

Final Project Documentation: Advanced Operating System Simulator

Brysen Pfingsten, Nathaniel Savoury,
David Fields

December 11, 2025

CSAS 3111 - Operating Systems
Fall 2025
Seton Hall University

Contents

1	Introduction	3
1.1	Problem Statement	3
1.2	Outline	3
2	Module 1: Process Simulation	4
2.1	Problem Statement	4
2.2	Implementation	4
2.2.1	Core CPU Components and Registers	4
3	Overview	5
4	Registers	5
4.1	General-Purpose Registers	5
4.2	Special Registers	5
5	Instruction Formats	5
5.1	R-Type Format	5
5.2	I-Type Format	6
5.3	J-Type Format	6
6	Instruction Encoding and Semantics	6
6.1	Integer Arithmetic (R-Type)	6
6.2	Multiply and Divide (R-Type)	6
6.3	Logical and Bitwise (R-Type)	6
6.4	Shift Instructions (R-Type)	7
6.5	Immediate Arithmetic and Logical (I-Type)	7
6.6	Load and Store (I-Type)	7
6.7	Branches (I-Type)	7
6.8	Jumps (J-Type and R-Type)	8
6.9	System and Exception Instructions	8
6.10	Exception Return (ERET)	8
7	Exception and Interrupt Model	8
7.1	Exception Types	8
7.2	Exception Entry	9
7.3	Exception Return	9
8	Module 2: Advanced Memory Management	10
8.1	Problem Statement	10
8.2	Implementation	10
8.2.1	Hierarchical Memory System	10
8.2.2	Memory Table	12
8.2.3	Dynamic Memory Allocation and Deallocation	12
9	Module 3: Process Scheduling and Context Switching	15
9.1	Problem Statement	15
9.2	Implementation	15
9.2.1	Process Control Block Enhancements	15
9.2.2	Scheduling Algorithms	16
9.2.3	Context Switching	20
9.2.4	Integration with Fetch-Decode-Execute Cycle	21

10 Module 4: Interrupt Handling and Dispatcher	23
10.1 Problem Statement	23
10.2 Implementation	23
10.2.1 Interrupt Types	23
10.2.2 Handling Interrupts	24
11 Module 5: Efficiency Analysis of Concurrency	27
11.1 Problem Statement	27
11.2 Implementation	27
11.2.1 Performance Metrics Setup	27
11.2.2 Implementation of Time Tracking	27
11.2.3 Data Comparison	27
11.2.4 Performance Comparison	27
11.2.5 Visualization and Reporting	27
12 The Simulation	28
13 Testing and Debugging	38
14 Conclusion	39
15 Appendix A: Screenshots	40

1 Introduction

1.1 Problem Statement

The goal of this project is to guide you through the process of building an advanced operating system simulator. It will challenge you to apply and integrate the key concepts you have learned in previous projects while introducing more advanced features such as multitasking, inter-process communication, memory hierarchy, and efficient process scheduling with real-time constraints.

The purpose of this project is to help you understand how a modern operating system manages CPU resources and handles tasks concurrently. You will explore various scheduling algorithms to ensure safe and efficient access to shared resources, all while simulating real-world CPU and memory behaviors. By the end of this project, you will have a solid understanding of how operating systems work and gain hands-on experience in building a fully functional OS simulator capable of efficiently managing multiple processes in a concurrent environment.

Eventually, this project aims to deepen your understanding of system-level programming and OS concepts, preparing you for real-world applications and advanced studies in computer science.

1.2 Outline

- Module 1: Process Simulation
- Module 2: Advanced Memory Management
- Module 3: Process Scheduling and Context Switching
- Module 4: Interrupt Handling and Dispatcher
- Module 5: Efficiency Analysis of Concurrency

2 Module 1: Process Simulation

2.1 Problem Statement

2.2 Implementation

2.2.1 Core CPU Components and Registers

3 Overview

This document specifies a small, real 32-bit ISA based closely on the original MIPS I architecture. All instructions are 32 bits wide and follow one of three formats: R-type, I-type, or J-type. The ISA includes:

- 32 general-purpose registers (GPRs).
- Special registers: PC, HI, LO.
- Basic system control registers: **Status**, **Cause**, **EPC**.
- Integer arithmetic and logical operations.
- Multiply and divide.
- Shift operations.
- Load and store instructions for bytes, halfwords, and words.
- Branch and jump instructions.
- System call and exception/interrupt return.

4 Registers

4.1 General-Purpose Registers

There are 32 general-purpose registers, each 32 bits wide.

Number	Name	Role / Convention
\$0	zero	Constant zero, reads as 0, writes ignored
\$1	at	Assembler temporary
\$2–\$3	v0--v1	Function return values
\$4–\$7	a0--a3	Function arguments
\$8–\$15	t0--t7	Temporaries (caller-saved)
\$16–\$23	s0--s7	Saved registers (callee-saved)
\$24–\$25	t8--t9	Temporaries
\$26–\$27	k0--k1	Reserved for kernel / OS use
\$28	gp	Global pointer
\$29	sp	Stack pointer
\$30	fp/s8	Frame pointer or extra saved register
\$31	ra	Return address for calls (JAL , JALR)

4.2 Special Registers

- **PC** (Program Counter): 32-bit address of the current instruction.
- **HI**, **LO**: 32-bit registers used to hold results of multiply and divide.
- **Status**: System status register (holds interrupt enable and mode bits).
- **Cause**: Encodes reason for last exception or interrupt.
- **EPC** (Exception Program Counter): Holds the address to return to after an exception, used by **ERET**.

5 Instruction Formats

All instructions are 32 bits. Bit 31 is the most significant bit (MSB).

5.1 R-Type Format

31–26	25–21	20–16	15–11	10–6	5–0
opcode	rs	rt	rd	shamt	funct
6 bits	5 bits	5 bits	5 bits	5 bits	6 bits

For all R-type instructions:

$$\text{opcode} = 0.$$

5.2 I-Type Format

31–26	25–21	20–16	15–0
opcode	rs	rt	immediate
6 bits	5 bits	5 bits	16 bits

The 16-bit immediate is sign-extended or zero-extended depending on the instruction.

5.3 J-Type Format

31–26	25–0
opcode	target
6 bits	26 bits

The effective jump address is formed as:

$$PC_{\text{next}} = (PC_{\text{current}}[31:28] \ll 28) \mid (\text{target} \ll 2).$$

6 Instruction Encoding and Semantics

This section lists the instructions in the ISA, along with their encoding and semantic meaning. All arithmetic is on 32-bit two's complement integers unless otherwise stated. For brevity, we use the following notation:

- GPR[i]: contents of general-purpose register i .
- PC: program counter.
- HI, LO: special multiply/divide registers.
- Mem[a]: memory access at byte address a .
- sext $_n(x)$: sign extension of x from n bits to 32 bits.
- zext $_n(x)$: zero extension of x from n bits to 32 bits.

6.1 Integer Arithmetic (R-Type)

All of these have `opcode` = 0.

Mnemonic	Format	Encoding	Description	Semantics
ADD	R	<code>funct</code> = 0x20	Add (signed)	$GPR[rd] = GPR[rs] + GPR[rt]$
ADDU	R	<code>funct</code> = 0x21	Add (unsigned)	Same as ADD but ignore signed overflow
SUB	R	<code>funct</code> = 0x22	Subtract (signed)	$GPR[rd] = GPR[rs] - GPR[rt]$
SUBU	R	<code>funct</code> = 0x23	Subtract (unsigned)	Same as SUB but ignore signed overflow

6.2 Multiply and Divide (R-Type)

Mnemonic	Format	Encoding	Description	Semantics
MULT	R	<code>funct</code> = 0x18	Signed multiply	$\{HI, LO\} = \text{sext}_{64}(GPR[rs]) \times \text{sext}_{64}(GPR[rt])$
MULTU	R	<code>funct</code> = 0x19	Unsigned multiply	$\{HI, LO\} = \text{zext}_{64}(GPR[rs]) \times \text{zext}_{64}(GPR[rt])$
DIV	R	<code>funct</code> = 0x1A	Signed divide	$LO = GPR[rs]/GPR[rt]$, $HI = GPR[rs] \bmod GPR[rt]$
DIVU	R	<code>funct</code> = 0x1B	Unsigned divide	$LO = GPR[rs]_u/GPR[rt]_u$, $HI = GPR[rs]_u \bmod GPR[rt]_u$
MFHI	R	<code>funct</code> = 0x10, <code>rs</code> = <code>rt</code> = 0	Move from HI	$GPR[rd] = HI$
MFLO	R	<code>funct</code> = 0x12, <code>rs</code> = <code>rt</code> = 0	Move from LO	$GPR[rd] = LO$
MTHI	R	<code>funct</code> = 0x11, <code>rt</code> = <code>rd</code> = 0	Move to HI	$HI = GPR[rs]$
MTLO	R	<code>funct</code> = 0x13, <code>rt</code> = <code>rd</code> = 0	Move to LO	$LO = GPR[rs]$

