CMPSCI 377 Operating Systems **Fall 2009**

Lecture 15

Lecturer: Emery Berger *Scribe: Bruno Silva,Jim Partan*

15.1 Paging

In recent lectures, we have been discussing virtual memory. The valid addresses in a process' virtual address space correspond to actual data or code somewhere in the system, either in physical memory or on the disk. Since physical memory is fast and is a limited resource, we use the physical memory as a cache for the disk (another way of saying this is that the physical memory is "backed by" the disk, just as the L_1 cache is "backed by" the L_2 cache).

Just as with any cache, we need to specify our policies for when to read a page into physical memory, when to *evict* a page from physical memory, and when to write a page from physical memory back to the disk.

15.1.1 Reading Pages into Physical Memory

For reading, most operating systems use *demand paging*. This means that pages are only read from the disk into physical memory when they are needed. In the page table, there is a *resident* status bit, which says whether or not a valid page resides in physical memory. If the MMU tries to get a physical page number for a valid page which is not resident in physical memory, it issues a *pagefault* to the operating system. The OS then loads that page from disk, and then returns to the MMU to finish the translation.[1]

In addition, many operating systems make some use of *pre-fetching*, which is called pre-paging when used for pages. The OS guesses which page will be needed next, and begins loading it in the background, to avoid future pagefaults. This depends heavily on locality of accesses, namely that future accesses will be near recent accesses, and this often true.

15.1.2 Evicting and Writing Pages from Physical Memory

When do we write a page from physical memory back to the disk?

In general, caches have two broad types of writing policies. One approach is a *write-through* cache. In this case, when a value in the cache is written, it is immediately written to the backing store as well (in this case, the disk). The cache and backing store are always synchronized in this case, but this can be very slow. The other main approach is a *write-back* cache. In this case, the backing store and the cache are sometimes out of sync, but this approach is much faster. This is what is used with paging, for obvious speed reasons.

When a page is loaded from the disk to physical memory, it is initially *clean*, i.e. the copy in physical memory matches the copy on disk. If the copy in memory is ever changed, then its page-table entry is marked *dirty*, and it will need to be written back to the disk later.

[1] In contrast, if the MMU issues a pagefault for an *invalid* virtual address, then the OS will issue a *segfault* to the process, which usually terminates the process. Segfault is an old term for "segmentation fault", or "segmentation violation", leading to the name of the corresponding Unix signal: SIGSEGV.

When physical memory fills up, and a non-resident page is requested, then the OS needs to select a page to *evict*, to make room for the new page. The evicted page is called the *victim*, and is saved to the so-called "swap" space.[2] The swap space is a separate region of the disk from the file system, and the size of the swap space limits the total virtual address space of all programs put together (though in practice, there is a lot of memory shared between processes, for instance shared libraries).

There are a variety different strategies for choosing which page to evict, with tradeoffs for each strategy. These strategies will be discussed later. One thing to note is that evicting a clean page is fast, since it doesn't need to be written back to the disk. A second note is that to speed up the process of evicting pages, the OS can write dirty pages back to disk as a background task. In this way, more pages will be clean and can therefore be evicted a lot more quickly, when it is time to do so.

15.1.3 `mmap()`

In the last section, we discussed how the swap space on disk is a backing store for physical memory, and that the swap space is an area on disk distinct from the filesystem. Since the disk's filesystem already stores files, it would be redundant to store files both on the filesystem and also in the swap space. It would be very convenient if we could use the filesystem itself as an additional backing store for different parts of physical memory. Doing this is very common, and uses the `mmap()` function.

The `mmap()` function maps a file into virtual memory as a big array. For instance, a word-processing document file could be mapped into memory, making it much easier for the programmer to support skipping forward and back in the file, editing it, and saving it back to disk, compared with if the programmer had to rely exclusively on file I/O stream functions. One of the main reasons we need 64-bit architectures is to be able to `mmap()` extremely large files, such as databases.

15.1.4 A Day in the Life of a page

Suppose your process starts up, and allocates some memory with `malloc()`. The allocator will then give part of a memory page to your process. The OS then updates the corresponding page-table entry (PTE), marking the virtual page as *valid*. If you then try to modify part of that page, only then will the OS actually allocate a physical page, and so the PTE will now be marked as *resident*. In addition, since the memory has been modified, the PTE will be marked *dirty*.

If the OS ever needs to evict this page, then, since it is dirty, the OS has to copy that page to disk (i.e. it *swaps* it, or performs *paging*). The PTE for the page is still marked valid but is now marked non-resident.

And, if we ever touch that page again (i.e. if we try to read or write it), the OS may have to evict some other page in order to bring our page back into physical memory.

One obvious implication of this is that pagefaults are slow to resolve, since disk accesses are performed. Therefore one possible optimization is for the OS to write dirty pages to disk constantly as an idle background task. Then, when it is time to evict those pages, they will be clean, and the OS won't have to write them to disk. This makes the eviction process much faster.

[2]The term "swap space" is an old term from batch-processing days, when entire processes were swapped back and forth between the physical memory and the backing store. Generally, "paging" is a more accurate and more modern term than "swapping" is.