CSE312 -- Semester Project
05/06/2025

Cemal BOLAT
STUDENT NO: 210104004010

Introduction:

This project aims to simulate a simple computer system by designing a custom instruction set architecture (ISA) and implementing a minimal operating system (OS) that can manage multiple user-level threads. The system is built around a hypothetical CPU called GTU-C312, which operates with a unique assembly-like instruction set and a memory-based register model. The project involves developing a CPU simulator in Python, writing an operating system using the GTU-C312 instruction language, and executing concurrent threads that perform algorithmic tasks such as sorting and searching. Through this simulation, we explore low-level OS concepts including thread management, cooperative multitasking, system calls, and memory protection, thereby gaining hands-on experience in both systems programming and CPU simulation.

Os Structure:

The CPU for this project was fully implemented in Python, and the memory model was explicitly divided into two separate spaces: data memory and instruction memory. This architectural decision simplified the simulator logic by eliminating any need to cross-check whether a memory access belonged to data or instruction context during execution.

A critical design feature of the OS is the careful and structured memory layout between addresses 0 and 999. The memory layout is segmented into clearly defined blocks for registers, OS variables, and thread metadata, as depicted in the accompanying memory design diagram.

Memory Layout:

- **Registers (0–20):**
  This region includes essential CPU registers such as
  the Program Counter (PC) at address 0,
  the Stack Pointer (SP) at address 1,
  the SysCall Result register at 2, and an
  Addresses 3–20 are used as general-purpose or temporary registers for flexible instruction handling.
- **OS Variables (21–49):**
  - This section is heavily utilized by the OS and includes:
  - Current TID, Active Threads, and Last Executed TID to manage scheduling.
  - Multiple temporary registers for handling complex instructions such as CPYI2 and USER transitions.
  - System-wide counters such as Block Counter used during system calls like PRN.

- **Special OS Utility addresses continue.. (990–999):**

Address	Purpose
993	Temporary register used to track **instruction execution counts** during operations such as scheduling for instruction that controls by OS.
994	Holds the target jump address used during the execution of the **USER instruction**, enabling mode transitions.
995	Acts as a return address register (**similar to $RA in MIPS**) to store instruction addresses before jumping in OS.
996	Permanently stores the value 0, useful for **jump conditions (e.g., JIF)** and comparison logic without needing to set it manually.
997	Temporarily holds the **Thread ID (TID)** of the currently executing thread. This is useful during context switches.
998	Keeps track of the **instruction segment base address** for the next thread, assisting the OS in locating and scheduling threads dynamically.
999	Serves as the **global clock** tracking the total number of executed instructions since boot. It is used for implementing delays, particularly for **blocking behavior in SYSCALL PRN**, where threads must yield for 100 cycles.

- **Thread Table (50–459)**:

    - Each thread is assigned a block of 40 memory cells for its metadata and registers

- **Syscall Handlers:**

System call entry points are also reserved in the memory:

- SYSCALL PRN A: Address 490
- SYSCALL HLT : Address 505
- SYSCALL YIELD: Address 515
- HLT: Address 480e

OS Structure

Memory Design
Data & Instruction Memories different

Datas Locations In OS

START	END	Description
0	20	Registers
21	49	Os variables
50	89	Thread #1
90	129	Thread #2
130	169	Thread #3
170	209	Thread #4
210	249	Thread #5
250	289	Thread #6
290	329	Thread #7
330	369	Thread #8
370	409	Thread #9
410	459	Thread #10

Registers

0	PC
1	SP
2	SysCall Result
3-20	User Choice

Thread Table Design (40 byte offset)

Base	Thread ID
Base + 1	Start Time
Base + 2	Instruction Count
Base + 3	Thread State
Base + 4	Thread PC
Base + 6	Block Counter
Base + 10-30	Thread Registers
Base + 10-30	Thread Registers

