Formal Verification of Shortest Job First Scheduling Algorithm in Coq

Jian Lawrence Luteria
University of the Philippines
jgluteria@up.edu.ph

Earl Wilbur Nogra
University of the Philippines
eanogra@up.edu.ph

Andrei Tiangco
University of the Philippines
altiangco@up.edu.ph

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

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

ABSTRACT

Schedulers play a critical role in operating systems, particularly in real-time systems where timing guarantees are vital. This paper formally verifies the non-preemptive Shortest Job First (SJF) scheduling algorithm using the Coq proof assistant. Our focus is on two key properties: permutation (ensuring each job appears once in the schedule) and sortedness (scheduling jobs with the shortest burst times first). Through Coq verification, we establish the SJF algorithm's reliability. While this work explores the non-preemptive SJF variant, our methodology can extend to more complex scheduling algorithms and real-world scenarios with dynamic job arrivals and preemption capabilities.

KEYWORDS

Formal verification, correctness, scheduling, operating system

1 INTRODUCTION

An operating system (OS) serves as a fundamental software layer that manages computer hardware and provides a platform for various applications. Central to the functioning of an OS is the scheduler, which determines the allocation of the CPU to different processes [12]. Scheduling is the process of efficiently managing and prioritizing tasks in a computing system.

Different scheduling algorithms, such as First-In-First-Out (FIFO) and Round Robin, contribute to this process by implementing distinct strategies for task execution. FIFO follows a first-come, first-served approach, while Round Robin ensures fair CPU time distribution by allocating fixed time slices to each process in a circular manner.

Another common algorithm used for scheduling is the Shortest Job First (SJF) algorithm, which selects the waiting process with the smallest execution time to execute next [11]. It is particularly suitable for jobs with predetermined run times [10]. In this approach, each process in the ready queue is executed based on its shortest burst time, minimizing waiting times for individual processes and consequently reducing the overall average waiting time.

As operating systems evolve into complex programs, ensuring their adherence to intended functionality becomes crucial [4]. Kumar et al. [7] highlighted the complexity of safety-critical systems such as nuclear power plant control software, emphasizing the necessity to model these systems prior to implementation. Similarly, Juvva K. [5] emphasized the crucial nature of timely processing in real-time systems such as air traffic control, stating that "a missed deadline in hard real-time systems is catastrophic". These case studies reinforce the need of a verified process scheduling algorithm to minimize the risk of failure and enhance system safety.

The method of formal verification can be used to provide high assurance to critical software [4, 6]. It involves rigorously proving that a system accords with its formal specifications or properties through the use of automated tools. The proof assistant Coq provides a formal language for expressing logical statements and a mechanism for constructing machine-checked proofs [13]. It ensures elimination of human errors and comprehensive consideration of all pertinent cases.

Formal verification has ensured reliability across the domain of scheduling and algorithm design [3, 8]. A recent formal correctness proof in Coq focused on verifying a similar scheduling algorithm via the Earliest Deadline First (EDF) scheduler working in real-time systems [14]. The proof methodology is organized into distinct modules, addressing aspects such as the *election function*, which governs the selection of the next task to be scheduled based on certain criteria; the driver or *backend* code, responsible for the main execution of the scheduling algorithm; and *assumptions*, which encompass the foundational conditions crucial for the correct functioning of the scheduler. The authors arguably become the first to showcase a formally proved correct implementation of an EDF scheduler, a methodology they assert to be "general enough to be applied to other schedulers or other types of system code".

In another case study, Bedarkar et al. [1] verified the response-time analysis of First-In First-Out (FIFO) scheduling in Coq, showcasing the feasibility of formal verification in the domain. Their findings emphasize the importance of domain-specific libraries such as PROSA. This reinforces the idea that formal verification is not only achievable but efficient for real-time scheduling theory.

In recent years, formal verification has gained prominence as a reliable method for validating system correctness. This paper addresses the challenge of ensuring correctness in systems and critical software by leveraging formal verification techniques. The specific focus of this paper is to formally verify the SJF algorithm using the functional programming language in Coq.

