Introducing Monitors

- Separate the concerns of mutual exclusion and conditional synchronization
- What is a monitor?
  - One lock, and
  - Zero or more condition variables for managing concurrent access to shared data
- General approach:
  - Collect related shared data into an object/module
  - Define methods for accessing the shared data
- Monitors were first introduced as a programming language construct
  - Calling a method defined in the monitor automatically acquires the lock
  - Examples: Mesa, Java (synchronized methods)
- Monitors also define a programming convention
  - Can be used in any language (C, C++, …)

Locks and Condition Variables – Recap

- Locks
  - Provide mutual exclusion
  - Support two methods
    - Lock::Acquire() – wait until lock is free, then grab it
    - Lock::Release() – release the lock, waking up a waiter, if any
- Condition variables
  - Support conditional synchronization
  - Three operations
    - Wait(): Release lock; wait for the condition to become true; reacquire lock upon return
    - Signal(): Wake up a waiter, if any
    - Broadcast(): Wake up all the waiters
  - Two semantics for the implementation of wait() and signal()
    - Hoare monitor semantics
    - Hansen monitor semantics

Coke Machine Example

```
Class CokeMachine{
    …
    Lock lock;
    int count = 0;
    Condition notFull, notEmpty;
}
```

```
CokeMachine::Deposit(){
    lock→acquire();
    while (count == n) {
            notFull.wait(&lock); }
    Add coke to the machine;
    count++;
    notEmpty.signal();
    lock→release();
}
```

```
CokeMachine::Remove(){
    lock→acquire();
    while (count == 0) {
            notEmpty.wait(&lock); }
    Remove coke from to the machine;
    count--;
    notFull.signal();
    lock→release();
}
```


Hoare Monitors: Semantics

- Hoare monitor semantics:
  - Assume thread *T1* is waiting on condition *x*
  - Assume thread *T2* is in the monitor
  - Assume thread *T2* calls *x*.signal
  - *T2* gives up monitor, *T2* blocks!
  - *T1* takes over monitor, runs
  - *T1* gives up monitor
  - *T2* takes over monitor, resumes
- Example

```
fn1(…)
…
x.wait      // T1 blocks   ──→  fn4(…)
                                …
// T1 resumes          ←──     x.signal    // T2 blocks
Lock→release();
                       ──→     T2 resumes
```

Hansen Monitors: Semantics

- Hansen monitor semantics:
  - Assume thread *T1* waiting on condition *x*
  - Assume thread *T2* is in the monitor
  - Assume thread *T2* calls *x*.signal; wake up T1
  - *T2* continues, finishes
  - When T1 get a chance to run, *T1* takes over monitor, runs
  - *T1* finishes, gives up monitor
- Example:

```
fn1(…)
…
x.wait      // T1 blocks  ──→  fn4(…)
                               …
                               x.signal    // T2 continues
                     ←──────   // T2 finishes
// T1 resumes
// T1 finishes
```


Tradeoff

Hoare

- Claims:
  - Cleaner, good for proofs
  - When a condition variable is signaled, it does not change
  - Used in most textbooks
- …but
  - Inefficient implementation

```
CokeMachine::Deposit(){
   lock→acquire();
   if (count == n) {
            notFull.wait(&lock); }
   Add coke to the machine;
   count++;
   notEmpty.signal();
   lock→release();
}
```

Hansen

- Signal is only a "hint" that the condition may be true
  - Need to check condition again before proceeding
  - Can lead to synchronization bugs
- Used by most systems
- Benefits:
  - Efficient implementation
  - Condition guaranteed to be true once you are out of while !

```
CokeMachine::Deposit(){
   lock→acquire();
   while (count == n) {
            notFull.wait(&lock); }
   Add coke to the machine;
   count++;
   notEmpty.signal();
   lock→release();
}
```


