

# CS 518

## Mesa: Design and Implementation of a Thread System

Randy Wang  
Fall, 2000  
Princeton University

## Outline

- Background
- Design and implementation issues
- Performance tradeoffs

## Outline

- Background
- Design and implementation issues
- Performance tradeoffs

## Background

- Another example of pushing an idea to its logical extreme: light-weight threads
- “2nd System”
  - 1st system was the Alto (pre-PC PC)
  - want to aggressively go after multi-programming in the second system
- Debate of concurrent programming model
  - Message passing, or
  - Shared memory
  - Debate settled on the ground of ease of programming
- Debate of synchronization primitives
  - Non-preemptive scheduler
  - Locks/semaphores
  - Debate settled on the ground of structures

## Threads

Means up to three things

- Easy forking
- Shares a single address space
- Good performance: creation, and switching

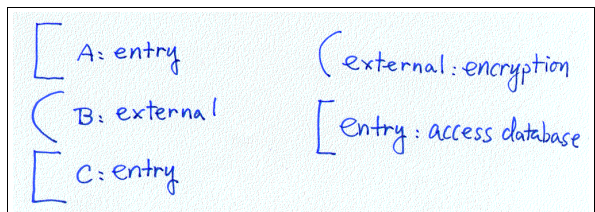
## Monitors

- Mutual exclusion via monitor locks (lock/unlock)
- Scheduling via conditional variables (wait/notify)
- Tied to module structure
- Language acquires/releases locks automatically

## Outline

- ~~Background~~
- **Design and implementation issues**
- Performance tradeoffs

## Separation of Procedures in a Monitor



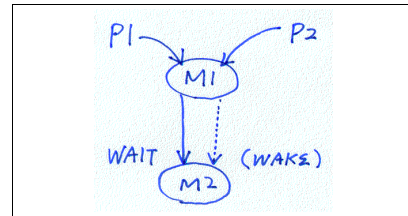
- Entry procedures
  - based on classic Hoare semantics
- Internal procedures
  - called by entry and internal procedures
  - lock held
- External procedures
  - no locking
  - for performance
  - for code sections that don't want locking
  - An example of synchronization messing up with modular structure

## Semantics of Notify

```
while (condition is false) {  
    WAIT(cond_var)  
}
```

- Alternatives
  - Cede lock to woken process
  - Keep lock, woken process gets priority
  - Just wake up \*
- Pros and cons
  - Ceding
    - Woken process can assume some monitor invariant
    - Complicated in terms who else to run after the waking event
  - Priority-based scheduling
    - What happens if you wake up a whole bunch?
  - Do nothing
    - Must code in stylized way to re-test condition
    - Simple semantics and implementation

## Deadlock in Nested Monitors



- Deadlock
  - Only release lock on the “last” monitor on sleep
  - Waker blocked in earlier monitor
- No pleasant solution
  - Recode to release lock in 1st monitor
  - Ugly and difficult to reason about
  - How about automatically releasing all locks? unacceptable semantic burden for reasoning about procedure calls

## Lock Granularity

- Hoare: one lock per module
- Mesa: one lock per object

## Interrupts

```
while (I/O not complete) {  
    WAIT(cond_var) ← notify  
}
```

- Want to use cond var notifies for interrupt
- Problem
  - Interrupts don't respect monitor locks
  - Conditional variables don't have states
  - Lost notify
- Solution
  - Stateful conditional variables

## Outline

- Background
- ~~Design and implementation issues~~
- **Performance tradeoffs**

## Context Switches vs. Procedure Calls

- Goal: want to make context switches as fast as procedure calls
- Two ways of accomplishing this:
  - Make context switches fast
  - Make procedure calls slow
  - They did both in Mesa
    - Context switch: 2x procedure call
    - Procedure calls: 30 instructions, 10x RISC cost
    - (Typical) Unix context switch: 1000-2000 instructions
- Reason
  - Heap-allocated procedure frames
  - (Implementation artifact)
- Is this a good idea?
  - What happens more often?
  - (Anecdote on Mesa)

## Process Creation

- Mesa: ~1100 instructions
- Unix: ~100,000 instructions
- Does this really matter?
  - If you have fast context switches
  - then keep a pool of pre-spawned threads
  - process creation doesn't matter