Chapter 18

Mutual exclusion

For full details see [AW04, Chapter 4] or [Lyn96, Chapter 10].

18.1 The problem

The goal is to share some critical resource between processes without more than one using it at a time—this is *the* fundamental problem in time-sharing systems.

The solution is to only allow access while in a specially-marked block of code called a **critical section**, and only allow one process at a time to be in a critical section.

A **mutual exclusion protocol** guarantees this, usually in an asynchronous shared-memory model.

Formally: We want a process to cycle between states **trying** (trying to get into critical section), **critical** (in critical section), **exiting** (cleaning up so that other processes can enter their critical sections), and **remainder** (everything else—essentially just going about its non-critical business). Only in the trying and exiting states does the process run the mutual exclusion protocol to decide when to switch to the next state; in the critical or remainder states it switches to the next state on its own.

The ultimate payoff is that mutual exclusion solves for systems without failures what consensus solves for systems with failures: if the only way to update a data structure is to hold a lock on it, we are guaranteed to get a nice clean sequence of atomic-looking updates. Of course, once we allow failures back in, mutex becomes less useful, as our faulty processes start crashing without releasing their locks, and with the data structure in some

broken, half-updated state.[1]

18.2 Goals

(See also [AW04, §4.2], [Lyn96, §10.2].)

Core mutual exclusion requirements:

Mutual exclusion At most one process is in the critical state at a time.

No deadlock (progress) If there is at least one process in a trying state, then eventually some process enters a critical state; similarly for exiting and remainder states.

Note that the protocol is not required to guarantee that processes leave the critical or remainder state, but we generally have to insist that the processes at least leave the critical state on their own to make progress.

An additional useful property (not satisfied by all mutual exclusion protocols; see [Lyn96, §10.4)]:

No lockout (**lockout-freedom**): If there is a particular process in a trying or exiting state, that process eventually leaves that state. This means that I don't starve because somebody else keeps jumping past me and seizing the critical resource before I can.

Stronger starvation guarantees include explicit time bounds (how many rounds can go by before I get in) or **bounded bypass** (nobody gets in more than k times before I do). Each of these imply lockout-freedom assuming no deadlock.

18.3 Mutual exclusion using strong primitives

See [AW04, §4.3] or [Lyn96, 10.9]. The idea is that we will use some sort of **read-modify-write** register, where the RMW operation computes a new value based on the old value of the register and writes it back as a single atomic operation, usually returning the old value to the caller as well.

[1] In principle, if we can detect that a process has failed, we can work around this problem by allowing some other process to bypass the lock and clean up. This may require that the original process leaves behind notes about what it was trying to do, or perhaps copies the data it is going to modify somewhere else before modifying it. But even this doesn't work if some zombie process can suddenly lurch to life and scribble its ancient out-of-date values all over our shiny modern data structure.

18.3.1 Test and set

A **test-and-set** operation does the following sequence of actions atomically:

```
1 oldValue ← read(bit)
2 write(bit, 1)
3 return oldValue
```

Typically there is also a second **reset** operation for setting the bit back to zero. For some implementations, this reset operation may only be used safely by the last process to get 0 from the test-and-set bit.

Because a test-and-set operation is atomic, if two processes both try to perform test-and-set on the same bit, only one of them will see a return value of 0. This is not true if each process simply executes the above code on a stock atomic register: there is an execution in which both processes read 0, then both write 1, then both return 0 to whatever called the non-atomic test-and-set subroutine.

Test-and-set provides a trivial implementation of mutual exclusion, shown in Algorithm 18.1.

```
1 while true do
      // trying
2     while TAS(lock) = 1 do nothing
      // critical
3     (do critical section stuff)
      // exiting
4     reset(lock)
      // remainder
5     (do remainder stuff)
```

Algorithm 18.1: Mutual exclusion using test-and-set

It is easy to see that this code provides mutual exclusion, as once one process gets a 0 out of lock, no other can escape the inner while loop until that process calls the `reset` operation in its exiting state. It also provides progress (assuming the lock is initially set to 0); the only part of the code that is not straight-line code (which gets executed eventually by the fairness condition) is the inner loop, and if lock is 0, some process escapes it, while if lock is 1, some process is in the region between the `TAS` call and the `reset`

call, and so it eventually gets to `reset` and lets the next process in (or itself, if it is very fast).

The algorithm does *not* provide lockout-freedom: nothing prevents a single fast process from scooping up the lock bit every time it goes through the outer loop, while the other processes ineffectually grab at it just after it is taken away. Lockout-freedom requires a more sophisticated turn-taking strategy.

18.3.2 A lockout-free algorithm using an atomic queue

Basic idea: In the trying phase, each process enqueues itself on the end of a shared queue (assumed to be an atomic operation). When a process comes to the head of the queue, it enters the critical section, and when exiting it dequeues itself. So the code would look something like Algorithm 18.2.

