Tiltle: Intelligent CPU Scheduler Simulator

Submitted To: Gagandeep Kaur

Section: K23RZ

Member Details:

1. **Name:** Ghushit Kumar Chutia
   Registration Number: 12306241
   Roll number: 47

2. **Name:** Manasvi Sharma
   Registration number: 12310706
   Roll number: 10

3. **Name:** Umang
   Registration number: 12305833
   Roll number: 9

Contributions:

1. **Ghushit:**
   1. Developed the backend using golang.
   2. Architected the structure of all project modules.
   3. Contributed to JavaScript functinality
   4. Managed Git and GitHub repositories, including maintaining a well-documented README file for a professional presentation.
2. **Manasvi:**
   1. Contributed to frontend components development.
   2. Contributed to JavaScript development.
   3. Contributed to CSS styling.
3. **Umang:**
   1. Contributed to frontend components development.
   2. Contributed to JavaScript development.
   3. Contributed to CSS styling.

1. Project Overview:

The Intelligent CPU Scheduler Simulator is a web-based application designed to visualize and simulate various CPU scheduling algorithms commonly studied in operating systems. This educational tool allows users to input process details such as arrival times, burst times, and priorities, and then observe how different scheduling algorithms would execute these processes. The application generates real-time visualizations including Gantt charts and calculates important performance metrics like average waiting time and turnaround time.

The project is structured with a clear separation between frontend and backend components. The frontend is built using React with Vite for optimal development experience, while the backend uses Go to handle the computational logic of the scheduling algorithms. This architecture enables efficient processing of scheduling algorithms while providing an interactive and responsive user interface.

2. Module-Wise Breakdown:

Backend Module:

- `go.mod`: Defines the Go module and its dependencies
- `go.sum`: Checksums file for Go module dependencies
- `main.go`: Entry point for the backend server, handles API routing and server configuration

Frontend Module:

- `node_modules/`: Contains all the JavaScript dependencies
- `public/`: Static assets accessible to the client
- `src/`: Source code for the React application components
- `.gitignore`: Specifies files to be excluded from version control
- `eslint.config.js`: Configuration for code linting
- `index.html`: Main HTML entry point
- `package-lock.json`: Exact version dependency tree
- `package.json`: Project configuration and dependencies
- `README.md`: Project documentation
- `vite.config.js`: Configuration for the Vite build tool
- `.env`: Environment variables for the application

3. Functionalities:

The Intelligent CPU Scheduler Simulator provides the following key functionalities:

- **Process Input Interface**: Users can add multiple processes with attributes like process ID, arrival time, burst time, and priority.
- **Algorithm Selection**: Users can select between multiple scheduling algorithms:
    - First-Come-First-Served (FCFS)
    - Shortest Job First (SJF) - both preemptive and non-preemptive versions
    - Round Robin with configurable time quantum
    - Priority Scheduling - both preemptive and non-preemptive versions
- **Real-time Visualization**:
    - Interactive Gantt charts showing the execution sequence
    - Timeline view of process states (ready, running, waiting, completed)
- **Performance Metrics Calculation**:
    - Average waiting time
    - Average turnaround time
    - CPU utilization
    - Throughput
- **Comparison View**: Allows users to compare the performance of different algorithms on the same set of processes
- **Process Data Import/Export**: Enables saving and loading process configurations

4. Technology Used:

Programming Languages:

- Golang (Backend): Handles the scheduling algorithm implementations and calculations
- JavaScript (Frontend): Powers the React application and user interface
- HTML/CSS: Structure and styling of the web application

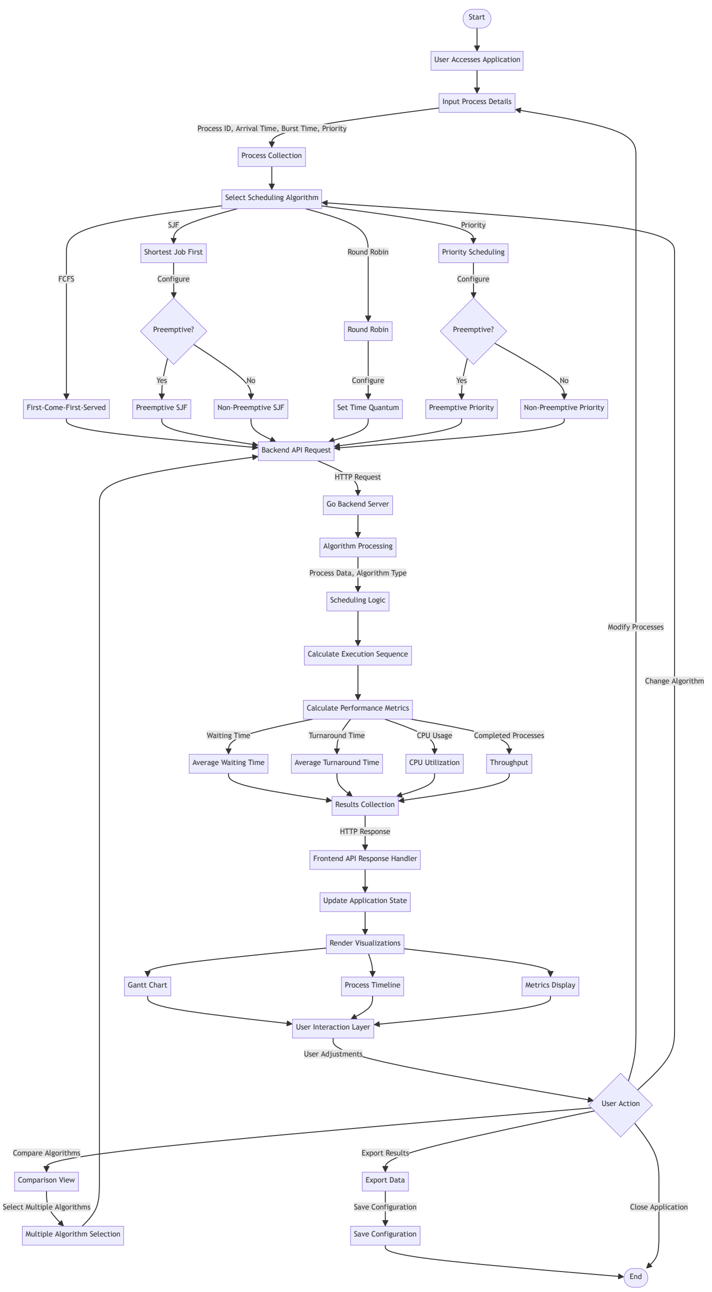
Libraries and Tools:

