Purdue University

CS 50011: Introduction to Systems II

Lecture 6: Memory Management and Virtual Memory

Prof. Jeff Turkstra

Lecture 07

- Virtual memory management
- Based on slides by Prof. George Adams III

Typical memory specs

Level	Size in bytes	Typ. access time (ns)
Registers (64)	512	0.25
DRAM (4 GB)	4,294,967,296	60.00
Hard disk (1 TB)	1,000,000,000,000	10,000,000.00

- Use of hard disk should be carefully managed for performance reasons

- “… a system has been devised to make the core and drum* combination appear to the programmer as a single level store, the requisite transfers taking place automatically.”
  - Kilburn et al., “One-level storage systems”, 1962

Motivation

- Efficient and safe (correct) sharing of memory among multiple programs
- Permit caching of hard drive data in main memory
- Allow programs to run even if footprint is larger than available main memory
- Sometimes motivations change as technology changes

Memory management

- Suppose we have:
  - Internet Exploder (100MB)
  - Micro$oft Word (100MB)
  - Yahoo Messenger (30MB)
  - Operating System (200MB)
- Computer has 256MB of RAM
  - Have to quit programs before starting others
- Virtual memory allows us to load/unload portions as needed

Programmer burden

- Without virtual memory, it's the programmer's job to make programs fit
  - Divide into mutually exclusive chunks
  - Dynamically load/unload chunks as needed
  - Same for libraries
- Sounds like fun. Not.

Isolation

- Virtual memory limits sharing to explicit cases
- How? Every program has its own address space
- Virtual memory translates virtual addresses to physical addresses
- Also enforces protection

More efficient memory utilization

- Keep in RAM only the portion of address space currently in use
  - Working set, Peter Denning, former head of Purdue CS department
- Swap space
- Can do deduplication to some degree
  - Shared libraries
  - Multiple processes of the same program

Can speed OS tasks

- Program loading
  - Demand-based, faulted in instead of loaded all at once
- Fork and copy-on-write
  - Again, no duplication of memory unless needed
  - Spawning new processes is fast
  - Critical for fork() / exec() paradigm

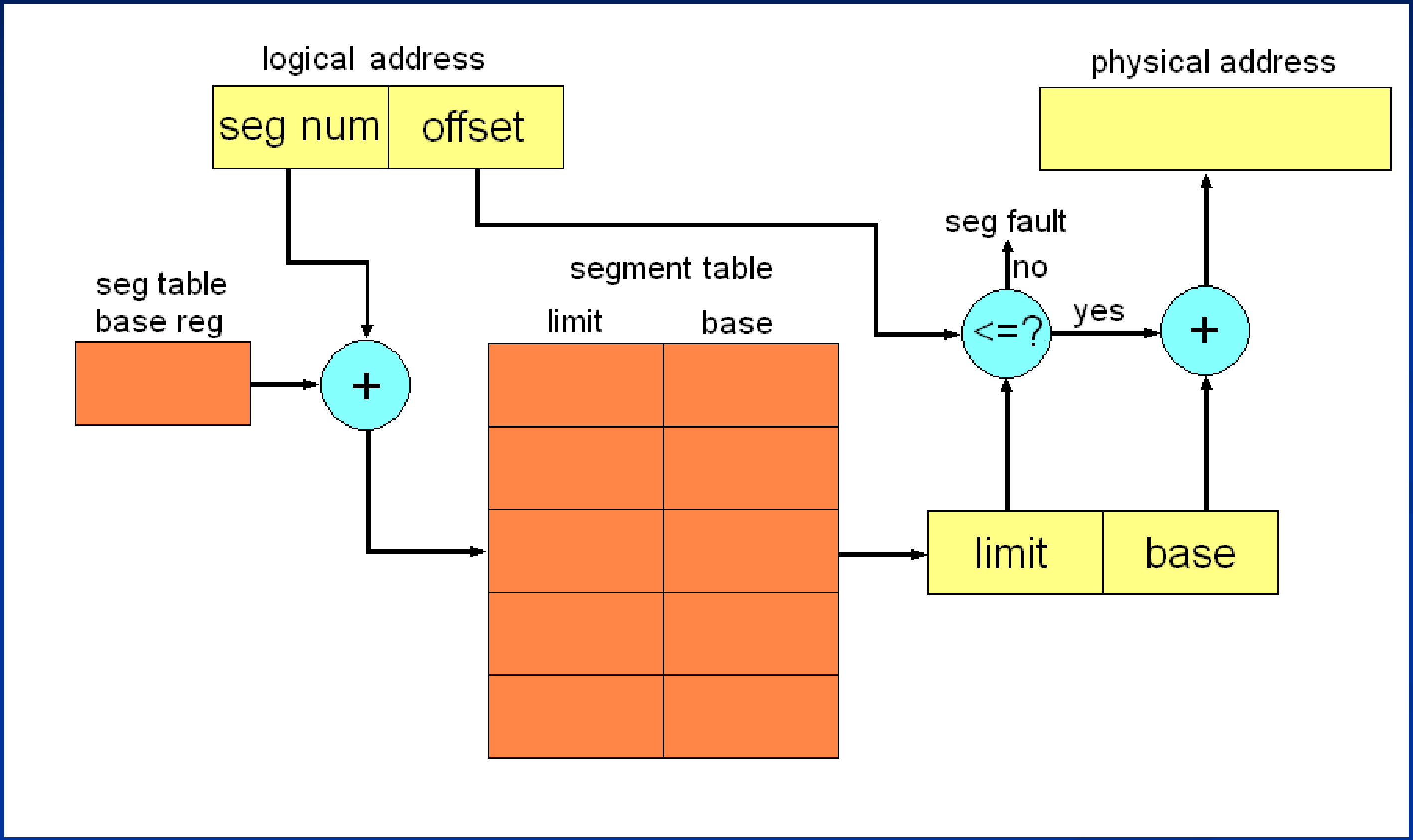
Sharing

- Permits simple, dynamic sharing among processes
  - Point the virtual addresses to the same physical addresses

Implementations

- Historic
  - Process swapping – entire memory footprint of process moved in and out (swapped) between memory and disk
  - Segment swapping – entire parts, “segments” (determined by programmer) are swapped
- Drawbacks
  - Too much information at a time
  - Slow, inefficient
  - Fragmentation

Segmentation



* http://cs.bc.edu/~donaldja/362/addresstranslation.html

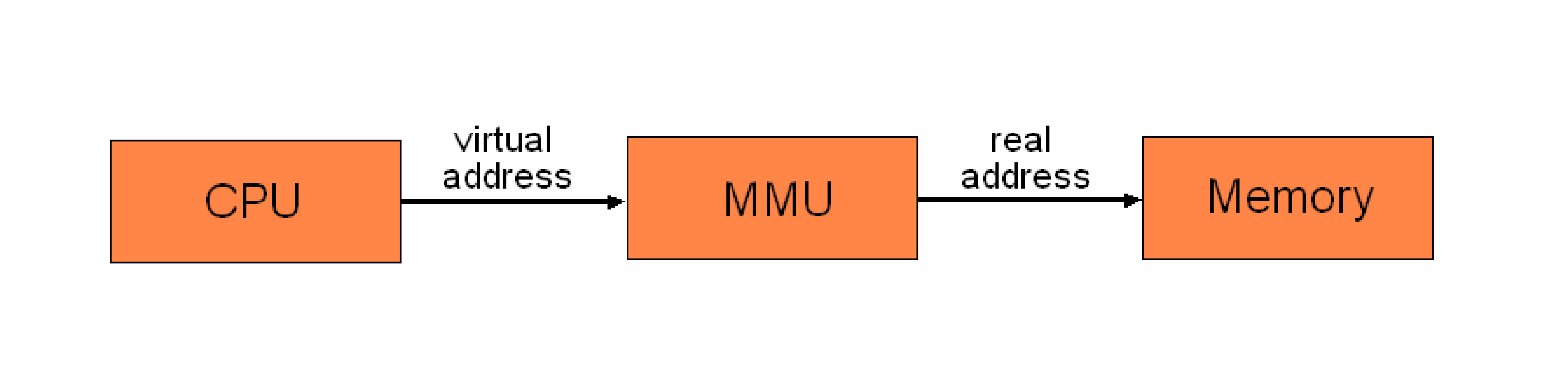
Demand-based paging

- Unit of memory swapped is a fixed-size page
  - Usually 4KiB now, can be 2MiB on x86_64 “long mode”
    - Also supports 1GB
- Eliminates external fragmentation
  - Not internal fragmentation

