Overview:

The basic design for my latency profiler keeps track of the latency of a given PID. In addition, for each PID stored, a list of all of its corresponding call traces and their latencies is stored as well. This is implemented through several mechanisms.

Data storage:

The main structure for storing the PIDs, as well as their process name and overall latency, is called a taskNode. These taskNodes are organized in two red-black trees, to optimize search times. The first RB tree uses the PID as a key, and whenever a task is deactivated or activated, the first step is to search for it using this RB tree. Because there is only ever 1 node in the tree with a given PID, this search is optimal for using the PID as key.

The second RB tree uses the latency of the taskNode as a key. This is optimal for identifying the nodes with the greatest latency, as they will all be on the right hand side of the tree, in descending order as you move more towards the left. This makes the rb_prev() function call ideal for iterating the tree to find the nodes with maximum values. Because of the potential issues related to nodes with the same key in an rb tree, I also added an offset value to the nodes. In the case where trying to add a node with the same latency, the offset values are compared. If they are the same, the offset is incremented, and the node must be re-compared to all other nodes in the tree, to prevent issues with unbalancing. When a node is updated, its offset value is always reset to zero.

To determine and store the latency of a given taskNode, two values are used, sleep_time and start_sleep. start_sleep is set to -1 when a task is awake, and is set to the value of rdtsc when it goes to sleep. Upon waking, the difference between rdtsc at that moment and start_sleep is added to the sleep_time, and start_sleep is set to -1 again.

Another set of important data that I had to keep track of was the various stack traces for a given PID. A list of these was maintained with a set of hash-tables. Each taskNode has its own hash-table of call traces and their corresponding sleep times, which are updated or added to whenever a task is woken up, as is appropriate. Only call-trace with the maximum latency will be printed for a given PID.

Kprobe:

The Kprobe hooks were inserted at the activate_task and deactivate_task function calls in <include/linux/sched/core.c>. These functions were chosen because they are the base functions for the scheduler subsystem that call implementation specific functions, such as __enqueue_task and __dequeue_task in the CFS. In both calls, a search of the RB-tree keyed to PID is done.

After the search is completed, 1 of several things can happen:

1. If a taskNode with a corresponding PID is found, it is updated. If this update includes an update to latency, it is removed from the latency based RB tree, and re-added
2. If no matching taskNode is found, and this is a deactivate_task call, a new node is created, and added to both trees
3. If no matching taskNode is found, and this is a activate_task call, the kprobe returns, and nothing is done

Data safety (locks):

In order to protect data in the various trees from corruption due to multiple tasks accessing, I implemented a rw_lock, rb_lock. The reader lock is used when printing out the highest latency tasks, although the printing uses the read_lock_irqsave method, to prevent interruption from the task scheduler and a deadlock. When writing to or updating the trees the write lock is used. The write lock function must encapsulate both the search and the adding to the tree; this is done to ensure the tree has not changed since it was read, as this could cause major issues. No lock is needed for the hash-tables, as all hash-table changes take place inside of the rb_lock.

Output (proc fs):

The proc fs implementation is very straightforward. On module insertion, it creates a file, /proc/lattop, and on removal, it deletes it. If the file is opened, the fs calls a printer function for the RB-tree keyed to latency, which first locks the tree as a reader, and then iterates through the highest latency nodes, until it has printed 1000 nodes, or encounters a node with 0 latency, or in other words has been put to sleep, but has not yet been woken up, at which point it unlocks the tree, and returns.