Mutual Exclusion

CS511

Example I: Undesired Interleavings

Consider two processes that increment a counter and print the result

```
global int x=0;

thread P: {
    x = x + 1;
    print(x);
}

thread Q: {
    x = x + 1;
    print(x);
}
```

What output do you expect? Let's run it and see what happens...

Atomicity Assumption on Assignments

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

```
load x,r
add r,1
write r,x
```

- Could cause undesired interleavings at assembly level

```
global int x=0;

thread P: {
    x = x + 1;
    print(x);
}

thread Q: {
    x = x + 1;
    print(x);
}
```

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

Atomicity Assumption on Assignments

Atomic Operation

An operation is atomic if it is executed until completion without interleaving statements from other processes

- Atomic operations are the smallest unit in which a path can be listed
- We assume throughout this course that assignment statements are atomic for scalar values (i.e. numbers, booleans, chars, etc.)
- Eg. `x=2` is atomic but `x=x+1` is not.

Atomicity Assumption on Assignments

- Unfortunately, even if assignments such as `x=x+1` were considered atomic, there are still situations where unwanted interleavings can appear
- Indeed, an easy example is obtained by "simulating" the problem with the counters using a temporary variable

Example II: Turnstile

- Consider two turnstiles to control access to a room
  - People can enter through any of them
  - People arrive at random times
  - We will assume that the room has capacity for 100 assistants
- Write a program in which the entering cycle, for each turnstile, runs in its own thread
- There should be a variable that counts the total number of people that entered

Example II: Turnstile

- Lets look at the code for one turnstile
- We assume a global variable `counter`
- Also, we assume that each turnstile only lets at most 50 people pass

```
global int counter=0;

thread P: {
      for(int j = 0; j < 50; j++){
        print("1: In comes "+ j);
        int temp = counter+1;
        counter = temp;
      }
      print("1: total " + counter);
    }
```

Example II in Hydra

```
global int counter=0;

thread P: {
       for(int j = 0; j < 50; j++){
         print("1: In comes "+ j);
         int temp = counter+1;
         counter = temp;
       }
       print("1: total " + counter);
}
thread Q: {
       for(int j = 0; j < 50; j++){
         print("2:In comes "+ j);
         int temp = counter+1;
         counter = temp;
       }
       print("2: total " + counter);
     }
```

Example II: Bad Path

(IP Thread1, IPThread2, temp Thread1, temp Thread2, counter)

$$\begin{aligned} & (P6, Q6, 0, 0, 0) \\ \rightarrow\ & (P7, Q6, 1, 0, 0) \\ \rightarrow\ & (P7, Q7, 1, 1, 0) \\ \rightarrow\ & (P8, Q7, 1, 1, 1) \\ \rightarrow\ & (P8, Q8, 1, 1, 1) \end{aligned}$$

```
1  global int counter=0;
2
3  thread P: {
4        for(int j = 0; j < 50; j++){
5          print("1: In comes "+ j);
6          int temp = counter+1;
7          counter = temp;
8        }
9        print("1: total " + counter);
10     }
```

Example II in Java

```
public class Example2 implements Runnable{
  public static int counter = 0;
  int id;
  public Example2(int esp){ id = esp; }
  public void run(){
    Random rnd= new Random();
    for(int i = 0; i < 50; i++){
            int temp = ++counter;
            counter = temp;
            System.out.println(id+"- In comes: "+i);
            try {
              Thread.sleep(rnd.nextInt(10));
            } catch (InterruptedException e) {
              e.printStackPath();
            }
    }
    System.out.println(id+"- Total incoming: "+ counter);
  }

  public static void main(String args[])
  {
    new Thread(new Example2(1)).start();
    new Thread(new Example2(2)).start();
  }
}
```

Exercise on Paths

```
global int counter=0;

thread P: {
      for(int j = 0; j < 10; j++){
        int temp = counter+1;
        counter = temp;
      }
    }

thread Q: {
      for(int j = 0; j < 10; j++){
        int temp = counter+1;
        counter = temp;
      }
    }
```

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

Exercise on Paths

```
global int counter=0;

thread P: {
      for(int j = 0; j < 10; j++){
        int temp = counter+1;
        counter = temp;
      }
    }

thread Q: {
      for(int j = 0; j < 10; j++){
        int temp = counter+1;
        counter = temp;
      }
    }
```

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

Properties of Paths

A property is an assertion that is true for every possible path

- Safety: A path never reaches a “bad” state
- Liveness: Eventually, every path will reach a “good” state

Synchronization

Mechanism that restricts the possible paths of a concurrent program in order to ensure safety and liveness properties.

