GTU-C312 Operating-System Project

Final Report – Spring 2025 200104004003

Emre Yavuz

June 6, 2025

Contents

1 Introduction

This report documents the design and implementation of a co-operative operating system and CPU simulator built for the GTU-C312 instruction-set architecture (ISA). The simulator, written in Python, executes a single `.gtu` file that packages the kernel, thread code and all data in one image file.:contentReference[oaicite:0]index=0

2 Overall Project Layout

- `cpu_simulator/` – simulator core (`cpu.py`, `bios.py`, `simulate.py`, `main.py`)
- `os_plus_threads.gtu` – the combined kernel & user-thread program
- **Debug artefacts** – `instructions_output.txt`, `debugX_output.txt`, etc.

:contentReference[oaicite:1]index=1

3 CPU Simulator (`cpu.py`)

The `cpu.py` module implements a complete simulation of the GTU-C312 CPU as required by the project specification. This simulator provides foundational functionality for executing both operating system code and user-level threads.

3.1 Memory Architecture

The CPU simulator models memory as a flat array of 11,000 signed integers to accommodate both OS routines and up to 10 user threads. Since Python supports arbitrary-precision integers, it can safely represent all valid GTU-C312 values without overflow.

- **Memory-Mapped Registers**: GTU-C312 does not include physical registers; instead, the following memory locations are reserved:
  - `memory[0]`: Program Counter (PC)
  - `memory[1]`: Stack Pointer (SP)
  - `memory[2]`: System Call Result
  - `memory[3]`: Instruction Execution Counter
- These registers are accessed via Python `property` methods for clean encapsulation.

3.2 CPU Modes and Memory Protection

The CPU supports dual-mode operation and enforces strict memory access rules.

- **Modes**: The CPU begins in `KERNEL` mode. Executing the `USER` instruction switches it to `USER` mode, while all system calls automatically switch it back to `KERNEL`.
- **Protection**: The simulator prevents instructions in `USER` mode from accessing any memory address below 1000. Violations halt the CPU to simulate thread termination.

3.3 Instruction Execution

All required GTU-C312 instructions are implemented within the `_decode_execute` method, including:

- **Data Transfer**: `SET, CPY, CPYI`
- **Arithmetic**: `ADD, ADDI, SUBI`
- **Control Flow**: `JIF, CALL, RET`
- **Stack**: `PUSH, POP`
- **System Management**: `HLT, USER`
- **Optional Instruction**: `CPYI2` is also implemented and tested.

3.4 System Call Handling

The simulator provides a flexible and extensible syscall infrastructure:

- **Supported Calls**: `SYSCALL_PRN`, `SYSCALL_HLT`, `SYSCALL_YIELD`
- **Handler Design**: SYSCALL instructions transfer control to handler addresses stored in reserved memory, enabling OS-level management.
- **Context Saving**: Before jumping to the OS:
  - The syscall type is written to `memory[10]`.
  - The return address is stored in `memory[2]`.
- `SYSCALL_PRN` is executed directly by the Python simulator and prints memory values to output.

3.5 Testing and Validation

A comprehensive function `run_all_cpu_tests()` ensures full coverage and correct behavior of all instructions. The test suite validates:

- Arithmetic operations and edge conditions
- Stack safety (CALL, RET, PUSH, POP)
- Control flow correctness
- Memory access protection
- All system call logic

This test-driven approach confirms the correctness and reliability of the simulator.

4 BIOS (`bios.py`)

4.1 GTU Loader (`bios.py`)

The loader parses a single `.gtu` program file and separates its contents into two distinct segments:

- **Data Segment:** address–value pairs to pre-load memory.
- **Instruction Segment:** address–instruction mappings.

The parsing is robust and includes:

1. Support for flexible formats like `ADDR = VALUE` or `ADDR VALUE`.
2. Case-insensitive section headers (`BEGIN DATA SECTION`, `BEGIN INSTRUCTION SECTION`, etc.).
3. Early detection of duplicates: the first definition is preserved, later ones ignored with warnings.
4. Overlap check: by default, code and data may not define the same memory address.

The result is a pair of sorted lists: one for data initialization, one for executable code. This separation simplifies memory initialization before simulation starts.

5 Kernel (`os_plus_threads.gtu`)

6 Summary of `os_plus_threads.gtu`

The `os_plus_threads.gtu` file is a complete assembly-level implementation of an operating system for the GTU-C312 simulator. It includes the OS kernel, system call handlers, thread table initialization, and three active user threads. The structure is logically split into a `DATA` section and an `INSTRUCTION` section, parsed by the simulator BIOS.

6.1 System Initialization and Memory Layout

- **Entry point:** The initial PC is set to 200, and SP to 990.
- **Syscall routing:** Memory addresses 11–13 point to handlers for HLT, YIELD, and PRN respectively.
- **Thread management:** Memory

  15]$tracksthecurrentlyrunningthread, andMemory$16]$storestheIDofthenextthreadtorun.$

6.2 Thread Control Blocks (TCBs)

Each thread has a TCB occupying 7 contiguous memory cells, containing:

1. Thread ID
2. State (0: Ready, 1: Running, 2: Blocked, 3: Halted)
3. Saved PC
4. Saved SP
5. Unblock instruction count (PRN delay)
6. Start instruction count (for IE tracking)
7. Total instructions used

TCBs are defined for the OS (ID 0) and threads 1 through 10. Only threads 1, 2, and 3 are active.

