



# OPERATING SYSTEMS

## Threads and Concurrency 07

Nitin V Pujari  
Faculty, Computer Science  
Dean - IQAC, PES University

## Course Syllabus - Unit 2

---

### UNIT 2: Threads and Concurrency

Introduction to Threads, types of threads, Multicore Programming, Multithreading Models, Thread creation, Thread Scheduling, PThreads and Windows Threads, Mutual Exclusion and Synchronization: software approaches, principles of concurrency, hardware support, Mutex Locks, Semaphores. Classic problems of Synchronization: Bounded-Buffer Problem, Readers -Writers problem, Dining Philosophers Problem concepts. Synchronization Examples - Synchronisation mechanisms provided by Linux/Windows/Pthreads. Deadlocks: principles of deadlock, tools for detection and Prevention.

## Course Outline - Unit 2

13	4.1.4.2	Introduction to Threads, types of threads, Multicore Programming.	4	42.8
14	4.3.5.4	Multithreading Models, Thread creation, Thread Scheduling	4	
15	4.4	Pthreads and Windows Threads	4	
16	6.1-6.3	Mutual Exclusion and Synchronization: software approaches	6	
17	6.3-6.4	principles of concurrency, hardware support	6	
18	6.5.6.6	Mutex Locks, Semaphores	6	
19	6.7.1-6.7.3	Classic problems of Synchronization: Bounded-Buffer Problem, Readers -Writers problem, Dining Philosophers Problem concepts	6	
20	6.9	Synchronization Examples - Synchronisation mechanisms provided by Linux/Windows/Pthreads.	6	
21	Handouts	Demonstration of programming examples on process synchronization		
22	7.1-7.3	Deadlocks: principles of deadlock, Deadlock Characterization.	7	
23	7.4	Deadlock Prevention, Deadlock example	7	
24	7.6	Deadlock Detection	7	

- **Process Synchronisation**
  - **Synchronization Hardware**
  - **Hardware Solution to Critical Section Problem**
  - **Test and Set Instruction**
  - **Compare and Swap Instruction**

# Synchronization Hardware

- Many systems provide **hardware** support for implementing the critical section code.
- All **solutions** below based on idea of **locking**
- **Protecting** critical regions via **locks**
- **Uniprocessor** – could disable **interrupts**
- Currently running code would execute without preemption
- Generally too **inefficient** on **multiprocessor** systems
- Operating systems using this **not** broadly **scalable**

# Synchronization Hardware

- Concurrent access to shared data may result in data inconsistency
- Maintaining data consistency requires mechanisms to ensure the orderly execution of cooperating processes
- Modern machines provide special **atomic** hardware instructions
- **Atomic** mean **non-interruptible**
- Either test memory word and set value or swap contents of two memory words

# Synchronization Hardware

- **Hardware** features can make any **programming** task **easier** and **improve** system **efficiency**.
- Some simple hardware instructions that are available on many systems and show how they can be used effectively in solving the critical-section problem.
- The critical-section problem could be solved simply in a uniprocessor environment if we could prevent interrupts from occurring while a shared variable was being modified.
- In this manner, one could be sure that the current sequence of instructions would be allowed to execute in order without preemption.

# Synchronization Hardware

---

- No other instructions would be run, so no unexpected modifications could be made to the shared variable.
- This is the **approach** taken by **non-preemptive** kernels. Unfortunately, this **solution** is not as **feasible** in a **multiprocessor** environment.
- **Disabling** interrupts on a **multiprocessor** can be **time consuming**, as the message is passed to all the processors.

# Synchronization Hardware

- This **message** passing **delays** entry into each critical section, and system **efficiency decreases**.
- Also, consider the effect on a system's clock, if the clock is kept updated by interrupts.
- Many modern computer systems therefore provide special hardware instructions that allow us either to test and modify the content of a word or to swap the contents of two words atomically that is, as one uninterruptible unit.

