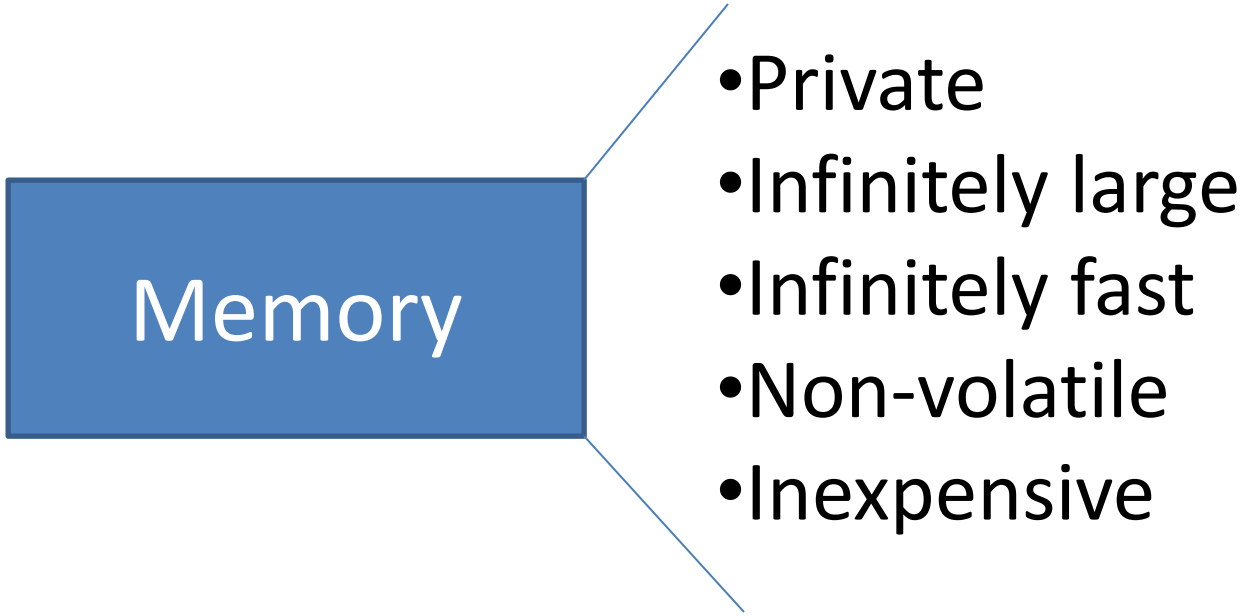
CSCI-GA.2250-001

Operating Systems

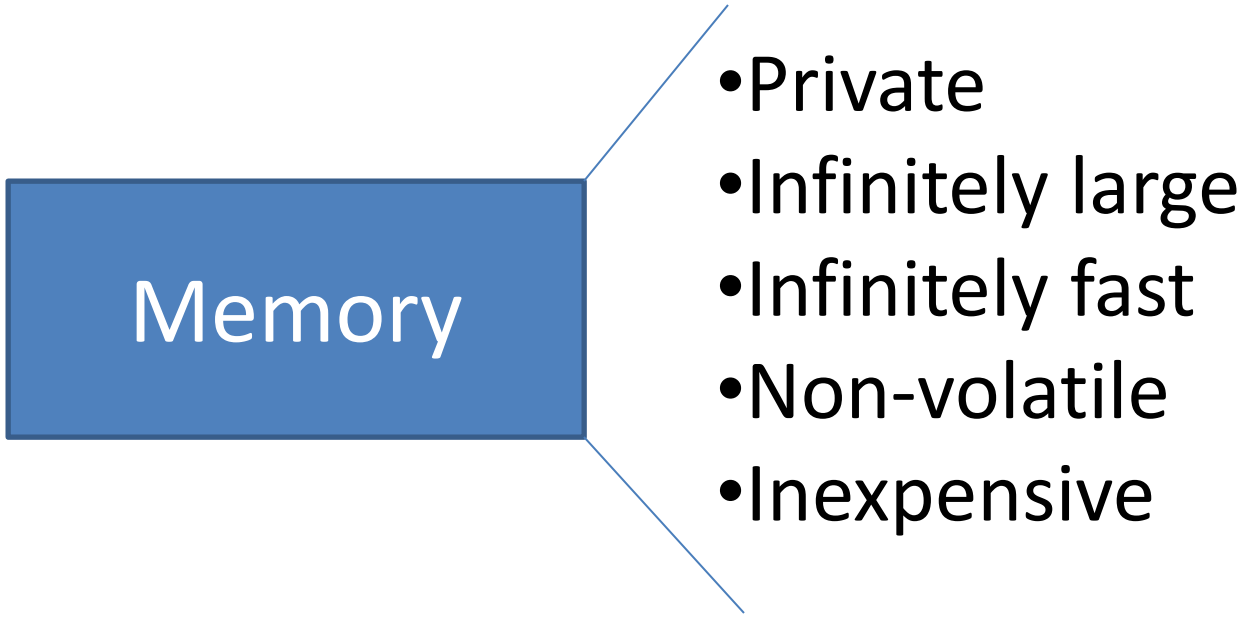
Lecture 5: Memory Management

Hubertus Franke
frankeh@cims.nyu.edu

Programmer's dream

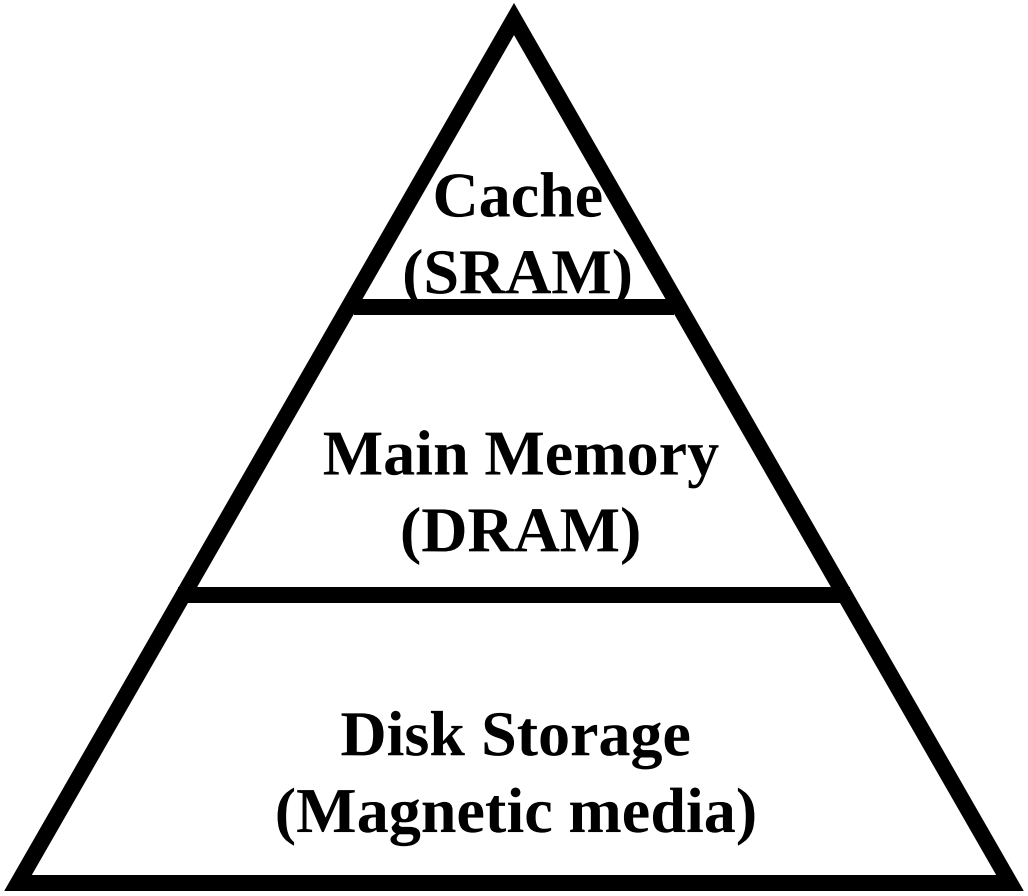


Programmer's Wish List

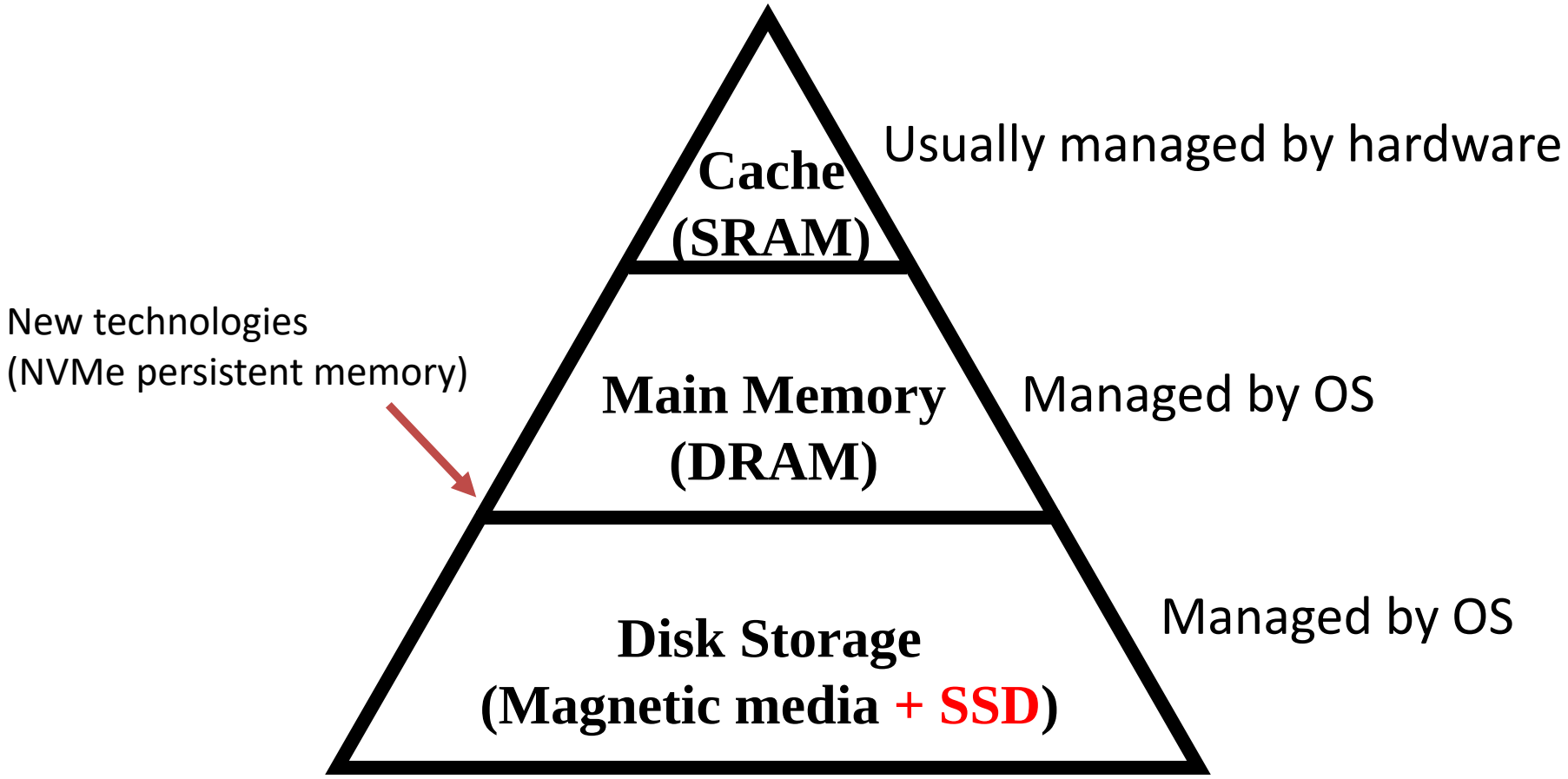


Programs are getting bigger faster than memories.

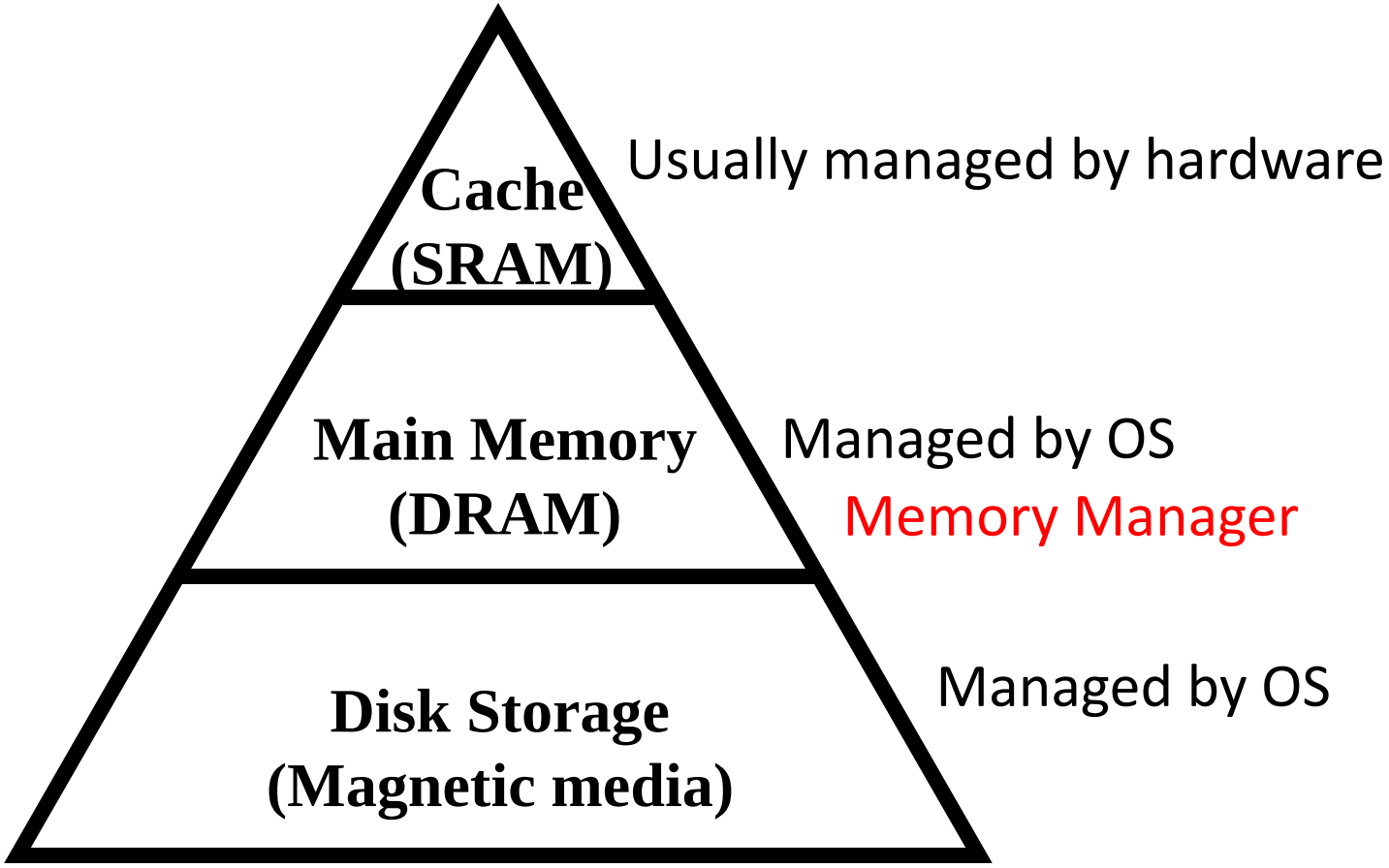
Memory Hierarchy



Memory Hierarchy



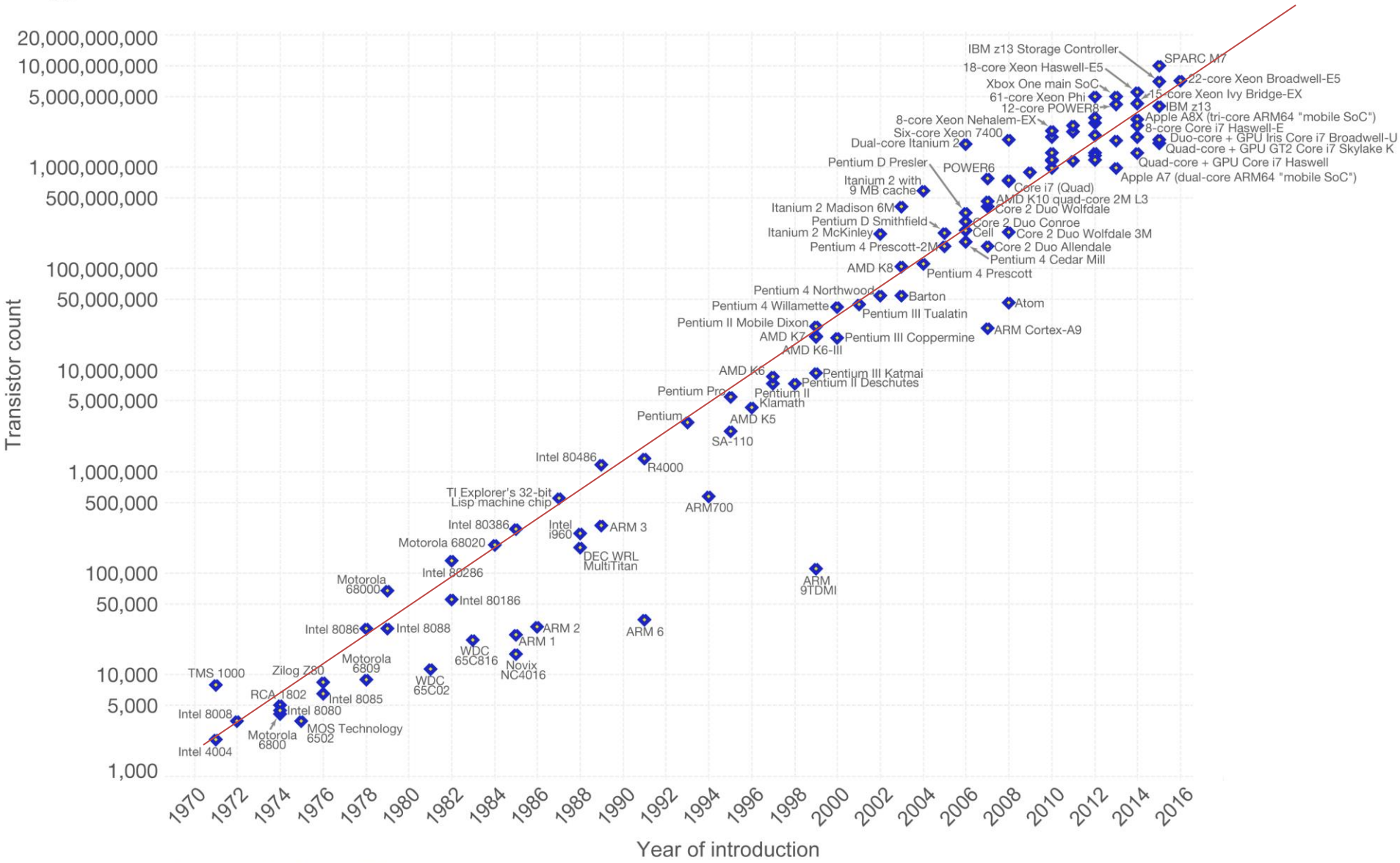
Memory Hierarchy



Moore’s Law – The number of transistors on integrated circuit chips (1971-2016)

Our World in Data

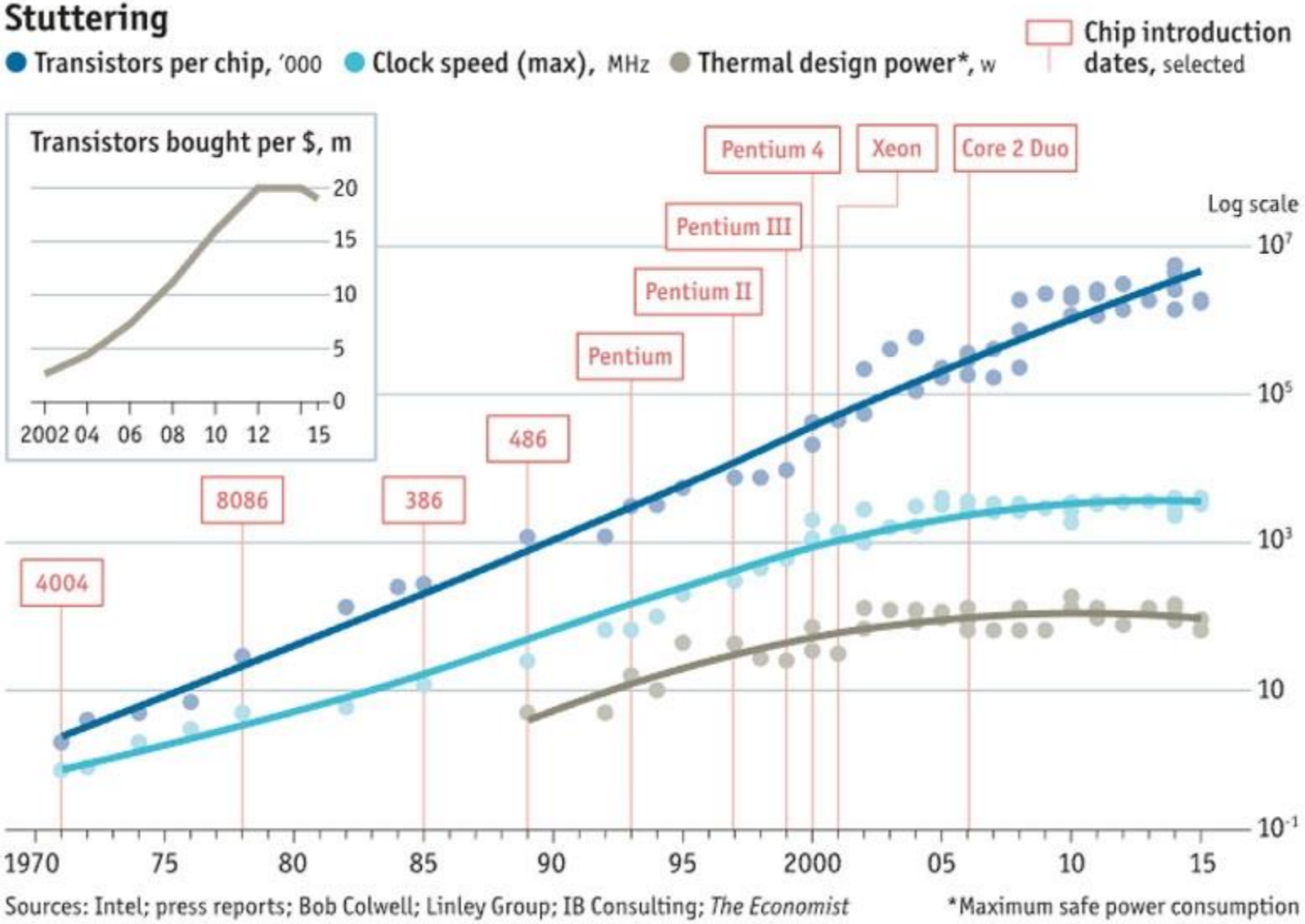
Moore's law describes the empirical regularity that the number of transistors on integrated circuits doubles approximately every two years. This advancement is important as other aspects of technological progress – such as processing speed or the price of electronic products – are strongly linked to Moore's law.



Data source: Wikipedia (https://en.wikipedia.org/wiki/Transistor_count)

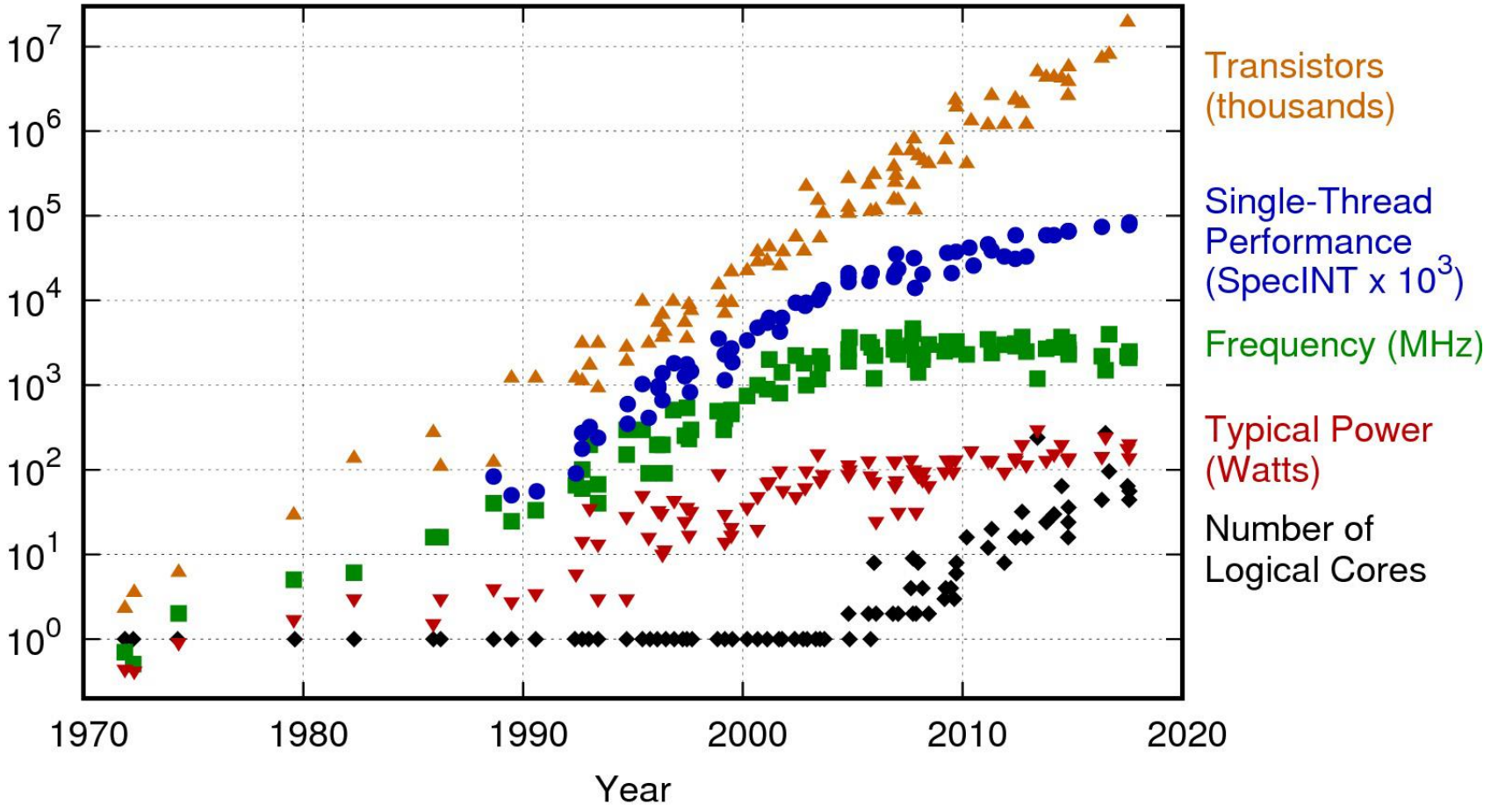
The data visualization is available at OurWorldinData.org. There you find more visualizations and research on this topic.

