Mutual Exclusion

CS511

Example I: Undesired Interleavings

Consider two processes that increment a counter and print the result

```
1  int x=0
2
3  Thread.start { //P
4      x = x + 1
5      print(x)
6  }
7
8  Thread.start { //Q
9      x = x + 1
10     print(x)
11 }
```

What output do you expect?

Atomicity Assumption on Statements

- An instruction such as `x=x+1` is actually compiled into simpler sets of assembly instructions

```
1 iload_1
2 iconst_1
3 iadd
4 istore_1
```

- Could cause undesired interleavings at assembly level

```
1  int x=0
2
3  Thread.start { //P
4      x = x + 1
5      print(x)
6  }
7
8  Thread.start { //Q
9      x = x + 1
10     print(x)
11 }
```

Atomicity Assumption on Statements

Atomic Operation

An operation is atomic if it cannot be interleaved at a lower level of abstraction

- Atomic operations are the smallest units in terms of which a path can be constituted
- In order to reason about concurrency at the source level, we need to know what is assumed atomic at the source level
- Following Ben-Ari: we assume throughout this course that

assignment operations and evaluation of conditions in control statemenst is atomic

Atomicity Assumption on Statements

- Lack of atomicity of assignment (at the source level) can be simulated by decomposing it into several simple instructions.
- For example,

```
counter = counter+1;
```

decomposed into

```
temp = counter+1;
counter = temp;
```

Atomicity Assumption on Statements

- An occurrence of variable v in P is a critical reference if
  1. it is assigned in P and occurs in another process Q, or
  2. it is read in P and is assigned in another process Q.
- A program satisfies the Limited Critical Reference (LCR) property if every statement contains at most one critical reference

```
int x,y=0
Thread.start { //P
    x = x + 1
}
```

```
Thread.start { //Q
    x = x + 1
}
```

```
int x,y=0
Thread.start { //P
    x = y + 1
}
```

```
Thread.start { //Q
    y = x + 1
}
```

```
int x,y=0
Thread.start { //P
    int temp = x + 1
    x = temp
}
```

```
Thread.start { //Q
    int temp = x + 1
    x = temp
}
```

Atomicity Assumption on Statements

- A program satisfies the Limited Critical Reference (LCR) property if every statement contains at most one critical reference
- Concurrent programs that satisfy the LCR restriction yield the same set of behaviors whether the statements are considered atomic or are compiled to a machine architecture with atomic load and store.
- Attempting to present programs that satisfy the LCR restriction is thus convenient

Race Condition

Race Condition

Arises if two or more threads access the same variables or objects concurrently and at least one does updates

- Whether it happens depends on how threads are scheduled
- Once thread T1 starts doing something, it needs to "race" to finish it because if thread T2 looks at the shared variable before T1 is done, it may see something inconsistent
- Hard to detect
- There can be race conditions in programs that satisfy LCR

Example II: Turnstile

- Each thread models a turnstile that updates a shared counter of people going through it
- Each turnstile lets 10 people in

```
1  int counter=0 // shared variable
2
3  P = Thread.start {
4        10.times {
5          counter = counter+1
6        }
7  }
8  Q = Thread.start {
9        10.times {
10         counter= counter+1
11       }
12     }
13
14 P.join() // wait for P to finish
15 Q.join() // wait for Q to finish
16
17 println(counter) // print value of counter
```

Example II in Java

```
1  public class Turnstile implements Runnable{
2    public static int counter = 0;
3    int id;
4    public Turnstile(int esp){ id = esp; }
5    public void run(){
6      Random rnd= new Random();
7      for(int i = 0; i < 10; i++){
8              counter++;
9              System.out.println(id+"- In comes: "+i);
10             try {
11               Thread.sleep(rnd.nextInt(10));
12             } catch (InterruptedException e) {
13               e.printStackTrace();
14             }
15     }
16   }
17   public static void main(String[]  args) {
18     Thread P = new Thread(new Turnstile(1));
19     Thread Q = new Thread(new Turnstile(2));
20     P.start();
21     Q.start();
22     P.join();
23     Q.join();
24     System.out.println(Turnstile.counter);
25   }
26 }
```

Exercise on Paths

```
1  int counter=0;
2
3  Thread.start { // P
4        10.times {
5          int temp = counter+1;
6          counter = temp;
7        }
8  }
9
10 Thread.start { // Q
11       10.times {
12         int temp = counter+1;
13         counter = temp;
14       }
15 }
```

- What are the possible values of `counter` after this program terminates?

