CS 5600
Computer Systems

Lecture 7: Virtual Memory

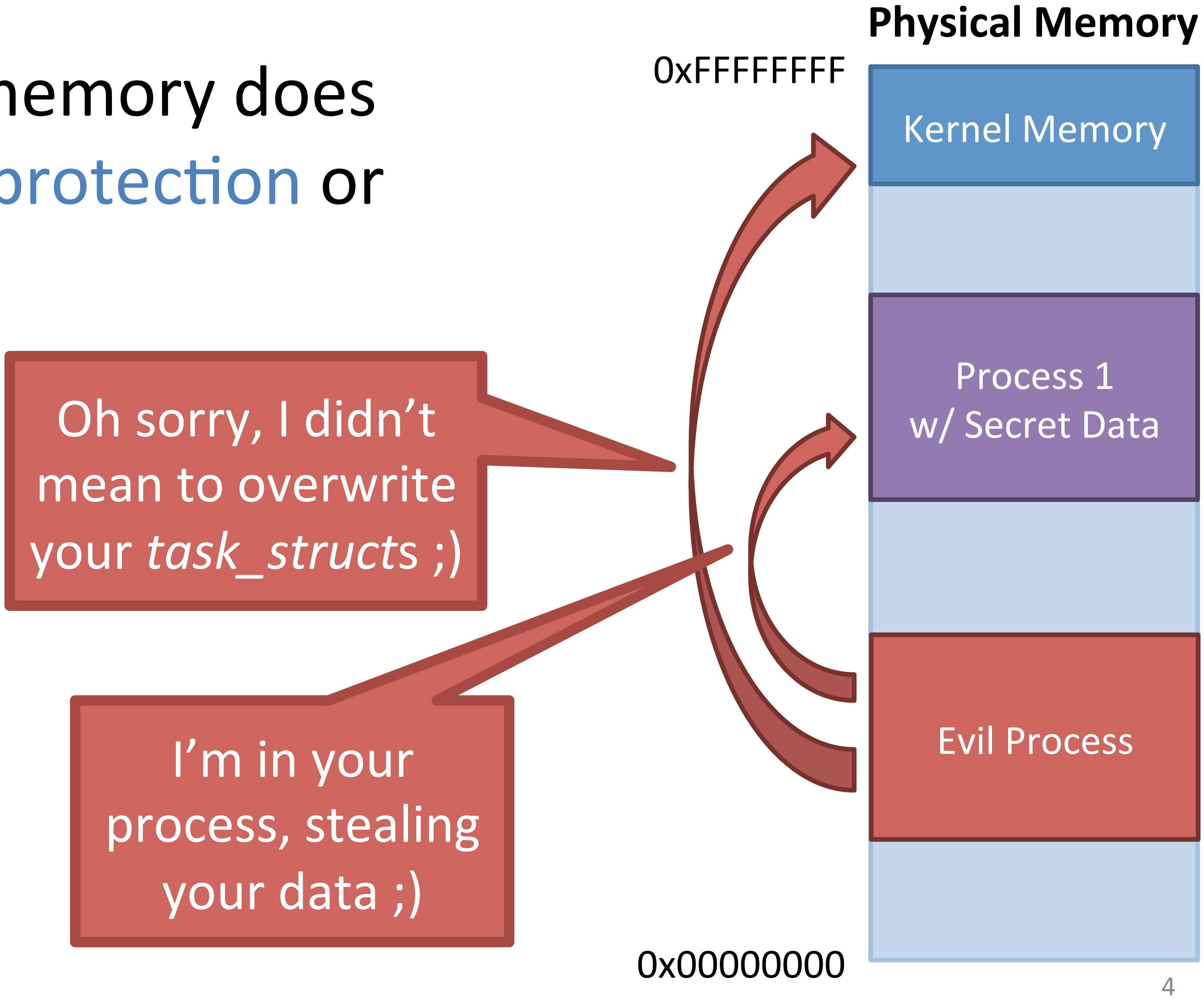
- Motivation and Goals
- Base and Bounds
- Segmentation
- Page Tables
- TLB
- Multi-level Page Tables
- Swap Space

Main Memory

- Main memory is conceptually very simple
  - Code sits in memory
  - Data is either on a stack or a heap
  - Everything gets accessed via pointers
  - Data can be written to or read from long term storage
- Memory is a simple and obvious device
  - So why is memory management one of the most complex features in modern OSes?

Protection and Isolation

- Physical memory does not offer protection or isolation



Physical Memory

0xFFFFFFFF

Kernel Memory

Process 1
w/ Secret Data

Evil Process

0x00000000

Oh sorry, I didn’t mean to overwrite your *task_structs* ;)

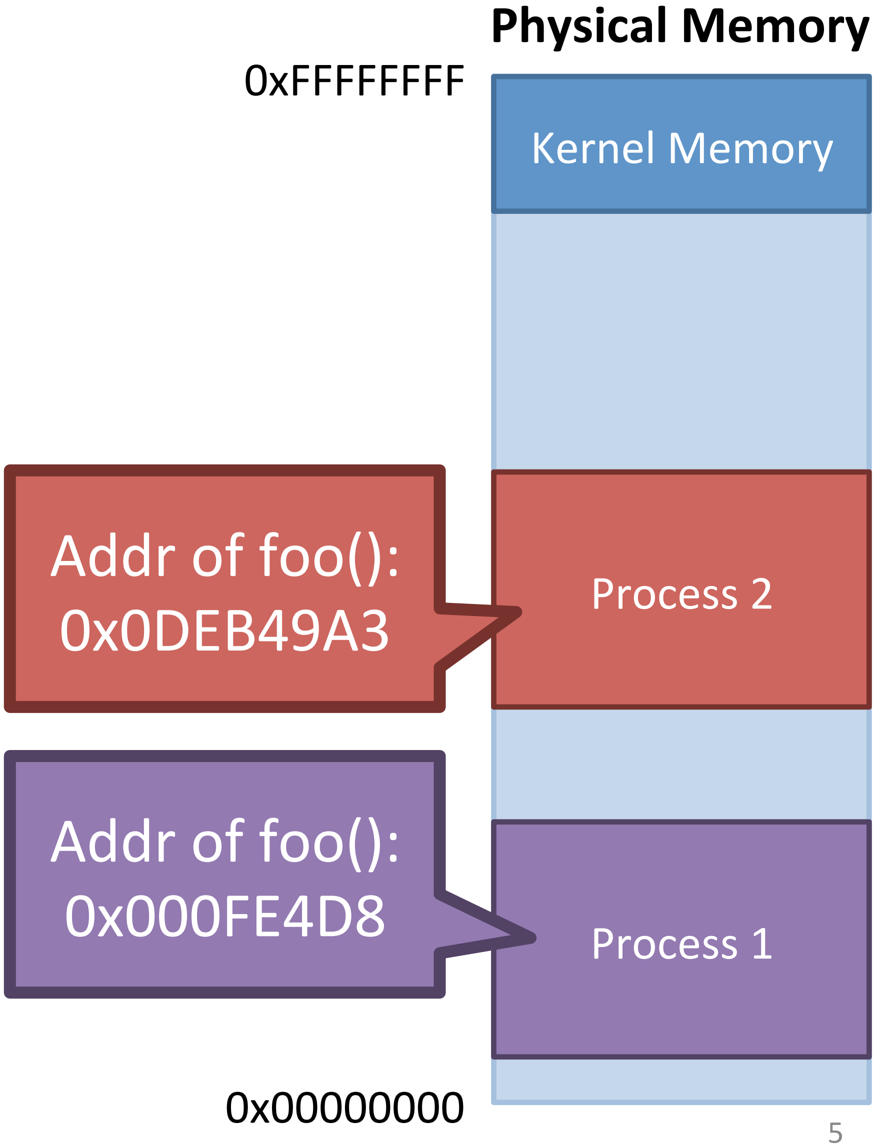
I’m in your process, stealing your data ;)

Compilation and Program Loading

- Compiled programs include fixed pointer addresses
- Example:

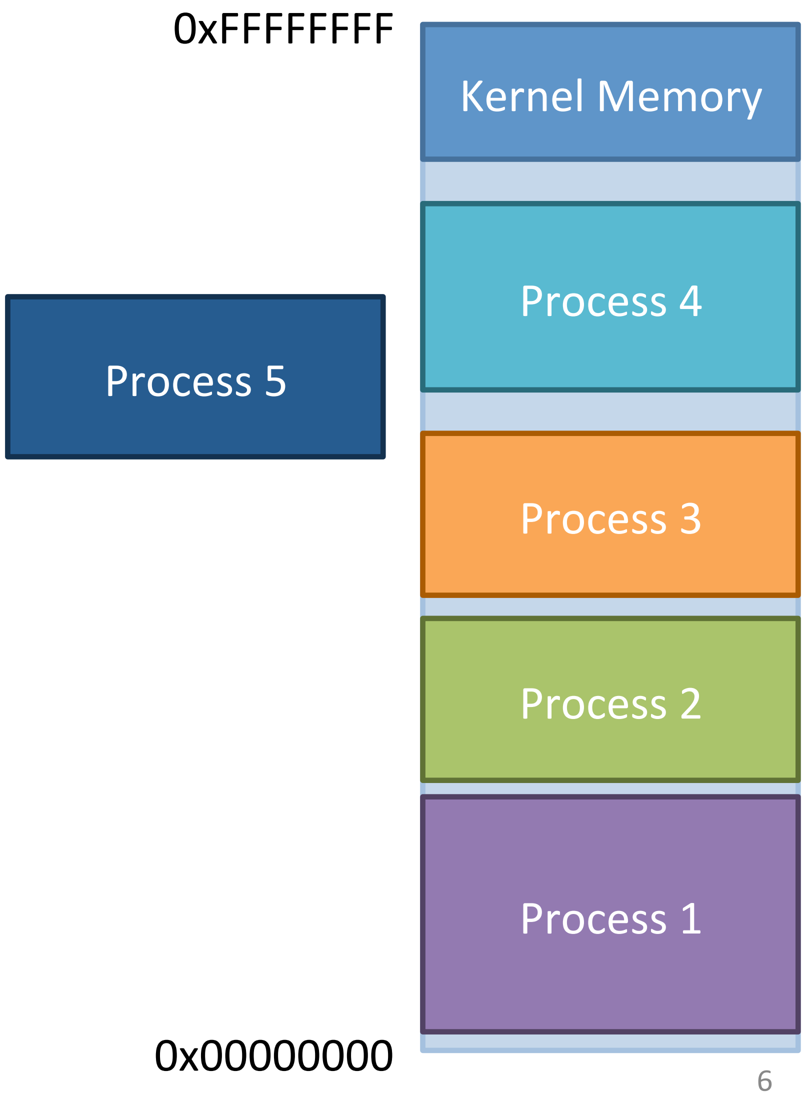
```
000FE4D8 <foo>:
…
000FE21A:    push eax
000FE21D:    push ebx
000FE21F:    call 0x000FE4D8
```

- Problem: what if the program is not loaded at corresponding address?



Physical Memory has Limited Size

- RAM is cheap, but not as cheap as solid state or cloud storage
- What happens when you run out of RAM?

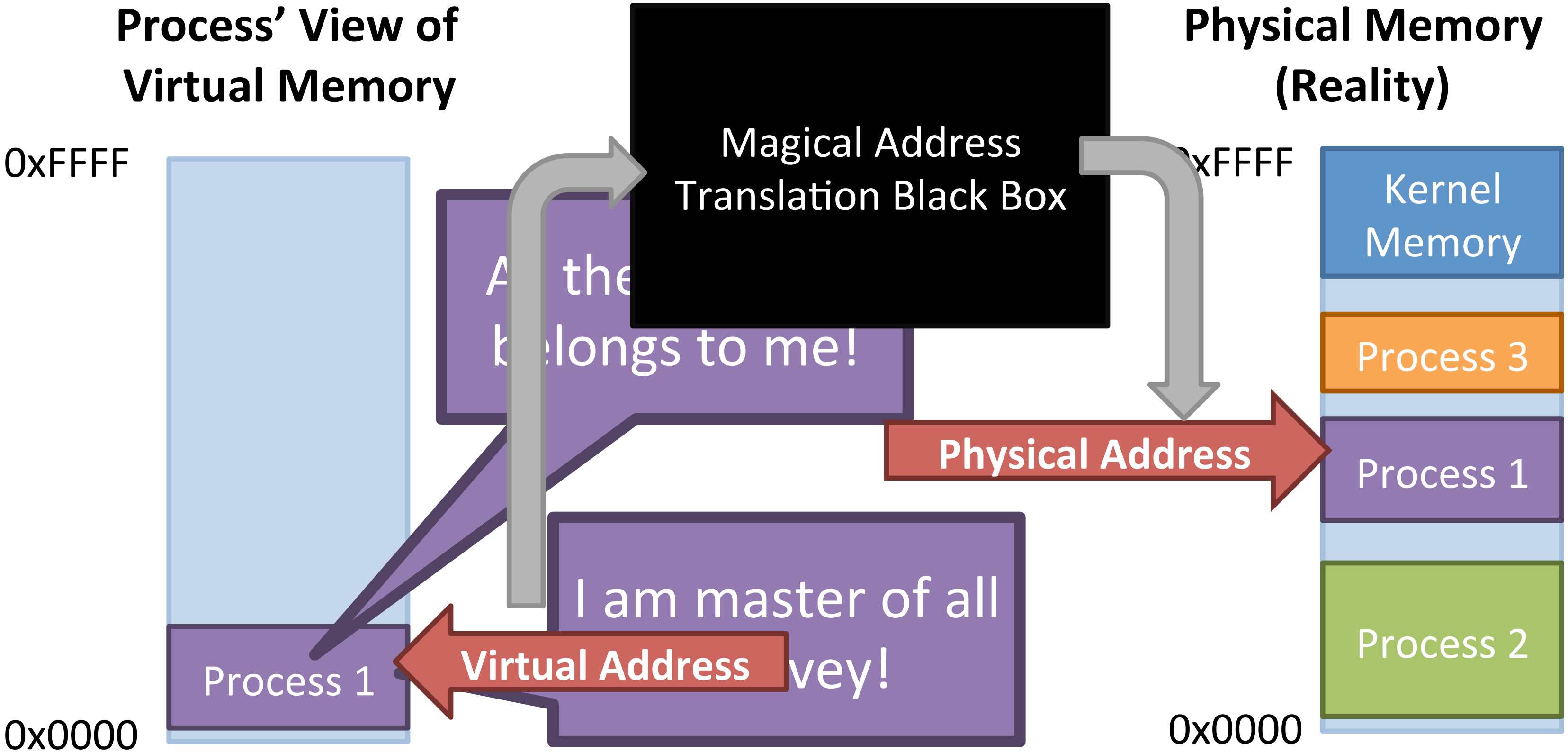


Physical vs. Virtual Memory

- Clearly, physical memory has limitations
  - No protection or isolation
  - Fixed pointer addresses
  - Limited size
  - Etc.
- Virtualization can solve these problems!
  - As well as enable additional, cool features

A Toy Example

- What do we mean by virtual memory?
  - Processes use virtual (or logical) addresses
  - Virtual addresses are translated to physical addresses



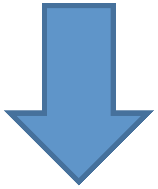
Implementing Address Translation

- In a system with virtual memory, each memory access must be translated
- Can the OS perform address translation?
  - Only if programs are interpreted
- Modern systems have hardware support that facilitates address translation
  - Implemented in the Memory Management Unit (MMU) of the CPU
  - Cooperates with the OS to translate virtual addresses into physical addresses

Virtual Memory Implementations

- There are many ways to implement an MMU
  - Base and bound registers
  - Segmentation
  - Page tables
  - Multi-level page tables

Old, simple, limited functionality



Modern, complex, lots of functionality

- We will discuss each of these approaches
  - How does it work?
  - What features does it offer?
  - What are the limitations?

