Larbi Ben M'hidi University – Oum El Bouaghi
Faculty of Exact Sciences and Nature and Life Sciences
Department of Mathematics and Computer Science
Module: Operating Systems II
Lecturer: Dr. Moustafa Sadek KAHIL
2025 – 2026

Chapter II: Process Synchronization

II.1. Race Conditions

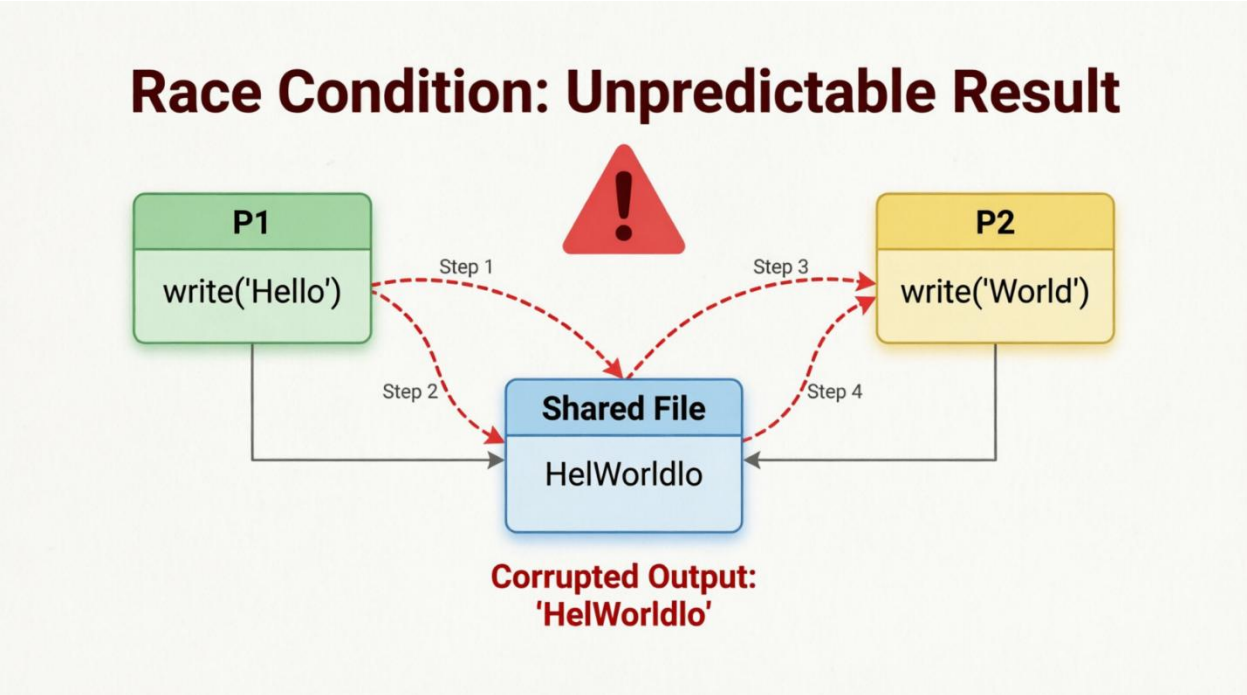
In a multiprogrammed system, multiple processes (or threads) execute concurrently while sharing available resources, such as the processor and memory. However, this concurrency can lead to data consistency problems when multiple processes simultaneously access the same resource (for reading or writing) without synchronization mechanisms. This phenomenon is known as a race condition.

Example 1:

Imagine two processes (P1 and P2) simultaneously accessing a file to perform operations:

- P1 wants to write "Hello" to the file.
- P2 wants to write "World" to the same file.

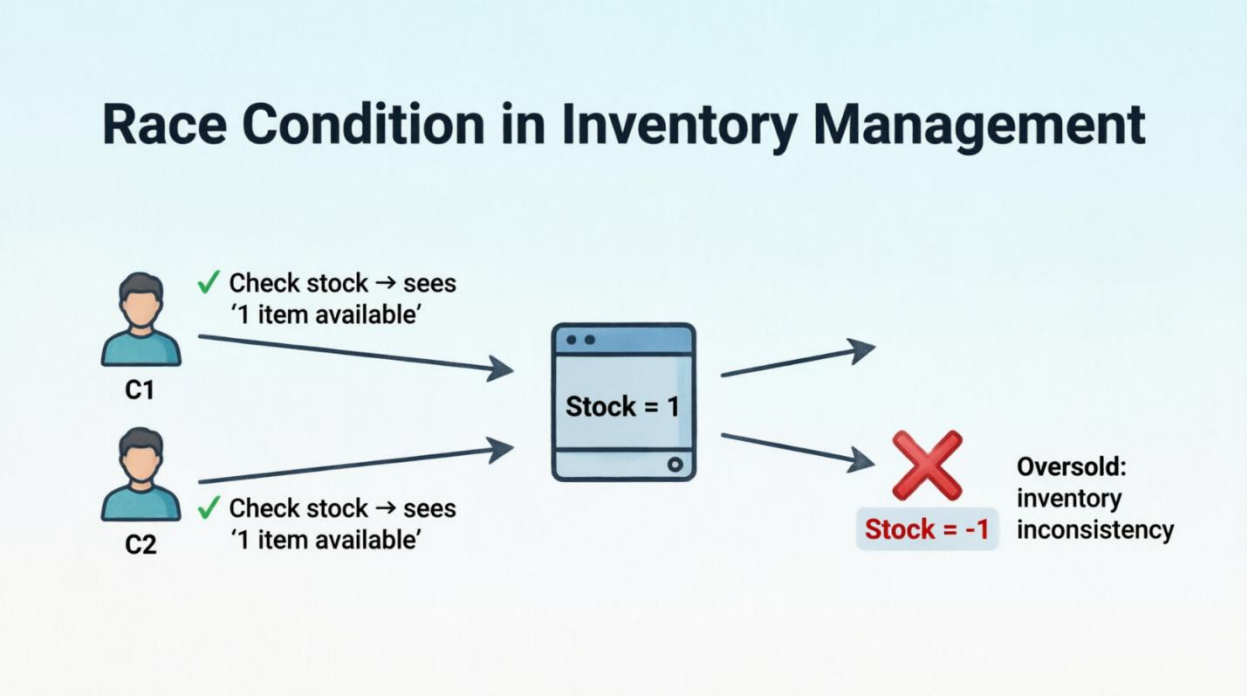
Without synchronization, the final content of the file could be inconsistent, such as "HelWorldlo" or "WorHelllod", depending on the execution order of the processes.