- React: Frontend UI library for building component-based interfaces
- Vite: Next-generation frontend build tool for faster development
- Tailwind CSS: Utility-first CSS framework for rapid UI development
- Lucide React: Icon library for the user interface
- ESLint: Code quality and style checking

Other Tools:

- GitHub for version control and collaboration
- npm/yarn for package management
- Go modules for dependency management

5. Flow Diagram:



6. Revision Tracking on GitHub:

- **Repository Name**: intelligent-cpu-scheduler-simulator
- **GitHub Link**: https://github.com/GhushitDevX/CPU_Scheduler

The project follows a structured approach to version control with:

- Feature branches for new functionalities
- Regular commits with descriptive messages
- Pull requests for code review before merging to the main branch
- Issue tracking for bug reports and feature requests

7. Conclusion and Future Scope:

The Intelligent CPU Scheduler Simulator successfully provides an interactive educational tool for understanding various CPU scheduling algorithms. The combination of a React frontend and Go backend creates a responsive and computationally efficient application that effectively visualizes complex scheduling concepts.

Future Scope:

- Implementation of additional scheduling algorithms such as Multilevel Queue and Multilevel Feedback Queue
- Enhanced visualization options including process state transitions
- Support for process dependencies and resource allocation
- Machine learning integration to suggest the optimal scheduling algorithm based on workload characteristics
- Support for exporting simulation results in different formats (PDF, CSV)
- Mobile-responsive design for tablet and smartphone access

8. References:

1. Operating System Concepts, 10th Edition by Abraham Silberschatz, Peter B. Galvin, and Greg Gagne
2. React Documentation: https://react.dev/
3. Go Documentation: https://golang.org/doc
4. Vite Documentation: https://vitejs.dev/guide
5. Tailwind CSS Documentation: https://tailwindcss.com/docs

Appendix

A: AI-Generated Project Elaboration/Breakdown Report:

The Intelligent CPU Scheduler Simulator is structured as a full-stack application with clear separation of concerns between the frontend and backend components.

Backend Architecture (Go): The Go backend implements the core logic for CPU scheduling algorithms. Each algorithm is implemented as separate modules that take process information as input and return the execution sequence and performance metrics. The backend exposes RESTful API endpoints that the frontend can call to perform scheduling simulations.

- `main.go`: Sets up the HTTP server, routes, and middleware
- Scheduling algorithm implementations:
    - FCFS (First-Come-First-Served): Processes in order of arrival
    - SJF (Shortest Job First): Prioritizes processes with shortest burst time
    - Round Robin: Time slicing with configurable quantum
    - Priority Scheduling: Processes based on priority values

The backend calculates key metrics including:

- Waiting time for each process
- Turnaround time for each process
- CPU utilization percentage
- Average metrics across all processes

Frontend Architecture (React/Vite): The React frontend provides an intuitive interface for users to interact with the scheduling algorithms. It's built using a component-based architecture for maintainability and reusability.

Key components include:

- Process input form for adding/editing processes
- Algorithm selection and configuration panel
- Visualization components:
    - Gantt chart renderer
    - Timeline visualization
    - Metrics display section
- Comparison view for multiple algorithms

The frontend uses React's state management to handle user interactions and display real-time updates. API calls to the backend are made asynchronously to ensure a responsive user experience.

Data Flow:

1. User inputs process data (ID, arrival time, burst time, priority)
2. User selects scheduling algorithm and configurations
3. Frontend sends process data to backend API
4. Backend executes the selected algorithm and returns results
5. Frontend renders the Gantt chart and displays performance metrics
6. User can adjust inputs and see real-time updates to the visualization

This architecture ensures a clean separation between the user interface and the computational logic, making the application both maintainable and extensible for future enhancements.

B. Problem Statement:

Intelligent CPU Scheduler Simulator

Description: Develop a simulator for CPU scheduling algorithms (FCFS, SJF, Round Robin, Priority Scheduling) with real-time visualizations. The simulator should allow users to input processes with arrival times, burst times, and priorities and visualize Gantt charts and performance metrics like average waiting time and turnaround time.

