



C340 Concurrency: Starvation and Deadlocks

Wolfgang Emmerich



Goals

- ***Reader/Writer problem***
- ***Starvation***
- ***Dining Philosophers Problem***
- ***Deadlocks***
- ***Liveness Analysis using LTS***



Reader / Writer Problem

- *Monitors and Java's synchronize statement guarantee mutual access to objects / methods*
- *Often it is ok for multiple readers to access the object concurrently*
- *Properties required:*

Demo: Reader/Writer

© Wolfgang Emmerich, 1998/99

3



Read/Write Monitor

```
class ReadWrite {
    private protected int readers = 0;
    private protected boolean writing = false;
    // Invariant: (readers>=0 and !writing) or
    // (readers==0 and writing)
    synchronized public void acquireRead() {
        while (writing) {... wait(); ...} ++readers;
    }
    synchronized public void releaseRead() {
        --readers; if(readers==0) notify();
    }
    synchronized public void acquireWrite() {
        while (readers>0||writing) {... wait(); ...}
        writing = true;
    }
    synchronized public void releaseWrite() {
        writing = false; notifyAll();
    }
}
```

Starvation

© Wolfgang Emmerich, 1998/99

4



Writer Starvation

- *NotifyAll awakes both readers and writers*
- *Program relies on Java having a fair scheduling strategy*
- *When readers continually read resource: Writer never gets chance to write. This is an example of starvation.*
- *Solution: Avoid writer starvation by making readers defer if there is a writer waiting*

© Wolfgang Emmerich, 1998/99

5



Read/Write Monitor (Version 2)

```
class ReadWrite {
  ... // as before
  private int waitingW = 0; // # waiting Writers
  synchronized public void acquireRead() {
    while (writing || waitingW>0) {... wait(); ... }
    ++readers;
  }
  synchronized public void releaseRead() {... }
  synchronized public void acquireWrite() {
    while (readers>0 || writing) {
      ++waitingW; ... try{ wait(); ... --waitingW; }
      writing = true;
    }
  }
  synchronized public void releaseWrite() {... }
}
```

Demo: Reader/Writer v2

© Wolfgang Emmerich, 1998/99

6



Reader Starvation

- *If there is always a waiting writer:
Readers starve*
- *Solution: Alternating preference between readers and writers*
- *To do so: Another boolean attribute
readersturn in Monitor that indicates
whose turn it is*
- *readersturn is set by releaseWrite()
and cleared by releaseRead()*

© Wolfgang Emmerich, 1998/99

7



Read/Write Monitor (Version 3)

```
class ReadWrite {
... // as before
private boolean readersturn = false;
synchronized public void acquireRead() {
    while(writing || (waitingW>0 && !readersturn))
        { ... wait(); ... }
    ++readers;
}
synchronized public void releaseRead() {
    --readers; readersturn=false;
    if(readers==0) notifyAll();
}
synchronized public void acquireWrite() {... }
synchronized public void releaseWrite() {
    writing=false; readersturn=true; notifyAll();
}
}
```

Demo: Reader/Writer v3

© Wolfgang Emmerich, 1998/99

8



Deadlocks

- *Process is in a deadlock if it is blocked waiting for a condition that will never become true*
- *Process is in a livelock if it is spinning while waiting for a condition that will never become true (busy wait deadlock)*
- *Both happen if concurrent processes and threads are mutually waiting for each other*
- *Example: Dining philosophers*

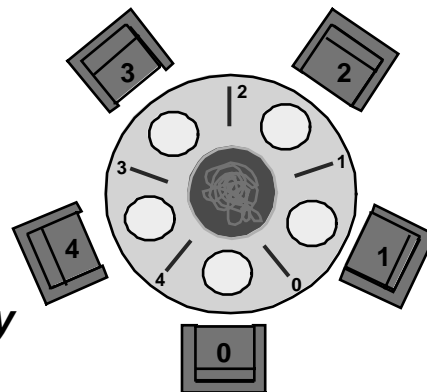
© Wolfgang Emmerich, 1998/99

9



Dining Philosopher Problem

- *5 Philosophers sit around table*
- *They think or eat*
- *Eat with 2 chopsticks*
- *Only 5 chopsticks available*
- *Each philosopher only uses sticks to her left and right*



© Wolfgang Emmerich, 1998/99

10



FSP Model of Dining Philosophers

```
PHIL=(hungry->left.get->right.get->eating->
      left.put->right.put->thinking->PHIL).
FORK = (left.get-> left.put -> FORK
        |right.get->right.put -> FORK).
|| COLLEGE(N=5)=
(phil[0..N-1]:PHIL||fork[0..N-1]:FORK)
/{phil[i:0..N-1].left/fork[i].left,
  phil[i:0..N-1].right/fork[((i-1)+N)%N].right}.
```

LTSA

© Wolfgang Emmerich, 1998/99

11



Dining Philosophers in Java

```
class Philosopher extends Thread {
    int identity;
    Chopstick left; Chopstick right;
    Philosopher(Chopstick left,Chopstick right){
        this.left = left; this.right = right;
    }
    public void run() {
        while (true) {
            try {
                sleep(...);           // thinking
                right.get(); left.get(); // hungry
                sleep(...);           // eating
                right.put(); left.put();
            } catch (InterruptedException e) {}
        }
    }
}
```

© Wolfgang Emmerich, 1998/99

12



Chopstick Monitor

```
class Chopstick {
    boolean taken=false;
    synchronized void put() {
        taken=false;
        notify();
    }
    synchronized void get() throws
        InterruptedException
    {
        while (taken) wait();
        taken=true;
    }
}
```

© Wolfgang Emmerich, 1998/99

13



Applet for Diners

```
for (int i =0; i<N; ++i)
    // create Chopsticks
    stick[i] = new Chopstick();
for (int i =0; i<N; ++i){
    // create Philosophers
    phil[i]=new Philosopher(
        stick[(i-1+N)%N],stick[i]);
    phil[i].start();
}
```

Demo: Diners

© Wolfgang Emmerich, 1998/99

14



Deadlock in Dining Philosopher

- ***If each philosopher has acquired her left chopstick the threads are mutually waiting for each other***
- ***Potential for deadlock exists independent of thinking and eating times***
- ***Only probability is increased if these times become shorter***



Analysing cause of Deadlock

- ***We can use LTS for deadlock analysis***
- ***A dead state in the composed LTS is one that does not have outgoing transitions***
- ***Are these dead states reachable?***
- ***Use of reachability analysis***
- ***Traces to dead states helps understanding the causes of a deadlock***

LTSA



Deadlock Avoidance

- ***Deadlock in dining philosophers can be avoided if one philosopher picks up sticks in reverse order (right before left).***

Demo: Deadlock free Diners

- ***What is the problem with this solution?***
- ***Are there other solutions?***
- ***Deadlock can also be avoided if there is always one philosopher who thinks***



Summary

- ***Reader / Writer Problem***
- ***Starvation***
- ***Avoidance of Starvation***
- ***Dining Philosophers Problem***
- ***Deadlocks and Livelocks***
- ***Deadlock Avoidance***
- ***Next Session: Safety***