Assignment 3

- If you have not completed A2a or A2b…
  - We will only be retesting widefork and hogparty from A2.
  - No other A3 tests use argument passing or process management functions (i.e. fork, waitpid, _exit, execv).
    - Only hogparty relies on argument passing.
    - In total widefork and hogparty are only worth 10% of the A3 grade.
  - If you did not get A2a or A2b working consider reverting back to your A1 code and build A3 on top of that (rather than spend a lot of time debugging A2a and A2b for only a limited amount of marks).

Assignment 3

- Dumbvm is a very limited virtual memory system with four major limitations.

  1. A full TLB leads to a kernel panic.
  2. Text (i.e. code) segment is not read-only.
  3. It never reuses physical memory (i.e. kfree does nothing).
     - Requires restarting the OS after each test
  4. It uses segmentation addresses.
     - which causes external fragmentation
     - *No need to fix this in W20.*

- Assignment 3 fixes these problems!
- Many former CS350 students say A3 is easier than A2.
- Caution: A3 reduces the amount of physical memory allowed for the tests so you should be using memory frugally, i.e. make sure you are using kfree when appropriate, do not have a large PID tables etc.

1. TLB Replacement

- VM related exceptions are handled by *vm_fault*()
- *vm_fault*() performs address translation and loads the virtual address to physical address mapping into the TLB.
  - Iterates through the TLB to find an unused/invalid entry.
  - Overwrites the unused entry with the virtual to physical address mapping required by the instruction that generated the TLB exception.
- Modify *vm_fault*() so that when the TLB is full, it calls *tlb_random*() to write the entry into a random TLB slot.
  - That's it for TLB replacement!
  - Make sure that virtual page fields in the TLB are unique.

2. Read-Only Text Segment

Modification 2a

- Currently, TLB entries are loaded with *TLBLO_DIRTY* on for all entries.
  - Therefore, all pages are readable and writeable.
- The text (i.e. code) segment should be read-only.
  - Load TLB entries for the text segment with *TLBLO_DIRTY* off, i.e. **`elo &= ~TLBLO_DIRTY;`**
- Determine the segment of the fault address by looking at the *vbase* and *vtop* addresses.

2. Read-Only Text Segment

Modification 2b

- Unfortunately, this change will cause *load_elf*() to throw a VM_FAULT_READONLY exception when it loads the object file.
  - The loader is trying to write to a memory location that is read-only.
- We must instead load TLB entries with TLBLO_DIRTY on until *load_elf*() has completed.
  - Consider adding a flag to struct *addrspace* to indicate whether or not *load_elf*() has completed.
  - When *load_elf*() completes, flush the TLB (with *as_activate*()) and ensure that all future TLB entries for the text segment has TLBLO_DIRTY off.

2. Read-Only Text Segment

Modification 2c

- Writing to read-only memory address will lead to a VM_FAULT_READONLY exception.
  - Currently this exception will cause a kernel panic.
- Instead of panicking, your VM system should kill the process.
  - I.e. detect when a user program tries to write to read-only memory.
  - Have *vm_fault*() return the appropriate error code / signal.
  - That will be picked up when *mips_trap* (which handles exceptions and interrupts) which calls *kill_curthread*().
  - Modify *kill_curthread* (which handles the situation where user-level code has a fatal fault) to kill the current process.

