Process/thread Synchronization

Critical section. Mutual exclusion. Peterson's solution. Hardware solutions. Producer-consumer problem. Condition synchronization. Semaphores. Java synchronization.

OS4: 15/2/2016
Textbook (SGG): Ch. 6.1-6.4, 6.5.1-6.5.2, 6.6.1, 6.8.1-6.8.2, 6.8.4

Producer-Consumer Problem

- A producer puts work into a shared buffer
- A consumer takes work from the buffer
- The buffer has a fixed (finite) size, i.e., *bounded buffer*
- If producer and consumer run as separate processes (or threads), how can we *ensure* that their concurrent execution is correct?
  - Each process/thread makes *non-zero progress*
  - But otherwise, can't assume anything about their relative speed of execution

Producer action

```
while (count == BUFFER.SIZE)
   ; // do nothing

// add an item to the buffer
buffer[in] = item;
in = (in + 1) % BUFFER.SIZE;
++count;
```

Count: holds number of work items now in the buffer

Consumer action

```
while (count == 0)
   ; // do nothing

// remove an item from the
buffer item = buffer[out];
out = (out + 1) % BUFFER.SIZE;
--count;
```

Race Condition

- count++ could be implemented as
  register1 = count
  register1 = register1 + 1
  count = register1
- count-- could be implemented as
  register2 = count
  register2 = register2 - 1
  count = register2
- Execution may interleave as (initially, count = 5):

  T0: producer execute register1 = count {register1 = 5}
  T1: producer execute register1 = register1 + 1 {register1 = 6}
  T2: consumer execute register2 = count {register2 = 5}
  T3: consumer execute register2 = register2 - 1 {register2 = 4}
  T4: producer execute count = register1 {count = 6 }
  T5: consumer execute count = register2 {count = 4}

  Is execution correct? Why?

NB: each process has own registers (as usual); count is shared.
load, store, and arithmetic operations are *atomic.*

The Critical Section Problem

- Need to （ at least) guarantee *mutual exclusion* in updates to count variable
- Update needs to be protected in a *critical section* of code
  - Processes can’t be inside their critical sections at the same time

Solution to Critical-Section Problem

A really correct solution should satisfy *all* of …

1. Mutual Exclusion - If process P_i is executing in its critical section, then no other processes can be executing in their critical sections.

2. Progress - If no process is executing in its critical section and there exist some processes that wish to enter their critical section, then the selection of the processes that will enter the critical section next cannot be postponed indefinitely.

3. Bounded Waiting - A bound must exist on the number of times that other processes are allowed to enter their critical sections after a process has made a request to enter its critical section and before that request is granted.
   - Assume that each process executes at a nonzero speed
   - No assumption concerning relative speed of the processes

NB: each critical section is of finite length

Solution template for typical process

```
while (true) {

    entry section

        critical section

    exit section

        remainder section

}
```

A First Attempt

Process 0 runs:

```
while (true) {
    while (wantEnter[1])
        ;
    wantEnter[0] = true;
    // critical section
        …
    wantEnter[0] = false;
    // remainder section
        …
}
```

Assume load/store instructions are atomic

A Second Attempt

Process 0 runs:

```
while (true) {
    while (turn != 0)
        ;
    // critical section
        …
    turn = 1;
    // remainder section
        …
}
```

Assume load/store instructions are atomic

Peterson's Solution

- Works for two processes (can be generalized to N of them)
- Again, assume that load/store instructions are atomic, i.e., they can't be interrupted
  - This assumption is true for some computers, but not all
  - Otherwise, it's a purely *software* solution
- The two processes share two variables:
  - int turn;
  - boolean flag[2]
- The variable turn indicates whose turn it is to enter the critical section
- The flag array is used to indicate if a process is ready to enter the critical section
  - flag[i] = true means that process P_i is ready

Peterson's Solution for Process P_i

```
while (true) {
        flag[i] = true;
        turn = j;
        while (flag[j] && turn == j);
        critical section
        flag[i] = false;
        remainder section
}
```

Two processes: one for i = 0, the other for i = 1

Homework 4.1

- Is the solution in Attempt 1 correct? Explain your answer.
- Is the solution in Attempt 2 correct? Explain your answer.
- Argue that Peterson's solution is correct with respect to
  - Mutual exclusion
  - Progress
  - Bounded waiting
- Due 23/2/2016, midnite

Synchronization Hardware

- Many systems provide *hardware* support for critical section code
  - Solutions become easier with hardware support
- Uniprocessors – could disable interrupts
  - Currently running code would execute without preemption – why?
  - Doesn't work in general for multiprocessors – why?
- Many modern machines provide special *atomic hardware instructions*
  - Two common instructions
    - Test original value of memory word and set its value
    - Swap two memory words

