**Write a shell script to sort an array of numbers using any sort method**

Ans:

# Store the array of numbers in a variable

numbers=(5 3 8 1 6 4 9 7 2)

# Use a for loop to iterate over the array

for ((i=0; i<${#numbers[@]}; i++))

do

# Use another for loop to iterate over the remaining elements in the array

for ((j=i+1; j<${#numbers[@]}; j++))

do

# If the current element is greater than the next element, swap them

if [[ ${numbers[i]} -gt ${numbers[j]} ]]

then

temp=${numbers[i]}

numbers[i]=${numbers[j]}

numbers[j]=$temp

fi

done

done

# Print the sorted array

echo "Sorted array: ${numbers[@]}"

2. Write down Linux commands for following statements: i. Redirect the output of ls command to a file named outfile. Use this as an input file for remaining commands. ii. To select the lines in a file which has digit as one of the character in that line and redirect the output to the file named list. iii. Assign execute permission to owner and remove read permission from other for an ordinary file named test by relative way. iv. Create an alias named rm that always deletes file interactively. v. Count the currently login users to the system

Ans:

i. ls > outfile

ii. grep "[0-9]" outfile > list

iii. chmod u+x,o-r test

iv. alias rm="rm -i"

v. w | wc -l

3. Write down Linux commands for following statements: i. Redirect the output of cat command to a file named outfile. Use this as an input file for remaining commands. ii. List all hidden files under current directory and store in “hidden” file iii. Assign write permission to owner and remove execute permission from other for an ordinary file named test by absolute way. iv. To create soft and hard link for given file v. To convert lowercase to upper case of a given file vi. To extract 1st and 10th character of a given file vii. To display how many times lines are repeated in a given file

Ans:

i. cat > outfile

ii. ls -d .\* > hidden

iii. chmod u+w,o-x /path/to/test

iv. ln file softlink and ln -s file hardlink

v. tr '[:lower:]' '[:upper:]' < file

vi. cut -c1,10 file

vii. uniq -c file

4. Write a program to solve reader-writer problem using Mutex in java

Ans:

import java.util.concurrent.locks.ReentrantLock;

public class ReaderWriterProblem {

private static ReentrantLock mutex = new ReentrantLock();

private static int readers = 0;

// Reader thread

static class Reader implements Runnable {

public void run() {

mutex.lock();

try {

readers++;

} finally {

mutex.unlock();

}

// read data

System.out.println("Reader reading data");

mutex.lock();

try {

readers--;

} finally {

mutex.unlock();

}

}

}

// Writer thread

static class Writer implements Runnable {

public void run() {

mutex.lock();

try {

while (readers > 0) {

// wait until all readers have finished reading

}

// write data

System.out.println("Writer writing data");

} finally {

mutex.unlock();

}

}

}

public static void main(String[] args) {

Thread reader1 = new Thread(new Reader());

Thread reader2 = new Thread(new Reader());

Thread writer1 = new Thread(new Writer());

reader1.start();

reader2.start();

writer1.start();

}

}

The **ReaderWriterProblem** class uses a **ReentrantLock** to synchronize access to the shared data by the reader and writer threads. The **Reader** thread acquires a lock on the mutex, increments the number of readers, releases the lock, reads the data, acquires the lock again, decrements the number of readers, and finally releases the lock.

The **Writer** thread acquires the lock on the mutex, checks if there are any readers currently reading the data, and waits until all readers have finished reading before writing to the data. Finally, it releases the lock on the mutex.

In the **main()** method, we create and start three threads: two **Reader** threads and one **Writer** thread. The program output will show that the writer thread waits until all reader threads have finished reading before writing to the data.

5. Write a program to solve producer-consumer problem using semaphore in java

Ans:

import java.util.concurrent.Semaphore;

public class ProducerConsumer {

static Semaphore semProd = new Semaphore(1);

static Semaphore semCons = new Semaphore(0);

static class Producer extends Thread {

@Override

public void run() {

try {

semProd.acquire();

System.out.println("Producer acquired the semaphore");

// produce something

semCons.release();

System.out.println("Producer released the semaphore");

} catch (InterruptedException e) {

e.printStackTrace();

}

}

}

static class Consumer extends Thread {

@Override

public void run() {

try {

semCons.acquire();

System.out.println("Consumer acquired the semaphore");

// consume something

semProd.release();

System.out.println("Consumer released the semaphore");

} catch (InterruptedException e) {

e.printStackTrace();

}

}

}

public static void main(String[] args) {

new Producer().start();

new Consumer().start();

}

}

In this program, the **Producer** and **Consumer** classes are threads that acquire and release semaphores to ensure that the producer produces something before the consumer consumes it. The **semProd** semaphore is used to ensure that only one producer can produce something at a time, and the **semCons** semaphore is used to ensure that the consumer consumes only after the producer has produced something.

The **main()** method starts both the **Producer** and **Consumer** threads, which run concurrently. The **run()** method of each thread acquires the semaphore it needs before performing its task, and then releases the semaphore after the task is done. This ensures that the producer produces something only when the consumer is not consuming, and vice versa.

Note that this is just a simple example to demonstrate how semaphores can be used to solve the producer-consumer problem in Java. In a real-world scenario, you would need to take care of additional details, such as the synchronization of access to shared resources and handling of potential exceptions.

6. Write a program to solve dining philosopher problem using semaphore in java

Ans:

import java.util.concurrent.Semaphore;

public class DiningPhilosophers {

private static final int NUM\_PHILOSOPHERS = 5;

private static final Semaphore[] forks = new Semaphore[NUM\_PHILOSOPHERS];

public static void main(String[] args) {

// Initialize the semaphores representing the forks

for (int i = 0; i < NUM\_PHILOSOPHERS; i++) {

forks[i] = new Semaphore(1);

}

// Create the philosophers and start them

for (int i = 0; i < NUM\_PHILOSOPHERS; i++) {

Philosopher p = new Philosopher(i);

p.start();

}

}

private static class Philosopher extends Thread {

private final int id;

public Philosopher(int id) {

this.id = id;

}

@Override

public void run() {

while (true) {

try {

// Think for a while

Thread.sleep(500);

// Pick up the left fork

forks[id].acquire();

System.out.println("Philosopher " + id + " picked up left fork.");

// Pick up the right fork

forks[(id + 1) % NUM\_PHILOSOPHERS].acquire();

System.out.println("Philosopher " + id + " picked up right fork.");

// Eat

System.out.println("Philosopher " + id + " is eating.");

Thread.sleep(1000);

// Put down the forks

forks[id].release();

System.out.println("Philosopher " + id + " put down left fork.");

forks[(id + 1) % NUM\_PHILOSOPHERS].release();

System.out.println("Philosopher " + id + " put down right fork.");

} catch (InterruptedException e) {

// Handle the exception

}

}

}

}

}

In this program, each philosopher is represented by a **Philosopher** thread, and each fork is represented by a **Semaphore**. The **Philosopher** thread tries to acquire the left and right forks (represented by semaphores) in turn, and then eats for a short time before releasing the forks. The program uses the **Semaphore** class to ensure that no two philosophers can pick up the same fork at the same time, preventing deadlock.

7. Draw the Gantt charts and compute the finish time, turnaround time andwaitingtime for the following algorithms: A. First come First serve B. Shortest Job First (Non Preemptive)

Ans: