Lecture 4: Operating Systems - Process Synchronization

Yonghao Lee

April 22, 2025

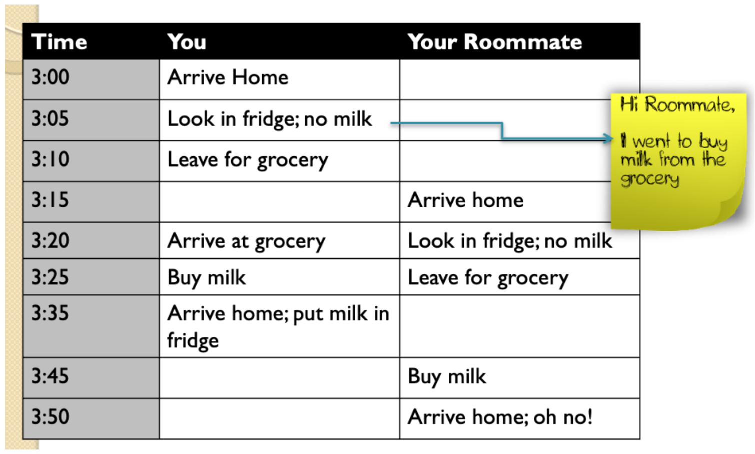
Introduction

We give some motivation first for learning synchronization. This topic is not about the OS itself, but rather about what happens between processes in the OS when they access shared resources.

Real-World Analogy: The Roommate Milk Problem

Process synchronization issues can be understood through a common real-world scenario:

- Two roommates share a refrigerator (shared resource)
- Both check the fridge, see no milk, and independently decide to buy milk
- Neither roommate knows about the other's decision to buy milk
- Both end up buying milk, resulting in duplicate purchases



Time	You	Your Roommate
3:00	Arrive Home	
3:05	Look in fridge; no milk	
3:10	Leave for grocery	
3:15		Arrive home
3:20	Arrive at grocery	Look in fridge; no milk
3:25	Buy milk	Leave for grocery
3:35	Arrive home; put milk in fridge	
3:45		Buy milk
3:50		Arrive home; oh no!

Hi Roommate,

I went to buy milk from the grocery

Figure 1: The Roommate Milk Problem Timeline

The timeline shows:

1. You arrive home at 3:00 PM and check the fridge at 3:05 PM
2. You leave for the grocery store at 3:10 PM
3. Your roommate arrives home at 3:15 PM
4. You arrive at the grocery store at 3:20 PM
5. Your roommate also discovers there's no milk at 3:20 PM
6. You buy milk at 3:25 PM
7. Your roommate leaves for grocery shopping at 3:25 PM
8. You return home and put milk in the fridge at 3:35 PM
9. Your roommate buys milk at 3:45 PM
10. Your roommate returns home at 3:50 PM to discover there's already milk

This real-world problem illustrates the concept of race conditions, where the final outcome depends on the timing of operations rather than program logic.

Two Printing Processes

In computing, we encounter similar issues. Consider the following example:

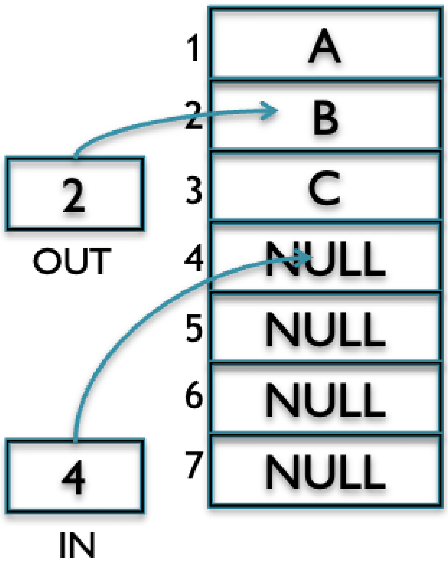


Figure 2: Spooler structure

Imagine that we have two processes/threads that want to print, with a single resource - a printer. We would like both processes to be able to print. To manage this, there is a spooler, where we write the jobs to. It is a shared queue from which the printer extracts print jobs. In this spooler system:

- **OUT** is the pointer to the next job to print (position 2 in the figure)
- **IN** is where we write new jobs (end of queue, position 4 in the figure)
- **NULL** indicates that there is no job to print
- Jobs are stored in slots (A, B, C in positions 1-3)

Code for Adding Jobs and Race Conditions

The code for adding jobs to the spooler is:

```
Spooler[IN] = job
IN++
```

The critical issue is that both the **IN** pointer and the **Spooler** array are shared among all processes. This creates potential race conditions when multiple processes attempt to add jobs simultaneously.

Let's look at 2 possible outcomes:

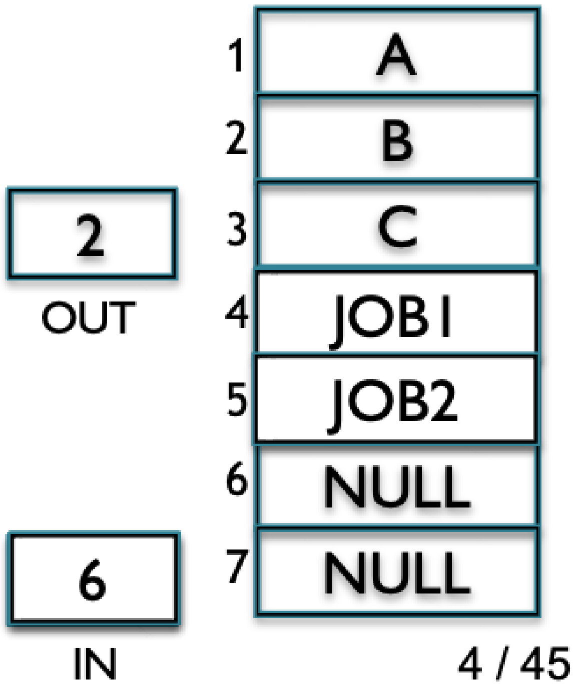


Figure 3: Correct Execution Sequence

1. Process 1 writes JOB1 to Spooler[IN] (position 4)
2. Process 1 increments IN to position 5
3. Context switch occurs

4. Process 2 writes JOB2 to Spooler[IN] (now position 5)
5. Process 2 increments IN to position 6
6. Result: Both JOB1 and JOB2 are correctly stored in the queue at positions 4 and 5