Example 2:

In an e-commerce system:

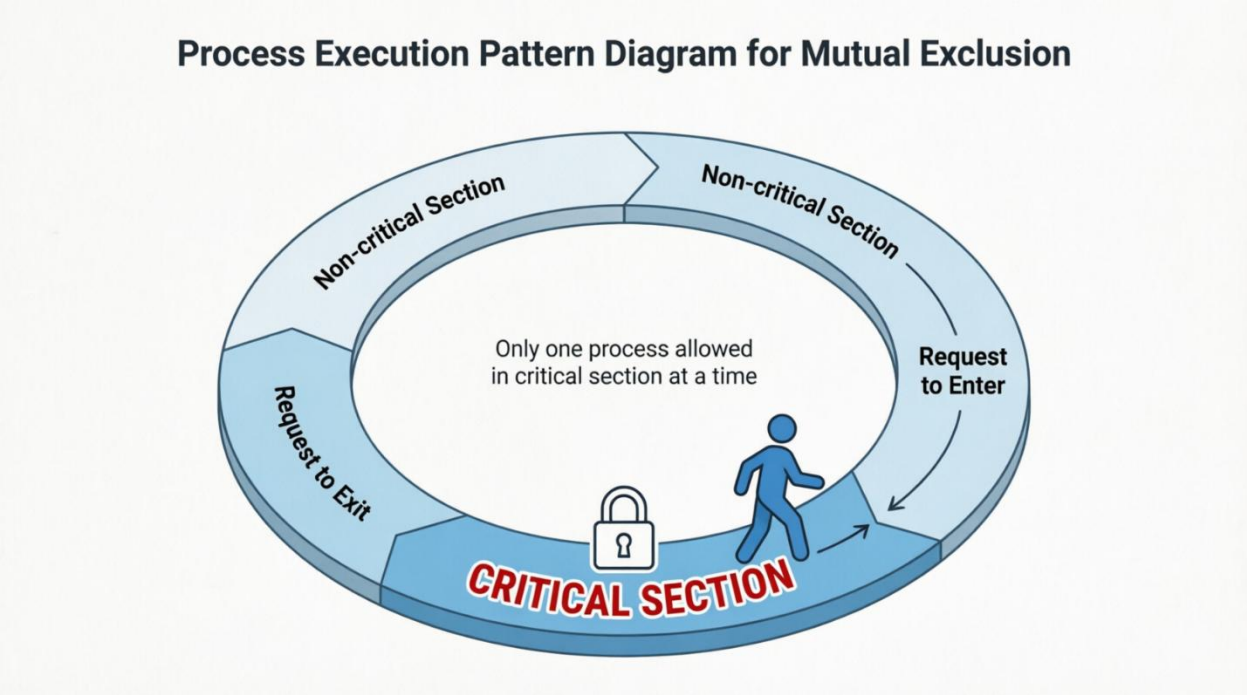
- Two customers (C1 and C2) simultaneously purchase the last item in stock (initial quantity: 1).
- C1 checks the stock, sees that 1 item remains, and places an order.
- Meanwhile, C2 makes the same check and also places an order. Without synchronization, both orders would be validated, even though only one item is available.



A **critical section** is a part of a program's code where a process (or thread) accesses a shared resource (such as a variable, a file, or a database) that may be modified. To avoid inconsistent results due to simultaneous access by multiple processes, only one entity must be able to execute the critical section at any given time.

In a concurrent environment, each process operates according to the following pattern:

```
Non-critical section
Request to enter critical section
Critical section
Request to exit critical section
Non-critical section
```



II.2. Mutual Exclusion Problem

The mutual exclusion problem is a fundamental challenge in the design of operating systems, particularly in multitasking or multiprocessor environments. It occurs when multiple processes or threads must access a shared resource (e.g., a variable, a file, a database) and it is essential that only one process or thread can access the resource at a given moment.

Mutual exclusion is a mechanism that guarantees that only one process or thread can access a shared resource at a time (for reading or writing). It prevents multiple processes from interfering with each other when each enters its critical section.

Mutual exclusion aims to ensure:

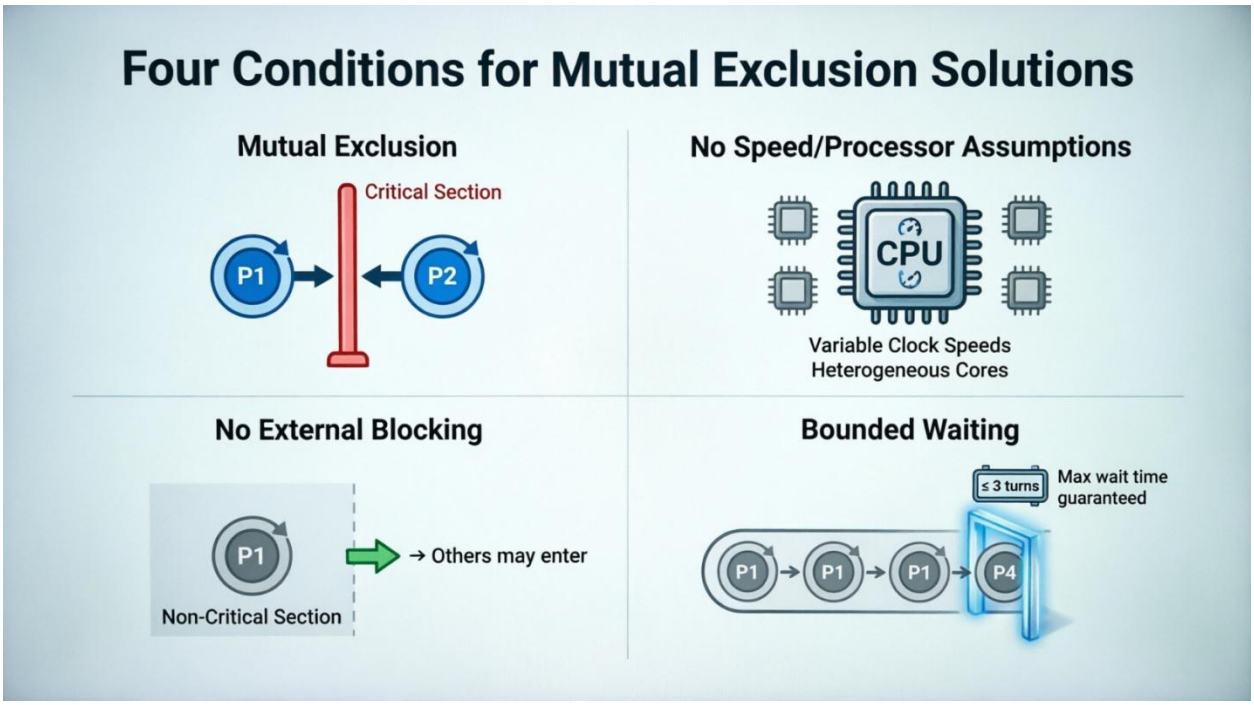
- **Security:** Data is not corrupted by simultaneous accesses.
- **Fairness:** All processes have a chance to access the resources.
- **Performance:** Resources are used efficiently without unnecessary blocking.

