Assignment 3

- Dumbvm is a very limited virtual memory system
  - A full TLB leads to a kernel panic
  - Text segment is not read-only
  - Uses fixed segmentation
    - External fragmentation
  - Never reuses physical memory
    - Requires restarting the OS after each test

- Assignment 3 fixes these problems!

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
- If the TLB is full, call *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

Read-Only Text Segment

- Currently, TLB entries are loaded with **TLBLO_DIRTY** on
  - Pages are therefore read and writeable
- Text segment should be read-only
  - Load TLB entries for the text segment with TLBLO_DIRTY off
  - elo &= ~TLBLO_DIRTY;
- Determine the segment of the fault address by looking at the vbase and vtop addresses

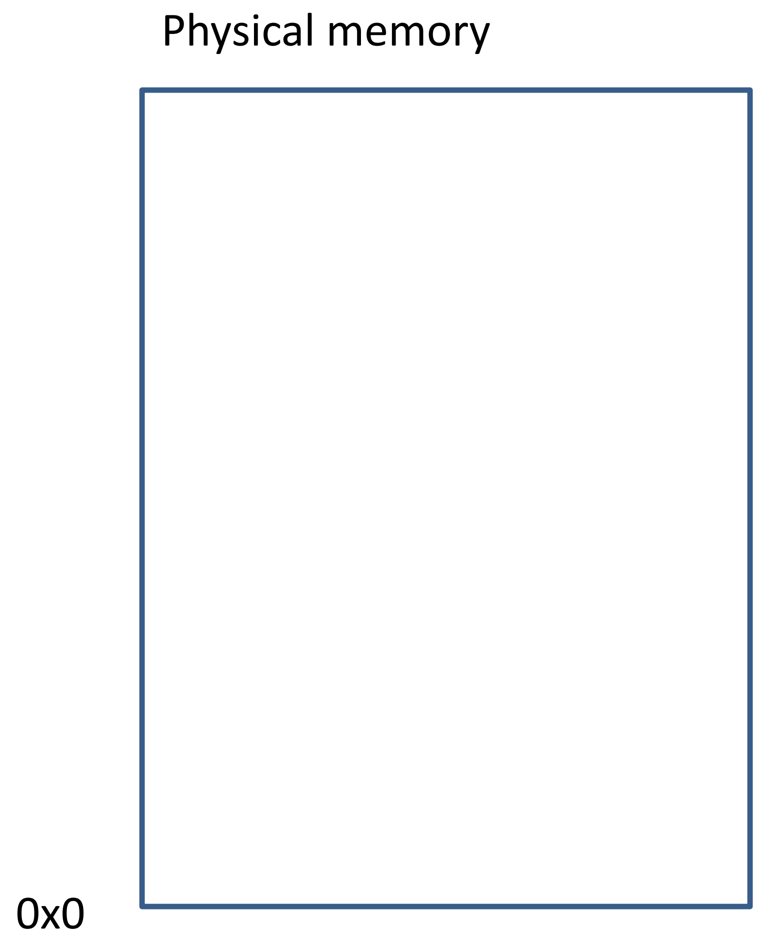
Read-Only Text Segment

- Unfortunately, this will cause load_elf to throw a VM_FAULT_READONLY exception
  - It 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, and ensure that all future TLB entries for the text segment has TLBLO_DIRTY off

Read-Only Text Segment

- Writing to read-only memory address will lead to a VM_FAULT_READONLY exception
  - This will currently cause a kernel panic
- Instead of panicing, your VM system should kill the process
  - Modify kill_curthread to kill the current process
    - Very similar to sys__exit. However, the exit code/status will be different
    - Consider which signal number this will trigger (look at the beginning of kill_curthread)