Definition of Hardware Instructions

```
public class HardwareData
{
    private boolean value = false;

    public HardwareData(boolean value) {
       this.value = value;
    }

    public boolean get() {
       return value;
    }

    public void set(boolean newValue) {
       value = newValue;
    }

    public boolean getAndSet(boolean newValue) {
       boolean oldValue = this.get();
       this.set(newValue);

       return oldValue;
    }

   public void swap(HardwareData other) {
      boolean temp = this.get();

      this.set(other.get());
      other.set(temp);
    }
}
```

Assume each method is atomic due to hardware support

CS Solution using getAndSet

When thread T calls yield(), T remains runnable, but it yields control to CPU scheduler to pick the next thread to run

```
// lock is shared by all threads
HardwareData lock = new HardwareData(false);

while (true) {
    while (lock.getAndSet(true))
       Thread.yield();

    // critical section
    lock.set(false);
    // remainder section
}
```

Does this solution provide bounded waiting?

Solution using swap instruction

```
// lock is shared by all threads
HardwareData lock = new HardwareData(false);

// each thread has a local copy of key
HardwareData key = new HardwareData(true);

while (true) {
    key.set(true);

    do {
       lock.swap(key);
    }
    while (key.get() == true);

    // critical section
    lock.set(false);
    // remainder section
}
```

Semaphore

- Peterson's solution and the hardware-assisted solutions all require *busy waiting*
- Using *semaphore*, you can express a solution without busy waiting
  - Semaphore is a high-level synchronization primitive (easier to use)
  - Other primitives exist, but semaphore is as powerful as any
- Semaphore defines
  - One integer state variable
  - Two atomic operations on the variable: **acquire()** and **release()**

```
acquire() {
    while value <= 0
       ; // no-op
    value--;
}

    release() {
        value++;
    }
```

Two flavors of semaphore

- Binary semaphore – integer value can only be 0 or 1
  - It provides *mutual exclusion*
- Counting semaphore – integer value can be 0, 1, 2, 3, …
  - Useful for more general *condition synchronization* (e.g., buffer not empty for consumer)
- Note: *initialization* of the semaphore (1 in this example) is important:

```
Semaphore sem = new Semaphore(1);

sem.acquire();

   // critical section

sem.release();

   // remainder section
```

Useful to think of semaphore's integer state variable as count of how many resources are now available for taking

Java Example Using Semaphores

```
public class Worker implements Runnable
{
    private Semaphore sem;

    public Worker(Semaphore sem) {
       this.sem = sem;
    }

    public void run() {
       while (true) {
          sem.acquire();
          criticalSection();
          sem.release();
          remainderSection();
       }
    }
}
```

Java Example Using Semaphores

```
public class SemaphoreFactory
{
    public static void main(String args[]) {
       Semaphore sem = new Semaphore(1);
       Thread[] bees = new Thread[5];

       for (int i = 0; i < 5; i++)
          bees[i] = new Thread(new Worker(sem));
       for (int i = 0; i < 5; i++)
          bees[i].start();
    }
}
```

Really no busy waiting?

- True that our semaphore-based code (Slide 6.25) has no busy waiting
- But in fact, we're cheating! – Why?
- What's wrong with busy waiting?
  - Generally bad for uniprocessors
  - Could be good for multiprocessors, but only if process now inside critical section is about to leave it (e.g., we know that CS is short)
    - The busy wait is called a *spinlock*
    - In general, we can't say for sure that the CS is short (it depends on the application)

Semaphore implementation without busy waiting

- How? By integrating implementation with CPU scheduler: can *block* and *unblock* (or wake up) processes
  - If acquire() can't complete, block caller until semaphore becomes available
- Associate a waiting queue (of processes) with each semaphore
- Two operations:
  - block – change calling process's CPU scheduling state to waiting/blocked, and add it to the appropriate waiting queue
  - wakeup – remove a process from the waiting queue and change its CPU scheduling state to ready/runnable

Semaphore implementation without busy waiting (cont'd)

- Implementation of **acquire()**:

```
acquire(){
    value--;
    if (value < 0) {
        add this process to list
        block;
    }
}
```

- Implementation of **release()**:

```
release(){
    value++;
    if (value <= 0) {
        remove a process P from list
        wakeup(P);
    }
}
```

Activity 4.1: Semaphore implementation

- For semaphore to work, **acquire ()** and **release ()** must be atomic
  - i.e., they are critical sections
  - *Wait a minute*: we define semaphores to solve the CS problem, but their implementation now requires a solution to the CS problem!
- How do we escape from the circular logic?
  - i.e., What can you do to guarantee the atomicity of acquire/release without semaphore?
  - So, we use your answer above to solve the CS problem for acquire/release, but we use semaphore for other kinds of CS problems. What's special about acquire/release?

