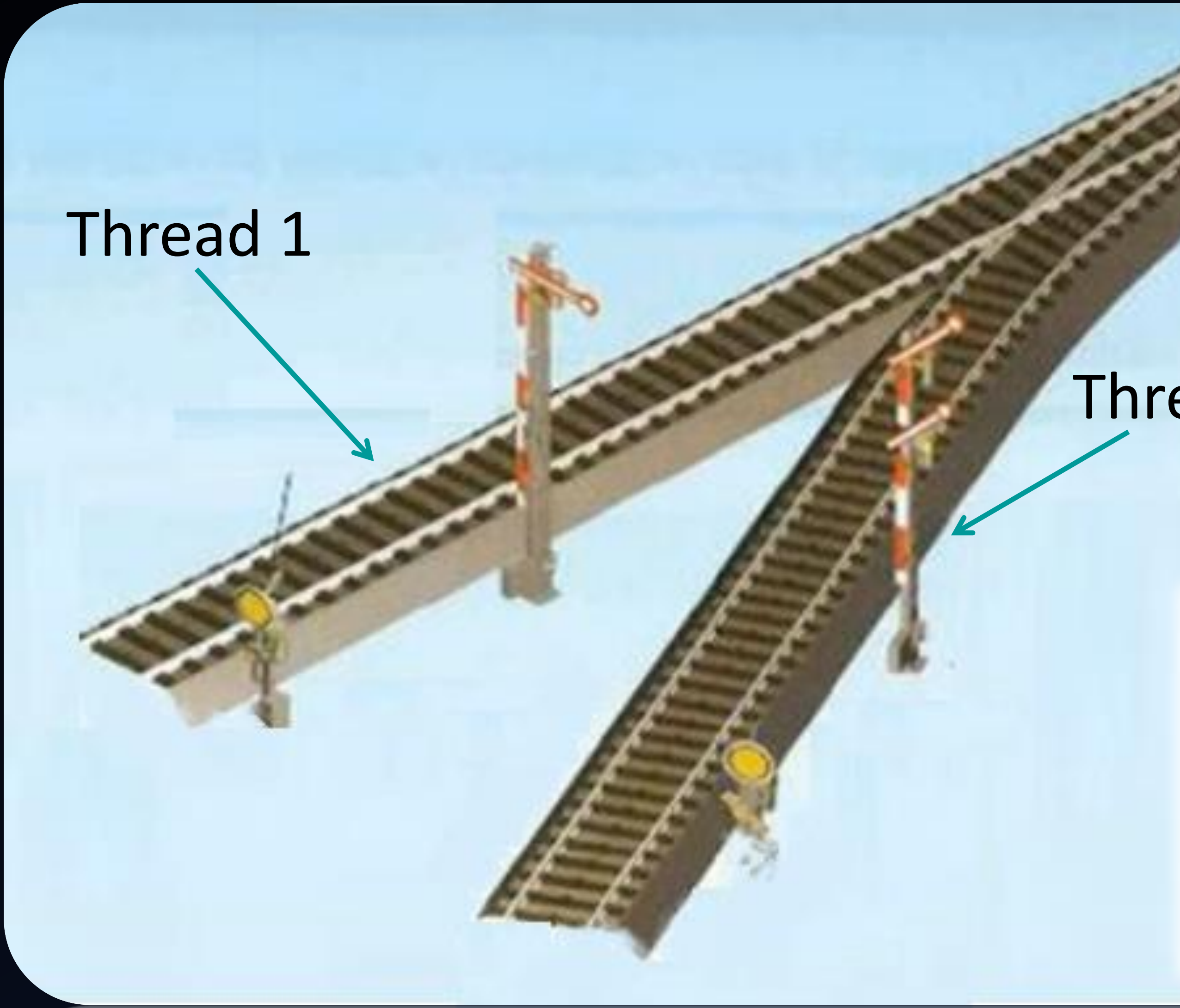
Dekker's algorithms for Semaphores Implementation

ALI TARIQ AL-KHAYYAT (**163101413**)
Submitted to : Prof. Dr. Huseyin Balik

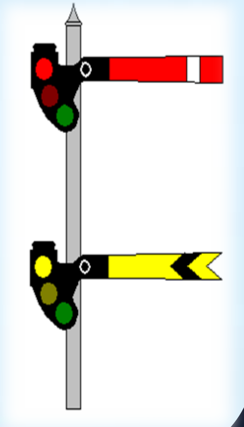
OUTLINES

- what is Dekker's algorithm.
- Dekker's General algorithms.
- What is Semaphores.
- Semaphores implementation.
- Semaphore Implementation Busy waiting.
- semaphores implementation for solving critical section by mutual exclusion.

