CS3701 - Operating Systems

Chapter 5 - Synchronization

Department of Computer Science
College of Computer Engineering and Science
Prince Sattam bin Abdulaziz University

AY - 2025/2026

Prepared by Dr. Mahdi Khemakhem, Reviewed by Dr. Essra Aldessouky
Updated on January 22, 2026

Outline

1. Process Synchronization
1.1 Motivation
1.2 Producer-Consumer Problem
1.3 Race Condition
1.4 Key Takeaways

2. Critical-Section Problem
2.1 Problem Definition
2.2 Solution Requirements
2.3 Key Takeaways

3. Mutex Locks
3.1 Basic Concept
3.2 Busy Waiting and Spinlocks
3.3 Key Takeaways

4. Semaphores
4.1 Semaphore Definition
4.2 Types of Semaphores
4.3 Implementation without Busy Waiting
4.4 Problems with Semaphores
4.5 Key Takeaways

Outline (Contd.)

5. Classical Synchronization Problems
5.1 Bounded-Buffer Problem
5.2 Readers-Writers Problem
5.3 Dining-Philosophers Problem
5.4 Key Takeaways

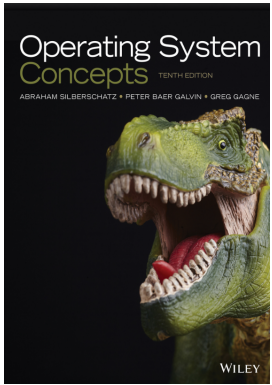
6. Exercices

Textbook Reference

Required Reading

Refer to Chapters 6 and 7 of the textbook:

- **Title:** *Operating System Concepts*
- **Authors:** Abraham Silberschatz, Peter Baer Galvin, and Greg Gagne
- **Publisher:** John Wiley & Sons
- **Edition:** 10th Edition (2018)



Outline

1. Process Synchronization
1.1 Motivation
1.2 Producer-Consumer Problem
1.3 Race Condition
1.4 Key Takeaways

2. Critical-Section Problem

3. Mutex Locks

4. Semaphores

5. Classical Synchronization Problems

6. Exercices

Process Synchronization: Motivation

Why do we need process synchronization?

- **Cooperating processes**: Multiple processes share data and resources
- Without proper coordination: **Data inconsistency** and errors
- **Goal: Ensure data integrity** and **maintain consistency**
- **Challenge:** Concurrent access to shared resources must be controlled

Impact of Missing Synchronization

Consider a bank account with balance = $1000: Transaction 1: Withdraw $500, Transaction 2: Withdraw $300

Without synchronization:

- Both read balance = $1000
- Both process withdrawal independently
- **Expected final balance: $200 → Final balance could be $500 or $700** (incorrect!)
  → **Proper synchronization is critical for correct execution!**

Concurrent Process Execution

Cooperating Processes

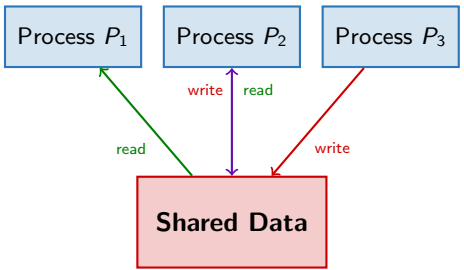
Processes that can affect or be affected by other processes executing in the system.
They **share data** and must **coordinate their actions**.

Types of Shared Resources

- **Memory**: Variables, arrays, data structures
- **Files**: Shared file systems
- **Devices**: Printers, displays
- **System Resources**: Semaphores, locks

Execution Models

- **True parallelism**: Multicore systems
- **Interleaved**: Single-core time-sharing



Race Condition Risk!

Benefits and Challenges of Cooperation

Why Allow Process Cooperation?

Cooperating processes provide significant advantages despite synchronization challenges.

