bash

### 1. Factorial (Using Recursion)

- This program calculates the factorial of a given number using recursion.

- The function `factorial()` calls itself until it reaches the base case where the input is 1 or less, returning 1.

- It then multiplies each number in the sequence back up, giving the factorial result.

- For example, if you enter `5`, it calculates `5 \* 4 \* 3 \* 2 \* 1 = 120`.

### 2. Palindrome (Number and String)

- This program checks if a given input (number or string) is a palindrome.

- It reverses the input and compares it to the original.

- If the reversed input matches the original, it’s a palindrome; otherwise, it’s not.

- For example, if you enter `121` or `madam`, it identifies them as palindromes.

### 3. Bubble Sort

- This program sorts a list of numbers in ascending order using the bubble sort algorithm.

- It compares each pair of adjacent elements and swaps them if they’re in the wrong order.

- This process repeats until the entire list is sorted.

- For example, if you enter `3 1 4 1`, it sorts to `1 1 3 4`.

### 4. Count Occurrences of a Substring

- This program counts how many times a substring appears within a main string.

- It uses the `grep` command to find occurrences and `wc -l` to count them.

- For example, if you enter the main string "hello world" and the substring "o," it counts `2`.

### 5. Sum of Digits

- This program calculates the sum of the digits of a given number.

- It divides the number by 10 in a loop, adding each remainder (last digit) to a sum.

- For example, if you enter `123`, it calculates `1 + 2 + 3 = 6`.

### 6. Reverse Digits

- This program reverses the digits of a given number.

- It extracts the last digit of the number repeatedly, forming a reversed version.

- For example, if you enter `1234`, it outputs `4321`.

### 7. Armstrong Number

- This program checks if a given number is an Armstrong number.

- An Armstrong number is equal to the sum of its digits raised to the power of the number of digits.

- For example, `153` is an Armstrong number because `1^3 + 5^3 + 3^3 = 153`.

### 8. Array Operations (Insert, Delete, Search, Display)

- This program performs basic operations on an array.

- \*\*Insert:\*\* Adds an element to the array.

- \*\*Delete:\*\* Removes a specified element.

- \*\*Search:\*\* Checks if an element is in the array.

- \*\*Display:\*\* Shows the array.

- For example, you can add `5` to the array `[1, 2, 3]`, delete `2`, search for `3`, or display the final array.

### 9. Factorial (Without Recursion)

- This program calculates the factorial of a given number using a loop instead of recursion.

- It multiplies each number in sequence from 1 up to the input.

- For example, if you enter `4`, it calculates `1 \* 2 \* 3 \* 4 = 24`.

### 10. Palindrome String

- This program checks if a given string is a palindrome.

- It works like the palindrome number and string check but is specific to strings.

- For example, if you enter "radar," it confirms that it is a palindrome.

Here’s a simple explanation of how each of these two programs works:

### Program 2: Demonstrating Use of Different System Calls

This program demonstrates several common system calls in C for file handling, process creation, and working with system time.

#### Steps and System Calls Used:

1. \*\*Opening a File (`open`)\*\*:

- The program opens (or creates) a file named `example.txt` in write mode.

- `open` creates the file if it doesn’t exist and prepares it for writing.

2. \*\*Writing to the File (`write`)\*\*:

- It writes a message to `example.txt`.

- Only the first 50 characters of the string are written for demonstration.

3. \*\*Creating a Child Process (`fork`)\*\*:

- `fork` is used to create a new child process.

- The child process and the parent process will execute different code based on the `pid` value.

4. \*\*In the Child Process\*\*:

- The child process opens `example.txt` in read-only mode.

- It reads 50 characters into a buffer and prints the contents to the console.

- Finally, it closes the file and exits.

5. \*\*In the Parent Process\*\*:

- The parent process waits for the child process to finish using `wait`.

- It displays a message that the child process has finished.

6. \*\*Getting the System Time (`time`)\*\*:

- The program retrieves and prints the current system time.

The purpose of this program is to demonstrate various system calls like `open`, `write`, `fork`, `wait`, `read`, and `time`, and how they interact with files, processes, and system resources.

