Process/thread Synchronization

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

OS4: 13/2/2018
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 a new item of work into a shared buffer
- A consumer takes an item of work from the buffer
- The buffer can store a fixed (finite) number of items, 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 variable 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
- Assume 1 CPU. 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. The instructions load, store, and arithmetic operations are *atomic* (i.e., can't be interrupted in the middle).

The Critical Section Problem

- Need to (at least) guarantee *mutual exclusion* in updates to **count** variable for code on Slides 6.3 and 6.4
  - If one process is in the middle of updating count, no other processes can be updating count at the same time
- Update needs to be protected in a *critical section* of code
  - A critical section is a segment of code (e.g., sequence of instructions that complete **count++** on Slide 6.5)
  - Processes can't be inside their critical sections at the same time (only *one* process can be)

*NB: We use **process** in our discussions; same ideas apply to **threads** as well*

Solution to Critical-Section Problem

A really correct solution should satisfy *all* these properties …

1. **Mutual Exclusion** - If process P_i is executing in its critical section (CS), then no other processes can be executing in their critical sections.
   1. *Safety* property: something bad (more than one processes in CS) can't happen
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.
   2. *Liveness* property: something good (a process entering CS) will eventually happen. *Mutual exclusion is trivial to satisfy without progress – why?*
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 (e.g., process P can run at same, much higher, or much lower speed than process Q)

NB: each critical section is of *finite length* (process will exit it after finite number of instructions – e.g., can't loop forever)

Solution template for typical process

```
while (true) {

        entry section

            critical section

        exit section

            remainder section

}
```

Our task: design the entry section and exit section

Attempt 1

Process 0 runs (the other process is Process 1, which will run a corresponding version of this code, i.e., with 0/1 exchanged):

Before Process i (i=0 or 1) enters, it needs to indicate its desire to enter

Process is "polite"

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

Assume load/store instructions are atomic

Attempt 2

Process 0 runs:

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

Let the processes take turns; **turn** variable indicates whose turn it is

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 (combined version of both Attempts 1 & 2):
  - int turn;
  - boolean flag[2] (similar to wantEnter[] in Attempt 1)
- The variable turn indicates whose turn it is to enter the critical section
- The flag array is used to indicate if a process wants to enter the critical section
  - flag[i] = true means that process P_i wants to enter

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
}
```

There are two processes i and j.

If i (or j) can't enter CS immediately, it must be waiting at **while** loop

Satisfies **mutual exclusion**: Assume i enters first and is now inside the CS. **Case 1**. flag[j] is false when i tests it in the while loop. In this case, before j tests its while loop, it must set turn to i, so that j must wait in the while loop until i exits or otherwise flag[i] must remain true and turn must remain i. **Case 2**. flag[j] is true when i tests it in the while loop. In this case, turn must be equal to i for i to exit the while loop. When j tests its while loop, j must wait until i exits or otherwise turn must remain i.

Satisfies **progress**: **Case 1**. Only i wants to enter, so that flag[j] = false. In this case, i must be able to exit the while loop since flag[j] = false. **Case 2**. Both i and j want to enter, so that flag[i] = flag[j] = true. In this case, the process whose id is equal to turn must be able to exit the while loop and enter.

Homework 4.1

- Is the solution in Attempt 1 (Slide 6.9) correct? Explain your answer.
- Is the solution in Attempt 2 (Slide 6.10) correct? Explain your answer.
- Argue that Peterson's solution is correct
  - We already showed it satisfies mutual exclusion and progress
  - Show that it satisfies bounded waiting also
- Due – 27 Feb 2019

Synchronization Hardware

- Many systems provide *hardware* support for critical section code
  - Solutions become easier with hardware support
- Mutual exclusion on uniprocessors – could disable interrupt on process j's entry (into CS) and restore interrupt on j's exit (from CS)
  - j must execute CS in *entirety* before any other process can have a chance to run – 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);
    }
}
```

*NB: **getAndSet** is guaranteed to be atomic by hardware, although it consists of multiple instructions; same for **swap***

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 (**value)** (intuitively, count how many units of a resource are currently available)
  - Two *atomic* operations on the variable: **acquire()** and **release()**

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

NB: **acquire** (or **release**) can't be interrupted in the middle), i.e., is atomic

Two basic synchronization problems: mutual exclusion + condition synchronization

- Binary semaphore – integer value can only be 0 or 1
  - It provides ***mutual exclusion*** (one unit of resource, which is taken when a process is inside CS)
- Counting semaphore – integer value can be 0, 1, 2, 3, …
  - Useful for more general ***condition synchronization*** (i.e., make sure the condition needed for a process to continue working is true; e.g., *buffer not empty* for consumer in bounded-buffer producer-consumer problem defined on Slide 6.2)
- Note: *initialization* of the semaphore (to 1 in this example) is important:

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

sem.acquire();

   // critical section

sem.release();

   // remainder section
```

NB: "Entry section" (see Slide 6.8) is the instruction sem.acquire();
What is "exit section"?

Java Example Using Semaphores

NB: (i) Worker implements runnable, so it can run as a separate thread.

(ii) When thread runs, it repeatedly enters critical section, then exits it to do something else (reminder section).

(iii) We want mutual exclusion for entering critical section.

