GTU-C312 Operating System Project

Project Overview

This project implements a simulated CPU (GTU-C312) with a custom instruction set and a simple operating system. The project was developed as part of the Operating Systems course at Gebze Technical University. The initial CPU structure and understanding of the instruction set was aided by AI tools, while the actual implementation and development of the operating system was done independently.

System Architecture

Memory Organization

The system uses a sophisticated memory management system with the following structure:

OS Memory Space (0-999)

1. Registers (0-20):
    - Register 0: Program Counter
    - Register 1: Stack Pointer
    - Register 2: System Call Result
    - Register 4: Thread Number (for context switching)
    - Register 5: Thread Start Time
    - Register 6: Thread Instruction Count
    - Register 7: Thread State (0: ready, 1: blocked, 2: running)
    - Register 17: System Call Type (0: HLT, 1: YIELD, 2: PRN)
    - Register 18: Thread Temp PC Switch
    - Register 19: Current Running Thread Number
    - Register 20: Total Instructions Executed
2. Thread Tables (21-324):
    - Each thread table occupies 25 memory locations
    - Thread table 0 (OS): [50-74]
    - Thread table 1: [75-99]
    - Thread table 2: [100-124] And so on...
3. Round Robin Scheduler (398-419):
    - Head pointer: [398]
    - Tail pointer: [399]
    - Thread entries: [400-419]
    - Each thread entry contains:
        - Thread number
        - Next thread pointer

Thread Memory Space (1000-10999)

Each thread has a 1000-size memory block with:

- Data section at the start
- Instruction section
- Stack (grows downwards)

Operating System Implementation

Thread Management

The OS implements a comprehensive thread table system that tracks:

- Thread ID (Register 4)
- Start time (Register 5)
- Instruction execution count (Register 6)
- Thread state (Register 7)
- Program Counter and Stack Pointer

Scheduling System

The OS uses a round-robin scheduling algorithm with:

1. A circular linked list implementation
2. Head and tail pointers for efficient thread rotation
3. Support for up to 10 threads
4. Thread state management (ready, blocked, running)

System Call Handling

The OS implements three system calls:

1. SYSCALL HLT (Type 0): Thread termination
2. SYSCALL YIELD (Type 1): CPU yield for scheduling
3. SYSCALL PRN (Type 2): Memory content printing

Memory Management

The system uses a sophisticated memory management system with:

1. OS Memory Blocks (0-999)
2. Thread Memory Blocks (1000-10999)
3. Separate data and instruction sections
4. Stack management for each thread
5. Memory protection (user mode can only access addresses 1000+)

Project Structure

Source Code Organization

```
src/
├── CPU.cpp/h         - CPU implementation
├── Memory.cpp/h      - Memory management
├── DataBlock.cpp/h   - Data segment handling
├── InstructionBlock.cpp/h - Instruction handling
├── AMemoryBlock.cpp/h    - Abstract memory block
└── main.cpp          - Program entry point

include/              - Header files
functions/            - OS functions
├── round_robin.os    - Scheduler implementation
└── round_robin_remove_thread.os - Thread management
```

Build System

The project uses a Makefile for building and includes debug modes for:

- Memory content tracing
- Thread table monitoring
- Step-by-step execution

Screenshots

Round Robin Scheduler Implementation

Round Robin Scheduler

Round Robin Scheduler Design (Miro)

Round Robin Scheduler Design

Round Robin Thread Removal Process

Round Robin Thread Removal

Development Process

The project development followed these steps:

1. Initial CPU structure and instruction set understanding (with AI assistance)
2. Independent implementation of the CPU simulator
3. Development of the operating system in GTU-C312 code
4. Implementation of thread management and scheduling
5. Testing and debugging

Conclusion

This project successfully implements a simulated CPU with a custom instruction set and a simple operating system. The implementation demonstrates key operating system concepts including thread management, scheduling, and system calls. The project was developed with minimal AI assistance, primarily used for understanding the initial CPU structure and instruction set, while the actual implementation was done independently.

Chat Usage

Project Description Request

Exported on 6/7/2025 at 23:51:06 GMT+3 from Cursor (1.0.0)

User

@subject.txt Please provide a detailed description for me for this project

---

Cursor

I'll analyze the project document and provide you with a detailed description of this Operating Systems term project.

Detailed Project Description: GTU-C312 CPU and Operating System Simulation

This is a comprehensive Operating Systems term project for CSE 312/CSE 504 at Gebze Technical University, due June 1, 2025. Here's a detailed breakdown:

Project Overview

You'll be building a complete computer system simulation consisting of:

1. **A custom CPU simulator** (GTU-C312)
2. **A simple operating system** written in the CPU's assembly language
3. **Multiple threads** running on the OS
4. **A simulation framework** with debugging capabilities

Part 1: The GTU-C312 CPU

Architecture Features

- **No traditional registers** - uses memory-mapped registers instead
- **Memory locations 0-20** serve as special registers:
    - Address 0: Program Counter
    - Address 1: Stack Pointer
    - Address 2: System call result
    - Address 3: Instruction execution counter
    - Addresses 4-20: Reserved

Instruction Set (15 instructions)

The CPU has a minimal but complete instruction set:

Data Movement:

- `SET B A` - Set memory location A to value B
- `CPY A1 A2` - Copy from A1 to A2
- `CPYI A1 A2` - Indirect copy (A1 contains address to copy from)
- `CPYI2 A1 A2` - Double indirect copy (optional)

Arithmetic:

- `ADD A B` - Add value B to memory location A
- `ADDI A1 A2` - Add contents of A2 to A1
- `SUBI A1 A2` - Subtract A2 from A1, store result in A2

Control Flow:

- `JIF A C` - Jump to instruction C if memory A ≤ 0
- `CALL C` - Call subroutine at instruction C
- `RET` - Return from subroutine

Stack Operations:

- `PUSH A` - Push memory A onto stack
- `POP A` - Pop from stack into memory A

System Operations:

- `USER A` - Switch to user mode, jump to address in location A
- `HLT` - Halt CPU
- `SYSCALL PRN A` - Print contents of memory A (blocks for 100 instructions)
- `SYSCALL HLT` - Terminate current thread
- `SYSCALL YIELD` - Yield CPU to OS scheduler

Part 2: Memory Layout and Protection

Memory Organization

```
Address Range    | Data Segment      | Instruction Segment | Access Level
0-20            | Registers         | OS area             | Kernel only
21-999          | OS data           | OS instructions     | Kernel only
1000-1999       | Thread #1 data    | Thread #1 code    | User mode
2000-2999       | Thread #2 data    | Thread #2 code    | User mode
...             | ...               | ...                 | ...
10000-10999     | Thread #10 data   | Thread #10 code   | User mode
```

