CMPS 111
Winter 2017
Darrell Long **Assignment 2**

Assignment 2 Design Doc -- Lottery Scheduling

Team Members:

Erik Andersen (Team Captain), Michael Cardoza, Yuzhuang Chen, Seongwoo Choi

1. Project Overview:

In the second assignment, we incorporate modifying the FreeBSD scheduler to utilize lottery scheduling for used processes, which gave a good understanding of how FreeBSD could be altered. The current version that we have downloaded on Virtual Machine has the unmodified FreeBSD scheduler, which has 64 run queues. Among these queues, some of them handle root processes, while others are for user processes. Most importantly, root processes have a user ID of zero meanwhile the non-root processes have non-zero userIDs. For this assignment, we created three queues for non-root processes; one for each of the following:

1. Interactive Processes
2. Timeshare Processes
3. Idle Processes.

New run queues will be used for user processings only. The existing 64 queues will be left for the non-user processes as they are now. After creating the three run queues, these new queues should be made accessible to the existing scheduling system and the only code that will be modified will be to implement the desired goal, which is to apply the lottery scheduling only to those three queues.

2. Lottery Scheduler Overview:

The idea of lottery scheduling is that we have randomly assigned each process some tickets and then randomly select a ticket and the process with the ticket selected is allowed to run for set time quantum. The lottery scheduler assigns tickets to processes and then, after a fixed amount of time, it will pick a ticket. Processes that possess more tickets would have a higher probability to be picked. However, the amount of time the processes get per pick is the same for all processes. Even though they all get the same amount of CPU time in every unit, the processes with more tickets will have more total cpu time. Finally students need to alter nice() to cap the number of tickets at 100,000 and to never let tickets go below 1.

In our assignment, the lottery scheduling implementation we are assigning has 500 tickets to each process. In this case, 1 and 100,000 are the minimum and maximum number of tickets respectively. For our scheduler, it will randomly pick tickets ranging from zero to the total number of tickets in the particular queue it is reading such as interactive, timeshare, and idle queues. Therefore, we have to account/track for number of processes added and removed from those queues. During this procedure, we reward processes which are idle for a long period of time that implies they are I/O intensive by multiplying the number of tickets those queues hold by a constant number (by a factor of 2). Moreover, in this procedure, the scheduler will punish CPU intensive processes by a factor of 2. These modifications are based upon on process priority. Most important of all, we are not calculating number of tickets during context switches in order to maintain the expected efficiency by the kernel. For this particular assignment, the nice() system call has been utilized to modify the number of tickets for a process.

3. Project Specifics:

To begin this assignment, we start off by adding and initializing new user queues as per assignment specifications. It is important to add each of our members' account into one FreeBSD virtual machine so that we can keep on track with our modification. The initialization implements the new default values for the lottery scheduler. To implement our lottery scheduler, we gave to divert user processes to our new user queues, away from the standard 64 used by the default scheduler in FreeBSD. By this procedure, each user thread created is initialized with 500 "tickets" and placed into the lottery scheduler queues. Each time a queue adds or removes a thread, it will update its specific "winning ticket" number. When we use a random number generating process, it will update the "winning ticket" number. In this process, when the processor calls the run queue to choose the highest priority thread to run, the lottery scheduler returns a user thread that was chosen by the lottery scheduler algorithm. This means that the scheduler takes a random number, and performs the modulo operator, yielding val. This goes down the list of processes in the queue, adding the number of tickets each one has, until it has reached a number that is larger than val. When it reached that number, it returns that thread and sets that thread as the process to run. The thread is then removed from the run queue. After it has been removed, 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 will be rewarded by increasing the number of tickets if it is I/O intensive and the CPU has a lot of idle time with it.

4. Files that should be modified:

Here are the files we are going to modify:

1. sched_ule.c
2. kern_switch.c
3. kern_resource.c
4. kern_thread.c
5. sysproto.h
6. runq.h
7. Proc.h
8. Syscalls.master
9. Syscall.h

5. Preparation of Building the Kernels:

First, the team captain (Erik) shared his repository with each member in our group and each member copied his repository.

