Lecture 7: Monitors

Keith Marzullo
Announcements

- *testthreads* now available!
Monitors

- A monitor is a programming language construct that controls access to shared data
  - Synchronization code added by compiler, enforced at runtime
  - Why is this an advantage?
- A monitor is a module that encapsulates
  - Shared data structures
  - Procedures that operate on the shared data structures
  - Synchronization between concurrent threads that invoke the procedures
- A monitor protects its data from unstructured access
- It guarantees that threads accessing its data through its procedures interact only in legitimate ways
Monitor Semantics

- A monitor guarantees mutual exclusion
  - Only one thread can execute any monitor procedure at any time (the thread is “in the monitor”)
  - If a second thread invokes a monitor procedure when a first thread is already executing one, it blocks
    » So the monitor has a wait queue…
  - If a thread within a monitor blocks, another one can enter

- What are the implications in terms of parallelism in a monitor?
Account Example

Monitor `account` {
  double balance;

  double `withdraw`(amount) {
    balance = balance – amount;
    return balance;
  }
}

Hey, that was easy!

But what if a thread wants to wait inside the monitor?
  » Such as “mutex(empty)” by reader in bounded buffer?
Monitors, Monitor Invariants and Condition Variables

- A monitor invariant is a safety property associated with the monitor, expressed over the monitored variables. It holds whenever a thread enters or exits the monitor.
- A condition variable is associated with a condition needed for a thread to make progress once it is in the monitor.

Monitor M {
    ... monitored variables
    Condition c;

    void enter_mon (...) {
        if (extra property not true) wait(c);  // waits outside of the monitor's mutex
        do what you have to do
        if (extra property true) signal(c);   // brings in one thread waiting on condition
    }
}
Condition Variables

- Condition variables support three operations:
  - **Wait** – release monitor lock and wait for condition to be signaled
  - **Signal** – wakeup one waiting thread
  - **Broadcast** – wakeup all waiting threads

- Condition variables are not boolean objects
  - “if (condition_variable) then” … does not make sense
  - “if (num_resources == 0) then wait(resources_available)” does
  - An example will make this more clear
Monitor Bounded Buffer

Monitor `bounded_buffer`

```plaintext
Resource buffer[N];
// Variables for indexing buffer
// monitor invariant involves these vars
Condition not_full; // space in buffer
Condition not_empty; // value in buffer

void put_resource (Resource R) {
    if (buffer array is full)
        wait(not_full);
    Add R to buffer array;
    signal(not_empty);
}

Resource get_resource() {
    if (buffer array is empty)
        wait(not_empty);
    Get resource R from buffer array;
    signal(not_full);
    return R;
}

// end monitor
```

What happens if no threads are waiting when signal is called?
Monitor Queues

Monitor \texttt{bounded\_buffer} \{

\texttt{Condition not\_full;}
\texttt{\ldots other variables\ldots}
\texttt{Condition not\_empty;}

\texttt{void put\_resource () \{
    \ldots wait(not\_full)\ldots
    \ldots signal(not\_empty)\ldots
\} }

\texttt{Resource get\_resource () \{
    \ldots
\} }
\}
Condition Vars != Semaphores

- Condition variables != semaphores
  - Although their operations have the same names, they have entirely different semantics (such is life, worse yet to come)
  - However, they each can be used to implement the other

- Access to the monitor is controlled by a lock
  - `wait()` blocks the calling thread, and gives up the monitor lock
    » To call wait, the thread has to be in the monitor (hence has lock)
    » Semaphore::wait just blocks the thread on the queue
  - `signal()` causes a waiting thread to wake up
    » If there is no waiting thread, the signal is lost
    » Semaphore::signal increases the semaphore count, allowing future entry even if no thread is waiting
    » Condition variables have no history
Signal Semantics

There are two flavors of monitors that differ in the scheduling semantics of signal (in Mesa, notify)

- **Hoare** monitors (original)
  - `signal()` immediately switches from the caller to a waiting thread
  - The condition that the waiter was anticipating is guaranteed to hold when waiter executes
  - Signaler must restore monitor invariants before signaling, and then waits on an *urgent queue* that has priority on re-entering the monitor.

- **Mesa** monitors (Mesa, Java)
  - `notify()` places a waiter on the ready queue, but signaler continues inside monitor
  - Condition is not necessarily true when waiter runs again
    - Returning from `wait()` is only a hint that something changed
    - Must recheck conditional case
Hoare vs. Mesa Monitors

- **Hoare**
  ```
  if (empty)
      wait(condition);
  ```

- **Mesa**
  ```
  while (empty)
      wait(condition);
  ```

- **Tradeoffs**
  - Mesa monitors easier to use, more efficient
    » Fewer context switches, easy to support broadcast
  - Hoare monitors leave less to chance
    » Easier to reason about the program
Monitor Readers and Writers

We'll use Mesa monitor semantics.

