CL2006 - Operating Systems Spring 2024
LAB # 10 MANUAL (Common)

Please note that all labs' topics including pre-lab, in-lab and post-lab exercises are part of the theory and labsyllabus. These topics will be part of your Midterms and Final Exams of lab and theory.

Objectives:

1. **To understand and learn using mutex and semaphore to protect critical sections while accessing shared memory in multi-process and multi-threaded applications.**

Lab Tasks:

1. Compile and run the code workouts to familiarize yourself with various aspects of mutexes and semaphores.
2. Observe and complete In-Lab to acquire skills to code using synchronization primitives.

Delivery of Lab contents:

Strictly following the following content delivery strategy. Ask students to take notes during the lab.

1st Hour

- Ask students to read background information on Mutexes and Semaphores and rewrite both codes on paper (30 + 30 minutes).

2nd Hour

- Perform Code workout # 1. Objective: Understand bounded buffer problem.

3rd Hour

- Perform Code workout # 2. Objective: Understand readers-writers problem.

Created by: Nadeem Kafi (19/04/2024)
DEPARTMENT OF COMPUTER SCEICEN, FAST-NU, KARACHI

** ChatGPT is heavily used to make the contents of this document along with other sources.

EXPERIMENT 10

Mutexes and Semaphores

Introduction to Mutexes

Mutex, short for mutual exclusion, is a synchronization primitive used to ensure that only one thread at a time can access a shared resource. Mutexes are commonly employed in multithreaded programming to prevent race conditions, where multiple threads try to access a shared resource simultaneously, leading to unpredictable behavior.

Applications of Mutexes

- Protecting Critical Sections: Mutexes are used to protect critical sections of code, ensuring that only one thread can execute the code block at a time. This prevents data corruption and maintains consistency when multiple threads access shared data.
- Thread Synchronization: Mutexes are essential for synchronizing access to shared resources such as variables, data structures, or files in a multithreaded environment.
- Resource Management: Mutexes are used to coordinate access to resources that can only be used by one thread at a time, such as hardware devices or database connections.

Basic Mutex Usage in Code:

```c
#include <pthread.h>
#include <stdio.h>

#define NUM_THREADS 5

int shared_data = 0;
pthread_mutex_t mutex;

void *thread_function(void *arg) {
  int thread_id = *((int *)arg);

  // Lock and unlock the mutex around shared resoruces
  pthread_mutex_lock(&mutex);
  shared_data++;  // Critical section - Thread is accessing shared_data
  pthread_mutex_unlock(&mutex);

  pthread_exit(NULL);
}

int main() {
  pthread_t threads[NUM_THREADS];
  int thread_args[NUM_THREADS];
  int i;

  pthread_mutex_init(&mutex, NULL);  // Initialize the mutex

  for (i = 0; i < NUM_THREADS; i++) {   // Create threads
    thread_args[i] = i;
    pthread_create(&threads[i], NULL, thread_function, &thread_args[i]);
  }
  for (i = 0; i < NUM_THREADS; i++)   // Join threads
    pthread_join(threads[i], NULL);

  pthread_mutex_destroy(&mutex);   // Destroy the mutex

  return 0;
}
```

In the above code example:

- We initialize a mutex using pthread_mutex_init.
- Each thread locks the mutex before accessing the shared resource using pthread_mutex_lock.
- After accessing the resource, the thread unlocks the mutex using pthread_mutex_unlock.
- Finally, we destroy the mutex using pthread_mutex_destroy after all threads have completed execution.

This code ensures that only one thread can access the shared_data variable at a time, preventing race conditions and ensuring data integrity.

Introduction to Semaphores

Semaphore is a synchronization primitive introduced by Edsger W. Dijkstra in the 1960s. It is a signaling mechanism used to control access to shared resources by multiple processes or threads. Semaphores can be either binary (0 or 1) or counting (can have a value greater than 1). A binary semaphore is a semaphore with only two possible integer values: 0 and 1. It acts as a simple on-off switch or a lock. A counting semaphore is a semaphore with an integer value greater than or equal to zero. It can be incremented or decremented by arbitrary values.

