Operating System
Lecture 5: Synchronization, Part 2

Yonghao Lee

May 1, 2025

Introduction to Semaphores

Motivation and Definition

In concurrent computing environments, multiple processes or threads frequently access shared resources, creating synchronization challenges. To address these coordination needs, operating systems provide fundamental synchronization constructs, with the **semaphore** being one of the most versatile mechanisms.

A semaphore S is implemented as a record containing two essential components:

- $S.value$: An integer variable that serves as both a counter and a signaling mechanism.
- $S.L$: A list structure (typically implemented as a queue) that holds processes or threads currently blocked on this semaphore.

Core Operations

Three fundamental, *atomic* operations define a semaphore's behavior:

$Init(S, v)$

This operation initializes the semaphore with a starting value. Typically performed only once at semaphore creation:

1: $S.value \leftarrow v$ ▷ Set initial value to v, where $v \geqslant 0$

$Down(S)$ **(also called** $Wait(S)$ **or** $P(S)$**)**

This operation attempts to acquire the semaphore, blocking the caller if the resource is unavailable:

1: $S.value \leftarrow S.value - 1$ ▷ Decrement value (request resource)
2: **if** $S.value < 0$ **then** ▷ If negative, resource was not available
3: Add calling process/thread to $S.L$
4: Block() ▷ Put process/thread to sleep
5: **end if**

$Up(S)$ **(also called** $Signal(S)$ **or** $V(S)$**)**

This operation releases the semaphore and potentially wakes up a waiting process:

1: $S.value \leftarrow S.value + 1$ ▷ Increment value (release resource)
2: **if** $S.value \leqslant 0$ **then** ▷ If non-positive, processes were waiting
3: Select a process/thread T from $S.L$
4: Remove T from $S.L$
5: Wakeup(T) ▷ Make T ready to run
6: **end if**

Note: The atomicity of these operations is crucial for correctness. The operating system ensures that each operation executes indivisibly without interruption.

Applications of Semaphores

Semaphores provide elegant solutions to various synchronization challenges in concurrent systems. The following sections explore several common applications.

Application 1: Mutual Exclusion (Coarse-Grained Locking)

Concept and Implementation Pattern

Mutual exclusion ensures that only one thread or process accesses a critical section at any given time, preventing race conditions on shared resources. This application typically involves a single lock protecting an entire operation.

- **Setup:** Define a shared semaphore (`lock`) initialized to 1: `Init(lock, 1)`.
- **Code Structure:** Enclose the critical section between `Down` and `Up` operations.

```
... Remainder Section ...
Down(lock);     // Acquire lock (blocks if held by another)
// --- Critical Section ---
// Access shared resources
// --- End Critical Section ---
Up(lock);       // Release lock
... Remainder Section ...
```

Semaphore Value and Properties (Mutex Context)

- **Value Interpretation:**
  - `lock.value = 1`: Lock is available.
  - `lock.value = 0`: Lock is held by exactly one process.
  - `lock.value = -x` ($x > 0$): Lock is held, and x processes are waiting.
- **Properties:** This pattern guarantees Mutual Exclusion. Progress and Bounded Waiting properties also hold, particularly when the waiting list $S.L$ implements a fair scheduling policy such as FIFO.
- **Binary Semaphore / Mutex:** A semaphore initialized to 1 and used in this manner is termed a **binary semaphore** and is functionally equivalent to a **Mutex**.

Example: Bank Transfers (Single Global Lock)

- **Scenario:** Multiple threads concurrently transfer money between accounts (e.g., T1: A→B, T2: B→A).
- **Problem:** Race conditions occur if account reads and writes interleave.

- **Solution:** Implement a single global `lock` (initialized to 1) to protect the entire `Move` operation across all threads.

```
// Thread performing any transfer X -> Y
Down(lock);
Move X -> Y; // Critical Section
Up(lock);
```

- **Analysis:** While this approach prevents race conditions, it limits concurrency by unnecessarily blocking unrelated transfers (e.g., A→B blocks C→D).

Application 2: Coordination (Execute B after A)

Concept and Implementation Pattern

This pattern enforces a specific execution order between operations in different threads or processes. The goal is to ensure that segment A in Process 1 completes before segment B in Process 2 begins.

- **Setup:** Define a shared semaphore (`flag`) initialized to 0: `Init(flag, 0)`. This initial value indicates 'A has not completed yet'.
- **Code Structure:** Process 1 signals after completing A, while Process 2 waits for this signal before starting B.

Process 1:

```
... Perform A ...
Up(flag);   // Signal A is complete
```

Process 2:

```
Down(flag); // Wait for signal (blocks if flag is 0 or less)
... Perform B ...
```

Mechanism Explanation

- **Case 1 - Process 2 arrives first:** When Process 2 calls `Down(flag)`, the semaphore value becomes -1, and Process 2 blocks. Later, when Process 1 calls `Up(flag)`, the value becomes 0, and Process 2 is awakened.
- **Case 2 - Process 1 arrives first:** When Process 1 calls `Up(flag)`, the value becomes 1. Later, when Process 2 calls `Down(flag)`, the value returns to 0, and Process 2 proceeds without blocking.

