|  |
| --- |
| **Lab Record**    **of**    **Operating System**  **(CAF614)**      **Submitted to: Submitted by:**  Mr. Ravi Dhaundiyal Mohit Kumar  Assistant Prof. 1000021153  School of Computing **MCA**-**A-P1**  DIT University 1st Year (2nd Sem)  **Session 2023-24** |
| **Index**   |  |  |  |  | | --- | --- | --- | --- | | **S.No** | **Title of Experiment/Objective** | **Date of Conduction** | **Signature of Faculty** | | 1.a. | Implement the concept of read, write, open system calls using C programming in Linux environment only. | 17-01-2024 |  | | 1.b. | Implement the concept of fork and wait system calls using C programming in Linux environment only. | 24-01-2024 |  | | 2. | Implement the following algorithm FCFS, SJF, Round Robin, Priority in Linux. | 07-02-2024 |  | | 3. | Implement the concept of threading in OS. Prefer threading in JAVA only. |  |  | | 4. | Write a Java program to simulate producer-consumer problems using semaphores. |  |  | | 5. | Write a Java program to simulate the concept of Dining Philosopher ‘s problem. |  |  | | 6. | Write a program using Linux to simulate Banker ‘s algorithm. |  |  | | 7. | Write a C program using Linux to simulate the following contiguous memory  allocation techniques:   1. Worst fit 2. Best fit 3. First fit. |  |  | | 8. | Write a Java program to simulate the disk scheduling algorithms:  a. FCFS  b. SCAN  c. C-scan |  |  | | 9. | Write a C program using Linux to implement page replacement algorithms:  a. FIFO  b. LRU  c. LFU |  |  | | 10. | Write a C program to compare the Optimal page replacement algorithm with FIFO and LRU page replacement algorithms. |  |  |     **Practical -1.a**  **Objective –** Implement the concept of read, write, open system calls using C programming in Linux environment only.  **Program:**  #include <stdio.h>  #include <stdlib.h>  #include <fcntl.h>  #include <unistd.h>  int main() {  int fileDescriptor = open("example.txt", O\_CREAT | O\_WRONLY | O\_TRUNC, 0666);  if (fileDescriptor == -1) {  perror("Error opening file");  exit(EXIT\_FAILURE);  }  const char \*dataToWrite = "Hello, Mohit!";  ssize\_t bytesWritten = write(fileDescriptor, dataToWrite, strlen(dataToWrite));  if (bytesWritten == -1) {  perror("Error writing to file");  close(fileDescriptor);  exit(EXIT\_FAILURE);  }  printf("%zd bytes written to the file.\n", bytesWritten);  close(fileDescriptor);  fileDescriptor = open("example.txt", O\_RDONLY);  if (fileDescriptor == -1) {  perror("Error opening file for reading");  exit(EXIT\_FAILURE);  }  char buffer[256];  ssize\_t bytesRead = read(fileDescriptor, buffer, sizeof(buffer));  if (bytesRead == -1) {  perror("Error reading from file");  close(fileDescriptor);  exit(EXIT\_FAILURE);  }  buffer[bytesRead] = '\0';  printf("Read from file: %s\n", buffer);  close(fileDescriptor);  return 0;  }  Output: |
| **Practical -1.b**  **Objective** - Implement the concept of fork and wait system calls using C programming in Linux environment only.  **Program:**  #include <stdio.h>  #include <stdlib.h>  #include <unistd.h>  #include <sys/wait.h>  int main() {  pid\_t childPid = fork();  if (childPid == -1) {  perror("Error during fork");  exit(EXIT\_FAILURE);  }  if (childPid == 0) {  printf("Child process: PID = %d\n", getpid());  printf("Child process: Parent PID = %d\n", getppid());  sleep(2);  printf("Child process: Exiting\n");  exit(EXIT\_SUCCESS);  } else {  printf("Parent process: PID = %d\n", getpid());  printf("Parent process: Child PID = %d\n", childPid);  int status;  pid\_t terminatedChild = waitpid(childPid, &status, 0);  if (terminatedChild == -1) {  perror("Error during wait");  exit(EXIT\_FAILURE);  }  if (WIFEXITED(status)) {  printf("Parent process: Child process exited with status %d\n", WEXITSTATUS(status));  } else if (WIFSIGNALED(status)) {  printf("Parent process: Child process terminated by signal %d\n", WTERMSIG(status));  }  }  return 0;  }  **Output:**    **Practical – 2**  **Objective** - Implement the following algorithm FCFS, SJF, Round Robin, Priority in Linux.  **FCFS:**  #include<stdio.h>  void findWaitingTime(int processes[], int n, int bt[], int wt[]) {  wt[0] = 0;  for (int i = 1; i < n ; i++)  wt[i] = bt[i-1] + wt[i-1];}  void findTurnAroundTime(int processes[], int n, int bt[], int wt[], int tat[]) {  for (int i = 0; i < n ; i++)  tat[i] = bt[i] + wt[i];}  void findavgTime(int processes[], int n, int bt[]) {  int wt[n], tat[n];  findWaitingTime(processes, n, bt, wt);  findTurnAroundTime(processes, n, bt, wt, tat);  printf("Processes Burst time Waiting time Turn around time\n");  int total\_wt = 0, total\_tat = 0;  for (int i = 0; i < n; i++) {  total\_wt = total\_wt + wt[i];  total\_tat = total\_tat + tat[i];  printf(" %d ", (i+1));  printf(" %d ", bt[i]);  printf(" %d", wt[i]);  printf(" %d\n", tat[i]);}  printf("Average waiting time = %f\n", (float)total\_wt / (float)n);  printf("Average turn around time = %f\n", (float)total\_tat / (float)n);}  int main() {  int processes[] = {1, 2, 3};  int n = sizeof processes / sizeof processes[0];  int burst\_time[] = {10, 5, 8};  findavgTime(processes, n, burst\_time);  return 0;  }  **Output:**    **SJF(Shortest Job First):**  #include<stdio.h>  void findWaitingTime(int processes[], int n, int bt[], int wt[]) {  int rt[n];  for (int i = 0; i < n; i++)  rt[i] = bt[i];  int complete = 0, t = 0, minm = 999999;  int shortest = 0, finish\_time;  bool check = false;  while (complete != n) {  for (int j = 0; j < n; j++) {  if ((rt[j] <= t) && (rt[j] < minm) && (rt[j] > 0)) {  minm = rt[j];  shortest = j;  check = true;  }  }  if (check == false) {  t++;  continue;  }  rt[shortest]--;  minm = rt[shortest];  if (minm == 0)  minm = 999999;  if (rt[shortest] == 0) {  complete++;  check = false;  finish\_time = t + 1;  wt[shortest] = finish\_time - bt[shortest];  if (wt[shortest] < 0)  wt[shortest] = 0;  }  t++;  }  }  void findTurnAroundTime(int processes[], int n, int bt[], int wt[], int tat[]) {  for (int i = 0; i < n; i++)  tat[i] = bt[i] + wt[i];  }  void findavgTime(int processes[], int n, int bt[]) {  int wt[n], tat[n];  findWaitingTime(processes, n, bt, wt);  findTurnAroundTime(processes, n, bt, wt, tat);  printf("Processes Burst time Waiting time Turn around time\n");  int total\_wt = 0, total\_tat = 0;  for (int i = 0; i < n; i++) {  total\_wt = total\_wt + wt[i];  total\_tat = total\_tat + tat[i];  printf(" %d ", (i+1));  printf(" %d ", bt[i]);  printf(" %d", wt[i]);  printf(" %d\n", tat[i]);  }  printf("Average waiting time = %f\n", (float)total\_wt / (float)n);  printf("Average turn around time = %f\n", (float)total\_tat / (float)n);  }  int main() {  int processes[] = {1, 2, 3};  int n = sizeof processes / sizeof processes[0];  int burst\_time[] = {6, 8, 7};  findavgTime(processes, n, burst\_time);  return 0;  }  **Output:**    **Round Robin:**  #include<stdio.h>  void findWaitingTime(int processes[], int n, int bt[], int wt[], int quantum) {  int rem\_bt[n];  for (int i = 0; i < n; i++)  rem\_bt[i] = bt[i];  int t = 0;  while (1) {  bool done = true;  for (int i = 0; i < n; i++) {  if (rem\_bt[i] > 0) {  done = false;  if (rem\_bt[i] > quantum) {  t += quantum;  rem\_bt[i] -= quantum;  }  else {  t = t + rem\_bt[i];  wt[i] = t - bt[i];  rem\_bt[i] = 0;  }  }  }  if (done == true)  break;  }  }  void findTurnAroundTime(int processes[], int n, int bt[], int wt[], int tat[]) {  for (int i = 0; i < n; i++)  tat[i] = bt[i] + wt[i];  }  void findavgTime(int processes[], int n, int bt[], int quantum) {  int wt[n], tat[n];  findWaitingTime(processes, n, bt, wt, quantum);  findTurnAroundTime(processes, n, bt, wt, tat);  printf("Processes Burst time Waiting time Turn around time\n");  int total\_wt = 0, total\_tat = 0;  for (int i = 0; i < n; i++) {  total\_wt = total\_wt + wt[i];  total\_tat = total\_tat + tat[i];  printf(" %d ", (i+1));  printf(" %d ", bt[i]);  printf(" %d", wt[i]);  printf(" %d\n", tat[i]);  }  printf("Average waiting time = %f\n", (float)total\_wt / (float)n);  printf("Average turn around time = %f\n", (float)total\_tat / (float)n);  }  int main() {  int processes[] = {1, 2, 3};  int n = sizeof processes / sizeof processes[0];  int burst\_time[] = {10, 5, 8};  int quantum = 2;  findavgTime(processes, n, burst\_time, quantum);  return 