Disk Management

Reg. No: CB.SC.U4CSE23161

Name: Kavin Karthic M

Introduction :

Tasks done :Disc Scheduling(SCAN,SSTF),FILE SCHEDULING(Continous, Linked,Indexex Allocations),RAID 0. Its been implemented with raspberry pi pico with freertos library and a sd card module connected with it.

1.SCAN

SCAN is a disk scheduling algorithm that moves the read/write head back and forth across the disk (like an elevator) to handle requests efficiently.The SCAN algorithm, also known as the elevator algorithm, moves the disk head across the entire disk surface, servicing requests in one direction (e.g., from the outermost to innermost track) until it reaches the end, then reverses direction. This ensures fairness but may delay requests at the extremes

2.SSTF

The Shortest Seek Time First (SSTF) algorithm, on the other hand, selects the request closest to the current head position, minimizing seek time for each step. While SSTF is efficient for quick responses, it can lead to starvation for requests far from the head if closer ones keep arriving. Both aim to optimize disk performance but balance efficiency and fairness differently.

3.FILE ALLOCATION:

File scheduling in an operating system typically refers to the strategies or methods used to manage and organize how files are stored, accessed, and retrieved on a storage device, such as a hard disk.File allocation in an OS defines how the system assigns disk space to files, ensuring efficient storage, retrieval, and management.These techniques aim to optimize access speed, minimize wasted space, and handle file growth effectively, tailored to the system's requirements

CONTINOUS ALLOCATION:

In contiguous allocation, a file is stored in a single, uninterrupted sequence of disk blocks. The operating system assigns a starting block and a length to the file, making it easy to locate and access sequentially since all blocks are adjacent. This method offers fast read and write performance due to minimal seek time. However, it struggles with external fragmentation—gaps between files that can’t be used efficiently—and resizing files is challenging, as there may not be enough consecutive free blocks available to expand.

INDEXED:

Indexed allocation uses a separate index block for each file, which contains pointers to all the file's data blocks. The directory points to this index block, allowing the file's data to be stored in non-contiguous blocks across the disk. This supports efficient random access, as the system can jump directly to any block via the index, and files can grow easily by adding pointers. The trade-off is the overhead of storing and managing the index block, which can waste space for small files, and accessing the index adds a slight delay compared to contiguous allocation.

LINKED:

Linked allocation stores a file as a chain of disk blocks, where each block contains a pointer to the next block in the sequence. The directory only needs to track the first block, and the file can grow dynamically by linking additional blocks anywhere on the disk, avoiding fragmentation issues. However, this method slows down sequential access because blocks may be scattered, increasing seek time, and random access is inefficient since the system must follow the chain. If a pointer is lost or corrupted, the rest of the file becomes inaccessible.

CODE EXPLANATION:

SCAN:

General Structure and Libraries

- Uses Arduino environment to simulate disk scheduling.
- Includes FreeRTOS to handle multitasking.
- Uses semaphores to manage synchronized access to shared resources (like the Serial monitor).

Global Variables and Definitions

- `xSerialMutex`: Semaphore used to control serial printing between tasks.
- `xSCANTaskHandle`: Handle for the SCAN scheduling task.
- `MAX_REQUESTS`: Maximum number of disk I/O requests allowed (set to 20).
- `DISK_SIZE`: Maximum size of the disk (cylinders range from 0 to 199).
- `requests[]`: Array to store the user-entered disk requests.
- `request_count`: Number of disk requests entered by the user.
- `head`: Current position of the disk arm (initial head position).
- `direction`: Direction of head movement; `1` means right (increasing cylinder numbers), `0` means left.

scan_task Function (Performs SCAN Scheduling)

- Displays initial information (requests, head position, direction) by taking a mutex for serial access.
- Adds disk boundaries (0 and 199) to the requests for the SCAN algorithm.
- Combines all requests and boundaries into a single array and sorts it in ascending order.
- Identifies the position of the head in the sorted array.
- Based on the direction:
    - If moving right, services all requests to the right of the head, then reverses to the left.
    - If moving left, services requests to the left, then reverses to the right.
- Calculates total seek operations and builds the seek sequence.
- Displays the final seek sequence, total seek operations, and average seek length.

- Deletes the task after completion using `vTaskDelete`.

read_input Function (Takes User Input)
- Prompts user to enter comma-separated disk requests through the Serial monitor.
- Reads and parses the input string to extract integers into the `requests[]` array.
- Prompts and reads the initial head position and direction of movement.
- Applies constraints to head and direction to ensure valid values.
- Displays a summary of the inputs using the serial monitor.

setup Function (Main Initialization)
- Initializes Serial communication.
- Waits for Serial to be ready.
- Displays introductory messages.
- Calls `read_input()` to get user data.
- Creates the serial mutex.
- Creates the FreeRTOS task `scan_task`.
- Starts the FreeRTOS scheduler to begin task execution.

loop Function
- Left empty because FreeRTOS handles all the scheduling and task execution.

