G52CON:
Concepts of Concurrency

Lecture 6: Algorithms for Mutual Exclusion II

Brian Logan

School of Computer Science

bsl@cs.nott.ac.uk

Outline of this lecture

- mutual exclusion with standard instructions
- example: Peterson's algorithm
- comparison with the Test-and-Set solution
- spin locks in Java
- Java memory model: atomicity, visibility, ordering
- Peterson's algorithm in Java

Spin lock using Test-and-Set

```
bool lock = false;        // shared lock variable

// Process i
init_i;
while(true) {
  while (TS(lock)) {};   // entry protocol
  crit_i;
  lock = false;          // exit protocol
  rem_i;
}
```

Simple spin lock using standard instructions

```
bool lock = false;          // shared lock variable

// Process i
init_i;
while(true) {
  while(lock) {};           // entry protocol
  lock = true;              // entry protocol
  crit_i;
  lock = false;             // exit protocol
  rem_i;
}
```

Mutual exclusion violation

```
// Process 1                    // Process 2

init1;                          init2;
while(true) {                   while(true) {
    while(lock)                     while(lock)
    lock = true;                    lock = true;
    crit1;                          crit2;


}                               }
```

lock == *true*

Dekker's algorithm

```
// Process 1
init1;
while(true) {
     c1 = 0;   // entry protocol
     while (c2 == 0) {
          if (turn == 2) {
               c1 = 1;
               while (turn == 2) {};
               c1 = 0;
          }
     }
     crit1;
     turn = 2; // exit protocol
     c1 = 1;
     rem1;
}
```

```
// Process 2
init2;
while(true) {
     c2 = 0;   // entry protocol
     while (c1 == 0) {
          if (turn == 1) {
               c2 = 1;
               while (turn == 1) {};
               c2 = 0;
          }
     }
     crit2;
     turn = 1; // exit protocol
     c2 = 1;
     rem2;
}
```

```
c1 == 1 c2 == 1 turn == 1
```

Peterson's algorithm

```
// Process 1
init1;
while(true) {
    // entry protocol
    c1 = true;
    turn = 2;
    while (c2 && turn == 2) {};
    crit1;
    // exit protocol
    c1 = false;
    rem1;
}
```

```
// Process 2
init2;
while(true) {
    // entry protocol
    c2 = true;
    turn = 1;
    while (c1 && turn == 1) {};
    crit2;
    // exit protocol
    c2 = false;
    rem2;
}
```

```
// shared variables
bool c1 = c2 = false;
integer turn = 1;
```

An example trace 1.1

```
// Process 1
init1;
```

```
// Process 2
init2;
```

```
}
```

```
}
```

```
// shared variables
bool c1 = false; c2 = false; integer turn = 1;
```

An example trace 1.2

```
// Process 1
init1;
while(true) {

}
```

```
// Process 2
init2;

}
```

```
// shared variables
bool c1 = false; c2 = false; integer turn = 1;
```

An example trace 1.3

```
// Process 1
init1;
while(true) {
    // entry protocol
    c1 = true;




}
```

```
// Process 2
init2;




}
```

```
// shared variables
bool c1 = true; c2 = false; integer turn = 1;
```

An example trace 1.4

```
// Process 1
init1;
while(true) {
    // entry protocol
    c1 = true;
    turn = 2;




}
```

```
// Process 2
init2;




}
```

```
// shared variables
bool c1 = true; c2 = false; integer turn = 2;
```

An example trace 1.5

```
// Process 1
init1;
while(true) {
    // entry protocol
    c1 = true;
    turn = 2;
    while (c2 && turn == 2) {};




}
```

```
// Process 2
init2;




}
```

```
// shared variables
bool c1 = true; c2 = false; integer turn = 2;
```

An example trace 1.6

```
// Process 1
init1;
while(true) {
    // entry protocol
    c1 = true;
    turn = 2;
    while (c2 && turn == 2) {};
    crit1;



}
```

```
// Process 2
init2;




}
```

```
// shared variables
bool c1 = true; c2 = false; integer turn = 2;
```

An example trace 1.7

```
// Process 1
init1;
while(true) {
    // entry protocol
    c1 = true;
    turn = 2;
    while (c2 && turn == 2) {};
    crit1;



}
```

```
// Process 2
init2;
while(true) {



}
```

```
// shared variables
bool c1 = true; c2 = false; integer turn = 2;
```

