GTU-C312 Operating System Project Report

June 6, 2025

1 Introduction

This report presents the implementation of a simple operating system (OS) for the GTUC312 CPU, a hypothetical processor designed for the CSE 312 / CSE 504 Operating Systems course at Gebze Technical University. The project aims to simulate a CPU with a custom instruction set architecture (ISA) and develop an OS using this ISA to manage multiple threads. The OS supports thread scheduling, system calls, and memory management, with three user-defined threads performing bubble sort, linear search, and Fibonacci sequence generation. The project demonstrates skills in CPU design, OS implementation, thread management, and systems simulation in Python, adhering to the requirements outlined in the project specification.

2 Project Overview

The GTU-C312 CPU simulator is implemented in Python, providing a platform to execute programs written in the GTU-C312 assembly-like language. The OS, written in this language, manages up to 10 threads using a round-robin scheduler and handles system calls for printing, yielding, and terminating threads. The project includes a CPU simulator class, an OS program, and three threads with distinct functionalities. The simulator supports multiple debug modes to monitor memory and thread states, facilitating analysis of program execution. This report details the CPU simulator, OS structure, thread implementations, and simulation process.

3 CPU Simulator Design

The GTU-C312 CPU simulator is encapsulated in the GTUC312CPU class, which manages a memory array of 20,000 signed long integers. Special memory locations serve as registers: address 0 for the program counter (PC), address 1 for the stack pointer, address 2 for system call results, and address 3 for the instruction count. Addresses 420 are reserved for temporary use. The simulator supports the following instructions:

- SET B A: Sets value B into memory address A.
- CPY A1 A2: Copies the content of address A1 to A2.
- CPYI A1 A2: Copies the content of the address stored at A1 to A2.
- CPYI2 A1 A2: Copies the content of the address stored at A1 to the address stored at A2.
- ADD A B: Adds value B to address A.

- ADDI A1 A2: Adds the content of A2 to A1.
- SUBI A1 A2: Subtracts the content of A2 from A1, storing the result in A2.
- JIF A C: Jumps to instruction C if the content of A is less than or equal to 0.
- PUSH A: Pushes the content of A onto the stack.
- POP A: Pops a value from the stack into A.
- CALL C: Calls a subroutine at instruction C, pushing the return address.
- RET: Returns from a subroutine by popping the return address.
- HLT: Halts the CPU.
- USER A: Switches to user mode and jumps to the address stored at A.
- SYSCALL PRN A: Prints the content of address A and blocks for 100 instructions.
- SYSCALL HLT: Terminates the current thread.
- SYSCALL YIELD: Yields the CPU for thread scheduling.

The simulator loads a program file (e.g., os.txt) containing data and instruction sections. The load_program method parses the file, storing data in the memory array and instructions in a dictionary keyed by their addresses. The execute method processes one instruction per cycle, updating the PC and instruction count. It enforces memory access restrictions: in kernel mode, all addresses are accessible; in user mode, only addresses 1000 and above are allowed, with violations causing thread termination. Debug modes (03) provide varying levels of output, including memory dumps and thread table information, as specified in the project requirements.

4 Operating System Structure

The OS is implemented in GTU-C312 assembly language within the os.txt file, starting at instruction address 100. It initializes a thread table at memory addresses 3089, supporting up to 10 threads. Each thread entry occupies 10 memory locations, storing the thread ID, state (0=ready, 1=running, 2=blocked), program counter, stack pointer, and additional registers. The OS itself is treated as thread 0, operating in kernel mode with full memory access.

The OS begins by setting the program counter for thread 1 (address 1000) and executing a USER instruction to switch to user mode and jump to thread 1s starting address. Thread scheduling is cooperative, relying on SYSCALL YIELD to trigger context switches. The ThreadManager class in Python handles scheduling by saving the current threads state (PC and stack pointer) and selecting the next ready thread in a round-robin order. System calls are processed as follows:

- SYSCALL PRN A: Prints the value at address A to the console and blocks the thread for 100 instruction cycles.

- SYSCALL HLT: Terminates the current thread, removing it from the thread table and scheduling the next thread.
- SYSCALL YIELD: Saves the current threads state and switches to the next ready thread.

The OS reserves memory addresses 21-999 for its data and instructions, ensuring usermode threads cannot access this region. The thread table is initialized with data for three active threads, with their program counters set to 1000, 2000, and 3000, respectively.

