Experiment 7:

1.

```
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <pthread.h>
#define BUFFER_SIZE 1
int buffer[BUFFER_SIZE];
int itemCount = 0;  // 0 = buffer empty, 1 = buffer has item
pthread_mutex_t mutex;
// Producer function
void *producer(void *arg) {
    int item = 1; // A simple item value to produce
    while (1) {
        pthread_mutex_lock(&mutex);
       // Produce only if the buffer is empty
        if (itemCount == 0) {
            buffer[0] = item;
            printf("Producer produced item: %d\n", item);
            itemCount = 1;
            item++;
        }
        pthread_mutex_unlock(&mutex);
        sleep(1);  // Slow down producer
    }
    pthread_exit(NULL);
}
```

```
// Consumer function
void *consumer(void *arg) {
    while (1) {
        pthread_mutex_lock(&mutex);
        // Consume only if buffer has an item
        if (itemCount == 1) {
            int consumedItem = buffer[0];
            printf("Consumer consumed item: %d\n", consumedItem);
            itemCount = 0;
        }
        pthread_mutex_unlock(&mutex);
        sleep(1);  // Slow down consumer
    }
    pthread_exit(NULL);
}
int main() {
    pthread_t producerThread, consumerThread;
    // Initialize the mutex
    pthread_mutex_init(&mutex, NULL);
    // Create producer and consumer threads
    pthread_create(&producerThread, NULL, producer, NULL);
    pthread_create(&consumerThread, NULL, consumer, NULL);
    // Join threads
    pthread_join(producerThread, NULL);
    pthread_join(consumerThread, NULL);
    // Destroy the mutex
    pthread_mutex_destroy(&mutex);
    return 0;
}
```

2.

```
#include <stdio.h>
#include <stdlib.h>
#include <pthread.h>
#include <unistd.h>
#include <sys/types.h>
#include <sys/wait.h>
#include <semaphore.h>
sem_t write_lock;          // Semaphore to ensure only one writer at a time
pthread_mutex_t mutex;      // Mutex for reader count management
int reader_count = 0;       // Counter for active readers
int shared_data = 0;        // Shared resource
// Reader function
void* reader(void* arg) {
    int reader_id = *((int*)arg);
    while (1) {
        // Acquire mutex before modifying reader_count
        pthread_mutex_lock(&mutex);
        reader_count++;
                // If this is the first reader, it locks the write_lock
        if (reader_count == 1) {
            sem_wait(&write_lock);
        }
        pthread_mutex_unlock(&mutex);
        // Reading section
        printf("Reader %d: read shared_data = %d\n", reader_id, shared_data);
        sleep(1); // Simulate reading time
        // Acquire mutex before modifying reader_count
        pthread_mutex_lock(&mutex);
        reader_count--;
        // If this is the last reader, it unlocks the write_lock
        if (reader_count == 0) {
            sem_post(&write_lock);
        }
        pthread_mutex_unlock(&mutex);
        sleep(1); // Simulate time before trying to read again
    }
}
// Writer function
void* writer(void* arg) {
    int writer_id = *((int*)arg);
    while (1) {
        // Wait for exclusive access to the write_lock
        sem_wait(&write_lock);
        // Writing section
        shared_data++;
        printf("Writer %d: updated shared_data to %d\n", writer_id, shared_data);
        sleep(2); // Simulate writing time
        // Release the write_lock after writing
        sem_post(&write_lock);
        sleep(2); // Simulate time before trying to write again
    }
```

```
}
int main() {
    pthread_t readers[5], writers[2];
    int reader_ids[5] = {1, 2, 3, 4, 5};
    int writer_ids[2] = {1, 2};
    // Initialize the semaphore and mutex
    sem_init(&write_lock, 0, 1);
    pthread_mutex_init(&mutex, NULL);
    // Create reader threads
    for (int i = 0; i < 5; i++) {
        pthread_create(&readers[i], NULL, reader, &reader_ids[i]);
    }
    // Create writer threads
    for (int i = 0; i < 2; i++) {
        pthread_create(&writers[i], NULL, writer, &writer_ids[i]);
    }
    // Wait for all threads to complete (in this example, they run indefinitely)
    for (int i = 0; i < 5; i++) {
        pthread_join(readers[i], NULL);
    }
    for (int i = 0; i < 2; i++) {
        pthread_join(writers[i], NULL);
    }
    // Destroy the semaphore and mutex
    sem_destroy(&write_lock);
    pthread_mutex_destroy(&mutex);
    return 0;
```

```
}
```