Benefits of Cooperation

- **Information sharing**: Exchange data between processes
- **Computation speedup**: Parallel execution on multicore
- **Modularity**: Divide system into separate processes
- **Convenience**: Multiple tasks running concurrently

Problems Without Synchronization

- **Data Inconsistency**
  - Non-deterministic results
  - Final values depend on timing
- **Lost Updates**
  - Changes overwritten
  - Data corruption

Real-World Example of Synchronization Issues

Real-World Example

Online Shopping Cart:

- Two users (husband and wife) access same account simultaneously using different devices
- **Item stock = 1, both try to purchase**
- **Without sync: Both see available, both purchase → overselling!**
- **With sync: One succeeds, other sees "out of stock" → correct!**

Producer-Consumer Problem: Definition

Classic Synchronization Problem (Producer-Consumer)

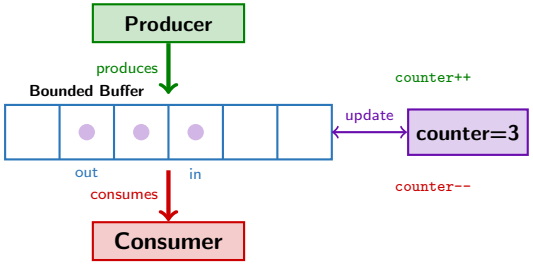
A producer process produces data items and places them in a buffer. A consumer process removes items from the buffer and consumes them. The buffer has a **fixed size**, requiring synchronization to prevent **overflows** and **underflows**.

Problem Components

- **Producer**: Creates items, adds to buffer
- **Consumer**: Removes items, consumes them
- **Bounded Buffer**: Fixed-size shared memory
- **Counter**: Tracks number of items

Synchronization Requirements

- **Producer** must **wait** if buffer is **full**
- **Consumer** must **wait** if buffer is **empty**
- Both must **coordinate** access to `counter`



Shared counter is the critical resource!

Producer-Consumer Problem: Implementation

Naive Implementation (Without Proper Synchronization)

Shared Data Structure

```
1  public class SharedBuffer {
2    static final int BUFFER_SIZE = 6;
3    static int[] buffer = new int[BUFFER_SIZE];
4    static int in = 0, out = 0;
5    static int counter = 0;  // items in buffer
6  }
```

Producer Process (Thread)

```
1  while (true) { // infinite loop
2    // produce next item
3    while (counter == BUFFER_SIZE)
4      ; // do nothing, buffer full, loop and
           wait
5    buffer[in] = item; // add item at pos. in
6    in = (in + 1) % BUFFER_SIZE; // circ.
         increment
7    counter++;  // increment count
8  }
```

Consumer Process (Thread)

```
1  while (true) { // infinite loop
2    while (counter == 0)
3      ; // do nothing, buffer empty, loop and wait
4    item = buffer[out]; // remove item from pos. out
5    out = (out + 1) % BUFFER_SIZE; // circ. increment
6    counter--;  // decrement count
7    // consume item
8  }
```

Key Issues:

- `counter` is a *shared variable* accessed by both producer and consumer
- Both processes **concurrently modify** `counter`
- Operations `counter++` and `counter--` appear atomic[a] but are **NOT atomic** at machine level
- Each operation translates to multiple assembly instructions (*load, modify, store*)
- *Race condition* occurs when operations interleave unpredictably

[a] Atomic operation runs completely without interruption.

Race Condition

Race Condition

A situation where several processes access and manipulate shared data concurrently, and the outcome depends on the particular order in which the access takes place.

Why does this happen?

High-level Java code appears atomic

```
1 counter++;  // looks simple
2 counter--;  // looks simple
```

Bytecode-level `counter++`

```
1 register1 = counter // load counter to register1
2 register1 = register1 + 1 // add 1 to register1
3 counter = register1 // store register1 to counter
```

Bytecode-level `counter--`

```
1 register2 = counter //  load counter to register2
2 register2 = register2 - 1 // sub 1 from register2
3 counter = register2 // store register2 to counter
```

Interleaved Execution

Initial: `counter = 5`

Possible Outcomes:

- **Correct**: $5 + 1 - 1 = 5$
- **Incorrect**: Final $= 4$
  - Both read 5
  - ++ writes 6
  - -- writes 4
  - Last write wins: 4
- **Incorrect**: Final $= 6$
  - Both read 5
  - -- writes 4
  - ++ writes 6
  - Last write wins: 6

Non-deterministic behavior!

