A3: What You Always Wanted To Know But Were Afraid To Ask

But first . . .

- Unconfusing Three Confusions
  - Where does the kernel live?
  - Does every kind of processor use a two-level page table?
  - Does a pointer have to refer to dynamically-allocated memory?

Where is the Kernel’s Address Space?

- Each process has a virtual address space, but where is the kernel’s virtual address space?
  - Separate virtual address space: change page tables on entry into privileged mode; change them again on the way out
  - Physical space: disable automatic hardware translation of virtual addresses on entry into privileged mode; re-enable on exit
  - Privileged region in each process’s virtual address space: Use page table or segment protections to protect kernel virtual memory from user-mode accesses
- Third approach used by Linux, Windows, OS161, 64-bit Mac OS X
  - Makes it easy for kernel to examine arguments in system calls, and return values to user-level
- 32-bit Mac OS uses a separate virtual address space for kernel
  - I don’t want to talk about it

OK LET’S TALK ABOUT IT CALM DOWN

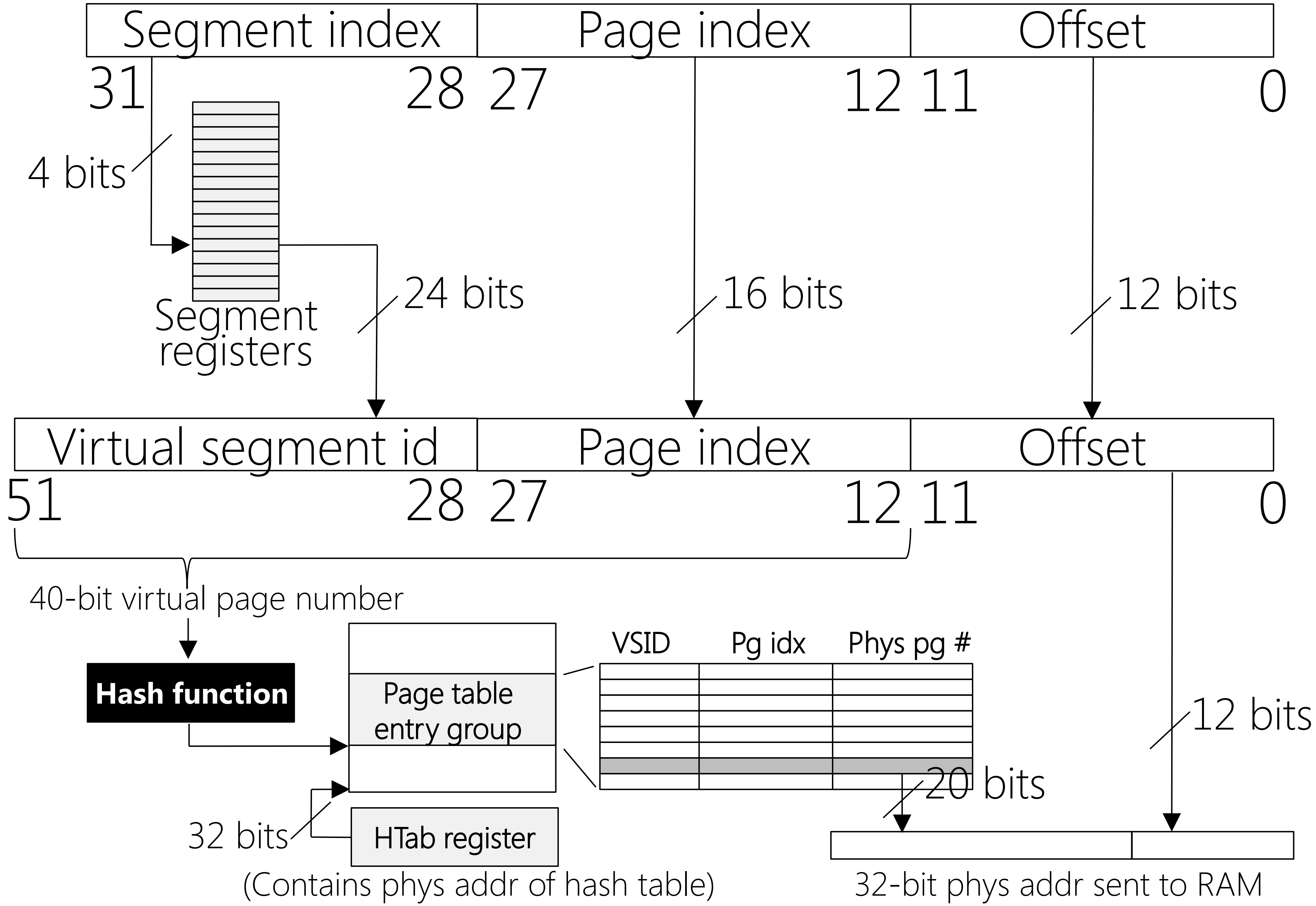
32-bit Mac OS X (pre-10.4)