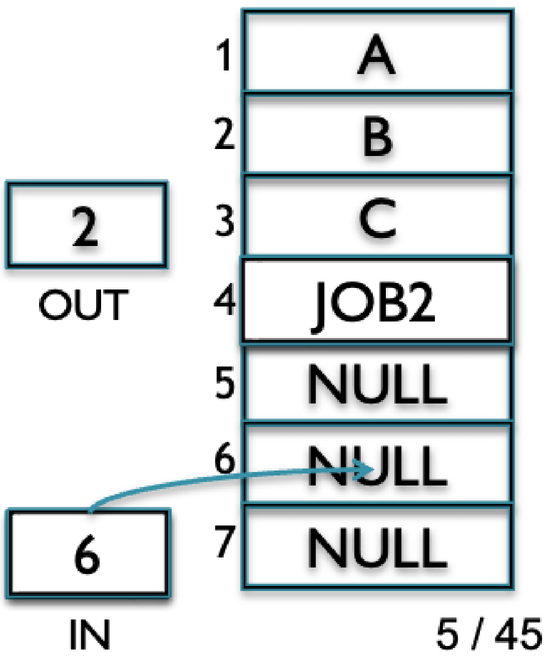


Figure 4: Incorrect Execution with Job Overwriting

1. Process 1 writes JOB1 to Spooler[IN] (position 4)
2. Context switch occurs before IN is incremented
3. Process 2 writes JOB2 to Spooler[IN] (still position 4), overwriting JOB1
4. Context switch occurs
5. Process 1 increments IN to position 5
6. Context switch occurs
7. Process 2 increments IN to position 6
8. Result: JOB1 is lost; a gap exists in the queue between JOB2 at position 4 and the next insertion point at position 6

Fundamental Synchronization Concepts

Race Conditions in Multithreaded Environments

Shared Resource

A global variable named `IN` is stored at memory address 100. This variable is accessible by multiple threads running in parallel.

Thread Operation Sequence

Each thread executes the following assembly instructions:

```
lw r1, 100    // Load value from address 100 into register r1
inc r1        // Increment the value in register r1
sw 100, r1    // Store the value from r1 back to address 100
```

Expected vs. Actual Behavior

- **Expected behavior:** If two threads each increment the variable once, the final value should be increased by 2.
- **Actual behavior:** Due to the race condition, the final value is only incremented by 1.

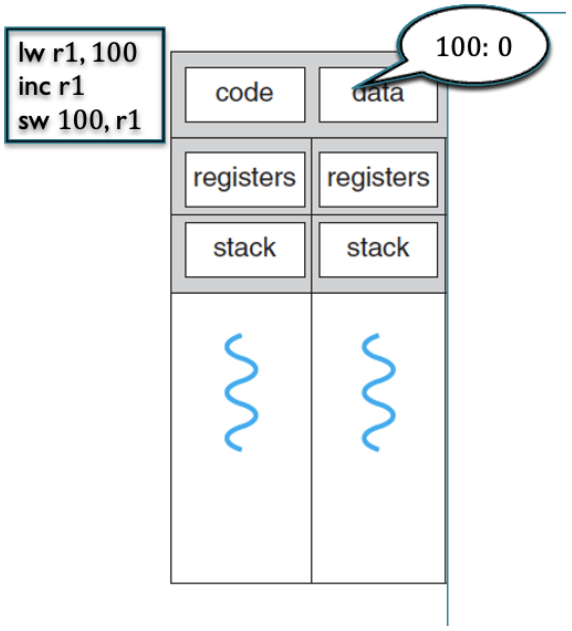


Figure 5: IN = 0 initially

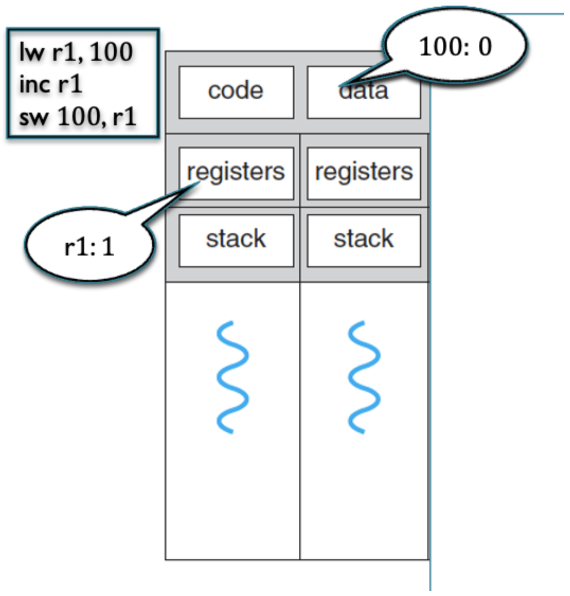


Figure 6: Context switch before thread 1 could write

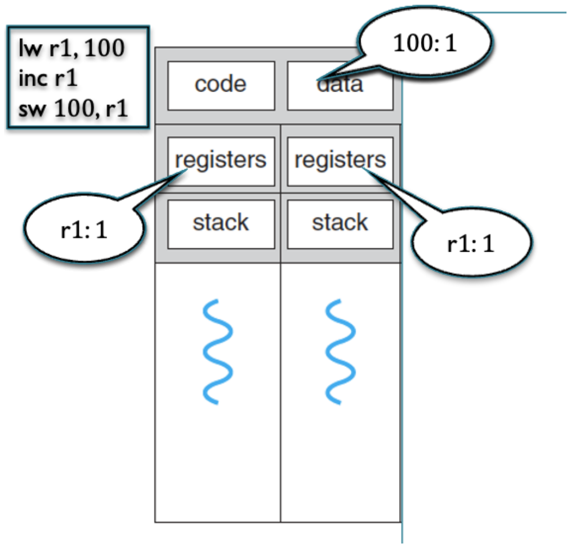


Figure 7: Thread 2 manages to write and switches back to thread 1

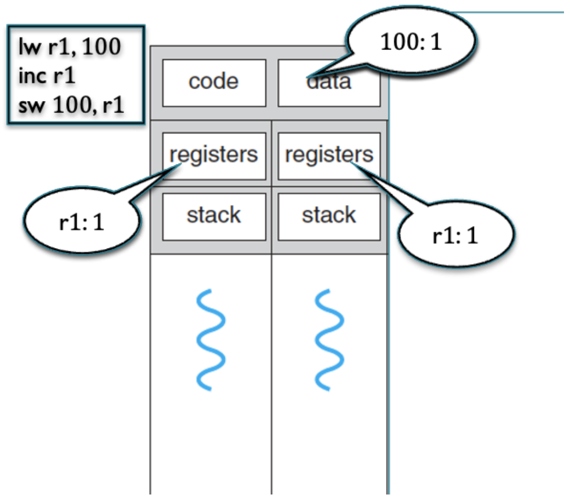


Figure 8: Thread 1 writes 1, overwriting thread 2's update

Detailed Execution Trace

1. Initial state: `IN = 0` (stored at address 100)
2. Thread 1 executes `lw r1, 100` (r1 in Thread 1 now contains 0)
3. Thread 1 executes `inc r1` (r1 in Thread 1 now contains 1)
4. *Context switch occurs to Thread 2*

5. Thread 2 executes `lw r1, 100` (r1 in Thread 2 now contains 0, since Thread 1 hasn't written back yet)
6. Thread 2 executes `inc r1` (r1 in Thread 2 now contains 1)
7. Thread 2 executes `sw 100, r1` (address 100 now contains 1)
8. *Context switch back to Thread 1*
9. Thread 1 executes `sw 100, r1` (address 100 still contains 1, overwriting the previous value)
10. Final state: `IN = 1` (instead of the expected value 2)

The Core Issue

This race condition occurs because the operation is not atomic. Each thread performs a read-modify-write sequence that can be interrupted. When Thread 2 reads the value, it gets the original value before Thread 1's increment is committed to memory. This results in Thread 1's increment being effectively lost.

