Task 2

Text Parsing

**Multithreaded Word Counting Program**

**Concept:**

* This program reads a text file and counts the frequency of each word using multiple threads.
* It utilizes **parallel processing** by assigning each sentence segment to a different thread to improve performance.
* To avoid race conditions while updating shared data (wordCount), it uses a **mutex** for thread-safe operations.

**Implementation Steps:**

1. **Read File Content:**
   * The read\_file() function opens and reads the entire contents of a text file into a string.
2. **Segment the Text:**
   * The segment() function splits the input text into segments (sentences), using periods (.) as delimiters.
3. **Launch Threads:**
   * Each text segment is assigned to a separate thread using std::thread.
   * Threads execute the countWords() function.
4. **Process Words in Threads:**
   * Each word in the segment is:
     + Converted to lowercase.
     + Counted in a shared map<string, int> wordCount.
   * A std::mutex (mtx) ensures only one thread updates the map at a time to avoid data corruption.
   * Execution time for each thread is calculated and printed.
5. **Join Threads:**
   * The main thread waits for all threads to complete using join().
6. **Output Results:**
   * After all threads finish, the final word frequencies are printed to the console.

**Observations:**

* **Different Load Distribution:**  
  Since text segments are split based on sentence boundaries, the amount of work per thread may vary .

Task 3

Paging Simulation Report

Overview

This program simulates a paging system using the Aging Algorithm for page replacement. It takes a sequence of page references from a file and simulates how pages are loaded into memory frames. The program measures the number of page faults for different frame sizes and provides output accordingly.

Key Variables

* numFrames: The number of available page frames in memory.
* frames: A vector that holds the current pages loaded in memory.
* pageMap: A mapping between page IDs and their index in the frame list.
* age: An 8-bit aging counter used to determine which page should be replaced.
* pageFaults: A counter that keeps track of the number of page faults encountered.

Data Structures

* Page Class: Represents a page in memory with an ID and an aging counter.
* AgingAlgorithm Class: Implements the Aging Algorithm for page replacement.

Algorithm Description

Reading the Input File

* The program reads a sequence of page references from a file.
* These references represent requests made by processes for specific pages.

First Loop: Updating Aging Counters

* Each page's age counter is right-shifted (>> 1) to reduce its priority.
* If a page is accessed, its most significant bit (MSB) is set to 1.

Second Loop: Checking and Handling Page Requests

* If a page is already in memory, its age is updated.
* If a page is not in memory, a page fault occurs, and the least recently used page is replaced based on the aging counter.
* The replaced page’s ID is removed from pageMap, and the new page is inserted with the highest priority.

Deadlock Prevention

* Since this is a paging simulation, deadlocks do not occur as pages are replaced dynamically.
* The algorithm ensures that even if frames are full, an older page is replaced to accommodate new requests.

Output and Results

* The program runs the simulation for different frame sizes (e.g., 2, 4, 8, 16).
* It prints the number of page faults for each frame size, helping analyze performance.

Main Function Execution

1. Reads the page reference sequence from a file.
2. Iterates over different frame sizes.
3. Runs the Aging Algorithm for each frame size and tracks page faults.
4. Outputs the number of page faults for each scenario.

Conclusion

This paging simulation effectively demonstrates how the Aging Algorithm manages memory by replacing pages based on an aging mechanism. The program highlights the impact of varying frame sizes on page faults, helping understand memory performance in a paging system

Task 4

Paging Sim

**Introduction**

This program simulates a FIFO (First-In-First-Out) page replacement algorithm and generates page reference sequences with a given probability of repeated accesses. The purpose is to evaluate page faults under different conditions using local and global FIFO strategies.

**Function: generate\_page**

**Purpose:** Generates multiple sequences of page references based on a probability that determines whether the next page remains the same or changes.

**Explanation:**

1. The function initializes a vector of vectors (pages) to store multiple page sequences.
2. srand(time(0)) is used to ensure randomization across runs.
3. The range of valid page numbers is determined by higher\_range - lower\_range + 1.
4. A loop runs for num\_of\_page\_sequences, generating individual sequences:
   * A random starting number is selected.
   * For each subsequent page reference, a probability check determines whether to repeat the current page or select a new one.
   * A do-while loop ensures that when a new page is chosen, it differs from the previous one.
5. The function returns the generated sequences.

**Function: fifo\_page\_replacement**

**Purpose:** Simulates the FIFO page replacement algorithm for a single sequence and calculates the number of page faults.

**Explanation:**

1. Uses a queue (frame) to maintain page order and an unordered\_set (current\_pages) for fast lookups.
2. Iterates through the page reference sequence:
   * If the page is not in the set, a page fault occurs.
   * If the frame is full, the oldest page is removed from both the queue and the set.
   * The new page is then added to both.
3. The function returns the total page faults encountered.

