Mutual Exclusion in Shared-memory Systems

From Theory to Practice

Qi-An Fu
fugoes.qa@gmail.com
https://blog.fugoes.xyz

Tunight at November 9, 2019

Outline

Outline

Dijkstra's mutual exclusion algorithm

- The first distributed algorithm (year 1965).
- Shared memory model
  - Atomic read/write register
  - Fix number of processes
- Mutual exclusion (safety): multiple processes share a resource, only one can enter a piece of code called critical section.
- Progress (liveness): if some processes want to enter the critical section, then eventually some process enters the critical section.

Dijkstra's mutual exclusion algorithm (cont'd)

- n processes labeled with 1, 2, $\cdots$, n
- Shared (atomic) variables
  - status[1..n] $\in$ { IDLE, GET_TURN, CHECK }, initially all set to IDLE.
  - turn $\in \{1, 2, \cdots, n\}$, initially arbitrary.
- For process i (with local variable j):

```
L0  status[i] = GET_TURN
L1  repeat
L2      while turn != i do if status[turn] == IDLE then turn = i
L3      status[i] = CHECK
L4      for j != i do if status[j] == CHECK then status[i] = GET_TURN
L5  until status[i] == CHECK
L6  { critical section }
L7  status[i] = IDLE
```

Correctness: mutual exclusion

Safety

```
L0  status[i] = GET_TURN
L1  repeat
L2      while turn != i do if status[turn] == IDLE then turn = i
L3      status[i] = CHECK
L4      for j != i do if status[j] == CHECK then status[i] = GET_TURN
L5  until status[i] == CHECK
L6  { critical section }
L7  status[i] = IDLE
```

Proof by contradiction: Assume there are two processes labeled a and b both inside L6 at some time t_0.

- At some time $t_1 < t_0$, a executes L3, after which a enters L6 successfully.
- At some time $t_2 < t_0$, b executes L3, after which b enters L6 successfully.

Correctness: mutual exclusion (cont'd)

Safety

```
L0  status[i] = GET_TURN
L1  repeat
L2      while turn != i do if status[turn] == IDLE then turn = i
L3      status[i] = CHECK
L4      for j != i do if status[j] == CHECK then status[i] = GET_TURN
L5  until status[i] == CHECK
L6  { critical section }
L7  status[i] = IDLE
```

- Without loss of generality, assume $t_1 < t_2$.
- b would find `status[a] == CHECK` inside L4 loop, so b would set `status[b]` to `GET_TURN` in L4, which makes the condition in L5 not true.
- b would not enter L6. Contradiction!

Correctness: progress

Liveness

```
L0  status[i] = GET_TURN
L1  repeat
L2      while turn != i do if status[turn] == IDLE then turn = i
L3      status[i] = CHECK
L4      for j != i do if status[j] == CHECK then status[i] = GET_TURN
L5  until status[i] == CHECK
L6  { critical section }
L7  status[i] = IDLE
```

Proof by contradiction: Assume there won't be any process successfully entering L6 after t_0 (We set t_0 to the earliest time satisfying the assumption).

- At some time $t_1 > t_0$, `status[turn] == IDLE`.
- After some time $t_2 > t_1$, all processes who wants to enter L6 will have `status` of either `GET_TURN` or `CHECK`.

Correctness: progress (cont'd)

Liveness

```
L0  status[i] = GET_TURN
L1  repeat
L2      while turn != i do if status[turn] == IDLE then turn = i
L3      status[i] = CHECK
L4      for j != i do if status[j] == CHECK then status[i] = GET_TURN
L5  until status[i] == CHECK
L6  { critical section }
L7  status[i] = IDLE
```

- After t_2, `turn` won't change.
- After t_2, `turn` points to some process `i` who wants to enter L6.
- After t_2, process `i` would eventually enters L6, since all other processes who wants to enter L6 are stuck in the L2 loop. Contradiction!

However

What is the model of computation?

- What is time point?
- What is after?
- What is shared variables?
- What is local variables?
- What is atomic?
- ...

Outline

Shared memory model: Shared object

A shared memory unit is called a shared object, which has:

- Values
- Operations (could have return values, and could change the object's value)

Examples of shared object

- Read/write register `R`
  - Values: integers (bounded?)
  - Operations: `read(R)`, `write(R, v)`
    - `read(R)` returns the current value of `R`, `write(R, v)` has no return value.
    - `read(R)` does not change the value, `write(R, v)` change the value of `R` to `v`.
- Single-writer/multi-reader register
  - Only one process can write, others can read.
- Multi-writer/multi-reader register
  - All processes can read/write to the register.

