Mutual Exclusion I

Maruth Goyal

UT Austin

Spring 2021
1 Introduction, Problem Setup

2 Computational Model

3 Formalizing Mutual Exclusion

4 Dijkstra’s ME Algorithm

5 Dijkstra’s ME Analysis

6 Summary
Until now, we have been looking at the synchronous model of distributed computation.
Until now, we have been looking at the synchronous model of distributed computation.

In reality, however, distributed computing is asynchronous: i.e., computation generally does not proceed in synchronized sequences of rounds.
Until now, we have been looking at the synchronous model of distributed computation.

In reality, however, distributed computing is asynchronous: i.e., computation generally does not proceed in synchronized sequences of rounds.

This introduces a whole new swath of challenges.
Until now, we have been looking at the synchronous model of distributed computation.

In reality, however, distributed computing is asynchronous: i.e., computation generally does not proceed in synchronized sequences of rounds.

This introduces a whole new swath of challenges.

Today we will look at a primitive/fundamental challenge: mutual exclusion.
Until now, we have been looking at the synchronous model of distributed computation.

In reality, however, distributed computing is asynchronous: i.e., computation generally does not proceed in synchronized sequences of rounds.

This introduces a whole new swath of challenges.

Today we will look at a primitive/fundamental challenge: mutual exclusion.

Roughly speaking, given some resource $R$ and processors $P_1, \ldots, P_n$ we need a protocol to ensure at most one processor is interacting with the resource at a given time.
For instance, if there is a printer on a network which prints the bytes it receives, unless mutual exclusion is enforced the streams of 2 processors may get garbled in arbitrary ways.
For instance, if there is a printer on a network which prints the bytes it receives, unless mutual exclusion is enforced the streams of 2 processors may get garbled in arbitrary ways.

We will be working in the shared memory model of asynchronous computing.
For instance, if there is a printer on a network which prints the bytes it receives, unless mutual exclusion is enforced the streams of 2 processors may get garbled in arbitrary ways.

We will be working in the shared memory model of asynchronous computing.

i.e., all the processors will have access to some shared piece of memory. Additional constraints may be placed on who can write where, and read what.
For instance, if there is a printer on a network which prints the bytes it receives, unless mutual exclusion is enforced the streams of 2 processors may get garbled in arbitrary ways.

We will be working in the **shared memory** model of asynchronous computing.

i.e., all the processors will have access to some shared piece of memory. Additional constraints may be placed on who can write where, and read what.

In practice, modern processors support many *atomic* operations such as *read-modify-write*, however, for now we will not assume their existence.
Table of Contents

1 Introduction, Problem Setup

2 Computational Model

3 Formalizing Mutual Exclusion

4 Dijkstra’s ME Algorithm

5 Dijkstra’s ME Analysis

6 Summary
Computing Model

- Each processor is modeled as a state machine
- The machine has possible states $\Sigma$, and a subset $\Sigma_{\text{start}}$ of start states.
- The machine also has a set of “actions”
  - Each action is classified as being either “input”, “output”, or “internal”
  - Internal actions are further divided into those that interact with the shared memory and those that don’t.
Computing Model

- Each processor is modeled as a state machine
- The machine has possible states $\Sigma$, and a subset $\Sigma_{\text{start}}$ of start states.
- The machine also has a set of “actions”
  - Each action is classified as being either “input”, “output”, or “internal”
  - Internal actions are further divided into those that interact with the shared memory and those that don’t.
- There is a transition relation $\tau$ which has triples of the form $(s, \pi, s')$ where $s, s'$ are states of the entire automaton representing the combination of all the processors and memory.
Computing Model

- Each processor is modeled as a state machine.
- The machine has possible states $\Sigma$, and a subset $\Sigma_{\text{start}}$ of start states.
- The machine also has a set of “actions”
  - Each action is classified as being either “input”, “output”, or “internal”.
  - Internal actions are further divided into those that interact with the shared memory and those that don’t.
- There is a transition relation $\tau$ which has triples of the form $(s, \pi, s')$ where $s, s'$ are states of the entire automaton representing the combination of all the processors and memory.
- An action is “enabled” with respect to state $s$ if there exists some $s'$ such that $(s, \pi, s') \in \tau$. 
Computing Model

- Each processor is modeled as a state machine
- The machine has possible states $\Sigma$, and a subset $\Sigma_{\text{start}}$ of start states.
- The machine also has a set of “actions”
  - Each action is classified as being either “input”, “output”, or “internal”
  - Internal actions are further divided into those that interact with the shared memory and those that don’t.
- There is a transition relation $\tau$ which has triples of the form $(s, \pi, s')$ where $s, s'$ are states of the entire automaton representing the combination of all the processors and memory.
- An action is “enabled” with respect to state $s$ if there exists some $s'$ such that $(s, \pi, s') \in \tau$.
- Transition actions must satisfy “locality”:
  - An action not involving shared memory most only involve the process’s local state.
  - An action accessing shared memory location $x$ may only involve the value there and the process’s local state.
Computational Model

Definition (Read-Write Variables)

In each step, a process can either read or write a single shared variable, but not both.