Note that this requires a queue that supports a `peek` operation that returns the head of the queue. Not all implementations of queues have this property.

```
1 while true do
    // trying
2   enq(q, myId)
3   while peek(q) ≠ myId do nothing
    // critical
4   (do critical section stuff)
    // exiting
5   deq(q)
    // remainder
6   (do remainder stuff)
```

Algorithm 18.2: Mutual exclusion using a queue

Here the proof of mutual exclusion is that only the process whose ID is at the head of the queue can enter its critical section. Formally, we maintain an invariant that any process whose program counter is between the inner while loop and the call to `deq`(q) must be at the head of the queue; this invariant is easy to show because a process can't leave the while loop unless the test fails (i.e., it is already at the head of the queue), no `enq` operation changes the head value (if the queue is nonempty), and the `deq` operation (which does change the head value) can only be executed by a process already at the head (from the invariant).

Deadlock-freedom follows from proving a similar invariant that every element of the queue is the ID of some process in the trying, critical, or exiting states, so eventually the process at the head of the queue passes the inner loop, executes its critical section, and dequeues its ID.

Lockout-freedom follows from the fact that once a process is at position k in the queue, every execution of a critical section reduces its position by 1; when it reaches the front of the queue (after some finite number of critical sections), it gets the critical section itself. Alternatively, we can argue lockout-freedom by showing bounded bypass: once I am in the queue, no process can execute two critical sections before I do, because once it leaves its first critical section, it enqueues behind me.

18.3.2.1 Replacing the queue with RMW

Following [AW04, §4.3.2], we can give an implementation of this algorithm using a single read-modify-write (RMW) register instead of a queue; this drastically reduces the (shared) space needed by the algorithm. The reason this works is because we don't really need to keep track of the position of each process in the queue itself; instead, we can hand out numerical tickets to each process and have the process take responsibility for remembering where its place in line is.

The RMW register has two fields, first and last, both initially 0. Incrementing last simulates an enqueue, while incrementing first simulates a dequeue. The trick is that instead of testing if it is at the head of the queue, a process simply remembers the value of the last field when it "enqueued" itself, and waits for the first field to equal it.

Algorithm 18.3 shows the code from Algorithm 18.2 rewritten to use this technique. The way to read the `RMW` operations is that the first argument specifies the variable to update and the second specifies an expression for computing the new value. Each `RMW` operation returns the old state of the object, before the update.

In practice, this algorithm is usually implemented using two objects, one of which implements a **fetch-and-increment** operation that increments a register and returns the value before the increment, and one of which is an ordinary atomic register. As in Algorithm 18.3, a process takes a position in line by calling the fetch-and-increment, and the head of the line is marked by the second register, which can only be incremented by a process in the exiting section. This implementation has the same properties of mutual exclusion and starvation-freedom as the single-RMW version.

```
1 while true do
    // trying
2   position ← RMW(V, ⟨V.first, V.last + 1⟩)
    // enqueue
3   while RMW(V, V).first ≠ position.last do
4     nothing
    // critical
5   (do critical section stuff)
    // exiting
6   RMW(V, ⟨V.first + 1, V.last⟩)
    // dequeue
    // remainder
7   (do remainder stuff)
```

Algorithm 18.3: Mutual exclusion using read-modify-write

18.4 Mutual exclusion and linearizability

Beyond controlling access to shared resources, mutual exclusion can instantly give us a linearizable implementation of any object for which we have a sequential implementation. The reason is that we can use a mutex to guard access to the shared data structure implementing the object.

Formally, we imagine that we have a read-modify-write object of some sort and an implementation from atomic registers that works for sequential executions. The simplest way to model this is to imagine that we have a single register r that contains the entire state of the object. A read-modify-write operation reads an old state q from r, computes a new state $f(q)$ and writes it back to r, and finally returns the old value q. This works as long as we don't have two or more processes executing operations concurrently. But we can enforce this with a mutex, as in Algorithm 18.4.

```
1 procedure RMW(f)
2   Enter critical section.
3   q ← r
4   r ← f(q)
5   Leave critical section.
6   return q
```

Algorithm 18.4: Building a concurrent RMW object using mutex

To show that this implementation is linearizable, observe that for any concurrent history H we can construct a sequential history S by assigning the invoke/respond times for each operation to when that operation enters and leaves the critical section. This gives a total order $<_S$ since no process can enter the critical section until the previous one leaves. Since the processes carry out the same operations on r in both H and S, both produce identical views. Given two operations $a <_H b$, a leaves its critical section before b enters its critical section, so $<_H \subseteq <_S$. We thus have a linearization of any given H.

18.5 Mutual exclusion using only atomic registers

While mutual exclusion is easier using powerful primitives, we can also solve the problem using only registers.

