CMPS 111 — Spring 2017 – ASGN2

Alan Duncan, Aryan Samuel, Khoa Hoang

Description

For this assignment, we are modifying the FreeBSD scheduler to use a lottery scheduling algorithm for user processes rather than the default one provided. Root processes are left to the default mechanisms in FreeBSD. A root process is identified with a value of 0 and non-root processes are identified with a non-zero value.

Files Modified in Assignment:

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

proc.h

For this header file, we are adding a very important variable to keep track of the ticket amount each thread has. Clearly, it is reasonable to add the variable here since this is where the thread structure is located. In addition, we also defined the boundaries for our tickets here so readability of the code is much easier.

Algorithm 1 proc.h

1: Define DEF_TIX to be 500
2: Define MIN_TIX to be 1
3: Define MAX_TIX to be 100,000
4: Declare number of tickets inside thread structure

kern_resource.c

donice()

For this function, we modified it to work for user processes. If a thread is found to be a root process, it is left to the default mechanisms. We first begin by checking the total amount of tickets a process has by iterating through every thread of that process and adding their tickets into a variable. We find the total amount of tickets because this function needs to adjust the nice value of a process based on its tickets. If the processes's total tickets is found to be greater than the bound(MAX_TIX), we set the nice value argument n to be $PRIO\_MIN$ or -20, since a nice value of -20 indicates highest priority. However, if a process's total tickets is found to be less than or equal to the lower bound MIN_TIX, then we set the nice value argument n to be $PRIO\_MAX$ or 20, indicating lowest priority.

If a process's ticket amount does not fall under the extreme cases, n is calculated based on the total amount of tickets divided by 2500 and subtracting this result by 20. This essentially converts the tickets, with range 1 to 100000, to a value that is relatively within the nice value range i.e. -20 to 20.

Algorithm 2 donice()

```
1:  if thread is in a root process then
2:      Use the default mechanisms
3:  else
4:      for each thread in the process do
5:          add thread tickets to total ticket variable
6:      end for
7:      if process total tickets > MAX_TIX then
8:          n = PRIO_MIN
9:      else if process total tickets <= MIN_TIX then
10:         n = PRIO_MAX
11:     else
12:         n = (process total tickets /2500) − 20
13:     end if
14: end if
```

runq.h

In this file we are modifying the header file to include our newly created function runq_lot_choose. Our function was added here since this is a function that modifies our run queues and most run queue functions were found here. In addition, we also added variables to our run queue structure to keep track of total amount of tickets and a variable to determine our winning thread from the lottery scheduler. We also declared our array for our random number generator pool here, declaring it with a size of 64. We determined this would be an appropriate size from our experiments with tracing the frequency of the occurrences between the high-level scheduler and low-level scheduler.

Pseudo-code provided:

Algorithm 3 runq.h

```
Add variables total_tix and rand_lot to run queue structure
DECLARE function struct thread *runq_lot_choose(struct runq *)
DECLARE array for our random number generator
```

sched_ule.c

struct tdq

This structure is modified to accommodate our newly created queues into the code. We found that the default queues for root threads were created here and thus made our user thread run queues here also.

Pseudo-code provided:

Algorithm 4 struct tdq

1: Declare user interactive run queue
2: Declare user time-share run queue
3: Declare user idle run queue

tdq_setup()

This function is modified to initialize the three user run-queues that we have added with the lottery scheduler by utilizing $runq\_init$.

Pseudo-code provided:

Algorithm 5 tdq_setup()

1: Initialize user interactive run queue
2: Initialize user time-share run queue
3: Initialize user idle run queue

tdq_choose()

This function is originally meant to pick the highest priority task, by checking each run queue, we have to do and run it. Here, we added a check for our new user run queues which will adjust the order of the run queues we are checking: standard real-time, standard timeshare, lottery real-time, lottery timeshare, standard idle, and lastly lottery idle. We did our checking in this order based on a TA suggestion. We basically added a check for our run queues, using the standard provided checking as inspiration for our code.

