Address Translation

Chapter 8 OSPP

Part I: Basics

Important?

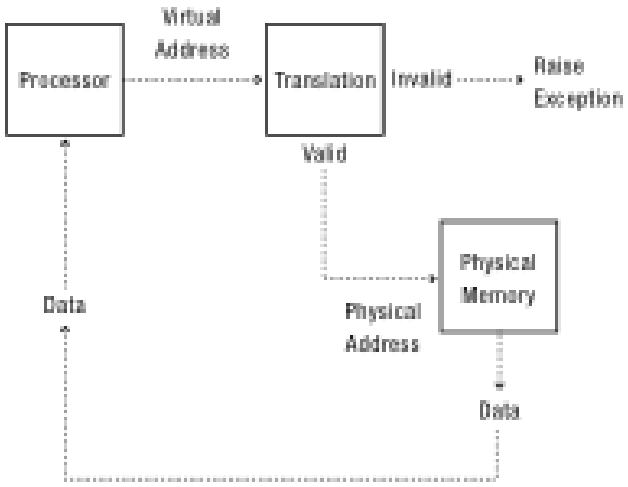
- Process isolation
- IPC
- Shared code
- Program initialization
- Efficient dynamic memory allocation
- Cache management
- Debugging
- Efficient I/O
- Memory mapped files
- Virtual memory
- Checkpoint/restart
- …

All problems in computer science can be solved by another level of indirection!

Main Points

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

Address Translation Concept



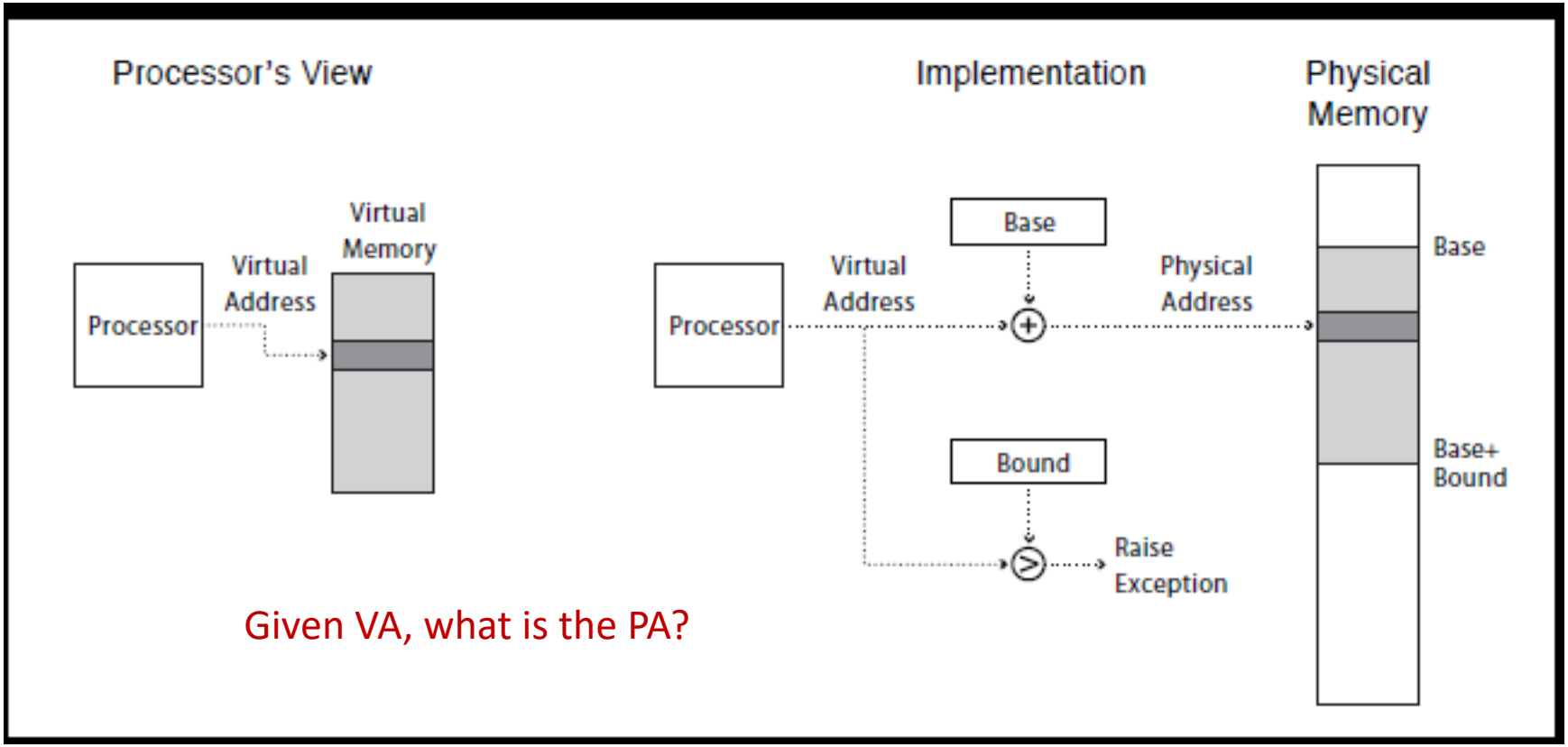
Address Translation Goals

- Memory protection
- Memory sharing
  - Shared libraries, shared-memory IPC
- Sparse addresses (64 bit addresses)
  - Multiple regions of dynamic allocation (heaps/stacks)
  - Allow room for growth
- Efficiency
  - Memory placement
  - Runtime lookup
  - Compact translation tables
- Portability
  - OS must exploit hardware

Bonus Feature

- What can you (OS) 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
  - Demand paging
  - Fill on demand
  - Memory mapped files

Virtually Addressed Base and Bounds



Hardware support is minimal: base register, bound register

Question

- With virtually addressed base and bounds, what is saved/restored on a process context switch w/r to memory?

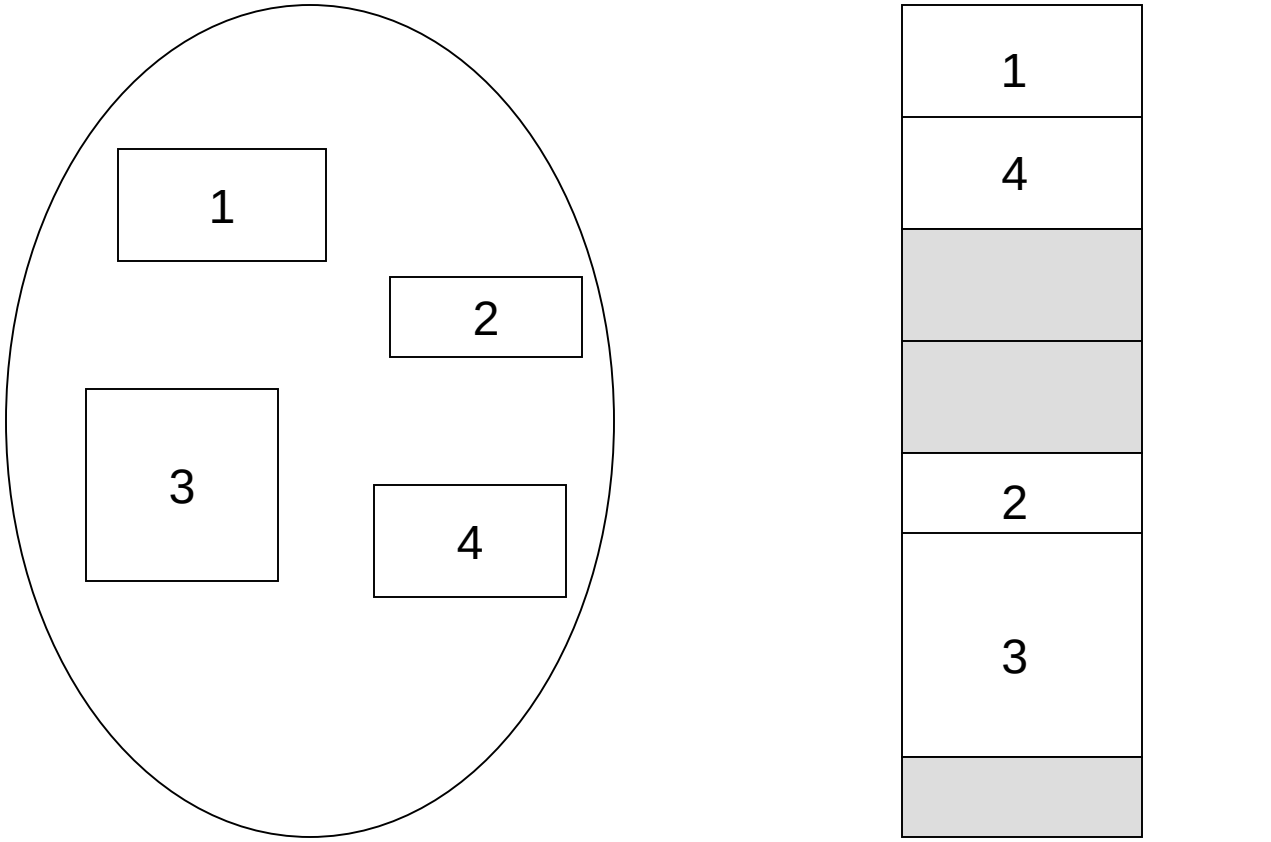
Virtually Addressed Base and Bounds

- Pros?
- Cons?

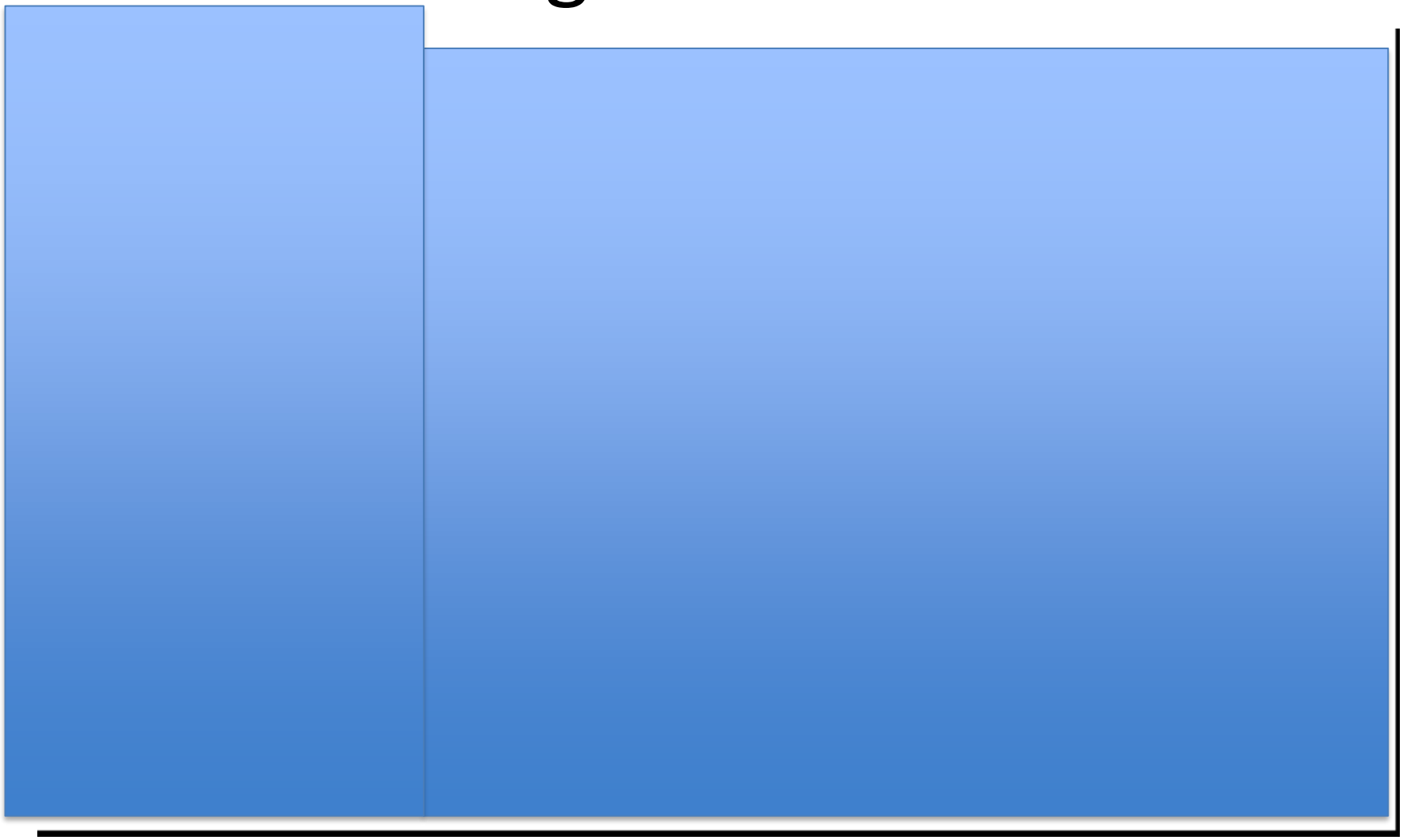
Segmentation

- Segment is a contiguous region of *virtual* memory
- Each process has a segment table (in hardware or mem)
  - Entry in table for each segment
- Segment can be located anywhere in physical memory
  - Each segment has: start, length, access permission
- Processes can share segments
  - Same start, length, same/different access permissions
  - Great for shared libraries

Logical View



Segmentation



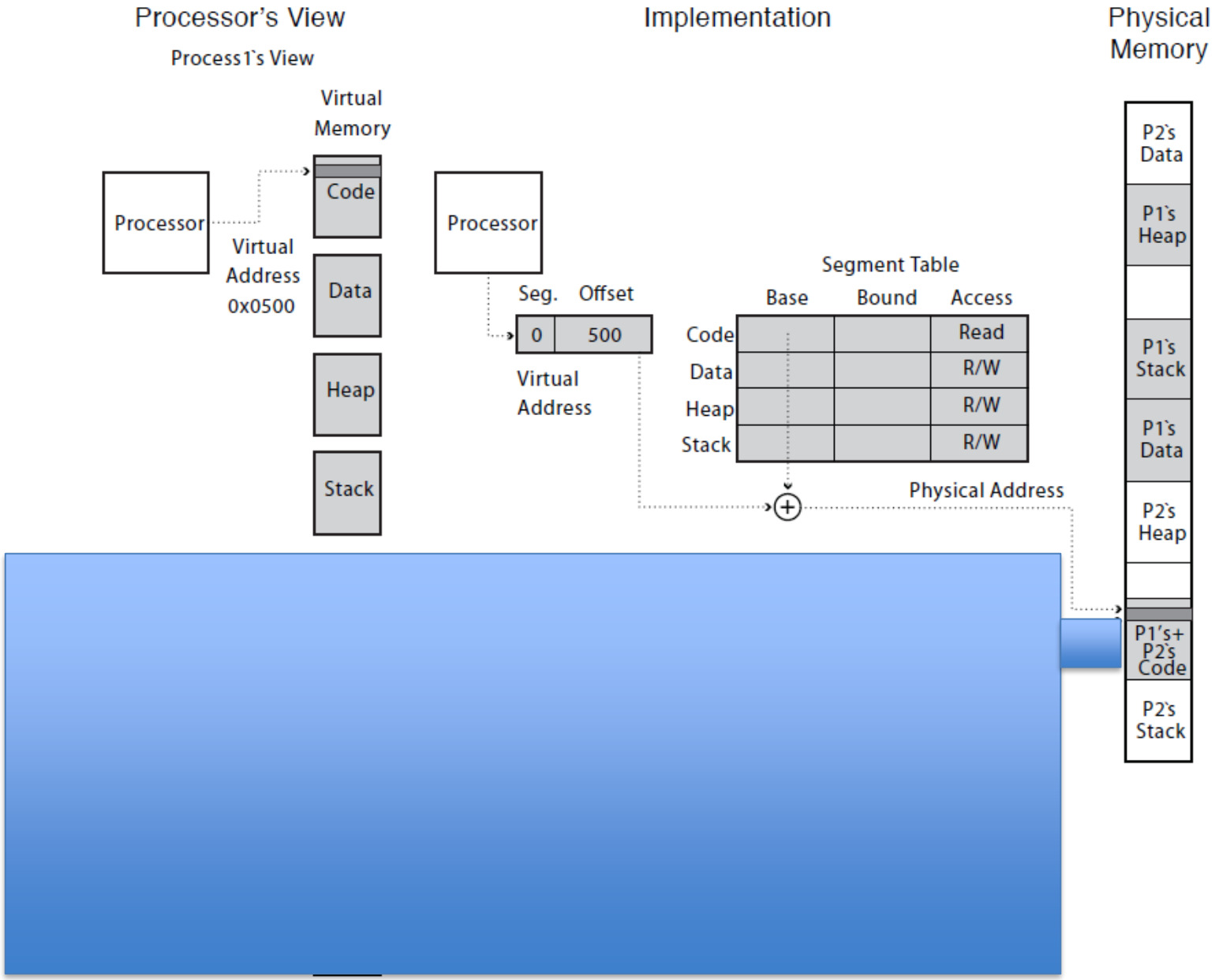
Hardware support: segment table start and length register (# segs)

Question

- With segmentation, what is saved/restored on a process context switch?

