

# Synchronization-II

Didem Unat

Lecture 10

COMP304 - Operating Systems (OS)

# Race Condition

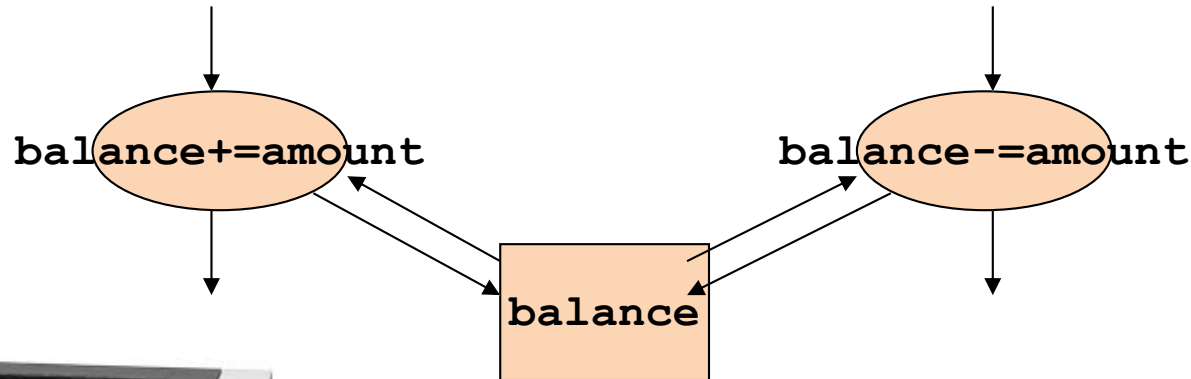
shared double balance;

## Code for $p_1$ (deposit)

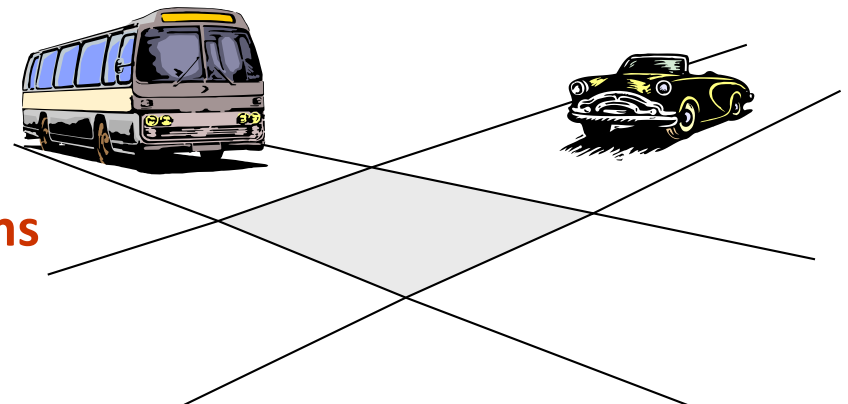
```
. . .  
balance = balance + amount;  
. . .
```

## Code for $p_2$ (withdraw)

```
. . .  
balance = balance - amount;  
. . .
```



Interleaved Printing



Traffic Intersections

# Load, Execute, Store

## Execution of p<sub>1</sub>

...  
load R1, balance  
load R2, amount

Timer interrupt (process p1 preemption)

Timer interrupt (process p2 preemption)

add R1, R2  
store balance, R1  
...

## Execution of p<sub>2</sub>

...  
load R1, balance  
load R2, amount  
sub R1, R2  
store balance, R1  
...

# Race Condition

- **Race condition:** The situation where several processes access and manipulate shared data concurrently. The final value of the shared data is non-deterministic and depends upon which process finishes last.
- To prevent race conditions, concurrent processes must be **synchronized**.

# The Critical Section Problem

- Each process has a code segment, called **critical section**, in which the shared data is accessed.
- **Problem** – ensure that when one process is executing in its critical section, no other process is allowed to execute in its critical section.
- Three properties for any solution for the critical section problem:
  - Mutual Exclusion, Progress, Bounded Waiting Time

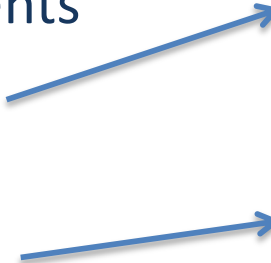
# Atomic Instructions

- Peterson's solution is a software-based solution
- Modern machines provide special atomic hardware instructions
  - **Atomic** = indivisible instructions

