Process Synchronization

Advanced Operating Systems — Week 5
Hunter College, CUNY

Faye

February 2026

Contents

1 The Critical Section Problem

A **critical section** (CS) is a segment of code in which a process accesses shared resources (variables, files, tables, etc.). The general structure of a process with respect to the CS problem is:

1. **Entry section** — code that requests permission to enter the CS.
2. **Critical section** — the code that manipulates shared data.
3. **Exit section** — code executed upon leaving the CS.
4. **Remainder section** — the rest of the code.

Any correct solution to the critical section problem must satisfy all three of the following requirements:

Three Requirements for a Valid CS Solution

1. **Mutual Exclusion** — If process P_i is executing in its critical section, no other process may be executing in its critical section at the same time.
2. **Progress** — If no process is executing in its critical section and some processes wish to enter, then *only* those processes not in their remainder section can participate in deciding which will enter next, and this selection cannot be postponed indefinitely.
3. **Bounded Waiting** — There exists a bound (limit) on the number of times other processes are allowed to enter their critical sections after a process has made a request to enter its CS and before that request is granted.

2 Peterson's Solution

Peterson's solution is a classic software-based solution restricted to **two processes** that alternate execution between their critical sections and remainder sections. It uses two shared variables:

- `bool flag[2]` — `flag[i] = true` means process P_i is ready to enter its CS.
- `int turn` — indicates whose turn it is to enter the CS.

Listing 1: Peterson's Solution — Process P_i

```
/* Process P_i */
flag[i] = true;           // I want to enter
turn = j;                 // But I give the other process a chance
while (flag[j] && turn == j)
    ;                     // Busy wait

/* Critical Section */

flag[i] = false;          // I am done
/* Remainder Section */
```

Proof that all three requirements are satisfied:

1. **Mutual Exclusion**: Both P_i and P_j can be in the CS simultaneously only if `flag[i] == flag[j] == true` and `turn == i` and `turn == j` at the same time, which is impossible since `turn` is a single integer.

2. **Progress**: If P_j does not want to enter (`flag[j] == false`), then P_i enters immediately. If both want to enter, `turn` decides, and one will proceed without indefinite postponement.

3. **Bounded Waiting**: After P_i sets `flag[i] = true` and `turn = j`, if P_j is in the CS it will set `flag[j] = false` on exit, letting P_i in. If P_j re-enters, it sets `turn = i`, so P_i enters next. The bound is 1.

Limitations of Peterson's Solution

- Only works for **two processes**. Generalizing to n processes requires more complex algorithms (e.g., the bakery algorithm).
- On modern architectures, compilers and CPUs may **reorder instructions** for optimization. Without explicit **memory barriers** (fences), the stores to `flag` and `turn` may be reordered, breaking correctness. Peterson's solution is therefore primarily of theoretical interest on contemporary hardware.

3 Hardware Synchronization

Many systems provide special **atomic hardware instructions** that simplify CS solutions. Two widely used instructions:

3.1 test_and_set

Listing 2: `test_and_set` instruction (executed atomically)

```
bool test_and_set(bool *target) {
    bool rv = *target;      // Save old value
    *target = true;         // Set to true
    return rv;              // Return old value
}
```

Usage pattern for mutual exclusion:

Listing 3: Mutual exclusion with `test_and_set`

```
// Shared: bool lock = false;

while (test_and_set(&lock))
    ;                       // Busy wait
/* Critical Section */
lock = false;
/* Remainder Section */
```

3.2 compare_and_swap (CAS)

Listing 4: `compare_and_swap` instruction (executed atomically)

```
int compare_and_swap(int *value, int expected, int new_value) {
    int temp = *value;
    if (*value == expected)
```

```
        *value = new_value;
    return temp;
}
```

Usage pattern:

Listing 5: Mutual exclusion with `compare_and_swap`

```
// Shared: int lock = 0;

while (compare_and_swap(&lock, 0, 1) != 0)
    ;                          // Busy wait
/* Critical Section */
lock = 0;
/* Remainder Section */
```

Hardware Instructions Are Atomic

Both `test_and_set` and `compare_and_swap` are **single atomic hardware instructions**. They cannot be interrupted mid-execution. The processor guarantees that the read–modify–write cycle completes as one indivisible unit, even on multiprocessor systems (via bus locking or cache coherence protocols).

4 Mutex Locks

A **mutex lock** (mutual exclusion lock) is the simplest synchronization tool. A process must `acquire()` the lock before entering the CS and `release()` it upon exit.

Listing 6: acquire() and release() for a mutex lock

```
void acquire() {
    while (!available)
        ;                      // Busy wait (spinlock)
    available = false;
}

void release() {
    available = true;
}
```

Both `acquire()` and `release()` must be performed **atomically** (typically implemented using hardware instructions like CAS).

