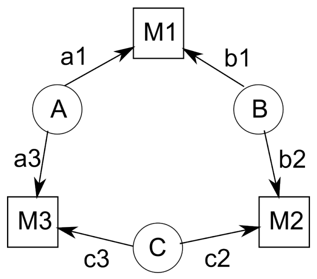
Exam – Solutions EDA040/DATE14 Concurrent and Real-Time Programming

2012–10–27, 14.00–19.00

1. a) Running, Ready, Blocked.
   b) Running<->Ready: By scheduler. Running->Blocked: By current thread. Blocked->Ready: By other thread.
   c) Switching is needed to make sure execution state is preserved for a thread. Each thread has its own stack where the call state is stored, and the execution state is stored in some kind of process record. Switch = save state (including program counter, stack pointer and other registers) into a precess (or thread) record, switch record, and then loading the new execution state from that new record.
   d) More expensive for scheduler-induced switching, and less expensive for call-inititated switching where fewer registers need to be taken care of.

2. a) Graph:



Calculation of blocking times:
$B_C = 0$
$B_B = max(c2, c3) = 120$
$B_A = b1 + c3 = 150$
Calculation of WCETs:
$C_A = a1 + a3 + 15 = 95$
$C_B = b1 + b2 + 45 = 85$
$C_C = c2 + c3 + 35 = 195$
Calculation of response times:
$R_A = C_A + B_A = 95 + 150 = 245 < 400 = T_A$
$R_B = C_B + B_B = 85 + 120 = 205$
$R_B = 205 + \lceil \frac{205}{400} \rceil \cdot 95 = 300$
$R_B = 205 + \lceil \frac{300}{400} \rceil \cdot 95 = 300 < 600 = T_B$
$R_C = C_C + B_C = 195$
$R_C = 195 + \lceil \frac{195}{400} \rceil \cdot 95 + \lceil \frac{195}{600} \rceil \cdot 85 = 375$
$R_C = 195 + \lceil \frac{375}{400} \rceil \cdot 95 + \lceil \frac{375}{600} \rceil \cdot 85 = 375 < 700 = T_C$

b) Assume the blocking times from D are negligible. Response time for D:
$R_D = 100$
$R_D = 100 + \lceil \frac{100}{400} \rceil \cdot 95 + \lceil \frac{100}{600} \rceil \cdot 85 + \lceil \frac{100}{700} \rceil \cdot 195 = 475$
$R_D = 100 + \lceil \frac{475}{400} \rceil \cdot 95 + \lceil \frac{475}{600} \rceil \cdot 85 + \lceil \frac{475}{700} \rceil \cdot 195 = 570$
$R_D = 100 + \lceil \frac{570}{400} \rceil \cdot 95 + \lceil \frac{570}{600} \rceil \cdot 85 + \lceil \frac{570}{700} \rceil \cdot 195 = 570$
The response time is 570 ms, but since D should be lowest priority the period time needs to be $T_D = 700 + \epsilon$ since the system is scheduled according to RMS.

3. The garbage collection thread for the real time threads should be medium priority to not be affected by the low priority threads, and not to disturb those with high priority. The GC for the low priority threads is carried out in the context of each such thread, and thereby with that priority.

4. In short, a higher priority task is indirectly pre-empted by a lower priority task. Scenario: the low priority task holds a resource needed by the high priority task. Meanwhile a medium priority task stops the low priority task from running, effectively inverting task priorities.

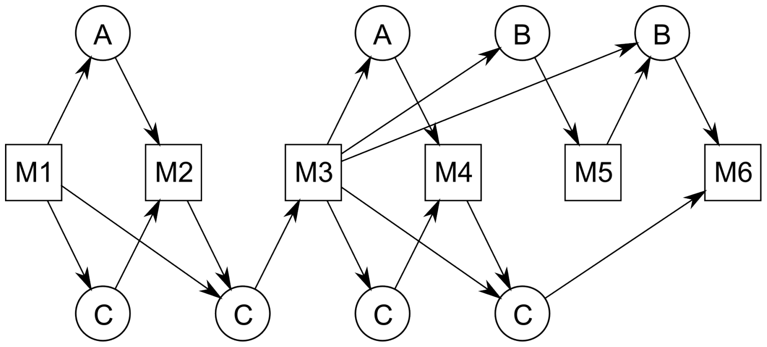
5. a) For instance:

```
class A extends Thread {
  void run() {
    M1.take();
    M2.take()
    useM1M2();
    M2.give();
    M1.give();
    M3.take();
    M4.take();
    useM3M4();
    M4.give();
    M3.give();
  }
}
```

```
class B extends Thread {
  void run() {
    M3.take();
    M5.take()
    M6.take();
    useM3M5M6();
    M6.give();
    M5.give();
    M3.give();
  }
}
```

```
class C extends Thread {
  void run() {
    M1.take();
    M2.take()
    M3.take();
    useM1M2M3();
    M2.give();
    M1.give();
    M4.take();
    M6.take()
    useM3M4M6();
    M3.give();
    M6.give();
    M4.give();
  }
}
```

b) Resource allocation graph:



6. A railroad crossing is to be simulated to determine typical waiting times for cars (in the worst-case direction of traffic). The available gate/train control system is kept, and accessed for our simulation purposes via the `gate` and `train` methods of the `ATC` class. Among other things, this means we need (a pool of) car threads but no train threads (since the trains are behind the `ATC` class). The crossing is represented by the `class Crossing`, which is the prescribed monitor.