To achieve this, any proposed solution must satisfy the following four conditions:

- Two processes cannot be in their critical sections at the same time.
- No assumptions may be made about the relative speeds of processes or the number of processors.
- No process outside its critical section may block other processes from entering their critical sections.
- No process should wait too long before entering its critical section (bounded waiting).

The table below explains each of these conditions in detail.

Condition	Explanation	Objective	Example
Two processes cannot be in their critical sections at the same time	A process accessing a shared resource blocks others. Prevents interferences and data inconsistencies. Ensures one process completes its operations before another accesses it.	Guarantee data integrity.	Two users performing banking transactions on the same account (simultaneous withdrawal and deposit). Without exclusion, the final balance would be inconsistent.
No assumptions about process speeds or number of processors	Works independently of process speed. Compatible with uniprocessor and multiprocessor systems. Does not depend on hardware or process priorities.	Ensure portability and robustness in all environments.	A fast process should not have priority access to a resource over a slow process. Mechanisms must be fair.
No process outside its critical section may block others	Processes outside their critical section must not interfere. Prevents unnecessary blocking.	Avoid unnecessary blocking and improve efficiency.	If a process is downloading data but not accessing the shared resource, it must not prevent other processes from using it.
No process should wait too long before entering its critical section	All processes must access the resource within a reasonable time. Prevents starvation. Guarantees fair waiting among processes.	Prevent starvation and guarantee fairness.	In an online ticketing system, each user must have a fair chance to buy tickets, even under high simultaneous demand.

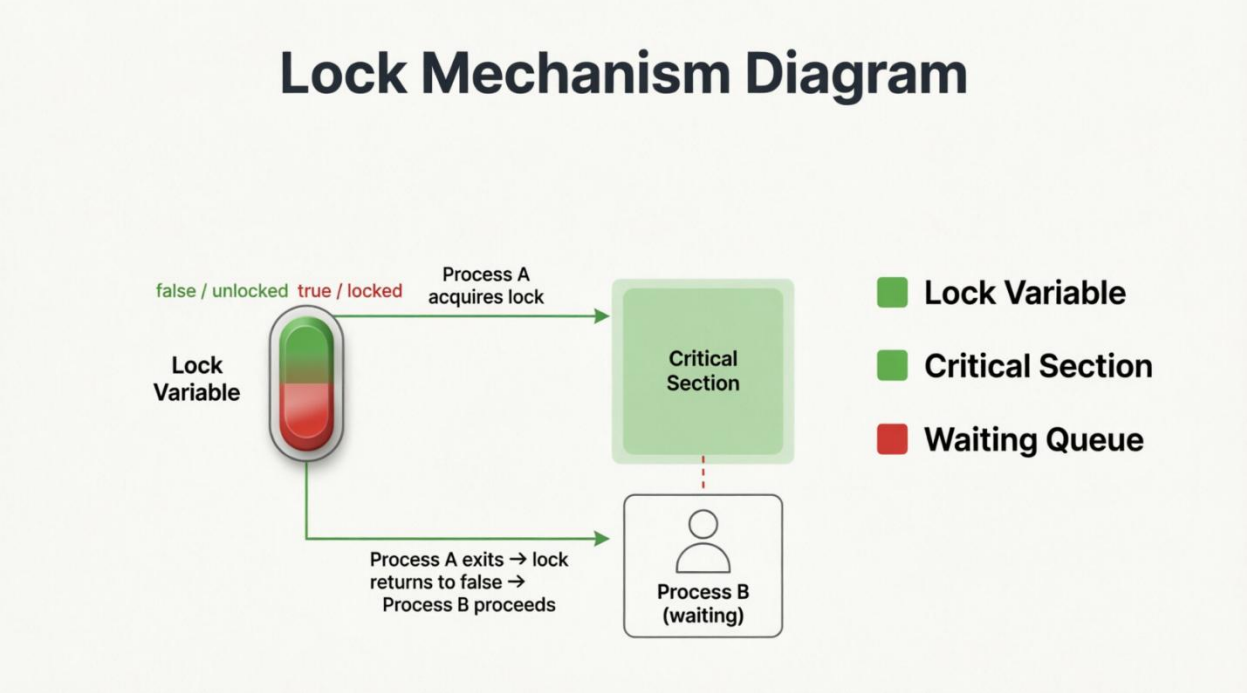


Software solutions for mutual exclusion can be based on active waiting (such as locks) or passive waiting (such as semaphores).

