

# Operating Systems

Prepared by Dr. Alaa Mohamed

## Module 3

جامعة الجميع الذكية

تسعى لتكون الكيان الأول الذي يتبنى مبدأ المجتمع التعليمي  
التعاوني ويقدم العلم مقابل العمل لتنمية وبناء المجتمعات.

Concurrency



جامعة الجميع الذكية  
EVERYONE'S SMART UNIVERSITY

- Objectives:
  - Background
  - The Critical-Section Problem
  - Software and Hardware Solutions
  - Semaphores
  - Classical Problems of synchronization

- Topics:
  - Describe the critical-section problem and illustrate a race condition
  - Illustrate solutions to the critical-section problem using Software, Hardware and Operating systems solutions
  - Demonstrate semaphores can be used to solve the critical section problem

- Concurrent processes usually need to share data (maintained either in shared memory or files) and resources
- If there is no controlled access to shared data, some processes will obtain an inconsistent view of this data
- The action performed by concurrent processes will then depend on the order in which their execution is interleaved

## An example

- Process P1 and P2 are running the procedure shown below and have access to the same variable “Counter”. Counter initially = 5
- Processes can be interrupted anywhere

| P1                                                    | P2                                                    |
|-------------------------------------------------------|-------------------------------------------------------|
| Counter = 5                                           |                                                       |
| <pre>X = Counter;<br/>X = X +1;<br/>Counter = X</pre> | <pre>Y = Counter;<br/>Y = Y -1;<br/>Counter = Y</pre> |

- The Initial value of Counter is 5, but due to the interruption of the process p1 and p2, the value of Counter is equal to either 4 or 6 but the final value should be 5. This is the problem of synchronization and concurrency.

| P1                                                    | P2                                                    |
|-------------------------------------------------------|-------------------------------------------------------|
| Counter = 5                                           |                                                       |
| <pre>X = Counter;<br/>X = X +1;<br/>Counter = X</pre> | <pre>Y = Counter;<br/>Y = Y -1;<br/>Counter = Y</pre> |

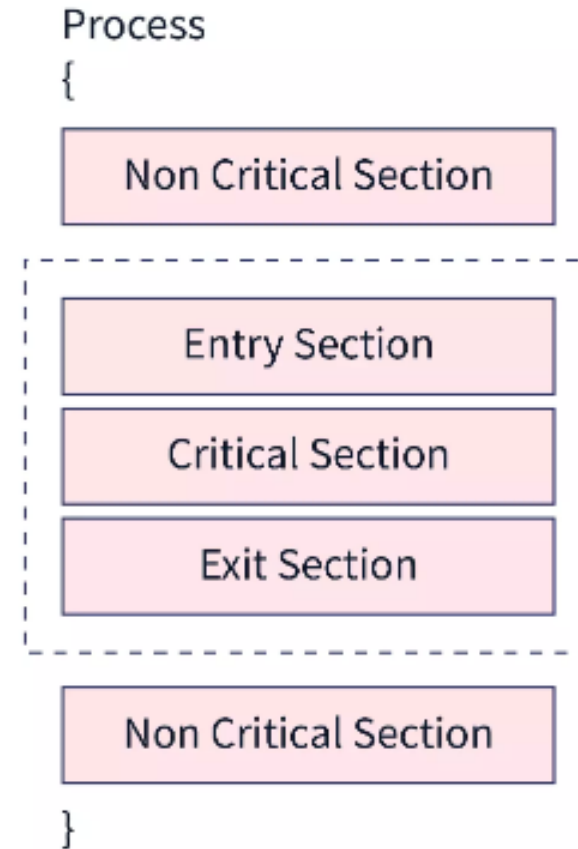
- Situations like this where processes access the same data concurrently and the outcome of execution depends on the particular order in which the access takes place is called a **race condition**
- How must the processes coordinate (or synchronise) in order to guard against race conditions?

- **critical section:** a section of code which reads/writes shared data
- **race condition:** potential for interleaved execution of a critical section by multiple processes => results are non-deterministic
- **mutual exclusion:** synchronization mechanism to avoid race conditions by ensuring exclusive execution of critical sections
- **deadlock:** permanent blocking of processes
- **starvation:** execution but no progress

- When a process executes code that manipulates shared data (or resource), we say that the process is in its **critical section** (CS) (for that shared data)
- The execution of critical sections must be **mutually exclusive**: at any time, only one process is allowed to execute in its critical section (even with multiple CPUs)
- Then each process must request the permission to enter its critical section (CS)

- The section of code implementing this request is called the **entry section**
- The critical section (CS) might be followed by an **exit section**
- The critical section problem is to design a protocol that the processes can use so that their action will not depend on the order in which their execution is interleaved (possibly on many processors)

General structure of process  $P_i$



## Requirements for a valid solution to the critical section problem

- **Mutual Exclusion**

- At any time, at most one process can be in its critical section (CS)

- **Progress**

- Only processes that are not executing in their RS can participate in the decision of who will enter next in the CS.
- This selection cannot be postponed indefinitely
  - Hence, we must have no deadlock

- **Bounded Waiting**

- After a process has made a request to enter its CS, there is a bound on the number of times that the other processes are allowed to enter their CS
  - otherwise the process will suffer from **starvation**

- **Software solutions**

- algorithms whose correctness does not rely on any other assumptions.