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, heap)
    - trap into kernel
    - make a copy of the segment and resume


Processor's View
Process1`s View
Virtual Memory
Processor
Virtual Address 0x0500
Code
Data
Heap
Stack
Implementation
Processor
Seg.
Offset
0
500
Virtual Address
Segment Table
Base
Bound
Access
Code
Read
Data
R/W
Heap
R/W
Stack
R/W
Physical Address
Physical Memory
P2`s Data
P1`s Heap
P1`s Stack
P1`s Data
P2`s Heap
P1's+ P2`s Code
P2`s Stack

Dynamic Segments & Zero-on-Reference

- Dynamic segments: not all impl. allow this
  - When program uses memory beyond bound (e.g. end of stack)
  - Segmentation fault into OS kernel
  - Kernel can then allocate some additional memory
    - How much?
- Zeros the memory
  - idea: set segment bound (i.e. stack) artificially low
  - at seg fault, kernel zeros the memory
  - avoid accidentally leaking information!
- Modify segment table
- Resume process

More on zero'ing

- If data is so sensitive, why not have programs zero their own memory?
  - `bzero` system call

- Background: when CPU is idle, we can zero memory not currently allocated

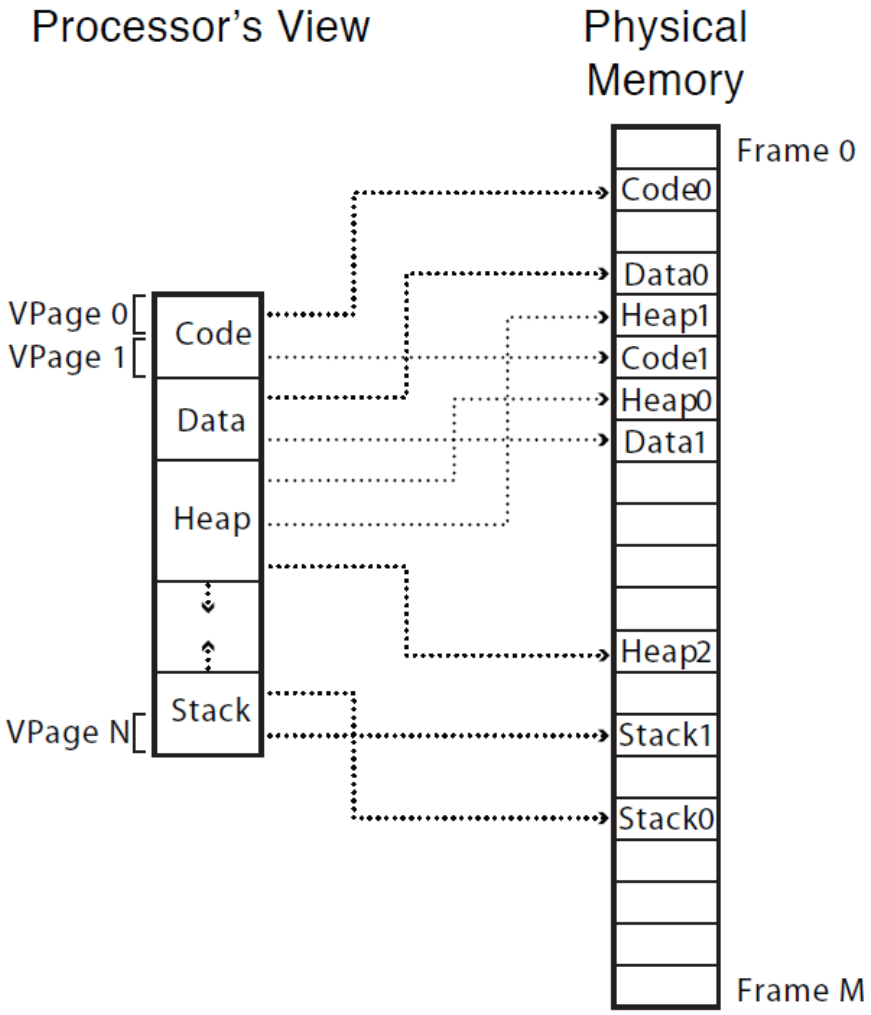
Segmentation

- Pros?
- Cons?

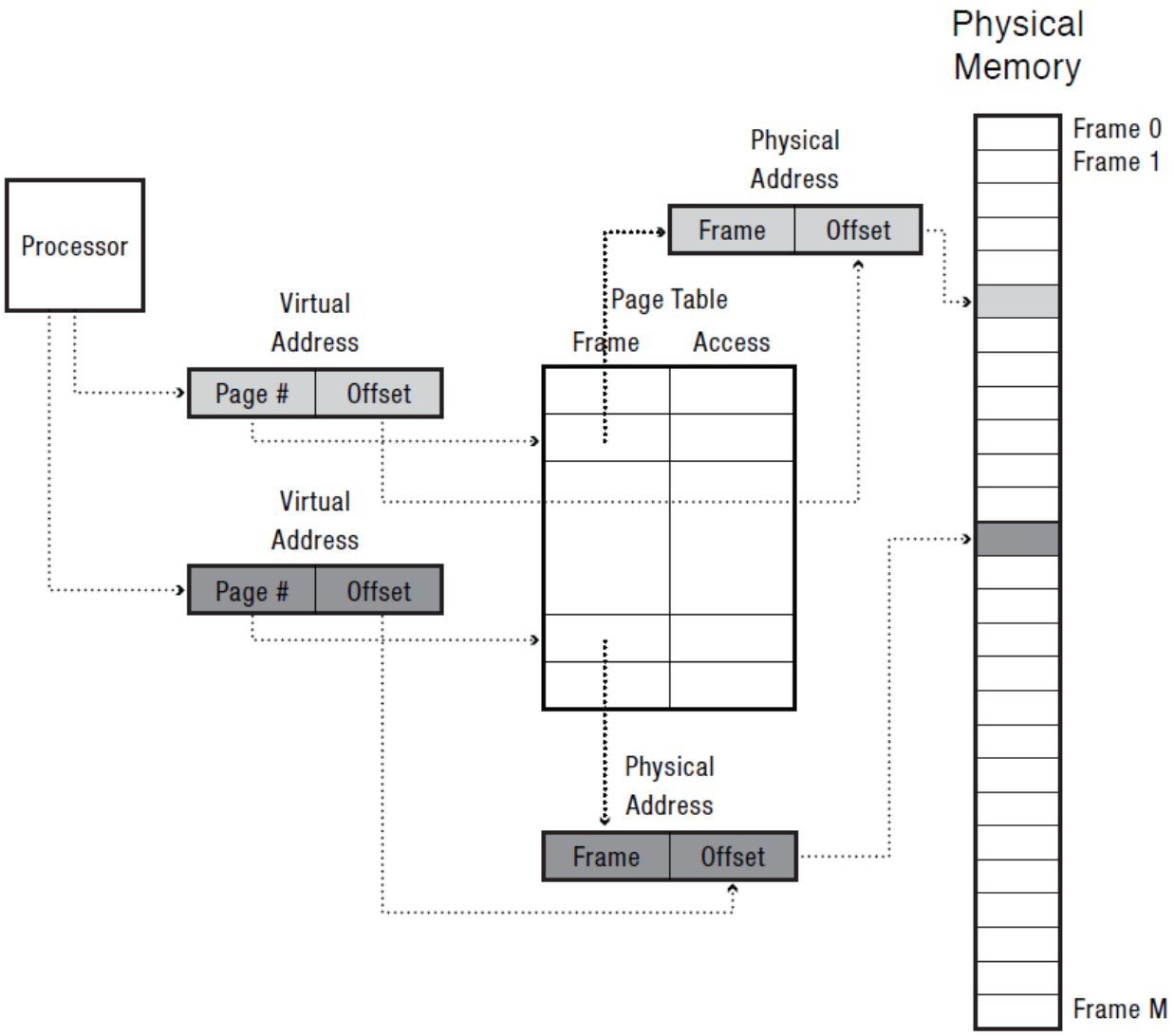
Solve Fragmentation: Paged Translation

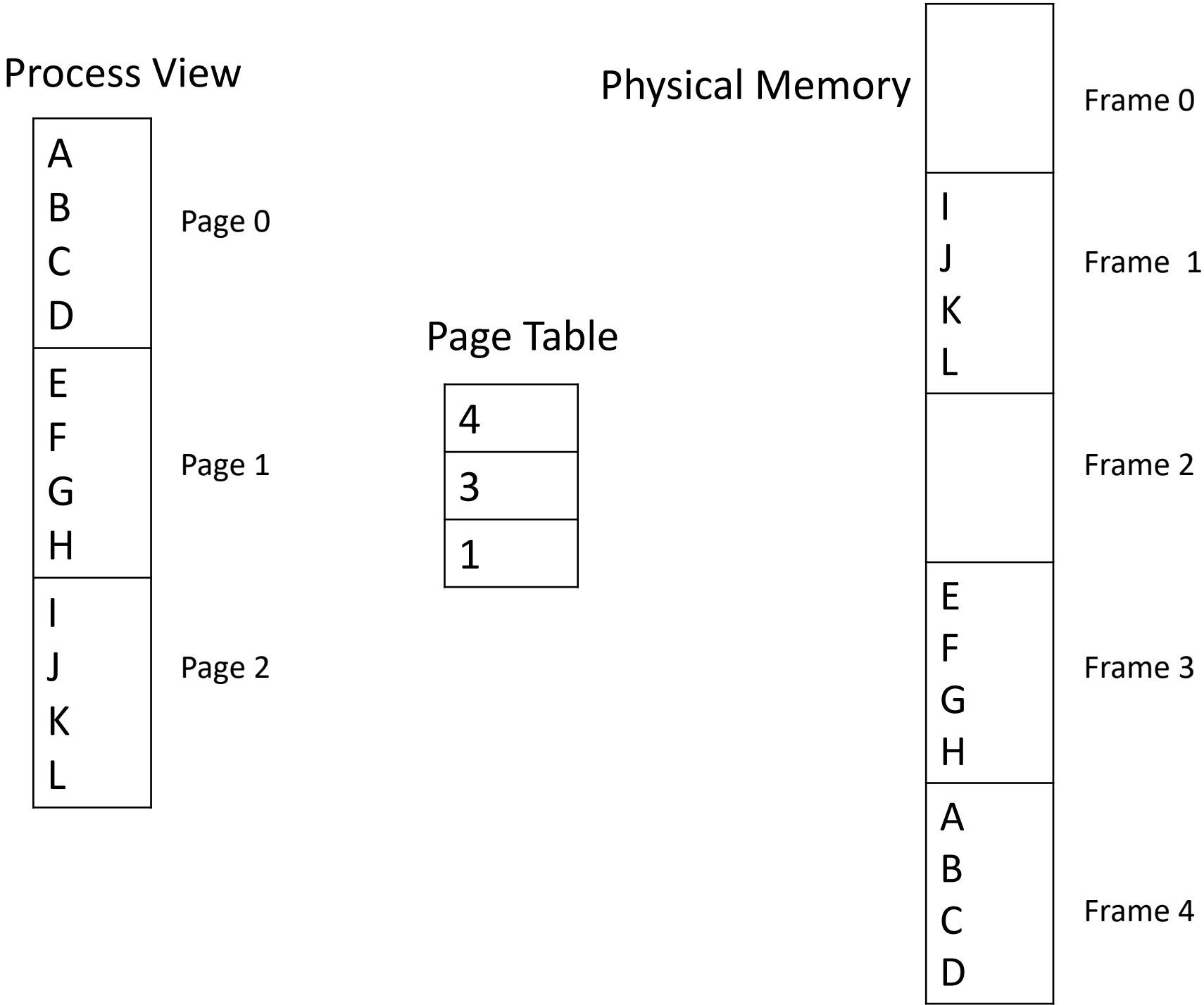
- Manage memory in fixed size units, or pages
- Finding a free page is easy
  - Bitmap allocation: 0011111100000001100
  - Each bit represents one physical page frame
- Each process has its own page table
  - Stored in physical memory
- Hardware registers
    - pointer to page table start
    - page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

Paged Translation (Abstract)



Paged Translation (Implementation)




Process View
A
B
C
D
Page 0
E
F
G
H
Page 1
I
J
K
L
Page 2
Page Table
4
3
1
Physical Memory
Frame 0
I
J
K
L
Frame 1
Frame 2
E
F
G
H
Frame 3
A
B
C
D
Frame 4

Comparison

- Like segmentation, paging adds a level of indirection
- Page size is smaller than segment size generally
- What about translation overhead?
- What about memory overhead (size) of paging vs. segmentation?

Paging Questions

- With paging, what is saved/restored on a process context switch?
  - Pointer to page table, size of page table
  - Page table itself is in main memory
- What if page size is very small?
  - Big page tables, lots of I/O (as we will see)
- What if page size is very large?
  - Internal fragmentation: if we don't need all of the space inside a fixed size chunk

Paging and Copy on Write

- Can we share memory between processes?
  - Set entries in both page tables to point to same page frames
  - Need *core map* of page frames to track which processes are pointing to which page frames (e.g., reference count): why?

- UNIX fork with copy on write
  - 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

Demand Paging/Fill On Demand

- Can I start running a program before its code is in physical memory?
  - Set all page table entries to invalid
  - When a page is referenced for first time, kernel trap, “page fault”
  - Kernel brings page in from disk
  - Resume execution
  - Remaining pages can be transferred in the background while program is running

Data Breakpoints

- Please trace variable A
- Mark page P containing A as read-only
- If P is changed, trap into kernel, and see if A actually changed?
- Why is this better with paging vs. segmentation?