- Time to load page is huge, 10^7 nanoseconds
- Main memory operates as a fully associative cache
- Try to avoid loading a page multiple times
  - Only compulsory misses and capacity misses

Terminology

- Physical memory divided into frames
- Virtual memory into pages
  - Any page can be placed in any frame
- Missing page? Called a page fault
- CPU emits virtual addresses
  - Translated/mapped by a combination of hardware and software
  - Memory mapping or address translation

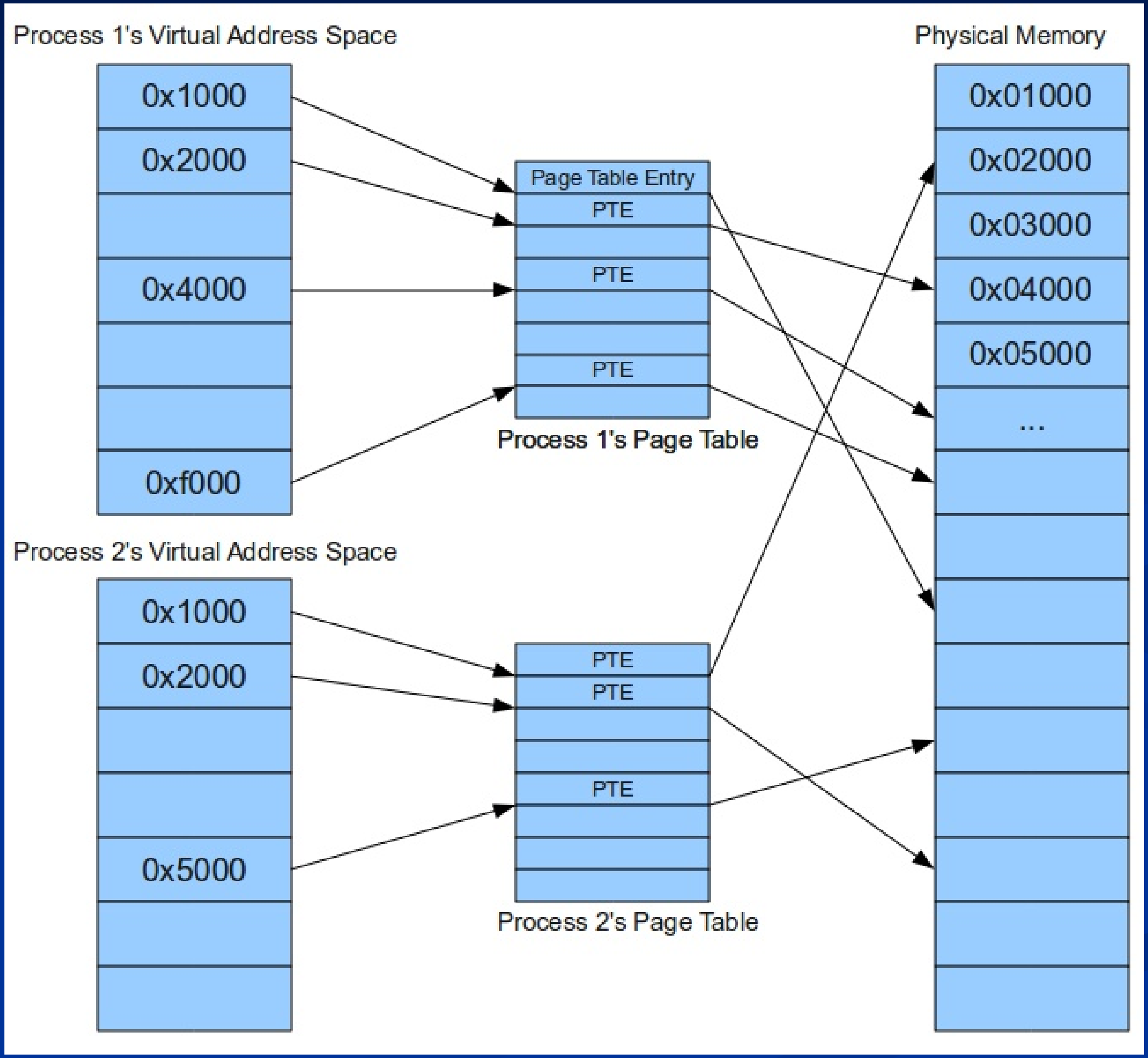

CPU
virtual
address
MMU
real
address
Memory

- Pages currently residing in main memory are resident
- Resident set refers to all in-memory pages for a given process
  - Ideally resident set ~= working set

