SS LAB – ALGORITHMS:

CYCLE 1

FCFS CPU scheduling algorithm:

1. Input: Read number of processes and their burst times.
2. Calculate Waiting Time: Set `wait_time[0] = 0`; for `i = 1` to `x-1`, compute `wait_time[i] = wait_time[i-1] + burst_time[i-1]`.
3. Calculate Turnaround Time: For each process, compute `turnaround_time[i] = burst_time[i] + wait_time[i]`.
4. Output: Print process details and average waiting and turnaround times.

Priority CPU scheduling algorithm:

1. Read the number of processes, burst times, and priority values.
2. Sort processes by priority in descending order.
3. Calculate waiting time for each process based on previous burst times.
4. Print execution order with start and end times.
5. Compute and print average waiting and turnaround times.

Algorithm: Round Robin CPU Scheduling

1. Input:
   - Read `n` (processes) and `ts` (time slice).
   - For each process, read burst time `bt`.
2. Initialize Queue: Enqueue all process IDs.
3. Execution:
   - While queue is not empty:
     - Dequeue process `id`.
     - If `bt[id] > ts`, update time and enqueue `id` again; else set `bt[id]` to `0`.
4. Calculate Waiting Time: `wt[i] = tat[i] - original_bt[i]`.
5. Output: Print PID, Burst Time, Turnaround Time, Waiting Time

Shortest Job First (SJF) scheduling algorithm:

1. Input:Read `n` and burst times `bt[]`.
2. Sort: Sort processes by burst time.
3. Calculate:
   - For each process:
     - Compute `ct[i]`, `tat[i]`, `wt[i]`.
4. Output: Print PID, Burst Time, Turnaround Time, Waiting Time.

Algorithm: FIFO Page Replacement

1. Input: Read `n`, `reference[]`, `fsize`.
2. Initialize: Set `frame[]` to -1.
3. Process: For each reference, check in `frame[]`; if not found, add it and manage the frame index.
4. Output: Print total faults, miss ratio, and hit ratio

Algorithm: LRU Page Replacement

```
1. Input: Read `n`, `reference[]`, `fsize`.
2. Initialize: Load the first `fsize` references into `frame[]`, count faults.
3. Process:
   - For each remaining reference:
     - Check if it's in `frame[]`.
     - If not, record a fault and replace the least recently used page.
4. Output: Print total faults, miss ratio, and hit ratio.
```

Algorithm: C-SCAN Disk Scheduling

```
1. Input: Read `max` (max cylinders), `n` (number of requests), and `req[]` (request array).
2. Validation: Check if any request exceeds `max - 1`; if yes, terminate.
3. Sort: Sort the request array `req[]`.
4. Input Current Head Position: Read the current position of the head.
5. Process Requests:
   - Move from the head to the end of the requests.
   - Jump to the start (cylinder 0), then process remaining requests.
6. Calculate Seek Time: Accumulate total seek time.
7. Output: Print total seek time and seek sequence.
```

Algorithm: FCFS Disk Scheduling

```
1. Input: Read `t` (total tracks), `n` (number of requests), `arr[]` (track requests), and `head` (initial head position).
2. Process Requests:
   - For each request in `arr[]`:
     - Calculate distance from the current track to the requested track.
     - Accumulate total seek count.
     - Update the previous track to the current request.
3. Output: Print seek sequence and total number of seek operations.
```

Here’s a concise algorithm for the provided C program that implements the SCAN disk scheduling algorithm:

Algorithm: SCAN Disk Scheduling

```
1. Input:Read `max` (max cylinders), `n` (number of requests), and `req[]` (track requests).
2. Validation:Check if the first request exceeds `max - 1`; if yes, terminate.
3. Sort: Sort the request array `req[]`.
4. Input Current Head Position: Read the current position of the head.
5. Process Requests:
   - Move from the head to the end of the requests.
   - Reverse direction and service remaining requests.
6. Calculate Seek Time: Accumulate total seek time.
7. Output: Print total seek time and seek sequence.
```

CYCLE 2

Algorithm for Pass-One Assembler

1. Initialize and Open Files.
2. Handle START:
   - If the first line has `START`, set `locctr` to the starting address; otherwise, set `locctr` to 0.
3. process Lines:
   - Read each line, write it to `inter.txt` with `locctr`.
   - If a label exists, add it to `symtab.txt`.
   - Check `opcode` in `optab`:
     - If found, increment `locctr` by 3.
     - Handle directives: `WORD` (3 bytes), `RESW` (3 × operand), `RESB` (operand bytes), `BYTE` (size based on operand).
4. Handle END and Close Files
5. Output Completion Message

Algorithm for Pass-Two Assembler

1. Open Files: Open `optab.txt`, `symtab.txt`, `inter.txt`, and `objpgm.txt`.
2. Process Lines in `inter.txt`:
   - If `START`, write header record.
   - For other opcodes, fetch opcode and symbol addresses to form object code.
   - If `WORD`, add operand directly to object code.
3. Write Text Record: Print and write text record header with object codes.
4. Output Object Codes: Print and write object codes to `objpgm.txt`.
5. Write End Record: Print and write end record with starting address.
6. Close Files.

Algorithm for Absolute Loader

1. Open File: Attempt to open `objpgm.txt`.
2. Check for Header Record:
   - Read the first line; if it starts with 'H', extract the program name, starting address, and length.
3. Process Text Records:
   - For each subsequent line starting with 'T':
     - Extract text address and length.
     - Parse hexadecimal data, converting and printing the byte value at the current address.
4. Process End Record:
   - If the line starts with 'E', extract and print the end address, then terminate reading.
5. Close File: Ensure the file is closed before exiting.