Non-Preemptive SJF Scheduling and the Efficacy of FIFO in Mitigating Starvation

Krisha Anne Chan, Enrico Baratang, Henry Adorna and Alfonso Labao

July 22, 2024

Non-preemptive SJF Scheduling and the Efficacy of FIFO in Mitigating Starvation

Krisha Chan, Enrico Baratang, Henry Adorna*, and Alfonso Labao*

University of the Philippines-Diliman, Diliman QC 1101, PH,
kechan1@up.edu.ph,
WWW home page: https://up.edu.ph/contact-us/

Abstract. Task scheduling is critical for operating system performance, determining task execution order on the CPU. The Non-preemptive Shortest Job First (SJF) algorithm aims to enhance efficiency by minimizing average waiting and turnaround times. However, SJF can lead to issues like deadlock and starvation, where tasks are indefinitely delayed. This case study models SJF behaviors to simulate and demonstrate these pitfalls. By defining axioms, functions, and properties, the study formalizes SJF scenarios and their negative implications. The study also examines the First-In-First-Out (FIFO) algorithm, showing its effectiveness in preventing deadlock and starvation by executing tasks in arrival order. Formal verification of these algorithms ensures system reliability and guides the development of robust scheduling policies.

Keywords: shortest job first, first-in-first-out, deadlock, starvation

1 Preliminaries

Task scheduling lies at the heart of operating system functionality, determining the order in which tasks are executed on the CPU. In real-world scenarios, such as managing a server or handling multiple user requests, efficient scheduling is paramount for system performance and user satisfaction. The SJF algorithm, which selects the shortest job (task) first, theoretically minimizes average waiting time and turnaround time, thus enhancing system efficiency.

However, the implementation of SJF is not without its pitfalls. One major concern is the potential for endless loops, deadlock, or starvation. Deadlock occurs when tasks are unable to proceed because they are waiting for resources held by other tasks, resulting in a system halt. Starvation happens when a task is perpetually delayed or denied access to resources it needs due to the scheduling algorithm's biases.

The aim of this case study is to more formally demonstrate this issue by first building a model that emulates the behaviors of a SJF task scheduler and then simulating the conditions that cause the starvation of low-priority tasks.

In our formal model, we define axioms, functions, and properties that capture the essence of SJF scheduling. Axioms may include assumptions about task arrival times, execution times, and resource availability. Functions represent the behavior of the scheduling algorithm, such as selecting the shortest job from the ready queue. Properties describe desirable system behaviors, such as absence of deadlock or fair allocation of resources. Specifically, we focus on scenarios where SJF may lead to deadlock or starvation. By formalizing these scenarios and proving their implications within our model, we aim to highlight the limitations of SJF and demonstrate the effectiveness of alternative scheduling algorithms like First-In-First-Out (FIFO) in avoiding these issues. FIFO, a non-preemptive algorithm, ensures fairness by executing tasks in the order they arrive, thereby mitigating the risk of deadlock and starvation associated with SJF.

In summary, the formal verification of scheduling algorithms like SJF is crucial for ensuring the reliability and robustness of operating systems. By applying formal methods, we can identify potential pitfalls, validate system behaviors, and inform the design of scheduling policies that meet performance and correctness requirements.

2 Formal Specification in ADT

With this section, the preliminaries are over, and we begin to formalize specifications using Abstract Data Types (ADT) as outlined in Alagar and Periyasamy's (2011) book, and the Coq proof assistant. To give context, an Abstract Data Type (ADT) is a concept in computer science used to define data structures in a way that abstracts away the details of their implementation. Think of it as a blueprint for a data structure that specifies what operations can be performed on the data and what those operations should do, without worrying about how these operations are implemented.

2.1 Task

A task is a unit of work that needs to be processed by the CPU. Conceptually, each task is characterized by several attributes:

- `id`: A unique identifier for the task.
- `arrivalTime`: The time at which the task arrives in the queue – the cost by which the tasks may be ranked when using SJFscheduling.
- `executionTime`: The time required for the task to complete execution once it starts.

