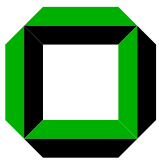

University of Karlsruhe
System Architecture Group
Gerd Liefländer

Teaching Assistant:
Philipp Kupferschmied

System Architecture 2008/09
Programming Assignment 3

Submission Deadlines: 23.01.2009, 12:00h (theoretical part)
13.02.2009, 12:00h (practical part)

1 Introduction

In this assignment you will implement the virtual memory subsystem of OS/161. The existing VM implementation in OS/161, `dumbvm`, is a minimal implementation with a number of shortcomings. In this assignment you will adapt OS/161 to take full advantage of the simulated hardware by implementing support for the software-managed translation look-aside buffer (TLB) available in the MIPS architecture. You will also write code to manage the physical system memory.

2 Overview on System/161 Memory Management Hardware

System/161 simulates the MIPS architecture, including TLB and memory map as will be discussed in this section.

2.1 The System/161 TLB

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

global: (1 bit) If set, the pid-bits in this TLB entry are ignored

valid: (1 bit), If set, this TLB entry contains a valid translation

dirty: (1 bit), If set, this page has been modified since it has been loaded into memory (e.g., from an executable). If clear, any `store` operation to this page will raise a *TLB Modified Exception*; the handler should then either allow write access by setting this bit or cause the faulting thread (or process) to be killed.

nocache: (1 bit) If set, the hardware will disable the cache for this page. *The nocache bit is unsupported in System/161.*

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

All these bits/values are maintained by the operating system. When the valid bit is set, the TLB entry contains a valid translation. This implies that the virtual page is present in physical memory. A TLB miss occurs when no valid TLB entry with a matching virtual page number and the current address space ID (PID) or set `global` bit is found.
Note: For this assignment, we strongly recommend that you ignore the `pid` field and mark all your pages as `global`. However, you must then flush the TLB on every context switch (why?).

2.2 The System/161 Virtual Address Space Map

The MIPS architecture divides its address space into several regions that have hardwired properties related to memory management. These are:

kseg2: TLB-mapped, cacheable kernel space

kseg1: direct-mapped, uncacheable kernel space

kseg0: direct-mapped, cacheable kernel space

kuseg: TLB-mapped cacheable user space

Both direct-mapped segments map to the first 512 MB of physical address space.
kuseg spans the lower 2 GB (0x00000000–0x7fffffff), kseg0 takes the following 512 MB (0x80000000–0x9fffffff), followed by 512 MB assigned to kseg1 (0xa0000000–0xbfffffff), leaving the final 1 GB of virtual addresses for kseg2 (0xc0000000–0xffffffff).

virtual address range	segment	special properties
`0xc0000000 - 0xffffffff`	kseg2	
`0xbfc00180 - 0xbfffffff`	kseg1	Exception handler if BEV is set.
`0xbfc00100 - 0xbfc0017f`	kseg1	UTLB exception handler if BEV is set.
`0xbfc00000 - 0xbfc000ff`	kseg1	Reset handler (start-up code).
`0xa0000000 - 0xbfbfffff`	kseg1	
`0x80000080 - 0x9fffffff`	kseg0	Exception handler if BEV is clear.
`0x80000000 - 0x8000007f`	kseg0	UTLB exception handler if BEV is clear.
`0x00000000 - 0x7fffffff`	kuseg	

3 Setting up the Assignment

In this section, you will create a new repository for the source code of this assignment.

3.1 Obtaining and setting up ASST3 in Mercurial

Only one of you must to do the following. Again, *s_hghost* has to perform the following commands.

```
$ mkdir -p ~/sysarch/sharedrepos/asst3-src
$ cd ~/sysarch/sharedrepos/asst3-src
$ hg init
$ cd ~/sysarch/
$ hg clone sharedrepos/asst3-src
$ wget http://i30www.ira.uka.de/~pkupfer/edu/2008ws/sysarch/asst3-src.tbz2
$ tar -xjf asst3-src.tbz2
$ rm asst3-src.tbz2
$ cd ~/sysarch/asst3-src
$ hg add
$ hg commit
$ hg tag asst-base
$ hg push
```

3.2 Obtaining a Working Copy

s_hghost has already cloned the shared repository during the setup process described above. *s_member* can clone it via

```
$ mkdir -p ~/sysarch
$ cd ~/sysarch
$ hg clone ssh://sysarch_hg/asst3-src
```

You are now ready to start with this assignment.

