CS2106R Final Report

Analysis of the BFS Scheduler in FreeBSD

By

Vaarnan Drolia

Department of Computer Science

School of Computing

National University of Singapore

2011/12

CS2106R Final Report

Analysis of the BFS Scheduler in FreeBSD

By

Vaarnan Drolia

Department of Computer Science

School of Computing

National University of Singapore

2011/12

Project No: A0075125L
Advisor: Dr. Wei Tsang Ooi
Deliverables:
Report: 1 Volume

Abstract

In this report, we analyse and discuss the implementation of the BFS Scheduler algorithm on the FreeBSD operating system. Other relevant aspects related to operating systems in general are also discussed.

Implementation Software and Hardware:
FreeBSD 8.2, Oracle VM VirtualBox

Acknowledgement

I would like to thank my advisor for the time and effort he put in every week for making me understand the key concepts relating to operating systems and also giving valuable insights with respect to the algorithm.

Additionally, I would like to acknowledge Con Kolivas and Rudo Tomori for work as well as their valuable articles with detailed explanations which made understanding the concepts easier.

Chapter 1

Introduction

1.1 Introduction

BFS is a task scheduler created by Con Kolivas for the Linux Kernel to address issues of desktop interactivity. Rudo Tomori re-implemented the scheduler in the FreeBSD operating system under the mentorship of Ivan Voras as a project for the Google Summer of Code 2011 Program.

In the earlier part of this report we discuss the details of the algorithm as proposed by Con Kolivas and in the latter part we focus primarily on the implementation by Rudo Tomori.

1.2 Background

BFS is a desktop orientated scheduler, with extremely low latencies for excellent interactivity by design rather than "calculated", with rigid fairness, nice priority distribution and extreme scalability within normal load levels (Kolivas, 2012b).

Another goal for the scheduler is to move away from the complex designs of the Completely Fair Scheduler (CFS) used in the Linux Kernel and shift to one with a more simple and basic design such that it can be fine-tuned by the user and achieve deterministic results in contrast to the existing CFS which is very complex to model and predict.

Chapter 2

BFS by Con Kolivas

2.1 Design Summary

2.1.1 Single Runqueue

The design consists of a single run-queue which is a double linked list shared by all the CPU's instead of using multiple run-queues.

Multiple run-queues imply that the CPU's manage their own run queues and achieve fairness and low latency with respect to a limited set of processes on their respective run queues. To get fairness among multiple CPU's, a complex load balancing algorithm is required which is not only difficult to implement, but the advantage of local run queues, namely, to avoid locking of a single run-queue is lost because the problem of having to lock multiple run-queues is created.

Thus, due to the above reasons BFS adopts a single run queue for its implementation.

The global run-queue has at most :

$$(\text{number of tasks requesting CPU time}) - (\text{number of logical CPU's}) + 1, \tag{2.1}$$

where +1 is the previously running task which is put back into the queue during scheduling.

2.1.2 Interactivity

I/O Bound tasks or interactive tasks are decided basis on the fact that the tasks which wake up have not used their quota of CPU time and have earlier effective deadlines making them likely to preempt CPU bound tasks of same priority. Even if they aren't able to preempt such a task, the Round-Robin interval will enable the task to be scheduled in a time frame which will not cause any noticeable loss in responsiveness or interactivity.

2.1.3 Round Robin Interval

This is the only tunable value and it is set to 6ms by default with a range from 1ms to 1000ms. A decrease in the round robin value decreases latency but also decreases throughput while an increase in the value will increase throughput but also worsen (increase) latency.

Thus, the default value is set to 6ms because humans can detect jitter at approximately 7ms so there is no apparent increase in interactivity and aiming for lower latencies might not add any added benefit in most cases.

2.1.4 Task Insertion

The task is inserted as an O(1) insertion to the double link list run-queue. On insertion, every running queue is checked to see if the newly queued task can run on any idle queue, or preempt the lowest running task on the system. This is how the cross-CPU scheduling of BFS achieves significantly lower latency per extra CPU the system has. The lookup is O(n) in the worst case where n is the number of CPUs on the system.