Protection Mechanism

- **Kernel mode**: Can access all memory (0-∞)
- **User mode**: Can only access addresses ≥ 1000
- Violation in user mode → thread termination

Part 3: Operating System Requirements

OS Implementation (in GTU-C312 assembly)

Your OS must implement:

1. **Thread Table Management**
   - Support for 10 threads + OS thread
   - Track for each thread:
     - Thread ID
     - Start time (instruction count at creation)
     - CPU time used
     - State (ready/blocked/running)
     - Saved registers (PC, stack pointer)
2. **Cooperative Scheduler**
   - Round-robin scheduling
   - Non-preemptive (threads must yield voluntarily)
   - Context switching on system calls
3. **System Call Handling**
   - `PRN`: Print and block for 100 instructions
   - `HLT`: Terminate thread
   - `YIELD`: Switch to next ready thread

Part 4: Required Threads (Minimum 3)

Thread 1: Sorting Algorithm

- Sort N numbers in ascending order
- Numbers provided in data segment
- Print sorted results

Thread 2: Linear Search

- Search for a key in N numbers
- Input: N, key, and array in data segment
- Output: Position of key (or not found)

Thread 3: Custom Program

- Your own design
- Must include loops and `PRN` calls
- Demonstrate creativity and understanding

Part 5: Simulation Framework

Implementation Language

- C/C++ or Python
- Must implement complete CPU simulator

Command Line Interface

```
./simulate filename -D <debug_mode>
```

Debug Modes

- **Mode 0**: Run program, dump memory after halt

- **Mode 1**: Dump memory after each instruction
- **Mode 2**: Dump memory + wait for keypress after each instruction
- **Mode 3**: Print thread table after context switches/system calls

Part 6: File Format

Programs use a structured format:

```
Begin Data Section
0 0      # PC
1 0      # Stack pointer
2 0      # Syscall result
...
255 0    # Other data
End Data Section

Begin Instruction Section
0 SET 10 50     # instruction 0
1 ADD 50 -1     # instruction 1
...
End Instruction Section
```

Project Deliverables

1. **CPU Simulator** (C/C++/Python)
2. **Operating System** (GTU-C312 assembly)
3. **Thread Programs** (GTU-C312 assembly)
4. **Simulation Framework** with debug modes
5. **Detailed Report** explaining your OS design
6. **AI Interaction Logs** (ChatGPT conversations, etc.)
7. **Test Outputs** showing system operation
8. **Live Demo** with Q&A session

Learning Objectives

By completing this project, you'll gain hands-on experience with:

- CPU architecture and instruction set design
- Operating system internals (scheduling, system calls, memory protection)
- Thread management and context switching
- Low-level programming and assembly language
- System simulation and debugging techniques

This project provides a complete end-to-end understanding of how computers work from the hardware level up through the operating system to application programs.

```
Address Range    | Data Segment      | Instruction Segment | Access Level
0-20            | Registers         | OS area             | Kernel only
21-999          | OS data           | OS instructions     | Kernel only
1000-1999       | Thread #1 data    | Thread #1 code    | User mode
2000-2999       | Thread #2 data    | Thread #2 code    | User mode
...             | ...               | ...                 | ...
10000-10999     | Thread #10 data   | Thread #10 code   | User mode
```

```
./simulate filename -D <debug_mode>
```

```
Begin Data Section
0 0      # PC
1 0      # Stack pointer
2 0      # Syscall result
...
255 0    # Other data
End Data Section

Begin Instruction Section
0 SET 10 50     # instruction 0
1 ADD 50 -1     # instruction 1
...
End Instruction Section
```

User

I didn't understand how the thread tables will be kept. In my cpu classes private field named thread_table or in the memory block of that thread *

Cursor

Great question! Let me clarify how the thread tables should be implemented based on the project requirements.

Based on the project specification, let me clarify how the thread tables should be implemented:

Thread Table Implementation: IN MEMORY (Not in CPU class)

The thread table should be stored **in the simulated memory** (specifically in the OS data area), **NOT** as a private field in your CPU class. Here's why and how:

Where the Thread Table Lives

Location: Memory addresses 21-999 (OS data area)

The thread table is part of the **Operating System's data structures**, and since your OS is written in GTU-C312 assembly code, all OS data must reside in the simulated memory that the GTU-C312 CPU can access.

Memory Layout for Thread Table

```
Memory Layout:
0-20:     CPU registers (PC, stack pointer, etc.)
21-999:   OS DATA AREA ← Thread table goes here
          ├── Thread table entries
          ├── OS scheduler variables
          ├── Other OS data structures
1000+:    User threads' memory areas
```

Thread Table Structure in Memory

For example, you might organize it like this:

```
Memory Address | Content
21            | Number of threads (e.g., 10)
22            | Current running thread ID
23            | Next thread to schedule

// Thread 0 (OS thread) entry starts at address 30
30            | Thread ID (0)
31            | Thread state (0=ready, 1=running, 2=blocked)
32            | Saved PC
33            | Saved stack pointer
34            | Start time
35            | CPU time used
36-39         | Reserved for future use

// Thread 1 entry starts at address 40
40            | Thread ID (1)
41            | Thread state
42            | Saved PC (initially 1000)
43            | Saved stack pointer (initially 1999)
44            | Start time
45            | CPU time used
46-49         | Reserved

// Thread 2 entry starts at address 50
50            | Thread ID (2)
51            | Thread state
52            | Saved PC (initially 2000)
53            | Saved stack pointer (initially 2999)
...and so on
```

How Your OS Manages the Thread Table

Your **OS code (written in GTU-C312 assembly)** will:

1. **Initialize the thread table** using `SET` instructions:

```
SET 10 21        # Set number of threads to 10
SET 0 22         # Set current thread to 0 (OS)
SET 0 30         # Thread 0 ID
SET 1 31         # Thread 0 state (running)
SET 1000 42      # Thread 1 initial PC
SET 1999 43      # Thread 1 initial stack pointer
```

2. **Update thread states** during scheduling:

```
# When switching from thread 1 to thread 2
CPY 0 42         # Save current PC to thread 1's saved PC
CPY 1 43         # Save current stack pointer to thread 1
SET 0 41         # Set thread 1 state to ready
CPY 52 0         # Load thread 2's PC into current PC
CPY 53 1         # Load thread 2's stack pointer
SET 1 51         # Set thread 2 state to running
```

What Your CPU Class Does

Your **CPU class (C/C++/Python)** should:

1. **NOT store thread table data** - it only simulates the hardware
2. **Provide memory access** for the OS to read/write thread table
3. **Handle mode switching** (kernel/user mode)
4. **Implement the `SYSCALL` instruction** which transfers control back to OS

Example CPU Class Structure

```
class CPU {
private:
    vector<long> memory;  // The simulated memory
    bool kernel_mode;
    bool halted;

    // NO thread_table here!

public:
    void execute();       // Execute one instruction
    void syscall();       // Handle system calls
    bool isHalted();
    void setMemory(int addr, long value);
    long getMemory(int addr);
};
```

Debug Mode 3 Implementation

When debug mode 3 is requested, your **CPU simulator** reads the thread table from memory addresses 21-999 and prints it, but the thread table data itself lives in the simulated memory, not in your CPU class.

