CMSC 330: Organization of Programming Languages

Threads
Classic Concurrency Problems
The Dining Philosophers Problem

- Philosophers either eat or think
- They must have two forks to eat
- Can only use forks on either side of their plate
- No talking!
- Avoid deadlock and starvation!
Bad Dining Philosophers Solution 1

• Philosophers all pick up the left fork first

• Deadlock!
  - all are holding the left fork and waiting for the right fork
Bad Dining Philosophers Solution 2

• Philosophers all pick up the left fork first
• Philosophers put down a fork after waiting for 5 minutes, then wait 5 minutes before picking it up again
• Starvation!
Possible Solutions

• The waiter solution
  – Third party arbiter (scheduler)
  – Each thread requests permission before acquiring a resource

• The resource hierarchy solution
  – Impose ordering on resources
  – Must obtain resources in order
  – Most practical solution
  – Sometimes hard to know in advance
Dining Philosophers Solution

• Number the philosophers
• Start by giving the fork to the philosopher with lower number. Initially, all forks are dirty.
• When a philosopher wants both forks, he sends a message to his neighbors
• When a philosopher with a fork receives a message: if his fork is clean, he keeps it, otherwise he cleans it and gives it up.
• After a philosopher eats, his forks are dirty. If a philosopher had requested his fork, he cleans it and sends it.
Dining Philosophers Example

Each philosopher begins with the forks shown.

All are dirty.
Dining Philosophers Example

Philosopher 2 sends a message to philosopher 1 that he wants his fork.

Their shared fork is dirty, so philosopher 1 cleans it and sends it.
Dining Philosophers Example

Philosopher 2 eats!

While he is eating philosopher 3 requests their shared fork.

Philosopher 2 is done eating, so his forks become dirty.
Philosopher 2 is done eating, so he honors philosopher 3’s request and cleans the fork and sends it.

Philosopher 3 eats!
Philosophers Implementation Needs

• Wait until notified about something by another philosopher
  – stay hungry until you have two forks
  – hold onto your fork until your neighbor needs it

• Send a message to a philosopher and have it processed at a later time
  – multiple philosophers can send messages to one
  – when philosopher done eating he should process all

… and here’s another problem with these needs…
Producer/Consumer Problem

• Suppose we are communicating with a shared variable
  – E.g., some kind of a fixed size buffer holding messages

• One thread *produces* input to the buffer
• One thread *consumes* data from the buffer

• Rules:
  – producer can’t add input to the buffer if it’s full
  – consumer can’t take input from the buffer if it’s empty
Producer / Consumer Idea

If the buffer is partially full, producer or consumer can run:

If the buffer is empty, only the producer can run:

If the buffer is full, only the consumer can run:
Needed Solution

• Need a way of having threads “wait” on a resource
• Also need a way to “notify” waiting threads when they can wake up

• Java 1.5 provides a very robust Condition mechanism to fill these needs
  – Slides are included for self-study
• However, Java 1.4 provided a simpler mechanism that we will discuss
Conditions (Java 1.5)

interface Lock {
    Condition newCondition(); ...
}
interface Condition {
    void await();
    void signalAll(); ...
}

- **Condition** created from a **Lock**
- **await** called with lock held
  - Releases the lock (on the fork or buffer)
    - But not any other locks held by this thread
  - Adds this thread to wait set for lock
  - Blocks the thread

when philosopher is waiting for a fork or consumer is waiting for non empty buffer
Conditions (Java 1.5)

interface Lock { Condition newCondition(); ... }
interface Condition {
    void await();
    void signalAll(); ... }

• **Condition** created from a **Lock**
  
  when philosopher is done eating
  or when buffer is non empty:

• **signalAll** called with lock held
  – Resumes all threads on lock’s wait set
  – Those threads must reacquire lock before continuing
    • (This is part of the function; you don’t need to do it explicitly)
Producer/Consumer Example

```java
Lock lock = new ReentrantLock();
Condition ready = lock.newCondition();
boolean bufferReady = false;
Object buffer;

void produce(Object o) {
    lock.lock();
    while (bufferReady) {
        ready.await();
    }
    buffer = o;
    bufferReady = true;
    ready.signalAll();
    lock.unlock();
}

Object consume() {
    lock.lock();
    while (!bufferReady) {
        ready.await();
    }
    Object o = buffer;
    bufferReady = false;
    ready.signalAll();
    lock.unlock();
}
```
Use This Design

• This is the right solution to the problem
  – Tempting to try to just use locks directly
  – Very hard to get right
  – Problems with other approaches often very subtle

... here are a few bad solutions...
Broken Producer/Consumer Example

```java
Lock lock = new ReentrantLock();
boolean valueReady = false;
Object value;

void produce(object o) {
    lock.lock();
    while (valueReady);
    value = o;
    valueReady = true;
    lock.unlock();
}

Object consume() {
    lock.lock();
    while (!valueReady);
    Object o = value;
    valueReady = false;
    lock.unlock();
}

Threads wait with lock held – no way to make progress
```
Lock lock = new ReentrantLock();
boolean valueReady = false;
Object value;

void produce(object o) {
    while (valueReady);
    lock.lock();
    value = o;
    valueReady = true;
    lock.unlock();
}

Object consume() {
    while (!valueReady);
    lock.lock();
    Object o = value;
    valueReady = false;
    lock.unlock();
}

valueReady accessed without a lock held – race condition
Broken Producer/Consumer Example

```java
Lock lock = new ReentrantLock();
Condition ready = lock.newCondition();
boolean valueReady = false;
Object value;

void produce(object o) {
    lock.lock();
    if (valueReady)
        ready.await();
    value = o;
    valueReady = true;
    ready.signalAll();
    lock.unlock();
}

Object consume() {
    lock.lock();
    if (!valueReady)
        ready.await();
    Object o = value;
    valueReady = false;
    ready.signalAll();
    lock.unlock();
}

what if there are multiple producers or consumers?
```
Why was it broken?

