Parallel Programming

Recitation Session 7

Thomas Weibel <weibelt@ethz.ch>

Laboratory for Software Technology,
Swiss Federal Institute of Technology Zürich

April 22, 2010

Executive Summary

- Determining when a thread has finished
- The Volatile Returns
- Solution to the last assignment
- Semaphores
- Implementing Monitors with Semaphores
- `wait()`, `notify()`, `notifyAll()`
- Hints for assignment 7

Outline

isAlive()

```
// Create and start a thread
Thread thread = new MyThread();
thread.start();

// Check if the thread has finished
// in a non-blocking way
if (thread.isAlive()) {
  // Thread has not finished
} else {
  // Finished
}
```

join(delayMillis)

```java
// Wait for the thread to finish but don't
// wait longer than a specified time
long delayMillis = 5000; // 5 seconds
try {
  thread.join(delayMillis);
  if (thread.isAlive()) {
    // Timeout occurred,
    // thread has not finished
  } else {
    // Finished
  }
} catch (InterruptedException e) {
  // Thread was interrupted
}
```

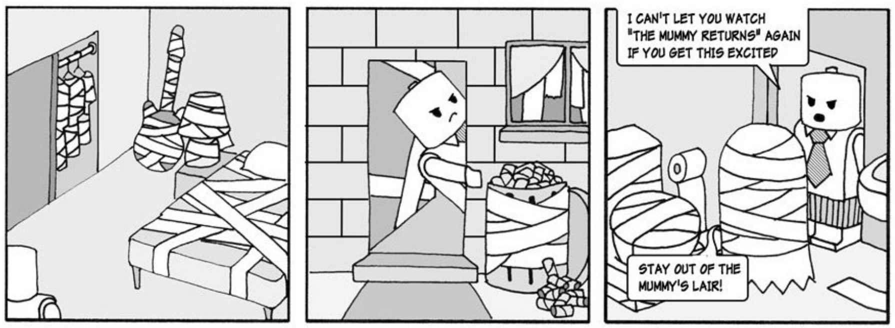

join()

```java
// Wait indefinitely for the thread to finish
try {
  thread.join();
  // Finished
} catch (InterruptedException e) {
  // Thread was interrupted
}
```


Outline

The Volatile Returns

- Volatile variables are not cached in registers or in caches where they are hidden from other processors
- A read of a volatile variable always returns the most recent write by any thread



Source: `http://thescope.ca/comics/everybodycheerup/the-mummy-returns`

Example

```
public class ShutdownDriver {
  boolean shutdown = false;

  public static void main(String[] args) throws InterruptedException {
    ShutdownDriver driver = new ShutdownDriver();
    new BusyTask(driver).start();
    Thread.sleep(2000);
    driver.shutdown = true;
  }
}

public class BusyTask extends Thread {
  private ShutdownDriver driver;

  public BusyTask(ShutdownDriver driver) {
    this.driver = driver;
  }

  public void run() {
    while (!driver.shutdown) {}
    System.out.println("Busy task stopped!");
  }
}
```


Example: Volatile

```
public class ShutdownDriver {
  volatile boolean shutdown = false;

  public static void main(String[] args) throws InterruptedException {
    ShutdownDriver driver = new ShutdownDriver();
    new BusyTask(driver).start();
    Thread.sleep(2000);
    driver.shutdown = true;
  }
}

public class BusyTask extends Thread {
  private ShutdownDriver driver;

  public BusyTask(ShutdownDriver driver) {
    this.driver = driver;
  }

  public void run() {
    while (!driver.shutdown) {}
    System.out.println("Busy task stopped!");
  }
}
```


Synchronization

Reading

Reading a volatile field is like acquiring a lock: The working memory is invalidated and the volatile field's current value is reread from memory

Writing

Writing a volatile field is like releasing a lock: the volatile field is immediately written back to memory

Limitations

- Although reading and writing a volatile field has the same effect on memory consistency as acquiring and releasing a lock, multiple reads and writes are not atomic
- For example, if `x` is a volatile variable, the expression `x++` will not necessarily increment `x` if concurrent threads can modify `x`
- One common usage pattern for volatile variables occurs when a field is read by multiple threads, but only written by one

No Atomicity!

