

Operating Systems

Prepared by Dr. Alaa Eldeen Sayed Ahmed

IT263 / CYB510
Concurrency and synchronization Tools

Lecture 5



- 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

- A cooperating process is one that can be affected by other processes executing in the system.
- Cooperating processes usually need to share data (maintained either in shared memory or files) and resources
- Concurrent Access to shared data may lead to inconsistent view of this data specially if the data access is not under control.

- One more issue, The action performed by concurrent processes will depend on the order in which their execution is interleaved
- In this chapter, we discuss different synchronization mechanisms to ensure the orderly execution of cooperating processes that share data, so the data consistency is maintained.

An Illustrative example

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

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

- Consider this execution interleaving with “count = 5” initially:
S0: P1 execute **X = counter** {X = 5}
S1: P1 execute **X = X + 1** {X = 6}
S2: P2 execute **Y = counter** {Y = 5}
S3: P2 execute **Y = Y - 1** {Y = 4}
S4: P1 execute **counter = X** {counter = 6}
S5: P2 execute **counter = Y** {counter = 4}
- We would arrive at this incorrect state because we allowed both processes to manipulate the variable counter concurrently.

- A situation like this, where several processes access and manipulate the same data concurrently and the outcome of the execution depends on a particular order in which the access takes place (Race condition)
- How must the processes coordinate (or synchronise) in order to guard against race conditions?

- 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 remaining code is called the Remainder section**

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

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

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

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**

- no process outside of a critical region may cause another process to block, Hence, we must have no deadlock

- **Bounded Waiting** No process must wait forever to enter its critical section, 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

- Not guaranteed to work on modern architectures! (But good algorithmic description of solving the problem)
- It is a Two Processes solution
- Assume that the `load` and `store` machine-language instructions are **atomic**; that is, cannot be interrupted

- The two processes share two variables:
 - **int turn;**
 - **boolean flag[2]**
- The variable **turn** indicates whose turn it is to enter the critical section
- The **flag** array is used to indicate if a process is ready to enter the critical section. **flag[i] = true** implies that process P_i is ready!

```
while (true){  
    flag[i] = true;  
    turn = j;  
    while (flag[j] && turn == j)  
        ;  
  
    /* critical section */  
  
    flag[i] = false;  
  
    /* remainder section */  
  
}
```

Provable that the three CS requirement are met:

1. Mutual exclusion is preserved
 P_i enters CS only if:
either **flag[j] = false** or **turn = i**
2. Progress requirement is satisfied
3. Bounded-waiting requirement is met

Hardware solutions: interrupt disabling

```
Process P1:  
repeat  
    disable interrupts  
    critical section  
    enable interrupts  
    remainder section  
forever
```

```
Process P2:  
repeat  
    disable interrupts  
    critical section  
    enable interrupts  
    remainder section  
forever
```

- This approach to solve critical section problem has the following drawback:
 - Interrupts could be disabled too long (i.e if the critical section has too many times of code), then other processes are blocked until this process reenables interrupts.
 - This solution is not efficient since to prevent p1 and p2 from interfering with one another; the solution blocks all p_i

- A Special hardware instruction (such as **Test and Set()**) is used to solve the synchronization problem.
- There is a shared variable lock which can take either of the two values True or False.
- Any process asks to acquire the lock before entering its critical section.
- If the lock is acquired by the process, it starts entering its critical section.
- But if the lock is used by another process, it waits and is not allowed to enter its critical section.

Definition:

```
boolean test_and_set (boolean *target)
{
    boolean rv = *target;
    *target = true;
    return rv;
}
```

1. Executed atomically
2. Returns the original value of passed parameter
3. Set the new value of passed parameter to **true**

Lecture 05

Hardware solutions: special machine Instructions- Test and set().

Shared Boolean variable lock, initialized to false

```
P1
do {
    while (test_and_set(&lock))
        ; /* do nothing */

    /* critical section */

    lock = false;
    /* remainder section */
} while (true);
```

```
P2
do {
    while (test_and_set(&lock))
        ; /* do nothing */

    /* critical section */

    lock = false;
    /* remainder section */
} while (true);
```

- **It is cleared This solution satisfy mutual exclusion.**
- **But it is doesn't satisfy bounded waiting property.**
- **Disadvantages**
 - Busy-waiting consumes processor time
 - Starvation situation is possible, when a process leaves a critical section, and more than one process is waiting.

- It is a more general solution to mutual exclusion
- Semaphore S is an integer variable that, is accessed only through 2 standard atomic operations.
- **wait()**, also called $P()$.
- **Signal()**, also called $V()$.

- OS provides ways to create and manipulate semaphores atomically
- **Counting semaphore** – integer value can range over an unrestricted domain
- **Binary semaphore** – integer value can range only between 0 and 1

Pseudo-code for classic semaphore: its value can't go below zero, i.e. classic semaphore is non-negative

Wait(S): Decrements the semaphore value. If the value becomes zero, the process/thread blocks until another process/thread signals it.

