Formal verification of the Brain Fuck Scheduler*

Meng Xuan Xia
meng.xia@mail.mcgill.ca

November 2014

Abstract

We modelled a simplified version of the Brain Fuck Scheduler (BFS) using Uppaal, an integrated tool environment for modeling, validation and verification of real-time systems modeled as networks of timed automata. An example of four non-realtime processes system was used to show that our simplified BFS model is verifiably starvation-free under this example. We discussed how our approach can be used to verify system of arbitrary set of processes. We also noted the limitations of our model, notably the impossibility to verify fairness and lack of multi-core execution modelling.

1 Introduction

Mainline Linux kernel uses Completely Fair Scheduler (CFS) as its default task scheduler. CFS has strong emphasis on scheduling fairness, as its name implies. However, this fairness property also comes with a cost of lower interactiveness and higher latency.

Brain Fuck Scheduler (BFS) is a Linux scheduler developed by Con Kolivas as an alternative to the default CFS used in vanilla mainline kernel. It is designed perform well on lower spec desktop computers . BFS favors low latency over rigidly calculated fairness. However, due to the way task look-up occur, BFS is unable to scale to machines with more than 16 logical cores. In a recent study done by graysky [gra12] in 2012, BFS is shown to outperform CFS in nearly all tests. We are interested to verify whether this performance gain comes with a cost of potentially negative side-effect, such

*Yes, this is how the original developer named it

as starvation and unfairness. Before proceeding with verifications, we shall first understand how BFS works as a task scheduler.

1.1 Tasks and Task Scheduling

In a typical Linux system, there are two types of tasks:

Realtime tasks are subjected to time constrains, such tasks have strict deadlines on when they should be processed. For example, a quad-copter controller's task is to read data from gyroscope, then use this information to adjust the speed of the rotors. This task is a realtime task, since the data read from the gyroscope is only valid for a very short period of time, it becomes useless to adjust the speed of the rotors when the quad-copter is in fact crashed.

Normal (non-realtime) tasks are not subjected to a time constrain. For example, when a user browses some web page, the time that the web page finishes loading is not mission critical, the user will only be slightly annoyed when the web page loads a few seconds later than expected.

Both realtime tasks and normal tasks can have priority parameters. In Linux, priorites are given by *nice* value. They typically range from -20 to 19. A task is less prioritized when it is nicer (has a higher nice value.)

There can be multiple tasks running simultaneously in a system. Thus, the role of the task scheduler is to give each task a fixed slice of time to run in a way that all tasks wanting to run will eventually get its turn to run – this is the starvation free property. In addition, we can also define fairness property, which simply states that every task of the same nice level must get the same share of running time.

1.2 BFS

We shall now take a look on how BFS does the scheduling for realtime tasks and normal tasks. A runqueue is a queue data structure where a task can be added and is also where the scheduler takes task from. The vanilla BFS implementation maintains 100 runqueues of realtime tasks, each correspond to a different realtime priority where the first runqueue is the most prioritized one and the last one is the least prioritized. To simplify modelling,

we abstracted the 100 runqueues in to only 1 realtime runqueue, where all realtime tasks all share the same priority. We will later explain why we can make such a simplification while keeping our model valid. BFS also has three runqueues, namely `SCHED_ISO`, `SCHED_NORMAL` and `SCHED_IDLEPRIO`. We will ignore `SCHED_ISO` and `SCHED_IDLEPRIO` in this paper because those two runqueues are unique to BFS, while other Linux schedulers do not have these features; programs need to be configured manually in order to be added to the above two special runqueues. `SCHED_NORMAL`, in turn, correspond to the runqueue of the normal tasks.

For BFS to pick the next task to run, these rules apply:

1. In a system with both realtime tasks and normal tasks, BFS always pick a task from the realtime task runqueue first. Until there are no more realtime task in the queue.

2. The realtime tasks are picked in a First Come First Serve basis.

3. To pick a normal task, BFS uses the virtual deadline mechanism. Every normal task in the system is given a virtual deadline, The deadlines are proportional to the nice level of the processes; the higher the nice level, the longer the deadline. These virtual deadlines specify how long each corresponding process is willing to wait in the runqueue. It is important to note that these deadlines are *virtual*, meaning that the system does not enter into a bad state even if some deadline is passed; the virtual deadlines are simply used as a mechanism for scheduling. To select a normal task to run, the scheduler first scans for the tasks that passed their virtual deadlines. If a task is found to pass his virtual deadline, it is selected to be run next. Otherwise, the scheduler selects the task that has the closest deadline.

