Operating Systems

(1DT020 & 1TT802)

Lecture 7

Process synchronisation : Semaphores, Monitors, and Condition Variables (cont'd)

April 24, 2008

Léon Mugwaneza

http://www.it.uu.se/edu/course/homepage/os/vt08

Goals for Today

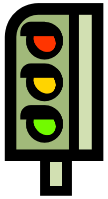
- Continue with Synchronization Abstractions
  - Semaphores, Monitors and condition variables
- Readers-Writers problem and solution
- Language Support for Synchronization

Note: Some slides and/or pictures in the following are adapted from slides ©2005 Silberschatz, Galvin, and Gagne, others from Kubiatowicz - CS162 ©UCB Fall 2007 (University of California at Berkeley)

Higher-level Primitives than Locks

- What is the right abstraction for synchronizing threads that share memory?
  - Want as high a level primitive as possible
- Good primitives and practices important!
  - Since execution is not entirely sequential, really hard to find bugs, since they happen rarely
  - UNIX is pretty stable now, but up until about mid-80s (10 years after started), systems running UNIX would crash every week or so – concurrency bugs
- Synchronization is a way of coordinating multiple concurrent activities that are using shared state
  - We will see a couple of ways of structuring the sharing

Semaphores



- Semaphores are a kind of generalized lock
  - First defined by Dijkstra in late 60s
  - Main synchronization primitive used in original UNIX
- Definition: a Semaphore has a non-negative integer value, a wait queue and supports the following two operations (apart from initialization):
  - P(): <u>an atomic operation</u> that does the following:

```
if value = 0 then sleep
else decrement value by 1
```

    - » Course book calls this operation wait()
  - V(): <u>an atomic operation</u> that does the following:

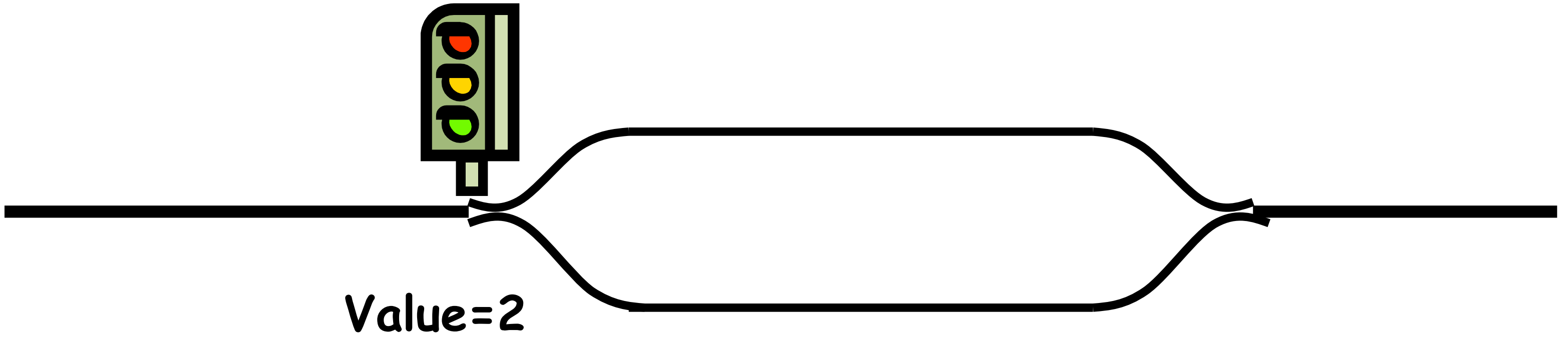
```
if there are any threads sleeping on that
   semaphore, wakeup 1 thread (at random)
else increment value by 1
```

    - » Course book calls this operation signal()
  - Note that P() stands for "*proberen*" (to test) and V() stands for "*verhogen*" (to increment) in Dutch
  - DOWN() sometimes used for P(), and UP() for V()

☛ Some implementations allow negative values (P always decrements value by one, and V always increments value by one)

Semaphores are not integers!