3.3 Building ASST3

Before proceeding any further, configure the sources and build the user land parts of OS/161 via

```
$ cd ~/sysarch/asst3-src
$ ./configure --ostree="$HOME/sysarch/root"
$ make
```

For your kernel development, we have provided you with another framework for ASST3. You have to configure your kernel before you can use this framework. The procedure for configuring and building a kernel is the same as before:

```
$ cd ~/sysarch/asst3-src/kern/conf
$ ./config ASST3
$ cd ../compile/ASST3
$ make
```

If you are told that the `compile/ASST3` directory does not exist, make sure you ran config for ASST3. If you now run the kernel as before (type `./sys161 kernel` in `~/sysarch/root`) you should get to the menu prompt. If you try to run a program, the kernel will panic with a message telling you that `vm_fault()` is not implemented. For example, try `p /bin/true` at the OS/161 menu prompt to run the program from `~/sysarch/root/bin/true`.
You will (again) need to increase the amount of RAM simulated by System/161. We suggest that you now allocate 16 MB to System/161 by editing `~/sysarch/root/sys161.conf`. Change the line for the `busctl` device to read

```
31 busctl ramsize=16777216  # 16 MB of RAM
```

You are now ready to start with this assignment.

4 Tutorial Exercises

Please answer the following questions and send them to `os161@ira.uka.de` before the **first** deadline has passed. The subject of your mail **must** contain your group name (i.e., *sapwX*) and should additionally contain some hint that you are submitting the solution to the theoretical part. You should be familiar enough with navigating the kernel sources that you can find the answers to the questions below by yourselves; you may want to employ tools such as `grep`, `ctags`, or `cscope`. You may also find it useful to look at a MIPS R3000 reference manual. Hand in your answers in plain text, with line-wraps. Please mention your group name and assignment number in the subject line.

Question 3.1: What is the difference between the different MIPS address space segments? What is the use of each segment?

Question 3.2: What functions exist to help you manage the TLB? Describe their use. (Hint: look in `kern/arch/mips/include/tlb.h`)

Question 3.3: Which macros are used to convert from a physical address to a kernel virtual address?

Question 3.4: What address should the initial user stack pointer be?

Question 3.5: What are the `EntryHi` and `EntryLo` coprocessor registers? Describe their contents.

Question 3.6: What do the `as_*()` functions do? Why do we need both `as_prepare_load()` and `as_complete_load()`?

Question 3.7: What does `vm_fault()` do? When is it called?

Question 3.8: Assuming an inverted page table, a physical address space of only 16 frames, and a single page and frame size of 4 kB. Further assume that the physical memory is initially empty and frame allocation sequentially uses frames 0, 1, 2, ...
For accesses to the virtual addresses (a) 0x100008, (b) 0x101008, (c) 0x1000f0, (d) 0x41000, and (e) 0x41b00, show

1. their respective page number and offset,

2. the translated address (after page allocation), and

3. the contents of the page table *after* the TLB miss has been handled.

5 Coding Assignment

This assignment involves designing and implementing a number of data structures.
Before you start, you should work out what data you need to keep track of and what operations are required. As in previous assignments, you are required to submit a small design document that identifies the major issues you tackled in this assignment, and also describes your solutions to these issues.
The document will be used to guide our markers in their evaluation of your solution to the assignment. In the case of poor results in the functional testing combined with a poor design document, we will base our assessment on these components alone. If you cannot describe your own solution clearly, you cannot expect us to reverse engineer the code to a poor and complex solution to the assignment.
Create your design document at the top of the source tree to OS/161 (i.e., in `~/sysarch/asst3-src`), and include it in the Mercurial repository as follows.

```
$ cd ~/sysarch/asst3-src
$ hg add design.txt
```

When you later commit your changes into your repository, your design document will be included in the commit, and later in your submission. Also, please word wrap your design doc if your have not already done so. You can use GNU `fmt` to achieve this if your editor does not.

5.1 Memory Management

This assignment requires you to keep track of physical memory. The current memory management implementation in `dumbvm` never releases memory; your implementation should handle both allocation of and releasing frames.
In the assignment you will need to keep track of whether a frame is used or not. You do not need an extra data structure for this purpose, as you can use the pointer of the collision chain to connect all the free frames in the inverted page table (IPT).
The functions that deal with memory are described in `kern/include/vm.h`. You may assume that only one page will be allocated at a time. Designing a page allocator that can allocate multiple pages at a time is surprisingly tricky. However, make sure that you never allocate memory (through `kmalloc`) that is larger than a page (use `assert()` to enforce this)!
Note that `alloc_kpages()` should return the *virtual* address of the allocated page, i.e., an address in kseg0.
Warning: `alloc_kpages()` may be called before `vm_bootstrap()` was called. This means that your implementation of `alloc_kpages()` must work even *before* your frametable is initialized. You should just call `ram_stealmem()` if the frametable has not been initialized.