Activity 4.2: *Multiple* producers-consumers, one bounded buffer

- Assume that the shared buffer size is N
- Make it general: allow *multiple* producers and consumers
- Skeleton producer code (no synchronization)

```
public void produce(Work item) {
        // is “in” a shared variable in this problem?
        buffer[in] = item;
        in = (in + 1) % BUFFER_SIZE;
}
```

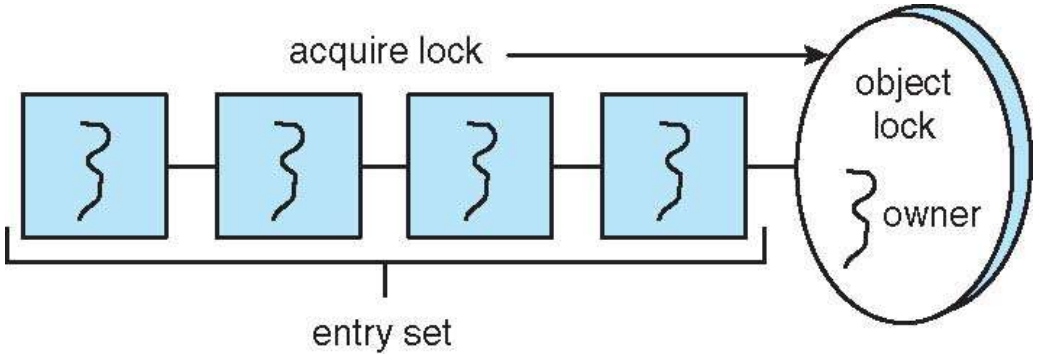
- Let’s design a solution using semaphores
  - What are the synchronization problems?
  - How many semaphores will you need?
  - How should you initialize these semaphores?
  - Now, write the produce (& consume) code w/ proper sychronization

Java Synchronization

- Java provides synchronization at the language-level.
- Each Java object has an associated lock.
- This lock is acquired by invoking a **synchronized** method.
- This lock is released when exiting the **synchronized** method.
- Threads waiting to acquire the object lock are placed in the **entry set** for the object lock.

Java Synchronization

- Each object has an associated **entry set**.



Condition synchronization in synchronized methods

- Synchronized insert() and remove() methods – what's wrong?

```
// Producers call this method
public synchronized void insert(E item) {
    while (count == BUFFER_SIZE)
       Thread.yield();

    buffer[in] = item;
    in = (in + 1) % BUFFER_SIZE;
    ++count;
}


// Consumers call this method
public synchronized E remove() {
    E item;

    while (count == 0)
       Thread.yield();

    item = buffer[out];
    out = (out + 1) % BUFFER_SIZE;
    --count;

    return item;
}
```

Condition synchronization by wait/notify()

- When a thread invokes **wait():**

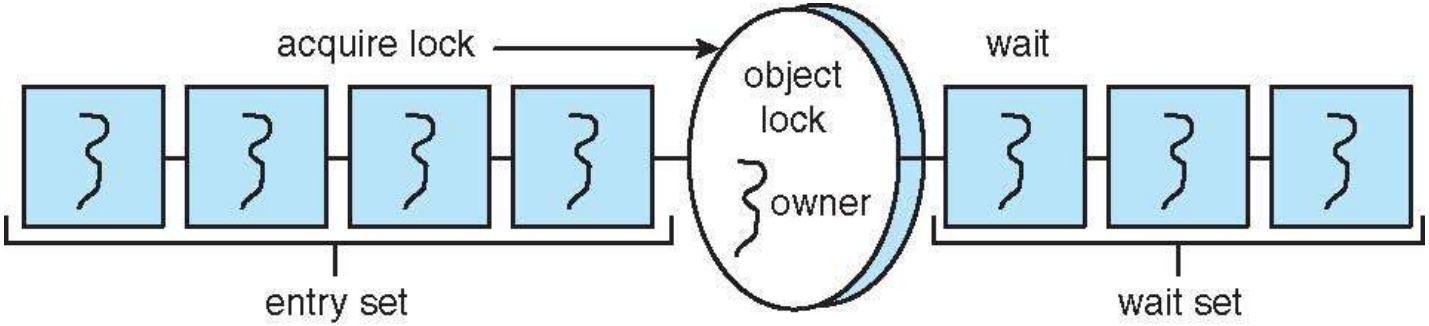
  1. The thread releases the object lock;
  2. The state of the thread is set to blocked;
  3. The thread is placed in the **wait set** for the object.

