

Cooperative Task Management without Manual Stack Management

Event-driven Programming is Not the Opposite of Threaded
Programming

Josh Triplett

January 31, 2006

Outline

- Previous work
- Typical thread and event models
- Five mostly orthogonal factors
- Task management versus stack management
- Implementation details

Outline

- Previous work
- Typical thread and event models
- Five mostly orthogonal factors
- Task management versus stack management
- Implementation details

Outline

- Previous work
- Typical thread and event models
- **Five mostly orthogonal factors**
- Task management versus stack management
- Implementation details

Outline

- Previous work
- Typical thread and event models
- Five mostly orthogonal factors
- **Task management versus stack management**
- Implementation details

Outline

- Previous work
- Typical thread and event models
- Five mostly orthogonal factors
- Task management versus stack management
- **Implementation details**

Lauer and Needham

- Threads and events “equivalent”
 - No fundamental reason for performance to differ
 - Conclusion: matter of preference
 - Use whichever best fits application

Lauer and Needham

- Threads and events “equivalent”
- No fundamental reason for performance to differ
- Conclusion: matter of preference
- Use whichever best fits application

Lauer and Needham

- Threads and events “equivalent”
- No fundamental reason for performance to differ
- **Conclusion: matter of preference**
- Use whichever best fits application

Lauer and Needham

- Threads and events “equivalent”
- No fundamental reason for performance to differ
- Conclusion: matter of preference
- Use whichever best fits application

Ousterhout

- Events simpler than threads
- No synchronization issues
- Use threads when true concurrency needed
- Issue: I/O

Ousterhout

- Events simpler than threads
- **No synchronization issues**
- Use threads when true concurrency needed
- Issue: I/O

Ousterhout

- Events simpler than threads
- No synchronization issues
- Use threads when true concurrency needed
- Issue: I/O

Ousterhout

- Events simpler than threads
- No synchronization issues
- Use threads when true concurrency needed
- Issue: I/O

SEDA

- **Compelling performance case**
 - High complexity
 - Shared data requires synchronization
 - Events imply breaking up computation

SEDA

- Compelling performance case
- High complexity
- Shared data requires synchronization
- Events imply breaking up computation

SEDA

- Compelling performance case
- High complexity
- Shared data requires synchronization
- Events imply breaking up computation

SEDA

- Compelling performance case
- High complexity
- Shared data requires synchronization
- Events imply breaking up computation

Typical thread-based architecture

- Automatic, preemptive scheduling
- Automatic stack management
- Blocking I/O OK
- Synchronization required

Typical thread-based architecture

- Automatic, preemptive scheduling
- Automatic stack management
- Blocking I/O OK
- Synchronization required

Typical thread-based architecture

- Automatic, preemptive scheduling
- Automatic stack management
- **Blocking I/O OK**
- Synchronization required

Typical thread-based architecture

- Automatic, preemptive scheduling
- Automatic stack management
- Blocking I/O OK
- Synchronization required

Typical event-based architecture

- **Manual scheduling of event handlers**
- Manual stack management - stack ripping
- Blocking I/O not OK
- Synchronization avoidable

Typical event-based architecture

- Manual scheduling of event handlers
- Manual stack management - stack ripping
- Blocking I/O not OK
- Synchronization avoidable

Typical event-based architecture

- Manual scheduling of event handlers
- Manual stack management - stack ripping
- **Blocking I/O not OK**
- Synchronization avoidable

Typical event-based architecture

- Manual scheduling of event handlers
- Manual stack management - stack ripping
- Blocking I/O not OK
- Synchronization avoidable

Typical models not the only way

- SEDA demonstrated an event-based system which used threads
- “Why Events Are A Bad Idea” demonstrated a thread system with event-like scheduling

Typical models not the only way

- SEDA demonstrated an event-based system which used threads
- “Why Events Are A Bad Idea” demonstrated a thread system with event-like scheduling

Five mostly orthogonal factors

- **Task management**
- Stack management
- I/O management
- Conflict management
- Data partitioning

Five mostly orthogonal factors

- Task management
- **Stack management**
- I/O management
- Conflict management
- Data partitioning

Five mostly orthogonal factors

- Task management
- Stack management
- **I/O management**
- Conflict management
- Data partitioning

Five mostly orthogonal factors

- Task management
- Stack management
- I/O management
- **Conflict management**
- Data partitioning

Five mostly orthogonal factors

- Task management
- Stack management
- I/O management
- Conflict management
- Data partitioning

Task management

- **What computation runs next?**
 - Thread scheduling: which thread to run?
 - Event scheduling: which event to process?
 - Options: serial, cooperative, preemptive

