Operating Systems Project 3 Travelogue

Noah Weiner

Part A

1. The different segments of MIPS memory are allocated to either kernel or user processes. Of the kernel segments, 2 are direct-mapped (cached or uncached) and one is TLB-mapped (cacheable).
2. (a) **tlb_random**: write a page to a random TLB slot.
   (b) **tlb_write**: write to a specific TLB slot.
   (c) **tlb_read**: read a page out of a slot.
   (d) **tbl_probe**: find a TLB entry matching a virtual page number.
3. **PADDR_TO_KVADDR**: returns the physical address of a virtual address in kernel space (kseg0/kseg1).
4. The user stack pointer is initially set to USERSTACKTOP, which is MIPS_KSEG0, which is 0x80000000
5. (a) **c0_entryhi**: the virtual page number of the current TLB entry, as well as its address space ID.
   (b) **c0_entrylo**: the physical page number of the current TLB entry, as well as address space ID, as well as various flag bits (cacheable, dirty, global, valid).
6. The as_* functions are used to manipulate address spaces. The as_prepare_load() and as_complete_load() functions are called before and after loading an executable into memory, in order to set the correct page table permissions.
7. vm_fault() handles page faults.
8. (a) **0x100008**
   i. Page no: 256 (0x100), offset: 8 (0x008)
   ii. Translated address: 0x000008
   iii. TLB contents:

Page #	Frame #
0x100	0x008

   (b) **0x101008**
   i. Page no: 257 (0x101), offset: 8 (0x008)
   ii. Translated address: 0x001008
   iii. TLB contents:

Page #	Frame #
0x100	0x000
0x101	0x001

   (c) **0x1000f0**
   i. Page no: 256 (0x100), offset: 240 (0x0f0)
   ii. Translated address: 0x0000f0
   iii. TLB contents:

Page #	Frame #
0x100	0x000
0x101	0x001

   (d) **0x41000**
   i. Page no: 65 (0x41), offset: 0 (0x0)
   ii. Translated address: 0x002000
   iii. TLB contents:

Page #	Frame #
0x100	0x000
0x101	0x001
0x041	0x002

   (e) **0x41b00**
   i. Page no: 65 (0x41), offset: 2816 (0xb00)
   ii. Translated address: 0x002b00
   iii. TLB contents:

Page #	Frame #
0x100	0x000
0x101	0x001
0x041	0x002

Part B

1. MIPS_KSEG0 is 0x80000000
2. firstpaddr is the physical address of the first free page

3. lastpaddr is one past the end of the last free page
4. firstfree is the first free virtual address.
5. If there is 508 MB of RAM or more, the largest physical address is 0x1FC00000.
6. start.S → firstfree → ram_bootstrap → ramsize → lastpaddr → firstpaddr → ram_stealmem

   ram_getsize is never actually called.
7. Physical memory setup → polling hardware → VM initialization
8. ram_bootstrap → alloc_kpages → ram_stealmem → vm_bootstrap

   Again, I don't think ram_getsize is ever called.

Part C

1. Dumbvm has 12 pages of user stack, which it fills sequentially and then flops. It abuses functions (as_prepare_load, meant for preparing to read executables, and ram_stealmem, meant for vm bootstrapping) to do so.
2. as_create initializes the structure, as_prepare_load actually allocates memory via getppages, and memory is zeroed via as_zero_region.
3. It's not, yet, given that virtual memory isn't actually implemented. PADDR_TO_KVADDR goes the other way though.
4. Same as the above.
5. kmalloc grabs memory via alloc_kpages if necessary, otherwise subpage_malloc.
6. kfree tries subpage_kfree, and then free_kpages if necessary.
7. Each page maintains a list of subpages, which can be any size that's a multiple of 4 (preferably 8, according to kmalloc.c). Allocation is essentially finding a free subpage that fits whatever size is needed.
8. as_create, which allocates memory for the addrspace struct and essentially initializes all addrspace fields to 0.
9. as_destroy, which frees the addrspace struct (or would, if kfree worked).
10. To activate a user addrspace, dumbvm checks that the current address space is valid, and then invalidates all entries in the TLB.
11. A region is a set of pages designated for different access controls. There are two per address space.
12. load_elf uses as_define_region to create a separate region to load the executable into.
13. Nope, addrspace regions aren't protected in the dumbvm.
14. Regions CAN be accessed through the addrspace struct, but as far as I can most memory is accessed directly through the physical address.
15. Address space functions (as_prepare_load, as_define_region, etc) access addrspace regions for intialization.
16. as_complete_load currently does nothing. When project 3 is completed, it will likely set the addrspace region containing the executable bytecode to readonly, and any other post-elf-loading memory management to be done.
17. as_define_stack returns an initial pointer to the stack for a new process.
18. runprogram() in runprogram.c

Part D

1. mips_trap in arch/mips/locore/trap.c
2. Whenever a VM fault is triggered by trying to write read-only memory or a TLB miss.
3. It attempts to write the page into the TLB.

Summary Memory management is done via a two-level page hierarchy: each 4096 byte page maintains a list of sub-pages of varying length. Memory is divided into three segments, for user space and different types of kernel space. The kernel maintains a TLB which the MIPS uses for address translation. Processes own pages via an address space, which can have up to two regions for different access controls (for user processes, a read-only region for the executable byte code and a read-write address space). Dumbvm implements memory management as crudely as possible, and is meant to be replaced in project 3. It essentially abuses some functions meant for bootstrapping to allocate memory, and then crashes when physical memory is full. It doesn't handle de-allocation or regional access controls.

Collaborators and co-conspirators Eli, Zack, Kalen, Gavin, Nathaniel, Austin, Matthew, and Michelle