Note that task selection solely does not make BFS fast, the scheduler also carefully selects the CPU core where the task will run on, in order to try to utilize all available cores. Tasks can also benefit from local cache of the CPU core when scheduled to run on the same core as it did in the previous time. We will not explore in detail with those technicalities. We are mostly interested by the correctness of the virtual deadline scheduling mechanism in BFS.

1.3 Related Work

In their paper "Verification of the legOS Scheduler using Uppaal" [HHI00], Halkjaer, Haervi and Ingolfsdottir used Uppaal to formally verify that legOS suffers from starvation problems with threads of lower priorities. Their work showed that they knew about the starvation problem from the beginning and was attempting to validate their knowledge by constructing a model where the starvation occurs. They then modelled the corrected scheduler in Uppaal and verified its correctness. Their approach is not reproducible in this BFS verification scenario because we do not know that whether BFS suffers from starvation or not. Nevertheless, their model of the system provides insight on how a model of a scheduler system resembles to.

Ke, Pettersson, Sierszecki and Angelov [KPSA08] worked on the verfication of COMDES-II Systems. The COMDES-II System is a embed system with a hard-deadline, where the deadlines must not be violated. This is different from the BFS's virtual deadline, where a violation is possible and *even necessary*[1] to ensure starvation-freedom.

Penix, Visser, Engstrom, Larson and Weininger [PVE+00] used Spin and Promela to formally verify a time partitioning bug in the DEOS that was detected and fixed in earlier formal review processes. In their approach, due to the lack of the notion of time in Spin/Promela, they had to implement a clock in Promela and synchronize the clock with the rest of the system. Yet, simulating the clock is one of the difficulties in their work, but such difficulty can be *easily avoided* by using timed-automata instead, which has the notion of clock built into it.

2 Formally verify BFS

We use *Uppaal* in our formal verification process. Uppaal is a integrated tool environment that allows one to create timed-automata, run simulations on those automata, finally uses Uppaal flavored Linear Temporal Logic (LTL) to specify the persistence properties and verify those properties on the timed automatons. Uppaal is developed in collaboration between the Department of Information Technology at Uppsala University, Sweden and the Department of Computer Science at Aalborg University in Denmark.

[1] Proving the necessity is left as an exercise for the reader.

In this paper, we will not introduce the full semantic of Uppaal, but rather explain the notation on a case by case basis. We invite the readers to read Halkjaer and his colleagues' paper [HHI00] which includes a fairly well written introductory material on Uppaal.

2.1 Modelling BFS

In order to formally verify BFS, one must first model the system running BFS using timed automaton. A timed automaton is a finite automaton[2] extended with a finite set of real-valued clocks. When a finite automaton is running, all clocks advance at the same speed. The values of the clocks can be compared with integers. These comparisons can be used as guards on the transition, i.e. a transition can be taken only if the clock-integer comparison satisfies. Guards provide effective way of constraining the possible behaviors of the automaton. Further more, it is possible to reset arbitrary clock value to zero.

There are two models in our system, one is the model for the scheduler and the other one is for the task/process. In a typical operating system, a process can be informally seen as a task, hence we will use the two terms process and task interchangeably from now on.

2.1.1 Process automaton

We shall first take a look at the process automaton, depicted in figure 1 on page 7. Every process has a waiting clock called $wait[i]$ where i uniquely identifies the process. This clock tracks the amount of time that the corresponding process waits in the runqueue. Each process is also initialized with two parameters: a boolean *realtime* and a integer *nice* where $-1 \leq nice \leq 1$. *Realtime* is set to *true* if the process is a realtime task, otherwise, it is set to *false*. *nice* is the nice value of the process.

We have a function named `getRRInterval` which takes a *nice* value and map it to the corresponding `RR_Interval`. A process's deadline is the current time plus its calculated `RR_Interval`. `RR_Interval` is proportional to the nice value.

[2] http://en.wikipedia.org/wiki/Finite-state_machine

A process starts in the *Added* state, this is indicated by the use of double circle. It can stay in this state as long as it does not exceed its deadline. This is indicated by the invariant $wait[p] \leq getRRInterval(task.nice)$ colored in purple. Once the deadline passed, it must transition to the *DeadlinePassed* state via a transition guarded by $wait[p] \geq getRRInterval(task.nice)$ in green.

