CS350 **Operating Systems** **Winter 2009**

Assignment Three

1 Introduction

OS/161 has a very simple virtual memory system, called *dumbvm*. Assignment 3 is to replace dumbvm with a new virtual memory system that relaxes some (not all) of dumbvm's limitations. Your new system will implement a replacement policy for the TLB, so that the kernel will not crash if the TLB fills up. It will also implement *on-demand loading* of pages. This will allow programs that have address spaces larger than physical memory to run, provided that they do not touch more pages than will fit in physical memory.

2 Code Review

As was the case for the first OS/161 assignment, you should begin with a careful review of the existing OS/161 code with which you will be working. The rest of this section of the assignment identifies some important files for you to consider. There are no code reading questions to be answered for Assignment 3. Nonetheless, it is important for you to understand these files.

In `kern/vm`

The machine-independent part of your virtual memory implementation should go in this directory. Currently, the only file here is `addrspace.c`, which contains stub implementations of several of the functions that you will need to implement. You may wish to add additional files in this directory.

In `kern/userprog`

`loadelf.c`: This file contains the functions responsible for loading an ELF executable from the filesystem into virtual memory space. You should already be familiar with this file from Assignment 2. Since you will be implementing on-demand page loading in Assignment 3, you will need to change the behaviour that is implemented here.

In `kern/include`

`addrspace.h`: Defines the addrspace interface. You will need to make changes here, at least to define an appropriate addrspace structure.

`vm.h`: Some VM-related definitions, including prototypes for some key functions, such as `vm_fault` (the TLB miss handler) and `alloc_kpages` (used, among other places, in `kmalloc`).

In `kern/arch/mips/mips`

The most important file in this directory is `dumbvm.c`. This file is not used at all in Assignment 3. However, you can use the code here as a starting place for your Assignment 3 work. This code also includes examples of how to do things like manipulate the TLB.

In `kern/arch/mips/include`

In this directory, the file `tlb.h` defines the functions that are used to manipulate the TLB.

3 Implementation Requirements

All code changes for this assignment should be enclosed in `#if OPT_A3` statements, as you have done with `OPT_A2` and `OPT_A1` in the previous assignments. For this to work, you must add `#include "opt-A3.h"` at the top of any file for which you make changes for this assignment.

By default, any code changes that you made for Assignments 1 and 2 will be included in your build when you compile for Assignment 3.

3.1 TLB Management

In the System/161 machine, each TLB entry includes a 20-bit virtual page number and a 20-bit physical page number as well as the following five fields:

global (1 bit): If set, ignore the pid bits in the TLB.

valid (1 bit): When the valid bit is set, the TLB entry is supposed to contain a valid translation. This implies that the virtual page is present in physical memory. A TLB miss exception (`EX_TLBL` or `EX_TLBS`) occurs when no valid TLB entry that maps the required virtual page is present in the TLB.

dirty (1 bit): In class, we used the term "dirty bit" to refer to a bit that is set by the MMU to indicate that a page has been modified. OS/161's "dirty" bit is *not* like this - it indicates whether it is possible to modify a particular page. OS/161 can clear this bit to indicate that a page is read-only, and to force the MMU to generate an `EX_MOD` exception if there is an attempt to write to the page.

nocache (1 bit): Unused in System/161. In a real processor, indicates that the hardware cache will be disabled when accessing this page.

pid (6 bits): A context or address space ID that can be used to allow entries to remain in the TLB after a context switch.

In OS/161, the global and pid fields are unused. This means that all of the valid entries in the TLB should describe pages in the address space of the currently running process, and the contents of the TLB should be invalidated when there is a context switch.

For this assignment, you are expected to write code to manage the TLB. When a TLB miss occurs, OS/161's exception handler should load an appropriate entry into the TLB. If there is free space in the TLB, the new entry should go into free space. Otherwise, OS/161 should choose a TLB entry to evict, and evict it to make room for the new entry. You should implement a very simple *first-in-first-out* replacement policy for the TLB. OS/161 should also take care of ensuring that all TLB entries refer to the currently running process. Presumably you will do this by invalidating the contents of the TLB when there is a context switch.

3.2 Read-Only Text Segment

In dumbvm, all three address space segments (text, data, and stack) are both readable and writable by the application. For this assignment, you should change this so that each application's text segment is *read-only*. Your kernel should set up TLB entries so that any attempt by an application to modify its text section will cause the MIPS MMU to generate a readonly memory exception (`VM_FAULT_READONLY`). If such an exception occurs, your kernel should terminate the process that attempted to modify its text segment. Your kernel should *not* crash.