```
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();
    }
}
```

Now, create 5 threads (bees[i] stores the i-th thread) and let them run

We want mutual exclusion between the threads (see Slide 6.20). So, *what kind of semaphore do we use? How is it initialized?*

Really no busy waiting?

- True that our semaphore-based code (Slide 6.19) has no busy waiting (no while loop)
- But in fact, we're cheating! – Why?
- What's wrong with busy waiting?
  - Generally bad for uniprocessors (executing instructions without doing anything useful)
  - Could be good for multiprocessors, but only if process now inside critical section is about to leave it (e.g., we know that critical section 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 semaphore implementation with CPU scheduler: can *block* and *unblock* (or wake up) processes
  - If **acquire**() can't complete (resource not available), block caller process P until semaphore becomes available (at which time wake up P)
- Associate a waiting queue (of processes) with each semaphore
- The two CPU scheduling 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);
    }
}
```

NB: In the above pseudo-code, "list" is queue of processes waiting to acquire the semaphore

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?
    - Use the solutions that do busy wait - what software and hardware solutions are available?
  - So, we use busy wait 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;
}
```

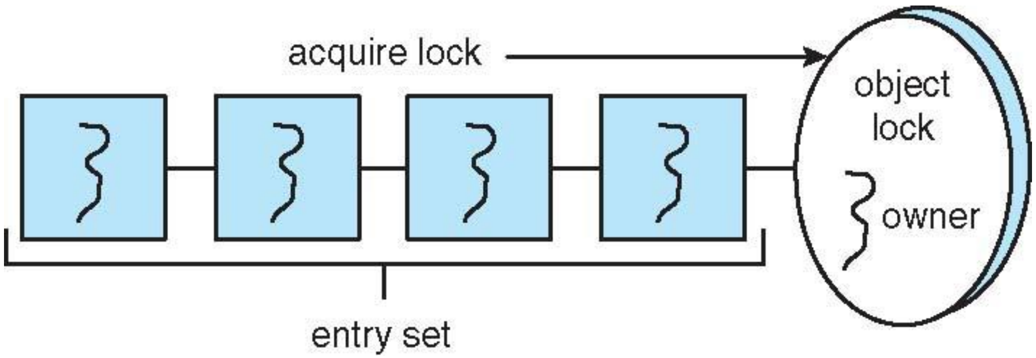
- Let's design a solution using semaphores
  - What are the synchronization problems?
  - How many semaphores will you need?
  - How should you initialize these semaphores?
  - Now, write the produce (& consume) code w/ proper sychronization

Java Synchronization

- Java provides synchronization at the language-level.
- Each Java object has an associated *binary* lock (i.e., lock is either taken or available).
- This lock is acquired by invoking a **synchronized** method.
- This lock is released when exiting the **synchronized** method.
- Hence, **mutual exclusion** is guaranteed for this object’s method – at most only *one* thread can be inside it at any time.
- 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**.



NB: entry set = queue of threads waiting to enter a (any) synchronized method for the object

Condition synchronization in synchronized methods

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

Producer thread holds object's lock in insert() when calling yield() because buffer is full; yield() won't give up the lock!

Can consumer thread call remove() to remove an item from the buffer?

```
// 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.

NB:

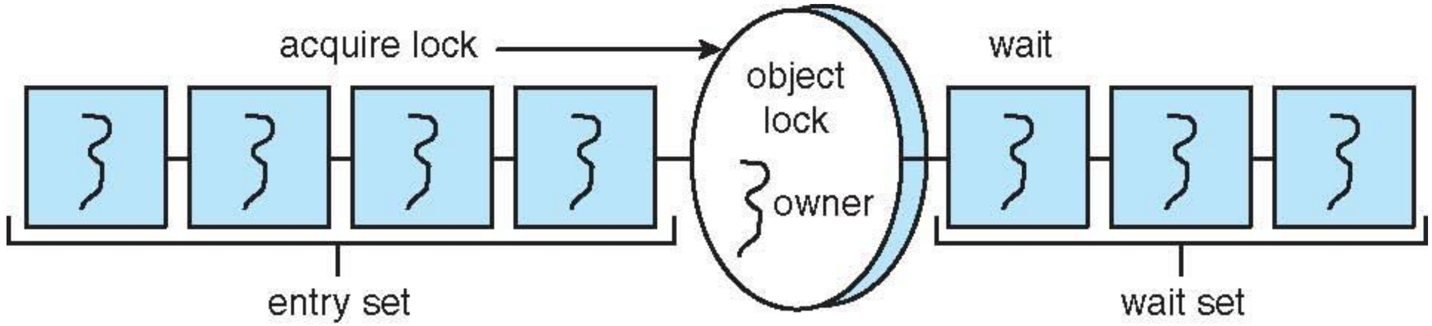
*(i) syncrhonized method solves **mutual exclusion** problem*

