Computer Science 161: Operating Systems
Section 7: Virtual Memory

CS161 Course Staff
cs161@fas.harvard.edu
http://www.courses.fas.harvard.edu/~cs161/
http://motelab.eecs.harvard.edu/bb/

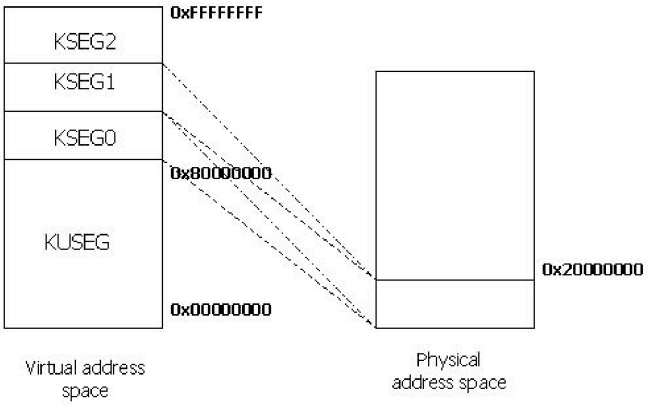
April 7, 2007

1 Introduction

Many students say that the VM assignment was their favorite. The tricky bit of this assignment is that it's hard to test incrementally. This is where design becomes really really important.

2 MIPS r2000/r3000 Review/Overview

2.1 Memory Segments



kuseg `0x00000000-0x7fffffff` TLB-mapped cacheable user space. Your VM system will deal with this memory.

kseg0 `0x80000000-0x9fffffff` direct-mapped cached kernel space. Pointers in this region—kernel pointers—map directly onto the first 512MB of physical memory.

kseg1 `0xa0000000-0xbfffffff` direct-mapped uncached kernel space: Sys/161 doesn't emulate the caching properties of the MIPS r3000, but if it did, devices would be mapped into this space.

kseg2 `0xc0000000-0xffffffff` TLB-mapped cacheable kernel space. If you were to swap kernel memory, you could use this—it undergoes the same translations that `kuseg` undergoes.

3 Your Tasks

- Handle TLB faults.
- Implement paging
  - Per-process data structures (page tables)
  - Global data structures (coremap)
  - Backing store support
  - Page eviction algorithms
- `sbrk()`
- Performance analysis (we'll talk about it later)

4 TLB Handling

- Take a look at `dumbvm` for examples. Vahalia pages 419-421 are at the end of this document.
- Implement two algorithms—these will both be pretty simple.
- We give you `TLB_Random`, `TLB_Write()`, `TLB_Read()`, `TLB_Probe`. `TLB_Random` reserves 8 of the TLB entries; it might be easier to just use `TLB_Write` and `random()`.
- Don't bother with the address-space id; just clear the whole TLB on every context switch. (Why?)

5 Paging

5.1 Introduction

Paging is subtle. You need to manage all the memory mappings for each process, as well as managing the memory of the system. The tricky bits are synchronization.

5.2 Bootstrapping

There's a chicken-and-egg problem. You can't `kmalloc()` until you setup your memory system, but you can't setup your memory system without `kmalloc()`ing stuff. Look at how `ram_stealmem()` works. Remember, all memory is yours to manage.

5.3 Data Structures

- Some sort of mapping between virtual addresses and physical addresses, per process.
- Sort sort of mapping between physical addresses and what's in them (global).

Be sure to fully flesh these data structures out for your design document. Think about the costs and benefits of your page table scheme. Think also about memory consumption.

You want to avoid linked-list data structures. In fact, you want to avoid linear-search for the right table. (This does *not* imply you should write a binary tree...)

5.4 Backing Store

Figure out how to write a page to disk, and how to read it back. Keep in mind that SPL synchronization doesn't work when you start IO operations—they call `thread_sleep()`.

You probably want a pager thread that goes around and writes pages to disk—makes them clean.

It's ok, in fact encouraged, that every page have its own place on disk. You can make your disks reasonably big. We provide bitmap functionality to keep track of your disk usage. If you put your disk in `/tmp` (a drive on the local machine), `sys161` will run faster.

Use `vfs_open()` on `lhd0raw:` and use the vnode you get back for swapping.

