Operating System
Tutorial 5 - Concurrency

Yonghao Lee

May 4, 2025

The General Critical Section Problem

Let there be n processes: $P_0, P_1, \ldots, P_{n-1}$.

The problem we face is to implement a general mechanism for processes to enter and leave a **critical section** (CS).

We introduce the following success criteria for implementations:

1. **Mutual Exclusion:** Only one process is allowed in the critical section at any given time.
2. **Progress:** Essentially, if processes want to enter the CS, one must eventually be allowed to enter.
3. **Starvation Free (Bounded Waiting):** There must exist a bound on the number of times that other processes are allowed to enter their critical sections after a process has made a request to enter its critical section and before that request is granted. (No process should wait indefinitely to enter the CS).
4. **Generality:** The solution should work correctly for N processes and potentially multiple processors.
5. **No Blocking in Remainder Section:** A process running in its remainder section (outside the CS) must not be able to prevent other processes from entering the critical section. This almost always holds.

The Dining Philosophers Problem

This is a classical synchronization problem.

Problem Description

- Five philosophers sit around a circular table.

- In the center of the table is a large bowl of rice.
- Between each pair of adjacent philosophers, there is a single chopstick.
- Philosophers alternate between thinking and eating.
- To eat, a philosopher must pick up the two chopsticks immediately to their left and right.
- A philosopher can only pick up one chopstick at a time.

The challenge is to design a protocol that allows philosophers to eat without deadlock (where all philosophers pick up one chopstick and wait indefinitely for the other) and without starvation (where a philosopher is perpetually unable to acquire both chopsticks).

Shared Data Representation

A common way to model this involves:

- The bowl of rice (representing a shared resource, though access to it is implicitly controlled by chopsticks).
- An array of semaphores, ‘chopstick[5]‘, where each semaphore represents one chopstick and is initialized to 1 (available).

```
while (true) {
    // Acquire chopsticks
    down(chopstick[i]);          // Pick up left chopstick
    down(chopstick[(i + 1) % 5]);  // Pick up right chopstick

    // Eat
    eat();

    // Release chopsticks
    up(chopstick[(i + 1) % 5]);    // Put down right chopstick
    up(chopstick[i]);            // Put down left chopstick

    // Think
    think();
}
```

This straightforward solution can lead to **deadlock**. If all five philosophers pick up their left chopstick simultaneously, none can pick up their right chopstick, and they will wait forever. This solution does solve the mutual exclusion problem.

Solutions to Deadlock

It's known that for the Dining Philosophers problem, there is no *symmetric* solution (where all philosophers run the exact same code without variation) that guarantees both deadlock-free and starvation-free operation using only semaphores for chopsticks in this simple manner. Any purely symmetric algorithm tends to lead to either deadlock or starvation.

Possible solutions often involve breaking this symmetry or introducing coordination:

- **Use a Waiter (Arbiter):** A central agent controls entry. A philosopher must ask the waiter for permission before picking up any chopsticks.
- **Use Asymmetric Solutions:** Philosophers behave slightly differently based on some criteria.
    1. **Odd/Even Rule:** Odd-numbered philosophers pick up their left chopstick first, then right. Even-numbered philosophers pick up right first, then left. This breaks the circular dependency.
    2. **Randomized Approach (Lehmann-Rabin):** Introduce randomness to break deterministic deadlock patterns.