6.3 Logical and Bitwise (R-Type)

Mnemonic	Format	Encoding	Description	Semantics
AND	R	funct = 0x24	Bitwise AND	$\text{GPR}[rd] = \text{GPR}[rs] \wedge \text{GPR}[rt]$
OR	R	funct = 0x25	Bitwise OR	$\text{GPR}[rd] = \text{GPR}[rs] \vee \text{GPR}[rt]$
XOR	R	funct = 0x26	Bitwise XOR	$\text{GPR}[rd] = \text{GPR}[rs] \oplus \text{GPR}[rt]$
NOR	R	funct = 0x27	Bitwise NOR	$\text{GPR}[rd] = \neg(\text{GPR}[rs] \vee \text{GPR}[rt])$

6.4 Shift Instructions (R-Type)

Mnemonic	Format	Encoding	Description	Semantics
SLL	R	funct = 0x00	Shift left logical (immediate)	$\text{GPR}[rd] = \text{GPR}[rt] \ll \text{shamt}$
SRL	R	funct = 0x02	Shift right logical (immediate)	$\text{GPR}[rd] = \text{GPR}[rt] \gg \text{shamt}$ (logical)
SRA	R	funct = 0x03	Shift right arithmetic (immediate)	Arithmetic right shift, preserving sign bit
SLLV	R	funct = 0x04	Shift left logical (variable)	$\text{GPR}[rd] = \text{GPR}[rt] \ll (\text{GPR}[rs] \& 0x1F)$
SRLV	R	funct = 0x06	Shift right logical (variable)	$\text{GPR}[rd] = \text{GPR}[rt] \gg (\text{GPR}[rs] \& 0x1F)$ (logical)
SRAV	R	funct = 0x07	Shift right arithmetic (variable)	Arithmetic right shift by low 5 bits of $\text{GPR}[rs]$

6.5 Immediate Arithmetic and Logical (I-Type)

Mnemonic	Format	Opcode	Description	Semantics
ADDI	I	0x08	Add immediate (signed)	$\text{GPR}[rt] = \text{GPR}[rs] + \text{sext}_{16}(\text{imm})$
ADDIU	I	0x09	Add immediate (unsigned)	Same as ADDI but ignore signed overflow
ANDI	I	0x0C	And immediate	$\text{GPR}[rt] = \text{GPR}[rs] \wedge \text{zext}_{16}(\text{imm})$
ORI	I	0x0D	Or immediate	$\text{GPR}[rt] = \text{GPR}[rs] \vee \text{zext}_{16}(\text{imm})$
XORI	I	0x0E	Xor immediate	$\text{GPR}[rt] = \text{GPR}[rs] \oplus \text{zext}_{16}(\text{imm})$
SLTI	I	0x0A	Set less than immediate (signed)	$\text{GPR}[rt] = (\text{GPR}[rs] < \text{sext}_{16}(\text{imm})) ? 1 : 0$
SLTIU	I	0x0B	Set less than immediate (unsigned)	Unsigned comparison version of SLTI
LUI	I	0x0F	Load upper immediate	$\text{GPR}[rt] = \text{imm} \ll 16$

6.6 Load and Store (I-Type)

Effective address:

$$\text{EA} = \text{GPR}[rs] + \text{sext}_{16}(\text{imm}).$$

Memory is typically treated as byte-addressed, little-endian.

Mnemonic	Format	Opcode	Description	Semantics
LW	I	0x23	Load word	$\text{GPR}[rt] = \text{Mem32}[\text{EA}]$
SW	I	0x2B	Store word	$\text{Mem32}[\text{EA}] = \text{GPR}[rt]$
LB	I	0x20	Load byte (signed)	$\text{GPR}[rt] = \text{sext}_8(\text{Mem8}[\text{EA}])$
LBU	I	0x24	Load byte (unsigned)	$\text{GPR}[rt] = \text{zext}_8(\text{Mem8}[\text{EA}])$
LH	I	0x21	Load halfword (signed)	$\text{GPR}[rt] = \text{sext}_{16}(\text{Mem16}[\text{EA}])$
LHU	I	0x25	Load halfword (unsigned)	$\text{GPR}[rt] = \text{zext}_{16}(\text{Mem16}[\text{EA}])$
SB	I	0x28	Store byte	$\text{Mem8}[\text{EA}] = \text{GPR}[rt] \& 0xFF$
SH	I	0x29	Store halfword	$\text{Mem16}[\text{EA}] = \text{GPR}[rt] \& 0xFFFF$

6.7 Branches (I-Type)

The branch target address is computed relative to the address of the instruction *following* the branch. Let PC_{next} be the PC after fetching the branch (i.e., $\text{PC} + 4$). Then:

$$\text{Target} = \text{PC}_{\text{next}} + (\text{sext}_{16}(\text{imm}) \ll 2).$$

Mnemonic	Format	Opcode	Description	Semantics
BEQ	I	0x04	Branch if equal	If $\text{GPR}[rs] = \text{GPR}[rt]$, then $\text{PC} = \text{Target}$
BNE	I	0x05	Branch if not equal	If $\text{GPR}[rs] \neq \text{GPR}[rt]$, then $\text{PC} = \text{Target}$

6.8 Jumps (J-Type and R-Type)

Mnemonic	Format	Opcode/Funct	Description	Semantics
J	J	<code>opcode</code> = 0x02	Jump	$\text{PC} = (\text{PC}_{\text{current}}[31:28] \ll 28) \mid (\text{target} \ll 2)$
JAL	J	<code>opcode</code> = 0x03	Jump and link	$\text{GPR}[31] = \text{PC}_{\text{next}}$; then same as J
JR	R	<code>funct</code> = 0x08	Jump register	$\text{PC} = \text{GPR}[rs]$
JALR	R	<code>funct</code> = 0x09	Jump and link register	$\text{GPR}[rd] = \text{PC}_{\text{next}}$; $\text{PC} = \text{GPR}[rs]$

6.9 System and Exception Instructions

For system and exception-related instructions, we describe them in prose rather than putting lists inside table cells (which can cause LaTeX errors).

SYSCALL

Encoded as an R-type instruction with `opcode` = 0 and `funct` = 0x0C. When executed, this instruction triggers a software exception. The simulator should:

1. Save the appropriate instruction address into **EPC** (either the address of the syscall or the next instruction, depending on your chosen convention).
2. Set **Cause** to a code representing a system call exception.
3. Update **Status** to indicate kernel mode and (optionally) disable further interrupts.
4. Set **PC** to the configured exception vector address (e.g. 0x80000180).

BREAK

Encoded as an R-type instruction with `opcode` = 0 and `funct` = 0x0D. When executed, this triggers a breakpoint exception, which is handled similarly to **SYSCALL**, but with a different **Cause** code to distinguish it (e.g. for debugging or traps).

6.10 Exception Return (ERET)

In real MIPS this is encoded as a coprocessor 0 instruction. For this ISA we define:

- `opcode` = 0x10 (COP0),
- `rs` = 0x10,
- bits 5–0 (`funct`) = 0x18,
- all other fields zero.

Decoding is implemented as a special case: “if `opcode` is 0x10 and `funct` = 0x18, execute **ERET**.”

Semantics.

- $\text{PC} \leftarrow \text{EPC}$.
- Restore user/kernel mode and interrupt enable bits in **Status** as appropriate.

7 Exception and Interrupt Model

7.1 Exception Types

Typical exception causes include:

- System call (**SYSCALL**).
- Breakpoint (**BREAK**).

- Arithmetic overflow (e.g., **ADD** with overflow).
- Invalid instruction.
- Address error on load/store.
- External interrupt (e.g., timer, I/O).

The simulator sets **Cause** to an integer code representing one of these reasons.

7.2 Exception Entry

On an exception or interrupt, the CPU performs:

1. Save the faulting instruction address or the following address into **EPC**.
2. Set **Cause** to the appropriate exception code.
3. Modify **Status** to:
 - switch to kernel mode,
 - optionally disable further interrupts.
4. Set **PC** to a fixed exception vector address, e.g. **0x80000180**.

