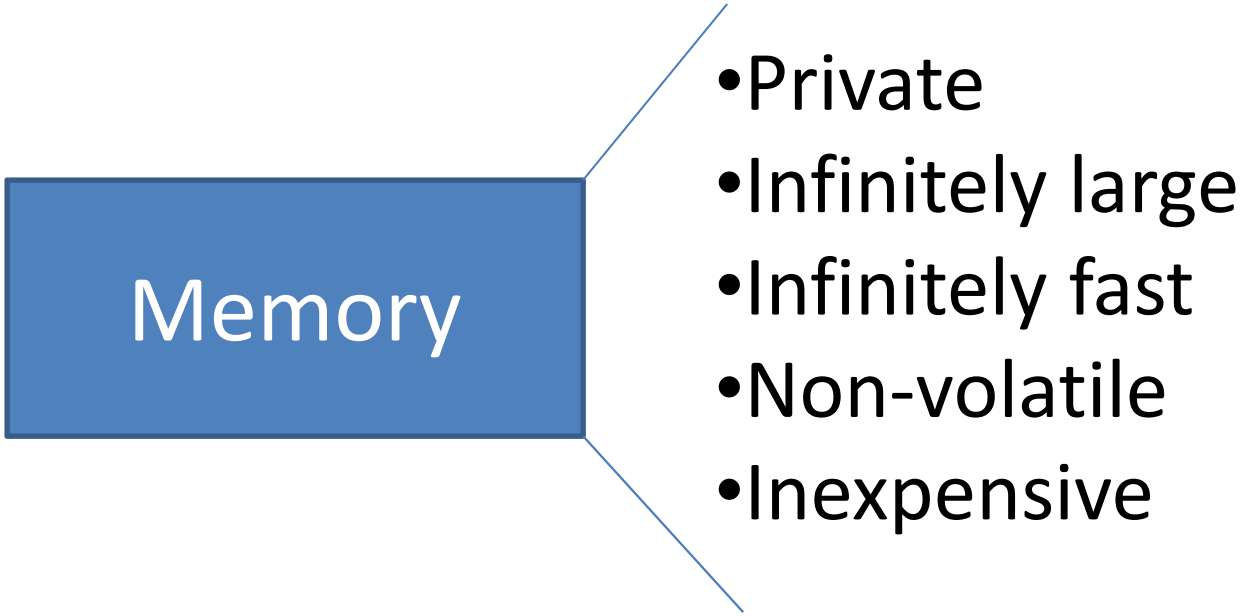
CSCI-GA.2250-001

Operating Systems

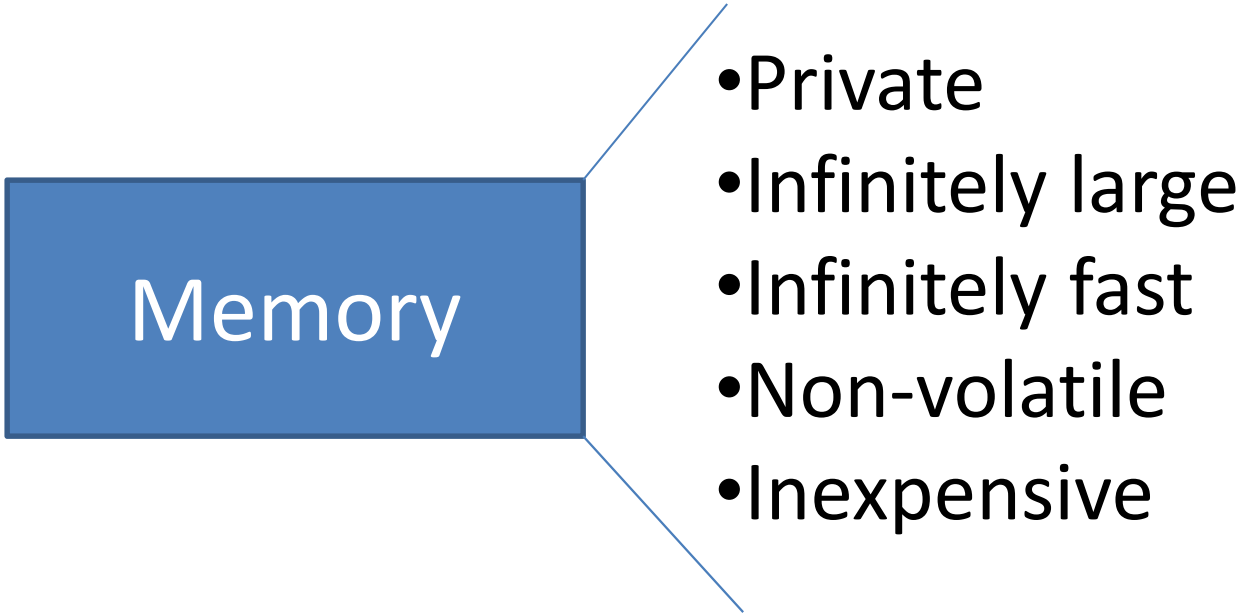
Lecture 5: Memory Management

Hubertus Franke
frankeh@cims.nyu.edu

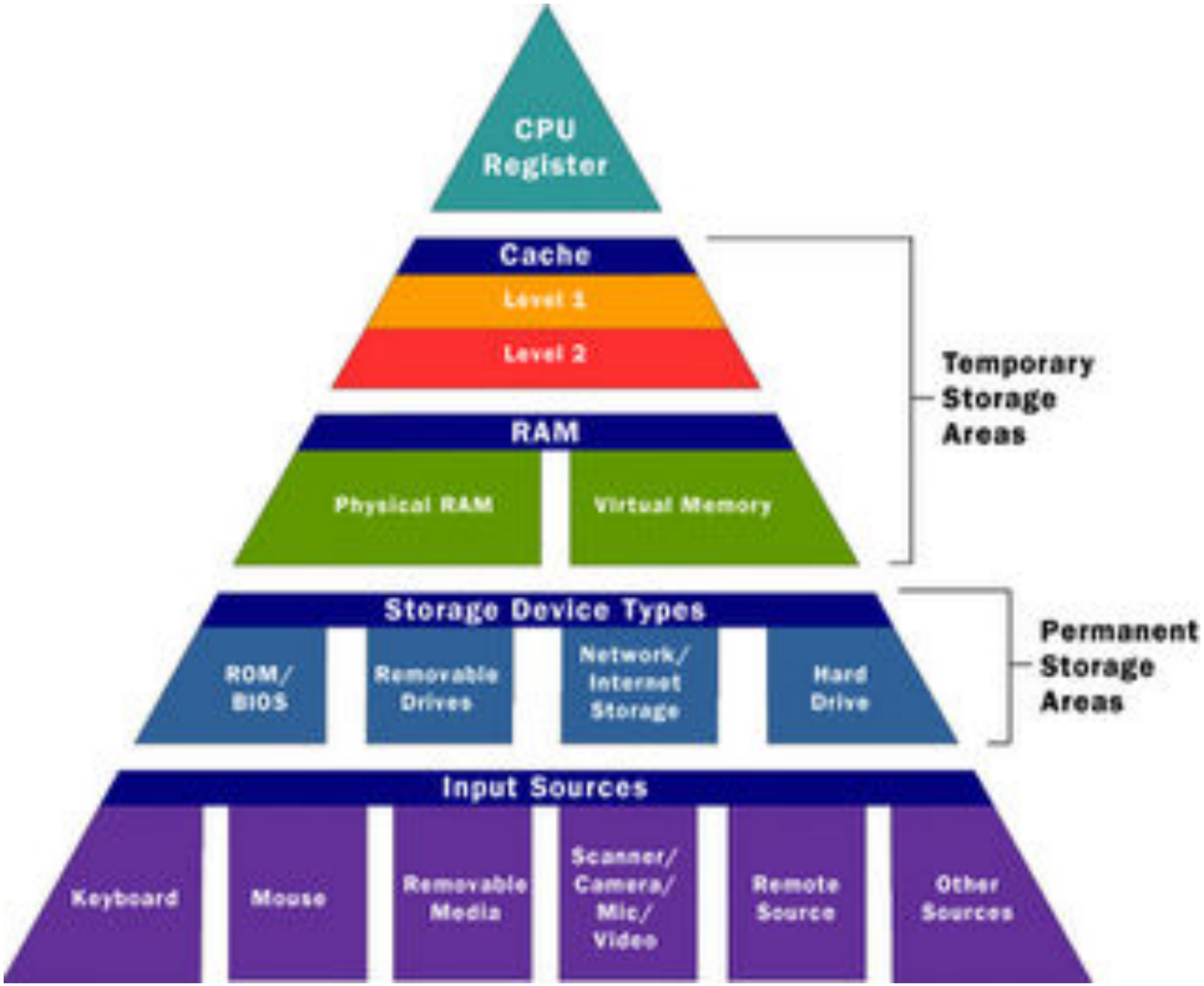
Programmer's dream



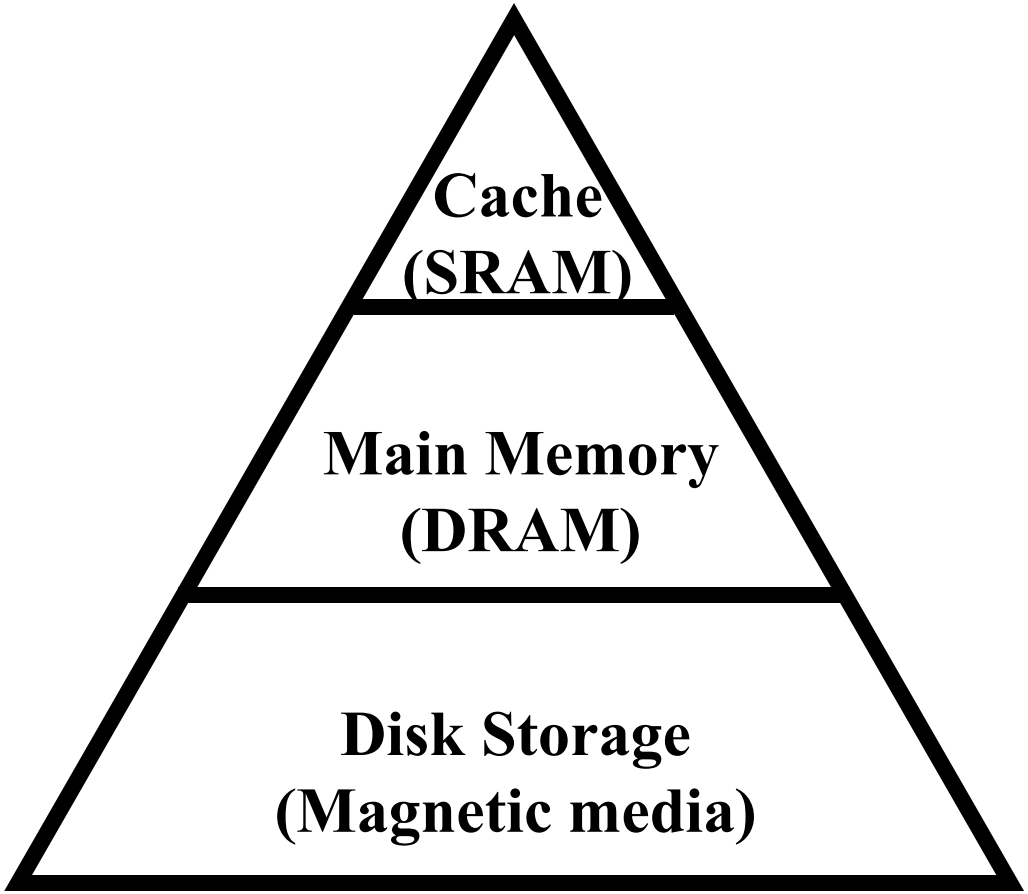
Programmer's Wish List



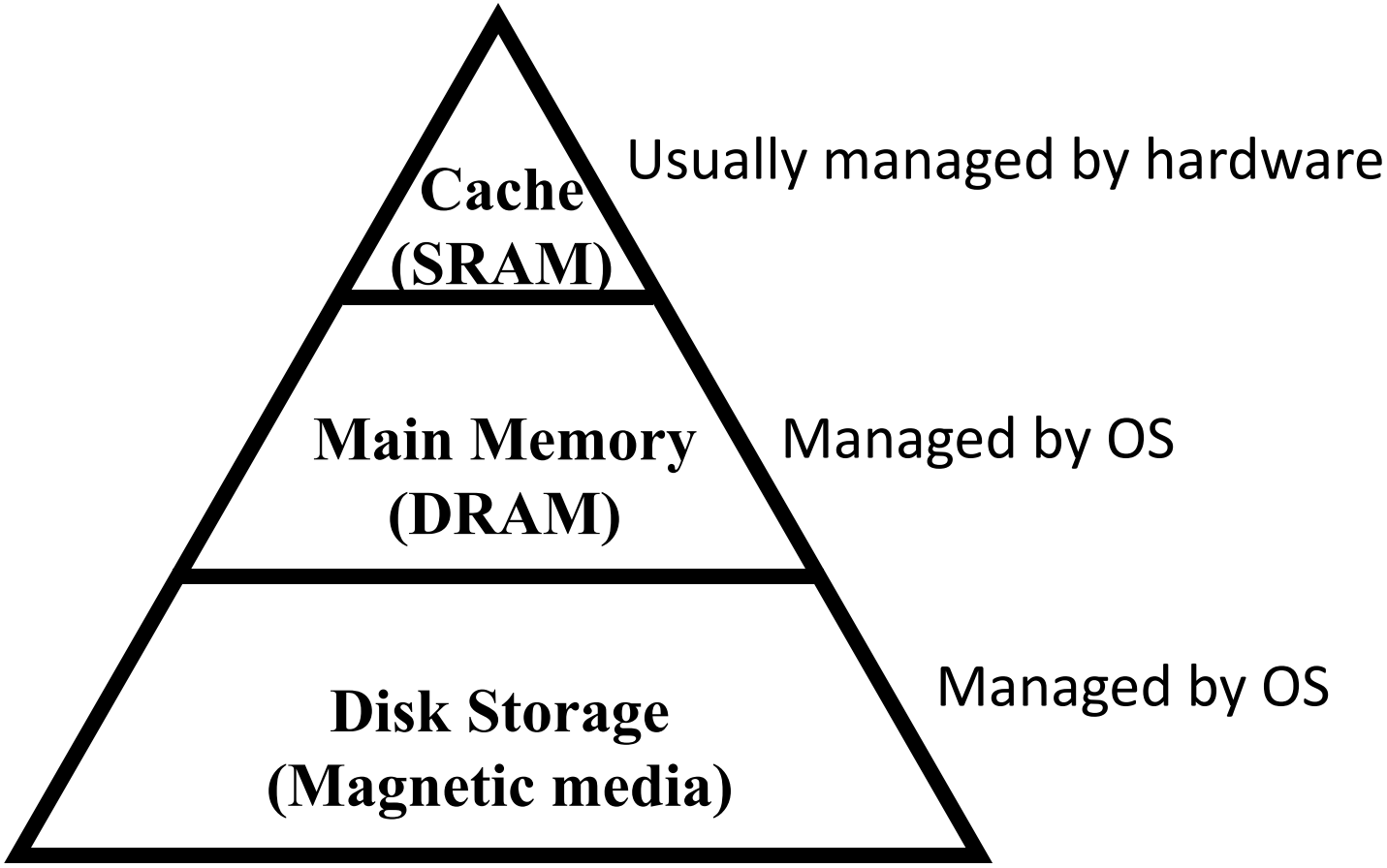
Programs are getting bigger faster than memories.


CPU
Register
Cache
Level 1
Level 2
RAM
Physical RAM
Virtual Memory
Temporary
Storage
Areas
Storage Device Types
ROM/
BIOS
Removable
Drives
Network/
Internet
Storage
Hard
Drive
Permanent
Storage
Areas
Input Sources
Keyboard
Mouse
Removable
Media
Scanner/
Camera/
Mic/
Video
Remote
Source
Other
Sources

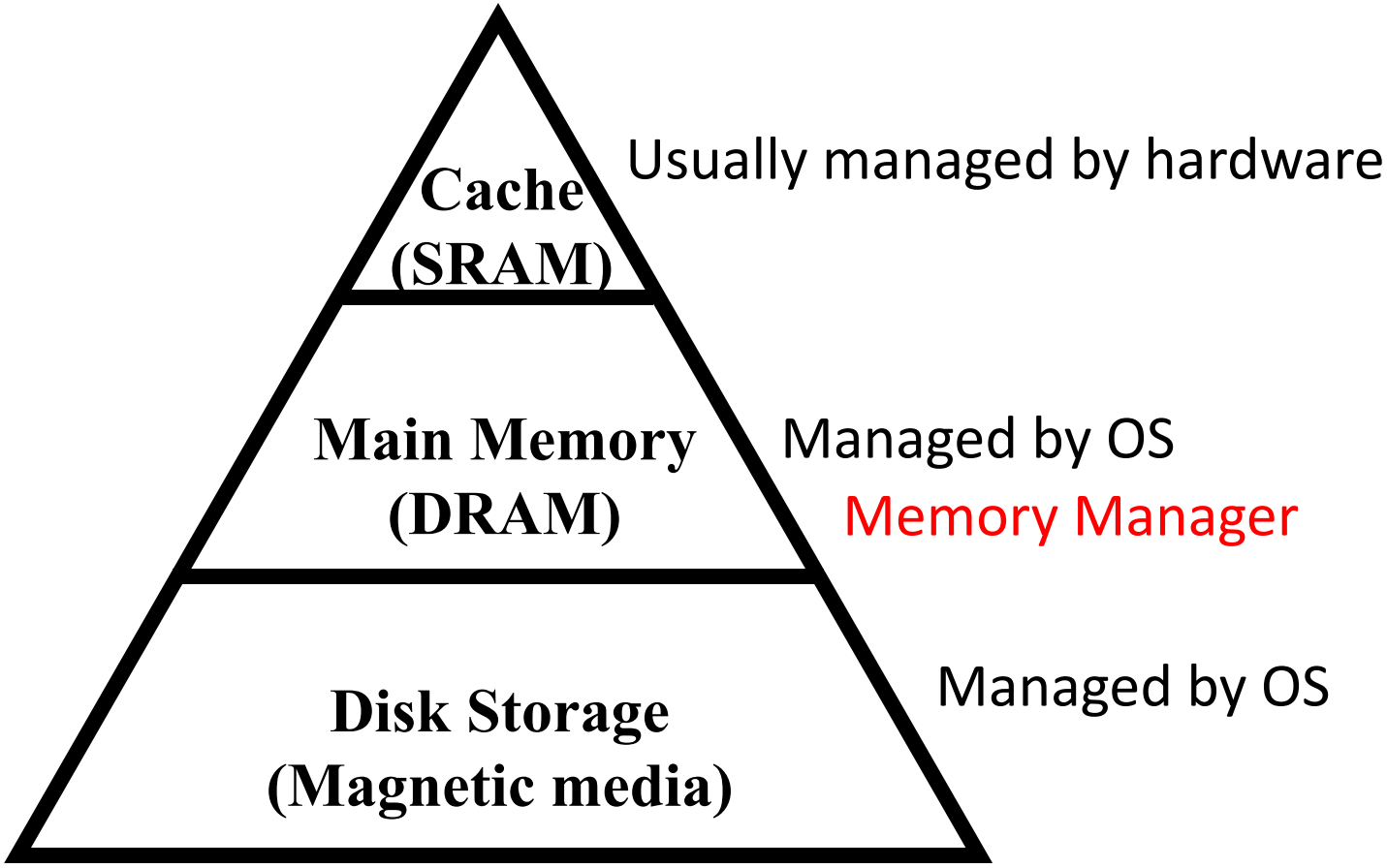
Memory Hierarchy



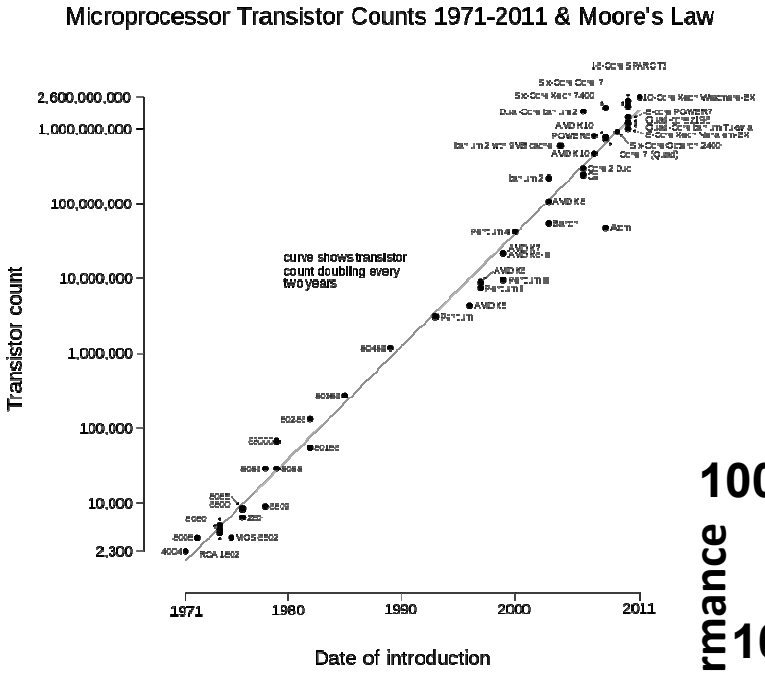
Memory Hierarchy



Memory Hierarchy



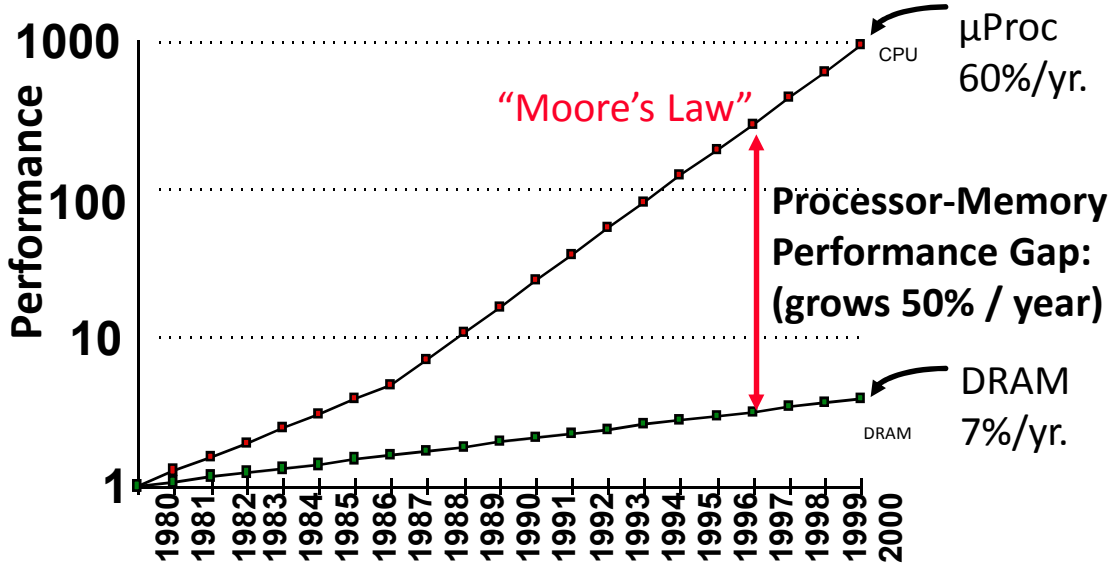
Question: Who Cares About the Memory Hierarchy?