Exercise on Paths

```
1  int counter=0;
2
3  Thread.start { // P
4        10.times {
5          int temp = counter+1;
6          counter = temp;
7        }
8  }
9
10 Thread.start { // Q
11       10.times {
12         int temp = counter+1;
13         counter = temp;
14       }
15 }
```

- What are the possible values of `counter` after this program terminates?
- Provide a path where this value is 2

Example II: Bad Path

$$\begin{array}{rc} & (\text{IP\_P}, \text{IP\_Q}, \text{temp\_P}, \text{temp\_Q}, \text{counter}) \\ & (P6, Q6, 0, 0, 0) \\ \rightarrow & (P7, Q6, 1, 0, 0) \\ \rightarrow & (P7, Q7, 1, 1, 0) \\ \rightarrow & (P8, Q7, 1, 1, 1) \\ \rightarrow & (P8, Q8, 1, 1, 1) \end{array}$$

```
1 int counter=0
2
3 // Thread P
4 Thread.start {
5       10.times {
6         int temp = counter+1
7         counter = temp
8       }
9 }
```

Race Conditions, Atomicity

Critical Sections

An Advanced Algorithm

Critical Section

Critical Section

A part of the program that accesses shared memory and which we wish to execute atomically

```
1 Thread.start { // P
2   while(true) {
3   // non-critical section
4   entry to critical section;
5   // CRITICAL SECTION
6   exit from critical section
7   // non-critical section
8   }
9 }
```

```
1 Thread.start { // Q
2   while(true) {
3   // non-critical section
4   entry to critical section
5   // CRITICAL SECTION
6   exit from critical section
7   // non-critical section
8   }
9 }
```

The Mutual Exclusion Problem (MEP)

The Mutual Exclusion Problem

Guarantee that:

1. Mutex: At any point in time, there is at most one thread in the critical section
2. Absence of livelock: If various threads try to enter the critical section, at least one of them will succeed
3. Free from starvation: A thread trying to enter its critical section will eventually be able to do so

- Brief historical account:
  Leslie Lamport: Turing lecture: The computer science of concurrency: the early years. Commun. ACM 58(6): 71-76 (2015)

Reasonable Assumptions for MEP

- There are no shared variables between the critical section and the non critical section (nor with the entry/exit protocol).
- The critical section always terminates.
- The scheduler is fair.
  - There are various notions of fairness
  - Here we mean: A process that is waiting to run, will be able to do so eventually

```
1 Thread.start { // P
2   while(true) {
3   // non-critical section
4   entry to critical section;
5   // CRITICAL SECTION
6   exit from critical section
7   // non-critical section
8   }
9 }
```

```
1 Thread.start { // Q
2   while(true) {
3   // non-critical section
4   entry to critical section
5   // CRITICAL SECTION
6   exit from critical section
7   // non-critical section
8   }
9 }
```

What is Required to Solve the MEP Problem?

- In theory, mechanisms involving shared variables suffice
  - We are going to see numerous examples
  - However: Too low-level, easy to get wrong
- In practice:
  - PLs provide high-level abstractions that serve for this, and other, purposes (eg. semaphores, monitors, etc.)
  - Communication without shared variables (eg. message passing)
  - Hardware support (special atomic instructions)

General Scheme to Address the MEP

```
1  shared variables;
2
3  Thread.start  {
4    while(true) {
5        // non-critical section
6        entry to critical section
7        // CRITICAL SECTION
8        exit from critical section
9        // non-critical section
10   }
11 }
```

Abbreviation:

```
while (cond) {}   ⟶   await !cond
```

Attempt I – Take Turns

```
int turn = 1;
```

```
1 Thread.start  { // P
2   // non-critical section
3   await (turn==1);
4   // CRITICAL SECTION
5   turn = 2;
6   // non-critical section
7 }
```

```
1 Thread.start { // Q
2   // non-critical section
3   await (turn==2);
4   // CRITICAL SECTION
5   turn = 1;
6   // non-critical section
7 }
```

