Unit 5
Process Synchronization

Main Topics

Background

The Critical-Section Problem

Peterson' s Solution

Synchronization Hardware

Mutex Locks

Semaphores

Classic Problems of Synchronization

Objectives

- *To present the concept of process synchronization.*
- *To introduce the critical-section problem, whose solutions can be used to ensure the consistency of shared data*
- *To explore several tools that are used to solve process synchronization problems*
- *To be familiar with several classical process-synchronization problems*

Background

- *Processes can execute concurrently*
  - *May be interrupted at any time, partially completing execution*
- *Concurrent access to shared data may result in data inconsistency*
- *Maintaining data consistency requires mechanisms to ensure the orderly execution of cooperating processes*

Illustration of the problem:

*Suppose that we wanted to provide a solution to the consumer-producer problem that fills **all** the buffers.*

*We can do so by having an integer **counter** that keeps track of the number of full buffers.*

*Initially, **counter** is set to 0. It is incremented by the producer after it produces a new buffer and is decremented by the consumer after it consumes a buffer.*

Illustration of the problem (producer consumer)

Producer

```
while (true){
   /* produce an item in
      next produced */
 while(counter==BUFFER_SIZE;
     /* do nothing */
   buffer[in]=next_produced;
   in=(in+1);//BUFFER_SIZE
   counter++;
              }
```

Consumer

```
while (true){
   while (counter == 0;
    /* do nothing */
   next_consumed=buffer[out];
   out=(out+1);//BUFFER_SIZE
    counter--;
    /*consume the item in
      next consumed */
               }
```

