

Address Translation (part 2)

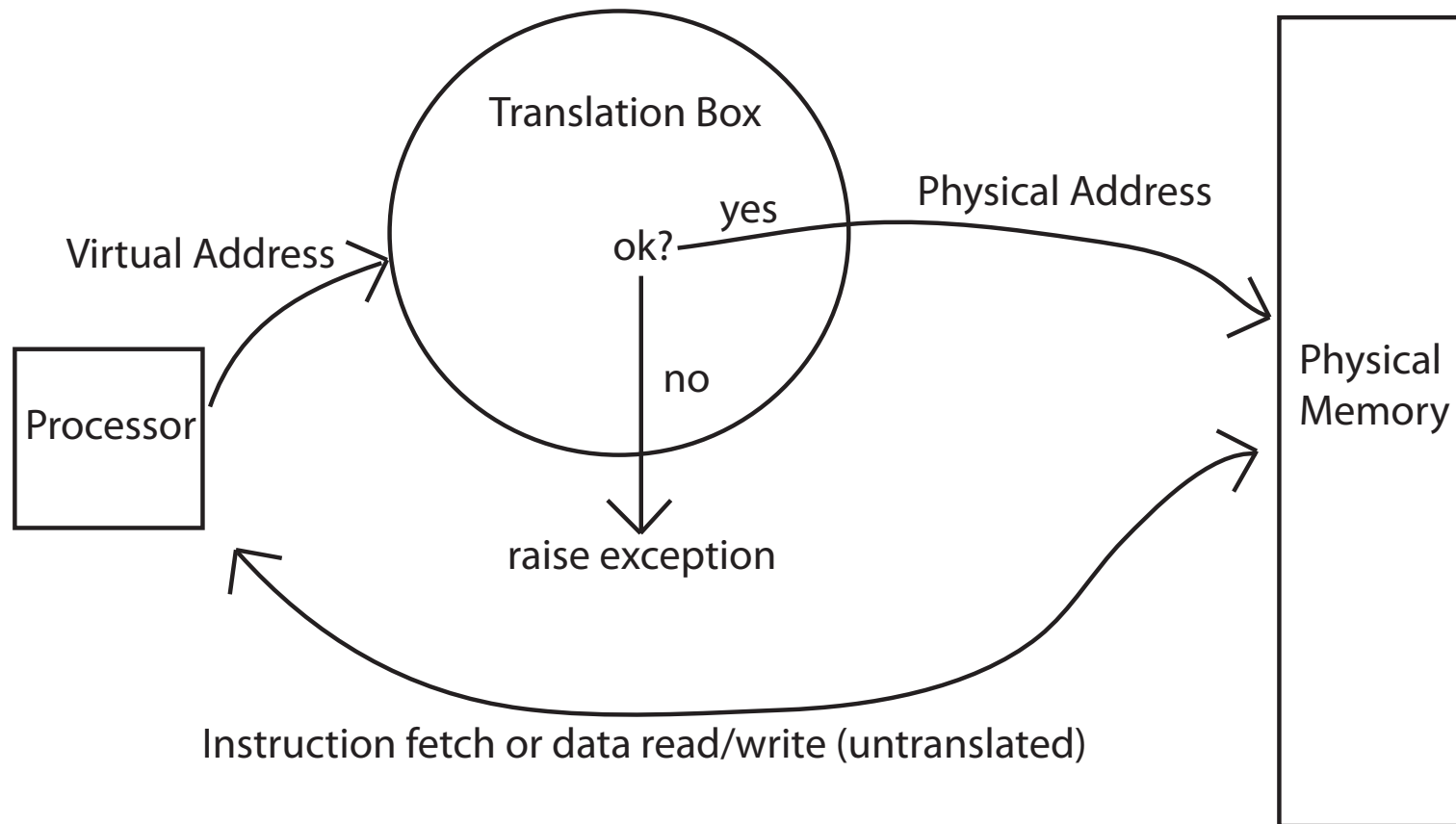
Last Time

- Address Translation Concept
- Flexible Address Translation
 - Base and bound
 - Segmentation
 - Program = multiple contiguous regions of memory
 - Modern programs have many segments
 - Code, data, per core heap, per thread stack, code/data for each separate library
 - Hardware relocation and bound on each reference
- Copy on write
- Zero on reference

Main Points

- Flexible Address Translation
 - Paged Translation
 - Segmentation + paged translation
 - Multi-level paged translation
 - Hashing
- Efficient Address Translation
 - Translation Lookaside Buffers (TLBs)
 - Virtually addressed and physically addressed caches

Address Translation Concept



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 from time to time 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: 00111111000000001100
 - 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

