CMPSCI 377 Operating Systems **Fall 2013**

Lecture 14: October 21

Lecturer: Prashant Shenoy *Scribe: Armand Halbert*

14.1 Demand Paged Virtual Memory

To this point, we have assumed that the virtual address space of a process fit in memory and that it was all in memory. Demand paging is way of using virtual memory to give processes the illusion of infinite available memory. In a system that uses demand paging, the operating system stores copies of pages in both disk and memory, and only copies a disk page into physical memory when an attempt is made to access it (i.e., if a page fault occurs). In this way, the disk is treated like a much larger, but slower type of main memory. The memory is then used as a cache for the disk. It follows that a process begins execution with none of its pages in physical memory, and many page faults will occur until most of a process's working set of pages is loaded into physical memory. This is an example of lazy loading techniques.

The page table indicates if the page is on disk or memory using a valid bit. Once a page is brought from disk into memory, the OS updates the page table and the valid bit. For efficiency reasons, memory accesses must reference pages that are in memory the vast majority of time. In practice this is mostly true because of *Locality* - the *working set* size of a process must fit in memory. The 90/10 rule states that, 90% of the time, the program is using only 10% of its code and data.

14.1.1 Advantages

- Demand paging does not load the pages that are never accessed, saving memory for other programs and increases the degree of multiprogramming.
- There is less loading latency at the program startup.
- There is less initial disk overhead because of fewer page reads.
- It does not need extra hardware support than what paging needs, since existing page fault mechanisms can be used to load pages from disk.
- Pages will be shared by multiple programs until they are modified by one of them, so a technique called copy on write can be used to save more resources.
- Ability to run large programs on the machine, even though it does not have sufficient memory to load the full program into RAM.

14.1.2 Disadvantages

- Individual programs face extra latency when they access a page for the first time.
- Page access times become variable and hard to predict because they may require loading the page from disk.
- Memory management with page replacement algorithms becomes slightly more complex.

14.1.3 When to load a page?

There are several mechanisms that can be used to decide when to load a page:

At process start: This is the technique assumed in the previous lectures, where all of the pages for a process were allocated and loaded when the process began. However, this places a limit on the maximum size of each process since all currently running processes must fit in memory at once.

Overlays: are a technique used in early systems where the application developer used code to indicate when to load and remove certain segments of code. This allows a system to support processes that require a virtual address space larger than the physical RAM, but it can be error-prone and requires very careful programming to be done correctly.

Request Paging: allows a process to tell the OS before it needs a page and then tell it again when it is complete. These hints can be used by the OS to reduce page access times by preemptively loading pages, but this again requires careful programming to be done effectively.

Demand Paging: always loads a page the first time it is referenced. Pages may be removed from memory to make room for new pages. This has the benefit of lowering memory usage since only actively used pages are kept in memory, but can incur high page loading times since pages are not loaded preemptively.

Pre-paging: attempts to guess in advance when a process will use each page and will preemptively load them into memory so they are available when first accessed. This improves performance since pages are loaded into memory while other tasks are running on the CPU, but can be difficult to do correctly since the OS needs a good heuristic for deciding when to load pages—especially hard to do when applications have many branches.

14.1.4 Implementation of Demand Paging

Demand paging is implemented by keeping a valid bit in the page table that indicates whether the corresponding page is in memory or disk. The valid bit being 0 indicates that the page is not in memory (i.e. either it is on disk or it is an invalid address). When a page is requested and it does not yet have the valid bit set to 1, then the OS first must verify that the address is valid. Then the OS performs the following steps -

1. selects a page to replace (page replacement algorithm)
2. invalidates the old page in the page table
3. starts loading new page into memory from disk
4. context switches to another process while I/O is being done
5. gets interrupt that page is loaded in memory
6. updates the page table entry
7. context switch back to faulting process

14.1.5 Swap Space

In the proceeding algorithm, it was possible for an old page to be invalidated if there was insufficient space in memory. If a page containing code is removed from memory, we can simply remove it since it is unchanged

and can be reloaded from disk. If the page containing data is removed from memory, we need to save the data (possibly changed from the last time) so that it can be reloaded of the process it belongs to refers to it again. In order to save this page, the OS must "swap out" the page to disk. The area on disk used for storing these memory pages is called swap space. Note that it is better in terms of performance to swap out pages that haven't been written to (i.e. are *clean* rather than *dirty*). At any given time, a page of virtual memory might exist in one or more of - the file system (on disk), physical memory, or the swap space. To support this, the page table must have indicator bits that can be used to track a pages location.

