Simulation Study of Multiprocessor Scheduling Algorithms for Real-Time Systems

Simulation Study in Rust

Chongliu Jia and Chao Shi

Simulation Study of Multiprocessor Scheduling Algorithms for Real-Time Systems

Introduction

Problem Statement

Solutions

Implementation

Testing/Evaluation Results

Conclusions

Learning Achieved through the project

Chongliu Jia and Chao Shi

Simulation Study of Multiprocessor Scheduling Algorithms for Real-Time Systems

Introduction

Managing multi-core processors efficiently is crucial in concurrent computing.

Multi-core technology demands advanced scheduling algorithms for optimized resource use, reduced latency, and increased throughput.

This study simulates advanced algorithms, addressing multi-core challenges.

As processors evolve, the need for complex and adaptive strategies grows, moving beyond traditional scheduling to meet multi-core's parallelism and communication needs.

Chongliu Jia and Chao Shi

Simulation Study of Multiprocessor Scheduling Algorithms for Real-Time Systems

Problem Statement

Current multi-core scheduler simulations face critical challenges:

1. **The SimSo simulation library, constrained by slow updates, only supports Python 2.7—a version no longer compatible with most systems.**

2. **using C/C++ for scheduler development introduces complex memory management issues that can jeopardize system stability and performance. This project aims to address these limitations by developing a more robust and memory-safe simulation environment for multi-core scheduling algorithms.**

Chongliu Jia and Chao Shi

Simulation Study of Multiprocessor Scheduling Algorithms for Real-Time Systems

Solutions

To address the limitations posed by outdated simulation tools and the memory management complexities of C/C++, the proposed solution leverages Rust—a modern, systems-level language that provides performance comparable to C/C++ while ensuring memory safety without a garbage collector.

Rust's rich ecosystem and advanced features such as ownership, zero-cost abstractions, and type safety, facilitate the development of efficient and reliable multi-core scheduler simulations.

Rust's adoption in the Linux kernel underscores its capabilities for system-level programming, positioning it as an ideal choice for implementing a robust framework for multi-core scheduling algorithms.

Simulation Study of Multiprocessor Scheduling Algorithms for Real-Time Systems

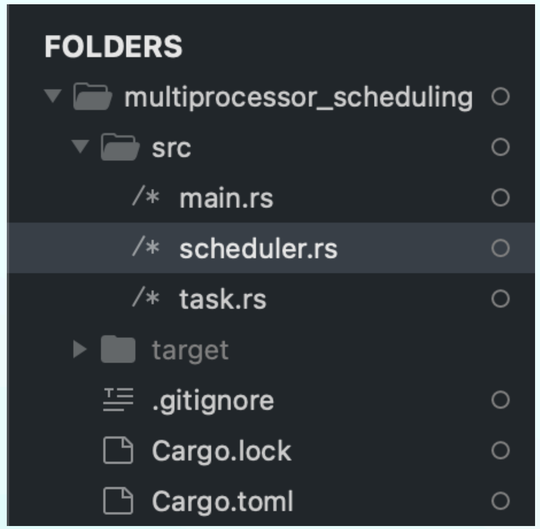
Solutions

Leveraging Rust's rich standard library and ecosystem, our framework provides a strong foundation for system-level programming.

by adopting the Factory pattern, we've modularized our project design, enabling easy integration and simulation of new scheduling algorithms as they emerge, ensuring our framework remains extensible and future-proof.

Simulation Study of Multiprocessor Scheduling Algorithms for Real-Time Systems

Implementation



In the implementation of our Rust-based scheduling framework, the project is modularized into three main files: task.rs, main.rs, and scheduler.rs.

task.rs

It defines the structure and behavior of the tasks that will be scheduled. This modular approach allows for clear definitions and easy manipulation of task properties, essential for any scheduling operation.

main.rs

It serves as the entry point of the program, responsible for initializing the system, orchestrating the simulation process, and interfacing with the user. This centralization simplifies the execution flow and system configuration.

scheduler.rs

It contains the logic for the scheduling algorithms. It is designed to be extensible, allowing for additional algorithms to be added without modifying the core system. This separation of concerns facilitates maintenance and scalability.