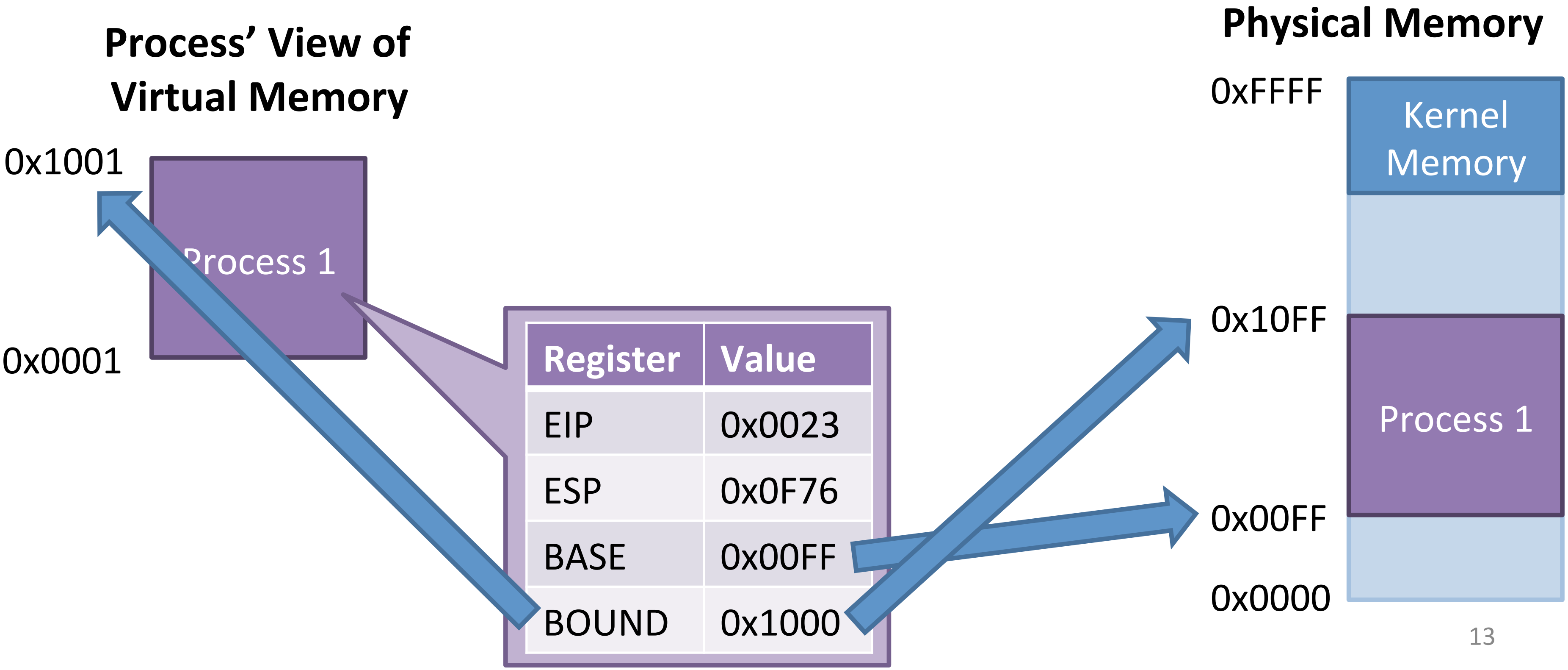
Goals of Virtual Memory

- Transparency
  - Processes are unaware of virtualization
- Protection and isolation
- Flexible memory placement
  - OS should be able to move things around in memory
- Shared memory and memory mapped files
  - Efficient interprocess communication
  - Shared code segments, i.e. dynamic libraries
- Dynamic memory allocation
  - Grow heaps and stacks on demand, no need to pre-allocate large blocks of empty memory
- Support for sparse address spaces
- Demand-based paging
  - Create the illusion of near-infinite memory

- Motivation and Goals
- Base and Bounds
- Segmentation
- Page Tables
- TLB
- Multi-level Page Tables
- Swap Space

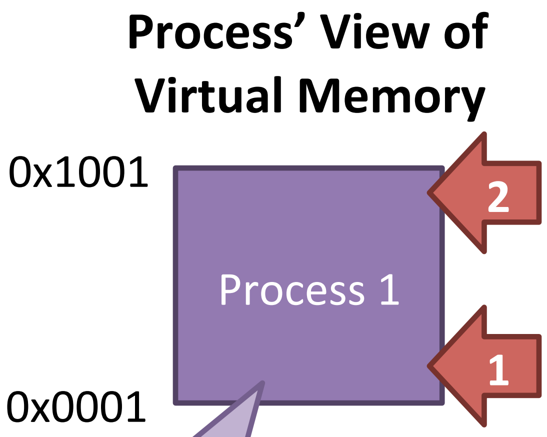
Base and Bounds Registers

- A simple mechanism for address translation
- Maps a contiguous virtual address region to a contiguous physical address region



Register	Value
EIP	0x0023
ESP	0x0F76
BASE	0x00FF
BOUND	0x1000

Base and Bounds Example



Register	Value
EIP	0x0023
ESP	0x0F76
BASE	0x00FF
BOUND	0x1000

0x0023 mov eax, [esp]

1) Fetch instruction

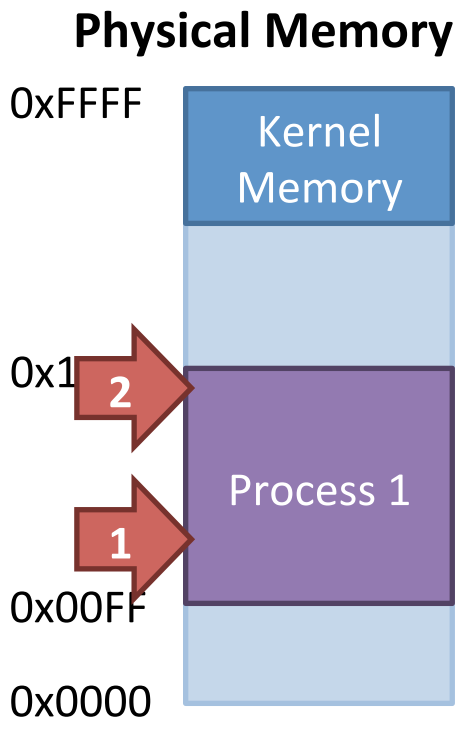
0x0023 + 0x00FF = 0x0122

2) Translate memory access

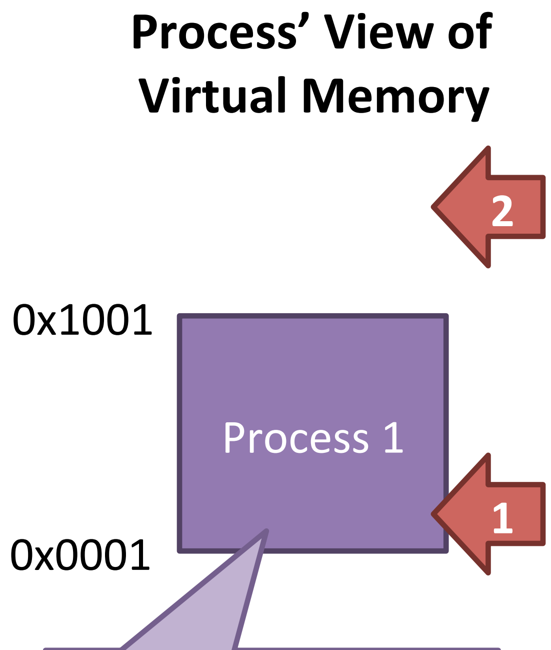
0x0F76 + 0x00FF = 0x1075

3) Move value to register

[0x1075] → eax



Protection and Isolation



Register	Value
EIP	0x0023
ESP	0x0F76
BASE	0x00FF
BOUND	0x1000

0x0023 mov eax, [0x4234]

1) Fetch instruction

0x0023 + 0x00FF = 0x0122

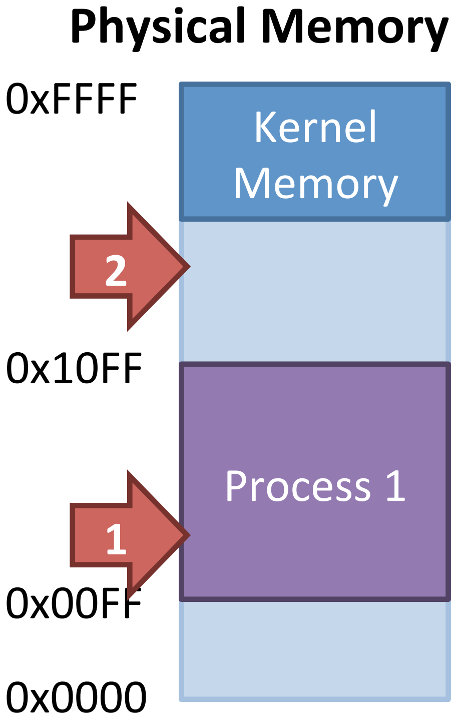
2) Translate memory access

0x4234 + 0x00FF = 0x4333

0x4333 > 0x10FF

(BASE + BOUND)

Raise Protection Exception!



Implementation Details

- BASE and BOUND are protected registers
  - Only code in Ring 0 may modify BASE and BOUND
  - Prevents processes from modifying their own sandbox
- Each CPU has one BASE and one BOUND register
  - Just like ESP, EIP, EAX, etc…
  - Thus, BASE and BOUND must be saved a restored during context switching

Base and Bound Pseudocode

```
PhysAddr = VirtualAddress + BASE
if (PhysAddr >= BASE + BOUND)
    RaiseException(PROTECTION_FAULT)
Register = AccessMemory(PhysAddr)
```

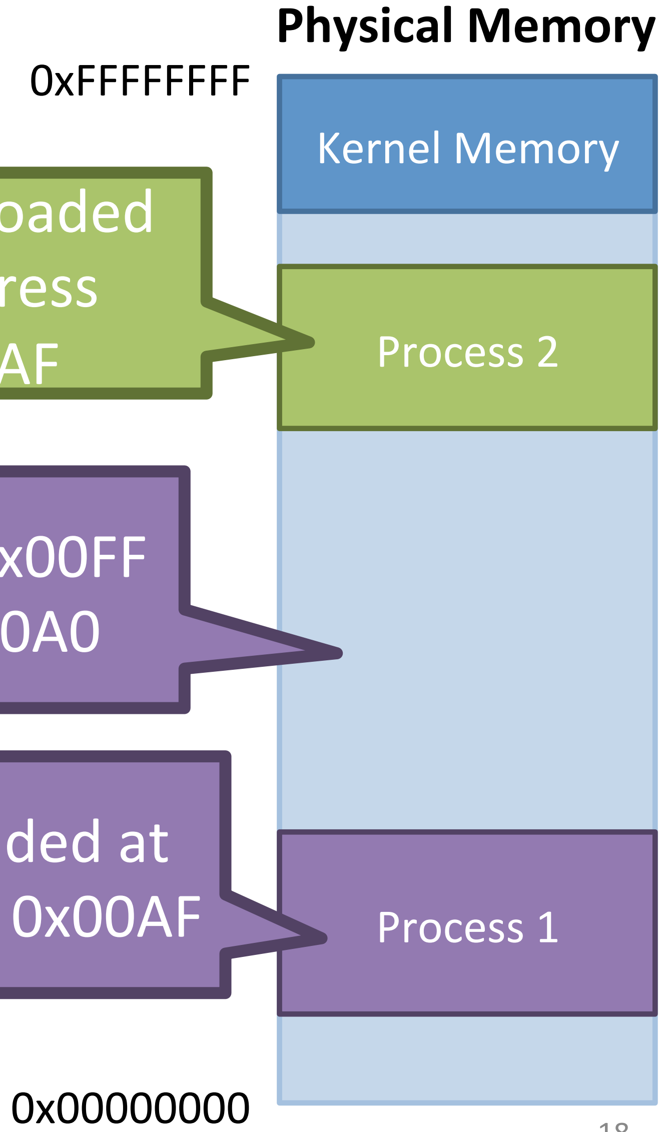
Advantages of Base and Bound

- Simple hardware imple
- Simple to manage eac virtual space
- Processes can l arbitrary fixed
- Offers protection and is
- Offers flexible placemen in memory

No, I’m loaded at address 0x00AF

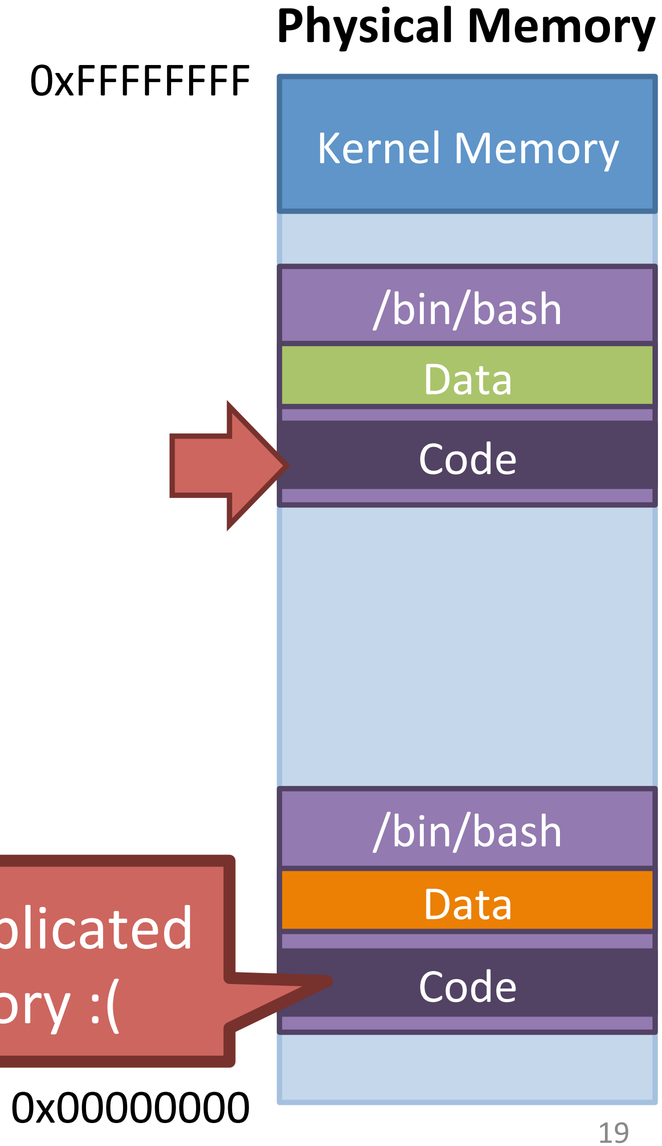
Previous BASE → 0x00FF
New BASE → 0x10A0