CPU-DRAM Gap:

- Longer per cycle memory access times
- → Memory seems "further away"
- → DDR1/2/3/4/5 (technologies)

Still problem widens

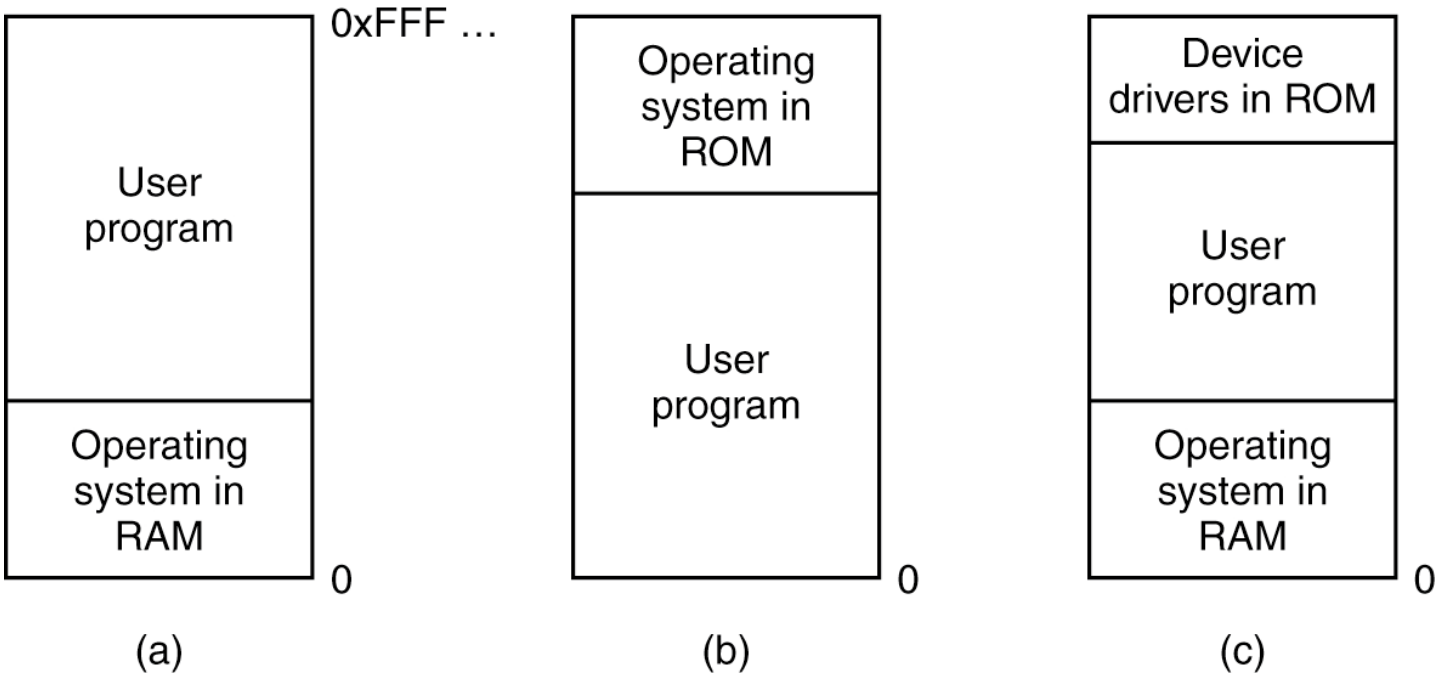


Memory Abstraction

- The hardware and OS memory manager makes you see the memory as a single contiguous entity
- How do they do that?
  - Abstraction

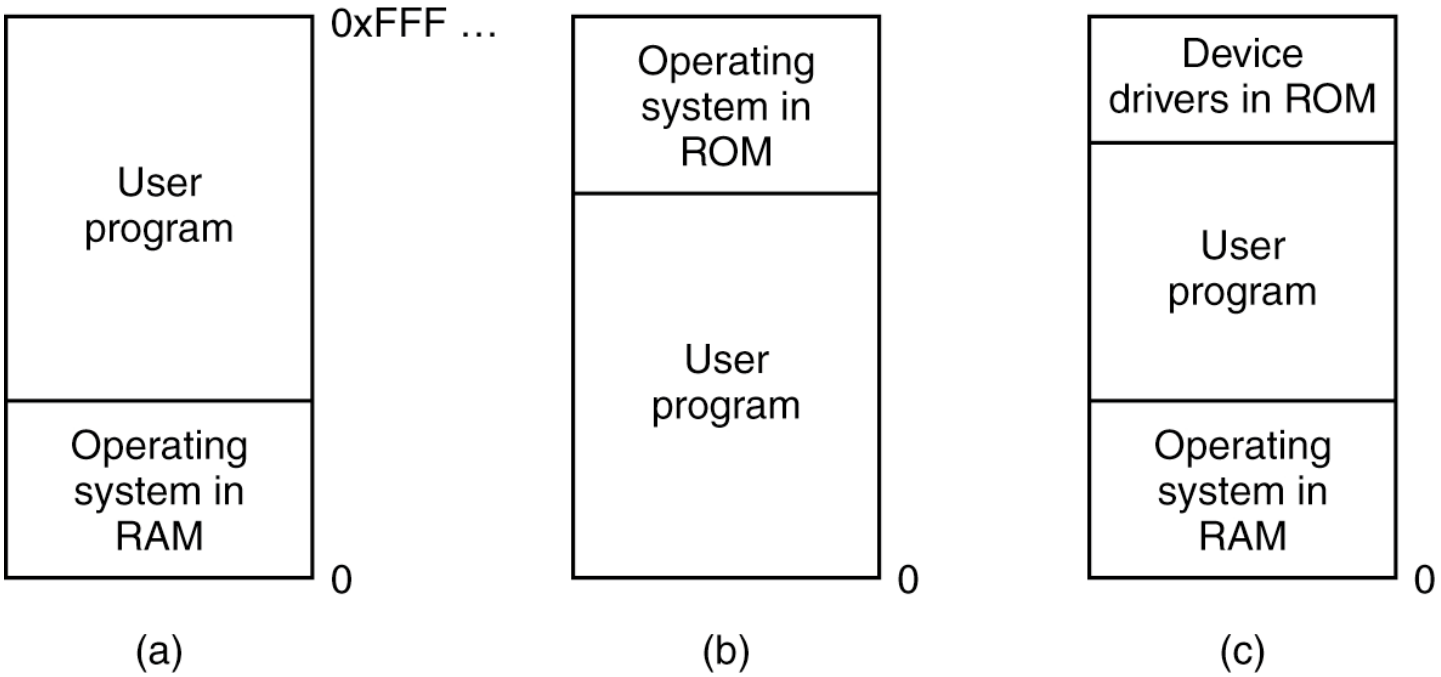
Is abstraction necessary?

No Memory Abstraction



Even with no abstraction, we can have several setups!

No Memory Abstraction



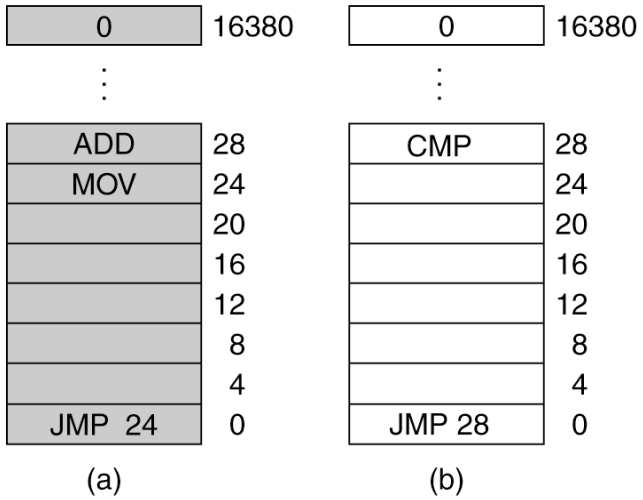
Only one process at a time can be running (threads??)

No Memory Abstraction

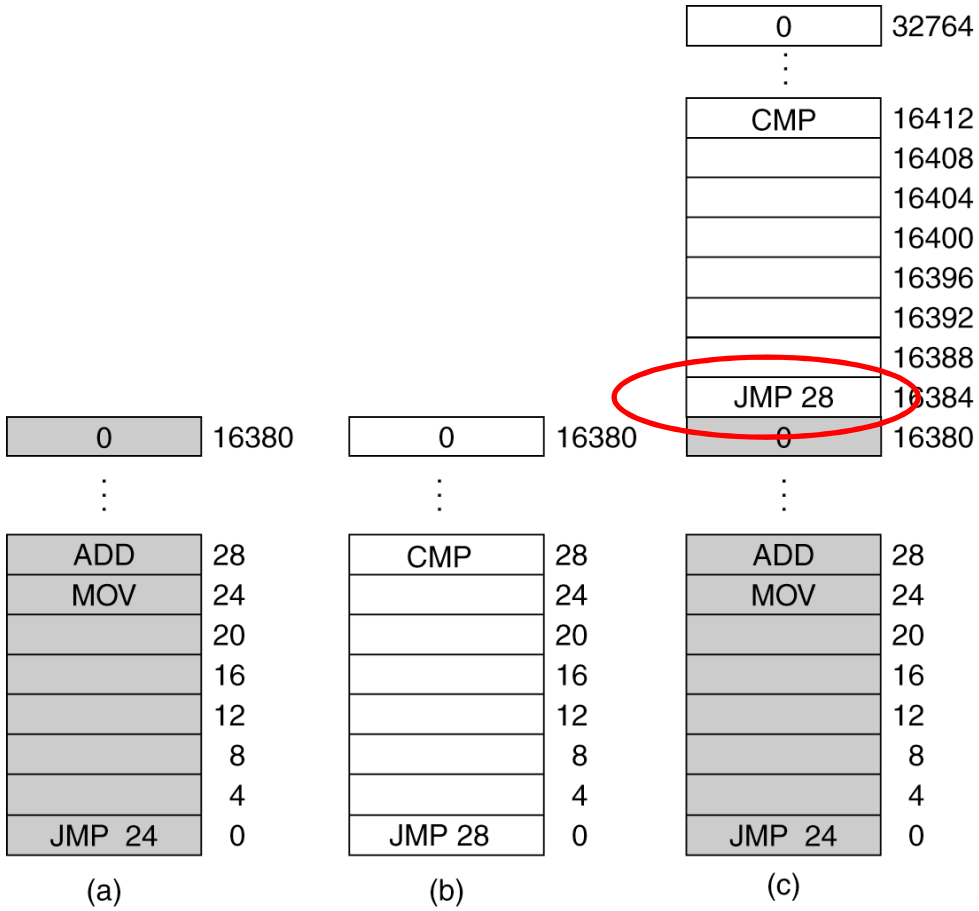
- What if we want to run multiple programs?
  - OS saves entire memory on disk
  - OS brings next program
  - OS runs next program

  } Swapping
- We can use swapping to run multiple programs concurrently
  - Memory divided into blocks
  - Each block assigned protection bits
  - Program status word contains the same bits
  - Hardware needs to support this
  - Example: IBM 360

No Memory Abstraction



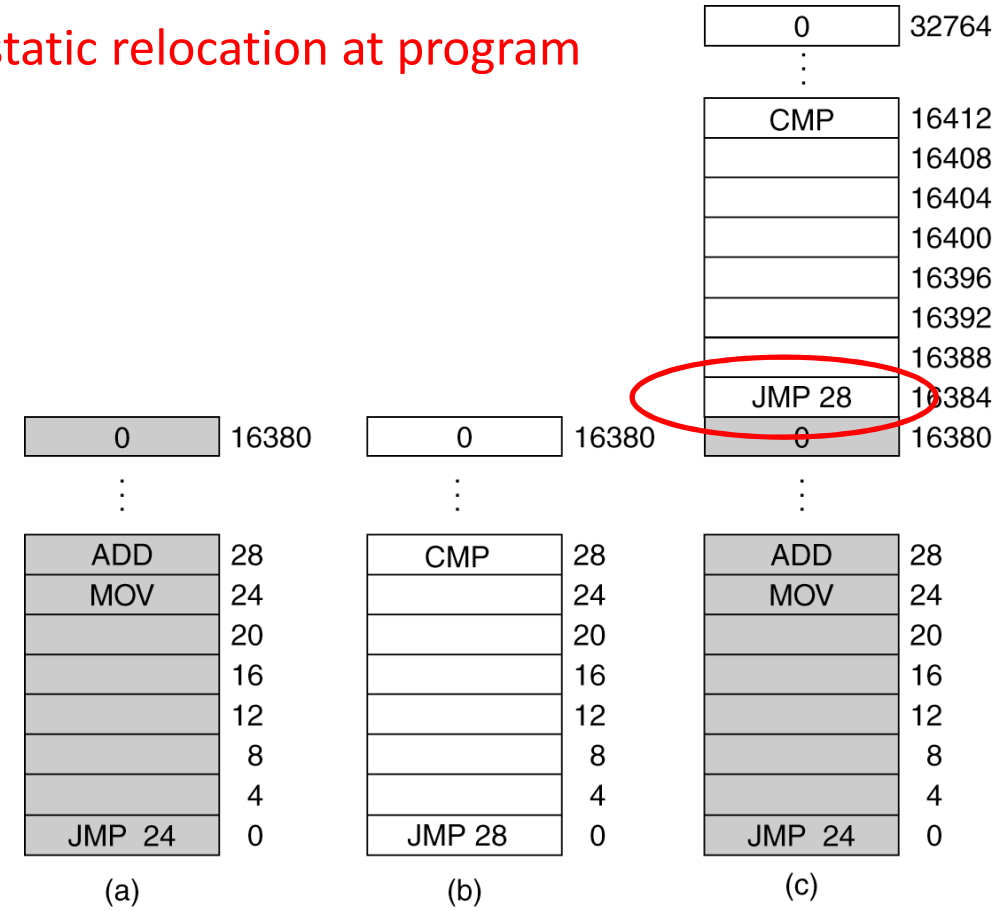
No Memory Abstraction



Using absolute address is wrong here

No Memory Abstraction

We can use static relocation at program load time



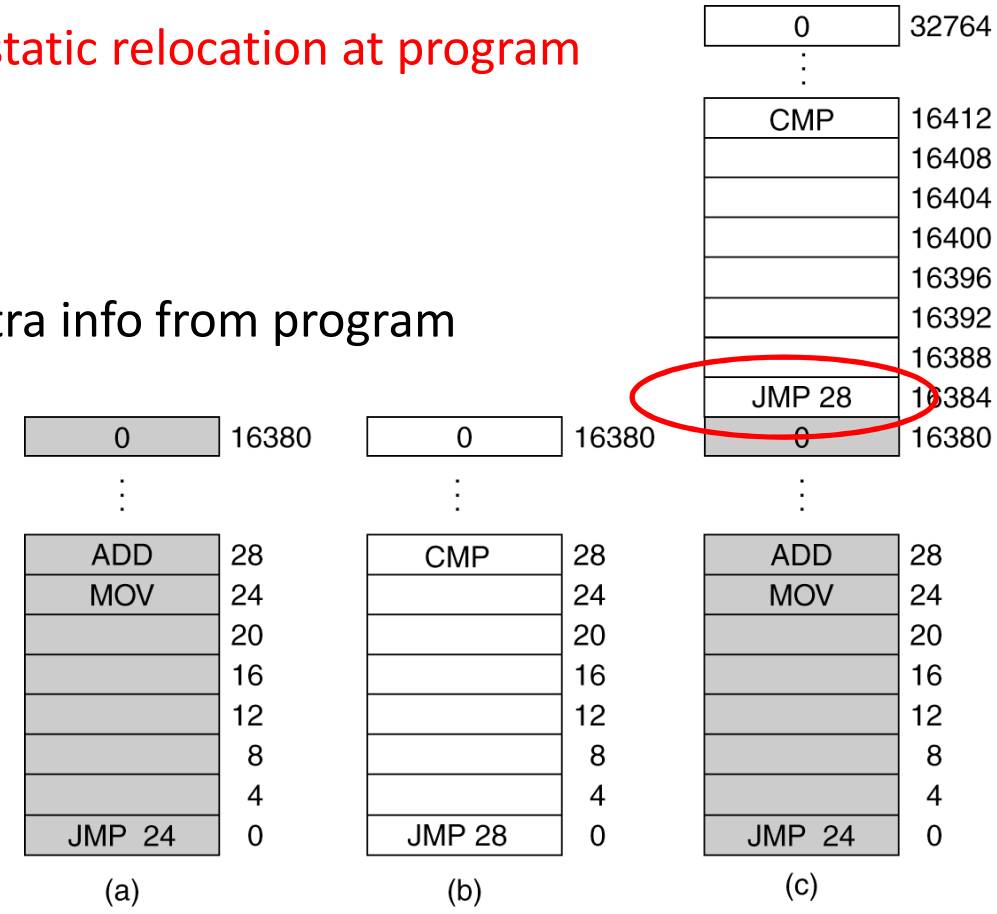
Using absolute address is wrong here

No Memory Abstraction

We can use static relocation at program load time

Bad Idea!

- Slow
- Require extra info from program



Using absolute address is wrong here

Bottom line: Memory abstraction is needed!

Memory Abstraction

- To allow several programs to co-exist in memory we need
  - Protection
  - Relocation
  - Sharing
  - Logical organization
  - Physical organization
- A new abstraction for memory: Address Space
- Address space = set of addresses that a process can use to address memory

Protection

- Processes need to acquire permission to reference memory locations for reading or writing purposes
- Location of a program in main memory is unpredictable
- Memory references generated by a process must be checked at run time

Relocation

- Programmers typically do not know in advance which other programs will be resident in main memory at the time of execution of their program
- Active processes need to be able to be swapped in and out of main memory in order to maximize processor utilization
- Specifying that a process must be placed in the same memory region when it is swapped back in would be limiting
  - may need to *relocate* the process to a different area of memory

Sharing

- It is advantageous to allow each process access to the same copy of the program rather than have their own separate copy
- Memory management must allow controlled access to shared areas of memory without compromising protection

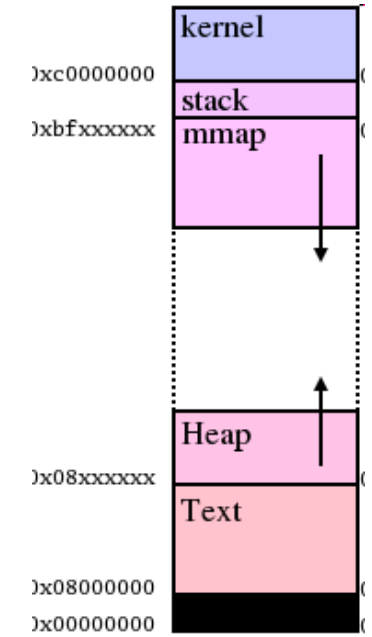
Logical Organization

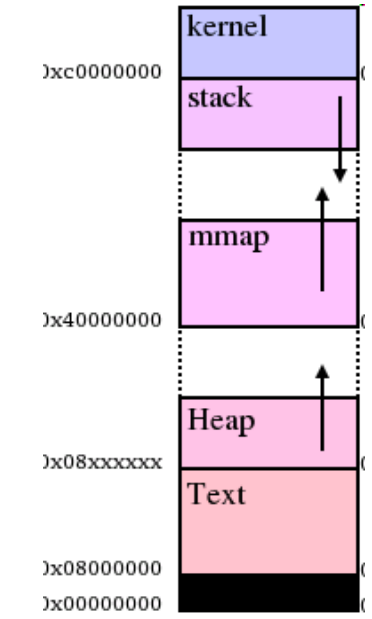
- We see memory as linear one-dimensional address space.
- A program = code + data + … = modules
- Those modules must be organized in that logical address space

Address Space

- Defines where sections of data and code are located in 32 or 64 address space
- Defines protection of such sections
- ReadOnly, ReadWrite, Execute

- Confined "private" addressing concept
  - → requires form of address virtualization





Physical Organization

- Memory is really a hierarchy
  - Several levels of caches
  - Main memory
  - Disk
- Managing the different modules of different programs in such a way as:
  - To give illusion of the logical organization
  - To make the best use of the above hierarchy

Address Space: Base and Limit

- Map each process address space onto a different part of physical memory
- Two registers: Base and Limit
  - **Base**: start address of a program in physical memory
  - **Limit**: length of the program
- For every memory access
  - Base is added to the address
  - Result compared to Limit
- Only OS can modify Base and Limit

Address Space: Base and Limit

Main drawback:

Need to add and compare for each memory address

What if memory space is not enough for all programs?

