Eli Sobylak

ANSWERS IN RED were completed by me

Collaborators were: Noah, Gavin, Kalen, Nathaniel, Austin, Matthew, and Michelle

To see who did what, view this document in color

Memory Management in OS161:

From answering these questions and working through the problems on the final, I learned that the OS161 system uses a lot of the techniques we talked about in lecture to manage its memory. For instance, OS161 employs a paged system to keep track of virtual address space, and frame #s, etc. Additionally, we got to see how OS161 handles its memory on a vary fine grain level. From reading about the variables that set ram sizes, and functions that search the address space. To examining lecture concepts such as TLBs and stacks.

- Eli Sobylak Answered Questions in Red
- Noah's stuff is Blue
- Gavin is Magenta
- Kalen is Green
- Nathaniel is Cheesy
- Austin is Cyan
- Matthew is Brown
- Michelle is yellow

OS161 VM Travel Guide

- **Part A.** Read Sys161 Memory Map and other low-level structures (UNSW). Look at the MIPS memory map. Read the Introduction, the Tutorial Exercises, Coding Assignment, Basic Assignment, and what you're NOT doing in the Advanced Assignment.

  Please answer the following questions and bring them to your tutorial in week 11. You should be familiar enough with navigating the kernel source that you can find the answers to the below questions by yourself (Hint: use the *grep* utility). You may also find the MIPS r3000 reference useful.

1. What is the difference between the different MIPS address space segments? What is the use of each segment?
   - kseg2, TLB-mapped cacheable kernel space
   - kseg1, direct-mapped uncached kernel space
     i. Exception address if BEV set.
     UTLB exception address if BEV set.
     Execution begins here after processor reset.
   - kseg0, direct-mapped cached kernel space
     i. Exception address if BEV not set.
     UTLB exception address if BEV not set.
   - kuseg, TLB-mapped cacheable user space

2. What functions exist to help you manage the TLB? Describe their use. (Hint: look in kern/arch/mips/include/tlb.h)
   - void tlb_random(uint32_t entryhi, uint32_t entrylo);
     i. write tlb entry "random": store c0_entryhi and c0_entrylo into the TLB entry named by the index field of c0_random.
   - void tlb_write(uint32_t entryhi, uint32_t entrylo, uint32_t index);

        i. write tlb entry indexed: store c0_entryhi and c0_entrylo into the TLB entry named by the index field of c0_index.
    - void tlb_read(uint32_t *entryhi, uint32_t *entrylo, uint32_t index);
        i. read tlb entry: load the TLB entry named by the index field of c0_index into c0_entryhi and c0_entrylo.
    - int tlb_probe(uint32_t entryhi, uint32_t entrylo);
        i. probe tlb: search TLB for entry matching c0_entryhi; set probe-failed bit and index field in c0_index.

3. What macros are used to convert from a physical address to a kernel virtual address?
    - PADDR_TO_KVADDR: returns the physical address of a virtual address in kernel space
4. What address should the initial user stack pointer be?
   The initial user stack is defined as such

```
#define USERSTACK     USERSPACETOP
#define USERSPACETOP  MIPS_KSEG0
#define MIPS_KSEG0  0x80000000
```

5. What are the entryhi and entrylo co-processor registers? Describe their contents.
    - c0_entryhi

Bits	Name	Description
12-31	VPN	Virtual page number (bits 12-31 of address) for VM mapping.
6-11	ASID	ID of address space in which virtual address exists.
0-5	unused	

    - c0_entrylo

Bits	Name	Description
12-31	PFN	Physical page number (bits 12-31 of address) for VM mapping.
11	N	Non-cacheable; if set, RAM cache is disabled accessing this page.
10	D	Dirty; if set, page may be written to.
9	V	Valid; if set, page may be accessed.
8	G	Global; if set, valid in every address space.
0-7	unused	

    - kern/arch/mips/include/specialreg.h:
    - #define c0_entrylo $2 /* TLB entry contents (low-order half) */
    - #define c0_entryhi $10 /* TLB entry contents (high-order half) */
6. What do the as_* functions do? Why do we need as_prepare_load() and as_complete_load()?
    - They manipulate address space
    - as_prepare_load - this is called before actually loading from an executable into the address space. And it prepares the load.
    - as_complete_load - this is called when loading from an executable is complete. This double checks the load for errors.
7. What does vm_fault() do? When is it called?