SSTF:

Included Libraries
- `FreeRTOS.h`, `task.h`, `semphr.h`: Provide support for multitasking, task creation, and semaphores.
- `stdio.h`: Used for `printf` instead of Arduino `Serial`.

Global Variables
- `xSerialMutex`: A semaphore to control access to `printf`, ensuring only one task prints at a time.
- `xSSTFTaskHandle`: Handle for the SSTF disk scheduling task.

sstf_task Function
- Contains a hardcoded list of disk I/O requests.
- `head` is initialized to 53 (starting position of the disk head).
- Acquires the `xSerialMutex` semaphore to ensure safe and uninterrupted printing.
- Prints the initial head position as the start of the seek sequence.
- Initializes `total_seek_time` to 0 and a `completed[]` array to track serviced requests.
- For each request:
  - Finds the uncompleted request with the shortest seek time from the current head.
  - Updates the head to the nearest request.
  - Adds the seek time to `total_seek_time`.
  - Marks the request as completed.
  - Prints the head movement step.
- After serving all requests:
  - Prints the total seek time.
  - Releases the `xSerialMutex`.
  - Prints a completion message.
- Deletes the task using `vTaskDelete`.

main Function
- Initializes the serial mutex.

- Creates the SSTF task using `xTaskCreate`.
- Starts the FreeRTOS scheduler using `vTaskStartScheduler`.
- Enters an infinite loop (`while (1) {}`) — after this, FreeRTOS handles the scheduling and task execution.

FILE ALLOCATION:

1. Include Required Modules
   - Include FreeRTOS libraries for task management and synchronization.
   - Include standard input/output headers for logging messages.
   - Define the size of the simulated disk (e.g., 1000 blocks).

2. Simulate Disk with Arrays
   - Create an array to represent the **disk blocks**, initialized as free.
   - Create a **next-pointer array** for simulating linked allocation (like pointers in a linked list).
   - Create a **2D index table** for indexed allocation where each row stores the blocks allocated to a file.

3. Initialize the Disk
   - Set all disk blocks as **free** (e.g., with a value like `-1`).
   - Reset the `next` pointer and index table arrays.

4. Contiguous Allocation Process
   - Allocate a **range of continuous blocks** from a given start location.
   - Mark each block in that range as **allocated**.
   - Log or print the start block and how many blocks were allocated.

5. Linked Allocation Process
   - Take a **list of block numbers** that are scattered on the disk.
   - Mark each as allocated.
   - Link each block to the next by updating the `next` pointer array.
   - Print the sequence to show how the file is linked across the disk.

6. Indexed Allocation Process
   - Choose one block as the **index block**.
   - Allocate other blocks and store their references in the index block (2D array).
   - Mark all involved blocks as allocated.
   - Print the index block and the blocks it points to.

7. Run All Allocations Inside a FreeRTOS Task
   - Create a FreeRTOS task to simulate the file allocations.
   - Inside the task:
     - Initialize the disk.
     - Perform contiguous allocation.
     - Perform linked allocation.
     - Perform indexed allocation.
   - Use a mutex to ensure clean and safe logging if multiple tasks are involved.

8. Start FreeRTOS Scheduler
   - Create the disk allocation task.
   - Start the FreeRTOS scheduler to run your task.
   - The task completes and deletes itself after all allocations are done.

RAID 0:

Setup Phase

- Initializes serial communication for user interaction.
- Initializes the SD card module (usually on pin 4).
- Checks if `disk1` and `disk2` files exist; if not, creates them.

Data Input

- Prompts the user to enter a string of data to write.
- Captures the input string via serial monitor.

Striping Logic (RAID 0 Writing)

- Data is **split alternately** between two files:
    - Characters at even positions → `disk1`
    - Characters at odd positions → `disk2`
- Simulates how RAID 0 splits data blocks between two disks to improve speed.

Reading Back Data

- Opens both `disk1` and `disk2` for reading.
- Reconstructs the original string by reading:
    - One character from `disk1`, then one from `disk2`, and so on.
- Displays the reassembled string to the user.

Failure Simulation

- Simulates the failure of `disk2`.
- Attempts to read only from `disk1`.
- Shows partial reconstruction (every second character missing) to demonstrate **no fault tolerance in RAID 0**.

Conceptual Takeaway

- RAID 0 increases performance by splitting data (striping).
- **But it has no redundancy** — if one disk fails, all data is lost.
- The code visually demonstrates both the speed-up idea and the risk of data loss.

REAL WORLD USECASE:

Sensor Task: Periodically generates simulated sensor readings.
Log Task: Logs the readings and associated action ("Pump ON"/"OFF") to both a main log file and a temporary journal file.
The journal file acts as a buffer for crash recovery. If the device reboots, unsaved entries in the journal are transferred to the main file. It ensures fault tolerance and efficient data logging for environmental monitoring systems like smart farming or irrigation setups.

CODE EXPLANATION:

- SD.h and SPI.h are used for interfacing with an SD card via SPI protocol.

- FreeRTOS.h, task.h, and queue.h are used to support multitasking with FreeRTOS on Arduino.
- Arduino.h is included to access core Arduino functions.

It defines constants:

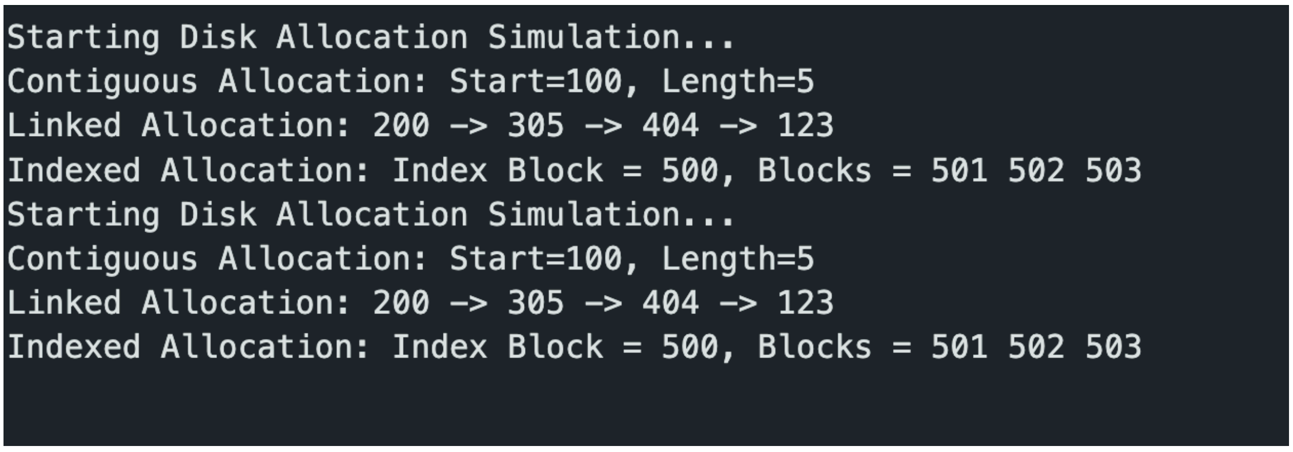
- A chip select (CS) pin for the SD card.
- File paths for a journal file (`/journal.txt`) and a main filesystem log file (`/fs.txt`).
- A maximum journal file size in bytes (`1000` bytes).

Two global `File` variables are declared for interacting with the SD card files.
A FreeRTOS queue is declared to allow communication between tasks. It is used to send SensorData structures between tasks.
A SensorData structure is defined to hold two values: temperature (as a float) and soil moisture (as an integer).
In the setup() function:

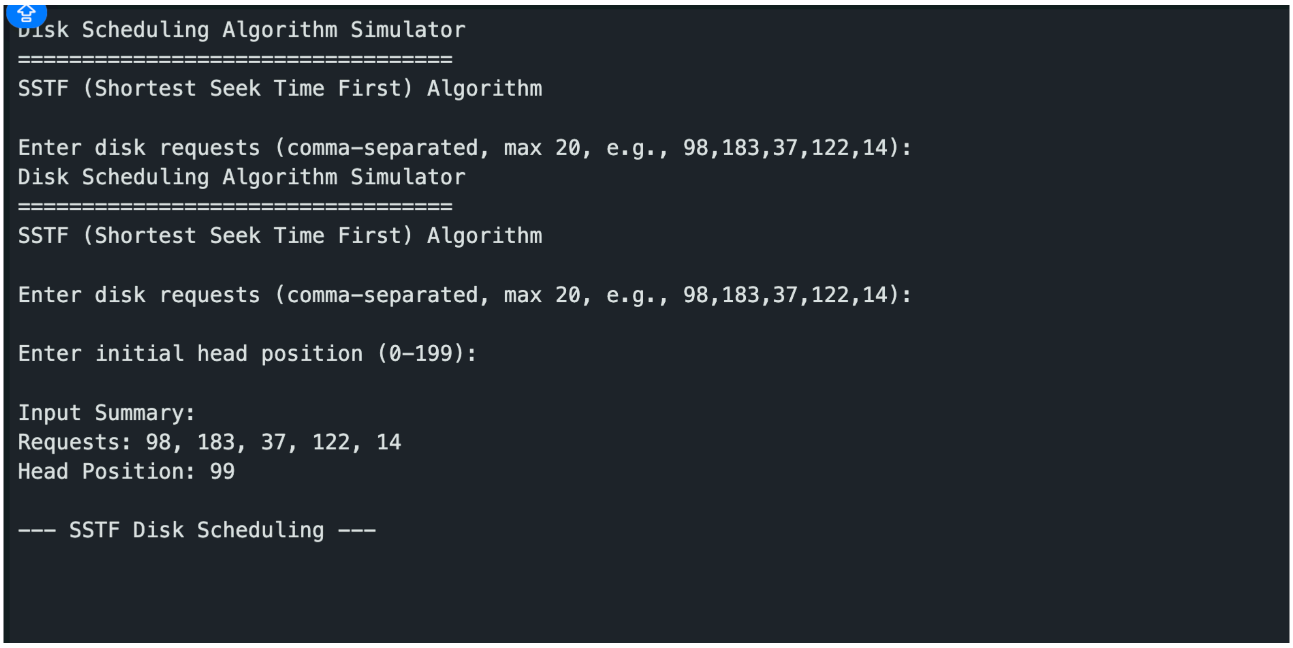
- Serial communication is initialized for debugging.
- The program checks if the SD card is accessible. If not, it halts.
- It then checks for a crash recovery scenario by calling recoverFromJournal(), which restores any unsaved data from the journal file to the main file.
- A FreeRTOS queue is created to hold sensor data.
- Two tasks are launched:
    - One for reading and sending sensor data (`sensorTask`).
    - One for logging that data to files (`logTask`).

The loop() function is intentionally left empty because FreeRTOS manages the task scheduling.
The sensorTask function runs in an infinite loop. It generates fake temperature and moisture readings, then sends them to the queue every 5 seconds.
The logTask function also runs in a loop. It receives data from the queue, formats a log entry string (including an action: "Pump ON" if moisture is low, otherwise "Pump OFF"), prints it to Serial, and writes it to both the journal and main log files.
Before writing, it checks if there's enough space left in the files. If storage is full, it skips the write operation and logs a warning.
The recoverFromJournal() function is responsible for crash recovery. If a journal file exists, it appends all entries from it to the main log file, then deletes the journal file. This ensures that no data is lost during a power failure or unexpected reset.
Finally, the getFileSize() function simply returns the size (in bytes) of a specified file on the SD

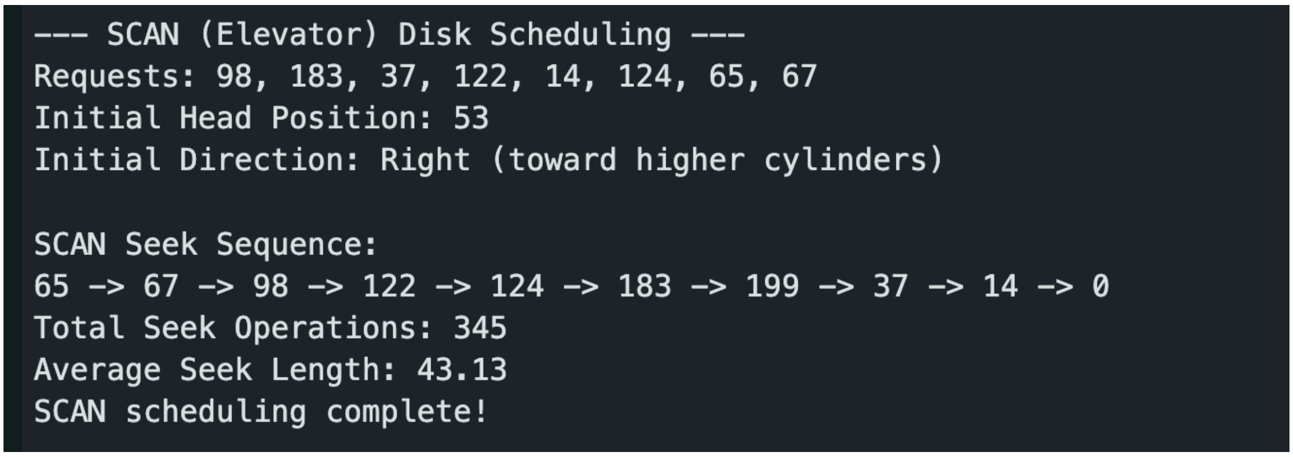
OUTPUTS:



```
Starting Disk Allocation Simulation...
Contiguous Allocation: Start=100, Length=5
Linked Allocation: 200 -> 305 -> 404 -> 123
Indexed Allocation: Index Block = 500, Blocks = 501 502 503
Starting Disk Allocation Simulation...
Contiguous Allocation: Start=100, Length=5
Linked Allocation: 200 -> 305 -> 404 -> 123
Indexed Allocation: Index Block = 500, Blocks = 501 502 503
```