Address Space: Base and Limit

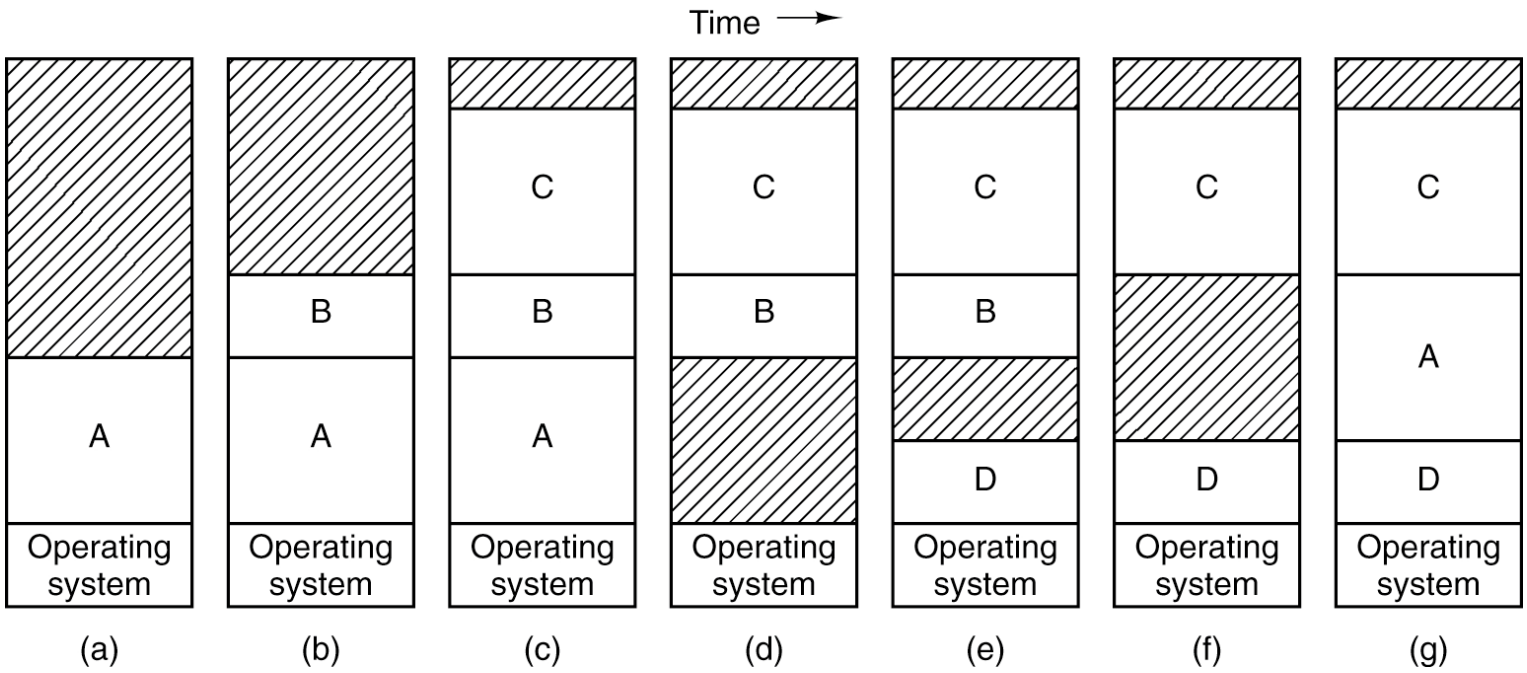
Main drawback:

Need to add and compare for each memory address

What if memory space is not enough for all programs?

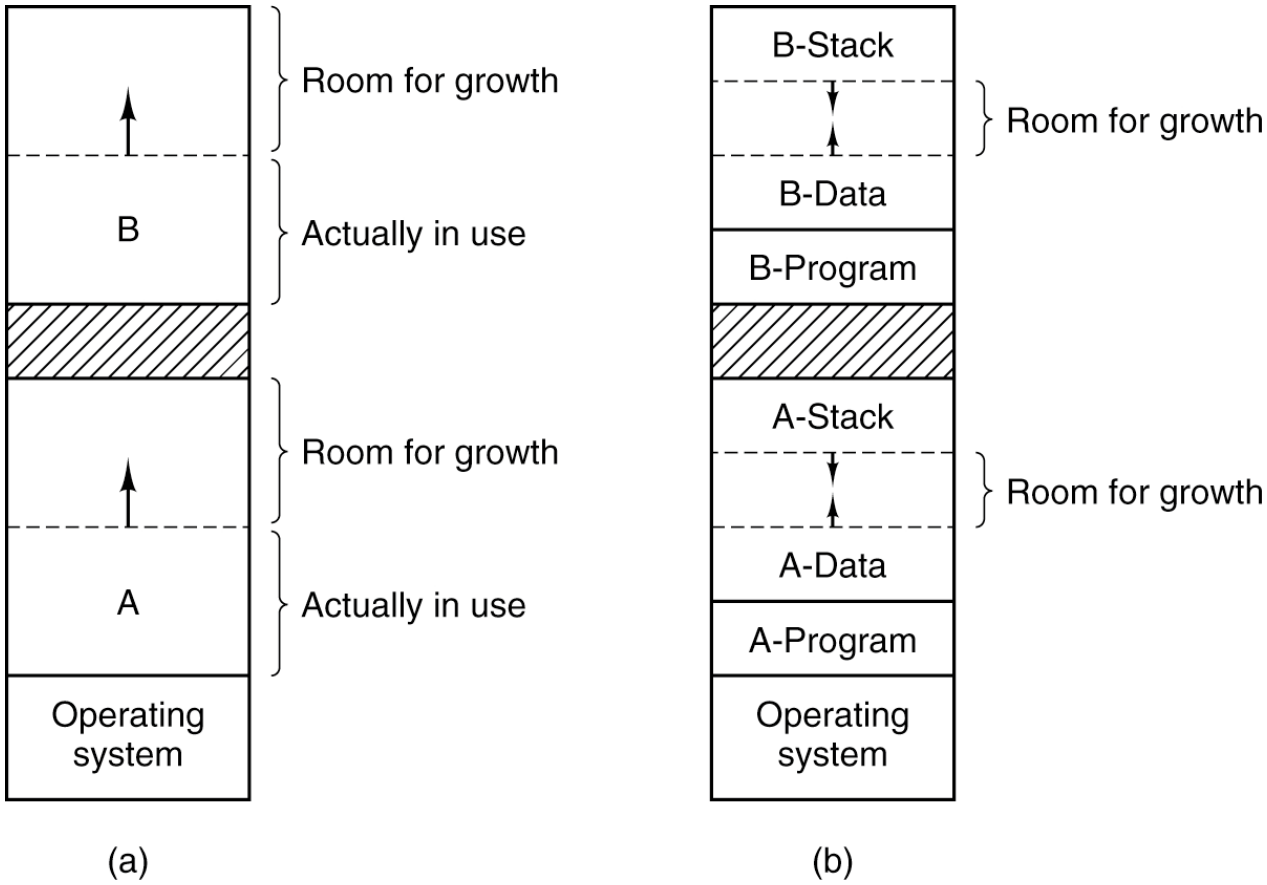
The we may need to swap some programs out of the memory.

Swapping

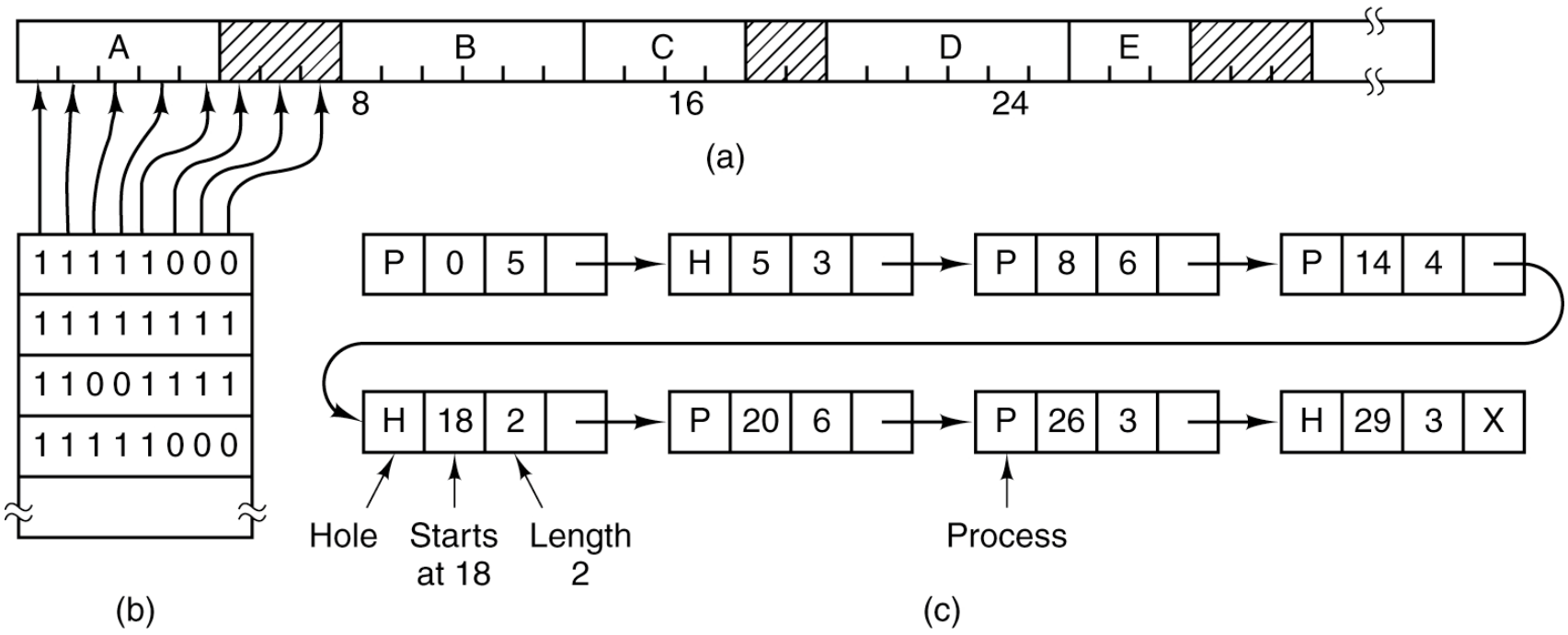


Swapping

- Programs move in and out of memory
- Holes are created
- Holes can be combined -> memory compaction
- What if a process needs more memory?
  - If a hole is adjacent to the process, it is allocated to it
  - Process has to be moved to a bigger hole
  - Process suspended till enough memory is there


Room for growth
B
Actually in use
Room for growth
A
Actually in use
Operating system
(a)
B-Stack
Room for growth
B-Data
B-Program
A-Stack
Room for growth
A-Data
A-Program
Operating system
(b)

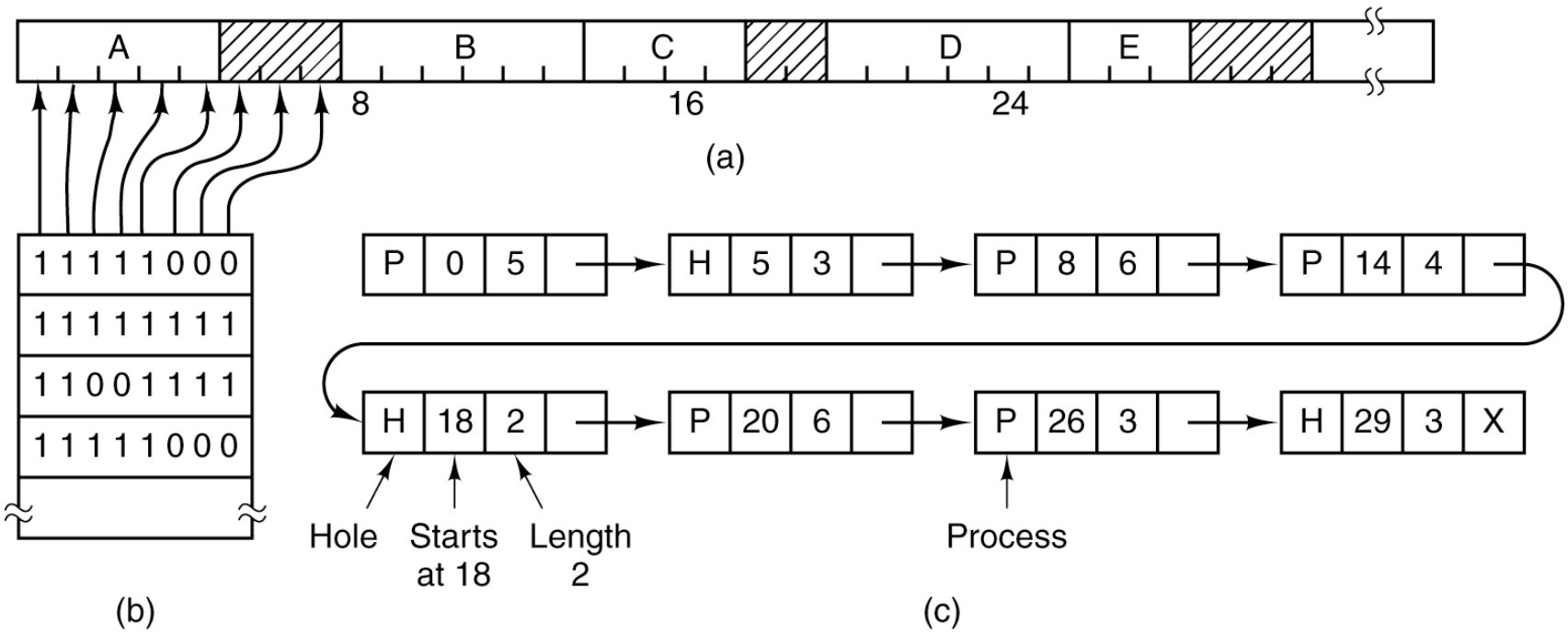
Managing Free Memory



Bitmap

Linked List

Managing Free Memory



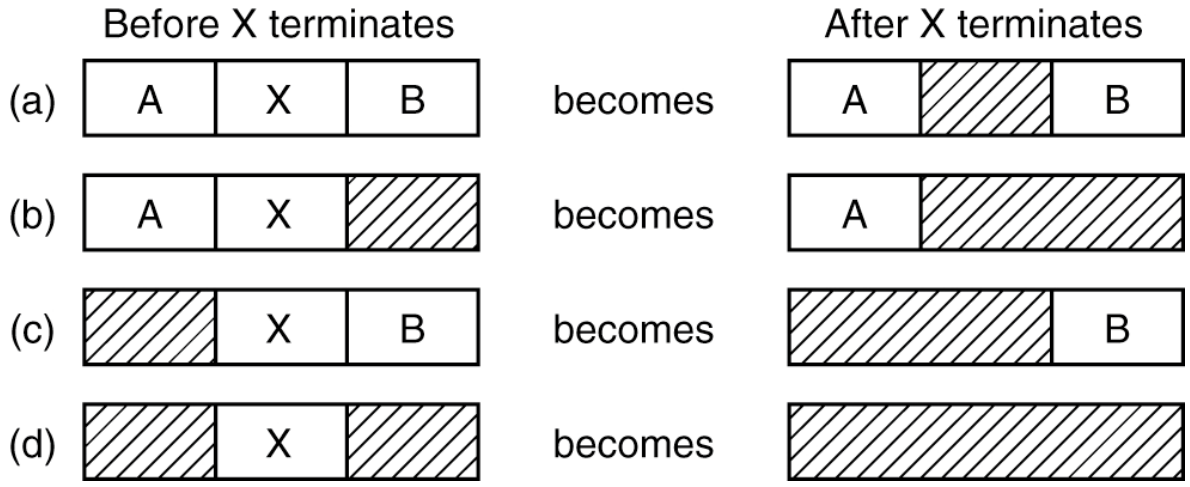
Bitmap

Linked List

Slow: To find k-consecutive 0s for a new process

Managing Free Memory: Linked List

- Linked list of allocated and free memory segments
- More convenient be double-linked list



Managing Free Memory: Linked List

- How to allocate?
  - First fit
  - Best fit
  - Next fit
  - Worst fit
  - …

Memory Management Techniques

- Memory management brings processes into main memory for execution by the processor
  - involves virtual memory
  - based on segmentation and paging

Mid - Conclusions

- Process is CPU abstraction
- Address space is memory abstraction
  - OS memory manager and the hardware helps providing this abstraction
- Two main tasks needed from OS regarding memory management:
  - managing free space
  - making best use of the memory hierarchy

What is the problem?

- Not enough memory
  - Have enough memory is not possible with current technology
    - How do you determine "enough"?
- Processor does not execute anything that is not in the memory.

But We Can See That …

- All memory references are logical addresses that are dynamically translated into physical addresses at run time
- A process may be broken up into a number of pieces that don't need to be contiguously located in main memory during execution.

So:

It is not necessary that all of the pieces of a process be in main memory during execution.

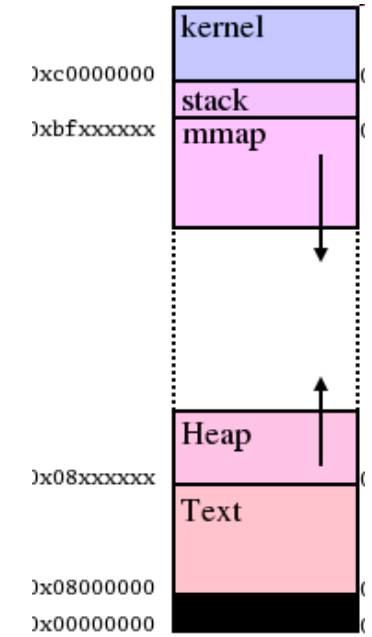
Scientific Definition of Virtual Memory

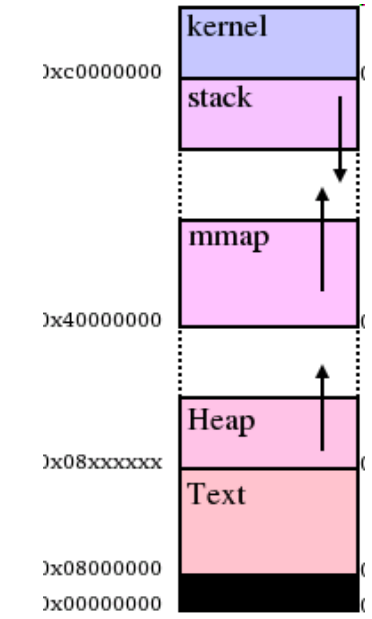
Mapping from logical (virtual) address space to physical address space

Address Space (reminder)

- Defines where sections of data and code are located in 32 or 64 address space
- Defines protection of such sections
- ReadOnly, ReadWrite, Execute

- Confined "private" addressing concept
  - → requires form of address virtualization





The Story

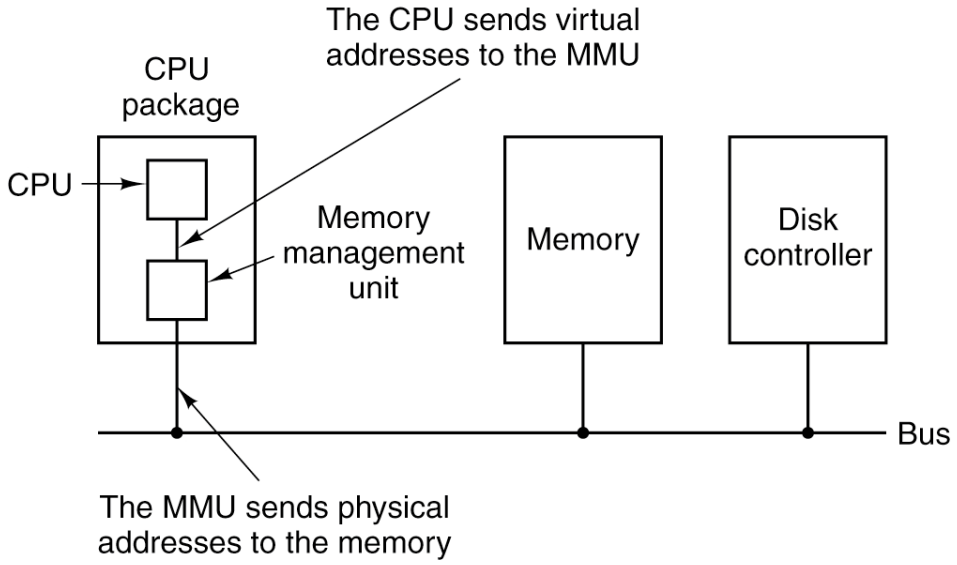
1. Operating system brings into main memory a few pieces of the program.
2. An interrupt is generated when an address is needed that is not in main memory.
3. Operating system places the process in a blocking state.
4. Operating system issues a disk I/O Read request.
5. Another process is dispatched to run while the disk I/O takes place.
6. An interrupt is issued when disk I/O is complete, which causes the operating system to place the affected process in the Ready state
7. Piece of process that contains the logical address is brought into main memory.

The Story

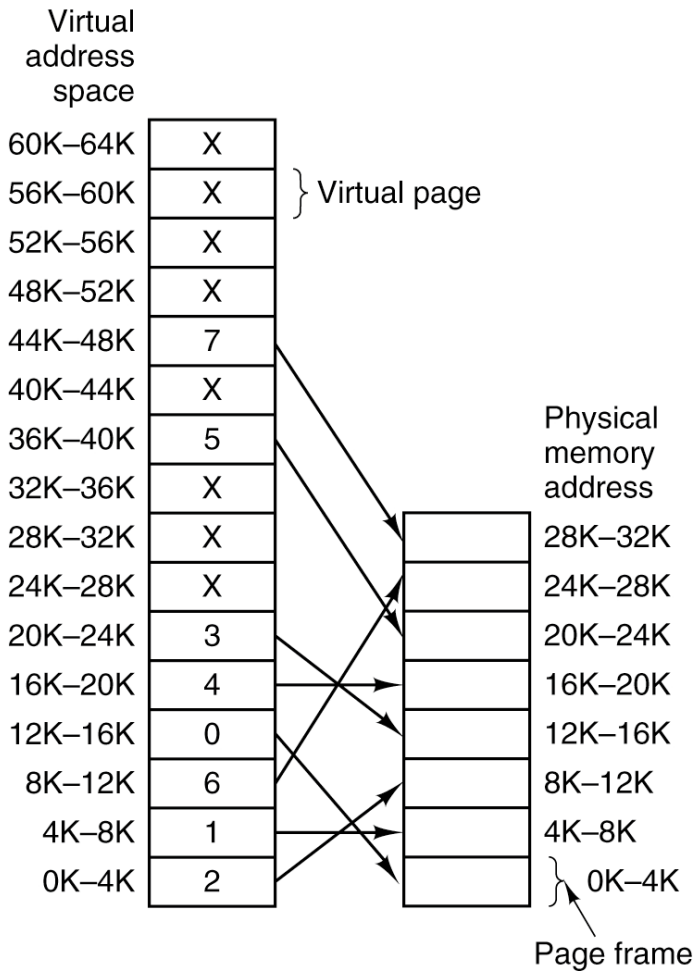
1. Operating system brings into main memory a few pieces of the program. What do you mean by "pieces"?
2. An interrupt is generated when an address is needed that is not in main memory. How do you know it isn't in memory?
3. Operating system places the process in a blocking state.
4. Operating system issues a disk I/O Read request. Why?
5. Another process is dispatched to run while the disk I/O takes place.
6. An interrupt is issued when disk I/O is complete, which causes the operating system to place the affected process in the Ready state
7. Piece of process that contains the logical address is brought into main memory. What if memory is full?