Thread State Meanings

0	Ready
1	Running
2	Halted
3	Blocked

OS Variables

21	Current TID
22	Active Threads
23	Table Offset
24	Last Executed TID
25	Next TID
26	Temp register for CPYI2 INSTR
27	Temp register for CPYI2 INSTR
30	Temp register
993	Temp For Exec Count
994	Temp register for USER INSTR
995	Temp register like $RA
996	Always 0, easy to use JIF
997	Temp for current TID
998	Table offset for next INSTR
999	Current Time for sleep in SYSCALL PRN

Syscall Handlers location

490	SYSCALL PRN A
505	SYSCALL HLT
515	SYSCALL YIELD
480	HLT

Threads:

Thread 1 – Sorting in Increasing Order

This thread implements a bubble sort algorithm to sort N = 10 elements starting from memory address 1001. The array contains positive, zero, and negative numbers. Temporary variables for the sorting process such as current and next element addresses, temporary values, and loop counters—are stored between addresses 1100 and 1125. The sorting is done by repeatedly comparing adjacent elements and swapping them if they are in the wrong order. Once sorted, the array is printed using SYSCALL PRN.

Data Segment:

```
# First threads data - Sorting in increasing order
1000 10        # N
1001 4         # arr[0]
1002 0        # arr[1]
1003 -1         # arr[2]
1004 2        # arr[3]
1005 3         # arr[4]
1006 4        # arr[5]
1007 5         # arr[6]
1008 8         # arr[7]
1009 9        # arr[8]
1010 -222      # arr[9]

1100 0   # i
1101 0   # j
1102 0   # addr_curr
1103 0   # addr_next
1104 0   # val_curr
1105 0   # val_next/tmp
1111 0   # tmp_val
1112 0   # src_ptr
1113 0   # dst_ptr
1121 0   # temp
1122 0   # temp
1123 0   # temp
1124 0   # temp
1125 0   # temp
```

Way That I implement

- The thread begins by yielding to allow proper scheduling by the OS. Initializes loop variables `i = 0`, `j = 0`, and prepares constants like zero (`0`) in address `1132` for use in `JIF` instructions.
- Enters the **outer loop**, which iterates from `i = 0` to `N - 1`.
- Within each outer iteration, the thread enters the **inner loop**, where it compares adjacent elements in the array.
- For each pair `(arr[j], arr[j+1])`, it:
  - Computes their addresses,
  - Reads their values using indirect memory access,
  - Compares the two values to decide if a swap is needed.
- If the values are out of order (`arr[j] > arr[j+1]`), it performs a **swap** using CPYI2 instructions** and temporary pointers.
- If no swap is needed, it simply increments `j` and continues the inner loop.
- After the inner loop is completed, it increments `i` and re-enters the outer loop.
- Once sorting is complete, it resets `j = 0` and enters a **print loop**.
- Each sorted array element is printed using `SYSCALL PRN` until all `N` elements are printed.
- Finally, the thread terminates with `SYSCALL HLT`.

Thread 2 – Linear Search for a Key

This thread performs a linear search to find the index of a specific key in an array. The key (10) and the array of 7 elements are defined starting at address 2050. The thread iterates through the array using a counter starting from 2000 and compares each element with the key. If a match is found, it returns the index via SYSCALL PRN; otherwise, it returns -1. The thread terminates with SYSCALL HLT.

Data Segment

```
# Second thread data - Searching for a key in an array
# Return the index of the key if found, otherwise return -1
2000 0       # counter (index)
2001 7       # number of elements
2002 2050    # array start address
2003 10      # key to search
2050 12      # First element
2051 13      # Second element
2052 14      # Third element
2053 15      # Fourth element
2054 17      # Fifth element
2055 18      # Sixth element
2056 19      # Seventh element
```

Thread 3 – Printing Numbers

This thread is designed to sequentially print a list of numbers. The array is located at 3050, and it contains 10 integers. A loop controlled by a counter at address 3000 is used to traverse and print each value one by one using SYSCALL PRN. This thread is mostly used for I/O testing and validating thread isolation and correct array access in user mode.