The kernel's exception handler at that address can then inspect **Cause**, **EPC**, and general registers to decide what to do.

7.3 Exception Return

When the kernel is finished handling the exception or interrupt, it executes **ERET**, which:

- restores **PC** from **EPC**,
- restores user/kernel mode (and possibly interrupt enable) from **Status**.

8 Module 2: Advanced Memory Management

needs Updating!!

8.1 Problem Statement

The objective of this module is to enhance the existing memory management system by implementing a hierarchical memory architecture consisting of main memory (RAM) and two levels of cache (L1 and L2). This design aims to simulate realistic memory behavior, where data is accessed through progressively slower layers of storage, improving efficiency through caching frequently used data. Additionally, a Memory Table will be introduced to track memory allocation and availability for each process, ensuring proper management of system resources. The module will support dynamic memory allocation and deallocation, allowing processes to request and release memory as needed using strategies such as First-Fit or Best-Fit. Through these implementations, the system will demonstrate efficient memory utilization, process-level memory tracking, and realistic cache behavior for multi-process and multithreaded environments.

8.2 Implementation

8.2.1 Hierarchical Memory System

To simulate a hierarchical memory system, we implemented the RAM as an array of word and the L1 and L2 caches are structures:

```
typedef struct
{
    Entry* items;
    int front;
    int count;
    int size;
} Cache;
```

where Entry is a structure:

```
typedef struct
{
    word val;
    mem_addr addr;
} Entry;
```

The two main functions for this module is `read_mem` and `write_mem`. The former takes in as input a memory address and returns the value at that address. It first checks the L1 cache, then the L2 cache, then finally the RAM. It also updates the hit/miss stats for the different caches. The latter takes a memory address and a value and writes that value at the given memory address. These are the core operations used to fetch and store information from the CPU to the main memory and vice versa.

```
//return the value at the given memory address
word read_mem(const mem_addr addr)
{
    int index;
    index = cache_search(&L1, addr);
    if(index != EMPTY_ADDR)
    {
        //Cache hit at L1
        L1cache_hit++;
        return L1.items[index].val;
    }

    //cache miss at L1
    L1cache_miss++;

    index = cache_search(&L2, addr);
    if(index != EMPTY_ADDR)
    {
        //Cache hit at l2
        L2cache_hit++;
    }
}
```

```

    word val = L2.items[index].val;
    //Update L1 cache to prevent future cache misses
    update_cache(&L1, addr, val);
    return val;
}

//Cache miss at L2
L2cache_miss++;

//Complete cache miss, so read RAM and update cache
word val = RAM[addr];
update_cache(&L1, addr, val);
update_cache(&L2, addr, val);
return val;
}

//write the given value to the given memory address.
void write_mem(const mem_addr addr, const word val)
{
    RAM[addr] = val;

    int index;

    //Update L1 Cache
    index = cache_search(&L1, addr);
    if(index != EMPTY_ADDR)
    {
        L1.items[index].val = val;
    }

    //Update L2 Cache
    index = cache_search(&L2, addr);
    if(index != EMPTY_ADDR)
    {
        L2.items[index].val = val;
    }
}

```

The helper functions `cache_search` and `update_cache` are used to manage interfacing with the caches. The former checks if the data can be found in the cache and if so returns the index. The latter inserts data into the cache following the interface of a double ended queue.

```

//find the address of the value if it exists in cache
int cache_search(Cache* cache, const mem_addr addr)
{
    for(int i = 0; i < cache->size; i++)
    {
        //the address we want was found in cache
        //so return the index of that address
        if(cache->items[i].addr == addr)
        {
            return i;
        }
    }
    //address not found so return signifier
    return EMPTY_ADDR;
}

//Update the given cache in case of misses
void update_cache(Cache* cache, const mem_addr addr, const word val)
{
    //calculate where in the cache to store the value
    int index = (cache->front + cache->count) % cache->size;
    cache->items[index].addr = addr;
    cache->items[index].val = val;
}

```

```

//update the size and count of the cache
if(cache->count < cache->size)
{
    //cache isn't full so we can just put the new
    //value in the next index in the cache
    cache->count++;
}
else
{
    //cache is full, so loop around to put the new
    //value at the front
    cache->front = (cache->front + 1) % cache->size;
}
}

```

8.2.2 Memory Table

We implement our Memory Table as a structure containing an array of memory blocks and a count. Each memory block contains the ID of the process it belongs to, the starting address in memory, the ending address in memory, and a flag for if that block is free.

```

typedef struct {
    int pid;
    dword start_addr;
    dword end_addr;
    bool is_free;
} MemoryBlock;

typedef struct
{
    MemoryBlock *blocks;
    int block_count;
} MemoryTable;

void init_memtable(const int size)
{
    //make the first entry into the table one large free block of memory
    MEMORY_TABLE.blocks = (MemoryBlock*)malloc(sizeof(MemoryBlock) * size);
    MEMORY_TABLE.blocks[0].pid = NO_PID;
    MEMORY_TABLE.blocks[0].is_free = true;
    MEMORY_TABLE.blocks[0].start_addr = 0;
    MEMORY_TABLE.blocks[0].end_addr = (size - 1);

    MEMORY_TABLE.block_count = 1;
    printf("initialized memory table with size %d \n" , size);
}

```

8.2.3 Dynamic Memory Allocation and Deallocation

We use the best fit method to dynamically allocate memory.

```

// allocate memory for a specific process
// using the best fit method
dword mallocate(int pid, int size)
{
    int best_size = WRS + 1;
    int index = -1;

    for(int i = 0; i < MEMORY_TABLE.block_count; i++)
    {
        if (MEMORY_TABLE.blocks[i].is_free)
        {

```

```

        int mem_block_size = (MEMORY_TABLE.blocks[i].end_addr - MEMORY_TABLE.blocks[i].
start_addr) + 1;
        if(mem_block_size >= size && mem_block_size < best_size)
        {
            best_size = mem_block_size;
            index = i;
        }
    }
}

//No memory free so ...idk
if(index == -1)
{
    printf("Could not fullfill process(PID %d)'s request for a (%d byte) chunk of memory: Not
Enough Free Space\n", pid, size);
    return -1;
}

//Modify the free space to house our process
//@david Boo! spooky mutation ~~oooooh~~
MemoryBlock* best_fit = &MEMORY_TABLE.blocks[index];

//save the old start and end addresses
DWORD old_start_addr = best_fit->start_addr;
DWORD old_end_addr = best_fit->end_addr;

DWORD new_end_addr = (old_start_addr + size) - 1;

//give the process the space
best_fit->pid = pid;
best_fit->is_free = false;
best_fit->end_addr = new_end_addr;

//cut down the size of the block
//to free up unused space
if(new_end_addr < old_end_addr)
{
    //shift all the blocks to the right to make room
    for(int i = MEMORY_TABLE.block_count; i > index + 1; i--)
    {
        MEMORY_TABLE.blocks[i] = MEMORY_TABLE.blocks[i - 1];
    }

    MEMORY_TABLE.blocks[index + 1].pid = NO_PID;
    MEMORY_TABLE.blocks[index + 1].is_free = true;
    MEMORY_TABLE.blocks[index + 1].start_addr = new_end_addr + 1;
    MEMORY_TABLE.blocks[index + 1].end_addr = old_end_addr;

    MEMORY_TABLE.block_count++;
}

printf("Process (PID %d) given (%d bytes) of memory from [%d --> %d]\n", pid, size,
best_fit->start_addr, best_fit->end_addr);
return best_fit->start_addr;
}

// free up the memory block associated with the process
void liberate(int pid)
{
    int index = 0;
    for(index = 0; index < MEMORY_TABLE.block_count; index++)
    {
        if(MEMORY_TABLE.blocks[index].pid == pid && !MEMORY_TABLE.blocks[index].is_free)
        {

```

```

    MEMORY_TABLE.blocks[index].pid = NO_PID;
    MEMORY_TABLE.blocks[index].is_free = true;
    printf("Freed (PID %d) at memory [%d --> %d]\n", pid, MEMORY_TABLE.blocks[index].
start_addr, MEMORY_TABLE.blocks[index].end_addr);
    break;
}
}
//if the pid was not found
if(index == MEMORY_TABLE.block_count){return;}

//merge newly freed block with the previous block if it's also free
if(index > 0 && MEMORY_TABLE.blocks[index - 1].is_free)
{
    MEMORY_TABLE.blocks[index - 1].end_addr = MEMORY_TABLE.blocks[index].end_addr;
    //shift everything left to clean the gap
    for(int i = index; i < MEMORY_TABLE.block_count - 1; i++)
    {
        MEMORY_TABLE.blocks[i] = MEMORY_TABLE.blocks[i + 1];
    }
    MEMORY_TABLE.block_count--;
    index--;
}

//merge with the next memory block if it's also free
if(index < MEMORY_TABLE.block_count - 1 && MEMORY_TABLE.blocks[index + 1].is_free)
{
    MEMORY_TABLE.blocks[index].end_addr = MEMORY_TABLE.blocks[index + 1].end_addr;
    //shift right to clean the gap
    for(int i = index + 1; i < MEMORY_TABLE.block_count - 1; i++)
    {
        MEMORY_TABLE.blocks[i] = MEMORY_TABLE.blocks[i + 1];
    }
    MEMORY_TABLE.block_count--;
} }

```

9 Module 3: Process Scheduling and Context Switching

This section describes how processes are represented, created, and completed within our simulated OS.