An example trace 1.8

```
// Process 1
init1;
while(true) {
    // entry protocol
    c1 = true;
    turn = 2;
    while (c2 && turn == 2) {};
    crit1;



}
```

```
// Process 2
init2;
while(true) {
    // entry protocol
    c2 = true;



}
```

```
// shared variables
bool c1 = true; c2 = true; integer turn = 2;
```

An example trace 1.9

```
// Process 1
init1;
while(true) {
    // entry protocol
    c1 = true;
    turn = 2;
    while (c2 && turn == 2) {};
    crit1;



}
```

```
// Process 2
init2;
while(true) {
    // entry protocol
    c2 = true;
    turn = 1;



}
```

```
// shared variables
bool c1 = true; c2 = true; integer turn = 1;
```

An example trace 1.10

```
// Process 1
init1;
while(true) {
    // entry protocol
    c1 = true;
    turn = 2;
    while (c2 && turn == 2) {};
    crit1;



}
```

```
// Process 2
init2;
while(true) {
    // entry protocol
    c2 = true;
    turn = 1;
    while (c1 && turn == 1) {};



}
```

```
// shared variables
bool c1 = true; c2 = true; integer turn = 1;
```

An example trace 1.11

```
// Process 1
init1;
while(true) {
    // entry protocol
    c1 = true;
    turn = 2;
    while (c2 && turn == 2) {};
    crit1;



}
```

```
// Process 2
init2;
while(true) {
    // entry protocol
    c2 = true;
    turn = 1;
    while (c1 && turn == 1) {};



}
```

```
// shared variables
bool c1 = true; c2 = true; integer turn = 1;
```

An example trace 1.12

```
// Process 1
init1;
while(true) {
    // entry protocol
    c1 = true;
    turn = 2;
    while (c2 && turn == 2) {};
    crit1;
    // exit protocol
    c1 = false;

}
```

```
// Process 2
init2;
while(true) {
    // entry protocol
    c2 = true;
    turn = 1;
    while (c1 && turn == 1) {};

}
```

```
// shared variables
bool c1 = false; c2 = true; integer turn = 1;
```

An example trace 1.13

```
// Process 1
init1;
while(true) {
    // entry protocol
    c1 = true;
    turn = 2;
    while (c2 && turn == 2) {};
    crit1;
    // exit protocol
    c1 = false;
    rem1;
}
```

```
// Process 2
init2;
while(true) {
    // entry protocol
    c2 = true;
    turn = 1;
    while (c1 && turn == 1) {};

}
```

```
// shared variables
bool c1 = false; c2 = true; integer turn = 1;
```

An example trace 1.14

```
// Process 1
init1;
while(true) {
    // entry protocol
    c1 = true;
    turn = 2;
    while (c2 && turn == 2) {};
    crit1;
    // exit protocol
    c1 = false;
    rem1;
}
```

```
// Process 2
init2;
while(true) {
    // entry protocol
    c2 = true;
    turn = 1;
    while (c1 && turn == 1) {};

}
```

```
// shared variables
bool c1 = false; c2 = true; integer turn = 1;
```

An example trace 1.15

```
// Process 1
init1;
while(true) {
    // entry protocol
    c1 = true;
    turn = 2;
    while (c2 && turn == 2) {};
    crit1;
    // exit protocol
    c1 = false;
    rem1;
}
```

```
// Process 2
init2;
while(true) {
    // entry protocol
    c2 = true;
    turn = 1;
    while (c1 && turn == 1) {};
    crit2;


}
```

```
// shared variables
bool c1 = false; c2 = true; integer turn = 1;
```

Question (a)

What happens if Process 2 is slow (or swapped out) and doesn't notice that Process 1 has left its critical section?