Shared memory model: Process

n processes $p_1, p_2, \ldots, p_n$

- Each process has a (possibly infinite) state machine.
- Each process has a set of states, one of which is the initial state.

Shared memory model: Process (cont'd)

- A set of state variables var1, var2, ... are used to represent states (local variables).
- Each state q corresponds to a particular set of values of the state variables, q.var1, q.var2, ... (function evaluation).
- Each state q has a special field q.wait $\in \{\texttt{TRUE}, \texttt{FALSE}\}$, initially FALSE:
  - TRUE: q is waiting for an operation on a shared object to complete.
- Each state q of process p has three special fields:
  - q.obj: the object to be accessed next (could be null).
  - q.op: the operation on q.obj to be executed.
  - q.in: the input parameter (if any) of the q.op.

Shared memory model: Configuration

- Configuration C of the system: states of all processes and values of all shared objects
  - $(q_1, \ldots, q_n, v_1, \ldots, v_m)$.
  - q_i is the state of process p_i.
  - v_j is the value of object o_j.
- Initial Configuration: All processes are in their initial states and all objects contain their initial values.

Shared memory model: Computation steps

- When p takes a step in a normal state (a.k.a p's state q satisfy q.wait is `FALSE`):
  - Communication step: If q.obj is not null.
    - Process p invoke operation q.op on q.obj with the input value q.in (if any)
    - p transitions to a new state q', where q'.wait is `TRUE`.
  - Local computation step: If $q.obj$ is null.
    - p transitions to a new state q', where q'.wait is `FALSE`.
- When p takes a step in a waiting state (a.k.a p's state q satisfy q.wait is `TRUE`):
  - Return from the invocation step: p takes the response v from the operation q.op on q.obj with parameter q.in, and p transitions to a new state q', where q'.wait is `FALSE`.

Shared memory model: Executions

- Execution: $C_0, s_1, C_1, s_2, C_2, s_3, C_3, \ldots$
  - Step s_j is on configuration C_{j-1}.
  - Application of s_j to C_{j-1} results in C_j.
  - C_0 is the initial configuration.
  - Could be finite or infinite.
  - Asynchronous: the number of steps between the two steps of the same process is not bounded.

Shared memory model: Summary

- Shared object access takes time.
- Each step is one of:
  - Communication step.
  - Local computation step.
  - Return from the invocation step.
- Computation is a sequence of steps that changes configurations.
- Processes are asynchronous.

Recall the proof

- Time point $\Leftrightarrow$ Configuration
- After some time point $\Leftrightarrow$ Configurations following some configuration
- Local variables
- Shared variables
  - Atomic?

Atomic single-writer/multi-reader register

- Values: integers (bounded?)
- Operations: Read() -> v and Write(v)
- All Read()s and Write(v)s can be made in a sequential order.
- If operation o1 completes before operation o2 starts, then o1 is ordered before o2 in the sequential order.
- In the sequential order, the semantics of reads and writes are preserved, i.e. a read returns the latest written value before the read.

Atomic single-writer/multi-reader register (cont'd)

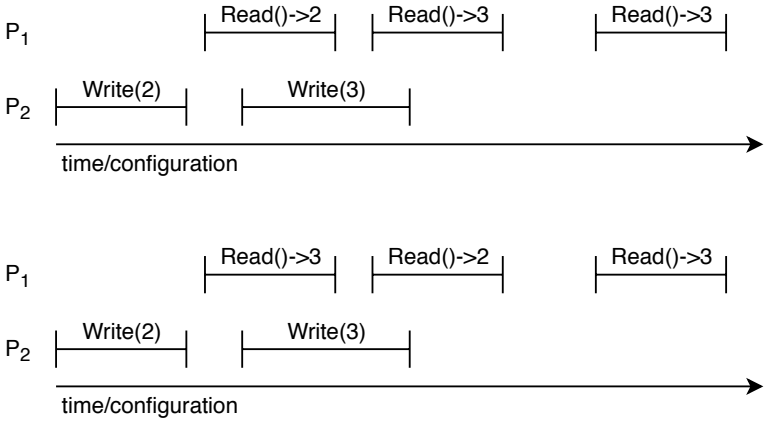


Figure: The first one is atomic, while the second one is not atomic.

