CSE 451: Operating Systems
Winter 2004

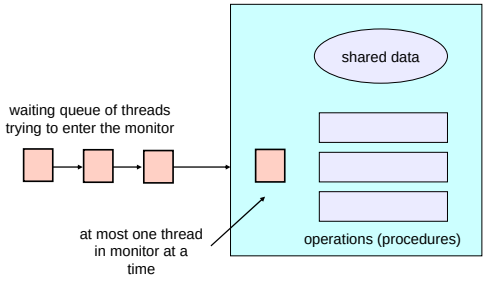
Module 7+
Monitor Supplement

Ed Lazowska
lazowska@cs.washington.edu
Allen Center 570

Monitors

- A *monitor* is a software module that encapsulates:
  - shared data structures
  - procedures that operate on the shared data
  - synchronization between concurrent threads that invoke those procedures
- Data can only be accessed from within the monitor
  - protects the data from unstructured access
- Synchronization code (calls to synchronization routines in the thread package) is added by compiler
  - why does this help?
- Addresses the key usability issues that arise with semaphores

A monitor



Monitor facilities

- "Automatic" mutual exclusion
  - only one thread can be executing inside at any time
    - thus, synchronization "comes for free" with monitor
  - if a second thread tries to execute a monitor procedure, it blocks until the first has left the monitor
- Condition variables
  - once inside, a thread may discover it can't continue, and may wish to block (or allow some other waiting thread to continue)
  - it can wait on a condition variable, or signal others to continue
    - condition variables can only be accessed from within monitor
    - a thread that waits "steps outside" the monitor (onto a wait queue associated with that condition variable)
    - what happens to a thread that signals depends on the precise monitor semantics that are used

Two kinds of monitors

- Hoare monitors: signal(c) means
  - run waiter immediately
  - signaller blocks immediately
    - condition guaranteed to hold when waiter runs
      - can use "if" rather than "while" in previous example
    - but, signaller must restore monitor invariants before signalling!
      - cannot leave a mess for the waiter, who will run immediately!
- Mesa monitors: signal(c) means
  - waiter is made ready, but the signaller continues
    - waiter runs when signaller leaves monitor (or waits)
    - condition is not necessarily true when waiter runs again
      - must use "while" as in previous example
  - signaller need not restore invariant until it leaves the monitor
  - being woken up is only a hint that something has changed
    - must recheck conditional case

Bounded buffer using Hoare monitors

```
Monitor bounded_buffer {
  buffer resources[N];
  condition not_full, not_empty;

  procedure add_entry(resource x) {
    if (array "resources" is full, determined maybe by a count)
      wait(not_full);
    insert "x" in array "resources"
    signal(not_empty);
  }
  procedure get_entry(resource *x) {
    if (array "resources" is empty, determined maybe by a count)
      wait(not_empty);
    *x = get resource from array "resources"
    signal(not_full);
  }
```


Runtime system calls for Hoare monitors

- EnterMonitor(m) {guarantee mutual exclusion}
- ExitMonitor(m) {hit the road, letting someone else run}
- Wait(c) {step out until condition satisfied}
- Signal(c) {if someone's waiting, step out and let him run}

Bounded buffer using Hoare monitors

```
Monitor bounded_buffer {
  buffer resources[N];
  condition not_full, not_empty;

  procedure add_entry(resource x) {                    EnterMonitor
    if (array "resources" is full, determined maybe by a count)
      wait(not_full);
    insert "x" in array "resources"
    signal(not_empty);                                 ExitMonitor
  }
  procedure get_entry(resource *x) {                   EnterMonitor
    if (array "resources" is empty, determined maybe by a count)
      wait(not_empty);
    *x = get resource from array "resources"
    signal(not_full);                                  ExitMonitor
  }
```


Runtime system calls for Hoare monitors

- EnterMonitor(m) {guarantee mutual exclusion}
  - if m occupied, insert caller into queue m
  - else mark as occupied, insert caller into ready queue
  - choose somebody to run
- ExitMonitor(m) {hit the road, letting someone else run}
  - if queue m is empty, then mark m as unoccupied
  - else move a thread from queue m to the ready queue
  - insert caller in ready queue
  - choose someone to run

- Wait(c) {step out until condition satisfied}
  - if queue m is empty, then mark m as unoccupied
  - else move a thread from queue m to the ready queue
  - put the caller on queue c
  - choose someone to run
- Signal(c) {if someone's waiting, step out and let him run}
  - if queue c is empty then put the caller on the ready queue
  - else move a thread from queue c to the ready queue, and put the caller into queue m
  - choose someone to run

Runtime system calls for Mesa monitors

- EnterMonitor(m) {guarantee mutual exclusion}
  - …
- ExitMonitor(m) {hit the road, letting someone else run}
  - …
- Wait(c) {step out until condition satisfied}
  - …
- Signal(c) {if someone's waiting, give him a shot after I'm done}
  - if queue c is occupied, move one thread from queue c to queue m
  - return to caller

- Broadcast(c) {food fight!}
  - move all threads on queue c onto queue m
  - return to caller