A mutex implemented with busy waiting is called a **spinlock**.

Mutex Lock Usage Pattern

```
pthread_mutex_t lock;
pthread_mutex_init(&lock, NULL);

pthread_mutex_lock(&lock);       // acquire
/* Critical Section */
pthread_mutex_unlock(&lock);     // release
```

When are spinlocks useful?

- When the CS is **short** (locking overhead of context switch would exceed the wait time).
- On **multiprocessor** systems where a thread can spin on one core while the lock holder runs on another.

5 Semaphores

A **semaphore** S is an integer variable that, apart from initialization, is accessed only through two standard atomic operations: `wait()` and `signal()`.

Listing 7: wait() and signal() — Classical Definitions

```
wait(S) {                  // Also called P() or down()
    while (S <= 0)
        ;                   // Busy wait
    S--;
}

signal(S) {                 // Also called V() or up()
    S++;
}
```

Types of semaphores:

- **Binary semaphore** — value can only be 0 or 1. Behaves like a mutex lock.
- **Counting semaphore** — value can range over an unrestricted integer domain. Used to control access to a resource with a finite number of instances.

5.1 Implementation Without Busy Waiting

To avoid busy waiting, the `wait()` operation can **block** the calling process and place it in a waiting queue associated with the semaphore. The `signal()` operation removes a process from the queue and wakes it up.

Listing 8: Semaphore with blocking queue

```
typedef struct {
    int value;
    struct list_head waiting_list;
} semaphore;

void wait(semaphore *S) {
    S->value--;
    if (S->value < 0) {
        // Add this process to S->waiting_list
```

```
        block();          // Suspend the process
    }
}

void signal(semaphore *S) {
    S->value++;
    if (S->value <= 0) {
        // Remove a process P from S->waiting_list
        wakeup(P);        // Resume process P
    }
}
```

P and V Operations (Historical Names)

- **P** (from Dutch *proberen*, "to test") ≡ `wait()` ≡ `down()` — decrements the semaphore. If the result is negative, the process blocks.
- **V** (from Dutch *verhogen*, "to increment") ≡ `signal()` ≡ `up()` — increments the semaphore. If processes are waiting, one is woken up.
- The `wait()` and `signal()` operations **must be atomic**: no two processes can execute them on the same semaphore simultaneously.

6 Classic Synchronization Problems

6.1 Bounded-Buffer Problem

A fixed-size buffer is shared between producers and consumers.

Shared data:

- `semaphore mutex = 1;` (protects access to the buffer)
- `semaphore full = 0;` (counts the number of full slots)
- `semaphore empty = N;` (counts the number of empty slots)

Listing 9: Bounded-Buffer — Producer

```
while (true) {
    /* Produce an item */
    wait(empty);              // Wait for an empty slot
    wait(mutex);              // Enter critical section

    /* Add item to buffer */

    signal(mutex);            // Leave critical section
    signal(full);             // Increment full count
}
```

Listing 10: Bounded-Buffer — Consumer

```
while (true) {
    wait(full);               // Wait for a full slot
    wait(mutex);              // Enter critical section

    /* Remove item from buffer */
```

```
    signal(mutex);              // Leave critical section
    signal(empty);              // Increment empty count
    /* Consume the item */
}
```

Bounded-Buffer Walkthrough (N

Step 1: Initial state: `mutex=1, full=0, empty=3`.

Step 2: Producer arrives: `wait(empty)` → `empty=2`, `wait(mutex)` → `mutex=0`, inserts item, `signal(mutex)` → `mutex=1`, `signal(full)` → `full=1`.

Step 3: Consumer arrives: `wait(full)` → `full=0`, `wait(mutex)` → `mutex=0`, removes item, `signal(mutex)` → `mutex=1`, `signal(empty)` → `empty=3`.

Step 4: If consumer arrives when `full=0`, it blocks until a producer signals.

6.2 Readers-Writers Problem

Multiple **readers** may access the shared database concurrently, but a **writer** requires exclusive access (no other readers or writers).

