#KALINUX#

Practical 1: Process Communication

1. Give solution to the producer-consumer problem using shared memory.

CODE

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

int mutex = 1;
int full = 0;
int empty = 3;
int x = 0;

void producer1();
void consumer1();
int wait(int);
int signal(int);

int main() {
   int n;

   printf("\n1. Producer\n2. Consumer\n3. Exit");

   while (1) {
      printf("\nEnter your choice: ");
      scanf("%d", &n);

      switch (n) {
         case 1:
            if (mutex == 1 && empty != 0)
```

```
            producer1();
        else
            printf("Buffer is full!\n");
        break;

    case 2:
        if (mutex == 1 && full != 0)
            consumer1();
        else
            printf("Buffer is empty!\n");
        break;

    case 3:
        exit(0);

    default:
        printf("Invalid choice. Please select 1, 2, or 3.\n");
        break;
  }
 }

 return 0;
}

int wait(int s) {
 return (--s);
}

int signal(int s) {
 return (++s);
}
```

```
void producer1() {
    mutex = wait(mutex);
    full = signal(full);
    empty = wait(empty);
    x++;
    printf("\nProducer produces the item %d\n", x);
    mutex = signal(mutex);
}

void consumer1() {
    mutex = wait(mutex);
    empty = signal(empty);
    full = wait(full);
    printf("\nConsumer consumes item %d\n", x);
    x += 2;
    mutex = signal(mutex);
}
```

2. Give solution to the producer-consumer problem using message passing

Code

```
#define MAX 100

void producer(void) {
   int item;
   message mesg;
   while (TRUE) {
      item = produce_item();
      create_message(&mesg, item);
      send(consumer, &mesg);
   }
}
```

```
void consumer(void) {
   int item, i;
   message mesg;

   for (i = 0; i < MAX; i++) {
      send(producer, &mesg);
   }

   while (TRUE) {
      receive(producer, &mesg);
      item = extract_item(&mesg);
      consume_item(item);
      send(producer, &mesg);
   }
}
```

Practical 2: Threads

(i) The Java version of a multithreaded program that determines the summation of a non-negative integer. The Summation class implements the Runnable interface. Thread creation is performed by creating an object instance of the Thread class and passing the constructor a Runnable object.

Code

```
#include <stdio.h>
#define MAXSIZE 10

int main() {
   int array[MAXSIZE];
   int i, num, negative_sum = 0, positive_sum = 0;
   float total = 0, average;

   printf("Enter the value of N:\n");
   scanf("%d", &num);

   printf("Enter %d numbers (negative, positive, and zero):\n", num);

   for (i = 0; i < num; i++) {
      scanf("%d", &array[i]);
   }

   printf("Input array elements:\n");
```

```
    for (i = 0; i < num; i++) {
        printf("%+3d ", array[i]);
    }

    /* Summation starts */
    for (i = 0; i < num; i++) {
        if (array[i] < 0) {
            negative_sum = negative_sum + array[i];
        } else if (array[i] > 0) {
            positive_sum = positive_sum + array[i];
        } else if (array[i] == 0) {
            total = total + array[i];
        }
    }

    average = total / num;

    printf("\nNegative Sum: %d\n", negative_sum);
    printf("Positive Sum: %d\n", positive_sum);
    printf("Total: %.2f\n", total);
    printf("Average: %.2f\n", average);

    return 0;
}
```

2.

Write a multithreaded Java program that outputs prime numbers. This program should work as follows: The user will run the program and will enter a number on the command line. The program will then create a separate thread that outputs all the prime numbers less than or equal to the number entered by the user.

Code

```
#include <stdio.h>

int main() {
    int temp = 0, i;
    int nlist[81]; // Adjust the array size according to your needs
    int limit;

    printf("Enter the limit: ");
    scanf("%d", &limit);

    for (i = 2; i <= limit; i++) {
```

```
            nlist[temp] = i;
            temp++;
        }

        for (i = 0; i < temp; i++) {
            if (nlist[i] != 0) {
                int t = i;
                int p = nlist[i];
                while ((t + p) < temp) {
                    t = t + p;
                    nlist[t] = 0;
                }
            }
        }

        printf("Prime numbers within the limit are: ");
        for (i = 0; i < temp; i++) {
            if (nlist[i] != 0) {
                printf("%d, ", nlist[i]);
            }
        }

        printf("\n");

        return 0;
    }
```

(iii) The Fibonacci sequence is the series of numbers 01, 1, 2, Formally, it can be expressed as: fi*b_{0} = 0 fi*b_{1} = 1 5. 8, fi*b_{n} =fib n-1 +fib n.a Write a program that generates the Fibonacci sequence using either the Java or C.

Code

1. (C CODE FOR FIBONACCI)

