Peterson's Algorithm

- Peterson's algorithm, also called the tie-breaker algorithm
  - It keeps track of which processes are in their critical sections
  - If two processes try to enter then the a deadlock results
  - This is resolved by letting the last one to start the entry protocol proceed

Peterson's Algorithm

```
// Shared data
boolean in1 = false;
boolean in2 = false;
int last = 1;
```

```
Process1 {
  initialise_1();
  while (true) {
    in1 = true;
    last = 1;
    while ( in2 &&
          (last==1)) {};
    critical_1();
    in1=false;
    non_critical_1();
  }
}
```

```
Process2 {
  intialise_2();
  while (true) {
    in2 = true;
    last = 2;
    while ( in1 &&
          (last==2)) {};
    critical_2();
    in2=false;
    non_critical_2();
  }
}
```


Peterson's Algorithm

- To see how this works we consider the possible situations when Process1 enters its critical section
  - The cases are symmetric for Process2
  - We consider possible stages that Process2 could be at in its protocol as Process1 exists the while loop and enters its critical section

Peterson's Algorithm

- If Process2 is outside of the critical section and exclusion protocol
  - in2 is false, so it is safe for Process1 to proceed
  - If Process2 reaches its critical section before Process1 finishes, then in1 will be true, and last will be 2, so it will wait

```
Process2 {
  intialise_2();
  while (true) {
    in2 = true;
    last = 2;
    while ( in1 &&
          (last==2)) {};
    critical_2();
    in2=false;
    non_critical_2();
  }
}
```


Peterson's Algorithm

- If Process2 has just set in2 to be true
  - in2 is true
  - last will be 1, since Process1 has just set it, and Process2 hasn't yet
  - As a result, Process1 will wait in a spin-lock until…

```
Process2 {
  intialise_2();
  while (true) {
    in2 = true;
    last = 2;
    while ( in1 &&
          (last==2)) {};
    critical_2();
    in2=false;
    non_critical_2();
  }
}
```


Peterson's Algorithm

- If Process2 has just set last to be 2
  - last is 2, so it is safe for Process1 to proceed
  - If Process2 reaches its critical section before Process1 finishes, then in1 will be true, and last will still be 2, so it will wait

```
Process2 {
  intialise_2();
  while (true) {
    in2 = true;
    last = 2;
    while ( in1 &&
          (last==2)) {};
    critical_2();
    in2=false;
    non_critical_2();
  }
}
```

Peterson's Algorithm

- If Process2 evaluating its while loop
  - in1 and in2 are both true
  - last is either 1 or 2, depending on how we got here
  - If last is 1, Process2 will proceed, and Process1 will have to wait
  - If last is 2, Process1 will proceed, and Process2 will have to wait

```
Process2 {
  intialise_2();
  while (true) {
    in2 = true;
    last = 2;
    while ( in1 &&
          (last==2)) {};
    critical_2();
    in2=false;
    non_critical_2();
  }
}
```


Peterson's Algorithm

- If Process2 is in its critical section
  - in1 and in2 are both true
  - Process2 must have seen last == 1 to enter the loop, and hasn't changed it
  - Process1 only ever sets last to 1
  - So last is still 1, and so Process1 waits

```
Process2 {
  intialise_2();
  while (true) {
    in2 = true;
    last = 2;
    while ( in1 &&
          (last==2)) {};
    critical_2();
    in2=false;
    non_critical_2();
  }
}
```


Peterson's Algorithm

- Does this satisfy
  - Mutual exclusion?
    - Yes
  - Absence of livelock?
    - No
  - Absence of unnecessary delay?
    - No
  - Eventual entry?
    - Yes, if scheduler is *weakly fair*

Peterson's Algorithm

- Note that fine detail is important
  - If we swapped the two statements shown, then it doesn't work

```
Process1 {
  initialise_1();
  while (true) {
    in1 = true;
    last = 1;
    while ( in2 &&
          (last==1)) {};
    critical_1();
    in1=false;
    non_critical_1();
  }
}
```


Multiple Processes

- Peterson's Algorithm can be extended to more than 2 processes
  - For *n* processes, each protocol loops through *n*-1 instances of the 2 process protocol
  - If a process makes it through all *n*-1 checks, then it is safe to proceed
  - This is complicated, but can be made to work

Peterson's vs Test-and-Set

- Test-and-Set
  - Provides mutual exclusion, absence of deadlock, and of unnecessary delay
  - Provides eventual entry if scheduler is *strongly fair*
  - Requires special operations
  - Easy to do for >2 processes
- Peterson's
  - Provides mutual exclusion, absence of deadlock, and of unnecessary delay
  - Provides eventual entry if scheduler is *weakly fair*
  - Requires no special operations
  - Difficult to do for >2 processes

The Ticket Algorithm

- Based on the idea of taking a ticket number and waiting at bakeries etc.
  - Each process on entry gets a number from a counter
  - They then wait for their number to come up before entering their critical sections

The Ticket Algorithm

- In this approach
  - There are two shared variables, **counter** and **next**
  - A process gets the value of the counter as its **turn**, then the counter is updated
  - Once the process's **turn** is equal to **next** it can go

```
// Process x
initialise_x();
while (true) {
  turn = counter;
  counter++;
  while(turn!=next){};
  critical_x();
  next++;
  non_critical_x();
}
```