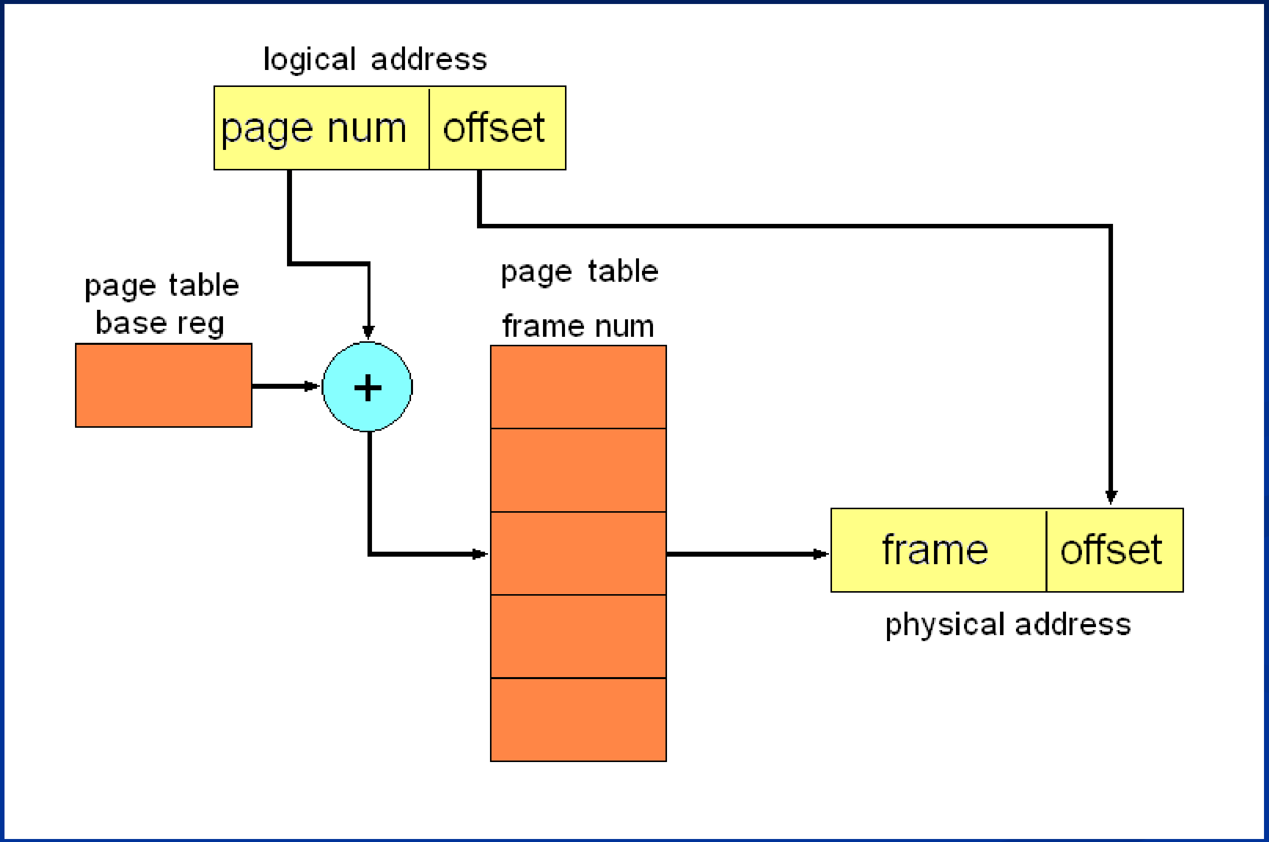
Page tables

- Page tables provide the mapping from a virtual address to a physical address
  - Stored in main memory
  - Managed by the OS
  - Referenced by the MMU

