Lecture Notes for CSCI 379: Distributed Computing

Set 3-Mutual Exclusion

Professor Talmage

February 13, 2023

Our goal here is two-fold: First, we want to explore the mutual exclusion problem, as it is ubiquitous in computing, distributed and otherwise. Second, we want to expand our comfort with different models of computation. Specifically, we'll introduce some of the formalism for working in shared memory.

1 Shared Memory

The idea of a shared memory model is that there are data objects with a defined interface, and processes can interact with those objects exactly as if they were local, sequential objects. So if your code uses a queue, heap, variable, etc., it will access the shared versions the same way. Thus, it is much easier to code in this model. One just needs to be careful with the logic of what you expect to find in a memory object, as another process may have changed it since you last looked.

1.1 Data Type Specifications

A data type is specified by an *Abstract Data Type*, which indicates how an object behaves, but omits all implementation details.

Definition 1. An *Abstract Data Type (ADT)* specifies the following:

1. Domain of possible values (often implicit)
2. Operations, including parameter and return types. Typically $OP(arg, ret)$, with $arg = -$ or $ret = -$ if there is no argument or return value, respectively.
3. What value is returned to any invocation of an operation. Two ways to specify this:
    - Sequences of legal operation instances (invocation-return pairs–These are separate in a distributed system since interesting things can happen between them.).
    - States and state transitions. This tends to be less general and portable, since it tends to include implementation details.

Example: An **integer-valued** register:

- Operations: $Read(-, v), Write(v, -)$ where $v \in \mathbb{Z}$
- Legal sequences:
    - The empty sequence is legal.
    - If ρ is a legal sequence of operation instances, then $\rho \cdot Write(v, -)$ is legal.

- If ρ is a legal sequence of operation instances, then $\rho \cdot Read(-, x)$ is legal iff x was the argument of the last $Write$ in ρ. If there is no $Write$ in ρ, then the sequence is legal if x is the initial value of the register.

Exercise:

- Write the specification for a Stack ADT in sequence style.
- Write the specification for an integer-valued register in the state style.
- Add a *CompareAndSwap* operation to your register. *CompareAndSwap* takes two parameters. If the current value equals the first parameter, store the second. Else, do nothing.
- Modify the legal-sequence style specification to allow a *CompareAndSwap*.

1.2 Shared Memory System

- n processes $p_0, \ldots, p_{n-1}$
- m shared memory objects $R_0, \ldots, R_{m-1}$. Each object is specified by an Abstract Data Type (ADT).
- A configuration of a system is now a list of configurations of all processes and current states of all shared objects (typically represented by a sequence of completed operation instances).
  - We will want to discuss configurations which are similar to each other: Two configurations C and C' are *P-similar* for a set P of processes if all processes in P and all shared objects are in the same states in C and C'. We write this as $C \sim_P C'$.
- All events are computation steps. An event at process p_i is
  1. Based on its current state, p_i invokes an operation $op(arg)$ on a particular shared object R_t.
  2. R_t performs the operation instance $op(arg, ret)$.
  3. p_i changes state, with ret as an input to the transition function.
- An event is *atomic*. That is, while asynchrony can delay and mix up events at different processes, an event cannot partially happen, then pause while another process does something, then return.
  - This is why changing ADTs changes what can be done, as it changes what can be guaranteed to happen together.
  - For example, *CompareAndSwap* guarantees that the check of current value and setting, if needed, happen together, without another $Write$ changing the value between reading and writing.
- Admissibility: Executions are still infinite. When a process terminates, it may no longer change any shared objects.
- Pseudocode looks exactly like sequential code. However, some of the data objects are shared, and you must account for the possibility of remote operations on those objects when proving an algorithm's correctness.

Complexity

- Message complexity is gone, as there are no messages.
- Instead, we care about shared space complexity. How many shared objects, storing how many bits/distinct values, and of what ADTs must we have to solve a problem?
- Time complexity can be tricky. Possibilities include number of computation steps, number of operations on shared objects, etc.

Exercise: Can you solve Leader Election in a network where all processes share a *Read/Write* register? A FIFO queue? A register with *CompareAndSwap*? For what size network do your algorithms work? *(Hint: Try $n = 2$.)*

2 Mutual Exclusion

The mutual exclusion problem is the essence of resource allocation: If multiple processes need something, but only one can have it at a time, how can we guarantee that that happens?