WHAT IS DEKKER'S ALGORITHMS ?



Dekker adds the idea of a favored thread and allows access to either thread when the request is uncontested



Flag represent

Favored Thread

DEKKER'S ALGORITHMS

- Dekker's algorithm is the first known correct solution to the mutual exclusion problem in concurrent programming, Dutch mathematician Dekker by Dijkstra .

- It allows two threads to share a single-use resource without conflict, using only shared memory for communication.

- If two processes attempt to enter a critical section at the same time, the algorithm will allow only one process in, based on whose turn it is, If one process is already in the critical section.

- the other process will busy wait for the first process to exit, This is done by the use of two flags, wants_to_enter[0] and wants_to_enter[1] , which indicate an intention to enter the critical section on the part of processes 0 and 1, respectively.

DEKKER'S GENERAL ALGORITHMS

```
variables
    wants_to_enter : array of 2 booleans
    turn : integer

wants_to_enter[0] ← false
wants_to_enter[1] ← false
turn ← 0   // or 1
```

```
p0:
   wants_to_enter[0] ← true
   while wants_to_enter[1] {
      if turn ≠ 0 {
         wants_to_enter[0] ← false
         while turn ≠ 0 {
           // busy wait
         }
         wants_to_enter[0] ← true
      }
   }

   // critical section
   ...
   turn ← 1
   wants_to_enter[0] ← false
   // remainder section
```

```
p1:
   wants_to_enter[1] ← true
   while wants_to_enter[0] {
      if turn ≠ 1 {
         wants_to_enter[1] ← false
         while turn ≠ 1 {
           // busy wait
         }
         wants_to_enter[1] ← true
      }
   }

   // critical section
   ...
   turn ← 0
   wants_to_enter[1] ← false
   // remainder section
```

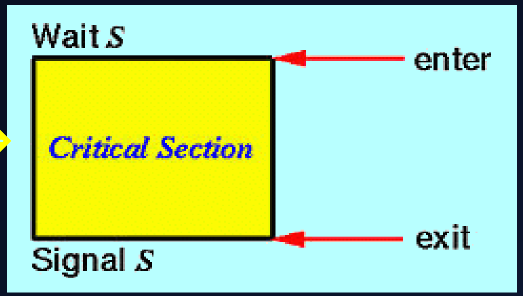
DEKKER'S ALGORITHM

- Assumes two threads, numbered 0 and 1

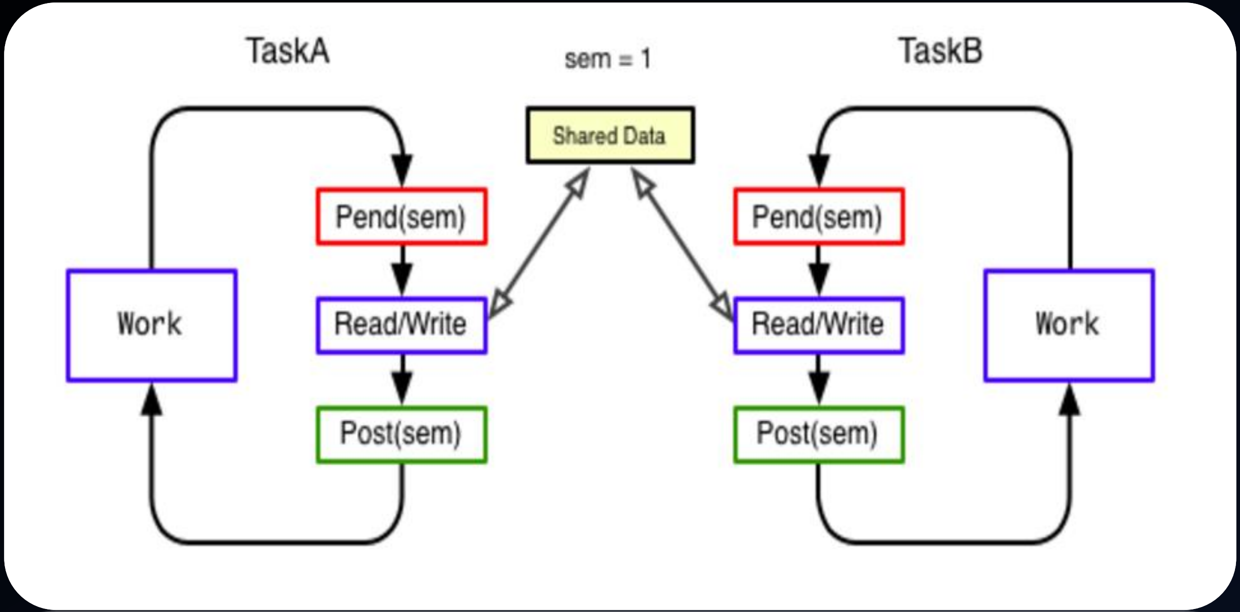
```
CSEnter(int i)
{
  inside[i] = true;
  while(inside[J])
  {
    if (turn == J)
    {
      inside[i] = false;
      while(turn == J) continue;
      inside[i] = true;
    }
  }}
```

```
CSExit(int i)
{
  turn = J;
  inside[i] = false;
}
```