`volatile` alone is not strong enough to implement a counter, some form of mutual exclusion is needed as well

Outline

1. Determining when a Thread has finished
2. Volatile
3. **Last Assignment**
4. Review of Semaphores
5. Semaphore Implementation of Monitors
6. Assignment 7

Code

```
A1 // non-critical section
A2 turn0.flag = 0;
A3 while(true)
     if(turn1.flag == 1)
       break;
A4   turn0.flag = 1;
A5   turn0.flag = 0;
   }
A6 // critical section
A7 turn0.flag = 1;
```

```
B1 // non-critical section
B2 turn1.flag = 0;
B3 while(true) {
     if(turn0.flag == 1)
       break;
B4   turn1.flag = 1;
B5   turn1.flag = 0;
   }
B6 // critical section
B7 turn1.flag = 1;
```


Invariants

1. `at(A6)` $\rightarrow$ `turn0.flag == 0`
2. `at(B6)` $\rightarrow$ `turn1.flag == 0`
3. `not [at(A6) AND at(B6)]`

We use the notation "`at(S)`" to indicate that execution is "at statement (location) `S`" $\Rightarrow$ all previous statements have executed while `S` has not yet started to execute

Proof (1)

- `at(A1)`, `at(A2)`, `at(A3)`, `at(A4)`, `at(A5)`, `at(A7)`: antecedent `at(A6)` is false
- `at(A6)`:
  - can only get to `A6` via `A3`;
  - can only get to `A3` via `A2`;
  - `A2` sets `turn0.flag` to `0`
- Thread `B` does not modify `turn0`

Proof (2)

Same way. Please do it if you had trouble with proof of (1).

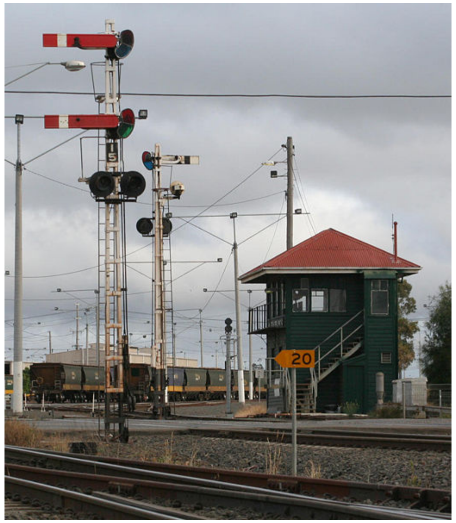
Proof (3): By Contradiction

- Assume thread `A` is at state `A6`:
  - Hence `turn0.flag == 0` (invariant (i))
  - After some time, thread `B` wants to enter `B6`, which would require `turn0.flag == 1`
- Contradiction: `turn0.flag` cannot be `0` and `1` at the same time
- Equivalent proof for thread `B`

Outline

Semaphores

- Special integer variable with two atomic operations
  - `P()`: Passeren, wait/up
  - `V()`: Vrijgeven/Verhogen, signal/down
- Names of operations reflect the Dutch origin of the inventor...



Source: http://en.wikipedia.org/wiki/File:Mechanical-signalling-north-geelong.jpg

Class Semaphore

```java
public class Semaphore {
  private int value;
  public Semaphore() {
    value = 0;
  }
  public Semaphore(int k) {
    value = k;
  }
  public synchronized void P() {
    /* see later */
  }
  public synchronized void V() {
    /* see later */
  }
}
```


P() Operation

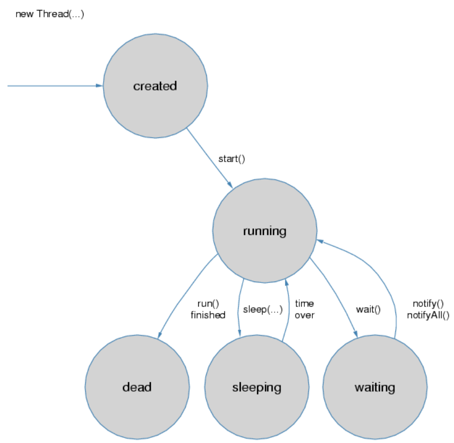
```java
public synchronized void P() {
  while (value == 0) {
    try {
      wait();
    }
    catch (InterruptedException e) {
    }
  }
  value--;
}
```


V() Operation

```java
public synchronized void V() {
  ++value;
  notifyAll();
}
```


wait(), notify(), notifyAll()