I’m loaded at address 0x00AF



Limitations of Base and Bound

- Processes can overwrite their own code
  - Processes aren't protected from themselves
- No sharing of memory
  - Code (read-only) is mixed in with data (read/write)
- Process memory cannot grow dynamically
  - May lead to internal fragmentation

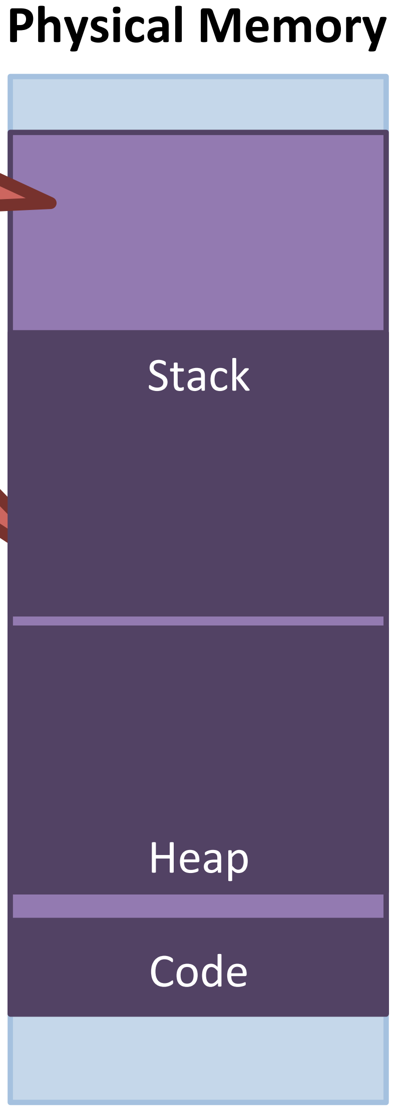


Code is duplicated in memory :(

Internal Fragmentation

- BOUND determines the max amount of memo to a process
- How much memo allocate?
  - Empty space leads to internal fragmentation
- What if we don’t allocate enough?
  - Increasing BOUND after the process is running doesn’t help

Increasing BOUND doesn’t move the stack away from the heap



- Motivation and Goals
- Base and Bounds
- Segmentation
- Page Tables
- TLB
- Multi-level Page Tables
- Swap Space

Towards Segmented Memory

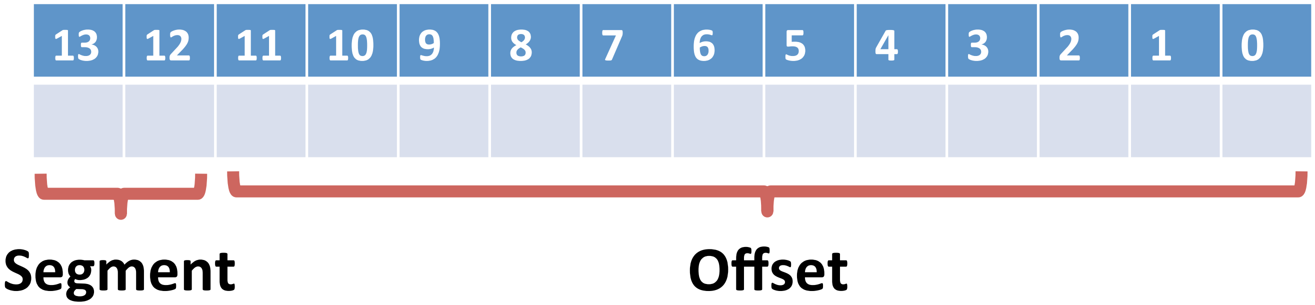
- Having a single BASE and a single BOUND means code, stack, and heap are all in one memory region
  - Leads to internal fragmentation
  - Prevents dynamically growing the stack and heap
- Segmentation is a generalization of the base and bounds approach
  - Give each process several pairs of base/bounds
    - May or may not be stored in dedicated registers
  - Each pair defines a segment
  - Each segment can be moved or resized independently

Segmentation Details

- The code and data of a process get split into several segments
  - 3 segments is common: code, heap, and stack
  - Some architectures support >3 segments per process
- Each process views its segments as a contiguous region of memory
  - But in physical memory, the segments can be placed in arbitrary locations
- Question: given a virtual address, how does the CPU determine which segment is being addressed?

Segments and Offsets

- Key idea: split virtual addresses into a segment index and an offset
- Example: suppose we have 14-bit addresses
  - Top 2 bits are the segment
  - Bottom 12 bits are the offset



13	12	11	10	9	8	7	6	5	4	3	2	1	0

Segment

Offset

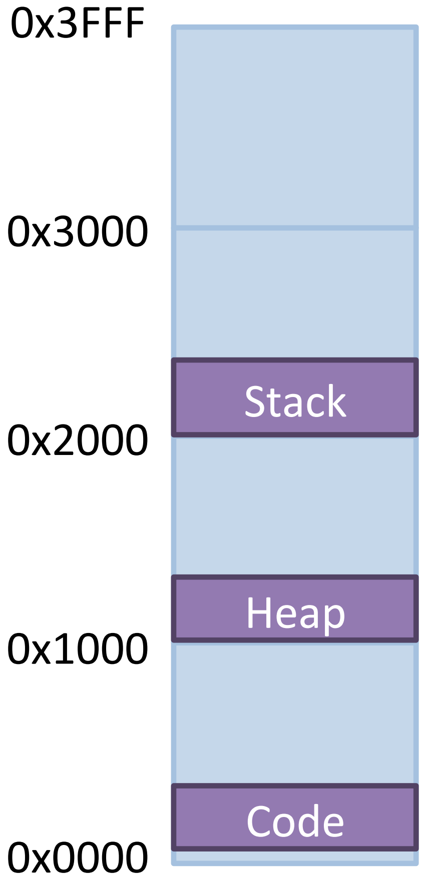
- 4 possible segments per process
  - 00, 01, 10, 11

Separation of Responsibility

- The OS manages segments and their indexes
  - Creates segments for new processes in free physical memory
  - Builds a table mapping segments indexes to base addresses and bounds
  - Swaps out the tables and segment registers during context switches
  - Frees segments from physical memory
- The CPU translates virtual addresses to physical addresses on demand
  - Uses the segment registers/segment tables built by the OS

Segmentation Example

Process' View of Virtual Memory



0x0023 mov eax, [esp]

1) Fetch instruction

0x0023 (EIP) - 00000000100011

0x0020 (CS) + 0x0023 = 0x0043

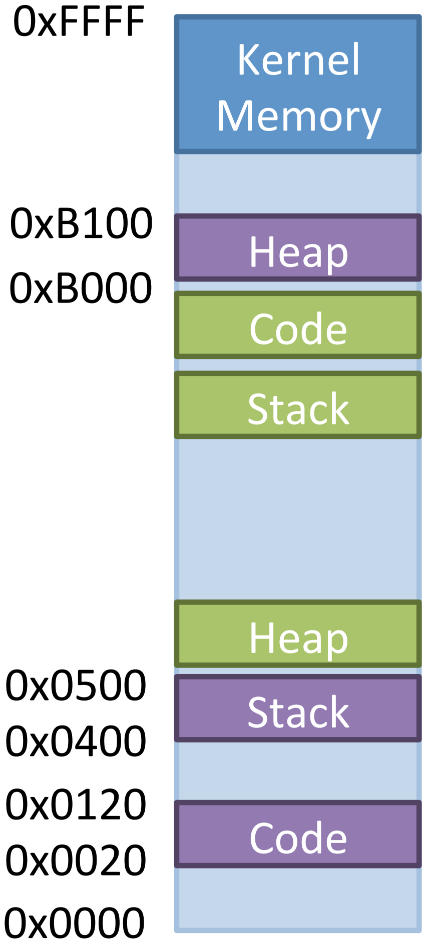
2) Translate memory access

0x2015 (ESP) – 10000000010101

0x0400 (SS) + 0x0015 = 0x0415

Segment	Index	Base	Bound
CS (Code)	00	0x0020	0x0100
HS (Heap)	01	0xB000	0x0100
SS (Stack)	10	0x0400	0x0100

Physical Memory



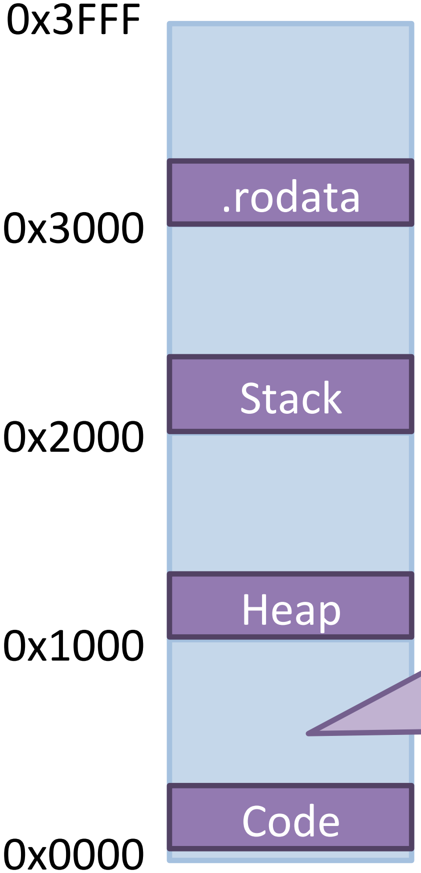
More on Segments

- In the previous example, we use a 14-bit address space with 2 bits reserved for the segment index
  - This limits us to 4 segments per process
  - Each segment is 2^{12} = 4KB in size
- Real segmentation systems tend to have
  1. More bits for the segments index (16-bits for x86)
  2. More bits for the offset (16-bits for x86)
- However, segments are course-grained
  - Limited number of segments per process (typically ~4)

Segment Permissions

- Many CPUs (including x86) support permissions on segments
  - Read, write, and executable
- Disallowed operations trigger an exception
  - E.g. Trying to write to the code segment

Process 1's View of Virtual Memory



Index	Base	Bound	Permissions
00	0x0020	0x0100	RX
01	0xB000	0x0100	RW
10	0x0400	0x0100	RW
11	0xE500	0x100	R

x86 Segments

- Intel 80286 introduced segmented memory
  - CS – code segment register
  - SS – stack segment register
  - DS – data segment register
  - ES, FS, GS – extra segment registers
- In 16-bit (real mode) x86 assembly, segment:offset notation is common

```
mov [ds:eax], 42  // move 42 to the data segment, offset
                  // by the value in eax
mov [esp], 23     // uses the SS segment by default
```

x86 Segments Today

- Segment registers and their associated functionality still exist in today’s x86 CPUs
- However, the 80386 introduced page tables
  - Modern OSes “disable” segmentation
  - The Linux kernel sets up four segments during bootup

Segment Name	Description	Base	Bound	Ring
KERNEL_CS	Kernel code	0	4 GB	0
KERNEL_DS	Kernel data	0	4 GB	0
USER_CS	User code	0	4 GB	3
USER_DS	User data	0	4 GB	3

Pages are used to virtualize memory, not segments

Used to label pages with protection levels

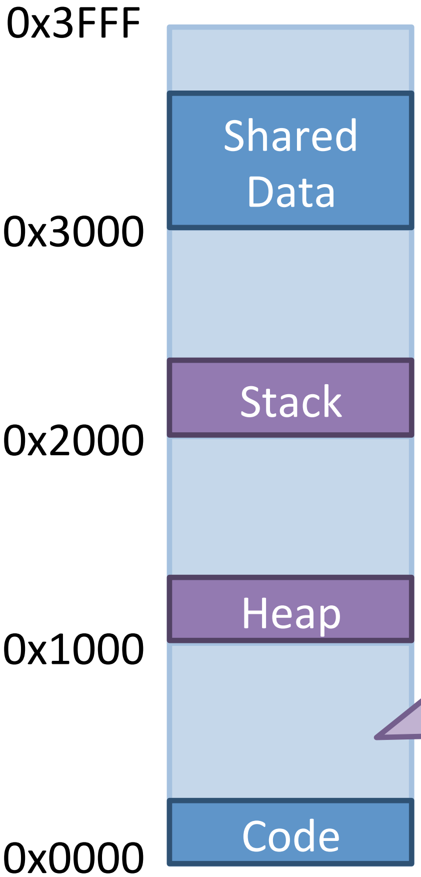
What is a Segmentation Fault?

