The Beast We Call A3

CS 161: Lecture 10
3/7/17

But first . . .

- Unconfusing Three Confusions
  - Where does the kernel live?
  - Does every kind of processor use a two-level page table?
  - Does everything have an address?

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

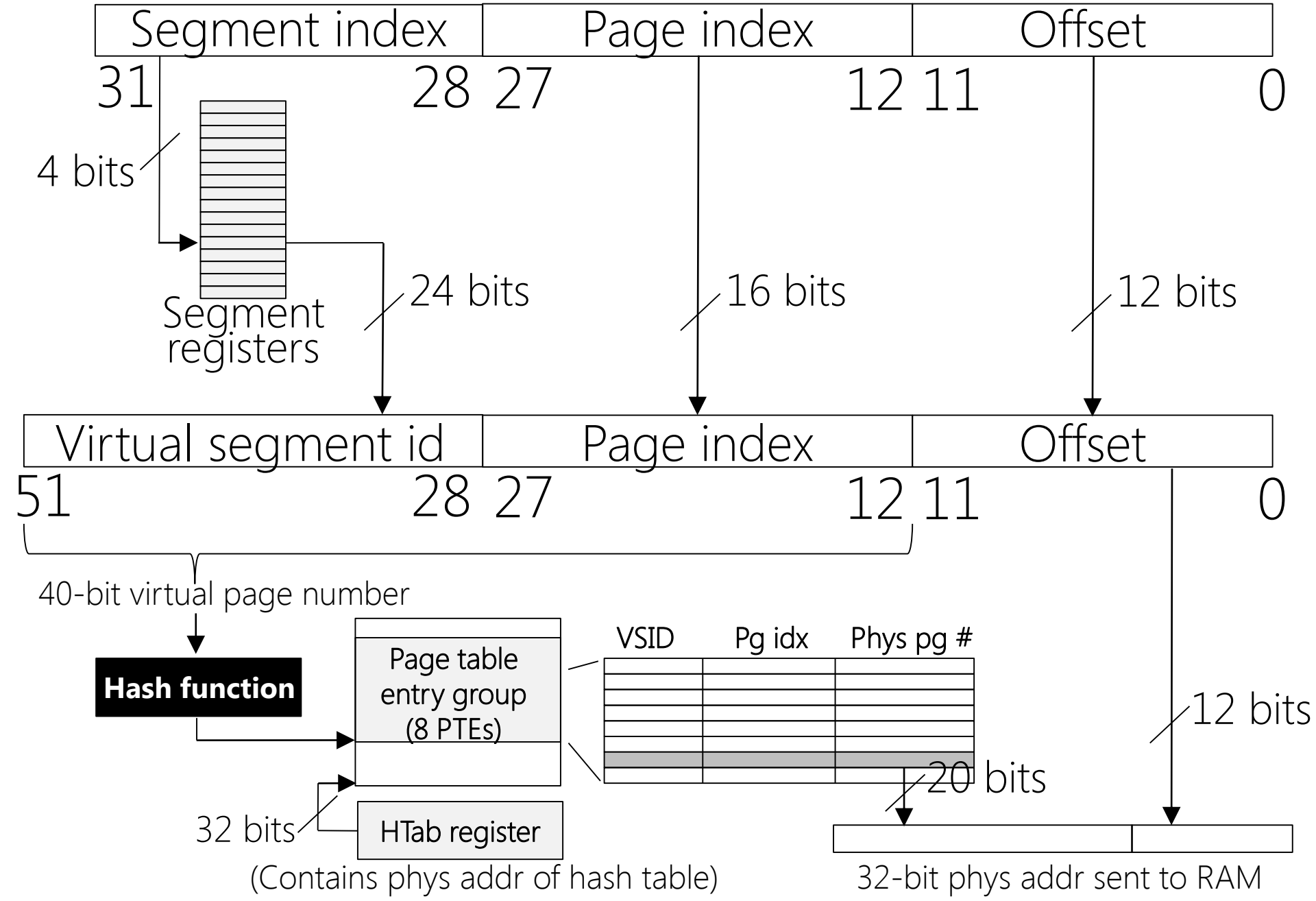
32-bit Mac OS X (pre-10.4)

