Nikola Panayotov (npanayot@ucsc.edu)
Marko Jerkovic (mjerkovi@ucsc.edu)
Edwin Ramirez (edalrami@ucsc.edu)
Jeremiah Liou (jliou@ucsc.edu)

Assignment 2 Design Document

Abstract:

Three queues were added to the thread struct for user-only processes: user_interactive, user_timeshare, and user_idle. To add a thread to these queues, the process ID of each thread is checked in tdq_runq_add() to determine if it is a user process or a root process. Based on the process ID, the thread is then placed in the appropriate queue. Whenever a thread is initialized, it is given 500 tickets to start, and the process it is running in adds 500 tickets to its total ticket count. The priority of the process would also be calculated using a ratio with the interactivity score, nice value, and an amount of tickets. The number of tickets would then be used to determine the interactivity of a process.

In our design, we assumed that each thread can have between 1 and 100,000 tickets, and that each process does not have a limit on the total number of tickets it can have. The way our scheduler works is it selects a random number using arc4random(), and then goes through each queue, starting with the interactive queue. However, arc4random() returns a uint32_t value, and thus we had to shift the bits to meet the 64 bit requirement of the assignment. We did this by generating two random numbers and ORing them together. Once the 64 bit number is generated, it then goes through the queue, subtracting tickets from each process until it hits 0 or a negative number. Once that happens, the process is chosen and selected to run. Once the process is done running it is removed from the queue and repeated for other user processes. If the interactive queue is empty, then the scheduler will head to the timeshare queue and pick threads there to run. Similarly if the interactive queue and timeshare queue is empty, then the scheduler will go to the idle queue to select processes to run.

Implementation:

These following files were modified to implement nice(), lottery scheduling, and gift. The format is the file in bold and then functions after with descriptions of what was changed.

nice():

sched_ule.c:

struct tdq:

Three queues were added for user processes. They were called user_interactive, user_realtime, and user_idle. These were used to store user processes that originally went into the queues FREEBSD had originally.

sched_priority:

The interactivity score was already calculated beforehand using sched_interact_score and the nice value. Since the score was already calculated before in freebsd, we used a ratio with the score to determine how much tickets each thread would obtain. That way the ratio would correlate with the score and determine how much tickets a thread obtains corresponding to their original interactivity and nice value. A negative nice score is said to mean that a process is more interactive. As a result, if the calculated interactivity score is higher, then the process would receive less tickets, and if the interactivity score was a low number, more tickets were given to the process. A variable was used to determine whether or not the thread was initialized. If it was not initialized, the tickets were set to 500 and the variable is set to false. Otherwise, depending on the score, the ratio would give each thread an amount of tickets. This is also where we kept track of the total tickets in a process. By iterating through the process' threads and adding up the tickets, the total process tickets can be stored.

Lottery Scheduling:

Sched_ule.c:

struct tdq:

Two additional fields were added to the tdq struct: an array of length 100 of uint64_t, and an integer, num_sched. The array of unsigned integers is used to create 100 random numbers (using the methodology with arc4random() described in the abstract), and the num_sched keeps track of every time 100 scheduling decisions are made, in order to generate the 100 random numbers once again.

tdq_runq_add:

After the priority is set using tickets, the threads need to be put in the correct queues. So in tdq struct three user queues were made to put the user processes. User processes needed to go into our queue and root processes needed to be put into the original queues. So the pid of the process was used to determine whether or not it was a user process or a root process. If it was a user process, the process would be put into our user queues. And every time the process was added into the user queue, the total

tickets for that queue would also increase by the original amount of tickets in the process to stay consistent.

tdq_runq_rem:

In tdq_runq_rem tickets were removed from the total tickets in the queue whenever a thread was removed so that the total tickets remained consistent. Once that was done, the thread was removed from the queue.

tdq_rand:

tdq_rand populates the length 100 array of unsigned integers in the tdq struct with 100 random numbers. These numbers are generated using arc4random() to generate an upper and lower 32 bits for the 64 bit random number and are then ORed together.

