Formal Verification of a Custom Scheduling Algorithm and Specification of a Round-Robin Implementation using Coq

Christian Choa
University of the Philippines Diliman
cjchoa@up.edu.ph

Brylle Logroño
University of the Philippines Diliman
bblogrono@up.edu.ph

Raymart Villos
University of the Philippines Diliman
rpvillos@up.edu.ph

Alfonso B. Labao
University of the Philippines Diliman
ablabao@up.edu.ph

Henry N. Adorna
University of the Philippines Diliman
hnadorna@up.edu.ph

ABSTRACT

The Operating System (OS) scheduler is crucial for resource allocation and task execution. However, ensuring the dependability of scheduling algorithms is challenging due to possible human errors in testing. Formal verification provides a systematic approach to prove algorithm correctness and soundness. By mathematically modeling and verifying properties, formal methods ensure scheduler implementations adhere to specifications and function correctly. In this study, we used Coq to formally verify a custom scheduling algorithm, *mysched*, and formulated related correctness and soundness theorems that were then proven using Coq proof tactics. Moreover, we also provided formal specifications of the Round-Robin scheduling algorithm in Coq.

CCS CONCEPTS

• **Theory of computation** → **Logic and verification**;

KEYWORDS

Formal Verification, Coq, Operating System, Round-Robin Scheduling Algorithm

ACM Reference Format:
Christian Choa, Brylle Logroño, Raymart Villos, Alfonso B. Labao, and Henry N. Adorna. 2024. Formal Verification of a Custom Scheduling Algorithm and Specification of a Round-Robin Implementation using Coq. In *Proceedings of Philippine Computing Science Congress (PCSC2024).* Laguna, Philippines, 6 pages.

1 INTRODUCTION

Operating Systems (OS) are software programs that facilitate the execution of tasks and manage computer resources. As an essential component of most computers, the different modules of the OS should be correctly programmed to ensure the entire system's security, functionality, and efficiency. The scheduler determines which task runs in the CPU and is among the most important parts of the OS [2]. Its purpose is to optimize the utilization and allocation of CPU resources, to improve system responsiveness, and to increase overall system performance.

The Formal verification process involves modeling the scheduler as a mathematical object and proving that it implements the desired functions correctly. Coq is a formal proof assistant that provides a formal language to write mathematical definitions, executable algorithms, and theorems [9]. Formal proof assistants like Coq are used to mechanize the proofs of the soundness and other related properties of scheduling algorithms. Formal verification of an operating system (OS) scheduler is crucial for ensuring its correctness and reliability.

This paper will show proof of concept through a custom scheduling algorithm called *mysched* as well as show Coq specifications for its algorithm, properties, and proofs of the properties. In addition, Coq specifications for the Round-robin algorithm will be shown afterwards.

The proof assistant of choice for our research was Coq because it is built upon a strongly typed core language called the Calculus of Inductive Constructions (CIC). This ensures a high degree of correctness in proofs by enforcing type consistency throughout the development process. Furthermore, the strong type system helps catch many errors at compile time rather than at runtime, leading to more reliable proofs. Another good thing about Coq is that it can automatically extract executable programs from specifications, generating Objective Caml or Haskell source code. This feature facilitates the practical application of formal methods by bridging the gap between formal specifications and executable implementations. Next, Coq also supports expressive types, including inductive structures, dependent types, and subset types (Σ-types). These types enable precise specifications and proofs, allowing users to capture complex mathematical structures and properties. Following, the syntax of Coq is more similar to a programming language as opposed to the syntax of Isabelle, a different formal proof assistant, which uses common mathematical ASCII symbols in proof which makes it slightly inconvenient when typing on an IDE [10]. Finally, Coq implements a functional programming language supporting its types. This allows users to express functions over inductive types, write proofs using case analysis, and interactively evaluate expressions, enhancing the flexibility and usability of Coq as a proof assistant [9].