1.1 Contributions

This study addresses the growing need for assurance in operating system scheduling, particularly for safety-critical applications where timing guarantees are essential. By employing the Coq proof assistant, we present a formal verification of a non-preemptive SJF scheduling algorithm. This formal approach mathematically

verifies the algorithm's correctness. This work thus contributes to the field of reliable scheduling by establishing a methodology for the SJF algorithm's behavior, offering valuable insights for the development and verification of more complex scheduling policies.

2 BACKGROUND

This study explores the formal verification of the SJF scheduling algorithm, specifically its correctness using Coq. The concept of correctness pertains to the subproperties of permutation and sortedness.

(1) *permutation*: the exact correspondence of jobs in the output set with those in the input set
(2) *sorted*: the actual intended behavior of the algorithm, i.e. the ascending order of burst times in the scheduled jobs

Real-world scheduling implementations must contend with factors such as context switching overhead, which incurs additional time for the processor to switch between tasks [9]. However, to simplify the analysis and focus on the core principles of SJF, we assume negligible time for context switches. Additionally, the verification of the algorithm assumes that all incoming tasks arrive at time t = 0 with potentially same burst times. Lastly, this verification process is conducted in an offline mode, where process data is provided in predefined batches rather than in real-time.

3 METHODOLOGY

We detail the formalization of the SJF algorithm by defining the algorithmic steps and properties using Coq's syntax with natural language correspondences.

An informal representation of the SJF scheduling algorithm serves as the preliminary step in our methodology. The use of a pseudocode offers a high-level, human-readable description of the algorithm's logic, enabling clear comprehension of its operational steps [2]. This pseudocode serves as a blueprint for the subsequent translation of the algorithm into the Coq proof assistant as shown in Figure 1. Through this translation process, we are able to encode the algorithm's logic into Coq.

3.1 Formalization of SJF Algorithm

Definition of Jobs. We begin by defining a *job* data type that encapsulates the essential properties of a job, including its ID and burst time. Hereafter, the list of jobs coming in and out of the scheduler is referred to as `joblist` which denotes a sequence of job elements, building upon Coq's built-in `list` structure.

```
Inductive job : Type :=
  | taskj (id : nat) (burst_time : nat).

Definition joblist := list job.
```

SJF Scheduler. We define the SJF scheduler function, which takes a list of jobs as input and produces a sorted list according to ascending burst times. This function employs the insertion sort subroutine, ensuring that shorter jobs are prioritized over longer jobs.

```
Fixpoint insert_job (j : job) (l : joblist) :=
  match l with
  | nil => j :: nil
  | h :: t => if leb (get_burst j) (get_burst h)
  then j :: h :: t else h :: insert_job j t
  end.

Fixpoint sjf (lst : joblist) : joblist :=
  match lst with
  | nil => nil
  | h :: t => insert_job h (sjf t)
  end.
```

3.2 Correctness Proof of SJF Algorithm

The correctness proof of the algorithm is subdivided into proving its permutation and sorted properties.

Permutation Property. We prove that applying the SJF scheduler function to a list of jobs results in a permutation of the original list. This property ensures that the scheduler produces a valid schedule without discarding or duplicating jobs.

Lemma 3.1. *For all x and l, the permutation of inserting x into list l is equivalent to x followed by the result of inserting x into the rest of the list.*

Proof. We prove by induction.
Base case: If l is empty, the permutation of inserting x into an empty list is x followed by the empty list itself, which is reflexive.
Inductive case:

- If the burst time of x is less than or equal to the burst time of the first element of l, then inserting x into l results in the permutation being the same as the original list.
- Otherwise, if the burst time of x is greater than the burst time of the first element a of l, then the permutation of inserting x into l is equivalent to swapping x with a, then inserting x into the rest of the list.