---

### Program 3: Multithreading for Matrix Operations using Pthreads

This program performs matrix addition and multiplication using multiple threads with the Pthreads library in C.

#### Steps and Multithreading Concepts Used:

1. \*\*Matrix Initialization\*\*:

- The program first initializes two 3x3 matrices, `A` and `B`, with user-provided values.

2. \*\*Matrix Addition Using Threads\*\*:

- A separate thread is created for each element in the result matrix `result\_add`.

- Each thread calculates one element in the addition result by adding corresponding elements from matrices `A` and `B`.

- \*\*Thread Creation and Joining\*\*:

- For each element, the program creates a thread using `pthread\_create`.

- It waits for each thread to complete using `pthread\_join`.

3. \*\*Displaying Addition Result\*\*:

- Once all addition threads finish, the program displays the resulting matrix `result\_add`.

4. \*\*Matrix Multiplication Using Threads\*\*:

- Similar to addition, the program creates a separate thread for each element in the multiplication result matrix `result\_mul`.

- Each thread calculates one element in the multiplication result by performing the matrix multiplication formula for that element.

- \*\*Thread Creation and Joining\*\*:

- For each element, a new thread is created and then joined to wait for completion.

5. \*\*Displaying Multiplication Result\*\*:

- Once all multiplication threads finish, the program displays the `result\_mul` matrix.

This program is an example of using threads to perform matrix operations in parallel. Each thread handles a small, independent part of the task, which can improve performance for large matrices.

Here’s an explanation of how each of these classic multi-threading synchronization problems works:

### Program 4: Reader-Writer Problem Using Threads and Mutexes

This program implements a solution to the \*\*Reader-Writer Problem\*\*, where multiple readers can access shared data simultaneously, but writers must have exclusive access.

#### Steps and Key Concepts:

1. \*\*Shared Resources and Counters\*\*:

- `shared\_data` is the data being read and written.

- `read\_count` tracks the number of readers currently accessing the data.

2. \*\*Mutexes\*\*:

- `read\_write\_lock` is a mutex that ensures exclusive access for writers when needed.

- `count\_lock` is used to protect `read\_count` so that only one thread can modify it at a time.

3. \*\*Reader Function\*\*:

- Each reader thread increments `read\_count` when it starts reading.

- If it’s the first reader, it locks `read\_write\_lock` to prevent writers from accessing `shared\_data`.

- After reading, it decrements `read\_count`, and if it’s the last reader, it unlocks `read\_write\_lock`, allowing writers to proceed.

4. \*\*Writer Function\*\*:

- Each writer thread locks `read\_write\_lock` before modifying `shared\_data`.

- This ensures that no other readers or writers can access `shared\_data` while a writer is modifying it.

- After writing, it unlocks `read\_write\_lock`, allowing readers or other writers to access `shared\_data`.

5. \*\*Main Function\*\*:

- Initializes mutexes.

- Creates threads for readers and writers.

- Waits for all threads to finish using `pthread\_join`.

- Destroys mutexes when done.

The program simulates the reader-writer problem where readers can read simultaneously but writers need exclusive access, ensuring data consistency and avoiding race conditions.

---

### Program 5: Producer-Consumer Problem Using Threads and Mutexes

This program solves the \*\*Producer-Consumer Problem\*\* using threads, mutexes, and condition variables. It models a scenario where producers generate items and add them to a shared buffer, and consumers remove items from this buffer.

#### Steps and Key Concepts:

1. \*\*Shared Buffer and Counters\*\*:

- `buffer` is a shared array used as a fixed-size circular buffer.

- `count` keeps track of the current number of items in the buffer.

2. \*\*Mutex and Condition Variables\*\*:

- `mutex` is used to ensure that only one thread (either a producer or a consumer) modifies `buffer` at a time.

- `not\_empty` is a condition variable that signals when there’s at least one item in the buffer, allowing consumers to consume.

- `not\_full` signals when there’s space available in the buffer, allowing producers to produce more items.

3. \*\*Producer Function\*\*:

- Each producer thread generates a random item.

- It locks `mutex` to access the buffer.

