Distributed Systems
COMP 212

Lecture 22

Othon Michail

UNIVERSITY OF
LIVERPOOL

Concurrency & Mutual Exclusion

Concurrency & Mutual Exclusion

- Multiple reads may occur concurrently at a shared resource without conflict
- Problems arise when modifying the state of the resource
- Shared resource
  - File on a hard drive
  - Database entry
  - Printer, …

- It is necessary to protect the shared resource using mutual exclusion
  - Only one process can be in the “critical section” at any given time

Terminology (Uniprocessor)

- Critical section is a non-re-entrant piece of code that can only be executed by one process at a time.
- Mutual exclusion is a collection of techniques for sharing resources so that different uses do not conflict and cause unwanted interactions.
- Semaphore is a datatype (integer) used to implement mutual exclusion.

- Java synchronized methods:
  - **int synchronized** criticalSection(){…}

Mutual exclusion

Essential requirements for mutual exclusion:

- **Safety**: At most one process may execute in the critical section (CS) at a time
  - Nothing bad happens
  - Satisfies mutual exclusion
- **Liveness**: Requests to enter and exit CS eventually succeed
  - Something good eventually happens
  - Liveness implies freedom from deadlocks and starvation (indefinite postponements of entry for a process that has requested it)

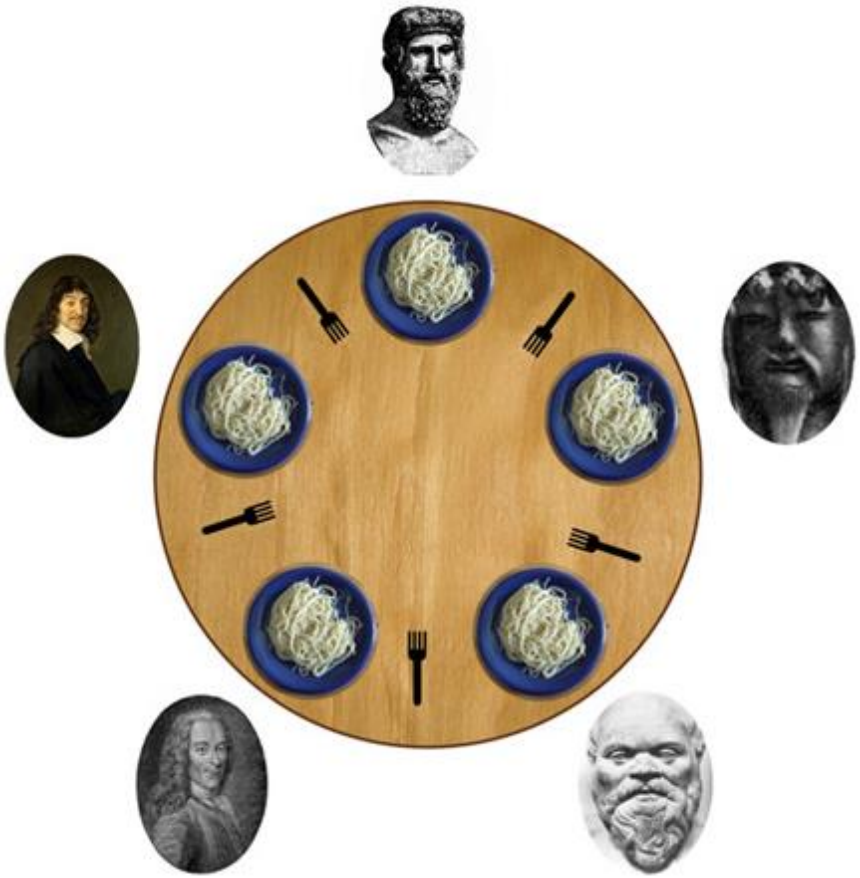
Deadlocks

Transaction *T*		Transaction *U*	
Operations	**Locks**	**Operations**	**Locks**
a.deposit(100);	write lock *A*		
		b.deposit(200)	write lock *B*
b.withdraw(100)	waits for *U*'s		
•••	lock on *B*	*a.withdraw(200);*	waits for *T*'s
		•••	lock on *A*
•••		•••	
•••		•••	

