Group Mutual Exclusion in Linear Time and Space

Yuan He[1] K. Gopalakrishnan[2] Eli Gafni [1]

[1]Department of Computer Science
University of California, Los Angeles

[2]Department of Computer Science
East Carolina University

ICDCN, 2016

Structure of a process in Mutual Exclusion Problem

repeat

  Remainder Section

  Entry Section

  Critical Section

  Exit Section

forever

Mutual Exclusion Problem

- Mutual Exclusion Problem
  - A collection of asynchronous processes
  - Need to ensure that processes access the resource exclusively in Critical Section
  - Develop code for Entry Section and Exit Section to guarantee the mutual exclusion property
- Mutual Exclusion property
  - No two processes can be in the Critical Section at the same time
- Dijkstra introduced the problem in 1965
- Huge body of literature on this classical problem.

Motivation for Group Mutual Exclusion (GME) Problem

- A common prayer room allocated to all students.
- Students use the room on a first come first served basis.
- Students of different faith can not use the room to pray concurrently.
- Students of same faith can pray concurrently.
- A student cannot overtake another waiting student just because a member of her faith is currently praying.
- Yet, a student should be able to pray without waiting if only others of same denomination are present.
- Students can change their religion between different trips to the room, if they want!

Group Mutual Exclusion (GME) Problem

- Introduced by Joung in 2000
- A process picks up a session number when it leaves the Remainder Section (the session can be different in different invocations)
- Two processes are friendly processes if they have the same session number
- Friendly processes can be in the Critical Section at the same time
- Processes with different session numbers are called conflicting processes
- P1-Mutual Exclusion: No two conflicting processes can be in the Critical Section at the same time

Structure of a process in GME Problem

repeat
  Remainder Section
  Session := *mysession*
  Entry Section
  Critical Section
  Exit Section
forever

- An execution of the last three sections is called an invocation.

Additional Desirable Properties

- P2 - Starvation Freedom: Any process entering the Entry Section is guaranteed to enter the Critical Section eventually
- P3 - Concurrent Entry: In the absence of conflicting processes, a process is guaranteed to enter the Critical Section within a bounded number of its own steps
- P4 - Bounded Exit: A process in the Exit Section is guaranteed to leave it within a bounded number of its own steps

Additional Desirable Properties (cont.)

- To be fair, we want to allow processes enter the Critical Section in the same order in which they made requests.

Entry Section
Doorway
Waiting Room

P5 - First-Come-First-Served (FCFS): If process i finished the doorway before process j started the doorway and they are conflicting processes, then process j must not enter the Critical Section before process i

Additional Desirable Properties (cont.)

- Deadlock Freedom: Deadlocks cannot occur in the system
  - Deadlock: One or more processes are trying forever to enter the Critical Section, but no process ever does so
- Starvation Freedom $\Rightarrow$ Deadlock Freedom (converse is not true)
- Deadlock Freedom + FCFS $\Rightarrow$ Starvation Freedom
- GME Problem: Develop solution satisfying all five properties.

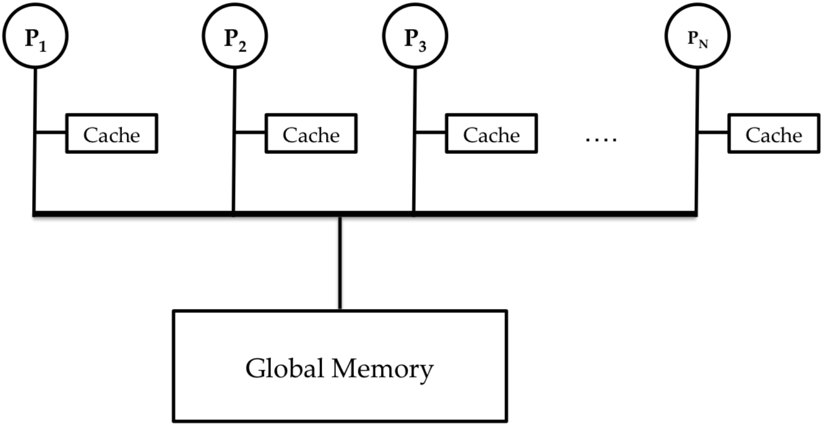
Model of Computation

- A system consisting of N processes and a set of shared variables
- Processes perform one of three actions
  - Perform some local computation
  - Read a shared variable
  - Write a shared variable
- Processes takes actions asynchronously
  - An unbounded number of other processes' actions can be executed between two successive actions of a process
- All processes are live
  - If a process has not terminated, it will eventually execute its next step