Consider the ADT of the Task below:

```
extend Nat by
sorts Task

operations
  Task: Nat x Nat x Nat → Task
  id: Task → Nat
  arrivalTime: Task → Nat
  executionTime: Task → Nat
```

```

axioms
  forall t: Task, i: Nat, a: Nat, e: Nat
    id(Task(i, a, e)) = i
    arrivalTime(Task(i, a, e)) = a
    executionTime(Task(i, a, e)) = e
```

The axioms ensure that for any task `t` created with identifier `i`, arrival time `a`, and execution time `e`, the functions `id`, `arrivalTime`, and `executionTime` will return `i`, `a`, and `e` respectively.

2.2 Queue

The behavior of accessing a queue is `first-in-first-out` and as such this will serve as our means of demonstrating FIFO scheduling as well as the default list-esque container for tasks. All of the functions on queues shown depict quite typical queue behavior.

```
 1 sorts Queue
 2
 3 operations
 4   emptyQueue: → Queue
 5   enqueue: Task x Queue → Queue
 6   dequeue: Queue → Queue
 7   front: Queue → Task
 8   isEmpty: Queue → Bool
 9
10 axioms
11   forall q: Queue, t: Task
12     isEmpty(emptyQueue) = true
13     isEmpty(enqueue(t, q)) = false
14     front(enqueue(t, emptyQueue)) = t
15     front(enqueue(t, enqueue(t1, q))) = front(enqueue(t1, q))
16     dequeue(emptyQueue) = emptyQueue
17     dequeue(enqueue(t, emptyQueue)) = emptyQueue
18     dequeue(enqueue(t, enqueue(t1, q))) = enqueue(t, dequeue(
           enqueue(t1, q)))
```

The General Case: Empty Queue
The `emptyQueue` operation creates an empty queue, denoted by the fact that `line 12` is always true. This operation establishes the initial state of a queue where no tasks are present.

The Adding Case: Enqueue
Adding a task `t` to any queue q results in a non-empty queue (`line 13`). Now, if you enqueue a task t to an empty queue, `t` becomes the front task (`line 14`).

The Removing Case: Dequeue
Removing a task from the front of an empty queue or a queue with a single task returns the queue to its empty state (`line 16-17`). For a queue with multiple tasks, the correct task is removed while maintaining the order of the remaining tasks (`line 18`).

2.3 Scheduler

```
sorts Scheduler

operations
  Scheduler: Queue x Maybe Task x List[Task] → Scheduler
  readyQueue: Scheduler → Queue
  currentTask: Scheduler → Maybe Task
  completedTasks: Scheduler → List[Task]
  emptyScheduler: → Scheduler
  addTask: Task x Scheduler → Scheduler
  executeNextTask: Scheduler → Scheduler

axioms
  forall s: Scheduler, t: Task
    readyQueue(Scheduler(q, c, l)) = q
    currentTask(Scheduler(q, c, l)) = c
    completedTasks(Scheduler(q, c, l)) = l
    emptyScheduler = Scheduler(emptyQueue, None, [])
    addTask(t, Scheduler(q, c, l)) = Scheduler(enqueue(t, q),
       c, l)
    executeNextTask(Scheduler(q, None, l)) =
      if isEmpty(q) then Scheduler(q, None, l)
      else
        let t = selectShortestJob(q) in
        Scheduler(dequeue(q), Some t, t :: l)
    executeNextTask(Scheduler(q, Some t, l)) = Scheduler(q,
        Some t, l)
```

The Scheduler is a system that manages and executes tasks using a structured queue, as defined in **2.2**. The `executeNextTask` operation is pivotal in this process. When called, if no task is currently being executed and the queue is non-empty, it selects the shortest job from the queue (adhering to the Shortest Job First scheduling algorithm), sets this task as the current task, and moves it to the list of completed tasks. If the queue is empty or a task is already being executed, the scheduler's state remains unchanged. The axioms governing the scheduler's behavior focuses on optimal task execution.