- Semaphores are like integers, except
  - No negative values
  - Only operations allowed are P and V – can’t read or write value, except to set it initially
  - Operations must be atomic
    - » Two P’s together can’t decrement value below zero
    - » Similarly, thread going to sleep in P won’t miss wakeup from V – even if they both happen at same time
- Semaphore from railway analogy
  - Here is a semaphore initialized to 2 for resource control:



Two uses of Semaphores

- Mutual Exclusion (initial value = 1)
  - Also called “Binary Semaphore”.
  - Can be used for mutual exclusion:

```
semaphore.P();
// Critical section goes here
semaphore.V();
```

- Scheduling Constraints (initial value = 0)
  - Locks are fine for mutual exclusion, but what if you want a thread to wait for something?
  - Example: suppose you had to implement ThreadJoin which must wait for thread to terminate:

```
Initial value of semaphore = 0
ThreadJoin {
   semaphore.P();
}
ThreadFinish {
   semaphore.V();
}
```

- What if initial value > 1?
  - Counting semaphore : consider a resource with N copies
    - » request a copy using P(), release copy using V()
    - » Scheduling constraints on resource utilization

Producer-consumer with a bounded buffer

- Problem Definition
  - Producer puts things into a shared buffer
  - Consumer takes them out
  - Need synchronization to coordinate producer/consumer
- Don’t want producer and consumer to have to work in lockstep, so put a fixed-size buffer between them
  - Need to synchronize access to this buffer
  - Producer needs to wait if buffer is full
  - Consumer needs to wait if buffer is empty
- Example 1: GCC compiler
  - cpp | cc1 | cc2 | as | ld
- Example 2: Coke machine
  - Producer can put limited number of cokes in machine
  - Consumer can’t take cokes out if machine is empty

Correctness constraints for solution

- Correctness Constraints:
  - Consumer must wait for producer to fill buffers, if none full (scheduling constraint)
  - Producer must wait for consumer to empty buffers, if all full (scheduling constraint)
  - Only one thread can manipulate buffer queue at a time (mutual exclusion)
- Remember why we need mutual exclusion
  - Because computers are “not very clever”
  - Imagine if in real life: the delivery person is filling the machine and somebody comes up and tries to stick their money into the machine
- General rule of thumb: Use a separate semaphore for each constraint
  - `Semaphore fullBuffers; // consumer's constraint`
  - `Semaphore emptyBuffers;// producer's constraint`
  - `Semaphore mutex;        // mutual exclusion`

Full Solution to Bounded Buffer

```
Semaphore fullBuffer = 0;  // Initially, no coke
Semaphore emptyBuffers = num;  // Initially, num empty slots
Semaphore mutex = 1;     // No one using machine

Producer(item) {
 While(True){
   do something else;    // incuding producing item
   emptyBuffers.P();    // Wait until space
   mutex.P();          // Wait until buffer free
   Enqueue(item);
   mutex.V();
   fullBuffers.V();   // Tell consumers there is more coke
 }
}

Consumer() {
 While(True){
   fullBuffers.P();    // Check if there's a coke
   mutex.P();          // Wait until machine free
   item = Dequeue();
   mutex.V();
   emptyBuffers.V();   // tell producer a slot is free
   do something else;  // including using item
  }
}
```

Discussion about Solution

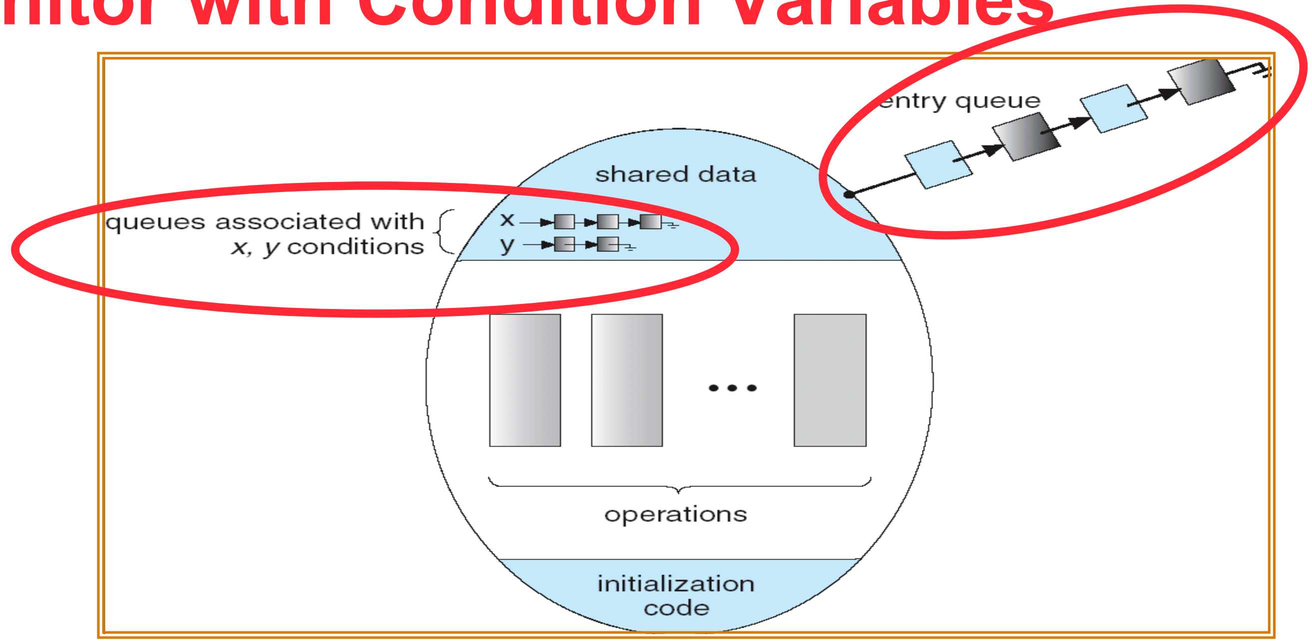
- Why asymmetry?
  - Producer does: `emptyBuffer.P(), fullBuffer.V()`
  - Consumer does: `fullBuffer.P(), emptyBuffer.V()`
- Is order of P's important?
  - Yes! Can cause deadlock
- Is order of V's important?
  - No, except that it might affect scheduling efficiency
- What if we have 2 producers or 2 consumers?
  - Do we need to change anything?