Race Condition: Detailed Timing Analysis

Step-by-step Execution Interleaving

Time	Operation	counter
T0	Producer: register1 = counter	5
T1	Producer: register1 = register1 + 1	5
T2	Consumer: register2 = counter	5
T3	Consumer: register2 = register2 - 1	5
T4	Producer: counter = register1	**6**
T5	Consumer: counter = register2	**4**

Analysis

- Initial `counter = 5`
- Expected final: **5**
- Actual final: **4**
- **Producer's update lost!**

Why This Happened?

1. Both processes read `counter = 5`
2. Both compute locally:
   - **Producer:** $5 + 1 = 6$
   - **Consumer:** $5 - 1 = 4$
3. Both write back
4. **Last write wins!**

The Core Problem

- Operations are **not atomic**
- **Context switches** occur mid-operation
- No **mutual exclusion**

Key Takeaways

Key Takeaways

- **Process synchronization** coordinates access to shared resources among cooperating processes
- **Producer-consumer problem** demonstrates bounded buffer synchronization challenges
- **Race conditions** occur when concurrent processes access shared data with timing-dependent outcomes
- **Solutions require** three properties: **mutual exclusion**, **progress**, and **bounded waiting**

Outline

1. Process Synchronization

2. Critical-Section Problem
2.1 Problem Definition
2.2 Solution Requirements
2.3 Key Takeaways

3. Mutex Locks

4. Semaphores

5. Classical Synchronization Problems

6. Exercices

Critical-Section Problem: Definition

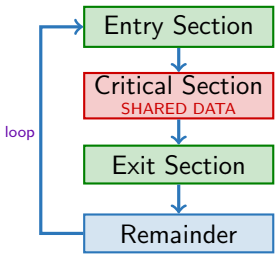
Critical Section

A segment of code where a process accesses shared resources that must not be concurrently accessed by other processes. It involves changing variables, updating tables, writing files, etc.

General Process (Thread) Structure

```
1 do {
2   entry_section;
3   critical_section;
4   exit_section;
5   remainder_section;
6 } while (true);
```

⇒ **Many processes (Threads) can execute this code concurrently.**



Only one process at a time can be in the critical section!

Section Descriptions:

- **Entry:** Request permission to enter
- **Critical:** Access shared resources
- **Exit:** Release permission
- **Remainder:** Non-critical code

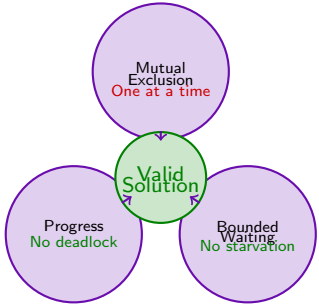
Critical-Section Solution Requirements

Three Essential Requirements

Any solution to the critical-section problem must satisfy three fundamental requirements to ensure correct and fair synchronization.

The Three Requirements

1. **Mutual Exclusion**
   - If P_i is in critical section, **no other process** $P_j, j \neq i$ can be in theirs
   - **Only one process** at a time accesses shared resources
   - **Prevents race conditions**
2. **Progress**
   - If no process is in critical section and some want to enter, **selection cannot be postponed**
   - Only processes **not in remainder** section decide
   - **Prevents deadlock** (see Chapter 6)
3. **Bounded Waiting**
   - There exists a **bound on the number of times** other processes can enter their critical sections **after a process has requested entry** and before that request is granted
   - This requirement **prevents starvation** by ensuring every process eventually gains access



⇒ **Missing any requirement = Invalid solution**

Key Takeaways

Key Takeaways

- **Critical section** is code segment accessing shared resources
- **Critical-section problem** requires synchronization to prevent concurrent access
- **Three requirements** for solutions: **mutual exclusion**, **progress**, **bounded waiting**

Outline

1. Process Synchronization

2. Critical-Section Problem

3. Mutex Locks
   3.1 Basic Concept
   3.2 Busy Waiting and Spinlocks
   3.3 Key Takeaways

4. Semaphores

5. Classical Synchronization Problems

6. Exercices

Mutex Locks: Basic Concept

Mutex Lock

A mutex (mutual exclusion) lock is the simplest synchronization tool — a boolean variable indicating lock availability.