9.1 Problem Statement

In this module, you will extend the simulator by implementing advanced process scheduling algorithms and context switching mechanisms. The goal is to manage multiple processes efficiently while ensuring fair distribution of CPU time and supporting real-time constraints. This module will prepare the simulator to handle diverse workloads and process types (CPU-bound, I/O-bound, and mixed), laying the foundation for multitasking and system responsiveness.

9.2 Implementation

This section outlines the our solution to the problem statement. Section 9.2.1 presents out we extended the PCB structure from Section 2. Section 9.2.2 describes out we implemented the seven advanced scheduling algorithms. Section 9.2.3 shows how we handle context switching for the preemptive scheduling algorithms. Finally, Section 9.2.4 explains how we integrated the fetch-decode-execute cycle with processes and scheduling logic.

9.2.1 Process Control Block Enhancements

Figure 1 Representation of Processes

```
typedef enum {
    READY,
    RUNNING,
    SUSPEND_READY,
    BLOCKED,
    SUSPEND_BLOCKED,
    NEW,
    FINISHED
} ProcessState;

//To represent a process
typedef struct {
    int pid; //process id
    ProcessState state; //state of the process
    int priority; //priority level
    int burstTime; //time left to complete
    float responseRatio; //calculated as (waiting time + service time)
    Cpu cpu_state; // the state of the cpu
    uint32_t text_start; // Where code is in memory
    uint32_t text_size; // where said code is
    uint32_t data_start; // Where data is in memory
    uint32_t data_size; // Size of said data
    uint32_t stack_ptr; // Stack pointer value
} Process;
```

Figure 1 displays the representation of processes in our OS. We follow a 7-state model where a process can be in one of seven states:

1. READY
2. RUNNING
3. SUSPEND_READY
4. BLOCKED
5. SUSPEND_BLOCKED

6. NEW

7. FINISHED

The process structure contains a process id (pid), the state of the process (state), the priority of the process (priority), the burst time of the process (burstTime), the response ratio for the response (responseRatio), the state of the cpu for context switching (cpu_state), the location to the start of code in memory for the process (text_start), the location of the code in memory for the process (text_size), the location to the state of data in memory for the process (data_start), the location of the data in memory for the process (data_size), and finally the stack pointer for the process (stack_ptr).

9.2.2 Scheduling Algorithms

Figure 2 Representation of Queue

```
//To represent a queue
typedef struct {
    int next; //the index to next open space
    int capacity; //The size of the queue
    Process PCB[]; //The block to hold processes
} Queue;
```

To implement the seven scheduling algorithms for our OS, we first needed a queue to hold all of the processes. This led to creating the Queue structure displayed in Figure 2. The queue structure contains the index to the next open space in the queue (next), the size of queue (capacity), and an array holding all of the processes (PCB). We made a queue for each state that the process can have as well as priority ready queue with respect to burst time and priority. In the interest of brevity, the scheduling algorithm will be explained at a high level of abstraction.

Figure 3 The Round-Robin Scheduling algorithm

```
//the round robin scheduling algorithm
static void roundRobin(void) {
    int idx = 0;
    while (Ready_Queue->next != 0) {
        transferProcesses(NORMAL);
        Process currentProcess = Ready_Queue->PCB[idx];
        set_current_process(currentProcess.pid);
        transitionState(currentProcess, NORMAL);

        for (int i = 0; i < QUANTUM; i++) {
            fetch();
            execute();
            currentProcess.burstTime-=1;
        }

        if (currentProcess.burstTime > 0 && idx+1 >= Ready_Queue->next) {
            context_switch(NORMAL, true);
            idx = 0;
        } else if (currentProcess.burstTime > 0) {
            context_switch(NORMAL, true);
            idx+=1;
        } else {
            transitionState(currentProcess, NORMAL);
        }
    }
}
```

Round-Robin

The Round-Robin scheduling algorithm is displayed Figure 3 and we chose to have a time quantum of 3. While the ready queue is not empty, we loop over the following sequence:

1. Check for new processes
2. Get the current process and transition it's state
3. Execute the time slice for the current process
4. Check if process is finished:
 - (a) If the process is finished, transition it's state
 - (b) Otherwise switch the context

Priority-Based Scheduling

Figure 4 The Priority Based Scheduling algorithm

```
//uses priorityPriorityQueue
//the priority based scheduling algorithm
static void priorityBased(void) {
    while (Ready_Queue->next != 0) {
        transferProcesses(PRIORITYPRIORITY);
        Process highestPriorityP = Ready_Queue->PCB[0];
        set_current_process(highestPriorityP.pid);
        transitionState(highestPriorityP, PRIORITYPRIORITY);

        while (highestPriorityP.burstTime > 0) {
            fetch();
            execute();
            highestPriorityP.burstTime--=1;
            transferProcesses(PRIORITYPRIORITY);
            Process newHighestP = Ready_Queue->PCB[0];

            if (&newHighestP != &highestPriorityP) {
                if (&Ready_Queue->PCB[1] == &newHighestP) {
                    context_switch(PRIORITYPRIORITY, true);
                } else {
                    context_switch(PRIORITYPRIORITY, true);
                }
            }
        }
        transitionState(highestPriorityP, PRIORITYPRIORITY);
    }
    set_current_process(SYSTEM_PROCESS_ID);
}
```

The implementation of the Priority-Based scheduling algorithm is found in Figure 4. We implemented a ready queue where the processes are sorted with respect to priority. While the ready queue is not empty, we loop over the following sequence:

1. Check for new processes
2. Get the current process and transition it's state
3. Execute the current process, check for new processes and get the process with the new highest priority:
 - (a) If new process is different from the current process, switch the context
 - (b) Otherwise continue executing the current process
4. When the current process is finished, transition it's state

Shortest Time Remaining

Figure 5 The Shortest Time Remaining Scheduling algorithm

```
//uses priorityBurstQueue
//the shortest time remaining scheduling algorithm
static void shortestRemainingTime(void) {
    while (Ready_Queue->next != 0) {
        transferProcesses(PRIORITYBURST);
        Process shortestBTimeP = Ready_Queue->PCB[0];
        set_current_process(shortestBTimeP.pid);
        transitionState(shortestBTimeP, PRIORITYBURST);

        while (shortestBTimeP.burstTime > 0) {
            fetch();
            execute();
            shortestBTimeP.burstTime-=1;
            transferProcesses(PRIORITYBURST);
            Process newShortestBTimeP = Ready_Queue->PCB[0];

            if (&newShortestBTimeP != &shortestBTimeP) {
                if (&Ready_Queue->PCB[1] == &newShortestBTimeP) {
                    context_switch(PRIORITYBURST, true);
                } else {
                    context_switch(PRIORITYBURST, true);
                }
            }
        }
        transitionState(shortestBTimeP, PRIORITYBURST);
    }
    set_current_process(SYSTEM_PROCESS_ID);
}
```

The implementation of the Shortest Time Remaining scheduling algorithm is found in Figure 5. We implemented a ready queue where the processes are sorted with respect to burst time. While the ready queue is not empty, we loop over the following sequence:

1. Check for new processes
2. Get the current process and transition it's state
3. Execute the current process, check for new processes and get the process with the new smallest burst:
 - (a) If new process is different from the current process, switch the context
 - (b) Otherwise continue executing the current process
4. When the current process is finished, transition it's state

Highest Response Ratio Next

The implementation of the Highest Response Ratio Next scheduling algorithm is found in Figure 6. While the ready queue is not empty, we loop over the following sequence:

1. Check for new processes
2. Get the current process, transition it's state and accumulate it's burst time
3. Execute the current process to completion and transition it's state
4. Update the response ratio for the remaining processes and get the process with the new highest response ratio