Mutual exclusion problem

The algorithm needs to satisfy the following properties:

- Mutual exclusion: In every configuration of every execution, at most one process is in the critical section.
- Progress (no deadlock): In every execution, if some process is in the trying section in a configuration, then there is a later configuration in which some process is in the critical section.

Addition useful properties:

- No lockout (no starvation): In every execution, if some process is in the trying section in a configuration, then there is a later configuration in which the same process is in the critical section.

Dijkstra's mutual exclusion algorithm (revisited)

- Multi-reader multi-writer atomic register (bounded).
- Mutual exclusion.
- Progress.
- Starvation.
- Use one n-valued atomic register, and n 3-valued atomic registers.
- The paper *Bounds on Shared Memory for Mutual Exclusion* in 1993 shows that:
  - Algorithms that solves the mutual exclusion problem with atomic registers should use at least as many shared atomic registers as processes number n.
  - There exists some algorithm with n shared atomic registers that solves the problem and these registers are 1-bit.
  - So that n 1-bit atomic registers is the tight lower bound for this problem.

Outline

C++11's new feature <atomic>

```
#include <atomic>
// For C11: #include <stdatomic.h>
// For Rust: use std::sync::atomic;
```

Let's code in C++: The Header

dijkstra.hpp

```
#ifndef DIJKSTRA_HPP
#define DIJKSTRA_HPP

#include <atomic>

struct Dijkstra {
  static const int N = 4, IDLE = 0, GET_TURN = 1, CHECK = 2;
  struct alignas(64) Status { std::atomic_int v_{IDLE}; };

  alignas(64) std::atomic_int turn_{0};
  alignas(64) Status status_[N];

  void lock(int p);
  void unlock(int p);
};

#endif //DIJKSTRA_HPP
```

Let's code in C++: The Header (cont'd)

- The magic number explained (false sharing)

```
> getconf LEVEL1_DCACHE_LINESIZE
64
```

- C++!

```
// since C++17
#include <new>
std::hardware_destructive_interference_size;
std::hardware_constructive_interference_size;
```

How to test

test_dijkstra.cpp

```cpp
#include <cstdio>
#include <thread>
#include "dijkstra.hpp"

int main() {
  std::thread ts[Dijkstra::N];
  Dijkstra lock;
  std::atomic_flag flag = ATOMIC_FLAG_INIT;
  for (int i = 0; i < Dijkstra::N; i++) {
    ts[i] = std::thread([i, &lock, &flag] {
      for (int k = 0; k < 10000000; k++) {
        lock.lock(i);
        if (flag.test_and_set(
            std::memory_order_acquire)) {
          printf("ERROR\n");
          std::exit(-1);
        }
        flag.clear(std::memory_order_release);
        lock.unlock(i);
      }
    });
  }
  for (auto &t: ts) t.join();
  printf("PASS\n");
  return 0;
}
```

Dijkstra's algorithm made wrong

dijkstra_wrong.cpp

```cpp
#include "dijkstra.hpp"
void Dijkstra::lock(int p) {
  status_[p].v_.store(
      GET_TURN, std::memory_order_release);
  for (;;) {
    for (;;) {
      auto turn = turn_.load(
          std::memory_order_acquire);
      if (turn == p) break;
      auto status = status_[turn].v_.load(
          std::memory_order_acquire);
      if (status == IDLE) turn_.store(p,
          std::memory_order_release);
    }
    status_[p].v_.store(
        CHECK, std::memory_order_release);
    bool success = true;
    for (int i = 0; i < N; i++) {
      if (i == p) continue;
      auto status = status_[i].v_.load(
          std::memory_order_acquire);
      if (status == CHECK) {
        status_[p].v_.store(
            GET_TURN, std::memory_order_release);
        success = false; break;
      }
    }
    if (success) return;
  }
}
void Dijkstra::unlock(int p) {
  status_[p].v_.store(
      IDLE, std::memory_order_release);
}
```