- `turn` is a "permission resource"
- Mutex:
- Absence livelock:
- Free from starvation:

Attempt I – Take Turns

```
int turn = 1;
```

```
1 Thread.start  { // P
2   // non-critical section
3   await (turn==1);
4   // CRITICAL SECTION
5   turn = 2;
6   // non-critical section
7 }
```

```
1 Thread.start { // Q
2   // non-critical section
3   await (turn==2);
4   // CRITICAL SECTION
5   turn = 1;
6   // non-critical section
7 }
```

- `turn` is a "permission resource"
- Mutex: Yes
- Absence livelock:
- Free from starvation:

Attempt I – Take Turns

```
int turn = 1;
```

```
1 Thread.start  { // P
2   // non-critical section
3   await (turn==1);
4   // CRITICAL SECTION
5   turn = 2;
6   // non-critical section
7 }
```

```
1 Thread.start { // Q
2   // non-critical section
3   await (turn==2);
4   // CRITICAL SECTION
5   turn = 1;
6   // non-critical section
7 }
```

- `turn` is a "permission resource"
- Mutex: Yes
- Absence livelock: Yes
- Free from starvation:

Attempt I – Take Turns

```
int turn = 1;
```

```
1 Thread.start  { // P
2   // non-critical section
3   await (turn==1);
4   // CRITICAL SECTION
5   turn = 2;
6   // non-critical section
7 }
```

```
1 Thread.start { // Q
2   // non-critical section
3   await (turn==2);
4   // CRITICAL SECTION
5   turn = 1;
6   // non-critical section
7 }
```

- `turn` is a "permission resource"
- Mutex: Yes
- Absence livelock: Yes
- Free from starvation: No

Attempt I – Take Turns

```
int turn = 1;
```

```
1 Thread.start  { // P
2   // non-critical section
3   await (turn==1);
4   // CRITICAL SECTION
5   turn = 2;
6   // non-critical section
7 }
```

```
1 Thread.start { // Q
2   // non-critical section
3   await (turn==2);
4   // CRITICAL SECTION
5   turn = 1;
6   // non-critical section
7 }
```

- `turn` is a "permission resource"
- Mutex: Yes
- Absence livelock: Yes
- Free from starvation: No (a process could remain indefinitely in its non-critical section)

Attempt I – Take Turns

```
int turn = 1;
```

```
1 Thread.start  { // P
2   // non-critical section
3   await (turn==1);
4   // CRITICAL SECTION
5   turn = 2;
6   // non-critical section
7 }
```

```
1 Thread.start { // Q
2   // non-critical section
3   await (turn==2);
4   // CRITICAL SECTION
5   turn = 1;
6   // non-critical section
7 }
```

- `turn` is a "permission resource"
- Mutex: Yes
- Absence livelock: Yes
- Free from starvation: No (a process could remain indefinitely in its non-critical section)

How do we know mutex and absence of livelock hold for sure?

Attempt I – Take Turns

- Mutex: Yes
- Absence livelock: Yes

How do we know these properties hold for sure?

- We can resort to an analysis of transition systems or prove it using deductive systems
- For now we choose the former
- Let's build the transition system

Attempt I – Take Turns

```
int turn = 1;
```

```
1 Thread.start { // P
2   while (true) {
3       // non-critical section
4       await (turn==1);
5       // CRITICAL SECTION
6       turn = 2;
7       // non-critical section
8   }
9 }
```

```
1 Thread.start { // Q
2  while (true) {
3     // non-critical section
4     await (turn==2);
5     // CRITICAL SECTION
6     turn = 1;
7     // non-critical section
8    }
9 }
```

Before drawing the transition system, we get rid of irrelevant commands: we are only interested in synchronization

Attempt I – Take Turns

```
int turn = 1;
```

```
1 Thread.start { // P
2   while (true) {
3       await (turn==1);
4       turn = 2;
5   }
6 }
```

