

Re: Miltu-core CPUs, threads vs AST driven approaches

Source: <http://unix.derkeiler.com/Newsgroups/comp.os.vms/2006-05/msg00322.html>

- *From:* Bill Todd <billtodd@xxxxxxxxxxxxxxx>
 - *Date:* Thu, 04 May 2006 16:29:19 -0400
-

Bill Todd wrote:

....

Assuming, that is, that Bob's comment about the ability to execute ASTs concurrently for different threads in a multi-CPU process is correct (i.e., that inter-thread behavior resembles inter-process behavior in this regard; otherwise, you might be better off with a process per CPU and some shared memory to use for inter-process coordination, but that would be noticeably messier).

And of course it turns out that the above assumption was false. Also, I seem to have been half-asleep when responding, since you can still accomplish about the same thing (distributing many requests efficiently in parallel across a number of threads equal to the number of available processors) even within that constraint.

The way to do that is to run the several 'worker' threads at normal (not AST) level, and use the AST routines only to process I/O (lock-wait, etc.) completion and as part of that completion move the continuation context for the relevant activity onto a 'to do' queue, which each thread visits whenever it has no processing to do (i.e., each time it completes an operation or has to wait for I/O or something similar; the AST routine also wakes a thread if it finds one or more asleep waiting for work). This preserves VMS's guarantee that no more than one AST will be active at a time, and the extremely brief processing required at AST level (compared with the processing required in each thread between interruptions) ensures that this will not impede progress even if many processors are working in parallel in the single process. One can gussy up things by having a separate 'to do' queue for each thread (assumed given affinity to some specific processor) which is used to continue work on already-existing operations (which may help leverage cached data across interruptions) and a main queue used only for new operations and visited only by a thread which has no existing work ready to continue (which is also desirable from the viewpoint of minimizing the multi-programming level without decreasing parallelism: the more operations active at once, the more they will tend to interfere with and delay each other) – though there then needs to be a way for an idle thread to pick up an operation started by some other thread rather than just twiddle its thumbs. You can also of course get into things like varied priorities, priority-inheritance on conflicts, etc. – a great way to roll your own OS facilities in user space if you don't find what the OS provides meets your needs.

The main gotcha here is that **all** the context for each operation (at least context which must be retained across such interruptions) must be held in that queued context structure, rather than conveniently on a thread's stack – since the thread's stack gets unwound before each such interruption. I've recently heard the term 'stack ripping' used for one way of doing this, and I suspect it works just about the way RMS-11's asynchronous

Re: Miltu-core CPUs, threads vs AST driven approaches

processing did almost 30 years ago: the relevant portion of the stack is copied into a save area in the context structure, and restored to the stack when the operation continues (which is fairly transparent save that one can't retain pointers to material in the stack across interruptions, since the stack may not be restored to the same virtual address range it was saved from: for this reason, one may instead choose to place context explicitly in the operation context structure and operate upon it there).

Generally fun stuff.

– bill

.