Final Project Documentation: Advanced Operating System Simulator

Brysen Pfingsten, Nathaniel Savoury,
David Fields

December 17, 2025

CSAS 3111 - Operating Systems
Fall 2025
Seton Hall University

Contents

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

This section specifies a small, real 32-bit ISA based closely on the original MIPS I architecture.

2.1 Problem Statement

In this module, you'll set up the basics for simulating a CPU that can handle multiple processes. The focus will be on building key CPU components, running the fetch-decode-execute cycle, and managing basic process states. This work will prepare you to tackle advanced scheduling and interrupt handling in the next modules.

2.2 Implementation

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.

3 Registers

3.1 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`)

3.2 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`.

4 Instruction Formats

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

4.1 R-Type Format

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

For all R-type instructions:

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

4.2 I-Type Format

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

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

4.3 J-Type Format

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

The effective jump address is formed as:

$$\mathrm{PC}_{\mathrm{next}} = \big(\mathrm{PC}_{\mathrm{current}}[31{:}28] \ll 28\big) \mid (\texttt{target} \ll 2).$$

5 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:

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

5.1 Integer Arithmetic (R-Type)

All of these have `opcode` = 0.

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

5.2 Multiply and Divide (R-Type)

Mnemonic	Format	Encoding	Description	Semantics
MULT	R	`funct` = 0x18	Signed multiply	$\{\mathrm{HI}, \mathrm{LO}\} = \mathrm{sext}_{64}(\mathrm{GPR}[rs]) \times \mathrm{sext}_{64}(\mathrm{GPR}[rt])$
MULTU	R	`funct` = 0x19	Unsigned multiply	$\{\mathrm{HI}, \mathrm{LO}\} = \mathrm{zext}_{64}(\mathrm{GPR}[rs]) \times \mathrm{zext}_{64}(\mathrm{GPR}[rt])$

Mnemonic	Format	Encoding	Description	Semantics
DIV	R	`funct` = 0x1A	Signed divide	LO = GPR[rs]/GPR[rt], HI = GPR[rs] mod GPR[rt]
DIVU	R	`funct` = 0x1B	Unsigned divide	LO = GPR[rs]$_u$/GPR[rt]$_u$, HI = GPR[rs]$_u$ mod GPR[rt]$_u$
MFHI	R	`funct` = 0x10, `rs` = `rt` = 0	Move from HI	GPR[rd] = HI
MFLO	R	`funct` = 0x12, `rs` = `rt` = 0	Move from LO	GPR[rd] = LO
MTHI	R	`funct` = 0x11, `rt` = `rd` = 0	Move to HI	HI = GPR[rs]
MTLO	R	`funct` = 0x13, `rt` = `rd` = 0	Move to LO	LO = GPR[rs]

5.3 Logical and Bitwise (R-Type)

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

5.4 Shift Instructions (R-Type)

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

5.5 Immediate Arithmetic and Logical (I-Type)

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

5.6 Load and Store (I-Type)

Effective address:

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

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

Mnemonic	Format	Opcode	Description	Semantics
LW	I	0x23	Load word	GPR[rt] = Mem32[EA]
SW	I	0x2B	Store word	Mem32[EA] = GPR[rt]
LB	I	0x20	Load byte (signed)	GPR[rt] = sext$_8$(Mem8[EA])
LBU	I	0x24	Load byte (unsigned)	GPR[rt] = zext$_8$(Mem8[EA])

Mnemonic	Format	Opcode	Description	Semantics
LH	I	0x21	Load halfword (signed)	$\mathrm{GPR}[rt] = \mathrm{sext}_{16}(\mathrm{Mem16}[\mathrm{EA}])$
LHU	I	0x25	Load halfword (unsigned)	$\mathrm{GPR}[rt] = \mathrm{zext}_{16}(\mathrm{Mem16}[\mathrm{EA}])$
SB	I	0x28	Store byte	$\mathrm{Mem8}[\mathrm{EA}] = \mathrm{GPR}[rt] \,\&\, 0xFF$
SH	I	0x29	Store halfword	$\mathrm{Mem16}[\mathrm{EA}] = \mathrm{GPR}[rt] \,\&\, 0xFFFF$

5.7 Branches (I-Type)

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

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

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

5.8 Jumps (J-Type and R-Type)

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

5.9 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).

5.10 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.

- PC ← `EPC`.
- Restore user/kernel mode and interrupt enable bits in `Status` as appropriate.

6 Exception and Interrupt Model

6.1 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.

6.2 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.

6.3 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`.

7 Module 2: Advanced Memory Management

needs Updating!!

7.1 Problem Statement

In this module, you'll expand the memory management system from Project 2 by adding features for efficient memory allocation, hierarchical memory structure, dynamic memory allocation, and shared memory access. Compared to Project 3, this module introduces enhanced features for memory efficiency and concurrency, simulating a more realistic and sophisticated operating system environment.

7.2 Implementation

7.2.1 Hierarchical Memory System

To simulate a hierarchical memory system, we implemented cache lines which stores the data in an array to allow for multiple sizes of data without overwriting data. The cache line is represented as structure:

```
typedef struct {
  uint32_t tag;
  bool is_valid;
  bool is_dirty;
  uint8_t data[CACHE_LINE_SIZE];
} CacheLine;
```

The L1 and L2 caches are structures:

```
typedef struct {
  CacheLine *lines;
  size_t front;
  size_t count;
  size_t line_count;
} Cache;
```

A `MemoryBlock` is a structure:

```
typedef struct {
  int pid;
  uint32_t start_addr;
  uint32_t end_addr;
  bool is_free;
} MemoryBlock;

typedef struct {
  MemoryBlock *blocks;
  size_t block_count;
  size_t capacity;
} MemoryTable;
```

The two main functions for this module is `read_mem` and `write_mem`. The former takes in as input a memory 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;
```

```c
}

//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;
  }
}
```

7.2.2 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.

```c
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);
}
```

7.2.3 Dynamic Memory Allocation and Deallocation

We use the best fit method to dynamically allocate memory.

```c
// 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 ~~ooooh~~
  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--;
} }
```

8 Module 3: Process Scheduling and Context Switching

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

8.1 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.

8.2 Implementation

This section outlines the our solution to the problem statement. Section 8.2.1 presents out we extended the PCB structure from **??**. Section 8.2.2 describes out we implemented the seven advanced scheduling algorithms. Section 8.2.3 shows how we handle context switching for the preemptive scheduling algorithms. Finally, Section 8.2.4 explains how we integrated the fetch-decode-execute cycle with processes and scheduling logic.

8.2.1 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).

8.2.2 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

Shortest Process Next

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

8.2.3 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);
  }
}
```

8.2.4 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.

9 Module 4: Interrupt Handling and Dispatcher

9.1 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.

9.2 Implementation

9.2.1 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++;
  }
```

9.2.2 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;
  }
```

```c
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;
}
```

10 Module 5: Efficiency Analysis of Concurrency

This sections outlines how we conducted an efficiency analysis of concurrency for our simulated `OS`.

10.1 Problem Statement

In this module, you will analyze the efficiency of your simulator by comparing the performance of different scheduling algorithms and concurrency mechanisms implemented in the previous modules. The goal is to evaluate the impact of scheduling strategies, concurrency, and synchronization on execution time, CPU utilization, memory usage, and overall system efficiency.

10.2 Implementation

10.2.1 Performance Metrics Setup

To collect performance metrics, we created the following structures:

```c
// Performance metrics for individual processes
typedef struct {
  int pid;
  int arrival_time;
  int burst_time;
  int completion_time;
  int waiting_time;
  int turnaround_time;
  int response_time;
  int priority;
} ProcessMetrics;

// Performance metrics for scheduling algorithms
typedef struct {
  char algorithm_name[64];
  double execution_time;          // Time to complete all processes
  int context_switches;           // Number of context switches
  double avg_waiting_time;        // Average waiting time
  double avg_turnaround_time;     // Average turnaround time
  double avg_response_time;       // Average response time
  double cpu_utilization;         // CPU utilization percentage
  double throughput;              // Processes completed per unit time

  // Memory metrics
  unsigned long l1_cache_hits;
  unsigned long l1_cache_misses;
  unsigned long l2_cache_hits;
  unsigned long l2_cache_misses;
  unsigned long write_backs;

  // Per-process metrics
  ProcessMetrics process_metrics[MAX_PROCESSES];
  int process_count;

  // Timing breakdown
  double scheduler_time;          // Time spent in scheduler
  double context_switch_time;     // Time spent context switching
  double execution_time_total;    // Total process execution time
  double idle_time;               // CPU idle time

  // System time tracking
  int start_time;
  int end_time;
  int total_burst_time;
} PerformanceMetrics;

// Global metrics storage
```

```c
typedef struct {
  PerformanceMetrics algorithms[MAX_ALGORITHMS];
  int algorithm_count;

  // System-wide cache stats (captured at start/end)
  unsigned long initial_l1_hits;
  unsigned long initial_l1_misses;
  unsigned long initial_l2_hits;
  unsigned long initial_l2_misses;
  unsigned long initial_write_backs;
} PerformanceTracker;

// Timer for measuring execution time
typedef struct {
  struct timespec start;
  struct timespec end;
  int is_running;
} PerfTimer;
```

These structures allowed stored all the information that we need to collect the performance metrics required.

10.2.2 Implementation of Time Tracking

To track time, we use a combination of the `time.h` and `sys/time.h` libraries, the `PerfTimer` structure and the following functions:

```c
void perf_timer_start(PerfTimer *timer) {
  if (!timer) return;

  clock_gettime(CLOCK_MONOTONIC, &timer->start);
  timer->is_running = 1;
}

double perf_timer_end(PerfTimer *timer) {
  if (!timer || !timer->is_running) return 0.0;

  clock_gettime(CLOCK_MONOTONIC, &timer->end);
  timer->is_running = 0;

  double elapsed = (timer->end.tv_sec - timer->start.tv_sec) * 1000.0;
  elapsed += (timer->end.tv_nsec - timer->start.tv_nsec) / 1000000.0;

  return elapsed;
}

double perf_timer_end_seconds(PerfTimer *timer) {
  return perf_timer_end(timer) / 1000.0;
}
```

10.2.3 Visualization and Reporting

Average Response Time Comparison

Figure 11 highlights how quickly a process receives its first CPU response after arrival. MLFQ and Round Robin have the lowest response times, making them well-suited for interactive systems. SRT also performs well, benefiting from preemption. SPN has a moderate response time due to its non-preemptive nature. FCFS, Priority, and HRRN show significantly higher response times, indicating slower initial responsiveness, especially under heavy workloads.

Average Turnaround Time Comparison

Figure 12 measures the total time from process arrival to completion. SRT again achieves the best performance, minimizing overall completion time by favoring short remaining jobs. SPN and Round Robin follow closely, offering balanced turnaround times. MLFQ performs moderately well due to its feedback-driven optimization.