Task management

- What computation runs next?
- Thread scheduling: which thread to run?
- Event scheduling: which event to process?
- Options: serial, cooperative, preemptive

Task management

- What computation runs next?
- Thread scheduling: which thread to run?
- **Event scheduling: which event to process?**
- Options: serial, cooperative, preemptive

Task management

- What computation runs next?
- Thread scheduling: which thread to run?
- Event scheduling: which event to process?
- Options: serial, cooperative, preemptive

Thread-based task management

- Typically preemptive
 - Can switch to another thread at any arbitrary point
 - Thread can't plan preemption points
 - Threads can be cooperative

Thread-based task management

- Typically preemptive
- Can switch to another thread at any arbitrary point
- Thread can't plan preemption points
- Threads can be cooperative

Thread-based task management

- Typically preemptive
- Can switch to another thread at any arbitrary point
- Thread can't plan preemption points
- Threads can be cooperative

Thread-based task management

- Typically preemptive
- Can switch to another thread at any arbitrary point
- Thread can't plan preemption points
- **Threads can be cooperative**

Event-based task management

- Typically cooperative
 - Thread explicitly yields control
 - Thread can plan preemption points
 - Some event systems are preemptive

Event-based task management

- Typically cooperative
- Thread explicitly yields control
- Thread can plan preemption points
- Some event systems are preemptive

Event-based task management

- Typically cooperative
- Thread explicitly yields control
- Thread can plan preemption points
- Some event systems are preemptive

Event-based task management

- Typically cooperative
- Thread explicitly yields control
- Thread can plan preemption points
- Some event systems are preemptive

Preemptive versus Cooperative

- Preemptive systems require synchronization
- Preemptive systems typically have less latency
- Cooperative systems require more care to avoid delays
- Cooperative systems have potentially greater performance

Preemptive versus Cooperative

- Preemptive systems require synchronization
- Preemptive systems typically have less latency
- Cooperative systems require more care to avoid delays
- Cooperative systems have potentially greater performance

Preemptive versus Cooperative

- Preemptive systems require synchronization
- Preemptive systems typically have less latency
- **Cooperative systems require more care to avoid delays**
- Cooperative systems have potentially greater performance

Preemptive versus Cooperative

- Preemptive systems require synchronization
- Preemptive systems typically have less latency
- Cooperative systems require more care to avoid delays
- Cooperative systems have potentially greater performance

Function not requiring I/O

- Receive parameters
- Declare local variables
- Do some work
- Return result

Function not requiring I/O

- Receive parameters
- Declare local variables
- Do some work
- Return result

Function not requiring I/O

- Receive parameters
- Declare local variables
- Do some work
- Return result

Function not requiring I/O

- Receive parameters
- Declare local variables
- Do some work
- **Return result**

Function doing blocking I/O

- Receive parameters
- Declare local variables
- Do some work
- Do I/O
- Do some more work
- Return result

Function doing blocking I/O

- Receive parameters
- **Declare local variables**
- Do some work
- Do I/O
- Do some more work
- Return result

Function doing blocking I/O

- Receive parameters
- Declare local variables
- Do some work
 - Do I/O
 - Do some more work
 - Return result

Function doing blocking I/O

- Receive parameters
- Declare local variables
- Do some work
- Do I/O
- Do some more work
- Return result

Function doing blocking I/O

- Receive parameters
- Declare local variables
- Do some work
- Do I/O
- Do some more work
- Return result

Function doing blocking I/O

- Receive parameters
- Declare local variables
- Do some work
- Do I/O
- Do some more work
- Return result

Function doing non-blocking I/O

- **Function 1**
 - Receive parameters and caller's continuation
 - Declare local variables
 - Do some work
 - Schedule I/O and register continuation
 - Return to scheduler
- **Function 2**
 - Receive I/O results
 - Retrieve parameters, caller's continuation, and local variables from continuation
 - Do some more work
 - Call caller's continuation with return value

Function doing non-blocking I/O

- **Function 1**
 - Receive parameters and caller's continuation
 - Declare local variables
 - Do some work
 - Schedule I/O and register continuation
 - Return to scheduler
- **Function 2**
 - Receive I/O results
 - Retrieve parameters, caller's continuation, and local variables from continuation
 - Do some more work
 - Call caller's continuation with return value

Function doing non-blocking I/O

- **Function 1**
 - Receive parameters and caller's continuation
 - Declare local variables
 - Do some work
 - Schedule I/O and register continuation
 - Return to scheduler
- **Function 2**
 - Receive I/O results
 - Retrieve parameters, caller's continuation, and local variables from continuation
 - Do some more work
 - Call caller's continuation with return value

