CS350 **Operating Systems** Winter 2020

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 a few of dumbvm's limitations.

2 Code Review

As usual, 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.

In `kern/vm`

This directory is intended to hold machine-independent parts of the virtual memory implementation.

- `kmalloc.c`: This file contains implementations of `kmalloc` and `kfree`, to support dynamic memory allocation for the kernel.
- `uw-vmstats.c`: contains code for tracking statistics related to the virtual memory system. Not used for this assignment.
- `copyinout.c`: contains code for copying data between applications and the kernel.

In `kern/syscall`

`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.

In `kern/include`

`addrspace.h`: Defines the addrspace interface. You may 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/vm`

`dumbvm.c`: This file contains the implementation of the dumbvm virtual memory system, including the implementation of the `addrspace` functions and `vm_fault`, which is the kernel's handler function for exceptions related to virtual memory. It also includes some very simple functions (e.g., `alloc_kpages`) for physical memory management.

`ram.c`: This file includes functions that the kernel uses to manage physical memory (RAM) while the kernel is booting up, before the VM system has been initialized. Since your VM system will essentially be taking over management of physical memory, you need to understand how these functions work.

In `kern/arch/mips/include`

`tlb.h`: This file defines the prototypes for the kernel's functions for managing the TLB (such as `tlb_write`), as well as definitions of the fields in each TLB entry.

`vm.h`: This file defines some macros and constants (such as the page size) related to address translation on the MIPS. Note that this `vm.h` is different from the `vm.h` in `kern/include`.

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 Handling a Full TLB

Currently, the dumbvm system will panic and crash if the TLB fills up with valid entries. Change this so that the kernel will not panic in this situation. If kernel needs to add a new entry to the TLB and the TLB is full, the kernel should simply overwrite one of the existing entries with the new entry. A simple way to do this is to use the `tlb_random` function to allow the hardware to choose a random entry to be overwritten.

The `vm_fault` function is responsible for managing the TLB, so you should be able to implement this part of the assignment with some small changes to that function.

You should get this part of the assignment finished first and test that it works correctly, because many of the test programs used for the other parts of the assignment will not work properly if the kernel panics when the TLB fills up.

3.2 Read-Only Text Segment

In the dumbvm system, 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 read-only 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.

Get this part of the assignment finished and tested before moving on to physical memory management.

3.3 Physical-Memory Management

The dumbvm virtual memory system has two severe limitations in the way it manages physical memory. First, it assumes that each segment will be allocated contiguously in physical memory. Second, it never re-uses physical memory. That is, when a process exits, the physical memory that was used to hold that process's address space does not become available for use by other processes. As a result, the kernel will quickly run out of physical memory.

Your objective in this assignment is to remove both of these limitations. In particular:

- It should be possible for the pages in process' address spaces to be placed into *any* free frame of physical memory. That is, the kernel should no longer require that address space segments be stored contiguously in physical memory.
- When a process terminates, the physical frames that were used to hold its pages should be freed, and should become available for use by other processes.

To implement these changes, your kernel will need some way of keeping track of where each page in each process's virtual address space is located, and some way of keeping track of which frames of physical memory are free, and which are in use.

The kernel uses physical memory both to hold its own data structures and to hold the address spaces of user processes. The kernel uses `kmalloc` and `kfree` to dynamically allocate space for new kernel data structures. `kmalloc` and `kfree`, in turn, call `alloc_kpages` and `free_kpages` to allocate or free physical memory as necessary. When you change the way the kernel's physical memory management works to satisfy the requirements above, *you must also ensure that* `kmalloc` *and* `kfree` *continue to work properly.* In addition, you should ensure that any physical pages freed by `kfree` (when it calls `free_kpages`) become free and available for re-use by the kernel. This will require you to implement `free_kpages`, which currently

does nothing. Note that there is no need for you to modify the implementations of `kmalloc` and `kfree` themselves. Instead, you should modifiy the functions (`alloc_kpages` and `free_kpages`) that `kmalloc` and `kfree` use to allocate and free physical memory.

Finally, note that when the kernel is booting—and before your VM system has been initialized—the kernel will allocate memory for its data structures. OS/161 has a simple physical-memory allocation mechanism (in `arch/mips/mips/ram.c`) which will support these early memory allocations. When you initialize your VM system, you will need to make sure that it is aware that some parts of physical memory are already being used by the kernel, so that it does not mistakenly allocate that physical memory for some other purpose. Your VM system is not required to re-use any parts of physical memory that are allocated to the kernel before your VM system is initialized—though it is allowed to do so, provided the kernel frees up the space. Once your VM system has been initialized during the boot process, all subsequent physical-memory requests (via `alloc_kpages` and `free_kpages`) from the kernel should be handled by your VM system.

4 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/os161-1.99/kern/conf
% ./config ASST3
% cd ../compile/ASST3
% bmake depend
% bmake
% bmake install
```

5 What to Submit

You should submit your kernel source code 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 `$HOME/cs350-os161/os161-1.99`. To submit your work, you should run `/u/cs350/bin/cs350_submit 3` in the directory `$HOME/cs350-os161/`. This will package up your OS/161 kernel code and submit it to the course account.

Important: The `cs350_submit` script packages and submits everything under the `os161-1.99/kern` directory, except for the subtree `os161-1.99/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 `my-testbin`. However, such changes will not be submitted when you run `cs350_submit`. Only your kernel code, under `os161-1.99/kern`, will be submitted.