14.1.6 Performance of Demand Paging

In the worst case, a process could access a new page with each instruction, resulting in a huge number of disk accesses. Since accessing disk is orders of magnitude slower than memory, this would render the system unusable. Fortunately, most processes exhibit *locality of reference*.

Temporal locality: If a process accesses an item in memory, it will tend to reference the same item again soon.

Spatial locality: If a process accesses an item in memory, it will tend to reference an adjacent item soon.

Both of these characteristics help prevent demand paging systems from suffering severe performance penalties: processes will tend to re-use the memory pages that have recently been loaded, and many memory accesses will happen near each other, e.g. within a single page.

14.1.7 Transparent Page Faults

The OS requires some special hardware support in order to make page faults transparent to running processes. When a page fault occurs, the current CPU state and the details of the instruction that caused the fault need to be saved. This can be particularly complicated in systems with complex instruction sets (CISC), as a single instruction may involve multiple stages. If part of the instruction has already been completed, the hardware must support rolling back the instruction to eliminate any side effects caused by it being started. For instructions that copy large regions of potentially overlapping memory, it can be necessary to verify that all relevant pages are loaded prior to starting the transfer–otherwise a partial transfer could overwrite data with no way to roll back.

14.2 Page Replacement Algorithms

When a page fault occurs, there may or may not be any free space in memory to hold the new page. If there is space, then a free frame is simply used for the new data. However, if no free frames are available, then an existing page must be "evicted" from memory to clear up space. Pages can be evicted globally or locally. Global page eviction means that if a process needs to load a new page into memory, then the page replacement algorithm can evict a page from another process if needed. Local page eviction means that only the pages of the process that needs a new page can be evicted. We will generally mean local page eviction when referring to page replacement. There are several page replacement algorithms which can be used to determine which page to evict.

MIN: This is the provably optimal page replacement algorithm. When a page needs to be swapped in, the operating system swaps out the page whose next use will occur farthest in the future. This algorithm cannot be implemented in the general purpose operating system because it is impossible to compute reliably how long it will be before a page is going to be used, except when all software that will run on a system is either

known beforehand and is amenable to the static analysis of its memory reference patterns, or only a class of applications allowing run-time analysis is allowed. Despite this limitation, algorithms exist that can offer near-optimal performance—the operating system keeps track of all pages referenced by the program, and it uses that data to decide which pages to swap in and out on subsequent runs. This algorithm can offer near-optimal performance, but not on the first run of a program, and only if the program's memory reference pattern is relatively consistent each time it runs.

FIFO: The first-in, first-out (FIFO) page replacement algorithm is a low-overhead algorithm that requires very little book-keeping on the part of the operating system. The idea is obvious from the name - the operating system keeps track of all the pages in memory in a queue, with the most recent arrival at the back, and the earliest arrival in front. When a page needs to be replaced, the page at the front of the queue (the oldest page) is selected. While FIFO is cheap and intuitive, it performs poorly in practical application because the frequency that a page is used does not impact how long it stays in memory. Thus, it is rarely used in its unmodified form.

LRU: The least recently used page (LRU) replacement algorithm keeps track of page usage over a short period of time. When a page must be evicted, it selects the one which was used the least recently (e.g. the oldest access time). LRU works on the idea that pages that have been most heavily used in the past few instructions are most likely to be used heavily in the next few instructions too. While LRU can provide near-optimal performance in theory, it is rather expensive to implement in practice. There are a few implementation methods for this algorithm that try to reduce the cost yet keep as much of the performance as possible.

Random: Random replacement algorithm simply replaces a random page in memory. This eliminates the overhead cost of tracking page references. Despite its simplicity, it usually fares better than FIFO, and for looping memory references it is better than LRU, although generally LRU performs better in practice.