Stuttering



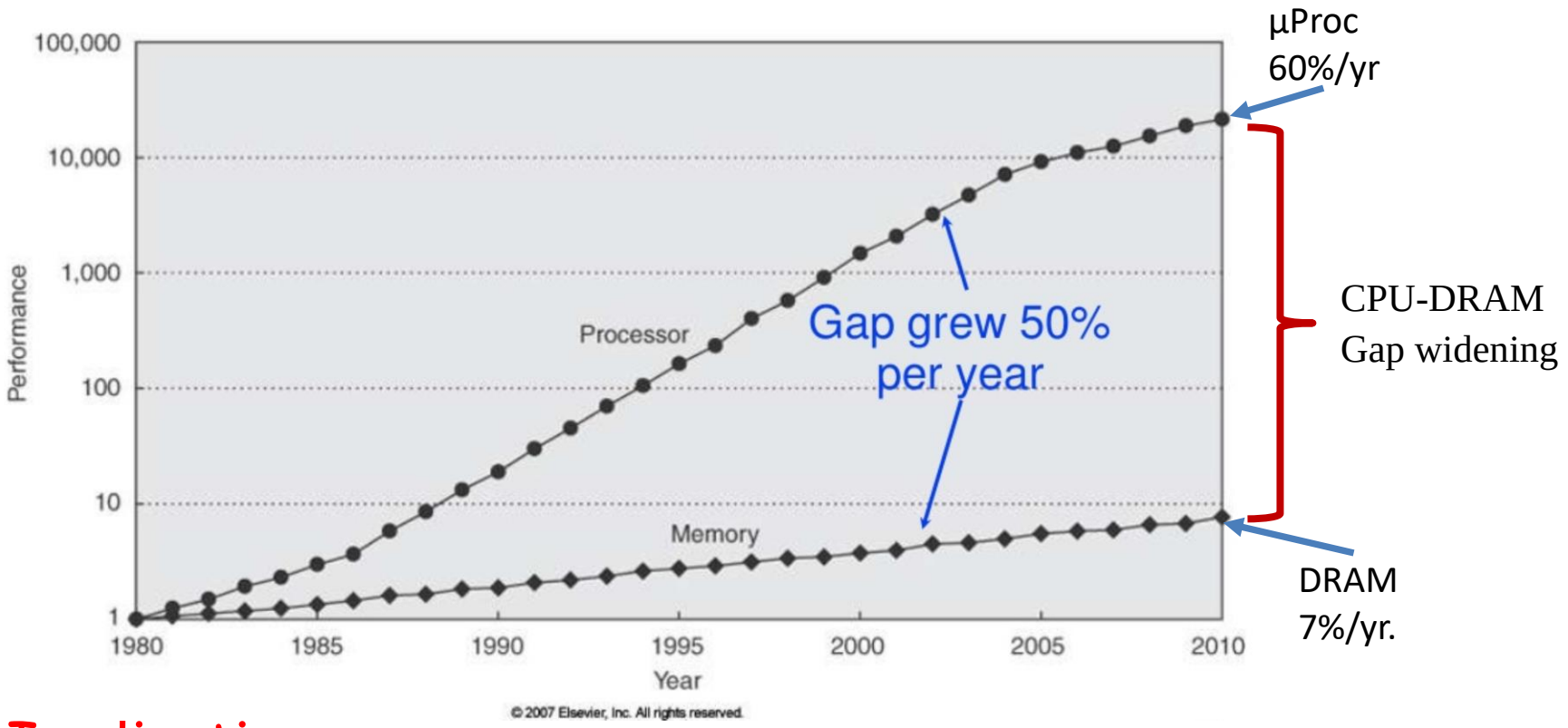
Sources: Intel; press reports; Bob Colwell; Linley Group; IB Consulting; *The Economist*

*Maximum safe power consumption



Original data up to the year 2010 collected and plotted by M. Horowitz, F. Labonte, O. Shacham, K. Olukotun, L. Hammond, and C. Batten
New plot and data collected for 2010-2017 by K. Rupp

Question: Who Cares About the Memory Hierarchy?



Implications:

- Longer per cycle memory access times → Memory seems "further away"
- New Technologies: SDR -> DDR-1/2/3/4/5 -> QDR (technologies)

  Still problem widens

Memory Abstraction

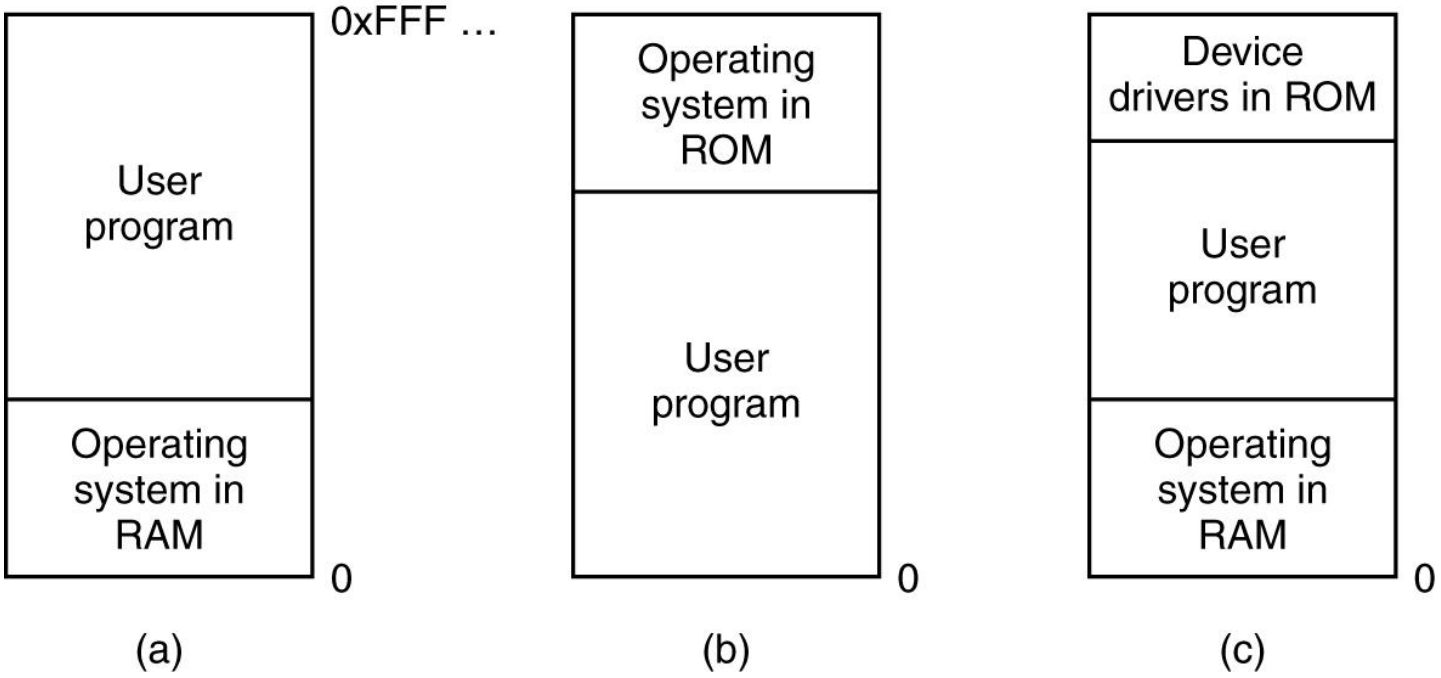
- The hardware and OS memory manager makes you see the memory in your process as a single contiguous entity

- How do they do that?
  - Abstraction

- Multiple processes of the same program see the same address space (in principle).
  - Addresses start at 0 .. 2^^N (N=32 or N=64)

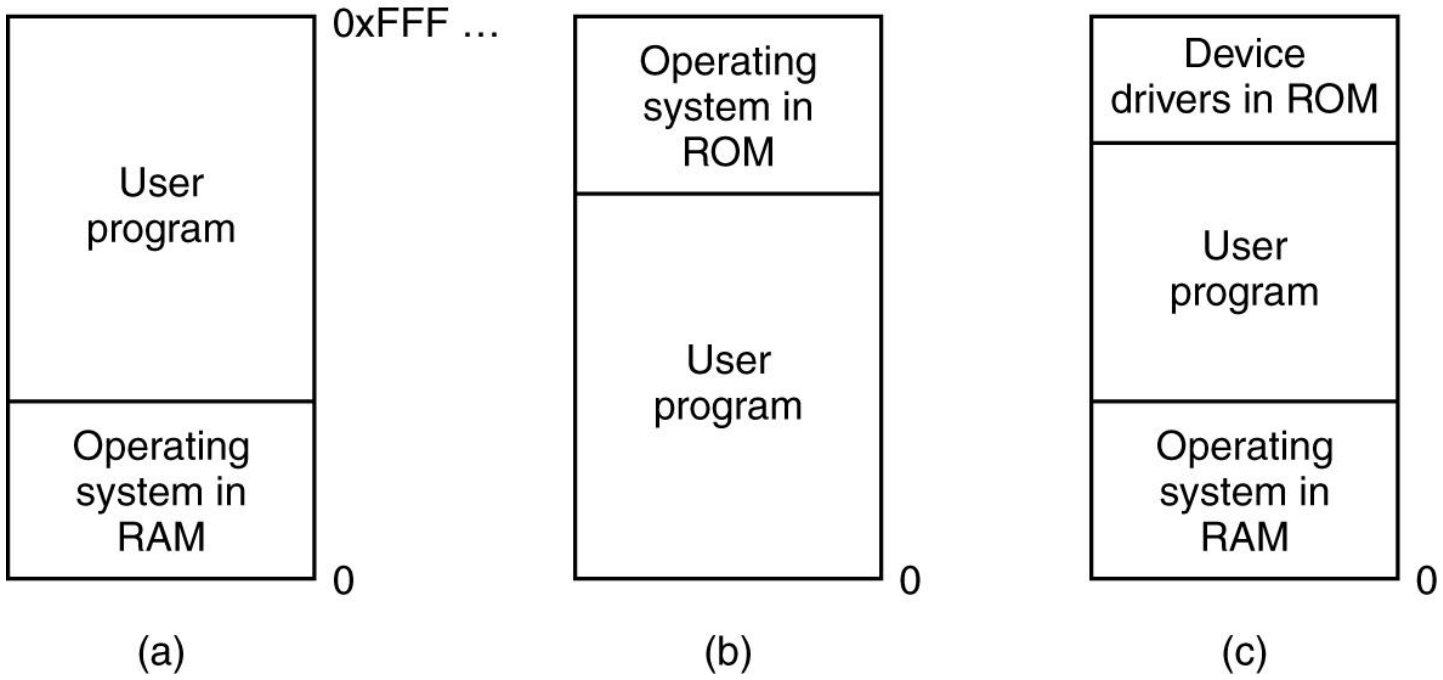
Is abstraction really necessary?

No Memory Abstraction



Even with no abstraction, we can have several setups!

No Memory Abstraction



Only one process at a time can be running
(remember threads share the address space of their process)

No Memory Abstraction

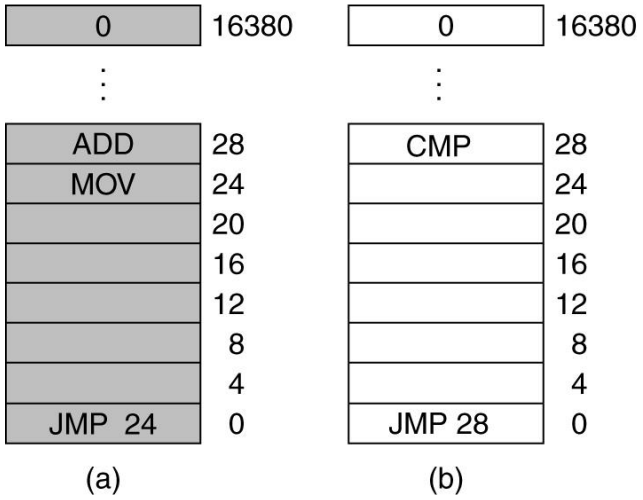
- What if we want to run multiple programs?
  - OS saves entire memory on disk
  - OS brings next program
  - OS runs next program

  } **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

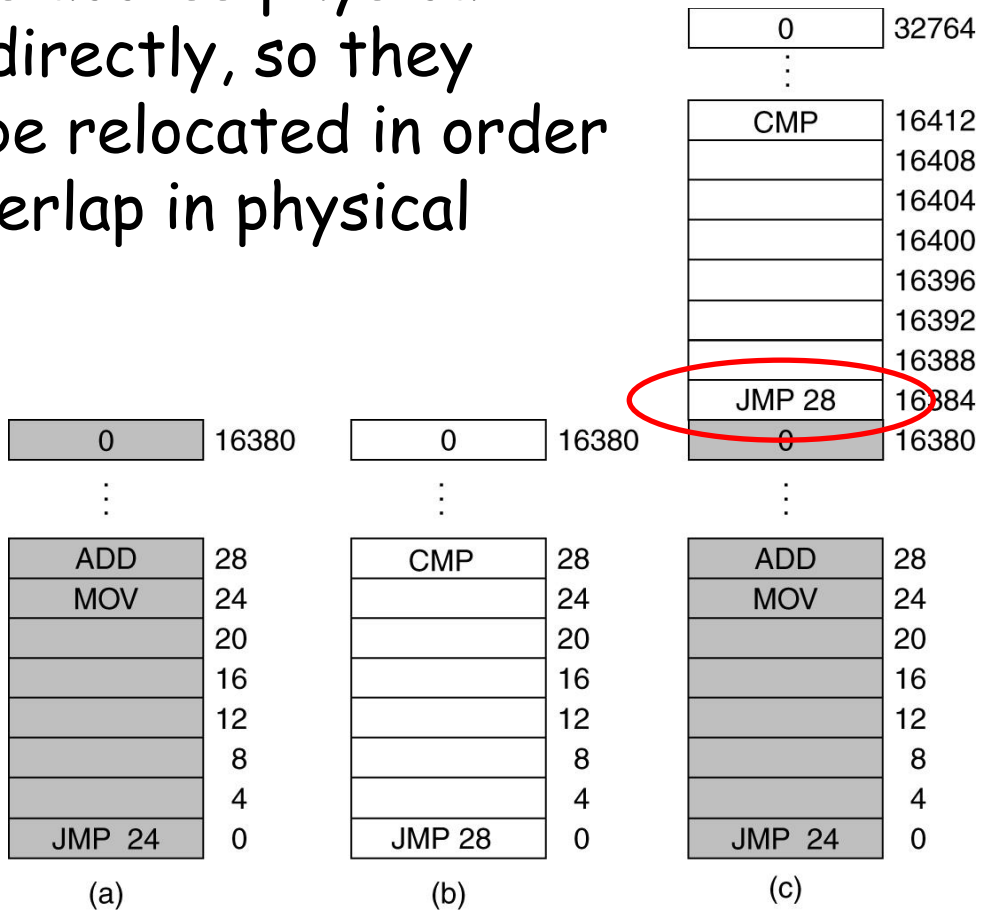
No Memory Abstraction (in the presence of multiple processes)

- Processes access physical memory directly



No Memory Abstraction (in the presence of multiple processes)

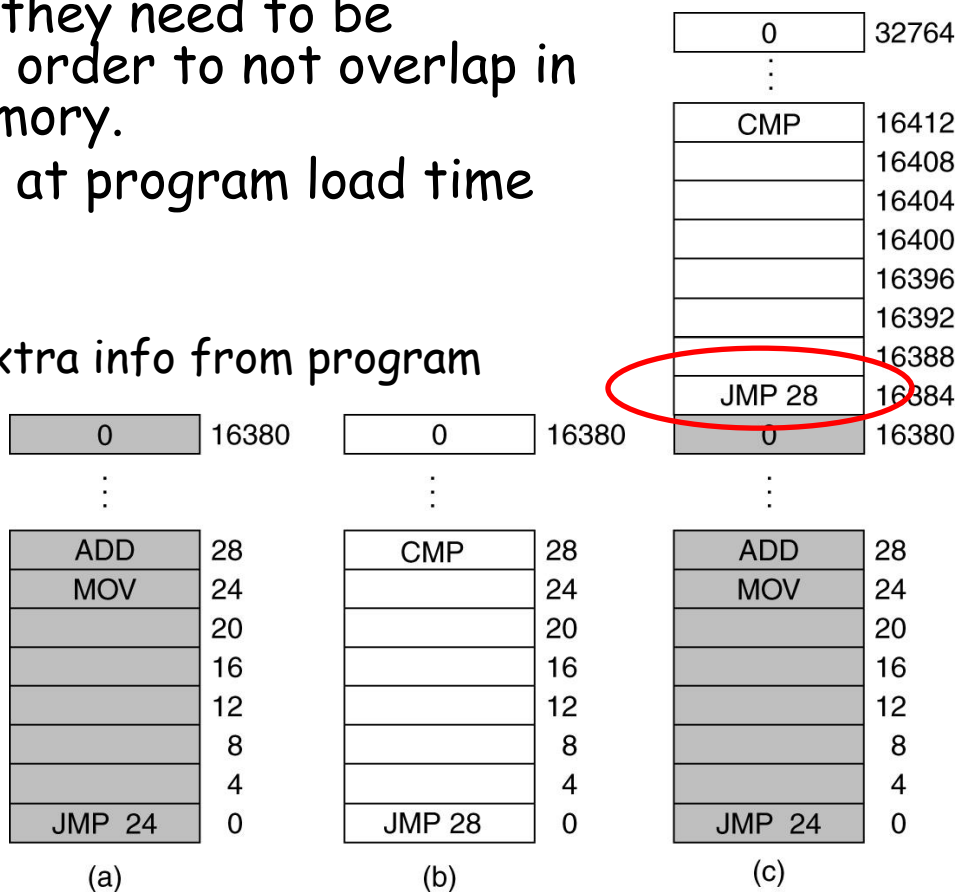
- Processes access physical memory directly, so they need to be relocated in order to not overlap in physical memory.



Using absolute address is wrong here

No Memory Abstraction (in the presence of multiple processes)

- Processes access physical memory directly, so they need to be relocated in order to not overlap in physical memory.
- Can be done at program load time
- Bad idea:
  - very 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
- Definition:

  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 (load / store) 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

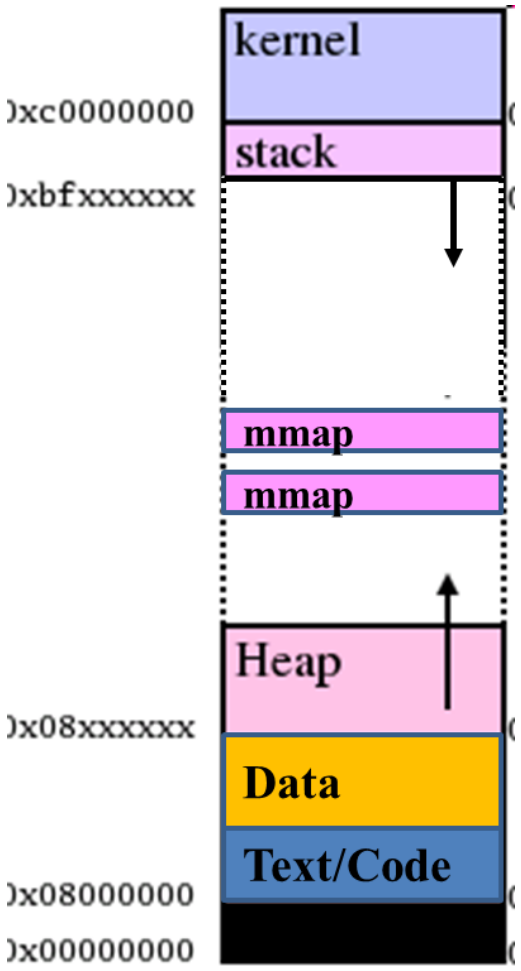
- Memory management must allow controlled access to shared areas of memory without compromising protection

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

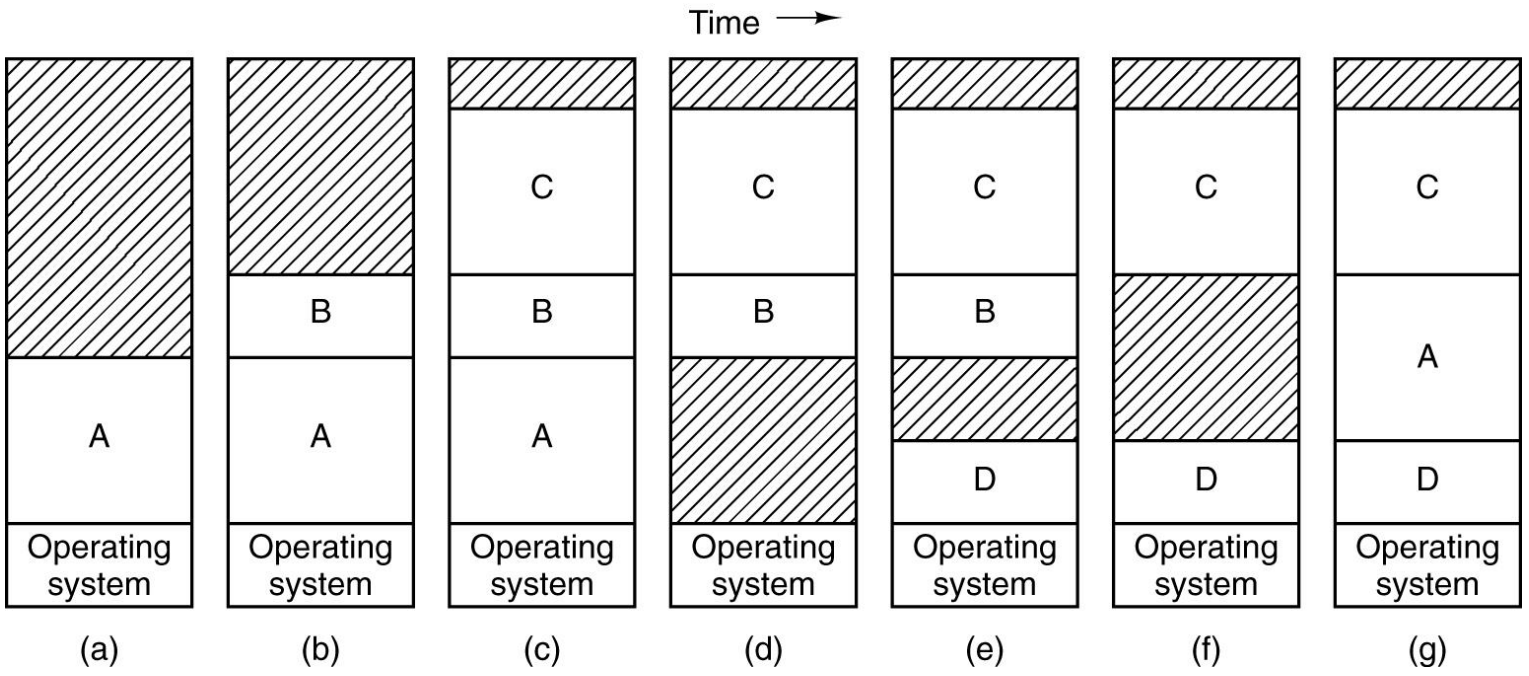
- Map each process address space onto a different part of physical memory
- Two registers: Base and Limit
  - **Base**: start address of a program in physical memory
  - **Limit**: length of the program
- For every memory access
  - Base is added to the address
  - Result compared to Limit
- Only OS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

Address Space: Base and Limit

- Need to add and compare for each memory address:
  - can be done in HW
  - doesn't add significantly add to latency

- What if memory space is not enough for all programs?
  - We may need to swap some programs out of the memory.
  - remember swapping means moving entire program to disk ➔ expensive

Swapping

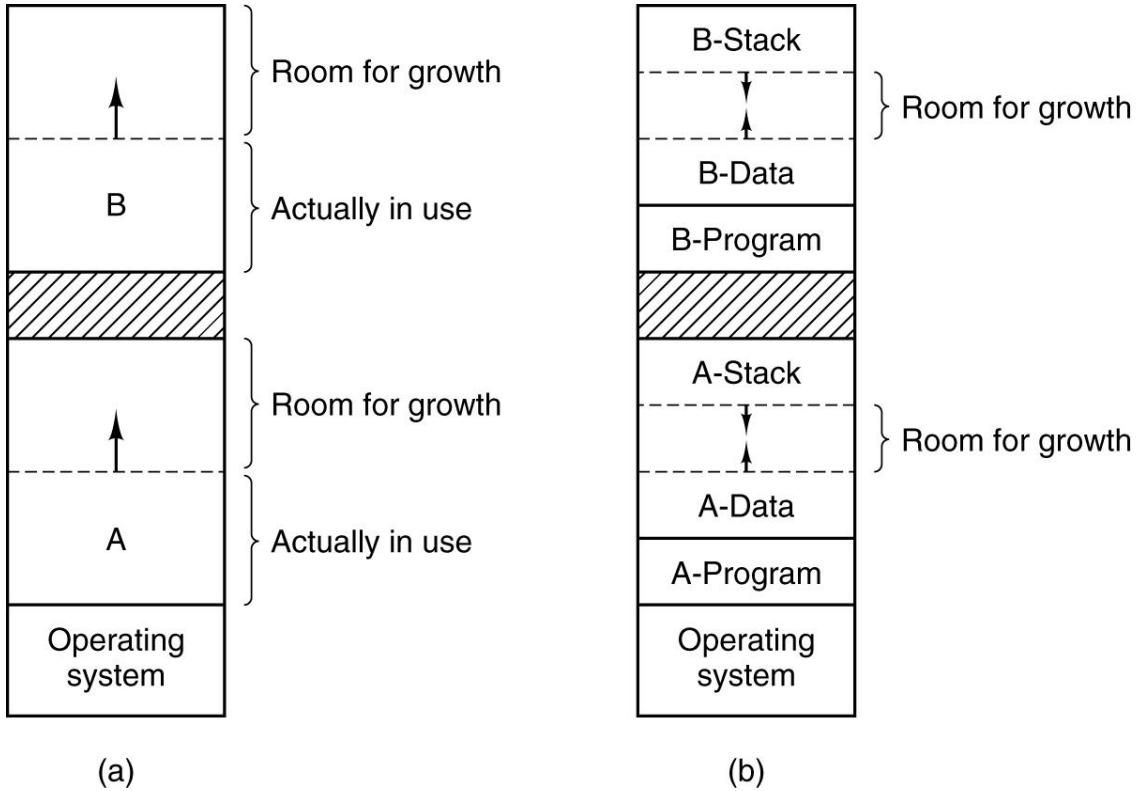


Swapping

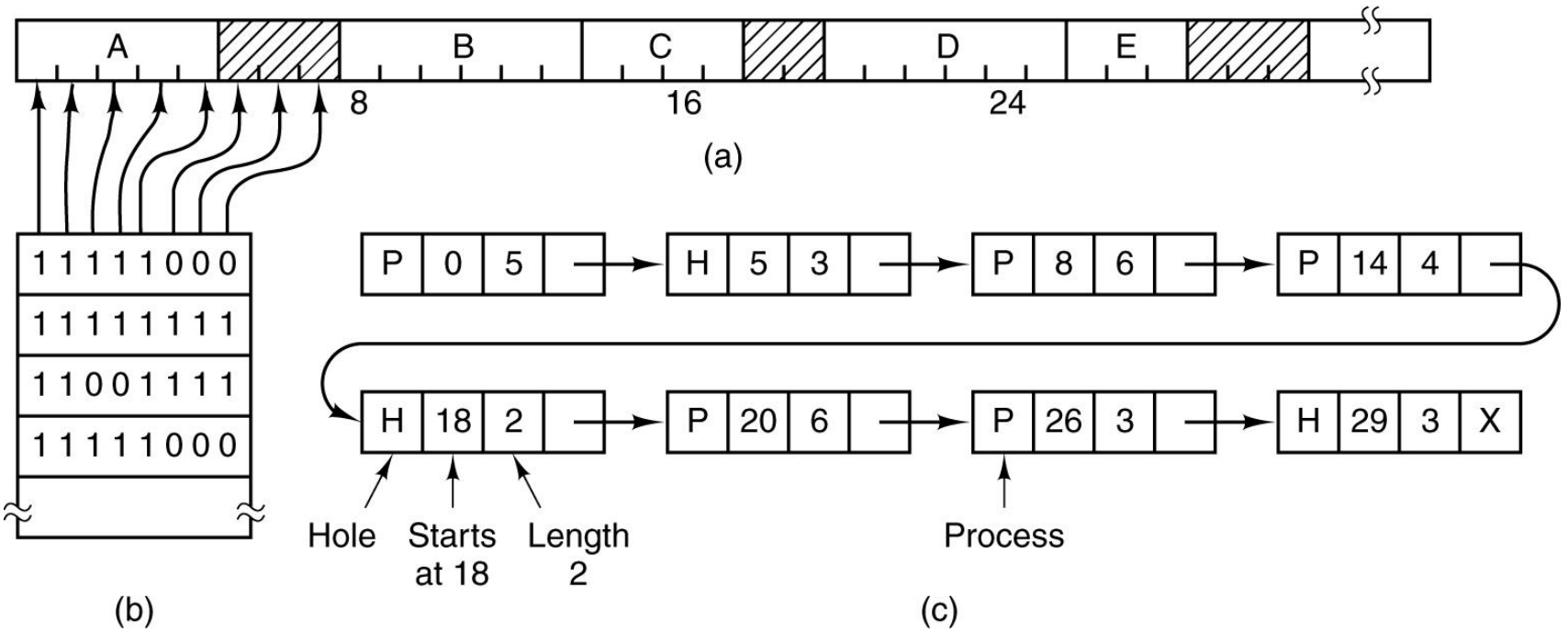
- Programs move in and out of memory
- Holes are created
- Holes can be combined -> memory compaction
- What if a process needs more memory?
  - If a hole is adjacent to the process, it is allocated to it
  - Process has to be moved to a bigger hole
  - Process suspended till enough memory is there

Dealing with unknown memory requirements by processes

- Anticipate growth of usable address space and leave gaps.
- Waste of phys memory as it will be allocated whether used or not.