- Kernel has 32-bit virtual address space, just like a regular process
  - Page table protections prevent a regular process from modifying the address space of the kernel (or any other regular process!)
- Good: Entire 4GB address space available to user processes
- Bad: context switches are more expensive (TLB flushes—the kernel is never "already there"!)
- Bad: copyin()/copyout() are trickier—can't just do a paranoid memcpy()
  - OS X solution: In kernel address space, reserve 0XE0000000—0xFFFFFFFF for "user memory window"
  - On system call, after context switch to kernel address space, use PTE trickeration to map kernel's user memory window to the memory region of user process that contains system call arguments (and will eventually contain the return value)

Q: Does every processor use N-level page tables?

A: No! With software-defined page tables, designs can be arbitrarily interesting. Even with hardware-defined page tables, N-level page tables are not the only option.

Case study: Page Tables on 32-bit PowerPC



Everything In Memory Has An Address!

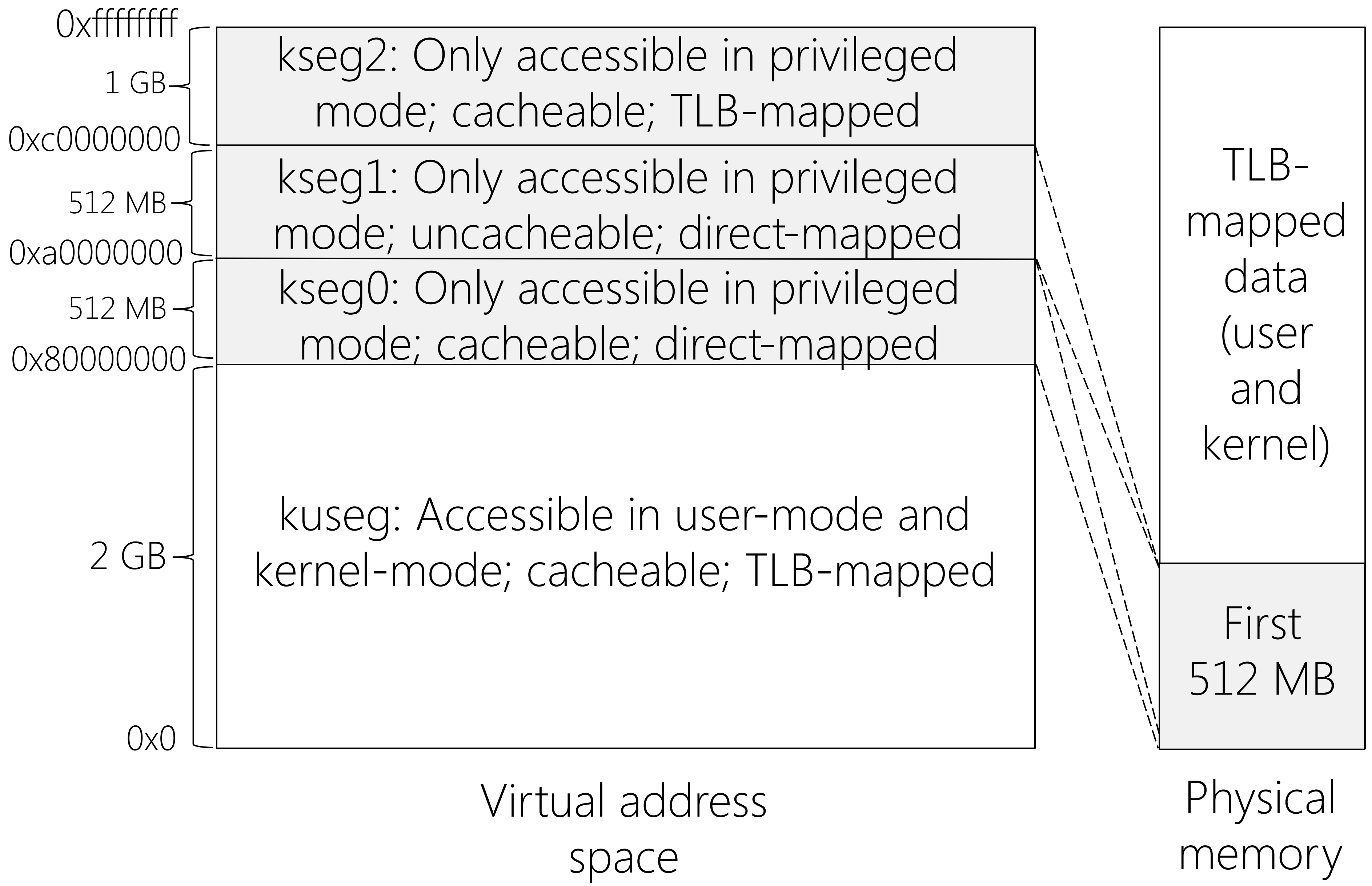
```
#include <stdio.h>
#include <stdlib.h>
int main(int argc, char *argv[]){
    printf(“Location of code: %p\n”, (void *) main);
    printf(“Location of heap: %p\n”, (void *) malloc(1));
    int x = 3;
    printf(“Location of stack: %p\n”, (void *) &x);
    return 0;
}
```

```
Location of code :          0x40057d
Location of heap :         0x12f9010
Location of stack : 0x7ffca580a02c
```

