OS CPU Scheduler: Project Report

Safae Hajjout, Yahya Mansoub

May 2025

Contents

1 Introduction

This project, for the Operating Systems course, serves as a general overview of the differnt scheduling algorithms covered in class, implemented. It presents five CPU scheduling algorithms: First-Come-First-Served (FCFS), Shortest Job First (SJF), Priority Scheduling, Round Robin, and Priority Round Robin.
The primary goal is to understand how different scheduling strategies impact process execution, waiting times, and overall CPU utilization.

2 Project Overview

2.1 Language and Tools

We began by evaluating several programming languages, guided by a few key requirements we had defined beforehand:

- The language needed to be low-level enough to align with the core concepts taught in our Operating Systems course.
- It had to support object-oriented programming to help structure our algorithms cleanly.
- It should allow for straightforward integration of various data structures.
- Lastly, we wanted the flexibility to implement visual representations of scheduling behavior.

After weighing our options, we selected **C++** as the most fitting choice. It offered both low-level control and high-level abstractions, along with a strong type system and built-in support for object-oriented design.

With the language chosen, we set up our development environment with the following tools:

- **g++** compiler with support for the C++17 standard
- **Make** for automating the build process (writing long commands to compile and run quickly gets tedious)
- **Git** for tracking changes and documenting our workflow
- **LucidChart and Figma** for planning the general user interaction flow

2.2 System Architecture

To support multiple scheduling algorithms and facilitate modularity, we designed our system around a base class, `Scheduler`. Each scheduling strategy—`FCFS`, `SJF`, `Priority`, `RR`, and `PriorityRR`—inherits from this class, implementing its own specific logic while sharing common infrastructure.

The graph below illustrates exactly how they intermingle:

The architecture is built around several core components:

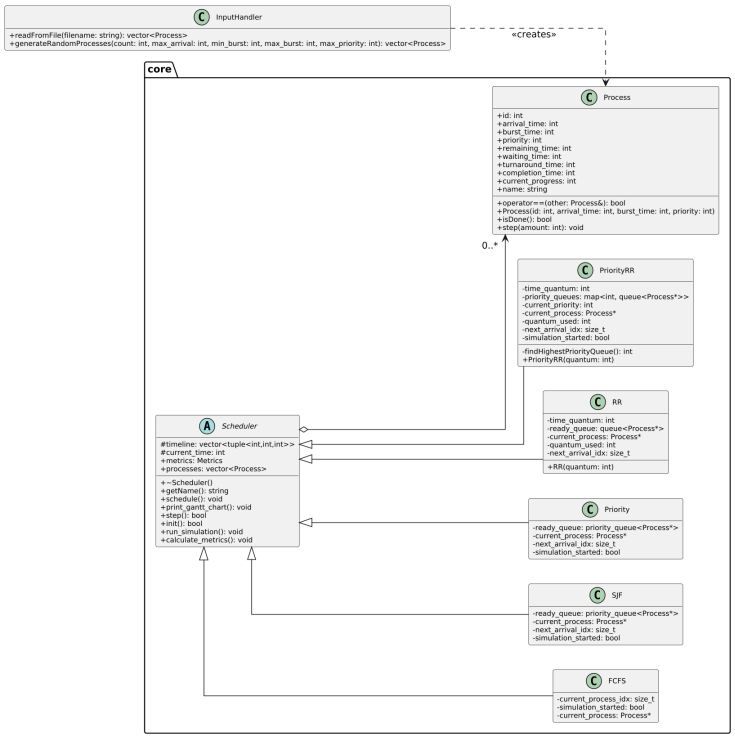

Figure 1: UML Diagram: Scheduler Base Class and Derived Implementations

- **Process Class**: Encapsulates all process attributes and state
- **Scheduler Base Class**: Defines the shared interface and behavior
- **Algorithm Implementations**: Concrete scheduler classes for each algorithm
- **Utils Module**: Contains **InputHandler** for loading process data and **Visualization** tools for terminal-based progress bars and metrics display

This way we ensure a clear separation of logic between the core/ containing all the scheduling algorithms, and utils/ specific to input handling and terminal-based visualization.