- Kernel has 32-bit virtual address space, just like a regular process
  - MMU-enforced isolation prevents 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



Linus hates PowerPC page tables

- Linus argued that:
  - TLBs are getting large . . .
  - . . . which means that hit rates are getting better, once the TLB has been warmed . . .
  - . . . so warming the TLB (i.e., "compulsory misses" that pull mappings from page tables in RAM) must be fast!
- PowerPC's hash tables have poor spatial locality: adjacent virtual page numbers usually hash to non-adjacent PTEGs in physical RAM
  - So, on the two compulsory misses for two adjacent virtual page numbers, the first miss will pull in a cache line that likely only contains the PTEG for first miss
  - In contrast, with a standard N-level page table, a compulsory TLB miss brings in a cache line with info for adjacent virtual pages; so, second compulsory TLB miss will hopefully hit in cache

Everything In Memory Has An Address!

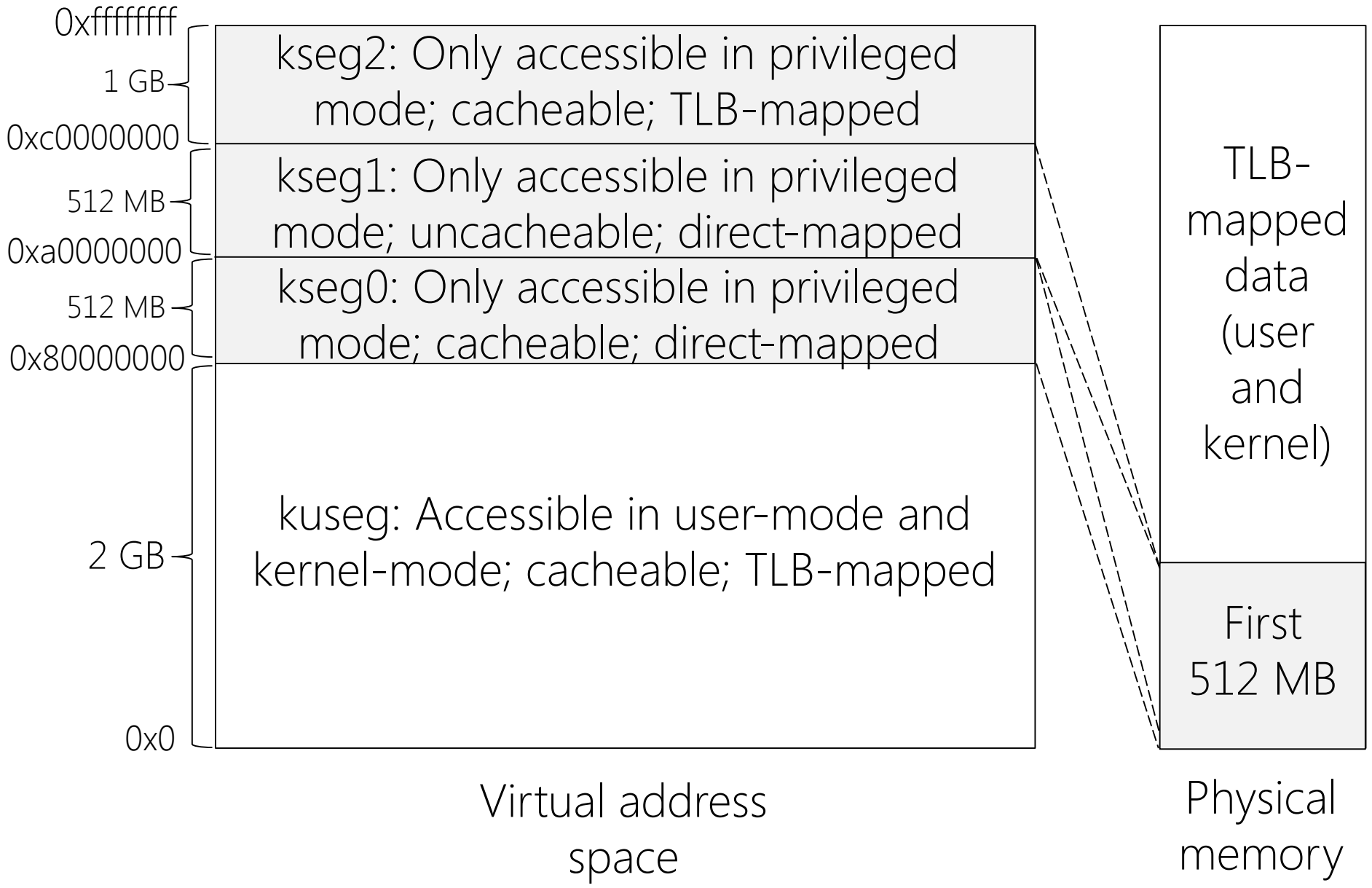
```
#include <stdio.h>
#include <stdlib.h>
int main(int argc, char *argv[]){
    printf("Location of code: %p\n", (void *) main);
    printf("Location of heap: %p\n", (void *) malloc(1));
    int x = 3;
    printf("Location of stack: %p\n", (void *) &x);
    return 0;
}
```

