Assignment 3 Section

- Learning Objective
  - Write a design document for assignment 3
- Topics:
  - Design Principles
  - Review MIPS Memory Map
  - Outline what you are asked to do in A3
  - TLB Handling
  - Paging
  - Address Spaces
  - Synchronization

Design Principles

1. If you find yourself copying lines of code, STOP!
   - Before you copy chunks of code, ask if there is a function somewhere in your future.
   - If you repeat the same sequence of lines multiple times, you are doubling the places you need to debug, and you will need to debug it.
   - Krinsky's Law: every line of code that has not been tested has a bug.
- Corollary: if two pieces of code look pretty similar, ask if they can be implemented as a parameterized function.
   - Fewer lines of code => fewer bugs.

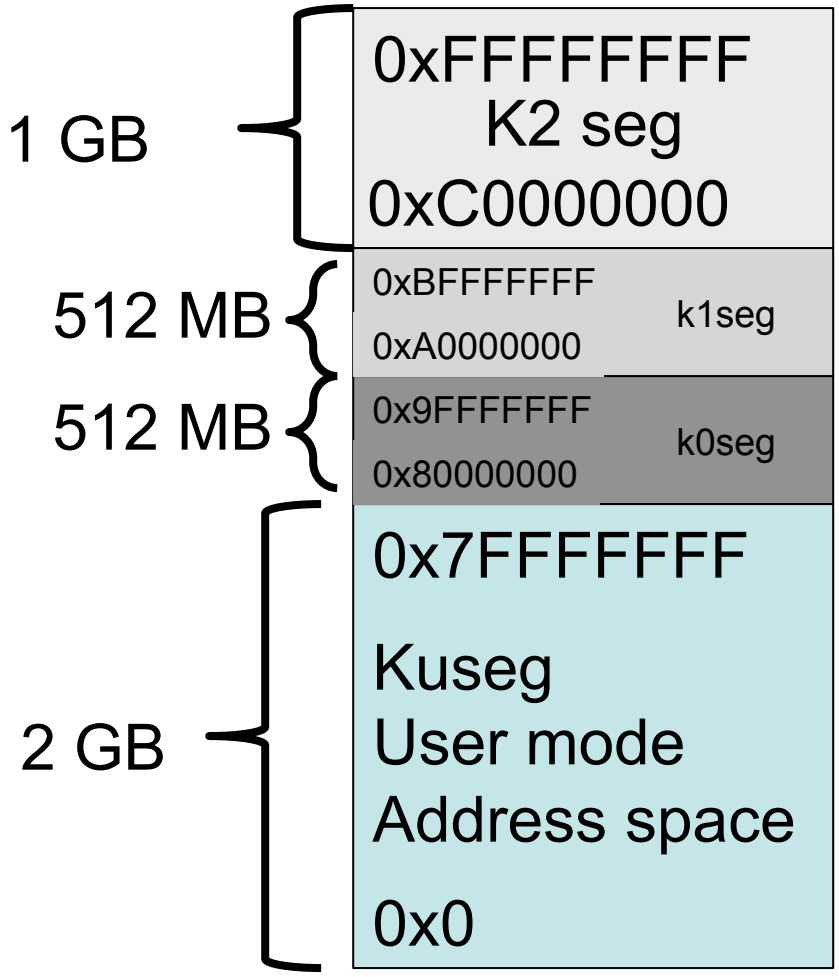
Design Principles

2. Design abstractions and live with them.
   - If you have the abstraction of a file table object or a process object, use that abstraction.
     - Have constructors/destructors
     - Don't let other code reach its grubby paws inside those objects; build interfaces.
     - Decide what code is responsible for the creation and destruction of objects.
     - Make the objects debuggable.
     - Objects may have both internal and external interfaces.
   - Use consistent error-handling methodology throughout
     - As discussed in section, error handling is a place to use `goto`s.
     - Think about error handling in your design.
     - Build things in early to check for and respond to errors.

Design Principles

3. Assertions are your friend.
   - Use them whenever you find yourself thinking, “OK at this point we know that X is true.”
   - VM bugs often manifest a long time after they actually happen; assertions are a way to catch them when they happen instead of much later when you notice that they’ve happened.
   - More asserts frequently mean you fail faster and ideally, closer to the location of the actual bug.
4. Test as you go.
   - Thoroughly testing small pieces is easier than crudely testing large pieces (and easier to debug).