Managing Free Memory

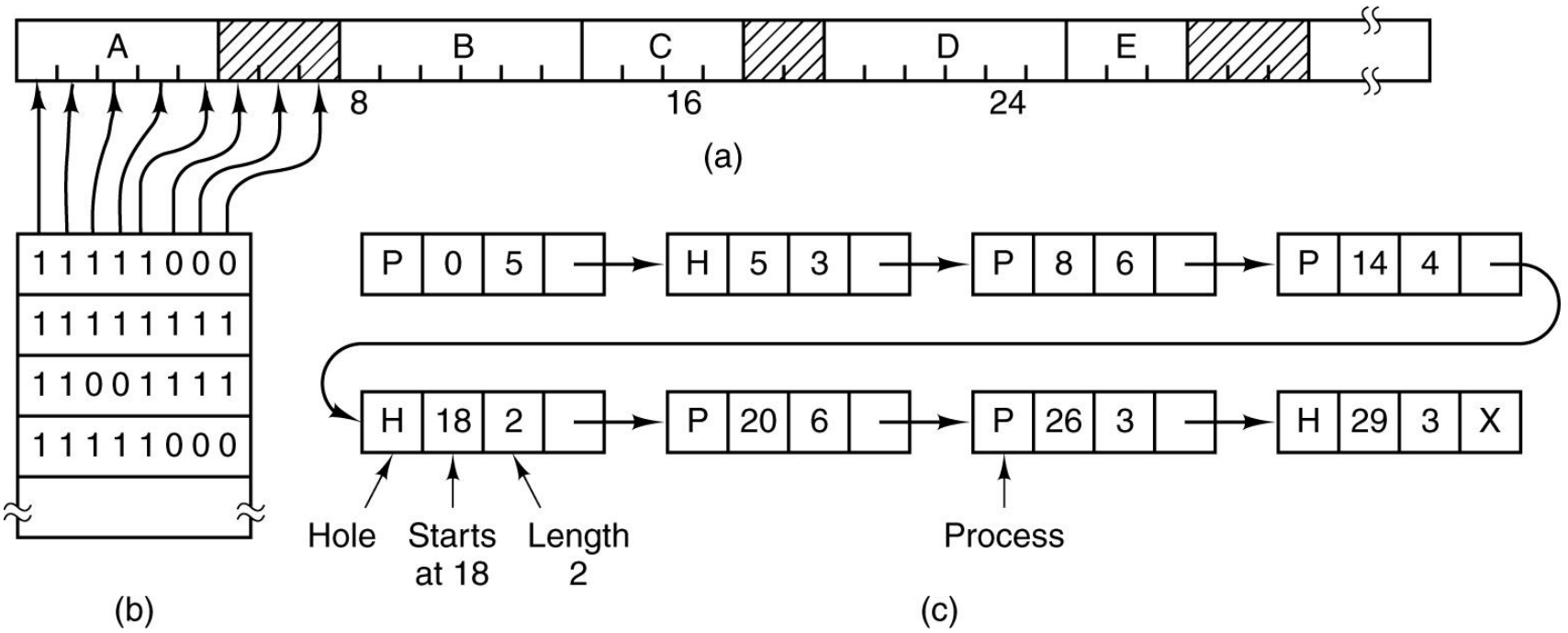


Bitmap

Linked List

- These are universal methods used in OS and applications. Other methods employ HEAPS.

Managing Free Memory



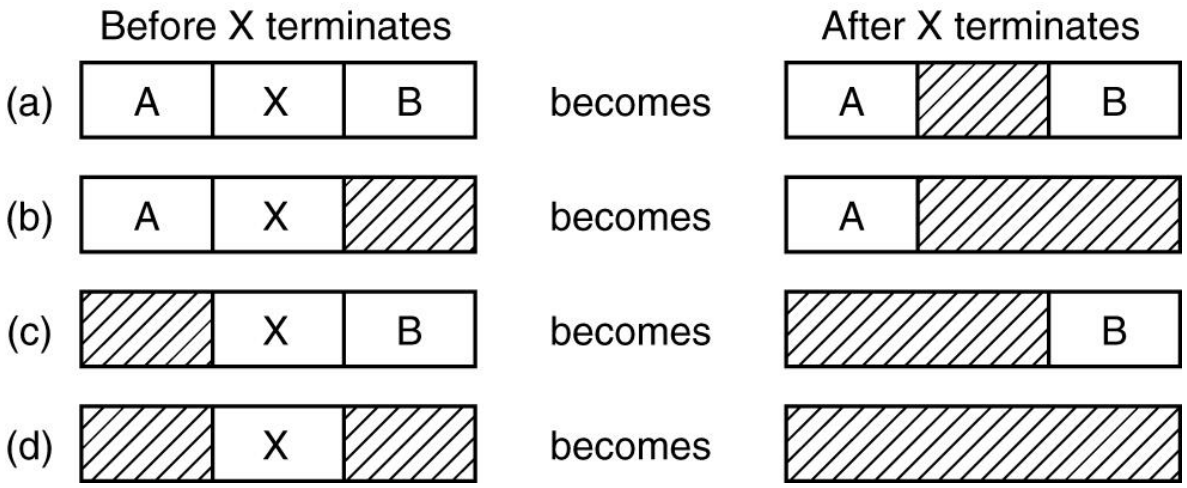
Bitmap

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 lists



Managing Free Memory: Which Available Piece to pick ?

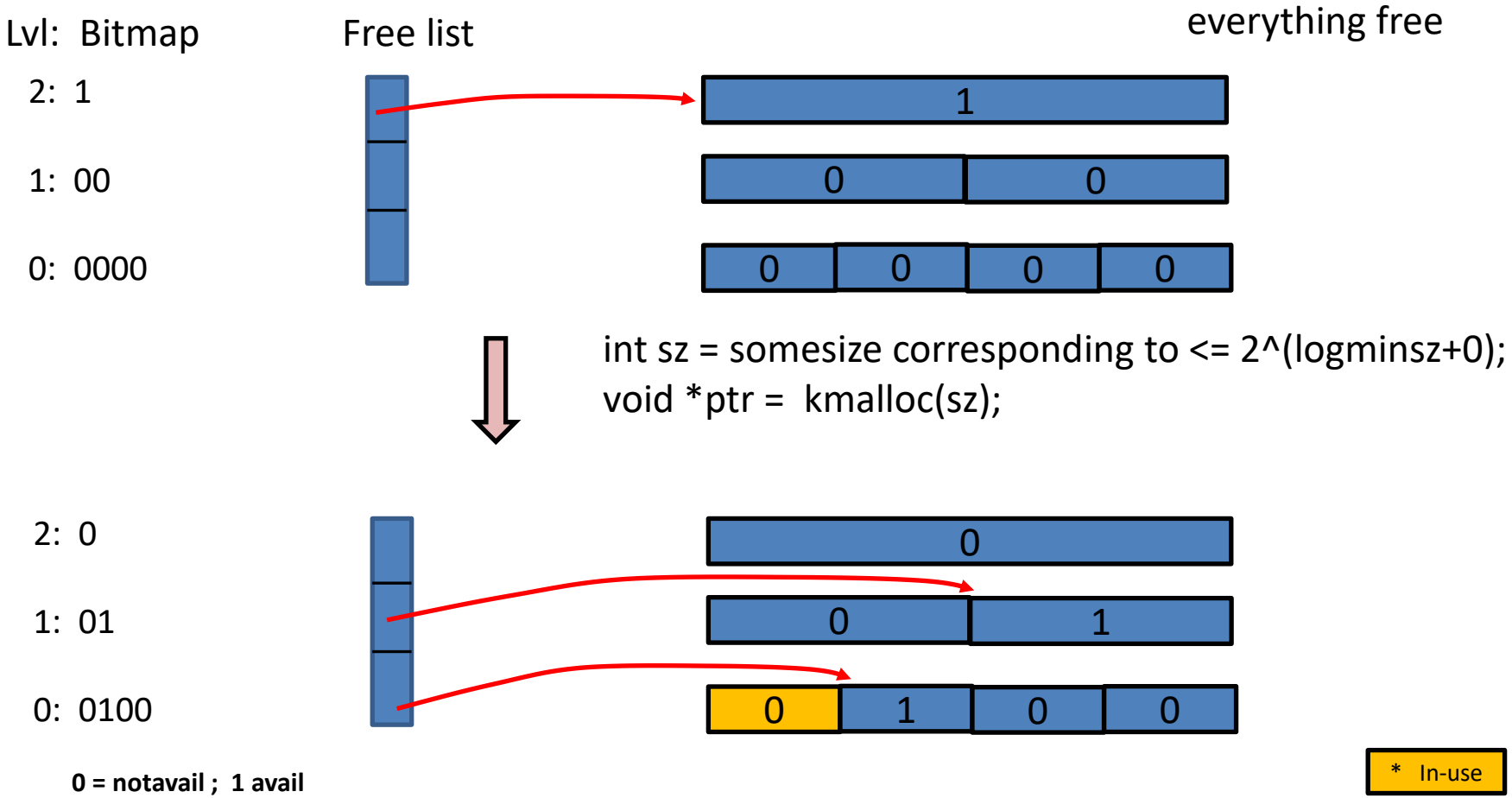
- How to allocate?
  - First fit
  - Best fit
  - Next fit
  - Worst fit
  - …
- Each has their philosophy to why and what their pros and cons are.
  At the end you must weight efficiency of space (fragmentation) and time (to manage)

Buddy Algorithm

- Considers blocks of memory only as 2^N
- Potential for fragmentation (drawback)
- If no block of a size is available it splits higher blocks into smaller blocks
- Easy to implement and fast
  O(log2(MaxBlockSz/MinBlockSize))
  e.g. 4K .. 128B = 2^(12-7) = 5 steps

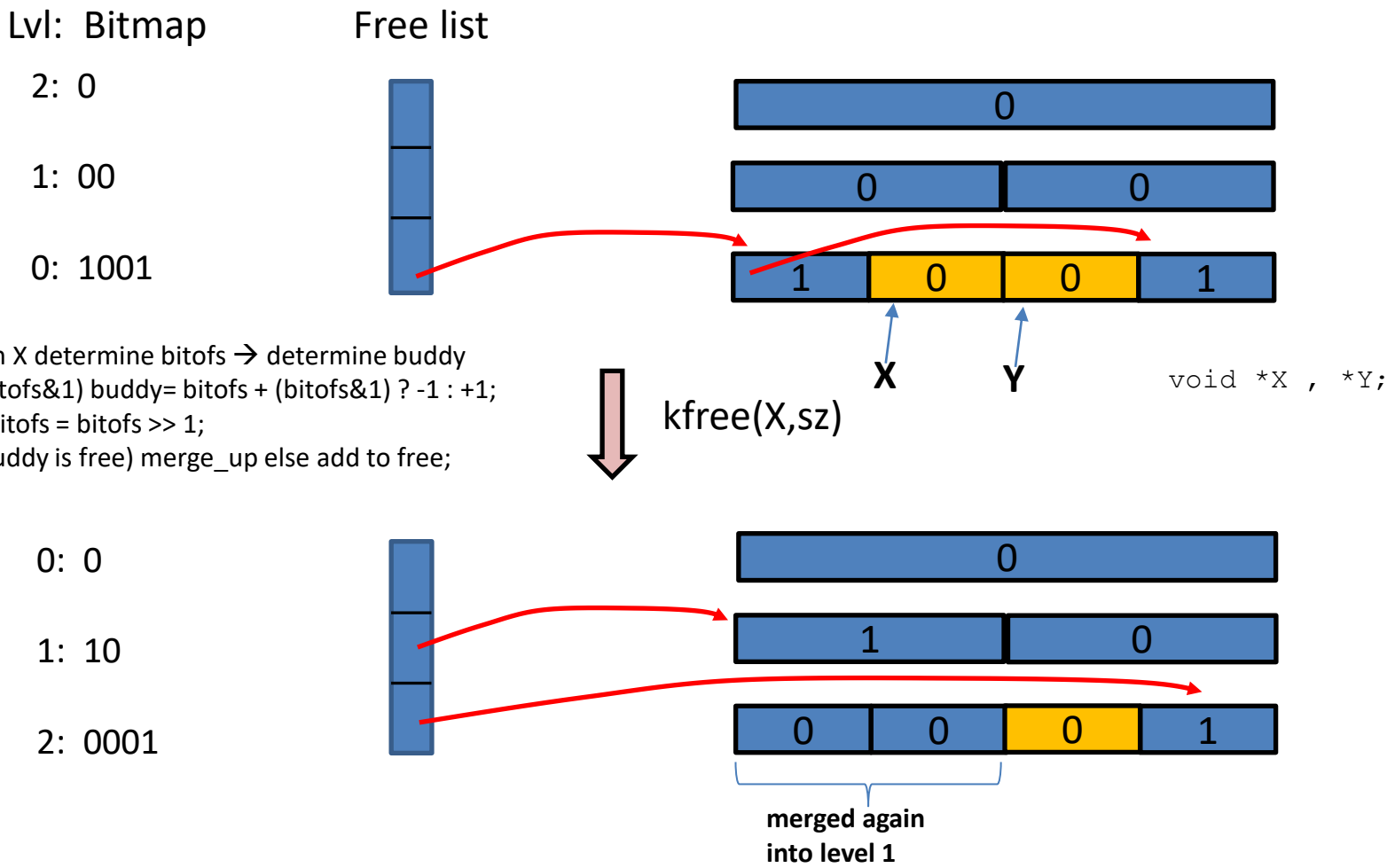
Buddy Algorithm

- Allocation at level 0



Buddy Algorithm

- Freeing "X" at level 2 leading to coalescing



What are really the problems?

- Memory requirement unknown for most apps
- Not enough memory
  - Having enough memory is not possible with current technology
    - How do you determine "enough"?
- Exploit and enforce one condition:

  Processor does not execute or access anything that is not in the memory
  (how would that even be possible ?)

  Enforce transparently (user is not involved)

Memory Management Techniques

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

But We Can See That …

- All memory references are logical addresses in a process's address space that are dynamically translated into physical addresses at run time

- An address space 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 an address space be in main memory during execution. Only the ones that I am "currently" accessing

Scientific Definition of Virtual Memory

Mapping from

logical (virtual) address space
(user / process perspective)

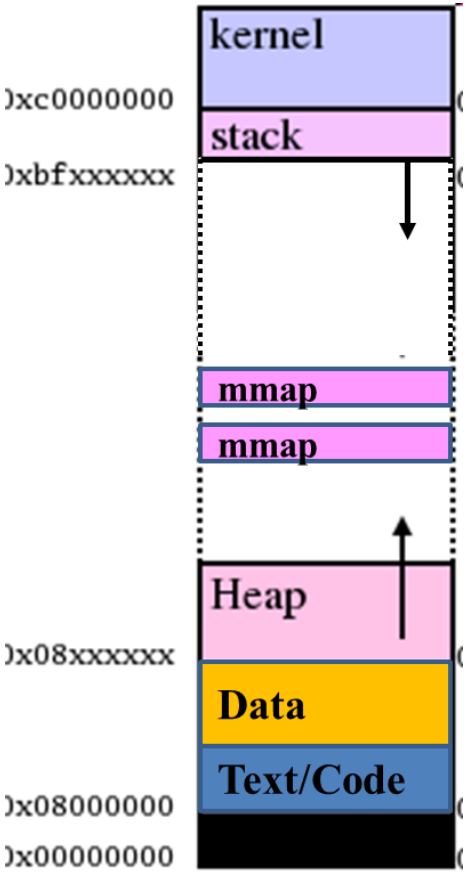
to

physical address space
(hardware perspective)

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 exception is generated when an address is needed that is not in main memory (will see how).
3. Operating system places the process in a blocking state.
4. Operating system allocates memory and optionally issues a disk I/O Read request to retrieve the content of that memory
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 has been brought into main memory.

The Story
Questions over Questions

1. Operating system brings into main memory a few pieces of the program.
   What do you mean by "pieces"?

2. An exception is generated when an address is needed that is not in main memory (will see how).
   How do you know it isn't in memory?

3. Operating system places the process in a blocking state.

4. Operating system allocates memory and optionally issues a disk I/O Read request to retrieve the content of that memory or set it otherwise.
   How do I know which one?
   What if memory is full and I can't allocate anymore ?

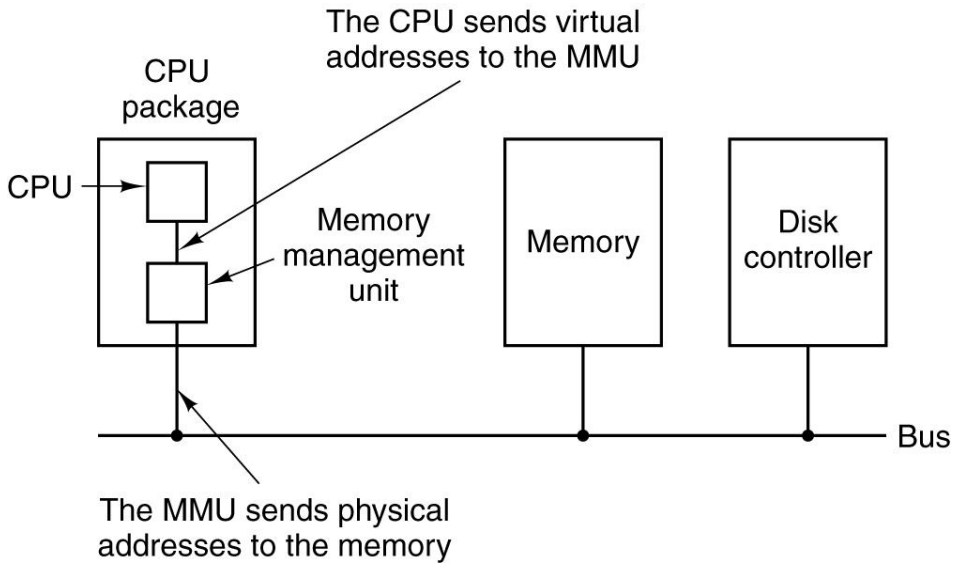
5. Another process is dispatched to run while the disk I/O takes place.

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 has been brought into main memory.

Virtual Memory

- Each program has its own address space
- This address space is divided into pages (e.g 4KB)
- Pages are mapped into physical memory chunks (called frames)
- By definition: sizeof(page) == sizeof(frame)
  (the size is determined by the hardware manufacturer)



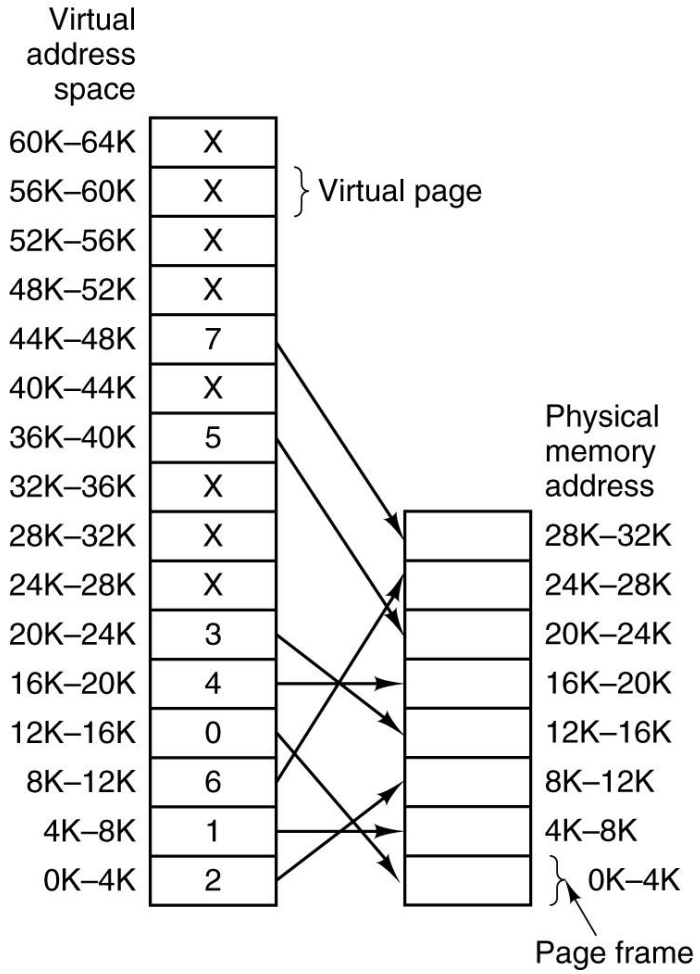
Virtual Memory

Single Process / Address Space view first

Allows to have larger virtual address space then physical memory available

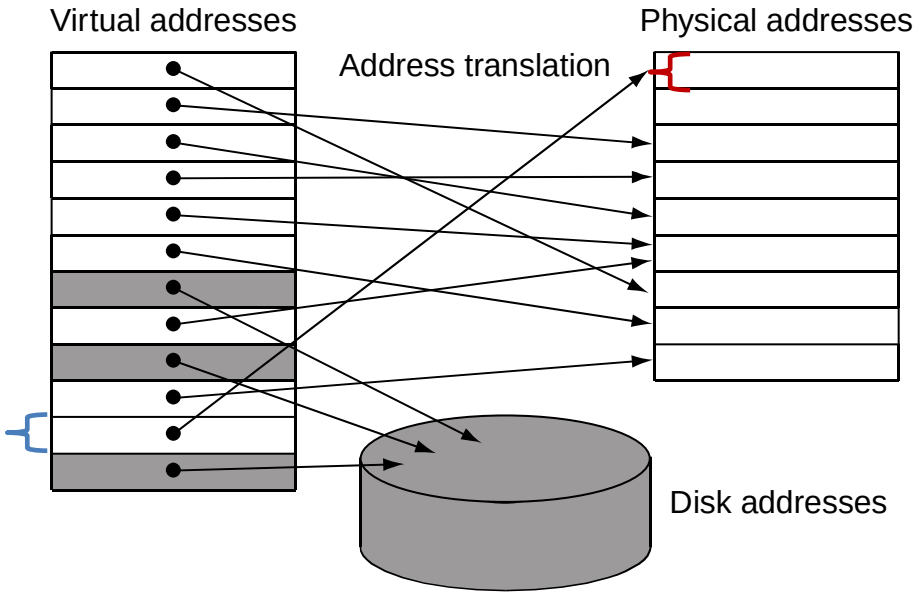
"X" → when you access a virtual page you get a page fault (see prior story) that needs to be resolved first

All other accesses are at hardware speed and don't require any OS intervention (with the exception of protection enforcement)

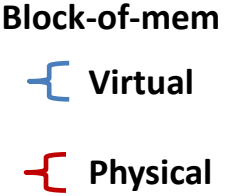


Virtual Memory

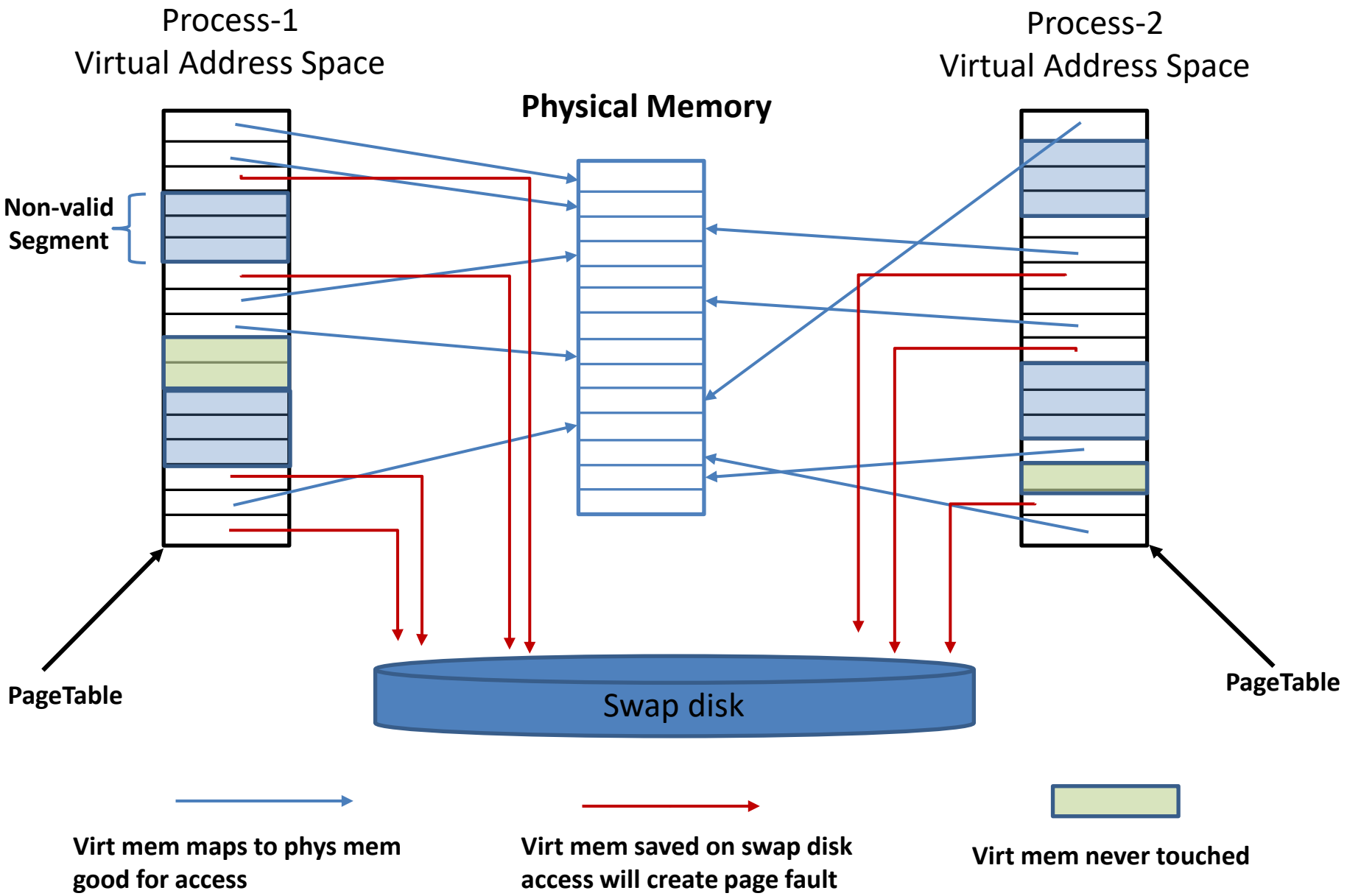
- Main memory can act as a cache for the secondary storage (disk)



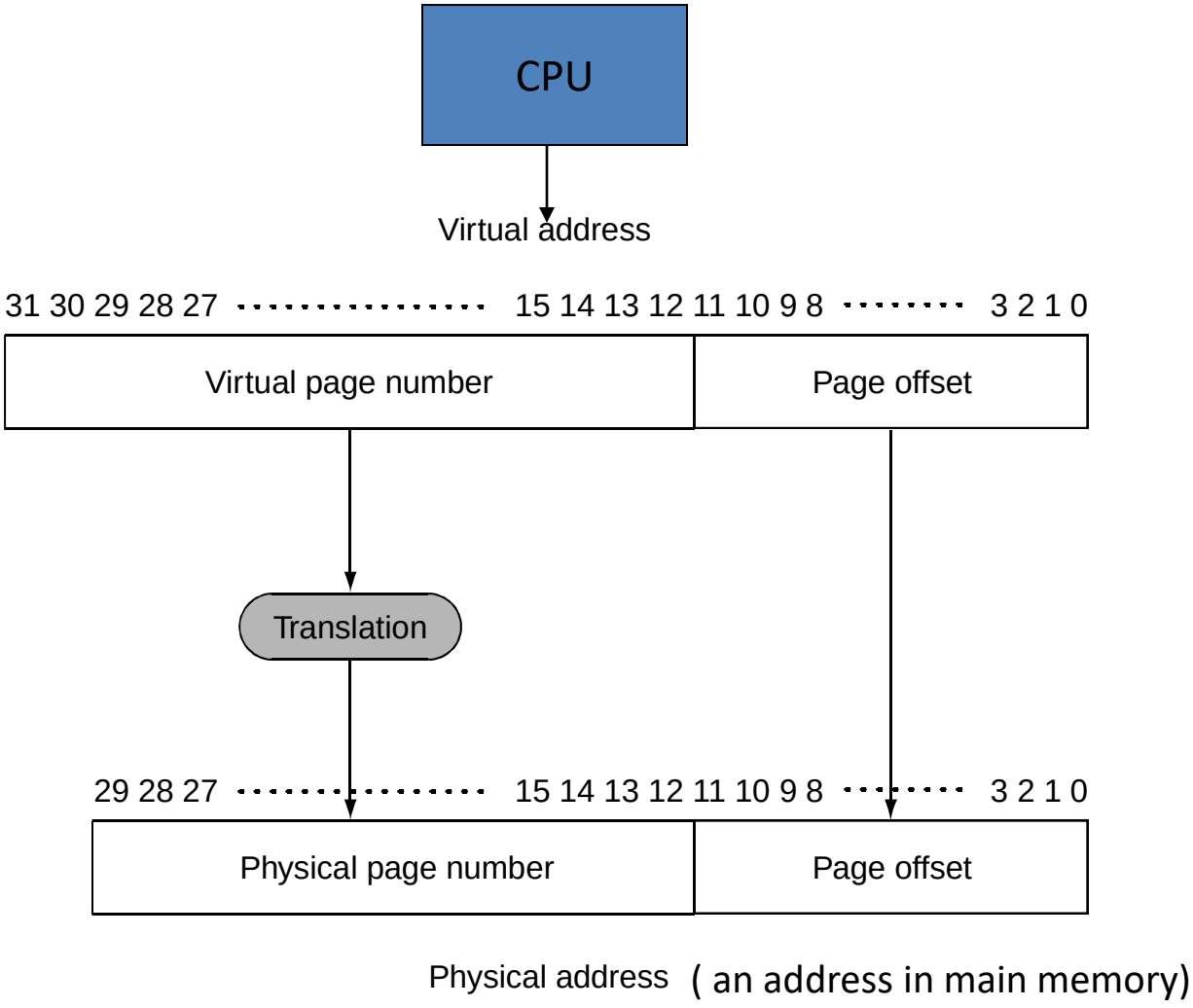
- Advantages:
  - illusion of having more physical memory
  - program relocation
  - protection