- If the buffer is full, it waits on `not\_full`, releasing the mutex temporarily until it’s signaled that space is available.

- Once there’s space, it adds the item to `buffer`, increments `count`, and signals `not\_empty` to wake up a waiting consumer.

- Finally, it unlocks the mutex.

4. \*\*Consumer Function\*\*:

- Each consumer thread locks `mutex` and waits if the buffer is empty (using `not\_empty`).

- Once there’s at least one item, it removes the item from `buffer`, decreases `count`, and signals `not\_full` to allow producers to add more items.

- It then unlocks the mutex and processes the item.

5. \*\*Main Function\*\*:

- Initializes threads for one producer and one consumer.

- Waits for both threads to finish with `pthread\_join`.

- Cleans up by destroying the mutex and condition variables.

This program ensures synchronization between the producer and consumer, avoiding overfilling the buffer or consuming from an empty buffer by using condition variables to coordinate production and consumption.

Here's a simple breakdown of each program and its functionality:

###6. Reader-Writer Problem (Using Threads and Semaphores)

The \*\*Reader-Writer Problem\*\* is about managing access to a shared resource where multiple "reader" threads can read simultaneously, but only one "writer" thread can modify it at a time.

- \*\*Semaphores\*\* are used here to manage access:

- `resourceAccess`: Ensures only one writer or multiple readers access the shared data at a time.

- `readCountAccess`: Controls access to `readCount`, which tracks the number of active readers.

- \*\*Reader Function\*\*:

- Each reader thread increments `readCount` before reading and locks the resource only if it’s the first reader.

- After reading, it decrements `readCount` and unlocks the resource if it’s the last reader.

- \*\*Writer Function\*\*:

- Each writer waits for exclusive access to `resourceAccess`, writes the data, and then releases `resourceAccess`.

### 7. Producer-Consumer Problem (Using Threads and Semaphores)

The \*\*Producer-Consumer Problem\*\* involves two types of threads: producers that add items to a shared buffer and consumers that remove items. The buffer has a limited capacity, so producers wait if it’s full, and consumers wait if it’s empty.

- \*\*Semaphores Used\*\*:

- `empty`: Tracks available slots in the buffer.

- `full`: Tracks filled slots in the buffer.

- `mutex`: Ensures only one thread modifies the buffer at a time.

- \*\*Producer Function\*\*:

- Each producer waits for an empty slot, acquires the lock, adds an item to the buffer, releases the lock, and signals that a slot is now full.

- \*\*Consumer Function\*\*:

- Each consumer waits for a filled slot, acquires the lock, removes an item, releases the lock, and signals that a slot is now empty.

### 8. Dining Philosopher Problem

In the \*\*Dining Philosopher Problem\*\*, philosophers alternately think and eat. They need two forks (shared resources) to eat, and they must avoid deadlock (where all philosophers are waiting forever for forks).

- \*\*Semaphores\*\* are used to represent each fork.

- Each philosopher picks up the two forks (by acquiring the corresponding semaphores) before eating and releases them after.

### 9. First Come First Serve (FCFS) Scheduling

The \*\*FCFS Scheduling Algorithm\*\* handles processes in the order they arrive. Each process runs to completion before the next one starts.

- \*\*Variables\*\*:

- `arrivalTime`: When the process arrives.

- `burstTime`: How long it takes to complete.

- `finishTime`, `turnaroundTime`, and `waitingTime` are calculated based on the order of arrival.

- \*\*Functionality\*\*:

- For each process, `finishTime` is set to the time it completes.

- `turnaroundTime` is the total time from arrival to completion.

- `waitingTime` is the time a process spends waiting for its turn to run.

### 10. Shortest Job First (SJF) Scheduling (Non-Preemptive)

The \*\*SJF Scheduling Algorithm\*\* prioritizes shorter jobs, so the shortest job gets scheduled next. In this non-preemptive version, once a process starts, it runs to completion.

- \*\*Sort by Arrival and Burst Time\*\*:

- Processes are sorted by arrival time and burst time, so the scheduler selects the shortest available job.