MIPS R2000/3000 Review

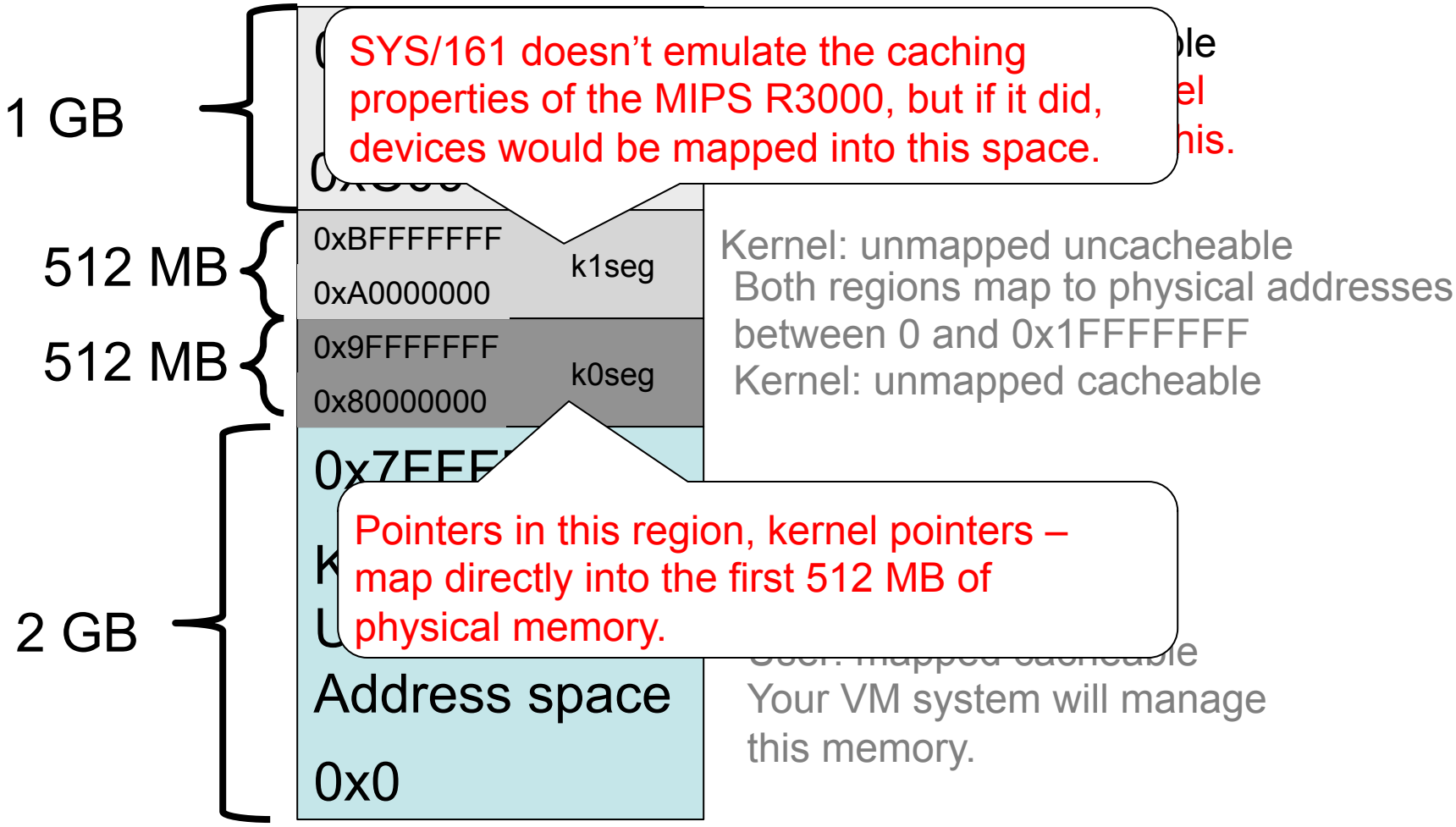


Kernel: mapped cacheable
If you were to swap kernel memory, you could use this.

Kernel: unmapped uncacheable
Both regions map to physical addresses between 0 and 0x1FFFFFFF
Kernel: unmapped cacheable

User: mapped cacheable
Your VM system will manage this memory.

MIPS R2000/3000 Review



Your Mission …

- Handle TLB faults
- Implement paging
  - Per-process data structures (page tables)
  - Global data structures (coremap)
  - Backing store support
  - Page eviction
- `sbrk()`

TLB Handling

- Read Vahalia, pages 419-422 (on resources page)
- Start with a simple replacement algorithm
- We provide routines:
  - `TLB_Random()`
  - `TLB_Write()`
  - `TLB_Read()`
  - `TLB_Probe()`
  - NOTE: TLB_Random() reserves 8 of the TLB entries; it might be easier to just use TLB_Write and random();
- Suggestion: Ignore address space IDs for now; just clear the entire TLB on every context switch.