1. Each of our members went to eraander/sys/amd64/conf and create a backup file by typing "cp GENERIC MYKERNEL". This backup file could help us to restore the original kernel in case we break the kernel.
2. Switch to root mood and ready to create our own kernel.
3. Create the kernel by the two following commands. "make buildkernel KERNCONF=MYKERNEL". It takes about 25 mins. After that, we are going to install the kernel by using "make installkernel KERNCONF=MYKERNEL". Which might take 5-10 mins.
4. Reboot the VM
5. Every time when we compile and run the kernel, we can speed up the process by using # make -DKERNFAST buildkernel KERNCONF=MYKERNEL
   # make installkernel KERNCONF=MYKERNEL
   This way, we do not have to wait for a long time to compile the kernel. (Do this on root)

6. Now we can modify our sched_ule.c.

6. Our Goal:

Our goal is to create three user running queue by applying lottery schedule; our lottery schedule should only apply to those three new queues and there should have no negative effects with the 64 existing queues. The main code should goes in sched_ule.c.

When we defined those three user-process-only queues, we should enable them to access the original system and our modification should not affect the original queues. We decide to let those three new runq to share only one index, because this will reduce the amount of work that we need to do when counting the number of tickets. The runq will store the total number of tickets by using an integer. Each process gets 500 tickets when it's created and each thread will have an integer to keep track of its own tickets.

7. How We Modified the Kernel:

The number for each procedure will be described in pseudocode on next part.

1. We need to declare three non-root process queues. Those non-root process queues are interactive processes, timeshare processes and idle processes and their priority are in decreasing order. Those three queues will do the lottery scheduling, and other original queue will do their jobs as before. We also have to modify tdq_print() to add those three queues. Those three new runq are only for user queues, and they will be modified to perform lottery scheduling.
2. We also want to define those three new queues on tdq_setup() function to initiate them before we use.
3. Modify the runq_init() in kern_switch.c function to initialize an integer for keeping track the number of tickets stored in a given queue. We should also add a similar integer to the thread structure.
4. We now need to modified the high level schedule and assign each thread with initial number of tickets 500. This step is really important; we need to modify the tdq_runq_add() in sched_ule.c to check which particular thread should include runq. runq_add() in kern_switch.c to add a thread to a runq. We should be careful that not change the behavior of those original process, only the user processes. We should modify the tdq_runq_add() to add user processes to an appropriate user runq and modify runq_add() to add the thread to one specific index (we will arbitrarily use 0 for all user threads). This means to modify the sched_ule somewhere in the sched_add or tqd_add functions. The most important thing is to separate user process with system process.
5. Now, we need to modify the low-level schedule to pick a random thread (according to their tickets number), which is located in tdq_rem(). This thread should be chosen properly. The priority of user thread is lower than the priority of root process, therefore we should wait until the root processes run. Another thing we should modify is the tdq_choose(), again, we have to distinguish the user process with root process. We can try to run the user queues when root queues have no processes need to run.

   We need to modify another function called tdq_choose(), which allows us to choose the processes in runq because we were using only one queue to store all threads for lottery schedule. We need another function to choose a proper process among those process. This function will choose a process randomly by generating a random number between 0 and the total number of tickets in the runq. (I need to see the code) It will go through the queue and subtract tickets from then number until it reaches 0. When the tickets reach 0, we may be removed it from the queue by other processes.

6. In this assignment, we are using lottery schedule to separate the user-processes from the root-processes. It will use lottery ticket numbers for user processes and default scheduling for root processes. We should modify the nice system call to allow the user to modify the number of tickets that were allocated to each process. If the number of tickets exceeds 100,000, then we reset the number into 100,000. If the number of tickets is below 1, then we reset the number into 1.

7. The next part is the gift() system call. We define

The following pseudo code were written according to each procedure above:

8. Pseudocode:

1. We declare three user runq in **kern/sched_ule.c**, ***line 252***

```
struct tdq{
        ….
struct runq        tdq_realtime;      /* Real-time run queue. */
struct runq        tdq_timeshare;     /* Timeshare run queue. */
struct runq        tdq_idle;          /* Queue of IDLE threads. */
        ….
}
```

2. Initialize their new structures in **kern/sched_ule.c**, ***line 1396***

```
static void tdq_setup(struct tdq *tdq){
        ….
runq_init (&tdq->tdq_user_interactive);     // initialize user_interactive queue
runq_init (&tdq->tdq_user_timeshare);       // initialize user_timeshare queue
runq_init (&tdq->tdq_user_idle);            // initialize user_idle queue
….
}
```

3. This step we are initiating the tickets and letting process to hold those tickets. **In sys/runq.h, *line 63*,**

```
struct runq {
        ….
        int tot_ticket_num;
        ….
};
```

