Mohamed Boudiaf
University
-M'sila-

جامعة محمد بوضياف - المسيلة
Université Mohamed Boudiaf - M'sila

Faculty of Mathematics
and Computer Science

Computer Science
Department

OPERATING SYSTEMS II

CHAPTER II
PROCESS SYNCHRONIZATION

SEMAPHORES

3rd Year, Computer Science – ISIL

Dr. Hanene CHERAIT

2025-2026

SYNCHRONISATION

- Events
- Locks
- **Semaphores**
- Monitors
- Critical Sections
- Path Expressions

RESOURCE WITH MULTIPLE INSTANCES

- **Printers**
  - If a system has **3 identical printers**, up to 3 processes can print simultaneously.
    - The **resource count = 3**.
    - Processes must check availability before printing.
- **Database Connections**
  - A database server might allow **N simultaneous connections**.
    - Each connection is an instance of the "database connection" resource.
- **Buffers / Memory Blocks**
  - A system might have a **pool of 5 memory buffers** for I/O.
    - Each process can use a free buffer; if all are in use, it must wait.
- **CPU Cores**
  - A multi-core system has **multiple CPUs**.
    - Each CPU can run one process at a time; the resource is the "CPU" with multiple instances.
- **Network Ports**
  - A server may allow **multiple simultaneous sockets** for clients.
    - Each socket/port instance is a separate resource.

RESOURCE WITH MULTIPLE INSTANCES

- Definition
  - Any resource that has more than one identical unit available for simultaneous use is a multiple-instance resource.
- Problem
  - Standard locks/events cannot efficiently handle multiple-instance resources without extra logic.
- Solution
  - need a counting mechanism
    - number of available instances.

→ Semaphore

SEMAPHORES

- Proposed by Dijkstra in 1965
- To overcome the shortcomings of synchronization using locks and events
  - the computation and retrieval of the number of processes blocked in the queue

Principle: It is to control synchronization by using an abstract data type called Semaphore.

SEMAPHORES

- A semaphore is an abstract object:
  - **An integer value (val):** always an integer indicating the number of processes that can use the semaphore without being blocked.
    - The state of this variable is used to determine whether a process can continue its execution or not.
  - **A waiting queue (Q):** containing all blocked processes.
    - Processes that cannot continue their execution are placed in a queue associated with the semaphore and move to the blocked state.
    - A blocked process **cannot wake up on its own**; another active process must wake it up.