Data Segment:

```
# Third thread data - Printing numbers
3000 0      # counter
3001 10     # number of elements to print
3002 3050   # array start address
3050 21      # First number
3051 22      # Second number
3052 23      # Third number
3053 24      # Fourth number
3054 25      # Fifth number
3055 26     # Sixth number
3056 27     # Seventh number
3057 28     # Eighth number
3058 21     # Ninth number
3059 20     # Tenth number
```

Thread 4 – Multiplication Using Repeated Addition

This thread calculates the result of multiplying two positive integers using repeated addition. The multiplicand (6) and multiplier (5) are located at 4000 and 4001, and the result is stored at 4002. The thread adds the multiplicand to the result repeatedly, based on the multiplier value, simulating 6 * 5 = 30. This thread demonstrates loop constructs and arithmetic computation without using a dedicated MUL instruction.

Thread 5 – Sum Until Zero

This thread adds all integers from i = 9 down to 1. It uses a simple decrement loop, storing the sum in-place. The purpose of this thread is to test indirect memory operations (ADDI, SUBI) and control flow (JIF). Once the sum is computed, it is printed via SYSCALL PRN.

Thread 6 – Addition Using CALL/RET
This thread tests function call and return instructions. It stores two values i = 17 and j = 25 and computes their sum by calling a subroutine that performs the addition. The result is stored at address 6002. It validates the correct behavior of CALL, RET, and stack-based return address management.
Also checking for SP storage properly.

Thread 7 – Stack Operations Using PUSH/POP
This thread evaluates the correctness of stack operations. It begins with a value i = 60, pushes it onto the stack, performs unrelated computations, and then pops the value back to verify that stack-based temporary storage works as expected. It uses memory addresses 7000 and 7001 for variables. Also checking for SP storage properly.
Thread 8 – Memory Violation
This thread just has one instruction which violates and tries to read OS specified register.
Thread 5-6-7's data segment.

```
# Fourth thread data - Positive Multiply function
4000 6     # muliplicand (base)
4001 5      # multiplier (exponent)
4002 0      # result (initially 1) (30)

# Fifth thread data - Sum till 0
#5000 10     # i (10 + 9 + 8 + ... + 1)
5000 9       # i (8 + 7 + 6 + ... + 1) : 36

# Sixth thread data - Usage of CALL and RET functions
6000 17     # i
6001 25     # j
6002 0      # sum = i + j (init it as 0)

# Seventh thread data - Usage of PUSH and POP functions
7000 60     # i
7001 0      # sum = 0 (init it as 0)

```

Thread 1: Test Case – Sorting as ascending order

```
Program loaded from ./sep_thread/1.txt
PRN: -222
PRN: -1
PRN: 0
PRN: 2
PRN: 3
PRN: 4
PRN: 4
PRN: 5
PRN: 8
PRN: 9
Halting CPU.
Final Memory State:
```

```
# =========== END FOR THREAD TABLE 490

# First threads data - Sorting in increasing order
1000 10        # N
1001 4         # arr[0]
1002 0        # arr[1]
1003 -1         # arr[2]
1004 2        # arr[3]
1005 3         # arr[4]
1006 4        # arr[5]
1007 5         # arr[6]
1008 8         # arr[7]
1009 9        # arr[8]
1010 -222      # arr[9]

1100 0   # i
1101 0   # j
1102 0   # addr_curr
1103 0   # addr_next
1104 0   # val_curr
1105 0   # val_next/tmp
1111 0   # tmp_val
1112 0   # src_ptr
1113 0   # dst_ptr
1121 0   # temp
1122 0   # temp
1123 0   # temp
1124 0   # temp
1125 0   # temp
```

