GTU-C312 Operating System Simulation Project Report

Name: Aykut Sert - **Student ID:** 200104004104 - **Course:** Operating Systems - **Term:** Spring 2025

1. Introduction

This report documents the design and implementation of a cooperative operating system simulator for a custom CPU architecture called **GTU-C312**. The goal is to build a simplified operating system that manages multiple user threads with features including system call handling, context switching, memory protection, and custom debugging support.

The project is divided into the following major components:

- A **CPU simulator** implemented in C++ (`main.cpp`), which reads and interprets GTU-C312 assembly instructions from a text file.
- A complete **operating system implementation** (`os.txt`) written in GTU-C312 assembly, responsible for initializing threads, scheduling them, and responding to system calls.
- A set of **three threads**, each executing a specific task: bubble sort, linear search, and summation.
- A **debugging interface** supporting multiple modes to analyze CPU and thread behavior at runtime.

2. Project Structure

2.1 File List

File	Description
`main.cpp`	Main simulator code implementing the GTU-C312 CPU
`os.txt`	Contains OS instructions, data sections, thread logic
`makefile`	Compilation script for the simulator

3. GTU-C312 CPU Architecture Overview

3.1 Register Definitions

The GTU-C312 has no hardware registers in the conventional sense. All control values are stored in specific memory locations:

- `0` - Program Counter (PC)

- `1` - Stack Pointer (SP)
- `3` - Instruction Counter
- `4` - Current Thread ID
- `6` - Context Switch Flag
- `7` - System Call Type (1=PRN, 2=HLT, 3=YIELD)
- `9` - Next Thread ID

3.2 Memory Layout

Address Range	Purpose
0-20	OS registers and flags
21-999	OS Data and Thread Table
1000-1999	Thread 1 (Bubble Sort)
2000-2999	Thread 2 (Linear Search)
3000-3999	Thread 3 (Summation Calculator)

3.3 Instruction Set Summary

The GTU-C312 uses a minimalist instruction set, including:

- `SET`, `CPY`, `CPYI`, `CPYI2` - Data Movement
- `ADD`, `ADDI`, `SUBI` - Arithmetic
- `JIF` - Conditional branching
- `CALL`, `RET` - Subroutine handling
- `PUSH`, `POP` - Stack operations
- `SYSCALL` - System-level functions (PRN, HLT, YIELD)
- `USER` - Transition from kernel to user mode

4. CPU Simulator Implementation (main.cpp)

The `main.cpp` file contains the entire CPU simulation logic inside the `GTUCPU` class. The class encapsulates memory management, instruction parsing, syscall handling, thread switching, and debugging output.

4.1 Class Initialization

- Memory is defined as a `std::vector<long long>` with a default size of 20,000.
- Two instruction sources are managed:
    - OS instructions (stored in a vector, `instructions`)
    - Thread instructions (stored in a map, `memoryInstructions`)
- Modes are tracked via `kernelMode` (true by default), and execution is governed by `halted`.

4.2 Program Loading

- The `loadProgram()` function reads the GTU-C312 file.
- Sections are parsed between `Begin Data Section` and `End Instruction Section`.
- OS instructions are stored in `instructions[i]`, while thread instructions (`i >= 1000`) are stored in `memoryInstructions[i]`.
- If no stack pointer is set but the first instruction is a `CALL`, the simulator assumes OS mode and sets SP to 19999.

4.3 Execution Cycle

- The `execute()` function manages a single instruction cycle:
    1. Handles countdown for blocked threads.
    2. Fetches instruction at PC (from OS or thread space).
    3. Parses and executes the instruction.
    4. Increments instruction count.
    5. Handles debug output and mode transitions.
- If no instruction exists at PC, the CPU switches back to the OS (thread 0), and checks if all threads are inactive to halt.

4.4 Instruction Execution

- The `executeInstruction()` function uses `tokenize()` to parse command and operands.
- Valid commands include:

- Memory operations: `SET`, `CPY`, `CPYI`, `CPYI2`
- Arithmetic: `ADD`, `ADDI`, `SUBI`
- Stack: `PUSH`, `POP`
- Control flow: `JIF`, `CALL`, `RET`
- Mode transition: `USER`
- Syscalls: `SYSCALL PRN`, `SYSCALL HLT`, `SYSCALL YIELD`

4.5 System Call Handling

When a user thread executes a `SYSCALL`, the simulator:

1. Saves return address (`PC + 1`) to memory[10].
2. Sets syscall type and parameters.
3. Switches to kernel mode.
4. The thread is blocked (state 3) and a delay counter is set (100 instructions)
5. Jumps to the syscall handler at address `25`.