- Deadlock: a state in which each member of a group (e.g., of transactions) is waiting for some other member to release a lock
- Dining Philosophers problem
  - Illustrates the challenges of avoiding deadlock in concurrent programming

Dining Philosophers Problem

- Think and eat
- Silent
- Philosophers can take a fork as soon as available
- Can only eat with 2 forks (left & right)
- Concurrent algorithm so that no philosopher starves



- Example of deadlock: All philosophers hold their left fork

Dealing with deadlocks

- Acquire all locks in some predefined order
  - but this can result in premature locking and a reduction in concurrency
- Deadlock detection
  - after detecting a deadlock, a transaction must be selected to be aborted to break the cycle
  - it is hard to choose a "victim"
- Maintain explicit graph of locks
  - detect cycles
- Timeouts
  - Known: a lock will never be held for more than *t* sec
    - If a transaction holds a lock for more than *t* sec, this indicates a deadlock
  - it is hard to select a suitable length for a timeout

Mutual Exclusion in Message-Passing Systems

DS Mutual Exclusion: Techniques

Two major approaches:

- **Centralised**: a single coordinator controls whether a process can enter a critical region
- **Distributed**: the group confers to determine whether or not it is safe for a process to enter a critical region

Centralised Algorithm

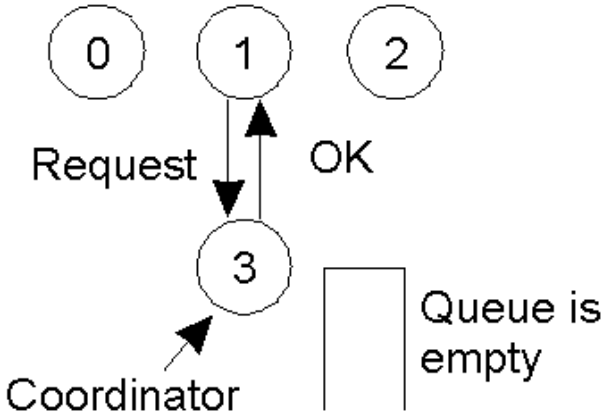
- *Assume a coordinator has been elected*
- A process sends a message to the coordinator requesting permission to enter a critical section. If no other process is in the critical section, permission is granted.
- If another process then asks permission to enter the same critical region, the coordinator does not reply (or it sends “permission denied”) and queues the request
- When a process exits the critical section, it sends a message to the coordinator
- The coordinator takes the first entry off the queue and sends that process a message granting permission to enter the critical section

Centralised Algorithm (2)

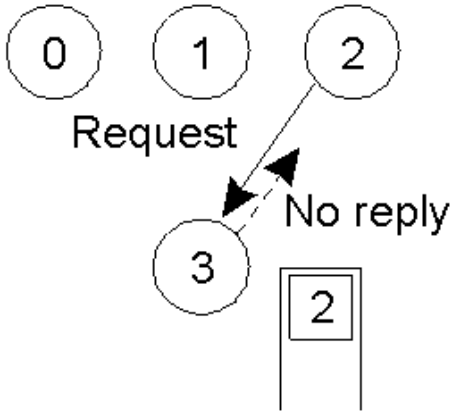
a) Process 1 asks the coordinator for permission to enter a critical region. Permission is granted.

b) Process 2 asks for permission to enter the same region. No reply.