0;  }  **Output:**    **Priority Scheduling:**  #include<stdio.h>  void findWaitingTime(int processes[], int n, int bt[], int wt[], int priority[]) {  int rt[n];  for (int i = 0; i < n; i++)  rt[i] = bt[i];  int complete = 0, t = 0, minm = 999999;  int shortest = 0, finish\_time;  bool check = false;  while (complete != n) {  for (int j = 0; j < n; j++) {  if ((rt[j] <= t) && (priority[j] < minm) && (rt[j] > 0)) {  minm = priority[j];  shortest = j;  check = true;  }  }  if (check == false) {  t++;  continue;  }  rt[shortest]--;  minm = priority[shortest];  if (rt[shortest] == 0) {  complete++;  check = false;  finish\_time = t + 1;  wt[shortest] = finish\_time - bt[shortest];  if (wt[shortest] < 0)  wt[shortest] = 0;  }  t++;  }  }  void findTurnAroundTime(int processes[], int n, int bt[], int wt[], int tat[]) {  for (int i = 0; i < n; i++)  tat[i] = bt[i] + wt[i];  }  void findavgTime(int processes[], int n, int bt[], int priority[]) {  int wt[n], tat[n];  findWaitingTime(processes, n, bt, wt, priority);  findTurnAroundTime(processes, n, bt, wt, tat);  printf("Processes Burst time Waiting time Turn around time\n");  int total\_wt = 0, total\_tat = 0;  for (int i = 0; i < n; i++) {  total\_wt = total\_wt + wt[i];  total\_tat = total\_tat + tat[i];  printf(" %d ", (i+1));  printf(" %d ", bt[i]);  printf(" %d", wt[i]);  printf(" %d\n", tat[i]);  }  printf("Average waiting time = %f\n", (float)total\_wt / (float)n);  printf("Average turn around time = %f\n", (float)total\_tat / (float)n);  }  int main() {  int processes[] = {1, 2, 3};  int n = sizeof processes / sizeof processes[0];  int burst\_time[] = {10, 5, 8};  int priority[] = {3, 2, 1}; // Lower integer means higher priority  findavgTime(processes, n, burst\_time, priority);  return 0;  }  **Output:**    **Practical – 3**  **Objective**: Implement the concept of threading in OS. Prefer threading in JAVA only.  **Program:**  public class MyThread implements Runnable {    private String threadName;  public MyThread(String threadName) {  this.threadName = threadName;  }  public void run() {  System.out.println("Thread " + threadName + " is running.");  try {  Thread.sleep(2000);  } catch (InterruptedException e) {  System.out.println("Thread " + threadName + " interrupted.");  }  System.out.println("Thread " + threadName + " exiting.");  }  public static void main(String[] args) {  System.out.println("Main thread starts.");  Thread thread1 = new Thread(new MyThread("Thread 1"));  Thread thread2 = new Thread(new MyThread("Thread 2"));  thread1.start();  thread2.start();  try {  thread1.join();  thread2.join();  } catch (InterruptedException e) {  System.out.println("Main thread interrupted.");  }    System.out.println("Main thread exits.");  }  }  **Output:**  **Practical – 4**  **Objective:** Write a Java program to simulate producer-consumer problems.  **Program:**  import java.util.LinkedList;  public class ProducerConsumer {  private LinkedList<Integer> buffer = new LinkedList<>();  private int capacity;  public ProducerConsumer(int capacity) {  this.capacity = capacity;  }  public void produce() throws InterruptedException {  int value = 0;  while (true) {  synchronized (this) {  while (buffer.size() == capacity) {  wait();  }  System.out.println("Producer produced: " + value);  buffer.add(value++);  notify();  Thread.sleep(1000); // Simulating production time  }  }  }  public void consume() throws InterruptedException {  while (true) {  synchronized (this) {  while (buffer.isEmpty()) {  wait();  }  int value = buffer.removeFirst();  System.out.println("Consumer consumed: " + value);  notify();  Thread.sleep(1500); // Simulating consumption time  }  }  }  public static void main(String[] args) {  ProducerConsumer pc = new ProducerConsumer(5); // Buffer capacity of 5  Thread producerThread = new Thread(() -> {  try {  pc.produce();  } catch (InterruptedException e) {  e.printStackTrace();  }  });  Thread consumerThread = new Thread(() -> {  try {  pc.consume();  } catch (InterruptedException e) {  e.printStackTrace();  }  });  producerThread.start();  consumerThread.start();  }  }  **Output:** |