Operating Systems

Dr. J.M. Khalifeh

Unit 2
Process Synchronization

Operating Systems

Dr. J.M. Khalifeh

Main Topics

Review

Semaphores

Classic Problems of Synchronization

Operating Systems

Dr. J.M. Khalifeh

Objectives

- To be familiar with several classical process-synchronization problems

Operating Systems

Dr. J.M. Khalifeh

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.

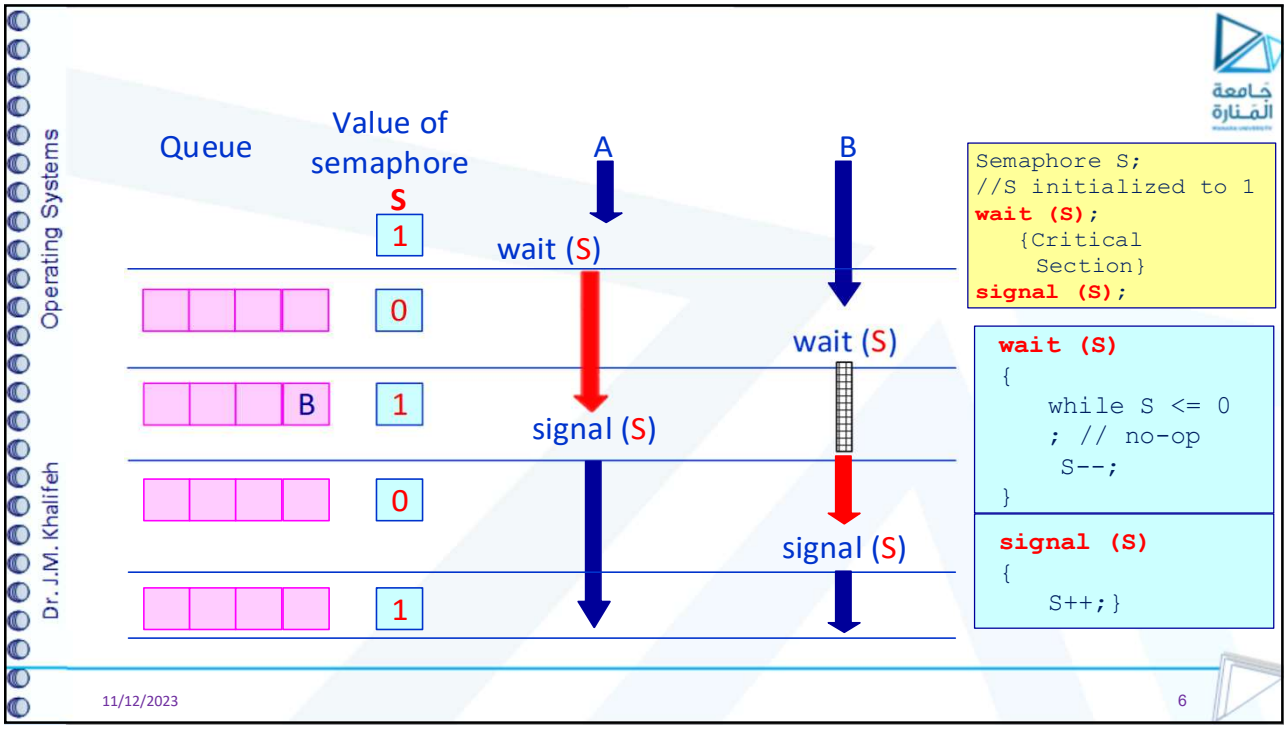
Semaphore as General Synchronization Tool

