Final Project Documentation: Advanced Operating System Simulator

Brysen Pfingsten, Nathaniel Savoury,
David Fields

December 9, 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 Key Concepts and Features

2.1 Project 1

2.2 Project 2

2.3 Project 3

Wasn't Assigned

2.4 Project 4

Wasn't Assigned

3 Module 1: Process Simulation

3.1 Problem Statement

3.2 Implementation

3.2.1 Core CPU Components and Registers

4 Overview

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

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

5 Registers

5.1 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`)

5.2 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`.

6 Instruction Formats

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

6.1 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:

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

6.2 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.

6.3 J-Type Format

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

The effective jump address is formed as:

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

7 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.

7.1 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

7.2 Multiply and Divide (R-Type)

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

7.3 Logical and Bitwise (R-Type)

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

7.4 Shift Instructions (R-Type)

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

7.5 Immediate Arithmetic and Logical (I-Type)

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

7.6 Load and Store (I-Type)

Effective address:

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

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

Mnemonic	Format	Opcode	Description	Semantics
LW	I	0x23	Load word	$\mathrm{GPR}[rt] = \mathrm{Mem32}[\mathrm{EA}]$
SW	I	0x2B	Store word	$\mathrm{Mem32}[\mathrm{EA}] = \mathrm{GPR}[rt]$
LB	I	0x20	Load byte (signed)	$\mathrm{GPR}[rt] = \mathrm{sext}_8(\mathrm{Mem8}[\mathrm{EA}])$
LBU	I	0x24	Load byte (unsigned)	$\mathrm{GPR}[rt] = \mathrm{zext}_8(\mathrm{Mem8}[\mathrm{EA}])$
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	$\mathrm{Mem16}[\mathrm{EA}] = \mathrm{GPR}[rt] \,\&\, 0xFFFF$

7.7 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 GPR[rs] = GPR[rt], then PC = Target
BNE	I	0x05	Branch if not equal	If GPR[rs] $\neq$ GPR[rt], then PC = Target

7.8 Jumps (J-Type and R-Type)

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

7.9 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).

7.10 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 $\leftarrow$ `EPC`.
- Restore user/kernel mode and interrupt enable bits in `Status` as appropriate.

8 Exception and Interrupt Model

8.1 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.

8.2 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.

8.3 Exception Return

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

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

8.3.1 Process Control Block

8.3.2 Fetch-Decode-Execute Cycle

9 Module 2: Advanced Memory Management

9.1 Problem Statement

9.2 Implementation

9.2.1 Hierarchical Memory System

9.2.2 Memory Table

9.2.3 Dynamic Memory Allocation and Deallocation

10 Module 3: Process Scheduling and Context Switching

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

10.1 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.

10.2 Implementation

This section outlines the our solution to the problem statement. Section 10.2.1 presents out we extended the PCB structure from Section 3. Section 10.2.2 describes out we implemented the seven advanced scheduling algorithms. Section 10.2.3 shows how we handle context switching for the preemptive scheduling algorithms. Finally, Section 10.2.4 explains how we integrated the fetch-decode-execute cycle with processes and scheduling logic.

10.2.1 Process Control Block Enhancements

Figure 1 Representation of Processes

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

//To represent a process
typedef struct {
    int pid; //process id
    int pc; //program counter
    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`

10.2.2 Scheduling Algorithms

Round-Robin

For the Round-Robin scheduling algorithm, we chose to have a time quantum of 3.

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;
```

```
//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; //could make ternary
    }

    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);
    }

  }
```

Priority-Based Scheduling

lsf

Shortest Time Remaining

lsf

Highest Response Ratio Next

lsv

First Come First Serve

kfs

Shortest Process Next

skf

Feedback Scheduling

jsnf

10.2.3 Context Switching

snfj

10.2.4 Integration with Fetch-Decode-Execute Cycle

jnsf

11 The Simulation

12 Testing and Debugging

13 Conclusion

14 Appendix A: Screenshots