counter++ could be implemented as

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

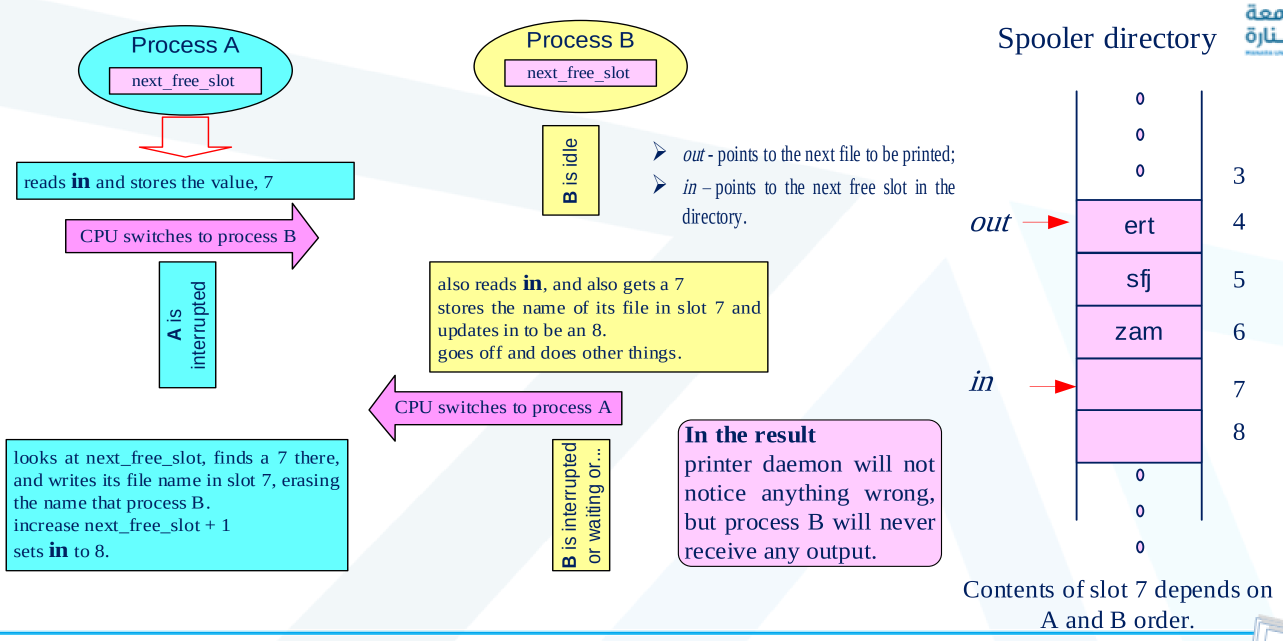
Counter-- could be implemented as

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

Producer Consumer Problem

T_0: √ √	producer	execute	$register_1$ = counter	{$register_1$ = 5}
T_1: √ √	producer	execute √ √ √	$register_1$ = $register_1$+1	{$register_1$ = 6}
T_2: √ √	consumer	execute	$register_2$ = counter	{$register_2$ = 5}
T_3 X	consumer	execute	$register_2$ = $register_2$-1	{$Register_2$ = 4}
T_4:	producer	execute	counter= $register_1$	{counter= 6}
T_5:	consumer	execute	counter= $register_2$	{counter= 4}

Read Write Problem



reads **in** and stores the value, 7

CPU switches to process B

- *out* - points to the next file to be printed;
- *in* – points to the next free slot in the directory.

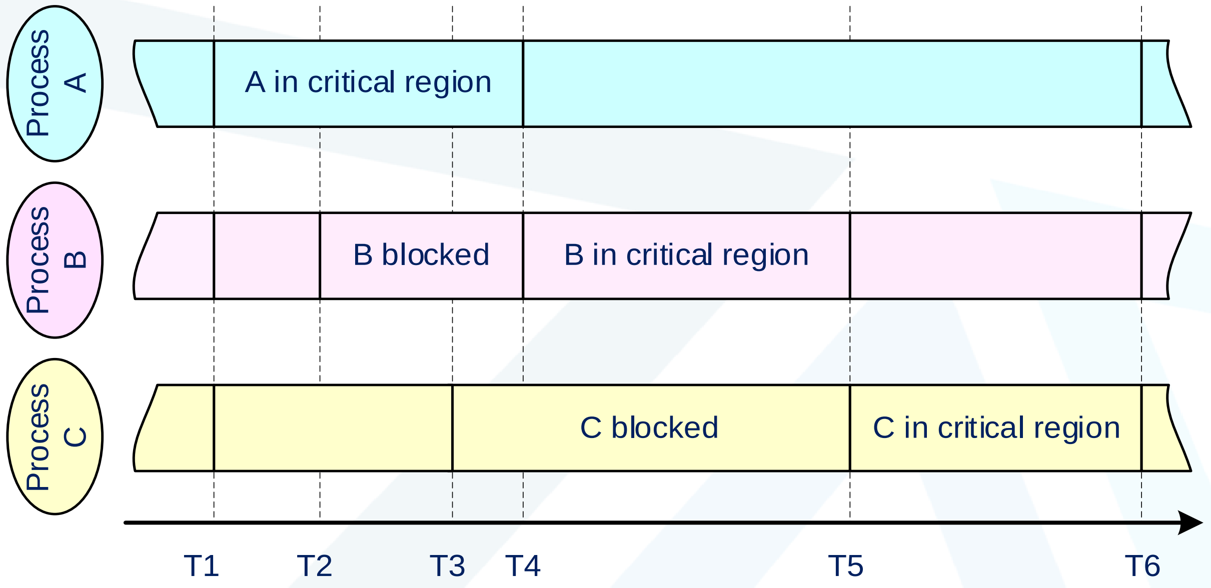
also reads **in**, and also gets a 7
stores the name of its file in slot 7 and updates in to be an 8.
goes off and does other things.

CPU switches to process A

looks at next_free_slot, finds a 7 there, and writes its file name in slot 7, erasing the name that process B.
increase next_free_slot + 1
sets **in** to 8.

In the result
printer daemon will not notice anything wrong, but process B will never receive any output.

Contents of slot 7 depends on A and B order.