In both scenarios, `Down(flag)` only completes after `Up(flag)` has executed, guaranteeing that A completes before B starts. This establishes a happens-before relationship between the processes.

Application 3: Fine-Grained Locking (Per-Resource Locks)

Concept and Motivation

To improve concurrency over coarse-grained approaches, fine-grained locking associates a separate semaphore with each shared resource. Operations then acquire only the locks for the specific resources they need to access.

Example: Bank Transfers (Per-Account Locks)

- **Setup:** Create semaphores `SA, SB, SC, ...` for each account, all initialized to 1.
- **Code Structure:** A transfer operation requires locking both source and destination accounts. Crucially, a consistent lock acquisition order must be followed to prevent deadlocks.

```
// Example: Thread performing Transfer A -> B
// Assumes a deadlock-prevention order is enforced (e.g., by account ID)
Acquire_Lock_In_Order(SA, SB); // Function acquires SA and SB in correct order

// --- Critical Section START ---
Move A -> B;
// --- Critical Section END ---

Release_Lock(SA);
Release_Lock(SB);
```

(Note: The lock acquisition/release is abstracted to emphasize the importance of consistent ordering)

Challenge: Deadlock Potential

- **Benefit:** Fine-grained locking enables non-conflicting operations (e.g., A→B and C→D) to execute concurrently.
- **Risk:** If different threads acquire multiple locks in inconsistent orders, they may create **deadlock** through circular wait conditions.
- **Prevention:** Avoiding deadlocks requires careful design, typically by enforcing a total ordering on lock acquisition (e.g., always lock resources in ascending numerical or alphabetical order).

Deadlock Analysis: Per-Account Bank Transfer Example

The following analysis explores how deadlocks can emerge in fine-grained locking systems when lock ordering is handled incorrectly.

Scenario Setup

- Four accounts: A, B, C, D.
- Four semaphores (mutexes): `SA, SB, SC, SD`, all initialized to 1.
- Four concurrent threads performing cyclic transfers: T1: $A \to B$, T2: $B \to C$, T3: $C \to D$, T4: $D \to A$.

Problematic Locking Strategy Used (Leads to Deadlock)

For this example, assume each thread acquires locks in the order: **destination first, then source**.

Thread 1 (A → B): Locks SB first, then SA.

Thread 2 (B → C): Locks SC first, then SB.

Thread 3 (C → D): Locks SD first, then SC.

Thread 4 (D → A): Locks SA first, then SD.

Execution Sequence Leading to Deadlock

Consider this possible concurrent execution:

1. **Each thread acquires its destination lock:**
   - T1 executes `Down(SB)` (acquires SB, SB.value=0).
   - T2 executes `Down(SC)` (acquires SC, SC.value=0).
   - T3 executes `Down(SD)` (acquires SD, SD.value=0).
   - T4 executes `Down(SA)` (acquires SA, SA.value=0).
2. **Each thread attempts to acquire its source lock:**
   - T1 attempts `Down(SA)`, but SA is held by T4 $\implies$ T1 blocks (SA.value=-1).
   - T2 attempts `Down(SB)`, but SB is held by T1 $\implies$ T2 blocks (SB.value=-1).
   - T3 attempts `Down(SC)`, but SC is held by T2 $\implies$ T3 blocks (SC.value=-1).
   - T4 attempts `Down(SD)`, but SD is held by T3 $\implies$ T4 blocks (SD.value=-1).

Result: Deadlock

The system now finds itself in a classic deadlock:

- T1 holds SB, waits for SA (held by T4).
- T2 holds SC, waits for SB (held by T1).
- T3 holds SD, waits for SC (held by T2).
- T4 holds SA, waits for SD (held by T3).

This forms a **circular wait** condition. No thread can proceed because each needs a resource held by another waiting thread.

Conclusion from Example

While fine-grained locking offers significant concurrency benefits, it requires rigorous attention to lock acquisition ordering to prevent deadlocks. A system-wide lock ordering protocol becomes essential for correctness when threads may need to acquire multiple locks simultaneously.