2.3 Scheduler Base Class

The abstract `Scheduler` class forms the foundation of our implementation, defining a common interface that all scheduling algorithms must implement.

```
1 class Scheduler {
2 protected:
```

```
    std::vector<std::tuple<int, int, int>> timeline;
    int current_time = 0;

public:
    Metrics metrics;
    std::vector<Process> processes;

    virtual ~Scheduler() = default;
    virtual std::string getName() const = 0;
    virtual void schedule() = 0;
    virtual void print_gantt_chart() = 0;
    virtual bool init() = 0;
    virtual bool step() = 0;

    void run_simulation();
    void calculate_metrics();
};
```

Listing 1: Scheduler Abstract Base Class

Key methods include:

- `schedule()`: Implements the core scheduling algorithm logic
- `init()`: Prepares the scheduler for step-by-step execution
- `step()`: Advances the simulation by one time unit
- `calculate_metrics()`: Computes performance statistics

2.4 Process Model

The `Process` class serves as our main data structure representative of a dummy process, with nearly all the needed attributes for a real scheduling system:

```
class Process {
public:
    int id;               // Unique process identifier
    int arrival_time;     // When the process enters the
    system
    int burst_time;       // Total CPU time required
    int priority;         // Priority value (lower = higher
    priority)
    int remaining_time;   // Runtime tracking
    int waiting_time;     // Performance metric
    int turnaround_time;  // Performance metric
    int completion_time;  // When process finishes
    int current_progress; // For visualization
    std::string name;     // Display name ("P" + id)

    // Methods
    bool isDone() const;
```

```
    void step(int amount = 1);
    // Getters and setters...
};
```

Listing 2: Process Class Implementation

The class includes methods for:

- Tracking execution progress (`step()`, `isDone()`)
- Computing performance metrics (`getTurnaroundTime()`, `getWaitingTime()`)
- Managing process state

2.5 Time Abstraction and Simulation Logic

Rather than relying on system clocks or third-party libraries, we abstracted time as integer units. This internal notion of `Time` helped ensure that simulation behavior remained deterministic, predictable, and portable across environments.

Each scheduler tracks the `current_time` and updates it as processes execute. When visualizing with step-by-step execution, each tick represents one unit of simulated time:

```
// In main.cpp
while (!done) {
    // Show current time
    std::cout << "\033[2;0H";
    std::cout << "Time: " << current_time << "    ";

    // Update progress bars
    for (size_t j = 0; j < scheduler->processes.size(); ++j)
    {
        drawProgressBar(scheduler->processes[j], width, j +
    4);
    }

    // Track finished processes
    for (auto &p : scheduler->processes) {
        if (p.isDone() &&
            std::find(finished_processes.begin(),
                    finished_processes.end(), p) ==
                    finished_processes.end()) {
            finished_processes.push_back(p);
        }
    }

    // Draw metrics table
    drawFinishedTable(finished_processes, 4, 70);

    // Take one simulation step
    done = scheduler->step();
```

```
    current_time++;

    // Add visual delay
    std::this_thread::sleep_for(
        std::chrono::milliseconds(delay_ms));
}
```

Listing 3: Step-by-Step Simulation Logic

2.6 Algorithm Implementations

2.6.1 First-Come-First-Served (FCFS)

FCFS is the simplest scheduling algorithm and the first we implemented. We simply sort the processes by arrival time then execute them in sequence without preemption:

```
void FCFS::schedule() {
    // Sort processes by arrival time
    std::sort(processes.begin(), processes.end(),
        [](const Process& a, const Process& b) {
            return a.getArrivalTime() < b.getArrivalTime();
        });

    current_time = 0;
    for (auto& p : processes) {
        // Handle idle time if needed
        if (current_time < p.getArrivalTime()) {
            current_time = p.getArrivalTime();
        }

        // Update timeline for Gantt chart
        timeline.emplace_back(p.getId(), current_time,
                              current_time + p.getBurstTime()
    );

        // Calculate metrics
        p.setCompletionTime(current_time + p.getBurstTime())
    ;
        p.setTurnaroundTime(p.getCompletionTime() -
                            p.getArrivalTime());
        p.setWaitingTime(p.getTurnaroundTime() -
                         p.getBurstTime());

        current_time += p.getBurstTime();
    }
}
```

Listing 4: FCFS Schedule Implementation

In the step-by-step implementation, the algorithm simply tracks the current process and advances its execution until completion before moving to the next one.