- `wait()` tells the calling thread to give up the monitor and go to sleep until some other thread enters the same monitor and calls `notify()`
- `notify()` wakes up a single thread that is waiting on this object's monitor. The choice is arbitrary and occurs at the discretion of the implementation.
- `notifyAll()` wakes up all threads that are waiting on this object's monitor



Comments

- You can only modify the value of a semaphore instance using the `P()` and `V()` operations.
  - Initialize in constructor
- Effect
  - `P()` may block
  - `V()` never blocks
- Application of semaphores:
  - Mutual exclusion
  - Conditional synchronization

Semaphores

- Binary semaphore
  - Value is either 0 or 1
  - Supports implemention of mutual exclusion:

```
Semaphore s = new Semaphore(1);
s.P()
//critical section
s.V()
```

- Counting (general) semaphore
  - Value can be any positive integer value

Fairness

- A semaphore is considered to be “fair” if all threads that execute a `P()` operation eventually succeed
- Semaphore is “unfair”: a thread blocked in the `P()` operation must wait forever while other threads (that executed the operation later) succeed.

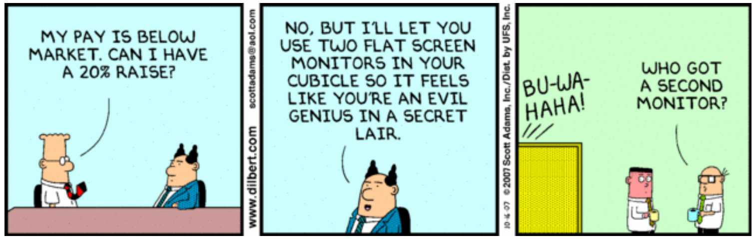
Semaphores in Java

- `java.util.concurrent.Semaphore`
  - `acquire()` instead of `P()`
  - `release()` instead of `V()`
- Constructors
  - `Semaphore(int permits)`
  - `Semaphore(int permits, boolean fair)`
    - `permits`: initial value
    - `fair`: if true then the semphore uses a FIFO to manage blocked threads

Outline

Semaphores and Monitors

- Monitor: model for synchronized methods in Java
- Both constructs are equivalent
- One can be used to implement the other



Example

See slides from March 18 for context

Buffer using condition queues

```
class BoundedBuffer extends Buffer {
  public BoundedBuffer(int size) {
    super(size);
  }
  public synchronized void insert(Object o)
    throws InterruptedException {
    while (isFull())
      wait();
    doInsert(o);
    notifyAll();
  }
  public synchronized Object extract()
    throws InterruptedException {
    while (isEmpty())
      wait();
    Object o = doExtract();
    notifyAll();
    return o;
  }
}
```

Emulation of monitor with semaphores

We need 2 semaphores:

- One to make sure that only one synchronized method executes at any given time
  - call this the "access semaphore" `access`
  - binary semaphore
- One semaphore to line up threads that are waiting for some condition
  - call this the "condition semaphore" `cond`
  - counting (general) semaphore
  - threads that wait must do an "acquire"

For convenience

Counter `waitThread` to count number of waiting threads i.e., threads in queue for `cond`

Basic idea

1. Frame all synchronized methods with `access.acquire()` and `access.release()`
   - This ensures that only one thread executes a synchronized method at any point in time
   - Recall: `access` is binary.
2. Translate `wait()` and `notifyAll()` to give threads waiting in line a chance to progress (these threads use `cond`)
   - To simplify the implementation, we require that `notifyAll()` is the last action in a synchronized method
   - Java does not enforce this requirement but the mapping of synchronized methods into semaphores is simplified

Buffer with auxiliary fields

```java
class BoundedBuffer extends Buffer {
  public BoundedBuffer(int size) {
    super(size);
    access = new Semaphore(1);
    cond = new Semaphore(0);
  }

  private Semaphore access;
  private Semaphore cond;
  private int waitThread = 0;

  // continued
```


(1) Framing all methods

```java
public void insert(Object o)
  throws InterruptedException {
  access.acquire(); // ensure mutual exclusion

  while (isFull())
    wait();

  doInsert(o);
  notifyAll();

  access.release();
}
```

Notes

- There is one semaphore for all synchronized methods of one instance of `BoundedBuffer`:
  - Must make sure that insert and extract don't overlap
- There must be separate semaphore to deal with waiting threads:
  - Imagine the buffer is full. A thread that attempts to insert an item must wait. But it must release the access semaphore. Otherwise a thread that wants to remove an item will not be able to executed the (synchronized!) extract method.