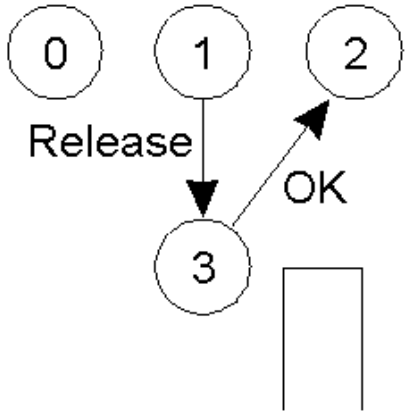
c) When Process 1 quits the critical region, it tells the coordinator, which then replies to Process 2



(a)



(b)



(c)

Comments:

Advantages:

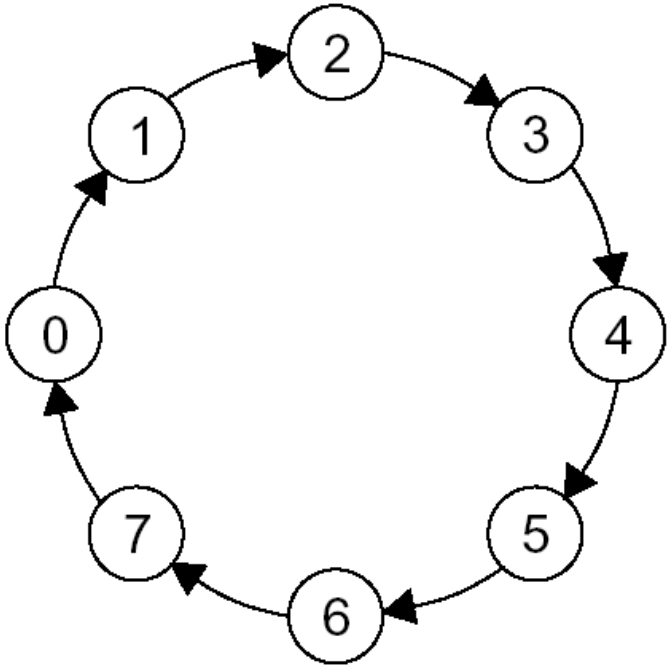
- It works
- It is fair
- There's no process starvation
- Easy to implement

Disadvantages:

- There's a single point of failure!
- The coordinator is a bottleneck in busy systems

Token-Ring Algorithm

- Processes are organised into a logical ring
- A token circulates around the ring
- A critical region can only be entered when the token is held
  - No token, no access!
- When the critical region is exited, the token is released



Comments

Advantages:

- It works (as there's only one token, so mutual exclusion is guaranteed)
- It's fair – everyone gets a shot at grabbing the token at some stage

Disadvantages:

- Lost token! How is the loss detected (it is in use or is it lost)? How is the token regenerated?
- Process failure can cause problems – a broken ring!

Distributed Algorithm (1)

- *Ricart and Agrawala algorithm (1981) assumes there is a mechanism for "totally ordering all events" in the system (e.g., Lamport's algorithm) and a reliable message system.*
- A process wanting to enter a critical section (cs) sends a message with (*cs name*, *process id*, *current time*) to all processes (including itself)
- When a process receives a cs request from another process, it reacts based on its current state with respect to the cs requested

Distributed Algorithm (2)

There are three possible cases:

1. If the receiver is not in that cs and it does not want to enter the cs, it sends an OK message to the sender
2. If the receiver is in that cs, it does not reply and queues the request
3. If the receiver wants to enter the cs but has not done yet, it compares the timestamp of the incoming message with the timestamp of its message sent to everyone

