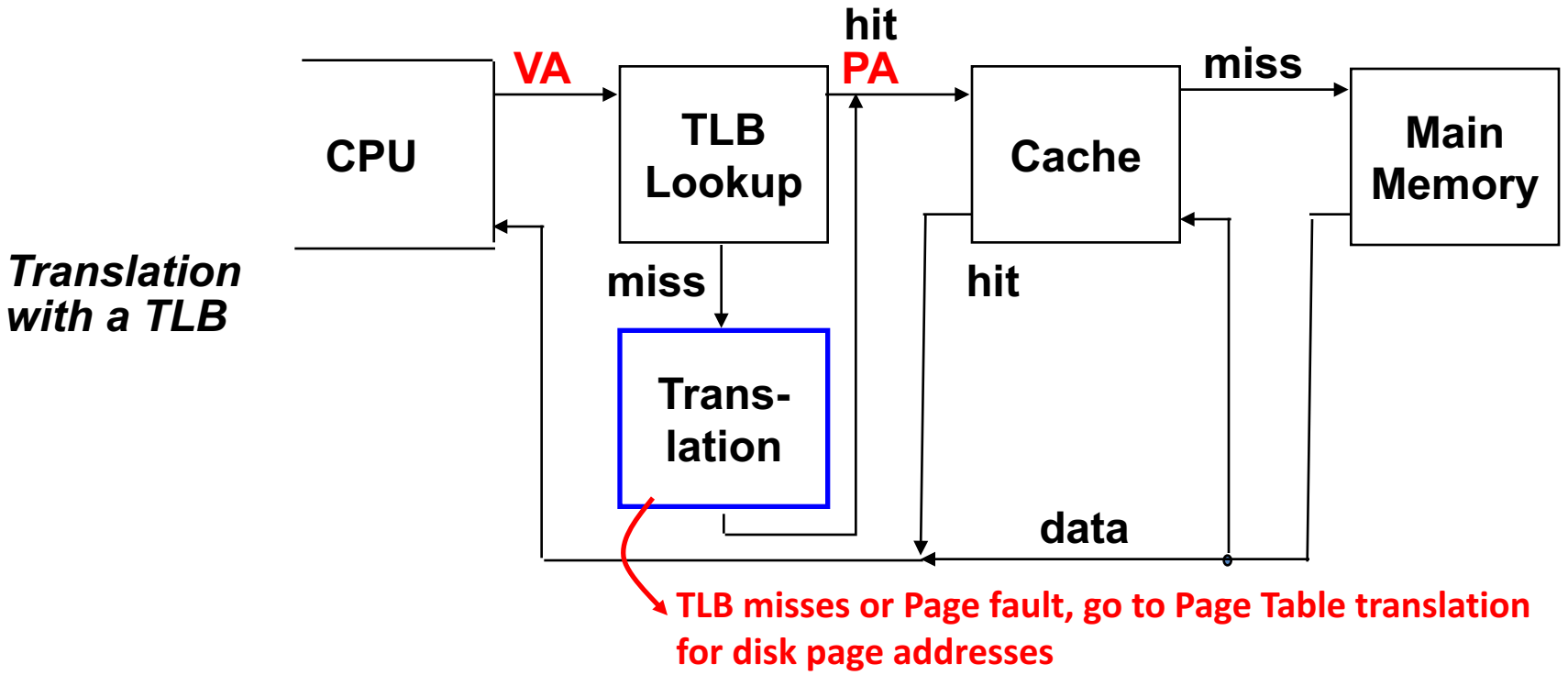
- Inverted page table: how to execute virtual-to-physical translation?
  - TLB helps! But what if a TLB misses?