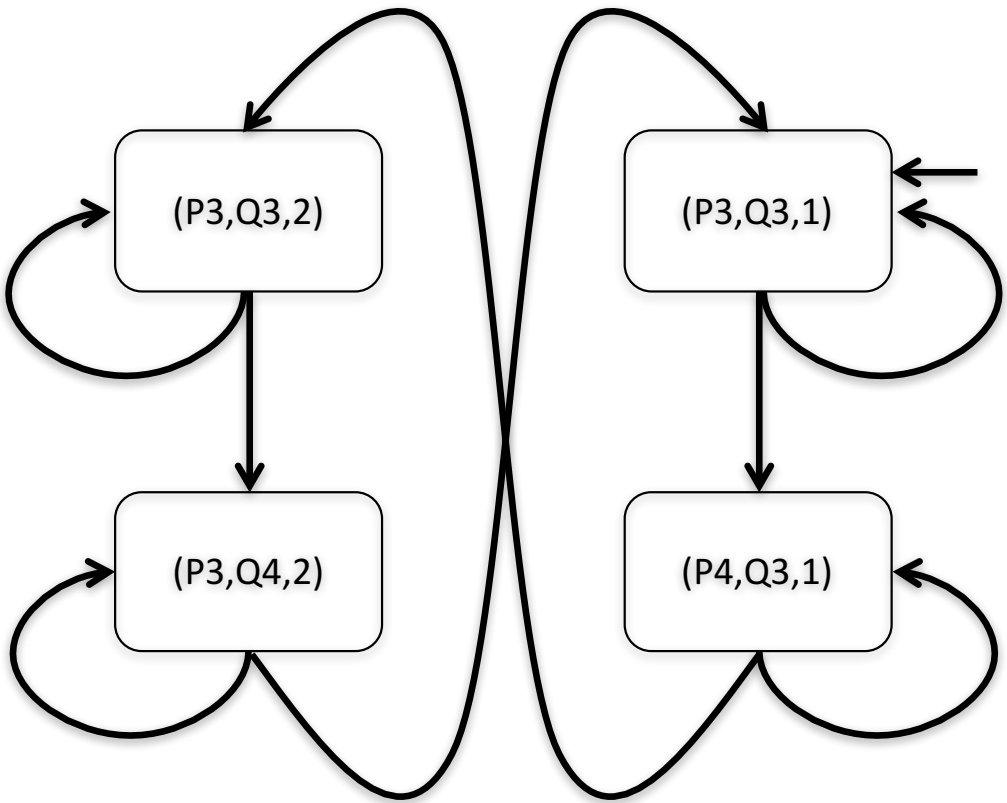
```
1 Thread.start { // Q
2  while (true) {
3     await (turn==2);
4     turn = 1;
5    }
6 }
```

Attempt I – Take Turns

```
int turn = 1;
```

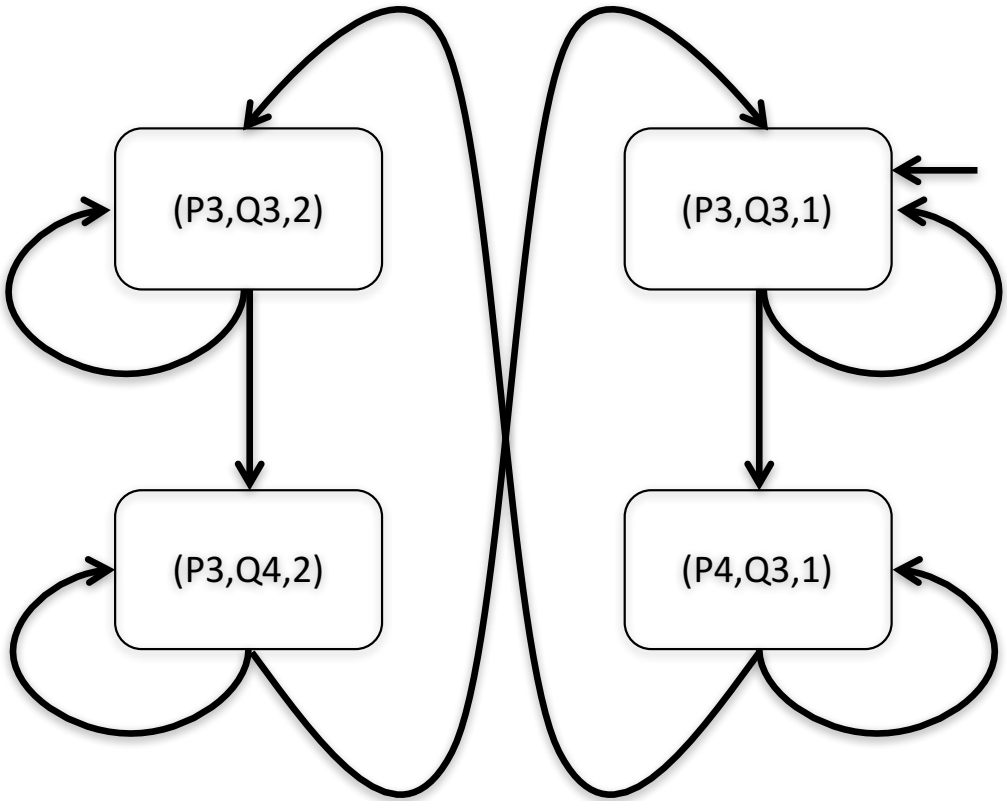
```
1 Thread.start { // P
2   while (true) {
3       await (turn==1);
4       turn = 2;
5   }
6 }
```

```
1 Thread.start { // Q
2  while (true) {
3      await (turn==2);
4      turn = 1;
5  }
6 }
```