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

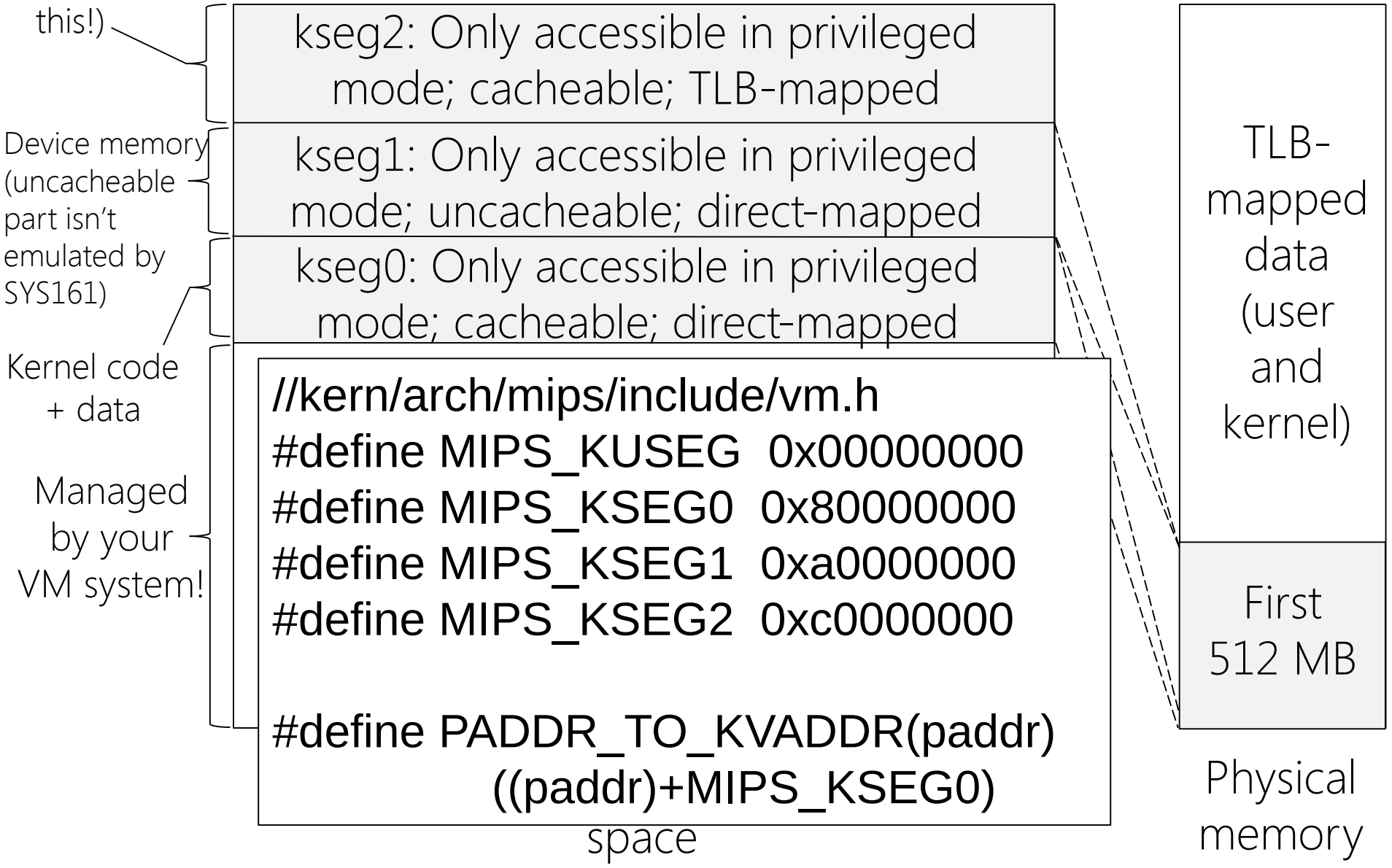
Deep-dive on Assignment 3

- Review of MIPS memory model
- Overview of your tasks in Assignment 3
- Case study: Swapping on Linux

