A Kernel-Based Synchronization Assignment for the Operating Systems Course

Mark A. Holliday
Department of Mathematics and Computer Science
Western Carolina University
Cullowhee, NC 28723
holliday@cs.wcu.edu

A key component of the first operating system course is illustrating how synchronization primitives can be used to solve concurrency problems. Illustrations of synchronization typically use entertaining, but artificial problems. An alternative is using examples from the design of the operating system kernel. Here we describe one example synchronization assignment which is based on the page replacement method used in the 4.3 Berkeley Standard Distribution (BSD) UNIX operating system. The programming is done in Java using threads and the Java synchronization primitives.

A naïve implementation of page fault handling would, upon a page fault if there are no free frames, take the following steps: 1) have the page fault handler select a frame containing a victim page, 2) copy the victim page to disk if it has been modified, 3) load the new page into the frame, 4) reexecute the instruction that caused the page fault. This implementation is inefficient since the latency of choosing a victim page and, possibly, writing it back to disk occurs after the page fault. A more decoupled approach not only has better performance; it also lends itself as a possible synchronization assignment. The decoupled approach uses three threads (the page fault handler, a pageout daemon, and a writer daemon) and two data structures (a freeClean list of frames and a freeDirty list of frames).

The page fault handler is in an infinite loop, blocking at the top of the loop waiting for the next page fault. Upon a page fault, the page fault handler simply chooses a frame from the freeClean list if it is nonempty. If the freeClean list size is below a lower bound, the page fault handler signals the pageout daemon with a notify call. If the freeClean list is empty, the page fault handler blocks using a wait call. As soon as the pageout daemon or the writer daemon makes the freeClean list nonempty, that daemon signals the waiting page fault handler using a notify call. The page fault handler completes handling the page fault and blocks waiting for the next page fault.

The pageout daemon is in an infinite loop, blocking at the top of the loop waiting for when the number of frames on the freeClean list drops below a lower bound. When a notify from the page fault handler unblocks the pageout daemon, the pageout daemon selects victim pages. Each victim page is moved to the freeClean list or freeDirty list depending on if it is dirty (has been modified). Any time the pageout daemon adds a frame to the freeClean list, it notifies the page fault handler. Any time the pageout daemon adds a frame to the freeDirty list, it notifies the writer daemon. When the number of frames on the freeClean list reaches an upper bound, the pageout daemon returns to the top of its infinite loop and blocks.

The writer daemon is also in an infinite loop, blocking at the top of the loop waiting for the pageout daemon to notify it whenever a frame has been added to the freeDirty list. The writer daemon writes the page in that frame to disk, moves the frame to the freeClean list, and notifies the page fault handler that the freeClean list has at least one page. The writer daemon repeats this until the freeDirty list is empty; it then blocks at the top of its loop.

The interplay between the three threads and the two data structures is an interesting example of synchronization and is a realistic design of an essential part of an operating system kernel. Besides the condition synchronization described above, mutual exclusion needs to be enforced during access to the freeClean and freeDirty data structures. The locking to implement mutual exclusion also raises the issue of whether deadlock is possible. Most importantly, this problem is simple enough that it can be assigned as an assignment in the undergraduate operating system course. A handout for this assignment is at http://cs.wcu.edu/~holliday/cware/synchAssg.html. A copy of a solution is available by contacting the author.