Figure 11 Avg Response Time Comparison

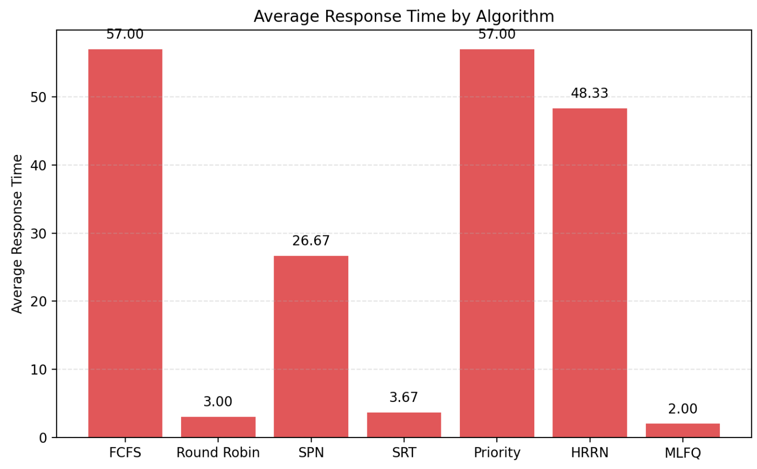
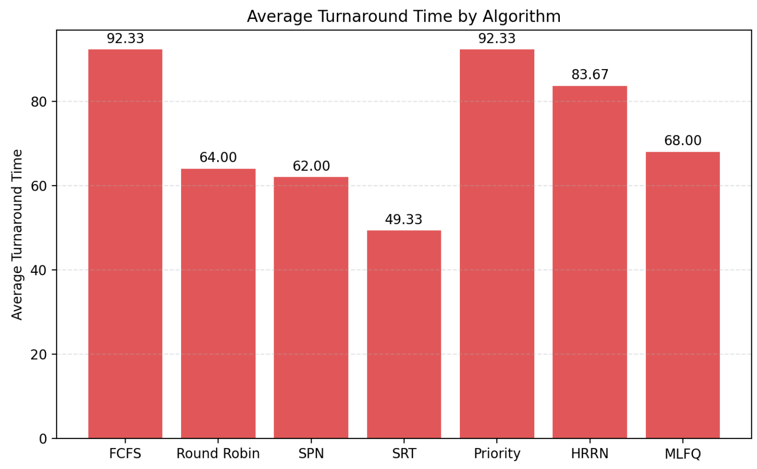


Figure 12 Avg Turnaround Time Comparison



FCFS and Priority scheduling have the highest turnaround times, reflecting inefficiencies caused by long waiting periods and poor handling of job length variability.

Average Wait Time Comparison

Figure 13 compares how long processes wait in the ready queue before execution under different scheduling algorithms. SRT (Shortest Remaining Time) performs best, with the lowest average waiting time, showing its efficiency in prioritizing shorter jobs dynamically. SPN (Shortest Process Next) and Round Robin also achieve relatively low waiting times, balancing fairness and efficiency. MLFQ falls in the mid-range, reflecting its adaptive but more complex scheduling behavior. FCFS and Priority scheduling show the highest waiting times, indicating potential inefficiencies such as convoy effects (FCFS) and starvation (Priority).

Context Switch Comparison

Figure 14 compares the number of context switches incurred by each scheduling algorithm during the simulation. Context switches represent the overhead associated with saving and restoring process state when the CPU switches between tasks, and higher values generally indicate increased scheduling overhead.

Round Robin exhibits a significantly higher number of context switches than all other algorithms. This is expected, as Round Robin relies on frequent preemption based on time quanta, causing the CPU to switch between processes often to ensure fairness. While this improves responsiveness, it introduces substantial overhead. In contrast, FCFS, SPN, Priority, and HRRN all show very low and nearly identical context switch counts. These algorithms are either non-preemptive or minimize preemption, resulting in fewer interruptions and lower scheduling overhead. SRT shows a slightly higher count due to its preemptive nature, but it remains far below Round Robin. MLFQ falls between the two extremes. Its moderate number of context switches reflects its adaptive, multi-level design, which balances responsiveness and efficiency by adjusting process priorities dynamically without excessive preemption.

Figure 13 Avg Wait Time Comparison

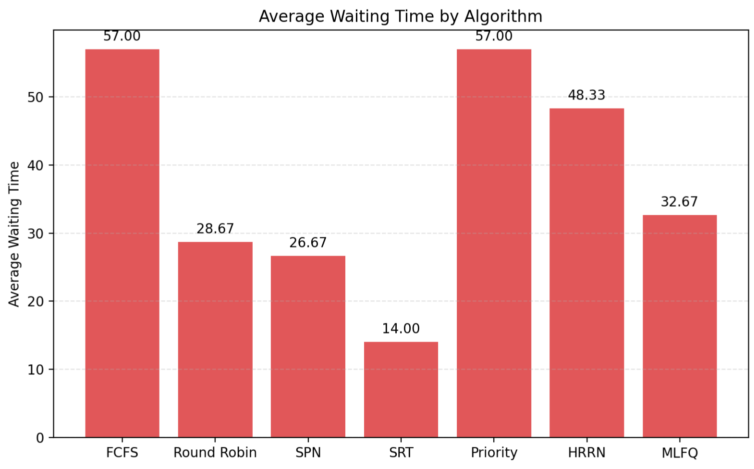
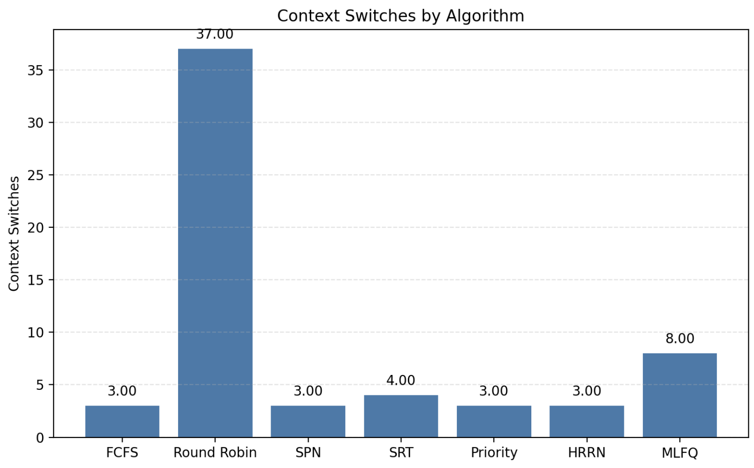


Figure 14 Context Switch Comparison



Overall, the results highlight the trade-off between responsiveness and overhead: highly preemptive algorithms like Round Robin incur more context switches, while simpler or non-preemptive algorithms achieve lower overhead at the cost of reduced responsiveness.

CPU Utilization Comparison

Figure 15 demonstrates that CPU utilization is consistently at 100% for all algorithms. This confirms that the simulator effectively keeps the CPU busy regardless of scheduling strategy. It also indicates that the workload is CPU-bound, meaning scheduling decisions influence how work is distributed among processes rather than how much idle time the CPU experiences.

L1 Cache: Hits vs Misses

Figure 16b depicts L1 cache hits steadily increase from FCFS to MLFQ. More advanced scheduling algorithms, such as HRRN and MLFQ, show significantly higher hit counts. This suggests that these algorithms execute more instructions overall or revisit cached data more frequently. While higher L1 hits are generally beneficial, they must be considered alongside miss rates and context switch overhead to fully evaluate cache efficiency.

Figure 16b mirrors the trend seen in L1 hits, with misses increasing as scheduling complexity increases. Algorithms like MLFQ and HRRN incur the highest miss counts, likely due to frequent preemption and context switching disrupting cache locality. This illustrates how aggressive multitasking can negatively impact cache performance, even when overall responsiveness improves.

L2 Cache: Hits vs Misses

Figure 17b depicts the L2 cache hits increase consistently across the algorithms, with MLFQ achieving the highest value. This indicates that while L1 locality may degrade, many memory accesses are still successfully

Figure 15 CPU Utilization Comparison

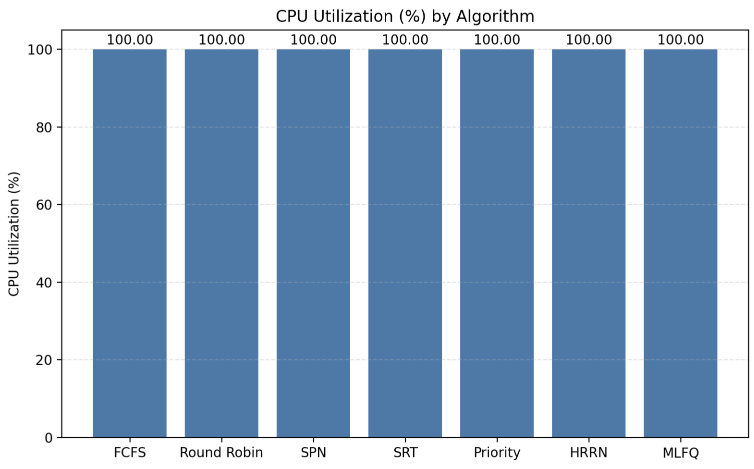
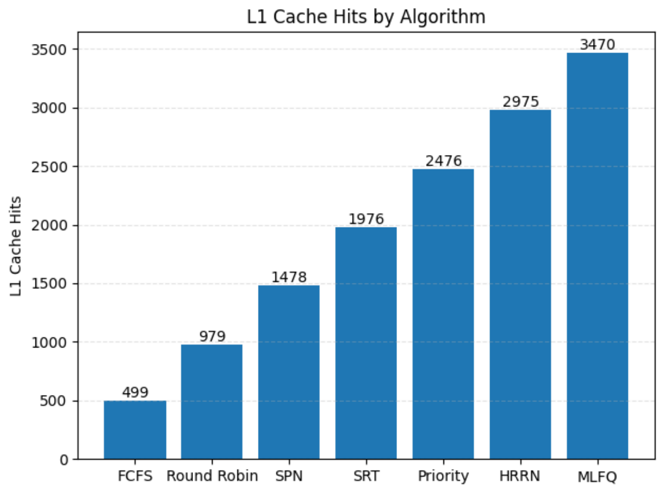
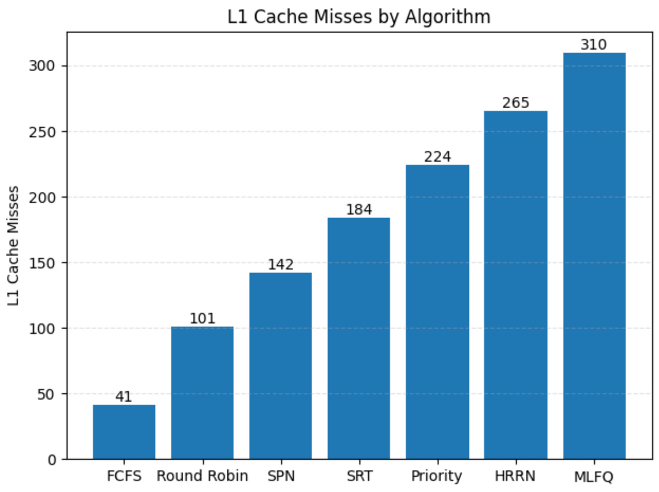


Figure 16 L1 Cache: Hits vs Misses



(a) L1 Cache Hits



(b) L1 Cache Misses

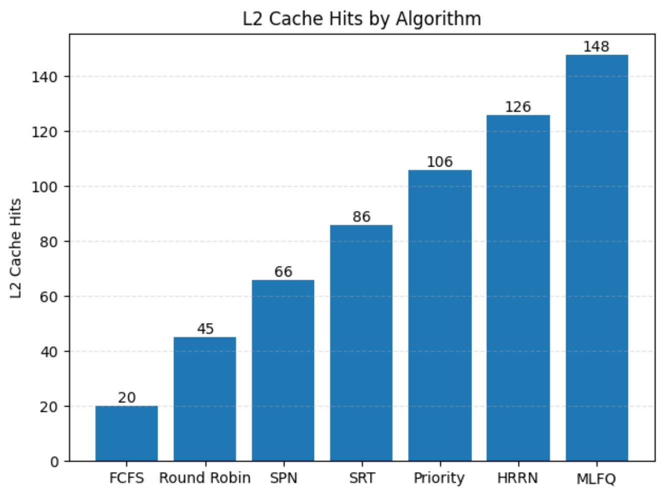
resolved at the L2 level. The trend suggests that deeper cache levels play an important role in mitigating the memory performance costs introduced by complex scheduling strategies.