2.1.5 Task Lookup

BFS has 103 priority queues with 100 dedicated to the static priority of realtime tasks, and the remaining 3 are, in order of best to worst priority, SCHED_ISO (isochronous), SCHED_NORMAL, and SCHED_IDLEPRIO (idle priority scheduling).

With each queued task, a bitmap of running priorities is set which shows the priorities with waiting tasks.

Lookup Mechanism

The bitmap is checked to see what static priority tasks are queued for the first 100 priorities and if any realtime priorities are found, the corresponding queue is checked and the first task listed there is taken (taking CPU affinity into consideration) and lookup is complete.

If the priority corresponds to a SCHED_ISO task, they are also taken in FIFO order and behave like round-robin tasks.

If the priority corresponds to either SCHED_NORMAL or SCHED_IDLEPRIO, then the lookup becomes O(n). Every task in the particular priority is checked to determine the one with the earliest deadline and, (with suitable CPU affinity) it is taken off the run queue and assigned to a CPU. However, if any task with an expired deadline is encountered, it is immediately chosen and the lookup is completed.

Thus, the lookup is O(n) in the worst case, where n is the total number of CPUs.

2.1.6 Task Switching

There are two conditions for the current task to be switched :

Time Slice

A task may run out of its allocated time slice and thus, will be descheduled, have its time slice replenished and the virtual deadline reset.

Sleep

A task may choose to sleep and not request any further CPU in which case its time slice and virtual deadline remain the same and the same values will be used again whenever the task is

scheduled to run next.

The next task to be chosen depends upon the property of earliest deadline used for preemption.

Preemption

If a newly waking task has higher priority than a currently running task on any CPU, it will be by virtue of the fact that it has an earlier virtual deadline than the current running task. Once a task is descheduled, it is put back on the queue and an O(n) lookup of all the queued-but-not-running tasks is done to determine which task has the next earliest deadline to be chosen to receive the CPU next.

2.1.7 Virtual Deadline

The key to achieving low latency, scheduling fairness, and "nice level" distribution in BFS is entirely in the virtual deadline mechanism. The one tunable in BFS is the rr_interval, or "round robin interval". This is the maximum time two SCHED_OTHER (or SCHED_NORMAL, the common scheduling policy) tasks of the same nice level will be running for, or looking at it the other way around, the longest duration two tasks of the same nice level will be delayed for.

When a task requests cpu time, it is given a quota (time slice) equal to the rr_interval and a virtual deadline. The deadline is referred to a virtual one because there is no guarantee that a task will be scheduled before or by this deadline but it is used to compute which task should be scheduled next.

Virtual Deadline Computation

The virtual deadline is offset from the current time in jiffies by this equation:

$$\text{jiffies} + (\text{prio\_ratio} * \text{rr\_interval}), \tag{2.2}$$

where prio_ratio increases by 10% for every nice level.

For example, starting from nice level -20 (as in FreeBSD), a process with nice level 10 will have a virtual deadline,

$$\text{Virtual Deadline} = (\text{“now” in jiffies}) + 1.1^{(10-(-20))} * \text{r\_interval} \tag{2.3}$$

$$\text{or, Virtual Deadline} = (\text{“now” in jiffies}) + 1.1^{30} * \text{r\_interval} \tag{2.4}$$

2.1.8 Accounting and Data Protection

BFS has only one lock preventing modification of local data of every task in the global queue and every modification of task data requires this global lock.

However, when a task is passed over to a CPU, so is a copy of it's accounting information such that only that CPU updates it which makes the update of the accounting data lockless (no locking of the global queue) and more efficient. The task is removed from the global run-queue at this time.

2.1.9 Additional Priority Tuning

Isochronous Scheduling

Isochronous scheduling provides near-real-time performance to ordinary users without the ability to starve the machine indefinitely. This is because SCHED_ISO tasks are prioritized between normal and real-time tasks and they run in a Round Robin fashion before normal tasks. However, they are not allowed to run beyond a tunable finite amount of time after which they are demoted to SCHED_NORMAL tasks. This finite amount of time is defined as a percentage of the total CPU available across the machine tunable using the “resource handling” property.

Idleprio Scheduling

Idleprio scheduling is designed to give a task the CPU only when the CPU would otherwise be idle. This allows tasks with extremely low priority to be run in the background such that they have virtually no effect on the tasks in the foreground.