Problem 1. Each process has a designated portion of its code, called the *critical section*. Guarantee that at most one process is in its critical section at any give time.

- Think of the critical section as the code which uses some particular, limited, shared resource. For example, accessing a GPU, science instrument, display, network controller, memory bus, etc. If multiple processes were to access these at the same time, they would conflict with each other and destroy their work.
- In distributed computing, Mutex is often used to control access to a particular set of shared variables to allow one process to use them for a time as if they were not shared, then pass them along to the next process.
- Don't need to worry about the Mutual Exclusion code needing to know what's in the critical code, as that can just be a call to some other function. (As long as the critical code doesn't touch the Mutex object, but we assume the critical code exists first, then we write a wrapper function.)

To discuss solutions to the Mutual Exclusion problem, we divide code into four sections, which are executed in a loop infinitely:

1. Entry: This process wants to enter its critical section
2. Critical
3. Exit: This process has finished its critical section, cleanup to allow next access
4. Remainder: Everything else

We say that an algorithm solves the Mutual Exclusion problem if it provides code for the Entry and Exit sections which guarantees that at most one process is in Critical at a time, for *any* code in the Critical and Remainder sections, subject to

- Processes must exit Critical after a finite amount of time
- Entry and Exit can create and use variables (including shared objects) that are not touched by Critical and Remainder code.

We adjust the admissibility requirement: Each process must take an infinite number of steps or terminate in its Remainder section. (Don't want a process terminating in Critical, preventing Exit from making the resource available to other processes.)

Exercise: Can you think of a way to solve Mutual exclusion that would be useless? In general, are there more properties that we would like to provide? What are other "bad" solutions, and how might we exclude them?

There are several properties we may consider when designing a Mutual Exclusion algorithm:

- **Mutual Exclusion**: At most one process in Critical at a time. The fundamental corretness condition.
- **No-Deadlock**: If there is a configuration in an execution with some process in Entry, there is a later configuration in that execution where a process is in Critical.
- **No-Lockout**: If there is a configuration in an execution with some process in Entry, there is a later configuration in that execution where *that same* process is in Critical.

3 Mutual Exclusion with Strong Shared Objects

3.1 *Test&Set*

For a first pass, we will assume that we have access to strong shared memory objects (we'll discuss later what makes a shared object "strong"). Consider a binary register with *Test&Set*. It will have the following operations and behavior (note that we no longer allow direct *Read* and *Write* calls):

- *Test&Set*: If the register currently contains 0, set it to 1, return 0. Otherwise, leave it alone, return 1.
- *Reset*: Set register to 0

Test&Set is powerful because it atomically reads and writes the register, with no possibility of another process changing what it holds between testing it and setting it.

Exercise: Devise a Mutual Exclusion algorithm using *Test&Set*.

- Recall: You write code for Entry and Exit.
- Use calls to shared objects to determine when to run Critical.
- Ensure some process will enter Critical.
- Ensure no other process will enter Critical until the first Exits.

Complexity We only use one, single-bit *Test&Set* register.

Correctness

Exercise: Which of the three correctness properties we discussed does this algorithm provide?

- Mutual Exclusion

Algorithm 1 Code for process p_i to solve Mutual Exclusion using a $Test\&Set$ register R.

```
    Entry:
1:  while R.Test&Set() ≠ 0 do
2:  end while
    Critical
    Exit:
3:  R.reset()
    Remainder
```

- No deadlock
- Lockout is possible, since we don't know the order of processes running their $Test\&Set$s.

> **Exercise:** How would you set up a correctness proof for this algorithm? What do you actually need to prove?

Proof by contradiction: Assume some p_j enters Critical while some p_i is already in it. WLOG, assume this is the first violation. p_i set R to 1 when it entered, and no process sets R to 0 until it exits, so some other process must have exited Critical since p_i entered it, contradicting the assumption that this was the first violation.

3.2 RMW Algorithm

Now, we'll extend the strength of a $Test\&Set$ to an even more powerful operation: $Read\text{-}Modify\text{-}Write$, or RMW. RMW takes a function as a parameter, which determines how to modify the register's value.

