



3C03 Concurrency: Mutual Exclusion

Wolfgang Emmerich



Goals of this lecture

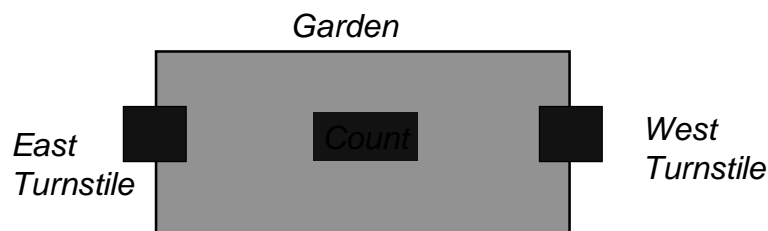
- **Thread interaction via shared memory**
- **Avoid interference**
- **Synchronisation**
- **Mutual exclusive access**





Ornamental Garden Problem

- **Garden open to the public**
- **Enter through either one of two turnstiles**
- **Computer to count number of visitors**



- **Each turnstile implemented by a thread**

© Wolfgang Emmerich, 1998/99

3



Ornamental Garden: Counter class

```
class Counter {
    int value_=0;
    public void increment() {
        int temp = value_; //read
        Simulate.interrupt();
        ++temp;             //add1
        value_=temp;       //write
    }
}
```

- **Simulated interrupt calls `yield()` to force thread switch.**

© Wolfgang Emmerich, 1998/99

4



Ornamental Garden: Turnstile class

```
class Turnstile extends Thread {
    Counter people_;
    Turnstile(Counter c) {
        people_ = c;
    }
    public void run() {
        while(true)
            people_.increment();
    }
}
```

- *For full implementation see online version*



Ornamental Garden: Program

```
Counter people_ = new Counter();
Turnstile west_ = new Turnstile(people_);
Turnstile east_ = new Turnstile(people_);
west_.start();
east_.start();
```

- *What will happen?*

Demo: Ornamental Garden



FSP Spec of Ornamental Garden

```
const N = 3 range T = 0..N
VAR = VAR[0],
VAR[u:T] = (read[u] -> VAR[u]
            | write[v:T]-> VAR[v]).
TURNSTILE = ( arrive -> INCREMENT
             | suspend-> resume-> TURNSTILE),
INCREMENT = (val.read[x:T] -> val.write[x+1]->
             TURNSTILE)+{val.read[T],val.write[T]}.
||GARDEN = (east:TURNSTILE || west: TURNSTILE
           || {east,west}::val:VAR
           )/{stop/east.suspend,
             stop/west.suspend,
             start/east.start,
             start/west.start}.
```

LTSA

© Wolfgang Emmerich, 1998/99

7



Interference

- ***FSP spec supports the following trace:***
east.arrive→*east.val.read.0*→*west.arrive*→
west.val.read.0→*east.val.write.1*→*west.val.write.1*
- ***This is an example of a destructive update***
- ***Destructive updates caused by arbitrary interleaving of read and write actions on shared variables is called interference***
- ***Avoid interference by making access to critical sections mutually exclusive***

© Wolfgang Emmerich, 1998/99

8



Critical Section

- A **critical section** is a sequence of actions that must be executed by at most one process or thread at a time
- Can be found by searching for sections of code that access or update variables or objects that are shared by concurrent processes.



Modelling Mutual Exclusion

- A lock can be modelled by:
`LOCK = (acquire->release->LOCK) .`
- Attaching lock to shared resource (VAR):
`||LOCKVAR = (LOCK || VAR) .`
- Critical section acquires/releases lock:
`INCREMENT = (val.acquire ->
 val.read[x:T] -> val.write[x+1]->
 val.release -> TURNSTILE)
 +{val.read[T],val.write[T]} .`



Critical Sections in Java

- **Synchronised methods implement mutual exclusion**
- **Implicitly locking objects**

```
class Counter {
    int value_=0;
    public synchronized void increment() {
        int temp = value_; //read
        Simulate.interrupt();
        ++temp;           //add1
        value_=temp;     //write
    }
}
```

Demo: Correct Ornamental Garden

© Wolfgang Emmerich, 1998/99

11



Synchronised Statements in Java

- **Locks on individual objects:**

```
public void run() {
    while(true)
        synchronized(people_){
            people_.increment();
        }
}
```

- **Less elegant than synchronized methods**
- **More efficient than synchronized methods**

© Wolfgang Emmerich, 1998/99

12



Summary

- ***Interference***
- ***Critical sections***
- ***Mutual Exclusion***
- ***Synchronised methods in Java***
- ***Synchronised statements in Java***