3.3 On-Demand Page Loading

Currently, when OS/161 loads a new program into an address space using `runprogram` (and, presumably, in your implementation of `execv` from Assignment 2), it pre-allocates physical frames for all of the program's virtual pages, and it pre-loads all of the pages into physical memory.

For this assignment, you are required to change this so that physical frame are allocated on-demand and virtual pages are loaded on demand. "On-demand" means that that the page should be loaded (and physical

space should be allocated for it) the first time that the application tries to use (read or write) that page. Pages that are *never* used by an application should never be loaded into memory, and should not consume a physical frame.

In order to do this, your kernel will need to have some means of keeping track of which parts of physical memory are in use, and which parts can be allocated to hold newly-loaded virtual pages. Your kernel will also need a way to keep track of which pages from each address space have been loaded into physical memory, and where in physical memory they have been loaded.

Since a program's pages will not be pre-loaded into physical memory when the program starts running, and since the TLB only maps pages that are in memory, the program will generate TLB miss exceptions as it tries to access its virtual pages. Here is a high-level description of what the OS/161 kernel must do when the MMU generates a TLB miss exception for a particular page:

1. Determine whether the page is already in memory.
2. If it is already in memory, load an appropriate entry into the TLB (replacing an existing TLB entry if necessary) and then return from the exception.
3. Otherwise, allocate a place in physical memory to store the page
4. Load the page, using information from the program's ELF file to do so
5. Update OS/161's information about this address space.
6. Load an appropriate entry into the TLB (replacing an existing TLB entry if necessary), and return from the exception.

You are *not* required to implement page replacement for this assignment, nor is your OS/161 required to reclaim physical space that is no longer needed by application programs (or by dynamically created kernel data structures that have been freed). Because you will not implement reclamation of physical frames, the total number of pages that can be loaded into memory over the life of your kernel will be limited by the physical memory size of the machine. Because you will not implement page replacement, you will not be able to run applications that touch more pages than will fit into physical memory. However, you should be able to run large programs provided that those programs do not touch more pages than will fit into memory.

3.4 Instrumentation

You are required to add instrumentation to your kernel to collect and display some statistics related to virtual memory activity. You are required to collect the following statistics:

TLB Faults: The number of TLB misses that have occurred

TLB Faults with Free: The number of TLB misses for which there was free space in the TLB to add the new TLB entry.

TLB Faults with Replace: The number of TLB misses for which there was no free space for the new TLB entry, and replacement was required.

TLB Invalidations: The number of times the TLB was invalidated.

TLB Reloads: The number of TLB misses for pages that were already in memory.

Page Faults (Disk): The number of TLB misses that required a page to be loaded from disk.

Page Faults (Zeroed): The number of TLB misses that required a new page to be zero-filled.

Note that the sum of "TLB Faults with Free" and "TLB Faults with Replace" should be equal to "TLB Faults". Also, the sum of "TLB Reloads", "Page Faults (Disk)", and "Page Faults (Zeroed)" should also be equal to "TLB Faults".

When it is shut down (e.g., in `vm_shutdown`), your kernel should display the statistics it has gathered. The display should look approximately like the example below. In particular, each line of output should begin with the characters "VMSTAT" so that we can easily identify it.

```
Shutting down.
VMSTAT  TLB Faults = 470
VMSTAT  TLB Faults with Free = 470
VMSTAT  TLB Faults with Replace = 0
VMSTAT  TLB Invalidations = 277
VMSTAT  TLB Reloads = 466
VMSTAT  Page Faults (Disk) = 3
VMSTAT  Page Faults (Zeroed) = 1
The system is halted.
sys161: 278170480 cycles (41840449k, 299023u, 236031008i)
sys161: 10216 irqs 8610 exns 0r/0w disk 18r/9181w console 7r/0w/5m emufs 0r/0w net
sys161: Elapsed real time: 11.156885 seconds (24.9326 mhz)
sys161: Elapsed virtual time: 11.126819200 seconds (25 mhz)
```

4 Discussion

This section provides some additional discussion of aspects of OS/161 that are relevant to the assignment.