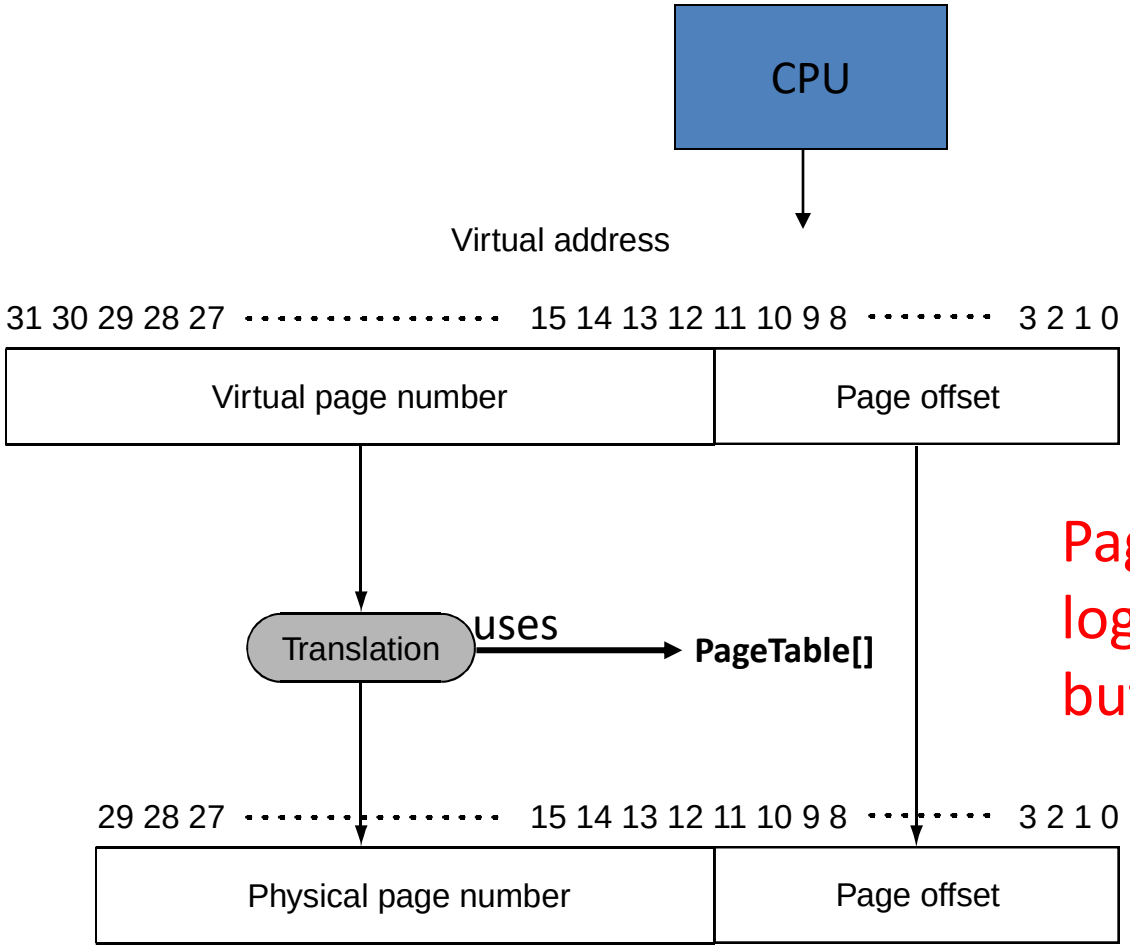
Virtual Memory



How does address translation work?



Physical address (an address in main memory)

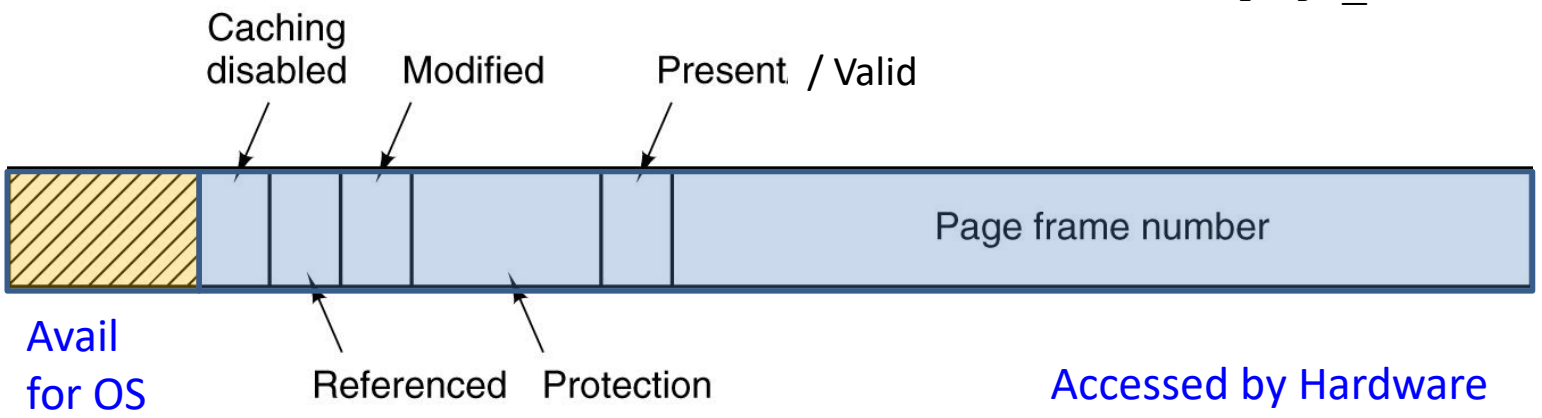


Page table referenced logically inside MMU, but is really in memory

MMU = Memory Management Unit

Structure of a Page Table Entry

```
struct PTE page_table[];
```

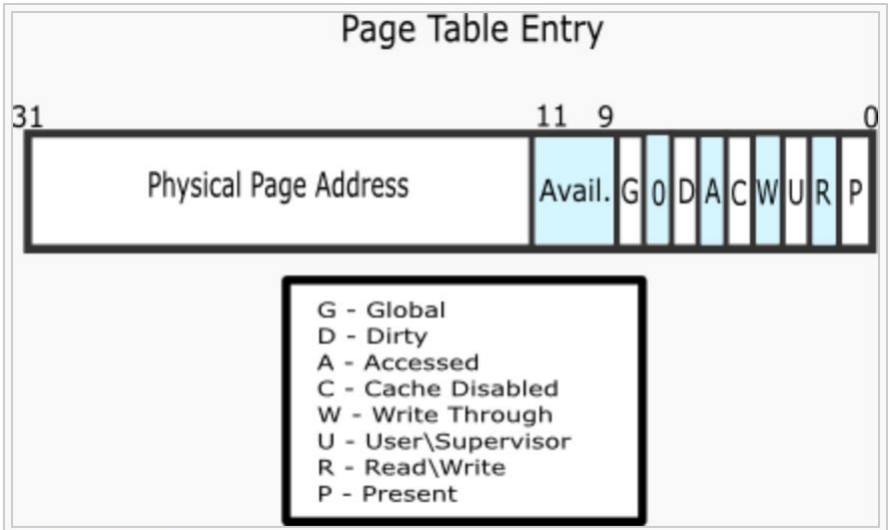


- **Present/Valid bit**: '1' if the values in this entry is valid, otherwise translation is invalid and a pagefault exception will be raised
- **Frame Number:** this is the physical frame that is accessed based on the translation.

- **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

- Other unused bits typically available for the operating system to "remember" information in

X86 examples for PTE



A Page Table Entry

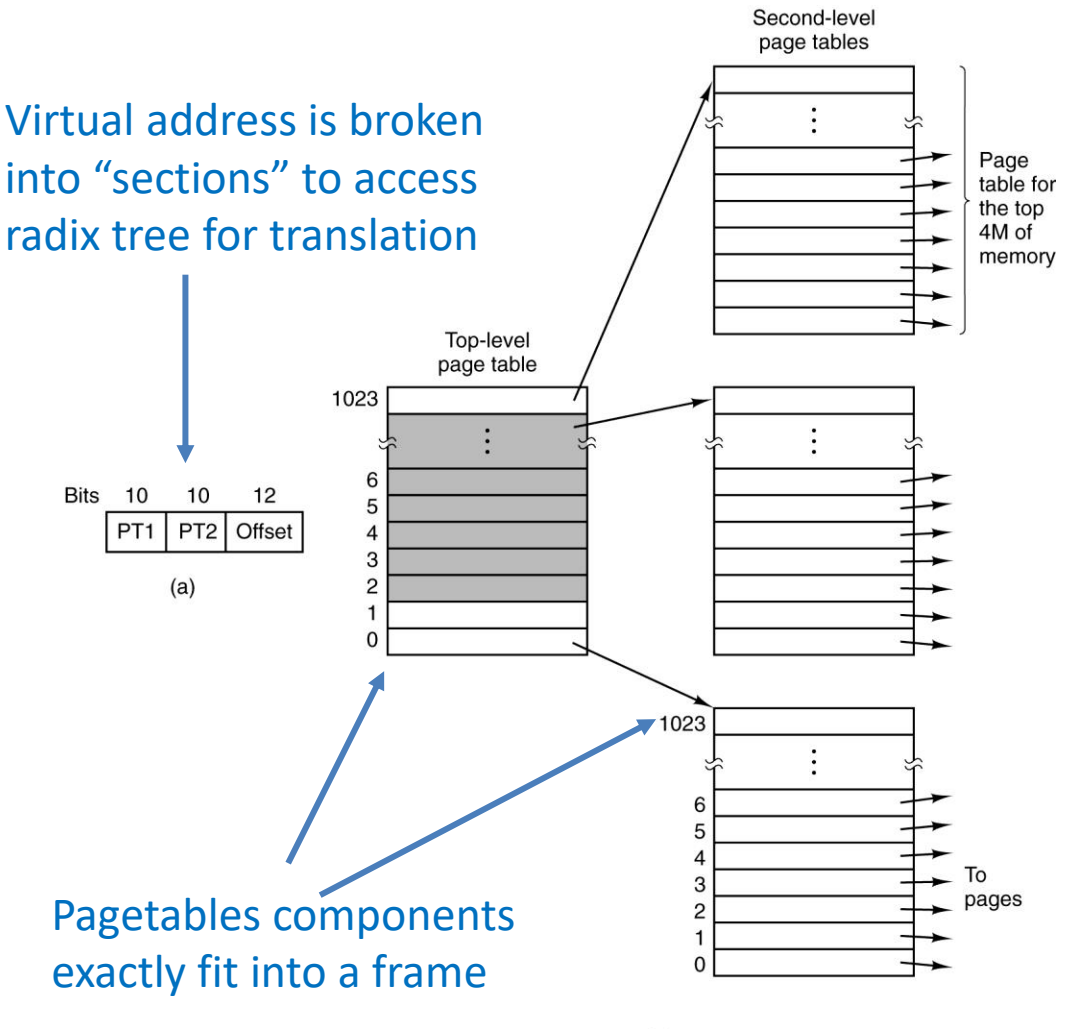
32-bit

```
typedef struct
{
  uint64 present                    :1;
  uint64 writeable                  :1;
  uint64 user_access                :1;
  uint64 write_through              :1;
  uint64 cache_disabled             :1;
  uint64 accessed                   :1;
  uint64 ignored_3                  :1;
  uint64 size                       :1; // must be 0
  uint64 ignored_2                  :4;
  uint64 page_ppn                   :28;
  uint64 reserved_1                 :12; // must be 0
  uint64 ignored_1                  :11;
  uint64 execution_disabled         :1;
} __attribute__((__packed__)) PageMapLevel4Entry;
```

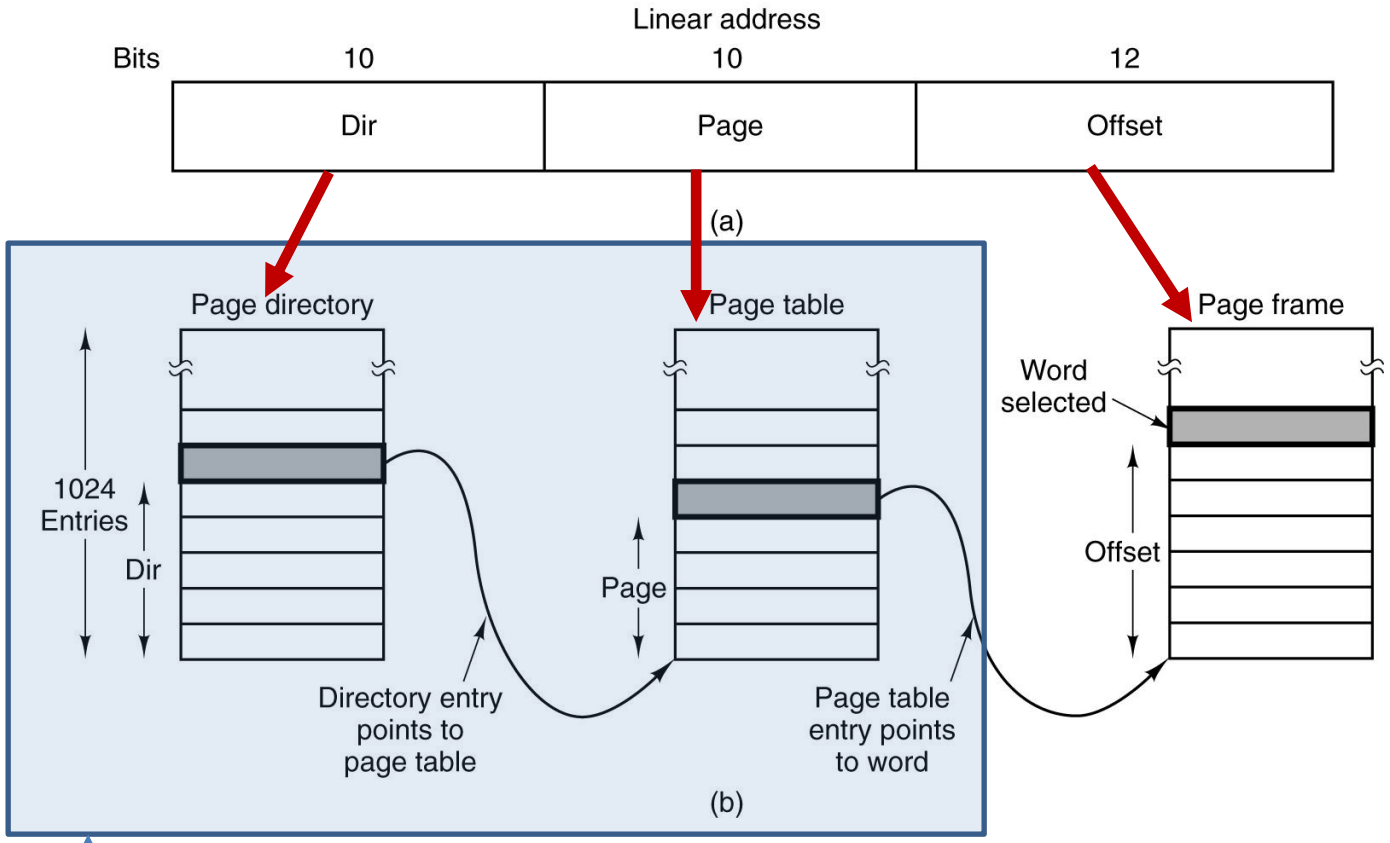
64-bit

Multi-Level Page Table
aka
RadixTree or Hierarchical Page Table

- To reduce storage overhead in case of large memories
- For sparse address spaces (most processes) only few 2nd tables required !!!



The Intel Pentium



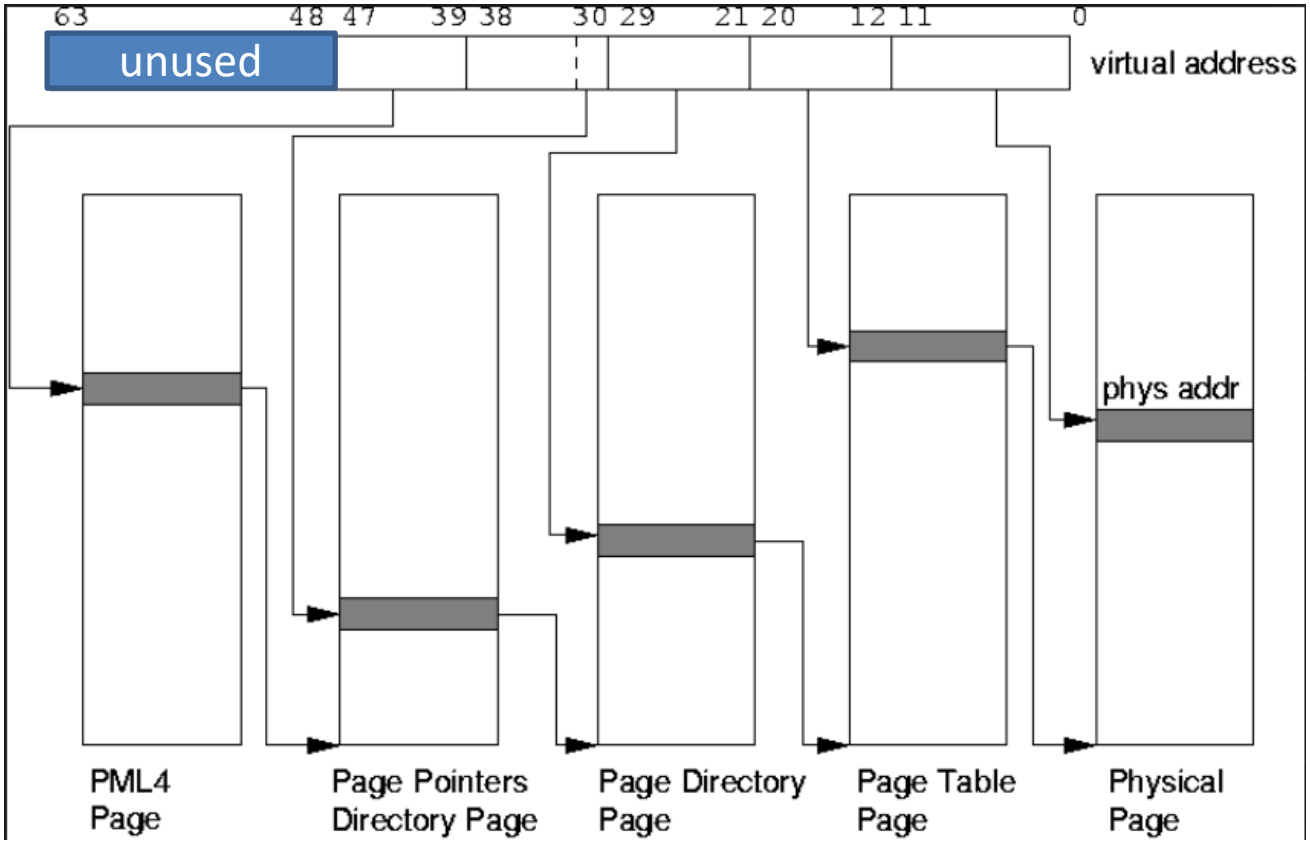
In its entirety this hierarchy is the "page table"

CR3

Each running program has a page table.

CR3: Privileged hardware register

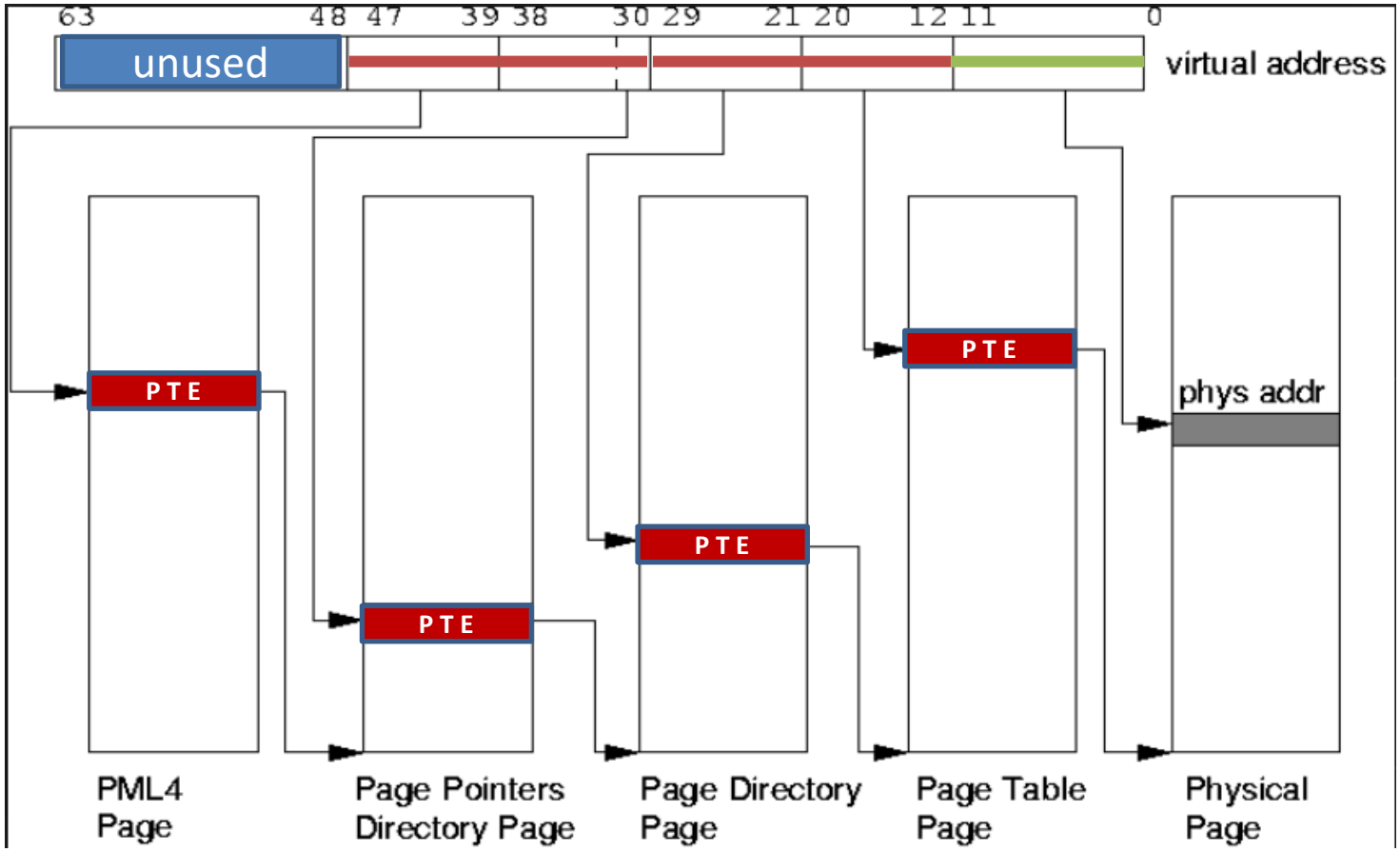
4 level PageTable for 64-bit arch



- This is a 48-bit virtual address space architecture.
- OS has to make sure no segment is allocated into high range of address space (63-48 bits)
- If bits are set the MMU will raise an exception (this is really an OS bug then)

Animation of Hardware

ldw r3, vaddr



General Formula of PageTable Management

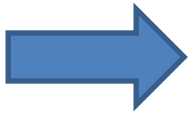
- **Hardware defines the frame size** as 2**N
- (virtual) page size is typically equal frame size (otherwise it's a power-of-two multiple, but that is rare and we shall not base on this exception)
- Everything the OS manages is based on pagesize (hence framesize)
- You can compute all the semantics with some basic variables defined by the 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

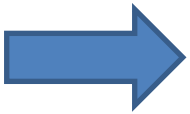
Speeding Up Paging

- Challenges:
  - Mapping virtual to physical address must be fast - we can not always traverse the page table to get the VA -> PA mapping



Translation Lookaside Buffer(TLB)

  - If address space is large, page table will be large (but remember the sparsity, not fully populated)



Multi-level page table

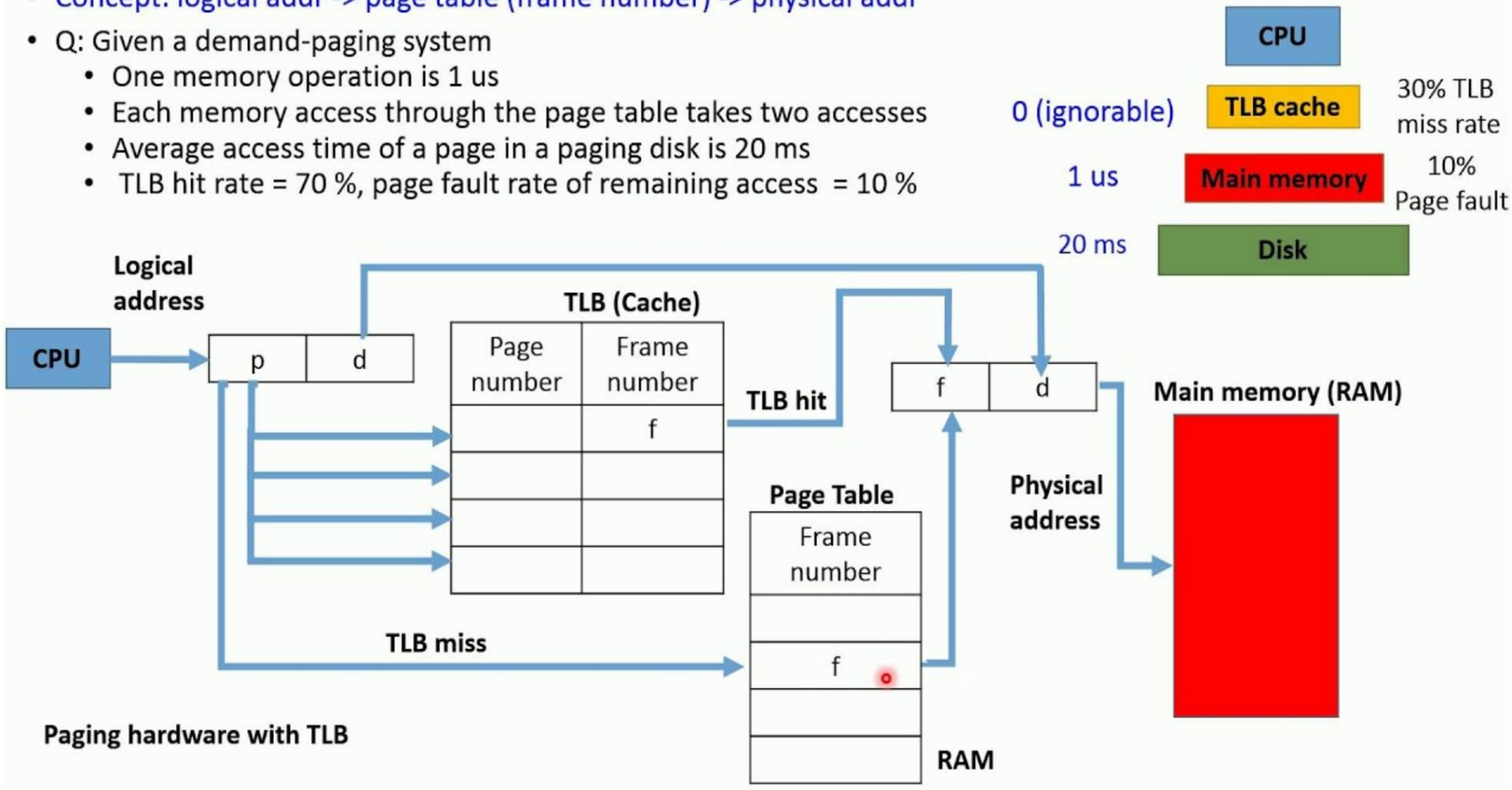
TLB

- **Observation**: most programs tend to make a large number of references to a small number of pages over a period of time -> only fraction of the page table is heavily used
(data and instruction locality !!!)

