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

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

\[
\text{Semaphore } \to \text{V()} \quad (\text{Vrijgeven; signal}) \\
\text{Increment } \text{sem} \text{ by 1} \\
\text{Wake up a thread waiting in P()}
\]

- 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, one can implement the other)
**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
  
  ```java
  Semaphore mutex;
  Semaphore fullBuffers;
  Semaphore emptyBuffers;
  ```

**Revisiting Coke Machine Example**

```java
Class CokeMachine{
    Semaphore new mutex(1):
    Semaphore new fullBuffers(0);
    Semaphore new emptyBuffers(numBuffers);
}
```

```java
CokeMachine::Deposit()
  emptyBuffers=P():
  mutex=P();
  Add coke to the machine:
  mutex=V():
  fullBuffers=V():
}
```

```java
CokeMachine::Remove()
  fullBuffers=P():
  mutex=P();
  Remove coke from the machine:
  mutex=V():
  emptyBuffers=V():
}
```

**Implementing Semaphores**

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

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

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

```c
Semaphore::V() {
    Disable interrupts;
    if wait queue is not empty {
        Move a waiting thread to ready queue;
    }
    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) {
    notEmpty.wait(&lock);
  }
  Remove coke from to the machine;
  count--;
  notFull.signal();
  lock.release();
}
```

Hoare Monitors: Semantics

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

```
Example:
fn(...)
{ ...
  x.wait // T1 blocks
  x.signal // T2 blocks
  // T1 resumes
  lock.release();
}
```

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 T2 get a chance to run, T2 takes over monitor, runs
  - T2 finishes, gives up monitor

```
Example:
fn(...)
{ ...
  x.wait // T1 blocks
  x.signal // T2 blocks
  // T1 resumes
  lock.release();
}
```

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

```
fn(...)
{ ...
  lock.acquire();
  if (count == n) {
    notFull.wait(&lock);
  }
  Add coke to the machine;
  count++;
  notFull.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!

```
fn(...)
{ ...
  lock.acquire();
  while (count == n) {
    notFull.wait(&lock);
  }
  Add coke to the machine;
  count++;
  notFull.signal();
  lock.release();
}
```
**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.

**Diagram:**

- Diagram illustrating the priority inversion problem with processes `P`, `R`, and `Q` communicating through shared buffers and using `signal` and `wait` operations.

- High-Priority Producer
- Low-Priority Consumer
- Shared Buffers
- `signal(notEmpty)`
- `wait(notEmpty)`
- `deposit()`
- `remove()`