5.5 Page Faults

There are three different types of page faults that you may receive, `VM_FAULT_READONLY`, `VM_FAULT_READ`, and `VM_FAULT_WRITE`. The first one indicates that the user is trying to write to a page that is currently read-only. This could be useful if implementing read-only memory, for instance, or for noting that this page of memory is no longer "clean," and is now "dirty." The latter two indicate that you are trying to read or write a page not in memory, respectively.

To process a fault on a page p, you will generally need to do something like:

- Confirm p Exists – To confirm that p exists, you should probably be able to just look in your page table for it.
- Find a Location for p – Finding a place to put p in memory starts to get a little bit more creative. If there is a free piece of memory open, then you may just want to use this. If there isn't, you will have to figure out who you want to kick out. This raises even more interesting questions – is kernel memory pageable? Should it be? Can you make it pageable?

  A partial answer is this – some of it cannot be pageable, while some of it could be made pageable. Your implementation can clarify this point – take a look at `kuseg`, `k0seg`, `k1seg`, and `k2seg` to see how MIPS differentiates between *wired* and *non-wired* memory.

  But wait, how can you figure out if a page of physical memory is free? Well, there is a pretty simple solution – maintain a *core map*. The core map maps pages of physical memory to their associated virtual pages. Thus, you can efficiently figure out who is using a page of memory, and inform them that their page is being evicted. It may have a fancy name, but it is really just a mapping from physical pages to virtual pages.
- Evict the Current Resident – Now that a location for p has been identified, you should copy the old page out of memory and into the backing store, if necessary. Update all page tables, as necessary.
- Load p into Memory – Copy the page into memory and update the page table.
- Update the TLB cache – Update the TLB cache to reflect the page's new location.

5.6 Tricky Stuff: kernel allocations

Your VM system also needs to handle kernel allocations. Kernel allocations are special: if you give a page to the kernel, you can't kick it out (don't implement pageable kernel-memory: it's hard to do). If the kernel needs N contiguous pages, you've got to find N contiguous physical pages.

6 Address Spaces

- `as_create`
- `as_destroy`

- `as_copy` (for fork)
- `as_activate` (context switch)

These are all data structure and synchronization issues; the code itself isn't too bad.

7 Synchronization

Synchronization is by far the hardest thing in this assignment.

- SPL synchronization won't work with I/O. Keep this mind.
- Don't create a lock per page: this consumes too much space. You might want to use a busy bit or play with `thread_sleep()`
- How does locking work when handling a page fault?
- How does locking work when evicting someone else's page?
- What if the page I want is in the middle of being evicted by someone else?
- What do I do in fork if a page that I want to copy is not resident?

Remember, over-syncrhonization is better than under-synchronization.

8 `sbrk()`

Make sure that you implement this so that it's compatible with the `malloc()` that we give you. It's essentially a little bit of book-keeping.

9 Keeping Statistics

Keep track of as many counters you can think of. For example:

- Total number of pages available
- Total number of pages managed by you
- Number of clean pages
- Number of dirty pages
- Number of kernel pages
- etc.

To help you in your debugging, you can write a routine that goes and counts (using your coremap) some of these statistics and notices discrepencies. You can enable/disable this by using a `#define`, *à la* `#SLOW` in `kmalloc()`.

10 What you don't have to do...

- You don't have to do copy-on-write. You can certainly implement things lazily, though.
- Multi-level page tables are not the only way to do it.
- You don't have to page kernel memory.

dressed; therefore, address translation is not required when there is a cache hit. Virtually addressed caches require kernel involvement to address some consistency problems. These issues are described in Section 15.13.

13.3.4 The MIPS R3000

The MIPS R3000 is a RISC system and has been a platform for SVR4 UNIX as well as Digital Equipment Corporation's ULTRIX (a 4.2BSD-based system). It has an unusual MMU architecture [Kane 88] in that there is no hardware support for page tables. The only address translations performed by the hardware are those defined by the on-chip TLB.

This has far-reaching implications on the division of memory management tasks and the interface between the hardware and the kernel. In the Intel *x*86 architecture, for instance, the structure of the TLB entry is opaque to the kernel. The only operations allowed are invalidation of single entries keyed by virtual address or of the entire TLB. In contrast, the MIPS architecture makes the format and contents of the TLB entry public to the kernel and allows operations to read, modify, and load specific entries.