Managing Memory

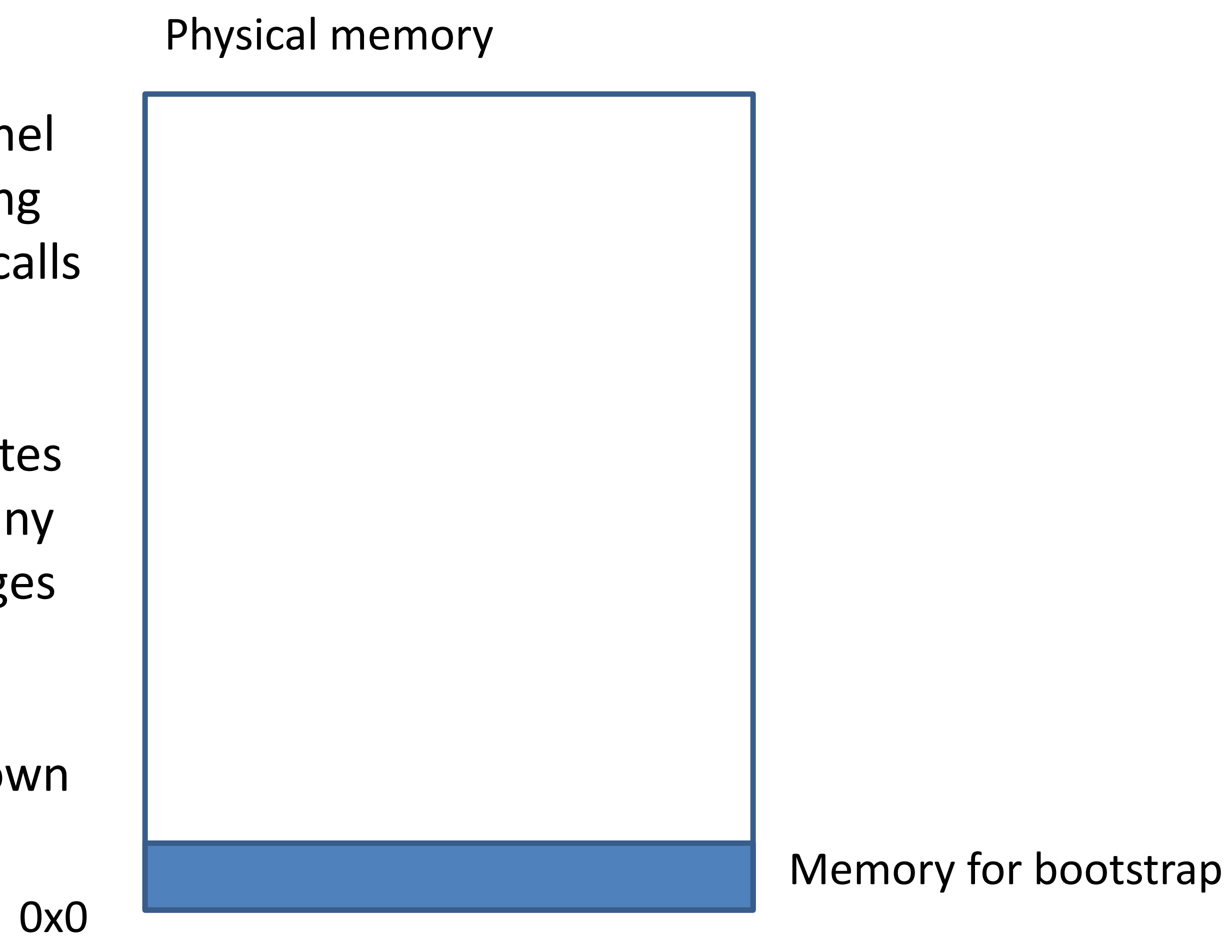


Managing Memory

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

ram_stealmem just allocates pages without providing any mechanism to release pages (see free_kpages)