Chapter 3

FreeBSD BFS

3.1 FreeBSD Facts and Differences

3.1.1 Process Scheduling, (Tomori, 2011)

Priority range

Processes in FreeBSD have priorities ranging from 0 (highest) - 255 (lowest)

Scheduling Classes

Processes are classified into scheduling classes as below:

Range	Class	Thread Type
0 - 63	ITHD	Bottom-half kernel (interrupt)
64 - 127	KERN	Top-half kernel (interrupt)
128 - 159	REALTIME	Real-time user
160 - 223	TIMESHARE	Time-sharing user
224 - 255	IDLE	Idle user

Priority Queue

All runnable threads (except for the currently running thread), are placed in a single scheduling run-queue based on process priority. There are 64 priority queues for which the kernel divides the priority value by 4 to give a mapping {0, 255} -> {0, 63}.

The scheduler algorithm does not differentiate among different process priorities in a single priority queue.

Differences between Linux BFS

Due to the differences between the definition of the run queue by the Operating System's Scheduler in Linux and FreeBSD, certain changes were made. The FreeBSD implementation of the scheduler has 64 priority queues in the global run queue while the original BFS uses 103 priority queues.

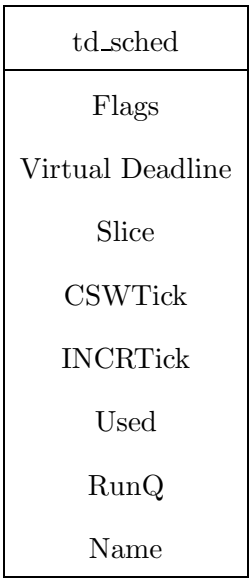
The original BFS differentiates 4 groups of threads. Static priority realtime task are placed into priority queues number 1 - 100. Queue 101 is used for isochronous scheduling, queue 102 is used for normal threads scheduling and queue 103 is used for idle threads scheduling.

On the other hand, the FreeBSD scheduler does not make changes to the Bottom-half and Top-half kernel threads. These are mapped to their respective run queues by dividing the priority number by 4. The last priority queue that is the priority queue number 63 is used for idle thread scheduling. The priority queue just before it, that is the priority queue number 62, is used for time-sharing scheduling. The queues 0 31 are used for Bottom-half and Top-half kernel threads. Priority queues 32 61 are used for real time scheduling threads. So the scheduler uses a separate priority queue for almost every single real-time scheduling thread. Only the last real time priority queue, that is the priority queue number 61, is used for real time threads with priorities 157, 158 and 159. There is no isochronous scheduling in the FreeBSD implementation due to the added complexity of defining a new scheduling class.

3.2 Scheduler Discussion

3.2.1 Schedulable Entity - td_sched

This is the basic schedulable entity for every single task and it adds scheduler specific parameters.



- ⋆ Flags: Additional flags which describe the state of the process
- ⋆ Virtual Deadline : As defined earlier
- ⋆ Slice: Remaining time slice of the thread in number of ticks
- ⋆ CSWTick : Time passed
- ⋆ INCRTick : Ticks incremented
- ⋆ Used : Time used
- ⋆ RunQ: CPU Run Queue the thread is currently on
- ⋆ Name : Thread name for Kernel Tracing Facility

3.2.2 Scheduler Setup - sched_setup()

The initialization of the CPU topology, time slice, priority ratios and run queues is done in the sched_setup function.

First, the current CPU topology is obtained from smp_topo() and all the logical CPUs are checked for presence and added to the scheduler's list of CPUs which is an array cpu_topology.

Next, the actual CPU frequency is obtained and the time slice of the scheduler is set to be one-tenth of that frequency which is generally around 100ms.

Then the prio_ratios array is initialized with the first element having a value of 128 and subsequent priorities getting values which are approximately 10% more than the previous one. The 128 base value is such that the rounding errors on increasing priorities is minimized.

Finally, setup_runqs() is called to setup the run-queues which basically tells the operating system to initialize the run queues.

3.2.3 Add to the schedule - sched_add()

When adding the thread, it is checked whether the thread should be scheduled on the CPU it had last run on using the preempt_lastcpu(td). Otherwise, the idle CPUs are checked in a loop traversing to the logical CPUs parent until it reaches a CPU which can be used.

