CS162 Discussion #11: All Your Page Are Belong To Us

Announcements

- Project 2 due TONIGHT @11:59pm (`submit proj2-code`)
    - Final design doc and evals due tomorrow @11:59pm (`submit proj2-final-design`)
- Project 3 starting: initial design due 11/17 @5:00pm, please give Angela a hardcopy as usual

Why you might want a virtual address

As you might remember from the beginning of the semester, running multiple processes means we have to prevent processes from messing with each other's memory. We do this by giving each process a virtual address space, which to the process looks like a chunk of contiguous memory that it has all to itself. While this looks nice and clean from the program's point of view, all of this memory could be broken up into various pieces scattered across physical memory. We use address translation to translate the virtual addresses the program uses into physical addresses that they are associated with on physical memory. This allows each program to refer to memory as if it is the only program running; not only does this provide protection between programs and make things easier for programmers, it also makes programs more modular because physical addresses aren't hardcoded into the program.

One method of memory management is to establish segments contiguous memory that start at a certain base location in physical memory. Virtual addresses in a segment start at 0 and can go up to a given address limit. To find the physical address, we add the base, specified by a segment base pointer, to the virtual address. A program could have multiple segments (i.e., for code, data, stack, etc.), each in a different part of memory; these segments are specified by segment ID, base, and limit, stored as entries in a translation table.

I'll page you ;)

While the simple segmentation scheme is simple and easy to understand, one of the disadvantages to simple segmentation (where each segment is a chunk of contiguous memory) is that variable segment sizes can cause memory to become fragmented. We can fix this problem by making sure that memory is always allocated in fixed-size chunks, which we call pages. These pages can then be assigned to different segments. Each page table entry contains a bunch of relevant information, including the physical page it's mapped to and permissions (read, write, valid, etc.). These pages are stored in a process' page table, which can be found by following the page table pointer. In a simple page table lookup, a process can locate the desired page based on virtual page number, then use the information stored in the page table entry to determine what physical page it should go to.

Not only that, we can also create segments out of pages. In this case, each virtual address will have a segment number in addition to a page number and an offset, and the physical address will be looked up using a multi-level page table. This page table will have one level indexed by segment number, which will contain another pointer to a page table indexed by virtual page number, and this entry will have the physical page number we're looking for. (We can also do the same thing just using two page indexes, without associating them with segment numbers.)

This lets us conveniently group pages into sets, which is particularly useful in sparse memory spaces where there are groups of pages that aren't currently needed. Note that there may be permission bits at every level of the table, so we can check at every step of the translation process whether or not a page is valid, writeable, etc. Trying to access pages that aren't currently in memory will cause a page fault, which occurs if a page table entry is marked as being invalid. The fault handler will then take care of bringing the page in from disk – we call this system demand paging. You'll see more of this during lecture in the coming week.

Cash In on Caching!

All that looking up in the page table seems like a lot of trouble, especially if you're going to be accessing the same virtual address a lot. It would be much better if we could remember where a virtual address took us the first time so would wouldn't have to go through that again – and that's what a cache is for. A cache keeps entries that are fast to look up, so we want to try to keep things we know will be looked up again in there to avoid taking the time to do the long address translation process all over again. The cache that's specially used for address translation is called the TLB.

Before it does a full-blown page table lookup, a process will check the TLB first to see if there's an entry already there that it can use. If not, this is a cache miss, and we don't have any choice but to go do the entire lookup. There are four different kinds of cache misses:

- *Compulsory.* The block hasn't been accessed before, so there's no way it could be in the cache.
- *Capacity.* There isn't enough room in the cache to keep all the blocks the process is trying to use.
- *Conflict.* Multiple memory locations are being mapped to the same place in the cache, so they kick each other out each time they're stored.
- *Coherence.* Memory has been updated so the entry isn't correct anymore.

In general, we do cache lookups by using an index to look for entries in the cache that might be what we're looking for. A tag then identifies the exact item we want. If nothing matches the tag, it's a cache miss. The specific lookup, however, depends on the type of cache we have. Here are some different kinds of caches:

- *Direct Mapped*. Each item is associated with one specific place in the cache using tag, index, and byte select.
- *N-way Set Associative*. Each index identifies a set of N blocks. The tag is compared between the N blocks to find the correct one, and then a byte select is used.
- *Fully Associative*. Every item can go in any block in the cache. The tag is compared between all blocks, then a byte select is used. There is no cache index here.

A TLB miss, which happens if the translation the process is looking for isn't currently stored in the TLB, will force us to look through the page table to find the page we want. Remember that pages don't necessarily have to be memory, though! Looking through the page table at this point could still cause a page fault, just like before. When the page has been found and loaded in memory, the TLB is updated with the new entry.

It's important to be careful how entries are being stored in the TLB, because accesses that cause a lot of conflicts can lead to thrashing. This means that you can potentially continuously create lots of cache misses if memory accesses are frequently kicking entries out of the TLB that the process will want to use again soon.