Figure 17b shows the number of L2 cache misses for each scheduling algorithm. FCFS has the fewest misses, while MLFQ has the highest. As scheduling strategies become more dynamic and aggressive (e.g., HRRN and MLFQ), L2 cache misses increase, likely due to more frequent context switches and reduced temporal locality. This highlights the trade-off between responsiveness and cache efficiency, where advanced schedulers may improve fairness or response time at the cost of cache performance.

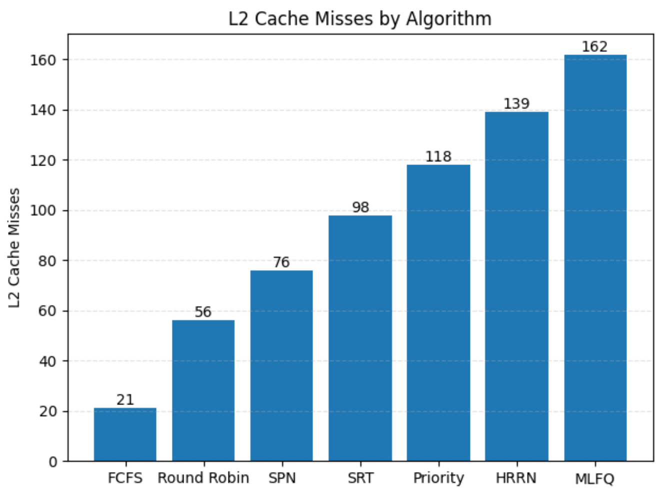
Throughput Comparison

Figure 18 indicates that all scheduling algorithms achieve nearly identical throughput. This suggests that, under the simulated workload, total completed work over time is largely unaffected by the choice of scheduling algorithm. Since CPU utilization is near 100% across all cases, throughput remains stable, reinforcing the idea that scheduling primarily affects latency-related metrics rather than overall system productivity in this scenario.

Figure 17 L2 Cache: Hits vs Misses

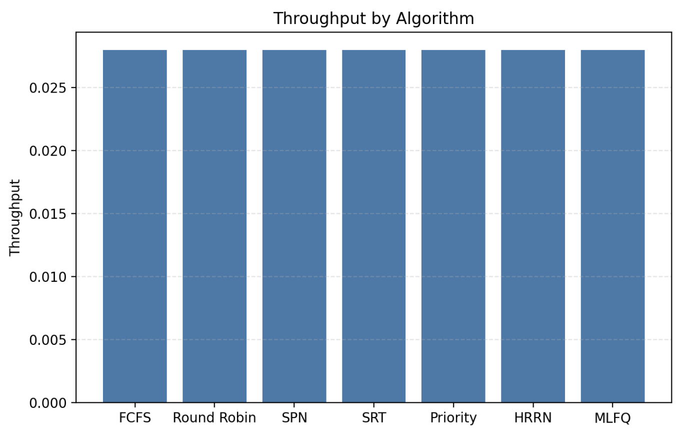


(a) L2 Cache Hits



(b) L2 Cache Misses

Figure 18 Throughput Comparison



11 The Simulation

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, and process management. It initializes all system components and assembles three programs (processes) — one prints “Hello, World” using CPU interrupts, another performs repeated arithmetic calculations and memory operations, and a third that is a combination of both I/O and CPU bound operations. 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 processes, interrupt-driven execution, and coordinated CPU-memory operations within a simulated environment.

```c
static void run_single_algorithm(SchedulingAlgorithm algo) {
  // Reinitialize queues for fresh run
  free_queues();
  init_queues();

  // Reset process storage
  reset_process_storage();

  // Recreate processes
  for (int i = 0; i < opts.program_count; i++) {
    if (!results[i].success) continue;

    uint32_t process_addr = makeProcess(
    i,
    results[i].program->entry_point,
    results[i].program->text_start,
    results[i].program->text_size,
    results[i].program->data_start,
    results[i].program->data_size,
    results[i].program->stack_ptr,
    opts.priorities[i],
    results[i].program->text_size // opts.burst_estimates[i]
    );

    if (process_addr == UINT32_MAX) {
      fprintf(stderr, "Failed to recreate process %d\n", i);
    }
  }

  // Run the scheduler
  scheduler(algo);
}

int main(int argc, char *argv[]) {
  int exit_code = EXIT_SUCCESS;

  signal(SIGSEGV, panic_handler);
  signal(SIGABRT, panic_handler);
  signal(SIGFPE, panic_handler);

  int panic_signal = setjmp(g_panic_buffer);
  if (panic_signal != 0) {
    fprintf(stderr, "Performing emergency cleanup after signal %d\n", panic_signal);
    exit_code = EXIT_FAILURE;
    goto cleanup;
  }

  parse_args(argc, argv);
```

```c
// In comparison mode, keep allocations around so the same
// text/data segments can be reused across multiple runs.
if (opts.compare_all_algorithms) {
  set_memory_freeze(true);
}

// Initialize memory system
printf("Initializing memory system...\n");
init_memory(opts.cache_policy);
memory_initialized = true;

// Initialize process queues
printf("Initializing process queues...\n");
init_queues();
queues_initialized = true;

// Initialize performance tracking
printf("Initializing performance tracking...\n");
init_performance_tracking();
perf_initialized = true;

// Allocate results array
results = calloc(opts.program_count, sizeof(AssemblyResult));
result_count = opts.program_count;
if(!results){
  fprintf(stderr, "Memory allocation failed for assembly results\n");
  exit_code = EXIT_FAILURE;
  goto cleanup;
}

// Assemble all programs
printf("\n=== Assembling Programs ===\n");
for (int i = 0; i < opts.program_count; i++) {
  printf("\n[%d/%d] Processing: %s\n", i+1, opts.program_count, opts.program_files[i]);

  results[i] = assemble(opts.program_files[i], i);

  if (!results[i].success) {
    fprintf(stderr, "  Assembly failed: %s\n", results[i].error_message);
    continue;
  }

  printf("   Assembly successful\n");
}

if (opts.compare_all_algorithms) {
  // Run all algorithms for comparison
  printf("\n");
  printf("================================================================================\n
");
  printf("           RUNNING ALL SCHEDULING ALGORITHMS FOR COMPARISON\n");
  printf("================================================================================\n
");

  SchedulingAlgorithm algorithms[] = {
    SCHED_FCFS,
    SCHED_ROUND_ROBIN,
    SCHED_SPN,
    SCHED_SRT,
    SCHED_PRIORITY,
    SCHED_HRRN,
    SCHED_MLFQ
  };
  const char *algo_names[] = {
    "FCFS",
    "Round Robin",
    "SPN (Shortest Process Next)",
    "SRT (Shortest Remaining Time)",
    "Priority",
    "HRRN (Highest Response Ratio Next)",
    "MLFQ (Multi-Level Feedback Queue)"
  };

  int num_algorithms = sizeof(algorithms) / sizeof(algorithms[0]);

  for (int i = 0; i < num_algorithms; i++) {
    printf("\n");
    printf("
********************************************************************************\n");
    printf("                    Running: %s\n", algo_names[i]);
    printf("
********************************************************************************\n");

    run_single_algorithm(algorithms[i]);

    printf("\n");
  }

  // Print comparison after all algorithms
  printf("\n");
  printf("################################################################################\n
");
  printf("#                        FINAL COMPARISON REPORT                               #\n
");
  printf("################################################################################\n
");
  print_comparison_table();

  if (opts.export_csv) {
    export_comparison_csv(opts.csv_filename);

    // Generate individual chart data files
    generate_chart_data("waiting_time", "chart_waiting_time.csv");
    generate_chart_data("cpu_util", "chart_cpu_utilization.csv");
    generate_chart_data("context_switches", "chart_context_switches.csv");
  }

} else {
  // Run single algorithm
  printf("\n");
  printf("================================================================================\n
");
  printf("                    Creating Processes for Scheduling\n");
  printf("================================================================================\n
");

  for (int i = 0; i < opts.program_count; i++) {
    if (!results[i].success) continue;

    uint32_t process_addr = makeProcess(
    i,
    results[i].program->entry_point,
    results[i].program->text_start,
    results[i].program->text_size,
    results[i].program->data_start,
    results[i].program->data_size,
    results[i].program->stack_ptr,
```

```c
    opts.priorities[i],
    opts.burst_estimates[i]
    );

    if (process_addr == UINT32_MAX) {
      fprintf(stderr, "   Failed to create process\n");
      exit_code = EXIT_FAILURE;
    } else {
      printf("   Process created (PID: %d, Entry: 0x%08x)\n",
      i, results[i].program->entry_point);
    }
  }

  // Run the scheduler
  printf("\n=== Starting Scheduler ===\n");
  printf("Algorithm: ");
  switch (opts.scheduler) {
    case SCHED_FCFS: printf("First-Come First-Served\n"); break;
    case SCHED_ROUND_ROBIN: printf("Round Robin\n"); break;
    case SCHED_PRIORITY: printf("Priority\n"); break;
    case SCHED_SRT: printf("Shortest Remaining Time\n"); break;
    case SCHED_HRRN: printf("Highest Response Ratio Next\n"); break;
    case SCHED_SPN: printf("Shortest Process Next\n"); break;
    case SCHED_MLFQ: printf("Multi-Level Feedback Queue\n"); break;
  }
  printf("\n");

  scheduler(opts.scheduler);
}

printf("\n=== Execution Complete ===\n");
print_cache_stats();

cleanup:
g_panic_handler_active = 1;

if (results) {
  for (int i = 0; i < result_count; i++) {
    if (results[i].success) {
      free_program(&results[i]);
    }
  }
  free(results);
  results = NULL;
}

if (opts.program_files) {
  free((void*)opts.program_files);
  opts.program_files = NULL;
}

if (opts.priorities) {
  free(opts.priorities);
  opts.priorities = NULL;
}

if (opts.burst_estimates) {
  free(opts.burst_estimates);
  opts.burst_estimates = NULL;
}

if (perf_initialized) {
  free_performance_tracking();
  perf_initialized = false;
```

```
    }

    if (memory_initialized) {
      free_memory();
      memory_initialized = false;
    }

    if (queues_initialized){
      free_queues();
      queues_initialized = false;
    }

    g_panic_handler_active = 0;
    return exit_code;
  }
```