- $RMW(f)$: Suppose r is the register's value.

```
1: function RMW(f)
2:     temp = r
3:     r = f(r)
4:     return temp
5: end function
```

> **Exercise:** Express binary $Test\&Set$ as a RMW function by choosing an appropriate f.

Next, we'll use a register with a RMW operation to solve the Mutual Exclusion problem. This is evidently possible, since we already discussed that $Test\&Set$ is a special case of RMW.

> **Exercise:** Why would we even bother with RMW if we can just use $Test\&Set$?

This code is perhaps a bit bizarre or confusing, so let's break it down:

- We can use a register to hold two values using a consistent encoding scheme–half the bits for the first value, half for the second, etc.
- When we call RMW, we have to specify *how* it should modify/write to the register. Thus, we have to pass a function. In real code, you might do this with lambdas or similar. Here, we have mathematical/set notation for functions. For example, the first says that if V holds $(head, tail)$, replace it with $(head, tail + 1)$.
- Passing the identity function to RMW makes it act like a simple $Read$ function.

Algorithm 2 Code for p_i, Solving Mutual Exclusion using *Read-Modify-Write* register V which holds a pair of values

Entry
1: $position = V.RMW(\{(head, tail + 1) \mid (head, tail)\})$
2: $queue = V.RMW(id)$ ▷ Identity function
3: **while** $queue[0] \neq position[1]$ **do**
4: $queue = V.RMW(id)$
5: **end while**
Critical
Exit
6: $V.RMW(\{(head + 1, tail) \mid (head, tail)\})$

- We're using the register to simulate a queue of consecutive integers, or just give sequence numbers for each request to enter Critical. Processes get the next number (*tail*), then wait until their number comes up.

Exercise: How large must our register be? Can you reduce it in size?

- Since processes can cycle infinitely, the register will have to hold infinitely large sequence numbers, so our space complexity is infinite.
- There can only be n distinct processes in the queue at any time, though, so really we can have the queue store integers modulo n, in $\log_2 n$ bits.

Exercise: What are some advantages/disadvantages of this algorithm?

- pro: No-lockout–Once a process gets in line, it will eventually get to the front and enter Critical.
- con: Non-uniform: Need to know n to have a register mod$n$.
- con: Busy-waiting is potentially very inefficient. Hard to avoid in general, but it is particularly bad when multiple processes are busy-waiting on the same shared object, as the shared object's performance may degrade for all processes.

Algorithm 3 Code for each p_i to solve Mutual Exclusion, using RMW register without mutual busy-waiting

Shared variables: *Last*–RMW integer register, *Flags*–Array of n binary *Read*/*Write* registers, initially $Flags[0] = 1$, $Flags[1] = 0$.
Entry
1: $place = Last.RMW(Last \to Last + 1 \bmod n)$
2: wait until $Flags[place] == 1$
3: $Flags[place] = 0$
Critical
Exit
4: $Flags[place + 1 \bmod n] = 1$
Remainder

Note that there's still busy-waiting, but only when waiting to enter Critical, when some form of waiting must happen regardless. More importantly, there are at most 2 processes touching any elements of *Flags*, not n.

Space Complexity $\log n$ bits for $Last$, n bits for $Flags$, total: $O(n)$ shared bits. May want to report/-consider $Read/Write$ bits and RMW bits separately, though, as they may be different types of resources.

Correctness

- At most one element of $Flags$ is set to 1 at any time.
- If no element of $Flags$ is 1, then some process is in the critical section.
- If $Flags[k] == 1$, then there are $(k - Last - 1) \bmod n$ processes in Entry, each spinning on a different element of $Flags$.

4 Lower Bound on Number of Memory States

We will shortly move to trying to solve Mutual Exclusion without strong shared objects like RMW and $Test\&Set$. Before we do, briefly consider the limits of optimizing solutions: What is the minimum amount of shared memory required to provide Mutual Exclusion? We'll be quantifying this in terms of the number of unique states available, not the number of bits (which will be roughly $\log_2$ as large). We also need to restrict ourselves to extra-nice Mutex algorithms (stronger form of no-lockout), as proving impossibility in the general case is harder.