2.6.2 Shortest Job First (SJF)

SJF selects the process with the shortest burst time whenever the CPU becomes available. We used a priority queue, with the comparator wired on the process' remaining time/burst time to efficiently identify the shortest job:

```
struct SJFComparator {
    bool operator()(const Process* a, const Process* b)
    const {
        return a->getRemainingTime() > b->getRemainingTime()
    ;
    }
};

// In schedule() method:
std::priority_queue<Process*, std::vector<Process*>,
                  SJFComparator> pq;

// When selecting next process
Process* p = pq.top();
pq.pop();
```

Listing 5: SJF Comparator and Queue Usage

2.6.3 Priority Scheduling

Similar to SJF, Priority scheduling uses a priority queue but selects based on the process priority attribute (lower number = higher priority):

```
struct PriorityComparator {
    bool operator()(const Process* a, const Process* b)
    const {
        return a->getPriority() > b->getPriority();
    }
};
```

Listing 6: Priority Scheduling Comparator

2.6.4 Round Robin (RR)

We implemented Round Robin using a standard FIFO queue and a time quantum limit. Each process executes for at most the quantum duration before being preempted:

```
// In step() method of RR class
if (current_process) {
```

```
    current_process->step(1);
    current_time++;
    quantum_used++;

    // Check time quantum
    if (current_process->isDone() ||
        quantum_used >= time_quantum) {
        if (current_process->isDone()) {
            // Process complete
            // Calculate metrics...
        } else {
            // Time slice exhausted, put back in queue
            ready_queue.push(current_process);
        }

        // Get next process
        if (!ready_queue.empty()) {
            current_process = ready_queue.front();
            ready_queue.pop();
            quantum_used = 0;
        } else {
            current_process = nullptr;
        }
    }
}
```

Listing 7: Round Robin Time Slice Logic

2.6.5 Priority Round Robin

This algorithm was the most complex, as it combined both priority-based scheduling and preemptiveness. We implemented it using a map of queues, where each priority level maps to its own queue of processes. This is built on top of an existing array of processes, each with an arrival time and other metadata.

Within each individual queue, Round Robin scheduling is applied using a fixed time quantum. However, if a higher-priority process arrives during execution, it preempts the current process, which is then returned to its corresponding queue.

```
// Priority level -> queue of processes
std::map<int, std::queue<Process*>> priority_queues;

// Finding highest priority non-empty queue
int findHighestPriorityQueue() const {
    int highest_priority = std::numeric_limits<int>::max();
    for (const auto& pair : priority_queues) {
        if (!pair.second.empty() &&
            pair.first < highest_priority) {
            highest_priority = pair.first;
        }
```

```
    }

    return (highest_priority ==
            std::numeric_limits<int>::max()) ? -1
                                             :
    highest_priority;
}
```

Listing 8: Priority Round Robin Data Structure

2.7 Standard Library Integration

We took advantage of C++'s Standard Template Library (STL), utilizing containers such as `std::vector`, `std::map`, `std::queue` and `std::priority_queue` to manage scheduling queues. Operator overloading for custom process comparisons allowed these containers to integrate seamlessly with our scheduling logic.

For example, our priority queues automatically maintain the correct ordering of processes:

```
// In SJF class
std::priority_queue<Process*,
                    std::vector<Process*>,
                    SJFComparator> ready_queue;

// In RR class
std::queue<Process*> ready_queue;

// In PriorityRR class
std::map<int, std::queue<Process*>> priority_queues;
```

Listing 9: STL Container Usage

We also used `std::tuple` for timeline entries, `std::chrono` for visualization delays, and modern C++ features like lambdas for sorting and filtering.

2.8 Visualization System

2.8.1 GUI Prototyping and Shift to Terminal UI

At the outset, we explored developing a graphical user interface using the `SFML` library. Our goal was to create an interactive experience where users could input process data, select scheduling algorithms, and observe execution visually via a timeline.