6.3 User Thread Data Regions

- **Thread 1 (1000–1999):** Uses variables at 1050–1052.
- **Thread 2 (2000–2999):** Uses 2050–2052 similarly.
- **Thread 3 (3000–3999):** Contains bubble sort data at 3550+, including array, counters, and temp storage.

6.4 System Call Handlers

SYSCALL_HLT (300): Marks the calling thread as Halted and jumps to the scheduler.

SYSCALL_YIELD (400): Saves thread context (PC, SP), marks it Ready, updates next_thread_to_run_id, and schedules.

SYSCALL_PRN (500): After CPU prints the value, blocks the thread by storing unblock time as current IE + 100, then invokes the scheduler.

6.5 Scheduler (600)

The core logic of the OS:

- Unblocks threads whose delay expired (by checking IE).
- Detects whether all threads have halted.
- Implements round-robin thread selection.
- Loads context of selected thread and transfers control via USER instruction.

7 User Threads and Multitasking Demonstration

7.1 Threads 1 and 2

These threads demonstrate basic computation loops. Both initialize a counter and a sum variable, increment the sum in each iteration, then print the result using SYSCALLPRN, which causes a blocking delay of 100 instruction cycles.

- **Thread 1:** Starts at address 1000. Uses variables at 1050–1052. Loops with initial counter set to 8.
- **Thread 2:** Starts at address 2000. Similar structure, counter set to 7. Uses 2050–2052.

They alternate execution due to the cooperative multitasking enforced by PRN-blocking and the round-robin scheduler.

7.2 Thread 3: Bubble Sort

This thread implements a full bubble sort algorithm. It accesses and manipulates an array of five values (3551–3555), using indirect memory access.

- Nested loops calculate and compare elements.
- If needed, elements are swapped using temporary variables and indirect CPY operations.

- After sorting, the thread prints each array element via SYSCALL$_PRN$. This shows the system's ability to manage complex stateful threads that perform nontrivial algorithms over time.

7.3 Threads 4–10

Threads 4 through 10 are defined in the TCB area with unique IDs and valid PC/SP values, but their code consists of a single SYSCALLHLT instruction. They serve as stubs and demonstrate scalability.

7.4 Conclusion

The complete osplusthreads.gtu file models a cooperative, user-space thread environment. With real system call handling, state transitions, blocking and context switching, it simulates key OS behaviors on a custom ISA.

8 User Threads

Three user threads are implemented in the system, each designed to test and demonstrate specific OS features such as blocking, context saving, and indirect memory access. These threads are assigned to fixed memory regions, with their own code, data, and private stack space.

Thread 1: Accumulator Loop (Starts at 1000)

Thread 1 is initialized with its PC at 1000 and SP at 1990. It uses the following memory locations:

- `1050:` Initial counter value ($i = 8$)
- `1051:` Accumulated sum
- `1052:` Loop continuation flag (dummy register)

Each iteration adds the counter value to the sum, decrements the counter, and prints the sum via `SYSCALL_PRN`. This system call blocks the thread for 100 instruction cycles, allowing other threads to be scheduled. The loop exits when $i = 0$, and the thread halts using `SYSCALL_HLT`.

Thread 2: Second Accumulator Loop (Starts at 2000)

This thread is structurally identical to Thread 1 but with a different counter:

- `2050:` Counter ($i = 7$)
- `2051:` Accumulated sum
- `2052:` Dummy control variable

Located in the 2000–2999 range, Thread 2 executes the same sum-accumulate-print cycle as Thread 1. The main purpose of this thread is to interleave with Thread 1 via the PRN-induced blocking and to test the round-robin scheduling mechanism.

Thread 3: Bubble Sort and Printing (Starts at 3000)

Thread 3 performs a full bubble sort on a static array of five integers. It operates on a much more complex memory layout:

- `3550:` Number of elements $N = 5$
- `3551{3555:` Array values
- `3560{3570:` Loop counters (i, j), array pointers, temporary swap variables, etc.
- `3580{3583:` Used during the final print phase

The thread:

1. Reads N and calculates loop limits for the nested bubble sort.
2. Computes addresses using `ADD` and indirect access via `CPYI`.
3. Compares and swaps values if needed.
4. After sorting, iterates over the array again and prints each value using `SYSCALL_PRN`.

This thread showcases indirect memory access, nested loops, and the system's ability to handle long-running, stateful computation.

Summary

All three threads:

- Are defined with their own TCB (e.g., Thread 1: TCB at 30–36).
- Use memory exclusively in their assigned range (1000s, 2000s, 3000s).
- Cooperate with the kernel through system calls for printing, yielding, or halting.

Threads 4 through 10 are defined with valid TCBs but contain only a `SYSCALL_HLT`, serving as placeholders for scalability demonstration.

9 Memory-use Summary

The system organizes memory into distinct regions to ensure modularity, thread isolation, and proper OS functionality.