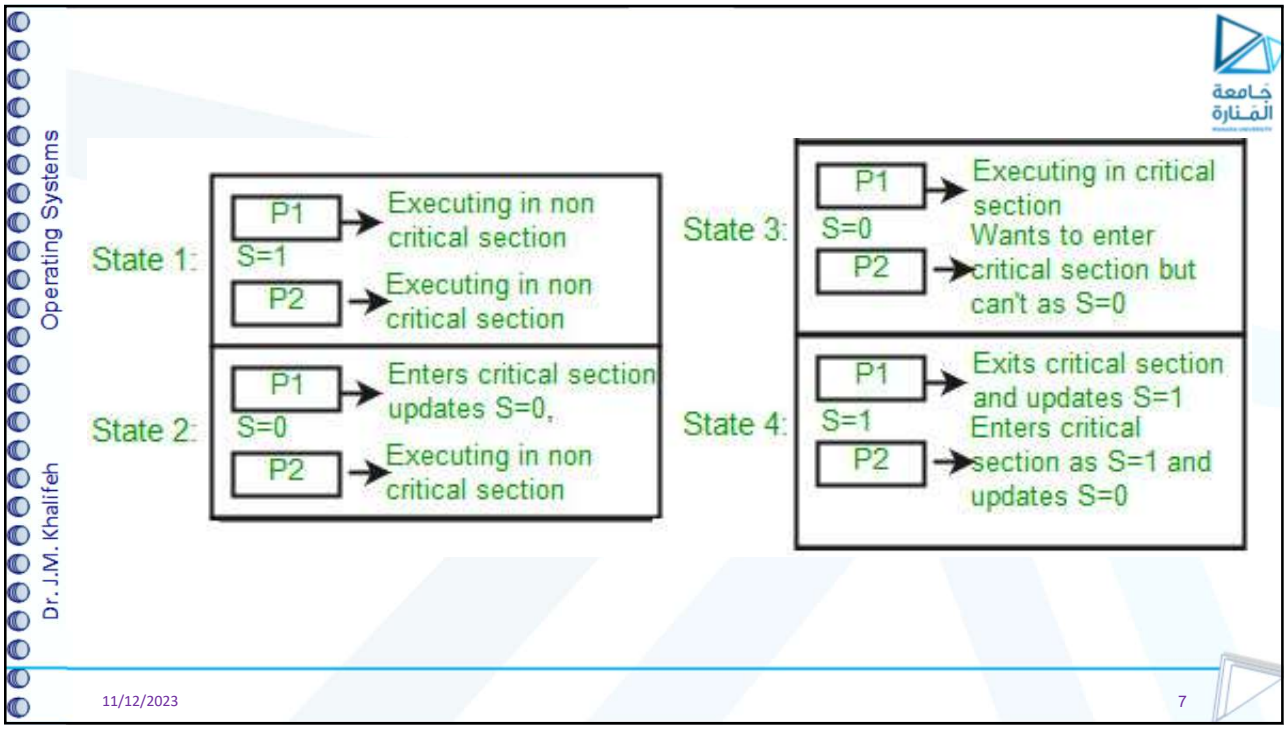
Operating Systems — Dr. J.M. Khalifeh

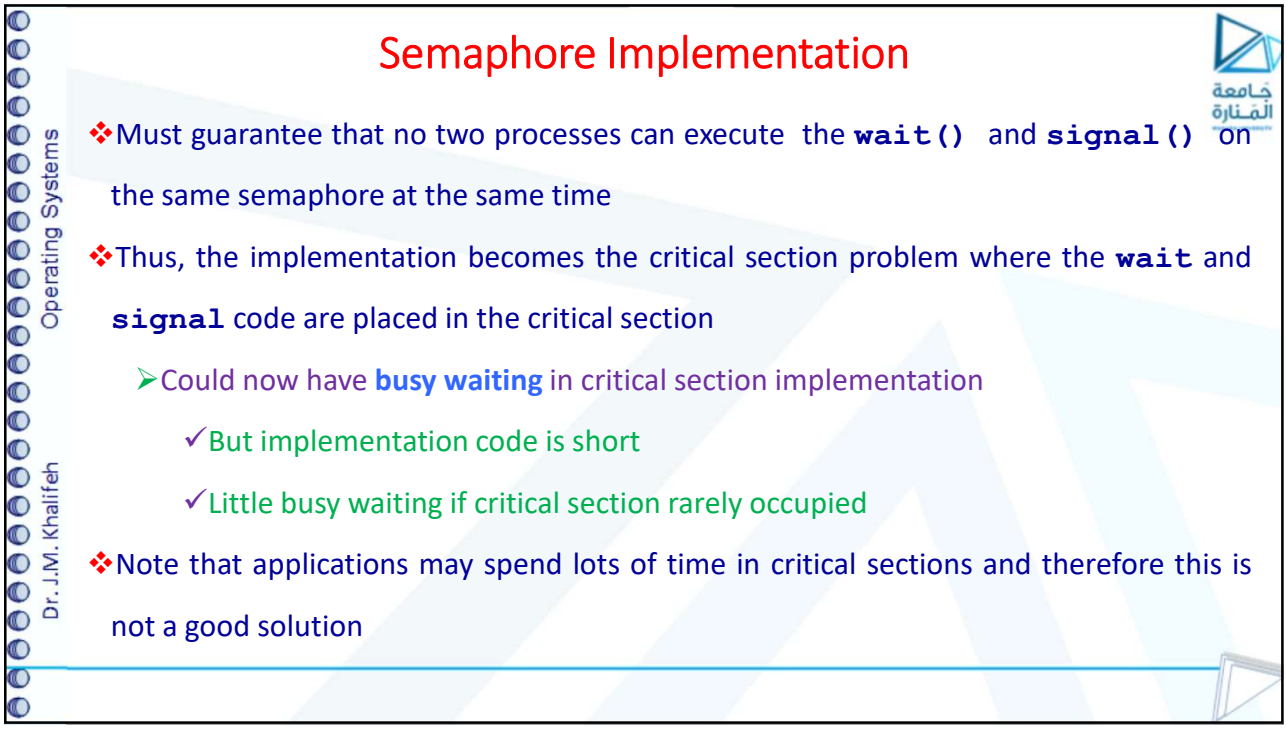
```
wait (S)
{
    while S <= 0
    ; // no-op
     S--;
}
```