- An example of a safety property follows

Critical Section

Critical Section

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

Mutual Exclusion

The problem of ensuring that two threads do not execute a critical section simultaneously.

```
thread P: {
  while(true) {
  // non-critical section
  entry to critical section;
  // CRITICAL SECTION
  exit from critical section
  // non-critical section
  }
}
```

```
thread Q: {
  while(true) {
  // non-critical section
  entry to critical section
  // CRITICAL SECTION
  exit from critical section
  // non-critical section
  }
}
```

Critical Section

Critical Section

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

Mutual Exclusion

The problem of ensuring that two threads do not execute a critical section simultaneously.

```
thread P: {
  while(true) {
  // non-critical section
  entry to critical section;
  // CRITICAL SECTION
  exit from critical section
  // non-critical section
  }
}
```

```
thread Q: {
  while(true) {
  // non-critical section
  entry to critical section
  // CRITICAL SECTION
  exit from critical section
  // non-critical section
  }
}
```

Critical Section

Critical Section

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

Mutual Exclusion

The problem of ensuring that two threads do not execute a critical section simultaneously.

```
thread P: {
  while(true) {
  // non-critical section
  entry to critical section;
  // CRITICAL SECTION
  exit from critical section
  // non-critical section
  }
}
```

```
thread Q: {
  while(true) {
  // non-critical section
  entry to critical section
  // CRITICAL SECTION
  exit from critical section
  // non-critical section
  }
}
```

Critical Section

```
thread P: {
  while(true) {
  // non-critical section
  entry to critical section;
  // CRITICAL SECTION
  exit from critical section
  // non-critical section
  }
}
```

```
thread Q: {
  while(true) {
  // non-critical section
  entry to critical section
  // CRITICAL SECTION
  exit from critical section
  // non-critical section
  }
}
```

Assumptions:

- 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

The Mutual Exclusion Problem (MEP)

1. Mutex: At any point in time, there is at most one thread in the critical section
2. Absence of deadlock: 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)

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

```
global variables;

thread  {
  ~~while(true) {~~
      // non-critical section
      entry to critical section
      // CRITICAL SECTION
      exit from critical section
      // non-critical section
  ~~}~~
}
```

Question

Can we solve the MEP for two processes assuming that the only atomic operations are

- read and
- write on variables?

Attempt 0 – `flag` to indicate whether someone is in the CS

```
global boolean flag = false;

thread P: {                         thread Q: {
  // non-critical section             // non-critical section
  while (flag) {};                     while (flag) {};
  flag = true;                         flag = true;
  // CRITICAL SECTION                  // CRITICAL SECTION
  flag = false;                        flag = false;
  // non-critical section             // non-critical section
}                                    }
```

Attempt 0 – `flag` to indicate whether someone is in the CS

```
global boolean flag = false;
```

```
thread P: {                        thread Q: {
  // non-critical section            // non-critical section
  while (flag) {};                   while (flag) {};
  flag = true;                       flag = true;
  // CRITICAL SECTION                // CRITICAL SECTION
  flag = false;                      flag = false;
  // non-critical section            // non-critical section
}                                  }
```

- `while (cond) { };` is called a busy-wait loop
- Abbreviation:

$$\texttt{while (cond) \{\}} \longrightarrow \texttt{await !cond}$$

- Let's apply this to the above example

Attempt 0 – `flag` to indicate whether someone is in the CS

```
global boolean flag = false;

thread P: { //                 thread Q: {
  // non-critical section        // non-critical section
  await !flag;                   await !flag;
  flag = true;                   flag = true;
  // CRITICAL SECTION            // CRITICAL SECTION
  flag = false;                  flag = false;
  // non-critical section        // non-critical section
}                              }
```

- Mutex:
- Absence deadlock:
- Free from starvation:

Attempt 0 – `flag` to indicate whether someone is in the CS

```
global boolean flag = false;

thread P: { //                    thread Q: {
  // non-critical section           // non-critical section
  await !flag;                      await !flag;
  flag = true;                      flag = true;
  // CRITICAL SECTION               // CRITICAL SECTION
  flag = false;                     flag = false;
  // non-critical section           // non-critical section
}                                 }
```

- Mutex: No
- Absence deadlock:
- Free from starvation:

Attempt 0 – `flag` to indicate whether someone is in the CS

```
global boolean flag = false;

thread P: { //                  thread Q: {
  // non-critical section         // non-critical section
  await !flag;                    await !flag;
  flag = true;                    flag = true;
  // CRITICAL SECTION             // CRITICAL SECTION
  flag = false;                   flag = false;
  // non-critical section         // non-critical section
}                               }
```

- Mutex: No
- Absence deadlock: Don't care
- Free from starvation:

Attempt 0 – `flag` to indicate whether someone is in the CS

```
global boolean flag = false;

thread P: { //                    thread Q: {
  // non-critical section           // non-critical section
  await !flag;                      await !flag;
  flag = true;                      flag = true;
  // CRITICAL SECTION               // CRITICAL SECTION
  flag = false;                     flag = false;
  // non-critical section           // non-critical section
}                                 }
```

- Mutex: No
- Absence deadlock: Don't care
- Free from starvation: Don't care

Attempt I – Take Turns

```
global int turn = 1;
```

```
thread P: {
  // non-critical section
  await (turn==1);
  // CRITICAL SECTION
  turn = 2;
  // non-critical section
}
```

```
thread Q: {
  // non-critical section
  await (turn==2);
  // CRITICAL SECTION
  turn = 1;
  // non-critical section
}
```

- `turn` is a “permission resource”
- Mutex:
- Absence deadlock:
- Free from starvation:

Attempt I – Take Turns

```
global int turn = 1;
```

```
thread P: {
  // non-critical section
  await (turn==1);
  // CRITICAL SECTION
  turn = 2;
  // non-critical section
}
```

```
thread Q: {
  // non-critical section
  await (turn==2);
  // CRITICAL SECTION
  turn = 1;
  // non-critical section
}
```

- `turn` is a “permission resource”
- Mutex: Yes
- Absence deadlock:
- Free from starvation:

Attempt I – Take Turns

```
global int turn = 1;
```

```
thread P: {
  // non-critical section
  await (turn==1);
  // CRITICAL SECTION
  turn = 2;
  // non-critical section
}
```

```
thread Q: {
  // non-critical section
  await (turn==2);
  // CRITICAL SECTION
  turn = 1;
  // non-critical section
}
```

- `turn` is a “permission resource”
- Mutex: Yes
- Absence deadlock: Yes
- Free from starvation:

Attempt I – Take Turns

```
global int turn = 1;
```

```
thread P: {
  // non-critical section
  await (turn==1);
  // CRITICAL SECTION
  turn = 2;
  // non-critical section
}
```

```
thread Q: {
  // non-critical section
  await (turn==2);
  // CRITICAL SECTION
  turn = 1;
  // non-critical section
}
```

- `turn` is a “permission resource”
- Mutex: Yes
- Absence deadlock: Yes
- Free from starvation: No

Attempt I – Take Turns

```
global int turn = 1;
```

```
thread P: {
  // non-critical section
  await (turn==1);
  // CRITICAL SECTION
  turn = 2;
  // non-critical section
}
```

```
thread Q: {
  // non-critical section
  await (turn==2);
  // CRITICAL SECTION
  turn = 1;
  // non-critical section
}
```

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

Attempt I – Take Turns

```
global int turn = 1;
```

```
thread P: {
  // non-critical section
  await (turn==1);
  // CRITICAL SECTION
  turn = 2;
  // non-critical section
}
```

```
thread Q: {
  // non-critical section
  await (turn==2);
  // CRITICAL SECTION
  turn = 1;
  // non-critical section
}
```

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

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

Attempt I – Take Turns

- Mutex: Yes
- Absence deadlock: 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

```
global int turn = 1;
```

```
thread P: {
  while (true) {
      // non-critical section
      await (turn==1);
      // CRITICAL SECTION
      turn = 2;
      // non-critical section
  }
}
```

```
thread Q: {
 while (true) {
    // non-critical section
    await (turn==2);
    // CRITICAL SECTION
    turn = 1;
    // non-critical section
   }
}
```

