# Assignment 2: Lottery Scheduling

Daniel Urrutia (deurruti@ucsc.edu)
Justin Wong (jujwong@ucsc.edu)
Joshua Mora (jommora@ucsc.edu)
git pull git@git2.soe.ucsc.edu:classes/cmps111/winter16/deurruti assignment_2_dejoju
git push git@git2.soe.ucsc.edu:classes/cmps111/winter16/deurruti assignment_2_dejoju
CMPS 111, Winter 2016

---

## Goals

Modify the FreeBSD scheduler to use lottery scheduling for user processes and maintain round-robin scheduling for system processes.

## Relevant Functions

**Header files**

**runq.h** is a header file that specifies the number of run queues RQ_NQS. We changed this from 64 to 65. Within this file, we will add two fields to the runq struct. One field will designate the ticket count for a thread (*int ticket_count*) and another field (*int rq_type*) which will be used to determine whether the thread will belong in the user threads runq index (index 64) or the original runq's (indices 0 to 63).

**proc.h** contains the structure definition for a thread and a field, *int tickets*, was added to keep track of the number of tickets that a thread has.

**Phase 0: Initialization Process**

Within **sched_ule.c**, the function *sched_setup()* determines if thread queues need to be initialized for a multiprocessor or single processor system. Either way, *tdq_setup()* gets called in order to initialize each of the three run queues by calling *runq_init()* which exists within **kern_switch.c**. The system runs through the array that represents each run queue and initializes the "head" of each doubly-linked list of future threads. We don't plan to add modifications to this process.

**Phase I: Adding a thread (default)**

First source file **sched_ule.c** contains **struct td_sched** which has fields for the current run queue and information about the quantum for a thread. struct tdq contains initialization of the three run queues.

*void sched_add()* selects a thread queue and sets a lock on the thread queue before calling *tdq_add()*. The lock is important to avoid synchronization issues regarding multiple modifications to the thread queue "at once."

*tdq_add()* asserts that a lock has been acquired and associated error checking before proceeding. Once this has finished, it compares the thread's priority to tdq's lowest priority thread. Sets the lowest priority of the tdq equal to the priority of the thread that got passed in if the priority of the thread that got passed in is lower than the lowest priority of the tdq.

*tdq_add()* then calls *tdq_runq_add()* which asserts the lock. Gets the thread's priority, gets the thread's pointer to the member *td_sched*. Then checks if the thread can migrate to another CPU (another thread

queue). The variable ts gets the thread's current td_sched. From there, it will determine whether it goes in the realtime runq, the timeshare runq, or idle runq.

Once the appropriate runq has been selected, the function then calls *runq_add()* to actually add the thread to the runq. *runq_add()* **does not** exist inside of **sched_ule.c**. It exists inside of **kern_switch.c**.

In the function *runq_add()*, a pointer to the runq head is declared and the priority of the thread to be added (thread passed in) gets divided by the macro *RUNQ_PPQ* (The priorities per queue [4]). Once that is calculated, the thread's field *td_rqindex* is equal to the calculated priority. *runq_setbit()* is called to let the system know that that specific runq is not empty. Once that it is done, the thread is inserted to either the tail of the queue (which is the "head") or the actual tail of the queue (the "end").

**Phase II: Choosing a thread (default)**

Within **sched_ule.c**, the function *sched_choose()* asserts a lock on a thread queue, which then calls the function *tdq_choose()*.

*tdq_choose()* calls the function *runq_choose()* on either a realtime runq, the timeshare runq, or idle runq depending on the priority of the thread.

Within **kern_switch.c**, we have the function *runq_choose()*. This function iterates over the entire runq array and currently selects the first nonempty doubly-linked list of threads and returns the tail (head).

Within **sched_ule.c**, the function *sched_priority()* determines which runq to assign a process to.

Within **sched_ule.c**, there are various functions that determine how long a thread has run. Another function determines if a thread has used up it's allotted time slice, the flags **TDF_NEEDSRESCHED** and **TDF_SLICEEND** are set for the struct field **td_flags**. This will initiate the beginning of the (re)scheduling process.

**Phase III: Rescheduling a thread**

When returning from the CPU, the scheduler determines the thread's interactivity score

Set "nice" value for a whole process by calling the function *donice()* inside of **kern_resource.c**. The default method currently takes in some value $n$ and either sets $n$ to either the macros **PRIO_MAX** or **PRIO_MIN**. From here, the function *sched_nice()* is called which is located inside of **sched_ule.c**. The function *sched_nice()* takes in the process and the nice value calculated in the function *donice()* and sets the process' nice field equal to the nice value that was passed in from *donice()*. For each thread in the process, the function then calculates their respective priorities. This can determine how a thread is scheduled in the (near) future.