*(ii) wait()/notify() (together) solve **condition synchronization** problem*

(iii) Bounded buffer producer-consumer problem has two condition synchronization problems. First, for producer to wait for buffer to become non-full. Second, for consumer to wait for buffer to become non-empty.

Java Synchronization

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



NB:
(i) **Entry set** contains those threads waiting to enter synchronized method (i.e., mutual exclusion)
(ii) **Wait set** contains those threads waiting for a condition to become true (i.e., condition synchronization, e.g., *buffer not empty*)
(iii) Wait set is per *object* – threads join this list no matter what condition (e.g., *buffer not full* or *buffer not empty*) they are waiting for

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();
}
```

NB:
(i)This **insert**() method is called by producer
(ii)If producer finds buffer full, it has to wait for the condition that the buffer becomes not full; hence, it calls **wait**();
(iii)This **wait**() solution solves the problem with **yield**() on Slide 6.29. Why? *Hint*: Recall the definition of **wait**() on Slide 6.30. What does it do that **yield**() doesn't do?

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;
}
```

Similar code for consumer; note how the condition synchronization problem is again solved by wait/notify

Condition synchronization by notifyAll()

- Note that thread uses **wait()** to wait for a *condition* (e.g., buffer not empty) logically, but gets placed in a wait set that is per *object* (i.e., *not* per 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 (e.g., the producer-consumer problem) 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 (the **while loop** on Slides 6.32 or 6.33 keeps rechecking the condition until the condition is true) – if thread is waked up for wrong reason, then when it rechecks the condition, it'll find the condition still false and wait again
    - Can also use fine grained Java *named condition variables* (Slide 6.37)
      - These condition variables are per logical condition, not per object

Multiple conditions in one object

Five threads take turns; thread i should run when turn = i

Assume turn = 2 when notify() is called by thread 1

When notify() is called, threads 2, 3, 4, 5 are all blocked in wait()

notify() can pick *any* of threads 2, 3, 4, 5 to wake up; it may not be thread 2 (the thread that should be waked up)

Above problem could be solved by replacing notify() by notifyAll()

```
/**
 * 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!

Locking requirement for wait/notify

- When wait() is called on Slide 6.33, note that the lock for mutual exclusion must be being held by the caller
  - The lock is the object lock of the synchronized method

- Otherwise, this could happen for say producer thread X and consumer thread Y
  - 1: Y checks that buffer is empty
  - 2: X puts item into buffer, makes buffer non-empty, calls notify() (to wake up Y in case Y is waiting)
  - 3: Y calls wait() after checking that buffer is empty in Step 1
  - **Problem**! - X called notify() already in Step 2, but Y only calls wait() in Step 3. Y is blocked even if buffer is not empty; it might wait forever!
  - **Lesson**: Y can't be preempted by X between Step 1 and Step 3 – this is what the mutual exclusion lock would guarantee

Java 5 named condition variables

- **Named 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*
  - In the above code, note that the condition variable **condVar** is associated with the lock **key**; in general, this association makes sense because a thread always holds a lock when a condition is being signaled or waited for (see Slide 6.36)
- Operations on condition variables: await() and signal() methods
  - Instead of **wait**() and **notify**() for the per-object unnamed condition
- Explicit condition variables allow fine-grained condition synchronization; can solve the "threads taking turns" problem more cleanly (than the **notifyAll**() solution on Slide 6.35) …

Threads taking turns by fine grained condition variables

- doWork() method with condition variables

NB:
(i) 5 threads taking turns, as before

(ii) Now, create an array of 5 named condition variables

(iii) Thread i always waits for the i-th condition variable, for its turn to arrive

(iv) Thread i is responsible for signaling specifically the turn of the next thread only, i.e., thread (i + 1) % 5

```
/**
 * 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
  - With semaphore, when consumer creates an empty slot, you call the **release**() method (your solution to Slide 6.26)
  - With Java synchronized method, when consumer creates an empty slot, you call the **notify**() method (Slide 6.32)
- 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 6.3 and 6.4. 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();
}
```