And the output

```
Initializing memory system...
Initialized cache at -> '0x5c6a83b15e80' <- with | 64 | bytes, and | 1 | lines [Line Size = 64]
Initialized cache at -> '0x5c6a83b15e40' <- with | 128 | bytes, and | 2 | lines [Line Size = 64]
Memory initialized with write-through cache policy
Initializing process queues...
Initializing performance tracking...
Performance tracking initialized

=== Assembling Programs ===

[1/3] Processing: programs/factorial.asm
mallocate: PID 0 allocated 136 bytes [0 -> 135]
Allocated text: 0x00000000 (136 bytes)
mallocate: PID 0 allocated 22 bytes [136 -> 157]
 Allocated data: 0x00000088 (22 bytes)
mallocate: PID 0 allocated 4096 bytes [160 -> 4255]
 Assembly successful

[2/3] Processing: programs/goodbye_planet.asm
mallocate: PID 1 allocated 72 bytes [4256 -> 4327]
Allocated text: 0x000010a0 (72 bytes)
mallocate: PID 1 allocated 47 bytes [4328 -> 4374]
 Allocated data: 0x000010e8 (47 bytes)
mallocate: PID 1 allocated 4096 bytes [4376 -> 8471]
 Assembly successful

[3/3] Processing: programs/hello_world.asm
mallocate: PID 2 allocated 24 bytes [8472 -> 8495]
Allocated text: 0x00002118 (24 bytes)
mallocate: PID 2 allocated 15 bytes [8496 -> 8510]
 Allocated data: 0x00002130 (15 bytes)
mallocate: PID 2 allocated 4096 bytes [8512 -> 12607]
 Assembly successful

================================================================================
RUNNING ALL SCHEDULING ALGORITHMS FOR COMPARISON
================================================================================

********************************************************************************
Running: FCFS
********************************************************************************
 Process created:
```

```
PID: 0
PC:  0x00000000
SP:  0x0000109c
Text: 0x00000000 - 0x00000088 (136 bytes)
Data: 0x00000088 - 0x0000009e (22 bytes)
Priority: 1, Burst: 136
 Process created:
PID: 1
PC:  0x000010a0
SP:  0x00002114
Text: 0x000010a0 - 0x000010e8 (72 bytes)
Data: 0x000010e8 - 0x00001117 (47 bytes)
Priority: 2, Burst: 72
 Process created:
PID: 2
PC:  0x00002118
SP:  0x0000313c
Text: 0x00002118 - 0x00002130 (24 bytes)
Data: 0x00002130 - 0x0000213f (15 bytes)
Priority: 3, Burst: 24

=== Starting performance tracking for: FCFS ===

Scheduling algorithm: FCFS
Total 3 tasks to be scheduled
=============================
<system time 0> process 0 starts running
Factorial result: 120

<system time 70> process 0 finished.
<system time 70> process 1 starts running
Countdown: Countdown: 321Goodbye, Planet Earth!
<system time 101> process 1 finished.
<system time 101> process 2 starts running
Hello, World!
<system time 106> process 2 finished.
<system time 106> All processes finished.
=== Performance tracking completed for: FCFS ===


========================================================================
ALGORITHM: FCFS
========================================================================

Timing Metrics:
Total Execution Time:      106.000 time units
CPU Active Time:           106.000 time units
CPU Idle Time:             0.000 time units
Scheduler Overhead:        0.357 ms
Context Switch Overhead:   0.082 ms

Process Metrics:
Number of Processes:       3
Average Waiting Time:      57.000
Average Turnaround Time:   92.333
Average Response Time:     57.000
```

```
System Metrics:
CPU Utilization:           100.00%
Throughput:                0.028 processes/unit
Context Switches:          3

Memory Statistics:
L1 Cache Hits:             499
L1 Cache Misses:           41
L1 Hit Rate:               92.41%
L2 Cache Hits:             20
L2 Cache Misses:           21
L2 Hit Rate:               48.78%
Write-Backs:               0

PROCESS  ARRIVAL  BURST  COMPLETION  WAITING  TURNAROUND  RESPONSE  PRIORITY
===============================================================================
P0       0        70     70          0        70          0         1
P1       0        31     101         70       101         70        2
P2       0        5      106         101      106         101       3
===============================================================================



********************************************************************************
Running: Round Robin
********************************************************************************
 Process created:
PID: 0
PC:  0x00000000
SP:  0x0000109c
Text: 0x00000000 - 0x00000088 (136 bytes)
Data: 0x00000088 - 0x0000009e (22 bytes)
Priority: 1, Burst: 136
 Process created:
PID: 1
PC:  0x000010a0
SP:  0x00002114
Text: 0x000010a0 - 0x000010e8 (72 bytes)
Data: 0x000010e8 - 0x00001117 (47 bytes)
Priority: 2, Burst: 72
 Process created:
PID: 2
PC:  0x00002118
SP:  0x0000313c
Text: 0x00002118 - 0x00002130 (24 bytes)
Data: 0x00002130 - 0x0000213f (15 bytes)
Priority: 3, Burst: 24

=== Starting performance tracking for: Round Robin ===

Scheduling algorithm: Round Robin
Total 3 tasks to be scheduled
=============================
<system time 0> process 0 starts running
<system time 3> process 1 starts running
Countdown: Countdown: <system time 6> process 2 starts running
Hello, World!
```

```
<system time 9> process 0 starts running
<system time 12> process 1 starts running
<system time 15> process 2 starts running
<system time 17> process 2 finished.
<system time 17> process 0 starts running
<system time 20> process 1 starts running
3<system time 23> process 0 starts running
<system time 26> process 1 starts running
<system time 29> process 0 starts running
<system time 32> process 1 starts running
<system time 35> process 0 starts running
<system time 38> process 1 starts running
2<system time 41> process 0 starts running
<system time 44> process 1 starts running
<system time 47> process 0 starts running
<system time 50> process 1 starts running
1<system time 53> process 0 starts running
<system time 56> process 1 starts running
<system time 59> process 0 starts running
<system time 62> process 1 starts running
Goodbye, Planet Earth!
<system time 65> process 0 starts running
<system time 68> process 1 starts running
<system time 69> process 1 finished.
<system time 69> process 0 starts running
<system time 72> process 0 starts running
<system time 75> process 0 starts running
<system time 78> process 0 starts running
<system time 81> process 0 starts running
<system time 84> process 0 starts running
<system time 87> process 0 starts running
<system time 90> process 0 starts running
<system time 93> process 0 starts running
<system time 96> process 0 starts running
Factorial result: <system time 99> process 0 starts running
120<system time 102> process 0 starts running


<system time 105> process 0 starts running
<system time 106> process 0 finished.
<system time 106> All processes finished.
=== Performance tracking completed for: Round Robin ===


========================================================================
ALGORITHM: Round Robin
========================================================================

Timing Metrics:
Total Execution Time:      106.000 time units
CPU Active Time:           106.000 time units
CPU Idle Time:             0.000 time units
Scheduler Overhead:        0.504 ms
Context Switch Overhead:   0.065 ms

Process Metrics:
Number of Processes:       3
```

```
Average Waiting Time:      28.667
Average Turnaround Time:   64.000
Average Response Time:     3.000

System Metrics:
CPU Utilization:           100.00%
Throughput:                0.028 processes/unit
Context Switches:          37

Memory Statistics:
L1 Cache Hits:             979
L1 Cache Misses:           101
L1 Hit Rate:               90.65%
L2 Cache Hits:             45
L2 Cache Misses:           56
L2 Hit Rate:               44.55%
Write-Backs:               0

PROCESS  ARRIVAL  BURST  COMPLETION  WAITING  TURNAROUND  RESPONSE  PRIORITY
==============================================================================
P2       0        5      17          12       17          6         3
P1       0        31     69          38       69          3         2
P0       0        70     106         36       106         0         1
==============================================================================



*******************************************************************************
Running: SPN (Shortest Process Next)
*******************************************************************************
 Process created:
PID: 0
PC:  0x00000000
SP:  0x0000109c
Text: 0x00000000 - 0x00000088 (136 bytes)
Data: 0x00000088 - 0x0000009e (22 bytes)
Priority: 1, Burst: 136
 Process created:
PID: 1
PC:  0x000010a0
SP:  0x00002114
Text: 0x000010a0 - 0x000010e8 (72 bytes)
Data: 0x000010e8 - 0x00001117 (47 bytes)
Priority: 2, Burst: 72
 Process created:
PID: 2
PC:  0x00002118
SP:  0x0000313c
Text: 0x00002118 - 0x00002130 (24 bytes)
Data: 0x00002130 - 0x0000213f (15 bytes)
Priority: 3, Burst: 24

=== Starting performance tracking for: SPN ===

Scheduling algorithm: SPN (Shortest Process Next)
Total 3 tasks to be scheduled
=============================
<system time 0> process 2 starts running
Hello, World!
<system time 5> process 2 finished.
<system time 5> process 0 starts running
Factorial result: 120

<system time 75> process 0 finished.
<system time 75> process 1 starts running
Countdown: Countdown: 321Goodbye, Planet Earth!
<system time 106> process 1 finished.
<system time 106> All processes finished.
=== Performance tracking completed for: SPN ===


========================================================================
ALGORITHM: SPN
========================================================================

Timing Metrics:
Total Execution Time:      106.000 time units
CPU Active Time:           106.000 time units
CPU Idle Time:             0.000 time units
Scheduler Overhead:        0.208 ms
Context Switch Overhead:   0.159 ms

Process Metrics:
Number of Processes:       3
Average Waiting Time:      26.667
Average Turnaround Time:   62.000
Average Response Time:     26.667

System Metrics:
CPU Utilization:           100.00%
Throughput:                0.028 processes/unit
Context Switches:          3

Memory Statistics:
L1 Cache Hits:             1478
L1 Cache Misses:           142
L1 Hit Rate:               91.23%
L2 Cache Hits:             66
L2 Cache Misses:           76
L2 Hit Rate:               46.48%
Write-Backs:               0

PROCESS  ARRIVAL  BURST  COMPLETION  WAITING  TURNAROUND  RESPONSE  PRIORITY
===============================================================================
P2       0        5      5           0        5           0         3
P0       0        70     75          5        75          5         1
P1       0        31     106         75       106         75        2
===============================================================================



********************************************************************************
Running: SRT (Shortest Remaining Time)
********************************************************************************
```

```
 Process created:
PID: 0
PC:  0x00000000
SP:  0x0000109c
Text: 0x00000000 - 0x00000088 (136 bytes)
Data: 0x00000088 - 0x0000009e (22 bytes)
Priority: 1, Burst: 136
 Process created:
PID: 1
PC:  0x000010a0
SP:  0x00002114
Text: 0x000010a0 - 0x000010e8 (72 bytes)
Data: 0x000010e8 - 0x00001117 (47 bytes)
Priority: 2, Burst: 72
 Process created:
PID: 2
PC:  0x00002118
SP:  0x0000313c
Text: 0x00002118 - 0x00002130 (24 bytes)
Data: 0x00002130 - 0x0000213f (15 bytes)
Priority: 3, Burst: 24

=== Starting performance tracking for: SRT ===

Scheduling algorithm: SRT (Shortest Remaining Time)
Total 3 tasks to be scheduled
=============================
Hello, World!
<system time 5> process 2 finished.
Countdown: Countdown: 321Goodbye, Planet Earth!
<system time 37> process 1 finished.
Factorial result: 120

<system time 106> process 0 finished.
<system time 106> All processes finished.
=== Performance tracking completed for: SRT ===


========================================================================
ALGORITHM: SRT
========================================================================

Timing Metrics:
Total Execution Time:      106.000 time units
CPU Active Time:           106.000 time units
CPU Idle Time:             0.000 time units
Scheduler Overhead:        0.149 ms
Context Switch Overhead:   0.001 ms

Process Metrics:
Number of Processes:       3
Average Waiting Time:      14.000
Average Turnaround Time:   49.333
Average Response Time:     3.667

System Metrics:
CPU Utilization:           100.00%
```

```
Throughput:               0.028 processes/unit
Context Switches:         4

Memory Statistics:
L1 Cache Hits:            1976
L1 Cache Misses:          184
L1 Hit Rate:              91.48%
L2 Cache Hits:            86
L2 Cache Misses:          98
L2 Hit Rate:              46.74%
Write-Backs:              0

PROCESS  ARRIVAL  BURST  COMPLETION  WAITING  TURNAROUND  RESPONSE  PRIORITY
==============================================================================
P2       0        5      5           0        5           0         3
P1       0        31     37          6        37          6         2
P0       0        70     106         36       106         5         1
==============================================================================



********************************************************************************
Running: Priority
********************************************************************************
 Process created:
PID: 0
PC:  0x00000000
SP:  0x0000109c
Text: 0x00000000 - 0x00000088 (136 bytes)
Data: 0x00000088 - 0x0000009e (22 bytes)
Priority: 1, Burst: 136
 Process created:
PID: 1
PC:  0x000010a0
SP:  0x00002114
Text: 0x000010a0 - 0x000010e8 (72 bytes)
Data: 0x000010e8 - 0x00001117 (47 bytes)
Priority: 2, Burst: 72
 Process created:
PID: 2
PC:  0x00002118
SP:  0x0000313c
Text: 0x00002118 - 0x00002130 (24 bytes)
Data: 0x00002130 - 0x0000213f (15 bytes)
Priority: 3, Burst: 24

=== Starting performance tracking for: Priority ===

Scheduling algorithm: Priority
Total 3 tasks to be scheduled
=============================
Factorial result: 120

<system time 70> process 0 finished.
Countdown: Countdown: 321Goodbye, Planet Earth!
<system time 101> process 1 finished.
Hello, World!
```

```
<system time 106> process 2 finished.
<system time 106> All processes finished.
=== Performance tracking completed for: Priority ===


========================================================================
ALGORITHM: Priority
========================================================================

Timing Metrics:
Total Execution Time:      106.000 time units
CPU Active Time:           106.000 time units
CPU Idle Time:             0.000 time units
Scheduler Overhead:        0.235 ms
Context Switch Overhead:   0.000 ms

Process Metrics:
Number of Processes:       3
Average Waiting Time:      57.000
Average Turnaround Time:   92.333
Average Response Time:     57.000

System Metrics:
CPU Utilization:           100.00%
Throughput:                0.028 processes/unit
Context Switches:          3

Memory Statistics:
L1 Cache Hits:             2476
L1 Cache Misses:           224
L1 Hit Rate:               91.70%
L2 Cache Hits:             106
L2 Cache Misses:           118
L2 Hit Rate:               47.32%
Write-Backs:               0

PROCESS  ARRIVAL  BURST  COMPLETION  WAITING  TURNAROUND  RESPONSE  PRIORITY
===============================================================================
P0       0        70     70          0        70          0         1
P1       0        31     101         70       101         70        2
P2       0        5      106         101      106         101       3
===============================================================================



********************************************************************************
Running: HRRN (Highest Response Ratio Next)
********************************************************************************
 Process created:
PID: 0
PC:  0x00000000
SP:  0x0000109c
Text: 0x00000000 - 0x00000088 (136 bytes)
Data: 0x00000088 - 0x0000009e (22 bytes)
Priority: 1, Burst: 136
 Process created:
PID: 1
```

```
PC:  0x000010a0
SP:  0x00002114
Text: 0x000010a0 - 0x000010e8 (72 bytes)
Data: 0x000010e8 - 0x00001117 (47 bytes)
Priority: 2, Burst: 72
 Process created:
PID: 2
PC:  0x00002118
SP:  0x0000313c
Text: 0x00002118 - 0x00002130 (24 bytes)
Data: 0x00002130 - 0x0000213f (15 bytes)
Priority: 3, Burst: 24

=== Starting performance tracking for: HRRN ===

Scheduling algorithm: HRRN (Highest Response Ratio Next)
Total 3 tasks to be scheduled
=============================
<system time 0> process 0 starts running
Factorial result: 120

<system time 70> process 0 finished.
<system time 70> process 2 starts running
Hello, World!
<system time 75> process 2 finished.
<system time 75> process 1 starts running
Countdown: Countdown: 321Goodbye, Planet Earth!
<system time 106> process 1 finished.
<system time 106> All processes finished.
=== Performance tracking completed for: HRRN ===


========================================================================
ALGORITHM: HRRN
========================================================================

Timing Metrics:
Total Execution Time:      106.000 time units
CPU Active Time:           106.000 time units
CPU Idle Time:             0.000 time units
Scheduler Overhead:        0.269 ms
Context Switch Overhead:   0.200 ms

Process Metrics:
Number of Processes:       3
Average Waiting Time:      48.333
Average Turnaround Time:   83.667
Average Response Time:     48.333

System Metrics:
CPU Utilization:           100.00%
Throughput:                0.028 processes/unit
Context Switches:          3

Memory Statistics:
L1 Cache Hits:             2975
L1 Cache Misses:           265
```

```
L1 Hit Rate:             91.82%
L2 Cache Hits:           126
L2 Cache Misses:         139
L2 Hit Rate:             47.55%
Write-Backs:             0

PROCESS  ARRIVAL  BURST  COMPLETION  WAITING  TURNAROUND  RESPONSE  PRIORITY
==============================================================================
P0       0        70     70          0        70          0         1
P2       0        5      75          70       75          70        3
P1       0        31     106         75       106         75        2
==============================================================================



*******************************************************************************
Running: MLFQ (Multi-Level Feedback Queue)
*******************************************************************************
 Process created:
PID: 0
PC:  0x00000000
SP:  0x0000109c
Text: 0x00000000 - 0x00000088 (136 bytes)
Data: 0x00000088 - 0x0000009e (22 bytes)
Priority: 1, Burst: 136
 Process created:
PID: 1
PC:  0x000010a0
SP:  0x00002114
Text: 0x000010a0 - 0x000010e8 (72 bytes)
Data: 0x000010e8 - 0x00001117 (47 bytes)
Priority: 2, Burst: 72
 Process created:
PID: 2
PC:  0x00002118
SP:  0x0000313c
Text: 0x00002118 - 0x00002130 (24 bytes)
Data: 0x00002130 - 0x0000213f (15 bytes)
Priority: 3, Burst: 24

=== Starting performance tracking for: MLFQ ===

Scheduling algorithm: MLFQ (Multi-Level Feedback Queue)
Total processes to be scheduled
=============================
Countdown: Countdown: Hello, World!
<system time 17> process 2 finished.
Factorial result: 120

<system time 81> process 0 finished.
321Goodbye, Planet Earth!
<system time 106> process 1 finished.
<system time 106> All processes finished.
=== Performance tracking completed for: MLFQ ===


========================================================================
ALGORITHM: MLFQ
========================================================================

Timing Metrics:
Total Execution Time:      106.000 time units
CPU Active Time:           106.000 time units
CPU Idle Time:             0.000 time units
Scheduler Overhead:        0.397 ms
Context Switch Overhead:   0.116 ms

Process Metrics:
Number of Processes:       3
Average Waiting Time:      32.667
Average Turnaround Time:   68.000
Average Response Time:     2.000

System Metrics:
CPU Utilization:           100.00%
Throughput:                0.028 processes/unit
Context Switches:          8

Memory Statistics:
L1 Cache Hits:             3470
L1 Cache Misses:           310
L1 Hit Rate:               91.80%
L2 Cache Hits:             148
L2 Cache Misses:           162
L2 Hit Rate:               47.74%
Write-Backs:               0

PROCESS  ARRIVAL  BURST  COMPLETION  WAITING  TURNAROUND  RESPONSE  PRIORITY
===============================================================================
P2       0        5      17          12       17          4         3
P0       0        70     81          11       81          0         1
P1       0        31     106         75       106         2         2
===============================================================================



################################################################################
#                         FINAL COMPARISON REPORT                              #
################################################################################

=====================================================================================================
SCHEDULING ALGORITHM COMPARISON
=====================================================================================================
Algorithm          Avg Wait Avg T.Around     Avg Resp      CPU%%   C.Switches
-----------------------------------------------------------------------------------------------------
FCFS                 57.000       92.333       57.000    100.00%            3
Round Robin          28.667       64.000        3.000    100.00%           37
SPN                  26.667       62.000       26.667    100.00%            3
SRT                  14.000       49.333        3.667    100.00%            4
Priority             57.000       92.333       57.000    100.00%            3
HRRN                 48.333       83.667       48.333    100.00%            3
MLFQ                 32.667       68.000        2.000    100.00%            8
=====================================================================================================
Best Performers:
Lowest Average Waiting Time:    SRT (14.000)
Lowest Average Turnaround Time: SRT (49.333)
Lowest Average Response Time:   MLFQ (2.000)
Highest CPU Utilization:        FCFS (100.00%)
Fewest Context Switches:        FCFS (3)


=== Execution Complete ===

=== Cache Statistics ===
L1 Cache:
Hits:   3470
Misses: 310
Hit Rate: 91.80%

L2 Cache:
Hits:   148
Misses: 162
Hit Rate: 47.74%
========================
```