C. Solution/Code:

<u>Backend:</u>

<u>main.go:</u>

```
package main

import (
    "log"
    "net/http"
    "sort"
    "time"

    "github.com/gin-contrib/cors"
    "github.com/gin-gonic/gin"
```

```go
)

type Process struct {
	ID             string `json:"id"`
	ArrivalTime    int    `json:"arrivalTime"`
	BurstTime      int    `json:"burstTime"`
	RemainingTime  int    `json:"-"`
	Priority       int    `json:"priority,omitempty"`

	StartTime      int  `json:"-"`
	IsStarted      bool `json:"-"`
	CompletionTime int  `json:"completionTime"`
	TurnaroundTime int  `json:"turnaroundTime"`
	WaitingTime    int  `json:"waitingTime"`
	ResponseTime   int  `json:"responseTime"`
}

type TimelineSegment struct {
	ProcessID string `json:"processId"`
	StartTime int    `json:"startTime"`
	EndTime   int    `json:"endTime"`
}

type SimulationRequest struct {
	Algorithm    string    `json:"algorithm"`
	IsPreemptive bool      `json:"isPreemptive"`
	TimeQuantum  int       `json:"timeQuantum,omitempty"`
	Processes    []Process `json:"processes"`
}

type SimulationResponse struct {
	Processes             []Process         `json:"processes"`
	Timeline              []TimelineSegment `json:"timeline"`
	AverageWaitingTime    float64           `json:"averageWaitingTime"`
	AverageTurnaroundTime float64           `json:"averageTurnaroundTime"`
	AverageResponseTime   float64           `json:"averageResponseTime"`
}

func main() {
	r := gin.Default()
```

```go
	// Configure CORS
	r.Use(cors.New(cors.Config{
		AllowOrigins:     []string{"*"},
		AllowMethods:     []string{"POST", "GET", "OPTIONS"},
		AllowHeaders:     []string{"Origin", "Content-Type"},
		ExposeHeaders:    []string{"Content-Length"},
		AllowCredentials: true,
		MaxAge:           12 * time.Hour,
	}))

	r.POST("/simulate", handleSimulation)

	log.Println("Server running on port 8080")
	r.Run(":8080")
}

func handleSimulation(c *gin.Context) {
	var req SimulationRequest
	if err := c.ShouldBindJSON(&req); err != nil {
		c.JSON(http.StatusBadRequest, gin.H{"error": "Invalid request format"})
		return
	}

	// Validate the request
	if len(req.Processes) == 0 {
		c.JSON(http.StatusBadRequest, gin.H{"error": "No processes provided"})
		return
	}

	// Initialize remaining time for all processes
	for i := range req.Processes {
		req.Processes[i].RemainingTime = req.Processes[i].BurstTime
	}

	var response SimulationResponse

	// Run appropriate scheduling algorithm
	switch req.Algorithm {
	case "FCFS":
```

```
        response = runFCFS(req.Processes)
    case "SJF":
        if req.IsPreemptive {
            response = runSRTF(req.Processes) // Preemptive SJF is SRTF
        } else {
            response = runSJF(req.Processes) // Non-preemptive SJF
        }
    case "RR":
        response = runRoundRobin(req.Processes, req.TimeQuantum)
    case "Priority":
        if req.IsPreemptive {
            response = runPreemptivePriority(req.Processes)
        } else {
            response = runNonPreemptivePriority(req.Processes)
        }
    default:
        c.JSON(http.StatusBadRequest, gin.H{"error": "Unknown algorithm"})
        return
    }

    // Calculate average times
    var totalWaitingTime, totalTurnaroundTime, totalResponseTime int
    for _, p := range response.Processes {
        totalWaitingTime += p.WaitingTime
        totalTurnaroundTime += p.TurnaroundTime
        totalResponseTime += p.ResponseTime
    }

    numProcesses := float64(len(response.Processes))
    response.AverageWaitingTime = float64(totalWaitingTime) / numProcesses
    response.AverageTurnaroundTime = float64(totalTurnaroundTime) / numProcesses
    response.AverageResponseTime = float64(totalResponseTime) / numProcesses

    c.JSON(http.StatusOK, response)
}

// First Come First Served (FCFS) scheduling algorithm
func runFCFS(processes []Process) SimulationResponse {
    // Make a copy of processes to avoid modifying the original
    procs := make([]Process, len(processes))
```

```go
copy(procs, processes)

// Sort processes by arrival time
sort.Slice(procs, func(i, j int) bool {
    return procs[i].ArrivalTime < procs[j].ArrivalTime
})

var timeline []TimelineSegment
currentTime := 0

// Process each job in order of arrival
for i := range procs {
    // If the process hasn't arrived yet, advance the clock
    if currentTime < procs[i].ArrivalTime {
        currentTime = procs[i].ArrivalTime
    }

    // Set start time and response time (first time CPU gets the process)
    procs[i].StartTime = currentTime
    procs[i].ResponseTime = procs[i].StartTime - procs[i].ArrivalTime

    // Add to timeline
    segment := TimelineSegment{
        ProcessID: procs[i].ID,
        StartTime: currentTime,
        EndTime:   currentTime + procs[i].BurstTime,
    }
    timeline = append(timeline, segment)

    // Update current time
    currentTime += procs[i].BurstTime

    // Calculate completion time, turnaround time, and waiting time
    procs[i].CompletionTime = currentTime
    procs[i].TurnaroundTime = procs[i].CompletionTime - procs[i].ArrivalTime
    procs[i].WaitingTime = procs[i].TurnaroundTime - procs[i].BurstTime
}

return SimulationResponse{
    Processes: procs,
```

```
        Timeline:  timeline,
    }
}

// Shortest Job First (SJF) - Non-preemptive
func runSJF(processes []Process) SimulationResponse {
    // Make a copy of processes
    procs := make([]Process, len(processes))
    copy(procs, processes)

    var timeline []TimelineSegment
    var completed []Process
    currentTime := 0
    remainingProcesses := len(procs)

    // Find process with earliest arrival time to set initial currentTime
    earliestArrival := procs[0].ArrivalTime
    for _, p := range procs {
        if p.ArrivalTime < earliestArrival {
            earliestArrival = p.ArrivalTime
        }
    }
    currentTime = earliestArrival

    for remainingProcesses > 0 {
        // Find the process with shortest burst time among arrived processes
        minBurstTime := -1
        selectedIdx := -1

        for i, p := range procs {
            if p.RemainingTime > 0 && p.ArrivalTime <= currentTime {
                if minBurstTime == -1 || p.BurstTime < minBurstTime {
                    minBurstTime = p.BurstTime
                    selectedIdx = i
                }
            }
        }

        // If no process is available at current time, advance time to next arrival
        if selectedIdx == -1 {
```

```
			nextArrival := -1
			for _, p := range procs {
				if p.RemainingTime > 0 {
					if nextArrival == -1 || p.ArrivalTime < nextArrival {
						nextArrival = p.ArrivalTime
					}
				}
			}
			currentTime = nextArrival
			continue
		}

		// Set start time for the selected process if it hasn't started yet
		if !procs[selectedIdx].IsStarted {
			procs[selectedIdx].StartTime = currentTime
			procs[selectedIdx].ResponseTime = procs[selectedIdx].StartTime -
procs[selectedIdx].ArrivalTime
			procs[selectedIdx].IsStarted = true
		}

		// Add to timeline
		segment := TimelineSegment{
			ProcessID: procs[selectedIdx].ID,
			StartTime: currentTime,
			EndTime:   currentTime + procs[selectedIdx].RemainingTime,
		}
		timeline = append(timeline, segment)

		// Update current time
		currentTime += procs[selectedIdx].RemainingTime

		// Set completion time, turnaround time, and waiting time
		procs[selectedIdx].CompletionTime = currentTime
		procs[selectedIdx].TurnaroundTime = procs[selectedIdx].CompletionTime -
procs[selectedIdx].ArrivalTime
		procs[selectedIdx].WaitingTime = procs[selectedIdx].TurnaroundTime -
procs[selectedIdx].BurstTime

		// Mark process as completed
		procs[selectedIdx].RemainingTime = 0