Page Table Issue

- 64 bit machines
- Page table(s) can get huge
- Need to address this
- 16 bit page size, 50 bits for pages, 2^50 entries in PT PER process!

Caching and Demand-Paged Virtual Memory

Chapter 9 OSPP

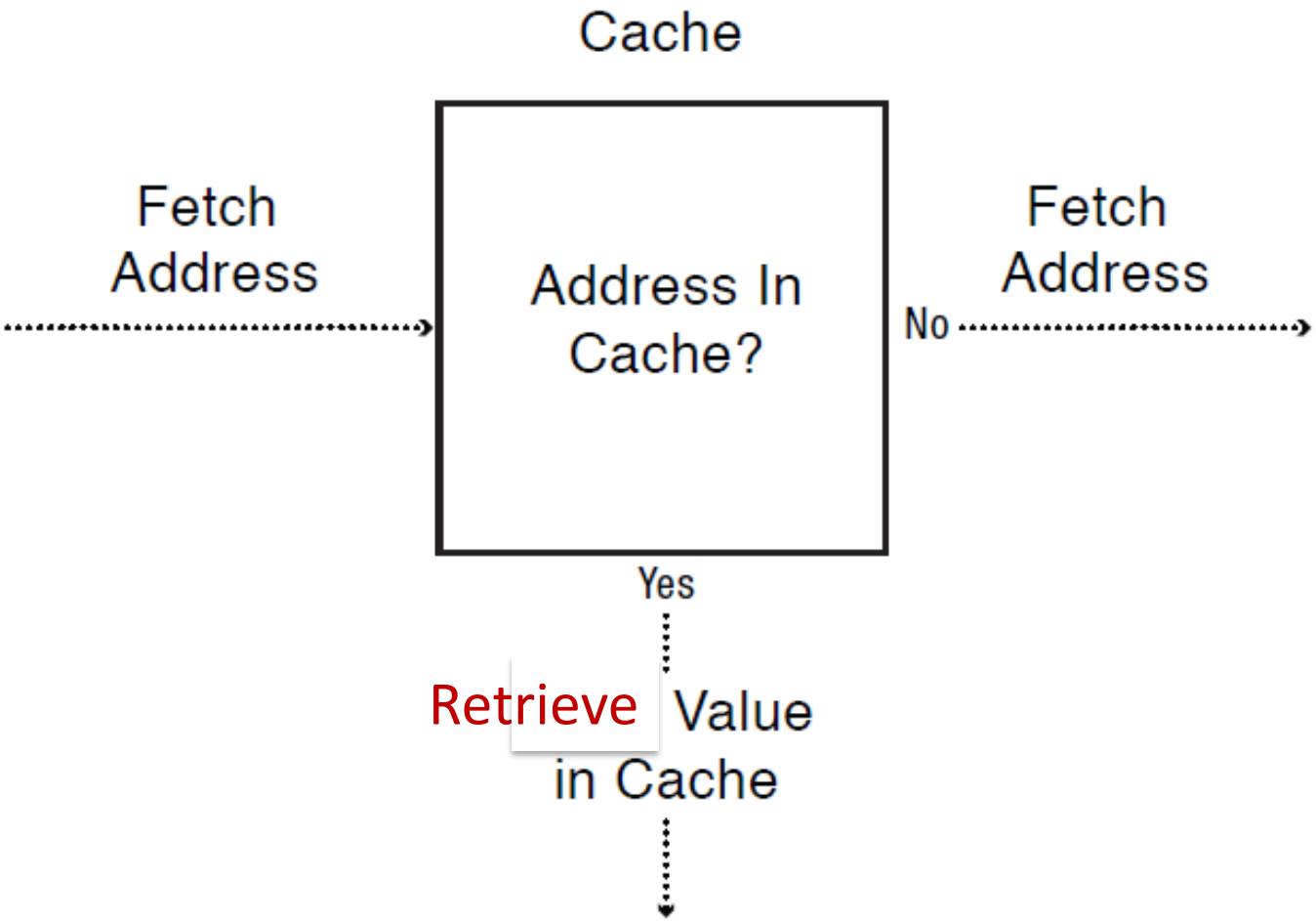
Caching: Address Translation, and Virtual Memory

- Caching
  - Speed up address translation (TLB)
  - Implement virtual memory (memory as a cache for backing store): demand-paging
  - Memory-mapped files

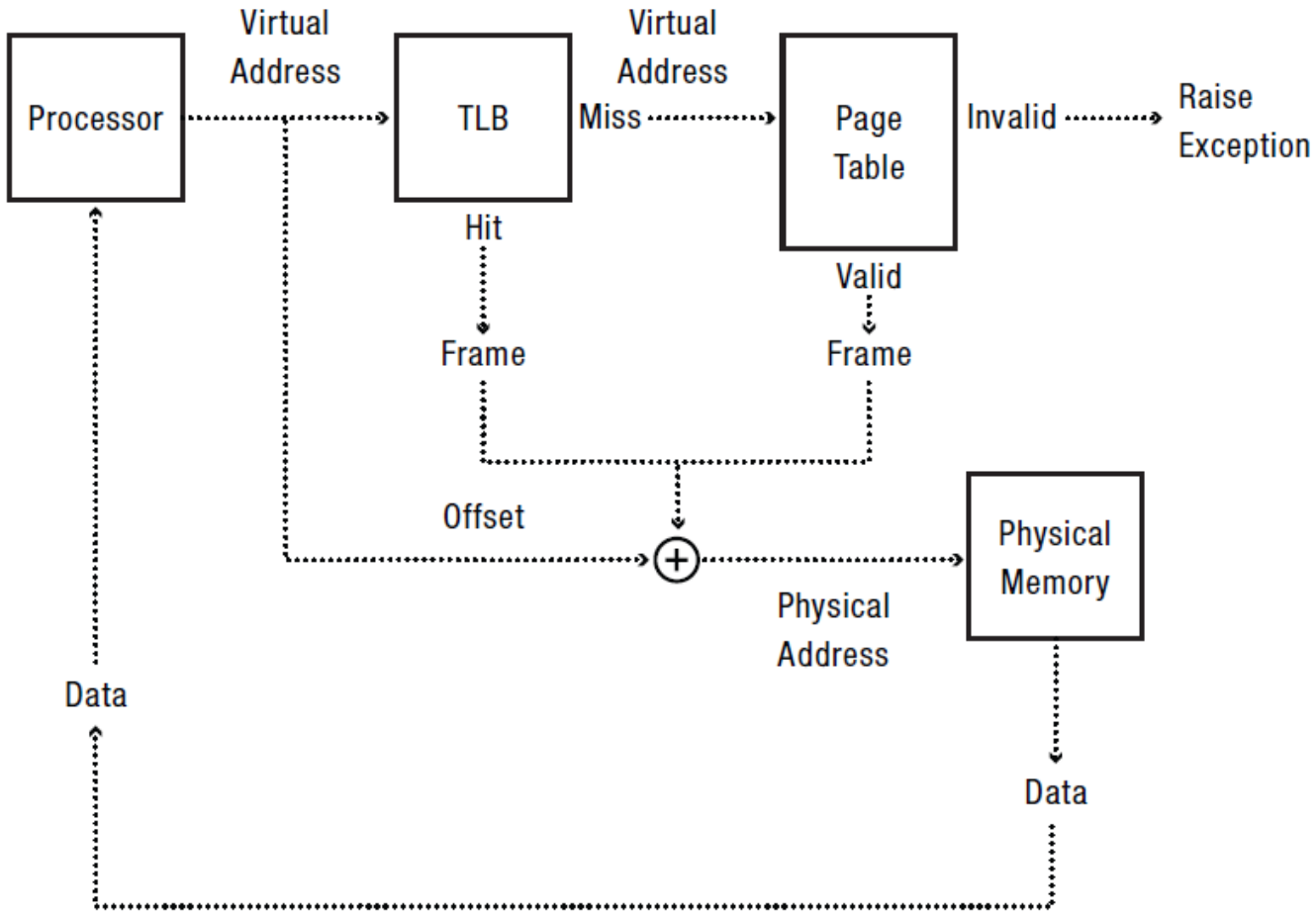
Definitions

- Cache
  - Copy of data that is faster to access than the original
  - Hit: if cache has copy
  - Miss: if cache does not have copy
- Cache block
  - Unit of cache storage (multiple memory locations)
- Temporal locality
  - Programs tend to reference the same memory locations multiple times
  - Example: instructions in a loop
- Spatial locality
  - Programs tend to reference nearby locations
  - Example: data in a loop

Cache Concept (Read)



TLB and Page Table Translation



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

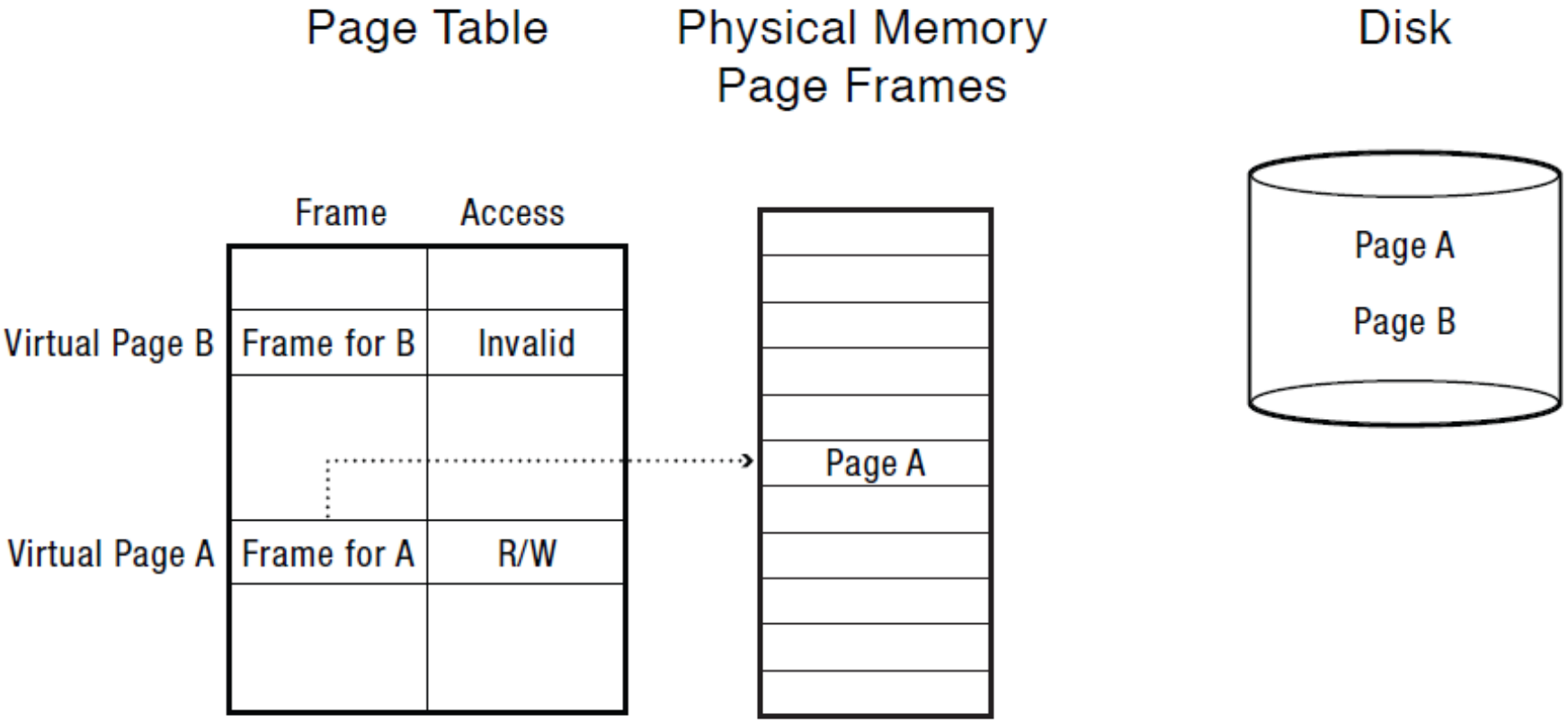
Main Points

- Can we provide the illusion of near infinite memory in limited physical memory?
  - Demand-paged virtual memory
  - Memory-mapped files
- How do we choose which page to replace?
  - FIFO, MIN, LRU, LFU, Clock
- What types of workloads does caching work for, and how well?
  - Spatial/temporal locality vs. Zipf workloads

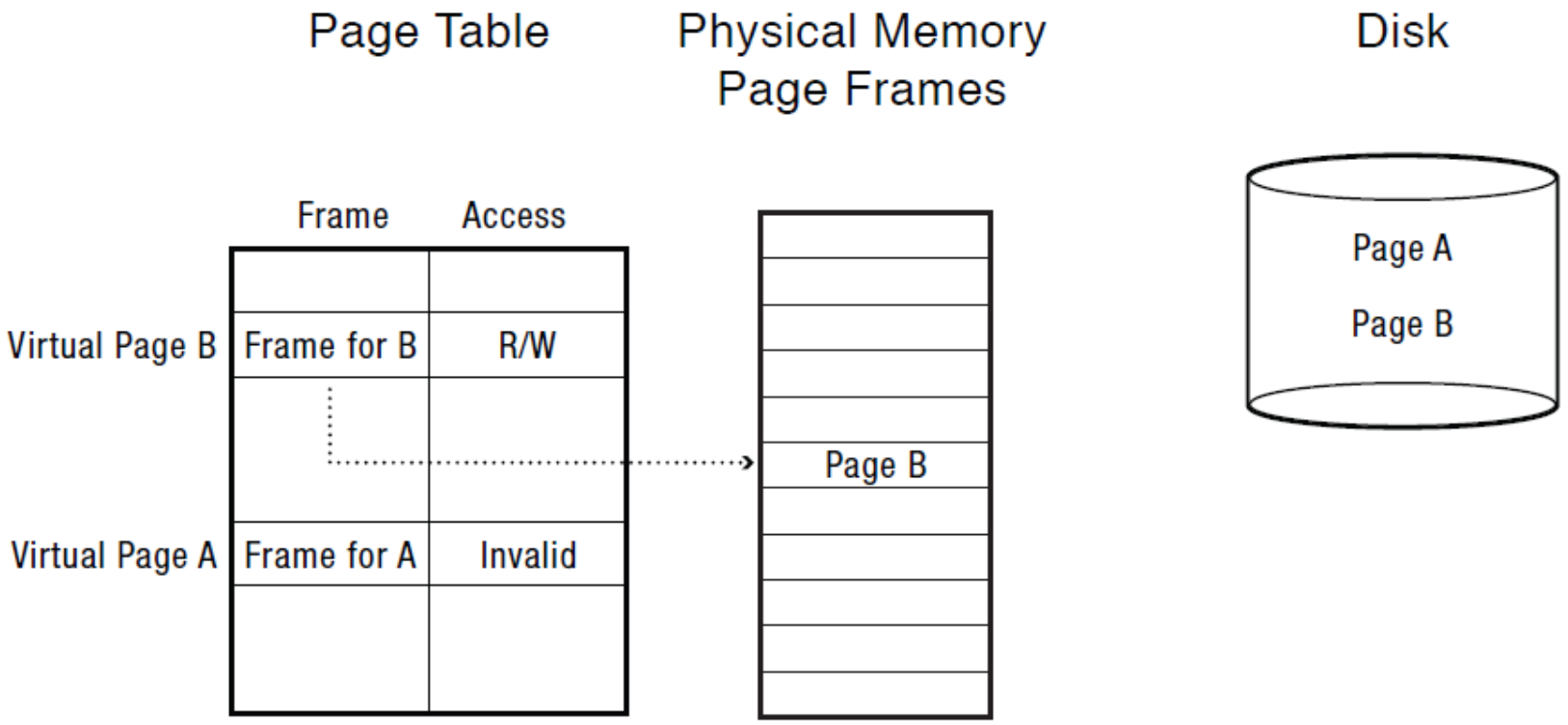
Hardware address translation is a power tool

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

Demand Paging (Before)



Demand Paging (After)



Demand Paging – quick walk

One page table per process!