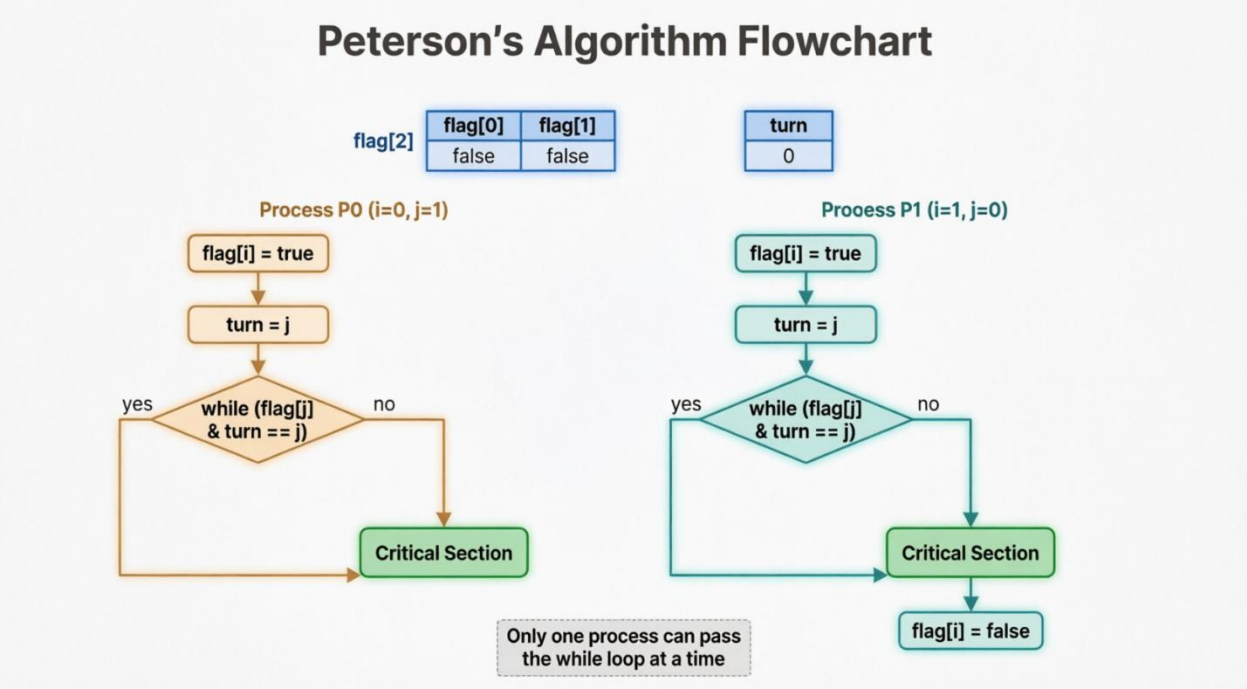
II.3. Locks

A lock is a variable shared by all processes that can be locked or unlocked.
A process must acquire the lock before entering a critical section and release it after leaving the critical section. If another process tries to acquire the lock while it is already held, it will be blocked until the lock is released.

- If the lock is false, the process sets it to true and enters the critical section.
- If the lock is true, the process waits until it becomes false to seize the resource.



Peterson's Algorithm:



Mutual exclusion based on a lock can be expressed with the following pseudo-algorithm:

```
// Shared variables:
lock : boolean (initialized to false)

// Process code:
function enterCriticalSection():
```

```
  while lock is true do:
    // Wait (active waiting loop or sleep)
  end while
  lock = true  // Acquire the lock

function exitCriticalSection():
  lock = false  // Release the lock

// Usage:
enterCriticalSection()
// ... critical section code ...
exitCriticalSection()
```

Example: Consider an example where two threads try to access a critical section (e.g., a shared counter). The lock prevents both threads from simultaneously modifying the counter, which could cause errors.

- **Shared variable: lock** — Initially, lock is set to false, indicating that the critical section is free. When a thread enters the critical section, it sets lock to true to prevent other threads from entering.
- **enterCriticalSection() function** — A thread checks if lock is true. If so, it stays in a waiting loop (active waiting). Once lock is false, the thread acquires the lock by setting it to true.
- **incrementCounter(threadID) function** — This function represents a critical section protected by the lock. It increments a shared variable (the counter) and displays information about the thread performing the operation.
- **exitCriticalSection() function** — After completing its work in the critical section, the thread releases the lock by setting lock to false.

The C code for this program is as follows:

```
#include <stdio.h>
#include <pthread.h>
#include <stdbool.h>
#include <unistd.h>  // For sleep()

// Shared variables
bool lock = false;  // Initialized to "unlocked"
int counter = 0;    // Shared resource

void enterCriticalSection() {
```

```
    while (lock) {
        // Busy wait
    }
    lock = true;  // Acquire the lock
}

void exitCriticalSection() {
    lock = false;  // Release the lock
}

void incrementCounter(int threadID) {
    enterCriticalSection();
    printf("Thread %d enters critical section.\n", threadID);
    counter++;
    printf("Thread %d incremented counter to %d.\n", threadID, counter);
    exitCriticalSection();
}

void* threadFunction(void* arg) {
    int threadID = *(int*)arg;
    for (int i = 0; i < 3; i++) {
        incrementCounter(threadID);
        sleep(1);
    }
    return NULL;
}

int main() {
    pthread_t thread1, thread2;
    int id1 = 1, id2 = 2;
    pthread_create(&thread1, NULL, threadFunction, &id1);
    pthread_create(&thread2, NULL, threadFunction, &id2);
    pthread_join(thread1, NULL);
    pthread_join(thread2, NULL);
    printf("Final counter value: %d\n", counter);
    return 0;
}
```

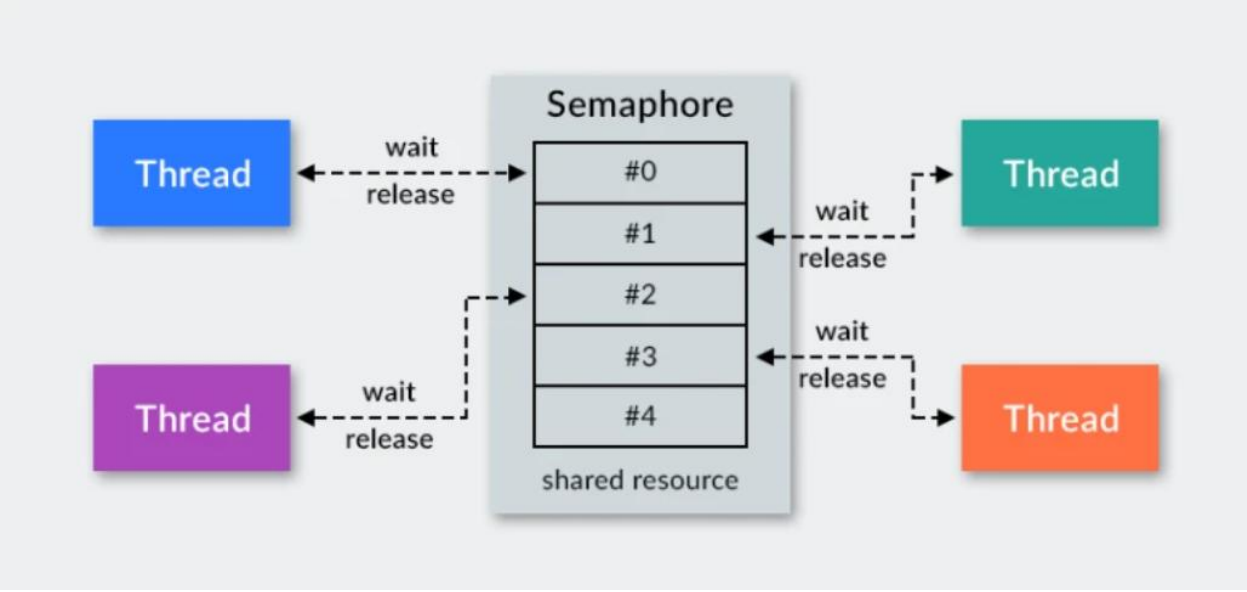
When this program is executed, the output is similar to the following (the order may vary depending on thread scheduling):

```
Thread 1 enters critical section.
Thread 1 incremented counter to 1.
Thread 2 enters critical section.
Thread 2 incremented counter to 2.
Thread 1 enters critical section.
Thread 1 incremented counter to 3.
Thread 2 enters critical section.
Thread 2 incremented counter to 4.
Thread 1 enters critical section.
Thread 1 incremented counter to 5.
Thread 2 enters critical section.
Thread 2 incremented counter to 6.
Final counter value: 6
```

