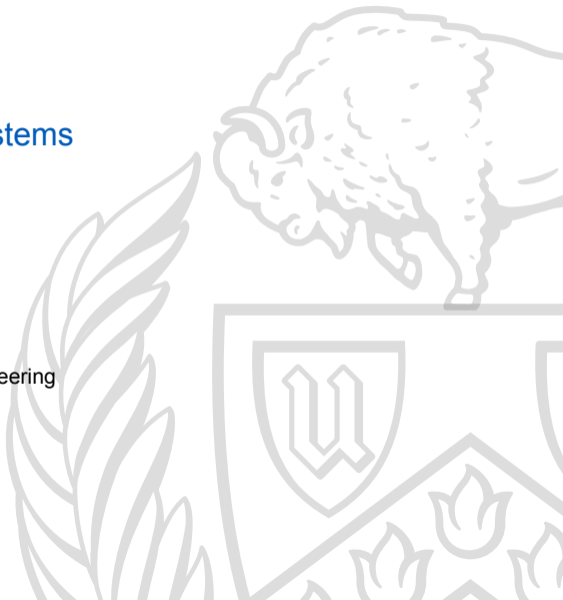


Mutual Exclusion

CSE 486/586: Distributed Systems

Ethan Blanton

Department of Computer Science and Engineering
University at Buffalo



Mutual Exclusion

Mutual Exclusion is another **consensus** problem.

What process has **exclusive access** to a resource at time t ?

Locally, this is typically handled by **hardware** or the **OS**.

In a distributed system, this becomes more difficult.

Races

Races, or race conditions, are situations where:

- Two or more events are **dependent upon each other**
- Some of the events **may happen in more than one order**, or even simultaneously
- There exists some ordering of the events that is **incorrect**

For example:

- Some state will be updated multiple times
- Output will be produced based on the state

If some order of updates results in invalid output, **this is a race**.

Critical Sections

A **critical section** is a **region of code** that must be accessed by **at most one concurrent process at a time**.

Typically, the region of code **mutates state** in some fashion.

Multiple concurrent mutations may result in **inconsistent state**.

For example, **storing to a Go map** is a critical section.

Multiple concurrent stores **can cause a panic**.

Mutual Exclusion

Mutual exclusion is a tool for ensuring that **only one concurrent process** accesses some resource.

It is one of the **most basic** synchronization methods.

Mutual exclusion maps almost directly to critical sections:

- The code of the critical section is the resource

The Mutex

A software tool for providing mutual exclusion is the **mutex**.

It provides two operations:

- Lock
- Unlock

<i>Operation</i>	<i>Mutex State</i>	<i>Action</i>
Lock	Unlocked	Lock mutex immediately ¹
Lock	Locked	Block until unlocked, then lock
Unlock	Locked	Unlock mutex immediately
Unlock	Unlocked	Implementation dependent

¹If a flow locks a mutex it has already locked, behavior is implementation-dependent.

Properties of Mutexes

We want our mutexes to exhibit three properties:

Safety:

No two processes enter the critical section at one time

Liveness:

When there are processes that want to enter the critical section, one of them **eventually** does

Fairness:

No process waits **indefinitely** to enter the critical section

Deadlock

Deadlock is a condition in concurrent programming where two or more processes are **waiting for each other** and thus can never make progress.

Consider:

process A:

```
lock mutex m0
lock mutex m1
do something
```

process B:

```
lock mutex m1
lock mutex m0
do something
```

If process A is interrupted by process B **after locking m0 and before locking m1**, deadlock occurs.

Neither process can **proceed**, and neither can **release the other**.

Avoiding Deadlock

Deadlock is **caused by synchronization**.

There are **various techniques to avoid deadlock**.

For deadlock caused by **mutual exclusion on multiple locks**, there is a simple solution:

- All mutexes in a system are **ordered** (perhaps artificially)
- All flows lock mutexes **in order**
- All flows unlock mutexes **in reverse order**

Distributed Mutual Exclusion

In a distributed system, mutual exclusion is by **message passing**.