Two Atomic Operations

acquire() — Request Lock:

```
acquire() {
  while (!available)
    ; // busy wait
  available = false;
}
```

release() — Release Lock:

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

⇒ **Both must execute atomically**

Using Mutex in Code

```
do {
  acquire();      // entry
    // critical section
  release();      // exit
    // remainder section
} while (true);
```

Key Properties:

- **Ensures mutual exclusion**
- **Simple to implement**
- **Uses busy waiting**

Busy Waiting and Spinlocks

Spinlock

A spinlock is a mutex lock that uses busy waiting — continuously looping while checking lock availability.

Disadvantages:

- **Wastes CPU cycles** while spinning
- **Inefficient** on single-processor systems
- **Poor** for long critical sections

Advantages:

- **No context switch** overhead
- **Lower latency** for short waits
- **Efficient on multicore** systems

When to Use Spinlocks?

GOOD
Short critical sections
Multicore systems
Low contention

BAD
Long critical sections
Single processor
High contention

Key Takeaways

Key Takeaways

- **Mutex locks** provide the simplest synchronization mechanism using boolean variables
- **Spinlocks** use busy waiting — trading CPU cycles for **no context switch** overhead
- **Best for multicore systems** with short critical sections and low contention (i.e., few competing processes)
- **Avoid on single processors** or for long critical sections due to wasted CPU cycles

Outline

1. Process Synchronization

2. Critical-Section Problem

3. Mutex Locks

4. Semaphores
4.1 Semaphore Definition
4.2 Types of Semaphores
4.3 Implementation without Busy Waiting
4.4 Problems with Semaphores
4.5 Key Takeaways

5. Classical Synchronization Problems

6. Exercices

Semaphores - Definition

Semaphore

A semaphore S is an integer variable accessed only through two atomic operations: `wait()` and `signal()`.

wait(S) — Request Resource

```
1 wait(S) {
2   while (S <= 0)
3     ; // busy wait
4   S--;
5 }
```

Also called `P()` or `down()`

signal(S) — Release Resource

```
1 signal(S) {
2   S++;
3 }
```

Also called `V()` or `up()`

⇒**Critical Property: All semaphore modifications must execute atomically — no interruption allowed!**

Key Insight: Semaphore value = number of available resources

Types of Semaphores

Two Types

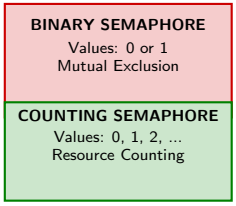
Semaphores come in two types based on their value range and use case.

1. Binary Semaphore

- **Range:** 0 or 1 only
- **Purpose:** Mutual exclusion
- **Similar to:** Mutex locks
- **Use case:** Protect critical sections

2. Counting Semaphore

- **Range:** Unrestricted integers
- **Purpose:** Resource counting
- **Initial value:** # of resources
- **Use case:** Manage multiple instances



⇒ **Choose type based on synchronization needs!**

⇒**Busy waiting exists in both types.**

Efficient Semaphore Implementation - Without Busy Waiting

No Busy Waiting

Improved implementation uses process blocking instead of busy waiting — no CPU cycles wasted!

Semaphore Structure

```
1  class Semaphore {
2    int value;
3    Queue<Process> list; // waiting queue
4  }
```

wait(S) — Block if needed

```
1  wait(Semaphore S) {
2    S.value--;
3    if (S.value < 0) {
4      S.list.add(currentP); // add this process to
           wait queue
5      block();  // block this process
6    }
7  }
```

signal(S) — Wake up process

```
1  signal(Semaphore S) {
2    S.value++;
3    if (S.value <= 0) {
4      Process P = S.list.remove(); // get process
           from wait queue
5      wakeup(P);  // resume process
6    }
7  }
```

Interpretation: `S.value < 0` $\Rightarrow$ `|S.value|` = number of waiting processes

Semaphore Pitfalls

Common Issues

Despite their power, semaphores can lead to serious problems if used incorrectly.