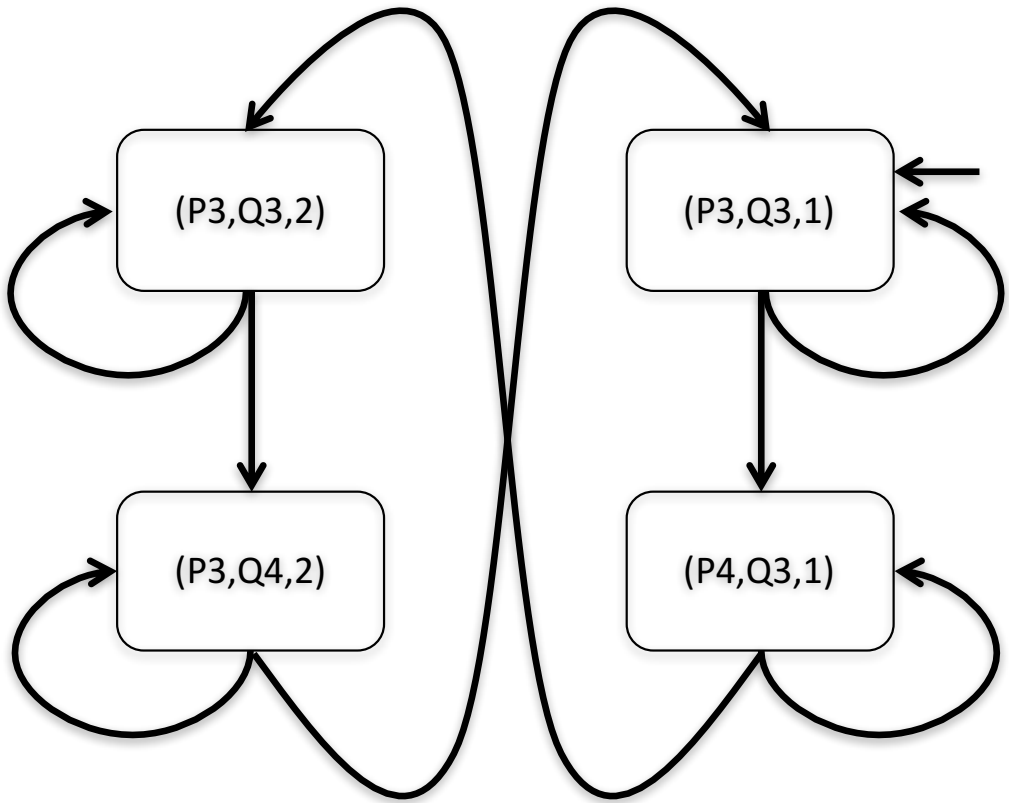
Proving the Desired Properties I

- Mutex: Holds if all accesible states do no contain a state of the form $(p4, q4, turn)$ for some value of turn



Proving the Desired Properties II

- Absence of livelock (if two or more processes are trying to enter their CS, i.e. executing an await, then at least one must eventually succeed)
- Recall that the awaits are on line 3