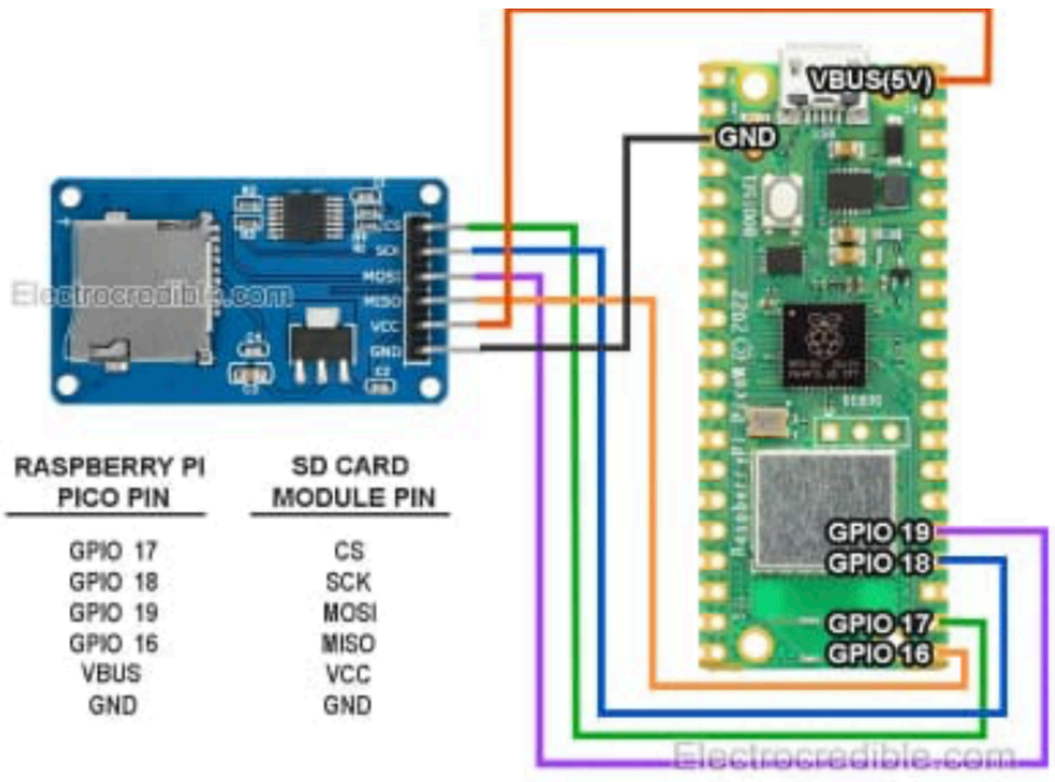


```
Disk Scheduling Algorithm Simulator
==================================
SSTF (Shortest Seek Time First) Algorithm

Enter disk requests (comma-separated, max 20, e.g., 98,183,37,122,14):
Disk Scheduling Algorithm Simulator
==================================
SSTF (Shortest Seek Time First) Algorithm

Enter disk requests (comma-separated, max 20, e.g., 98,183,37,122,14):

Enter initial head position (0-199):

Input Summary:
Requests: 98, 183, 37, 122, 14
Head Position: 99

--- SSTF Disk Scheduling ---
```



```
--- SCAN (Elevator) Disk Scheduling ---
Requests: 98, 183, 37, 122, 14, 124, 65, 67
Initial Head Position: 53
Initial Direction: Right (toward higher cylinders)

SCAN Seek Sequence:
65 -> 67 -> 98 -> 122 -> 124 -> 183 -> 199 -> 37 -> 14 -> 0
Total Seek Operations: 345
Average Seek Length: 43.13
SCAN scheduling complete!
```

Sd card module:



RASPBERRY PI PICO PIN	SD CARD MODULE PIN
GPIO 17	CS
GPIO 18	SCK
GPIO 19	MOSI
GPIO 16	MISO
VBUS	VCC
GND	GND