Critical Section Problem

- *Consider system of **n** processes {$p_0, p_1, \ldots p_{n-1}$}*
- *Each process has **critical section** segment of code*
  - *Process may be changing common variables, updating table, writing file, etc*
  - *When one process in critical section, no other may be in its critical section*
- ***Critical section problem** is to design protocol to solve this*
- *Each process must ask permission to enter critical section in **entry section**, may follow critical section with **exit section**, then **remainder section***

Simple Tool Solution

- Conceptually, any solution to the critical section problem can be viewed as constructing a simple tool, called a "lock"
- A process must acquire a lock before entering a critical section, and releases the lock when it exits the critical section

```
do
    {
   acquire lock

  critical section

  release lock

  remainder section

   } while (TRUE):
```

Solution to Critical-Section Problem

1. *Mutual Exclusion - If process P_i is executing in its critical section, then no other processes can be executing in their critical sections*

Solution to Critical-Section Problem

2. *Progress - If no process is executing in its critical section and there exist some processes that wish to enter their critical section, then the selection of the processes that will enter the critical section next cannot be postponed indefinitely*

Solution to Critical-Section Problem

3. *Bounded Waiting - A bound must exist on the number of times that other processes are allowed to enter their critical sections after a process has made a request to enter its critical section and before that request is granted*

Critical-Section Handling in OS

Two approaches depending on if kernel is preemptive or non- preemptive

- ***Preemptive*** *– allows preemption of process when running in kernel mode*
- ***Non-preemptive*** *– runs until exits kernel mode, blocks, or voluntarily yields CPU*
  - ✓ *Essentially free of race conditions in kernel mode*

Peterson's Solution

- *The two processes share two variables:*
  - ***int turn;***
  - ***Boolean flag[2]***

Int turn
boolean flag

- *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!*

Peterson's Solution

Process i
```
while  (true)
{
   flag[i] = TRUE;
   turn = j;
   while ( flag[j] && turn == j);
   //CRITICAL SECTION
   flag[i] = FALSE;
   //REMAINDER SECTION
}
```

Process j
```
while  (true)
{
   flag[j] = TRUE;
   turn = i;
   while ( flag[i] && turn == i);
   //CRITICAL SECTION
   flag[j] = FALSE;
   //REMAINDER SECTION
}
```

Int turn 🟨

boolean flag 🟨🟨

Peterson's Solution (Cont.)

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*

Recall Peterson's Solution:

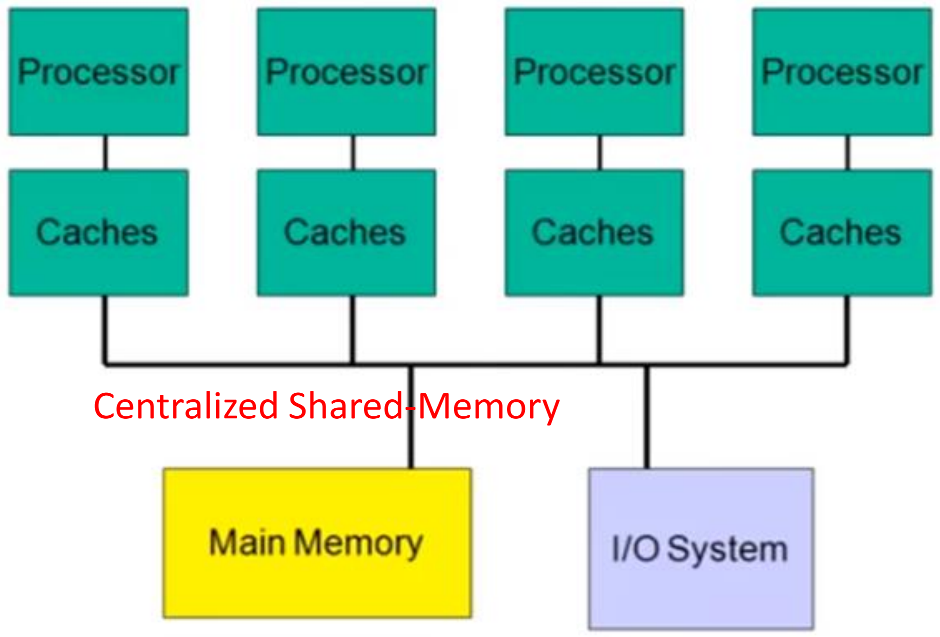
Process i

```
while (true)
{
   flag[i] = TRUE;
   turn = j;
   while ( flag[j] && turn == j);
   //CRITICAL SECTION
   flag[i] = FALSE;
   //REMAINDER SECTION
}
```