5 Thread Implementations

The OS manages three threads, each allocated a memory region for data and instructions, as specified in the project requirements.

5.1 Thread 1: Bubble Sort

Thread 1 operates in the memory range 1000-1999, performing a bubble sort on an array of five numbers (64, 34, 25, 12, 22) stored at addresses 10111015, with the array size (N=5) at address 1010. The algorithm executes four passes, each comparing adjacent elements and swapping them if they are in descending order. The implementation uses CPY to load values, SUBI to compare them, JIF to skip swaps if unnecessary, and a temporary variable at address 1090 for swapping. After each pass, SYSCALL YIELD is called to allow other threads to execute. Upon completion, the sorted array is printed using SYSCALL PRN for each element, and the thread terminates with SYSCALL HLT.

5.2 Thread 2: Linear Search

Thread 2, located at addresses 2000-2999, performs a linear search for a key (25) in an array of five numbers (64, 34, 25, 12, 22) stored at addresses 20122016, with the array size (N=5) at address 2010 and the key at address 2011. The thread sequentially compares each element to the key using CPY, SUBI, and JIF instructions, checking both directions (element - key and key - element) to confirm equality. The result (index or -1 if not found) is stored at address 2017. After the search, the result is printed using SYSCALL PRN, and the thread terminates with SYSCALL HLT.

5.3 Thread 3: Fibonacci Sequence

Thread 3, located at addresses 3000-3999, generates the first 10 numbers of the Fibonacci sequence. The count (10) is stored at address 3010, with variables for F(n-2), F(n-1), and F(n) at addresses 30113013, and a counter at address 3014. The thread initializes F(0)=0 and F(1)=1, printing them using SYSCALL PRN. It then enters a loop to compute the next eight numbers, using ADDI to add F(n-2) and F(n-1), storing the result in F(n), and updating the variables. Each number is printed, and the thread terminates with SYSCALL HLT after generating all 10 numbers.

6 Thread Management and Scheduling

The ThreadManager class manages the thread table, initialized from memory addresses 3089. Each threads entry includes its ID, state, program counter, stack pointer, start time, and instruction count. The setup_threads method populates the thread table by reading thread data from memory, setting thread 0 (OS) as running and others as ready. The yield_cpu method handles SYSCALL YIELD by saving the current threads state and selecting the next ready thread in a round-robin order. The halt_current_thread method removes a terminated thread and schedules the next one. Context switches restore the program counter and stack pointer from the thread table, and the update_thread_stats method tracks each threads instruction count.

7 Simulation and Debugging

The simulation is executed via the cpu.py script, which accepts a program file and a debug flag (-D 0, 1, 2, or 3). The main function initializes the CPU and thread manager, loads the program, and runs the execution loop until the CPU halts. Debug modes provide the following outputs:

- **Debug Mode 0**: Prints non-zero memory locations to the standard error stream after the CPU halts.
- **Debug Mode 1**: Prints non-zero memory locations after each instruction execution.
- **Debug Mode 2**: Prints the current instruction and memory state after each execution, pausing for user input.
- **Debug Mode 3**: Prints the thread table after context switches or system calls.

The simulator detects whether the program includes a USER instruction to enable OS mode and initialize the thread manager. It handles exceptions such as invalid memory accesses and provides error messages to the standard error stream.

8 Challenges and Design Decisions

Implementing the GTU-C312 CPU and OS required careful handling of memory access restrictions and thread scheduling. The decision to use a dictionary for instructions simplified lookup but required careful parsing of the program file. The thread tables design balanced simplicity with functionality, storing only essential data to meet project requirements. The cooperative scheduling model was chosen to align with the non-preemptive OS specification, using SYSCALL YIELD to ensure fair thread execution. The debug modes were implemented to provide comprehensive insights into the systems state, aiding in development and verification.

9 Conclusion

The GTU-C312 CPU simulator and OS implementation fulfill the project requirements by providing a functional platform for executing a custom instruction set and managing multiple threads. The OS supports a round-robin scheduler and system calls, with three threads demonstrating sorting, searching, and sequence generation. The simulators debug modes enable detailed analysis of memory and thread states, facilitating understanding of the systems behavior. This project demonstrates the integration of low-level programming, OS design, and simulation techniques, providing a foundation for further exploration of computer systems.