- If you try to read/write memory outside a segment assigned to your process
- Examples:

  -
    ```
    char buf[5];
    strcpy(buf, “Hello World”);
    return 0; // why does it seg fault when you return?
    ```
- Today “segmentation fault” is an anachronism
  - All modern systems use page tables, not segments

Shared Memory

Process 1's View of Virtual Memory

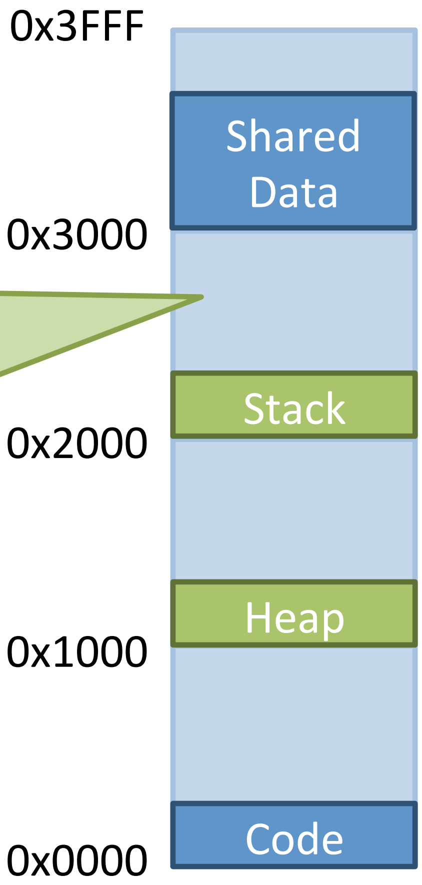


Same 00 and 11 physical segments

Different 01 and 10 physical segments

Index	Base	Bound
00	0x0020	0x0100
01	0xC000	0x0100
10	0x0600	0x0100
11	0xE500	0x0300

Process 2's View of Virtual Memory



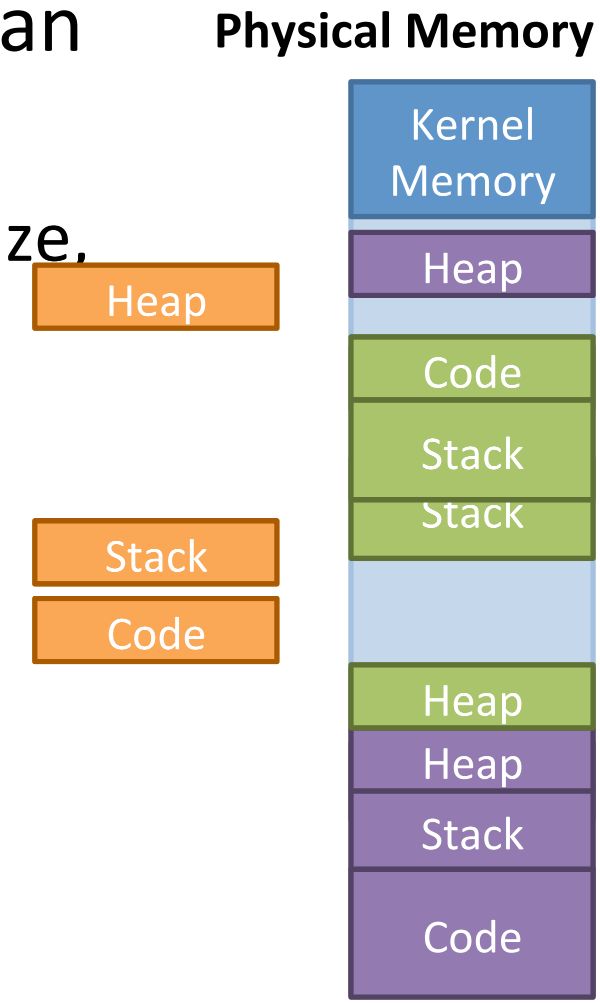
Index	Base	Bound
00	0x0020	0x0100
01	0xB000	0x0100
10	0x0400	0x0100
11	0xE500	0x0300

Advantages of Segmentation

- All the advantages of base and bound
- Better support for sparse address spaces
  - Code, heap, and stack are in separate segments
  - Segment sizes are variable
  - Prevents internal fragmentation
- Supports shared memory
- Per segment permissions
  - Prevents overwriting code, or executing data

External Fragmentation

- Problem: variable size segments can lead to external fragmentation
  - Memory gets broken into random size, non-contiguous pieces
- Example: there is enough free memory to start a new process
  - But the memory is fragmented :(
- Compaction can fix the problem
  - But it is extremely expensive



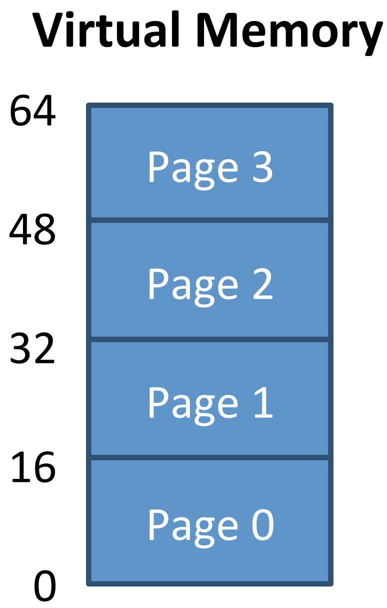
- Motivation and Goals
- Base and Bounds
- Segmentation
- Page Tables
- TLB
- Multi-level Page Tables
- Swap Space

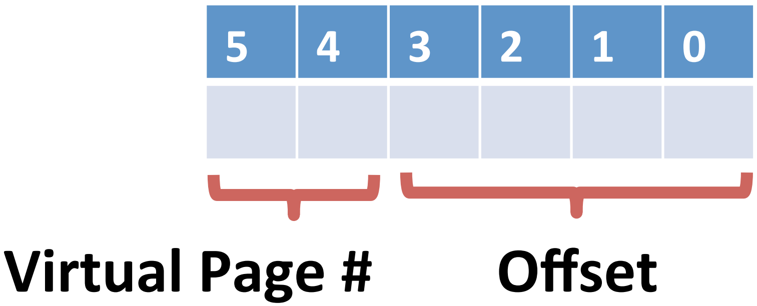
Towards Paged Memory

- Segments improve on base and bound, but they still aren't granular enough
  - Segments lead to external fragmentation
- The paged memory model is a generalization of the segmented memory model
  - Physical memory is divided up into physical pages (a.k.a. frames) of fixed sizes
  - Code and data exist in virtual pages
  - A table maps virtual pages → physical pages (frames)

Toy Example

- Suppose we have a 64-byte virtual address space
  - Lets specify 16 bytes per page
- How many bits do virtual addresses need to be in this system?
  - $2^6 = 64$ bytes, thus 6 bit addresses
- How many bits of the virtual address are needed to select the physical page?
  - 64 bytes / 16 bytes per page = 4 pages
  - $2^2 = 4$, thus 2 bits to select the page

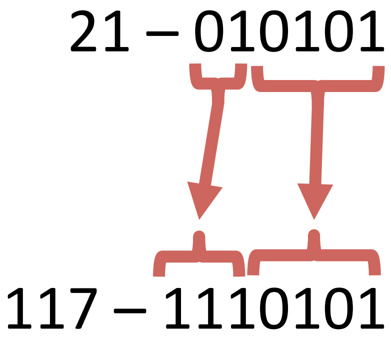




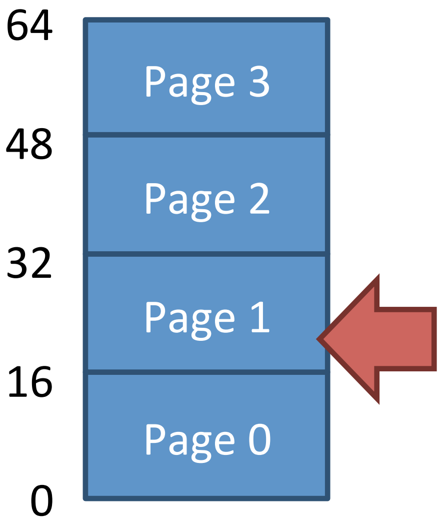
Toy Example, Continued

mov eax, [21]

Translation

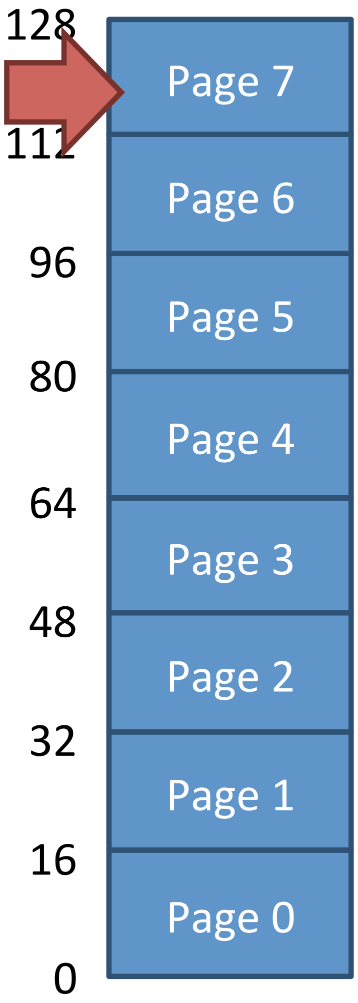


Virtual Memory



Virtual Page #	Physical Page #
00 (0)	010 (2)
01 (1)	111 (7)
10 (2)	100 (4)
11 (3)	001 (1)

Physical Memory

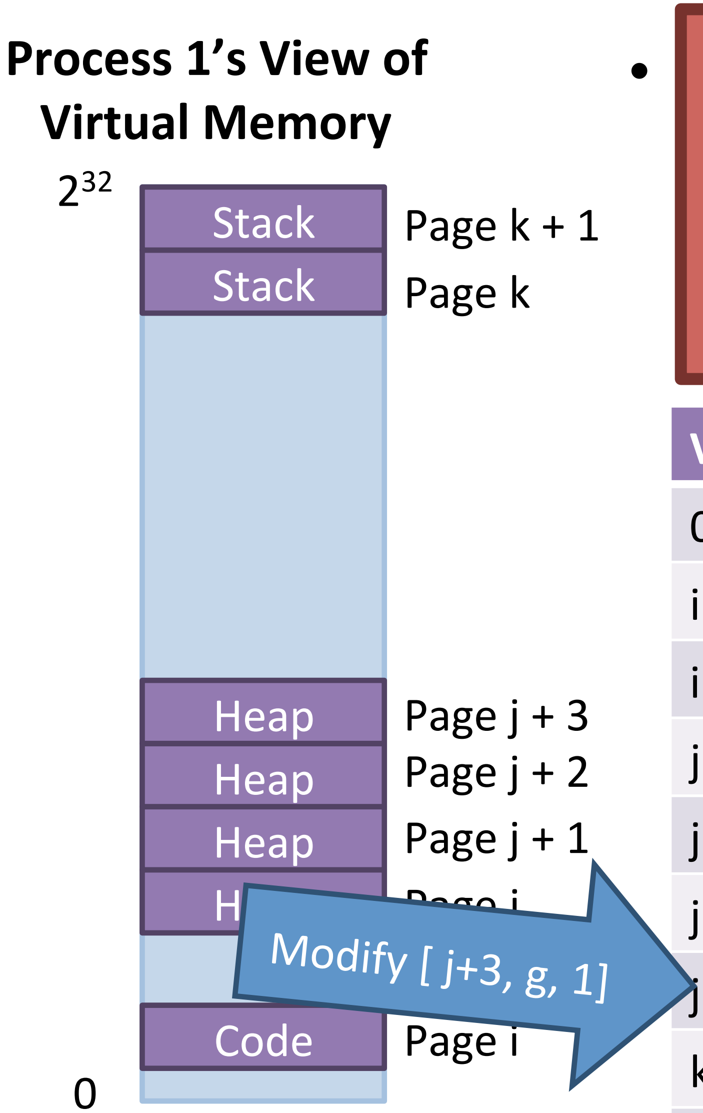


Concrete Example

- Assume a 32-bit virtual and physical address space
  - Fix the page size at 4KB (4096 bytes, 2^{12})
- How many total pages will there be?
  - $2^{32} / 2^{12}$ = 1048576 (2^{20})
- How many bits of a virtual address are needed to select the physical page?
  - 20 bits (since there are 1048576 total pages)
- Assume that eac
  - How big will the page ta e?
  - 1048586 * 4 bytes = 4MB of space

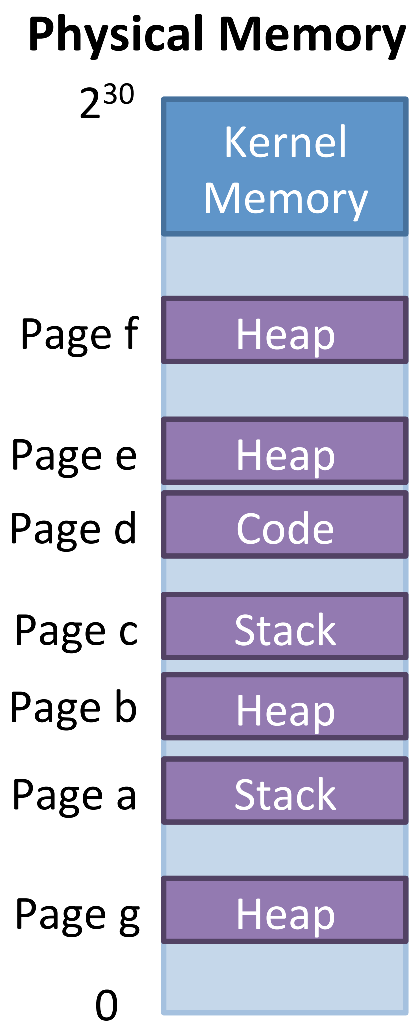
- Each process needs its own page table
- 100 processes = 400MB of page tables

Concrete Example, Continued



The vast majority of each process’ page table is empty, i.e. the table is sparse

VPN	PFN	Valid?
0 … i - 1	whatever	0
i	d	1
i + 1 … j – 1	whatever	0
j	b	1
j + 1	f	1
j + 2	e	1
j + 3 … k - 1	whatever	0
k	a	1
k + 1	c	1



Page Table Implementation

- The OS creates the page table for each process
  - Page tables are typically stored in kernel memory
  - OS stores a pointer to the page table in a special register in the CPU (CR3 register in x86)
  - On context switch, the OS swaps the pointer for the old processes table for the new processes table
- The CPU uses the page table to translate virtual addresses into physical addresses

x86 Page Table Entry

- On x86, page table entries (PTE) are 4 bytes

31 - 12	11 - 9	8	7	6	5	4	3	2	1	0
Page Frame Number (PFN)	Unused	G	PAT	D	A	PCD	PWT	U/S	W	P

- Bits related to permissions
  - W – writable bit – is the page writable, or read-only?
  - U/S – user/supervisor bit – can user-mode processes access this page?
- Hardware caching related bits: G, PAT, PCD, PWT
- Bits related to swapping
  - P – present bit – is this page in p
  - A – accessed bit – has this page been read recently?
  - D – dirty bit – has this page been written recently?

We will revisit these later in the lecture…

Page Table Pseudocode