We developed key UI components, including:

- `DropDownMenu`: For algorithm selection
- `InputUI`: For process data input
- `ProcessUIManager`: For visualizing process execution

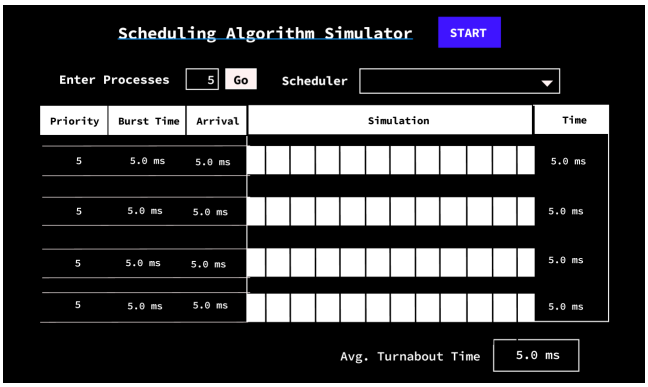

Figure 2: SFML Prototype for Graphical Visualization

- `Start`: For simulation control
- `UIController`: For tying all the components together

However, the complexity of coordinating real-time graphical updates with algorithm logic—and the challenges of debugging layout and animation—led us to pivot. Given the project timeline, we concluded that a fully polished GUI was not feasible. It was however a fruitful learning experience.

2.8.2 Terminal-Based Visualization with Progress Bars

As a practical alternative, we implemented a terminal-based visualization system that preserves interactivity and clarity without the overhead of a full GUI. Using ANSI escape sequences, we created:

- Color-coded process bars
- Real-time progress indicators
- A metrics table for completed processes
- Time and status displays

```
void drawProgressBar(const Process &process, int width, int
    line) {
    float fraction = float(process.current_progress) /
                     float(process.burst_time);
    int filled = int(fraction * width);
    std::string color = COLORS[process.id % 6];

    // Position cursor
    std::cout << "\033[" << line << ";0H";

```

```
    // Draw colored bar
    std::cout << color << "Process " << process.name << "
    [";
    for (int i = 0; i < filled; ++i)
        std::cout << "   ";
    for (int i = filled; i < width; ++i)
        std::cout << " ";
    std::cout << "] " << int(fraction * 100) << "%" << RESET
    ;
    std::cout.flush();
}
```

Listing 10: Terminal Visualization with Progress Bars

2.8.3 Metrics and Statistics

The system calculates and displays performance metrics in real-time:

```
void Scheduler::calculate_metrics() {
    float total_waiting = 0, total_turnaround = 0;
    int total_burst = 0;

    for (const auto& p : processes) {
        total_waiting += p.getWaitingTime();
        total_turnaround += p.getTurnaroundTime();
        total_burst += p.getBurstTime();
    }

    metrics.avg_waiting_time = total_waiting / processes.
    size();
    metrics.avg_turnaround_time = total_turnaround /
                                  processes.size();
    metrics.cpu_utilization = (total_burst /
                               static_cast<float>(current_time
    ))
                               * 100.0f;
}
```

Listing 11: Metrics Calculation

Our visualization displays these metrics both during execution and as a final summary.

2.9 Input Methods

The system supports three input methods:

- **File Input**: Reading process data from text files
- **Random Generation**: Creating random processes with configurable parameters

- **Manual Entry**: Direct input of process parameters

```
std::vector<Process> InputHandler::generateRandomProcesses(
    int count,
    int max_arrival,
    int min_burst,
    int max_burst,
    int max_priority)
{
    // Validate parameters
    if (count <= 0)
        throw std::invalid_argument("Invalid process count");
    if (max_arrival < 0)
        throw std::invalid_argument("Invalid max arrival time");
    if (min_burst <= 0 || max_burst < min_burst)
        throw std::invalid_argument("Invalid burst time range");
    if (max_priority < 0)
        throw std::invalid_argument("Invalid max priority");

    std::vector<Process> processes;
    std::random_device rd;
    std::mt19937 gen(rd());

    // Create distributions
    std::uniform_int_distribution<> arrival_dist(0, max_arrival);
    std::uniform_int_distribution<> burst_dist(min_burst, max_burst);
    std::uniform_int_distribution<> priority_dist(0, max_priority);

    // Generate processes
    for (int i = 0; i < count; ++i) {
        int arrival = arrival_dist(gen);
        int burst = burst_dist(gen);
        int priority = priority_dist(gen);

        processes.emplace_back(i, arrival, burst, priority);
    }

    // Sort by arrival time
    std::sort(processes.begin(), processes.end(),
              [](const Process &a, const Process &b) {
                  return a.getArrivalTime() < b.getArrivalTime();
              });

```

```
42      return processes;
43  }
```

Listing 12: InputHandler Random Process Generation

The file input method allows users to load process data from a text file. The expected format for each line in the file is as follows:

Process ID Arrival Time Burst Time Priority

where:

- **Process ID**: A unique identifier for each process.
- **Arrival Time**: The time at which the process arrives in the system.
- **Burst Time**: The time the process needs to execute on the CPU.
- **Priority**: The priority level of the process (higher values indicate higher priority).

