Parallel Programming

Recitation Session 8

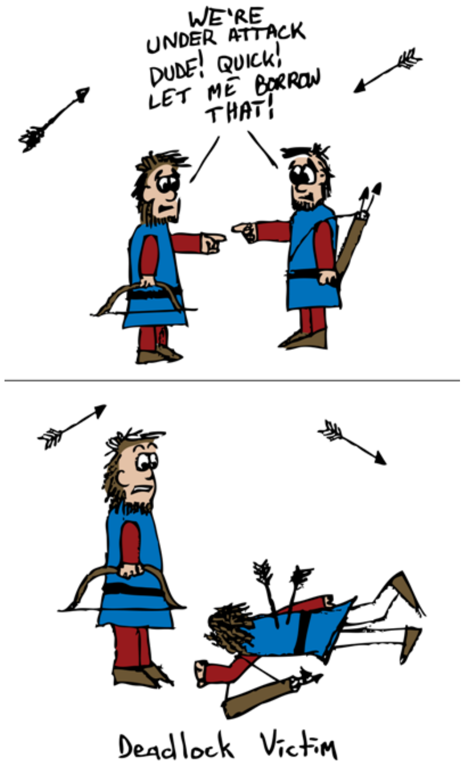
Thomas Weibel <weibelt@ethz.ch>

Laboratory for Software Technology,
Swiss Federal Institute of Technology Zürich

April 29, 2010

Executive Summary

- Some remarks regarding mutual exclusion proofs
- Repeat Read/Write locks
- Classroom exercise: Equivalence of Semaphores and Monitors
- Classroom exercise: Lock proof
- Implement MergeSort and Dining Philosophers with Communicating Sequential Processes



Source: http://dbwhisperer.blogspot.com/2009/07/deadlocks-explained.html

Outline

Notation

```
public void run() {
  while (true) {
    mysignal.request();
    while (true) {
      if (othersignal.read() == 1) break;
      mysignal.free();
      mysignal.request();
    }
    // critical section
    mysignal.free();
  }
}
```

Notation

```
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;
```


Implication

1. at(A6) → turn0.flag == 0

```
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;
```

Why is invariant (1) true at(A1), at(A2), at(A3), at(A4), at(A5), at(A7)?

Implication: Truth Table

$$A \rightarrow B$$

$\rightarrow$	1	0
1	1	0
0	1	1

1. `at(A6)` $\rightarrow$ `turn0.flag == 0`

- `at(A1)`: $0 \rightarrow x = 1$
- `at(A2)`: $0 \rightarrow x = 1$
- `at(A3)`: $0 \rightarrow x = 1$
- `at(A4)`: $0 \rightarrow x = 1$
- `at(A5)`: $0 \rightarrow x = 1$
- `at(A7)`: $0 \rightarrow x = 1$

Equivalence

1. `turn0.flag == 0` $\leftrightarrow$
   (`at(A3)` $\vee$ `at(A4)` $\vee$ `at(A6)` $\vee$ `at(A7)`)

```
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;
```

`A4` $\rightarrow$ `A5` and `A7` $\rightarrow$ `A1`: `turn0.flag == 1`, why does invariant (1) still hold?

Equivalence: Truth Table

$$A \leftrightarrow B$$

$\leftrightarrow$	1	0
1	1	0
0	0	1

1. `turn0.flag == 0` $\leftrightarrow$ `(at(A3)`$\vee$`at(A4)`$\vee$`at(A6)`$\vee$`at(A7))`

- `A4`$\rightarrow$`A5` $==$ `at(A5)`: $0 \leftrightarrow 0 = 1$
- `A7`$\rightarrow$`A1` $==$ `at(A1)`: $0 \leftrightarrow 0 = 1$

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

Assignment 7

- 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...

