CS350 **Operating Systems** **Winter 2013**

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. In addition, you will implement page replacement.

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/lib`

`kheap.c`: This file contains implementations of `kmalloc` and `kfree`, to support dynamic memory allocation for the kernel. It should not be necessary for you to change the code in this file, but you do need to understand how the kernel's dynamic memory allocation works so that your physical-memory manager will interact properly with it.

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`

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

`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`

In this directory, the file `tlb.h` defines the functions that are used to manipulate the TLB. In addition, `vm.h` includes some macros and constants 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 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. In the `dumbvm` system, TLB invalidation is accomplished by the `as_activate` function, which invalidates all of the TLB entries. OS/161's context-switch code calls `as_activate` after every context switch (as long as the newly running thread has an address space). If you preserve this functionality in `as_activate` in your new VM system, you will have taken care of TLB invalidation. However, this approach is inefficient since it invalidates the TLB even when it is not necessary to do so. A more efficient approach, which you should use in your kernel, is to invalidate the TLB entries only if the address space changes during the 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. As described above, OS/161 should also take care to ensure that all TLB entries refer to the currently running process.

Round-Robin TLB Replacement

Your kernel must implement a very simple *round-robin* TLB replacement policy. This is like first-in-first-out except that we do not actually worry about when each page was replaced and the algorithm works as follows:

```
int
tlb_get_rr_victim()
{
        int victim;
        static unsigned int next_victim = 0;

        victim = next_victim;
        next_victim = (next_victim + 1) % NUM_TLB;
        return victim;
}
```

Instrumentation

You will be tracking and printing several statistics (more details will be provided in Section 3.4) related to the performance of the virtual memory sub-system (including TLB misses and TLB replacements) so be certain to implement TLB replacement as described so we can easily examine and compare these statistics.

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.

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 frames 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:

- Determine whether the page is already in memory.
- 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.
- If it is not already in memory, then

- Allocate a place in physical memory to store the page.
- Load the page, using information from the program's ELF file to do so.
- Update OS/161's information about this address space.
- Load an appropriate entry into the TLB (replacing an existing TLB entry if necessary) and return from the exception.

Until you implement page replacement, you will not be able to run applications that touch more pages than will fit into physical memory, but you should be able to run large programs provided that those programs do not touch more pages than will fit.

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 (not including faults that cause a program to crash).

TLB Faults with Free: The number of TLB misses for which there was free space in the TLB to add the new TLB entry (i.e., no replacement is required).

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

TLB Invalidations: The number of times the TLB was invalidated (this counts the number times the entire TLB is invalidated NOT the number of TLB entries invalidated)

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

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

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

Page Faults from ELF: The number of page faults that require getting a page from the ELF file.

Page Faults from Swapfile: The number of page faults that require getting a page from the swap file.

Swapfile Writes: The number of page faults that require writing a page to the swap file.

The last two statistics relate to page replacement, which is discussed in Section 3.6. 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 be equal to "TLB Faults." So this means that you should not count TLB faults that do not get handled (i.e., result in the program being killed). The code for printing out stats will print a warning if this these equalities do not hold. In addition the sum of "Page Faults from ELF" and "Page Faults from Swapfile" should be equal to "Page Faults (Disk)".

You are provided with code to collect the necessary statistics in `kern/vm/uw-vmstats.c`. Be sure to use this code to collect the statistics. When your kernel is shut down (e.g., in `vm_shutdown`), it should display the statistics it has gathered. The display should look like the example below.

```
Shutting down.
VMSTATS:
VMSTAT                TLB Faults =       6744
VMSTAT      TLB Faults with Free =       3303
VMSTAT   TLB Faults with Replace =       3441
VMSTAT         TLB Invalidations =          1
VMSTAT               TLB Reloads =       3318
```

```
VMSTAT      Page Faults (Zeroed) =        513
VMSTAT        Page Faults (Disk) =       2913
VMSTAT      Page Faults from ELF =         19
VMSTAT Page Faults from Swapfile =       2894
VMSTAT           Swapfile Writes =       3221
VMSTAT TLB Faults with Free + TLB Faults with Replace = 6744
VMSTAT TLB Reloads + Page Faults (Zeroed) + Page Faults (Disk) = 6744
VMSTAT ELF File reads + Swapfile reads = 2913
The system is halted.
```

3.5 Physical-Memory Management

The dumbvm system does not provide any way to re-use physical memory. Once physical memory has been allocated, there is no way to free it when it is no longer needed. As a result, the kernel will quickly run out of physical memory.

Your VM system must manage physical memory so that it can be re-used. In particular, when a process terminates, any physical frames that were used to hold that process's pages should be freed, and they should be available for use to hold pages from other processes. This will require your kernel to provide some mechanism for keeping track of which frames are free and which have been allocated.