```
// Extract the VPN from the virtual address
VPN = (VirtualAddress & VPN_MASK) >> SHIFT
// Form the address of the page-table entry (PTE)
PTEAddr = PTBR + (VPN * sizeof(PTE))
// Fetch the PTE
PTE = AccessMemory(PTEAddr)
if (PTE.Valid == False) // Check if process can access the page
    RaiseException(SEGMENTATION_FAULT)
else if (CanAccess(PTE.ProtectBits) == False)
    RaiseException(PROTECTION_FAULT)
// Access is OK: form physical address and fetch it
offset = VirtualAddress & OFFSET_MASK
PhysAddr = (PTE.PFN << PFN_SHIFT) | offset
Register = AccessMemory(PhysAddr)
```

Tricks With Permissions and Shared Pages

- Recall how fork() is implemented
  - OS creates a copy of all pages controlled by the parent
- fork() is a slooooow operation
  - Copying all that memory takes a looooong time
- Can we improve the efficiency of fork()?
  - Yes, if we are clever with shared pages and permissions!

Copy-on-Write

- Key idea: rather than copy all of the parents pages, create a new page table for the child that maps to all of the parents pages
  - Mark all of the pages as read-only
  - If parent or child writes to a page, a protection exception will be triggered
  - The OS catches the exception, makes a copy of the target page, then restarts the write operation
- Thus, all unmodified data is shared
  - Only pages that are written to get copied, on demand

Copy-on-Write Example

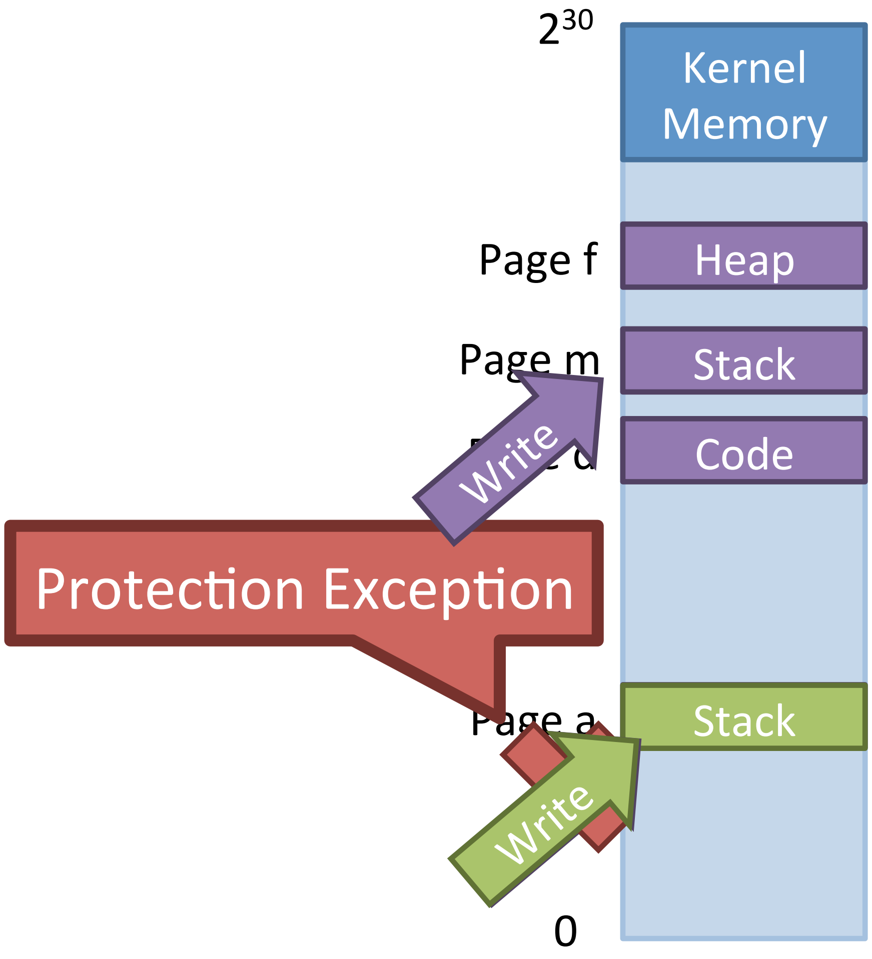
Parents Page Table

Function	VPN	PFN	Writable?
Code	i	d	0
Heap	j	b	0
Stack	k	m	1

Childs Page Table

Function	VPN	PFN	Writable?
Code	i	d	0
Heap	j	b	0
Stack	k	a	1

Physical Memory



Zero-on-Reference

- How much physical memory do we need to allocate for the heap of a new process?
  - Zero bytes!
- When a process touches the heap
  - Segmentation fault into OS kernel
  - Kernel allocates some memory
  - Zeros the memory
    - Avoid accidentally leaking information!
  - Restart the process

Advantages of Page Tables

- All the advantages of segmentation
- Even better support for sparse address spaces
  - Each page is relatively small
  - Fine-grained page allocations to each process
  - Prevents internal fragmentation
- All pages are the same size
  - Each to keep track of free memory (say, with a bitmap)
  - Prevents external fragmentation
- Per segment permissions
  - Prevents overwriting code, or executing data

Problems With Page Tables

- Page tables are huge
  - On a 32-bit machine with 4KB pages, each process' table is 4MB
  - On a 64-bit machine with 4KB pages, there are 240 entries per table → 240 * 4 bytes = 4TB
  - And the vast majority of entries are empty/invalid!
- Page table indirection adds significant overhead to all memory accesses

Page Tables are Slow

```
0x1024 mov [edi + eax * 4], 0x0
0x1028 inc eax
0x102C cmp eax, 0x03E8
0x1030 jne 0x1024
```

- How many memory accesses occur during each iteration of the loop?
  - – 4 instructions are read from memory
  - – [edi + eax * 4] writes to one location in memory
  - – 5 page table lookups
    - Each memory access must be translated
    - … and the page tables themselves are in memory
- Naïve page table implementation doubles memory access overhead

- Motivation and Goals
- Base and Bounds
- Segmentation
- Page Tables
- TLB
- Multi-level Page Tables
- Swap Space

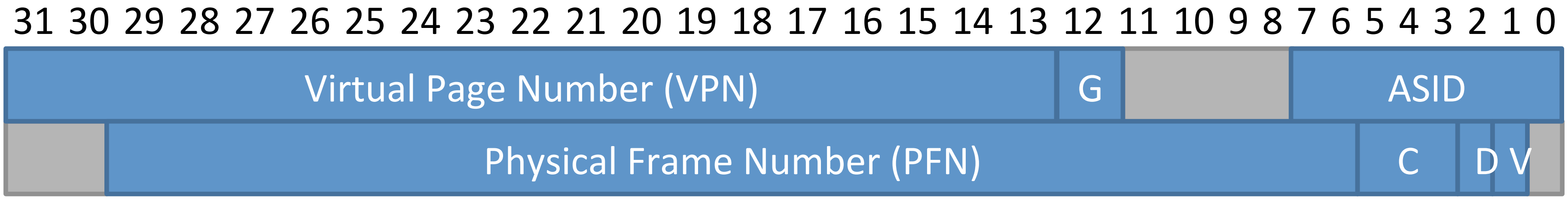
Problem: Page Table Speed

- Page tables give us a great deal of flexibility and granularity to implement virtual memory
- However, page tables are large, thus they must go in RAM (as opposed to in a CPU register)
  - Each virtual memory access must be translated
  - Each translation requires a table lookup in memory
  - Thus, memory overhead is doubled
- How can we use page tables without this memory lookup overhead?

Caching

- Key idea: cache page table entries directly in the CPU's MMU
  - Translation Lookaside Buffer (TLB)
  - Should be called *address translation cache*
- TLB stores recently used PTEs
  - Subsequent requests for the same virtual page can be filled from the TLB cache
- Directly addresses speed issue of page tables
  - On-die CPU cache is very, very fast
  - Translations that hit the TLB don't need to be looked up from the page table in memory

Example TLB Entry



31 30 29 28 27 26 25 24 23 22 21 20 19 18 17 16 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0

Virtual Page Number (VPN) | G | ASID

Physical Frame Number (PFN) | C | D | V

- VPN & PFN – virtual and physical pages
- G – is this page global (i.e. accessible by all processes)?
- ASID – address space ID
- D – dirty bit – has this page been written recently?
- V – valid bit – is this entry in the TLB valid?
- C – cache coherency bits – for multi-core systems

More on this later…

TLB Control Flow Psuedocode

```
VPN = (VirtualAddress & VPN_MASK) >> SHIFT
(Success, TlbEntry) = TLB_Lookup(VPN)
if (Success == True) // TLB Hit
    if (CanAccess(TlbEntry.ProtectBits) == True)
        Offset = VirtualAddress & OFFSET_MASK
        PhysAddr = (TlbEntry.PFN << SHIFT) | Offset
        AccessMemory(PhysAddr)
    else
        RaiseException(PROTECTION_FAULT)
else // TLB Miss
    PTEAddr = PTBR + (VPN * sizeof(PTE))
    PTE = AccessMemory(PTEAddr)
    if (PTE.Valid == False)
        RaiseException(SEGMENTATION_FAULT)
    else if (CanAccess(PTE.ProtectBits) == False)
        RaiseException(PROTECTION_FAULT)
    TLB_Insert(VPN, PTE.PFN, PTE.ProtectBits)
    RetryInstruction()
```

Fast Path

Make sure we have permission, then proceed

Slow Path

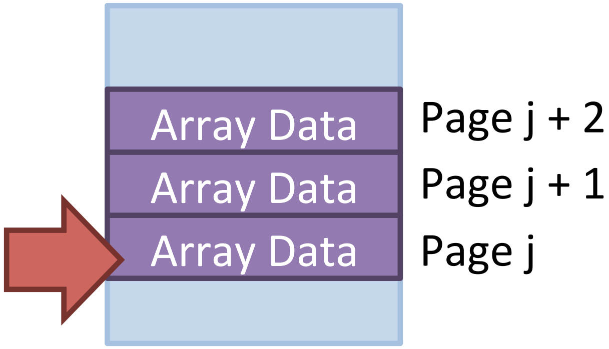
Load the page table entry from memory, add it to the TLB, and retry

Reading an Array (no TLB)

- Suppose we have a 10KB array of integers
  - Assume 4KB pages
- With no TLB, how many memory accesses are required to read the whole array?
  - 10KB / 4 = 2560 integers in the array
  - Each requires one page table lookup, one memory read
  - 5120 reads, plus more for the instructions themselves

Reading an Array (with TLB)

Process 1's View of Virtual Memory



TLB

VPN	PFN

- Same example, now with TLB
  - 10KB integer array
  - 4KB pages
  - Assume the TLB starts off cold (i.e. empty)
- How many memory accesses to read the array?
  - 2560 to read the integers
  - 3 page table lookups
  - 2563 total reads
  - TLB hit rate: 96%

Locality

- TLB, like any cache, is effective because of locality
  - **Spatial locality**: if you access memory address *x*, it is likely you will access *x + 1* soon
    - Most of the time, *x* and *x + 1* are in the same page
  - **Temporal locality**: if you access memory address *x*, it is likely you will access *x* again soon
    - The page containing *x* will still be in the TLB, hopefully

Be Careful With Caching

- Recall: TLB entries have an ASID (address space ID) field. What is this for?
  - Here's a hint: think about context switching

Process 1's Page Table

VPN	PFN
i	d
j	b
k	a

TLB

VPN	PFN
i	d
J	b
k	a

Process 2 Page Table

VPN	PFN
i	r
j	u
k	s

- Problem: TLB entries may not be valid after a context switch

VPNs are the same, but PFN mappings have changed!

Potential Solutions

1. Clear the TLB (mark all entries as invalid) after each context switch
   - Works, but forces each process to start with a cold cache
   - Only solution on x86 (until ~2008)
2. Associate an ASID (address space ID) with each process
   - ASID is just like a process ID in the kernel
   - CPU can compare the ASID of the active process to the ASID stored in each TLB entry
   - If they don't match, the TLB entry is invalid

Replacement Policies

- On many CPUs (like x86), the TLB is managed by the hardware
- Problem: space in the TLB is limited (usually KB)
  - Once the TLB fills up, how does the CPU decide what entries to replace (evict)?
- Typical replacement policies:
  - FIFO: easy to implement, but certain access patterns result in worst-case TLB hit rates
  - Random: easy to implement, fair, but suboptimal hit rates
  - LRU (Least Recently Used): algorithm typically used in practice

Hardware vs. Software Management

- Thus far, discussion has focused on hardware managed TLBs (e.g. x86)

```
PTE = AccessMemory(PTEAddr)
TLB_Insert(VPN, PTE.PFN, PTE.ProtectBits)
```

  - CPU dictates the page table format, reads page table entries from memory
  - CPU manages all TLB entries
- However, software managed TLBs are also possible (e.g. MIPS and SPARC)

Software Managed TLB Pseudocode