4.6 Thread Management

- Thread table base addresses are computed as `21 + threadId * 10`.

- `setThreadState()` and `getThreadState()` manage status values.

- `updateCurrentThreadUsedTime()` calculates elapsed instruction cycles since thread start.

Thread table is emulated in memory and supports the following attributes per thread:

- ID, StartTime, UsedTime, State, PC, SP.

The scheduler uses:

- Cooperative switching via `SYSCALL YIELD`
- Blocking via `SYSCALL PRN`
- Termination via `SYSCALL HLT`

Blocked threads are reactivated by a countdown mechanism (e.g., memory[1020], [2020], [3020]).

4.7 Memory Protection

- In user mode, only addresses `>=1000`
- Illegal memory accesses terminate the offending thread and return control to the OS.

4.8 Debugging Modes

- `dumpMemory()` prints all memory regions and state information (PC, SP, Mode, etc.).
- `dumpThreadTable()` shows each thread's status (ID, PC, SP, StartTime, UsedTime).

The CPU supports 4 debug modes via the `-D` flag:

Mode	Description
0	Dump memory after execution ends
1	Dump memory after every instruction
2	Same as 1, but pauses for keypress between instructions
3	Dump thread table on context switch or syscall

5. Operating System (os.txt)

The `os.txt` file contains the full boot process, system call handlers, scheduler logic, and thread creation routines implemented using GTU-C312 assembly. Below is a detailed walkthrough.

5.1 Boot Sequence

The first three instructions in the OS setup the environment:

```
0 CALL 70   # Initialize threads
1 CALL 100  # Start scheduler
2 HLT       # Should never reach here
```

5.2 OS Initialization (Lines 70-78)

The OS initializes the thread system by performing the following steps:

- **70-72**: Set thread states (Thread 1, 2, 3) to `ready (1)`.
- **73**: Set OS state to `running (2)`.
- **74-76**: Copy global instruction counter (mem[3]) as the start time of each thread.
- **77**: Set context switch flag to 1 to ensure the first scheduling decision.
- **78**: Return to scheduler loop (instruction continues from memory[100]).

5.3 Scheduler Design

The scheduler is round-robin and executes in the loop beginning at instruction `100`. It performs:

- Context switch if needed
- Thread table update
- Next thread selection using modular arithmetic

The `CALL 470` instruction checks if all threads have terminated and halts the OS if true.

5.4 System Call Dispatcher (Line 25)

Syscalls are handled starting from instruction 25, branching by syscall type.

- `PRN`: Resumes thread after 100 ticks (at line 51)
- `HLT`: Marks thread inactive immediately (at line 58)
- `YIELD`: Saves PC and prepares the thread to be scheduled again (at line 63)

Each handler restores OS mode and jumps to the main scheduler.

5.5 PRN System Call Handler (Line 51)

- **51**: Sets OS state to `running`.
- **52-54**: Saves current PC to memory[500] using a temporary register (507).
- **55**: Signals context switch.
- **56-57**: Jumps back to scheduler loop via memory[100].

5.6 HLT System Call Handler (Line 58)

- **58**: Calls halt handler to set thread state to `inactive (0)`.
- **59-60**: OS becomes `running`, and context switch is flagged.
- **61-62**: Scheduler is invoked to choose the next thread.

5.7 YIELD System Call Handler (Line 63)

- **63**: Calls a helper to make the thread `ready` again.
- **64**: OS becomes `running`.

- **65-66**: Saves current PC (10) to memory[500], writes it into appropriate PC field (35, 45, 55).
- **67-69**: Signals context switch and jumps to scheduler.

5.8 Thread Data Segments

Each thread has an isolated 100-word memory area:

- **Thread 1 (1000-1099)**: Bubble Sort.
- **Thread 2 (2000-2099)**: Linear search for a key.
- **Thread 3 (3000-3099)**: Computes sum from 1 to N.