Virtual Address:

| | |
|--------|-------------|
| page # | page offset |
|--------|-------------|

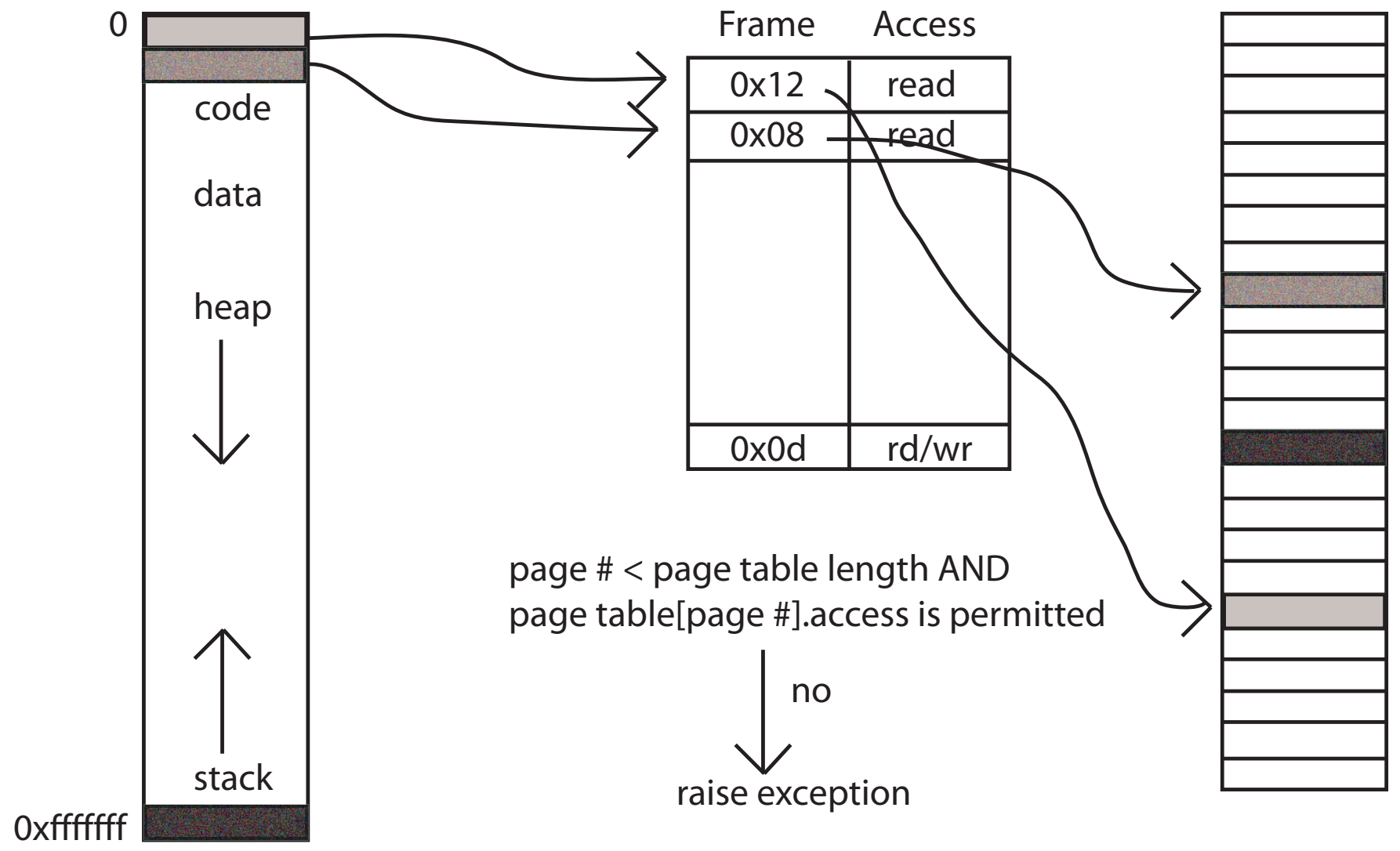
Physical Address:

| | |
|--------------------------|-------------|
| page table[page #].frame | page offset |
|--------------------------|-------------|

Process View of Memory

Page Table

Physical Memory



Process View

| |
|---|
| A |
| B |
| C |
| D |
| E |
| F |
| G |
| H |
| I |
| J |
| K |
| L |

Physical Memory

| |
|---|
| |
| I |
| J |
| K |
| L |
| |
| E |
| F |
| G |
| H |
| A |
| B |
| C |
| D |

Page Table

| |
|---|
| 4 |
| 3 |
| 1 |

Paging Questions

- What must be 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?
- 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
- UNIX fork with copy on write at page granularity
 - Copy page table entries to new process
 - Mark all pages as read-only
 - Trap into kernel on write (in child or parent)
 - Copy page and resume execution

Paging and Fast Program Start

- 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
 - Trap to OS kernel
 - OS kernel brings in page
 - Resumes execution
 - Remaining pages can be transferred in the background while program is running

