Assignment 2 Design Doc – Lottery Scheduling
Team Members: Prateek Chawla, Adel Danandeh, Tim Mertogul (Team Captain)

Project Overview:

Assignment 2 incorporates modifying the FreeBSD scheduler to utilize lottery scheduling for user processes. The unmodified FreeBSD scheduler has 64 run queues. Some of these queues handle root processes, while others are for user processes. Notably, root processes have a userID of zero and non-root processes have non-zero userIDs. We created three queues for non-root processes; one for each of the following:

Interactive processes
Timeshare processes
Idle processes.

Priority for checking the queues will decrease as we descend down the queues, with interactive having the highest priority and idle having the least. Importantly, root processes will still be placed in the original 64 run queues.

Lottery Scheduler Overview:

The idea of lottery scheduling is that we randomly assign each process *x* tickets then randomly select a ticket and the process with the ticket selected is allowed to run for set time quantum. The more tickets a process has the higher the likelihood of it getting chosen to run is. In the our lottery scheduling implementation we assign 500 tickets to each process. Each thread is only allowed to have tickets in a range 1 to 100,000 tickets, which 1 and 100,000 are being minimum and maximum number of tickets respectively. Our scheduler will randomly pick tickets ranging from 0 to the total number of tickets in the particular queue it is reading (i.e. interactive, timeshare, and idle queues). Thusly, we have to account/track for number of processes added/removed from our three queues. We reward processes which are idle for a long period of time that implies they are IO intensive by multiplying the number of tickets they hold by a constant number (by a factor of 2). And also punish CPU intensive processes by a factor of 2 . These modifications are based upon on process priority. Importantly, we are not calculating number of tickets during context switches in order to maintain the expected efficiency by the kernel. Additionally, the nice() system call can be utilized to modify the number of tickets for a process.

Project Specifics:

We start off by adding and initializing new user queues as per assignment specifications. Our initialization implements the new default values for our lottery scheduler. We implement our lottery scheduler by diverting user processes to our new user queues, away from the standard 64 used by the default scheduler in FreeBSD. Each user thread created is initialized with 500 “tickets” and placed into our lottery scheduler queues. Each time a queue adds or removes a thread, it updates its specific “winning ticket” number, by using a random number generating process (explained in more detail in

Modified Files and Methods). When the processor calls the run queue to choose the highest priority thread to run, the lottery scheduler returns a user thread chosen by the lottery scheduler algorithm. That is to say, the scheduler takes a random number, and performs the modulo operator, yielding *val*. It goes down the list of processes in the queue, adding the number of tickets each one has, until it has reached a number larger than *val*, returns that thread as that's the process to run. The thread is then removed from the run queue, and the run queue is updated to reflect the loss of the thread. After the thread has run, it is either penalized for being CPU intensive by decreasing the number of tickets or it is rewarded by increasing the number of tickets if it is I/O intensive and the CPU has a lot of idle time with it.

Files Modified:

- sched_ule.c
- kern_switch.c
- kern_resource.c
- runq.h
- kern_thread.h
- proc.h

Modified Files and Methods:

----------sys/kern/sched_ule.c----------

tdq structure (struct tdq)

The three required queues for lotto scheduling were added to the tdq structure. These three queues were added to the tdq structure as this is where the, original unmodified, root queues are located.

```
declare user interactive queue     //i.e. struct runq tdq_lottery_interactive
declare user timeshare queue
declare user idle lottery queue
```

tdq_choose()

In the *tdq_choose()* method, we first check our three newly created queues in order of priority and returned a non-null *thread* from the *runq* with the highest priority. Importantly, this order of priority matches the design specifications in the assignment, where we can initially check *tdq_lottery_interactive*, followed by *tdq_lottery_timeshare* and finally *tdq_lottery_idle*. Each of these checks consist of calling our *runq_choose_lotto()* method. Wherein if *runq_choose_lotto()* returns NULL, the next queue is checked, but if it returns a thread, that thread is returned. We selected *tdq_choose()* to do the following, because this is where threads are selected to run.

```
Check user interactive queue and assign return to td.
if td value is not NULL
        return td.
end if
Check user timeshare queue and assign return to td.
if td value is not NULL
        return td.
end if
Check user idle lottery queue and assign return to td.
if td value is not NULL
        return td.
end if
```

tdq_setup()

tdq_setup() initializes the three user run queues by calling *runq_init()*. Specifically, this initialization occurs after the root queues are initialized. *tdq_setup()* was primarily selected to do perform this initialization to mirror the root queue functionality.

```
initialize user interactive queue.
initialize user timeshare queue
initialize user idle lottery queue
```

sched_nice()