Monitor

```
public class Monitor {
  final int MAX_THREADS;
  final int MAX_READERS = 2;
  FIFOQueue waitList;
  int readers = 0;
  int writers = 0;
  boolean writing = false;

  public Monitor(int maxThreads) {
    MAX_THREADS = maxThreads;
    waitList = new FIFOQueue(maxThreads);
  }

  public void readLock()    { /* ... */ }
  public void readUnlock()  { /* ... */ }
  public void writeLock()   { /* ... */ }
  public void writeUnlock() { /* ... */ }
}
```


readLock()

```
public synchronized void readLock() {
  if (readers >= MAX_READERS || writing || !waitList.isEmpty()) {
    waitList.enq(Thread.currentThread().getId());

    while (true) {
      try {
        wait();
      } catch (InterruptedException e) {
        e.printStackTrace();
      }

      if (waitList.getFirstItem() == Thread.currentThread().getId()
          && !writing && readers < MAX_READERS) {
        waitList.deq();
        break;
      }
    }
  }

  readers++;
  if (readers < MAX_READERS)
    notifyAll();
  System.out.println("READ LOCK ACQUIRED " + readers);
}
```

readUnlock()

```java
public synchronized void readUnlock() {
  readers--;
  System.out.println("READ LOCK RELEASED " +
                              readers);
  notifyAll();
}
```


writeLock()

```java
public synchronized void writeLock() {
  if (readers > 0 || writers > 0 || !waitList.isEmpty()) {
    waitList.enq(Thread.currentThread().getId());
    while (true) {
      try {
        wait();
      } catch (InterruptedException e) {
        System.out.println(e.getMessage());
      }

      if (waitList.getFirstItem() == Thread.currentThread().getId()
          && !writing && readers == 0) {
        waitList.deq();
        break;
      }
    }
  }

  writers++;
  writing = true;
  System.out.println("WRITE LOCK ACQUIRED " + writers);
}
```

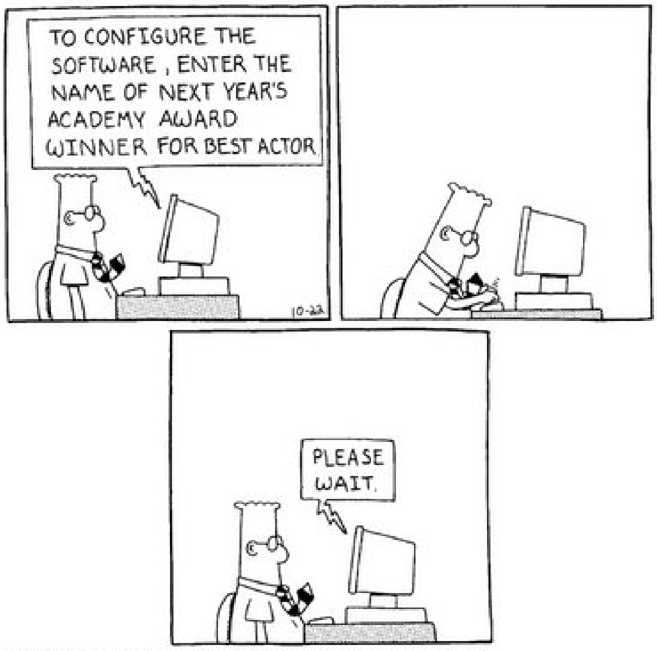
writeUnlock()

```
public synchronized void writeUnlock() {
  writing = false;
  writers--;
  System.out.println("WRITE LOCK RELEASED " +
                               writers);
  notifyAll();
}
```


Outline

Classroom Exercise

- Are semaphores and monitors equivalent?
- How can you implement semaphores with monitors?
- How can you implement monitors with semaphores?
  - What about `wait()` and `notifyAll()`?