Key Concepts

- **Race Condition:** The scheduling of processes/threads changes the final result
  - Different order → different results
  - Typical scenario: Most of the time everything is OK, but sometimes not → very hard to debug
- **Atomic Instruction:** Instruction that completes in a single step relative to other threads
  - $x = x + 1$ was not atomic
  - Read (lw) and write (sw) were atomic
- **Busy-waiting (spinning, busy-looping):** Technique in which a process repeatedly checks to see if a condition is true
  - "do nothing" loops: while(flag[$1 - i$] and turn==$1 - i$);
  - Usually, should be avoided as it wastes CPU cycles and yields large overhead

Critical Sections

In summary, the fundamental issue is uncoordinated access to shared resources (such as data structures, devices, or memory locations) by multiple threads, processes, or processors/cores. This creates race conditions where the final state depends on the exact timing of operations.

The portion of code that accesses these shared resources is called the “critical section.” A critical section can range from a single instruction to multiple lines of code. The key requirement is that critical sections accessing the same shared resource must not be concurrently executed by more than one process/thread.

To ensure correctness, we need synchronization mechanisms that provide **Mutual exclusion** - only one thread/process can execute in the critical section at a time.

Without proper synchronization, we end up with race conditions like those demonstrated in both the technical (IN variable) and real-world (roommate milk) examples, where the final outcome is inconsistent and depends on execution timing rather than program logic.

The solution to this problem is by using a mutual exclusion algorithm, which avoids the simultaneous execution of a critical section.

But sometimes mutual exclusion is not enough alone, imagine that the order of execution matters: we need thread 1 to execute in the critical section before thread 2, we are going to talk about it next week.

Dijkstra’s Model and Requirements

Dijkstra’s Model

Edsger Dijkstra suggested modeling the mutual exclusion problem with four distinct sections of code:

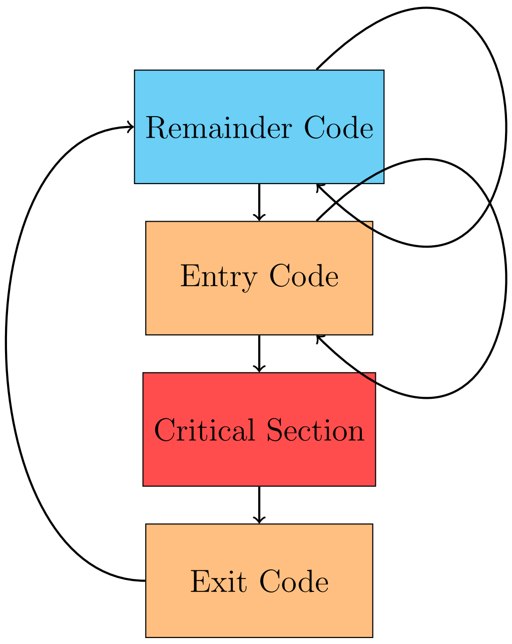


Figure 9: Dijkstra's model for the mutual exclusion problem

Each process repeatedly executes these sections in order:

1. **Remainder Code**: The non-critical section where the process performs operations that don't involve shared resources.

2. **Entry Code**: Code that requests access to the critical section, implementing mutual exclusion mechanisms.

3. **Critical Section**: The portion of code that accesses shared resources and must be executed mutually exclusively.

4. **Exit Code**: Code that releases access to the critical section, allowing other processes to enter.

The challenge is to design entry and exit code algorithms that guarantee mutual exclusion while avoiding deadlocks and ensuring fairness in access to the critical section.

Success Criteria

For the implementation of entry code and exit code to be considered good, the following criteria should be met:

1. **Mutual Exclusion:** No two processes are in the critical section simultaneously.

2. **Progress:** If a process is trying to enter the critical section, then some process eventually enters the critical section.
3. **Starvation Freedom:** A process trying to enter the critical section eventually succeeds.
4. **Generality:** No assumptions about speeds or number of participants. The algorithm must cope with any possible number of threads and be independent of their running times.
5. **No Blocking in the Remainder:** No process running outside its critical section may block another process.

Note that criterion 3 (Starvation Freedom) implies criterion 2 (Progress), but not vice versa. Progress only guarantees that *some* process will enter the critical section, while Starvation Freedom ensures that *every* requesting process will eventually enter.

Solution Approaches

Simple Approaches

Disabling Interrupts

A simple approach to implementing critical sections is by disabling interrupts:

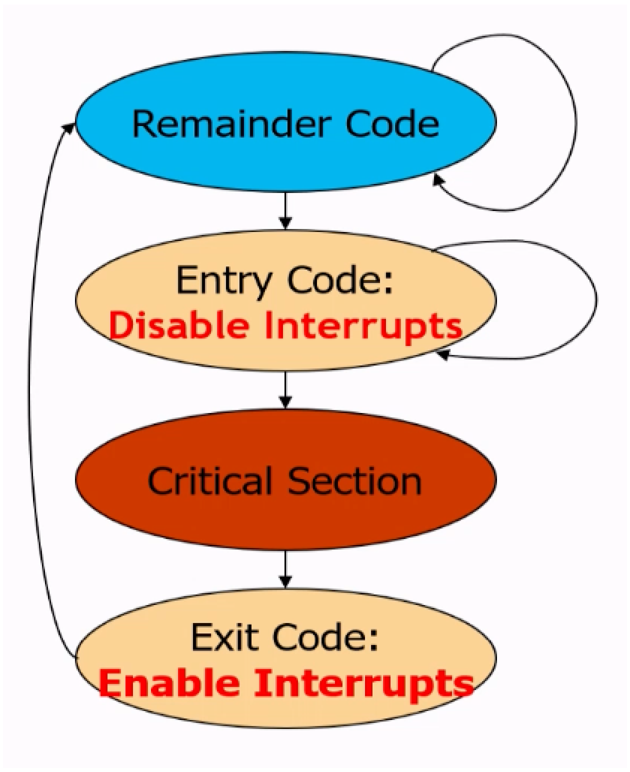


Figure 10: Critical Section Implementation Using Interrupt Disabling

This method attempts to solve synchronization issues that arise from context switches by preventing them entirely through interrupt disabling. When a process enters its critical section, it disables interrupts, ensuring it cannot be preempted until it completes the critical section and re-enables interrupts.