We want to manage our own memory after bootstrap

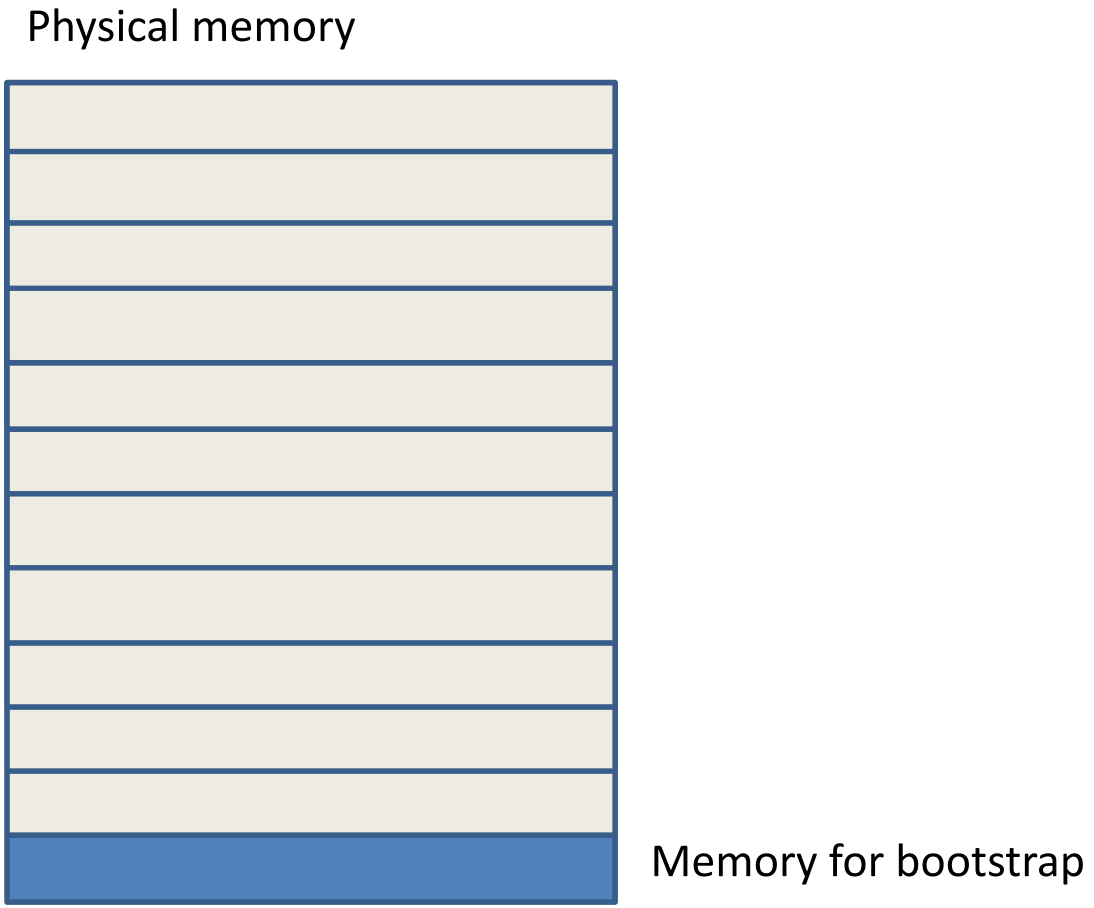


Managing Memory

In vm_bootstrap, call *ram_getsize* to get the remaining physical memory in the system. Do not call ram_stealmem again!

Logically partition the memory into fixed size frames. Each frame is PAGE_SIZE bytes.

Keep track of the status of each frame (core-map data structure).