Paging

- The tricky bits:
  - Managing all the memory mappings for each process.
  - Managing the system's memory.
  - Synchronization!
- Bootstrapping
  - The canonical chicken and egg problem:
    - You cannot `kmalloc()` until you set up your memory system.
    - You cannot set up your memory system without `kmalloc()`-ing stuff.
    - Look at `how ram_stealmem()` works.
    - Remember: YOU must manage ALL of memory.

Data Structures

- What are the key data structures?
  - Per-process virtual to physical mappings
  - Global mapping from physical address to a process and virtual address pair.
- Design these before you write your design document.
  - Get up in front of a whiteboard and draw!
  - The white board is one of your most useful tools during design.
- Flesh out the structure and API for your design document.
- Analyze the costs and benefits of your page tables.
  - How much memory do they consume?
  - Do they require linear searches? (Hint: The correct answer is no.)
  - How do you simply and efficiently do better than linear time?

Backing Store

- Figure out how to write to/read from disk.
- You will want 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 disks quite large.
  - We provide bitmap functionality that is useful for managing disk space.
  - If you put your disk in `/tmp` (a drive on the local machine), `sys161` will run faster.
  - Use `vfs_open()` on `lhd0raw`: and use the vnode you get back for swapping.

Page Faults

- Three types of page faults:
  - VM_FAULT_READONLY: a process is trying to write a page that has only read permissions.
  - VM_FAULT_READ: a process is trying to read a page that is not in memory.
  - VM_FAULT_WRITE: a process is trying to write a page that is not in memory.
- Handling a fault for page P:
  - Confirm that P exists. Check page table.
  - Decide where to place P.
    - If there is free memory, use it!
    - If there isn't, you'll need to evict someone, who?
    - Is kernel memory pageable? Can it be? YES Should it be? NO
    - How do you know if a page of memory is free?
    - Aha – the *coremap* (that mapping from PA to process/VA).
  - Evict the current resident of your target page frame.
    - Write it to backing store if necessary.
    - Update page tables.
  - Read P into memory.
    - Update the page table.
  - Update the TLB

Tricky Stuff: Kernel Allocations

- Hint: Do not implement pageable kernel memory.
- Given that: when you give the kernel a page, it stays there … forever (unless the kernel voluntarily gives it back).
- When the kernel needs N pages of contiguous virtual address space, you need to find N pages of contiguous *physical* memory.
- Think carefully about how to do this!

Address Spaces

- Operations on address spaces:
  - `as_create`
  - `as_destroy`
  - `as_copy (for fork)`
  - `as_activate (for context switching)`
- The challenges here are in data structures and synchronization; the code isn't too bad.
- But – think carefully about good data structure design and synchronization.
  - When possible, make objects synchronize themselves.
  - Which is better?

```
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;
```

Synchronization

- Points to ponder:
  - SPL synchronization won't work with I/O. Keep this mind.
  - Don't create a lock per page: this consumes too much space. You might want to use a busy bit.
  - How does locking work when handling a page fault?
  - How does locking work when evicting someone else's page?
  - What if the page I want is in the middle of being evicted by someone else?
  - What do I do in `fork` if a page that I want to copy is not resident?
- Holland's Hint: It is easier to debug a VM system with deadlocks than a VM system with race conditions.

`sbrk()`

- This is mostly bookkeeping.
- BUT – make sure it is compatible with the `malloc()` implementation we give you.
- Hint: this means you have to read the `malloc` code.
  - We used to make you write your own `malloc`, but we don't any more.
  - However, you need to understand it to make `sbrk` work.
  - Try explaining it to your partner.
  - Evaluate its design.

Statistics

- You may find yourself wanting to tune your system.
  - We frequently gather and post class performance stats.
  - This is just for fun and not for any lasting fame and fortune.
- Tuning will require that you know what's going on.
- Even if you don't want to tune, statistics will help you understand and debug your system.
- Add statistics now!
- For example:
  - Total number of pages available
  - Total number of pages managed by you
  - Number of clean pages
  - Number of dirty pages
  - Number of kernel pages
  - Your good idea goes here.

What you do NOT have to do

- Copy-on-write
- Pageable kernel memory
- Memory-mapped files
- 22-disk swap partitions

- Margo's Mantra: Get something simple working first.
  - Make sure it is robust.
  - Only then should you consider adding anything fancy.