1. TLB miss
2. Page table walk
3. Page fault (page invalid in page table)
4. Trap to kernel
5. Convert virtual address to file + offset
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

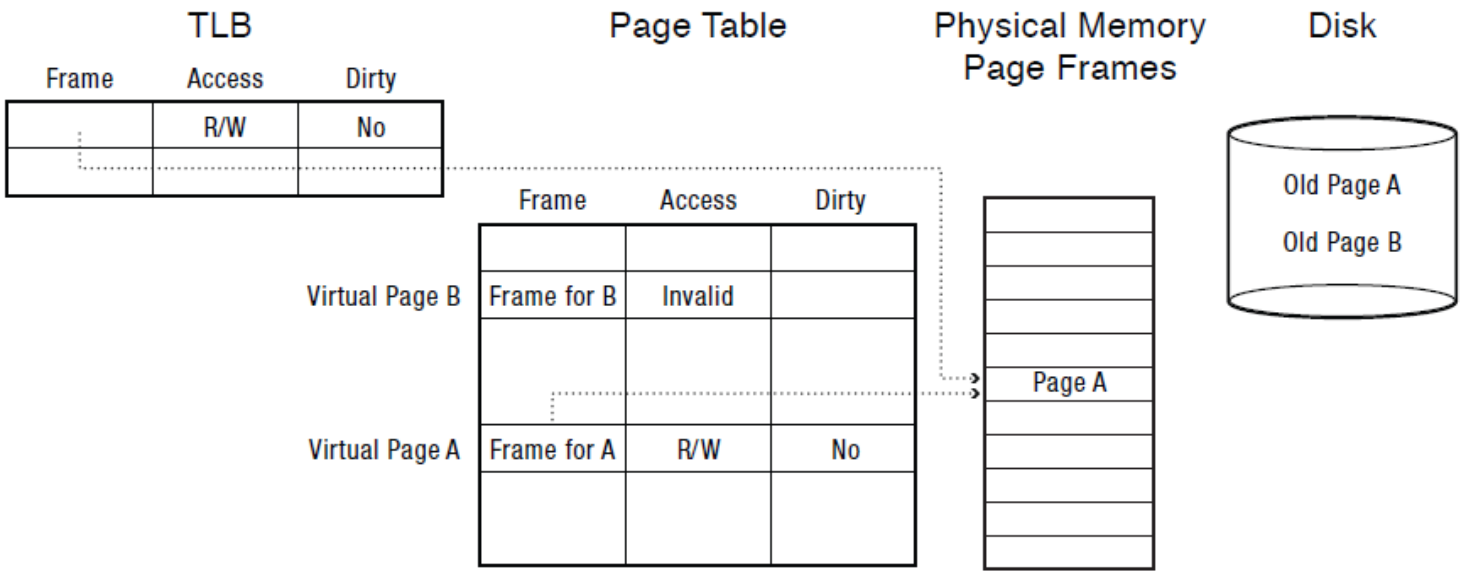
Allocating a Page Frame

- Select old page to evict – which one?
- Find all page table entries that refer to old page
  - If page frame is shared
- Set each page table entry to invalid
- Remove any TLB entries
  - Copies of now invalid page table entry
- Write changes on page back to disk, if necessary

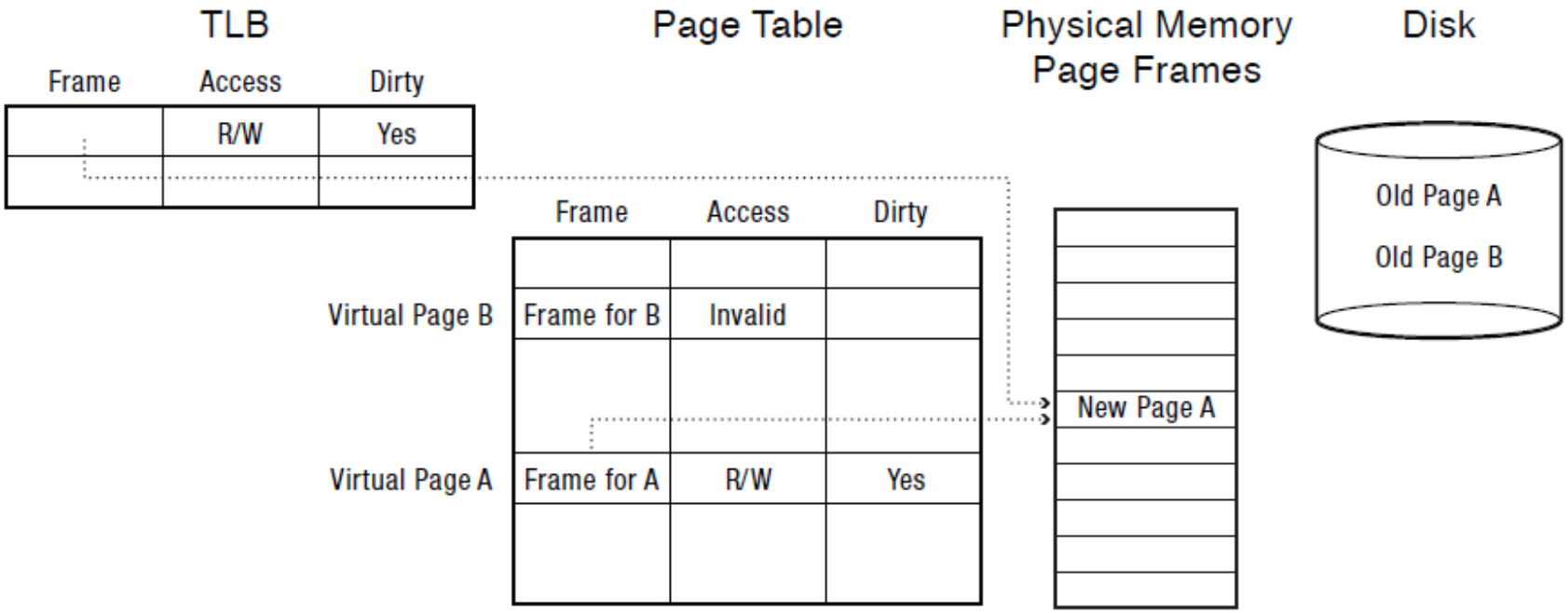
How do we know if page has been modified?

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

Keeping Track of Page Modifications (Before)



Keeping Track of Page Modifications (After)



Virtual or Physical Dirty/Use Bits

- Most machines keep dirty/use bits in the page table entry
- 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

Tidbit: Emulating a Modified Bit

- Some processor archs. 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 sets modified bit, marks page as read-write
  - Resume execution
- Kernel needs to keep track of both
  - Current page table permission (e.g., read-only)
  - True page table permission (e.g., writeable)
- Can also emulate a recently used bit

Memory-Mapped Files

- Explicit read/write system calls for files
  - 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 instruction
  - `mmap` in Linux

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 (automatically)
- Interprocess communication
  - Shared memory segment vs. temporary file

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

Memory is a Cache for Disk: 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

A Simple Policy

- Random?
  - Replace a random entry
- FIFO?
  - Replace the entry that has been in the cache the longest time
  - What could go wrong?

FIFO in Action

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

Lab #2

- Lab #1 was more about mechanism
  - How to implement a specific features
- Lab #2 is more about policy
  - Given a mechanism, how to use it

Caching and Demand-Paged Virtual Memory

Chapter 9 OSPP

MIN

- 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
  - Can we know the future?
  - Maybe: compiler might be able to help.

LRU, LFU

- Least Recently Used (LRU)
  - Replace the cache entry that has not been used for the longest time in the past
  - Approximation of MIN
  - Past predicts the future: code?
- Least Frequently Used (LFU)
  - Replace the cache entry used the least often (in the recent past)

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		+		+					

Belady's Anomaly

FIFO (3 slots)												
Reference	A	B	C	D	A	B	E	A	B	C	D	E
1	A			D			E					+
2		B			A			+		C		
3			C			B			+		D	
FIFO (4 slots)												
1	A				+		E				D	
2		B				+		A				E
3			C						B			
4				D						C		

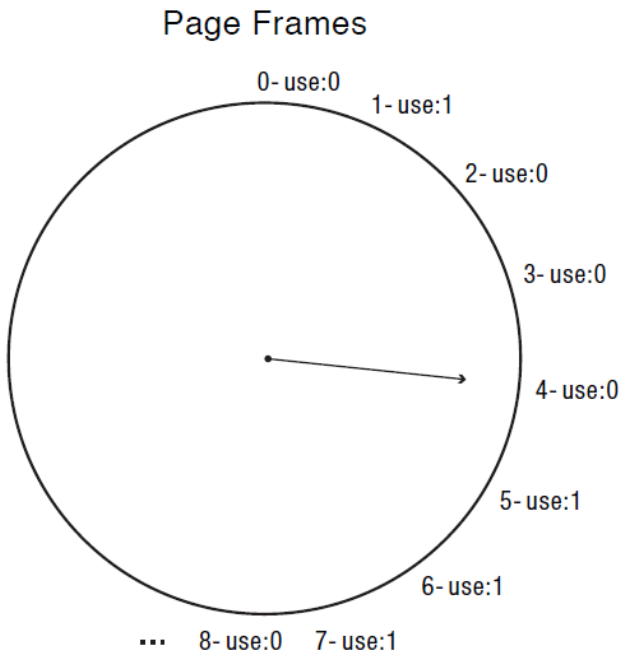
More memory does worse! LRU does not suffer from this.

True LRU

- Hard to do in practice: why?

Clock Algorithm: Estimating LRU

- Periodically, sweep through all/some pages
- If page is unused, reclaim (no chance)
- If page is used, mark as unused
- remember clock hand for next time