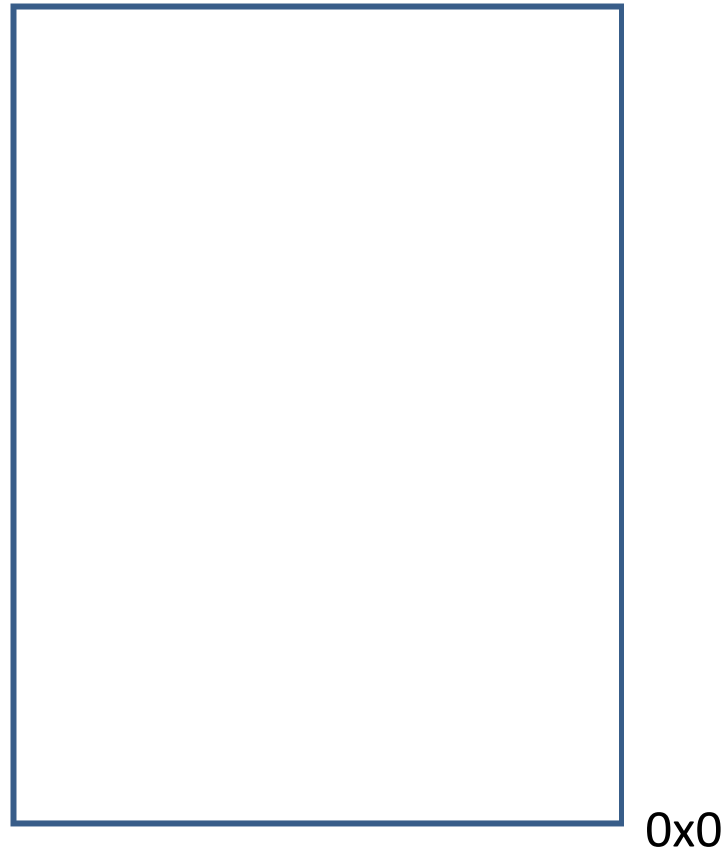
2. Read-Only Text Segment

Modification 2c

- There are three different approaches to modifying *kill_curthread*.
  1. Add the code to kill the thread to *kill_curthread*. But this approach is not reusing code.
  2. Create your own function very similar to *sys___exit* (say *sys_kill*) except that the exit code/status will be different.
  3. Modify your implementation of *sys___exit* to take a parameter that is the reason why *sys___exit* was called.
- Consider which signal number this will trigger. Hint: look at the beginning of *kill_curthread*.

3. Managing Memory

Initially physical memory is unused.



3. Managing Memory

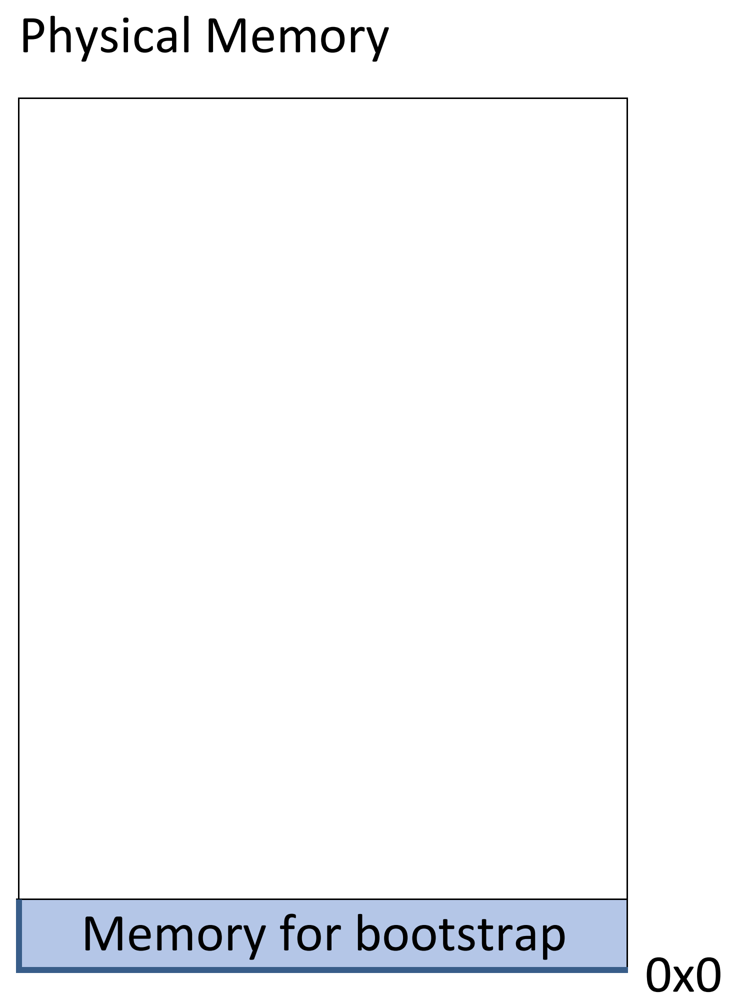
During bootstrap, the kernel allocates memory by calling *getppages*, which in turn calls *ram_stealmem*(num_*pages*).

ram_stealmem just allocates pages without providing any mechanism to free these pages (see *free_kpages*).

Do not modify this part of the code.

Instead, we want to manage physical memory *after* the bootstrap process.

I.e. manage the rest of physical memory using paging with a data structure called a core-map.



3. Managing Memory

Core-map

- Keep track of whether the frame is in use (1) or not (0).
- To allocate RAM search through core-map to find a large enough space.
- For allocations of multiple continuous pages, keep track of how many pages have been allocated in the core-map and free it as one big unit.

Version 1

Frame #	In Use?	Page	of
0	1	1	2
1	1	2	2
2	0	0	0
3	1	1	1
4	1	1	3
5	1	2	3
6	1	3	3
7	0	0	0

Version 2

Page
1
2
0
1
1
2
3
0

- e.g. Frame 0 and 1 are part of one big allocation and so a call to free frame 0 will free both frames 0 and 1.
- Version 2 of the core-map just keeps the essential information.

3. Managing Memory

Core-map Version 2

- Allocation would be the same.
- To free pages you need to check its successor to see if it is part of a larger allocation, i.e. is its count one higher than your count.
- Must also keep track of where the 0^{th} frame is located in physical memory so that when memory is requested the kernel can return an address to the start of the allocation.

Version 2

Page
1
2
0
1
1
2
3
0

For Both Version 1 or 2.

- With either implementation, since you are implementing the core of memory allocation, so you do not call *kmalloc* to allocate space for the core-map, you simply calculate its size and leave the rest of RAM as frames to be allocated.