An example trace 2.1

```
// Process 1
init1;
while(true) {
    // entry protocol
    c1 = true;
    turn = 2;
    while (c2 && turn == 2) {};
    crit1;



}
```

```
// Process 2
init2;
while(true) {
    // entry protocol
    c2 = true;
    turn = 1;
    while (c1 && turn == 1) {};



}
```

```
// shared variables
bool c1 = true; c2 = true; integer turn = 1;
```

An example trace 2.2

```
// Process 1
init1;
while(true) {
    // entry protocol
    c1 = true;
    turn = 2;
    while (c2 && turn == 2) {};
    crit1;
    // exit protocol
    c1 = false;

}
```

```
// Process 2
init2;
while(true) {
    // entry protocol
    c2 = true;
    turn = 1;
    while (c1 && turn == 1) {};

}
```

```
// shared variables
bool c1 = false; c2 = true; integer turn = 1;
```

An example trace 2.3

```
// Process 1
init1;
while(true) {
    // entry protocol
    c1 = true;
    turn = 2;
    while (c2 && turn == 2) {};
    crit1;
    // exit protocol
    c1 = false;
    rem1;
}
```

```
// Process 2
init2;
while(true) {
    // entry protocol
    c2 = true;
    turn = 1;
    while (c1 && turn == 1) {};

}
```

```
// shared variables
bool c1 = false; c2 = true; integer turn = 1;
```

An example trace 2.4

```
// Process 1
init1;
while(true) {




}
```

```
// Process 2
init2;
while(true) {
    // entry protocol
    c2 = true;
    turn = 1;
    while (c1 && turn == 1) {};




}
```

```
// shared variables
bool c1 = false; c2 = true; integer turn = 1;
```

An example trace 2.5

```
// Process 1
init1;
while(true) {
    // entry protocol
    c1 = true;




}
```

```
// Process 2
init2;
while(true) {
    // entry protocol
    c2 = true;
    turn = 1;
    while (c1 && turn == 1) {};




}
```

```
// shared variables
bool c1 = true; c2 = true; integer turn = 1;
```

An example trace 2.6

```
// Process 1
init1;
while(true) {
    // entry protocol
    c1 = true;
    turn = 2;




}
```

```
// Process 2
init2;
while(true) {
    // entry protocol
    c2 = true;
    turn = 1;
    while (c1 && turn == 1) {};




}
```

```
// shared variables
bool c1 = true; c2 = true; integer turn = 2;
```

An example trace 2.7

```
// Process 1
init1;
while(true) {
    // entry protocol
    c1 = true;
    turn = 2;
    while (c2 && turn == 2) {};




}
```

```
// Process 2
init2;
while(true) {
    // entry protocol
    c2 = true;
    turn = 1;
    while (c1 && turn == 1) {};



}
```

```
// shared variables
bool c1 = true; c2 = true; integer turn = 2;
```

An example trace 2.8

```
// Process 1
init1;
while(true) {
    // entry protocol
    c1 = true;
    turn = 2;
    while (c2 && turn == 2) {};




}
```

```
// Process 2
init2;
while(true) {
    // entry protocol
    c2 = true;
    turn = 1;
    while (c1 && turn == 1) {};




}
```

```
// shared variables
bool c1 = true; c2 = true; integer turn = 2;
```

An example trace 2.9

```
// Process 1
init1;
while(true) {
    // entry protocol
    c1 = true;
    turn = 2;
    while (c2 && turn == 2) {};




}
```

```
// Process 2
init2;
while(true) {
    // entry protocol
    c2 = true;
    turn = 1;
    while (c1 && turn == 1) {};




}
```

```
// shared variables
bool c1 = true; c2 = true; integer turn = 2;
```

An example trace 2.10

```
// Process 1
init1;
while(true) {
    // entry protocol
    c1 = true;
    turn = 2;
    while (c2 && turn == 2) {};




}
```

```
// Process 2
init2;
while(true) {
    // entry protocol
    c2 = true;
    turn = 1;
    while (c1 && turn == 1) {};
    crit2;




}
```

```
// shared variables
bool c1 = true; c2 = true; integer turn = 2;
```

Question (b)

How many times can one process that wants to enter its critical section be ‘bypassed’ by the other before the first process gets to enter its critical section?

Question (c)

What would happen if we swapped the order of the statements in the entry protocol? Is the algorithm still correct?

Properties of Peterson's algorithm

The solution based on Peterson's algorithm has the following properties:

- **Mutual Exclusion:** yes
- **Absence of Livelock:** yes
- **Absence of Unnecessary Delay:** yes
- **Eventual Entry:** is guaranteed even if scheduling policy is only *weakly fair*.