In these lines of code, we initiate the ticket number and use this to track total number of tickets in the entire runq **kern/kern_switch.c, *line 240.***

```
void runq_init(struct runq *rq) {
        int i;
        ….
        rq->tot_ticket_num = 0;
        ….
}
```

In sys/proc.h, *line 290.*

```
Struct thread{
        …
int ticket_num;     //the base ticket number will be 500
        …
}
```

4. We need to specify the user processes and system processes. Then assign them in different runq.
   In **kern/sched_ule.c**, ***lines 473 - 486***

```
tdq_runq_add(struct tdq *tdq, struct thread *td, int flags){
        if (pri < PRI_MIN_BATCH){
                if(process=root process){
            ts->ts_runq=&tdq->tdq_realtime;
                }else{
            tdq->user_tdq_realtime.append(td)
        }
         else if (pri<=PRI_MAX_BATCH) {
                if (process = root access) {
            ts -> ts_runq=&tdq->tdq_timeshare;
                } else {
            tdq->tdq_user_timeshare.append(td);
        ….
        } else {
                if (process = root access) {
            ts -> ts_runq=&tdq->tdq_idle;
        } else {
            tdq->tdq_user_idle.append(td);
        }
```

In kern_switch, *339 line*

```
void runq_add(struct runq *rq, struct thread *td, int flags){
        ….
        if (process =user_process){
                rq->rq_rqueues[63].append(td)
        }

}
```

5. kern/sched_ule.c   line 1354

```
static struct thread * tdq choose(struct tdq *tdq){
 …
 td = runq_choose(&tdq->tdq_realtime);
if (runq_choose fails)
        td = runq_choose(&tdq->user_tdq_realtime);
 if (runq_choose fails)
```

```
                return (td);
 td = runq_choose from(&tdq->tdq_timeshare, tdq->tdq_ridx);
if(runq_choose fails)
               td = runq choose from(&tdq->user_tdq_timeshare,tdq->tdq_ridx);
 if (runq_choose fails) {
…
 return (td);
 }
                td = runq_choose(&tdq->tdq idle);
if(runq_choose fails)
               td = runq_choose(&tdq-> tdq idle);
 if (runq_choose fails) { …
 }
```

6. kern/sched_ule.c line 2030

```
void sched_nice(struct proc *p, int nice){
…
FOREACH_THREAD_IN_PROC(p, td){
        if(td = user process){
               …..
               int new_total = td-> ticket_num + 10*nice;
               if(new_ticket_num > Max_number_tickets){
                              new_ticket_num = Max_number_tickets;
               }else if(new_ticket_num < Min_number_tickets){
                              new_ticket_num = Min_number_tickets;
               }else
                              td->ticket_num = new_ticket_num;
               }
               ….
}
```

7. gift() system call. kern/kern_resource.c line 823-826

```
#ifndef _SYS_SYSPROTO_H_
struct gift_args {
               int t;
               pid_t pid;
}

Int sys_gift{...}
```

kern/syscalls.master line 991

```
548       AUE_NULL       STD       { int gift(int t, pid_t pid); }
```

Modified Files and Methods of Modification:

sys/kern/sched_ule.c

We first explored the source code of FreeBSD inside of its kernel. We first read over *tdq structure(struct tdq)* and we found out that there were three required queues for the lottery scheduling procedure. Those queues were added to the tdq structure. Those queues were the originally unmodified and root queues were added. We had to declare: *user interactive queue, user timeshare queue, user idle lottery queue.*

tdq_choose()

In this queue method, we explored and checked the three queues that were recently created in order of priority and returned a non-null thread from the runq with the highest priority. The most important of all, this order of priority we need to check if those match the design specifications in the assignment.

Algorithm Analysis

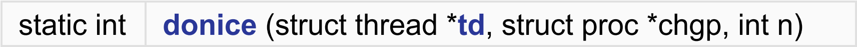
We first need to check user interactive queue and assign return to td_queue. This is the reason why we should check if value of thread is not NULL. If it is not , then we have to return to td. Else, we need to check user timeshare queue and assign return to td_queue. If value of thread is not NULL, then return to thread. Close the if statement and then check the user idle lottery queue and assign return to thread. If the value of thread is not NULL, then return to thread.

tdq_setup()

This method initializes the three user run queues by calling runq_init(). What this means is that this initialization will occur after the root queues are initialized. tdq_setup() was primarily selected to perform this initialization to have a reflection of the root queue functionality. For programming procedure, we need to initialize: *user interactive queue*, *user timeshare queue,* and *user idle lottery queue*

sched_nice()