5.2 Address Space Management

OS/161 has an address space abstraction, the `struct addrspace`. To enable OS/161 to interact with your VM implementation, you will need to fill in the functions in `kern/vm/addrspace.c`. The semantics of these functions are documented in `kern/include/addrspace.h`.
You may use a fixed-sized stack for each process (e.g., 16 pages).

5.3 Address Translation

The main goal for this assignment is to provide virtual memory translation for user programs. To do this, you will need to implement a TLB refill handler. You will also need to implement a page table. For this assignment, you will implement an *inverted page table* (IPT).

Hashing functions for TLBs tend to be optimized for speed rather than uniform coverage and are therefore very simple. An appropriate choice of the hash is to take the least significant bits of the page number.
The following questions may assist you in designing the contents of your page table:

- What information do you need to store for each page?
- How does the page table get populated?

5.4 Testing and Debugging Your Assignment

To test this assignment, you should run a process that requires more virtual memory than the TLB can map at any one time. You should also ensure that touching memory not in a valid region will raise an exception. The `huge` and `faulter` tests in `testbin/` may be useful. Apart from GDB, you may also find the `trace161` command useful. `trace161` will run the simulator with tracing enabled, for example

```
$ ./trace161 -t t -f outfile kernel
```

will record all TLB accesses in `outfile`. The `trace161` binary is accessible from the same directory as `sys161`, i.e., in `~/sysarch/root`.

5.5 Hints

Have a close look at the `dumbvm` implementation, especially `vm_fault()`. Although it is simple, you should get an idea on how to approach the rest of the assignment.
We suggest you implement the assignment in the following order:

1. Understand how a page table works and its relationship with the TLB.
2. Understand the specification and the supplied code.
3. Work out a basic design for your implementation.
4. Start simple: assume a small address space. This means you can use a straightforward static array as the page table (without collision chain), and can keep your code and data structures simple.
5. Implement the TLB exception handlers in `vm.c` using this simplified page table.

   Note: Your final solution should use an inverted page table!
6. Implement the functions in `kern/vm/addrspace.c` that are required for basic functionality (e.g., `as_create()`, `as_prepare_load()`). Allocating user pages in `as_define_region()` may also simplify your assignment.
7. Test and debug this. Use the debugger! If you really get stuck, submit at least this much of the solution and you should get some marks for it.
8. Understand how the inverted page table works.
9. Decide exactly what data structures you need.
10. Work out the design for the proper solution, using an IPT.
11. Modify your implementation to include the IPT.
12. Write routines in `kern/vm/frametable.c` to manage free frames and copy pages. Also modify functions in `kern/arch/mips/mips/vm.c` to create and delete page table entries and keep the TLB consistent with the page table.
13. Use these routines to finish the functions in `kern/vm/addrspace.c`.
14. Test and debug.

5.6 Assignment Submission

As with the previous assignments, you will again submit a diff of your changes to the original tree. First, both team members need to commit their latest local changes, and push these changesets back to the shared repository, using

```
$ cd ~/sysarch/asst3-src
$ hg commit
$ hg push
```

You might have to manually resolve conflicts.
The one who is going to create the diff and submit it should verify that his/her local repository is up-to-date

```
$ cd ~/sysarch/asst3-src
$ hg pull
$ hg update
```

If everything is all right, the latest revision can be tagged, and the diff can be created:

```
$ cd ~/sysarch/asst3-src
$ hg tag asst-final
$ hg diff -r asst-base -r asst-final > ~/asst3.diff
```

Send the diff (`~/asst3.diff`) of your solutions via email to `os161@ira.uka.de`. Don't forget to include your group name and assignment number in the subject line.
Congratulations! By now you have (probably) completed all the programming assignments of this course!
Note: If for some reason you need to change and re-submit your assignment after you have tagged it "asst-final", you will need to either delete the "asst-final" tag, commit the new changes, re-tag, and re-diff your assignment, or choose a different final tag name and commit the new changes, tag with the new tag, and re-diff with the new tag. To delete a tag, use `hg tag --remove` *tagname*
Even though the generated diff output should represent all the changes you have made to the supplied code, occasionally students do something "ingenious" and generate non representative diff output. We strongly suggest to keep your Mercurial repository intact to allow for recovery of your work if need be.