Concurrent Algorithms
Lecture 5 - Algorithms for Mutual Exclusion

Eugene Kenny

eugkenny.lit@gmail.com

Limerick Institute of Technology

October 23rd 2014

Outline

- mutual exclusion protocols
- criteria for a solution
  - safety properties
  - liveness properties
- simple spin lock
- spin lock using turns
- spin lock using the Test-and-Set special instruction
- mutual exclusion with standard instructions
- example: Petersons algorithm
- comparison with the Test-and-Set solution
- spin locks in Java
- Java memory model: atomicity, visibility, ordering
- Petersons algorithm in Java

Archetypical Mutual Exclusion

Any program consisting of n processes for which mutual exclusion is required between critical sections belonging to just one class can be written:

```
// Process 1          // Process 2    ...    // Process n
init1;                init2;                 initn;
while(true) {         while(true) {          while(true) {
  crit1;                crit2;                 critn;
  rem1;                 rem2;                  remn;
}                     }                      }
```

where $\texttt{init}_i$ denotes any (non-critical) initialisation, $\texttt{crit}_i$ denotes a critical section and $\texttt{rem}_i$ denotes the (non-critical) remainder of the program, and i is 1, 2, n.

Archetypical Mutual Exclusion

We assume that `init`, `crit` and `rem` may be of any size:

- `crit` must execute in a finite time
- `init` and `rem` may be infinite.
- `crit` and `rem` may vary from one pass through the while loop to the next

With these assumptions it is possible to rewrite *any* process with critical sections into the archetypical form.

Ornamental Gardens Problem

```
// West turnstile
init1;
while(true) {
  // wait for turnstile
  < count = count + 1; >
  // other stuff ...

}
```

```
// East turnstile
init2;
while(true) {
  // wait for turnstile
  < count = count + 1; >
  // other stuff ...

}
```

```
// Shared datastructures
count == 0
```

Ornamental Gardens Problem

```
// West turnstile

init1;
while(true) {
  // wait for turnstile
  < INCR count; >
  // other stuff ...

}
```

```
// East turnstile

init2;
while(true) {
  // wait for turnstile
  < INCR count; >
  // other stuff ...

}
```

```
// Shared datastructures
count == 0
```

Limitations of Special Instructions

This solution will work if:

- we have a multiprogramming implementation of concurrency (or we can lock memory)
- we have an atomic increment instruction available on the target CPU
- we know how a given high-level program statement will be compiled

However, the range of things you can do with a single atomic action is limited – we cant write a critical section longer than one instruction.

Shared Queue Problem

```
// Process 1

init1
while(true) {
  tail = tail + 1;
  queue[tail] = data1;

  // other code ...
  rem1
}
```

```
// Process 2

init2
while (true) {
  tail = tail + 1;
  queue[tail] = data2;

  // other code ...
  rem2
}
```

```
// Shared datastructures

Object queue[SIZE];
integer tail;
```

Example: Coarse-Grained Atomic Action

```
// Process 1

init1
while(true) {
 < tail = tail + 1;
   queue[tail] = data1; >

  // other code ...
}
```

```
// Process 2

init2
while (true) {
 < tail = tail + 1;
   queue[tail] = data2; >

  // other code ...
}
```

```
// Shared datastructures

Object queue[SIZE];
integer tail;
```

Defining a Mutual Exclusion Protocol

To solve the mutual exclusion problem, we adopt a standard Computer Science approach:

- we design a *protocol* which can be used by concurrent processes to achieve mutual exclusion and avoid interference;
- our protocol will consist of a sequence of instructions which is executed before and possibly after the critical section;
- such protocols can be defined using standard sequential programming primitives, special instructions and what we know about when process switching can happen.

There are many ways to implement such a protocol.