While this approach satisfies properties 1, 2, 3, and 5 of Dijkstra's criteria, it fails to meet property 4 (Generality) for several important reasons:

- **Multi-processor systems:** In a computer with multiple physical processors, disabling interrupts on one processor has no effect on others. If Process 1 runs on Processor 1 and Process 2 runs on Processor 2, both could disable interrupts on their respective processors and enter critical sections simultaneously, violating mutual exclusion.
- **System reliability concerns:** Disabling interrupts turns off essential operating system mechanisms. This could prevent the system from handling urgent hardware issues, potentially leading to system failures.
- **Privilege issues:** User processes should not be allowed to disable interrupts, as this is a privileged operation that could compromise system integrity.
- **Time constraints:** Even when appropriate, interrupts should only be disabled for very short periods to maintain system responsiveness.

Therefore, while conceptually simple, this approach is unsuitable for modern computing environments where real parallelism exists.

Software-Based Solutions

First Try

We try our second implementation without doing tricks like disabling interrupts.

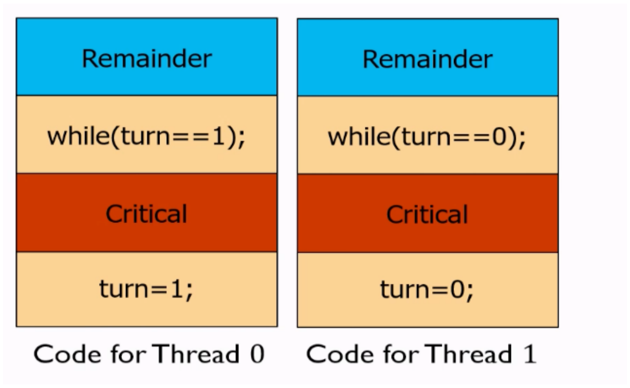


Figure 11: First Implementation Without Cheating

This involves busy waiting (the waiting process continuously checks the condition in a loop without doing any useful work), and it is for 2 threads/processes. It uses a global variable named turn that cannot be used in the remainder or the critical section.

Analysis of the First Try Let's evaluate this implementation against Dijkstra's criteria:

Mutual Exclusion: This property is satisfied. The proof by contradiction is as follows:

Proof. Assume both processes are in their critical sections simultaneously.

Without loss of generality, assume Thread 0 entered its critical section first. This means Thread 0 passed its entry code (`while(turn==1)`).

Therefore, `turn` must equal 0 when Thread 0 entered its critical section.

The value of `turn` can only be changed by a process in its exit section.

Thread 0 only changes `turn` to 1 after completing its critical section.

For Thread 1 to be in its critical section, it must have passed `while(turn==0)`.

This means `turn` must equal 1, which contradicts our earlier deduction. □

Note that the first try implementation violates the properties of progress, starvation freedom, and no blocking in the remainder. Consider this scenario: if the variable `turn` is initialized to 0 and thread 1 attempts to enter its critical section, it will be stuck in its entry code (`while(turn == 0)`) indefinitely. Meanwhile, if thread 0 remains in its remainder section with no intention of entering its critical section, then no thread will ever progress. This clearly violates the progress property. Moreover, thread 0 effectively blocks thread 1 while staying in its remainder section, violating the no blocking in the remainder property. Starvation freedom is consequently violated as well, since thread 1 can never enter its critical section despite attempting to do so.

Second Try

We improve the first try by introducing an array called flag.

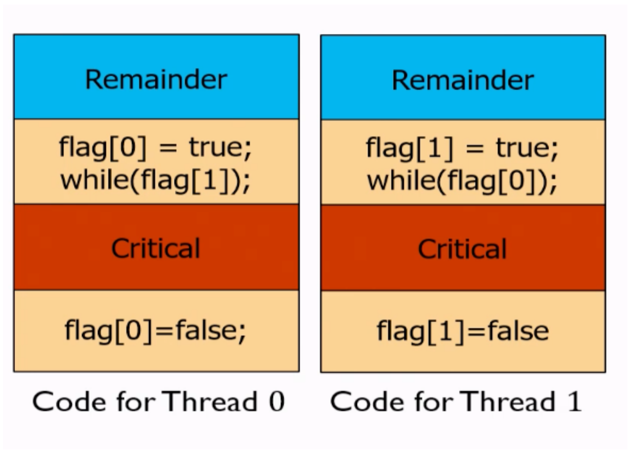


Figure 12: Second Implementation Using Flag Array

This implementation uses a boolean array `flag[]` to indicate which thread intends to enter its critical section.

Analysis of the Second Try Let's evaluate this implementation against Dijkstra's criteria:

1. **Mutual Exclusion:** This property is satisfied. The proof by contradiction is as follows:

   Proof. Assume both processes are in their critical sections simultaneously.

   Without loss of generality, assume Thread 0 entered its critical section first.

   For Thread 0 to enter its critical section, it must have passed its entry code, which means:

   - Thread 0 set `flag[0] = true` to indicate its intention
   - Thread 0 passed `while(flag[1])`, meaning `flag[1] = false` at that moment

   Similarly, for Thread 1 to be in its critical section:

   - Thread 1 set `flag[1] = true`
   - Thread 1 passed `while(flag[0])`, meaning `flag[0] = false` at that moment

   But Thread 0 only sets `flag[0] = false` in its exit section, after completing its critical section.

   This creates a contradiction: Thread 1 can only enter if `flag[0] = false`, but Thread 0 only sets `flag[0] = false` after exiting its critical section. □

2. **No Blocking in the Remainder:** This property is satisfied. A thread can block another thread only if its flag is true, which implies it is either in its entry, critical, or exit section. When a thread is in its remainder section, its flag is set to false, and it cannot block other threads.

3. **Progress:** This property does not hold. The algorithm allows for deadlock situations.

4. **Starvation Freedom:** Since progress is not guaranteed, starvation freedom also does not hold.

5. **Generality:** The algorithm works for exactly two threads and does not easily extend to more.

Deadlock in the Second Try A critical flaw in this implementation is the possibility of deadlock:

The deadlock scenario unfolds as follows:

1. Thread 0 executes `flag[0] = true` in its entry section

2. Context switch occurs before Thread 0 checks `flag[1]`

3. Thread 1 executes `flag[1] = true` in its entry section

4. Thread 1 begins waiting in its `while(flag[0])` loop because `flag[0] = true`
5. Context switch back to Thread 0
6. Thread 0 begins waiting in its `while(flag[1])` loop because `flag[1] = true`
7. Result: Both threads are now waiting for each other, creating a deadlock situation

This deadlock is not a result of scheduler failures—both threads have ample opportunity to run. Rather, it stems from the fundamental design of the algorithm, where each thread can set its flag and then find itself permanently blocked by the other thread's flag.