18.5.1 Peterson's algorithm

Algorithm 18.5 shows Peterson's lockout-free mutual exclusion protocol for two processes p_0 and p_1 [Pet81] (see also [AW04, §4.4.2] or [Lyn96, §10.5.1]). It uses only atomic registers.

This uses three bits to communicate: present[0] and present[1] indicate which of p_0 and p_1 are participating, and waiting enforces turn-taking. The protocol requires that waiting be multi-writer, but it's OK for present[0] and present[1] to be single-writer.

In the description of the protocol, we write Lines 8 and 10 as two separate lines because they include two separate read operations, and the order of these reads is important.

18.5.1.1 Correctness of Peterson's protocol

Intuitively, let's consider all the different ways that the entry code of the two processes could interact. There are basically two things that each process does: it sets its own present variable in Line 5 and grabs the waiting variable in Line 6. Here's a typical case where one process gets in first:

1. p_0 sets present[0] $\leftarrow 1$
2. p_0 sets waiting $\leftarrow 0$
3. p_0 reads present[1] $= 0$ and enters critical section

```
   shared data:
 1 waiting, initially arbitrary
 2 present[i] for i ∈ {0, 1}, initially 0
 3 Code for process i:
 4 while true do
       // trying
 5     present[i] ← 1
 6     waiting ← i
 7     while true do
 8         if present[¬i] = 0 then
 9             break
10         if waiting ≠ i then
11             break
       // critical
12     (do critical section stuff)
       // exiting
13     present[i] = 0
       // remainder
14     (do remainder stuff)
```

Algorithm 18.5: Peterson's mutual exclusion algorithm for two processes

4. p_1 sets $\mathsf{present}[1] \leftarrow 1$
5. p_1 sets $\mathsf{waiting} \leftarrow 1$
6. p_1 reads $\mathsf{present}[0] = 1$ and $\mathsf{waiting} = 1$ and loops
7. p_0 sets $\mathsf{present}[0] \leftarrow 0$
8. p_1 reads $\mathsf{present}[0] = 0$ and enters critical section

The idea is that if I see a 0 in your present variable, I know that you aren't playing, and can just go in.

Here's a more interleaved execution where the waiting variable decides the winner:

1. p_0 sets $\mathsf{present}[0] \leftarrow 1$
2. p_0 sets $\mathsf{waiting} \leftarrow 0$
3. p_1 sets $\mathsf{present}[1] \leftarrow 1$
4. p_1 sets $\mathsf{waiting} \leftarrow 1$
5. p_0 reads $\mathsf{present}[1] = 1$
6. p_1 reads $\mathsf{present}[0] = 1$
7. p_0 reads $\mathsf{waiting} = 1$ and enters critical section
8. p_1 reads $\mathsf{present}[0] = 1$ and $\mathsf{waiting} = 1$ and loops
9. p_0 sets $\mathsf{present}[0] \leftarrow 0$
10. p_1 reads $\mathsf{present}[0] = 0$ and enters critical section

Note that it's the process that set the waiting variable last (and thus sees its own value) that stalls. This is necessary because the earlier process might long since have entered the critical section.

Sadly, examples are not proofs, so to show that this works in general, we need to formally verify each of mutual exclusion and lockout-freedom. Mutual exclusion is a safety property, so we expect to prove it using invariants. The proof in [Lyn96] is based on translating the pseudocode directly into automata (including explicit program counter variables); we'll do essentially the same proof but without doing the full translation to automata. Below, we write that p_i is at line k if it the operation in line k is enabled but has not occurred yet.

Lemma 18.5.1. *If* present$[i] = 0$*, then* p_i *is at Line 5 or 14.*

Proof. Immediate from the code. □

Lemma 18.5.2. *If* p_i *is at Line 12, and* $p_{\neg i}$ *is at Line 8, 10, or 12, then* waiting $= \neg i$*.*

Proof. We'll do the case $i = 0$; the other case is symmetric. The proof is by induction on the schedule. We need to check that any event that makes the left-hand side of the invariant true or the right-hand side false also makes the whole invariant true. The relevant events are:

- Transitions by p_0 from Line 8 to Line 12. These occur only if present$[1] = 0$, implying p_1 is at Line 5 or 14 by Lemma 18.5.1. In this case the second part of the left-hand side is false.
- Transitions by p_0 from Line 10 to Line 12. These occur only if waiting $\neq 0$, so the right-hand side is true.
- Transitions by p_1 from Line 6 to Line 8. These set waiting to 1, making the right-hand side true.
- Transitions that set waiting to 0. These are transitions by p_0 from Line 6 to Line 10, making the left-hand side false.

□

We can now read mutual exclusion directly off of Lemma 18.5.2: if both p_0 and p_1 are at Line 12, then we get waiting $= 1$ and waiting $= 0$, a contradiction.