There is no **shared memory** or **operating system primitive** on which to implement the mutex!

In an **asynchronous** system, this has the complications we've discussed:

- Messages are delayed
- Messages arrive out of order
- Processes fail to respond
- *etc.*

Properties of Distributed Mutexes

Distributed mutexes have several more properties that interest us:

Synchronization Delay:

The **duration of time** between some process releasing the mutex and the next process acquiring it

Throughput:

The **number of lock/unlock pairs** per unit time; this is typically measured with “empty” critical sections

Message Complexity:

The **number of messages** required to obtain (or obtain and release) a lock

Centralized Mutual Exclusion

Centralization is an obvious solution to this problem.

The simplest case:

- A server maintains the current lock holder
- Processes send lock/unlock requests

This reduces distributed mutual exclusion to a [local mutex](#).

Let's look at its properties.

Properties of Centralized Mutexes

Safety: **as safe as** the central mutex

Liveness: Vulnerable to **single point of failure**

Fairness: **as fair as** the central mutex (or server algorithm)

Synchronization Delay: Twice the message **one-way delay** d

Throughput: $1/2d$

Message Complexity: Minimum 3: lock request, lock grant, unlock request

Single Point of Failure

If the server fails, **deadlock occurs**:

- If the mutex is locked, **it cannot be unlocked**.
- If the mutex is unlocked, **it cannot be locked**.

Elections can be used to solve this.

Some technique must be used to **recover the mutex state**.

Tokens

Access to resources can be tied to a **token**.

The holder of a token may use a resources.

With **ownership semantics**, this can provide mutual exclusion:

- There is **exactly one token**
- The token must be **passed** between processes
- The process in possession of the token **may enter the critical section**

Token Rings

Le Lann [3] proposed a **ring structure** for token passing.

(This is the same paper that introduced ring elections!)

Some process **creates the (unique) token**, and:

1. Executes the critical section if it desires
2. Hands the token to the next process on the ring

The token is only passed when a process is **not** in the critical section.

Only the process with the token may enter the critical section.

Maintaining the Token

Correct operation requires **exactly one** token:

- More than one token **violates mutual exclusion**
- No token causes deadlock

Therefore:

- Failure of **the process with the token** is a problem.
- Generation of **the initial token** is a problem.

Elections

The ring election protocol was **originally proposed** for generating the token!

Exactly **one process** generates exactly **one token**.

If the token is lost, **an election occurs**.

This is where FLP comes in:

How do you know that the token was lost?

The answer is **timeouts**.

In an asynchronous system, **no timeout is guaranteed** to be long enough.

Observations

If you expect **high contention**, this is very fair (round robin).

If you expect **low contention**, the throughput is poor.

Token rings have been popular for other network protocols.
(Token ring was a competitor to (broadcast-based) Ethernet.)

CSP-style concurrency control can be thought of as token passing **but without (necessarily) a ring structure**.

Properties

Safety: As long as there is **one token**

Liveness: Vulnerable to **token loss**

Fairness: Very good (at least $1/n$)

Synchronization Delay: $O(n)$ times message delay

Throughput: 1 over message delay **or worse**

Message Complexity: Minimum 1, Maximum n

Lamport Distributed Mutexes

Lamport Clocks [2] propose a distributed mutex.

Individual processes **reserve a place** in a queue.

Logical clocks **total order** the queue and **grant the lock** in cooperation with other processes.

It is loosely based on another Lamport story-paper [1].

Ricart and Agrawala

Ricart and Agrawala proposed an **improved algorithm** [4].

Its operation is simple:

- A process wishing to **enter the critical section** sends a REQUEST message to every other process.
- A process which does not object sends a REPLY message.
- A process which objects **delays its reply**.
- Once **all processes reply**, the process can enter.