Virtual Memory

- Each program has its own address space
- This address space is divided into pages
- Pages are mapped into physical memory

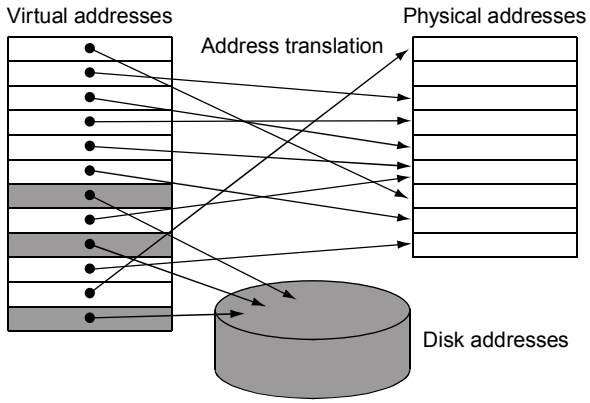


Virtual Memory

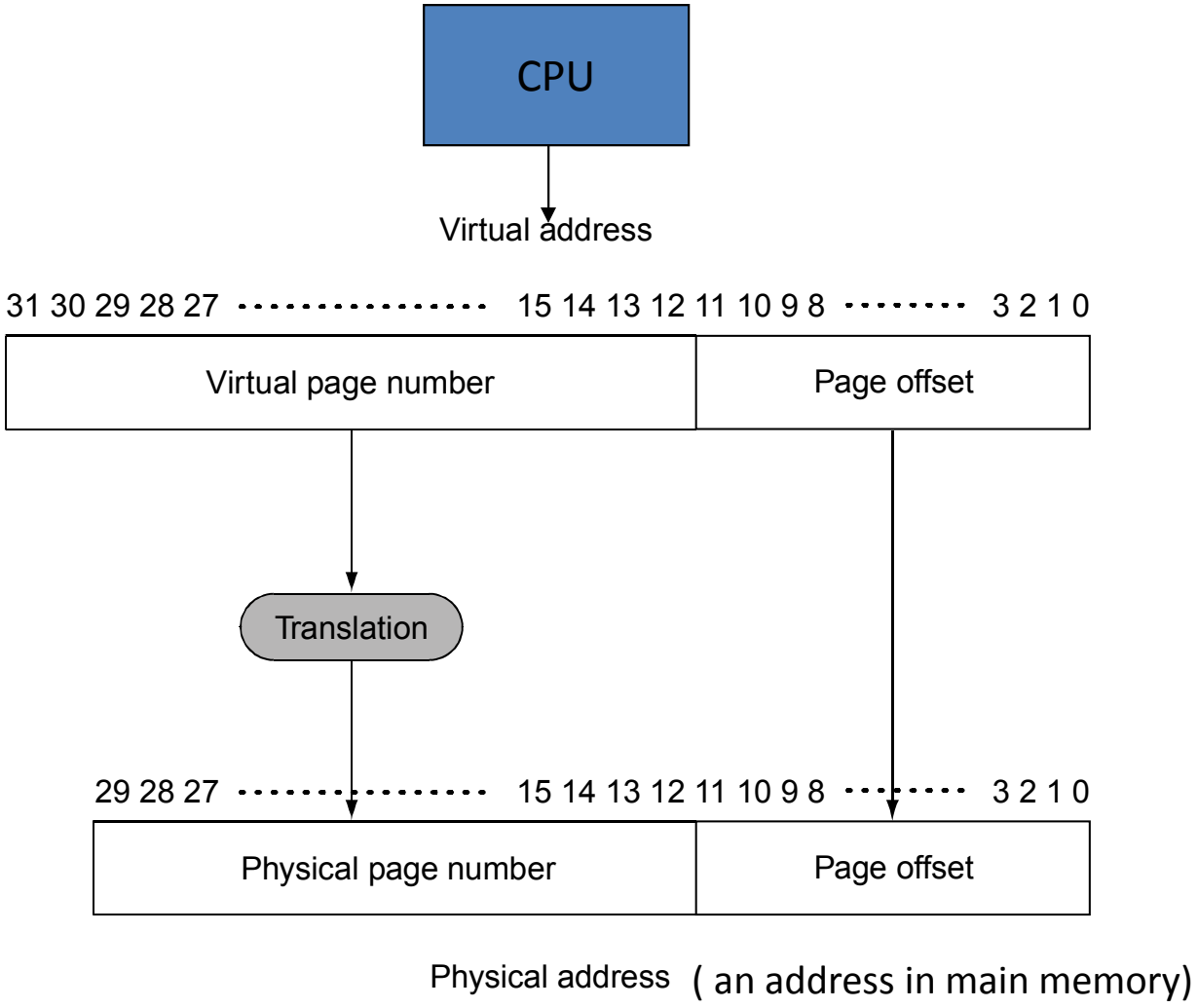


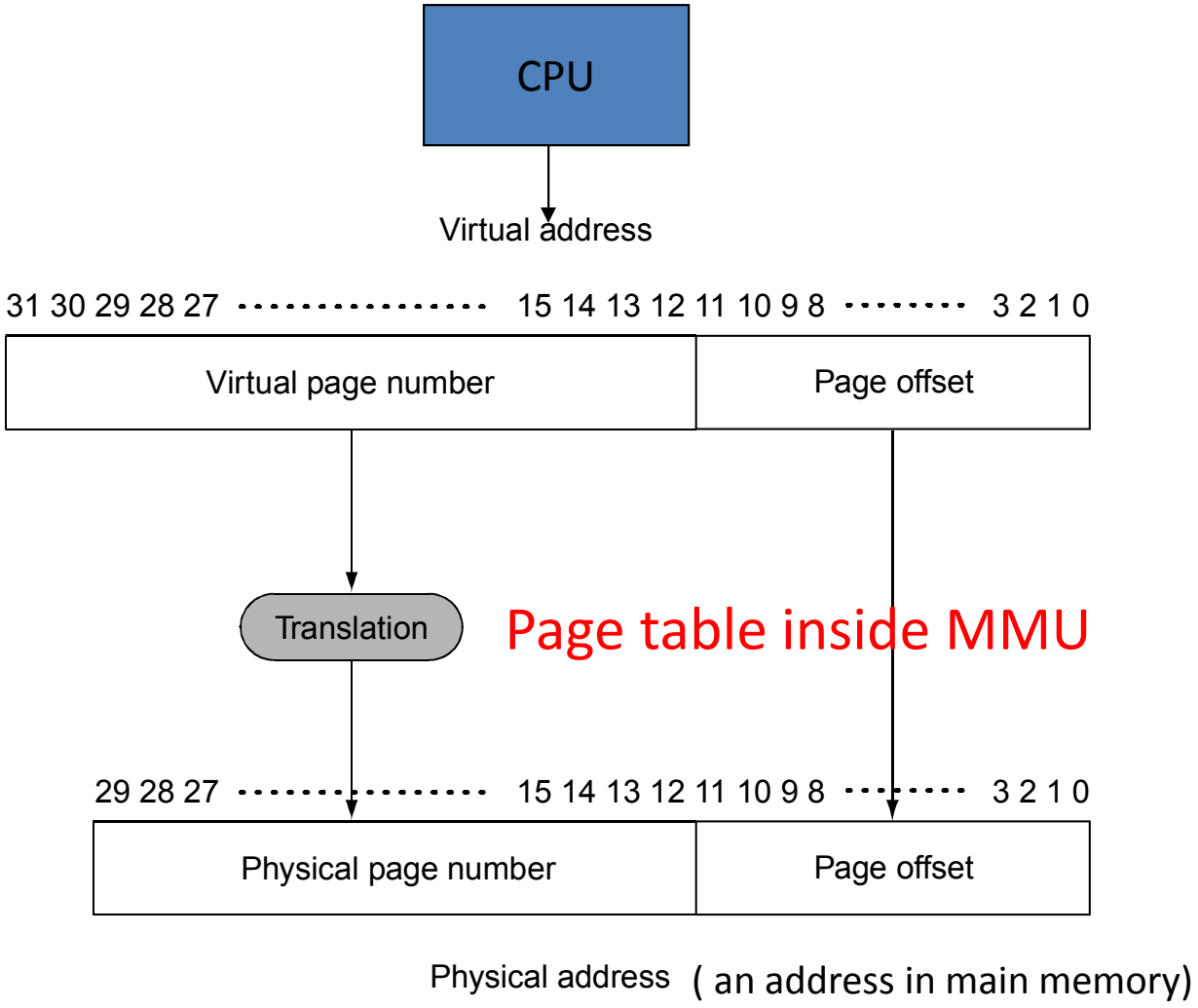
Virtual Memory

- Main memory can act as a cache for the secondary storage (disk)



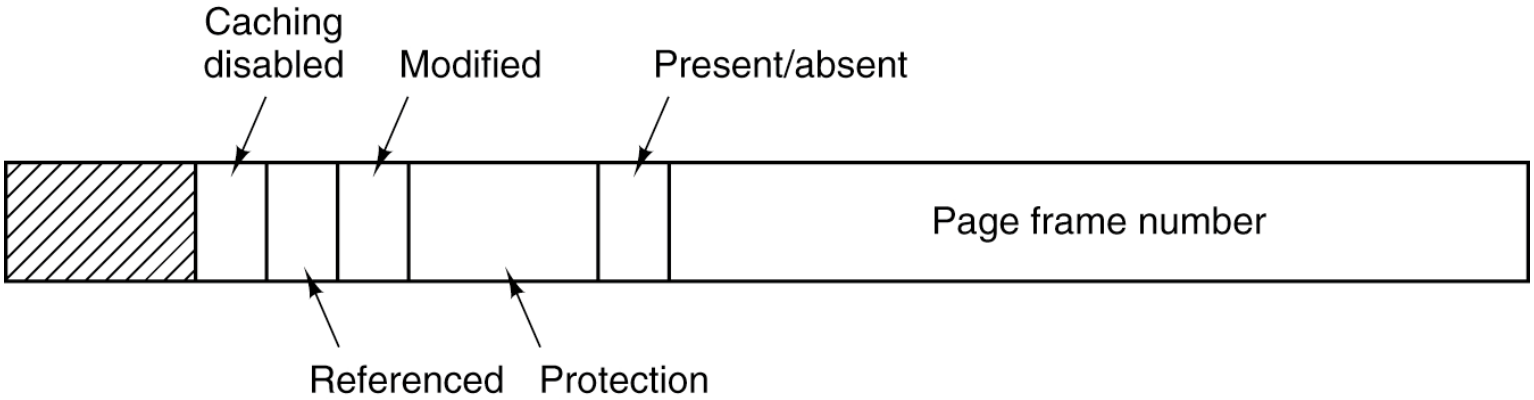
- Advantages:
  - illusion of having more physical memory
  - program relocation
  - protection


CPU
Virtual address
31 30 29 28 27 ..... 15 14 13 12 11 10 9 8 ..... 3 2 1 0
Virtual page number
Page offset
Translation
29 28 27 ..... 15 14 13 12 11 10 9 8 ..... 3 2 1 0
Physical page number
Page offset
Physical address (an address in main memory)



MMU = Memory Management Unit

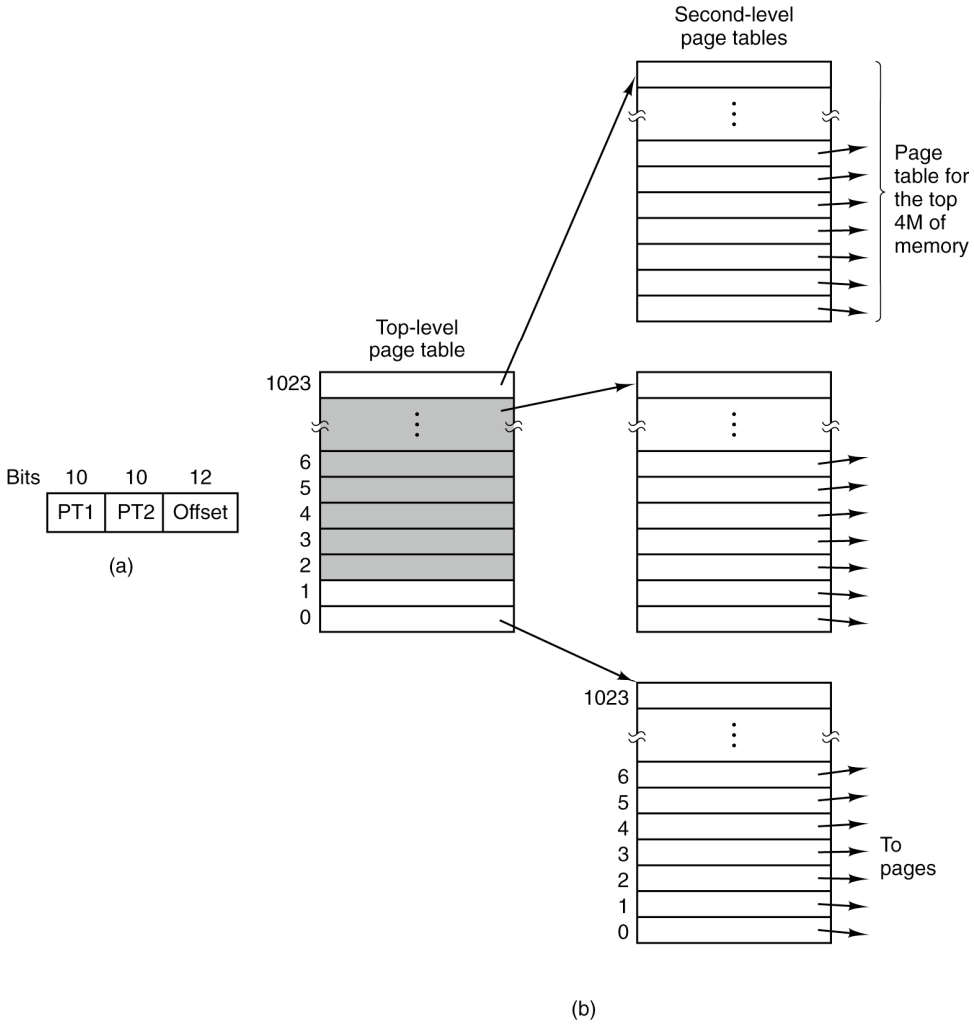
Structure of a Page Table Entry



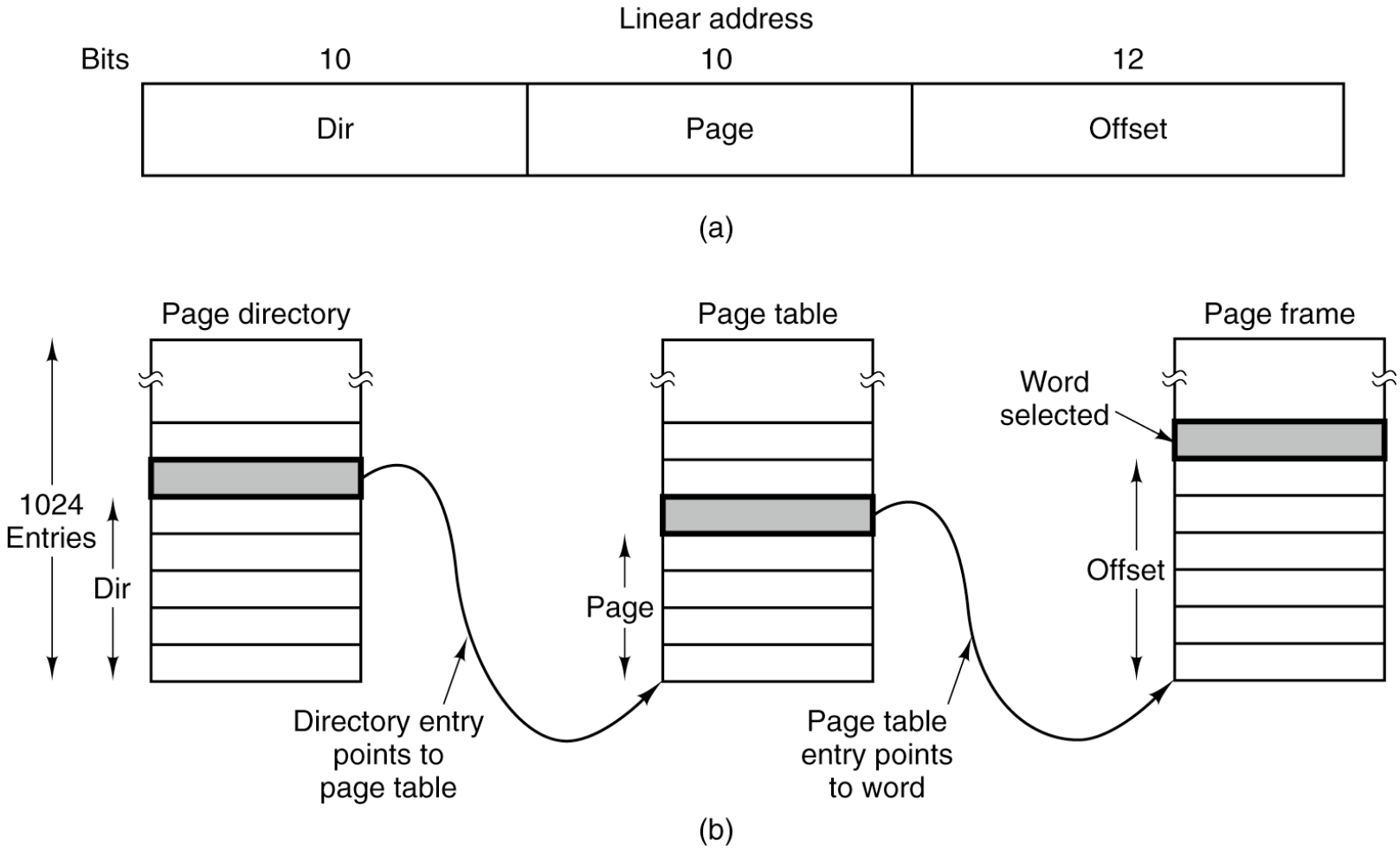
- **Present bit**: '1' if the values in this entry is valid, otherwise translation is invalid and an pagefault exception will be raised
- **Protection bits**: 'kernel' + 'w' specifies who and what can be done on this page
  if *kernel bit* is set then only the kernel can translate this page. If user accesses the page a 'privilege exception' will be raised.
  If *writeprotect bit* is set the page can only be read (load instruction). If attempted write (store instruction), a write protection exception is raised.
- **Reference bit:** every time the page is accessed (load or store), the reference bit is set.
- **Modified bit:** every time the page is written to (store), the modified bit is set.
- **Caching Disabled:** required to access I/O devices, otherwise their content is in cpu cache
- **Frame Number:** this is the physical frame that is accessed based on the translation.

Multi-Level Page Table

- To reduce storage overhead in case of large memories

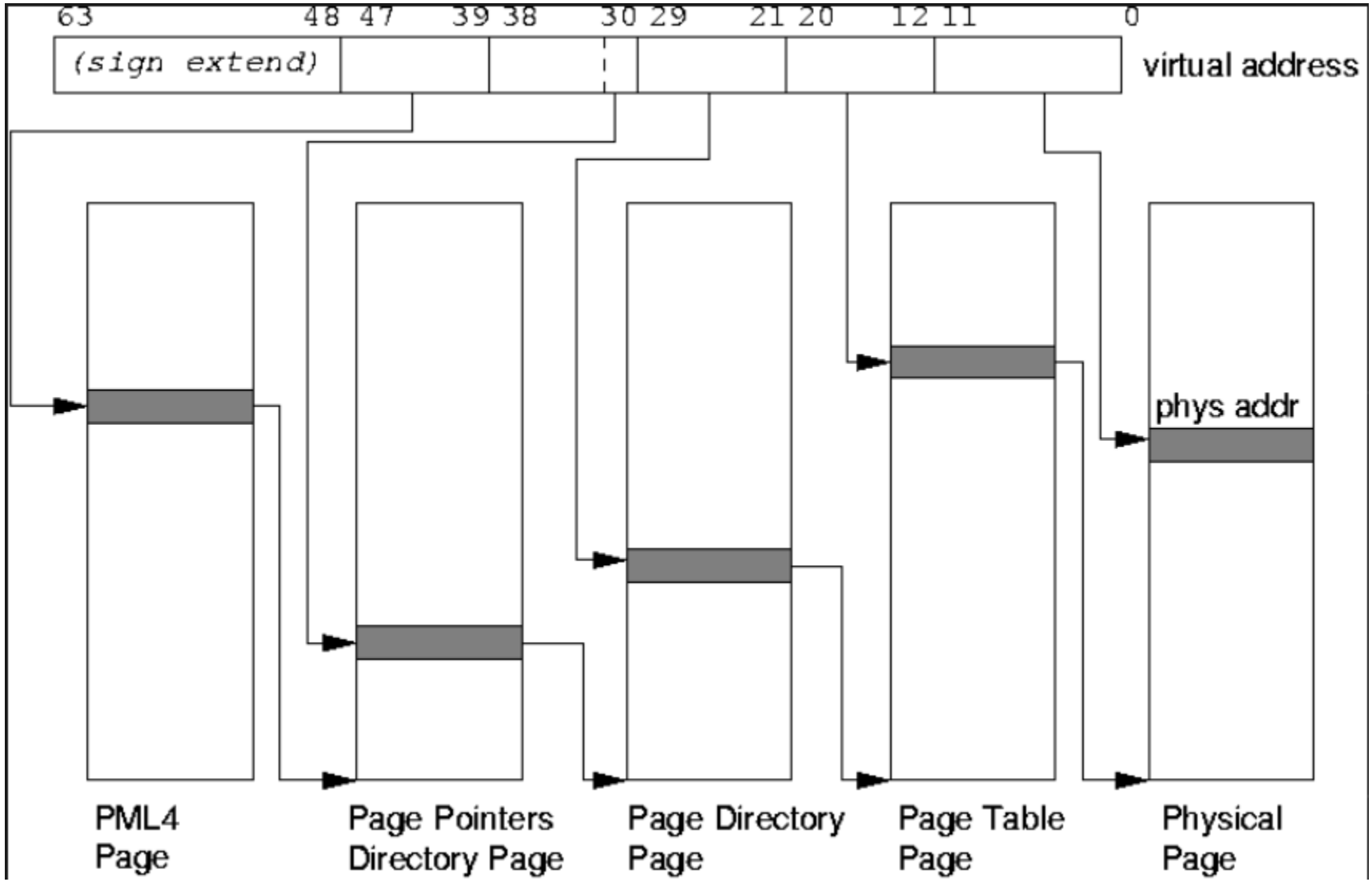


The Intel Pentium



Each running program has a page directory.

4 level PgTable



General Formula of PageTable Management

- Hardware defines the frame size as 2**N
- (virtual) page size is typically equal frame size (otherwise it's a power-of-two multiple, but that is rare and we shall not base on this exeption)
- Everything the OS manages is based on pagesize (hence framesize)
- You can compute all the semantics with some basic variables defined by hardware:
  - Virtual address range (e.g. 48-bit virtual address, means that the hardware only considers the 48-LSB bits from an virtual address for ld/st, all other raise a SEGV error)
  - Frame size
  - PTE size (e.g. 4byte or 8byte)
  - From there you can determine the number of page table hierarchies, offsets
- Example:
  - Framesize = 4KB → 12bit offset to index into frame 12bits
  - PTE size = 8Bytes -> 512 entries in each page table hierarchy 9bits / hierarchy
  - Virtual address range 48-bits: → 12 + N*9bits must add up to 48 bits N=4 hierarchies
  - ^^^^^^ the hardware does all the indexing as described before
- The OS must create its page table and VMM management following the hardware definition.