Lehmann-Rabin Algorithm

This algorithm provides a randomized solution to avoid deadlock. Each philosopher executes the following logic concurrently:

Pseudocode (for one philosopher)

```
repeat
    // 1. Randomly decide which chopstick to try first
    if coinflip() == 0 then
        first = left
    else
        first = right
    end if

    // 2. Wait until the first choice is available and pick it up
    wait until chopstick[first] == false  // Assuming false means available
    chopstick[first] = true               // Assuming true means taken

    // 3. Check if the second chopstick is available
    if chopstick[~first] == false then   // ~first means the other chopstick
        // 4a. If yes, pick it up and exit the loop to eat
        chopstick[~first] = true
        break  // Exit repeat loop, ready to eat
    else
        // 4b. If no, release the first chopstick and try again
        chopstick[first] = false
    end if
end repeat  // Continue loop if second chopstick wasn't available

// 5. Eat
eat()

// 6. Release both chopsticks
chopstick[left] = false
chopstick[right] = false

// 7. Think
think()  // Not explicitly shown in pseudocode, but follows releasing
```

Explanation

The algorithm avoids deadlock through two main mechanisms when executed concurrently by all philosophers:

- **Randomization:** Each philosopher independently chooses whether to try left or right first. This makes it highly unlikely that all philosophers will attempt to grab chopsticks in the same pattern that leads to the circular wait of the naive deadlock. Symmetry is broken probabilistically.
- **Release on Failure:** If a philosopher picks up their first chosen chopstick but finds the second one is unavailable, they *release* the first one before trying again. This breaks the "hold and wait" deadlock condition, preventing philosophers from holding one resource indefinitely while waiting for another that may never become free in a deadlock situation. This allows other philosophers a chance to acquire the released chopstick.

Probabilistically, every philosopher is expected to eventually get their turn. The randomization ensures that over time, the chances of one philosopher being continuously denied access become vanishingly small.

The Bounded-Buffer Problem (Producer-Consumer Problem)

This classical synchronization problem involves a shared, fixed-size (typically cyclic) buffer capable of holding N items.

Two types of processes interact with the buffer:

- **Producers:** Generate items and place them into the buffer.
- **Consumers:** Remove items from the buffer and process (consume) them.

Synchronization Constraints

The core challenges are to ensure correct coordination:

- Producers must not add data to the buffer if it is full. They must wait until space becomes available.
- Consumers must not attempt to remove data from the buffer if it is empty. They must wait until an item becomes available.

Furthermore, since the buffer is a shared resource accessed concurrently, operations that modify the buffer (like adding or removing items and updating pointers/counts) must be protected by **mutual exclusion** to prevent race conditions.

Semaphore-Based Solution

A common solution utilizes three counting semaphores:

- `mutex`: A binary semaphore (or mutex) initialized to 1. It ensures mutual exclusion for operations that access the buffer structure itself (e.g., pointers, indices). Only one process can hold the mutex at a time.
- `fillCount` (or `filledSlots`): A counting semaphore initialized to 0. It tracks the number of items currently present in the buffer (i.e., filled slots). Consumers perform a ‘down()‘ (wait) operation on this semaphore; if it’s 0 (buffer empty), they block until a producer increments it. Producers perform an ‘up()‘ (signal) operation after adding an item.
- `emptyCount` (or `emptySlots`): A counting semaphore initialized to the buffer capacity, N. It tracks the number of empty slots available in the buffer. Producers perform a ‘down()‘ (wait) operation on this semaphore; if it’s 0 (buffer full), they block until a consumer increments it. Consumers perform an ‘up()‘ (signal) operation after removing an item.

Producer and Consumer Pseudocode

The logic for producers and consumers uses the semaphores to ensure correct synchronization:

- **Producers** wait on ‘emptyCount‘ before producing (blocking if the buffer is full) and signal ‘fillCount‘ after producing.
- **Consumers** wait on ‘fillCount‘ before consuming (blocking if the buffer is empty) and signal ‘emptyCount‘ after consuming.
- Both use the ‘mutex‘ semaphore to ensure exclusive access when actually manipulating the buffer data structure (adding or removing items).