To show progress, observe that the only place where both processes can get stuck forever is in the loop at Lines 8 and 10. But then waiting isn't changing, and so some process i reads waiting $= \neg i$ and leaves. To show lockout-freedom, observe that if p_0 is stuck in the loop while p_1 enters the critical section, then after p_1 leaves it sets present$[1]$ to 0 in Line 13 (which lets p_0 in if p_0 reads present$[1]$ in time), but even if it then sets present$[1]$ back to 1 in Line 5, it still sets waiting to 1 in Line 6, which lets p_0 into the critical section. With some more tinkering this argument shows that p_1 enters the critical section at most twice while p_0 is in the trying state, giving 2-bounded bypass; see [Lyn96, Lemma 10.12]. With even more tinkering we get a constant time bound on the waiting time for process i to enter the critical section, assuming the other process never spends more than $O(1)$ time inside the critical section.

18.5.1.2 Generalization to n processes

(See also [AW04, §4.4.3].)

The easiest way to generalize Peterson's two-process algorithm to n processes is to organize a tournament in the form of log-depth binary tree; this method was invented by Peterson and Fischer [PF77]. At each node of the tree, the roles of the two processes are taken by the winners of the subtrees, i.e., the processes who have entered their critical sections in the two-process algorithms corresponding to the child nodes. The winner of the tournament as a whole enters the real critical section, and afterwards walks back down the tree unlocking all the nodes it won in reverse order. It's easy to see that this satisfies mutual exclusion, and not much harder to show that it satisfies lockout-freedom—in the latter case, the essential idea is that if a winner at some node reaches the root infinitely often, then lockout-freedom at that node means that a winner of each child node reaches the root infinitely often.

The most natural way to implement the nodes is to have present[0] and present[1] at each node be multi-writer variables that can be written to by any process in the appropriate subtree. Because the present variables don't do much, we can also implement them as the OR of many single-writer variables (this is what is done in [Lyn96, §10.5.3]), but there is no immediate payoff to doing this since the waiting variables are still multi-writer.

Nice properties of this algorithm are that it uses only bits and that it's very fast: $O(\log n)$ time in the absence of contention.

18.5.2 Fast mutual exclusion

With a bit of extra work, we can reduce the no-contention cost of mutual exclusion to $O(1)$, while keeping whatever performance we previously had in the high-contention case. The trick (due to Lamport [Lam87]) is to put an object at the entrance to the protocol that diverts a solo process onto a "fast path" that lets it bypass the n-process mutex that everybody else ends up on.

Our presentation mostly follows [AW04][§4.4.5], which uses the **splitter** abstraction of Moir and Anderson [MA95] to separate out the mechanism for diverting a lone process.[2] Code for a splitter is given in Algorithm 18.6.

A splitter assigns to each processes that arrives at it the value right, down, or stop. The useful properties of splitters are that if at least one process

[2] Moir and Anderson call these things **one-time building blocks**, but the name **splitter** has become standard in subsequent work.

```
   shared data:
1  atomic register race, big enough to hold an ID, initially ⊥
2  atomic register door, big enough to hold a bit, initially open
3  procedure splitter(id)
4      race ← id
5      if door = closed then
6          return right
7      door ← closed
8      if race = id then
9          return stop
10     else
11         return down
```

Algorithm 18.6: Implementation of a splitter

arrives at a splitter, then (a) at least one process returns right or stop; and (b) at least one process returns down or stop; (c) at most one process returns stop; and (d) any process that runs by itself returns stop. The first two properties will be useful when we consider the problem of **renaming** in Chapter 25; we will prove them there. The last two properties are what we want for mutual exclusion.

The names of the variables race and door follow the presentation in [AW04, §4.4.5]; Moir and Anderson [MA95], following Lamport [Lam87], call these X and Y. As in [MA95], we separate out the right and down outcomes—even though they are equivalent for mutex—because we will need them later for other applications.

The intuition behind Algorithm 18.6 is that setting door to closed closes the door to new entrants, and the last entrant to write its ID to race wins (it's a slow race), assuming nobody else writes race and messes things up. The added cost of the splitter is always $O(1)$, since there are no loops.

To reset the splitter, write open to door. This allows new processes to enter the splitter and possibly return stop.

Lemma 18.5.3. *After each time that* door *is set to* open*, at most one process running Algorithm 18.6 returns* stop*.*

Proof. To simplify the argument, we assume that each process calls `splitter` at most once.

Let t be some time at which door is set to open ($-\infty$ in the case of the initial value). Let S_t be the set of processes that read open from door after

time t and before the next time at which some process writes closed to door, and that later return stop by reaching Line 9.