□

```
Lemma insert_perm: forall x l, Permutation
(x :: l) (insert_job x l).
Proof.
  induction l.
  - simpl. apply Permutation_refl.
  - simpl. bdestruct (leb (get_burst x)
  (get_burst a)).
    ++ apply Permutation_refl.
    ++ assert (R: Permutation (x :: a :: l)
    (a :: x :: l)).
        { apply perm_swap. }
       rewrite -> R. apply perm_skip. apply IHl.
       Qed.
```

With the aid of the `Permutation` module from Coq's standard library, the lemma `insert_perm` is provided to inductively show that inserting a job into a list preserves the permutation of the list elements. In the base case, when the list is empty, the lemma trivially holds as inserting a job into an empty list results in a singleton list, which is permutation-equivalent to itself. For the inductive step, when the list contains elements, the lemma utilizes the `bdestruct` tactic to analyze whether the burst time of the job to be inserted is less than or equal to the burst time of the first element in the list. If this condition holds, the lemma again trivially holds as the insertion

```
// Initialize variables
ready_queue <- queue for processes ready
    to execute
current_process <- None
// Sort the ready queue from its shortest
    burst time to longest burst time
    through
// insertion sort
for j <- 2 to length of ready queue do
    key <- ready_queue[j]
    i <- j - 1
    while i > 0 and ready_queue[i] > key
        do
        ready_queue[i + 1] <-
            ready_queue[i]
        i <- i - 1
    end while
    ready_queue[i + 1] <- key
end for
```

```
Fixpoint insert_job (j : job) (l : joblist) :=
  match l with
  | nil => j :: nil
  | h :: t => if leb (get_burst j) (get_burst
      h) then j :: h :: t else h ::
      insert_job j t
  end.

Fixpoint sjf (lst : joblist) : joblist :=
  match lst with
  | nil => nil
  | h :: t => insert_job h (sjf t)
  end.
```

Figure 1: Side by side comparison between pseudocode and Coq implementation of SJF algorithm

operation does not alter the relative order of elements. However, if the condition is false, the lemma utilizes the perm_swap lemma to swap the positions of the job to be inserted and the first element in the list, establishing a permutation between the list before and after the insertion operation. Finally, the lemma recursively applies the induction hypothesis to the remaining elements of the list, ensuring that the permutation property is preserved throughout the insertion process.

Lemma 3.2. *For any job a, and joblists x and y, if x is a permutation of y, then inserting a into x yields the same permutation as inserting a into y.*

Proof. We prove by cases.

(1) If the lists x and y are the same, inserting a into both lists results in the same list.
(2) If x is a permutation of x_0 and x_0 is a permutation of y, then inserting a into x is equivalent to inserting a into x_0 twice.
(3) If x is obtained from x_0 by swapping two elements and x_0 is a permutation of y, then inserting a into x is equivalent to swapping a with those two elements in y, followed by inserting a into the rest of the list.
(4) If x is obtained from x_0 by swapping two elements and x_0 is obtained from y by swapping two different elements, then inserting a into x is equivalent to inserting a into x_0 followed by applying the same swaps.

Thus, in all cases, inserting a into x yields the same permutation as inserting a into y. □

```
Lemma lem2 : forall a x y, Permutation x y ->
Permutation (insert_job a x) (insert_job a y).
Proof.
  intros. simpl. inversion H.
  - apply Permutation_refl.
  - simpl. bdestruct (leb (get_burst a)
  (get_burst x0)).
    + apply perm_skip. apply perm_skip. apply H0.
    + apply perm_skip. rewrite <- insert_perm.
    rewrite <- insert_perm. apply perm_skip.
    apply H0.
  - rewrite <- insert_perm. rewrite <-
  insert_perm.
  rewrite -> H0. rewrite -> H1. apply perm_skip.
  apply H.
  - rewrite <- insert_perm. rewrite <-
  insert_perm.
  apply perm_skip. apply H. Qed.
```

Another lemma, lem2, establishes that inserting a job into two permutation-equivalent lists results in lists that are also permutation-equivalent. The proof begins by applying induction on the permutation between the two lists. In the base case, where the permutation is reflexivity, the lemma trivially holds as inserting a job into lists that are already permutation-equivalent yields lists that remain permutation-equivalent. For the inductive steps corresponding to transitivity and symmetry of permutations, the lemma employs the insert_perm lemma to ensure that inserting a job into the corresponding lists preserves their permutation equivalence. Additionally, when the permutation involves swapping adjacent elements, the lemma utilizes the perm_skip tactic to insert the job while maintaining the permutation property.