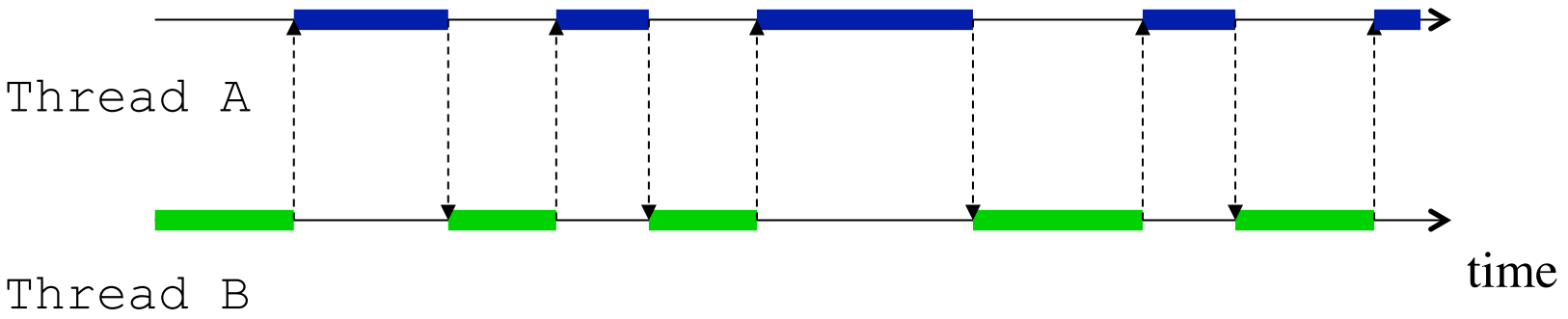
  A weakly fair scheduling policy guarantees that if a process requests to enter its critical section (and does not withdraw the request), the process will *eventually* enter its critical section.

Comparison with Test-and-Set

	Test-and-Set	**Peterson's Algorithm**
Mutual Exclusion:	yes	yes
Absence of Deadlock:	yes	yes
Absence of Unnecessary Delay:	yes	yes
Eventual Entry:	scheduling strongly fair	scheduling weakly fair
Practical issues:	special instructions, any number of processes	standard instructions, > 2 processes complex

Java execution

Consider a Java program consisting of two threads:



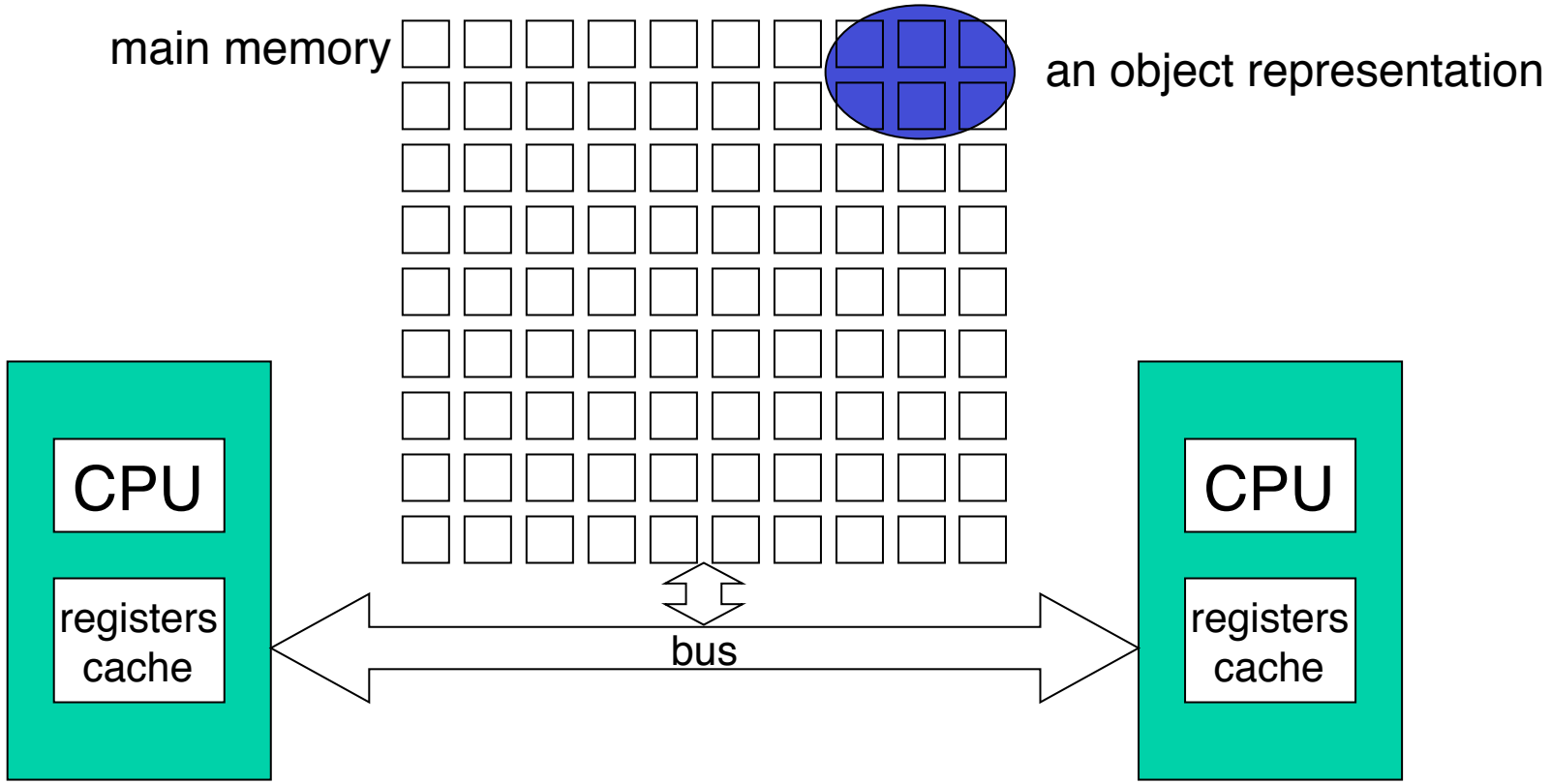
Given a single processor, the JVM executes a sequence of instructions which is an *interleaving* of the instruction sequences for each thread.

