

Distributed Systems

Programming assignment overview

Announcement

- We will accept the coursework submission as 2 separate files:
- Single source code file for programming part
- Single PDF for theory/written part
- These will be submittable separately using submit command
 - to be finalized

Interlude: Coursework

- We have now covered all material needed for the practical part of the coursework
- In the rest of lecture, I will give an overview of what is required / expected
- Overview:
 - Simulator
 - Logical clocks
 - Mutual exclusion

Overview

- You will implement a **simulator** for a distributed algorithm
- Takes in:
 - A description of the planned activities of several processes
 - Annotated with "mutual exclusion" blocks grouping atomic operations
 - "print" is also atomic: implicitly needs mutual exclusion
- Produces:
 - A sequence of events obtained by simulating the system
 - Annotated with Lamport clock timestamps
 - Satisfying mutual exclusion

Example input

```
begin process p1
  send p2 m1
  begin mutex
    print abc
    print def
  end mutex
end process
```

```
begin process p2
  print x1
  recv p1 m1
  print x2
  send p1 m2
  print x3
end process p2
```

Example output

```
begin process p1
  send p2 m1
  begin mutex
    print abc
    print def
  end mutex
end process
```

```
begin process p2
  print x1
  rcv p1 m1
  print x2
  send p1 m2
  print x3
end process p2
```

```
printed p2 x1 1
sent p1 m1 p2 1
received p2 m1 p1 2
printed p1 abc 2
printed p1 def 3
printed p2 x2 3
sent p2 m2 p1 4
printed p2 x3 5
```

OK

Example output

```
begin process p1
  send p2 m1
  begin mutex
    print abc
    print def
  end mutex
end process
```

```
begin process p2
  print x1
  recv p1 m1
  print x2
  send p1 m2
  print x3
end process p2
```

```
sent p1 m1 p2 1
printed p2 x1 1
received p2 m1 p1 2
printed p2 x2 3
sent p2 m2 p1 4
printed p1 abc 2
printed p1 def 3
printed p2 x3 5
```

OK

Incorrect output

```
printed p2 x1 1  
sent p1 m1 p2 42  
received p2 m1 p1 2  
printed p1 abc 2  
printed p1 def 3  
printed p2 x2 3  
sent p2 m2 p1 4  
printed p2 x3 5
```



```
printed p2 x1 1  
sent p1 m1 p2 42  
received p2 m1 p1 43  
printed p1 abc 45  
printed p1 def 47  
printed p2 x2 44  
sent p2 m2 p1 45  
printed p2 x3 50
```



- Timestamps should be correct
 - it is OK for there to be gaps, reflecting internal events (e.g. messages used for other protocols)

Incorrect output


```
printed p2 x1 1
sent p1 m1 p2 1
received p2 m1 p1 2
printed p1 def 3 X
printed p1 abc 2 X
sent p2 m2 p1 4
printed p2 x2 3 X
printed p2 x3 5 X
```

```
printed p2 x1 1
sent p1 m1 p2 42
received p2 m1 p1 2
printed p1 abc 2
printed p2 x2 3 X
```

- no skipped events, events in same order as in process log
- no processes "waiting" indefinitely to receive a message that has been sent
 - if a process can make progress, it should do so
 - (Processes may be deadlocked; you don't need to worry about this though.)

Incorrect output

```
printed p2 x1 1
sent p1 m1 p2 1
received p2 m1 p1 2
printed p1 abc 2
printed p2 x2 3
printed p1 def 3
sent p2 m2 p1 4
printed p2 x3 5
```



- Mutex blocks must be respected
- It should not be possible for events from mutex blocks in two different processes to be interleaved

Implementation

- You do **not** have to use Java
- You can use **any** language you like
 - provided it can read in and print out text
- We provide some Java sample code to get started
 - assuming many people know Java or a similar language
- But you do **not** have to use Java

What do we mean by simulator?

- For our purposes a simulator is **any** program that reads in the specification and produces legal outputs
 - describing complete runs of the system
- Two obvious approaches (not exclusive list!)
 - represent processes as classes/data structures and implement a scheduler that switches among them (and coordinates communication)
 - represent processes as (Java) threads, use inter-process communication primitives (or concurrent queues and synchronization) to support process communication
- **Either strategy is fine and can receive full credit; we are not assuming you are familiar with thread programming.**
- In particular, if you are not already familiar with multithreading, **the first approach is recommended.**

Scheduling

- The simulator should interleave the process behaviors
- There are several reasonable ways to do this, and part of point of exercise is for you to think about this
 - (if you haven't already seen it in an OS class)
 - See also Ch. 7 of Coulouris et al.

Lamport clocks

- The first part of the assignment asks you to add support for Lamport clocks
- Each process should maintain a counter and increment / max when it performs an event or receives a message
 - the sent/received messages should have Lamport clock values piggybacked on them
 - the receivers unpiggyback the timestamps and take the max with their current time
- Timestamps associated with events should be reported in the log

Mutual exclusion

- Mutex blocks should be handled so that no events in different mutex blocks happen concurrently
 - in particular: no interleaving of events from mutex blocks in different processes
- This should be done using messages
 - not by repurposing existing shared-memory mutual exclusion techniques like Java's `synchronized`
- Processes can send and receive additional messages to support mutual exclusion
 - But these should not be reported in the log
- For full credit this should be **fair**, i.e. requests should be granted in an order that respects happens-before ordering
 - e.g. Ricart-Agrawala algorithm
- But other simpler algorithms (ring, server) are OK and will receive partial credit