Addressing and Virtual Memory (plus some UNIX and Mach)

Tom Anderson

Winter 2017

Shell

- A shell is a job control system
  - Allows programmer to create and manage a set of programs to do some task
  - Windows, MacOS, Linux all have shells

- Example: to compile a C program

```
cc –c sourcefile1.c
cc –c sourcefile2.c
ln –o program sourcefile1.o sourcefile2.o
```

Windows CreateProcess

- System call to create a new process to run a program
  - Create and initialize the process control block (PCB) in the kernel
  - Create and initialize a new address space
  - Load the program into the address space
  - Copy arguments into memory in the address space
  - Initialize the hardware context to start execution at ``start''
  - Inform the scheduler that the new process is ready to run

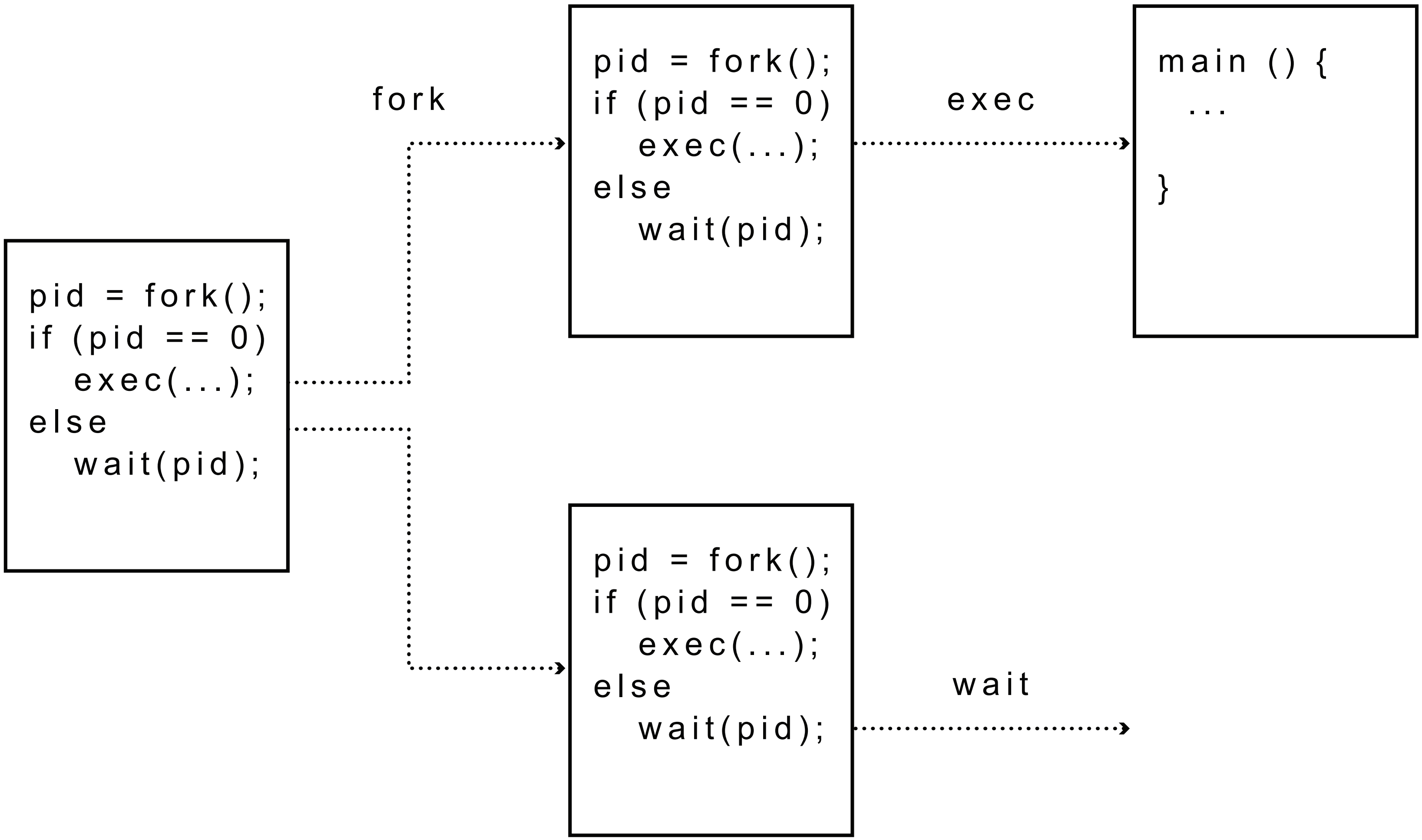
Windows CreateProcess API (simplified)

```
if (!CreateProcess(
   NULL,          // No module name (use command line)
   argv[1],       // Command line
   NULL,          // Process handle not inheritable
   NULL,          // Thread handle not inheritable
   FALSE,         // Set handle inheritance to FALSE
   0,             // No creation flags
   NULL,          // Use parent's environment block
   NULL,          // Use parent's starting directory
   &si,           // Pointer to STARTUPINFO structure
   &pi )          // Pointer to PROCESS_INFORMATION structure
)
```

UNIX Process Management

- UNIX fork – system call to create a copy of the current process, and start it running
  - No arguments!
  - Returns 0 to child, child process ID to parent
- UNIX exec – system call to change the program being run by the current process
- UNIX wait – system call to wait for a process to finish
- UNIX signal – system call to send a notification to another process

UNIX Process Management



```
pid = fork();
if (pid == 0)
   exec(...);
else
   wait(pid);
```

```
pid = fork();
if (pid == 0)
   exec(...);
else
   wait(pid);
```

```
main () {
  ...

}
```

```
pid = fork();
if (pid == 0)
   exec(...);
else
   wait(pid);
```

Questions

- Can UNIX fork() return an error? Why?
- Can UNIX exec() return an error? Why?
- Can UNIX wait() ever return immediately? Why?

Implementing UNIX fork

Steps to implement UNIX fork

- Create and initialize the process control block (PCB) in the kernel
- Create a new address space
- Initialize the address space with a copy of the entire contents of the address space of the parent
- Inherit the execution context of the parent (e.g., any open files)
- Inform the scheduler that the new process is ready to run

Implementing UNIX exec

- Steps to implement UNIX fork
  - Load the program into the current address space
  - Copy arguments into memory in the address space
  - Initialize the hardware context to start execution at ``start''

UNIX I/O

- Uniformity
  - All operations on all files, devices use the same set of system calls: open, close, read, write
- Open before use
  - Open returns a handle (file descriptor) for use in later calls on the file
- Byte-oriented
- Kernel-buffered read/write
- Explicit close
  - To garbage collect the open file descriptor

UNIX File System Interface

- UNIX file open is a Swiss Army knife:
  - Open the file, return file descriptor
  - Options:
    - if file doesn't exist, return an error
    - If file doesn't exist, create file and open it
    - If file does exist, return an error
    - If file does exist, open file
    - If file exists but isn't empty, nix it then open
    - If file exists but isn't empty, return an error
    - …

Interface Design Question

- Why not separate syscalls for open/create?

```
if (!exists(name))
    create(name);   // can create fail?
fd = open(name);   // does the file exist?
```

UNIX Retrospective

- Designed for computers 10^-8 slower than today
- Radical simplification relative to Multics
- Key ideas behind the project
    1. ease of programming and interactive use
    2. size constraint: underpowered machine, small memory
    3. Eat your own dogfood. UNIX dev on UNIX.

Question

- We are still using UNIX (on servers, on smartphones) because of
  - its system call API?
  - its simple implementation?
  - Its shell can can be scripted?
  - Its file directories are stored as files?
  - …

Christensen Disruption

- Attempt to answer a puzzle
  - Why do very successful tech companies often miss the most important tech shifts?
- Disruption !=
  - anytime a tech company goes bankrupt
  - any tech advance
- Lesson: what makes you strong kills you

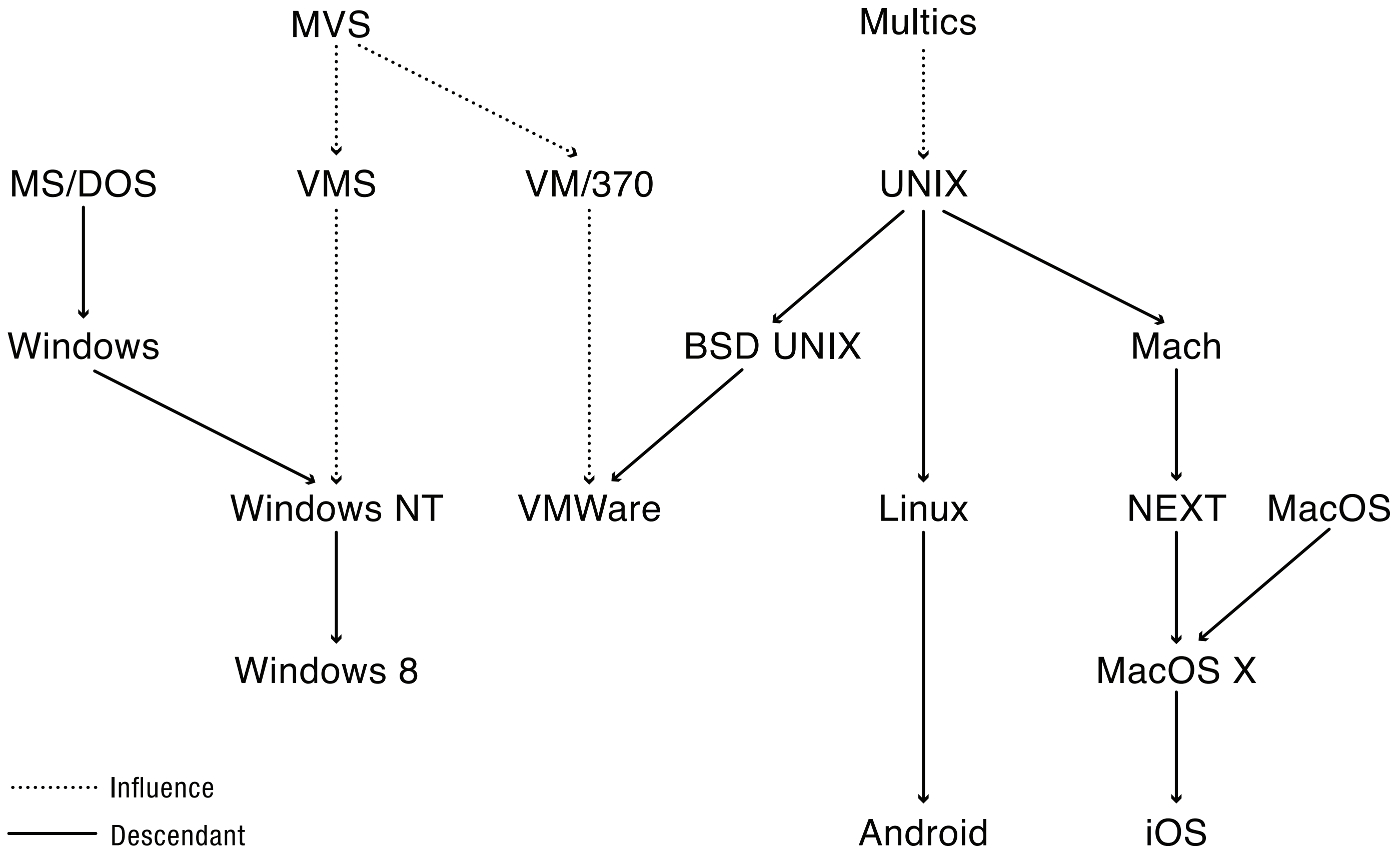
Christensen Disruption

- Successful companies do what their customers want
  - Incorporate new tech, if it is “better”
- Disruptive technology design pattern
  - Worse for typical user, but less expensive
  - Both old and new tech improve over time
  - Eventually, new tech is cheaper and good enough

Operating Systems

- Lowest layer of software on the computer
- Hides specifics of underlying hardware from users, developers
  - Portability => hardware as commodity
- API for developers
  - Ease of use => application lock in
- User facing
  - User lock in

OS History


MVS
Multics
MS/DOS
VMS
VM/370
UNIX
Windows
BSD UNIX
Mach
Windows NT
VMWare
Linux
NEXT
MacOS
Windows 8
MacOS X
Influence
Descendant
Android
iOS

Other Examples

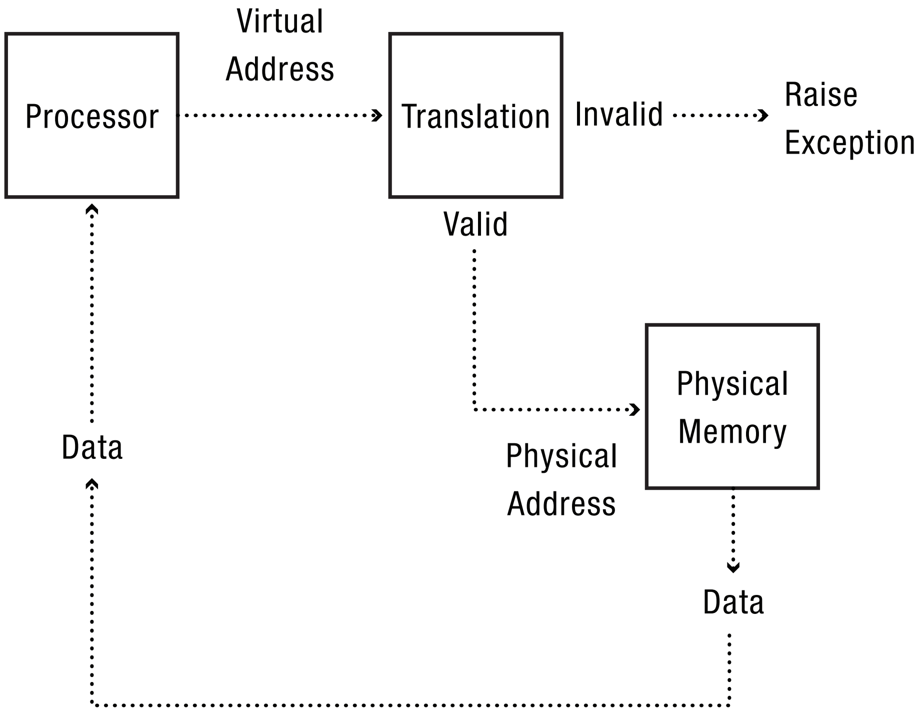
- Internet vs. telephony
- Web vs. network file systems
- SQL vs. hierarchical databases
- Mapreduce vs. SQL
- C vs. Fortran (or assembly)
- Java/Python/Go vs. C++

Address Translation

Main Points

- Address Translation Concept
  - How do we convert a virtual address to a physical address?
- Flexible Address Translation
  - Segmentation
  - Paging
  - Multilevel translation
- Efficient Address Translation
  - Translation Lookaside Buffers
  - Virtually and physically addressed caches

Address Translation Concept



Address Translation Goals

- Memory protection
- Memory sharing
  - Shared libraries, interprocess communication
- Sparse addresses
  - Multiple regions of dynamic allocation (heaps/stacks)
- Efficiency
  - Memory placement
  - Runtime lookup
  - Compact translation tables
- Portability

Bonus Feature

- What can you do if you can (selectively) gain control whenever a program reads or writes a particular virtual memory location?
- Examples:
  - Copy on write
  - Zero on reference
  - Fill on demand
  - Demand paging
  - Memory mapped files
  - …

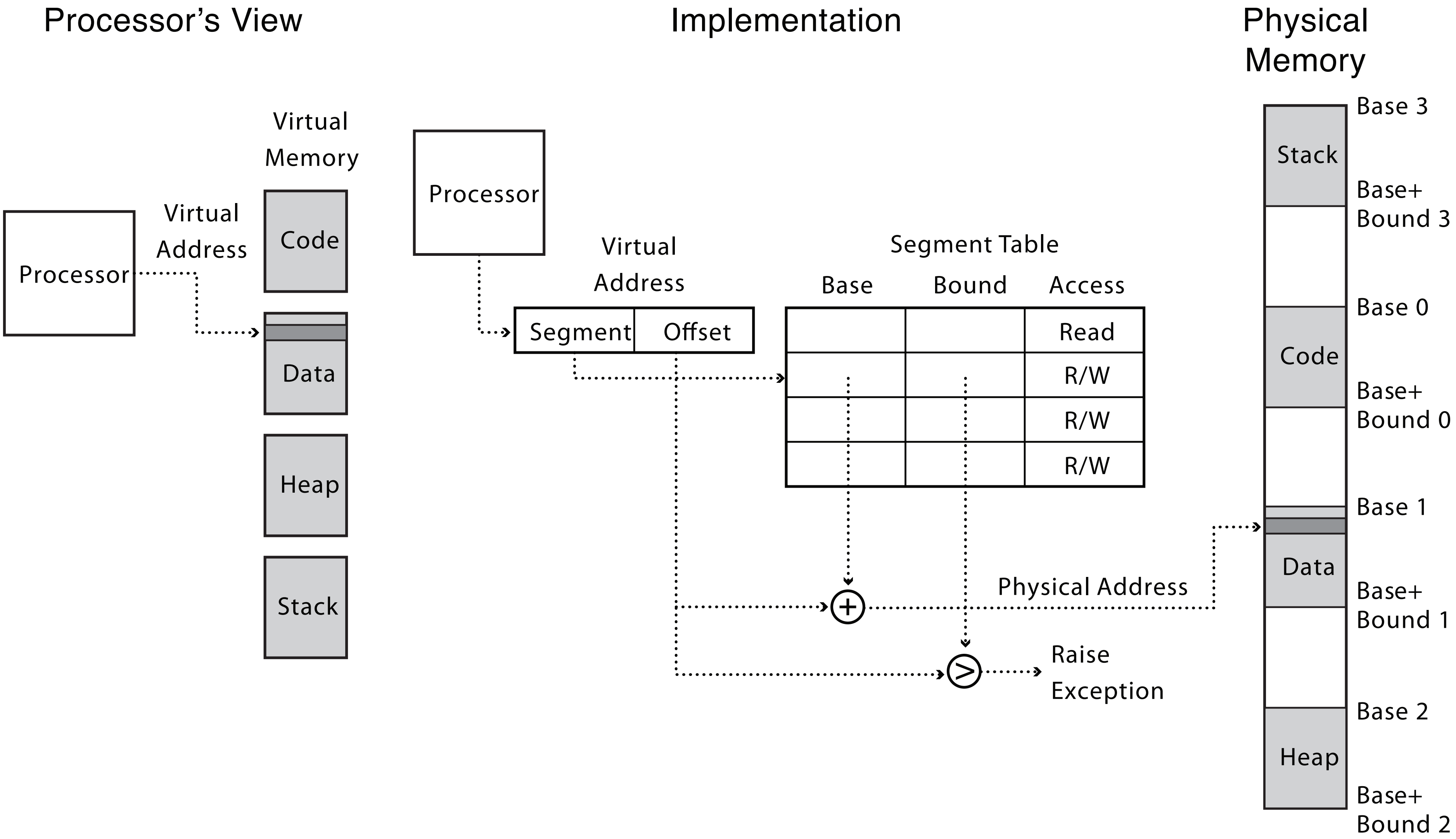
Process Regions or Segments

- Every process has logical regions or segments
  - Contiguous region of process memory
  - Code, data, heap, stack, dynamic library (code, data), memory mapped files, …
  - Each region has its own protection: read-only, read-write, execute-only
  - Each region has its own sharing: e.g., code vs. data

Segmentation

- Segment is a contiguous region of *virtual* memory
  - What if we store it in contiguous *physical* memory?
- Translation: per-process segment table
  - Each segment has: start, length, access permission
  - Instruction encoding: fewer bits of address
- Processes can share segments
  - Same start, length, same/different access permissions

Segmentation



Segment Table

Base	Bound	Access
		Read
		R/W
		R/W
		R/W

2 bit segment #
12 bit offset

	Segment start	**length**
code	0x4000	0x700
data	0	0x500
heap	-	-
stack	0x2000	0x1000

