Communicating Between Threads

Josh Wilcox (jw14g24)

March 19, 2025

Table of Contents

Producer-Consumer Model

Producer-Consumer Problem

- A common concurrency pattern where:
  - **Producer** threads generate data/resources
  - **Consumer** threads use the produced data
  - Both interact through a **Shared Resource**
- Examples:
  - Web server (producer) and browser (consumer)
  - Data processing pipelines
  - Buffer/queue management systems

Potential Problems

- **Race Conditions:**
  - Producer adds product name without content, then thread switches
  - Consumer receives correct name but outdated content
  - Example: COMP1322 name with COMP1312 content
- **Resource Management Issues:**
  - **Buffer Overflow:** Producer creates data faster than consumer can process
  - **Buffer Underflow:** Consumer attempts to retrieve data when none exists
  - **Duplicate Consumption:** Consumer processes same data multiple times
- **Solution Requirements:**
  - Synchronization between producer and consumer
  - Mechanism to signal when buffer is full or empty
  - Atomic operations for adding/removing from shared resource

wait and notify

- **wait()**: Causes the current thread to pause execution and release any locks it holds until another thread issues a notification.
  - Must be called within a synchronized block or method
  - Releases the lock on the object during waiting
  - Can specify a timeout as an optional parameter
- **notify()**: Wakes up a single thread that is waiting on the object's monitor.
  - Must be called within a synchronized block or method
  - The JVM chooses which thread to wake up if multiple threads are waiting
  - Does not release the lock; the awakened thread will not run until the current thread exits the synchronized block
- **notifyAll()**: Wakes up all threads that are waiting on the object's monitor.
  - Ensures that all waiting threads are given a chance to proceed
  - Useful when multiple threads might be interested in a state change
  - More resource-intensive than notify(), but prevents missed signals
  - Like notify(), must be called from within a synchronized context

Monitors

Abstract Concept

3 Monitors
- Abstract Concept
- Monitors in Java

- A monitor is a synchronization construct that enables controlled access to shared resources
- Invented by C.A.R. Hoare and Per Brinch Hansen in the 1970s as an abstract concept
- Provides a higher-level abstraction for thread coordination than lower-level primitives
- Consists of:
  - **Mutex (mutual exclusion)**: Ensures only one thread can execute monitor code at a time
  - **Condition variables**: Allow threads to wait for specific conditions to be met
  - **Entry/exit protocol**: Handles the mechanics of thread queueing and scheduling
  - **Thread queue**: Where threads wait when they cannot enter the monitor or are waiting on conditions
- Key properties:
  - **Encapsulation**: Monitors encapsulate both data and the operations on that data
  - **Atomicity**: Operations within the monitor are executed atomically
  - **Signaling**: Allows threads to notify others when state changes occur
  - **Automatic locking**: The monitor implementation handles locking, not the programmer

Monitors in Java

- Every object in Java can act as a monitor
- Every object has a mutex lock and has a queue for waiting threads associated with it
- `synchronized(O){ code }` places code inside the monitor
- Every block of code surrounded by `synchronized(O){}` will be placed inside the monitor for `o` and protected by the mutex lock

Monitors in Java - **The wait() method**

- If a thread T calls `o.wait()`:
  - Thread T must be **executing inside** `o`
  - Thread T is then blocked and placed into the wait queue for the monitor
  - Thread T then releases the mutex lock to allow other threads to access the code block

Monitors in Java - **More on the notify() method**

- If a thread calls `o.notify()`:
    - Thread T must be **executing inside `o`**
    - Another arbitrary thread is selected from the wait queue
        - This thread is then unblocked
        - May not resume execution immediately as it needs to fetch the mutex lock
    - Thread T holds and keeps the mutex lock