Multicore Programming Project

By Elior Migdal & Itay Bernstein

Introduction

This paper will address the Starvation-Freedom Mutual Exclusion synchronization problem in the Memory-Anonymous computing model. We will focus on two main articles:

1. Memory-Anonymous Starvation-Free Mutual Exclusion: Possibility and Impossibility Result
   An article that solves the existence of starvation-free mutual exclusion algorithms for two or more processes. The article discusses both possibility and impossibility results, as it shows that there is a memoryless starvation-free mutual exclusion algorithm for 2 processes, but not for 3 and above. ***This paper will mainly focus on the possibility results.***
   The algorithm provided uses an array of registers of size m where m is odd and $m \geq 7$, and each process is given a unique permutation of the array, canceling the a-priori agreement between processes on the names of shared memory locations. Both processes are trying to own $m - 2$ registers, and the first one who does may enter the critical section, while the other waits, and flags itself as waiting. As soon as one process leaves the critical section, it frees all shared resources and prioritizes the waiting process. By so, the algorithm achieves both starvation-freedom and mutual exclusion.

2. Coordination Without Prior Agreement
   An article that introduces the memory anonymous computing model and solves the Starvation-Freedom Mutual Exclusion synchronization problem for 2 processes, by providing an algorithm.

Summary

1. Memory-Anonymous Starvation-Free Mutual Exclusion: Possibility and Impossibility Results
   The main publication proves the following claims:
   a. *There is a symmetric memoryless starvation-free mutual exclusion algorithm for two processes using $m \geq 7$ anonymous registers* ***if and only if*** *m is odd.*
      To prove this claim, the publication provides an algorithm that satisfies the requirements above.
      The **only if** direction is proved in a referenced publication which we would cover next.

      1) A High-Level view of the algorithm
         Let A, B be 2 processes. Each process has a unique identifier.
         Both A, B are given an odd-sized array of registers of length $m \geq 7$, and each process is given a unique permutation of the array.
         Also, each one of the processes can write one of the following values to any register: Its unique identifier or a "Waiting" symbol which is similar for both processes. By writing its own identifier to a register, the process "owns" the register, while by writing the "Waiting" symbol, it signals the other process that it is waiting.
         WLOG, let's assume process A tries to obtain the lock.
         a) Step 1: Prioritizing a waiting process
            At first, A scans the shared array of registers, looking for the first "Free" register, which has a value of 0. After it finds one, it claims it, and then it searches for any registers that are signaled as "Waiting". If it finds any, it will free its owned register and wait until the other process is done waiting. This part is basically how starvation-freedom is guaranteed.
         b) Step 2: Owning enough registers
            Later, A tries to own $m - 2$ registers. If it succeeded, A would go on and enter its CS.
            Else, A would signal waiting in at-most 2 registers, and free the rest, if it owns any, and then A would wait until B will release the CS, and then enter the CS without owning any more registers.
         c) Step 3: Releasing the Critical Section
            Once A is done, it will free up its owned registers, and start over at step 1.

2) Proof of correctness
Let A, B be two processes.

The **mutual exclusion** attribute is proved by assuming the contrary. In the proof, it is shown that a process may enter its critical section in two ways: Either it owns $m-2$ registers, or it waits for the other process to release the critical section, and then it enters.

a) It is shown that it is impossible for both processes A, B to own $m-2$ registers, since the losing process may own $\left\lfloor \frac{m}{2} \right\rfloor$ registers at most, and then it waits until the winning process has released the critical section.
b) It is also impossible for both to write "Waiting" and later enter the critical section, which means that both own less than $\left\lceil \frac{m}{2} \right\rceil$ of the registers.
c) WLOG, it is impossible for process A to signal "Waiting", and later enter the critical section, while process B owns $m-2$ registers, since process A waits until all registers are free or waiting.
d) It is impossible for one process to own $m-2$ registers and then enter the critical section, while the other process has signaled "Waiting", waited for the critical section to be free, and then entered its critical section, since the algorithm starts by prioritizing the waiting process.