Function doing non-blocking I/O

- **Function 1**
 - Receive parameters and caller's continuation
 - Declare local variables
 - Do some work
 - Schedule I/O and register continuation
 - Return to scheduler
- **Function 2**
 - Receive I/O results
 - Retrieve parameters, caller's continuation, and local variables from continuation
 - Do some more work
 - Call caller's continuation with return value

Function doing non-blocking I/O

- Function 1
 - Receive parameters and caller's continuation
 - Declare local variables
 - Do some work
 - Schedule I/O and register continuation
 - Return to scheduler
- Function 2
 - Receive I/O results
 - Retrieve parameters, caller's continuation, and local variables from continuation
 - Do some more work
 - Call caller's continuation with return value

Function doing non-blocking I/O

- Function 1
 - Receive parameters and caller's continuation
 - Declare local variables
 - Do some work
 - Schedule I/O and register continuation
 - Return to scheduler
- Function 2
 - Receive I/O results
 - Retrieve parameters, caller's continuation, and local variables from continuation
 - Do some more work
 - Call caller's continuation with return value

Function doing non-blocking I/O

- Function 1
 - Receive parameters and caller's continuation
 - Declare local variables
 - Do some work
 - Schedule I/O and register continuation
 - Return to scheduler
- Function 2
 - Receive I/O results
 - Retrieve parameters, caller's continuation, and local variables from continuation
 - Do some more work
 - Call caller's continuation with return value

Function doing non-blocking I/O

- Function 1
 - Receive parameters and caller's continuation
 - Declare local variables
 - Do some work
 - Schedule I/O and register continuation
 - Return to scheduler
- Function 2
 - Receive I/O results
 - Retrieve parameters, caller's continuation, and local variables from continuation
 - Do some more work
 - Call caller's continuation with return value

Function doing non-blocking I/O

- Function 1
 - Receive parameters and caller's continuation
 - Declare local variables
 - Do some work
 - Schedule I/O and register continuation
 - Return to scheduler
- Function 2
 - Receive I/O results
 - Retrieve parameters, caller's continuation, and local variables from continuation
 - Do some more work
 - Call caller's continuation with return value

Function doing non-blocking I/O

- Function 1
 - Receive parameters and caller's continuation
 - Declare local variables
 - Do some work
 - Schedule I/O and register continuation
 - Return to scheduler
- Function 2
 - Receive I/O results
 - Retrieve parameters, caller's continuation, and local variables from continuation
 - Do some more work
 - Call caller's continuation with return value

Function doing non-blocking I/O

- Function 1
 - Receive parameters and caller's continuation
 - Declare local variables
 - Do some work
 - Schedule I/O and register continuation
 - Return to scheduler
- Function 2
 - Receive I/O results
 - Retrieve parameters, caller's continuation, and local variables from continuation
 - Do some more work
 - Call caller's continuation with return value

What is a Continuation?

- **Saved program state**
 - Nonlocal goto to restore state
 - Similar to C's setjmp/longjmp but more capable
 - (setjmp/longjmp can't create true concurrency)
 - Some languages have native continuations

What is a Continuation?

- Saved program state
- **Nonlocal goto to restore state**
- Similar to C's `setjmp/longjmp` but more capable
- (`setjmp/longjmp` can't create true concurrency)
- Some languages have native continuations

What is a Continuation?

- Saved program state
- Nonlocal goto to restore state
- **Similar to C's setjmp/longjmp but more capable**
- (setjmp/longjmp can't create true concurrency)
- Some languages have native continuations

What is a Continuation?

- Saved program state
- Nonlocal goto to restore state
- Similar to C's `setjmp/longjmp` but more capable
- **(`setjmp/longjmp` can't create true concurrency)**
- Some languages have native continuations

What is a Continuation?

- Saved program state
- Nonlocal goto to restore state
- Similar to C's setjmp/longjmp but more capable
- (setjmp/longjmp can't create true concurrency)
- Some languages have native continuations

Manual stack management

- Requires explicit implementation of basic language features
- Function definitions split up
- Automatic variables become manual
- Control structures split across functions
- Debugging stack incomplete
- “Advantage”: explicit notion of yielding passed up stack

Manual stack management

- Requires explicit implementation of basic language features
- **Function definitions split up**
- Automatic variables become manual
- Control structures split across functions
- Debugging stack incomplete
- “Advantage”: explicit notion of yielding passed up stack

Manual stack management

- Requires explicit implementation of basic language features
- Function definitions split up
- **Automatic variables become manual**
- Control structures split across functions
- Debugging stack incomplete
- “Advantage”: explicit notion of yielding passed up stack

Manual stack management

- Requires explicit implementation of basic language features
- Function definitions split up
- Automatic variables become manual
- **Control structures split across functions**
- Debugging stack incomplete
- “Advantage”: explicit notion of yielding passed up stack

