Goal:

The objective of this project is to modify the current FreeBSD scheduler for user processes from a round robin priority based scheduler, to a lottery scheduler. The goal of this assignment is to understand of how threads are classified and placed into the interactive, timeshare, and idle run queues accordingly, and to gain familiarity with different types of process scheduling algorithms. In addition, this project gave us the opportunity to learn how to modify and build the FreeBSD kernel. We have additionally gained experience debugging using just printf statements and not with breakpoints. Even though this came to be a pain, it was fascinating to learn how hard it is to code in the Kernel space and how even tiny syntax errors could have us sitting for hours without anything to gain.

Design:

The general design of this project involves creating three new queues that take in user processes: interactive, idle, and timeshare. There already exist 64 queues for each of these types of processes and inside these 64 queues are user and root processes. These queues choose which processes to run following a round robin schedule. The first step in implementing a lottery scheduler for the user processes is to separate the root and user processes, but by definition of the assignment all root processes must run before the user processes. To separate the processes and to ensure that root processes run before user ones, we will add an additional rqhead struct called usr_rq_queue to the run queue struct and change the way processes are inserted into these run queues. When inserting into a queue, we will check the user id (UID) of the thread of the process. If this value determines it is a user process, we will insert it directly into usr_queue in the respective process-type array, otherwise we will do nothing and let the scheduler handle the thread normally (because it is a root process). When picking a process to run from this queue, we will use a number from an array of randomly generated integers, and the process with the “winning” ticket will be run. This will be done by using the random() function which generates a 64 bit pseudorandom integer, once we generate this, we take the mod (%) of this number using the total number of tickets in the run queue. When we have this number, we will then iterate over our “user_queue” and keep track of tickets using an integer variable. If the number of tickets a thread has exceeds the random number we have generated, we know we have found our thread that has “won”. Once we have the thread, we decrement the total ticket count from the runqueue, remove the thread from the user_queue, and return the thread. If the thread ticket count doesn’t exceed random, it is clear that we haven’t found our thread and we keep iterating.

New/Modified Members:

Design/Modification of Each Function:

In runq.h

```
#define RQ_NQS          (64)            /* Number of run queues. */
#define RQ_PPQ          (4)             /* Priorities per queue. */
#define POOL_SIZE       (1000)             /* size of randpool array */

struct runq {
       struct  rqbits rq_status;
      struct  rqhead rq_queues[RQ_NQS];
       struct  rqhead user_queue;
      int     total_tix;
       int randPool[1000];
       int counter;
};
```

In proc.h

```
struct thread {
/*
/ Add a td_tickets variable that is of type integer
*/
};
```

In kern_thread.c

```
 static int thread_init(void *mem, int size, int flags)
 {
         struct thread *td;

         td = (struct thread *)mem;

         td->td_sleepqueue = sleepq_alloc();
         td->td_turnstile = turnstile_alloc();
         td->td_rlqe = NULL;
         EVENTHANDLER_INVOKE(thread_init, td);
         td->td_sched = (struct td_sched *)&td[1];
         umtx_thread_init(td);
         td->td_kstack = 0;
         td->td_sel = NULL;
           td->td_tickets = 2000;
         return (0);
 }
```

In kern_switch.c

```
/* Custom function to select process via tickets */

choose_utd (struct runq *rq){
       int currTix = 0;
       int randomNum = rq->randPool[rq->counter];
       counter++;
       struct rqh *rq;
       rqh = rq->user_queue;
       ITERATE OVER RUNQ Head
              currTix += td->td_tickets
              if currTix > randomNum
                     rq->total_tix -= td->td_tickets
                     remove td
                     return td
```

```
runq_choose(struct runq *rq) // used for realtime & idle
       while queue array index pri is valid
              Run runq_choose as currently implemented // root process
              return thread which is chosen to run
       end while
       if(rq->total_tix != 0)
         td = choose_utd;
         if(td) return td;
       }
       return NULL // no thread to run

runq_choose_from(struct runq *rq, u_char idx) // used for timeshare
       if run queue index is valid
              Run runq_choose_from as currently implemented // root process
              return thread which is chosen to run
       else
              Return thread returned by call to choose_utd

/* Called by tdq_runq_add after selecting what kind of process a thread is */

runq_add(struct runq *rq, struct thread *td, int flags)
       if uid != 0 // this is a user process
              rq->total_tix += td->td_tickets
              Place in rq->user_queue
              return;
       else
              Run runq_add as currently implemented

runq_add_pri(struct runq *rq, struct thread *td, int flags)
       if uid != 0 // this is a user process
              rq->total_tix += td->td_tickets
              Place in rq->user_queue
              return;
       else
              Run runq_add as currently implemented
```

```
void
runq_remove_idx(struct runq *rq, struct thread *td, u_char *idx)
{
       struct rqhead *rqh;
       u_char pri;

       if(td->td_proc->p_ucred->cr_uid != 0){
              rqh = &rq->user_queue;
              rq->total_tix -= td->td_tickets;
              TAILQ_REMOVE(rqh, td, td_runq);
              return;
       }
       Else run runq_remove_idx as currently implemented
}


void
runq_init(struct runq *rq)
```

```
{
        int i;
        rq->counter = 0;
       rq->total_tix = 0;
       TAILQ_INIT(&rq->user_queue);
       bzero(rq, sizeof *rq);
       for (i = 0; i < RQ_NQS; i++)
               TAILQ_INIT(&rq->rq_queues[i]);
}
```

In sched_ule.c

```
sched_nice(struct proc *p, int nice)
{
       struct thread *td;

       PROC_LOCK_ASSERT(p, MA_OWNED);

       p->p_nice = nice;
       FOREACH_THREAD_IN_PROC(p, td) {
              thread_lock(td);
              if td->td_tickets += nice > 100000
                     td->td_tickets = 100000
              else if td->tickets += nice < 1
                     td->td_tickets = 1
              else
                     td->td_tickets+=nice
              sched_priority(td);
              sched_prio(td, td->td_base_user_pri);
              thread_unlock(td);
       }
}

// Implementing random number buffer pool
sched_clock (struct thread *td) {
  struct tdq *tdq;
  tdq = TDQ_SELF();

       tdq->counter's = 0;
  for (int i = 0; i < 1000; i++) {
       Int rand = random();
       tdq->tdq_realtime->randPool[i] = rand;
       tdq->tdq_timeshare->randPool[i] = rand;
       tdq->tdq_idle->randPool[i] = rand;
  }
}
```

Testing

In order to test if our program actually works as it is supposed to, we can execute the longrun program and check how many times one process gets picked in comparison to another. If we were to give a process, say process A, five times more tickets than process B by using the nice function, then process A should be selected 5 times more often than process B. This can be seen when running longrun.c, if the values of process A are incrementing at a five times faster rate than process B. Other methods that we have used for testing and debugging purposes are printf statements that we literally put everywhere to find the locations as to where the bugs are located. Eventually, through much debugging, we were finally able to complete this program.