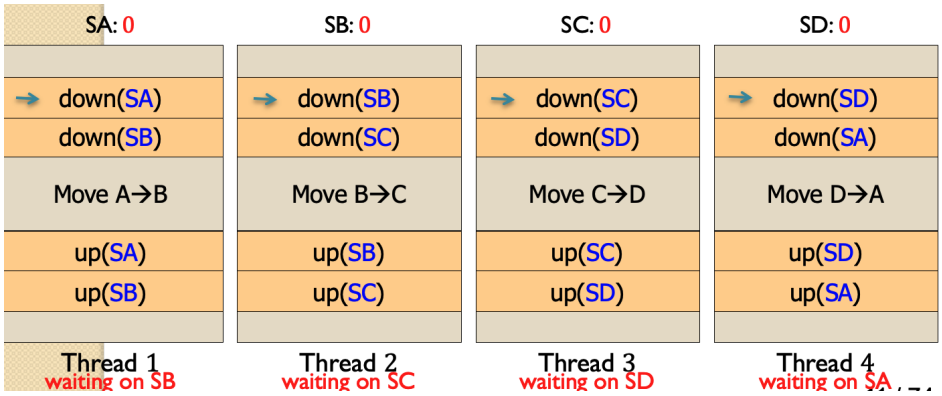

Figure 1: Deadlock Situation

The Dining Philosophers Problem

This canonical problem elegantly illustrates fundamental challenges in concurrency control, particularly deadlock and starvation scenarios.

- **Setup:**
  - Philosophers alternate between two activities: `Thinking` and `Eating`.
  - They sit around a circular table.
  - Between each adjacent pair of philosophers is a single chopstick.

- To eat, a philosopher needs *both* chopsticks: one to their left and one to their right.
- A philosopher can pick up only one chopstick at a time.
- Chopsticks are shared resources; two philosophers cannot simultaneously hold the same chopstick.

- **Initial Semaphore Solution:**
  - Each chopstick $c[i]$ is represented by a semaphore initialized to 1.
  - Code for philosopher i (where N is the number of philosophers, e.g., 6):

```
loop
    Think()
    down(chopstick[i])           // Acquire left chopstick
    down(chopstick[(i+1) mod N]) // Acquire right chopstick
    Eat()
    up(chopstick[i])             // Release left chopstick
    up(chopstick[(i+1) mod N])   // Release right chopstick
end loop
```

- **Deadlock Scenario:**
  - Consider the case where every philosopher executes their first step, `down(chopstick[i])`, simultaneously.
  - Each philosopher successfully acquires their **left** chopstick. All chopstick semaphores become 0.
  - Next, every philosopher attempts `down(chopstick[(i+1) mod N])` to acquire their **right** chopstick.
  - However, the right chopstick for philosopher i is the left chopstick for philosopher $(i + 1) \bmod N$, which is already held.
  - Every philosopher blocks, waiting for a chopstick held by their neighbor. This circular wait creates a deadlock.

Dining Philosophers: Deadlock Prevention by Breaking Symmetry

The deadlock in the original Dining Philosophers solution arises from symmetric behavior, where all philosophers follow identical strategies, creating a circular wait. Breaking this symmetry offers effective deadlock prevention. We examine two approaches, assuming $N = 6$ philosophers and chopsticks.

Solution 1: Asymmetric Last Philosopher

This approach breaks symmetry by making one philosopher (the last one) acquire chopsticks in reverse order compared to all others.

Code for Philosophers 0-4:

```
loop
    Think()
    down(chopstick[i]);           // Acquire left chopstick FIRST
    down(chopstick[(i+1) mod N]); // Acquire right chopstick SECOND
    Eat()
    up(chopstick[i]);             // Release left chopstick
    up(chopstick[(i+1) mod N]);   // Release right chopstick
end loop
```

Code for Philosopher 5:

```
loop
    Think()
    down(chopstick[(5+1) mod N]); // Acquire right chopstick (c[0]) FIRST
    down(chopstick[5]);           // Acquire left chopstick (c[5]) SECOND
    Eat()
    up(chopstick[5]);             // Release left chopstick
    up(chopstick[(5+1) mod N]);   // Release right chopstick
end loop
```

We now prove that this strategy prevents deadlock:

Proof. We will prove that if any philosopher tries to pick up a chopstick, eventually some philosopher will eat.

Assume, for contradiction, that philosopher i attempts to pick up a chopstick, but no philosopher ever eats. This implies that no philosopher can acquire both chopsticks needed for eating.

Consider philosopher i where $i = 0, 1, 2, 3, 4$:

Case 1: Philosopher i tries to take chopstick i (left), but fails.

- This implies philosopher $i - 1$ took chopstick i (as its right chopstick)
- For this to happen, philosopher $i - 1$ must have already acquired its left chopstick
- This means philosopher $i - 1$ has both chopsticks and is eating, contradicting our assumption

Case 2: Philosopher i acquires chopstick i (left) and waits for chopstick $i + 1$ (right)

- If i cannot take chopstick $i + 1$, it means philosopher $i + 1$ has taken it (as its left chopstick)

- Following this pattern, philosopher $i+2$ must have taken chopstick $i+2$ and waits for chopstick $i+3$
- This chain continues until philosopher 5
- Since philosopher 5 follows the reversed strategy (taking right chopstick first), this chain breaks
- Either philosopher 5 will eat, or some philosopher in the chain will eat

For philosopher 5, the analysis follows similar logic, always leading to a contradiction of our assumption.