Summary

- **Thread table = OS data structure**
- **Stored in simulated memory (addresses 21-999)**
- **Managed by OS code (GTU-C312 assembly)**
- **CPU class only provides memory access, not storage**
- **This simulates how real computers work** - the OS manages process tables in RAM, not in CPU hardware

This approach makes your simulation more realistic and forces you to implement proper OS data structure management using the limited GTU-C312 instruction set!

```
Memory Layout:
0-20:     CPU registers (PC, stack pointer, etc.)
21-999:   OS DATA AREA ← Thread table goes here
          ├── Thread table entries
          ├── OS scheduler variables
          ├── Other OS data structures
1000+:    User threads' memory areas
```

```
Memory Address | Content
21            | Number of threads (e.g., 10)
22            | Current running thread ID
23            | Next thread to schedule

// Thread 0 (OS thread) entry starts at address 30
30            | Thread ID (0)
31            | Thread state (0=ready, 1=running, 2=blocked)
32            | Saved PC
33            | Saved stack pointer
34            | Start time
35            | CPU time used
36-39         | Reserved for future use

// Thread 1 entry starts at address 40
40            | Thread ID (1)
41            | Thread state
42            | Saved PC (initially 1000)
43            | Saved stack pointer (initially 1999)
44            | Start time
45            | CPU time used
46-49         | Reserved

// Thread 2 entry starts at address 50
50            | Thread ID (2)
51            | Thread state
52            | Saved PC (initially 2000)
53            | Saved stack pointer (initially 2999)
...and so on
```

```
   SET 10 21        # Set number of threads to 10
   SET 0 22         # Set current thread to 0 (OS)
   SET 0 30         # Thread 0 ID
   SET 1 31         # Thread 0 state (running)
   SET 1000 42      # Thread 1 initial PC
   SET 1999 43      # Thread 1 initial stack pointer
```

```
   # When switching from thread 1 to thread 2
   CPY 0 42         # Save current PC to thread 1's saved PC
   CPY 1 43         # Save current stack pointer to thread 1
   SET 0 41         # Set thread 1 state to ready
   CPY 52 0         # Load thread 2's PC into current PC
   CPY 53 1         # Load thread 2's stack pointer
   SET 1 51         # Set thread 2 state to running
```

```
class CPU {
private:
    vector<long> memory;  // The simulated memory
    bool kernel_mode;
    bool halted;

    // NO thread_table here!

public:
    void execute();       // Execute one instruction
    void syscall();       // Handle system calls
    bool isHalted();
    void setMemory(int addr, long value);
    long getMemory(int addr);
};
```

User

would it be good practise to use different structures for data and instruction segments

---

Cursor

Excellent question! Yes, it would be **very good practice** to use different structures for data and instruction segments. Let me explain why and show you some implementation approaches.

Why Separate Data and Instruction Structures?

1. Reflects Real Computer Architecture

Real CPUs have separate memory spaces or at least logical separation between code and data (Harvard vs. von Neumann architectures). This separation helps with:

- **Security** (execute permissions vs. read/write permissions)
- **Optimization** (instruction caching vs. data caching)
- **Memory protection**

2. Type Safety and Clarity

Different structures make your code more maintainable and less error-prone:

```
// Bad - everything is just numbers
vector<long> memory;  // Is memory[100] an instruction or data?

// Good - clear separation
vector<long> data_segment;
vector<Instruction> instruction_segment;
```

3. Easier Debugging and Validation

- You can validate instruction format during loading
- Debug output can show instructions in human-readable format
- Easier to detect invalid instruction access

Recommended Implementation Approaches

Option 1: Separate Containers with Instruction Class

```
class Instruction {
public:
    string opcode;
    vector<long> operands;

    Instruction(string op, vector<long> ops) : opcode(op), operands(ops) {}

    // For debugging
    string toString() const {
        string result = opcode;
        for (long op : operands) {
            result += " " + to_string(op);
        }
        return result;
    }
};

class CPU {
private:
    vector<long> data_segment;           // For data storage
    vector<Instruction> instruction_segment;  // For instructions
    bool kernel_mode;
    bool halted;

public:
    void loadProgram(const string& filename);
    void execute();
    long getData(int address);
    void setData(int address, long value);
    Instruction getInstruction(int address);
};
```

Option 2: Union-based Approach

```
struct MemoryCell {
    enum Type { DATA, INSTRUCTION } type;
    union {
        long data_value;
        Instruction* instruction;
    };

    MemoryCell() : type(DATA), data_value(0) {}
};

class CPU {
private:
    vector<MemoryCell> memory;
    // ...
};
```

Option 3: Hybrid Approach (Recommended)

```
class CPU {
private:
    // Unified memory for data (like real computers)
    vector<long> memory;

    // Separate instruction storage for validation and debugging
    map<int, InstructionBlock> instructions;

    bool kernel_mode;
    bool halted;

public:
    void loadProgram(const string& filename);
    void execute();

    // Memory access with protection
    long getMemory(int address) {
        if (!kernel_mode && address < 1000) {
            throw runtime_error("User mode memory violation");
        }
        return memory[address];
    }

    void setMemory(int address, long value) {
        if (!kernel_mode && address < 1000) {
            throw runtime_error("User mode memory violation");
        }
        memory[address] = value;
    }

    // Instruction access
    InstructionBlock getInstruction(int pc) {
        if (instructions.find(pc) == instructions.end()) {
            throw runtime_error("Invalid instruction address: " + to_string(pc));
        }
        return instructions[pc];
    }
};
```

File Loading Implementation

```
void CPU::loadProgram(const string& filename) {
    ifstream file(filename);
    string line;
    bool in_data_section = false;
    bool in_instruction_section = false;

    while (getline(file, line)) {
        // Remove comments
        size_t comment_pos = line.find('#');
        if (comment_pos != string::npos) {
            line = line.substr(0, comment_pos);
        }

        // Trim whitespace
        line = trim(line);
        if (line.empty()) continue;

        if (line == "Begin Data Section") {
            in_data_section = true;
            continue;
        } else if (line == "End Data Section") {
            in_data_section = false;
            continue;
        } else if (line == "Begin Instruction Section") {
            in_instruction_section = true;
            continue;
        } else if (line == "End Instruction Section") {
            in_instruction_section = false;
            continue;
        }

        if (in_data_section) {
            // Parse: address value
            istringstream iss(line);
            int address;
            long value;
            iss >> address >> value;
            memory[address] = value;
        } else if (in_instruction_section) {
            // Parse: address opcode operand1 operand2 ...
            istringstream iss(line);
            int address;
            string opcode;
            iss >> address >> opcode;

            vector<long> operands;
            long operand;
            while (iss >> operand) {
                operands.push_back(operand);
            }

            instructions[address] = InstructionBlock(rest_of_line);
        }
    }
}
```

