CSE 120
Principles of Operating Systems

Spring 2023

Lecture 7: Condition Variables and Deadlock

Amy Ousterhout
Homework #1
  - Solutions on the course webpage
  - Grades on Canvas and Gradescope

Project 1
  - Ongoing, due 5/2

Homework #2
  - Ongoing, due 5/2

Midterm in class on 5/4
  - We will review in class on 5/2
  - Extra office hour 10-11 am on 5/2
Synchronization Primitives

- Sometimes we want semantics beyond just mutual exclusion
  - Wait for shared resources to become available
  - Allow multiple threads to generate different resources
  - Use certain conditions to decide when to enter a critical section

- Example synchronization problems
  - Producer-Consumer Problem
  - Readers-Writers Problem

- Semaphores
  - Synchronization variable with a non-negative integer value
  - `wait()`: waits for the semaphore to become positive and then decrements by 1
  - `signal()`: increments the semaphore by 1
Producer-Consumer with Locks and Sleep/Wake

- Naïve attempt to solve the Producer-Consumer problem
- Both threads sleep and never wake up
Producer-Consumer with Semaphores

- This approach works

```c
Producer
while (1) {
    produce an item
    wait(empty_count)
    acquire(lock);
    insert item in buffer
    count++;
    release(lock);
    signal(full_count)
}

Consumer
while (1) {
    wait(full_count)
    acquire(lock);
    remove item from buffer
    count--;
    release(lock);
    signal(empty_count)
    consume an item
}
```

count = 2
N = 8
Semaphores can be used to solve traditional synchronization problems:
- Enforce critical sections (mutual exclusion)
- Enable coordination between threads (scheduling)

But they have some drawbacks:
- No coordination between the semaphore and the controlled data
- Used for both critical sections and coordination - this can be confusing!
- Sometimes hard to use and prone to bugs

What can we do instead?
Today’s Outline

• Condition Variables
  ♦ Another way of synchronizing threads

• Monitors
  ♦ Leverage language support for synchronization

• Deadlock
  ♦ What can go wrong with concurrency?
  ♦ What can we do about it?
Producer-Consumer with Locks and Sleep/Wake

Producer

while (1) {
    produce an item
    if (count == N)
        sleep();
    acquire(lock);
    insert item in buffer
    count++;
    release(lock);
    if (count == 1)
        wakeup(consumer)
}

Consumer

while (1) {
    if (count == 0)
        sleep();
    acquire(lock);
    remove item from buffer
    count--;
    release(lock);
    if (count == N-1)
        wakeup(producer)
        consume an item
}

• Naïve attempt - is there another way to fix this?
• The problem was that a context switch could occur between if and sleep
• What if we moved the calls to acquire earlier?
Producer-Consumer with Locks and Sleep/Wake

**Producer**

```java
while (1) {
    produce an item
    acquire(lock);
    if (count == N)
        sleep();
    insert item in buffer
    count++;
    release(lock);
    if (count == 1)
        wakeup(consumer)
}
```

**Consumer**

```java
while (1) {
    acquire(lock);
    if (count == 0)
        sleep();
    remove item from buffer
    count--;
    release(lock);
    if (count == N-1)
        wakeup(producer)
    consume an item
}
```

- Move the calls to acquire earlier
- Does this work?
- No, a thread can sleep while holding the lock!
Condition Variables

• Goal: make it possible to go to sleep inside a critical section, by atomically releasing the lock at the same time we go to sleep

• A synchronization primitive that enables a queue of threads waiting for something inside a critical section

• Condition Variables support three operations:

  ♦ wait(): release the lock, go to sleep, wake up and re-acquire the lock when signaled (releasing the lock and going to sleep is atomic)
    » Also sleep() in Nachos

  ♦ signal(): wake up a waiting thread, if any
    » Also wake() in Nachos or notify()

  ♦ broadcast(): wake up all waiting threads, if any
    » Also wakeAll() in Nachos or notifyAll()
Condition Variables

• Used in conjunction with **locks**
  ♦ On creation, must specify which lock it is associated with
  ♦ Must hold the lock when invoking condition variable operations
  ♦ Lock will be atomically released and acquired during `wait()`

• Can be used to implement semaphores (and vice versa)

• Contrast with semaphore:
  ♦ No counting involved
  ♦ Memoryless
    » If `signal()` is called when no thread is waiting, it does nothing
  ♦ More intuitive to many people
  ♦ More commonly used in modern programming
Producer-Consumer with Condition Variables