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

- Synchronization tool that does not require busy waiting
- Semaphore *S* – integer variable that can be accessed only by two standard operations modify **S: `wait()`** and **`signal()`**
  - Originally called **P()** and **V()**
- Less complicated
- Can only be accessed via two indivisible (atomic) operations




Operating Systems
Dr. J.M. Khalifeh
State 1:
P1
S=1
Executing in non critical section
P2
Executing in non critical section
State 2:
P1
S=0
Enters critical section updates S=0,
P2
Executing in non critical section
State 3:
P1
S=0
Executing in critical section
Wants to enter
P2
critical section but can't as S=0
State 4:
P1
S=1
Exits critical section and updates S=1
Enters critical
P2
section as S=1 and updates S=0
11/12/2023
7


Operating Systems
Dr. J.M. Khalifeh
Semaphore Implementation
❖Must guarantee that no two processes can execute the wait() and signal() on the same semaphore at the same time
❖Thus, the implementation becomes the critical section problem where the wait and signal code are placed in the critical section
➢Could now have busy waiting in critical section implementation
✓But implementation code is short
✓Little busy waiting if critical section rarely occupied
❖Note that applications may spend lots of time in critical sections and therefore this is not a good solution

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 and Starvation

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

- **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**

Operating Systems
Dr. J.M. Khalifeh

❖Interchanging the order in which the wait() and signal():

```
signal(mutex);
...
critical section
...
wait(mutex);
```

In this situation, several processes may be executing in their critical sections simultaneously, violating the mutual-exclusion requirement.

❖Suppose that a program replaces signal(mutex) with wait(mutex).

```
wait(mutex);
...
critical section ...
wait(mutex);
```

In this case, the process will permanently block on the second call to wait(), as the semaphore is now unavailable.

❖Suppose that a process omits the wait(mutex), or the signal(mutex), or both.

In this case, either mutual exclusion is violated or the process will permanently block.

Operating Systems
Dr. J.M. Khalifeh

Classical Problems of Synchronization

❖Classical problems used to test newly-proposed synchronization schemes

- ➢Bounded-Buffer Problem
- ➢Readers and Writers Problem
- ➢Dining-Philosophers Problem

Operating Systems
Dr. J.M. Khalifeh

Bounded-Buffer Problem "Problem Statement"

Buffer of n Slots

- We have a buffer of fixed size.
- A producer can produce an item and can place in the buffer.
- A consumer can pick items and can consume them.
- We need to ensure that when a producer is placing an item in the buffer, then at the same time consumer should not consume any item.
- In this problem, buffer is the critical section.

Operating Systems
Dr. J.M. Khalifeh

Bounded-Buffer Problem

- ***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

Some of the issues that might arise in the Producer-Consumer

- The producer should generate data only if the buffer is not full. When the buffer is filled, the producer should not be able to add any more data to it.
- When the buffer is not empty, the consumer can consume the data. The consumer should not be able to take any data from the buffer if it is empty.
- The buffer should not be used by both the producer and the consumer at the same time.