Your physical-memory manager should also support dynamic memory allocation of kernel data structures, via `kmalloc` and `kfree`. The simplest way to do this is to change the implementations of the functions `alloc_kpages` and `free_kpages`. `alloc_kpages` is called by `kmalloc` when it needs additional pages, i.e., frames) of physical memory to manage. Similarly, `free_kpages` is called by `kfree` if there are frames that it no longer requires. You should modify `alloc_kpages` to obtain the frames it needs from your physical-memory manager. Similarly, `free_kpages` should return freed frames to your physical-memory manager so that it can re-use them for other purposes.

Note that some of the physical memory of the machine is used to hold the kernel's code and data. In addition, when the kernel is booting—and before your VM system has been initialized—the kernel will dynamically allocate additional 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 of which parts of physical memory are already being used by the kernel, so that it does not mistakenly allocate them 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, all subsequent physical-memory requests (via `alloc_kpages` and `free_kpages`) from the kernel should be handled by your VM system.

3.6 Page Replacement

You should implement a page replacement policy of your choosing. A very simple (even poor) algorithm that works is preferred to a more complex algorithm that you can't get to work. Do not worry about implementing techniques to avoid or control thrashing. Pages that need to be written to disk should be written to a file named `SWAPFILE`. This file will be limited to 9 MB (i.e., 9 * 1024 * 1024 bytes). If at run time more than 9 MB of swap space is required your kernel should call `panic("Out of swap space")`. **Make the maximum size of the swap file easy to change at compile time, in case we need to change this requirement before final submissions.**

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/os161-1.11/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. In 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.

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.

Be sure to read the online hints for the 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
5. Physical-Memory Management
6. Page Replacement

It is probably best to implement and test these features in this order. Keeping in mind that in the end you would like to have all features implemented.

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 later parts of the assignment.

6 Source Code Organization

We will be spending some time looking your code and some of your mark for this assignment will be based on the code. We will be checking if we can easily find what we are looking for, if a proper style is used that is consistent with the preexisting code, if comments properly are used, and if we can read and understand the code easily.

You are free and will need to modify existing kernel code. In addition, you will probably need some code to create and use some new abstractions. If you uses any or all of the following abstractions please place that code in the directory `os161-1.11/kern/vm` using the following file names:

- `addrspace.c`: some code and stubs are provided, you may need to add things here (things that were in dumbvm.c but are not provided in Assignment 3, e.g, `vm_fault`).
- `coremap.c`: keep track of free physical frames
- `pt.c`: page tables and page table entry manipulation go here

- `segments.c`: code for tracking and manipulating segments
- `vm_tlb.c`: code for manipulating the TLB (including replacement)
- `swapfile.c`: code for managing and manipulating the swapfile
- `uw-vmstats.c`: is already provided (for tracking stats)

If you need them, corresponding header files should be placed in `os161-1.11/kern/include` in files named: `addrspace.h, coremap.h, pt.h, segments.h, vm_tlb.h`, and `swapfile.h`.

The file `os161-1.11/kern/include/uw-vmstats.h` is already provided for you.

7 Design Questions

You are expected to prepare short answers to the following questions about your work on A3.

Question 1: Briefly describe the data structure(s) that your kernel uses to manage the allocation of physical memory. What information is recorded in this data structure? When your VM system is initialized, how is the information in this data structure initialized?

Question 2: When a single physical frame needs to be allocated, how does your kernel use the above data structure to choose a frame to allocate? When a physical frame is freed, how does your kernel update the above data structure to support this?

Question 3: Does your physical-memory system have to handle requests to allocate/free multiple (physically) contiguous frames? Under what circumstances? How does your physical-memory manager support this?

Question 4: Are there any synchronization issues that arise when the above data structures are used? Why or why not?

Question 5: Briefly describe the data structure(s) that your kernel uses to describe the virtual address space of each process. What information is recorded about each address space?

Question 6: When your kernel handles a TLB miss, how does it determine whether the required page is already loaded into memory?

Question 7: If, on a TLB miss, your kernel determines that the required page is not in memory, how does it determine where to find the page?

Question 8: How does your kernel ensure that read-only pages are not modified?

Question 9: Briefly describe the data structure(s) that your kernel uses to manage the swap file. What information is recorded and why? Are there any synchronization issues that need to be handled? If so what are they and how were they handled?

Question 10: What page replacement algorithm did you implement? Why did you choose this algorithm? Is this a good choice, why or why not? What were some of the issues you encountered when trying to design and implement this algorithm?

If portions of your design are not implemented, you may still receive some credit for well thought-out answers to these questions, but 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 your answers. Submission of answers that do not match the submitted implementation (and which do 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`. This document should be *at most* four pages long using a 10-point (or larger) font and 3/4-inch (or larger) top, bottom and side margins.

8 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.