Theorem 3.3. *For any joblist l, l is a permutation of $(sjf\ l)$.*

Proof. We prove by induction.

Base Case: If the list l is empty, then l is a permutation of itself, which is reflexive.

Inductive Case: If l is obtained by inserting an element into the list l_0 and l_0 is a permutation of (sjf l_0), then l is also a permutation of (sjf l).

By induction hypothesis, l_0 is a permutation of (sjf l_0). Then, by Lemma lem2, inserting an element into l_0 yields the same permutation as inserting it into (sjf l_0). Hence, l is a permutation of (sjf l). Thus, for all cases, l is a permutation of (sjf l). □

```
Theorem sjf_perm: forall l, Permutation l (sjf l).
 Proof.
   intros. induction l.
   - simpl. apply Permutation_refl.
  - simpl. rewrite -> insert_perm. apply lem2.
   apply IHl. Qed.
```

Now we prove the permutation property through induction: in the base case, it demonstrates that applying the SJF scheduler to an empty list results in the identity permutation. In the inductive step, the proof utilizes the `insert_perm` lemma and recursively applies the induction hypothesis to establish the preservation of permutation when scheduling non-empty lists.

Sorted Property. We prove that the sorted list produced by the SJF scheduler satisfies the SJF criteria, wherein jobs with shorter burst times precede those with longer burst times.

```
Inductive sorted : joblist -> Prop :=
  |sorted_nil : sorted [ ]
  |sorted_1 : forall x, sorted [x]
  |sorted_cons : forall x y l, get_burst x <=
 get_burst y -> sorted (y::l) -> sorted (x::y::l).
```

First we define the concept of a sorted list of jobs within Coq, inductively defined by the `sorted` predicate. This predicate asserts that a list of jobs is sorted if it satisfies one of three conditions: either it is an empty list, denoted by `sorted_nil`; it contains a single job, represented by `sorted_1`; or for any two consecutive jobs `x` and `y` along with the remaining list `l`, if the burst time of job `x` is less than or equal to that of job `y`, and the rest of the list `l`, is also sorted, then the entire list consisting of `x`, `y`, and `l` is considered sorted.

Lemma 3.4. *For any job a and any joblist l, if l is sorted in ascending order of burst times, then inserting job a into l while preserving the sorted order results in a new sorted list.*

Proof. We begin by induction on the list l.

Base Case: When l is an empty list, inserting a creates a singleton list containing only a, which is trivially sorted.

Inductive Step: Suppose l is non-empty and consists of a head job x and a tail list of jobs l'. We assume that inserting a into l' while preserving the sorted order yields a new sorted list.

Now, we consider two cases based on the burst time comparison between a and x:

- If the burst time of a is less than or equal to the burst time of x, then a should be inserted before x to maintain the sorted order. We insert a at the beginning of l and recursively apply the induction hypothesis to the tail list l'.
- If the burst time of a is greater than the burst time of x, we continue recursively with the induction on the tail list l'.

□

```
Lemma insert_sorted : forall a l, sorted(l) ->
sorted (insert_job a l).
Proof.
  intros a l S. induction S; simpl.
  + apply sorted_1.
  + bdestruct (leb (get_burst a) (get_burst x)).
    - apply sorted_cons.
      ++ apply H.
      ++ apply sorted_1.
    - apply sorted_cons.
      ++ lia.
      ++ apply sorted_1.
  + bdestruct (leb (get_burst a) (get_burst x)).
    - apply sorted_cons.
      ++ apply H0.
      ++ apply sorted_cons.
          -- apply H.
          -- apply S.
   - bdestruct (leb (get_burst a) (get_burst x)).
    ++ bdestruct (leb (get_burst a) (get_burst y)).
          -- apply sorted_cons.
              +++ lia.
              +++ apply sorted_cons.
                  ---- apply H2.
                  ---- apply S.
          -- lia.
    ++ bdestruct (leb (get_burst a) (get_burst y)).
          -- apply sorted_cons.
              +++ lia.
              +++ apply sorted_cons.
                  ---- apply H2.
                  ---- apply S.
          -- apply sorted_cons.
              +++ apply H.
             +++ simpl. unfold insert_job in IHS.
              bdestruct (leb (get_burst a)
              (get_burst y)).
                    ----- lia.
                    ----- apply IHS. Qed.
```