2.4 Shortest Job First (SJF)

```
operations
  selectShortestJob: Queue → Task

axioms
  forall q: Queue, t: Task
    selectShortestJob(enqueue(t,q)) =
      if executionTime(t) < executionTime(front(q)) then t
      else front(q)
```

The operation `selectShortestJob` is designed to identify the task with the shortest execution time from a given queue. The axioms define its behavior: for any queue `q` and task `t`, when `t` is added to `q`, the operation checks if the execution time of `t` is less than the execution time of the task at the front of the queue. If it is, `t` becomes the selected task; otherwise, the task currently at the front remains the selected task.

3 Formal Specification in Coq

Overview of What Has Been Established So Far

```
SORT Task
SORT Queue
SORT Scheduler

OP emptyQueue:→ Queue
OP enqueue: Task x Queue → Queue
OP dequeue: Queue → Task
OP SJFSchedule: Queue → Scheduler
```

In this ADT, we define a **Task** sort to represent tasks, a **Queue** sort to represent the queue of jobs awaiting execution, and a **Scheduler** sort to represent the scheduling algorithm. We also define operations for manipulating the queue, such as `emptyQueue` to create an empty queue, `enqueue` to add a job to the queue, and `dequeue` to remove a job from the queue. Finally, we define the `SJFSchedule` operation to specify the behavior of the SJF scheduling algorithm.

To specify this behavior or prove a theorem formally, we use a tool called Coq Proof Assistant. Coq is somewhat like a program that helps automate and facilitate methods of mathematical proof. It allows you to write out your proof and checks the validity of your statements as you go. It provides an interactive environment where you can incrementally develop your proofs, receiving immediate feedback on each step. This interactivity ensures that any mistakes or incomplete steps are identified and corrected in real-time.

Translating this ADT to Coq gives us:

```
Inductive Task : Type := (*Represents a task *)
  |TaskType.

Inductive Queue : Type := (* Represents a queue of tasks *)
  | EmptyQueue
  | Enqueue (t : Task) (q : Queue).

Inductive Scheduler : Type := (*Represents the scheduling algortihm *)
  |SJFSchedule (q : Queue).
```

Now, let's define some properties of our scheduling algorithm in ADT language:

```
PROP NonEmptyQueue: Queue → bool
MEASURE length : Queue → Nat
AXIOM DequeueEnqueue: ∀ t : Task, ∀ q : Queue,
        dequeue(enqueue(t, q)) = t.
AXIOM SJFOptimality: ∀ q : Queue, NonEmptyQueue(q) →{∀ t :
    Task | t ∈ q }
        selectShortestJob(q) ≤ t.
```

We define a property `NonEmptyQueue` which checks if a queue is non-empty, and an axiom `DequeueEnqueue` stating that dequeuing after enqueuing a job results in the same job. The axiom `SJFOptimality` specifies that the SJF scheduling algorithm selects the shortest job in the queue.