(2) Translate `wait()`

```
waitThread++;

// other threads can execute
// synchronized methods
access.release();

// wait till condition changes
cond.acquire();
access.acquire();

waitThread--;
```


(2) Translate `notifyAll()`

```
if (waitThread > 0) {
  for (int i=0; i < waitThread; i++) {
    cond.release();
  }
}
```

- All threads waiting are released and will compete to (re)acquire `access`
- They decrement `waitThread` after they leave `cond.acquire()`
- Note that to enter the line (i.e., increment `waitThread`) the thread must hold the access semaphore `access`

(2) Translate `wait()` and `notifyAll()`

- Recall that `access.release()` is done at the end of the synchronized method
- So all the threads that had lined up waiting for `cond` compete to get access to `access`
- No thread can line up while the `cond.release()` operations are done since this thread holds `access`

Note

- We wake up all threads – they might not be able to enter their critical section if the condition they waited for does not hold, but all threads get a chance.
- This approach is different from what we discussed in the lecture

Translate wait()

```java
public void insert(Object o)
  throws InterruptedException {
  access.acquire();
  while (isFull()) {
    waitThread++;
    access.release(); // let other thread access object
    cond.acquire(); // wait for change of state
    access.acquire()
    waitThread--;
  }
  doInsert(o);
  notifyAll();
  access.release();
}
```


Translate notifyAll()

```java
public void insert(Object o)
  throws InterruptedException {
  access.acquire();
  while (isFull()) {
    waitThread++;
    access.release(); // let other thread access object
    cond.acquire(); // wait for change of state
    access.acquire()
    waitThread--;
  }
  doInsert(o);
  if (waitThread > 0) {
    for (int i; i < waitThread; i++) {
      cond.release();
    }
  }
  access.release();
}
```


Example

Consider the buffer, one slot is empty, four operations

1. `insert I1`
2. `insert I2`
3. `insert I3`
4. `extract E`

Example

- I1 – access.acquire()
- I2 – access.acquire(): blocks on access
- I3 – access.acquire(): blocks on access
- I1 – waitThread == 0
- I1.release
- I2 – access.acquire() completes
- I2 – buffer full
- waitThread = 1
- I2 – access.release()
- I2 – cond.acquire() – blocks on cond
- I3 – access.acquire() completes
- I3 – buffer full

Example

- waitThread = 2
- I3 – access.release()
- I3 – cond.acquire – blocks on cond
- E – access.acquire()
- remove item
- E – cond.release()
- E – cond.release()
- E – access.release()

One of I2 or I3 will succeed with access.acquire() and be able to insert the next item

Exercise

How would the method extract look like if we used semaphores to emulate monitors?

Outline

Read/Write Lock

- Many shared objects have the property that most method calls return information about the object's state without modifying the object (**readers**) while only a small number of calls actually modify the object (**writers**)
- There is no need for readers to synchronize with one another
  - It is perfectly safe for them to access the object concurrently
- Writers, on the other hand, must lock out readers as well as other writers
- A Read/Write Lock allows multiple readers or a single writer to enter the critical section concurrently

Overview

- Your task is to implement a Read/Write Lock
- At most four threads
- At most two reader threads (shared access is allowed) and one writer thread
- A thread that executes `read()` is a reader
  - At a later time it can be a writer...

Challenges

- No starvation
- Efficient implementation
  - If there are fewer than two readers and no waiting writers then the next reader must be allowed to proceed
  - If there is no contention then a thread must be allowed to proceed immediately
- Your implementation must be fair:
  - You may want to use `FIFOQueue.java` (part of the skeleton)

Comments

- Keep your solution as simple as possible
- Decide what you want to use:
  - Monitors (synchronized methods)
  - Semaphores
    (see `http://java.sun.com/javase/6/docs/api/`)
- Only change class `Monitor.java` (part of the skeleton)
  - If you feel it's necessary to change other files, please let us know
- Please comment your code!

Hints

- `Thread.currentThread()` returns a handle to the current thread and might be useful for the assignment.
- `Thread.currentThread().getId()`:
  - `waitList.enq(Thread.currentThread().getId())`
  - `waitList.getFirstItem() == Thread.currentThread().getId()`

Summary

- Different ways to determine if a thread finished
- Volatile
- Semaphores
- Equivalence of Semaphores and Monitors
- Read/Write Lock