Lecture 10

- Administration

Dekker's Algorithm


```
flag=[0]= flag[1] = false;
Turn = 0;

/* Process P0: */
flag[0] = true; //indicate the need to enter critical section
while (flag[1]) //while the other process (P1) is also entering
            if (turn == 1) // if it is P1's turn
            { flag[0] = false; //reset the flag to false
            while (turn == 1) //wait
while it is still P1's turn /* do nothing */ ;
flag[0] = true; //indicate entering the critical section
}
/* critical section */;
turn = 1; //give the turn to P1
flag [0] = false; //change status

/* Process P1: */
flag[1] = true; //indicate the need to enter critical section
while (flag[0]) //while the other process (P1) is also entering
            if (turn == 0) // if it is P1's turn
            { flag[1] = false; //reset the flag to false
            while (turn == 0) //wait
while it is still P0's turn /* do nothing */ ;
flag[1] = true; //indicate entering the critical section
}
/* critical section */;
turn = 0; //give the turn to P1
flag [1] = false; //change status
```

A simple attempt

```
/* Process P0: */
flag[0] = true;
while (flag[1])
        // WAIT ---
/* enter critical section */;
flag [0] = false;
```

```
/* Process P1: */
flag[1] = true;
while (flag[0])
        // WAIT ---
/* enter critical section */;
flag [1] = false;
```

Assumptions:
(a) we can read concurrency
(b) Read/writes occur in the order of the program
(c) If two processes write the same variable one of the two values are assigned

A second attempt

```
/* Process P0: */
turn = 0;
while (turn != 0)
        // WAIT ---
/* enter critical section */;
turn = 1;
```

```
/* Process P1: */
turn = 1;
while (turn != 1)
        // WAIT ---
/* enter critical section */;
turn = 0;
```

A third attempt

```
/* Process P0: */
turn = 0;
while (turn == 0)
        // WAIT ---
/* enter critical section */;
```

```
/* Process P1: */
turn = 1;
while (turn == 1)
        // WAIT ---
/* enter critical section */;
```

Peterson's Algorithm


```
flag=[0]= flag[1] = false;
Turn = 0;

/* Process P0: */
flag[0] = true; //indicate the need to enter critical section
Turn = 1;
while (flag[1]==true && turn==1) //while the other process (P1) is also entering
            // busy wait
}
/* critical section */;
flag [0] = false; //change status



/* Process P1: */
flag[1] = true; //indicate the need to enter critical section
Turn = 0;
while (flag[0]==true && turn==0) //while the other process (P1) is also entering
            // busy wait
}
/* critical section */;
flag [1] = false; //change status
```

N Processes, Filter Lock

Hardware

- Interrupts – turn off interrupts so that the access is atomic (does not work in a multiprocessor system).
- Special Machine Instructions (test&set, compare&swap)
- Language support (Volatile, memory fences)

Hardware

Advantages of the machine-instruction approach:

- Any number of processes on either a single processor or multiple processors sharing main memory
- It is simple and therefore easy to verify.
- It can be used to support multiple critical sections

Disadvantages:

- **Busy waiting:** while a process is waiting for access to a critical section, it continues to consume processor time.
- **Starvation is possible**
- **Deadlock is possible:** if a low priority process has the critical region and a higher priority process needs, the higher priority process will obtain the processor to wait for the critical region

Bakery Algorithm


```
// declaration and initial values of global variables
Entering: array [1..NUM_THREADS] of bool = {false};
Number: array [1..NUM_THREADS] of integer = {0};

Routine: lock(integer i) {
     Entering[i] = true;
       Number[i] = 1 + max(Number[1], ..., Number[NUM_THREADS]);
     Entering[i] = false;
     for (j = 1; j <= NUM_THREADS; j++) {
                // Wait until thread j receives its number:
                while (Entering[j]) { /* nothing */ }
                //Wait until all threads with smaller numbers or with the same
                // number, but with higher priority, finish their work:
               while ((Number[j] != 0) && ((Number[j], j) < (Number[i], i)))
                              { /* nothing */ }
}}

Routine: unlock(integer i) { Number[i] = 0; }

Routine: Thread(integer i) {
   while (true) {
      lock(i);
       // The critical section goes here...
       unlock(i);
       // non-critical section...
}}
```

Linearizability

Motivation

- An object in languages such as Java and C++ is a container for data.

- Each object provides a set of methods that are the only way to manipulate that object's state.

- Each object has a class which describes how its methods behave.

- There are many ways to describe an object's behavior, for example: API.

Motivation

- In a sequential model computation, where a single thread manipulates a collection of objects, this works fine.
- But, for objects shared by multiple threads, this successful and familiar style of documentation falls apart!
- So that is why we need 'Linearizability' to describe objects.
- **Linearizability** is a correctness condition for concurrent objects.

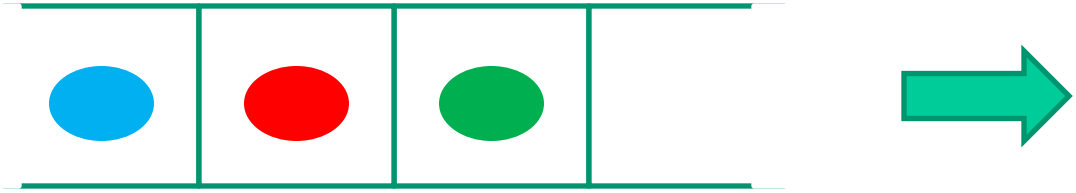
Motivation

- It permits programmers to specify and reason about concurrent objects using known techniques from the sequential domain.