```

```
        completed = append(completed, procs[selectedIdx])
        remainingProcesses--
    }

    return SimulationResponse{
        Processes: completed,
        Timeline:  timeline,
    }
}

// Shortest Remaining Time First (SRTF) - Preemptive SJF
func runSRTF(processes []Process) SimulationResponse {
    // Make a copy of processes
    procs := make([]Process, len(processes))
    copy(procs, processes)

    var timeline []TimelineSegment
    currentTime := 0
    remainingProcesses := len(procs)

    // Initialize tracking variables
    for i := range procs {
        procs[i].IsStarted = false
    }

    // Find process with earliest arrival time
    earliestArrival := procs[0].ArrivalTime
    for _, p := range procs {
        if p.ArrivalTime < earliestArrival {
            earliestArrival = p.ArrivalTime
        }
    }
    currentTime = earliestArrival

    // Track the currently running process
    var currentProcess int = -1
    var currentSegmentStart int = 0

    // Continue until all processes complete
    for remainingProcesses > 0 {
```

```
// Find process with shortest remaining time among arrived processes
minRemainingTime := -1
selectedIdx := -1

for i, p := range procs {
   if p.RemainingTime > 0 && p.ArrivalTime <= currentTime {
      if minRemainingTime == -1 || p.RemainingTime < minRemainingTime {
         minRemainingTime = p.RemainingTime
         selectedIdx = i
      }
   }
}

// If no process is available, advance time to next arrival
if selectedIdx == -1 {
   nextArrival := -1
   for _, p := range procs {
      if p.RemainingTime > 0 {
         if nextArrival == -1 || p.ArrivalTime < nextArrival {
            nextArrival = p.ArrivalTime
         }
      }
   }

   // If we had a process running before, add its segment to timeline
   if currentProcess != -1 {
      timeline = append(timeline, TimelineSegment{
         ProcessID: procs[currentProcess].ID,
         StartTime: currentSegmentStart,
         EndTime:   currentTime,
      })
      currentProcess = -1
   }

   currentTime = nextArrival
   continue
}

// If this is the first time this process gets CPU, record response time
if !procs[selectedIdx].IsStarted {
```

```
			procs[selectedIdx].StartTime = currentTime
			procs[selectedIdx].ResponseTime = currentTime - 
procs[selectedIdx].ArrivalTime
			procs[selectedIdx].IsStarted = true
		}

		// If there's a process switch, record the previous process's segment
		if currentProcess != selectedIdx && currentProcess != -1 {
			timeline = append(timeline, TimelineSegment{
				ProcessID: procs[currentProcess].ID,
				StartTime: currentSegmentStart,
				EndTime:   currentTime,
			})
			currentSegmentStart = currentTime
		} else if currentProcess == -1 {
			currentSegmentStart = currentTime
		}

		currentProcess = selectedIdx

		// Determine how long this process will run
		// Either until completion or until next process arrival that could preempt it
		timeSlice := procs[selectedIdx].RemainingTime

		// Find next arrival time that might preempt this process
		for _, p := range procs {
			if p.RemainingTime > 0 && p.ArrivalTime > currentTime && p.ArrivalTime 
< currentTime+timeSlice {
				// Only consider arrivals that could preempt (have shorter remaining time)
				if p.BurstTime < procs[selectedIdx].RemainingTime-(p.ArrivalTime-
currentTime) {
					timeSlice = p.ArrivalTime - currentTime
				}
			}
		}

		// Update current time and process's remaining time
		currentTime += timeSlice
		procs[selectedIdx].RemainingTime -= timeSlice
```

```go
        // If process completes
        if procs[selectedIdx].RemainingTime == 0 {
            // Add final segment to timeline
            timeline = append(timeline, TimelineSegment{
                ProcessID: procs[selectedIdx].ID,
                StartTime: currentSegmentStart,
                EndTime:   currentTime,
            })

            // Set completion time and calculate metrics
            procs[selectedIdx].CompletionTime = currentTime
            procs[selectedIdx].TurnaroundTime = procs[selectedIdx].CompletionTime -
procs[selectedIdx].ArrivalTime
            procs[selectedIdx].WaitingTime = procs[selectedIdx].TurnaroundTime -
procs[selectedIdx].BurstTime

            remainingProcesses--
            currentProcess = -1
        }
    }

    return SimulationResponse{
        Processes: procs,
        Timeline:  timeline,
    }
}

// Round Robin scheduling algorithm
func runRoundRobin(processes []Process, timeQuantum int) SimulationResponse {
    if timeQuantum <= 0 {
        timeQuantum = 1 // Default time quantum
    }

    // Make a copy of processes
    procs := make([]Process, len(processes))
    copy(procs, processes)

    var timeline []TimelineSegment
    var readyQueue []int // Queue of process indices
    currentTime := 0
```

```
// Initialize tracking variables
for i := range procs {
    procs[i].IsStarted = false
}

// Find earliest arrival
earliestArrival := procs[0].ArrivalTime
for _, p := range procs {
    if p.ArrivalTime < earliestArrival {
        earliestArrival = p.ArrivalTime
    }
}
currentTime = earliestArrival

// Add initially available processes to ready queue
for i, p := range procs {
    if p.ArrivalTime <= currentTime {
        readyQueue = append(readyQueue, i)
    }
}

// Continue until all processes complete
completedCount := 0
for completedCount < len(procs) {
    if len(readyQueue) == 0 {
        // Find next arriving process if ready queue is empty
        nextArrival := -1
        nextIndex := -1
        for i, p := range procs {
            if p.RemainingTime > 0 && p.ArrivalTime > currentTime {
                if nextArrival == -1 || p.ArrivalTime < nextArrival {
                    nextArrival = p.ArrivalTime
                    nextIndex = i
                }
            }
        }
        if nextIndex == -1 {
            break // No more processes to execute
        }
```

```
        currentTime = nextArrival
        readyQueue = append(readyQueue, nextIndex)
    }

    // Get next process from ready queue
    currentProcessIdx := readyQueue[0]
    readyQueue = readyQueue[1:] // Dequeue

    // Record response time if process hasn't started
    if !procs[currentProcessIdx].IsStarted {
        procs[currentProcessIdx].StartTime = currentTime
        procs[currentProcessIdx].ResponseTime = currentTime -
procs[currentProcessIdx].ArrivalTime
        procs[currentProcessIdx].IsStarted = true
    }

    // Calculate execution time for this quantum
    executeTime := timeQuantum
    if procs[currentProcessIdx].RemainingTime < executeTime {
        executeTime = procs[currentProcessIdx].RemainingTime
    }

    // Add to timeline
    timeline = append(timeline, TimelineSegment{
        ProcessID: procs[currentProcessIdx].ID,
        StartTime: currentTime,
        EndTime:   currentTime + executeTime,
    })

    // Update time and remaining time
    currentTime += executeTime
    procs[currentProcessIdx].RemainingTime -= executeTime

    // Check for new arrivals during this time quantum
    for i, p := range procs {
        if p.RemainingTime > 0 && p.ArrivalTime > currentTime-executeTime &&
p.ArrivalTime <= currentTime && !contains(readyQueue, i) {
            readyQueue = append(readyQueue, i)
        }
    }
```

```
        // If process still has remaining time, add back to ready queue
        if procs[currentProcessIdx].RemainingTime > 0 {
            readyQueue = append(readyQueue, currentProcessIdx)
        } else {
            // Process completed
            procs[currentProcessIdx].CompletionTime = currentTime