Dijkstra's algorithm made correct

dijkstra.cpp

```cpp
#include "dijkstra.hpp"
void Dijkstra::lock(int p) {
  status_[p].v_.store(
      GET_TURN, std::memory_order_release);
  for (;;) {
    for (;;) {
      auto turn = turn_.load(
          std::memory_order_acquire);
      if (turn == p) break;
      auto status = status_[turn].v_.load(
          std::memory_order_acquire);
      if (status == IDLE) turn_.store(p,
          std::memory_order_release);
    }
    status_[p].v_.store(
        CHECK, std::memory_order_seq_cst);
    bool success = true;
    for (int i = 0; i < N; i++) {
      if (i == p) continue;
      auto status = status_[i].v_.load(
          std::memory_order_seq_cst);
      if (status == CHECK) {
        status_[p].v_.store(
            GET_TURN, std::memory_order_release);
        success = false; break;
      }
    }
    if (success) return;
  }
}
void Dijkstra::unlock(int p) {
  status_[p].v_.store(
      IDLE, std::memory_order_release);
}
```

From theory to practice: Memory order in C++

```
#include <atomic>
std::memory_order_acquire;
std::memory_order_release;
std::memory_order_seq_cst;
```

From theory to practice: Memory order in C++ (cont'd)

Acquire and release

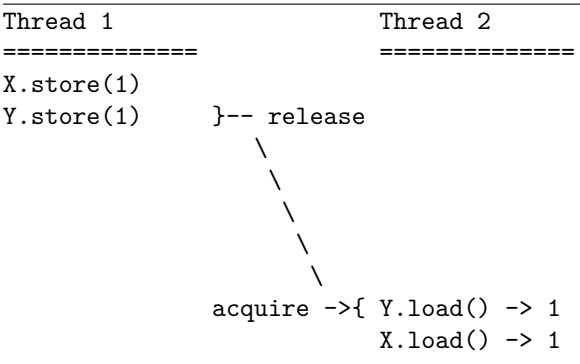
Initially `X == 0` and `Y == 0`

Thread 1	Thread 2
`X.store(1)`	`Y.load()`
`Y.store(1)`	`X.load()`

- If `Y.load()` returns 1, then `X.load()` returns 1.
- `Y.load() -> 1`, `X.load() -> 0` not possible.

From theory to practice: Memory order in C++ (cont'd)

Acquire and release



From theory to practice: Memory order in C++ (cont'd)

Acquire and release

`std::memory_order_acquire`: A load operation with this memory order performs the acquire operation on the affected memory location: no reads or writes in the current thread can be reordered before this load. All writes in other threads that release the same atomic variable are visible in the current thread.
`std::memory_order_release`: A store operation with this memory order performs the release operation: no reads or writes in the current thread can be reordered after this store. All writes in the current thread are visible in other threads that acquire the same atomic variable and writes that carry a dependency into the atomic variable become visible in other threads that consume the same atomic.
— https://en.cppreference.com/w/cpp/atomic/memory_order

From theory to practice: Memory order in C++ (cont'd)

Acquire and release

Is `load(std::memory_order_acquire)` and `store(std::memory_order_release)` atomic? No
Initially `X == 0`:

```
Thread 1                    Thread 2
==============              ==============
X.store(1)
                            X.load() -> 0
```

- No violation of acquire/release memory order.
- Not atomic.
- Acquire/release is not enough for Dijkstra's algorithm's safety property.

From theory to practice: Memory order in C++ (cont'd)

Sequentially-consistent

> *`std::memory_order_seq_cst`: Atomic operations tagged `memory_order_seq_cst` not only order memory the same way as release/acquire ordering, but also establish a single total modification order of all atomic operations that are so tagged.*
> — *https://en.cppreference.com/w/cpp/atomic/memory_order*

- `load(std::memory_order_seq_cst)` and `store(std::memory_order_seq_cst)` is atomic.

Correctness: mutual exclusion (revisited)

Safety

```
L0  status[i] = GET_TURN
L1  repeat
L2      while turn != i do if status[turn] == IDLE then turn = i
L3      status[i] = CHECK
L4      for j != i do if status[j] == CHECK then status[i] = GET_TURN
L5  until status[i] == CHECK
L6  { critical section }
L7  status[i] = IDLE
```

- In the correct implementation `dijkstra.cpp`, the `store(status[i], CHECK)` in L3 is atomic, the `load(status[j])` in L4 is also atomic.
- The proof of safety property only relies on these two atomic operations.

Correctness: progress (revisited)

Liveness

```
L0  status[i] = GET_TURN
L1  repeat
L2      while turn != i do if status[turn] == IDLE then turn = i
L3      status[i] = CHECK
L4      for j != i do if status[j] == CHECK then status[i] = GET_TURN
L5  until status[i] == CHECK
L6  { critical section }
L7  status[i] = IDLE
```

- Note that all requirements are state in 'at some time', 'after some time'.
- Acquire/release operations guarantee local writes would be visible globally eventually.