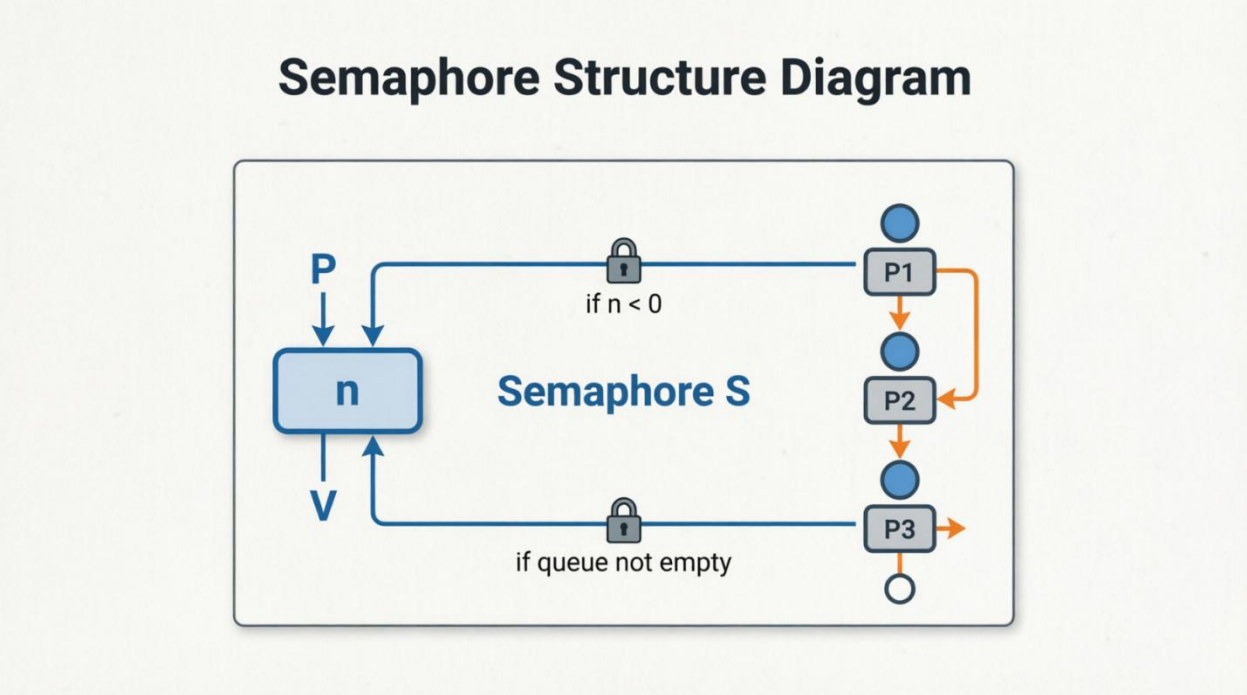
II.4. Semaphores

A semaphore is a variable that can be used to control access to a shared resource. A semaphore has a value that represents the number of available resources.
As shown in the figure, a semaphore is characterized by:

- A list (queue) of waiting processes.
- A counter (integer value).



Example:



The following code shows the structure of a semaphore:

```
typedef struct {
   int n;      // Semaphore value
   int* queue;  // Process queue (simulated by a dynamic array)
} Semaphore;
```

When a process needs to access the resource, it decrements the semaphore value. When a process releases the resource, it increments the semaphore value. If the semaphore value is less than zero, it means the resource is not available and the process must wait.

II.4.1. Primitives P and V

A semaphore is manipulated by two indivisible primitives (functions): P and V.

- **P(S):** Proberen (Dutch for "Can I?").
- **V(S):** Verhogen (Dutch for "Go ahead!").