Virtual Memory

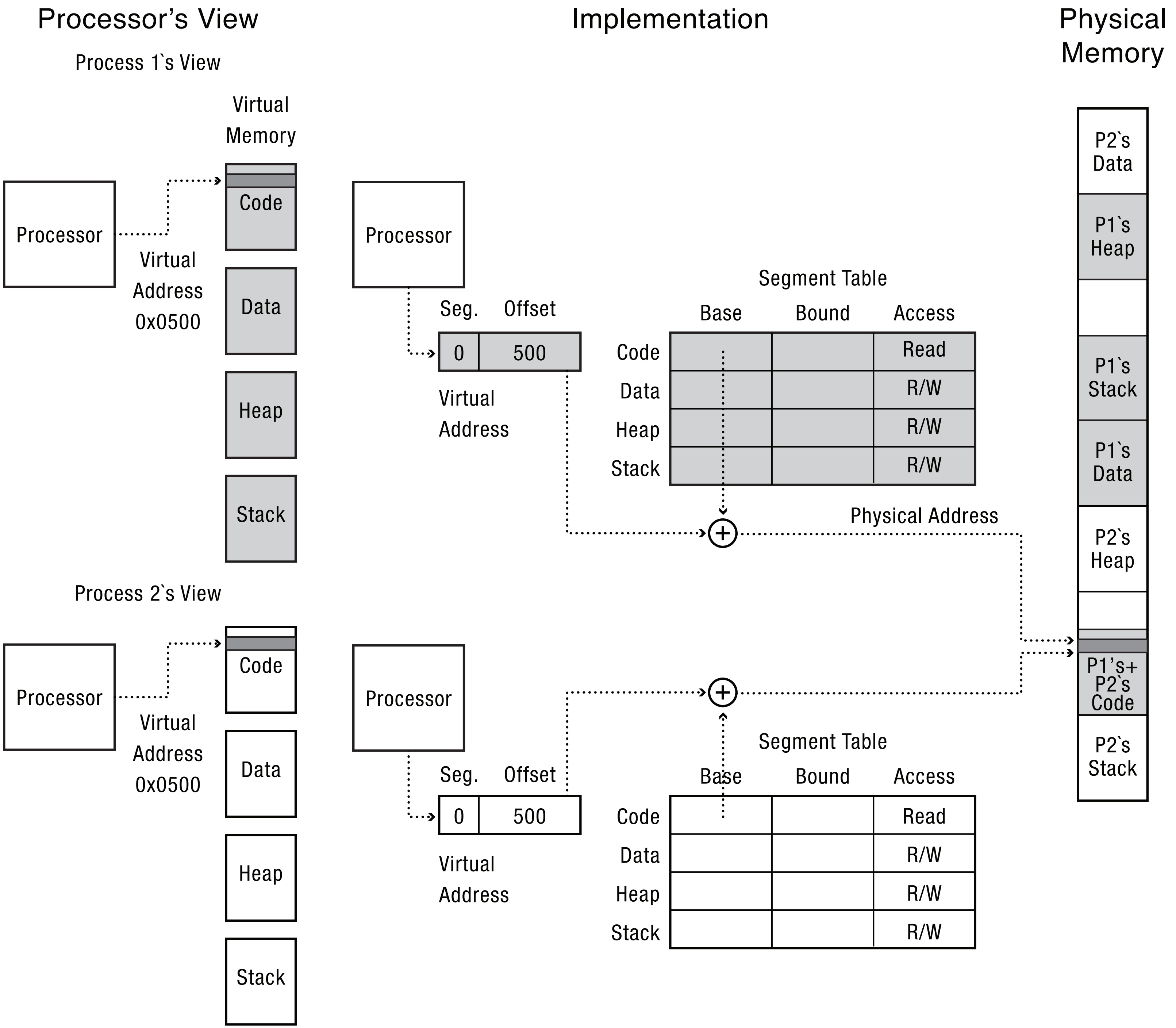
main: 240	store #1108, r2
244	store pc+8, r31
248	jump 360
24c	
…	
strlen: 360	loadbyte (r2), r3
…	…
420	jump (r31)
…	
x: 1108	a b c \0
…	

Physical Memory

x: 108	a b c \0
…	
main: 4240	store #1108, r2
4244	store pc+8, r31
4248	jump 360
424c	
…	…
strlen: 4360	loadbyte (r2),r3
…	
4420	jump (r31)
…	

UNIX fork and Copy on Write

- UNIX fork
  - Makes a complete copy of a process
- Segments allow a more efficient implementation
  - Copy segment table into child
  - Mark parent and child segments read-only
  - Start child process; return to parent
  - If child or parent writes to a segment (ex: stack)
    - trap into kernel
    - make a copy of the segment and resume



Segment Table

	Base	Bound	Access
Code			Read
Data			R/W
Heap			R/W
Stack			R/W

Segment Table

	Base	Bound	Access
Code			Read
Data			R/W
Heap			R/W
Stack			R/W

Question

- How much physical memory is needed for the stack or heap?

Expand Stack on Reference

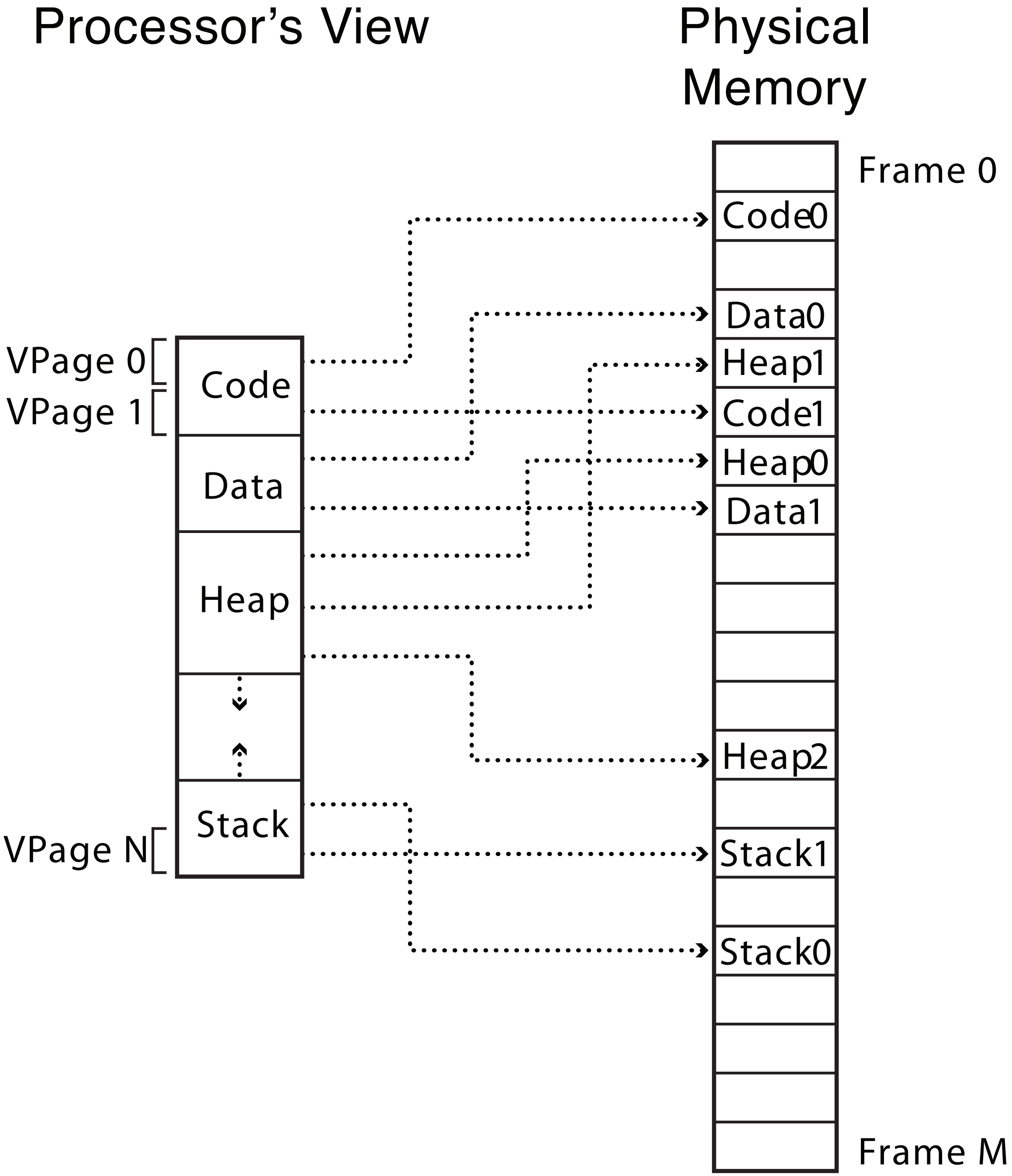
- When program references memory beyond end of stack
  - Segmentation fault into OS kernel
  - Kernel allocates some additional memory
    - How much?
  - Zeros the memory
    - avoid accidentally leaking information!
  - Modify segment table
  - Resume process

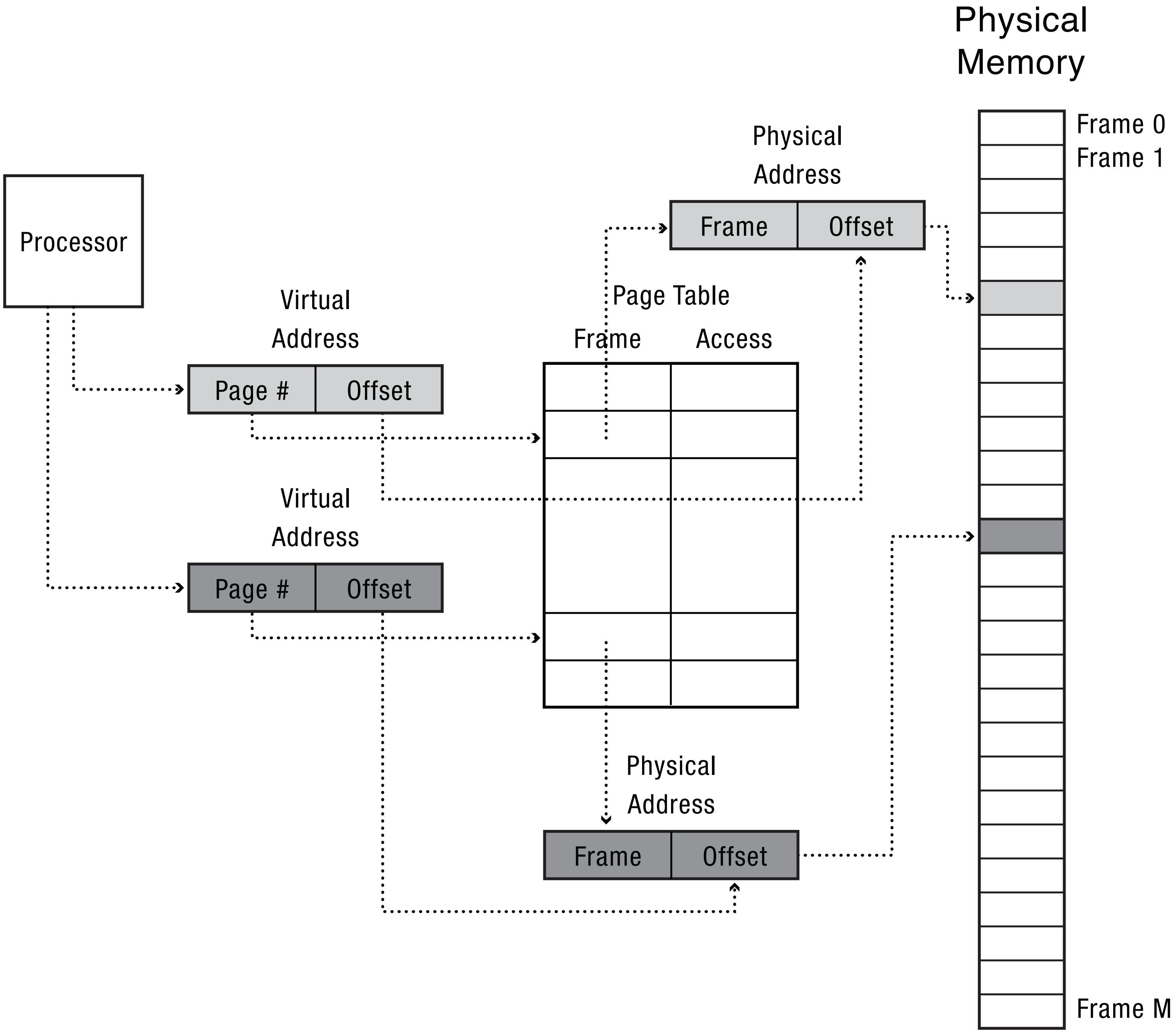
Segmentation

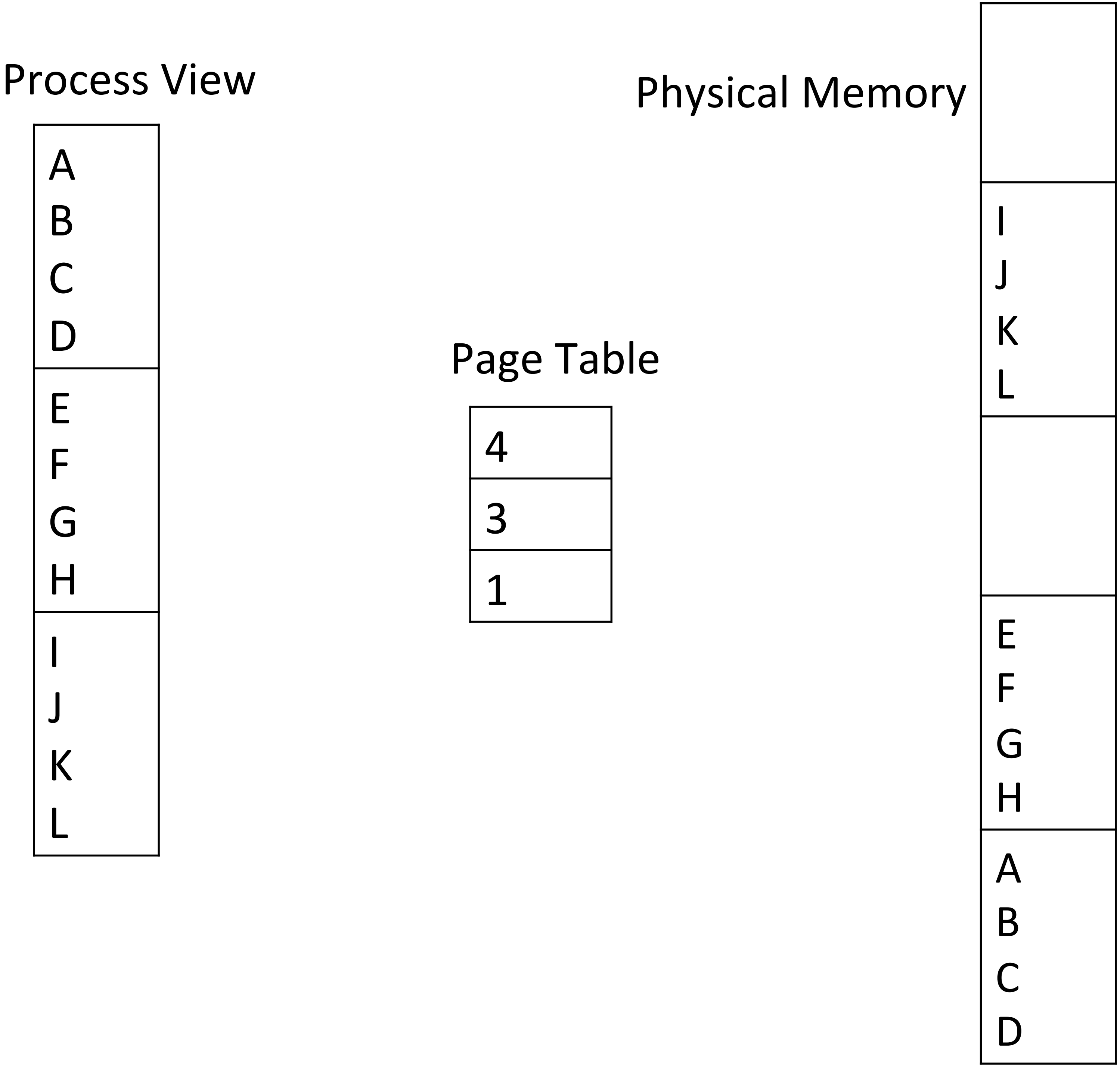
- Pros?
  - Can share code/data segments between processes
  - Can protect code segment from being overwritten
  - Can transparently grow stack/heap as needed
  - Can detect if need to copy-on-write
- Cons? Complex memory management
  - Need to find chunk of a particular size
  - May need to rearrange memory to make room for new segment or growing segment
  - External fragmentation: wasted space between chunks

Paged Translation

- Manage memory in fixed size units, or pages
- Finding a free page is easy
  - Bitmap allocation: 0011111100000001100
  - Each bit represents one physical page frame
- Each process has its own page table
  - Stored in physical memory
  - Hardware registers
    - pointer to page table start
    - page table length


Processor’s View
Physical Memory
Frame 0
Code0
Data0
Heap1
Code1
Heap0
Data1
Heap2
Stack1
Stack0
Frame M
VPage 0
VPage 1
Code
Data
Heap
Stack
VPage N


Physical Memory
Frame 0
Frame 1
Frame M
Physical Address
Frame
Offset
Processor
Virtual Address
Page #
Offset
Page Table
Frame
Access
Virtual Address
Page #
Offset
Physical Address
Frame
Offset



Process View

A
B
C
D

E
F
G
H

I
J
K
L

Page Table

4
3
1

Physical Memory

I
J
K
L

E
F
G
H

A
B
C
D

Paging and Sharing

- Can we share memory between processes?

- Set both page tables point to same page frames
- Need *core map*
  - Array of information about each physical page frame
  - Set of processes pointing to that page frame
  - When zero, can reclaim!

Paging and Copy on Write

- UNIX fork
  - Copy page table of parent into child process
  - Mark all pages (in new and old page tables) as read-only
  - Trap into kernel on write (in child or parent)
  - Copy page
  - Mark both as writeable
  - Resume execution

Question

- Can I run a program when only some of its code is in physical memory?

Fill On Demand

- Set all page table entries to invalid
- When a page is referenced for first time, kernel trap
- Kernel brings page in from disk
- Resume execution
- Remaining pages can be transferred in the background while program is running

A Case for Sparse Address Spaces

- Might want many separate segments
  - Per-processor heaps
  - Per-thread stacks
  - Memory-mapped files
  - Dynamically linked libraries
- What if virtual address space is large?
  - 32-bits, 4KB pages => 500K page table entries
  - 64-bits, 4KB pages => 4 quadrillion table entries (!)

Multi-level Translation

- Tree of translation tables
  - Paged segmentation
  - Multi-level page tables
  - Multi-level paged segmentation
- All have pages as lowest level; why?

