Design Document for Assignment 2 - Lottery Scheduling in FreeBSD

Marian Vladoi
Kenneth Mayer

Contents

1 Introduction

For this assignment, we were to implement lottery scheduling in FreeBSD for non-root user processes.

1.1 Scheduling

Scheduling is an operating system concept that can be of great importance for multiprogramming environments. A scheduler is an algorithm that chooses which currently active processes should be allowed to run at any given moment. A scheduler aims to give a reasonable amount of CPU time to each process, complete high importance processes quickly and make sure that CPU intensive jobs can complete eventually. Ideally, you want a good balance of response time (to realtime jobs) and throughput (how many jobs are completed per unit time).

There are many different types of scheduling schemes that can be used, some of them work well in different types of systems. For example, a scheduling algorithm like **shortest job first**, where the shortest job is selected might be appropriate for a batch system (assuming it can be implemented at all), but in a realtime or timeshare system, might result in particularly long jobs being starved for resources.

For **interactive systems**, like the personal computers that an operating system like FreeBSD might be used on, the scheduler must be **preemptive**, which means that a process may be suspended involuntarily, even if they don't block for I/O or resources. In FreeBSD, and in many other interactive systems, a process is given a maximum period of time to run, called a **quantum**. If a process is allowed to run indefinitely, CPU intensive jobs may run for a very long time, and prevent the user from interacting with the computer.

FreeBSD uses a combination of priority scheduling and round robin. There are three **run queues** in FreeBSD, each of these run queues has 64 run queue heads, which are also queues, and are where the processes are actually placed. There is one run queue for realtime (highly interactive threads), one for time share threads (longer jobs that may run in the background) and one for idle threads. When deciding which program to run, FreeBSD looks at the highest priority, non-empty queue head, and then runs the first process in that queue, then goes down the list using a round robin scheme.

1.2 Lottery Scheduling

We are going to add a new scheduling scheme on top of this one, called **lottery scheduling**. In lottery scheduling, each thread on a run queue is assigned a certain number of tickets. Every time a thread is chosen to run, a random number (between 0 and the total number of tickets in the queue) is chosen. Sum the ticket count of the processes in the queue starting at the head, until we get a value higher than the random number. Whichever thread's ticket count was added last is chosen to run, still for the same fixed quantum.

In this case, priority can be adjusted by adjusting the number of tickets. The more tickets a thread has, the more likely it is to run. This algorithm is suitable for use with interactive systems because it uses preemption.

2 Implementation

In this section, I will talk about what changes we made to the FreeBSD kernel to add lottery scheduling. We modified the following files

```
/usr/src/sys/sys/proc.h
/usr/src/sys/sys/runq.h
/usr/src/sys/kern/kern_thread.c
/usr/src/sys/kern/kern_switch.c
```

```
/usr/src/sys/kern/sched_ule.c
```

Most (hopefully all) modified code is marked with the named Vladoi Marian.

2.1 proc.h

We added a few new useful macros to this header file

```
#define DEF_TIK 500     // number of tickets a thread is assigned by default
#define MIN_TIK 1       // minimum number of tickets a thread may have
#define MAX_TIK 100000  // maximum number of tickets a thread may have
$define EMPTY 0         // make sure threads are never empty
```

We also modified the `struct thread` definition to add the member

```
int number_tik // the number of tickets belonging to this thread
```

Which is fairly self-explanatory. Even root threads and kernel threads will have this member, though it will not be used. We discovered that this must be placed at the end of the `struct thread` definition because the kernel will assert that the members of the struct are properly aligned.

Finally, we added a few new function prototypes that we use in the file sched_ule.c

```
void td_give_tickets(struct thread *td, int score);
void td_remove_tickets(struct thread *td, int score);
```

These functions accept the thread that should have its tickets reduced or increased. score refers to an interactivity score assigned by FreeBSD's `sched_priority()` function. Interestingly, the value of score will influence which function is called, so it might have been better to combine these into one function.

2.2 runq.h

In this thread, we added one macro at the top

```
#define POLL_SIZE 5 // the size of the pool of random numbers we generate at boot
```

We also modified the `struct runq` to add the members

```
int random_winner;          // the random number we selected from the pool (mod total_tik)
int total_tik;              // total number of tickets in a queue
int poll[POLL_SIZE];        // pool of random numbers associated with this queue
int count;                  // the position we're looking at in the pool
struct rqhead rq_user_queue // the queue for user processes
```

We also declared a function prototype for choosing threads that use lottery scheduling.

```
runq_lott_choose(struct runq *);
```

`runq_lott_choose()` accepts a run queue as input and returns a thread from the rqhead containing user threads within it. It should be called under the same circumstances as FreeBSD's `runq_choose()`, but should only be called when one wants to schedule a non-root user program.

2.3 kern_thread.c

We only modified one function in this file, name `thread_init`, where we initialize the number of tickets it has to the default, 500.

```
static int
thread_init(void *mem, int size, int flags)
{
        struct thread *td;

        td = (struct thread *)mem;
        td->td_sleepqueue = sleepq_alloc();
        td->td_turnstile = turnstile_alloc();
        td->td_rlqe = NULL;
        EVENTHANDLER_INVOKE(thread_init, td);
        umtx_thread_init(td);
        td->td_kstack = 0;
        td->td_sel = NULL;
        td->number_tik = DEF_TIK; // new addition: initialize the thread's tickets
        //printf("Initializing thread.\n");
        return (0);
}
```

This addition means that every thread will at least have the 500 tickets associated with them, even if they aren't actually threads that will be used with our lottery scheduling. My feeling is that this has no effect on threads that don't use lottery scheduling, and checking whether they should have tickets assigned to them will take longer.