However, if there is no suitable CPU, the worst running thread is obtained. If the worst running thread has greater priority than the thread being added, the function terminates. Instead, if the worst running thread has lesser priority than the thread being added, a context switch takes place. If both threads have the same priority then the thread is added is according to the one with the earliest deadline first.

3.2.4 Should run on last CPU - preempt_lastcpu()

If the thread did not run on any CPU as yet, the function returns false. Else the physical CPU is checked for having an idle thread in which case an Additional Software Trap, to do the context switch, is sent to the CPU and the function returns true.

Otherwise, the priority of the idle thread is checked and if it is less than that of the thread to be scheduled, then the CPU is sent the AST and set to require rescheduling. The function returns true.

In the final case, the current running thread's priority and virtual deadline is checked and if it is greater or earlier than the current thread, this CPU is skipped and the function return false. There is distinction made here depending on whether the process is an idle one or a time sharing process.

3.2.5 Worst Running Thread - worst_running_thread()

This function loops through all the currently running threads on all CPUs and selects the thread which has the lowest priority among all the running threads. If the thread has an equal priority with a thread, the one with the greater deadline is chosen. This "largest deadline and lowest priority" running thread is then returned as the worst running thread.

3.2.6 Select thread to run - sched_choose()

This function selects the threads to run by using the runq_choose_bfs() function and then setting the thread's flag that it did run after removing it from the run queue.

3.2.7 Choose BFS - runq_choose_bfs()

This function performs the main BFS algorithm for choosing the next thread to execute. All the run queues are iterated and this begins with the real time run queues. If a queue has threads waiting to be scheduled, its bit mask is one and the queues with no waiting processes are ignored.

Next, if the priorities are those of RQ_TIMESHARE or RQ_IDLE namely, normal or CPU idleprio threads, then the earliest deadline first algorithm is applied to get the thread to run.

Otherwise, the first thread in the run queue by FIFO is chosen for the other priority levels. The priority of each queue is calculated by multiplying the run queue status word by size of a run queue status word raised to the i_{th} power of two where i is the level of the priority.

3.2.8 Earliest Deadline First - edf_choose()

This function gets the head of the run queue on which it has to apply the earliest deadline first algorithm. The function itself is very straightforward as it enumerates through the linked list of the run queue and looks for threads which can be scheduled and have a deadline earlier than the minimum deadline it has obtained till then.

Finally, it returns the schedulable thread with the earliest deadline in the run queue.

One thing to remark is the deviation from the original BFS algorithm where whenever an expired deadline is encountered, that is automatically chosen. Here, the one with the earliest deadline is always chosen. Thus, the original is O(n) in the worst case while this implementation is always O(n).

3.2.9 Priority Changing - sched_prio()

First the thread of the base priority is updated. If the thread is borrowing another thread's priority which is less than its own priority, we don't lower the thread's priority and set the thread as borrowing.

Otherwise, the priority of the thread is updated with the new priority. If the thread is on a turnstile (sleeping state in a queue) then the turnstile is made to update itself.

3.2.10 Clock - sched_clock()

This is where the thread's time slice is updated and it is incrementally decreased for every clock cycle used until it reaches 0. After that, the virtual deadline is recalculated by the formula for the virtual deadline,

$$\text{Virtual Deadline} = \text{ticks} + \text{time\_slice} * \text{prio\_ratio}/128, \tag{3.1}$$

which is similar to the one stated in the original algorithm with the exception of the 128 term. This is the initial priority assigned to the first priority and it is a scaling factor introduced to decrease rounding errors.

Next, the time slice of the thread is replenished and the thread is marked as needing to be rescheduled.

3.2.11 Fork Thread - sched_fork_thread()

The forking of a new thread from a parent involves getting the scheduler's lock. After that the CPU set of the child is set to the same one as the parent. The time slice of the child is halved and its virtual deadline and used ticks are set the same as its parent.

3.2.12 CPU Affinity - sched_affinity()

The TSF_AFFINITY flag is set if there is at least one CPU this thread can't run on. This is done by removing the TSF_AFFINITY flag from the thread and then looping over all the CPUs, and if a CPU is present and the thread cannot be scheduled on it, the TSF_AFFINITY is set for the thread.