Sparse Address Spaces

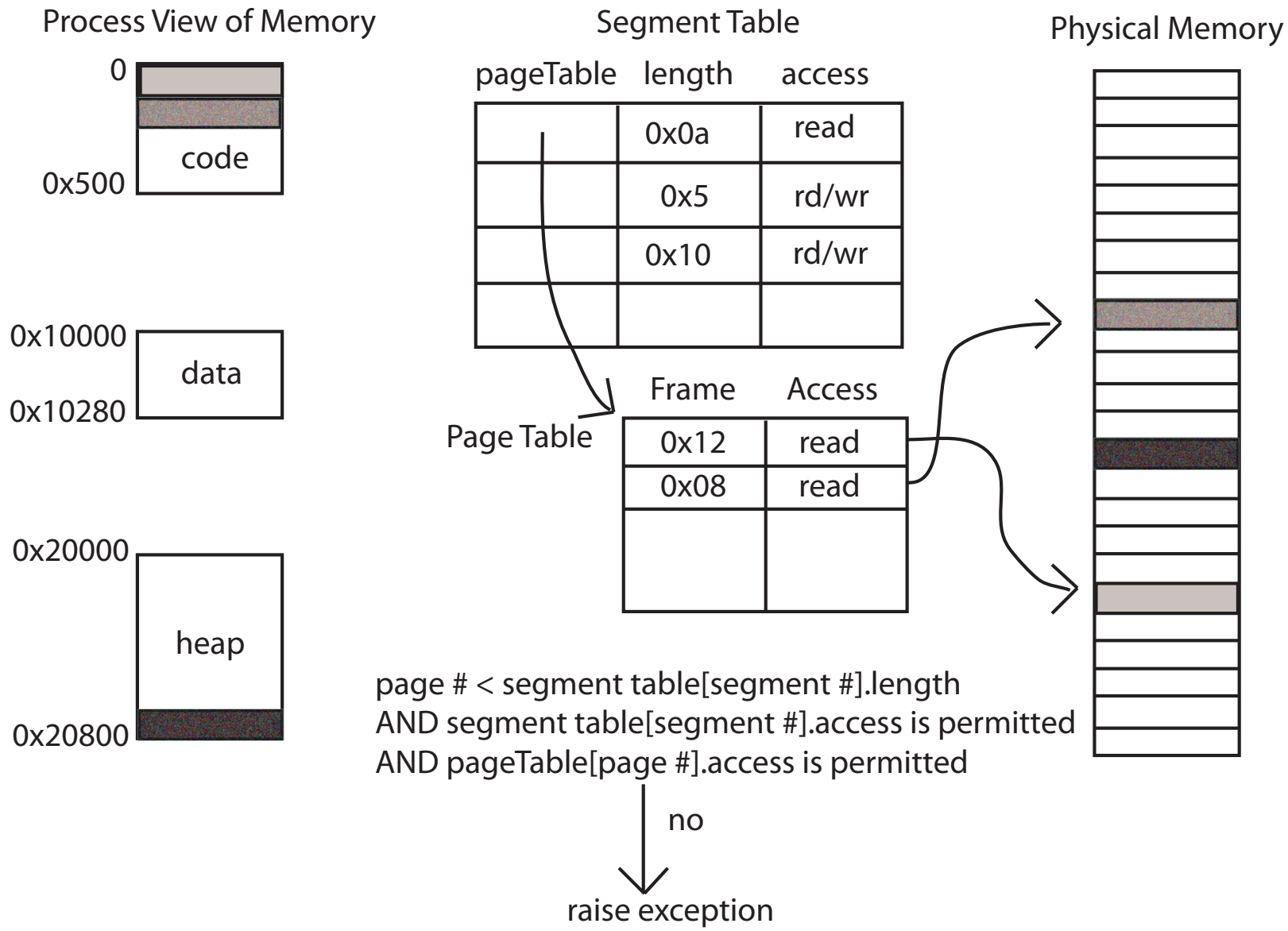
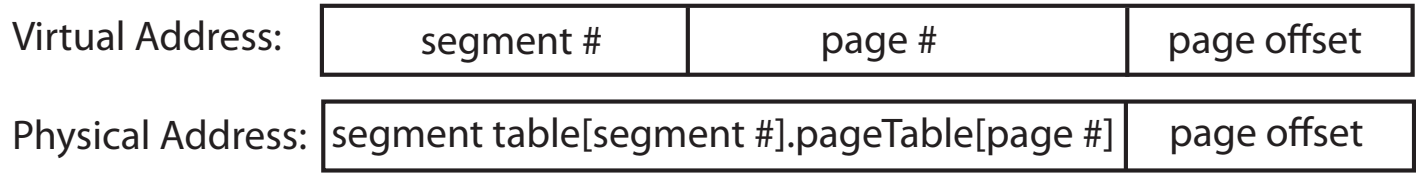
- Might want many separate segments
 - Per-processor heaps
 - Per-thread stacks
 - Memory-mapped files
 - Dynamically linked libraries
- What if virtual address space is sparse?
 - On 32-bit UNIX, code starts at 0
 - Stack starts at 2^{31}
 - 4KB pages => 500K page table entries
 - 64-bits => 4 quadrillion page table entries

Multi-level Translation

- Tree of translation tables
 - Paged segmentation
 - Multi-level page tables
 - Multi-level paged segmentation
- All 3: Fixed size page as lowest level unit
 - Efficient memory allocation
 - Efficient disk transfers
 - Easier to build translation lookaside buffers
 - Efficient reverse lookup (from physical -> virtual)
 - Page granularity for protection/sharing