1.1 Related Work

A case study done by Bedarkar et.al verified the Response-Time Analysis (RTA) of FIFO scheduling with the use of Coq Proof Assistance. The study aims to motivate more researchers to explore the use of proof assistants to minimize human error in mathematical

reasoning. In addition, the study points out that there have been a lot of intuitive results in the past that, at first, seemed sound but was later to be proven incorrect. Verification was done by defining the system model, encoding the scheduling policy and preemption model, proving abstract work conservation, bounding the maximum busy-window length and delay within a busy window, and defining the search space [3]. This process will be similarly seen in the following listed studies.
In another study, Sun and Lei formally verified a task scheduler that selects the highest priority ready task for Embedded Operating Systems (EOS) [8]. Here, the researchers verified the EOS task scheduler at different abstraction levels under a unified framework. They had three parts which are: defining the API specifications, defining the functional specifications, and lastly, linking together the aforementioned modules once verified.
Lastly, the formal verification of the OS Kernel seL4, which uses a round-robin scheduler, was done through Isabelle by Klein et.al [7]. According to the researchers, seL4 is the first-ever general-purpose OS kernel that is fully formally verified for functional correctness which will allow for the construction of more secure and reliable systems on top. Verification process is done through different specification layers such as abstract specification, executable specification, C implementation, and machine model which only focuses on the cache and TLB. The researchers conclude that performance does not have to be sacrificed for formal verification. In addition, it was also concluded that "future application proofs can rely on the formal kernel specification that seL4 has been proven to implement". Isabelle and Coq have similarities and differences with one another. Both Isabelle and Coq have automatic tactics for proving or simplification of complex statements. They both as well have approximately the same size of proof and both have a large set of libraries. Isabelle is different to Coq in its expressiveness of underlying logic. While Isabelle uses classical higher-order logic, Coq uses intuitionistic logic [10]. In addition, the author Yushkovskiy believes that Coq requires a deeper understanding of underlying logic theory; however, both Isabelle and Coq would require unfamiliar users a lot of required additional learning in order to understand the process of proving [10].

1.2 Significance

Computers with operating systems are used everywhere in the world. The computer has integrated itself to be a necessity in all aspects of the world. From day-to-day devices such as smartphones to complex computer systems that manage transportation systems [1], ensuring that these computers work as intended holds significant importance in maintaining the safety of those that rely on them. One of the roles of a computer's operating system is scheduling. There are different kinds of scheduling algorithms, such as the First In, First Out algorithm, Shortest Job to Completion algorithm, and the Round-robin algorithm, each with its own strengths, weaknesses, and appropriateness of usage. [4]
This study aims to formally verify the round-robin algorithm through Coq proof assistance. In doing so, the researchers will be able to determine if the algorithm is sound and will be able to find any errors, limitations, and bugs that exist in the algorithm.
If the round-robin algorithm is successfully formally verified, it is ensured that operating systems that use the round-robin algorithm use a sound scheduling algorithm, therefore ensuring that computers work as intended in that aspect. Additionally, future researchers will be able to use this verification of the round-robin algorithm to check the soundness of other algorithms that is based on the round-robin algorithm. [5].
Further down the line, the formal verification of a round-robin algorithm will provide a step in the right direction to formally verify real-time systems. The round-robin algorithm, being the simplest preemptive scheduling algorithm, has so many different applications, uses, and algorithms based on it. Below is a study on a round-robin based load balancing in Software Defined Networking (SDN):

> Results show that round-robin strategy is better than random strategy because round-robin distributes the load uniformly while random does not. [6]

The verification of the round-robin algorithm will allow for the transition from the random strategy to the round-robin strategy to be simpler. It would remove the need to ensure that the round-robin algorithm is sound since it has already been verified through Coq proof assistance.
Formal verification of a widely used algorithm, such as the round-robin algorithm, ensures that programs and real-world systems that use the algorithm will work and behave as intended which will allow for assurance that computers used all around the world stay safe and work as intended.

