CS 61C Lecture Notes Week 11

Topic: Virtual memory

Reading: Patterson & Hennessy, Sections 7.4–7.9

Until now we've been talking as if one program had the computer to itself; in particular, we've talked as if one program had all of memory to itself. In a real system, several processes share memory, along with the operating system kernel.

Hardware relocation

A typical early solution to this problem was that each process had one chunk of memory, so the overall memory picture looked like this:

```
+--------+
|  free  |
|        |
+--------+
|process |
|   C    |
|        |
+--------+
|process |
|   B    |
+--------+
|        |
|process |
|   A    |
|        |
|        |
|        |
+--------+
|        |
|   OS   |
| kernel |
|        |
+--------+
```

In this approach, the computer has two special registers, the *relocation* register and the *protection* register. The relocation register contains the memory address of the beginning of the currently running process; the protection register contains the size of the process (how much memory it requires).

Whenever a process makes a memory reference, the processor takes two extra steps. First, it compares the requested address with the protection register. If the requested address is not less than the value in that register, then the process is trying to use memory it doesn't have, and so an exception is generated and the operating system stops that process. Second, the value in the relocation register is added to the address that the program requested, to find the real address of the desired data.

This system allows several programs to coexist in memory, but it isn't ideal. As different-sized programs come and go, we may find that there is a slice of free memory in the middle that's too small to hold any program. This problem is called *fragmentation.* Also, if a program is very large, it probably doesn't need the entire program text and data in memory at the same time; we can fit more processes if a program can be partly in memory and partly saved on disk.

Virtual memory goals

We would like the operating system to provide the following features to user programs:

• The illusion of continuous memory (the entire program in memory, occupying consecutive addresses).

• The illusion of unlimited memory (so that we can run very large programs on a small computer, and several programs at once).

• The sharing of memory among processes, but in a way that protects each process from the others.

We accomplish these goals in part by using disk space to hold parts of a program that are not in main memory at a given moment. In a sense, the relationship between memory and disk in a virtual memory system is analogous to the relationship between the cache and the main memory: We can simulate the *size* of the larger memory with almost the *speed* of the slower memory. (But the analogy is only approximate, because in both cases the user process thinks it's looking at main memory, which is the large, slow device in the cache system but the small, fast device in the virtual memory system.)

A program's "virtual memory" means the collection of addresses that the program can use — in other words, the simulated large, contiguous memory that the program thinks it occupies.

Note to Macintosh users: Apple, which likes to invent aw-shucks terminology to make it harder to understand what their machines are really doing, talk as though "virtual memory" were an extension to physical memory; Mac users say things like "my Mac has 8 Meg of physical memory plus 20 more Meg of virtual memory." This is completely wrong. What they mean is, "my Mac allows programs 28 Meg of virtual memory, using 8 Meg of physical memory plus enough disk space to hold the other 20 Meg." The program sees (virtual) memory as one single, uniform address space; the part of the program that's in use at any moment must be in physical memory in order to run.

Analogy with cache

When we use a cache as a small interface to the larger main memory, several design decisions must be made. Each of these corresponds to an issue in virtual memory organization.

Cache size: A larger cache size leads to a better hit rate. Similarly, the more memory you buy, the more large programs you can run without needing to transfer data between memory and disk. Main memory is cheaper than cache memory, though, so a memory of many megabytes is common.

Block size: A large cache slot increases the hit rate, because of locality of reference, but also increases the miss penalty, because more data must be transferred on a miss. The big difference between caching and VM is that when a disk is involved, the cost for one large transfer is much less than the cost for two smaller transfers, because of seek time and rotational latency. (In transferring between cache and memory, the time is proportional to the block size, so two smaller transfers don't take longer than one large one.) This is why the block size for caching is about 16–64 bytes, but the block size for VM is 1–4 kilobytes.

Associativity: An associative cache costs more to build than a direct mapped cache, but leads to a slight increase in the hit rate. In the case of VM, the added cost is much lower than for caching (we'll see why as we look at the details), and more importantly, the miss penalty is really bad (because it requires a disk operation). So a VM system is always fully associative, even though most caches today are direct mapped.

Write-through vs. write-back: A write-through cache makes stores more expensive but loads cheaper. Also, a write-through cache will do *more* actual writes to memory than a write-back cache, in cases where two stores to the same cache block happen before the block needs to be written back to memory. In the case of VM, since the blocks are much larger, it's very likely that many stores will be done in the same block in quick succession. Also, the write cost is very high (because it's a disk operation). Therefore, all virtual memory systems use write-back, even though for caches the tradeoff can go either way.