# Synchronization Hardware

---

- One can use these **special** instructions to solve the critical-section problem in a **relatively simple** manner.
- Rather than discussing one specific instruction for one specific machine, one can **abstract** the main concepts behind these types of instructions.

# Solution to the CSP using Locks

Process P0

```
do {  
    acquire_lock()  
    critical section  
    release_lock()  
    remainder section  
} while (TRUE);
```

Process P1

```
do {  
    acquire_lock()  
    critical section  
    release_lock()  
    remainder section  
} while (TRUE);
```

# Solution to the CSP using test\_and\_set

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

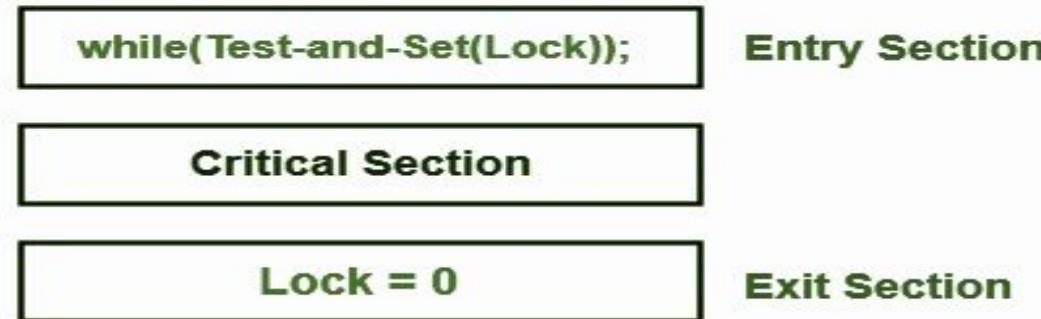
- Executed atomically
- Returns the original value of passed parameter
- Set the new value of passed parameter to “TRUE”.

# Solution to the CSP using test\_and\_set

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

- The important characteristic is that this **instruction** is executed **atomically**.
- Thus, if **two** TestAndSet instructions are executed simultaneously each on a **different CPU**, they will be executed **sequentially** in some **arbitrary** order.
- **If** the machine **supports** the **TestAndSet ()** instruction, then we can implement **mutual exclusion** by declaring a Boolean variable **lock**, initialized to **false**.

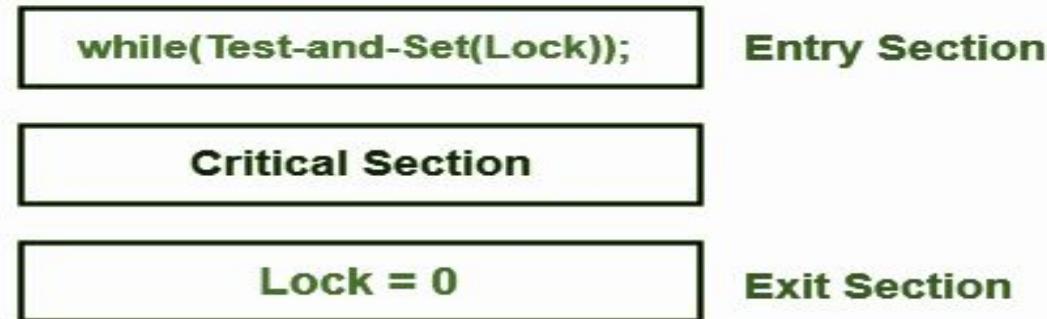
# Solution to the CSP using test\_and\_set



Initially, lock value is set to 0.

- Process  $P_0$  arrives.
- It executes the test-and-set(Lock) instruction.
- Since lock value is set to 0, so it returns value 0 to the while loop and sets the lock value to 1.
- The returned value 0 breaks the while loop condition.
- Process  $P_0$  enters the critical section and executes.
- Now, even if process  $P_0$  gets preempted in the middle, no other process can enter the critical section.
- Any other process can enter only after process  $P_0$  completes and sets the lock value to 0.