MIPS: The Memory Model



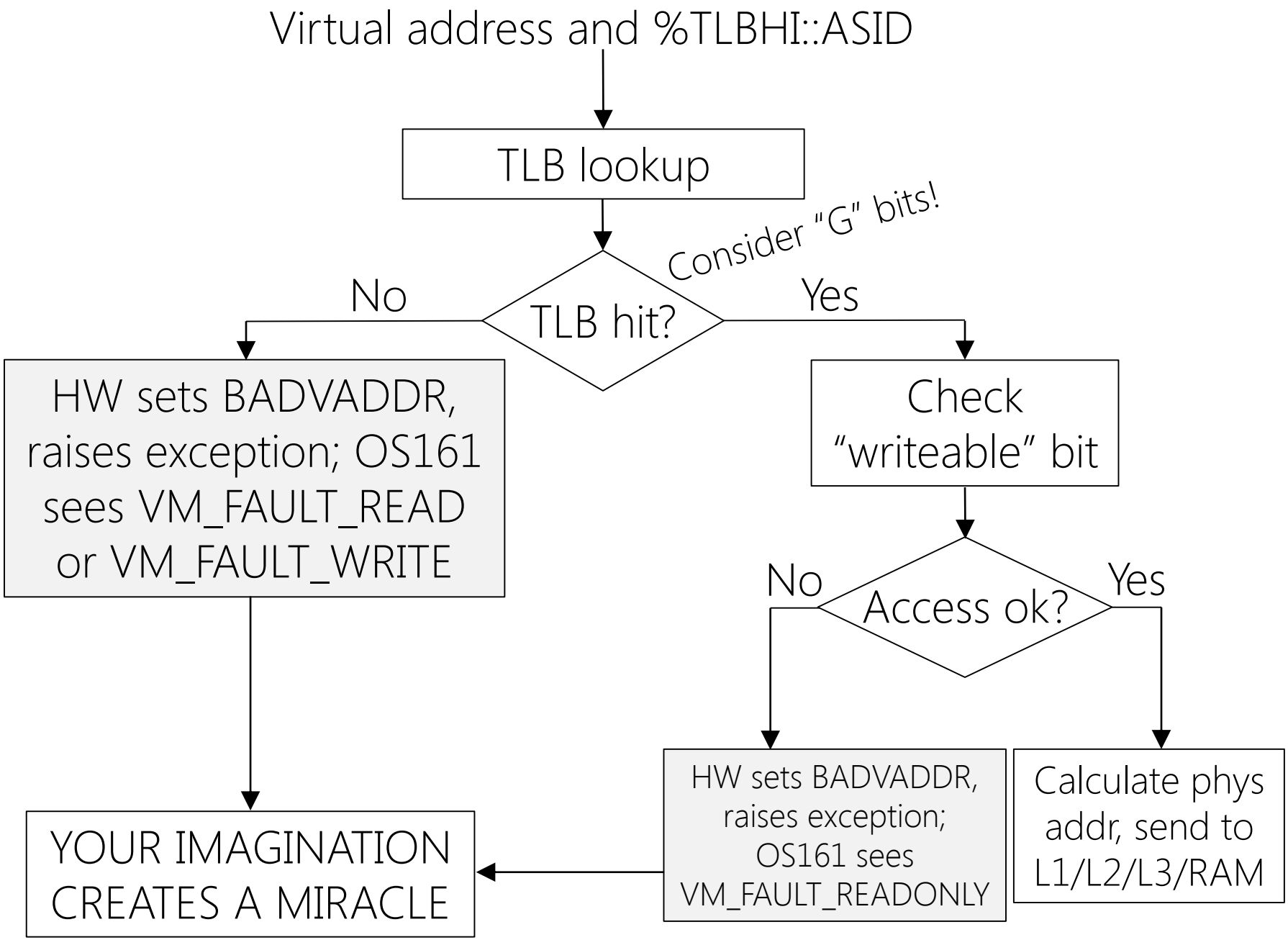
MIPS: The Memory Model



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



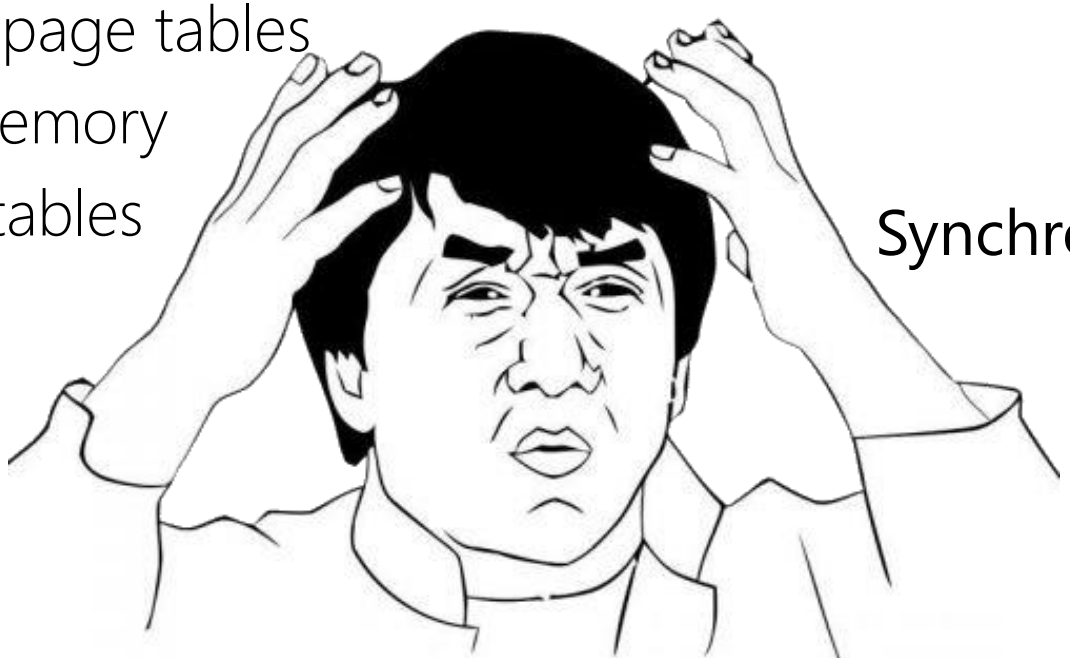
```
//We already provide four TLB handling
//functions in arch/mips/include/tlb.h.

/*Update specific TLB entry*/
void tlb_write(uint32_t entryhi,
               uint32_t entrylo,
               uint32_t index);

/*Update random TLB entry*/
void tlb_random(uint32_t entryhi,
                uint32_t entrylo);

/*Read a specific TLB entry*/
void tlb_read(uint32_t *entryhi,
              uint32_t *entrylo,
              uint32_t index);

/*Search TLB to see if it contains a
 *match for a given virtual page number*/
int tlb_probe(uint32_t entryhi,
              uint32_t entrylo /*Unused*/);
```

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!
- 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; just clear the entire TLB on a context switch (less efficient, but correct!)