General Form of a Solution

We assume that each of the n processes have the following form, $i = 1, \ldots, n$

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

Shared Queue Problem

```
// Process 1

init1
while(true) {
  // entry protocol
  tail = tail + 1;
  queue[tail] = data1;
  // exit protocol

  // other code ...
}
```

```
// Process 2

init2
while (true) {
  // entry protocol
  tail = tail + 1;
  queue[tail] = data2;
  // exit protocol

  // other code ...
}
```

```
// Shared datastructures

Object queue[SIZE];
integer tail;
```

Correctness of Concurrent Programs

A concurrent program must satisfy two types of property:

- **Safety Properties:** requirements that something should never happen, e.g., failure of mutual exclusion or condition synchronisation, deadlock etc.
- **Liveness Properties:** requirements that something will eventually happen, e.g. entering a critical section.

Note that establishing liveness may require proving safety properties.

Criteria for a Solution

The protocols should satisfy the following properties

- **Mutual Exclusion:** at most one process at a time is executing its critical section
- **Absence of Deadlock (Livelock):** if no process is in its critical section and two or more processes attempt to enter their critical sections, at least one will succeed
- **Absence of Unnecessary Delay:** if a process is trying to enter its critical section and other processes are executing their noncritical sections (or have terminated), the first process is not prevented from entering its critical section
- **Eventual Entry:** a process that is attempting to enter its critical section will eventually succeed

Deadlock vs Livelock

A processes is deadlocked or livelocked when it is unable to make progress because it is waiting for a condition that will never become true

- a deadlocked process is blocked waiting on the condition, e.g, in `wait()` – process does not consume any CPU
- a livelocked process is alive and waiting on the condition, e.g, busy waiting – process does consume CPU

A Simple Spin Lock

```
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;
}
```

Properties of the Simple Spin Lock

Does the simple spin lock satisfy the following properties:

- **Mutual Exclusion:** yes/no
- **Absence of Livelock:** yes/no
- **Absence of Unnecessary Delay:** yes/no
- **Eventual Entry:** yes/no

An Example Trace

```
// Process 1                    // Process 2

init1;                          init2;









}                               }
```

lock == *false*

An Example Trace

```
// Process 1                    // Process 2

init1;                          init2;
while(true) {

}                               }
```

lock == *false*

An Example Trace

```
// Process 1                    // Process 2

init1;                          init2;
while(true)                     while(true)




}                               }
```

lock == *false*

An Example Trace

```
// Process 1                    // Process 2

init1;                          init2;
while(true) {                   while(true)
   ┌─while(lock)
   └─►

}                               }
```

lock == *false*

An Example Trace

```
// Process 1

init1;
while(true) {
    while(lock)



}
```

```
// Process 2

init2;
while(true) {
    while(lock)



}
```

lock == *false*

An Example Trace

```
// Process 1

init1;
while(true) {
    while(lock)
    lock = true;



}
```

```
// Process 2

init2;
while(true) {
    while(lock)



}
```

lock == *true*

An Example Trace

```
// Process 1

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


}
```

```
// Process 2

init2;
while(true) {
     while(lock)


}
```

lock == *true*

An Example Trace

```
// Process 1

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


}
```

```
// Process 2

init2;
while(true) {
    while(lock)
    lock = true;


}
```

lock == *true*

An Example Trace

```
// Process 1

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


}
```

```
// Process 2

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


}
```

lock == *true*

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*

Properties of the Simple Spin Lock

The simple spin lock has the following properties:

- **Mutual Exclusion:** no
- **Absence of Livelock:** yes
- **Absence of Unnecessary Delay:** yes
- **Eventual Entry:** is guaranteed only if the scheduling policy is *strongly fair*.

A *strongly fair* scheduling policy guarantees that if a process requests to enter its critical section infinitely often, the process will *eventually* enter its critical section.