```
VPN = (VirtualAddress & VPN_MASK) >> SHIFT
(Success, TlbEntry) = TLB_Lookup(VPN)
if (Success == True) // TLB Hit
    if (CanAccess(TlbEntry.ProtectBits) == True)
                        ET_MASK
                        HIFT) | Offset
                        Addr)
    else
        RaiseException(PROTECTION_FAULT)
else // TLB Miss
    RaiseException(TLB_MISS)
```

The hardware does not:

1. Try to read the page table
2. Add/remove entries from the TLB

Implementing Software TLBs

- Key differences vs. hardware managed TLBs
  - CPU doesn't insert entries into the TLB
  - CPU has no ability to read page tables from memory
- On TLB miss, the OS must handle the exception
  - Locate the correct page table entry in memory
  - Insert the PTE into the TLB (evict if necessary)
  - Tell the CPU to retry the previous instruction
- Note: TLB management instructions are privileged
  - Only the kernel can modify the TLB

Comparing Hardware and Software TLBs

Hardware TLB

- Advantages
  - Less work for kernel developers, CPU does a lot of work for you
- Disadvantages
  - Page table data structure format must conform to hardware specification
  - Limited ability to modify the CPUs TLB replacement policies

Easier to program

Software TLB

- Advantages
  - No predefined data structure for the page table
  - OS is free to implement novel TLB replacement policies
- Disadvantages
  - More work for kernel developers
  - Beware infinite TLB misses!
    - OSes page fault handler must always be present in the TLB

Greater flexibility

TLB Summary

- TLBs address the slowdown associated with page tables
  - Frequently used page table entries are cached in the CPU
  - Prevents repeated lookups for PTEs in main memory
- Reduce the speed overhead of page tables by an order of magnitude or more
  - Caching works very well in this particular scenario
  - Lots of spatial and temporal locality

- Motivation and Goals
- Base and Bounds
- Segmentation
- Page Tables
- TLB
- Multi-level Page Tables
- Swap Space

Problem: Page Table Size

- At this point, we have solved the TLB speed issue
- However, recall that pages tables are large and sparse
  - Example: 32-bit system with 4KB pages
  - Each page table is 4MB
  - Most entries are invalid, i.e. the space is wasted
- How can we reduce the size of the page tables?
  - Many possible solutions
  - Multi-layer page tables are most common (x86)

Simple Solution: Bigger Pages

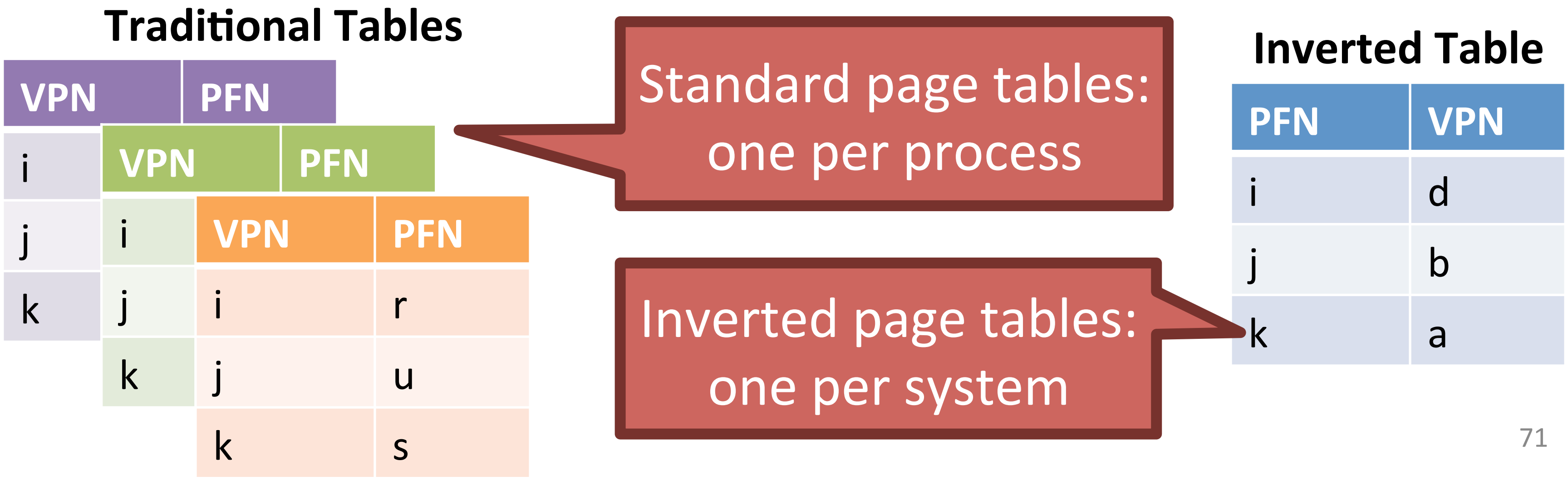
- Suppose we increase the size of pages
  - Example: 32-bit system, 4MB pages
  - $2^{32} / 2^{22}$ = 1024 pages per process
  - 1024 * 4 bytes per page = 4KB page tables
- What is the drawback?
  - Increased internal fragmentation
  - How many programs actually have 4MB of code, 4MB of stack, and 4MB of heap data?

Alternate Data Structures

- Thus far, we've assumed linear page tables
  - i.e. an array of page table entries
- What if we switch to an alternate data structure?
  - Hash table
  - Red-black tree
- Why is switching data structures not always feasible?
  - Can be done if the TLB is software managed
  - If the TLB is hardware managed, then the OS must use the page table format specified by the CPU

Inverted Page Tables

- Our current discussion focuses on tables that map virtual pages to physical pages
- What if we flip the table: map physical pages to virtual pages?
  - Since there is only one physical memory, we only need one inverted page table!



Traditional Tables

VPN	PFN
i	
j	
k	

VPN	PFN
i	
j	
k	

VPN	PFN
i	r
j	u
k	s

Standard page tables: one per process

Inverted Table

PFN	VPN
i	d
j	b
k	a

Inverted page tables: one per system

Normal vs. Inverted Page Tables

- Advantage of inverted page table
  - Only one table for the whole system
- Disadvantages
  - Lookups are more computationally expensive

Table must be scanned to locate a given VPN, thus O(n) lookup time

VPN serves as an index into the array, thus O(1) lookup time

Traditional Table

VPN	PFN
i	d
j	b
k	a

Inverted Table

PFN	VPN
i	d
j	b
k	a

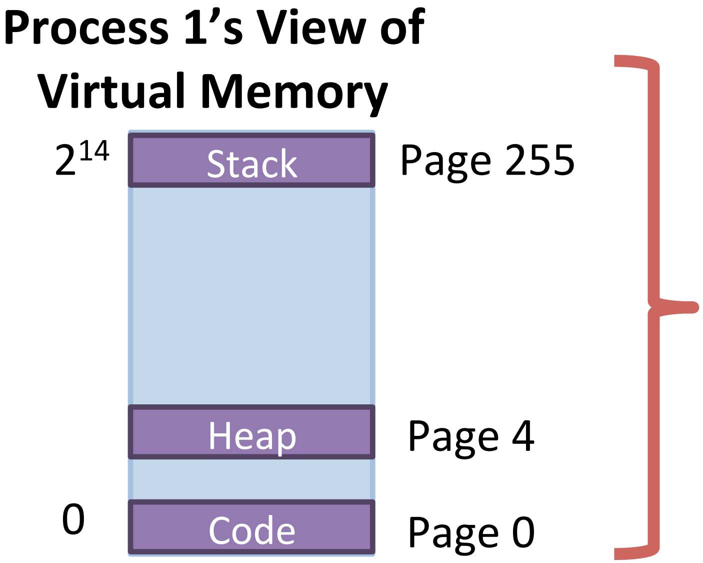
- How to implement shared memory?

Multi-Level Page Tables

- Key idea: split the linear page table into a tree of sub-tables
  - Benefit: branches of the tree that are empty (i.e. do not contain valid pages) can be pruned
- Multi-level page tables are a space/time tradeoff
  - Pruning reduces the size of the table (saves space)
  - But, now the tree must be traversed to translate virtual addresses (increased access time)
- Technique used by modern x86 CPUs
  - 32-bit: two-level tables
  - 64-bit: four-level tables

Multi-Level Table Toy Example