            procs[currentProcessIdx].TurnaroundTime =
procs[currentProcessIdx].CompletionTime - procs[currentProcessIdx].ArrivalTime
            procs[currentProcessIdx].WaitingTime =
procs[currentProcessIdx].TurnaroundTime - procs[currentProcessIdx].BurstTime
            completedCount++
        }
    }

    return SimulationResponse{
        Processes: procs,
        Timeline:  timeline,
    }
}

// Non-Preemptive Priority Scheduling
func runNonPreemptivePriority(processes []Process) SimulationResponse {
    // Make a copy of processes
    procs := make([]Process, len(processes))
    copy(procs, processes)

    var timeline []TimelineSegment
    currentTime := 0
    remainingProcesses := len(procs)

    // Find earliest arrival
    earliestArrival := procs[0].ArrivalTime
    for _, p := range procs {
        if p.ArrivalTime < earliestArrival {
            earliestArrival = p.ArrivalTime
        }
    }
    currentTime = earliestArrival
```

```
    for remainingProcesses > 0 {
        // Find process with highest priority (lowest number) among arrived processes
        highestPriority := -1
        selectedIdx := -1

        for i, p := range procs {
            if p.RemainingTime > 0 && p.ArrivalTime <= currentTime {
                if highestPriority == -1 || p.Priority < highestPriority {
                    highestPriority = p.Priority
                    selectedIdx = i
                }
            }
        }

        // If no process is available, advance time to next arrival
        if selectedIdx == -1 {
            nextArrival := -1
            for _, p := range procs {
                if p.RemainingTime > 0 {
                    if nextArrival == -1 || p.ArrivalTime < nextArrival {
                        nextArrival = p.ArrivalTime
                    }
                }
            }
            currentTime = nextArrival
            continue
        }

        // Record start time and response time if not started
        if !procs[selectedIdx].IsStarted {
            procs[selectedIdx].StartTime = currentTime
            procs[selectedIdx].ResponseTime = currentTime -
procs[selectedIdx].ArrivalTime
            procs[selectedIdx].IsStarted = true
        }

        // Add to timeline
        segment := TimelineSegment{
            ProcessID: procs[selectedIdx].ID,
            StartTime: currentTime,
```

```
            EndTime:   currentTime + procs[selectedIdx].RemainingTime,
        }
        timeline = append(timeline, segment)

        // Update time
        currentTime += procs[selectedIdx].RemainingTime

        // Set completion time and metrics
        procs[selectedIdx].CompletionTime = currentTime
        procs[selectedIdx].TurnaroundTime = procs[selectedIdx].CompletionTime -
procs[selectedIdx].ArrivalTime
        procs[selectedIdx].WaitingTime = procs[selectedIdx].TurnaroundTime -
procs[selectedIdx].BurstTime

        // Mark process as completed
        procs[selectedIdx].RemainingTime = 0
        remainingProcesses--
    }

    return SimulationResponse{
        Processes: procs,
        Timeline:  timeline,
    }
}

// Preemptive Priority Scheduling
func runPreemptivePriority(processes []Process) SimulationResponse {
    // Make a copy of processes
    procs := make([]Process, len(processes))
    copy(procs, processes)

    var timeline []TimelineSegment
    currentTime := 0
    remainingProcesses := len(procs)

    // Initialize tracking variables
    for i := range procs {
        procs[i].IsStarted = false
    }
```

```
// Find earliest arrival
earliestArrival := procs[0].ArrivalTime
for _, p := range procs {
    if p.ArrivalTime < earliestArrival {
        earliestArrival = p.ArrivalTime
    }
}
currentTime = earliestArrival

// Track the currently running process
var currentProcess int = -1
var currentSegmentStart int = 0

for remainingProcesses > 0 {
    // Find process with highest priority (lowest number) among arrived processes
    highestPriority := -1
    selectedIdx := -1

    for i, p := range procs {
        if p.RemainingTime > 0 && p.ArrivalTime <= currentTime {
            if highestPriority == -1 || p.Priority < highestPriority {
                highestPriority = p.Priority
                selectedIdx = i
            }
        }
    }

    // If no process is available, advance time to next arrival
    if selectedIdx == -1 {
        nextArrival := -1
        for _, p := range procs {
            if p.RemainingTime > 0 {
                if nextArrival == -1 || p.ArrivalTime < nextArrival {
                    nextArrival = p.ArrivalTime
                }
            }
        }

        // If we had a process running before, add its segment to timeline
        if currentProcess != -1 {
```

```
            timeline = append(timeline, TimelineSegment{
                ProcessID: procs[currentProcess].ID,
                StartTime: currentSegmentStart,
                EndTime:   currentTime,
            })
            currentProcess = -1
        }

        currentTime = nextArrival
        continue
    }

    // If this is the first time this process gets CPU, record response time
    if !procs[selectedIdx].IsStarted {
        procs[selectedIdx].StartTime = currentTime
        procs[selectedIdx].ResponseTime = currentTime -
procs[selectedIdx].ArrivalTime
        procs[selectedIdx].IsStarted = true
    }

    // If there's a process switch, record the previous process's segment
    if currentProcess != selectedIdx && currentProcess != -1 {
        timeline = append(timeline, TimelineSegment{
            ProcessID: procs[currentProcess].ID,
            StartTime: currentSegmentStart,
            EndTime:   currentTime,
        })
        currentSegmentStart = currentTime
    } else if currentProcess == -1 {
        currentSegmentStart = currentTime
    }

    currentProcess = selectedIdx

    // Determine how long this process will run
    timeSlice := procs[selectedIdx].RemainingTime

    // Find next arrival time that might preempt this process
    for _, p := range procs {
```

```
            if p.RemainingTime > 0 && p.ArrivalTime > currentTime && p.ArrivalTime
< currentTime+timeSlice {
                // Only consider arrivals that could preempt (have higher priority)
                if p.Priority < procs[selectedIdx].Priority {
                    timeSlice = p.ArrivalTime - currentTime
                }
            }
        }

        // Update current time and process's remaining time
        currentTime += timeSlice
        procs[selectedIdx].RemainingTime -= timeSlice

        // If process completes
        if procs[selectedIdx].RemainingTime == 0 {
            // Add final segment to timeline
            timeline = append(timeline, TimelineSegment{
                ProcessID: procs[selectedIdx].ID,
                StartTime: currentSegmentStart,
                EndTime:   currentTime,
            })

            // Set completion time and calculate metrics
            procs[selectedIdx].CompletionTime = currentTime
            procs[selectedIdx].TurnaroundTime = procs[selectedIdx].CompletionTime -
procs[selectedIdx].ArrivalTime
            procs[selectedIdx].WaitingTime = procs[selectedIdx].TurnaroundTime -
procs[selectedIdx].BurstTime

            remainingProcesses--
            currentProcess = -1
        }
    }

    return SimulationResponse{
        Processes: procs,
        Timeline:  timeline,
    }
}
```

```
// Helper function to check if a slice contains a value
func contains(slice []int, val int) bool {
    for _, item := range slice {
        if item == val {
            return true
        }
    }
    return false
}
```

