Team:
William Lynch
Karina Ruzinov
Jonathan Maltz
Artem Titulmenko

Areas of change:

The primary area where we made changes in this project was sched.c, with a focus almost entirely on the schedule() method itself. Additionally, our changes required modification to both the user_struct and task_struct structures, both of which were made in sched.h. The other changes we made were minor book-keeping and initialization changes within user.c, fork.c, and the necessary files for the system call.

Implementation details:

Our implementation is based off of two fields that we added to the task_struct: current_weight and max_weight and one field that we added to user_struct: hasRun. The algorithm then proceeds as follows.

First, we scan through the list of tasks on the runqueue, checking each task to see if the user which owns the task has run a process this round, if not, the user will have a hasRun value of 0. This is the next user to run. Once found, we look through all tasks again, searching for any tasks with a current_weight > 0 and that belong to the next user to run. If a process is found matching this criteria, it becomes the next process that will run. If no process is found matching this criteria, then we iterate over all tasks again and set the current weight of all tasks which belong to the next user to run equal to their max_weight. We then run through the list one last time to select the proper task to run.

If no user is found that has not run this round, then we iterate over the list of all tasks, and set the hasRun value of all users to 0. We then repeat the previous process to find the next task to run.

Once we have found a task, we decrement its current_weight value and set the hasRun value of its user to 1. The scheduler then proceeds as it did before our patch.

To implement the system call, we simply multiply the max_weight of the current task by 2. Thus, the next time that specific task has its weight reset, the task will be selected twice as many times before it is disqualified for having a current_weight of zero.

Overhead:

The two fields we added to the task_struct are integers, each of which is 4 bytes, so we added 8 bytes to the task_struct. The hasRun value is an char, so that adds one byte per user. Thus, the additional space is O(T + U), however, since U is bounded by T in all cases, O(T) space is added.

In terms of our runtime, in the worst case our algorithm will scan over the list of tasks five times. First, it will try to find a user to run, and it will fail. Next, it will iterate over all tasks to reinitialize hasRun fields for all users. Next, another iteration over the runqueue is needed to select a task with a current_weight greater than zero. Then, an iteration would be needed to reinitialize all task values, and finally, one iteration would be needed to find the correct task. Thus, the worst case to find a task is 5T or O(T) time, where T is the total number of tasks. Since our algorithm looks over only the runqueue to find the next task to run, this worst case will only be achieved when all tasks are on the runqueue, which is highly unlikely.

Further improvements:

Most of the shortcomings of this algorithm come from the number of times we perform a naive scan through the list of tasks. We could optimize this significantly by keeping a queue of users and giving the users knowledge of the tasks which they own. With that information, we could find the next user to run in O(1) time, find the next process to run in O(1) time, and schedule tasks such that reinitialization of their weight value can be done in O(1) time, thus making the entire scheduling process O(1). The difficulty in this method comes from keeping track of the states of tasks as they move between queues.