Producer Logic:

```
while (true) {
  produce an item

  // Wait for an empty slot
  down(emptyCount);

  // Acquire mutex for buffer access
  down(mutex);

  // Add item to buffer
  add the item

  // Release mutex
  up(mutex);

  // Signal that buffer has one more item
  up(fillCount);
}
```

Consumer Logic:

```
while (true) {
  // Wait for a filled slot
  down(fillCount);

  // Acquire mutex for buffer access
  down(mutex);

  // Remove item from buffer
  pop an item

  // Release mutex
  up(mutex);

  // Signal that buffer has one more empty slot
  up(emptyCount);

  // Consume the item (outside mutex)
  consume the item
}
```

The Readers-Writers Problem

This is another classic synchronization problem involving access to a shared data structure by multiple concurrent processes.

Two types of processes access the data:

- **Readers:** Only read the data structure; they do not perform any updates.
- **Writers:** Can both read and write the data structure.

Synchronization Goals

The problem requires implementing access control that satisfies these conditions:

- Multiple readers should be allowed to read the data concurrently.
- Only one writer should be allowed to access the data structure at any given time (for writing or reading).
- If a writer is currently writing, no reader should be allowed to read.

Essentially, readers can share access with other readers, but writers need exclusive access.

First Solution Attempt (Reader Priority)

This approach uses semaphores to coordinate access. However, it gives priority to readers, potentially leading to writer starvation if readers arrive continuously.

Shared Data

- `int readCount`: Initialized to 0. Tracks the number of readers currently accessing the data.
- `Semaphore readCount_mutex`: Initialized to 1. Provides mutual exclusion for updating `readCount`.
- `Semaphore write`: Initialized to 1. Controls access for writers and is used by the first/last reader to block/allow writers.

Writer Process Pseudocode

The writer's logic is straightforward: acquire exclusive access using the `write` semaphore, perform the write, and release the lock.
Writer Logic:

```
while (true) {
  // Wait for exclusive access
  down(write);

  // Critical Section: Perform writing
  writing_is_performed();

  // Release exclusive access
  up(write);
}
```

Explanation: The `down(write)` ensures only one writer can enter the critical section. If the `write` semaphore is held (either by another writer or by readers, as seen below), the writer blocks. `up(write)` releases the lock.

Reader Process Pseudocode

The reader's logic manages the `readCount` and interacts with the `write` semaphore only when the first reader enters or the last reader leaves.

Reader Logic:

```
while (true) {
  // Entry Section
  down(readCount_mutex);
  readCount++;
  if (readCount == 1) {
    down(write);  // First reader locks out writers
  }
  up(readCount_mutex);

  // Critical Section: Perform reading
  reading_is_performed();

  // Exit Section
  down(readCount_mutex);
  readCount--;
  if (readCount == 0) {
    up(write);  // Last reader allows writers
  }
  up(readCount_mutex);

  // Remainder section (if any)
}
```

Explanation:

- Readers use `readCount_mutex` to safely increment/decrement `readCount`.
- The **first reader** (`readCount == 1`) attempts to acquire the `write` semaphore. If a writer holds it, the reader blocks. If successful, it prevents any writer from starting. Note that if reader 1 blocks (on `down(write)`), it essentially also holds the mutex for `readCount_mutex`, which blocks other readers from entering (at their `down(readCount_mutex)` step).
- Subsequent readers just increment `readCount` (once they get past the mutex) and proceed to read concurrently.
- The **last reader** (`readCount == 0` after decrementing) releases the `write` semaphore, potentially allowing a blocked writer to proceed.

Flaw: Writer Starvation

This solution works correctly in terms of mutual exclusion and allowing concurrent reads. However, if readers keep arriving frequently enough, `readCount` may never become 0. This means the `write` semaphore, once acquired by the first reader, might never be released by the last reader, causing any waiting writers to **starve** (wait indefinitely). Furthermore, as noted in the explanation above, the implementation detail of holding `readCount_mutex` while attempting `down(write)` can unnecessarily block subsequent readers if the first reader has to wait for a writer.

Second Solution Attempt (Writer Priority)

To address the writer starvation problem of the first solution, this approach adds mechanisms to give priority to waiting writers.