Figure 6 The Highest Response Ratio Next Scheduling algorithm

```
//the highest response ratio next scheduling algorithm
static void highestResponseRatioNext(void) {
    transferProcesses(NORMAL);
    if (Ready_Queue->next != 0) {
        int total_time = 0;
        Process currentProcess = Ready_Queue->PCB[0];
        set_current_process(currentProcess.pid);
        total_time = currentProcess.burstTime;
        transitionState(currentProcess, NORMAL);

        while (Ready_Queue->next != 0) {
            transferProcesses(NORMAL);
            while (currentProcess.burstTime > 0) {
                fetch();
                execute();
                currentProcess.burstTime-=1;
            }

            transitionState(currentProcess, NORMAL);
            updateResponseRatio(Ready_Queue, total_time);
            currentProcess = getHighestResponseRatio();
            total_time = currentProcess.burstTime;
            transitionState(currentProcess, NORMAL);
        }
    }
    set_current_process(SYSTEM_PROCESS_ID);
}
```

Figure 7 The First Come First Serve Scheduling algorithm

```
//The first come first serve scheduling algorithm
static void firstComeFirstServe(void) {
    while (Ready_Queue->next != 0) {
        transferProcesses(NORMAL);
        Process currentProcess = Ready_Queue->PCB[0];
        set_current_process(currentProcess.pid);
        transitionState(currentProcess, NORMAL);

        while (currentProcess.burstTime > 0) {
            fetch();
            execute();
            currentProcess.burstTime-=1;
        }
        transitionState(currentProcess, NORMAL);
    }

    set_current_process(SYSTEM_PROCESS_ID);
}
```

First Come First Serve

The implementation of the First Come First Serve scheduling algorithm is found in Figure 7. While the ready queue is not empty, we loop over the following sequence:

1. Check for new processes
2. Get the current process and transition it's state
3. Execute the current process to completion then transition it's state

Figure 8 The Shortest Process Next Scheduling algorithm

```
//uses priorityBurstQueue
//the shortest process next scheduling algorithm
static void shortestProcessNext(void) {
    while (Ready_Queue->next != 0) {
        transferProcesses(PRIORITYBURST);
        Process shortestProcess = Ready_Queue->PCB[0];
        set_current_process(shortestProcess.pid);
        transitionState(shortestProcess, PRIORITYBURST);

        while(shortestProcess.burstTime > 0) {
            fetch();
            execute();
            shortestProcess.burstTime-=1;
        }
        transitionState(shortestProcess, PRIORITYBURST);
    }

    set_current_process(SYSTEM_PROCESS_ID);
}
```

The implementation of the Shortest Process Next scheduling algorithm is found in Figure 8. We implemented a ready queue where the processes are sorted with respect to burst time. While the ready queue is not empty, we loop over the following sequence:

1. Check for new processes
2. Get the process with the smallest burst time and transition it's state
3. Execute the current process to completion and transition it's state

Feedback Scheduling

The implementation of the Feed Back scheduling algorithm is found in Figure 9. We implemented a 3 ready queues where the processes are moved to the lower queues if they are not completed. While all three queues are not empty, we loop over the following sequence:

1. Check for new processes
2. Execute the process in the highest priority queue, then middle priority queue and finally the lowest priority queue
3. Move the unfinished processes from the highest priority to the middle priority queue
4. Move the unfinished processes from the middle priority queue to the lowest priority queue
5. Move finished processes from lowest priority queue to the finished queue
6. When the current process is finished, transition it's state

9.2.3 Context Switching

The implementation for context switching is shown in Figure 10. This function takes in as input an integer designating the type of queue (e.i. a normal queue, a priority queue WRT priority or a priority queue WRT burst time) and a boolean denoting whether the two processes should have their states transitioned. The body of function extracts the first process from the running queue and the first process from the ready queue, called **curr** and **nxt** respectively. The current state of the cpu is saved into **curr** and the state of the cpu found within **nxt** is extracted and used to update the state of the cpu. Lastly, the processes transition their states if the **needTransition** argument is true.

Figure 9 The Feed Back Scheduling algorithm

```
//the feedback scheduling algorithm
static void feedBack(void) {
    Queue* feedBack_Q2 = init_FeedBack_Queue(MAX_PROCESSES);
    Queue* feedBack_Q3 = init_FeedBack_Queue(MAX_PROCESSES);
    int quantum1 = 2;
    int quantum2 = 4;

    while(true) {
        transferProcesses(NORMAL);
        //handle processes in highest priority queue with 2 quantum
        handleQueue(Ready_Queue, quantum1, false);
        //handle processes in middle priority queue with 4 quantum
        handleQueue(feedBack_Q2, quantum2, false);
        //handle processes in lowest priority queue FCFS
        handleQueue(feedBack_Q3, 0, true);

        //Moves unfinished processes from ready_queue to middle priority queue
        moveProcesses(Ready_Queue);
        //Moves unfinished processes from middle priority queue to lowest queue
        moveProcesses(feedBack_Q2);
        //move process from lowest priority queue to finished queue
        moveProcesses(feedBack_Q3);

        if (Ready_Queue->next == 0 && feedBack_Q2->next == 0 && feedBack_Q3->next == 0) {
            break;
        }
    }

    free_Queue(feedBack_Q2);
    free_Queue(feedBack_Q3);
    set_current_process(SYSTEM_PROCESS_ID);
}
```

Figure 10 Context Switching Implementation

```
//Switches from one process to another
static void context_switch(int queue_type, bool needTransition) {
    Process curr = Running_Queue->PCB[0];
    Process nxt = Ready_Queue->PCB[0];

    //save current's state
    curr.cpu_state = THE_CPU;

    set_current_process(nxt.pid);

    //start the next process
    THE_CPU = nxt.cpu_state;

    if (needTransition) {
        transitionState(curr, queue_type);
        transitionState(nxt, queue_type);
    }
}
```

9.2.4 Integration with Fetch-Decode-Execute Cycle

With the addition of schedulers to our OS, this requires additional logic for the fetch-decode-execute cycle. As a consequence of the way we implemented the algorithms, we handle most, if not all of the integration within

the schedule itself. Each scheduler checks if there a new processes and when applicable, reorders the priority queue such the process with the smallest burst time or priority. In addition, all of the preemptive schedulers handle context switching.

10 Module 4: Interrupt Handling and Dispatcher

10.1 Problem Statement

The objective of this module is to develop an advanced interrupt handling and dispatching system that enables the CPU to manage asynchronous events and process transitions efficiently. The system must allow the CPU to pause its current execution in response to various interrupts—such as timer, I/O, system call, trap, and priority interrupts—and appropriately service each event before resuming or switching processes. An Interrupt Vector Table (IVT) will be implemented to map interrupt types to their corresponding handlers, enabling fast and accurate interrupt resolution. Each handler will process its specific event, update process states, and invoke the dispatcher when necessary. The dispatcher will perform context switching by saving the current process state and restoring the next process's context based on the selected scheduling algorithm (Round-Robin or Priority-Based). This module ensures smooth and controlled transitions between processes, accurate interrupt servicing, and efficient CPU utilization in a multitasking environment.

10.2 Implementation

10.2.1 Interrupt Types

```
typedef enum irq{
    SAY_HI = 0x1,
    SAY_GOODBYE,
    TIMER_INTR,
    IO_INTR,
    SYS_CALL_INTR,
    TRAP_INTR,
    PRIORITY_INTR,
    EOI, //end of interrupt, make sure this is the last in the list
} IRQ;
```

We provide three types of interrupt opcodes SAY_HI, SAY_GOODBYE, and EOI.

```
typedef enum irq{
    SAY_HI = 0x1,
    SAY_GOODBYE,
    EOI, //end of interrupt
} IRQ;
```

Each interrupt instance contains one of these opcodes and a priority with 0 being the most important.

```
typedef struct {
    IRQ irq;
    int priority;
} Interrupt;
```

The interrupt controller is structured as an `InterruptHeap` which is a min-heap with constant time access to the highest priority interrupt and logarithmic insertion for new interrupts.

```
typedef struct {
    Interrupt data[MAX_INTERRUPTS];
    int size;
} InterruptHeap;
```

Finally, we store CPU states in a stack which allows for the pausing and resumption of processes by storing the information necessary to restart them.

```
typedef struct {
    Cpu* items;
    int SP;
} stack;
```