After bound checking is completed in the donice() method, the number of tickets assigned to a process is correspondingly increased or decreased by the given nice value. This modification is only for user processes, as the default mechanism handles root processes. Essentially, by modifying donice() and sched_nice() the nice() system call has been converted to adjust tickets assigned to user processes. Notice, we treat every process as though it only has one thread as per a conversation with Professor Long in office hours.

```
if thread is a root process:
        Default behavior.
  else thread is a user process:
        for each thread in process
                Adjust number of tickets by nice value.
        end for
  end if
```

sched_switch()

Here we implemented our core punish/reward mechanism. Our scheduler rewards I/O intensive processes by multiplying their number of tickets by a factor of 2. We reward the processes that leave the CPU idle for long periods of time because that I/O is much slower than CPU.

Our scheduler punishes the CPU intensive processes by dividing their number by a factor of 2. We punish the processes that are CPU intensive. Our methodology for implementing punishing and rewarding in sched_switch because the comment block from the writers of FreeBSD said that sched_switch is the function that handles threads coming in while blocked for some reason, running, or idle.

```
if the thread is idle
        if thread is a user process
                Multiply threads number of tickets by factor of 2
        end if
else if the thread is running
        if thread is a user process
                Divide threads number of tickets by factor of 2.
        end if
end if
```

----------sys/kern/kern_switch.c----------

runq_init()

We added initialization for our new fields in each individual runq to their default values.

rand_lotto_winner is the current winning ticket number for a specific runq.

total_lotto_tix associates to the total number of tickets in a specific runq.

We chose to initialize the runq fields *rand_lotto_winner* and *total_lotto_tix* to 0 here because each new *runq* starts out without any threads in it, therefore its *total_lotto_tix* is equal to 0, and the runq hasn't chosen a *rand_lotto_winner*, because winning tickets are chosen at random, when a thread is added or removed from the runq.

```
initialize lotto winner to zero.
initialize total lotto tickets to zero
runq_init:
   rand_lotto_winner = 0
   total_lotto_tix = 0
```

runq_add()

Here the user and root processes are separated based on their given userids, where zero specifies a root process, and non-zero is non-root. Our changes only affect non-root processes, as per the design specifications of the the project, therefore all the default mechanisms handles root processes. Specifically, this is done through checking if the process is root or non-root. If non-root, the given bounds for number of tickets is checked. Specifically, if the number of tickets for a thread is larger than the *MAX_LOTTO_TICKETS* constant it is reset back to that value. Additionally, if the number of tickets is smaller than the *MIN_LOTTO_TICKETS* the ticket value is reset to that minimum. Lastly, if the number of tickets is equivalent to the *EMPTY_LOTTO_QUEUE* constant the value is reset to the *DEFAULT_LOTTO_TICKETS* constant. The reasoning for resetting these values to their inclined bounded bases rather than throwing an error is due to an effort to maintain constancy (and user friendliness). There is no added benefit to erroring out, as it ruins the black box form of abstraction the user is presented with when using an OS. The new tickets are added to the their specific run queues ticket total. This is done so the three new run queues can keep track of their maximum value (in order to calculate the random number and check bounds). Lastly, the random number is generated.

The current value correspondence for the constants listed above is:
MAX_LOTTO_TICKETS = 100,000
MIN_LOTTO_TICKETS = 1
EMPTY_LOTTO_QUEUE = 0
DEFAULT_LOTTO_TICKETS = 500
Notably, these values were provided in the project guidelines.

```
if thread is a root process
        Default behavior.
else thread is a user process
        if thread tickets is greater than 100,000
                Set number of tickets to 100,000.
        else if thread tickets is less than 1
                Set number of tickets to 1.
        else if thread tickets is equal to 0
                Set number of tickets to 500.
        end if
        Increment number of tickets in run queue by number of tickets added.
        if the total lotto tickets in a queue is greater than 0
                Calculate random number.
        end if
end if
```

##Random_Number_Generation##

The number of lotto tickets in a specific run queue has to be greater than zero for a random number to be generated. This random number is generated for every thread added and removed from a run queue. This is done to ensure the random number is not generated upon context switches, which would be inefficient. Essentially, the value is tracked as processes are added and removed from queues. Notice, the *random()* function in *FreeBSD* provides a 32 bit number so the function was called twice and cast to a 64 bit *long long*. Afterwards, the modulo operator is utilized to compute the range (modulo *total_lotto_tix*).

```
rq->rand_lotto_winner = (long long)(random() + random());
rq->rand_lotto_winner = rq->rand_lotto_winner % rq->total_lotto_tix;
```

runq_choose_lotto()