- The statements

**counter++;**

**counter--;**

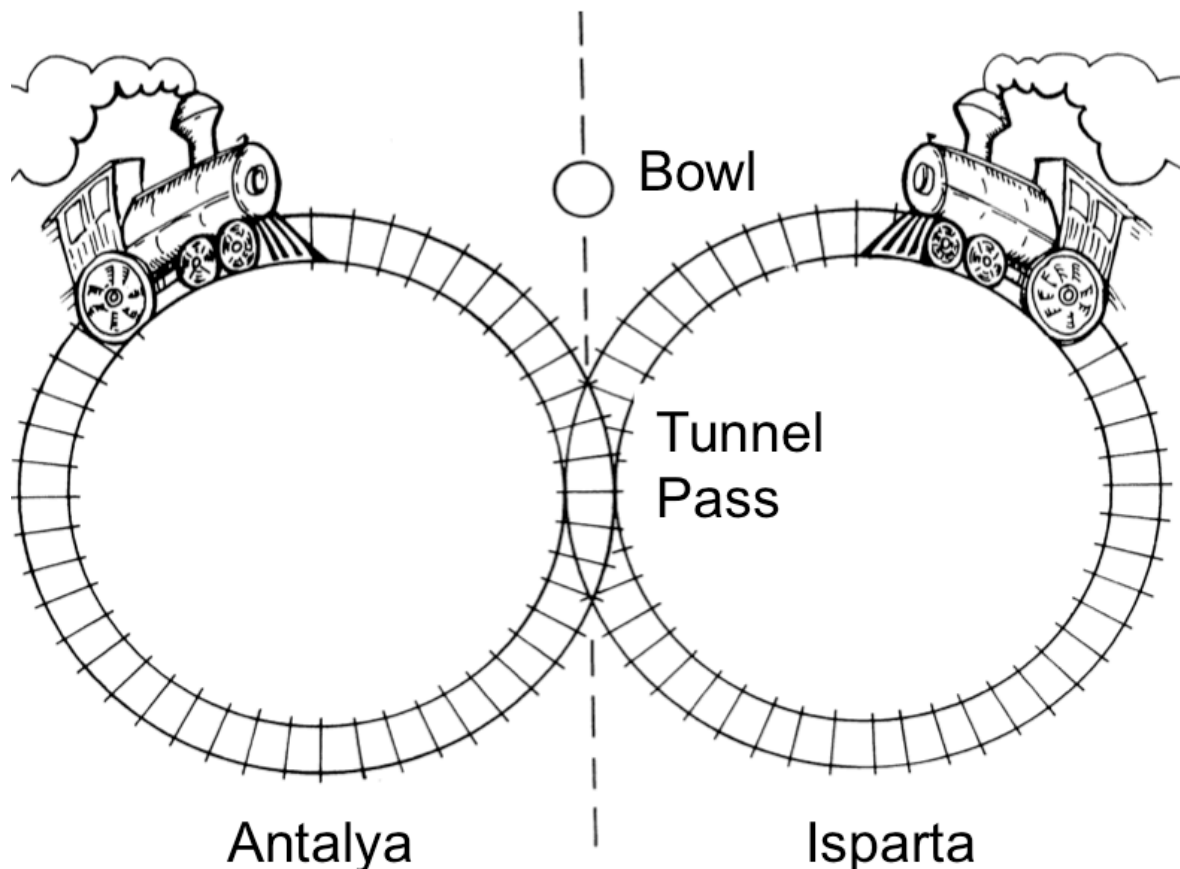


```
register1 = counter  
register1 = register1 + 1  
counter = register1
```

```
register2 = counter  
register2 = register2 - 1  
counter = register2
```

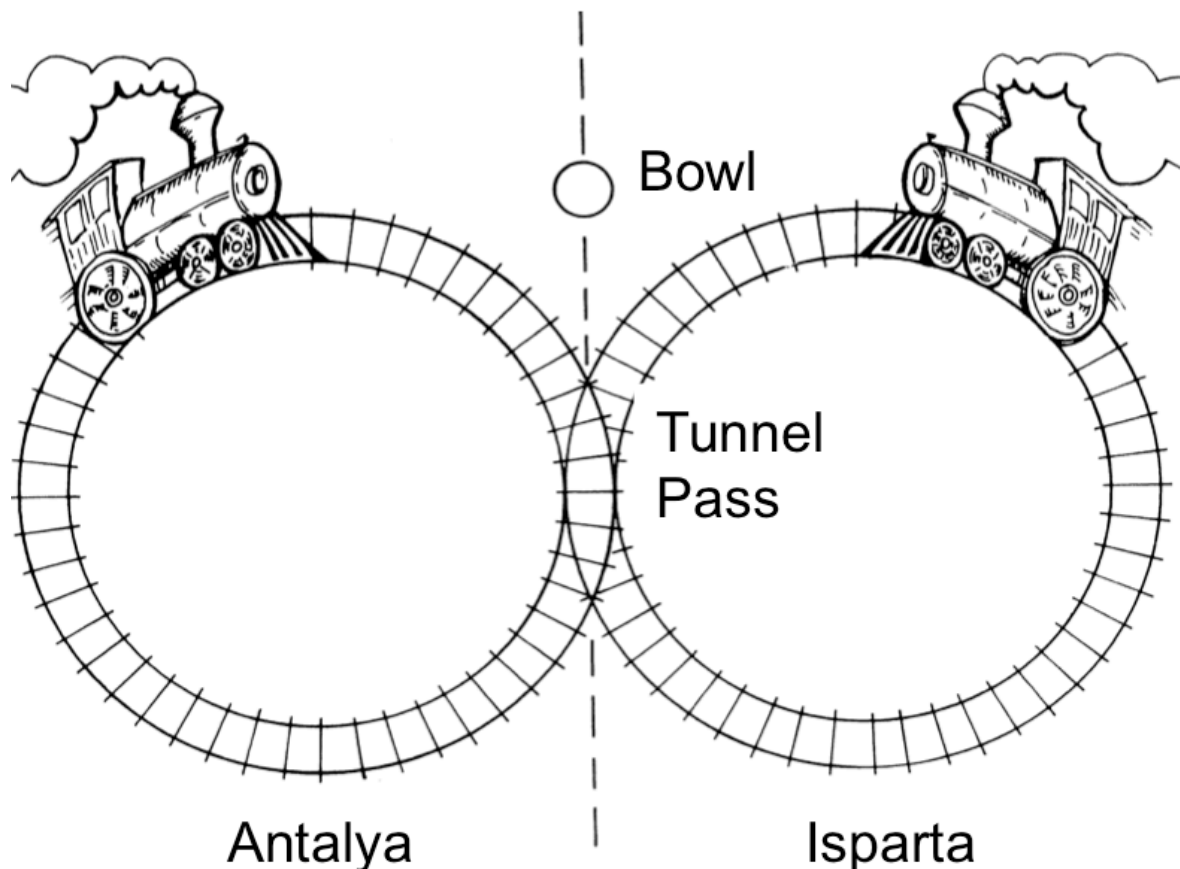
- must be performed **atomically**.
- Atomic operation means an operation that completes in its entirety **without interruption**.