Third Try

We improve upon the second try by changing the while loop condition and introducing a turn variable:

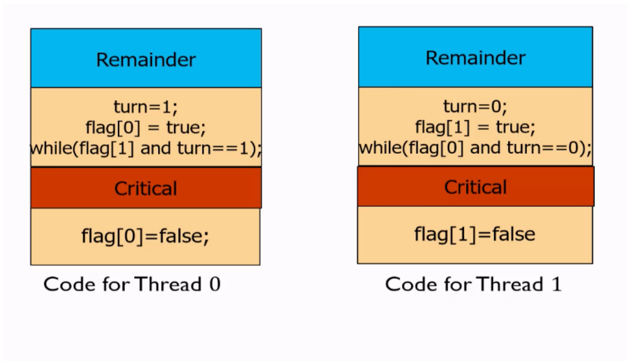


Figure 13: Third Implementation with Turn Variable

Analysis of the Third Try This implementation attempts to prevent deadlock by using `turn` as a tie-breaker, but unfortunately, it fails to guarantee mutual exclusion:

Consider the following execution sequence:

1. Thread 0 sets `turn = 1` (giving priority to Thread 1)
2. Context switch occurs before Thread 0 sets its flag
3. Thread 1 sets `turn = 0` (giving priority to Thread 0)
4. Thread 1 sets `flag[1] = true`
5. Thread 1 checks while condition: `flag[0]` is false, so it proceeds regardless of `turn`
6. Thread 1 enters its critical section
7. Context switch to Thread 0
8. Thread 0 sets `flag[0] = true`

9. Thread 0 checks while condition: `flag[1]` is true, but `turn = 0`, so it also proceeds
10. Thread 0 enters its critical section while Thread 1 is still there
11. Result: Mutual exclusion is violated

This third try fails because the order of operations in the entry code is incorrect. By setting `turn` before setting the flag, a thread can lose its "reservation" if a context switch occurs at the wrong time.

Peterson's Algorithm - A Correct Solution

Peterson's algorithm solves the problem by simply swapping the first two lines of the entry code in our third try:

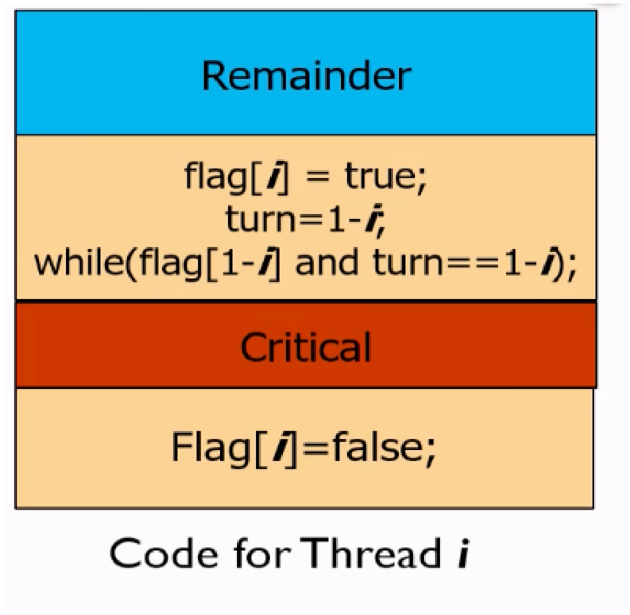


Figure 14: Peterson's Algorithm for Mutual Exclusion

Analysis of Peterson's Algorithm Peterson's algorithm guarantees mutual exclusion. The proof by contradiction is as follows:

Proof. Assume that the two processes are in the critical section simultaneously.

Without loss of generality, assume Thread 0 left its entry section (and entered the critical section) first.

At that point, either `flag[1] = false` or `turn = 0` (or both), since Thread 0 passed its while condition.

Case 1: If `flag[1] = false`, then Thread 1 has not executed Line 1 of its entry code yet. When it eventually executes Line 2, it will set `turn` to 0 and wait. The only place `turn` is set to 1 is in Thread 0's entry code, but Thread 0 has already left the entry section. This creates a contradiction.

Case 2: If `turn = 0`, then Thread 1 must have executed Line 2 between Thread 0's Line 2 and 3. This means Thread 1 set `turn = 0` after Thread 0 set `turn = 1`. However, this would mean Thread 0 would be waiting in its while loop since both `flag[1] = true` and `turn = 0`. This contradicts our assumption that Thread 0 entered its critical section.

Therefore, our assumption is false and mutual exclusion is guaranteed. □

Let's evaluate Peterson's algorithm against all of Dijkstra's criteria:

1. **Mutual Exclusion:** As proven above, this property is satisfied.

2. **Progress:** If both threads want to enter the critical section, the `turn` variable ensures that one of them will.

3. **No Blocking in the Remainder:** This property is satisfied. A thread can only block another thread when its flag is set to true, which only happens when it is trying to enter, is in, or is exiting its critical section. When a thread is in its remainder section, its flag is false and cannot block other threads.

4. **Starvation Freedom:** This property is satisfied. The proof is as follows:

   Proof. Assume Thread 1 tries to get into the critical section but is blocked (on Line 3) for an infinite time. This means Thread 0 must enter the critical section an infinite number of times while Thread 1 is waiting.

   The second time Thread 0 executes its entry code, it will set `flag[0] = true` and `turn = 1`, and it will then block on its while condition because Thread 1 has `flag[1] = true`.

   The next time Thread 1 gets CPU time, the while condition (`flag[0] && turn == 0`) will not hold because `turn = 1`, allowing Thread 1 to enter its critical section.

   This contradicts our assumption that Thread 1 is blocked indefinitely. □

5. **Generality:** The algorithm works for exactly two threads with any execution speeds. However, it cannot be easily extended to more than two threads. More complex algorithms like the Bakery Algorithm (Lamport's Algorithm) are needed for multi-thread scenarios.

Despite its elegance, Peterson's algorithm has several practical limitations:

- **Two threads only:** The algorithm works specifically for two threads and cannot be easily extended to handle more threads.

- **Busy waiting:** The algorithm employs busy waiting (spinning), where a thread continuously checks a condition in a loop. This wastes CPU cycles that could be used for other tasks, making it inefficient in high-contention scenarios.

- **Compiler optimizations:** Modern compilers and processors may reorder instructions for performance optimization, potentially breaking the algorithm's correctness. Memory barriers or volatile declarations are needed to prevent such reordering.

Lamport's Bakery Algorithm

Lamport's bakery algorithm is one of many mutual exclusion algorithms designed to prevent concurrent threads entering critical sections of code concurrently to eliminate the risk of data corruption.