Clarification on livelock and transition systems

Absence of livelock (AoL) means that if two or more processes are trying to enter their CS, then at least one must eventually succeed

- If our processes could block when trying to access the CS, then we would speak of absence of deadlock: there are no states that are not endstates and that have no outgoing edges
- If processes cannot block (as is our case for now) we talk about absence of livelock rather than AoD. Livelock means that even though processes are not blocked (i.e. in a waiting state) they make no progress (i.e., there is no path leading to the CS)

Proving the Desired Properties III

Freedom from starvation (if a process continuously tries to access the CS, it will eventually be able to do so)

- Consider what happens if Q is trying to access its critical section but $turn = 1$ and P fails or loops in its NCS (*)

```
int turn = 1;
```

```
1 Thread.start { // P
2   while (true) {
3       // non-critical section (*)
4       await (turn==1);
5       // CRITICAL SECTION
6       turn = 2;
7       // non-critical section
8   }
9 }
```

```
1 Thread.start { // Q
2  while (true) {
3     // non-critical section
4     await (turn==2);
5     // CRITICAL SECTION
6     turn = 1;
7     // non-critical section
8    }
9 }
```

Attempt I – Assessment

- Starvation is due to the fact that both processes test and set a single shared variable
- If a process dies, the other is blocked
- Let us try to give each process its own variable to indicate that it wants to enter the CS

Attempt II

```
boolean wantP = false;
boolean wantQ = false;

Thread.start  { // P
 while (true) {
  // non-critical section
   await !wantQ;
   wantP = true;
   // CRITICAL SECTION
   wantP = false;
   // non-critical section
 }
}
```

```
Thread.start { // Q
  while (true) {
   // non-critical section
   await !wantP;
   wantQ = true;
   // CRITICAL SECTION
   wantQ = false;
   // non-critical section
 }
}
```

- Mutex:

Attempt II

```
boolean wantP = false;
boolean wantQ = false;

Thread.start  { // P
 while (true) {
  // non-critical section
   await !wantQ;
   wantP = true;
   // CRITICAL SECTION
   wantP = false;
   // non-critical section
 }
}
```

```
Thread.start { // Q
  while (true) {
   // non-critical section
   await !wantP;
   wantQ = true;
   // CRITICAL SECTION
   wantQ = false;
   // non-critical section
 }
}
```

- Mutex: No

Attempt III

```
boolean wantP = false;
boolean wantQ = false;
```

```
Thread.start  { // P
  while (true) {
   // non-critical section
   wantP = true;
   await !wantQ;
   // CRITICAL SECTION
   wantP = false;
   // non-critical section
  }
}
```

```
Thread.start { // Q
  while (true) {
   // non-critical section
   wantQ = true;
   await !wantP;
   // CRITICAL SECTION
   wantQ = false;
   // non-critical section
  }
}
```

- Mutex:
- Absence livelock:
- Free from starvation:

Attempt III

```
boolean wantP = false;
boolean wantQ = false;

Thread.start  { // P
  while (true) {
   // non-critical section
   wantP = true;
   await !wantQ;
   // CRITICAL SECTION
   wantP = false;
   // non-critical section
  }
}
```

```
Thread.start { // Q
  while (true) {
   // non-critical section
   wantQ = true;
   await !wantP;
   // CRITICAL SECTION
   wantQ = false;
   // non-critical section
  }
}
```

- Mutex: Yes
- Absence livelock:
- Free from starvation:

Attempt III

```
boolean wantP = false;
boolean wantQ = false;

Thread.start  { // P
  while (true) {
   // non-critical section
   wantP = true;
   await !wantQ;
   // CRITICAL SECTION
   wantP = false;
   // non-critical section
  }
}
```

```
Thread.start { // Q
  while (true) {
   // non-critical section
   wantQ = true;
   await !wantP;
   // CRITICAL SECTION
   wantQ = false;
   // non-critical section
  }
}
```

- Mutex: Yes
- Absence livelock: No
- Free from starvation:

Attempt III

```
boolean wantP = false;
boolean wantQ = false;

Thread.start  { // P
  while (true) {
   // non-critical section
   wantP = true;
   await !wantQ;
   // CRITICAL SECTION
   wantP = false;
   // non-critical section
  }
}
```

```
Thread.start { // Q
  while (true) {
   // non-critical section
   wantQ = true;
   await !wantP;
   // CRITICAL SECTION
   wantQ = false;
   // non-critical section
  }
}
```