2 SCHEDULERS

In developing a proof for the algorithm, an accurate translation to Coq is one of the primary priorities. There are many ways to achieve an implementation that satisfies the definition of the scheduling algorithm.
However, careful considerations should be kept in mind because of how it will affect the complexity of proving the theorems. Representations must also be chosen such that they satisfy the structural requirements of Coq. Coq follows a functional paradigm, hence, mutable states and variables are not used. Recursive functions must also be well-founded. Well-founded recursion generalizes both strong induction and recursion.
In this paper, the behavior of an actual operating system scheduler is simplified to achieve a bounded complexity in the theorems we aim to prove. Nevertheless, these simplified models do not have behaviors that we do not expect from a process task being scheduled by an operating system. The tasks or processes all have unique identifiers, some amount of burst time, and in the case of the Round-robin Scheduler, a total burst time that is divisible by the time slice.
Further delimitations were applied to the scheduler. All processes to be scheduled arrive at the same time at $t = 0$, the burst times of all the processes are equal, and the algorithm is offline. Offline algorithms are given the whole problem data from the beginning.
Finally, the overhead for context-switching between two processes is disregarded.

2.1 Custom Scheduling

The custom scheduling algorithm, *mysched*, processes a list of task priority numbers. It iterates through the priorities from left to right, executing a task immediately if its priority is higher than the sum of all remaining priorities to its right. Otherwise, it stores the task in a FIFO queue for later execution. After examining all tasks, it executes them in the order they were stored in the FIFO queue.

Algorithm 1 mysched Algorithm

```
1:  procedure MYSCHED(tasks)
2:      $n \leftarrow$ length of $tasks$
3:      $queue \leftarrow$ empty queue
4:      for $i \leftarrow 1$ to $n$ do
5:          if tasks$[i] > \sum_{j=i+1}^{n}$ tasks$[j]$ then
6:              execute task tasks$[i]$
7:          else
8:              push tasks$[i]$ onto $queue$
9:          end if
10:     end for
11:     while $queue$ is not empty do
12:         execute task pop$(queue)$
13:     end while
14: end procedure
```

2.1.1 *Algorithm.*

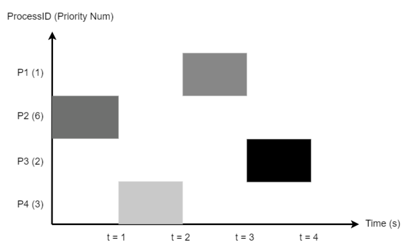

Figure 1: Custom Scheduler Sample Execution

Process ID	Priority Number	Arrival Time	Burst Time
1	1	0	1
2	6	0	1
3	2	0	1
4	3	0	1

Table 1: Example Process Table

The *mysched* algorithm takes in as input a list of task priority numbers seen in Table 1. The processes arrive at the same time. However, the scheduler still has to determine which among the processes to process first. Hence, we break ties by ordering them on increasing process IDs. The scheduler then examines the list from left to right. If it finds that the priority number of a task is strictly greater than the sum of all the other priority numbers to the right, the algorithm will immediately execute the task. Consider process P2 in Table 1. It executed immediately since its priority number $6 > 2 + 3$. It is the first process to execute since, for the other case, the algorithm places the task in a First-in-First-Out(FIFO) queue first. It then goes on to examine the next number in the list. Once the list has been examined, the algorithm executes the FIFO queue.

2.1.2 *Coq Translation.*

```
Fixpoint mysched (l : list nat) : list nat :=
    match l with
    | [ ] => [ ]
    | h::t =>   if ((sum t) <? h) then
    ↪   h::(mysched t)
                else (mysched t) ++ [h]
end.
```

The mysched function operates on a list of natural numbers (nat). When provided an empty list, it returns an empty list as well. However, for a non-empty list l, which has a head element h and a tail list t, if the sum of the elements in t is less than h, h is placed at the beginning of the result obtained by recursively applying mysched to t. On the other hand, if the sum of elements in t is greater than or equal to h, h is appended at the end of the result obtained by recursively applying mysched to t.