Summary

- Synchronization
  - Coordinating execution of multiple threads that share data structures
- Past lectures:
  - Locks → provide mutual exclusion
  - Condition variables → provide conditional synchronization
- Today: Historical perspective
  - Semaphores
    - Introduced by Dijkstra in 1960s
    - Two types: binary semaphores and counting semaphores
    - Supports both mutual exclusion and conditional synchronization
  - Monitors
    - Separate mutual exclusion and conditional synchronization

Concurrent Programming Issues: Summary

Summary of Our Discussions

- Developing and debugging concurrent programs is hard
  - Non-deterministic interleaving of instructions
- Synchronization constructs
  - Locks: mutual exclusion
  - Condition variables: conditional synchronization
  - Other primitives:
    - Semaphores
      - Binary vs. counting
      - Can be used for mutual exclusion and conditional synchronization
- How can you use these constructs effectively?
  - Develop and follow strict programming style/strategy

Programming Strategy

- Decompose the problem into objects
- Object-oriented style of programming
  - Identify shared chunk of state
  - Encapsulate shared state and synchronization variables inside objects

General Programming Strategy

- Two step process
- Threads:
  - Identify units of concurrency – these are your threads
  - Identify chunks of shared state – make each shared "thing" an object; identify methods for these objects (how will the thread access the objects?)
  - Write down the main loop for the thread
- Shared objects:
  - Identify synchronization constructs
    - Mutual exclusion vs. conditional synchronization
  - Create a lock/condition variable for each constraint
  - Develop the methods –using locks and condition variables – for coordination

Coding Style and Standards

- Always do things the same way
- Always use locks and condition variables
- Always hold locks while operating on condition variables
- Always acquire lock at the beginning of a procedure and release it at the end
  - If it does not make sense to do this → split your procedures further
- Always use while to check conditions, not if

```
while (predicate on state variable) {
        conditionVariable→wait(&lock);
        };
```

- (Almost) never sleep() in your code
  - Use condition variables to synchronize

Readers/Writers: A Complete Example

- Motivation
  - Shared databases accesses
    - Examples: bank accounts, airline seats, …
- Two types of users
  - Readers: Never modify data
  - Writers: read and modify data
- Problem constraints
  - Using a single lock is too restrictive
    - Allow multiple readers at the same time
    - …but only one writer at any time
  - Specific constraints
    - Readers can access database when there are no writers
    - Writers can access database when there are no readers/writers
    - Only one thread can manipulate shared variables at any time

Readers/Writer: Solution Structure

- Basic structure: two methods

```
Database::Read() {
      Wait until no writers;
      Access database;
      check out – wake up waiting writers;
}
```

```
Database::Write() {
      Wait until no readers/writers;
      Access database;
      check out – wake up waiting readers/writers;
}
```

- State variables

```
AR = 0;   // # of active readers
AW = 0;   // # of active writers
WR = 0;   // # of waiting readers
WW = 0;  // # of waiting writers
Condition okToRead;
Condition okToWrite;
Lock lock;
```


Solution Details: Readers

```
Public Database::Read() {
      StartRead();
      Access database;
      DoneRead();
}
```

```
Provate Database::StartRead() {
      lock.Acquire();
      while ((AW+WW) > 0) {
             WR++;
             okToRead.wait(&lock);
             WR--;
      }
      AR++;
      lock.Release();
}
```

```
Provate Database::DoneRead() {
      lock.Acquire();
      AR--;
      if (AR ==0 && WW > 0) {
             okToWrite.signal();
      }
      lock.Release();
}
```

Solution Details: Writers

```
Database::Write() {
	StartWrite();
	Access database;
	DoneWrite();
}
```

```
Provate Database::StartWrite() {
	lock.Acquire();
	while ((AW+AR) > 0) {
		WW++;
		okToWrite.wait(&lock);
		WW--;
	}
	AW++;
	lock.Release();
}
```

```
Provate Database::DoneWrite() {
	lock.Acquire();
	AW--;
	if (WW > 0) {
		okToWrite.signal();
	}
	else if (WR > 0) {
		okToRead.broadcast();
	}
	lock.Release();
}
```