Frontend:

AlgorithmSelector.jsx:

```
const AlgorithmSelector = ({
  algorithm,
  onAlgorithmChange,
  isPreemptive,
  onPreemptiveChange,
  timeQuantum,
  onTimeQuantumChange
}) => {
  const algorithms = [
    { id: 'FCFS', name: 'First Come First Served' },
    { id: 'SJF', name: 'Shortest Job First' },
    { id: 'RR', name: 'Round Robin' },
    { id: 'Priority', name: 'Priority Scheduling' }
  ];

  return (
    <div className="mb-8">
      <h2 className="text-2xl font-bold mb-4">Select Scheduling Algorithm</h2>

      <div className="grid grid-cols-2 md:grid-cols-4 gap-4 mb-6">
        {algorithms.map(algo => (
          <button
            key={algo.id}
            className={`p-4 rounded-lg text-center transition-all ${
              algorithm === algo.id
                ? 'bg-blue-600 shadow-lg'
```

```
          : 'bg-gray-700 hover:bg-gray-600'
      }`}
      onClick={() => onAlgorithmChange(algo.id)}
    >
      <div className="font-medium">{algo.name}</div>
    </button>
  ))}
</div>

{(algorithm === 'SJF' || algorithm === 'Priority') && (
  <div className="mb-6">
    <h3 className="text-xl font-semibold mb-3">Execution Mode</h3>
    <div className="flex space-x-4">
      <button
        className={`px-4 py-2 rounded-lg ${
          !isPreemptive ? 'bg-blue-600' : 'bg-gray-700'
        }`}
        onClick={() => onPreemptiveChange(false)}
      >
        Non-Preemptive
      </button>
      <button
        className={`px-4 py-2 rounded-lg ${
          isPreemptive ? 'bg-blue-600' : 'bg-gray-700'
        }`}
        onClick={() => onPreemptiveChange(true)}
      >
        Preemptive
      </button>
    </div>
  </div>
)}

{algorithm === 'RR' && (
  <div className="mb-6">
    <h3 className="text-xl font-semibold mb-3">Time Quantum</h3>
    <div className="flex items-center">
      <input
        type="number"
        min="1"
```

```
          value={timeQuantum}
          onChange={(e) => onTimeQuantumChange(e.target.value)}
          className="bg-gray-700 text-white px-4 py-2 rounded-lg w-24
focus:outline-none focus:ring-2 focus:ring-blue-500"
         />
         <span className="ml-2">time units</span>
        </div>
       </div>
      )}
    </div>
  );
};

export default AlgorithmSelector;
```