- **Hardware solutions**

- rely on some special machine instructions

- **Operation System solutions**

- provide some functions and data structures to the programmer

- **On a uniprocessor:** mutual exclusion is preserved but efficiency of execution is degraded: while in CS, we cannot interleave execution with other processes that are out of CS
- **On a multiprocessor:** mutual exclusion is not preserved
  - CS is now **atomic but not mutually exclusive**
  - Generally not an acceptable solution

```
Process Pi:  
repeat  
    disable interrupts  
    critical section  
    enable interrupts  
    remainder section  
forever
```

- Special hardware instructions that allow us to either test-and-modify the content of a word, or two swap the contents of two words atomically (uninterruptibly.)
- **Test-and-Set instruction**
- **Compare-and-Swap instruction**

- Consider we have three processes P0, P1, and P2 are interested to enter their critical section. So the structure for achieving mutual exclusion is as follow.
- Assume first that the lock initial value = False

```
do {  
  
    while (TestAndSet(&lock));  
  
    // critical section  
  
    lock = FALSE;  
  
    // remainder section  
  
} while (TRUE);
```



- **Disadvantages**

- Busy-waiting consumes processor time
- Starvation is possible when a process leaves a critical section and more than one process is waiting.
- Deadlock
  - If a low priority process has the critical section and a higher priority process needs, the higher priority process will obtain the processor to wait for the critical section

- Synchronization tool (provided by the OS) that do not require busy waiting
- A semaphore S is an integer variable that, apart from initialization, can only be accessed through 2 **atomic and mutually exclusive** operations:
  - wait(S)
  - signal(S)
- To avoid busy waiting: when a process has to wait, it will be put in a **blocked queue** of processes waiting for the same event

- Synchronization tool that provides more sophisticated ways for process to synchronize their activities.
- Semaphore  $S$  – integer variable
- Can only be accessed via two indivisible (atomic) operations
  - **wait()** and **signal()**
    - (Originally called **P()** and **V()**)

Definition of the `wait()` operation

```
wait(S) {  
    while (S <= 0)  
        ; // busy wait  
    S--;  
}
```

Definition of the `signal()` operation

```
signal(S) {  
    S++;  
}
```



**Counting semaphore** – integer value can range over an unrestricted domain

**Binary semaphore** – integer value can range only between 0 and 1

Can solve various synchronization problems

Consider  $P_1$  and  $P_2$  that require  $S_1$  to happen before  $S_2$

Create a semaphore “**synch**” initialized to 0

P1 :

$S_1$  ;

**signal** (synch) ;

P2 :

**wait** (synch) ;

$S_2$  ;



## Using semaphores to synchronize processes

Semaphore S1= 0, S2 = 0

| Process A                                                        | Process B                                                        |
|------------------------------------------------------------------|------------------------------------------------------------------|
| A1: Statement X1<br>signal(s1);<br>wait(s2);<br>A2: Statement X2 | B1: Statement Y1<br>wait(s1);<br>B2: Statement Y2<br>signal(s2); |

- In what order will the statements execute?



- For n processes
- Initialize S to 1
- Then only 1 process is allowed into CS (mutual exclusion)
- To allow k processes into CS, we initialize S to k

```
Process Pi:  
repeat  
    wait(S) ;  
    CS  
    signal(S) ;  
    RS  
forever
```

- **Deadlock** – two or more processes are waiting indefinitely for an event that can be caused by only one of the waiting processes
- Let  $s$  and  $q$  be two semaphores initialized to 1

| $P_0$                    | $P_1$                    |
|--------------------------|--------------------------|
| <code>wait(S) ;</code>   | <code>wait(Q) ;</code>   |
| <code>wait(Q) ;</code>   | <code>wait(S) ;</code>   |
| <code>...</code>         | <code>...</code>         |
| <code>signal(S) ;</code> | <code>signal(Q) ;</code> |
| <code>signal(Q) ;</code> | <code>signal(S) ;</code> |

- Consider if  $P_0$  executes `wait(S)` and  $P_1$  `wait(Q)`. When  $P_0$  executes `wait(Q)`, it must wait until  $P_1$  executes `signal(Q)`
- However,  $P_1$  is waiting until  $P_0$  execute `signal(S)`.
- Since these `signal()` operations will never be executed,  $P_0$  and  $P_1$  are **deadlocked**.