The design choice to separate these concerns into distinct files adheres to Rust's philosophy of safety and modularity, ensuring each component is clearly defined, easily testable, and can evolve independently.

Simulation Study of Multiprocessor Scheduling Algorithms for Real-Time Systems

Implementation — Global Earliest Deadline

```rust
pub struct GlobalEDFScheduler {
    tasks: Arc<Mutex<Vec<Task>>>,
}

impl GlobalEDFScheduler {
    pub fn new() -> Self {
        GlobalEDFScheduler {
            tasks: Arc::new(Mutex::new(Vec::new())),
        }
    }

    pub fn add_task(&self, task: Task) {
        let mut tasks = self.tasks.lock().unwrap();
        tasks.push(task);
        tasks.sort_by_key(|task| task.deadline);
    }

    pub fn schedule(&self) -> Option<Task> {
        let mut tasks = self.tasks.lock().unwrap();
        if let Some(next_task) = tasks.iter().min_by_key(|task| task.deadline) {
            let index = tasks.iter().position(|task| task.name == next_task.name).unwrap();
            Some(tasks.remove(index))
        } else {
            None
        }
    }
}

impl SchedulingStrategy for GlobalEDFScheduler {
    fn schedule(&mut self, tasks: &mut Vec<Task>) -> Option<Task> {
        tasks.sort_by_key(|task| task.deadline);
        tasks.pop()
    }

    fn add_task(&mut self, task: Task) {
        self.tasks.lock().unwrap().push(task);
    }
}
```

The GlobalEDFScheduler struct in Rust encapsulates the scheduling logic for the Earliest Deadline First algorithm.

It holds a thread-safe list of tasks, ensuring concurrent access management. The new function initializes an empty task list, while add_task adds a new task and sorts the list by deadlines.

The schedule function then selects the task with the nearest deadline for execution. This structure and its methods reflect the concurrency and safety features of Rust, demonstrating how tasks are prioritized and managed efficiently.

Simulation Study of Multiprocessor Scheduling Algorithms for Real-Time Systems

Implementation — CFS

```rust
pub struct CFS {
    tasks: Arc<Mutex<Vec<Task>>>,
}

impl CFS {
    pub fn new() -> Self {
        CFS {
            tasks: Arc::new(Mutex::new(Vec::new())),
        }
    }

    pub fn add_task(&self, task: Task) {
        let mut tasks = self.tasks.lock().unwrap();
        tasks.push(task);
    }

    pub fn internal_schedule(&self) -> Option<Task> {
        let mut tasks = self.tasks.lock().unwrap();

        if tasks.is_empty() {
            return None;
        }

        let next_task_index = tasks.iter().enumerate().min_by_key(|&(_, task)| task.executed_time).map(|(index, _)| index);

        next_task_index.map(|index| tasks.remove(index))
    }
}

impl SchedulingStrategy for CFS {
    fn schedule(&mut self, tasks: &mut Vec<Task>) -> Option<Task> {
        self.internal_schedule()
    }

    fn add_task(&mut self, task: Task) {
        self.tasks.lock().unwrap().push(task);
    }
}
```

The CFS struct represents the Completely Fair Scheduler, designed for equitable task scheduling. On instantiation, it initializes a protected task list with Arc and Mutex.

The add_task method adds tasks, ensuring thread safety. internal_schedule picks the next task based on executed time, exemplifying fair CPU time allocation.

This implementation in Rust demonstrates the language's ability to handle concurrency and encapsulate complex logic within a safe, efficient scheduling strategy.

Simulation Study of Multiprocessor Scheduling Algorithms for Real-Time Systems

Implementation — Global Queue Scheduler

```rust
pub struct GlobalQueueScheduler {
    tasks: Arc<Mutex<Vec<Task>>>,
}

impl GlobalQueueScheduler {
    pub fn new() -> Self {
        GlobalQueueScheduler {
            tasks: Arc::new(Mutex::new(Vec::new())),
        }
    }

    pub fn add_task(&self, task: Task) {
        let mut tasks = self.tasks.lock().unwrap();
        tasks.push(task);
    }

    pub fn schedule(&self) -> Option<Task> {
        let mut tasks = self.tasks.lock().unwrap();
        tasks.pop()
    }
}

impl SchedulingStrategy for GlobalQueueScheduler {
    fn schedule(&mut self, tasks: &mut Vec<Task>) -> Option<Task> {
        let mut tasks = self.tasks.lock().unwrap();
        tasks.pop()
    }

    fn add_task(&mut self, task: Task) {
        let mut tasks = self.tasks.lock().unwrap();
        tasks.push(task);
    }
}
```

The GlobalQueueScheduler struct in Rust is designed for a straightforward first-in-first-out scheduling approach. It features a thread-safe global queue, enabling tasks to be added and scheduled across multiple threads without race conditions.