Acquire a lock

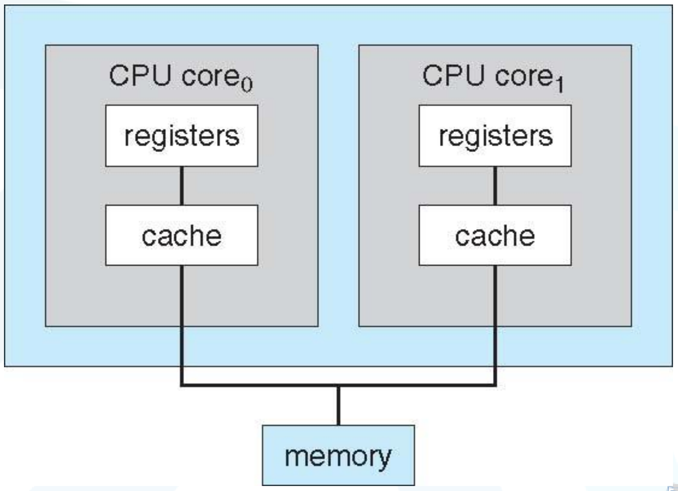
Release the lock

Symmetric Multiprocessing Architecture

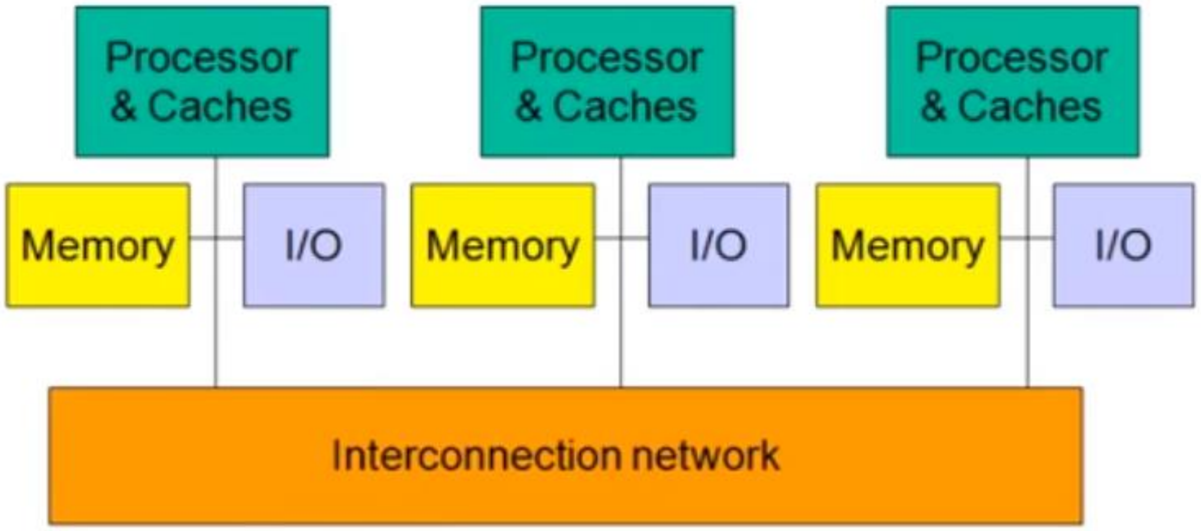


A Dual-Core Design

- Multi-chip and **multicore**
- Systems containing all chips
  - Chassis containing multiple separate systems