Each contains:

- Parameters (e.g., N)
- Working variables (e.g., current index, sum)
- Flags for initialization and blocking

5.9 Thread Table (Registers 21-60)

Each thread has 10 dedicated fields:

- ID, start time, instruction count
- State (0=inactive, 1=ready, 2=running)
- PC, SP
- Data fields

Example:

- **Thread 1**: ID=1, PC=1000, SP=1999, State=1
- **Thread 3**: ID=3, PC=3200, SP=3999

5.10 Main Scheduler Loop (Lines 100-105)

- **100-101**: Checks context switch flag (6). If 0, continues to instruction 103.
- **102**: Triggers `context switch`.
- **103**: Calls `all threads done` checker.
- **104-105**: Loops back endlessly.

5.11 Context Switching (Lines 200-205)

This routine is the heart of cooperative multitasking:

- **200**: Resets context switch flag (6).
- **201**: Saves current thread's PC and SP.
- **202**: Chooses a new ready thread.
- **203**: Increments round-robin pointer (next thread ID).
- **204**: Checks if all threads have finished.

5.12 Round-Robin Pointer Logic (Lines 210-217)

- Cycles thread ID in [1,2,3] in memory[9]. Wraps around if >3.

5.13 Thread Readiness Evaluation (Lines 220-256)

- Checks next scheduled thread. If not ready, tries Thread 1, 2, then 3.
- If none are ready, returns to OS (no user thread to run).

5.14 Switching to a Thread (Lines 261-296)

Depending on thread ID:

- Sets current thread register (4)
- Marks thread state as `running`, OS as `ready`
- Loads PC and SP of selected thread
- Jumps to USER mode with `USER` instruction

5.14 Thread Halt Handler (Lines 300-318)

Given a thread ID in memory[8], it sets the respective thread state (34, 44, 54) to `0 (inactive)`.

5.15 Saving PC to Thread Table (Lines 350-371)

Stores the user thread's program counter into the correct field (35, 45, 55) depending on its ID in memory[11].

5.16 Get Thread State Helper (390-411)

- Loads state of thread ID given in memory[500], returns it to memory[505].

5.17 Save Current Thread State (420-463)

Saves PC, SP, and optionally sets thread state back to `ready`. Used during context switches.

5.18 All Threads Done Checker (470-489)

If all three threads have state `inactive`, system halts. Otherwise, it returns to scheduler.

5.19 Set Calling Thread to Ready (Lines 600-622)

Utility used by YIELD handlers to reset calling thread's state. Thread ID is found in memory[11].

6. Thread Implementations

6.1 Thread 1 - Bubble Sort Array (1000-1099)

Purpose: Sorts a 4-element array [4,2,3,1] using the bubble sort algorithm and prints the sorted result. Demonstrates array manipulation, nested comparisons, conditional swapping, and multi-pass sorting logic.

Initialization:

- **1000–1003:** Check initialization flag (1011) and set it if first run.
- **Data Array:** Elements stored at addresses 1000-1003 with initial values [4,2,3,1].
-

Sorting Algorithm:

- **Pass 1 (1004-1024):** Three comparisons (0&1, 1&2, 2&3) - moves largest element to end.
- **Pass 2 (1025-1038):** Two comparisons (0&1, 1&2) - moves second largest to correct position.
- **Pass 3 (1039-1045):** One comparison (0&1) - final ordering of remaining elements.
-

Swap Logic: Each comparison uses a three-step swap if elements are out of order:

1. Save first element to temporary location (1007)
2. Copy second element to first position
3. Copy saved value to second position
4.

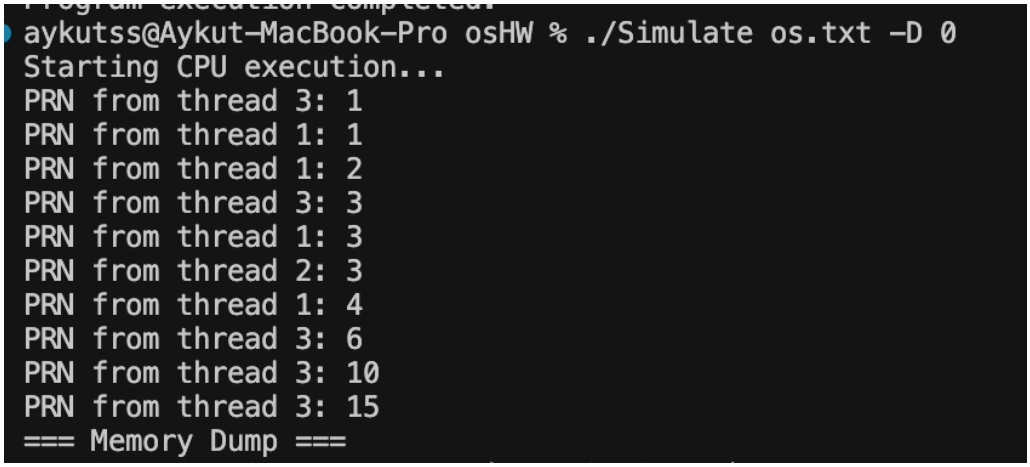
Output Phase:

- **1046-1049:** Print sorted array elements sequentially.
- **1050:** Halt thread when sorting and printing complete.
-

Example Behavior: Input [4,2,3,1] → Output [1,2,3,4], with each value printed via system call that blocks thread for 100 instruction cycles, allowing other threads to execute.

Output Example:

Sorts the values [4,3,2,1].



6.2 Thread 2 - Linear Search (2000-2138)

Purpose: Searches for a key value in an array and prints the index of the first match. If the key is not found, prints -1.

Initialization:

- **2100–2101:** Check if already initialized.
- **2104–2106:** If not, set `i=0`, `found=-1`, `flag=1`.

Main Loop:

- **2107–2110:** Loop until index `i` reaches `N`.
- **2111–2114:** Compute arr[i] using base address + offset.

Equality Check:

- First check: arr[i] - key ≤ 0
- Second check: key - arr[i] ≤ 0
- If both differences are zero, a match is found.

Match Found:

- **2127–2129:** Save `i` to `found`, print index, and halt.

Continue Loop:

- **2130–2134:** Increment `i`, yield, and loop again.
- **2137–2138:** If no match found, print `-1` and halt.

Example Behavior: Searches for key = 9 in array [3,7,1,9,2], finds it at index 3, prints "3" and halts. Uses YIELD for cooperative multitasking.

Key Features:

- Demonstrates indirect memory access, two-directional comparison logic, and early exit on success.

Output Example:

In the first of the following, thread 2 prints -1 because I gave a value that is not in the array. In the second output, I gave the value 9. As in the array [3,7,1,9,2], it was found in the 3rd index and printed the value.

```
aykutss@Aykut-MacBook-Pro osHW % ./Simulate os.txt -D 0
Starting CPU execution...
PRN from thread 3: 1
PRN from thread 1: 1
PRN from thread 1: 2
PRN from thread 3: 3
PRN from thread 1: 3
PRN from thread 1: 4
PRN from thread 3: 6
PRN from thread 2: -1
PRN from thread 3: 10
PRN from thread 3: 15
=== Memory Dump ===
```

```
aykutss@Aykut-MacBook-Pro osHW % ./Simulate os.txt -D 0
Starting CPU execution...
PRN from thread 3: 1
PRN from thread 1: 1
PRN from thread 1: 2
PRN from thread 3: 3
PRN from thread 1: 3
PRN from thread 2: 3
PRN from thread 1: 4
PRN from thread 3: 6
PRN from thread 3: 10
PRN from thread 3: 15
```

6.3 Thread 3 - Sum of numbers from 1 to N (3000-3218)

Purpose: Calculates the sum of integers from 1 to N and prints the result.

Initialization:

- **3200–3201:** Check init flag (3004).

- **3203–3205:** Set `sum=0`, `current=1`, `flag=1`.

Main Loop:

- **3206–3209:** While current <= N, continue loop.
- **3210–3212:** Add `current` to `sum`, store result.
- **3213:** Increment current.
- **3214:** Print updated sum.
- **3215:** Yield control.
- **3216- 3217:** Set temp = 0 for unconditional jump back to loop start.