Each line in the file should represent a single process, and the values should be separated by spaces. For example:

1 5 10 3

indicates a process with ID 1, an arrival time of 5, a burst time of 10, and a priority of 3.

The system validates the input values for each line to ensure:

- The arrival time is non-negative.
- The burst time is positive.
- The priority is non-negative.

If any of these conditions are violated, an error is raised with the line number where the issue occurred. If the file is successfully read and parsed, the processes are stored in a vector and returned for further processing.

In case the file cannot be opened or no valid processes are found, appropriate error messages are generated to inform the user.

2.10 Terminal Visualization

Initially, the `main` function was overloaded with several responsibilities, handling various terminal visualization tasks such as drawing progress bars, displaying tables, showing final metrics, and running comparative analysis. Specifically, functions like:

```
void drawProgressBar(const Process &process, int width, int
    line);
void drawFinishedTable(const std::vector<Process> &finished,
     int startLine, int startCol);
void drawFinalMetrics(const Scheduler &scheduler, int
    startLine);
void runComparativeAnalysis();
int selectOperationMode();
void displayComparativeResults(const std::vector<std::pair<
    std::string, Metrics>> &results, int processCount);
void configureComparativeAnalysis(int &numProcesses, int &
    maxArrival, int &minBurst,
                                  int &maxBurst, int &
    maxPriority, int &quantum);
```

This approach violated the **Single Responsibility Principle** (SRP) from object-oriented design principles, which suggests that a class or function should have one reason to change, i.e., each component should be responsible for a specific aspect of the system. Moreover, overloading the `main` function made the system harder to maintain and extend, which could lead to potential bugs and inefficiencies.

To improve this, we refactored the code by introducing three dedicated classes: `Drawer`, `Selector`, and `Comparator`. Each of these classes encapsulates specific functionality related to terminal visualization and analysis, thereby improving modularity and maintainability.

- **Drawer**: Responsible for drawing the progress bar, finished table, and final metrics, as well as displaying comparative results.
- **Selector**: Manages the selection of operation mode, scheduling algorithms, and input methods.
- **Comparator**: Handles the configuration and execution of comparative analysis for different scheduling algorithms.

Class Design and UML Diagram: The new design is illustrated in the updated UML diagram, which is shown below. This diagram represents the relationships between the `Drawer`, `Selector`, and `Comparator` classes, with each class containing specific static methods.

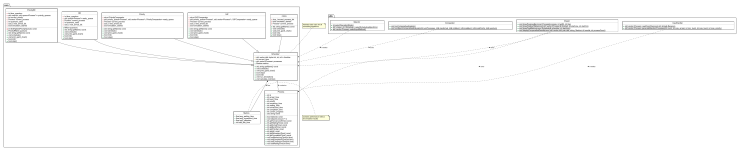

Figure 3: Updated UML Diagram of the System

Class Definitions:

```
#pragma once

class Comparator
{
public:
    static void runComparativeAnalysis();
    static void configureComparativeAnalysis(int &numProcesses, int &maxArrival, int &minBurst,
                                             int &maxBurst, int &maxPriority, int &quantum);
};
```

```

#pragma once

#include <vector>
#include "core/Process.h"
#include "core/Scheduler.h"

class Drawer
{
public:
    static void drawProgressBar(const Process &process, int width, int line);
    static void drawFinishedTable(const std::vector<Process> &finished, int startLine, int startCol);
    static void drawFinalMetrics(const Scheduler &scheduler, int startLine);
    static void displayComparativeResults(const std::vector<std::pair<std::string, Metrics>> &results, int processCount);
};
```

```
#pragma once

#include "core/Scheduler.h"
#include <memory>
#include <vector>

class Selector
{
public:
    static int selectOperationMode();
    static std::unique_ptr<Scheduler> selectSchedulingAlgorithm();
    static std::vector<Process> selectInputMethod();
};
```

By encapsulating these functions into specific classes, we adhere to the **Single Responsibility Principle** and make the system more modular. This

makes it easier to maintain, extend, and test each part of the program separately.

