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 encapsulates
  - Shared data structures
  - Procedures that operate on the shared data structures
  - Synchronization between concurrent threads that invoke the procedures
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 to have a wait queue...

- Monitors also support waiting on conditions
  - Situation: we enter a monitor and find that we need to wait
    » E.g., producer when the buffer is full
  - If we just wait, the monitor is blocked
    » So, monitors support waiting while releasing the monitor
Account Example

Monitor `account` {
  double balance;

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

 Threads block waiting to get into monitor

- 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, wait for C/V to be signaled
    » So condition variables have wait queues, too
  - **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 \texttt{bounded\_buffer} { 
  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 \texttt{put\_resource} (Resource R) { 
    if (buffer array is full) 
      wait(not\_full);
      \textit{Add R to buffer array};
    signal(not\_empty);
  } 

  Resource \texttt{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 `bounded_buffer`

```
Condition not_full;
...other variables...
Condition not_empty;

void put_resource () {
  ...wait(not_full)...
  ...signal(not_empty)...
}

Resource get_resource () {
  ...
}
```
Condition Vars != Semaphores

● Condition variables != semaphores
  ✷ Although their operations have the same names, they have entirely different semantics
  ✷ 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 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()
  - **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
  - **Mesa** monitors (Mesa, Java)
    - signal() 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

Using Mesa monitor semantics.

- Will have four methods: **StartRead**, **StartWrite**, **EndRead** and **EndWrite**
- Monitored data: **nr** (number of readers) and **nw** (number of writers) with the monitor invariant
  
  \[(nr \geq 0) \land (0 \leq nw \leq 1) \land ((nr > 0) \Rightarrow (nw = 0))\]

- Two conditions:
  - **canRead**: \(nw = 0\)
  - **canWrite**: \((nr = 0) \land (nw = 0)\)
Monitor Readers and Writers

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

Monitor `RW`

```c
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 signal() and broadcast()

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

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

    void EndRead () {
        nr--;
        if (nr == 0) signal(canWrite);
    }

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

    void EndWrite () {
        nw--;
        broadcast(canRead);
        signal(canWrite);
    }

} // end monitor
```

```
can we put a signal here?
```

```
can we put a signal here?
```

```
can we put a signal here?
```

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?
Condition Vars & Locks

- Condition variables are also used without monitors in conjunction with blocking locks.
- A monitor is “just like” a module whose state includes a condition variable and a lock.
  - Difference is syntactic; with monitors, compiler adds the code.
- It is “just as if” each procedure in the module calls acquire() on entry and release() on exit.
  - But can be done anywhere in procedure, at finer granularity.
- With condition variables, the module methods may wait and signal on independent conditions.
Using Cond Vars & Locks

- Alternation of two threads (ping-pong)
- Each executes the following:

```c
Lock lock;
Condition cond;

void ping_pong () {
    acquire(lock);
    while (1) {
        printf("ping or pong\n");
        signal(cond, lock);
        wait(cond, lock);
    }
    release(lock);
}
```

- Must acquire lock before you can wait (similar to needing interrupts disabled to call Sleep in Nachos)
- Wait atomically releases lock and blocks until signal()
- Wait atomically acquires lock before it returns
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::signal()
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
  - Inside monitors, or outside with locks
Looking ahead

- Scheduling – how to decide which thread/process to run next?
  - Does it matter? Yes!

- Other concurrency issues
  - Deadlock – the deadly embrace!

- Midterm!
More synchronization practice

- Cars cross a narrow bridge. Bridge be open in one direction at a time. It can hold no more than three cars. Build threads to simulate this situation.

- Change the problem above to avoid starvation of one direction. Let's say we don’t let more than 10 cars cross in any direction if there are cars waiting on the other direction.
You take a break from studying cs153 to play frisbee with your friends. We have one or more frisbees, and there are two or more of you. Each student waits until they have a frisbee and their neighbor doesn’t have one and then throws the frisbee.

What happens if the number of frisbees is equal to the number of players?
More synchronization practice

- CS153 students are studying for the midterm over the national CS dish (Pizza). Each pizza pie has 8 slices. Each student eyes the pie, then grabs the next slice.
  - What race conditions can happen?
  - How can you resolve them?

- A student that grabs the last slice should order the next pie. Extend your implementation to do that