The schedule method retrieves and removes the first task in the queue, reflecting the FIFO principle.

This implementation highlights Rust's powerful concurrency tools, providing a simple yet effective solution for managing tasks in a multi-core environment.

Chongliu Jia and Chao Shi

Simulation Study of Multiprocessor Scheduling Algorithms for Real-Time Systems

Implementation — Dynamic Priority Partitioned Scheduler

```rust
pub struct DynamicPriorityPartitionedScheduler {
    queues: Vec<Arc<Mutex<Vec<Task>>>>,
}

impl DynamicPriorityPartitionedScheduler {
    pub fn new(core_count: usize) -> Self {
        let queues = (0..core_count)
            .map(|_| Arc::new(Mutex::new(Vec::new())))
            .collect();

        DynamicPriorityPartitionedScheduler { queues }
    }

    pub fn add_task(&mut self, task: Task, core_id: usize) {
        let queue = &self.queues[core_id];
        queue.lock().unwrap().push(task);
    }

    pub fn schedule(&self, core_id: usize) -> Option<Task> {
        let queue = &self.queues[core_id];
        let mut tasks = queue.lock().unwrap();

        tasks.sort_by(|a, b| (b.priority as u32).cmp(&a.dynamic_priority));

        tasks.pop()
    }
}

impl SchedulingStrategy for DynamicPriorityPartitionedScheduler {
    fn schedule(&mut self, tasks: &mut Vec<Task>) -> Option<Task> {
        tasks.sort();
        tasks.pop()
    }

    fn add_task(&mut self, task: Task) {
        let queue = &self.queues[task.core_id];
        let mut queue_tasks = queue.lock().unwrap();
        queue_tasks.push(task);
    }
}
```

The DynamicPriorityPartitionedScheduler is designed to efficiently manage tasks across multiple cores with dynamic priorities. It maintains a vector of queues, where each queue represents a core in the system.

The new(core_count: usize) function initializes the scheduler with the specified number of cores, creating a separate queue for each core. These queues are implemented using Mutex for thread safety.

When a task is added using add_task(task: Task, core_id: usize), it is placed into the queue corresponding to the specified core_id. This ensures that tasks are assigned to the appropriate core for execution.

The schedule(core_id: usize) -> Option<Task> function is responsible for selecting the next task to execute on a given core. It locks the core's queue, sorts the tasks within the queue by dynamic priority, and returns the task with the highest priority. This dynamic priority-based scheduling helps ensure that high-priority tasks are executed first.

Chongliu Jia and Chao Shi

Simulation Study of Multiprocessor Scheduling Algorithms for Real-Time Systems

Implementation — MultiLevel Queue Scheduler

```rust
pub struct MultiLevelQueueScheduler {
    high_priority_queue: VecDeque<Task>,
    low_priority_queue: VecDeque<Task>,
}

impl MultiLevelQueueScheduler {
    pub fn new() -> Self {
        MultiLevelQueueScheduler {
            high_priority_queue: VecDeque::new(),
            low_priority_queue: VecDeque::new(),
        }
    }

    pub fn add_task(&mut self, task: Task) {
        if task.priority > 5 {
            self.high_priority_queue.push_back(task);
        } else {
            self.low_priority_queue.push_back(task);
        }
    }

    pub fn schedule(&mut self) -> Option<Task> {
        self.high_priority_queue.pop_front()
            .or_else(|| self.low_priority_queue.pop_front())
    }
}

impl SchedulingStrategy for MultiLevelQueueScheduler {
    fn schedule(&mut self, tasks: &mut Vec<Task>) -> Option<Task> {
        self.high_priority_queue.pop_front()
            .or_else(|| self.low_priority_queue.pop_front())
    }

    fn add_task(&mut self, task: Task) {
        if task.priority > 5 {
            self.high_priority_queue.push_back(task);
        } else {
            self.low_priority_queue.push_back(task);
        }
    }
}
```

For MultiLevelQueueScheduler Struct:

high_priority_queue and low_priority_queue are two VecDeque data structures used to store high-priority and low-priority tasks, respectively.