DILBERT: © Scott Adams/Dist. by United Feature Syndicate, Inc

Semaphores and Monitors

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

Semaphore Implementation

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


Semaphore Implementation: acquire()

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

Semaphore Implementation: `release()`

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


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`

Monitor with Semaphores

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`)

Monitor with Semaphores: Auxiliary Fields

```
class FooBar {
  private Semaphore access;
  private Semaphore cond;
  private int waitThread = 0;

  public FooBar() {
    access = new Semaphore(1);
    cond = new Semaphore(0);
  }

  // continued
}
```

(1) Framing all methods

```
public void qux() {
  // Ensure mutual exclusion
  access.acquire();

  // Critical section

  access.release();
}
```

is equivalent to

```
public synchronized void qux() {
  // Critical section
}
```


(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.
- `notifyAll()` calls `cond.release()` `waitThread`-times
  - If 3 threads were waiting, `cond.value` is set to 3.
  - Now one of the threads wakes up, decrements `waitThreads`, runs to the end and again calls `cond.release()` 2 times → `cond.value` will now be 4 even though only 2 threads are in the wait section of the code
  - A thread may awake a few times and not find that the condition has changed. As long as all the `wait()` are in `while (some_condition)` loop there will be no harm

`wait()` in while-loop

```
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();
}
```

Outline

Classroom Exercise

```
class MyLock implements Lock {
  private int turn;
  private boolean busy = false;

  public void lock() {
    int me = ThreadID.get();
    while (turn != me) {
      while (busy) {
        turn = me;
      }
      busy = true;
    }
  }

  public void unlock() {
    busy = false;
  }
}
```

- Does this protocol satisfy mutual exclusion?
- Is this protocol starvation-free?
- Is this protocol deadlock-free?

Lock

```
class MyLock implements Lock {
  private int turn;
  private boolean busy = false;

  public void lock() {
/* S1 */     int me = ThreadID.get();
/* S2 */     while (turn != me) {
/* S3 */       while (busy) {
/* S4 */         turn = me;
               }
/* S5 */       busy = true;
             }
  }

  public void unlock() {
/* S6 */     busy = false;
  }
}
```


Mutual Exclusion

Protocol does not satisfy mutual exclusion:

- Assume ThreadIDs start with a value != 0: ThreadIDs don't start with 0 → they start with 1 in Java
- No thread can enter critical section (could argue this is mutual exclusion but you have to clarify your answer)
- turn is initialized to 0 → one aspect of the broken protocol

Mutual Exclusion

T0 attempts to enter the critical section:

```
/* S2 */ → true // me != 0
/* S3 */ false
/* S5 */ busy = true
/* S2 */ true
/* S3 */ true // bye virtue of S5,
               // previous iteration
/* S4 */ turn = me
/* S3 */ true // no exit from loop
```

If another thread attempts to enter the critical section then it will experience the same sequence

Mutual Exclusion

- Assume that `ThreadID`s start at 0 or `turn` is initialized to the `ThreadID` of the first thread (i.e. 1) to enter the critical section
- This thread succeeds but does not set busy to true:

  ```
  /* S1 */    me = 0 // by assumption
  /* S2 */    → false
  ```

  Loop exits but busy unchanged (`false`)

Starvation

- Protocol is not starvation-free
- Assume there are two threads:
  - If `turn` is not initialized to the `ThreadID` of the first thread to acquire the lock, then this thread will be not able to enter its critical section
  - If `turn` is initialized then the second thread never enters its critical section if the first thread leaves its critical section (with `unlock()`) before the second thread attempts to enter its critical section

Deadlock

- Protocol is not deadlock-free
- Situation for deadlock is similar to the situation for starvation in this case
- The thread will enter the lock method but never exit

Outline

Semaphores

- Special Integer variable with 2 atomic operations
  - `P()`: Passeren, wait/up
  - `V()`: Vrijgeven/Verhogen, signal/down
- Map into JCSP by defining a semaphore process

Semaphores in JCSP

- Semaphore `CSProcess`
- Two channels for `P` and `V`
- Two + Two channels
  - `P` – request/confirm
  - `V` – request/confirm

Outline

Overview

- Thread hierarchy similar to previous mergesort assignment
- Threads are replaced with JCSP processes
- Communication allowed only via JCSP channels – no shared data between processes
- Use the JCSP library provided on the website