Fixed Size Pages at Lowest Level

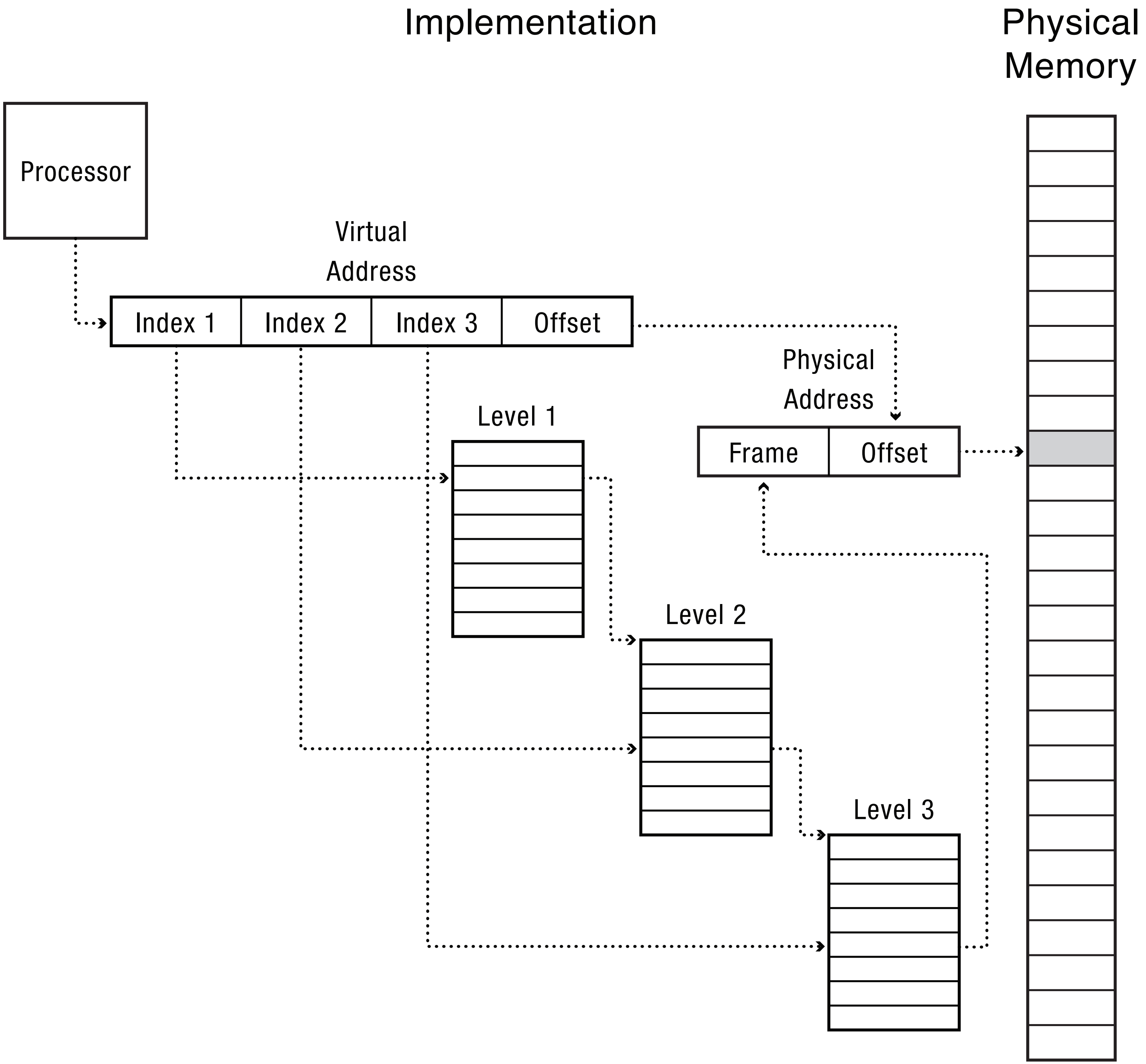
- Efficient memory allocation (vs. segments)
- Efficient for sparse addresses (vs. paging)
- Efficient disk transfers (fixed size units)
- Easier to build translation lookaside buffers
- Efficient reverse lookup (from physical -> virtual)
- Variable granularity for protection/sharing

Paged Segmentation

- Process memory is segmented
- Segment table entry:
  - Pointer to page table
  - Page table length (# of pages in segment)
  - Access permissions
- Page table entry:
  - Page frame
  - Access permissions
- Share/protection at either page or segment-level

Multilevel Paging

What if each page table points to a page table?


Implementation
Physical Memory
Processor
Virtual Address
Index 1
Index 2
Index 3
Offset
Physical Address
Frame
Offset
Level 1
Level 2
Level 3

Question

- Write pseudo-code for translating a virtual address to a physical address for a system using 3-level paging, with 8 bits of address per level

x86 Multilevel Paged Segmentation

- Global Descriptor Table (segment table)
  - Pointer to page table for each segment
  - Segment length
  - Segment access permissions
  - Context switch: change global descriptor table register (GDTR, pointer to global descriptor table)
- Multilevel page table
  - 4KB pages; each level of page table fits in one page
  - 32-bit: two level page table (per segment)
  - 64-bit: four level page table (per segment)
  - Omit sub-tree if no valid addresses

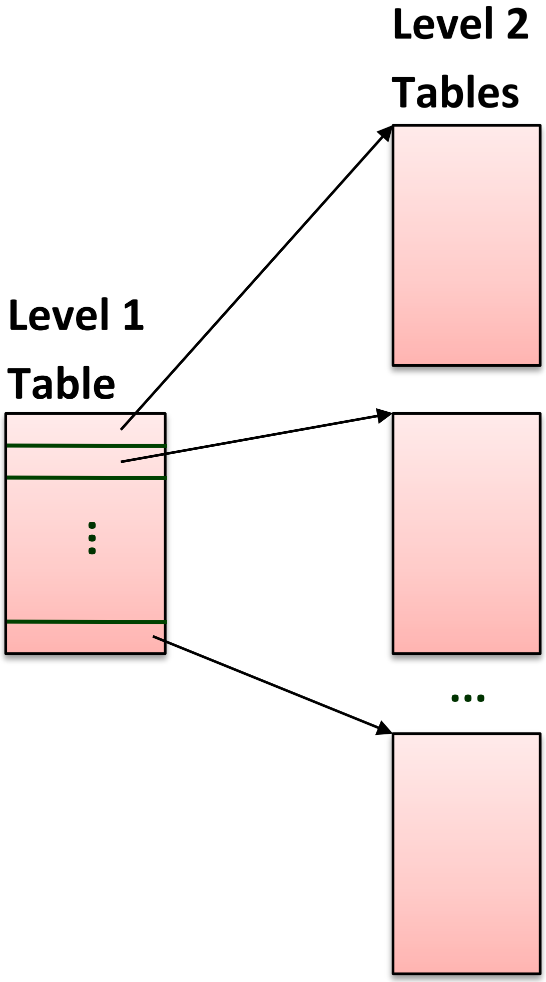
Multilevel Translation

- Pros:
  - Allocate/fill only page table entries that are in use
  - Simple memory allocation
  - Share at segment or page level
- Cons:
  - Space overhead: one pointer per virtual page
  - Multiple lookups per memory reference

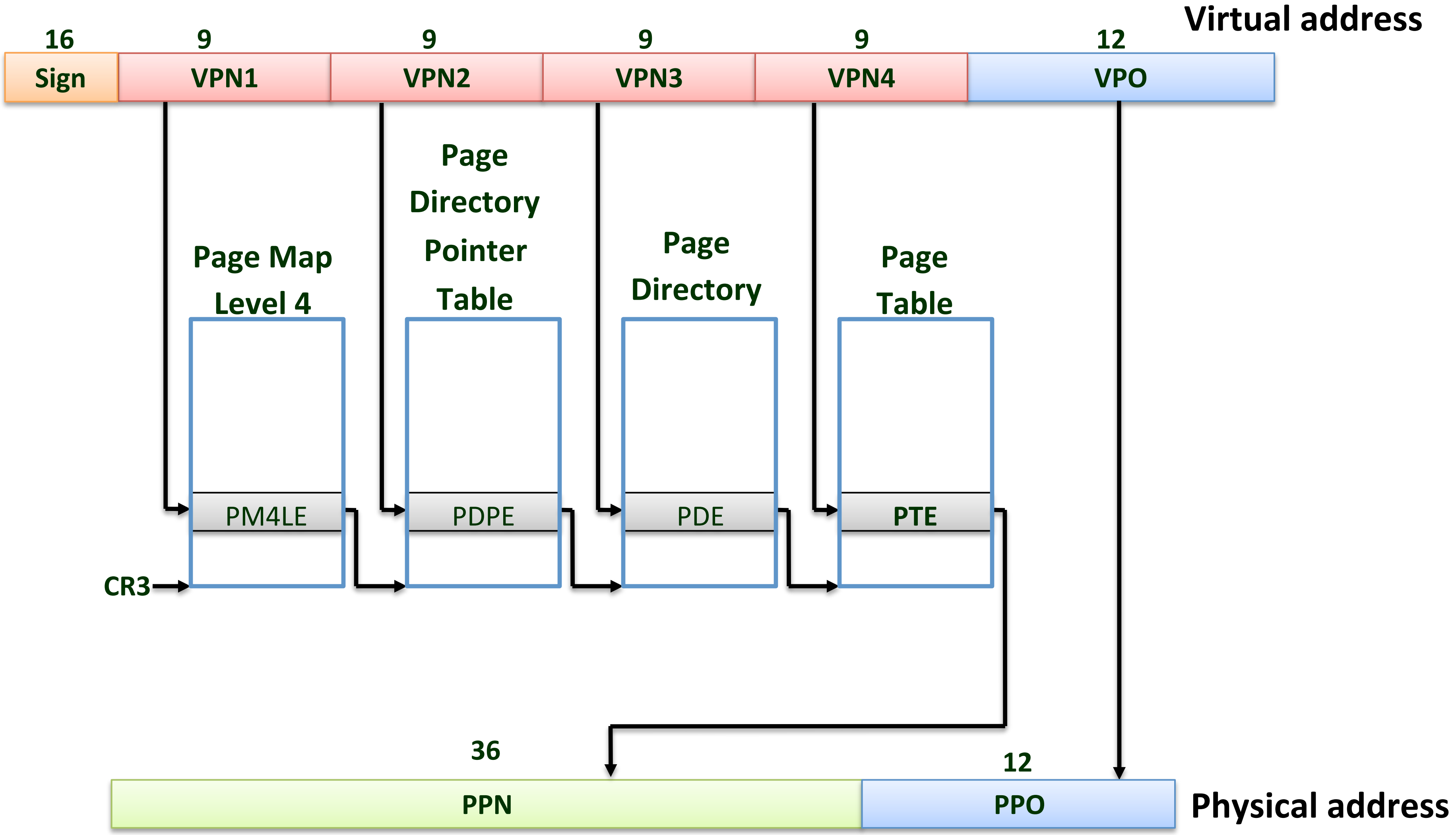
Multi-level or hierarchical page tables

- Example: 2-level page table
  - Level 1 table: each PTE points to a page table
  - Level 2 table: each PTE points to a page (paged in and out like other data)

- Level 1 table stays in memory
- Level 2 tables might be paged in and out



x86-64 Paging



Features

- Saves memory for mostly-empty address spaces
  - But more memory references required for lookup
- Natural support for superpages
  - “Early termination” of table walk
- Frequently implemented in hardware (or microcode…)
  - x86, ARM (and others)
- Also the generic page table “abstraction” in Linux and Windows

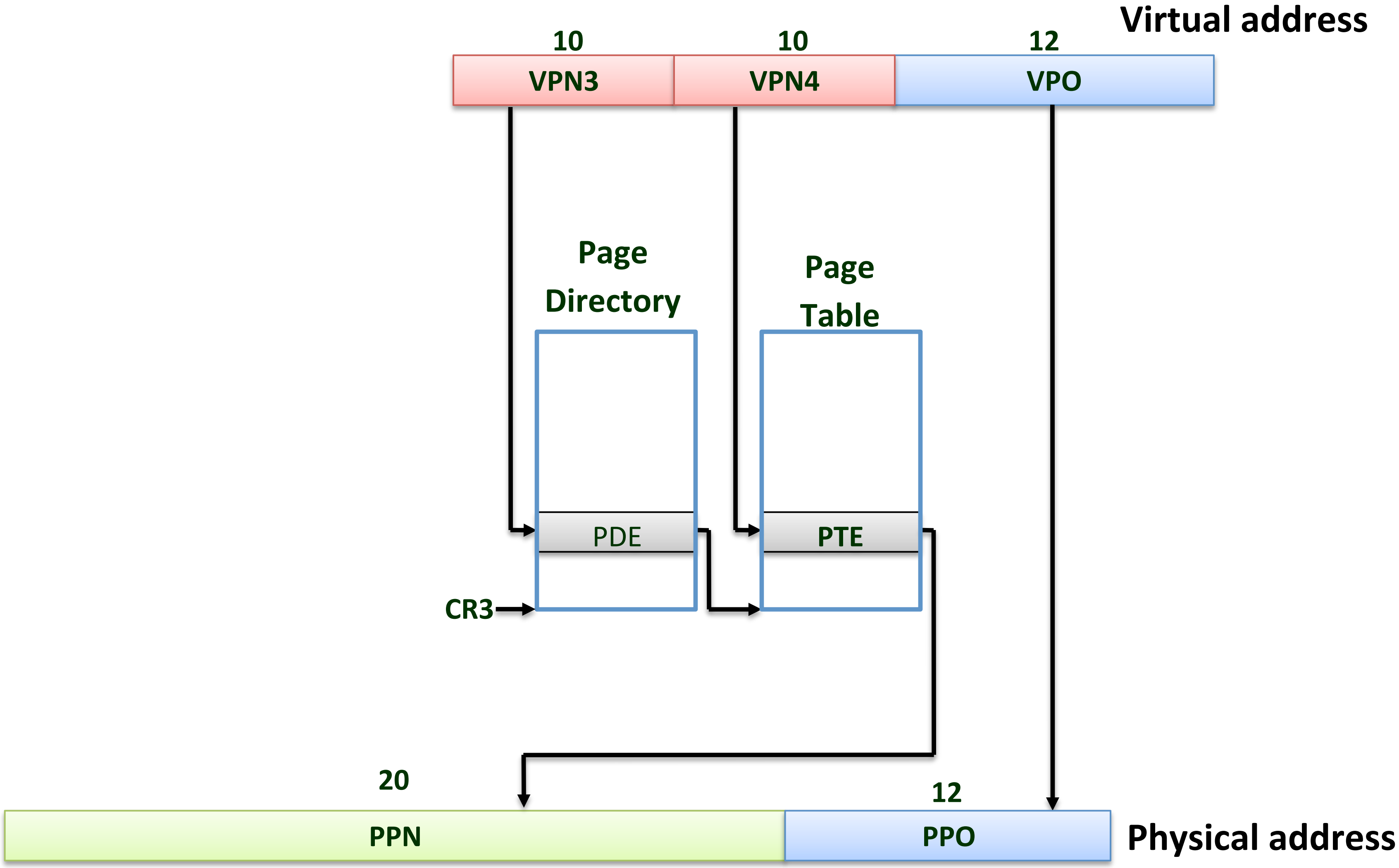
Problems with hierarchical page tables

- Depth scales with size of virtual address space
  - 5–6 levels deep for a full 64-bit address space
  - X86-64 (48-bit virtual address) needs 4 levels
- A sparse address-space layout requires lots of memory for (mostly empty) tables
  - Not a big problem for the traditional UNIX memory model

x86-32 page tables

- MMU supports:
  - Large pages (4 MBytes)
  - Small pages (4 KBytes)
- 2-level hierarchical page table:
  - Page directory
  - Page table

x86-32 Paging



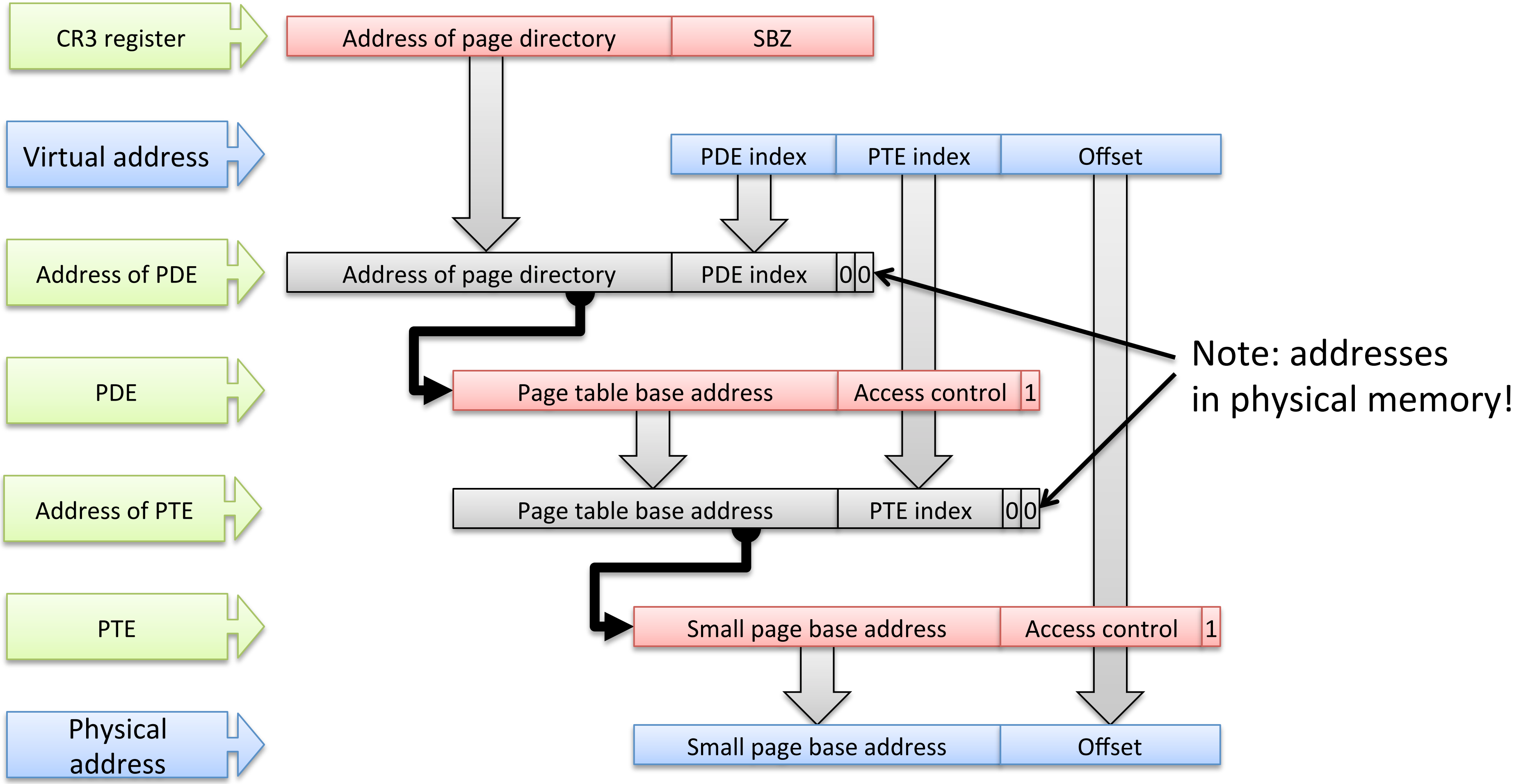
Page directory entries

Empty	Ignored											0
4MB page	Bits 31:22 of address of 4MB page frame	0	Ign	G	1	D	A	PCD	PWT	U/S	R/W	1
Page table	Bits 31:12 of address of page table		Ign		0	Ign	A	PCD	PWT	U/S	R/W	1

Page table entries

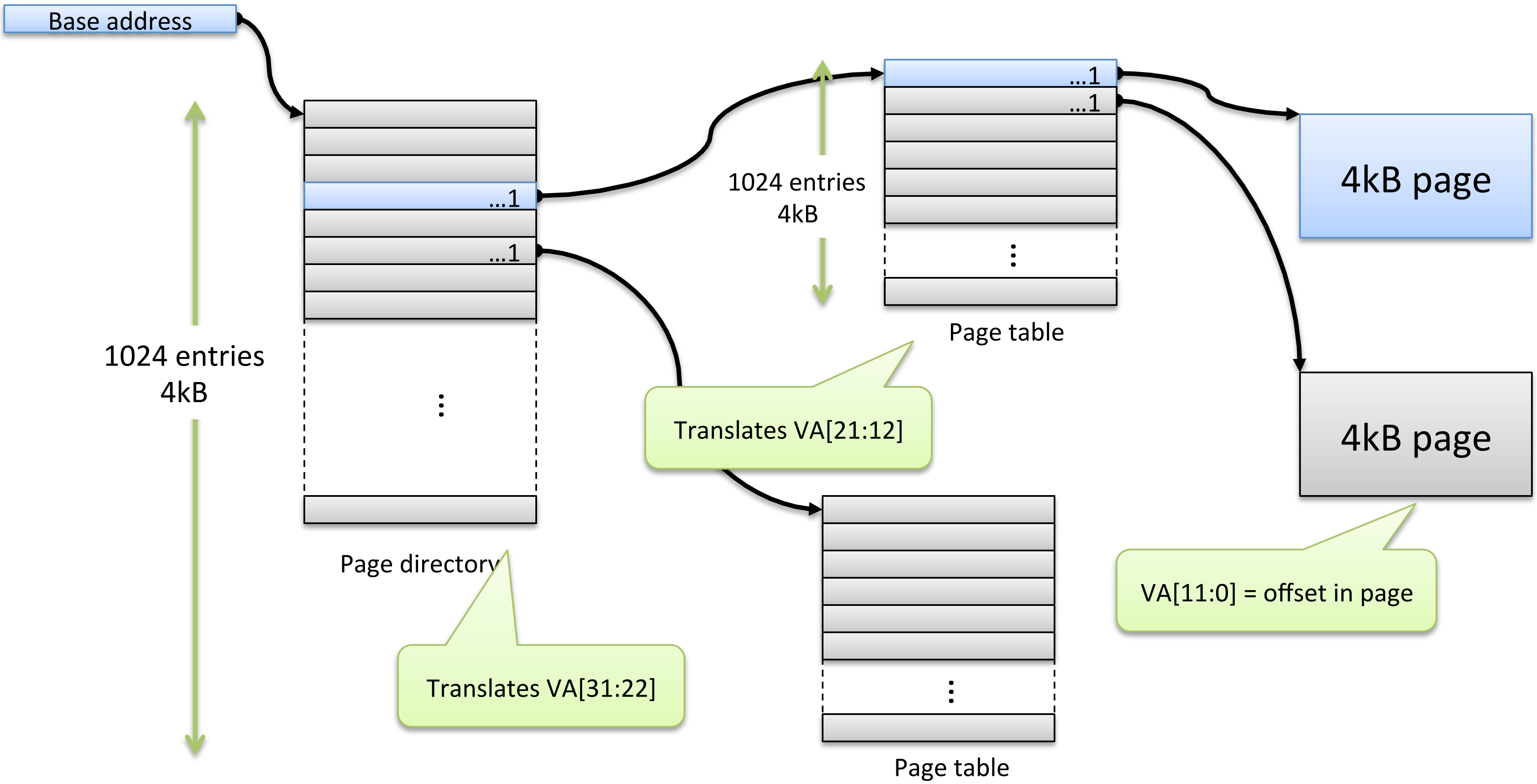
Empty	Ignored										0
4KB page	Bits 31:12 of address of page frame	Ign	G	0	D	A	PCD	PWT	U/S	R/W	1

Small page translation



Note: addresses in physical memory!