- Other forms of deadlock:
- **Starvation** – indefinite blocking
  - A process may never be removed from the semaphore queue in which it is suspended
- **Priority Inversion** – Scheduling problem when lower-priority process holds a lock needed by higher-priority process
- Solved via **priority-inheritance protocol**

- Classical problems used to test newly-proposed synchronization schemes
  - **Bounded-Buffer Problem**
  - Readers and Writers Problem
  - Dining Philosopher

## Bounded-Buffer Classical Problem

- The Bounded Buffer Problem( Producer Consumer Problem), is one of the classic problems of synchronization.
- There is Buffer of  $n$  slots and each slot is capable of storing one unit of data.
- There Are two processes running, namely, Producer and Consumer, which are operating on the buffer.



- The Producer tries to insert data into an empty slot of the buffer.
- The consumer tries to remove data from a filled slot in the buffer.
- The problems Comes from the following:
  - **The Producer must not insert data when the buffer is full.**
  - **The Consumer must not remove data when the buffer is empty.**
  - **The Producer and Consumer should not insert and remove data simultaneously.**

- We will make use of three semaphores:
  - **Semaphore mutex** initialized to the value 1, which is used to acquire and release the lock.
  - **Semaphore full** initialized to the value 0, since initially there is no full slots.
  - **Semaphore empty** initialized to the value  $n$ , since initially all slots are empty.
- $n$  buffers, each can hold one item

## Bounded-Buffer Classical Problem

- The structure of the producer process

```
while (true) {  
    ...  
    /* produce an item in next_produced */  
    ...  
    wait(empty);  
    wait(mutex);  
    ...  
    /* add next produced to the buffer */  
    ...  
    signal(mutex);  
    signal(full);  
}
```

$n$  buffers, each can hold one item

Semaphore **mutex** initialized to the value 1

Semaphore **full** initialized to the value 0

Semaphore **empty** initialized to the value  $n$



## Bounded-Buffer Classical Problem

- The structure of the consumer process

```
while (true) {  
    wait(full);  
    wait(mutex);  
  
    ...  
    /* remove an item from buffer to next_consumed */  
    ...  
    signal(mutex);  
    signal(empty);  
  
    ...  
    /* consume the item in next consumed */  
    ...  
}
```



## Questions on Semaphore

1. In the following code, three processes produce output using the routine “putc” and synchronize using two semaphores “L” and “R.”  
semaphore L = 3, R = 0; /\* initialization \*/

```
/* Process 1 */  
while( true) {  
    wait(L);  
    putc('C');  
    signal(R);  
}
```

```
/* process 2 */  
while( true) {  
    wait(R);  
    putc('A');  
    putc('B');  
    signal (R);  
}
```

```
/* process 3 */  
while( true) {  
    wait(R);  
    putc('D');  
}
```

- a. Is **CABABDDCABCABD** a possible output sequence when this set of processes runs?
- b. Is **CABACDBCABDD** a possible output sequence when this set of processes runs?

## Questions on Semaphore

- semaphore  $L = 3, R = 0$ ; /\* initialization \*/

```
/* Process 1 */  
while( true) {  
    wait(L);  
    putc('C');  
    signal(R);  
}
```

```
/* process 2 */  
while( true) {  
    wait(R);  
    putc('A');  
    putc('B');  
    signal (R);  
}
```

```
/* process 3 */  
while( true) {  
    wait(R);  
  
    putc('D');
```

- a. Is **CABABDDCABCABD** a possible output sequence when this set of processes runs?  
-----→ **NO: DD cannot be output consecutively in this place.**
- a. Is **CABACDBCABDD** a possible output sequence when this set of processes runs?  
----→ **YES**

2- The following pair of processes share a common set of variables "counter", "tempA" and "tempB":

Process A

...

A1: tempA = counter + 1;

A2: counter = tempA;

...

Process B

...

B1: tempB = counter + 2;

B2: counter = tempB;

...

The variable "counter" initially has the value 10 before either process begins to execute.

- a. What different values of "counter" are possible when both processes have finished executing? Give an order of execution of statements from processes A and B that would yield each of the values you give. For example, execution order A1, A2, B1, B2 would yield the value 13.

a. There are three possible values for X. Here are the possible ways in which statements from A and B can be interleaved.

A1 A2 B1 B2:  $X = 13$

A1 B1 A2 B2:  $X = 12$

A1 B1 B2 A2:  $X = 11$

B1 A1 B2 A2:  $X = 11$

B1 A1 A2 B2:  $X = 12$

B1 B2 A1 A2:  $X = 13$



- a. Modify the above programs for processes A and B by adding appropriate signal and wait operations on the binary semaphore "sync" such that the only possible final value of "counter" is 13. Indicate what should be the initial value of the semaphore "sync".

We need to ensure that A and B run uninterrupted, but it doesn't matter which runs first.

semaphore sync = 1;

| Process A                                                                                          | Process B                                                                                           |
|----------------------------------------------------------------------------------------------------|-----------------------------------------------------------------------------------------------------|
| <pre>wait(sync);<br/>A1: tempA = counter + 1;<br/>A2: counter =<br/>tempA;<br/>signal(sync);</pre> | <pre>wait(sync);<br/>B1: tempB = counter + 2;<br/>B2: counter = tempB;<br/><br/>signal(sync);</pre> |

## End of Module 3