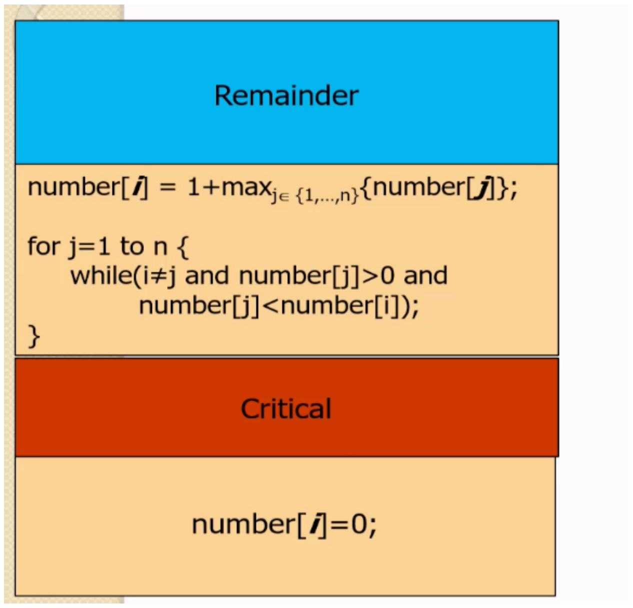
We can think of it as:

- Take a number: before trying to enter, each process announces its intent and picks a ticket one higher than the current maximum.
- Wait your turn: it then waits until no other process has a smaller ticket (or the same ticket but a lower ID), ensuring that only one "customer" (process) is served at a time and in roughly first-come-first-served order.

But it hinges on the correct implementation. We present several implementations:

First Try

```
function lock($i$):
    number[$i$] ← 1 + $\max_{j\in\{1,\dots,n\}}$(number[$j$]);
    for $j = 1$ to $n$ do
        while $i \neq j$ and $number[j] > 0$ and $number[j] < number[i]$ do
        end
    end
function unlock($i$):
    number[$i$] ← 0;
```



Remainder
number[i] = 1+max_{j∈{1,...,n}}{number[j]};
for j=1 to n {
while(i≠j and number[j]>0 and
number[j]<number[i]);
}
Critical
number[i]=0;

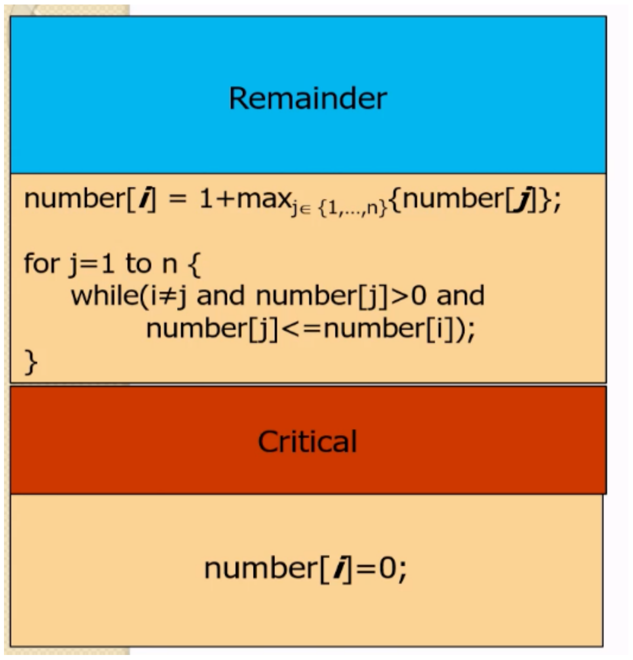
Figure 15: First Try of Bakery Algorithm

This implementation fails to ensure mutual exclusion because:

1. **Race condition in ticket assignment:** Multiple processes can read the maximum ticket value concurrently and assign themselves consecutive tickets.

2. **Incomplete waiting condition:** The waiting loop only checks for processes with smaller ticket numbers (`number`[j] ¡ `number`[i]), but does not address the case when two processes have the same ticket number.

3. **No tie-breaking mechanism:** When two processes i and k get the same ticket number (`number`[i] = `number`[k]), both will pass their waiting loops with respect to each other since neither ticket is smaller than the other.



```
number[i] = 1+max_{j∈{1,...,n}}{number[j]};

for j=1 to n {
    while(i≠j and number[j]>0 and
          number[j]<=number[i]);
}
```

```
number[i]=0;
```

Figure 16: Second Try of Bakery Algorithm

Second Try The second try uses less than or equal in the loop, but this obviously creates a deadlock.

Third Try Instead of using only the numbers, we use a lexicographic order as part of the condition of the while loop:

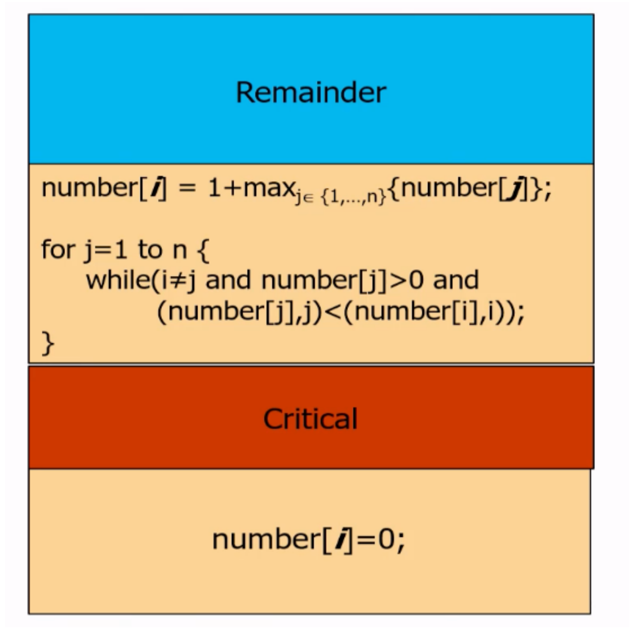

Remainder
number[i] = 1+max_{j∈{1,...,n}}{number[j]};
for j=1 to n {
while(i≠j and number[j]>0 and
(number[j],j)<(number[i],i));
}
Critical
number[i]=0;

Figure 17: Third Try with Lexicographic Ordering

In the standard Bakery Algorithm with lexicographic ordering, when two threads acquire the same ticket number, the thread with the lower ID should enter the critical section first. However, the third try contains a subtle race condition that undermines this property.