Properties of the Simple Spin Lock

- **Mutual Exclusion:** doesn't hold because there are interleavings which allow both processes to pass their entry protocols
- **Absence of Livelock:** holds because if all processes are outside their critical sections, `lock` must be *false*, and hence (at least) one of the processes will be allowed to enter its critical section
- **Absence of Unnecessary Delay:** holds because if all the other processes are outside their critical sections and stay there, `lock` is *false* and stays *false*, and hence the process that is trying to enter can immediately do so
- **Eventual Entry:** holds because if a process tests `lock` infinitely often, it must eventually see the value *false* - `lock` must become *false* eventually as no process can spend infinitely long in its critical section, so must eventually execute its exit protocol, setting `lock` to *false*

Spin Lock Using Turns

```
// Process 1                          // Process 2

init_1;                               init_2;
while(true) {                         while(true) {
    // entry protocol                     // entry protocol
    while(turn == 2) {};                  while(turn == 1) {};
    crit_1;                               crit_2;
    // exit protocol                      // exit protocol
    turn = 2;                             turn = 1;
    rem_1;                                rem_2;
}                                     }
```

turn == 1

Properties of Round Robin

Does round robin satisfy the following properties:

- **Mutual Exclusion:** yes/no
- **Absence of Livelock:** yes/no
- **Absence of Unnecessary Delay:** yes/no
- **Eventual Entry:** yes/no

Properties of Round Robin

Does round robin satisfy the following properties:

- **Mutual Exclusion:** yes
- **Absence of Livelock:** no
- **Absence of Unnecessary Delay:** no
- **Eventual Entry:** no

Properties of Round Robin

- **Mutual Exclusion:** holds because `turn` can't be both 1 and 2, so at most one process can be in its critical section at any given time
- **Absence of Livelock:** doesn't hold – if there are three processes, one of which has terminated (e.g., in `rem`), then the other two processes may not be able to enter their critical sections
- **Absence of Unnecessary Delay:** fails for two reasons – (1) if any processes terminates outside its critical section, then a process that wants to enter may be unable to do so; (2) even if no process terminates, all processes are constrained to enter their critical sections in order and equally often
- **Eventual Entry:** doesn't hold because the processes can Livelock

Test-and-Set Instruction

The Test-and-Set instruction effectively executes the function

```
bool TS(bool lock) {
    bool v = lock;
    lock = true;
    return v;
}
```

as an atomic action.

Spin Lock Using Test-and-Set

```
// Process i

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

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

An Example Trace

```
// Process 1                    // Process 2

init1;                          init2;




}                               }
```

lock == *false*

An Example Trace

```
// Process 1                    // Process 2

init1;                          init2;
while(true)

}                               }
```

lock == *false*

An Example Trace

```
// Process 1                    // Process 2

init1;                          init2;
while(true)                     while(true)




}                                }
```

lock == *false*

An Example Trace

```
// Process 1                    // Process 2

init1;                          init2;
while(true) {                   while(true) {
   while(TS(lock))

}                               }
```

lock == true

An Example Trace

```
// Process 1                    // Process 2

init1;                          init2;
while(true) {                   while(true) {
   while(TS(lock))                  while(TS(lock))

}                               }
```

lock == *true*

An Example Trace

```
// Process 1                    // Process 2

init1;                          init2;
while(true) {                   while(true) {
   while(TS(lock)) {};              while(TS(lock)) {};

}                               }
```

lock == true

An Example Trace

```
// Process 1                    // Process 2

init1;                          init2;
while(true) {                   while(true) {
    while(TS(lock)) {};             while(TS(lock)) {};
    crit1;

}                               }
```

lock == *true*

An Example Trace

```
// Process 1                    // Process 2

init1;                          init2;
while(true) {                   while(true) {
    while(TS(lock)) {};             while(TS(lock)) {};
    crit1;
    lock = false;

}                               }
```

lock == false

An Example Trace

```
// Process 1                    // Process 2

init1;                          init2;
while(true) {                   while(true) {
    while(TS(lock)) {};             while(TS(lock)) {};
    crit1;                          crit2;
    lock = false;
    rem1;
}                               }
```

lock == true

An Example Trace