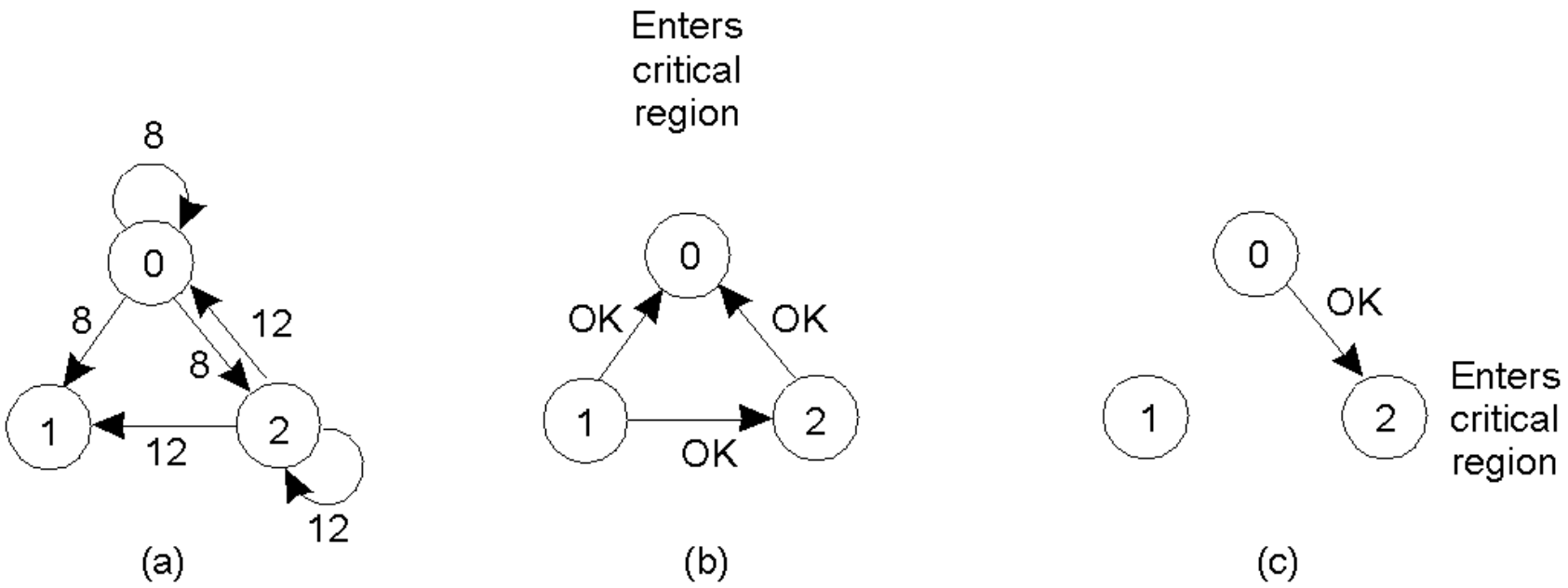
   The lowest timestamp wins:

   – If the incoming timestamp is lower, the receiver sends an OK message to the sender
   – If its own timestamp is lower, the receiver queues the request and sends nothing

Distributed Algorithm (3)

- After a process sends out a request to enter a cs, it waits for an OK from all the other processes. When all are received, it enters the cs.
- Upon exiting cs, it sends OK messages to all processes on its queue for that cs and deletes them from the queue

The Distributed Algorithm in Action



- Process 0 and 2 wish to enter the critical region "at the same time"
- Process 0 wins as it's timestamp is lower than that of process 2
- When process 0 leaves the critical region, it sends an OK to 2

Comments

Advantages:

- It works
  - The algorithm works because in the case of a conflict, the lowest timestamp wins as everyone agrees on the total ordering of the events in the distributed system
- There is no single point of failure

Disadvantages:

- We now have multiple points of failure!!!
- A "crash" is interpreted as a denial of entry to a critical region
- (A patch to the algorithm requires all messages to be acknowledged)
- Worse is that all processes must maintain a list of the current processes in the group (and this can be tricky)
- Worse still is that one overworked process in the system can become a bottleneck to the entire system – so, everyone slows down

Mutual Exclusion in Shared Memory

Shared Memory

- We have mostly focused so far on the message-passing communication paradigm
- Shared memory: the other major communication paradigm for DSs
- Now processors communicate via a common memory area
  - contains a set of shared variables

Shared Variables

- Several types
- Type: specifies the operations allowed and the values returned by them

Examples:

- read/write register
- read-modify-write register
- test&set register
- compare&swap register

The Mutual Exclusion Problem

- A group of processors occasionally need to access some resource
- The resource cannot be used simultaneously by more than a single processor
- Each processor may need to execute a critical section (code segment), such that
  - Mutual Exclusion: at most one processor is in the critical section at any time
  - No Deadlock: if one or more processors try to enter the critical section, then one of them eventually succeeds (no processor stays in it forever)