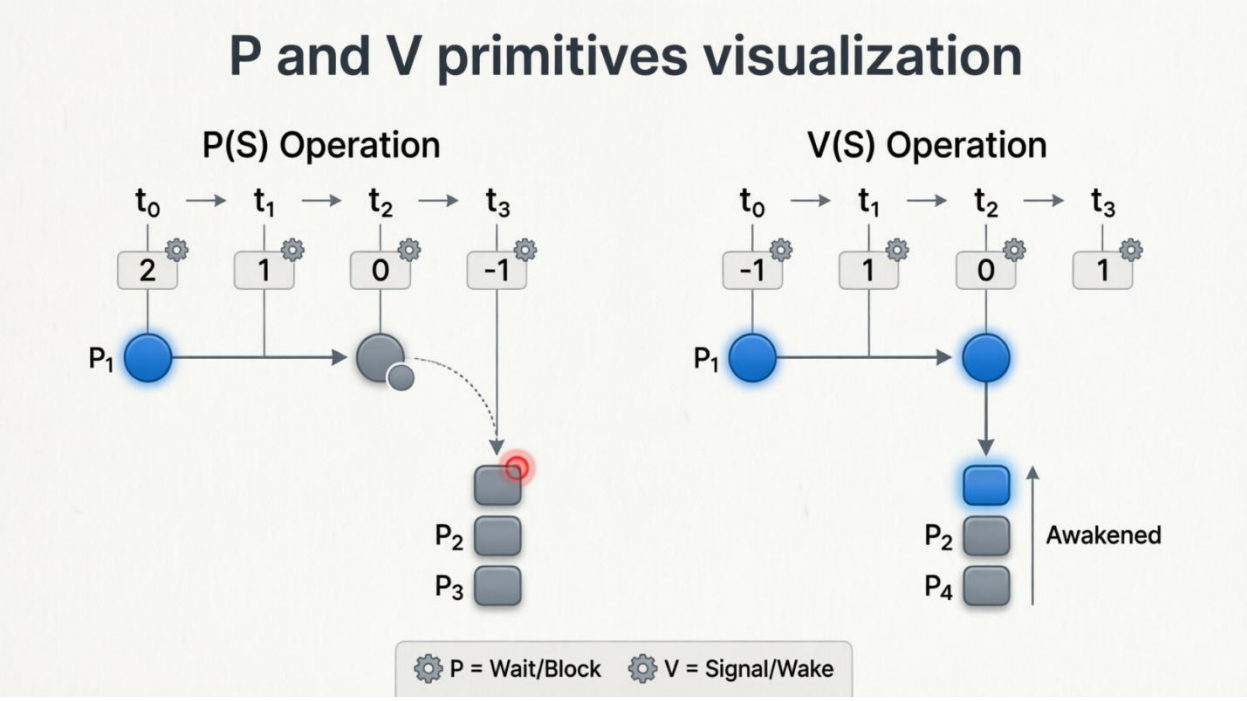
The P operation is also referred to as a wait, suspend, or block operation, while the V operation is referred to as a signal, wake-up, or activate operation.

Both operations are atomic. Atomicity means that the read, modify, and update actions on the variable are performed in an indivisible and uninterrupted manner. In other words, no other concurrent operation can intervene between these steps, thus ensuring the integrity of the variable.

II.4.2. The "init" Primitive

This function initializes a semaphore by defining its attributes.

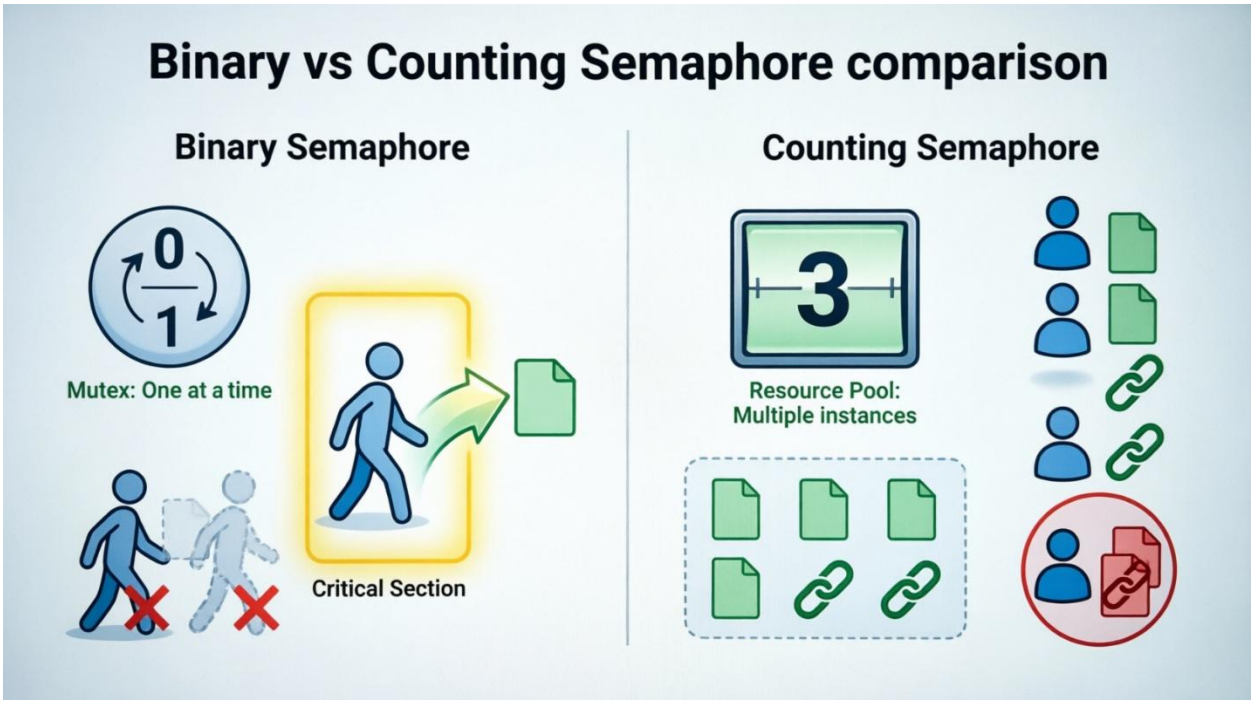
- The semaphore value (n) is set to the specified initial value.
- The blocked process queue (queue) is initially null to indicate there are no waiting processes.



The following code shows an example of semaphore initialization and its use:

```
void init(Semaphore* sem, int initial_value) {
   sem->n = initial_value;  // Initialize semaphore value
   sem->queue = NULL;      // Initialize queue to NULL
}

int main() {
   Semaphore sem;
   init(&sem, 3);
   printf("Initial semaphore value: %d\n", sem.n);
   return 0;
}
```

II.4.3. Types of Semaphores



(a) Binary Semaphore:

A binary semaphore is a semaphore whose value is limited to 0 or 1. It is used to ensure mutual exclusion (mutex) in the access to a shared resource among multiple threads or processes.