- 2 condition variables to indicate when the buffer becomes not_full and not_empty
- Does this work?

```
Producer
while (1) {
    produce an item
    acquire(lock);
    if (count == N)
        wait(not_full);
    insert item in buffer
    count++;
    if (count == 1)
        signal(not_empty);
    release(lock);
}

Consumer
while (1) {
    acquire(lock);
    if (count == 0)
        wait(not_empty);
    remove item from buffer
    count--;
    if (count == N-1)
        signal(not_full);
    release(lock);
    consume item
}
```
Producer-Consumer with Condition Variables

Producer

```
while (1) {
    produce an item
    acquire(lock);
    if (count == N)
        wait(not_full);
    insert item in buffer
    count++;
    if (count == 1)
        signal(not_empty);
    release(lock);
}
```

Consumer

```
while (1) {
    acquire(lock);
    if (count == 0)
        wait(not_empty);
    remove item from buffer
    count--;
    if (count == N-1)
        signal(not_full);
    release(lock);
    consume item
}
```

- What could go wrong?
  - Producer 1 waits, consumer consumes, but then producer 2 produces!
Signal Semantics – Mesa vs. Hoare

• What happens when `signal()` is called?
  ♦ Only one thread can hold the lock at once
  ♦ Should the signaler or the woken thread run first?

• Mesa semantics
  ♦ Signaler keeps the lock and continues running
  ♦ Waiter is put on the ready queue
  ♦ Used by Nachos, most real operating systems
  ♦ Easier to implement

• Hoare semantics
  ♦ Signaler passes the lock to the waiter, waiter runs immediately
Signal Semantics – Mesa vs. Hoare

- **Mesa semantics**: the condition is not necessarily true when the signaled thread runs again
  - Returning from `wait()` is only a hint that something has changed
  - Must recheck the conditional case
- **Hoare semantics**: the condition is true when the signaled thread runs again
  - No need to recheck the conditional case

```c
Mesa semantics
while (count == N)
    wait(not_full);
```

```c
Hoare semantics
if (count == N)
    wait(not_full);
```
Producer-Consumer with Condition Variables

**Producer**

```java
while (1) {
    produce an item
    acquire(lock);
    while (count == N)
        wait(not_full);
    insert item in buffer
    count++;  
    if (count == 1)
        signal(not_empty);
    release(lock);
}
```

**Consumer**

```java
while (1) {
    acquire(lock);
    while (count == 0)
        wait(not_empty);
    remove item from buffer
    count--;
    if (count == N-1)
        signal(not_full);
    release(lock);
    consume item
}
```

- Recheck the condition by replacing the `if` with a `while`
- Does this work?
- Yes!
Common Pitfalls with Condition Variables #1

- CVs cannot be “tested”
- Need to maintain a separate flag
Common Pitfalls with Condition Variables #2

- Do not release the lock before using the CV
  - Using a CV requires that the thread holds the lock
- Purpose of a CV is to enable threads to block while in a critical section

```plaintext
acquire(lock);
...
release(lock);
wait(cond_var);
acquire(lock);
...
release(lock);
```
Common Pitfalls with Condition Variables #3

- Need to hold the lock while testing the condition
- The condition involves shared variables (e.g., `flag`) and is at risk of race conditions otherwise

```java
while (flag != true) {
    acquire(lock);
    wait(cond_var);
    release(lock);
}
```

```java
acquire(lock);
...
while (flag != true) {
    wait(cond_var);
}
...
release(lock);
```
Today’s Outline

• Condition Variables
  ♦ Another way of synchronizing threads

• Monitors
  ♦ Leverage language support for synchronization

• Deadlock
  ♦ What can go wrong with concurrency?
  ♦ What can we do about it?
Monitors

- A monitor is a programming language construct that controls access to shared data
  - Synchronization code is added by the compiler, enforced at runtime
- A monitor is a module that encapsulates:
  - Shared data structures
  - Procedures that operate on the shared data structures
  - Synchronization between concurrent threads that invoke the procedures
- A monitor protects its data from unstructured access
- It guarantees that threads accessing its data through its procedures interact only in legitimate ways
Monitor Semantics

- A monitor guarantees mutual exclusion
  - Only one thread can execute any monitor procedure at any time (the thread is “in the monitor”)
- Threads can use condition variables within a monitor
  - If a thread blocks within a monitor, another one can enter
**Producer-Consumer with a Monitor**