Algorithm 6 tdq_choose()

1: Check standard realtime, perform default mechanism if not empty
2: Check standard timeshare, perform default mechanism if not empty
3: Check lottery realtime, if not empty(NULL), return thread
4: Check lottery timeshare, if not empty(NULL), return thread
5: Check standard idle, perform default mechanism if not empty
6: Check lottery idle, if not empty(NULL), return thread

tdq_runq_add()

Here we are modifying the function to, once again, work for our user processes. We check our thread priorities here and assign it to the proper run queue based on its priority. The thread's assignment into a run queue is based on the default mechanisms provided by FreeBSD (e.g. if the priority of our thread is $< PRI\_MIN\_BATCH$ then the thread is assigned to the real time queue). We are also generating our random number pool here since we do not want to be generating random numbers every context switch. This function (which is in the high level scheduler) calls functions in kern_switch.c (which is the low level scheduler), which is where we need our random pool.

Pseudo-code provided:

Algorithm 7 tdq_runq_add()

```
1: if thread is in a root process then
2:     Perform default mechanisms
3: else
4:     if thread priority < PRI_MIN_BATCH then
5:         Assign thread to real time user queue
6:     else if thread priority <= PRI_MAX_BATCH then
7:         Assign thread to time share user queue
8:     else
9:         Assign thread to idle user queue
10:    end if
11: end if
```

sched_nice()

We first begin our function by doing a boundary check on our threads. Once those initial check are done, we begin adjusting the number of tickets that is assigned to a user process based on the nice value. The nice value conditions are determined and designed by the project team. Threads with a lower nice value are given more tickets over threads with a higher nice value.

Pseudo-code provided:

Algorithm 8 sched_nice()

```
1: if process is a root process then
2:     Use default mechanisms
3: else
4:     for each thread in process do
5:         Change tickets of thread based on nice value
6:     end for
7: end if
```

kern_switch.c

runq_add()

For this function we insert threads into its appropriate queue specified by its priority and then set the corresponding status bit to indicate it is no longer empty(if it was empty). Any further references to "thread" will refer to user-level threads. We begin by finding the amount of tickets the thread we are adding into the queue has and then storing it into a variable(tix_added). This variable keeps track of the amount of tickets we are going to be adding into the total tickets of our run queue. For every thread we are adding into a queue, we perform a boundary check on the thread's tickets to ensure it is within limits of the minimum amount of tickets($MIN\_TIX, 1$) and the maximum amount ($MAX\_TIX, 100,000$). If a thread has 0 tickets, it's ticket amount is set to the default amount ($DEF\_TIX, 500$) that each thread starts with.

After the boundary checks are performed, we can move on to actually adding a thread to the queue. We handle adding to the queues of our run queue similar to how FreeBSD manages root threads: we set the appropriate bit and insert at the head or tail based on the flags. Pseudo-code provided:

Algorithm 9 runq_add()

```
1: if thread is in a root process then
2:     Use the default mechanisms
3: else
4:     add thread tickets to tix_added
5:     if amount of thread's tickets > MAX_TIX then
6:         Lock the thread
7:         Set thread's tickets to MAX_TIX
8:         Unlock the thread
9:         Set tix_added to MAX_TIX
10:    else if amount of thread's tickets <= MIN_TIX then
11:        Lock the thread
12:        Set thread's tickets to MIN_TIX
13:        Unlock the thread
14:        Set tix_added to MIN_TIX
15:    else if amount of thread's tickets 0 = DEF_TIX then
16:        Lock the thread
17:        Set thread's tickets to DEF_TIX
18:        Unlock the thread
19:        Set tix_added to DEF_TIX
20:    end if
21: end if
22: Set priority
23: Lock the thread
24: Set current thread index to priority
25: Unlock the thread
26: Set run bit to indicate run queue is no longer empty
27: Run queue head variable set to queue[priority]
28: Insert thread at the tail
29: Update total tickets in run queue
```

runq_init()

We are simply initializing our run queues here. We set our run queues total tickets and random lottery number to 0. We are also initializing each individual queue inside our run queue through a provided macro function $TAILQ\_INIT$.