The Race Condition Consider the following execution sequence:

1. Threads 4 and 5 begin execution concurrently.
2. Thread 4 computes its ticket number as $\text{ticket}_4 = 1+\max\{\text{number}[j] : j \in \{1, \ldots, n\}\} = 1 + 0 = 1$.
3. Before Thread 4 stores this value to memory (i.e., completes $\text{number}[4] \leftarrow 1$), Thread 5 begins computing its own ticket.
4. Since Thread 4 has not yet written its ticket to shared memory, Thread 5 observes the same maximum value and computes $\text{ticket}_5 = 1 + 0 = 1$.
5. Both threads now possess identical ticket numbers.
6. In the waiting phase, Thread 5 examines other threads:
   - For Thread 4, it evaluates: $\text{number}[4] > 0$ and $(\text{number}[4], 4) < (\text{number}[5], 5)$
   - Since Thread 4 either has not written its value yet or has the same ticket value, and $4 < 5$, the condition may not hold.
7. Thread 5 enters the critical section without waiting for Thread 4.
8. Thread 4 also enters the critical section since it has a lower ID.
9. Both threads are now simultaneously in the critical section, violating mutual exclusion.

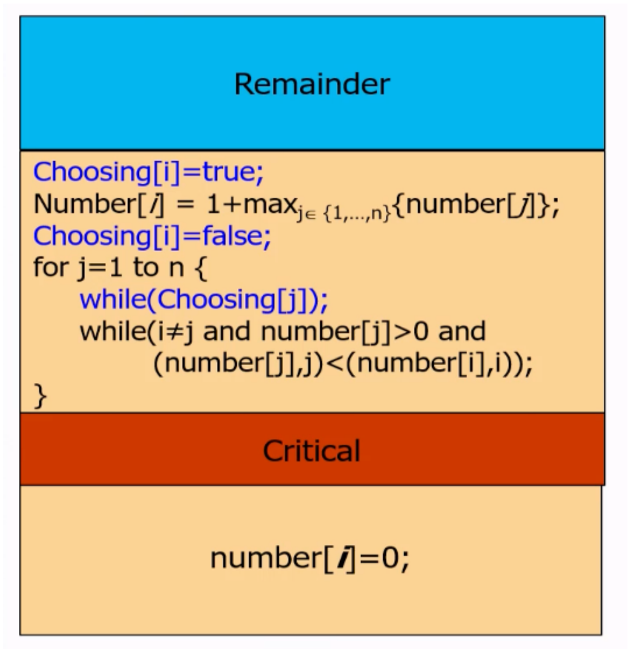
The Fundamental Issue The problem stems from the non-atomic nature of the ticket acquisition process. There exists a temporal window between:

1. Reading the current maximum ticket value
2. Computing a new ticket value
3. Storing the computed value to shared memory

During this window, other threads may read an inconsistent state of the system, leading to duplicate ticket assignments. Even with lexicographic ordering to break ties, the waiting condition does not account for threads that are in the process of selecting their tickets.

Correct Implementation The complete implementation addresses the race condition by:

1. Using a `choosing` array (highlighted in blue) to signal when a thread is selecting a ticket
2. Making threads wait for others that are in the process of choosing (`while(choosing[j])`)
3. Using lexicographic ordering to break ties for threads with the same ticket number
4. Setting the ticket to 0 when exiting the critical section



```
Choosing[i]=true;
Number[i] = 1+max_{j∈{1,…,n}}{number[j]};
Choosing[i]=false;
for j=1 to n {
    while(Choosing[j]);
    while(i≠j and number[j]>0 and
          (number[j],j)<(number[i],i));
}
```

```
number[i]=0;
```

Figure 18: Correct Implementation of Bakery Algorithm

This implementation also ensures a notion of FIFO (First-In-First-Out) ordering.

Hardware-Based Solutions

Read-Modify-Write Instructions

In concurrent programming, proper synchronization is essential to avoid race conditions. A **race condition** occurs when multiple threads or processes access and manipulate shared data simultaneously, and the final outcome depends on the relative timing or ordering of their execution. These conditions lead to unpredictable behavior and bugs that are difficult to reproduce.

We assumed so far that individual read (load) and write (store) operations are atomic instructions. However, problems arise when we have a sequence of operations (read-modify-write) that need to be executed together. As we've seen, there might be a context switch between a read and its corresponding write, which can lead to race conditions.

Modern CPUs solve this problem by supporting hardware-level atomic RMW instructions that execute an entire read-modify-write sequence as a single, uninterruptible operation. Three key examples are:

- **Test&Set(&lock):** This atomic instruction performs the following operations as a single unit:
  {i=*lock; *lock=1; return i;}

  Here, *lock* refers to a memory address containing a lock variable (typically 0 for unlocked, 1 for locked). This instruction atomically reads the current value of the lock, sets it to 1 (locked), and returns the original value. If the original value was 0, the thread successfully acquired the lock. This prevents the race condition where two threads both see the lock as free and both try to acquire it simultaneously.

- **Fetch&Add(&p,inc):** This atomic instruction performs:
  {val=*p; *p=val+inc; return val}

  Here, *p* refers to a memory address containing a value that needs to be incremented. The instruction atomically reads the value at address *p*, adds *inc* to it, stores the result back, and returns the original value. This prevents the race condition in counters where two threads might read the same value, increment it separately, and both write back the same incremented value, effectively losing one update.

- **Compare&Swap(&p,old,new):** This atomic instruction performs:
  {if(*p≠old) return false; *p=new; return true;}

  Here, *p* is a memory address, *old* is the expected current value, and *new* is the value to be written. This instruction checks if the current value at *p* matches the expected *old* value. If it does, it updates it to *new* and returns true; otherwise, it leaves the memory unchanged and returns false. This prevents race conditions by ensuring the memory hasn't been modified by another thread between when this thread read the value and when it attempts to update it.

These hardware-supported atomic instructions guarantee that no other thread can observe or modify the memory location between the read and write portions of the operation. By using these primitives, we can build higher-level synchronization mechanisms that protect critical sections and ensure proper coordination between threads.

For example, without atomic RMW operations, a scenario like incrementing a shared counter could fail:

- Thread 1: Reads counter value (5)
- Thread 2: Reads the same counter value (5)
- Thread 1: Adds 1 and writes back (→6)
- Thread 2: Adds 1 and writes back (→6)

The expected result should be 7, but we get 6 because both threads read the original value before either one updated it. Using Fetch&Add would guarantee the correct result by making the entire operation atomic.