- **0–20: Registers & Kernel Globals** Reserved for CPU and OS internal use. Includes:
  - Program Counter (PC), Stack Pointer (SP)
  - Syscall result holder
  - Instruction execution counter
  - Syscall handler addresses (memory[11–13])
  - Current and next thread IDs (memory[15–16])
- **20–149: Thread Control Block (TCB) Area** Each TCB uses 7 words:
  - Thread ID, State, Saved PC, Saved SP
  - Unblock IE count, Start IE, Used IE

  Threads 0 through 10 are pre-defined here. Only Threads 1–3 are active in this project.
- **200–999: Kernel Code Region** Contains OS startup, syscall handlers (HLT, YIELD, PRN), scheduler, and thread unblock logic. Kernel instructions are executed in privileged mode and interact with memory-mapped syscall control variables.
- **User Thread Memory Blocks** Each thread owns a private block of 1000 addresses:
  - **Thread 1:** Code and data from 1000 to 1999; stack grows downward from 1990.
  - **Thread 2:** Code and data from 2000 to 2999; stack starts at 2990.
  - **Thread 3:** Code and data from 3000 to 3999; stack starts at 3990.

  These blocks include both static variables (e.g., counters, array buffers) and dynamic memory for runtime use.

10 Conclusion

This project demonstrates a full simulation of a cooperative operating system and user-level threading model built on a custom instruction set (GTU-C312). The system achieves functional integration between hardware-level simulation and OS-level thread management.

Key Accomplishments

- A fully functional Python-based CPU simulator capable of instruction execution, syscall handling, and memory tracing.
- An operating system written entirely in assembly-like GTU-C312 code, featuring system call handlers for halting, yielding, and printing.
- A working scheduler that implements round-robin logic and handles thread blocking/unblocking based on instruction timing.
- Three user threads that run concurrently, demonstrating simple loops, I/O blocking, and a non-trivial algorithm (bubble sort).

The current system serves as a strong foundation for learning operating system internals and low-level systems programming. It reflects real-world OS mechanisms such as context switching, syscall dispatching, and scheduler logic — all within a pedagogically controlled environment.

graphicx

11 Tests

To verify the correctness and behavior of the OS and CPU simulator, several test scenarios were executed. Each test captures a specific aspect of system functionality, including thread switching, system call effects, and memory consistency.

12 Tests

Thread 1 - Thread 2: Alternating Output

Thread 1 Value Register Test

Thread 3: Bubble Sort Execution

Instruction ve Debug Modu Çıktıları

```
#---------------------------------------------------
    # THREAD 1 KOMUTLARI (Baslangic: 1000) - Basitlestirilmis Hali
    #---------------------------------------------------
1000: SET 8, 1050       #SYSCALL_HLT       i = 8
1001: SET 0, 1052
1002: SET 0, 1051        # sum = 0
1003: ADDI 1051, 1050      # sum = sum + i
1004: ADD 1050, -1       # i = i - 1
1005: SYSCALL_PRN 1051  # print current sum
1006: JIF 1050, 1008     # if i <= 0 goto end
1007: JIF 1052, 1003   # give control back to OS
1008: SYSCALL_HLT      # end of thread


#---------------------------------------------------
    # THREAD 2 KOMUTLARI (Baslangic: 2000) - Basitlestirilmis Hali
    #---------------------------------------------------
2000: SET 7, 2050        #SYSCALL_HLT        # i = 5
2001: SET 0, 2052        # dummy for loop (true)
2002: SET 0, 2051        # sum = 0
2003: ADDI 2051, 2050    # sum += i
2004: ADD 2050, -1       # i--
2005: SYSCALL_PRN 2051   # print sum
2006: JIF 2050, 2008     # if i <= 0 goto HLT
2007: JIF 2052, 2003     # loop back
2008: SYSCALL_HLT        # end thread
```

Figure 1: T1 ve T2'nin sıralı şekilde çalıştığı temel test.

t2value.jpg

```
SYSCALL_PRN VALUE=8| TID=1 | PC=1005 | IE=56
SYSCALL_PRN VALUE=7| TID=2 | PC=2005 | IE=167
SYSCALL_PRN VALUE=100| TID=3 | PC=3059 | IE=611
SYSCALL_PRN VALUE=15| TID=1 | PC=1005 | IE=724
SYSCALL_PRN VALUE=13| TID=2 | PC=2005 | IE=834
SYSCALL_PRN VALUE=101| TID=3 | PC=3059 | IE=956
SYSCALL_PRN VALUE=21| TID=1 | PC=1005 | IE=1069
SYSCALL_PRN VALUE=18| TID=2 | PC=2005 | IE=1179
SYSCALL_PRN VALUE=102| TID=3 | PC=3059 | IE=1301
SYSCALL_PRN VALUE=26| TID=1 | PC=1005 | IE=1414
SYSCALL_PRN VALUE=22| TID=2 | PC=2005 | IE=1524
SYSCALL_PRN VALUE=103| TID=3 | PC=3059 | IE=1646
SYSCALL_PRN VALUE=30| TID=1 | PC=1005 | IE=1759
SYSCALL_PRN VALUE=25| TID=2 | PC=2005 | IE=1869
SYSCALL_PRN VALUE=104| TID=3 | PC=3059 | IE=1991
SYSCALL_PRN VALUE=33| TID=1 | PC=1005 | IE=2104
SYSCALL_PRN VALUE=27| TID=2 | PC=2005 | IE=2214
SYSCALL_PRN VALUE=35| TID=1 | PC=1005 | IE=2430
SYSCALL_PRN VALUE=28| TID=2 | PC=2005 | IE=2528
SYSCALL_PRN VALUE=36| TID=1 | PC=1005 | IE=2669
```