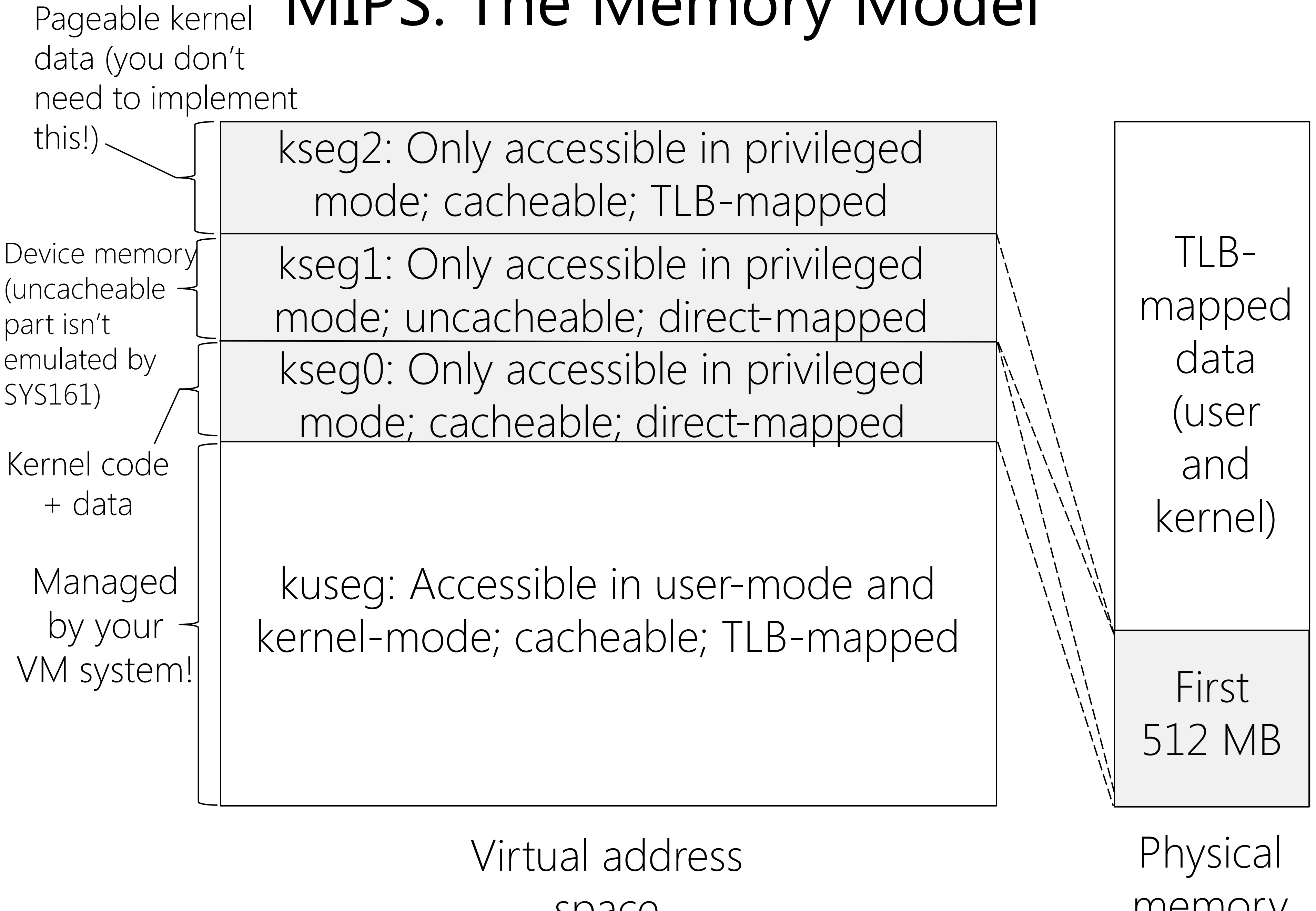
Deep-dive on Assignment 3

- Review of MIPS memory model
- Overview of your tasks in Assignment 3
- Case study: Swapping on Linux
- Case study: Inside the heap created by dlmalloc

