Students

David Krieger - dakriege@ucsc.edu - 1427712

Design

proc.h

- **struct td** - We add a uint64 called tickets in order to add the field of tickets to threads. We will use tickets for our three lottery queues in order to determine the priority of each thread on there. Although they still have the priority field, we will be using our lottery scheduler to focus on the ticket value.

```
declare uint64_t tickets
```

kern_resource.c

- **donice** - We check if it is user or root. If it is root, do the appropriate checks. If it is not, continue on to normal sched_nice call where the appropriate conditions are checked and met

```
//Check if root or user
if root
    do the normal code checking for priority permisions
if user
    continue //will be handled in sched_nice
```

- **gift** - The gift call will take tickets from the current process and move it to the targeted process. We first check the source and target if they are valid processes (and not root). Then we calculate the number of threads and total tickets for both processes. Make sure we have enough source tickets to give, and that the target process can accept all of them.

  If so, calculate the average amount of tickets each source thread should lose, then the amount of tickets each target thread should gain. However, because these are uint64 decimals will be rounded down and we will lose some tickets. This is covered by 'debt' and 'bonus tickets'. These two values are acquired by doing modulo on the total mod (source or target) threads. We pass the two proccesses source target, average source amount to lose, debt, average target gain, and bonus tickets all to schedule_gift.

  Note**The first thread of either source or target will try and eat the debt or bonus tickets if possible. If they reach their bounds, then set the tickets to the bound amount, and add remaining tickets to either debt or bonus tickets again. Hope the next thread can hand the avg + extra stuff. Cascade until a thread can handle both of them.

  Source - We will actually now allocate the tickets to all the threads. We first start by subtracting the amount of tickets from the source while accounting for debt. If fails, set equal to 1, then increase debt by amount not sucessfully subtracted. If the debt is successfully paid off, we can set it to zero. However, in the future another thread may have a low amount of tickets so it can then add tickets back to debt that other source threads must try and handle.

  Once all the source threads have had their tickets subtracted, it is time to go add tickets to the target process.

  Add the expected average target ticket to the current threads ticket values. If it is over the max boundry, add it to the bonus ticket value and set that thread to the maximum amount. If the next

thread successfully handles the bonus tickets and exepected tickets then zero out the bonus tickets. Note in future, bonus tickets can become non zero again!

Return success! Here is the psuedo code to help out.. alot. See our attatched file to see how the math all works out. The file containts all the expected C code needed to run this. We never implemented it into our kernel though because it always broke it. The pseudo code can help the grader understand it. The actual C code is there to see how to did the math to calculate said tickets to give and take from each thread of desired process.

```
Check if target and source are valid process
    Unable to find process, return failed
If either is a root process
    return failed
else
    Count total tickets in source process
    count # of source threads

    Count total tickets in target process
    Count # of target threads

    if asking for too many tickets
        return # of tickets that can be requested from the source

    //(all threads would have more than 100000)
    else if getting too many tickets
        return # of tickets you can add to target


    //we can safely add and subtract tickets
    else
        calc avg source tickets to take from each thread
        calc the tickets that were snipped from taking an int->unit
        //use modulo to calc that
            Make this the source debt

        calc avg target tickets to add to each thread

        calc the tickets that were snipped from taking an int->uint
        //use modulo to calc that
            Make this bonus tickets


        Pass this to sched_gift

//Imagine this is sched_gift.c

for every thread in source
    subtract tickets by avg source loss + debt

        //thread cant handle that amount of tickets to lose
        If reaches lower bound (1),
            set this threads tickets to 1
            add the remaining tickets it failed to process to debt
```

```
            //Thread successfuly processed the avg source loss
            //as well as potential debt that could be carried
            //over by prior thread
            else

                Set thread to the amount that it  accepted

        for every thread in target
            add tickets by avg target gain + bonus tickets

            //thread cant handle amount of tickets to gain
            If it reaches lower bound (100000)
                set this threads tickets to 100000,
                add remaining tickets it failed to process to bonus tickets

            //Thread successfuly processed the avg target gain
            //as well as potential bonus tickets that could be
            //carried over by prior thread
            else
                Set thread to the amount that it accepted
```

runq.h

- **struct runq** - We added winning (the lottery number) and total ticket count. Lottery number and total ticket count is unique to the runq so we don't have to worry about overlap from the different queues.

```
declare uint64_t total_tickets
declare uint64_t winner
```

- **Headers added** - We added two new headers. runq_lottery_add. runq_lottery_choose. You can read about what each of these functions do when they are called. Located in kern_switch.c.

```
declare runq_lottery_add
declare runq_lottery_choose
```

sched_ule.c

- **struct_tdq** - We added three new run ques of type lottery to the already existing struct of type 'tdq'. It matches exactly how the original code looked.

```
declare all three lottery queues
```