Next we define the `insert_sorted` lemma, which states that inserting a job into a sorted list while preserving its sorted order results in another sorted list. The proof of this lemma is carried out through induction on the input job list. In the base case, when the input list is either empty or consists of a single job, the lemma trivially holds as inserting a job maintains the sorted property. For the inductive step, we consider the case where the input list contains multiple jobs, and we demonstrate that inserting a new job into this list while preserving its sorted order ensures that the resulting list remains sorted. This proof involves utilizing boolean destruction or the `bdestruct` tactic to handle cases involving comparisons of burst times between jobs.

Theorem 3.5. *For any joblist l, the schedule generated by (sjf l) will be sorted.*

Proof. Base case: If the list l is empty, there's nothing to sort, so it is trivially considered sorted.

Inductive case:

- Assume that when applying the sjf function to a joblist l, the resulting schedule will be sorted. In other words, (sorted (sjf l)) is true.
- The sjf function schedules jobs in accordance to their burst times. It needs to be proven that when job a is inserted into l, the overall schedule remains sorted. Through using the lemma insert_sorted, this shows that the entire schedule with the job a inserted remains sorted.

□

```
Theorem sjf_sorted: forall l, sorted (sjf l).
Proof.
  intros. simpl. induction l.
    - simpl. apply sorted_nil.
    - simpl. apply insert_sorted. apply IHl. Qed.
```

As in the permutation proof, we now verify through induction that that the SJF algorithm produces a sorted list of jobs according to ascending burst times. In the base case, it establishes that the scheduler produces a sorted list when applied to an empty list via the `sorted_nil` case. In the inductive step, the proof simplifies the list and recursively applies the `insert_sorted` lemma, which ensures that inserting a job into a sorted list maintains its sorted property.

Correctness Property. Building upon the individual proofs of permutation and sortedness, we define a predicate `is_sorted_perm` that captures both properties for the given scheduling function. This predicate asserts that for any input list of jobs, applying the function results in a permutation of the original list and produces a sorted list according to the SJF criteria. By unifying these properties into a single theorem `sjf_correct`, we formally verify that the SJF scheduler satisfies both permutation and sorted requirements simultaneously.

Theorem 3.6. *Asserts that the sjf function is correct in the sense that it produces a schedule that fulfills two requirements:*

(1) *Theorem 3.3: All jobs from the original list are included in the schedule exactly once. There are no missing or duplicate jobs.*
(2) *Theorem 3.5: The jobs in the schedule are ordered by their burst times, with the shortest jobs scheduled first.*

Proof. sjf_perm guarantees the permutation property, and sjf_sorted guarantees the sortedness property. By applying these theorems, we establish that the sjf function satisfies both requirements for a correct SJF implementation. □

```
Definition is_sorted_perm (f : joblist -> list
job) :=forall al, Permutation al (f al) /\
sorted (f al).

Theorem sjf_correct: is_sorted_perm sjf.
Proof.
  unfold is_sorted_perm. intros al.
  split.
    + apply sjf_perm.
    + apply sjf_sorted.
    Qed.
```

4 DISCUSSION

The methodology outlined utilizes the Coq proof assistant to formally verify the correctness of a non-preemptive SJF scheduling algorithm. This approach offers several advantages over traditional testing methods.