MIPS: The Memory Model


0xffffffff
1 GB
0xc0000000
512 MB
0xa0000000
512 MB
0x80000000
2 GB
0x0
kseg2: Only accessible in privileged mode; cacheable; TLB-mapped
kseg1: Only accessible in privileged mode; uncacheable; direct-mapped
kseg0: Only accessible in privileged mode; cacheable; direct-mapped
kuseg: Accessible in user-mode and kernel-mode; cacheable; TLB-mapped
TLB-mapped data (user and kernel)
First 512 MB
Virtual address space
Physical memory

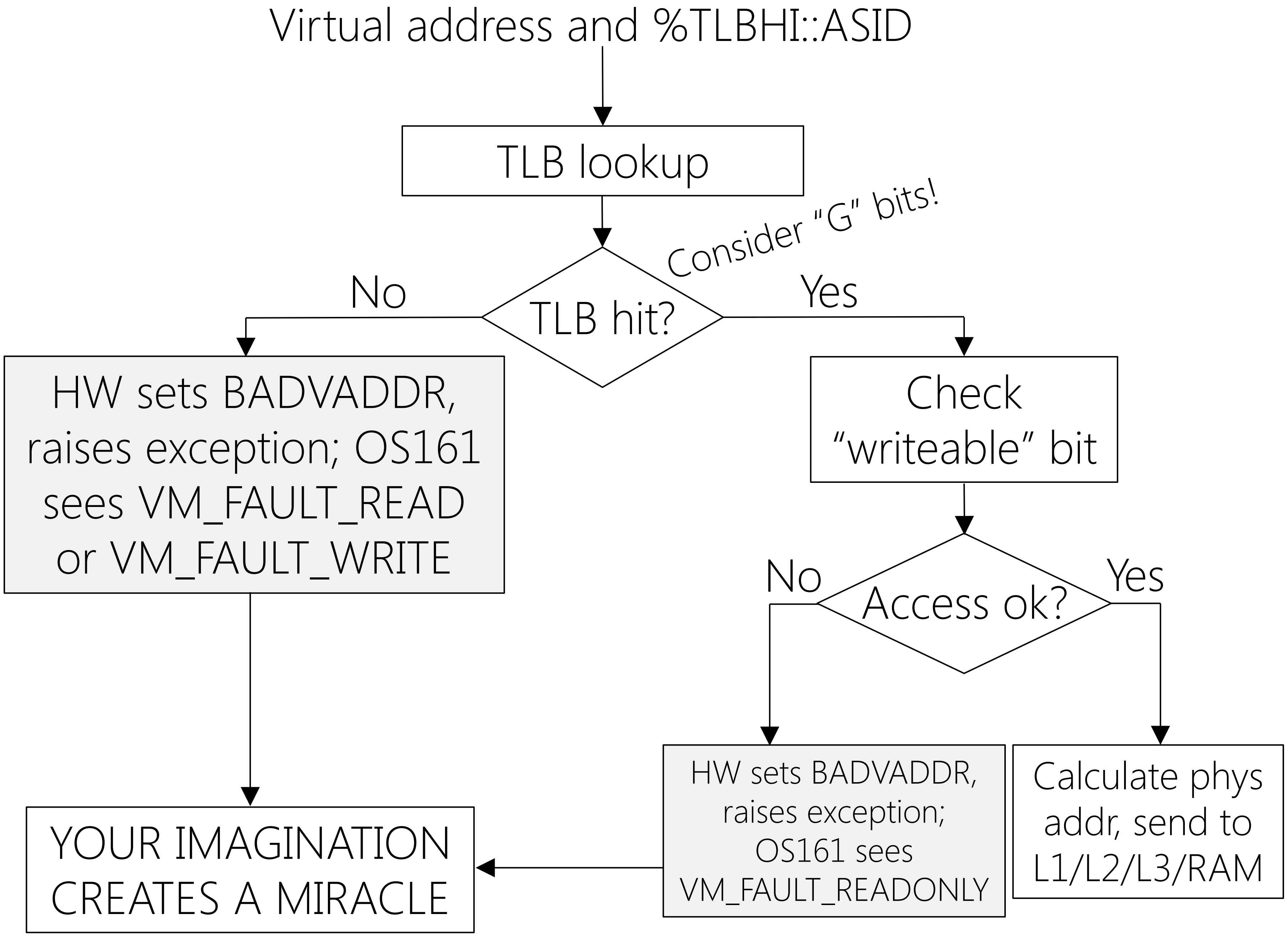
MIPS: The Memory Model


Pageable kernel data (you don't need to implement this!)
Device memory (uncacheable part isn't emulated by SYS161)
Kernel code + data
Managed by your VM system!
kseg2: Only accessible in privileged mode; cacheable; TLB-mapped
kseg1: Only accessible in privileged mode; uncacheable; direct-mapped
kseg0: Only accessible in privileged mode; cacheable; direct-mapped
kuseg: Accessible in user-mode and kernel-mode; cacheable; TLB-mapped
TLB-mapped data (user and kernel)
First 512 MB
Virtual address space
Physical memory