Memory Models

- Two common memory models
  - Distributed Shared Memory (DSM) model
  - Cache-Coherent (CC) model
- Focus on the CC model
  - All shared variables are located in a global memory module
  - Each processor has its private cache
  - When a process wants to write a shared variable
    - It writes the shared variable in the global memory
    - The Hardware protocol immediately invalidates the cached-values of this shared variable in other processors
  - When a process wants to read a shared variable
    - Check whether it is available in its local cache
    - If it is in the cache, then reads it
    - If it is not in the cache, then accesses the global memory, and migrates it to local cache and then reads it

Cache-Coherent Model



Time and Space Complexity

- Accessing the global memory requires more time than accessing the local cache
  - Requires interconnection network traffic
- Therefore, for time complexity, we only count the number of global memory accesses. This is called the Remote Memory Reference (RMR) complexity
- For space complexity, we only count the space of the global memory (shared space complexity)

Lamport's Bakery Algorithm

- Lamport developed the Bakery Algorithm in 1974
- Every process picks a token number larger than that of others.
- Process with the smallest token number enters the CS.
- Waits until other process has picked a number before comparing.
- Ties resolved using process id.

Generalizing Lamports Bakery Algorithm to GME

- Additionally, use a shared array for session numbers.
- Let processes in the Remainder Section have the session number of 0
- A process does not wait on another process if it has the same session number. This ensures the concurrent entry property.

shared variables:

Session: **array**[1..N] of **integer**, initially all 0
Choosing: **array**[1..N] of **boolean**, initially all false
Token: **array**[1..N] of **integer**, initially all 0

private variables:

mysession: **integer**, initially 0

GME Bakery Algorithm

repeat
    REMAINDER SECTION

    $Choosing[i] := \textbf{true}$
    $Session[i] := mysession$
    $Token[i] := 1+$ max of other token numbers
    $Choosing[i] := \textbf{false}$

    for $j := 1$ **to** N **do**
        await $((Choosing[j] = \textbf{false}) \vee (Session[j] \in \{0, mysession\}))$
        await
        $(((Token[i], i) < (Token[j], j)) \vee (Session[j] \in \{0, mysession\}))$
    end for

    CRITICAL SECTION

    $Token[i] := 0$
    $Session[i] := 0$
forever

Proof of Correctness

- Mutual Exclusion
  - A process with a larger token number is forced to wait on a conflicting process with a smaller token number. So no two conflicting processes can be in the Critical Section at the same time.
- Concurrent Entry
  - Process i does not wait on process j if it has the same session number as process j
- Bounded Exit
  - Exit Section is made up of two simple write instructions
- FCFS
  - If process i doorway-precedes process j and they are conflicting processes, then process j will have a larger token number than process i. Therefore, process j can not enter the Critical Section before process i.

Proof of Correctness (cont.)

- Deadlock Freedom
  - No process can wait at line 8 forever
    - **await** $((Choosing[j] = \text{false}\ ) \vee (Session[j] \in \{0, mysession\}))$
  - All active processes must wait on line 9.
    - **await** $(((Token[i], i) < (Token[j], j)) \vee (Session[j] \in \{0, mysession\}))$
  - Some process p has that has the smallest token number and that process will enter the Critical Section, contradiction!

- Starvation Freedom
  - Deadlock Freedom + FCFS $\Rightarrow$ Starvation Freedom

Previous attempts

- Takamura and Igarashi attempted to generalize Lamports Bakery Algorithm and developed three algorithms.
  - First algorithm: Simple, but does not satisfy the starvation freedom property.
  - Second algorithm: Satisfies starvation freedom, but does not satisfy the concurrent entry property and bounded exit property, despite being complex.
  - Third algorithm: Provides more concurrency than their second algorithm but still does not satisfy the properties of concurrent entry and bounded exit and remains complex.

Pros and Cons

Advantages

- Satisfy all five properties
- Simple and elegant
- Linear time and linear space

Disadvantages

The shared variable *Token* will grow in an unbounded manner

- How to overcome this disadvantage, while maintaining all advantages?

Flaw in the literature

- In 2001, Hadzilacos presented the first FCFS algorithm with bounded registers for GME.
- Hadzilacos claimed his algorithm has $O(N)$ RMR Complexity and $O(N^2)$ shared space complexity.
- He left it as an open problem to reduce the space complexity.
- His algorithm is a modular composition of a FCFS algorithm and an ME algorithm.
- The ME algorithm used is Burns-Lamport 1-bit algorithm.
- We show that Burns-Lamport algorithm actually has $O(N^2)$ RMR Complexity.
- This invalidates Hadzilaco's claim.

Flaw in the literature (cont.)