• Suppose you have 2 consumers, 1 producer
• Producer starts. valueReady set to true.
• Both consumers exit while loop and try to acquire lock.
• One consumer gets the lock and consumes the input.
• The next consumer is still able to get the lock.
  – ERROR!
More on the Condition Interface

interface Condition {
    void await();
    boolean await (long time, TimeUnit unit);
    void signal();
    void signalAll();
    ... }

• `await(t, u)` waits for time `t` and then gives up
  – Result indicates whether woken by signal or timeout
• `signal()` wakes up only one waiting thread
  – Tricky to use correctly
    • Have all waiters be equal, handle exceptions correctly
  – Highly recommended to just use `signalAll()`
Await and SignalAll Gotcha’s

• **await** *must* be in a loop
  – Don’t assume that when wait returns conditions are met

• Avoid holding other locks when waiting
  – **await** only gives up locks on the object you wait on
Wait and NotifyAll (Java 1.4)

- Recall that in Java 1.4, use synchronize on object to get associated lock

  object o

  o’s lock

  o’s wait set

- Objects also have an associated wait set
Wait and NotifyAll (cont’d)

- **o.wait()**  (same as await)
  - Must hold lock associated with `o`
  - Release that lock
    - And no other locks
  - Adds this thread to wait set for lock
  - Blocks the thread

- **o.notifyAll()**  (same as signalAll)
  - Must hold lock associated with `o`
  - Resumes all threads on lock’s wait set
  - Those threads must reacquire lock before continuing
    - (This is part of the function; you don’t need to do it explicitly)
public class ProducerConsumer {
    private boolean valueReady = false;
    private Object value;

    synchronized void produce(Object o) {
        while (valueReady) wait();
        value = o; valueReady = true;
        notifyAll();
    }

    synchronized Object consume() {
        while (!valueReady) wait();
        valueReady = false;
        Object o = value;
        notifyAll();
        return o;
    }
}
Aspects of Synchronization

• Atomicity
  – Locking to obtain mutual exclusion
  – What we most often think about

• Visibility
  – Ensuring that changes to object fields made in one thread are seen in other threads

• Ordering
  – Ensuring that you aren’t surprised by the order in which statements are executed
Guidelines for Programming w/Threads

• Synchronize access to shared data
• Don’t hold multiple locks at a time
  – Could cause deadlock
• Hold a lock for as little time as possible
  – Reduces blocking waiting for locks
• While holding a lock, don’t call a method you don’t understand
  – E.g., a method provided by someone else, especially if you can’t be sure what it locks
  – Corollary: document which locks a method acquires
Thread Cancellation

• Example scenarios: want to cancel thread
  – Whose processing the user no longer needs (i.e., she has hit the “cancel” button)
  – That computes a partial result and other threads have encountered errors, … etc.

• Java used to have `Thread.kill()`
  – But it and `Thread.stop()` are deprecated
  – Use `Thread.interrupt()` instead
Why No Thread.kill()?

• What if the thread is holding a lock when it is killed? The system could
  – Free the lock, but the data structure it is protecting might be now inconsistent
  – Keep the lock, but this could lead to deadlock

• A thread needs to perform its own cleanup
  – Use InterruptedException and isInterrupted() to discover when it should cancel
Thread.interrupt()

• Tries to wake up a thread
  – Sets the thread’s interrupted flag
  – Flag can be tested by calling
    • interrupted() method
      – Static, and clears the interrupt flag
    • isInterrupted() method
      – Instance, and does not clear the interrupt flag

• Won’t disturb the thread if it is working
  – Not asynchronous!
public class CancellableReader extends Thread {
    private FileInputStream dataFile;
    public void run() {
        try {
            while (!Thread.interrupted()) {
                try {
                    int c = dataFile.read();
                    if (c == -1) break;
                    else process(c);
                } catch (IOException ex) { break; }
            }
        } finally { // cleanup here }
    }
}

What if the thread is blocked on a lock or wait set, or sleeping when interrupted?

This could acquire locks, be on a wait set, etc.
InterruptedException

- Exception thrown if interrupted on certain ops
  - wait, await, sleep, join, and lockInterruptibly
  - Also thrown if call one of these with interrupt flag set
Responses to Interruption

• Early Return
  – Clean up and exit without producing errors
  – May require rollback or recovery

• Continuation (i.e., ignore interruption)
  – When it is too dangerous to stop
  – When partial actions cannot be backed out
  – When it doesn’t matter
Responses to Interruption (cont’d)

• Re-throw `InterruptedException`
  – When callers must be alerted on method return

• Throw a general failure exception
  – When interruption is a reason method may fail

• In general
  – Must reset invariants before cancelling
  – E.g., close file descriptors, notify other waiters, etc.
Handling InterruptedException

```
synchronized (this) {
    while (!ready) {
        try { wait(); } 
        catch (InterruptedException e) {
            // make shared state acceptable
            notifyAll();
            // cancel processing
            return;
        }
        // do whatever
    }
}
```