Replacement policy: If the cache is full and we need a new slot, which old slot should we replace? For direct mapped caches this is a trivial question, because there is only one possibility. For associative caches, random choice is generally used, even though it slightly worsens the hit rate, because the miss penalty is not very severe and we don't want to increase the hit cost by bookkeeping about which entries have been used recently. In the case of VM, the miss penalty *is* severe, and so we try harder to approximate a Least Recently Used (LRU) replacement policy.

The page table

Here's the idea of virtual memory organization: We divide physical memory into *pages* of, let's say, 4K bytes. (So a typical 64Mb home computer would have 16,000 pages available.) A program thinks it has a large memory available to it; on the MIPS, user processes think they can address 2 billion bytes. Most programs don't use nearly that much of the address space, but they do use both the top and the bottom ends, because of the stack/heap allocation system. So there's a "hole" in the middle of each program's address space. (In more complicated systems there may be several holes.)

We divide the program's 2-gigabyte address space, also, into pages of 4K bytes. Some pages are in use; others aren't. Each page that's in use may or may not be in physical memory at this moment; if so, we keep track of which physical page corresponds to which virtual page. (For the virtual pages that are in use but aren't in memory right now, we keep track of where on the disk that page is kept.)

The virtual address space of 2Gb corresponds to about one million pages. As a first version, simple but somewhat unrealistic, let's say we have in memory a *page table* with one million entries. Each entry consists of a Valid bit (as in a cache entry) and, if the valid bit is on, a physical page number. A special processor register contains the address of the beginning of the page table.

Every time a program asks to load or store some (virtual) memory address, the machine figures out the virtual page number of that address. The page number is all but the last 14 bits of the address, if a page is 4K (2^{14}) bytes. The machine uses that virtual page number as an index into the page table array, and reads the entry for that page. If the valid bit is set, the machine computes a physical address by replacing the virtual page number from the requested address with the *physical* page number that it found in the page table. The machine can then load or store the requested information.

If the valid bit is *off*, the hardware generates an exception and the operating system has to figure out what to do. This can mean one of two things. The address may be a valid address, but its page is not in memory right now. (We say that the page is "paged out" or "swapped out.") This case is called a *page fault*. In this case, the operating system must find (or create) a vacant physical page, load the data from disk into that page, update the process's page table, and then allow the program to continue. Alternatively, the address may be invalid. (That is, the program may be dereferencing a pointer whose value is not an address that the program is supposed to be using.) This case is called, in Unix, a *segmentation violation*, and the operating system will stop the program and give the user an error message.

You should see that this system is similar in spirit to the protection/relocation system described earlier. It implements a function that maps virtual addresses to physical addresses, along with a validity check on the virtual address. (It does introduce the new case of a page fault for a valid address.) The main difference is that the function implemented by the simpler relocation scheme is sort of like a direct mapped cache, in that each virtual page of a process must be in one particular physical page of memory. The page table is more like an associative cache, in that any virtual page can be in any physical page.

Making paging practical

In principle this is almost everything there is to say about paging hardware. In practice, though, the system as described so far would be very slow, because every memory reference that the program makes will require *two* actual memory references: one for the page table entry, and one for the request itself. This is unacceptable.

The solution is yet another application of the cache idea. The machine has a *Translation Lookaside Buffer*

(TLB), which is basically a special-purpose cache for page table entries. Because of locality of reference, once the program has used a particular page, it's likely to make many more references to the same page. So we keep that page's entry inside the processor, in the TLB. Only when the program refers to a page that hasn't been used before (or at least not recently) do we have to look at the page table in memory.

So now there are four possible outcomes when a program makes a memory reference:

1. *TLB hit.* The desired page is in memory, and has been used recently, and so the processor knows which physical page to use all by itself.

2. *TLB miss.* The desired page is in memory, but has not been used recently, so the processor doesn't remember where in physical memory it is. In this case a reference to the page table is needed. (In some systems this case is handled entirely by the hardware; in other systems the hardware signals a TLB Miss exception and the operating system checks the page table.)

3. *Page fault.* Not only isn't this virtual page number in the TLB, it's not in the page table either. (More precisely, its page table entry is marked invalid.) In this case, the operating system must find out where this virtual page is located on the disk, and bring it into physical memory.

4. *Segmentation violation.* Not only isn't this number in the page table, but it shouldn't be, because it's not a page that this process has allocated in its virtual address space. The operating system should signal an error to the user.

In the MIPS architecture, as befits a RISC design, the goal was to minimize the complexity of the hardware. Therefore, the hardware doesn't know anything about a page table. It has a TLB only. Cases 2, 3, and 4 above are all treated as TLB misses by the hardware; the operating system software has to distinguish among them.