- Will have four methods: `StartRead`, `StartWrite`, `EndRead` and `EndWrite`
- Monitored data: \( \text{nr} \) (number of readers) and \( \text{nw} \) (number of writers) with the monitor invariant:
  \[
  (\text{nr} \geq 0) \land (0 \leq \text{nw} \leq 1) \land ((\text{nr} > 0) \Rightarrow (\text{nw} = 0))
  \]
- Two conditions:
  - `canRead`: \( \text{nw} = 0 \)
  - `canWrite`: \( (\text{nr} = 0) \land (\text{nw} = 0) \)
Monitor Readers and Writers

- Write with just `wait()` (will be safe, maybe not live - why?)

```c
Monitor RW {
    int nr = 0, nw = 0;
    Condition canRead, canWrite;

    void StartRead () {
        while (nw != 0) do wait(canRead);
        nr++;
    }

    void EndRead () {
        nr--;
    }

    void StartWrite {
        while (nr != 0 || nw != 0) do wait(canWrite);
        nw++;
    }

    void EndWrite () {
        nw--;
    }
} // end monitor
```
Monitor Readers and Writers

- add notify() and broadcast()

Monitor \texttt{RW}\{ 
  \texttt{int nr = 0, nw = 0;}
  \texttt{Condition canRead, canWrite;}

  \texttt{void StartRead ()}
  \{ 
    \texttt{while (nw != 0) do wait(canRead);}
    \texttt{nr++;}
  \}

  \texttt{void EndRead ()}
  \{ 
    \texttt{nr--;}
    \texttt{if (nr == 0) notify(canWrite);}
  \}

  \texttt{void StartWrite ()}
  \{ 
    \texttt{while (nr != 0 || nw != 0) do wait(canWrite);}
    \texttt{nw++;}
  \}

  \texttt{void EndWrite ()}
  \{ 
    \texttt{nw--;}
    \texttt{broadcast(canRead);}
    \texttt{notify(canWrite);}
  \}

\} // end monitor
Monitor Readers and Writers

- Is there any priority between readers and writers?
- What if you wanted to ensure that a waiting writer would have priority over new readers?
Election monitor

- Monitor for N threads to vote either 0 or 1.
- Method int Vote (int how) returns election outcome once it is determined.
- Consider only one-shot election, and ignore all issues of authentication.
Monitor Election {
    int vote0 = 0, vote1 = 0;
    condition elected;

    int Vote (int how) {
        if (how == 0) vote0++;
        else vote1++;
        while (2*vote0 ≤ N && 2*vote1 ≤ N)
            wait(elected);
        broadcast(elected);
        return (2*vote0 > N ? 0 : 1);
    }
}
Monitors and Java

- A lock and condition variable are in every Java object
  - No explicit classes for locks or condition variables

- Every object is/has a monitor
  - At most one thread can be inside an object’s monitor
  - A thread enters an object’s monitor by
    - Executing a method declared “synchronized”
      - Can mix synchronized/unsynchronized methods in same class
    - Executing the body of a “synchronized” statement
      - Supports finer-grained locking than an entire procedure

- Every object can be treated as a condition variable
  - Object::notify() has similar semantics as Condition::notify()
  - Object::notifyAll() has similar semantics as Condition::broadcast()
A Java monitor

```java
public class BSem {
    int value;

    public BSem(int initial) {
        if (initial == 0) value = 0;
        else value = 1;
    }

    public synchronized void P() {
        while (value == 0) {
            try { wait(); }
            catch (java.lang.InterruptedException e) { };
            value = 0;
        }
    }

    public synchronized void V() {
        value = 1;
        notify();
    }
}
```
Java Monitors

- Why do you think the Java designers eliminated explicit condition variables?
- What drawbacks are there to their design decision?
Summary

- **Semaphores**
  - `wait()`/`signal()` implement blocking mutual exclusion
  - Also used as atomic counters (counting semaphores)
  - Can be inconvenient to use

- **Monitors**
  - Synchronizes execution within procedures that manipulate encapsulated data shared among procedures
    - Only one thread can execute within a monitor at a time
  - Relies upon high-level language support

- **Condition variables**
  - Used by threads as a synchronization point to wait for events
Next time...

- Read Chapters 5 and 7