**Lab 9: Deadlock and Concurrency**

1. **Producer Consumer**

|  |
| --- |
| import java.util.\*;  public class ProducerConsumer {  static Scanner sc = new Scanner(System.in);  static int mutex = 1;  static int pos = -1;  static int n = 3;  static String item;  static Stack<String> newBuffer = new Stack<String>();    public static int wait(int s) {  while (s != 1) ;  return (--s);  }  public static int signal(int s) {  return (++s);  }  public static void producer () {  mutex = wait(mutex);  pos = signal(pos);  if (pos < n) {  System.out.print("Enter Item to Produce: ");  String item = sc.next();  System.out.println("Produced item '" + item + "'");  newBuffer.push(item);  }  mutex = signal(mutex);  }  public static void consumer () {  mutex = wait(mutex);  pos--;  if (pos >= -1) {  item = newBuffer.pop();  System.out.println("Consumed item '" + item + "'");  }  mutex = signal(mutex);  }  public static void display () {  if (newBuffer.size() == 0) {  System.out.print("Buffer -> EMPTY");  }  else {  System.out.print("Buffer -> ");  for (String i : newBuffer) {  System.out.print(i + " ");  }  System.out.println();  }  }  public static void main(String[] args) {  System.out.print("Enter Buffer size : ");  n = sc.nextInt();  System.out.println("\n1. Producer\n2. Consumer\n3. Display Buffer\n4. Exit");  boolean loop = true;  while (loop) {  System.out.print("\nEnter your choice: ");  int choice = sc.nextInt();  switch (choice) {  case (1) -> {  if (mutex == 1 && (pos+1) < n) {  producer();  } else {  System.out.println("Buffer is full, There's no space to Produce!");  }  }  case (2) -> {  if (mutex == 1 && pos >= 0) {  consumer();  } else {  System.out.println("Buffer is empty, There's nothing to Consume!");  }  }  case (3) -> display();  case (4) -> {  System.out.println("\nThank You!");  loop = false;  }  default -> System.out.println("Please Enter correct Choice");  }  }  }  } |
| **Output:** |

1. **Bankers Algorithm**

|  |
| --- |
| #include<stdio.h>  #include<stdbool.h>  int P = 5;  int R = 3;  *//finding needs of each process*  void calculateNeed(int need[P][R], int max[P][R], int allot[P][R]) {      for (int i = 0 ; i < P ; i++) {          for (int j = 0 ; j < R ; j++) {              need[i][j] = max[i][j] - allot[i][j];          }        }  }  *// Function to find the system is in safe state or not*  bool isSafe(int processes[], int avail[], int max[P][R], int allot[P][R]) {      int need[P][R];      calculateNeed(need, max, allot);        bool finish[5] = {0,0,0,0,0};      bool found;      int safeSeq[P];        int work[R];      for (int i = 0; i < R ; i++){          work[i] = avail[i];      }        int count = 0;      while (count < P) {          found = false;          for (int i = 0; i < P; i++) {              if (finish[i] == 0) {                  int j;                  for (j = 0; j < R; j++) {                      if (need[i][j] > work[j]) {                          break;                      }                  }                    if (j == R) {                        for (int k = 0 ; k < R ; k++) {                          work[k] += allot[i][k];                      }                        safeSeq[count++] = i;                      finish[i] = 1;                      found = true;                  }              }          }      }        if (found == false) {          printf("System is not in safe state");          return false;      }        printf("System is in safe state.\n");      printf("Safe sequence is: ");      for (int i = 0; i < P ; i++) {        printf("%d ", safeSeq[i]);      }      return true;  }  void main() {      int processes[] = {0, 1, 2, 3, 4};    *// Available matrix*      int avail[] = {3, 3, 2};    *// max matrix*      int max[5][3] = {      {7, 5, 3},      {3, 2, 2},      {9, 0, 2},      {2, 2, 2},      {4, 3, 3}};    *// allotted matrix*      int allot[5][3] = {      {0, 1, 0},      {2, 0, 0},      {3, 0, 2},      {2, 1, 1},      {0, 0, 2}};        isSafe(processes, avail, max, allot);  }  **Output:** |