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:

<table>
<thead>
<tr>
<th>1 bit</th>
<th>1 bit</th>
<th>1 bit</th>
<th>1 bit</th>
<th>12 bits</th>
</tr>
</thead>
<tbody>
<tr>
<td>Valid</td>
<td>Readable</td>
<td>Writeable</td>
<td>Executable</td>
<td>Frame number</td>
</tr>
</tbody>
</table>

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.)
**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?)

<table>
<thead>
<tr>
<th>Index</th>
<th>Valid</th>
<th>PFN</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td></td>
<td></td>
</tr>
<tr>
<td>1</td>
<td></td>
<td></td>
</tr>
<tr>
<td>2</td>
<td></td>
<td></td>
</tr>
<tr>
<td>3</td>
<td></td>
<td></td>
</tr>
<tr>
<td>4</td>
<td></td>
<td></td>
</tr>
<tr>
<td>5</td>
<td></td>
<td></td>
</tr>
<tr>
<td>6</td>
<td></td>
<td></td>
</tr>
<tr>
<td>7</td>
<td></td>
<td></td>
</tr>
<tr>
<td>8</td>
<td></td>
<td></td>
</tr>
<tr>
<td>9</td>
<td></td>
<td></td>
</tr>
<tr>
<td>10</td>
<td></td>
<td></td>
</tr>
<tr>
<td>11</td>
<td></td>
<td></td>
</tr>
<tr>
<td>12</td>
<td></td>
<td></td>
</tr>
<tr>
<td>13</td>
<td></td>
<td></td>
</tr>
<tr>
<td>14</td>
<td></td>
<td></td>
</tr>
<tr>
<td>15</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
**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.)

<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>0x702d</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>0x60bb</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>0x27f3</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>0x7006</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>0x7015</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
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. Usualy 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.