Therefore, if any philosopher attempts to pick up a chopstick, eventually some philosopher will eat, proving deadlock-freedom. □

Solution 2: Odd/Even Strategy

This approach prevents deadlock by dividing philosophers into two groups with different chopstick acquisition orders.

Concept

Philosophers are categorized based on their index:

- **Odd-Indexed Philosophers** (1, 3, 5): Use one acquisition order.
- **Even-Indexed Philosophers** (0, 2, 4): Use the opposite acquisition order.

This ensures that adjacent philosophers never compete for the same chopstick simultaneously in the same order, preventing circular wait conditions.

Code for Odd Philosophers ($i = 1, 3, 5, \ldots$):

```
loop
    Think()
    down(chopstick[(i+1) mod N]); // Acquire RIGHT chopstick FIRST
    down(chopstick[i]);           // Acquire LEFT chopstick SECOND
    Eat()
    up(chopstick[(i+1) mod N]);   // Release right chopstick
    up(chopstick[i]);             // Release left chopstick
end loop
```

Code for Even Philosophers ($i = 0, 2, 4, \ldots$):

```
loop
    Think()
    down(chopstick[i]);           // Acquire LEFT chopstick FIRST
    down(chopstick[(i+1) mod N]); // Acquire RIGHT chopstick SECOND
```

```
    Eat()
    up(chopstick[(i+1) mod N]);   // Release right chopstick
    up(chopstick[i]);             // Release left chopstick
end loop
```

This approach not only prevents deadlock but also allows multiple philosophers to eat concurrently, improving system throughput.

Deadlock: Necessary Conditions (Coffman Conditions)

For a deadlock to occur in a system, four conditions must simultaneously hold:

1. **Mutual Exclusion:** At least one resource must be held in a non-shareable mode, where only one process can use it at a time. When another process requests that resource, it must wait until the resource is released.
2. **Hold and Wait:** A process must be holding at least one resource while simultaneously waiting to acquire additional resources currently held by other processes.
3. **Non-Preemption:** Resources cannot be forcibly taken away; they can only be released voluntarily by the process holding them after that process has completed its task.
4. **Circular Wait:** A closed chain of processes must exist such that each process holds resources needed by the next process in the chain, creating a cycle of waiting dependencies.

The first and third conditions relate to fundamental resource allocation policies; the second condition represents typical process behavior; the fourth condition is the critical cycle that often arises from race conditions in concurrent systems.

Deadlock Prevention

Each prevention strategy targets one of the necessary conditions:

- **Preventing Mutual Exclusion:**
  - Not always feasible, as many resources are inherently non-shareable.
  - Some resources can be virtualized or made shareable through special mechanisms.
- **Preventing Hold and Wait:**
  - Require processes to request all needed resources simultaneously before execution begins.
  - If all requested resources are available, grant them; otherwise, the process must wait without holding any resources.

- Drawback: Reduces concurrency and resource utilization, as processes may hold resources for longer than necessary.

- **Allowing Preemption:**
  - Enable the operating system to reclaim resources from processes when needed.
  - If a process requests a resource that cannot be immediately allocated, all resources it currently holds are released.
  - Works well for resources whose state can be easily saved and restored.

- **Preventing Circular Wait:**
  - Impose a total ordering on all resource types.
  - Require processes to request resources in strictly increasing order of enumeration.
  - For example, if Lock A is #5 and Lock B is #10, processes must acquire A before B.
  - This approach prevents cycles in the resource allocation graph, as used in the Dining Philosophers solutions.

Deadlock Avoidance

This approach allows the necessary conditions to exist but ensures the system never enters an "unsafe" state from which deadlock might eventually occur. It requires advance knowledge of resource needs.

- **Banker's Algorithm (Dijkstra):**
  - **State Representation:**
    - A: Vector of available resources of each type.
    - M_p: Maximum potential demand of each process p for each resource type.
    - C_p: Current allocation to each process p.