Interrupts are checked for every CPU cycle. If there is no interrupt then nothing happens. If there is an interrupt, its priority is checked against the potential current interrupt; if its priority is higher then it is immediately executed, if not it is added to the heap.

```
void check_for_interrupt() {
    if (!CPU.flags.INTERRUPT) return;
```



```

if (INTERRUPTCONTROLLER.size == 0){
    //no more interrupts in que, so clear flag
    set_interrupt_flag(false);
    return;
}

Interrupt intrpt = next_interrupt();
if(curr_intrrpt.irq == -1 || intrpt.priority < curr_intrrpt.priority){
    curr_intrrpt = intrpt;
    interrupt_handler(curr_intrrpt);
}else{
    interrupt_handler(curr_intrrpt);
    add_interrupt(intrpt.irq, intrpt.priority);
}

//clear flag if heap is empty after handle
if(INTERRUPTCONTROLLER.size == 0) set_interrupt_flag(false);
}

```

Interrupts are dispatched to their appropriate functions via the `interrupt_handler`:

```

void interrupt_handler(Interrupt intrpt) {
    //push the current CPU state to stack
    Cpu init_cpu_state = CPU;
    callstack.items[callstack.SP] = init_cpu_state;
    callstack.SP++;

    //decode the given interrupt and handle it
    switch(intrpt.irq) {
        case SAY_HI :
            printf("INTERRUPT: hello\n");
            set_interrupt_flag(false);
            reset_curr_interrupt();
            break;
        ...
        case EOI :
            set_interrupt_flag(false);
            reset_curr_interrupt();
            break;
        default:
            printf("ERROR: Invalid irq -> %u <-\n", (unsigned)intrpt.irq);
            CPU.PC = CPU_HALT;
            break;
    }

    //decrement the CPU stack
    callstack.SP--;
    //reset the CPU to it's original state
    CPU = callstack.items[callstack.SP];
    //increment the PC to start normal execution
    //CPU.PC++;
}

```

10.2.2 Handling Interrupts

We use a min-heap to store our interrupts based on priority giving us logarithmic insertions and constant time removals.

```

static void swap(Interrupt* a, Interrupt* b)
{
    Interrupt tmp = *a;
    *a = *b;
    *b = tmp;
}

```

```

static int parent(int i) { return (i - 1) / 2; }
static int left(int i)   { return 2 * i + 1; }
static int right(int i)  { return 2 * i + 2; }

/* ----- Core Functions ----- */

void init_interrupt_controller(void)
{
    INTERRUPTCONTROLLER.size = 0;

    for (int i = 0; i < MAX_INTERRUPTS; i++) {
        INTERRUPTCONTROLLER.data[i].irq = EOI;
        INTERRUPTCONTROLLER.data[i].priority = 100000;
    }

    callstack.SP = 0;
    callstack.items = malloc(sizeof(Cpu) * CALLSTACK_SIZE);
    if (!callstack.items)
    {
        fprintf(stderr, "Failed to allocate callstack\n");
        exit(EXIT_FAILURE);
    }

    curr_intrrpt.irq = EOI;
    curr_intrrpt.priority = 100000;

    printf("Initialized interrupt controller.\n");
}

// No more memory leaks yay :)
void free_interrupt_controller(void) {
    free(callstack.items);
    callstack.items = NULL;
}

// Add an interrupt to the heap (lower number = higher priority)
void add_interrupt(IRQ irq, int priority)
{
    if (INTERRUPTCONTROLLER.size >= MAX_INTERRUPTS)
    {
        fprintf(stderr, "Interrupt queue full!\n");
        return;
    }

    int i = INTERRUPTCONTROLLER.size++;
    INTERRUPTCONTROLLER.data[i].irq = irq;
    INTERRUPTCONTROLLER.data[i].priority = priority;

    while (i != 0 && INTERRUPTCONTROLLER.data[parent(i)].priority > INTERRUPTCONTROLLER.data[i].priority)
    {
        swap(&INTERRUPTCONTROLLER.data[i], &INTERRUPTCONTROLLER.data[parent(i)]);
        i = parent(i);
    }

    printf("[INTERRUPT] Queued IRQ %d (priority %d)\n", irq, priority);
}

/* Pop the highest-priority interrupt */
static Interrupt next_interrupt(void)
{
    if (INTERRUPTCONTROLLER.size <= 0)
    {

```

```

    Interrupt none = { EOI, 100000 };
    return none;
}

Interrupt root = INTERRUPTCONTROLLER.data[0];
INTERRUPTCONTROLLER.data[0] = INTERRUPTCONTROLLER.data[--INTERRUPTCONTROLLER.size];

int i = 0;
while (1)
{
    int l = left(i), r = right(i), smallest = i;

    if (l < INTERRUPTCONTROLLER.size &&
        INTERRUPTCONTROLLER.data[l].priority < INTERRUPTCONTROLLER.data[smallest].priority)
        smallest = l;

    if (r < INTERRUPTCONTROLLER.size &&
        INTERRUPTCONTROLLER.data[r].priority < INTERRUPTCONTROLLER.data[smallest].priority)
        smallest = r;

    if (smallest != i)
    {
        swap(&INTERRUPTCONTROLLER.data[i], &INTERRUPTCONTROLLER.data[smallest]);
        i = smallest;
    }
    else break;
}

return root;
}

```

11 Module 5: Efficiency Analysis of Concurrency

11.1 Problem Statement

11.2 Implementation

11.2.1 Performance Metrics Setup

11.2.2 Implementation of Time Tracking

11.2.3 Data Comparison

11.2.4 Performance Comparison

11.2.5 Visualization and Reporting

12 The Simulation

needs Updating!!

This program serves as an integrated system test for an advanced operating system simulator, demonstrating the interaction between multiple subsystems including the CPU, memory hierarchy, interrupt controller, DMA, and process management. It initializes all system components and launches three concurrent threads (processes) — one prints “Hello, Professor” using CPU interrupts, another performs repeated arithmetic calculations and memory operations, and a third simulates DMA data transfers from SSD and HDD into RAM. At the same time, separate timer and I/O interrupt threads generate asynchronous events to test interrupt handling and CPU responsiveness. After these processes finish, a demo CPU program is loaded into memory and executed to verify instruction execution, arithmetic logic, and memory storage. Finally, the program outputs memory contents, CPU register states, and cache statistics before freeing all resources. Overall, it tests the system’s ability to handle multithreading, interrupt-driven execution, DMA transfers, and coordinated CPU-memory operations within a simulated multitasking environment.

```
// Thread synchronization flags
volatile int process1_done = 0;
volatile int process2_done = 0;
volatile int process3_done = 0;

// ===== PROCESS 1: Print "Hello, Professor" using CPU Interrupts =====
void* process1_hello_professor(void* arg) {
    printf("\n[Process 1 Started] - Printing 'Hello, Professor' using CPU interrupts\n");

    // Allocate memory for this process
    dword proc1_mem = malloc(1, 100);

    // Store the string "Hello, Professor!\n" in memory (one char per dword)
    const char* message = "Hello, Professor!\n";
    int msg_len = 0;
    for (int i = 0; message[i] != '\0'; i++) {
        write_mem(proc1_mem + i, (dword)message[i]);
        msg_len++;
    }
    write_mem(proc1_mem + msg_len, 0); // Null terminator

    printf("[Process 1] Loaded message into memory at address 0x%X\n", proc1_mem);

    // Print the message 5 times using CPU interrupts
    for (int i = 0; i < 5; i++) {
        printf("[Process 1] Iteration %d/5: ", i + 1);

        // Set BX register to point to our string in memory
        THE_CPU.registers[BX] = proc1_mem;

        // Create and execute INT_PUTS interrupt instruction
        dword puts_instruction = (INTR << 24) | INT_PUTS;
        execute_instruction(INTR, puts_instruction);

        usleep(500000); // 0.5 seconds delay
    }

    // Store a completion marker in memory
    write_mem(proc1_mem + 50, 0xC0FFEE);
    printf("[Process 1] Stored completion marker at address 0x%X\n", proc1_mem + 50);

    // Free memory
    liberate(1);

    process1_done = 1;
    printf("[Process 1 Completed]\n\n");
    return NULL;
}

// ===== PROCESS 2: Complex Arithmetic Operations =====
void* process2_arithmetic(void* arg) {
```

```

printf("\n[Process 2 Started] - Performing arithmetic operations\n");
printf("[Process 2] Goal: Add to ACC until > 1,000,000, divide by 100, repeat 10 times, then
multiply by 2 and store\n\n");

// Allocate memory for this process
dword proc2_mem = malloc(2, 100);

// Create a local CPU state for this process
Cpu local_cpu;
local_cpu.registers[ACC] = 0;
local_cpu.registers[AX] = 1000000; // Target value
local_cpu.registers[BX] = 100;     // Divisor
local_cpu.registers[CX] = 0;       // Iteration counter

printf("[Process 2] Starting with ACC = %u\n", local_cpu.registers[ACC]);

// Repeat 10 times
for (int iteration = 0; iteration < 10; iteration++) {
    printf("[Process 2] === Iteration %d/10 ===\n", iteration + 1);

    // Add to ACC until it's greater than 1,000,000
    int add_count = 0;
    while (local_cpu.registers[ACC] <= local_cpu.registers[AX]) {
        local_cpu.registers[ACC] += 12345; // Add increments
        add_count++;
    }

    printf("[Process 2] Added %d times, ACC = %u\n", add_count, local_cpu.registers[ACC]);

    // Divide by 100
    local_cpu.registers[ACC] /= local_cpu.registers[BX];
    printf("[Process 2] After division by 100: ACC = %u\n", local_cpu.registers[ACC]);

    usleep(300000); // 0.3 seconds delay
}

// Multiply final result by 2
local_cpu.registers[ACC] *= 2;
printf("\n[Process 2] Final ACC value after multiplying by 2: %u\n", local_cpu.registers[ACC]);

// Store the result in memory
dword storage_addr = proc2_mem + 10;
write_mem(storage_addr, local_cpu.registers[ACC]);
printf("[Process 2] Stored final result %u at memory address 0x%X\n",
local_cpu.registers[ACC], storage_addr);

// Verify the stored value
dword verify = read_mem(storage_addr);
printf("[Process 2] Verification: Read back value %u from memory\n", verify);

// Free memory
liberate(2);

process2_done = 1;
printf("[Process 2 Completed]\n\n");
return NULL;
}

// ===== PROCESS 3: DMA Transfer from SSD/HDD =====
void* process3_dma_transfer(void* arg) {
    printf("\n[Process 3 Started] - Waiting for DMA transfer from SSD/HDD\n");

    // Allocate memory for this process