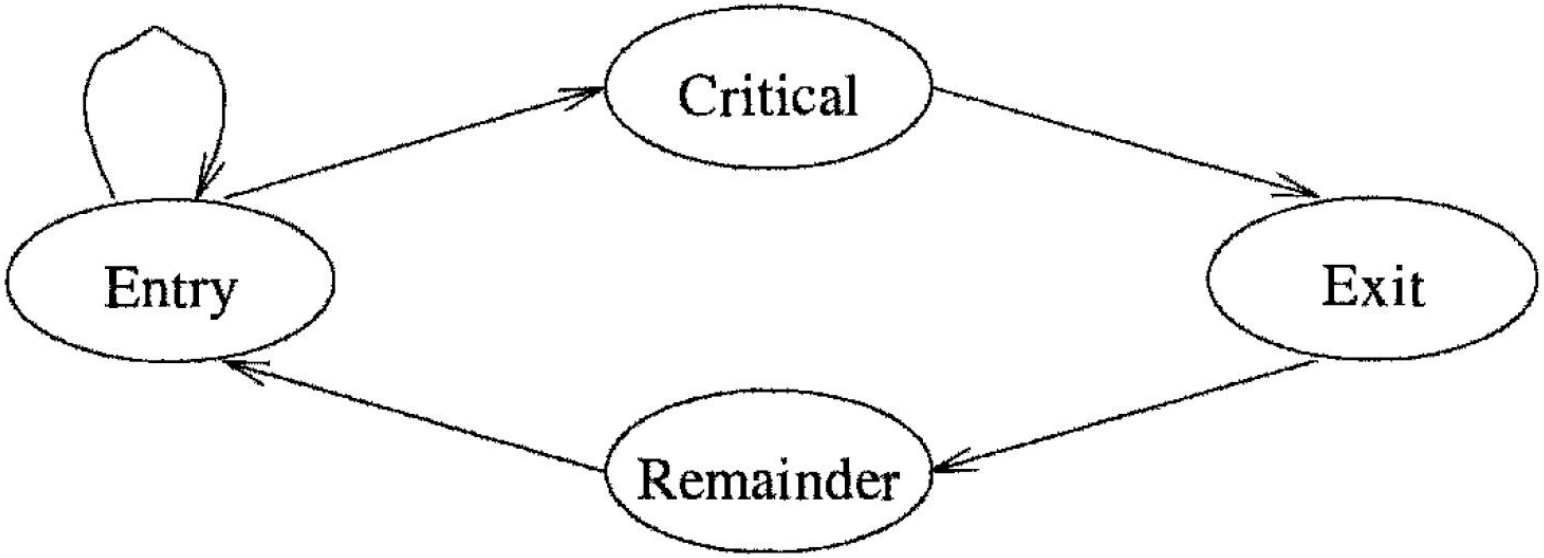
General Structure of Solutions

Programs are partitioned into the following sections:

- Entry (trying): the code executed in preparation for entering the critical section
- Critical: the code to be protected from concurrent execution
- Exit: the code executed on leaving the critical section
- Remainder: the rest of the code

General Structure of Solutions

- Processors cycle through these in the order:
  remainder, entry, critical, and exit



- A mutual exclusion algorithm consists of code for the entry and exit sections
  - Should work no matter what the other two sections implement

Requirements: More Formally

An algorithm solves mutual exclusion if the following hold:

- Mutual Exclusion: In every configuration of every execution, at most one processor is in the critical section

and

- No Deadlock: In every execution, if some processor is in the entry section in a configuration, then there is a later configuration in which some processor is in the critical section

and/or

- No Lockout: In every execution, if some processor is in the entry section in a configuration, then there is a later configuration in which *that same* processor is in the critical section

