1.a

using System;

using System.Threading;

public class MySemaphore

{

private Mutex mutex = new Mutex();

private int max;

private int current;

public MySemaphore(int starting,int max)

{

this.max = max;

this.current = starting;

}

public bool WaitOne()

{

while (true)

{

mutex.WaitOne();

if (current > 0)

{

current--;

mutex.ReleaseMutex();

return true;

}

mutex.ReleaseMutex();

}

}

public bool release(int num = 1)

{

mutex.WaitOne();

int newCount = current + num;

if (newCount > max)

{

mutex.ReleaseMutex();

return false;

}

current = newCount;

mutex.ReleaseMutex();

return true;

}

}

b. yes. Peterson algorithm can be used for three processes (and even more) using The filter algorithm generalizes Peterson's algorithm to *N*>2 processes. Instead of a boolean flag, it requires an integer variable per process, stored in a single-writer/multiple-reader atomic [register](https://en.wikipedia.org/wiki/Register_machine), and *N*−1 additional variables in similar registers. The registers can be represented in [pseudocode](https://en.wikipedia.org/wiki/Pseudocode) as [arrays](https://en.wikipedia.org/wiki/Array_data_type):

level: array of N integers

last\_to\_enter: array of N-1 integers

// Shared variables

bool flag[3] = {false, false, false}

int turn = 0

// Process i (where i is 0, 1, or 2)

void process\_i() {

int j = (i + 1) % 3

int k = (i + 2) % 3

while (true) {

flag[i] = true

turn = k

while ((flag[j] || flag[k]) && (turn == k || turn == j));

// Critical Section

flag[i] = false

// Remainder Section

}

}

c. Dekker's Algorithm vs. Mutex/Semaphore:

Dekker's Algorithm: Advantages:

1. Provides a software-only solution for mutual exclusion.
2. Doesn't require special hardware support.
3. Guarantees strict alternation between processes.

Disadvantages:

1. Limited to two processes.
2. Can be complex to implement and understand.
3. May lead to busy-waiting, which can be inefficient.

Mutex/Semaphore: Advantages:

1. Can handle multiple processes/threads.
2. Often implemented with hardware support, making them more efficient.
3. Simpler to use and understand.
4. Can avoid busy-waiting through blocking.

Disadvantages:

1. Require operating system support.
2. May introduce overhead for very short critical sections.
3. Can lead to priority inversion if not carefully managed.

d. Real-world deadlock scenario with four threads:

Scenario: Consider a banking system with four threads handling different operations:

* T1: Transfer funds
* T2: Update account balance
* T3: Generate report
* T4: Process loan application

Resources:

* R1: Customer account database
* R2: Transaction log
* R3: Reporting database
* R4: Credit score system

Deadlock situation:

* T1 holds R1, waiting for R2
* T2 holds R2, waiting for R3
* T3 holds R3, waiting for R4
* T4 holds R4, waiting for R1

Allocation Graph:

Copy

T1 → R2

↑ ↓

R1 ← T4

↑ ↓

T2 → R3

↑ ↓

R2 ← T3

Allocation Matrix:

Copy

R1 R2 R3 R4

T1 1 0 0 0

T2 0 1 0 0

T3 0 0 1 0

T4 0 0 0 1

R1 R2 R3 R4

T1 0 1 0 0

T2 0 0 1 0

T3 0 0 0 1

T4 1 0 0 0

e. Deadlock scenario with four threads and six mutexes:

Scenario:

* T1 holds M1 and M2, waiting for M3
* T2 holds M3 and M4, waiting for M5
* T3 holds M5 and M6, waiting for M1
* T4 holds M2, waiting for M4

Allocation Graph:

Copy

T1 → M3

↑ ↓

M1 ← T3

↑ ↓

M6 M5

↑ ↓

T3 ← T2

↑ ↓

M2 ← T4 → M4

Allocation Matrix:

Copy

M1 M2 M3 M4 M5 M6

T1 1 1 0 0 0 0

T2 0 0 1 1 0 0

T3 0 0 0 0 1 1

T4 0 1 0 0 0 0

M1 M2 M3 M4 M5 M6

T1 0 0 1 0 0 0

T2 0 0 0 0 1 0

T3 1 0 0 0 0 0

T4 0 0 0 1 0 0

This scenario involves all four threads and five out of the six mutexes in the deadlock situation.