Additional Shared Data

In addition to `readCount`, `readCount_mutex`, and `write` from the first solution, we add:

- `Semaphore read`: Initialized to 1. Used by writers to block incoming readers when a writer wants to write. Readers must acquire this before checking `readCount_mutex`.
- `int writeCount`: Initialized to 0. Tracks the number of writers that are waiting to write or are currently writing.
- `Semaphore writeCount_mutex`: Initialized to 1. Provides mutual exclusion for updating `writeCount`.
- `Semaphore queue`: Initialized to 1. Used by readers to ensure only one reader at a time attempts to acquire the `read` lock, preventing them from unfairly blocking waiting writers in succession. Acts as a turnstile or entry queue for readers.

Writer Process Pseudocode (Writer Priority)

The writer logic now includes steps to increment/decrement `writeCount` and potentially block readers using the `read` semaphore.

Writer Logic (Writer Priority):

```
while (true) {
  // Entry section: Announce intent to write
  down(writeCount_mutex);
  writeCount++;
  if (writeCount == 1) {
    down(read);  // First waiting writer blocks readers
  }
  up(writeCount_mutex);

  // Wait for exclusive write access (same as before)
  down(write);
  // --- Critical Section: Perform writing ---
  writing_is_performed();
  up(write);

  // Exit section: Announce finished writing
  down(writeCount_mutex);
  writeCount--;
  if (writeCount == 0) {
    up(read);  // Last waiting writer unblocks readers
  }
  up(writeCount_mutex);
}
```

Explanation: Writers first register their intent by incrementing ‘writeCount‘ (protected by ‘writeCount_mutex‘). If a writer is the *first* one waiting (‘writeCount == 1‘), it acquires the ‘read‘ semaphore, blocking any new readers. Then, it waits for the ‘write‘ semaphore as before to ensure exclusive writing. After writing, it decrements ‘writeCount‘, and if it was the *last* writer (‘writeCount == 0‘), it releases the ‘read‘ semaphore, allowing readers again. This prioritizes waiting writers over incoming readers.

Reader Process Pseudocode (Writer Priority)

The reader logic is modified to check the ‘read‘ semaphore (respecting writer priority) and use the ‘queue‘ semaphore.

Reader Logic (Writer Priority):

```
while (true) {
  // Entry Queue & Check Writer Priority
  down(queue);            // Ensure fair reader queueing
  down(read);             // Wait if writers are waiting/active

  // Standard reader entry (from first solution)
  down(readCount_mutex);
  readCount++;
  if (readCount == 1) {
    down(write);          // First reader locks resource from writers
  }
  up(readCount_mutex);

  // Allow next reader through checks & let others proceed past read lock
  up(read);
  up(queue);

  // --- Critical Section: Perform reading ---
  reading_is_performed();

  // Standard reader exit (from first solution)
  down(readCount_mutex);
  readCount--;
  if (readCount == 0) {
    up(write);            // Last reader releases resource for writers
  }
  up(readCount_mutex);
}
```

Explanation: Readers first wait on the `queue` semaphore. This acts as an entry queue or turnstile, ensuring only one reader at a time proceeds to check writer priority. This prevents a scenario where multiple readers might simultaneously attempt `down(read)` just as a writer signals its intent, potentially complicating fairness. After passing the queue, a reader attempts `down(read)`; this blocks if any writer is waiting or writing. Only after passing the `read` lock does the reader proceed with the standard logic (using `readCount_mutex` and `write`) from the first solution to manage concurrent reading. `up(read)` is called relatively quickly to allow other waiting readers (who have passed the queue) through the `read` check if possible, and `up(queue)` allows the next reader in line to attempt entry. This structure gives writers priority.

Potential Flaw

While this solution prioritizes writers and solves their starvation problem, it can potentially lead to **reader starvation** if writers arrive continuously.