- No lockout implies no deadlock but not inversely

No Deadlock

- We use binary test&set registers
- A test&set variable *V* supports two **atomic** operations:

test&set(*V* : memory address) returns binary value:

```
temp := V
V := 1
return (temp)
```

reset(*V* : memory address):

```
V := 0
```

Simple algorithm for mutual exclusion with no deadlock that uses only 1 such register

Solution: Pseudocode

Algorithm Test&SetME
// Mutual exclusion using a test&set register
Code for every processor:

Initially $V = 0$

⟨Entry⟩:
  wait until test&set(V) = 0

⟨Critical Section⟩

⟨Exit⟩:
  reset(V)

⟨Remainder⟩

No Lockout

- We use read-modify-write registers
- A read-modify-write variable V supports the following **atomic** operation:

rmw(*V* : memory address, *f* : function) returns value:

temp := *V*

V := *f*(*V*)

return (*temp*)

- *f* is a parameter that specifies how the new value is related to the old one
- test&set is a special case of rmw, where *f*(*V*) = 1 for any value of *V*

Mutual exclusion algorithm that guarantees no lockout using only 1 such register

Solution: Informal Description

- Every processor has two local variables:
  - *position*
  - *queue*
- One read-modify-write register is used, consisting of two fields:
  - *first*
  - *last*
- Represent "tickets" of the first and last processors in the queue, respectively

Solution: Informal Description

- When a new processor arrives at the entry section:

1. it enqueues
   - reads *V* to local variable *position*
   - the current value of *V* serves as the processor's ticket
   - in the same atomic operation it increases *V.last* by 1

2. after enqueing waits to "get served"
   - waits until it reaches the head of the queue
   - i.e., waits until *V.first* becomes equal to its ticket (i.e., to *position.last*)
   - when this happens, the processor enters the critical section

- When a processor leaves the critical section
  - dequeues to allow the next processor on the queue to enter the critical section
  - i.e., increases *V.first* by 1

Intuition

- *V.last*: machine issuing tickets 0, 1, 2, 3, 4, … initially 0
- *V.first*: monitor showing who is being served 0, 1, 2, 3, 4, … initially 0
- When a processor wants to get served, issues a ticket (ticket number increases by 1) and waits until the monitor shows its ticket number
- When this happens it leaves and then the monitor increases by 1 to serve the next processor

- Note: When ticket $n - 1$ has been printed we can assume that the ticket numbering can start over from 0
  - because the processors that can be waiting at any given time are at most n
  - the same for the monitor

Solution: Pseudocode

Algorithm Read-Modify-WriteME
// Mutual exclusion using a read-modify-write register
Code for every processor:

Initially $V = \langle 0, 0 \rangle$

⟨Entry⟩:
  position := rmw(*V*, ⟨*V.first*, *V.last* + 1⟩) // enqueuing at the tail
  repeat
    queue := rmw(*V*, *V*) // read head of queue
  until (*queue.first* = *position.last*) // until becomes first

⟨Critical Section⟩

⟨Exit⟩:
  rmw(*V*, ⟨*V.first* + 1, *V.last*⟩) // dequeueing

⟨Remainder⟩

Read/Write Registers

- Less powerful shared memory primitive
  - reads and writes are independent
  - in read-modify-write the power was in that a processor could read and write in a single atomic step without giving a chance to other processors to intervene
- Even to just achieve no deadlock we cannot avoid using *n* separate read/write registers
  - With more powerful registers, just 1 register was sufficient
- Lamport's Bakery Algorithm
  - Considers the processors entering the critical section as customers in a bakery!
  - Again numbered tickets

Mutual exclusion algorithm that guarantees no lockout using $O(n)$ such registers

Summary

- Processes in DSs are autonomous
- Quite typically many of them might want to access the same shared resource concurrently
  - e.g., a printer
- We cannot allow them all to do this at the same time
  - much like as it wouldn't work to have all students write on a single whiteboard at the same time
- Problem: Mutual Exclusion
- Additional guarantees: no deadlock, no starvation (or no lockout)
- Centralised vs Distributed solutions
- Message-passing systems vs shared memory systems