The UML diagram, representing these classes and their relationships, has been compressed and included as an image for clarity. It is present in the project for consulting as the one present here is not possible to be intelligible.

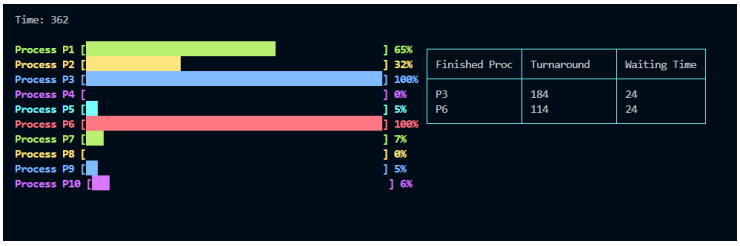

Figure 4: Terminal Visualization During The Scheduling

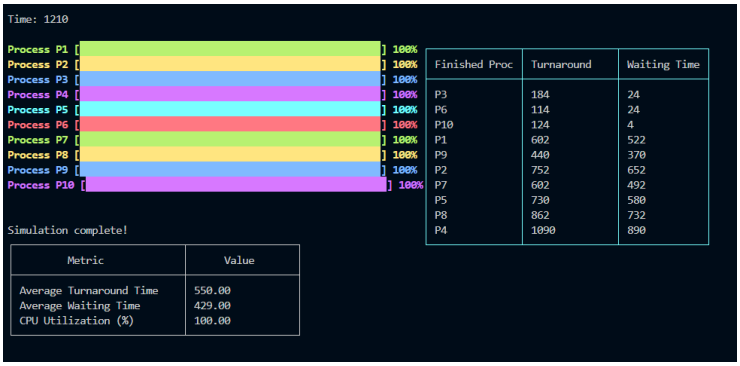

Figure 5: Terminal Visualization After The Scheduling

Remark: It is recommended to use processes with burst and arrival times in seconds (with 100 units corresponding to 1 second in the program's input). This ensures that the visualizations and animations are clear and not overly fast or long.

3 Results and Analysis

3.1 Algorithm Performance Comparisons

Our simulator enables direct comparison of scheduling algorithms using consistent process sets. Some key observations:

- **FCFS**: Simple but can cause convoy effect when a long process blocks shorter ones

- **SJF**: Minimizes average waiting time but can starve longer processes
- **Priority**: Ensures important processes run first but can also lead to starvation
- **Round Robin**: Provides fair CPU time distribution but increases context switches
- **Priority RR**: Balances priority with fairness

3.2 Limitations

Our implementation has several limitations:

- No simulation of I/O bursts or blocking
- No dynamic priority adjustment
- Process arrivals must be predetermined

4 Conclusion and Future Work

This project successfully implemented and analyzed five fundamental CPU scheduling algorithms—FCFS, SJF, Priority, Round Robin, and Priority Round Robin—through an interactive terminal-based simulator. By combining low-level C++ implementation with modern object-oriented design principles, we demonstrated how different scheduling strategies balance competing objectives like fairness, efficiency, and priority handling.

Key achievements include:

- A modular architecture enabling clean algorithm implementations
- Terminal visualization system showing real-time execution progress
- Comparative analysis framework for evaluating scheduling performance
- Practical demonstration of scheduling tradeoffs through metrics

While limited to CPU-bound processes with static priorities, the system provides a robust foundation for understanding core OS scheduling concepts. The observed behaviors align with theoretical expectations—FCFS simplicity, SJF optimality for waiting time, RR fairness, and priority-driven tradeoffs—validating our implementations.

Future enhancements could include:

- Completing the SFML-based GUI
- Adding more advanced scheduling algorithms (Multilevel Feedback Queue, etc.)

- Simulating I/O operations and preemption due to I/O
- Implementing dynamic priorities and aging
- Adding system resource constraints
- Developing aging mechanisms to prevent starvation
- Incorporating interrupt handling and memory management simulations

The modular architecture of our system, with its clear separation between core scheduling logic and support components, makes these extensions relatively straightforward to implement. This project framework serves as both an effective educational tool and a foundation for more complex OS scheduling prototypes.