Controlling resource consumption by meting out work items

devblogs.microsoft.com/oldnewthing/20060314-00

March 14, 2006

Raymond Chen

At the PDC, one person came to talk to me for advice on a resource management problem they were having. To simplify, their system generated dozens of work items, each of which required significant resource consumption. For the sake of illustration, let's say that each of the work items was a single-threaded computationally-intensive operation that required 180MB of memory. The target machine was, say, a quad-processor machine with 1GB of RAM. (This was a standalone system, so it could assume that no other programs were running on the computer.) Their first design involved creating a thread for each work item and letting them all fight it out for resources. This didn't work out so great because all of the work items would be fighting over the four CPUs and requiring several times the available system memory, resulting in thrashing both the scheduler (more runnable threads than CPUs) as well as the memory manager (working set larger than available memory). The result was horrible. The second design was to serialize the work items. Run one work item, then when it completes, run the next work item, and so on until all the work items were complete. This ran much better because only one work item was active at a time, so it could monopolize the CPU and memory without interference from other work items. However, this didn't scale because the performance of the program didn't improve after adding processors or memory. Since only one work item ran at a time, all of the extra CPUs and memory simply sat idle. The solution to this problem is to make sure you maximize one of the limiting resources. Here, we have two limiting resources, CPU and memory. Since each work item required an entire CPU, running more work items simultaneously than available CPUs would result in scheduler thrashing, so the first cap on the number of work items was determined by the number of CPUs. Next, since each work item required 180MB of memory, you could run five of them in a 1GB machine before you started thrashing the memory manager. Therefore, this work item list will saturate the CPUs first, at four work items. Once you figure out how many work items you can run at once, it's a simple matter of running only that many. A master scheduler thread maintained a list of work to be done and fired off the first four, then waited for them to complete, using the `WaitForMultipleObjects` function and passing `bWaitAll = FALSE` so that it was woken up as soon as any work item completed. (This was not a GUI application so it didn't need to worry about pumping messages.) As soon as one of the work items completed, a new one was started. In this manner, there were always

four work items in progress, taking maximum advantage of the resources available. (Because our preliminary mathematics showed that running five work items simultaneously would cause the scheduler to thrash.)

In real life, some of the work items were really child processes, and there were dependencies among the work items, but those complications could be accomodated in the basic design.

Raymond Chen

Follow