The new() method is used to create a new MultiLevelQueueScheduler instance and initialize both queues as empty.

The add_task(&mut self, task: Task) method is responsible for adding tasks to the scheduler. If a task's priority is greater than 5, it is added to the high-priority queue; otherwise, it is added to the low-priority queue.

The schedule(&mut self) -> Option<Task> method selects the next task to execute from the queues. It first attempts to retrieve a task from the high-priority queue. If the high-priority queue is empty, it falls back to the low-priority queue.

Chongliu Jia and Chao Shi

Simulation Study of Multiprocessor Scheduling Algorithms for Real-Time Systems

Implementation — Work Stealing Scheduler

```rust
pub struct WorkStealingScheduler {
    core_queues: Vec<Mutex<VecDeque<Task>>>,
    current_core_id: usize,
}

impl WorkStealingScheduler {
    pub fn new(core_count: usize) -> Self {
        let core_queues = (0..core_count)
            .map(|_| Mutex::new(VecDeque::new()))
            .collect();

        WorkStealingScheduler {
            core_queues,
            current_core_id: 0,

        }
    }

    pub fn add_task(&self, task: Task) {
        let core_id = task.core_id % self.core_queues.len();
        self.core_queues[core_id].lock().unwrap().push_back(task);
    }

    pub fn schedule(&self, core_id: usize) -> Option<Task> {
        let queue = &self.core_queues[core_id];
        let mut queue = queue.lock().unwrap();
        queue.pop_front().or_else(|| {
            for other_queue in self.core_queues.iter() {
                if let Ok(mut other_queue) = other_queue.try_lock() {
                    if let Some(task) = other_queue.pop_front() {
                        return Some(task);
                    }
                }
            }
            None
        })
    }
}

impl SchedulingStrategy for WorkStealingScheduler {
    fn schedule(&mut self, tasks: &mut Vec<Task>) -> Option<Task> {
        let core_id = self.current_core_id;

        let task = self.core_queues[core_id].lock().unwrap().pop_front();

        if task.is_none() {
            for other_queue in self.core_queues.iter().enumerate().filter(|&(id, _)| id != core_id) {
                let (_, queue) = other_queue;
                if let Ok(mut other_queue) = queue.try_lock() {
                    if let Some(stolen_task) = other_queue.pop_front() {
                        return Some(stolen_task);
                    }
                }
            }
        }
```

For WorkStealingScheduler Struct:
core_queues: This is a vector of mutex-protected queues (VecDeque<Task>), where each queue represents a core's task queue. The number of queues is determined by the core_count provided during initialization.
current_core_id: This field keeps track of the current core's ID. It is used to distribute tasks evenly when adding tasks to the scheduler.

new(core_count: usize) Constructor:
The new method initializes a new WorkStealingScheduler instance. It takes the number of cores (core_count) as an argument and creates a mutex-protected queue for each core, storing them in the core_queues vector.
The current_core_id is initialized to 0, indicating the starting core for task distribution.

add_task(&self, task: Task) Method:
This method adds a task to the scheduler. It calculates the core ID to which the task should be assigned based on the task's core_id property. The modulus operation (%) is used to ensure that the task is assigned to one of the available cores.
The task is then pushed to the corresponding core's queue, which is protected by a mutex to ensure thread safety.

schedule(&self, core_id: usize) -> Option<Task> Method:
This method is used to retrieve and schedule a task from a specific core's queue (core_id). It locks the queue for the specified core and attempts to pop the front task. If a task is available in the core's queue, it is returned.
If the core's queue is empty, the method attempts to steal a task from other cores. It iterates through all the core queues, excluding the current core (to prevent stealing from itself). For each core, it tries to lock the queue and pop the front task. If a task is found, it is returned.
If no task can be retrieved from any core, the method returns None.