Let's read the assembly!

```
<Dijkstra::lock(int)>:
   0:        movslq %esi,%rcx
   3:        lea    0x40(%rdi),%r9
   7:        add    $0x1,%rcx
   b:        shl    $0x6,%rcx
   f:        add    %rdi,%rcx
  12:        movl   $0x1,(%rcx)
  18:        movslq (%rdi),%rax
  1b:        cmp    %eax,%esi
  1d:        je     40 <Dijkstra::lock(int)+0x40>
  1f:        add    $0x1,%rax
  23:        shl    $0x6,%rax
  27:        add    %rdi,%rax
  2a:        mov    (%rax),%eax
  2c:        test   %eax,%eax
  2e:        jne    18 <Dijkstra::lock(int)+0x18>
  30:        mov    %esi,(%rdi)
  32:        movslq (%rdi),%rax
  35:        cmp    %eax,%esi
  37:        jne    1f <Dijkstra::lock(int)+0x1f>
  39:        nopl   0x0(%rax)
  40:        mov    %r9,%rdx
  43:        xor    %eax,%eax
  45:        movl   $0x2,(%rcx)
  4b:        mfence
  4e:        cmp    %eax,%esi
  50:        je     5b <Dijkstra::lock(int)+0x5b>
  52:        mov    (%rdx),%r8d
  55:        cmp    $0x2,%r8d
  59:        je     70 <Dijkstra::lock(int)+0x70>
  5b:        add    $0x1,%eax
  5e:        add    $0x40,%rdx
  62:        cmp    $0x4,%eax
  65:        jne    4e <Dijkstra::lock(int)+0x4e>
  67:        retq
  68:        nopl   0x0(%rax,%rax,1)
  6f:
  70:        movl   $0x1,(%rcx)
  76:        jmp    18 <Dijkstra::lock(int)+0x18>
  78:        nopl   0x0(%rax,%rax,1)
  7f:

<Dijkstra::unlock(int)>:
  80:        movslq %esi,%rsi
  83:        add    $0x1,%rsi
  87:        shl    $0x6,%rsi
  8b:        add    %rsi,%rdi
  8e:        movl   $0x0,(%rdi)
  94:        retq
```

From theory to practice: Memory order on x86_64

In a multiple-processor system, the following ordering principles apply:

- *Individual processors use the same ordering principles as in a single-processor system.*
- *Writes by a single processor are observed in the same order by all processors.*
- *Writes from an individual processor are NOT ordered with respect to the writes from other processors.*
- *Memory ordering obeys causality (memory ordering respects transitive visibility).*
- *Any two stores are seen in a consistent order by processors other than those performing the stores.*
- *Locked instructions have a total order.*

— *Intel 64 and IA-32 Architectures Software Developer's Manual*

Go beyond

`mutex` in `libc` (Linux kernel): use `CAS` and `futex`.

- Fast path: `CAS` in user space
- Under contention:
  - Exponential back off
    - pause instruction
    - https://code.woboq.org/userspace/glibc/sysdeps/generic/adaptive_spin_count.h.html
    - https://www.gnu.org/software/libc/manual/html_node/Tunables.html
    - https://aloiskraus.wordpress.com/2018/06/16/why-skylakex-cpus-are-sometimes-50-slower-how-intel-has-broken-existin
  - `futex` wait

Reading materials

- *Intel 64 and IA-32 Architectures Software Developer's Manual* (Vol. 3A 8.2 Memory Ordering)
- https://www.kernel.org/doc/Documentation/memory-barriers.txt
- https://en.cppreference.com/w/cpp/atomic/memory_order
- *The Art of Multiprocessor Programming*

References I


Hagit Attiya, Rachid Guerraoui, Danny Hendler, Petr Kuznetsov, Maged M Michael, and Martin Vechev.
Laws of order: expensive synchronization in concurrent algorithms cannot be eliminated.
In *ACM SIGPLAN Notices*, volume 46, pages 487–498. ACM, 2011.


James E Burns and Nancy A Lynch.
Bounds on shared memory for mutual exclusion.
Information and Computation, 107(2):171–184, 1993.


EW Dijkstra.
Solution of a problem in concurrent programming control.
Communications of the ACM, 8(9):569, 1965.

References II


Leslie Lamport.
A new solution of dijkstra's concurrent programming problem.
Communications of the ACM, 17(8):453–455, 1974.