critical section

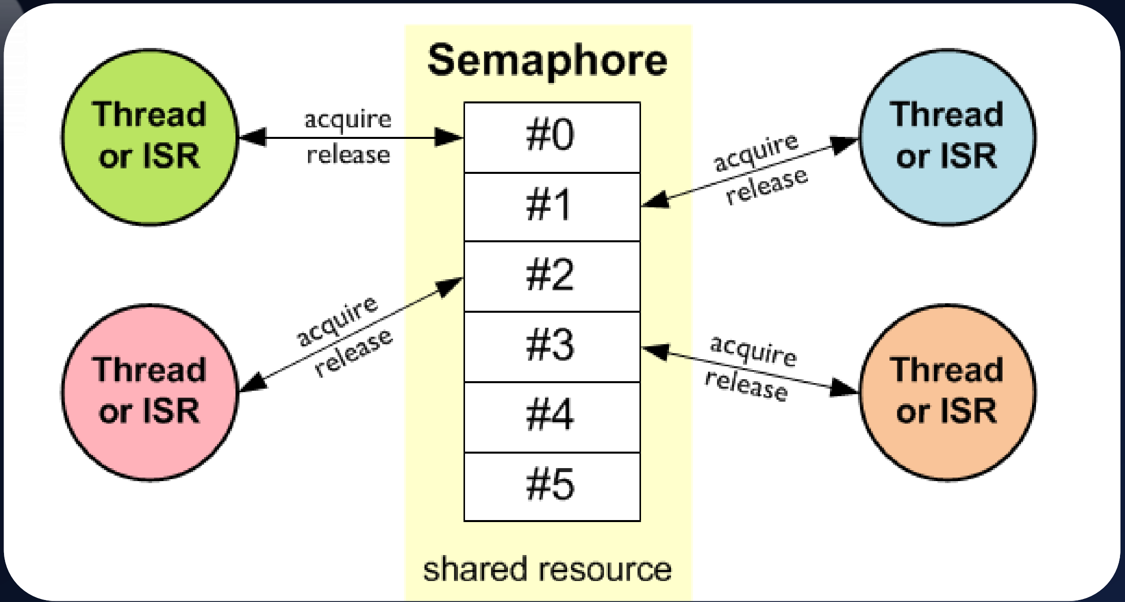


SEMAPHORES



- If a process is waiting for a signal, it is suspended until that signal is sent.
- **Wait and Signal operations cannot be interrupted.**
- A queue is used to hold processes waiting on the semaphore.