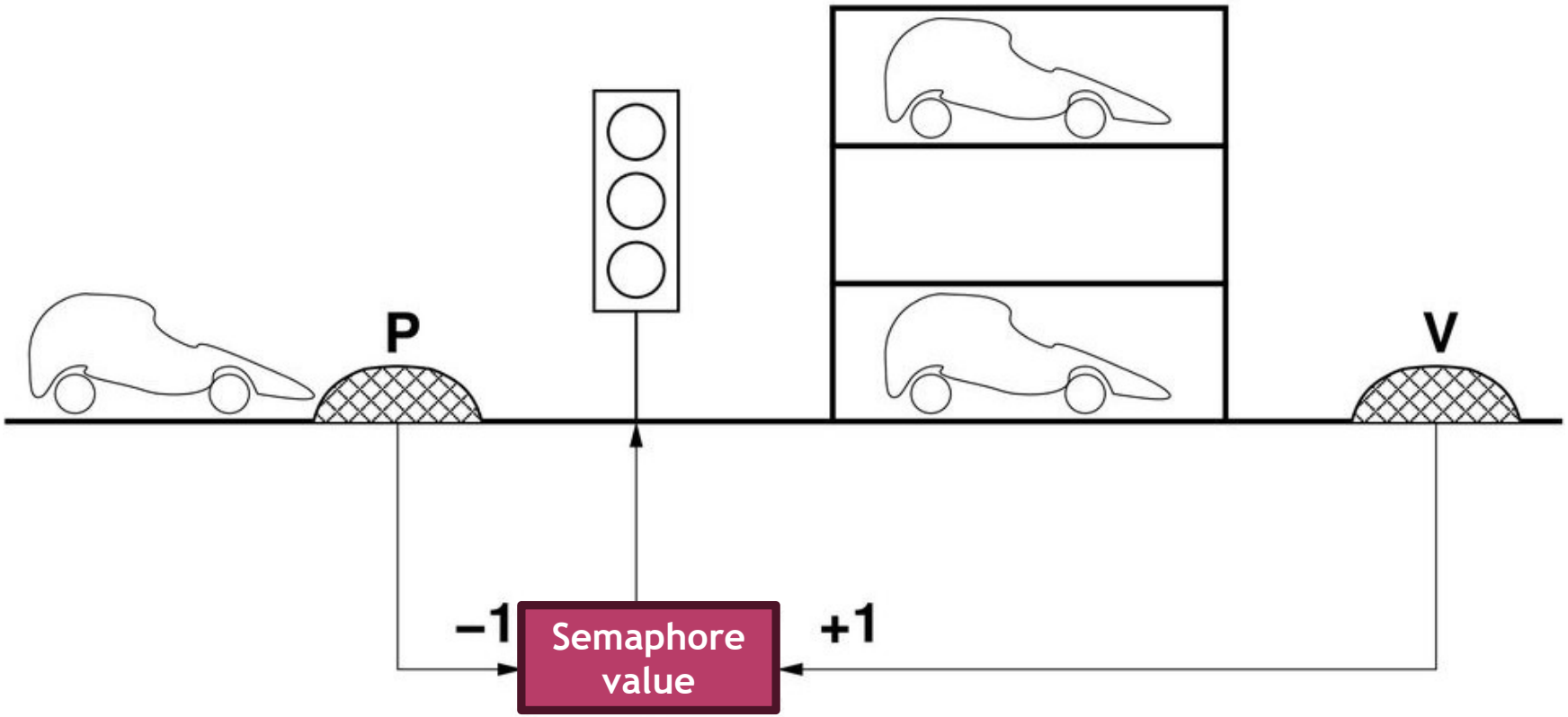
SEMAPHORES

- The semaphore variable can only be manipulated by two **atomic primitives:**
  - **P** (or Wait, Down, Acquire, Decrement)
  - **V** (or Signal, Up, Release, Increment)

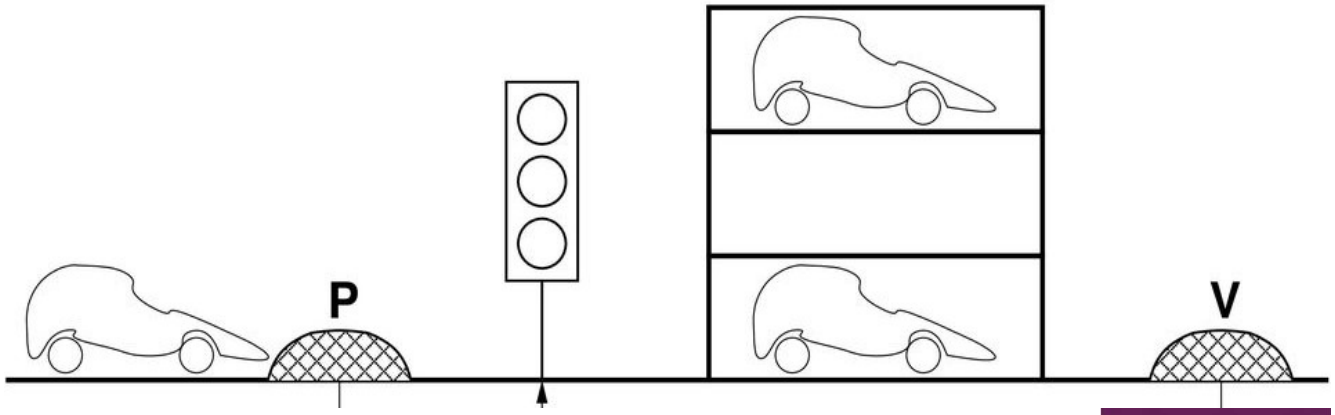
- The terms **P** and **V** for semaphores come from **the Dutch language** because **Edsgar Dijkstra** was Dutch.
  - **P** stands for "**proberen**", which means "**to test**" or "**to try**". It represents the operation where a process **tests and decrements** the semaphore.
  - **V** stands for "**verhogen**", which means "**to increment**" or "**to raise**". It represents the operation where a process **increments the semaphore** and potentially wakes up a waiting process.