Similarly, to the built in function *runq_choose()* the thread selected to run is chosen here. Critically, this is where the lotto scheduler selects the "winning" thread. Firstly, we check if the run queue has greater than zero tickets. If the number of tickets is less than zero no thread is returned as the queue is empty (NULL return value). Otherwise, the list of threads is iterated through, where the number of tickets is summed until a number larger than the corresponding random number is found. This thread is the one selected to run. It is returned. This methodology gives a statistical guarantee that the thread with most tickets is the most likely to run, but does not guarantee it as any number is equally likely as the next (it's random). A larger number of tickets only guarantees a larger interval or sum accumulated, therefore increasing the likelihood of selection.

```
Initialize a counter to zero.
if run queue is not empty
        for each thread in run queue
                Increment counter by number of tickets in currently selected thread.
                if counter is greater than the randomly selected lottery winner
                        Return thread.
                end if
        end for
end if
```

runq_remove_idx()

This method was modified to keep track of thread tickets and run queue tickets. Once, a thread is removed from the run queue the number of total number of lottery tickets that queue maintains must be decremented/maintained. Importantly, this check only occurs for non-root processes as the default mechanism handles root processes. Additionally, random number is regenerated upon removing a thread

from the run queue. This generation occurs in an effort to preserve kernel efficiency. See *##Random_Number_Generation##* for a full explanation of random number generation and why it occurs in this particular manner/location.

```
if thread is a user process
    Decrement number of tickets in run queue by number of tickets removed.
    if run queue is not empty
        Generate random number.
    end if
end if
```

----------sys/kern/kern_resource.c----------

donice()

The handling of nice system call is separated for root and user processes. This is primarily done because the nice value for root processes is between PRIO_MAX and PRIO_MIN and tickets can be assigned a value between MIN_LOTTO_TICKETS (1) and MAX_LOTTO_TICKETS (100,000). The two nice values are not interchangeable as the bounds differ greatly. The functionality for root processes is unchanged. For user processes the maximum and minimum bounds are checked for, where if the bounds are exceeded the nice value is reset to the respective bound exceeded. The values are reset rather than producing an error to maintain the underlying consistency and fluidity of the lottery scheduler. This value is passed to sched_nice().

```
if thread is a root process
    Default behavior.
else thread is a user process
    Increment nice value by number of tickets.
    if new nice value is greater than 100,000
        Assign nice to 100,000.
    else if new nice value is less than 1
        Assign nice to 1.
    end if
end if
call sched_nice with updated nice value
```

----------sys/sys/runq.h----------

Function declaration: runq_choose_lotto()

A header for our new lottery choose thread function is added here as this section corresponds to the run queues.

```
struct thread *runq_choose_lotto(struct runq *);
```

runq structure (struct runq)

The field *total_lotto_tix,* was added to the runq structure in order to track the total number of tickets in the runq. This is required in bound checking and random number generation. Additionally, *rand_lotto_winner* field holds the current "winning" lotto ticket for a specific run queue. A specific winner per run queue is required as the number of entries in each run queue can differ. This winning ticket results from the random number generation explained in the *##Random_Number_Generation##* subsection above. Both of these fields are necessary and sufficient to build our lottery scheduler implementation.

```
declare total number of tickets in the run queue
declare current "winning" lotto ticket
```

----------sys/sys/kern_thread.h----------

thread_init()

Since, new threads are initialized in thread_init(), logically we initialized the the number of tickets here. The number of tickets is a new field in the thread structure, so each new thread is initialized to the DEFAULT_LOTTO_TICKETS value (500 tickets).

```
initialize number of tickets to 500
```

----------sys/sys/proc.h----------

 # Define:

We added definitions to our scheduler to help keep track of the acceptable range of tickets, and be able to set and reset our threads and run queues to proper values. Constants were utilized to avoid magic numbers and make for easy replacement if the values were to change (i.e. changing one declaration rather than search of all usages across multiple files/methods).

```
#define MIN_LOTTO_TICKETS 1                 // Min number of tickets = 1
#define MAX_LOTTO_TICKETS 100000            // Max number of tickets = 10000
#define DEFAULT_LOTTO_TICKETS 500           // Default number of tickets = 500
```

```
#define EMPTY_LOTTO_QUEUE 0                    // Empty queue size = 0
```

thread structure (struct thread)

The *num_of_tixs* field was added to keep track of a specific threads number of tickets. This is required to maintain ticket counts for lotto scheduling. As the number of tickets are attributes of threads, the only reasonable location for adding the field was this structure.

```
declare the number of tickets field
```

Testing:

The following programs were tested successfully in our kernel.

