Sammanfattning EDAF05

Meris Bahti & Felix Mul

16 oktober 2013

1 Deadlock Analysis

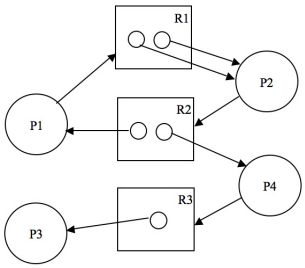
Resource allocation graphs are used to determine if a program can deadlock. For a program to end up in a deadlock there are a few requirements.

- 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

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 correctl 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.

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 Exclustion

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 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.)

2.7 Drifting

The following piece of code will cause accumulative drift.

Listing 2: Dekker's Algorithm

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

Sleep specifies a minimun time to sleep, and a context switch may have ocurred after sleep and before the method call.