```
// Process 1                    // Process 2

init1;                          init2;
while(true) {                   while(true) {
    while(TS(lock)) {};             while(TS(lock)) {};
    crit1;                          crit2;
    lock = false;
    rem1;
}                               }
```

lock == *true*

An Example Trace

```
// Process 1                          // Process 2

init1;                                init2;
while(true) {                         while(true) {
    while(TS(lock)) {};                   while(TS(lock)) {};
    crit1;                                crit2;
    lock = false;
    rem1;
}                                     }
```

lock == *true*

An Example Trace

```
// Process 1                    // Process 2

init1;                          init2;




}                               }
```

lock == *false*

Properties of the Test-and-Set Solution

The solution based on Test-and-Set has the following properties:

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

Solving the Shared Queue Problem

```
// Process 1

init1
while(true) {

  tail = tail + 1;
  queue[tail] = data1;

  // other code ...
}
```

```
// Process 2

init2
while (true) {

  tail = tail + 1;
  queue[tail] = data2;

  // other code ...
}
```

```
// Shared datastructures

Object queue[SIZE];
integer tail;
```

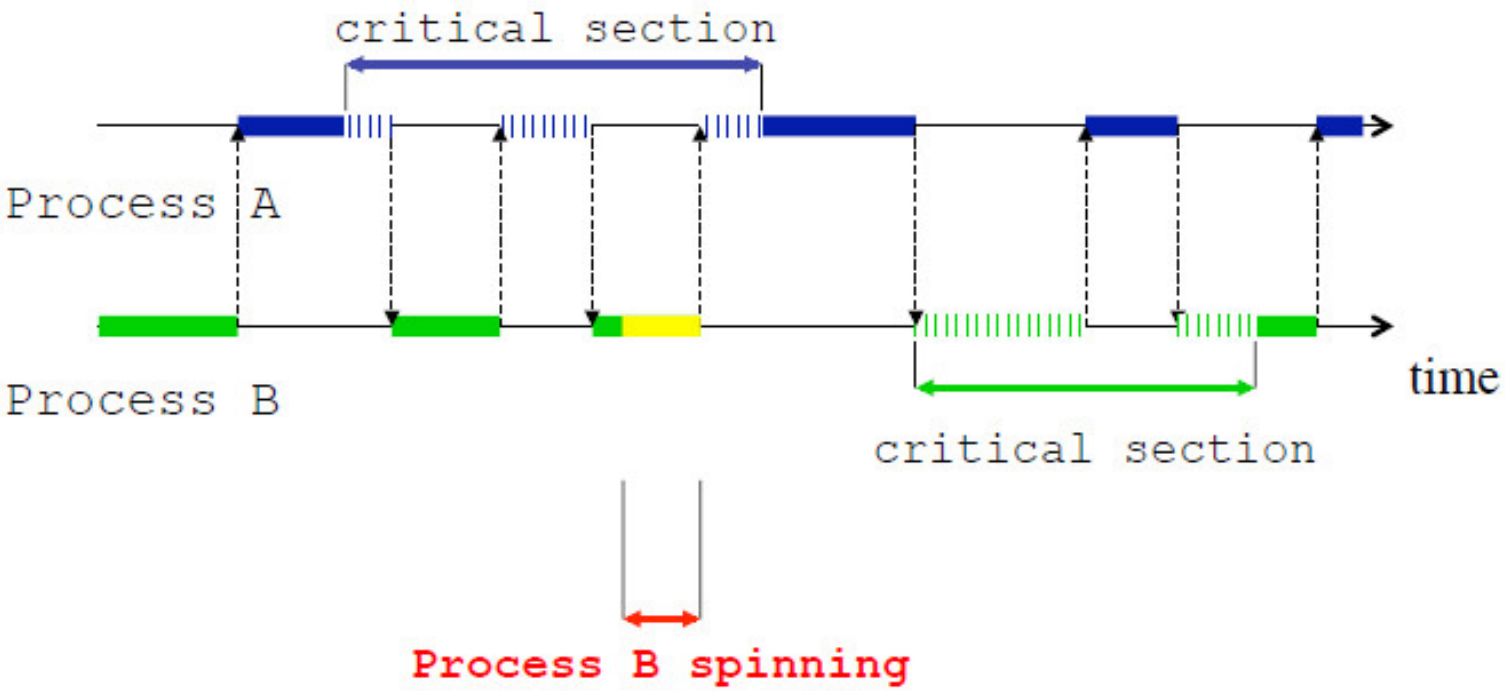
Solving the Shared Queue Problem