2.1.3 *Soundness Property.*

We define a theorem on the custom scheduler soundness.

Theorem 2.1. *For all input list L, the output of the algorithm are permutations of the list L.*

```
Theorem mysched_is_sound :
    forall l, Permutation l (mysched l).
Proof.
    intros. induction l.
    - simpl. apply Permutation_refl.
    - simpl. bdestruct (sum l <? a).
        + change (a::l) with ([a]++l).
            change (a::mysched l) with
            ↪   ([a]++(mysched l)).
            apply Permutation_app_head. apply
            ↪   IHl.
        + change (a::l) with ([a]++l).
            apply perm_trans with (l ++ [a]).
               -- apply Permutation_app_comm.
               -- apply Permutation_app_tail.
                -- apply IHl.
Qed.
```

This theorem states that running mysched with a given input list, its output list will be a rearrangement of the input list. The algorithm does not add or remove any items on the list, for it only changes the order of the input list. This theorem indicates that the algorithm works correctly by ensuring that no items in the list are added or removed since the output list is only a reordering of the input list.

The Coq proof shows that the mysched function is sound by showing that the function outputs the same list but rearranged. It proves this by looking at two cases: when the list *l* is empty and when it is not.

In the case that the list is empty, there is nothing to rearrange; therefore, the rearrangement of the list is the same as the input.

When the list has elements, it checks if the sum of the remaining elements is less than the first element. If it is, it sets the first element as the first item in the output list, otherwise, it is placed at the end of the output list.

The proof proceeds by breaking down the list into its constituents and applying appropriate permutations to maintain the desired ordering. This is achieved through various applications of permutation properties, such as reflexivity, transitivity, and properties of list concatenation, ultimately proving that mysched produces a permutation of the input list.

2.1.4 Correctness Property. To aid with the proof for the correctness of the custom scheduler, we have a helper function that gets the index of a task.

```
Fixpoint getindex (l : list nat) (v : nat) :
↪  nat :=

match l with
| [ ] => 0
| h::t => if (v =? h) then 0 else
            1 + getindex t v
end.
```

The following is the Coq code for the proof of correctness to demonstrate that the intended behavior of the algorithm is assured.

```
Theorem mysched_propone : forall a b l,
    (b =? a = false) ->
    (0 <=? getindex (mysched l) a = true) ->
    (1 <=? getindex (mysched l) b = true) ->
   (0 <=? getindex (mysched l++[b]) a = true)
   ↪  ->
   (1 <=? getindex (mysched l++[b]) b = true)
   ↪  ->
    (sum (b::l) <? a = true) ->
    ((getindex (mysched (a::b::l)) a <?
    ↪  getindex (mysched (a::b::l)) b) =
    ↪  true).
Proof.
    intros. simpl. bdestruct (b + sum l <? a).
    + simpl. bdestruct (b =? a).
        -- discriminate H.
        -- bdestruct (sum l <? b).
            ++ simpl. bdestruct (b =? b).
                --- bdestruct (a =? a).
                    +++ apply Nat.ltb_lt. lia.
                    +++ bdestruct (a =? b).
                        ---- apply helper_one in
                           ↪  H9. destruct H9.
                                 reflexivity.
                        ---- apply helper_one in
                           ↪  H9. destruct H9.
                                 reflexivity.
                --- bdestruct (a =? a).
                    +++ apply Nat.ltb_lt. lia.
                    +++ bdestruct (a =? b).
                        ---- apply Nat.ltb_lt.
                           ↪  lia.
                        ---- apply Nat.ltb_lt.
                             apply Nat.leb_le
                               ↪  in H0.
                             apply Nat.leb_le
                               ↪  in H1.
                                 lia.
            ++ simpl. bdestruct (a =? a).
                --- apply Nat.ltb_lt. lia.
                --- apply Nat.leb_le in H2.
                    apply Nat.leb_le in H3.
                    lia.
    + simpl. bdestruct (sum l <? b).
        -- apply Nat.leb_le in H4. simpl in H4.
            apply helper_two in H4. destruct
            ↪  H4. apply H5.
        -- apply Nat.leb_le in H4. simpl in H4.
            apply helper_two in H4. destruct
            ↪  H4. apply H5.
Qed.
```

This theorem guarantees that when a new item task b is added to an existing list of tasks, the mysched algorithm will make sure that the priority ordering of task a and the newly added task b will be maintained. If a has a higher priority than the new task b, it will maintain that status in the list.

