Text

Description automatically generated with medium confidence

**CPEN 452 Operating Systems**

Simulation of Operating System Algorithms for Process Management, Memory Page Replacement, and Disk Scheduling using C++ Program

School of Engineering

Department of Computer Engineering

Carlos Roque

Faustino R. Fuertes Millán

Introduction

Simulating operating system algorithms for process management, memory page replacement, and disk scheduling using C++ program is essential to understand the behavior of these algorithms under different conditions. This understanding can help system designers to make informed decisions about selecting the appropriate algorithms for specific system requirements. Moreover, simulating these algorithms can also aid in the identification of potential performance bottlenecks, allowing for the development of more efficient algorithms to overcome these limitations.

The simulation of OS algorithms has numerous applications. For example, it can be used to evaluate the performance of an OS in a virtualized environment, where multiple operating systems are running on a single physical machine. It can also be used to simulate the behavior of different scheduling algorithms in real-time operating systems, where the timeliness of response is critical. Additionally, it can be used to evaluate the effectiveness of memory page replacement algorithms in a virtual memory system, where the available memory is limited.

Summary

Project Design

1. Data dictionary diagram

|  |  |
| --- | --- |
| Data | Description |
| calculateNonPreemptive | Calculate Waiting Time with Non-preemption.This function will caculate the waiting time for each process sequentially in the order that they are passed.Will return the waiting time |
| findWaitingTimeRoundRobin | Function to calculate waiting times of processes. |
| Process | Struct for each node in a linked list |
| generateProcessList | Generate Process List: This method will generate a LinkedList by first asking the user how many Nodes to add (i.e Processes to add) and then for each, ask the user for the burst time It first creates a head node as the starting point.Will return head. |
| printSubMenu | Print the sub menu.After the user has selected the option from the first menu, this method will be called and it will hold a string array that contains the names of the algorithms that can be used. Return selection. |
| printMenu | Print Main Menu. This is the first menu that will be shown to the user for selection. It will hold 3 options where each has their own correspondent algorithm for processing. Return NULL |
| dFCFS | Function to calculate the total number of head movements and print the sequence of movements. Return total\_values |
| sstf | Selects the next request to be serviced based on the shortest seek time from the current head position. |
| cscan | Takes an array of disk requests as input and returns the total number of disk movements required to service all the requests. Starting from the head position and moving towards the smallest request. Return movements. |
| scan | The algorithm works by first setting the initial head position as the first request in the array, and then sorting the array in ascending order.Starting from the head position and moving towards the largest request. Return movements. |
| generateNewList | to create lists of processes |
| deleteLinkedList | to delete lists of processes |
| firstComeFirstServe | This method will receive a linked list in order as presented and return an array, or rather a pointer to the first element in the array that contains the waiting time for each process. |
| shortestJobFirst | This function will calculate the waiting time of each process in a linkedlist: It will first take the arrival time and the burst time to decide which processwill go first and at what time (ms) the dispatcher will step over to the second process considering the size of its burst time |
| priorityScheduling | This function will receive a Sorted linked list of process in the order based off the lowest priority among them |
| roundRobin | This function will calculate the waiting time and turnaround time of a process it receives from a linked list and will schedule them using Round Robin considering the Quantum time |
| sortByPriority | ort Processes by Priority This method will receive an unsorted linked list and reorder it based on their priority. The algorithm used for sorting is Selection Sort, which will take the smallest number in the linked list and will swap it with the first element in the linked list. Best and worst case for it is O(n^2), unless the list is already sorted in which case the time complexity is O(n) |
| sortByburstTime | Sort a linked list of processes based on their burst time in ascending order.Return head; |
| processReconstruct | This function takes a parameter of type process which is a singly linked list by default. The function will then reconstruct it so that it can become a Circular Doubly Linked List of processes.Return head. |
|  |  |
|  |  |

1. UML(OOP) diagrams and data flow diagrams (structured programming)

UML DIAGRAM:

Diagram

Description automatically generated

1. Adequate code documentation: explanation of each class, data structure, module, function, and/or procedure

In **calculate.h** file we see that is made up of two functions, calculateNonPreemptive and findWaitingTimeRoundRobin.



* Calculates how long each process has to wait its turn to run, based on how long all the previous processes took. This is called non-preemptive because once a process starts running, it cannot be interrupted until it finishes.



* Function is used in a scheduling algorithm called Round Robin. This algorithm runs each process for a fixed amount of time, called a quantum, and then switches to the next process in the queue. If a process hasn't finished after one quantum, it goes back to the end of the queue to wait its turn again. This function keeps track of how long each process must wait before it can start running, and how long it takes for each process to finish running.

In **disk.h** file we implemented four disk scheduling algorithms: FCFS (First Come-First-Serve), SSTF (Shortest-Seek-Time-First), SCAN, and CSCAN (Circular-SCAN)



* Implements the "first-come, first-served" algorithm. This algorithm simply processes requests in the order they are received, with no regard for their location on the disk. It calculates the total number of head movements required by adding up the absolute differences between adjacent requests.



* Implements the "shortest seek time first" algorithm. This algorithm services requests based on their proximity to the current head position. It sorts the remaining requests in ascending order of distance from the current position and selects the request with the smallest distance from the current position. It then moves the head to that request and repeats the process until all requests have been serviced.



* Implements the "elevator" or "scan" algorithm. This algorithm services requests in one direction until it reaches the edge of the disk, at which point it reverses direction and services requests in the opposite direction until it reaches the other edge of the disk. It sorts the requests in ascending order and starts servicing them from the current head position in the direction of the closest disk edge. When it reaches the edge, it changes direction and services requests in the opposite direction until it reaches the other edge.



* It sorts the requests in ascending order, divides them into two arrays (those less than the current position, and those greater than or equal to it), and processes each array separately in a circular manner (moving right from the current position to the highest request, and then moving left from the highest request to the lowest request). It returns the total number of head movements required.

Development

Links to Github repository:

<https://github.com/iSuperstar/Proyecto_Operating_Systems>

Links for the files:

Calculate.h: <https://github.com/iSuperstar/Proyecto_Operating_Systems/blob/main/calculate.h>

disk.h:

<https://github.com/iSuperstar/Proyecto_Operating_Systems/commit/b122e72c90ac0f22618a288058f5f86975197b42>

main.cpp:

<https://github.com/iSuperstar/Proyecto_Operating_Systems/commit/b122e72c90ac0f22618a288058f5f86975197b42>