Paged Segmentation

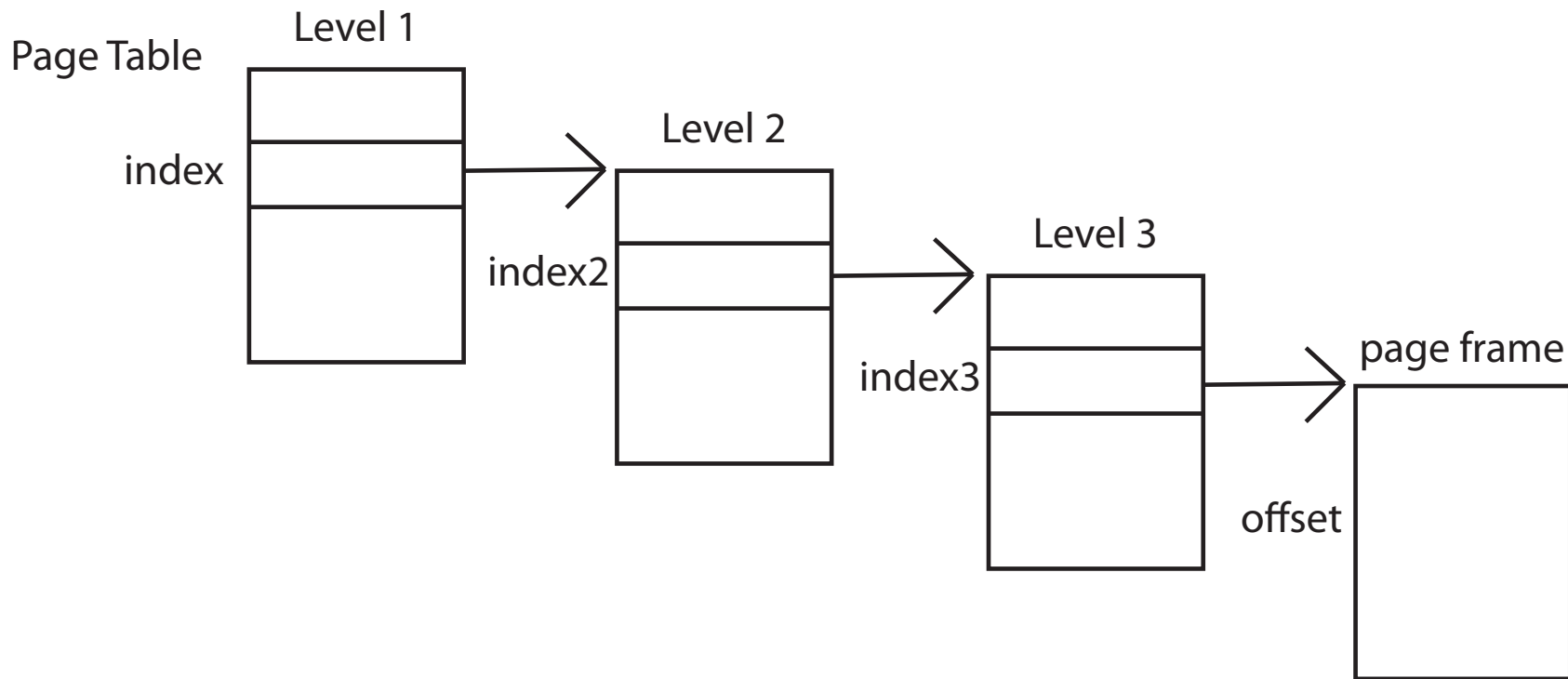
- Process memory is segmented
- Segment table entry:
 - Pointer to page table
 - Page table length (# of pages in segment)
 - Access permissions
- Page table entry:
 - Page frame
 - Access permissions
- Share/protection at either page or segment-level



Multilevel Paging



$$\text{Physical Address} = \text{pageTable}[\text{index}].\text{pageTable}[\text{index2}].\text{pageTable}[\text{index3}] | \text{page offset}$$



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
 - Only fill page table if needed
 - 32-bit: two level page table (per segment)
 - 64-bit: four level page table (per segment)

Multilevel Translation

- Pros:
 - Allocate/fill only as many page tables as used
 - Simple memory allocation
 - Share at segment or page level
- Cons:
 - Space overhead: at least one pointer per virtual page
 - Two or more lookups per memory reference

Portability

- Many operating systems keep their own memory translation data structures
 - List of memory objects (segments)
 - Virtual -> physical
 - Physical -> virtual
 - Simplifies porting from x86 to ARM, 32 bit to 64 bit
- Inverted page table
 - Hash from virtual page -> physical page
 - Space proportional to # of physical pages

Do we need multi-level page tables?

- Use inverted page table in hardware instead of multilevel tree
 - IBM PowerPC
 - Hash virtual page # to inverted page table bucket
 - Location in IPT => physical page frame
- Pros/cons?