4.1 Configuring and Building

Before you do any coding for Assignment 3, you will need to reconfigure your kernel for this assignment. Follow the same procedure that you used to configure for the first two assignments, but use the ASST3 configuration file instead:

```
% cd cs350-os161/os161-1.11/kern/conf
% ./config ASST3
% cd ../compile/ASST3
% make depend
% make
% make install
```

Warning: Once you reconfigure your kernel for Assignment 3, it will probably not compile. Even if it does compile, it will no longer work. This is normal. It happens because your kernel relied, for Assignments 1 and 2, on the `dumbvm` virtual memory implementation. Starting with Assignment 3, the `dumbvm` implementation is no longer configured to be part of your kernel, i.e., it is ignored. For this assignment, you will be re-implementing the functions that were formerly implemented by `dumbvm`. Until you do that, anything that relied on `dumbvm` (which means most of your kernel) will not work.

4.2 kmalloc and kfree

Your kernel needs to be able to dynamically allocate memory using `kmalloc`. Space that is allocated by `kmalloc` consumes physical memory (of course), and that physical memory is (of course) not available to hold applications' virtual pages. You will need to review the implementation of `kmalloc` to learn how it allocates physical space, and you will need to ensure that your virtual memory implementation is compatible with `kmalloc`, so that `kmalloc` will work correctly when it needs to obtain more physical memory.

`kfree`, as provided to you, does not actually release physical pages once they have been consumed by `kmalloc`. This means that once a physical page has been claimed by `kmalloc`, there is no way to "give it back" so that it can be used to hold an application page. You are *not* required to fix this problem for Assignment 3. `kfree` should still work as it did in the first two assignments, but you do not need to improve it.

5 Where to Start

You should start your work on this assignment by studying and understanding how the dumbvm implementation works. Also have a look at the addrspace stubs, and think about what changes you will need to make to dumbvm to meet the requirements for this assignment.

Your initial focus should be on getting your kernel *back* to a state in which you can run a single simple program (e.g., palin) from the kernel command menu.

Once this is done, we recommend that you implement the various parts of the assignment in the following order:

1. TLB Management
2. Instrumentation
3. Read-Only Text Segment
4. On-Demand Page Loading

Some of our testing will be based on the statistics that are produced by your instrumentation, so you should definitely get that working before you start work on the on-demand page loading, which is the biggest part of the assignment.

6 Design Document

You are expected to prepare a short design document describing your virtual memory implementation. Your design document must address at least the following issues:

physical memory: What information about physical memory does your kernel track? Where is this information recorded? Why is this particular information recorded? Are there any synchronization issues that arise when this information is used? Why or why not?

virtual address space segments: What information is recorded about each segment, where is that information recorded, and why is it recorded? Are there any synchronization issues that arise when it is used?

virtual address space pages: What information is recorded about each page, where is that information recorded, and why is it recorded? Are there any synchronization issues that arise when it is used?

TLB management: How do you ensure that read-only pages are not modified? How does your kernel construct the necessary TLB entry when a TLB miss occurs? When does your kernel invalidate the TLB? Are there any synchronization issues that arise in TLB management?

If portions of your design are not implemented, you may still receive some credit for a high-quality design even if it is not implemented. However, if your implementation does not match your design, **your design document must clearly indicate which parts of the design are not implemented, or are not implemented as described in the design document. Submission of a design document that does not match the submitted implementation (and which does not flag any such differences) will be considered an act of academic dishonesty.**

Please prepare your design document as a PDF file called `design3.pdf`. Your design document should be *at most* three pages long using 1 inch margins and a minimum font size of 11 points.

7 What to Submit

You should submit your kernel source code and your design document using the `cs350_submit` command, as you did for the previous assignments. It is important that you use the `cs350_submit` command. Do not use the regular `submit` command directly.

Assuming that you followed the standard OS/161 setup instructions, your OS/161 source code will be located in `$HOME/cs350-os161/os161-1.11`. To submit your work, you should

1. place `design3.pdf` in the directory `$HOME/cs350-os161/`

2. run `/u/cs350/bin/cs350_submit` in the directory `$HOME/cs350-os161/`

This will package up your OS/161 kernel code and submit it, along with the PDF file, to the course account.

Important: The `cs350_submit` script packages and submits everything under the `os161-1.11/kern` directory, except for the subtree `os161-1.11/kern/compile`. You are permitted to make changes to the OS/161 source code outside the `kern` subdirectory. For example, you might create a new test program under `testbin`. However, such changes will not be submitted when you run `cs350_submit`. Only your kernel code, under `os161-1.11/kern`, will be submitted.