Motivation for Monitors and Condition Variables

- **Semaphores are a huge step up; just think of trying to do the bounded buffer with only loads and stores**
  - **Problem is that semaphores are dual purpose:**
    - **They are used for both mutex and scheduling constraints**
    - **Example: the fact that flipping of P's in bounded buffer gives deadlock is not immediately obvious. How do you prove correctness to someone?**

- **Cleaner idea: Use *locks* for mutual exclusion and *condition variables* for scheduling constraints**
- **Monitor: zero or more condition variables for managing concurrent access to shared data, together with operations that are guaranteed to be mutual exclusive**
  - **Monitors are language constructs (programming paradigms)**
  - **Some languages like Java provide this natively**
  - **Most others use actual locks and condition variables**

Monitor with Condition Variables



- **Lock**: the lock provides mutual exclusion to shared data
  - Always acquire before accessing shared data structure
  - Always release after finishing with shared data
  - Lock initially free
- **Condition Variable**: a queue of threads waiting for something *inside* a critical section
  - Two operations on conditions : condition.wait() and condition.signal()
  - Key idea: make it possible to go to sleep inside critical section by atomically releasing lock at time we go to sleep
  - Contrast to semaphores: Can’t wait inside critical section

Simple Monitor Example (version 1)

- Here is an (infinite) synchronized queue

```
Lock lock;
Queue queue;


AddToQueue(item) {
   lock.Acquire();          // Lock shared data
   queue.enqueue(item);     // Add item
   lock.Release();          // Release Lock
}


RemoveFromQueue() {
   lock.Acquire();          // Lock shared data
   item = queue.dequeue();// Get next item or null
   lock.Release();          // Release Lock
   return(item);            // Might return null
}
```

Condition Variables

- How do we change the RemoveFromQueue() routine to wait until something is on the queue?
  - Could do this by keeping a count of the number of things on the queue (with semaphores), but error prone
- **Condition Variable**: a queue of threads waiting for something *inside* a critical section
  - Key idea: allow sleeping inside critical section by atomically releasing lock at time we go to sleep
- Operations:
  - `Wait()`: Atomically release lock and go to sleep. Re-acquire lock later, before returning.
  - `Signal()`: Wake up one waiter, if any
  - Note some monitor definitions have a 3rd operation :
    - `Broadcast()`: Wake up all waiters
- Rule: Must hold lock when doing condition variable operations

Complete Monitor Example (with condition variable)

- Here is an (infinite) synchronized queue

```
Lock lock;
Condition dataready;
Queue queue;

AddToQueue(item) {
   lock.Acquire();              // Get Lock
   queue.enqueue(item);         // Add item
   dataready.signal();          // Signal any waiters
   lock.Release();              // Release Lock
}

RemoveFromQueue() {
   lock.Acquire();              // Get Lock
   while (queue.isEmpty()) {
      dataready.wait(&lock); // If nothing, sleep
   }
   item = queue.dequeue();      // Get next item
   lock.Release();              // Release Lock
   return(item);
}
```