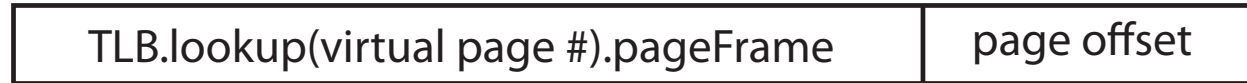
Efficient Address Translation

- Translation lookaside buffer (TLB)
 - Cache of recent virtual page \rightarrow physical page translations
 - If cache hit, use translation
 - If cache miss, walk multi-level page table
- Cost of translation =
Cost of TLB lookup +
 $\text{Prob}(\text{TLB miss}) * \text{cost of page table lookup}$

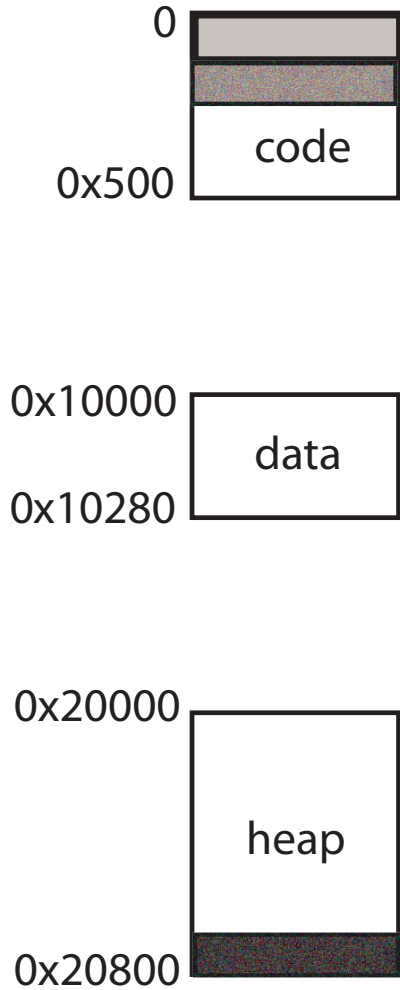
Virtual Address:



Physical Address:



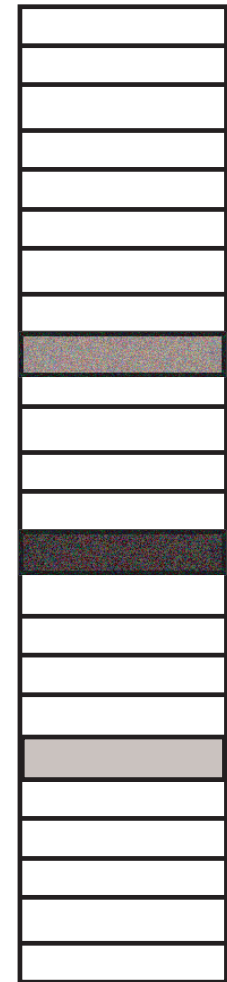
Process View of Memory



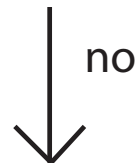
Translation Lookaside Buffer (TLB)

| | virtualPage | pageFrame | access |
|----|-------------|-----------|---------|
| =? | 0x0000 | 0x8 | read |
| =? | 0x40ff | 0x12 | rd/wr |
| =? | 0x0001 | 0xd | read |
| =? | | | invalid |

Physical Memory



TLB contains virtual page #?