Düzenle t1value.jpg

```
SYSCALL_PRN VALUE=8| TID=1 | PC=1005 | IE=56
SYSCALL_PRN VALUE=7| TID=2 | PC=2005 | IE=167
SYSCALL_PRN VALUE=100| TID=3 | PC=3059 | IE=611
SYSCALL_PRN VALUE=15| TID=1 | PC=1005 | IE=724
SYSCALL_PRN VALUE=13| TID=2 | PC=2005 | IE=834
SYSCALL_PRN VALUE=101| TID=3 | PC=3059 | IE=956
SYSCALL_PRN VALUE=21| TID=1 | PC=1005 | IE=1069
SYSCALL_PRN VALUE=18| TID=2 | PC=2005 | IE=1179
SYSCALL_PRN VALUE=102| TID=3 | PC=3059 | IE=1301
SYSCALL_PRN VALUE=26| TID=1 | PC=1005 | IE=1414
SYSCALL_PRN VALUE=22| TID=2 | PC=2005 | IE=1524
SYSCALL_PRN VALUE=103| TID=3 | PC=3059 | IE=1646
SYSCALL_PRN VALUE=30| TID=1 | PC=1005 | IE=1759
SYSCALL_PRN VALUE=25| TID=2 | PC=2005 | IE=1869
SYSCALL_PRN VALUE=104| TID=3 | PC=3059 | IE=1991
SYSCALL_PRN VALUE=33| TID=1 | PC=1005 | IE=2104
SYSCALL_PRN VALUE=27| TID=2 | PC=2005 | IE=2214
SYSCALL_PRN VALUE=35| TID=1 | PC=1005 | IE=2430
SYSCALL_PRN VALUE=28| TID=2 | PC=2005 | IE=2528
SYSCALL_PRN VALUE=36| TID=1 | PC=1005 | IE=2669
```

Figure 2: Thread 1 ve 2'nin ikinci versiyon test sonucu.

```
    #----------------------------------------------------
1000: SET 14, 1050       #SYSCALL_HLT       i = 8
1001: SET 0, 1052
1002: SET 0, 1051        # sum = 0
1003: ADDI 1051, 1050      # sum = sum + i
1004: ADD 1050, -1       # i = i - 1
1005: SYSCALL_PRN 1051  # print current sum
1006: JIF 1050, 1008     # if i <= 0 goto end
1007: JIF 1052, 1003   # give control back to OS
1008: SYSCALL_HLT      # end of thread


#----------------------------------------------------
    # THREAD 2 KOMUTLARI (Baslangic: 2000) - Basitlestirilmis Hali
    #----------------------------------------------------
2000: SET 13, 2050        #SYSCALL_HLT        # i = 5
2001: SET 0, 2052        # dummy for loop (true)
2002: SET 0, 2051        # sum = 0
2003: ADDI 2051, 2050    # sum += i
2004: ADD 2050, -1       # i--
2005: SYSCALL_PRN 2051   # print sum
2006: JIF 2050, 2008     # if i <= 0 goto HLT
2007: JIF 2052, 2003     # loop back
2008: SYSCALL_HLT        # end thread
```

Figure 3: .

```
SYSCALL_PRN VALUE=14| TID=1 | PC=1005 | IE=56
SYSCALL_PRN VALUE=13| TID=2 | PC=2005 | IE=167
SYSCALL_PRN VALUE=100| TID=3 | PC=3059 | IE=611
SYSCALL_PRN VALUE=27| TID=1 | PC=1005 | IE=724
SYSCALL_PRN VALUE=25| TID=2 | PC=2005 | IE=834
SYSCALL_PRN VALUE=101| TID=3 | PC=3059 | IE=956
SYSCALL_PRN VALUE=39| TID=1 | PC=1005 | IE=1069
SYSCALL_PRN VALUE=36| TID=2 | PC=2005 | IE=1179
SYSCALL_PRN VALUE=102| TID=3 | PC=3059 | IE=1301
SYSCALL_PRN VALUE=50| TID=1 | PC=1005 | IE=1414
SYSCALL_PRN VALUE=46| TID=2 | PC=2005 | IE=1524
SYSCALL_PRN VALUE=103| TID=3 | PC=3059 | IE=1646
SYSCALL_PRN VALUE=60| TID=1 | PC=1005 | IE=1759
SYSCALL_PRN VALUE=55| TID=2 | PC=2005 | IE=1869
SYSCALL_PRN VALUE=104| TID=3 | PC=3059 | IE=1991
SYSCALL_PRN VALUE=69| TID=1 | PC=1005 | IE=2104
SYSCALL_PRN VALUE=63| TID=2 | PC=2005 | IE=2214
SYSCALL_PRN VALUE=77| TID=1 | PC=1005 | IE=2430
SYSCALL_PRN VALUE=70| TID=2 | PC=2005 | IE=2528
SYSCALL_PRN VALUE=84| TID=1 | PC=1005 | IE=2669
SYSCALL_PRN VALUE=76| TID=2 | PC=2005 | IE=2767
SYSCALL_PRN VALUE=90| TID=1 | PC=1005 | IE=2908
SYSCALL_PRN VALUE=81| TID=2 | PC=2005 | IE=3006
SYSCALL_PRN VALUE=95| TID=1 | PC=1005 | IE=3147
SYSCALL_PRN VALUE=85| TID=2 | PC=2005 | IE=3245
SYSCALL_PRN VALUE=99| TID=1 | PC=1005 | IE=3386
SYSCALL_PRN VALUE=88| TID=2 | PC=2005 | IE=3484
SYSCALL_PRN VALUE=102| TID=1 | PC=1005 | IE=3625
SYSCALL_PRN VALUE=90| TID=2 | PC=2005 | IE=3723
SYSCALL_PRN VALUE=104| TID=1 | PC=1005 | IE=3864
SYSCALL_PRN VALUE=91| TID=2 | PC=2005 | IE=3962
SYSCALL_PRN VALUE=105| TID=1 | PC=1005 | IE=4103
```