We translate this into Coq:

```
Require Import Coq.Arith.Arith.
Require Import Coq.Lists.List.
Import ListNotations.

(* Define the type for tasks *)
Inductive Task : Set :=
| TaskElem : nat → Task.

(* Define the type for queues *)
Inductive Queue : Set :=
| EmptyQueue : Queue
| Enqueue : Task → Queue → Queue.

(*Function to calculate the length of a queue *)
Fixpoint length (q : Queue) : nat :=
  match q with
  | EmptyQueue ⇒ 0
  | Enqueue _ q' ⇒ S (length q')
  end.

(* Function to check if a queue is non-empty *)
```

```
Definition NonEmptyQueue (q : Queue) : bool :=
  match q with
  | EmptyQueue ⇒ false
  | _ ⇒ true
  end.

(* Function to enqueue a task in the queue *)
Definition enqueue (t : Task) (q : Queue) : Queue :=
  Enqueue t q.

(* Function to dequeue a task from the queue *)
Fixpoint dequeue (q : Queue) : option Task :=
  match q with
  | EmptyQueue ⇒ None
  | Enqueue t EmptyQueue ⇒ Some t
  | Enqueue _ q' ⇒ dequeue q'
  end.

(* Function to get the shortest task in the queue for SJF Scheduling *)
Fixpoint SJFSchedule (q : Queue) : option Task :=
  match q with
  | EmptyQueue ⇒ None
  | Enqueue t EmptyQueue ⇒ Some t
  | Enqueue t q' ⇒
    match SJFSchedule q' with
    | None ⇒ Some t
    | Some t' ⇒
      match t, t' with
      | TaskElem len1, TaskElem len2 ⇒
        if Nat.leb len1 len2 then Some t else Some t'
      end
    end
  end.
```

4 Formal Property Verification Using Coq

As briefly mentioned before, starvation is the state wherein a task is perpetually delayed because higher-priority tasks keep getting executed first. Think of it like a hobby that you never get to work on because you always seem to have more important things to do. The hobby is less important, and thus lower priority, than all the other tasks that get introduced into your life — like work, school, errands, etc, and as such, these tasks take precedence over the hobby and get addressed first. **FIFO does not have this**. Intuitively, if there is no mechanism of priority wherein one task can take precedence over another, where the addition of a new task cannot cause an older task to be pushed back on the to-do-list, then you cannot have an issue where an older task keeps getting pushed back, and going unaddressed.