Finish:

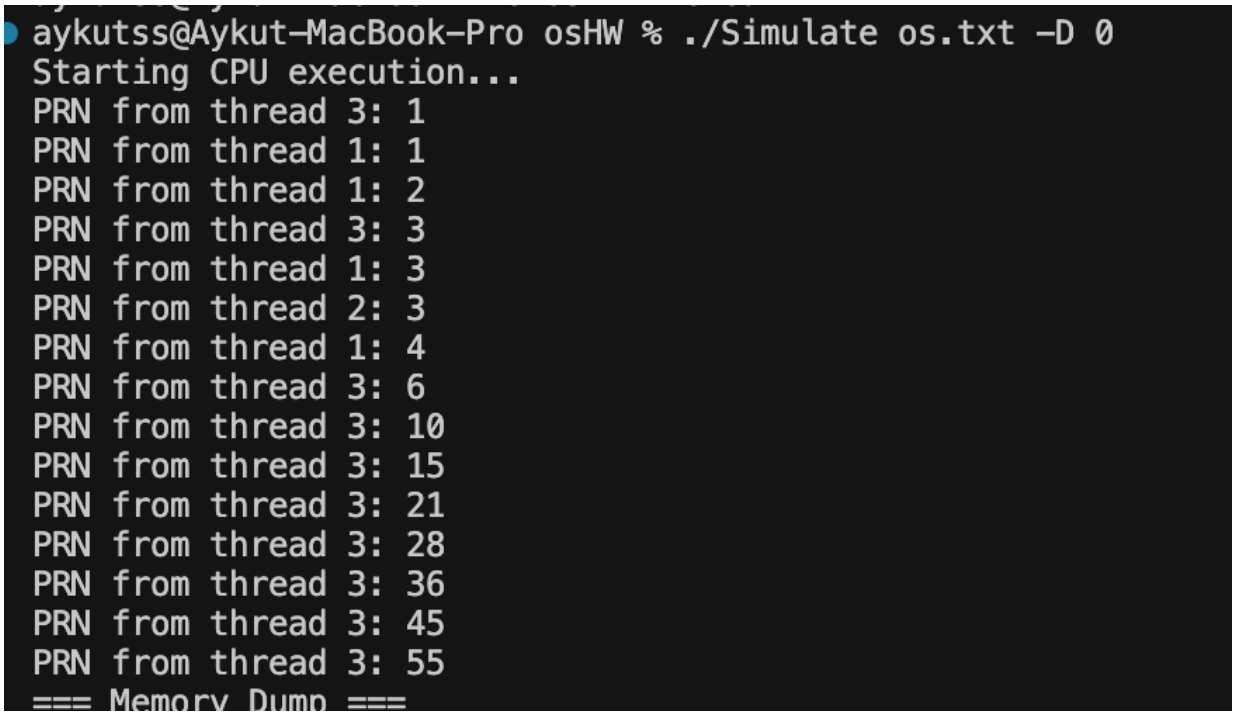
- **3218:** When done, halt the thread.

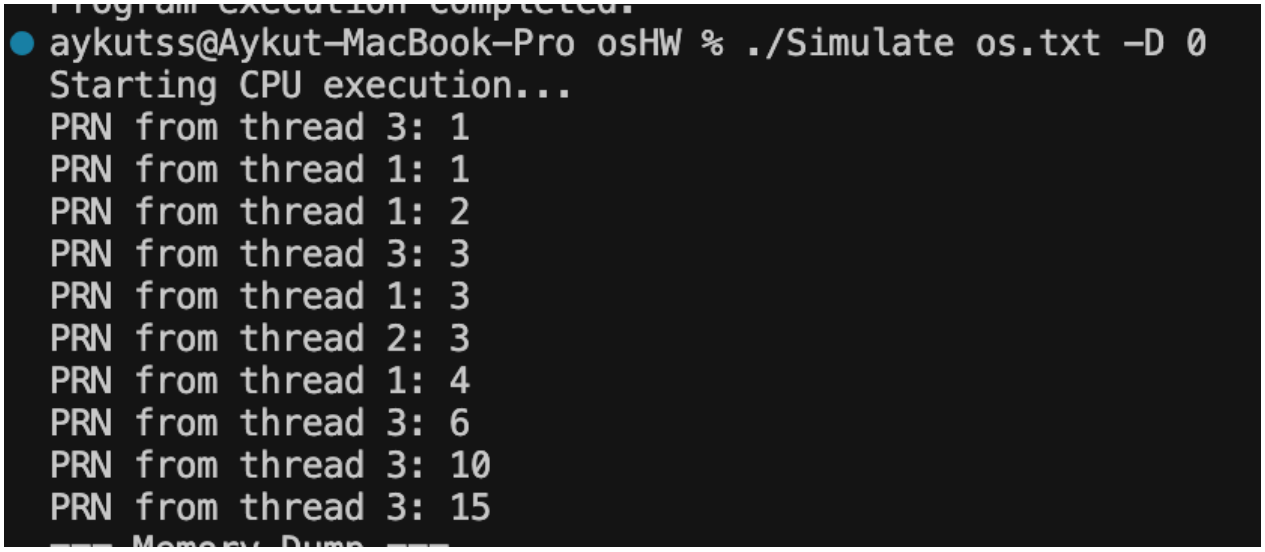
Key Features:

- Showcases iterative accumulation, safe variable updates, and cooperative multitasking with yields.