Its meant to be executed in a loop

Attempt I – Take Turns

```
global int turn = 1;
```

```
thread P: {
  while (true) {
      await (turn==1);
      turn = 2;
  }
}
```

```
thread Q: {
 while (true) {
    await (turn==2);
    turn = 1;
   }
}
```

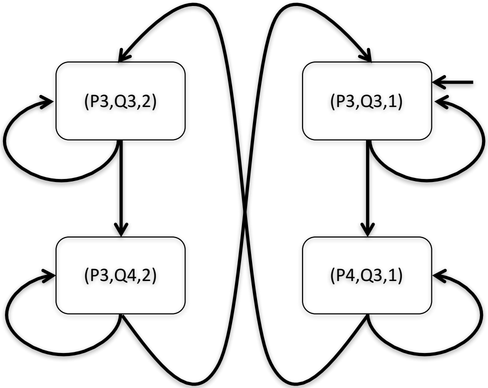
Before drawing the transition system, we get rid of irrelevant commands: we are only interested in synchronization

Attempt I – Take Turns

```
global int turn = 1;
```

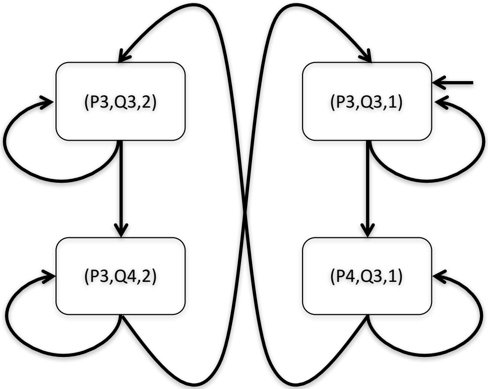
```
thread P: {
  while (true) {
      await (turn==1);
      turn = 2;
  }
}
```

```
thread Q: {
 while (true) {
     await (turn==2);
     turn = 1;
 }
}
```



Proving the Desired Properties I

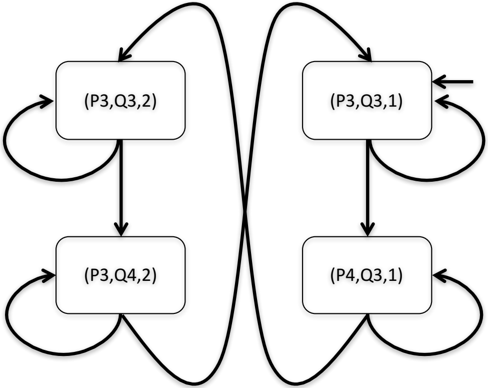
- Mutex: Holds if all accesible states do no contain a state of the form $(p4, q4, turn)$ for some value of turn



Proving the Desired Properties II

Absence of deadlock (if some processes are trying to enter their CS, i.e. executing an await, then one must eventually succeed)

- Recall that the awaits are on line 3



Proving the Desired Properties III

Freedom from starvation (if any process is about to execute its preprotocol, then eventually it will succeed in entering the CS)

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

```
global int turn = 1;
```

```
thread P: {
  while (true) {
      // non-critical section (*)
      await (turn==1);
      // CRITICAL SECTION
      turn = 2;
      // non-critical section
  }
}
```

```
thread Q: {
 while (true) {
    // non-critical section
    await (turn==2);
    // CRITICAL SECTION
    turn = 1;
    // non-critical section
   }
}
```

Attempt I – Assessment

- Starvation is due to the fact that both processes test and set a single global 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

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

```
thread P: {                          thread Q: {
 while (true) {                        while (true) {
  // non-critical section               // non-critical section
   await !wantQ;                        await !wantP;
   wantP = true;                        wantQ = true;
   // CRITICAL SECTION                  // CRITICAL SECTION
   wantP = false;                       wantQ = false;
   // non-critical section              // non-critical section
 }                                     }
}                                     }
```

- Mutex:

Attempt II

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

thread P: {                 thread Q: {
 while (true) {               while (true) {
  // non-critical section      // non-critical section
   await !wantQ;               await !wantP;
   wantP = true;               wantQ = true;
   // CRITICAL SECTION         // CRITICAL SECTION
   wantP = false;              wantQ = false;
   // non-critical section     // non-critical section
 }                           }
}                           }
```

- Mutex: No

Attempt III

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

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

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

- Mutex:
- Absence deadlock:
- Free from starvation:

Attempt III

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

thread P: {                     thread Q: {
  while (true) {                  while (true) {
   // non-critical section          // non-critical section
   wantP = true;                    wantQ = true;
   await !wantQ;                    await !wantP;
   // CRITICAL SECTION              // CRITICAL SECTION
   wantP = false;                   wantQ = false;
   // non-critical section          // non-critical section
  }                               }
}                               }
```

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

Attempt III

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

thread P: {                          thread Q: {
  while (true) {                       while (true) {
   // non-critical section              // non-critical section
   wantP = true;                        wantQ = true;
   await !wantQ;                        await !wantP;
   // CRITICAL SECTION                  // CRITICAL SECTION
   wantP = false;                       wantQ = false;
   // non-critical section              // non-critical section
  }                                    }
}                                    }
```

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

Attempt III

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

thread P: {                        thread Q: {
  while (true) {                     while (true) {
   // non-critical section            // non-critical section
   wantP = true;                      wantQ = true;
   await !wantQ;                      await !wantP;
   // CRITICAL SECTION                // CRITICAL SECTION
   wantP = false;                     wantQ = false;
   // non-critical section            // non-critical section
  }                                  }
}                                  }
```

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

Attempt III – Assessment

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

```
thread P: {                        thread Q: {
  while (true) {                     while (true) {
   // non-critical section            // non-critical section
   wantP = true;                      wantQ = true;
   await !wantQ;                      await !wantP;
   // CRITICAL SECTION                // CRITICAL SECTION
   wantP = false;                     wantQ = false;
   // non-critical section            // 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

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

```
thread P: {
  while (true) {
   // non-critical section
   wantP = true;
   while wantQ {
     wantP = false;
     wantP = true;
   }
   // CRITICAL SECTION
   wantP = false;
   // non-critical section
  }
}
```

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

- Mutex:
- Absence deadlock:
- Free from starvation:

Attempt IV – Naive Back-Out

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

```
thread P: {
  while (true) {
   // non-critical section
   wantP = true;
   while wantQ {
     wantP = false;
     wantP = true;
   }
   // CRITICAL SECTION
   wantP = false;
   // non-critical section
  }
}
```

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

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

Attempt IV – Naive Back-Out

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

```
thread P: {
  while (true) {
   // non-critical section
   wantP = true;
   while wantQ {
     wantP = false;
     wantP = true;
   }
   // CRITICAL SECTION
   wantP = false;
   // non-critical section
  }
}
```

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

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

Attempt IV – Naive Back-Out

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

thread P: {                          thread Q: {
  while (true) {                       while (true) {
   // non-critical section              // non-critical section
   wantP = true;                        wantQ = true;
   while wantQ {                        while wantP {
     wantP = false;                       wantQ = false;
     wantP = true;                        wantQ = true;
   }                                    }
   // CRITICAL SECTION                  // CRITICAL SECTION
   wantP = false;                       wantQ = false;
   // non-critical section              // non-critical section
  }                                    }
}                                    }
```

- Mutex: Yes
- Absence deadlock: Yes
- 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)

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

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

Right to insist on entering is passed between the two processes

Peterson's Algorithm (1981)

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

thread P: {                   thread Q: {
  while (true) {                while (true) {
   // non-critical section       // non-critical section
   wantP = true;                 wantQ = true;
   last = 1;                     last = 2;
   await !wantQ or last==2;      await !wantP or last==1;
   // CRITICAL SECTION           // CRITICAL SECTION
   wantP = false;                wantQ = false;
   // non-critical section       // 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 deadlock: Yes
- Free from starvation: Yes

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

```
global int np,nq =0;

thread P: {                          thread Q: {
  while (true) {                       while (true) {
   // non-critical section              // non-critical section
   [np = nq + 1];                       [nq = np + 1];
   await nq==0 or np<=nq;               await np==0 or nq<np;
   // CRITICAL SECTION                  // CRITICAL SECTION
   np = 0;                              nq = 0;
   // non-critical section              // 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

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

thread {
   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

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

thread {
  // 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