- TLB
  - Hardware cache 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 based translation

- Concept: logical addr -> page table (frame number) -> physical addr
- Q: Given a demand-paging system
  - One memory operation is 1 us
  - Each memory access through the page table takes two accesses
  - Average access time of a page in a paging disk is 20 ms
  - TLB hit rate = 70 %, page fault rate of remaining access = 10 %



TLB

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

TLB

- Sample content of a TLB
- Size often 256 – 512 entries
- 512*4KB = 2^9 * 2^12 = 2^21 = 2MB address coverage at any given time

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

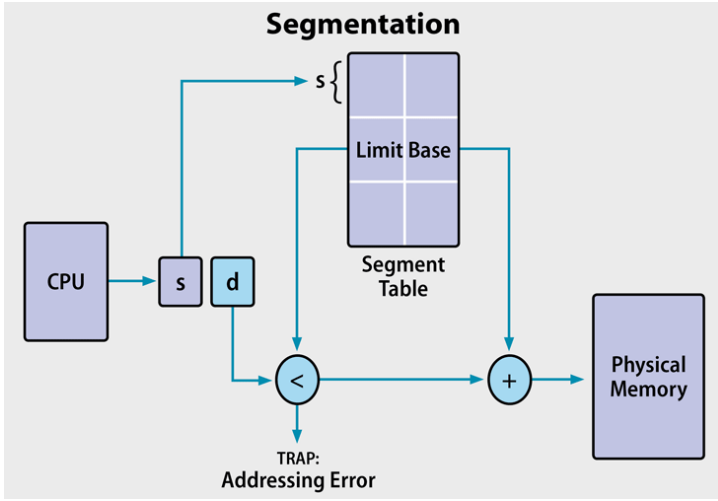
TLB management

- TLB entries are not written back on access, just record the R and M bits in the PTE (it is a cache after all) upon access
- On TLB capacity miss, if the entry is dirty, write it back to its associated PageTableEntry (PTE)
- On changes to the PageTable, potential entries in the TLB need to be flushed or invalidated
  - "TLB invalidate" e.g. when mmap area disappears.
  - *"TLB flush"* write back when changes in TLB to PageTable (either global or per address) must be recorded, think when a "*TLB invalidate*" might be insufficient

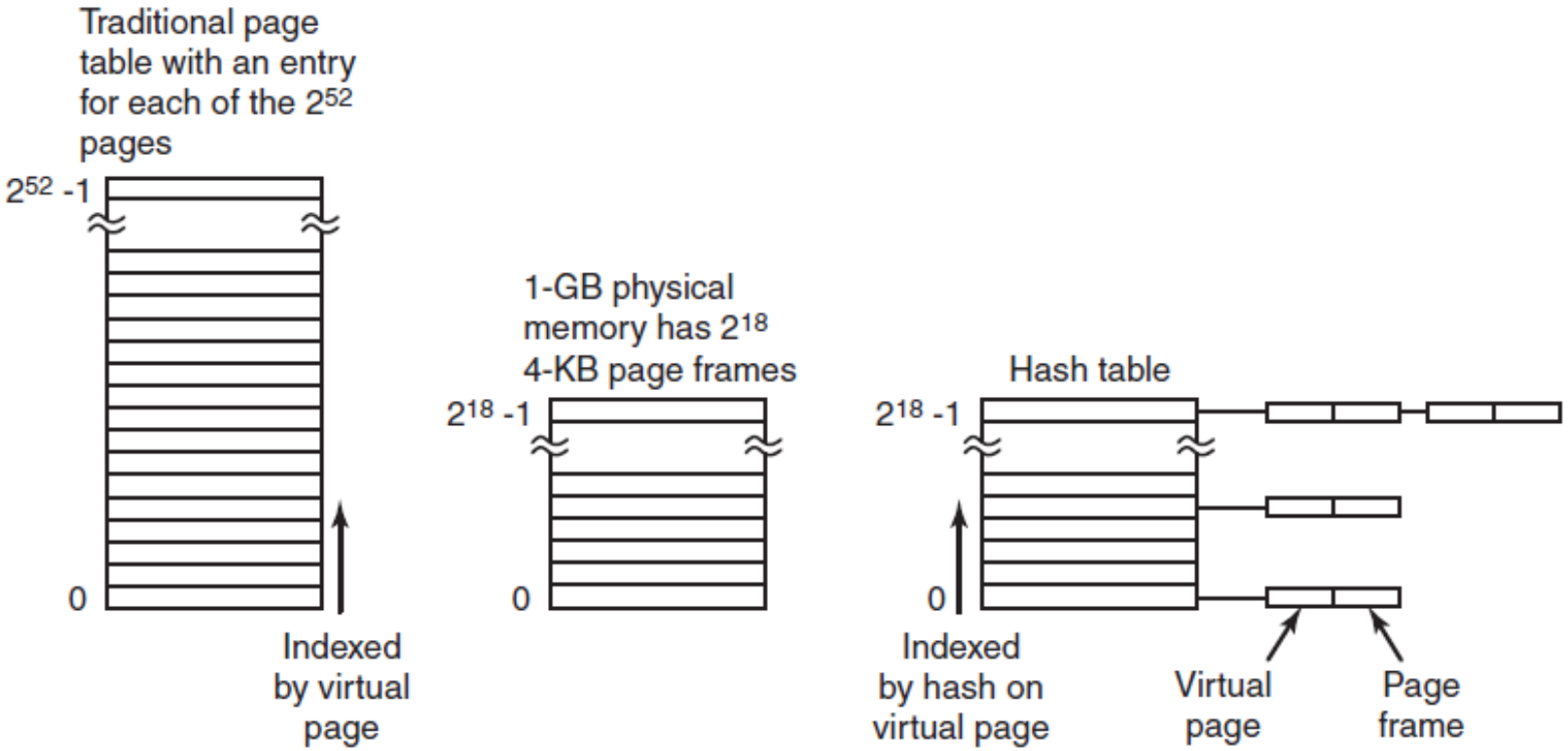
Other Translation Organizations

- Inverted Page Tables
  - PowerPC , Sparc, IA64

- Segmentation
  - 80x86 (but not x86-64)

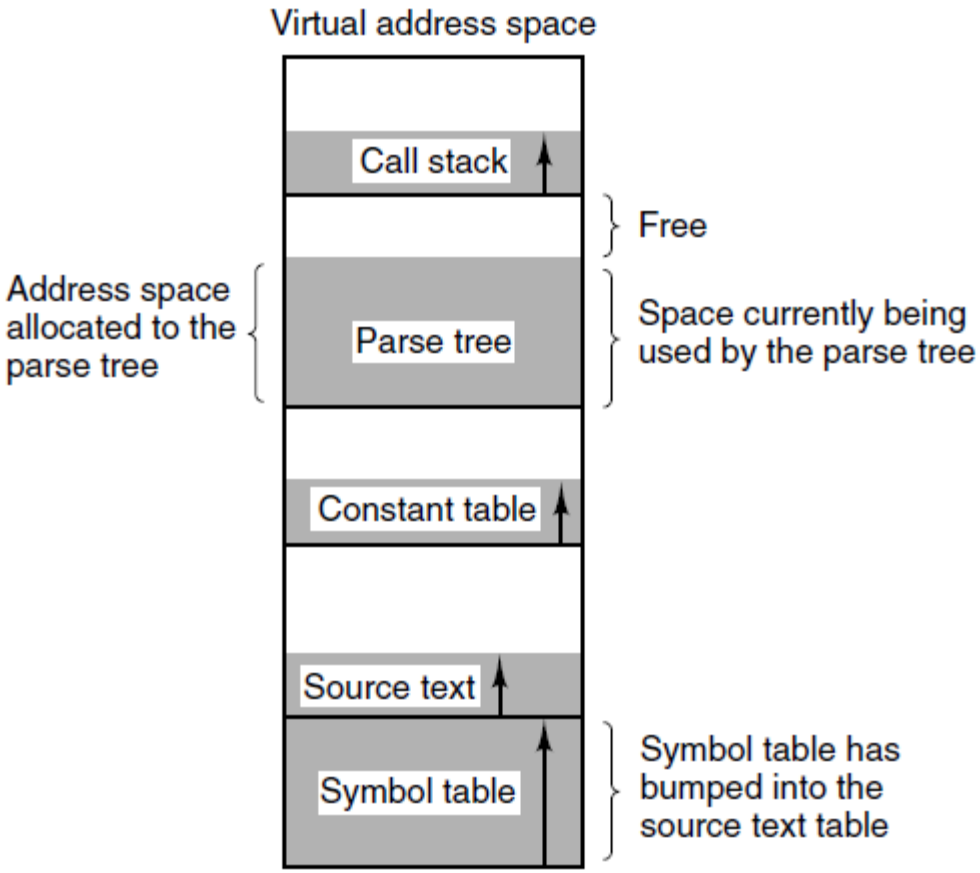


Inverted Page Tables



Comparison of a traditional page table with an inverted page table.

Segmentation (1)



In a one-dimensional address space with growing tables, one table may bump into another.

Segmentation (2)

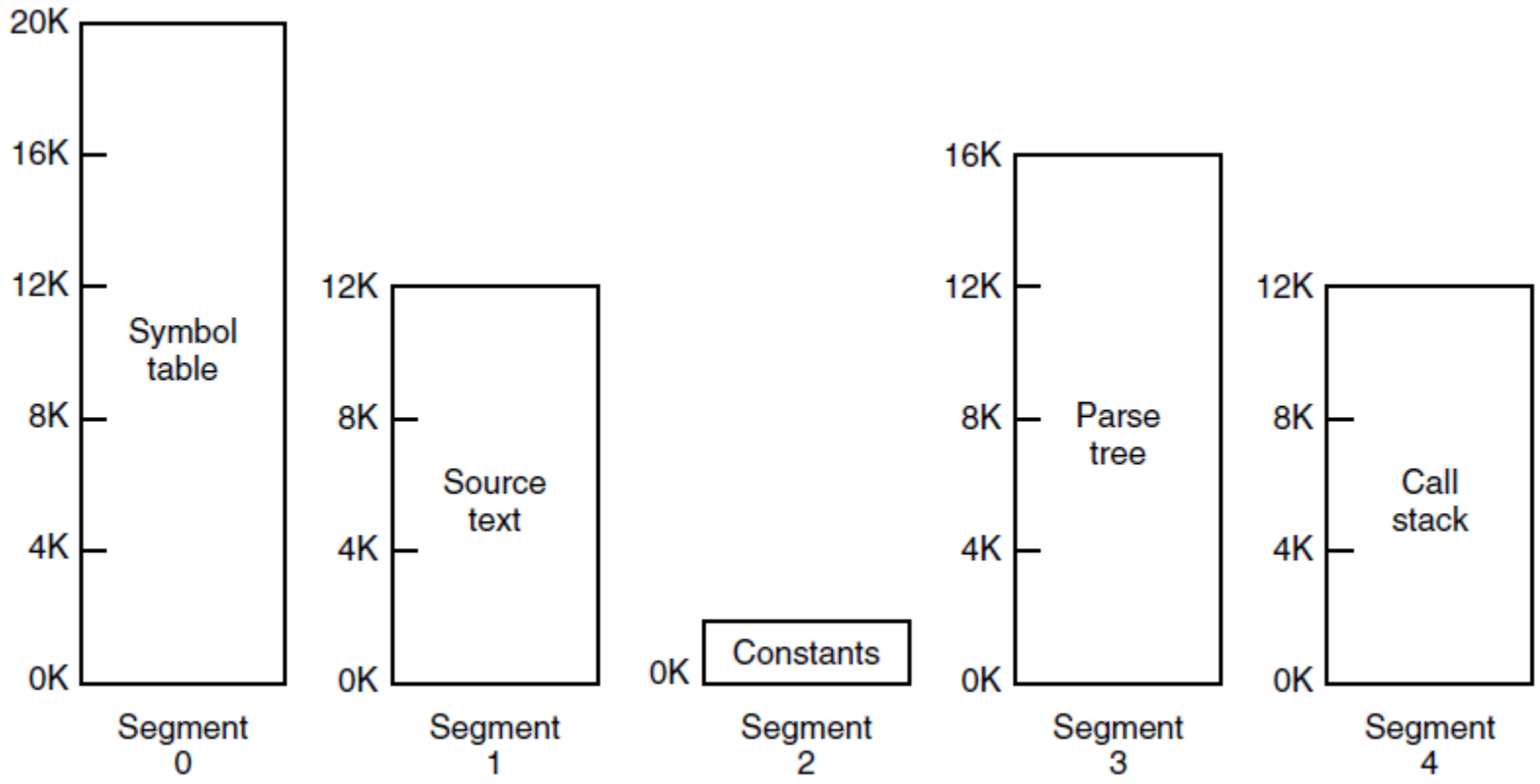


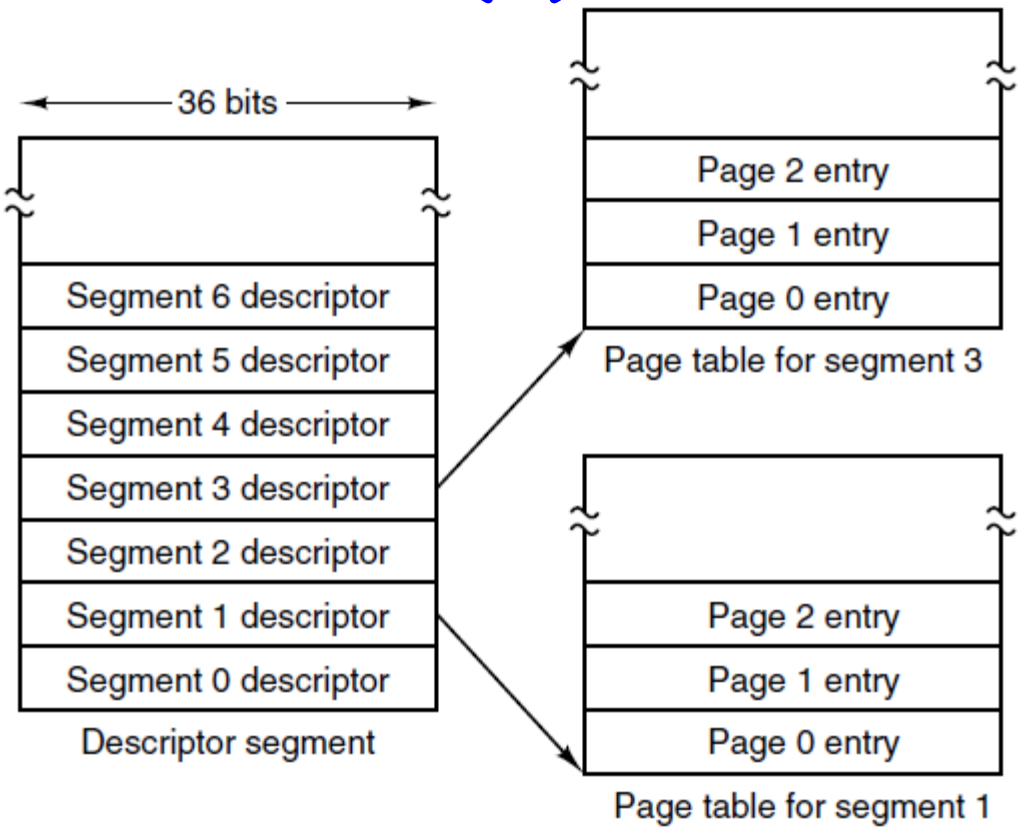
Figure 3-31. A segmented memory allows each table to grow or shrink independently of the other tables.

Segmentation (3)

Consideration	Paging	Segmentation
Need the programmer be aware that this technique is being used?	No	Yes
How many linear address spaces are there?	1	Many
Can the total address space exceed the size of physical memory?	Yes	Yes
Can procedures and data be distinguished and separately protected?	No	Yes
Can tables whose size fluctuates be accommodated easily?	No	Yes
Is sharing of procedures between users facilitated?	No	Yes
Why was this technique invented?	To get a large linear address space without having to buy more physical memory	To allow programs and data to be broken up into logically independent address spaces and to aid sharing and protection

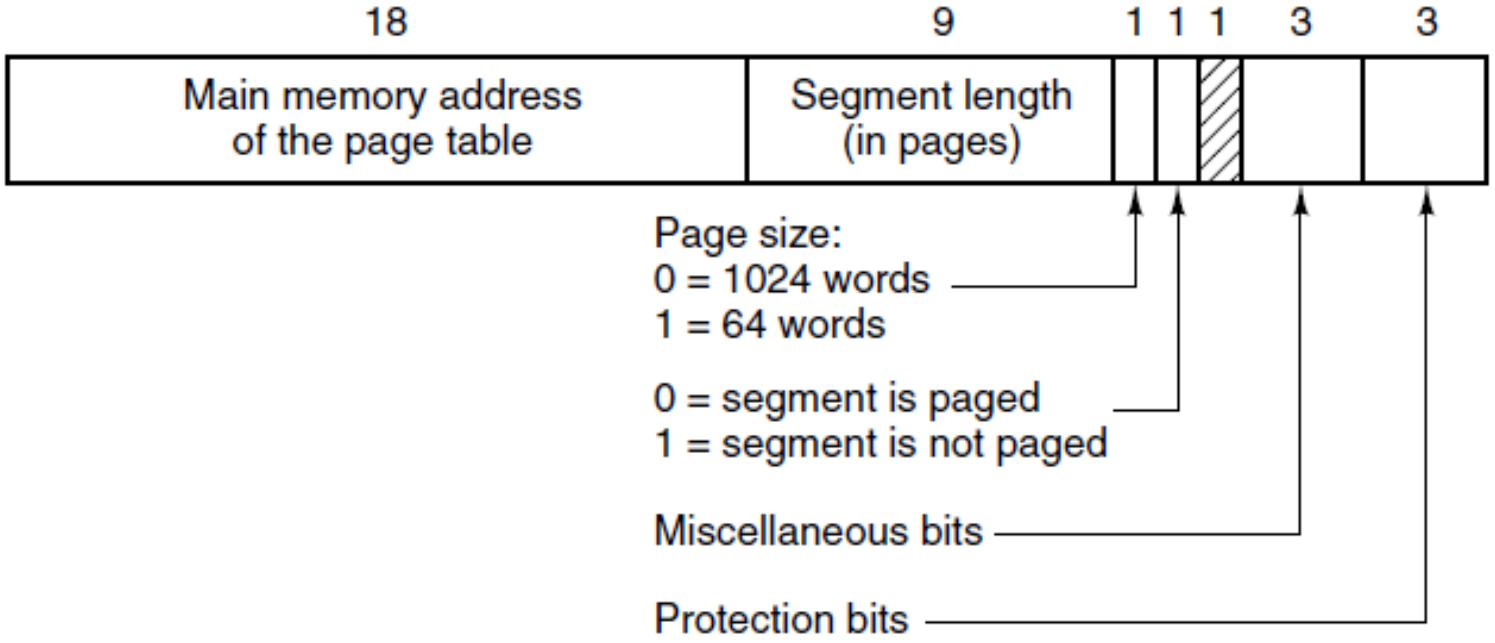
Comparison of paging and segmentation

Segmentation with Paging: MULTICS (1)



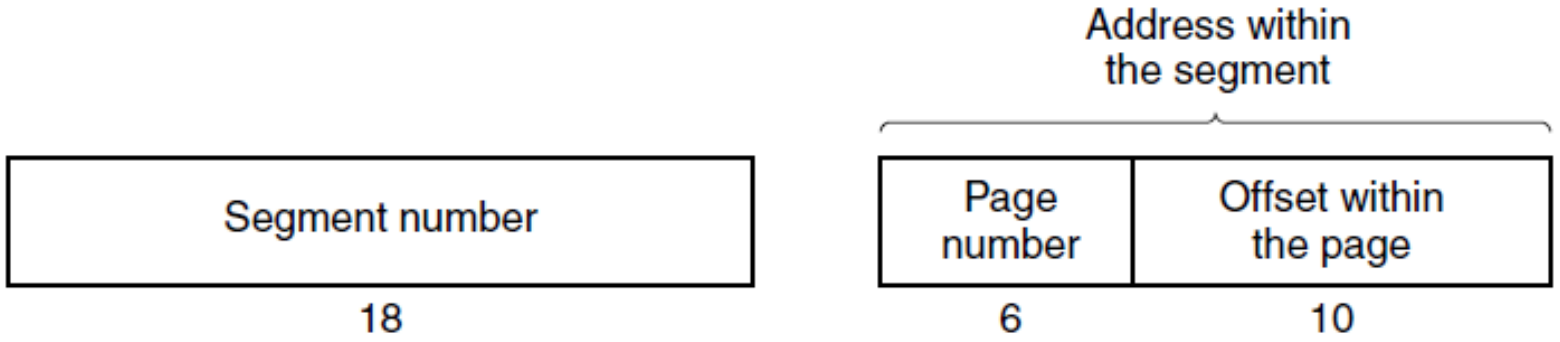
The MULTICS virtual memory. (a) The descriptor segment pointed to the page tables.

Segmentation with Paging: MULTICS (2)



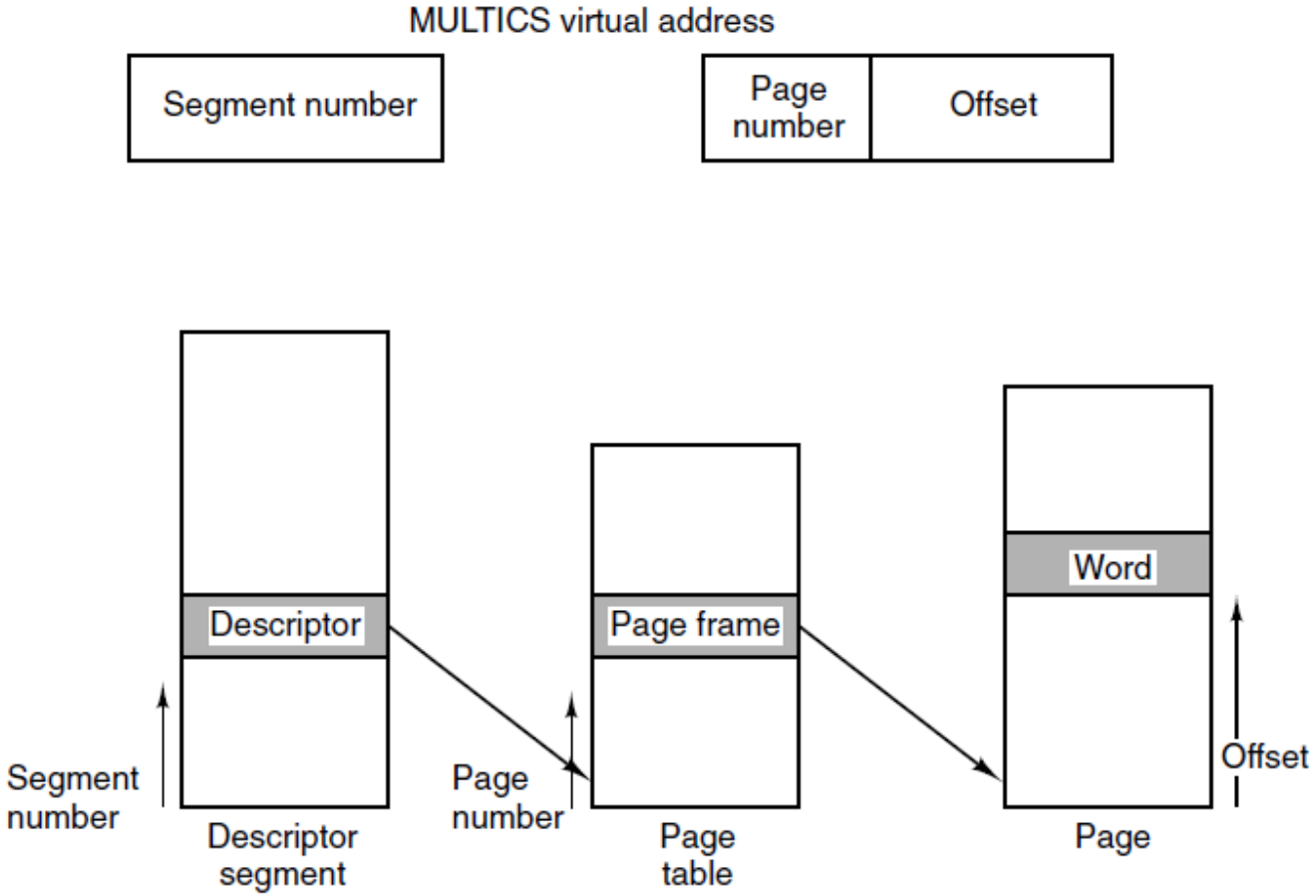
The MULTICS virtual memory. (b) A segment descriptor. The numbers are the field lengths.

Segmentation with Paging: MULTICS (3)



A 34-bit MULTICS virtual address.

Segmentation with Paging: MULTICS (4)



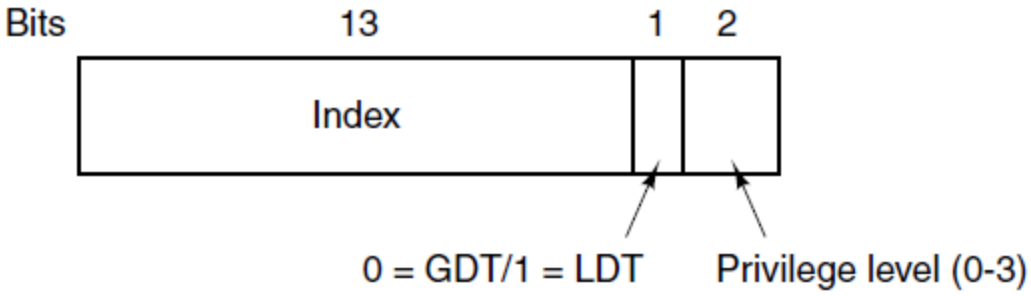
Conversion of a two-part MULTICS address into a main memory address.

Segmentation with Paging: MULTICS (5)

Comparison field: Segment number	Comparison field: Virtual page	Page frame	Protection	Age	Is this entry used?
4	1	7	Read/write	13	1
6	0	2	Read only	10	1
12	3	1	Read/write	2	1
					0
2	1	0	Execute only	7	1
2	2	12	Execute only	9	1

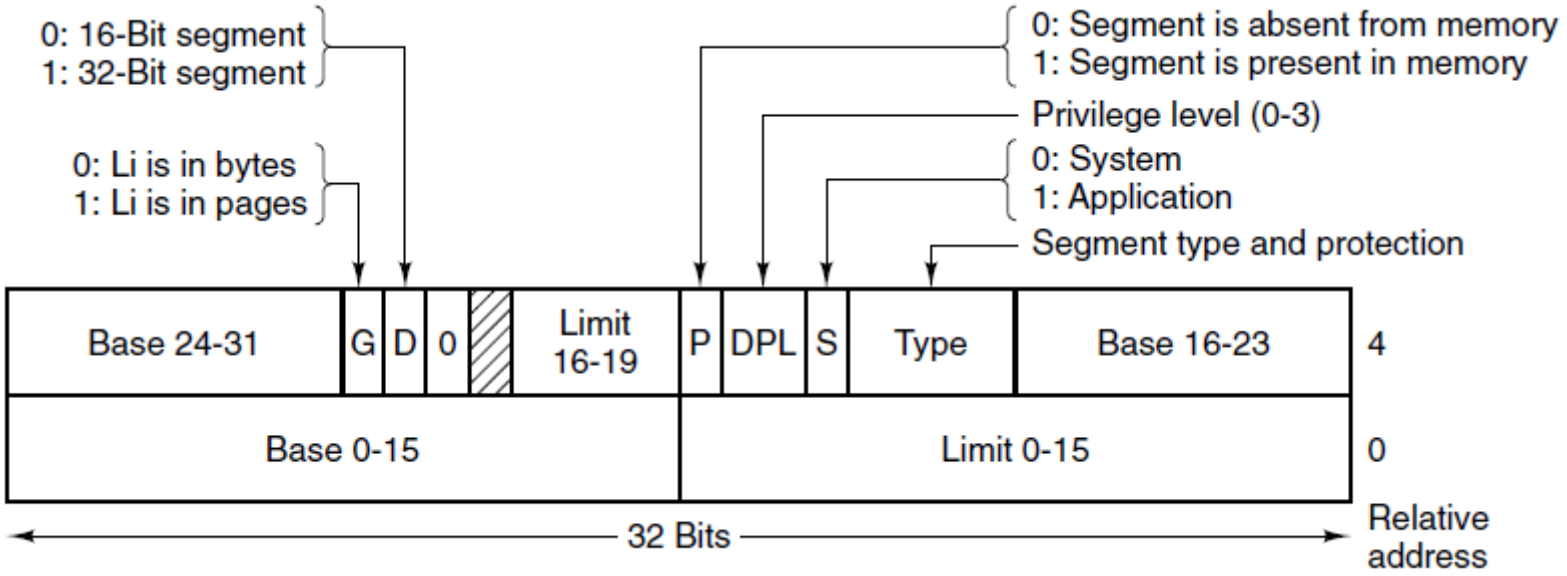
A simplified version of the MULTICS TLB. The existence of two page sizes made the actual TLB more complicated.