Java code optimisation

Not only may concurrent executions be interleaved, they may also be reordered and otherwise manipulated to increase execution speed:

- the compiler may rearrange the order of the statements;
- the processor may rearrange the execution order of the machine instructions;
- the memory system may rearrange the order in which writes are committed to memory;
- the compiler, processor and/or memory system may maintain variable values in, e.g., CPU registers, rather than writing them to memory so long as the code has the "intended" effect.

Java Memory Model



Working memory

Java allows threads that access shared variables to keep private ‘working copies’ of variables:

- each thread is defined to have a *working memory* (an abstraction of caches and registers) in which to store values;
- this allows a more efficient implementation of multiple threads.

Model properties

The Java Memory Model specifies when values must be transferred between main memory and per-thread *working memory*:

- **atomicity**: which instructions must have indivisible effects
- **visibility**: under what conditions are the effects of one thread visible to another; and
- **ordering**: under what conditions the effects of operations can appear out of order to any given thread.

Atomicity

Reads and writes to memory cells corresponding to fields of any type *except* `long` or `double` are guaranteed to be atomic:

- when a field (other than `long` or `double`) is used in an expression, you will get either its initial value or some value that was written by some thread;
- however you are not guaranteed to get the value most recently written by any thread.

Visibility

Without synchronization, changes to fields made by one thread are not guaranteed to be visible to other threads:

- the first time a thread accesses a field of an object, it sees either the initial value of the field or a value since written by some other thread; and
- when a thread terminates, all written variables are flushed to main memory.

Ordering

The apparent order in which the instructions in a method are executed can differ:

- from the point of view of the thread executing the method, instructions *appear* to be executed in the proper order (*as-if-serial* semantics);
- from the point of view of other threads executing unsynchronised methods almost anything can happen;

Example: Peterson's algorithm

```
// Process 1
init1;
while(true) {
    // entry protocol
    c1 = true;
    turn = 2;
    while (c2 && turn == 2) {};
    crit1;
    // exit protocol
    c1 = false;
    rem1;
}
```

```
// Process 2
init2;
while(true) {
    // entry protocol
    c2 = true;
    turn = 1;
    while (c1 && turn == 1) {};
    crit2;
    // exit protocol
    c2 = false;
    rem2;
}
```

```
// shared variables
bool c1 = c2 = false;
integer turn = 1;
```

Assumptions for Peterson's algorithm

Peterson's algorithm implicitly relies on:

- **atomicity:** variable reads and writes being atomic;
- **visibility:** the values written to the variables being immediately propagated to the other process (thread);
- **ordering:** the ordering of the instructions being preserved; and
- that the **scheduling policy** is at least *weakly fair*, otherwise eventual entry is not guaranteed.