Shared data:

- `int read_count = 0;` (number of active readers)
- `semaphore mutex = 1;` (protects `read_count`)
- `semaphore rw_mutex = 1;` (mutual exclusion for writers; first/last reader also acquires/releases it)

Listing 11: Readers-Writers — Writer

```
while (true) {
    wait(rw_mutex);             // Exclusive access

    /* Writing is performed */

    signal(rw_mutex);
}
```

Listing 12: Readers-Writers — Reader

```
while (true) {
    wait(mutex);
    read_count++;
    if (read_count == 1)
        wait(rw_mutex);         // First reader locks out writers
    signal(mutex);

    /* Reading is performed */

    wait(mutex);
    read_count--;
    if (read_count == 0)
        signal(rw_mutex);       // Last reader lets writers in
    signal(mutex);
```

```
}
```

Variants:

- **First readers-writers** (shown above): readers have priority; writers may starve.
- **Second readers-writers**: once a writer is waiting, no new readers may start; readers may starve.
- Neither variant is fully satisfactory; practical systems use fairness mechanisms.

6.3 Dining Philosophers Problem

Five philosophers sit around a circular table. Between each pair of adjacent philosophers is a single chopstick (5 total). A philosopher needs **both** adjacent chopsticks to eat.

Naive solution (deadlock-prone):

Listing 13: Dining Philosophers — Naive (deadlocks!)

```
// semaphore chopstick[5]; each initialized to 1
while (true) {
    wait(chopstick[i]);           // Pick up left
    wait(chopstick[(i+1) % 5]);   // Pick up right

    /* Eat */

    signal(chopstick[(i+1) % 5]);
    signal(chopstick[i]);

    /* Think */
}
```

Deadlock scenario: If all 5 philosophers simultaneously pick up their left chopstick, each holds one and waits for the other — circular wait ⇒ deadlock.

Solutions to Prevent Deadlock in Dining Philosophers

1. **Limit to 4 philosophers at the table** — use a counting semaphore initialized to 4. At most 4 can attempt to pick up chopsticks, so at least one can always eat.
2. **Atomic pickup** — a philosopher picks up both chopsticks only if both are available (inside a critical section). If either is unavailable, neither is picked up.
3. **Asymmetric ordering** — odd-numbered philosophers pick up left then right; even-numbered pick up right then left. This breaks the circular wait condition.

7 Monitors

A **monitor** is a high-level synchronization construct that encapsulates shared data, operations on that data, and synchronization into a single module.

Key property: Only **one process** may be active within the monitor at a time. The compiler or runtime enforces mutual exclusion automatically.

Condition variables: Used within monitors for processes that need to wait for a particular condition. A condition variable `x` supports two operations:

- `x.wait()` — the calling process is suspended and placed in a queue associated with `x`.

- `x.signal()` — resumes exactly one suspended process waiting on `x`. If no process is waiting, the signal has no effect (unlike semaphore `signal`, which always increments).

Listing 14: Monitor pseudocode structure

```
monitor MonitorName {
    // Shared variable declarations
    condition x, y;

    procedure P1(...) {
        // ...
        x.wait();   // Block if condition not met
        // ...
    }

    procedure P2(...) {
        // ...
        x.signal(); // Wake up one waiting process
        // ...
    }

    initialization_code() {
        // ...
    }
}
```

Monitor Support in Modern Languages

- **Java**: `synchronized` keyword on methods or blocks. `wait()`, `notify()`, `notifyAll()` on objects serve as condition variables.
- **C#**: `lock` statement; `Monitor.Enter()`/`Monitor.Exit()`. `Monitor.Wait()`/`Monitor.Pulse()` for condition variables.
- **Python**: `threading.Condition` provides `acquire()`, `release()`, `wait()`, `notify()`, and `notify_all()`.

8 Deadlock Preview

A set of processes is in a **deadlock** state when every process in the set is waiting for an event that can only be caused by another process in the set.

Four Necessary Conditions for Deadlock

All four must hold simultaneously for a deadlock to occur:

1. **Mutual Exclusion** — At least one resource must be held in a non-sharable mode.
2. **Hold and Wait** — A process must be holding at least one resource and waiting to acquire additional resources held by other processes.
3. **No Preemption** — Resources cannot be forcibly removed from a process; they must be released voluntarily.
4. **Circular Wait** — A circular chain of processes exists such that each process holds a resource needed by the next process in the chain: $P_0 \to P_1 \to \cdots \to P_n \to P_0$.

9 Key Takeaways

Week 5 Summary

1. The **critical section problem** requires mutual exclusion, progress, and bounded waiting.
2. **Peterson's solution** solves the two-process case but needs memory barriers on modern hardware.
3. Hardware instructions (`test_and_set`, `compare_and_swap`) provide atomic building blocks for locks.
4. **Mutex locks** and **semaphores** are the fundamental OS synchronization primitives.
5. Classic problems (Bounded-Buffer, Readers-Writers, Dining Philosophers) illustrate common synchronization patterns and pitfalls.
6. **Monitors** raise the level of abstraction, letting the compiler enforce mutual exclusion.
7. **Deadlock** requires four simultaneous conditions; breaking any one prevents it.