Assignment 2

Changing FreeBSD's scheduler
by implementing lottery scheduling

Roman Sodermans rsoderma
Georgios gekaragi
Michael Tang miytang
Calvin Yang cahyang

Altered files:

sys/kern

- sched_ule.c
- kern_swich.c
- kern_thread.c
- kern_resource.c
- syscalls.master

sys/sys

- proc.h
- runq.h

Changes made in files

Changes in sys/kern/sched_ule.c

- Added macros to reward/punish tickets in sched_tickets
- Added three runq structs in tdq struct
- Added sched_tickets function
  - Takes thread and int as an argument.
  - Rewards or punishes the threads tickets based on interactivity score
- Changes in tdq_runq_add,
  - check for user or root process , puts into our own runq for user processes
  - Puts the thread into the correct runq based on interactivity score
- Changes in tdq_choose
  - If original 3 runqs are empty, we will choose our own runqs
- Changes in tdq_setup
  - Added code to initialize our own runqs
- Added function sched_tickets
  - Function we added to reward or punish threads number of tickets
- Changes in sched_nice
  - Add or subtract tickets based on nice value

Changes in sys/kern/kern_switch.c

- Changes in runq_init

  Initialized runq total_tickets = 0
- Added function lottery_choose (struct runq *rq)

  Chooses the next thread to run using our lottery scheduling policy

Changes in sys/kern/kern_thread.c

- Changes in thread_init

  Initialized threads tickets = 500

  Initialized interactivity score = -1

Changes in sys/kern/kern_resource.c

- Added system call function sys_gift

Changes in syscalls.master

- Initialized system call for function int gift (int tickets, pid_t pid)

Changes in proc.h

- Changes in struct thread

  Added int tickets, int user_thread, int cur_IS

  Tickets = threads tickets it holds

  User_thread = flag to check if user_thread

  cur_IS = current interactivity score to use in sched_tickets

Changes in runq.h

- Changes in runq struct

  Added int total_tickets

  Total_tickets = total tickets of all the threads in the runq
- Declared struct thread *lottery_choose (struct runq)

  Function is used in kern_switch.c to choose a thread using our lottery scheduling policy

New functions implemented:

- **sched_tickets(thread *td, int score)**
  - *Arguments*:
    - a thread pointer
    - Interactivity score of thread
  - *Return Type*: void
  - *Description*: Punishes or reward processes with tickets according to their interactivity score
  - *Assumptions*: the thread pointer is not null
- **lottery_choose(struct runq *rq)**

- *Arguments*:
    - A runq pointer that we want to choose from (interactive/timeshare/idle)
- *Return Type*: Thread pointer
- *Description*: Chooses the next thread to run next from runq using a lottery scheduling algorithm
- *Assumptions*: the runq pointer is not null

- **sys_gift(int tickets, pid_t pid)**
    - *Arguments*:
        - Tickets we want to gift
        - Process id of the process we want to gift tickets to.
    - *Return Type*:
        - Integer
    - *Description*:
        - We give a process given by the pid tickets given by the first argument by taking away tickets from the process we called the system call from.
    - *Assumptions*:
        - If we try to give more tickets than the tickets the process has we return the number of tickets the process has.
        - If we try to gift 0 tickets we return the maximum number of tickets we can gift.
        - If the number of tickets we try to give will make the threads of the receiving process exceed 100,000, we return with an error message.

Lottery Scheduler Policy:

Iterate over all threads in the runq before lottery to get total ticket count. The benefit of this over keeping a field up to date every time a process is added or removed is that, for example, when we added our new system call gift we could do so without worrying about updating the total tickets. When interactivity score is calculated, reward or penalize ticket count based on its new interactivity score

Our policy for rewarding and penalizing tickets is based on relative penalties and non-relative rewards. What this does is subtract more tickets from a thread with many tickets, and the flat addition of tickets is worth more relatively to processes with a lower number of tickets. There is an equilibrium point where the amount of tickets subtracted is equal to the number of tickets added. We want this equilibrium to be above the ticket value of a new process (which is 500) so that new processes are not favored unfairly over existing processes. The exact value of tickets for the equilibrium would need far more extensive testing, but after testing 500 (add 50, subtract 10%) and 1000 (add 100, subtract 10%) we decided on 10,000 (add 1000, subtract 10%). We created the macros <> to modify these if desired.

Nice Policy:

Nice policy was implemented in sys/kern/sched_ule.c in the function sched_nice. A nice integer can have a range from -20 through 20. A negative nice value will give tickets to each thread in the process. A positive nice value will take away tickets from each thread in the process. We calculate how much tickets to give or take away using the equation
"tickets += - (nice *5,000)". So a nice value of "-20" for example will give each thread in the process 100,000 tickets. A nice value of "20" will take away 100,000 tickets from each thread in the process. And a nice value of 0 will not change the number of tickets. And a nice value of "-10" will give 50,000 tickets to each thread in the process. The equation that we use to calculate how much tickets to give or take away is scaled linearly and gives the user very specific controls over the amount of tickets to give or take away from every thread in a process. Our nice function also makes sure that a thread's amount of tickets will always be in the range of 1~100,000.

Gift Algorithm:

Gift Usage: syscall (548, int tickets, pid_t pid)
548 = syscall ID
tickets = tickets we want to give
Pid = pid of process we want to gift too

Check if gift process has enough tickets among its threads. If so, we can safely proceed. Check if receiving process has enough capacity among its threads. If so, we can proceed. The algorithms can run with the knowledge that there exists a solution because of these checks. The number of tickets specified as an argument reflect the total number of tickets to be collected by the gifting process for the receiving process. We chose this because the user can check the number of tickets available in the gift process and be able to use every single ticket. Similarly, the user can check the number of tickets available in the receiving process and be able to give it every ticket that is actually available. We wanted our algorithm to give consistent results no matter the distribution of tickets amongst any number of threads.

- **To Gift:**
  - Gifting process takes relative amounts of tickets from each of its threads to total the specified number of tickets to gift. Gifting thread gives half its tickets. If all threads have given and not enough tickets have been given, iterate again until enough tickets given. We chose the gifting policy because it subtracts relative to a threads tickets on each iteration, which punishes threads with larger ticket values more (this falls in line with our lottery policy).
- **To Receive:**
  - Receiving process will attempt to spread tickets evenly across all of its threads. Receiving thread receive an equal share of tickets that have been gifted. If a thread hit max tickets, we will have a remainder. Iterate over threads again until all tickets allocated. We chose the receiving policy because it adds equal values to each threads tickets on each iteration, this rewards threads with smaller ticket values more than those with larger ticket values (this falls in line with our lottery policy).