**Function: fifo\_global**

**Purpose:** Simulates a global FIFO page replacement strategy where multiple processes share the same memory.

**Explanation:**

1. Similar to fifo\_page\_replacement, but processes are iterated sequentially.
2. Keeps track of the total pages processed.
3. If a page is not in memory and the frame is full, the oldest page is removed.
4. The function returns the total number of page faults encountered.

**Main Function**

**Purpose:** Coordinates the program execution by generating page sequences and evaluating them under FIFO policies.

**Explanation:**

1. Defines parameters for sequence generation (length, number of sequences, range, and probability).
2. Calls generate\_page to create test data.
3. Prints the generated sequences for verification.
4. Loops through each sequence:
   * Calls fifo\_page\_replacement with a frame size of 3 and prints page fault results.
   * Calls fifo\_global with a frame size of 4 and prints results.
5. The output provides insight into how FIFO performs for different sequences.

**Conclusion**

The program successfully simulates FIFO page replacement strategies. The local FIFO strategy manages each process separately, while the global FIFO strategy shares frames across all processes. By analyzing page fault counts, this program provides a clear comparison of memory management efficiency under varying conditions.

Task 5

Group by Bandwidth (directory)

**Introduction** This program analyzes file sizes in a given directory and categorizes them into bandwidth increments. The purpose is to count the number of files within each bandwidth range and display the results accordingly.

**Function: print\_file\_sizes** **Purpose:** Processes files in a directory and groups them based on size ranges.

**Explanation:**

1. Initializes a map to store file names and their corresponding sizes.
2. Iterates through the given directory using filesystem::directory\_iterator to retrieve file sizes.
3. Determines the largest file size to define the range of bandwidths.
4. Iterates through file size ranges, checking how many files fall within each bandwidth interval.
5. Uses an iterator to safely remove processed files from the map to avoid redundant calculations.
6. Outputs the number of files within each bandwidth range.

**Main Function** **Purpose:** Calls print\_file\_sizes with a specified directory and bandwidth increment.

**Explanation:**

1. Defines a directory path where the file size analysis will be performed.
2. Calls print\_file\_sizes with a bandwidth\_increments value to categorize files.
3. Displays the results of file distribution across different size ranges.

**Conclusion** This program effectively categorizes and counts files within predefined size intervals. By leveraging the C++ filesystem library, it efficiently processes directory contents and provides insights into file size distribution. The use of a map ensures structured storage of file information, and the iterator-based approach optimizes processing by avoiding unnecessary re-iterations.

Task 6

Print Directory

* The program lists files in a specified directory using the C++ filesystem library.
* It handles multiple directories as command-line arguments.
* Error handling is implemented for invalid directories or permission issues.

Functionality

The program defines a function list\_files that takes a directory path as input and iterates through its contents. It prints each filename found in the directory.

void list\_files(const string& directory) {

try {

for (const fs::directory\_entry& entry : fs::directory\_iterator(directory)) {

cout << entry.path().filename().string() << endl;

}

} catch (const fs::filesystem\_error& exception\_error) {

cerr << "Error: " << exception\_error.what() << endl;

}

}

Error Handling

* If no directory is provided, the program outputs an error message.
* If a directory does not exist or cannot be accessed, an exception is caught and an error is displayed.

int main(int files\_count, char\* files[]) {

if (files\_count < 2) {

cerr << "Error: No directory provided. Please enter a valid directory." << endl;

return 1;

}

for (int i = 1; i < files\_count; i++) {

cout << files[i] << ":" << endl;

list\_files(files[i]);

cout << endl;

}

return 0;}

Execution

* Compile the program using a C++ compiler with filesystem support (C++17 or later).
* Run the executable, providing directory paths as arguments.
* Example usage: ./program\_name /path/to/directory1 /path/to/directory2

The program will print the filenames of each directory provided as input.

Task 7

Deadlock Detection

* The finished array tracks which processes have successfully completed.
* finished\_count helps determine when all processes are done.
* available holds the current availability of resources

A white square with red writing on it

AI-generated content may be incorrect.

This matrix represents our allocation and request matrices. Where each processes has a row and each column is the amount of resources of the respective column. The numbers represent the way we are moving (iterations)

The first nested for loop:

help us iterate over the matrix and subtracts the resources already allocated to processes from the total system resources moving column by column, so for example we finish all the A’s them move over to B’s etc.

for (int i = 0; i < resource\_types; i++)

{

available[i] = num\_resources[i];

for (int j = 0; j < num\_process; j++) {

available[i] -= matrix\_allocation[j][i]; }

}

Second Nested for Loop :

* This loop iterates through all the unfinished processes.
* Checks if the process’s resource request can be satisfied using the currently available resources.

for (int i = 0; i < num\_process; i++) {

if (!finished[i]) {

bool have\_enough\_resources = true;