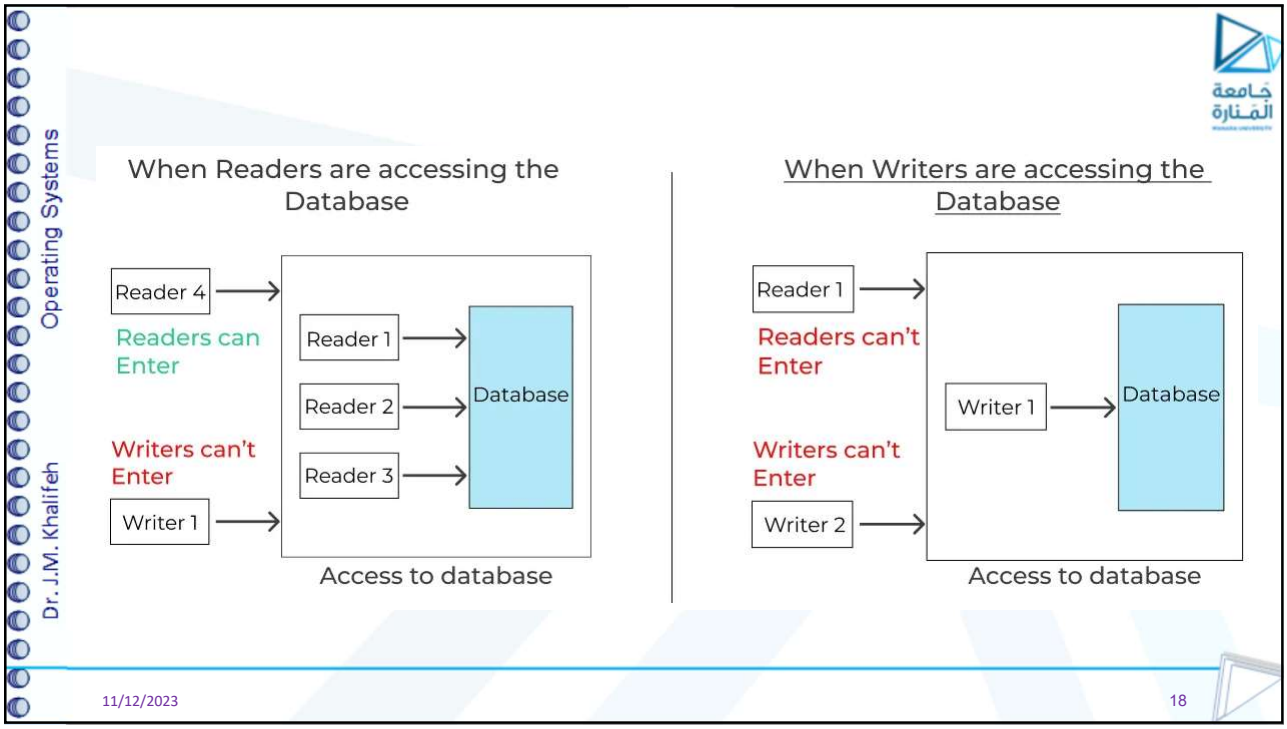
Bounded Buffer Problem (Cont.)
The structure of the producer and consumer processes

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

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

Readers-Writers Problem

- 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
- Several variations of how readers and writers are considered – all involve some form of priorities
- Shared Data
  - Data set
  - Semaphore `rw_mutex` initialized to 1
  - Semaphore `mutex` initialized to 1
  - Integer `read_count` initialized to 0



Readers-Writers Problem (Cont.)