- Mutex: Yes
- Absence livelock: No
- Free from starvation: No

Attempt III – Assessment

```
boolean wantP = false;
boolean wantQ = false;

Thread.start  { // P
  while (true) {
   // non-critical section
   wantP = true;
   await !wantQ;
   // CRITICAL SECTION
   wantP = false;
   // non-critical section
  }
}
```

```
Thread.start { // Q
  while (true) {
   // non-critical section
   wantQ = true;
   await !wantP;
   // CRITICAL SECTION
   wantQ = false;
   // non-critical section
  }
}
```

- Processes should:
  1. either back out if they discover they are contending with the other (Naive back-out);
  2. take turns (Dekker's algorithm);
  3. give priority to the first one that wanted access (Peterson's algorithm).
- We consider all these options next

Attempt IV – Naive Back-Out

```
boolean wantP = false;
boolean wantQ = false;
```

```
Thread.start { //
  while (true) {
   // non-critical section
   wantP = true;
   while wantQ {
     wantP = false;
     wantP = true;
   }
   // CRITICAL SECTION
   wantP = false;
   // non-critical section
  }
}
```

```
Thread.start  { // Q
  while (true) {
   // non-critical section
   wantQ = true;
   while wantP {
     wantQ = false;
     wantQ = true;
   }
   // CRITICAL SECTION
   wantQ = false;
   // non-critical section
  }
}
```

- Mutex:
- Absence livelock:
- Free from starvation:

Attempt IV – Naive Back-Out

```
boolean wantP = false;
boolean wantQ = false;
```

```
Thread.start { //
  while (true) {
   // non-critical section
   wantP = true;
   while wantQ {
     wantP = false;
     wantP = true;
   }
   // CRITICAL SECTION
   wantP = false;
   // non-critical section
  }
}
```

```
Thread.start  { // Q
  while (true) {
   // non-critical section
   wantQ = true;
   while wantP {
     wantQ = false;
     wantQ = true;
   }
   // CRITICAL SECTION
   wantQ = false;
   // non-critical section
  }
}
```

- Mutex: Yes
- Absence livelock:
- Free from starvation:

Attempt IV – Naive Back-Out

```
boolean wantP = false;
boolean wantQ = false;

Thread.start { //
  while (true) {
   // non-critical section
   wantP = true;
   while wantQ {
     wantP = false;
     wantP = true;
   }
   // CRITICAL SECTION
   wantP = false;
   // non-critical section
  }
}
```

```
Thread.start  { // Q
  while (true) {
   // non-critical section
   wantQ = true;
   while wantP {
     wantQ = false;
     wantQ = true;
   }
   // CRITICAL SECTION
   wantQ = false;
   // non-critical section
  }
}
```

- Mutex: Yes
- Absence livelock: No
- Free from starvation:

Attempt IV – Naive Back-Out

```
boolean wantP = false;
boolean wantQ = false;

Thread.start { //
  while (true) {
   // non-critical section
   wantP = true;
   while wantQ {
     wantP = false;
     wantP = true;
   }
   // CRITICAL SECTION
   wantP = false;
   // non-critical section
  }
}
```

```
Thread.start  { // Q
  while (true) {
   // non-critical section
   wantQ = true;
   while wantP {
     wantQ = false;
     wantQ = true;
   }
   // CRITICAL SECTION
   wantQ = false;
   // non-critical section
  }
}
```

- Mutex: Yes
- Absence livelock: No
- Free from starvation: No

Dekker's Algorithm (1965)

- Combines I and IV
- Attributed to Theodorus Jozef Dekker by Edsger W. Dijkstra in 1965

Dekker's Algorithm (I + IV)

```
int turn = 1;
boolean wantP = false;
boolean wantQ = false;

Thread.start  { // P
  while (true) {
   // non-critical section
   wantP = true;
   while wantQ
     if (turn == 2) {
       wantP = false;
       await (turn==1);
       wantP = true;
     }
   // CRITICAL SECTION
   turn = 2;
   wantP = false;
   // non-critical section
  }
}
```

```
Thread.start  { // Q
  while (true) {
   // non-critical section
   wantQ = true;
   while wantP
     if (turn == 1) {
       wantQ = false;
       await (turn==2);
       wantQ = true;
     }
   // CRITICAL SECTION
   turn = 1;
   wantQ = false;
   // non-critical section
  }
}
```