Assignment 3: Your Mission

- Handle TLB faults
- Implement paging
  - Per-process data structures (e.g., page tables)
  - Global data structures (e.g., core map: physical page number -> virtual page info)
  - Page eviction + backing store support
  - Background writing of dirty pages to disk
- sbrk()

The Lifecycle of a Memory Reference on MIPS


Virtual address and %TLBHI::ASID
TLB lookup
TLB hit?
Consider "G" bits!
No
Yes
HW sets BADVADDR, raises exception; OS161 sees VM_FAULT_READ or VM_FAULT_WRITE
Check "writeable" bit
Access ok?
No
Yes
HW sets BADVADDR, raises exception; OS161 sees VM_FAULT_READONLY
Calculate phys addr, send to L1/L2/L3/RAM
YOUR IMAGINATION CREATES A MIRACLE

TLB Handling

- For an additional reference on the MIPS TLB architecture, see the Vahalia reference on the CS161 “Resources” page
- Start with a simple replacement algorithm first!
- We provide four C-level functions to interact with the TLB:
  - TLB_Write(): Write to a specified TLB entry
  - TLB_Read(): Read a specified TLB entry
  - TLB_Probe(): Search TLB to see if it contains a match for a given virtual frame number
  - TLB_Random(): Write to a random TLB entry
- Note that TLB_Random() never selects 8 of the 64 TLB entries, so you may want to use TLB_Write() and random()
- Suggestion: Ignore ASIDs and “G” bit for now; just clear the entire TLB on a context switch (less efficient, but correct!)

Paging and Virtual Memory

- Bootstrapping is tricky
  - You cannot kmalloc() until you set up your memory system . . .
  - . . . but you cannot set up your memory system without allocating kernel memory
  - Hint: Look at how ram_stealmem() works
- Key data structures for paging:
  - Per-process virtual-to-physical mappings
  - Global mapping from physical pages to a process and an address space
- Think carefully about the page tables!
  - How much memory do they consume?
  - Do they require linear searches? (The answer should be “no”!)

Handling a Page Fault for Page P

- Check page table, confirm that P exists
- If so, decide where to put P
    - If there's free memory, use it! (Hint: consult the core map)
    - If there isn't free memory:
        - Select a frame to evict
        - Write it to the backing store if necessary
        - Update page tables
- Read P into memory
- Update page tables
- Update TLB

Synchronization

Address Space Manipulations

- Operations on address spaces
  - as_create()
  - as_destroy()
  - as_copy()   //For fork()
  - as_activate() //For context switching
- Code isn't too bad—the main challenges are data structures and synchronization
  - Often best to have data structures synchronize themselves . . .

```
int foo_manipulator(){
    lock foo;
    manipulate foo;
    unlock foo;
}

//Somewhere else
ret = foo_manipulator();
```

```
int foo_manipulator(){
    manipulate foo;
}

//Somewhere else
lock foo;
ret = foo_manipulator();
unlock foo;
```

Kernel Allocations

- Hint: Don’t try to implement pageable kernel memory!
  - So, when you allocate a page to the kernel, it stays allocated unless the kernel gives it back
  - When the kernel asks for N pages of contiguous virtual address space, you need to find N pages of contiguous *physical* memory!

Backing Store

- You need a pager thread that proactively writes dirty pages to disk (making them clean)
- Hint: You should never sleep while holding a spinlock!
- Hint: Every page can have its own place on disk
    - You can make your disk quite large
    - OS161 already provides bitmap functionality (useful for determining which disk blocks are free)
    - Use vfs_open() on “lhd0raw:” and use the vnode you get back for swapping