Using Test&Set for Mutual Exclusion

We can use these tools to solve the mutual exclusion problem:

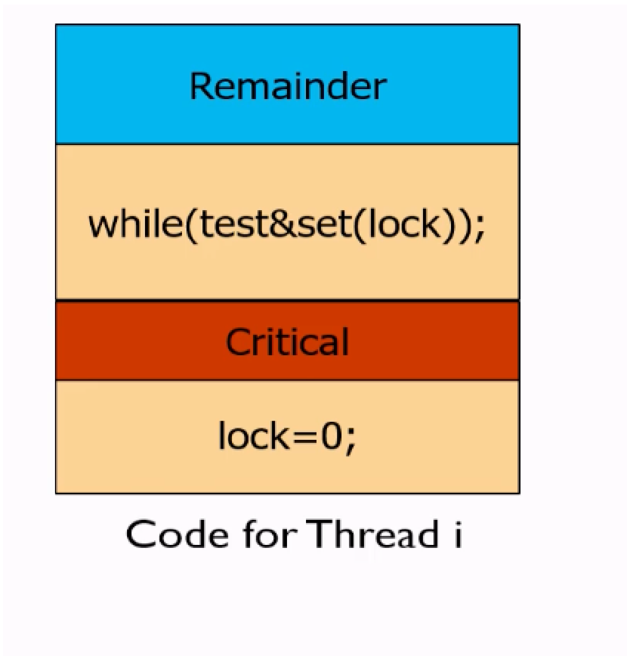


Figure 19: Mutual Exclusion with Test & Set

The entry code simply uses test and set: when the return value of test and set is 1, the state becomes occupied and someone is in the critical section. When thread 0 arrives, it calls test and set, if it returns 0, then thread 0 enters and sets the return value to 1, preventing entry of other threads until it exits the exit code which changes lock to 0.

The problem is that starvation freedom does not hold: one thread may enter all the time, while another thread waits indefinitely.

Burns' Algorithm

Burns' algorithm uses two arrays:

- `waiting[]` - Boolean array indicating whether a process intends to enter the critical section
- `key[]` - Integer array used to determine priority among competing processes

The algorithm also uses a global `lock` variable that processes attempt to acquire using the atomic test-and-set operation.

This operation returns the old value of `lock` and sets it to 1 in one atomic step. If it returns 0, the lock was free and the caller has now acquired it.

Entry Protocol The entry protocol works as follows:

1. Process P_i sets $waiting[i]$ to `true` to indicate its intention to enter the critical section.
2. It sets $key[i]$ to 1, establishing its initial priority.
3. It enters a waiting loop where it attempts to acquire the lock using the atomic test-and-set operation.
4. The process continues to attempt to acquire the lock as long as both $waiting[i]$ and $key[i]$ are true.
5. When the process successfully acquires the lock ($key[i]$ becomes 0) or is signaled by another process ($waiting[i]$ becomes false), it exits the loop.
6. The process sets $waiting[i]$ to `false` before entering the critical section.

Exit Protocol After executing its critical section, process P_i executes the exit protocol:

1. It starts checking other processes, beginning with the next process ($(i + 1) \bmod n$).
2. It continues checking until it either finds a waiting process or completes a full circle back to itself.
3. If it finds a waiting process P_j, it signals that process by setting $waiting[j]$ to `false`.
4. If no process is waiting, it releases the lock by setting $lock$ to 0.

Note that key = 0 means that the lock is successfully acquired by the process.

Test-and-set is fundamental to Burns' algorithm as it prevents the scenario where: Thread A reads lock = 0, thread B reads lock = 0, and both set lock to 1 and enter the critical section simultaneously.

We note that all the solutions we've discussed so far use busy-waiting, which is a waste of CPU resources.

Mutual exclusion is only one synchronization problem; we will discover more later in the course.

Operating System Support

Synchronization Primitives

In the world of solving mutual exclusion problems, there is a problem of busy-waiting. Therefore, the OS defines some abstract data types, which are used to provide synchronization without busy-waiting. These are usually provided by programming language libraries.

Semaphores

A semaphore consists of 2 fields: value and List. It has 2 operations, up and down, where down means acquiring the lock and up releases the lock. These operations are executed atomically.

As a result, we use down for the entry code and up for the exit code

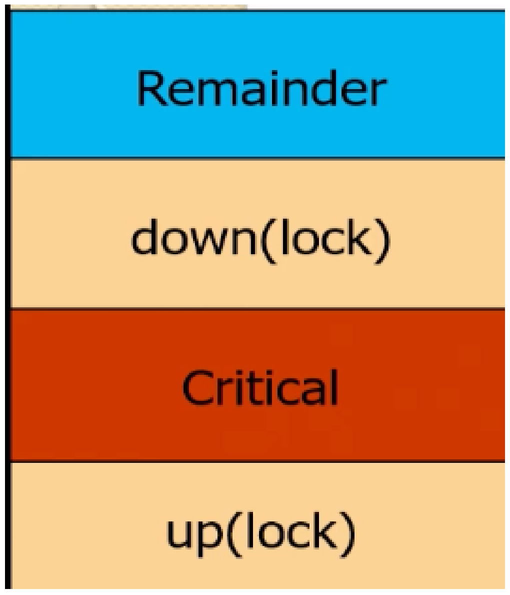


Figure 20: Semaphore Structure

Detailed Explanation of Semaphores

A semaphore is a synchronization primitive introduced to solve the mutual exclusion problem without busy waiting. Unlike the solutions we've discussed previously (Peterson's Algorithm, Lamport's Bakery Algorithm, etc.), semaphores avoid consuming CPU cycles when a process is waiting for access to a critical section.

Semaphore Structure A semaphore consists of:

- **Value:** An integer that represents the state of the semaphore:
  - Positive value: Number of resources available (or number of processes that can enter), which cannot be more than 1.

  - Zero: One process in the Critical section.
  - Negative value: The absolute value indicates how many processes are waiting

- **List (L):** A queue that holds processes that are waiting to access the semaphore

Semaphore Operations A semaphore supports three atomic operations:

- **Init(S, v):** Initializes the semaphore S with value v

```
S.value = v
```

- **Down(S):** Also known as P, wait, or acquire in different systems

```
S.value = S.value - 1
if S.value < 0 then
{
    add this thread to S.L
    sleep()
}
```

- **Up(S):** Also known as V, signal, or release in different systems

```
S.value = S.value + 1
if S.value <= 0 then
{
    remove a thread T from S.L
    wakeup(T)
}
```

How Semaphores Avoid Busy Waiting Unlike previous solutions that use busy waiting (continuously checking a condition in a loop), semaphores put waiting processes to sleep, releasing the CPU for other tasks. When the semaphore becomes available, the operating system wakes up a waiting process, allowing it to continue execution. This approach is much more efficient in terms of CPU utilization.

How Mutual Exclusion Works with Semaphores

1. The lock semaphore is initialized to 1, indicating that one process can enter the critical section.

2. When a process wants to enter the critical section, it performs Down(lock):

   - If lock.value is 1, it decrements it to 0 and enters the critical section.

- If lock.value is 0, it decrements it to -1, adds itself to the waiting list, and goes to sleep.

3. When a process exits the critical section, it performs Up(lock):

   - It increments lock.value.
   - If processes are waiting (lock.value $\leqslant 0$ after the increment), it wakes up one waiting process.