- Linearizability provides the illusion that each operation applied by concurrent processes takes effect instantaneously between its invocation and its response.

Concurrent objects

Consider a FIFO queue:



q.enq( )

q.deq() /

Defining concurrent queue implementations:

- In general, to make our intuitive understanding that the algorithm is correct we need a way to specify a concurrent queue object.

- Need a way to prove that the algorithm implements the object's specification.

Correctness and progress

- There are two elements in which a concurrent specification imposes terms on the implementation: safety and liveness, or in our case, correctness and progress.

- We will mainly talk about correctness.

Sequential Specification

- We use pre-conditions and post-conditions.
- **Pre-condition** defines the state of the object before we call the method.
- **Post-condition** defines the state of the object after we call the method. Also defines returned value and thrown exception.

deq method

Pre-condition:

queue is not empty.

Post-condition:

- Returns first item in queue.
- Removes first item in queue.

Pre-condition:

queue is empty.

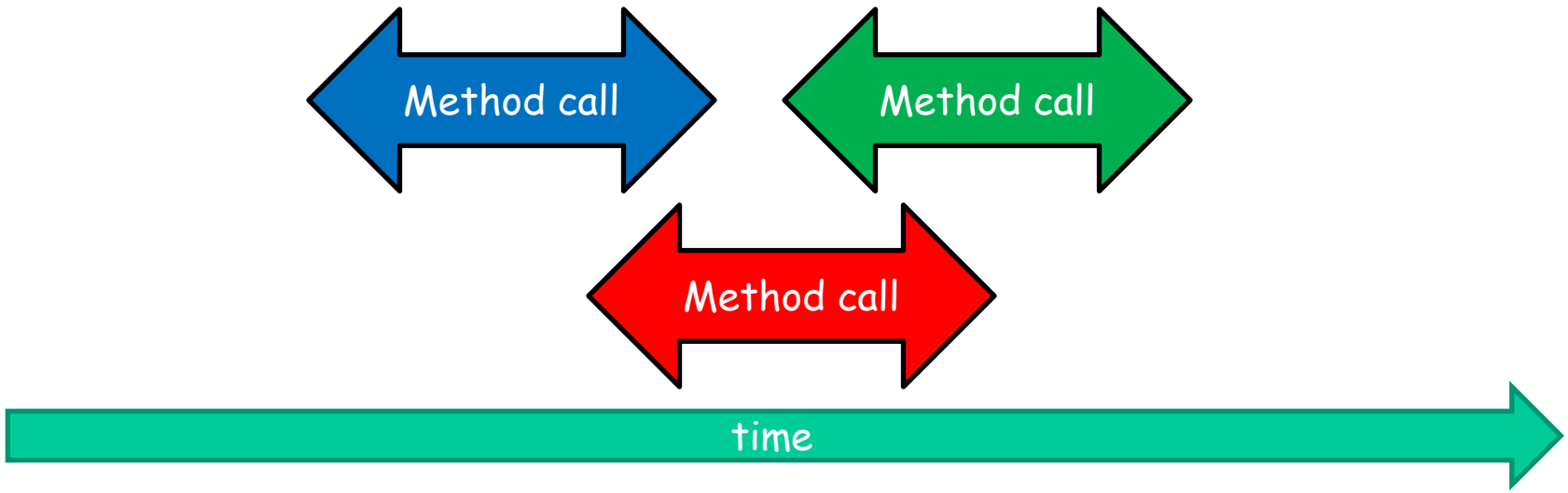
Post-condition:

- Throws EmptyException.
- Queue state is unchanged.

This makes your life easier, you don't need to think about interactions between methods and you can easily add new methods without changing descriptions of old methods.

Concurrent Specifications

- We need to understand that methods here "take time".
- In sequential computing, methods take time also, but we don't care.
  - In sequential: method call is an event.
  - In concurrent: method call is an interval.
- Methods can also take overlapping time.



Sequential vs. Concurrent

Sequential	Concurrent
Each method described independently.	Need to describe all possible interactions between methods. (what if enq and deq overlap? …)
Object's state is defined between method calls.	Because methods can overlap, the object may never be between method calls…
Adding new method does not affect older methods.	Need to think about all possible interactions with the new method.

Here we come – Linearizability!

- A way out of this dilemma: *The Linearizability Manifesto:*
  - Each method call should appear to "take effect" instantaneously at some moment between its invocation and response.

- This manifesto is not a scientific statement, it cannot be proved or disapproved. But, it has achieved wide-spread acceptance as a correctness property.

Here we come – Linearizability!

- An immediate advantage of linearizability is that there is no need to describe vast numbers of interactions between methods.
- We can still use the familiar pre- and post-conditions style.
- But still, there will be uncertainty.
- example: if x and y are enqueued on empty FIFO queue during overlapping intervals, then a later dequeue will return either x or y, but not some z or exception.

Linearizability

Linearizability has 2 important properties:

- local property: a system is linearizable iff each individual object is linearizable. It gives us composability.

- non-blocking property: one method is never forced to wait to synchronize with another.

Let's try to explain the notion "concurrent" via "sequential":

So actually we got a "sequential behaviour"!