This theorem checks different scenarios to confirm this property. It considers where tasks a and b are positioned in the output list of mysched, their priorities relative to each other, and how they compare to other tasks in the list.

The theorem *mysched_propone* is focused on proving that the mysched algorithm maintains the correct order between tasks a and b in the output list, even after adding task b to an existing list of tasks. The proof examines various scenarios to ensure this correctness:

(1) If the combined priority of task b and the tasks in the list is lower than the priority of task a, then the proof makes sure that a retains its priority over task b in the resulting list.
(2) If the combined priority of task b and the tasks in the list is greater than the priority of task a, then the proof makes sure that b gets priority over task a in the resulting list.
(3) Additional conditions are considered to handle cases where tasks a and b have equal priority or are not present in the list, ensuring the correctness of the algorithm under various circumstances.

The proof relies on breaking down scenarios using boolean decomposition and employing helper lemmas to address each case. By doing so, it demonstrates that the mysched algorithm successfully preserves the relative ordering of tasks a and b based on their priorities, therefore confirming the algorithm's correctness even after task b is added to the list.

2.2 Round-Robin Scheduling

2.2.1 *Algorithm.*

Algorithm 2 Round-Robin

```
// Initialize variables
$ready\_queue \leftarrow$ Queue for processes ready to execute
$curr\_proc \leftarrow$ None
$quantum \leftarrow 5$
while $ready\_queue$ is not empty do
    $curr\_proc \leftarrow ready\_queue.dequeue()$
    $remaining\_time \leftarrow \min(quantum, curr\_proc.remaining\_burst)$
    for t $\leftarrow$ 1 to $remaining\_time$ do
        $curr\_proc.remaining\_burst \leftarrow curr\_proc.remaining\_burst - 1$
        if $curr\_proc.remaining\_burst = 0$ then
            // Process finished, no longer in the ready queue
            break
        end if
    end for
    if $curr\_proc.remaining\_burst > 0$ then
        // Process not finished, enqueue back to the ready queue
        $ready\_queue.enqueue(curr\_proc)$
    end if
end while
// All processes finished, algorithm ends
```

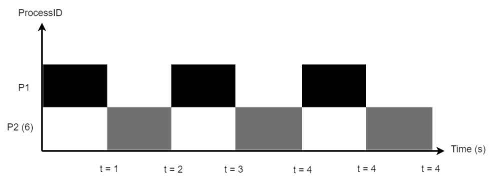

Figure 2: Round-robin Algorithm Execution Example

Process ID	Arrival Time	Burst Time
1	0	3
2	0	3

Table 2: Example Process Table

The Round-Robin scheduling algorithm assigns a fixed time unit, known as quantum, to each process during which it can execute. If a process does not complete within its quantum, it is temporarily suspended, and the next process in the queue is given a chance to execute. The suspended process is then placed at the end of the ready queue to await its next turn. This process continues until all processes have completed execution. The pseudo code shown in Algorithm 2 shows a possible implementation of the Round-Robin algorithm and will not necessarily be the exact specification used for the Coq implementation.

We can see from Table 2 that the two processes—P_1 and P_2—both have the same burst times of 3 ms and arrive at 0 ms. It is also evident that both processes alternately run for 1 ms each in the order P_1 and P_2, respectively until their burst times are satisfied. Since they both have a burst time of 3 ms, it is expected that the last process, P_2, would end after 6 ms, which is the case.