Page table for small pages



Page Translation in the OS

- OS's need to keep their own data structures
  - List of memory objects (segments)
  - Virtual page -> physical page frame
  - Physical page frame -> set of virtual pages
- An option: Inverted page table
  - Hash from virtual page -> physical page
  - Space proportional to # of physical pages
- Why not just reuse the hardware page tables?

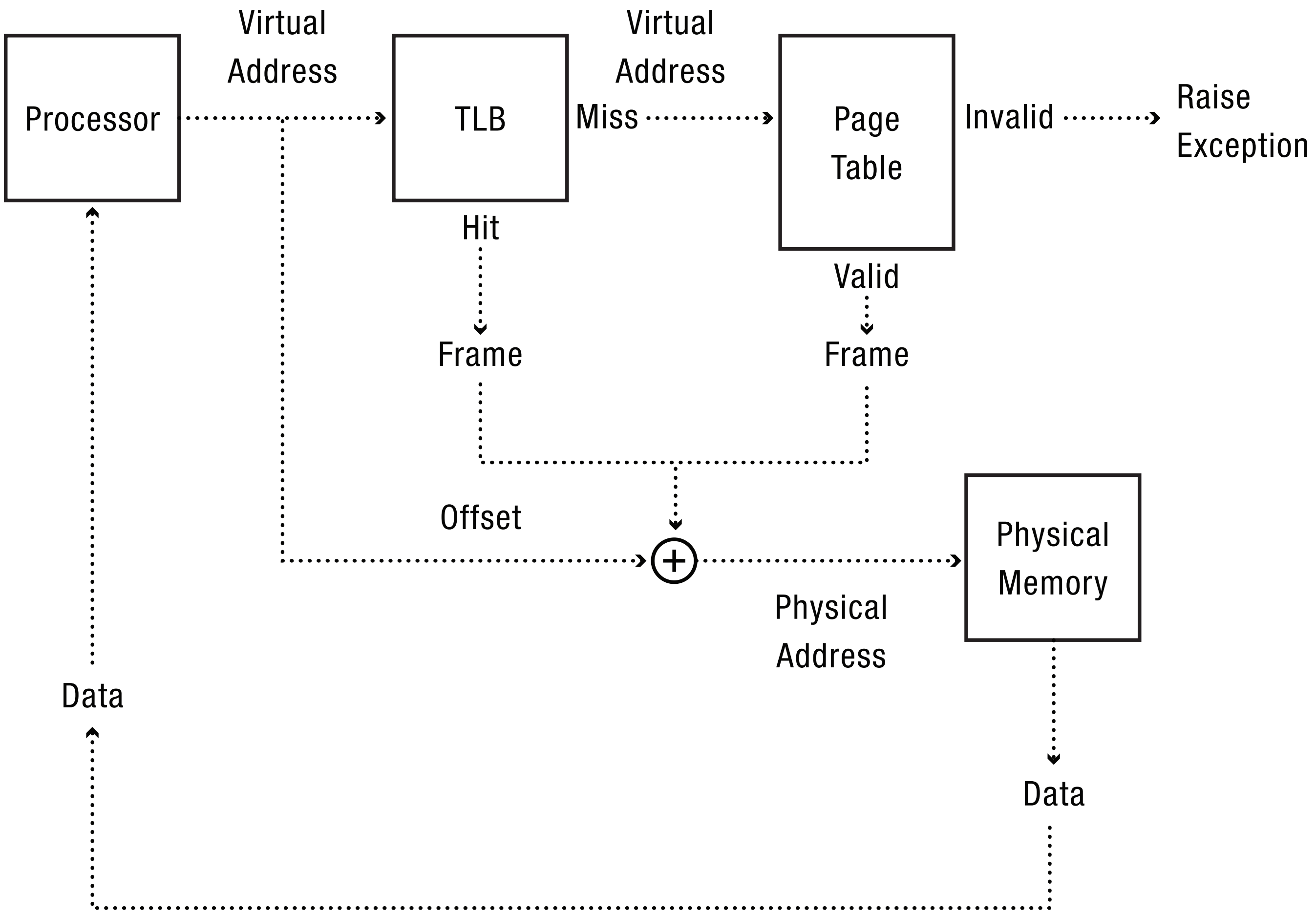
Efficient Address Translation

- Translation lookaside buffer (TLB)
  - Cache of recent virtual page -> physical page translations
  - If cache hit, use translation
  - If cache miss, walk multi-level page table
- Cost of translation =

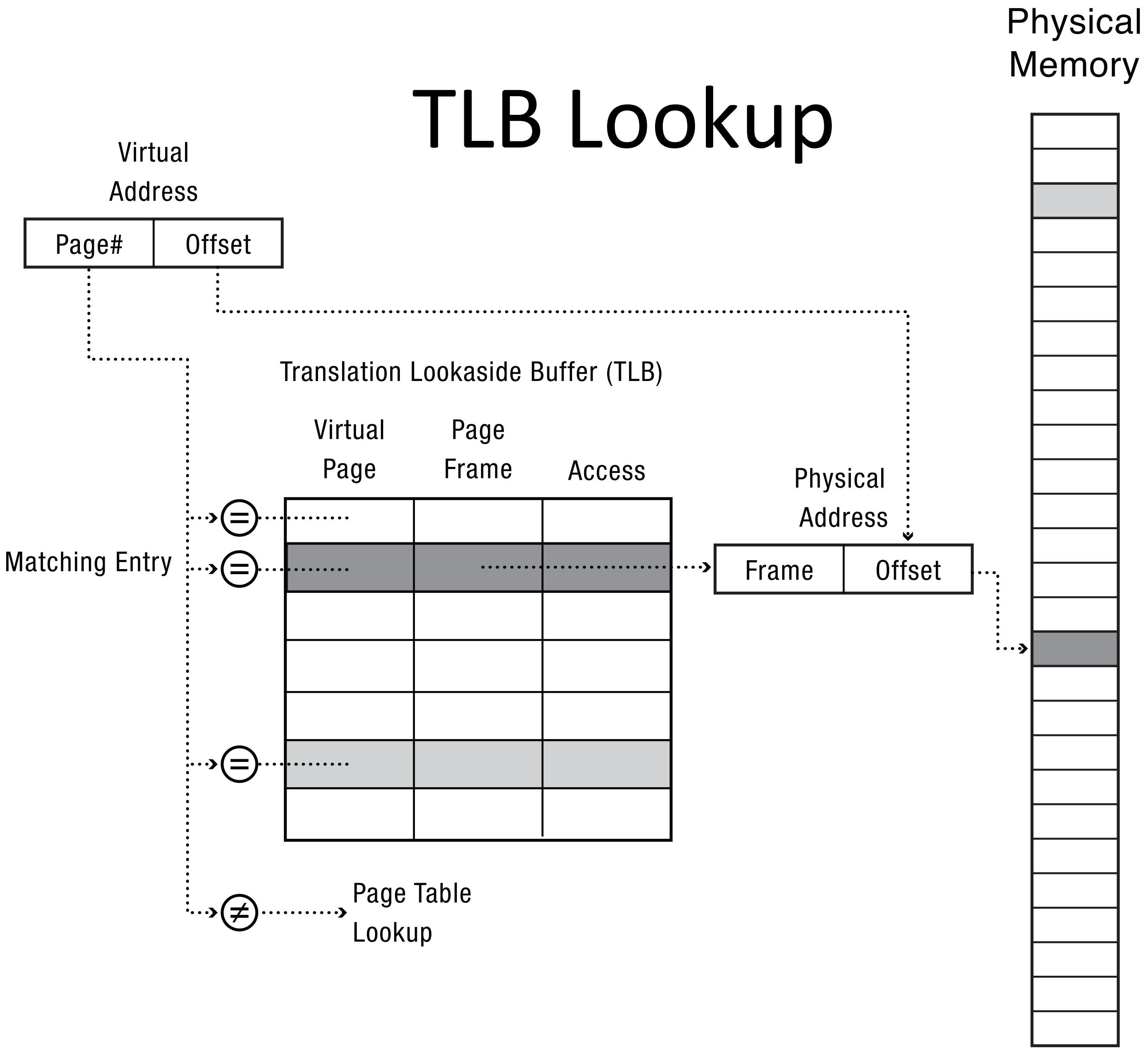
  Cost of TLB lookup +

  Prob(TLB miss) * cost of page table lookup

TLB and Page Table Translation



TLB Lookup

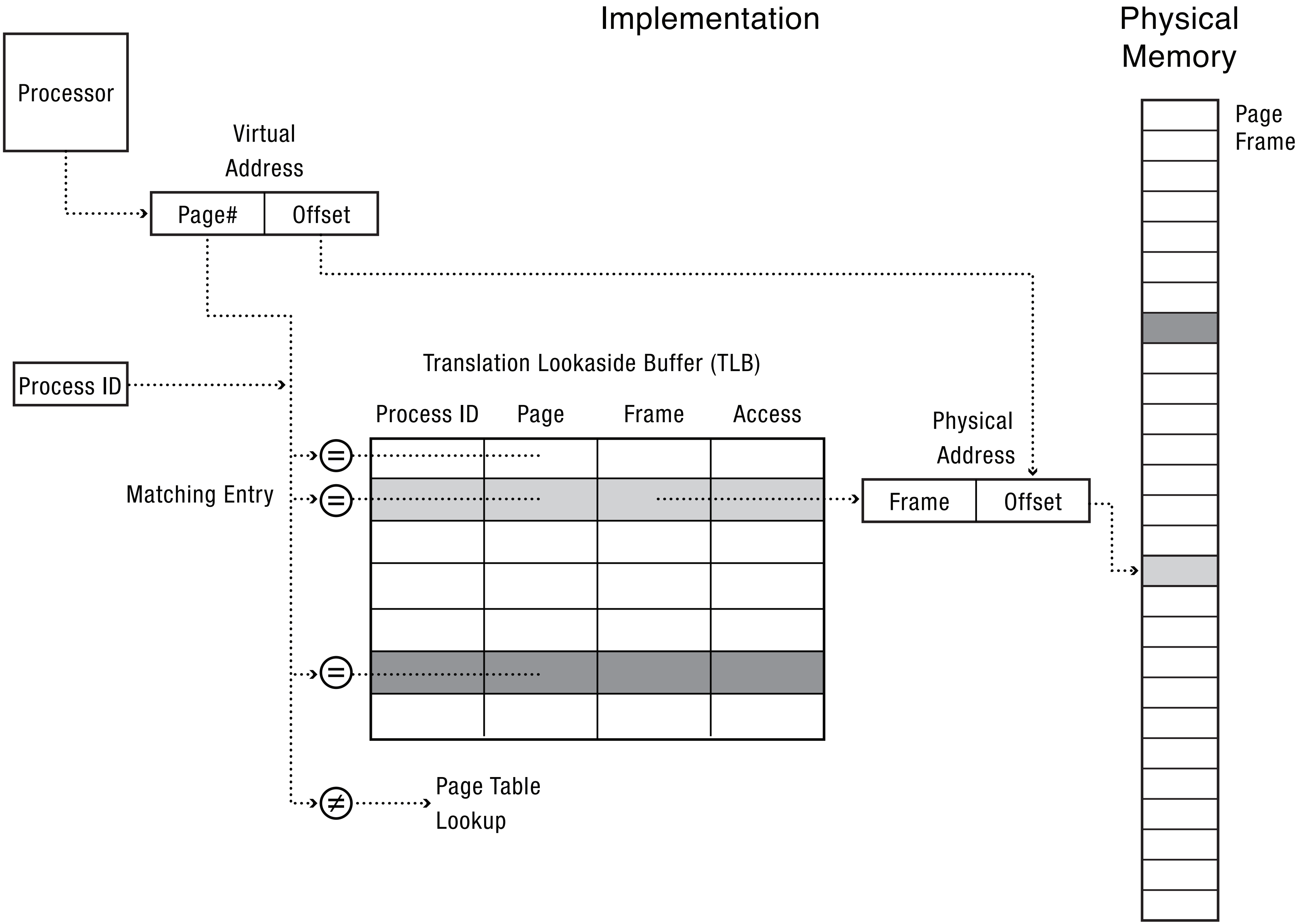

Physical
Memory
Virtual
Address
Page#
Offset
Translation Lookaside Buffer (TLB)
Virtual
Page
Page
Frame
Access
Physical
Address
Frame
Offset
Matching Entry
=
=
=
≠
Page Table
Lookup

Question

- What happens on a context switch?
  - Reuse TLB?
  - Discard TLB?

- Solution: Tagged TLB
  - Each TLB entry has process ID
  - TLB hit only if process ID matches current process

Implementation



Physical Memory

Processor

Virtual Address

Page#

Offset

Page Frame

Process ID

Translation Lookaside Buffer (TLB)

Process ID

Page

Frame

Access

Physical Address

Matching Entry

Frame

Offset

Page Table Lookup

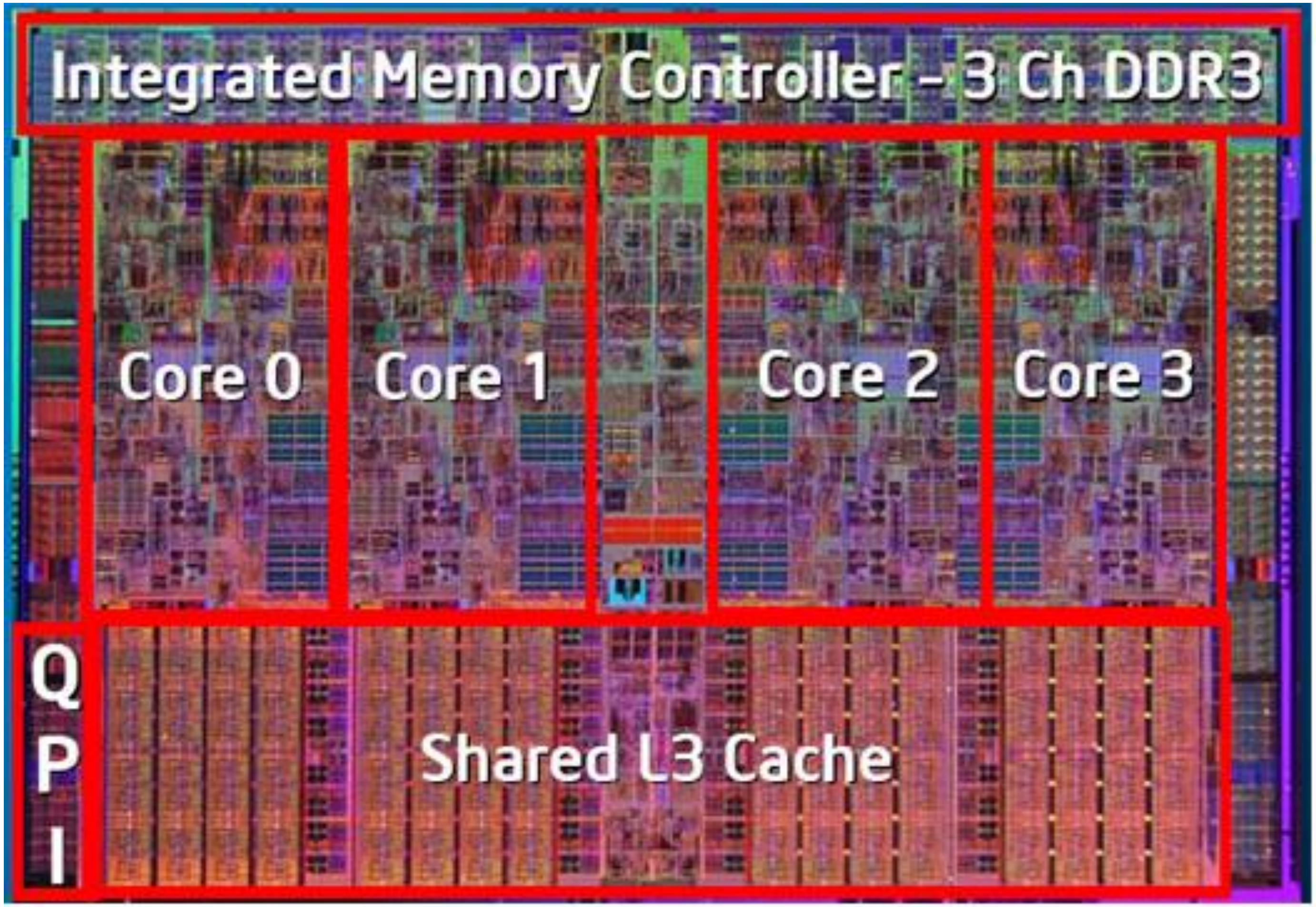
MIPS Software Loaded TLB

- Software defined translation tables
  - – If translation is in TLB, ok
  - – If translation is not in TLB, trap to kernel
  - – Kernel computes translation and loads TLB
  - – Kernel can use whatever data structures it wants
- Pros/cons?