Swapping a Page P Into Memory

- On page fault, check page table and 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 and a (non-exhaustive) list of what those operations might do
  - as_create(): Allocate paging structures for address space
  - as_destroy(): Deallocate paging structures, and release any physical frames used by the address space
  - as_copy(): Clone the source paging structures, then copy the necessary page data to the destination address space
  - as_activate(): Flush the TLB
- Main challenges are data structures and synchronization
  - Often best to have data structures synchronize themselves . . .

```
void foo_manipulator(){
    lock_foo();
    manipulate(foo);
    unlock(foo);
}

//Somewhere else
foo_manipulator();
```

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

//Somewhere else
lock_foo();
foo_manipulator();
unlock_foo();
```

Kernel Allocations

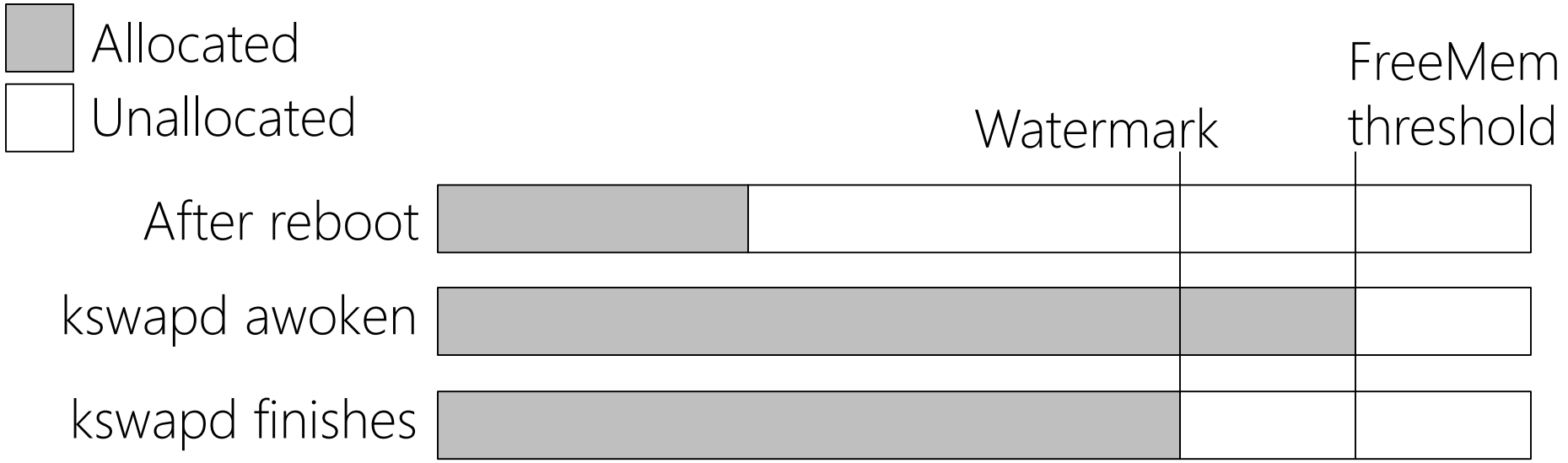
- Hint: Don't try to implement pageable kernel memory!
  - So, when you allocate a page to the kernel, it stays allocated unless the kernel gives it back
  - When the kernel asks for N pages of contiguous virtual address space, you need to find N pages of contiguous *physical* memory!

Backing Store

- You need a background kernel thread that proactively writes dirty pages to disk (making them clean)
  - Goal: Avoid synchronous writes of dirty pages that need eviction
- 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_swapon() on “lhd0raw:” and use the vnode you get back for swapping

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!



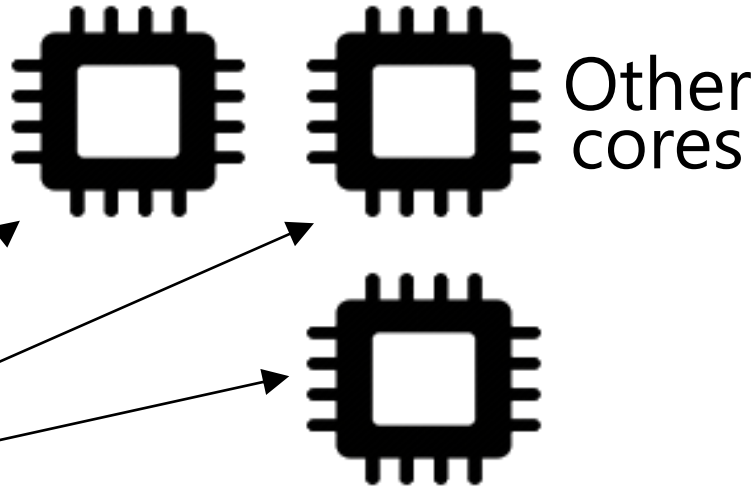
Linux Case Study: Swapping

- 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

TLB Shootdowns

- On a multicore processor, each core has its own TLB
- Sometimes, one core will need to invalidate a TLB entry that resides in another core
  - Ex: The first core is running a thread that does a copy-on-write fork(), and needs to mark a page as read-only; the second core runs a different thread in the same address space
  - Ex: The first core needs to evict a page in the address space that is running on the second core
- The first core cannot directly access the TLB of the second core
  - Thus, the first core must use inter-processor interrupts (IPIs) to perform a "TLB shootdown"

TLB Shootdowns: High-level Overview
[*Details vary according to OS implementation!*]



1) Disable interrupts, lock the page table whose mapping is being altered

2) Send IPIs to other cores

3) Each core disables interrupts, invalidates the appropriate TLB entry

4) Each core signals to initiating core that the TLB entry has been invalidated

5) Update the local TLB; unlock page table; signal to other cores that the update is done

6) Resume normal execution

TLB Shootdowns in OS161

