Joseph Paris
Cs443

Cooperative Task Management without Manual Stack Management

Or, Event-driven Programming is Not the Opposite of Threaded Programming

By A. Adya, et al.

Perhaps the better of the two is the sub-title for this paper. The idea here is to effectively combine multithreaded programming with event-based programming in order to achieve the 'best of both worlds' concept. Simply, you want to achieve the ease of reasoning allowed for concurrency in the 'even-driven' model, while preserving the readability and maintainability of code associated with 'multithreaded' programming. At a lower level, this exemplifies the differences between manual and automatic stack managing. In the multi-threaded case, automatic stack managing allows for the current state to be kept in data stored on a procedure's program stack while the task is waiting on a blocking operation. Manual stack management, as used by the event-driven model, requires a programmer to rip the code for any given task into event handlers that run to completion without blocking.

Their solution is a hybrid approach that enables both styles to coexist in the same code base, using adaptors to connect between them. This allows for a project to be written in one style but incorporate legacy code written in another. The solution was implemented under the Windows operating system using preemptive threads and cooperative fibers. Cooperative task management is achieved by scheduling multiple fibers on a single thread. A scheduler runs on a special fiber called the *MainFiber* that schedules both manual and automatic stack management code. Both types of stack management code are scheduled by the same scheduler because the Windows fiber package only supports explicitly switching from one fiber to another. When a function, written with manual stack management, calls code with automatic stack management, the caller code is written expecting to never block on I/O, while the callee expects to always block on I/O. To deal with this, a new fiber is created to execute the callee code. The caller resumes as soon as the first burst of execution of the fiber completes. The fiber may run and block on I/O, but when it finishes its work it executes the caller's continuation to resume the caller's part of the task. In this way the caller code does not block while the callee code can at will. On the alternate case, a function with automatic stack management calls a function that manually manages its stack. This case calls for the former function to block for I/O, but the latter function simply schedules I/O and returns. To solve this problem, a special adaptor that calls the manual-stack-management code with a special continuation is required, the function then relinquishes control to the *MainFiber* forcing the adaptor's caller to remain blocked. When the I/O completes, this special continuation runs on the *MainFiber* and resumes the fiber of the blocked adaptor, which in turn resumes the original function waiting for the I/O result.

This is a pretty nice way of dealing with the combination of event-driven and threaded technologies. While I like the concept, I felt their intro, summary, and related work sections were rather weak. It felt like they did a ton of over-explaining for a rather direct problem/solution set.