Virtual memory



Translation

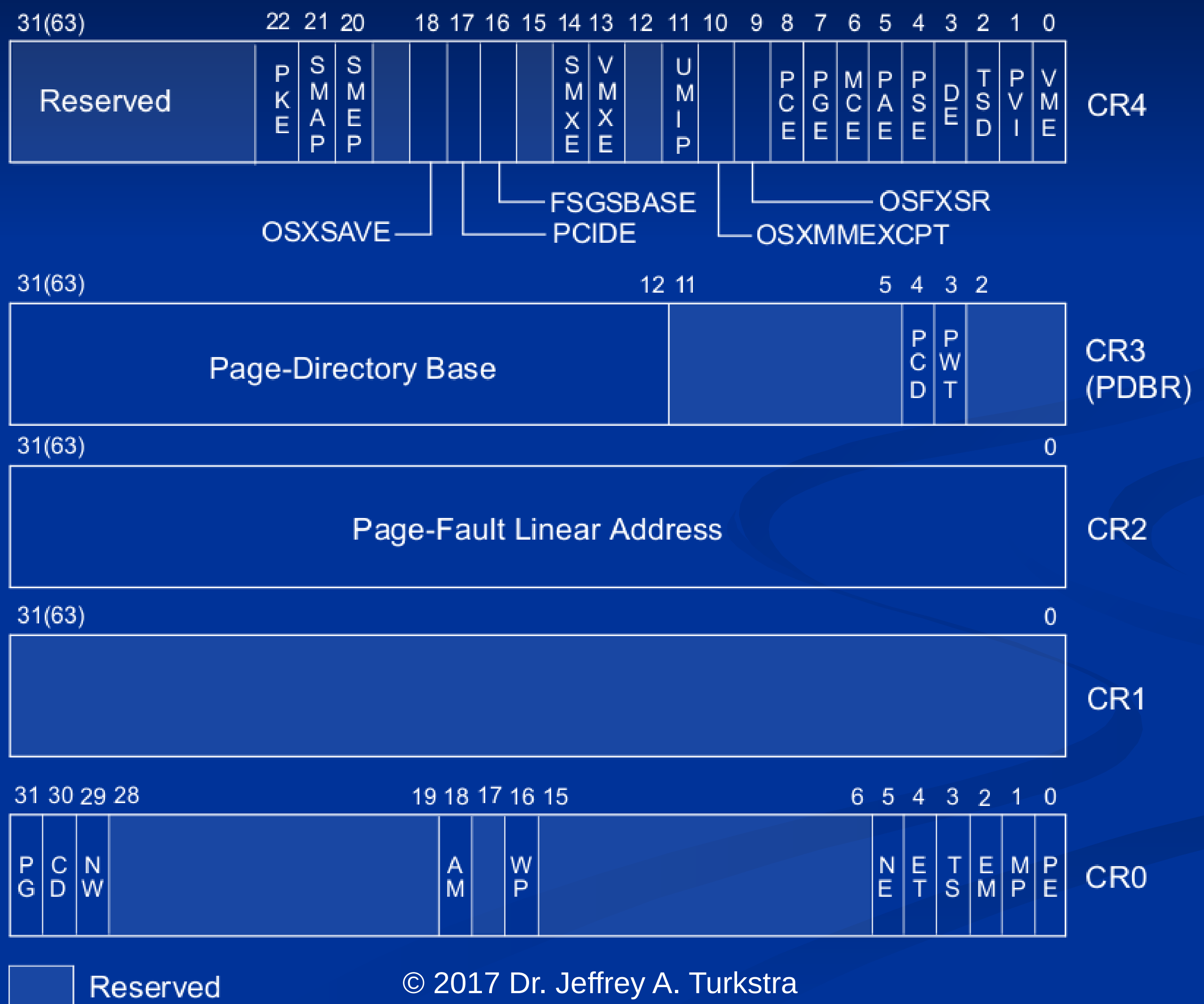


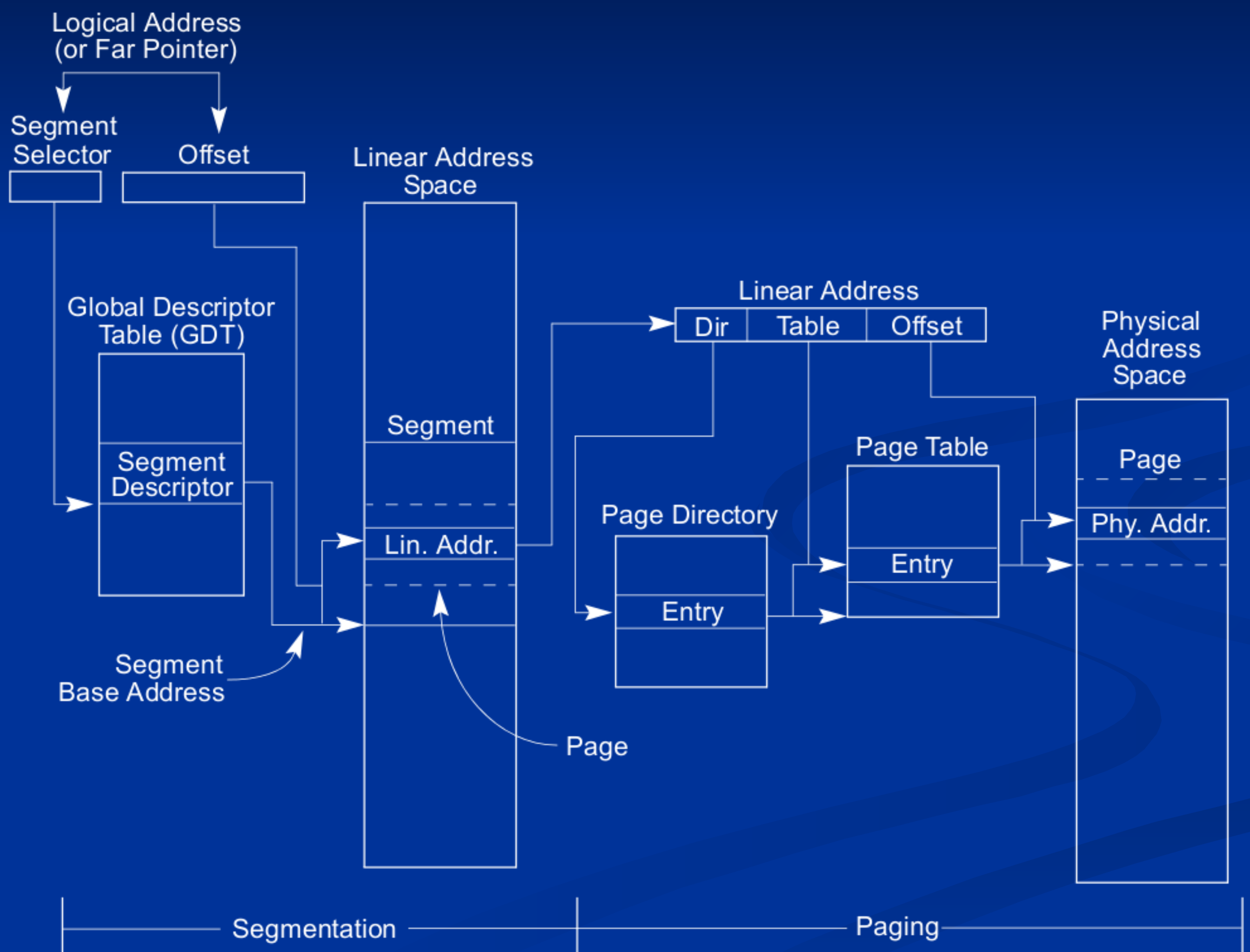
* http://cs.bc.edu/~donaldja/362/addresstranslation.html

Hardware/software approach

- Hardware handles the common case
  - Translate virtual address for a resident page to a physical address/frame
- Software invoked for exceptions
  - Page fault – moving pages between disk and memory
  - Context switches
  - Configuring hardware

Control registers



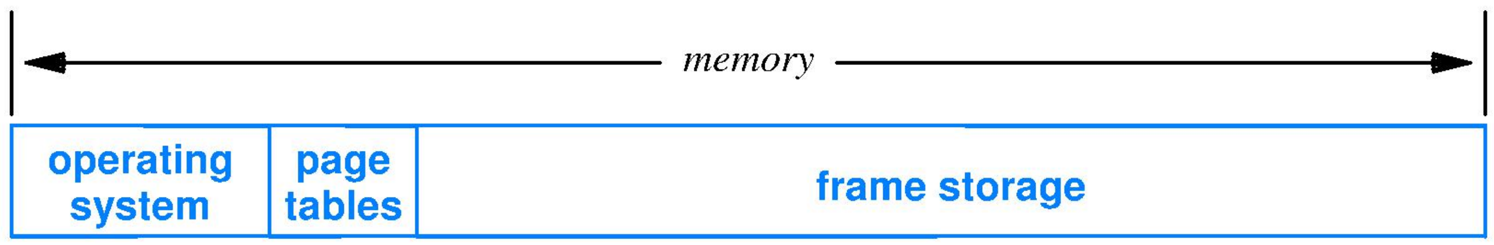

Logical Address
(or Far Pointer)
Segment
Selector
Offset
Linear Address
Space
Global Descriptor
Table (GDT)
Segment
Descriptor
Segment
Lin. Addr.
Segment
Base Address
Page
Linear Address
Dir
Table
Offset
Page Directory
Entry
Page Table
Entry
Physical
Address
Space
Page
Phy. Addr.
Segmentation
Paging

Page table

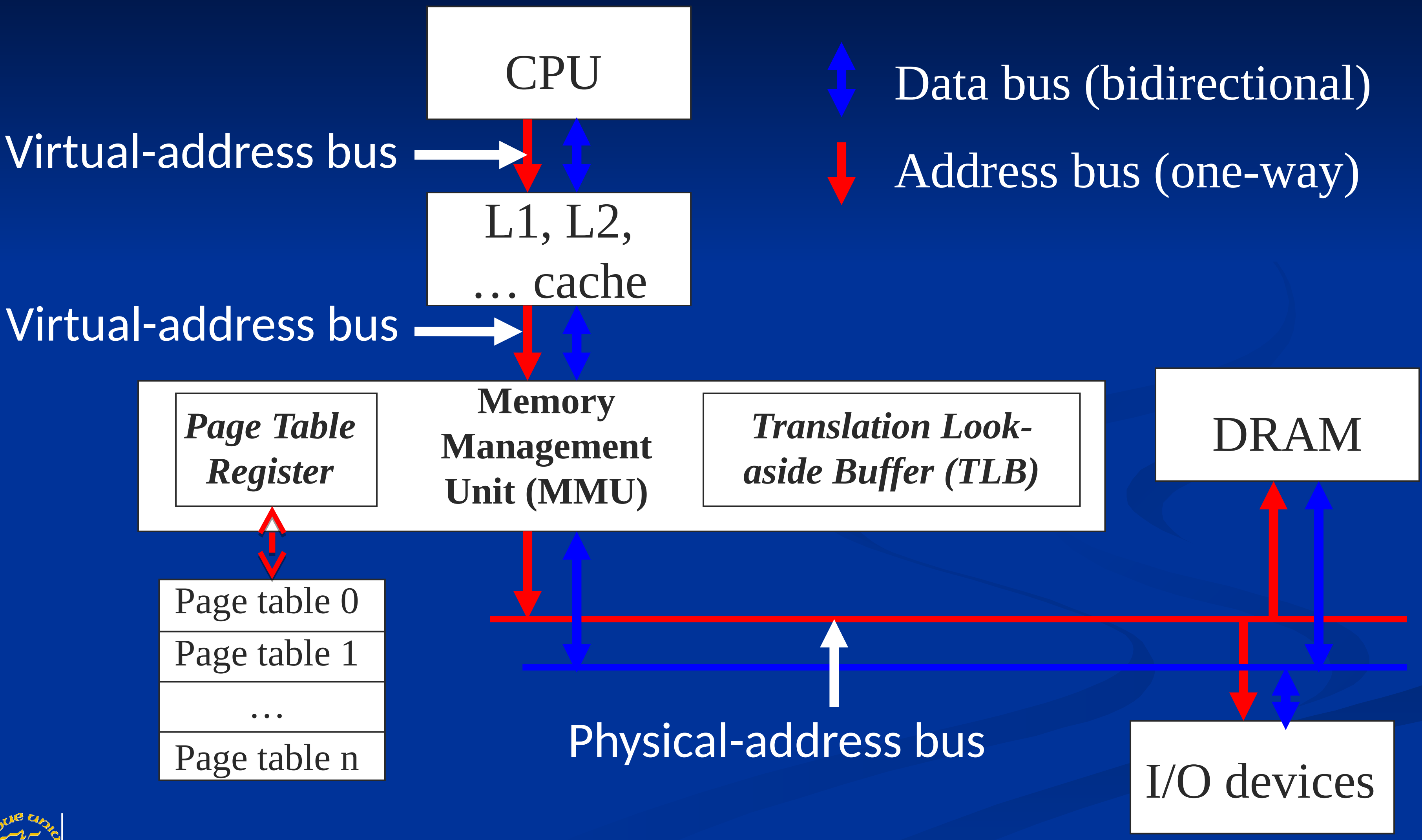
Virtual address range	Virtual page	Mapped to	Page metadata
0x00000000 to 0x00000FFF	$0x00000 = (0)_{10}$	Page frame 0x024	Read only, not Dirty, Executable (Text)
0x00001000 to 0x00001FFF	0x00001	Page frame 0xF05	Read, write (Data), not Dirty
0x00002000 to 0x00002FFF	0x00002	Page frame 0xXXX	Invalid
0x00003000 to 0x00003FFF	0x00003	Swap space (via inode)	Read, write, Swap, not Dirty
…	…	…	…
0xFFFFE000 to 0xFFFFEFFF	0xFFFFE	Page frame 0xF43	Read, write, not Dirty
0xFFFFF000 to 0xFFFFFFFF	$0xFFFFF = (2^{20}-1)_{10}$	Page frame 0x010	Read, write, Dirty

