CMPSCI 377 Operating Systems **Fall 2008**

Lecture 17: November 4

Lecturer: Prashant Shenoy *Scribe: Shashi Singh*

17.1 Demand Paging

Demand paging is an application of virtual memory. In a system that uses demand paging, the operating system copies a disk page into physical memory only if an attempt is made to access it (i.e., if a page fault occurs). 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 located in 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 execution only 10% of the code.

17.1.1 Advantages

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

17.1.2 Disadvantages

- Individual programs face extra latency when they access a page for the first time. So prepaging, a method of remembering which pages a process used when it last executed and preloading a few of them, is used to improve performance.
- Programs running on low-cost, low-power embedded systems may not have a memory management unit that supports page replacement.
- Memory management with page replacement algorithms becomes slightly more complex.

17.1.3 Pre-paging

OS guesses in advance which pages the process will need and pre-loads them into memory. If the guess is right most of the time, less page faults will occur, which in turn allows more overlap of CPU and I/O. The branches in a code make it difficult to guess the set of pages that should be pre-paged.

17.1.4 Implementation of Demand Paging

A valid bit in the page table being 1 indicates that the corresponding page is in memory. 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). If the page is not in memory, trap to the OS on the first reference. The OS upon verifying that the address is valid, 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. continues faulting process

17.1.5 Swap Space

It is a portion of the disk that is reserved for storing pages that are evicted from 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. Such pages are stored in the swap space. 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. Page table must be sophisticated to figure out where to find a page.

17.1.6 Performance of Demand Paging

In the worst case, a process could access a new page with each instruction. But, typically 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. If p is the probability of a page fault ($0 \leq p \leq 1$), then Effective access time $= (1-p) \cdot ma + p \cdot (pagefaulttime)$

17.2 Page Replacement Algorithms

- **MIN:** This is the theoretically 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 those 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 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. 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. 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 replaces a random page in memory. This eliminates the overhead cost of tracking page references. Usually it fares better than FIFO, and for looping memory references it is better than LRU, although generally LRU performs better in practice.