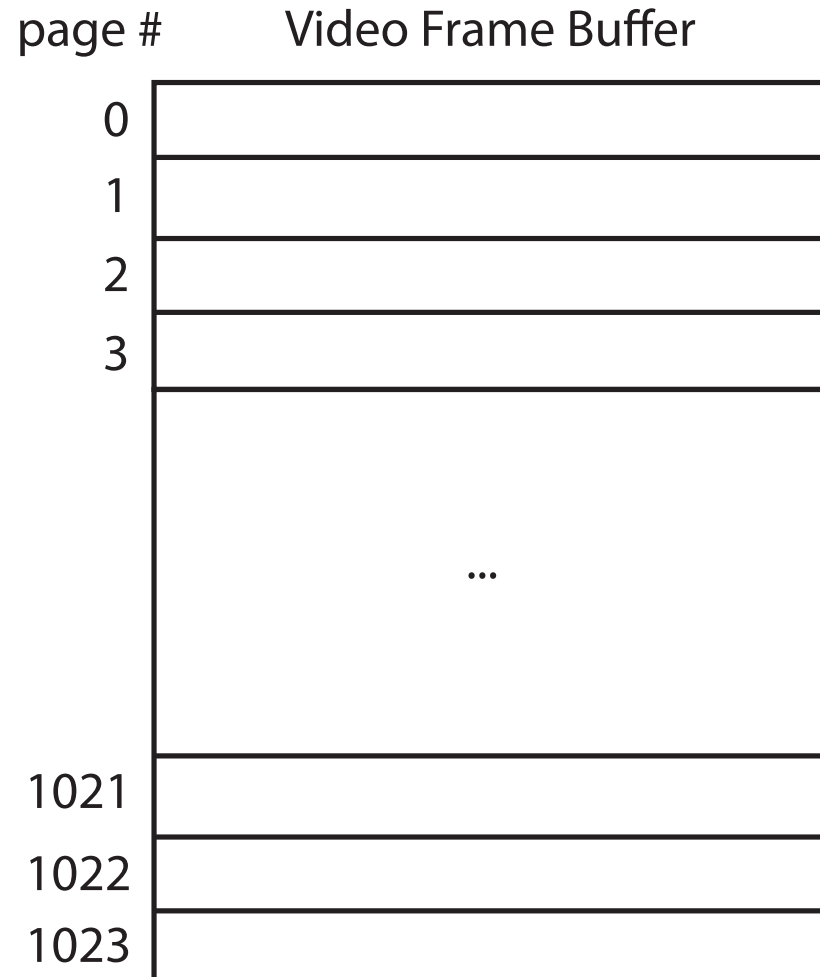


do page table lookup

Software Loaded TLB

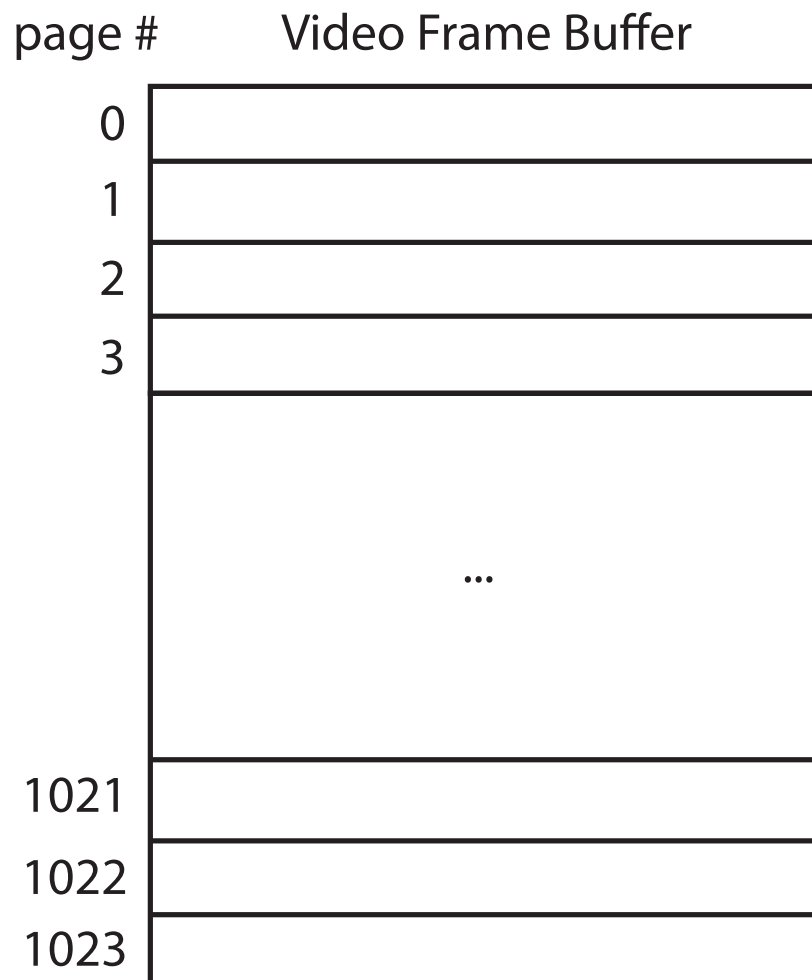
- Do we need a page table at all?
 - MIPS processor architecture
 - If translation is in TLB, ok
 - If translation is not in TLB, trap to kernel
 - Kernel computes translation and loads TLB
 - Kernel can use whatever data structures it wants
- Pros/cons?

When Do TLBs Work/Not Work?



When Do TLBs Work/Not Work?

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



Superpages

- TLB entry can be
 - A page
 - A superpage: a set of contiguous pages
 - x86: superpage is set of pages in one page table
 - x86 TLB entries
 - 4KB
 - 2MB
 - 1GB

When Do TLBs Work/Not Work, part 2

- What happens on a context switch?
 - Reuse TLB?
 - Discard TLB?
- Motivates hardware tagged TLB
 - Each TLB entry has process ID
 - TLB hit only if process ID matches current process

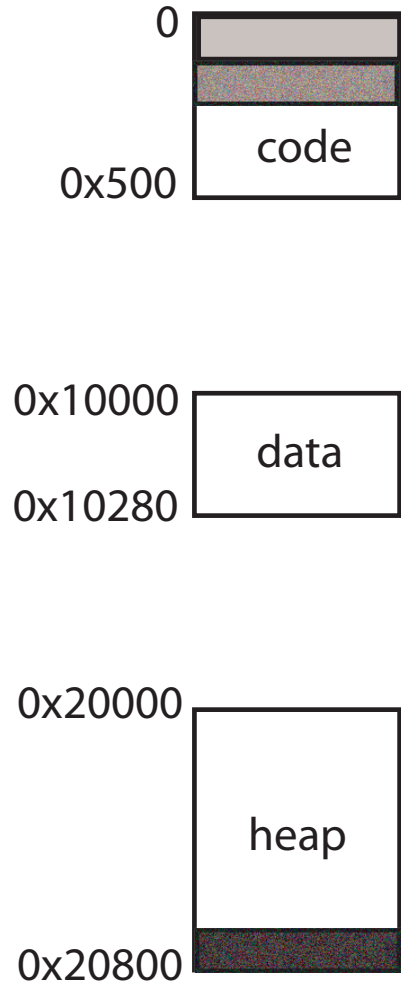
Virtual Address:

| | |
|---------------|-------------|
| virtualPage # | page offset |
|---------------|-------------|