Linux Case Study: Swapping

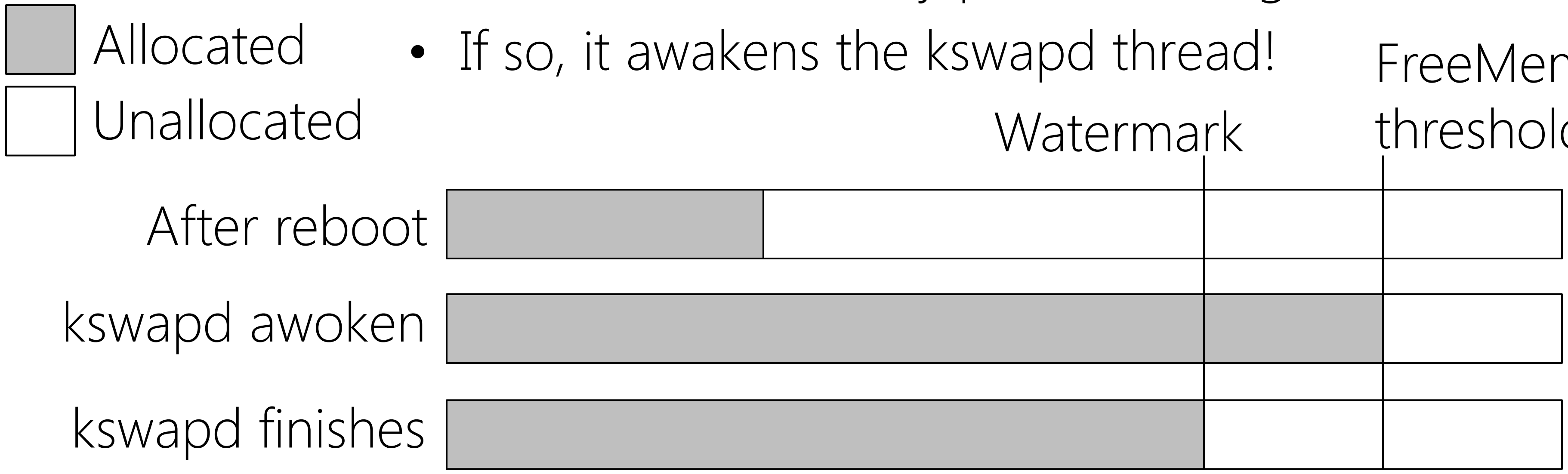
- Linux page cache lives in physical RAM, and has a bunch of stuff, including:

  “Mapped”
  - Disk/SSD data that has been read/written
  - Code pages from user process’s virtual address space

  “Anonymous”
  - Stack and heap pages from user process’s virtual address space
- When memory pressure is low, kernel is lazy about removing data—why spend the effort if somebody might need the data in the future?
- Kernel must evict pages from RAM when memory pressure is high
  - Dirty mapped pages written to backing files
  - Dirty anonymous pages written to swap space

Linux Case Study: Swapping

- Linux has a kswapd thread for each processor
  - Allows for parallel memory reclamation
  - Useful for NUMA machines in which some RAM is "close" to one core and "far" from others (kswapd thread will focus on pages in "close" RAM)
- Each kswapd thread sleeps on a wait queue
  - When the kernel allocates memory, it checks whether the memory pressure is high
  - If so, it awakens the kswapd thread!