Definition 2. A Mutual Exclusion algorithm provides *k-bounded-waiting* if, in any execution, no process enters Critical more than k times while another process is in Entry.

Definition 3. A configuration of a Mutual Exclusion algorithm is *quiescent* if all processes are in Remainder.

Theorem 1. *If an algorithm solves Mutual Exclusion and provides k-bounded waiting for some finite k, then the algorithm uses at least n distinct memory states.*

- No, that's not k memory states. The value of k doesn't affect the required number of states. If we want finite lockout, we need n states.

Proof. We proceed by contradiction. Assume that some algorithm A solves Mutex with k-Bounded Waiting using fewer than n distinct memory states.

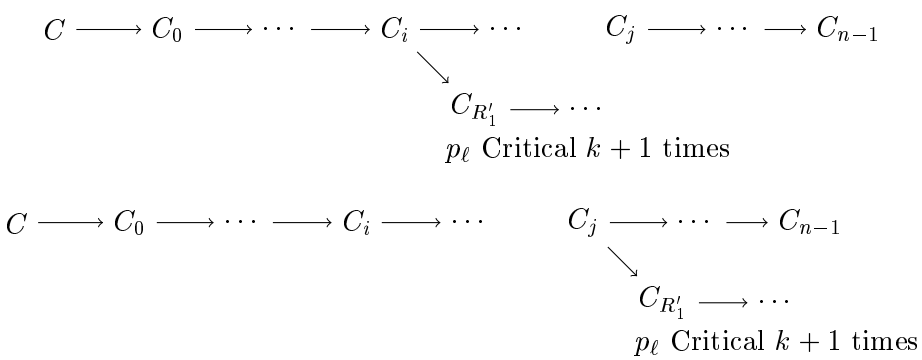
Consider a quiescent configuration C, such as the initial configuration. If we let p_0 run by itself, it must eventually enter the critical section to satisfy no deadlock[1]. Call this schedule t_0 and C_0 the resulting configuration. Then, let p_1 run alone until it begins *Entry*, calling this schedule fragment t_1 and the resulting configuration C_1. Similarly define t_i and C_i for $2 \leq i < n$ by letting each process run alone until it is in Entry, in turn.

$$C \longrightarrow C_0 \longrightarrow C_1 \longrightarrow C_2 \longrightarrow \cdots \longrightarrow C_{n-1}$$

Since there are $< n$ distinct memory states, there exist C_i and C_j s.t. the shared memory is in the same state in both configurations. Assume WLOG that $i < j$. p_0 through p_i take no steps between C_i and C_j, so for $P = \{p_x \mid 0 \leq x \leq i\}$, $C_i \sim_P C_j$. Recall that in both C_i and C_j, p_0 is in Critical and $p_1, \ldots, p_i$ are in Entry.

Construct an execution fragment R' starting from C_i, by allowing processes $p_0, \ldots, p_i$ to run infinitely while all other processes sleep. R' is admissible, so no-deadlock implies that some process $p_\ell \in \{p_0, \ldots, p_i\}$ enters Critical an infinite number of times.

[1] Or terminate, but that's a boring case. For this proof, assume that all processes loop forever, repeatedly trying to enter Critical.



Let R be a finite prefix of R' in which p_ℓ enters Critical $k+1$ times (k from k-BW). Since C_i and C_j are P-similar, we can run R from C_j and p_ℓ will enter Critical $k+1$ times. But p_j is in Entry in C_j, and does not act in R, so in this execution, p_ℓ enters Critical $k+1$ times while p_j is in Entry, violating k-Bounded Waiting. We can extend this to an admissible execution by allowing $p_0, \ldots, p_j$ to run infinitely while $p_{j+1}, \ldots p_{n-1}$ stay in remainder. Thus, there is an admissible execution which violates k-BW, so we must have at least n distinct shared memory states. □

5 Mutual Exclusion Without Strong Primitives

What if we only have *Read/Write* registers without any more powerful (mixed) atomic operations? This is perhaps more realistic, or at least more generally useful, as *RMW* registers and the like are expensive to implement, either in hardware or on top of a message passing system (max message delay or more per *RMW* instance, instead of per *Read-Write* pair). The challenge here is that it is difficult to know the current configuration, as once you *Read* a shared object, it may change before you have a chance to do anything else, including tell others you are going to enter Critical.