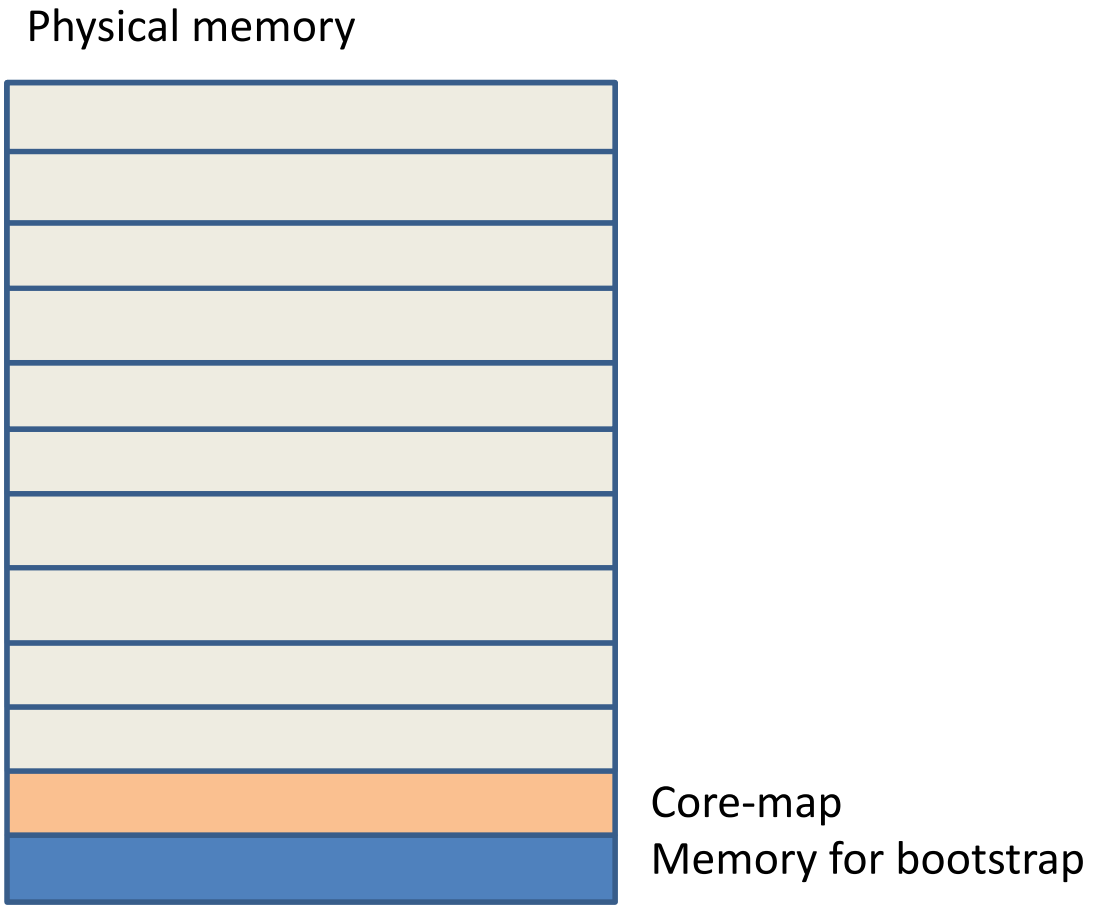


Managing Memory

Where should we store the core-map data structure?

Store it in the start of the memory returned by ram_getsize. The frames should start after the core-map data structure.

The core-map should track which frames are used and available. It should also keep track of contiguous memory allocations (why?)



Alloc and Free

- alloc_kpages(int npages):
  - Allocate frames for both kmalloc and address spaces
  - Frames need to be contiguous
- free_kpages(vaddr_t addr):
  - Free pages allocated with alloc_kpages
  - We don’t specify how many pages we need to free. It should free the same number of pages that was allocated.
  - Update core-map to make frames available after free_kpages
- Consider adding some information in the core-map to help determine the number of pages that need to be freed
  - E.g. If 4 contiguous frames were allocated using alloc_kpages, then store 4 in the core-map entry for the start of the four frames

Page Tables

- In order to avoid external fragmentation, we need to introduce paging
- New VM system combines segmentation with paging
- Three segments:
  - Text (read-only)
  - Data
  - Stack
- Create a page table for each segment
  - Each page table entry should include the frame number

Page Tables

- In dumbvm, struct addrspace has the following fields:
  - vaddr_t as_vbase1
  - paddr_t as_pbase1
  - size_t as_npages1
  - vaddr_t as_vbase2
  - paddr_t as_pbase2
  - size_t as_npages2
  - paddr_t as_stackpbase
- With segmentation and paging, what fields should we have instead?
  - Replace pbase with page table

Page Tables

- as_create:
  - Initialize address space data structures

- as_define_region:
  - A region is essentially a segment
  - Allocate (kmalloc) and initialize the page table for the specified segment
  - Size of the segment is a parameter of as_define_region
    - Because we perform preloading, segment size will never grow!
    - Size of the page table is based on the segment size
  - Setup the read/write permissions for this segment
  - Optionally have permissions per page

Page Tables

- as_prepare_load:
  - Pre-allocate frames for each page in the segments
  - Frames do not need to be contiguous
  - Allocate each frame one at a time
  - Need to create a page table for the stack
    - as_define_region not called for the stack segment

Page Tables

- as_copy:
  - Call *as_create* to create the address space
  - Create segments based on old address space
  - Allocate frames for the segments
  - memcpy frames from the old address space to the frames in the new address space

- as_destroy:
  - Call free_kpages on the frames for each segment
  - kfree the page tables

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 0x80000000 are user space addresses that are TLB mapped
- Addresses between 0x80000000 and 0xa0000000 are kernel virtual addresses that are converted by the MMU directly
  - Kernel virtual address – 0x80000000 = physical address
- kmalloc always returns a kernel virtual address. Do not use kmalloc to allocate frames