Final Project Documentation: Advanced Operating System Simulator

Brysen Pfingsten, Nathaniel Savoury,
David Fields

December 10, 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	$\text{GPR}[rd] = \text{GPR}[rs] \wedge \text{GPR}[rt]$
OR	R	`funct` = 0x25	Bitwise OR	$\text{GPR}[rd] = \text{GPR}[rs] \vee \text{GPR}[rt]$
XOR	R	`funct` = 0x26	Bitwise XOR	$\text{GPR}[rd] = \text{GPR}[rs] \oplus \text{GPR}[rt]$
NOR	R	`funct` = 0x27	Bitwise NOR	$\text{GPR}[rd] = \neg(\text{GPR}[rs] \vee \text{GPR}[rt])$

7.4 Shift Instructions (R-Type)

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

7.5 Immediate Arithmetic and Logical (I-Type)

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

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

7.7 Branches (I-Type)

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

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

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

```
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).

10.2.2 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

10.2.3 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);
  }
}
```

10.2.4 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.

11 Module 4: Interrupt Handling and Dispatcher

las

11.1 Problem Statement

11.2 Implementation

11.2.1 Types of Interruption

11.2.2 Interrupt Vector Table

11.2.3 Context Switching

12 Module 5: Efficiency Analysis of Concurrency

12.1 Problem Statement

12.2 Implementation

12.2.1 Performance Metrics Setup

12.2.2 Implementation of Time Tracking

12.2.3 Data Comparison

12.2.4 Performance Comparison

12.2.5 Visualization and Reporting

13 The Simulation

14 Testing and Debugging

15 Conclusion

16 Appendix A: Screenshots