ADVANCED OPERATING SYSTEMS

Mesa, Why Threads, Why Events

Experience with Processes and Monitors in Mesa (1980)

- What kind of paper is this?
  - New big idea?
  - Measurement paper?
  - Experiences/lessons learnt paper?
  - A system description?
  - Performance study?
  - Refute-conventional wisdom?
  - Survey paper?
  - Something else?

Experience with Processes and Monitors in Mesa (1980)

- What kind of paper is this?
  - Experiences/lessons learnt paper
  - A system description

Focus of this paper

- Light-weight processes (threads in today's terminology) and how they synchronize with each other

History

- 2nd system; followed the Alto
- Planned to build a large system using many programmers (some thoughts about commercializing)
- Advent of things like server machines and networking introduced applications that are heavy users of concurrency

Considerations

- Chose to build a single address space system:
  - Single user system, so protection not an issue (safety was to come from the language)
  - Wanted global resource sharing
- Large system, took many programmers
  - Module-based programming with information hiding
- Since they were starting “from scratch”, they could integrate the hw, the runtime sw, and the language with each other

Programming model choice

- Programming model for inter-process communication: shared memory (monitors) vs. message passing
  - Needham & Lauer claimed the two models are duals of each other

How to synchronize processes?

- Non-preemptive scheduler tends to yield very delicate systems. Why?
- What is non-preemptive scheduling?

Non-preemptive vs. preemptive scheduling

- In non-preemptive scheduling, a thread runs until it terminates, stops, blocks, suspends, or yields.
- In preemptive scheduling, even if the current thread is still running, a context switch will (likely) occur when its time slice is used up. Ways for thread to leave running state:
  - It ceases to be ready to execute (e.g., by calling a blocking I/O method)
  - It gets preempted by a high-priority thread which becomes ready to execute
  - It explicitly calls a thread-scheduling method such as wait or suspend

How to synchronize processes?

- Non-preemptive scheduler tends to yield very delicate systems. Why?

How to synchronize processes?

- Non-preemptive scheduler tends to yield very delicate systems. Why?
  - Have to know whether or not a yield might be called for *every* procedure you call. Violates information hiding
  - Prohibits multiprocessor systems
  - Need a separate preemptive mechanism for I/O anyway
  - Can't do multiprogramming across page faults

How to synchronize processes?

- Simple locking (e.g., semaphores): too little structuring discipline
  - No guarantee that locks will be released on every code path
  - Wanted something that could be integrated into a Mesa language construct
- Message passing (vs shared memory)
  - Needham & Lauer claimed two models are duals of each other
  - Hard to integrate with Mesa
  - Chose shared memory model because they thought they could fit it into Mesa as a language construct more naturally

How to synchronize processes?

- Chose preemptive scheduling of light-weight processes and monitors

The Mesa programming language

- Modules, strong type checking
- Lightweight processes
- Monitors

Mesa processes – lightweight processes

- All processes share a single address space
- Easy forking: any procedure can be forked
- Fast performance for creation, switching and synchronization: low storage overhead
- Integrated in the language
  - Process is a first-class type in language
  - Why is this good?

Mesa processes – lightweight processes

- All processes share a single address space
- Easy forking: any procedure can be forked
- Fast performance for creation, switching and synchronization: low storage overhead
- Integrated in the language
  - Process is a first-class type in language
  - Why is this good?
    - Subject to same strict type checking as other constructs so compiler can catch frequent errors

Monitors

- Monitor lock (for synchronization)
  - Tied to module structure of the language: makes it clear what's being monitored
  - like processes, monitors were a language construct
  - Language automatically acquires and releases the lock
- Tied to a particular ***invariant,*** which helps users think about the program
- Condition variables (for scheduling)

Monitors

- 3 types of procedures in a monitor module
  - Entry (acquires and releases lock)
  - Internal (no locking done): can't be called from outside the module
  - External (no locking done): externally callable. Why is this useful?