fork_bomb - simple program that consumes CPU time calling fork(), looping a long time and printing
Takes a number, *n* from the user and loops *n* times in 10 different child processes, occasionally the program prints: *Child number, process ID, number of iterations left.*

write_bomb - simple program that copies a file using read() and write()
Takes an *input file* and *output file* from the user and reads in the *input file* while writing it onto the *output file*, creating a new file if doesn't already exist

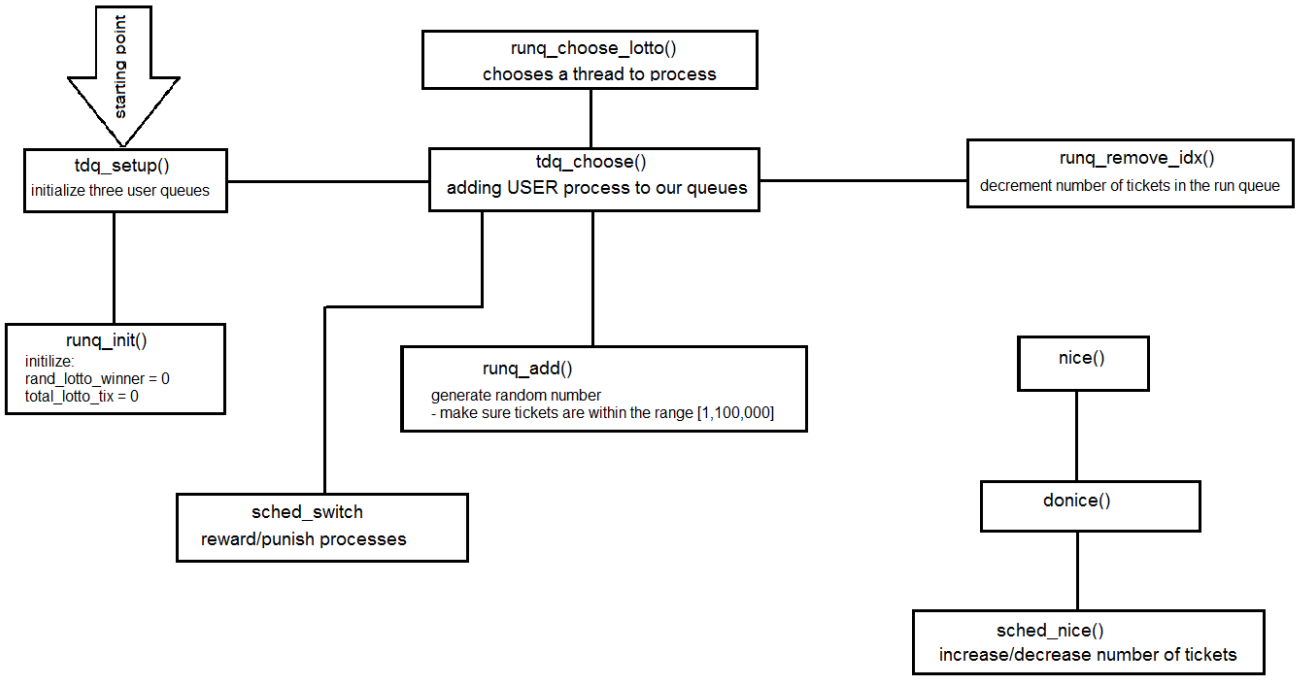
Test Examples:

./fork_bomb 10000000 > in
This created file greater than 300MB called *in* which just contained all of the printed output statements from the different processes.

./write_bomb in out; diff in out
This copied our file, *in* into an identical output file. The *diff command* confirmed the files are identical

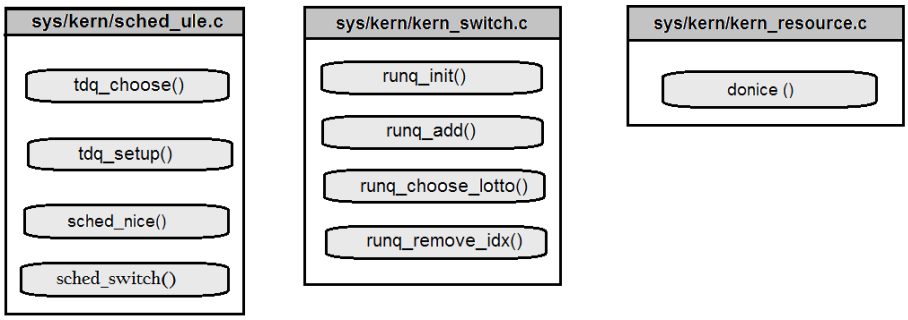
See Next Page For workflow and pictorial overview.

Essential workflow:



Pictorial Overview:

These are the modified implemented .c files. under sys/kern directory.



These are the header files modified.