> **Exercise:** Try to develop a Mutex algorithm using only *Read/Write* registers. *Hint: Imagine you are at the DMV or similar and have to take a number, then wait until your number is called. How could you implement that algorithmically, and where are the problems?*

Idea When a process begins Entry, it gets a positive integer *related to* the order in which it made its request (like *Fetch&Increment*, but without the mixed operation). When a process exits Critical, the process in Entry with the next lowest number enters.

- Known as the "Bakery" algorithm.

Note that I didn't say the numbers taken are equal to the order in which processes request entry, as that's not trivial. We'll thus still need to break ties. But now we can use IDs: if p_i has $number[i]$ as its sequence counter, use the pair $(number[i], i)$ as its ticket for comparison. The process with lowest ticket, lexicographically, enters Critical.

> **Exercise:** How is this different than just allowing the lowest-ID process to enter first? Every process could begin Entry at the same time and get the same sequence number, so the lowest-ID process would win.

- Yes, lowest-ID goes first, but cannot go *again* until all other processes with the original sequence number have gone.
- This upgrades us to a no-lockout algorithm.

Algorithm 4 Bakery Algorithm: Pseudocode for each p_i.

Shared variables: $number[0..n-1] = [0, \ldots, 0], choosing[0..n-1] = [False, \ldots, False]$
Entry
1: $choosing[i] = True$
2: $number[i] = 1 + max(number[0], \ldots, number[n-1])$
3: $choosing[i] = False$
4: **for** $j = 0$ to $n-1$ (excluding i) **do**
5: wait until $choosing[j] == False$
6: wait until $number[j] == 0$ or $(number[j], j) > (number[i], i)$
7: **end for**
Critical
Exit
8: $number[i] = 0$
Remainder

Complexity (Space)

- *number*: n integers
- *choosing*: n Booleans
- Total: Unbounded! Can't wrap around as we can't guarantee some process isn't stuck between the *Read*s and the $Write$ on line 2, about to set its *number* to 1.

Correctness

- Mutual Exclusion: A process can only be in Critical if its ticket pair is smaller than any other complete ticket pair. Thus, if there were more than one, they'd all be smaller than each other.
- No-Lockout (implies No-Deadlock): Let p_i be the process with the smallest $(number[i], i)$ pair which starves (stays in Entry forever). (That is, it has completed line 2. It must form a complete ticket, as that doesn't require waiting for any other process.) Every process which starts Entry after p_i completes line 2 will get a *number* larger than p_i's, so will not enter Critical before p_i. Since no process with a smaller *number* starves, they will all enter and exit Critical. Similarly, any p_j with $number[j] == number[i]$ but $j < i$ will complete Critical, then p_i will enter Critical, contradicting our assumption that it starves.

6 Dining Philosophers

There are, of course, many different variations of resource-allocation problems, and we can't cover all of them in this course. We will briefly consider one of the most famous such problems, though. This is a specific type of general resource allocation, where there may be many different limited resources which processes want to access and different processes' critical sections may need access to different resources.

Problem 2. A group of philosophers are seated around a circular table. They spend most of their time thinking, but occasionally decide to eat. To eat, they need two forks, one in each hand. There is exactly one fork between each pair of adjacent philosophers. Thus, for a philosopher to eat, they must be able to pick up the forks on either side, and when they are done eating, they put the forks down. A hungry philosopher must be allowed to eat eventually, or they will flip the table in anger.

The forks here (or chopsticks might make more sense) represent shared resources shared by neighboring processes in a ring. To complete its critical section, a process needs both of the adjacent resources, but doesn't care about resources on the other side of the ring.

Exercise: Try to come up with an algorithm to solve the Dining Philosophers problem. What tends to happen?

6.1 Impossibility of Symmetric Algorithms

It turns out that no algorithm can solve this problem if all processes act the same way (and all shared resources start in the same state). The idea is the same as the proof that there is no anonymous Leader Election algorithm in a ring.