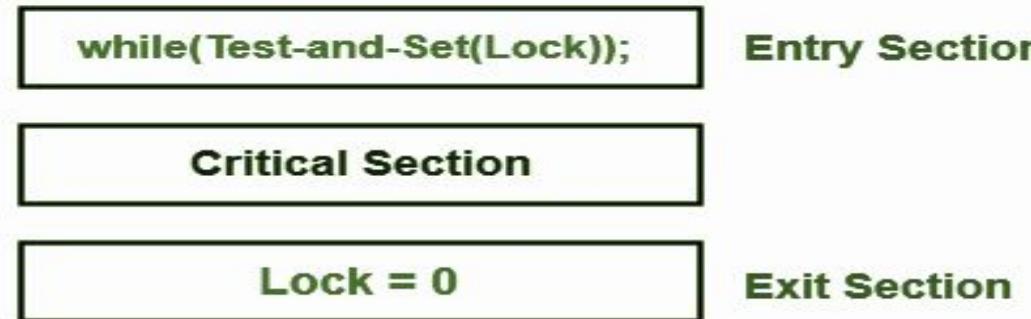
# Solution to the CSP using test\_and\_set



Initially, lock value is set to 0.

- Another process  $P_1$  arrives.
- It executes the test-and-set(Lock) instruction.
- Since lock value is now 1, so it returns value 1 to the while loop and sets the lock value to 1.
- The returned value 1 does not break the while loop condition.
- The process  $P_1$  is trapped inside an infinite while loop.
- The while loop keeps the process  $P_1$  busy until the lock value becomes 0 and its condition breaks.

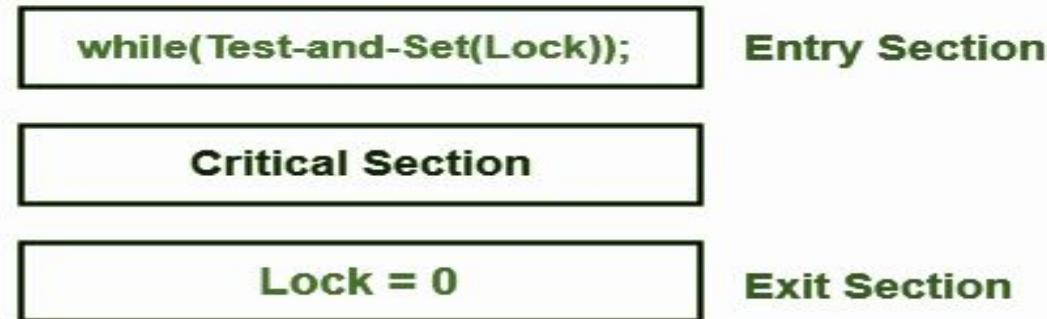
# Solution to the CSP using test\_and\_set



Initially, lock value is set to 0.

- Process  $P_0$  comes out of the critical section and sets the lock value to 0.
- The while loop condition breaks.
- Now, process  $P_1$  waiting for the critical section enters the critical section.
- Now, even if process  $P_1$  gets preempted in the middle, no other process can enter the critical section.
- Any other process can enter only after process  $P_1$  completes and sets the lock value to 0.

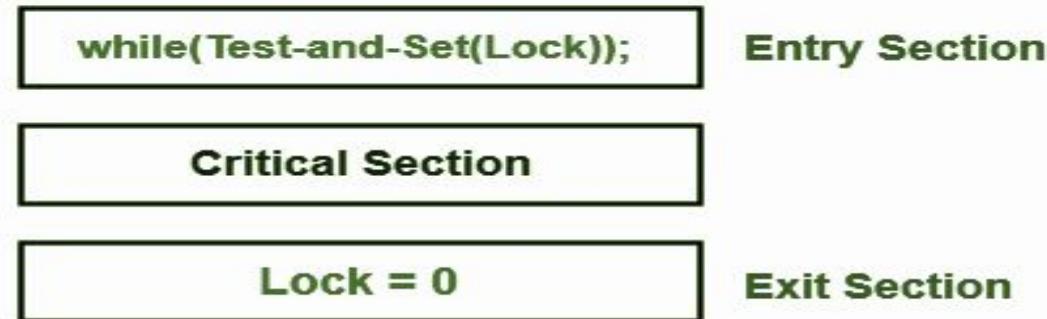
# Solution to the CSP using test\_and\_set



Initially, lock value is set to 0.