Mesa vs. Hoare monitors

- **Need to be careful about precise definition of signal and wait. Consider a piece of our dequeue code:**

```
while (queue.isEmpty()) {
   dataready.wait(&lock); // If nothing, sleep
}
item = queue.dequeue();// Get next item
```

  - **Why didn't we do this?**

```
if (queue.isEmpty()) {
   dataready.wait(&lock); // If nothing, sleep
}
item = queue.dequeue();// Get next item
```

- **Answer: depends on the type of scheduling**
  - **Hoare-style (most textbooks):**
    - » **Signaler gives lock, CPU to waiter; waiter runs immediately**
    - » **Waiter gives up lock, processor back to signaler when it exits critical section or if it waits again**
  - **Mesa-style (most real operating systems):**
    - » **Signaler keeps lock and processor**
    - » **Waiter placed on ready queue with no special priority**
    - » **Practically, need to check condition again after wait**

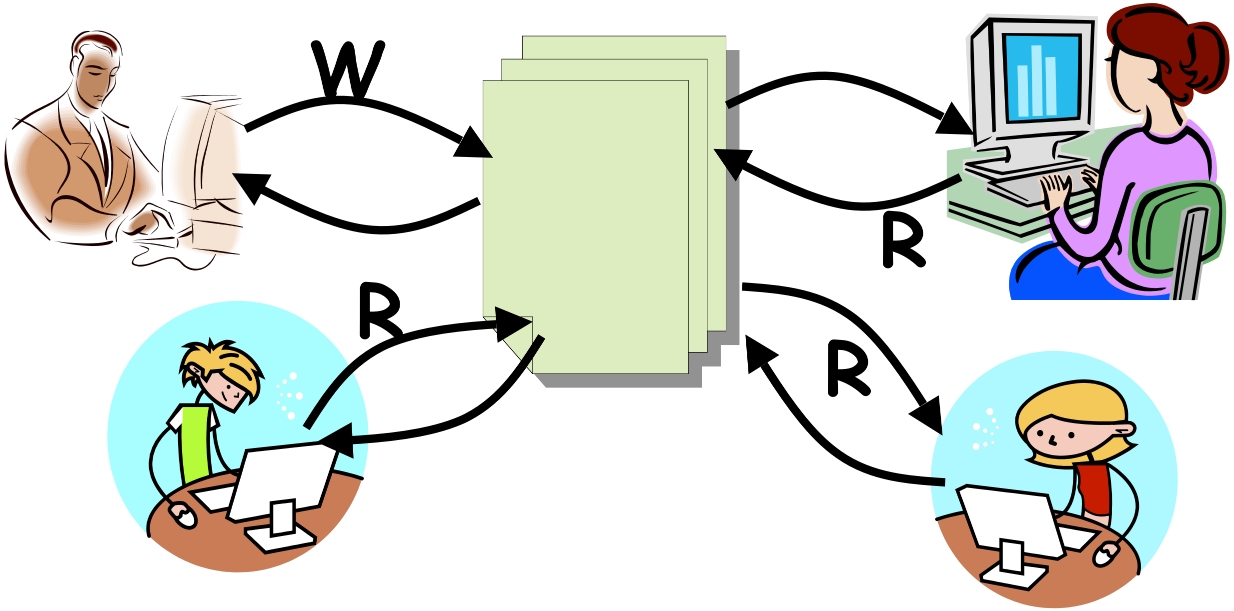
Monitors are language constructs

- **Programmer does not have to bother about lock :**

```
Monitor queueMonitor{
    Condition dataready;
    Queue queue;
    //init{…}
    // internal procedures (do not use cond. var.)
    // AddToQueue & RemoveFromQueue are external ops
    AddToQueue(item) {
       queue.enqueue(item);          // Add item
       dataready.signal();           // Signal any waiters
    }
    RemoveFromQueue() {
       while (queue.isEmpty()) {
          dataready.wait(); // If nothing, sleep
       }
       item = queue.dequeue();       // Get next item
       return(item);
    }
 } // end Monitor queueMonitor
```

☛ **lock, and system call to lock.Acquire() and lock.Release() will be inserted by the compiler**

Readers/Writers Problem



- **Motivation: Consider a shared database**
  - **Two classes of users:**
    - **Readers – never modify database**
    - **Writers – read and modify database**
  - **Is using a single lock on the whole database sufficient?**
    - **Like to have many readers at the same time**
    - **Only one writer at a time**