However, if the thread can run on all CPUs, then the function returns without modifying the TSF_AFFINITY.

Pinned or bound threads to one CPU should be "left alone" according to the FreeBSD documentation.

Finally, the state of the thread is checked and if it is running, it is checked if the current CPU is one on which the thread can be scheduled. If not, a context switch is forced on the CPU using AST.

3.2.13 Switch threads - sched_switch()

This whole function takes place in the critical region and it does so by acquiring several locks and mutexes. Firstly, if the thread is still marked as running, then it is put back on the run queue as it has not been suspended or stopped. This is because idle threads are never put on the run queue.

If the new thread is not NULL, the thread is run as if it had been added to the run queue and selected. It came from a preemption, upcall or a followon. After switching to the thread and sleeping on the CPU, any amount of time may have passed and the CPU may have changed.

Thus, the time passed and time used are updated taking the difference from the current ticks and the value the thread had before it went to sleep.

3.2.14 Remove thread from run queue - sched_rem()

This function removes a thread from a run queue. First it checks whether the thread is in memory or whether it has been swapped out.

Next, it is asserted that the thread is on the run queue. Finally, the scheduler lock is acquired and if the thread has a load flag, the thread count is increased by one and the thread is removed from the run queue. The thread is then set to the runnable state.

3.2.15 Priority Lending - sched_lend/unlend_prio()

When a thread is lent another thread's priority, its flag is set to TDF_BORROWING and the function for changing the priority is invoked.

For unlending another thread's priority, it restores a thread's priority when priority propagation is over. The priority argument passed to the function (prio) is the minimum priority the thread needs to have to satisfy other possible priority lending requests. If the thread's regularly priority is less important than prio the thread will keep a priority boost of prio.

Chapter 4

Additional Notes

4.1 Priority Inversion and Lending

4.1.1 Priority Inversion

A high priority process has to wait for a lower priority process which is in kernel space (non-preemptive kernel).

4.1.2 Priority Inheritance or Priority Lending

Solves the priority inversion problem. When a higher priority thread needs resources used by a lower priority thread - the higher priority thread lends its priority to the lower priority thread.

4.2 Scalability

Due to a single run-queue shared by all the CPUs, BFS is not very scalable and will show significant performance drops after the number of logical CPUs exceeds approximately 16. This is because multiple run-queues for different CPUs will have lower lock contention as the number of CPUs increases.

There are some scalability features in BFS such as local copies of task data for different CPUs to make it lockless and CPU affinity towards the last CPU the task was run on to take advantage

of caching. Additionally, there is the concept of a sticky flag to bias heavily against tasks being scheduled on a different CPU unless the CPU would be idle otherwise.

4.3 SMP

Symmetric multiprocessing (SMP) involves a multiprocessor computer hardware architecture where two or more identical processors are connected to a single shared main memory and are controlled by a single OS instance. Most common multiprocessor systems today use an SMP architecture. In the case of multi-core processors, the SMP architecture applies to the cores, treating them as separate processors.

4.4 NUMA Architecture

Non-Uniform Memory Access (NUMA) is a computer memory design used in Multiprocessing, where the memory access time depends on the memory location relative to a processor. Under NUMA, a processor can access its own local memory faster than non-local memory, that is, memory local to another processor or memory shared between processors. It logically follows in scaling from symmetric multiprocessing (SMP) architectures.

4.5 Miscellaneous

I was required to compile and build the kernel to patch it with the FreeBSD kernel. This was quite simple since there were no problems with dependencies and I was running as the "root" user.

Also, I learned about CPU topologies and how the logical CPUs represented for the operating system will differ from their physical representation as multiple cores and hyper-threading are used.

References

Kolivas, C. (2012a). Bfs scheduler. http://ck.kolivas.org/patches/bfs/sched-BFS.txt.

Kolivas, C. (2012b). Faqs about bfs. v0.330. http://ck.kolivas.org/patches/bfs/bfs-faq.txt.

Tomori, R. (2011). Bfs freebsd. rudot.blog.com.

Wikipedia (2012). Several articles. http://en.wikipedia.org/.