- Fault handling function called by trap code

8. Assuming a 2-level hierarchical page table (4k-sized page, page size is 4096), show for the following virtual addresses:
   - The page number and offset;
     4kb = 2^{12} → Offset = 12 bits
     Virtual address = 24 bits
   - the translated address (after any page allocation); and
   - the contents of the page table after the TLB miss.

   The page table is initially empty, with no L2 pages. You may assume that the allocator returns frames in order, so that the first frame allocated is frame 0, then frames 1, 2, 3, etc.

Virtual Address	Page Number	Offset	Translated Address
0x100 008	0x100	0x008	0x000008
0x101 008	0x101	0x008	0x001008
0x100 0F0	0x100	0x0F0	0x0000F0
0x041 000	0x041	0x000	0x003000
0x041 B00	0x041	0xB00	0x003B00?

- **Part B.** Read kern/arch/mips/vm/ram.c.
  1. What is the value of MIPS_KSEG0?
     vm.h:52:#define MIPS_KSEG0 0x80000000
     - The starting value for the stack pointer at user level. Because the stack is subtract-then-store, this can start as the next address after the stack area.
  2. What is firstpaddr?
     - address of first free physical page
  3. What is lastpaddr?
     - one past end of last free physical page
  4. What is firstfree?
     - first free virtual address; set by start.S
  5. What is the maximum physical address size possible in OS161?

     "Both direct-mapped segments map to the first 512 megabytes of the physical address space."
     >Implies that the physical address space is larger than 512 megabytes

```
if (ramsize > 508*1024*1024) {
        ramsize = 508*1024*1024;
}
```

Just realized that the 1024*1024 are unit conversion, so 508mb is the max

6. What is the serial order in which the following are called or used: start.S, firstfree, firstpaddr, lastpaddr, ramsize, ram_bootstrap, ram_stealmem, ram_getsize?
   - startS
   - firstfree (vaddr_t firstfree; IN ram.c) ← correct, but actually in start.S (line 112)
   - ram_bootstrap (startup/main.c: ram_bootstrap(); IN main.c)
   - firstpaddr (static paddr_t firstpaddr; IN ram.c)
   - lastpaddr (static paddr_t lastpaddr; IN ram.c)
   - ramsize (ramsize = mainbus_ramsize(); IN ram.c)
   - ram_stealmem (paddr_t ram_stealmem(unsigned long npages); IN vm.h)
   - ram_getsize (void ram_getsize(paddr_t *lo, paddr_t *hi)); IN vm.h) ← this is declared, but as far as I can tell never actually called

7. What is the serial order of the following abstract actions: Polling the hardware for first available virtual address, VM initialization, Physical Memory setup.

   Polling the hardware is first
   Physical Memory Setup is second
   /*Noah had Physical Memory Setup first and Polling the hardware second, so it might be that */
   VM initialization is third

8. In what order are the following called during Os161 boot and what is the associated abstract activity: ram_bootstrap, alloc_kpages, ram_stealmem, ram_getsize, vm_bootstrap?
   ram_bootstrap //main.c
   vm_bootstrap //main.c
   ram_stealmem //mips/include/vm.c
   ram_getsize //mips/include/vm.c
   alloc_kpages //include/vm.c

- **Part C.** Read kern/arch/mips/vm/dumbvm.c.
  1. Explain how memory is managed in the dumbvm.c.
     /* Assert that the address space has been set up properly. */
     /* make sure it's page-aligned */
     /* Disable interrupts on this CPU while frobbing the TLB. */
     /* Kernel threads don't have an address spaces to activate */

/* Disable interrupts on this CPU while frobbing the TLB. */
/* Align the region. First, the base... */

2. How is memory initialized?
    - #define DUMBVM_STACKPAGES   12
        - under dumbvm, always have 48k of user stack

   The dumbvm system has as_create() to initialize address space, and as_destroy() to free it.
   as_zero_region() zeros out all the bits in an address space, and is called by as_prepare_load().