If no other process is currently running and no realtime process is in the queue, it can take a transition to the *Runnable* state. The transition from *DeadlinePassed* to *Runnable* is labelled by *immediatelyRunnable!*, this symbol is a synchronization label. In short, every transition labelled by the same synchronization label must happen at the same time. A synchronization label ending with a exclamation mark actively triggers the synchronization, where as a label ending with a question mark passively waits for synchronization. The corresponding question mark label is used in the BFS automaton figure 2. A synchronization label can either be declared as normal or urgent. A timed-automaton always takes the urgent synchronization whenever possible.

On the other hand, an automaton may non-deterministically choose not to take a normal synchronization immediately. A process may also transition from Added state to Runnable state without going through the DeadlinePassed state. In this case, there are few possibilities:

1. First, it is only possible to transition if there is no other process running.
2. If the process is a realtime process and it is at the head of the realtime FIFO runqueue, then it can take the transition.
3. If the process p is a normal process and there are currently no item in the realtime queue, and that no other process q has passed its deadline, and p is the one that has the closest deadline among all other processes, then it can take the transition.
4. In all other cases, this transition is disabled.

When a process is Runnable state, it is *committed*, this is marked by a symbol C inside the circle. In a committed state, clocks do not advance and the next transition must happen from the states marked as *committed*. In the transition from Runnable to Running, we synchronize with BFS using *run?* label. Once synchronized, we run the *updateRunQueue* routine which removes the current task from the runqueue and then we set *hasRunning*

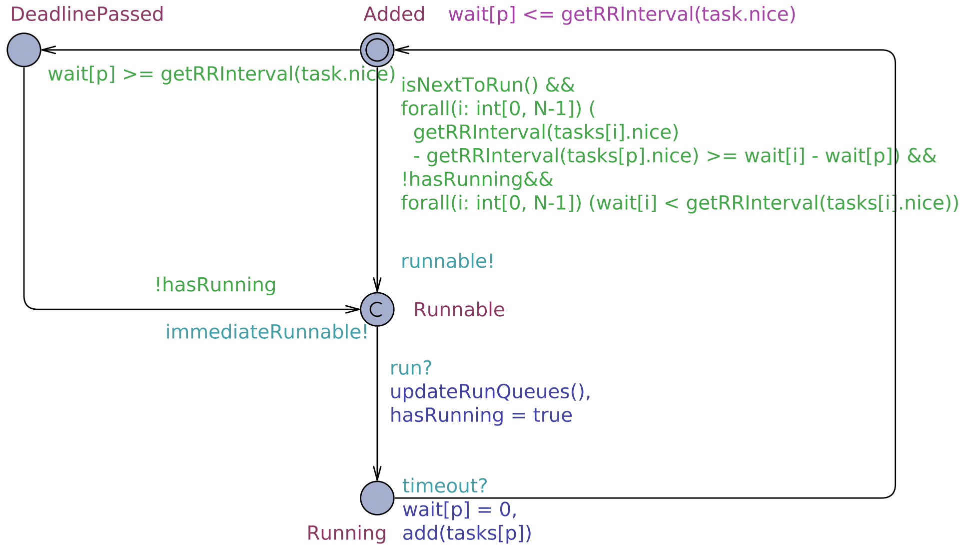

Figure 1: Process timed-automaton

equal to true, using it as a mutex so that only one process may be running at any time. The process stays in the Running state and waits for timeout synchronization from BFS. BFS only allows a process to run for a finite time slice. After the timeout occurs, we take the Running → Added transition, reset the process' wait clock to zero and then add the process back to the runqueue. This entire process repeats indefinitely.

2.1.2 BFS automaton

The BFS scheduler automaton is depicted in figure 2 on page 8. BFS has a clock t that tracks the amount of time a scheduled process has been running. BFS starts in *Init* state, it takes the transition to Idle states by initializing the data structures for storing the queues. It can transition to the *schedulable* state by either synchronizing with *immediatelyRunnable?* label or the *runnable?* label. Since immediatelyRunnable synchronization label is an urgent one, it is always prioritized. This makes sure that processes with passed deadline get to run as soon as possible. BFS in the schedulable state make a transition to scheduled state while synchronizing via the *run!* label. At the same time, the process that is in the runnable state and waiting for *run?* synchronization transitions to Running state. BFS can only stay in