Whether or not a process should object depends on:

- Whether it is currently in the critical section
- Whether it thinks it should be allowed to enter first

Logical Clocks

The ordering of entries is maintained by a **logical clock**.

Every REQUEST has a clock timestamp which is its **priority**.
The clock is **incremented before stamping** the REQUEST.

Numerically lower priorities are serviced before numerically higher priorities.

Incoming requests advance the local clock.

Entering the Critical Section

To **enter the critical section**, the process does the following:

1. Increment the logical clock `Time`
2. Set `Requesting = true`
3. Set `Sequence = Time`
4. Timestamp a REQUEST message
5. Send the request to **all processes**
6. Wait for a REPLY message from **all processes**

Processing a Request

When a REQUEST with timestamp k is received, the process will:

1. Set the local timestamp to $\text{MAX}(k, \text{Time})$
2. If `Requesting == false` or $k < \text{Time}$, send a REPLY immediately
3. Otherwise, enqueue the request on RequestQueue

To leave the critical section, the process will:

1. Set `Requesting = false`
2. Send a REPLY for all messages in RequestQueue

Observations

The set of participating nodes **must be known**.

Messages must be **reliable**.

Transmission delay must be **bounded**.

Note that it does **not require a synchronous system**, however:

- Messages can arrive out of order
- The bound on transmission delay can be **arbitrarily long**

Properties

Safety:

- Every process must REPLY to allow a critical section
- No process will REPLY if its own critical section is active or pending at a higher priority

Liveness: Deadlock if **any process fails**, but **not otherwise**

Fairness: Perfectly fair

Synchronization Delay: **One-way** message delay

Throughput: ... complicated?

Message Complexity: $2n - 2$

Failures and FLP

This process **deadlocks on node failure**.

The paper suggests a **timeout-based** failure detector.

This is the **out for FLP!**

If messages can be **arbitrarily delayed**, then **no timeout is sufficient**.

An early timeout leads to **violation of safety**.

Summary

We will see mutual exclusion again.

- Mutual exclusion is valuable for distributed systems
- Races occur when ordering is important and not maintained
- Mutexes model mutual exclusion
- Deadlocks can arise when mutexes are used
- Logical clocks can be used to implement distributed mutexes

References I

Required Readings

- [2] Leslie Lamport. “Time, Clocks, and the Ordering of Events in a Distributed System”. In: *Communications of the ACM* 21.7 (July 1978). Ed. by R. Stockton Gaines, pp. 558–565. URL: <http://lamport.azurewebsites.net/pubs/time-clocks.pdf>.

Optional Readings

- [1] Leslie Lamport. “A New Solution of Dijkstra’s Concurrent Programming Problem”. In: *Communications of the ACM* 17.8 (Aug. 1974), pp. 453–455. DOI: 10.1145/361082.361093. URL: <http://lamport.azurewebsites.net/pubs/bakery.pdf>.

References II

- [3] Gérard Le Lann. “Distributed Systems—Towards a Formal Approach”. In: *Information Processing 77*. Ed. by Bruce Gilchrest. North-Holland Publishing Company, 1977, pp. 155–160. URL: <https://www.rocq.inria.fr/novaltis/publications/IFIP%20Congress%201977.pdf>.
- [4] Glenn Ricart and Ashok Agrawala. “An optimal algorithm for mutual exclusion in computer networks”. In: *Communications of the ACM 24.1* (Jan. 1981). Ed. by R. Stockton Gaines, pp. 9–17. DOI: 10.1145/358527.358537. URL: https://search.lib.buffalo.edu/permalink/01SUNY_BUF/12pkqkt/cdi_crossref_primary_10_1145_358527_358537.

Copyright 2021 Ethan Blanton, All Rights Reserved.

Reproduction of this material without written consent of the author is prohibited.

To retrieve a copy of this material, or related materials, see <https://www.cse.buffalo.edu/~eblanton/>.