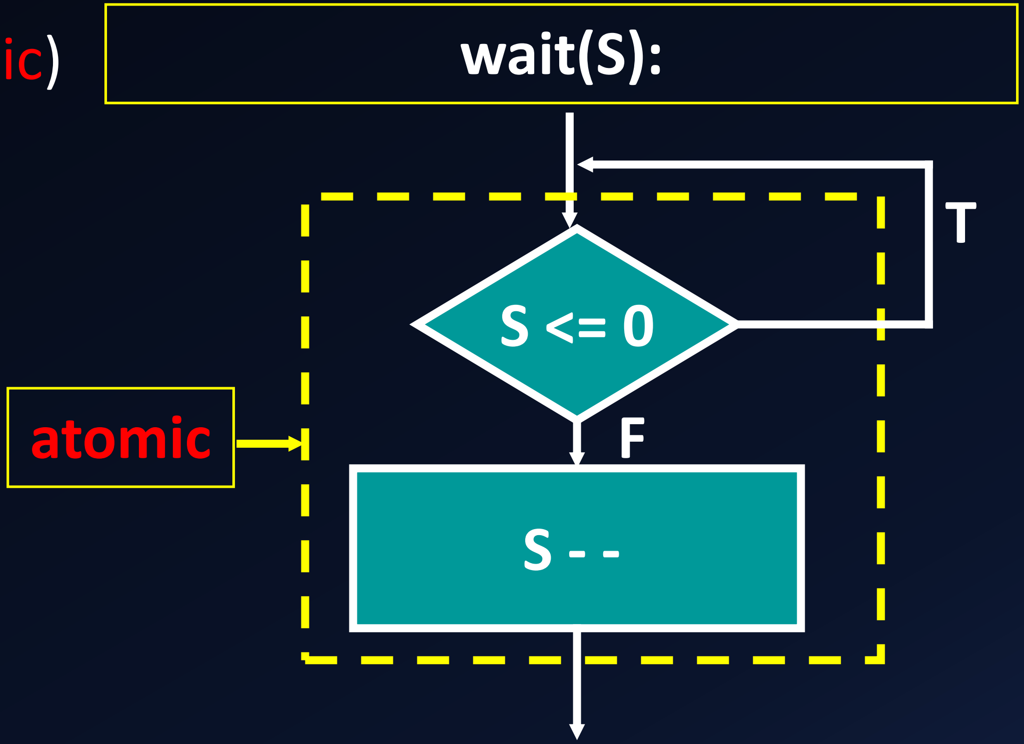
atomic and
mutually exclusive



SEMAPHORES

- Synchronization tool that does not require busy waiting , Semaphore is un integer flag, indicated that it is safe to proceed.
- Two standard operations modify S: wait() and signal()
  - Originally called P() and V() , Less complicated.
- Can only be accessed via two indivisible (atomic) operations.

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



SEMAPHORES IMPLEMENTATION

- Must guarantee that no two processes can execute wait () and signal () on the same semaphore at the same time.
- Thus, 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 Implementation Block and Wakeup

- With each semaphore there is an associated waiting queue. Each entry in a waiting queue has two data items:
  - value (of type integer)
  - pointer to next record in the list
- Two operations:
  - block – place the process invoking the operation on the suitable waiting queue.
  - wakeup – remove one of processes in the waiting queue and place it in the ready queue.

Semaphore Implementation with Busy Waiting

- Implementation of wait:

```
wait (S){
    value--;
    if (value < 0) {
        add this process to waiting queue
        block();  }        }
```

- Implementation of signal:

```
Signal (S){
    value++;
    if (value <= 0) {
        remove a process P from the waiting queue
        wakeup(P);  }        }
```

SEMAPHORES IMPLEMENTATION FOR SOLVING CRITICAL SECTION BY MUTUAL EXCLUSION

- For n processes
- Initialize semaphore "mutex" to 1
- Then only one process is allowed into CS (mutual exclusion)
- To allow 2 processes into CS at a time, simply initialize mutex to 2

```
Process P0:
repeat
  wait(mutex);
   CS
  signal(mutex);
   RS
forever
```

SEMAPHORES IN ACTION

Initialize **mutex** to 1

Process P_0:

```
repeat
wait(mutex);
    CS
signal(mutex);
    RS
forever
```

Process P_1:

```
repeat
  wait(mutex);
    CS
  signal(mutex);
    RS
forever
```

THE REFERENCES

https://en.wikipedia.org/wiki/Dekker%27s_algorithm

- “Operating Systems”, William Stallings, ISBN 0-13-032986-X
- “Operating Systems – A modern perspective”, Garry Nutt, ISBN 0-8053-1295-1