- **tdq_setup** - Here we initialize the three new added lottery ques. Makes use of the call runq_init(). runq_init will set total tickets to zero and set lottery winner to zero. Although the standard run ques will have a total ticket value of zero, it will not matter because we don't redirect their behavior based on tickets.

```
initialize all three lottery ques
//Work is done within runq_init()
```

- **tdq_choose** - Here the runq priority is determined. We mirrored the original code method and place our three new runq's beneath the already given three standard queues. Our placement of which queue gets called first is according to the design requirements: Realtime ¿ timeshare ¿ Idle

```
//Copy original code method
//Add beneath original to ensure propery priority
// Realtime > timeshare > Idle

td = runq_choose(&tdq->tdq_lottery_realtime);
if (td != NULL)
        return (td);

//Repeat x2 for other ques, calling correct name w/ correct order
```

- **tdq_runq_add** - Here we add threads to an existing runq. We first separate them based on if they are user or root. If they are root, they carry out their original behavior w/ the original code. However, if they are user, then they must all go to the new lottery queues. The thread to add to which ever queue is decided by the original priorities, but their ticket values can change. The ticket values will actually determine the likely hood of being chosen, simulating priority level. This keeps the priority: realtime, timeshare, idle.

```
if root
    do Original code
    //will add to one of three standard ques

//must be user, all custom code here
else
        Put into one of the three lottery ques
        Choose appropriate lottery based on original priority
```

- **sched_nice** - We must determine if it is a root or user call. However, a thread was not passed to the function. Instead of adjusting headers, we chose to reuse existing code within this function and break. We call FOREACH THREAD IN PROC and after we get the first thread, we break out of the loop. Once we have said thread, then we can check if the process is root or user. If root, carry out normal behavior. If it is user we must allocate tickets of said process. We chose to add 't' amount of tickets to every single process. This is opposed to adding (t / of threads). We take the appropriate per cautions to ensure ticket output is valid.

```
//Get a thread and save it
//Use thread to determine user vs root

if root
    Normal behavior
else
    for every thread
        add to thread tickets by value t

        if result < 1
            set tickets to 1

        else if result > 100000
```

```
        set tickets to 100000

    else
        set tickets = result
```

kern_switch.c

- **runq_init** - Initialize the the runq for whatever type was passed. This goes for both the standard ques, and the lottery ques. We initial winning lottery number to zero as well as the total number of tickets. This doesn't matter for the standard ques because we never redirect their actions based on tickets.

```
//added
set winner to 0
set total tickets to 0

//keep original
initialize the linked list
```

- **runq_lottery_add()** - A thread is being added to one of the lottery queues. We check to see if it has any tickets, if it doesn't, give it default 500 tickets. We then have to update the total tickets in that queue. Then since a thread has been added, we must generate a new random number. Once this all occurs, add this thread to then the end of the list. We chose to use priority one, because the default is normally zero. This way we can quickly distinguish any user process by being greater than 0 priority. Although we never actually made us of this, we were trying to be aware of it. we can also then user priority as a '1' for bit indication in later on code statements.

```
//estabish our linked list possition as in priority 1
if tickets = 0
    td->tickets = 500;

//Update totale ticket count
rq->total_tickets = rq->total_tickets + td->tickets;

//Udate our random number
rq->winner = (uint64_t)random();
rq->winner = rq->winner;  (rq->total_tickets + 1);

//Set the index for the remove function
td->td_rqindex = priority;

//set the bit to 1 becuase there is something in the linked list
runq_setbit(rq, priority);

//setting the head to be added to
runq_head = &rq->rq_queues[priority];

//add to tail of linked list
TAILQ_INSERT_TAIL(runq_head, td, td_runq);
```

- **runq_lottery_choose()** - Chooses a thread from our lottery chooses. This Function takes in a runq and returns a randomly selected thread. The priority is 1 because why use more than 1 linked list. We get the winner for that runq and use it to select a thread from the q. The total tickets are also stored in the runq. We check for a valid lottery number, and generate one if need be. We then select a thread using the algorithm in the assignment. See below for more details.

```
//the setup
struct rqhead *runq_head;
struct thread *td;

//only use the first q becuase we are only adding them to the first q
int priority = 1;
//get the winner and total ticket of this q from the struct
uint64_t lottery_number = rq->winner;
uint64_t total_tickets = rq->total_tickets;
uint64_t ticket_counter =0;

runq_head = &rq->rq_queues[priority];


//The main selection process
TAILQ_FOREACH(td, runq_head, td_runq)
{
        if (ticket_counter >= lottery_number)
                if(td != NULL)
                {
                        return (td);
                }
        ticket_counter += td->tickets;
}
```