Question from the 2012 B Exam

3. When multiple processes access shared data structures, problems may arise.

   (a) Below is a solution presented for the critical section problem:

```
shared boolean flag[2] = {false};
shared int turn = 0;
// Code for process i (i is 0 or 1)
do
{
    flag[i] = true;
    turn = i;                     // Process i sets turn to itself
    while (flag[1-i] && turn == i); // Wait if other wants in AND it's my turn
    // --- critical section ---
    critical section
    flag[i] = false;              // Indicate process i is done
    // --- remainder section ---
    remainder section
} while (true);
{}
```

   What is your opinion on this implementation?

   (i) It ensures mutual exclusion in a system with two processes.

   (ii) It does not ensure a solution to the critical section problem in a system with two processes.

   (iii) It is possible to solve the critical section problem for two processes by adding a few characters to the code.

   (iv) It is possible to solve the critical section problem for any number of processes with a single line change in the program. (2 points)

Analysis of the Code and Options

The provided code attempts to implement a solution similar to Peterson's Algorithm but contains a subtle flaw in the line `turn = i;`. This flaw can lead to deadlock, violating the Progress requirement of a correct critical section solution.

Let's evaluate the options based on this flawed implementation:

(i) It ensures mutual exclusion in a system with two processes.
Consider the following execution sequence:

1. **P0 executes L1:** Sets `flag[0] = true`.

2. **P0 executes L2:** Sets `turn = 0`. (Shared State: `flag={true, false}, turn=0`)
3. **P0 executes L3:** Evaluates `while (flag[1] && turn == 0)`.
   - Reads `flag[1]` (false).
   - The first part of the AND condition (`flag[1]`) is false.
   - The entire `while` condition is false.
4. **P0 executes L4:** P0 enters its critical section.
5. *Context Switch: P0 is preempted while in its critical section.*
6. **P1 executes L1:** Sets `flag[1] = true`.
7. **P1 executes L2:** Sets `turn = 1`. (Shared State: `flag={true, true}, turn=1`, P0 is in CS)
8. **P1 executes L3:** Evaluates `while (flag[0] && turn == 1)`.
   - Reads `flag[0]` (true).
   - Reads `turn` (1).
   - The second part of the AND condition (`turn == 1`) is true.
   - The entire `while` condition is true (`true && true` is true).
9. **P1 waits:** P1 blocks at the while loop.
10. **Mutual Exclusion Succeeds:** P1 does not enter its critical section while P0 is inside.

(ii) It does not ensure a solution to the critical section problem in a system with two processes.
Explanation: This statement is **TRUE**. Although the algorithm ensures mutual exclusion, it fails the progress requirement. If both processes try to enter simultaneously, they can set their respective flags and turns, leading to a situation where P0 waits for P1 and P1 waits for P0—a deadlock.

(iii) It is possible to solve the critical section problem for two processes by adding a few characters to the code.
Explanation: This statement is **TRUE**. The deadlock can be fixed by changing the line `turn = i;` to the standard Peterson's assignment `turn = 1-i;` (yielding the turn). This modification (changing one character to three) makes the algorithm correct for two processes.

(iv) It is possible to solve the critical section problem for any number of processes with a single line change in the program.
Explanation: This statement is **FALSE**. Peterson's algorithm, corrected or not, is fundamentally for two processes only. N-process solutions require different algorithms.