    For example, in a system with 3 resource types, $A = (3, 0, 1)$ means 3 units of resource type 1, 0 of type 2, and 1 of type 3 are available.
  - **Safe State Concept:** A state is safe if there exists a sequence $< P_1, P_2, ..., P_n >$ such that each process P_i can have its maximum resource needs satisfied using currently available resources plus resources held by all processes P_j with $j < i$. If such a sequence exists, the system can complete all processes without deadlock.
  - **Algorithm Implementation:**

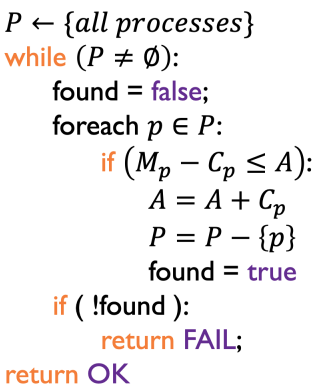

Figure 2: Algorithm for Determining Safe State

- **Operational Logic:**
  - The algorithm initializes set P to contain all processes.
  - It iteratively searches for a process whose remaining resource needs $(M_p - C_p)$ can be satisfied with available resources A.
  - Upon finding such a process, it simulates its completion by:
    - Adding its held resources back to the available pool: $A = A + C_p$
    - Removing it from the set under consideration: $P = P - \{p\}$
    - Setting a flag indicating progress: $found = true$
  - If an iteration finds no suitable process ($found$ remains $false$), the algorithm returns $FAIL$, indicating an unsafe state.
  - If all processes are eventually accommodated (set P becomes empty), the algorithm returns OK, confirming a safe state.
  - This approach effectively determines whether a sequence exists where all processes can complete without deadlock.

Deadlock Detection and Recovery

This approach allows deadlocks to occur but implements mechanisms to detect and resolve them:

- **Detection:** Periodically construct a wait-for graph and search for cycles, which indicate deadlock.
- **Recovery:** Once deadlock is detected, resolve it through:
  - **Process Termination:** Abort one or more processes involved in the deadlock cycle.
  - **Resource Preemption:** Temporarily take resources from processes and give them to others.

Ostrich Algorithm

The most pragmatic approach used in many general-purpose operating systems is to simply ignore the possibility of deadlock. This strategy:

- Assumes deadlocks are rare enough that prevention/avoidance/detection overhead exceeds the cost of occasional manual intervention.
- Relies on system administrators to handle deadlocks when they occur (often through system restart).
- Places responsibility on application developers to design deadlock-free programs.
- Is surprisingly effective in practice for many systems where deadlocks are infrequent.

The Producer-Consumer Problem

This canonical synchronization problem illustrates the challenges of coordinating data exchange between concurrent processes.

Problem Definition and Challenges

- **Scenario:** One or more **Producer** threads generate data items and place them into a shared buffer of fixed size N. One or more **Consumer** threads retrieve these items and process them. The buffer is typically implemented as a circular structure (ring buffer), with indices `IN` (next insertion position) and `OUT` (next retrieval position).
- **Synchronization Challenges:**
    1. **Race Conditions on Buffer Access:** When multiple producers or consumers simultaneously access the buffer and modify indices, operations can interfere, resulting in data corruption. This necessitates mutual exclusion for buffer operations.
    2. **Buffer Full Condition:** Producers must not add items to a full buffer. When a producer encounters a full buffer, it must wait until a consumer removes at least one item.
    3. **Buffer Empty Condition:** Consumers must not attempt to remove items from an empty buffer. When a consumer encounters an empty buffer, it must wait until a producer adds at least one item.

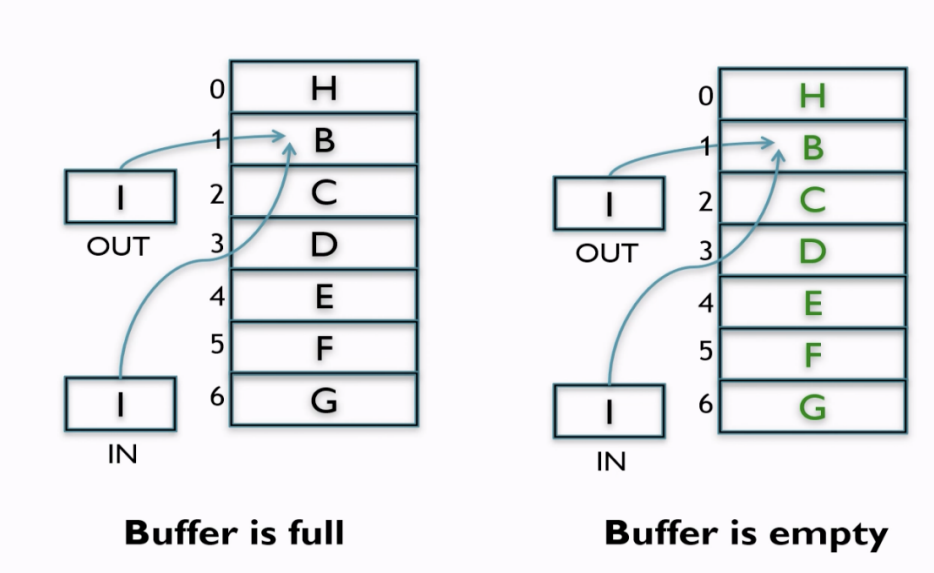

Figure 3: Cyclic Buffer States

When confronted with identical `IN` and `OUT` values as shown in the image above, the system must distinguish between completely full and completely empty states. This ambiguity represents a fundamental challenge in buffer management.