Page tables

- Are stored in main memory



- Memory management unit (MMU)
  - Historically external, newer architectures include it on-chip


CPU
Data bus (bidirectional)
Virtual-address bus
Address bus (one-way)
L1, L2, … cache
Virtual-address bus
Page Table Register
Memory Management Unit (MMU)
Translation Look-aside Buffer (TLB)
DRAM
Page table 0
Page table 1
…
Page table n
Physical-address bus
I/O devices

Virtually addressed caches

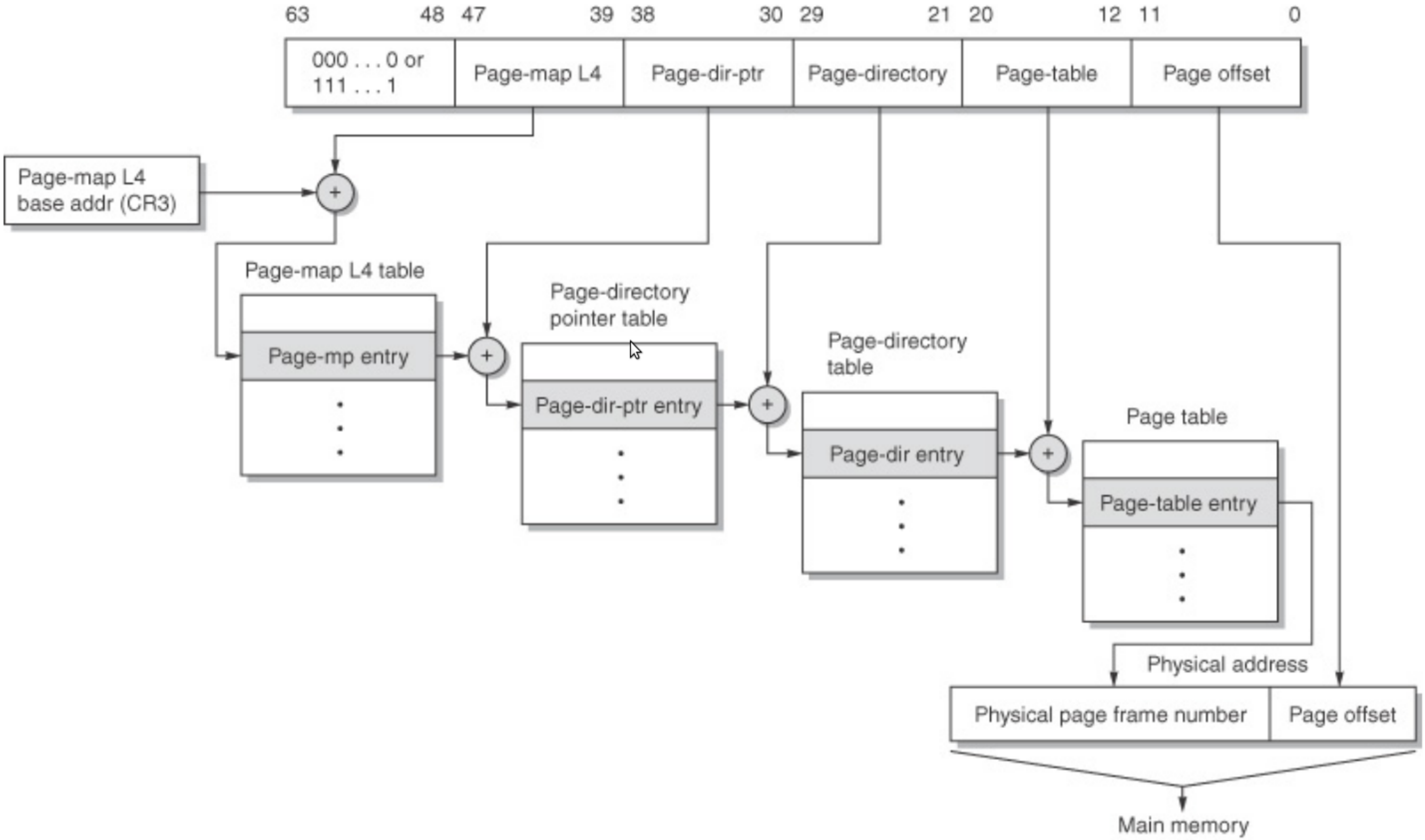
- MMU/TLB below the cache
- Have to distinguish virtual addresses from different processes
  - Invalidate all entries on context switch
  - Augment cache to include ASID (address space identifier) along with tag
    - Still have to worry about aliasing
- Some designs have MMU/TLB above the cache

Fast translation is critical

- Translation lookaside buffer, TLB
  - Caches recent translations
- Invented by IBM
- MMU looks in TLB same time as page table lookup starts
  - In TLB, win!
  - Program locality → 90% of the time or more
- Context switches? Tagged TLB or TLB shootdown

Multi-level page tables

- For modern, large address spaces they are necessary
- Used even in 32-bit land
- Page directory
- Or...


63
48
47
39
38
30
29
21
20
12
11
0
000 . . . 0 or
111 . . . 1
Page-map L4
Page-dir-ptr
Page-directory
Page-table
Page offset
Page-map L4
base addr (CR3)
Page-map L4 table
Page-mp entry
Page-directory
pointer table
Page-dir-ptr entry
Page-directory
table
Page-dir entry
Page table
Page-table entry
Physical address
Physical page frame number
Page offset
Main memory

PTE metadata

- Valid bit, dirty bit
- Permission bits
- Page replacement algorithm support
- LRU, etc

Processing a page fault

- Program attempts read/write to non-resident page
  - Fetch next instruction
  - Access non-resident data
- MMU attempts translation, finds valid bit = 0
  - Generates interrupt to CPU

- Interrupt handler saves return address and registers
- Jumps to appropriate handler
  - If page is invalid, SIGSEGV
  - If valid, load it from disk and establish the mapping
    - What if there are no free frames?
- Restore registers, resume executing instruction