To calculate their priorities, the function *sched_nice()* calls *sched_priority()* which scales the priority according to the interactivity of the process. This calculation only applies to system processes. We will be introducing our own code for calculating the priorities of user processes.

# Design

**Phase I: Adding a thread (modified)**

Unless problems arise relating to switching a thread from one CPU to another CPU, we will assume that it is currently safe to **initialize a thread's tickets to 500 inside of *tdq_add()***.

**UPDATE:** We realized that this was an incorrect place to initialize thread tickets to 500 after finding the function *thread_init()* function inside of **kern_thread.c**.

We will need to add a check within *runq_add()* so that we will know which process a thread is associated with (either a root process or a user process).

td_process = td–>td_proc;
proc_ucred = td_process–>p_ucred;
eff_uid = proc_ucred–>cr_uid;

This code will obtain the user ID of the thread we are currently working with. In the code, if the user ID > 0, then we know that we have a user thread. This means that we will be placing the thread in one of the three user run queues, depending on its priority. If it is a user process, we want to make sure that it gets added to the appropriate runq (the 65th [index 64]) place of the runq it's being placed in.

**Phase II: Scheduling a thread (modified)**
Inside of *runq_choose()*, we checked if the run queue that was passed in was a user run queue. This was done by adding a field to the runq struct named *rq_type*. If *rq_type* is 1, then we know we have a user runq. Otherwise, it is a system runq and we let the system handle that.

After determining that we are working with a user runq, we call a function we wrote called *lottery_choose()*.

$i \leftarrow arc4random()$
$random\_num \leftarrow i \quad \% \quad ticket\_count - 1$
**while** the runq is not empty **do**
  **for** each thread in runq **do**
    td_tickets = thread–>tickets
    running_ticket_sum += td_tickets
    **if** running_ticket_sum > random_num **then**
      return thread
    **end if**
  **end for**
**end while**

The above will choose the next user thread to run, return it and that is the chosen thread to be run next.

**Phase III: Reallocating Tickets (modified)**

While a context switch is happening in the function *sched_switch()* inside of **sched_ule.c**, we have one of two conditions at any given ticket reallocation:

The thread being chosen for the next run is the thread that was already running

or

The thread being chosen for the next run is different from the thread that just ran

In any case:

We look at the thread struct's time slice field to see the remaining time left. If it used all of its quantum in the previous run, we punish the thread by subtracting a ticket. We attempted to use a scaling factor to

punish a thread (dividing by, say, 1.3), but we ran into boot loops. We are unsure as to why this problem was occurring, because we have guards to check if the thread's ticket count is less than 0 or greater than 100000. If the thread's ticket count ended up being less than 0 after scaling down, we set the ticket count to 1.

If it is the case that the thread did not use all of it's allotted time in the previous run, we reward the thread by giving it two tickets. Again, we ran into issues with scaling despite the fact that we had ticket count guards.

To modify the "nice" system call, we added code inside of the function *sched_nice()*. This code checks the nice value that was passed in:

```
if nice > 0 then
// decrease priority
    td–>tickets -= 10 * nice
    if td–>tickets < 0 then td–>tickets = 1
    end if
else
// increase priority
    td–>tickets += (10 * -1 * nice)
    if td–>tickets > 100000 then td–>tickets = 100000
    end if
end if
```

We apply the ticket allocation change to every thread (even system threads), but we know that only user threads will actually be affected by ticket count modifications, because the system threads are handled by the default scheduler (which does not make use of tickets in any way). The operating system handles error checking for a nice system call. For example, if a non root user attempts to increase the priority of a thread, they will be given a "Permission denied" error by the system. This of course means that we do not need to check if a non root user is attempting to increase a thread's priority inside of *donice()*.

# Testing

Our main method of testing was building and running our modified kernel. We attempted file system navigation, opening user programs like vi. We wrote a couple of simple programs to consume CPU time. The main test program we used was a program written by Ethan Miller called **longrun.c**. This program was found at *https://classes.soe.ucsc.edu/cmps111/Spring15/content/longrun.c*. As shown in the program's source code comments, "this program will run for a very long time and will print out messages to identify itself."