Basic Readers/Writers Solution

- **Correctness Constraints:**
  - Readers can access database when no writers
  - Writers can access database when no readers or writers
  - Only one thread manipulates state variables at a time
- **Basic structure of a solution:**
  - `Reader()`

    ```
    Wait until no writers
    Access data base
    Check out – wake up a waiting writer
    ```
  - `Writer()`

    ```
    Wait until no active readers or writers
    Access database
    Check out – wake up waiting readers or writer
    ```
  - Monitor DataBase
    - » 4 external procedures :
      - BeginRead, EndRead, BeginWrite, EndWrite
    - » State variables (Protected inside monitor)
      - int AR: Number of active readers; initially = 0
      - int WR: Number of waiting readers; initially = 0
      - int AW: Number of active writers; initially = 0
      - int WW: Number of waiting writers; initially = 0
      - Condition okToRead = NIL
      - Condition okToWrite = NIL

Code for Readers and Writers

```
Reader() {
  DataBase.BeginRead()
  // Now we are active!
  // Perform actual read-only access
  AccessDatabase(ReadOnly);
  DataBase.EndRead();
}

Writer() {
  DataBase.BeginWrite()
  // Now we are active!
  // Perform actual read/write access
  AccessDatabase(ReadWrite);
  DataBase.EndWrite();
}
```

DataBase Monitor’s operations

```
BeginRead() {
   while ((AW + WW) > 0) {        // -Is it safe to read?
      WR++;                       //   -No. Writers exist
      okToRead.wait();            //     ->Sleep on cond var
      WR--; // No longer waiting
   }
   AR++; // Now we are active!
 }
```

```
EndRead(){
   AR--;    // No longer active
   if (AR == 0 && WW > 0)         // No other active readers
      okToWrite.signal();         // Wake up one writer
}
```

```
  BeginWrite() {
     while ((AW + AR) > 0) {        // -Is it safe to write?
        WW++;                       //    -No. Active users exist
        okToWrite.wait();           //      -> Sleep on cond var
        WW--; // No longer waiting
     }
     AW++;    // Now we are active!
  }
```

```
 EndWrite() {
     AW--;    // No longer active
     if (WW > 0){                  // Give priority to writers
        okToWrite.signal();        // Wake up one writer
     } else if (WR > 0) {          // Otherwise, wake reader
        okToRead.broadcast();      // Wake all readers
     }
  }
```

Simulation of Readers/Writers solution

- Consider the following sequence of operators:
  - R1, R2, W1, R3
- On entry, each reader checks the following:

```
while ((AW + WW) > 0) { // Is it safe to read?
   WR++;                // No. Writers exist
   okToRead.wait();     // Sleep on cond var
   WR--;                // No longer waiting
}
AR++;                   // Now we are active!
```

- First, R1 comes along:
  AR = 1, WR = 0, AW = 0, WW = 0
- Next, R2 comes along:
  AR = 2, WR = 0, AW = 0, WW = 0
- Now, readers may take a while to access database
  - Situation: Locks released
  - Only AR is non-zero

Simulation(2)

- **Next, W1 comes along:**

```
while ((AW + AR) > 0) { // Is it safe to write?
   WW++;                // No. Active users exist
   okToWrite.wait();    // Sleep on cond var
   WW--;                // No longer waiting
}
AW++;
```

- **Can't start because of readers, so go to sleep:**

  AR = 2, WR = 0, AW = 0, WW = 1

- **Finally, R3 comes along:**
  AR = 2, WR = 1, AW = 0, WW = 1
- **Now, say that R2 finishes before R1:**
  AR = 1, WR = 1, AW = 0, WW = 1
- **Finally, last of first two readers (R1) finishes and wakes up writer:**

```
if (AR == 0 && WW > 0)  // No other active readers
   okToWrite.signal();  // Wake up one writer
```

Simulation(3)

- When writer wakes up, get:
  AR = 0, WR = 1, AW = 1, WW = 0
- Then, when writer finishes:

```
if (WW > 0){              // Give priority to writers
   okToWrite.signal();// Wake up one writer
} else if (WR > 0) { // Otherwise, wake reader
   okToRead.broadcast(); // Wake all readers
}
```

  - Writer wakes up reader, so get:

    AR = 1, WR = 0, AW = 0, WW = 0
- When reader completes, we are finished

Questions

- Can readers starve? Consider BeginRead() code:

```
while ((AW + WW) > 0) {        // Is it safe to read?
   WR++;                       // No. Writers exist
   okToRead.wait();            // Sleep on cond var
   WR--;                       // No longer waiting
}
AR++;                          // Now we are active!
```

- What if we erase the condition check in EndRead()?

```
AR--;                          // No longer active
if (AR == 0 && WW > 0)         // No other active readers
   okToWrite.signal();         // Wake up one writer
```

- Further, what if we turn the signal() into broadcast()

```
AR--;                          // No longer active
okToWrite.broadcast();         // Wake up all writers
```

- Finally, what if we use only one condition variable (call it "`okToContinue`") instead of two separate ones?
  - Both readers and writers sleep on this variable
  - Must use broadcast() instead of signal()

Can we construct Monitors from Semaphores?