Case 2:

```
Program loaded from ./sep_thread/1.txt
PRN: -222
PRN: -1
PRN: 9
PRN: 28
PRN: 37
PRN: 49
PRN: 55
PRN: 64
PRN: 92
PRN: 780
Halting CPU.
Final Memory State:
```

```
# ========== END FOR THREAD TABLE 490

# First threads data - Sorting in increasing order
1000 10        # N
1001 49         # arr[0]
1002 780        # arr[1]
1003 -1         # arr[2]
1004 92        # arr[3]
1005 37         # arr[4]
1006 64        # arr[5]
1007 55         # arr[6]
1008 28         # arr[7]
1009 9        # arr[8]
1010 -222      # arr[9]
```

Thread 2: Test Case – Linear Search

Key in 4th index

```
=== END OF MEMORY STATE ===
GTU-C312 CPU initialized.
Program loaded from ./sep_thread/2.txt
PRN: 4
Halting CPU.
Final Memory State:
```

```
# Return the index of the key if found, otherwise return -1
2000 0        # counter (index)
2001 7        # number of elements
2002 2050     # array start address
2003 17       # key to search
2050 12       # First element
2051 13       # Second element
2052 14       # Third element
2053 15       # Fourth element
2054 17       # Fifth element
2055 18       # Sixth element
```

Key not found

```
=== END OF MEMORY STATE ===
GTU-C312 CPU initialized.
Program loaded from ./sep_thread/2.txt
PRN: -1
Halting CPU.
Final Memory State:
```

```
# Second thread data - Searching for a key in an array
# Return the index of the key if found, otherwise return -1
2000 0        # counter (index)
2001 7        # number of elements
2002 2050     # array start address
2003 23       # key to search
2050 12       # First element
2051 13       # Second element
2052 14       # Third element
2053 15       # Fourth element
2054 17       # Fifth element
2055 18       # Sixth element
2056 19       # Seventh element
```

Key in index 0

```
Program loaded from ./sep_thread/2.txt
PRN: 0
Halting CPU.
Final Memory State:
```

```
# Second thread data - Searching for a key in an array
# Return the index of the key if found, otherwise return -1
2000 0        # counter (index)
2001 7        # number of elements
2002 2050     # array start address
2003 12       # key to search
2050 12       # First element
2051 13       # Second element
2052 14       # Third element
2053 15       # Fourth element
2054 17       # Fifth element
2055 18       # Sixth element
2056 19       # Seventh element
```

Thread 3: Printing array

```
=== END OF MEMORY STATE ===
GTU-C312 CPU initialized.
Program loaded from ./sep_thread/3.txt
PRN: 21
PRN: 22
PRN: 23
PRN: 24
PRN: 25
PRN: 26
PRN: 27
PRN: 28
PRN: 21
PRN: 20
PRN: 0
Halting CPU.
Final Memory State:
```

```
# Third thread data - Printing numbers
3000 0        # counter
3001 10       # number of elements to print
3002 3050     # array start address
3050 21        # First number
3051 22        # Second number
3052 23        # Third number
3053 24        # Fourth number
3054 25        # Fifth number
3055 26       # Sixth number
3056 27       # Seventh number
3057 28       # Eighth number
3058 21       # Ninth number
3059 20       # Tenth number
```

Thread 4: Positive Multiply function with usaging ADDI

```
GTU-C312 CPU initialized.
Program loaded from ./sep_thread/4.txt
PRN: 30
Halting CPU.
Final Memory State:
```

```
# Fourth thread data - Positive Multiply function
4000 6      # muliplicand (base)
4001 5       # multiplier (exponent)
4002 0       # result (initially 1) (30)
```

Case 2: -- 16 * 53 = 848

```
GTU-C312 CPU initialized.
Program loaded from ./sep_thread/4.txt
PRN: 848
Halting CPU.
Final Memory State:
```

```
sep_thread > 4.txt
# Fourth thread data - Positive Multiply function
4000 16      # muliplicand (base)
4001 53       # multiplier (exponent)
4002 0       # result (initially 1) (30)
```