Segmentation with Paging: The Intel x86 (1)



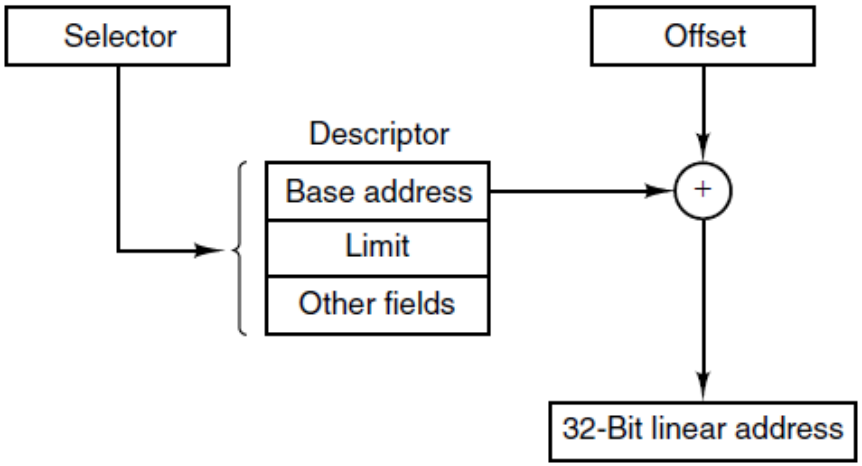
An x86 selector.

Segmentation with Paging: The Intel x86 (2)



x86 code segment descriptor.
Data segments differ slightly.

Segmentation with Paging: The Intel x86 (3)



Conversion of a (selector, offset) pair to a linear address.

The 8086 architecture introduced 4 segments:
CS (code segment)
DS (data segment)
SS (stack segment)
ES (extra segment)
the 386 architecture introduced two new general segment registers **FS**, **GS**.

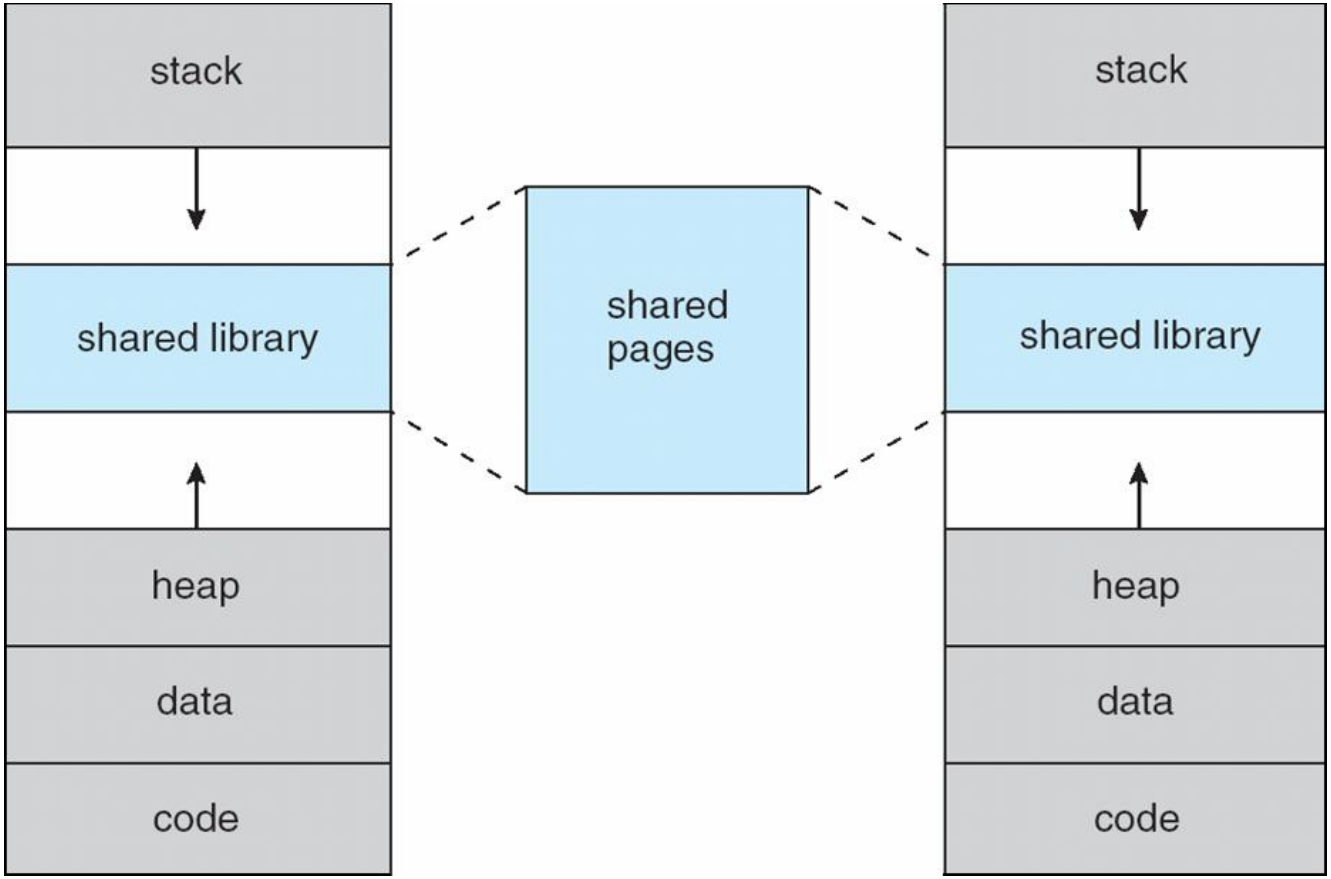
```
mov dword [fs:eax], 42
```

```
mov dx, 850h
mov es, dx ; Move 850h to es segment register mov
es:cx, 15h ; Move 15 to es:cx
```

Shared Pages (efficient usage of memory)

- **Shared code**
  - One copy of read-only (**reentrant**) code shared among processes (i.e., text editors, compilers, window systems)
  - Similar to multiple threads sharing the same process space
  - Also useful for inter-process communication if sharing of read-write pages is allowed
- **Private code and data**
  - Each process keeps a separate copy of the code and data
  - The pages for the private code and data can appear anywhere in the logical address space

Shared Library Using Virtual Memory



Copy-on-Write

- **Copy-on-Write** (COW) allows both parent and child processes to initially ***share*** the same pages in memory
  - If either process modifies a shared page, only then is the page copied

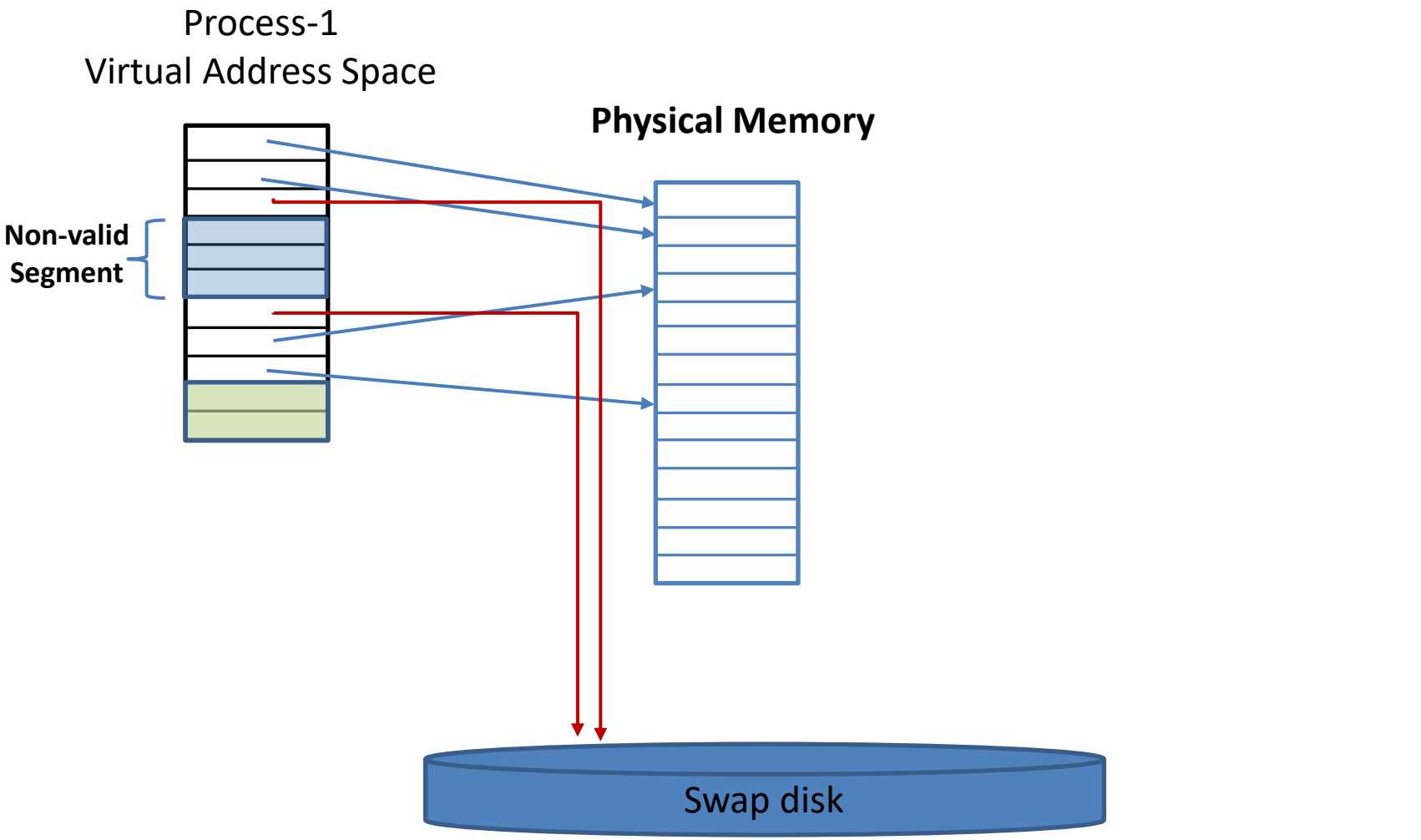
- COW allows more efficient process creation as only modified pages are copied

Copy-on-Write (cont)

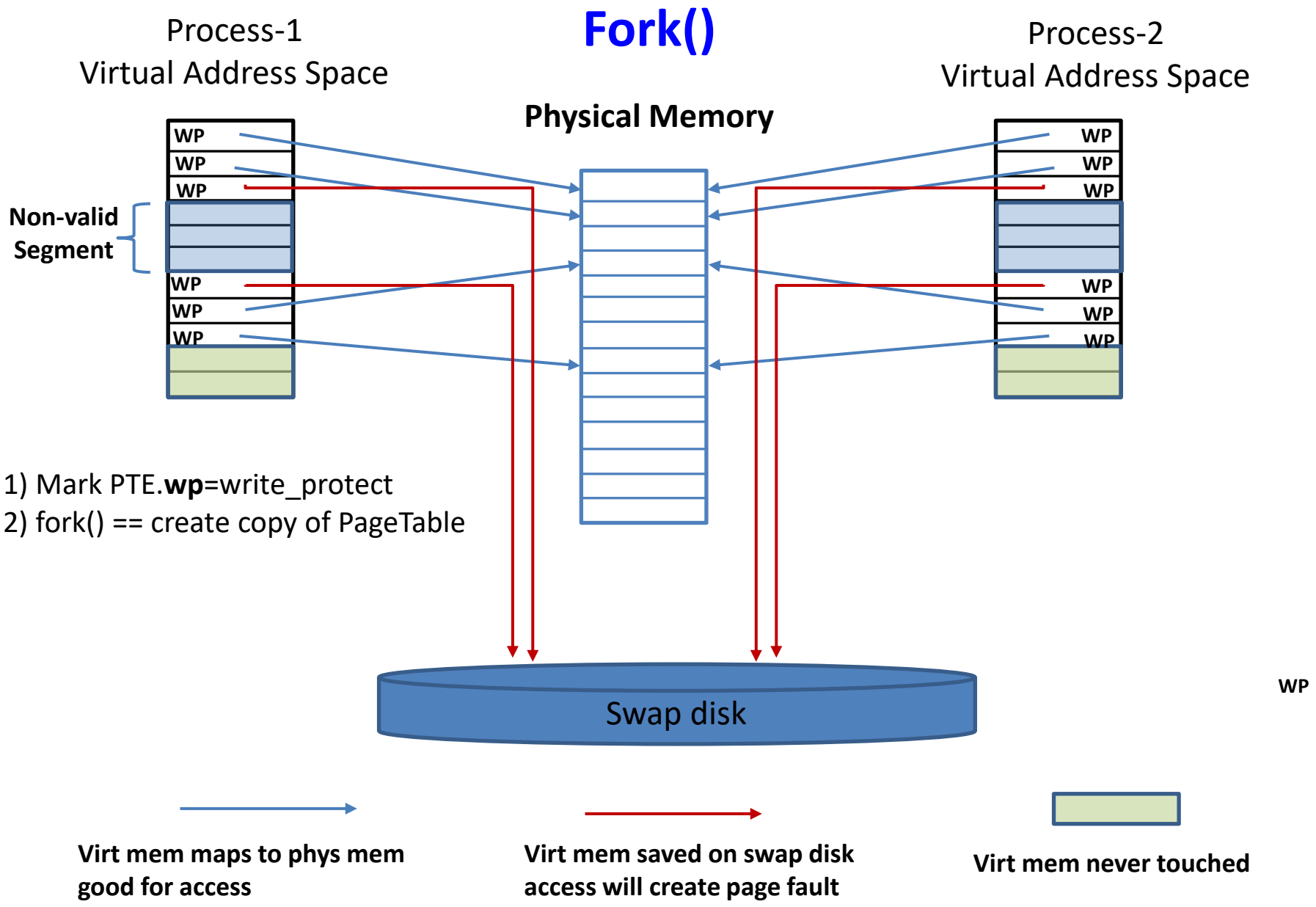
- In general, free pages are allocated from a **pool** of **zero-fill-on-demand** pages
  - Pool should always have free frames for fast demand page execution
    - Don't want to have to free a frame when one is needed for processing on page fault
  - Why zero-out a page before allocating it?
    -> so we get a known state of a page.

- `vfork()` variation on `fork()` system call has parent suspend and child using copy-on-write address space of parent
  - Designed to have child call `exec()`
  - Very efficient

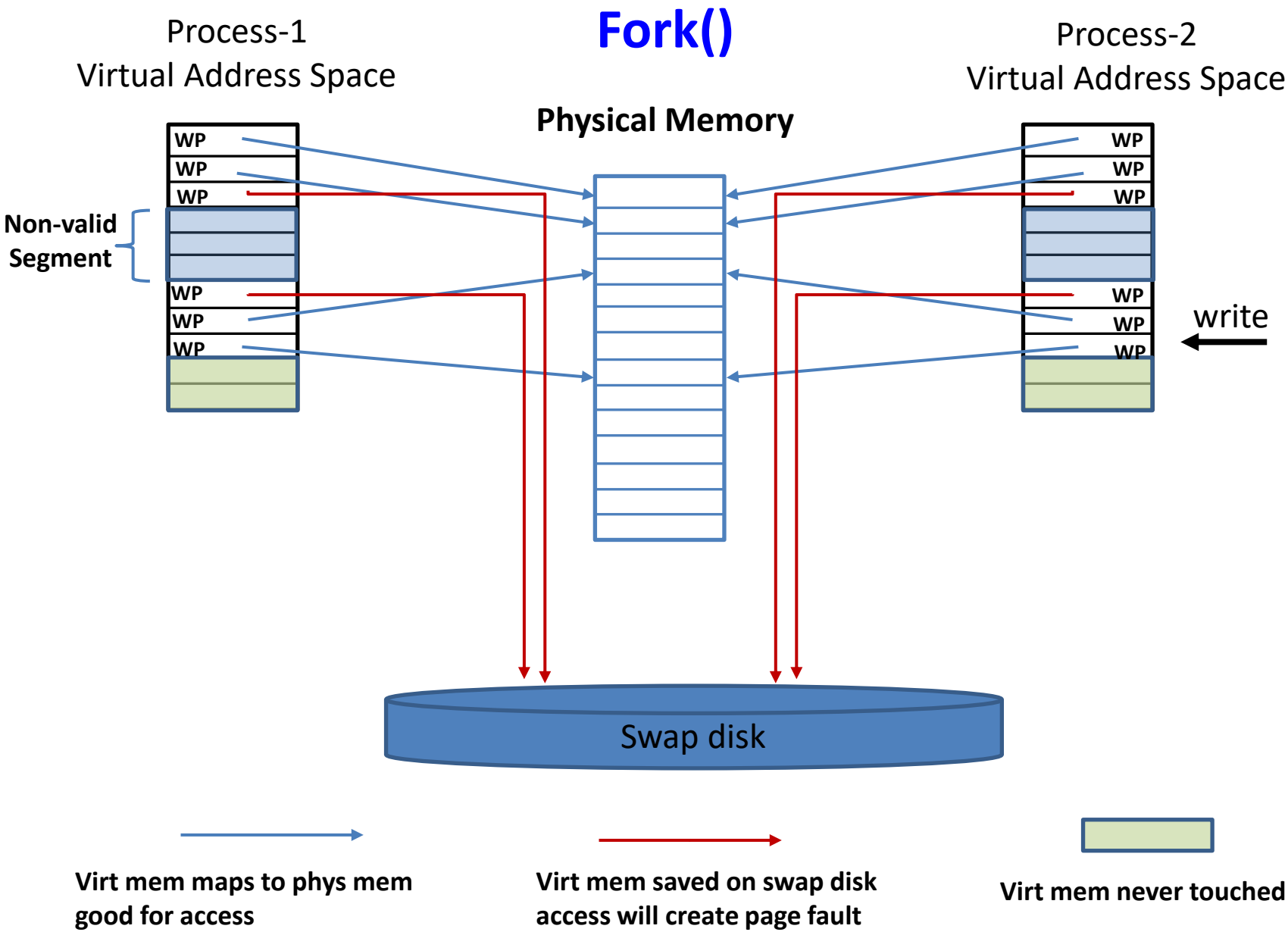
Virtual Memory



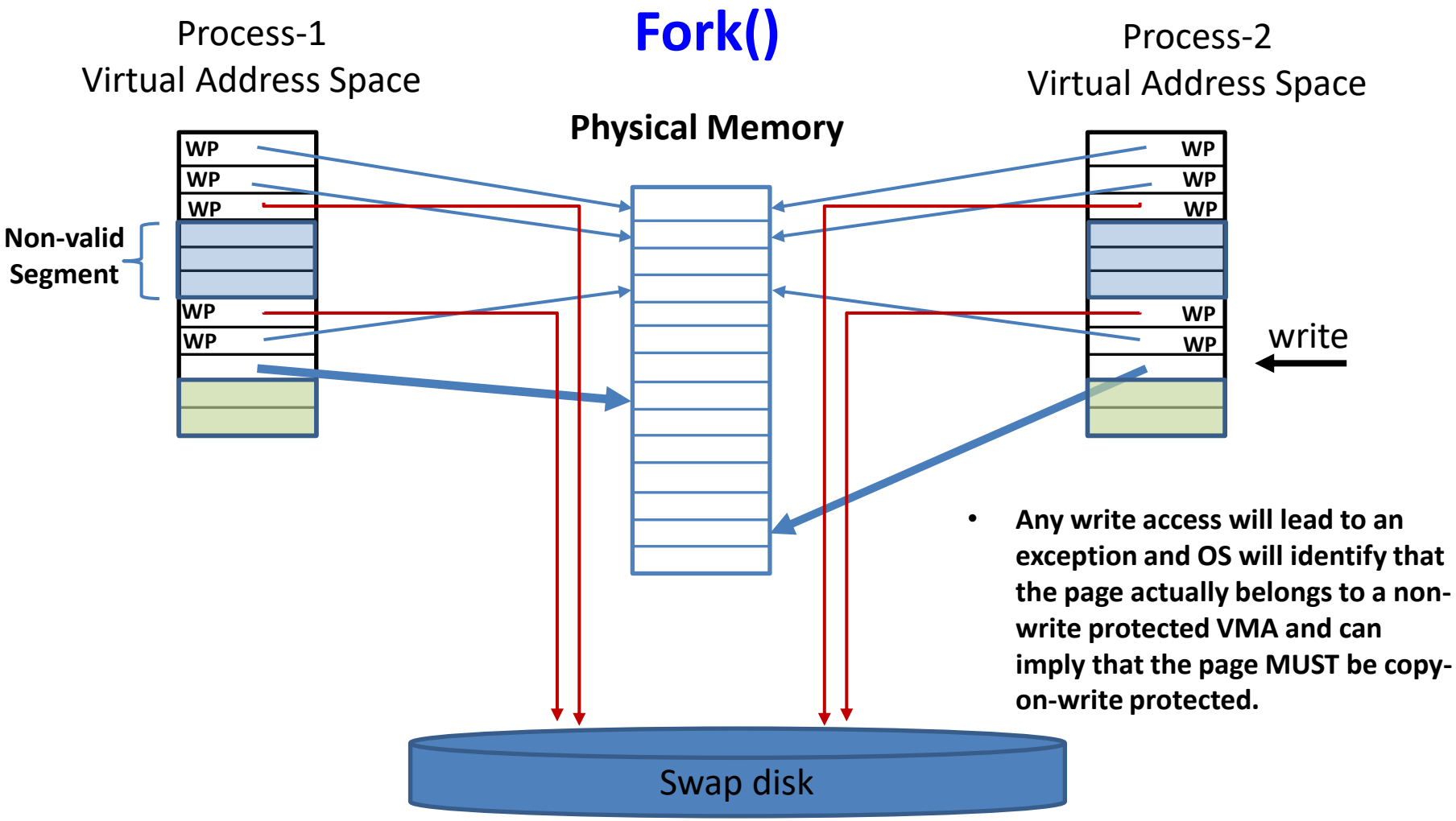
Virtual Memory



Virtual Memory (COW)



Virtual Memory (COW)



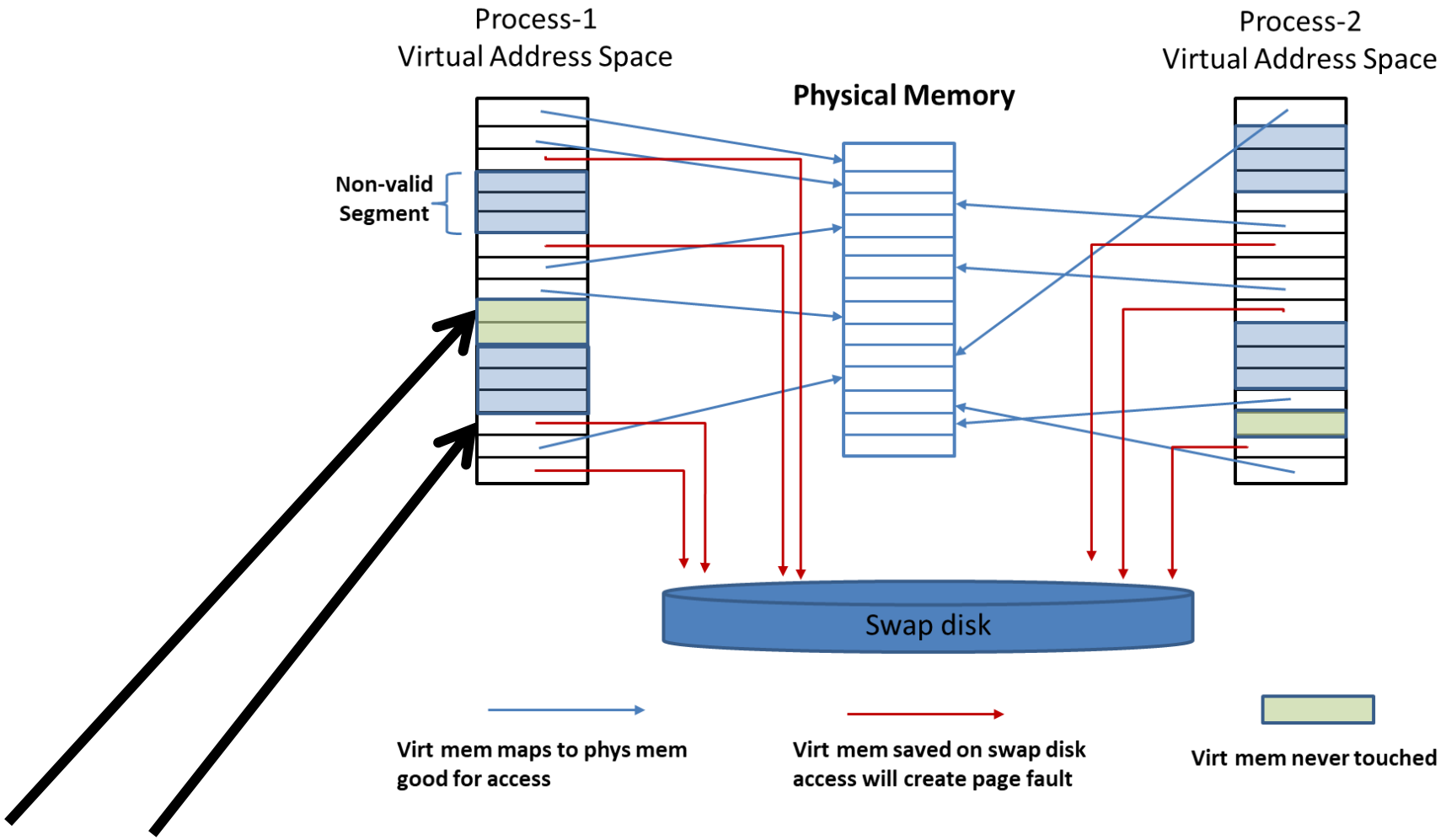
- **Any write access will lead to an exception and OS will identify that the page actually belongs to a non-write protected VMA and can imply that the page MUST be copy-on-write protected.**

Virt mem maps to phys mem good for access

Virt mem saved on swap disk access will create page fault

Virt mem never touched

We are running out of memory now what ?



In case of page fault:

which page to remove from memory and swap ➔ stealing from self or some other process ?

Replacement Policies

- Used in many contexts when storage is not enough (not just operating systems)
  - caches
  - web servers
  - Pages
- Things to take into account when designing a replacement policy
  - measure of success
  - cost

Thin[gk]s to consider

- Local Policies:
  - search processes' pagetables for candidates
  - O(N * sizeof(PageTable), where N = number of processes, sizeof(PageTable) ~= size of Virtual address space
  - Does not have a global view of usage
  - Enables running "global" algorithms for page replacement which are now O(F), where as F is #frames vs O(N*sizeof(PageTable), where as N is number of processes.

- Global Policies:
  - search frames for candidates
  - O(F) where F is number of physical frames (~= size of physical memory)

- In general global policies are preferred as O(F) << O(N*sizeof(PageTable))

for now let's first talk about
- a single page table
- talk about page replacement in that single page table
- Describe this algorithm that way .. → local policy with N=1
- Later we generalize with global policy and multiple processes.

Data Structures Required

- Address Space (one per process)
  - Represented as a sequence of VMAs (virtual memory areas)
    - [start-addr, length, Read/Write/Execute, …]
  - PageTable
- PageTable (`struct PTE pgtable[]`)
- Frame Table (`struct frame frame_table[]`)
  - each chunk of physical memory (e.g. 4K) is described by meta data [`struct frame`]
  - Used and how often, locked ?
  - Back reference to pte(s) that refer to this frame. Required because we need access to PTE's R and M bits to make replacement decisions.
  - In global algorithms we loop through the frametable but reach back to the PTEs
  - Note only pages that are mapped to frames can be candidates for paging !!!!

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
  (just used for theoretical evaluations)

More realistic Algos

- We don't have to make the best, optimal decision
- We need to make a reasonable decision at a reasonable overhead
- Its even OK to make the absolute worst decision occasionally as the system will implement correct behavior nevertheless.