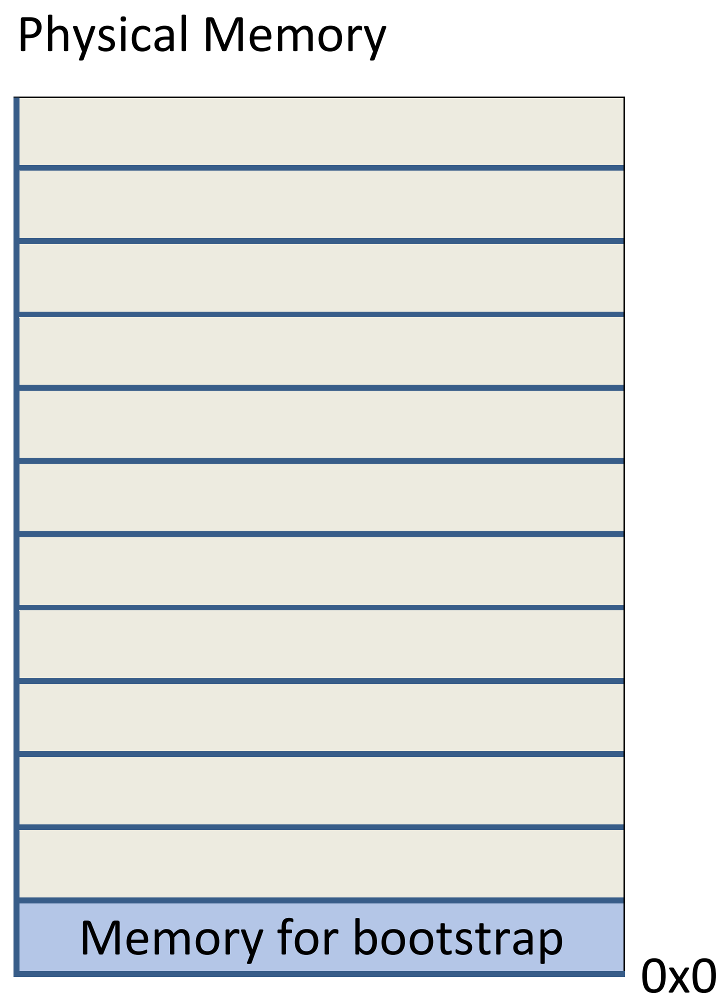
3. Managing Memory

In *vm_bootstrap*, call *ram_getsize* to get the remaining physical memory in the system.

It will give a low (just after memory for bootstrap) and a high address.

Once *ram_getsize* has been called, do not call *ram_stealmem* again!

Logically partition the remaining physical memory into fixed size frames. Each frame is PAGE_SIZE bytes and its address must be an integer multiple of the page size (i.e. it is page aligned).



3. Managing Memory

Where should we store the core-map data structure?

Store it in the start of the memory returned by *ram_getsize* (i.e. the area just after the memory used for bootstrap).

The frames that the core-map manages should start after the core-map data structure (rounded up to be a multiple of the page size).

I.e. the core-map should not track its own memory usage. Tracking its own usage can lead to bugs that are hard to find.

3. Managing Memory

- You never have to kfree the core-map. You use it until the system shuts down in which case kfreeing it is no longer necessary.
- There are parts of the OS that will be calling *kmalloc* before you create the coremap so …
  - You will need to create a flag to indicate when the kernel can stop using *ram_stealmem* and starting using the core-map to allocated physical memory.
  - Look at *vm_bootstrap* to help decide exactly when you create the core-map.
  - You must also modify the two functions *alloc_kpages*(*int npages*) and *free_kpages*(*vaddr_t addr*) to use the core-map once it has been created.

Alloc and Free

- *alloc_kpages*(*int npages*):
  - Allocates frames for both *kmalloc* and for address spaces.
  - Frames need to be contiguous.
  - Do not have *alloc_kpages* interact directly with core map.
  - Instead look at a function it uses, *getppages*, and modify it so it uses *ram_stealmem* before the core-map is created and uses the core-map after it is created.
  - The reason for this is because some parts of the kernel call *getppages* directly rather than calling *alloc_kpages*.
- *free_kpages*(*vaddr_t addr*):
  - It currently does not do anything but it should be freeing pages allocated with *alloc_kpages.*
  - We don't specify how many pages we need to free so it should free the same number of pages that was allocated.
  - It should update the core-map to make those frames available after *free_kpages* is called.

User Address / Kernel Virtual Address / Physical Address

- Remember that you are always working with virtual addresses.
  - Only use physical addresses when loading entries in the TLB.
  - Virtual addresses are converted either by the TLB or by the MMU directly.
- Addresses below 0x8000 0000 are user-space addresses that are TLB mapped.
- Addresses between 0x8000 0000 and 0xa000 0000 are kernel virtual addresses that are converted by the MMU directly, i.e.
  Kernel virtual address – 0x8000 0000 = physical address
- *kmalloc* always returns a kernel virtual address.
- Do not use *kmalloc* to allocate frames.