❖ The structure of a reader process

```
do {
    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);
} while (true);
```

❖ The reader requests entry to the critical section

❖ If permitted,

- ➢ it increments the number of readers within the critical section. If this reader is the first to enter, the wrt semaphore is locked, preventing writers from entering if any other reader is present
- ➢ it then signals mutex, indicating that any new reader may enter while others are currently reading
- ➢ it leaves the critical section after reading. When departing, it checks to see whether there are any more readers within and if there are, it signals the semaphore "wrt," indicating that the writer can now enter the critical region

❖ If it is not permitted, it will continue to wait

Operating Systems
Dr. J.M. Khalifeh

```
do {
  /* The reader requests entry to the critical section*/
  wait(mutex);
 /* Now,the number of readers has incremented by 1*/
  readCount++;
 /* there is minimum one reader in the critical section, this ensures that no writer can enter if there is even one reader, hence readers are given preference here*/
  if (readCount==1)
  wait(wrt);
 /* other readers can enter while the current reader is inside the critical section*/
```

```
signal(mutex);
 /* current reader performs reading*/
  wait(mutex); /* a reader wants to exit*/
  readCount--;
 /* i.e., no reader is left in the critical section*/
  if (readCount == 0)
   signal(wrt); /* writers can enter now*/
  signal(mutex); /* reader exits*/
} while(true);
```

Operating Systems
Dr. J.M. Khalifeh

Readers-Writers Problem-Writer process

```
do {
   /* the writer requests entry to the critical section*/
   wait(wrt);
   /* performs the write*/
   /* exits the critical section*/
   signal(wrt);
} while(true);
```

Operating Systems
Dr. J.M. Khalifeh

Readers-Writers Problem Variations

- ***First*** variation – no reader kept waiting unless writer has permission to use shared object
- ***Second*** variation – once writer is ready, it performs the write ASAP
- Both may have starvation leading to even more variations
- Problem is solved on some systems by kernel providing reader-writer locks

Dining-Philosophers Problem

- 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

Dining-Philosophers Problem Algorithm

- The structure of Philosopher *i*:

```
do {
    wait (chopstick[i] );
     wait (chopStick[ (i + 1) % 5] );

                //  eat

     signal (chopstick[i] );
     signal (chopstick[ (i + 1) % 5] );

                //  think

} while (TRUE);
```

- What is the problem with this algorithm?

Dining-Philosophers Problem Algorithm (Cont.)