- When a thread invokes **notify()**:

  1. An arbitrary thread T from the wait set is selected;
  2. T is moved from the wait to the entry set;
  3. The state of T is set to runnable.

Java Synchronization

- Entry and wait sets
- Note that these sets are per *object*



Java Synchronization – wait/notify

- Corrected synchronized insert() method

```
// Producers call this method
public synchronized void insert(E item) {
    while (count == BUFFER_SIZE) {
       try {
          wait();
       }
       catch (InterruptedException e) { }
    }

    buffer[in] = item;
    in = (in + 1) % BUFFER_SIZE;
    ++count;

    notify();
}
```

Java Synchronization – wait/notify

- Corrected synchronized remove() method

```
// Consumers call this method
public synchronized E remove() {
    E item;

    while (count == 0) {
       try {
          wait();
       }
       catch (InterruptedException e) { }
    }

    item = buffer[out];
    out = (out + 1) % BUFFER_SIZE;
    --count;

    notify();

    return item;
}
```

Condition synchronization by notifyAll()

- Note that thread uses **wait()** to wait for a *condition* logically, but gets placed in a wait set that is per *object* (i.e., not condition)
- Similarly, another thread uses **notify()** to signal a condition logically, but **notify()** wakes up an arbitrary thread from the *object's* wait set
  - If there are more than one conditions associated with the object, **notify()** may wake up a wrong thread (one *not* waiting for the condition being notified)!
  - Solution:
    - Use **notifyAll()** to wake up *all* the threads in the wait set
    - When a thread returns from **wait()**, it *must* recheck the condition it was waiting for (see the while loop in Slides 6.63 and 6.64)
    - Can also use fine grained Java *condition variables*

Multiple conditions in one object

```
/**
 * myNumber is the number of the thread
 * that wishes to do some work
 */
public synchronized void doWork(int myNumber) {
   while (turn != myNumber) {
      try {
         wait();
      }
      catch (InterruptedException e) { }
   }

   // Do some work for awhile . . .

   /**
    * Finished working. Now indicate to the
    * next waiting thread that it is their
    * turn to do some work.
    */
   turn = (turn + 1) % 5;

   notify(); <-------------- notify() may not notify the correct thread!
}
```

Threads take turns, more than one threads waiting for their turns, but it's current turn for only one of the threads

Java 5 named condition variables

- Condition variable is created *explicitly* by first creating a reentrant lock, then invoking the lock's newCondition() method

```
Lock key = new ReentrantLock();
Condition condVar = key.newCondition();
```

  - A lock is *reentrant* is if it's safe to acquire the lock again by a caller already holding the lock
  - Note that a condition variable is always associated with a lock – why?
- Operations on condition variables: await() and signal() methods
- Explicit condition variables allow fine-grained condition synchronization; can solve the "threads taking turns" problem more cleanly …

Threads taking turns by fine grained condition variables

- doWork() method with condition variables

```
/**
 * myNumber is the number of the thread
 * that wishes to do some work
 */
public void doWork(int myNumber) {
   lock.lock();

   try {
      /**
       * If it's not my turn, then wait
       * until I'm signaled
       */
      if (myNumber != turn)
        condVars[myNumber].await();

      // Do some work for awhile . . .

      /**
       * Finished working. Now indicate to the
       * next waiting thread that it is their
       * turn to do some work.
       */

      turn = (turn + 1) % 5;
      condVars[turn].signal();
   }
   catch (InterruptedException ie) { }
   finally {
      lock.unlock();
   }
}
```

Activitiy 4.3: Release vs. Notify

- Consider the producer-consumer problem
  - In semaphore, when consumer creates an empty slot, you use the release() method
  - In Java synchronized method, when consumer creates an empty slot, you use the notify() method
- So release() and notify() look similar. But they also have a subtle difference: When customer creates an empty slot without any producers already waiting for the slot
  - What will be the effect of release()?
  - What will be the effect of notify()?
- Your semaphore solution to the producer-consumer problem doesn't need the count variable that's used in the solution on Slides 6.63 and 6.64. Why is it not needed?

Java Block Synchronization

- Rather than synchronize an entire method, **block synchronization** allows blocks of code to be declared as synchronized

```
Object mutexLock = new Object();
. . .
public void someMethod() {
    nonCriticalSection();

    synchronized(mutexLock) {
       criticalSection();
    }

    remainderSection();
}
```

Synchronized block is finer grained than synchronized method: allows more parallelism

Wait/notify in block synchronization

- Block synchronization using wait()/notify()

```
Object mutexLock = new Object();
. . .
synchronized(mutexLock) {
    try {
       mutexLock.wait();
    }
    catch (InterruptedException ie) { }
}

synchronized(mutexLock) {
    mutexLock.notify();
}
```