- \*\*Completion Calculations\*\*:

- `completionTime` is calculated by adding the `burstTime` to the `time`.

- `waitingTime` is the difference between `turnaroundTime` and `burstTime`.

This program also prints the completion, turnaround, and waiting times for each process based on the SJF non-preemptive strategy.

Here's a breakdown of each program you provided.

---

### 12. Round Robin Scheduling Program

This program calculates the completion time, turnaround time, and waiting time for each process using the \*\*Round Robin scheduling algorithm\*\*. Round Robin is a CPU scheduling method that allocates a fixed time quantum to each process in a cyclic order.

\*\*Key Concepts\*\*:

- \*\*Process\*\*: A unit with an ID (`pid`), burst time (`bt`), arrival time (`at`), completion time (`ct`), waiting time (`wt`), turnaround time (`tat`), and remaining burst time (`rem\_bt`).

- \*\*Time Quantum\*\*: The fixed amount of time each process can run in one go before moving to the next.

\*\*Steps in Code\*\*:

1. \*\*Input\*\*: Number of processes, time quantum, and details (arrival and burst time) of each process.

2. \*\*Round Robin Scheduling Logic\*\*:

- Initialize the current time and track remaining processes.

- For each process with remaining burst time:

- If it can finish within the quantum, update time and calculate its completion, turnaround, and waiting times.

- If it can't finish in the quantum, decrease its remaining burst time and update the time.

- Move to the next process in a circular manner.

3. \*\*Output\*\*: Print the computed arrival time, burst time, completion time, turnaround time, and waiting time for each process.

---

### 13. Banker's Algorithm for Deadlock Avoidance

This program checks if a system is in a \*\*safe state\*\* using the Banker's algorithm, which is designed to avoid deadlocks by ensuring resources are allocated safely based on maximum demand and current availability.

\*\*Key Concepts\*\*:

- \*\*Available Resources\*\*: The number of instances available for each resource.

- \*\*Maximum Demand\*\*: The maximum demand each process can have for each resource.

- \*\*Allocation\*\*: The resources currently allocated to each process.

- \*\*Need\*\*: Calculated as `Maximum - Allocation`, indicating the remaining resources a process might still request.

\*\*Steps in Code\*\*:

1. \*\*Input\*\*: The number of processes, number of resources, available resources, maximum demand of each process, and allocated resources for each process.

2. \*\*Calculate Need Matrix\*\*: Calculate the need for each process based on maximum demand and allocated resources.

3. \*\*Safety Algorithm\*\*:

- Start with the `work` array initialized to available resources.

- Find a process that can complete with the current `work` resources (i.e., its `Need <= Work`).

- If found, "complete" the process by releasing its allocated resources back to `work`, mark the process as finished, and add it to the safe sequence.

- If no such process is found and not all processes are finished, the system is unsafe.

4. \*\*Output\*\*: If the system is safe, display the safe sequence; otherwise, indicate an unsafe state.

---

### Example Inputs and Outputs:

\*\*Round Robin Program\*\*:

- \*\*Input\*\*: 4 processes, arrival and burst times like 0 5, 1 4, etc., and time quantum (e.g., 4).

- \*\*Output\*\*: Completion, turnaround, and waiting times for each process based on Round Robin scheduling.

\*\*Banker's Algorithm\*\*:

- \*\*Input\*\*: 5 processes, 3 resources, availability (3,3,2), max demand and allocation matrices for each process.

- \*\*Output\*\*: Safe state and sequence or indicate an unsafe state.

These programs cover key scheduling and deadlock avoidance algorithms commonly used in operating systems.

Let's break down each of these programs to understand them in simpler terms.

---

### 14. Deadlock Detection Program

This program uses a \*\*Wait-For Graph\*\* to detect a deadlock in a system with multiple processes. A wait-for graph is a representation where each process is a node, and a directed edge from process `i` to process `j` indicates that process `i` is waiting for a resource held by process `j`.

#### Key Concepts:

- \*\*Cycle in Graph\*\*: A deadlock occurs if there's a cycle in the wait-for graph, meaning processes are in a circular wait, each waiting for a resource held by the next process in the cycle.

#### Steps in Code:

1. \*\*Input\*\*: The number of processes and edges (i.e., waiting relationships between processes).

2. \*\*Initialize Graph\*\*: Set all values in the `wait\_for\_graph` matrix to 0.

3. \*\*Add Edges\*\*: Add an edge between processes `i` and `j` if `i` is waiting for `j`.

4. \*\*Detect Cycle\*\*:

- For each process, use \*\*Depth-First Search (DFS)\*\* to detect cycles.

- If a process is revisited in the DFS recursion (i.e., it’s in the recursion stack), then a cycle is detected, meaning there’s a deadlock.

5. \*\*Output\*\*: Indicate if a deadlock is detected.

#### Example Input:

- \*\*Processes\*\*: 4, \*\*Edges\*\*: 4

- Edges: (0 -> 1), (1 -> 2), (2 -> 3), (3 -> 1)

\*\*Output\*\*: "Deadlock detected" because there’s a cycle.

---

### 15. FIFO Page Replacement Program

This program simulates the \*\*First-In-First-Out (FIFO) page replacement algorithm\*\*. This algorithm is used in memory management to decide which page to remove when a new page needs to be loaded.

#### Key Concepts:

- \*\*FIFO Queue\*\*: Pages are managed like a queue; the first page loaded is the first to be replaced.

- \*\*Page Fault\*\*: Occurs when a requested page is not in memory.

#### Steps in Code:

1. \*\*Input\*\*: Reference string (sequence of page numbers requested) and the number of page frames (slots).

2. \*\*Initialize Memory\*\*: Set all frames (memory slots) to -1 (empty).

3. \*\*Process Each Page\*\*:

- If the page is already in memory, it’s a \*\*page hit\*\*.

- If the page is not in memory, it’s a \*\*page fault\*\*:

- Replace the page at the front of the FIFO queue.

- Move the front pointer forward in the circular queue.

- Display the memory state after each page request.

4. \*\*Output\*\*: Print the number of page faults.

#### Example Input:

- \*\*Reference String\*\*: 7, 0, 1, 2, 0, 3, 0, 4, 2, 3, 0, 3, 4

- \*\*Page Frames\*\*: 3

\*\*Output\*\*: Number of page faults and memory contents after each access.

---

### 16. LRU Page Replacement Program

This program simulates the \*\*Least Recently Used (LRU) page replacement algorithm\*\*, another memory management strategy where the page that hasn’t been used for the longest time is replaced.

#### Key Concepts:

- \*\*LRU Replacement\*\*: Tracks how recently each page in memory was accessed. When a page fault occurs, the page that was least recently used is replaced.

- \*\*Page Fault\*\*: Occurs when a requested page is not found in memory.

#### Steps in Code:

1. \*\*Input\*\*: Reference string (sequence of page numbers requested) and the frame capacity (number of slots).

2. \*\*Initialize Memory\*\*: Set all frames to -1 (empty).

3. \*\*Process Each Page\*\*:

- If the page is already in memory, it’s a \*\*page hit\*\*, so just update the last used time.

- If the page is not in memory, it’s a \*\*page fault\*\*:

- Check if there’s an empty slot in memory.

- If there’s no empty slot, find the \*\*Least Recently Used (LRU)\*\* page using the `find\_LRU` function and replace it with the new page.

- Display the memory state after each page request.

4. \*\*Output\*\*: Print the total number of page faults.

#### Example Input:

- \*\*Reference String\*\*: 7, 0, 1, 2, 0, 3, 0, 4, 2, 3, 0, 3, 4

- \*\*Frame Capacity\*\*: 3

\*\*Output\*\*: Number of page faults and memory contents after each access.

---

These programs illustrate different methods for deadlock detection and memory management in operating systems. The deadlock detection program identifies if processes are in a state of circular waiting. The FIFO and LRU page replacement algorithms determine which memory pages to keep or replace based on different policies, optimizing for fewer page faults.

Here are explanations for each of the given programs:

---

### 17. Optimal Page Replacement Algorithm

\*\*Purpose\*\*:

This program simulates the Optimal Page Replacement algorithm, calculating the number of page faults. The Optimal algorithm replaces the page that won’t be needed for the longest period in the future.