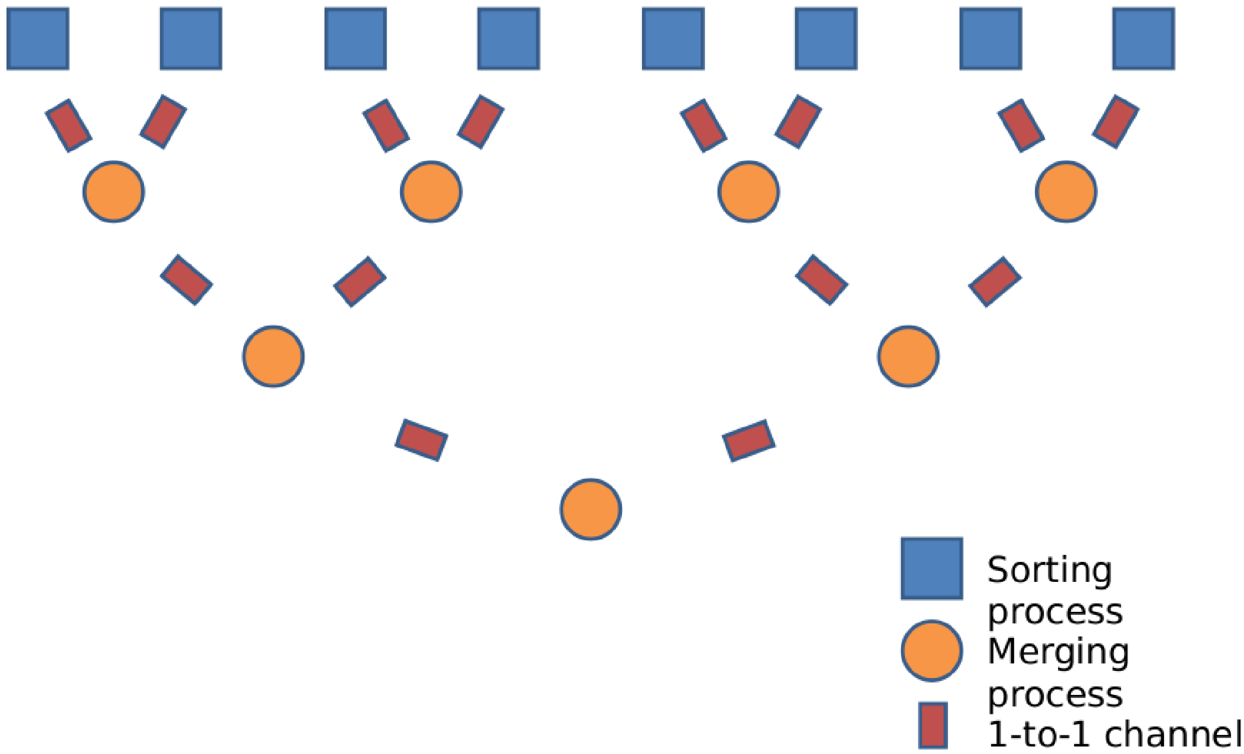
Entities Description

Sorting process: randomly generates a subarray, sorts it sequentially, and then passes the result to its parent merging process

Merging process: takes 2 sorted subarrays from its children, merges and passes the result to its parent merging process

Communication channel: allows data passing between processes

Entities Layout



Reading Policy

How can the data from 2 children be read?