The **starvation freedom** attribute is proved by analyzing each one of the 3 loops of the algorithm. WLOG, we'll assume process A is stuck in each loop, and analyze it. (The lines are referencing the original lines of pseudo-code of the algorithm.)

a) A is stuck in loop#1 (lines 6-8): That means, A has prioritized B as the waiting process. That means A will be in loop#1 until B exits its critical section and frees all registers.
b) A is stuck in loop#2 (lines 24-26): That means, A has tried to own $m-2$ registers, but failed, and therefore must wait until all registers are "free" or "waiting". Since A lost, the only "waiting" registers are written by A, which means when B frees his owned registers, even if B tries to immediately acquire the lock again, it will prioritize A as the waiting process, and therefore A would exit from loop#2.
c) A is stuck in loop#3 (outer loop of lines 10-29): That means, A is not stuck in loop#2, and therefore owns at least $\left\lceil \frac{m}{2} \right\rceil$ of the registers, and waiting for B to free its owned registers, which are less than $\left\lceil \frac{m}{2} \right\rceil$ of the registers. When B does free its owned registers, it would free A from the loop.

*b. There is **no** symmetric memoryless starvation-free mutual exclusion algorithm for $n \geq 3$ processes using any number of anonymous registers.*
Let A, B, C be 3 processes.
The main idea is that, by appropriately assigning names to registers, A, B can run in their remainder sections, writing to all registers before either enters its critical section. This is done by renaming unwritten registers on the fly, ensuring that if both A, B are about to write to the same unwritten register, they write to distinct ones. This approach hides the write operations of process C, preventing it from entering its critical section.

2. Coordination Without Prior Agreement
This article introduces the memory anonymous computing model: A model in which there is no a-priori agreement on registers' names and provides a Mutual Exclusion Deadlock-Freedom algorithm for 2 processes.

a. Memory anonymity
Memory anonymity means that there is no a-priori agreement on the names of shared registers between different processes. Saying R is a shared register between process A and process B. In a memory-anonymous system, A may know R as S_A, while B may know R as S_B, but both will be referencing the same resource. It is assumed that all the registers of an anonymous memory are assumed to be initialized to the same value (otherwise, they may be distinguished).

b. Theorem:
There is a memory-anonymous symmetric deadlock-free mutual exclusion algorithm for 2 processes using $m \geq 2$ registers if and only if m is odd.

Proof:

1) A High-Level view of the algorithm
   Let A, B be 2 processes. Each process has a unique identifier.
   Both A, B are given an odd-sized array of registers of length $m \geq 3$, and each process is given a unique permutation of the array.
   Each of the processes can write its own unique identifier into each register.
   WLOG, we'll assume A is trying to acquire the lock.
   a) At first, A attempts to own all m registers by writing its identifier in them.
      i. If A owns at least $\left\lceil \frac{m}{2} \right\rceil$ of the registers, it means A "won" and B must free its owned registers so that A could own all of them.
      ii. Else, A frees its owned registers and waits until B releases the critical section.
   b) A enters the critical section.
   c) A release all the registers and (maybe) attempts to acquire the lock again.

2) Proof of correctness
   The **mutual exclusion** attribute is guaranteed by the main loop condition, which is that to enter its critical section, a process must own **all** shared registers. Since each process has a unique identifier, only one process can own **all** the shared registers at a single point, and therefore only one process can enter its critical section at a time.

   The **deadlock-freedom** attribute is guaranteed by the fact that there is an odd number of registers, so it is guaranteed that one process will own less than $\left\lceil \frac{m}{2} \right\rceil$ of the registers, and then free its owned registers for the other process to own them and enter its critical section. Since each process releases all registers at the exit code, it guarantees that the next time a process tries to acquire the lock, a process may succeed.

Now, the "only if" direction is proved by proving a more general theorem:
There is a memory-anonymous symmetric deadlock-free mutual exclusion algorithm for n processes using $m \geq 2$ registers only if for every positive integer $1 < l \leq n$, m and l are relatively prime.