SEMAPHORE

Example: A parking with **N spaces** controlled by a traffic light



SEMAPHORE



- The semaphore value (val) represents the number of available parking spaces.
- Each car that enters performs a P (Wait) operation:
  - val is decremented.
  - If val < 0, the car waits (blocked) until a space becomes free.
- Each car that leaves performs a V (Signal) operation:
  - val is incremented.
  - If there are blocked cars waiting, one of them is allowed to enter.

```
// Definition of a new semaphore type
Type semaphore = record
    Val : integer;
      Q : queue;
End;
```

```
 // Initialization of a semaphore variable
Shared context:
   // n is the initial value assigned to S.Val
   S : semaphore := n;
```

```
// Definition of the P primitive
Primitive P(S)
Begin
     S.Val := S.Val - 1;
     If (S.Val < 0) then
     Enqueue(Pi, S.Q); //Block and place process Pi in queue F
     EndIf;
End;
```

```
// Definition of the V primitive
Primitive V(S)
Begin
     S.Val := S.Val + 1;
     If (S.Val ≤ 0) then
          Dequeue(Pi, S.Q);  // Remove process Pi from queue F
     EndIf;
End;
```

SEMAPHORES

- A semaphore cannot be initialized to a negative value, but it can become negative after a certain number of P operations.
- A process that invokes the V primitive on a semaphore will wake up another process blocked on that semaphore if its value is less than or equal to 0.
- When a process invokes the P primitive on a semaphore, one of the following occurs:
  - If the semaphore value is less than zero, the process is blocked and placed in the list associated with the semaphore.
  - If the semaphore value is greater than or equal to zero, the process continues execution normally.
- The value of a semaphore indicates:
  - Either the number of processes blocked behind this semaphore (value < 0),
  - Or the number of processes that can execute the P operation without being blocked (value > 0).

COMMON PROBLEM TYPES AND THEIR SOLUTIONS USING SEMAPHORES

1. **Mutual Exclusion**
2. **Order Relations between Two Processes**
3. **Rendezvous Point**
4. **Producer/Consumer Problem**
5. **Readers/Writers Problem**
6. **Dijkstra's Dining Philosophers Problem**

COMMON PROBLEM TYPES AND THEIR SOLUTIONS USING SEMAPHORES

1. **Mutual Exclusion**
2. **Order Relations between Two Processes**
3. **Rendezvous Point**
4. **Producer/Consumer Problem**
5. **Readers/Writers Problem**
6. **Dijkstra's Dining Philosophers Problem**

SOLUTION TO MUTUAL EXCLUSION USING SEMAPHORES

- **Binary semaphore**

- provides a simple and efficient solution:
  - A semaphore variable is initialized to one (1).
  - The **P (Wait)** operation is executed **before entering the critical section**.
  - The **V (Signal)** operation is executed **after leaving the critical section**.

- This approach is valid for n processes that wish to access their critical sections.

SOLUTION TO MUTUAL EXCLUSION USING SEMAPHORES

```
var mutex : semaphore = 1;   // binary semaphore
```

```
process P1()
begin
    ::::::::::::::::::
    P(mutex)
    CRITICAL SECTION;
    V(mutex)
    ::::::::::::::::::
end;
```

```
process P1()
begin
    ::::::::::::::::::
    P(mutex)
    CRITICAL SECTION;
    V(mutex)
    ::::::::::::::::::
end;
```

```
begin
    parbegin
        P1();   // Execute P1 in parallel
        P2();   // Execute P2 in parallel
    parend
end.
```