2.2.2 *Coq Translation.*

```
Fixpoint rr (l1 : list nat) (l2 : list nat)
↪  (n : nat) : list nat :=
match n with
| O => []
| S n' =>    match l1, l2 with
             | [], [] => []
             | [], h::t => h::(rr [] t n')
             | h::t, [] => h::(rr t [] n')
             | a::b, c::d => a::c::(rr b d n')
             end
end.
```

The fixpoint rr recursively implements an implementation of a Round-robin algorithm. The fixpoint rr takes in two lists of the nat type, takes the first element of each list, and concatenates them into an output list of type nat.

2.2.3 *Soundness Property.*

For the theorem on the algorithms soundness, we will define two axioms that will assist in formulating the proof.

Axiom 1. *If it is true that all id's in list l1 is ID i, then the length of adding another job with id i to l1 is 1 + the length of previous joblist.*

Axiom 2. *If it is true that all id's in list l1 is id i, then it is true as well for the tail of the list.*

We will need another axiom for the second argument of the algorithm.

Axiom 3. *If it is true that all id's in list l2 is ID i, then the length of adding another job with id i to l2 is 1 + the length of previous job list.*

For the theorem on soundness, we have the following definition.

Theorem 2.2. *For all lists l1, l2, and an element x: If the length of the result of a function rr applied to an empty list [], l2, and x is 0, and If the uniform function applied to l1 is true, Then the length of l1 is equal to the length of the result of the rr function applied to l1, l2, and x.*

```
Axiom firstaxiom: forall l1 l2 x a, uniform
↪  l1 1 = true ->
   getlength (rr (a :: l1) l2 x) 1 = S
    ↪  (getlength (rr (l1) l2 x) 1).

Axiom secondaxiom: forall l a, uniform (a ::
↪  l) 1 = true -> uniform (l) 1 = true.

Theorem soundnessone : forall l1 l2 x,
        0 = getlength (rr [] l2 x) 1
        -> uniform l1 1 = true
```

```
          -> (length l1) = (getlength (rr l1 l2
          ↪  x) 1).
Proof.
    intros. induction l1.
    - simpl. apply H.
    - simpl. rewrite firstaxiom. rewrite IHl1.
    ↪  reflexivity.
        apply secondaxiom in H0. apply H0.
        apply secondaxiom in H0. apply H0.
Qed.

Axiom thirdaxiom: forall l1 l2 x a, uniform
↪  l2 2 = true ->
    getlength (rr l1 (a::l2) x) 2 = S
    ↪  (getlength (rr (l1) l2 x) 2).

Axiom fourthaxiom: forall l a, uniform (a ::
↪  l) 2 = true -> uniform (l) 2 = true.

Theorem soundnesstwo : forall l1 l2 x,
        0 = getlength (rr l1 [] x) 2
        -> uniform l2 2 = true
        -> (length l2) = (getlength (rr l1 l2
        ↪  x) 2).
Proof.
    intros. induction l2.
    - simpl. apply H.
    - simpl. rewrite thirdaxiom. rewrite IHl2.
    ↪  reflexivity.
        apply fourthaxiom in H0. apply H0.
        apply fourthaxiom in H0. apply H0.
Qed.
```

In the Coq code, we can see the use of Axioms within Coq's proof environment. Axioms are crucial since they allow users to express mathematical ideas and theories that may not be fully captured by the system's built-in rules and definitions. However, it must be noted that careful selection and reasoning about axioms are essential to ensure that the formalized theories are both complete and consistent.

The first axiom states that if all id's in the list are 1, then adding another job with id 1 in the list will result to its length being one more than the length of the previous job list

The first Soundness theorem states that if all id's in the list are 1, then applying the round-robin algorithm to the list will result in a list where the number of jobs with id 1 is equal to the length of the list.

We can observe that the third and fourth axioms and the second Soundness theorem are counterparts for the second list input.