The virtual address space itself is divided into four segments, as shown in Figure 13-11. The *kuseg*, spanning the first two gigabytes, contains the user address space. The other three segments are accessible only in kernel mode. *kseg0* and *kseg1* each map directly to the first 512 megabytes of physical memory, thus requiring no TLB mapping. Of these, *kseg0* uses the data/instruction caches, but *kseg1* does not. The top gigabyte is devoted to *kseg2*, which is the mapped, cacheable kernel segment. Addresses in *kseg2* can be mapped to any physical memory location.

Figure 13-12 describes the MMU registers and the format of the TLB entry. The MIPS page size is fixed at 4 kilobytes; thus the virtual address is divided into a 20-bit virtual page number and a 12-bit offset. The TLB contains 64 entries, and each entry is 64 bits in size. The **entryhi** and **entrylo** registers have the same format as the high and low 32 bits of the TLB entry, respectively, and are used to read and write a TLB entry. The `VPN` (virtual page number) and `PFN` (physical frame number) fields allow translation of virtual to physical page numbers. The `PID` field acts as a tag, associating each TLB entry with a process. This `PID`, which is 6 bits in size, can take the values 0

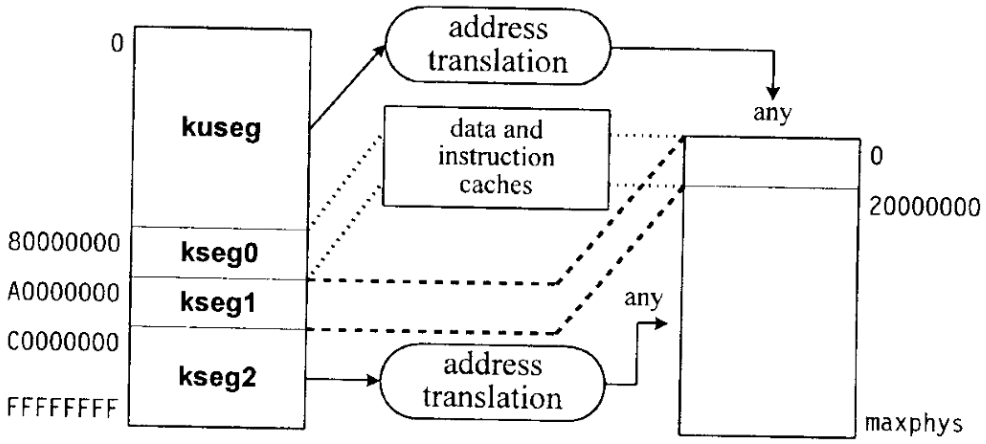


Figure 13-11. MIPS R3000 virtual address space.

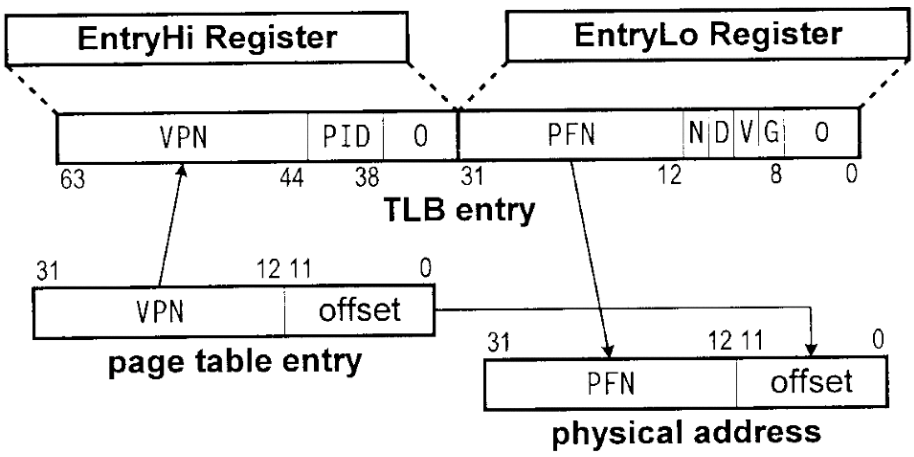


Figure 13-12. MIPS R3000 address translation.

through 63, and is not the same as the traditional process ID. Each process that may have active TLB entries will be assigned a *tlbpid* between 0 and 63. The kernel sets the PID field in the **entryhi** register to the *tlbpid* of the current process. The hardware compares it to the corresponding field in the TLB entries, and rejects translations that do not match. This allows the TLB to contain entries for the same virtual page number belonging to different processes without conflict.

