

# Sammanfattning EDA040

Felix Mulder

16 oktober 2013

# 1 Deadlock Analysis

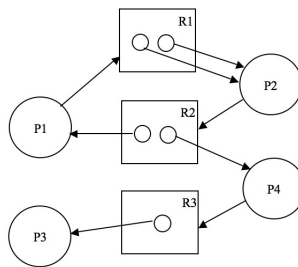
Resource allocation graphs are used to determine if a program can deadlock. For a program to end up in a deadlock there are a few requirements.

- Mutual exclusion: at least one resource is held in a non-shareable mode.
- Hold and wait: there must exist a process that is holding at least one resource and simultaneously waiting for resources that are held by other processes.
- No preemption: resources cannot be preempted; the resource can only be released voluntarily by the resource holding it.
- Circular wait: There must exist a set of processes waiting for each other in a circular structure. I.e: p1 waits for p2, p2 waits for p3, p3 waits for p1.

To draw a resource allocation graph from source code:

1. Draw boxes for each resource.
2. For each thread (i) and line (j), draw a bubble with  $T_{ij}$ . If a thread takes, then draw a line to the resource. For  $T_{i(j+1)}$  draw a line from the resource to the thread.
3. If  $T_{ij}$  only emits or only absorbs arrows, you don't have to keep it in the graph.
4. For resources that exist as multiple instances, draw dots inside the resource. If a cycle exists containing a multiple instance resource, then it may be a false cycle.

Cycles in the graph indicate the possibility of deadlocks.



# 2 Process synchronization

The critical section problem could be solved simply by disallowing interrupts on a single core cpu. With multiple cores, however, disabling these interrupts will be too time consuming.

## 2.1 Dekker's Algorithm

Dekker's algorithm solves the process synchronization problem with busy waits. Meaning: using the below specified code results in a correct handling of critical areas. Alas, the threads spend CPU cycles in the while loop, needlessly. If we implement Dekker's we should compliment it with wait/notify functionality. Without this improvement the semaphores can be referred to as spinlocks. The only advantage with using spinlocks is that there is no context switch required.

Listing 1: Dekker's Algorithm

```
public class Dekkers extends MutualExclusion {
    public Dekkers () {
        flag[0] = false;
        flag[1] = false;
        turn = TURN_0;
    }

    public void enteringCriticalSection (int t) {
        int other;

        other = 1 - t;

        flag[t] = true;
        turn = other;

        while ((flag[other] == true) && (turn == other)) {
            Thread.yield();
        }
    }

    public void leavingCriticalSection (int t) {
        flag[t] = false;
    }

    private volatile int turn;
    private volatile boolean[] flag = new boolean[2];
}
```

## 2.2 Race condition

A race condition is when multiple threads access and manipulate the same data concurrently, and where outcome of the execution depends on the particular order in which access takes place.

## 2.3 Mutual Exclusion

If thread  $T_i$  is executing in its critical section, then no other threads can be executing in their critical sections.

## 2.4 Progress

If no thread is executing in its critical section and there exist threads that wish to enter their critical sections, then only the threads not executing in their critical section get to partake in the process of deciding which thread gets to execute its critical section next.

## 2.5 Starvation

When some threads are allowed to execute and make progress, but others are left “starving.”

## 2.6 Livelock

No thread makes progress, but they keep executing.

## 2.7 Bounded waiting

There exists a limit to the amount of times a thread will wait for other threads before its request to enter a critical area is granted. (This prevents starvation in a single thread.)

## 2.8 Drifting

The following piece of code will cause accumulative drift.

### Listing 2: Drift example

```
while (!isInterrupted()) {  
    sleep(100); foo.bar();  
}
```

Sleep specifies a minimum time to sleep, and a context switch may have occurred after sleep and before the method call thus drift is accumulated.

### Listing 3: Drift fixed

```
long t = System.currentTimeMillis();  
while (!isInterrupted()) {  
    foo.bar();  
    t += 100;  
    long diff = t - System.currentTimeMillis();  
    if (diff > 0) sleep(diff);  
}
```

Even with this fix, sleep still causes a minimum busy wait.

## 2.9 volatile, transient keywords

Volatile means that the compiler is not allowed to cache the value of this variable. It should be updated before evaluation.

Transient means that the variable has no meaning outside of its current context if the variable is passed along with a serialized object over a network, it gets set to “null” or its equivalence.

## 3 Scheduling

### 3.1 Priority inversion phenomenon

Occurs when a low priority thread manages to lock a resource and this thread is then interrupted by a higher priority thread. When the thread requests the same resource, the lower thread blocks the higher thread and can thus resume its execution. (Despite being lower prioritized than the other thread.) If we called the highest prioritized thread A, and call the lowest Z. If Z blocks A, and Z is interrupted by a higher prioritized thread (M?) that doesn't share its resources, then this thread (M) also blocks A. This is called a *priority inversion* since Z and M will execute before A.

### 3.2 Priority inheritance protocol

The basic idea is to modify the priority of the tasks causing the blocking. In particular when Z blocks higher prioritized tasks, it temporarily inherits the highest priority of the blocked tasks. This prevents medium prioritized threads from preempting Z and prolonging the blocking duration.

#### 3.2.1 Basic

Raises the priority of the low priority thread temporarily

#### 3.2.2 Ceil

<http://fileadmin.cs.lth.se/cs/Education/EDA040/lecture/rtpL5.pdf>

#### 3.2.3 Immediate