Benefits of This Approach

1. Better Error Detection

```
void CPU::execute() {
    int pc = memory[0];  // Get PC from memory location 0

    try {
        InstructionBlock inst = getInstruction(pc);
        executeInstruction(inst);
    } catch (const runtime_error& e) {
        cout << "Error at PC " << pc << ": " << e.what() << endl;
        halted = true;
    }
}
```

2. Enhanced Debugging

```
void CPU::debugPrint() {
    int pc = memory[0];
    cout << "PC: " << pc << " -> ";

    if (instructions.find(pc) != instructions.end()) {
        cout << instructions[pc].toString() << endl;
    } else {
        cout << "INVALID INSTRUCTION" << endl;
    }
}
```

3. Instruction Validation

```
void CPU::validateInstruction(const InstructionBlock& inst) {
    if (inst.opcode == "SET" && inst.operands.size() != 2) {
        throw runtime_error("SET instruction requires 2 operands");
    }
    if (inst.opcode == "ADD" && inst.operands.size() != 2) {
        throw runtime_error("ADD instruction requires 2 operands");
    }
    // ... more validations
}
```

Recommendation

I'd recommend **Option 3 (Hybrid Approach)** because it:

- Maintains unified memory like real computers
- Provides instruction validation and debugging
- Supports memory protection
- Makes debugging much easier
- Follows good software engineering practices

This approach will make your CPU simulator more robust, easier to debug, and more educational in terms of understanding real computer architecture!

```
// Bad - everything is just numbers
vector<long> memory;  // Is memory[100] an instruction or data?

// Good - clear separation
vector<long> data_segment;
vector<InstructionBlock> instruction_segment;

class Instruction {
public:
    string opcode;
    vector<long> operands;

    Instruction(string op, vector<long> ops) : opcode(op), operands(ops) {}

    // For debugging
    string toString() const {
        string result = opcode;
        for (long op : operands) {
            result += " " + to_string(op);
        }
        return result;
    }
};

class CPU {
private:
    vector<long> data_segment;           // For data storage
    vector<InstructionBlock> instruction_segment;  // For instructions
    bool kernel_mode;
    bool halted;

public:
    void loadProgram(const string& filename);
    void execute();
    long getData(int address);
    void setData(int address, long value);
    InstructionBlock getInstruction(int address);
};

struct MemoryCell {
    enum Type { DATA, INSTRUCTION } type;
    union {
        long data_value;
        InstructionBlock* instruction;
    };

    MemoryCell() : type(DATA), data_value(0) {}
};

class CPU {
private:
    vector<MemoryCell> memory;
    // ...
};

class CPU {
private:
    // Unified memory for data (like real computers)
    vector<long> memory;

    // Separate instruction storage for validation and debugging
    map<int, InstructionBlock> instructions;

    bool kernel_mode;
    bool halted;

public:
    void loadProgram(const string& filename);
    void execute();

    // Memory access with protection
    long getMemory(int address) {
        if (!kernel_mode && address < 1000) {
            throw runtime_error("User mode memory violation");
        }
        return memory[address];
    }

    void setMemory(int address, long value) {
        if (!kernel_mode && address < 1000) {
            throw runtime_error("User mode memory violation");
        }
        memory[address] = value;
    }

    // Instruction access
    InstructionBlock getInstruction(int pc) {
        if (instructions.find(pc) == instructions.end()) {
            throw runtime_error("Invalid instruction address: " + to_string(pc));
        }
        return instructions[pc];
    }
};

void CPU::loadProgram(const string& filename) {
    ifstream file(filename);
    string line;
    bool in_data_section = false;
    bool in_instruction_section = false;

    while (getline(file, line)) {
        // Remove comments
        size_t comment_pos = line.find('#');
        if (comment_pos != string::npos) {
            line = line.substr(0, comment_pos);
        }

        // Trim whitespace
        line = trim(line);
        if (line.empty()) continue;

        if (line == "Begin Data Section") {
            in_data_section = true;
            continue;
        } else if (line == "End Data Section") {
            in_data_section = false;
            continue;
        } else if (line == "Begin Instruction Section") {
            in_instruction_section = true;
            continue;
        } else if (line == "End Instruction Section") {
            in_instruction_section = false;
            continue;
        }

        if (in_data_section) {
            // Parse: address value
            istringstream iss(line);
            int address;
            long value;
            iss >> address >> value;
            memory[address] = value;
        } else if (in_instruction_section) {
            // Parse: address opcode operand1 operand2 ...
            istringstream iss(line);
            int address;
            string opcode;
            iss >> address >> opcode;

            vector<long> operands;
            long operand;
            while (iss >> operand) {
                operands.push_back(operand);
            }

            instructions[address] = InstructionBlock(opcode, operands);
        }
    }
}

void CPU::execute() {
    int pc = memory[0];  // Get PC

    try {
        InstructionBlock inst = getInstruction(pc);
        executeInstruction(inst);
    } catch (const runtime_error& e) {
        cout << "Error at PC " << pc << ": " << e.what() << endl;
        halted = true;
    }
}

void CPU::debugPrint() {
    int pc = memory[0];
    cout << "PC: " << pc << " -> ";

    if (instructions.find(pc) != instructions.end()) {
        cout << instructions[pc].toString() << endl;
    } else {
        cout << "INVALID INSTRUCTION" << endl;
    }
}

void CPU::validateInstruction(const InstructionBlock& inst) {
    if (inst.opcode == "SET" && inst.operands.size() != 2) {
        throw runtime_error("SET instruction requires 2 operands");
    }
    if (inst.opcode == "ADD" && inst.operands.size() != 2) {
        throw runtime_error("ADD instruction requires 2 operands");
    }
    // ... more validations
}
```

User

I need a data structer that can reach elements by array like way in cpp. I will use 10 * 1000 memory cells. Would it be best practise for my case to split memory for each thread and also into two parts (instruction and data). Explain this purpose in a wide way to make me sure about the "seperation". Because if I try to change this concept it will be bad for me.

Cursor

Looking at your code snippet and requirements, let me give you a comprehensive explanation of why memory separation is crucial for your project and how to implement it effectively.