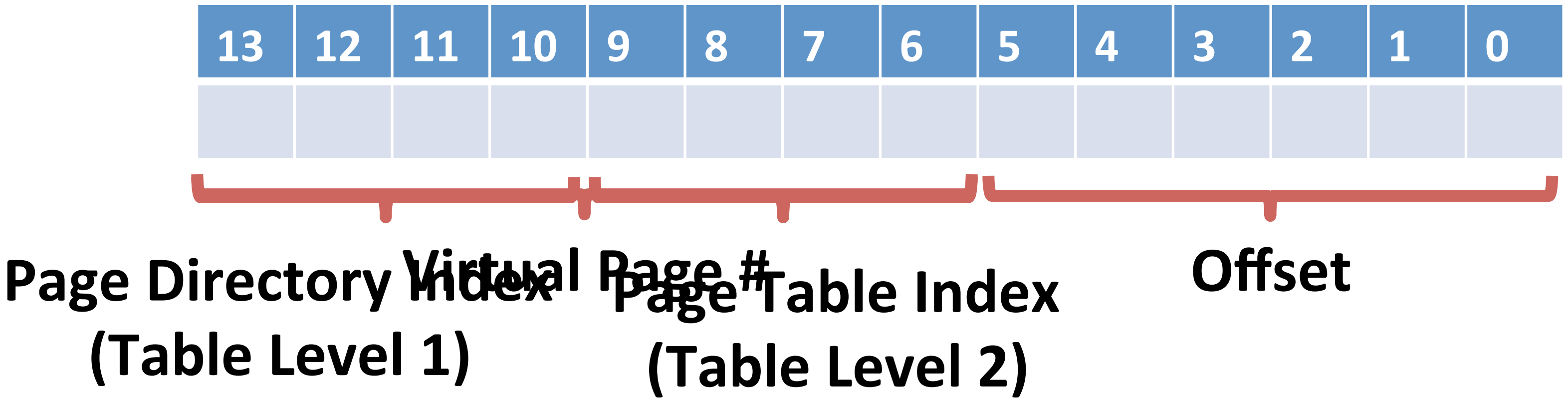
- Imagine a small, 16KB address space
  - 64-byte pages, 14-bit virtual addresses, 8 bits for the VPN and 6 for the offset
- How many entries does a linear page table need?
  - $2^8 = 256$ entries



Assume 3 pages out of 256 total pages are in use

From Linear to Two-levels Tables

- How do you turn a linear table into a multi-level table?
  - Break the linear table up into page-size units
- 256 table entries, each is 4 bytes large
  - 256 * 4 bytes = 1KB linear page tables
- Given 64-byte pages, a 1KB linear table can be divided into 16 64-byte tables
  - Each sub-table holds 16 page table entries



13 12 11 10 9 8 7 6 5 4 3 2 1 0

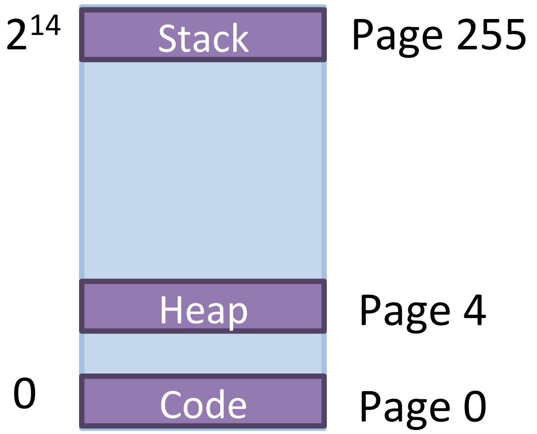
Virtual Page #

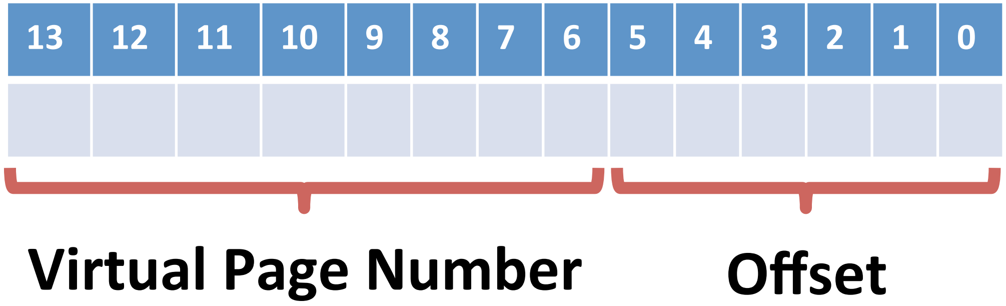
Page Directory Index
(Table Level 1)

Page Table Index
(Table Level 2)

Offset

Process 1's View of Virtual Memory



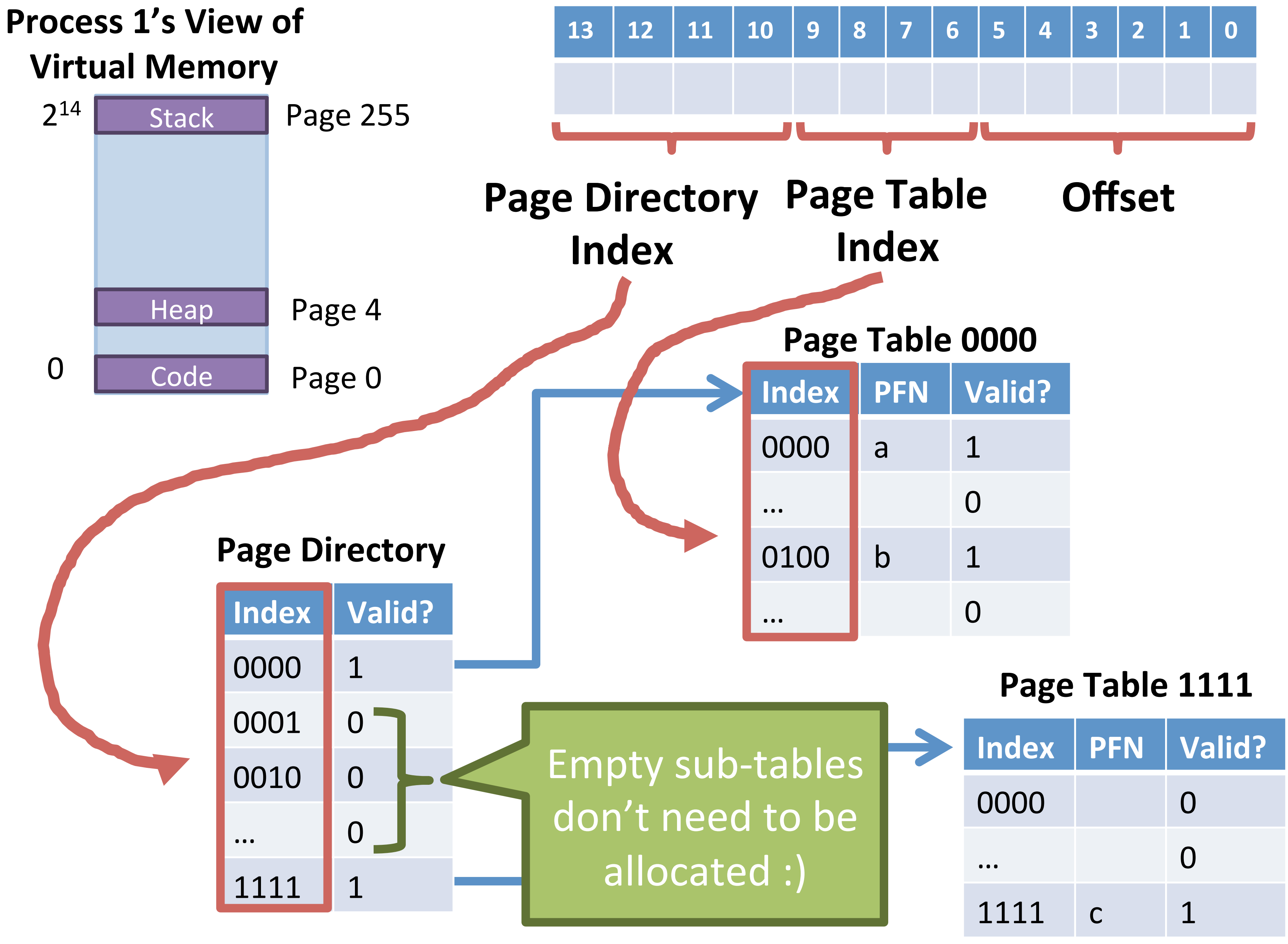


Linear Page Table

VPN	PFN	Valid?
00000000	a	1
…		0
00000100	b	1
…		0
11111111	c	1

253 tables entries are empty, space is wasted :(

Process 1's View of Virtual Memory



13	12	11	10	9	8	7	6	5	4	3	2	1	0

Page Directory Index

Page Table Index

Offset

Page Table 0000

Index	PFN	Valid?
0000	a	1
…		0
0100	b	1
…		0

Page Directory

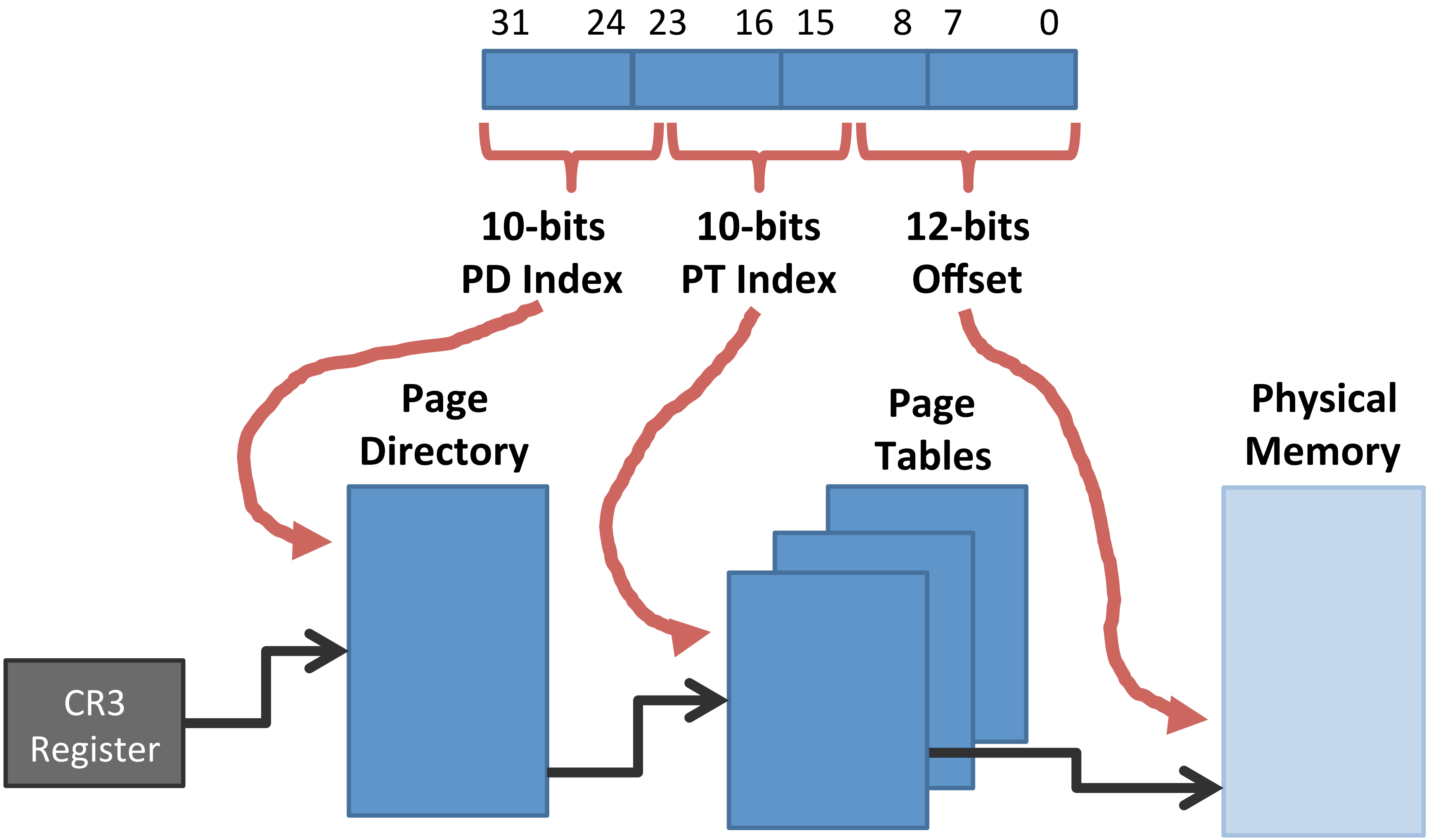
Index	Valid?
0000	1
0001	0
0010	0
…	0
1111	1

Empty sub-tables don't need to be allocated :)

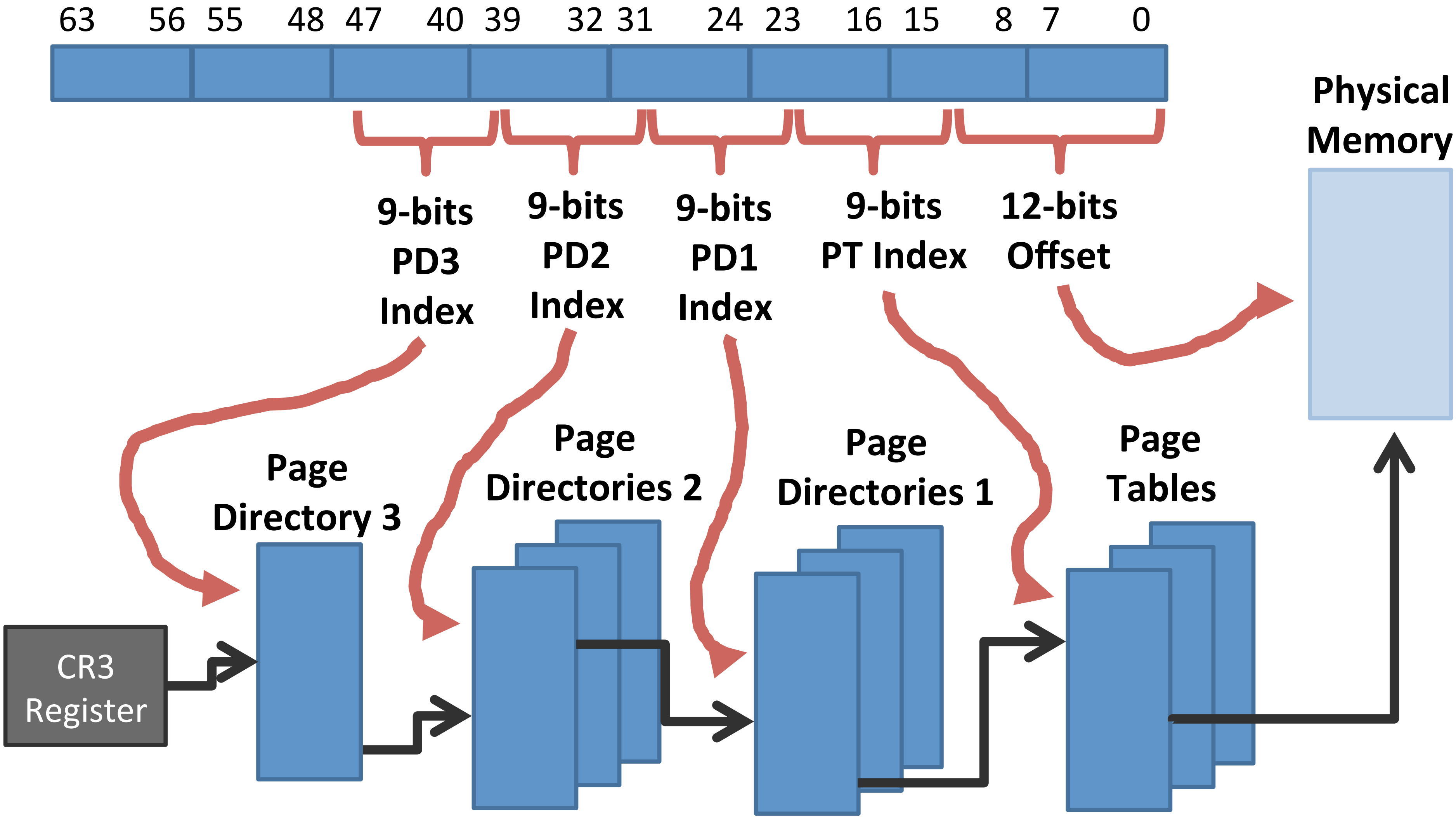
Page Table 1111

Index	PFN	Valid?
0000		0
…		0
1111	c	1

32-bit x86 Two-Level Page Tables



64-bit x86 Four-Level Page Tables



Don’t Forget the TLB

- Multi-level pages look complicated
  - And they are, but only when you have to traverse them
- The TLB still stores VPN → PFN mappings
  - TLB hits avoid reading/traversing the tables at all

Multi-Level Page Table Summary

- Reasonably effective technique for shrinking the size of page tables
  - Implemented by x86
- Canonical example of a space/time tradeoff
  - Traversing many levels of table indirection is slower than using the VPN as an index into a linear table
  - But, linear tables waste a lot of space

- Motivation and Goals
- Base and Bounds
- Segmentation
- Page Tables
- TLB
- Multi-level Page Tables
- Swap Space

Status Check

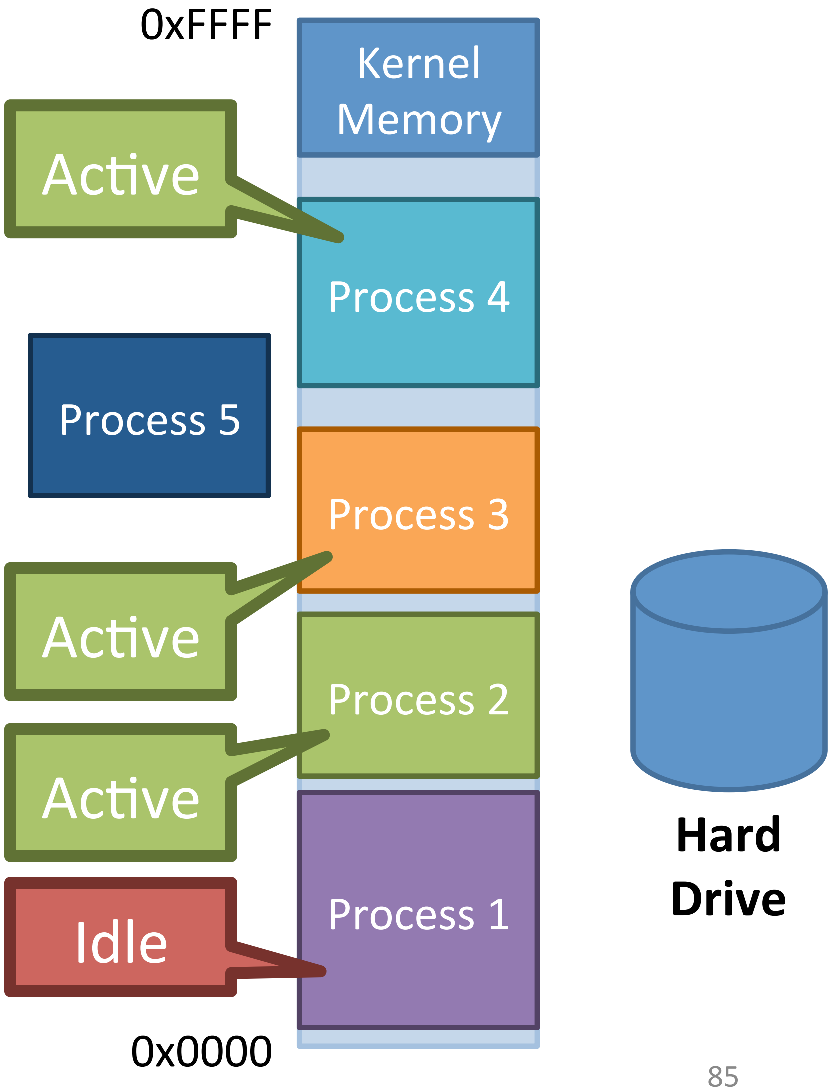
- At this point, we have a full-featured virtual memory system
  - Transparent, supports protection and isolation
  - Fast (via TLBs)
  - Space efficient (via multi-level tables)
- Are we done?
  - No!
- What if we completely run out of physical memory?
  - Can virtualization help?

Swap Space

- Key idea: take frames from physical memory and swap (write) them to disk
  - This frees up space for other code and data
- Load data from swap back into memory on-demand
  - If a process attempts to access a page that has been swapped out…
  - A page-fault occurs and the instruction pauses
  - The OS can swap the frame back in, insert it into the page table, and restart the instruction

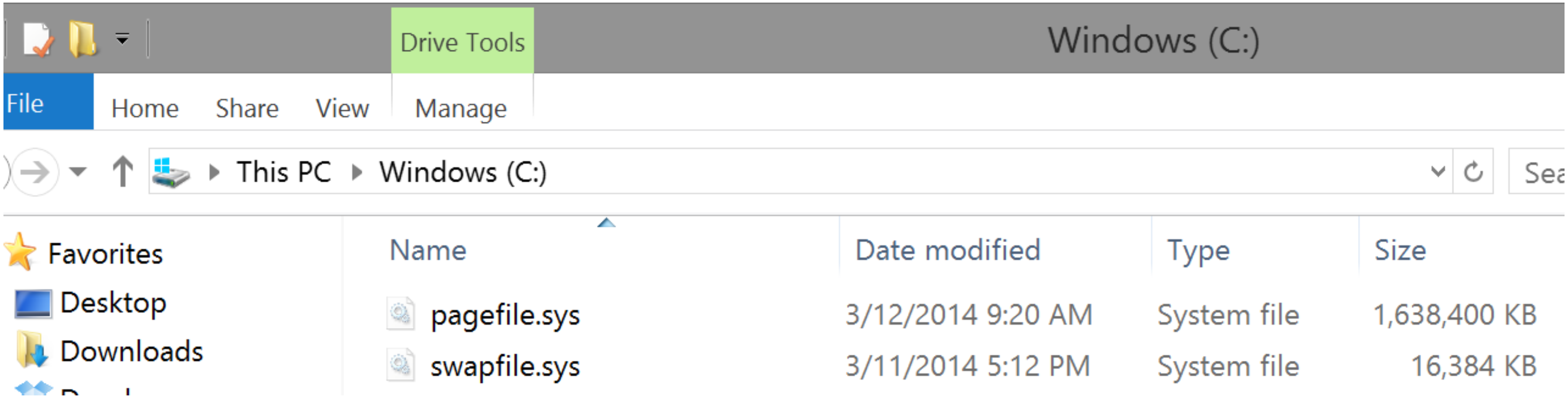
Swapping Example

- Suppose memory is full
- The user opens a new program
- Swap out idle pages to disk
- If the idle pages are accessed, page them back in



All Modern OSes Support Swapping

- On Linux, you create a *swap partition* along with your normal ext3/4 filesystem
  - – Swapped pages are stored in this separate partition
- Windows



Name	Date modified	Type	Size
pagefile.sys	3/12/2014 9:20 AM	System file	1,638,400 KB
swapfile.sys	3/11/2014 5:12 PM	System file	16,384 KB

Implementing Swap

1. Data structures are needed to track the mapping between pages in memory and pages on disk
2. Meta-data about memory pages must be kept
   - When should pages be evicted (swapped to disk)?
   - How do you choose which page to evict?
3. The functionality of the OSes page fault handler must be modified

x86 Page Table Entry, Again

- On x86, page table entries (PTE) are 4 bytes

31 - 12	11 - 9	8	7	6	5	4	3	2	1	0
Page Frame Number (PFN)	Unused	G	PAT	D	A	PCD	PWT	U/S	W	P

- P – present bit – is this page in physical memory?
  - – OS sets or clears the present bit based on its swapping decisions
    - 1 means the page is in physical memory
    - 0 means the page is valid, but has been swapped to disk
  - – Attempts to access an invalid page **or** a page that isn't present trigger a page fault

Handling Page Faults

- Thus far, we have viewed page faults as bugs
  - i.e. when a process tries to access an invalid pointer
  - The OS kills the process that generate page faults
- However, now handling page faults is more complicated
  - If the PTE is invalid, the OS still kills the process