Manual stack management

- Requires explicit implementation of basic language features
- Function definitions split up
- Automatic variables become manual
- Control structures split across functions
- Debugging stack incomplete
- “Advantage”: explicit notion of yielding passed up stack

Manual stack management

- Requires explicit implementation of basic language features
- Function definitions split up
- Automatic variables become manual
- Control structures split across functions
- Debugging stack incomplete
- “Advantage”: explicit notion of yielding passed up stack

Automatic stack management

- **Allows normal use of language features**
- Control flow more obvious
- “Disadvantage”: Hides whether a function yields
- If yielding declared, can be checked statically or dynamically

Automatic stack management

- Allows normal use of language features
- **Control flow more obvious**
- “Disadvantage”: Hides whether a function yields
- If yielding declared, can be checked statically or dynamically

Automatic stack management

- Allows normal use of language features
- Control flow more obvious
- “Disadvantage”: Hides whether a function yields
- If yielding declared, can be checked statically or dynamically

Automatic stack management

- Allows normal use of language features
- Control flow more obvious
- “Disadvantage”: Hides whether a function yields
- If yielding declared, can be checked statically or dynamically

Existing framework

- **Event-based framework**
- Manual task management
- Manual stack management
- Non-blocking I/O
- Used continuations
- C++ object holding state; manually packaged and unpackaged

Existing framework

- Event-based framework
- **Manual task management**
- Manual stack management
- Non-blocking I/O
- Used continuations
- C++ object holding state; manually packaged and unpackaged

Existing framework

- Event-based framework
- Manual task management
- **Manual stack management**
- Non-blocking I/O
- Used continuations
- C++ object holding state; manually packaged and unpackaged

Existing framework

- Event-based framework
- Manual task management
- Manual stack management
- **Non-blocking I/O**
- Used continuations
- C++ object holding state; manually packaged and unpackaged

Existing framework

- Event-based framework
- Manual task management
- Manual stack management
- Non-blocking I/O
- **Used continuations**
- C++ object holding state; manually packaged and unpackaged

Existing framework

- Event-based framework
- Manual task management
- Manual stack management
- Non-blocking I/O
- Used continuations
- C++ object holding state; manually packaged and unpackaged

Fibers

- Windows-specific, but portable analogues exist
 - Manually scheduled
 - No generic “yield”
 - Explicitly switch to specified fiber

Fibers

- Windows-specific, but portable analogues exist
- **Manually scheduled**
- No generic “yield”
- Explicitly switch to specified fiber

Fibers

- Windows-specific, but portable analogues exist
- Manually scheduled
- No generic “yield”
- Explicitly switch to specified fiber

Fibers

- Windows-specific, but portable analogues exist
- Manually scheduled
- No generic “yield”
- Explicitly switch to specified fiber

Adaptors

- Allow use of manual and automatic stack management in one framework
- Use fibers to maintain multiple stacks
- Blocking operations run on alternate fibers

Adaptors

- Allow use of manual and automatic stack management in one framework
- Use fibers to maintain multiple stacks
- Blocking operations run on alternate fibers

Adaptors

- Allow use of manual and automatic stack management in one framework
- Use fibers to maintain multiple stacks
- Blocking operations run on alternate fibers

Similar work in “Why Events are a Bad Idea”

- Same idea of separating different concepts
- Cooperative thread-based model with event-like properties
- New concept: practical adaptors between the two models

Similar work in “Why Events are a Bad Idea”

- Same idea of separating different concepts
- Cooperative thread-based model with event-like properties
- New concept: practical adaptors between the two models

Similar work in “Why Events are a Bad Idea”

- Same idea of separating different concepts
- Cooperative thread-based model with event-like properties
- **New concept: practical adaptors between the two models**

Summary

- **Task management independent from stack management**
- Cooperative task management useful
- Automatic stack management preferable
- Can achieve both with cooperative threading
- Can use adaptors to support both models simultaneously

Summary

- Task management independent from stack management
- **Cooperative task management useful**
- Automatic stack management preferable
- Can achieve both with cooperative threading
- Can use adaptors to support both models simultaneously

Summary

- Task management independent from stack management
- Cooperative task management useful
- **Automatic stack management preferable**
- Can achieve both with cooperative threading
- Can use adaptors to support both models simultaneously

Summary

- Task management independent from stack management
- Cooperative task management useful
- Automatic stack management preferable
- **Can achieve both with cooperative threading**
- Can use adaptors to support both models simultaneously

Summary

- Task management independent from stack management
- Cooperative task management useful
- Automatic stack management preferable
- Can achieve both with cooperative threading
- Can use adaptors to support both models simultaneously