Why Memory Separation is Essential for Your Project

1. Project Requirements Demand Separation

Based on the project specification, you **MUST** implement memory separation because:

```
Address Range    | Purpose           | Access Level
0-20            | CPU Registers     | Kernel only
21-999          | OS Data/Code      | Kernel only
1000-1999       | Thread #1         | User mode
2000-2999       | Thread #2         | User mode
```

```
...              | ...               | ...
10000-10999      | Thread #10        | User mode
```

This is **not optional** - it's a core requirement for:

- Memory protection (user mode can't access 0-999)
- Thread isolation
- OS functionality

2. Array-like Access with Logical Separation

You can have **both** array-like access AND logical separation:

```
class Memory {
private:
    std::vector<long> unified_memory;  // 10,000+ cells, array-like access

public:
    // Array-like access
    long& operator[](int address) {
        return unified_memory[address];
    }

    // Protected access with validation
    long getData(int address, bool kernel_mode) {
        if (!kernel_mode && address < 1000) {
            throw std::runtime_error("Memory protection violation");
        }
        return unified_memory[address];
    }

    void setData(int address, long value, bool kernel_mode) {
        if (!kernel_mode && address < 1000) {
            throw std::runtime_error("Memory protection violation");
        }
        unified_memory[address] = value;
    }
};
```

Recommended Architecture for Your Project

Option 1: Unified Memory with Logical Separation (Recommended)

```
class CPU {
private:
    // Single array for all memory - easy access
    std::vector<long> memory;  // Size: 11000 (0-10999)

    // Separate instruction storage for validation/debugging
    std::map<int, InstructionBlock> instructions;

    bool kernel_mode;
    bool halted;

public:
    // Array-like access (internal use)
    long& operator[](int address) { return memory[address]; }

    // Protected memory access
    long getMemory(int address) {
        if (!kernel_mode && address < 1000) {
            throw std::runtime_error("User mode violation at address " + std::to_string(address));
        }
        return memory[address];
    }

    void setMemory(int address, long value) {
        if (!kernel_mode && address < 1000) {
            throw std::runtime_error("User mode violation at address " + std::to_string(address));
        }
        memory[address] = value;
    }

    // Instruction access
    InstructionBlock getInstruction(int pc) {
        if (instructions.find(pc) == instructions.end()) {
            throw std::runtime_error("No instruction at PC " + std::to_string(pc));
        }
        return instructions[pc];
    }
};
```

Option 2: Separated Memory Blocks

```
class CPU {
private:
    // Separate memory regions
    std::vector<long> registers;        // 0-20
    std::vector<long> os_memory;        // 21-999
    std::vector<std::vector<long>> thread_data;     // 10 threads × 1000 cells each

    std::map<int, InstructionBlock> os_instructions;
    std::vector<std::map<int, InstructionBlock>> thread_instructions;  // Per thread

public:
    long getMemory(int address) {
        if (address <= 20) {
            return registers[address];
        } else if (address <= 999) {
            if (!kernel_mode) throw std::runtime_error("Kernel memory access violation");
            return os_memory[address - 21];
        } else {
            // Thread memory: 1000-1999 = thread 0, 2000-2999 = thread 1, etc.
            int thread_id = (address - 1000) / 1000;
            int offset = (address - 1000) % 1000;
            return thread_data[thread_id][offset];
        }
    }
};
```

Why Separation is Critical for Your Success

1. Memory Protection Implementation

```
// Without separation - IMPOSSIBLE to implement protection
std::vector<long> memory;  // How do you know what's kernel vs user?

// With separation - EASY to implement protection
long getMemory(int address) {
    if (!kernel_mode && address < 1000) {
        // This is the CORE requirement of your project!
        throw std::runtime_error("Protection violation");
    }
    return memory[address];
}
```

2. Thread Isolation

```
// Each thread gets its own 1000-cell space
void switchToThread(int thread_id) {
    int base_address = 1000 + (thread_id * 1000);

    // Set thread's PC to its instruction area
    memory[0] = base_address;  // PC

    // Set thread's stack pointer to end of its data area
    memory[1] = base_address + 999;  // Stack pointer
}
```

3. Debug Mode Implementation

```
void printMemoryByRegion() {
    std::cerr << "=== REGISTERS (0-20) ===" << std::endl;
    for (int i = 0; i <= 20; i++) {
        std::cerr << i << ": " << memory[i] << std::endl;
    }

    std::cerr << "=== OS MEMORY (21-999) ===" << std::endl;
    for (int i = 21; i <= 999; i++) {
        if (memory[i] != 0) {  // Only show non-zero
            std::cerr << i << ": " << memory[i] << std::endl;
        }
    }

    for (int thread = 0; thread < 10; thread++) {
        std::cerr << "=== THREAD " << thread << " MEMORY ===" << std::endl;
        int base = 1000 + (thread * 1000);
        for (int i = 0; i < 1000; i++) {
            if (memory[base + i] != 0) {
                std::cerr << (base + i) << ": " << memory[base + i] << std::endl;
            }
        }
    }
}
```

Integration with Your InstructionBlock

Based on your code snippet, here's how to integrate:

```
class CPU {
private:
    std::vector<long> memory;  // Unified memory (11000 cells)
    std::map<int, InstructionBlock> instructions;

public:
    void loadProgram(const std::string& filename) {
        std::ifstream file(filename);
        std::string line;
        bool in_data = false, in_instruction = false;

        while (std::getline(file, line)) {
            // Parse sections...
            if (in_instruction) {
                std::istringstream iss(line);
                int address;
                std::string rest_of_line;
                iss >> address;
                std::getline(iss, rest_of_line);

                // Use your existing InstructionBlock constructor
                instructions[address] = InstructionBlock(rest_of_line);
            }
        }
    }

    void execute() {
        int pc = memory[0];  // Get PC

        if (instructions.find(pc) == instructions.end()) {
            throw std::runtime_error("No instruction at PC " + std::to_string(pc));
        }

        InstructionBlock& inst = instructions[pc];
        executeInstruction(inst);

        // Update PC and instruction counter
        memory[0]++;  // Increment PC (unless it was a jump)
        memory[3]++;  // Increment instruction counter
    }
};
```

Final Recommendation: Use Option 1

Unified Memory with Logical Separation because:

1. □ **Array-like access**: `memory[address]`
2. □ **Easy to implement**: Single vector
3. □ **Memory protection**: Check address ranges
4. □ **Thread isolation**: Address space partitioning
5. □ **Debug support**: Easy to dump entire memory
6. □ **Realistic**: How real computers work
7. □ **Project requirements**: Meets all specifications

Don't change this concept - it's fundamental to:

- Memory protection implementation
- Thread scheduling
- System call handling
- Debug mode functionality
- OS data structure management