# Question



- High in the Toros mountains, there are two circular railway lines. One line is in Antalya, the other in Isparta.
- The Antalya train runs twenty times a day, the Isparta train runs ten times a day.
- They share a common section of track where the lines cross a tunnel pass that lies on the city border.
- Unfortunately, the two trains occasionally collide when simultaneously entering the common section of track (the tunnel pass).
- The problem is that the drivers of the two trains are both blind and deaf, so they can neither see nor hear each other.
- Three OS students suggested the following methods to prevent accidents.
- For each method, explain if the method **prevents collision, whether it is starvation-free and respects the train schedule.**

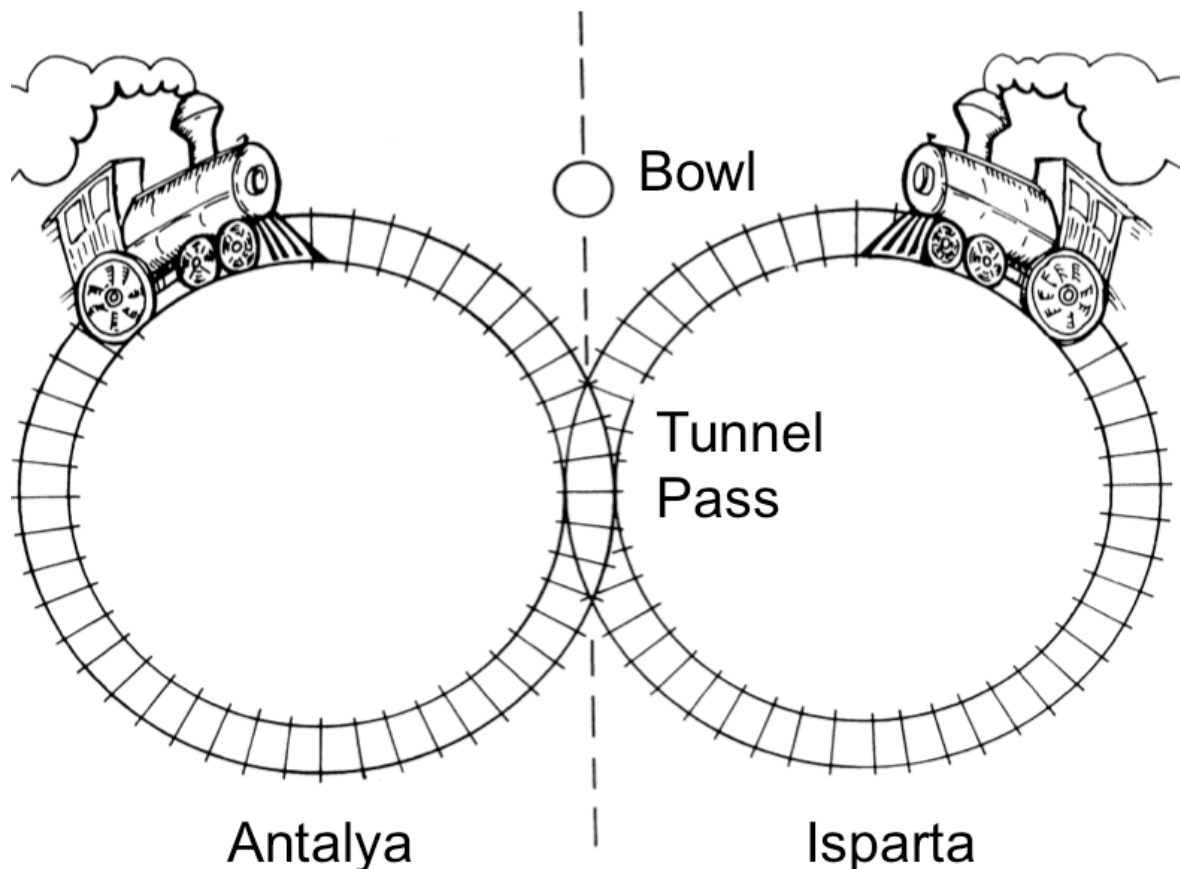
# Method 1



- First method suggests that they set up a large bowl at the entrance to the tunnel.
- Before entering the tunnel, a driver must stop his train, walk over to the bowl, and reach into it to see if it contains a rock.
- If the bowl is empty, the driver finds a rock and drops it in the bowl, indicating that his train is entering the tunnel;
- Once his train has cleared the tunnel, he must walk back to the bowl and remove his rock, indicating that the tunnel is no longer being used. Finally, he walks back to the train and continues down the line.
- If a driver arriving at the tunnel finds a rock in the bowl, he leaves the rock there; he repeatedly takes a nap and rechecks the bowl until he finds it empty. Then he drops a rock in the bowl and drives his train into the pass.