The Ticket Algorithm

- There is a problem
  - Two processes could interfere with each other and get the same turn
  - Can be overcome if we have a fetch-and-add instruction
  - FA(x) returns the current value of x then does x++ atomically

```
// Process x
initialise_x();
while (true) {
  turn = FA(counter);
  while(turn!=next){};
  critical_x();
  next++;
  non_critical_x();
}
```


The Bakery Algorithm

- The ticket algorithm needs a special instruction to work
- However, a variant of it, the Bakery algorithm does not
  - Uses ideas from Peterson's algorithm to deal with two processes having the same number
  - We need to define a comparison between ordered pairs, (a,b) < (c,d), as
    (a< c) || ((a==c) && (b<d))

The Bakery Algorithm

```
// For process i out of n total processes
while (true) {
  choosing[i] = true;
  number[i] = max(number[1]…number[n]) + 1;
  choosing[i] = false;
  for j = 1 to n except i {
    while(choosing[j]) {}
    while((number[j] != 0) &&
          ([number[j],j] < number[i],i)) {}
  }
  critical_i();
  number[i] = 0;
  non_critical_i();
}
```


The Bakery Algorithm

- How it works:
  - Each process has an identifier, **i**
  - To enter a critical section, each process takes a number bigger than all others assigned
  - While taking a number, the process is in the state 'choosing'
  - It then looks at all the other Processes, **j**
  - If they are choosing, it waits until they finish
  - Otherwise, it checks if their numbers are lower
  - In a tie, the process with the lower identifier goes first

The Bakery Algorithm

- This algorithm
  - Meets the requirements of a mutual exclusion protocol
  - Is safe even on multiple processors
  - Turns out to be safe, even if reads and writes to the shared variables interfere with each other – not even these need be atomic actions!

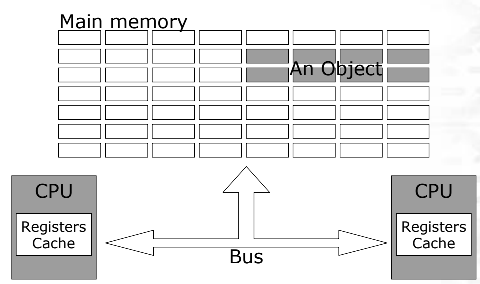
Java

- When a Java program has multiple threads their statements can be interleaved, but
  - They can also be reordered by various things (next slide)
  - (Copies of) values can be kept in caches and registers while being used
  - These things can be done as long as the 'meaning' is unchanged

Ordering of Instructions

- The order of instructions can change
  - By the Java compiler in order to increase efficiency
  - By the CPU when it gets the machine instructions to execute
  - By the memory system when scheduling reads and writes to memory

Java Memory Model



Java Memory Model

- Each thread has a *working memory*
  - This is an abstraction of the registers and caches on the CPU
  - Values can be stored here while being processed, but they are only copies of the value from main memory until any updates are written
  - This improves efficiency

Java Memory Model

- The Java Memory Model specifies when values must be transferred between working and main memory
  - *Atomicity*, when instructions have indivisible effects
  - *Visibility*, when actions of one thread need to be seen by another
  - *Ordering*, when the actions of a thread can be seen to be out of original order

Atomicity

- Reads or writes to memory locations other than longs and doubles
  - Are guaranteed to return the initial value, or some value written by some thread
  - Not guaranteed to return the last value written by the any thread – that is, you could get an out-of-date value

Visibility

- Without synchronisation, there are no guarantees about when one thread's changes are visible to others
  - When a thread accesses an object it sees a value as defined by atomicity
  - When a thread finishes any written values are flushed to main memory

Ordering

- The order of operations may change
  - From a single thread's point of view they appear to occur in the original order (as-if-serial semantics)
  - From another thread's point of view, almost anything can happen

Synchronisation

- If code is properly **`synchronized`** then this is simplified
  - The **`synchronized`** blocks are atomic and visible with respect to any other **`synchronized`** code with the same lock (usually on the same object)
  - Processing of code in **`synchronized`** code is in the program specified order

Peterson's Algorithm

- Peterson's algorithm relies on
  - Atomicity – when it reads and writes variables
  - Visibility – so that processes can see what each other are doing when they are in the entry protocol
  - Ordering – if we set **`in1`** and **`last`** in the wrong order it doesn't work

Spin Locks in Java

- Without synchronization
  - We might have to wait for an arbitrarily long time for values of **`in1`**, **`in2`**, and **`last`** to become visible
  - The order of setting **`in1`** and **`last`** could be changed – the only guarantee is that the thread itself can't tell the difference, and there is no real difference until interference occurs

Java Scheduler Fairness

- Most protocols need the scheduler to be weakly or strongly fair
  - The Java spec makes no guarantees, not even that threads get to make progress
  - Most implementations have some sort of fairness with respect to running threads
  - You can’t rely on it though

Spin Locks and Priority

- Java lets you set the priority of threads
  - In general a high-priority thread gets to go before a low-priority thread
  - If a high-priority thread is in a spin lock, waiting for a low-priority thread to exit its critical section then it might spin forever