   Since queing at the crossing should be in FIFO order, and a monitor waiting queue would be (system specific or) priority based, the queue needs to be represented explicitly. That can be done in many ways, including buffers and lists, but it could also be based on the array of `Car` objects as outlined by the available monitor attributes. In the following example solution, the array elements represent the ordered wait-places for the cars, which driven by their thread advances towards the gate whenever possible. Although this could have been an example when `notify` (not `notifyAll`) could have been used to notify the next car in line, we ignore efficiency aspects and `notifyAll` is used in the following.

   a) The Gate and Train threads are as follows. The Reporter thread is similar but may depend on your design and assumptions, and is left out here.

```
public class Gate extends Thread {
  private Crossing      c;
  private ATC           atc;
  private ATC.GateState state;

  public Gate(Crossing c, ATC atc) {
    this.c = c;
    this.atc = atc;
  }

  public void run() {
    while (!isInterrupted()) {
      state = atc.gate(state);
      c.setGate(state);
    }
  }
}

class Train extends Thread {
  private Crossing       c;
  private ATC            atc;
  private ATC.TrainState state;

  public Train(Crossing c, ATC atc) {
    this.c = c;
    this.atc = atc;
  }

  public void run() {
    while (true) {
      state = atc.train(state);
      c.setTrain(state);
    }
  }
}
```

   b) The corresponding methods are as follows:

```
synchronized void setGate(ATC.GateState gateState) {
  this.gateState = gateState;
```

```
    notifyAll();
    if (gateState == ATC.GateState.IS_DOWN && carsOnTrack > 0) {
      gateReport = true;
      while (gateReport) {
        try {
          wait();
        }
        catch (InterruptedException exc) {/* ignore */}
      }
    }
  }

  synchronized void setTrain(ATC.TrainState trainState) {
    this.trainState = trainState;
    notifyAll();
    if (trainState == ATC.TrainState.NEAR && carsOnTrack > 0) {
      trainReport = true;
      while (trainReport) {
        try {
          wait();
        }
        catch (InterruptedException exc) {/* ignore */}
      }
    }
  }

  synchronized String report() {
    StringBuffer ans = new StringBuffer();
    while (!statReport && !trainReport && !gateReport) {
      try {
        wait();
      }
      catch (InterruptedException exc) {/* ignore */}
    }
    if (statReport) {
      ans.append("statistics.toString()");
    }
    if (trainReport) {
      ans.append("Cars on track when train is near; crash?");
      trainReport = false;
    }
    if (gateReport) {
      ans.append("Cars on track when gates are down; crash?");
      gateReport = false;
    }
    notifyAll();
    return ans.toString();
  }
```

c)

```
  synchronized void carPassing() {
    // notice arrival time
    long t0 = System.currentTimeMillis();
    Car thisCar = (Car) Thread.currentThread();
    // if queue, stop in it
    int k = n; int k0 = n;
    do {
      do {
        --k;
      } while (k >= 0 && q[k] == null);
      if (k > 0) { // still not first in queue
        ++k; // index to first null/vacant place
        if (k == n) { // some space in queue so it is full
          try {
            wait();
          }
          catch (InterruptedException exc) {/* ignore */}
        } else { // move to free slot and wait there
          q[k] = thisCar;
          if (k != k0) notifyAll(); // let anyone behind move on
          long movetime = 2000 * (k0 - k); // 2s per car length
          long dt0 = System.currentTimeMillis();
          try {
            long dt;
            while ((dt = dt0 + movetime - System.currentTimeMillis()) > 0)
              wait(dt);
          }
          catch (InterruptedException exc) {/* ignore */}
          q[k] = null;
          k0 = k;
        }
      }
    } while (k > 0);
    // if not open, stop as first in queue
    while (gateState != ATC.GateState.IS_UP) {
      try {
        wait();
      }
      catch (InterruptedException exc) {/* ignore */}
    }
    // when free/open: notice time
    long time = System.currentTimeMillis() - t0;
    sumWaiting += time;
    numCars++;
    // inc cars passing now
    carsOnTrack++;
    long movetime = 2000; // 2s to cross tracks
    long dt0 = System.currentTimeMillis();
    try {
      long dt;
      while ((dt = dt0 + movetime - System.currentTimeMillis()) > 0)
        wait(dt);
    }
    catch (InterruptedException exc) {/* ignore */}
    q[0] = null;
    notifyAll();
    // after 4s, dec
    carsOnTrack--;
  }
```

7. An reasonable design (no complete solution shown here) should consider the following:
   - The node attached to a track sensor is polled by a periodic thread, and transmits the signal to the PLC. Due to the character of the application, the PLC should stop trains if the track sensors do not all report values periodically and with consistent values.
   - The train light signal is kept red except after an order to go to green. Preferably, the green-order should be repeated periodically, and when no input is obtained for (say) two such periods the light should turn red.
   - Each gate is to be controlled by a thread running on that node. That control thread also checks the gate position (open/closed) sensors, and reports the status to the PLC according to the corresponding enumerations in the Crossing class.
   - The color and all blinking of the crossing light should be controlled by one thread. Another thread running on that node should receive the orders from the PLC concerning the requested state of the crossing lights. By implementing the interaction between these two threads via a monitor, in which wait(blinktime) is used, the change of light color/blinking is resynchronized by a notification of receiving such an order from the PLC.
   - The crossing monitor can in several ways be simplified if there is a PLC thread taking care of all incoming (and outgoing) messages, and then calling the Crossing methods accordingly. The overall system is more complex due to the distribution aspect including the networking, but the central data part can be simplified. That is, since the problem is more of a queuing system as such, the prescribed monitor-based design according to previous problem can be questioned. If one thread only deals with the instance of the Crossing class, it does not need to be synchronized. However, with one calling thread only, the methods have to be redesigned since such that each wait-notify interaction stage turns into a state of the Crossing object. Furthermore, it will then be simpler to make it an event driven (instead of time driven) simulation, but for real-time simulation it is more natural to have the different activities represented as threads all the way (as in the previous problem).