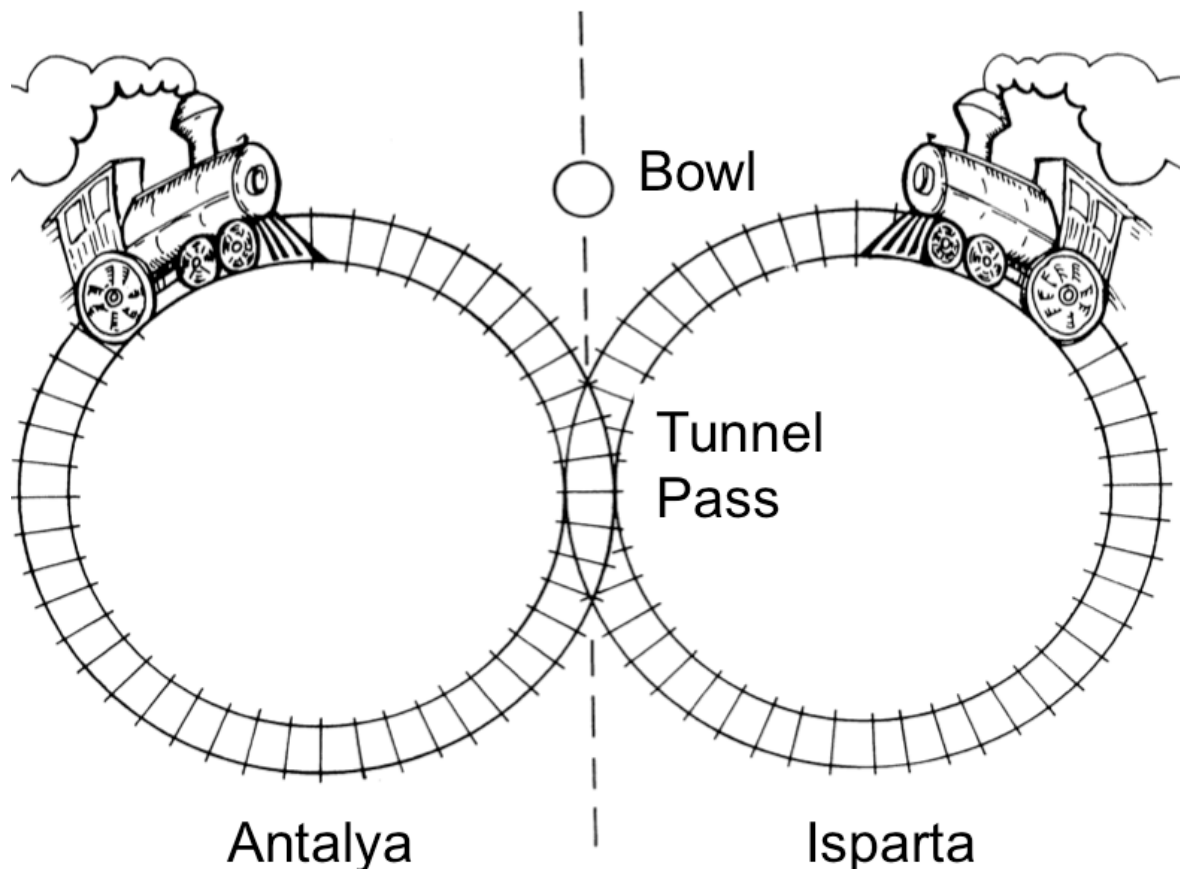


## Method 2



- This method uses the bowl in a different way.
- The driver from the Isparta train must wait at the entry to the pass until the bowl is empty, drive through the pass and walk back to put a rock in the bowl. The driver from Antalya must wait at the entry until the bowl contains a rock, drive through the pass and walk back to remove the rock from the bowl.

# Method 3

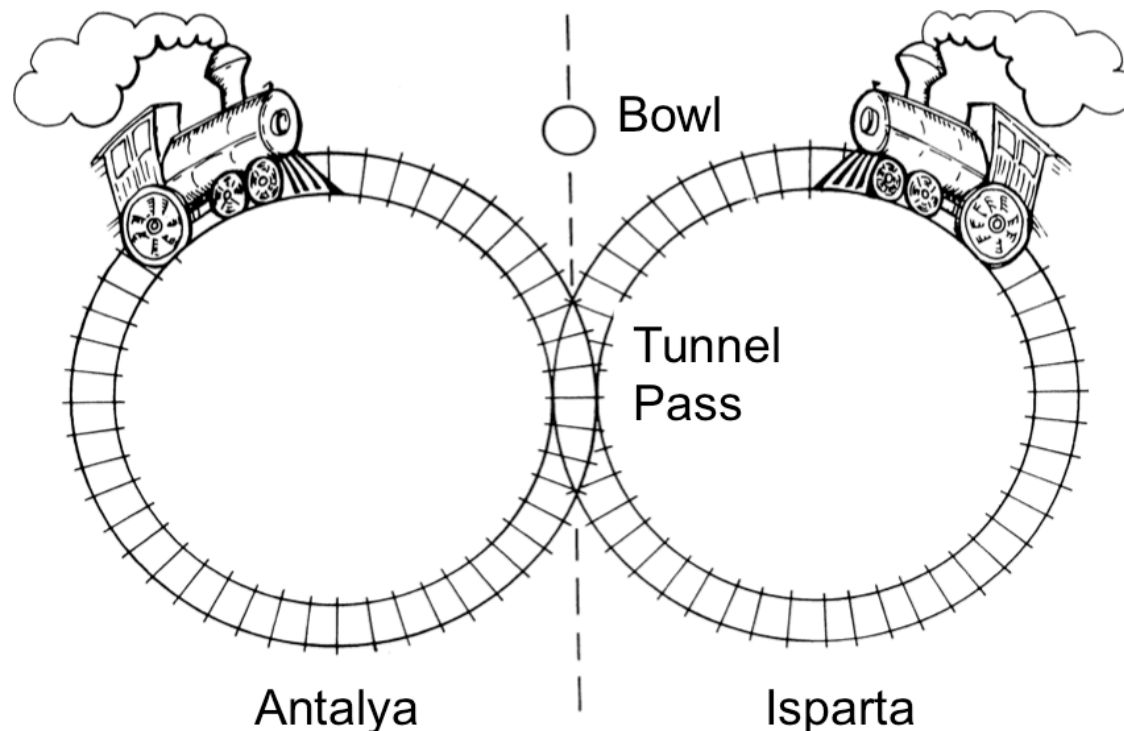


The third method suggests to use two bowls, one for each driver:

- When a driver reaches the entry, he first drops a rock in his bowl, then checks the other bowl to see if it is empty.
- If he finds a rock in the other bowl, he goes back to his bowl and removes his rock. Then he takes a nap, again drops a rock in his bowl and re-checks the other bowl, and so on, until he finds the other bowl empty.
- If the bowl is empty, then he drives his train through the tunnel pass. Stops and walks back to remove his rock.

# Your suggestion

- Suggest a method that works
  - Collision free
  - Starvation free
  - Respects the train schedule



# Attempt 3: Peterson's Solution

Combined shared variables of algorithms 1 and 2.

```
bool flag[0] = false;
bool flag[1] = false;
int turn;
```

```
P0: flag[0] = true;
P0: turn = 1;

while (flag[1] && turn == 1) {
    // busy wait }

// critical section

flag[0] = false;

//remainder section
```

```
P1: flag[1] = true;
P1: turn = 0;

while (flag[0] && turn == 0) {
    // busy wait }

// critical section

flag[1] = false;

//remainder section
```

Meets all three requirements; solves the critical-section problem for two processes.

# Peterson's Solution

- Peterson solution is **not correct** on today's modern computers
  - Because update to turn is not specified as **atomic**
  - We have **caches and multiple copies** of the same data (turn variable) in the hardware
- Process 0 may see turn 1
- Process 1 may see turn 0
- Resulting in data race
- We need hardware support for avoiding race conditions.

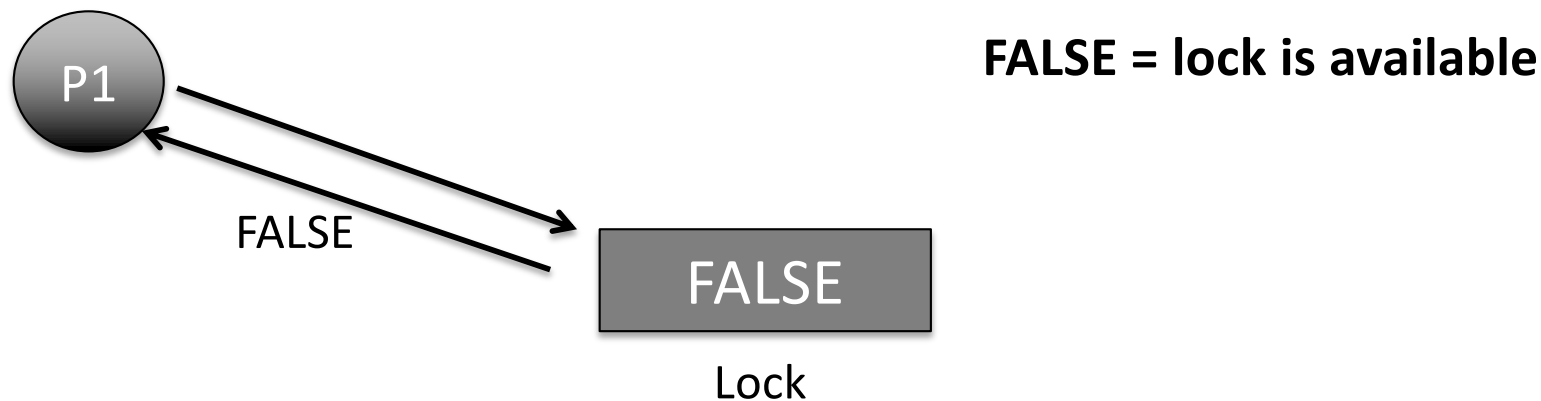
# Atomic Test-and-Set Instruction

- The term *locks* are used to indicate getting a key to enter a critical section.
- The **test-and-set** instruction is an instruction used to write to a memory location and return its old value as a single atomic (i.e., non-interruptible) operation.
- If multiple processes may access the same memory location, and if a process is currently performing a test-and-set, no other process may begin another test-and-set until the first process is done.

```
boolean TestAndSet(boolean *lock) {  
    boolean initial = *lock;  
    *lock = true;  
    return initial;  
}
```

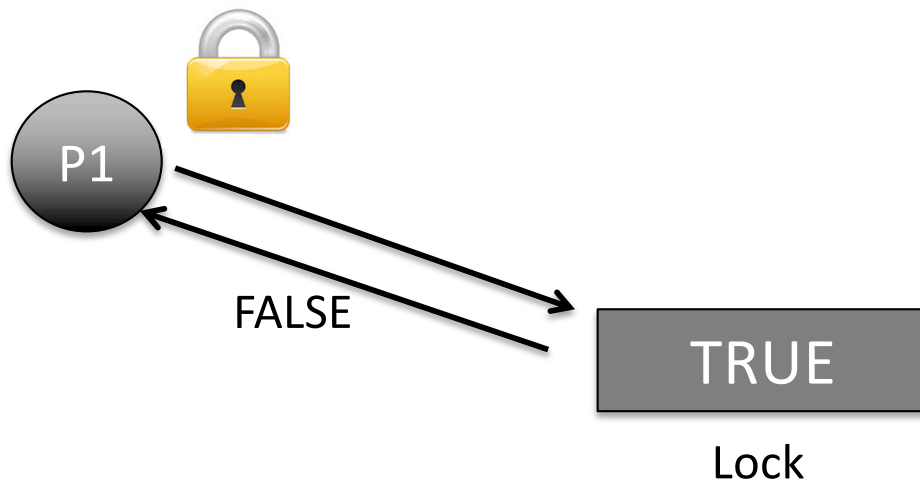
# Test-and-set

- Test-and-set does two things atomically:
  - Test a lock (whose value is returned)
  - Set the lock
- Lock obtained when the return value is FALSE
  - If TRUE, someone already had the lock (and still has it)