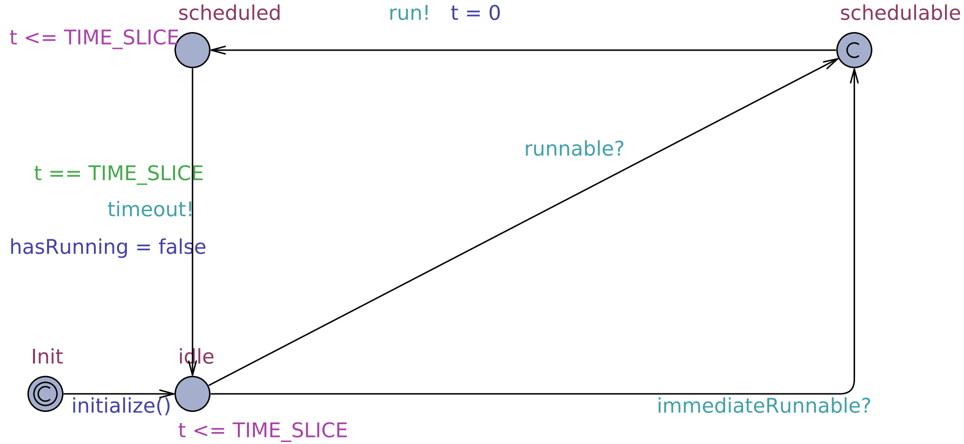

Figure 2: Process timed-automaton

the scheduled state for some time $t \leq TIME_SLICE$ where TIME_SLICE is a constant of the system. In our implementation, it is set to 4. It must take the transition to idle state when the time slice is reached. This transition synchronizes the *timeout* label, so that the corresponding running process also transition to its Added state. This entire process repeats until the system shuts down.

2.2 Verification of properties

Now suppose we have a working set of processes P_0, P_1, P_2, P_3 that are all non-realtime, we would like to formally verify whether this system, running these four processes with BFS, is indeed a starvation free run.

Process 0 has nice level -1 and its deadline is the current tick plus 5 ticks.

Process 1 has nice level 0 and its deadline is the current tick plus 10 ticks.

Process 2 has nice level 1 and its deadline is the current tick plus 20 ticks.

Process 3 has nice level 0 and its deadline is the current tick plus 10 ticks.

We start by formally defining the starvation-free property. A process p is said to be starvation-free if and only if always when p is in Added state implies that p will eventually be in the Running state. In Uppaal, this linear temporal logic is represented using the *lead to* ($\rightarrow$) notation. We also want to guarantee that a process does not run forever without going back to idle

state, we can formally express this as always p in Running state implies that eventually p in Added state. This is also expressible using the $\rightarrow$ notation. And lastly, we wish to guarantee that always there can be only one process running at any given time. Uppaal uses `A[]` to denote always.

Hence we produce the following properties that we wish to test against this system of 4 processes.

Starvation-free

```
Process(0).Added --> Process(0).Running

Process(1).Added --> Process(1).Running

Process(2).Added --> Process(2).Running

Process(3).Added --> Process(3).Running
```

Finite run-time per dispatch

```
Process(0).Running --> Process(0).Added

Process(1).Running --> Process(1).Added

Process(2).Running --> Process(2).Added

Process(3).Running --> Process(3).Added
```

Mutual exclusion `A[] forall(i: int[0, N-1]) forall(j: int[0, N-1]) Process(i).Running && Process(j).Running imply i == j` where $N = 4$

We use the two automata and the properties define above as input to Uppaal's verifier and instruct it verify the satisfiability of the persistence properties on the automatons. Uppaal reported that all properties were satisfied[3]. Hence, we can say that BFS is verifiably starvation-free for a system containing the 4 processes of our choice.

A system with realtime processes is not starvation-free because BFS always runs realtime processes first. Since it is impossible to determine whether the realtime processes ever halt or not[4], an arbitrary realtime process may run forever, hence starving the rest of the processes from CPU resource. This is easily verified by changing one of the processes from normal to realtime and

[3]This took 2 hours of computation time on my laptop.

[4]Refer to the halting problem

re-run the verifications, the starvation-free property of the modified system is no longer satisfied.

3 Discussion, Limitations and Improvements

In our formal verification, we made several simplifications to the original BFS system. Some of these simplifications can be justified while the others suggest that our model is subjected to improvements. We shall list the simplifications we made to BFS and discuss how they affect the completeness of the verification.

Reducing the number of realtime runqueues from 100 to 1. Our result showed that as soon as a realtime task is added to the system, the starvation-free property is violated. This result does not change whether we have one realtime runqueue or 100 runqueues.