```

```

dword proc3_mem = mallocate(3, 200);

printf("[Process 3] Simulating data on SSD and HDD...\n");

// Prepare data on SSD
for (int i = 0; i < 10; i++) {
    SSD[i] = 0xAA00 + i;
}
printf("[Process 3] Prepared 10 words on SSD\n");

// Prepare data on HDD
for (int i = 0; i < 15; i++) {
    HDD[i] = 0xBB00 + i;
}
printf("[Process 3] Prepared 15 words on HDD\n");

// Simulate waiting for I/O
printf("[Process 3] Waiting for I/O operations...\n");
usleep(1000000); // 1 second delay

// DMA Transfer from SSD to RAM
printf("[Process 3] Initiating DMA transfer from SSD to RAM...\n");
initiateDMA(SSD, &RAM[proc3_mem], 10);
printf("[Process 3] DMA transfer from SSD complete\n");

// Verify SSD transfer
printf("[Process 3] Verifying SSD data in RAM:\n");
for (int i = 0; i < 10; i++) {
    printf("    RAM[%d] = 0x%X (expected 0x%X)\n",
        proc3_mem + i, RAM[proc3_mem + i], 0xAA00 + i);
}

usleep(500000); // 0.5 seconds delay

// DMA Transfer from HDD to RAM
printf("[Process 3] Initiating DMA transfer from HDD to RAM...\n");
initiateDMA(HDD, &RAM[proc3_mem + 20], 15);
printf("[Process 3] DMA transfer from HDD complete\n");

// Verify HDD transfer
printf("[Process 3] Verifying HDD data in RAM:\n");
for (int i = 0; i < 15; i++) {
    printf("    RAM[%d] = 0x%X (expected 0x%X)\n",
        proc3_mem + 20 + i, RAM[proc3_mem + 20 + i], 0xBB00 + i);
}

// Process the transferred data
dword sum = 0;
for (int i = 0; i < 10; i++) {
    sum += RAM[proc3_mem + i];
}
printf("[Process 3] Sum of SSD data: 0x%X\n", sum);

sum = 0;
for (int i = 0; i < 15; i++) {
    sum += RAM[proc3_mem + 20 + i];
}
printf("[Process 3] Sum of HDD data: 0x%X\n", sum);

// Free memory
liberate(3);

process3_done = 1;
printf("[Process 3 Completed]\n\n");

```



```

    return NULL;
}

// ===== INTERRUPT GENERATORS =====
void* timer_interrupt_thread(void* arg) {
    int count = 0;
    while (!process1_done || !process2_done || !process3_done) {
        sleep(2);
        add_interrupt(SAY_HI, 5);
        count++;
        if (count >= 3) break; // Limit interrupts
    }
    return NULL;
}

void* io_interrupt_thread(void* arg) {
    int count = 0;
    while (!process1_done || !process2_done || !process3_done) {
        sleep(3);
        add_interrupt(SAY_GOODBYE, 3);
        count++;
        if (count >= 2) break; // Limit interrupts
    }
    return NULL;
}

// ===== DEMO CPU PROGRAM =====
void load_demo_program() {
    printf("\n[System] Loading demo CPU program into memory...\n");

    // Simple program that demonstrates the instruction set
    int addr = 0;

    // Initialize some values in memory
    write_mem(0x100, 42);
    write_mem(0x101, 10);

    // Program: Load, Add, Multiply, Store
    // LOAD AX, 0x100
    RAM[addr++] = (LOAD << 24) | (AX << 20) | 0x100;

    // ADD AX, AX, #8 (immediate)
    RAM[addr++] = (ADD << 24) | (AX << 20) | (AX << 16) | (1 << 12) | 8;

    // MUL BX, AX, #2 (immediate)
    RAM[addr++] = (MUL << 24) | (BX << 20) | (AX << 16) | (1 << 12) | 2;

    // STORE BX, 0x102
    RAM[addr++] = (STORE << 24) | (BX << 20) | 0x102;

    // LOAD CX, 0x101
    RAM[addr++] = (LOAD << 24) | (CX << 20) | 0x101;

    // SUB DX, BX, CX
    RAM[addr++] = (SUB << 24) | (DX << 20) | (BX << 16) | (CX);

    // STORE DX, 0x103
    RAM[addr++] = (STORE << 24) | (DX << 20) | 0x103;

    // HALT instruction – use INT_HALT interrupt
    RAM[addr++] = (INTR << 24) | INT_HALT;

    printf("[System] Demo program loaded (%d instructions)\n\n", addr);
}

```

```

// ===== MAIN =====
int main() {
    printf("\n");
    printf("=====\n");
    printf("|          Advanced Operating System Simulator Demo          |\n");
    printf("|          Final - Fall 2025                                |\n");
    printf("=====\n");

    // ===== INITIALIZATION =====
    printf("\n[INITIALIZATION PHASE]\n");
    printf("=====\n");

    init_cache(&L1, L1CACHE_SIZE);
    init_cache(&L2, L2CACHE_SIZE);
    init_ram(RAM_SIZE);
    init_HDD(HDD_SIZE);
    init_SSD(SSD_SIZE);
    init_interrupt_controller();
    init_cpu(&THE_CPU);
    init_processes();

    printf("\n All systems initialized successfully\n");

    // ===== LOAD DEMO PROGRAM =====
    load_demo_program();

    // ===== CREATE PROCESS THREADS =====
    printf("\n[PROCESS EXECUTION PHASE]\n");
    printf("=====\n");
    printf("\nStarting 3 concurrent processes...\n");

    pthread_t proc1_thread, proc2_thread, proc3_thread;
    pthread_t timer_thread, io_thread;

    // Launch interrupt generators
    pthread_create(&timer_thread, NULL, timer_interrupt_thread, NULL);
    pthread_create(&io_thread, NULL, io_interrupt_thread, NULL);

    // Launch process threads
    pthread_create(&proc1_thread, NULL, process1_hello_professor, NULL);
    pthread_create(&proc2_thread, NULL, process2_arithmetic, NULL);
    pthread_create(&proc3_thread, NULL, process3_dma_transfer, NULL);

    // Wait for all processes to complete
    pthread_join(proc1_thread, NULL);
    pthread_join(proc2_thread, NULL);
    pthread_join(proc3_thread, NULL);

    printf("\n[CPU EXECUTION PHASE]\n");
    printf("=====\n");
    printf("\nExecuting loaded CPU program...\n\n");

    // Run the CPU with the loaded program
    cpu_run(7);

    // ===== RESULTS AND STATISTICS =====
    printf("\n[RESULTS AND STATISTICS]\n");
    printf("=====\n");

    printf("\nFinal Memory Contents:\n");
    printf("  Address 0x100: 0x%X (initial value)\n", read_mem(0x100));
    printf("  Address 0x101: 0x%X (initial value)\n", read_mem(0x101));
    printf("  Address 0x102: 0x%X (result of computation)\n", read_mem(0x102));