Three Major Problems

1. **Deadlock**
- **Circular wait for resources**
- **Example:** P_0 holds Q, waits for S; P_1 holds S, waits for Q

2. **Starvation**
- **Process never removed from queue**
- **Cause:** Poor queue management (e.g., LIFO)

3. **Priority Inversion**
- **Low-priority blocks high-priority**
- **Effect:** High-priority indirectly preempted

Programming Errors

CORRECT
wait(S)
critical section
signal(S)

WRONG
signal(S) + signal(S)
violates mutex

WRONG
wait(S) + wait(S)
causes deadlock

Key Takeaways

Key Takeaways

- **Semaphores** are integer variables with atomic `wait()` and `signal()` operations
- **Two types**: binary (mutual exclusion) and counting (resource management)
- **Efficient implementations** avoid busy waiting using queues and process blocking
- **Common pitfalls**: deadlock, starvation, priority inversion, and programming errors

Outline

1. Process Synchronization

2. Critical-Section Problem

3. Mutex Locks

4. Semaphores

5. Classical Synchronization Problems
   5.1 Bounded-Buffer Problem
   5.2 Readers-Writers Problem
   5.3 Dining-Philosophers Problem
   5.4 Key Takeaways

6. Exercices

Bounded-Buffer Problem

Problem Description:

- n **buffers**, each holding one item
- Semaphore `mutex` initialized to 1 (mutual exclusion), it protects access to the buffer
- Semaphore `full` initialized to 0 (count of full buffers), it tracks filled slots
- Semaphore `empty` initialized to n (count of empty buffers), it tracks empty slots

Producer

```
do {
  /* Produce item */
  wait(empty);
  wait(mutex);
  /* Add item to buffer */
  signal(mutex);
  signal(full);
} while (true);
```

Consumer

```
do {
  wait(full);
  wait(mutex);
  /* Remove item from buffer */
  signal(mutex);
  signal(empty);
  /* Consume item */
} while (true);
```

Readers-Writers Problem

Problem Description:

- **Database shared among concurrent processes**
- **Readers:** Only read data (**multiple readers allowed**)
- **Writers:** Read and write data (**exclusive access required**)

Shared Data:

```
semaphore rw_mutex = 1;//controls access to database
semaphore mutex = 1;//protects read_count
int read_count = 0;//number of active readers
```

Writer:

```
do {
  wait(rw_mutex); // acquire exclusive access
  /* writing */
  signal(rw_mutex); // release exclusive access
} while (true);
```

Reader:

```
do {
  wait(mutex); //protect read_count
  read_count++; //increment number of readers
  if (read_count == 1)
    wait(rw_mutex);//first reader locks writers
  signal(mutex); //release mutex
  /* reading */
  wait(mutex); //protect read_count
  read_count--;
  if (read_count == 0)
    signal(rw_mutex); //last reader unlocks writers
  signal(mutex); //release mutex
} while (true);
```

Readers-Writers Problem: Variations

Two Variations:

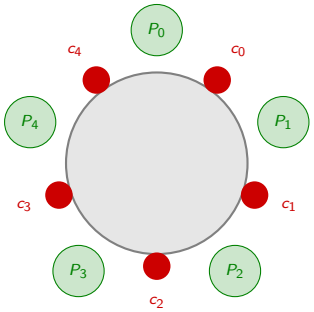
1. **First Readers-Writers Problem:** *(implemented in previous slide)*
   - No reader should wait unless a writer has already obtained permission
   - Readers have priority, i.e., new readers can keep coming in while writers wait
   - **Problem:** Writers may starve
2. **Second Readers-Writers Problem:** *(student exercise)*
   - Once a writer is ready, it performs write as soon as possible
   - Writers have priority, i.e., new readers wait if a writer is waiting
   - **Problem:** Readers may starve

⇒**Both solutions can lead to starvation of either readers or writers**
⇒**More complex solutions may provide reader-writer locks with fairness guarantees**, such as limiting consecutive reads or writes, i.e., alternating access.

Dining-Philosophers Problem

Problem Description:

- **5 philosophers (P_0, P_1, P_2, P_3, P_4) sitting at a round table**
- **5 chopsticks (c_0, c_1, c_2, c_3, c_4), one between each pair of philosophers**
- **Each philosopher alternates randomly between two states:**
  - **Thinking**
  - **Eating** (requires 2 adjacent chopsticks)
- **Each philosopher P_i needs chopstick left c_i and right $c_{(i+1) \mod 5}$ to eat**



$\Rightarrow$**Challenge: Design a protocol where philosophers will never starve. Avoid deadlock and ensure fairness.**

Dining-Philosophers: First Solution

Semaphore Solution

```
1 semaphore c[5] = {1, 1, 1, 1, 1}; // chopsticks availability
2 do {
3   /* Thinking */
4   wait(c[i]); // pick up left chopstick
5   wait(c[(i+1) % 5]); // pick up right chopstick
6   /* Eating */
7   signal(c[i]); // put down left chopstick
8   signal(c[(i+1) % 5]); // put down right chopstick
9 } while (true);
```

Key Insight: Each philosopher picks up left chopstick first, then right.