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) {
    notInUseSince = 0;
} else if (notInUseSince < N) {
    notInUseSince++;
} else {
     reclaim page;
}
```

Paging Daemon

- Periodically run some version of clock/Nth chance: background
- Goal to keep # of free frames > %
- Clean (write-back) and free frames as needed

Recap

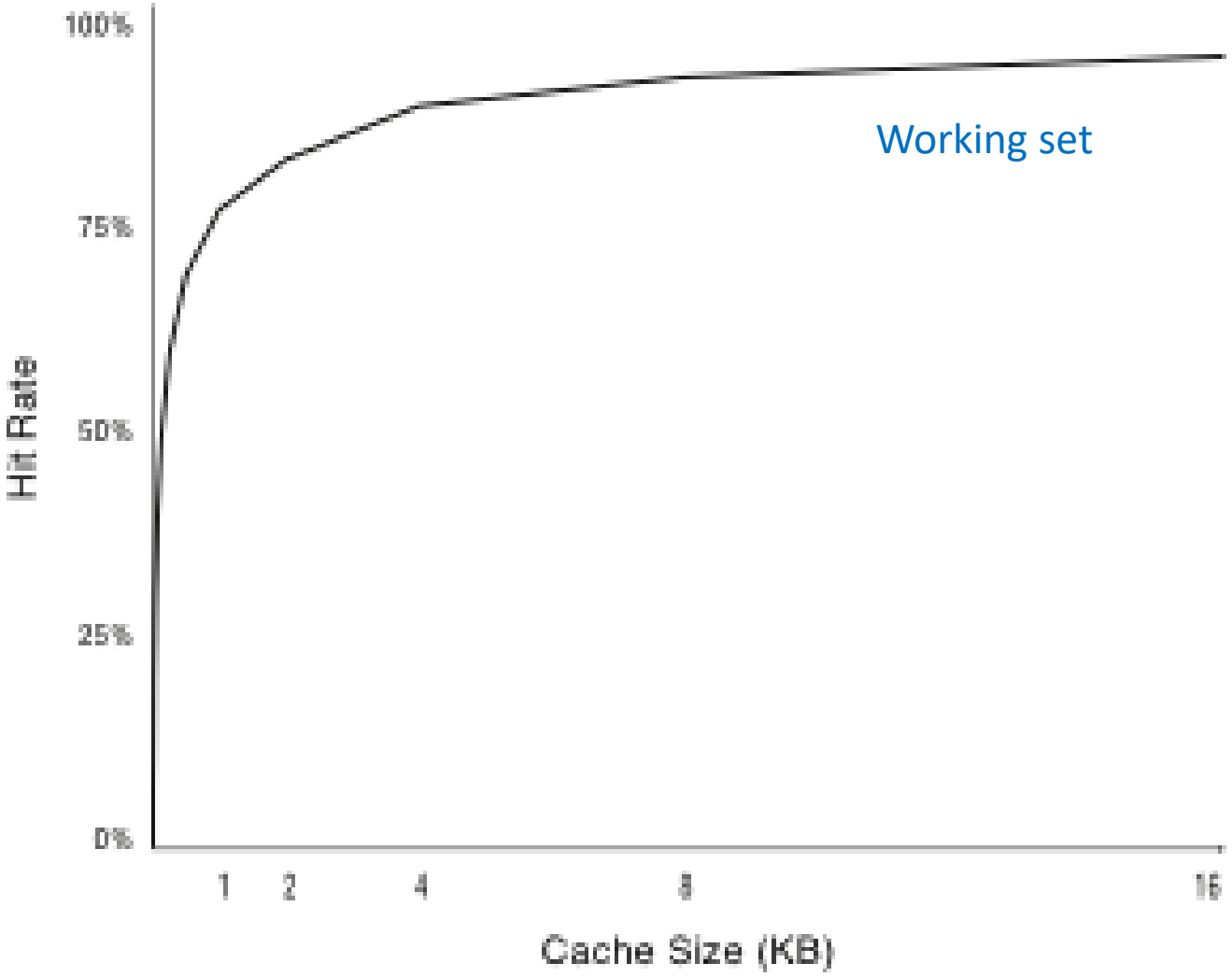
- MIN is optimal
  - replace the page or cache entry that will be used farthest into the future
- LRU is an approximation of MIN
  - For programs that exhibit spatial and temporal locality
- Clock/Nth Chance is an approximation of LRU
  - Bin pages into sets of “not recently used”

Working Set Model

- Working Set (WS): set of memory locations that need to be cached for reasonable cache hit rate
  - top: RES(ident) field (~ WS)
  - Driven by locality
  - Programs get whatever they need (to a point)
  - Pages accessed in last τ time or *k* accesses
  - Uses some version of clock (conceptually): min-max WS

- Thrashing: when cache (i.e. memory) is too small
  - Σ of WS_i > Memory for all i running processes

Cache Working Set



Memory Hogs

- How many pages to give each process?
- Ideally their working set
- But a hog or rogue can steal pages
  - For global page stealing, thrashing can cascade
- Solution: self-page
  - Problem?
  - Local solutions (e.g. multiple queues) are suboptimal

Sparse Address Spaces

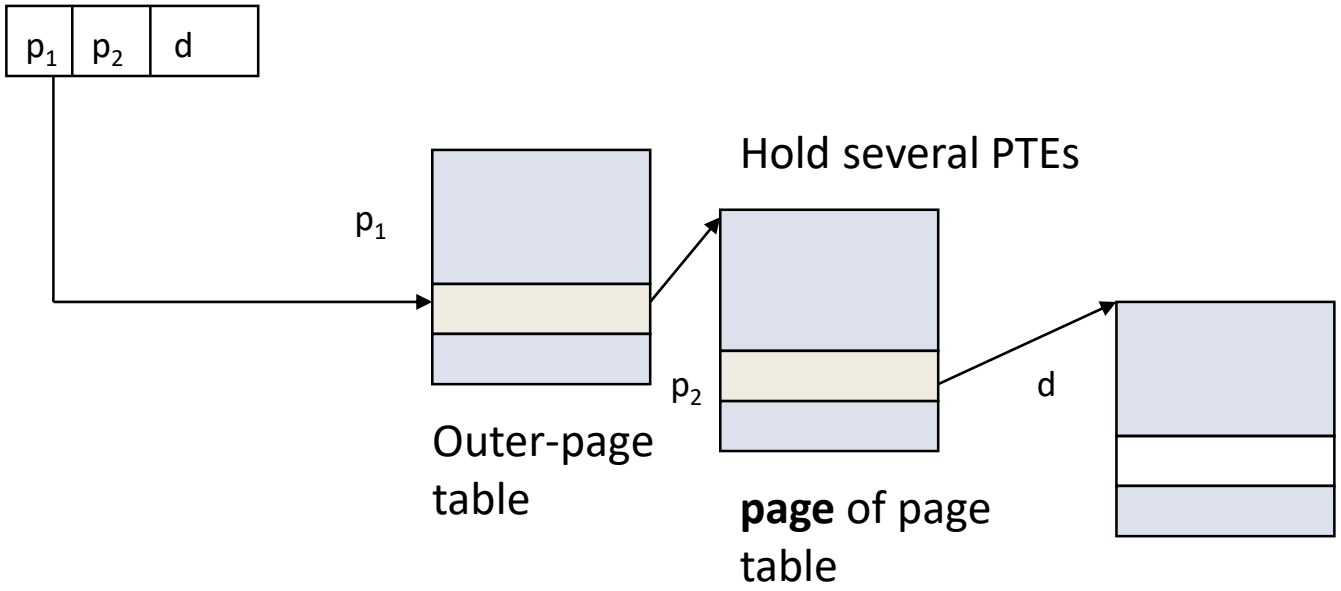
- What if virtual address space is large?
  - 32-bits, 4KB pages => 500K page table entries
  - 64-bits => 4 quadrillion page table entries
  - Famous quote:
  - “Any programming problem can be solved by adding a level of indirection”
- Today’s OS allocate page tables on the fly, even on the backing store!
  - Allocate/fill only page table entries that are in use
  - STILL, can be really big

Multi-level Translation

- Tree of translation tables
  - Multi-level page tables
  - Paged segmentation
  - Multi-level paged segmentation
- Stress: hardware is doing the translation!
- Page the page table or the segments! … or both

Address-Translation Scheme

- Address-translation scheme for a two-level 32-bit paging architecture

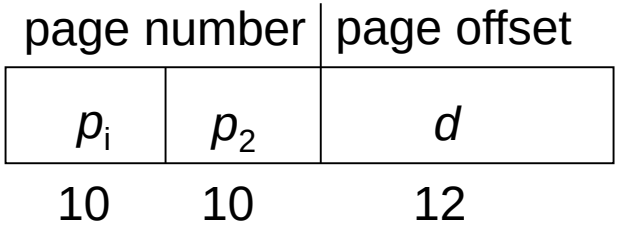


<board>

Two-Level Paging Example

- A VA on a 32-bit machine with 4K page size is divided into:
  - a page number consisting of 20 bits
  - a page offset consisting of 12 bits (set by hardware/OS)
  - assume trivial PTE of 4 bytes (just frame #)

- Since the page table is paged, the page number is further divided into:
  - a 10-bit page number
  - a 10-bit page offset (to each PTE)
- Thus, a VA is as follows:



- where p_i is an index into the outer page table, and p_2 is the displacement within the page of the outer page table (i.e the PTE entry).