12 Testing and Debugging

This test program evaluates the complete functionality and integration of the simulated computer system, including CPU execution, memory management, interrupt handling and process schedulers. It launches three concurrent processes: one that prints a message using CPU interrupts, another that performs intensive arithmetic operations and memory storage, and a third that simulates process being scheduled and completed by the process schedulers. Meanwhile, separate threads generate timer and I/O interrupts to test asynchronous event handling and process coordination. The system also runs a demo CPU program to verify correct instruction execution, memory access, and cache behavior. Overall, this test assesses whether all system components—CPU, memory hierarchy, interrupt controller, and process handling are work together smoothly to emulate a functioning multitasking operating system environment.

```c
static void reset_cpu_and_memory(void) {
  memset(&THE_CPU, 0, sizeof(Cpu));
  free_memory();
  init_memory(CACHE_WRITE_THROUGH);
  set_current_process(SYSTEM_PROCESS_ID);
}

// ==========================================
// CPU Initialization Tests
// ==========================================

TEST_CASE(CPU, InitializationSetsPC) {
  reset_cpu_and_memory();
  init_cpu(0x1000);
  ASSERT_EQ(HW_REGISTER(PC), 0x1000);
}

TEST_CASE(CPU, InitializationSetsZeroFlag) {
  reset_cpu_and_memory();
  init_cpu(0x2000);
  ASSERT_TRUE(HW_REGISTER(FLAGS) & F_ZERO);
}

TEST_CASE(CPU, InitializationClearsRegisters) {
  reset_cpu_and_memory();
  init_cpu(0x3000);
  for (int i = 1; i < GP_REG_COUNT; i++) {
    ASSERT_EQ(GP_REGISTER(i), 0);
  }
}

TEST_CASE(CPU, InitializationSetsZeroRegister) {
  reset_cpu_and_memory();
  init_cpu(0x4000);
  ASSERT_EQ(GP_REGISTER(REG_ZERO), 0);
}

// ==========================================
// Fetch Tests
// ==========================================

TEST_CASE(CPU, FetchLoadsInstructionFromMemory) {
  reset_cpu_and_memory();
  uint32_t test_addr = 0x1000;
  uint32_t test_instruction = 0x12345678;

  write_word(test_addr, test_instruction);
  init_cpu(test_addr);
  fetch();

  ASSERT_EQ(HW_REGISTER(IR), test_instruction);
}
```

```
TEST_CASE(CPU, FetchIncrementsPC) {
  reset_cpu_and_memory();
  uint32_t start_addr = 0x2000;
  write_word(start_addr, 0xAAAAAAAA);

  init_cpu(start_addr);
  fetch();

  ASSERT_EQ(HW_REGISTER(PC), start_addr + 4);
}

TEST_CASE(CPU, FetchMultipleInstructions) {
  reset_cpu_and_memory();
  uint32_t base_addr = 0x3000;

  write_word(base_addr, 0x11111111);
  write_word(base_addr + 4, 0x22222222);
  write_word(base_addr + 8, 0x33333333);

  init_cpu(base_addr);

  fetch();
  ASSERT_EQ(HW_REGISTER(IR), 0x11111111);
  ASSERT_EQ(HW_REGISTER(PC), base_addr + 4);

  fetch();
  ASSERT_EQ(HW_REGISTER(IR), 0x22222222);
  ASSERT_EQ(HW_REGISTER(PC), base_addr + 8);

  fetch();
  ASSERT_EQ(HW_REGISTER(IR), 0x33333333);
  ASSERT_EQ(HW_REGISTER(PC), base_addr + 12);
}

// ==========================================
// Register Access Tests
// ==========================================

TEST_CASE(CPU, GeneralPurposeRegisterReadWrite) {
  reset_cpu_and_memory();
  GP_REGISTER(REG_T0) = 0xDEADBEEF;
  ASSERT_EQ(GP_REGISTER(REG_T0), 0xDEADBEEF);
}

TEST_CASE(CPU, AllGPRegistersIndependent) {
  reset_cpu_and_memory();
  for (int i = 1; i < GP_REG_COUNT; i++) {
    GP_REGISTER(i) = i * 0x11;
  }
  for (int i = 1; i < GP_REG_COUNT; i++) {
    ASSERT_EQ(GP_REGISTER(i), (uint32_t)(i * 0x11));
  }
}

TEST_CASE(CPU, HardwareRegisterAccess) {
  reset_cpu_and_memory();
  HW_REGISTER(PC) = 0x5000;
  HW_REGISTER(IR) = 0xABCDEF12;
  HW_REGISTER(FLAGS) = F_ZERO | F_CARRY;

  ASSERT_EQ(HW_REGISTER(PC), 0x5000);
  ASSERT_EQ(HW_REGISTER(IR), 0xABCDEF12);
  ASSERT_EQ(HW_REGISTER(FLAGS), (uint32_t)(F_ZERO | F_CARRY));
```

```
}

TEST_CASE(CPU, StackPointerRegister) {
  reset_cpu_and_memory();
  GP_REGISTER(REG_SP) = 0x7FFFFFFC;
  ASSERT_EQ(GP_REGISTER(REG_SP), 0x7FFFFFFC);

  GP_REGISTER(REG_SP) -= 4;
  ASSERT_EQ(GP_REGISTER(REG_SP), 0x7FFFFFF8);
}

TEST_CASE(CPU, ReturnAddressRegister) {
  reset_cpu_and_memory();
  GP_REGISTER(REG_RA) = 0x1234;
  ASSERT_EQ(GP_REGISTER(REG_RA), 0x1234);
}

// ==========================================
// Flag Tests
// ==========================================

TEST_CASE(CPU, ZeroFlagSetAndClear) {
  reset_cpu_and_memory();
  HW_REGISTER(FLAGS) = 0;
  HW_REGISTER(FLAGS) |= F_ZERO;
  ASSERT_TRUE(HW_REGISTER(FLAGS) & F_ZERO);

  HW_REGISTER(FLAGS) &= ~F_ZERO;
  ASSERT_TRUE(!(HW_REGISTER(FLAGS) & F_ZERO));
}

TEST_CASE(CPU, CarryFlagSetAndClear) {
  reset_cpu_and_memory();
  HW_REGISTER(FLAGS) = 0;
  HW_REGISTER(FLAGS) |= F_CARRY;
  ASSERT_TRUE(HW_REGISTER(FLAGS) & F_CARRY);

  HW_REGISTER(FLAGS) &= ~F_CARRY;
  ASSERT_TRUE(!(HW_REGISTER(FLAGS) & F_CARRY));
}

TEST_CASE(CPU, OverflowFlagSetAndClear) {
  reset_cpu_and_memory();
  HW_REGISTER(FLAGS) = 0;
  HW_REGISTER(FLAGS) |= F_OVERFLOW;
  ASSERT_TRUE(HW_REGISTER(FLAGS) & F_OVERFLOW);

  HW_REGISTER(FLAGS) &= ~F_OVERFLOW;
  ASSERT_TRUE(!(HW_REGISTER(FLAGS) & F_OVERFLOW));
}

TEST_CASE(CPU, MultipleFlagsSimultaneous) {
  reset_cpu_and_memory();
  HW_REGISTER(FLAGS) = F_ZERO | F_CARRY | F_OVERFLOW;

  ASSERT_TRUE(HW_REGISTER(FLAGS) & F_ZERO);
  ASSERT_TRUE(HW_REGISTER(FLAGS) & F_CARRY);
  ASSERT_TRUE(HW_REGISTER(FLAGS) & F_OVERFLOW);
}

TEST_CASE(CPU, ClearAllFlags) {
  reset_cpu_and_memory();
  HW_REGISTER(FLAGS) = F_ZERO | F_CARRY | F_OVERFLOW;
  HW_REGISTER(FLAGS) = 0;
  ASSERT_EQ(HW_REGISTER(FLAGS), 0);
}

// ================================================
// HI/LO Register Tests
// ================================================

TEST_CASE(CPU, HIRegisterAccess) {
  reset_cpu_and_memory();
  HW_REGISTER(HI) = 0x12345678;
  ASSERT_EQ(HW_REGISTER(HI), 0x12345678);
}

TEST_CASE(CPU, LORegisterAccess) {
  reset_cpu_and_memory();
  HW_REGISTER(LO) = 0xABCDEF00;
  ASSERT_EQ(HW_REGISTER(LO), 0xABCDEF00);
}

TEST_CASE(CPU, HILOIndependentAccess) {
  reset_cpu_and_memory();
  HW_REGISTER(HI) = 0x11111111;
  HW_REGISTER(LO) = 0x22222222;

  ASSERT_EQ(HW_REGISTER(HI), 0x11111111);
  ASSERT_EQ(HW_REGISTER(LO), 0x22222222);
}

// ================================================
// Fetch-Execute Cycle Integration Tests
// ================================================

TEST_CASE(CPU, SimpleFetchExecuteCycle) {
  reset_cpu_and_memory();
  uint32_t addr = 0x1000;

  // NOP instruction (sll zero, zero, 0)
  uint32_t nop = 0x00000000;
  write_word(addr, nop);

  init_cpu(addr);
  fetch();
  execute();

  ASSERT_EQ(HW_REGISTER(PC), addr + 4);
}

TEST_CASE(CPU, ExecutePreservesNonTargetRegisters) {
  reset_cpu_and_memory();

  // Set up registers
  GP_REGISTER(REG_T0) = 0xAAAAAAAA;
  GP_REGISTER(REG_T1) = 0xBBBBBBBB;
  GP_REGISTER(REG_T2) = 0xCCCCCCCC;

  // Execute NOP
  HW_REGISTER(IR) = 0x00000000;
  execute();

  // T0, T1, T2 should be unchanged
  ASSERT_EQ(GP_REGISTER(REG_T0), 0xAAAAAAAA);
  ASSERT_EQ(GP_REGISTER(REG_T1), 0xBBBBBBBB);
  ASSERT_EQ(GP_REGISTER(REG_T2), 0xCCCCCCCC);
```

```
}

// ============================================
// Memory Access Through CPU Tests
// ============================================

TEST_CASE(CPU, CPUMemoryReadWrite) {
  reset_cpu_and_memory();
  uint32_t addr = 0x2000;
  uint32_t value = 0x12345678;

  write_word(addr, value);
  ASSERT_EQ(read_word(addr), value);
}

TEST_CASE(CPU, InstructionFetchFromDifferentAddresses) {
  reset_cpu_and_memory();

  write_word(0x1000, 0x11111111);
  write_word(0x2000, 0x22222222);
  write_word(0x3000, 0x33333333);

  init_cpu(0x1000);
  fetch();
  ASSERT_EQ(HW_REGISTER(IR), 0x11111111);

  HW_REGISTER(PC) = 0x2000;
  fetch();
  ASSERT_EQ(HW_REGISTER(IR), 0x22222222);

  HW_REGISTER(PC) = 0x3000;
  fetch();
  ASSERT_EQ(HW_REGISTER(IR), 0x33333333);
}

// ============================================
// Register Naming Tests
// ============================================

TEST_CASE(CPU, ArgumentRegisters) {
  reset_cpu_and_memory();
  GP_REGISTER(REG_A0) = 1;
  GP_REGISTER(REG_A1) = 2;
  GP_REGISTER(REG_A2) = 3;
  GP_REGISTER(REG_A3) = 4;

  ASSERT_EQ(GP_REGISTER(REG_A0), 1);
  ASSERT_EQ(GP_REGISTER(REG_A1), 2);
  ASSERT_EQ(GP_REGISTER(REG_A2), 3);
  ASSERT_EQ(GP_REGISTER(REG_A3), 4);
}

TEST_CASE(CPU, ReturnValueRegisters) {
  reset_cpu_and_memory();
  GP_REGISTER(REG_V0) = 0x100;
  GP_REGISTER(REG_V1) = 0x200;

  ASSERT_EQ(GP_REGISTER(REG_V0), 0x100);
  ASSERT_EQ(GP_REGISTER(REG_V1), 0x200);
}

TEST_CASE(CPU, TemporaryRegisters) {
  reset_cpu_and_memory();
  for (int i = REG_T0; i <= REG_T7; i++) {
    GP_REGISTER(i) = i * 0x10;
  }
  for (int i = REG_T0; i <= REG_T7; i++) {
    ASSERT_EQ(GP_REGISTER(i), (uint32_t)(i * 0x10));
  }
}

TEST_CASE(CPU, SavedRegisters) {
  reset_cpu_and_memory();
  for (int i = REG_S0; i <= REG_S7; i++) {
    GP_REGISTER(i) = i * 0x100;
  }
  for (int i = REG_S0; i <= REG_S7; i++) {
    ASSERT_EQ(GP_REGISTER(i), (uint32_t)(i * 0x100));
  }
}

// ==========================================
// Edge Cases
// ==========================================

TEST_CASE(CPU, PCWrapAround) {
  reset_cpu_and_memory();
  HW_REGISTER(PC) = 0xFFFFFFFC;
  // This would wrap on increment, but we test the value itself
  ASSERT_EQ(HW_REGISTER(PC), 0xFFFFFFFC);
}

TEST_CASE(CPU, MaximumRegisterValue) {
  reset_cpu_and_memory();
  GP_REGISTER(REG_T0) = 0xFFFFFFFF;
  ASSERT_EQ(GP_REGISTER(REG_T0), 0xFFFFFFFF);
}

TEST_CASE(CPU, MinimumRegisterValue) {
  reset_cpu_and_memory();
  GP_REGISTER(REG_T0) = 0x00000000;
  ASSERT_EQ(GP_REGISTER(REG_T0), 0x00000000);
}

// ==========================================
// State Consistency Tests
// ==========================================

TEST_CASE(CPU, RepeatedInitialization) {
  reset_cpu_and_memory();

  init_cpu(0x1000);
  ASSERT_EQ(HW_REGISTER(PC), 0x1000);

  init_cpu(0x2000);
  ASSERT_EQ(HW_REGISTER(PC), 0x2000);

  init_cpu(0x3000);
  ASSERT_EQ(HW_REGISTER(PC), 0x3000);
}

TEST_CASE(CPU, StateAfterMultipleOperations) {
  reset_cpu_and_memory();

  init_cpu(0x1000);
  GP_REGISTER(REG_T0) = 42;
  HW_REGISTER(FLAGS) = F_ZERO;
  ASSERT_EQ(HW_REGISTER(PC), 0x1000);
  ASSERT_EQ(GP_REGISTER(REG_T0), 42);
  ASSERT_TRUE(HW_REGISTER(FLAGS) & F_ZERO);
}

// ==================================================
// MAR and MBR Tests
// ==================================================

TEST_CASE(CPU, MemoryAddressRegister) {
  reset_cpu_and_memory();
  HW_REGISTER(MAR) = 0x5000;
  ASSERT_EQ(HW_REGISTER(MAR), 0x5000);
}

TEST_CASE(CPU, MemoryBufferRegister) {
  reset_cpu_and_memory();
  HW_REGISTER(MBR) = 0xDEADBEEF;
  ASSERT_EQ(HW_REGISTER(MBR), 0xDEADBEEF);
}

// ==================================================
// I/O Register Tests
// ==================================================

TEST_CASE(CPU, IOAddressRegister) {
  reset_cpu_and_memory();
  HW_REGISTER(IO_AR) = 0x100;
  ASSERT_EQ(HW_REGISTER(IO_AR), 0x100);
}

TEST_CASE(CPU, IOBufferRegister) {
  reset_cpu_and_memory();
  HW_REGISTER(IO_BR) = 0xFF;
  ASSERT_EQ(HW_REGISTER(IO_BR), 0xFF);
}

static void reset_memory(void) {
  free_memory();
  init_memory(CACHE_WRITE_THROUGH);
  set_current_process(SYSTEM_PROCESS_ID);
}

static void reset_memory_write_back(void) {
  free_memory();
  init_memory(CACHE_WRITE_BACK);
  set_current_process(SYSTEM_PROCESS_ID);
}

// ==================================================
// Basic Read/Write Tests
// ==================================================

TEST_CASE(Memory, ReadWriteByte) {
  reset_memory();
  write_byte(100, 0x42);
  ASSERT_EQ(read_byte(100), 0x42);
}

TEST_CASE(Memory, ReadWriteHalfword) {
  reset_memory();
  write_hword(200, 0x1234);
  ASSERT_EQ(read_hword(200), 0x1234);
}
```

```
TEST_CASE(Memory, ReadWriteWord) {
  reset_memory();
  write_word(300, 0x12345678);
  ASSERT_EQ(read_word(300), 0x12345678);
}

TEST_CASE(Memory, WriteBytePreservesOtherBytes) {
  reset_memory();
  write_word(400, 0xAABBCCDD);
  write_byte(401, 0xFF);
  ASSERT_EQ(read_word(400), 0xAABBFFDD);
}

TEST_CASE(Memory, WriteHalfwordPreservesOtherBytes) {
  reset_memory();
  write_word(500, 0x11223344);
  write_hword(502, 0xAABB);
  ASSERT_EQ(read_word(500), 0xAABB3344);
}

TEST_CASE(Memory, MultipleWordWrites) {
  reset_memory();
  for (uint32_t i = 0; i < 10; i++) {
    write_word(1000 + (i * 4), i * 100);
  }
  for (uint32_t i = 0; i < 10; i++) {
    ASSERT_EQ(read_word(1000 + (i * 4)), i * 100);
  }
}

// ============================================
// Cache Tests (Write-Through)
// ============================================

TEST_CASE(Memory, CacheWriteThroughUpdatesRAM) {
  reset_memory();
  write_word(2000, 0xDEADBEEF);
  ASSERT_EQ(read_word(2000), 0xDEADBEEF);
}

TEST_CASE(Memory, CacheHitOnRepeatedRead) {
  reset_memory();
  write_word(3000, 0x12345678);
  uint32_t first = read_word(3000);
  uint32_t second = read_word(3000);
  ASSERT_EQ(first, second);
  ASSERT_EQ(first, 0x12345678);
}

TEST_CASE(Memory, CacheLineBoundary) {
  reset_memory();
  // Write at cache line boundary (64 bytes)
  write_word(0, 0xAAAAAAAA);
  write_word(64, 0xBBBBBBBB);
  ASSERT_EQ(read_word(0), 0xAAAAAAAA);
  ASSERT_EQ(read_word(64), 0xBBBBBBBB);
}

TEST_CASE(Memory, CacheWriteThroughConsistency) {
  reset_memory();
  uint32_t addr = 4000;
  write_word(addr, 0x11111111);
  write_word(addr, 0x22222222);
  write_word(addr, 0x33333333);
  ASSERT_EQ(read_word(addr), 0x33333333);
}

// ==============================================
// Cache Tests (Write-Back)
// ==============================================

TEST_CASE(Memory, WriteBackDelaysRAMWrite) {
  reset_memory_write_back();
  write_word(5000, 0xFEEDFACE);
  ASSERT_EQ(read_word(5000), 0xFEEDFACE);
}

TEST_CASE(Memory, WriteBackMultipleWrites) {
  reset_memory_write_back();
  uint32_t addr = 6000;
  write_word(addr, 0x11111111);
  write_word(addr, 0x22222222);
  write_word(addr, 0x33333333);
  ASSERT_EQ(read_word(addr), 0x33333333);
}

TEST_CASE(Memory, WriteBackCacheLineEviction) {
  reset_memory_write_back();
  // Write enough data to force cache eviction
  for (uint32_t i = 0; i < 100; i++) {
    write_word(7000 + (i * 64), i);
  }
  for (uint32_t i = 0; i < 100; i++) {
    ASSERT_EQ(read_word(7000 + (i * 64)), i);
  }
}

// ==============================================
// Memory Allocation Tests
// ==============================================

TEST_CASE(Memory, AllocateMemoryForProcess) {
  reset_memory();
  uint32_t addr = mallocate(1, 1024);
  ASSERT_TRUE(addr != UINT32_MAX);
}

TEST_CASE(Memory, AllocateMultipleBlocks) {
  reset_memory();
  uint32_t addr1 = mallocate(1, 512);
  uint32_t addr2 = mallocate(2, 512);
  ASSERT_TRUE(addr1 != UINT32_MAX);
  ASSERT_TRUE(addr2 != UINT32_MAX);
  ASSERT_TRUE(addr1 != addr2);
}

TEST_CASE(Memory, AllocateZeroSizeFails) {
  reset_memory();
  uint32_t addr = mallocate(1, 0);
  ASSERT_EQ(addr, UINT32_MAX);
}

TEST_CASE(Memory, FreeAllocatedMemory) {
  reset_memory();
  uint32_t addr = mallocate(1, 1024);
  ASSERT_TRUE(addr != UINT32_MAX);
  liberate(1);
  // Should be able to allocate same size again
  uint32_t addr2 = mallocate(2, 1024);
  ASSERT_TRUE(addr2 != UINT32_MAX);
}

TEST_CASE(Memory, BestFitAllocation) {
  reset_memory();
  // Allocate and free to create fragmentation
  uint32_t addr1 = mallocate(1, 512);
  uint32_t addr2 = mallocate(2, 1024);
  uint32_t addr3 = mallocate(3, 512);

  ASSERT_TRUE(addr1 != UINT32_MAX);
  ASSERT_TRUE(addr2 != UINT32_MAX);
  ASSERT_TRUE(addr3 != UINT32_MAX);

  liberate(2); // Free middle block

  // Should fit in freed space
  uint32_t addr4 = mallocate(4, 512);
  ASSERT_TRUE(addr4 != UINT32_MAX);
}

TEST_CASE(Memory, AllocateAfterMultipleFrees) {
  reset_memory();
  uint32_t addr1 = mallocate(1, 256);
  uint32_t addr2 = mallocate(2, 256);
  uint32_t addr3 = mallocate(3, 256);

  liberate(1);
  liberate(2);
  liberate(3);
  (void)addr1;
  (void)addr2;
  (void)addr3;

  uint32_t addr4 = mallocate(4, 512);
  ASSERT_TRUE(addr4 != UINT32_MAX);
}

// ============================================
// Process Isolation Tests
// ============================================

TEST_CASE(Memory, ProcessCanAccessOwnMemory) {
  reset_memory();
  uint32_t addr = mallocate(1, 1024);
  set_current_process(1);
  write_word(addr, 0x12345678);
  ASSERT_EQ(read_word(addr), 0x12345678);
}

TEST_CASE(Memory, SystemProcessCanAccessAll) {
  reset_memory();
  set_current_process(SYSTEM_PROCESS_ID);
  write_word(1000, 0xAAAAAAAA);
  ASSERT_EQ(read_word(1000), 0xAAAAAAAA);
}

TEST_CASE(Memory, ProcessCanAccessTextSegment) {
  reset_memory();
  set_current_process(0);
  uint32_t text_addr = TEXT_BASE;
  write_word(text_addr, 0x11223344);
  ASSERT_EQ(read_word(text_addr), 0x11223344);
}

TEST_CASE(Memory, ProcessCanAccessDataSegment) {
  reset_memory();
  set_current_process(0);
  uint32_t data_addr = DATA_BASE;
  write_word(data_addr, 0x55667788);
  ASSERT_EQ(read_word(data_addr), 0x55667788);
}

// ============================================
// Edge Cases and Bounds Tests
// ============================================

TEST_CASE(Memory, ReadAtAddressZero) {
  reset_memory();
  write_word(0, 0xCAFEBABE);
  ASSERT_EQ(read_word(0), 0xCAFEBABE);
}

TEST_CASE(Memory, WriteReadSequentialAddresses) {
  reset_memory();
  for (uint32_t i = 0; i < 100; i++) {
    write_byte(10000 + i, (uint8_t)(i & 0xFF));
  }
  for (uint32_t i = 0; i < 100; i++) {
    ASSERT_EQ(read_byte(10000 + i), (uint8_t)(i & 0xFF));
  }
}

TEST_CASE(Memory, LargeDataTransfer) {
  reset_memory();
  uint32_t base = 20000;
  for (uint32_t i = 0; i < 256; i++) {