- Locking aspect is easy: Just use a mutex
- Can we implement condition variables this way?

```
Wait()   { semaphore.P(); }
Signal() { semaphore.V(); }
```

- Does this work better?

```
Wait(Lock lock) {
   lock.Release();
   semaphore.P();
   lock.Acquire();
}
```

Construction of Monitors from Semaphores (con't)

- Problem with previous try:
  - P and V are commutative – result is the same no matter what order they occur
  - Condition variables are NOT commutative
- Does this fix the problem?

```
Wait(Lock lock) {
    lock.Release();
    semaphore.P();
    lock.Acquire();
}
Signal() {
    if semaphore queue is not empty
        semaphore.V();
}
```

  - Not legal to look at contents of semaphore queue
  - There is a race condition – signaler can slip in after lock release and before waiter executes semaphore.P()
- It is actually possible to do this correctly
  - Complex solution for Hoare scheduling in book
  - Can you come up with simpler Mesa-scheduled solution?

Monitor Conclusion

- **Monitors represent the logic of the program**
  - **Wait if necessary**
  - **Signal when change something so any waiting threads can proceed**
- **Basic structure of monitor-based program:**

`Use monitor procedure` } Check and/or update state variables
Wait if necessary

`Do something so no need to wait`

`Use monitor procedure` } Check and/or update state variables

Java Language Support for Synchronization

- **Java has explicit support for threads and thread synchronization**
- **Bank Account example:**

```
class Account {
  private int balance;
  // object constructor
  public Account (int initialBalance) {
    balance = initialBalance;
  }
  public synchronized int getBalance() {
    return balance;
  }
  public synchronized void deposit(int amount) {
    balance += amount;
  }
}
```

  - **Every object has an associated lock which gets automatically acquired and released on entry and exit from a *synchronized* method.**

Java Language Support for Synchronization (con't)

- Java also has *synchronized* statements:

```
synchronized (object) {
    …
}
```

  - Since every Java object has an associated lock, this type of statement acquires and releases the object's lock on entry and exit of the body
  - Works properly even with exceptions:

```
synchronized (object) {
  …
  DoFoo();
  …
}
void DoFoo() {
  throw errException;
}
```

Java Language Support for Synchronization (con’t 2)

- In addition to a lock, every object has a single condition variable associated with it
  - How to wait inside a synchronization method or block:
    - `void wait(long timeout); // Wait for timeout`
    - `void wait(long timeout, int nanoseconds); //variant`
    - `void wait();`
  - How to signal in a synchronized method or block:
    - `void notify();  // wakes up oldest waiter`
    - `void notifyAll(); // like broadcast, wakes everyone`
  - Condition variables can wait for a bounded length of time. This is useful for handling exception cases:

```
t1 = time.now();
while (!ATMRequest()) {
   wait (CHECKPERIOD);
   t2 = time.new();
   if (t2 – t1 > LONG_TIME) checkMachine();
}
```

  - Not all Java VMs equivalent!
    - Different scheduling policies, not necessarily preemptive!

Summary

- Semaphores : a non-negative integer value and queue with following operations:
  - Only time can set integer directly is at initialization time
  - P(): an atomic operation that waits for semaphore to become positive, then decrements it by 1 (Think of this as the wait() operation)
  - V(): an atomic operation that increments the semaphore by 1, waking up a waiting P, if any (This of this as the signal() operation)
- Monitors: A lock plus one or more condition variables
  - State variables and mutually exclusive operations
  - Use condition variables to wait inside critical section
    - » Three Operations: `Wait()`, `Signal()`, and `Broadcast()`
- Readers/Writers
  - Readers can access database when no writers
  - Writers can access database when no readers
  - Solution using a monitor
- Language support for synchronization:
  - Java provides synchronized keyword and one condition-variable per object (with `wait()` and `notify()`)