

Computer Science 161: Operating Systems

VM Examples, Midterm Prep

CS161 Course Staff / GWA
cs161@fas.harvard.edu

March 12, 2007

1 Introduction

1.1 Deadlines

- (03/15/2007): Assignment 2 due at 5 p.m.
- (03/22/2007): Midterm in class.

2 Overview

Today we'll begin by reviewing some of the VM concepts illustrated in class last week. We'll discuss the MIPS R3000 software-managed TLB architecture and how it differs from some of the examples we saw in class.

Next, we'll begin some midterm preparation by working through a midterm question from last year's exam on VM. Last, I'll leave some time at the end for ASST2 hints/pointers.

3 Lecture Review Warm-Up Questions

In particular when presenting VM material in class we are attempting to present you with a somewhat coherent view of a *particular* set of design choices. In particular, when you begin ASST3 you may find some of these design choices helpful; others you may want to reexamine. Let's examine some of these:

- What is “demand paging”? Why is it implemented? Can you think of a time that it may be a drag on performance? (Hint: random disk I/O is **slow!**)
- What are “multi-level” page tables? Why do we use them? What do they allow us to do quickly? And what is the overhead of this performance improvement?
- What is a “software-managed” TLB? How does this differ from a “hardware-managed” TLB? What are the advantages/disadvantages of each?

4 OS/161 “dumbvm” Example

Let's look in detail at what happens in your ASST2 system when a page fault occurs (gdb example).

5 2006 Midterm VM Problem

Let's work through the following problem together:

5.1 2006 Midterm Problem 4: Virtual Memory Management

In this question you will act as the MMU for a simple virtual memory architecture. (A very slow MMU, to be sure, but we won't hold that against you.) This processor has a 16-bit address space, and each address accesses a single 8-bit byte. A two-level page table scheme is used with 16 entries in the top-level page table and 256 entries in the second-level page table. Each page table entry is two bytes wide and has the following format:

<i>1 bit</i>	<i>1 bit</i>	<i>1 bit</i>	<i>1 bit</i>	<i>12 bits</i>
Valid	Readable	Writeable	Executable	Frame number

Page table entries are stored in memory in **big endian** order.

For top-level page tables, the R, W, and X bits are unused and should be set to zero. For second-level page tables, the readable (R) bit indicates whether the page can be read by the process; the writeable (W) bit indicates whether the page can be written by the process; and the executable (X) bit indicates whether the page can be used to fetch instructions for execution.

A partial listing of the machine's physical memory is shown below. (For convenience we are showing the contents of memory four bytes at a time, although the machine is byte-addressed. The first byte in each entry is the lowest-order byte. For example, the value of physical address 0x01 in memory is 0x0b.)