Signal (S): Increments the semaphore value. If any process/thread is waiting, it gets unblocked.

Definition of the `wait()` operation

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

Definition of the `signal()` operation

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

Initial Value Significance:

The initial value of a semaphore determines how many processes can execute after calling the wait() operation. There are three common scenarios:

1- **Binary Semaphore** (Mutex Lock): Initialized to 1. It ensures mutual exclusion by allowing only one process at a time.

P1

wait(S)

CRITICAL SECTION;

signal(S)

P2

wait(S)

CRITICAL SECTION;

signal(S)

P3

wait(S)

CRITICAL SECTION;

signal(S)

2- **Counting Semaphore:** Its value can range over an unrestricted domain. Used to control access to resources with multiple instances.

P1

wait(S)

Resource;

signal(S)

P2

wait(S)

resource;

signal(S)

P3

wait(S)

resource;

signal(S)

3- Ensuring orderly execution in multi-process/thread environments. The initial value of the semaphore is equal to 0;

P1

```
Instruction1;  
signal (S) ;
```

P2

```
wait (S);  
Instruction2;
```

- Processes may have to wait indefinitely while trying to acquire a synchronization tool such as a mutex lock or semaphore.
- Waiting indefinitely violates the progress and bounded-waiting criteria discussed at the beginning of this chapter.
- Liveness** refers to a set of properties that a system must satisfy to ensure processes make progress.
- Indefinite waiting is an example of a liveness failure.

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

```
wait(S) ;  
wait(Q) ;  
...  
signal(S) ;  
signal(Q) ;
```

P_1

```
wait(Q) ;  
wait(S) ;  
...  
signal(Q) ;  
signal(S) ;
```

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**

Lecture 05

Liveness

Example that explain a scenario that cause Priority inversion

Process L

Wait(S)

Resource

Signal (S)

ProcessH

Wait(S)

Resource

Signal (S)

Process M

-

-

-

-

-

To prevent this from occurring, a **priority inheritance protocol** is used. This simply allows the priority of the highest process waiting to access a shared resource to be assigned to the process currently using the resource. Thus, the current owner of the resource is assigned the priority of the highest priority process wishing to acquire the resource.

Classical problems used to test newly-proposed synchronization schemes

- Bounded-Buffer Problem
- Readers and Writers Problem
- Dining-Philosophers 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 (binary)** initialized to the value 1, which is used to acquire and release the lock.
 - **Semaphore full (counting)** initialized to the value 0, since initially there is no full slots.
 - **Semaphore empty (counting)** initialized to the value n , since initially all slots are empty.
- n buffers, each can hold one item

Lecture 05

Bounded-Buffer 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);  
}
```

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 */  
    ...  
}
```

- A data set is shared among a number of concurrent processes
- **Readers** – only read the data set; they do ***not*** perform any updates
- **Writers** – can both read and write
- Problem – allow multiple readers to read at the same time
Only one single writer can access the shared data at the same time

Shared Data

Data set

Semaphore **rw_mutex** initialized to 1

Semaphore **mutex** initialized to 1

Integer **read_count** initialized to 0

Lecture 05

Reader Writer Problem

The structure of a writer process

```
while (true) {  
    wait(rw_mutex);  
    ...  
    /* writing is performed */  
    ...  
    signal(rw_mutex);  
}
```

The structure of a reader process

```
while (true) {  
    wait(mutex);  
    read_count++;  
    if (read_count == 1)  
        wait(rw_mutex);  
    signal(mutex);  
    ...  
    /* reading is performed */  
    ...  
    wait(mutex);  
    read_count--;  
    if (read_count == 0)  
        signal(rw_mutex);  
    signal(mutex);  
}
```

- Philosophers spend their lives alternating thinking and eating
- Don't interact with their neighbors, occasionally try to pick up 2 chopsticks (one at a time) to eat from bowl
- Need both to eat, then release both when done
- In the case of 5 philosophers



- **Shared data**

Bowl of rice (data set)

Semaphore chopstick [5] initialized to 1

Semaphore Solution

The structure of Philosopher i :

```
while (true){  
    wait (chopstick[i] );  
    wait (chopstick[ (i + 1) % 5] );  
  
    /* eat for awhile */  
  
    signal (chopstick[i] );  
    signal (chopstick[ (i + 1) % 5] );  
  
    /* think for awhile */  
}
```

What is the problem with this algorithm?

- deadlock

How to solve it?

1- Semaphore approach:

- Use binary **semaphore S** with initial value 1 to allow Each philosopher waits for both chopsticks (left and right) before eating.

If a fork(chopstick) is unavailable, the philosopher waits.

- When done eating, they release both forks.

•**2-Limit Concurrent Access:**

- Leave one seat unoccupied (one philosopher fewer).
- This ensures that at least one chopstick is always available.
- Philosophers can then take turns without deadlock.



Thank You