Speeding Up Paging

- Challenges:
  - Mapping virtual to physical address must be fast
  - If address space is large, page table will be large

Speeding Up Paging

- Challenges:
  - Mapping virtual to physical address must be fast → Translation Lookaside Buffer(TLB)
  - If address space is large, page table will be large →
    - Multi-level page table
    - Inverted page table

TLB

- **Observation**: most programs tend to make a large number of references to a small number of pages -> only fraction of the page table is heavily used
- TLB
  - Hardware device inside the MMU
  - **Caches** PageTable translations (VA -> PA)
  - Maps virtual to physical address without going to the page table (unless there's a miss)

TLB

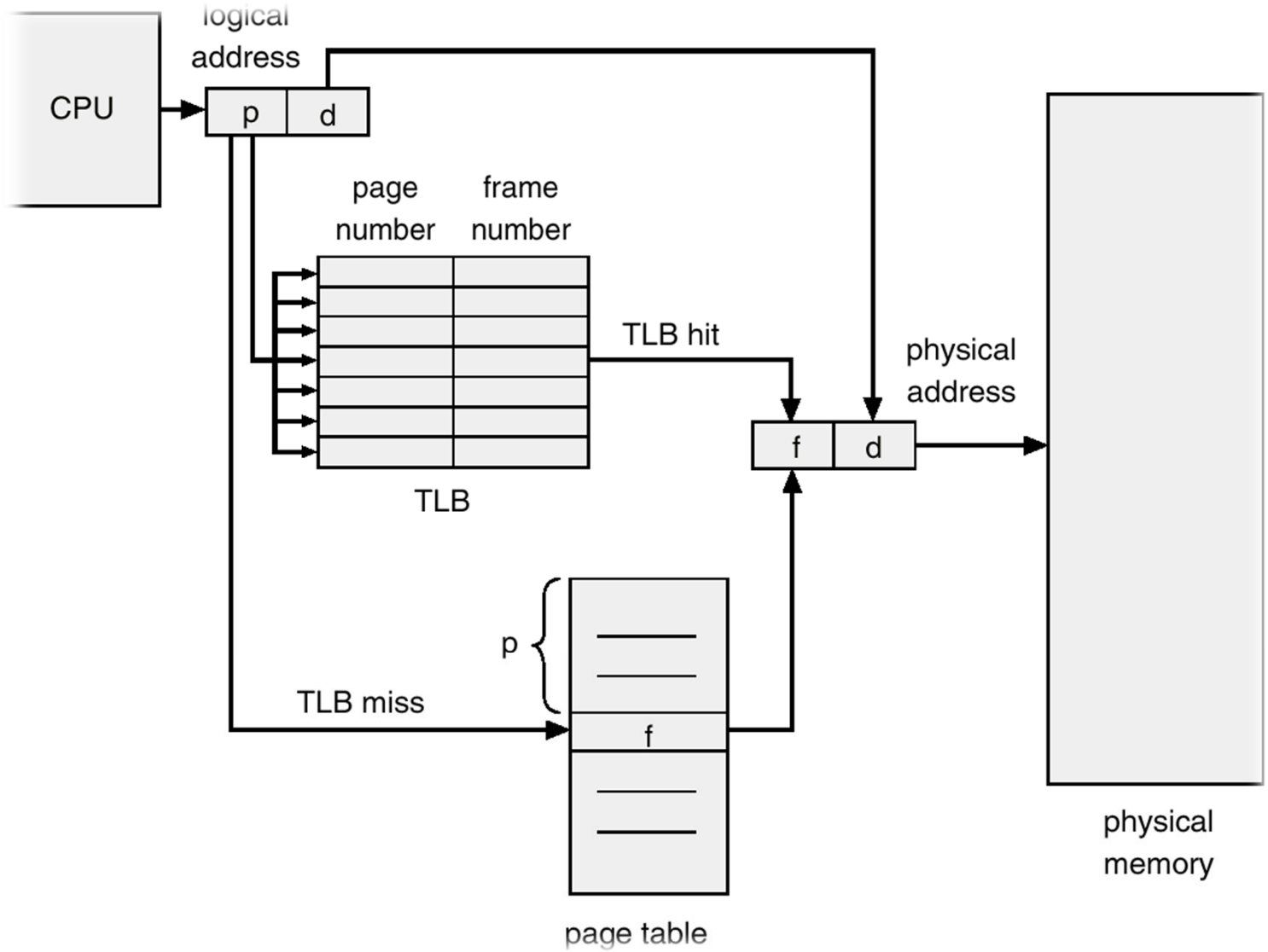
- In case of TLB miss -> MMU accesses page table
- TLB misses occur more frequently than page faults
- Optimizations
  - Software TLB management
    - Simpler MMU
    - Becomes rare in modern system

TLB

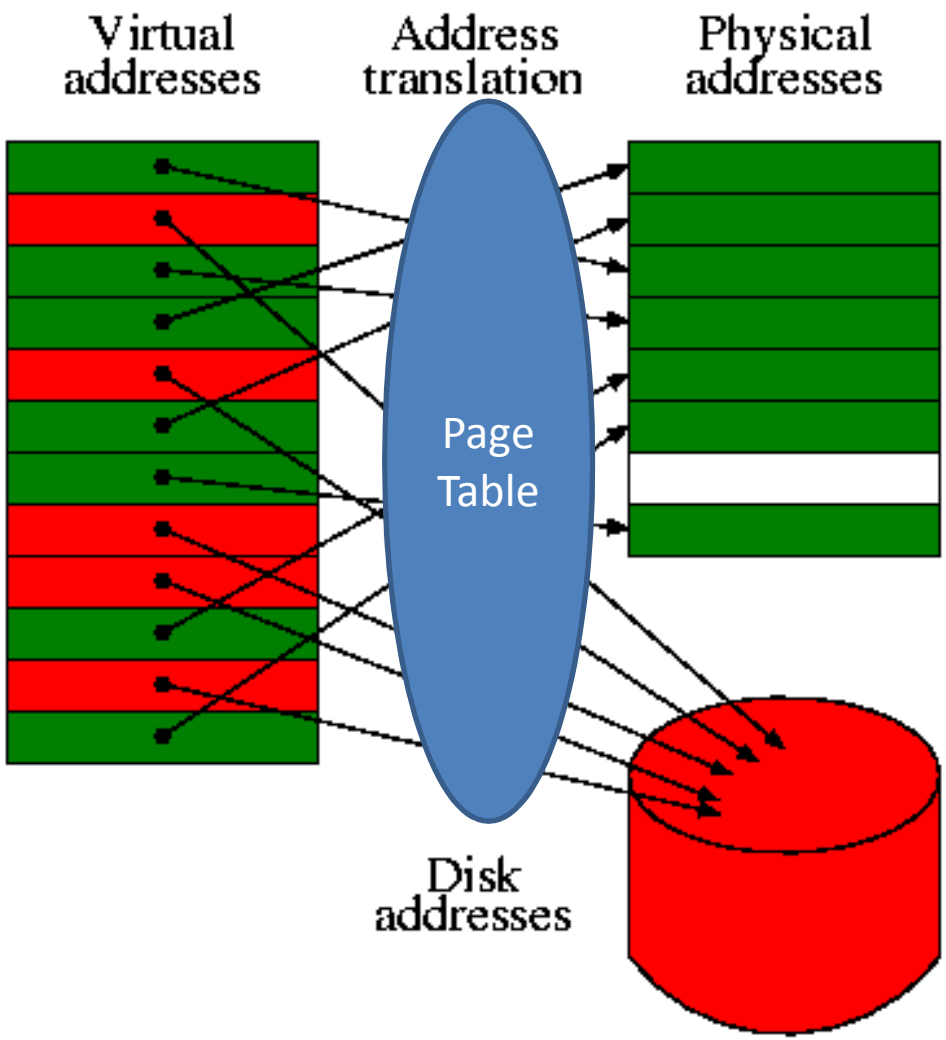
Valid	Virtual page	Modified	Protection	Page frame
1	140	1	RW	31
1	20	0	R X	38
1	130	1	RW	29
1	129	1	RW	62
1	19	0	R X	50
1	21	0	R X	45
1	860	1	RW	14
1	861	1	RW	75

TLB management

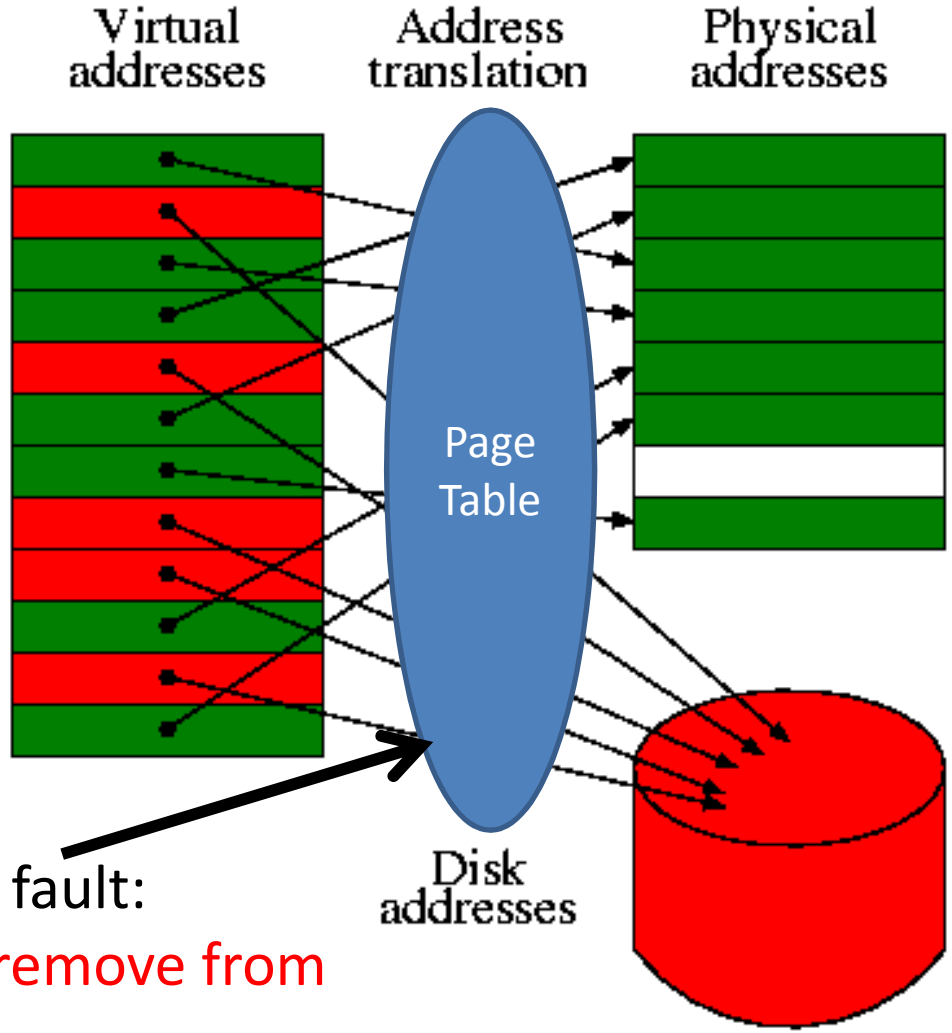
- TLB entries are not written back on access, just record the R and M bits in the PTE (it is a cache after all)
- On TLB capacity miss, if the entry is dirty, write it back to PageTable
- On changes to the PageTable, potential entries in the TLB need to be flushed.
  - TLB invalidation (either global or per address)


CPU
logical address
p
d
page number
frame number
TLB hit
physical address
f
d
TLB
p
TLB miss
f
page table
physical memory

Conceptual view of VMM



Conceptual view of VMM



In case of page fault:

which page to remove from Memory?

Replacement Policies

- Used in many contexts when storage is not enough
  - caches
  - web servers
  - pages
- Things to take into account when designing a replacement policy
  - measure of success
  - cost

Optimal Page Replacement Algorithm

- Each page labeled with the number of instructions that will be executed before this page is referenced
- Page with the highest label should be removed
- Impossible to implement

The Not Recently Used Replacement Algorithm

- Two status bits with each page
  - R: Set whenever the page is referenced (used)
  - M: Set when the page is written / dirty
- R and M bits are available in most computers implementing virtual memory
- Those bits are updated with each memory reference
  - Must be updated by hardware
  - Reset only by the OS
- Periodically (e.g. on each clock interrupt) the R bit is cleared
  - To distinguish pages that have been referenced recently

The Not Recently Used Replacement Algorithm

	R	M
Class 0:	0	0
Class 1:	0	1
Class 2:	1	0
Class 3:	1	1

NRU algorithm removes a page at random from the lowest numbered unempty class

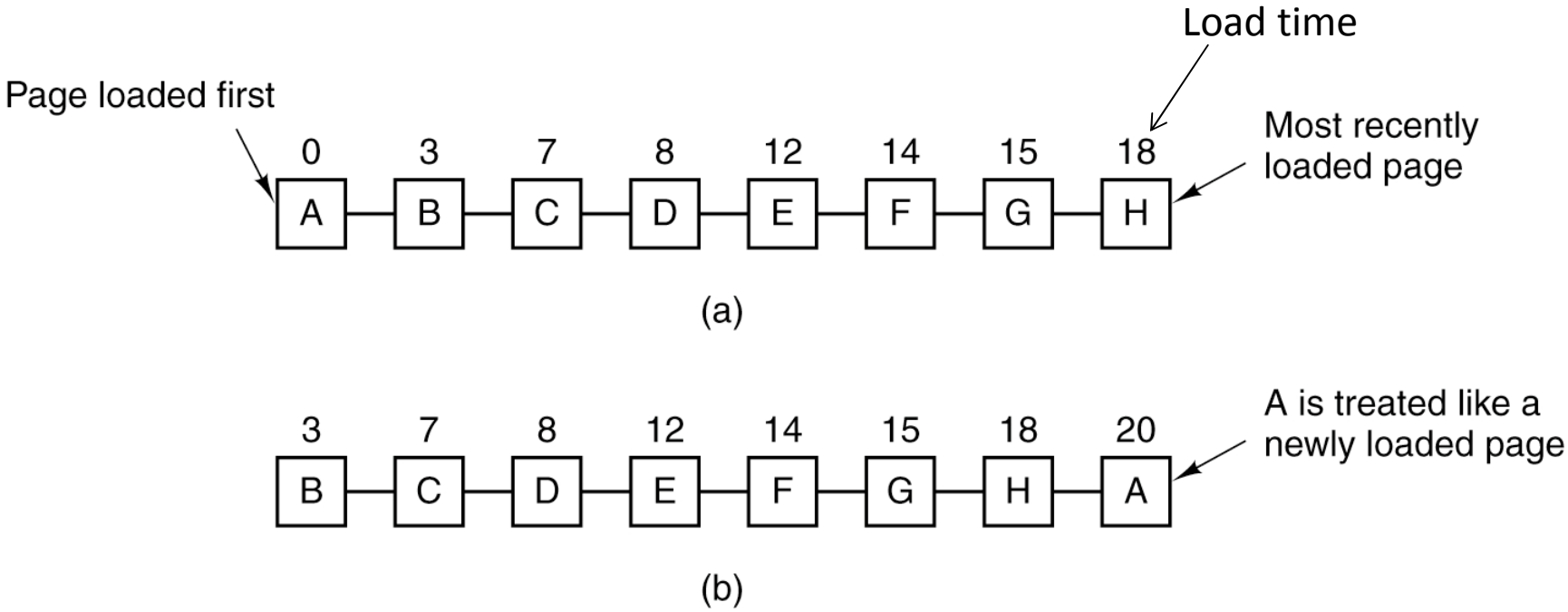
The FIFO Replacement Algorithm

- OS maintains a list of the pages currently in memory
- The most recent arrival at the tail
- On a page fault, the page at the head is removed

The Second-Chance Page Replacement Algorithm

- Modification to FIFO
- Inspect the R bit of the oldest page
  - If R=0 page is old and unused -> replace
  - If R=1 then
    - bit is cleared
    - page is put at the end of the list
    - the search continues
- If all pages have R=1, the algorithm degenerates to FIFO

The Second-Chance Page Replacement Algorithm

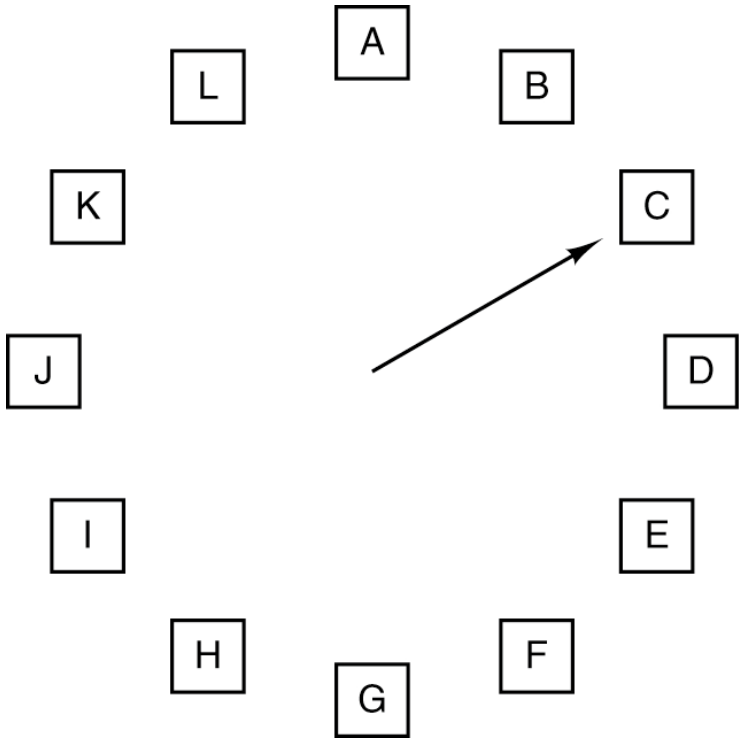


Moving pages around on the lists is inefficient

The Clock Page Replacement Policy

- Keep page frames on a circular list in the form of a clock
- The hand points to the oldest page
- When page fault occurs
  - The page pointed to by the hand is inspected
  - If R=0
    - page evicted
    - new page inserted into its place
    - hand is advanced
  - If R = 1
    - R is set to 0
    - hand is advanced

The Clock Page Replacement Policy



When a page fault occurs, the page the hand is pointing to is inspected. The action taken depends on the R bit:

R = 0: Evict the page

R = 1: Clear R and advance hand

The Least Recently Used (LRU) Page Replacement Algorithm

- Good approximation to optimal
- When page fault occurs, replace the page that has been unused for the longest time
- Realizable but not cheap

LRU
Hardware Implementation 1

- 64-bit counter increment after each instruction
- Each page table entry has a field large enough to include the value of the counter
- After each memory reference, the value of the counter is stored in the corresponding page entry.
- At page fault, the page with lowest value is discarded

LRU
Hardware Implementation 1

- 64-bit counter increment after each instruction
- Each page table entry has a field large enou unter
- Afte alue of th corresponding page entry.
- At page fault, the page with lowest value is discarded

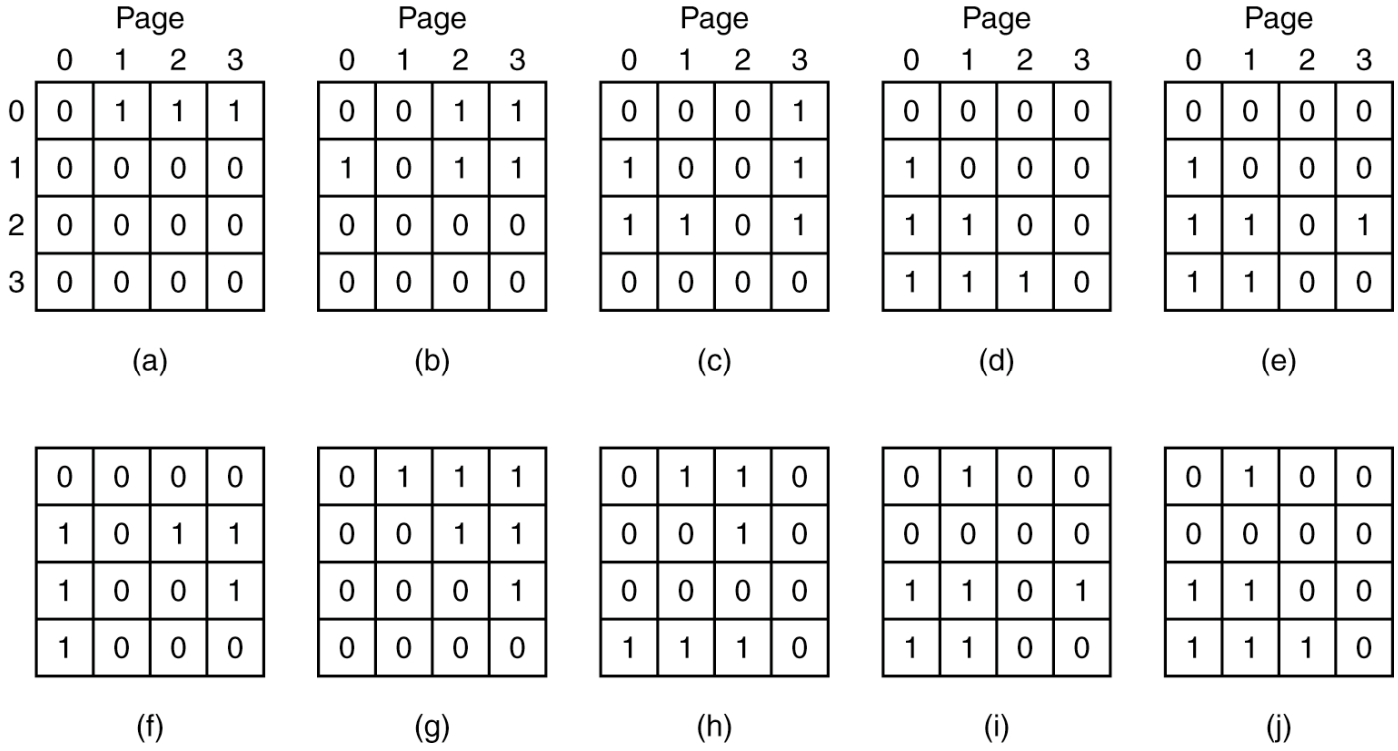
Too expensive!