Then every process in S_t reads door before any process in S_t writes door. It follows that every process in S_t writes race before any process in S_t reads race. If some process p is not the *last* process in S_t to write race, it will not see its own ID, and will not return stop. But only one process can be the last process in S_t to write race.[3] □

Lemma 18.5.4. *If a process runs Algorithm 18.6 by itself starting from a configuration in which* door = open*, it returns* stop*.*

Proof. Follows from examining a solo execution: the process sets race to id, reads open from door, then reads id from race. This causes it to return stop as claimed. □

To turn this into an n-process mutex algorithm, we use the splitter to separate out at most one process (the one that gets stop) onto a **fast path** that bypasses the **slow path** taken by the rest of the processes. The slow-path process first fight among themselves to get through an n-process mutex; the winner then fights in a 2-process mutex with the process (if any) on the fast path.

Releasing the mutex is the reverse of acquiring it. If I followed the fast path, I release the 2-process mutex first then reset the splitter. If I followed the slow path, I release the 2-process mutex first then the n-process mutex. This gives mutual exclusion with $O(1)$ cost for any process that arrives before there is any contention ($O(1)$ for the splitter plus $O(1)$ for the 2-process mutex).

A complication is that if nobody wins the splitter, there is no fast-path process to reset it. If we don't want to accept that the fast path just breaks forever in this case, we have to include a mechanism for a slow-path process to reset the splitter if it can be assured that there is no fast-path process left in the system. The simplest way to do this is to have each process mark a bit in an array to show it is present, and have each slow-path process, while still holding all the mutexes, check on its way out if the door bit is set and no processes claim to be present. If it sees all zeros (except for itself) after seeing door = closed, it can safely conclude that there is no fast-path process and reset the splitter itself. The argument then is that the last slow-path process to leave will do this, re-enabling the fast path once there is

[3] It's worth noting that this last process still might not return stop, because some later process—not in S_t—might overwrite race. This can happen even if nobody ever resets the splitter.

no contention again. This approach is taken implicitly in Lamport's original algorithm, which combines the splitter and the mutex algorithms into a single miraculous blob.

18.5.3 Lamport's Bakery algorithm

See [AW04, §4.4.1] or [Lyn96, §10.7] for some textbook presentations; the original algorithm is appears in [Lam74].

This is a lockout-free mutual exclusion algorithm that uses only single-writer registers (although some of the registers may end up holding arbitrarily large values). Code for the Bakery algorithm is given as Algorithm 18.7.

```
   shared data:
1  choosing[i], an atomic bit for each i, initially 0
2  number[i], an unbounded atomic register, initially 0
3  Code for process i:
4  while true do
       // trying
5      choosing[i] ← 1
6      number[i] ← 1 + max_{j≠i} number[j]
7      choosing[i] ← 0
8      for j ≠ i do
9          loop until choosing[j] = 0
10         loop until number[j] = 0 or ⟨number[i], i⟩ < ⟨number[j], j⟩
       // critical
11     (do critical section stuff)
       // exiting
12     number[i] ← 0
       // remainder
13     (do remainder stuff)
```

Algorithm 18.7: Lamport's Bakery algorithm

Note that several of these lines are actually loops; this is obvious for Lines 9 and 10, but is also true for Line 6, which includes an implicit loop to read all $n - 1$ values of number$[j]$.

Intuition for mutual exclusion is that if you have a lower number than I do, then I block waiting for you; for lockout-freedom, eventually I have the smallest number. (There are some additional complications involving the choosing bits that we are sweeping under the rug here.) For a real proof

see [AW04, §4.4.1] or [Lyn96, §10.7].

Selling point is a strong near-FIFO guarantee and the use of only single-writer registers (which need not even be atomic—it's enough that they return correct values when no write is in progress). Weak point is unbounded registers.

18.6 RMR complexity

It's not hard to see that we can't build a shared-memory mutex without busy-waiting: any process that is waiting can't detect that the critical section is safe to enter without reading a register, but if that register tells it that it should keep waiting, it is back where it started and has to read it again. This makes our standard step-counting complexity measures useless for describe the worst-case complexity of a mutual exclusion algorithm.

However, the same argument that suggests we can ignore local computation in a message-passing model suggests that we can ignore local operations on registers in a shared-memory model. Real multiprocessors have memory hierarchies where memory that is close to the CPU (or one of the CPUs) is generally much faster than memory that is more distant. This suggests charging only for **remote memory references**, or RMRs, where each register is local to one of the processes and only operations on non-local registers are expensive. This has the advantage of more accurately modeling real costs [MCS91, And90], and allowing us to build busy-waiting mutual exclusion algorithms with costs we can actually analyze.