```
// Process 1

init1
while(true) {
  while(TS(lock)) {};
  tail = tail + 1;
  queue[tail] = data1;
  lock = false;
  // other code ...
}
```

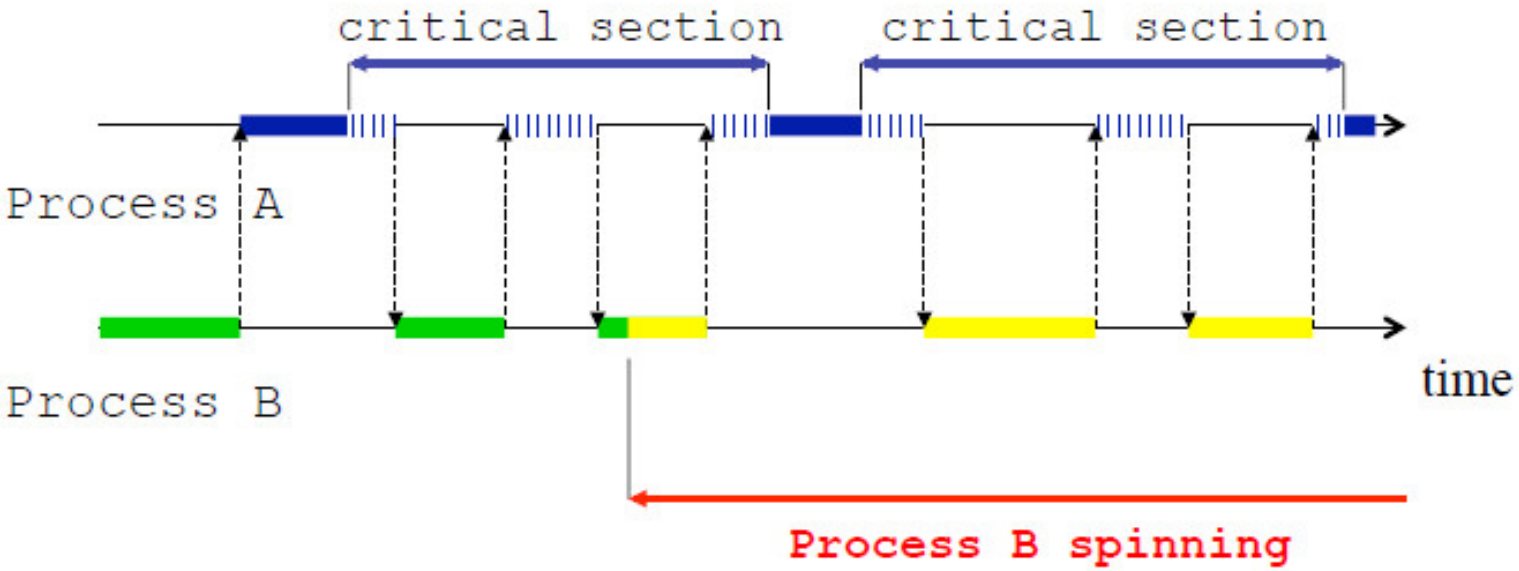
```
// Process 2

init2
while (true) {
  while(TS(lock)) {};
  tail = tail + 1;
  queue[tail] = data2;
  lock = false;
  // other code ...
}
```