- Locking is implicit
  - Compiler adds the code
  - Equivalent to each procedure in the monitor calling `acquire()` on entry and `release()` on exit

```c
Monitor producer_consumer {
    Condition not_full;
    Condition not_empty;

    void put_resource() {
        ...
        wait(not_full);
        ...
        signal(not_empty);
    }

    void get_resource() {
        ...
        wait(not_empty);
        ...
        signal(not_full);
    }
}
```
Synchronization Primitives Summary

- **Locks**
  - Only provide mutual exclusion

- **Semaphores**
  - Provide mutual exclusion (binary semaphores)
  - Enable coordination between threads (counting semaphores)

- **Condition variables**
  - Synchronization point to wait for events
  - Used with locks or inside monitors

- **Monitors**
  - Synchronized execution using high-level language support
Synchronization in Real Life

- Example scenarios:
  - Crowded lab space  counting semaphore
    » Goal: only 8 people or fewer can be in the lab at once
  - Too much talking  lock or binary semaphore
    » Goal: avoid multiple people talking at the same time
  - Feed the picky eaters  condition variable + lock
    » Each kid wants to eat specific foods for dinner (e.g., kid 1: 1 cookie + 2 sandwiches, kid 2: 5 cookies + 1 carrot)
    » Cooks prepare food (e.g., 12 cookies, 30 carrots)
    » Goal: a kid can only take food when they can take their whole dinner at once

- Which synchronization primitive(s) would you use? Why?
- What does each primitive represent?
Today’s Outline

• Condition Variables
  ♦ Another way of synchronizing threads

• Monitors
  ♦ Leverage language support for synchronization

• Deadlock
  ♦ What can go wrong with concurrency?
  ♦ What can we do about it?
• Incorrect use of synchronization can block all threads
  ♦ We have seen examples already!
• Threads acquiring resources generate dependencies
  ♦ Locks, semaphores, etc. protect resources
• Example:
  ♦ T1 tries to acquire a resource that T2 holds and vice-versa
  ♦ They can never make progress
• This is a deadlock
Dining Philosophers’ Problem

- Dijkstra 1971
- Philosophers eat and think
- Eating requires two forks
  - Pick up one fork at a time
- Same number of forks as philosophers
- What can go wrong?
- What if they all pick up their right fork at the same time?
Deadlock Definition

- **Deadlock** is a problem that can arise:
  - When threads compete for access to limited resources
  - When threads are incorrectly synchronized

- **Definition:**
  - Deadlock exists among a set of threads if every thread is waiting for an event that can be caused only by another thread in the set

```
Thread 1
acquire(lock1);
...
acquire(lock2);

Thread 2
acquire(lock2);
...
acquire(lock1);
```

Deadlock!
Deadlock with Join

- How can we cause a deadlock with \texttt{join()}?

- Thread A
  ```
  ...
  B.join();
  ...
  ```

- Thread B
  ```
  ...
  A.join();
  ...
  ```
Conditions for Deadlock

• Deadlock can exist if an only if the following conditions hold simultaneously:
  ♦ Mutual exclusion: a resource is assigned to at most one thread at once
  ♦ Hold and wait: threads holding resources can request new resources while continuing to hold old resources
  ♦ No preemption: resources cannot be taken away once obtained
  ♦ Circular wait: one thread waits for another in a circular fashion

• Eliminating any condition eliminates deadlock!
We can illustrate deadlock using a resource allocation graph (RAG).

- Thread A holds resource R

- Thread B requests resource S

- If the graph has a cycle: deadlock may exist

- No cycles: no deadlock

Example:
- Thread 1 holds lock 1
- Thread 2 holds lock 2
- Each requests the others’ lock

Deadlock!
Resource Allocation Graph Examples

- Represent multiple resources with multiple boxes (e.g., with semaphores)
- Is there a deadlock?

![Graph Example 1](image1)
- Deadlock!

![Graph Example 2](image2)
- A cycle but no deadlock
Multi-Unit vs. Single-Unit Resources

- Multiple resources of some types
  - If the graph has a cycle: deadlock may exist
- Single resource of each type
  - If the graph has a cycle: deadlock exists
  - Useful for tracking locks
Strategies for Dealing with Deadlock

- **Ignore** the problem
  - Ostrich algorithm
- **Prevention**
  - Make it impossible for deadlock to happen
- **Avoidance**
  - Control allocation of resources
- **Detection and Recovery**
  - Look for a cycle in dependencies
- To be continued next class…
For next class…

- Read chapters 7 and 8