Multi-level Page Tables

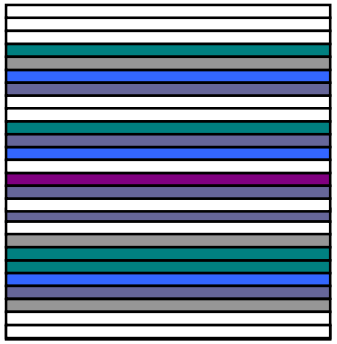
- How big should the outer-page table be?

  Size of the page table for process (PTE is 4): $2^{20}x4=2^{22}$

  Page this (divide by page size): $2^{22}/2^{12} = 2^{10}$

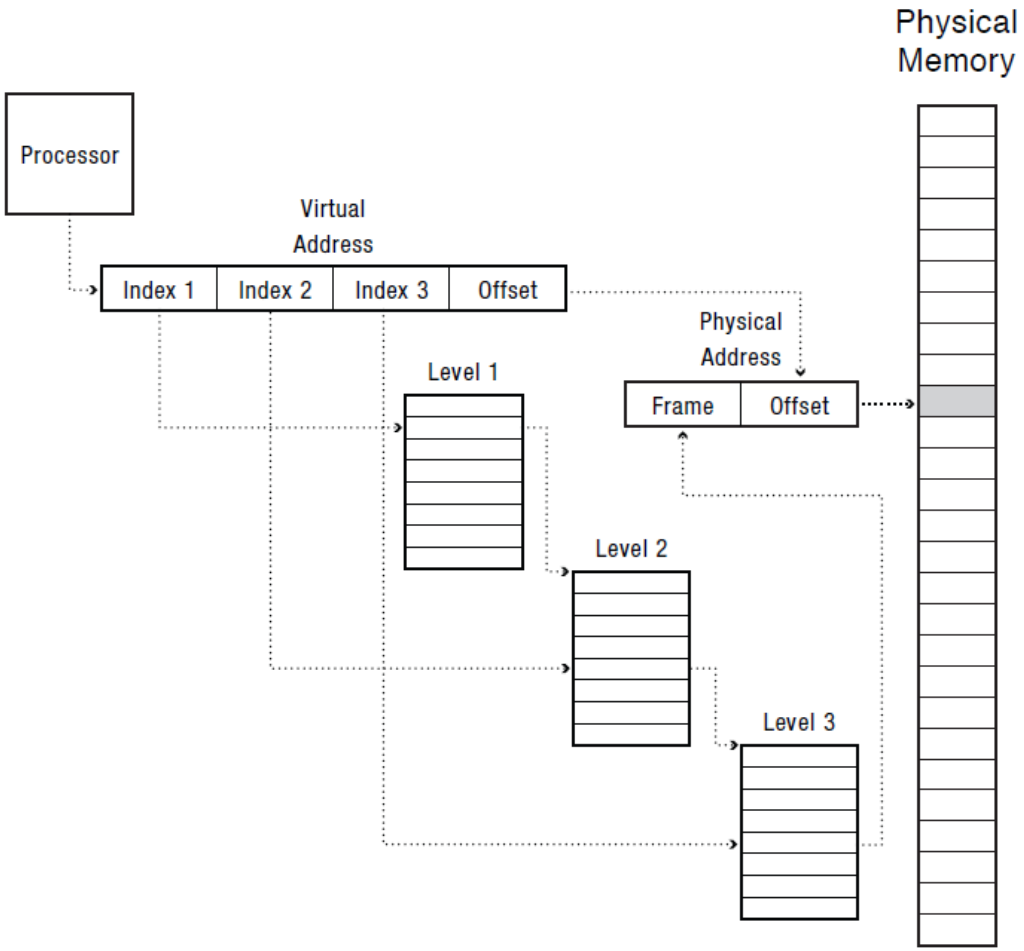
  Answer: 2^{10} x4=2^{12}

- How big is the virtual address space now?
- Have we reduced the amount of memory required for paging?



Multilevel Paging

- Can keep paging!



Multilevel Paging and Performance

- Can take 3 memory accesses (if TLB miss)
- Suppose TLB access time is 20 ns, 100 ns to memory
- Cache hit rate of 98 percent yields:

  effective access time = 0.98 x 120 + 0.02 x 320 = 124 nanoseconds
  24% slowdown

- Can add more page tables and can show that slowdown grows slowly:

  3-level: 26 %

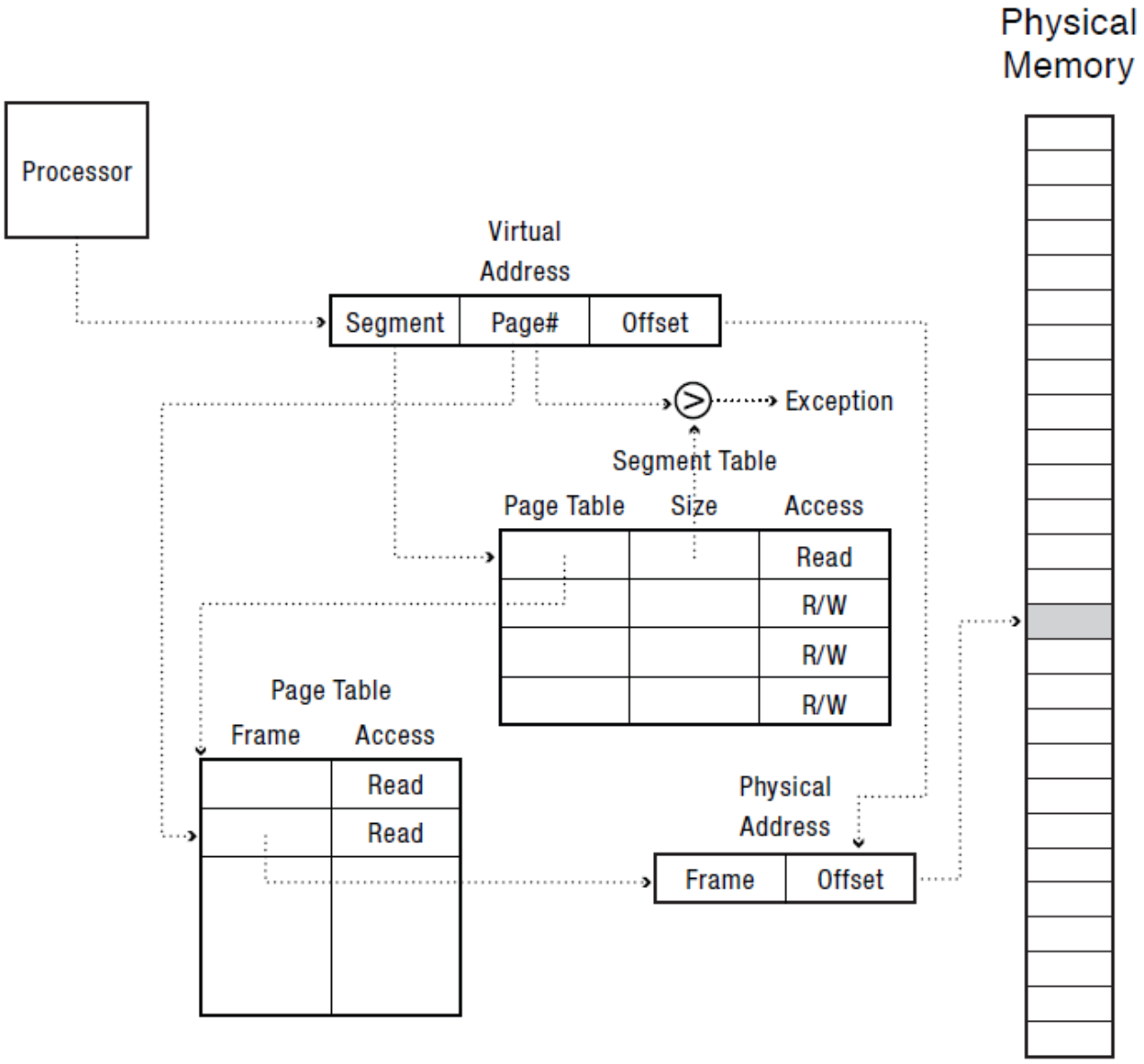
  4-level: 28%

- Q: why would I want to do this!

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

Paged Segmentation (Implementation)



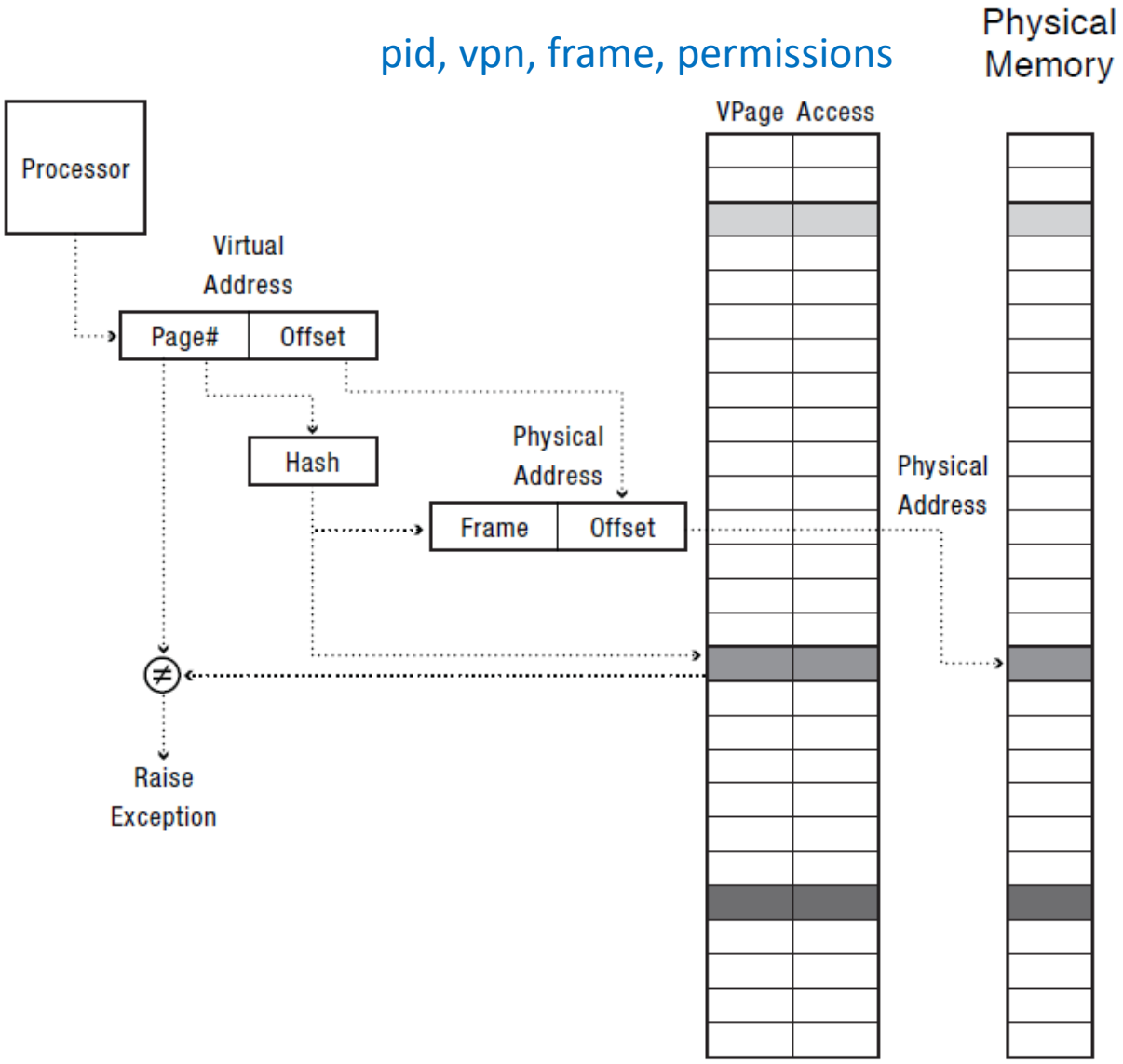
Multilevel Translation

- Pros:
  - Simple and flexible memory allocation (i.e. pages)
  - Share at segment or page level
  - Reduced fragmentation
- Cons:
  - Space overhead: extra pointers
  - Two (or more) lookups per memory reference, but TLB

Portability

- Many operating systems keep their own memory translation data structures for portability, e.g.
  - List of memory objects (segments), e.g. fill-from location
  - Virtual page -> physical page frame (shadow page table)
    - Different from h/w: extra bits (C-on-Write, Z-on-Ref, clock bits)
  - Physical page frame -> set of virtual pages
    - Why?
- Inverted page table : replace all page tables; solve
  - Hash from virtual page -> physical page
  - Space proportional to # of physical frames – sort of

Inverted Page Table



Address Translation

Chapter 8 OSPP

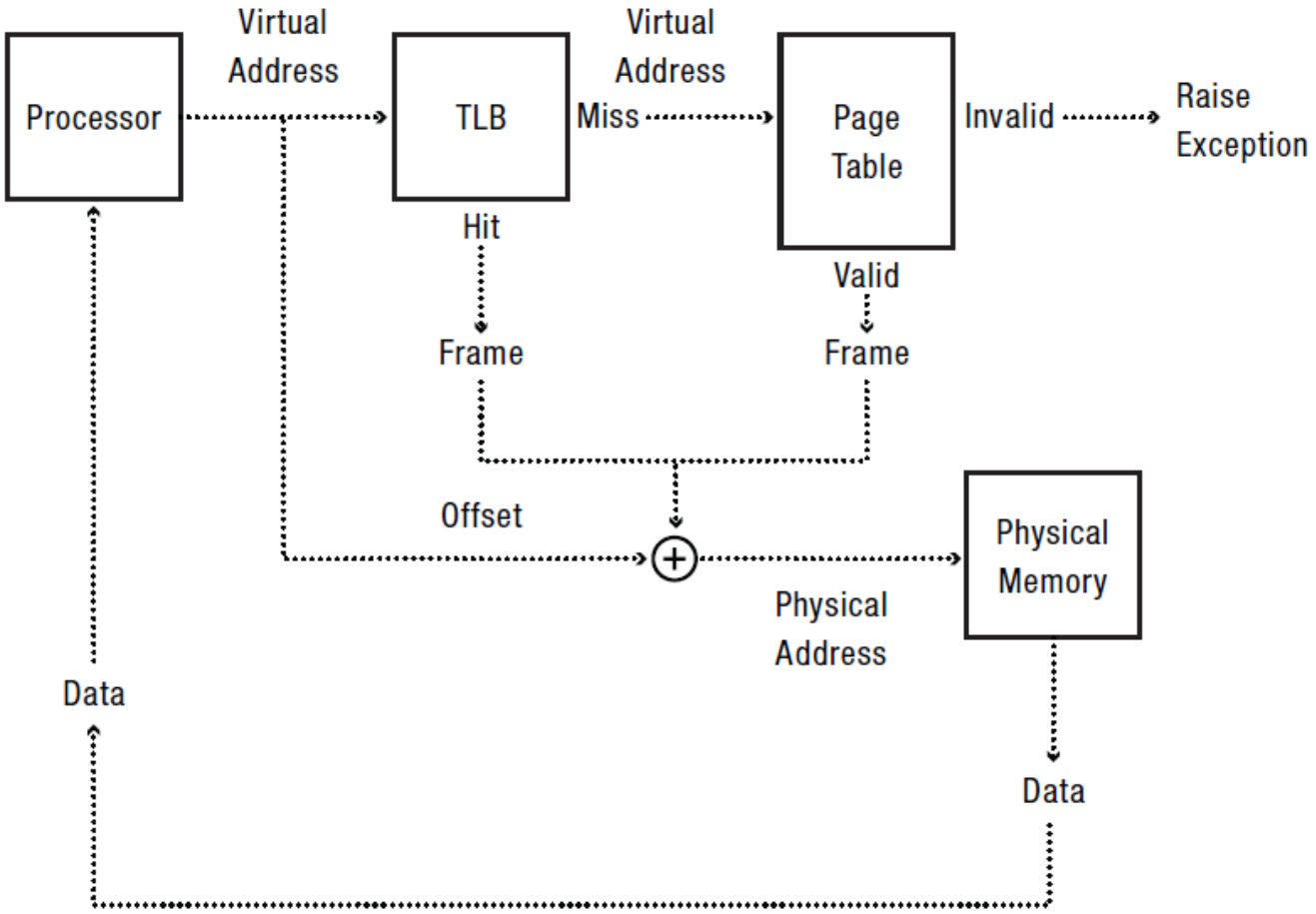
Advanced, Memory Hog paper

Back to TLBs

Pr(TLB hit) * cost of TLB lookup +

Pr(TLB miss) * cost of page table lookup

TLB and Page Table Translation



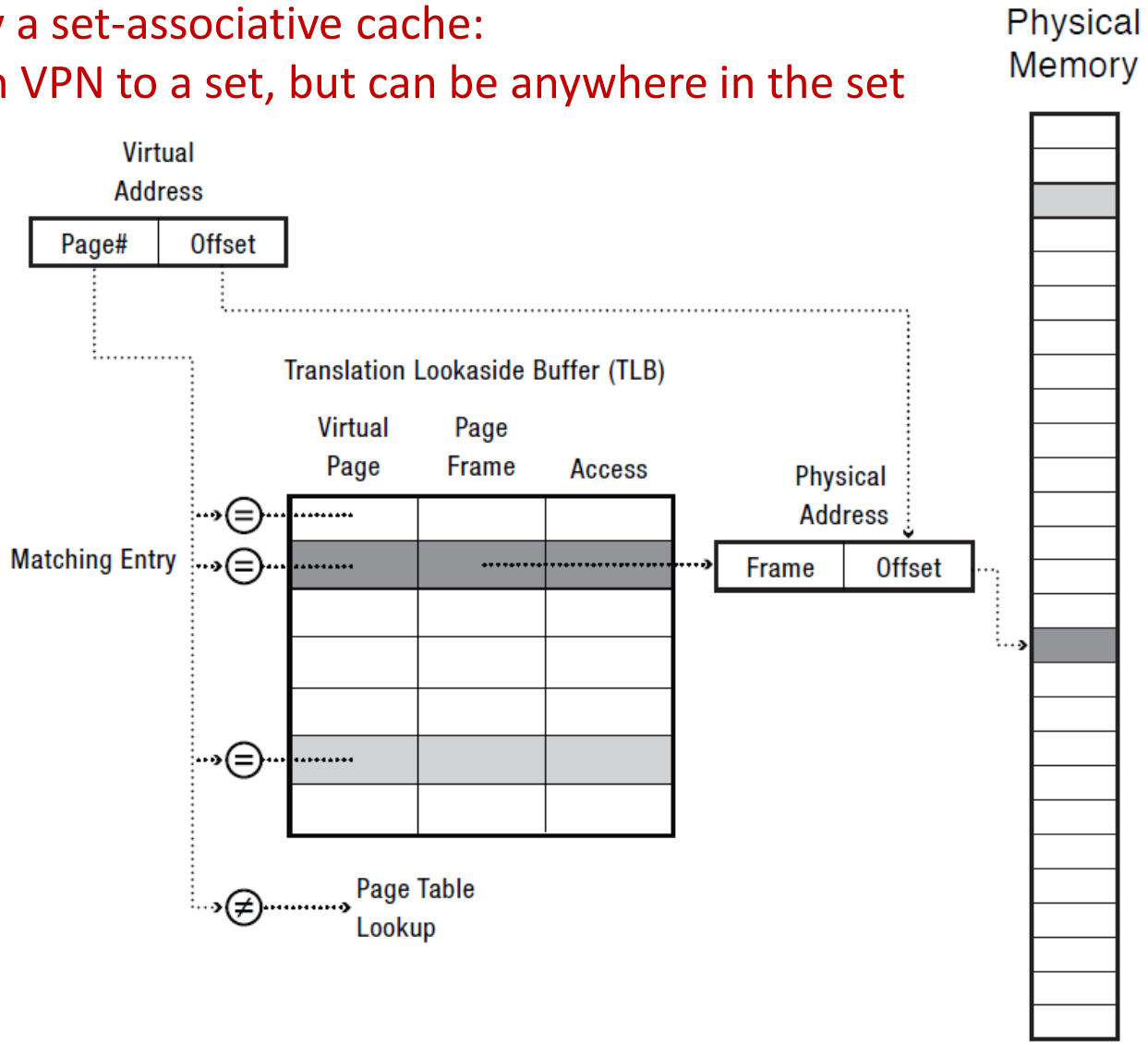
TLB Miss

- Done all in hardware

- Or in software (software-loaded TLB)
  - Since TLB miss is rare …
  - Trap to the OS on TLB miss
  - Let OS do the lookup and insert into the TLB
  - A little slower … but simpler hardware

TLB Lookup

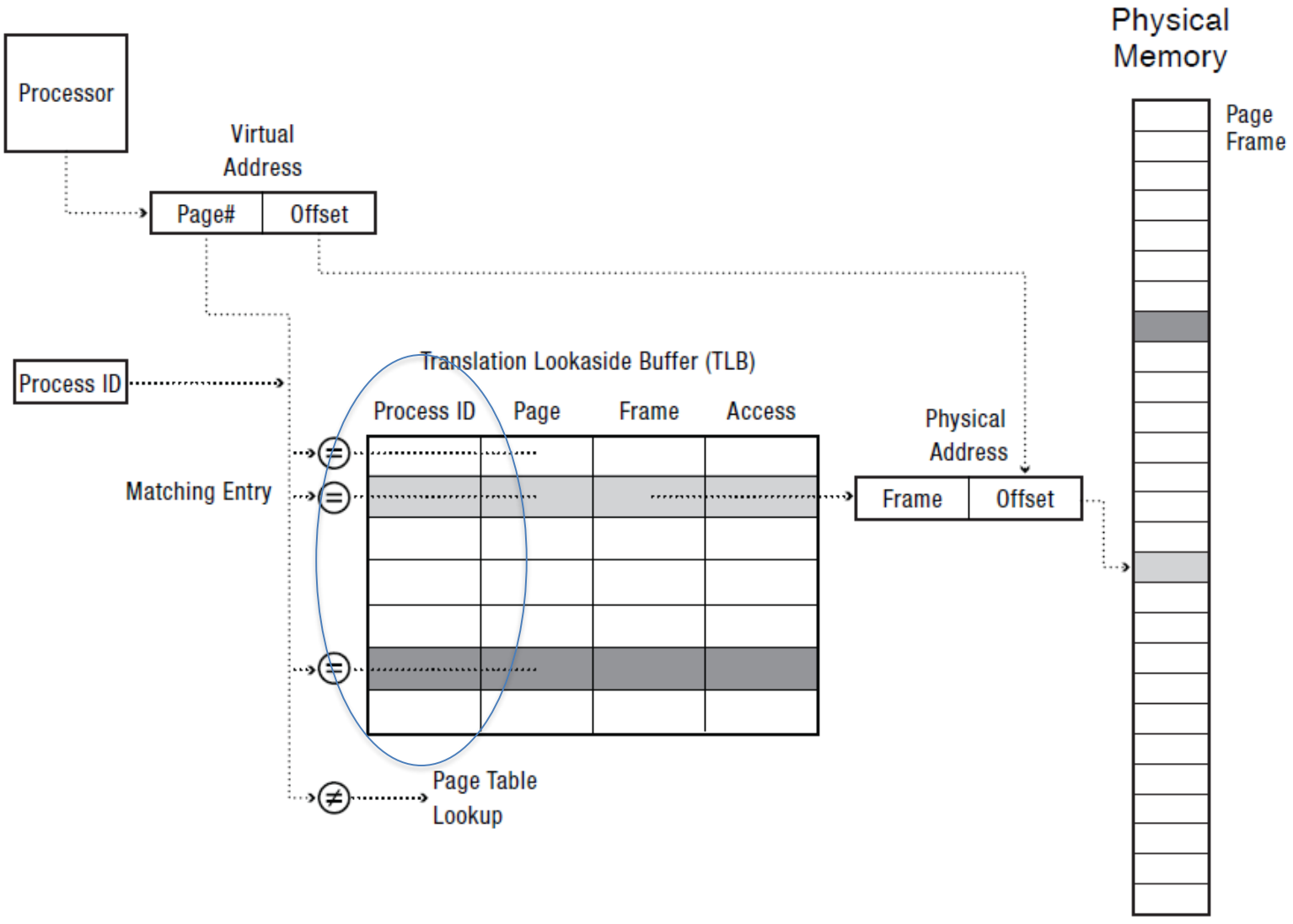
TLB usually a set-associative cache:
Direct hash VPN to a set, but can be anywhere in the set



TLB is critical

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

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

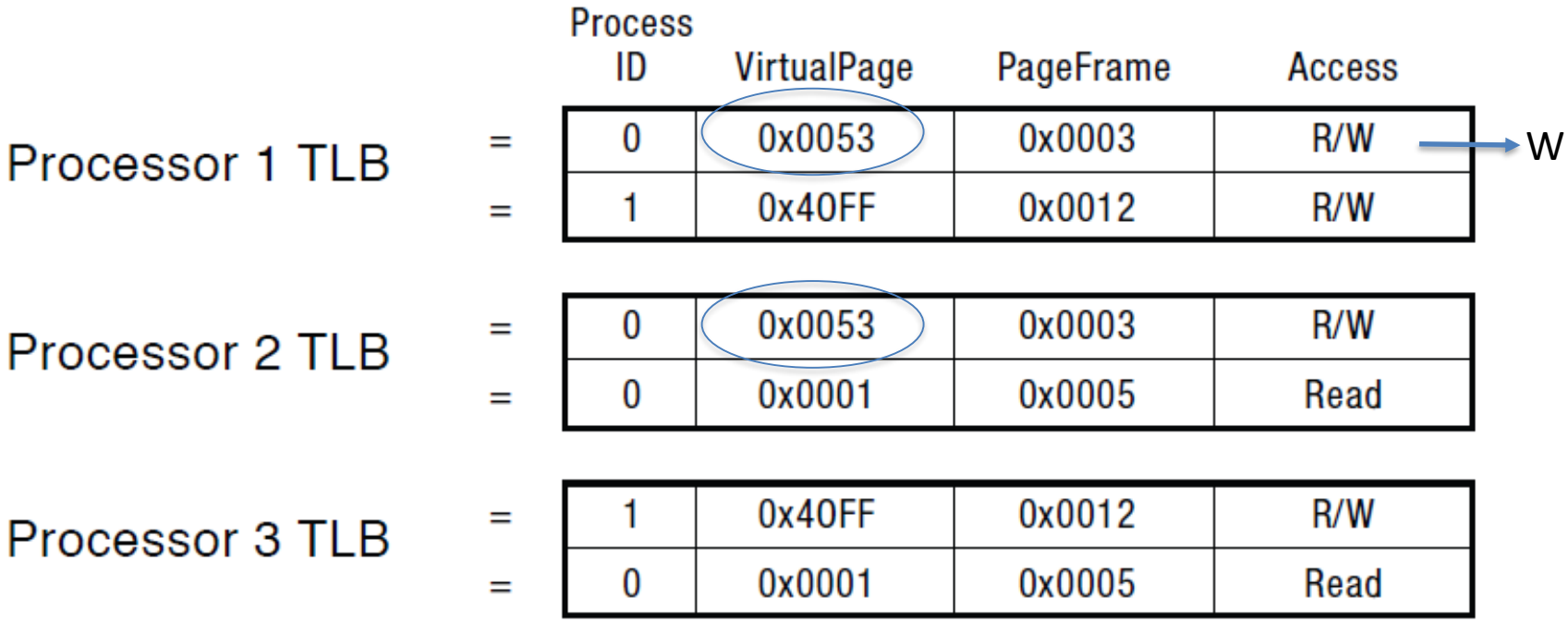


Avoid flushing the TLB on a context-switch