Chongliu Jia and Chao Shi

Simulation Study of Multiprocessor Scheduling Algorithms for Real-Time Systems

Testing/Evaluation Results

```
    Finished dev [unoptimized + debuginfo] target(s) in 0.68s
     Running `target/debug/multiprocessor_scheduling`
1: Global Earliest Deadline First Scheduler
2: Completely Fair Scheduler
3: Global Queue Scheduler
4: Dynamic Priority Partitioned Scheduler
5: Multi-Level Queue Scheduler
6: Work-Stealing Scheduler
Plz choose (1, 2, ...): 1
  [00:00:00] [########################################]     100/100     Core 0 all tasks done!
  [00:00:00] [########################################]     100/100     Core 1 all tasks done!
  [00:00:00] [########################################]     100/100     Core 2 all tasks done!
  [00:00:00] [########################################]     100/100     Core 3 all tasks done!
  [00:00:00] [########################################]     100/100     Core 4 all tasks done!
  [00:00:00] [########################################]     100/100     Core 5 all tasks done!
  [00:00:00] [########################################]     100/100     Core 6 all tasks done!
  [00:00:00] [########################################]     100/100     Core 7 all tasks done!
```

The Result of Global Earliest Deadline First Scheduler

```
1: Global Earliest Deadline First Scheduler
2: Completely Fair Scheduler
3: Global Queue Scheduler
4: Dynamic Priority Partitioned Scheduler
5: Multi-Level Queue Scheduler
6: Work-Stealing Scheduler
Plz choose (1, 2, ...): 2
Core 0: Task Task 1 completed
  [00:00:01] [####>-----------------------------------]      10/100     Core 0: Executing Task Task 1
  [00:00:01] [########################################]     100/100     Core 0 all tasks done!
Core 1: Task Task 2 completed
  [00:00:01] [########################################]     100/100     Core 0 all tasks done!
Core 2: Task Task 3 completed
  [00:00:01] [########################################]     100/100     Core 0 all tasks done!
Core 3: Task Task 4 completed
  [00:00:01] [########################################]     100/100     Core 0 all tasks done!
Core 4: Task Task 5 completed
  [00:00:01] [########################################]     100/100     Core 0 all tasks done!
Core 5: Task Task 6 completed
  [00:00:01] [########################################]     100/100     Core 0 all tasks done!
Core 6: Task Task 7 completed
  [00:00:01] [########################################]     100/100     Core 0 all tasks done!
Core 7: Task Task 8 completed
  [00:00:01] [########################################]     100/100     Core 0 all tasks done!
  [00:00:02] [########################################]     100/100     Core 1 all tasks done!
  [00:00:03] [########################################]     100/100     Core 2 all tasks done!
  [00:00:04] [########################################]     100/100     Core 3 all tasks done!
  [00:00:05] [########################################]     100/100     Core 4 all tasks done!
  [00:00:06] [########################################]     100/100     Core 5 all tasks done!
  [00:00:07] [########################################]     100/100     Core 6 all tasks done!
  [00:00:08] [########################################]     100/100     Core 7 all tasks done!
```