SOLUTION TO MUTUAL EXCLUSION USING SEMAPHORES/EXAMPLE

```
var mutex : semaphore = 1;   // binary semaphore
var counter : integer = 0 ;
```

```
process P1()
begin
    ::::::::::::::::::
    P(mutex)
    counter := counter + 1;
    V(mutex)
    ::::::::::::::::::
end;
```

```
process P1()
begin
    ::::::::::::::::::
    P(mutex)
    counter := counter + 1;
    V(mutex)
    ::::::::::::::::::
end;
```

mutex ensures that only one process can update the counter at a time. With **P(mutex)** and **V(mutex)**, the critical section is protected, and the final counter value is **correct** (Respects the four conditions of Dijkestra)

COMMON PROBLEM TYPES AND THEIR SOLUTIONS USING SEMAPHORES

1. **Mutual Exclusion**
2. **Order Relations between Two Processes**
3. **Rendezvous Point**
4. **Producer/Consumer Problem**
5. **Readers/Writers Problem**
6. **Dijkstra's Dining Philosophers Problem**

ORDER RELATIONS BETWEEN TWO PROCESSES

- We consider a process P_0 whose execution progress depends on a signal sent by another process P_1.

```
Process P0
begin
    A1; A2; A3;....An;
    wait for a signal from P1;
    ::::::::::::::::
end;
```

```
Process P1
begin
B1; B2; B3;....Bm;
send the activation signal to P0;
::::::::::::::::
end;
```

ORDER RELATIONS BETWEEN TWO PROCESSES

- Solution. We define a semaphore called *signal*, initialized to 0.

```
var signal : semaphore = 0;

process P0                    process P1
begin                         begin
   A1; A2; A3;....An;         B1; B2; B3;....Bm;
   P(signal);                 V(signal);
   ::::::::::::::::           ::::::::::::::::
end;                          end;
```

In this example, two cases may occur:

Case 1: Process **P0** is already blocked on the primitive **P(signal)** when the signal arrives. → When process **P1** executes the primitive **V(signal)**, it wakes up process **P0**.

Case 2: Process **P0** is active when the signal is sent (it is executing instruction *Ai*). → The value of the semaphore **signal** is set to 1 by **P1**, and when **P0** later executes the primitive **P(signal)**, it will not be blocked.

COMMON PROBLEM TYPES AND THEIR SOLUTIONS USING SEMAPHORES

1. **Mutual Exclusion**
2. **Order Relations between Two Processes**
3. **Rendezvous Point**
4. **Producer/Consumer Problem**
5. **Readers/Writers Problem**
6. **Dijkstra's Dining Philosophers Problem**

RENDEZVOUS POINT

- A process, called **RDV**, must wait until **N other processes** have reached a specific point before continuing its execution.
- The **N−1 processes** are blocked, waiting for the **last process** to arrive.
- The **last (Nth) process** will then wake up the **RDV process**.
- In turn, the **RDV process** will wake up all the other blocked processes.

```
const n=...;
var count : integer;
waitRDV : semaphore initial value = 0; // to block RDV
waitothers : semaphore initial value = 0; // to block Pi
mutex: semaphore initial value =1; // to protect count
```

```
process Pi;
 begin
:::::::::::::::
P(mutex);
count++;
if count == n then begin
  V(mutex);
  V(waitRDV);// Last Pi signals RDV
end;
else begin
    V(mutex);
    P(waitothers);// Other Pi wait for
RDV
end;
:::::::::::::::::
end;
```

```
process RDV;
begin
:::::::::::::::
// Wait until all Pi have arrived
P(waitRDV);
while (count<>1) do begin
    V(waitothers);// Wake one Pi
    count--;
end;
:::::::::::::::
end;
```

```
begin
    parbegin
        P1; P2; P3; .....Pi;
        RDV;
    parend;
end.
```

IMPORTANT NOTE

- The release of the shared resource is a **priority instruction** compared to other instructions.
  - **V(mutex-semaphore)**

- Therefore, after using a shared variable, the **V primitive** must be executed **before proceeding to other instructions.**
  - The **third Dijkstra condition** must be satisfied.