```

```

printf("  Address 0x103: 0x%X (result of subtraction)\n", read_mem(0x103));

print_cache_stats();

printf("\nCPU Final State:\n");
printf("  AX: 0x%X\n", THE_CPU.registers[AX]);
printf("  BX: 0x%X\n", THE_CPU.registers[BX]);
printf("  CX: 0x%X\n", THE_CPU.registers[CX]);
printf("  DX: 0x%X\n", THE_CPU.registers[DX]);

// ===== CLEANUP =====
printf("\n[CLEANUP PHASE]\n");
printf("=====\n");

// Cancel interrupt threads
pthread_cancel(timer_thread);
pthread_cancel(io_thread);
pthread_join(timer_thread, NULL);
pthread_join(io_thread, NULL);

// Free allocated resources
free_interrupt_controller();
free(L1.items);
free(L2.items);
free(RAM);
free(HDD);
free(SSD);

printf("\n All resources freed successfully\n");

printf("\n");
printf("=====\n");
printf("|                      Demo Completed Successfully                      |\n");
printf("=====\n");
printf("\n");
printf("Press Enter to exit...");
getchar();

return 0;
}

```

And the output

```

=====
|          Advanced Operating System Simulator Demo          |
|          Project 2 - Fall 2025                             |
=====

[INITIALIZATION PHASE]
=====
Initialized cache at -> '0x55d0da1a5160' <- with size: 5
Initialized cache at -> '0x55d0da1a5140' <- with size: 20
initialized ram with size: 500
initialized memory table with size 500
Initialized interrupt controller.
Initialized the cpu!
CPU STATE
AX: 0x0
BX: 0x0
CX: 0x0
DX: 0x0
PC:  0
ACC: 0
IR:  FFFFFFFF
FLAGS:

```

```
ZERO:      1
OVERFLOW:  0
CARRY:     0
```

All systems initialized successfully

[System] Loading demo CPU program into memory...

[System] Demo program loaded (8 instructions)

[PROCESS EXECUTION PHASE]

=====

Starting 3 concurrent processes...

[Process 1 Started] - Printing 'Hello, Professor' using CPU interrupts

Process (PID 1) given (100 bytes) of memory from [0 --> 99]

[Process 1] Loaded message into memory at address 0x0

[Process 1] Iteration 1/5: Hello, Professor!

[Process 2 Started] - Performing arithmetic operations

[Process 2] Goal: Add to ACC until > 1,000,000, divide by 100, repeat 10 times, then multiply by 2 and store

Process (PID 2) given (100 bytes) of memory from [100 --> 199]

[Process 2] Starting with ACC = 0

[Process 2] === Iteration 1/10 ===

[Process 2] Added 82 times, ACC = 1012290

[Process 2] After division by 100: ACC = 10122

[Process 3 Started] - Waiting for DMA transfer from SSD/HDD

Process (PID 3) given (200 bytes) of memory from [200 --> 399]

[Process 3] Simulating data on SSD and HDD...

[Process 3] Prepared 10 words on SSD

[Process 3] Prepared 15 words on HDD

[Process 3] Waiting for I/O operations...

[Process 2] === Iteration 2/10 ===

[Process 2] Added 81 times, ACC = 1010067

[Process 2] After division by 100: ACC = 10100

[Process 1] Iteration 2/5: Hello, Professor!

[Process 2] === Iteration 3/10 ===

[Process 2] Added 81 times, ACC = 1010045

[Process 2] After division by 100: ACC = 10100

[Process 2] === Iteration 4/10 ===

[Process 2] Added 81 times, ACC = 1010045

[Process 2] After division by 100: ACC = 10100

[Process 3] Initiating DMA transfer from SSD to RAM...

[Process 3] DMA transfer from SSD complete

[Process 3] Verifying SSD data in RAM:

RAM[200] = 0xAA00 (expected 0xAA00)

RAM[201] = 0xAA01 (expected 0xAA01)

RAM[202] = 0xAA02 (expected 0xAA02)

RAM[203] = 0xAA03 (expected 0xAA03)

RAM[204] = 0xAA04 (expected 0xAA04)

RAM[205] = 0xAA05 (expected 0xAA05)

RAM[206] = 0xAA06 (expected 0xAA06)

RAM[207] = 0xAA07 (expected 0xAA07)

RAM[208] = 0xAA08 (expected 0xAA08)

RAM[209] = 0xAA09 (expected 0xAA09)

[Process 1] Iteration 3/5: Hello, Professor!

[Process 2] === Iteration 5/10 ===

[Process 2] Added 81 times, ACC = 1010045

```

[Process 2] After division by 100: ACC = 10100
[Process 1] Iteration 4/5: [Process 3] Initiating DMA transfer from HDD to RAM...
[Process 3] DMA transfer from HDD complete
[Process 3] Verifying HDD data in RAM:
RAM[220] = 0xBB00 (expected 0xBB00)
RAM[221] = 0xBB01 (expected 0xBB01)
RAM[222] = 0xBB02 (expected 0xBB02)
RAM[223] = 0xBB03 (expected 0xBB03)
RAM[224] = 0xBB04 (expected 0xBB04)
RAM[225] = 0xBB05 (expected 0xBB05)
RAM[226] = 0xBB06 (expected 0xBB06)
RAM[227] = 0xBB07 (expected 0xBB07)
RAM[228] = 0xBB08 (expected 0xBB08)
RAM[229] = 0xBB09 (expected 0xBB09)
RAM[230] = 0xBB0A (expected 0xBB0A)
RAM[231] = 0xBB0B (expected 0xBB0B)
RAM[232] = 0xBB0C (expected 0xBB0C)
RAM[233] = 0xBB0D (expected 0xBB0D)
RAM[234] = 0xBB0E (expected 0xBB0E)
[Process 3] Sum of SSD data: 0x6A42D
[Process 3] Sum of HDD data: 0xAF569
Freed (PID 3) at memory [200 --> 399]
[Process 3 Completed]

```

Hello, Professor!

```

[Process 2] === Iteration 6/10 ===
[Process 2] Added 81 times, ACC = 1010045
[Process 2] After division by 100: ACC = 10100
[Process 2] === Iteration 7/10 ===
[Process 2] Added 81 times, ACC = 1010045
[Process 2] After division by 100: ACC = 10100
[INTERRUPT] Queued IRQ 1 (priority 5)
[Process 1] Iteration 5/5: Hello, Professor!
[Process 2] === Iteration 8/10 ===
[Process 2] Added 81 times, ACC = 1010045
[Process 2] After division by 100: ACC = 10100
[Process 2] === Iteration 9/10 ===
[Process 2] Added 81 times, ACC = 1010045
[Process 2] After division by 100: ACC = 10100
[Process 1] Stored completion marker at address 0x32
Freed (PID 1) at memory [0 --> 99]
[Process 1 Completed]

```

```

[Process 2] === Iteration 10/10 ===
[Process 2] Added 81 times, ACC = 1010045
[Process 2] After division by 100: ACC = 10100
[INTERRUPT] Queued IRQ 2 (priority 3)

```

```

[Process 2] Final ACC value after multiplying by 2: 20200
[Process 2] Stored final result 20200 at memory address 0x6E
[Process 2] Verification: Read back value 20200 from memory
Freed (PID 2) at memory [100 --> 199]
[Process 2 Completed]

```

[CPU EXECUTION PHASE]

=====

Executing loaded CPU program...

[RESULTS AND STATISTICS]

=====

Final Memory Contents:
Address 0x100: 0x2A (initial value)
Address 0x101: 0xA (initial value)
Address 0x102: 0xFFFFFFFF (result of computation)
Address 0x103: 0xFFFFFFFF (result of subtraction)

Cache statistics:

L1 hits: 0
L1 misses: 5
L2 hits: 0
L2 misses: 5

CPU Final State:

AX: 0x4
BX: 0x0
CX: 0x0
DX: 0x0

[CLEANUP PHASE]

=====

All resources freed successfully

=====

	Demo Completed Successfully	
--	-----------------------------	--

=====

13 Testing and Debugging

14 Conclusion

15 Appendix A: Screenshots