Figure 4: .

```
#----------------------------------------------------
    # THREAD 1 KOMUTLARI (Baslangic: 1000) - Basitlestirilmis Hali
    #----------------------------------------------------
1000: SET 111, 1050      # Ilk degeri ayarla
1001: SYSCALL_PRN 1050   # Ilk degeri yazdir (111 bekleniyor)
1002: SYSCALL_YIELD      # CPU'yu OS'e birak
1003: SET 112, 1051      # YIELD sonrasi ikinci degeri ayarla
1004: SYSCALL_PRN 1051   # Ikinci degeri yazdir (112 bekleniyor)
1005: SYSCALL_HLT        # Thread 1'i sonlandir

#----------------------------------------------------
    # THREAD 2 KOMUTLARI (Baslangic: 2000) - Basitlestirilmis Hali
    #----------------------------------------------------
2000: SET 221, 2050      # Ilk degeri ayarla
2001: SYSCALL_PRN 2050   # Ilk degeri yazdir (221 bekleniyor)
2002: SYSCALL_YIELD      # CPU'yu OS'e birak
2003: SET 222, 2051      # YIELD sonrasi ikinci degeri ayarla
2004: SYSCALL_PRN 2051   # Ikinci degeri yazdir (222 bekleniyor)
2005: SYSCALL_HLT        # Thread 2'yi sonlandir
```

Figure 5: "context switch sıklığı daha yoğun

```
simulator > os_plus_threads.gtu
#----------------------------------------------------
# --- THREAD 3 KOMUTLARI (Başlangıç: 3000) - DÜZELTİLMİŞ Bubble Sort ---
3000: CPY  3550, 3563    # n = N (5)
3001: ADD  3563, -1      # n-1 = 4
3002: SET  0, 3560       # i = 0

# --- DIŞ DÖNGÜ BAŞLANGICI (i döngüsü) ---
3003: CPY  3560, 3564    # temp_i = i
3004: CPY  3563, 3565    # temp_n1 = N-1
3005: SUBI 3565, 3564    # temp_comparison = (N-1)-i ilk - ikinci sonra ikincie yaz
3006: JIF  3564, 3050    # Eğer comparison <= 0 ise devam, değilse yazdırmaya git
3007: ADD  3564, -1       # temp_comparison -1
3008: SET  0, 3561       # j = 0

# --- İÇ DÖNGÜ BAŞLANGICI (j döngüsü) ---
3009: CPY  3561, 3566    # temp_j = j
3010: CPY  3563, 3567    # temp_n1 = N-1
3011: CPY  3560, 3568    # temp_i = i
3012: SUBI 3567, 3568    # limit = (N-1) - i
3013: SUBI 3568, 3566    # comparison = limit -j
3014: JIF  3566, 3040     # Eğer (limit - j) ≤ 0 ise, i++
3015: ADD  3566, -1       # comparison - 1


# --- DİZİ ELEMAN ADRESLERİNİ HESAPLA ---
3016: SET  3551, 3565    # base = 3051 (array başlangıç adresi)
3017: CPY  3561, 3566    # temp_j = j
3018: ADDI 3565, 3566    # array[j] adresi = base + j
3019: CPY  3565, 3567    # array[j] adresini sakla
3020: ADD  3567, 1       # array[j+1] adresi = array[j] adresi + 1

# --- DİZİ ELEMANLARININ DEĞERLERİNİ AL ---
3021: CPYI 3565, 3568    # 3068 = array[j] değeri
3022: CPYI 3567, 3569    # 3069 = array[j+1] değeri

# --- KARŞILAŞTIRMA: array[j] > array[j+1] MI? ---
3023: CPY  3568, 3562    # temp = array[j]
3024: SUBI 3562, 3569    # temp = array[j] - array[j+1]
3025: JIF  3562, 3035    # if (array[j] <= array[j+1]) ise j++ yap
```

```
cpu_simulator > os_plus_threads.gtu
3026: CPY  3568, 3562    # temp = array[j] değeri
3027: SET  3569, 3580    # Store address containing array[j+1] value
3028: SET  3562, 3581    # Store address containing temp value
3029: CPY 3580, 3565   # Copy from address pointed by 3570 to address pointed by 3565
3030: CPY 3581, 3567   # Copy from address pointed by 3571 to address pointed by 3567

# --- T ---
3031: SET  0, 3600
3032: JIF 3600, 3035

# --- J'Yİ 1 ARTTIR ve İÇ DÖNGÜYE DÖN (DÜZELTİLMİŞ) ---
3035: ADD  3561, 1       # j++
3036: JIF  3600, 3009       # PC = 3008'e git (döngü başı) ## düzelt

# --- İ DÖNGÜSÜ İÇİN: i++ ve DIŞ DÖNGÜ BAŞINA DÖN (DÜZELTİLMİŞ) ---
3040: ADD  3560, 1       # i++
3041: JIF  3600, 3003       # PC = 3003'e git (dış döngü başı) ## düzelt

# --- SONUÇLARI YAZDIRMA AŞAMASI ---
3050: SET  0, 3580       # print_index = 0

# --- YAZDIRMA DÖNGÜSÜ (DÜZELTİLMİŞ) ---
3051: CPY  3580, 3581    # temp_index = print_index
3052: CPY  3550, 3582    # temp_N = N
3053: SUBI 3582, 3581    # comparison = n- print_index
3054: JIF  3581, 3065    # if (n-print_index <=0) ise HALT

# --- DİZİ ELEMANINI YAZDIR ---
3055: SET  3551, 3581    # base_addr = 3551
3056: CPY  3580, 3582    # temp_index = print_index
3057: ADDI 3581, 3582    # element_addr = base_addr + print_index
3058: CPYI 3581, 3583    # element_value = array[print_index]
3059: SYSCALL_PRN 3583   # ekrana yazdır

# --- YAZDIRMA İNDEXİNİ ARTTIR (DÜZELTİLMİŞ) ---
3060: ADD  3580, 1       # print_index++
3061: JIF  3600, 3051       # PC = 3051'e git (yazdırma döngüsü başı)

# --- PROGRAM SONU ---
3065: SYSCALL_HLT        # Thread'i sonlandır
```