Monitors

- 3 types of procedures in a monitor module
  - Entry (acquires and releases lock)
  - Internal (no locking done): can't be called from outside the module
  - External (no locking done): externally callable. Why is this useful?
    - Allows grouping of related things into a module
    - Allows doing some of the work outside the monitor lock
    - Allows controlled release and reacquisition of monitor lock
  - Caller can't tell difference between External and Entry
  - Caller can't even tell that module has a monitor

Mesa and Java

- Monitors (and Mesa in particular) led to several aspects of Java.
  - Java's synchronized objects are the object-oriented programming version of monitors

Monitors	Java Synchronized Objects
external	?
internal	?
entry	?

Mesa and Java

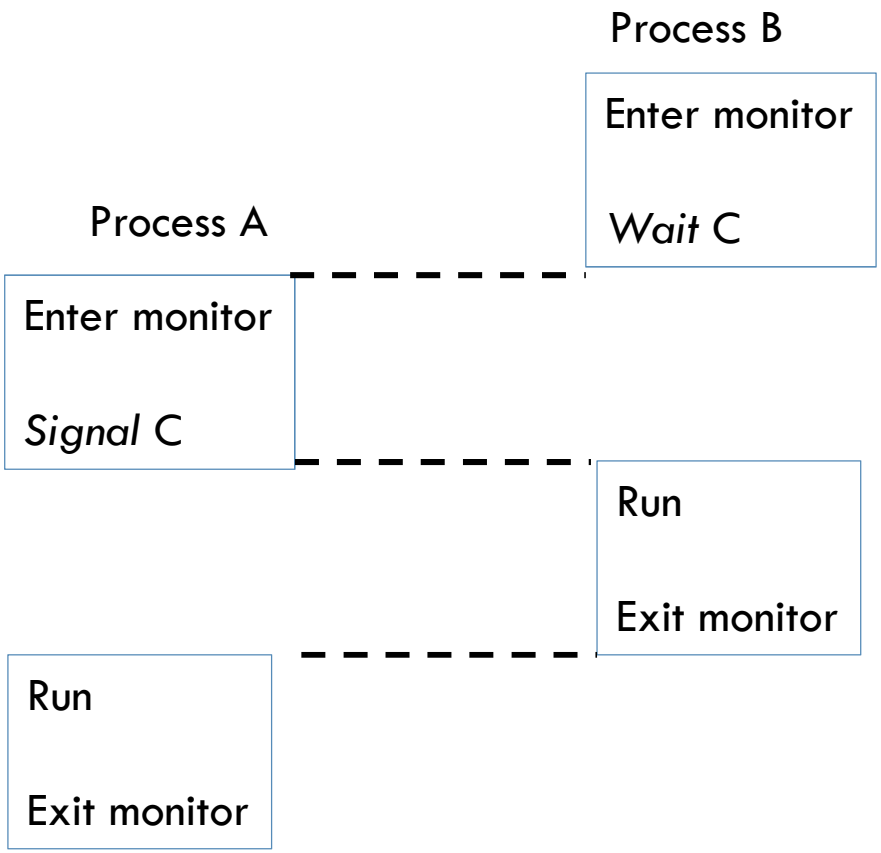
- Monitors (and Mesa in particular) led to several aspects.
  - Java's synchronized objects are the object-oriented programming version of monitors

Monitors	Java Synchronized Objects
external	public
internal	private synchronized
entry	public synchronized

Notify semantics options

- Cede lock to waking process (Hoare style)
  - Let waking process run right away
  - Waking process knows the condition it was waiting on is guaranteed to hold