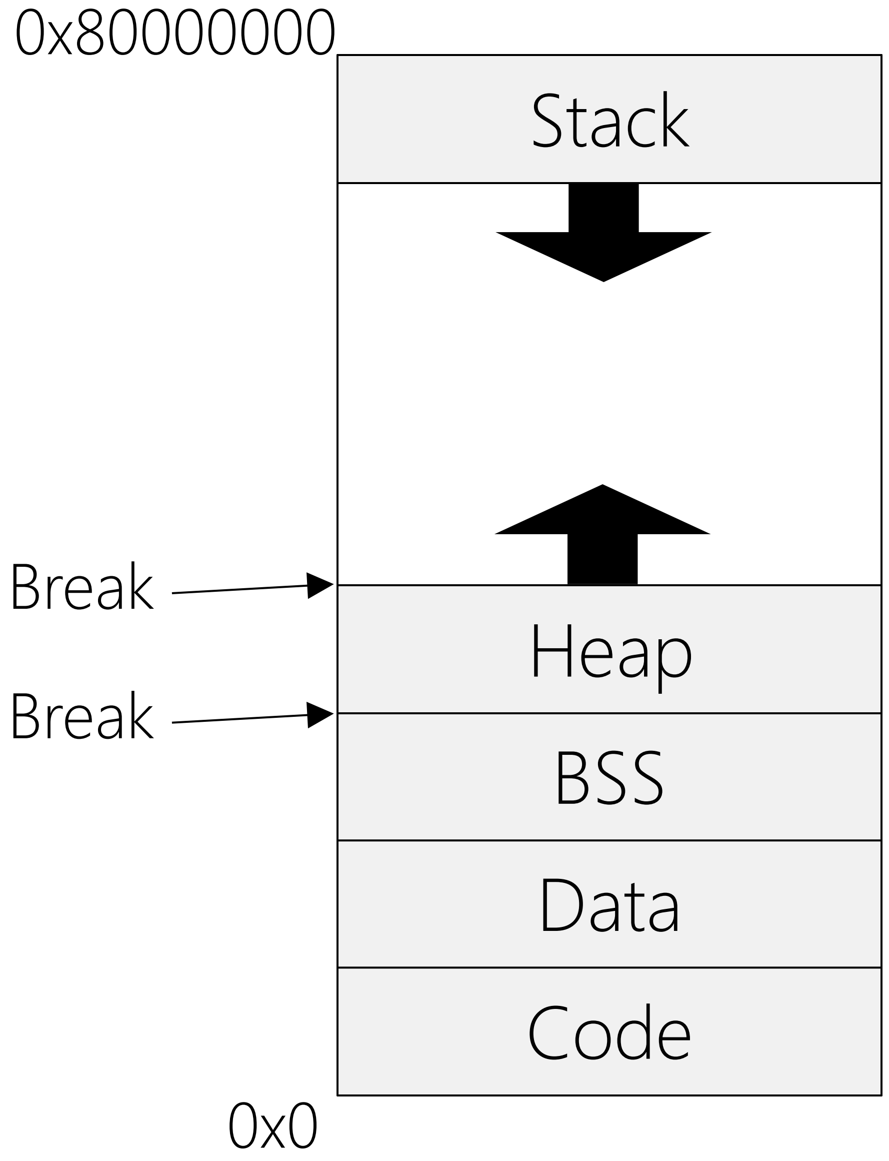
Linux Case Study: Swapping

- Linux uses LRU to determine which pages to remove
  - For each process, kernel maintains two page lists: active and inactive
  - On x86, leverages the "Accessed" PTE bit that's automatically updated by hardware
  - When memory pressure is high, kernel evicts pages from the inactive list
- Q: What if all the disk buffers are evicted and swap space is filled, but there's still memory pressure?
- A: The OOM killer uses heuristics to kill processes and reclaim their memory + swap space
  - Hate processes w/lots of allocated virtual memory
  - Hate processes w/low static priority
  - DO NOT HATE KERNEL TASKS LIKE INIT
  - Do not hate processes w/direct access to hardware

Synchronization in your OS161 VM Subsystem

- Things to consider
  - SPL synchronization doesn’t work with IO!
  - Don’t create a lock per page, since this would consume too much space; consider a busy bit
  - How does locking work when handling a page fault?
  - How does locking work when evicting a page?
  - What if a page that I want to modify/evict is in the middle of being evicted by someone else?
  - What happens in fork() if a page to copy is not resident?
- Holland’s hint: It is easier to debug a VM system with deadlocks than with race conditions
- Think carefully about synchronization in your design doc!

malloc() and sbrk()



- The “break” refers to the end of a process’s heap memory
- Before a process has dynamically allocated memory, the break is after the static code+data
- As the application calls malloc(), the break pointer moves upward
- . . . but what exactly is in the heap?

THE HEAP IS FILLED WITH REVERSE CENTAURS

The Reverse Centaur
↳ the upper-body of a horse on a man's body
approximately 10 feet tall
can whinny
sweet hair/mane
extremely top-heavy
hooves make archery near-impossible
un-saddle-able
has trouble shopping for turtle necks
can scratch its own butt unlike normal centaurs
legs supporting over 1000 lbs.
10 toes for optimum balance