Restarting instructions

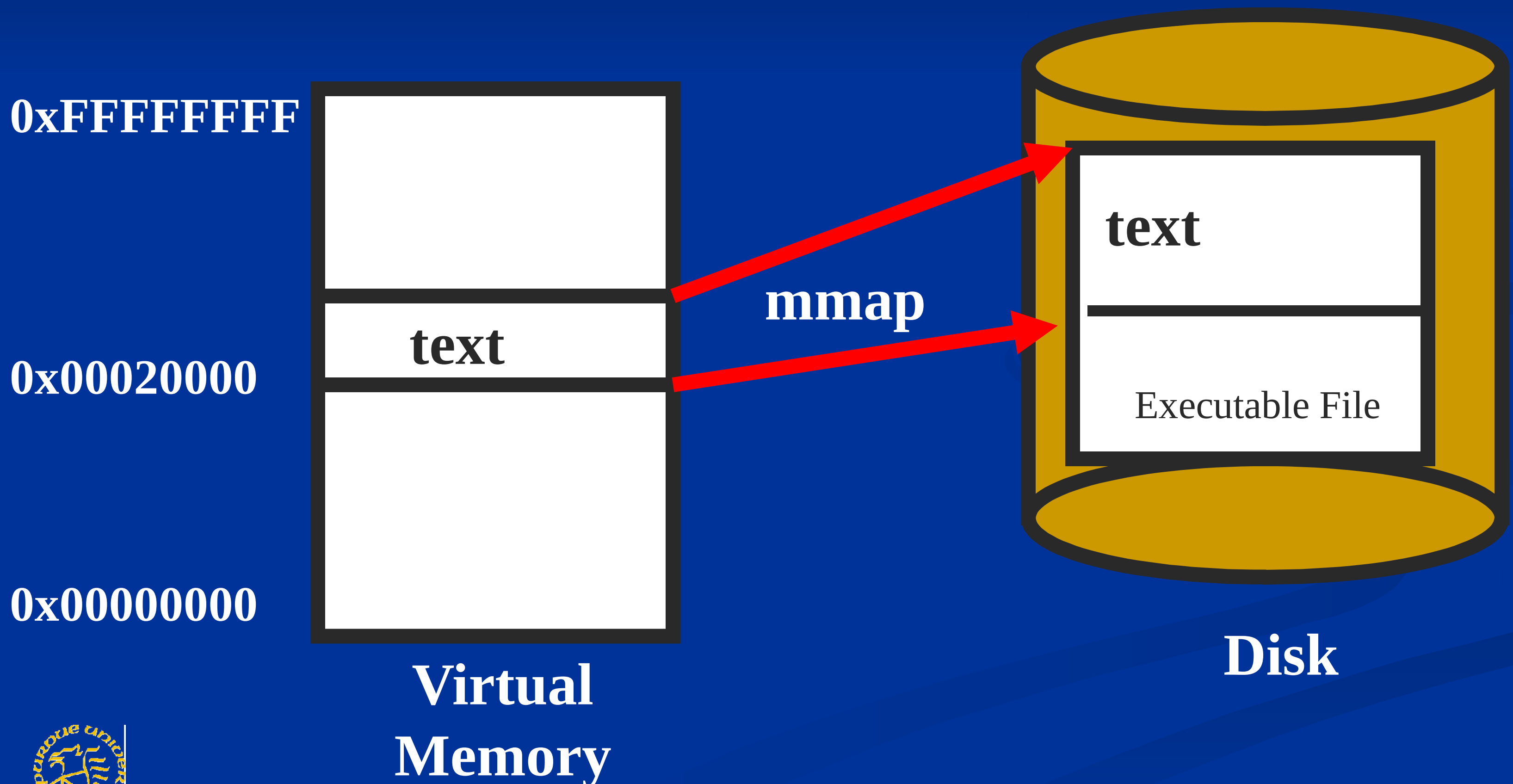
- Not always easy
- Page fault may have been in the middle of an instruction
  - Can it be skipped?
  - Restart from beginning?
    - Where?
  - Side effects (eg, autoincrementing address modes)
- Hardware support for tracking side effects and rolling back

Page replacement

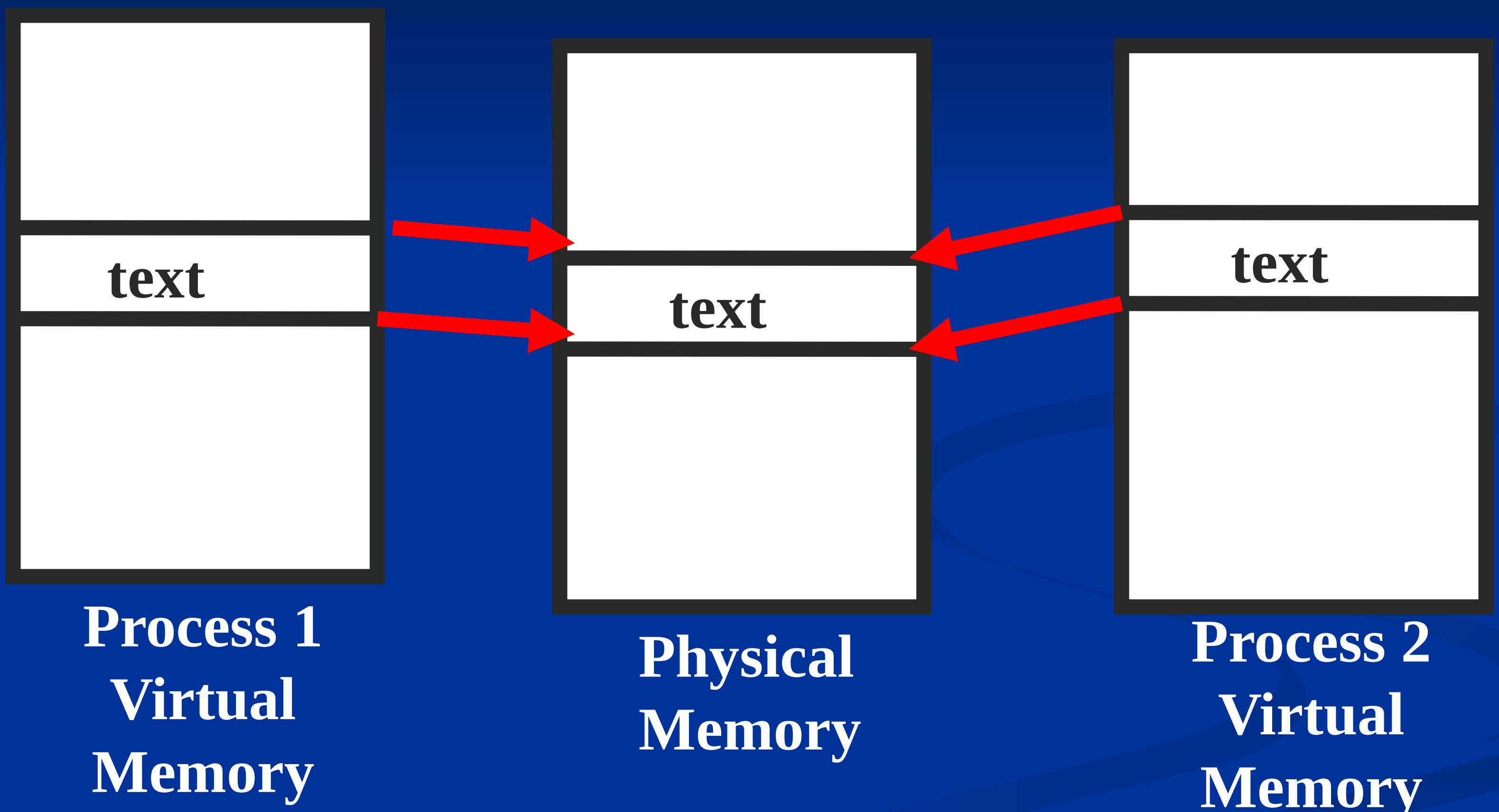
- Optimal (MIN), Belady's Algorithm
  - Replace pages that won't be used for longest time
  - Only works offline, minimal page faults though
- FIFO
- NRU
- FIFO with second chance, "Clock Algorithm"
- etc

mmap()