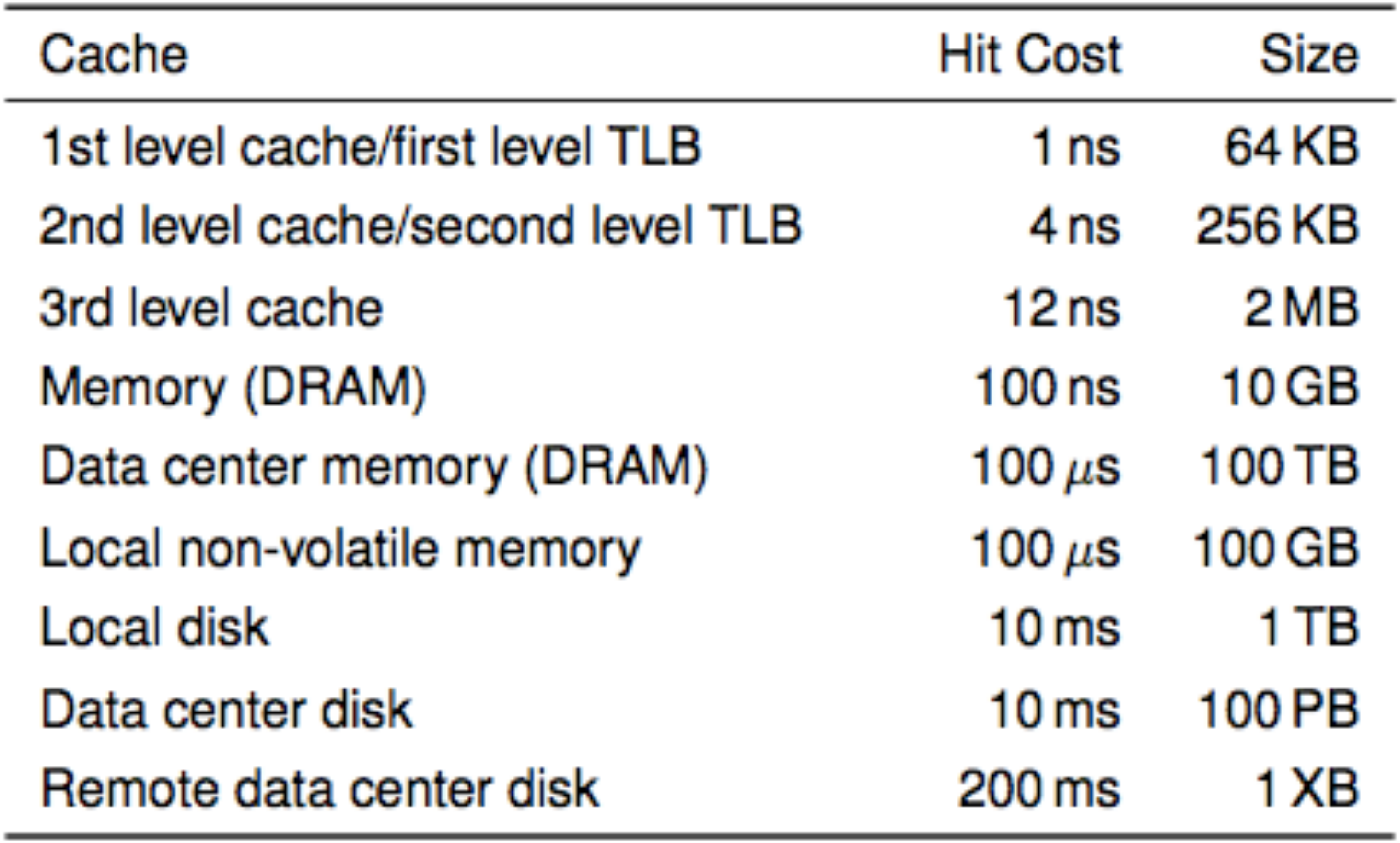
Question

- What is the cost of a TLB miss on a modern processor?
  - Cost of multi-level page table walk
  - MIPS: plus cost of trap handler entry/exit

Intel i7


Integrated Memory Controller – 3 Ch DDR3
Core 0
Core 1
Core 2
Core 3
QPI
Shared L3 Cache

Memory Hierarchy



Cache	Hit Cost	Size
1st level cache/first level TLB	1 ns	64 KB
2nd level cache/second level TLB	4 ns	256 KB
3rd level cache	12 ns	2 MB
Memory (DRAM)	100 ns	10 GB
Data center memory (DRAM)	100 μs	100 TB
Local non-volatile memory	100 μs	100 GB
Local disk	10 ms	1 TB
Data center disk	10 ms	100 PB
Remote data center disk	200 ms	1 XB

i7 has 8MB as shared 3rd level cache; 2nd level cache is per-core

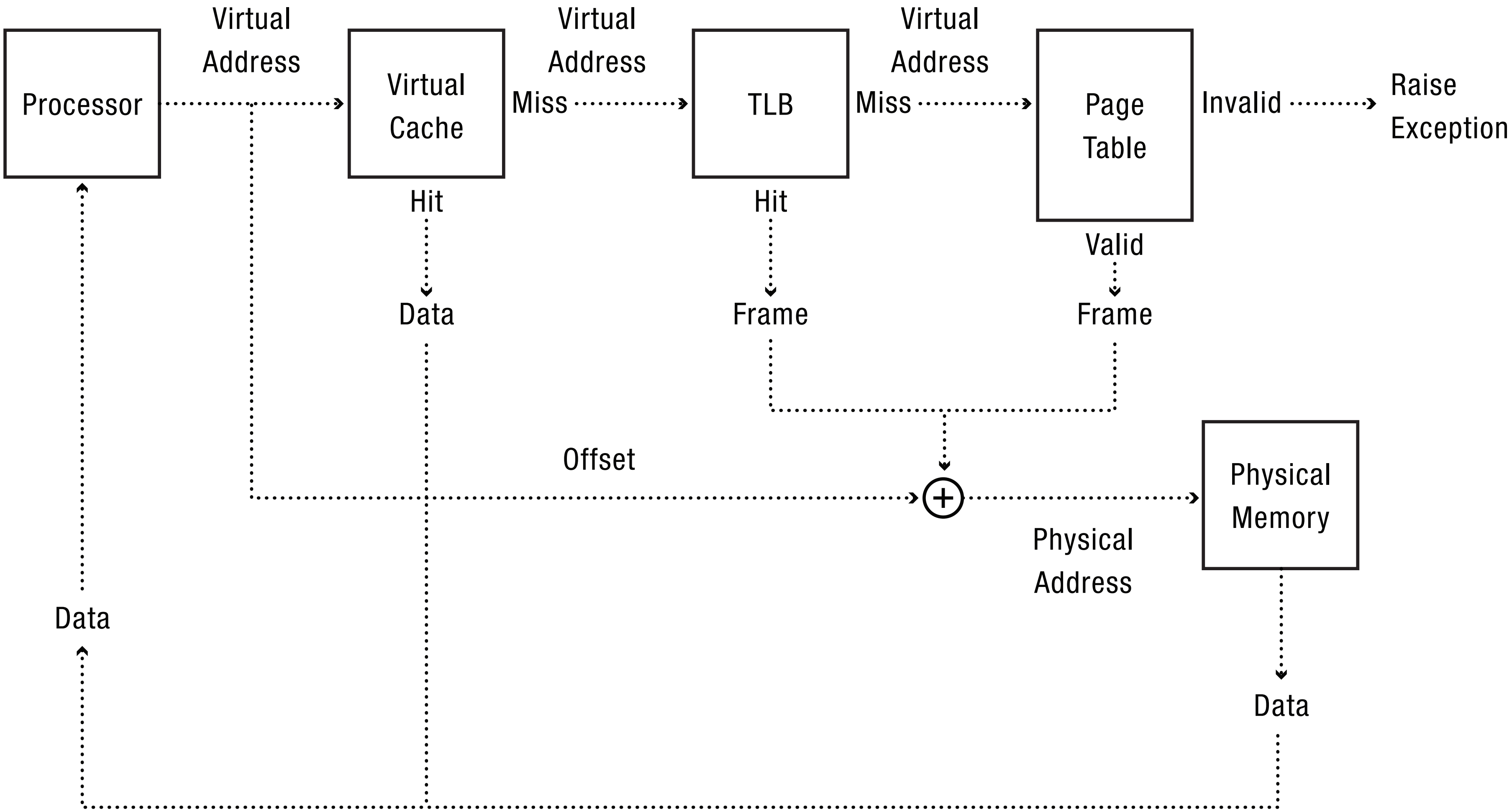
Question

- What is the cost of a first level TLB miss?
  - Second level TLB lookup
- What is the cost of a second level TLB miss?
  - x86: 2-4 level page table walk
- How expensive is a 4-level page table walk on a modern processor?

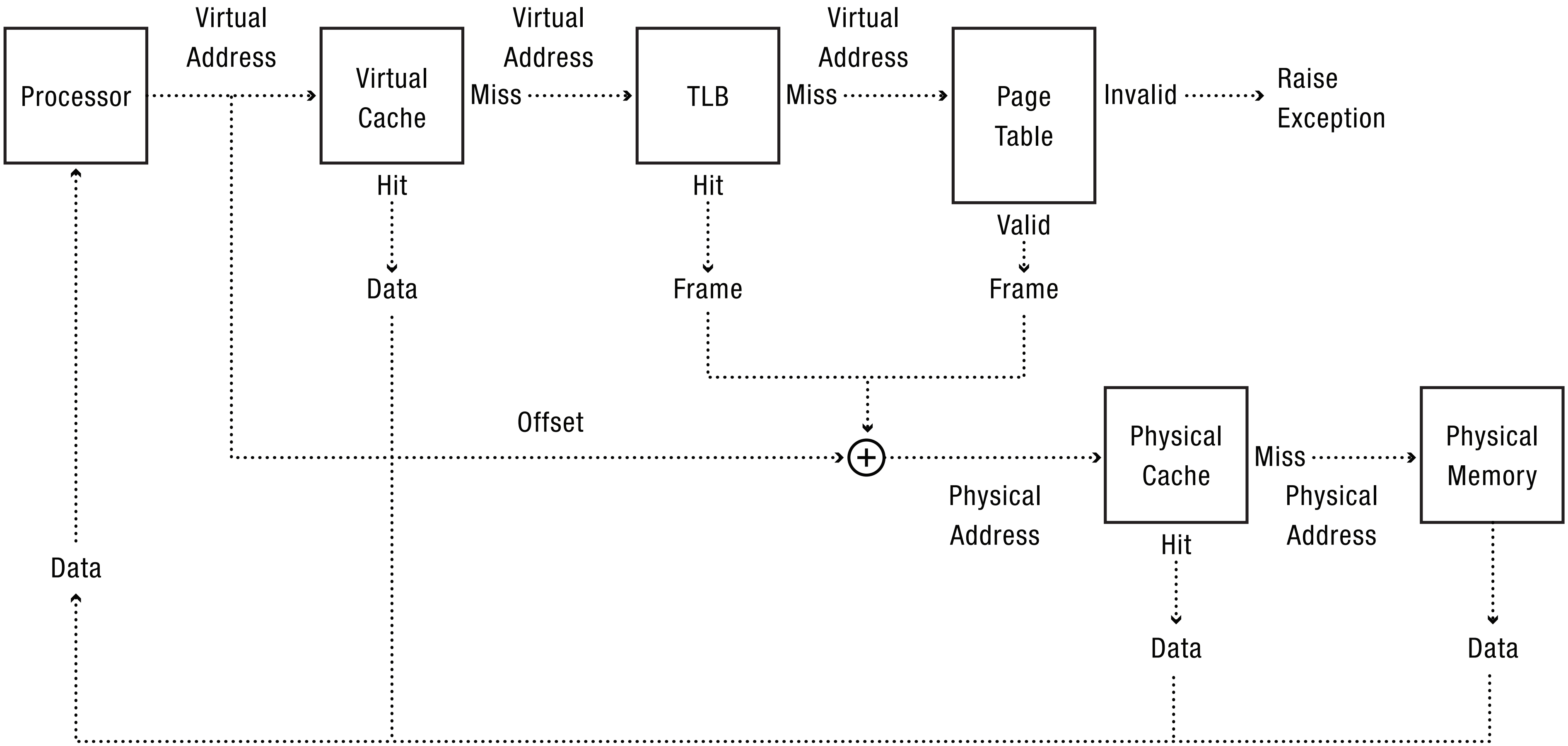
Virtually Addressed vs. Physically Addressed Caches

- Too slow to first lookup TLB to find physical address, then look up address in the cache
- Instead, first level cache is virtually addressed
- In parallel, lookup TLB to generate physical address in case of a cache miss

Virtually Addressed Caches



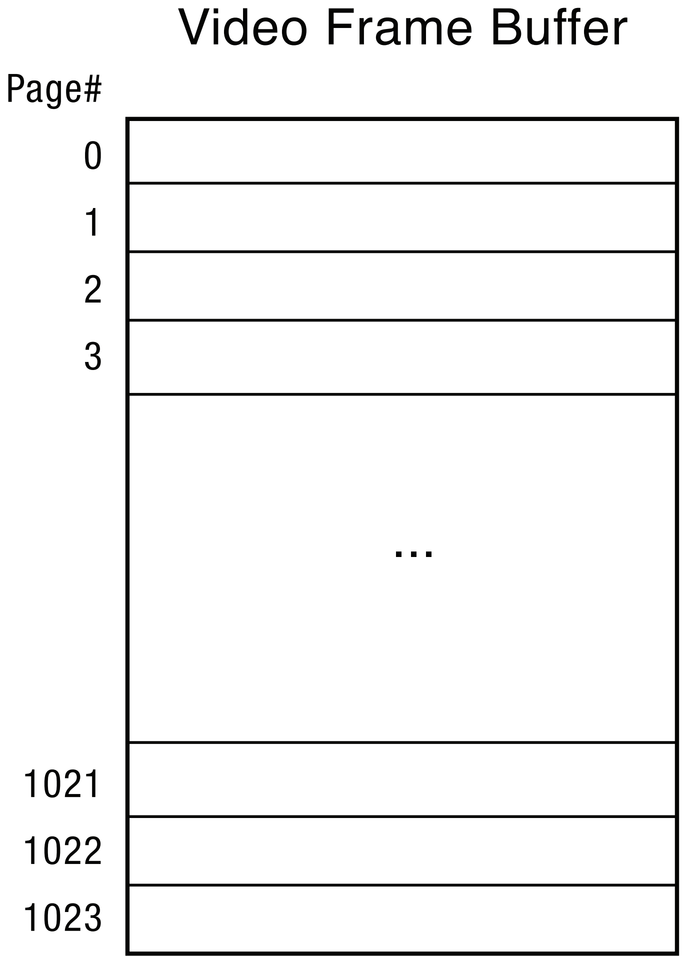
Physically Addressed Cache



When Do TLBs Work/Not Work?

Video Frame Buffer: 32 bits x 1K x 1K = 4MB

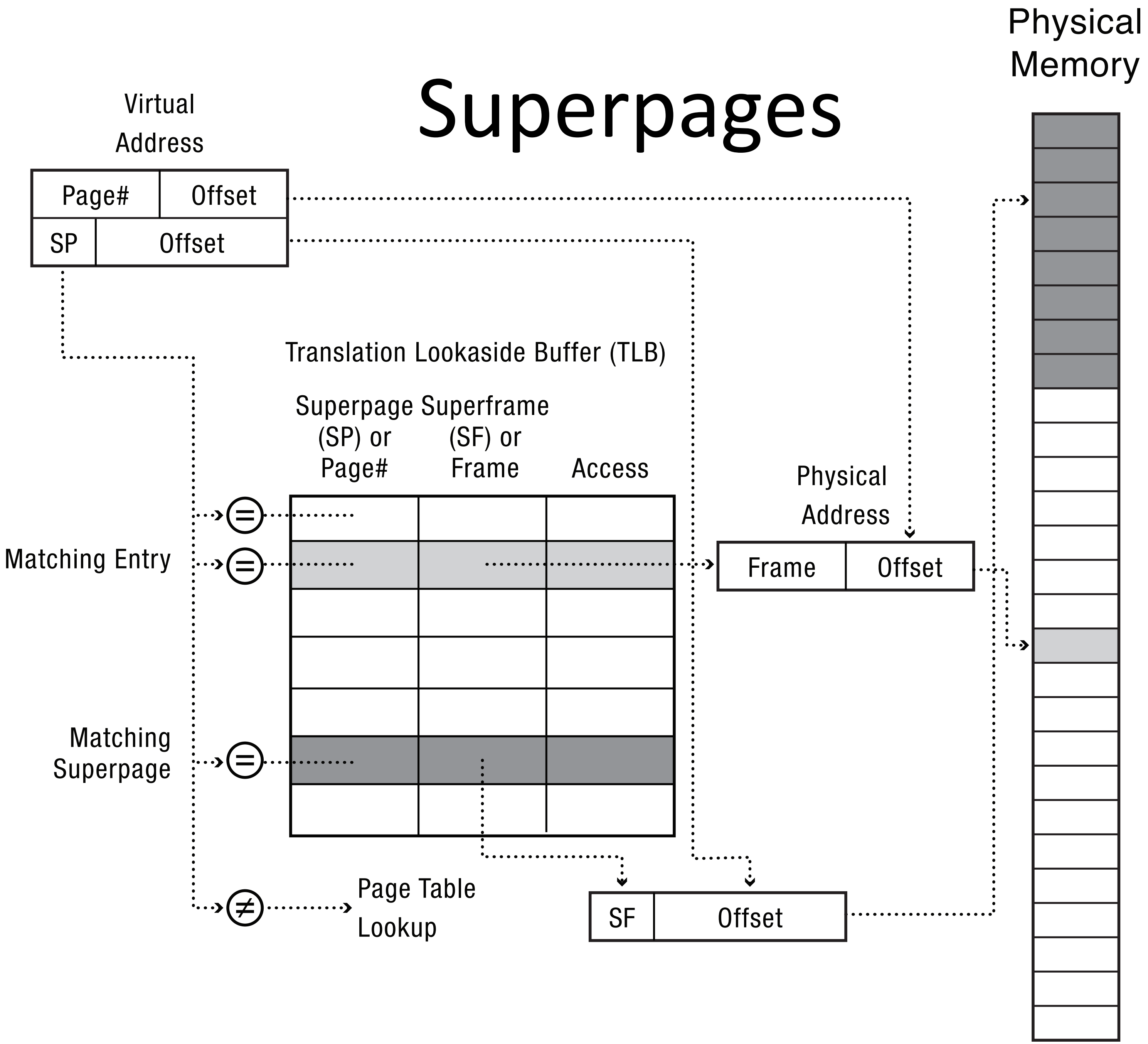
1st level TLB = 100 entries



Superpages

- On many systems, TLB entry can be
  - A page
  - A superpage: a set of contiguous, aligned pages
- x86: superpage is set of pages in one page table
  - One page: 4KB
  - One page table: 2MB
  - One page table of page tables: 1GB
  - One page table of page tables of page tables: 0.5TB

Superpages


Virtual Address
Page#
Offset
SP
Offset
Translation Lookaside Buffer (TLB)
Superpage (SP) or Page#
Superframe (SF) or Frame
Access
Matching Entry
Matching Superpage
Page Table Lookup
Physical Address
Frame
Offset
SF
Offset
Physical Memory

When Do TLBs Work/Not Work, part 2

- What happens when the OS changes the permissions on a page?
  - For demand paging, copy on write, zero on reference, …
- TLB may contain old translation
  - OS asks hardware to purge TLB entry
  - Possibly lazy or batched
- On a multicore: TLB shootdown
  - OS asks each CPU to purge TLB entry
  - Possibly lazy or batched

TLB Shootdown

		Process ID	VirtualPage	PageFrame	Access
Processor 1 TLB	=	0	0x0053	0x0003	R/W
	=	1	0x4OFF	0x0012	R/W
Processor 2 TLB	=	0	0x0053	0x0003	R/W
	=	0	0x0001	0x0005	Read
Processor 3 TLB	=	1	0x4OFF	0x0012	R/W
	=	0	0x0001	0x0005	Read

Virtual Cache Shootdown

- Do we also need to shoot down the contents of the virtual cache on every CPU?
- Lazy shootdown of the virtual cache contents:
  - Lookup virtually addressed cache and TLB in *parallel*
  - Use the TLB to verify virtual address is still valid!
  - Evict entry from cache if not

Virtual Cache Aliases

- Alias: two (or more) virtual cache entries that refer to the same physical memory
  - A consequence of a tagged virtually addressed cache!
  - A write to one copy needs to update all copies
- Solution:
  - Virtual cache keeps both virtual and physical address for each entry
  - Lookup virtually addressed cache and TLB in *parallel*
  - Check if physical address from TLB matches any other entries, and update/invalidate those copies

x86 caches

- 64 byte line size
- Physically indexed
- Physically tagged
- Write buffer

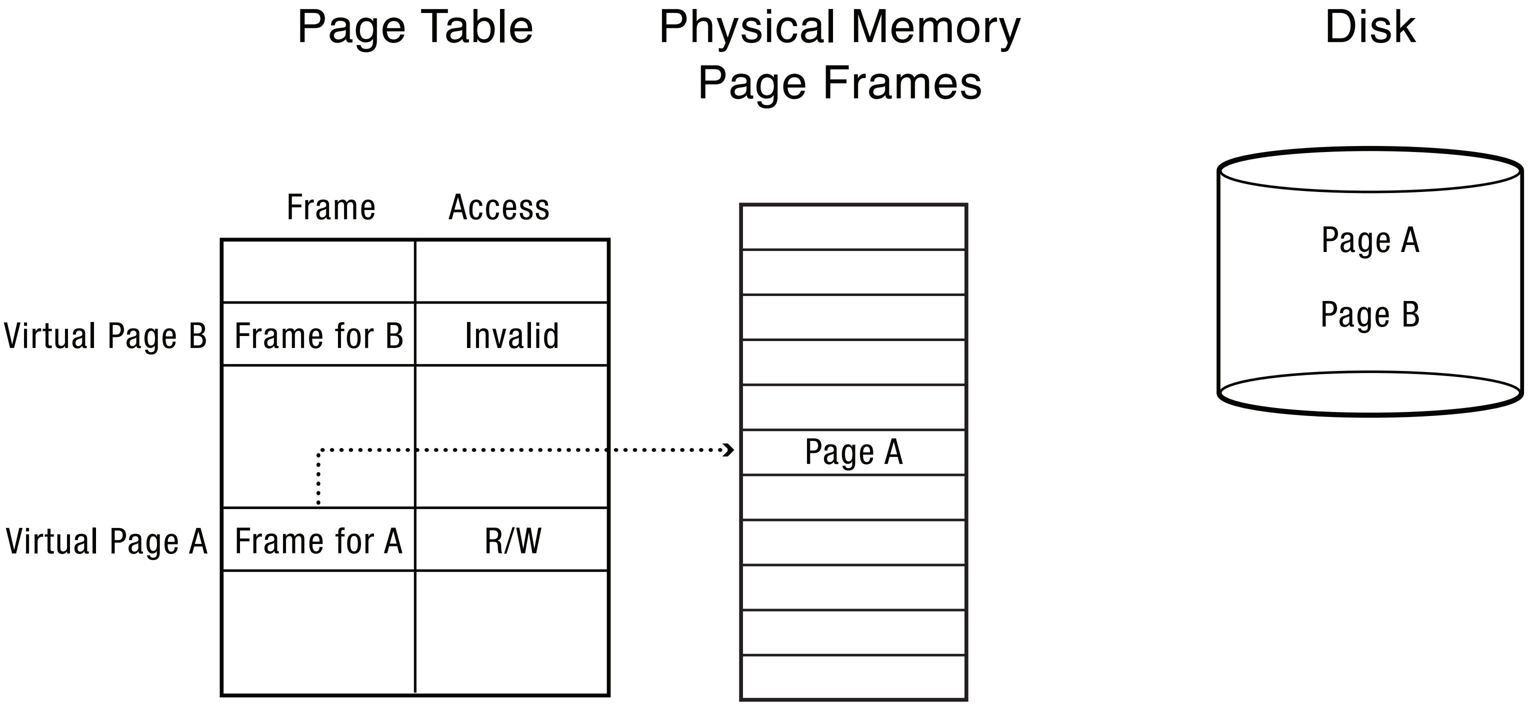
Hardware address translation is a power tool