The separation is **logical, not physical** - you still get array-like access while maintaining the security and isolation required by the project!

```
Address Range    | Purpose           | Access Level
0-20            | CPU Registers     | Kernel only
21-999          | OS Data/Code      | Kernel only
1000-1999       | Thread #1         | User mode
2000-2999       | Thread #2         | User mode
...             | ...               | ...
10000-10999     | Thread #10        | User mode
```

```
class Memory {
private:
    std::vector<long> unified_memory;  // 10,000+ cells, array-like access

public:
    // Array-like access
    long& operator[](int address) {
        return unified_memory[address];
    }

    // Protected access with validation
    long getData(int address, bool kernel_mode) {
        if (!kernel_mode && address < 1000) {
            throw std::runtime_error("Memory protection violation");
        }
        return unified_memory[address];
    }

    void setData(int address, long value, bool kernel_mode) {
        if (!kernel_mode && address < 1000) {
            throw std::runtime_error("Memory protection violation");
        }
        unified_memory[address] = value;
    }
};

class CPU {
private:
    // Single array for all memory - easy access
    std::vector<long> memory;  // Size: 11000 (0-10999)

    // Separate instruction storage for validation/debugging
    std::map<int, InstructionBlock> instructions;
    bool kernel_mode;
    bool halted;

public:
    // Array-like access (internal use)
    long& operator[](int address) { return memory[address]; }

    // Protected memory access
    long getMemory(int address) {
        if (!kernel_mode && address < 1000) {
            throw std::runtime_error("User mode violation at address " + std::to_string(address));
        }
        return memory[address];
    }

    void setMemory(int address, long value) {
        if (!kernel_mode && address < 1000) {
            throw std::runtime_error("User mode violation at address " + std::to_string(address));
        }
        memory[address] = value;
    }

    // Instruction access
    InstructionBlock getInstruction(int pc) {
        if (instructions.find(pc) == instructions.end()) {
            throw std::runtime_error("No instruction at PC " + std::to_string(pc));
        }
        return instructions[pc];
    }
};

class CPU {
private:
    // Separate memory regions
    std::vector<long> registers;        // 0-20
    std::vector<long> os_memory;        // 21-999
    std::vector<std::vector<long>> thread_data;     // 10 threads × 1000 cells each

    std::map<int, InstructionBlock> os_instructions;
    std::vector<std::map<int, InstructionBlock>> thread_instructions;  // Per thread

public:
    long getMemory(int address) {
        if (address <= 20) {
            return registers[address];
        } else if (address <= 999) {
            if (!kernel_mode) throw std::runtime_error("Kernel memory access violation");
            return os_memory[address - 21];
        } else {
            // Thread memory: 1000-1999 = thread 0, 2000-2999 = thread 1, etc.
            int thread_id = (address - 1000) / 1000;
            int offset = (address - 1000) % 1000;
            return thread_data[thread_id][offset];
        }
    }
};

// Without separation - IMPOSSIBLE to implement protection
std::vector<long> memory;  // How do you know what's kernel vs user?

// With separation - EASY to implement protection
long getMemory(int address) {
    if (!kernel_mode && address < 1000) {
        // This is the CORE requirement of your project!
        throw std::runtime_error("Protection violation");
    }
    return memory[address];
}

// Each thread gets its own 1000-cell space
void switchToThread(int thread_id) {
    int base_address = 1000 + (thread_id * 1000);

    // Set thread's PC to its instruction area
    memory[0] = base_address;  // PC

    // Set thread's stack pointer to end of its data area
    memory[1] = base_address + 999;  // Stack pointer
}

void printMemoryByRegion() {
    std::cerr << "=== REGISTERS (0-20) ===" << std::endl;
    for (int i = 0; i <= 20; i++) {
        std::cerr << i << ": " << memory[i] << std::endl;
    }

    std::cerr << "=== OS MEMORY (21-999) ===" << std::endl;
    for (int i = 21; i <= 999; i++) {
        if (memory[i] != 0) {  // Only show non-zero
            std::cerr << i << ": " << memory[i] << std::endl;
        }
    }

    for (int thread = 0; thread < 10; thread++) {
        std::cerr << "=== THREAD " << thread << " MEMORY ===" << std::endl;
        int base = 1000 + (thread * 1000);
        for (int i = 0; i < 1000; i++) {
            if (memory[base + i] != 0) {
                std::cerr << (base + i) << ": " << memory[base + i] << std::endl;
            }
        }
    }
}

class CPU {
private:
    std::vector<long> memory;  // Unified memory (11000 cells)
    std::map<int, InstructionBlock> instructions;

public:
    void loadProgram(const std::string& filename) {
        std::ifstream file(filename);
        std::string line;
        bool in_data = false, in_instruction = false;

        while (std::getline(file, line)) {
            // Parse sections...

            if (in_instruction) {
                std::istringstream iss(line);
                int address;
                std::string rest_of_line;
                iss >> address;
                std::getline(iss, rest_of_line);

                // Use your existing InstructionBlock constructor
                instructions[address] = InstructionBlock(rest_of_line);
            }
        }
    }

    void execute() {
        int pc = memory[0];  // Get PC

        if (instructions.find(pc) == instructions.end()) {
            throw std::runtime_error("No instruction at PC " + std::to_string(pc));
        }

        InstructionBlock& inst = instructions[pc];
        executeInstruction(inst);

        // Update PC and instruction counter
        memory[0]++;  // Increment PC (unless it was a jump)
        memory[3]++;  // Increment instruction counter
    }
};
```

User

How to use stack pointers in that project

Cursor

Great question! Let me explain how stack pointers work in the GTU-C312 project and how to implement them properly.

Stack Pointer Usage in GTU-C312 Project

Let me explain how stack pointers work in your project and provide comprehensive implementation details.

Key Stack Concepts in GTU-C312

1. Stack Pointer Location

- **Memory address 1**: Contains the current stack pointer
- **Stack grows downward** (from high addresses to low addresses)
- Each thread has its own stack within its memory region

2. Stack Memory Layout Per Thread

```
Thread Memory Layout (e.g., Thread 1: addresses 1000-1999):
┌─────────────────────────────────────┐
│ 1000-1500: Data area (variables, etc.) │
│ 1501-1998: Stack area               │ ← Stack grows down
│ 1999: Initial stack pointer         │ ← Stack starts here
└─────────────────────────────────────┘
```