Experiment 8

1.

```
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <sys/types.h>
#include <sys/stat.h>
#include <pthread.h>
#include <fcntl.h>
#include <string.h>
// Function for the sender thread
void* sender_thread(void* arg) {
    const char* message = "Hello from sender!";
        // Create the named pipe (FIFO) if it doesn't exist
    if (mkfifo("send_pipe", 0666) == -1) {
        perror("Error creating send_pipe");
        pthread_exit(NULL);
    }
    // Open the send_pipe for writing
    int fd = open("send_pipe", O_WRONLY);
    if (fd == -1) {
        perror("Failed to open send_pipe");
        pthread_exit(NULL);
    }
    // Write the message to the pipe
    write(fd, message, strlen(message) + 1);  // Include null terminator
    printf("Sender: Sent message: %s\n", message);
    // Close the pipe
    close(fd);
    pthread_exit(NULL);
}
// Function for the receiver thread
void* receiver_thread(void* arg) {
    char buffer[100];
    // Create the named pipe (FIFO) if it doesn't exist
    if (mkfifo("receive_pipe", 0666) == -1) {
        perror("Error creating receive_pipe");
        pthread_exit(NULL);
    }
    // Open the receive_pipe for reading
    int fd = open("receive_pipe", O_RDONLY);
    if (fd == -1) {
        perror("Failed to open receive_pipe");
        pthread_exit(NULL);
    }

    // Read the message from the pipe
    read(fd, buffer, sizeof(buffer));
    printf("Receiver: Received message: %s\n", buffer);
    // Close the pipe
    close(fd);
    pthread_exit(NULL);
}
```

```
int main() {
    pthread_t sender, receiver;
    // Create the sender and receiver threads
    pthread_create(&sender, NULL, sender_thread, NULL);
    pthread_create(&receiver, NULL, receiver_thread, NULL);
    // Wait for both threads to complete
    pthread_join(sender, NULL);
    pthread_join(receiver, NULL);
    return 0;
}
```

2.

```
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <sys/types.h>
#include <pthread.h>
// Shared memory buffer
char shared_buffer[100];
int data_available = 0; // Flag to indicate data availability
pthread_mutex_t mutex = PTHREAD_MUTEX_INITIALIZER;
pthread_cond_t cond = PTHREAD_COND_INITIALIZER;
// Producer thread function
void* producer(void* arg) {
    // Produce a message
    const char* message = "Hello from producer!";
        // Lock the mutex before accessing the shared buffer
    pthread_mutex_lock(&mutex);
```

```
      // Write the message into the shared buffer
   snprintf(shared_buffer, sizeof(shared_buffer), "%s", message);
   data_available = 1; // Set flag indicating data is available
      printf("Producer: Wrote message to shared memory\n");
      // Signal the consumer that data is available
   pthread_cond_signal(&cond);
      // Unlock the mutex
   pthread_mutex_unlock(&mutex);
      pthread_exit(NULL);
}
// Consumer thread function
void* consumer(void* arg) {
   // Lock the mutex before accessing the shared buffer
   pthread_mutex_lock(&mutex);
     // Wait until the producer signals that data is available
   while (data_available == 0) {
      pthread_cond_wait(&cond, &mutex);
   }
   // Read the message from the shared buffer
   printf("Consumer: Read message from shared memory: %s\n", shared_buffer);
   data_available = 0; // Reset flag indicating data has been consumed
   // Unlock the mutex
   pthread_mutex_unlock(&mutex);
   pthread_exit(NULL);
}
int main() {
   pthread_t producer_thread, consumer_thread;
```

```
        // Create producer and consumer threads
    pthread_create(&producer_thread, NULL, producer, NULL);
    pthread_create(&consumer_thread, NULL, consumer, NULL);
        // Wait for the producer and consumer threads to complete
    pthread_join(producer_thread, NULL);
    pthread_join(consumer_thread, NULL);
        // Clean up and exit
    pthread_mutex_destroy(&mutex);
    pthread_cond_destroy(&cond);
        return 0;
}
```

3.

```
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <sys/types.h>
#include <signal.h>
#include <pthread.h>
// Signal handler for Thread 1
void signal_handler1(int sig) {
    printf("Thread 1: Received signal %d. Performing specific action.\n", sig);
}
// Signal handler for Thread 2
void signal_handler2(int sig) {
   printf("Thread 2: Received signal %d. Performing specific action.\n", sig);
}
// Function for Thread 1
```

```
void* thread1_func(void* arg) {
    // Register signal handler for Thread 1
    signal(SIGUSR1, signal_handler1);
    // Wait for signal indefinitely
    printf("Thread 1: Waiting for signal...\n");
    pause();  // Wait for signal
    pthread_exit(NULL);
}
// Function for Thread 2
void* thread2_func(void* arg) {
    // Register signal handler for Thread 2
    signal(SIGUSR2, signal_handler2);
    // Send signal to Thread 1 after a delay
    sleep(2);
    printf("Thread 2: Sending SIGUSR1 to Thread 1.\n");
    pthread_kill(*(pthread_t*)arg, SIGUSR1);
    // Wait for signal indefinitely
    printf("Thread 2: Waiting for signal...\n");
    pause();  // Wait for signal
    pthread_exit(NULL);
}
int main() {
    pthread_t thread1, thread2;
    // Create Thread 1
    pthread_create(&thread1, NULL, thread1_func, NULL);
    // Create Thread 2 and pass Thread 1 ID to it
    pthread_create(&thread2, NULL, thread2_func, (void*)&thread1);
    // Wait for both threads to complete
    pthread_join(thread1, NULL);
    pthread_join(thread2, NULL);
    return 0;
}
```