```
#include <stdio.h>

int main() {
    int n = 10; // Change this value to generate more or fewer Fibonacci numbers
    long long fibonacci[n];
```

```
    fibonacci[0] = 0;
    fibonacci[1] = 1;

    for (int i = 2; i < n; i++) {
        fibonacci[i] = fibonacci[i - 1] + fibonacci[i - 2];
    }

    printf("Fibonacci Sequence:\n");
    for (int i = 0; i < n; i++) {
        printf("%lld ", fibonacci[i]);
    }

    return 0;
}
```

2.(Java code for fibonacci)

```
public class Fibonacci {
    public static void main(String[] args) {
        int n = 10; // Change this value to generate more or fewer Fibonacci numbers
        long[] fibonacci = new long[n];

        fibonacci[0] = 0;
        fibonacci[1] = 1;

        for (int i = 2; i < n; i++) {
            fibonacci[i] = fibonacci[i - 1] + fibonacci[i - 2];
        }

        System.out.println("Fibonacci Sequence:");
        for (int i = 0; i < n; i++) {
```

```
            System.out.print(fibonacci[i] + " ");
        }
    }
}
```

Practical 3: Synchronization

(i) Give Java or C solution to Bounded buffer problem.

Here we add and remove items from the bounded buffer (producer/consumer problem).

Here we remove, add and initialize a bounded buffer for the consumer/producer problem.

Code

```
#include <pthread.h>
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include "bbuffer.h"

#define BOUNDED_BUFFER_SIZE 4

int count;
pthread_mutex_t mutex;
int bounded_buffer[BOUNDED_BUFFER_SIZE];
int in, out;

void initialize_bounded_buffer() {
```

```
    int status;
    count = 0;
    in = 0;
    out = 0;

    status = pthread_mutex_init(&mutex, NULL);

    if (status != 0) {
        fprintf(stderr, "Error creating buffer_mutex\n");
        exit(1); // Exit on error
    }
}

void add_to_buffer(int value) {
    pthread_mutex_lock(&mutex);
    if (count < BOUNDED_BUFFER_SIZE) {
        bounded_buffer[in] = value;
        in = (in + 1) % BOUNDED_BUFFER_SIZE;
        count++;
        printf("added\n");
    } else {
        printf("buffer full\n");
    }
    pthread_mutex_unlock(&mutex);
}

int remove_from_buffer() {
    int value = -1;
    pthread_mutex_lock(&mutex);
    if (count > 0) {
        value = bounded_buffer[out];
```

```
        out = (out + 1) % BOUNDED_BUFFER_SIZE;
        count--;
        printf("removed\n");
    } else {
        printf("could not remove\n");
    }
    pthread_mutex_unlock(&mutex);
    return value;
}
```

(Java code)

```
import java.util.concurrent.locks.*;

public class BoundedBuffer {
    private static final int BOUNDED_BUFFER_SIZE = 4;
    private int count;
    private int in;
    private int out;
    private int[] boundedBuffer;
    private Lock lock;
    private Condition notFull;
    private Condition notEmpty;

    public BoundedBuffer() {
        count = 0;
        in = 0;
        out = 0;
        boundedBuffer = new int[BOUNDED_BUFFER_SIZE];
        lock = new ReentrantLock();
        notFull = lock.newCondition();
```

```
    notEmpty = lock.newCondition();
}

public void add(int value) throws InterruptedException {
    lock.lock();
    try {
        while (count == BOUNDED_BUFFER_SIZE) {
            notFull.await();
        }
        boundedBuffer[in] = value;
        in = (in + 1) % BOUNDED_BUFFER_SIZE;
        count++;
        System.out.println("Added");
        notEmpty.signal();
    } finally {
        lock.unlock();
    }
}

public int remove() throws InterruptedException {
    lock.lock();
    try {
        while (count == 0) {
            notEmpty.await();
        }
        int value = boundedBuffer[out];
        out = (out + 1) % BOUNDED_BUFFER_SIZE;
        count--;
        System.out.println("Removed");
        notFull.signal();
        return value;
```