Problem: Deadlock!

If all philosophers pick up their left chopstick simultaneously, deadlock occurs!

Dining-Philosophers: Possible Remedies

Deadlock Prevention Strategies

- **Resource Hierarchy Solution:** Impose an order on resource acquisition (e.g., always pick lower-numbered chopstick first)
- **Asymmetric Resource Acquisition:** Odd-numbered philosophers pick up right chopstick first, even-numbered pick up left first
- **Randomized Backoff:** If unable to acquire both chopsticks, put down any held chopstick(s) and wait a random time before retrying

$\Rightarrow$**Each strategy has trade-offs in complexity, fairness, and performance.**

Key Takeaways

Key Takeaways

- **Bounded-buffer problem** uses semaphores to manage producer-consumer synchronization
- **Readers-writers problem** balances concurrent read access with exclusive write access
- **Dining-philosophers problem** illustrates challenges of circular resource allocation and deadlock prevention

Outline

1. Process Synchronization
2. Critical-Section Problem
3. Mutex Locks
4. Semaphores
5. Classical Synchronization Problems
6. Exercices

Exercise 1: Readers-Writers (Second Version)

Readers-Writers Problem - Version 2

Implement the second readers-writers problem using semaphores, ensuring that writers have priority over readers. Provide code and explain how your solution prevents starvation for writers while allowing readers to access the database when no writers are waiting.

Solution: Exercise 1

Solution Outline:

- Use a semaphore to track waiting writers
- Modify reader logic to check for waiting writers before accessing the database

Semaphore and Variables:

```
1 semaphore rw_mutex = 1; // controls access to database
2 semaphore mutex = 1; // protects read_count
3 semaphore write_wait = 1; // counts waiting writers
4 int read_count = 0; // number of active readers
```

Writer Code:

```
1 do {
2   wait(write_wait); // indicate a writer is waiting
3   wait(rw_mutex); // acquire exclusive access
4   signal(write_wait); // no longer waiting
5   /* writing */
6   signal(rw_mutex); // release exclusive access
7 } while (true);
```

Solution: Exercise 1 (Continued)

Reader Code:

```
do {
  wait(write_wait); // check for waiting writers
  signal(write_wait); // release immediately
  wait(mutex); // protect read_count
  read_count++; // increment number of readers
  if (read_count == 1)
    wait(rw_mutex); // first reader locks writers
  signal(mutex); // release mutex
  /* reading */
  wait(mutex); // protect read_count
  read_count--;
  if (read_count == 0)
    signal(rw_mutex); // last reader unlocks writers
  signal(mutex); // release mutex
} while (true);
```

Explanation:

- Writers increment `write_wait` when they want to write
- Readers check `write_wait` before reading; if writers are waiting, they wait
- This ensures writers get priority access, preventing starvation

Exercise 2: Dining Philosophers with Asymmetric Resource Acquisition

Dining Philosophers Problem - Asymmetric Resource Acquisition

Implement a solution to the dining philosophers problem using the asymmetric resource acquisition strategy, where odd-numbered philosophers pick up the right chopstick first and even-numbered philosophers pick up the left chopstick first. Provide code and explain how this approach prevents deadlock.

Solution: Exercise 2

Semaphore Initialization:

```
semaphore c[5] = {1, 1, 1, 1, 1}; // chopsticks availability
```

Philosopher Code:

```
do {
  /* Thinking */
  if (i % 2 == 0) { // even-numbered philosopher
    wait(c[i]); // pick up left chopstick
    wait(c[(i+1) % 5]); // pick up right chopstick
  } else { // odd-numbered philosopher
    wait(c[(i+1) % 5]); // pick up right chopstick
    wait(c[i]); // pick up left chopstick
  }
  /* Eating */
  signal(c[i]); // put down left chopstick
  signal(c[(i+1) % 5]); // put down right chopstick
} while (true);
```

End of Chapter 5