In addition to hardware simplicity, another advantage of the MIPS approach is that the system is not committed to a particular page table structure, as it would be if the hardware knew about the page table. Another impracticality of the simplified paging mechanism described above is that a million-word page table would itself take up several thousand pages! Unix typically gives each process two small page tables, one for the area near the bottom of virtual memory (the text, global data, and heap) and one for the area near the top (the stack). It doesn't waste memory on vacant page table slots for the big hole in the middle. Some architectures have this two-page-table idea built in, but that seems kludgy. It's better to be able to make that choice in software, in case we want to run some other operating system that organizes memory differently.

One possible approach, used in some systems, is to page the page table. That is, the page table is essentially an array of data in the kernel's memory; kernel pages can be paged just as user pages can. You have to be careful not to get in an infinite regress; the pages containing the page table had better not be paged out! There are other approaches, as you'll learn in 162.

Write protection

In a multi-user system there may be 20 Emacs processes running at once. We'd like to have only one copy of the Emacs text in memory, rather than 20 identical copies. But to make this possible while still protecting each user from errors in another process, we must ensure that the text part of the process address space is not modified while the program is running.

The virtual memory hardware can help with this. In the page table (and in the TLB), each entry has a Write-Enable bit. When the process tries to store into an address, if the virtual page number is found in the TLB but the entry is not write enabled, the hardware signals an illegal write exception and the operating system stops the process.

Replacement policy

There are actually two replacement issues. One is that the TLB can fill up, and in order to get a new page table entry into the TLB we have to eliminate some existing entry. The TLB is essentially a special-purpose cache for main memory, so the replacement issue for the TLB is relatively unimportant and we can use random replacement. (Actually there is one small but important exception. Since the addition of new entries to the TLB is handled by the operating system on the MIPS, we'd better make sure that the TLB management procedures in the operating system are always accessible! So a few TLB slots are considered permanent and are used for these crucial pieces of the operating system.)

The other issue is about page replacement in main memory. Here the miss penalty is very severe, so we want to try hard to pick a really old, not recently used, page to remove from memory. The VM hardware can help with this. Most systems include a Referenced bit in each page table (or TLB) entry; the hardware turns the bit on whenever any memory in that page is used. Every so often the operating system turns the Referenced bits off, perhaps recording old values of the bits somewhere. A page whose Referenced bit is on has been used recently and therefore shouldn't be paged out.

When a page *is* paged out, do we have to write that page onto the disk? The answer may be no, if nothing in the page has been modified since the page was loaded from the disk. Therefore, the hardware can help save time by providing a Modified bit (also called the Dirty bit) which is turned on whenever the machine writes into the corresponding page of memory. If a page is going to be paged out, and the Dirty bit is *off*, then there is no need to write the page to disk.

The Valid and Write-Enable bits are set by the software to give information to the hardware. The Referenced and Dirty bits are the opposite; they're set by the hardware to give information to the software.

If a machine doesn't have Referenced and Dirty bits, the operating system can fake it, at some cost in speed. To simulate the Referenced bit, every so often the operating system turns off the Valid bit for all user pages. The result is a flurry of page faults, one for each referenced user page. The operating system can note that the page has been referenced and turn the Valid bit back on, so only the first reference to each page will cause a fault.

To simulate the Dirty bit, the operating system can keep Write-Enable off for every page brought into memory. (It keeps the information about which pages should really be writeable in some other data structure.) The first time a page is written, it will cause an exception; the system can then maintain a Dirty bit in software. The system also turns Write-Enable on, if the page should be writeable, so that only the first write into any given page causes an exception.

MIPS memory map

The 4 Gigabyte virtual address space on the MIPS is divided into sections. The first half (0-2 Gb) is available to user programs. The second half is not; if the user mode bit is set, any reference to the top half of virtual address space causes an exception. (The operating system kernel *can* make reference to user addresses, though.)

Of the system's half of virtual memory, half (2-3 Gb) is paged using the TLB just as user addresses are paged. This space is used for the kernel's ordinary code and data. The remaining addresses are divided in half again. Addresses in the range 3-3.5 Gb are unpaged but cached; addresses in the range 3.5-4 Gb are unpaged and not cached. "Unpaged" means that these addresses always map directly into physical addresses at the bottom of memory (0-0.5 Gb).

The kernel uses unpaged, cached addresses for its startup code, which must be able to run before any TLB entries are set up. It uses unpaged, uncached addresses for device registers. (That's why the ones we use are in the `0xffff....` range.) Device registers must not be cached, because they are volatile. In other words, the contents of a device register can be different on two successive reads, even though the processor has not stored a new value in between.