The Result of Completely Fair Scheduler

Chongliu Jia and Chao Shi

Simulation Study of Multiprocessor Scheduling Algorithms for Real-Time Systems

Testing/Evaluation Results

```
    Finished dev [unoptimized + debuginfo] target(s) in 0.01s
     Running `target/debug/multiprocessor_scheduling`
1: Global Earliest Deadline First Scheduler
2: Completely Fair Scheduler
3: Global Queue Scheduler
4: Dynamic Priority Partitioned Scheduler
5: Multi-Level Queue Scheduler
6: Work-Stealing Scheduler
Plz choose (1, 2, ...): 3
Core 0: Task Task 1 completed
. [00:00:01] [####>-------------------------------]      10/100     Core 0: Executing Task Task 1
  [00:00:01] [####################################]     100/100     Core 0 all tasks done!
Core 1: Task Task 2 completed
  [00:00:01] [####################################]     100/100     Core 0 all tasks done!
Core 2: Task Task 3 completed
  [00:00:01] [####################################]     100/100     Core 0 all tasks done!
Core 3: Task Task 4 completed
  [00:00:01] [####################################]     100/100     Core 0 all tasks done!
Core 4: Task Task 5 completed
  [00:00:01] [####################################]     100/100     Core 0 all tasks done!
Core 5: Task Task 6 completed
  [00:00:01] [####################################]     100/100     Core 0 all tasks done!
Core 6: Task Task 7 completed
  [00:00:01] [####################################]     100/100     Core 0 all tasks done!
Core 7: Task Task 8 completed
  [00:00:01] [####################################]     100/100     Core 0 all tasks done!
  [00:00:02] [####################################]     100/100     Core 1 all tasks done!
  [00:00:03] [####################################]     100/100     Core 2 all tasks done!
  [00:00:04] [####################################]     100/100     Core 3 all tasks done!
  [00:00:05] [####################################]     100/100     Core 4 all tasks done!
  [00:00:06] [####################################]     100/100     Core 5 all tasks done!
  [00:00:07] [####################################]     100/100     Core 6 all tasks done!
  [00:00:08] [####################################]     100/100     Core 7 all tasks done!
(base) jiachongliu@Tomcat ~/Desktop/558 project/multiprocessor_scheduling  master
```

The Result of Global Queue Scheduler

```
    Finished dev [unoptimized + debuginfo] target(s) in 0.01s
     Running `target/debug/multiprocessor_scheduling`
1: Global Earliest Deadline First Scheduler
2: Completely Fair Scheduler
3: Global Queue Scheduler
4: Dynamic Priority Partitioned Scheduler
5: Multi-Level Queue Scheduler
6: Work-Stealing Scheduler
Plz choose (1, 2, ...): 4
  [00:00:00] [####################################]     100/100     Core 0 all tasks done!
  [00:00:00] [####################################]     100/100     Core 1 all tasks done!
  [00:00:00] [####################################]     100/100     Core 2 all tasks done!
  [00:00:00] [####################################]     100/100     Core 3 all tasks done!
  [00:00:00] [####################################]     100/100     Core 4 all tasks done!
  [00:00:00] [####################################]     100/100     Core 5 all tasks done!
  [00:00:00] [####################################]     100/100     Core 6 all tasks done!
  [00:00:00] [####################################]     100/100     Core 7 all tasks done!
(base) jiachongliu@Tomcat ~/Desktop/558 project/multiprocessor_scheduling  master
```

The Result of Dynamic Priority Partitioned Scheduler

Chongliu Jia and Chao Shi

Simulation Study of Multiprocessor Scheduling Algorithms for Real-Time Systems

Testing/Evaluation Results

```
    Finished dev [unoptimized + debuginfo] target(s) in 0.06s
     Running `target/debug/multiprocessor_scheduling`
1: Global Earliest Deadline First Scheduler
2: Completely Fair Scheduler
3: Global Queue Scheduler
4: Dynamic Priority Partitioned Scheduler
5: Multi-Level Queue Scheduler
6: Work-Stealing Scheduler
Plz choose (1, 2, ...): 5
Core 0: Task Task 1 completed
⠁ [00:00:01] [####>-------------------------------]      10/100     Core 0: Executing Task Task 1
  [00:00:01] [####################################]     100/100     Core 0 all tasks done!
Core 1: Task Task 2 completed
Core 2: Task Task 3 completed
  [00:00:01] [####################################]     100/100     Core 0 all tasks done!
  [00:00:01] [####################################]     100/100     Core 0 all tasks done!
Core 3: Task Task 4 completed
  [00:00:01] [####################################]     100/100     Core 0 all tasks done!
Core 4: Task Task 5 completed
  [00:00:01] [####################################]     100/100     Core 0 all tasks done!
Core 5: Task Task 6 completed
  [00:00:01] [####################################]     100/100     Core 0 all tasks done!
Core 6: Task Task 7 completed
  [00:00:01] [####################################]     100/100     Core 0 all tasks done!
Core 7: Task Task 8 completed
  [00:00:01] [####################################]     100/100     Core 0 all tasks done!
  [00:00:02] [####################################]     100/100     Core 1 all tasks done!
  [00:00:03] [####################################]     100/100     Core 2 all tasks done!
  [00:00:04] [####################################]     100/100     Core 3 all tasks done!
  [00:00:05] [####################################]     100/100     Core 4 all tasks done!
  [00:00:06] [####################################]     100/100     Core 5 all tasks done!
  [00:00:07] [####################################]     100/100     Core 6 all tasks done!
  [00:00:08] [####################################]     100/100     Core 7 all tasks done!
```