ProcessInputForm.jsx:

```
import { useState } from 'react';
import { PlusCircle, Trash2 } from 'lucide-react';

const ProcessInputForm = ({ algorithm, isPreemptive, processes, onProcessesChange
}) => {
  const addProcess = () => {
    const newId = `P${processes.length + 1}`;
    onProcessesChange([...processes, { id: newId, arrivalTime: 0, burstTime: 1,
priority: 1 }]);
  };

  const removeProcess = (index) => {
    if (processes.length > 1) {
      const newProcesses = [...processes];
      newProcesses.splice(index, 1);
      // Update process IDs to be sequential
      const updatedProcesses = newProcesses.map((p, idx) => ({
        ...p,
        id: `P${idx + 1}`
      }));
      onProcessesChange(updatedProcesses);
    }
  };
```

```
  const updateProcess = (index, field, value) => {
    const newProcesses = [...processes];
    newProcesses[index] = { ...newProcesses[index], [field]: value };
    onProcessesChange(newProcesses);
  };

  return (
    <div>
      <div className="flex justify-between items-center mb-4">
        <h2 className="text-2xl font-bold">Process Details</h2>
        <button
          onClick={addProcess}
          className="flex items-center space-x-1 text-blue-400 hover:text-blue-300"
        >
          <PlusCircle size={20} />
          <span>Add Process</span>
        </button>
      </div>

      <div className="bg-gray-900 rounded-lg p-4 overflow-x-auto">
        <table className="w-full">
          <thead>
            <tr className="text-left border-b border-gray-700">
              <th className="pb-2 px-4">Process ID</th>
              <th className="pb-2 px-4">Arrival Time</th>
              <th className="pb-2 px-4">Burst Time</th>
              {algorithm === 'Priority' && <th className="pb-2 px-4">Priority</th>}
              <th className="pb-2 px-4 w-16">Actions</th>
            </tr>
          </thead>
          <tbody>
            {processes.map((process, index) => (
              <tr key={index} className="border-b border-gray-800 hover:bg-
gray-800">
                <td className="py-3 px-4">
                  <input
                    type="text"
                    value={process.id}
                    onChange={(e) => updateProcess(index, 'id', e.target.value)}
```

```
                className="bg-gray-700 px-3 py-1 rounded w-full focus:outline-none
focus:ring-1 focus:ring-blue-500"
              />
            </td>
            <td className="py-3 px-4">
              <input
                type="number"
                min="0"
                value={process.arrivalTime}
                onChange={(e) => updateProcess(index, 'arrivalTime', e.target.value)}
                className="bg-gray-700 px-3 py-1 rounded w-full focus:outline-none
focus:ring-1 focus:ring-blue-500"
              />
            </td>
            <td className="py-3 px-4">
              <input
                type="number"
                min="1"
                value={process.burstTime}
                onChange={(e) => updateProcess(index, 'burstTime', e.target.value)}
                className="bg-gray-700 px-3 py-1 rounded w-full focus:outline-none
focus:ring-1 focus:ring-blue-500"
              />
            </td>
            {algorithm === 'Priority' && (
              <td className="py-3 px-4">
                <input
                  type="number"
                  min="1"
                  value={process.priority}
                  onChange={(e) => updateProcess(index, 'priority', e.target.value)}
                  className="bg-gray-700 px-3 py-1 rounded w-full focus:outline-none
focus:ring-1 focus:ring-blue-500"
                />
              </td>
            )}
            <td className="py-3 px-4">
              <button
                onClick={() => removeProcess(index)}
                disabled={processes.length <= 1}
```

```
              className="text-red-400 hover:text-red-300 disabled:opacity-30"
            >
              <Trash2 size={18} />
            </button>
          </td>
        </tr>
      ))}
    </tbody>
  </table>
 </div>
</div>
 );
};

export default ProcessInputForm;
```

SimulationResults.jsx:

```
import { useEffect, useRef } from 'react';

const SimulationResults = ({ results }) => {
  const ganttChartRef = useRef(null);

  useEffect(() => {
    if (ganttChartRef.current && results.timeline && results.timeline.length > 0) {
      const canvas = ganttChartRef.current;
      const ctx = canvas.getContext('2d');
      const totalTime = results.timeline[results.timeline.length - 1].endTime;



      const displayWidth = canvas.clientWidth;
      canvas.width = displayWidth;
      canvas.height = 120;

      const timeScale = displayWidth / totalTime;

      ctx.clearRect(0, 0, canvas.width, canvas.height);

      const barHeight = 60;
      const barY = 20;
```

```
const colors = [
  '#4C51BF', '#4299E1', '#38B2AC', '#48BB78',
  '#F6AD55', '#ED8936', '#EF4444', '#9F7AEA'
];

ctx.fillStyle = '#1A202C';
ctx.fillRect(0, barY, canvas.width, barHeight);

results.timeline.forEach((segment, index) => {
  const x = segment.startTime * timeScale;
  const width = (segment.endTime - segment.startTime) * timeScale;

  if (width < 1) return;

  const processIndex = parseInt(segment.processId.replace('P', '')) % colors.length;
  ctx.fillStyle = colors[processIndex];
  ctx.fillRect(x, barY, width, barHeight);

  ctx.fillStyle = 'white';
  ctx.font = '12px sans-serif';
  ctx.textAlign = 'center';
  ctx.textBaseline = 'middle';

  if (width > 30) {
    ctx.fillText(segment.processId, x + width/2, barY + barHeight/2);
  }
});

ctx.strokeStyle = '#4A5568';
ctx.fillStyle = '#A0AEC0';
ctx.font = '10px sans-serif';
ctx.textAlign = 'center';

const timeStep = Math.ceil(totalTime / 10);
for (let t = 0; t <= totalTime; t += timeStep) {
  const x = t * timeScale;
  ctx.beginPath();
  ctx.moveTo(x, barY + barHeight);
  ctx.lineTo(x, barY + barHeight + 5);
  ctx.stroke();
```

```
      ctx.fillText(t, x, barY + barHeight + 15);
    }
  }
}, [results]);

if (!results) return null;

return (
  <div className="bg-gray-800 rounded-xl shadow-2xl p-6 mb-8 animate-fadeIn">
    <h2 className="text-3xl font-bold mb-6 text-center">Simulation Results</h2>

    <div className="mb-8">
      <h3 className="text-xl font-semibold mb-4">Gantt Chart</h3>
      <div className="bg-gray-900 p-4 rounded-lg">
        <canvas ref={ganttChartRef} className="w-full h-32"></canvas>
      </div>
    </div>

    <div className="grid grid-cols-1 md:grid-cols-3 gap-6 mb-8">
      <div className="bg-gray-900 p-6 rounded-lg">
        <div className="text-4xl font-bold text-blue-400
mb-2">{results.averageWaitingTime.toFixed(2)}</div>
        <div className="text-gray-400">Average Waiting Time</div>
      </div>
      <div className="bg-gray-900 p-6 rounded-lg">
        <div className="text-4xl font-bold text-green-400
mb-2">{results.averageTurnaroundTime.toFixed(2)}</div>
        <div className="text-gray-400">Average Turnaround Time</div>
      </div>
      <div className="bg-gray-900 p-6 rounded-lg">
        <div className="text-4xl font-bold text-purple-400
mb-2">{results.averageResponseTime.toFixed(2)}</div>
        <div className="text-gray-400">Average Response Time</div>
      </div>
    </div>

    <div className="mb-6">
      <h3 className="text-xl font-semibold mb-4">Process Details</h3>
      <div className="bg-gray-900 rounded-lg p-4 overflow-x-auto">
        <table className="w-full">
```

```
          <thead>
            <tr className="text-left border-b border-gray-700">
              <th className="p-3">Process ID</th>
              <th className="p-3">Arrival Time</th>
              <th className="p-3">Burst Time</th>
              {results.processes[0].priority !== undefined && <th
className="p-3">Priority</th>}
              <th className="p-3">Completion Time</th>
              <th className="p-3">Turnaround Time</th>
              <th className="p-3">Waiting Time</th>
              <th className="p-3">Response Time</th>
            </tr>
          </thead>
          <tbody>
            {results.processes.map((process, index) => (
              <tr key={index} className="border-b border-gray-800 hover:bg-
gray-800">
                <td className="p-3">{process.id}</td>
                <td className="p-3">{process.arrivalTime}</td>
                <td className="p-3">{process.burstTime}</td>
                {process.priority !== undefined && <td
className="p-3">{process.priority}</td>}
                <td className="p-3">{process.completionTime}</td>
                <td className="p-3">{process.turnaroundTime}</td>
                <td className="p-3">{process.waitingTime}</td>
                <td className="p-3">{process.responseTime}</td>
              </tr>
            ))}
          </tbody>
        </table>
      </div>
    </div>
  </div>
  );
};

export default SimulationResults;
```