- It **ensures** mutual exclusion.
- It is deadlock **free**.
- It **does not** guarantee **bounded waiting** and may cause starvation.
- It suffers from **spin lock**.
- It is **not** architectural **neutral** since it requires the operating system to support test-and-set instruction.
- It is a busy waiting solution which keeps the CPU busy when the process is actually waiting.

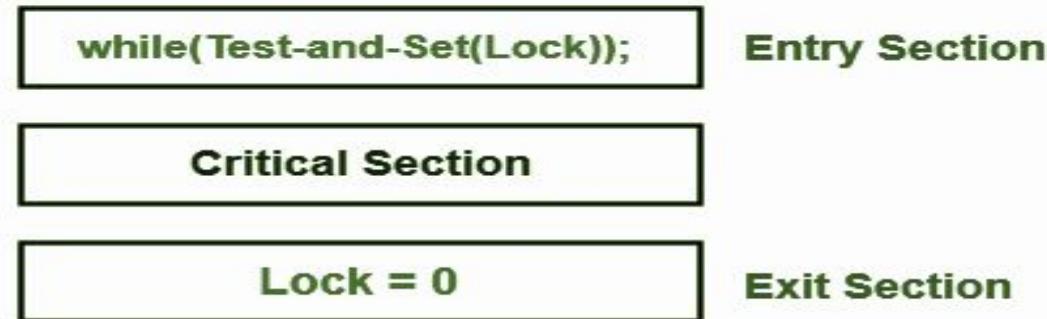
# Solution to the CSP using test\_and\_set



Initially, lock value is set to 0.

- This synchronization mechanism guarantees mutual exclusion.
- The success of the mechanism in providing mutual exclusion lies in the test-and-set instruction.
- Test-and-set instruction returns the old value of memory location (lock) and updates its value to 1 simultaneously.
- The fact that these two operations are performed as a single atomic operation ensures mutual exclusion.
- Preemption after reading the lock value was a major cause of failure of lock variable synchronization mechanism.
- Now, no preemption can occur immediately after reading the lock value.

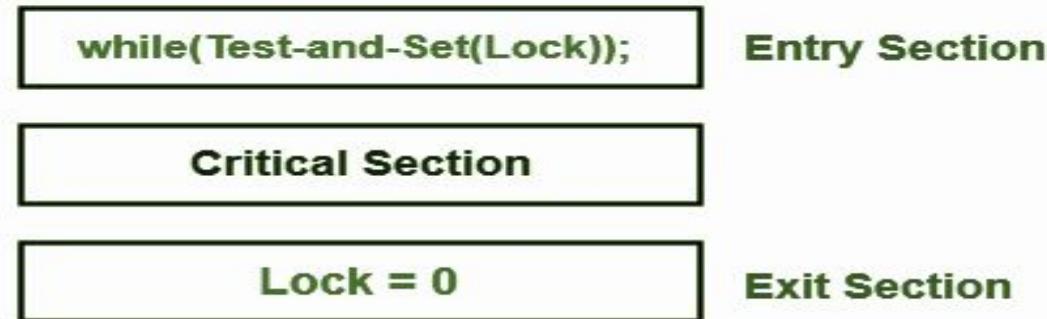
# Solution to the CSP using test\_and\_set



Initially, lock value is set to 0.

- **This synchronization mechanism guarantees freedom from deadlock.**
- After arriving, process executes the test-and-set instruction which returns the value 0 to while loop and sets the lock value to 1.
- Now, no other process can enter the critical section until the process that has begun the test-and-set finishes executing the critical section.
- Other processes can enter only after the process that has begun the test-and-set finishes and set the lock value to 0.
- This prevents the occurrence of deadlock.