Stack Pointer Implementation

1. CPU Class Stack Operations

```
class CPU {
private:
    std::vector<long> memory;

public:
    void executePush(long address) {
        // Get current stack pointer from memory[1]
        long stack_pointer = memory[1];

        // Get value to push from the specified address
        long value = memory[address];

        // Push value onto stack (stack grows downward)
        memory[stack_pointer] = value;

        // Decrement stack pointer
        memory[1] = stack_pointer - 1;

        std::cout << "PUSH: Pushed " << value << " to stack at " << stack_pointer << std::endl;
    }

    void executePop(long address) {
        // Increment stack pointer first (since stack grows downward)
        memory[1] = memory[1] + 1;

        // Get current stack pointer
        long stack_pointer = memory[1];

        // Pop value from stack
        long value = memory[stack_pointer];

        // Store popped value at specified address
        memory[address] = value;

        std::cout << "POP: Popped " << value << " from stack at " << stack_pointer << std::endl;
    }

    void executeCall(long instruction_address) {
        // Save return address (current PC + 1) on stack
        long return_address = memory[0] + 1;  // PC + 1

        // Push return address
        long stack_pointer = memory[1];
        memory[stack_pointer] = return_address;
        memory[1] = stack_pointer - 1;  // Decrement SP

        // Jump to subroutine
        memory[0] = instruction_address;  // Set PC

        std::cout << "CALL: Saved return address " << return_address << ", jumping to " << instruction_address << std::endl
    }

    void executeRet() {
        // Pop return address from stack
        memory[1] = memory[1] + 1;  // Increment SP
        long stack_pointer = memory[1];
        long return_address = memory[stack_pointer];

        // Jump back to return address
        memory[0] = return_address;  // Set PC

        std::cout << "RET: Returning to address " << return_address << std::endl;
    }
};
```

2. OS Thread Initialization

Your OS must set up stack pointers for each thread:

```
# OS code to initialize Thread 1 (GTU-C312 assembly)
# Thread 1 memory: 1000-1999

# Set Thread 1's initial stack pointer to end of its memory region
SET 1999 42    # Store 1999 in thread table entry for Thread 1's stack pointer

# When switching to Thread 1:
CPY 42 1       # Copy Thread 1's saved stack pointer to current SP (memory[1])
```

3. Thread Table Stack Pointer Management

```
// Thread table structure in memory (addresses 21-999)
struct ThreadTableEntry {
    long thread_id;        // e.g., address 30 for thread 0
    long state;           // e.g., address 31 for thread 0
    long saved_pc;        // e.g., address 32 for thread 0
    long saved_sp;        // e.g., address 33 for thread 0 ← STACK POINTER
    long start_time;      // e.g., address 34 for thread 0
    long cpu_time;        // e.g., address 35 for thread 0
};
```

4. Context Switching with Stack Pointers

```
# OS Context Switch Code (GTU-C312 assembly)
# Save current thread's stack pointer
```

```
CPY 1 33       # Save current SP to thread 0's saved_sp (address 33)

# Load next thread's stack pointer
CPY 43 1       # Load thread 1's saved_sp (address 43) to current SP
```

Practical Stack Usage Examples

1. Function Call Example

```
# Thread code using stack for function calls
Begin Instruction Section
1000 SET 5 1010      # Store parameter 5 at address 1010
1001 PUSH 1010       # Push parameter onto stack
1002 CALL 1020       # Call function at address 1020
1003 POP 1011        # Get result from stack
1004 SYSCALL PRN 1011 # Print result
1005 SYSCALL YIELD   # Yield to OS

# Function at address 1020
1020 POP 1015        # Get parameter from stack
1021 ADD 1015 10     # Add 10 to parameter
1022 PUSH 1015       # Push result back onto stack
1023 RET             # Return to caller
End Instruction Section
```

2. Stack for Local Variables

```
# Using stack to save/restore variables
1000 SET 100 1010    # local_var = 100
1001 PUSH 1010       # Save local_var on stack
1002 SET 200 1010    # local_var = 200 (temporary use)
1003 SYSCALL PRN 1010 # Print 200
1004 POP 1010        # Restore original local_var
1005 SYSCALL PRN 1010 # Print 100 (restored value)
```

3. Recursive Function Example

```
# Factorial function using stack
1000 SET 5 1010      # Calculate factorial of 5
1001 PUSH 1010       # Push n onto stack
1002 CALL 1020       # Call factorial function
1003 POP 1011        # Get result
1004 SYSCALL PRN 1011 # Print factorial result

# Factorial function (recursive)
1020 POP 1015        # Get n from stack
1021 JIF 1015 1030   # If n <= 0, return 1
1022 ADD 1015 -1     # n = n - 1
1023 PUSH 1015       # Push (n-1)
1024 CALL 1020       # Recursive call factorial(n-1)
1025 POP 1016        # Get factorial(n-1)
1026 ADD 1015 1      # Restore n
1027 # Multiply n * factorial(n-1) - simplified for this example
1028 PUSH 1016       # Push result
1029 RET

1030 SET 1 1017      # Base case: return 1
1031 PUSH 1017
1032 RET
```

Stack Pointer Initialization in Your Program

1. Initial Data Section Setup

```
Begin Data Section
0 0          # PC
1 999        # Initial stack pointer (OS stack)
2 0          # System call result
3 0          # Instruction counter
...
# Thread table entries
33 1999      # Thread 1's initial stack pointer
43 2999      # Thread 2's initial stack pointer
53 3999      # Thread 3's initial stack pointer
...
End Data Section
```

2. Thread Stack Boundaries

```
void initializeThreadStacks() {
    for (int thread_id = 0; thread_id < 10; thread_id++) {
        int base_address = 1000 + (thread_id * 1000);
        int stack_top = base_address + 999;  // End of thread's memory

        // Store in thread table
        int thread_table_sp_address = 33 + (thread_id * 10);  // Assuming 10 entries per thread
        memory[thread_table_sp_address] = stack_top;

        std::cout << "Thread " << thread_id << " stack initialized at " << stack_top << std::endl;
    }
}
```

Important Stack Rules

1. Stack Overflow Protection

```
void executePush(long address) {
    long stack_pointer = memory[1];

    // Check for stack overflow (assuming thread 1: 1000-1999)
    int current_thread = getCurrentThread();
    int thread_base = 1000 + (current_thread * 1000);

    if (stack_pointer <= thread_base) {
        throw std::runtime_error("Stack overflow in thread " + std::to_string(current_thread));
    }

    memory[stack_pointer] = memory[address];
    memory[1] = stack_pointer - 1;
}
```

2. Stack Underflow Protection

```
void executePop(long address) {
    int current_thread = getCurrentThread();
    int thread_top = 1000 + (current_thread * 1000) + 999;

    if (memory[1] >= thread_top) {
        throw std::runtime_error("Stack underflow in thread " + std::to_string(current_thread));
    }

    memory[1] = memory[1] + 1;
    memory[address] = memory[memory[1]];
}
```

Summary

Stack pointers in your project are used for:

1. **Function calls** (`CALL`/`RET` instructions)
2. **Local variable storage** (`PUSH`/`POP` instructions)
3. **Parameter passing** between functions
4. **Context switching** (saving/restoring thread state)

