Case Study on Semaphores

Semaphores are synchronisation primitives used in concurrent programming. They control access to shared resources through wait and signal operations. Processes or threads wait for available resources by decrementing the semaphore's value, and release resources by incrementing it. Semaphores help prevent race conditions and ensure mutual exclusion in multi-threaded environments.

Types of Semaphores:

1. Binary Semaphore:

- Also known as mutex semaphore.
- Can only take on the values 0 and 1.
- Used for mutual exclusion, where only one process can access the critical section at a time.

2. Counting Semaphore:

- Can take on any non-negative integer value.
- Used to control access to a finite number of identical resources.

Types of Problems:

1. Dining Philosophers Problem:

- Scenario: Several philosophers sit at a table with a bowl of spaghetti. Each philosopher spends his life alternating between eating and thinking. To eat, a philosopher needs two forks, one on his left and one on his right.
- Challenge: Avoid deadlock and starvation while allowing each philosopher to eat and think.

2. Reader-Writer Problem:

- Scenario: A database is shared between multiple readers and writers. Multiple readers can access the database simultaneously, but only one writer can access it at a time, and when a writer is writing, no readers can read.
- Challenge: Allow multiple readers to access the database concurrently while ensuring that writers have exclusive access to the database.

3. Producer-Consumer Problem:

- Scenario: There are two types of processes, producers and consumers, who share a common, fixed-size buffer. Producers place items into the buffer, and consumers remove items from it.
- Challenge: Coordinate the producers and consumers so that producers do not try to add items into a full buffer and consumers do not try to remove items from an empty buffer.

Algorithms:

1. Dining Philosophers Problem:

- Each fork is represented by a binary semaphore.
- Each philosopher is a separate process.
- When a philosopher wants to eat, they must acquire the two forks adjacent to them (left and right).
- If both forks are available, the philosopher can eat; otherwise, they wait until the forks become available.

```
Semaphore fork[5]; // One semaphore for each fork

void philosopher(int i)
{
   while (true)
{
      think();

      wait(fork[i]); // Pick up left fork
      wait(fork[(i+1) % 5]); // Pick up right fork

      eat();

      signal(fork[i]); // Put down left fork
      signal(fork[(i+1) % 5]); // Put down right fork
   }
}
```

2. *Reader-Writer Problem*:

- Two semaphores are used: a mutex semaphore for ensuring exclusive access to the shared resource (the database) and a semaphore for readers to coordinate their access.
- Readers must acquire the mutex semaphore before incrementing the reader count. If they are the first reader, they also acquire the semaphore for readers.
- Writers must wait until no readers are reading before they can access the database.

```
Semaphore mutex = 1; // Controls access to shared data
Semaphore wrt = 1; // Controls access for writers

int reader_count = 0;

void reader()
{

   while (true)
         {
               wait(mutex);
               reader_count++;
               if (reader_count == 1)
         {
         wait(wrt); // Acquire writer lock
      }

      signal(mutex);

      // Read data
      wait(mutex);

      reader_count--;
      if (reader_count == 0) {
         signal(wrt); // Release writer lock
      }
      signal(mutex);
   }
}


void writer()
{
   while (true)
         {
                  wait(wrt); // Acquire writer lock
               // Write data
               signal(wrt); // Release writer lock
   }
}
```

3. Producer-Consumer Problem:

- Two semaphores are used: empty and full, representing the number of empty and full slots in the buffer, respectively.
- Producers must wait until there is space in the buffer (i.e., empty is greater than 0) before adding an item to it. After adding an item, they decrement empty and increment full.
- Consumers must wait until there are items in the buffer (i.e., full is greater than 0) before removing an item. After removing an item, they decrement full and increment empty.

```
Semaphore mutex = 1; // Controls access to shared buffer
Semaphore empty = n; // Counts empty slots in buffer
Semaphore full = 0; // Counts filled slots in buffer

int buffer[n];
int in = 0, out = 0;

void producer()
{
   while (true)
{
      produce_item(item);

      wait(empty); // Wait for empty slot
      wait(mutex); // Enter critical section

      buffer[in] = item;
      in = (in + 1) % n;

      signal(mutex); // Exit critical section
      signal(full); // Increment count of filled slots
   }
}

void consumer()
{
   while (true)
{
      wait(full); // Wait for filled slot
      wait(mutex); // Enter critical section

      item = buffer[out];
      out = (out + 1) % n;

      signal(mutex); // Exit critical section
      signal(empty); // Increment count of empty slots

      consume_item(item);
   }
}
```

- By <u>Rudra Potghan</u>