Theorem 2. *There is no symmetric algorithm to solve the Dining Philosophers problem without deadlock.*

Proof. Assume in contradiction that there is a symmetric algorithm solving the problem. Let all processes run at the same rate. They will thus take the same steps, and all begin Entry at the same time. Since some process p_i must make progress (be first to enter Critical), p_i will acquire one of the adjacent resources (pick up a fork). Assume, WLOG, that p_i acquires the resource to its left. Then the process to p_i's right, p_{i-1}, will simultaneously acquire the resource to p_i's right. It is then impossible for p_i to enter Critical until p_{i-1} releases the resource it holds. But, since the algorithm is symmetric, p_i will also release the resource to its left. Induction shows that it is impossible for p_i to acquire both adjacent resources simultaneously, contradicting the assumption that it is the first to enter Critical. □

6.2 Right-Left Algorithm

As we just showed, there must be some initial symmetry-breaking to allow an algorithm to succeed. Suppose that we can have different code for odd- and even-position processes. That is, alternate processes run one of two different programs. For ease of presentation, assume n is even, though the solution can be extended to work for odd n, as well. This code actually works with less balanced distributions of processes running each version of the algorithm, but performance may suffer. One note is that this algorithm uses augmented queues (*Enqueue*, *Dequeue*, and *Peek*), which are strong primitives.

Idea For each shared resource, create an augmented queue shared between the two processes which may want to access that resource. When processes want to use a resource, they put their ID in the queue. They may then enter Critical when their ID is at the top of both adjacent queues, and remove their ID from both queues when exiting.

Exercise: Write pseudocode to implement this algorithm. (Call shared resources $r_0, \ldots, r_{n-1}$ and shared queues $q_0, \ldots, q_{n-1}$.) Argue that it satisfies both Mutual Exclusion for each resource and no-deadlock.

Algorithm 5 Code for odd-indexed processes (code for even-indexed processes is symmetric)

```
    Entry
1:  q_{i-1}.Enqueue(i)                          ▷ index arithmetic mod n
2:  wait until q_{i-1}.Peek() == i
3:  q_i.Enqueue(i)                              ▷ index arithmetic mod n
4:  wait until q_i.Peek() == i
    Critical
    Exit
5:  q_{i-1}.Dequeue()
6:  q_i.Dequeue()
    Remainder
```

Theorem 3. *The max time, T, from when any p_i begins Enter until it begins Critical is at most $3c + 18\ell$, where c is the max time a process spends in Critical and ℓ is the max time between steps at a single process.*

Proof. Define S as the max time any process takes from being first on one of the adjacent queues to when it enters Critical. Note that $S < T$. When p_i begins Entry, it immediately (in time ℓ, for one computation step) places its ID on one adjacent queue. If that queue is empty, then $T = \ell + S$. If not, then p_i waits for p_{i-1} to release the resource, which takes at most $S + c + \ell$ time. This bound comes from the fact that every shared resource is either the first both of its neighbors seek or the second for both. Thus, p_i waits for p_{i-1} to enter Critical after getting its first resource, finish Critical, and *Dequeue* itself. In either case, it takes at most S more time for p_i to enter Critical. We then have

$$T \leq c + 2\ell + 2S$$

We must now consider how large S can be. It may take ℓ time for p_i to discover it is first on the queue for its first resource, then another ℓ to *Enqueue* its ID on the second. At this point, the worst case is that it must wait for the second resource. But, since this is the second resource for both adjacent processes, if it is already taken, then the adjacent process has both necessary resources and will execute Critical, taking at most $2\ell + c + 2\ell$ time to realize it has both, complete Critical, and exit. Within ℓ time after that, p_i will discover it has its second resource, and another ℓ to enter Critical. Thus,

$$S \leq 2\ell + (4\ell + c) + 2\ell = c + 8\ell$$

Combining our bounds gives

$$T \leq c + 2\ell + 2(c + 8\ell) = 3c + 18\ell$$

□

This is interesting because the time for a process to enter Critical is not dependent on n in any way. Broadly, this may be unintuitive, since more processes typically means more contention and potential for delay. In the context of a ring, though, where processes only need local resources, it makes sense that increasing the size of the ring may not have much effect. Counterpoint to that, though, is the fact that if processes are not alternating in priority, we can have dependency chains as large as the ring, where a process is waiting on that to its left, which is waiting on that to its left, and so on.