- Kernel trap on read/write to selected addresses
  - Copy on write
  - Fill on reference
  - Zero on use
  - Demand paged virtual memory
  - Memory mapped files
  - Modified bit emulation
  - Use bit emulation

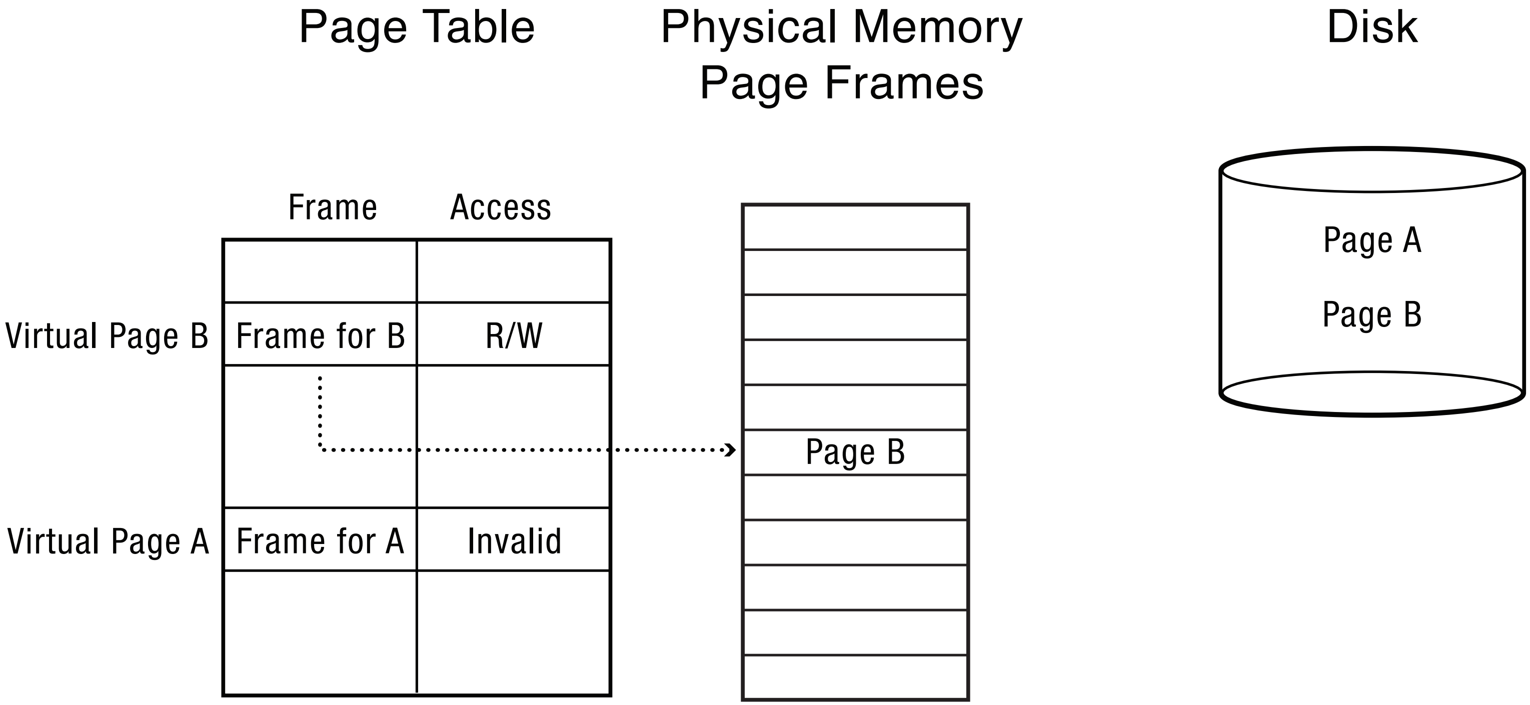
Demand Paging

- Illusion of (nearly) infinite memory, available to every process
- Multiplex virtual pages onto a limited amount of physical page frames
- Pages can be either
  - resident (in physical memory, valid page table entry)
  - non-resident (on disk, invalid page table entry)
- First reference to non-resident page, copy into memory, replacing some resident page
  - From the same process, or a different process

Demand Paging (Before)



Demand Paging (After)



Demand Paging Questions

- How does the kernel provide the illusion that all pages are resident?
- Where are non-resident pages stored on disk?
- How do we find a free page frame?
- Which pages have been modified (must be written back to disk) or actively used (shouldn't be evicted)?
- Are modified/use bits virtual or physical?
- What policy should we use for choosing which page to evict?

Demand Paging on MIPS

1. TLB miss
2. Trap to kernel
3. Page table walk
4. Find page is invalid
5. Locate page on disk
6. Allocate page frame
   - Evict page if needed
7. Initiate disk block read into page frame
8. Disk interrupt when DMA complete
9. Mark page as valid
10. Load TLB entry
11. Resume process at faulting instruction
12. Execute instruction

Demand Paging

1. TLB miss
2. Page table walk
3. Page fault (page invalid in page table)
4. Trap to kernel
5. Locate page on disk
6. Allocate page frame
   - Evict page if needed
7. Initiate disk block read into page frame
8. Disk interrupt when DMA complete
9. Mark page as valid
10. Resume process at faulting instruction
11. TLB miss
12. Page table walk to fetch translation
13. Execute instruction

Locating a Page on Disk

- When a page is non-resident, how do we know where to find it on disk?
- Option: Reuse page table entry
  - If resident, page frame
  - If non-resident, disk sector
- Option: Use file system
  - Code pages: executable image (read-only)
  - Data/Heap/Stack: per-segment file in file system, offset in file = offset within segment

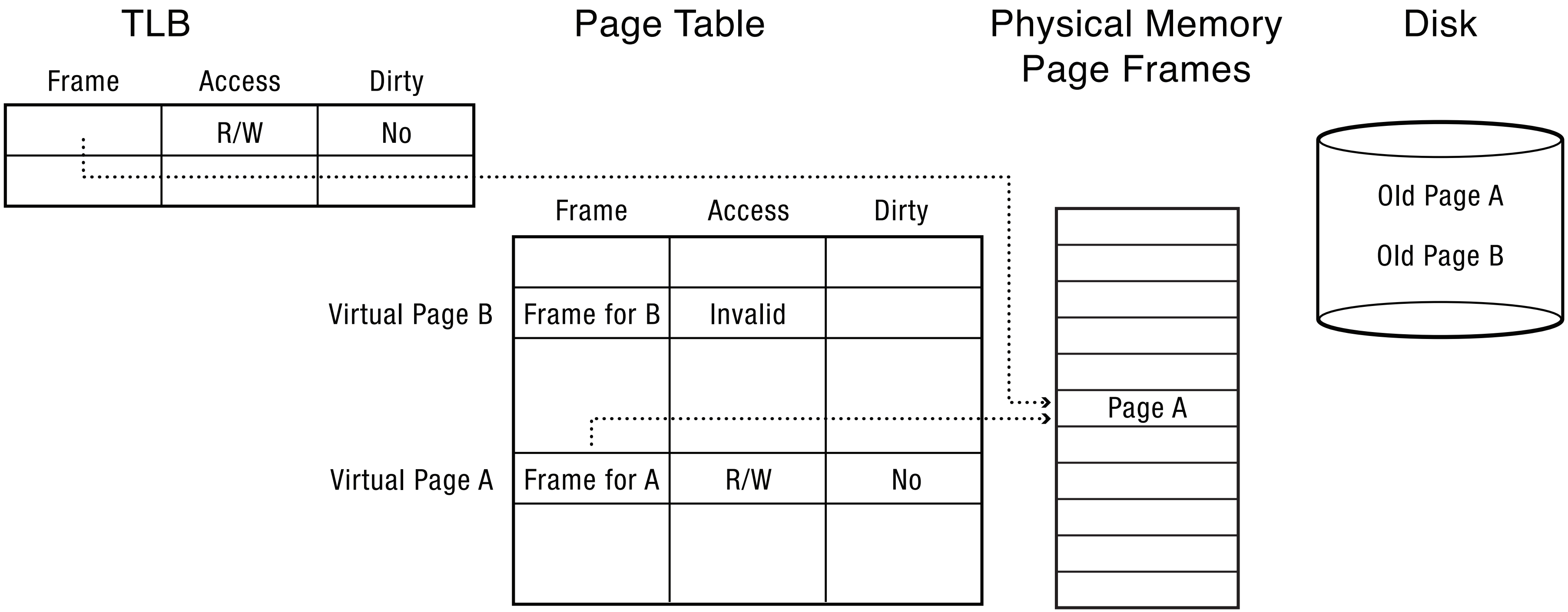
Allocating a Page Frame

- Select old page to evict
- Find all page table entries that refer to old page
  - If page frame is shared (hint: use a coremap)
- Set each page table entry to invalid
- Remove any TLB entries (on any core)
  - Why not: remove TLB entries then set to invalid?
- Write changes on page back to disk, if necessary
  - Why not: write changes to disk then set to invalid?

Has page been modified/recently used?

- Every page table entry has some bookkeeping
  - Has page been modified?
    - Set by hardware on store instruction
    - In both TLB and page table entry
  - Has page been recently used?
    - Set by hardware on in page table entry on every TLB miss
- Bookkeeping bits can be reset by the OS kernel
  - When changes to page are flushed to disk
  - To track whether page is recently used

Tracking Page Modifications (Before)



TLB

Frame	Access	Dirty
	R/W	No

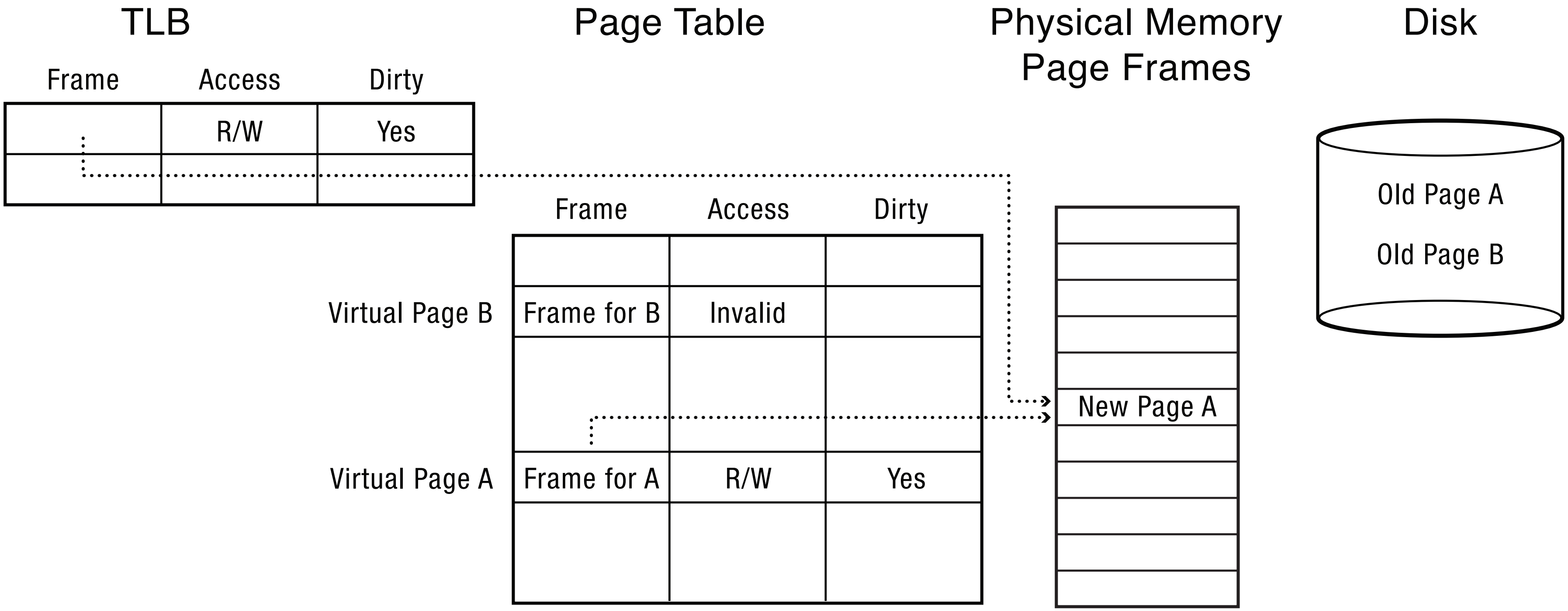
Page Table

	Frame	Access	Dirty
Virtual Page B	Frame for B	Invalid	
Virtual Page A	Frame for A	R/W	No

Physical Memory
Page Frames

Disk

Tracking Page Modifications (After)



TLB

Frame	Access	Dirty
	R/W	Yes

Page Table

	Frame	Access	Dirty
Virtual Page B	Frame for B	Invalid	
Virtual Page A	Frame for A	R/W	Yes

Physical Memory
Page Frames

New Page A

Disk

Old Page A

Old Page B

Modified/Use Bits are (often) Virtual

- Most machines keep modified/use bits in the page table entry (not the core map) – why?
- Physical page is
  - Modified if *any* page table entry that points to it is modified
  - Recently used if *any* page table entry that points to it is recently used
- On MIPS, ok to keep modified/use bits in the core map (map of physical page frames)

Use Bits are Fuzzy

- Page-modified bit must be ground truth
  - What happens if we evict a modified page without writing the changes back to disk?
- Page-use bit can be approximate
  - What happens if we evict a page that is currently being used?
  - “Evict any page not used for a while” is nearly as good as “evict the single page not used for the longest”

Emulating Modified/Use Bits w/ MIPS Software Loaded TLB

- MIPS TLB entries can be read-only or read-write
- On a TLB read miss:
  - If page is clean (in core map), load TLB entry as read-only
  - if page is dirty, load as read-write
  - Mark page as recently used in core map
- On TLB write miss:
  - Mark page as modified/recently used in core map
  - Load TLB entry as read-write
- On a TLB write to an unmodified page:
  - Mark page as modified/recently used in core map
  - Reset TLB entry to be read-write

Emulating a Modified Bit (Hardware Loaded TLB)

- Some processor architectures do not keep a modified bit per page
  - Extra bookkeeping and complexity
- Kernel can *emulate* a modified bit:
  - Set all clean pages as read-only
  - On first write to page, trap into kernel
  - Kernel set modified bit in core map
  - Kernel set page table entry as read-write
  - Resume execution
- Kernel needs to keep track
  - Current page table permission (e.g., read-only)
  - True page table permission (e.g., writeable, clean)

Emulating a Recently Used Bit (Hardware Loaded TLB)

- Some processor architectures do not keep a recently used bit per page
  - Extra bookkeeping and complexity
- Kernel can emulate a recently used bit:
  - Set all pages as invalid
  - On first read or write, trap into kernel
  - Kernel set recently used bit in core map
  - Kernel mark page table entry as read or read/write
  - Resume execution
- Kernel needs to keep track
  - Current page table permission (e.g., invalid)
  - True page table permission (e.g., read-only, writeable)

Models for Application File I/O

- Explicit read/write system calls
  - Data copied to user process using system call
  - Application operates on data
  - Data copied back to kernel using system call
- Memory-mapped files
  - Open file as a memory segment
  - Program uses load/store instructions on segment memory, implicitly operating on the file
  - Page fault if portion of file is not yet in memory
  - Kernel brings missing blocks into memory, restarts process

Advantages to Memory-mapped Files

- Programming simplicity, esp for large files
  - Operate directly on file, instead of copy in/copy out
- Zero-copy I/O
  - Data brought from disk directly into page frame
- Pipelining
  - Process can start working before all the pages are populated
- Interprocess communication
  - Shared memory segment vs. temporary file

Implementing Memory-Mapped Files

- Memory mapped file is a (logical) segment
  - Per segment access control (read-only, read-write)
- File pages brought in on demand
  - Using page fault handler
- Modifications written back to disk on eviction, file close
  - Using per-page modified bit
- Transactional (atomic, durable) updates to memory mapped file requires more mechanism

From Memory-Mapped Files to Demand-Paged Virtual Memory

- Every process segment backed by a file on disk
  - Code segment -> code portion of executable
  - Data, heap, stack segments -> temp files
  - Shared libraries -> code file and temp data file
  - Memory-mapped files -> memory-mapped files
  - When process ends, delete temp files
- Unified memory management across file buffer and process memory

Mach VM

- Goals
  - Portability: many different machine types
  - Small, simple machine dependent layer
  - Feature-rich machine independent layer
- Abstractions
  - Address space
  - Memory objects (permanent storage area)
  - Bind portions of objects to portions of address space
  - User-level handlers

Mach VM Implementation

- Resident page table (OSPP core map)
  - indexed by physical page number (PPN)
- Each PPN in multiple lists
  - Per-object list of resident pages
  - Allocation queues (e.g., global LRU list)
  - Hash table map <object, VPN> -> PPN

Mach VM Implementation

- Address map (one per address space)
  - Linked list of of memory regions/objects
- Memory object
  - Information about permanent storage (e.g., file)
- Pmap (one per address space)
  - Machine-dependent map of VPN -> PPN
  - Operations:
    - Make a PPN addressable at a VPN
    - Unmap a VPN
    - Load context: use this pmap for CPU execution
    - Change protection of a VPN

Mach VM Innovations

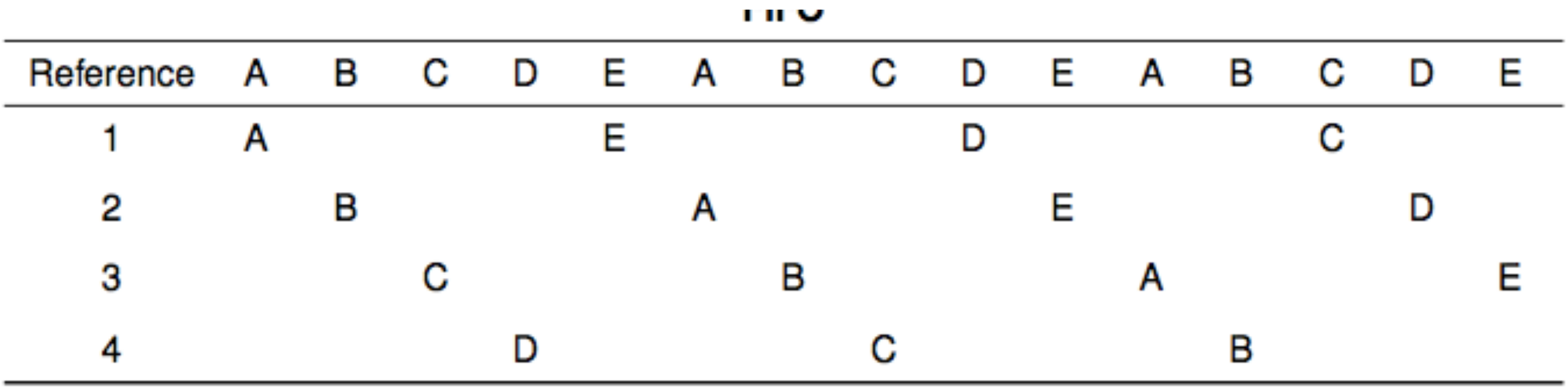
- “real” information is machine-independent data structures
  - Page tables can be discarded
- Machine-independent code only depends on machine-independent data structures
- Shadow objects (e.g., for copy on write)
- Memory objects and user-level pagers
  - See later: scheduler activations

Cache Replacement Policy

- On a cache miss, how do we choose which entry to replace?
  - Assuming the new entry is more likely to be used in the near future
  - In direct mapped caches, not an issue!