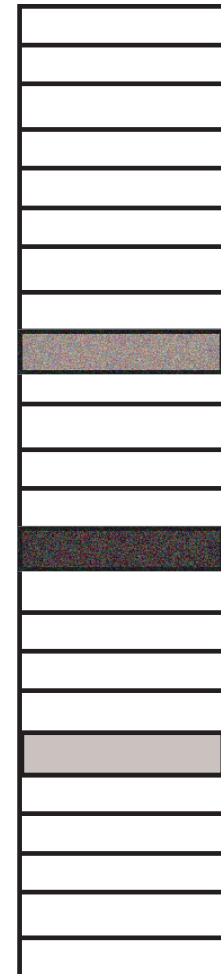
Physical Address:

| | |
|--------------------------------------|-------------|
| TLB.lookup(virtual page #).pageFrame | page offset |
|--------------------------------------|-------------|

Process 1 View of Memory



Physical Memory



Translation Lookaside Buffer (TLB)

| | process ID | virtualPage | pageFrame | access |
|----|------------|-------------|-----------|--------|
| =? | 0 | 0x0053 | 0x3 | rd/wr |
| =? | 1 | 0x40ff | 0x12 | rd/wr |
| =? | 1 | 0x0001 | 0xd | read |
| =? | 0 | 0x0001 | 0x5 | read |

TLB contains virtual page #
for the current process?

no

do page table lookup

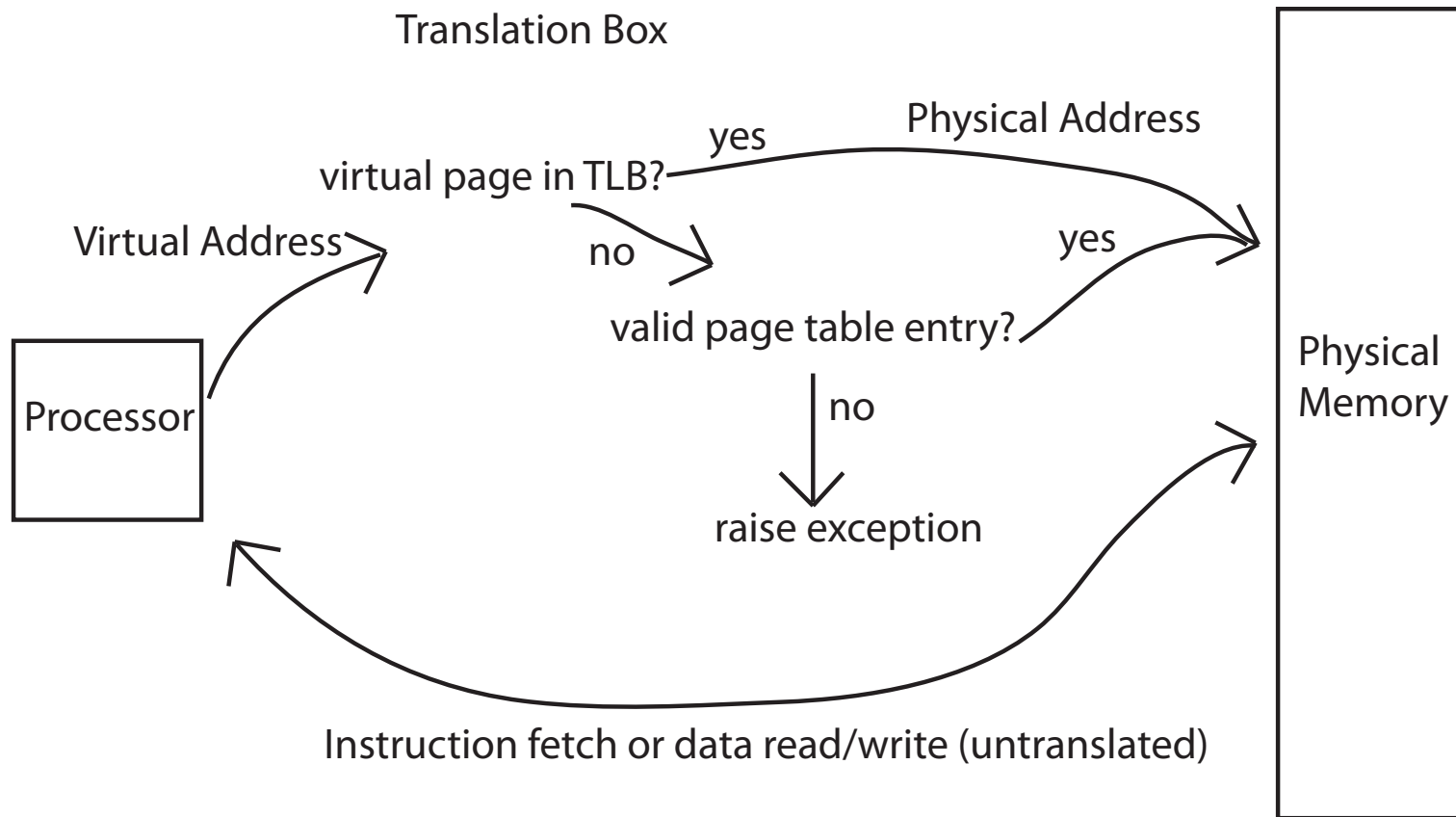
When Do TLBs Work/Not Work, part 3

- 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 must ask hardware to purge TLB entry
- On a multicore: TLB shutdown
 - OS must ask each CPU to purge TLB entry