4.1 Proof that FIFO can prevent starvation

```
Require Import Coq.Arith.Arith.
Require Import Coq.Lists.List.
Import ListNotations.

(* A record for representing a process *)
Record Process := {
  pid : nat; (* process identifier *)
  exec_time : nat (* execution time *)
}.

(* A function to insert a process into a sorted list of processes *)
Fixpoint insert_process (p : Process) (ps : list Process) : list Process :=
  match ps with
  | [] ⇒ [p]
  | p' :: ps' ⇒ if Nat.leb p.(exec_time) p'.(exec_time)
                 then p :: ps
                 else p' :: insert_process p ps'
  end.

(* A function to sort a list of processes using insertion sort *)
Fixpoint sort_processes (ps : list Process) : list Process :=
  match ps with
  | [] ⇒ []
  | p :: ps' ⇒ insert_process p (sort_processes ps')
  end.

(* A function to retrieve the first process in the list (FIFO) *)
Definition get_first_process (ps : list Process) : option Process :=
  match ps with
  | [] ⇒ None
  | p :: _ ⇒ Some p
  end.

(* A function to simulate fifo *)
Fixpoint execute_fifo (pid_to_watch : nat) (ps : list Process) : nat :=
  match ps with
  | [] ⇒ 0 (* Base case: process not found *)
  | p :: ps' ⇒
      (* If this is the process we're watching, return 1 *)
      if Nat.eqb p.(pid) pid_to_watch then 1
      else let steps := execute_fifo pid_to_watch ps' in
           S steps (* Increment the number of steps *)
  end.

(* List of processes *)
Definition list_processes : list Process :=
  [
    {| pid := 1; exec_time := 7 |};
  {| pid := 2; exec_time := 5 |};
  {| pid := 3; exec_time := 1 |};
  {| pid := 4; exec_time := 4 |};
  {| pid := 5; exec_time := 3 |}
 ].

(* Prove that any process will eventually be executed under FIFO *)

(* Helper function to return the position of a process in the list *)
Fixpoint position (pid_to_find : nat) (ps : list Process) : option nat :=
 match ps with
 | [] => None
 | p :: ps' => if Nat.eqb p.(pid) pid_to_find then Some 0
               else match position pid_to_find ps' with
                    | None => None
                    | Some pos => Some (S pos)
                    end
 end.

(* Lemma: A process in the list will have a finite position *)
Lemma position_finite: forall pid_to_find ps,
 In pid_to_find (map pid ps) -> exists pos, position pid_to_find ps = Some pos.
Proof.
 intros pid_to_find ps H.
 induction ps as [|p ps' IH].
 - simpl in H. contradiction.
 - simpl in H. destruct H as [H | H].
   + subst. simpl. rewrite Nat.eqb_refl. exists 0. reflexivity.
   + simpl. destruct (Nat.eqb_spec p.(pid) pid_to_find) as [Heq | Hneq].
     * subst. exists 0. reflexivity.
     * simpl. apply IH in H. destruct H as [pos Hpos].
       exists (S pos). simpl. rewrite Hpos. reflexivity.
Qed.

(* Theorem: FIFO prevents starvation *)
Theorem fifo_prevents_starvation: forall pid_to_find ps,
 In pid_to_find (map pid ps) -> exists steps, execute_fifo pid_to_find ps = steps.
Proof.
 intros pid_to_find ps H.
 induction ps as [|p ps' IH].
 - simpl in H. contradiction.
 - simpl in H. destruct H as [H | H].
   + subst. simpl. rewrite Nat.eqb_refl. exists 1. reflexivity.
   + simpl. destruct (Nat.eqb_spec p.(pid) pid_to_find) as [Heq | Hneq].
     * subst. exists 1. reflexivity.
     * simpl. apply IH in H. destruct H as [steps Hsteps].
       exists (S steps). simpl. rewrite Hsteps. reflexivity.
Qed.

(* Function to Print the Number of Steps for Each Process *)
```

```
 99 Fixpoint print_fifo_steps (ps : list Process) : list (nat ∗ nat) :=
100   match ps with
101   | []  ⇒ []
102   | p ::  ps’ ⇒ (p.(pid), execute_fifo p.(pid) list_processes) :: print_fifo_steps ps’
103   end.
104 
105 (* Function to extract the PIDs and execution times from a list of processes *)
106 Definition extract_pid_exec_times (ps : list Process) : list (nat ∗ nat) :=
107   map (fun p ⇒ (p.(pid), p.( exec_time))) ps.
108 
109 (* Sorting the example list of processes based on execution time *)
110 Definition sorted_processes := sort_processes list_processes.
111 
112 (* Function to find the index of a process in a list by PID *)
113 Fixpoint find_index (p : Process) (ps : list Process) (index : nat) : option nat :=
114   match ps with
115   | []  ⇒ None
116   | p’  ::  ps’  ⇒ if Nat.eqb p.(pid) p’.(pid) then Some index
117                  else find_index p ps’ (S index)
118   end.
119 
120 (* Function to create the original_pid → sorted position list *)
121 Definition original_to_sorted_positions (original sorted : list Process) : list (nat ∗ nat) :=
122   map (fun p ⇒ match find_index p sorted 1 with
123                 | Some index ⇒ (p.(pid), index)
124                 | None ⇒ (p.(pid), 0) (* Should not happen *)
125                 end) original.
126 
127 (* Execute Sorted List with Positions *)
128 Eval compute in original_to_sorted_positions list_processes sorted_processes.
129 
130 (* Execute Fifo Results *)
131 Eval compute in print_fifo_steps list_processes.
```

If we run that code into the CoqIDE, we will get the following results:

```
1      = [(1, 5); (2, 4); (3, 1); (4, 3); (5, 2)]
2      : list (nat ∗ nat)
3      = [(1, 1); (2, 2); (3, 3); (4, 4); (5, 5)]
4      : list (nat ∗ nat)
```

We can see that with the following list of processes in `lines 46-53` and the results, there are a few noticeable differences.

