Pseudocodes

Simulating Semaphores

```
Initialize a semaphore 'sem' to 1

function wait(sem):
   while sem <= 0:   # Busy wait if semaphore is not available
      continue
   sem = sem - 1

function signal(sem):
   sem = sem + 1

function critical_section():
   print("Critical section accessed")

function process():
   wait(sem)            # Acquire the semaphore (lock)
   critical_section()      # Enter the critical section
   signal(sem)           # Release the semaphore (unlock)

main():
   initialize sem = 1      # Semaphore initialized to 1 (binary semaphore)

   create two threads t1, t2
   t1 runs process
   t2 runs process

   join t1, t2           # Wait for both threads to complete
```

Producer Consumer Problem using Semaphore

```
Initialize semaphores:
  mutex = 1        # Controls access to the buffer (binary semaphore)
  full = 0         # Number of full slots (counting semaphore)
  empty = N        # Number of empty slots (N-size buffer) (counting semaphore)

function wait(sem):
  while sem <= 0:
    continue
  sem = sem - 1

function signal(sem):
  sem = sem + 1

function producer():
  while true:
    produce_item()              # Create an item
    wait(empty)                 # Check for an empty slot
    wait(mutex)                 # Enter critical section

    add item to buffer          # Add item to shared buffer
    signal(mutex)               # Leave critical section
    signal(full)                # Increment count of full slots

function consumer():
  while true:
    wait(full)                  # Check for full slot
    wait(mutex)                 # Enter critical section

    remove item from buffer     # Remove item from shared buffer
    signal(mutex)               # Leave critical section
    signal(empty)               # Increment count of empty slots

    consume_item()              # Consume the item

main():
  initialize mutex = 1, full = 0, empty = N

  create producer thread
  create consumer thread

  join producer and consumer threads
```

Reader Write Problem using Semaphores

```
Initialize semaphores:
  mutex = 1       # Binary semaphore for protecting the reader count
  wrt = 1         # Binary semaphore for controlling access to the shared resource
  read_count = 0    # Integer for tracking number of readers

function wait(sem):
  while sem <= 0:
    continue
  sem = sem - 1

function signal(sem):
  sem = sem + 1

function reader():
  while true:
    wait(mutex)              # Protect critical section for read_count
    read_count = read_count + 1
    if read_count == 1:
      wait(wrt)            # If it's the first reader, block writers
    signal(mutex)            # Leave critical section

    read_data()              # Access the shared resource

    wait(mutex)              # Protect critical section for read_count
    read_count = read_count - 1
    if read_count == 0:
      signal(wrt)          # If it's the last reader, allow writers
    signal(mutex)            # Leave critical section

function writer():
  while true:
    wait(wrt)              # Block readers and other writers
    write_data()             # Write to the shared resource
    signal(wrt)              # Allow others (readers or writers)

main():
  initialize mutex = 1, wrt = 1, read_count = 0

  create multiple reader threads
  create multiple writer threads

  join all threads
```

Peterson's Problem

```
Initialize shared variables:
  flag[2] = {false, false}    # Flag to indicate if process wants to enter critical section
  turn = 0                    # Variable to decide whose turn it is to enter critical section

function process_0():
  while true:
    flag[0] = true            # Indicate that process 0 wants to enter
    turn = 1                  # Give priority to process 1
    while flag[1] == true and turn == 1:
      continue                # Busy wait until process 1 finishes

    critical_section()        # Enter critical section

    flag[0] = false           # Leave critical section

function process_1():
  while true:
    flag[1] = true            # Indicate that process 1 wants to enter
    turn = 0                  # Give priority to process 0
    while flag[0] == true and turn == 0:
      continue                # Busy wait until process 0 finishes

    critical_section()        # Enter critical section

    flag[1] = false           # Leave critical section

main():
  create two threads t0, t1
  t0 runs process_0
  t1 runs process_1

  join t0, t1
```

Dining Philosophers Problem Using Semaphore

```
Initialize:
  N = 5                    # Number of philosophers (and forks)
  semaphore chopstick[N] = {1, 1, 1, 1, 1}   # One semaphore for each fork, initialized to
1

function wait(sem):
  while sem <= 0:           # Busy wait if the semaphore is not available
    continue
  sem = sem - 1             # Acquire the semaphore (lock)

function signal(sem):
  sem = sem + 1             # Release the semaphore (unlock)

function philosopher(id):
  while true:
    think()                 # Philosopher is thinking

    wait(chopstick[id])       # Pick up the left chopstick (semaphore)
    wait(chopstick[(id+1) % N])  # Pick up the right chopstick (semaphore)

    eat()                   # Philosopher is eating

    signal(chopstick[id])     # Put down the left chopstick
    signal(chopstick[(id+1) % N]) # Put down the right chopstick

main():
  initialize chopstick[N] to 1     # All chopsticks are free (available)

  create N philosopher threads
  for i = 0 to N-1:
    create philosopher(i) thread  # Each philosopher runs in parallel

  join all philosopher threads
```