COMMON PROBLEM TYPES AND THEIR SOLUTIONS USING SEMAPHORES

1. **Mutual Exclusion**
2. **Order Relations between Two Processes**
3. **Rendezvous Point**
4. **Producer/Consumer Problem**
5. **Readers/Writers Problem**
6. **Dijkstra's Dining Philosophers Problem**

PRODUCER/CONSUMER PROBLEM

- We consider two categories of processes:
  - Producers and Consumers, which access a shared memory structure called the Buffer.
  - The buffer has a fixed capacity of N, meaning it can store up to N items simultaneously.
  - Producers generate items (of arbitrary values) and insert them into the buffer.
  - Consumer processes retrieve and process the items stored in the buffer.

PRODUCER/CONSUMER PROBLEM

- The operation of the processes must satisfy the following constraints:
  - Producers do not place items into the buffer when it is full.
  - Consumers do not remove items from the buffer when it is empty.
  - Only one process may access the buffer at any given time.
  - Items must not be lost or consumed more than once.

PRODUCER/CONSUMER PROBLEM

- **Use of three semaphores:**
  - **Empty**: the producer is blocked if there is no empty space in the buffer.
    - buffer empty
      - Initialized to N
  - **Full**: the consumer is blocked if there is no filled space in the buffer.
    - buffer empty
      - Initialized to 0
  - **Mutex**: ensures mutual exclusion for access to the buffer.
    - allowing only **one process to access the buffer at a time.**
      - Initialized to 1

```
Program ProducerConsumer;
Const  N = ...;  // Buffer size
Type
Item = ...;
Var
Buffer: Array[0..N-1] of Item;
   Mutex: Semaphore Initial Value = 1;
   Empty: Semaphore Initial Value = N;
   Full: Semaphore Initial Value = 0;


Begin
   ParBegin
      Producer-1; Producer-2; Producer-3; ...; Producer-I;
      Consumer-1; Consumer-2; Consumer-3; ...; Consumer-J;
   ParEnd;
End.
```

```
Process Producer-I;
Var
   producedItem: Item;
Begin
   Repeat
      Produce(producedItem);
      P(Empty);     // Wait if no empty space
      P(Mutex);     // Enter critical section
      Deposit(producedItem, Buffer);
      V(Mutex);   // Leave critical section
      V(Full);    // Signal that buffer has a new item
   Until  true;
End;
```

```
Process Consumer-J;
Var
   consumedItem: Item;
Begin
   Repeat
      P(Full);      // Wait if buffer is empty
      P(Mutex);     // Enter critical section
      Remove(Buffer, consumedItem);
      V(Mutex);    // Leave critical section
      V(Empty);  // Signal that buffer has free space
      Consume(consumedItem);
   Until true;
End;
```

COMMON PROBLEM TYPES AND THEIR SOLUTIONS USING SEMAPHORES

1. **Mutual Exclusion**
2. **Order Relations between Two Processes**
3. **Rendezvous Point**
4. **Producer/Consumer Problem**
5. **Readers/Writers Problem**
6. **Dijkstra's Dining Philosophers Problem**

READERS/WRITERS PROBLEM

- There are two categories of processes that access a single shared resource (e.g., file, database).
  - The first category represents **Readers**, who have **read-only access**.
  - The second category represents **Writers**, who can **read and update** the resource.
- **Synchronization constraints:**
  - Allow multiple reader processes to access the resource simultaneously.
  - Prevent simultaneous access of writer processes to the resource.
  - Prevent a writer process from accessing the resource simultaneously with one or more reader processes.