Applications of Binary Semaphores:

- Mutual Exclusion: Binary semaphores are commonly used to implement mutual exclusion, ensuring that only one process or thread accesses a shared resource at a time.
- Producer-Consumer Problem: Binary semaphores can be used to synchronize producer and consumer processes or threads in a bounded buffer scenario.
- Deadlock Avoidance: Binary semaphores can help avoid deadlock situations by providing a mechanism for processes to request and release resources in a controlled manner.

Applications of Counting Semaphores:

- Resource Allocation: Counting semaphores are useful for managing finite resources, such as memory buffers, database connections, or I/O devices, where multiple instances of the resource may be available.
- Concurrency Control: Counting semaphores can be used to control the maximum number of concurrent executions of a particular section of code or a critical resource.
- Task Synchronization: In scenarios where multiple tasks need to synchronize at certain points in their execution, counting semaphores can be employed to coordinate their actions.

Basic Semaphore Usage in Code:

```c
#include <pthread.h>
#include <semaphore.h>
#include <stdio.h>

#define BUFFER_SIZE 5
#define NUM_PRODUCERS 2
#define NUM_CONSUMERS 2

int buffer[BUFFER_SIZE];
sem_t empty_slots, full_slots;
pthread_mutex_t mutex;

void *producer(void *arg) {
  int item = *((int *)arg);
  sleep(1);   // Produce item  // Sleep to simulate production time
  sem_wait(&empty_slots);   // Wait for empty slot in buffer
  pthread_mutex_lock(&mutex);   // Acquire mutex lock before accessing buffer
  for (int i = 0; i < BUFFER_SIZE; i++) {   // Add item to buffer
    if (buffer[i] == -1) {
      buffer[i] = item;
      break;
    }
  }
  pthread_mutex_unlock(&mutex);   // Release mutex lock
  sem_post(&full_slots);   // Signal that buffer has a new item

  pthread_exit(NULL);
}

void *consumer(void *arg) {
  sem_wait(&full_slots);   // Wait for buffer to have data
  pthread_mutex_lock(&mutex);   // Acquire mutex lock before accessing buffer
  int item = -1;
  for (int i = 0; i < BUFFER_SIZE; i++) {   // Remove item from buffer
    if (buffer[i] != -1) {
      item = buffer[i];
      buffer[i] = -1;
      break;
```

```c
    }
  }
  pthread_mutex_unlock(&mutex);   // Release mutex lock
  sem_post(&empty_slots);   // Signal that an empty slot is available in buffer
  printf("Consumed item: %d\n", item);   // Consume item

  pthread_exit(NULL);
}

int main() {
  pthread_t producer_threads[NUM_PRODUCERS], consumer_threads[NUM_CONSUMERS];
  int producer_args[NUM_PRODUCERS] = {1, 2}; // Argument for producers

  pthread_mutex_init(&mutex, NULL);   // Initialize mutex and semaphores
  sem_init(&empty_slots, 0, BUFFER_SIZE); // Initialize empty_slots to BUFFER_SIZE
  sem_init(&full_slots, 0, 0);            // Initialize full_slots to 0
  for (int i = 0; i < NUM_PRODUCERS; i++) {   // Create producer threads
    pthread_create(&producer_threads[i], NULL, producer, &producer_args[i]);
  }
  for (int i = 0; i < NUM_CONSUMERS; i++) {  // Create consumer threads
    pthread_create(&consumer_threads[i], NULL, consumer, NULL);
  }
  // Join threads
  for (int i = 0; i < NUM_PRODUCERS; i++) pthread_join(producer_threads[i], NULL);
  for (int i = 0; i < NUM_CONSUMERS; i++) pthread_join(consumer_threads[i], NULL);

  // Destroy mutex and semaphores
  pthread_mutex_destroy(&mutex);
  sem_destroy(&empty_slots); sem_destroy(&full_slots);

  return 0;
}
```