- Policy goal: reduce cache misses
  - Improve expected case performance
  - Also: reduce likelihood of very poor performance

FIFO in Action



FIFO

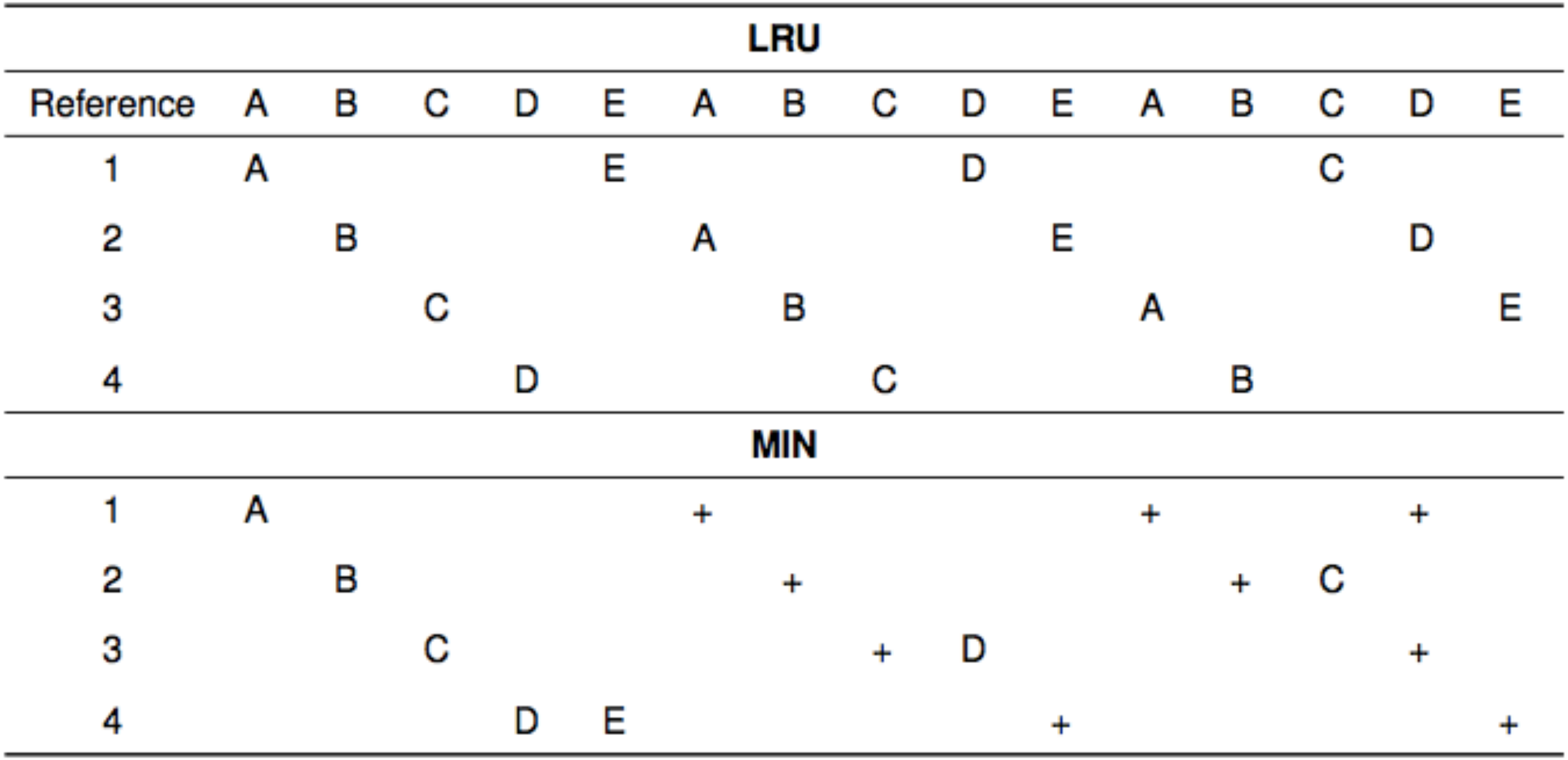
Reference	A	B	C	D	E	A	B	C	D	E	A	B	C	D	E
1	A				E				D				C		
2		B				A				E				D	
3			C				B				A				E
4				D				C				B			

Worst case for FIFO is if program strides through memory that is larger than the cache

MIN, LRU, LFU

- MIN
  - Replace the cache entry that will not be used for the longest time into the future
  - Optimality proof based on exchange: if evict an entry used sooner, that will trigger an earlier cache miss
- Least Recently Used (LRU)
  - Replace the cache entry that has not been used for the longest time in the past
  - Approximation of MIN
- Least Frequently Used (LFU)
  - Replace the cache entry used the least often (in the recent past)

LRU/MIN for Sequential Scan



LRU															
Reference	A	B	C	D	E	A	B	C	D	E	A	B	C	D	E
1	A				E				D				C		
2		B				A				E				D	
3			C				B				A				E
4				D				C				B			
MIN															
1	A					+					+			+	
2		B					+					+	C		
3			C					+	D					+	
4				D	E					+					+

LRU

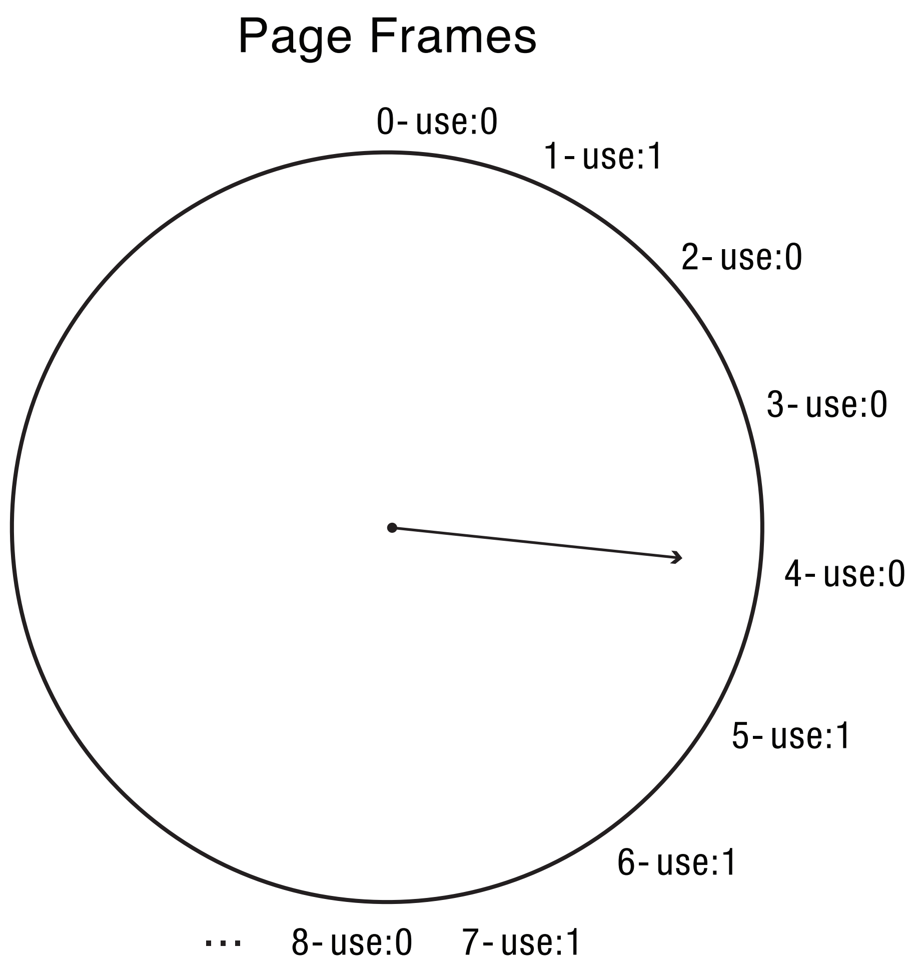
Reference	A	B	A	C	B	D	A	D	E	D	A	E	B	A	C
1	A		+				+				+			+	
2		B			+								+		
3				C					E			+			
4						D		+		+					C
FIFO															
1	A		+				+		E						
2		B			+						A			+	
3				C								+	B		
4						D		+		+					C
MIN															
1	A		+				+				+			+	
2		B			+								+		C
3				C					E			+			
4						D		+		+					

Question

- How accurately do we need to track the least recently/least frequently used page?
  - If miss cost is low, any approximation will do
    - Hardware caches
  - If miss cost is high but number of pages is large, any not recently used page will do
    - Main memory paging with small pages
  - If miss cost is high and number of pages is small, need to be precise
    - Main memory paging with superpages

Clock Algorithm: Estimating LRU

- Hardware sets use bit
- Periodically, OS sweeps through all pages
- If page is unused, reclaim
- If page is used, mark as unused



Nth Chance: Not Recently Used

- Instead of one bit per page, keep an integer
  - notInUseSince: number of sweeps since last use
- Periodically sweep through all page frames

```
if (page is used) {
    notInUseForXSweeps = 0;
} else if (notInUseForXSweeps < N) {
    notInUseForXSweeps++;
} else {
     reclaim page; write modifications if needed
}
```

Implementation Note

- Clock and Nth Chance can run synchronously
  - In page fault handler, run algorithm to find next page to evict
  - Might require writing changes back to disk first
- Or asynchronously
  - Create a thread to maintain a pool of recently unused, clean pages
  - Find recently unused dirty pages, write mods back to disk
  - Find recently unused clean pages, mark as invalid and move to pool
  - On page fault, check if requested page is in pool!
  - If not, evict page from the pool

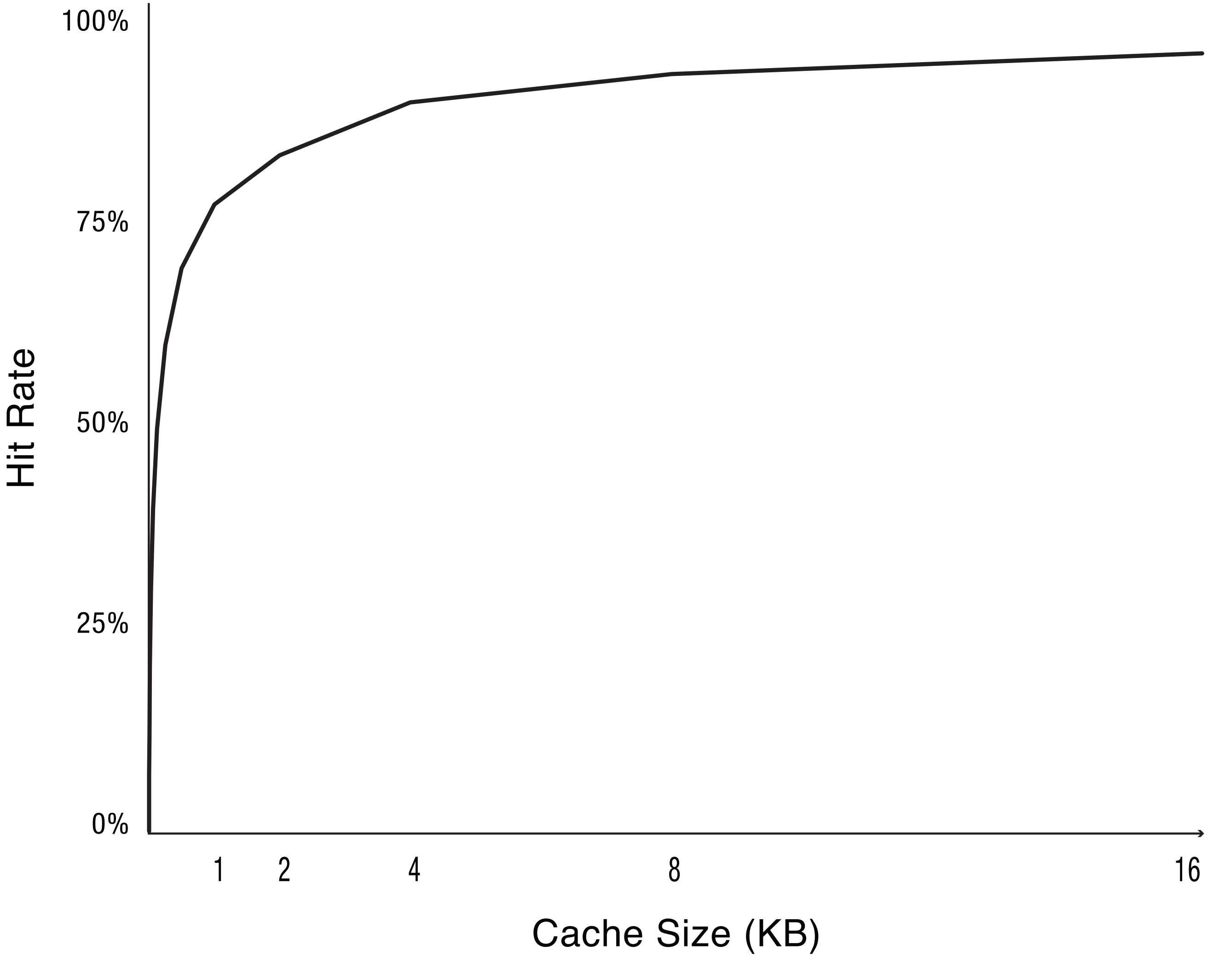
Recap

- MIN is optimal
  - replace the page or cache entry that will be used farthest into the future
- LRU is an approximation of MIN
  - For programs that exhibit spatial and temporal locality
- Clock/Nth Chance is an approximation of LRU
  - Bin pages into sets of “not recently used”

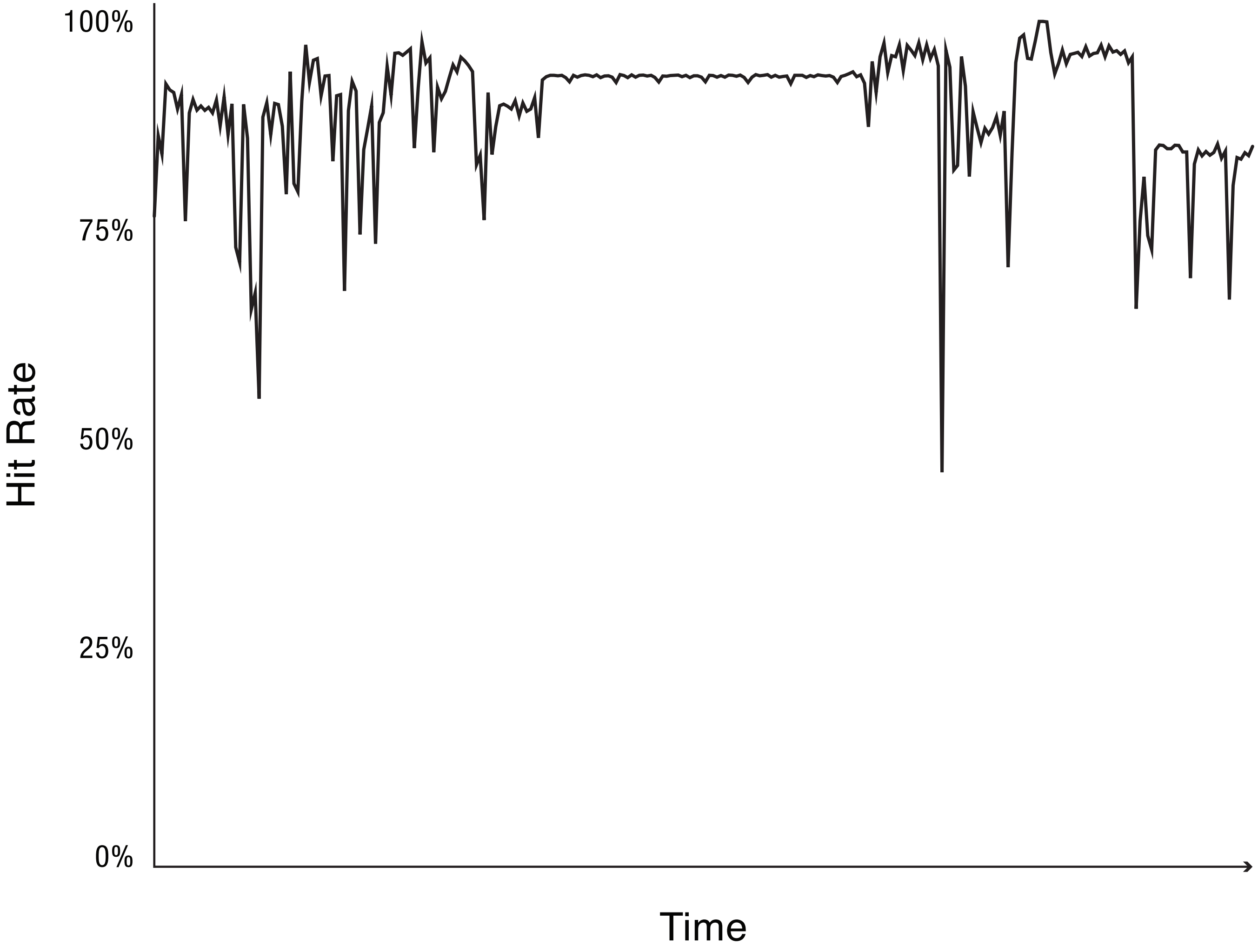
Working Set Model

- Working Set: set of memory locations that need to be cached for reasonable cache hit rate
- Thrashing: when system has too small a cache
  - For set of processes running concurrently

Cache Working Set



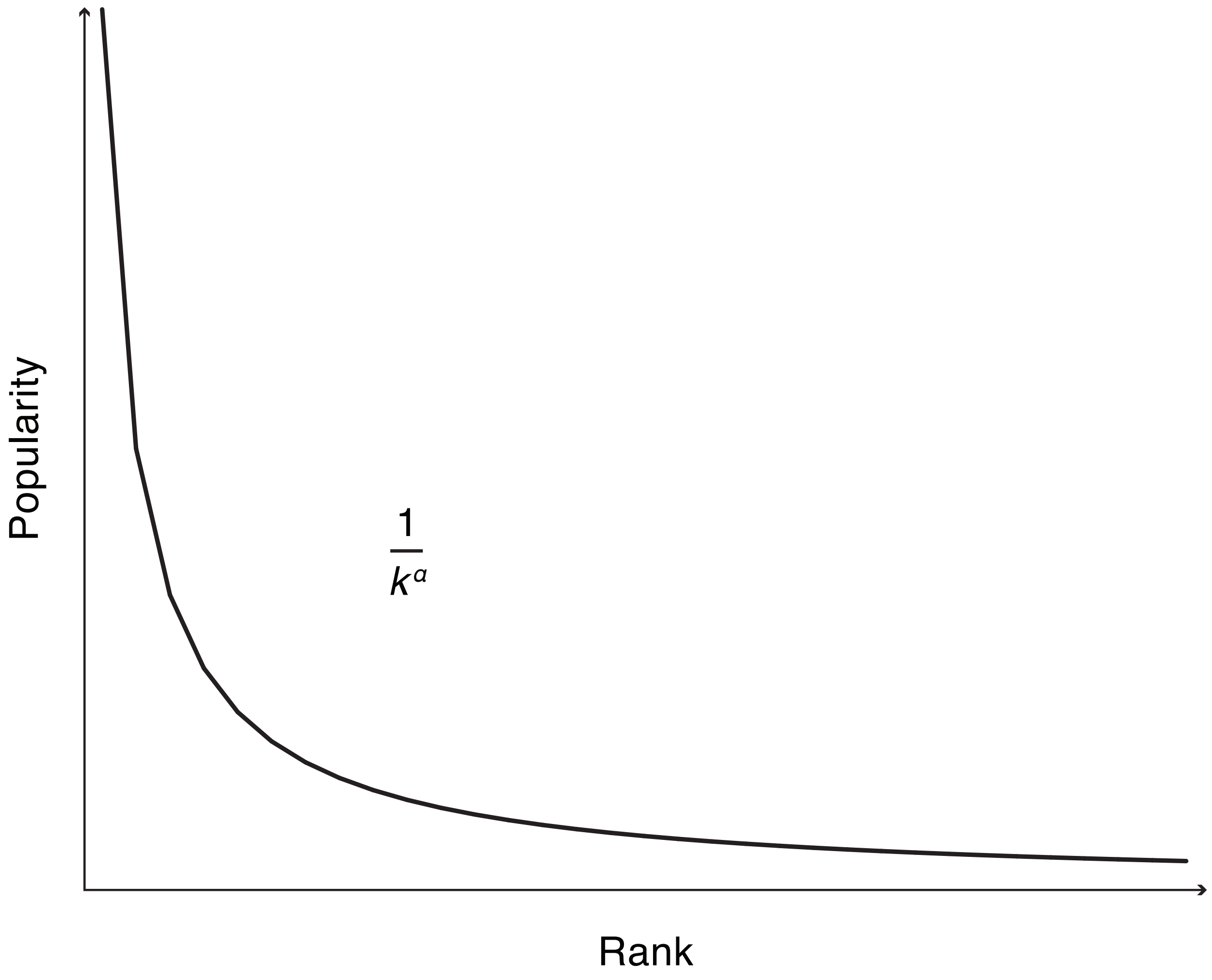
Phase Change Behavior



Zipf Distribution

- Caching behavior of many systems are not well characterized by the working set model
- An alternative is the Zipf distribution
  - Popularity ~ 1/k^c, for kth most popular item, 1 < c < 2