READERS/WRITERS PROBLEM

- **Readers are not mutually exclusive among themselves.**
- It is necessary to **count the number of readers** using the resource simultaneously.
- **Use of two semaphores:**
  - **Write**: ensures mutual exclusion among writers.
    - allowing only **one writer to access the resource at a time.**
      - Initialized to 1
  - **Mutex**: semaphore for mutual exclusion when updating the reader count.
    - allowing only **one reader to access the counter at a time.**
      - Initialized to 1

```
Program ReadersWriters;
Var   ReaderCount: integer;  Mutex: Semaphore Initial Value = 1;  Write: Semaphore Initial Value = 1;

  Process Reader-I;
  Begin
     P(Mutex);
     ReaderCount := ReaderCount + 1;
     IF ReaderCount = 1 THEN
        P(Write);   // First reader locks writers
     V(Mutex);

     READ;          // Reading section

     P(Mutex);
     ReaderCount := ReaderCount - 1;
     IF ReaderCount = 0 THEN
        V(Write);   // Last reader releases writers
     V(Mutex);
  End;
```

```
Process Writer-J;
Begin
   P(Write);       // Writer acquires exclusive access
   WRITE;          // Writing section
   V(Write);       // Release exclusive access
End;
```

```
Begin
   ReaderCount := 0;
   ParBegin
      Reader-1; Reader-2; Reader-3; ...; Reader-I;
      Writer-1; Writer-2; Writer-3; ...; Writer-J;
   ParEnd;
End;
```

READERS/WRITERS PROBLEM

FAIR SOLUTION

- Readers can only start reading if no writer is waiting.
- Writers get exclusive access but don't block active readers until they finish.
- Use an extra semaphore readTry to prevent new readers from entering when a writer is waiting.
  - readTry initialized to 1

```
Program ReadersWriters;
Var   ReaderCount: integer;  Mutex: Semaphore Initial Value = 1;  Write: Semaphore Initial Value = 1;
      readTry: Semaphore Initial Value = 1;
  Process Reader-I;
  Begin
    P(readTry);
    P(Mutex);
    ReaderCount := ReaderCount + 1;
    IF ReaderCount = 1 THEN
      P(Write);   // First reader locks writers
    V(Mutex);
    V(readTry);
    READ;          // Reading section
    P(Mutex);
    ReaderCount := ReaderCount - 1;
    IF ReaderCount = 0 THEN
      V(Write);   // Last reader releases writers
    V(Mutex);
  End;
```

```
Process Writer-J;
Begin
  P(readTry);
  P(Write);       // Writer acquires exclusive access
  V(readTry);
  WRITE;          // Writing section
  V(Write);       // Release exclusive access
End;
```

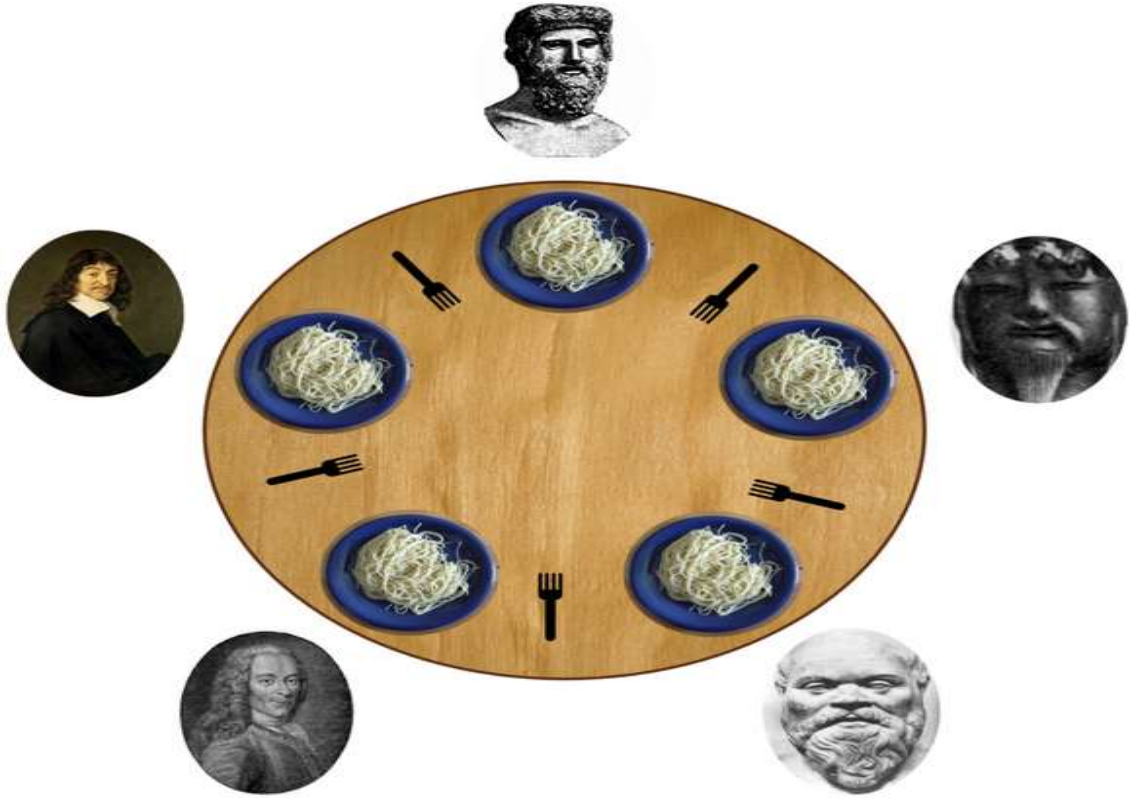
```
Begin
  ReaderCount := 0;
  ParBegin
    Reader-1; Reader-2; Reader-3; ...; Reader-I;
    Writer-1; Writer-2; Writer-3; ...; Writer-J;
  ParEnd;
End;
```

