Semaphores and Monitors:
High-level Synchronization Constructs

Synchronization Constructs

- **Synchronization**
  - Coordinating execution of multiple threads that share data structures

- **Past few lectures:**
  - Locks: provide mutual exclusion
  - Condition variables: provide conditional synchronization

- **Today: Historical perspective**
  - Monitors
    - Alternate high-level language constructs
  - Semaphores
    - Introduced by Dijkstra in 1960s
    - Main synchronization primitives in early operating systems
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 first introduced as 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++, ...)

Critical Section: Monitors

- Basic idea:
  - Restrict programming model
  - Permit access to shared variables only within a critical section
- General program structure
  - Entry section
    - "Lock" before entering critical section
    - Wait if already locked
    - Key point: synchronization may involve wait
  - Critical section code
  - Exit section
    - "Unlock" when leaving the critical section
- Object-oriented programming style
  - Associate a lock with each shared object
  - Methods that access shared object are critical sections
  - Acquire/release locks when entering/exiting a method that defines a critical section
Remember Condition Variables

- **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 (Java wait())
    - Signal(): Wake up a waiter, if any (Java notify())
    - Broadcast(): Wake up all the waiters (Java notifyAll())
  - Two semantics for implementation of wait() and signal()
    - Hoare monitor semantics
    - Hansen (Mesa) monitor semantics

Coke Machine – Example Monitor

```java
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 the machine;
    count--;
    notFull.signal();
    lock.release();
}
```

Does the order of acquire/while() {wait} matter?
Order of release/signal matter?
Every monitor function should start with what?
- A. wait
- B. signal
- C. lock acquire
- D. lock release
- E. signalAll

Hoare Monitors: Semantics

- Hoare monitor semantics:
  - Assume thread $T_1$ is waiting on condition $x$
  - Assume thread $T_2$ is in the monitor
  - Assume thread $T_2$ calls $x$.signal
  - $T_2$ gives up monitor, $T_2$ blocks!
  - $T_1$ takes over monitor, runs
  - $T_1$ gives up monitor
  - $T_2$ takes over monitor, resumes

- Example

```plaintext
fn1(…)
...
x.wait; // T1 blocks -> fn4(…)
...
// T1 resumes ← x.signal // T2 blocks
Lock.release(); → T2 resumes
```
Hansen (Mesa) Monitors: Semantics

- Hansen monitor semantics:
  - Assume thread $T_1$ waiting on condition $x$
  - Assume thread $T_2$ is in the monitor
  - Assume thread $T_2$ calls $x$.signal; wake up $T_1$
  - $T_2$ continues, finishes
  - When $T_1$ get a chance to run, $T_1$ takes over monitor, runs
  - $T_1$ finishes, gives up monitor

- Example:

  ```
  fnA(…)
  ...
  $x$.wait  // $T_1$ blocks
  ...
  $x$.signal  // $T_2$ continues
  // $T_1$ resumes
  // $T_1$ 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
    - Not modular - correctness depends on correct use and implementation of signal

- 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 (e.g., Java)
  - Benefits:
    - Efficient implementation
    - Condition guaranteed to be true once you are out of while!

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

Nested Monitor Calls

- What happens when one monitor calls into another?
  - What happens to CokeMachine::lock if thread sleeps in CokeTruck::Unload?
  - What happens if truck unloader wants a coke?

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