Too Slow!

LRU: Hardware Implementation 2

- Machine with n page frames
- Hardware maintains a matrix of nxn bits
- Matrix initialized to all 0s
- Whenever page frame k is referenced
  - Set all bits of row k to 1
  - Set all bits of column k to 0
- The row with lowest value is the LRU

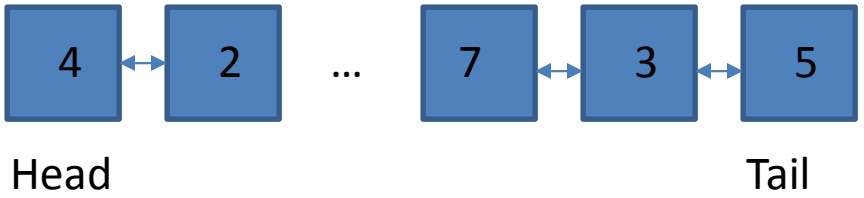
LRU: Hardware Implementation 2



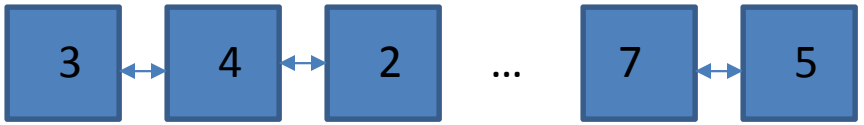
Pages referenced: 0 1 2 3 2 1 0 3 2 3

LRU
Hardware Implementation 3

- Maintain the LRU order of accesses in frame list by hardware



- After accessing page 3



- Use this in the lab

LRU Implementation

- Slow
- Few machines have required hardware

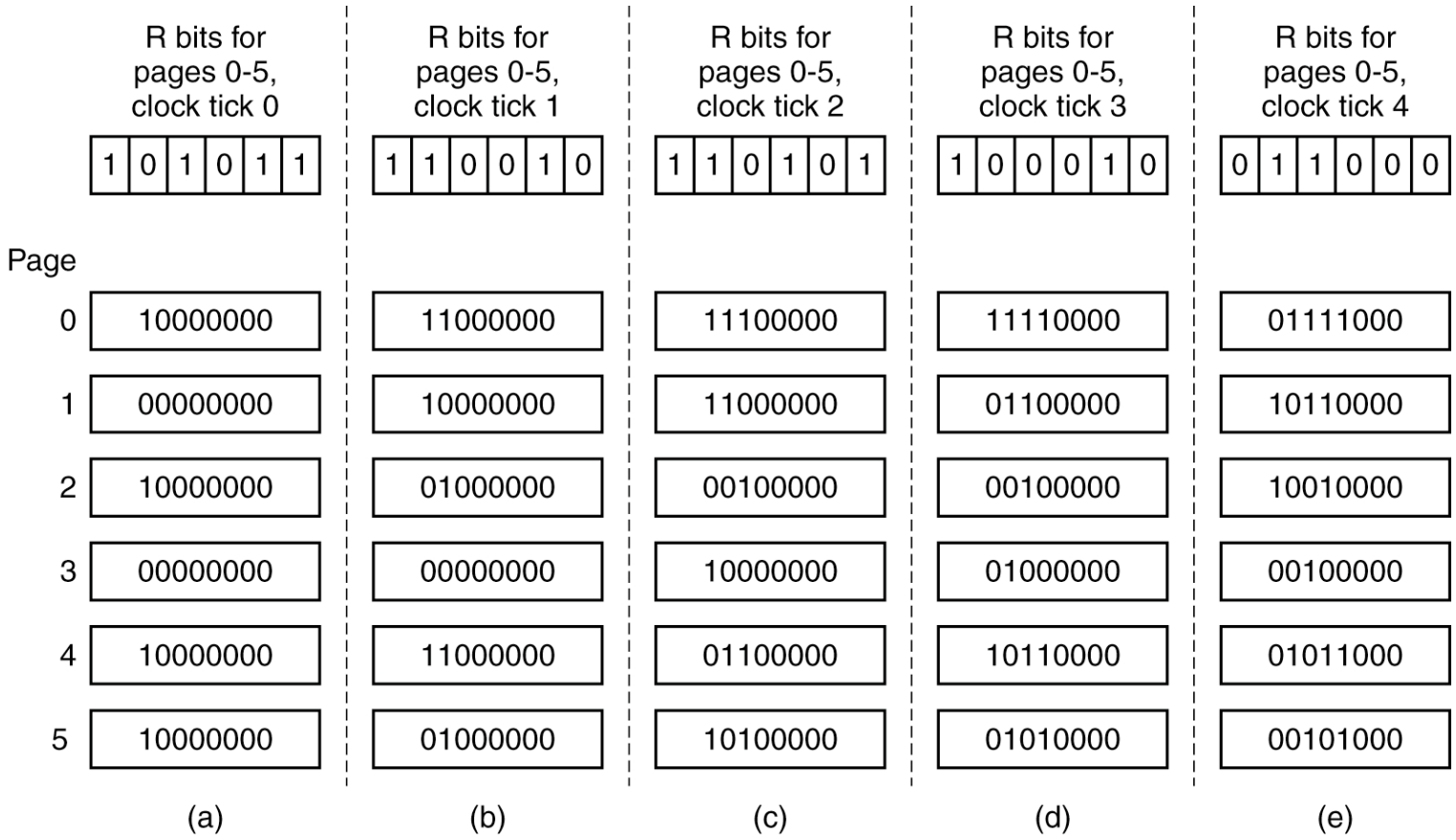
Simulating LRU in Software

- Not Frequently Used (NFU) algorithm
- Software counter associated with each page, initially zero
- At each clock interrupt, the OS scans all pages and add the R bit to the counter
- At page fault: the page with lowest counter is replaced

Enhancing NRU

- NRU never forgets anything -> high inertia
- Modifications:
  - shift counter right 1 bit before adding R
  - R is added to the leftmost
- This modified algorithm is called aging
- The page whose counter is lowest is replaced at page fault

Aging Algorithm



The Working Set Model

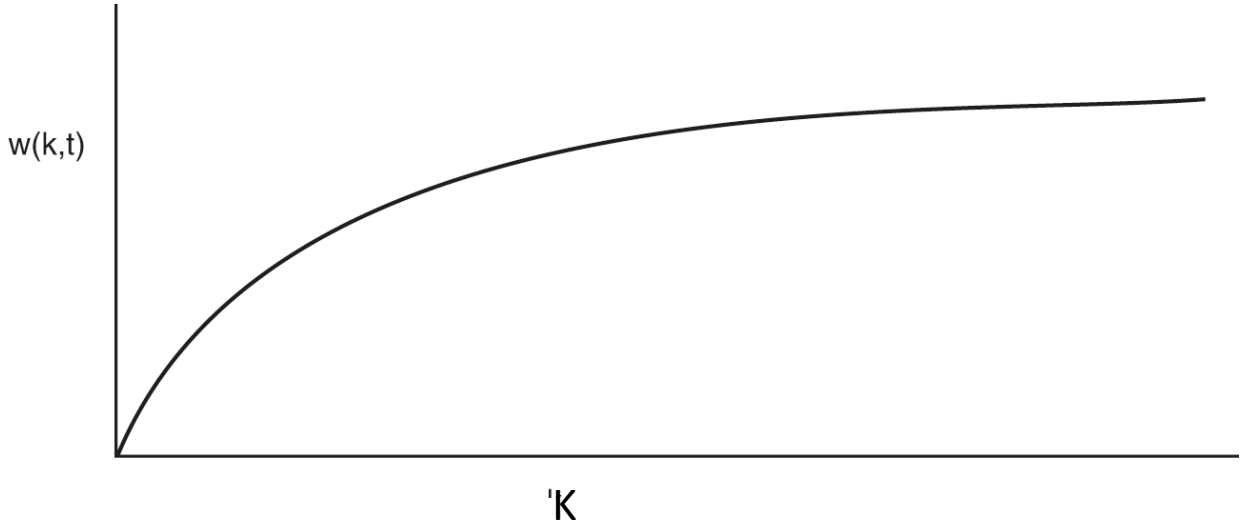
- Working set: the set of pages that a process is currently using
- Thrashing: a program causing page faults every few instructions

An important question:

In multiprogramming systems, processes are sometimes swapped to disk (i.e. all their pages are removed from memory). When they are brought back, which pages to bring?

The Working Set Model

- Try to keep track of each process' working set and make sure it is in memory before letting the process run.

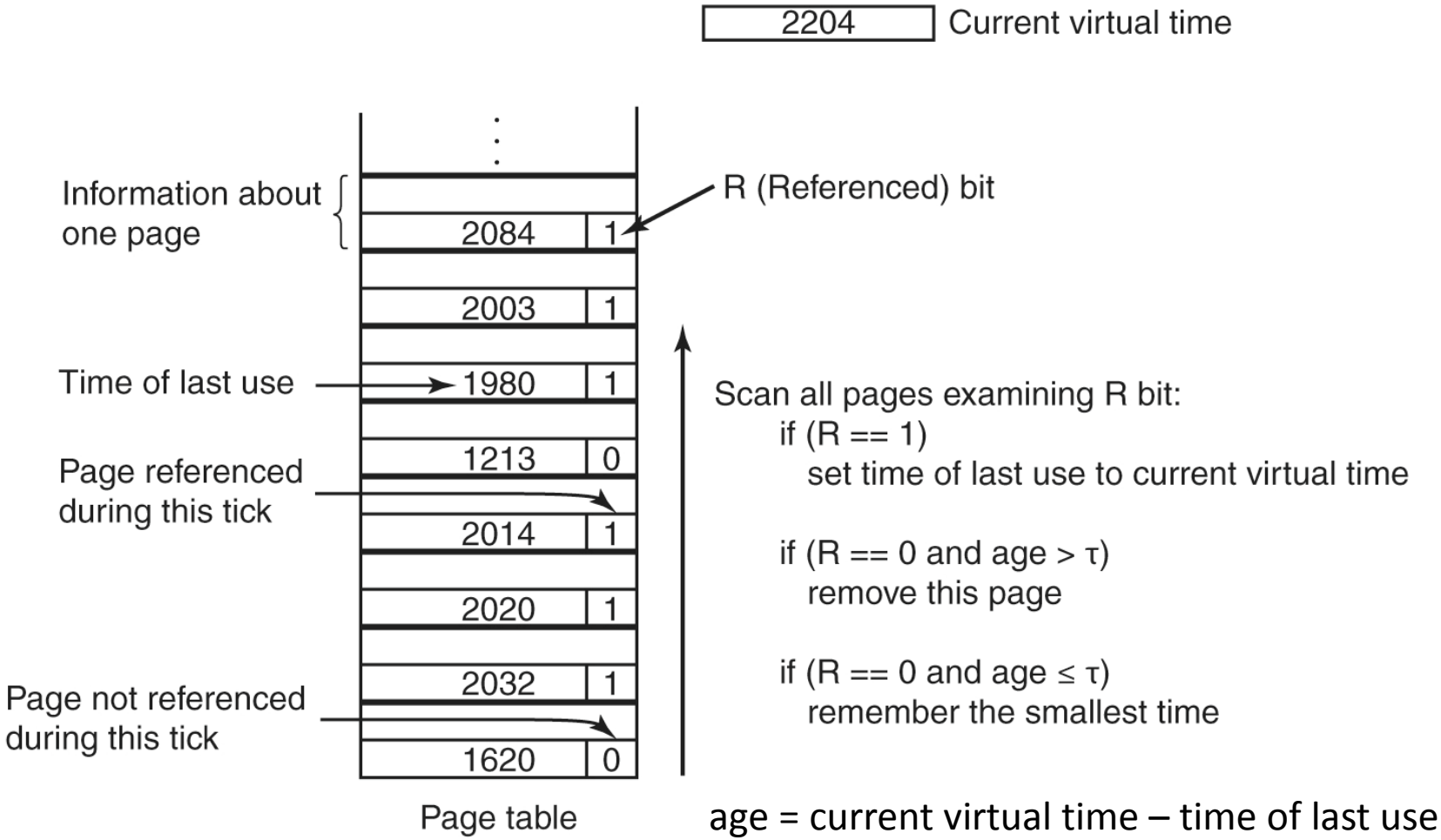


w(k,t): the set of pages accessed in the last k references at instant t

The Working Set Model

- OS must keep track of which pages are in the working set
- Replacement algorithm: evict pages not in the working set
- Possible implementation (but expensive):
  - working set = set of pages accessed in the last k memory references
- Approximations
  - working set = pages used in the last 100 msec

Working Set Page Replacement Algorithm

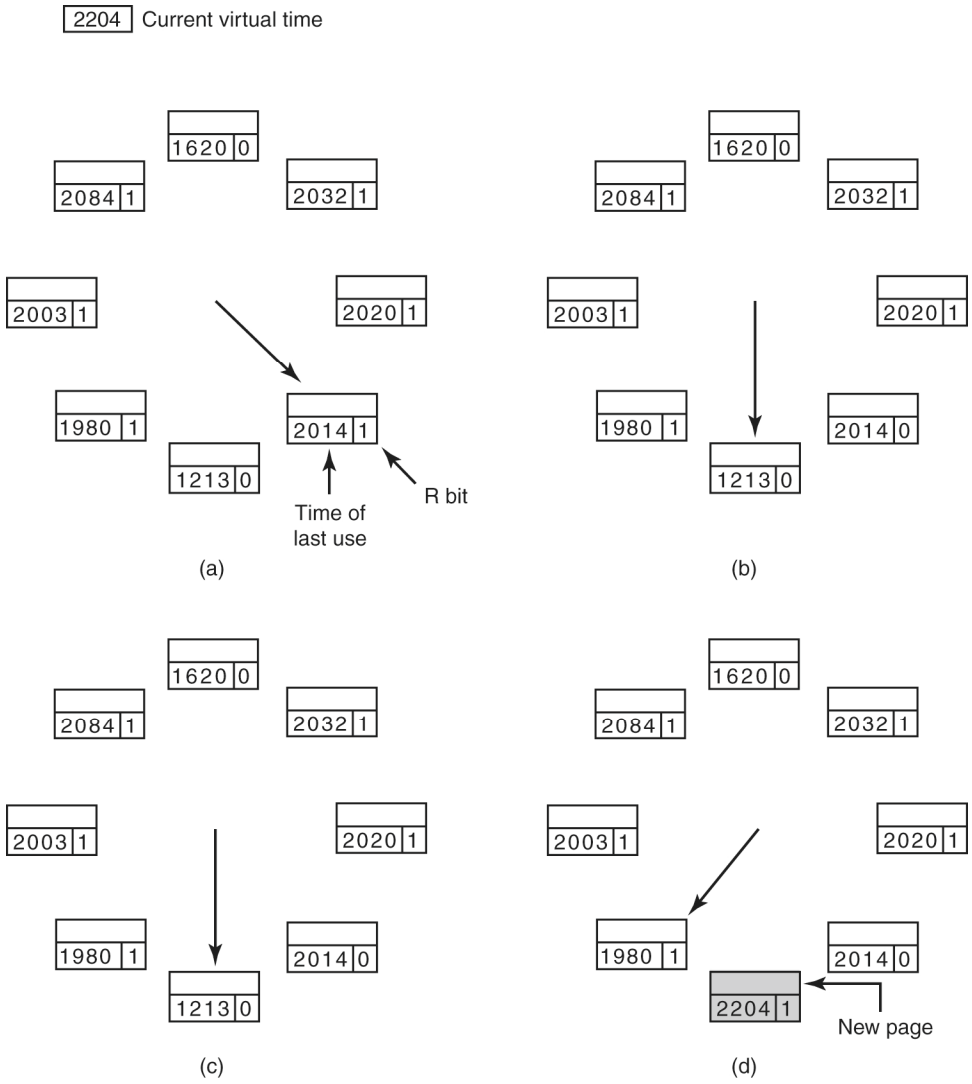


age = current virtual time – time of last use

The WSClock Page Replacement Algorithm

- Based on the clock algorithm and uses working set
- data structure: circular list of page frames
- Each entry contains: time of last use, R bit, and M bit
- At page fault: page pointed by hand is examined
  - If R = 1, the hand advances to next page and R is reset
  - If R = 0
    - If age > threshold and page is clean -> it is reclaimed
    - If page is dirty -> write to disk is scheduled and hand advances

The WSClock Page Replacement Algorithm



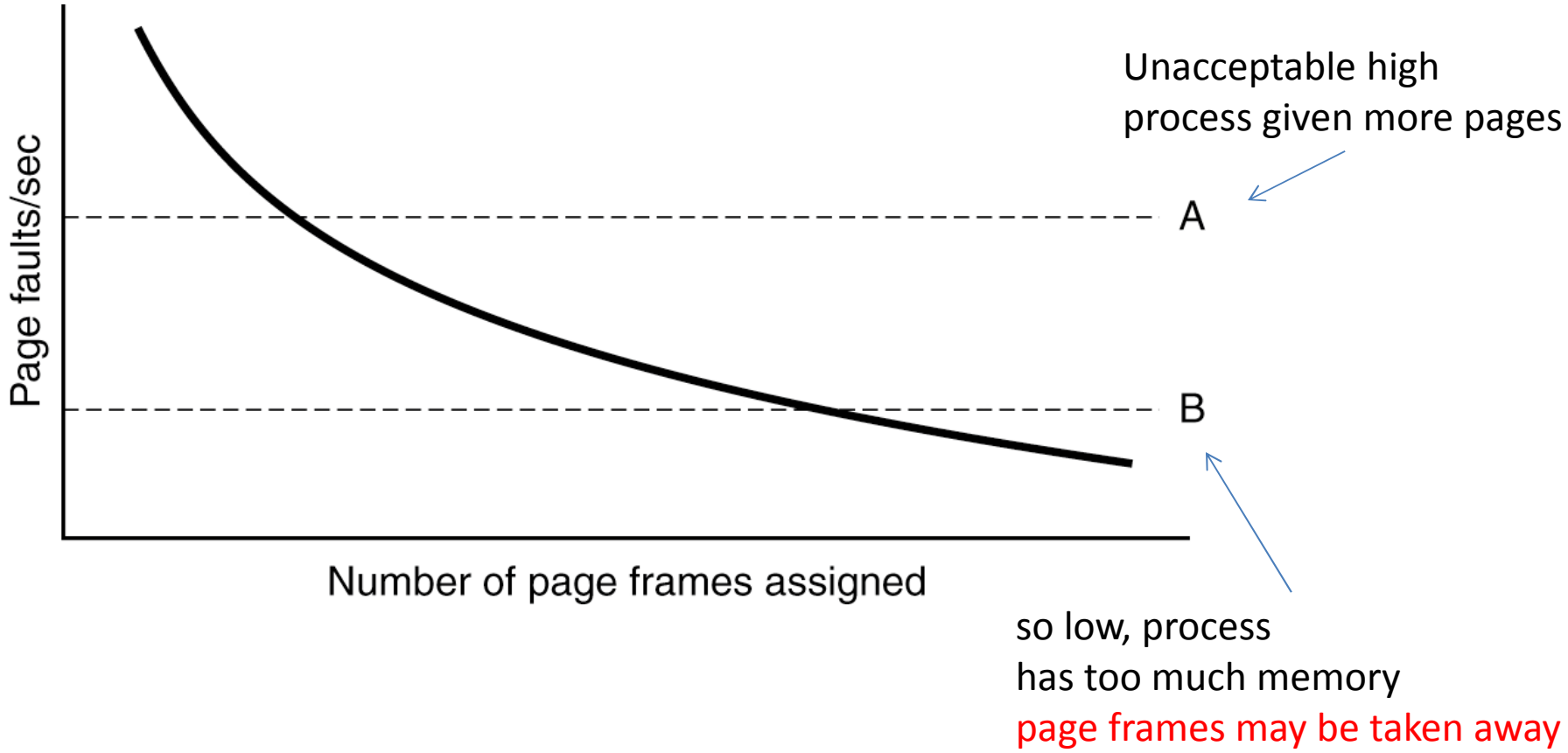
Design Issues for Paging: Local vs Global Allocation

- How memory should be allocated among the competing runnable processes?
- Local algorithms: allocating every process a fixed fraction of the memory
- Global algorithms: dynamically allocate page frames
- Global algorithms work better
  - If local algorithm used and working set grows → thrashing will result
  - If working set shrinks → local algorithms waste memory

Global Allocation

- Method 1: Periodically determine the number of running processes and allocate each process an equal share
- Method 2 (better): Pages allocated in proportion to each process total size
- Page Fault Frequency (PFF) algorithm: tells when to increase/decrease page allocation but says nothing about which page to replace.