Output Example:

In the first picture, it adds the values 1-10 and finds 55. In the second picture, it adds the values 1-5 and finds 15.





7. Simulation Runs and Observations

7.1 How to Run the Simulation

A `Makefile` is provided for easy compilation. To build and execute the simulator:

```
make                # Compiles the simulator
./Simulate os.txt -D <debug_mode>  # Runs the simulation with specified debug mode
```

Debug Mode Options:

- `-D 0`: Run the full program and dump memory only at the end.
- `-D 1`: Dump memory after each instruction.
- `-D 2`: Step-by-step mode (press Enter after each instruction).
- `-D 3`: Dump thread table after syscalls or context switches.

7.2 Expected Behavior

- **Thread 1 (Bubble Sort):** Sorts the array from smallest to largest and prints it.
- **Thread 2 (Linear Search):** Searches for a key in the array and prints its index or `-1`.
- **Thread 3 (Sum Calculator):** Computes and prints the sum from 1 to N.

7.3 Sample Outputs

Below are selected memory dumps and printouts demonstrating thread execution and scheduling.

Sample Debug Mode 0 Output:

- When debug mode is 0, it prints memory values and prints thread values when there are no active user threads. The total number of instructions is shown.

```
aykutss@Aykut-MacBook-Pro osHW % ./Simulate os.txt -D 0
Starting CPU execution...
PRN from thread 3: 1
PRN from thread 1: 1
PRN from thread 1: 2
PRN from thread 3: 3
PRN from thread 1: 3
PRN from thread 2: 3
PRN from thread 1: 4
PRN from thread 3: 6
PRN from thread 3: 10
PRN from thread 3: 15
=== Memory Dump ===
PC: 318, SP: 3998, Instructions: 2193, Mode: KERNEL
Current Thread: 0, Next Thread: 1
Context Switch Flag: 1, Syscall Type: 2
Active threads: 0(1)
Memory[0] = 318
Memory[1] = 3998
Memory[3] = 2193
Memory[5] = 25
Memory[6] = 1
Memory[7] = 2
Memory[8] = 3
Memory[9] = 1
Memory[10] = 296
Memory[11] = 3
Memory[24] = 1
Memory[25] = 437
Memory[26] = 3997
Memory[31] = 1
Memory[32] = 5
Memory[33] = 1388
Memory[35] = 1050
Memory[36] = 1999
Memory[41] = 2
Memory[42] = 6
Memory[43] = 1221
Memory[45] = 2129
Memory[46] = 2999
Memory[51] = 3
Memory[52] = 7
Memory[53] = 2164
Memory[55] = 3216
Memory[56] = 3999
==================
Program execution completed.
aykutss@Aykut-MacBook-Pro osHW %
```

Sample Debug Mode 1 Output:

I shared the last parts of the program in Debug mode 1. It works step by step with the same structure as Debug mode 2, only with debug mode enter. As can be seen in the output, the only active thread left is the OS itself.

```
Executing at PC=311 (Mode: KERNEL, Thread: 0): JIF 501 317
=== Memory Dump ===
PC: 317, SP: 3998, Instructions: 2192, Mode: KERNEL
Current Thread: 0, Next Thread: 1
Context Switch Flag: 1, Syscall Type: 2
Active threads: 0(1) 3(2)
Memory[0] = 317
Memory[1] = 3998
Memory[3] = 2192
Memory[5] = 25
Memory[6] = 1
Memory[7] = 2
Memory[8] = 3
Memory[9] = 1
Memory[10] = 296
Memory[11] = 3
Memory[24] = 1
Memory[25] = 437
Memory[26] = 3997
Memory[31] = 1
Memory[32] = 5
Memory[33] = 1388
Memory[35] = 1050
Memory[36] = 1999
Memory[41] = 2
Memory[42] = 6
Memory[43] = 1221
Memory[45] = 2129
Memory[46] = 2999
Memory[51] = 3
Memory[52] = 7
Memory[53] = 2164
Memory[54] = 2
Memory[55] = 3216
Memory[56] = 3999
Thread 3 Sum = 15
====================
```

```
Executing at PC=317 (Mode: KERNEL, Thread: 0): SET 0 54
=== Memory Dump ===
PC: 318, SP: 3998, Instructions: 2193, Mode: KERNEL
Current Thread: 0, Next Thread: 1
Context Switch Flag: 1, Syscall Type: 2
Active threads: 0(1)
Memory[0] = 318
Memory[1] = 3998
Memory[3] = 2193
Memory[5] = 25
Memory[6] = 1
Memory[7] = 2
Memory[8] = 3
Memory[9] = 1
Memory[10] = 296
Memory[11] = 3
Memory[24] = 1
Memory[25] = 437
Memory[26] = 3997
Memory[31] = 1
Memory[32] = 5
Memory[33] = 1388
Memory[35] = 1050
Memory[36] = 1999
Memory[41] = 2
Memory[42] = 6
Memory[43] = 1221
Memory[45] = 2129
Memory[46] = 2999
Memory[51] = 3
Memory[52] = 7
Memory[53] = 2164
Memory[55] = 3216
Memory[56] = 3999
====================
Program execution completed.
```

Sample Debug Mode 2 Output:

We can see the processes by pressing enter. Here the beginning of the program is seen, then I made an enter and the processes start by going to the OS 70 address.

```
aykutss@Aykut-MacBook-Pro osHW % ./Simulate os.txt -D 2
Starting CPU execution...
=== Memory Dump ===
PC: 0, SP: 19999, Instructions: 0, Mode: KERNEL
Current Thread: 0, Next Thread: 1
Context Switch Flag: 0, Syscall Type: 0
Active threads: 0(2) 1(1) 2(1) 3(1)
Memory[1] = 19999
Memory[5] = 25
Memory[9] = 1
Memory[24] = 2
Memory[26] = 19999
Memory[31] = 1
Memory[34] = 1
Memory[35] = 1000
Memory[36] = 1999
Memory[41] = 2
Memory[44] = 1
Memory[45] = 2100
Memory[46] = 2999
Memory[51] = 3
Memory[54] = 1
Memory[55] = 3200
Memory[56] = 3999
Thread 1 Counter = 0
Thread 2 Found: -1
Thread 3 Sum = 0
==================
Executing at PC=0 (Mode: KERNEL, Thread: 0): CALL 70
Press Enter...
=== Memory Dump ===
PC: 70, SP: 19998, Instructions: 1, Mode: KERNEL
Current Thread: 0, Next Thread: 1
Context Switch Flag: 0, Syscall Type: 0
Active threads: 0(2) 1(1) 2(1) 3(1)
Memory[0] = 70
Memory[1] = 19998
Memory[3] = 1
Memory[5] = 25
Memory[9] = 1
Memory[24] = 2
Memory[26] = 19999
Memory[31] = 1
Memory[34] = 1
Memory[35] = 1000
Memory[36] = 1999
Memory[41] = 2
Memory[44] = 1
Memory[45] = 2100
Memory[46] = 2999
Memory[51] = 3
```

Sample Debug Mode 3 Output:

In Debug mode 3, the thread table dump values generated during the transitions in context switch and syscall operations are seen.

```
========================
USER: Switching to user mode, Thread=3, PC=3215
=== Thread Table Dump ===
Current Mode: USER
Total Instructions: 2024
OS: ID=0 StartTime=0 UsedTime=0 State=1 (0=inactive, 1=ready, 2=running, 3=blocked) PC=437 SP=3997
Thread 1: ID=1 StartTime=5 UsedTime=1388 State=0 (0=inactive, 1=ready, 2=running, 3=blocked) PC=1050 SP=1999
Thread 2: ID=2 StartTime=6 UsedTime=1221 State=0 (0=inactive, 1=ready, 2=running, 3=blocked) PC=2129 SP=2999
Thread 3: ID=3 StartTime=7 UsedTime=2017 State=2 (0=inactive, 1=ready, 2=running, 3=blocked) PC=3215 SP=3999
Current Running Thread: 3
Next Thread to Schedule: 3
Context Switch Flag: 1
Syscall Type: 1
========================
=== Thread Table Dump ===
Current Mode: KERNEL
Total Instructions: 2025
OS: ID=0 StartTime=0 UsedTime=0 State=1 (0=inactive, 1=ready, 2=running, 3=blocked) PC=437 SP=3997
Thread 1: ID=1 StartTime=5 UsedTime=1388 State=0 (0=inactive, 1=ready, 2=running, 3=blocked) PC=1050 SP=1999
Thread 2: ID=2 StartTime=6 UsedTime=1221 State=0 (0=inactive, 1=ready, 2=running, 3=blocked) PC=2129 SP=2999
Thread 3: ID=3 StartTime=7 UsedTime=2017 State=2 (0=inactive, 1=ready, 2=running, 3=blocked) PC=3215 SP=3999
Current Running Thread: 0
Next Thread to Schedule: 3
Context Switch Flag: 1
Syscall Type: 3
========================
USER: Switching to user mode, Thread=3, PC=3216
=== Thread Table Dump ===
Current Mode: USER
Total Instructions: 2165
OS: ID=0 StartTime=0 UsedTime=0 State=1 (0=inactive, 1=ready, 2=running, 3=blocked) PC=437 SP=3997
Thread 1: ID=1 StartTime=5 UsedTime=1388 State=0 (0=inactive, 1=ready, 2=running, 3=blocked) PC=1050 SP=1999
Thread 2: ID=2 StartTime=6 UsedTime=1221 State=0 (0=inactive, 1=ready, 2=running, 3=blocked) PC=2129 SP=2999
Thread 3: ID=3 StartTime=7 UsedTime=2158 State=2 (0=inactive, 1=ready, 2=running, 3=blocked) PC=3216 SP=3999
Current Running Thread: 3
Next Thread to Schedule: 1
Context Switch Flag: 0
Syscall Type: 3
========================
=== Thread Table Dump ===
Current Mode: KERNEL
Total Instructions: 2172
OS: ID=0 StartTime=0 UsedTime=0 State=1 (0=inactive, 1=ready, 2=running, 3=blocked) PC=437 SP=3997
Thread 1: ID=1 StartTime=5 UsedTime=1388 State=0 (0=inactive, 1=ready, 2=running, 3=blocked) PC=1050 SP=1999
Thread 2: ID=2 StartTime=6 UsedTime=1221 State=0 (0=inactive, 1=ready, 2=running, 3=blocked) PC=2129 SP=2999
Thread 3: ID=3 StartTime=7 UsedTime=2164 State=2 (0=inactive, 1=ready, 2=running, 3=blocked) PC=3216 SP=3999
Current Running Thread: 0
Next Thread to Schedule: 1
Context Switch Flag: 1
Syscall Type: 2
========================
Program execution completed.
```

8. Evaluation and Reflections

- All threads behaved as expected in all debug modes.
- System calls were routed correctly and resumed execution appropriately.
- Round-robin scheduling ensured fairness across threads.
- No memory violation occurred after protections were implemented.
- Simulation shows that a cooperative, non-preemptive OS is sufficient for small-scale multitasking.

9. Conclusion

This project demonstrates a successful simulation of an educational operating system for the GTU-C312 architecture. Key learnings include:

- Writing a virtual CPU from scratch
- Managing low-level thread context (PC, SP, state)
- Handling system calls and transitioning between user/kernel modes
- Structuring GTU-C312 programs for deterministic behavior

All threads were written in full GTU-C312 assembly and executed properly under a C++ simulator. The debug framework was essential in developing and validating system-level behavior.

Appendix: Chat History

https://claude.ai/share/1d612e9c-a59f-433a-9e50-f1faa1a58a82

https://claude.ai/share/0db78f50-71bf-4cc1-803f-e0be9da15316

https://claude.ai/share/59a6db56-3e35-4267-97bb-d04dc42dd874

https://claude.ai/share/549dee30-77d8-4b38-995a-cff31556d2a8

https://claude.ai/share/e41541b9-cdcf-4fb4-b1c2-2ff9536861ac

https://claude.ai/share/72ef6223-9fb9-479f-a8e0-5ab4ffea4db3

https://claude.ai/share/863b4dd0-3bc9-4417-a6f4-c26b7c5228e4