Distributed Memory Multiprocessors



Synchronization Hardware

- *As discussed, software-based solutions (like Peterson's Solution) are not guaranteed to work on modern computer architectures*
- *Many systems provide hardware support for synchronization*
  - *Uniprocessor systems*
  - *Multiprocessor systems*

Uniprocessor &Multiprocessor Systems

- *Disable interrupts*
  - *Currently running code would execute without preemption*
  - *Generally too inefficient on multiprocessor systems*

Memory Barriers

- *Memory model are the memory guarantees that a computer architecture makes to application programs.*
- *Memory models may be either:*
  - ***Strongly ordered*** *– where a memory modification of one processor is immediately visible to all other processors.*
  - ***Weakly ordered*** *– where a memory modification of one processor may not be immediately visible to all other processors.*
- *A memory barrier is an instruction that forces any change in memory to be propagated (made visible) to all other processors.*

Recall Peterson's Solution

- *Two threads share the data:*

```
boolean flag = false; int x = 0;
```

- *Thread 1 performs*

```
while (!flag) ;
print x
```

- *Thread 2 performs*

```
x = 100;
flag = true
```

- *What is the expected output?*

Recall Peterson's Solution

- After Instruction Reordering . 100 is the expected output.
- However, the operations for Thread 2 may be reordered:

  flag = true; x = 100;

- If this occurs, the output may be 0!
- The effects of instruction reordering in Peterson's Solution
- This allows both processes to be in their critical section at the same time!

Solution using Memory Barrier

- To ensure Thread 1 outputs 100:
- Thread 1 now performs

```
while (!flag)
memory_barrier();
print x
```

- Thread 2 now performs

```
x = 100;
memory_barrier();
flag = true
```

Hardware Instructions

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

- *Test-and-Set instruction*
- *Compare-and-Swap instruction*

Solution to Critical-section Problem Using Locks

```
do
    {
  acquire lock
    critical section
  release lock
    remainder section
  } while (TRUE):
```

- *Modern machines provide special atomic hardware instructions*
- *Atomic = non-interruptible*
- *Either test memory word and set value*
- *Or swap contents of two memory words*