1. **(read):** Process \( i \) reads register \( x \) and uses the value read to modify the state of process \( i \).

2. **(write):** Process \( i \) writes a value determined from process \( i \)'s state to register \( x \).
Computational Model

Definition (Read-Write Variables)

In each step, a process can either read or write a single shared variable, but not both.

1. **(read):** Process $i$ reads register $x$ and uses the value read to modify the state of process $i$.

2. **(write):** Process $i$ writes a value determined from process $i$’s state to register $x$.

Definition (Fairness Condition)

For each process $i$, one of the following holds:

1. The entire execution is finite, and in the final state no locally controlled action of process $i$ is enabled.

2. The execution is infinite, and there are either infinitely many occurrences of locally controlled actions of $i$, or else infinitely many places where no such action is enabled.
Table of Contents

1. Introduction, Problem Setup
2. Computational Model
3. Formalizing Mutual Exclusion
4. Dijkstra’s ME Algorithm
5. Dijkstra’s ME Analysis
6. Summary
Assume there are $n$ users $U_1, \ldots, U_n$, each trying to access a shared resource $R$.

In order to get access to $R$, each $U_i$ must interact with process $P_i$ which is part of the shared memory ME protocol.
Assume there are $n$ users $U_1, \ldots, U_n$, each trying to access a shared resource $R$.

In order to get access to $R$, each $U_i$ must interact with process $P_i$ which is part of the shared memory ME protocol.

Define the following sequence of events:

1. $U_i$ send a TRY request to $P_i$ (try to acquire lock)
Assume there are \( n \) users \( U_1, \ldots, U_n \), each trying to access a shared resource \( R \).

In order to get access to \( R \), each \( U_i \) must interact with process \( P_i \) which is part of the shared memory ME protocol.

Define the following sequence of events:

1. \( U_i \) send a \texttt{TRY} request to \( P_i \) (try to acquire lock)
2. \( U_i \) waits until it receives a \texttt{CRITICAL} message from \( P_i \) (lock acquired)
Assume there are $n$ users $U_1, \ldots, U_n$, each trying to access a shared resource $R$.

In order to get access to $R$, each $U_i$ must interact with process $P_i$ which is part of the shared memory ME protocol.

Define the following sequence of events:

1. $U_i$ send a **TRY** request to $P_i$ (try to acquire lock)
2. $U_i$ waits until it receives a **CRITICAL** message from $P_i$ (lock acquired)
3. $U_i$ now has exclusive access to resource, does its thing.
Assume there are $n$ users $U_1, \ldots, U_n$, each trying to access a shared resource $R$.

In order to get access to $R$, each $U_i$ must interact with process $P_i$ which is part of the shared memory ME protocol.

Define the following sequence of events:

1. $U_i$ send a **Try** request to $P_i$ (try to acquire lock)
2. $U_i$ waits until it receives a **Critical** message from $P_i$ (lock acquired)
3. $U_i$ now has exclusive access to resource, does its thing.
4. $U_i$ sends **Exit** request to $P_i$ (release lock), and waits for a **Rem** message from $P_i$ (lock released).
Formalizing Mutual Exclusion

- Assume there are $n$ users $U_1, \ldots, U_n$, each trying to access a shared resource $R$.
- In order to get access to $R$, each $U_i$ must interact with process $P_i$ which is part of the shared memory ME protocol.
- Define the following sequence of events:
  1. $U_i$ send a TRY request to $P_i$ (try to acquire lock)
  2. $U_i$ waits until it receives a CRITICAL message from $P_i$ (lock acquired)
  3. $U_i$ now has exclusive access to resource, does its thing.
  4. $U_i$ sends EXIT request to $P_i$ (release lock), and waits for a REM message from $P_i$ (lock released).
- The time between (1) and (2) is called the Trying region ($T$), (3) and (4) the Critical region ($C$), and after (4) the remainder region ($R$).
Formalizing Mutual Exclusion

\[ U_i \]

\[ \text{ports} \quad \text{processes} \quad \text{shared variables} \]

\[ \ldots \]

\[ \ldots \]

\[ \ldots \]

\[ try_i \]

\[ crit_i \]

\[ exit_i \]

\[ rem_i \]

\[ \ldots \]

\[ \ldots \]

\[ \ldots \]

\[ \ldots \]

\[ \ldots \]

\[ \ldots \]
A shared memory system $A$ solves the mutual exclusion problem for a collection of users if:

1. **Well-formedness**: In any execution, and for any $i$, the subsequence describing the interaction between $U_i$ and $A$ is well-formed for $i$ (i.e., follows cyclic pattern of API).
Formalizing Mutual Exclusion

Definition (Solving Mutual Exclusion)

A shared memory system $A$ solves the mutual exclusion problem for a collection of users if:

1. **(Well-formedness):** In any execution, and for any $i$, the subsequence describing the interaction between $U_i$ and $A$ is well-formed for $i$ (i.e., follows cyclic pattern of API).

2. **(Mutual exclusion):** There is no reachable system state (that is, a combination of an automaton state for $A$ and states for all the $U_i$) in which more than one user is in the critical region $C$. 
Formalizing Mutual Exclusion

Definition (Solving Mutual Exclusion)

A shared memory system $A$ solves the mutual exclusion problem for a collection of users if:

1. **(Well-formedness)**: In any execution, and for any $i$, the subsequence describing the interaction between $U_i$ and $A$ is well-formed for $i$ (i.e., follows cyclic pattern of API).

2. **(Mutual exclusion)**: There is no reachable system state (that is, a combination of an automaton state for $A$ and states for all the $U_i$) in which more than one user is in the critical region $C$.

3. **(Progress)**: At any point in *fair execution*:
   - If at least one user is in $T$ and no user is in $C$, then at some later point some user enters $C$.
   - If at least one user is in $E$, then at some later point some user enters $R$.
The well-formedness and mutual exclusion properties are safety properties, whilst the progress property is a liveness property. (eventually something good happens).
The well-formedness and mutual exclusion properties are safety properties, whilst the progress property is a liveness property. (eventually something good happens).

Note progress assumes a fair execution: i.e., every user/process keeps executing.
The well-formedness and mutual exclusion properties are safety properties, whilst the progress property is a liveness property. (eventually something good happens).

Note progress assumes a fair execution: i.e., every user/process keeps executing.

This definition does have a “problem”:

1. **Starvation**: progress only requires that some user enters $C$. i.e., it’s fine if there’s a processor forever stuck in the trying phase so long as there exists a process entering $C$. 

---

Maruth Goyal (UT Austin)
1. Introduction, Problem Setup

2. Computational Model

3. Formalizing Mutual Exclusion

4. Dijkstra’s ME Algorithm

5. Dijkstra’s ME Analysis

6. Summary
Dijkstra’s ME Algorithm

- Essentially a distributed Spin-Lock without atomics (*nasty*).
- Not the most elegant, or efficient algorithm, neither does it satisfy the strongest conditions.
- First example of such algorithm though, good to study.
Dijkstra’s ME Algorithm

Process $i$:

** Remainder region **

$\text{try}_i$

$\text{L: } flag(i) := 1$

while $\text{turn} \neq i$ do

if $\text{flag(\text{turn})} = 0$ then $\text{turn} := i$

$\text{flag}(i) := 2$

for $j \neq i$ do

if $\text{flag}(j) = 2$ then goto L

$\text{crit}_i$

** Critical region **

$\text{exit}_i$

$\text{flag}(i) := 0$

$\text{rem}_i$
In the first loop all processors try to see if the processor which was previously in the critical section has left.

If so, they “plant their flag” by setting the turn to their id.
Dijkstra's ME Algorithm

- In the first loop all processors try to see if the processor which was previously in the critical section has left.
- If so, they “plant their flag” by setting the `turn` to their `id`.
- Next, they set their flag to 2 as a way of indicating that they are about to try and enter the critical section next.
- Now each process checks that no one else is simultaneously planning the same thing. If so, they go back to the start.

1. **Note:** The next time, whichever processor set `turn` last will skip the first loop, and directly go the second loop. It will also succeed in this loop and enter the critical section.

2. **Note:** Every processor which set `turn` before that processor will be stuck in the first loop till it exits, since that processor’s `flag` was already set to 2.
Dijkstra’s ME Algorithm

- In the first loop all processors try to see if the processor which was previously in the critical section has left.
- If so, they “plant their flag” by setting the turn to their id.
- Next, they set their flag to 2 as a way of indicating that they are about to try and enter the critical section next.
- Now each process checks that no one else is simultaneously planning the same thing. If so, they go back to the start.