Figure 6:

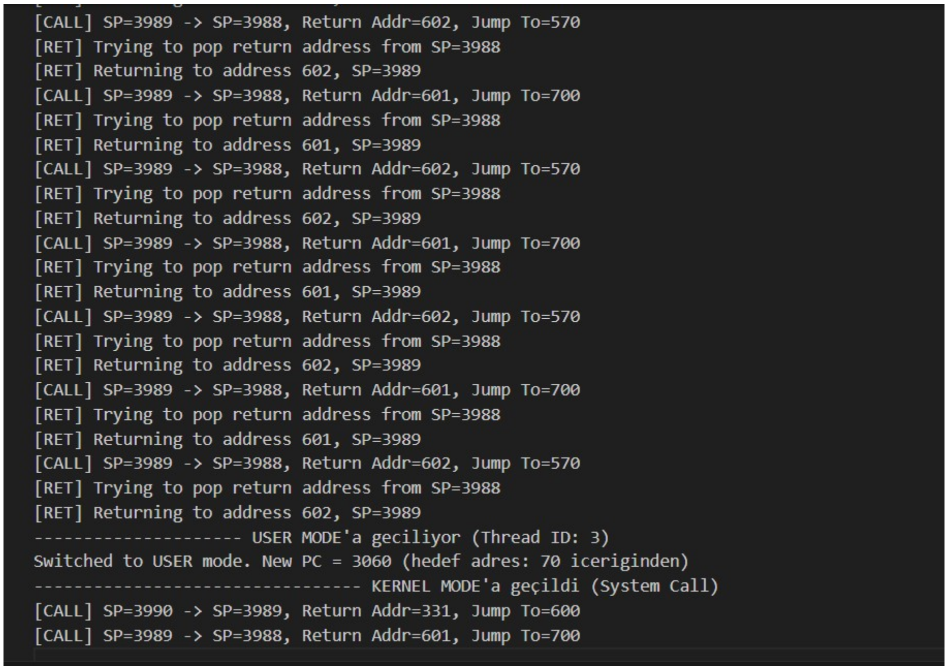

Figure 7: Instruction log çıktısı: çalıştırılan tüm komutlar.