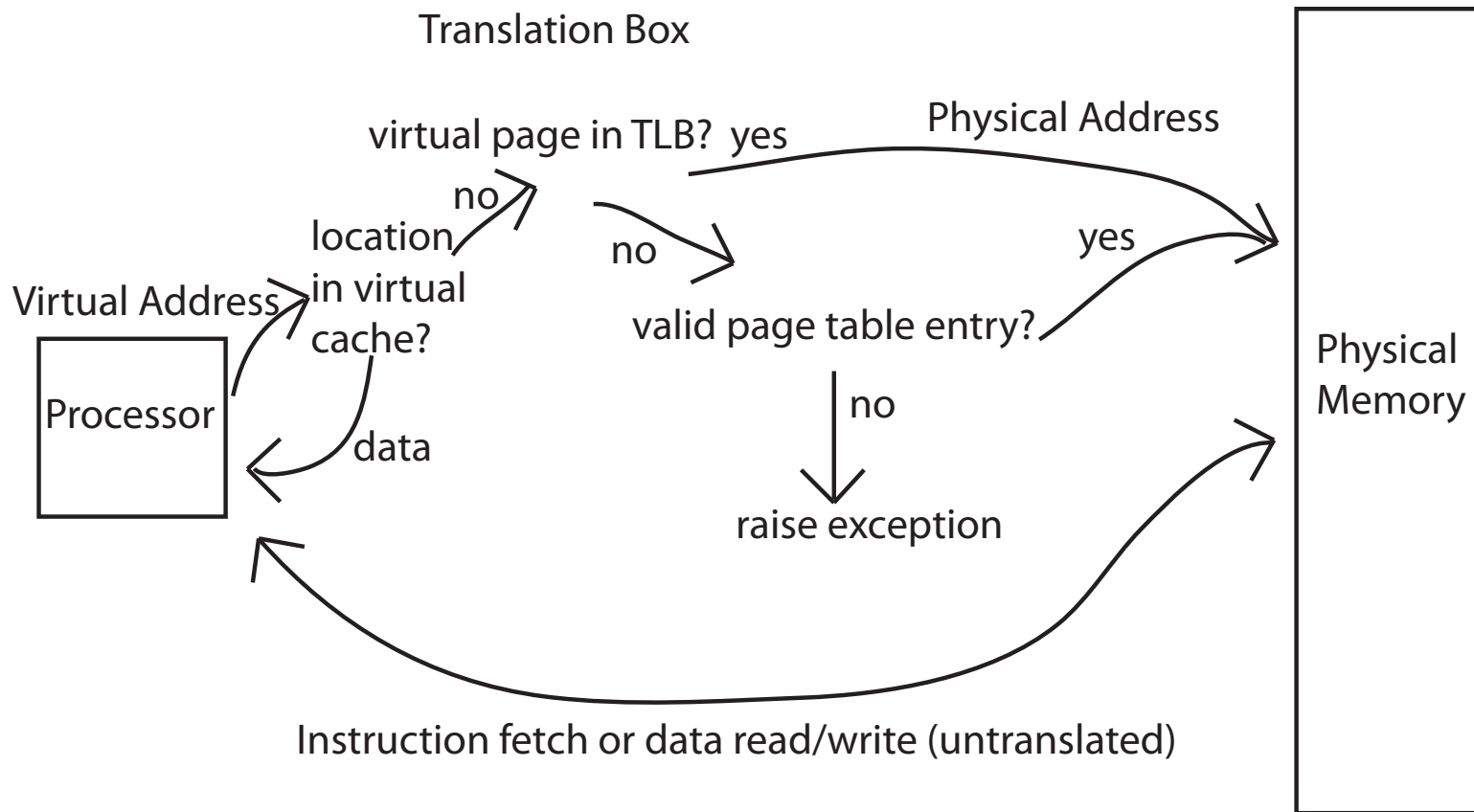
TLB Shutdown

| | process ID | virtualPage | pageFrame | access | |
|-----------------|------------|-------------|-----------|--------|-------|
| Processor 1 TLB | =? | 0 | 0x53 | 0x3 | rd/wr |
| | =? | 1 | 0x40ff | 0x12 | rd/wr |
| Processor 2 TLB | =? | 0 | 0x53 | 0x3 | rd/wr |
| | =? | 0 | 1 | 0x5 | read |
| Processor 3 TLB | =? | 1 | 0x40ff | 0x12 | rd/wr |
| | =? | 0 | 1 | 0x5 | read |

Address Translation with TLB



Virtually Addressed Caches



Translation on a Modern Processor