This theorem is proved by assuming the contrary, and constructing a scenario the leads to violation of deadlock-freedom and mutual exclusion. The scenario is as follwed:

1) Consider each process with an initial register and an ordering for scanning registers. Registers are arranged in a unidirectional ring of size m.
2) Pick l processes, assign them the same ring ordering but potentially different initial registers. The initial registers are spaced such that the distance between any two neighboring initial registers is m/l.
3) Run the ll processes in lock steps, meaning each process takes one step at a time in sequence. Since only equality comparisons are allowed, processes taking the same number of steps remain in the same state.
4) Due to identical states and steps, symmetry cannot be broken. Thus, either all processes enter their critical sections simultaneously (violating mutual exclusion), or none can enter (violating deadlock-freedom). Contradiction.

Suggestion

We wanted to make the algorithm more efficient in some way, and figured the proof was solid enough that you can't outright improve it. As the algorithm uses non-atomic registers, threads can overwrite one another and thus, the algorithm needs an odd number of 7 or more registers to ensure mutual exclusion and starvation freedom. We thought that by switching out non-atomic registers for atomic ones, we could reduce the number of registers needed to ensure the two prior requirements and as a result, potentially reduce the memory used in the implementation of the lock.

By using atomic registers, we can reduce the number of registers being used in the lock to 3 and still ensure the requirements above:

Mutual Exclusion:

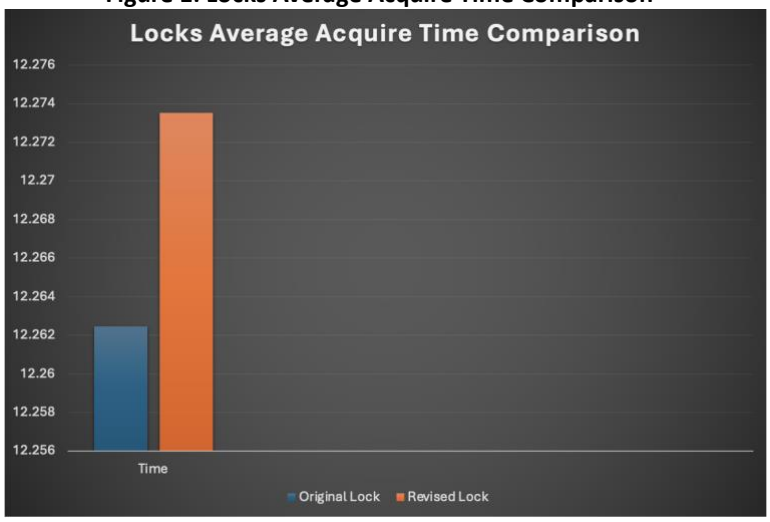
By using atomic registers' CAS, you can ensure mutual exclusion with only 3 registers due to only changing the value of the register if the expected value is the same as the value currently in the register.

Starvation Freedom:

Due to the mutual exclusion being guaranteed by only writing into 2 registers to be able to enter the CS, the "losing" thread only needs to write a waiting signal on a single register, that register will not be overwritten by the winning thread and thus the thread will be able to signal waiting successfully and eventually acquire the lock.

So, we implemented the algorithm using 3 atomic registers instead of 7 non-atomic registers. We tested the average lock acquiring time on both versions of the lock, and got the following results:

Figure 1: Locks Average Acquire Time Comparison*



* Note that:
a) All number values are ms.
b) Each process had a critical section of ~10ms.
c) Using 3 atomic registers may theoretically reduce space complexity, but we couldn't test it as we couldn't predict the Garbage Collector behavior.

We concluded that implementing the lock using 3 atomic registers worsens the average lock acquiring time, even if by a small fraction.

For implementations, please visit our GitHub repository.

References

1. G. Taubenfeld. Memory-Anonymous Starvation-Free Mutual Exclusion: Possibility and Impossibility Result.
2. G. Taubenfeld. Coordination with prior agreement.
3. G. Taubenfeld. Anonymous shared memory.