**Condition Synchronization**

**Beyond Locks**

- Locks ensure mutual exclusion
- Is this sufficient?
  - What if you want to synchronize on a condition?
  - Example: Producer-consumer problem

```java
class BoundedBuffer{
  Lock lock;
  int count = 0;
}

BoundedBuffer::Deposit(c){
  lock->acquire();
  while (count == n);
  Add c to the buffer;
  count++;
  lock->release();
}

BoundedBuffer::Remove(c){
  lock->acquire();
  while (count == 0);
  Remove c from buffer;
  count--;
  lock->release();
}
```

**Introducing Condition Variables**

- Correctness requirements for bounded buffer producer-consumer problem
  - Only one thread manipulates the buffer at any time (mutual exclusion)
  - Consumer must wait for producer when the buffer is empty (scheduling/synchronization constraint)
  - Producer must wait for the consumer when the buffer is full (scheduling/synchronization constraint)
- Solution: condition variables
  - An abstraction that supports conditional synchronization
  - Key idea:
    - Enable threads to wait inside a critical section by atomically releasing lock at the same time

**Condition Variables: Operations**

- Three operations
  - Wait():
    - Release lock
    - Go to sleep
    - Reacquire lock upon return
    - Signal():
      - Wake up a waiter, if any
    - Broadcast():
      - Wake up all the waiters
- Implementation
  - Requires a per-condition variable queue to be maintained
  - Threads waiting for the condition wait for a signal()
Implementing `Wait()` and `Signal()`

```c
Condition::Wait(lock){
    numWaiting++;
    lock->release();
    Put TCB on the waiting queue for the CV;
    switch();
    lock->acquire();
}
```

```c
Condition::Signal(){
    if (numWaiting > 0) {
        Move a TCB from waiting queue to ready queue;
        numWaiting--;
    }
}
```

Using Condition Variables: An 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?
  - What is the class definition?
  - How many lock and condition variables do we need?

**Coke Machine Example**

```c
class CokeMachine{
    Lock lock;
    int count = 0;
    Condition notFull, notEmpty;
}
```

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

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

Semaphores and Monitors: High-level Synchronization Constructs

A Historical Perspective
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
  - Semaphores
    - Introduced by Dijkstra in 1960s
    - Main synchronization primitives in early operating systems
  - Monitors
    - Alternate high-level language constructs

Semaphores

- An abstract data type
- A non-negative integer variable with two atomic operations

<table>
<thead>
<tr>
<th>Semaphore(\rightarrow) P() (Passeren; wait)</th>
</tr>
</thead>
<tbody>
<tr>
<td>If (sem &gt; 0), then decrement (sem) by 1</td>
</tr>
<tr>
<td>Otherwise &quot;wait&quot; until (sem &gt; 0) and then</td>
</tr>
<tr>
<td>decrement</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Semaphore(\rightarrow) V() (Vrijgeven; signal)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Increment (sem) by 1</td>
</tr>
<tr>
<td>Wake up a thread waiting in P()</td>
</tr>
</tbody>
</table>

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 mainly 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)

Using Semaphores for Mutual Exclusion

- Use a binary semaphore for mutual exclusion

<table>
<thead>
<tr>
<th>Semaphore = new Semaphore(1);</th>
</tr>
</thead>
<tbody>
<tr>
<td>Semaphore(\rightarrow) P();</td>
</tr>
<tr>
<td>Critical Section;</td>
</tr>
<tr>
<td>Semaphore(\rightarrow) V();</td>
</tr>
</tbody>
</table>

- Using Semaphores for producer-consumer with bounded buffer

<table>
<thead>
<tr>
<th>Semaphore mutex;</th>
</tr>
</thead>
<tbody>
<tr>
<td>Semaphore fullBuffers;</td>
</tr>
<tr>
<td>Semaphore emptyBuffers;</td>
</tr>
</tbody>
</table>

Use a separate semaphore for each constraint.
Revisiting Coke Machine Example

Class CokeMachine{
    Semaphore new mutex(1);
    Semaphores new fullBuffers(0);
    Semaphores new emptyBuffers(numBuffers);
}

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

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

Does the order of P matter? V?

Implementing Semaphores

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

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

Implementation Semaphores

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

Semaphore::V() {
    Disable interrupts;
    if wait queue is not empty {
        Move a waiting thread to ready queue;
    } else {
        value++;
    }
    Enable interrupts;
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:
   mutex→V();
   fullBuffers→V();
CokeMachine::Remove()
   fullBuffers→P();
   mutex→P();
   Remove coke from the machine:
   mutex→V();
   emptyBuffers→V();

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) {
      notFull.wait(&lock);
   }
   Remove coke from to the machine:
   count--;
   notFull.signal();
   lock→release();
**Hoare Monitors: Semantics**

- Hoare monitor semantics:
  - Assume thread $T_1$ is waiting on condition $x$
  - Assume thread $T_2$ is in the monitor
  - $T_2$ 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
  ```
  fn1(...)
  ...
  x.wait       // $T_1$ blocks
  // $T_1$ resumes
  Lock->release();
  T2 resumes
  ```

**Hansen 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:
  ```
  fn1(...)
  ...
  x.wait       // $T_1$ blocks
  // $T_1$ resumes
  ```

**Tradeoff**

**Hoare**

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

- but
  - Inefficient implementation

**Hansen**

- Signal is only a "hint" that the condition may be true
  - Used by most systems
  - Efficient implementation
  - Condition guaranteed to be true once you are out of while!

**Hansen v. Hoare semantics**

The priority inversion problem

Consider a set of communicating processes with varying priority
With Hoare semantics a low priority process can delay the progress of a high priority process.
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

Summary

- Non-deterministic order of thread execution → concurrency problems
- Multiprocessing
  - A system may contain multiple processors → cooperating threads/processes can execute simultaneously
- Multi-programming
  - Thread/process execution can be interleaved because of time-slicing

Goal:
- Ensure that your concurrent program works under all possible interleaving

Approach:
- Define synchronization constructs and programming style for developing concurrent programs
- Locks → provide mutual exclusion
- Condition variables → provide conditional synchronization