tdq_choose:

In tdq_choose, the function originally picks the highest priority task and returns it. The original tdq_choose would choose the correct queue to run. So three parts were added to choose the user queues made. If the user_interactive queue is not null, then choose that queue to run processes. If the user_timeshare queue is not null, then choose that queue to run processes. And if the user_idle queue is not null, then choose that queue to run processes. Instead of calling the runq_choose originally, a new function called runq_user_choose was made and run instead. Runq_user_choose will be described later about what is done. Additionally, tdq_choose keeps track of the amount of scheduling decisions made, in order to call tdq_rand every 100 times. This way, the random number to use for the lottery scheduling algorithm is not generated at every context switch, but rather ever 100 context switches.

tdq_setup:

In tdq_setup, the queues needed to be initialized. So runq_init(our queues) was added to initialize our queues. Runq_init will be described later about what is done.

kern_thread.c:

thread_init:

In this function, it is the initialization of the thread. The only thing new in this initialization is that the is_new variable is set to 1 to know whether or not that thread has been initialized before using in the future. This variable was added in the proc.h struct discussed later.

kern_switch.c

runq_init:

In runq_init, the tickets in a queue are initialized to be 0.

runq_user_choose:

This function chooses which processes to run in a user queue. The arguments passed here are the given user run queue to select from, as well as the generated random number. The function will iterates through the 64 queues and start at the head. Then it will check if the head is null, if it is not null, then start iterating from that head and next with a certain amount of tickets. The number of tickets to iterate through is calculated through through the random number mod (%) total amount of tickets in the queue. We would then subtract tickets from this number while iterating through the queue until the number of tickets is 0 or negative. Once it hits 0 or negative, the process is chosen and run.

proc.h:

struct thread:

A few variables were added in this struct. The first variable added was is_new which determines whether or not the thread has been initialized before being used. The other field added was tickets. This field is used to determine how many tickets are in the thread. And the tickets determine the interactivity of the thread.

struct proc:

In this struct a new field called total_tickets was added to keep track of how much tickets there are in a process.

runq.h:

struct runq:

In struct runq, a field called queue_tickets was added to keep track of the total tickets in each queue.

A header function was also added for the new function we created called:
struct thread *runq_user_choose(struct runq *);

Gift():

Note: Our implementation of gift has the arguments in the order gift(t, pid).

In our implementation of gift, we did not evenly distribute tickets to the receiving process. As an example, if the receiving process was given 100 tickets and had only 2 threads, each thread would not receive 50 tickets. Instead, if the first thread has room to accept all 100 tickets, then all of the tickets would be transferred to the first thread only. **Might want to change our gift implementation slightly. If each thread is in a different priority queue, then only one thread will get all the tickets and it will be as if gift was never called on the other threads.

kern_resource.c:

sys_gift:

This is where the system call function was created. A struct gift_args was made with an int and pid. The giving process and receiving process is defined here and passed to the schedule gift function later. The giver would be the current process. The receiver would be found using pfind() to obtain the process with the corresponding pid. Once this is done, this call is over and passed to the schedule gift function.

sched_ule.c

sched_gift:

Sched_gift was implemented for our system call. First, the amount of tickets to be gifted is checked to ensure that the giving process does not give away more tickets than it has, or that the receiving process does not obtain more than the maximum amount of tickets (each thread can have a max of 100,000 in our implementation). After that is determined, we first set the amount of tickets needed to subtract into a variable. Afterward, tickets are subtracted while iterating the threads until running out of tickets. If there are more tickets to gift than there are tickets in a given thread, then the thread's ticket count - 1 is taken, leaving the thread with only 1 ticket remaining. After this is complete, tickets were then given to another process. This time, the amount of tickets can only go to 100,000 in a thread. So if adding the tickets would exceed the maximum value, the highest amount of tickets to reach 100,000 were added to the thread, then iterate to the next thread, and continue the gifting of tickets. This would repeat until all the tickets were given. Once that is complete, then the tickets were successfully transferred from the calling process to the receiving process.