To resolve this ambiguity, we can introduce a counter variable that tracks the number of items:

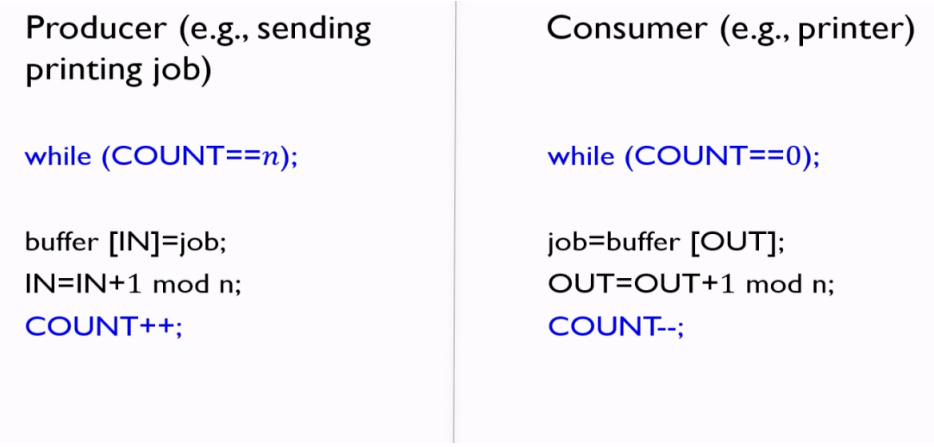

Figure 4: Single Producer, Single Consumer Implementation

The implementation must ensure that operations on the counter are atomic to maintain correctness.

For systems with multiple producers and consumers, additional synchronization mechanisms are required to prevent job duplication or loss when concurrent threads manipulate shared buffer state.

Semaphore-Based Solution

This elegant solution uses three semaphores to manage buffer access and capacity constraints without busy waiting. The implementation assumes a buffer of size n.

Semaphore Initialization:

- `mutex`: Initialized to 1 (provides mutual exclusion for buffer operations).
- `empty`: Initialized to n (counts available buffer slots).
- `full`: Initialized to 0 (counts occupied buffer slots).

Producer Code:

```
1: Produce item (outside critical section)
2: Down(empty)                    ▷ Wait if buffer full; decrement available slots
3: Down(mutex)                    ▷ Acquire exclusive buffer access
4: // Critical Section:
5: buffer[IN] = item
6: IN = (IN + 1) mod n
7: // End Critical Section
8: Up(mutex)                      ▷ Release buffer access
9: Up(full)                       ▷ Increment filled slots; signal consumers
```

Consumer Code:

```
1: Down(full)                     ▷ Wait if buffer empty; decrement filled slots
2: Down(mutex)                    ▷ Acquire exclusive buffer access
3: // Critical Section:
4: item = buffer[OUT]
5: OUT = (OUT + 1) mod n
6: // End Critical Section
7: Up(mutex)                      ▷ Release buffer access
8: Up(empty)                      ▷ Increment available slots; signal producers
9: Consume item (outside critical section)
```

This implementation ensures mutual exclusion for buffer operations through `mutex`, prevents buffer overflow via `empty`, and prevents buffer underflow via `full`. The solution efficiently uses blocking waits rather than wasteful busy waiting.

Monitors

Monitors provide a higher-level synchronization abstraction supported by some programming languages, combining data encapsulation with synchronization mechanisms.