3 CONCLUSIONS

We have demonstrated the application of formal verification techniques using Coq to ensure the correctness and soundness of a custom scheduling algorithm, referred to as mysched, and of a Round-Robin scheduling algorithm in a simpler case. By mathematically modeling these algorithms and proving their properties using Coq, we have shown that formal methods can be effectively used to ensure that scheduler implementations adhere to specifications and function correctly based on how they were defined.

We have selected a subset of properties to be proven through the formal verification process over a simplified model of schedulers. We, nonetheless, have highlighted the importance of scheduling algorithms in operating systems and the need for rigorous verification methods to ensure their dependability. The formal verification of these algorithms is crucial for ensuring the correctness and reliability of operating systems, which are integral to various computer systems and applications worldwide.

Moving forward, the study can be expanded to more general cases. Some cases future researchers could consider would be varying the arrival time and total burst times of the processes. This, however, entails increased complexity in creating the proofs. The theorems demonstrated could also serve as a starting point for proofs of the general case. The general case is desired since it models actual OS behavior.

The use of formal verification techniques, as demonstrated in this study, will continue to play a vital role in ensuring the safety and functionality of computer systems in various applications and industries.

REFERENCES

[1] Md. Zahangir Alam, Mahfuzulhoq Chowdhury, and Parijat Prashun Purohit. 2014. Development of an Intelligent Traffic Management System Based on Modified Round-Robin Algorithm. *International Journal of Control and Automation* 7, 12 (Dec. 2014), 121–132. https://doi.org/10.14257/ijca.2014.7.12.12

[2] Remzi H. Arpaci-Dusseau and Andrea C. Arpaci-Dusseau. 2018. *Operating Systems: Three Easy Pieces.* CreateSpace Independent Publishing Platform, North Charleston, SC, USA. https://doi.org/10.5555/3299537

[3] Kimaya Bedarkar, Mariam Vardishvili, Sergey Bozhko, Marco Maida, and Bjorn B. Brandenburg. 2022. From Intuition to Coq: A Case Study in Verified Response-Time Analysis 1 of FIFO Scheduling. In *2022 IEEE Real-Time Systems Symposium (RTSS).* IEEE. https://doi.org/10.1109/rtss55097.2022.00026

[4] S Bharathi, MP Chethan, and SN Darshan. 2022. Comprehensive Analysis of CPU Scheduling Algorithms. *International Research Journal of Modernization in Engineering Technology and Science* (2022).

[5] Yosef Hasan Jbara. 2019. A new Improved Round Robin-Based Scheduling Algorithm-A comparative Analysis. In *2019 International Conference on Computer and Information Sciences (ICCIS).* IEEE. https://doi.org/10.1109/iccisci.2019.8716476

[6] Sukhveer Kaur, Krishan Kumar, Japinder Singh, and Navtej Singh Ghumman. 2015. Round-robin based load balancing in Software Defined Networking. In *2015 2nd International Conference on Computing for Sustainable Global Development (INDIACom).* 2136–2139.

[7] Gerwin Klein, Kevin Elphinstone, Gernot Heiser, June Andronick, David Cock, Philip Derrin, Dhammika Elkaduwe, Kai Engelhardt, Rafal Kolanski, Michael Norrish, Thomas Sewell, Harvey Tuch, and Simon Winwood. 2009. seL4: formal verification of an OS kernel. In *Proceedings of the ACM SIGOPS 22nd symposium on Operating systems principles (SOSP09).* ACM. https://doi.org/10.1145/1629575.1629596

[8] Haiyong Sun and Hang Lei. 2020. Formal verification of a task scheduler for embedded operating systems. *Journal of Intelligent amp; Fuzzy Systems* 38, 2 (Feb. 2020), 1391–1399. https://doi.org/10.3233/jifs-179502

[9] The Coq Development Team. [n. d.]. *The Coq Reference Manual.*

[10] Artem Yushkovskiy. 2018. Comparison of Two Theorem Provers: Isabelle/HOL and Coq. In proceedings of the Seminar in Computer Science (CS-E4000), Aalto Univeristy, Autumn 2017. (2018). arXiv:arXiv:1808.09701