Solution to Critical-section Problem Using Locks

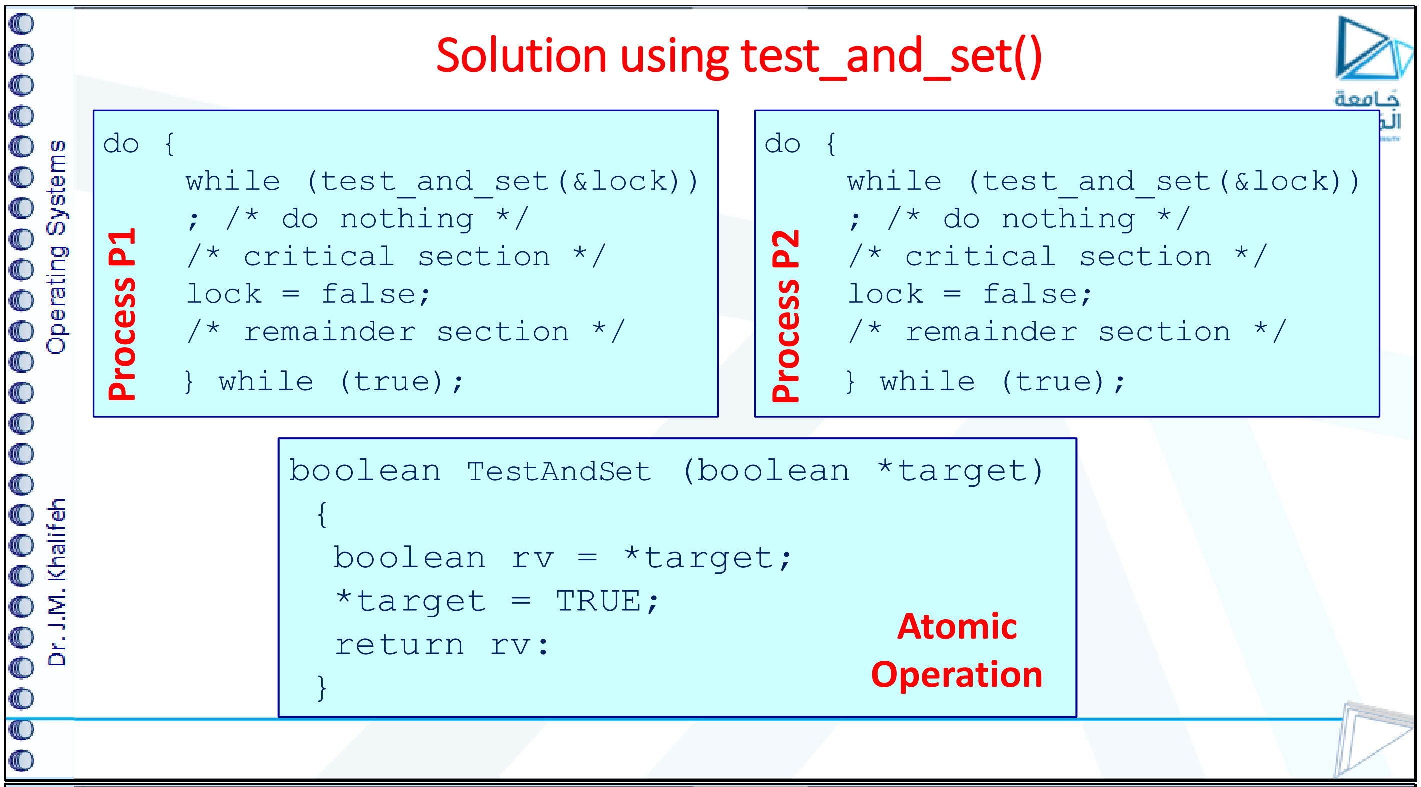
```
do
    {
   acquire lock
      critical section
      release lock
      remainder section
   } while (TRUE):
```

- *Uniprocessors – could disable interrupts*
- *Currently running code would execute without preemption*
- *Generally too inefficient on multiprocessor systems*
- *Operating systems using this not broadly scalable*