1. **Note:** The next time, whichever processor set turn last will skip the first loop, and directly go the second loop. It will also succeed in this loop and enter the critical section.
2. **Note** every processor which set turn before that processor will be stuck in the first loop till it exits, since that processor’s flag was already set to 2.

- Essentially *last-come-first-serve*-ish. Everyone tries first, but then if multiple went at the same time, whoever went last goes first.
Proof of correctness essentially by case-work.

Theorem

**Dijkstra’s ME algorithm guarantees well-formedness**
Dijkstra’s ME Analysis

- Proof of correctness essentially by case-work.

**Theorem**

*Dijkstra’s ME algorithm guarantees well-formedness*

**Theorem**

*Dijkstra’s Algorithm guarantees Mutual Exclusion*
Dijkstra’s ME Analysis

- Proof of correctness essentially by case-work.

**Theorem**

*Dijkstra’s ME algorithm guarantees well-formedness*

**Theorem**

*Dijkstra’s Algorithm guarantees Mutual Exclusion*

**Proof.**

1. By contradiction. Suppose $U_i, U_j$ simultaneously exist in the critical region.
2. Both $i, j$ must set their $\text{flag}$ to 2 before entering the critical region.
3. $\text{flag}$ will be 2 for both until they exit the region.
Dijkstra’s ME Analysis

- Proof of correctness essentially by case-work.

**Theorem**

_Dijkstra’s ME algorithm guarantees well-formedness_

**Theorem**

_Dijkstra’s Algorithm guarantees Mutual Exclusion_

**Proof.**

1. By contradiction. Suppose $U_i, U_j$ simultaneously exist in the critical region.
2. Both $i, j$ must set their flag to 2 before entering the critical region.
3. Flag will be 2 for both until they exit the region.
4. Suppose WLOG $U_i$ enters first. But then, flag$(i)$ must be 2, and thus $j$’s test in the while loop must fail.
Dijkstra's ME Analysis

**Theorem**

*Dijkstra's algorithm guarantees progress*
Dijkstra’s algorithm guarantees progress

Proof.

1. Exit case trivial.
2. Suppose there’s at least one user in $T$ and none in $C$. Every user in $E$ leaves by previous case.
Dijkstra’s ME Analysis

Theorem

*Dijkstra’s algorithm guarantees progress*

Proof.

1. **Exit** case trivial.
2. Suppose there’s at least one user in $T$ and none in $C$. Every user in $E$ leaves by previous case.
3. Call processes stuck in $T$ *contenders*. 
Dijkstra’s ME Analysis

Theorem

*Dijkstra’s algorithm guarantees progress*

Proof.

1. **Exit** case trivial.
2. Suppose there’s at least one user in $T$ and none in $C$. Every user in $E$ leaves by previous case.
3. Call processes stuck in *T contenders*.
4. For each contender $i$, $\text{flag}(i) \geq 1$. 
Lemma

\textit{turn} eventually acquires a contender's index.
Lemma

*turn eventually acquires a contender’s index.*

**Proof.**

1. By contradiction. Suppose it acquires $j$, which is not a contender, and thus $\text{flag}(j) = 0$.

2. Then, for any contender $i$, in the first while loop it will see that $\text{flag}(\text{turn}) = 0$, and set the flag to $i$. 

Proof of Main thm cont’d.

1. The value of \( \text{turn} \) eventually stabilizes to the value \( i \), of some contender.
Proof of Main thm cont’d.

1. The value of \( \text{turn} \) eventually stabilizes to the value \( i \), of some contender.

2. For any contender \( j \neq i \) it gets stuck in the first loop. Thus, their flag is set to 1.
Proof of Main thm cont’d.

1. The value of turn eventually stabilizes to the value $i$, of some contender.
2. For any contender $j \neq i$ it gets stuck in the first loop. Thus, their flag is set to 1.
3. Contender $i$ proceeds and sets its flag to 2, and is the only one to satisfy this and thus will enter the critical region.
# Table of Contents

1. Introduction, Problem Setup
2. Computational Model
3. Formalizing Mutual Exclusion
4. Dijkstra’s ME Algorithm
5. Dijkstra’s ME Analysis
6. Summary
Summary

- Introduced Shared Memory asynchronous model of computation.
- Motivated and Introduced Mutual Exclusion.
- Presented Dijkstra’s ME algorithm.
- Analyzed safety and liveness properties of Dijkstra’s algorithm.
Questions?
Thank You!