TLB consistency

- What happens when the OS changes the permissions on a page?
  - For demand paging, copy on write, zero on reference, … or is marked invalid!
- TLB may contain old translation or permissions
  - OS must ask hardware to purge TLB entry
- On a multicore: TLB shootdown
  - OS must ask each CPU to purge TLB entry
  - Similar to above

TLB Shootdown

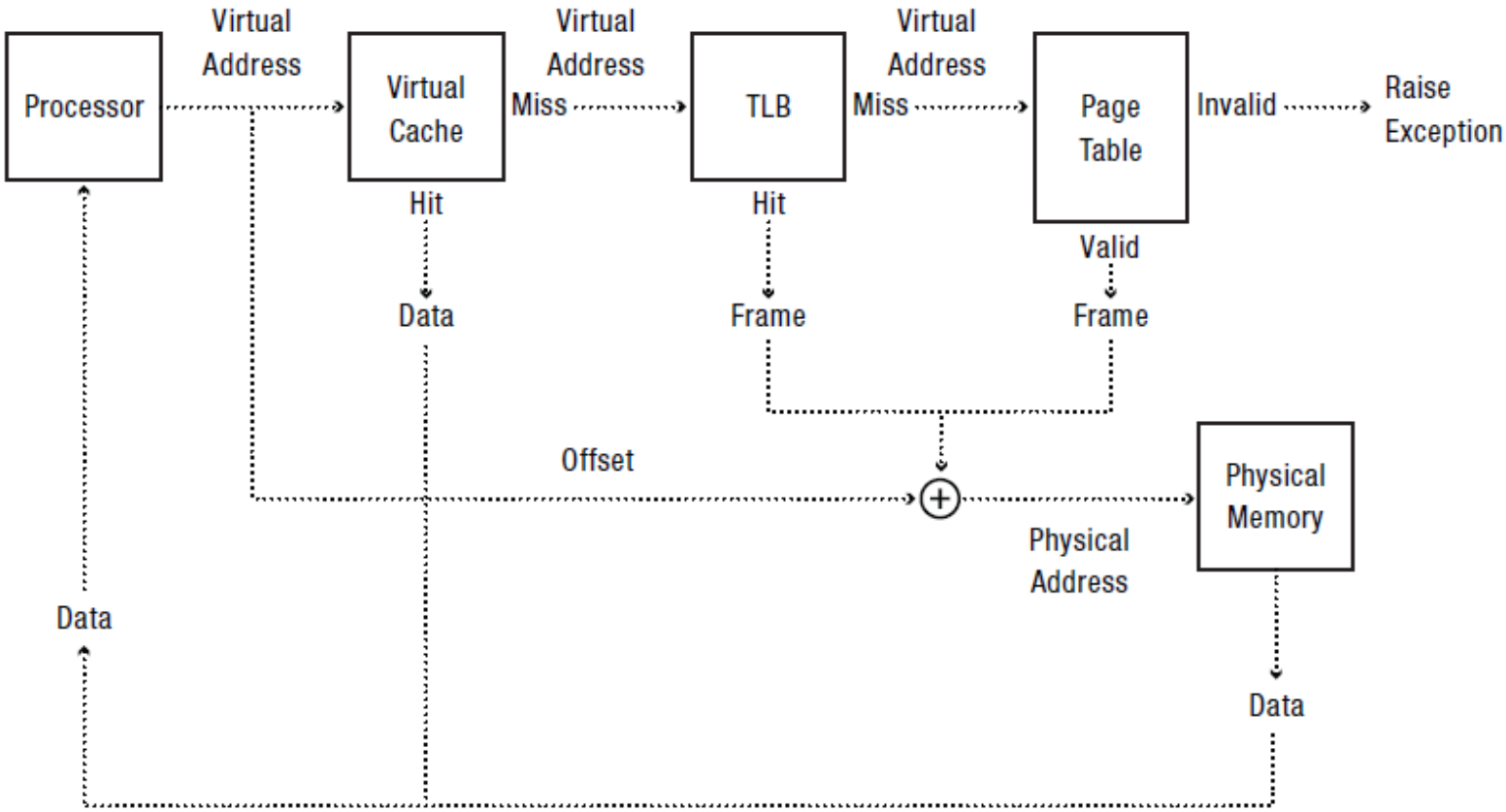


TLB Optimizations

Virtually Addressed vs. Physically Addressed Data Caches

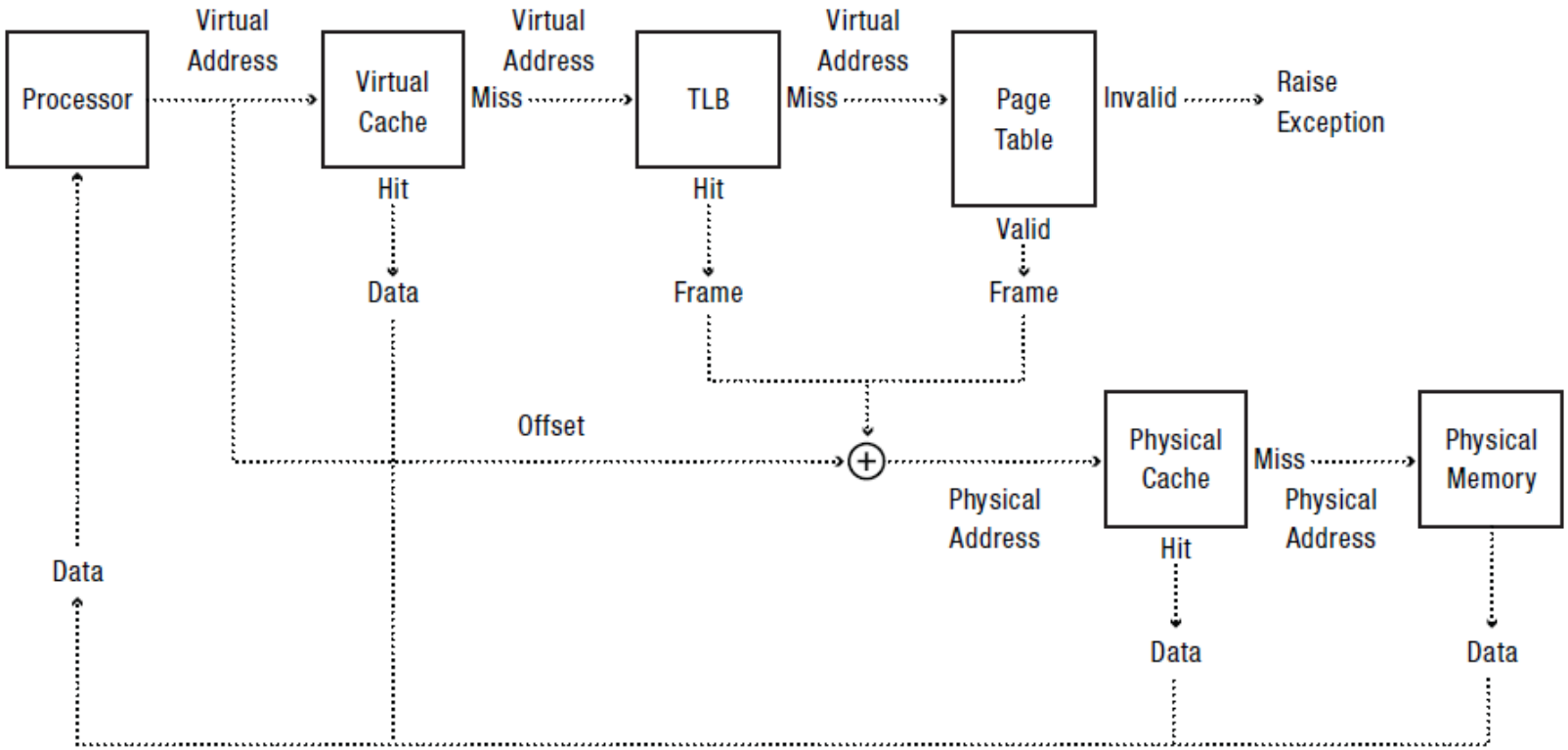
- How about we cache data!
- Too slow to first access TLB to find physical address … particularly for a TLB miss
  - VA -> PA -> data
  - VA -> data
- Instead, first level cache is virtually addressed
- In parallel, access TLB to generate physical address (PA) in case of a cache miss
  - VA -> PA -> data

Virtually Addressed Caches



Same issues w/r to context-switches and consistency

Physically Addressed Cache



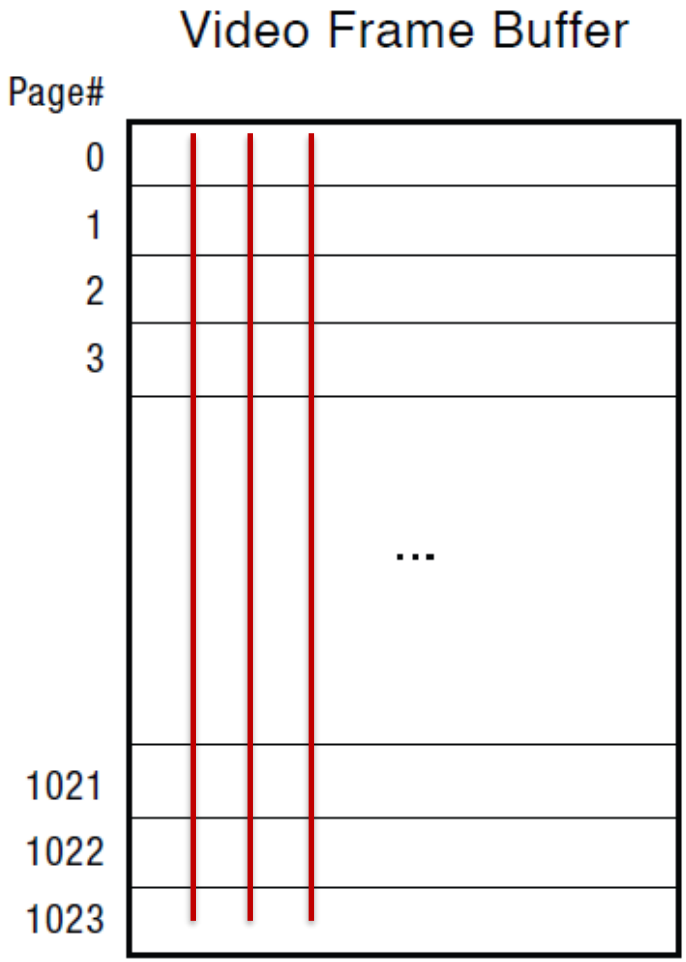
Cache physical translations: at any level! (e.g. frame->data)

Superpages

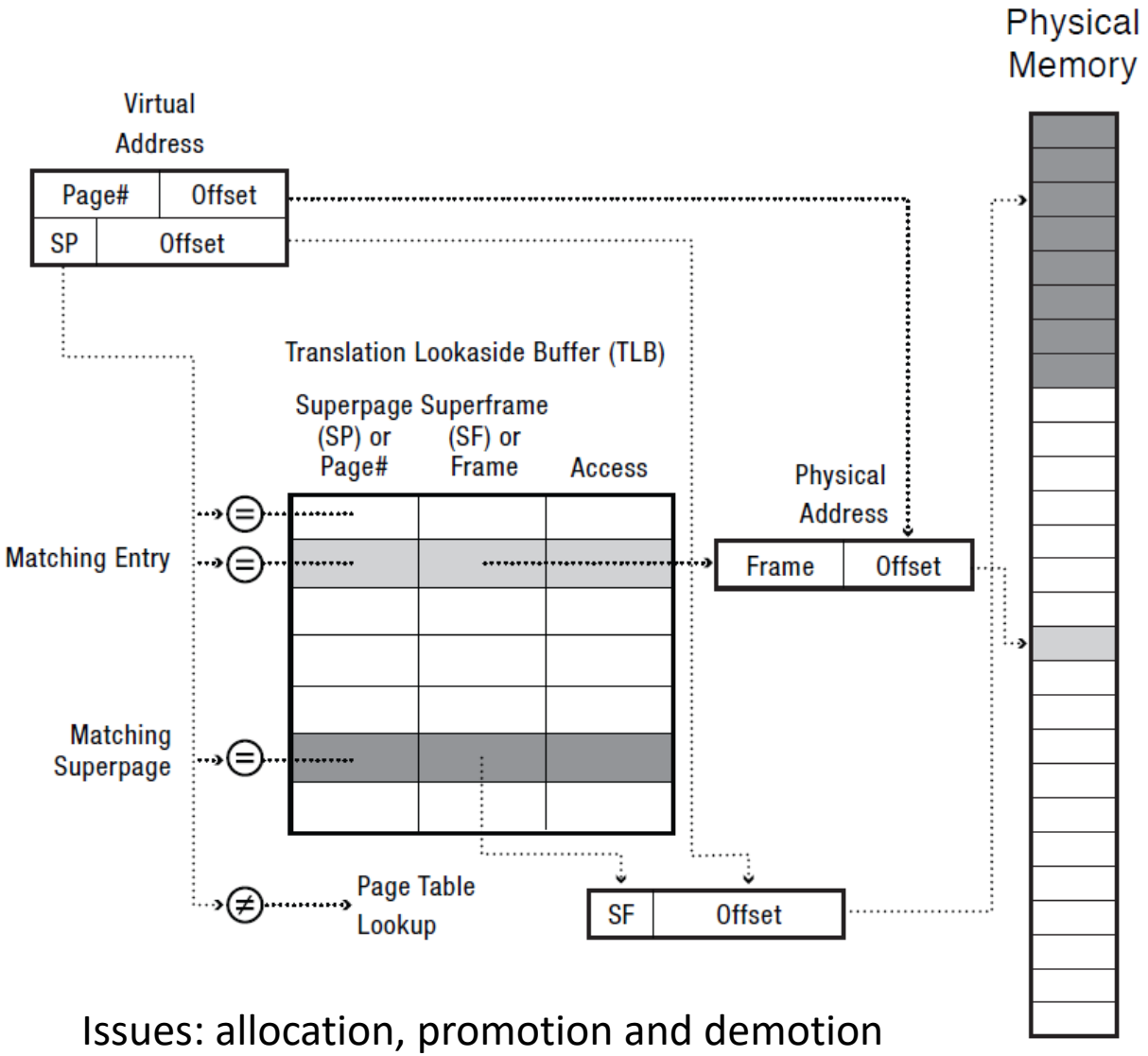
- On many systems, TLB entry can be
  - A page
  - A superpage: a set of contiguous pages
- x86: superpage is set of pages in one page table
  - superpage is memory contiguous
  - x86 also supports a variety of page sizes, OS can choose
    - 4KB
    - 2MB
    - 1GB

Walk an Entire Chunk of Memory

- Video Frame Buffer:
  - 32 bits x 1K x 1K = 4MB
- Very large working set!
  - Draw a ~~horizontal~~ vertical line
  - Lots of TLB misses
- Superpage can reduce this
  - 4MB page



Superpages



Issues: allocation, promotion and demotion

Overview

- Huge data sets => memory hogs
  - Insufficient RAM
  - “out-of-core” applications > physical memory
  - E.g. scientific visualization
- Virtual memory + paging
  - Resource competition: processes impact each other
  - LRU penalizes interactive processes … why?

The Problem

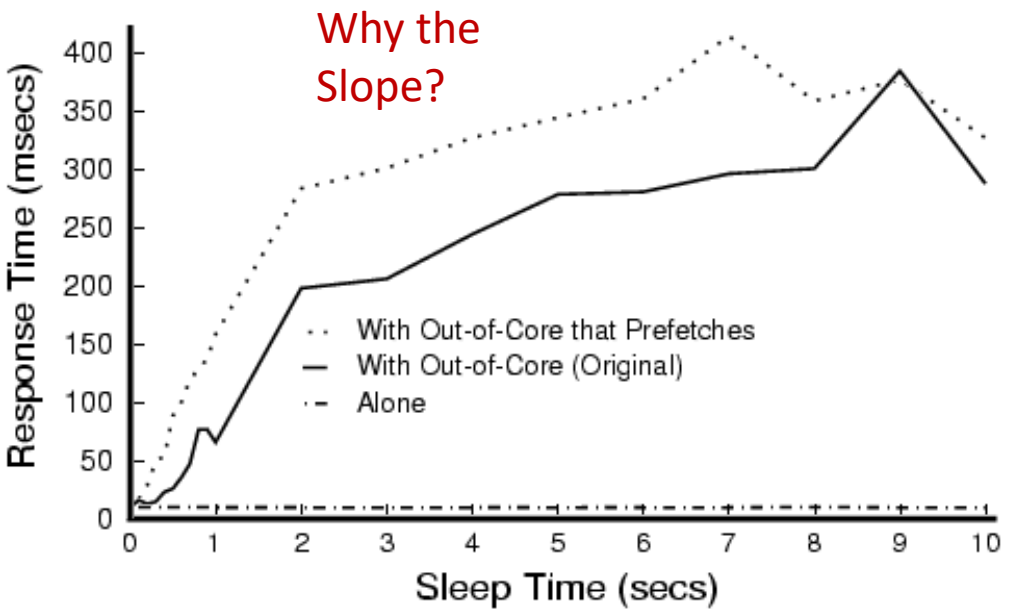


Figure 1. Impact of sharing the machine with an out-of-core matrix-vector multiplication (MATVEC) on the response time of an interactive task across a range of sleep times between touching 1 MB of data.

Page Replacement Options

- Local
  - this would help but very inefficient
  - allocation not according to need
- Global
  - no regard for ownership
  - global LRU ~ clock

Be Smarter

- I/O cost is high for out-of-core apps (I/O waits)
  - Pre-fetch pages before needed: prior work to reduce latency (helps the hog!)
  - Release pages when done (helps everyone!)
- Application may know about its memory use
  - Provide hints to the OS
  - Automate in compiler

Compiler Analysis Example

(a) Source code for averaging nearest-neighbors