Each thread has its own stack within its 1000-byte memory region, and the OS manages stack pointers during context switches by saving/restoring them in the thread table.

```
Thread Memory Layout (e.g., Thread 1: addresses 1000-1999):
┌────────────────────────────────────────┐
│ 1000-1500: Data area (variables, etc.) │
│ 1501-1998: Stack area                  │ ← Stack grows down
│ 1999: Initial stack pointer            │ ← Stack starts here
└────────────────────────────────────────┘
```

```
class CPU {
private:
    std::vector<long> memory;

public:
    void executePush(long address) {
        // Get current stack pointer from memory[1]
        long stack_pointer = memory[1];

        // Get value to push from the specified address
        long value = memory[address];

        // Push value onto stack (stack grows downward)
        memory[stack_pointer] = value;

        // Decrement stack pointer
        memory[1] = stack_pointer - 1;

        std::cout << "PUSH: Pushed " << value << " to stack at " << stack_pointer << std::endl;
    }

    void executePop(long address) {
        // Increment stack pointer first (since stack grows downward)
        memory[1] = memory[1] + 1;

        // Get current stack pointer
        long stack_pointer = memory[1];

        // Pop value from stack
        long value = memory[stack_pointer];

        // Store popped value at specified address
        memory[address] = value;

        std::cout << "POP: Popped " << value << " from stack at " << stack_pointer << std::endl;
    }
    void executeCall(long instruction_address) {
        // Save return address (current PC + 1) on stack
        long return_address = memory[0] + 1;  // PC + 1

        // Push return address
        long stack_pointer = memory[1];
        memory[stack_pointer] = return_address;
        memory[1] = stack_pointer - 1;  // Decrement SP

        // Jump to subroutine
        memory[0] = instruction_address;  // Set PC

        std::cout << "CALL: Saved return address " << return_address << ", jumping to " << instruction_address << std::endl
    }

    void executeRet() {
        // Pop return address from stack
        memory[1] = memory[1] + 1;  // Increment SP
        long stack_pointer = memory[1];
        long return_address = memory[stack_pointer];

        // Jump back to return address
        memory[0] = return_address;  // Set PC

        std::cout << "RET: Returning to address " << return_address << std::endl;
    }
};

# OS code to initialize Thread 1 (GTU-C312 assembly)
# Thread 1 memory: 1000-1999

# Set Thread 1's initial stack pointer to end of its memory region
SET 1999 42    # Store 1999 in thread table entry for Thread 1's stack pointer

# When switching to Thread 1:
CPY 42 1       # Copy Thread 1's saved stack pointer to current SP (memory[1])

// Thread table structure in memory (addresses 21-999)
struct ThreadTableEntry {
    long thread_id;        // e.g., address 30 for thread 0
    long state;           // e.g., address 31 for thread 0
    long saved_pc;        // e.g., address 32 for thread 0
    long saved_sp;        // e.g., address 33 for thread 0 ← STACK POINTER
    long start_time;      // e.g., address 34 for thread 0
    long cpu_time;        // e.g., address 35 for thread 0
};

# OS Context Switch Code (GTU-C312 assembly)
# Save current thread's stack pointer
CPY 1 33       # Save current SP to thread 0's saved_sp (address 33)

# Load next thread's stack pointer
CPY 43 1       # Load thread 1's saved_sp (address 43) to current SP

# Thread code using stack for function calls
Begin Instruction Section
1000 SET 5 1010      # Store parameter 5 at address 1010
1001 PUSH 1010       # Push parameter onto stack
1002 CALL 1020       # Call function at address 1020
1003 POP 1011        # Get result from stack
1004 SYSCALL PRN 1011 # Print result
1005 SYSCALL YIELD   # Yield to OS

# Function at address 1020
1020 POP 1015        # Get parameter from stack
1021 ADD 1015 10     # Add 10 to parameter
1022 PUSH 1015       # Push result back onto stack
1023 RET             # Return to caller
End Instruction Section

# Using stack to save/restore variables
1000 SET 100 1010    # local_var = 100
1001 PUSH 1010       # Save local_var on stack
1002 SET 200 1010    # local_var = 200 (temporary use)
1003 SYSCALL PRN 1010 # Print 200
1004 POP 1010        # Restore original local_var
1005 SYSCALL PRN 1010 # Print 100 (restored value)

# Factorial function using stack
1000 SET 5 1010      # Calculate factorial of 5
1001 PUSH 1010       # Push n onto stack
1002 CALL 1020       # Call factorial function
1003 POP 1011        # Get result
1004 SYSCALL PRN 1011 # Print factorial result

# Factorial function (recursive)
1020 POP 1015        # Get n from stack
1021 JIF 1015 1030   # If n <= 0, return 1
1022 ADD 1015 -1     # n = n - 1
1023 PUSH 1015       # Push (n-1)
1024 CALL 1020       # Recursive call factorial(n-1)
1025 POP 1016        # Get factorial(n-1)
1026 ADD 1015 1      # Restore n
1027 # Multiply n * factorial(n-1) - simplified for this example
1028 PUSH 1016       # Push result
1029 RET
```

```
1030 SET 1 1017      # Base case: return 1
1031 PUSH 1017
1032 RET

Begin Data Section
0 0         # PC
1 999       # Initial stack pointer (OS stack)
2 0         # System call result
3 0         # Instruction counter
...
# Thread table entries
33 1999     # Thread 1's initial stack pointer
43 2999     # Thread 2's initial stack pointer
53 3999     # Thread 3's initial stack pointer
...
End Data Section
```

```
void initializeThreadStacks() {
    for (int thread_id = 0; thread_id < 10; thread_id++) {
        int base_address = 1000 + (thread_id * 1000);
        int stack_top = base_address + 999;  // End of thread's memory

        // Store in thread table
        int thread_table_sp_address = 33 + (thread_id * 10);  // Assuming 10 entries per thread
        memory[thread_table_sp_address] = stack_top;

        std::cout << "Thread " << thread_id << " stack initialized at " << stack_top << std::endl;
    }
}

void executePush(long address) {
    long stack_pointer = memory[1];

    // Check for stack overflow (assuming thread 1: 1000-1999)
    int current_thread = getCurrentThread();
    int thread_base = 1000 + (current_thread * 1000);

    if (stack_pointer <= thread_base) {
        throw std::runtime_error("Stack overflow in thread " + std::to_string(current_thread));
    }

    memory[stack_pointer] = memory[address];
    memory[1] = stack_pointer - 1;
}

void executePop(long address) {
    int current_thread = getCurrentThread();
    int thread_top = 1000 + (current_thread * 1000) + 999;

    if (memory[1] >= thread_top) {
        throw std::runtime_error("Stack underflow in thread " + std::to_string(current_thread));
    }

    memory[1] = memory[1] + 1;
    memory[address] = memory[memory[1]];
}
```