```
// Shared datastructures
Object queue[SIZE];
integer tail;
lock = false;
```

Overhead of Spin Locks



Possible Starvation with Spin Locks



Test-and-Set Summary

- Test-and-Set must be atomic
- in a multiprocessing implementation Test-and-Set must effectively lock memory
- if both processes dont try to enter their critical section at the same time neither will have to wait (no *Unnecessary Delay*)
- if there is contention, so long as the critical sections are short the amount of time that each process should have to spend spinning (or *busy waiting*) will be small
- for Eventual Entry, the scheduling policy must be strongly fair
- since all processes execute the same protocol it works for any number of processes

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

```
// Process 1
init1;

}
```

```
// Process 2
init2;

}
```

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

An Example Trace

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

}
```

```
// Process 2
init2;

}
```

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

An Example Trace

```
// 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

```
// 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

```
// Process 1                          // Process 2
init1;                                init2;
while(true) {
    // entry protocol
    c1 = true;
    turn = 2;
    while (c2 && turn == 2) {};




}                                     }
```

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

An Example Trace

```
// Process 1                          // Process 2
init1;                                init2;
while(true) {
     // entry protocol
    c1 = true;
    turn = 2;
    while (c2 && turn == 2) {};
    crit1;



}                                     }
```

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

An Example Trace

```
// 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

```
// 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

```
// 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

```
// 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

```
// 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

```
// 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

```
// 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

```
// 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

```
// 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 doesnt notice that Process 1 has left its critical section?

An Example Trace

```
// 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

```
// 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

```
// 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

```
// 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

```
// 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

```
// 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

```
// 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

```
// 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