Definition of test_and_set Instruction

```
boolean TestAndSet (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".*



Solution using test_and_set()

Process P1

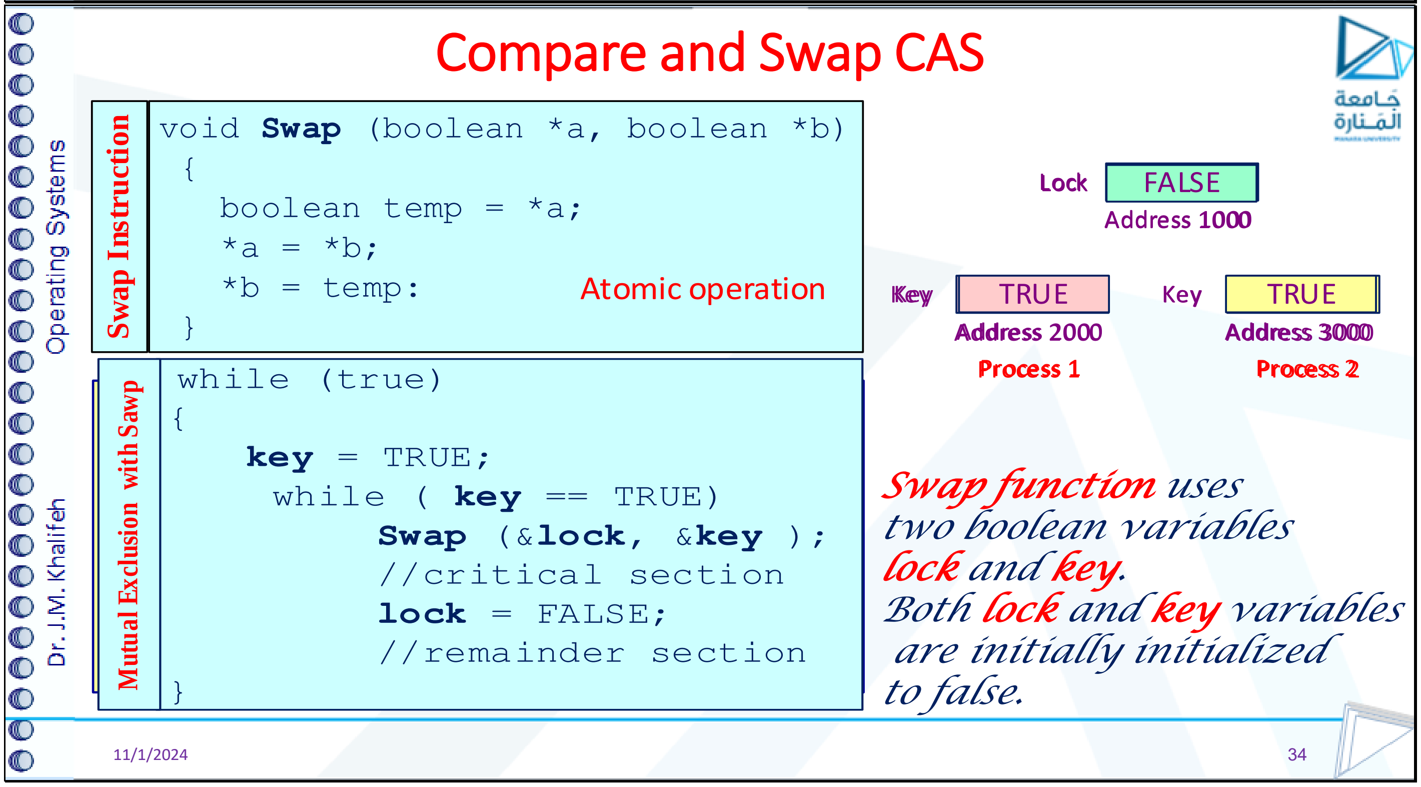
```
do {
    while (test_and_set(&lock))
    ; /* do nothing */
    /* critical section */
    lock = false;
    /* remainder section */
} while (true);
```

Process P2

```
do {
    while (test_and_set(&lock))
    ; /* do nothing */
    /* critical section */
    lock = false;
    /* remainder section */
} while (true);
```

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

Atomic Operation



Compare and Swap CAS

Swap Instruction

```
void Swap (boolean *a, boolean *b)
 {
   boolean temp = *a;
   *a = *b;
   *b = temp:
 }
```

Atomic operation

Mutual Exclusion with Sawp

```
while (true)
{
    key = TRUE;
     while ( key == TRUE)
          Swap (&lock, &key );
          //critical section
          lock = FALSE;
          //remainder section
}
```

Lock FALSE
Address 1000

Key TRUE
Address 2000
Process 1

Key TRUE
Address 3000
Process 2

Swap function uses two boolean variables lock and key. Both lock and key variables are initially initialized to false.

Solution using test_and_set()

Process P1

```
do {
     while (test_and_set(&lock))
     ; /* do nothing */
     /* critical section */
     lock = false;
     /* remainder section */
    } while (true);
```

Acquire a lock

Release the lock

Compare and Swap CAS

Mutual Exclusion with Sawp