- Text segment, for example
- Pages not read by default
  - On-demand as accessed
  - Fast startup
  - Reduced memory usage
- Physical pages for text segment and shared libraries can be shared
  - Protections: PROT_READ|PROT_EXEC
  - MAP_PRIVATE

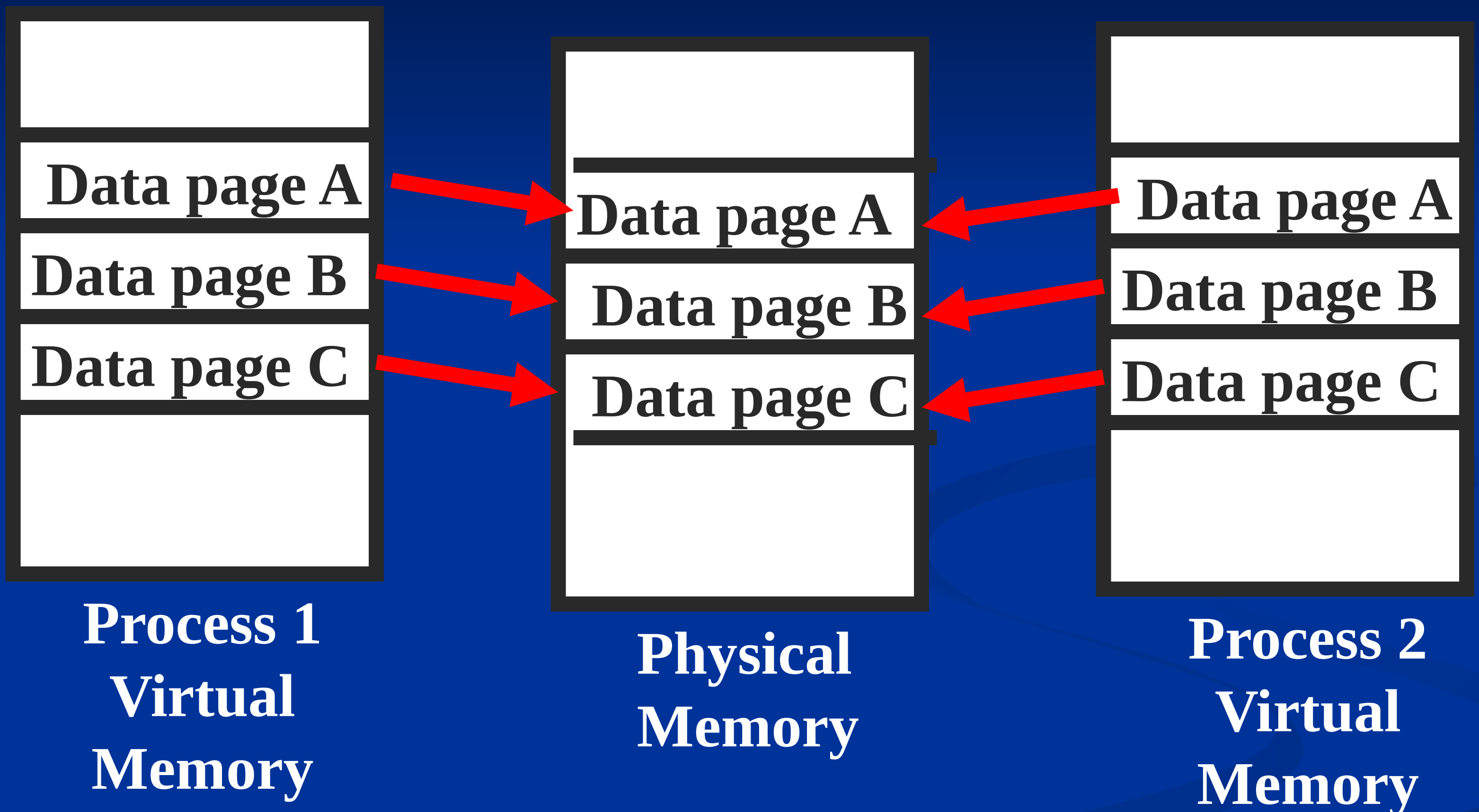

0xFFFFFFFF
0x00020000
0x00000000
text
Virtual Memory
mmap
text
Executable File
Disk