The following code shows the declaration and use of a binary semaphore in C:

```
#include <stdio.h>
#include <pthread.h>
#include <semaphore.h>

// Binary semaphore declaration
sem_t semaphore;

void* criticalSection(void* arg) {
   int threadID = *(int*)arg;
   // Wait (P operation)
   sem_wait(&semaphore);
   printf("Thread %d enters critical section.\n", threadID);
   sleep(1);  // Simulate operation
   printf("Thread %d exits critical section.\n", threadID);
   // Signal (V operation)
   sem_post(&semaphore);
   return NULL;
```

```
}

int main() {
   pthread_t threads[2];
   int threadIDs[2] = {1, 2};
   // Initialize binary semaphore with value 1
   sem_init(&semaphore, 0, 1);
   pthread_create(&threads[0], NULL, criticalSection, &threadIDs[0]);
   pthread_create(&threads[1], NULL, criticalSection, &threadIDs[1]);
   pthread_join(threads[0], NULL);
   pthread_join(threads[1], NULL);
   sem_destroy(&semaphore);
   return 0;
}
```

- This binary semaphore guarantees that only one thread accesses the critical section at a time.
- The semaphore is initialized to 1 to allow one thread to enter.
- The call to sem_wait() blocks a thread if the semaphore is at 0.
- The call to sem_post() releases the semaphore, allowing another thread to enter.

(b) Counting Semaphore:

A counting semaphore is a semaphore whose value can be any non-negative integer. It is used to manage multiple instances of a shared resource or to synchronize a set of events. The following code shows an example of a counting semaphore in C:

```
#include <stdio.h>
#include <pthread.h>
#include <semaphore.h>
#define MAX_RESOURCES 3

sem_t semaphore;

void* useResource(void* arg) {
   int threadID = *(int*)arg;
   sem_wait(&semaphore);  // P operation
   printf("Thread %d is using a resource.\n", threadID);
   sleep(1);
   printf("Thread %d releases the resource.\n", threadID);
   sem_post(&semaphore);  // V operation
```

```
    return NULL;
}

int main() {
    pthread_t threads[5];
    int threadIDs[5] = {1, 2, 3, 4, 5};
    sem_init(&semaphore, 0, MAX_RESOURCES);
    for (int i = 0; i < 5; i++) {
        pthread_create(&threads[i], NULL, useResource, &threadIDs[i]);
    }
    for (int i = 0; i < 5; i++) {
        pthread_join(threads[i], NULL);
    }
    sem_destroy(&semaphore);
    return 0;
}
```

- This semaphore is initialized to 3, allowing 3 threads to simultaneously access a resource.
- If all 3 threads are using the resource, others are blocked until a resource is released (call to sem_post()).

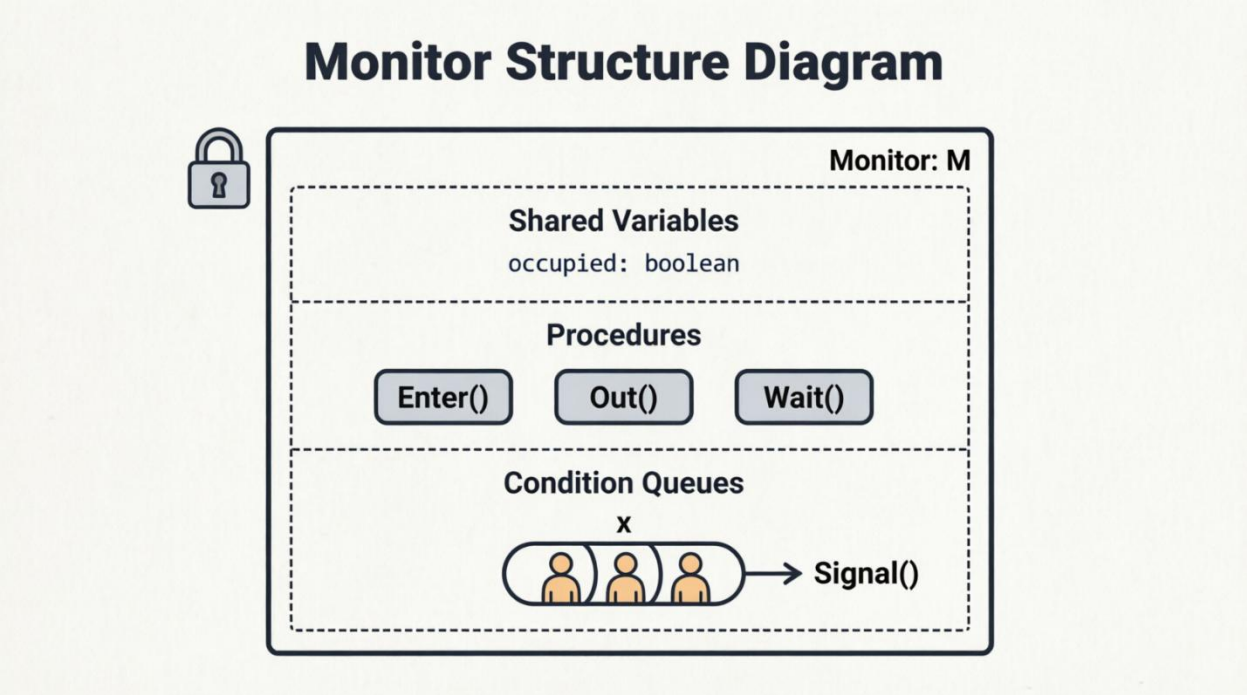
II.5. Monitors

A monitor is an abstraction used in operating systems to solve the problem of process synchronization. It allows grouping of shared variables along with the procedures acting on these variables, while guaranteeing mutual exclusion during their access.

II.5.1. Structure of a Monitor

A monitor consists of:

1. Internal data (or shared state variables).
2. Access primitives and procedures (entry points).
3. Queues (associated with condition-type variables).

