CS39002 Operating Systems Laboratory
Spring 2014

Assignment 4
Due on 07–Mar–2014, 1:00pm

In this assignment, you implement a parallel merge sort of an integer array which resides in a shared-memory segment. A tree of processes is created recursively for this purpose.

The root process first reads the size n of the array from the user. It then creates a shared-memory segment A capable of storing n integers with the key derived from the directory */usr/local/lib/*. It then populates the array (after attaching) with randomly generated integer keys. Subsequently, the process tree is created as explained below. You predetermine the maximum level ML of the tree. You also predetermine a minimum size ms of an array segment. These parameters may be read from the user or may be hard-coded definitions in your program.

Let P be a process which is assigned the task of sorting the array segment $A[i \ldots j]$. Suppose that the process is residing at level l. For the root process, $i = 0$, $j = n - 1$, and $l = 0$. For other processes, these values are as dictated by the recursive calls. If $l > ML$ or $j - i + 1 < ms$, then no recursive call is made by P. It uses any sorting algorithm (like your favorite bubble sort) to sort the segment $A[i \ldots j]$. On the other hand, if $l \leq ML$ and $j - i + 1 \geq ms$, then P creates two child processes L and R. Let $M = (i + j) / 2$. The process L handles the sorting of $A[i \ldots M]$, whereas the process R handles the sorting of $A[M+1 \ldots j]$. Since l is the level of P in the recursion tree, the levels of both L and R will be $l + 1$. The process P waits until the two child processes L and R terminate.

If P is the root process, the entire array A is sorted when its two children exit. The root process then outputs the sorted array to the user (for large arrays the output should go to a file). Finally, the root process deletes the shared-memory segment, and exits.

If P is a non-root process, then it has a sibling; call it Q. Both P and Q were forked by their parent. P and Q communicate with one another whether they are ready for parallel merging of their array chunks. Notice that P is in charge of $A[i \ldots j]$, whereas Q is in charge of $A[j+1 \ldots k]$. The two processes merge the two chunks in $A[i \ldots k]$ in parallel. P creates the sorted chunk $A[i \ldots j]$ by working forward from index i through index j, whereas Q creates the sorted chunk $A[j+1 \ldots k]$ by working backward from index k through index $j+1$. Both P and Q first create two local (non-shared) sorted arrays. When both are done, they again communicate to convey that they are both ready to write back their temporary arrays to the shared array A. They do that in parallel. Since the two write-back locations are disjoint, the writing can proceed safely, and P and Q can exit without any further communication between them.

The synchronizing communications between P and Q can be handled in several ways. They may make some blocking calls involving pipes or message queues. An alternative approach is to use another (or the same) shared-memory segment storing the status of the processes. Each process has a dedicated index in the status segment, initialized to zero. When its status changes (on two occasions), it updates its status entry. During synchronization, a process periodically checks its sibling's status. When both P and Q know that they are synchronized, they proceed to the next stage. A third alternative is to use semaphores which are not yet covered. So implement one of the first two approaches.

Bonus part (not for submission)

If you have a multi-core machine, implement the same merging algorithm by a single recursive function in the same process, and check how much speedup the parallel multi-process sorting program achieves over your single-process implementation.

Submit a single file *parMergeSort.c*.