COMP 530H
Spring 2017

Programming Assignment 5
Module for user-weighted round-robin scheduling
Date assigned: October 13, 2017
Date due: November 3, 2017 (to allow for fall break)

Implement a module to provide user-weighted round-robin (UWRR) scheduling of processes. The module will essentially extend the Linux Real-Time scheduling class for processes that use the **SCHED_RR** policies. Each process should be given an associated weight that determines its ***proportional share*** allocation of CPU time allocated in units of its weighted time slice (quantum). For this assignment, assume that all processes calling your module are CPU-bound.

Relative weights of processes must range between 1 and 20, inclusively. The actual time slice for a process is determined by multiplying the weight by 10 milliseconds. Thus a weight of 10 corresponds to the normal Linux time slice of 100 milliseconds. As an example, suppose that process A has a weight of 5 and process B has a weight of 20, both with the same priority. Then, on average process B should receive four times as much CPU time as process A over a fixed interval of time. Note that in the original Linux implementation, all **SCHED_RR** processes have the same default fixed time slice so all processes with the same priority in this class receive approximately equal amounts of CPU time over a fixed interval of time. Your module will alter this equal sharing to provide proportional sharing.

The input to your module from a calling program is a character string with the following syntax:

```
  uwrr<space><integer>
```

where **`<integer>`** is the character representation of the integer value that is the relative weight for this process. The process making a call to your module must be using the **SCHED_RR** policy and have real time priority of **1**. It is an error if both of these are not true. The procedure for running a process with these attributes is explained below. For this module, there is no return string but the module should return **-1** to the **debugfs write()** call if there is any error and **0** when the call succeeds without error. Since there is no return string, your module should treat a **debugfs read()** call as a NOOP (return **0** always).

Your extension of the Linux Real-Time scheduling class can be accomplished by making a function pointer substitution for one function pointer contained in the **sched_class** "object" pointed to by the **task_struct** of the calling process. The necessary steps are:

- ***On the first valid call to your module***, initialize a global memory copy of a structure with type **sched_class**. This structure will be shared by all processes that call your module. Initialize the structure by copying (with **memcpy()**) the **sched_class** structure referenced by the first calling process (**current->sched_class**) to your local copy. Replace the function pointer in the structure element **task_tick** with a pointer to your local function that implements the weighted scheduling. Be sure to save a pointer to the original function for use in your substitute function. The **task_tick** function (and your function substituted for it) has the prototype declaration of: **static void task_tick(void *rq, struct task_struct *p, int queued**);
- On each valid call to your module from a process (including the first one), replace the pointer to **sched_class** in the calling **task_struct** with a pointer to your local **sched_class** structure.

A fragment of C code, **sched_example**, giving examples of these steps is provided in **/home/smithfd/530H/f17/source.**

Be sure to consider how you will store and access the weights for a number of processes (fix the maximum number of such processes at 25). Once your **sched_class** structure with your new function pointer for **task_tick** has been initialized, your function will be called on each Linux kernel timer tick (default is every millisecond). The C declaration for struct **sched_class** is not part of any Linux include file. A substitute file, **SchedCl.h**, that can be included by your module is provided in **/home/smithfd/530H/f17/source.**

Study the kernel implementation of the function **task_tick_rt()** in module **kernel/sched/rt.c** to determine how your function can control the time slice allocations to implement user weighted round robin scheduling.. ***Hint: don't try to make this harder than it needs to be!. You should call the original task_tick function as part of your module, but consider what steps you need to take before and after calling the original function***.

.

This assignment has a significant experimental component to evaluate the results of assigning different weights to a set of processes in the **SCHED_RR** class. Use the **printk** function to log useful data in **/var/log/kern.log** for each user-weighted process – for example, log a timestamp when a time slice begins and ends. The kernel function **ktime_get()** can be used to obtain a microsecond precision timestamp like this:

```
    u64 time_stamp;
     time_stamp = (u64) ktime_to_us(ktime_get());
```

(note, the **u64** type requires a **%llu** specification in a format string).

To conduct your experiments you will need to run several CPU-bound processes concurrently (with different weights) that complete some number of iterations in a fixed interval of time. A C code fragment, **spinner.c**, that can be incorporated into your experiment program is also in **/home/smithfd/530H/f17/source** (note: when compiling this code add the switch **–lcrypt** to **gcc**). Synchronizing the start of multiple processes can be accomplished by also inserting your barrier synchronization module from assignment 3 into the kernel. The basic structure of your experimental spinner program then would be:

- Call **uwrr** in the kernel module to set its scheduling weight
- Call barrier synchronization module to wait ***non-exclusively*** on a known event ID
- When event is signaled, begin the CPU-bound iterations.

Of course you will need a caller program for your barrier synchronization module to create an event and to signal that event once all processes in the experiment are waiting.