In the above code example:

- We initialize two semaphores: empty_slots (initially set to the buffer size) and full_slots (initially set to 0).
- Producers wait for empty slots in the buffer (sem_wait(&empty_slots)) before producing items and signal that a new item is available (sem_post(&full_slots)).
- Consumers wait for the buffer to have data (sem_wait(&full_slots)) before consuming items and signal that an empty slot is available (sem_post(&empty_slots)).
- Mutex is used to protect the critical section (accessing the buffer) from simultaneous access by multiple threads.

Mutex vs Semaphore

<u>Use a Mutex When:</u>
- Mutual Exclusion is Needed: If you need to ensure that only one thread can access a shared resource at a time to prevent data corruption or race conditions, a mutex is the appropriate choice. Mutexes provide binary synchronization, ensuring exclusive access to a critical section of code.
- Simple Locking Mechanism: When you require a straightforward locking mechanism with only two states (locked or unlocked), a mutex is typically more appropriate. Mutexes are simpler and more lightweight than semaphores, making them suitable for scenarios where only binary synchronization is needed.
- Low Resource Consumption: Mutexes generally have lower overhead than semaphores, making them more efficient in terms of resource consumption. If efficiency is a concern and binary synchronization suffices for your needs, a mutex is a suitable choice.

<u>Use a Semaphore When:</u>
- Resource Counting is Required: If you need to manage multiple instances of a resource or control access to a resource based on a count, a semaphore is the appropriate choice. Counting semaphores allow you to specify an initial count and increment or decrement the count based on resource availability.
- Complex Synchronization Scenarios: When your synchronization requirements involve more complex scenarios beyond simple mutual exclusion, such as managing multiple resources with different availability conditions or coordinating multiple threads/tasks, semaphores provide more flexibility and expressive power.
- Task Synchronization: If you need to coordinate the synchronization of multiple tasks or processes at various points in their execution, semaphores can be used to signal events or conditions between them.

Code workout # 1:

Synchronization Examples: Bounded Buffer Problem.

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

#define BUFFER_SIZE 5

sem_t mutex, empty, full;
int buffer[BUFFER_SIZE];
int in = 0, out = 0;

void *producer(void *arg) {
    int item;
    while (1) {
        item = rand() % 100; // Generate a random item to produce
        sem_wait(&empty);
        sem_wait(&mutex);

        buffer[in] = item;
        printf("Produced: %d\n", item);
        in = (in + 1) % BUFFER_SIZE;

        sem_post(&mutex);
        sem_post(&full);
        sleep(rand() % 3); // Simulate some processing time
    }
}

void *consumer(void *arg) {
    int item;
    while (1) {
        sem_wait(&full);
        sem_wait(&mutex);

        item = buffer[out];
        printf("Consumed: %d\n", item);
        out = (out + 1) % BUFFER_SIZE;

        sem_post(&mutex);
        sem_post(&empty);
        sleep(rand() % 3); // Simulate some processing time
    }
}

int main() {
    pthread_t producer_thread, consumer_thread;

    // Initialize semaphores
    sem_init(&mutex, 0, 1);
    sem_init(&empty, 0, BUFFER_SIZE);
    sem_init(&full, 0, 0);

    // Create producer and consumer threads
    pthread_create(&producer_thread, NULL, producer, NULL);
    pthread_create(&consumer_thread, NULL, consumer, NULL);

    // Join threads
    pthread_join(producer_thread, NULL);
    pthread_join(consumer_thread, NULL);