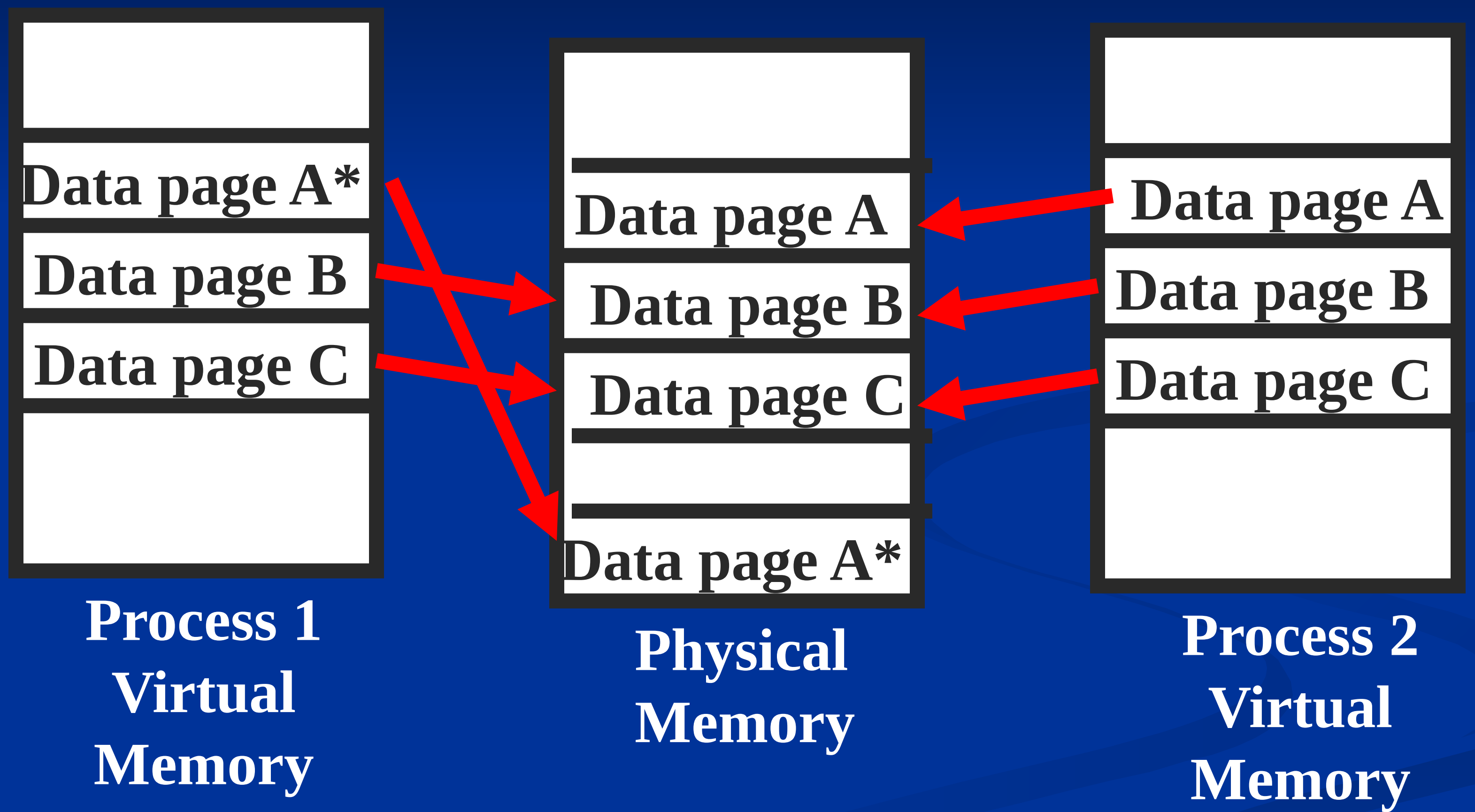
Shared code/library



Data segment

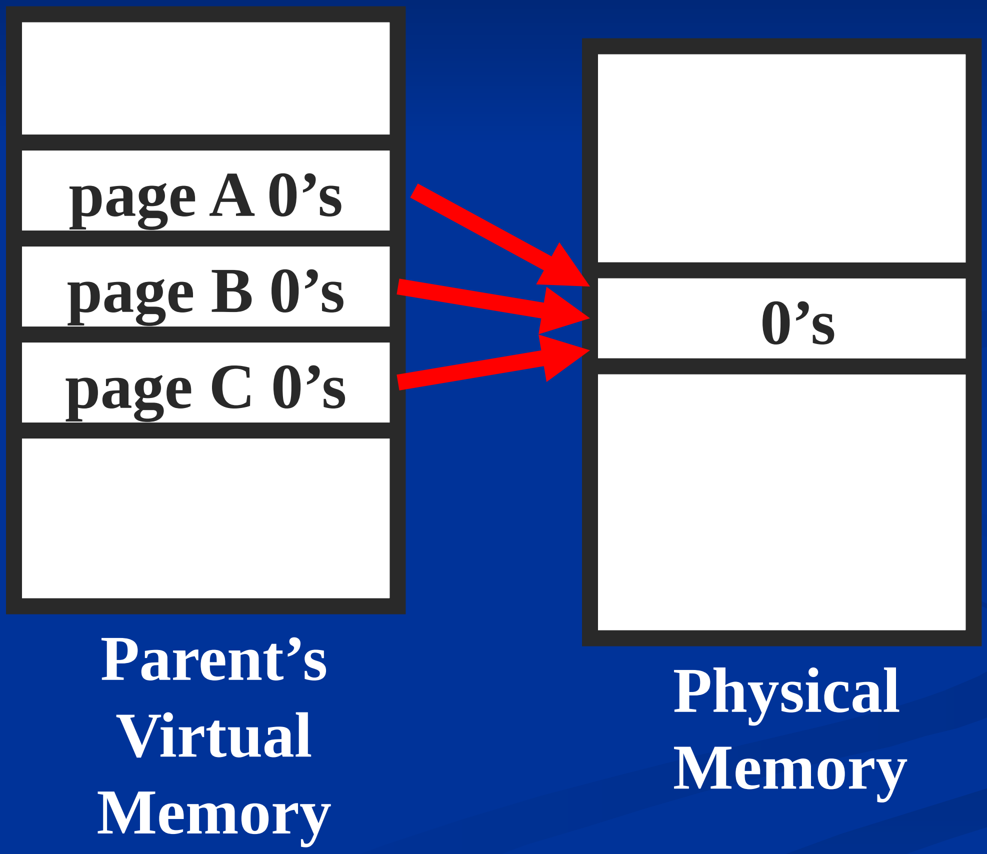
- Multiple instances?
  - Data shared until write
  - “Copy on write”

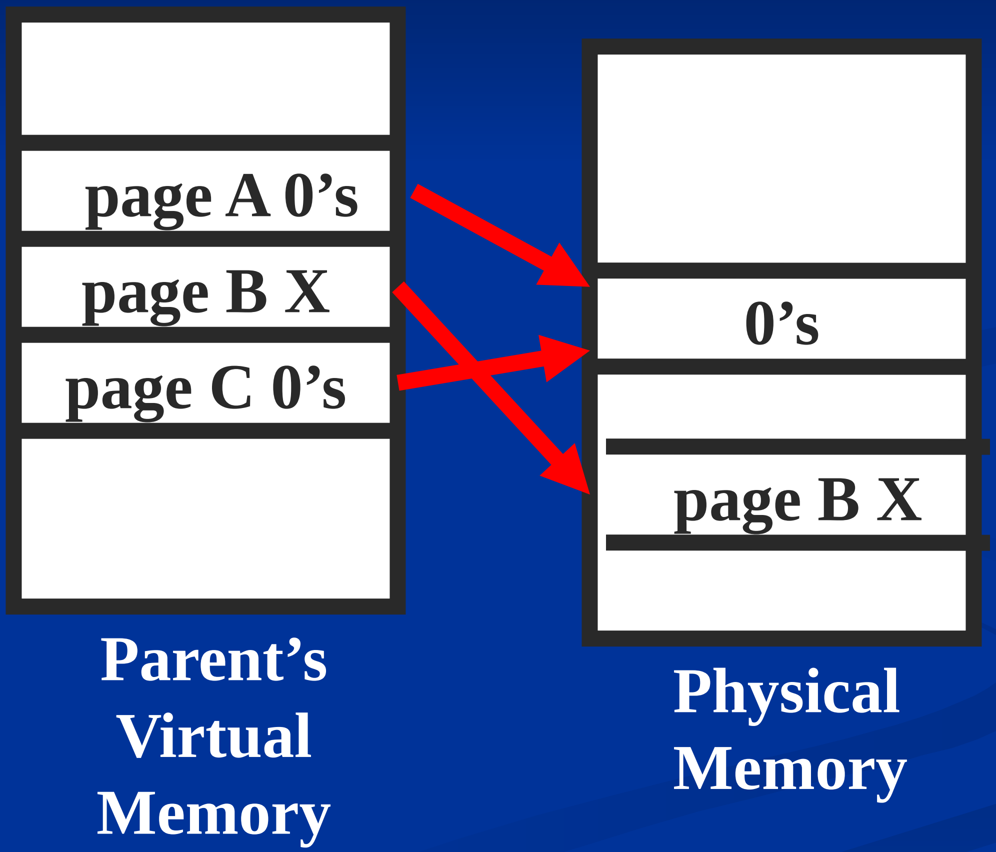

Data page A
Data page B
Data page C
Process 1
Virtual
Memory
Data page A
Data page B
Data page C
Physical
Memory
Data page A
Data page B
Data page C
Process 2
Virtual
Memory
41


Data page A*
Data page B
Data page C
Process 1 Virtual Memory
Data page A
Data page B
Data page C
Data page A*
Physical Memory
Data page A
Data page B
Data page C
Process 2 Virtual Memory

Copy-on-write

- Happens on fork() too
- Critical optimization that allows fork()/exec() approach to work


page A 0’s
page B 0’s
page C 0’s
0’s
Parent’s Virtual Memory
Physical Memory


page A 0’s
page B X
page C 0’s
Parent’s Virtual Memory
0’s
page B X
Physical Memory

Questions?