```
// 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

```
// 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.

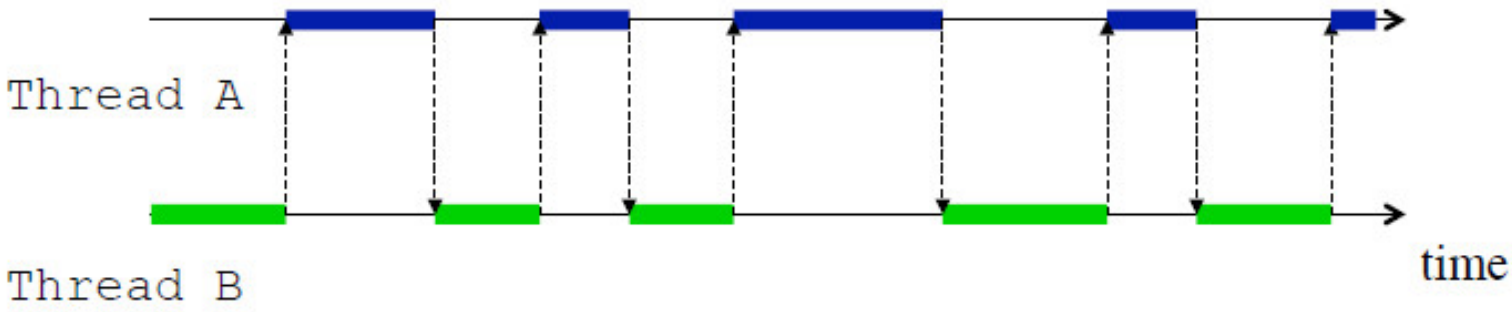
A weakly fair scheduling policy guarantees that if a process requests to enter its critical section (and does not withdraw the request), the process will *eventually* enter its critical section.

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:



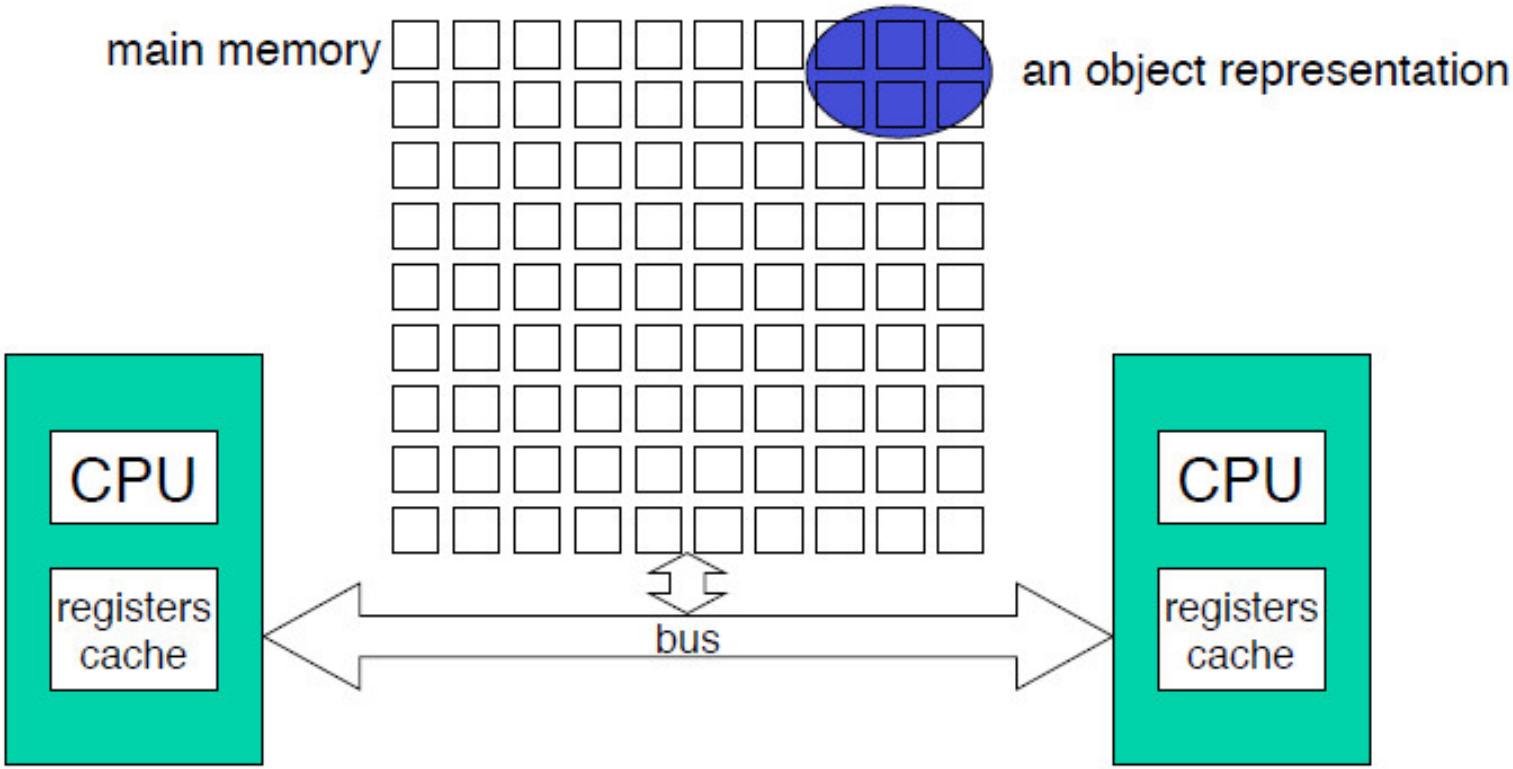
Given a single processor, the JVM executes a sequence of instructions which is an *interleaving* of the instruction sequences for each thread.

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 cant 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 **long**s and **double**s);
- 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; and
- that the scheduling policy is at least *weakly fair*, otherwise eventual entry is not guaranteed.

Archetypical Mutual Exclusion

We have assumed that:

- the initialisation, critical sections and remainder may be of any size and 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 dont execute in finite time, the scheduling policy cant 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 cant depend on this.

Spin :ocks 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 Javas 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.