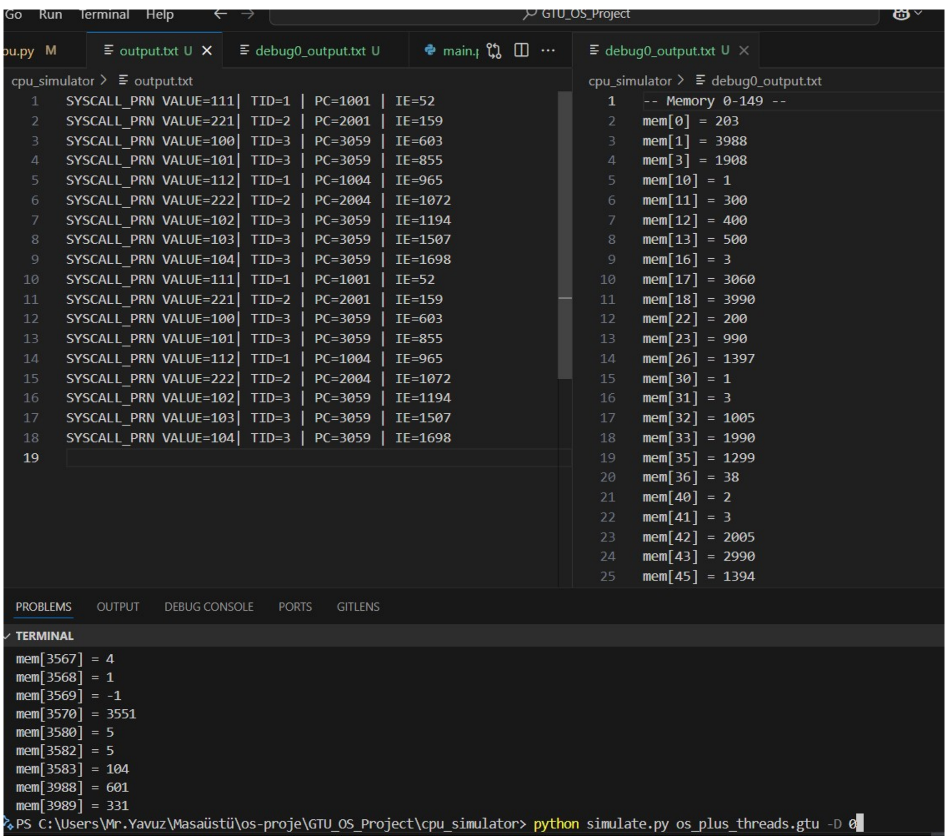

Figure 8: Debug Mode 0: Tüm bellek içeriği.

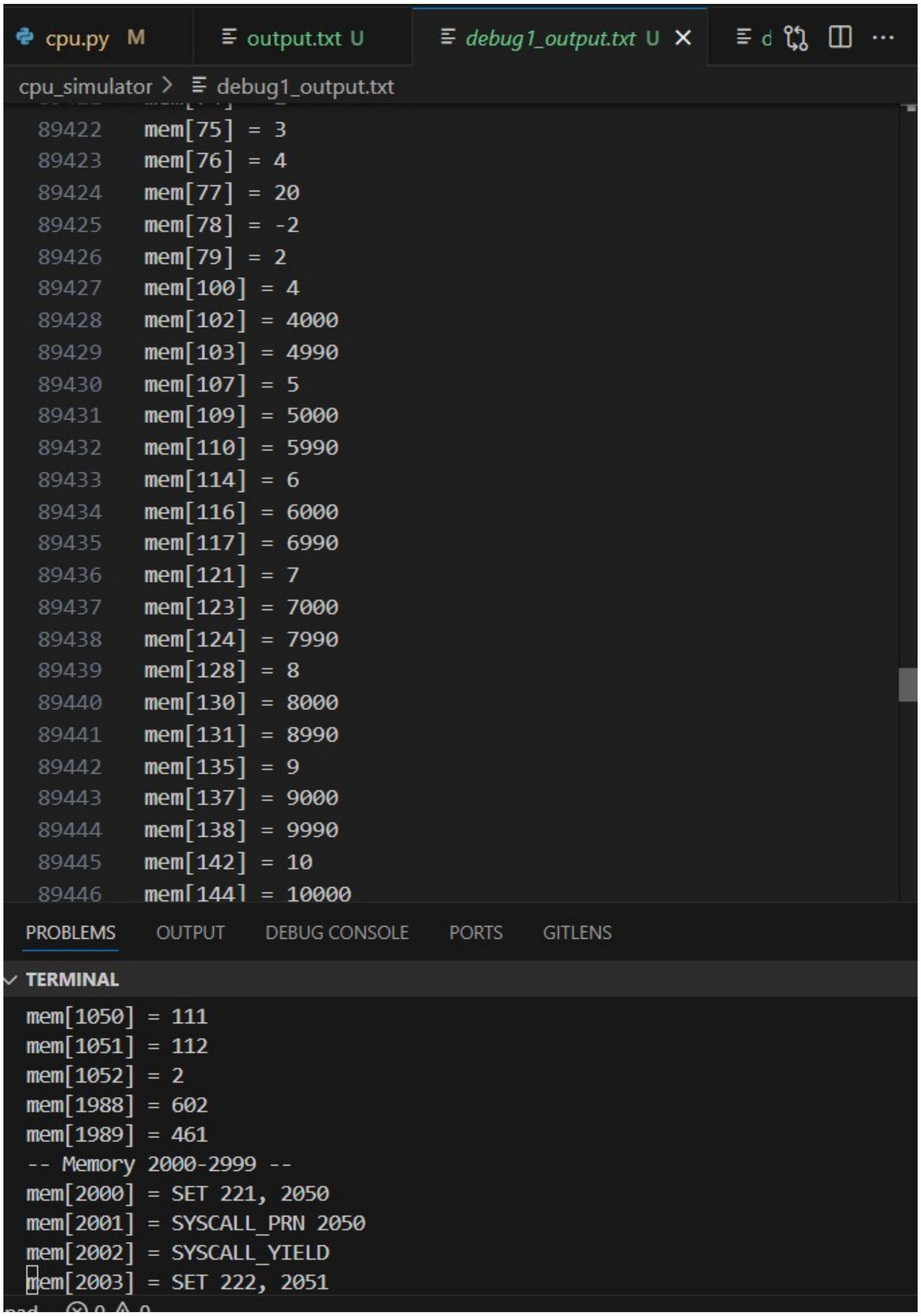

Figure 9: Debug Mode 1: Değişen bellek bölgeleri.