    write_word(base + (i * 4), i * 0x11111111);
  }
  for (uint32_t i = 0; i < 256; i++) {
    ASSERT_EQ(read_word(base + (i * 4)), i * 0x11111111);
  }
}

TEST_CASE(Memory, AlternatingReadWrite) {
  reset_memory();
  uint32_t addr = 30000;
  write_word(addr, 0x12345678);
  ASSERT_EQ(read_word(addr), 0x12345678);
  write_word(addr, 0xAABBCCDD);
  ASSERT_EQ(read_word(addr), 0xAABBCCDD);
  write_word(addr, 0xFFEEDDCC);
  ASSERT_EQ(read_word(addr), 0xFFEEDDCC);
}

// ============================================
// Endianness Tests
// ============================================

TEST_CASE(Memory, LittleEndianByteOrder) {
  reset_memory();
  write_word(40000, 0x12345678);
  ASSERT_EQ(read_byte(40000), 0x78);
  ASSERT_EQ(read_byte(40001), 0x56);
  ASSERT_EQ(read_byte(40002), 0x34);
  ASSERT_EQ(read_byte(40003), 0x12);
```

```c
}

TEST_CASE(Memory, HalfwordEndianness) {
  reset_memory();
  write_word(41000, 0xAABBCCDD);
  ASSERT_EQ(read_hword(41000), 0xCCDD);
  ASSERT_EQ(read_hword(41002), 0xAABB);
}

// ==================================================
// Cache Statistics Tests
// ==================================================

TEST_CASE(Memory, CacheStatsInitiallyZero) {
  reset_memory();
  // Just verify it doesn't crash
  print_cache_stats();
  ASSERT_TRUE(1);
}

TEST_CASE(Memory, ReadGeneratesCacheActivity) {
  reset_memory();
  for (int i = 0; i < 10; i++) {
    read_word(50000 + (i * 4));
  }
  // Cache should have some activity
  ASSERT_TRUE(1);
}

// ==================================================
// Memory Coalescing Tests
// ==================================================

TEST_CASE(Memory, CoalesceAdjacentFreeBlocks) {
  reset_memory();
  uint32_t addr1 = mallocate(1, 256);
  uint32_t addr2 = mallocate(2, 256);

  ASSERT_TRUE(addr1 != UINT32_MAX);
  ASSERT_TRUE(addr2 != UINT32_MAX);

  liberate(1);
  liberate(2);

  // Should be able to allocate larger block
  uint32_t addr3 = mallocate(3, 512);
  ASSERT_TRUE(addr3 != UINT32_MAX);
}

// ==================================================
// Stress Tests
// ==================================================

TEST_CASE(Memory, ManyAllocationsAndFrees) {
  reset_memory();
  for (int i = 0; i < 20; i++) {
    uint32_t addr = mallocate(i, 128);
    ASSERT_TRUE(addr != UINT32_MAX);
    if (i % 2 == 0) {
      liberate(i);
    }
  }
}
```

```
TEST_CASE(Memory, InterleavedCacheOperations) {
  reset_memory();
  for (int i = 0; i < 50; i++) {
    uint32_t addr = 60000 + (i * 8);
    write_word(addr, i);
    uint32_t val = read_word(addr);
    ASSERT_EQ(val, (uint32_t)i);
  }
}

static uint32_t make_i_instruction(uint32_t opcode, uint32_t rs, uint32_t rt,
uint32_t immediate) {
  return (opcode & 0x3F) << OPCODE_SHIFT | (rs & 0x1F) << RS_SHIFT |
  (rt & 0x1F) << RT_SHIFT | (immediate & 0xFFFF);
}

static void reset_cpu_state(void) {
  memset(&THE_CPU, 0, sizeof(THE_CPU));
}


static void reset_cpu_and_memory(void) {
  reset_cpu_state();
  free_memory();
  init_memory(CACHE_WRITE_THROUGH);
  set_current_process(SYSTEM_PROCESS_ID);
}

static inline uint32_t mask_reg_index(uint32_t reg) {
  return (uint32_t)reg & 0x1F;
}

static inline int32_t read_gpr(uint32_t reg) {
  uint32_t idx = mask_reg_index(reg);
  if (idx == REG_ZERO || idx >= GP_REG_COUNT) {
    return 0;
  }
  return THE_CPU.gp_registers[idx];
}

static inline void write_gpr(uint32_t reg, uint32_t value) {
  uint32_t idx = mask_reg_index(reg);
  if (idx == REG_ZERO || idx >= GP_REG_COUNT) {
    return;
  }
  THE_CPU.gp_registers[idx] = value;
}

TEST_CASE(ITypeImmediate, AddiAddsSignedImmediate) {
  reset_cpu_and_memory();
  write_gpr(REG_T0, 5);
  execute_instruction(make_i_instruction(OP_ADDI, REG_T0, REG_T1, 0xFFFD));
  ASSERT_EQ(read_gpr(REG_T1), 2);
}

TEST_CASE(ITypeImmediate, AddiuWrapsUnsignedResult) {
  reset_cpu_and_memory();
  write_gpr(REG_T0, (uint32_t)0xFFFFFFFF);
  execute_instruction(make_i_instruction(OP_ADDIU, REG_T0, REG_T1, 0x0001));
  ASSERT_EQ((uint32_t)read_gpr(REG_T1), (uint32_t)0x00000000);
}

TEST_CASE(ITypeImmediate, AndiZeroExtendsImmediate) {
  reset_cpu_and_memory();
  write_gpr(REG_T0, (uint32_t)0xF0F0FFFF);
  execute_instruction(make_i_instruction(OP_ANDI, REG_T0, REG_T1, 0x8001));
  ASSERT_EQ((uint32_t)read_gpr(REG_T1), (uint32_t)0x00008001);
}

TEST_CASE(ITypeImmediate, OriCombinesBits) {
  reset_cpu_and_memory();
  write_gpr(REG_T0, 0x00FF0000);
  execute_instruction(make_i_instruction(OP_ORI, REG_T0, REG_T1, 0x1234));
  ASSERT_EQ((uint32_t)read_gpr(REG_T1), (uint32_t)0x00FF1234);
}

TEST_CASE(ITypeImmediate, XoriFlipsBits) {
  reset_cpu_and_memory();
  write_gpr(REG_T0, 0xFFFF0000);
  execute_instruction(make_i_instruction(OP_XORI, REG_T0, REG_T1, 0x0F0F));
  ASSERT_EQ((uint32_t)read_gpr(REG_T1), (uint32_t)0xFFFF0F0F);
}

TEST_CASE(ITypeImmediate, SltiPerformsSignedComparison) {
  reset_cpu_and_memory();
  write_gpr(REG_T0, -2);
  execute_instruction(make_i_instruction(OP_SLTI, REG_T0, REG_T1, 0x0001));
  ASSERT_EQ(read_gpr(REG_T1), 1);
}

TEST_CASE(ITypeImmediate, SltiuPerformsUnsignedComparison) {
  reset_cpu_and_memory();
  write_gpr(REG_T0, (uint32_t)0x80000000);
  execute_instruction(make_i_instruction(OP_SLTIU, REG_T0, REG_T1, 0xFFFF));
  ASSERT_EQ(read_gpr(REG_T1), 1);
}

TEST_CASE(ITypeImmediate, LuiLoadsUpperImmediate) {
  reset_cpu_and_memory();
  execute_instruction(make_i_instruction(OP_LUI, REG_ZERO, REG_T1, 0x1234));
  ASSERT_EQ((uint32_t)read_gpr(REG_T1), (uint32_t)0x12340000);
}

TEST_CASE(ITypeImmediate, LoadWordFetchesStoredValue) {
  reset_cpu_and_memory();
  write_gpr(REG_T0, 100);
  write_gpr(REG_T1, 0x12345678);
  execute_instruction(make_i_instruction(OP_SW, REG_T0, REG_T1, 0));
  write_gpr(REG_T1, 0);
  execute_instruction(make_i_instruction(OP_LW, REG_T0, REG_T1, 0));
  ASSERT_EQ((uint32_t)read_gpr(REG_T1), (uint32_t)0x12345678);
}

TEST_CASE(ITypeImmediate, LoadByteSignExtendsValue) {
  reset_cpu_and_memory();
  write_gpr(REG_T0, 120);
  write_gpr(REG_T1, 0x000000FF);
  execute_instruction(make_i_instruction(OP_SB, REG_T0, REG_T1, 0));
  write_gpr(REG_T2, 0);
  execute_instruction(make_i_instruction(OP_LB, REG_T0, REG_T2, 0));
  ASSERT_EQ((uint32_t)read_gpr(REG_T2), (uint32_t)0xFFFFFFFF);
}

TEST_CASE(ITypeImmediate, LoadByteUnsignedZeroExtends) {
  reset_cpu_and_memory();
  write_gpr(REG_T0, 140);
  write_gpr(REG_T1, 0x00000080);
  execute_instruction(make_i_instruction(OP_SB, REG_T0, REG_T1, 0));
  write_gpr(REG_T2, 0);
  execute_instruction(make_i_instruction(OP_LBU, REG_T0, REG_T2, 0));
  ASSERT_EQ((uint32_t)read_gpr(REG_T2), (uint32_t)0x00000080);
}

TEST_CASE(ITypeImmediate, LoadHalfSignExtendsValue) {
  reset_cpu_and_memory();
  write_gpr(REG_T0, 160);
  write_gpr(REG_T1, 0x0000F234);
  execute_instruction(make_i_instruction(OP_SH, REG_T0, REG_T1, 0));
  write_gpr(REG_T2, 0);
  execute_instruction(make_i_instruction(OP_LH, REG_T0, REG_T2, 0));
  ASSERT_EQ((uint32_t)read_gpr(REG_T2), (uint32_t)0xFFFFF234);
}

TEST_CASE(ITypeImmediate, LoadHalfUnsignedZeroExtends) {
  reset_cpu_and_memory();
  write_gpr(REG_T0, 180);
  write_gpr(REG_T1, 0x0000ABCD);
  execute_instruction(make_i_instruction(OP_SH, REG_T0, REG_T1, 0));
  write_gpr(REG_T2, 0);
  execute_instruction(make_i_instruction(OP_LHU, REG_T0, REG_T2, 0));
  ASSERT_EQ((uint32_t)read_gpr(REG_T2), (uint32_t)0x0000ABCD);
}

TEST_CASE(ITypeImmediate, StoreByteUpdatesSingleByte) {
  reset_cpu_and_memory();
  write_gpr(REG_T0, 200);
  write_gpr(REG_T1, 0x11223344);
  execute_instruction(make_i_instruction(OP_SW, REG_T0, REG_T1, 0));
  write_gpr(REG_T2, 0x000000AA);
  execute_instruction(make_i_instruction(OP_SB, REG_T0, REG_T2, 2));
  write_gpr(REG_T3, 0);
  execute_instruction(make_i_instruction(OP_LW, REG_T0, REG_T3, 0));
  ASSERT_EQ((uint32_t)read_gpr(REG_T3), (uint32_t)0x11AA3344);
}

TEST_CASE(ITypeImmediate, StoreHalfWritesTwoBytes) {
  reset_cpu_and_memory();
  write_gpr(REG_T0, 220);
  write_gpr(REG_T1, 0x00001234);
  execute_instruction(make_i_instruction(OP_SH, REG_T0, REG_T1, 0));
  write_gpr(REG_T2, 0);
  execute_instruction(make_i_instruction(OP_LW, REG_T0, REG_T2, 0));
  ASSERT_EQ((uint32_t)read_gpr(REG_T2), (uint32_t)0x00001234);
}

TEST_CASE(ITypeImmediate, BeqBranchesWhenEqual) {
  reset_cpu_and_memory();
  THE_CPU.hw_registers[PC] = 0x100;
  write_gpr(REG_T0, 7);
  write_gpr(REG_T1, 7);
  execute_instruction(make_i_instruction(OP_BEQ, REG_T0, REG_T1, 0x0002));
  ASSERT_EQ(THE_CPU.hw_registers[PC], (uint32_t)(0x100 + 4 + (2 << 2)));
}

TEST_CASE(ITypeImmediate, BneBranchesWhenNotEqual) {
  reset_cpu_and_memory();
  THE_CPU.hw_registers[PC] = 0x80;
  write_gpr(REG_T0, 1);
  write_gpr(REG_T1, 2);
  execute_instruction(make_i_instruction(OP_BNE, REG_T0, REG_T1, 0x0003));
  ASSERT_EQ(THE_CPU.hw_registers[PC], (uint32_t)(0x80 + 4 + (3 << 2)));
}
```