THE HEAP IS FILLED WITH OCAML

```
[1;2;3]
```

```
let x = 42
```

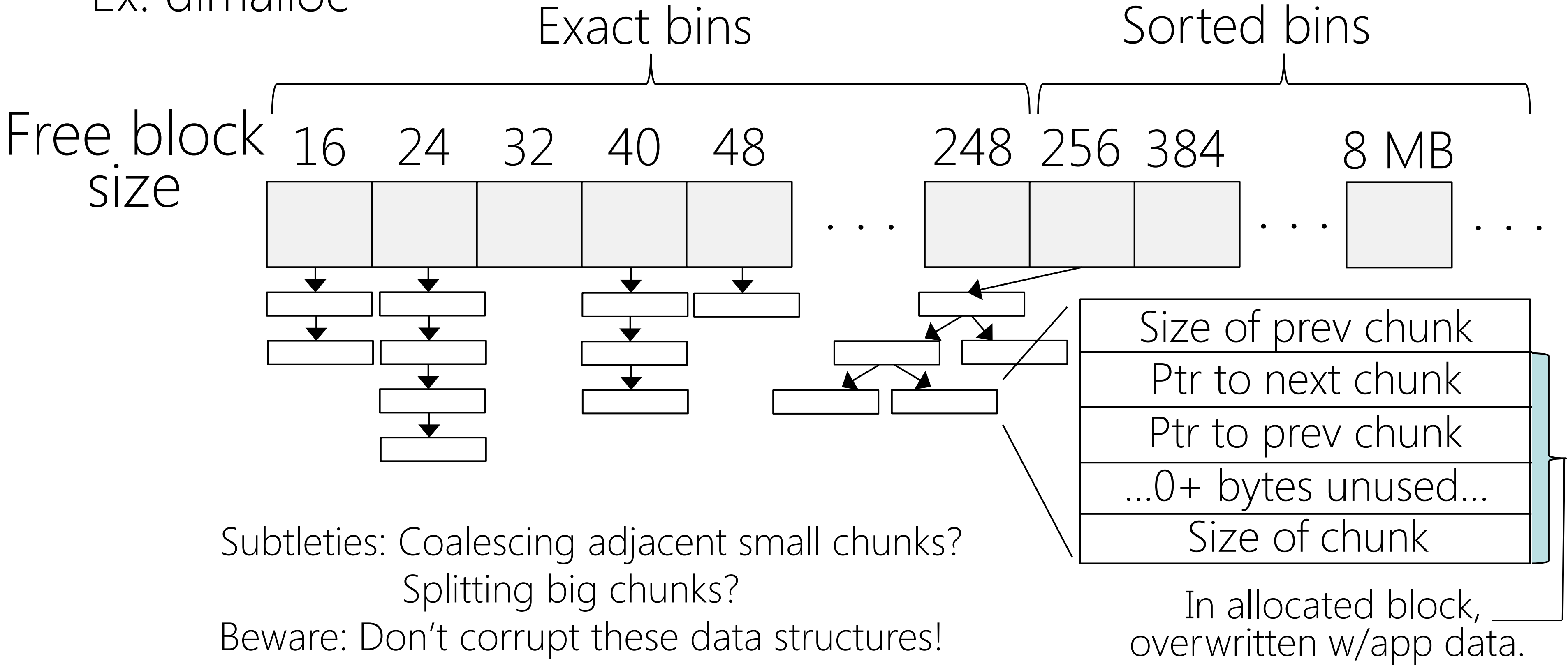
```
let what_i_want now = {
   side_effects = “yes”;
}
```

```
[“OCaml”; “Erlang”;
 “Other languages I
    don’t use”]
```

```
(fun x -> (x/0, “LOL”))
```

THE HEAP IS FILLED WITH DATA STRUCTURES

- malloc()/free() implementations track which parts of the heap are allocated, and which are unallocated
- Ex: dlmalloc



Subtleties: Coalescing adjacent small chunks?
Splitting big chunks?
Beware: Don't corrupt these data structures!

- OS161's malloc()/free() is much simpler, but you must understand it for Assignment 3; read it before implementing sbrk()!

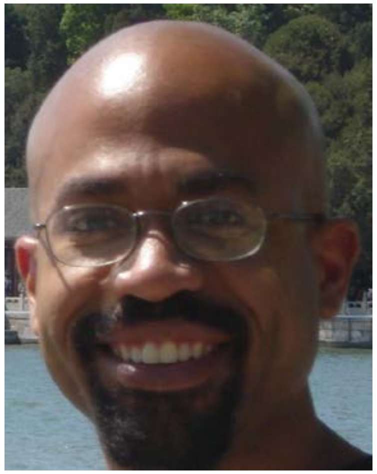
VM Statistics

- Statistics will help with debugging and performance debugging!
- Real-life virtual memory managers collect many stats, e.g.,:
  - Total number of physical pages available
  - Total number of physical pages allocated
  - Number of clean in-memory pages
  - Number of dirty in-memory pages
  - Number of kernel pages
  - Number of swapped-out pages
  - Your brilliant statistic goes here

Things That You Don't Have To Do

- Copy-on-write memory (e.g., for use by fork())
- Pageable kernel memory (using kseg2)
- Memory-mapped files (e.g., to minimize read()/write() overheads)
- 22-disk swap partitions (e.g., to support hilariously high levels of memory pressure)

Margo's Mantra:
Get something simple working first! Only consider fancier stuff afterwards.



James's Mantra:
LISTEN TO MARGO.

In-class Exercises

- Where does OS161 provide bitmap functionality? Find the location and familiarize yourself with the code—it will come in handy during Assignment 3!
- When an sbrk() happens, what per-process data structure(s) must be updated? When the stack grows, what per-process data structure(s) must be updated? Does anything need to happen when the stack *shrinks* due to user-mode function calls returning?
- Think about how a multicore architecture interacts with virtual memory management. For example:
    - Do you need to do anything special with page tables and TLB entries if a process executes on core X, is context-switched off, and later resumed on core Y?
    - Suppose that core R discovers that it must evict a page from memory. What implications does this have for the TLB on core R? What implications does this have for the TLBs on other cores?
    - Suppose that, while fork() executes on one core, the other core is running the kernel thread which asynchronously writes dirty pages to disk. What synchronization strategies can ensure that both operations proceed safely?