```
for (i = 0; i < N; i++)
   for (j = 0; j < N; j++)
      a[i][j] = (a[i+1][j-1] + a[i+1][j]
         + a[i+1][j+1] + a[i][j-1] + a[i][j]
         + a[i][j+1] + a[i-1][j-1] + a[i-1][j]
         + a[i-1][j+1])/9.0;
```

(b) View of data references to the matrix a

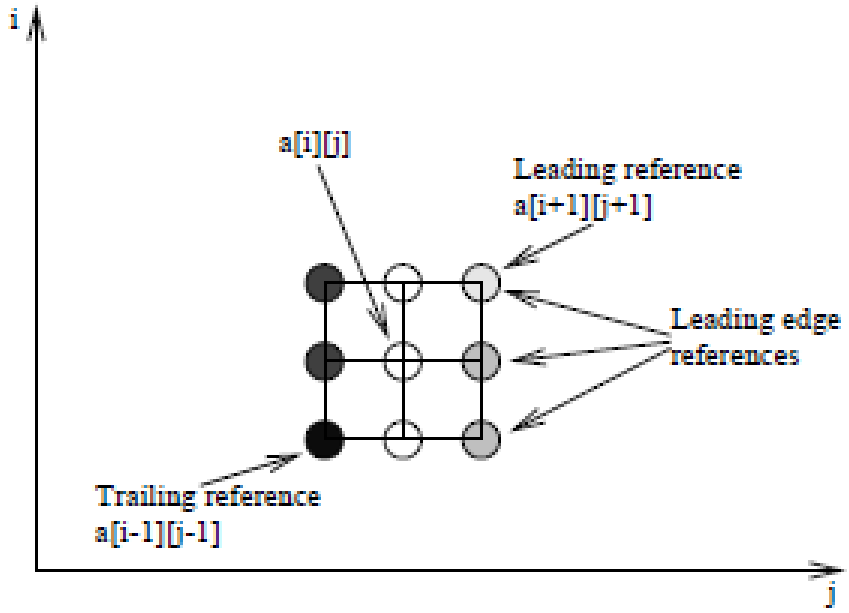
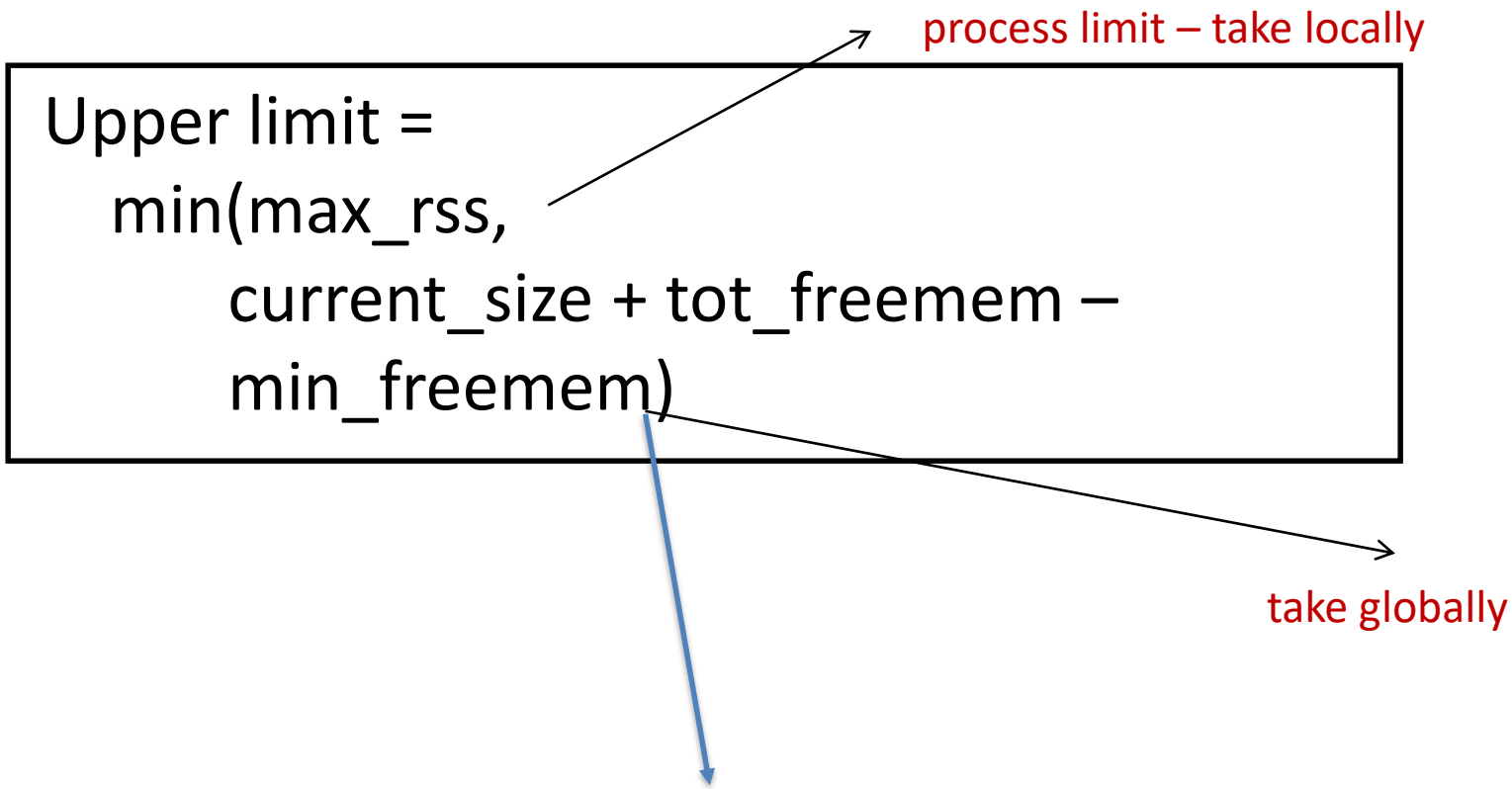


Figure 3. Example source code showing multiple references with different types of reuse, and graphical view of the data accesses during a single iteration of the innermost loop.

OS Support

- Releaser – new system daemon
  - Identify candidate pages for release – how?
  - Prioritized
  - Leave time for rescue
  - Victims: Write back dirty pages

OS Support



Prevent default LRU page cleaning from running

Compiler support

- Infer memory access patterns
- Most useful with arrays with static sizes, nested loops
- Schedule prefetches
- Schedule releases
  - Assign priority

Out-of-core app performance

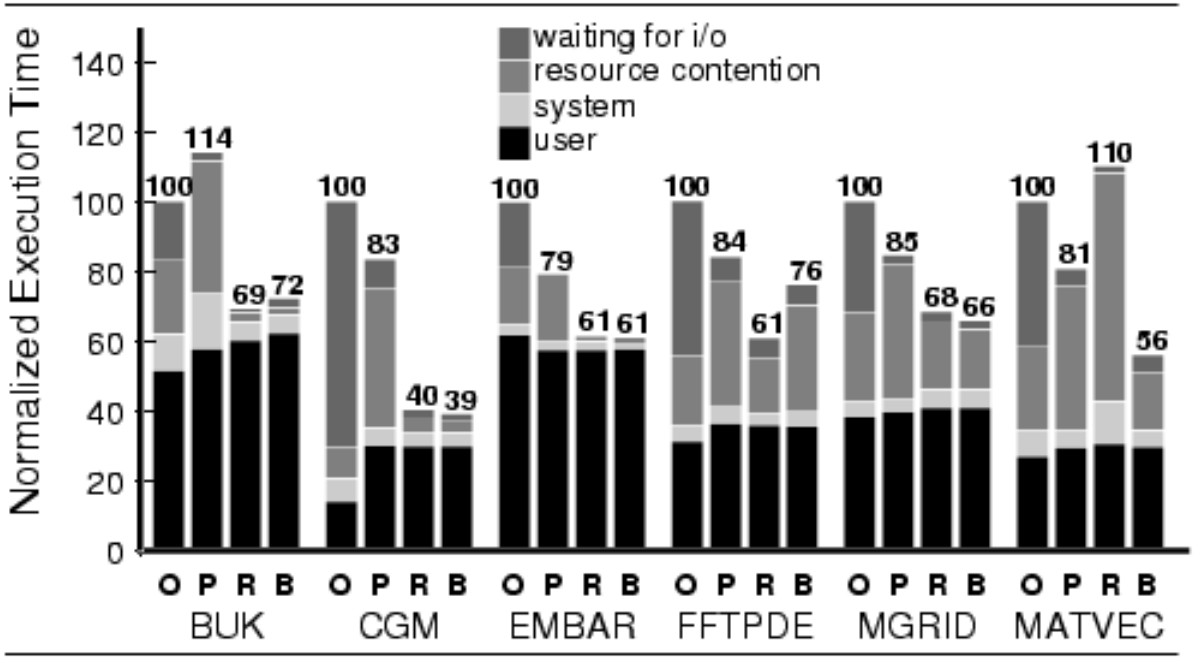


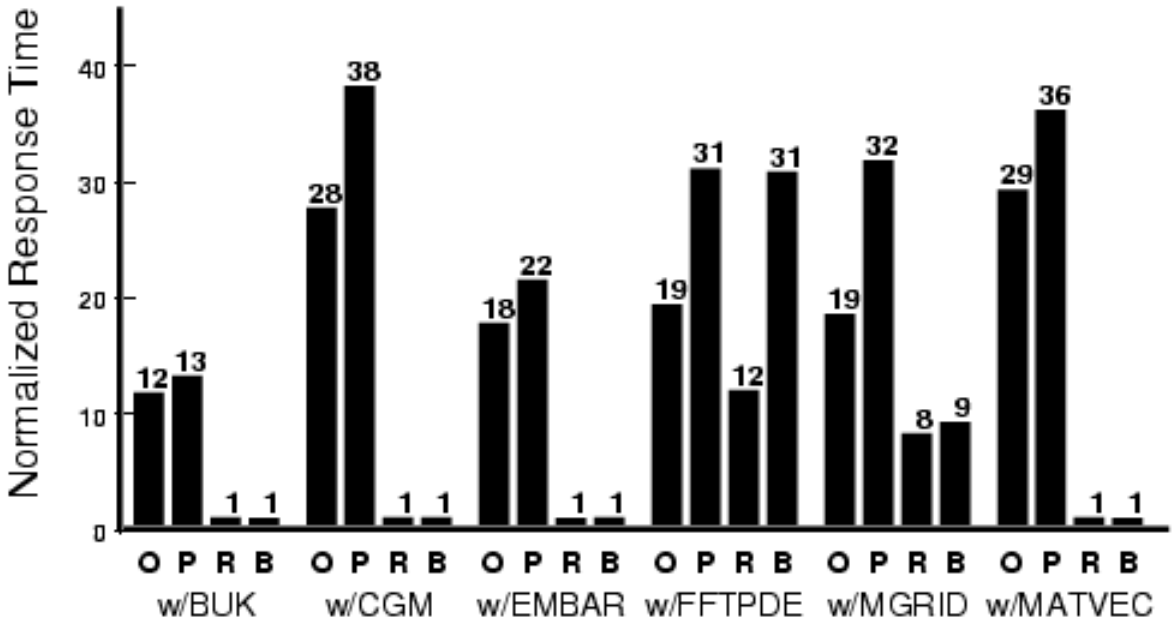
Figure 7. Impact of prefetching and releasing on the execution times of the out-of-core applications. (O = original, P = with prefetching, R = with prefetching and releasing, B = with prefetching and release buffering)

Why does release help out-of-core apps?

smarter page replacement; paging daemon avoided

Impact on interactive apps

(a) Impact of MATVEC on response time (last 3 lines in key overlap).



Conclusion

- Too much data, not enough memory
- Application can help out the OS
- Compiler inserts data pre-fetch and release
- Adaptive run-time system
- Reduces thrashing, improves performance, plays nicely with other apps on the system

Next Time

- File systems
- Chap 11, OSPP

- Have a great weekend!