The FIFO Replacement Algorithm

- OS maintains a list of the pages currently in memory [that would be for instance frame table]
- The most recent arrival at the tail
- On a page fault, the page at the head is removed
- In lab3: for simplicity you implement FIFO with a simple round robin using a HAND == pointer to the frametable

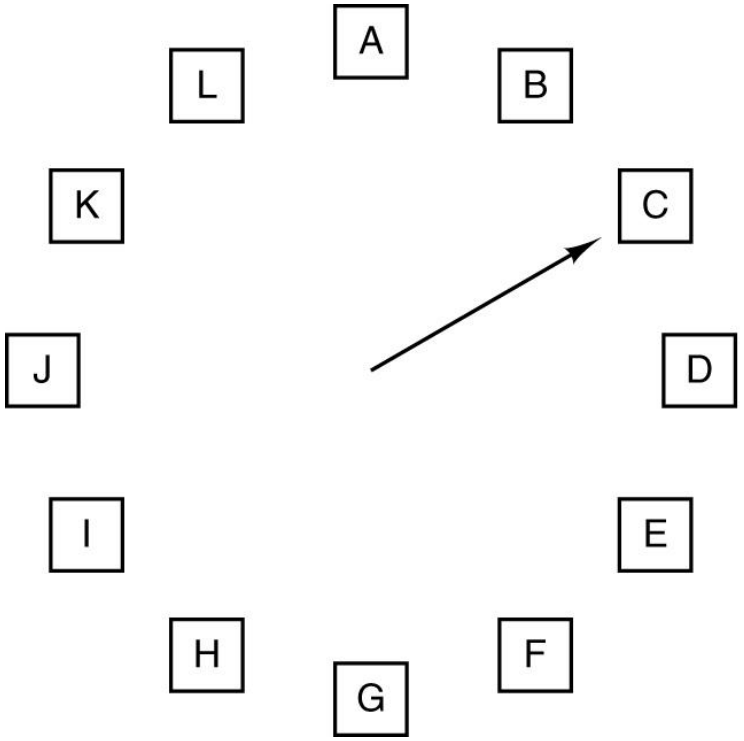
The Second-Chance Page Replacement Algorithm

- 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 Clock Page Replacement Policy

- Keep page frames on a circular list in the form of a clock
- The hand points to the oldest uninspected 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
- This is essentially an implementation of 2nd-Chance

The Clock Page Replacement Policy

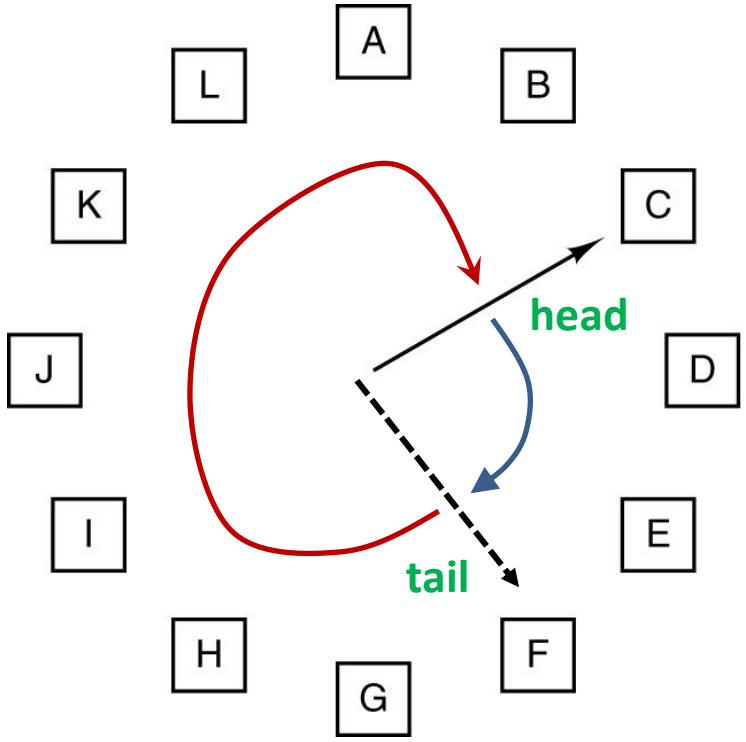


When a page fault occurs, the page the hand is pointing to is inspected. The action taken depends on the R bit:

R = 0: Evict the page

R = 1: Clear R and advance hand

The Clock Page Replacement Policy (an optimization)



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

Tail moves in front and “processes” candidates

- writing optimistically modified pages back to swap and clear “M” bit and R bit

The Not Recently Used (NRU) Replacement Algorithm

Sometimes called: Enhanced Second Chance 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 un-empty Class.
Note: class_index = 2*R+M

Note: in lab3 to we select the first page encountered in a class so we don't have to track all of them, we also use a clock like algorithm to circle through pages/frames

Ref-Bit reset doesn't happen each and every search.
Typically done at specific times through a daemon.

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
- Afte

Too expensive!

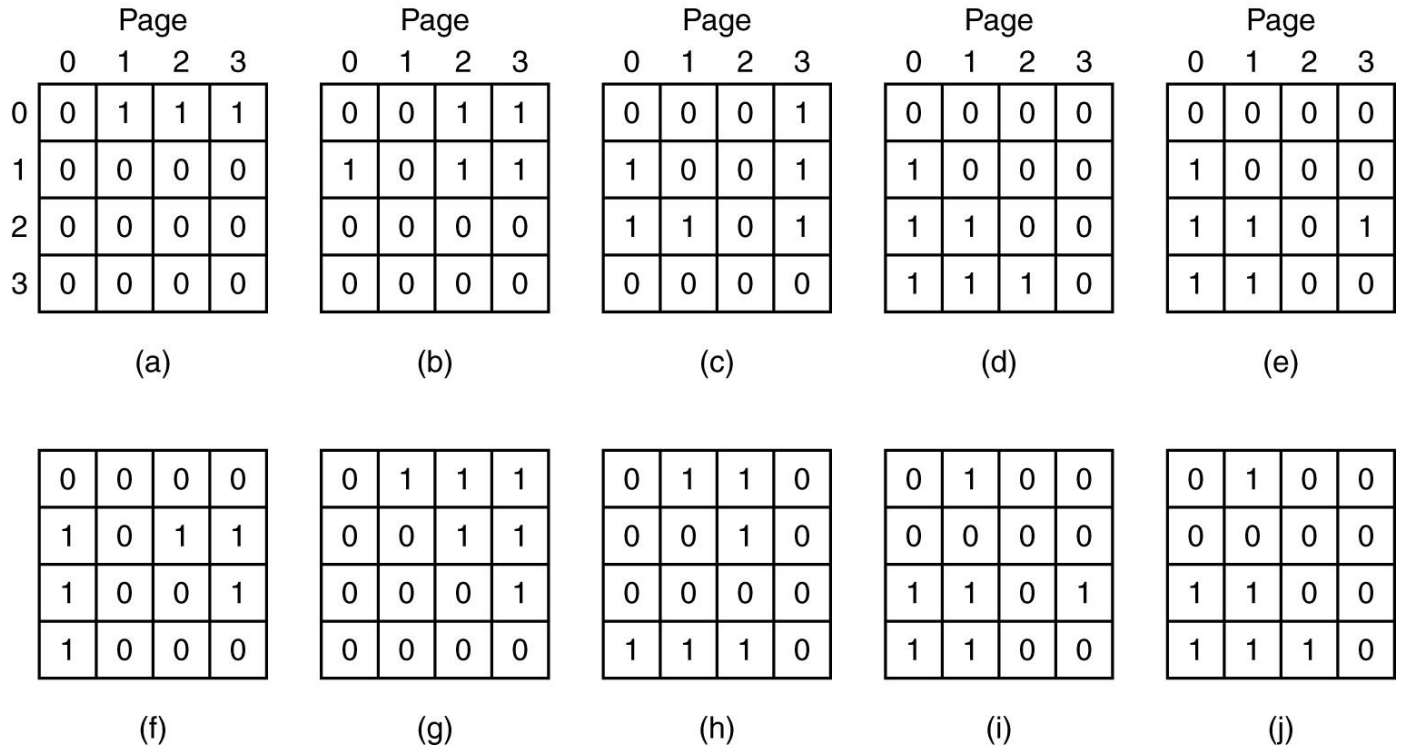
Too Slow!

- At page fault, the page with lowest value is discarded

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

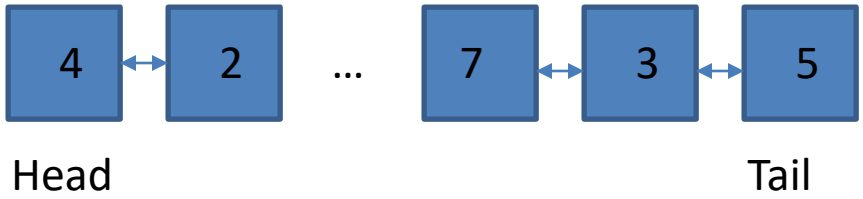
LRU: Hardware Implementation 2



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

LRU Implementation

- Slow
- Few machines (if any) have required hardware

Approximating LRU in Software

- Not Frequently Used (NFU) algorithm
- Software counter associated with each page, initially zero
- At some periodicity (e.g. a second), the OS scans all page table entries and adds the R bit to the counter of that PTE
- At page fault: the page with lowest counter is replaced

Still a lot of overhead (not really practical either)

Aging Algorithm

- NRU never forgets anything
  -> high inertia

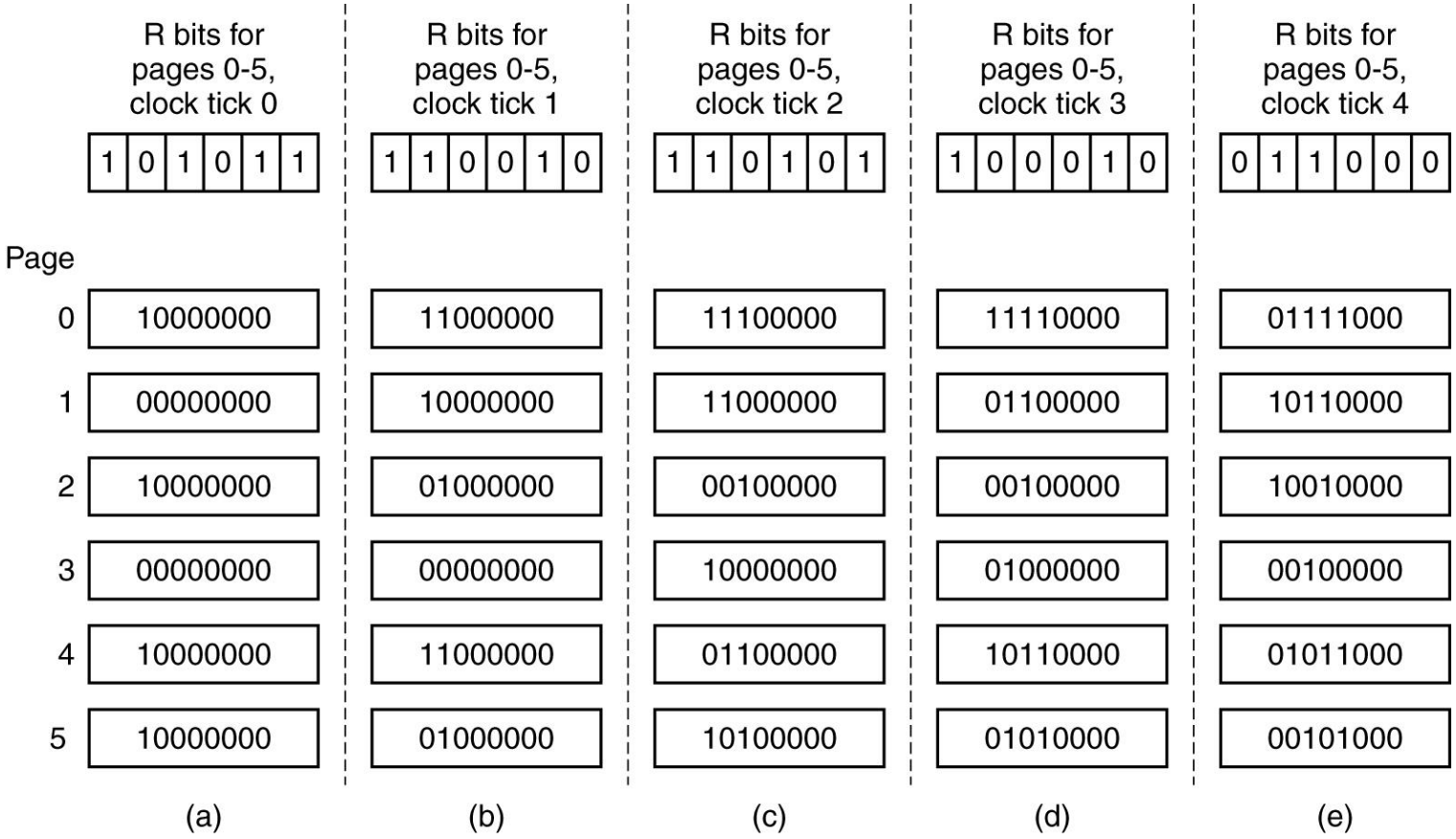
- Modifications:
  - shift counter right by 1
  - add R bit as the leftmost bit
  - reset R bit

- This modified algorithm is called aging

- Each bit in vector represents a period

- The page whose counter is lowest is replaced at page replacement

Aging Algorithm



The Working Set Model

- Working set: the set of pages that a process is currently using

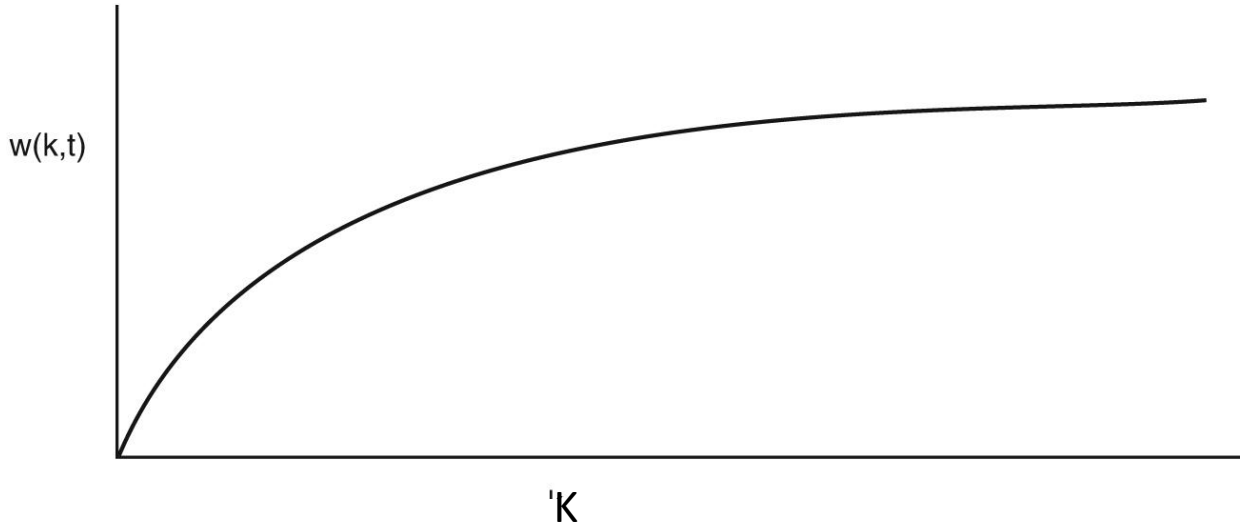
- Thrashing: a program causing page faults at high rates (e.g. pagefaults/instructions metric)

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.

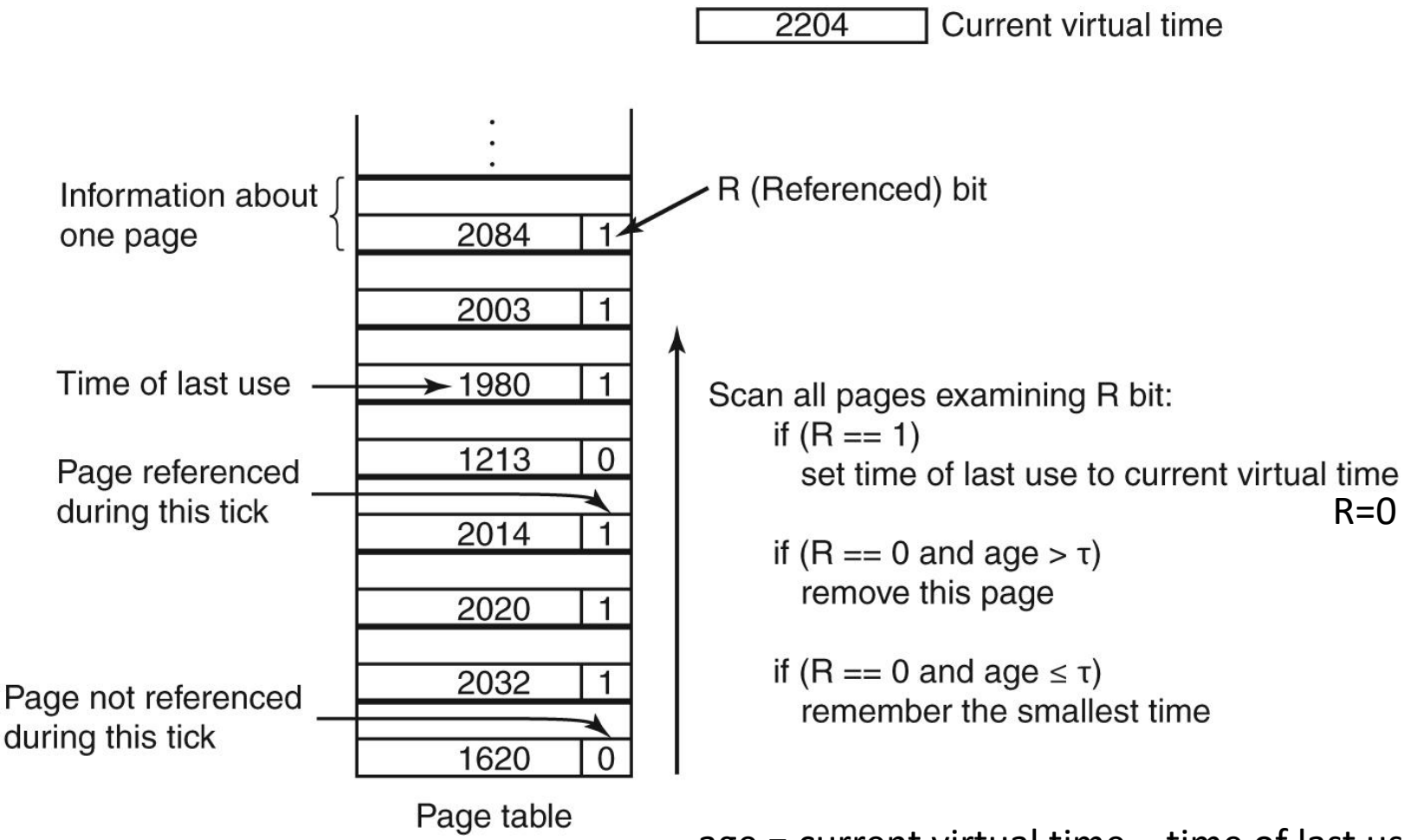


w(k,t): the set of pages accessed in the last k references at instant t

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 or 1 sec (etc.)

Working Set Page Replacement Algorithm



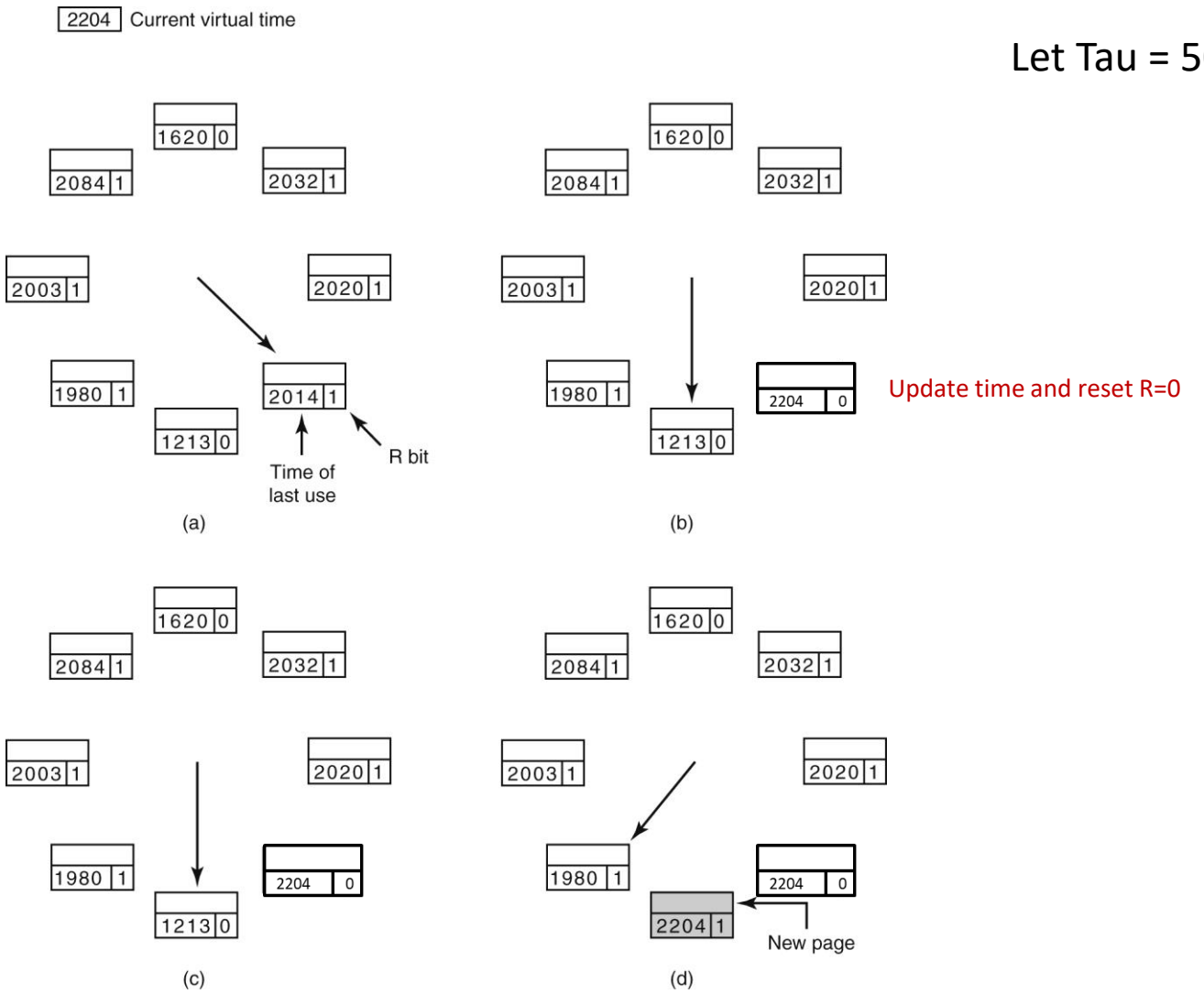
age = current virtual time – time of last use

time of last use == ref-bit was reset

The WSClock Page Replacement Algorithm
(specific implementation of Working Set Replacement)

- Based on the clock algorithm and uses working set
- data structure: circular list of page frames (clock)
- Each entry contains: time of last use, R bit
- At page fault: page pointed by hand is examined
  - If R = 1:
    - Record current time , reset R
    - Advance hand to next page
  - 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 (note in lab3, we skip this step for simplicity reasons, but think why this is advantageous ?)
    - Advance hand to next page

The WSClock Page Replacement Algorithm



Let Tau = 500

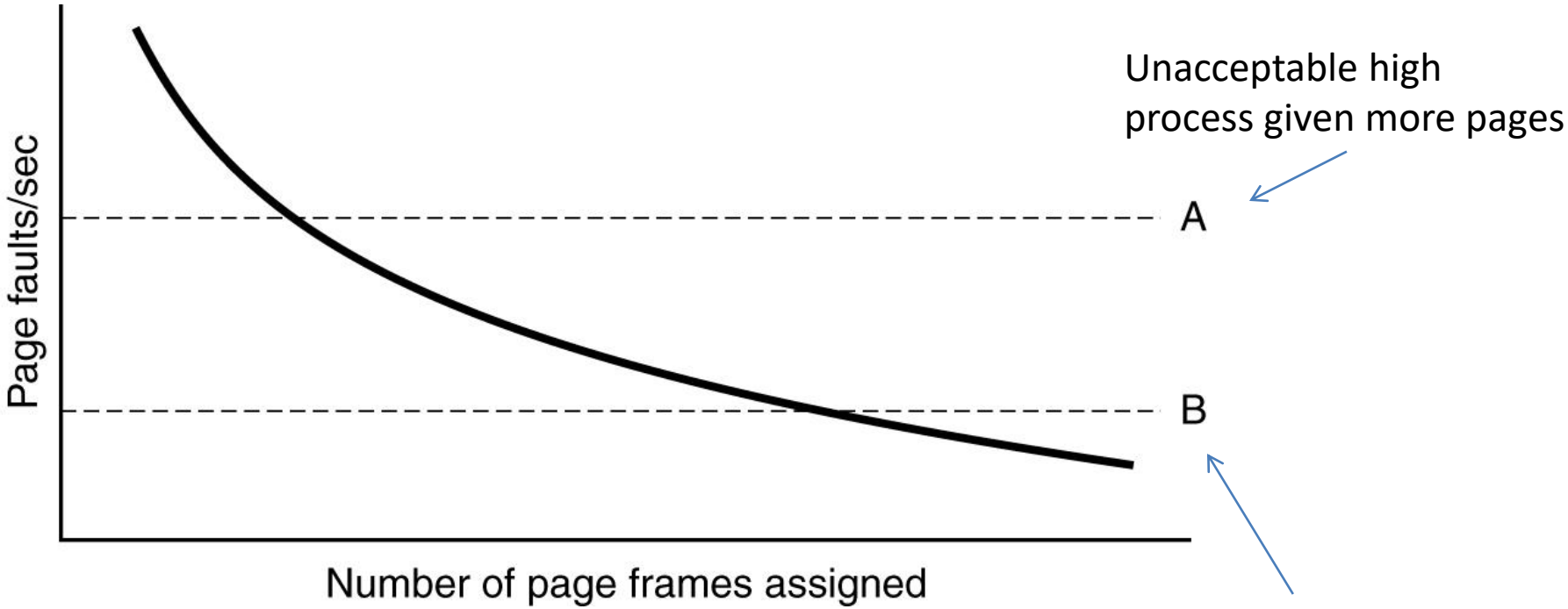
Design Issues for Paging: Local vs Global Allocation

- How should memory 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

Global Policies and OS Frame Tables

- In general the OS keeps meta data for each frame, typically 8-16 bytes.
  → 2-4% of memory required to describe memory.

- Meta Data includes
  - reverse mapping (pid, vpage) of user
  - Helper data for algorithm (e.g. age, tau)

- Enables running "global" algorithms for page replacement which are now O(F), where as F is #frames vs O(N*sizeof(PageTable), where as N is number of processes.

- Replacement algo can now run *something* similar to this.

```
for ( i=0 ; i<num_frames ; i++) {
     derive user's PTE from frametable[i].<pid,vpage>
     do what ever you have to do based on PTE
     determine best frame to select
}
```

Design Issues: Page Size

- Large page size → internal fragmentation
- Small page size →
  - larger page table
  - More overhead transferring from disk
- Remember the Hardware Designers decide the frame/page sizes !!

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: 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

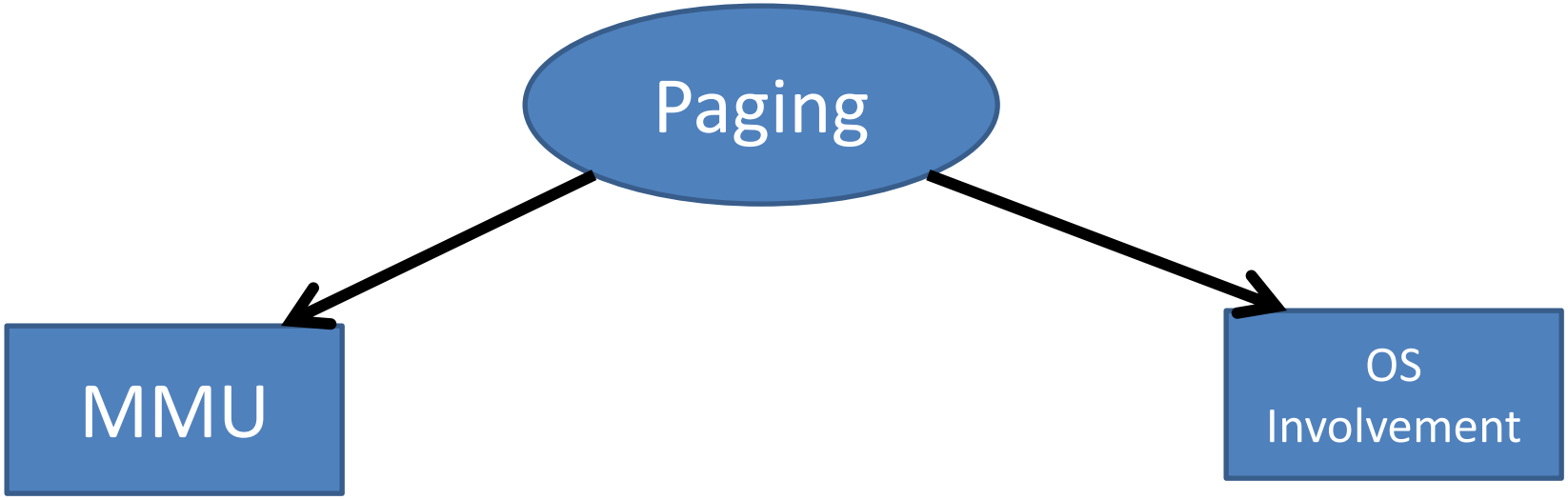
- Dynamically linked
- Loaded when program is loaded or when functions in them are called for the first time
- Compilers must not produce instructions using absolute addresses in libraries → position-independent code

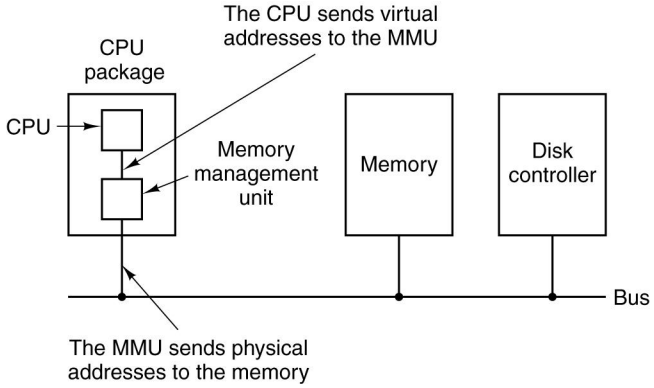
Design Issues: Cleaning Policy

- Paging daemon
- Sleeps most of the time
- Awakened periodically to inspect state of the memory
- If too few pages/frames are free -> daemon begins selecting pages to evict and gets more aggressive the lower the free page count sinks.