Right to insist on entering is passed between the two processes

Peterson's Algorithm (1981)

```
int last = 1;
boolean wantP = false;
boolean wantQ = false;

Thread.start  { // P
  while (true) {
   // non-critical section
   wantP = true;
   last = 1;
   await !wantQ or last==2;
   // CRITICAL SECTION
   wantP = false;
   // non-critical section
  }
}
```

```
Thread.start  { // Q
  while (true) {
   // non-critical section
   wantQ = true;
   last = 2;
   await !wantP or last==1;
   // CRITICAL SECTION
   wantQ = false;
   // non-critical section
  }
}
```

Similar to Dekker except that if both want access, priority is given to the first one that wanted to access

Dekker and Peterson

- Mutex: Yes
- Absence livelock: Yes
- Free from starvation: Yes

Race Conditions, Atomicity

Critical Sections

An Advanced Algorithm

The Bakery Algorithm

- Developed by Leslie Lamport
- Think of a Bakery shop
  - People take a ticket from a machine
  - If nobody is waiting, tickets don't matter
  - When several people are waiting, ticket order determines order in which they can make purchases
- Process interested in entering critical section acquires a ticket whose value is greater than all outstanding tickets
- Waits until its ticket is lowest value of all outstanding tickets

Bakery Algorithm – A Simplified Version for Two Processes

```
int np,nq =0;

Thread.start {  // P
  while (true) {
   // non-critical section
   [np = nq + 1];
   await nq==0 or np<=nq;
   // CRITICAL SECTION
   np = 0;
   // non-critical section
  }
}
```

```
Thread.start  { // Q
  while (true) {
   // non-critical section
   [nq = np + 1];
   await np==0 or nq<np;
   // CRITICAL SECTION
   nq = 0;
   // non-critical section
  }
}
```

- Assignment is assumed atomic
  - This is not realistic but is required for otherwise mutex fails (see exercise booklet 2)
  - This assumption will be dropped in the final Bakery Algorithm
- Disadvantage for implementation:
  - Ticket numbers (held in `np` and `nq`) can be unbounded

The Bakery Algorithm – Almost the Final Version

```
int[] number = new int[n]; // {0,0,...0}

Thread.start {
   while (true) {
    // non-critical section
    [number[id] = 1 + maximum(number);]
    for all other processes j
      await (number[j]=0) or (number[id]<<number[j])
    // CRITICAL SECTION
    number[id] = 0;
    // non-critical section
  }
}
```

`number[i]<<number[j]` abbreviates

```
(number[i]<number[j]) or (number[i]==number[j] and i<j)
```

- Ticket numbers still unbounded
- Assumes assignment and computing the maximum of an array is atomic
  - Otherwise mutex fails
  - We get rid of this assumption in the final version

The Bakery Algorithm

```
boolean[] choosing = new boolean[n]; // {false,..,}
int[] number = new int[n]; // {0,0,...0}

Thread.start {
  // non-critical section
  choosing[id] = true;
  number[id] = 1 + maximum(number);
  choosing[id] = false;
  for (all other processes j) {
    await !choosing[j];
    await (number[j] == 0) or (number[id]<<number[j]);
  }

  // CRITICAL SECTION

  number[id] = 0;
  // non-critical section
}
```

This algorithm solves the problem for n threads.

Bakery Algorithm - Assessment

- Solves the MEP problem
- Drawbacks
  - Unbounded ticket numbers
  - Inefficient if there is no contention
    - a process has to query all other processes with an await even if it is the only one wanting to enter
    - This drawback was addressed here:

      Leslie Lamport: A Fast Mutual Exclusion Algorithm. ACM Trans. Comput. Syst. 5(1): 1-11 (1987)

Summary

- Difficult to solve the MEP using just atomic load and store
- If an atomic statement for both load and store were available, we could provide much easier solutions
- We'll see examples of this when we introduce such atomic statements next class