As usual, there is a bit of a divergence here between theory and practice. Practically, we are interested in algorithms with good real-time performance, and RMR complexity becomes a heuristic for choosing how to assign memory locations. This gives rise to very efficient mutual exclusion algorithms for real machines, of which the most widely used is the beautiful MCS algorithm of Mellor-Crummey and Scott [MCS91]. Theoretically, we are interested in the question of how efficiently we can solve mutual exclusion in our formal model, and RMR complexity becomes just another complexity measure, one that happens to allow busy-waiting on local variables.

18.6.1 Cache-coherence vs. distributed shared memory

The basic idea of RMR complexity is that a process doesn't pay for operations on local registers. But what determines which operations are local?

In the **cache-coherent** model (CC for short), once a process reads a register it retains a local copy as long as nobody updates it. So if I do a

sequence of read operations with no intervening operations by other processes, I may pay an RMR for the first one (if my cache is out of date), but the rest are free. The assumption is that each process can cache registers, and there is some cache-coherence protocol that guarantees that all the caches stay up to date. We may or may not pay RMRs for write operations or other read operations, depending on the details of the cache-coherence protocol, but for upper bounds it is safest to assume that we do.

In the **distributed shared memory** model (DSM), each register is assigned permanently to a single process. Other processes can read or write the register, but only the owner gets to do so without paying an RMR. Here memory locations are nailed down to specific processes.

In general, we expect the cache-coherent model to be cheaper than the distributed shared-memory model, if we ignore constant factors. The reason is that if we run a DSM algorithm in a CC model, then the process p to which a register r is assigned incurs an RMR only if some other process q accesses p since p's last access. But then we can amortize p's RMR by charging q double. Since q incurs an RMR in the CC model, this tells us that we pay at most twice as many RMRs in DSM as in CC for any algorithm.

The converse is not true: there are (mildly exotic) problems for which it is known that CC algorithms are asymptotically more efficient than DSM algorithms [Gol11, DH04].

18.6.2 RMR complexity of Peterson's algorithm

As a warm-up, let's look at the RMR complexity of Peterson's two-process mutual exclusion algorithm (Algorithm 18.5). Acquiring the mutex requires going through mostly straight-line code, except for the loop that tests $\mathsf{present}[\neg i]$ and $\mathsf{waiting}$.

In the DSM model, spinning on $\mathsf{present}[\neg i]$ is not a problem (we can make it a local variable of process i). But $\mathsf{waiting}$ is trouble. Whichever process we don't assign it to will pay an RMR every time it looks at it. So Peterson's algorithm behaves badly by the RMR measure in this model.

Things are better in the CC model. Now process i may pay RMRs for its first reads of $\mathsf{present}[\neg i]$ and $\mathsf{waiting}$, but any subsequent reads are free unless process $\neg i$ changes one of them. But any change to either of the variables causes process i to leave the loop. It follows that process i pays at most 3 RMRs to get through the busy-waiting loop, giving an RMR complexity of $O(1)$.

RMR complexities for parts of a protocol that access different registers add just like step complexities, so the Peterson-Fischer tree construction

described in §18.5.1.2 works here too. The result is $O(\log n)$ RMRs per critical section access, but only in the CC model.

18.6.3 Mutual exclusion in the DSM model

Yang and Anderson [YA95] give a mutual exclusion algorithm for the DSM model that requires $\Theta(\log n)$ RMRs to reach the critical section. This is now known to be optimal for deterministic algorithms [AHW08]. The core of the algorithm is a 2-process mutex similar to Peterson's, with some tweaks so that each process spins only on its own registers. Pseudocode is given in Algorithm 18.8; this is adapted from [YA95, Figure 1].

```
1  C[side(i)] ← i
2  T ← i
3  P[i] ← 0
4  rival ← C[¬side(i)]
5  if rival ≠ ⊥ and T = i then
6  |  if P[rival] = 0 then
7  |  |  P[rival] = 1
8  |  while P[i] = 0 do spin
9  |  if T = i then
10 |  |  while P[i] ≤ 1 do spin
   // critical section goes here
11 C[side(i)] ← ⊥
12 rival ← T
13 if rival ≠ i then
14 |  P[rival] ← 2
```

Algorithm 18.8: Yang-Anderson mutex for two processes

The algorithm is designed to be used in a tree construction where a process with ID in the range $\{1 \ldots n/2\}$ first fights with all other processes in this range, and similarly for processes in the range $\{n/2+1 \ldots n\}$. The function side(i) is 0 for the first group of processes and 1 for the second. The variables $C[0]$ and $C[1]$ are used to record which process is the winner for each side, and also take the place of the present variables in Peterson's algorithm. Each process has its own variable $P[i]$ that it spins on when blocked; this variable is initially 0 and ranges over $\{0, 1, 2\}$; this is used to signal a process that it is safe to proceed, and tests on P substitute for tests

on the non-local variables in Peterson's algorithm. Finally, the variable T is used (like waiting in Peterson's algorithm) to break ties: when $T = i$, it's i's turn to wait.