The Result of Multi-Level Queue Scheduler

```
    Finished dev [unoptimized + debuginfo] target(s) in 0.01s
     Running `target/debug/multiprocessor_scheduling`
1: Global Earliest Deadline First Scheduler
2: Completely Fair Scheduler
3: Global Queue Scheduler
4: Dynamic Priority Partitioned Scheduler
5: Multi-Level Queue Scheduler
6: Work-Stealing Scheduler
Plz choose (1, 2, ...): 6
Core 0: Task Task 1 completed
⠁ [00:00:01] [####>-------------------------------]      10/100     Core 0: Executing Task Task 1
Core 1: Task Task 2 completed
  [00:00:01] [####################################]     100/100     Core 0 all tasks done!
Core 2: Task Task 3 completed
  [00:00:01] [####################################]     100/100     Core 0 all tasks done!
  [00:00:01] [####################################]     100/100     Core 0 all tasks done!
Core 3: Task Task 4 completed
Core 4: Task Task 5 completed
  [00:00:01] [####################################]     100/100     Core 0 all tasks done!
  [00:00:01] [####################################]     100/100     Core 0 all tasks done!
Core 5: Task Task 6 completed
  [00:00:01] [####################################]     100/100     Core 0 all tasks done!
Core 6: Task Task 7 completed
  [00:00:01] [####################################]     100/100     Core 0 all tasks done!
Core 7: Task Task 8 completed
  [00:00:01] [####################################]     100/100     Core 0 all tasks done!
  [00:00:02] [####################################]     100/100     Core 1 all tasks done!
  [00:00:03] [####################################]     100/100     Core 2 all tasks done!
  [00:00:04] [####################################]     100/100     Core 3 all tasks done!
  [00:00:05] [####################################]     100/100     Core 4 all tasks done!
  [00:00:06] [####################################]     100/100     Core 5 all tasks done!
  [00:00:07] [####################################]     100/100     Core 6 all tasks done!
  [00:00:08] [####################################]     100/100     Core 7 all tasks done!
```

The Result of Work-Stealing Scheduler

Chongliu Jia and Chao Shi

Simulation Study of Multiprocessor Scheduling Algorithms for Real-Time Systems

Conclusions

The project successfully simulated six multi-core scheduling algorithms in Rust, offering insights into their theoretical performance in an idealized environment. Each algorithm highlighted its core principles, from strict deadline-driven prioritization to dynamic workload balancing. The simulations provided valuable observations but did not fully account for real-world complexities.

Global Earliest Deadline First Scheduler: Suitable for time-critical applications but untested in unpredictable scenarios.

Completely Fair Scheduler: Demonstrated fairness in a clean environment, without accounting for external factors.

Global Queue Scheduler: Efficient for homogeneous task sets, less reflective of real-world diversity.

Dynamic Priority Partitioned Scheduler: Adaptable to shifting priorities but not challenged by real-time changes.

Multi-Level Queue Scheduler: Effective at priority-based task categorization, unexplored in priority inversion scenarios.

Work Stealing Scheduler: Promising for load balancing but simulated without complex system factors.

Chongliu Jia and Chao Shi

Simulation Study of Multiprocessor Scheduling Algorithms for Real-Time Systems

Learning Archived through the project

Through the project, I gained valuable experience and knowledge in both Rust programming and the intricacies of multi-core programming. Using Rust for simulation allowed me to become proficient in its syntax and best practices. Additionally, I acquired essential skills related to multi-core program development. This newfound expertise will undoubtedly prove invaluable in my future research endeavors. Whether I'm working on research projects or developing software systems, the insights and techniques I've learned from this project will provide a strong foundation. Specifically, they will greatly assist me in the creation of software and systems, such as file detection systems, by enabling efficient multi-threaded processing and optimization.