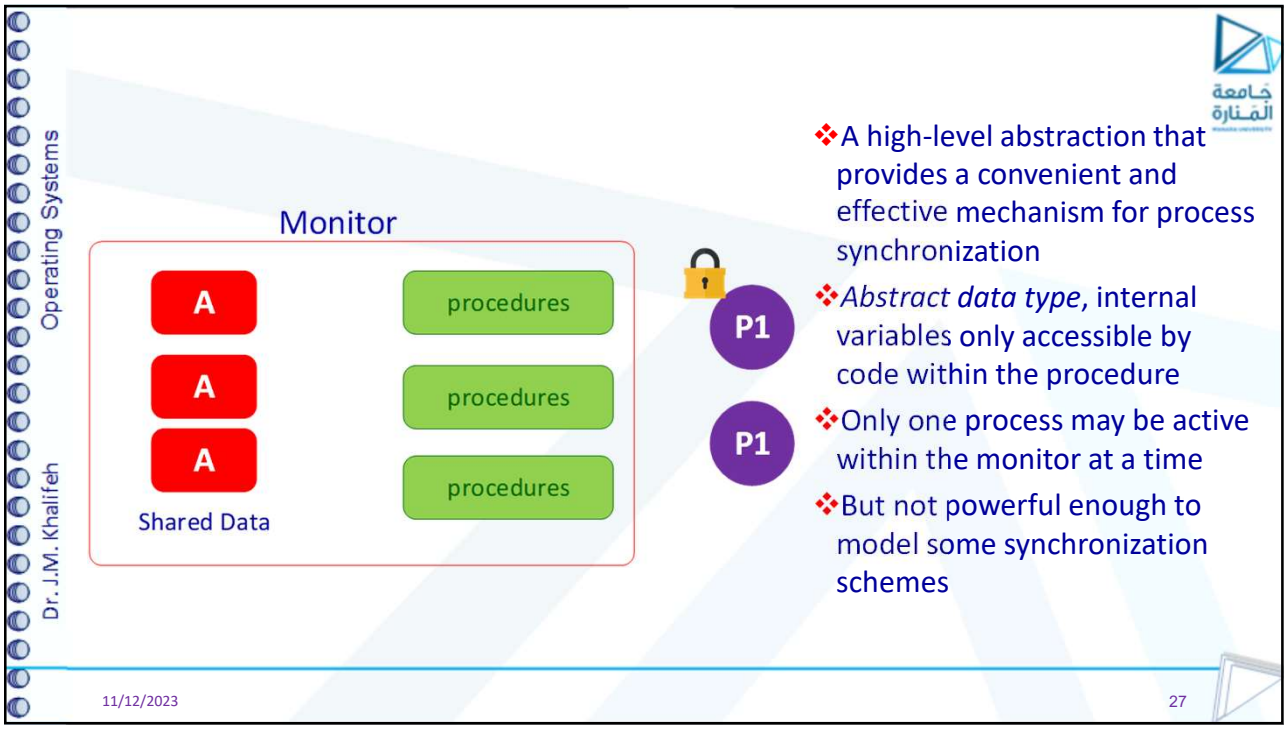
Operating Systems
Dr. J.M. Khalifeh

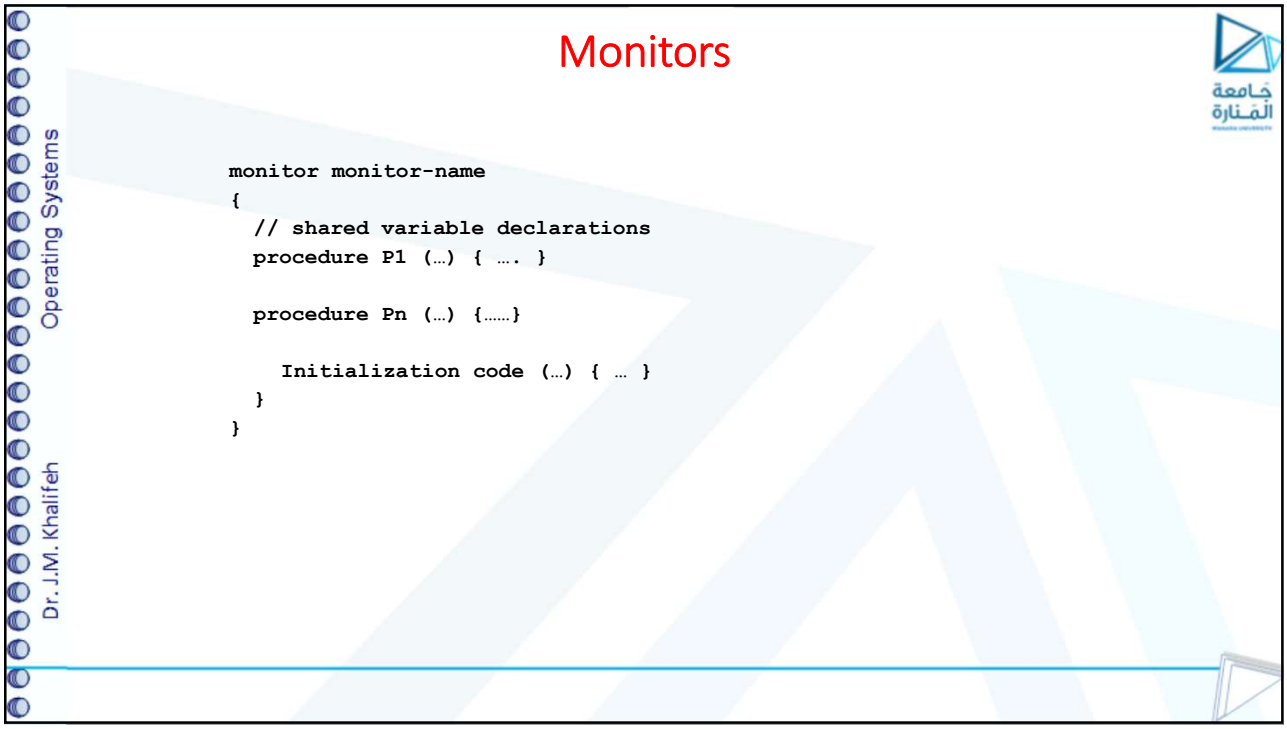
- Deadlock handling
  - Allow at most 4 philosophers to be sitting simultaneously at the table.
  - Allow a philosopher to pick up the forks only if both are available (picking must be done in a critical section.
  - Use an asymmetric solution -- an odd-numbered philosopher picks up first the left chopstick and then the right chopstick. Even-numbered philosopher picks up first the right chopstick and then the left chopstick.