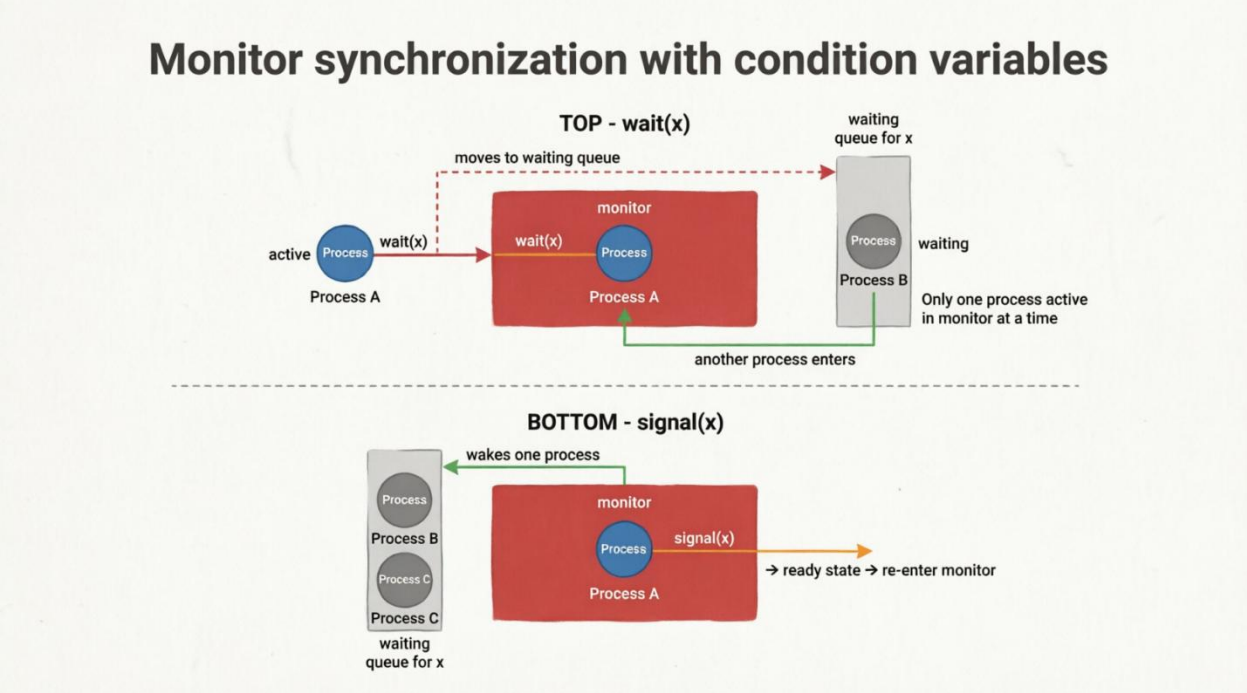


Example of a monitor structure:

```
Monitor: monitor-name

// Declaration of shared variables

Procedure enter P1() { }
Procedure enter P2() { }
Procedure enter Pn() { }

Begin
  // Initialization of variables
End
```

II.5.2. Fundamental Properties of a Monitor

4. Access to monitor procedures is mutually exclusive: only one process can be active in a monitor at any given time.
5. The entire monitor is implemented as a critical section.
6. Condition-type variables are used to synchronize processes.



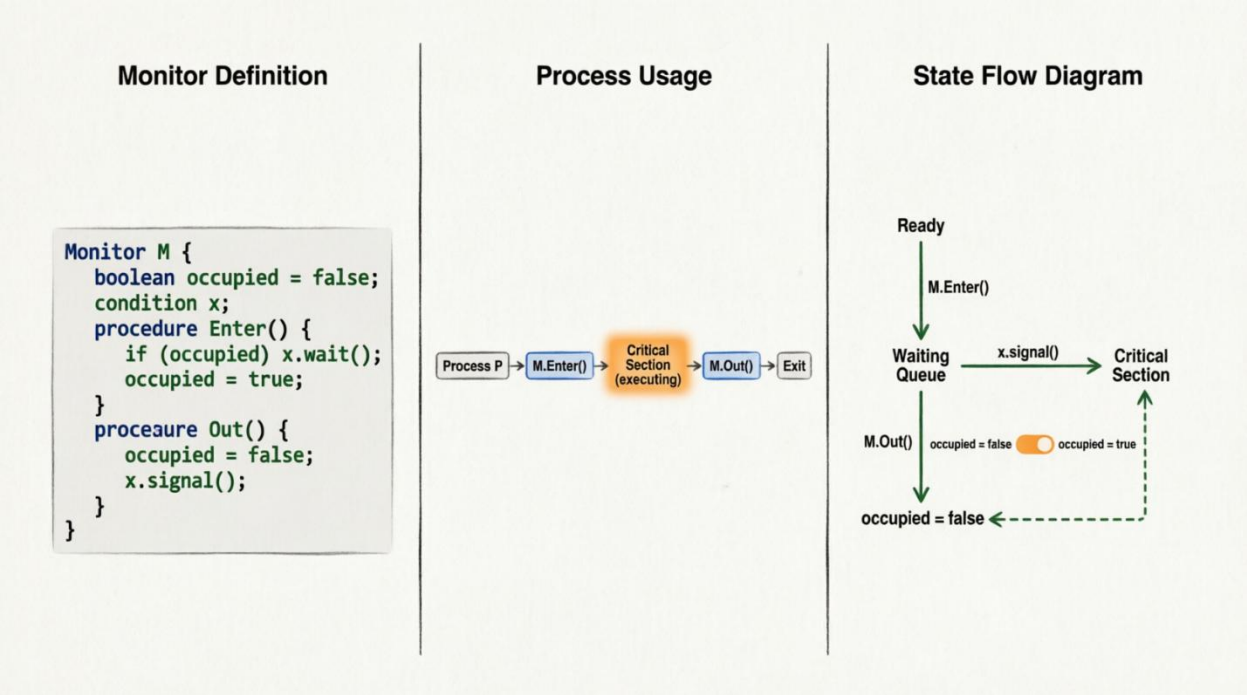
II.5.3. Synchronization in a Monitor

Synchronization in a monitor relies on two main primitives:

7. **Wait(x: condition):** Blocks the executing process. Places the process in the waiting queue associated with condition x.
8. **Signal(x: condition):** Wakes up a process blocked in the waiting queue associated with condition x. If the queue is empty, this primitive has no effect.

II.5.4. Monitor Primitives

- **Condition variables:** Defined with the condition type. They have no value (unlike semaphores). They represent queues of blocked processes.
- **Properties of condition variables:** Do not require initialization. Are manipulated only by the Wait() and Signal() primitives. Each condition variable corresponds to a queue.
- **Empty(c: condition) function: boolean:** Tests whether the queue associated with a condition is empty. Returns true if the queue is empty, otherwise false.



II.5.5. Advantages of Monitors

- Provide simplified management of critical sections and process synchronization.
- Guarantee that access to shared variables is done safely through mutual exclusion.

II.5.6. Example of Critical Section Access Management via a Monitor

```
Monitor M
   Occupied: Boolean
   x: condition

   Procedure Enter() {
      if (occupied) { wait(x); }
      occupied = true;
   }

   Procedure Out() {
      occupied = false;
      signal(x);
   }
Using the monitor from processes:
// Call monitor procedure
M.enter;
<critical section>
```

```
M.out;
```