After bound checking is completed in the donice() method, As the number of tickets assigned was added, the given nice value will increase and decrease correspondingly. Keep in mind that this process is only for user processes and as the default mechanism handles root processes, this method will be only for user processes. By modifying these two functions: donice() and sched_ule(), the nice() system call will be converted to adjust tickets assigned to user processes. Moreover, we should treat every process as though it only has one thread.



```
static int  donice (struct thread *td, struct proc *chgp, int n)
```

Algorithm:

If thread is in a root access, then it will initiate a default behavior, else a thread is a user process, then for each thread in process, adjust the number of tickets by nice value.

sched_switch()

The aim of this part of procedure is to implement the core punish/reward mechanism. The scheduler will be rewarded with I/O intensive processes by multiplying their number of tickets by a factor of 2. After that, we have to leave the CPU idle for a while because that I/O is so much slower than CPU itself. We will initiate the scheduler to punish the CPU intensive processes by dividing their number by a factor of 2. If there exists any CPU intensive, then we will have the scheduler to punish the CPU. The sched_switch is the function that handles threads coming in while blocked in some sort of reason. IF the thread is idle, and if the thread is a user process, then initiate multiplying threads to the number of tickets by factor of 2. If thread is running, and if that thread is a user process, then divide thread to the number of tickets by a factor of 2.

sys/kern/kern_switch.c

runq_choose_user()

This is a new function we created that chooses from the user queues using the lottery scheduler technique. A random number is calculated, the correct tailq is iterated over, and a thread is chosen based on the random number calculated (when the ticket count surpasses the random number generated).

sys/sys/kern/**kern_thread.h**

thread_init()

Dividing number of tickets into a number of threads

We get 500 tickets initially and then we can divide those tickets depended on the number of threads. Basically, we can simply use dividing. The number of tickets is the thread structure and this means that new thread is initialized to the default number of lotto tickets, which is 500 tickets.

sys/sys/**runq.h**

Explanation goes here:

Testing Programs

The following programs were tested successfully for testing our new modified FreeBSD kernel.

read_test - is a simple program that reads in the value of CPU time calling fork(), and this program will loop a long time and print. This is similar to the first assignment where we had to build a simple shell program to take in the value of user input and fork() it. The testing program will take a number that the user set and then it will loop n times in 10 different child processes, and then prints out the values of child number, process ID, and number of iteration left inside the program. In addition to the results of the output, we have created a new function in a read_test file where we can print out nice process in the result. If there are less number of tickets in the process, then the process should run less often.

cpu_test - is a simple program that copies a file using read() and write(). Inspired by the first assignment, the cpu_test program will take an input file and output file from the user and then reads in the input file. While it reads the input file, it will write onto the output file. If there is no existing file on the kernel, then this program will create a new file and store the output, so that the user can see the result clearly. After the program has handled the files mentioned above, it begins to enter a series of for loops. It will print the test variable that is being incremented until the program reaches the last three for loops. The last three for loops, a for loop inside of each other, do not print out to stdout but rather to a file. We have yet to see a completion of this for loop thus giving us a good feeling that it is in fact cpu intensive.

gift_nice_test - is a simple program that tests gift function and nice function. This test takes in the header file that we built for this assignment from syscalls.master and then run the program to see the results of gift and nice functions within the kernel.

Testing Examples:

```
./read_test 100000 > input
./cpu_test input output
./nice_gift_test
```

Extra Notes

When we were approaching this project, we have faced many challenges such as the crash of kernel, saving the progress of the project and restoring the progress, and we also had to face challenge of saving the versions of C codes in the kernel. First of all, every member had to save the current state of the kernel, so that if there was any problem during the progress, then we could go back to the state before the modification. For restoring the FreeBSD kernel, there was a function called ‘snapshot’. By clicking snapshot, and click ‘save’ will store the current state of the kernel. We simply can say that this procedure is a restoration of the kernel. By doing so, we could detect where we had a problem and where had our mistakes. We learned through our mistakes and we improved by communicating with our team members. We constantly talked to each other whenever we had challenges. When one person was working on sched_ule.c, every member gathered around to see the progress. This member, who was in charge of modifying sched_ule.c did not push the changes to the git repository until there was any progress. Moreover, in the kernel, there is a file called, ‘kernel.old’ that showed how we did the works so far.