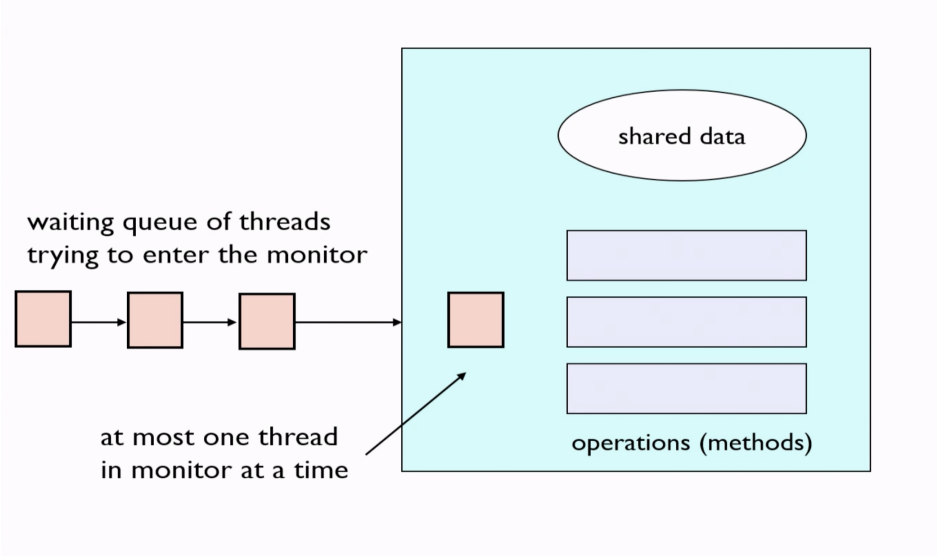

Figure 5: Monitor Structure

- **Structure:** A monitor encapsulates:
  - Shared data structures (variables, buffers, etc.).
  - Operations (procedures/methods) that manipulate this data.
  - Built-in synchronization mechanisms that coordinate access.
- **Key Feature: Implicit Mutual Exclusion:**
  - Only one thread can execute code within the monitor at any time.
  - If a thread calls a monitor procedure while another thread is active inside the monitor, the calling thread automatically blocks until the monitor becomes available.
  - This implicit mutual exclusion eliminates the need for explicit lock acquisition and release.
- **Advantages:**
  - Simplifies synchronization by handling locking automatically.
  - Reduces programmer error by enforcing structured access patterns.
  - Encapsulates both data and synchronization logic together.
- **Limitation:**
  - If a thread inside the monitor needs to wait for a specific condition (e.g., a consumer finding an empty buffer), it cannot simply block while holding the monitor lock, as this would prevent other threads from entering to change the condition.
  - This limitation is addressed through condition variables.

Condition Variables (Rendezvous Points)

Condition variables extend monitors by providing a mechanism for threads to temporarily relinquish the monitor lock while waiting for specific conditions to be satisfied.

- **Purpose:** They serve as designated waiting points associated with specific conditions that threads may need to wait for.
- **Essential Component:** Without condition variables, monitors would be impractical for many synchronization scenarios that require conditional waiting.
- **Broader Application:** Even in systems that don't implement full monitor constructs, condition variables are often provided as standalone synchronization primitives due to their utility.
- **Implementation:** Each condition variable typically maintains a queue of threads that are currently waiting for that particular condition to be signaled.

Operations on Condition Variables

Three fundamental operations define the behavior of a condition variable `c`:

- `wait(c)` — (Java equivalent: `c.wait()`)
  - **Monitor Lock Release:** The calling thread automatically releases the exclusive monitor lock, allowing other threads to enter.
  - **Thread Suspension:** The thread is placed in a suspended state on the wait queue associated with condition variable `c`.
  - **Atomicity:** The lock release and thread suspension occur as a single atomic operation to prevent race conditions.
- `signal(c)` — (Java equivalent: `c.notify()`)
  - **Wake Operation:** Activates at most one thread currently waiting on condition variable `c`.
  - **No Persistence:** If no threads are waiting on `c`, the signal has no effect and is not remembered for future waiters.
  - **Contrast with Semaphores:** Unlike semaphore `Up`/`V` operations that always increment the counter, signals without waiters are lost.
- `broadcast(c)` — (Java equivalent: `c.notifyAll()`)
  - **Mass Activation:** Wakes up all threads currently waiting on condition variable `c`.
  - **Use Cases:** Appropriate when multiple waiting threads may be able to proceed once a condition changes, or when the signaling thread cannot determine which specific waiter should be prioritized.

Using condition variables, a thread inside a monitor can temporarily suspend its execution and release the monitor lock by calling `wait(c)`. This allows other threads to enter the monitor, potentially modifying the shared state. When conditions change, these threads can signal or broadcast to the waiting threads, allowing them to resume execution when the monitor becomes available again.