Problems with Semaphores

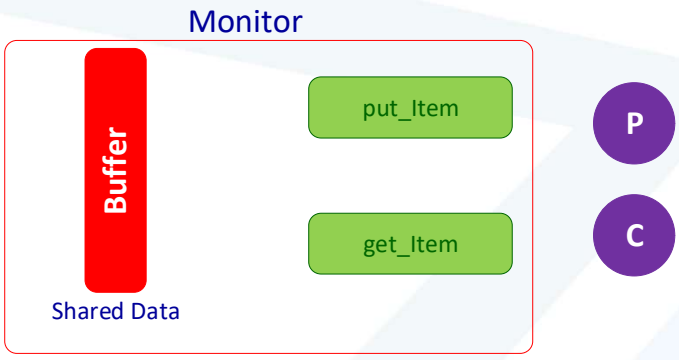
Operating Systems
Dr. J.M. Khalifeh

- Incorrect use of semaphore operations:
  - signal (mutex) …. wait (mutex)
  - wait (mutex) … wait (mutex)
  - Omitting of wait (mutex) or signal (mutex) (or both)
- Deadlock and starvation are possible.


Operating Systems
Dr. J.M. Khalifeh
Monitor
A
A
A
Shared Data
procedures
procedures
procedures
P1
P1
A high-level abstraction that provides a convenient and effective mechanism for process synchronization
Abstract data type, internal variables only accessible by code within the procedure
Only one process may be active within the monitor at a time
But not powerful enough to model some synchronization schemes
11/12/2023
27

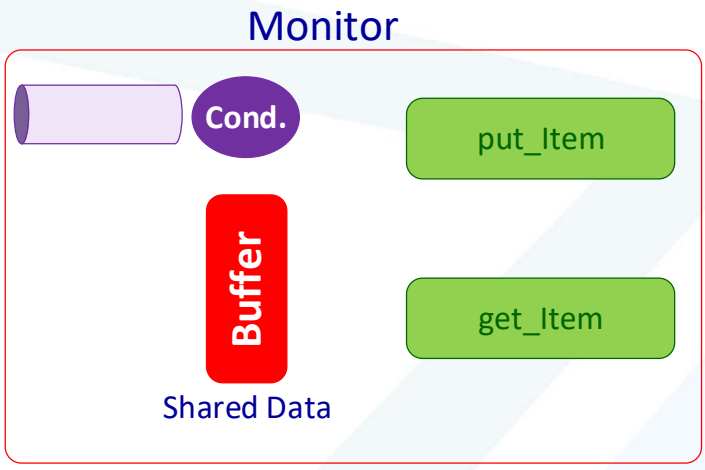

Operating Systems
Dr. J.M. Khalifeh
Monitors
monitor monitor-name
{
// shared variable declarations
procedure P1 (…) { …. }
procedure Pn (…) {……}
Initialization code (…) { … }
}
}

Problem with using monitors



- If consumer finds no other process in the monitor, it will enter.
- If there are no item in the buffer, the consumer will enter the monitor and get stuck there, preventing the producer to enter the monitor.

Conditional Variables



- Conditional variables provides synchronization inside monitors.
- Three operations can be performed:
  - wait()
  - signal()
  - broadcast()

Condition Variables

- `condition x, y;`
- Two operations are allowed on a condition variable:
  - `x.wait()` – a process that invokes the operation is suspended until `x.signal()`
  - `x.signal()` – resumes one of processes (if any) that invoked `x.wait()`
    - ✓ If no `x.wait()` on the variable, then it has no effect on the variable

Condition Variables Choices

- If process P invokes `x.signal()`, and process Q is suspended in `x.wait()`, what should happen next?
  - Both Q and P cannot execute in paralel. If Q is resumed, then P must wait
- Options include
  - **Signal and wait** – P waits until Q either leaves the monitor or it waits for another condition
  - **Signal and continue** – Q waits until P either leaves the monitor or it waits for another condition
  - Both have pros and cons – language implementer can decide
  - Monitors implemented in Concurrent Pascal compromise
    - ✓ P executing signal immediately leaves the monitor, Q is resumed
  - Implemented in other languages including Mesa, C#, Java