Global Allocation: PFF



Design Issues: Load Control

- What if PFF indicates that some processes need more memory but none need less?
- Swap some processes to disk and free up all the pages they are holding.
- Which process(es) to swap?
  - Strive to make CPU busy (I/O bound vs CPU bound processes)
  - Process size

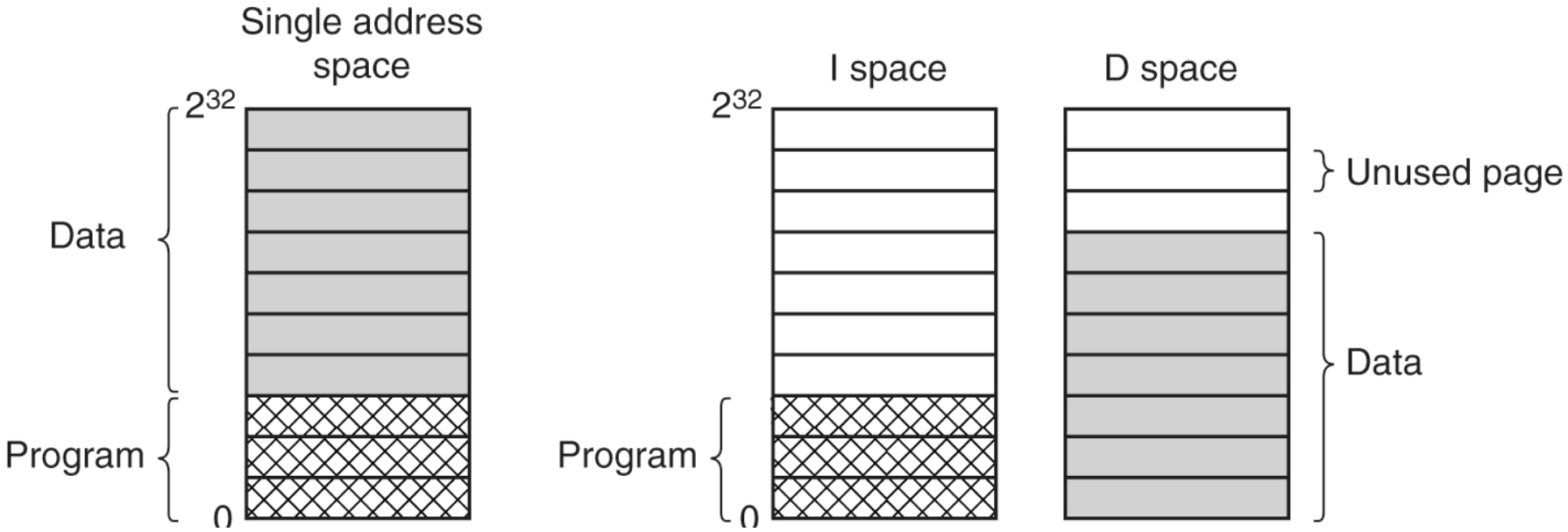
Design Issues: Page Size

- Large page size → internal fragmentation
- Small page size →
  - larger page table
  - More overhead transferring from disk

Design Issues: Page Size

- Assume:
  - s = process size
  - p = page size
  - e = size of each page table entry
- So:
  - number of pages needed = s/p
  - occupying: se/p bytes of page table space
  - wasted memory due to fragmentation: p/2
  - overhead = se/p + p/2
- We want to minimize the overhead:
  - Take derivative of overhead and equate to 0:
  - $-se/p^2 + \frac{1}{2} = 0 \rightarrow p = \sqrt{2se}$

Design Issues:
Separate Instruction and Data Spaces



- The linker must know about it
- Paging can be used in each separately

Design Issues: Shared Pages

- To save space, when same program is running by several users for example
- If separate I and D spaces: process table has pointers to Instruction page table and Data page table
- In case of common I and D spaces:
  - Special data structure is needed to keep track of shared pages
  - Copy on write for data pages

Design Issues: Shared Libraries

- Dynamically linked
- Loaded when program is loaded or when functions in them are called for the first time
- Compilers must not produce instructions using absolute addresses in libraries → position-independent code

Design Issues: Cleaning Policy

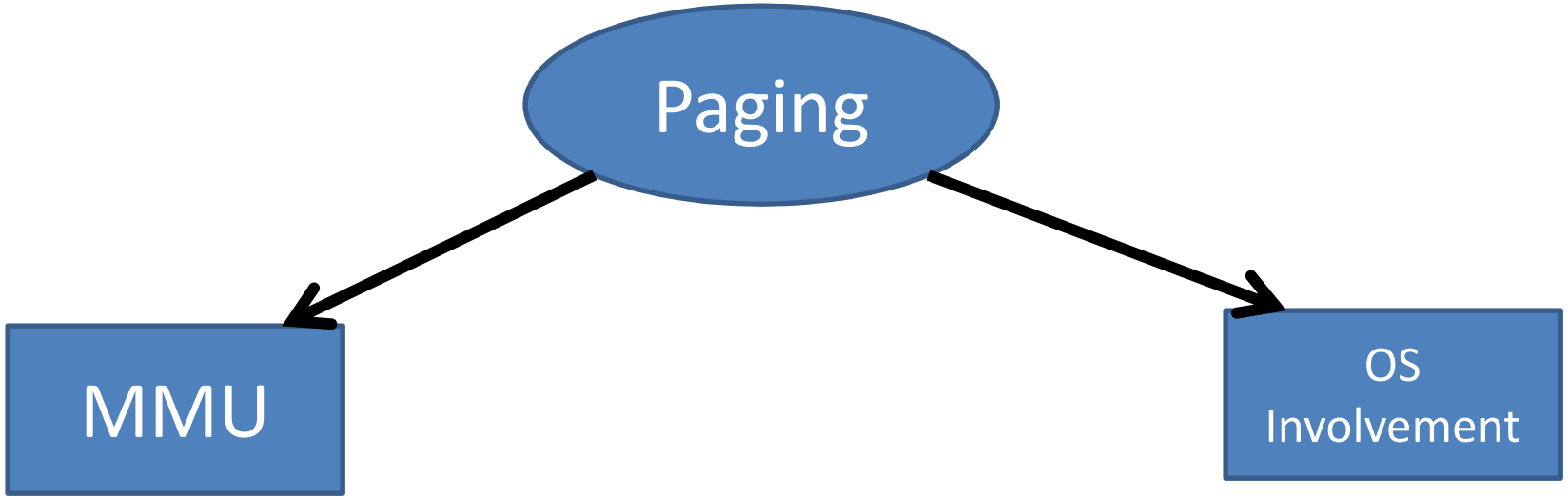
- Paging daemon
- Sleeps most of the time
- Awakened periodically to inspect state of the memory
- If too few pages are free -> daemon begins selecting pages to evict

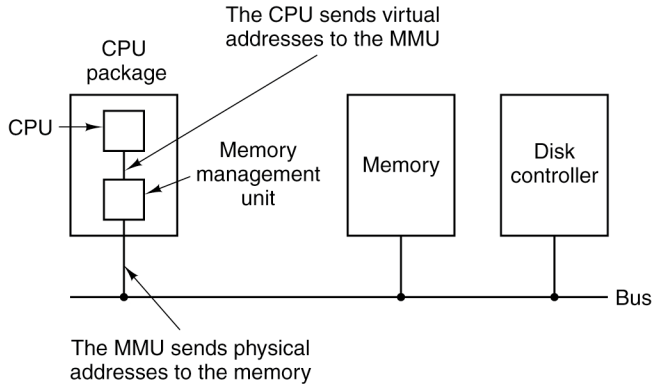
Semi-Conclusions

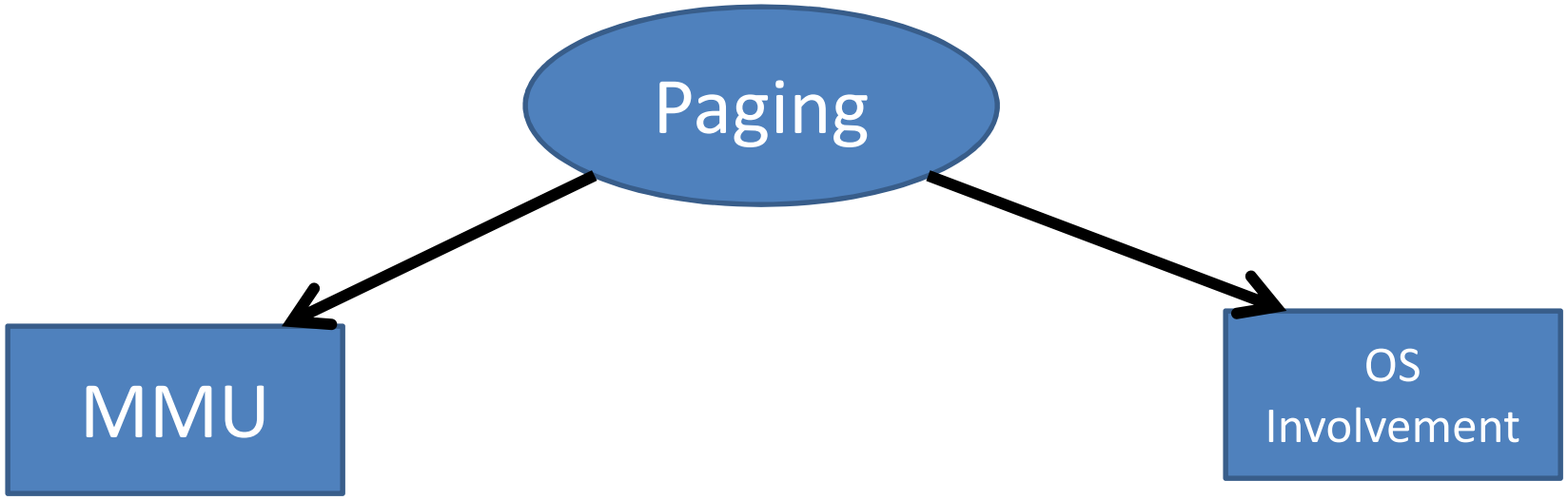
- Virtual memory is very widely used
- Many design issues for paging systems:
  - Page replacement algorithm
    - The two best ones: aging and WSClock
  - Page size
  - Local vs Global Allocation
    - Global algorithms work better
  - Load control
  - Dealing with shared pages

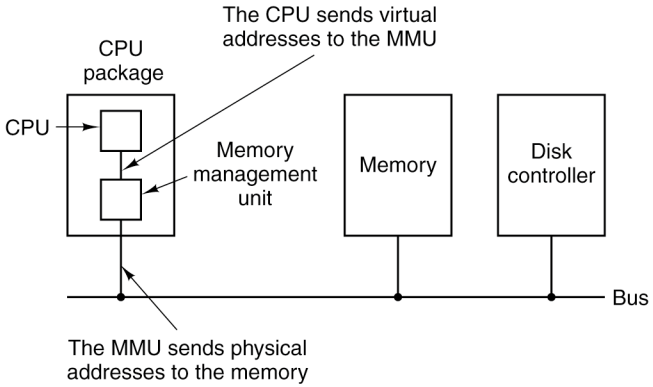
Summary of Paging

- Virtual address space bigger than physical memory
- Mapping virtual address to physical address
- Virtual address space divided into fixed-size units called pages
- Physical address space divided into fixed-size units called pages frames
- Virtual address space of a process can be non-contiguous in physical address space


Paging
MMU
OS
Involvement


The CPU sends virtual addresses to the MMU
CPU package
CPU
Memory management unit
Memory
Disk controller
Bus
The MMU sends physical addresses to the memory


Paging
MMU
OS
Involvement


The CPU sends virtual addresses to the MMU
CPU package
CPU
Memory management unit
Memory
Disk controller
Bus
The MMU sends physical addresses to the memory

?

OS Involvement With Paging

- When a new process is created
- When a process is scheduled for execution
- When process exits
- When page fault occurs

OS Involvement With Paging

- When a new process is created
  - Determine how large the program and data will be (initially)
  - Create page table
  - Allocate space in memory for page table
  - Record info about page table and swap area in process table
- When a process is scheduled for execution
- When process exits
- When page fault occurs

OS Involvement With Paging

- When a new process is created
- When a process is scheduled for execution
- When process exits
- When page fault occurs

OS Involvement With Paging

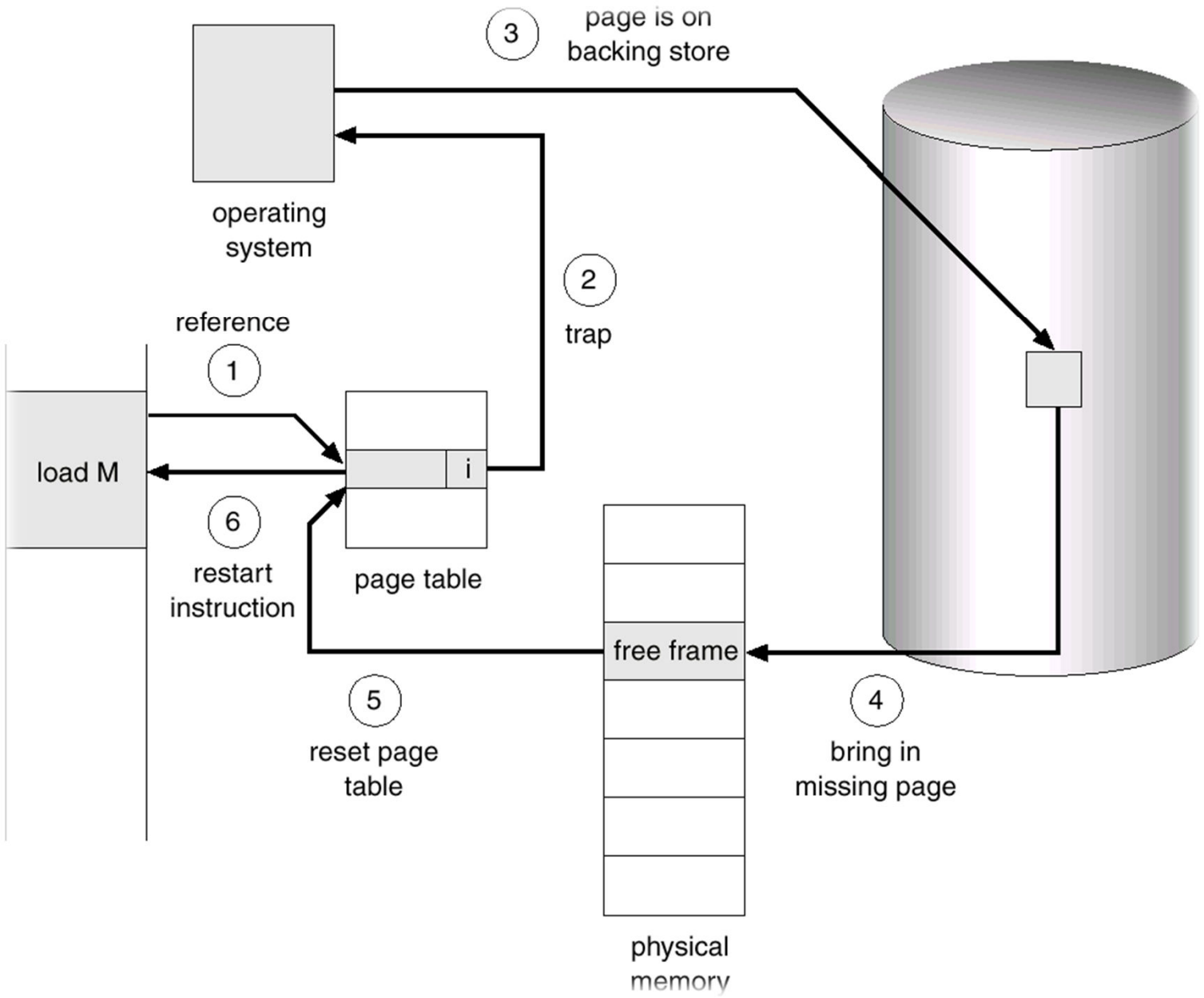
- When a new process is created
- When a process is scheduled for execution
  - MMU reset for the process
  - TLB flushed
  - Process table made current
- When process exits
- When page fault occurs

OS Involvement With Paging

- When a new process is created
- When a process is scheduled for execution
- When process exits
  - OS releases the process page table
  - Frees its pages and disk space
- When page fault occurs

OS Involvement With Paging

- When a new process is created
- When a process is scheduled for execution
- When process exits
- When page fault occurs


3
page is on
backing store
operating
system
2
trap
reference
1
load M
i
6
restart
instruction
page table
free frame
5
reset page
table
4
bring in
missing page
physical
memory

Page Fault Handling

1. The hardware:
   - Saves program counter
   - Traps to kernel
2. An assembly routine saves general registers and calls OS
3. OS tried to discover which virtual page is needed
4. OS checks address validation and protection and assign a page frame (page replacement may be needed)

Page Fault Handling

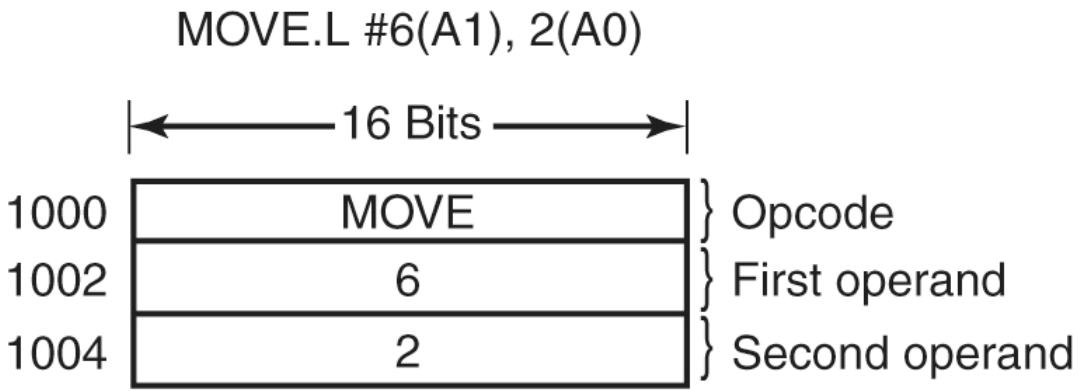
5. If page frame selected is dirty
   - Page scheduled to transfer to disk
   - Frame marked as busy
   - OS suspends the current process
   - Context switch takes place
6. Once the page frame is clean
   - OS looks up disk address where needed page is
   - OS schedules a disk operation
   - Faulting process still suspended
7. When disk interrupts indicates page has arrived
   - OS updates page table

Page Fault Handling

8. Faulting instruction is backed up to its original state before page fault and PC is reset to point to it.
9. Process is scheduled for execution and OS returns to the assembly routine.
10. The routine reloads registers and other state information and returns to user space.

Instruction Backup At Page Fault

- In order to restart the instruction, the OS needs to know where the first byte of the instruction is.



The value of the PC at the time of trap depends on which operand faulted and the CPU microcode

Interesting Scenario: Virtual Memory & I/O Interaction

- Process issues a syscall to read a file into a buffer
- Process suspended while waiting for I/O
- New process starts executing
- This other process gets a page fault
- If paging algorithm is global there is a change the page containing the buffer be removed from memory.
- The I/O operation of the first process will write some data into the buffer and some other on the just-loaded page!

Interesting Scenario: Virtual Memory & I/O Interaction

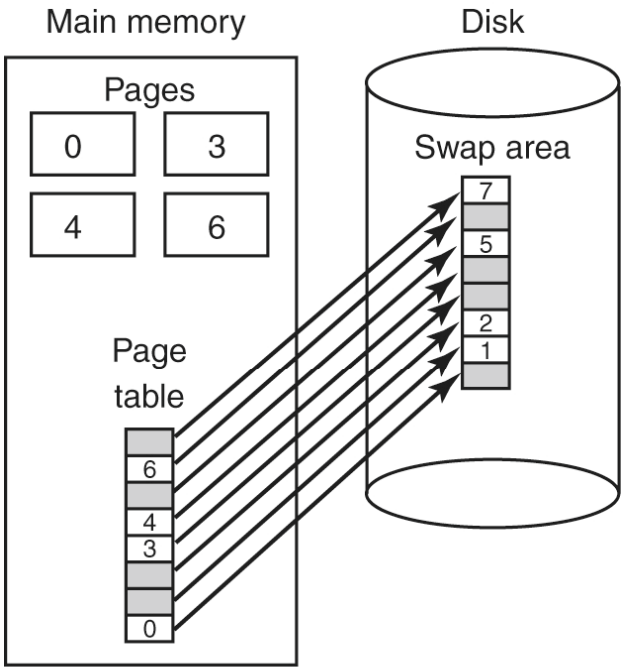
- Process issues a syscall to read a file into a buffer
- Process suspended while waiting for I/O
- New process starts executing
- This other process gets a page fault
- If paging algorithm is global there is a chance the page containing the buffer be removed from memory.
- The I/O operation of the first process will write some data into the buffer and some other on the just-loaded page!

One solution: Locking (pinning) pages engaged in I/O so that they will not be removed.

Backing Store

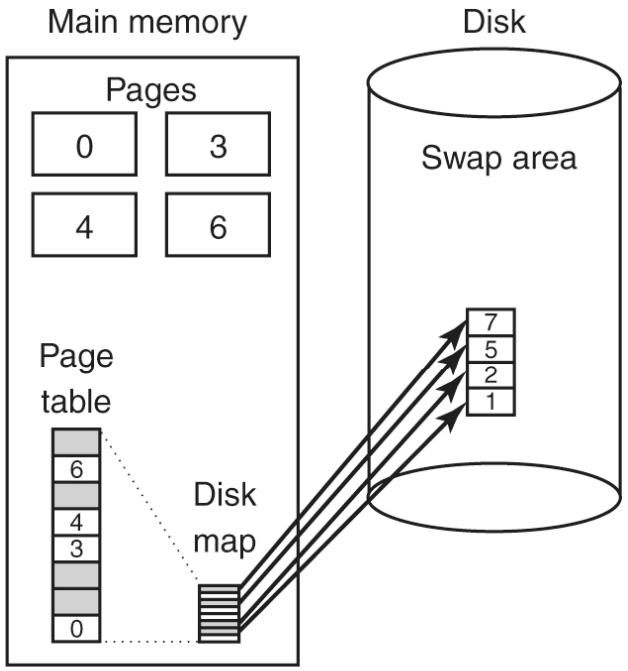
- Swap area: not a normal file system on it.
- Associates with each process the disk address of its swap area; store in the process table
- Before process starts swap area must be initialized
  - One way: copy all process image into swap area [static swap area]
  - Another way: don't copy anything and let the process swap out [dynamic]
- Instead of disk partition, one or more preallocated files within the normal file system can be used [Windows uses this approach.]