2.4 kern_switch.c

This file contains a lot of functions that operate on threads that are currently in the run queue, including choosing which one should be removed, adding a thread to the queue and removing a thread from the queue. We modified a few of these to accomodate our threads, and we provided the definition of `runq_lott_choose()`.

2.4.1 runq_add()

In the functions `runq_add()` and `runq_add_pri()`, we first check the thread's credentials and look at the user ID of its owner, by accessing the member `td->td_ucred->cr_uid`. If this value is nonzero, it is a user thread, and thus should be added to our new user rqhead. We assign the rqhead to be added to as our user rqhead and add the ticket count of the new thread to `runq->total_tik`. If the thread is a root thread, we let the original routine continue.

2.4.2 runq_lott_choose()

Originally, we planned to modify the functions `runq_choose()` and `runq_choose_from()`, but eventually we decided to define our own function and call that from other functions directly.

Our new function, `runq_lott_choose()` is probably the most important addition that we made to this file. This function has a runq passed to it. We first look at this runq's random number pool at index `rq->count`, which we then increment. We assign this chosen value (mod `rq->total_tik`) to `rq->random_winner`. Then we look at the first thread in the user queue and iterate through it until the sum of tickets is greater than or equal to `rq->random_winner`. We then return a pointer to the chosen thread.

2.4.3 runq_remove_idx()

Finally, we adjusted `runq_remove_idx()`. We once again check the thread's credentials to see if it is root or non-root. If it is, we remove the thread from the user queue and subtract from the total ticket count, the number of tickets the removed thread held.

2.5 sched_ule.c

This file is the heart of the scheduler, and contains thousands of lines of code. Thanfully, we didn't have to make too many modifications to this one.

2.5.1 tdq_choose()

This function is in charge of calling the `runq_choose()` functions. One problem posed by this function is the question of where we should call `runq_lott_choose()`. We want to prioritize root threads and kernel threads (except idle threads) over our user threads, so we let this function check the default runqs, using the priority algorithm first, then if no threads are returned, we call `runq_lott_choose()`, first on the realtime queue, then on the timeshare queue, then on the root/kernel idle queue and finally on the user idle queue.

2.5.2 sched_priority()

This function is in charge of changing the priority of running processes. We use it to change the number of tickets. We, however, don't use the priority, we use the interactivity score, which seems to range from 0 (highest priority) to 50 (highest priority). Thus, if the score is greater than `(50 − sched_interact)/2` we subtract tickets and if it is greater, we add tickets. Realtime threads are always given tickets. `sched_interact` is an integer that represents the cutoff for an interactive thread.

2.5.3 sched_nice()

This function is linked with the `nice()` system call. In this function, we check if the thread that's calling it is a user thread. If it is, we add (linear) or subtract tickets (exponential) based on its nice value. Else, we use the original algorithm.

2.5.4 td_give_tickets()

This is a newly defined function. First, this function defines a maximum number of tickets the thread should have based on its current score. If its score is 0 (most interactive), it can have up to 100,000 and if its score is 0 (least interactive), it can have up to one.

We then add tickets based (50-score) * 5, so that not too many tickets are added once.

2.5.5 td_give_tickets()

This is a newly defined function. First, this function defines a minimum number of tickets the thread should have based on its current score. If its score is 50 (most interactive), it can have up to one.

We then subtract tickets based on score * 5, so not too many tickets are subtracted at once.

2.5.6 Reward and Punishment Algorithms

I thought it was best to have linear rewards and punishments; I felt that, given the high upper bound on the number of tickets, subtracting and adding relatively few tickets to each thread would help to keep the thread priority balanced, while using exponential punishment might unduly punish a timeshare thread that happens to fall below a certain threshold. If a thread is truly CPU intensive, it will fairly quickly be pulled down its minimum number of tickets anyway.

You may recall that the nice function punishes threads exponentially. I felt that since nice calls were fairly infrequent, it would be best to ensure that a fairly large reduction is given every time.

2.5.7 sched_clock()

This function runs a few times every second, each "clock tick", and we use it to refill our random number pool with five new random numbers.

3 Conclusion

This assignment was a useful introduction to modifying the kernel and provided a useful look at the topic of scheduling and scheduling in FreeBSD specifically.

Our implementation of lottery scheduling at least works fairly well. During testing, we found that obnoxious programs are indeed punished with ticket reduction and highly interactive tasks (like accepting keyboard input) very quickly gain a lot of tickets, which is at least close to the expected result. However, I am not fully confident that the implementation is good as it possibly could be. I don't know any method for measuring the scheduling performance, and I am not entirely sure how to run tests where lots of interactive threads are run at once.

A few things I think could be done to improve the implementation still exist. I'd like to find a way to generate the random number pool (a larger one, with 1000 indices or so) at boot time, rather than generating five new ones every clock tick. This can be done with a global variable, but our attempts doing it during `runq_init()` was unsuccessful. Also, I think that the ticket adjusting algorithms that are called during `sched_priority()` could be improved.