App.jsx:

```
import { useState } from 'react';
import AlgorithmSelector from './components/AlgorithmSelector';
import ProcessInputForm from './components/ProcessInputForm';
import SimulationResults from './components/SimulationResults';

function App() {
  const [algorithm, setAlgorithm] = useState('FCFS');
  const [isPreemptive, setIsPreemptive] = useState(false);
  const [timeQuantum, setTimeQuantum] = useState(1);
  const [processes, setProcesses] = useState([
    { id: 'P1', arrivalTime: 0, burstTime: 5, priority: 1 }
  ]);
  const [results, setResults] = useState(null);
  const [isLoading, setIsLoading] = useState(false);
  const [error, setError] = useState(null);

  const handleAlgorithmChange = (algo) => {
    setAlgorithm(algo);
    setIsPreemptive(false);
    setResults(null);
  };

  const handlePreemptiveChange = (value) => {
    setIsPreemptive(value);
  };

  const handleTimeQuantumChange = (value) => {
    setTimeQuantum(value);
  };

  const handleProcessChange = (newProcesses) => {
    setProcesses(newProcesses);
  };

  const runSimulation = async () => {
    setIsLoading(true);
    setError(null);
```

```
    try {
      const payload = {
        algorithm,
        isPreemptive,
        timeQuantum: algorithm === 'RR' ? parseInt(timeQuantum) : undefined,
        processes: processes.map(p => ({
          id: p.id,
          arrivalTime: parseInt(p.arrivalTime),
          burstTime: parseInt(p.burstTime),
          priority: algorithm === 'Priority' ? parseInt(p.priority) : undefined
        }))
      };

      const response = await fetch('http://localhost:8080/simulate', {
        method: 'POST',
        headers: {
          'Content-Type': 'application/json',
        },
        body: JSON.stringify(payload),
      });

      if (!response.ok) {
        throw new Error('Simulation failed');
      }

      const data = await response.json();
      setResults(data);
    } catch (err) {
      setError(err.message);
    } finally {
      setIsLoading(false);
    }
  };

  return (
    <div className="min-h-screen bg-gradient-to-br from-blue-900 to-indigo-900
text-white p-6">
      <div className="max-w-6xl mx-auto">
        <header className="text-center mb-10">
          <h1 className="text-4xl font-bold mb-2">CPU Scheduler Simulator</h1>
```

```
        <p className="text-xl opacity-80">Visualize and analyze different CPU
scheduling algorithms</p>
      </header>

      <div className="bg-gray-800 rounded-xl shadow-2xl p-6 mb-8">
        <AlgorithmSelector
          algorithm={algorithm}
          onAlgorithmChange={handleAlgorithmChange}
          isPreemptive={isPreemptive}
          onPreemptiveChange={handlePreemptiveChange}
          timeQuantum={timeQuantum}
          onTimeQuantumChange={handleTimeQuantumChange}
        />

        <ProcessInputForm
          algorithm={algorithm}
          isPreemptive={isPreemptive}
          processes={processes}
          onProcessesChange={handleProcessChange}
        />

        <div className="mt-6 flex justify-center">
          <button
            onClick={runSimulation}
            disabled={isLoading}
            className="px-8 py-3 bg-gradient-to-r from-blue-500 to-indigo-600
rounded-lg text-lg font-semibold shadow-lg hover:shadow-xl transition-all transform
hover:-translate-y-1 disabled:opacity-50"
          >
            {isLoading ? 'Simulating...' : 'Run Simulation'}
          </button>
        </div>

        {error && (
          <div className="mt-4 p-3 bg-red-500 bg-opacity-30 border border-red-500
rounded-lg text-center">
            {error}
          </div>
        )}
      </div>
```

```
        {results && <SimulationResults results={results} />}
      </div>
    </div>
  );
}

export default App;
```

main.jsx:

```
import { StrictMode } from 'react'
import { createRoot } from 'react-dom/client'
import './index.css'
import App from './App.jsx'

createRoot(document.getElementById('root')).render(
  <StrictMode>
    <App />
  </StrictMode>,
)
```