3. How are kernel virtual addresses translated to physical addresses?
   I think this should be:
   firstpaddr = firstfree - MIPS_KSEG0;
   kprintf("%uk physical memory available\n",
   (lastpaddr-firstpaddr)/1024);

   Look in ram.c at the comment right above this “translation”. It says
   /*
   * Get first free virtual address from where start.S saved it.
   * Convert to physical address.
   */
   We know it’s the kernel because of the definition of MIPS_KSEG0
   ^ Agreed

```
Vaddr_t
alloc_kpages(int npages)
{
        paddr_t pa;
        pa = getppages(npages);
        if (pa==0) {
        return 0;
        }
return PADDR_TO_KVADDR(pa);
}
```

4. How are user virtual addresses translated to physical addresses?
   I would say same as above, but in this case use MIPS_KUSEG so…
   firstpaddr = firstfree - MIPS_KUSEG
   (lastpaddr-firstpaddr)/1024;

   In as_define_region():

```
size_t npages;

sz += vaddr & ~(vaddr_t)PAGE_FRAME;
vaddr &= PAGE_FRAME;

sz = (sz + PAGE_SIZE - 1) & PAGE_FRAME;

npages = sz / PAGE_SIZE;
```

5. How is kmalloc handled?
    - kmalloc is called in the as_create function in dumbvm.c and is used to allocate memory based off of the size of the address space struacture in the code. This makes sense, because when the address space gets created, we have to allocate a block of memory to assign to it.

6. How is kfree handled?
    - kfree is a mirror function to kmalloc and gets called in as_destroy. This is reasonable because when the address space is finished being used, you need to free up the memory that was being used.

7. What is subpage allocation and what is it for?
    - subpage allocation is allocating one page at a time and fills it with objects of a given size for any objects. Each page in memory has it's own pool called a freelist, and each page also has a freecount so we can determine when the page is empty and can be released. It is a way to allocate pages to various objects in the os161 system.
8. How are user address spaces created?
    - User address space gets created by calling the as_created function
9. How are user address spaces destroyed?
    - User address space then gets destroyed when as_destroy gets called
10. What is done to activate user address spaces?
    - It appears that as_activate increments over the number of TLB slots available and for each one does a tlb_write. It also has a way of disabling interrupts on this CPU while it is probing the TLB
    - This appears to invalidate every entry in the TLB, essentially clearing the TLB every time a different address space is activated. (I think).
11. What is an address space "region"? How many VM regions are there and what are they?
    - There are 2 possible regions. The first one, the second one, and then there is a function for the zero address space region.
12. Who calls as_define_region?
    - The only function that calls as_define_region is load_elf() in load_elf.c

        - I assume it creates a new region to load the executable code into
    13. Are regions protected from each other? If so, then what code protects regions?
        - The regions are not protected from each other anywhere in dumbvm..
    14. Are regions accessed differently? If so, then how?
        - The regions are not accessed differently other than the fact that physical and virtual memory is handled differently.
    15. What code accesses each region?
        - as_prepare_load makes sure that everything is in place for the load to go properly
    16. What does as_prepare_load do and what is it for?
        - as_prepare_load allocates memory up front as a process is created.
    17. Who calls as_prepare_load?
        - Both load_elf() from load_elf.c and as_copy from dumbvm.c call as_prepare_load. I assume it prepares an address space for allocation
    18. What is as_complete_load for?

        "Code to load an ELF-format executable into the current address space.
        It makes the following address space calls:
        - first, as_define_region once for each segment of the program;
        - then, as_prepare_load;
        - then it loads each chunk of the program;
        - finally, as_complete_load."

        as_complete load is called when loading from an executable is complete.
        The function takes an address space and voids it

    19. How does as_define_stack work?
        as_define_stack simply assigns the stack pointer to USERSTACK
        It checks to see if the stack pointer base is 0, and then sets the stack pointer to the userstack.

    20. Who calls as_define_stack?

        runprogram() in runprogram.c calls as_define_stack().

- **Part D.** Read vm_fault in dumbvm.c.
    1. Who calls vm_fault?
        - arch/mips/locore/trap.c
    2. When is vm_fault called?
        - Call vm_fault on the TLB exceptions.
    3. What does vm_fault do in the dumbvm?
        - Checks for error cause and writes a new TLB entry