\*\*Explanation\*\*:

- \*\*`find\_optimal()`\*\* function:

- Takes in the current pages in memory and the upcoming pages.

- For each page in memory, it checks how far in the future it will next be used.

- Returns the position of the page that won’t be used for the longest time, or that won’t be used again.

- \*\*`optimal\_page\_replacement()`\*\* function:

- Iterates through each page in the sequence.

- If the page is not in memory, it counts it as a page fault.

- If there’s an empty slot, it adds the page to memory; if not, it replaces the least-used page found by `find\_optimal()`.

---

### 18. FCFS Disk Scheduling

\*\*Purpose\*\*:

This program simulates the First-Come, First-Serve (FCFS) disk scheduling algorithm, calculating the total seek time and presenting results in a table format.

\*\*Explanation\*\*:

- \*\*`fcfs\_disk\_scheduling()`\*\* function:

- Moves the disk head in the order of the requests, calculating the distance (seek time) it has to move each time.

- Adds up the seek times to get the total.

- \*\*`print\_table()`\*\* function:

- Formats and displays each request, the position it seeks, and the seek time for that request.

- Finally, it displays the total seek time.

---

### 19. SSTF Disk Scheduling

\*\*Purpose\*\*:

This program simulates the Shortest Seek Time First (SSTF) disk scheduling algorithm, which selects the request closest to the current head position to minimize seek time.

\*\*Explanation\*\*:

- \*\*`find\_closest\_request()`\*\* function:

- Finds the unprocessed request closest to the current head position.

- \*\*`sstf\_disk\_scheduling()`\*\* function:

- Selects the request with the shortest seek time (closest) at each step, moving the head to that position and marking it as processed.

- Accumulates the total seek time.

- \*\*`print\_table()`\*\* function:

- Prints each request in the order they’re served along with their individual seek times, then displays the total seek time.

---

These programs help demonstrate various algorithms for managing memory and disk scheduling, balancing efficiency by minimizing page faults and seek time.

Here are simplified explanations for each of the disk scheduling programs for SCAN and C-SCAN:

---

### 20. SCAN Disk Scheduling (Also Known as the Elevator Algorithm)

\*\*Purpose\*\*:

This program simulates the SCAN disk scheduling algorithm. SCAN moves the disk arm towards the end in one direction, services all requests in its path, and then reverses direction once it reaches the end.

\*\*Explanation\*\*:

- \*\*`sort()` function\*\*:

- Sorts the disk requests array in ascending order to make it easier to process requests in a linear path.

- \*\*`scan\_disk\_scheduling()` function\*\*:

- \*\*Input Parameters\*\*:

- `requests[]` is the list of disk requests.

- `head` is the starting position of the disk head.

- `disk\_size` is the total size of the disk.

- `direction` indicates the initial direction (0 for left, 1 for right).

- \*\*Steps\*\*:

- It first sorts the requests.

- It finds the closest request in the direction the head is moving (either left or right).

- \*\*If moving right\*\*:

- It services all requests to the right, moving toward the end of the disk.

- Once it reaches the end, it reverses direction, services requests on the left, and moves toward the start.

- \*\*If moving left\*\*:

- It services all requests to the left, moving toward the start of the disk.

- Once it reaches the start, it reverses direction, servicing requests to the right.

- It accumulates the total seek time during this process.

- \*\*`print\_table()` function\*\*:

- Displays each request's order and seek time in a tabular format, then prints the total seek time.

---

### 21. C-SCAN Disk Scheduling (Circular SCAN)

\*\*Purpose\*\*:

The program simulates the C-SCAN (Circular SCAN) disk scheduling algorithm. C-SCAN moves the disk head in one direction to the end, then jumps back to the beginning of the disk without servicing requests on the reverse path. This provides a more uniform wait time.

\*\*Explanation\*\*:

- \*\*`compare()` function\*\*:

- Used for sorting the requests in ascending order.

- \*\*`calculateSeekTime()` function\*\*:

- \*\*Input Parameters\*\*:

- `requests[]` is the list of disk requests.