# Test-and-set

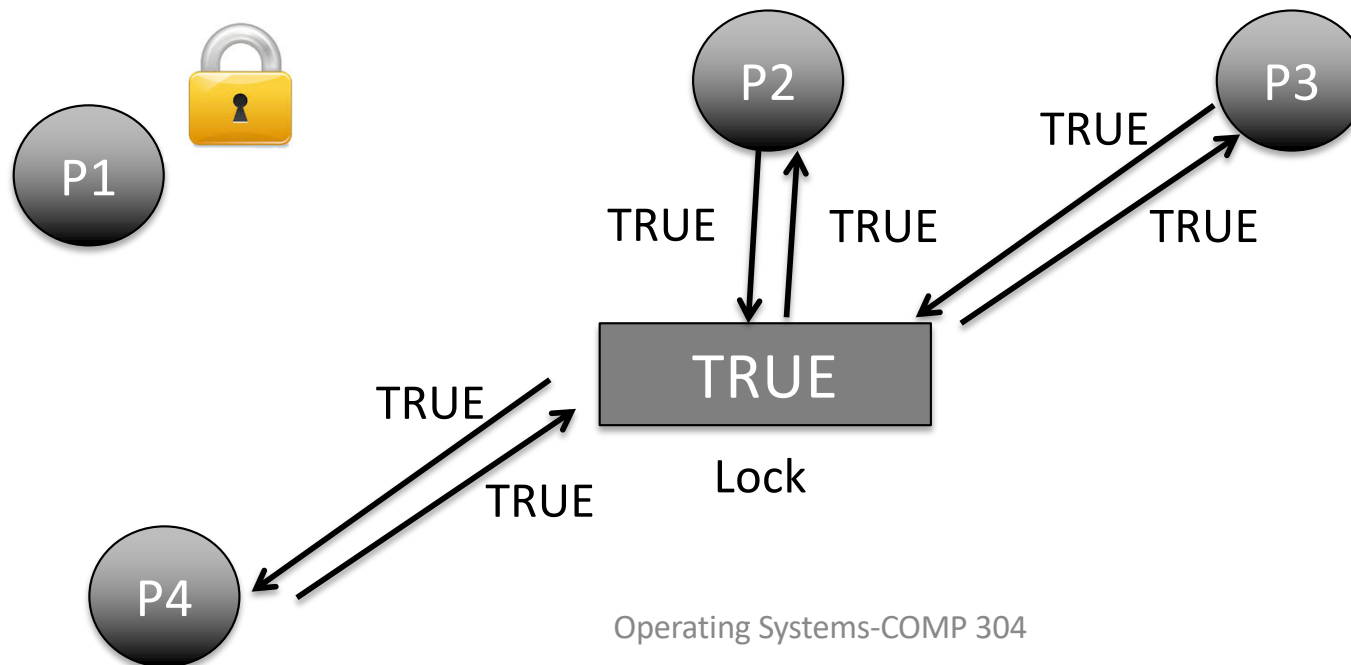
- Test-and-set does two things atomically:
  - Test a lock (whose value is returned)
  - Set the lock
- Lock obtained when the return value is FALSE
  - If TRUE, someone already had the lock (and still has it)