```java
CokeTruck::Unload()
  lock.acquire();
  while (soda.atDoor() != coke) {
    cokeAvailable.wait(&lock);
  }
  Unload soda closest to door;
  soda.pop();
  Signal availability for soda.atDoor();
  lock.release();
```

More Monitor Headaches

The priority inversion problem

- Three processes (P1, P2, P3), and P1 & P3 communicate using a monitor M. P3 is the highest priority process, followed by P2 and P1.
- 1. P1 enters M.
- 2. P1 is preempted by P2.
- 3. P2 is preempted by P3.
- 4. P3 tries to enter the monitor, and waits for the lock.
- 5. P2 runs again, preventing P3 from running, subverting the priority system.
- A simple way to avoid this situation is to associate with each monitor the priority of the highest priority process which ever enters that monitor.
Other Interesting Topics

- Exception handling
  - What if a process waiting in a monitor needs to time out?

- Naked notify
  - How do we synchronize with I/O devices that do not grab monitor locks, but can notify condition variables.

- Butler Lampson and David Redell, “Experience with Processes and Monitors in Mesa.”

Semaphores

- Study these for history and compatibility
  - Don’t use semaphores in new code

- A non-negative integer variable with two atomic and isolated operations

  Semaphore \( \rightarrow P() \) (*Passeren*; wait)
  
  If \( \text{sem} > 0 \), then decrement \( \text{sem} \) by 1
  
  Otherwise "wait" until \( \text{sem} > 0 \) and
  
  then decrement

  Semaphore \( \rightarrow V() \) (*Vrijgeven*; signal)
  
  Increment \( \text{sem} \) by 1
  
  Wake up a thread waiting in \( P() \)

- We assume that a semaphore is *fair*
  - No thread \( t \) that is blocked on a \( P() \) operation remains blocked if the \( V() \) operation on the semaphore is invoked infinitely often
  - In practice, FIFO is mostly used, transforming the set into a queue.
### Important properties of Semaphores

- Semaphores are *non-negative* integers
- The *only* operations you can use to change the value of a semaphore are `P()` and `V()` (except for the initial setup)
  - `P()` can block, but `V()` never blocks
- Semaphores are used both for
  - *Mutual exclusion*, and
  - *Conditional synchronization*
- Two types of semaphores
  - Binary semaphores: Can either be 0 or 1
  - General/Counting semaphores: Can take any non-negative value
  - Binary semaphores are as expressive as general semaphores (given one can implement the other)

---

- **How many possible values can a binary semaphore take?**
  - A. 0
  - B. 1
  - C. 2
  - D. 3
  - E. 4
Using Semaphores for Mutual Exclusion

- Use a *binary semaphore* for mutual exclusion
  
  ```java
  Semaphore = new Semaphore(1);
  Semaphore->P();
  Critical Section;
  Semaphore->V();
  ```

- Using Semaphores for producer-consumer with bounded buffer

```
int count;
Semaphore mutex;
Semaphore fullBuffers;
Semaphore emptyBuffers;
```

Use a separate semaphore for each constraint

---

Coke Machine Example

- Coke machine as a shared buffer
- Two types of users
  - Producer: Restocks the coke machine
  - Consumer: Removes coke from the machine
- Requirements
  - Only a single person can access the machine at any time
  - If the machine is out of coke, wait until coke is restocked
  - If machine is full, wait for consumers to drink coke prior to restocking
- How will we implement this?
  - How many lock and condition variables do we need?
    - A. 1 B. 2 C. 3 D. 4 E. 5
Revisiting Coke Machine Example

```c
class CokeMachine{
    int count;
    Semaphore new mutex(1);
    Semaphores new fullBuffers(0);
    Semaphores new emptyBuffers(numBuffers);
}

void CokeMachine::Deposit(){
    emptyBuffers->P();
    mutex->P();
    Add coke to the machine;
    count++;
    mutex->V();
    fullBuffers->V();
}

void CokeMachine::Remove(){
    fullBuffers->P();
    mutex->P();
    Remove coke from the machine;
    count--;
    mutex->V();
    emptyBuffers->V();
}
```

Does the order of P matter?  Order of V matter?

Implementing Semaphores

```c
Semaphore::P() {
    if (value == 0) {
        Put TCB on wait queue for semaphore;
        Switch(); // dispatch a ready thread
    } else {value--;
}
}

Semaphore::V() {
    if wait queue is not empty {
        Move a waiting thread to ready queue;
    } value++;
}
```

Does this work?
Implementing Semaphores

Semaphore::P() {
    while (value == 0) {
        Put TCB on wait queue for semaphore;
        Switch(); // dispatch a ready thread
    }
    value--;
}

Semaphore::V() {
    if wait queue is not empty {
        Move a waiting thread to ready queue;
    }
    value++;
}

The Problem with Semaphores

◆ Semaphores are used for dual purpose
   ➢ Mutual exclusion
   ➢ Conditional synchronization

◆ Difficult to read/develop code

◆ Waiting for condition is independent of mutual exclusion
   ➢ Programmer needs to be clever about using semaphores

CokeMachine::Deposit(){
    emptyBuffers->P();
    mutex->P();
    Add coke to the machine;
    count++;
    mutex->V();
    fullBuffers->V();
}

CokeMachine::Remove(){
    fullBuffers->P();
    mutex->P();
    Remove coke from to the machine;
    count--;
    mutex->V();
    emptyBuffers->V();
}
Comparing Semaphores and Monitors

CokeMachine::Deposit()
{
    lock->acquire();
    while (count == n) {
        notFull.wait(&lock);
    }
    Add coke to the machine;
    count++;
    mutex->P();
    fullBuffers->V();
}

CokeMachine::Remove()
{
    fullBuffers->P();
    mutex->P();
    Remove coke from the machine;
    count--;
    mutex->V();
    emptyBuffers->V();
}

Which is better?
A. Semaphore
B. Monitors

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

CokeMachine::Remove()
{
    lock->acquire();
    while (count == 0) {
        notEmpty.wait(&lock);
    }
    count--;
    notFull.notify();
    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:
  - 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