Thread 5: PDF example

```
# Fourth thread data - Positive Multiply function
4000 6      # muliplicand (base)
4001 5       # multiplier (exponent)
4002 0       # result (initially 1) (30)

# Fifth thread data - Sum till 0
#5000 10      # i (10 + 9 + 8 + ... + 1)
5000 9        # i (8 + 7 + 6 + ... + 1) : 36

# Sixth thread data - Usage of CALL and RET functions
6000 17      # i
6001 25      # j
6002 0       # sum = i + j (init it as 0)

# Seventh thread data - Usage of PUSH and POP functions
7000 60      # i
7001 0       # sum = 0 (init it as 0)
```

```
Whole Instruction Memory without empty strings:
=== END OF MEMORY STATE ===
GTU-C312 CPU initialized.
Program loaded from ./sep_thread/5.txt
PRN: 9
PRN: 17
PRN: 24
PRN: 30
PRN: 35
PRN: 39
PRN: 42
PRN: 44
PRN: 45
Halting CPU.
Final Memory State:
```

Thread 6: Usage of CALL & RET and as you can see sp is stored properly.

```
# Fifth thread data - Sum till 0
#5000 10      # i (10 + 9 + 8 + ... + 1)
5000 9        # i (8 + 7 + 6 + ... + 1) : 36

# Sixth thread data - Usage of CALL and RET functions
6000 17      # i
6001 25      # j
6002 0       # sum = i + j (init it as 0)

# Seventh thread data - Usage of PUSH and POP functions
```

```
=== END OF MEMORY STATE ===
GTU-C312 CPU initialized.
Program loaded from ./sep_thread/6.txt
PRN: 6900
PRN: 6899
PRN: 6899
PRN: 42
Halting CPU.
Final Memory State:
```

Thread 7: Usage of PUSH & POP (sum = [7000] + 11) sp is stored properly too.

```
# Sixth thread data - Usage of CALL and RET functions
6000 17      # i
6001 25      # j
6002 0       # sum = i + j (init it as 0)

# Seventh thread data - Usage of PUSH and POP functions
7000 60      # i
7001 0       # sum = 0 (init it as 0)
```

```
GTU-C312 CPU initialized.
Program loaded from ./sep_thread/7.txt
PRN: 7899
PRN: 7898
PRN: 11
PRN: 7899
PRN: 60
PRN: 7900
PRN: 71
Halting CPU.
```

Thread 8: Memory Violation test

```
# Thread 8 - Memory Violation Test
8000 SYSCALL YIELD
8002 CPY 8000 105         # Copy i to register 105 (Memory Violation)
##### Instruction section
9001 SYSCALL HLT

10001 SYSCALL HLT

End Instruction Section
```

```
Whole Instruction Memory without empty strings:
=== END OF MEMORY STATE ===
GTU-C312 CPU initialized.
Program loaded from ./sep_thread/8.txt
Memory access violation in user mode: 105. Thread ID: 8. PC: 8002
Halting CPU.
Final Memory State:
```

ChatGPT links:

AI-Based Assistance Acknowledgment:

During the development of this project, AI-assisted tools were used for support in both the Python-based CPU simulator and the GTU-C312 assembly-level operating system code. While many interactions were performed through GitHub Copilot—which does not retain chat logs—a subset of conversations with ChatGPT were documented and are included below. These interactions contributed to clarifying memory layout decisions, designing system call behaviors, and debugging thread-level scheduling and sorting logic.

Report Creation
Bug Fixing (sp was not saving properly)
Bubble Sort
Bubble Sort (v2) (bug fixing)
Assembly (MIPS) & C version of bubble sort
Assembly Bubble Sort (v2)
CPP to Python Convertion (for easy to use in different enviroment).