Initially, $C[0] = C[1] = \perp$ and $P[i] = 0$ for all i.

When I want to enter my critical section, I first set $C[\texttt{side}(i)]$ so you can find me; this also has the same effect as setting $\mathsf{present}[\texttt{side}(i)]$ in Peterson's algorithm. I then point T to myself and look for you. I'll block if I see $C[\neg\texttt{side}(i)] \neq \perp$ and $T = i$. This can occur in two ways: one is that I really write T after you did, but the other is that you only wrote $C[\neg\texttt{side}(i)]$ but haven't written T yet. In the latter case, you will signal to me that T may have changed by setting $P[i]$ to 1. I have to check T again (because maybe I really did write T later), and if it is still i, then I know that you are ahead of me and will succeed in entering your critical section. In this case I can safely spin on $P[i]$ waiting for it to become 2, which signals that you have left.

There is a proof that this actually works in [YA95], but it's 27 pages of very meticulously-demonstrated invariants (in fairness, this includes the entire algorithm, including the tree parts that we omitted here). For intuition, this is not much more helpful than having a program mechanically check all the transitions, since the algorithm for two processes is effectively finite-state if we ignore the issue with different processes i jumping into the role of $\texttt{side}(i)$.

A slightly less rigorous but more human-accessible proof would be analogous to the proof of Peterson's algorithm. We need to show two things: first, that no two processes ever both enter the critical section, and second, that no process gets stuck.

For the first part, consider two processes i and j, where $\texttt{side}(i) = 0$ and $\texttt{side}(j) = 1$. We can't have both i and j skip the loops, because whichever one writes T last sees itself in T. Suppose that this is process i and that j skips the loops. Then $T = i$ and $P[i] = 0$ as long as j is in the critical section, so i blocks. Alternatively, suppose i writes T last but does so after j first reads T. Now i and j both enter the loops. But again i sees $T = i$ on its second test and blocks on the second loop until j sets $P[i]$ to 2, which doesn't happen until after j finishes its critical section.

Now let us show that i doesn't get stuck. Again we'll assume that i wrote T second.

If j skips the loops, then j sets $P[i] = 2$ on its way out as long as $T = i$; this falsifies both loop tests. If this happens after i first sets $P[i]$ to 0, only i can set $P[i]$ back to 0, so i escapes its first loop, and any j' that enters from the 1 side will see $P[i] = 2$ before attempting to set $P[i]$ to 1, so $P[i]$ remains at 2 until i comes back around again. If j sets $P[i]$ to 2 before i sets

$P[i]$ to 0 (or doesn't set it at all because $T = j$, then $C[\texttt{side}(j)]$ is set to $\perp$ before i reads it, so i skips the loops.

If j doesn't skip the loops, then $P[i]$ and $P[j]$ are both set to 1 after i and j enter the loopy part. Because j waits for $P[j] \neq 0$, when it looks at T the second time it will see $T = i \neq j$ and will skip the second loop. This causes it to eventually set $P[i]$ to 2 or set $C[\texttt{side}(j)]$ to $\perp$ before i reads it as in the previous case, so again i eventually reaches its critical section.

Since the only operations inside a loop are on local variables, the algorithm has $O(1)$ RMR complexity. For the full tree this becomes $O(\log n)$.

18.6.4 Lower bounds

For deterministic algorithms, there is a lower bound due to Attiya, Hendler, and Woelfel [AHW08] that shows that any one-shot mutual exclusion algorithm for n processes incurs $\Omega(n \log n)$ total RMRs in either the CC or DSM models (which implies that some single process incurs $\Omega(\log n)$ RMRs). This is based on an earlier breakthrough lower bound of Fan and Lynch [FL06] that proved the same lower bound for the number of times a register changes state. Both bounds are information-theoretic: a family of $n!$ executions is constructed containing all possible orders in which the processes enter the critical section, and it is shown that each RMR or state change only contributes $O(1)$ bits to choosing between them.

For randomized algorithms, Hendler and Woelfel [HW11] have an algorithm that uses $O(\log n / \log \log n)$ expected RMRs against an adaptive adversary, beating the deterministic lower bound. This is the best possible for an adaptive adversary, due to a matching lower bound of Giakkoupis and Woelfel [GW12b] that holds even for systems that provide compare-and-swap objects.

For an oblivious adversary, an algorithm of Giakkoupis and Woelfel [GW14] achieves $O(1)$ expected RMRs using compare-and-swap in the DSM model. A more recent algorithm of Giakkoupis and Woelfel [GW17] gives the same $O(1)$ expected RMRs in the CC model; this also uses compare-and-swap. Curiously, there also exist linearizable $O(1)$-RMR implementations of CAS from registers in this model [GHHW12]; however, it is not clear that these implementations can be combined with the Giakkoupis-Woelfel algorithm to give $O(1)$ expected RMRs using registers, because variations in scheduling of randomized implementations may produce subtle conditioning that gives different behavior from actual atomic objects in the context of a randomized algorithm [GHW11].

