Team: Evanescence
Captain: Mackenzie Huynh
Member: Anthony Lin

CMPS111 - Prof. Long
May 4, 2019

Assignment 2: Splatter Scheduling

In this assignment, we were asked to modify the existing FreeBSD scheduler which assigns the threads run queues based on priority and uses the FIFO scheduling algorithm. However, we are now asked to implement splatter scheduling -- which assigns the threads a run queue randomly -- and also we are asked to implement a priority queue, which chooses the highest priority thread to run from the run queue.

We are to compare the four cases:

- Case 1 : Base (Default) Case: FIFO, No Splatter
- Case 2 : Priority Queue, No Splatter
- Case 3 : FIFO, Splatter
- Case 4 : Priority Queue, Splatter

And observe the change in scheduling performances between each of the cases.

Implementation

We have four cases which we have to compare, so to switch between the cases we added a system variable in the kernel which we can read from and write to. We used the ***SYSCTL_INT*** to create a integer system variable called ***sched_switch_case*** which will be used to switch between cases. When changing the value of ***sched_switch_case*****,** the values: 1, 2, 3, and 4 is used to denote the scheduling cases which is respective to the four cases we are required to implement.

Now, to implement Splatter Scheduling, we looked around in the kernel code specifically in ***sched_ule.c*** for where threads are being assigned a type of schedule (***ts_runq***) which is determined by its priority. In ***runq_add()*****,** this function is where the thread is actually added into a run queue by its priority. Thus, the functions we modified were ***tdq_runq_add()*** in ***sched_ule.c*** and ***runq_add()*** in ***kern_switch.c***. In ***tdq_runq_add()***, we check which schedule we are currently performing, so if it's 3 or 4, we know we are doing a schedule which requires Splatter. Now, we will determine if the process is a user process by checking if the user id of the process is greater than 0 because Splatter is only for user processes. By knowing if it's a user process, we will assign the type of runq it should be in which is ***tdq_timeshare***, and then find a random run queue for it which will be accomplished by the ***runq_add()*** function. Inside the ***runq_add()***, we made a

check for Splatter by determining if the scheduling is either 3 or 4. Then, check if the process is a user process, and if it is we will assign a random run queue for for this thread by generating a random number by using ***arc4random()***. The number generated by the ***arc4random()*** function will only be accepted if the number falls within the ranges for user scheduling class. Thus, real-time from 48 -79, timesharing from 120-223 and idle from 224-225. Otherwise, if not Splatter scheduling or a user process, we will assign a run queue normally based on it's priority.

Next, to implement priority queue, we found that the ***runq_add()*** function is where the threads are actually being inserted into a runq, so we decided to do the priority queue in that function and use the existing ***TAILQ*** data structure that FreeBSD uses. Now, to do the priority queue, we inserted our priority queue code in the else statement at the end. We did this because the if statement checks to see whether the chosen runq is empty or not, and if it is then we will insert the thread at the head of the runq. Otherwise, the runq isn't empty and has a thread already in it. Now, since the runq is not empty, we need to figure out where the thread we are currently trying to add (***td***) should be place. We traverse through the queue using a while loop and the ***TAILQ_NEXT()***. While traversing the runq, we are going to compare if the (***td***), the thread we are trying to add, has a priority number that is less than one that is found while traversing the runq. If so, we will insert our thread (***td***) before it using ***TAILQ_INSERT_BEFORE()*** because lower priority value indicates higher priority in scheduling. However, if our thread hasn't been inserted into the runq during the traverse, then that means everything in the runq already has a higher priority for scheduling. Thus, we will just insert the thread at the tail of the runq using ***TAILQ_INSERT_TAIL()***. We are able to check if the thread (***td***) was inserted into the runq during the traverse by keeping a variable ***isInserted*** to check. In conclusion, we implemented our priority queue code this way because the scheduler will always pick the first thread in the runq using ***TAILQ_FIRST*** in ***runq_choose()*.** Thus, having the order of threads in the runq based on priority from highest to lowest, ***runq_choose()*** will always pick the first thread which is the highest priority thread in the runq accomplished by our priority queue to run each time.