Question 8 from 2024 A Exam

Below is a C++ implementation of a proposed mutual exclusion algorithm for N threads:

```
// Global (shared) variables
#define N ... // N >=1
bool Choosing[N]; // all entries initialized to false
uint8_t Number[N]; // initialized to 0

// REMAINDER CODE, no variable values are changed

// Entry code for thread i where 0<= i <N
Choosing[i]=true;
uint8_t mx=Number[0];
for (int k=0;k<N;k++)
{
    mx=(Number[k]>mx)?Number[k]:mx;
}
Number[i]=mx+1;
Choosing[i]=false;
for (int j=0;j<N;j++)
{
    while(Choosing[j]);
    while((Number[j]!=0) && ((Number[j]<Number[i]) || ((Number[j]==
       Number[i]) && (j<i))));
}

// CRITICAL SECTION

// Exit code for thread i where 0<= i <N
Number[i]=0;
```

The parts marked in the code as “Remainder Code” and “Critical Section” have been omitted. Assume that the programmer allows each thread to run indefinitely amount of times. From among the following statements related to the algorithm described above, mark the correct one(s):

1. This is a correct implementation of the bakery algorithm.
2. This is an incorrect implementation of the bakery algorithm that does not maintain mutual exclusivity.
3. This is an incorrect implementation of the bakery algorithm that does not maintain progress.
4. This is an incorrect implementation of the bakery algorithm that does not maintain starvation freedom.
5. This is an incorrect implementation of the bakery algorithm that does not maintain generality.

Answer

The correct options to choose are:

- Option 2: This is an incorrect implementation of the bakery algorithm that does not maintain mutual exclusivity.
- Option 4: This is an incorrect implementation of the bakery algorithm that does not maintain starvation freedom.

The Core Problem: Ticket Number Overflow

1. **Assigning Tickets:** When a thread `i` wants to enter the critical section, it looks at all current ticket numbers, finds the maximum (`mx`), and tries to set its own ticket number `Number[i]` to `mx` + 1.
2. **The Overflow:** If the highest current ticket number `mx` is 255, then `mx` + 1 calculates to 256. However, since `Number[i]` is a `uint8_t`, it cannot store 256. The value wraps around (like an odometer rolling over), and `Number[i]` becomes `0`.

This unintended assignment of `0` breaks the algorithm in two critical ways:

1. Failure of Mutual Exclusion (Why Option 2 is Correct)

- **Intended Meaning of 0:** In the Bakery algorithm, `Number[j] = 0` is supposed to mean that thread `j` is *not* interested in entering the critical section.
- **The Conflict:** When thread `i` gets `Number[i] = 0` due to overflow, it now looks like it's "not interested" according to the algorithm's logic, even though it just took a ticket!
- **Skipping the Wait:** The code includes a wait condition:
  `while((Number[j]!=0) && ((Number[j]<Number[i]) || ((Number[j]==Number[i]) && (j<i))));`
  This means thread `i` should wait if thread `j` is interested (`Number[j]!=0`) *and* thread `j` has a higher priority (lower number or lower index with the same number).
- **The Bug:** If thread `i` has `Number[i] = 0` (from overflow) and checks thread `j` which *is* in the critical section (so `Number[j] > 0`), the priority check `(Number[j] < Number[i])` becomes `(Number[j] < 0)`. Since `Number[j]` is unsigned and positive, this is always false. Thread `i` incorrectly determines it has priority over `j`.
- **Result:** Thread `i` bypasses the wait loop and enters the critical section, even if another thread `j` is already inside. This violates mutual exclusion – multiple threads can be in the critical section simultaneously.

2. Failure of Starvation Freedom (Why Option 4 is Correct)

- **Overflow Breaks Fairness:** Imagine thread `A` arrives and gets ticket `Number[A] = 255`. Later, thread `B` arrives, and due to overflow, gets `Number[B] = 0`.
- **Unfair Priority:** According to the algorithm's comparison logic `((Number[j] < Number[i]) || ((Number[j] == Number[i]) && (j < i)))`, the ticket `0` is the "best" or highest priority ticket. So, thread `B` (with 0) gets higher priority than thread `A` (with 255), even though `A` arrived earlier.
- **Result:** Thread `B` will enter the critical section before `A`. If new threads keep arriving, and occasionally one overflows to get the ticket `0`, thread `A` (stuck with a high number like 255) might be overtaken indefinitely. It could wait forever, never getting its turn. This violates starvation freedom.