Weak fairness

A *weakly fair* scheduling policy guarantees that if a process requests to enter its critical section (and does not withdraw the request), the process will *eventually* enter its critical section.

Spin locks in Java

With unsynchronized code, all that is guaranteed by the Java Memory Model is that the variable reads and writes are atomic:

- we may have to wait an arbitrarily long time for new values of, e.g., `c1` or `turn`, to be propagated to the other thread
- an optimising compiler could reorder the instructions so long as the threads themselves can't tell the difference, e.g., the compiler could swap the order of

```
c1 = 1;
turn = 2;
```

  in the entry protocol, since the thread executing the statements can't tell the difference.

`volatile` fields

If a field is declared `volatile`, a thread must reconcile its working copy of the field with the master copy every time it accesses the variable.

- reads and writes to a `volatile` field are guaranteed to be atomic (even for `longs` and `doubles`);
- new values are immediately propagated to other threads; and
- from the point of view of other threads, the relative ordering of operations on `volatile` fields are preserved.

However the ordering and visibility effects surround only the single read or write to the `volatile` field itself, e.g, '++' on a `volatile` field is not atomic.

Spin locks with `volatile`

```
// Process 1
init1;
while(true) {
    // entry protocol
    c1 = true;
    turn = 2;
    while (c2 && turn == 2) {};
    crit1;
    // exit protocol
    c1 = false;
    rem1;
}
```

```
// Process 2
init2;
while(true) {
    // entry protocol
    c2 = true;
    turn = 1;
    while (c1 && turn == 1) {};
    crit2;
    // exit protocol
    c2 = false;
    rem2;
}
```

```
// shared variables
volatile bool c1 = c2 = false;
volatile integer turn = 1;
```

Assumptions for Peterson's algorithm

Peterson's algorithm implicitly relies on:

- **atomicity:** variable reads and writes being atomic;
- **visibility:** the values written to the variables being immediately propagated to the other process (thread);
- **ordering:** the ordering of the instructions being preserved;
- that the **scheduling policy** is at least *weakly fair*, otherwise eventual entry is not guaranteed.

Archetypical mutual exclusion

In lecture 2, we assumed that:

- the initialisation, critical sections and remainder may be of any size any may take any length of time to execute–each may vary from one pass through the `while` loop to the next;
- the critical sections must execute in a finite time; i.e., each process must leave its critical section after a finite period of time; and
- the initialisation and remainder of each process may be infinite.

If the critical sections don't execute in finite time, the scheduling policy can't be weakly fair.

Properties of the Java scheduler

However Java makes *no* promises about scheduling or fairness, and does not even strictly guarantee that threads make forward progress:

- most Java implementations display some sort of weak, restricted or probabilistic fairness properties with respect to executing runnable threads
- however you can't depend on this.

Spin locks and `Thread` priorities

`Threads` have priorities which heuristically influence schedulers:

- each thread has a priority in the range `Thread.MIN_PRIORITY` to `Thread.MAX_PRIORITY`
- when there are more runnable threads than CPUs, a scheduler is generally biased in favour of threads with higher priorities.
- *typically*, a thread will run until one of the following conditions is true:
  - a higher-priority thread becomes runnable;
  - the thread yields or its `run()` method exits; or
  - on systems that support time-slicing, its quantum has expired.
- *in general*, lower-priority threads will run only when higher-priority threads are blocked (not runnable).

Spin locks in Java (scheduling)

A consequence of Java's weak scheduling guarantees is that spin locks of the form:

```
while (c2 && turn == 2) {
    // do nothing
}
```

may spin *forever*. Even a loop of the form:

```
while (c2 && turn == 2) {
    Thread.yield();
}
```

is not guaranteed to be effective in allowing other threads to execute and change the condition.

The next lecture

Semaphores

Suggested reading:

- Andrews (2000), chapter 4, sections 4.1–4.2;
- Ben-Ari (1982), chapter 4;
- Burns & Davies (1993), chapter 6.