Zipf Distribution

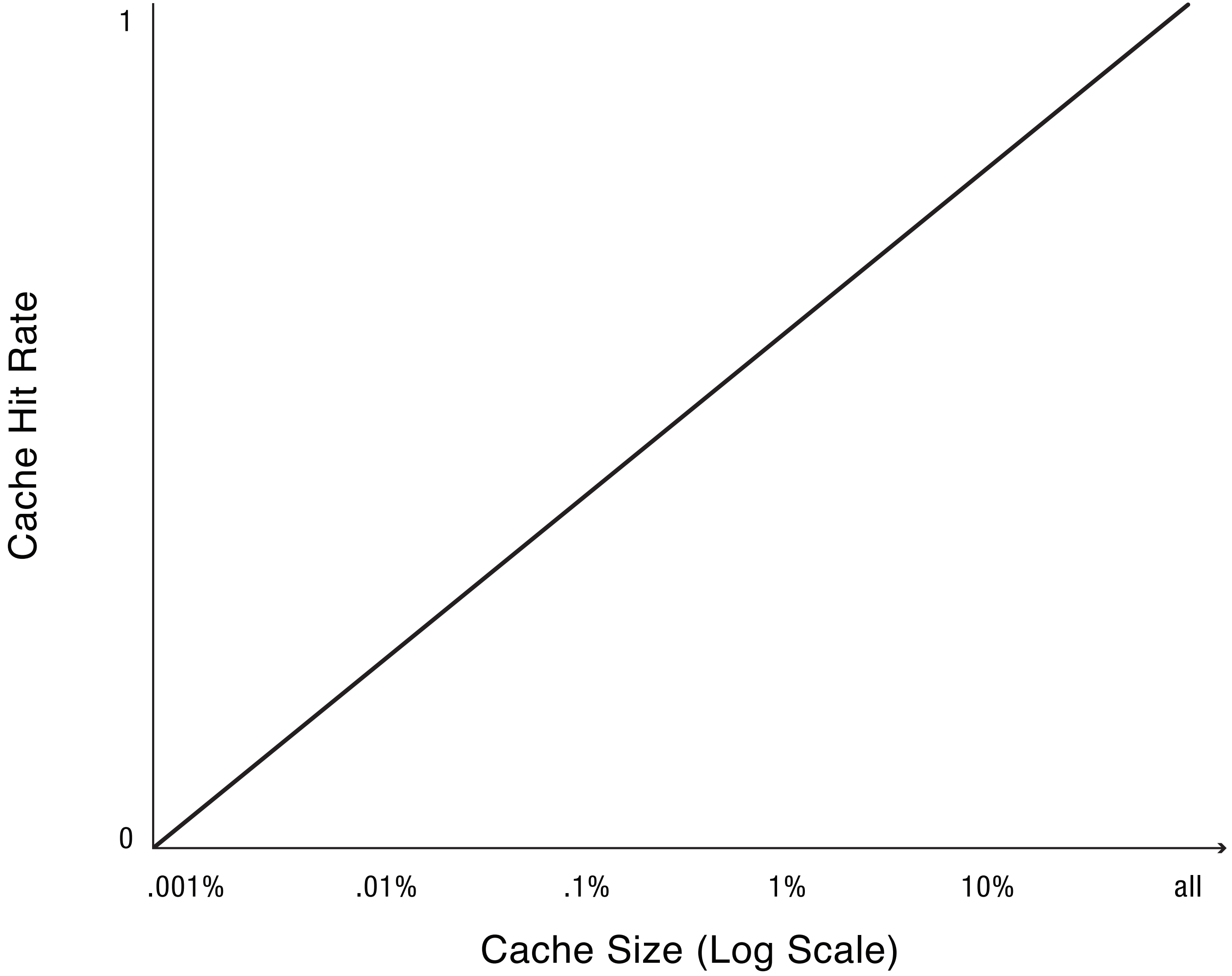


Zipf Examples

- Web pages
- Movies
- Library books
- Words in text
- Salaries
- City population
- …

Common thread: popularity is self-reinforcing

Zipf and Caching



Implementing LFU

- First time an object is referenced, is it:
  - Unpopular, so evict quickly?
  - New and possibly popular, so avoid evicting?
- Compute frequency from first observation
  - # of references/time since first loaded into cache
- Ordering changes dynamically, even when there are no misses
  - re-prioritize each time we need to evict a page?

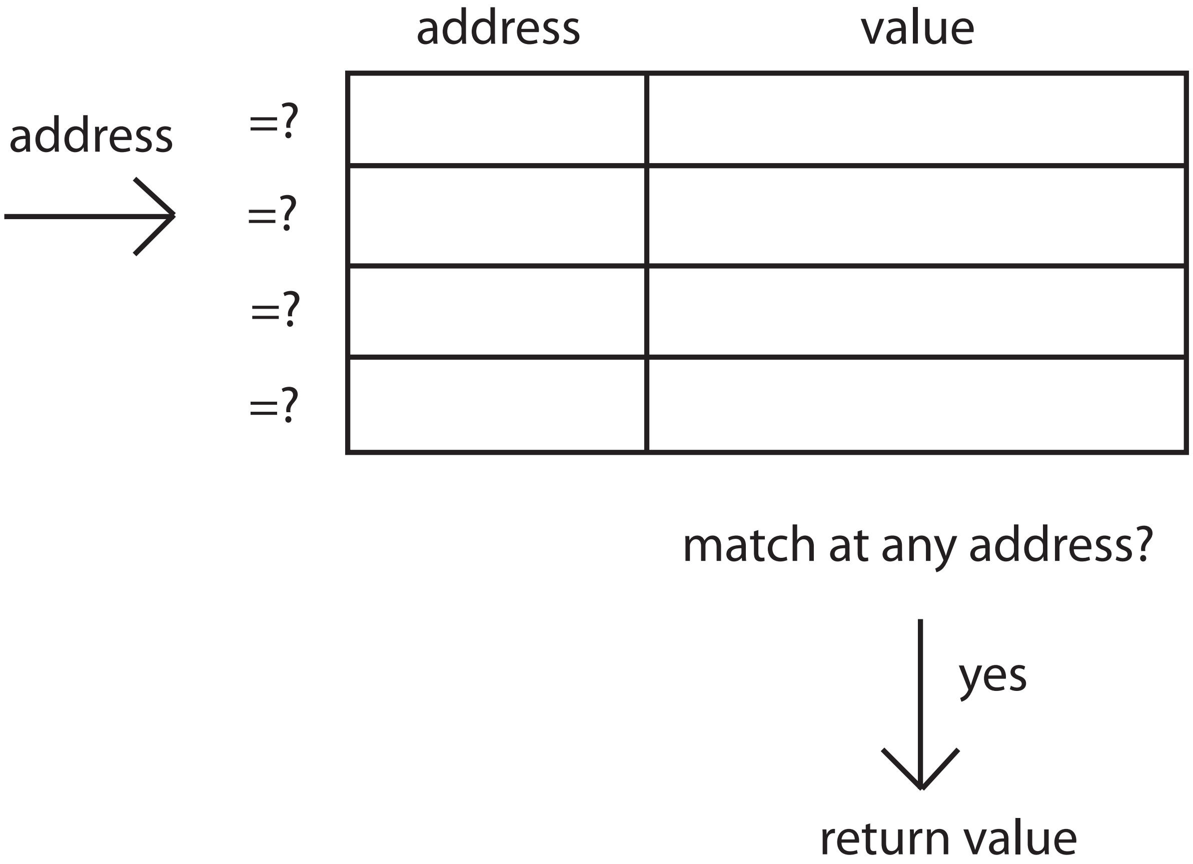
More complexities

- What if object size can vary
  - Evict bigger objects to make room for more smaller ones?
- What if cost of refetch can vary
  - Cost of fetch from flash vs. fetch from disk
  - If item needs computation
- Replacement algorithm can be very complex

Power of Choices

- Pick k objects at random
- Evaluate (using any function) which is best object to replace
  - Evict that one!
- Keep next best 2-3 objects in pool for next iteration
- If k ~ 10, better than a 10th chance list!

Cache Lookup: Fully Associative



address

value

address

=?

=?

=?

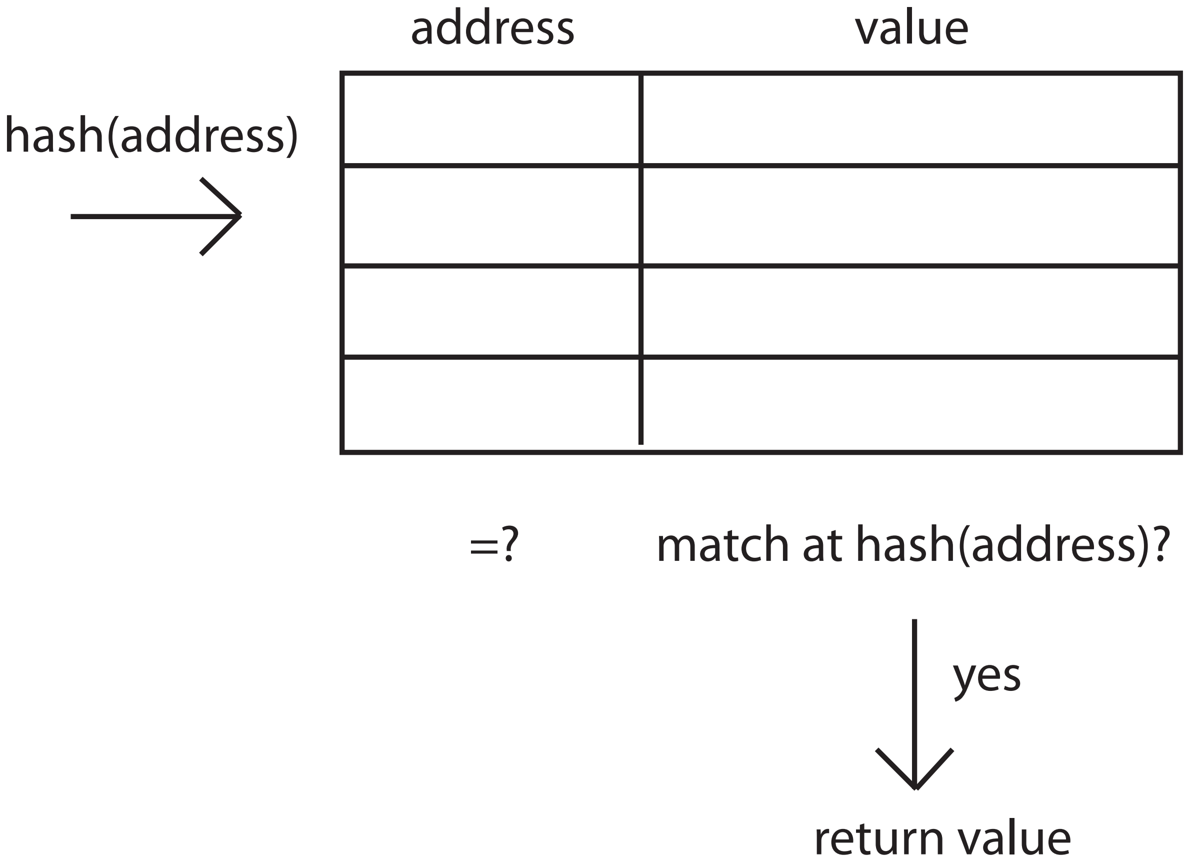
=?

match at any address?

yes

return value

Cache Lookup: Direct Mapped



address

value

hash(address)

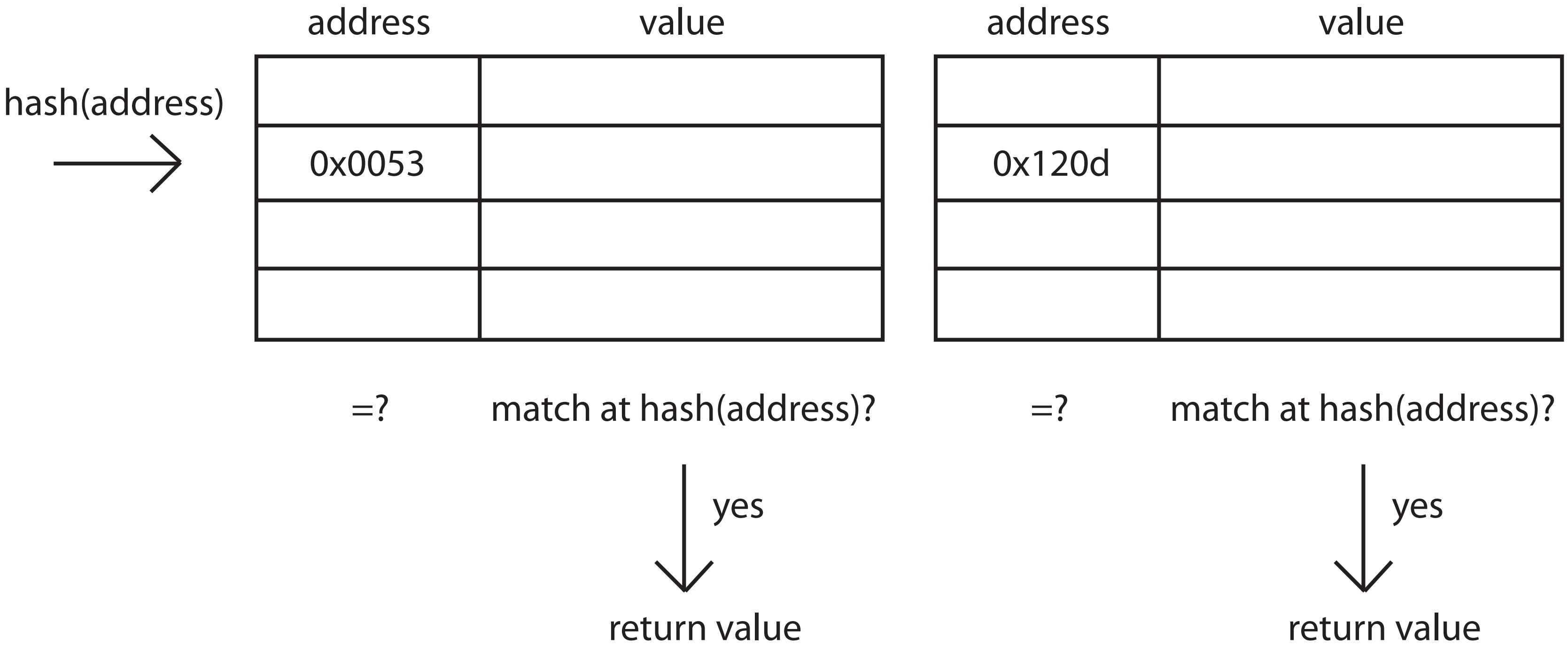
=?

match at hash(address)?

yes

return value

Cache Lookup: Set Associative



hash(address)

address	value
0x0053	

=? match at hash(address)?

yes

return value

address	value
0x120d	

=? match at hash(address)?

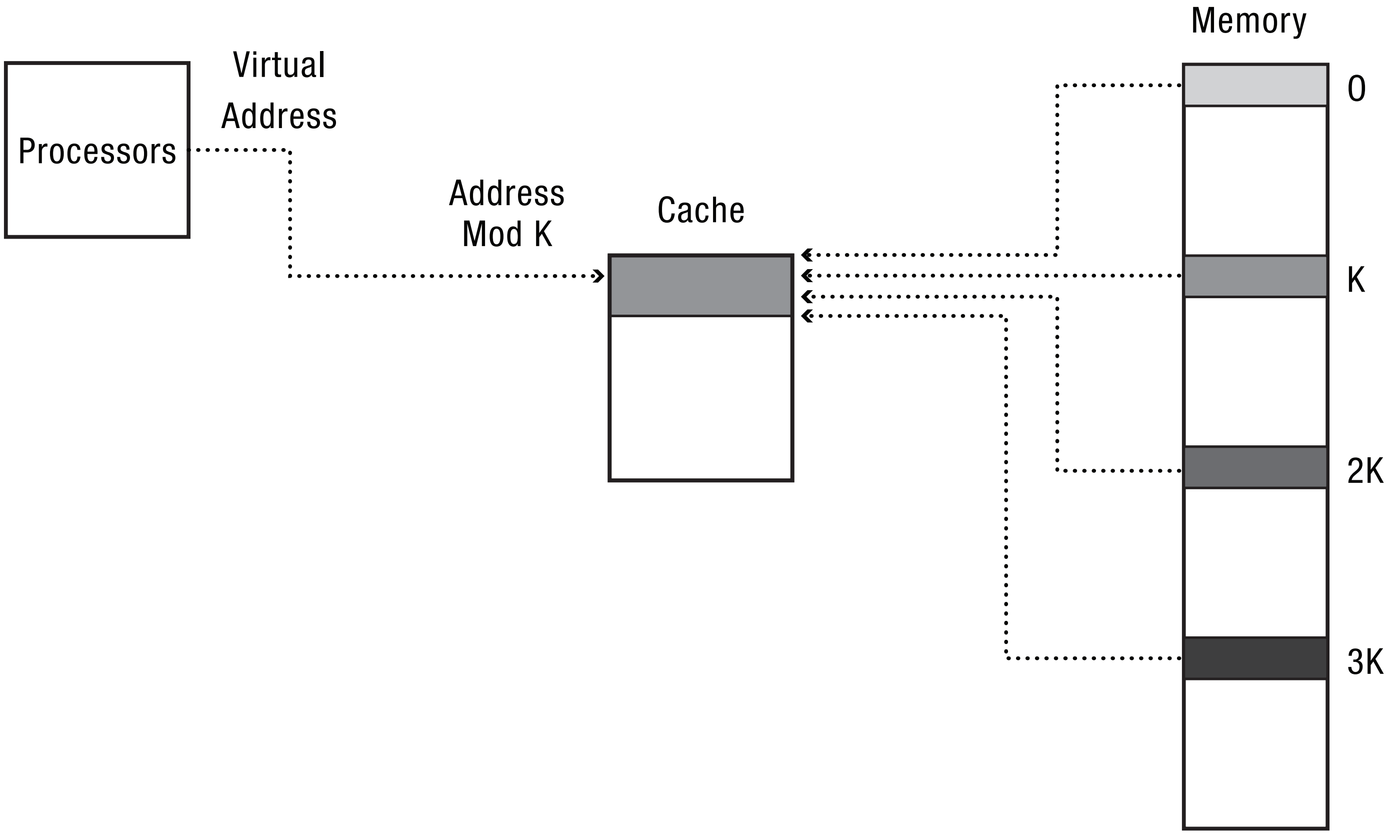
yes

return value

Page Coloring

- What happens when cache size >> page size?
  - Direct mapped or set associative
  - Multiple pages map to the same cache line
- OS page assignment matters!
  - Example: 8MB cache, 4KB pages
  - 1 of every 2K pages lands in same place in cache
- What should the OS do?

Page Coloring


Memory
Virtual
Address
Processors
Address
Mod K
Cache
0
K
2K
3K