0x00	da 0b 6a ff	0xa0	6b d5 1f 4c	0x140	13 ef 40 4e
0x04	9f 07 7a 8e	0xa4	93 12 4c 1f	0x144	f4 a1 2c 6a
0x08	42 47 b8 2e	0xa8	72 be bc b4	0x148	ed 6e 6e 5c
0x0c	b4 38 60 85	0xac	f2 bc 4e 89	0x14c	ce a2 8b 8f
0x10	0f 1e 21 9e	0xb0	58 19 a0 2d	0x150	58 ae fe b5
0x14	e3 8e 2b 36	0xb4	7f f2 ab 03	0x154	1f a3 0d 82
0x18	f1 98 97 6e	0xb8	1a 01 d2 e1	0x158	1a af 41 78
0x1c	c6 17 d2 41	0xbc	84 01 4b 80	0x15c	e5 9e cb 97
0x20	34 bc e0 07	0xc0	19 ba 67 3f	0x160	fd cd ee f6
0x24	c0 12 d0 04	0xc4	78 82 75 95	0x164	1d e0 0e 18
0x28	50 11 f0 00	0xc8	84 52 00 08	0x168	da 4c 36 a2
0x2c	0b 3f 30 1a	0xcc	30 b5 17 dd	0x16c	49 0c 20 9d
0x30	11 81 d0 72	0xd0	f2 ba 51 0a	0x170	8b 42 47 f3
0x34	b6 ff f2 4c	0xd4	bc 04 ed 19	0x174	ac 3e e7 89
0x38	53 f0 13 31	0xd8	57 84 dd 3f	0x178	78 49 5d ba
0x3c	e6 30 d2 71	0xdc	5a 2a 3d 4e	0x17c	34 43 4d 8e
0x40	f3 ce a4 28	0xe0	b9 a7 89 d8	0x180	c8 96 d8 40
0x44	16 31 61 7c	0xe4	6c d2 d8 65	0x184	56 78 69 ea
0x48	96 7b d6 5b	0xe8	3a b3 87 81	0x188	90 37 16 89
0x4c	38 ab 42 06	0xec	58 ea f6 78	0x18c	f2 89 34 f1
0x50	e3 bb ef e6	0xf0	d6 94 50 24	0x190	19 e4 26 16
0x54	c0 5a 18 25	0xf4	1a f4 90 74	0x194	17 90 91 a0
0x58	64 0f aa ac	0xf8	c5 ee 69 84	0x198	1d ad 95 de
0x5c	0b a1 c6 d4	0xfc	b8 ba a8 92	0x19c	af 87 3d d7
0x60	95 95 b9 be	0x100	80 04 80 01	0x1a0	fe 42 dd 41
0x64	72 d2 07 b6	0x104	00 10 00 00	0x1a4	52 d2 0b 3a
0x68	c7 1a a3 ca	0x108	00 03 80 07	0x1a8	44 06 28 55
0x6c	0b c2 01 f1	0x10c	00 a7 80 02	0x1ac	31 f3 b4 ee
0x70	c9 f7 f8 88	0x110	80 03 00 41	0x1b0	31 56 3d 23
0x74	cc 12 c2 50	0x114	00 16 80 06	0x1b4	8c d1 19 c6
0x78	6b f7 30 7b	0x118	00 03 80 00	0x1b8	fa 69 8b 2f
0x7c	05 ca 78 2b	0x11c	80 05 00 09	0x1bc	75 16 cb f7
0x80	ad 6c 01 22	0x120	89 2d c6 d5	0x1c0	32 5a 6d 2c
0x84	44 16 6e 48	0x124	b2 d3 53 58	0x1c4	73 23 20 04
0x88	02 ce 70 3e	0x128	8a be 9b 09	0x1c8	02 55 bb 8d
0x8c	5b fd ab 63	0x12c	e1 7a d9 12	0x1cc	62 ee 09 20
0x90	3a d1 2b ca	0x130	22 9c 18 67	0x1d0	62 db d1 5c
0x94	d0 df 45 e6	0x134	b5 37 23 8f	0x1d4	2b b7 42 2b
0x98	b0 d7 14 06	0x138	b6 94 c6 d8	0x1d8	eb 3c 2f ec
0x9c	2d 15 31 1a	0x13c	cc f7 3f e8	0x1dc	2c 0f 44 29

Question 4a (1 pt). What is the size of each page in bytes?

Question 4b (1 pt). What is the maximum size of the physical memory in bytes?

Question 4c (8 pts - 0.5 pts per entry). Assume the current process's top-level page table starts at physical address 0x100. List the contents of the top-level page table here. (Hint: How can the R, W, and X bits in the top-level page table tell you that you are decoding the memory correctly?)

Index	Valid	PFN
0		
1		
2		
3		
4		
5		
6		
7		
8		
9		
10		
11		
12		
13		
14		
15		

Question 4d (20 pts - 4 pts per entry). Translate each of the following virtual addresses into a physical address. (If you want us to follow your answer, it is probably a good idea to clearly write down the primary and secondary page number in the address, the corresponding page table entry, and so forth.)

Virt addr	Valid?	Readable?	Writeable?	Executable?	Physical address
0x702d					
0x60bb					
0x27f3					
0x7006					
0x7015					

6 ASST2 Hints

- **File System Calls** (`open()`, `close()`, `read()`, `write()`, etc.): think carefully about the design of your data structures here. You should probably not be adding any new *global kernel* data structures, only per-process ones. Think about what happens on `fork()` in particular.
- **Process Support:** you should probably *not* be modifying any architecture-dependent structures to complete this assignment!
- **`execv()`:** you *really* want to get the layout of the arguments right the first time, since this can be a huge headache to debug. Work it out on a whiteboard; several times. Then write pseudo-code. Then translate that to C. Then cross your fingers. You also need to think a bit about buffer management.
- **`fork()`:** there are already functions we provide you that do most of the heavy lifting for `fork()`. However, think about how the parent and child return to userland (easier for the parent). There are two sticking points here: a) synchronization and b) modifying the return value for the child. Otherwise things are straightforward.
- **`waitpid()/_exit()`:** make sure you look at the man pages carefully here. Usually UNIX semantics dictate that a parent can **always** recover the exit code of its child by calling `waitpid()`, regardless of whether or not the child has exited previously! This pair also provides a great chance to use one of the synchronization primitives you developed for ASST1.
- **`getpid()`:** this should be really, very simple. Let's use it as a chance to do a little systems design tutorial...
- **Testing:** you are responsible for understanding and using the contents of `bin/` and `testbin/` to test your assignment. For ASST2 in particular, among others, you will probably be interested in running `testbin/forktest`, `testbin/badcall` and `testbin/randcall`, as well as a much of things from `bin/` including `bin/sh`.