```
//include/cpu.h

/* IPI types */
#define IPI_PANIC          0       /* System has called
                                    * panic() */
#define IPI_OFFLINE        1       /* CPU is requested to go
                                    * offline */
#define IPI_UNIDLE         2       /* Runnable threads are
                                    * available */
#define IPI_TLBSHOOTDOWN   3       /* MMU mapping(s) need
                                    * invalidation */

void ipi_send(struct cpu *target, int code);
void ipi_broadcast(int code); /* Sends an IPI to all cpus
                               * but the current one */
void ipi_tlbshootdown(struct cpu *target,
                      const struct tlbshootdown *mapping);
                             /* Like ipi_send() but carries
                              * shootdown data */
```

```
//arch/mips/include/vm.h
struct tlbshootdown{
    /*
     * Change this to what you need for your VM design.
     */
    int ts_placeholder;
};

//thread/thread.c
void ipi_tlbshootdown(struct cpu *target,
                      const struct tlbshootdown *mapping){
   unsigned n;

   spinlock_acquire(&target->c_ipi_lock);
   n = target->c_numshootdown;
   if(n == TLBSHOOTDOWN_MAX){
       /* If you have problems with this panic going off,
        * consider: (1) increasing the maximum, (2) putting
        * logic here to sleep until space appears (may
        * interact awkwardly with VM system locking), (3)
        * putting logic here to coalesce requests together,
        * and/or (4) improving VM system state tracking to
        * reduce the number of unnecessary shootdowns.
        */
       panic("ipi_tlbshootdown: Too many shootdowns
             queued\n");
   }else{
       target->c_shootdown[n] = *mapping;
       target->c_numshootdown = n+1;
   }

   target->c_ipi_pending |= (uint32_t)1 << IPI_TLBSHOOTDOWN;
   mainbus_send_ipi(target);

   spinlock_release(&target->c_ipi_lock);
}
```