Semaphores and Monitors: High-level Synchronization Constructs

A Historical Perspective

**Semaphores**

- An abstract data type
- A non-negative integer variable with two *atomic* operations
  - Semaphore→P() (Passeren; wait)
    - Atomically: If sem > 0, then decrement sem by 1
    - Otherwise, "wait" until sem > 0
  - Semaphore→V() (Vrijgeven; signal)
    - Atomically: Increment sem by 1

- We assume that a semaphore is *fair*
  - No thread 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
  - 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)

**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
Using Semaphores for Mutual Exclusion

- Use a binary semaphore for mutual exclusion
  ```
  Semaphore = new Semaphore(1);
  Semaphore=P(); // Critical Section; Semaphore=V();
  ```
- Using Semaphores for producer-consumer with bounded buffer
  ```
  Semaphore mutex;
  Semaphore fullBuffers;
  Semaphore emptyBuffers;
  ```

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();
}
```

Comparing code

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

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

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

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

CokeMachine::Remove() {  // Attempt to remove coke from the machine
    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
    (for example: Meta, Java (synchronized methods))
- Monitors also define a programming convention
  - Can be used in any language (C, C++, Java, ...)

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,重新 acquire 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**

```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++;
    notFull.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 T1 is waiting on condition x
  - Assume thread T2 is the monitor
  - Assume thread T2 calls x.signal, wake up T1
  - T2 continues, finishes
  - When T1 gets a chance to run, T2 takes over monitor, runs
  - T1 finishes, gives up monitor

**Example:**

```java
fn( )
    x.wait // T1 blocks
    fn( )
    x.signal // T2 blocks
    // T2 resumes
    lock->release();
    // T1 resumes
    lock->release();
```

**Hansen Monitors: Semantics**

- Hansen monitor semantics:
  - Assume thread T1 is 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, T2 takes over monitor, runs
  - T2 finishes, gives up monitor

**Example:**

```java
fn( )
    x.wait // T1 blocks
    fn( )
    x.signal // T2 blocks
    // T2 resumes
    lock->release();
    // T1 continues
    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

- **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

**Example:**

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

CokeMachine::Remove(){
    lock->acquire();
    Add coke to the machine;
    count--;
    notFull.signal();
    lock->release();
}
```

- **Benefits**:
  - Efficient implementation
  - Condition guarantees 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
  - 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
    - 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
- (Almost) never sleep() in your code
  - Use condition variables to synchronize

**Readers/Writers: A Complete Example**

**Motivation**
- Shared database access
  - 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

```cpp
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**

```cpp
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**

```cpp
public: database::read()
    startRead();
    Access database;
    doneRead();
}

private: database::startRead()
    lock.acquire();
    while (AR > 0) {
        WR ++;
        okToRead.wait(&lock);
        WR --;
    }
    AR ++;
    lock.release();
}

private: database::doneRead()
    lock.acquire();
    AR --;
    if (AR == 0 && WW > 0) {
        okToWrite.signal();
    }
    lock.release();
```
Solution Details: Writers

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

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

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