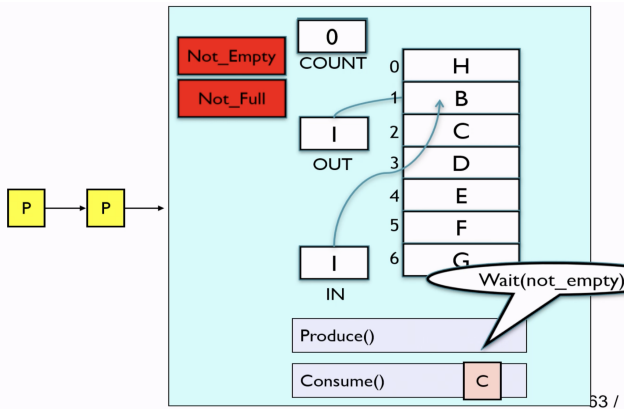

Figure 6: Consumer Entering Monitor

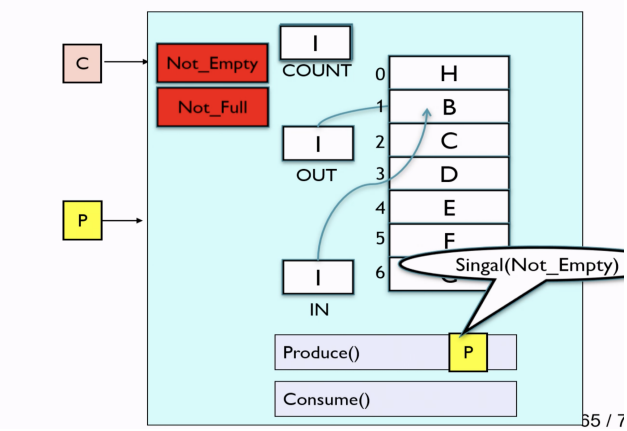

Figure 7: Consumer Entering Wait Queue

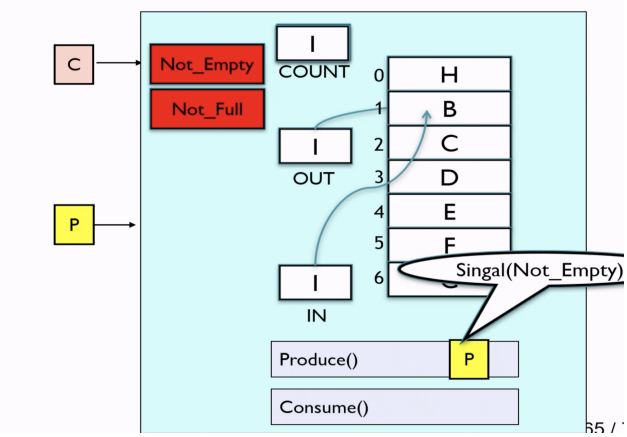

Figure 8: Producer Entering Monitor

Synchronization with Locks

Locks provide fundamental synchronization primitives for controlling access to shared resources. They come in various forms with different granularity levels and specialized implementations to address diverse synchronization requirements.

Focus on Shared Resources

Locks can be applied at different levels of granularity to balance protection with concurrency:

- **Multiple locks for different resources:** Enables fine-grained control, improving concurrent access.
- **Granularity spectrum:**
  - **Coarse-grained:** A single, general-purpose lock protecting an entire subsystem (e.g., file system lock).
  - **Medium-grained:** Separate locks for major components (e.g., directory-level locks).
  - **Fine-grained:** Distinct locks for individual elements (e.g., per-file or per-record locks).

Reader-Writer Locks

Reader-writer locks optimize concurrent access patterns by distinguishing between read operations (which don't modify data) and write operations (which do):

- **Observation:** Multiple concurrent readers can safely access data simultaneously without interference.
- **Dual-mode locking:**

  - **Read mode:** Allows other concurrent readers but excludes writers.
  - **Write mode:** Provides exclusive access, blocking both readers and other writers.

- **Access rules:**
  - Multiple simultaneous read locks can coexist.
  - A write lock excludes all other locks (both read and write).
  - Readers and writers are mutually exclusive.

- **Implementation (Read-Preferring):**

```
// Initialize
b = 0       // Integer counter tracking active readers
Init(r, 1)  // Reader mutex for protecting counter operations
Init(g, 1)  // Global mutex controlling resource access

// Begin Read
Down(r)     // Acquire reader counter mutex
++b         // Register this reader
If (b = 1) then Down(g)  // First reader blocks writers
Up(r)       // Release reader counter mutex

// End Read
Down(r)     // Acquire reader counter mutex
--b         // Deregister this reader
If (b = 0) then Up(g)  // Last reader allows writers to proceed
Up(r)       // Release reader counter mutex

// Begin Write
Down(g)     // Obtain exclusive access

// End Write
Up(g)       // Relinquish exclusive access
```

- **Reader preference characteristic:** This implementation prioritizes readers over writers. If readers continuously arrive while others are active, they can acquire access without interruption, potentially leading to writer starvation under high read loads.

Example: The Ready Queue

A practical example illustrating lock usage in operating systems:

- **Multiple system components interact with the ready queue:**
  - **Scheduler:** Removes processes from the queue for execution.

  - **Disk interrupt handler:** Adds previously blocked processes that are now ready.
  - **Process creation functions:** Add newly created process control blocks.

- **Synchronization requirement:** All operations must lock the ready queue to maintain its integrity during modifications.

Locks: Pitfalls

Several challenging scenarios can arise when using locks:

- **Priority Inversion:** A scheduling anomaly where effective priorities are reversed:
  - Thread 1 (low priority) acquires a critical lock.
  - Thread 2 (high priority) attempts to acquire the same lock and blocks.
  - Thread 3 (medium priority) preempts Thread 1, delaying its progress.
  - Result: The high-priority thread waits for a medium-priority thread, inverting the expected scheduling order.
- **Solution:** Priority inheritance protocols temporarily elevate the priority of lock-holding threads to the highest priority among waiting threads.
- **Deadlocks:** Circular wait conditions where each thread holds resources needed by others.
  - These must be addressed through prevention, avoidance, detection, or recovery techniques as discussed in previous sections.