```
    } finally {
        lock.unlock();
    }
}

public static void main(String[] args) {
    BoundedBuffer buffer = new BoundedBuffer();

    // Example usage
    Thread producerThread = new Thread(() -> {
        try {
            buffer.add(1);
            buffer.add(2);
            buffer.add(3);
        } catch (InterruptedException e) {
            e.printStackTrace();
        }
    });

    Thread consumerThread = new Thread(() -> {
        try {
            int value1 = buffer.remove();
            int value2 = buffer.remove();
            System.out.println("Consumed: " + value1 + ", " + value2);
        } catch (InterruptedException e) {
            e.printStackTrace();
        }
    });

    producerThread.start();
    consumerThread.start();
```

```
    }
}
```

(ii) Give solution to the readers-writers problem using Java synchronization.

Code

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

pthread_t readers[5], writers[5];
sem_t wsem, mutex;
int readcount = 0;

void *reader(void *arg) {
    int c = (int)arg;
    printf("\nReader %d is created\n", c);
    sleep(1);

    sem_wait(&mutex);
    readcount++;

    if (readcount == 1) {
        sem_wait(&wsem);
    }

    sem_post(&mutex);

    // Critical Section (Reading)
```

```
    printf("Reader %d is reading\n", c);
    sleep(1);
    printf("Reader %d finished reading\n", c);

    sem_wait(&mutex);
    readcount--;

    if (readcount == 0) {
        sem_post(&wsem);
    }

    sem_post(&mutex);

    return NULL;
}

void *writer(void *arg) {
    int c = (int)arg;
    printf("\nWriter %d is created\n", c);
    sleep(1);

    sem_wait(&wsem);

    // Critical Section (Writing)
    printf("Writer %d is writing\n", c);
    sleep(1);
    printf("Writer %d finished writing\n", c);

    sem_post(&wsem);

    return NULL;
```

```
}

int main() {
    int a = 1, b = 1;
    sem_init(&wsem, 0, 1);
    sem_init(&mutex, 0, 1);

    for (int i = 0; i < 3; i++) {
        pthread_create(&readers[i], NULL, reader, (void *)a);
        a++;
        pthread_create(&writers[i], NULL, writer, (void *)b);
        b++;
    }

    for (int i = 0; i < 3; i++) {
        pthread_join(readers[i], NULL);
        pthread_join(writers[i], NULL);
    }

    printf("Main terminated\n");

    return 0;
}
```

(JAVA CODE)

```
import java.util.concurrent.Semaphore;

class ReaderWriter {
    static Semaphore wsem = new Semaphore(1);
    static Semaphore mutex = new Semaphore(1);
    static int readcount = 0;
```

```
static class Reader implements Runnable {
    int readerId;

    Reader(int id) {
        readerId = id;
    }

    public void run() {
        try {
            System.out.println("Reader " + readerId + " is created");
            Thread.sleep(1000);

            mutex.acquire();
            readcount++;

            if (readcount == 1) {
                wsem.acquire();
            }

            mutex.release();

            // Critical Section (Reading)
            System.out.println("Reader " + readerId + " is reading");
            Thread.sleep(1000);
            System.out.println("Reader " + readerId + " finished reading");

            mutex.acquire();
            readcount--;

            if (readcount == 0) {
```

```
                wsem.release();
            }

            mutex.release();
        } catch (InterruptedException e) {
            e.printStackTrace();
        }
    }
}

static class Writer implements Runnable {
    int writerId;

    Writer(int id) {
        writerId = id;
    }

    public void run() {
        try {
            System.out.println("Writer " + writerId + " is created");
            Thread.sleep(1000);

            wsem.acquire();

            // Critical Section (Writing)
            System.out.println("Writer " + writerId + " is writing");
            Thread.sleep(1000);
            System.out.println("Writer " + writerId + " finished writing");

            wsem.release();
        } catch (InterruptedException e) {
```

```
                e.printStackTrace();
            }
        }
    }

    public static void main(String[] args) {
        int numReaders = 2;
        int numWriters = 2;

        for (int i = 1; i <= numReaders; i++) {
            new Thread(new Reader(i)).start();
        }

        for (int i = 1; i <= numWriters; i++) {
            new Thread(new Writer(i)).start();
        }
    }
}
```