Lösningsförslag till Tentamen, EDAF85 Realtidssystem

2022–04–27, 08.00–13.00

1. Upon a context switch, the operating system switches execution from one thread to another. One thread is temporarily suspended, and another one starts execution. Once the suspended thread resumes, it continues from where it was suspended.

2. A race condition is a situation where the outcome, and often also the correctness, of a program depends on how the threads of the program happen to be scheduled.

3. a) $B_A = 0.3ms$ (direct blocking), $B_B = 0ms$ (no blocking), $B_C = 0.4ms$ (direct blocking), $B_D = 0ms$ (no blocking).

   b) $B_A = 0.3ms$ (direct blocking), $B_B = 0.2 + 0.3 = 0.5ms$ (direct blocking + push-through blocking), $B_C = 0.3ms$ (push-through blocking), $B_D = 0ms$ (no blocking).

4. Here, we need to calculate the response times of the threads. We cannot use the schedulability test of Liu and Layland (i.e., check the CPU utilization) since it assumes that the deadline for each thread is equal to its period. Since there is no blocking in the system, the easiest way to calculate the response times is to draw a scheduling diagram showing how the threads are scheduled at the critical instant when all threads are released simultaneously. But here we choose to calculate the response times.

   a) The order of priority are, according to RMS, C–B–A.

   $R_C^0 = 2$
   $R_C^1 = 2$

   $R_B^0 = 1$
   $R_B^1 = 1 + \left\lceil \frac{1}{6} \right\rceil \cdot 2 = 3$
   $R_B^2 = 1 + \left\lceil \frac{3}{6} \right\rceil \cdot 2 = 3$
   $R_A^0 = 3$
   $R_A^1 = 3 + \left\lceil \frac{3}{6} \right\rceil \cdot 2 + \left\lceil \frac{3}{8} \right\rceil \cdot 1 = 6$
   $R_A^2 = 3 + \left\lceil \frac{6}{6} \right\rceil \cdot 2 + \left\lceil \frac{6}{8} \right\rceil \cdot 1 = 6$

   We now see that thread A will miss its deadline (4). The system is thus not schedulable.

b) DMS states that priorities should be assigned according to deadline. The order of priority now becomes A–C–B. We calculate the new response times:

$R_A^0 = 3$
$R_A^1 = 3$

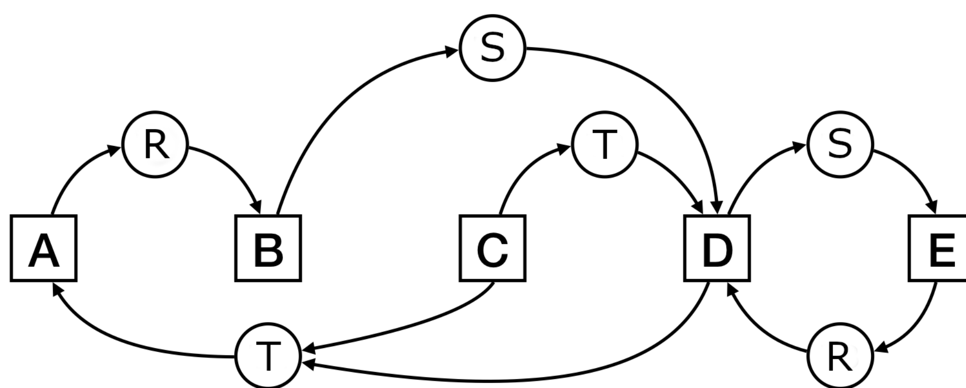
$R_C^0 = 2$
$R_C^1 = 2 + \lceil \frac{2}{10} \rceil \cdot 3 = 5$
$R_C^2 = 2 + \lceil \frac{5}{10} \rceil \cdot 3 = 5$
$R_B^0 = 1$
$R_B^1 = 1 + \left\lceil \frac{1}{10} \right\rceil \cdot 3 + \left\lceil \frac{1}{6} \right\rceil \cdot 2 = 6$
$R_B^2 = 1 + \lceil \frac{6}{10} \rceil \cdot 3 + \lceil \frac{6}{6} \rceil \cdot 2 = 6$

The worst case response time for each thread is now shorter or equal to its deadline. The system is therefore schedulable.

5. a) The resource allocation graph (below) shows two cycles:

- A→(R)→B→(S)→D→(T)→A
- D→(S)→E→(R)→D

Each cycle indicates a potential **circular-wait** situation. This means we have two potential states of deadlock: one involving all three threads R, S, and T; and one involving R and S.



b) We can avoid circular wait (and thus deadlock) by always allocating resources in a particular order. In this example, we can use the order ABCDE. This means that hold-and-wait dependencies in the graph above should always go from left to right, never from right to left.

The following changes (one in R, one in T) ensure allocations are made in order ABCDE:

- Switch the order of line 25 and 26.
- Move line 5 to before line 3.

7.
```
class Selector implements SelectorInterface{
    private int trig=-1;

    public synchronized void trigger(int id) {
        trig = id;
        notifyAll();
        while (trig==id) {
            try { wait(); } catch(InterruptedException e) {}
        }
    }

    public synchronized void awaitTrigger(int id) {
        while (trig!=id) {
            try { wait(); } catch(InterruptedException e) {}
        }
        trig = -1;
        notifyAll();
    }
}
```

8. The solution to this problem can be optimized to minimize the CPU-utilization by using different sampling times during heating and cooling. A simpler solution, albeit not optimal, with a fixed execution period is shown below. This was considered good enough for maximum points.

We start by determining a reasonable sampling rate. The temperature should be kept within a 5°C intervall. To minimize the sampling rate, we should try to use this interval as much as possible. Faster sampling rates makes it possible to use more of the interval, but consumes more CPU power. Let us see what sampling rate we need if we switch off heating at 1°C below the target temperature. The temperature will rise by 1°C in 1/5 seconds, yielding a minimal sampling rate of 5 Hz. To compensate for jitter and measuring errors we can increase the sampling rate somewhat, for example to 10 Hz (every 100 ms). That still seems to be a reasonable sampling rate (1000 Hz or more are NOT reasonable sampling rates!). If we design for 5 Hz, we should turn heating on 1/5°C above the minimum allowed temperature to guarantee that we do not go outside the allowed interval. Actually sampling at 10 Hz should put us on the safe side.

```
public class TemperatureRegulator extends Thread {

    private boolean heating;

    public TemperatureRegulator() {
        heating = false;
        ProcessControl.heating(false);
    }

    public void run(} {
        while(true) {
            double temp = ProcessControl.readTemp();
            if (!heating && temp<=ProcessControl.targetTemp()-4.8} {
                heating = true;
                ProcessControl.heating(true);
            } else {
                if (heating && temp>ProcessControl.targetTemp()-1.0} {
                    heating = false;
                    ProcessControl.heating(false);
                }
            }
            try {
                Thread.sleep(100); // Sample approximately at 10 Hz
            } catch (InterruptedException e) {}
    }
}
```