The N *(no-cache)* bit, if set, says that the page should not go through the data or instruction caches. The G *(global)* bit specifies that the PID should be ignored for this page. If the V *(valid)* bit is clear, the entry is invalid, and if the D *(dirty)* bit is clear, the entry is write-protected. Note that there is neither a *referenced* bit nor a *modified* bit.

In translating *kuseg* or *kseg2* addresses, the virtual page number is compared with all TLB entries simultaneously. If a match is found and the G bit is clear, the PID of the entry is compared with the current *tlbpid*, stored in the **entryhi** register. If they are equal (or if the G bit is set) and the V bit is set, the PFN field yields the valid physical page number. If not, a **TLBmiss** exception is raised. For write (store) operations, the D bit must be set, or else a **TLBmod** exception will be raised.

Since the hardware provides no further facilities (such as page table support), these exceptions must be handled by the kernel. The kernel will look at its own mappings, and either locate a valid translation or send a signal to the process. In the former case, it must load a valid TLB entry and restart the faulting instruction. The hardware imposes no requirements on whether the kernel mappings should be page table-based and what the page table entries should look like. In practice, however, UNIX implementations on MIPS use page tables so as to retain the basic memory management design. The format of the **entrylo** register is the natural form of the PTEs, and the eight low-order bits, which are unused by hardware, may be used by the kernel in any way.

The lack of *referenced* and *modified* bits places further demands on the kernel. The kernel must know which pages are modified, since they must be saved before reuse. This is achieved by write-protecting all clean pages (clearing the D bit in their TLBs), so as to force a **TLBmod** exception on the first write to them. The exception handler can then set the D bit in the TLB and set ap-

propriate bits in the software PTE to mark the page as dirty. Reference information must also be collected indirectly, as shown in Section 13.5.3.

This architecture leads to a larger number of page faults, since every TLB miss must be handled by the software. The need to track page modifications and references causes even more page faults. This is offset by the speed gained by a simpler memory architecture, which allows very fast address translation when there is a TLB cache hit. Further, the faster CPU speed helps keep down the cost of the page fault handling. Finally, the unmapped region *kseg0* is used to store the static text and data of the kernel. This increases the speed of execution of kernel code, since address translations are not required. It also reduces contention on the TLB, which is needed only for user addresses and for some dynamically allocated kernel data structures.

13.4 4.3BSD — A Case Study

So far we have described the basic concepts of demand paging, and how hardware characteristics can influence the design. To understand the issues involved more clearly, we use 4.3BSD memory management as a case study. The first UNIX system to support virtual memory was 3BSD. Its memory architecture evolved incrementally over the subsequent releases. 4.3BSD was the last Berkeley release based on this memory model. 4.4BSD adopted a new memory architecture based on Mach; this is described in Section 15.8. [Leff 89] provides a more complete treatment of 4.3BSD memory management. In this chapter, we summarize its important features, evaluate its strengths and drawbacks, and develop the motivation for the more sophisticated approaches described in the following chapters.

Although the target platform for the BSD releases was the VAX-11, it has been successfully ported to several other platforms. The hardware characteristics impact many kernel algorithms, in particular the lower-level functions that manipulate page tables and the translation buffer. Porting BSD memory management has not been easy, since the hardware dependencies permeate through all parts of the system. *As a result, several BSD-based implementations emulate the VAX memory architecture in software, including its address space layout and its page table entry format.* We avoid a detailed description of the VAX memory architecture, since the machine is now obsolete. Instead, we describe some of its important features as part of the BSD description.

4.3BSD uses a small number of fundamental data structures—the *core map* describes physical memory, the *page tables* describe virtual memory, and the *disk maps* describe the swap areas. There are also resource maps to manage allocation of resources such as page tables and swap space. Finally, some important information is stored in the `proc` structure and `u` area of each process.

13.4.1 Physical Memory

Physical memory can be viewed as a linear array of bytes ranging from 0 to n, where n is the total amount of memory on the system. It is logically divided into pages, with the page size dependent on the machine architecture. This memory can be divided into three sections, as shown in Figure 13-13. At the low end is the *nonpaged pool*, which contains the kernel code and the portion of the kernel data that can be allocated either statically or at boot time. Since a kernel page fault can block a process in the kernel at an inconvenient point, most UNIX implementations require all kernel pages to