- `head` is the initial head position.

- `disk\_size` is the total size of the disk.

- \*\*Steps\*\*:

- It separates requests into two arrays: one for requests to the left of the head and one for requests to the right.

- Sorts both the left and right request arrays.

- \*\*Servicing Right Side First\*\*:

- Services all requests to the right of the head, moving toward the end of the disk.

- When it reaches the end, it jumps back to the start and services requests from the beginning to the initial head position’s left.

- Adds up the seek distances at each step to calculate the total seek time.

- \*\*Output\*\*:

- Displays the request queue, head movement order, total seek time, and average seek time in a formatted output.

---

Both SCAN and C-SCAN optimize seek time compared to FCFS by reducing the movement of the disk head, with SCAN servicing requests in both directions and C-SCAN offering more uniform wait times by only servicing in one direction.

Here’s a simple explanation of the given programs:

### 24. \*\*Copy File Contents (Character by Character)\*\*

This program copies the content of one file to another, character by character.

\*\*Steps:\*\*

- The `copyFileContent` function takes two file names: a source and a destination.

- It opens the source file in read mode and the destination file in write mode.

- Then, it reads each character from the source file using `fgetc` and writes it to the destination file using `fputc`.

- This continues until the entire content is copied.

- It closes both files after finishing.

\*\*Key operations:\*\*

- `fgetc(source)`: Reads one character from the source file.

- `fputc(ch, destination)`: Writes that character to the destination file.

### 25. \*\*Load ALP Program to Memory\*\*

This program simulates loading an Assembly Language Program (ALP) into a "main memory" array.

\*\*Steps:\*\*

- The program asks the user for the path to an ALP input file.

- It opens this file and reads each line (instruction) into a memory array, simulating a simple main memory.

- For each line, it removes any newline characters.

- It checks if the memory is full (using `MAX\_MEMORY`) and stops if the limit is reached.

- Finally, it displays the contents of the memory.

\*\*Key operations:\*\*

- `fgets()`: Reads one line from the file.

- `strcspn()`: Removes the newline character at the end of the line.

### 26. \*\*Check Opcode Error and Raise Interrupt\*\*

This program checks whether the opcodes in a job file (ALP file) are valid.

\*\*Steps:\*\*

- The program reads the file line by line.

- It extracts the opcode (the first word of the instruction).

- It checks whether this opcode is in the list of valid opcodes.

- If it is invalid, it raises an interrupt by printing an error message (simulating an interrupt).

- It continues checking the next lines even if an error is found.

\*\*Key operations:\*\*

- `sscanf()`: Extracts the opcode from the instruction.

- `strcmp()`: Compares the extracted opcode with valid opcodes.

- If an invalid opcode is found, it raises an interrupt by printing an error message.

### 27. \*\*Check Operand Error and Raise Interrupt\*\*

This program checks whether the number of operands in an ALP job file matches what is expected for the corresponding opcode.

\*\*Steps:\*\*

- The program reads the job file line by line.

- It extracts the opcode and operands from each line.

- It checks the expected number of operands for the opcode by referring to a predefined list.

- If the number of operands does not match the expected number, it raises an interrupt (simulating an error).

- It continues checking the rest of the lines even after finding errors.

\*\*Key operations:\*\*

- `sscanf()`: Extracts the opcode and operands from the instruction.

- `getExpectedOperands()`: Checks the expected number of operands for each opcode.

- If the number of operands does not match, it raises an interrupt by printing an error message.

### Key Concepts Used:

- \*\*File Handling\*\*: Opening files for reading/writing (`fopen()`), reading lines (`fgets()`), reading characters (`fgetc()`), and writing characters (`fputc()`).

- \*\*Error Handling\*\*: The programs handle errors like file not opening (`fopen()` fails) and incorrect opcode/operand issues.

- \*\*String Manipulation\*\*: Extracting parts of a string (like opcodes or operands) and comparing them (`strcmp()`, `sscanf()`).

These programs simulate the process of working with assembly-like code, checking for errors (invalid opcodes or incorrect operands) in a simple way.