1. Read all data from first child, then read all data from second child
   - Read whole subarray
2. Read data from both children simultaneously
   - Read whole subarray
3. Similar to (2), but read data in pairs and proceed to output current element of the not-yet-fully-sorted subarray
   - Read subarray element-wise

Skeleton

```java
import org.jcsp.lang.*;

public class MergeSort {
  // TODO: add possible fields and methods

  public static void main(String[] args) {
    // the total number of JCSP processes that will be run
    // (will be changed, of course, by your code)
    int numberOfProcesses = 0;

    // TODO: add code here

    // create process array
    CSProcess[] allProcesses = new CSProcess[numberOfProcesses];

    // TODO: add code here

    // run all JCSP processes, they should synchronize via the
    // communication channels
    Parallel parallel = new Parallel(allProcesses);
    parallel.run();
  }
}
```


Skeleton: Sorting

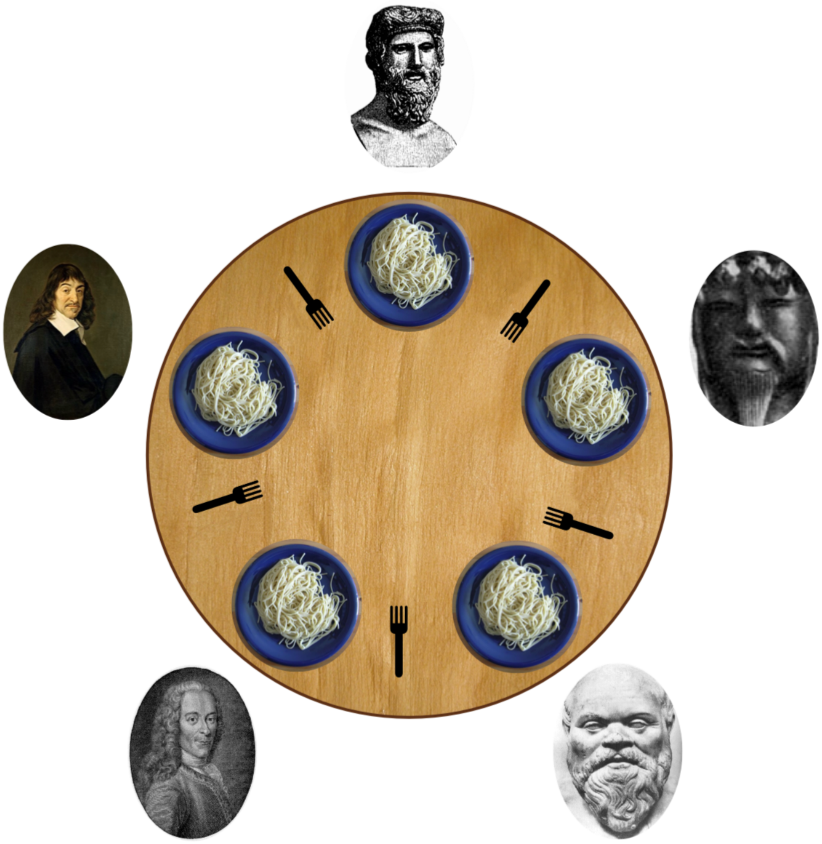
```java
import java.util.Random;
import org.jcsp.lang.*;

public class SortingProcess implements CSProcess {
  // TODO: add possible fields/methods

  public SortingProcess() {
    // TODO: add code, change signature if necessary
  }

  public void run() {
    // TODO: initialize random subarray (or do it in
    //       the constructor, alternatively), sort
    //       it sequentially, and write it to parent
  }
}
```

Skeleton: Merging

```
import org.jcsp.lang.*;

public class MergingProcess implements CSProcess {
  // TODO: add possible fields/methods

  public MergingProcess() {
    // TODO: add code, change signature if necessary
  }

  // the process' run() method
  public void run() {
    // TODO: read subarrays from children, merge them,
    //       write merged subarray to parent
  }
}
```


Outline

Dining Philosophers



Source: http://en.wikipedia.org/wiki/File:Dining_philosophers.png

Dining Philosophers

- Five philosophers sitting at a table doing one of two things:
  - eating or
  - thinking
- While eating, they are not thinking, and while thinking, they are not eating
- Philosophers sit at a circular table with a large bowl of spaghetti in the center
- Each philosopher has one fork to his left and one fork to his right
- Philosopher must eat with two forks: Can only use the forks on his immediate left and immediate right