COMMON PROBLEM TYPES AND THEIR SOLUTIONS USING SEMAPHORES

1. **Mutual Exclusion**
2. **Order Relations between Two Processes**
3. **Rendezvous Point**
4. **Producer/Consumer Problem**
5. **Readers/Writers Problem**
6. **Dijkstra's Dining Philosophers Problem**

DIJKSTRA'S DINING PHILOSOPHERS PROBLEM

- Consider **five philosophers** sitting around a round table, who spend their time **thinking** and **eating**.
- In front of each philosopher is a plate of spaghetti so slippery that **two forks are required to eat it**.



DIJKSTRA'S DINING PHILOSOPHERS PROBLEM

- The following constraints must be satisfied to provide a correct solution to this problem:
  1. Only **one philosopher at a time** can hold a fork.
  2. Philosophers **must not enter a deadlock**.
  3. Multiple philosophers **cannot eat simultaneously** if it would violate fork availability.
  4. A philosopher **must not starve** while waiting for a fork.
- **Use of private semaphores (Sempriv) for each philosopher.**
  - Each Sempriv is **initialized to 0**.

```
Program Philosophers;
Var
   State: Array[0..4] of (Thinking, Hungry, Eating);
   Mutex: Semaphore Initial Value = 1;
   Sempriv: Array[0..4] of Semaphore Initial Value = 0;
   i: integer;
```

```
Process Philosopher(var i: integer);
Begin
   Repeat
      Think;
      PickForks(i);
      Eat;
      PutForks(i);
   Until false;
End;
```

```
Begin
   For i := 0 to 4 Do
      State[i] := Thinking;
   ParBegin
      Philosopher(0); Philosopher(1); Philosopher(2);
      Philosopher(3); Philosopher(4);
   ParEnd;
End;
```

```
Procedure PickForks(i: integer);
Begin
   P(Mutex);
   State[i] := Hungry;
   Test(i);
   V(Mutex);
   P(Sempriv[i]);
End;
```

```
Procedure PutForks(i: integer);
Begin
   P(Mutex);
   State[i] := Thinking;
   Test((i + 4) mod 5);  // Test left neighbor
   Test((i + 1) mod 5);  // Test right neighbor
   V(Mutex);
End;
```

```
Procedure Test(i: integer);
Begin
   If (State[i] = Hungry) and (State[(i + 4) mod 5] <> Eating) and (State[(i + 1) mod 5] <> Eating)
Then
   Begin
      State[i] := Eating;
      V(Sempriv[i]);
   End;
End;
```

THANK YOU FOR YOUR ATTENTION