18.7 Space complexity

There is a famous result due to Burns and Lynch [BL93] that any mutual exclusion protocol using only read/write registers requires at least n of them. Details are in [Lyn96, §10.8]. A slightly different version of the argument is given in [AW04, §4.4.4]. The proof is another nice example of an indistinguishability proof, where we use the fact that if a group of processes can't tell the difference between two executions, they behave the same in both.

Assumptions: We have a protocol that guarantees mutual exclusion and progress. Our base objects are all atomic registers.

Key idea: In order for some process p to enter the critical section, it has to do at least one write to let the other processes know it is doing so. If not, they can't tell if p ever showed up at all, so eventually either some p' will enter the critical section and violate mutual exclusion or (in the no-p execution) nobody enters the critical section and we violate progress. Now suppose we can park a process p_i on each register r_i with a pending write to i; in this case we say that p_i **covers** r_i. If every register is so covered, we can let p go ahead and do whatever writes it likes and then deliver all the covering writes at once, wiping out anything p did. Now the other processes again don't know if p exists or not. So we can say something stronger: before some process p can enter a critical section, it has to write to an uncovered register.

The hard part is showing that we can cover all the registers without letting p know that there are other processes waiting—if p can see that other processes are waiting, it can just sit back and wait for them to go through the critical section and make progress that way. So our goal is to produce states in which (a) processes $p_1 \ldots, p_k$ (for some k) between them cover k registers, and (b) the resulting configuration is indistinguishable from an **idle configuration** to $p_{k+1} \ldots p_n$, where an idle configuration is one in which every process is in its remainder section.

Lemma 18.7.1. *Starting from any idle configuration C, there exists an execution in which only processes $p_1 \ldots p_k$ take steps that leads to a configuration C' such that (a) C' is indistinguishable by any of $p_{k+1} \ldots p_n$ from some idle configuration C'' and (b) k distinct registers are covered by $p_1 \ldots p_k$ in C'.*

Proof. The proof is by induction on k. For $k = 0$, let $C'' = C' = C$.

For larger k, the essential idea is that starting from C, we first run to a configuration C_1 where $p_1 \ldots p_{k-1}$ cover $k - 1$ registers and C_1 is indistinguishable from an idle configuration by the remaining processes, and

then run p_k until it covers one more register. If we let $p_1 \dots p_{k-1}$ go, they overwrite anything p_k wrote. Unfortunately, they may not come back to covering the same registers as before if we rerun the induction hypothesis (and in particular might cover the same register that p_k does). So we have to look for a particular configuration C_1 that not only covers $k-1$ registers but also has an extension that covers the same $k-1$ registers.

Here's how we find it: Start in C. Run the induction hypothesis to get C_1; here there is a set W_1 of $k-1$ registers covered in C_1. Now let processes p_1 through p_{k-1} do their pending writes, then each enter the critical section, leave it, and finish, and rerun the induction hypothesis to get to a state C_2, indistinguishable from an idle configuration by p_k and up, in which $k-1$ registers in W_2 are covered. Repeat to get sets W_3, W_4, etc. Since this sequence is unbounded, and there are only $\binom{r}{k-1}$ distinct sets of registers to cover (where r is the number of registers), eventually we have $W_i = W_j$ for some $i \neq j$. The configurations C_i and C_j are now our desired configurations covering the same $k-1$ registers.

Now that we have C_i and C_j, we run until we get to C_i. We now run p_k until it is about to write some register not covered by C_i (it must do so, or otherwise we can wipe out all of its writes while it's in the critical section and then go on to violate mutual exclusion). Then we let the rest of p_1 through p_{k-1} do all their writes (which immediately destroys any evidence that p_k ran at all) and run the execution that gets them to C_j. We now have $k-1$ registers covered by p_1 through p_{k-1} and a k-th register covered by p_k, in a configuration that is indistinguishable from idle: this proves the induction step. □

The final result follows by the fact that when $k = n$ we cover n registers; this implies that there are n registers to cover.

It's worth noting that the execution constructed in this proof might be *very, very long*. It's not clear what happens if we consider executions in which, say, the critical section is only entered a polynomial number of times. If we are willing to accept a small probability of failure over polynomially-many entries, there is a randomized mutual exclusion protocol that uses $O(\log n)$ space [AHTW18], at the cost of $O(n)$ amortized RMR complexity in the cache-coherent model. It is still open whether it is possible to reduce the space complexity below $O(n)$ for polynomial-length executions without allowing for a small probability of failure or without having such high RMR complexity.