```
while (true)
{
    key = TRUE;
     while ( key == TRUE)
         Swap (&lock, &key );
         //critical section
         lock = FALSE;
         //remainder section
}
```

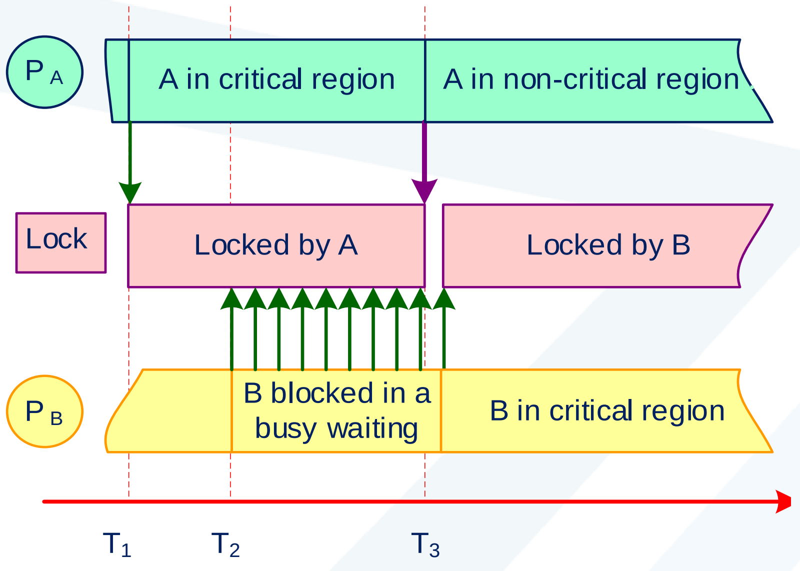
Acquire a lock

Release the lock

Mutex Locks

- Protect a critical section by first **`acquire()`** a lock then **`release()`** the lock
  - Boolean variable indicating if lock is available or not
- Calls to **`acquire()`** and **`release()`** must be atomic
  - Usually implemented via hardware atomic instructions
- But this solution requires **busy waiting**
  - This lock therefore called a **spinlock**

Mutex Locks



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

```
release() {
   available=true;
    }
```

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

Semaphore

Semaphore

- *Synchronization tool that provides more sophisticated ways (than Mutex locks) for process to synchronize their activities.*
- *Semaphore S – **integer variable** can only be accessed via two indivisible (**atomic**) operations.*

Semaphore as General Synchronization Tool

```
wait (S)
{
    while S <= 0
    ; // no-op
     S--;
}
```

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

- A semaphore is a key that the code acquires in order to continue execution
- If a semaphore is already in use, the requesting task is suspended until the semaphore is released by its current owner
- In other words, the requesting tasks says: "Give me the key. If someone else is using it, I am willing to wait for it!"

V stands for verhogen ("increase"),
P stands fro probeer ("try")

Semaphore as General Synchronization Tool

```
wait (S)
{
    while S <= 0
    ; // no-op
     S--;
}
```

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

```
Semaphore S;
//S initialized to 1
wait (S);
    {Critical
      Section}
signal (S);
```

Semaphore with no Busy waiting

```
wait(S)
{
  value--;
  if (value < 0)
  {
    /*add this process
    to waiting queue*/
    block();}
}
Signal (S)
{
  value++;
  if (value <= 0)
  {
   /*remove a process P
   from the waiting queue*/
   wakeup(P);}
}
```

- While a process is in its critical section, any other process that tries to enter its critical section must loop continuously in the entry code(***busy waiting***).
- Rather than ***busy waiting***, the process can ***block*** itself.
- The `block()` operation places a process into a waiting queue associated with the semaphore, and the state of the process is switched to the waiting state.
- Then, control is transferred to the CPU scheduler, which selects another process to execute.
- A process should be restarted when some other process executes a ***`signal()`*** operation.
- The process is restarted by a ***`wakeup()`*** operation
- The process is then placed in the ready queue.

Deadlock

P_0	P_1
`wait(S);`	`wait(Q);`
`wait(Q);`	`wait(S);`
`...`	`...`
`signal(S);`	`signal(Q);`
`signal(Q);`	`signal(S);`

❖*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*

Starvation

- ❖ *Starvation – indefinite blocking*
  - ➢ *A process may never be removed from the semaphore queue in which it is suspended*

Priority Inversion

- ❖ *Priority Inversion – Scheduling problem when lower-priority process holds a lock needed by higher-priority process*
  - ➢ *Solved via priority-inheritance protocol*