By converting the SJF algorithm into Coq, we ensure a precise and unambiguous representation of its logic. This eliminates potential ambiguities that could arise from natural language descriptions or pseudo-code. Secondly, by verifying that the algorithm outputs a permutation of the input job list, we establish that all jobs are scheduled exactly once, avoiding issues like job starvation. Finally, by verifying that the jobs are sorted in ascending order of burst times within the schedule, we formally guarantee that the benefit of the SJF algorithm is adhered to—minimizing the average wait time for jobs in the system.

However, it is important to acknowledge the limitations of this approach. The verification focuses on a simplified, offline version of the SJF algorithm. Real-world scheduling often includes dynamic job arrivals and potentially more complex scheduling criteria. Future work could explore the verification of a less restrictive system, such as considering variable burst times, to extend the verification method to dynamic scheduling environments.

Furthermore, while Coq offers a high degree of assurance, the quality of the verification ultimately depends on the correctness of the initial formalization of the algorithm. Hence, diligent validation of the formalization is essential to ensure the overall validity of the formal verification process.

5 CONCLUSION

Throughout this paper, we have provided our methodology for creating a formal correctness proof for the Shortest Job First algorithm in Coq. Our approach involves defining data types for jobs, implementing the SJF scheduler function, and providing proof of correctness via permutation and sortedness properties. By directly embedding the SJF algorithm in Coq and conducting rigorous correctness proofs, we ensure the reliability and predictability of the scheduler's behavior. Furthermore, we have shown how our formalization serves as a foundation for future research and development in scheduling algorithms. We invite readers to examine the provided SJF code and accompanying proofs to verify our findings and contribute to the advancement of formal methods in computer science.

REFERENCES

[1] Kimaya Bedarkar, Mariam Vardishvili, Sergey Bozhko, Marco Maida, and Björn 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, 197–210.
[2] Andrew Dessler. 2024. *ATMO 321 PYTHON FOR ATMOSPHERIC SCIENCES*. LibreTexts.
[3] Simon Doherty, Lindsay Groves, Victor Luchangco, and Mark Moir. 2004. Formal Verification of a Practical Lock-Free Queue Algorithm. (2004).
[4] Yalin Hu and Robert Armstrong. 2011. A Survey of Formal Verification in Mission-critical, High-consequence Applications. (2011).

[5] Kanaka Juvva. 1998. Real-Time systems. (1998). https://users.ece.cmu.edu/koopman/dess99/realtime/
[6] Moez Krichen. 2023. A survey on formal verification and validation techniques for internet of things. *Applied Sciences* 13, 14 (2023), 8122.
[7] Pramod Kumar, Lalit Kumar Singh, and Chiranjeev Kumar. 2020. Performance evaluation of safety-critical systems of nuclear power plant systems. *Nuclear Engineering and Technology* 52, 3 (2020), 560–567.
[8] Peng Li, Binoy Ravindran, and Syed Suhaib. 2004. A Formally Verified Application-Level Framework for Real-Time Scheduling on POSIX Real-Time Operating Systems. *IEEE TRANSACTIONS ON SOFTWARE ENGINEERING* (2004).
[9] Divi Pruthvi. 2013. Reducing Context Switching Overhead by Processor Architecture Modification. *Advances in Electronic and Electric Engineering* (2013).
[10] Tri Dharma Putra. 2020. Analysis of Preemptive Shortest Job First (SJF) Algorithm in CPU Scheduling. *International Journal of Advanced Research in Computer and Communication Engineering* 9, 4 (2020), 41–45.
[11] Syed Shah, Ahmad Mahmood, and Alan Oxley. 2009. Hybrid Scheduling and Dual Queue Scheduling. (2009).
[12] Abraham Silberschatz, Peter Baer Galvin, and Greg Gagne. 2012. Operating system concepts. [Sl].
[13] The Coq Development Team. 2021. The Coq Proof Assistant Reference Manual – Version V8.13.1. (2021). http://coq.inria.fr
[14] Florian Vanhems, Vlad Rusu, David Nowak, and Gilles Grimaud. 2022. A Formal Correctness Proof for an EDF Scheduler Implementation. (2022), 281–292.