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 and are non-contiguous in physical address space

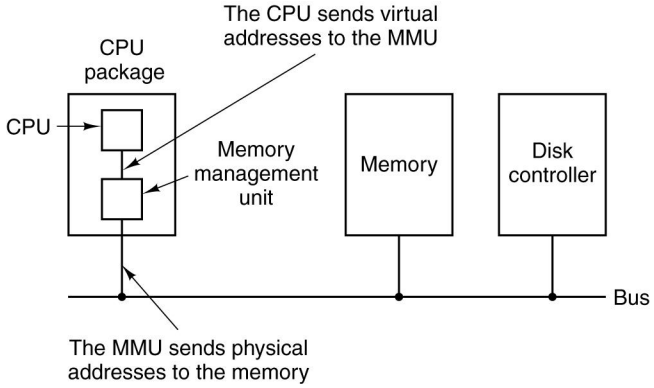

Paging
MMU
OS
Involvement


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



?

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

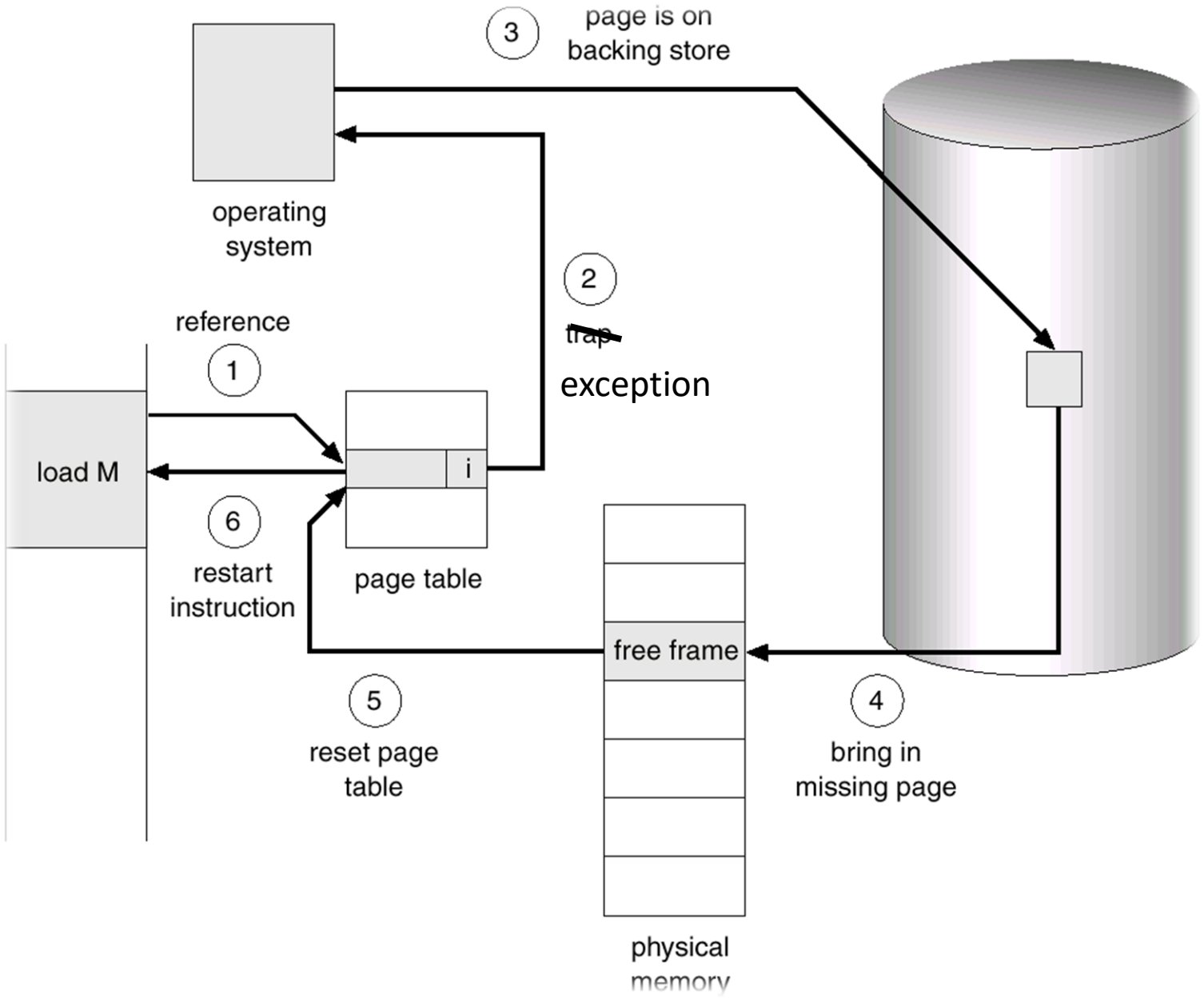
- When a new process is created
- When a process is scheduled for execution
  - TLB flushed for current process
  - MMU reset for the process
  - Process table of next process made current
- When process exits
- When page fault occurs

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
exception
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 Exception Handling

1. The hardware:
   - Saves program counter
   - Exception leads to kernel entry

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.

Virtual Memory & I/O Interaction
(Interesting Scenario)

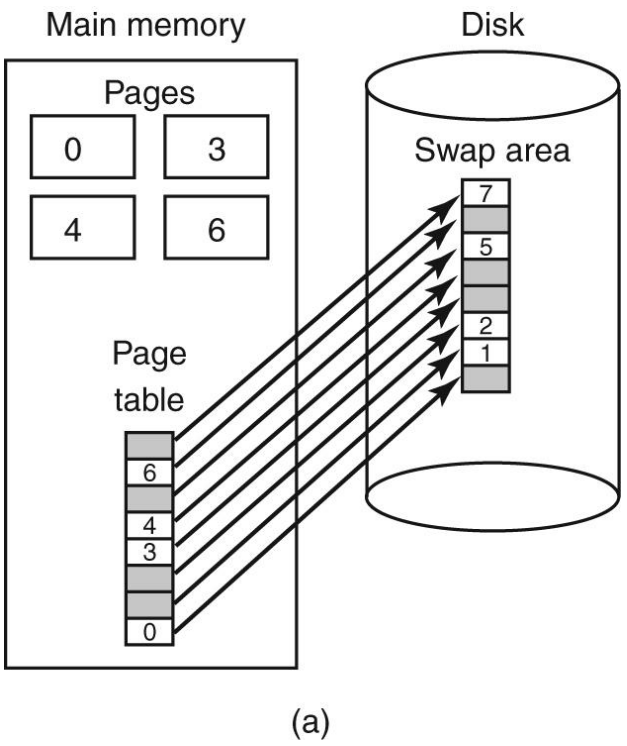
- Process issues a `syscall()` to read a file into a buffer
- Process suspended while waiting for I/O
- New process starts executing
- This other process gets a page fault
- If paging algorithm is global there is a chance the page containing the buffer could be removed from memory
- The I/O operation 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 by replacement algo

Backing Store

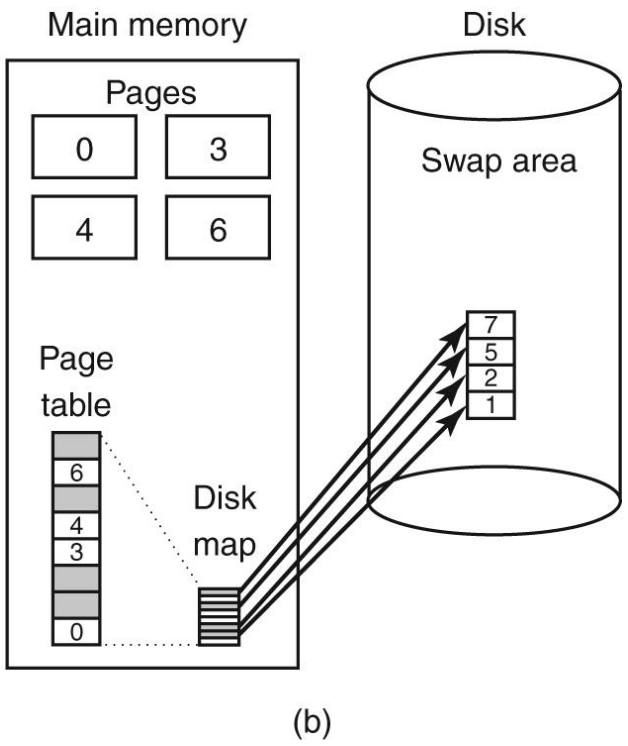
- Swap area: not a normal file system on it.
- Associates with each process the disk address of its swap area; store in the process table
- Before process starts swap area must be initialized
  - One way: copy all process image into swap area [static swap area]
  - Another way: don't copy anything and 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

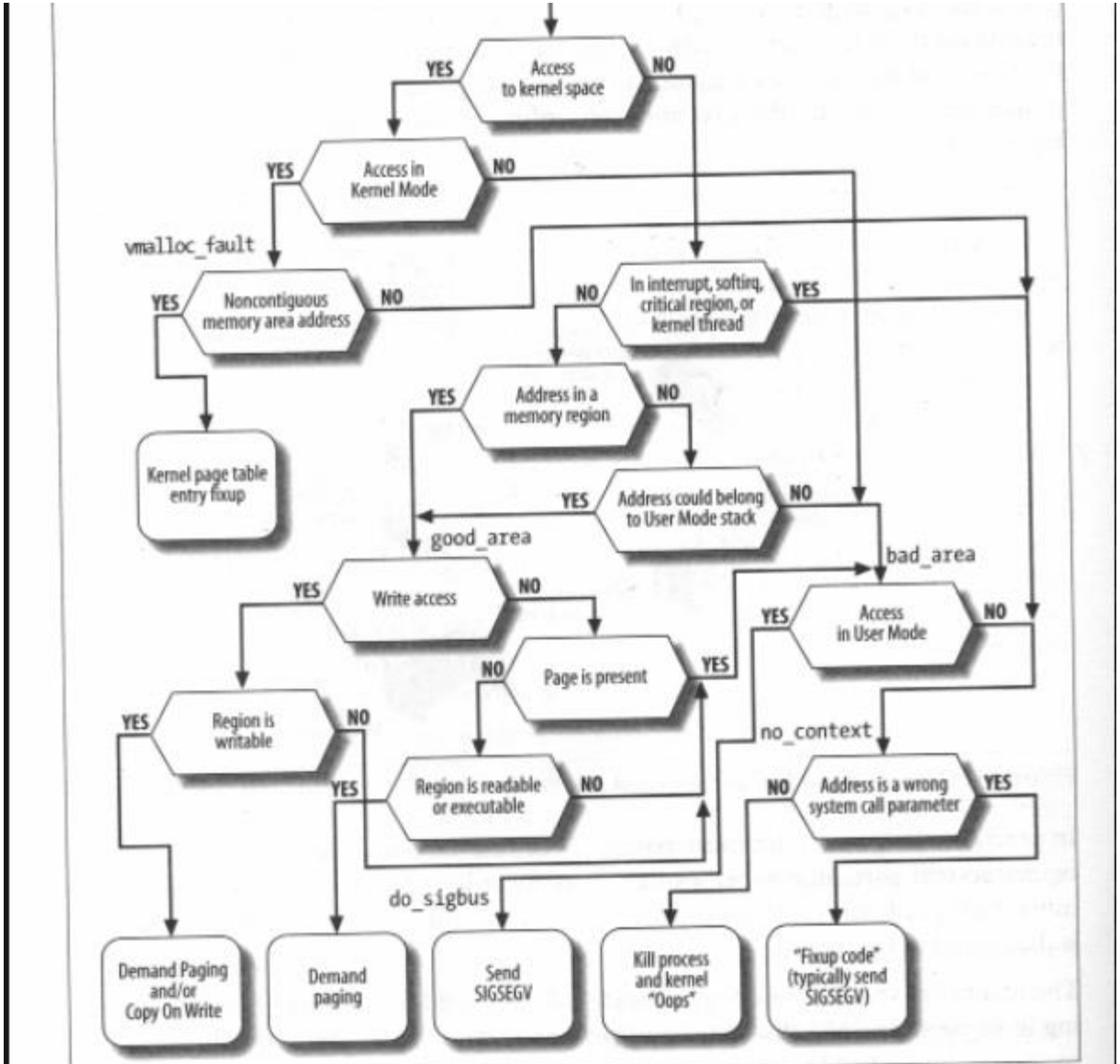


Figure 9-5. The flow diagram of the Page Fault handler

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

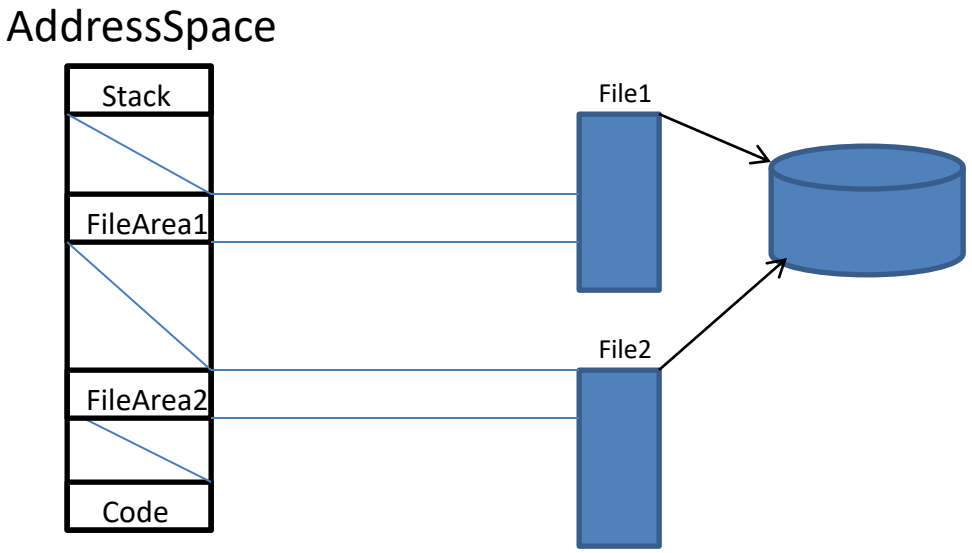
```
b7773000-b777a000 r--s 00000000 08:01 269322     /usr/lib/gconv/gconv-modules.cache
b777a000-b777b000 r--p 00000000 08:01 404953     /usr/lib/locale/en_US.utf8/LC_IDENTIFICATION
b777b000-b777d000 rw-p 00000000 00:00 0
bfd9a000-bfe5d000 rw-p 00000000 00:00 0          [stack]
```

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 (see `mmap()` `fd` argument)
- It's contiguous
- 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 (different versions though)

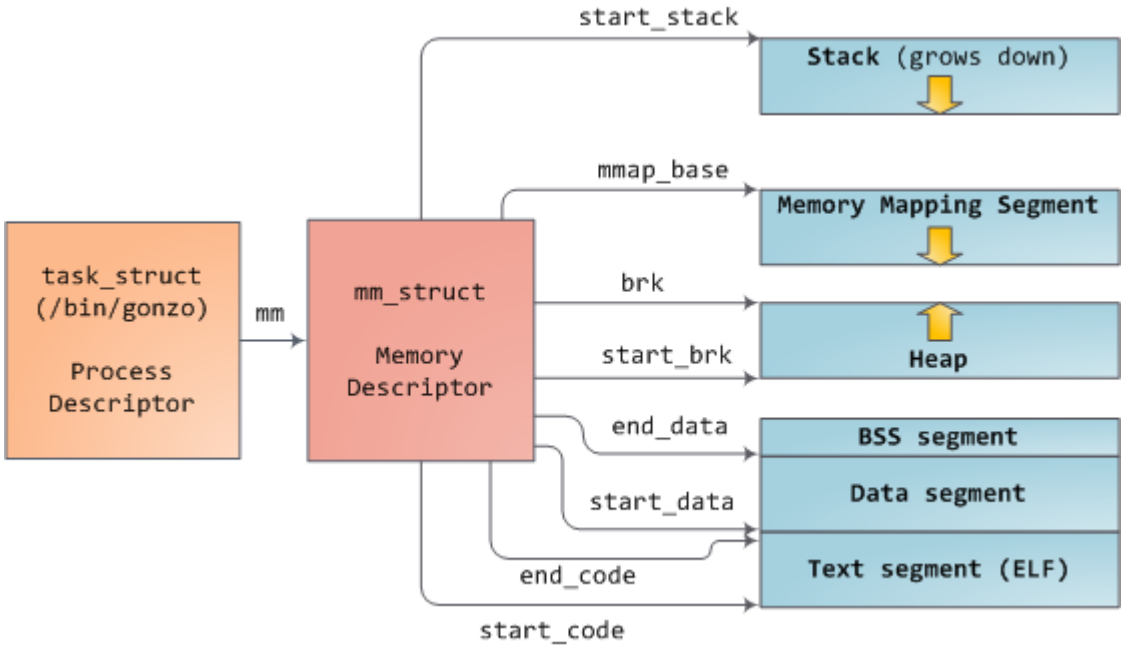


Benefits

- Can perform load/store operations vs input/output

Linux Example #1

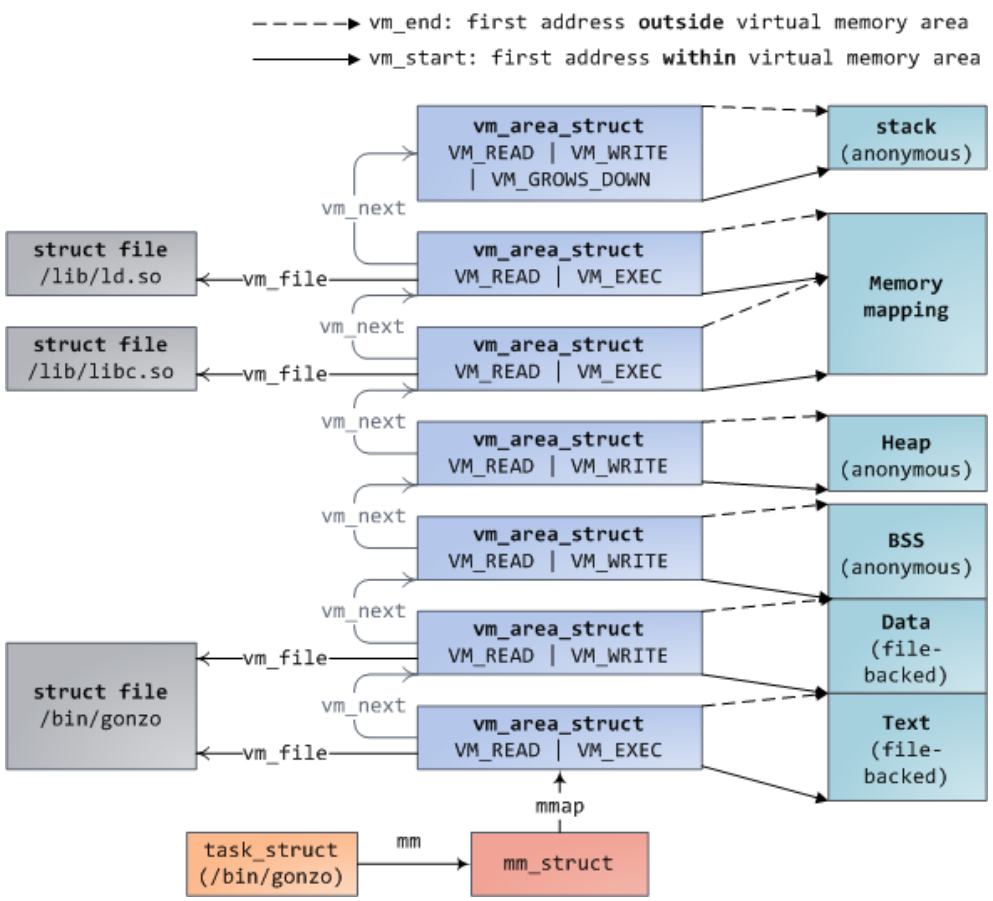
- Objects:



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

Linux Example

- VMA areas
- Anonymous vs. filebacked

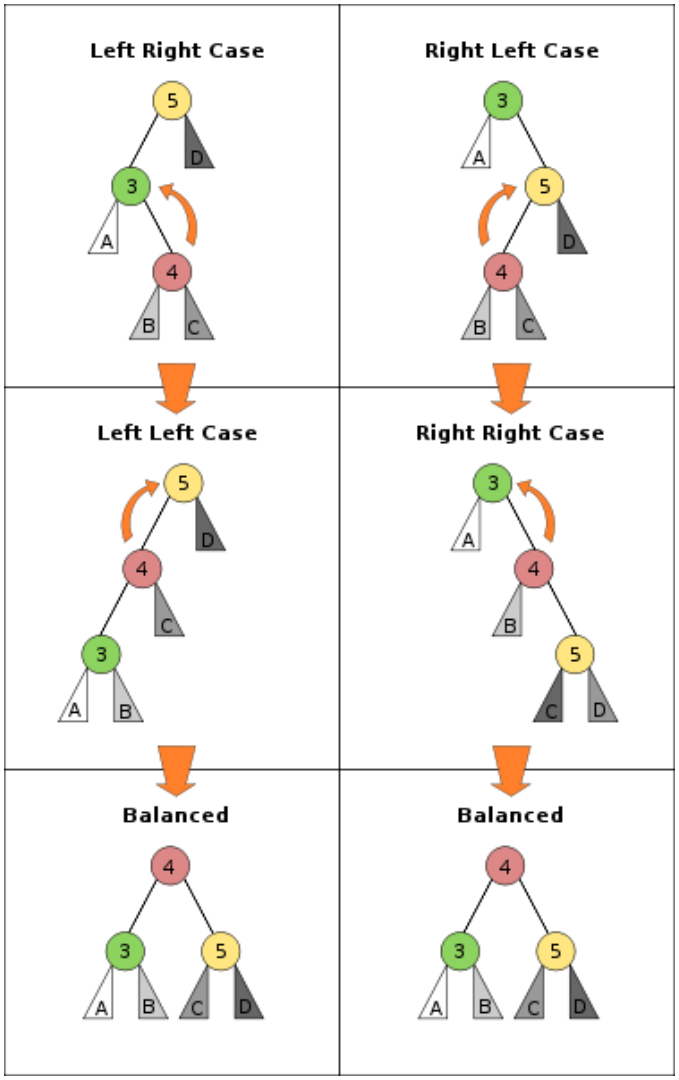


Organization of Memory Regions

- 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

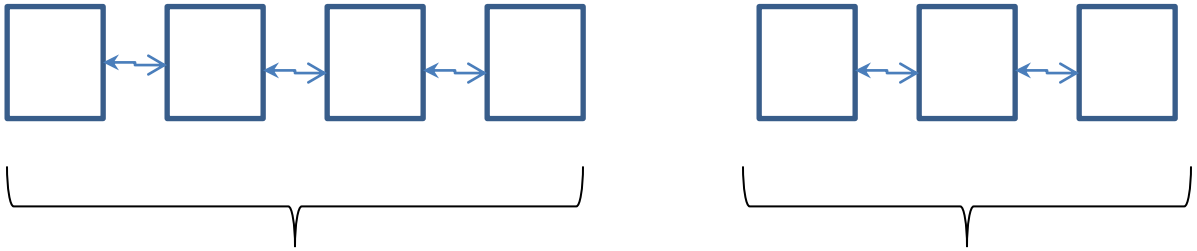
- Organized as balanced tree
- Node [start – end]
- On add/delete
  → rebalance
- Lookup in O(log(n))



Linux MemMgmt LRU Approximation

- Arrange the entries into LRU
- Two queues

FrameTable
(one entry related to
each physical frame)



Active Queue

Inactive Queue

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

Minor Faults forced by setting present=0
and setting "minor-fault sw-bit" in PTE.
Move page from inactive to active and set present

Conclusions

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

Lab3 (how to approach)

- You are emulating HW / SW behavior:
- For every instruction simulated
  - Check whether PTE is present
  - If not present → pagefault

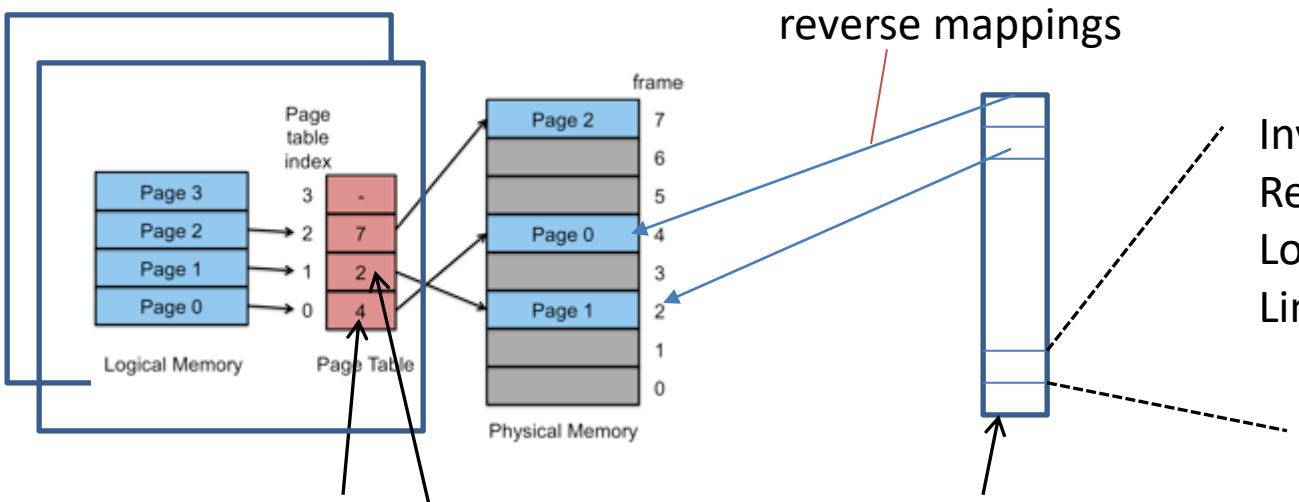
    -> OS must resolve:
    - select a victim frame to replace

      (make pluggable with different replacement algorithms)
    - Unmap its current user (UNMAP)
    - Save frame to disk if necessary (OUT / FOUT)
    - Fill frame with proper content of current instruction's address space (IN, FIN,ZERO)
    - Map its new user (MAP)
    - Its now ready for use and instruction
  - Mark reference/modified bit as indicated by instruction

Lab3

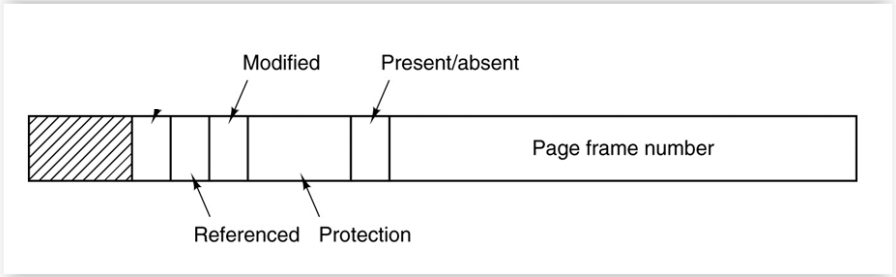
- Create your frame table
- Make all algorithms run through frame table
- Make all algorithms maintains a "hand" from where to start the next "selection"
- On process exit() return all used frames to a free pool and start getting a free frame from there again till free pool is empty; then continue algorithm at hand.
  (this can at times select a frame that was just used .. C'est la vie .. You can make algorithms to complicated, rather make simple and occasionally not make an optimal decision)

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
- Used by OS to keep state for each frame