13 Conclusion

13.1 Summary

The operating system simulator represents a comprehensive exploration of core OS concepts through hands-on implementation and analysis. One of the key accomplishments of this project was the successful simulation of concurrent process execution, allowing multiple processes to be managed simultaneously while accurately modeling CPU behavior. Through advanced process simulation, the system demonstrates realistic handling of process states, execution flow, and synchronization challenges inherent in multitasking environments.

A major focus of the project was memory management and its tight integration with concurrency. By designing memory allocation and access mechanisms that operate correctly under concurrent workloads, the simulator highlights the complexities of shared resources, data consistency, and performance trade-offs. This integration reinforced the importance of coordination between memory systems and process execution in modern operating systems.

The simulator also implements several process scheduling strategies, emphasizing context switching under real-time and non-real-time constraints. Supporting multiple scheduling algorithms required careful handling of timing, preemption, and priority management. Through interrupt handling, the system models asynchronous events such as I/O completion and timer interrupts, further enhancing realism and demonstrating how operating systems maintain responsiveness and fairness in the presence of unpredictable events.

Finally, a comprehensive scheduling algorithm analysis module enables direct comparison of different scheduling techniques under varied workload scenarios. By collecting and evaluating metrics such as waiting time, turnaround time, response time, and cache behavior, the project provides insight into the strengths and weaknesses of each algorithm. Overall, this simulator strengthened understanding of how individual OS components interact as a cohesive system and deepened practical knowledge of performance evaluation, concurrency control, and system-level design.

13.2 Key Learning Insights

Working on this simulator deepened understanding of how operating system components interact under concurrent workloads. In particular, it highlighted the complexity of context switching, the impact of scheduling decisions on performance metrics, and the challenges of coordinating memory management with process execution. The project also reinforced the importance of empirical analysis when evaluating scheduling algorithms, demonstrating how different workloads can significantly influence fairness, responsiveness, and throughput. In addition, we learned how to make an assembler as well as the MIPS I architecture.