    // Destroy semaphores
    sem_destroy(&mutex);
    sem_destroy(&empty);
    sem_destroy(&full);

    return 0;
}
```

The **sleep** subroutine suspends the current process for whole seconds. The **usleep** subroutine suspends the current process in microseconds, and the **nsleep** subroutine suspends the current process in nanoseconds.

Observations and DIY code modifications (as per questions In-lab)

a) Compile and run this code from bash command-line.
b) Explain what functionality is implemented by full and empty semaphores.
c) What purpose the function call sleep () is accomplishing in producer and consumer functions?

Code workout # 2:

Synchronization Examples: Readers-Writers Problem

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


#define NUM_READERS 5
#define NUM_WRITERS 2
#define STRING_LENGTH 60

pthread_t readers[NUM_READERS], writers[NUM_WRITERS];
sem_t mutex, rw_mutex;
int readers_count = 0;
FILE *file;

char generateRandomChar() {
    return (char)('a' + rand() % 26); // Generating a random lowercase character
}

void *reader(void *arg) {
    while (1) {
        sem_wait(&mutex);
        readers_count++;
        if (readers_count == 1) {
            sem_wait(&rw_mutex);
        }
        sem_post(&mutex);

        // Reading from file
        fseek(file, 0, SEEK_SET);
        char buffer[256];
        while (fgets(buffer, sizeof(buffer), file) != NULL) {
            fprintf(stdout,"Reader %ld: %s", (long)arg, buffer);
        }

        sem_wait(&mutex);
        readers_count--;
        if (readers_count == 0) {
            sem_post(&rw_mutex);
        }
        sem_post(&mutex);

        // Perform other tasks
        usleep(1000);
    }
}

void *writer(void *arg) {
    while (1) {
        sem_wait(&rw_mutex);

        // generate random string to be written to file
        srand(time(NULL)); // Seed for random number generator
        char randomString[STRING_LENGTH + 1]; // +1 for null terminator
        for (int i = 0; i < STRING_LENGTH; i++) {
            randomString[i] = generateRandomChar();
        }
        randomString[STRING_LENGTH] = '\0'; // Null terminate the string
         // Writing to file
        fseek(file, 0, SEEK_END);
        fprintf(file, "%s\n",randomString); // writer to file on drive
        fprintf(stdout, "Writer %ld: %s\n", (long)arg, randomString); // display
        fflush(file);

        sem_post(&rw_mutex);

        // Perform other tasks
        usleep(1000);
    }
}
```

```
int main() {
    file = fopen("shared_file.txt", "a+");
    if (file == NULL) {
        perror("Error opening file");
        exit(EXIT_FAILURE);
    }

    sem_init(&mutex, 0, 1); sem_init(&rw_mutex, 0, 1);

    int i;
    for (i = 0; i < NUM_WRITERS; i++) pthread_create(&writers[i], NULL, writer, (void *)(long)i);
    for (i = 0; i < NUM_READERS; i++) pthread_create(&readers[i], NULL, reader, (void *)(long)i);

    for (i = 0; i < NUM_READERS; i++) pthread_join(readers[i], NULL);
    for (i = 0; i < NUM_WRITERS; i++) pthread_join(writers[i], NULL);
    fprintf(stdout, "reader pthread join completed\n");


    sem_destroy(&mutex); sem_destroy(&rw_mutex);
    fclose(file);

    return 0;
}
```

Observations and DIY code modifications (as per questions In-lab)

a) Create a file **shared_file.txt** in the directory of the executable file. Write a few lines of text in the file.
b) Compile and execute the file. Observe Writer and Reader process working both on screen output and entries in the shared_file.txt.
c) Dry run the code of functions: i) reader, and ii) writer and submit your handwritten work.
d) Verify that both reader and writer functions ensured the following:
   i. No reader be kept waiting unless a writer has already obtained permission to use the shared object. In other words, no reader should wait for other readers to finish simply because a writer is waiting.
   ii. Once a writer is ready, that writer performs its writing as soon as possible. In other words, if a writer is waiting to access the object, no new readers may start reading.
e) Submit your dry run and answer of part (d) i & ii to your lab instructor.

----------(X)----------