Backing Store



(a)

Static Swap Area



(b)

Dynamic

Page Fault Handler

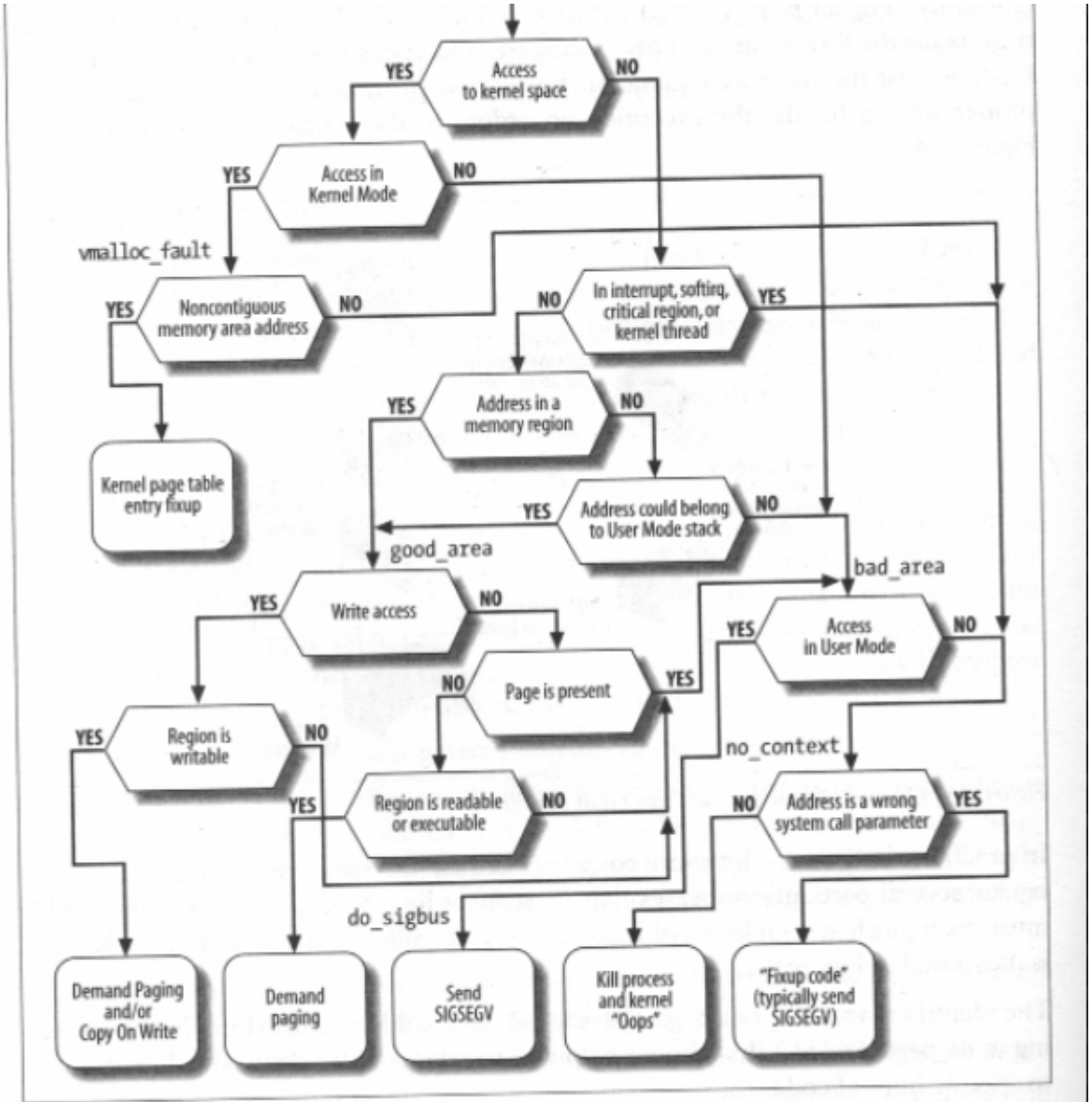


Figure 9-5. The flow diagram of the Page Fault handler

Memory Mappings

- Each process consists of many memory areas:
  - Aka:
    - segments
    - regions
    - **VMAs** virtual memory areas.
  - Ex: Heap, stack, code, data, ronly-data, etc.
- Each has different characteristics
  - Protection (executable, rw, rdonly)
  - Fixed, can grow (up or down) [heap, stack]
- Each process can have 10s-100s of these.

Example: emacs VMAs

```
00110000-001a3000 r-xp 00000000 08:01 267740     /usr/lib/libgdk-x11-2.0.so.0.2000.1
001a3000-001a5000 r--p 00093000 08:01 267740     /usr/lib/libgdk-x11-2.0.so.0.2000.1
001a5000-001a6000 rw-p 00095000 08:01 267740     /usr/lib/libgdk-x11-2.0.so.0.2000.1
001a6000-001bf000 r-xp 00000000 08:01 267488     /usr/lib/libatk-1.0.so.0.3009.1
001bf000-001c0000 ---p 00019000 08:01 267488     /usr/lib/libatk-1.0.so.0.3009.1
001c0000-001c1000 r--p 00019000 08:01 267488     /usr/lib/libatk-1.0.so.0.3009.1
001c1000-001c2000 rw-p 0001a000 08:01 267488     /usr/lib/libatk-1.0.so.0.3009.1
001c2000-001cc000 r-xp 00000000 08:01 265243     /usr/lib/libpangocairo-1.0.so.0.2800.0
001cc000-001cd000 r--p 00009000 08:01 265243     /usr/lib/libpangocairo-1.0.so.0.2800.0
001cd000-001ce000 rw-p 0000a000 08:01 265243     /usr/lib/libpangocairo-1.0.so.0.2800.0
001ce000-001d1000 r-xp 00000000 08:01 267773     /usr/lib/libgmodule-2.0.so.0.2400.1
001d1000-001d2000 r--p 00002000 08:01 267773     /usr/lib/libgmodule-2.0.so.0.2400.1
001d2000-001d3000 rw-p 00003000 08:01 267773     /usr/lib/libgmodule-2.0.so.0.2400.1
001d3000-001e8000 r-xp 00000000 08:01 267367     /usr/lib/libICE.so.6.3.0
001e8000-001e9000 r--p 00014000 08:01 267367     /usr/lib/libICE.so.6.3.0
001e9000-001ea000 rw-p 00015000 08:01 267367     /usr/lib/libICE.so.6.3.0
001ea000-001ec000 rw-p 00000000 00:00 0
001ec000-001fb000 r-xp 00000000 08:01 267433     /usr/lib/libXpm.so.4.11.0
001fb000-001fc000 r--p 0000e000 08:01 267433     /usr/lib/libXpm.so.4.11.0
001fc000-001fd000 rw-p 0000f000 08:01 267433     /usr/lib/libXpm.so.4.11.0
```

336 VMAs

```
b75fc000-b763b000 r--p 00000000 08:01 395228     /usr/lib/locale/en_US.utf8/LC_CTYPE
b763b000-b763c000 r--p 00000000 08:01 395233     /usr/lib/locale/en_US.utf8/LC_NUMERIC
b763c000-b763d000 r--p 00000000 08:01 404948     /usr/lib/locale/en_US.utf8/LC_TIME
b763d000-b775b000 r--p 00000000 08:01 395227     /usr/lib/locale/en_US.utf8/LC_COLLATE
b775b000-b776c000 rw-p 00000000 00:00 0
b776c000-b776d000 r--p 00000000 08:01 404949     /usr/lib/locale/en_US.utf8/LC_MONETARY
b776d000-b776e000 r--p 00000000 08:01 404950     /usr/lib/locale/en_US.utf8/LC_MESSAGES/SYS_LC_MESSAGES
b776e000-b776f000 r--p 00000000 08:01 395442     /usr/lib/locale/en_US.utf8/LC_PAPER
b776f000-b7770000 r--p 00000000 08:01 395102     /usr/lib/locale/en_US.utf8/LC_NAME
b7770000-b7771000 r--p 00000000 08:01 404951     /usr/lib/locale/en_US.utf8/LC_ADDRESS
b7771000-b7772000 r--p 00000000 08:01 404952     /usr/lib/locale/en_US.utf8/LC_TELEPHONE
b7772000-b7773000 r--p 00000000 08:01 395529     /usr/lib/locale/en_US.utf8/LC_MEASUREMENT
b7773000-b777a000 r--s 00000000 08:01 269322     /usr/lib/gconv/gconv-modules.cache
b777a000-b777b000 r--p 00000000 08:01 404953     /usr/lib/locale/en_US.utf8/LC_IDENTIFICATION
b777b000-b777d000 rw-p 00000000 00:00 0
bfd9a000-bfe5d000 rw-p 00000000 00:00 0          [stack]
```

More on memory regions

Memory mapped files:
- Typically **no** swap space required
- The file is the swap space

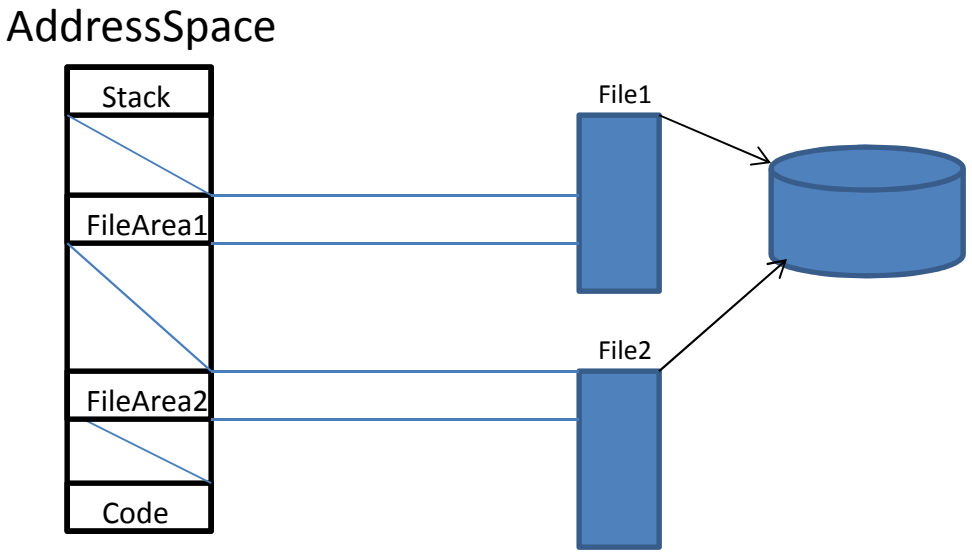
```
b7773000-b777a000 r--s 00000000 08:01 269322     /usr/lib/gconv/gconv-modules.cache
b777a000-b777b000 r--p 00000000 08:01 404953     /usr/lib/locale/en_US.utf8/LC_IDENTIFICATION
b777b000-b777d000 rw-p 00000000 00:00 0
bfd9a000-bfe5d000 rw-p 00000000 00:00 0          [stack]
```

Anonymous memory:
- Swap space required

```
NAME
       mmap, munmap - map or unmap files or devices into memory

SYNOPSIS
       #include <sys/mman.h>

       void *mmap(void *addr, size_t length, int prot, int flags,
                  int fd, off_t offset);
       int munmap(void *addr, size_t length);

DESCRIPTION
       mmap()  creates  a  new mapping in the virtual address space of the calling process.
       The starting address for the new mapping is specified in addr.  The length  argument
       specifies the length of the mapping.
```

Memory Mapped Files

- Maps a VMA -> file segment
- It's continouos
- On PageFault, it fetches the content from file at the appropriate offset into a virtual page of the address space
- On "swapout", writes back to file

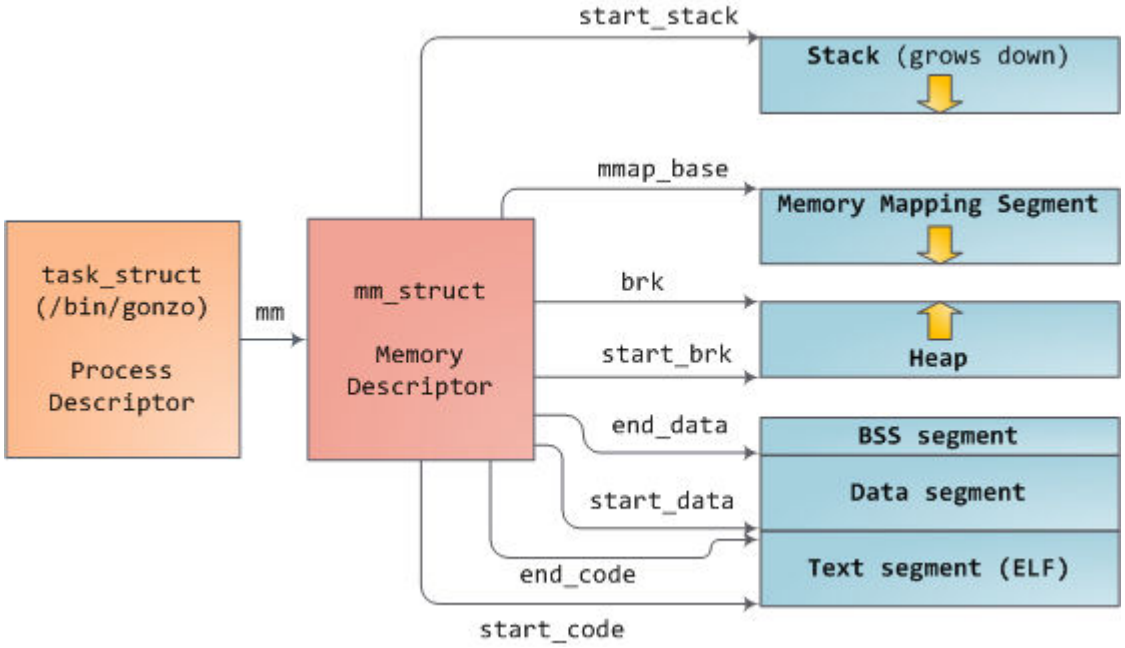


Benefits

- Can perform load/store operations vs input/output

Linux Example #1

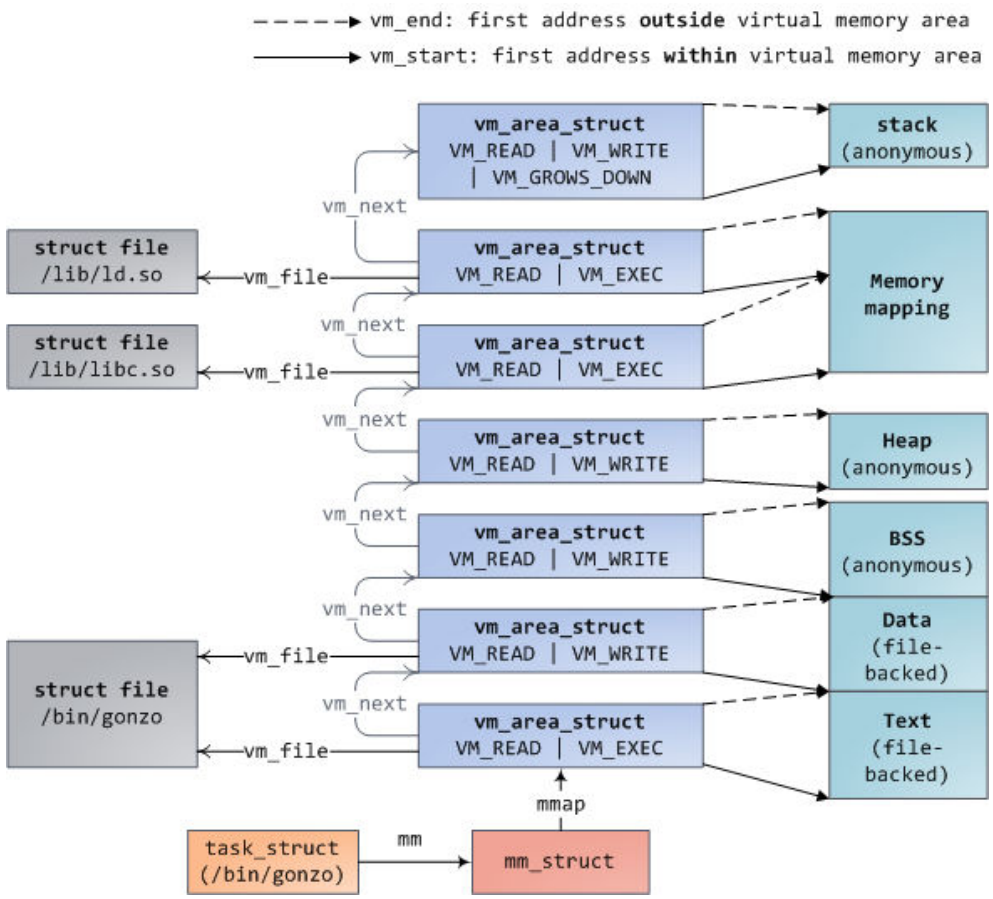
- Objects:



Source: http://duartes.org/gustavo/blog/post/how-the-kernel-manages-your-memory/

Linux Example

- Vma areas
- Anonymous vs. filebacked

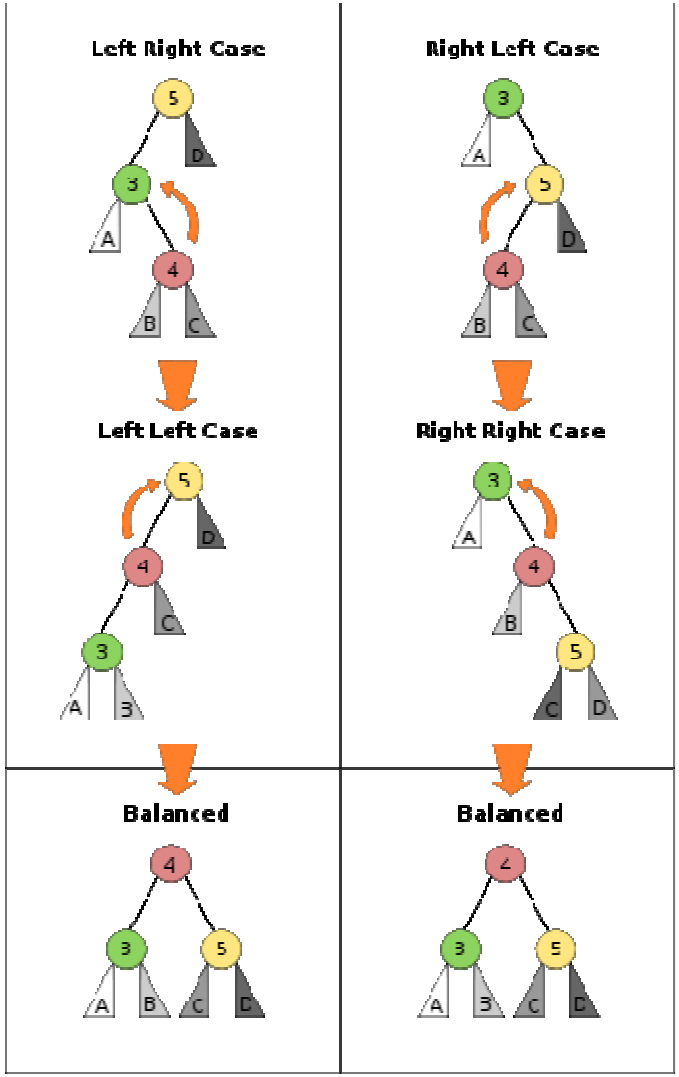


Organization of Memory Regions

- Cells are by default non-overlapping
  - Called VMA (Virtual Memory Areas)
- Organized as AVL trees
- Identify in O(logN) time during pgfault
  - Pgfault(vaddr) → VMA
- Rebalanced when VMA is added or deleted

VMA organization

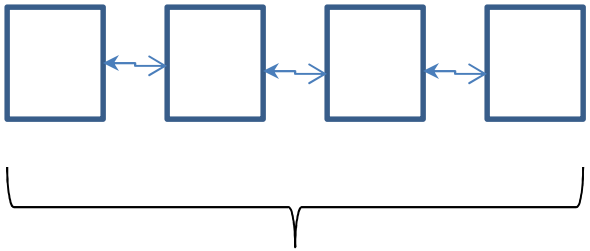
- Organized as balanced tree
- Node [start – end]
- On add/delete
  → rebalance
- Lookup in O(log(n))



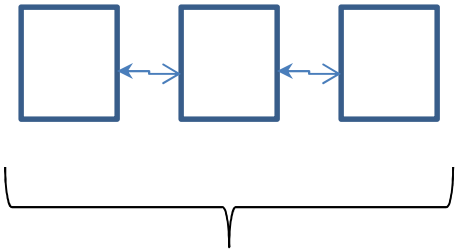
Linux MemMgmt LRU Approximation

- Arrange the entries in LRU
- Two queues

FrameTable
(one entry related to
each physical frame)



Active Queue

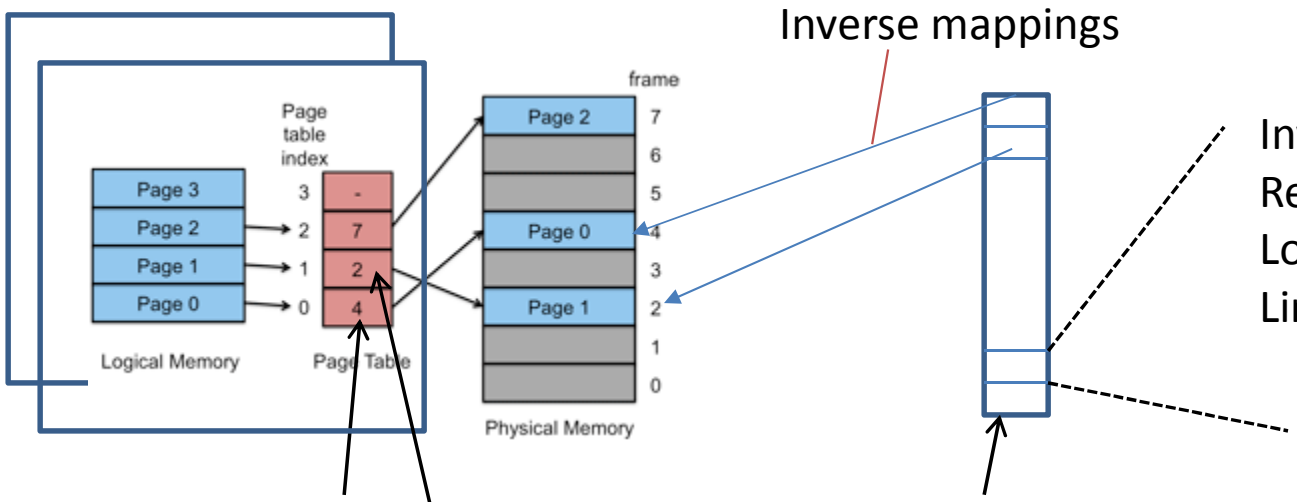


Inactive Queue

Maintain Ratio
Inactive Queue .. → proactively clean
Inactive Queues create minor faults

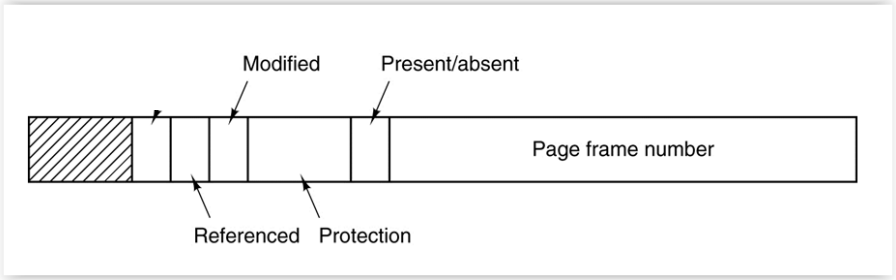
Minor Faults forced by setting present=0

Data Structures (also lab3)



PageTable (one per process)

PageTable Entry (one per virtual page)



FrameTable:

- This is a data structure the OS maintains to track the usage of each frame by a pagetable (speak reverse mapping).
- one entry related to each physical frame)

Lab3 (how to approach)

- You are emulating HW / SW behavior:
- For every instruction simulated
  - Check whether PTE is present
  - If not present → pagefault
    -> OS must resolve:
    - select a frame to replace (make pluggable with different replacement algorithms)
    - Unmap its current user (UNMAP)
    - Save frame to disk if necessary (OUT)
    - Fill frame with proper content of current instruction's address space (IN,ZERO)
    - Map its new user (MAP)
    - Its now ready for use and instruction
  - Mark reference/modified bit as indicated by instruction

Conclusions

- We are done with Chapter 3
- Main goal
  - Provide CPU with illusion of large and fast memory
- Constraints
  - Speed
  - Cost
  - Protection
  - Transparency
  - Efficiency
- Memory management
  - Paging
  - Segmentation
  - Paged segments