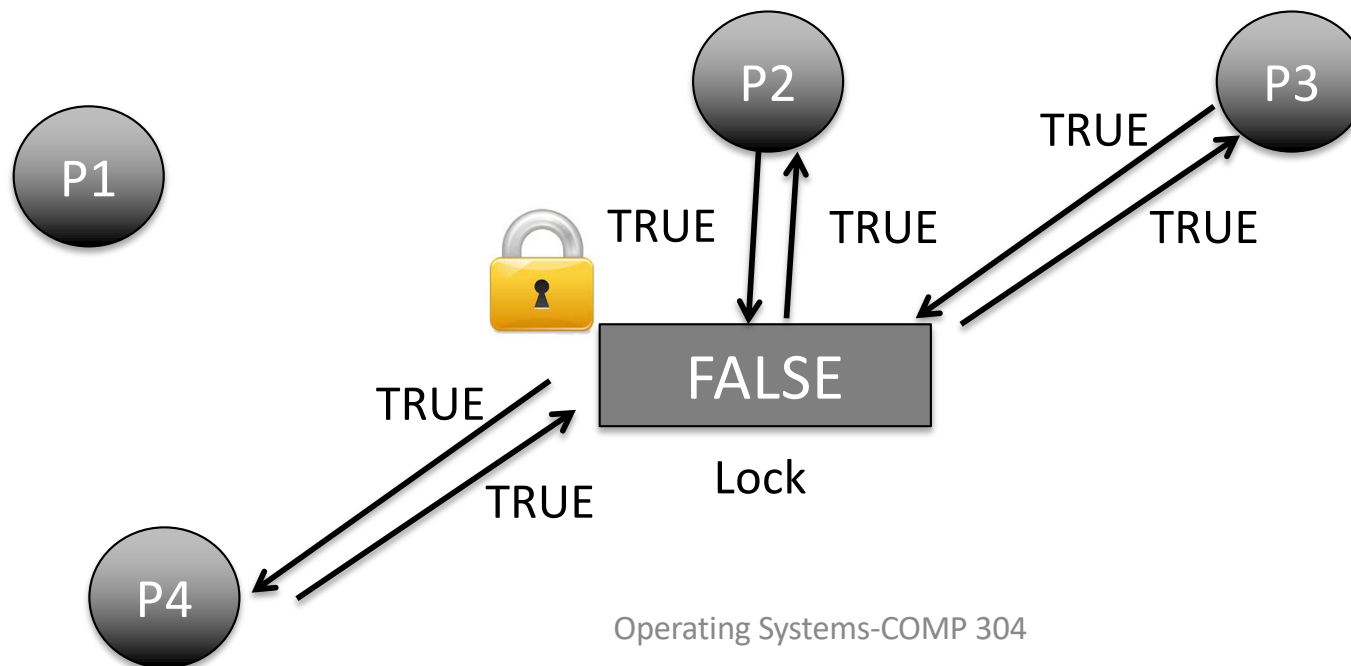
# Test-and-set

- Test-and-set does two things atomically:
  - Test a lock (whose value is returned)
  - Set the lock
- Lock obtained when the return value is FALSE
  - If TRUE, someone already had the lock (and still has it)



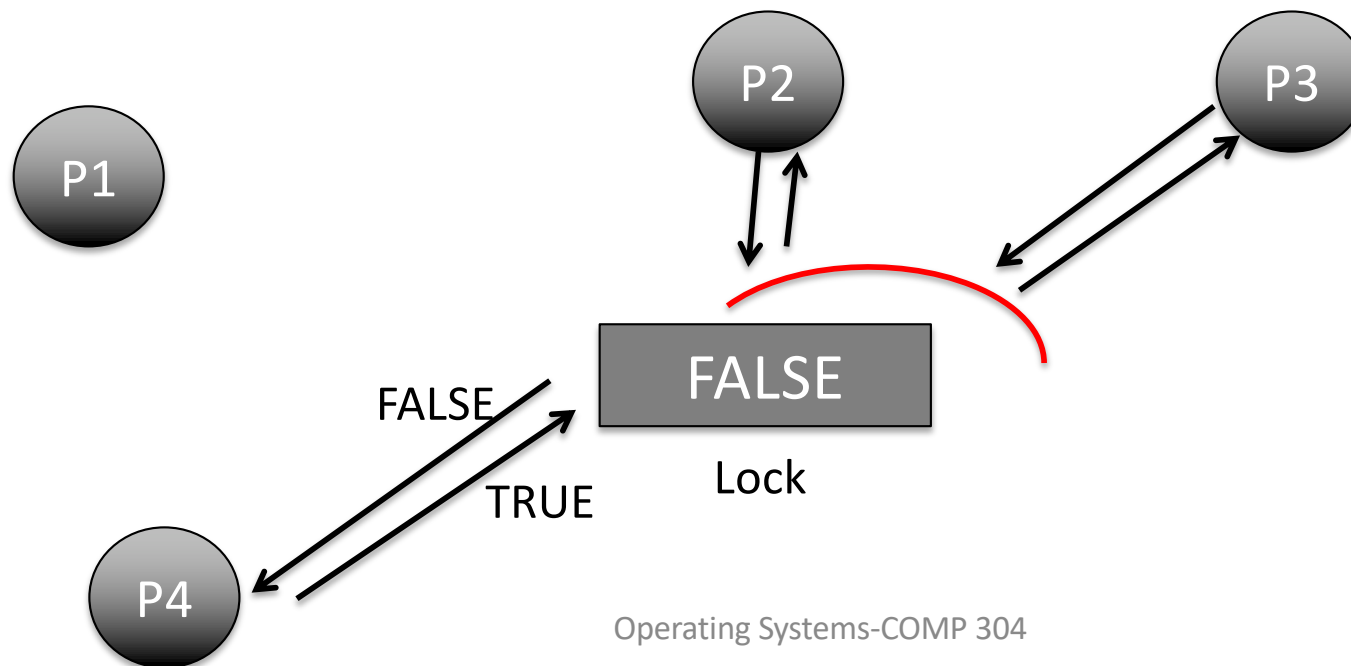
# Test-and-set

- Test-and-set does two things atomically:
  - Test a lock (whose value is returned)
  - Set the lock
- Lock obtained when the return value is FALSE
  - If TRUE, someone already had the lock (and still has it)



