Synchronization Problems

The Readers-Writers Problem

Suppose several concurrent processes are doing operations on a database. Some processes may want only to read the database, while others may want to update (read and write) the database.

In the following problem, we'll refer to the former as **readers** and to the latter as **writers**.

There is no harm if two readers access the shared data in the same time. The problem arises if at least one writer is involved, as chaos may ensure.

To make sure we have no difficulties, we demand that the writers have **exclusive access** to the shared database while writing to it.

This synchronization problem is known as **the readers-writers problem**.

There are multiple variations to this problem, and all of them are based on priorities:

1. **First** readers-writers problem, requires that **no reader be kept waiting** unless a writer has already obtained permission to use the shared object.
2. **Second** readers-writers problem requires that, **once a writer is ready**, that writer performs its write as soon as possible.

Note:

> A solution to either problem may result in **starvation**.
>
> In the first case, writers may starve.
>
> In the second case, readers may starve.

Starvation-free solution to the first variation

```
#include <stdlib.h>
#include <stdio.h>
#include <pthread.h>
#include <semaphore.h>
#include <unistd.h>
```

```
#define NUM_READERS 5
#define NUM_WRITERS 2

int shared_data = 0; // shared "database"
sem_t rw_mutex, mutex; // binary semaphores

// counting semaphore
// keeps count on how many processes are
// currently reading the object
int read_count = 0;

void *reader(void *args)
{
   int id = *(int *)args;

   while (1)
   {
      sem_wait(&mutex); // I want to update the number of readers

      read_count++;

      // If this is the first reader
      if (read_count == 1)
         // it means there were no active readers before
         // this blocks writers from writing
         // when readers are active
         sem_wait(&rw_mutex);

      sem_post(&mutex); // Release the mutex

      // shared_data is not protected because
      // multiple readers can access it concurrently
      printf("Reader %d is reading the data: %d\n", id, shared_data);

      sem_wait(&mutex); // I want to decrement the number of readers

      read_count--;
```

```
        // If this is the last reader
        if (read_count == 0)
            // rw_mutex is released
            // now writers can proceed
            sem_post(&rw_mutex);

        sem_post(&mutex); // Release the mutex

        sleep(rand() % 3 + 1);
    }

    return NULL;
}

void *writer(void *args)
{
    int id = *(int *)args;

    while (1)
    {
        sem_wait(&rw_mutex); // Only one writer at a time has exclusive access

        printf("Writer %d is writing...\n", id);
        shared_data++;

        sem_post(&rw_mutex); // Now someone else can access the shared dat

        sleep(rand() % 3 + 1);
    }

    return NULL;
}

int main()
{
    pthread_t readers[NUM_READERS], writers[NUM_WRITERS];
    int readers_ids[NUM_READERS], writers_ids[NUM_WRITERS];
    // rw_mutex is common to both reader and writer processes
    // mutual exclusion semaphore for the writers
    // used only by the first and last reader that enters/exits cs
    sem_init(&rw_mutex, 0, 1);

    sem_init(&mutex, 0, 1); // to ensure mutual exclusion when read_count is up

    for (int i = 0; i < NUM_READERS; i++)
    {
        readers_ids[i] = i + 1;
        pthread_create(&readers[i], NULL, reader, &readers_ids[i]);
    }

    for (int i = 0; i < NUM_WRITERS; i++)
    {
        writers_ids[i] = i + 1;
        pthread_create(&writers[i], NULL, writer, &writers_ids[i]);
    }

    for (int i = 0; i < NUM_READERS; i++)
    {
        pthread_join(readers[i], NULL);
    }

    for (int i = 0; i < NUM_WRITERS; i++)
    {
        pthread_join(writers[i], NULL);
    }

    sem_destroy(&mutex);
    sem_destroy(&rw_mutex);

    return 0;
}
```

The Dining-Philosophers Problem

Consider five philosophers who spend their lives thinking and eating. The philosophers share a circular table surrounded by five chairs, each belonging to one philosopher.

In the center of the table is a bowl of rice, and the table is laid with five single chopsticks (or forks). From time to time, a philosopher gets hungry and tries to pick up the chopsticks closest to him and he may pick up only one chopstick at a time.

Obviously, no one can pick up a chopstick that is already in the hand of a neighbor. When a hungry philosopher has both his chopsticks at the same time, he eats without releasing the chopsticks. When he finished eating, he puts down both chopsticks and starts thinking again.

This problem is important because it's an example of a large class of concurrency-control problems. It raises the question: **How can we allocate several resources among several processes in a deadlock-free and starvation-free manner?**

Semaphore Solution

We can represent each chopstick with a semaphore.

A philosopher tries to grab a chopstick by executing a `wait()` operation on that semaphore. He releases his chopstick by executing the `signal()` operation on the appropiate semaphores.

The shared data become:

```
semaphore chopsticks[5];
```

with all the elements initialized to 1.

```
while(true) {
    wait(chopstick[i]);
    wait(chopstick[(i+1)%5]);

    // eat

    signal(chopstick[i]);
    signal(chopstick[(i+1)%5]);
    // think
  }
```

Note

> This solution guarantees that no two neighbors are eating simultaneously, but it must be rejected because it could create a **deadlock**.
>
> Suppose that all five philosophers are hungry at the same time and they all grab their left chopstick. All the semaphores are now equal to 0 and they cannot grab their right chopstick, because they're waiting for a signal.