Hoare-style Notify semantics

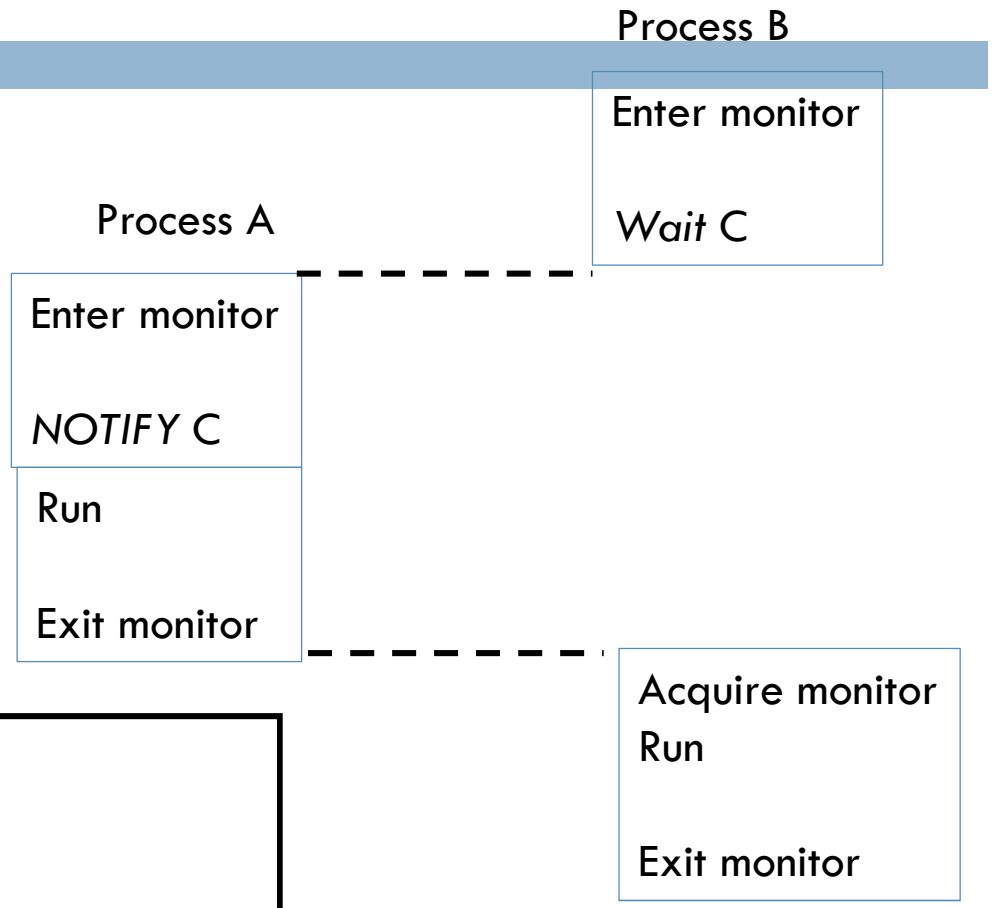


- Problem with Hoare-style semantics:
  - Must establish invariant before executing *Signal*
  - Requires additional context switches

Notify semantics options

- Cede lock to waking process (Hoare style)
  - Let waking process run right away
  - Waking process knows the condition it was waiting on is guaranteed to hold
- Notifier keeps lock, waking process get put in front of monitor queue.
  - Doesn't work in the presence of priorities
- **What they chose:** Notifier keeps lock, wakes process with no guarantees
  - Waking process must recheck its condition

Mesa-style Notify semantics



- Advantages:
  - Can notify at anytime. Notify is a hint.
  - Fewer context switches
- Disadvantage:
  - Process must check after Wait returns.

  If (!ready to go) WAIT() → while (!ready to go) WAIT()

Aside: hints vs guarantees

- Notify is only a hint
  - Don't have to wake up the right process, don't have to change the notifier if we slightly change the wait condition (the two are decoupled)
  - Easier to implement, because it's always OK to wake up too many processes. If we get lost, we could even wake up everybody (broadcast)
  - Enables timeouts and aborts
- General principle: use hints for performance that have little effect or no effect on correctness

Other Mesa wakeups

- Timeouts: wait until notified or 10 seconds
- Abort: feeble sort of process termination
  - Allows target process to reach a wait or monitor exit and then it voluntarily aborts
  - No need to re-establish the invariant – as compared to just killing the process outright!
- Broadcast: wake everybody who is waiting
  - What's the problem with example in Sec. 3.1?
  - Why not always use broadcast?