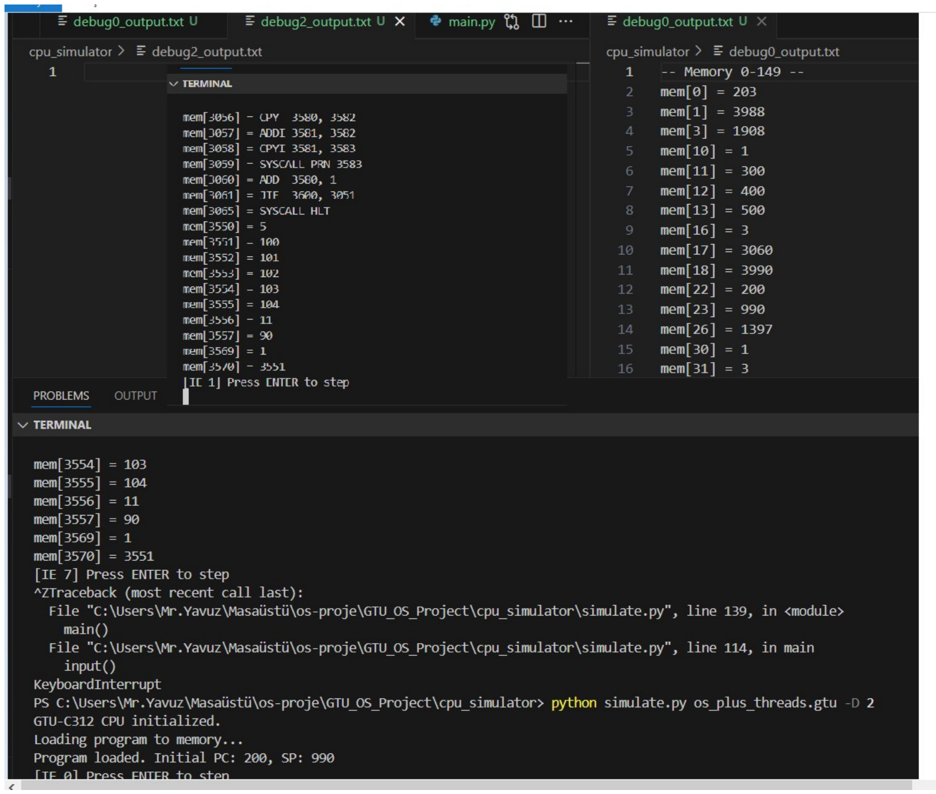

Figure 10: Debug Mode 2: Her adım sonrası kullanıcı onaylı yürütme.

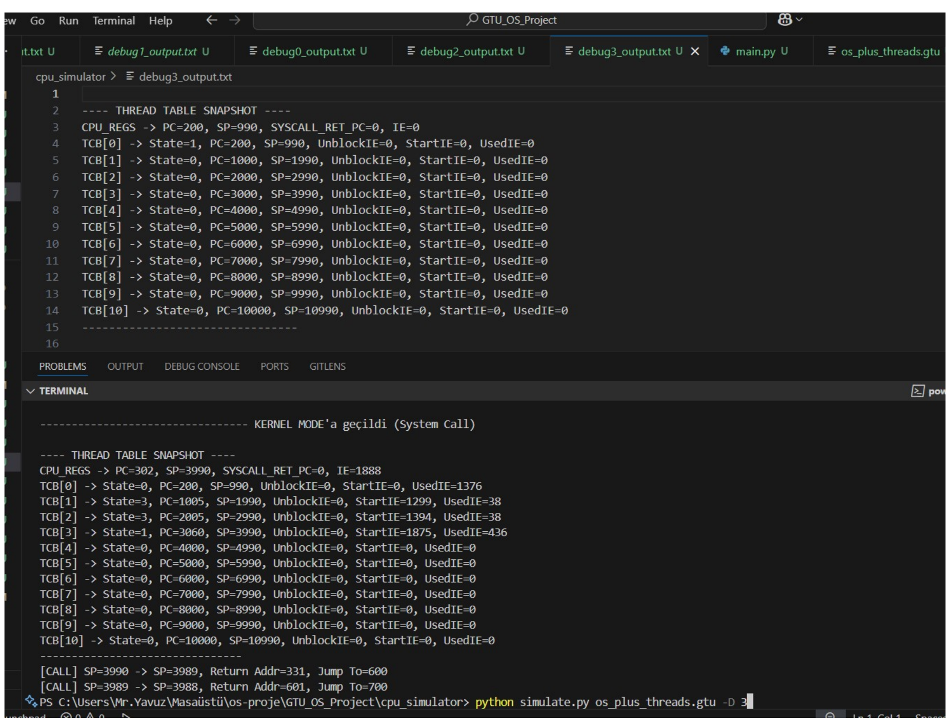

Figure 11: Debug Mode 3: Her context switch sonrası TCB görüntüsü.