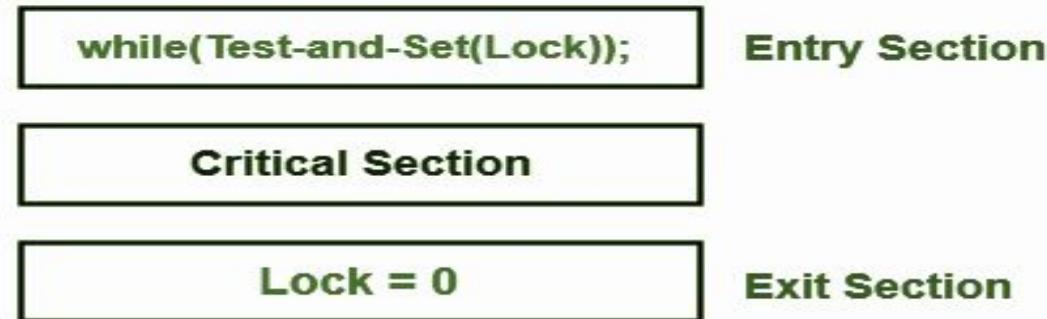
# Solution to the CSP using test\_and\_set



Initially, lock value is set to 0.

- **This synchronization mechanism does not guarantee bounded waiting.**
- This synchronization mechanism may cause a process to starve for the CPU.
- There might exist an unlucky process which when arrives to execute the critical section finds it busy.
- So, it keeps waiting in the while loop and eventually gets preempted.
- When it gets rescheduled and comes to execute the critical section, it finds another process executing the critical section.
- So, again, it keeps waiting in the while loop and eventually gets preempted.
- This may happen several times which causes that unlucky process to starve for the CPU.

# Solution to the CSP using test\_and\_set



Initially, lock value is set to 0.

- This synchronization mechanism suffers from spin lock where the execution of processes is blocked.

Consider a scenario where-

- Priority scheduling algorithm is used for scheduling the processes.
- On arrival of a higher priority process, a lower priority process is preempted from the critical section.

Now,

- Higher priority process comes to execute the critical section.
- But synchronization mechanism does not allow it to enter the critical section before lower priority process completes.
- But lower priority process cannot be executed before the higher priority process completes execution.
- Thus, the execution of both the processes is blocked.

# Solution to the CSP using compare\_and\_swap

- **Compare and swap** is a technique used when designing concurrent algorithms.
- Basically, compare and swap compares an expected value to the concrete value of a variable, and if the concrete value of the variable is equals to the expected value, swaps the value of the variable for a new variable.
- Compare and swap may sound a bit complicated but it is actually reasonably simple once one understand it,
- CAS is a technique used to obtain synchronization during multiple writes where each write value depends upon the current state of the shared variable.
- It basically means that a variable will first be compared with a value to see if it has changed.

# Solution to the CSP using compare\_and\_swap

- If it has **changed**, then it means its value has been updated by some other process or thread and hence swap is not possible.
- In such a case, current value of the shared variable is obtained and new value is calculated from it.
- If it has not changed, then it means that its value has not been modified by any other process or thread and so it can be swapped.

```
int compare_and_swap(int *value, int expected, int new_value) {  
    int temp = *value;  
    if (*value == expected)  
        *value = new_value;  
    return temp;  
}
```

- Executed atomically
- Returns the original value of passed parameter “value”
- Set the variable “value” the value of the passed parameter “new\_value” but only if “value” ==“expected”. That is, the swap takes place only under this condition.

# Dekker's Solution Algorithm for the CS Problem

## Topics Uncovered in this Session

---

- Process Synchronisation
  - Synchronization Hardware
  - Hardware Solution to Critical Section Problem
  - Test and Set Instruction
  - Compare and Swap Instruction



**THANK YOU**

**Nitin V Pujari  
Faculty, Computer Science  
Dean - IQAC, PES University**

**nitin.pujari@pes.edu**

**For Course Deliverables by the Anchor Faculty click on [www.pesuacademy.com](http://www.pesuacademy.com)**