  - If the PTE is valid, but present = 0, then
    1. The OS swaps the page back into memory
    2. The OS updates the PTE
    3. The OS instructs the CPU to retry the last instruction

When Should the OS Evict Pages?

- Memory is finite, so when should pages be swapped?
- On-demand approach
  - If a page needs to be created and no free pages exist, swap a page to disk
- Proactive approach
  - Most OSes try to maintain a small pool of free pages
  - Implement a high watermark
  - Once physical memory utilization crosses the high watermark, a background process starts swapping out pages

What Pages Should be Evicted?

- Known as the page-replacement policy
- What is the optimal eviction strategy?
  - Evict the page that will be accessed **furthest in the future**
  - Provably results in the maximum cache hit rate
  - Unfortunately, impossible to implement in practice
- Practical strategies for selecting which page to swap to disk
  - FIFO
  - Random
  - LRU (Least recently used)
- Same fundamental algorithms as in TLB eviction

Examples of Optimal and LRU

Assume the cache can store 3 pages

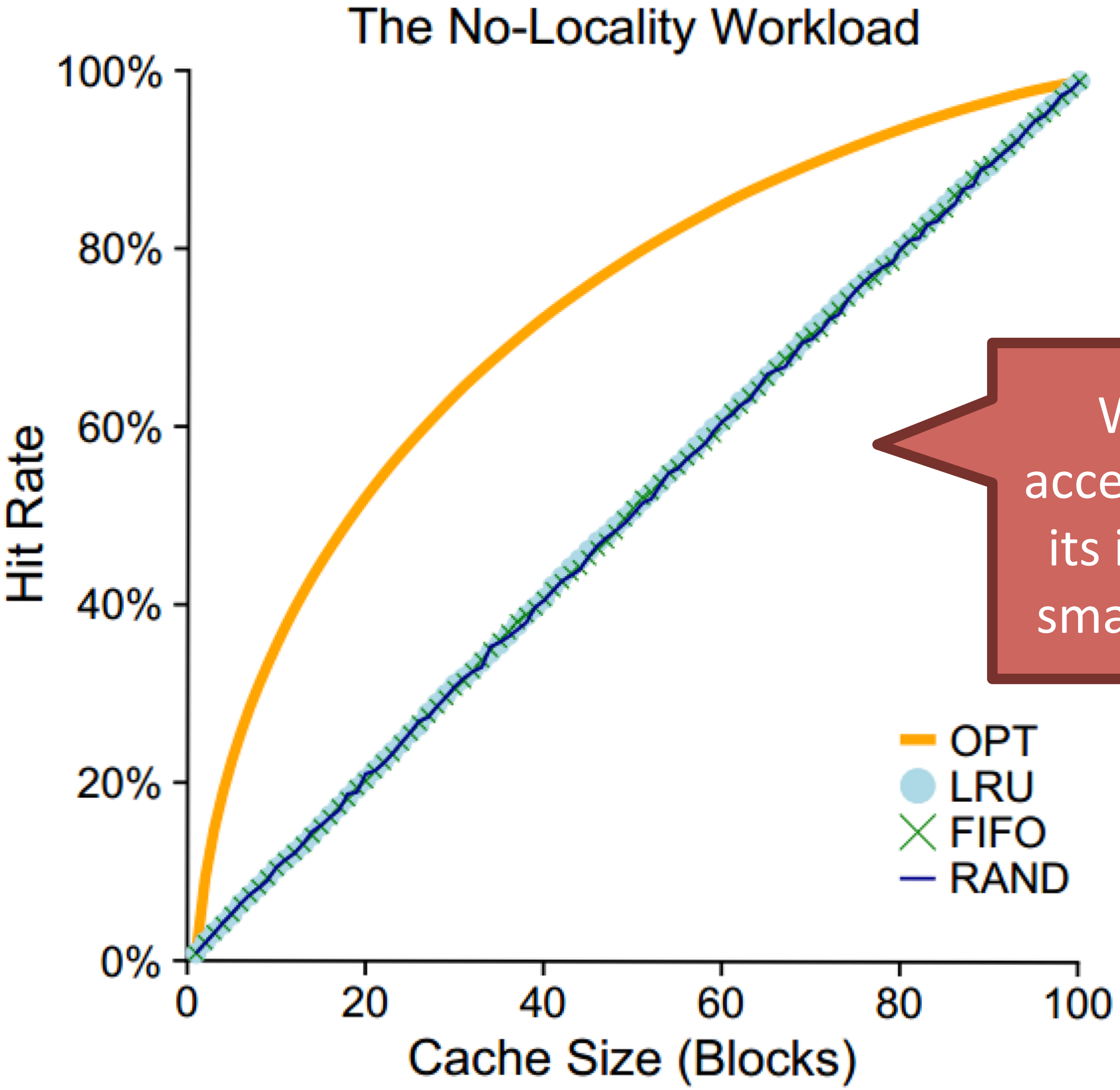
Optimal (Furthest in the Future)

Access	Hit/Miss?	Evict	Cache State
0	Miss		0
1	Miss		0, 1
2	Miss		0, 1, 2
0	Hit		0, 1, 2
1	Hit		0, 1, 2
3	Miss	2	0, 1, 3
0	Hit		0, 1, 3
3	Hit		0, 1, 3
1	Hit		0, 1, 3
2	Miss	3	0, 1, 2
0	Hit		0, 1, 2

LRU

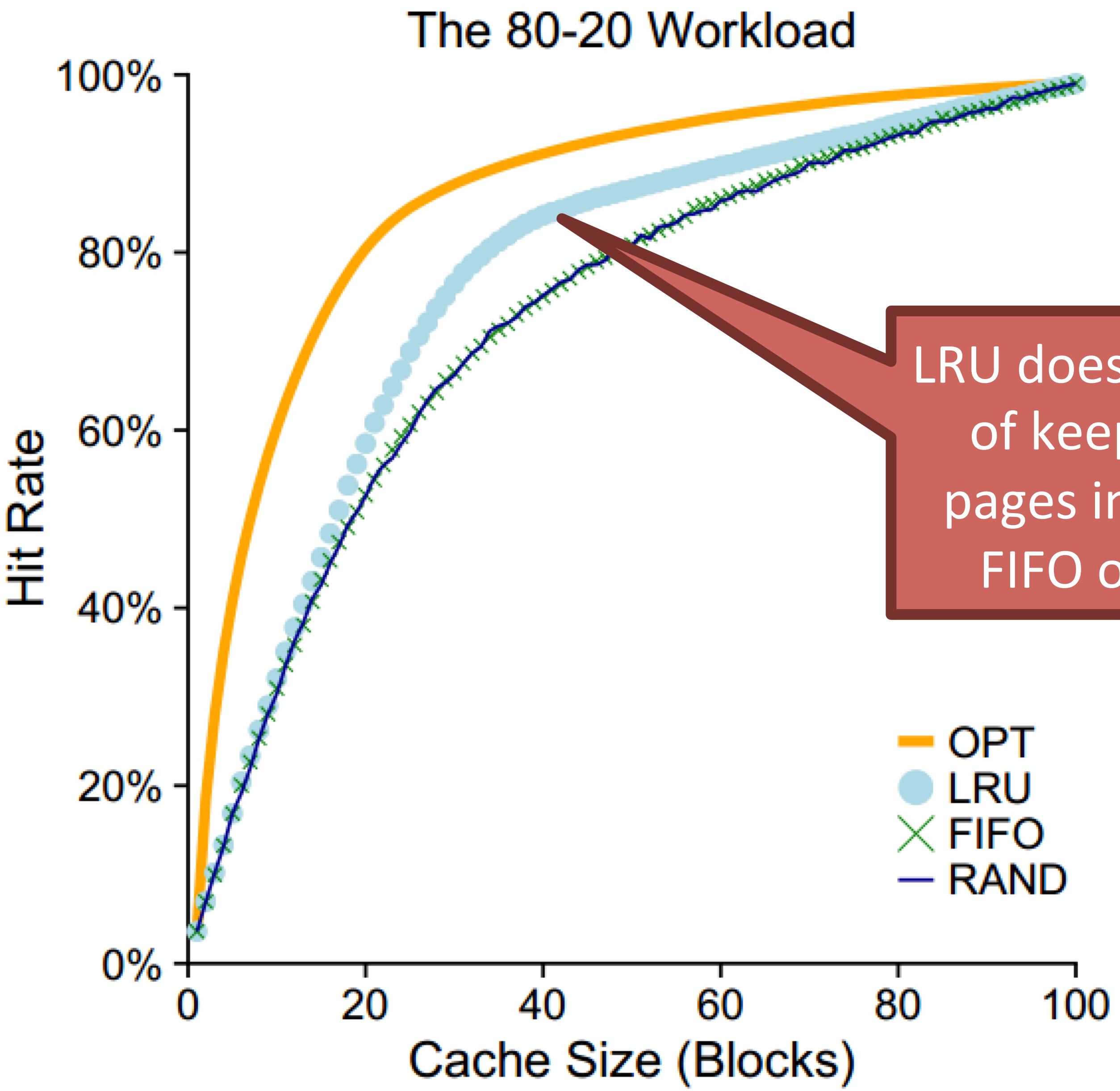
Access	Hit/Miss?	Evict	Cache State
0	Miss		0
1	Miss		0, 1
2	Miss		0, 1, 2
0	Hit		0, 1, 2
1	Hit		0, 1, 2
3	Miss	2	0, 1, 3
0	Hit		0, 1, 3
3	Hit		0, 1, 3
1	Hit		0, 1, 3
2	Miss	0	1, 2, 3
0	Miss	3	0, 1, 2

- All memory accesses are to 100% random pages



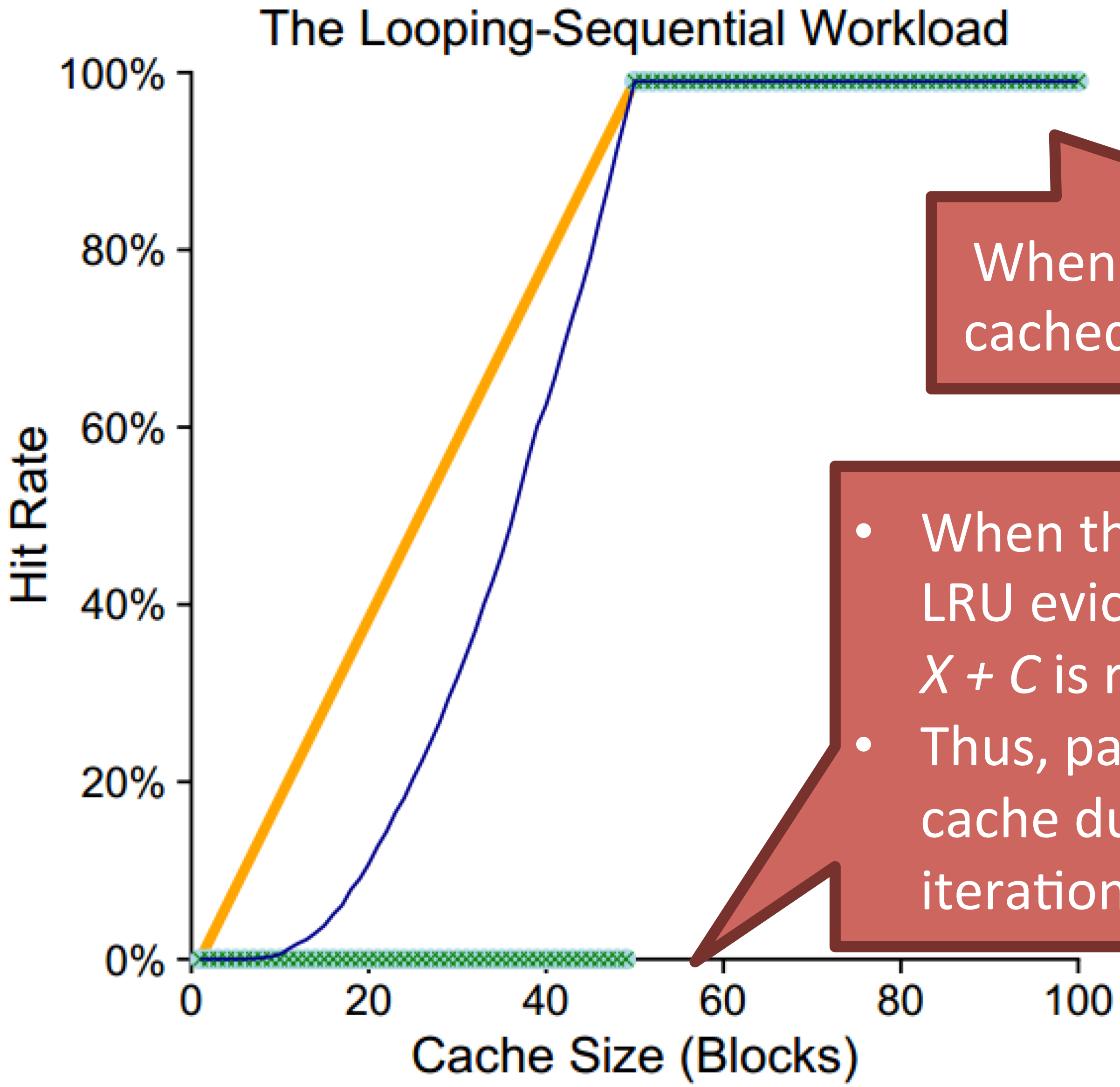
When memory accesses are random, its impossible to be smart about caching

- 80% of memory accesses are for 20% of pages



LRU does a better job of keeping “hot” pages in RAM than FIFO or random

- The process sequentially accesses one memory address in 50 pages, then loops



When C >= 50, all pages are cached, thus hit rate = 100%

- When the cache size is *C* < 50, LRU evicts page *X* when page *X + C* is read
- Thus, pages are not in the cache during the next iteration of the loop

Implementing Historical Algorithms

- LRU has high cache hit rates in most cases…
- … but how do we know which pages have been recently used?
- Strategy 1: record each access to the page table
  - Problem: adds additional overhead to page table lookups
- Strategy 2: approximate LRU with help from the hardware

x86 Page Table Entry, Again

- On x86, page table entries (PTE) are 4 bytes

31 - 12	11 - 9	8	7	6	5	4	3	2	1	0
Page Frame Number (PFN)	Unused	G	PAT	D	A	PCD	PWT	U/S	W	P

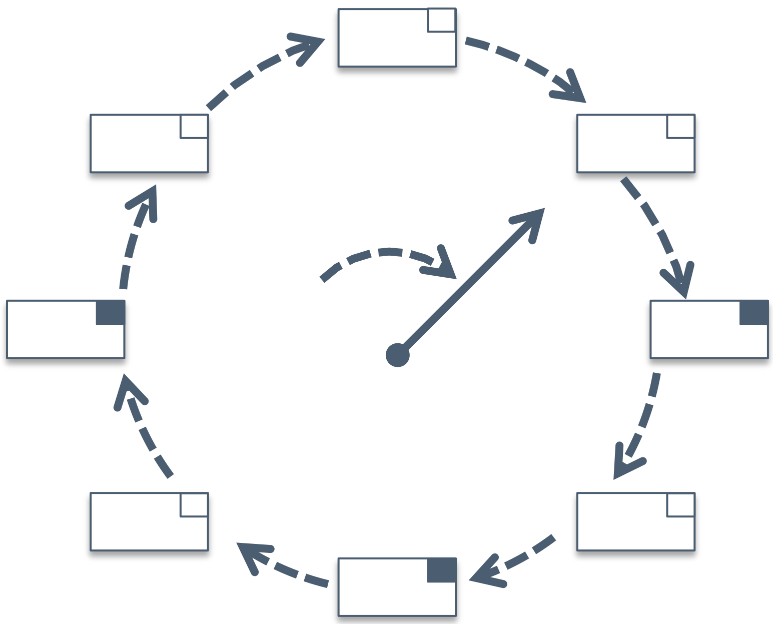
- Bits related to swapping
  - A – accessed bit – has this page been read recently?
  - D – dirty bit – has this page been written recently?
  - The MMU sets the accessed bit when it reads a PTE
  - The MMU sets the dirty bit when it writes to the page referenced in the PTE
  - The OS may clear these flags as it wishes

Approximating LRU

- The accessed and dirty bits tell us which pages have been recently accessed
- But, LRU is still difficult to implement
  - On eviction, LRU needs to scan all PTEs to determine which have not been used
  - But there are millions of PTEs!
- Is there a clever way to approximate LRU without scanning all PTEs?
  - Yes!

The Clock Algorithm

- Imagine that all PTEs are arranged in a circular list
- The clock hand points to some PTE *P* in the list



```
function clock_algo() {
    start = P;
    do {
        if (P.accessed == 0) {
            evict(P);
            return;
        }
        P.accessed = 0;
        P = P.next;
    } while (P != start);
    evict_random_page();
}
```

Incorporating the Dirty Bit

- More modern page eviction algorithms also take the dirty bit into account
- For example: suppose you must evict a page, and all pages have been accessed
  - Some pages are read-only (like code)
  - Some pages have been written too (i.e. they are dirty)
- Evict the non-dirty pages first
  - In some cases, you don't have to swap them to disk!
  - Example: code is already on the disk, simply reload it
- Dirty pages must always be written to disk
  - Thus, they are more expensive to swap

RAM as a Cache

- RAM can be viewed as a high-speed cache for your large-but-slow spinning disk storage
  - You have GB of programs and data
  - Only a subset can fit in RAM at any given time
- Ideally, you want the most important things to be resident in the cache (RAM)
  - Code/data that become less important can be evicted back to the disk