- The first result (`lines 1-2`) executes the job with the shortest execution time. This means that if a new job is added with a shorter execution time than those currently in the list, it will be executed first, causing the existing jobs to wait. If we keep adding jobs with shorter execution times, it creates a starvation problem where jobs with longer execution times are continuously starved or delayed. For instance, the first job in the list, which had an execution time of `7`, was executed **last** (`5th`) as shown in `(1, 5)`, while the third job in the list, with an execution time of 1, was executed **first** as shown in `(3, 1)`. The average time complexity for this execution is $O(n^2)$ as each element will need to be compared and potentially shifted about half of the elements in the sorted portion of the list.

- The second result (`lines 3-4`), however, does not follow this rule. Instead, it implements a first-in-first-out execution order, regardless of the job's execution time. Therefore, the first job gets executed first − `(1, 1)`, the second job gets executed second − `(2, 2)`, and so on. This gives us the time complexity of $O(1)$. This is much better if we're dealing with real life scenarios where consistent response time is needed. It can particularly be useful for index lookups in web applications, allowing for fast retrieval of data. In embedded systems such as automotive control units or medical devices, operations need to be completed within strict time constraints or it might lead to potentially life-threatening situations.

Now going back to the proof, let's discuss and break down the FIFO scheduling algorithm where no process will be indefinitely delayed or starved.

1. **Induction**: The proof starts by inducting on the list of processes `ps`. This means it considers two cases: when the list is empty and when it's not.
2. **Base Case**: If the list of processes is empty, it immediately concludes that the process ID `pid_to_find` cannot be found in an empty list, leading to a contradiction. This establishes the base case.
3. **Inductive Step**: It considers the non-empty list case.
   a. If the process ID `pid_to_find` is the same as the process ID of the first process in the list, it concludes that `pid_to_find` will execute in 1 step because it's the first process.
   b. If `pid_to_find` is not the same as the process ID of the first process, it recursively applies the induction hypothesis to the rest of the list of processes. It proves that in the remaining list, `pid_to_find` will eventually execute after a finite number of steps (represented by steps).
   c. It then concludes that in the current list, `pid_to_find` will execute after one more step than in the rest of the list, establishing that `pid_to_find` will execute in a **finite number of steps** overall.

By exhaustively considering the cases of an empty list and a non-empty list and proving that in each case, the process ID `pid_to_find` will execute in a finite number of steps, the proof demonstrates that FIFO prevents starvation by ensuring every process eventually gets its turn to execute.

5 Conclusion

The proof, as have been established by the previous sections, likens executing tasks in a certain order to creating an arranged list of tasks to be executed. For this proof, the implementations of the sorts defined above are altered and simplified to help aid in the proof. For instance the implementation of Task, Process, lacks the `arrival_time` field because this will be interpreted as its position in the arranged list.

Assuming the list of tasks to be done behaves like a queue, with new entries being appended to the end, the use of SJF scheduling could be likened to sorting the list in decreasing order based on `execution_time/cost` to create the schedule, and FIFO likened to using the queue as it is.

Basing the arrival time of any task as its position in that arranged list, or schedule, we can see that if we try and track one particular task, let us name this task "`Heavy`", that is strictly more costly than any other task in that list, we can see that applying FIFO makes it such that Heavy's arrival time, indicated by its position in the schedule, is solely dependent on the time or order when it was added to the queue of tasks to be done.

6 Limitations

So far, we have only established and proved that FIFO executes in a finite number of steps; hence, effectively preventing starvation. What has yet to be formally shown is that because SJF always picks the "shortest/lowest cost-task first", `Heavy`, being strictly longer/more costly than the other tasks, always is picked last. As such its arrival time becomes dependent on how many tasks of lower cost are in the queue of tasks to be done. So assuming an indefinite amount of lower cost tasks, analogous to a continuous stream of shorter tasks, `Heavy` is made to wait an indefinite amount of time before execution.

References

1. Alagar, V.S., Periyasamy, K.: Specification of Software Systems. Springer, London (2011). doi:10.1007/978-0-85729-277-3