Chit no 1

This code implements the Shortest Job First (SJF) scheduling algorithm for processes. It accepts the number of processes and their burst times from the user, sorts them based on burst time, and calculates the waiting time and turnaround time for each process. It outputs the process ID, burst time, waiting time, and turnaround time in a table format, along with the average waiting time and turnaround time for all processes.

For the provided input example, it will print the respective times for each process and calculate the average waiting and turnaround times.

Chit no 2

This code implements the Priority Scheduling algorithm for process scheduling. Each process has an associated priority, and the processes are executed based on their priority (higher priority is executed first). The code calculates and displays the waiting time and turnaround time for each process, along with the average waiting and turnaround times.

1. The processes are sorted based on their priority (highest priority first).
2. Waiting time and turnaround time for each process are calculated.
3. The processes' IDs, burst time, waiting time, and turnaround time are displayed.
4. Finally, the average waiting time and average turnaround time are calculated and printed.

For the provided input, it will execute the processes in order of priority and compute the corresponding times.

Chit no 3

This code implements the Round Robin scheduling algorithm. It calculates and prints the waiting time and turnaround time for each process based on their burst times and a given time quantum. It also calculates the average waiting and turnaround times. The processes are executed in a cyclic manner, with each process getting executed for a fixed quantum of time until completion.

Chit no 4

This code implements the FIFO (First In, First Out) page replacement algorithm. It simulates loading pages into memory, keeping track of page faults. When a page is not in memory, it gets loaded, and if the memory is full, the oldest page is replaced. The code prints whether a page is loaded or already in memory and calculates the total number of page faults.

For the provided example (page references: [2, 3, 4, 2, 1, 3, 7, 5, 4, 3], memory capacity: 3), it tracks which pages are loaded and the total number of page faults.

Chit no 5

This code implements the LRU (Least Recently Used) page replacement algorithm. It loads pages into a cache, evicting the least recently used page when the cache exceeds its capacity. It tracks and prints page faults, indicating when pages are loaded or already in memory.

Chit no 6

This code implements the Optimal Page Replacement algorithm, where pages are loaded into memory, and when the memory is full, the page that will not be used for the longest time in the future is replaced. It tracks and prints the total number of page faults.

Chit no 7

This code implements the First-Fit memory allocation algorithm. It assigns processes to the first available block that can accommodate them and then reduces the block size accordingly. If a process can't be allocated to any block, it's marked as "Not Allocated." The code outputs the process allocation details, including which process is assigned to which block or if it isn't allocated.

Chit no 8

The code implements the **Best-Fit memory allocation** algorithm. It allocates each process to the smallest block that fits. If no block fits, the process isn't allocated. The result shows which block each process is allocated to or if it's not allocated.

Chit no 9

The code implements the **Worst-Fit memory allocation** algorithm. It assigns each process to the largest available block that fits. If no block fits, the process isn't allocated. The result shows which block each process is assigned to or if it's not allocated.

Chit no 10

The code implements the **Next-Fit memory allocation** algorithm. It assigns each process to the first block that fits, starting from the last allocated block and wrapping around if necessary. The result shows which block each process is assigned to or if it's not allocated.

Chit no 11

The code processes assembly code, generating symbol, literal tables, and intermediate code. It handles directives like `START`, `END`, and `LTORG`, and allocates addresses to labels and literals. The result is a set of tables showing symbols, literals, and their addresses.

Chit no 12

The code implements a two-pass assembler. **Pass 1** processes assembly instructions, generating a **symbol table**, **literal table**, and **intermediate code**. **Pass 2** translates these into **machine code** using opcode mappings and addresses from the symbol and literal tables.

Chit no 13

The code simulates a **macro processor** that identifies **macro definitions** and replaces **macro invocations** with their respective bodies in a source code. It processes the code in **Pass 1**, storing macros and replacing invocations accordingly.

Chit no 14

The code simulates a **macro processor** in two passes:

1. **Pass-I** stores macro definitions and replaces macro invocations in the source code with their respective bodies.
2. **Pass-II** generates the final output (intermediate code) from the expanded code.

It processes the source code, expands macros, and outputs the processed code.

Chit no 15

The code implements **First Come First Serve (FCFS) scheduling** to calculate the **waiting time** and **turnaround time** for processes. It then computes and prints the **average waiting time** and **average turnaround time** for all processes. The output shows the burst time, waiting time, and turnaround time for each process, along with the averages for the entire set of processes.

Chit no 16

The code you've provided is for an Arduino program that controls several output pins based on the input from an IR sensor connected to pin `A0`. Here's a short breakdown:

- **Setup**:
  - Pin `A0` is set as an input to read from the IR sensor.
  - Pins `0`, `1`, `2`, `4`, and `13` are set as outputs to control LEDs or other components.
- **Loop**:
  - The program continuously checks the state of the IR sensor on `A0`.
  - If the sensor detects an input (logic HIGH), it turns on all the output pins (`0`, `1`, `2`, `4`, `13`).
  - If the sensor does not detect any input (logic LOW), it turns off all the output pins.

This setup could be used for an IR sensor-based on/off switch or for triggering actions when the sensor detects something.

Chit no 17

This Arduino code reads and displays temperature and humidity data using a DHT11 sensor connected to pin `4`:

- **Libraries**: The `DHT.h` library is used to interface with the DHT11 sensor, and `SD.h` is included (though not used in this code).
- **Setup**: The serial monitor is initialized at `9600` baud rate. The DHT11 sensor is initialized using the `dht.begin()` method.
- **Loop**: Every 2 seconds, the code reads the temperature and humidity from the DHT11 sensor and prints the values to the serial monitor.

The sensor provides real-time temperature (°C) and humidity (%) readings, displayed every 2 seconds.

Chit no 19

The Python script `math_operations.py` defines basic mathematical operations and demonstrates their usage:

- **Functions**:
  - `add(a, b)`: Adds two numbers.
  - `subtract(a, b)`: Subtracts the second number from the first.
  - `multiply(a, b)`: Multiplies two numbers.
  - `divide(a, b)`: Divides the first number by the second. If the second number is zero, it raises a `ValueError`.
- **Main execution**: The script sets `a = 10` and `b = 5` and then prints the results of the addition, subtraction, multiplication, and division of these two numbers.

The code effectively demonstrates basic arithmetic operations, with error handling for division by zero.