// Check if the process can be satisfied

for (int j = 0; j < resource\_types; j++) {

if (matrix\_requests[i][j] > available[j]) {

have\_enough\_resources = false;

break;

}

}

Allocating Requested Resources and Marking done:

* If a process's request can be fulfilled, it is marked as complete.
* The resources allocated to it are released back to the system.

if (have\_enough\_resources) {

for (int j = 0; j < resource\_types; j++)

{

available[j] += matrix\_allocation[i][j]; // return the allocated resources

}

finished[i] = true;

finished\_count++;

any\_process\_completed = true;

}

}

Deadlock Detection Check

* If no process could proceed in the current iteration, a deadlock is detected.
* The function prints "Deadlock detected!" and exits.
* If all processes finish successfully, the system is free of deadlocks.

if (!any\_process\_completed) {

cout << "Deadlock detected!";

return;}

cout << "No Deadlock detected!";

The Main Function

* num\_process and resource\_types store the number of processes and resource types.
* num\_resources[10] holds the total available resources for each type.
* matrix\_allocation[10][10] represents the resources allocated to each process.
* matrix\_requests[10][10] contains the resource requests made by processes
* Initialize the path to the file and then we call on the functions .

int num\_process, resource\_types;

int num\_resources[10], matrix\_allocation[10][10], matrix\_requests[10][10]; // Intialize the arrays with random sizes (not the most efficient)

string filename = "C:\\Users\\salla\\OneDrive\\Desktop\\DeadLock\_Detection\\Deadlock\_Detection\\Deadlock.txt";

read\_parameters(filename, num\_process, resource\_types, num\_resources, matrix\_allocation, matrix\_requests);

determine\_deadlock(num\_process, resource\_types, num\_resources, matrix\_allocation, matrix\_requests);

Task 9

Scheduling Algorithms

Summary Report: CPU Scheduling Algorithms Implementation

Introduction This document provides a detailed analysis of the CPU scheduling code, which implements three different scheduling algorithms: First Come First Serve (FCFS), Round Robin (RR), and Shortest Job First (SJF). Each section breaks down the code into functional blocks, explaining their roles and execution flow.

Class Definition: Process Management

Class Name: Proccess  
Purpose: The Proccess class models a process in the system, containing attributes relevant to scheduling.

Attributes:

* id: Unique identifier for the process.
* arrival\_time: Time at which the process arrives.
* burst\_time: Total execution time required.
* waiting\_time: Time spent waiting in the queue.
* turnaround\_time: Total time from arrival to completion.
* completion\_time: When the process finishes execution.
* remaining\_time: Remaining burst time (for RR and SJF calculations).

Constructor:

* Initializes attributes and ensures unique process IDs.

First Come First Serve Algorithm

Concept:

* The simplest CPU scheduling algorithm.
* Processes are executed in order of their arrival time.

Implementation Steps:

1. Sort processes based on arrival time.
2. Iterate through processes and update completion, turnaround, and waiting times.
3. Compute averages and display results.

Round Robin Algorithm

Concept:

* We keep iterating over the queue till the queue is empty
* If a process does not complete within the time quantam, it is placed back in the queue.

Implementation Steps:

1. Initialize a queue with all processes.
2. Process execution in cycles until all processes complete.
3. Decrement remaining time based on time quantum.
4. Requeue unfinished processes.
5. Calculate turnaround and waiting times.

Hurdles:

1. I thought about making the program have a loop that increase the same amount as the time quantum and then check remaining time of the process and compare it to the time quantum, but then I had no iterator for the Processes so I would have make another loop for it.
2. Also Thought about making a vector of pairs <Process,int> in which I store the object and the remaining time, but then I would have had to change the remaining time in the vector and in the object attributes and the reordering would have been a nightmare .

Shortest Job First Algorithm

Concept:

* The process with the shortest burst time is executed first.
* Non-preemptive: Once execution starts, the process runs to completion.
* More efficient than FCFS but may cause starvation of longer jobs.

Implementation Steps:

1. Find shortest available process at the current time.
2. Execute the process and update completion, turnaround, and waiting times.
3. Repeat until all processes finish.

Hurdles:

1. I initially thought about looping for a set time (100) and checking if a new process arrived, if there is a new process we sort the vector<pair<Process, int>> that holds the object and remaining\_time again and see which process to run now, but looks inefficient and the sorting on every step would increase the time complexity.
2. I thought about making a global vector that would store the arrival times and using it to determine when each process should run. But again that would separate arrival times from the object data, making it harder to track which process corresponds to which arrival time.

Conclusion This program demonstrates three different CPU scheduling techniques:

* FCFS is simple but may lead to high waiting times.
* Round Robin ensures fairness but increases overhead.
* SJF minimizes turnaround time but may starve longer processes.

Selecting the right algorithm depends on system requirements and workload characteristics.