Additional problem: deadlock

- *Wait* only releases the current monitor locks
  - nested calls to monitors are not released
- Need to avoid cyclic dependencies between monitors
  - Impose partial ordering
- General problem with modular systems and synchronization
  - Synchronization requires *global* knowledge about locks, which violates the information hiding paradigm of modular programming
  - Absolute hierarchy of locks isn't always feasible

Lock granularity

- Mesa has finer-grain locking than single lock for code
  - Monitored record: a monitor lock per data object
  - Useful for parallelism in multiprocessors
- General locking trade-off: fine-grained vs coarse-grained

Interrupts

- Device can't block waiting to acquire a monitor lock
- Introduced *naked* notifies: notifies done without holding the monitor lock
- Had to worry about a timing race: the notify could occur between a monitor's condition check and its call on Wait.
- Removes race condition by adding *wakeup-waiting* flag to condition variable

Priority inversion

- High-priority processes may block on lower-priority processes
- Solution: temporarily increase the priority of the holder of the monitor lock to that of the highest priority block process
- The Mars rover stalled due to this kind of bug and had to be debugged and fixed from earth!

Exceptions

- Must restore monitor invariant as you unwind the stack. What does Java do? (you must use a sequence of try-finally blocks)
- Idea that you can just kill a process and release its locks is naïve
  - Each lock protects some invariant that must be restored before you can release the lock
  - Entry procedures that have an exception but no exception handler DO NOT release the monitor lock.
    - Ensures deadlock but at least it maintains the invariant

Three key features of the paper

- Describes the experiences designers had with designing, building, and using a large system that aggressively relies on heavy-weight processes and monitor facilities for all its concurrency needs
- Describes various *subtle issues* of implementing a threads-with-monitors design in real life for a large system
- Discusses the performance and overhead of various primitives and three representative apps, but doesn't give big picture of how important various things turned out to be

Some flaws

- Gloss over how hard it is to program with locks and exceptions sometimes. (Not clear if there are better ways!)
- Performance discussion doesn't give the big picture
- **A lesson: the light-weight threads-with-monitors programming paradigm can be used to successfully build large systems, but there are subtle points that have to be correct in the design and implementation to do so.**

The Mesa Legacy

- Ironed out the practical aspects of monitor usage
- Created standard idioms of concurrent programming in shared address spaces
- Created standardized interfaces
- Influenced design of Java
- Had a huge impact on Topaz threads design, which had a big impact on the POSIX pthread interface

Events versus Threads (over) simplified

- Threads can block, so we make use of the CPU by switching between threads
- Event handlers cannot block, so we can make use of the CPU by simply running events to completion
  - We must write handlers so they are so small they don't ever block, and if they need to block, they create an event that someone else will have to handle

Events versus Threads (over) simplified

- With threads, getting good concurrency relies on switching between them
- With event-based programming, getting good concurrency relies on writing events in a way that I can process a lot of them in a row really quickly and keep the system busy

Why Events are a Bad Idea (for high con-currency servers) (2003)

- What kind of paper is this?
  - New big idea?
  - Measurement paper?
  - Experiences/lessons learnt paper?
  - A system description?
  - Performance study?
  - Refute-conventional wisdom?
  - Survey paper?
  - Something else?

Why Events are a Bad Idea (for high concurrency servers) (2003)

- What kind of paper is this?
  - A position paper

What's the position?

What's the position?

- "Event-based programming is the wrong choice for highly concurrent systems"
- Perceived weaknesses of threads are "artifacts of specific threading implementations and are not inherent to the threading paradigm"

“Problems” with Threads (1)

- Performance
  - Criticism: Many attempts to use threads for high concurrency have not performed well
  - Response: the implementation is to blame; many current packages have ops that are O(n) in number of threads
    - Repeat SEDA benchmark with their package and show performance matches event-based server

"Problems" with Threads (2)

- Control flow
  - Criticism: Threads have restrictive control flow
    - Makes programmer think "too linearly about control flow", potentially precluding use of more efficient control flow patterns
  - Response: complicated control flow patterns are rare in practice
    - Most control flow patterns fall into:
      - Call/return
      - Parallel calls
      - Pipelines
    - These patterns are expressed more naturally with threads
      - Common event patterns map cleanly onto call/return mechanism of threads
      - Robust systems need acks for error handling, cleanup, etc. so they need a "return" event in the event model

“Problems” with Threads (3)

- Synchronization
  - Criticism: thread synchronization mechanisms are too heavyweight
    - Event systems claim that cooperative multitasking gives them synchronization “for free” -- what does this mean?

"Problems" with Threads (3)

- Synchronization
  - Criticism: thread synchronization mechanisms are too heavyweight
    - Event systems claim that cooperative multitasking gives them synchronization "for free" -- no overhead from supporting mutexes, handling wait queue, etc.
  - Response?

"Problems" with Threads (3)

- Synchronization
  - Criticism: thread synchronization mechanisms are too heavyweight
    - Event systems claim that cooperative multitasking gives them synchronization "for free" -- no overhead from supporting mutexes, handling wait queue, etc.
  - Response: thread systems can also perform cooperative multitasking
  - "free" synchronization due to cooperative multitasking holds only on uniprocesses. Why?

“Problems” with Threads (4)

- State management
  - Criticism: thread stacks are an ineffective way to manage live state, why?

“Problems” with Threads (4)

- State management
  - Criticism: thread stacks are an ineffective way to manage live state
    - Tradeoff between risking stack overflow and wasting virtual address space on large stacks
  - Response?

"Problems" with Threads (5)

- Scheduling
  - Criticism: the virtual processor model provided by threads forces runtime system to be too generic and prevents it from making optimal scheduling decisions
    - Event systems can schedule events to be delivered flexibly (e.g., app chooses shortest remaining completion time scheduling, favors certain request streams, etc.)
  - Response: same scheduling tricks can be applied to cooperatively scheduled threads
    - Making argument for user-level threads

The case for threads (1)

- Threads are more appropriate abstraction for high-concurrency servers
  - In modern servers, concurrent requests are largely independent
  - Code that handles each request is usually sequential

The case for threads (2)

- Event based programming obfuscates control flow of the app
  - Often "call" a method in another module by sending an event and expect a "return" from that method via another event
  - Programmer must mentally match call/return pairs that are in different parts of the code
  - Programmer must manually save and restore live state
    - major burden for programmers using event systems
- Thread systems allow programmers to express control flow and encapsulate state more naturally

The case for threads (3)

- Cleaning task state after exceptions and after normal termination simpler in threaded system
  - Thread stack naturally tracks the live state for that task
- In event systems, task state is heap allocated
  - Freeing state at correct time is tough due to branches in control flow of app
  - Garbage collection mechanisms inadequate

The case for threads (4)

- In existing systems, threads are preferred (at least in most complex parts)

Compiler support for threads

- Dynamic stack growth
- Live state management
- Synchronization

Compiler support for threads

- Dynamic stack growth
  - Size of stack adjusted at run-time
  - Estimate amount of stack space needed when calling each function
- Live state management
  - Compiler purges unnecessary state from stack before making function calls
- Synchronization – use compiler-based analyses for
  - Race detection
  - Determining which atomic sections of code can run concurrently

Why Threads are a Bad Idea (1996)

- Keynote talk by John Ousterhout at Usenix Conference

What is main position of the talk?

What is main position of the talk?

- For most purposes proposed for threads, events are better
- Threads should be used only when true CPU concurrency is needed

Questions

- Ousterhout outlines why threads are hard to program
  - How does this claim mesh with von Behren et al paper's claim that threads are more natural to think about than event-driven programming? What do you think?
- Ousterhout claims debugging easier with events
  - "timing dependencies only related to events, not to internal scheduling"; what do you think?