Cooperative Task Management
without Manual Stack Management

Atul Adya, Jon Howell, Marvin Theimer,
William J. Bolosky, John R. Douceur
Microsoft Research
Two Sub-Topics

Paper presented in two parts

• 1\textsuperscript{st} presents orthogonal concepts of task and stack management
  – States that conventional notions of “Multithreaded” and “Event-Driven” programming incorrectly mix these two concepts.
  – Promotes an alternate model which obtains the benefits and best features of each.

• 2\textsuperscript{nd} suggests that some programmers are not willing to work outside their favorite programming model.
  – Presents a technical solutions which allows “Event-Driven” and “Multithreaded” code to be used within the same application.
Task Management

• Definition: Task – A single flow of control with access to common state data shared with other tasks.

• Task Management Models:
  – Serial
  – Preemptive
  – Cooperative
Stack Management

• Definition: Stack – Set of state data which are local or private to a task.

• Stack Management Models:
  – Automatic –Handled by the compiler, stored on the stack. Local function variables preserved across thread context switch.
  – Manual –Handled by the programmer. Local variables must be manually saved in heap data structures before thread context switch.
Related Concepts

• IO Management
  – Synchronous vs. Asynchronous

• Conflict Management
  – Atomic (safe) access to shared state data
  – May employ locks and condition variables or may be inherent to task management model.

• Data Partitioning
  – Arrangement of shared data to minimize potential conflicts.
“Event-Driven” vs. “Multithreaded” Programming Models

• Conventional usage of model names improperly merges orthogonal concepts of task and stack management

• Event-Driven
  – Cooperative task management with manual stack management

• Multithreaded
  – Preemptive task management with automatic stack management
Benefits of “Multithreaded” Programming Model

• Conceptual task to be performed by program expressed as a single language procedure.
• More comfortable, easier to understand programming style for most programmers.
• Leverages the automatic stack management features of the compiler. Local state data simply placed in local (“automatic”) variables.
• Acceptable to block on synchronous I/O calls, which are more widely available than asynchronous I/O calls.
Disadvantages of “Multithreaded” Programming Model

• Shared state data must be protected by locks or other constructs manually invoked by the programmer.
• Locks must be carefully considered and applied otherwise data corruption and deadlocks may occur.
• Shared state data cached locally may no longer be consistent after yielding system or I/O calls.
• Not always apparent which calls will yield, may be hidden beneath called application functions.
• Failure to apply locks correctly may lead to subtle, timing dependent errors which are hard to reproduce and debug.
Benefits of “Event-Driven” Programming Model

• Event handlers run to completion without blocking or pre-emption.
  – Atomicity guaranteed by thread management model.
  – Shared data guaranteed to be consistent when handler starts.
  – No possibility of deadlock or data corruption due to misapplied locks or condition variables.
Disadvantages of “Event-Driven” Programming Model

• Threads typically must be split across multiple event handlers. This can make intended program function difficult to analyze and understand.
• Local state must be manually stored when I/O or other delayed calls are made.
  – Parameters, local variables, and next function pointer stored in continuation objects.
• Software evolution may require addition of continuation object references to entire call graph branch.
• Debugging is difficult due to absence of event chain call stack.
• Asynchronous I/O must be used, but not always provided by the OS or programming framework.
Cooperative Task Management withAutomatic Stack Management

• Benefits of “Event-Driven” model primarily due to atomicity provided by cooperative task management.

• Benefits of “Multithreaded” model primarily due to automatic stack management provided by compiler.

• Possible to take advantage of the best of both worlds. Use cooperative task management with automatic stack management.
Cooperative Task Management with Automatic Stack Management

• Each task continues to run until it explicitly yields.

• A task yields when it completes its task and returns or when making certain system calls.

• A task must never block, therefore, only asynchronous I/O calls can be used.
Benefits of Cooperative Task Management with Automatic Stack Management

- Can use a procedural programming style which is more natural and easily understood to many programmers.
- Atomicity guaranteed by task management model, no need for locks and their inherent risks.
- Stack management provided by the compiler, no need to store local state in heap allocated continuation objects.
- Debugging features are preserved. When a breakpoint is hit the program control flow and all local variables are available for inspection on the stack.
- Software evolution is made easier. Changes to add yielding calls do not require changes to function signatures or program structure.
Disadvantages of Cooperative Task Management with Automatic Stack Management

• If asynchronous I/O is not provided by the OS, then I/O calls must be wrapped by functions which start I/O operations and yield. Provisions must be made to mark the task as ready when I/O completes.
• Global state may have been modified upon return from a yielding call. Any locally cached state or intermediate results must be reloaded and/or recalculated.
• Typically no language features to explicitly identify which functions may yield.
• Authors propose a method to identify each function’s atomicity assumptions and a run-time mechanism to identify when those assumptions have been violated.
Hybrid Approach

- Some programmers are more experienced and comfortable with automatic stack management as used in the “Multithreaded” model.
- Other are more comfortable with manual stack management as used in the “Event-driven” model.
- Some programmers may be unwilling to use a common programming model which is different from their favorite.
- To resolve this problem, the authors designed a framework which allows code written in either model to work together within a single application.
- Also allows legacy code written in a different style to be called without significant refactoring.
Hybrid Approach

• A scheduler was designed which schedules threads cooperatively within an application.
• Scheduler runs on the application main thread.
• Procedures written in “Event-handling” model (Event handlers) also executed in context of the main thread.
• Procedures written in “Multithreaded” model are executed in the context of their own thread.
Hybrid Approach

• I/O or other potentially blocking calls must be wrapped in functions which make them asynchronous.
• Control returned to main thread when these calls are encountered.
• Scheduler then schedules next ready thread.
• Control returned to “blocked” thread when I/O operation is complete.
• Per definition of cooperative thread model, only one thread within the application runs at a time.
Hybrid Approach

• To allow functions written for threaded model to be called from functions written for event model (and vice-versa) adapter functions must be written.

• Adapter functions hide the details of the called function’s stack management model from the caller.
Calling Threaded from Event Code

• Threaded code expects to be able to block, but event code must run to completion without blocking.
• Adapter function creates thread to run threaded code and starts thread. If thread blocks on I/O, then control is returned to adapter function. Adapter returns to caller (event handler), which returns and allows scheduler to run.
• Continuation created by I/O call allows blocked thread to be scheduled when I/O operation completes.
• Continuation created by adapter allows next event handler to be scheduled when threaded code completes.
Calling Event from Threaded Code

• A single logical procedure may be broken into multiple event handlers.

• Threaded code expects to call one function which returns when procedure is complete.

• Adapter function allows event handlers to return immediately. If procedure is complete, then adapter returns to threaded caller. If not, then adapter switches to the main thread to allow the scheduler to run.
Summary

• Conventional usage of “Multithreaded” and “Event-driven” terms confuses orthogonal issues of task and stack management.

• Many programmers unnecessarily adopt “Event-driven” model due to conflict management concerns inherent with preemptive task management.

• The manual stack management inherent in “Event-driven” programming makes code evolution unnecessarily difficult.

• Conflict management can be resolved using cooperative task management while facilitating easier code maintenance and evolution using automatic stack management.