Integrating TLB, Cache, and VM

Just like any other cache, the TLB can be organized as fully associative, set associative, or direct mapped

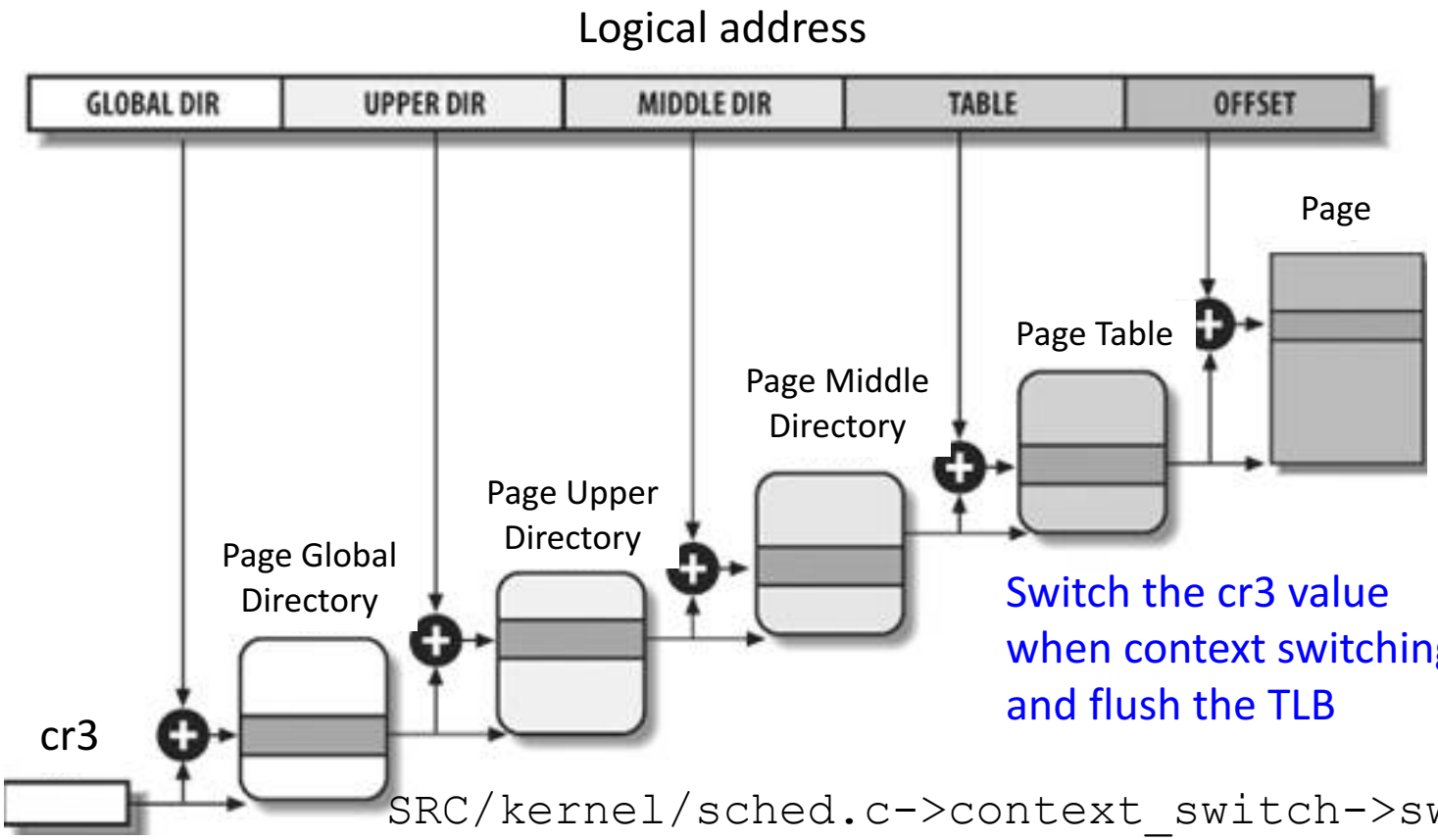
TLBs are usually small, typically not more than 128 - 256 entries even on high end machines. This permits fully associative lookup on these machines. Most mid-range machines use small n-way set associative organizations.



Put it all together: Linux and X86

A common model for 32-bit (two-level, 4B pte) and 64-bit (four-level, 8B pte)

`SRC/include/linux/sched.h->task_struct->mm->pgd`



Switch the cr3 value
when context switching threads
and flush the TLB

`SRC/kernel/sched.c->context_switch->switch_mm`

Summary

- Two tasks of memory management
- Why memory abstraction?
- Manage free memory
- Two ways to deal with memory overload
  - Swapping and virtual memory
- Virtual memory
- Paging and Page table