- In 2003, Jayanti et. al. came up with a clever modification to Hadzilacos's algorithm.
- This reduced the space complexity to $O(N)$.
- In view of Hadzilacos's erroneous claim, this algorithm was deemed to be of linear time and space.
- However, Jayanti retains the idea of modular composition and moreover uses Burns-Lamport ME algorithm.
- So, Jayanti's algorithm is also of $O(N^2)$ RMR Complexity.
- So, the problem of developing a linear time and linear space algorithm for GME with bounded registers is actually still open.
- We next present such an algorithm.
- It is a generalization of the elegant Black-White Bakery Algorithm developed by Gadi Taubenfeld.

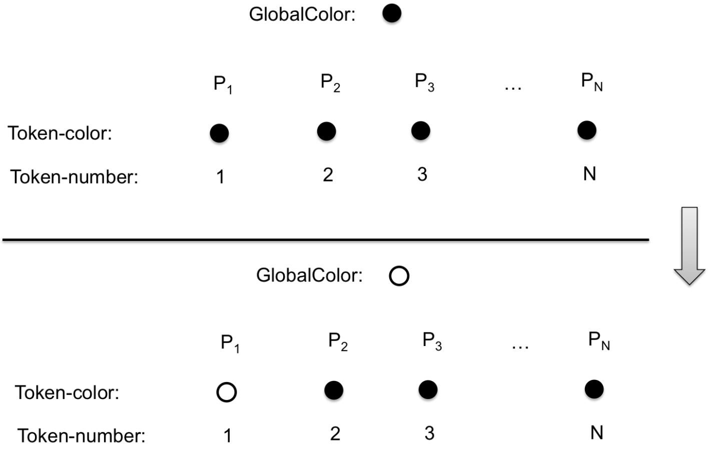
Idea of Black-White Bakery Algorithm

- The token: token-color and token-number
- Token-color: The same color as the current *GlobalColor*
- Token-number: Larger than token-numbers of all processes with the same token-color
- Waiting room: Waits until it has priority over all other processes and then enters the Critical Section
  - If two processes have the same token-color, the process with the smaller token-number has priority
  - If two processes have different token-colors, the process whose token-color is different from the *GlobalColor* has priority
- Exit Section: Updates the *GlobalColor* to be the opposite of its own token-color

Sketch of proof of correctness

- Processes whose token-colors are same with the *GlobalColor* will wait until processes whose token-colors are different finish their current invocation
- Mutual Exclusion
  - Processes with different token-colors are mutually excluded by the token-color
  - Processes with the same token-color are mutually excluded by the token-number
- Also satisfies the properties of starvation freedom, bounded exit and FCFS
- The maximum value of token-number is N, where N is the total number of processes

Bounding the Token-number



Generalizing Black-White Bakery Algorithm to solve GME

- Token has three parts: token-color, token-number and session-number.
- Token-color: Selects the token-color as in the Black-White Bakery Algorithm
- Token-number: Choose the maximum of token-number of conflicting process with the same token-color and then add 1.
- This strategy helps in controlling the growth of token-numbers
- To ensure concurrent entry, processes always check whether the other process requests the same session in all busy-wait loops.

Generalizing Black-White Bakery Algorithm to solve GME (cont.)

- In GME, processes with different token-colors can be in the Critical Section at the same time by the concurrent entry property, which is not true in the original Black-White Bakery Algorithm
- This makes it difficult to ensure the mutual exclusion property
- The process updates the *GlobalColor* as before, but only if its token-number is greater than 1 and there is no active process (not necessarily conflicting process) with the opposite token-color.
- This new color updating mechanism in the Exit Section is crucial in proving the mutual exclusion property.

Generalizing Black-White Bakery Algorithm to solve GME (cont.)

- Other properties also hold good.
- The algorithm has $O(N)$ RMR complexity and $O(N)$ shared space complexity
- The value of token-number can not grow beyond $N+1$. So, we are only using bounded registers.

Conclusions

We made three contributions.

1. Presented a linear time and linear space GME algorithm satisfying all five properties using unbounded registers. It is a simple generalization of Lamport's Bakery Algorithm.
2. We proved that the bounded register algorithms by Hadzilacos and Jayanti et al. are actually of $\Theta(N^2)$ RMR complexity.
3. Presented a linear time and linear space GME algorithm satisfying all five properties using bounded registers. It is a non-trivial generalization of Taubenfeld's Black-White Bakery Algorithm.

Open Problems

- Is it possible to bound the registers by a constant?
- Is it possible to ensure additional properties such as FIFE, Strong Concurrent Entering Property?
- Is it possible to devise a GME algorithm that has constant RMR complexity, perhaps by using more complex synchronization primitives such as Fetch&Add?