Reducing the nice value range. We reduced the nice value range from $[-20, 19]$ to $[-1, 1]$. This simplification does not affect the correctness of he verification because we did that simply to reduce implementation burdens. One can easily extend the range of nice value to any finite range to gain even finer granularity.

No multi-core support. We modelled BFS running on a system with only one core because of the time constraint. It is left as a future work to extend this model with multi-core support.

No preemption. A process may be preempted, we did not model this in our work for the same reason as above.

No `SCHED_ISO` and `SCHED_IDLEPRIO`. Recall in the introduction of BFS, we ignored tasks with `SCHED_ISO` and `SCHED_IDLEPRIO` schedulability. Tasks with `SCHED_IDLEPRIO` scheduling will only run if the system has no other task to run. It is inherently vulnerable to starvation because if the system is always busy, the tasks with `SCHED_IDLEPRIO` never get a chance to run. Including `SCHED_ISO` is considered to be a future work.

Did not verify starvation-free in general. We only verified starvation-free on one set of processes. However it is very easy to modify the model to replace our set of example processes with any arbitrary set and re-run the verification on the new set. It is theoretically possible

to verify the starvation-free properties of BFS in general because due the memory footprint constrain, any system must have only a finite number maximum concurrently running processes. In Linux, a system may only have up to 2^{16} processes. Since each process can have nice value between -20 to 19, there can be $O(40^{2^{16}})$ possible sets of processes. However it would take too long time to verify all of the sets and hence not practical.

Did not verify fairness. We could not verify fairness of BFS in our study because it was not expressible with standard Linear Temporal Logic. For example, a system is fair if every process of the same nice level get to run for exactly the same cumulative duration; LTL along can not express this property. Although Timed-LTL can express this property, it is not supported in Uppaal.

4 Conclusion and future work

In this paper, we modelled a simplified version of BFS using Uppaal and conclude that the simplified BFS is verifiably starvation-free for our selection of processes. We note that it is possible to enter arbitrary set of processes to the model and run the verification against the new set. This allows us to extend this model to verify system of arbitrary processes dispatched by BFS. We also verified that as soon as a realtime task is added to the system, the starvation-free property gets violated. BFS is inherently starvation-prone when system has realtime tasks. This is due to the BFS algorithm always running realtime tasks before normal tasks. We made remark that although it is theoretically possible to verify starvation-free property of BFS in general, it is not practical due to the size of system and the running time of each individual verification.

Due to time constrain, we made several simplifications to the original BFS algorithms, namely that we did not consider multi-core behavior and preemption. Extending the model can be considered as a future work for this project. We note that Hansen and Madsen [BHM08] have shown that it is (non-trivially) possible to model multi-core processors in Uppaal. Their work may be helpful for modelling BFS multi-core behavior. Lastly, we were not able to verify fairness, however we hope to be able to obtain some probabilistic guarantee using Statistical Model Checking (SMC) which simulates the system for some finite run and use hypothesis testing to infer

whether the samples provides a statistical evidence for satisfaction or violation. Although SMC can not verify anything in the strict sense, it is worth investigating in order to know how fair BFS can be in a probabilistic way.

References

[BHM08] Aske Brekling, Michael R. Hansen, and Jan Madsen. Models and formal verification of multiprocessor system-on-chips. *The Journal of Logic and Algebraic Programming*, 77(1–2):1 – 19, 2008. The 16th Nordic Workshop on the Prgramming Theory (NWPT 2006).

[gra12] graysky. CPU SCHEDULERS COMPARED, November 2012.

[HHI00] Lone Halkjaer, Karen Haervi, and Anna Ingolfsdottir. Verification of the legOS scheduler using uppaal. *Electronic Notes in Theoretical Computer Science*, 39(3):273 – 292, 2000. {MTCS} 2000 (Satellite Workshop of {CONCUR} 2000).

[KPSA08] Xu Ke, P. Pettersson, K. Sierszecki, and C. Angelov. Verification of COMDES-II systems using UPPAAL with model transformation. In *Embedded and Real-Time Computing Systems and Applications, 2008. RTCSA '08. 14th IEEE International Conference on*, pages 153–160, August 2008.

[PVE+00] J. Penix, W. Visser, E. Engstrom, A. Larson, and N. Weininger. Verification of time partitioning in the DEOS scheduler kernel. In *Software Engineering, 2000. Proceedings of the 2000 International Conference on*, pages 488–497, 2000.