Pseudo-code provided:

Algorithm 10 runq_init()

```
1: Set run queue tickets to 0
2: Set run queue random lottery number to 0
3: Use bzero to initialize run queue
4: for each queue in our run queue do
5:     Initialize queue
6: end for
```

runq_lot_choose()

Here we have a function that chooses a process through our lottery scheduler. We begin by creating our function similar to how FreeBSD manages root processes: creating a run queue head pointer and thread pointer then setting the rqh to point to the queue of a run queue based on thread priority / RQ_PPH. However, we then diverge from the default mechanisms and check the run queue's total tickets to make sure it is not empty. We then choose a random number from the random number pool to choose our winner: we iterate through every thread of our process, adding up the number of tickets of each thread until we reach a point where we pass the random number and the thread that made us pass the random number is the winner(thus chosen to run for a set quantum). Otherwise, a NULL is returned if there are no processes inside our user run queues.

Note: if the random pool has been depleted, we regenerate our RGN pool here based on a TA suggestion(it is more efficient option)

Pseudo-code provided:

Algorithm 11 runq_lot_choose()

```
1: Initialize run queue head pointer, rqh
2: Initialize thread pointer, td
3: Set pri based on thread priority/RQ_PPH
4: Set rqh to point to queue[pri]
5: if run queue total tickets != 0 then
6:     Choose random number from random number pool
7:     for each thread in our process do
8:         Add thread tickets into a variable, tix_count
9:         if tix_count > random number then
10:            Return thread as winner
11:        end if
12:    end for
13: end if
       return NULL
```

runq_remove_idx()

For this function, we modified it so it can work on user processes. Similar to how FreeBSD manages root processes, once we remove a thread from a process, the total tickets in the run queue must be decreased. We remove threads from the tail end of the queue and if it was the last process inside the queue, we set the according status bit to signify it is empty.

Pseudo-code provided:

Algorithm 12 runq_remove_idx()

```
1: if thread is in a root process then
2:     Use default mechanisms
3: else
4:     Decrement amount of thread tickets from total tickets
5:     Remove thread from queue
6:     if queue is now empty then
7:         Set status bit to empty
8:     end if
9: end if
```

kern_thread.c

thread_init()

Here we are simply initializing the number of tickets in our threads. Naturally, the default amount of tickets is 500 so every thread will have the default ticket amount of 500.

Algorithm 13 thread_init()

```
1: Initialize thread lottery tickets to DEF_TIX, 500
```

gift(pid, t)

For this section, we are implementing a system call into the FreeBSD operating system. Pid in this case would be the identifier for which process we are gifting t tickets to. The gifted process will be receiving tickets from another process that is invoking the call, possibly a user. However, calling gift(0,0) will return the amount of tickets the invoking process contains. We will first begin by modifying the syscalls.master file so that the OS recognizes our function as a system call. Adding system calls in general may present serious risks to system stability and security, so we must be careful.

We first begin our function by initializing several pointers to keep track of the receiving and ”gifter” process. We then initialize a few more variables(more detail in the pseudo code) to keep track of the amount of tickets we have left to gift. Our gift function works by taking half of a thread’s tickets from each thread in our process until we reach the desired amount to gift. If taking half of the thread’s tickets exceeds our wanted gift amount, we revert the taking and take exactly as much as we need from the current thread. The reason for doing this is to provide fairness among thread distribution. At the end, once we finish taking the amount of tickets from the threads, we begin gifting the tickets we have taken from one process and distributing it evenly across another processes’ threads.

1. Declare system call in sys/kern/syscalls.master

Pseudo-code provided:

Algorithm 14 gift()

Initialize pointers for current and receiving processes, $p\_current$ and $p\_receiver$ respectively
Initialize thread pointer, td
Initialize variable to hold current total tickets and tickets to give, $p\_current\_tot\_tix$ $p\_current\_tot\_tix\_giv$ respectively
Initialize and declare process ID variable, pid
Initialize and declare variable to store current tickets being gifted, t
Set $p\_current$ to td_proc for "gifter" process
Set $p\_receiver$ = $pfind(pid)$
if $pid = 0$ and $t = 0$ **then**
    Print out invoking processes' tickets
else
    Initialize variable to keep track of how many tickets remaining to take away, tix_count
end if
for each thread in our process **do**
    if thread tickets > 1 **then**
        if $tix\_count <$ half of thread tickets **then**
            Take only the amount of tickets we need
        else
            Take half of thread tickets we are taking from
            Decrement tix_count by amount taken away
        end if
    end if
end for
for thread in our process **do**
    Give tickets to pid process, distributed across threads
end for