Deadlock and Starvation

Deadlock

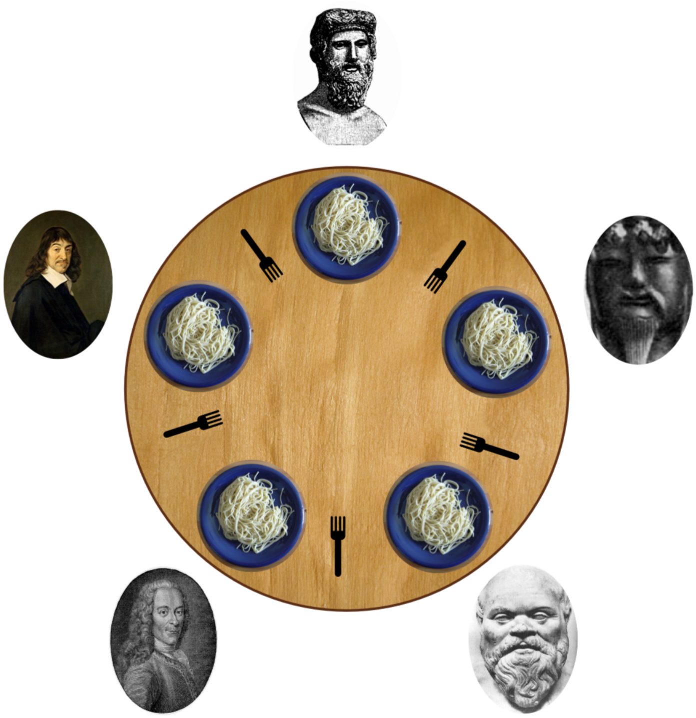
Philosophers never speak to each other: creates a dangerous possibility of deadlock when every philosopher holds a left fork and waits perpetually for a right fork (or vice versa)

Starvation

Starvation – pun intended – might also occur independently of deadlock if a philosopher is unable to acquire both forks because of a timing problem

Strategies to Avoid Deadlock

- Security Guard: limits entry to dining hall
- Asymmetric philosophers: 4 philosophers pick up left fork first, 1 philosopher picks up right fork first
- See `http://en.wikipedia.org/wiki/Dining_philosophers_problem#Solutions` for other strategies



Source: `http://en.wikipedia.org/wiki/File:Dining_philosophers.png`

Skeleton: Main

```
import org.jcsp.lang.CSProcess;
import org.jcsp.lang.Parallel;

public class Main implements CSProcess {
  private final static int NUM_PHILOSOPHERS = 5;
  private final CSProcess go;

  public Main() {
    final Philosopher[] philosophers = new Philosopher[NUM_PHILOSOPHERS];
    for (int i = 0; i < philosophers.length; i++) {
      philosophers[i] = new Philosopher(i);
    }
    go = new Parallel(new CSProcess[] {new Parallel(philosophers)});
  }

  @Override
  public void run() {
    go.run();
  }

  public static void main(String[] args) {
    new Main().run();
  }
}
```


Skeleton: Philosopher

```
import org.jcsp.lang.CSProcess;

public class Philosopher implements CSProcess {
  private final int ID;
  public Philosopher(int id) {
    this.ID = id;
  }

  @Override
  public void run() {
    try {
      for (;;) {
        think(); eat();
      }
    } catch (InterruptedException e) {
      e.printStackTrace();
    }
  }
  private void think() throws InterruptedException {
    Thread.sleep((long)(Math.random() * 500));
    System.out.println("Philosopher " + ID + " : Thinking...");
  }
  private void eat() throws InterruptedException {
    Thread.sleep((long)(Math.random() * 500));
    System.out.println("Philosopher " + ID + " : Please, give me two forks!");
  }
}
```

Summary

- Mutual exclusion proofs
- Read/Write locks
- Equivalence of Semaphores and Monitors
- Lock proof
- MergeSort and Dining Philosophers in JCSP