To run your experimental program with the **SCHED_RR** policy and priority **1**, use the **chrt** command (written by the Robert Love who is the author of our reference text). It has a man page and here is an example: **sudo chrt –r 1 ./spinner &**
Warning: these CPU-bound iterations have a higher scheduling priority than other normal processes including a shell. Your shell(s) and the console will be totally unresponsive once the spinners start until they complete (you should debug your experimental program before running it with **chrt** to be sure it does terminate normally).

Note on running experiments – by default Linux limits process with real-time priorities 1 – 99 to 95% aggregate CPU utilization in any one second interval (to avoid completely starving other processes). If this 95% limit is exceeded, the real-time processes are "throttled" (have limited CPU allocations in future time intervals). You will definitely encounter this throttling when running CPU-bound processes, so throttling should be turned off after booting your VM by creating a root shell with **% sudo -i**: and run

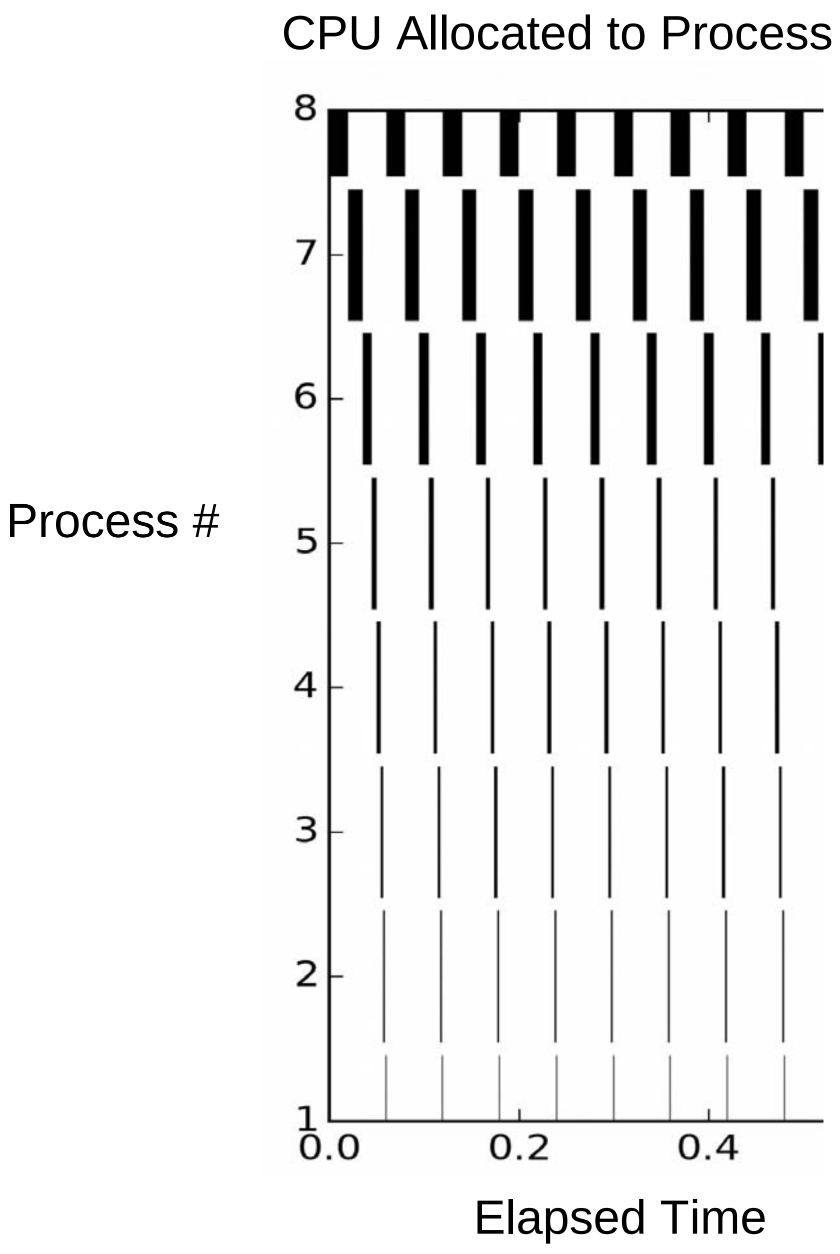
```
  % echo -1 >/proc/sys/kernel/sched_rt_runtime_us
```

Write a short report (3-4 pages maximum) that discusses the data, observations, and conclusions from your experiments. You should present your data and results in some combination of tables and plots. Be sure to consider such issues as how the number of iterations achieved in CPU-bound processes relates to their specified weights and how accurately the kernel allocates variable CPU time slices. What happens if the CPU-bound processes are written to return after a (large) **fixed number of iterations** instead of a fixed interval of time?

Extra credit: Use your favorite plotting tool with your logged data to produce a plot of CPU execution times for each weighted process over experimental runs using the one given below for inspiration.

Submitting your program:

Follow the instructions from assignment 4 for submitting your programs and report.


CPU Allocated to Process
Process #
Elapsed Time