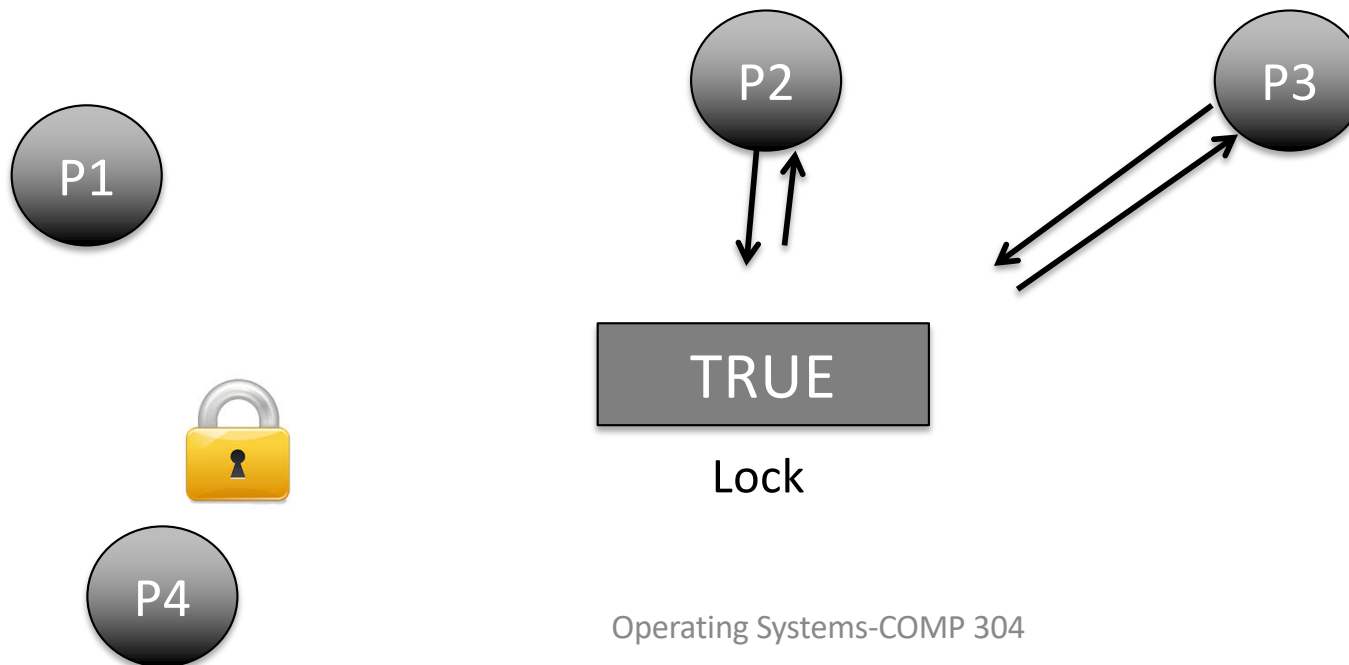
# Test-and-set

- Test-and-set does two things atomically:
  - Test a lock (whose value is returned)
  - Set the lock
- Lock obtained when the return value is FALSE
  - If TRUE, someone already had the lock (and still has it)



# Test-and-set

- Test-and-set does two things atomically:
  - Test a lock (whose value is returned)
  - Set the lock
- Lock obtained when the return value is FALSE
  - If TRUE, someone already had the lock (and still has it)



# Mutual Exclusion with Test-and-Set

- Shared variable:

```
boolean lock = false;
```

- Process  $P_i$

```
do {  
    while (TestAndSet(&lock)) { };  
    critical section  
    lock = false;  
    remainder section  
} while (true);
```

```
boolean TestAndSet(boolean *lock) {  
    boolean initial = *lock;  
    *lock = true;  
    return initial;  
}
```

The calling process obtains the lock if the old value was **False**. It spins until it acquires the lock. When it acquires, the value turns to **True** preventing other processes to acquire the lock.

Must be careful if these approaches are to satisfy a bounded wait condition - must use round robin

# compare\_and\_swap instruction

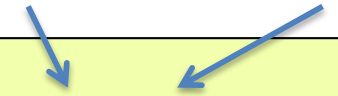
- It compares the contents of a memory location to a given value and, **only if they are the same**, modifies the contents of that memory location to a given new value.
- Done as a **single atomic operation**
- if the value had been updated by another process in the meantime, the write would fail.

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

# Use of compare\_and\_swap instruction

- Shared boolean variable *lock* initialized to FALSE (0)

Expected value      New value



```
do {  
    while (compare_and_swap(&lock, 0, 1) != 0)  
        ; /* do nothing */  
  
    /* critical section */  
  
    lock = FALSE;  
    /* remainder section */  
} while (true);
```

# Mutex Locks

- Previous hardware solutions are complicated and generally inaccessible to application programmers
- OS designers build software tools to solve critical section problem using the support in the hardware
- Enter critical regions by first **acquire()** a lock then **release()** it
  - Boolean variable indicates if lock is available or not
  - Next lecture, we will cover those
- Note that these solutions still use hardware solutions/support underneath



# acquire() and release()

```
do {  
    acquire lock  
    critical section  
    release lock  
    remainder section  
} while (true);
```

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

- Calls to **acquire()** and **release()** must be atomic
  - Usually implemented via *hardware atomic instructions* discussed few slides back.
- This solution requires **busy waiting**
  - This type of lock is called a **spinlock**

# Bounded-waiting Mutual Exclusion with test\_and\_set

```
//Round-robin implementation
Boolean waiting[N];
int j;
//takes on values from 0 to N-1
Boolean key;
```

Each process tries to test\_and\_set for the lock. Only one succeeds with key=false, then it sets its waiting to false. Enters the critical section. Before it exits, it tries to find a process j who is waiting. If j is not i, then process i hands in the lock to process j.

```
do {
    waiting[i] = true;
    key = true;
    while (waiting[i] && key)
        key = test_and_set(&lock);
    waiting[i] = false;

    /* critical section */

    j = (i + 1) % n;
    while ((j != i) && !waiting[j])
        j = (j + 1) % n;
    if (j == i)
        lock = false;
    else
        waiting[j] = false;

    /* remainder section */

} while (true);
```

# Reading

- Read Chapter 6
- Acknowledgments
  - These slides are adapted from
    - Öznur Özkasap (Koç University)
    - Operating System and Concepts (9<sup>th</sup> edition) Wiley
    - Jerry Breecher